TP 4 : Tables de Hachage, première implémentation d'une structure de dictionnaire

On rappelle qu'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 de recherche d'un couple (clé, élément) à partir de la clé, d'ajout d'un couple (clé, élément) ou de suppression d'un couple à partir de sa clé. Une réalisation de la structure de dictionnaire se fait par exemple à l'aide *d'arbres binaires de recherche* (qu'on verra plus tard). On va voir ici une autre réalisation, au moyen de *tables de hachage*. L'idée est la suivante : pour chaque clé 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 tels que $h_w(k) = i$. Dans ce TP, on n'accordera pas beacoup d'importance à l'élément e pour se concentrer sur la clé k, mais dans un but applicatif, on s'en servirait : par exemple pour un dictionnaire au sens usuel, les clés k sont les mots admis, les éléments e sont leurs définitions.

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 clés dans les différentes cases de la table, c'est-à-dire, de manière informelle, qu'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.

Entiers naturels. Pour le cas où les clés sont des entiers, le *hachage par division* de largeur w consiste simplement à hacher la clé 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. Écrire une fonction `hachage_entier` de signature `int -> int -> int` telle que `hachage_entier w k` retourne le haché de la clé entière k en utilisant un hachage par division de largeur w.

```
# hachage_entier 14 201;;
- : int = 5
```

Chaînes de caractères. Dans le cas où les clés 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 127 identifiant le caractère de manière unique). Une chaîne $s = s_0 \dots s_{n-1}$ peut alors être vue comme la représentation en base 128 de l'entier

$$\sum_{k=0}^{n-1} \text{code}(s_k) \times 128^k$$

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

Question 2. Écrire une fonction `hachage_chaine` de signature `int -> string -> int`, de hachage par division des chaînes de caractères. *Attention à prendre le modulo régulièrement pour éviter le dépassement de capacité sur les entiers naturels de Caml !* (ne pas évaluer la somme précédente pour prendre le modulo « à la fin »). On rappelle que si `s` est une chaîne, `String.length s` est sa longueur et `s.[i]` son i-ème caractère (l'indexation démarre à 0).

```
# hachage_chaine 12 "Bonjour !" ;;
- : int = 2
# hachage_chaine 12 "oh oui youpi dansons la carioca";;
- : int = 11
```

Remarque : n'utiliser que des caractères ASCII dans vos tests : pas d'accents !

2 Tables de hachage de taille fixe

On représente en Caml une table de hachage de clés de type `'a` avec des données de type `'b` par un enregistrement du type suivant :

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

Soit `t` une table de hachage de largeur w. `t.hache` est la fonction de hachage utilisée pour hacher les clés stockées dans la table (notée dans l'énoncé h_w) et `t.donnees` est un tableau de longueur w. La case numéro i de ce tableau contient la liste des entrées (k, e) de la table telles que $h_w(k) = i$. Les listes ne sont pas supposées ordonnées.

Question 3. Écrire une fonction `creer_table` de signature `('a -> int) -> int -> ('a, 'b) table_hachage` telle que `creer_table h w` retourne une nouvelle table de hachage vide de largeur w utilisant la fonction de hachage h.

```
# creer_table (hachage_entier 5) 5 ;;
- : (int, '_a) table_hachage = {hache = <fun>; donnees = [|[]; []; []; []; []|]}
```

(Remarque : `hachage_entier 5` est une fonction car `hachage_entier` est une fonction curryfiée à deux arguments).

On va maintenant écrire des fonctions effectuant les opérations de dictionnaire sur ces tables : recherche, ajout et suppression. Attention : pour que la structure de données soit efficace, il faudra pour chacune de ces fonctions, ne visiter que « l'alvéole » (la liste `t.donnes.(i)`) adéquate. Dans la suite, on prend comme exemple le `petit_exemple` suivant :

```
let petit_exemple = {
   hache = hachage_entier 3;
   donnees =
   [|[(15, "truc"); (468, "ocaml"); (498, "confinement"); (144, "TP")];
   [(1, "machin"); (154, "coucou")]; [(185, "info"); (512, "MPSI")]|]
} ;;
```

C'est une table de hachage stockant des couples (entier, chaîne), via la fonction `hachage_entier 3`. Vous pouvez voir que les nombres de la première liste sont divisibles par 3, ceux de la seconde congru à 1 modulo 3, ceux de la dernière à 2 modulo 3.

