Solution du devoir 2

Auteur : Alexandre Blondin Massé

Question	1	2	3	4	Total
Sur	30	15	15	40	100
Note					

1. (30 points) Nous avons vu en classe que la complexité amortie d'une insertion dans un tableau redimensionnable est $\mathcal{O}(1)$. Supposons que nous souhaitions également compresser le tableau de moitié lorsque 1/4 ou moins des cases sont occupées.

   Soit $\alpha(T) = T.\text{utilisé} \ / \ T.\text{capacité}$. On définit une fonction potentielle

$$\Phi(T) = \begin{cases} 2 \cdot T.\text{utilisé} - T.\text{capacité}, & \text{si } \alpha(T) \geq 1/2; \\ T.\text{capacité}/2 - T.\text{utilisé}, & \text{si } \alpha(T) < 1/2. \end{cases}$$

   (a) (4 points) Montrez que $\Phi(T)$ est toujours positive ou nulle.

   (b) (6 points) Montrez que

$$\Phi(T) = \begin{cases} 0, & \text{si } \alpha(T) = 1/2; \\ T.\text{utilisé}, & \text{si } \alpha(T) = 1 \text{ ou } \alpha(T) = 1/4. \end{cases}$$

   (c) (20 points) Montrez que le coût amorti d'une suppression est constant. *Remarque :* Vous devez distinguer plusieurs cas, selon la valeur de $\alpha(T_{i-1})$ et selon qu'il y a une compression ou non.

Solution: (a) D'une part, si $\alpha(T) \geq 1/2$, alors $2T.\text{utilisé} \geq T.\text{capacité}$ et donc

$$\Phi(T) = 2T.\text{utilisé} - T.\text{capacité} \geq T.\text{capacité} - T.\text{capacité} = 0.$$

D'autre part, si $\alpha(T) < 1/2$, alors $2T.\text{utilisé} < T.\text{capacité}$, puis

$$\begin{aligned} \Phi(T) &= T.\text{capacité}/2 - T.\text{utilisé} = \frac{T.\text{capacité} - 2T.\text{utilisé}}{2} \\ &> \frac{T.\text{capacité} - T.\text{capacité}}{2} = 0. \end{aligned}$$

(b) Si $\alpha(T) = 1/2$, alors $\Phi(T) = 2T.\text{utilisé} - T.\text{capacité} = T.\text{capacité} - T.\text{capacité} = 0$.

Si $\alpha(T) = 1$, alors T.utilisé $= T$.capacité et donc $\Phi(T) = 2T$.utilisé$-T$.capacité $=$ $2T$.utilisé $- T$.utilisé $= T$.utilisé.

Finalement, si $\alpha(T) = 1/4$, alors $4T$.utilisé $= T$.capacité, ce qui entraîne que $\Phi(T) = T$.capacité$/2 - T$.utilisé $= 4T$.utilisé$/2 - T$.utilisé $= T$.utilisé.

(c) Soit u_i la valeur T.utilisé lorsqu'on applique la i-ème opération et s_i la valeur de T.capacité.

(i) Si $\alpha(T_{i-1}) \geq 1/2$, alors la i-ème opération ne déclenchera une compression que si l'on passe de 1 à 0 élément ou de 2 à 1 élément. Comme il s'agit d'un nombre fini de cas de base, on peut ignorer ces deux situations. Supposons donc que la i-ème opération ne déclenche pas de compression. Dans un premier temps, supposons que $\alpha(T_i) \geq 1/2$. Alors le coût amorti est

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi(T_i) - \Phi(T_{i-1}) \\
&= 1 + (2u_i - s_i) - (2u_{i-1} - s_{i-1}) \\
&= 1 + (2u_i - s_i) - (2(u_i + 1) - s_i) \\
&= 1 + 2u_i - s_i - 2u_i - 2 + s_i) \\
&= -1.
\end{aligned}$$

