X2017 — INF411
Contrôle classant / 19 novembre 2018 / 9h–12h

Tous documents de cours autorisés (polycopié, transparents, notes personnelles). Le dictionnaire papier est autorisé pour les EV2 et EV3. Les ordinateurs, Ipad, dictionnaires électroniques, les tablettes et téléphones portables sont interdits.

L'énoncé est composé de deux problèmes indépendants, que vous pourrez traiter dans n'importe quel ordre (en revanche, merci de clairement numéroter les réponses). Vous pourrez, quand vous traiterez une question, considérer comme déjà traitées les questions précédentes du même problème.

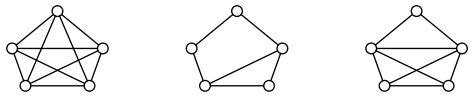
Le correcteur prêtera attention à la qualité de la rédaction et vous remercie d'avance d'écrire lisiblement. Merci également de numéroter vos copies en indiquant leur nombre total.

Quelques éléments de la bibliothèque standard Java sont rappelés à la fin du sujet. Ils ne sont pas forcément tous nécessaires. Vous pouvez utiliser librement tout autre élément de la bibliothèque standard Java.

1 Chemins eulériens

Dans ce problème, on s'intéresse uniquement à des graphes *non orientés* et *connexes*, c'est-à-dire où il existe toujours un chemin entre deux sommets donnés. Dans ce contexte, un *cycle eulérien* est un cycle qui emprunte *chaque arc une et une seule fois*.

Question 1 Pour chacun des trois graphes suivants, indiquer s'il possède ou non un cycle eulérien. Dans le cas positif, en exhiber un. On ne demande pas de justification.



Correction : oui / non / non

Question 2 Montrer que si un graphe admet un cycle eulérien alors tous ses sommets sont de degré pair. On rappelle que le *degré* d'un sommet est le nombre de ses voisins.

Correction : Soit un cycle eulérien partant du sommet s. Soit k_u le nombre de fois que le sommet u est visité par ce cycle eulérien. Pour $u \neq s$, le degré de u est alors $2k_u$ et donc pair. Pour le sommet s, le degré est $2k_s - 2$ et donc pair également.

Réciproque. Dans la suite de ce problème, on va démontrer que la réciproque est vraie, c'est-à-dire qu'un graphe dont tous les sommets sont de degré pair admet un cycle eulérien. On va le faire en programmant un algorithme qui construit un cycle eulérien. On suppose donnée une classe `V` pour les sommets, contenant en particulier une méthode `equals` pour comparer deux sommets.

Pour construire le cycle eulérien, on utilise la classe `Edge` donnée figure 1. Un objet de la classe `Edge` représente un arc du graphe entre les sommets `src` et `dst`. Les deux autres champs `prev` et

```
class Edge {
  V    src, dst;      // un arc entre src et dst
  Edge next, prev;    // chaînage vers l'arc suivant/précédent

  Edge(V src, V dst) {
    this.src = src;
    this.dst = dst;
    this.next = null;
    this.prev = null;
  }

  // relie l'arc this à l'arc e, en supposant this.dst = e.src
  void connect(Edge e) {
    this.next = e;
    e.prev = this;
  }

  // joint les deux cycles this et cycle, en supposant this.src = cycle.src
  void join(Edge cycle) {
    Edge p = this.prev;
    Edge q = cycle.prev;
    p.connect(cycle);
    q.connect(this);
  }
}
```

FIGURE 1 – Une classe pour représenter les chemins.

`next` permettent de chaîner les objets de la classe `Edge` dans une liste circulaire doublement chaînée, afin de représenter un cycle dans le graphe. Même si le graphe est non orienté, on choisit de voir un objet de la classe `Edge` comme allant de `src` à `dst`, de telle sorte que si deux objets a et b de la classe `Edge` sont liés, dans cet ordre, on aura

$$\begin{aligned} a.\texttt{next} &= b \\ b.\texttt{prev} &= a \\ a.\texttt{dst} &= b.\texttt{src} \end{aligned}$$

Une méthode `connect` est fournie pour réaliser une telle connection. Une autre méthode, `join`, est fournie, qui sera utilisée plus loin.

