Fiche résumé nº 5 – Arbres et navigation

1. Arbres binaires avec possibilité de navigation

Le TAD vu précédemment est parfaitement utilisable pour gérer des arbres binaires. Cependant cette vision très « orientée fonctionnel » du TAD arbre ne permet que des parcours en profondeur.

Cette vue très fonctionnelle est comme si, pour parcourir les listes, nous ne disposions que des opérations *tete*(*liste*) et *reste*(*liste*) vues dans le chapitre des listes. C'est d'ailleurs justement le choix de base fait en langage fonctionnel Lisp (*List Processing*[1]) spécialisé pour les listes (avec ses 2 fonctions *car & cdr*[2]) puis dans d'autres langages avec *first & rest* ou *head & tail*, etc.

Dans le même esprit que pour les listes, il est alors naturel de souhaiter disposer d'opérations supplémentaires pour se déplacer librement dans les arbres, en ajoutant la notion de *nœud courant* : cette information, que l'on peut déplacer librement, représente la position où l'on se trouve actuellement dans l'arbre.

a) Navigation dans l'arbre : fonctions à ajouter au TAD précédent

Nouvelle fonctions ajoutées au TAD Arbre pour gérer une position courant dans l'arbre :

```
fonctions
    positRac :      Arbre[E] ——→ Arbre[E]

    positFilsG :    Arbre[E] —/→ Arbre[E]
    positFilsD :    Arbre[E] —/→ Arbre[E]
    positPère :     Arbre[E] —/→ Arbre[E]

    valNoeud :      Arbre[E] —/→ E

    aFilsG :        Arbre[E] ——→ Booléen
    aFilsD :        Arbre[E] ——→ Booléen
    aPère :         Arbre[E] ——→ Booléen

    horsArbre :     Arbre[E] ——→ Booléen

préconditions
    ∀a de type Arbre[E]
    pré(valNoeud(a))    = non horsArbre(a)
    pré(positFilsG(a)) = non horsArbre(a)
    pré(positFilsD(a)) = non horsArbre(a)
    pré(positPère(a))   = non horsArbre(a)
```

N.B. On peut imaginer étoffer le TAD d'autres fonctions utiles, notamment pour indiquer si la position courante est sur une feuille ou sur la racine, mais elles peuvent se construire à partir des fonctions existantes.

1 https://fr.wikipedia.org/wiki/Lisp
2 https://en.wikipedia.org/wiki/CAR_and_CDR

Exemple d'utilisation : réécriture différente possible des parcours en profondeur avec ce nouvel outillage.

```
algorithme parcoursInfixe(a : Arbre)
 si non estVide(a) alors
    a ← positRac(a)
    parcoursInfixeRec(a)   # Appelle le traitement auxiliaire récursif
 fin
fin

algorithme parcoursInfixeRec(a : Arbre)
 si aFilsG(a) alors
    a ← positFilsG(a)
    parcoursInfixeRec(a)
    a ← positPere(a)
 fin
 afficher(valNoeud(a)) # La visite peut consister en un autre traitement que afficher
 si aFilsD(a) alors
    a ← positFilsD(a)
    parcoursInfixeRec(a)
    a ← positPere(a)
 fin
fin
```

N.B. Les parcours préfixe et postfixe se déduisent en déplaçant la visite au début ou à la fin de la fonction auxiliaire.

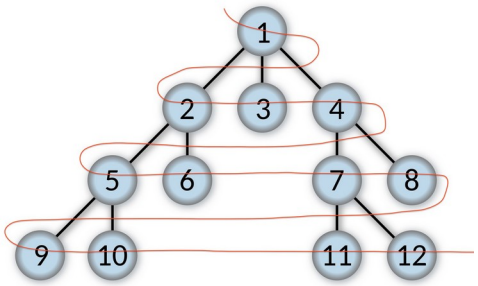
b) Modification de l'arbre : fonctions supplémentaires à ajouter au TAD