Maintenant, supposons que $\alpha(T_i) < 1/2$. Comme $\alpha(T_{i-1}) \geq 1/2$, on a $2u_{i-1} \geq s_{i-1}$ et donc

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi(T_i) - \Phi(T_{i-1}) \\
&= 1 + (s_i/2 - u_i) - (2u_{i-1} - s_{i-1}) \\
&= 1 + (s_{i-1}/2 - (u_{i-1} - 1)) - (2u_{i-1} - s_{i-1}) \\
&= 1 + s_{i-1}/2 - u_{i-1} + 1 - 2u_{i-1} + s_{i-1} \\
&= \frac{3s_{i-1} - 6u_{i-1} + 2}{2} \\
&\leq \frac{6u_{i-1} - 6u_{i-1} + 2}{2} \\
&= 1.
\end{aligned}$$

(ii) Si $\alpha(T_{i-1}) < 1/2$ et qu'il n'y a pas compression, alors

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi(T_i) - \Phi(T_{i-1}) \\
&= 1 + (s_i/2 - u_i) - (s_{i-1}/2 - u_{i-1}) \\
&= 1 + (s_{i-1}/2 - (u_{i-1} - 1)) - (s_{i-1}/2 - u_{i-1}) \\
&= 1 + s_{i-1}/2 - u_{i-1} + 1 - s_{i-1}/2 + u_{i-1} \\
&= 2.
\end{aligned}$$

(iii) Finalement, si $\alpha(T_{i-1}) < 1/2$ et qu'il y a compression, alors $s_i/2 = s_{i-1}/4 = u_{i-1} = u_i + 1$ et on obtient

$$\hat{c}_i = c_i + \Phi(T_i) - \Phi(T_{i-1})$$

$$\begin{aligned} &= (u_i + 1) + (s_i/2 - u_i) - (s_{i-1}/2 - u_{i-1}) \\ &= (u_i + 1) + ((u_i + 1) - u_i) - (2(u_i + 1) - (u_i + 1)) \\ &= 1. \end{aligned}$$

2. (15 points) En géométrie algorithmique, il est possible de représenter des solides à l'aide d'opérations ensemblistes. Par exemple, à la figure 1, on construit un représente un solide à l'aide de la réunion $\cup$, l'intersection $\cap$ et la différence $-$. Cet objet est appelé *solide de géométrie de construction*. On suppose que les feuilles sont des solides simples, qui offrent des fonctions S.IsInside(P), S.IsOutside(P) et S.IsOnBoundary(P) qui retournent vrai respectivement selon que le point P se trouve à l'intérieur, à l'extérieur ou sur la frontière du solide S.

   Étendez les fonctions S.IsInside(P), S.IsOutside(P) et S.IsOnBoundary(P) au cas où S est un solide de géométrie de construction. Vous pouvez supposer que les seuls opérateurs ensemblistes permis sont $\cup$, $\cap$ et $-$. De plus, vous pouvez supposer qu'il existe une fonction S.IsAtomic() qui retourne vrai si S est un solide de base (autrement dit, c'est une feuille dans la représentation), une fonction S.Operator() qui retourne la valeur Inter, Union ou Diff (comme pour un type énumératif) selon que le solide courant est obtenu par l'opération ensembliste $\cap$, $\cup$ ou $-$, ainsi que des fonctions S.Left() et S.Right() qui retournent le fils gauche et le fils droit de S lorsqu'il n'est pas un solide de base (autrement dit, c'est un noeud interne dans la représentation).

