Table de hachage en adressage ouvert

1 Constructeur, destructeur et recherche d'éléments

1. On utilise le fait que la décomposition binaire de $2^p - 1$ est formée de p bits égaux à 1. Comme le reste de la division euclidienne de x par 2^p est obtenue en ne gardant que les p chiffres de poids faible dans la décomposition binaire de x, on obtient le code suivant

```
uint64_t hash(uint32_t x, int p) {
    uint64_t one = 1;
    return x & ((one << p) - one);
}
```

   L'utilisation de `one` à la place de `1` permet de signaler que cet entier est de type `uint64_t` et non de type `int`. Si nous avions utilisé `1`, ce dernier aurait été considéré comme un entier signé de type `int` et l'opération `1 << p` causerait un overflow et donc un « undefined behaviour » pour $p \geqslant 31$.

2. On obtient facilement le code suivant. Il est inutile d'initialiser les valeur `s->a[k].element` puisqu'elles ne seront jamais lues.

```
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;
}
```

   Le code générant notre exemple se fait aussi facilement.

```
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;
}
```

3. On libère d'abord la mémoire associée au tableau `a`, puis la mémoire utilisée pour la structure.

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

4. On commence par calculer le hachage $i := \mathrm{hash}_p(x)$ et la valeur `mask` égale à $2^p - 1$ qui sera utilisée afin de calculer efficacement les indices modulo 2^p lors du sondage linéaire. On lance une boucle while dont on sortira par un `return`. Si à un moment, on trouve une case vide, c'est que l'élément cherché n'est pas dans la table. Si lors du sondage on trouve une case occupée par l'élément cherché, c'est qu'il est présent dans la table. Sinon, on continue notre sondage. La boucle termine car le tableau contient au moins une case libre.

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

2 Parcourt de la table

1. L'écriture de la fonction `set_get` est immédiate.

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

2. La fonction `set_begin` cherche la case « occupée » ayant le plus petit indice. Elle renvoie $m := 2^p$ si elle ne trouve aucune case « occupée ».

```
uint64_t set_begin(set *s) {
    uint64_t i = 0;
    uint64_t one = 1;
    uint64_t m = one << s->p;
    while (i < m && s->a[i].status == empty) i++;
    return i;
}
```

La fonction `set_end` est immédiate.

```
uint32_t set_end(set *s) {
    uint64_t one = 1;
    uint64_t m = one << s->p;
    return m;
}
```

Enfin, la fonction `set_next` est très similaire à la fonction `set_begin`, que l'on pourrait d'ailleurs implémenter à l'aide de cette dernière : `uint64_t set_begin(set *s) { return set_next(s, -1); }`. Notons au passage que cela ne pose aucun problème s'appeler `set_next` avec la valeur -1 bien que cette fonction attende un entier non signé. En effet, -1 est alors réduit modulo 2^{64} en $2^{64} - 1$ qui est le plus grand nombre représentable par le type `uint64_t`. Un incrément sur cette valeur la transformera donc en 0.

```
uint64_t set_next(set *s, uint64_t i) {
    i++;
    uint64_t one = 1;
    uint64_t m = one << s->p;
    while (i < m && s->a[i].status == empty) i++;
    return i;
}
```

3 Ajout d'éléments

1. La fonction `set_search` est très similaire à la fonction `set_is_member` écrite plus haut.

```
uint64_t set_search(set *s, uint32_t x, bool *found) {
    uint64_t i = hash(x, s->p);
```

```
    uint64_t one = 1;
    uint64_t mask = (one << s->p) - one;
    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 = i & mask;
    }
}
```

Cette dernière se récrit alors facilement :

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

Notons au passage que la valeur renvoyée par la fonction `set_search` est simplement ignorée.

