TP n° 35 : Tables de hachage

Soient $\mathcal{K}$ et $\mathcal{V}$ deux ensembles, les éléments de $\mathcal{K}$ étant appelés *clés* et les éléments de $\mathcal{V}$ étant appelés *valeurs*. Un *dictionnaire* (ou *tableau associatif*) d de $\mathcal{K}$ vers $\mathcal{V}$ associe à chaque clé k une valeur $v = d(k)$. Formellement, un dictionnaire est une fonction $d \subseteq \mathcal{K} \times \mathcal{V}$ tel que pour toute clé $k \in \mathcal{K}$ il existe au plus une valeur $v \in \mathcal{V}$ telle que le couple (k, v) soit dans d. On s'intéresse principalement aux opérations de recherche de la valeur associée à une clé, de l'ajout (ou de la mise à jour) d'un couple clé-valeur ou de la suppression d'une clé et de la valeur associée. On cherche évidemment une complexité optimale pour ces opérations, ce que l'on peut réaliser en $\Theta(n)$ avec des listes de couples, en $\Theta(\log n)$ avec des arbres binaires de recherche et en $\Theta(1)$ en moyenne avec des tables de hachage.

Exemple 1

Pour compter les mots d'un texte, on peut associer à chaque mot (la clé) son nombre d'occurrences (sa valeur).

Exemple 2

*Le DNS (*Domain Name System*) permet de traduire des noms de domaines internet (la clé) en adresse IP (*Internet Protocol*). Par exemple, cela permet d'associer au nom de domaine* `www.data.gouv.fr` *l'adresse IP* `37.59.183.93`.

De manière plus générale, on peut envisager des liaisons multiples [1]. Une même clé $k \in \mathcal{K}$ peut alors être liée à plusieurs valeurs $v \in \mathcal{V}$ dans la table. Chaque nouvelle liaison *masque* une liaison précédente. La recherche de *la* valeur associée à une clé correspond à la dernière valeur associée à cette clé. La suppression de *la* valeur associée à une clé supprime uniquement la dernière valeur associée à cette clé et *révèle* ainsi la liaison précédente.

1. Les tables de hachage avec liaisons multiples ne sont pas au programme, mais c'est le cas des tables de hachage proposées dans le module `Hashtbl` qui lui est bien au programme.

Exemple 3

En CAML*, l'environnement peut être vu comme un dictionnaire qui associe à chaque identifiant (une clé) sa valeur (une valeur). Par exemple, après*

OCAML
```
let reponse = 42
let zero = 0
let ma_reponse =
  let reponse = 43 in
  reponse - 1 + zero
```

l'environnement associé à la clé `reponse` *la valeur* `42`*, à la clé* `zero` *la valeur* `0` *et à la clé* `ma_reponse` *la valeur* `42`*. Pour évaluer la valeur à associer à* `ma_reponse`*, une nouvelle liaison associant* `reponse` *à* `43` masque *temporairement l'ancienne liaison. Lorsque qu'un identifiant apparaît dans une expression (ici* `zero`*) il faut rechercher dans l'environnement la (dernière) valeur associée à cette clé.*

1 Listes d'association

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 $\mathcal{K} \times \mathcal{V}$.

Question 1

Écrire une fonction `assoc : 'a -> ('a * 'b) list -> 'b` telle que `assoc k l` renvoie la valeur associée à la clé `k` dans la liste d'associations `l`. Si plusieurs valeurs sont associées à cette clé, cette fonction doit renvoyer la première rencontrée dans la liste. Si la clé n'est pas présente, cette fonction doit lever l'exception prédéfinie `Not_found`.

Question 2

Écrire une fonction `assoc_opt : 'a -> ('a * 'b) list -> 'b option` qui a le même comportement que `assoc` mais qui renvoie une option sur la première valeur associée à une clé s'il en existe une et `None` sinon.

Question 3

Écrire une fonction `mem_assoc : 'a -> ('a * 'b) list -> bool` qui a le même comportement que `assoc` mais qui renvoie simplement `true` si une liaison existe et `false` sinon.

