Dictionnaires

I Principe d'un dictionnaire

I.1 Définition

Un dictionnaire est une structure de donnée abstraite permettant de représenter une association entre des clés et des valeurs.

Par exemple, on pourrait imaginer que les clés soient des chaînes de caractères et les valeurs des nombres.

Plus précisément, si le type des clés est k et celui des valeurs v. Un dictionnaire est la donnée d'un type `t` (dépendant de `k` et de `v`) et des opérations :

- création d'un dictionnaire vide
- ajout d'une correspondance entre une clé `c` et une valeur `v`
- suppression d'une correspondance associée à une clé
- test de la présence d'une correspondance vis-à-vis d'une clé
- récupération de la valeur associée à une clé

On remarque qu'il est possible d'avoir une structure mutable ou non. Dans une structure mutable, le dictionnaire est modifié par les opérations alors que dans une structure immuable, les opérations renverront un nouveau dictionnaire.

Un point crucial des dictionnaires est d'avoir une relation d'égalité, ou même une relation d'ordre sur les clés.

I.2 Lien avec les ensembles finis

On retrouve beaucoup de points communs entre cette structure de donnée abstraite et celle d'ensemble finis. En effet, il suffit de considérer une ensemble fini de couples (clé, valeur) et de ne faire porter les comparaisons entre élément de l'ensemble uniquement sur la première composante, la clé.

Ainsi, dans l'ensemble `{ ("riri", 12), ("fifi", 14) }` on pourra récupérer le couple de la forme `("fifi", *)` pour obtenir la valeur qui lui est associé.

Si on sait implémenter des ensembles finis, on peut en déduire une implémentation des dictionnaires.

Pour cela, il suffit de considérer une interface sur les ensembles finis permettant de définir sa propre fonction de comparaison, égalité ou ordre selon les cas. On va donc chercher à implémenter la structure, immuable ou mutable, suivante :

- création d'un ensemble vide
- ajout d'un élément
- suppression d'un élément
- test et récupération d'un élément

La récupération de l'élément est crucial dans le cas des dictionnaires. En effet, dans l'exemple précédent, on obtiendra le couple `("fifi", 14)` en testant la valeur `"fifi"` grâce à la fonction de comparaison ne portant que sur la première composante.

II Implémentation immuable : ABR

Dans cette implémentation, on utilise les arbres binaires de recherches vu dans le chapitre sur les arbres et dont on sait qu'il peuvent représenter des ensembles finis de valeurs **pour lesquelles on a un ordre total**.

Pour implémenter un dictionnaire, on considère un ensemble C totalement ordonné de *clés* et un ensemble V de valeurs. On va représenter un dictionnaire par un ABR étiquetté par $C \times V$. Pour effectuer des opérations sur l'arbre, on se contente de regarder la première composante. Lors d'une recherche, on peut renvoyer la valeur.

On a alors ajout, suppression et recherche en $O(\log n)$ où n est le nombre d'entrées **sous réserve de bon équilibrage des abr**.

L'implémentation se déduit alors très naturellement de celle des arbres binaires de recherche :

OCaml
```
type ('a, 'b) dico = Nil
    | Noeud of ('a, 'b) dico * ('a * 'b) * ('a, 'b) dico

let rec recherche dico cle =
    match dico with
    | Nil -> None
    | Noeud(_, (y,v), _) when cle = y -> v
    | Noeud(g, (y,_), _) when cle < y -> recherche g cle
    | Noeud(_, (y,_), d) -> recherche g cle
```

III Implémentation mutable : les tables de hachage

III.1 Hachage

Le principe du hachage (*hashing* en anglais) est, étant donné un ensemble de valeurs E et un entier n, de réaliser une **bonne** fonction

$$h : E \to [\![0, n-1]\!]$$

Que veut-on dire par **bonne** fonction ? Naturellement, avec E infini ou de cardinal très grand devant n, il est illusoire d'imaginer pouvoir réaliser une fonction injective. Cependant, on peut faire en sorte que deux valeurs proches aient des valeurs associées, on parle de *hâchés*, différents. Cela représenterait ainsi une forme d'injectivité locale.

Quand $x \neq y$ et $h(x) = h(y)$, on dit que la paire $\{x, y\}$ est une *collision* pour h.

Exemple Si $E \subset \mathbb{N}$, on peut considérer le reste modulo n comme fonction de hachage. Des éléments consécutifs auront ainsi des hachés distincts.

Toutes les paires $\{kn + r, ln + r\}$, où $k \neq l$ et $r \in [\![0, n-1]\!]$, sont des collisions.

Ainsi, cette fonction est très prévisible.

Remarque Dans un contexte cryptographique, par exemple pour stocker un mot de passe, on souhaite éviter de remonter du haché à la valeur. En effet, en cas de compromission des accès, il suffirait d'*inverser* h pour retrouver le mot de passe de l'utilisateur.

La fonction de l'exemple précécent est ainsi peu sécurisé.

Il existe des fonctions de chachages comme `SHA1` qui sont securisées mais peuvent être mal utilisées : en hachant tous les mots du dictionnaire, on peut constituer sa propre table de correspondance. On parle de *rainbow table* et une manière simple de s'en prémunir et de ne pas stocker les mots de passe hachés mais la concaténée d'un mot de passe et d'une donnée générée par le serveur : le sel.

III.2 Tables de hachage pour un ensemble fini

L'idée première des tables de hachage pour stocker des éléments de E est d'utiliser un tableau de n cases et une fonction de hachage $h : E \to [\![0, n-1]\!]$. Ainsi, pour stocker $x \in E$, on le place dans le tableau à l'adresse $h(x)$.

Naturellement, cette idée est très naïve en raison des collisions éventuelles. Tout l'intérêt des tables de hachages est donc de gérer les collisions. Il y a plusieurs stratégies plus ou moins efficaces en machine.

Pour chacune de ces stratégies, il sera donc nécessaire de faire plus de travail qu'un simple accès dans une case. En général, on a des stratégies qui sont linéaires dans le nombre d'éléments en collision. En pire cas, ce

nombre est celui du nombre d'éléments. Mais, en pratique, si les collisions sont peu probables, trois éléments entrant en collision le seront encore moins, ce qui fait qu'on peut supposer que le nombre de collisions est faible devant le nombre d'éléments. On pourra donc faire l'hypothèse que les accès sont en $O(1)$.

On va montrer ici l'implémentation de différentes stratégies en utilisant un même cadre : on cherche à représenter un ensemble fini d'éléments de type `'a` où on on a une fonction de hachage `hache : 'a -> int` et un test de comparaison `eq : 'a -> 'a -> bool` testant l'égalité.

On considère aussi définie une constante `n` et l'assurance que `hache x` soit dans $[\![0, n-1]\!]$.

III.3 Listes de collisions

La stratégie la plus simple de gestion des collisions est de ne pas stocker des éléments dans chaque case mais une liste chaînée d'éléments.

Cette stratégie a le désavantage de fragmenter la mémoire car il faut descendre le long des maillons.

OCaml
```
type htbl = 'a list array

let htbl_creer () = Array.make n []

let htbl_cherche t x =
    let found = ref None in
    List.iter (fun y -> if eq x y then found := y) t.(hache x)

let htbl_ajout t x =
    let h = hache x in
    t.(h) <- x : : t.(h)

let htbl_supprime t x =
    let h = hache x in
    t.(h) <- List.filter (fun y -> not (eq x y)) t.(h)
```