Chapitre 7 : Introduction aux arbres

1 Les arbres comme objets mathématiques

1.1 Définitions

Définition 1. *Un arbre A est un ensemble non vide muni d'une relation binaire $\prec$ vérifiant :*

- $\exists! r \in A \quad \forall x \in A \quad \neg(r \prec x)$. *L'élément r s'appelle la racine de l'arbre.*
- $\forall x \in A \backslash \{r\} \quad \exists! t \in A \quad x \prec y$. *On dit que y est* le *parent (ou père) de x, et x est* un *fils de y.*
- $\forall x \in A \backslash \{r\} \quad \exists n > 0 \quad \exists (x_1, \ldots, x_n) \in A^n \quad x \prec x_1 \prec x_2 \prec \cdots \prec x_n = r$.

La relation binaire $x \prec y$ signifie que x est un enfant de y. Les conditions peuvent se résumer ainsi : mis à part la racine r, chaque élément a un unique parent, et en suivant ces liens de parenté on aboutit à la racine. Visuellement, on représente un arbre avec des nœuds reliés par des arêtes, la racine de l'arbre étant situé tout en haut [1] Par exemple :

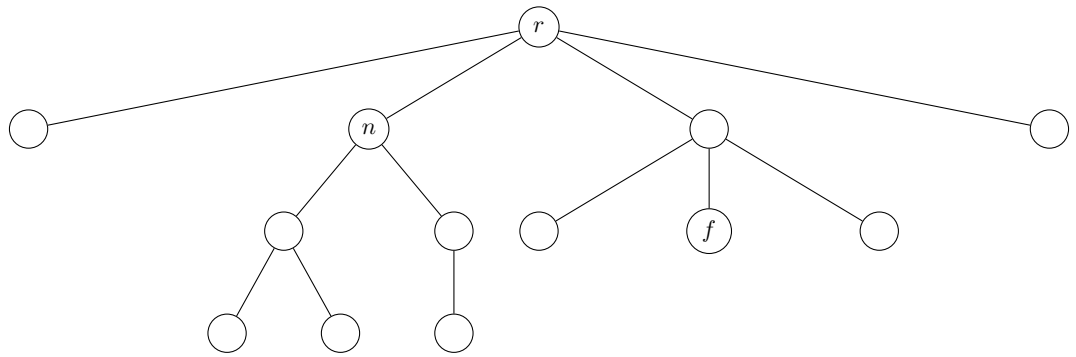

FIGURE 1: Un arbre

Définition 2 (feuilles et nœuds internes)**.** *Les éléments de A sont appelés les* nœuds *de l'arbre. Pour x un nœud, on appelle* arité *de x le nombre de fils de x. Un nœud d'arité 0 est appelé une* feuille*, sinon c'est un* nœud interne.

Par exemple en figure 1, n est un nœud interne, f est une feuille. On étudiera plus particulièrement les arbres évoqués dans la définition suivante.

Définition 3 (arbre binaire)**.** *On appelle arbre binaire un arbre dont les nœuds sont d'arité* au plus deux*, et arbre binaire entier un arbre binaire dont les nœuds sont tous d'arité* zéro ou deux*.*

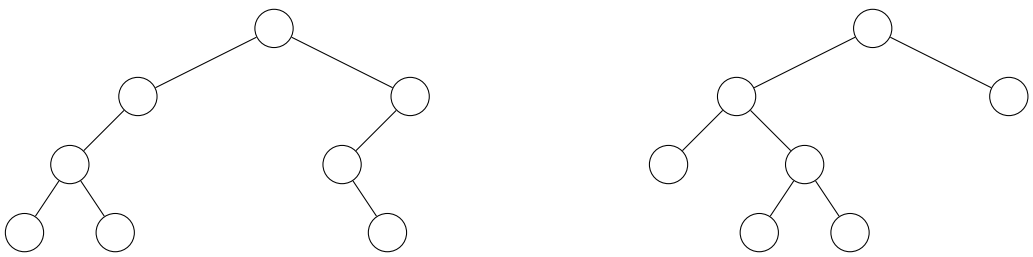

FIGURE 2: Un arbre binaire, et un arbre binaire entier

Définition 4 (Profondeur et hauteur)**.** *Avec r la racine d'un arbre A et x un de ses nœuds, on a vu qu'il existait un unqiue entier $n \geq 0$ et d'uniques nœuds $x_1, \ldots, x_{n-1}$ tels que $x \prec x_1 \prec \cdots \prec x_{n-1} \prec x_n = r$.*