Question 4

Écrire une fonction `remove_assoc : 'a -> ('a * 'b) list -> ('a * 'b) list` telle que `remove_assoc k l` supprime la première liaison de clé `k` s'il en existe une et renvoie la liste à l'identique sinon.

☞ **Remarque 1**

Ce sont les fonctions `List.assoc`*,* `List.assoc_opt`*,* `List.mem_assoc`

Question 5

Quelles sont les complexités dans le pire et dans le meilleur cas de ces fonctions ? Intuitivement quelle pourrait être la complexité en moyenne de ces fonctions (pour des données réparties de manière intuitivement uniforme, sans rentrer dans les détails).

2 Arbres binaires de recherche

Une implémentation d'un dictionnaire peut également être obtenu en utilisant des arbres binaires de recherche dont les nœuds sont étiquetés par des couples de $\mathcal{K} \times \mathcal{V}$.

Question 6

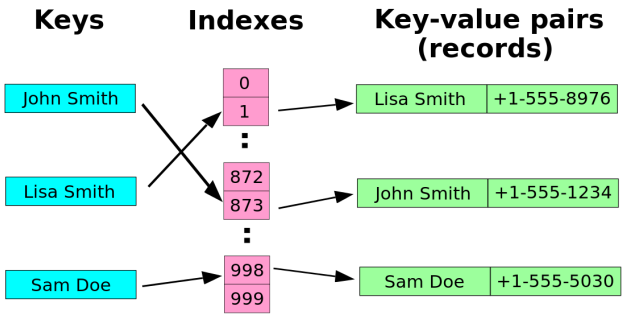
Fermer les yeux et se remémorer rapidement comment implémenter les opérations de recherche, d'ajout et de suppression dans le cadre d'un *dictionnaire* — et non plus d'un ensemble — implémenté par un arbre binaire de recherche. Nous n'avons pas le temps de le faire dans le cadre ce TP mais c'est un *excellent* exercice de révision.

Question 7

Comment peut-on garantir que les arbres sont équilibrés ? Quelles sont alors les complexités dans le pire et dans le meilleur cas ? Intuitivement quelle pourrait être la complexité en moyenne (pour des données réparties de manière intuitivement uniforme, sans rentrer dans les détails).

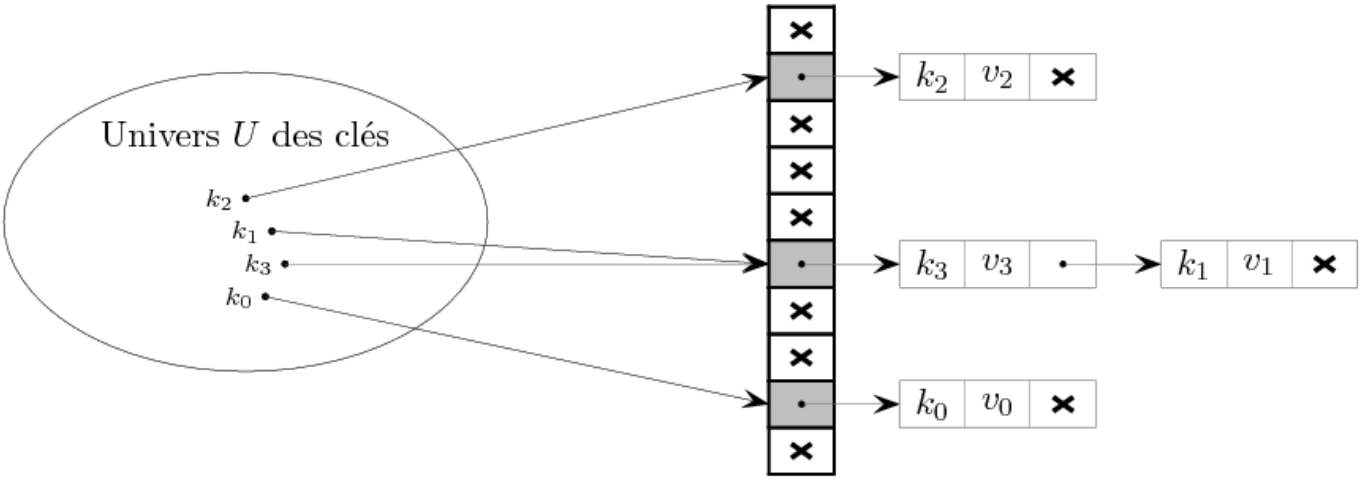
3 Tables de hachage

Les tables de hachage (*hash tables* en anglais) sont une implémentation souvent plus efficace d'une telle structure de données. Sous des hypothèses raisonnables, on peut montrer qu'en moyenne la complexité des opérations ci-dessus est en $\Theta(1)$.



