Travaux pratiques de programmation

On s'attend à ce que les candidats traitent de manière au moins partielle chacune des parties du sujet : programmation en OCaml, programmation en C et audit de code.

1 Programmation en C et OCaml

1.1 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 aussi 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 élevé).

Un cache ayant une capacité limitée, il est nécessaire d'avoir un mécanisme pour décider quelles valeurs garder en mémoire. Parmi les différentes 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 cache LRU, en combinant une liste doublement chaînée avec une table de hachage.

1.1.1 Listes doublement chaînées

On suggère de représenter les listes doublement chaînées en OCaml à l'aide des types suivants (disponible dans `~/lru.ml`) :

```
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 entre 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 la manipulation de listes doublement chaînées. On attend spécifiquement des fonctions permettant :

— l'ajout en tête d'un chaînon ;

— la suppression d'un chaînon quelconque.

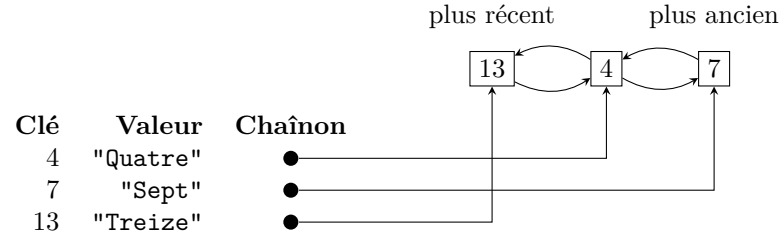
Vous pourrez ajouter toute autre fonction vous semblant appropriée.

1.1.2 Cache LRU

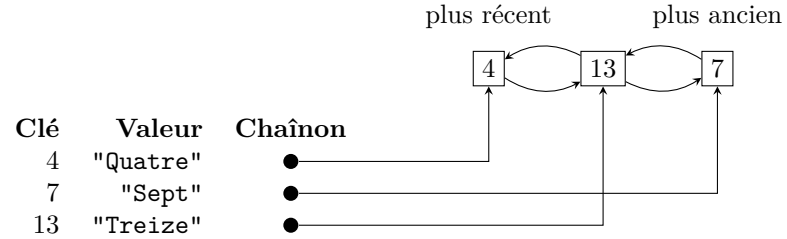
Un cache LRU peut être obtenu en combinant deux structures de données : d'une part une liste doublement chaînée indiquant pour chaque clé présente en cache son classement selon sa date d'accès la plus récente, classement indiqué par la position dans la chaîne et une table de hachage indiquant pour chaque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

En plus de la création d'une nouvelle structure de cache LRU, on a besoin de l'opération `get` qui renvoie 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.

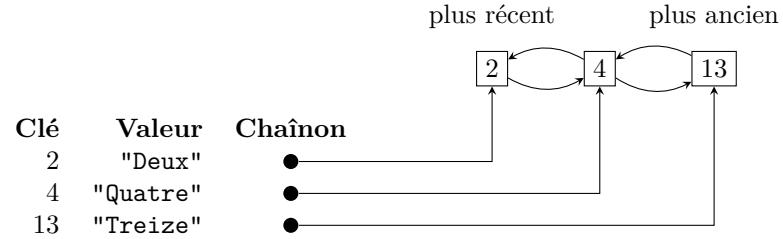
Voici un exemple de cache de taille 3, contenant les clés 4, 7 et 13 :



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 :



Question 2. Définir une structure de données implémentant un cache LRU, correspondant à un type `('a, 'b) lru` (pour associer à des clés de type `'a` des valeurs de type `'b`) et disposant, au minimum, des fonctions suivantes :

— une fonction d'initialisation prenant en argument la capacité n du cache LRU ainsi qu'une fonction d'association de type `'a -> 'b` permettant de déterminer la valeur associée à une clé ;
— une fonction `get` qui renvoie la valeur associée à la clé passée en argument et qui met à jour le cache LRU de la manière décrite précédemment.

Une présentation rapide des fonctions permettant la manipulation de tables de hachage est présentée en section 3.

1.2 Table 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). Le code ci-dessous se trouve dans `~/hashtable.c`.