Le graphe est représenté par la classe `Graph` donnée figure 2, sous la forme d'ensembles d'adjacence, comme vu en cours. Plus précisément, `adj.get(v)` est l'ensemble des voisins du sommet `v`, sous la forme d'un ensemble d'objets de la classe `Edge`. Pour chaque objet e de cet ensemble, on a e.`src` = `v` et e.`prev` = e.`next` = `null`. Autrement dit, les arcs ne sont pas connectés entre eux initialement. Au fur et à mesure de la construction du cycle eulérien, ces arcs seront retirés du graphe et connectés entre eux pour former un cycle. Une méthode `anyEdge` est fournie. Étant donné un sommet `v` pour lequel il reste encore des arcs dans le graphe, elle renvoie un arc `e` tel que `e.src` = `v`, après l'avoir supprimé du graphe. Toutes les méthodes demandées dans les questions à venir seront écrites dans la classe `Graph`.

Question 3 On commence par un simple test que le critère est vérifié. Écrire une méthode

```
class Graph {
  HashMap<V, HashSet<Edge>> adj;

  // renvoie un arc issu de v, après l'avoir supprimé du graphe
  Edge anyEdge(V v) { ... }
}
```

FIGURE 2 – Une classe pour représenter le graphe.

```
boolean isEulerian()
```

qui détermine si le graphe possède uniquement des sommets de degré pair.

Correction :

```
boolean isEulerian() {
  for (V v: adj.keySet())
    if (adj.get(v).size() % 2 == 1)
      return false;
  return true;
}
```

Question 4 Soit un graphe dont tous les sommets ont un degré pair. Montrer qu'il possède un cycle (pas nécessairement eulérien).

Correction : Prenons un sommet v_0 quelconque. Comme le graphe est connexe, il y a des arcs sortant de v_0. Prenons un arc $v_0 - v_1$ quelconque. On le retire du graphe et on l'ajoute à notre cycle en construction. Comme v_1 a un degré pair, il existe au moins un autre arc $v_1 - v_2$. On le retire du graphe et on l'ajoute à notre cycle en construction. Si $v_2 = v_0$, on a terminé. Sinon, on continue à partir de v_2. Le processus termine nécessairement, car on retire à chaque fois un arc du graphe. On a bien utilisé le fait que le degré de chaque sommet est pair et maintenu cela comme un invariant (chaque fois qu'on entre et sort d'un nouveau sommet v_i, on supprime exactement deux arcs de v_i).

Question 5 On suppose que le graphe contient encore des arcs sortant d'un sommet `start` et que tous les sommets sont de degré pair. Écrire une méthode

```
Edge roundTrip(V start)
```

qui construit un cycle à partir du sommet `start` en utilisant des arcs encore présents dans le graphe. L'arc renvoyé est l'arc de ce cycle dont le champ `src` est `start`. Cette méthode ne pourra pas échouer, en vertu de la question précédente. On prendra soin de bien mettre à jour les champs `next` et `prev` des arcs utilisés, en appelant la méthode `connect` fournie.

Correction :

```
Edge roundTrip(V start) {
  Edge path = anyEdge(start), e = path;
  for (; !e.dst.equals(path.src); e = e.next)
```

```
    e.connect(anyEdge(e.dst));
  e.connect(path);
  return path;
}
```

Question 6 Écrire une méthode

```
Edge findVertexWithEdges(Edge cycle)
```

qui reçoit en argument un cycle et cherche le long de ce cycle un sommet qui possède encore des arcs. S'il existe un tel sommet v, cette méthode renvoie l'arc e de ce cycle pour lequel $e.\texttt{src} = v$. Si en revanche il n'existe aucun sommet le long du cycle possédant encore des arcs, cette méthode renvoie `null`.

Correction :

```
Edge findVertexWithEdges(Edge cycle) {
  Edge e = cycle;
  do {
    Set<Edge> s = edges.get(e.src);
    if (!s.isEmpty()) return e;
    e = e.next;
  } while (e != cycle);
  return null;
}
```

Algorithme de Hierholzer. Pour construire un cycle eulérien, on peut procéder ainsi. On commence par construire un cycle arbitraire, avec la méthode `roundTrip` et en partant d'un sommet quelconque. Si tous les arcs ont été utilisés, on a terminé. Sinon, on prend un sommet v sur le cycle qui possède encore des arcs. C'est possible, car le graphe est connexe. À partir de v, on construit un nouveau cycle, toujours avec la méthode `roundTrip`. Puis on joint les deux cycles pour n'en former qu'un seul. Et ainsi de suite jusqu'à épuisement des arcs. Pour joindre deux cycles, on utilise la méthode `join` fournie dans la classe `Edge`.

