INF411 — X2021

Devoir à la maison

à rendre pour le dimanche 16 octobre 2022 à 18h

Ce devoir à faire chez soi doit être rendu au format PDF *sur Moodle avant le 16 octobre 2022 à 18h. Quel que soit l'outil utilisé (LaTeX, Word, LibreOffice, scanner, etc.), veuillez porter attention à la lisibilité. La rédaction doit être concise mais précise.*

Il est fortement conseillé d'utiliser un environnement de développement Java pour compiler et tester *le code qui est demandé, même si ce n'est pas une obligation. Des squelettes de code sont fournis. Quand vous aurez complété les classes* `Validator` *et* `Allocator`*, le programme* `AllocationSimulator.java` *vous permettra de visualiser le fonctionnement de votre allocateur et de le mettre sous le contrôle de votre validateur.*

Introduction

L'objectif de ce problème est la conception d'un *allocateur de mémoire*. Un allocateur travaille sur un ensemble d'indice $\{0, 1, \ldots, N-1\}$ et répond à deux types de requêtes : l'allocation et la libération.

Une requête d'allocation demande une plage de s indices consécutifs. L'allocateur fournit alors un indice a de sorte que la plage d'indices $\{a, a+1, \ldots, a+s-1\}$ (notée simplement $[a, a+s[$) soit disjointe des plages déjà allouées. L'allocateur lève une exception s'il ne parvient pas à honorer la requête.

Une requête de libération rend une plage d'indice $[a, a+s[$ précedemment allouée. L'allocateur peut alors de nouveau utiliser ces indices pour répondre aux futures requêtes d'allocation. On ne peut pas rendre une partie seulement d'une plage allouée, ou rendre des indices qui n'ont pas été alloués : le paramètre a doit être exactement la valeur de retour d'une allocation précédente de taille s.

Concrètement, l'interface d'un allocateur est la suivante.

```
class Allocator {
  Allocator(int N);
  int alloc(int size);
  void free(int addr, int size);
}
```

Mémo Java

— Obtenir le plus grand élément d'un `TreeSet<Integer> S` inférieur ou égal à x (ou `null` s'il n'y en a pas), complexité logarithmique

```
S.floor(x)
```

— Obtenir un élément d'un `HashSet<E> S` non vide

```
S.iterator().next()
```

— Lever une exception

```
throw new RuntimeException();
```

— Calculer 2^k, avec $0 \leqslant k < 31$ entier

```
1 << k
```

1 Un vérificateur d'allocateur

Pour tester un allocateur, c'est-à-dire vérifier qu'il n'alloue pas un indice déjà alloué, on propose l'interface suivante.

```
class Validator {
  Validator(int N);
  void alloc(int addr, int size);
  void free(int addr, int size);
  bool isfree(int addr, int size);
}
```

La méthode `alloc` signifie au validateur que la plage $[a, a+s[$ est désormais allouée. La méthode `free` signifie au validateur qu'une plage est libérée. La méthode `isfree` renvoie `true` si et seulement si la plage donnée en argument est incluse dans $[0, N[$ et aucun de ses éléments n'est alloué. Contrairement à un allocateur, le validateur ne fait aucun choix, il permet simplement de vérifier qu'une plage donnée est libre. On donne l'implémentation suivante du constructeur et des deux premières méthodes.

```
class Validator {
  TreeSet<Integer> startpoints, endpoints;
  int N;

  Validator(int N) {
    startpoints = new TreeSet<Integer>();
    endpoints = new TreeSet<Integer>();
    this.N = N;
  }

  void alloc(int addr, int size) {
    startpoints.add(addr);
    endpoints.add(addr + size);
  }

  void free(int addr, int size) {
    startpoints.remove(addr);
    endpoints.remove(addr + size);
  }
}
```

1. Écrire et expliquer la méthode `isfree`, en s'assurant d'une complexité logarithmique en le nombre de plages allouées.

2 L'allocateur *buddy*

Supposons que $N = 2^K$ est une puissance de 2. L'allocateur *buddy* décompose l'ensemble des indices en blocs. Un bloc de niveau $k \leqslant K$ est un ensemble d'indice $\left[a, a + 2^k\right[$, inclus dans $[0, N[$, où a est divisible par 2^k. L'adresse d'un bloc est l'entier a correspondant.

