1 Implémentation d'une structure de tableau redimensionnable

Dans cette section, on implémente une structure de tableau redimensionnable : on va obtenir une structure très proche des listes Python. En Python, une liste `L` supporte les opérations suivantes :

- accès aux éléments (via `L[i]` pour $0 \leq i <$ `len(L)`) en temps constant ;
- modification des éléments (via `L[i]=x`) en temps constant ;
- ajout d'un élément en fin de liste (via `L.append(x)`) en temps constant **amorti** ;
- suppression de l'élément en fin d'une liste non vide (via `L.pop()`) en temps constant.

Un tableau standard permet de réaliser les deux premières opérations (accès et modification des éléments). Pour pouvoir réaliser l'opération d'ajout lorsque le tableau est plein, il faut pouvoir changer le tableau avec lequel on travaille. Pour ce faire, on alloue un nouveau tableau de taille supérieure, dans lequel on copie les éléments de l'ancien tableau, ainsi que le nouvel élément. Passer d'un tableau de taille n à un tableau de taille n + 1 est très mauvais pour l'ajout successif d'éléments. Par contre doubler la taille du tableau alloué permet, lorsqu'on passe d'une taille n à une taille $2n$ (on prendra en fait $2n+1$), de pouvoir réaliser $O(n)$ ajouts en temps constant après le premier ajout (de coût $O(n)$). Ceci mène au type suivant :

```
type 'a tab_redim = {mutable nb: int ; mutable tab: 'a array}
```

Dans une instance de ce type, le champ `nb` contient le nombre d'éléments effectivement présents dans le tableau redimensionnable. Le tableau `tab` contient au moins `nb` éléments, ceux effectivement présents sont ceux d'indice variant entre 0 et `nb` -1.

Rappel : si `t` est de type `tab_redim`, on accède aux champs via `t.nb` et `t.tab`. On peut modifier les champs mutables comme `t.nb` par `t.nb <- ...`.

Exercice 1 : On donne la fonction de création d'un tableau redimensionnable vide, de type `unit -> 'a tab_redim` (remarque : le tableau vide `[| |]` est polymorphe).

```
let creer_tab () = {nb = 0; tab = [| |]} ;;
```

Écrire les fonctions suivantes :

```
acces : 'a tab_redim -> int -> 'a
modif : 'a tab_redim -> int -> 'a -> unit
ajout : 'a tab_redim -> 'a -> unit
suppr : 'a tab_redim -> 'a
```

- `acces t i` et `modif t i x` doivent respectivement renvoyer ou modifier le i-ème élément du tableau redimensionnable `t`.
- `ajout t x` rajoute un élément au tableau redimensionnable. Si le tableau est plein, on le remplacera par un tableau de taille $2n+1$ où n est la taille actuelle de `t.tab`.
- `suppr t` supprime le dernier élément du tableau redimensionnable `t` et le renvoie. En pratique, il suffit de décrémenter `t.nb`, pas besoin de modifier `t.tab`.

Tester vos fonctions, par exemple :

```
let t=creer_tab () ;;

for i=0 to 9 do
  ajout t i
done ;;
```

```
acces t 3 ;;

for i=0 to 9 do
  print_int (suppr t) ; print_string " "
done ;;
```

doit afficher :

```
#- : unit = ()
#9 8 7 6 5 4 3 2 1 0 - : unit = ()
```

Après ce test, `t` doit contenir :

```
#t ;;
- : int tab_redim =
 {nb = 0; tab = [|0; 1; 2; 3; 4; 5; 6; 7; 8; 9; 7; 7; 7; 7; 7|]}
```

2 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. Vous verrez l'an prochain une implémentation de cette structure via les arbres binaires de recherche. On va voir ici une autre implémentation, 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 beaucoup 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.

Exercice 2 (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 $i \in [\![0, w-1]\!]$, la probabilité que $h_w(k) = i$ soit proche de $1/w$. Dans cet exercice, nous donnons quelques exemples de telles fonctions.

Entier 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 ($k \mod w$) qui appartient bien à l'intervalle $[\![0, w-1]\!]$.

1. Écrire une fonction `hachage_entier` de type `int -> int -> int` telle que `hachage_entier w k` retourne le haché de ma 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 de 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 calculé en OCaml par la fonction `int_of_char`, de type `char -> int`.

2. Écrire une fonction `hachage_chaine` de type `int -> string -> int`, de hachage par division des chaînes de caractères. On fera attention à prendre le modulo **régulièrement** pour éviter le dépassement d'entiers en OCaml (prendre le modulo seulement à la fin risque de ne pas suffire).
   Rappel : si `s` est une chaîne de caractère, `String.length s` pour avoir sa longueur, et `s.[i]` pour avoir son i-ème caractère.

```
# 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.

Exercice 3 (Tables de hachage de taille fixe) : On représente en OCaml 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 obtenues dans la table (notée h_w dans l'exercice précédent), et `t.donnees` est un tableau de longueur w. La case numéro i de ce tableau contient la liste des entrée (k, e) de la table telles que $h_w(k) = i$.

1. Écrire une fonction `creer_table` de type `('a -> int) -> int -> ('a, 'b) table_hachage` telle que `creer_table h w` renvoie une 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.

Exemple

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

Dans la suite, on prend comme exemple le `petit_exemple` ci-dessus. 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 $\mod 3$, ceux de la dernière à 2 $\mod 3$.

2. Écrire une fonction `recherche t k` de type `('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 $h_w(k) = i$, avec h_w 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 1ère liste *)
- : bool = true
# recherche petit_exemple 499 ;; (* 499 = 1 mod 3, ne visiter que la 2ème liste *)
- : bool = false
```

3. Écrire une fonction `element t k` de type `('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.
```

4. Écrire une fonction `ajout t k e` de type `('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 déjà présente.

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

5. Écrire enfin une fonction `suppression t k` de type `('a, 'b) table_hachage -> 'a -> unit` supprimant l'entrée de la clé k dans la table `t`. On ne fera rien 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")]
```

Exercice 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 de l'exercice précédent, 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.

On se propose ici 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 $(h_w)_{w>0}$.

1. Écrire une fonction `creer_table_dyn h w`, prenant pour arguments une fonction de hachage et une largeur initiale et créant une table dynamique, ainsi que deux fonctions `recherche_dyn` et `element_dyn`.

```
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 utilise 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.

2. Écrire une fonction `rearrange_dyn t` qui réarrange une table en doublant sa largeur. Elle a pour type `('a, 'b) table_dyn -> unit`.
3. En déduire une fonction `ajout_dyn t k e` de type `('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.

4. 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 type `('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.