Solution: Écrivons d'abord la fonction S.IsInside(P) :

```
 1: fonction S.IsInside(P : point)
 2:     si ¬S.IsAtomic() alors
 3:         c ← S.Operator()
 4:         si c = Union alors
 5:             retourner S.Left().IsInside(P) ∨ S.Right().IsInside(P)
 6:         sinon si c = Inter alors
 7:             retourner S.Left().IsInside(P) ∧ S.Right().IsInside(P)
 8:         sinon si c = Diff alors
 9:             retourner S.Left().IsInside(P) ∧ S.Right().IsOutside(P)
10:         fin si
11:     fin si
12: fin fonction
```

On constate qu'elle fait appel à la fonction IsOutside. Définissons maintenant la fonction IsOnBoundary :

```
 1: fonction S.IsOnBoundary(P : point)
 2:     si ¬S.IsAtomic() alors
```

```
3:          c ← S.Operator()
4:          si c = Union alors
5:              retourner (S.Left().IsOnBoundary(P) ∧ ¬S.Right().IsInside(P))
6:                        ∨(S.Right().IsOnBoundary(P) ∧ ¬S.Left().IsInside(P))
7:          sinon si c = Inter alors
8:              retourner (S.Left().IsOnBoundary(P) ∧ S.Right().IsInside(P))
9:                        ∨(S.Right().IsOnBoundary(P) ∧ S.Left().IsInside(P))
10:         sinon si c = Diff alors
11:             retourner (S.Left().IsOnBoundary(P) ∧ ¬S.Right().IsInside(P))
12:                       ∨(S.Right().IsOnBoundary(P) ∧ S.Left().IsInside(P))
13:         fin si
14:     fin si
15: fin fonction
```

Finalement, la fonction IsOutside est obtenue en prenant la négation du résultat obtenu dans les deux autres cas :

```
1: fonction S.IsOutside(P : point)
2:     retourner ¬IsInside(P) ∧ ¬IsOnBoundary(S)
3: fin fonction
```

3. (15 points) Soit n un entier positif et $N = \{1, 2, \ldots, n\}$. Dans cette question, on s'intéresse à la représentation d'une famille de sous-ensembles de N à l'aide d'un arbre et/ou. Par exemple, on peut représenter la famille

$$\{\{1, 3\}, \{1, 4\}, \{2, 4\}, \{2, 3\}, \{2, 5\}\}$$

à l'aide de l'arbre illustré dans la figure 2. Ainsi, un noeud $\vee$ signifie qu'on peut choisir n'importe quel enfant alors qu'un noeud $\wedge$ indique qu'on doit choisir toutes les combinaisons possibles de chacun des enfants. Supposez qu'aucun sous-ensemble n'est répété dans l'arbre et/ou, et que les valeurs qui apparaissent dans chaque sous-arbre d'un noeud $\wedge$ sont distinctes. De plus, supposez que les fonctions suivantes sont déjà implémentées et s'effectuent toutes en $\mathcal{O}(1)$:

- Pour tout arbre T, T.Contenu() retourne la valeur contenue ($\wedge$, $\vee$ ou un élément de N) dans le noeud racine de T;
- Pour tout arbre T, T.SousArbres() retourne une liste ordonnée des sous-arbres de T;
- Pour tout arbre T, T.Cardinalite() retourne le nombre total de sous-ensembles représentés par T;
- Une fonction Uniforme() qui retourne un nombre réel aléatoire entre 0 inclusivement et 1 exclusivement.

Écrivez une fonction dont la complexité est $\mathcal{O}(n)$ qui retourne un sous-ensemble aléatoire représenté par un arbre et/ou quelconque, où n est le nombre de noeuds dans l'arbre.

Chaque sous-ensemble doit avoir la même probabilité d'être choisi (dans l'exemple ci-haut, chaque ensemble a donc une probabilité 1/5 d'être choisi).

