Contrôle classant X2012 INF421

David Monniaux

30 septembre 2014

1 Filtres de Bloom

Il est parfois utile de pouvoir tester l'appartenance à un sous-ensemble $\mathcal{E}$ d'un type de données $\mathcal{X}$ de façon approchée : une réponse négative veut dire que l'élément testé n'appartient pas à l'ensemble, mais une réponse positive veut dire que l'élément testé peut ou non lui appartenir. Nous désirons de plus que la probabilité (dans un sens que nous expliciterons plus loin) d'obtenir une réponse positive alors que l'élément n'appartient pas à l'ensemble soit faible. Enfin, nous souhaitons une réponse très rapide et une structure de données compacte en mémoire.

Une telle structure peut, par exemple, servir en préalable à l'interrogation par clef d'une base de données plus lente : on peut répondre très rapidement par la négative à la plupart des demandes d'éléments n'appartenant par à l'ensemble, et on n'interroge la base de données que pour les clefs effectivement existantes et une faible proportion des clefs absentes.

Les *filtres de Bloom* sont une telle structure de données. Il s'agit d'une variante des tables de hachage. Par souci de simplicité, nous nous intéresserons au cas où le filtre de Bloom représente un ensemble de chaînes de caractères ($\mathcal{X} = \texttt{String}$).

Un filtre de Bloom se compose d'un tableau de m bits (indexés de 0 à $m-1$) et d'une famille de n *fonctions de hachage* $h_k : \mathcal{X} \to \{0, \ldots, m-1\}$, pour $0 \leq k < n$. Pour le moment, nous ne prendrons pas d'hypothèses supplémentaires sur les fonctions de hachage par rapport à ce qui a été dit dans le cours et dans le poly.

On ajoute un élément x dans un filtre de Bloom en mettant à 1 les bits du tableau d'indices $h_k(x)$ pour $0 \leq k < n$. On teste sa présence en vérifiant si tous les bits $h_k(x)$ $(0 \leq k < n)$ sont mis à 1, auquel cas on répond « peut-être », et sinon on répond « non ».

1.1 Hachage

Les n fonctions de hachage h_k sont implantées par une unique fonction Java `public static int hash(int k, int m, String x)`. On se pro-

pose d'utiliser la fonction de hachage suivante, semblable à celle donnée dans le poly (on pourra supposer qu'on l'appelle sur `m` > 0) :

```
1     public static int hash(int k, int m, String x) {
2         if (k < 0 || k >= PRIMES.length)
3             throw new ArrayIndexOutOfBoundsException();
4         long h=0, p = PRIMES[k];
5         for(int i=0; i<x.length(); i++) {
6             h = h*p + x.charAt(i);
7         }
8         h *= p;
9         h = h%m;
10        return h;
11    }
```

La différence avec celle vue dans le poly est qu'au lieu d'utiliser un multiplicateur premier fixé, celui-ci dépend de k. [1]

Nous rappelons que l'opération $a\%b$ est, pour $b > 0$ et $a \geq 0$, le reste de la division euclidienne de a par b, et pour $b > 0$ et $a < 0$, l'opposé du reste de la division euclidienne de $-a$ par b. Ainsi, `(-7)% 4` s'évalue en -3.

Question 1.1. *Cette fonction ne compile pas : le compilateur proteste à la ligne 10. Pourquoi ?*

Solution : La variable `h` est de type **`long`**, tandis que la fonction est censée renvoyer **`int`**. Java refuse d'effectuer une conversion d'un type entier long à un type entier court à moins qu'on ne le lui dise explicitement de le faire : **`return`** `(`**`int`**`)(h%m);`. □

Question 1.2. *Cette fonction est censée renvoyer un nombre dans* $0, \ldots, m-1$. *Est-ce bien le cas ? Sinon, proposez un correctif.*

Solution : Comme rappelé dans l'énoncé, l'opérateur `%` de Java n'est *pas* le modulo mathématique. Il nous faut donc gérer le cas négatif, par exemple ainsi :

```
9         h = h%m;
10        if (h < 0) h+=m;
11        return (int) h;
```

□

1. Pour nos essais pour préparer l'énoncé, nous avons mis dans le tableau `PRIMES` les 5 premiers nombres premiers suivant 2^{40}, mais cela n'a aucune importance pour la résolution du problème.

