TP1 : Autour du hachage.

9 et 16 septembre

Ce TP comporte deux parties indépendantes. La première partie concerne la conception d'un cache LRU en OCaml qui donnera l'occasion d'utiliser la bibliothèque des tables de hachage de OCaml. La seconde partie (à traiter en C) réalise une table de hachage avec résolution des collisions par sondage linéaire (c'est une méthode alternative à la résolution par chainage vue en cours en première année).

1 Cache LRU en OCaml

En informatique, un cache est un composant logiciel ou matériel permettant de délivrer rapidement des données qui ont été déjà obtenues auparavant : on y stocke par exemple des résultats de calculs (permettant ainsi d'éviter de les effectuer à nouveau) ou des données initialement présentes sur une mémoire de masse ou en réseau (et dont le temps d'obtention peut être élévé).

Un cache ayant une capacité limitée, il est nécessaire d'avoir un mécanisme pour décider quelles valeurs garder en mémoire. Parmi les différents mécanismes possibles, la méthode LRU (pour least recently used) est particulièrement simple et efficace : on ne garde en mémoire que les données ayant été utilisées le plus récemment. En particulier, à chaque fois que l'on a besoin de libérer de la place pour mettre en cache une nouvelle valeur, on supprime la donnée ayant été le moins récemment utilisée.

Pour effectuer cela en temps constant, on utilise de façon conjointe une liste doublement chaînée qui stocke les données par ordre de dernière date d'utilisation (la donnée utilisée le plus récemment se trouvant en tête, celle utilisée le moins récemment en queue) et une table de hachage qui permet de trouver en temps constant si une valeur est présente en cache et, le cas échéant, de trouver le chaînon correspondant dans la liste.

Nous allons commencer par implémenter en OCaml la structure de liste doublement chaînée avec quelques fonctions nécessaires pour la suite. Puis, nous implémenterons une structure de dictionnaire basée sur un cache LRU, en interfaçant une liste doublement chaînée avec une table de hachage.

1.1 Listes doublement chaînées : **CETTE PARTIE DOIT ÊTRE TRAITÉE AVANT LA SÉANCE DE TP**

On représentera les listes chainées en OCaml avec les types suivants :

```
type 'a cell = {
   mutable data : 'a;
   mutable prev : 'a cell option;
   mutable next : 'a cell option;
}

type 'a chain = {
   mutable first : 'a cell option;
   mutable last : 'a cell option;
}
```

On rappelle la définition du type `option` présent dans la bibliothèque standard, qui permet de distinguer l'absence de valeur (à l'aide du constructeur `None`) et la présence d'une valeur `v` (sous la forme de `Some v`) :

```
type 'a option = None | Some of 'a
```

Ainsi, une chaîne vide sera représentée par :

```
{first = None; last=None}
```

▷ **Question 1.** Implémenter les fonctions de base permettant de manipuler des listes doublement chaînées. On demande d'écrire les fonctions de prototypes suivants (la fonction d'insertion insère en tête une cellule passée en argument et la fonction de suppression prend en entrée la cellule à supprimer) :

```
init_chain : unit-> 'a chain

insert : 'a chain -> 'a cell -> unit

remove: 'a chain->'a cell->unit
```

◁

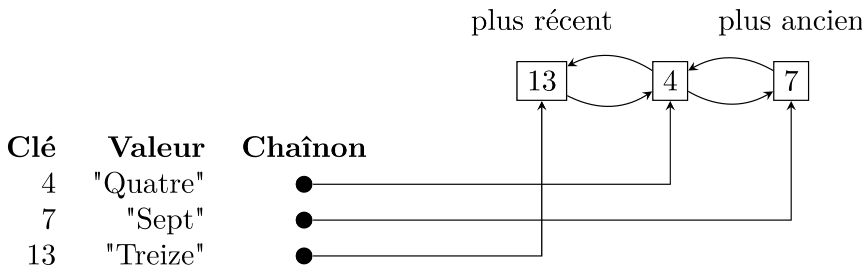
Avant de passer à la partie suivante, vous testerez vos fonctions avec le contenu du fichier `test_tp1.ml`.