1. En informatique, les arbres poussent de haut en bas.

— *On appelle* profondeur *de x l'entier $n \geq 0$;*

— *On appelle* hauteur *d'un arbre la profondeur maximale de ses nœuds.*

Exemple 5. *La racine est l'unique nœud à profondeur 0. Les arbres des figures 1 et 2 ont tous hauteur 3.*

Définition 6 (sous-arbre enraciné)**.** *Soit x un nœud d'un arbre A. On considère l'ensemble*

$$A_x = \{y \in A, \exists n \in \mathbb{N}, \exists x_1, \ldots, x_{n-1} \in A \mid y = x_0 \prec x_1 \prec \cdots \prec x_n = x\}$$

Alors on vérifie aisément que la restriction de $\prec$ à A_x munit A_x d'une structure d'arbre, de racine x. Cet arbre se nomme le sous-arbre de A enraciné en x. On dit aussi que les éléments de A_x forment la descendance de x dans A.

1.2 Un peu de dénombrement

1.2.1 Inégalités entre hauteur et nombre de nœuds

On donne ici des encadrements faisant intervenir la hauteur et le nombre de nœuds d'un arbre, en fonction de l'arité maximale des nœuds.

Proposition 7. *Pour A un arbre de hauteur h dont les nœuds sont d'arité au plus a, le nombre n de nœuds de l'arbre vérifie si $a > 1$:*

$$h + 1 \leq n \leq \frac{a^{h+1} - 1}{a - 1}$$

Démonstration. — Considérons un nœud à profondeur maximale h. Sur le chemin de ce nœud à la racine, il y a $h + 1$ nœuds, d'où $n \geq h + 1$. Avec a l'arité maximale, on montre aisément par récurrence qu'il y a au plus a^p nœuds à profondeur p. L'autre inégalité s'obtient donc par somme : $\sum_{p=0}^{h} a^p = \frac{a^{h+1}-1}{a-1}$.

□

Remarque 8. *Si un arbre est d'arité maximale 1, il a exactement $h + 1$ nœuds, avec h sa hauteur.*

Corollaire 9. *La hauteur h d'un arbre à n nœuds tous d'arité au plus $a > 1$ vérifie*

$$\log_a((a-1)n + 1) - 1 \leq h \leq n - 1$$

Appliquons ce résultat dans le cas $a = 2$:

Corollaire 10. *Soit A un arbre binaire à n nœuds. Sa hauteur h vérifie $\lfloor \log_2(n) \rfloor \leq h \leq n - 1$*

Démonstration. On prend donc $a = 2$ dans le corollaire précédent. Ainsi $h + 1 \geq \log_2(n + 1) > \log_2(n) \geq \lfloor \log_2(n) \rfloor$. Ainsi $h + 1$ est un entier strictement supérieur à l'entier $\lfloor \log_2(n) \rfloor$, donc $h \geq \lfloor \log_2(n) \rfloor$. □

1.2.2 Feuilles et nœuds internes dans un arbre binaire

Proposition 11. *Un arbre binaire entier ayant p nœuds internes possède $p + 1$ feuilles.*

Démonstration. La démonstration se fait par récurrence forte (sur p par exemple).

— Si $p = 0$, l'arbre a une seule feuille (sa racine) donc la relation est vérifiée.

— Sinon soit $p > 0$. Considérons un arbre A ayant $p > 0$ nœuds internes. La racine étant un nœud interne, notons alors n_g et n_d le nombre de nœuds internes des sous-arbres enracinés en les deux fils de la racine. Ces arbres sont également binaires entiers, et vérifient $n_g < p$ et $n_d < p$, donc par hypothèse de récurrence, ils ont respectivement $n_g + 1$ et $n_d + 1$ feuilles. Dans l'arbre A, il y a donc $n_g + n_d + 1$ nœuds internes et $(n_g + n_d + 1) + 1$ feuilles, donc la propriété est vraie pour un arbre de hauteur p ;

— Par principe de récurrence, la propriété est démontrée.