Question 7 Écrire une méthode

```
Edge eulerianCycle(V start)
```

qui construit et renvoie un cycle eulérien en suivant l'algorithme de Hierholzer.

Correction :

```
Edge eulerianCycle(V start) {
  Edge path = roundTrip(start);
  while (true) {
    Edge e = findVertexWithEdges(path);
    if (e == null) break;
    Edge cv = roundTrip(e.src);
    e.join(cv);
```

```
    }
    return path;
  }
```

Question 8 Donner la complexité dans le pire des cas de l'algorithme de Hierholzer tel que nous l'avons écrit, en fonction du nombre E d'arcs du graphe. On peut supposer que la complexité de `anyEdge` est $O(1)$.

Est-il possible d'obtenir une meilleure complexité pour la construction d'un cycle eulérien ? (Si oui, on ne demande pas d'écrire le code, mais seulement d'en décrire les idées.)

Correction : La méthode `roundTrip` a une complexité proportionnelle au nombre d'arcs du cycle qu'elle construit, qui sont retirés du graphe. La méthode `findVertexWithEdges` a une complexité proportionnelle à la taille du cycle dans lequel on fait la recherche, dans le pire des cas. Formellement, soit $C_1, C_2, \ldots, C_K$ les tailles des cycles successivement construits par `roundTrip`, avec donc $E = C_1 + C_2 + \cdots + C_K$. La construction du cycle C_i coûte

$$(C_1 + \cdots + C_{i-1}) + C_i$$

d'où un total

$$\sum_i (K + 1 - i) C_i \leq E \sum C_i = E^2.$$

La complexité est donc en $O(E^2)$. On a effectivement une complexité quadratique dans certains cas, par exemple une succession de cycles de longueur 2.

Pour obtenir une meilleure complexité, il faut diminuer le coût de `findVertexWithEdges`. Il suffit pour cela de maintenir dans un ensemble (de type `HashSet`) les sommets présents sur le cycle déjà construit et possédant encore des arcs disponibles. Pendant `roundTrip`, on prend soin de mettre à jour cet ensemble (ajouter les nouveaux sommets et supprimer les sommets qui n'ont plus d'arcs) mais cela ne change pas la complexité de `roundTrip` car les opérations d'ajout/suppression se font en temps constant. On a alors au final un algorithme en $O(E)$, ce qui est optimal.

Chemin eulérien. Un *chemin* eulérien dans un graphe est un chemin qui emprunte chaque arc une et une seule fois, sans nécessairement revenir au point de départ.

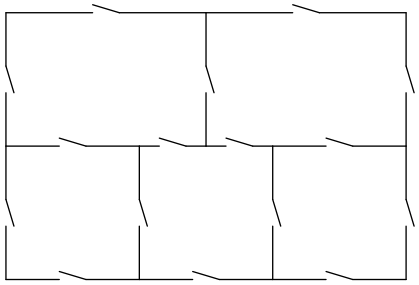
Question 9 Montrer qu'un graphe admet un chemin eulérien si et seulement s'il possède 0 ou 2 sommets de degré impair. Pour la réciproque, on s'attachera à décrire un algorithme construisant le chemin (mais on ne demande pas d'en écrire le code Java).

Correction : Soit un graphe admettant un chemin eulérien du sommet a au sommet b. On ajoute un nouveau sommet c et deux arcs $a - c - b$. On a alors un cycle eulérien et donc uniquement des sommets de degré pair. En supprimant le sommet c et les deux arcs $a - c - b$ on a bien au plus deux sommets de degré impair (aucun dans le cas où $a = b$, c'est-à-dire que le chemin était un cycle).

Pour la réciproque, on peut procéder exactement de la même façon. Si le graphe a deux sommets de degré impair, a et b, on ajoute un nouveau sommet c et deux arcs $a - c - b$. On a alors uniquement des sommets de degré pair et donc un cycle eulérien, que l'on peut

construire avec l'algorithme de Hierholzer. En supprimant les arcs $a - c - b$, on obtient un chemin eulérien de a à b. C'est bien là un algorithme qui construit un chemin eulérien.

Question 10 Est-il possible de tracer un chemin continu qui traverse les cinq pièces suivantes en empruntant chaque porte une et une seule fois ?