1.2 Tableau de bits

Il nous serait bien entendu possible de représenter un tableau de m bits par un objet de type **boolean**[], mais, avec la plupart des machines virtuelles Java, un **boolean** (1 bit) occupe la même place qu'un **byte** (8 bits). Nous proposons donc d'utiliser un tableau de **long**, chaque élément étant utilisé comme un vecteur de 64 bits.

```
1  class Bitvector {
2      final int size;
3      final long[] tab;
4      final static int WORD_SIZE = 64;
5
6      public int size() {
7          return size;
8      }
9
10     public Bitvector(int n) {
11         if (n < 0) throw new ArrayIndexOutOfBoundsException
               ();
12         size = n;
13         tab = new long[(n+(WORD_SIZE-1))/WORD_SIZE];
14     }
15
16     public boolean test(int k) {
17         if (k < 0 || k >= size)
18           throw new ArrayIndexOutOfBoundsException();
19         return ((tab[k/WORD_SIZE] >>> (k%WORD_SIZE)) & 1)
               != 0;
20     }
21
22     public void set(int k) {
23         if (k < 0 || k >= size)
24           throw new ArrayIndexOutOfBoundsException("set " +
                 k + " in " + size);
25         tab[k/WORD_SIZE] = tab[k/WORD_SIZE] | (1 << (k%
               WORD_SIZE));
26     }
```

Rappelons que a<<b où a de type **int** (resp. **long**) renvoie un **int** (resp. **long**) égal à a décalé de b bits vers la gauche. Par exemple, `3 << 4` vaut 48 : l'entier 3 s'écrit

0000 0000 0000 0000 0000 0000 0000 0011

(32 chiffres binaires) et le décalage de 4 bits vers la gauche donne

0000 0000 0000 0000 0000 0000 0011 0000

c'est-à-dire 48 écrit en binaire. Si l'on avait eu affaire à un **long**, on aurait travaillé sur 64 chiffres binaires. Lorsqu'il n'y a pas de dépassement de

capacité (c'est-à-dire : des bits à 1 qui finissent par disparaître à gauche de l'entier, faute d'emplacements disponibles : 32 chiffres pour un **int**, 64 pour un **long**), cela correspond à une multiplication par 2^b. De même, $a>>>b$ renvoie a décalé de b bits vers la droite.

Question 1.3. *La ligne 25 contient un bug. Sauriez-vous dire lequel ? Et comment le corriger ?*

Solution : Le décalage à gauche opère sur un **int**, donc le résultat sera incorrect pour `k%WORD_SIZE` ≥ 32 puisqu'il y aura un dépassement de capacité, comme expliqué dans l'énoncé. Il faudrait avoir la constante 1 sous forme d'un **long**, par exemple en écrivant `1L` ou `(long)1`. □

1.3 Le filtre !

```
1  class BloomFilter {
2      /* hash() est definie ici */
3
4      final Bitvector v;
5      final int n;
6
7      public BloomFilter(int m, int n) {
8          v = new Bitvector(m);
9          this.n = n;
10     }
11
12     public void add(String val) {
13         final int m = v.size();
14         for(int k=0; k<=n; k++) {
15             v.set(hash(k, m, val));
16         }
17     }
18
19     public boolean test(String val) {
20         final int m = v.size();
21         for(int k=0; k<n; k++) {
22             if (v.test(hash(k, m, val))) return false;
23         }
24         return true;
25     }
26 }
```

La fonction `add` est censée ajouter un élément x dans le filtre, en marquant les bits d'indices $h_k(x)$ pour $0 \leq k < n$. La fonction `test` est censée tester la présence d'un élément x, en renvoyant **true** (« peut-être ») si tous les $h_k(x)$ sont vrais, et **false** sinon.

Question 1.4. *La fonction* `add` *contient un bug. Lequel et comment le corriger ?*

Solution : Le test <= fait que l'on utilise une fonction de hachage de trop (incohérent avec `test` et avec le constructeur). Utiliser plutôt <. □

Question 1.5. *La fonction* `test` *contient un bug. Lequel et comment le corriger ?*

Solution : Il manque une négation dans le test : `if (!v.test(hash(k, m, val)))`. □

Question 1.6. *Justifiez que si la procédure* `test` *corrigée répond* **`true`** *alors* `val` *peut être ou ne pas être dans l'ensemble représenté par le filtre. (Ici, on ne supposera rien sur les fonctions de hachage et vous pouvez notamment prendre des m, n quelconques et $h_k(x)$ fonctions quelconques vous permettant de construire un contre-exemple.)*

Solution : Pour $n = 1$ c'est le problème classique des collisions dans une table de hachage : deux éléments x et y différents peuvent être hachés vers le même indice.