2. On commence par créer une table de hachage vide de taille $m := 2^p$ sans oublier de mettre à jour le nombre de cases vides dans notre tableau. On effectue ensuite une boucle utilisant un itérateur sur les différents éléments de la table de hachage d'origine. Pour chacun de ces éléments, on utilise la fonction `set_search` afin de trouver l'indice du tableau de la nouvelle table dans lequel on va le placer. On n'oublie pas de libérer l'ancien tableau ainsi que la structure d'ensemble dont nous n'avons plus besoin.

```
void set_resize(set *s, int p) {
    uint64_t one = 1;
    uint64_t m = one << 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;
    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);
    *s = *s_new;
    free(s_new);
}
```

3. L'ajout d'un nouvel élément se fait maintenant de manière simple. On commence par redimensionner le tableau si le nombre de cases libre est inférieur ou égal à 1 ou si le nombre de cases libres est inférieur au tiers de la taille $m := 2^p$ du tableau. Notons au passage, que pour des raisons d'efficacité, on préfère effectuer ce test à l'aide d'une multiplication plutôt qu'avec une division. On n'oublie pas de mettre à jour le nombre de cases « libres ».

```
void set_add(set *s, uint32_t x) {
    uint64_t one = 1;
    uint64_t m = one << s->p;
    if (s->nb_empty <= 1 || 3 * s->nb_empty <= m) {
        set_resize(s, s->p + 1);
    }
```

```
    bool found;
    uint64_t j = set_search(s, x, &found);
    s->a[j].status = occupied;
    s->a[j].element = x;
    s->nb_empty--;
}
```

4 Suppression d'éléments

1. On voit bien que dans l'exemple donné ensuite dans l'énoncé, si la case de 1492 était marquée comme libre, la recherche de 1939 nous dirait que cet élément n'est pas présent dans la table. En effet, le calcul du hachage de 1939 donne 3, case déjà occupée par 1515. Or un sondage linéaire à partir de cette case rencontre la case d'indice 0 qui est libre et cela, avant de trouver la case contenant 1939.
2. On reprend le code `set_search` écrit plus haut. On utilise en plus la variable `i_tomstone` qui est initialisée à -1. Comme c'est un entier non signé 64 bits, cette valeur est réduite modulo 2^{64} à $2^{64}-1$ qui est la plus grande valeur de ce type. Elle ne peut donc pas être un indice du tableau `s->a` et elle signifie d'une certaine manière que pour le moment `i_tombstone` ne contient aucune valeur valide. On parcourt ensuite le tableau en effectuant un sondage linéaire. Si on rencontre une case « pierre tombale », il faut continuer à chercher. Notons qu'on garde dans la variable `i_tombstone` l'indice de la première « pierre tombale » rencontrée. Si à un moment on trouve une case « libre », c'est que l'élément cherché n'est pas présent dans le tableau. Si on a déjà rencontré une « pierre tombale », c'est l'indice de cette case qu'il faut renvoyer. Sinon, c'est l'indice de la case libre. Enfin, si on rencontre une case occupée par notre élément cherché, c'est qu'il est présent et on renvoie donc l'indice de cette case.

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

On commence par implémenter `set_next` qui cherche une case tant qu'elle n'est pas occupée.

```
uint64_t set_next(set *s, uint64_t i) {
    i++;
    uint64_t one = 1;
    uint64_t m = one << s->p;
    while (i < m && s->a[i].status != occupied) i++;
    return i;
}
```

Comme remarqué plus haut, la fonction `set_begin` s'exprime à l'aide de cette dernière.

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

Enfin, il faut ajuster la fonction `set_add` pour diminuer le nombre de cases « libres » uniquement dans le cas où l'élément n'est pas placé sur une « pierre tombale ».

```c
void set_add(set *s, uint32_t x) {
    uint64_t one = 1;
    uint64_t m = one << s->p;
    if (s->nb_empty <= 1 || 3 * s->nb_empty <= m) {
        set_resize(s, s->p + 1);
    }
    bool found;
    uint64_t j = set_search(s, x, &found);
    if (a[j].status == empty) {
        s->nb_empty--;
    }
    s->a[j].status = occupied;
    s->a[j].element = x;
}
```

