TP n° 36 : Adressage ouvert

Le but de ce TP est de construire une implémentation efficace de la structure de donnée impérative « ensemble » (*set* en anglais). Elle permet de manipuler des ensembles de valeurs de type `T`. La spécification d'une telle structure est donnée ci-dessous.

C

```c
set *set_new(void);
bool set_is_member(set *s, T x);
void set_add(set *s, T x);
void set_remove(set *s, T x);
void set_delete(set *s);
```

Ces fonctions nous permettent de créer un ensemble vide, de savoir si un élément x fait partie de l'ensemble s, d'ajouter ou d'enlever un élément à notre ensemble et enfin de libérer la mémoire utilisée par s. On peut imaginer l'utilisation d'une telle structure pour gérer l'ensemble des adresses IP bannies d'un réseau pour des raisons de sécurité. Les adresses IP étant codées sur 32 bits, dans la suite de ce TP, nous utiliserons pour `T` le type `uint32_t`, le symbole `T` étant simplement utilisé comme un raccourci.

C

```c
typedef uint32_t T;
```

Une approche naïve consiste à implémenter une telle structure à l'aide d'un tableau dynamique contenant l'ensemble des valeurs. Si un tel ensemble contient n éléments, la fonction `set_is_member` devra effectuer une recherche linéaire et sa complexité dans le pire des cas sera en $O(n)$, ce qui n'est pas efficace. Si on choisit de maintenir ce tableau trié dans l'ordre croissant, une recherche dichotomique permet d'implémenter `set_is_member` avec une complexité en $O(\log n)$ mais c'est l'insertion qui devient alors en $O(n)$, alors qu'elle était en $O(1)$ amortie dans le cas précédent.

Nous allons utiliser une implémentation utilisant une *table de hachage en adressage ouvert*. Cette structure fonctionne à l'aide d'un tableau de taille 2^p (où $p \in [\![1, 63]\!]$ est amené à évoluer au cours de la durée de vie de la structure) ainsi qu'une fonction de signature

C
```c
uint64_t hash(T x, int p);
```

appelée *fonction de hachage*. À tout élément x de type `T` et tout entier $p \in \mathbb{N}$, elle associe un indice de tableau $i \in [\![0, 2^p[\![$ dans lequel nous souhaitons placer l'élément x. La fonction de hachage la plus simple à notre disposition est définie par

$$\forall x \in \mathbb{Z}, \quad \text{hash}_p(x) = x \bmod 2^p.$$

C'est celle que nous utiliserons dans la première partie de ce TP. Si $p = 2$, en partant d'une table de hachage vide, l'ajout des valeurs $x = 1492$ et $x = 1515$ dont les hachages respectifs sont $\text{hash}_2(1492) = 0$ et $\text{hash}_2(1515) = 3$, aboutira au tableau suivant :

En suivant, cette stratégie, il est alors facile de voir que 1515 est présent dans le tableau : il suffit de calculer son hachage $\text{hash}_2(1515) = 3$ et de constater que l'élément 1515 est bien dans la case d'indice 3.

Le rôle d'une bonne fonction de hachage est de répartir le plus uniformément possible les éléments de type `T` dans les différentes cases du tableau. Malheureusement, il est possible que des valeurs x et y soient différentes tout en ayant $\text{hash}_p(x) = \text{hash}_p(y)$; on parle alors de *collision*. Imaginons un instant que l'élément x ait déjà été placé dans le tableau à la case $\text{hash}_p(x)$. Il nous est alors impossible de placer y dans la même case. La stratégie d'une table de hachage en « adressage ouvert » consiste à le placer dans une case adjacente en suivant la stratégie décrite dans le paragraphe suivant.

Tout d'abord, afin de savoir si une case du tableau est vide ou occupée, nous allons les marquer d'une couleur. Sur nos schémas, les cases seront par défaut de couleur verte pour signifier qu'elles sont « libres ». L'ajout et la recherche d'un élément se passe alors de la manière suivante.

Ajout d'un élément L'orsque l'on souhaite ajouter l'élément x dans notre tableau, on commence par calculer son hachage $i := \text{hash}_p(x)$.

