

STRUCTURES DE DONNÉES ET ALGORITHMES (SINF1121)

Mission 2 - Réponses aux questions

– 9 octobre 2013 –

TRAVAIL DU GROUPE 9 :

GÉGO Anthony	28581100
GENA Xavier	xxxxxx00
JOVENEAU Quentin	78491300
LIBIOULLE Thibault	60271100

MOYAU Arnold	xxxxxx00
NAVEAU Adrien	xxxxxx00
PAYEN Marlon	xxxxxx00

1 Question 1

Qu'entend-on par les notions de hauteur, de profondeur et de niveau dans le contexte de structures arborescentes ? Décrivez précisément ces notions et les liens éventuels entre elles ? Ces notions dépendent-elles du fait que l'on parle d'arbres binaires ou d'arbres en général ? Ces notions dépendent-elles de la structure de données utilisée pour représenter des arbres ? Justifiez vos réponses.

- La profondeur correspond au nombre d'ancêtres d'un noeud.
- Le niveau d d'un arbre correspond à l'ensemble des noeuds dont la profondeur est d. Graphiquement parlant, il correspond à un étage de l'arbre.
- La hauteur de l'arbre correspond au nombre total de niveaux, autrement dit la distance entre le noeud le plus éloigné et la racine.

Ces définitions sont valables pour n'importe quel type d'arbre, binaires ou pas. Ces notions ne dépendent pas forcément de la structure de données choisie. Nous verrons effectivement ci-dessous qu'il est possible de représenter un arbre sous forme de tableaux dans certains cas.

2 Question 2

Un arbre dont chaque noeud possède au plus deux fils est-il nécessairement binaire ? Qu'entend-on par arbre ordonné ? L'ordre dépend-il des valeurs mémorisées dans l'arbre ? Un arbre binaire impropre est-il désordonné ? Si l'on s'intéresse à des arbres dont la profondeur maximale est connue et fixée, y a-t-il une structure de données particulièrement bien adaptée à ce cas ? Si oui laquelle, sinon pourquoi ? Cela dépend-il du fait que l'arbre soit binaire ou non ? Cela dépend-il des opérations effectuées sur l'arbre ?

Pour correspondre à la définition, l'arbre doit également être ordonné. On parle alors de noeud enfant à gauche et à droite. Le noeud de gauche précède le noeud de droite. Un arbre binaire impropre correspond à un arbre dont les noeuds n'ont pas forcément 0 ou 2 noeuds enfants. Il n'est donc pas spécialement désordonné.

En supposant que l'arbre possède également un nombre d'enfant maximal fixé pour chaque noeud, on peut attribuer un indice à chaque noeud. Ainsi, les éléments peuvent facilement être stockés dans un tableau, une structure de données très simple.

Dans le cas contraire, il n'est pas possible de faire appel à cette méthode. Il faudra utiliser une structure plus complexes telles que des noeuds liés entre eux. Ainsi, un noeud contiendra une référence vers sa valeur et ses différents noeuds enfants. Il est donc nécessaire de maintenir une référence vers la racine.

3 Question 3

Qu'est-ce qu'un arbre équilibré ? Un arbre binaire équilibré est-il nécessairement propre ? Si oui, démontrez pourquoi, sinon donnez un exemple d'arbre équilibré impropre ? Comment définiriez-vous ce qu'est un arbre binaire (essentiellement) complet ? Pourquoi, à votre avis, est-il utile de préciser "(essentiellement)" ? Un arbre complet est-il toujours équilibré ? Un arbre équilibré est-il toujours complet ? Justifiez vos réponses.

Un arbre équilibré est un arbre ordonné à la hauteur logarithmique dont la valeur de tous les noeuds descendants respectent le critère de tri par rapport à ce noeud. Ainsi, par exemple, dans un arbre binaire équilibré, si 50 est le noeud racine, 17 peut être le fils de gauche, et 72 peut être le fils de droite, mais 62 ne sera jamais le fils de droite de 17. Il n'est donc pas nécessairement propre. Un arbre binaire complet est un arbre binaire propre. On dit 'essentiellement' car il peut y avoir un seul noeud fils par noeud. Un arbre complet n'est pas forcément équilibré

4 Question 4

Qu'entend-on par une implémentation d'un arbre par une structure chaînée ? En quoi cette notion de structure chaînée est-elle différente (ou plus générale) par rapport à celle de liste chaînée ? Quels sont les points communs entre liste et structure chaînée ? Quelle est la classe qui implémente un arbre par une structure chaînée dans DSAJ-5 ? Serait-il possible de remplacer cette implémentation par une autre utilisant une liste chaînée ?

