Algorithmique et structures de données : Mission 2 (produit)

Groupe 1.2: Ivan Ahad - Jérôme Bertaux - Rodolphe Cambier
Baptiste Degryse - Wojciech Grynczel - Charles Jaquet

07 octobre 2014

Question 1 La profondeur est le nombre de parents qu'un noeud comporte. La racine est donc de profondeur 0, et ses enfants sont de profondeur 1. La hauteur est le nombre maximum de générations en dessous du noeud. Une feuille a une hauteur de 0, et les autre noeuds ont la hauteur de leur enfant le plus haut+1. Le niveau n est l'ensemble des nœuds de profondeur n. Ces notions ne dépendent pas du style d'arbre, elles s'appliquent aux arbres en général car il suffit d'avoir la notion de racine, feuille, enfant et parent pour pouvoir appliquer ces définitions. Ces notions ne dépendent pas de la structure de données utilisée car la représentation ne dépend pas de l'implémentation.

Question 2

Un arbre dont chaque noeud possède au plus deux noeuds est-il nécessairement binaire? Un arbre binaire se définit comme étant un arbre possédant au plus deux fils par noeud. Ceci étant dit, un arbre dont chaque noeud possède au plus deux fils est donc nécessairement un arbre binaire, tan qu'aucun noeud n'ait plus de deux fils.

Qu'entend-on par arbre ordonné? On dit qu'un arbre est ordonné si les enfants de chaque noeud sont ordonnés.Il existe trois types d'ordres dans les arbres :

-Ordre préfixe : Un arbre ordonné suivant l'ordre préfixe est un arbre dont la racine précède l'enfant de gauche, et l'enfant de gauche précède l'enfant de droite.

-Ordre infixe : Un arbre ordonné suivant l'ordre infixe est un arbre dont l'enfant gauche précède la racine, et la racine précède l'enfant de droite.

-Ordre postfixe : Un arbre ordonné suivant l'ordre postfixe est un arbre dont l'enfant de gauche précède l'enfant de droite, et l'enfant de droite précède la racine.

L'ordre dépend-il des valeurs mémorisées dans l'arbre? Si l'on parle de l'ordre des éléments contenus dans un arbre alors l'ordre dépend en effet des valeurs mémorisées dans celui-ci. Cependant, si l'on parle de la manière dont on parcourt un arbre, cela ne dépendra pas des valeurs mémorisées.

Exemple : Si l'arbre suit un ordre infixe, l'élément de la racine sera plus petit que l'élément du fils gauche, celui-ci étant plus petit que le fils droit, mais on pourrait très bien parcourir l'arbre en postorder traversal, c'est à dire parcourir l'arbre en commençant par l'enfant gauche, puis l'enfant droite, puis par la racine.

Un arbre binaire impropre est-il désordonné? Un arbre binaire propre est un arbre dont tous les noeuds internes ont soit aucun ou soit deux enfants chacun. Ceci étant dit, un arbre impropre est un arbre qui possède au moins un noeud n'ayant pas exactement deux fils.

Cependant, un arbre est binaire si et seulement s'il est ordonné et si chaque noeud possède au plus deux fils. Dès lors, un arbre binaire impropre sera d'office ordonné de par le fait qu'il est binaire, peu

importe qu'il soit propre ou impropre.

Y a-t-il une structure de données particulièrement bien adaptée à ce cas? Si la profondeur maximale de l'arbre est connue et fixée, les structures chainées sont adaptées pour représenter un arbre. Les nodes de la structure chainée peuvent aisément représenter la racine et les enfants d'un arbre, en contenant leur élément, une référence vers leurs enfants, et une référence vers leur parent.

Cela ne dépend pas du fait que l'arbre soit binaire ou non car on peut adapter les nodes de la structure chainée si chaque noeud de l'arbre possède plus de deux enfants. C'est pourquoi les structures chainées sont bien adaptées, car elles sont modulables en fonction du degré de chaque noeud de l'arbre. Cela ne dépend pas non plus des opérations effectuées sur l'arbre car les structures chainées peuvent être adaptées selon les opérations effectuées.

Question 3

Qu'est-ce qu'un arbre équilibré? Un arbre est équilibré si la différence des hauteurs de chaque sous-arbre gauche et droite de chaque noeud est au plus un.