Pour $n > 1$: soit $\mathcal{E} = \{x, y\}$ et $z \notin \mathcal{E}$. On peut très bien avoir $h_1(x) = 3$, $h_2(y) = 4$, $h_1(z) = 4$ et $h_2(z) = 3$ et alors on répondra « peut-être » puisque les bits d'indices 3 et 4 sont marqués. □

Question 1.7. *Supposons que calculer* `hash(k, m, v)` *prend un temps proportionnel à la longueur $|v|$ de la chaîne v. Quelle est la complexité du* `test` *?*

Solution : $O(n|v|)$. Notez que cette complexité est indépendante de m. □

1.4 Analyse probabiliste

Nous allons maintenant faire une hypothèse très forte : les fonctions $(h_k)_{1 \leq k \leq n}$ sont aléatoires, chaque $h_k(x)$ (pour $x \in \mathcal{X}$ et $0 \leq k < n$) étant une variable aléatoire indépendante uniformément distribuée dans $0 \ldots m-1$. Autrement dit, nous allons raisonner probabilistiquement comme si chaque $h_k(x)$ était tirée d'un immense tableau indexé par x, dont chaque case serait, au début de l'exécution du programme, choisie uniformément distribuée dans $\{0, \ldots, m-1\}$ indépendamment des autres. Notons que, *dans une exécution donnée*, $h_k(x)$ a une valeur précise (deux appels à h avec les mêmes paramètres k et x renvoient la même valeur) : l'aspect probabiliste intervient parce que l'on considère la distribution de probabilité induite sur les exécutions du programme et donc ses résultats finaux.

Question 1.8. *Quelle est la probabilité $p(m, n, f)$ qu'un filtre de Bloom, avec un tableau de m bits, n fonctions de hachage et f éléments dans $\mathcal{E}$ (l'ensemble approximativement représenté), renvoie « peut-être » quand il est interrogé sur un élément x qui n'est pas dans l'ensemble ? Justifiez.*

Solution : Soit $x \in \mathcal{X}$ fixé.

Fixons temporairement k. Pour chaque $y \in \mathcal{E}$ et chaque k', $h_{k'}(y)$ est uniformément distribuée dans $\{0, \ldots, m-1\}$. Elle a donc une probabilité $1 - 1/m$ d'être différente de $h_k(x)$. Par l'hypothèse d'indépendance des probabilités, $h_k(x)$ a une probabilité de n'être égale à aucune $h_{k'}(y)$ de :

$$\left(1 - \frac{1}{m}\right)^{fn} \tag{1}$$

Cet évènement est le même que « le test est faux à l'itération k ».

La fonction de test renvoie « vrai » si les tests à toutes les itérations sont vrais, donc avec une probabilité

$$p(m, n, f) = \left(1 - \left(1 - \frac{1}{m}\right)^{fn}\right)^{n} \tag{2}$$

Remarquons que tout ce que nous avons écrit ne s'applique pas si $x \in \mathcal{E}$: en effet, dans ce cas, pour $y = x$, les variables aléatoires $h_k(x)$ et $h_k(y)$ ne sont évidemment pas indépendantes ! □

Question 1.9. *Montrez que* $p(m, n, f) \simeq (1 - \exp(-fn/m))^n$ *quand* m *est grand devant* fn.

Solution : $\log(1 - 1/m)^{fn} = fn \log(1 - 1/m) \simeq fn/m$, d'où le résultat. □

2 Recherche de plus courts chemins sur une carte

Nous nous intéressons ici au problème suivant : étant donné une carte routière, représentée par un graphe orienté dont les arêtes portent des poids positifs (par exemple, distance par route, ou temps de trajet), et deux sommets a et b, calculer le trajet le plus court entre ces deux points (c'est-à-dire le chemin partant de a et finissant sur b de plus faible poids total). Les arêtes et poids ne sont pas forcément symétriques (il peut y avoir des sens uniques, des routes que l'on ne parcourt pas à la même vitesse dans les deux sens, etc.).

Dans la suite du problème, sauf précision, nous entendrons par « distance » entre deux points a et b le poids d'un chemin de poids minimal entre a et b (ou $+\infty$ s'il n'existe pas de tel chemin), quelque soit le sens donné aux poids.