Lorsqu'on implémente un arbre avec une structure chaînée, chaque élément a une référence vers l'élément qui est son parent et des références vers ses différents enfants. On peut donc ainsi facilement créer un arbre avec tout en haut un élément qui a une référence null comme parent et plusieurs enfants, qui ont chacun une référence vers leur parent et d'éventuels enfants et ainsi de suite. La différence avec une liste chaînée habituelle est que dans une liste chaînée en général, chaque élément a une référence vers l'élément suivant uniquement et l'on a une référence vers le premier élément de la liste pour pouvoir la parcourir. La classe `Tree` implémente un arbre par une structure chaînée. Oui on pourrait représenter un arbre par une liste chaînée 5 (?), mais il serait plus difficile d'accéder à chaque élément.

5 Question 5

On considère une variante de la représentation d'un arbre binaire en structure chaînée (DSAJ-5, pages 301 et suivantes). Dans cette variante, un noeud ne contient pas de référence vers son parent, ni de méthodes `parent` ou `iterator`. Ce fait constitue-t-il un problème pour réaliser des parcours de l'arbre binaire ? Pourquoi ? Donnez un algorithme pour réaliser la méthode `parent` sur base des autres méthodes disponibles dans l'interface. Si toutes les méthodes déjà disponibles s'exécutent en temps constant, quelle est la complexité temporelle de votre algorithme ?

Non, il est tout à fait possible de parcourir un arbre binaire en ayant une référence sur le premier élément tout en haut de l'arbre et ensuite de descendre dans l'arborescence de l'arbre. Dans ce cas-ci on retient la position du nœud ou on se situe en mémoire. Lorsque le programme parcourt l'arbre, une liste des positions des nœuds par lesquels on passe est établie.

6 Question 6

Un arbre binaire peut également être défini récursivement comme suit. Un arbre binaire est : – soit vide (sans aucun noeud), – soit il contient un noeud racine, un fils gauche qui est un arbre binaire et un fils droit qui est un arbre binaire. On considère l'interface `RBinaryTree` qui spécifie les méthodes essentielles d'un arbre binaire conformément à cette définition. Celle-ci fait appel à l'interface `Position` décrite dans DSAJ-5. Le code de ces deux interfaces est disponible sur le site iCampus, suivre Documents et liens/missions/m2/. Écrivez en Java la classe `LinkedRBinaryTree` qui implémente l'interface `RBinaryTree`.

```
1
3 public interface Position<E> {
4     /** Return the element stored at this position. */
5     E element();
6 }
```

Listing 1 – Position.java

```
2
3 /**
4  * Interface for a Binary Tree defined recursively.
5  *
6  * This interface uses the Position interface described in DSAJ-4.
7  */
8 public interface RBinaryTree<E> {
```

```

10  /**
    * @pre -
    * @post return true if this is empty, false otherwise.
12  */
    public boolean isEmpty();
14
16  /**
    * @pre -
    * @post return the number of nodes of this.
18  *     Note: the number of nodes of an empty tree is 0.
    */
20  public int size();
22
24  /**
    * @pre this is not empty.
    * @post return a reference to the tree root.
    */
26  public Position<E> root();
28
30  /**
    * @pre this is not empty
    * @post return true if this is reduced to a leaf (External Node),
    *     false otherwise
32  */
    public boolean isLeaf();
34
36  /**
    * @pre this is not empty.
    * @post return a reference to the left subtree.
38  */
    public RBinaryTree<E> leftTree();
40
42  /**
    * @pre this is not empty.
    * @post return a reference to the right subtree.
44  */
    public RBinaryTree<E> rightTree();
46
48  /**
    * @pre this is not empty.
    * @post o is the element stored at the root of this.
50  */
    public void setElement (E o);
52
54  /**
    * @pre this is not empty.
    * @post tree is the left subtree of this.
56  */
    public void setLeft (RBinaryTree<E> tree);
58
60  /**
    * @pre this is not empty.
    * @post tree is the right subtree of this.
62  */
    public void setRight (RBinaryTree<E> tree);
64
66  /**
    * @pre -
    * @post an iterable collection of the positions of this,
68  *     following an inorder traversal, is returned.
    */
70  public Iterable<Position<E>> positions();
    }

```

Listing 2 – RBinaryTree.java

```

1  public class Node<E> implements Position<E>
    {
3      private E value;

5      public Node(E value)
        {
7          this.value = value;
        }

9      @Override