- Si la case d'indice i est libre, on y stocke l'élément x et on la colorie en mauve pour signifier qu'elle est « occupée ».
- Si cette case n'est pas libre, nous allons tester successivement les cases d'indices $i + 1 \bmod 2^p$, $i + 2 \bmod 2^p$, $i + 3 \bmod 2^p$, ... jusqu'à trouver une case qui est libre ; on parle de *sondage linéaire*. Dès qu'une telle case est trouvée, on y place l'élément x et on la colorie en mauve pour signifier qu'elle est « occupée ».

Bien entendu, un adressage ouvert suppose que le nombre d'éléments présents dans le tableau est toujours inférieur ou égal à 2^p. Pour simplifier la recherche, on imposera de plus que le tableau contienne toujours au moins une case « libre ». Lorsque l'occupation du tableau deviendra trop grande, il sera nécessaire de le redimensionner ; sa taille sera alors doublée.

Recherche d'un élément Lorsqu'on cherche la présence d'un élément x, il suffit de calculer son hachage $i := \text{hash}_p(x)$ et de chercher, par sondage linéaire, la présence de x à partir de l'indice i.

- Si au cours de la recherche, on trouve une case « libre », c'est que l'élément n'est pas présent.
- Si la recherche passe par une case « occupée » contenant l'élément x, c'est qu'il est présent dans la table.

Afin de mieux comprendre notre stratégie, en partant d'un tableau de taille 4 initialement vide, après les opérations :

→ ajout de l'élément 1492 dont le hachage est $\text{hash}_2(1492) = 0$,

→ ajout de l'élément 1515 dont le hachage est $\text{hash}_2(1515) = 3$,

→ ajout de l'élément 1939 dont le hachage est $\text{hash}_2(1939) = 3$,

voici l'état de notre table de hachage.

Les éléments 1492, 1515 sont tout d'abord placés dans les cases vides données par leur hachage. L'élément 1939 a pour hachage 3. Puisque cette case est déjà « occupée », on va sonder les cases suivantes de manière circulaire jusqu'à en trouver une de « libre ». C'est la case d'indice $j := 1$ qui est trouvée.

Pour prendre en compte la couleur de chacune de nos cases, nous allons utiliser un « statut » qui peut prendre deux valeurs :

- `empty = 0`, pour signifier que la case est « libre ».
- `occupied = 1`, pour signifier que la case est « occupée ».

Notre table sera donc formée d'un tableau de « sceaux » (*bucket* en anglais), chacun composé d'un statut et d'un élément. Nous utiliserons donc les structures suivantes

C

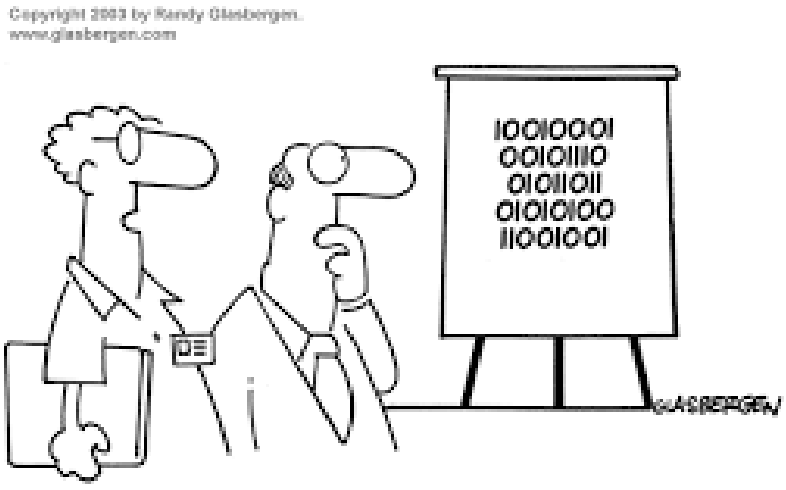
```c
const uint8_t empty = 0;
const uint8_t occupied = 1;

struct bucket {
    uint8_t status;
    T element;
};
typedef struct bucket bucket;

struct set {
    int p;
    bucket *a;
    uint64_t nb_empty;
};
typedef struct set set;
```

où $p \in [\![1, 63]\!]$ et le tableau `buckets` est de taille 2^p. L'entier `nb_empty` contiendra le nombre de cases « libres » du tableau.

1 Opérations bit à bit (ou pas !)



"We've devised a new security encryption code. Each digit is printed upside down."