2.1 Dijkstra

La première idée pour le résoudre est l'algorithme de Dijkstra, qui parcourt les nœuds du graphe par distance depuis a croissante, jusqu'à arriver

sur b ou avoir parcouru tous les nœuds accessibles depuis a (dans ce dernier cas, cela veut dire qu'il n'y a pas de chemin de a à b).

Nous supposerons les types suivants :

```
1  class Edge {
2      double value;
3      int target;
4
5      Edge(int target, double value) {
6          this.target = target;
7          this.value = value;
8      }
9  }
10
11 class Point {
12     double x, y;
13
14     Point(double x, double y) {
15         this.x = x;
16         this.y = y;
17     }
18
19     static double sqr(double x) {
20         return x*x;
21     }
22
23     double distance(Point b) {
24         return Math.sqrt(sqr(b.x-this.x)+sqr(b.y-this.y));
25     }
26 }
27
28 class Node {
29     String name;
30     Point position;
31     List<Edge> outgoing;
32
33     Node(String name, Point position) {
34         this.name = name;
35         this.position = position;
36         outgoing = new Vector<Edge>();
37     }
38 }
39
40 class Item implements Comparable<Item> {
41     double distance;
42     int node;
43
44     Item(int node, double distance) {
45         this.distance = distance;
46         this.node = node;
```

```
47      }
48
49      public int compareTo(Item y) {
50          double d1 = distance, d2 = y.distance;
51          if (d1 < d2) return -1;
52          else if (d1 > d2) return 1;
53          else return 0;
54      }
55  }
56
57  class ValuedGraph {
58      Node[] nodes;
59
60      ValuedGraph(int nbNodes) {
61          this.nodes = new Node[nbNodes];
62      }
63  }
```

Les nœuds seront identifiés à leur indice dans le tableau `nodes`. Notez que la structure du graphe n'est pas donnée par une table d'adjacence : chaque sommet, de type `Node`, contient la liste de ses arêtes sortantes dans le champ `outgoing`.

La bibliothèque standard Java dispose d'une classe `PriorityQueue<T>`, représentant une file de priorité, munie notamment d'un constructeur sans paramètre et des méthodes suivantes :

- `void add(T)` : ajoute un élément dans la liste ;
- `T poll()` : renvoie `null` si la file ne contient aucun élément, ou sinon renvoie l'élément minimal *et le supprime de la file.*

Le type `T` doit être muni d'une opération de comparaison ; nous avons proposé ci-dessus le type `Item`, qui contient un indice de nœud `node` et un champ `distance`, tel que les nœuds sont ordonnés par `distance` croissante.

Question 2.1. *Programmez dans la classe* `ValuedGraph` *une méthode* `double shortestPath(int a, int b)` *qui calcule le poids total d'un chemin de plus court poids du nœud a au nœud b, en utilisant* `PriorityQueue<Item>`.

Question 2.2. *Modifiez cette fonction pour renvoyer le chemin, sous la forme d'un tableau d'entiers contenant les indices des nœuds du chemin de a à b inclus. Il pourra être utile de modifier le type* `Item` *pour lui rajouter des champs.*

Vous pouvez indiquer vos modifications sans réécrire toute la fonction.

2.2 A*

Un défaut de l'algorithme de Dijkstra est qu'il ne tire par partie des informations géographiques globales. Par exemple, si l'on recherche un plus court chemin de Paris à Marseille, il va parcourir toutes les villes par distances

FIGURE 1 – L'algorithme de Dijkstra, partant de Paris, va d'abord parcourir Lille (distance routière depuis Paris : 220 km) avant de parcourir Lyon (465 km). En revanche, A* va d'abord parcourir Lyon (distance routière à Paris + distance à vol d'oiseau à Marseille : 741 km) et ne parcourra pas Lille (distance routière Paris–Marseille : 775 km, inférieure à Lille – Marseille à vol d'oiseau : 837 km).

croissantes depuis Paris, y compris Lille (qui est au nord de Paris) alors que l'on cherche un chemin vers une ville très au sud. Un humain rechercherait plutôt en priorité des chemins en direction de la ville de destination (donc, grossièrement vers le sud) et aurait plutôt tendance à spontanément chercher un trajet passant par Lyon (Fig. 1).