Solution:

```
fonction ChoixAleatoire(A : arbre et/ou) : ensemble d'entiers
    si A.Contenu() = ∧ alors
                        ▷ On choisit un ensemble au hasard dans chaque enfant
                        ▷ puis on prend la réunion
        S ← ∅
        pour B ∈ A.SousArbres() faire
            S ← S ∪ ChoixAleatoire(B)
        fin pour
        retourner S
    sinon si A.Contenu() = ∨ alors
                        ▷ On choisit un enfant au hasard selon la cardinalité
        u ← Uniforme()
        p ← 0
        pour B ∈ A.SousArbres() faire
            p ← p + B.Cardinalite()/A.Cardinalite()
            si u < p alors
                retourner ChoixAleatoire(B)
            fin si
        fin pour
    sinon
        retourner {A.Contenu()}
    fin si
fin fonction
```

4. (40 points) En utilisant votre langage de programmation préféré parmi Java, C/C++ et Python, implémentez

   (a) (10 points) un algorithme permettant de transformer un arbre rouge-noir en un arbre 2-3-4;

   (b) (10 points) un algorithme permettant de transformer un arbre 2-3-4 en un arbre rouge-noir.

   Dans les deux cas, la transformation doit préserver le contenu de l'arbre (c'est-à-dire que les mêmes clés doivent apparaître). Illustrez le fonctionnement de vos transformations sur au moins 2 exemples chacune.

   Note : Il n'est pas nécessaire d'implémenter les opérations d'insertion et de suppression dans les arbres rouge-noir et 2-3-4 pour répondre à cette question. Vous pouvez représenter les structures de données simplement par des tuples (en Python) ou à l'aide de classes (tous les langages) sans les valider (on suppose que l'utilisateur qui construit les arbres respecte la spécification).

(c) (20 points) Dites si vos transformations sont injectives/surjectives/bijectives, en le démontrant dans chaque cas ou en fournissant un contre-exemple.

Solution: (a) (b) Il y a sans doute plusieurs solutions possibles à question, mais certaines transformations semblent plus naturelles. Une première observation importante est qu'il est plus facile d'utiliser, dans les deux cas, une transformation qui fait en sorte que la hauteur noire d'un arbre rouge-noir est la même que la hauteur (habituelle) d'un arbre 2-3-4.

Commençons d'abord par la transformation d'un arbre 2-3-4 en un arbre rouge-noir. Il suffit de procéder comme suit :

- Lorsqu'on a un 2-noeud, on le remplace par un noeud noir, puis on transforme récursivement les deux noeuds enfants;
- Lorsqu'on a un 3-noeud, on le remplace par un noeud noir, qui a pour enfant un noeud rouge et un noeud noir, puis on transforme récursivement les trois noeuds enfants;
- Lorsqu'on a un 4-noeud, on le remplace par un noeud noir, qui a pour enfants deux noeuds rouges, puis on transforme récursivement les quatre noeuds enfants.

Réciproquement, il est possible de construire un arbre 2-3-4 à partir d'un arbre rouge-noir en inspectant la couleur des deux enfants du noeud courant :

- Si les deux enfants du noeud sont noirs, alors on le substitue par un 2-noeud, puis on transforme récursivement ses deux enfants;
- Si un enfant et noir et l'autre rouge, alors on le substitue par un 3-noeud, puis on transforme récursivement l'enfant noir et les deux enfants du noeud rouge;
- Si les deux enfants sont rouges, alors on le substitue par un 4-noeud, puis on trasnforme récursivement les quatre enfants des deux noeuds rouges.