```

```

11 public E element() {
12     return value;
13 }
14
15 }

```

Listing 3 – Node.java

```

1 import java.util.ArrayList;
2
3 public class LinkedRBinaryTree<E> implements RBinaryTree<E> {
4
5     private Position<E> value;
6     private RBinaryTree<E> left, right;
7
8     public LinkedRBinaryTree(E value, RBinaryTree<E> left, RBinaryTree<E> right)
9     {
10         this.value = new Node<E>(value);
11         this.left = left;
12         this.right = right;
13     }
14
15     @Override
16     public boolean isEmpty() {
17         return root() == null && leftTree() == null && rightTree() == null;
18     }
19
20     @Override
21     public int size() {
22         return ((ArrayList<Position<E>>) positions()).size();
23     }
24
25     @Override
26     public Position<E> root() {
27         return value;
28     }
29
30     @Override
31     public boolean isLeaf() {
32         return leftTree() == null & rightTree() == null;
33     }
34
35     @Override
36     public RBinaryTree<E> leftTree() {
37         return left;
38     }
39
40     @Override
41     public RBinaryTree<E> rightTree() {
42         return right;
43     }
44
45     @Override
46     public void setElement(E o) {
47         value = new Node<E>(o);
48     }
49
50     @Override
51     public void setLeft(RBinaryTree<E> tree) {
52         left = tree;
53     }
54
55     @Override
56     public void setRight(RBinaryTree<E> tree) {
57         right = tree;
58     }
59
60     @Override
61     public Iterable<Position<E>> positions() {
62         ArrayList<Position<E>> positions = new ArrayList<Position<E>>();
63         inorderPositions(this, positions);
64         return positions;
65     }
66 }

```

```

69 private void inorderPositions(RBinaryTree<E> v, ArrayList<Position<E>> pos)
71 {
    if(v.leftTree() != null)
73         inorderPositions(v.leftTree(), pos);
    if(v.root() != null)
75         pos.add(v.root());
    if(v.rightTree() != null)
77         inorderPositions(v.rightTree(), pos);
79 }

```

Listing 4 – LinkedRBinaryTree.java

7 Question 7

*Une expression arithmétique peut contenir les quatre opérateurs fondamentaux +, -, *, / et des scalaires, comme par exemple : $3 * 10 - 2/4$. Une expression analytique, telle que $x^2 + x \sin x^3$ peut contenir également des variables x, y, \dots , d'autres opérateurs comme la fonction puissance entière $^$ ou d'autres fonctions mathématiques comme \sin ou \cos . Une expression arithmétique peut être représentée par un arbre. Quelles sont les caractéristiques de cet arbre ? Pourquoi cette représentation est-elle utile ? Citez deux exemples de manipulation d'une expression arithmétique et exprimez comment ces manipulations sont mise en oeuvre à l'aide de cette représentation. Quelles sont les caractéristiques supplémentaires pour un arbre représentant une expression analytique ?*

Effectivement, une expression arithmétique peut être représentée par un arbre binaire propre. Pour ce faire, les opérandes sont représentés par des nœuds externes (nœuds n'ayant pas d'enfants et qui sont représentés par des variables ou des constantes) et les opérateurs sont représentés par des nœuds internes.

Cette représentation est utile car l'arbre est binaire et propre car les opérateurs nécessitent 2 opérandes pour réaliser une opération. Les 2 opérandes sont des nœuds externes et l'opérateur est un nœud interne donc il est facile de déterminer l'ordre de ces différents éléments. 2 exemples :

- $2 + 3$: Les opérandes 2 et 3 sont des nœuds externes et l'opérateur « + » est un nœud interne, donc nous pouvons réaliser l'opération en respectant l'ordre qui aboutit au résultat de cette addition.
- $6 / 3$: C'est le même principe que dans le premier exemple à savoir que l'on effectue une division entre les 2 opérandes qui sont 6 et 3.

8 Question 8

En supposant qu'un arbre représente une expression analytique, quel type de parcours de cet arbre permet-il de construire l'expression analytique complètement parenthésée qui lui correspond ? Donnez le pseudo-code d'une méthode public String toString() qui réalise cette construction.