L'idée est de généraliser la notion de tableau à accès direct, en utilisant les clés comme indices d'un tableau de taille $m \in \mathbb{N}$. Seulement les clés n'ont aucune raison d'être des entiers dans $[\![0; m-1]\!]$. L'idée est alors, pour chaque clé $k \in \mathcal{K}$, de calculer un indice $h_m(k)$ compris entre 0 et $m-1$, appelée valeur de hachage de k à l'aide d'une *fonction de hachage* $h_m : \mathcal{K} \to [\![0; m-1]\!]$. L'entier m est appelé la *largeur* de la table. On utilise ensuite un tableau de taille m pour stocker les valeurs associées aux clés. Les cases de ce tableau, lui-même appelé *table de hachage* sont appelées *alvéoles*. Pour un couple clé-valeur (k, v) on stocke donc la valeur v dans l'alvéole $h_m(k)$. On dit que l'on a *haché* l'élément de clé k dans l'alvéole $h_m(k)$.

En pratique il est difficile, et même le plus souvent impossible de construire une fonction de hachage h_m injective. L'univers possibles des clés $\mathcal{K}$ est souvent très grand par rapport au nombre effectif de couples clés-valeurs réellement présents, nombre que nous noterons n, et on aimerait que la largeur m de la taille soit de l'ordre de n et surtout pas de l'ordre de $|\mathcal{K}|$ qui peut même être potentiellement infini (par exemple si $\mathcal{K}$ désigne l'ensemble des noms de domaines possibles ou l'ensemble des phrases possibles en langue française). Inévitablement donc, certaines clés seront hachées vers la même alvéole ce qui compromet donc notre approche. Lorsque deux clés sont hachées vers une même alvéole, on dit qu'il y a *collision*.



Une solution est d'utiliser, dans l'alvéole i, une liste d'association pour stocker tous les couples (k, v) de la table d'association tels que $h_m(k) = i$. Dans le pire cas, tous les éléments seront hachés vers la même alvéole et on retrouve la même complexité que celle du pire cas de la partie 1. Mais on peut montrer que si l'on choisit la fonction de hachage de manière relativement aléatoire, les listes d'associations des alvéoles auront, en moyenne, une taille bornée.

4 Fonctions de hachage

Pour implémenter une table de hachage, il est donc nécessaire de disposer d'une fonction de hachage h_m de K vers $[\![0, m-1]\!]$. Comme indiqué précédemment, 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, que étant donné un entier i de $[\![0, m-1]\!]$, la probabilité que $h_m(k) = i$ soit voisine de $1/m$. Dans cette section, nous donnons quelques exemples de ce que pourraient être des fonctions de hachage simplifiées.

4.1 Entiers naturels

Dans le cas où les clés sont des entiers naturels, le *hachage par division* de largeur m consiste à hacher la clé entière k en utilisant le reste de la division de k par m (`k mod m`) qui appartient bien à l'intervalle $[\![0, m-1]\!]$.

Question 8

Écrire une fonction `hash_int : int -> int -> int` telle que `hash_int m k` renvoie le haché de la clé entière k en utilisant un hachage par division de largeur `m`.

Le hachage par division est simple mais rarement une bonne idée. Il est rare que la répartition des clés ne permette à un hachage par division de donner de bons résultats. Une autre possibilité,

généralement meilleure est d'utiliser *hachage par multiplication*. Pour cela, on choisit un réel $a \in]0;1[$ fixé et on définit h_m par

$$h_m(k) = \lfloor m(ak - \lfloor ak \rfloor) \rfloor$$

4.2 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 255 identifiant le caractère de manière unique). Une chaîne de caractères $s = s_0 \cdots s_{n-1}$ peut alors être vue comme la représentation en base 256 de l'entier

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

Le code d'un caractère est renvoyé en Caml par la fonction `int_of_char : char -> int`.