Afin de proposer une implémentation efficace, on peut utiliser les opérations bit à bit disponibles en C, qui ne sont cependant pas explicitement au programme et qui compliquent un peu les choses. Deux possibilités vous sont donc offertes en fonction de vos envies. Si vous êtes plutôt du genre à tracer, rapides et efficaces, et que vous vous identifiez à un lièvre (🐇), vous pouvez lire la section suivante et traiter le sujet avec des opérations bit à bit (c'est plus difficile). Inversement, patients et minutieux, si vous préférez asseoir solidement les fondamentaux, telle une petite fourmi (🐜), la section suivant n'est pas faite pour vous.

1.1 Lièvres 🐇 seulement

On pourra consulter rapidement le lien suivant :

https://en.wikipedia.org/wiki/Bitwise_operations_in_C

1.2 Fourmis 🐜 et lièvres 🐇

Question 1

Écrire une fonction `uint64_t p2(int p)` qui renvoie 2^p. Pour les lièvres (🐇) : utiliser un décalage de bits pour faire cela efficacement et prendre garde au fait que `1` est une constante littéral entière qui est interprétée comme un `int` et pas comme un `uint64_t`. De plus, pour être efficace, il vaut mieux éviter les appels de fonctions et donc réécrire directement cet opération à chaque fois (ou bien marquer la fonction `inline`).

Question 2

Écrire la fonction de prototype `uint64_t hash(T x, int p)` implémentant le hachage $\text{hash}_p(x) := x \bmod 2^p$. Pour les lièvres (🐇) il faut faire cela sans utiliser l'opérateur `%` mais uniquement des opérations bit à bit.

Dans la suite du sujet, la distinction entre fourmis (🐜) et lièvres (🐇) ne sera pas systématiquement indiquée. Si vous êtes un lièvre (🐇) il faut utiliser systématiquement les opérations bit à bit.

2 Constructeur, destructeur et recherche d'éléments

Question 3

Écrire une fonction auxiliaire `set *empty_table(int p)` créant une table de hachage de taille 2^p et dont toutes les cases sont « libres ».

Question 4

En déduire la fonction `set *set_new(void)` créant une table de hachage pour laquelle $p = 1$ et dont toutes les cases sont « libres ».

Question 5

Afin de pouvoir tester plus facilement nos prochaines fonctions, écrire directement une fonction `set *set_example(void)` générant artificiellement une table de hachage pour laquelle $p = 2$ et contenant les dates 1492, 1515 et 1939 comme décrit dans l'exemple plus haut.

Question 6

Écrire la fonction `void set_delete(set *s)` permettant de libérer la mémoire utilisée par `s`.

Afin de préparer la possibilité d'ajouter des éléments à notre table, nous allons factoriser un peu le code et écrire une fonction auxiliaire

C
```c
uint64_t search(set *s, T x, bool *found);
```

qui renvoie un entier i et qui écrit un booléen dans `*found`. Ces valeurs doivent posséder les caractéristiques suivantes :

- Si x est un élément de s, l'entier i renvoyé est l'indice de la case contenant x et `*found` est égal à `true`.
- Sinon, on calcule l'indice i de la case dans laquelle on placerait x si on avait à l'ajouter à s. On renvoie alors i et `*found` est égal à `false`.

Question 7

Écrire cette fonction `search`. Pour les lièvres (🐇) : afin de proposer une implémentation efficace, on utilisera les opérations bit à bit pour les calculs modulo 2^p.

Question 8

Justifier que cette fonction termine.

Question 9

Écrire la fonction `bool set_is_member(set *s, T x)` permettant de déterminer si x est un élément de s.

3 Itérateur sur la table

Afin de parcourir la table, nous allons utiliser un *itérateur* qui va prendre successivement les indices des cases « occupées » de notre tableau. Il commencera à l'indice i_{begin} qui est égal, soit au plus petit index i tel que `buckets[i]` est occupé (si une telle valeur existe), soit à 2^p. Il terminera par la valeur i_{end} := 2^p. Si i est l'indice d'une case occupée, la fonction `set_next(s, i)` renverra, soit l'indice de la prochaine case occupée, si une telle case existe, soit 2^p.

C

```c
uint64_t set_begin(set *s);
uint64_t set_end(set *s);
uint64_t set_next(set *s, uint64_t i);
```

Nous utiliserons également une fonction `set_get(s, i)` qui renvoie l'élément contenu dans la case d'index i (i désignant bien évidemment un index de case occupée).