En Python, on obtient le code suivant :

```python
BLACK = 'b'
RED = 'r'
INF = float('inf')

class RedBlackTree(object):
    r"""
    A red-black tree basic implementation.
    """

    def __init__(self, key=None, first=None, second=None, color=None):
        if key is None:
            assert first is None
```

```
            assert second is None
            assert color is None
            self.key = None
            self.subtrees = None
            self.color = BLACK
        else:
            assert first is not None
            assert second is not None
            assert color in [BLACK, RED]
            self.key = key
            self.subtrees = (first, second)
            self.color = color
            assert self.has_red_black_property()
            assert self.is_search_tree()

    def __repr__(self):
        if self.is_empty():
            return 'e'
        else:
            return '(%s, %s, %s, %s)' % (self.key,
                                         'R' if self.color == RED else 'B',
                                         str(self.subtrees[0]),
                                         str(self.subtrees[1]))

    def is_empty(self):
        r"""
        Returns True if ``self`` is an empty red-black tree.
        """
        return self.subtrees is None

    def black_height(self):
        r"""
        Returns the black height of ``self``.

        The black height of a red-black tree is the number of black nodes
        encountered when going from the root toward any leaf. If the number of
        such black nodes is not the same, then this is not a red-black tree and
        the value ``-1`` is returned.
        """
        if self.is_empty():
            return 1
        else:
            left = self.subtrees[0].black_height()
            right = self.subtrees[1].black_height()
            if left == -1 or right == -1 or left != right:
                return -1
            else:
                return left + int(self.color == BLACK)

    def has_red_black_property(self):
        r"""
        Returns True if ``self`` has the red-black property.

        A tree has the red-black property if its black height is well-defined
        and if there does not exist two consecutive red nodes on some path from
        the root toward a leaf.

        NOTE: In principle, we should also check if the root is black, but it
        is not pertinent for our purpose.
        """
        if self.black_height() == -1:
            return False
        elif self.is_empty():
            return True
```

```
        else:
            left = self.subtrees[0]
            right = self.subtrees[1]
            if self.color == RED and (left.color == RED or right.color == RED):
                return False
            else:
                return left.has_red_black_property() and right.
                    has_red_black_property()

    def is_search_tree(self):
        r"""
        Returns True if ``self`` verifies the "search tree" property.

        The "search tree" property ensures that all keys appearing in the left
        subtree are lower than the current key, and all keys appearing in the
        right subtree are higher.
        """
        return self._is_search_tree(-INF, INF)

    def _is_search_tree(self, left, right):
        r"""
        Helper function for ``is_search_tree``.

        The ``left`` and ``right`` parameters are used to verify that the key
        stored at the root of ``self`` is in the interval ``[left, right]``.
        """
        if self.is_empty():
            return True
        elif left <= self.key and self.key <= right:
            return self.subtrees[0]._is_search_tree(left,  self.key - 1) and\
                   self.subtrees[1]._is_search_tree(self.key + 1, right)
        else:
            return False

    def to_234_tree(self):
        if self.is_empty():
            return Tree234()
        else:
            left  = self.subtrees[0]
            right = self.subtrees[1]
            if left.color == BLACK and right.color == BLACK:
                return Tree234((self.key,),
                               left.to_234_tree(),
                               right.to_234_tree())
            elif left.color == RED and right.color == BLACK:
                return Tree234((left.key, self.key),
                               left.subtrees[0].to_234_tree(),
                               left.subtrees[1].to_234_tree(),
                               right.to_234_tree())
            elif left.color == BLACK and right.color == RED:
                return Tree234((self.key, right.key),
                               left.to_234_tree(),
                               right.subtrees[0].to_234_tree(),
                               right.subtrees[1].to_234_tree())
            else:
                return Tree234((left.key, self.key, right.key),
                               left.subtrees[0].to_234_tree(),
                               left.subtrees[1].to_234_tree(),
                               right.subtrees[0].to_234_tree(),
                               right.subtrees[1].to_234_tree())

class Tree234(object):
    r"""
    A basic implementation of a 2-3-4 tree.
```

```
    """

    def __init__(self, keys=[], *subtrees):
        assert len(keys) <= 3
        if not keys:
            self.subtrees = None
        else:
            assert len(subtrees) == len(keys) + 1
            self.keys = keys
            self.subtrees = subtrees
