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

Question 3

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 
```

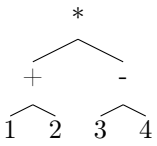
Question 7
Une expression arithmétique peut contenir les quatre opérateurs fondamentaux $+, -, *, /$ et des scalaires, comme par exemple : $3 * 10 - 2/4$. Une expression analytique, telle que $x^2 + x\ sin\ x - 3$ peut contenir également des variables $x, y, ...$, d'autres opérateurs comme la fonction 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 ?
Arbre binaire est un arbre avec une racine, et où chacun des nœuds possède :

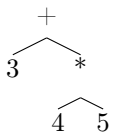
- soit aucun successeur,
- soit un successeur, à gauche ou à droite,
- soit deux successeurs.

Pourquoi cette représentation est-elle utile ?
//TODO Techniques for parsing the common infix notation (where the operator is typically between two operands) and converting it into a binary tree are well known. Expressions are converted into binary trees, where unary operators (i.e., the unary minus or the sine function) are converted into degenerate nodes in the binary trees, with a single son. See Fig. 1 for the arithmetic expression of sin . **Citez deux exemples de manipulation d'une expression arithmétique et exprimez comment ces manipulations sont mise en oeuvre à l'aide de cette représentation + Exemple !!!!!!!.**

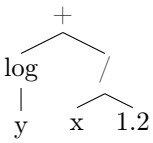
$$((1 + 2) * (3 - 4))$$



$$(3 + (4 * 5))$$



Quelles sont les caractéristiques supplémentaires pour un arbre représentant une expression analytique ?.
Une expression analytique peut contenir des opérations unaires, 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

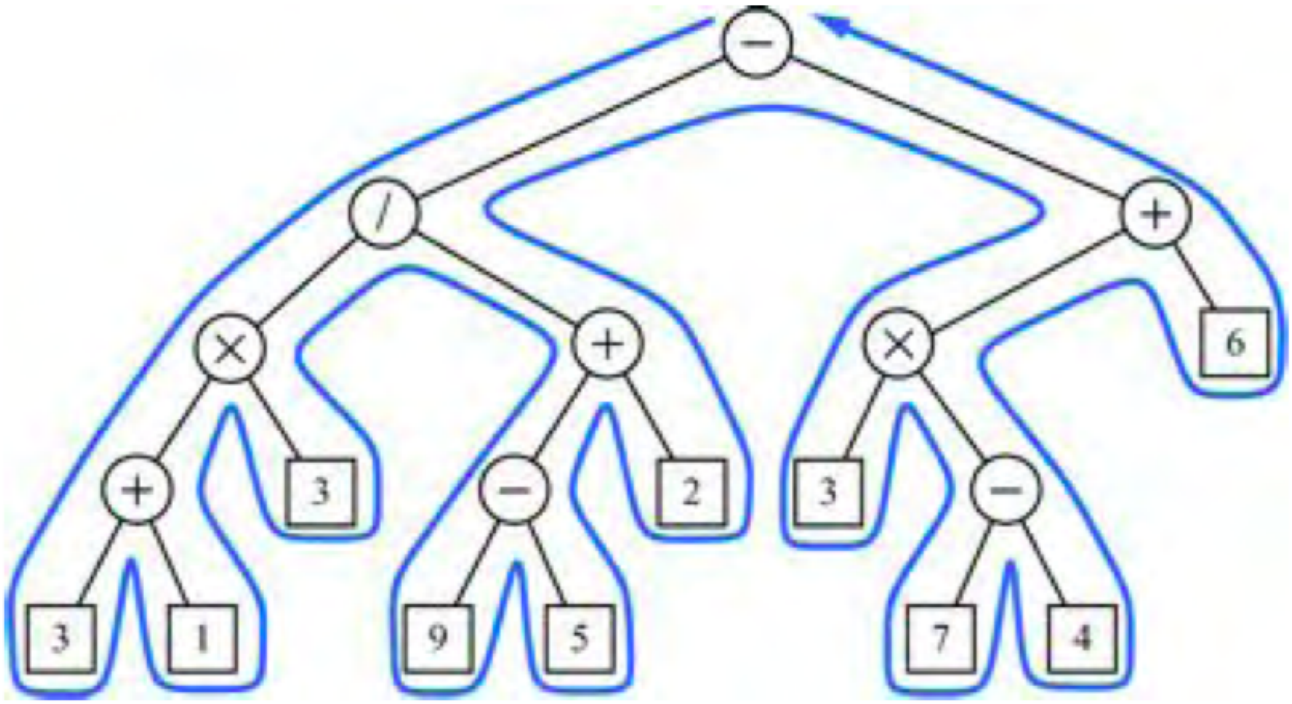


Figure 1: inorder traversal path (from Data Structure And Algorithms in Java p 424)