Le chemin n'a pas besoin d'être cyclique. Si oui, exhiber un tel chemin ; si non, justifier.

Correction : On construit un graphe ayant un sommet pour chaque pièce, ainsi qu'un sixième sommet pour l'extérieur. On relie ensuite les sommets chaque fois qu'il y a une porte entre les deux. (Il y a plusieurs arcs entre deux sommets, ce qui n'est pas un problème en soi. On peut ajouter des sommets intermédiaires à l'extérieur pour que ce ne soit pas le cas.)

On se retrouve alors avec quatre sommets de degrés impairs (trois pièces à 5 portes et l'extérieur à 9 portes). En vertu de la question précédente, il n'existe pas de chemin eulérien pour ce graphe et donc pas de chemin traversant les cinq pièces.

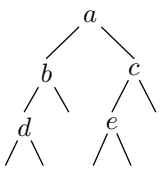
---

2 Tableaux flexibles

Dans ce problème, on représente un tableau par un arbre binaire. Si le tableau est vide, il est représenté par l'arbre vide. Sinon, l'élément d'indice 0 du tableau est stocké à la racine de l'arbre, le sous-arbre gauche contient une représentation du sous-tableau des éléments d'indices de la forme $2i+1$ et le sous-arbre droit contient une représentation du sous-tableau des éléments d'indices de la forme $2i + 2$. Ainsi, le tableau de taille 5

0	1	2	3	4
a	b	c	d	e

est représenté par l'arbre binaire



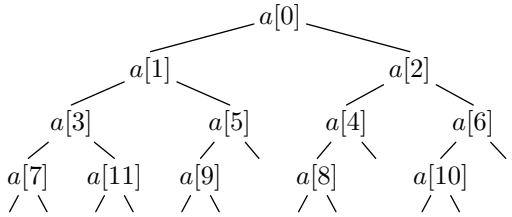
L'intérêt de cette représentation est qu'elle permet des tableaux qui peuvent être agrandis/rétrécis *aux deux extrémités*, tout en gardant des opérations raisonnablement efficaces. De tels tableaux portent le nom de *tableaux flexibles*.

```
class FlexibleArray {
  private Node root;
  private int  size;

  FlexibleArray() { this.root = null; this.size = 0; }

  Node find(Node n, int i) { ... }
  T get(int i) { return find(this.root, i).value; }
  void set(int i, T v) { find(this.root, i).value = v; }

  void highRemoval()      { ... }
  void highExtension(T x) { ... }
  void lowRemoval()       { ... }
  void lowExtension(T x)  { ... }
}

class Node {
  T value;
  Node left, right;

  Node(Node left, T value, Node right) {
    this.left = left; this.value = value; this.right = right;
  }

  Node(T value) {
    this.left = null; this.value = value; this.right = null;
  }
}
```

FIGURE 3 – Deux classes pour représenter les tableaux flexibles.

Question 11 Dessiner l'arbre binaire correspondant à un tableau de taille 12.

Correction :



Mise en œuvre en Java. On suppose que les éléments sont d'un type `T` donné qui n'est pas précisé. Un tableau flexible est représenté par un objet de la classe `FlexibleArray` donnée figure 3. Un tel objet encapsule un arbre binaire dans son champ `root` et le nombre total d'éléments dans son champ `size`. Les arbres binaires sont représentés par la classe `Node`, également donnée dans la figure 3. L'arbre vide est représenté par `null`.

Question 12 Écrire la méthode `Node find(Node n, int i)` qui cherche dans l'arbre `n` le nœud correspondant à l'indice `i` et le renvoie. On suppose que ce nœud existe.

Correction : Il suffit de descendre du bon côté, en testant la parité de `i`. On peut le faire récursivement, ou avec une boucle `while` comme ceci :

```
Node find(Node n, int i) {
  while (i != 0)
    if (i % 2 == 1) {
      n = n.left;
      i = i / 2;
    } else {
      n = n.right;
      i = i / 2 - 1;
    }
  return n;
}
```

Invariant structurel. Dans la suite, on note $\langle l, x, r\rangle$ un arbre binaire dont la racine a la valeur x, dont le sous-arbre gauche est l et dont le sous-arbre droit est r. La taille d'un arbre t, notée $|t|$, est son nombre de nœuds. Dit autrement, la taille de l'arbre vide est 0 et $|\langle l, x, r\rangle| = 1 + |l| + |r|$. On remarque que dans un tableau flexible, le sous-arbre gauche a toujours au moins autant de nœuds que le sous-arbre droit, car il y a au moins autant d'indices de la forme $2i + 1$ que d'indices de la forme $2i + 2$. Plus précisément, *pour tout sous-arbre* $\langle l, x, r\rangle$ d'un tableau flexible, on a

$$|r| \leq |l| \leq |r| + 1 \tag{1}$$

Question 13 Montrer que la hauteur d'un tableau flexible de taille N est en $O(\log N)$. En déduire la complexité des méthodes `get` et `set` données figure 3.