L'algorithme A* est une modification de l'algorithme de Dijkstra où l'on parcourt les nœuds v du graphe par valeurs croissantes de $f(v)$ définie comme la somme de la distance de a à v (poids d'un chemin de plus faible poids de a à v) et d'un minorant positif ou nul de la distance entre v et b. Par exemple, si les poids sont des distances routières entre villes, on peut prendre comme minorant de la distance sur le graphe entre v et b la distance « à vol d'oiseau » (euclidienne) entre v et b. Si les poids sont des temps de trajet, on peut prendre comme minorant de la distance sur le graphe entre v et b le quotient de la distance « à vol d'oiseau » entre v et b par la vitesse maximale autorisée (puisqu'un temps de trajet est minoré par le quotient de la distance routière par la vitesse maximale autorisée, et la distance routière est minorée par la distance à vol d'oiseau).

On suppose donnée une méthode `distanceEstimate(int v, int b)` donnant ce minorant.

Question 2.3. *Indiquez des modifications à apporter à* `shortestPath` *et à la classe* `Item` *pour implémenter cet algorithme.*

Solution :

```
1
2  class Item implements Comparable<Item> {
3      double distance;
4      double estimate;
5      int node;
6      int from;
7
8      public int compareTo(Item y) {
9          double d1 = distance+estimate, d2 = y.distance+y.
               estimate;
10         if (d1 < d2) return -1;
11         else if (d1 > d2) return 1;
12         else return 0;
13     }
14
15     Item(int node, double distance, double estimate, int
           from) {
16         this.distance = distance;
17         this.estimate = estimate;
18         this.node = node;
19         this.from = from;
20     }
21 }
```

```
22
23 class ValuedGraph {
24   ...
25
26     int[] shortestPath(int from, int dest, boolean aStar) {
27         int[] reverse = new int[nodes.length];
28         for(int i=0; i<nodes.length; i++) reverse[i] = -2;
29
30         PriorityQueue<Item> queue = new PriorityQueue<Item
               >();
31
32         queue.add(new Item(from, 0, aStar ?
               distanceEstimate(from, dest) : 0.0, -1));
33
34         while(true) {
35             if (aStar) stepsAStar++; else stepsDijkstra++;
                   // PERF
36
37             Item curItem = queue.poll();
38             if (curItem==null) break;
39             int cur = curItem.node;
40             if (reverse[cur] >= -1) continue;
41             reverse[cur] = curItem.from;
42             if (cur == dest) break;
43             for(Edge edge : nodes[cur].outgoing) {
44                 queue.add(new Item(edge.target, curItem.
                       distance+edge.value,
45                       aStar ? distanceEstimate(edge.target,
                           dest) : 0.0, cur));
46             }
47         }
48
49         if (reverse[dest]<0) return null;
50         int count=0;
51         for(int cur=dest; cur>=0; cur=reverse[cur]) {
52             //System.out.println(cur);
53             count++;
54         }
55         int[] steps = new int[count];
56
57         for(int cur=dest, i=count-1; i>=0; cur=reverse[cur
               ], i--) {
58             steps[i] = cur;
59         }
60         return steps;
61     }
62
63   ...
64 }
```

□

Question 2.4. *Montrez que l'algorithme A* (parcours comme dans Dijkstra par $f(v)$ croissant) calcule effectivement le poids total d'un chemin de plus petit poids du nœud a au nœud b.*

Solution : La preuve est la même que celle du cours pour Dijkstra, à quelques modifications près. Les invariants de l'algorithme sont :

1. $a \in \mathit{visited} \cup \mathit{pqueue}$;
2. $\mathit{distance}[\mathit{source}] = 0$;
3. $\forall v \in \mathit{visited} \cup \mathit{pqueue}$; $a \xrightarrow{\mathit{distance}[v]}{}^{*} v$;
4. $\forall v \in \mathit{visited}$, $\mathit{distance}[v]$ est la longueur du plus court chemin de a à v ;
5. $\forall v \in \mathit{visited}$, $\forall w$ t.q. $v \xrightarrow{d} w$, $w \in \mathit{visited} \cup \mathit{pqueue}$ et $f(w) = \mathit{distance}[w] + \mathit{estimation}(w, b) \leq \mathit{distance}[v] + d + \mathit{estimation}(v, b)$
6. $\forall v$, si $a \xrightarrow{d}{}^{*} v$ et $d + \mathit{estimation}(v, b) < \min(\mathit{pqueue})$ alors $v \in \mathit{visited}$.