1.2 Cache LRU

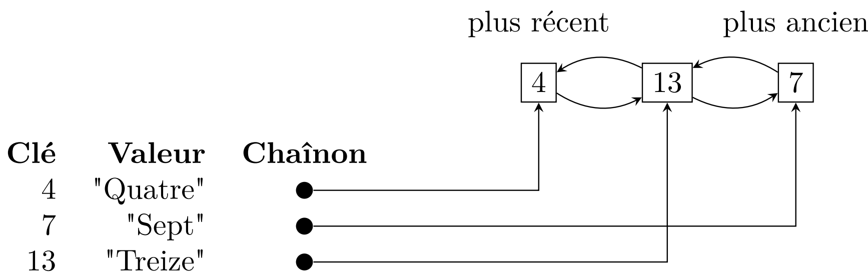
Un cache LRU peut être obtenu en combinant deux structures de données : d'une part une liste doublement chaînée indiquant pour chaque clé présente en cache son classement selon sa date d'accès la plus récente, classement indiqué par la position dans la chaîne, et une table de hachage indiquant pour chaque clé présente en cache d'une part la valeur qui lui est associée et le chaînon correspondant dans la liste doublement chaînée.

L'unique opération, en plus de la création d'une nouvelle structure de cache LRU, est l'opération `get` qui retourne la valeur associée à la clé passée en argument. Si le couple clé/valeur n'est pas déjà présent dans le cache, la valeur associée à la clé est calculée à l'aide d'une fonction d'association, qui est connue du cache et potentiellement coûteuse. Dans tous les cas, à la fin, ce couple clé/valeur est présent dans le cache, et la clé est positionnée comme étant la plus récente.

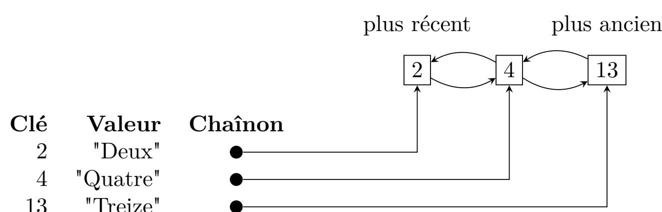
Voici un exemple de cache de taille 3, contenant les clés 4, 7, et 13 :



Supposons que l'on accède à la valeur 4 à l'aide d'un `get`. Cet élément est maintenant celui accédé le plus récemment, et l'ordre dans la liste doublement chaînée est alors :



Supposons maintenant que l'on effectue un `get` sur la clé 2. Comme celle-ci n'est pas présente, un appel va être effectué à la fonction d'association pour la calculer. De plus, le cache étant plein (on l'a supposé de capacité 3), il faut supprimer une valeur du cache pour pouvoir insérer 2. Il s'agit de 7, la clé dont le dernier accès est le plus ancien, qui est supprimée. Le cache devient alors :



Pour représenter un cache LRU, nous utiliserons le type suivant :

```
type ('a,'b) lru = {
   capacity : int;
   chain : 'a chain;
   compute : 'a->'b;
   table : ('a, 'b*'a cell) Hashtbl.t;
}
```

La fonction `compute` permet de calculer la valeur associée à une clé qui n'est pas présente dans le cache alors que la table de hachage appelée `table` permet de récupérer la valeur et la cellule associées à une clé présente dans le cache.

▷ **Question 2.** Ecrire une fonction une fonction d'initialisation qui prend en entrée la capacité du cache, et une fonction d'association de signature :

```
initialize_lru : int-> ('a->'b) -> ('a,'b) lru
```

◁

▷ **Question 3.** Ecrire un fonction `get : ('a,'b) lru-> 'a -> 'b` qui renvoie la valeur associée à la clé passée en argument et qui met à jour le cache LRU comme décrit précédemment. ◁

Avant de passer à la partie suivante, vous testerez vos fonctions avec le contenu du fichier `test_tp1.ml`.

Voici un petit memo sur les fonctions utiles de la librairie `Hashtbl` qui manipule des objets de type `Hashtbl.t`.

1. `val create : int -> ('a, 'b) t Hashtbl.create n` permet de créer une nouvelle table de hachage vide, de taille initiale `n`.
2. `val clear : ('a, 'b) t -> unit`. Cette fonction permet de vider une table de hachage.
3. `val add : ('a, 'b) t -> 'a -> 'b -> unit`.
   `Hashtbl.add tbl key data` ajoute le couple key/data dans la table `tbl`. Si un couple de clé `(key,_)` y figurait déjà, celui-ci n'est pas supprimé mais est caché, et redeviendra visible après avoir exécuté `Hashtbl.remove tbl key`.
4. `val find : ('a, 'b) t -> 'a -> 'b`.
   `Hashtbl.find tbl x` retourne la valeur associée à la clé `x` dans `tbl`, ou lève l'exception `Not_found` si `x` n'est associé à aucune valeur.
5. `val find_opt : ('a, 'b) t -> 'a -> 'b option`.
   `Hashtbl.find_opt tbl x` renvoie `Some val` si `val` est la valeur associée à `x`, et `None` en l'absence de valeur associée.
6. `val mem : ('a, 'b) t -> 'a -> bool`.
   Indique si une valeur est associée à `x` dans `tbl`.
7. `val remove : ('a, 'b) t -> 'a -> unit`.
   Supprime la dernière association de `x`, en rendant éventuellement visible l'association précédente.
8. `val length : ('a, 'b) t -> int`.
   Indique le nombre d'associations présentes dans `tbl`.