□

Corollaire 12. *Dans un arbre binaire à p nœuds internes, il y a au plus $p + 1$ feuilles.*

Démonstration. Rajouter un fils (une feuille) aux nœuds d'arité 1 transforme l'arbre en arbre binaire entier, sans changer le nombre de nœuds internes. □

2 Les arbres en Caml

En informatique, les arbres sont utilisés pour stocker de l'information : à chaque nœud est attaché une *étiquette*, qui peut être un entier, une chaîne de caractères, voire même un couple (voir chapitre suivant). De plus, les fils d'un nœud sont en général ordonnés : par exemple pour un arbre binaire entier, on parlera du fils gauche et du fils droit d'un nœud interne. On propose dans cette section une implémentation *persistante* des arbres : comme pour les listes chaînées, les fonctions sur la structure d'arbre renverront de nouveaux arbres plutôt que de les modifier (une implémentation impérative sera vue au chapitre suivant).

2.1 Arbres généraux

Pour des arbres généraux, on peut représenter un nœud par la liste de ses fils, qui sont eux même des arbres. On distinguera alors une feuille et un nœud interne suivant si la liste des fils est vide ou non. Pour que les nœuds de l'arbre puissent porter des étiquettes d'un certain type, on peut définir le type (récursif) suivant :

```
type 'a arbre = N of 'a * 'a arbre list;;
```

Voici un exemple d'arbre à étiquettes entières :

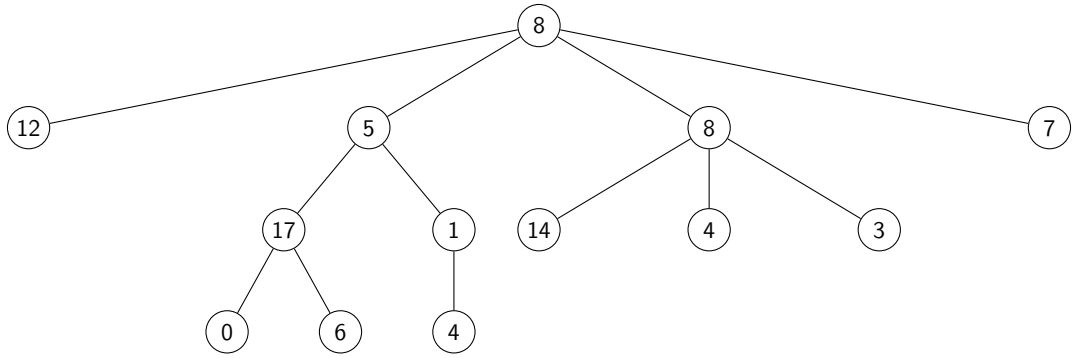

FIGURE 3: Un arbre à étiquettes entières

En Caml cet arbre est implémenté comme suit :

```
#ex_arbre ;;
- : int arbre =
 N (8,
    [N (12, []);
     N (5, [N (17, [N (0, []); N (6, [])]); N (1, [N (4, [])])]);
     N (8, [N (14, []); N (4, []); N (3, [])]);
     N (7, [])])
```

Pour parcourir un tel arbre, on utilise en général deux fonctions : l'une qui prend en entrée un arbre, et l'autre une liste d'arbres. Elles vont s'appeler l'une l'autre et donc être mutuellement récursives. Voici un exemple de parcours, pour calculer la hauteur d'un tel arbre :

```
let rec hauteur a=match a with
  | N(_,q) -> 1+max_h q
and max_h q=match q with
  | [] -> -1
  | x::p -> max (hauteur x) (max_h p)
;;
```

Si on veut faire la distinction entre feuilles et nœuds internes (et leur donner des étiquettes de type différents), on peut utiliser par exemple le type suivant.

```
type ('a, 'b) arbre = F of 'a | N of 'b * ('a, 'b) arbre list;;
```

2.2 Arbres binaires entiers

En informatique, les arbres sont souvent des arbres binaires (entiers ou non). On va donc utiliser une implémentation un peu moins générale que celle de la section précédente. Pour les arbres binaires entiers, un arbre (informatique) est :

— soit une feuille ;

— soit la donnée d'une étiquette, et de deux arbres (ses sous-arbres gauche et droit).

En suivant cette description, on obtient le type suivant, où l'on distingue les étiquettes des feuilles et des nœuds internes.

```
type ('a, 'b) arbre =  F of 'a | N of 'b*('a, 'b) arbre * ('a, 'b) arbre;;
```

Voici un exemple d'utilisation : une expression arithmétique se représente naturellement par un arbre binaire entier, les étiquettes des nœuds internes étant les opérateurs et celles des feuilles étant les opérandes, voir figure 4.