Un arbre binaire équilibré est-il nécessairement propre? Un arbre binaire équilibré n'est pas nécessairement propre. Prenons un arbre dont la racine possède deux enfants, l'enfant de gauche possède un enfant et l'enfant de droite possède deux enfants. Dû au fait que l'enfant de gauche n'ait qu'un enfant, cet arbre est impropre, et l'arbre reste équilibré.

Comment définir un arbre binaire essentiellement complet? Un arbre essentiellement complet est un arbre dont tous les noeuds internes ont exactement deux enfants. On utilise le terme "essentiellement" car tous les noeuds ont exactement deux enfants, sauf les noeuds ayant une hauteur de zéro, soit les noeuds les plus profonds. Ces noeuds, appelés "feuilles" n'ont pas d'enfant.

Un arbre complet est-il toujours équilibré? Un arbre complet implique que tous les enfants d'un même parent ont la même hauteur. Dès lors, la différence des hauteurs d'enfants ayant le même parent sera toujours égale à 0, donc un arbre complet est toujours équilibré.

Un arbre équilibré est-il toujours complet? Un arbre équilibré peut ne pas être complet. Reprenons l'arbre décrit précédemment. L'enfant gauche possède un enfant, et l'enfant droite possède deux enfants. Cet arbre est équilibré mais n'est pas complet car l'enfant de gauche n'a qu'un seul enfant.

Question 4 Une implémentation d'un arbre par une structure chaînée signifie que pour réaliser l'arbre on utilise une structure chaînée. C'est-à-dire que chaque noeud de l'arbre contient une référence vers son noeud parent, vers son noeud enfant de gauche et vers son noeud enfant de droite en plus de contenir l'élément. L'arbre contient la référence du noeud racine de l'arbre et la taille ce celui-ci.

Cette notion de structure chaînée est plus générale qu'une liste chaînée car chaque noeud de l'arbre contient plus d'une référence. Dans un arbre à partir d'un noeud il est toujours possible de remonter vers le noeud parent ou alors descendre vers un des noeuds enfants alors que dans une liste chaînée on ne peut que se déplacer vers le noeud suivant. Donc la recherche d'un noeud s'effectue plus rapidement dans un arbre en structure chaînée.

Les points communs entre liste et structure chaînée :

- L'objet général (Arbre et liste) contiennent tous les deux la taille et une référence vers le noeud racine.
- Dans les deux implémentations les noeuds contiennent au minimum une référence vers un autre noeud.

La classe qui implémente un arbre par une structure chaînée est LinkedBinaryTree (DSAJ-6 page 297). Il est possible d'utiliser une implémentation utilisant une liste chaînée seulement si on utilise une liste double chaînée.

Question 5 Dans la majorité des cas de parcours, cela ne pose pas de problème, car on parcours l'arbre depuis la racine jusqu'aux feuilles. Cela peut cependant être handicapant dans certaines situations. Dans le cas où l'on voudrait, par exemple, parcourir tous les noeuds de l'arbre situés à un même niveau, ne pas avoir la possibilité de remonter peut poser problème et ralentir fortement le processus.

Pour réaliser la méthode parent, on peut faire comme suit: Garder le pointeur sur le noeud dont on veut le parent. Vérifier si ce noeud n'est pas le root, auquel cas on retourne null. Parcourir l'arbre noeud par noeud, en vérifiant pour chacun si un de ses fils n'est pas le noeud dont on cherche le parent. On finit donc par trouver le parent du noeud de départ.Puisqu'il faut, au pire, parcourir tout l'arbre pour trouver le père, on a une complexité en O(n).