Pour pouvoir modifier l'arbre, à la position du nœud courant, ajoutons ces opérations supplémentaires :

```
fonctions
    modifNoeud :    Arbre[E]×E ⇸ Arbre[E]
    modifFilsG :    Arbre[E]×E ⇸ Arbre[E]
    modifFilsD :    Arbre[E]×E ⇸ Arbre[E]
    ôterNoeud :     Arbre[E] ⇸ Booléen
    ôterArbre :     Arbre[E] ⇸ Booléen

préconditions
    ∀ a de type Arbre[E], ∀ e de type E
    pré(modifNoeud(a,e))   = non horsArbre(a)
    pré(modifFilsG(a,e))   = non horsArbre(a)
    pré(modifFilsD(a,e))   = non horsArbre(a)
    pré(ôterArbre(a))      = non horsArbre(a)
    pré(ôterNoeud(a))      = non horsArbre(a) et non(aFilsG() et aFilsD())
```

N.B. La précondition pour ôter le nœud courant impose qu'il n'ait que 0 ou 1 fils pour pouvoir être réalisée : tout simplement parce que, dans le cas général, on se sait pas comment raccrocher les 2 branches qui en résulteraient si le nœud avait 2 fils.
Il peut cependant exister une solution même avec 2 fils lorsque l'on a plus d'information sur l'organisation de la structure de l'arbre : par exemple on peut raccrocher intelligemment le contenu des 2 branches lorsqu'il s'agit d'un arbre de recherche.

c) Parcours en largeur (*Breadth-First Traversal*) : fonctions ajoutées au TAD précédent



À la manière du parcours en largeur ci-dessus, il peut être intéressant de pouvoir aussi se déplacer librement dans dans le sens de la largeur et non uniquement dans le sens de la profondeur.
Pour ce faire, on ajoute les opérations suivantes :

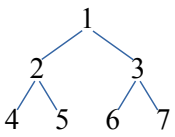
```
fonctions
    positFrèreG :   Arbre[E] –/→ Arbre[E]
    positFrèreD :   Arbre[E] –/→ Booléen
    aFrèreG :       Arbre[E] ——→ Booléen
    aFrèreD :       Arbre[E] ——→ Booléen

préconditions
    ∀a de type Arbre[E]
    pré(positFrèreG(a))    = non horsArbre(a)
    pré(positFrèreD(a))    = non horsArbre(a)
```

d) Réflexion sur la structure de données interne liée aux différents parcours

Intrinsèquement, un parcours en profondeur utilise une pile (LIFO : *Last In, First Out*). Une mise en œuvre classique comme nous l'avons déjà vue utilise en fait la pile d'exécution (appels récursifs) mais on peut expliciter cette pile et rendre son utilisation visible.

```
algorithme parcours avec pile (version préfixe)
 Pile[Noeud] lifo
 ajouter la racine de l'arbre dans lifo
 tant-que lifo est non vide faire
    sortir x de lifo
    afficher x
    si x possède un fils droit  alors ajouter ce filsD à lifo fin
    si x possède un fils gauche alors ajouter ce filsG à lifo fin
 fin
fin
```

Exercice : exécuter cet algorithme sur l'arbre suivant :



Pour généraliser le parcours précédent (préfixe) à tout parcours en profondeur, il ne faut plus empiler seulement les nœuds, mais aussi si c'est la 1re ou la 2e fois qu'on les rencontre.
=> on utilisera une pile de doublets (nœud, n° de visite)

```
algorithme parcours avec pile (version 2)
 Pile[doublet(Noeud,Entier)] lifo
 ajouter (racine,1) dans lifo
 tant-que lifo est non vide faire
    sortir (x,i) de lifo
    si i=2
      alors afficher x
      sinon
       si x possède un fils droit  alors ajouter (filsD,1) à lifo fin
       ajouter (x,2) à lifo
       si x possède un fils gauche alors ajouter (filsG,1) à lifo fin
    fin
 fin
fin
```

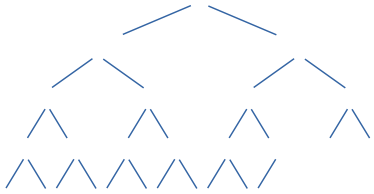
N.B. On peut déplacer l'empilement de (x,2) avant ou après le test des fils pour obtenir un parcours postfixe ou un parcours préfixe.

Pour obtenir un algorithme de parcours en largeur, il suffit de remplacer la pile (LIFO) du parcours préfixe par une file (FIFO : *First In, First Out*).

2. Mise en œuvre Java

a) Représentation contiguë

Un **arbre binaire complet à gauche** est un arbre binaire où tous les niveaux sont complets, sauf éventuellement le dernier niveau mais qui se complète en partant de la gauche.