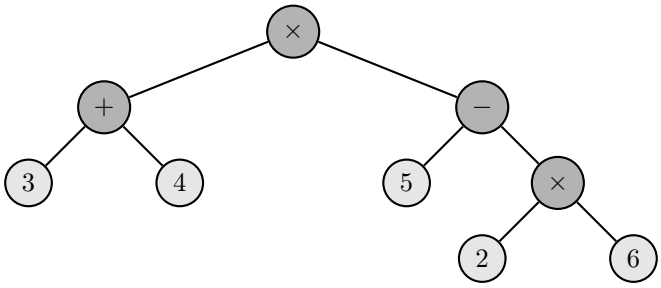

FIGURE 4: L'arbre associé à l'expression arithmétique $(3+4)\times(5-(2\times 6))$.

En Caml on représente l'expression de la figure 4 comme :

```
type op = Plus | Moins | Fois | Div | Mod ;;
let expr = N (Fois, N (Plus, F 3, F 4), N (Moins, F 5, N (Fois, F 2, F 6))) ;;
```

On a utilisé un type énuméré pour définir les opérateurs. L'évaluation d'une telle expression se fait récursivement, par filtrage :

```
let rec evalue e=match e with
  | F x -> x
  | N (op, a, b) -> (traduit op) (evalue a) (evalue b)
;;
```

Où la fonction `traduit` renvoie la fonction `int -> int -> int` associée à l'opérateur `op`, qu'on laisse au lecteur le soin d'écrire. Testons :

```
# evalue ;;
- : (int, op) arbre -> int = <fun>
# evalue expr ;;
- : int = -49
```

2.3 Arbres binaires

On propose une implémentation des arbres binaires qui sera reprise dans le chapitre suivant, pour l'implémentation des arbres binaires de recherche. Elle est très proche de l'implémentation précédente des arbres binaires entiers : en effet, si on considère un arbre binaire, et qu'en chaque nœud x on fait pousser $2-a$ fils où a est l'arité de x, on obtient un arbre binaire entier. On note « Vide » les nœuds que l'on fait pousser, de sorte que les feuilles du nouvel arbre sont toutes « Vide ».

Avec cette représentation, un arbre binaire est :

— soit vide ;

— soit la donnée d'une étiquette, et deux deux sous-arbres binaires.

Ceci mène à la définition du type suivant :

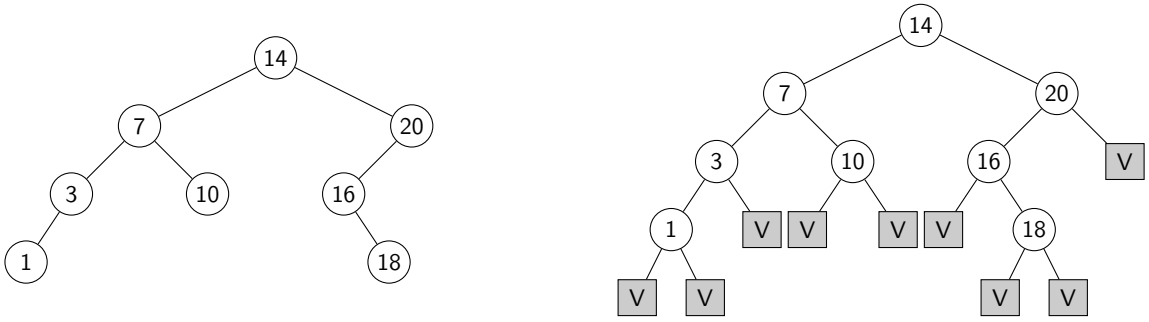

FIGURE 5: Un arbre binaire à étiquettes entières, et l'arbre binaire entier associé.

```
type 'a arbre = Vide | N of 'a * 'a arbre * 'a arbre ;;
```

Voici l'implémentation de l'arbre correspondant à l'exemple de la figure 5 :

```
let ex_ab = N (14,
   N(7, N(3, N(1, Vide, Vide), Vide), N(10, Vide, Vide)),
   N(20, N(16, Vide, N(18, Vide, Vide)), Vide)) ;;
```