Question 6 (Charles Jacquet) Définition de la classe LinkedRBinaryTree qui implémente l'interface RBinaryTree:

```java
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

/**
 *
 * @author charles
 *
 * @param <E>
 */
public class LinkedBinaryTree<E> implements RBinaryTree<E>, Position<E>{
    //variable d'instance:
    private LinkedBinaryTree<E> left;
    private LinkedBinaryTree<E> right;
    private E element;
    private int size;

    //constructeur:
    public LinkedBinaryTree(E elem, LinkedBinaryTree<E> leftChild, LinkedBinaryTree<E> rightChild) {
        element = elem;
        left = leftChild;
        right = rightChild;
        if (elem == null) size = 0;// pas d'élément dans la liste ..
        else size = 1; // si il y a un element la taille est de 1
        if(left != null) size+=left.size(); // rajoute de la taille des enfants
        if(right != null) size+=right.size();
    }
    /*
     * @post: implémentation de l'interface position
     */
    @Override
    public E element() {
        return element;
    }
    /*
     * @post: implémentation de l'interface RBinaryTree, renvoie true si l'arbre est vide
     */
    @Override
    public boolean isEmpty() {
```

```
40          return (size == 0);
41      }
42      /*
43       * @post: implémentation de l'interface RBinaryTree, renvoie
  la taille de l'arbre
44       */
45      @Override
46      public int size() {
47          return size;
48      }
49      /*
50       * @post: implémentation de l'interface RBinaryTree, renvoie
  la position de root.
51       *       Mais vu que nous implémentons le fait qu'un arbre
  est soit vide, soit
52       *       il contient un noeud racine avec un fils gauche et
  un fils droite.
53       *  Dès lors, chaque noeud est root, nous utiliserons donc
  cette fonction pour obtenir
54       *  la position d'un élément.
55       */
56      @Override
57      public Position<E> root() {
58          return this;
59      }
60      /*
61       * @post : implémentation de l'interface RBinaryTree, renvoie
  true si l'arbre ne possède pas d'enfant
62       */
63      @Override
64      public boolean isLeaf() {
65          return ( (left == null) && (right == null));
66      }
67      /*
68       * @post : implémentation de l'interface RBinaryTree, renvoie
  le fils gauche
69       */
70      @Override
71      public LinkedBinaryTree<E> leftTree() {
72          return left;
73      }
74      /*
75       * @post : implémentation de l'interface RBinaryTree, renvoie
  le fils droit
```

```
 76      */
 77     @Override
 78     public LinkedBinaryTree<E> rightTree() {
 79         return right;
 80     }
 81     /*
 82      * @post : implémentation de l'interface RBinaryTree, permet
    de modifier l'élement
 83      */
 84     @Override
 85     public void setElement(E o) {
 86         element = o;
 87     }
 88     /*
 89      * @post : implémentation de l'interface RBinaryTree, permet
    de remplacer/ ajouter un fils gauche
 90      */
 91     @Override
 92     public void setLeft(RBinaryTree<E> tree) {
 93         left = (LinkedBinaryTree<E>) tree;
 94     }
 95     /*
 96      * @post : implémentation de l'interface RBinaryTree, permet
    de remplacer/ ajouter un fils droit
 97      */
 98     @Override
 99     public void setRight(RBinaryTree<E> tree) {
100         right = (LinkedBinaryTree<E>)tree;
101 
102     }
103 
104     // Itérateur d'éléments grace a un itérateur de position:
105     public class ElementIterator implements Iterator<E>{
106         Iterator<Position<E>> posIterator =
    positions().iterator();
107         public boolean hasNext(){ return posIterator.hasNext();}
108         public E next(){ return posIterator.next().element();}
109         public void remove(){posIterator.remove();}
110     }
111     public Iterator<E> iterator(){ return new ElementIterator();}
112 
113     //fonction de "tri" "non ordonné" récursive de la liste:
114     private void inorderSubtree(LinkedBinaryTree<E> p,
    List<Position<E>> snapshot){
```

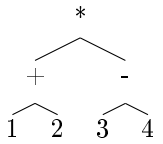
```
115         if(p.leftTree() != null)
116             inorderSubtree(p.leftTree(), snapshot);
117         snapshot.add(p.root());
118         if(p.rightTree() != null)
119             inorderSubtree(p.rightTree(), snapshot);
120 
121     }
122 
123     //traversée non ordonnée de la liste:
124     public Iterable<Position<E>> inorder(){
125         List<Position<E>> snapshot = new ArrayList<>(); // pk
    rien ds le <> ?
126         if(!isEmpty()){
127             inorderSubtree(this,snapshot);
128         }
129         return snapshot; // pourtant ce n'est pas un iterateur
    qu'on renvoie ???
130     }
131     /*
132      * @post : implémentation de RBinaryTree, permt de renvoyer
    un itérateur de positions
133      */
134     @Override
135     public Iterable<Position<E>> positions() {
136         return inorder();
137     }
138 
139     public String toString(){
140         Iterator<E> coucou = iterator();
141         StringBuffer buf = new StringBuffer();
142         while(coucou.hasNext()){
143             buf.append(coucou.next() + "\n");
144         }
145         return buf.toString();
146     }
147 
148 }
149 
```

