Fiche résumé nº 4 – Les arbres (binaires)

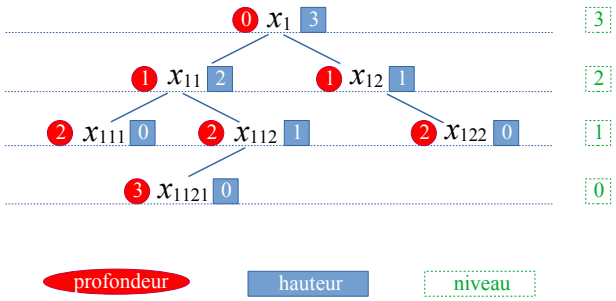
1. Arbres binaires (vue fonctionnelle)

a) Définitions

Un **arbre orienté** est une structure de données telle que :
- il existe un unique élément d'entrée : la **racine**.
- tout élément (sauf la racine) possède un et un seul prédécesseur : son **père**.
- on peut atteindre tout élément à partir de la racine par un **chemin unique**.

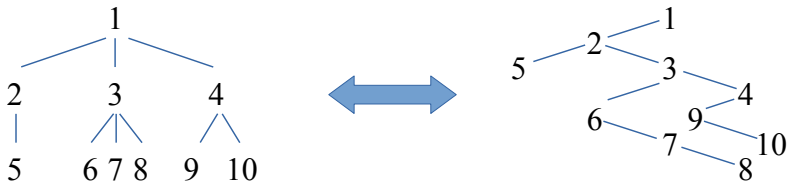
Un **arbre ordonné** est un arbre orienté tel que tous les successeurs d'un élément sont ordonnés :
- il existe la notion de 1er successeur, 2e successeur, etc.

Un **arbre binaire** est un arbre ordonné tel que tout élément a au plus 2 successeurs :
- ces 2 successeurs sont appelés fils gauche et fils droit.



Terminologie usuelle :
- x_i : les **nœuds** de l'arbre
- x_1 : le nœud **racine** de l'arbre
- x_{111}, x_{1121}, x_{122} : les nœuds **feuilles** de l'arbre
- tout chemin de la racine à une feuille est appelé **branche**, constituée d'**arêtes** pour relier les nœuds
- **profondeur**(x) = longueur du chemin, en nombre d'arêtes, de la racine à x
- **hauteur**(x) = longueur maximale des chemins allant de x à une feuille
- par extension, la **hauteur d'un arbre** = hauteur(racine)
- **niveau**(x) = hauteur(racine) - profondeur(x)

Remarque :
Limiter l'étude de cette structure de données arbre au cas binaire n'est pas réducteur, car il existe un procédé bijectif qui transforme un arbre quelconque en arbre binaire. Le principe général à répéter en toute position se résume comme suit :
- on place les fils d'un nœud successivement dans le sous-arbre gauche de ce nœud
- on place les frères successivement dans le sous-arbre droit de ce nœud



b) TAD Arbre

```
sorte Arbre[E]
utilise Booléen
fonctions
    new :          ⟶ Arbre[E]
    new :          E×Arbre[E]×Arbre[E] ⟶ Arbre[E]
    estVide :      Arbre[E] ⟶ Booléen
    vider :        Arbre[E] ⟶ Arbre[E]
    racine :       Arbre[E] -/→ E
    arbreG :       Arbre[E] ⟶ Arbre[E]
    arbreD :       Arbre[E] ⟶ Arbre[E]
    modifRacine : Arbre[E]×E ⟶ Arbre[E]
    modifArbreG : Arbre[E]×Arbre[E] ⟶ Arbre[E]
    modifArbreD : Arbre[E]×Arbre[E] ⟶ Arbre[E]

préconditions
    ∀ a de type Arbre[E]
    pré(racine(a)) = non estVide(a)

axiomes
    ∀ a,g,d de type Arbre[E], ∀ r de type E
    racine(new(r,g,d)) = r
    arbreG(new(r,g,d)) = g
    arbreD(new(r,g,d)) = d

    estVide(new) = VRAI
    estVide(new(r,g,d)) = FAUX
    estVide(vider(a)) = VRAI
    estVide(arbreG(new)) = VRAI
    estVide(arbreD(new)) = VRAI

    racine(modifRacine(a,r)) = r
    arbreG(modifArbreG(a,g)) = g
    arbreD(modifArbreD(a,d)) = d
```

N.B. Du fait des 2 seuls créateurs disponibles dans les fonctions, on constate que cette spécification implique que toute feuille a un arbre vide comme fils gauche et comme fils droit. C'est un choix classique pour une telle vision fonctionnelle des arbres, qui rend l'utilisation plus pratique (on peut par exemple demander sans erreur un sous-arbre fils d'une feuille et on obtient un arbre vide).

Cependant, un autre point de vue révise parfois cette spécification du TAD pour faciliter des mises en œuvre plus économes, mais moins pratiques (pas d'arbres vides sous chaque feuille), via 3 modifications :

1. une nouvelle fonction pour un 3e créateur d'arbre réduit à une feuille
2. des préconditions ajoutées pour arbreG et arbreD qui deviennent des fonctions partielles
3. des axiomes modifiés pour spécifier le nouveau créateur, et aussi pour supprimer la spécification vide des sous-arbres fils.

