TD XXXI

# ADRESSAGE OUVERT

Le but de ce sujet est de construire une réalisation efficace de la structure de donnée impérative SET. Elle permet de manipuler des ensembles de valeurs de type `T`. La signature souhaitée est donnée ci-dessous :

```
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 `T = uint32_t`, le symbole `T` étant simplement utilisé dans l'énoncé comme un raccourci.

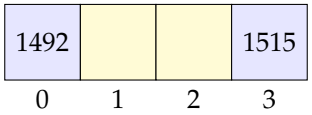
Nous allons réaliser cette signature en utilisant une *table de hachage en adressage ouvert*. Cette structure fonctionne à l'aide d'un tableau de taille $2^p$ (où $p \in [\![1 \dots 63]\!]$ est amené à évoluer au cours de la durée de vie de la structure) ainsi qu'une fonction de prototype

```
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 $i \in [\![0 \dots 2^p[\![$ dans lequel nous souhaitons placer l'élément $x$. La fonction de hachage le plus simple à notre disposition est définie par

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

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 notre 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 jaune pour signifier qu'elles sont « libres ». L'ajout et la recherche d'un élément se passent alors de la manière suivante.

**Ajout d'un élément** Lorsqu'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 bleu 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 bleu 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.

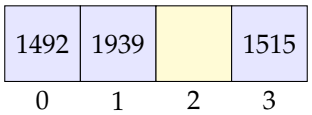
**Recherche d'un élément** Lorsqu'on cherche la présence d'un élément $x$, il suffit de calculer son hachage $i = \text{hash}_p(x)$ et de chercher, par sondage linéaire, la présence de $x$ à partir de l'indice $i$.

- 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 d'alvéoles (on parle aussi de seaux, ou *bucket* en anglais) chacune composée d'un statut et d'un élément. Nous utiliserons donc les structures suivantes :

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

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

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

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

## 1 Constructeur, destructeur et recherche d'éléments

▶ **Question 1** Quelle est l'écriture binaire de l'entier renvoyé par cette fonction ?

```
uint64_t ones(int p){
  return (1ull << p) - 1ull;
}
```

*On utilise `1ull` à la place de `1` pour que les calculs intermédiaires s'effectuent sur des entiers 64 bits non signés. Par défaut ces calculs utiliseraient des `int`, ce qui poserait problème.*

▶ **Question 2** Écrire la fonction `hash` implémentant le hachage $\text{hash}_p(x) = x \bmod 2^p$.

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

Afin de proposer une implémentation efficace, on utilisera les opérations bit à bit disponibles en C.

▶ **Question 3** Écrire la fonction `set *set_new(void)` créant une table de hachage pour laquelle $p = 1$, dont toutes les cases sont « libres ». Afin de pouvoir tester plus facilement nos prochaines fonctions, on écrira aussi 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 4** Écrire la fonction `void set_delete(set *s)` libérant la mémoire utilisée par `s`.

▶ **Question 5** Écrire la fonction `bool set_is_member(set *s, T x)` permettant de déterminer si $x$ est un élément de $s$. Afin de proposer une implémentation efficace, on utilisera les opérations bit à bit pour les calculs modulo $2^p$. On donnera de plus un argument justifiant la terminaison de cette fonction.

## 2 Parcours de la table

Pour permettre de parcourir les éléments de l'ensemble sans avoir à se préoccuper des détails d'implémentation, nous allons fournir un itérateur qui va prendre successivement les indices des cases « occupées » de notre tableau. Il commencera à l'index $i_{\text{begin}}$ qui est égal, soit au plus petit index $i$ tel que `a[i]` est occupé (si une telle valeur existe), soit à $2^p$. Il terminera par la valeur $i_{\text{end}} = 2^p$. 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 $2^p$.

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

Nous utiliserons aussi la 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).

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

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

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

▶ **Question 6** Écrire la fonction `set_get`.

▶ **Question 7** Écrire les fonctions `set_begin`, `set_end` et `set_next`.