C

```c
T set_get(set *s, uint64_t i);
```

Ces fonctions permettent de parcourir efficacement l'ensemble des valeurs de notre table de hachage. Par exemple, la fonction suivante devra permettre de savoir si tous les éléments de la table sont pairs.

C

```c
bool all_even(set *s) {
  uint64_t i = set_begin(s);
  while (i != set_end(s)) {
    if (set_get(s, i) % 2 == 1) {
      return false;
    }
    i = set_next(s, i);
  }
  return true;
}
```

Question 10

Écrire la fonction `T set_get(set *s, uint64_t i)`. On suppose comme précondition que l'indice i est valable et que la case d'indice i est occupée.

Question 11

Écrire les fonctions `set_begin`, `set_end` et `set_next`.

4 Ajout d'éléments

Question 12

Écrire une fonction auxiliaire[a] `void add_no_resize(set *s, T x)` qui ajoute un élément dans la table, si celui-ci n'y était pas déjà, sans se préocuper de redimensionner la table.

a. Qui comme toutes les fonctions auxiliaires ne sera sera pas partagée avec les utilisateurs. Seules les fonctions qui commencent par `set_` vont être publiques. Nous verrons prochainement comment réaliser cette restriction.

Question 13

Écrire une fonction auxiliaire `void resize(set *s, int p)` prenant en entrée une table de hachage possédant n éléments et « redimensionnant » son tableau en un tableau de taille 2^p. On supposera que $n < 2^p$ et on placera tous les éléments dans ce nouveau tableau en utilisant la fonction de hachage hash_p associée à cette nouvelle taille de tableau. *Indication : on pourra commencer par créer une nouvelle table de hachage temporaire, avant de modifier s. On pourra également utiliser l'itérateur de la section précédente pour parcourir les éléments.*

Question 14

Écrire la fonction `void set_add(set *s, T x)` ajoutant l'élément x à la table s, si celui-ci n'y est pas déjà. Afin de toujours conserver une case « libre » dans notre tableau et de ne pas trop le charger, on décidera de doubler la taille du tableau dès que le nombre de cases « libres » est inférieur au tiers de la taille du tableau.

5 Suppression d'éléments

On souhaite désormais pouvoir supprimer des éléments de notre table de hachage.

Question 15

Expliquer pourquoi il n'est pas possible de supprimer un élément de la table en changeant simplement le statut de sa case en case « libre ».

Afin de remédier à ce problème, nous allons créer un nouveau statut appelé « pierre tombale » (*tombstone* en anglais). Lorsqu'un élément de la table sera supprimé, le statut de sa case passera d' « occupé » à celui de « pierre tombale ». Lors du sondage intervenant dans la *recherche* d'un élément, il faudra considérer les pierres tombales comme des cases ne contenant aucun élément mais signalant que la recherche doit continuer. En revanche, si nous cherchons une case pour y *placer* un nouvel élément, il faudra déterminer la première case qui est soit libre, soit une pierre tombale. On ajoute donc le statut

C

```c
const uint8_t tombstone = 2;
```

que nous représenterons sur nos schémas par le gris.

Par exemple, en partant d'un tableau de taille 4 ayant toutes ses cases « libres », après les opérations suivantes

→ ajout de l'élément 1492 dont le hachage est $\text{hash}_2(1492) = 0$,

→ ajout de l'élément 1515 dont le hachage est $\text{hash}_2(1515) = 3$,

→ ajout de l'élément 1939 dont le hachage est $\text{hash}_2(1939) = 3$,

→ suppression de l'élément 1492.

on obtient l'état suivant pour la notre table de hachage :

Question 16

Modifier (éventuellement) vos implémentations des fonctions `set_begin` et `set_next` pour que celles-ci fonctionnent bien avec les pierres tombales.

Question 17

Modifier votre implémentation de la fonction `search` pour que celle-ci fonctionne désormais avec les pierres tombales. On rappelle que dans le cas où l'élément n'est pas trouvé, il faut renvoyer l'indice de la case, « libre » ou « pierre tombale » dans laquelle on placerait x, qui est la toute première rencontrée.

Question 18

Que faut-il modifier dans la fonction `add_no_resize`

Question 19

Implémenter la fonction `void set_remove(set *s, T x)` permettant d'enlever l'élément x de la table s. Si l'élément n'est pas présent, cette fonction n'a pas d'effet. On ne cherchera pas à réduire la taille de la table.