Prouvons tout d'abord ceci : $\mathit{distance}[u]$ est la longueur du plus court chemin de a à v, au moment où u sort de la file. Soit un chemin strictement plus court.

□

3 Arbres PATRICIA

Nous voulons représenter des fonctions partielles f des **`int`** dans, par exemple, les **`double`**. Étant donnée la représentation d'une fonction f, nous voulons obtenir efficacement la représentation des fonctions

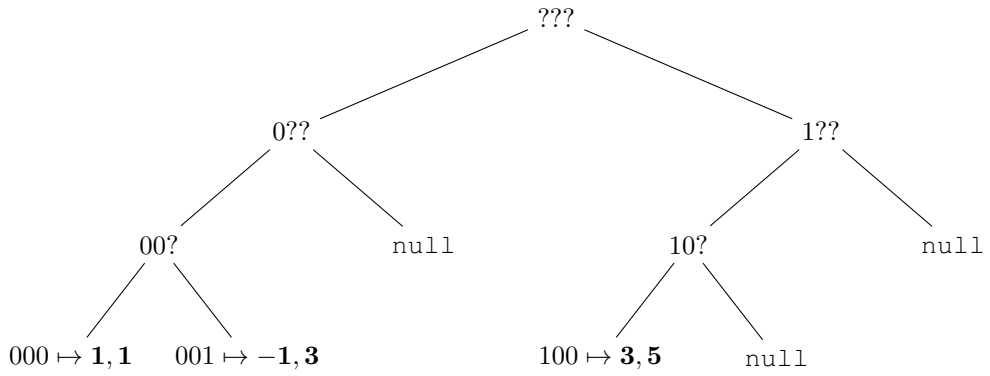
— $\mathit{update}(f, x_0, y)$, la fonction qui à x_0 associe y, et à $x \neq x_0$ associe $f(x)$;
— $\mathit{remove}(f, x_0)$, la fonction identique à f mais dont on a supprimé la clef x_0 (autrement dit, x_0 n'est plus dans le support de la fonction).

De plus, **on souhaite que, même après calcul de** $\mathit{update}(f, x_0, y)$ **ou** $\mathit{remove}(f, x_0)$**, l'ancien** f **reste valable et non modifié**.

Nous nous limiterons à des x dans l'intervalle $0, \ldots, 2^{H+1} - 1$ où $0 \leq H \leq 31$. On se propose d'utiliser la structure de données suivante : un arbre binaire incomplet (certaines branches peuvent valoir **`null`**) dont les feuilles contiennent un **`double`** et dont chaque nœud interne correspond à un chiffre de la décomposition de x en binaire (sur H chiffres binaires) en commençant par les chiffres de plus fort poids. Par ailleurs, on ne construit jamais de nœud interne dont les deux branches sont **`null`** (on met directement **`null`**).

Un tel arbre soit est **`null`** (fonction définie nulle part) soit est de hauteur exactement H ; notamment les feuilles sont exactement à profondeur H.

Expliquons cela sur un exemple : pour $H = 2$, pour représenter la fonction partielle $\{0 \mapsto 1, 1; 1 \mapsto -1, 3; 4 \mapsto 3, 5\}$, on considère les associations $\{000 \mapsto 1, 1; 001 \mapsto -1, 3; 100 \mapsto 3, 5\}$ et on construit l'arbre



Remarquez que l'on ne peut avoir de feuille qu'au niveau $H + 1$ de l'arbre, 0 étant la racine.

On vous donne le squelette de classe suivant ; vous ne changerez dans aucune des questions suivantes les déclarations des champs et constructeurs. La constante `HIGH_BIT` est H.

```
1  class PTree {
2      final PTree b0, b1;
3      final double leaf;
4
5      PTree(double leaf) {
6          this.leaf = leaf;
7          this.b0 = null;
8          this.b1 = null;
9      }
10
11     PTree(PTree b0, PTree b1) {
12         this.b0 = b0;
13         this.b1 = b1;
14         this.leaf = Double.NaN;
15     }
16
17     public static PTree empty() {
18         return null;
19     }
20
21     final static int HIGH_BIT = 31;
22
23     static boolean getBit(int x, int k) {
```

```
24          return (x & (1 << k)) != 0;
25      }
26  }
```

`getBit(`x`, `k`)` où $0 \leq k \leq H$ retourne le bit numéro k de l'entier x, $k = 0$ donnant le bit de poids faible.