L'expression peut être reconstituée à l'aide d'un parcours infixe. Il faudra s'assurer pour des expressions comme $\sin(x)$ que x soit bien le nœud fils droit de \sin .

```

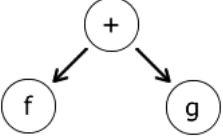
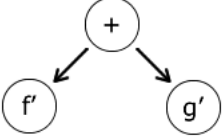
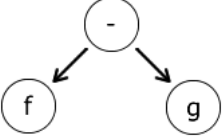
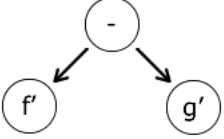
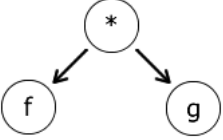
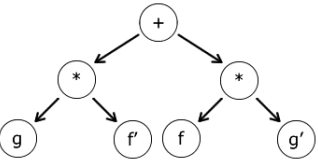
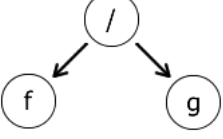
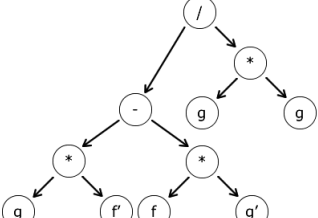
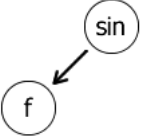
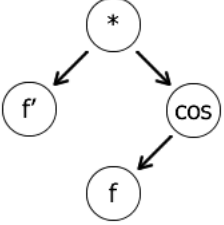
public String toString()
2 {
    resultat <- "(";
    4 si this a un noeud fils gauche g alors
        resultat <- resultat + g.toString();
    6 resultat <- resultat + this.element().toString()
    si this a un noeud fils droit d alors
    8         resultat <- resultat + d.toString();
    resultat <- resultat + ")";
10 retourner resultat;
}

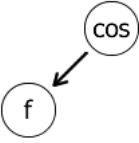
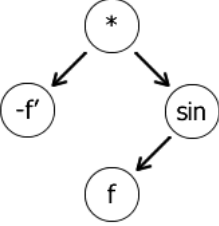
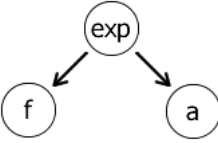
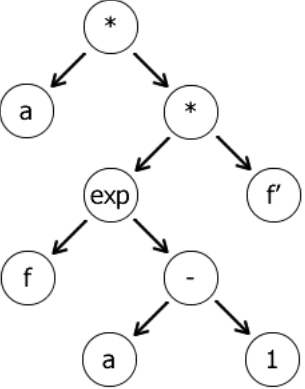
```

Listing 5 – Pseudo-code d'une méthode public String toString()

9 Question 9

Représentez graphiquement toutes les opérations de dérivation formelle (voir section 8) comme des opérations de manipulation d'un arbre et donnez une description sous forme de pseudo-code de ces opérations.

		<pre>deriveOpPlus(Tree root) { root.setLeft(derive(root.getLeft())); root.setRight(derive(root.getRight())); }</pre>
		<pre>deriveOpMinus(Tree root) { root.setLeft(derive(root.getLeft())); root.setRight(derive(root.getRight())); }</pre>
		<pre>deriveOpMultiply(Tree root) { Tree left = new Tree('*', root.getRight(), derive(root.getLeft())); Tree right = new Tree('*', root.getLeft(), derive(root.getRight())); root.setLeft(left); root.setRight(right); root.setRoot('+'); }</pre>
		<pre>deriveOpDivision(Tree root) { root.setRight(new Tree('*', root.getRight(), root.getRight())); root.setLeft(new Tree('-', new Tree('*', root.getRight(), derive(root.getLeft())), new Tree('*', root.getLeft(), derive(root.getRight()))); }</pre>
		<pre>deriveOpSin(Tree root) { root.setRoot('*'); root.setLeft(derive(root.getLeft())); root.setRight(new Tree('cos', root.getLeft())); }</pre>

 <pre> graph TD cos((cos)) --> f((f)) </pre>	 <pre> graph TD star1((*)) --> nf'((-f')) star1 --> sin((sin)) nf' --> f1((f)) sin --> f2((f)) </pre>	<pre> deriveOpCos(Tree root) { root.setRoot('*'); root.setLeft(- derive(root.getLeft())); root.setRight(new Tree('sin', root.getLeft())); } </pre>
 <pre> graph TD exp((exp)) --> f((f)) exp --> a((a)) </pre>	 <pre> graph TD star2((*)) --> a1((a)) star2 --> star3((*)) star3 --> exp((exp)) star3 --> fp((f')) exp --> f2((f)) exp --> minus((-)) minus --> a2((a)) minus --> 1((1)) </pre>	<pre> deriveOpExp(Tree root) { root.setRoot('*'); root.setLeft(root.getRight()); root.setRight('*', new Tree('exp', root.getLeft(), new Tree('-', root.getRight(), 1)), derive(root.getLeft())); } </pre>