## 3 Ajout d'éléments

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

```
uint64_t set_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'index de la case contenant $x$ et `*found` est égal à `true`;
- sinon, on calcule l'index `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 8** Écrire `set_search` et réimplémenter `set_is_member` à l'aide de cette nouvelle fonction.

▶ **Question 9** Écrire la fonction `void set_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 $\text{hash}_p$ associée à cette nouvelle taille de tableau. On pourra commencer par créer une nouvelle table de hachage, avant de modifier $s$.

▶ **Question 10** Écrire la fonction `void set_add(set *s, T x)` ajoutant l'élément $x$ à la table $s$ (la fonction n'aura aucun effet si l'élément était déjà présent). 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.

## 4 Suppression d'éléments

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

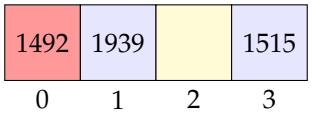
▶ **Question 11** Expliquer pourquoi il n'est pas possible de supprimer un élément de la table en changeant simplement le statut de sa case en case « libre ».

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 de « occupé » à « 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. 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 suivant (que nous représenterons en rouge sur nos schémas) :

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

Par exemple, le schéma ci-dessous donne l'état de la structure si l'on part d'un tableau de taille 4 ayant toutes ses cases « libres », et qu'on effectue 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.



▶ **Question 12** Proposer des nouvelles implémentations des fonctions `set_search`, `set_begin`, `set_next` et `set_add` fonctionnant avec les pierres tombales.

▶ **Question 13** Implémenter la fonction

```
void set_remove(set *s, T x);
```

permettant d'enlever l'élément $x$ de la table $s$. Cette fonction n'aura pas d'effet si l'élément n'était pas présent.

## 5 La liste des 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 $d_k \in [0 \dots 255]$. Le fichier `ip.txt` contient une liste de 172 754 adresses IP que nous souhaitons charger dans notre table de hachage.
Vous trouverez dans le fichier fourni une fonction

```
T *read_data(char *filename, int *n);
```

Cette fonction a le comportement suivant :

- la fonction renvoie un pointeur vers un bloc alloué sur le tas, contenant les adresses lues dans le fichier ;
- si une erreur s'est produite (si le fichier n'existe pas, par exemple), le pointeur renvoyé sera nul ;
- l'argument `n` est un argument de sortie. Après l'appel, la valeur pointée par `n` sera égale à la taille du tableau renvoyé.

▶ **Question 14** Écrire une fonction `read_set` qui prend en entrée le nom d'un fichier et renvoie un `set*` dont les éléments sont les adresses présentes dans le fichier.

```
set *read_set(char *filename);
```

▶ **Question 15** Écrire une fonction

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

calculant le nombre 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.

## 6 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 $\text{hash}_p$ par

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

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 $\varphi$, on peut montrer que lorsque $\varphi$ est proche de

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

la fonction $\text{hash}_p$ 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 16** Montrer que la fonction

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

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

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

▶ **Question 17** 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 18** En déduire une implémentation

```
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 19** 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 ?

## 7 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 $i+1 \bmod 2^p$, $i+2 \bmod 2^p$, $i+3 \bmod 2^p, \dots$ nous allons sonder les cases d'indices $i+1 \bmod 2^p$, $i+(1+2) \bmod 2^p$, $i+(1+2+3) \bmod 2^p, \dots$ 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 20** Implémenter cette nouvelle méthode et observer son influence sur les nombres de sondage à effectuer pour notre ensemble d'adresses IP.

▶ **Question 21** 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.

# Solutions

▶ **Question 1** Cette fonction renvoie le nombre $\overline{1\ldots1}^2$ (avec p chiffres 1).

▶ **Question 2** Le reste de la division modulo $2^p$ est obtenu en ne conservant que les p bits de poids faible, ce qui peut se faire ainsi :

```
uint64_t hash(uint32_t k, int p) {
    return k & ones(p);
}
```

▶ **Question 3**

```
set *set_new(void) {
    set *s = malloc(sizeof(set));
    s->p = 1;
    s->a = malloc(2 * sizeof(bucket));
    s->a[0].status = empty;
    s->a[1].status = empty;
    s->nb_empty = 2;
    return s;
}
```

```
set *set_example(void) {
    set *s = malloc(sizeof(set));
    s->p = 2;
    s->a = malloc(4 * sizeof(bucket));
    s->a[0].status = occupied;
    s->a[0].element = 1492;
    s->a[1].status = occupied;
    s->a[1].element = 1939;
    s->a[2].status = empty;
    s->a[3].status = occupied;
    s->a[3].element = 1515;
    s->nb_empty = 1;
    return s;
}
```