La mise en œuvre d'un tel arbre peut se faire facilement dans un tableau :

```
public class ArbreCompletAG implements Arbre {
    private final static int N = 8;
    private Object[] val;       // stockage des valeurs des nœuds
    private int taille;         // nombre actuel de nœuds
    private int nc;             // position du nœud courant

    /** Constructeur d'arbre vide. */
    public ArbreCompletAG() {
        val = new Object[N];    // taille initiale qui pourra être augmentée
        taille = 0;
        positRac();             // initialise nc
    }
    // ...
}
```

Toute la suite des opérations de l'interface issue du TAD peut s'écrire en utilisant de simples calculs pour suivre les liens de parenté. En effet, en considérant la racine en case 1, les liens de parentés se déduisent facilement[3] pour un nœud stocké en case i :

- le père est en i / 2 (division entière)
- le fils gauche est en $2i$
- le fils droit est en $2i + 1$

AVANTAGE : l'adressage est calculé, et donc rapide

3 Exercice : adapter les formules pour une racine en case 0, qui est le premier indice en Java.

DÉFAUT : bien adapté seulement pour les arbres binaires complets à gauche.

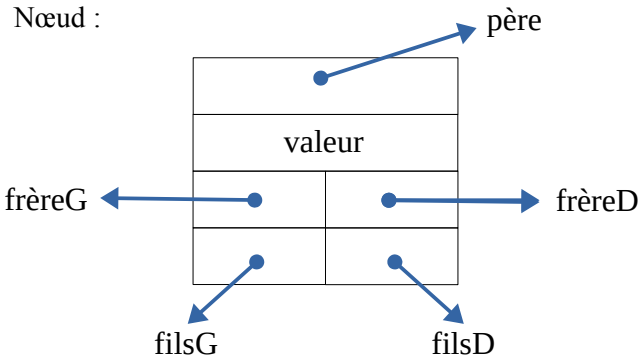
Cependant, on peut aussi utiliser cette mise en œuvre pour des arbres binaires quelconques :

- le principe d'adressage (×2 et /2) reste le même
- mais il apparaît de nombreux « trous » pour les nœuds absents.

=> c'est une perte rédhibitoire pour une structure arborescente car un arbre binaire de hauteur h nécessite jusqu'à $2^{h+1} - 1$ cases, et donc une branche inexistante rend inutile un grand nombre de cases, qui croît exponentiellement à mesure que l'arbre grandit !

b) Représentation chaînée

Pour éviter ces problèmes liés à la contiguïté des nœuds on met en place une structure chaînant les nœuds (selon un principe similaire aux listes chaînées) :




```
public class ArbreNoeudsChaînés implements Arbre {
    private Noeud racine;   // nœud racine
    private Noeud nc;       // nœud courant lorsqu'on se déplace dans l'arbre

    // Classe auxiliaire à usage interne
    private class Noeud {
        Object val;   // valeur du nœud
        Noeud pere, frereG, frereD, filsG, filsD; // nœuds des liens de parenté
    }

    /** Constructeur d'arbre vide. */
    public ArbreNoeudsChaînés() {
        racine  = null;
        positRac();   // initialise nc
    }
    // ...
}
```

Remarques :

- Bien souvent on omet les 2 références vers les frères si les déplacements en largeur ne sont pas utilisés. De plus, si comme dans le chapitre précédent, on n'utilise pas le mécanisme de navigation (qui peut remonter), alors on peut aussi se passer de la référence vers le père.
- Plutôt que ce nœud conteneur « à tout faire » on peut modéliser des types de nœuds plus spécialisés (par exemple par dérivation en programmation par objet), afin de décrire plus précisément l'arbre particulier considéré : des noeuds feuilles, des nœuds internes, des nœuds opérateurs, des nœuds valeurs, etc.
- Cette structure de données contient son propre mécanisme de navigation : on pourrait imaginer d'en séparer un équivalent du principe de l'itérateur mis en place pour parcourir une liste. Cependant comme il existe de nombreux parcours différents, un tel itérateur reste une mise en œuvre non classique. Ce qui s'en rapproche le plus serait le concept de Visiteur (cf. le futur cours de patrons de conception (*Design Patterns*) qui explicitera ce principe de classe externe qui fait un rapport sur le contenu d'une autre classe).