```c
#define FNV_OFFSET 14695981039346656037UL
#define FNV_PRIME 1099511628211UL

// Return 64-bit FNV-1a hash for key (NUL-terminated).
static uint64_t hash_key(const char *key)
{
    uint64_t hash = FNV_OFFSET;
    for (const char *p = key; *p != 0; p++)
    {
        hash ^= (uint64_t)(unsigned char)(*p);
        hash *= FNV_PRIME;
    }
    return hash;
}
```

FIGURE 1 – Fonction de hachage FNV-1a

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 définie dans la figure 1 et dont le code figure dans le fichier `hashtable.c`. Voici dans la figure 2 un exemple avec la fonction de hashage FNV-1a 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

FIGURE 2 – Les valeurs de hachage pour différents clés

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 atteinte, 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.

Dans la figure 3 est représenté l'état de la table de hachage où l'on a inséré les valeurs dans l'ordre de la figure 2 :

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`			

FIGURE 3 – La table de hachage

Question 3. Implémenter une table de hachage respectant la structure présentée plus haut avec les fonctionnalités suivantes :
- création d'une nouvelle table d'une capacité donnée en argument,
- suppression d'une table existante,
- insertion d'une nouvelle clé et
- test de la présence d'une clé dans la table.

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.

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*, 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 4. 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.

Ainsi, dans l'exemple précédent, si l'on supprime la clé `bazz`, alors la clé `x` (de hash modulo 16 égal à 7) deviendrait introuvable sans réorganisation.

Question 5. Implémenter la suppression de clés.

2 Audit de code

Dans cette partie de l'épreuve, il est demandé au candidat d'étudier un fichier source qui comporte des erreurs ou des maladresses, de qualité de code ou de fonctionnement et il est demandé d'auditer ce fichier, c'est-à-dire :
- de comprendre et d'être capable d'expliquer le fonctionnement du code à l'oral ;
- de proposer des corrections en réécrivant certaines parties afin de corriger les erreurs ou maladresses éventuelles et de rendre le code plus clair, notamment dans une optique pédagogique ;
- de proposer des améliorations de la complexité en temps ou en mémoire, ou de la sûreté du code.

Le temps indicatif de préparation de cette partie est d'une heure.

2.1 Code en C

Le code ci-dessous est disponible dans `~/audit_c.c`.

```c
#include <stdlib.h>
#include <assert.h>

typedef struct truc
{
    int x;
    struct truc *a, *b;
} t;

int estla(t *v, int n)
{
    if (v->x == n)
        return 1;
    if (n > v->x)
        return estla(v->a, n);
    else
        return estla(v->b, n);
    if (v == NULL)
        return 0;
}

t *nouveau(int x)
{
    t *p;
    p = malloc(sizeof(t));
    p->x = x;

    return p;
}

t *enplus(t *m, int x)
{
    if (estla(m, x) == 0)
    {
        if (m == NULL)
        {
            t *p;
            p = malloc(sizeof(t));
            p->x = x;
            return p;
        }
        else if (x > m->x)
            m->a = enplus(m->a, x);
        else
            m->b = enplus(m->b, x);
    }
    return m;
}

int main()
{
```

```
    t *b;
    b = nouveau(20);
    enplus(b, 5);
    enplus(b, 1);
    enplus(b, 15);
    enplus(b, 9);
    enplus(b, 7);
    enplus(b, 12);
    assert(estla(b, 5) == 1);
    assert(estla(b, 6) == 0);
    return 0;
}
```

3 Module `Hashtbl`

On rappelle ici quelques fonctions utiles pour la manipulation de tables de hachage en OCaml. Les tables de hachages qui à des clés de type `'a` associent des valeurs de type `'b` sont modélisées par des éléments de type `('a, 'b) Hashtbl.t`.

- `val create : int -> ('a, 'b) t`
  `Hashtbl.create n` permet de créer une nouvelle table de hachage vide, de taille initiale n.
- `val clear : ('a, 'b) t -> unit`
  Cette fonction permet de vider une table de hachage.
- `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`.
- `val find : ('a, 'b) t -> 'a -> 'b`
  `Hashtbl.find tbl x` renvoie la valeur associée à la clé `x` dans `tbl`, ou lève l'exception `Not_found` si `x` n'est associé à aucune valeur.
- `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.
- `val mem : ('a, 'b) t -> 'a -> bool`
  Indique si une valeur est associée à `x` dans `tbl`.
- `val remove : ('a, 'b) t -> 'a -> unit`
  Supprime la dernière association de `x`, en rendant éventuellement visible l'association précédente.
- `val length : ('a, 'b) t -> int`
  Indique le nombre d'associations présentes dans `tbl`.
- `val iter : ('a -> 'b -> unit) -> ('a, 'b) t -> unit`
  `Hashtbl.iter f tbl` applique `f` pour toutes les associations présentes dans `tbl`.