2. Exemples d'utilisation

a) Calcul de la hauteur d'un arbre

```
algorithme hauteur(a : Arbre) retourne entier
    si estVide(a)
          alors retourner -1
          sinon retourner 1 + max(hauteur(arbreG(a)),hauteur(arbreD(a)))
    fin
fin
```

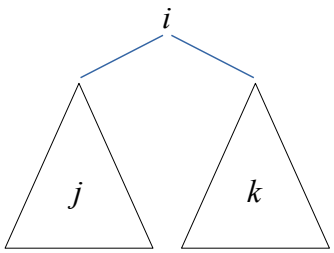
Remarques :

- La valeur terminale de -1 pour le cas d'arrêt de la récursivité (et non 0) est celle nécessaire pour fournir le bon résultat conforme à la définition.
- Dans la définition, la hauteur d'un arbre vide est inconnue et non définie. Donc un tel algorithme devrait être encapsulé dans une fonction supérieure filtrant en erreur l'appel avec un arbre vide.

b) Parcours d'arbres en profondeur (*Depth-first Traversal*)

Il existe 3 parcours classiques d'arbres, en profondeur : préfixe, infixe, postfixe.

Le parcours préfixe (*Pre-order Traversal*) :



∀ nœud $i \in$ arbre
∀ nœud $j \in$ sous-arbre gauche de i
∀ nœud $k \in$ sous-arbre droit de i

on a visite(i) < visite(j) < visite(k)

N.B. Le signe < signifie ici « est avant ».

Le parcours infixe (*In-order Traversal*) :
Avec les mêmes notations, on a visite(j) < visite(i) < visite(k)

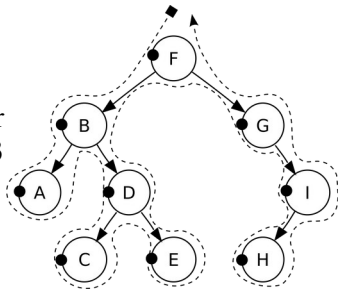
Le parcours postfixe (*Post-order Traversal*) :
Avec les mêmes notations, on a visite(j) < visite(k) < visite(i)

```
algorithme parcoursInfixe(a : Arbre)
    si non estVide(a) alors
          parcoursInfixe(arbreG(a))
          afficher(racine(a))      # La visite peut consister en un autre traitement que afficher
          parcoursInfixe(arbreD(a))
    fin
fin
```

Remarques :

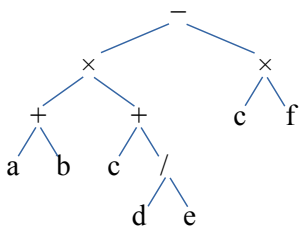
- On note que cet algorithme exploite le fait que notre TAD fournit des arbres vides en fils des feuilles.
- Les 2 autres algorithmes se déduisent tout simplement en déplaçant la visite au début ou à la fin du bloc d'instructions.

Exemple : le parcours qui visite les nœuds de l'arbre ci-contre par ordre alphabétique est-il quelconque ou correspond-il à un des 3 classiques ?



c) Exercices sur les expressions[1]

Structurer l'expression suivante sous forme d'arbre binaire : $((a+b)\times(c+(d/e)))-(c\times f)$



Donner l'énumération des nœuds de l'arbre selon les 3 parcours :

- préfixe => – × + a b + c / d e × c f
- infixe => a + b × c + d / e – c × f
- postfixe => a b + c d e / + × c f × –

Remarques :

- Aux parenthèses près, pour régler la priorité des calculs, l'écriture infixe est la **notation mathématique** usuelle
- En ajoutant des parenthèses aussi (et des séparateurs virgules non utiles), l'écriture préfixe est la **notation fonctionnelle** usuelle : sub(mul(add(a,b),add(c,div(d,e))) , mul(c,f))
- L'écriture postfixe est celle nommée **notation polonaise inverse** : on constate qu'elle exprime l'expression sans ambiguïté sans nécessiter de parenthèses, ce qui la rend rapide et compacte : c'est ce qui a fait le succès à l'origine des calculatrices Hewlett-Packard.

d) Arbres de recherche

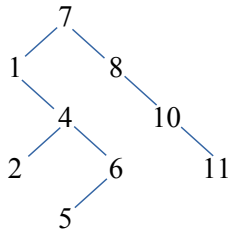
Un arbre binaire de recherche est un cas particulier où les valeurs possèdent une relation d'ordre, qui est exploitée pour les placer aux nœuds de l'arbre en respectant la règle suivante :

∀ nœud $x \in$ arbre :

- $x_i \in$ sous-arbre gauche de $x \quad \Rightarrow x_i \leq x$
- $x_j \in$ sous-arbre droit de $x \quad \Rightarrow x_j \geq x$

N.B. Quand les valeurs sont égales, on peut fixer un côté privilégié.

Par exemple, l'ajout successif des valeurs 7, 1, 8, 4, 6, 10, 11, 2, 5, donne :



Remarques :

- Un simple parcours infixe d'un arbre de recherche fournit les valeurs triées dans l'ordre.
- On note que l'ajout ou la recherche dans un tel arbre sont proportionnels à la hauteur de l'arbre. Ces actions seront donc optimisées si l'arbre est bien équilibré (c'est-à-dire que toutes ses branches sont de mêmes longueurs, ou sont plus longues de seulement 1 arête).

1 Des mises en œuvre d'arbres binaires en Java sont proposées en fin de chapitre suivant.