Question 4. Écrire une fonction `recherche t k` de signature `('a, 'b) table_hachage -> 'a -> bool` retournant un booléen indiquant si la clé k est présente dans la table t.
Remarque : vous ne devez en aucun cas essayer de farfouiller dans *toutes* les listes de la table ! L'intérêt du hachage est que l'on peut se contenter de chercher dans la liste d'indice i tel que $h_\omega(k) = i$, avec h_ω la fonction de hachage.
Rappel : une liste se parcourt en récursif. La liste à parcourir est constituée de couples, et il faut tester si la première composante est k.

```
# recherche petit_exemple 498 ;; (* 498 = 0 mod 3, ne visiter que la première liste *)
- : bool = true
# recherche petit_exemple 499 ;; (* 499 = 1 mod 3, visiter la deuxième *)
- : bool = false
```

Question 5. Écrire une fonction `element t k` de signature `('a, 'b) table_hachage -> 'a -> 'b` retournant l'élément e associé à la clé k dans la table t. Si la clé n'est pas présente dans la table `t`, votre fonction lèvera l'exception `Not_found` (avec `raise Not_found`).

```
# element petit_exemple 498 ;;
- : string = "confinement"
# element petit_exemple 499 ;;
Exception: Not_found.
```

Question 6. 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 déja présente.

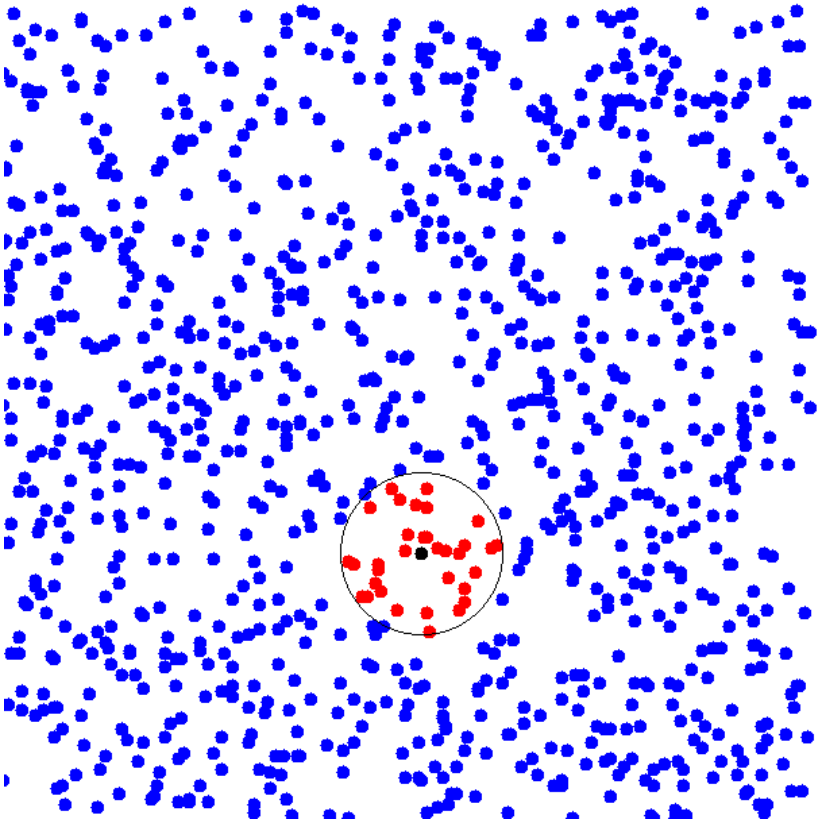
```
# ajout petit_exemple 38 "metallica" ;;
- : unit = ()
# petit_exemple.donnees.(2) ;; (* 38 = 2 mod 3 *)
- : (int * string) list = [(38, "metallica"); (185, "info"); (512, "MPSI")]
```

Question 7. 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.

```
# suppression petit_exemple 498 ;;
- : unit = ()
# petit_exemple.donnees.(0) ;;
- : (int * string) list = [(15, "truc"); (468, "ocaml"); (144, "TP")]
```

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

Le but de cette section est de répondre efficacement à la question : parmi un nuage de points du plan, quels sont ceux qui sont à une distance au plus $d_{\max}$ d'un certain point (x, y) ? La figure suivante montre un nuage de 1000 points, et ceux en rouge sont proches du point (x, y) noir (c'est-à-dire à au plus une certaine distance $d_{\max}$ symbolisée par le cercle).