▶ **Question 4**

```
void set_delete(set *s) {
    free(s->a);
    free(s);
}
```

▶ **Question 5**

```
bool set_is_member(set *s, uint32_t x) {
    uint64_t i = hash(x, s->p);
    while (true) {
        if (s->a[i].status == empty) {
            return false;
        } else if (s->a[i].element == x) {
            return true;
        }
        i += 1;
        i &= ones(p);
    }
}
```

▶ **Question 6**

```
uint32_t set_get(set *s, uint64_t i) {
    return s->a[i].element;
}
```

▶ **Question 7** `set_end` est immédiat :

```
uint64_t set_end(set *s) {
    return 1ull << s->p;
}
```

Pour `set_begin`, on parcourt le tableau jusqu'à trouver une case occupée, ou en sortir :

```
uint64_t set_begin(set *s) {
    uint64_t i = 0;
    while (i < set_end(s) && s->a[i].status == empty)
        i++;
    return i;
}
```

`set_next` est similaire, sauf que l'on commence le parcours à la case $i + 1$.

```
uint64_t set_next(set *s, uint64_t i) {
    i++;
    while (i < set_end(s) && s->a[i].status == empty)
        i++;
    return i;
}
```

▶ **Question 8** On parcourt la table, à partir de l'indice $\text{hash}_p(x)$ jusqu'à tomber soit sur une case vide, soit sur une case contenant $x$. La fonction `set_is_member` ne pose ensuite aucune difficulté.

```
uint64_t set_search(set *s, uint32_t x, bool *found) {
    uint64_t i = hash(x, s->p);
    while (true) {
        if (s->a[i].status == empty) {
            *found = false;
            return i;
        } else if (s->a[i].element == x) {
            *found = true;
            return i;
        }
        i += 1;
        i &= ones(s->p);
    }
}

bool set_is_member(set *s, uint32_t x) {
    bool found;
    set_search(s, x, &found);
    return found;
}
```

▶ **Question 9** Il n'y a rien de compliqué, mais il y a plusieurs étapes et il faut faire attention aux potentielles fuites de mémoire :

```
void set_resize(set *s, int p) {
    // Prepare an empty table of size 2**p
    uint64_t m = 1ull << p;
    set *s_new = malloc(sizeof(set));
    s_new->p = p;
    s_new->a = malloc(m * sizeof(bucket));
    for (uint64_t i = 0; i < m; ++i) {
        s_new->a[i].status = empty;
    }
    s_new->nb_empty = m;
    // Add the elements of s to that table
    for (uint64_t i = set_begin(s); i != set_end(s); i = set_next(s, i)) {
        uint32_t x = set_get(s, i);
        bool found;
        uint64_t j = set_search(s_new, x, &found);
        s_new->a[j].status = occupied;
        s_new->a[j].element = x;
        s_new->nb_empty--;
    }
    free(s->a);
    // The next three lines could be replaced with *s = *s_new;
    s->a = s_new->a;
    s->p = p;
    s->nb_empty = s_new->nb_empty;

    free(s_new);
}
```

▶ **Question 10** On commence par vérifier si l'élément est déjà présent (rien à faire dans ce cas). Cela permet en même temps de savoir où il faudra insérer l'élément. Ensuite :

- si nécessaire, on redimensionne (sans oublier de recalculer ensuite le point d'insertion, qui aura changé) ;
- on effectue l'insertion.

```
void set_add(set *s, uint32_t x) {
    bool found;
    uint64_t j = set_search(s, x, &found);
    if (found) return;

    uint64_t m = 1ull << s->p;
    if (s->nb_empty <= 1 || 3 * s->nb_empty <= m) {
        set_resize(s, s->p + 1);
        j = set_search(s, x, &found);
    }

    s->a[j].status = occupied;
    s->a[j].element = x;
    s->nb_empty--;
}
```

▶ **Question 11** Si l'on supprime un élément en passant le statut de la case à « libre », on va casser les séquences de recherche. Par exemple, supposons qu'on insère successivement $x$ et $y$ dans une table vide, et que ces deux éléments soient en collision ($\text{hash}_p(x) = \text{hash}_p(y) = 0$, par exemple). On va alors placer $x$ dans la case 0 et $y$ dans la case 1. Si l'on supprime $x$ puis que l'on cherche $y$, on va tomber sur une case vide, et en déduire que $y$ n'est pas présent !