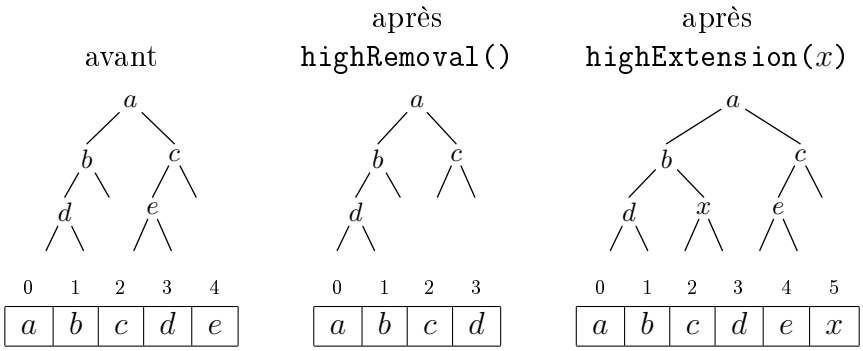
Correction : Montrons par récurrence sur N qu'un tableau flexible de taille N et de hauteur h vérifie

$$2^{h-1} \leq N \leq 2^h - 1$$

C'est vrai pour un arbre vide, où $N = h = 0$. Si $N = 2M + 1$, alors l'arbre a deux sous-arbres de taille M et de hauteur $h - 1$, avec $2^{h-2} \leq M \leq 2^{h-1} - 1$ par hypothèse de récurrence. Du coup, $2^{h-1} < 2M + 1 \leq 2^h - 1$. Si en revanche $N = 2M$, alors l'arbre a un sous-arbre gauche de taille M et de hauteur $h - 1$ et un sous-arbre droit de taille $M - 1$ et de hauteur $h' \leq h - 1$. Par hypothèse de récurrence sur le sous-arbre gauche, on a $2^{h-2} \leq M \leq 2^{h-1} - 1$ et donc $2^{h-1} \leq 2M < 2^h - 1$.

On en déduit que la complexité de `find` est en $O(\log N)$ et donc également celle des méthodes `get` et `set`.

Agrandir/rétrécir du côté droit. Notre objectif est de réaliser quatre opérations pour respectivement agrandir ou rétrécir le tableau d'une unité sur l'une ou l'autre de ses extrémités. Commençons par le côté droit, c'est-à-dire celui des indices les plus grands. Si on reprend l'exemple du tableau donné en introduction, ces deux opérations donnent respectivement les arbres suivants :



En dessous de chaque arbre, on a dessiné le tableau qu'il représente. Le code de la méthode `highRemoval` est écrit de la façon suivante :

```
void highRemoval() {
  if (this.size == 0) throw new IllegalArgumentException("highRemoval");
  this.root = highRemovalAux(this.size, this.root);
  this.size--;
}
Node highRemovalAux(int s, Node n) {
  if (s == 1) return null;
  if (s % 2 == 0)
    n.left = highRemovalAux(s / 2, n.left);
  else
    n.right = highRemovalAux(s / 2, n.right);
  return n;
}
```

La méthode auxiliaire `highRemovalAux` modifie l'arbre `n`, supposé de taille `s`, en y supprimant l'élément correspondant au dernier indice. La valeur renvoyée par cette méthode est la racine de l'arbre après la suppression (comme pour les arbres binaires de recherche vus en cours).

Question 14 Donner la complexité de la méthode `highRemoval` en fonction de la taille N du tableau flexible sur lequel elle est appelée.

Correction : C'est $O(\log N)$ en vertu de la question 13. En effet, le code de `highRemovalAux` fait une simple descente dans l'arbre `n`, avec un seul appel récursif à chaque fois sur l'un des deux sous-arbres.