Un nœud interne est représenté par `b0`$\neq$**`null`** ou `b1`$\neq$**`null`** ; dans le cas où `b0`=`b1`=**`null`** c'est qu'on est sur une feuille, dont le contenu est dans `leaf`.

Question 3.1. *Donnez une fonction* **`static double`** `get(PTree t, int x)` *qui retourne la valeur en x de la fonction représentée par t, ou la constante* `Double.NaN` *si x n'est pas dans le support de t (autrement dit, si t n'a pas de valeur en x).*

Vous pourrez programmer une deuxième fonction `getRec(PTree t, int x, int level)`.

Question 3.2. *Donnez une fonction* **`public static`** `PTree update(PTree t, int x, double y)`. *Rappel : l'ancien t doit rester inchangé.*

Vous pourrez programmer une deuxième fonction `updateRec(PTree t, int x, double y, int level)`.

Question 3.3. *Donnez une fonction* **`static`** `PTree newNode(PTree b0, PTree b1)` *qui construit un nœud interne sauf si* `b0` *et* `b1` *sont* **`null`**, *auquel cas elle renvoie* **`null`**.

Question 3.4. *Donnez une fonction* **`public static`** `PTree remove(PTree t, int x)`. *Rappel : l'ancien t doit rester inchangé.*

Vous pourrez programmer une deuxième fonction `removeRec(PTree t, int x, int level)`.

Soit maintenant l'interface suivante, représentant une fonction des **`double`** dans les **`double`** :

```
1  interface UnaryOp {
2      double op(double a);
3  }
```

Question 3.5. *Donnez une fonction* **`public static`** `PTree map(PTree t, UnaryOp f)` *qui retourne la représentation de $f \circ t$.*

Indice : il s'agit d'un arbre avec la même structure que t mais dont on a remplacé les feuilles par leur image par f.