▶ **Question 12** Remarquons d'abord qu'il n'y a aucune modification à apporter à `set_end`. Pour `set_search`, on commence par calculer $i = \text{hash}_p(x)$ et l'on parcourt le tableau (circulairement) à partir de ce $i$. Simultanément, on maintient une variable `i_tombstone` qui vaut initialement `set_end(s)` (ce qui n'est jamais un indice valable du tableau). Si l'on rencontre au moins une pierre tombale lors de la recherche, `i_tombstone` deviendra égal à l'indice de la première pierre tombale rencontrée. La recherche se termine dans l'un des deux cas suivants :

- on trouve l'élément, auquel cas il faut l'indiquer dans `found` et renvoyer sa position ;
- on tombe sur une case vide, ce qui signifie que l'élément n'est pas présent. Dans ce cas, on renvoie l'indice de la première pierre tombale rencontrée s'il y en a une, l'indice de la case libre sur laquelle on a terminé sinon.

```
uint64_t set_search(set *s, uint32_t x, bool *found) {
    uint64_t i = hash(x, s->p);
    uint64_t i_tombstone = set_end(s);
    while (true) {
        if (s->a[i].status == empty) {
            *found = false;
            if (i_tombstone == set_end(s)) return i;
            else return i_tombstone;
        } else if (s->a[i].status == tombstone && i_tombstone == set_end(s)) {
            i_tombstone = i;
        } else if (s->a[i].element == x) {
            *found = true;
            return i;
        }
        i += 1;
        i &= ones(s->p);
    }
}
```

Pour `set_next`, c'est un plus simple : il suffit de modifier la fonction écrite plus haut pour sauter les cases vides *et les pierres tombales*.

```
uint64_t set_next(set *s, uint64_t i) {
    i++;
    while (i < set_end(s) && (s->a[i].status == empty || s->a[i].status == tombstone))
        i++;
    return i;
}
```

Pour `set_begin`, le plus simple est d'apporter la même modification. Cependant, il est en fait possible de voir cette fonction comme un cas particulier de `set_next` : en effet, on a `set_begin(s) == set_next(s, -1)`. Noter que le deuxième argument est non signé, donc le $-1$ est en fait implicitement converti en $2^{64}-1$ : ça ne pose pas de problème, les calculs se font modulo $2^{64}$ et on a bien `(uint64_t)-1 + (uint64_t)1 == 0`.

```
uint64_t set_begin(set *s){
  return set_next(s, -1);
}
```

▶ **Question 13** Aucune difficulté si l'on pense à utiliser `set_search` :

```c
void set_remove(set *s, uint32_t x) {
    bool found;
    uint64_t i = set_search(s, x, &found);
    if (!found) return;
    s->a[i].status = tombstone;
}
```

## 1 La liste des adresses IP

▶ **Question 14** La fonction `read_data` fournie, qui utilise certaines choses sur la gestion des fichiers que nous n'avons pas encore vues (et des détails sordides sur les conversions implicites).

```c
uint32_t *read_data(char *filename, int *n) {
    FILE *file = fopen(filename, "r");
    if (file == NULL) return NULL;

    int nb_lines = 0;
    char line[16];
    while (!feof(file)) {
        fscanf(file, "%15s\n", line);
        nb_lines++;
    }
    rewind(file);

    uint32_t *t = malloc(nb_lines * sizeof(uint32_t));
    int a, b, c, d;
    for (int i = 0; i < nb_lines; ++i) {
        int nb_read = fscanf(file, "%d.%d.%d.%d", &a, &b, &c, &d);
        if (nb_read != 4){
            fclose(file);
            free(t);
            return NULL;
        }
        t[i] = (((a * 256u) + b) * 256u + c) * 256u + d;
    }

    fclose(file);
    *n = nb_lines;
    return t;
}
```