Un argument encore plus simple consiste à dire que `s`, valant initialement N, est divisé par deux à chaque étape, ce qui ne peut être fait que $O(\log N)$ fois.

Question 15 Justifier que la méthode `highRemovalAux` ne provoque pas de `NullPointerException` lorsqu'elle est appelée depuis `highRemoval`.

Correction : La précondition de `highRemovalAux` est `s` $= |$`n`$| \geq 1$. On montre par récurrence sur `s` qu'elle est bien vérifiée et qu'il y a bien absence de `NullPointerException`. Si `s` $= 1$, alors c'est clair (`return null`). Si `s` > 1, alors on considère sa parité. Si `s` est pair (premier cas), alors `s` ≥ 2 et donc `s/2` $= |$`n.left`$| \geq 1$. On en déduit d'une part que `n` $\neq$ `null`, et donc que `n.left` ne provoque pas de `NullPointerException`, et d'autre part que l'hypothèse de récurrence s'applique. Si `s` est impair (second cas), alors `s` ≥ 3 et donc

`s/2` = |`n.right`| ≥ 1. De même, `n.right` ne provoque pas de `NullPointerException` et l'hypothèse de récurrence s'applique.

Question 16 Le code de la méthode `highExtension` est écrit sur le même modèle que celui de la méthode `highRemoval`, comme ceci :

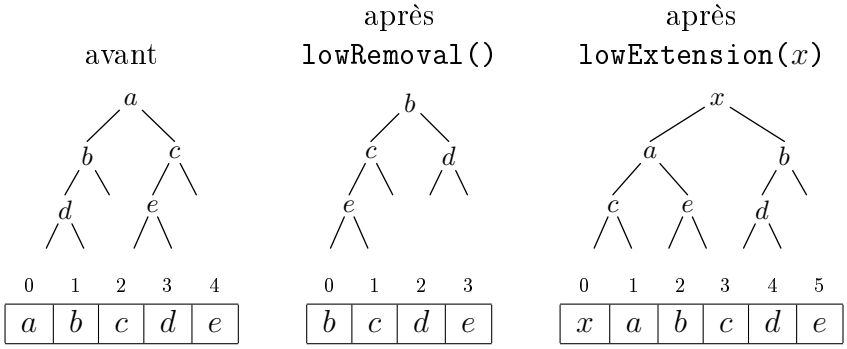
```
void highExtension() {
  this.root = highExtensionAux(this.size, this.root, x);
  this.size++;
}
Node highExtensionAux(int s, Node n, T x) { ... }
```

La méthode `highExtensionAux` modifie l'arbre `n`, supposé de taille `s`, en lui ajoutant l'élément `x` à la position correspondant à l'indice `s` dans le tableau qu'il représente. La valeur renvoyée par cette méthode est la racine de l'arbre après l'insertion (comme pour `highRemovalAux`). Donner le code de la méthode `highExtensionAux`. On garantira une complexité $O(\log \mathtt{s})$.

Correction : La parité de `s` nous permet de savoir si l'insertion doit se faire à gauche ou à droite.

```
Node highExtensionAux(int s, Node n, T x) {
  if (n == null)
    return new Node(v);
  if (s % 2 == 1)
    n.left = highExtensionAux(s / 2, n.left, x);
  else
    n.right = highExtensionAux(s / 2 - 1, n.right, x);
  return n;
}
```

Agrandir/rétrécir du côté gauche. Agrandir ou rétrécir le tableau du côté gauche, c'est-à-dire du côté des indices les plus petits, est plus délicat à réaliser, car on décale alors les indices des éléments. Si on reprend toujours le même exemple, alors ces deux opérations donnent respectivement les arbres suivants :



On prendra le temps de bien comprendre ce qui se passe ici. En particulier, on observera comment les éléments passent d'un côté à l'autre de l'arbre. En effet, les éléments situés à des indices pairs (resp. impairs) avant se retrouvent à des indices impairs (resp. pairs) après.

Question 17 Donner le code de la méthode `lowRemoval`. On pourra introduire pour cela une méthode auxiliaire `Node lowRemovalAux(Node n)` sur le même principe que pour les méthodes `highRemoval` et `highExtension`. On garantira une complexité logarithmique.