Donnons pour terminer une fonction permettant de calculer la hauteur [2] d'un tel arbre :

```
let rec hauteur a=match a with
  | Vide -> -1
  | N(_,g,d) -> 1 + max (hauteur g) (hauteur d)
;;
```

Par exemple :

```
#hauteur ex_ab ;;
- : int = 3
```

3 Parcours d'arbres binaires entiers

On veut énumérer les nœuds d'un arbre binaire entier, de même type que défini en sous-section 2.2. On va stocker le résultat dans une liste. On introduit un type pour stocker dans une même liste les étiquettes des nœuds internes et des feuilles :

```
type ('a, 'b) etiq = A of 'a | B of 'b ;;
```

3.1 Parcours en profondeur

Le parcours en profondeur d'un arbre consiste à s'enfoncer le plus possible dans l'arbre avant de revenir en arrière explorer les autres branches. Pour un arbre binaire entier de racine r, et de sous-arbres gauche et droit g et d, il y a trois choix naturels :

— racine r, enumération de g, énumération de d : on parle de parcours préfixe ;

— enumération de g, racine r, énumération de d : on parle de parcours infixe ;

— enumération de g, énumération de d, racine r : on parle de parcours postfixe (ou suffixe).

Voici une écriture du parcours préfixe :

```
let rec prefixe a=match a with
  | F x -> [B x]
  | N(x,g,d) -> (A x)::(prefixe g)@(prefixe d)
;;
```

2. On convienr que l'arbre « Vide » a pour hauteur -1 : ceci est cohérent avec le fait que les feuilles « Vide » ne font pas vraiment partie de l'arbre.

Testons avec l'expression arithmétique de la figure 4.

```
# prefixe expr ;;
- : (op, int) etiq list =
[A Fois; A Plus; B 3; B 4; A Moins; B 5; A Fois; B 2; B 6]
```

On peut de même écrire des fonctions `infixe` et `postfixe` :

```
# infixe expr ;;
- : (op, int) etiq list =
[B 3; A Plus; B 4; A Fois; B 5; A Moins; B 2; A Fois; B 6]
# postfixe expr ;;
- : (op, int) etiq list =
[B 3; B 4; A Plus; B 5; B 2; B 6; A Fois; A Moins; A Fois]
```

Il est notable de voir que l'énumération donnée par le parcours préfixe ou le parcours postfixe permet de reconstruire l'arbre, contrairement à celle du parcours infixe.

Proposition 13. *Si les étiquettes des nœuds internes et des feuilles ont des types différents, alors à une énumération préfixe ou postfixe correspond un seul arbre binaire entier.*

Voici une preuve dans le cas de l'énumération postfixe, on laisse au lecteur le soin de l'adapter à l'énumération préfixe. Elle débute par un lemme.

Lemme 14. *Considérons l'énumération postfixe d'un arbre binaire entier. On parcourt l'énumération avec un compteur s initialisé à 0, on ajoute $+1$ pour une feuille, et -1 pour un nœud interne. Alors s est toujours strictement positif après le début de l'énumération, et termine par* 1.

Démonstration. Le lemme se démontre par récurrence sur la longueur de l'énumération.

- pour une énumération de longueur 1 (correspondant à une unique feuille), c'est immédiat ;
- sinon, l'énumération est constituée de l'énumération du sous-arbre gauche (par récurrence, s est toujours strictement positif et termine à 1), celle du sous-arbre droit (s reste supérieur à 1 et termine à 2), et enfin la racine (s termine à 1).
- Par principe de récurrence, le lemme est démontrée.

□

Preuve de la proposition 13. On procède par récurrence sur la longueur de l'énumération.

- Une énumération de taille 1 correspond à un arbre à une unique feuille, l'unicité est donc évidente.
- Donnons nous maintenant une énumération postfixe d'un arbre binaire entier, de taille strictement supérieure à 1, et supposons la propriété démontrée pour des énumérations plus petites. Par nature de l'énumération, se trouve d'abord toute l'énumération du sous-arbre gauche, puis celle du sous-arbre droit, et enfin la racine de l'arbre. Pour pouvoir appliquer l'hypothèse de récurrence et terminer la preuve, il suffit de savoir où se situe la frontière entre les énumérations des sous-arbres gauche et droit. Or, on déduit du lemme que cette frontière se situe juste après que le compteur s du lemme ait pris la valeur 1 pour la dernière fois (avant la racine) : on peut donc reconstruire l'arbre par hypothèse de récurrence.
- Par principe de récurrence, la propriété est démontrée.