La fonction que l'on demandait d'écrire, elle, ne pose pas trop de problème. Attention cependant à ne pas oublier de libérer le tableau !

```c
set *read_set(char *filename){
    set *s = set_new();
    int n = 0;
    uint32_t *arr = read_data(filename, &n);
    assert(arr != NULL);
    printf("read_set : n = %d\n", n);
    for (int i = 0; i < n; i++){
        set_add(s, arr[i]);
    }
    free(arr);
    return s;
}
```

▶ **Question 15**

```c
void set_skip_stats(set *s, double *average, uint64_t *max) {
    uint64_t nb_skip_total = 0;
    uint64_t nb_skip_max = 0;
    uint64_t nb_elements = 0;
    for (uint64_t i = set_begin(s); i != set_end(s); i = set_next(s, i)) {
        nb_elements++;
        uint32_t x = set_get(s, i);
        uint64_t j = hash(x, s->p);
        uint64_t nb_skip = (i - j) & ones(s->p);
        nb_skip_total += nb_skip;
        if (nb_skip > nb_skip_max) {
            nb_skip_max = nb_skip;
        }
    }
    *average = ((double)nb_skip_total) / nb_elements;
    *max = nb_skip_max;
}
```

On obtient un nombre moyen d'éléments sautés d'environ 912 et un peu plus de 19 000 comme nombre maximum. C'est catastrophique, mais ce n'est pas surprenant puisque notre fonction de hachage est profondément stupide. En effet, on se contente de prendre $x$ modulo $2^p$, ce qui revient très précisément à conserver les $p$ bits de poids faible de $x$. Ici, une grande partie des adresses (plus de la moitié, à vue d'œil) sont de la forme a.b.c.0, et se terminent donc par 8 bits nuls (il s'agit de masques de sous-réseaux). On essaie donc de les envoyer sur $1/256^{\text{ème}}$ des cases de la table, ce qui crée évidemment d'innombrables collisions.

**Remarque**

Hacher un entier en prenant simplement sa valeur modulo la taille de la table est une technique certes rustique, mais pas complètement déraisonnable. En revanche, il ne faut surtout pas prendre une table de taille $2^p$ dans ce cas (typiquement, on prend un nombre premier).

▶ **Question 16** Soit $x, s \in \mathbb{N}$. Puisque $x$ et $s$ sont des entiers non signés, $x * s$ est le reste de la division euclidienne de $xs$ par $2^{64}$. On effectue donc une telle division euclidienne : il existe $q, r \in \mathbb{N}$ tels que $xs = q2^{64} + r$ et $0 \leqslant r < 2^{64}$. On a donc :

$$\begin{aligned}
\{x\varphi\} &= \left\{\frac{xs}{2^{64}}\right\} \\
&= \left\{\frac{q2^{64} + r}{2^{64}}\right\} \\
&= \left\{q + \frac{r}{2^{64}}\right\} \\
&= \frac{r}{2^{64}} && \text{car } q \in \mathbb{N} \text{ et } 0 \leqslant r < 2^{64}
\end{aligned}$$

▶ **Question 17** On décompose $x_s$ en base 2 : $x_s = \sum_{k=0}^{63} d_k 2^k$. On a alors :

$$\begin{aligned}
\text{hash}_p(x) &= \lfloor 2^p \{x\varphi\} \rfloor \\
&= \left\lfloor 2^p \frac{x_s}{2^{64}} \right\rfloor \\
&= \left\lfloor 2^{p-64} \sum_{k=0}^{63} d_k 2^k \right\rfloor \\
&= \left\lfloor \sum_{k=0}^{63} d_k 2^{k-(64-p)} \right\rfloor \\
&= \left\lfloor \sum_{k=0}^{64-(p+1)} d_k 2^{k-(64-p)} + \sum_{k=64-p}^{63} d_k 2^{k-(64-p)} \right\rfloor \\
&= \sum_{k=64-p}^{63} d_k 2^{k-(64-p)} = \sum_{k=0}^{p-1} d_{64-p+k} 2^k
\end{aligned}$$

On en déduit que la décomposition en base 2 de $\text{hash}_p(x)$ est formé des $p$ bits de poids fort de la décomposition en base 2 de $x_s$.

