TP : Hashtbl en OCaml

L'objectif de ce TP est l'implémentation en OCaml d'une structure de dictionnaire par table de hachage. Pour simplifier, les clés seront de type `string`.

Par convention, pour une chaîne de caractère s, on note s_i son i-ème caractère.

Fonction de hachage

Le haché *DJB2 (Dan Bernstein)* est un algorithme simple de hachage de chaîne de caractères datant de 1991. Il associe un entier positif à toute chaîne de caractères.

On donne, pour l'implémenter, le code suivant :

```
let hash_djb2 (s : string) : int =
  let h = ref 5381 in
  String.iter (fun c ->
    h := (!h lsl 5) + !h + Char.code c
    )
    s;
  !h land max_int;;
```

Dans ce code, `Char.code c` désigne le code ASCII du caractère `c` (ainsi, `Char.code 'a'` vaut 97).

Si `s` est de longueur n et $i \in [\![0, n-1]\!]$, on note s_i le i-ème caractère de s. Par exemple, si `s` est `"abcd"` alors s_0 désigne `'a'`.

Question 1.

L'appel `hash_djb2 s` effectue AVANT la dernière ligne le calcul $P(x)$ où P est un polynôme et x une valeur.

Identifier P et x.

Question 2.

Expliquer la dernière ligne.

Remarque. OCaml fournit une fonction de hachage `Hashtbl.hash` disponible par défaut.

```
# Hashtbl.hash "toto";;
- : int = 946608321
```

Dictionnaire par table de hachage

On veut implanter le module dont voici la signature :

```
module StringHashtbl :
  sig
    type key = string
    type 'v bucket = (key * 'v) list
    type 'v sht = {
      mutable buckets : 'v bucket array;
      hash : key -> int;
      mutable size : int;
    }
    val max_load_factor : float
    val create : ?hash:(key -> int) -> int -> 'a sht
    val resize : 'a sht -> unit
    val add : 'a sht -> key -> 'a -> unit
    val find : 'a sht -> key -> 'a
    val remove : 'a sht -> key -> unit
  end
```

Il réalise une structure de dictionnaire par table de hachage avec gestion des collisions par chaînage et redimensionnement si le tableau sous-jacent est trop petit. Les clés sont des chaînes de caractères.

Le type `'v t` est un enregistrement dont les champs sont :

— `buckets` : le tableau sous-jacent dans lequel on met les seaux. On appelle *capacité* la taille de `buckets`.

— `hash` : Une fonction de hachage choisie au moment de la création (par exemple `djb2`).

— `size` : le nombre total d'associations (clé,valeur) dans le dictionnaire.

Question 3.

Écrire la fonction `create ?(hash = Hashtbl.hash) (initial_capacity : int) :` . Elle prend un paramètre optionnel `hash` qui est une fonction de hachage et une indication de la taille du tableau sous-jacent. Elle renvoie un dictionnaire vide. La fonction de hachage fournie par défaut est `Hashtbl.hash`.

```
# StringHashtbl.create 4;;
- : '_weak2 StringHashtbl.sht =
{StringHashtbl.buckets = [|[]; []; []; []|]; hash = <fun>; size = 0}
# StringHashtbl.create ~hash:hash_djb2 4;;
- : '_weak3 StringHashtbl.sht =
{StringHashtbl.buckets = [|[]; []; []; []|]; hash = <fun>; size = 0}
```

Remarque. L'appel `create 4` produit un dictionnaire vide dont la fonction de hachage associée est `Hashtbl.hash`. C'est uniquement si on veut une autre fonction de hachage qu'on renseigne l'argument optionnel.

Lorsque la table devient trop pleine, on double la capacité, puis on réinsère chaque élément dans la nouvelle table. La constante `max_load_factor` est une quantité telle que le redimensionnement est déclenché si le ratio nombre d'associations / capacité est supérieur à cette quantité. On peut prendre $\frac{2}{3}$ comme en `Python` pour cette valeur.

Question 4.

Écrire la fonction `resize (tbl : 'a sht)` double la taille du tableau sous-jacent et réinsère les associations dans l'ordre où elles étaient auparavant.

Question 5.

Écrire fonction `add (tbl : 'a sht) (k : key) (v: 'a) : unit` qui ajoute une association au dictionnaire. Éventuellement, si le ratio taille/capacité est trop élevé, on effectue d'abord un redimensionnement. La fonction `add` se contente d'ajouter une association `(k,v)`. Il est possible qu'une autre association de même clé se trouve plus loin dans le seau.

Question 6.

Écrire la fonction `find (tbl : 'a sht) (k:key) : 'a` qui renvoie la valeur associée à la clé passée en paramètre et soulève `Not_found` sinon. Si plusieurs associations de même clé se trouvent dans le seau, alors `find` renvoie la valeur de la première rencontrée sans se préoccuper des autres.

Question 7.

Écrire la fonction `remove (tbl : 'a sht) (k:key) : unit` qui supprime une association dont la clé est `k`. Elle soulève `Not_found` en cas d'absnece. Si plusieurs associations de même clé se trouvent dans le seau, alors `remove` supprime la première rencontrée sans se préoccuper des autres.

Remarque. On ne le demande pas dans ce TP mais la fonction `remove` pourrait agir à l'inverse de la fonction `add` : en cas de ratio taille/capacité trop petit, un redimensionnement pour un tableau des buckets de moindre capacité pourrait être effectué.

```
# let test_resize () =
  (* Crée une table avec petite capacité initiale; fction de ha par défaut *)
  let tbl = StringHashtbl.create 2 in
  Printf.printf "Capacité initiale : %d\n" (Array.length tbl.buckets);
  (* Insère plusieurs paires pour dépasser le seuil de charge *)
  for i = 1 to 20 do
    let key = string_of_int i in
    StringHashtbl.add tbl key i
  done;
  Printf.printf "Capacité après insertions : %d (taille = %d)\n"
    (Array.length tbl.buckets) tbl.size;
  (* Vérification que toutes les clés sont trouvables *)
  for i = 1 to 20 do
    let key = string_of_int i in
    let v = StringHashtbl.find tbl key in
    Printf.printf "  clé %s -> %d\n" key v
  done;;
val test_resize : unit -> unit = <fun>
# let () =
  Printf.printf "=== Test de redimensionnement ===\n";
  test_resize ();;
=== Test de redimensionnement ===
Capacité initiale : 2
Capacité après insertions : 32 (taille = 20)
  clé 1 -> 1
  clé 2 -> 2
  clé 3 -> 3
  clé 4 -> 4
  clé 5 -> 5
  clé 6 -> 6
  clé 7 -> 7
  clé 8 -> 8
  clé 9 -> 9
  clé 10 -> 10
  clé 11 -> 11
  clé 12 -> 12
  clé 13 -> 13
  clé 14 -> 14
  clé 15 -> 15
  clé 16 -> 16
  clé 17 -> 17
  clé 18 -> 18
  clé 19 -> 19
  clé 20 -> 20
```