□

Remarque 15. *Cette propriété sur l'énumération postfixe a été utilisée dans certaines calculatrices*[3]*, et s'étend à des expressions faisant usage d'opérateurs d'arité différente de 2. L'intérêt est que les parenthèses sont inutiles pour donner l'expression arithmétique, ce qui fournit un gain de temps à l'utilisateur. Une pile suffit pour écrire une fonction d'évaluation d'une expression donnée sous la forme du parcours postfixe, qu'on laisse au lecteur le soin d'implémenter :*

```
#evalue_postfixe (postfixe expr) ;;
- : int = -49
```

Remarque 16. *L'énumération infixe ne suffit pas pour reconstruire l'arbre, comme le montre l'exemple des deux arbres ci-dessous, ayant même énumération :*

3. À notation *polonaise inversée*, qui est un autre nom pour l'énumération postfixe de l'arbre associé à l'expression.
4. La priorité de $\times$ sur $+$ et $-$ permet de s'affranchir de certaines parenthèses... Mais ce n'est qu'une convention !

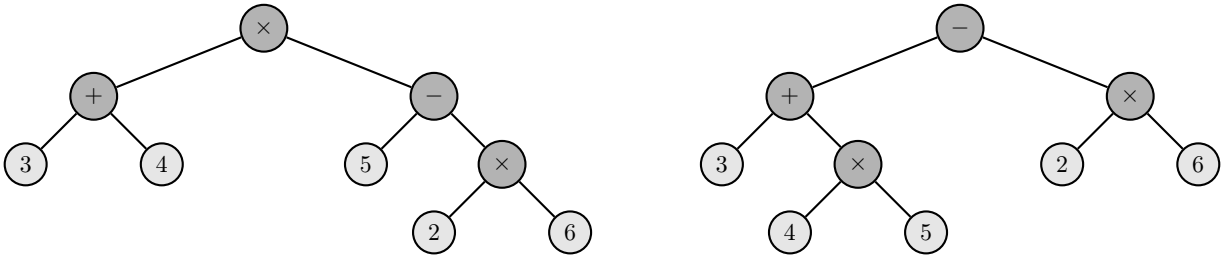

FIGURE 6: Deux arbres d'énumération infixe $3 + 4 \times 5 - 2 \times 6$: les parenthèses sont obligatoires [4]pour donner un sens à l'expression.

3.2 Parcours en largeur

Contrairement au parcours en profondeur, le parcours en largeur liste les nœuds par profondeur croissante. Par convention, les nœuds situés à gauche sont énumérés en premier. L'énumération de ce parcours est un peu plus délicate à obtenir qu'une énumération d'un parcours en profondeur. On commence par écrire deux fonctions qui prennent en entrée une liste d'arbres, et renvoient respectivement l'énumération de leurs racines, et la liste de leurs sous-arbres.

```
let rec racines q=match q with
    | [] -> []
    | (F x)::p -> (B x)::(racines p)
    | N(x,_,_)::p -> (A x)::(racines p)
;;

let rec ss_arbres q=match q with
  | [] -> []
  | (F _)::p -> ss_arbres p
  | N(_,g,d)::p -> g::d::ss_arbres p
;;
```

On peut maintenant écrire une fonction de parcours en largeur :

```
let largeur p=
  let rec aux q=match q with
    | [] -> []
    | _ -> (racines q)@(aux (ss_arbres q))
  in aux [p]
;;
```

La fonction `aux` de `largeur` renvoie l'énumération « en largeur » d'une liste d'arbres : si la liste est non vide, elle énumère les racines, et se rappelle récursivement sur la liste des sous-arbres. `largeur` se contente d'un unique appel à `aux`. Testons :

```
#largeur expr ;;
- : (op, int) etiq list =
[A Fois; A Plus; A Moins; B 3; B 4; B 5; A Fois; B 2; B 6]
```

Remarque 17. *Les parcours en profondeur préfixe et postfixe, ainsi que le parcours en largeur, se généralisent à d'autres arbres que les arbres binaires entiers, avec des fonctions assez semblables à celles vues dans ce chapitre. Le parcours infixe se généralise à des arbres binaires (non nécessairement binaires entiers), mais pas à des arbres quelconques.*