▶ **Question 18** On obtient :

```
uint64_t hash(uint32_t x, int p) {
    uint64_t s = 11400714819323198549u;
    return (x * s) >> (64 - p);
}
```

En réalité, on pouvait écrire cela directement. En effet, on a vu en cours que si $n$ était un entier positif, alors `n >> k` renvoyait le quotient de la division euclidienne de $n$ par $2^k$, c'est-à-dire $\lfloor n/2^k \rfloor$. Or ici, on a précisément $\text{hash}_p(x) = \lfloor x_s/2^{64-p} \rfloor$.

▶ **Question 19** Avec cette nouvelle fonction de hachage, le nombre moyen d'étapes de sondage passe à 0.98. Le maximum de sondages passe quant à lui à 56. On a donc une chute drastique de ces valeurs par rapport à la fonction de hachage naïve utilisée dans la première partie.

## 2 Sondage quadratique

▶ **Question 20** Il faut modifier `set_search` et `set_skip_stats`. On ne donne que la nouvelle version de `set_search` :

```c
uint64_t set_search(set *s, uint32_t x, bool *found) {
    uint64_t i = hash(x, s->p);
    uint64_t i_tombstone = set_end(s);
    uint64_t i_step = 1;
    while (true) {
        if (s->a[i].status == empty) {
            *found = false;
            return i_tombstone == set_end(s) ? i : i_tombstone;
        } else if (s->a[i].status == tombstone && i_tombstone == set_end(s)) {
            i_tombstone = i;
        } else if (s->a[i].element == x) {
            *found = true;
            return i;
        }
        i += i_step;
        i_step++;
        i = i & ones(s->p);
    }
}
```

Avec cette nouvelle stratégie, le nombre moyen d'étapes de sondage passe à 0.78. Le maximum de sondages passe quant à lui à 22. On a donc une baisse de ces valeurs par rapport à la technique de sondage linéaire.

**Remarque**

En pratique, les deux méthodes sont utilisées : le sondage quadratique a tendance à donner des séquences de recherche plus courtes (comme c'est le cas ici), mais pas forcément plus rapides (pour des raisons techniques).

► **Question 21** Il reste à montrer que cette stratégie de sondage est correcte, c'est-à-dire que si le tableau possède au moins une case vide, le sondage va la trouver. À l'étape $k$, le sondage va visiter la case d'indice $i+1+\cdots+k \bmod 2^p = i+k(k+1)/2 \bmod 2^p$. Nous allons montrer que pour $0 \leqslant k < 2^p$, ces sondages se font dans des cases deux à deux distinctes. Pour prouver cela, on se donne $k_1, k_2 \in [0 \dots 2^p[$ tels que

$$\frac{k_1(k_1+1)}{2} \equiv \frac{k_2(k_2+1)}{2} \ [2^p]$$

et on souhaite montrer que $k_1 \equiv k_2 \ [2^p]$. On a donc

$$\begin{aligned} k_1(k_1+1) &\equiv k_2(k_2+1) \ [2^{p+1}] \\ \text{donc} \quad (k_1-k_2)(k_1+k_2+1) &\equiv 0 \ [2^{p+1}] \end{aligned}$$

donc $2^{p+1} | (k_1-k_2)(k_1+k_2+1)$.

- Supposons que $2 | (k_1-k_2)$. Alors $k_1$ et $k_2$ ont même parité, donc $k_1+k_2+1$ est impair, donc $k_1+k_2+1$ est premier avec $2^{p+1}$. D'après le lemme de Gauss, on en déduit que $2^{p+1} | k_1-k_2$, donc $2^p | k_1-k_2$ donc $k_1 = k_2$.
- Sinon, 2 est premier avec $k_1-k_2$, donc $2^{p+1}$ est premier avec $k_1-k_2$. D'après le lemme de Gauss, on en déduit que $2^{p+1}$ divise $k_1+k_2+1$. Or $0 \leqslant k_1 \leqslant 2^p-1$ et $0 \leqslant k_2 \leqslant 2^p-1$, donc $1 \leqslant k_1+k_2+1 \leqslant 2^{p+1}-1$. C'est absurde.

Donc $k_1 = k_2$, ce qui montre que les $2^p$ premiers sondages de la progression quadratique visitent des cases deux à deux distinctes. Comme il y en a $2^p$, elles sont toutes visitées.