        assert self.is_perfectly_balanced()
        assert self.is_search_tree()

    def __repr__(self):
        if self.is_empty():
            return 'e'
        else:
            return '(%s, %s)' % (' '.join(str(key) for key in self.keys),
                                 ', '.join(str(subtree) for subtree in self.subtrees
                                    ))

    def is_empty(self):
        r"""
        Returns True if ``self`` is an empty tree.
        """
        return self.subtrees is None

    def arity(self):
        r"""
        Returns the arity of ``self``.

        The arity of a tree is the number of children of its root.
        """
        if self.is_empty():
            return 0
        else:
            return len(self.subtrees)

    def is_perfectly_balanced(self):
        r"""
        Returns True if ``self`` is a perfectly balanced tree.

        A perfectly balanced tree is either an empty tree or a tree whose
        subtrees are also perfectly balanced and whose height is the same.
        """
        if self.is_empty():
            return True
        else:
            return all(subtree.height() == self.subtrees[0].height()\
                           for subtree in self.subtrees) and\
                       all(subtree.is_perfectly_balanced() for subtree in self.subtrees)

    def height(self):
        r"""
        Returns the height of ``self``.
        """
        if self.is_empty():
            return 0
        else:
            return 1 + max(subtree.height() for subtree in self.subtrees)

    def is_search_tree(self):
        r"""
        Returns True if the keys respect the "search tree" property.
```

```
        The "search tree" property ensures that all keys stored in a given node
        induce a partition of intervals such that all keys in the corresponding
        subtree belong to the corresponding interval.
        """
        return self._is_search_tree(-INF, INF)

    def _is_search_tree(self, left, right):
        r"""
        Helper function to ``is_search_tree``.

        The ``left`` and ``right`` parameters are used to verify that the keys
        stored at the root of ``self`` are in the interval ``[left, right]``.
        """
        if self.is_empty():
            return True
        elif all(left <= key and key <= right for key in self.keys):
            bounds = (left,) + tuple(self.keys) + (right,)
            return all(subtree._is_search_tree(bounds[i] + 1, bounds[i + 1] - 1)\
                       for (i,subtree) in enumerate(self.subtrees))
        else:
            return False

    def to_red_black_tree(self):
        r"""
        Builds a red-black tree from ``self``.
        """
        if self.is_empty():
            return RedBlackTree()
        elif self.arity() == 2:
            return RedBlackTree(self.keys[0],
                                self.subtrees[0].to_red_black_tree(),
                                self.subtrees[1].to_red_black_tree(),
                                BLACK)
        elif self.arity() == 3:
            left  = RedBlackTree(self.keys[0],
                                 self.subtrees[0].to_red_black_tree(),
                                 self.subtrees[1].to_red_black_tree(),
                                 RED)
            right = self.subtrees[2].to_red_black_tree()
            return RedBlackTree(self.keys[1], left, right, BLACK)
        elif self.arity() == 4:
            left  = RedBlackTree(self.keys[0],
                                 self.subtrees[0].to_red_black_tree(),
                                 self.subtrees[1].to_red_black_tree(),
                                 BLACK)
            right = RedBlackTree(self.keys[2],
                                 self.subtrees[2].to_red_black_tree(),
                                 self.subtrees[3].to_red_black_tree(),
                                 BLACK)
            return RedBlackTree(self.keys[1], left, right, RED)

# Main
RBT = RedBlackTree(4,
                   RedBlackTree(1,
                                RedBlackTree(),
                                RedBlackTree(3,
                                             RedBlackTree(),
                                             RedBlackTree(),
                                             RED),
                                BLACK),
                   RedBlackTree(7,
                                RedBlackTree(),
                                RedBlackTree(),
```

```
                                          BLACK),
                          RED)
print '--------------'
print 'Red-black tree'
print '--------------'
print RBT
print 'Its corresponding 2-3-4 tree'
print RBT.to_234_tree()
print 'Back to red-black tree'
print RBT.to_234_tree().to_red_black_tree()
print 'Thus the transformation is not injective'
print RBT.to_234_tree().to_red_black_tree().to_234_tree()

T234 = Tree234((2, 5),
               Tree234((1,),
                       Tree234(),
                       Tree234()),
               Tree234((3, 4),
                       Tree234(),
                       Tree234(),
                       Tree234()),
               Tree234((6,),
                       Tree234(),
                       Tree234()))
print '----------'
print '2-3-4 tree'
print '----------'
print 'Some 2-3-4 tree'
print T234
print 'Its corresponding red-black tree'
print T234.to_red_black_tree()
print 'Back to 2-3-4 tree'
print T234.to_red_black_tree().to_234_tree()
```