Solution :

```
1  import java.util.Random;
2
3  interface UnaryOp {
4      double op(double a);
```

```
5  }
6
7  interface BinaryOp {
8      double op(double a, double b);
9      double op1(double a);
10     double op2(double b);
11 }
12
13 class PTree {
14     final PTree b0, b1;
15     final double leaf;
16
17     PTree(double leaf) {
18         this.leaf = leaf;
19         this.b0 = null;
20         this.b1 = null;
21     }
22
23     PTree(PTree b0, PTree b1) {
24         this.b0 = b0;
25         this.b1 = b1;
26         this.leaf = Double.NaN;
27     }
28
29     public static PTree empty() {
30         return null;
31     }
32
33     final static int HIGH_BIT = 31;
34
35     static boolean getBit(int x, int k) {
36         return (x & (1 << k)) != 0;
37     }
38
39     static PTree updateRec(PTree t, int x, double y, int
           level) {
40         if (level < 0) {
41             assert(t==null || (t.b0 == null && t.b1 == null
                   ));
42             return new PTree(y);
43         }
44         if (getBit(x, level)) {
45             if (t != null) {
46                 return new PTree(t.b0, updateRec(t.b1, x, y
                       , level-1));
47             } else {
48                 return new PTree(null, updateRec(null, x, y
                       , level-1));
49             }
```

```
50          } else {
51              if (t != null) {
52                  return new PTree(updateRec(t.b0, x, y,
                        level-1), t.b1);
53              } else {
54                  return new PTree(updateRec(null, x, y,
                        level-1), null);
55              }
56          }
57      }
58
59      public static PTree update(PTree t, int x, double y) {
60          return updateRec(t, x, y, HIGH_BIT);
61      }
62
63      static double getRec(PTree t, int x, int level) {
64          if (t==null) return Double.NaN;
65          if (level < 0) {
66              assert(t.b0 == null);
67              assert(t.b1 == null);
68              return t.leaf;
69          }
70          if (getBit(x, level)) {
71              return getRec(t.b1, x, level-1);
72          } else {
73              return getRec(t.b0, x, level-1);
74          }
75      }
76
77      public static double get(PTree t, int x) {
78          return getRec(t, x, HIGH_BIT);
79      }
80
81      static PTree newNode(PTree b0, PTree b1) {
82          if (b0 == null && b1 == null) return null;
83          return new PTree(b0, b1);
84      }
85
86      public static PTree removeRec(PTree t, int x, int level)
            {
87          if (t == null || level < 0) return null;
88          if (getBit(x, level)) {
89              return newNode(t.b0, removeRec(t.b1, x, level-1)
                    );
90          } else {
91              return newNode(removeRec(t.b0, x, level-1), t.b1
                    );
92          }
93      }
```

```
94
95      public static PTree remove(PTree t, int x) {
96          return removeRec(t, x, HIGH_BIT);
97      }
98
99      public static PTree map(PTree t, UnaryOp f) {
100         if (t == null) return null;
101         if (t.b0 == null && t.b1 == null) {
102             return new PTree(f.op(t.leaf));
103         }
104         return new PTree(map(t.b0, f), map(t.b1, f));
105     }
106
107     public static PTree map2(PTree x, PTree y, BinaryOp f) {
108         if (x==null) {
109             if (y==null) return null;
110             if (y.b0 == null && y.b1 == null) {
111                 return new PTree(f.op2(y.leaf));
112             } else {
113                 return new PTree(map2(null, y.b0, f), map2(
                        null, y.b1, f));
114             }
115         }
116         if (y==null) {
117             if (x.b0 == null && x.b1 == null) {
118                 return new PTree(f.op1(x.leaf));
119             } else {
120                 return new PTree(map2(x.b0, null, f), map2(x
                        .b1, null, f));
121             }
122         }
123         if (x.b0 == null && x.b1 == null && y.b0 == null &&
                y.b1 == null) {
124             return new PTree(f.op(x.leaf, y.leaf));
125         }
126         return new PTree(map2(x.b0, y.b0, f), map2(x.b1, y.
                b1, f));
127     }
128
129     static int[] randomInts(Random rng, int n, int max) {
130         int[] t = new int[n];
131         for(int i=0; i<n; i++) {
132             t[i] = rng.nextInt(max);
133         }
134         return t;
135     }
136
137     static double[] randomDoubles(Random rng, int n) {
138         double[] t = new double[n];
```

```
139         for(int i=0; i<n; i++) {
140             t[i] = rng.nextDouble();
141         }
142         return t;
143     }
144 
145     public static void main(String[] args) {
146         final int N = 1000;
147         Random rng = new Random(0xDEADBEEFL);
148         int[] x = randomInts(rng, N, 1000000000);
149         double[] y = randomDoubles(rng, N);
150         PTree pt = empty();
151         for(int i=0; i<N; i++) {
152             pt = update(pt, x[i], y[i]);
153         }
154         for(int i=0; i<N; i++) {
155             if (get(pt, x[i]) != y[i]) {
156                 System.out.println("wrong␣get␣at␣" + x[i]);
157             }
158         }
159         PTree pt0 = pt;
160         for(int i=0; i<N/2; i++) {
161             pt = remove(pt, x[i]);
162         }
163         for(int i=N/2; i<N; i++) {
164             if (get(pt, x[i]) != y[i]) {
165                 System.out.println("wrong␣get␣at␣" + x[i]);
166             }
167         }
168         for(int i=N/2; i<N; i++) {
169             pt = remove(pt, x[i]);
170         }
171         if (pt != null) {
172             System.out.println("wrong␣after␣remove");
173         }
174         PTree pt1 = map(pt0, new UnaryOp() { public double
                op(double x) { return x+2.0; }});
175         for(int i=0; i<N; i++) {
176             if (get(pt1, x[i]) != y[i]+2.0) {
177                 System.out.println("wrong␣map␣at␣" + x[i]);
178             }
179         }
180         pt1 = pt0;
181         for(int i=0; i<N; i+=2) {
182             pt1 = remove(pt1, x[i]);
183         }
184         PTree pt2 = map2(pt0, pt1, new BinaryOp() {
185                 public double op(double x, double y) {
                        return x*y; }
```

```
186                 public double op1(double x) { return x+3.0;
                        }
187                 public double op2(double y) { return y-1.0;
                        } });
188         for(int i=0; i<N; i++) {
189             if (i%2 == 0) {
190                 if (get(pt2, x[i]) != y[i]+3.0) {
191                     System.out.println("wrong map2 at " + i)
                            ;
192                 }
193             } else {
194                 if (get(pt2, x[i]) != y[i]*y[i]) {
195                     System.out.println("wrong map2 at " + i
                            + " " + y[i] + " " + get(pt2, x[i]));
196                 }
197             }
198         }
199     }
200 }
```

□