6 Liste d'adresses IP

Dans un réseau, chaque ordinateur possède une unique adresse nommée *adresse* IP. Le standard IPv4 définit une telle adresse comme un entier non signé 32 bits, généralement représenté sous forme de sa décomposition en base 256 où les « chiffres » sont séparés par des points : $d_3.d_2.d_1.d_0$ où $d_k \in [\![0;256[\![$. Par exemple, le site `www.data.gouv.fr` est hébergé à l'adresse IP `37.59.183.93`. Le fichier `ip.txt` contient une liste de 172 754 adresses IP que nous souhaitons charger dans notre notre table de hachage.

Question 20

Écrire une fonction `T *read_data(char *filename, int *n)` lisant un fichier texte d'adresses IP encodées sous la forme $d_3.d_2.d_1.d_0$ sous la forme d'un tableau d'entiers 32 bits. L'entier pointé par `n` contiendra la taille du tableau renvoyé. Si une erreur se produit, la fonction doit renvoyer un pointeur nul et un code d'erreur strictement négatif dans `*n`.

Question 21

Écrire une fonction de prototype

C
```c
void set_skip_stats(set *s, double *average, uint64_t *max);
```

calculant le nombre de moyen et le nombre maximum de sondages nécessaires pour trouver une adresse x appartenant à s. En observant le fichier des adresses IP, expliquer pourquoi ces nombres sont si élevés.

7 Une meilleure fonction de hachage

Afin de résoudre le problème soulevé dans la partie précédente, nous allons implémenter une fonction de hachage plus efficace. Pour cela, on choisit un réel $\varphi \in [0,1]$ et on définit hash_p par

$$\forall x \in \mathbb{Z} \quad \text{hash}_p(x) := \lfloor 2^p \{x\varphi\} \rfloor$$

où $\{a\} := a - \lfloor a \rfloor$ désigne la partie fractionnaire de $a \in \mathbb{R}$. Même si cette méthode fonctionne quelle que soit la valeur de φ, on peut montrer que lorsque φ est proche de

$$\frac{\sqrt{5}-1}{2} \approx 0.618034$$

la fonction hash_p va favoriser la répartition uniforme des valeurs x dans notre tableau (voir les théorèmes d'ergodicité sur l'équirépartition modulo 1). Nous utiliserons donc une approximation de $(\sqrt{5}-1)/2$ de la forme $\varphi := s/2^{64}$ où $s \in \mathbb{N}$.

Question 22

Montrer que la fonction

C
```c
uint64_t f(uint64_t x, uint64_t s) {
    return x * s;
}
```

calcule l'entier $x_s \in [\![0, 2^{64}[\![$ tel que

$$\{x\varphi\} = \frac{x_s}{2^{64}}.$$

Question 23

Montrer que la décomposition en base 2 de $\text{hash}_p(x)$ est formée des p bits de poids forts de la décomposition en base 2 de x_s.

Question 24

En déduire une fonction `uint64_t hash(uint32_t x, int p)` permettant de calculer efficacement $\text{hash}_p(x)$. On utilisera la valeur

$$s := 11\,400\,714\,819\,323\,198\,549$$

qui a été choisie pour être un nombre premier et pour que $s/2^{64}$ soit une bonne approximation de $(\sqrt{5}-1)/2$.

Question 25

Quel est le nombre moyen et le nombre maximum de sondages nécessaires pour trouver une adresse IP présente dans notre base avec cette nouvelle fonction de hachage ?

8 Sondage quadratique

Afin de faire encore baisser le nombre de sondages nécessaires pour trouver un élément dans notre table, nous allons changer la technique de sondage. Au lieu de sonder les cases d'indices $i+1 \bmod 2^p$, $i+2 \bmod 2^p$, $i+3 \bmod 2^p$, ... nous allons sonder les cases d'indices $i+1 \bmod 2^p$, $i+(1+2) \bmod 2^p$, $i+(1+2+3) \bmod 2^p$, ... afin d'éviter la formation de clusters qui ont tendance à apparaitre avec la technique de sondage linéaire. Cette méthode est appelée méthode de sondage quadratique.

Question 26

Implémenter cette nouvelle méthode et observer son influence sur les nombres de sondage à effectuer pour notre ensemble d'adresses IP.

Question 27

Prouver enfin que cette méthode de sondage est correcte, c'est-à-dire que si il existe une case libre dans notre tableau, le sondage finira par la trouver.