Le résultat est

```
--------------
Red-black tree
--------------
(4, R, (1, B, e, (3, R, e, e)), (7, B, e, e))
Its corresponding 2-3-4 tree
(4, (1 3, e, e, e), (7, e, e))
Back to red-black tree
(4, B, (3, B, (1, R, e, e), e), (7, B, e, e))
Thus the transformation is not injective
(4, (1 3, e, e, e), (7, e, e))
----------
2-3-4 tree
----------
Some 2-3-4 tree
(2 5, (1, e, e), (3 4, e, e, e), (6, e, e))
Its corresponding red-black tree
(5, B, (2, R, (1, B, e, e), (4, B, (3, R, e, e), e)), (6, B, e, e))
```

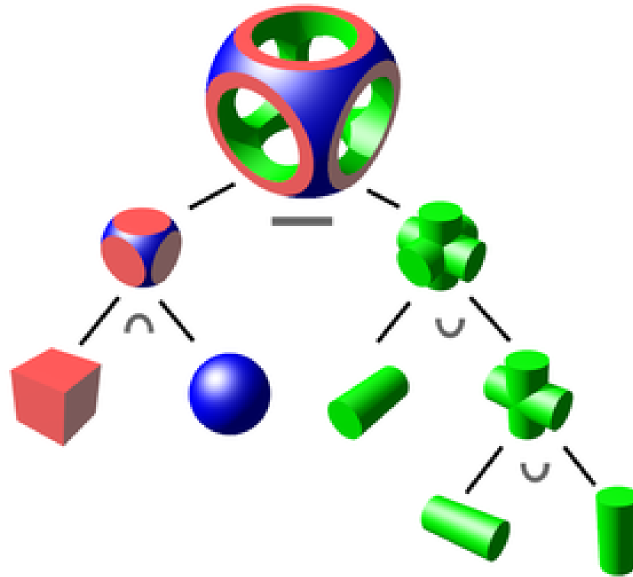

Figure 1: Un solide de géométrie de construction.

```
Back to 2-3-4 tree
(2 5, (1, e, e), (3 4, e, e, e), (6, e, e))
```

(c) Appelons T la transformation d'un arbre rouge-noir en un arbre 2-3-4 et T' celle qui transforme un arbre 2-3-4 en un arbre rouge-noir.

D'une part, il suit des exemples présentés en (a) (b) que T n'est pas injective. De la même façon, T' n'est pas surjective, puisque lorsqu'on transforme un 3-noeud, on place toujours le noeud rouge à gauche, ce qui nous empêche d'obtenir les arbres rouge-noir pour lesquels le noeud rouge se trouve plutôt à droite.

D'autre part, par définition de T et T', nous avons que $T \circ T' = \mathrm{Id}$. Ceci entraîne que T' est injective et T est surjective.

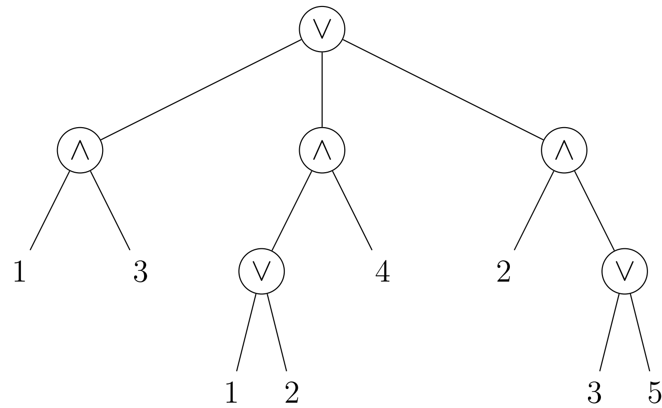

Figure 2: Un arbre et/ou représentant une famille de 5 sous-ensembles.