A priori, il n'y a pas vraiment d'autre solution que de parcourir intégralement l'ensemble de points pour tester ceux qui sont à une distance d'au plus $d_{\max}$ de (x, y). Par contre, dans une situtation où on a besoin d'effectuer plusieurs fois ce genre de requêtes (le nuage étant fixé), il peut être utile d'avoir une bonne structure de données pour stocker le nuage de points. Pour simplifier, on suppose que $d_{\max}$ est fixée aussi. Supposons que le nuage de points soit un ensemble d'entiers inclus dans $[\![0, d-1]\!] \times [\![0, d-1]\!]$. Une idée est de partitionner le plan en secteurs carrés de taille $d_{\max} \times d_{\max}$: il y aura donc $(d/d_{\max})^2$ secteurs, et l'idée va être de stocker les points du nuage dans une table de hachage, la fonction de hachage associant à un point p de $[\![0, d-1]\!] \times [\![0, d-1]\!]$ le secteur dans lequel il se trouve. Lorsqu'on veut déterminer les points du nuage à distance au plus $d_{\max}$ de (x, y) :

— on calcule dans quel secteur se trouve (x, y) ;

— on extrait les points du nuage qui se trouvent dans ce secteur où l'un de ses 8 voisins (on est sûr d'avoir tous les points à distance au plus $d_{\max}$, voir figure 1)

— on garde parmi ces points-là uniquement ceux qui sont à distance au plus $d_{\max}$ de (x, y).

L'intérêt ce cette approche est que si les points sont bien répartis dans $[\![0, d-1]\!]^2$, on ne va tester si un point p du nuage est à distance au plus $d_{\max}$ de (x, y) que pour très peu de points par rapport à l'examen du nuage en entier : uniquement ceux qui sont dans l'un des 9 secteurs décrits précédemment.

Question 8. On fixe la distance maximale dans une constante `distance_max` (voir fichier joint). Écrire une fonction `creation_hach nuage d` prenant en paramètres le nuage (sous forme de tableau de couples d'entiers) et la borne d, créant et renvoyant la table de hachage dans laquelle les points du nuage sont stockés. On utilisera la fonction de hachage

$$H(x, y) = \left\lfloor \frac{x}{d_{\max}} \right\rfloor + \frac{d}{d_{\max}} \left\lfloor \frac{y}{d_{\max}} \right\rfloor$$

et on stockera dans la table des éléments de la forme $((x, y), i)$, où i est l'indice dans le nuage du point (x, y) (la clé est (x, y), l'élément est i). Pour simplifier, on suppose que $d_{\max}$ divise d.

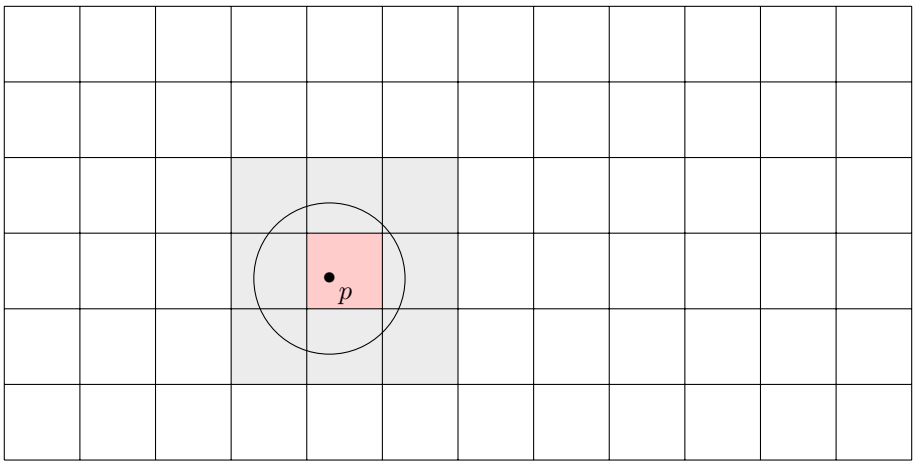


FIGURE 1: Le secteur contenant $p = (x, y)$, et ses 8 secteurs voisins

Question 9. Application : générer un nuage aléatoire avec la fonction fournie et stocker dans une variable la table de hachage donnée par la fonction précédente.