3. On peut enfin écrire la fonction permettant le suppression d'un élément.

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

5 La liste des adresses IP

1. Voici un code qui commence par compter le nombre de lignes présents dans le fichier avant de lire les adresses IP, ligne par ligne. Afin d'éviter le calcul avec des entiers de type `int` lors du calcul de la valeur de l'adresse IP, on utilise l'entier `256u` de type `unsigned int` qui est, sur les plateformes courantes, un entier non signé codé sur 32 bits.

```c
uint32_t *read_ip(char *filename, int *n, int *error) {
    FILE *file = fopen(filename, "r");
    if (file == NULL) {
        *error = 1;
        return NULL;
    }

    int nb_lines = 0;
    char line[16];
    while (!feof(file)) {
        fscanf(file, "%s\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++) {
        fscanf(file, "%d.%d.%d.%d", &a, &b, &c, &d);
        t[i] = (((a * 256u) + b) * 256u + c) * 256u + d;
    }

    fclose(file);

    *n = nb_lines;
    *error = 0;
    return t;
}
```

2. Voici une version de la fonction calculant la valeur moyenne et le nombre maximal d'étapes du sondage lors de la recherche d'un élément appartenant à la table.

```
void set_skip_stats(set *s, double *average, uint64_t *max) {
    uint64_t nb_skip_total = 0;
    uint64_t nb_elements = 0;
    uint64_t nb_skip_max = 0;
    uint64_t one = 1;
    uint64_t mask = (one << s->p) - one;
    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 = 0;
        bool found = false;
        while (!found) {
            if (s->a[j].status == tombstone ||
                (s->a[j].status == occupied && s->a[j].element != x)) {
                nb_skip++;
            } else {
                nb_skip_total += nb_skip;
                if (nb_skip > nb_skip_max) {
                    nb_skip_max = nb_skip;
                }
                found = true;
            }
            j += 1;
            j = j & mask;
        }
    }
    *average = ((double)nb_skip_total) / nb_elements;
    *max = nb_skip_max;
}
```

On trouve un nombre moyen de sondages de l'ordre de 912 et un nombre maximal de sondages de 19 159. Ces deux valeurs sont très grandes et elles sont dûes à de nombreuses collisions. En effet, une observation rapide du fichier des adresses IP montre que de nombreuses adresses finissent par un 0. Ces adresses sont toutes des multiples de 256 et leur hachage sera un multiple de 256. L'algorithme tente donc de placer ces adresses dans une proportion très faible de cases du tableau. Il y a donc énormément de collisions et donc des séquences de sondage très longues.

6 Une meilleure fonction de hachage

1. 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 $r \in [\![0, 2^{64}[\![$. On a donc :

$$\begin{aligned} \{x\varphi\} &= \left\{\frac{xs}{2^{64}}\right\} \\ &= \left\{\frac{q2^{64}+r}{2^{64}}\right\} \\ &= \left\{q + \frac{r}{2^{64}}\right\} \\ &= \left(q + \frac{r}{2^{64}}\right) - q \\ &= \frac{r}{2^{64}} \end{aligned}$$

   car $q \in \mathbb{N}$ et $r \in [\![0, 2^{64}[\![$.

2. On effectue la décomposition de x_s en base 2. Il existe donc $d_0, d_1, \ldots, d_{63} \in \{0, 1\}$ tels que

$$x_s = \sum_{k=0}^{63} d_k 2^k.$$

On en déduit que

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

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

3. L'implémentation de la fonction `hash` devient immédiate.

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

4. 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.

7 Sondage quadratique

1. Cette nouvelle méthode nécessite de changer la fonction `set_search` qui devient alors :

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

La fonction `set_skip_stats` doit être changée de la même façon. 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.

2. 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 $i + 1 + \cdots + k \mod 2^p =$

$i + k(k+1)/2 \bmod 2^p$. Nous allons montrer que pour $k \in [\![0, 2^p[\![$, ces sondages se font dans des cases deux à deux distinctes. Pour prouver cela, on se donne $k_1, k_2 \in [\![0, 2^p[\![$ 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

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

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.