Question 9

Écrire une fonction `hash_string: int -> string -> int` qui réalise un hachage par division de l'entier qui représente une chaîne de caractère pour ainsi réaliser une fonction de hachage pour les chaînes de caractères. *Indication : réduire modulo m au plus tôt en utilisant un schéma de Horner pour ne pas avoir de dépassements d'entiers.*

Question 10

Pour $m = 100$ que vaut le haché de « Gros », « Gras », « Gres », « Gris », « J'aime le hachage », de votre prénom et nom, de votre nom et prénom ? Pouvez-vous trouver une collision ?

4.3 La fonction `Hashtbl.hash`

Le module `Hashtbl` fournit une fonction de hachage polymorphe `Hashtbl.hash : 'a -> int` qui permet de hacher n'importe quel objet CAML, de manière compatible avec l'égalité structurelle (si `x = y` alors `Hashtbl.hash x = Hashtbl.hash y`).

Question 11

Que vaut le haché pour cette fonction de hachage de 42, 43, 44 et 45 ? Et celui de votre prénom et nom, de votre nom et prénom ? Pouvez-vous trouver facilement une collision ? En existe-t-il ?

On peut ensuite réduire modulo m si l'on souhaite un indice dans $[\![0; m-1]\!]$. On dispose ainsi d'une famille de fonctions de hachage $(h_m)_{m \in \mathbb{N}^*}$ sous la forme d'une fonction `hash : int -> 'a -> int` telle que l'application partielle `hash m` de type `'a -> int` donne une fonction de hachage $h_m : \mathcal{K} \to [\![0; m-1]\!]$.

OCAML
```
let hash m x = Hashtbl.hash x mod m
```

Par la suite on utilisera cette fonction `hash : int -> 'a -> int` générique.

5 Implémentation des tables de hachage

✔

Toutes ces fonctions on la même spécification que celles du module `Hashtbl` et sont au programme. Il faut savoir les implémenter et les utiliser. On pourra consulter la documentation de ce module si besoin : https://ocaml.org/api/Hashtbl.html

Nous représentons en CAML une table de hachage de clés de type `'a` vers des valeurs de type `'b` par un enregistrement de type `('a, 'b) hashtbl` :

OCAML
```
type ('a, 'b) hashtbl = {
  mutable data : ('a * 'b) list array;
  mutable size : int;
  mutable h : 'a -> int
}
```

Si `t` est une une table de hachage de type `('a, 'b) hashtbl`, la tableau des alvéoles est `t.data` de largeur m. Remarquons que ce tableau est modifiable, ce qui nous permettra par la suite de le redimensionner si nécessaire. On conserve à tout moment dans le champ `t.size`, pour y avoir accès en temps constant, le nombre d'entrées de la table de hachage (y compris les liaisons masquées). Ce champ est également déclaré mutable de manière à pouvoir être mis à jour à chaque ajout ou suppression. Pour plus de simplicité on conserve également dans le champ `t.h` une fonction $h_m : \mathcal{K} \to [\![0; m-1]\!]$, où m est la largeur de la table, qui est la fonction de hachage utilisée pour hacher les clés stockées dans la table. Ce champ est encore modifiable puisque la fonction de hachage change si la largeur de la table change. Rappelons que la case d'indice i du tableau `t.data` contient la liste des entrées (k, v) de la table tels que $h_m(k) = i$.

Question 12

Écrire une fonction `create : int -> ('a, 'b) hashtbl` telle que l'appel `create m` renvoie une nouvelle table de hachage vide de largeur $m > 0$.

Question 13

Écrire une fonction `length : ('a, 'b) hashtbl -> int` qui renvoie le nombre d'entrées de la table (y compris les liaisons masquées). Quelle est sa complexité ?

Nous allons maintenant écrire des fonctions permettant d'effectuer les opérations de base sur ces tables : recherche, ajout et suppression, sans se préocuper pour l'instant de la taille des listes de collisions.

Question 14

Écrire une fonction `mem : ('a, 'b) hashtbl -> 'a -> bool` telle que `mem t k` renvoie un booléen indiquant si la clé `k` est présente dans la table `t`.