Question 10. Écrire une fonction `candidats_accessibles d x y th` prenant en paramètre :

— la distance d ;

— deux entiers $x, y \in [\![0, n-1]\!]$;

— une table de hachage (semblable à la précédente)

et renvoyant la liste des éléments du nuage (sous la forme $((x', y'), i)$ tels qu'ils sont stockés dans la table de hachage) situés dans le secteur de (x, y) ou dans un secteur voisin. Indications :

— `@` permet la concaténation de listes. Utiliser une référence vers une liste initialement vide ;

— les secteurs voisins sont ceux associés à $(x+\mathrm{d}x, y+\mathrm{d}y)$, avec $\mathrm{d}x, \mathrm{d}y \in \{-d_{\max}, 0, d_{\max}\}$, si le point $(x+\mathrm{d}x, y+\mathrm{d}y)$ est bien dans $[\![0, d-1]\!]^2$ (il peut y avoir moins de 8 secteurs voisins si (x, y) est proche d'un « bord » du nuage).

Question 11. En déduire enfin une fonction `accessibles d x y th nuage` qui donne les indices des éléments du nuage qui sont à une distance au plus $d_{\max}$ de (x, y). Tester avec le code fourni.

4 Tables de hachage dynamiques

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 clé considérée. Dans les tables que nous avons considérées 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 est susceptible d'augmenter et, par conséquent, le coût des recherches aussi. Dans cette section, on se propose d'améliorer ce point en utilisant des tables de hachage de taille variable : 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) table_dyn = {hache: int -> 'a -> int ; mutable taille: int ; mutable donnees: ('a * 'b) list array} ;;
```

Dans cette nouvelle représentation, on dispose d'un champ supplémentaire, `taille`, 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. La fonction de hachage d'une table de hachage dynamique prend également un argument supplémentaire : la largeur de hachage. Ainsi, notre nouvelle fonction de hachage peut être vue comme une famille de fonctions $(h_\omega)_{\omega>0}$.

Question 12. Écrire une fonction `creer_table_dyn h w`, prenant pour arguments une fonction de hachage et une largeur initiale et créant une table dynamique. Écrire ensuite deux fonctions `recherche_dyn` et `element_dyn`. Voici les signatures :

```
creer_table_dyn: (int -> 'a -> int) -> int -> ('a, 'b) table_dyn
recherche_dyn: ('a, 'b) table_dyn -> 'a -> bool
element_dyn: ('a, 'b) table_dyn -> 'a -> 'b
```

On utilisera 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 courante w, alors on réarrange la table sur une largeur $2w$ avant de procéder à l'ajout.

Question 13. Écrire une fonction `rearrange_dyn t` qui réarrange une table en doublant sa largeur. Elle a pour signature `('a, 'b) table_dyn -> unit`

Question 14. En déduire une fonction `ajout_dyn t k e` de signature `('a, 'b) table_dyn -> 'a -> 'b -> unit` ajoutant l'entrée (k, e) à la table t en effectuant un réarrangement si nécessaire.

On s'intéresse maintenant à la suppression d'une entrée : À la suppression d'un élément, si la table devient trop « creuse », on peut réduire sa largeur pour économiser de la mémoire.

Question 15. Que pensez-vous de la stratégie consistant à réarranger une table lors d'une suppression si sa taille devient inférieure à sa largeur (raisonner en terme de complexité amortie) ? Proposer une autre solution et implémenter ainsi une fonction `suppression_dyn` de signature `('a, 'b) table_dyn -> 'a -> unit` supprimant une entrée d'une table dynamique.

Avec la structure ainsi implémentée, on obtient une structure très efficace. Sous réserve d'une bonne répartition des clés dans les alvéoles, les opérations de création, recherche de clé, et modification se font en temps constant, tandis que les opérations d'ajout et de suppression sont en temps constant amorti. De plus, l'espace mémoire utilisé est à tout moment linéaire en la somme des tailles des données stockées.