Question 7 (GRYNCZEL Wojciech) **Une expression arithmétique peut être représentée par un arbre. Quelles sont les caractéristiques de cet arbre ?**
Arbre binaire est un arbre avec une racine, et où chacun des nœuds possède :

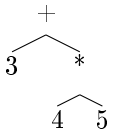
- soit aucun successeur,
- soit un successeur, à gauche ou à droite,
- soit deux successeurs.

Pourquoi cette représentation est-elle utile ?
Un arbre binaire peut facilement représenter une expression arithmétique, tout en gardant la priorité des opérations.
Citez deux exemples de manipulation d'une expression arithmétique et exprimez comment ces manipulations sont mise en oeuvre à l'aide de cette représentation.

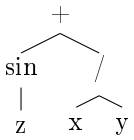
1. ((1+2)*(3-4))



2. (3+(4*5))



Quelles sont les caractéristiques supplémentaires pour un arbre représentant une expression analytique ?.
Une expression analytique contenant des opérations unaires, possède seulement un fils. Par exemple :

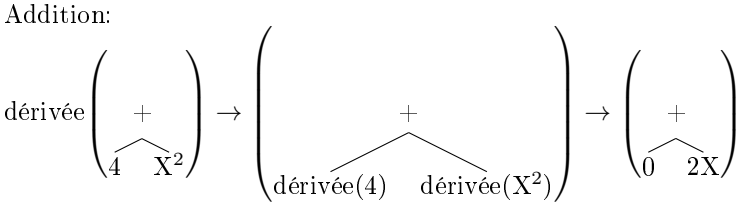


Question 8 C'est le parcours "inorder traversal" d'un arbre binaire qui permet de parcourir l'arbre dans le bon sens. Voir figure 1

```
public String toString(){
if ( T.isExternal(v) )
  return v.getElement().toString();
else
  return "("+ v.left() + v.getElement() + v.right() +")";
}
```

Question 9 (Charles Jacquet) Représentation des opérations de dérivation:

1. Addition:



2. Soustraction:

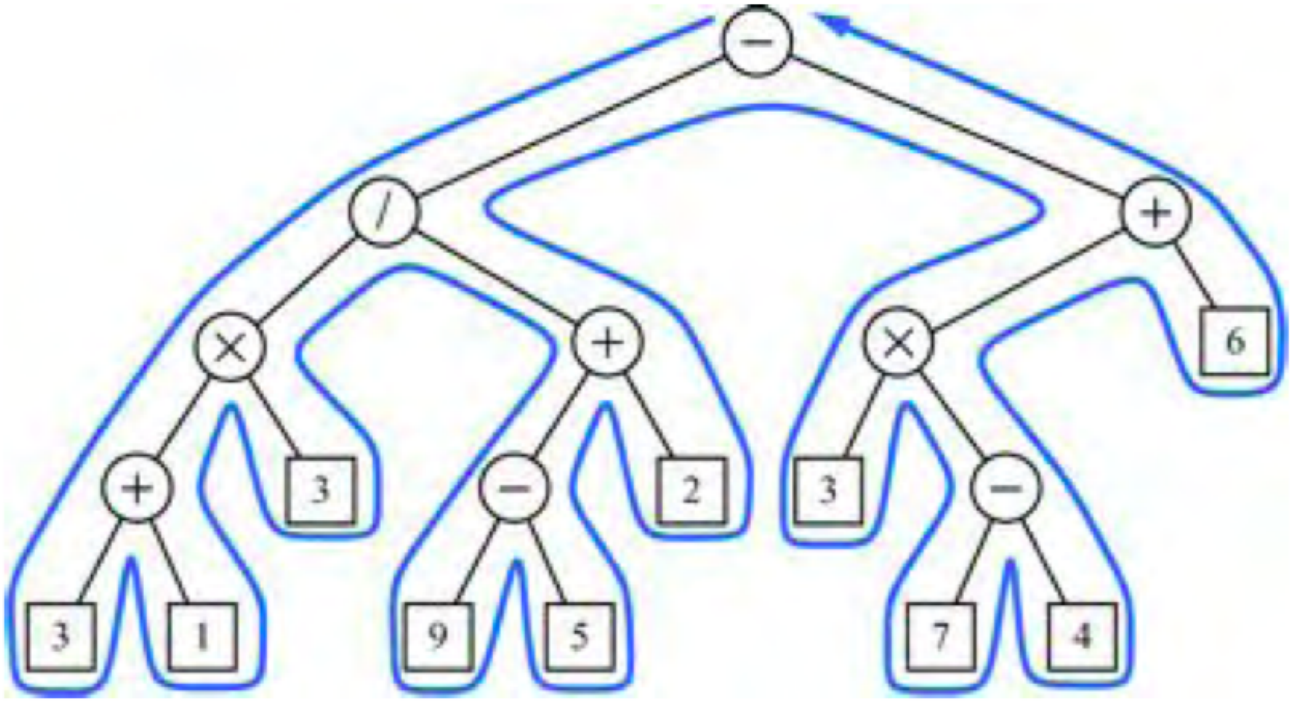


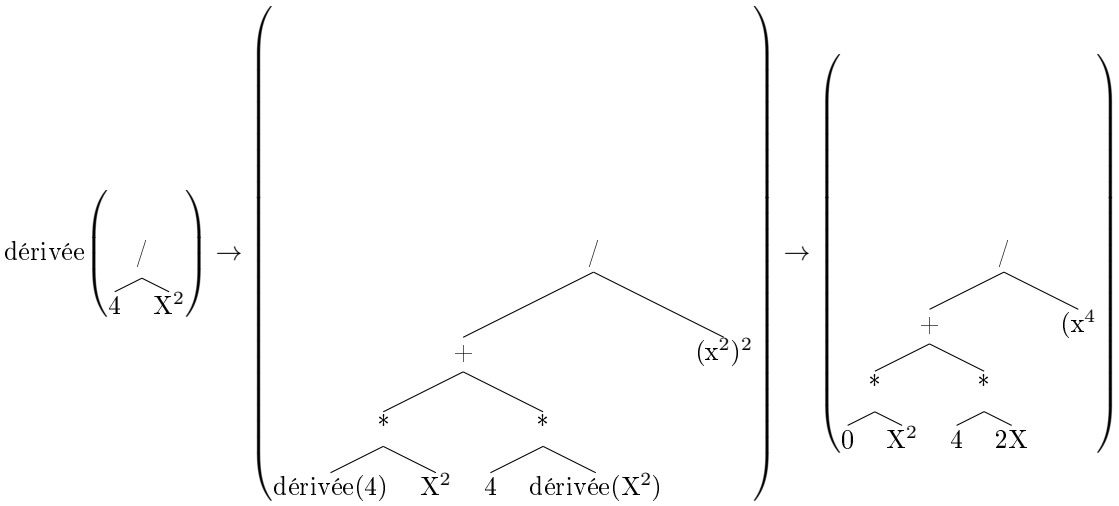
Figure 1: inorder traversal path (from Data Structure And Algorithms in Java p 424)

$$\text{dérivée}\left(\begin{array}{c}-\\ 4 \quad X^2\end{array}\right) \rightarrow \left(\begin{array}{c}-\\ \text{dérivée}(4) \quad \text{dérivée}(X^2)\end{array}\right) \rightarrow \left(\begin{array}{c}-\\ 0 \quad 2X\end{array}\right)$$

3. Multiplication:

$$\text{dérivée}\left(\begin{array}{c}*\\ 4 \quad X^2\end{array}\right) \rightarrow \left(\begin{array}{c}+\\ *\quad\quad *\\ \text{dérivée}(4) \quad X^2 \quad 4 \quad \text{dérivée}(X^2)\end{array}\right)\left(\begin{array}{c}+\\ *\quad\quad *\\ 0 \quad X^2 \quad 4 \quad 2X\end{array}\right)$$

4. Division:


dérivée
/
4
X²
→
/
+
(x²)²
*
*
dérivée(4)
X²
4
dérivée(X²)
→
/
+
(x⁴
*
*
0
X²
4
2X