2 Tables de hachage en C.

Nous allons maintenant étudier la réalisation d'une version simplifiée de table de hachage, dans laquelle on ne stocke que des clés et pas de valeurs (on représente donc un ensemble et non un dictionnaire) et que nous utiliserons exclusivement pour stocker des chaînes de caractères (dans la représentation usuelle du langage C, c'est-à-dire se terminant par le caractère nul).

Voici le principe de notre table de hachage simplifiée : nous allons utiliser un tableau pour stocker des clés, l'emplacement de la clé étant obtenu à partir de sa valeur de hachage, calculée à l'aide d'une fonction de hachage qui transforme une clé en un nombre d'apparence aléatoire, et qui doit toujours associer le même nombre à une même clé.

Plus précisément, pour insérer une clé dans la table de hachage, nous allons lui appliquer la fonction de hachage et calculer son reste modulo la taille de la table. Comme fonction de hachage sur les chaînes de caractères, nous utiliserons la fonction de Fowler-Noll-Vo dont le code figure dans le fichier `hashtable.c`.

Voici un exemple avec la fonction de hachage FNV et un tableau de taille 16 pour différentes clés :

clé	hash	hash modulo 16
bar	16101355973854746	10
bazz	11123581685902069096	8
bob	21748447695211092	4
buzz	18414333339470238796	12
foo	15902901984413996407	7
jane	10985288698319103569	1
x	12638214688346347271	7

On remarque que les clés foo et x donnent la même clé de hachage modulo 16. C'est pourquoi il faut avoir un mécanisme de gestion des collisions.

Nous allons utiliser le mécanisme du sondage linéaire : si nous essayons d'insérer un élément à un indice déjà utilisé, nous passons simplement à l'emplacement suivant, jusqu'à trouver un emplacement libre. Si durant cette procédure la fin du tableau est atteint, on recommence au début.

Pour trouver une valeur, on part de sa clé de hachage modulo la taille du tableau, et on teste toutes les cases jusqu'à trouver cette valeur (recherche fructueuse) ou jusqu'à arriver à un emplacement vide (recherche infructueuse).

Notons que cette approche suppose qu'à tout moment, il existe au moins un emplacement vide dans le tableau.

Voici l'état de la table de hachage où l'on a inséré les valeurs dans l'ordre du tableau précédent :

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
clé		jane			bob			foo	bazz	x	bar		buzz			

On utilisera l type suivant pour travailler sur nos tables de hachage :

```
struct hashtbl {
   char** table;
   int capacity;};

 typedef struct hashtbl hashtbl;
```

▷ **Question 4.** Implémenter une table de hachage respectant la structure présentée plus haut avec les fonctions suivantes :

1. création d'une nouvelle table : `hashtbl* create_ht(int capacity)`.
2. suppression d'une table existante : `void destroy_ht(hashtbl* t)`.
3. insertion d'une nouvelle clé : `void insert_ht(hashtbl* t, char* s)`.
4. test de la présence d'une clé dans la table : `bool appar_ht(hashtbl* t, char* s)`.

Dans cette première version, nous ne traiterons pas le cas où la table de hachage est pleine. Nous supposerons donc qu'au moins une case est inoccupée.
◁

Nous avons vu plus haut que pour fonctionner, une table de hachage par sondage linéaire doit comporter au moins un emplacement vide. De plus, pour que les manipulations de la table (ajout et test de présence) se fassent en temps constant (en moyenne), il faut que le facteur de charge (càd le nombre de cases occupées divisé par la taille de la table), soit majoré par un facteur strictement inférieur à 1 fixé.

Dans notre approche simplifiée, nous utiliserons un facteur de charge maximal de 0.5. Autrement dit, dès que le tableau actuel est à moitié rempli, nous allouerons et utiliserons pour stocker les données un nouveau tableau de taille double par rapport au précédent.

▷ **Question 5.** Intégrer la gestion de l'allocation dynamique de mémoire à votre table de hachage. ◁

L’autre opération importante à implémenter pour la manipulation d’une table de hachage est la suppression de clé. Dans le cas de la représentation par sondage linéaire, la principale difficulté découle du fait qu’il faut s’assurer que la suppression d’une valeur ne conduit pas à ce que d’autres valeurs encore présentes dans la table deviennent introuvables.

▷ **Question 6.** Implémenter la suppression de clés. ◁

Pour tester vos fonctions, vous pouvez utiliser les tests écrits dans le fichier `hashtable.c`.