Par exemple, pour $N = 16$, on a un bloc $[0, 16[$ de niveau 4 (adresse 0), deux blocs $[0, 8[$ et $[8, 16[$ de niveau 3 (adresse 0 et 8 respectivement), quatre blocs $[0, 4[$, $[4, 8[$, $[8, 12[$ et $[12, 16[$ de niveau 2, huit blocs de niveau 1 et seize blocs de niveau 0.

Un bloc peut être réservé ou non. Un bloc libre est un bloc qui est disjoint des blocs réservés. Pour répondre à une requêtes d'allocation de taille s, l'allocateur va chercher un bloc libre de niveau $k = \lceil \log_2 s \rceil$ et le réserver. Ce bloc sera donc de la forme $\left[a, a + 2^k\right[$, et l'allocateur alloue la plage $[a, a + s[$. Les $2^k - s$ indices qui font partie du bloc réservé mais sont en dehors de la plage allouée ne sont pas utilisés. Quand une plage est libérée, le bloc réservé qui la contenait est de nouveau libre.

2. Avec $N = 16$, à combien de requêtes d'allocation de taille 3 peut répondre l'allocateur *buddy* avant d'echouer ? Après l'allocation réussie d'une plage de longueur 10, à combien d'allocations de taille 1 peut répondre l'allocateur avant d'échouer ?

Deux blocs sont *compagnons* si leur réunion est encore un bloc. Un bloc a exactement un compagnon, à gauche ou à droite, sauf l'unique bloc de niveau K qui n'en a pas.

3. Avec $N = 16$, quel est le compagnon du bloc $[4, 8[$? Et celui du bloc $[8, 12[$?
4. Donner l'équation qui relie l'adresse d'un bloc de niveau k à celle de son compagnon. Écrire une méthode `int buddy(int a, int k)` qui calcule l'adresse du compagnon du bloc de niveau k à l'adresse a (en supposant qu'il existe).

Un bloc libre est *maximal* s'il n'est pas inclus dans un bloc libre plus grand. Autrement dit, un bloc libre est maximal si c'est le bloc racine ou si son bloc compagnon n'est pas libre. Pour réserver un bloc en évitant la fragmentation, on choisira toujours de le placer dans un bloc libre maximal de plus petit niveau.

Pour implémenter l'allocateur, on propose la structure de classe suivante.

```
class Allocator {
  final int maxk;
  Vector<HashSet<Integer>> freeblocks;
}
```

La variable `maxk` contient K (rappelons que $N = 2^K$). Le tableau `freeblocks` contient $K + 1$ éléments qui sont des ensembles d'entiers, représentés par des `HashSet<Integer>`. On maintiendra l'invariant suivant :

> `freeblocks.get(k)` est l'ensemble des adresses des blocs libres maximaux de niveau k.

On pourra utiliser la notation F_k pour se référer à `freeblocks.get(k)`.

5. Avec $N = 16$, décrire précisément l'état d'une instance d'`Allocator` quand aucun bloc n'est réservé. Faire de même quand seul le bloc $[3, 4[$ est réservé.
6. Écrire une méthode `int reserve(int k)` qui réserve un bloc libre de niveau k et renvoie son adresse. On lèvera une exception `RuntimeException` s'il n'y a aucun bloc libre de niveau k.

Indication — S'il y a un bloc libre maximal de niveau k, on pourra prendre celui-ci. S'il n'y en a pas, on pourra réserver un bloc de niveau $k+1$ et en rendre une moitié.

7. Écrire la méthode `alloc`.
8. Écrire la méthode `free`.

   Indication — Bien maintenir l'invariant. Reprendre l'exemple de la question 5.

9. Avec $N \geqslant 4$ une puissance de deux, montrer que toute suite d'allocations et de libérations de plages de taille 1 ou 2 réussit toujours tant que le nombre d'indices demandés reste constamment inférieur à $\frac{2}{3}N$.

   Indication — C'est une question difficile. On pourra partitionner les blocs de niveau 1 en trois catégories : les blocs libres, les blocs pleins (entièrement alloués) et les blocs fragmentés (contenant un seul indice alloué). On pourra montrer que le nombre de blocs fragmentés est toujours inférieur à $\frac{1}{3}N + \frac{1}{2}$.