Correction : Là, pas besoin d'argument `s` pour déterminer de quel côté agir. C'est forcément du côté gauche que l'on fait la suppression, en échangeant ensuite les deux sous-arbres. Le cas de base est celui d'un arbre à un élément c'est-à-dire `n.left == null`.

```
void lowRemoval() {
  if (this.size == 0) throw new IllegalArgumentException("lowRemoval");
  this.root = lowRemovalAux(this.root);
  this.size--;
}
Node lowRemovalAux(Node n) { // n != null
  if (n.left == null) return null;
  n.value = n.left.value;
  Node l = n.left;
  n.left = n.right;
  n.right = lowRemovalAux(l);
  return n;
}
```

Question 18 Donner le code de la méthode `lowExtension`. Là encore, on pourra introduire une méthode auxiliaire. On garantira une complexité logarithmique.

Correction : D'une façon duale, on ajoute au sous-arbre droit puis on échange les deux sous-arbres. Le cas de base est celui d'un arbre vide.

```
void lowExtension(T v) {
  this.root = lowExtensionAux(v, this.root);
  this.size++;
}
Node lowExtensionAux(T v, Node n) {
  if (n == null) return new Node(v);
  Node l = n.left;
  n.left = lowExtensionAux(n.value, n.right);
  n.value = v;
  n.right = l;
  return n;
}
```

Construction à partir d'un vrai tableau. Si on se donne un (vrai) tableau `a` contenant N éléments de type `T`, on peut chercher à construire un tableau flexible contenant les mêmes éléments que `a`. Si on le fait naïvement en appelant N fois la méthode `highExtension` (ou N fois la méthode `lowExtension`) alors la complexité sera $O(N \log N)$ au total. Mais on peut faire mieux.

Question 19 Écrire un constructeur `FlexibleArray(T[] a)` qui reçoit en argument un tableau `a` de taille N et construit un tableau flexible ayant les mêmes éléments que `a`, en temps $O(N)$. Indication : on pourra introduire une méthode auxiliaire qui construit un tableau flexible à partir des éléments d'indices $d + 2^k, d + 2 \times 2^k, d + 3 \times 2^k, \ldots$ de `a`.

Correction : On introduit une méthode auxiliaire `ofArray` comme suggéré, qui prend en argument un indice initial `ofs` ainsi qu'un coefficient multiplicatif `m` (qui sera toujours une puissance de deux).

```
FlexibleArray(T[] a) {
  this.size = a.length;
  this.root = ofArray(a, 1, 0);
}
Node ofArray(T[] a, int m, int ofs) {
  if (ofs >= a.length) return null;
  return new Node(ofArray(a, 2*m, ofs+m), a[ofs], ofArray(a, 2*m, ofs+2*m));
}
```

Question 20 En supposant que le type `T` est `Integer`, on considère la méthode suivante :

```
FlexibleArray mystery(int n) {
  FlexibleArray a = new FlexibleArray();
  while (--n >= 0) {
    a.root = new Node(a.root, n, a.root);
    a.size = 2 * a.size + 1;
  }
  return a;
}
```

Expliquer ce que contient le tableau flexible renvoyé par cette méthode. Donner le coût en temps et en espace de cette méthode, en fonction de `n`.

Correction : Ce code construit l'arbre suivant :

Il correspond donc au tableau

0	1	1	2	2	2	2	...

formé de paquets de i de taille 2^i chacun, pour une taille totale $2^{\mathtt{n}} - 1$. La construction prend clairement un temps et un espace $O(\mathtt{n})$.

A Bibliothèque standard Java

`class HashSet<E>`	un ensemble dont les éléments sont de type `E`
`void add(E x)`	ajoute l'élément `x`
`boolean contains(E x)`	indique la présence de l'élément `x`
`int size()`	renvoie le cardinal de l'ensemble

On peut parcourir tous les éléments d'un ensemble `s` avec la construction
`for (E x: s) ...`

`class HashMap<K, V>`	un dictionnaire dont les clés sont de type `K` et les valeurs de type `V`
`void put(K k, V v)`	associe la valeur `v` à la clé `k` (en écrasant toute valeur précédemment associée à `k`, le cas échéant)
`boolean containsKey(K k)`	indique s'il existe une valeur associée à `k`
`V get(K k)`	renvoie la valeur associée à `k`, si elle existe, et `null` sinon
`Set<K> keySet()`	renvoie l'ensemble de toutes les clés de la table

On peut parcourir toutes les clés d'un dictionnaire `d` avec la construction
`for (K k: d.keySet()) ...`

* *
*