Question 15

Écrire une fonction `add : ('a, 'b) hashtbl -> 'a -> 'b -> unit` telle que `add t k v` ajoute l'entrée (k, v) à la table de hachage t. Une liaison précédente est masquée par cette nouvelle liaison mais pas supprimée (même comportement qu'avec les listes d'asso-

ciations).

Question 16

Écrire une fonction `find : ('a, 'b) hashtbl -> 'a -> 'b` telle que `find t k` renvoie la valeur `v` associé à la clé `k` dans la table `t`. Si plusieurs valeurs sont associées à cette clé, cette fonction doit renvoyer la dernière association ajoutée à la table. Si la clé `k` n'est pas présente dans la table `t`, la fonction doit l'exception `Not_found`.

Question 17

Deviner ce que doit réaliser, puis implémenter, la fonction `find_opt : ('a, 'b) hashtbl -> 'a -> 'b option`

Question 18

Écrire une fonction `find_all : ('a, 'b) hashtbl -> 'a -> 'b list` qui renvoie la liste de toutes les associations d'une clé avec les valeurs associées (masquées ou non[a]).

a. Exactement une liaison n'est pas masquée, sauf si la clé n'est pas présente auquel cas il n'existe aucune liaison.

Question 19

Écrire une fonction `remove : ('a, 'b) hashtbl -> 'a -> unit` telle que `remove t k` supprime la dernière entrée associant la clé `k` dans la table `t`. Ceci révèle éventuellement une association antérieure. Si la clé n'était pas présente cette fonction n'a pas d'effet.

Question 20

Écrire une fonction `replace : ('a, 'b) hashtbl -> 'a -> 'b -> unit` telle que `replace t k v` ajoute l'entrée (k, v) à la table de hachage t. Une éventuelle liaison précédente est remplacée par cette nouvelle liaison et donc supprimée (uniquement la plus récente s'il y en avait plusieurs. Remarquons que cela revient à effectuer `remove t k` suivi de `add t k`, mais on demande ici de l'implémenter directement.

Question 21

Écrire une fonction `iter : ('a -> 'b -> unit) -> ('a, 'b) hashtbl -> unit` telle que l'appel `iter f t` avec une fonction `f : 'a -> 'b -> unit` applique la fonction `f` à toutes les entrées de la table de hachage. La fonction `f` sur une entrée (k, v) reçoit `k` comme premier paramètre et `v` comme second. Chaque liaison est présenté exactement une fois à la fonction. L'ordre dans lequel cette fonction est effectuée sur les associations n'est pas spécifiés, mais si la table contient des liaisons multiples, celles-ci doivent être passées à f dans l'ordre inverse de leur introduction dans la table. En particulier, la liaison la récente (qui masque les autres) est passée en premier.

6 Utilisation de tableaux dynamiques

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 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, augmenter le coût des recherches.

Dans cette section, on se propose d'améliorer ce point en utilisant des tableaux dynamiques : 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.

Nous utiliserons le principe de redimensionnement suivant : lors de l'ajout d'une entrée, si la taille de la table (c'est-à-dire le nombre d'entrées) dépasserait le double de la largeur courant m, alors on réarrange la table sur une largeur $2m$ avant de procéder à l'ajout.

Question 22

Écrire une fonction `resize : ('a, 'b) hashtbl -> unit` qui réarrange une table de hachage en doublant sa largeur. *Attention : la fonction de hachage change et il faut remettre les éléments « au bon endroit dans le bon ordre »*

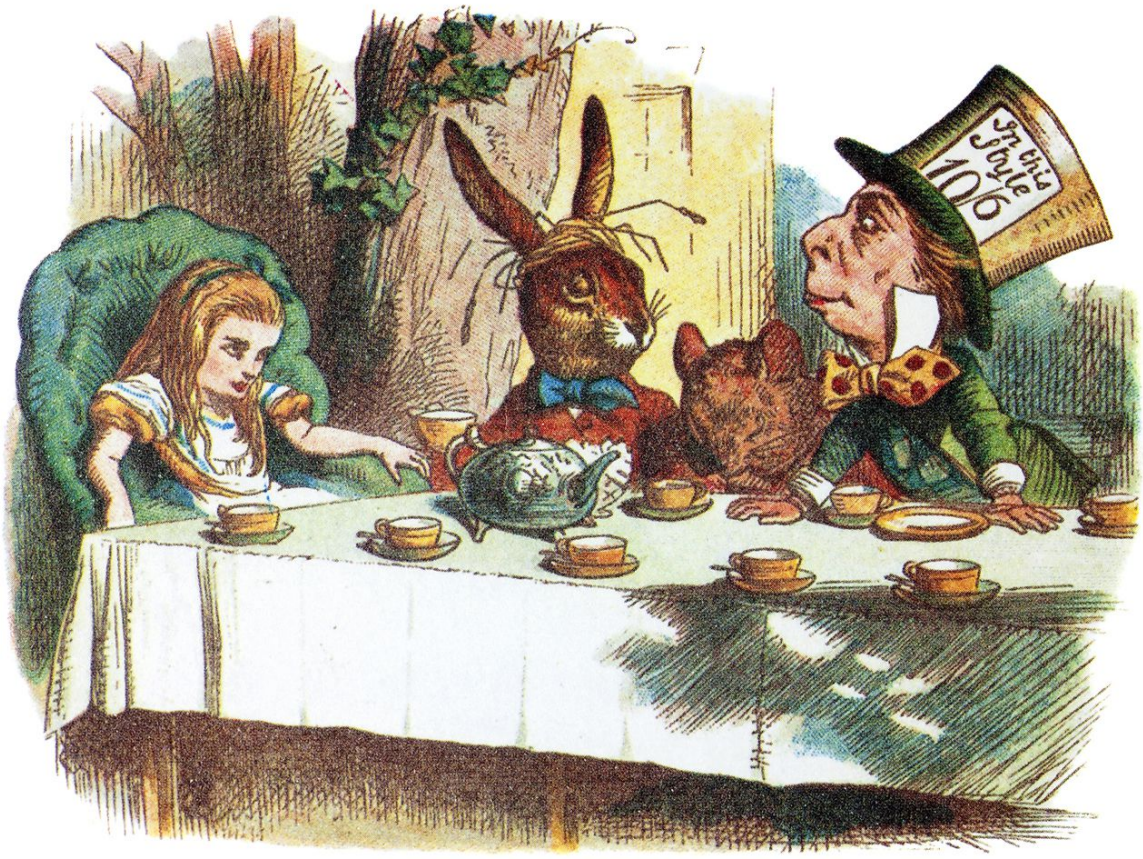
Question 23

Modifier la fonction `add` pour tenir compte de cette stratégie.

Question 24

Que pourrait-on faire pour économiser de la place si le nombre d'éléments devient suffisamment petit ? Quelle stratégie mettre en place ?

7 Application



Le fichier `alice_in_wonderland.txt` contient le texte du roman publié en 1865 par Lewis CARROLL. Par la suite on considère que les espaces et les retours à la ligne séparent le textes en *mots*. Un mot est donc une suite de lettre consécutives sans espace ou retour à la ligne maximale.

Question 25

Écrire une fonction `split : string -> string list` qui découpe une chaîne de caractère suivant les espaces (donc en mots).

Question 26

Afficher les mots dont le nombre d'occurrences dans le texte est supérieur ou égale à 20 avec ce nombre d'occurrences (peu importe l'ordre).

Question 27

Quelle est la proportion de mots utilisées une seule fois ?

8 Statistiques

On considère le type suivant, défini dans le module `Hashtbl`.

OCaml
```
type statistics = {
  (* Number of bindings present in the table. Same value as returned
     by Hashtbl.length. *)
  num_bindings : int;
  (* Number of buckets in the table. *)
  num_buckets : int;
  (* Maximal number of bindings per bucket. *)
  max_bucket_length : int;
  (* Histogram of bucket sizes. This array histo has length
     max_bucket_length + 1. The value of histo.(i) is the number of
     buckets whose size is i. *)
  bucket_histogram : int array
}
```

Question 28

Écrire une fonction `stats : ('a, 'b) hashtbl -> statistics` qui calcule et renvoie ces statistiques.