Arbre binaire
Notation polonaise

Christophe Viroulaud

Terminale - NSI

Algo 06

En 1920 le mathématicien Jan Łukasiewicz présente la *notation polonaise* qui permet d'exprimer des expressions mathématiques sans utiliser de parenthèse, mais traitant néanmoins toute formule sans ambiguïté.
L'expression arithmétique

$$2 \times (3 + 4)$$

devient en notation polonaise

$$\times\ 2 + 3\ 4$$

Dans les années 50, Charles L. Hamblin s'intéresse à variante *inversée* de cette notation. Elle est en effet particulièrement bien adaptée à la manière dont les processeurs traitent leurs opérandes. En notation polonaise inversée, l'expression précédente s'écrit

$$2\ 3\ 4\ +\ \times$$

Déterminer une structure de données adaptée au calcul en notation polonaise inversée.

Sommaire

Arbre binaire - définition

À retenir

Un **arbre binaire** est une structure arborescente où chaque nœud possède **au plus** deux fils. L'ordre des nœuds-fils est pris en compte : on parle alors de fils *gauche* et fils *droit*.

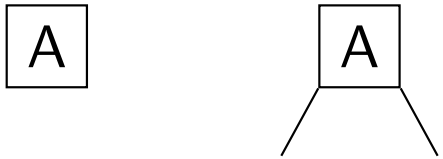

FIGURE 1 – Représentations d'un nœud

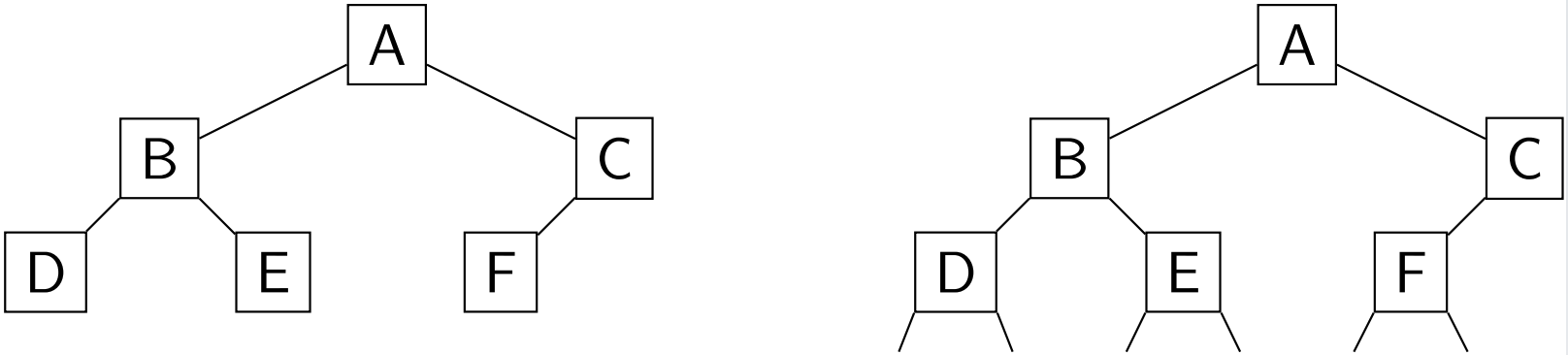

FIGURE 2 – Représentations d'un arbre binaire

Sommaire

À retenir

Un arbre binaire est **complet** si tous les niveaux sont remplis sauf éventuellement le dernier ; les feuilles sont alors *tassées à gauche*

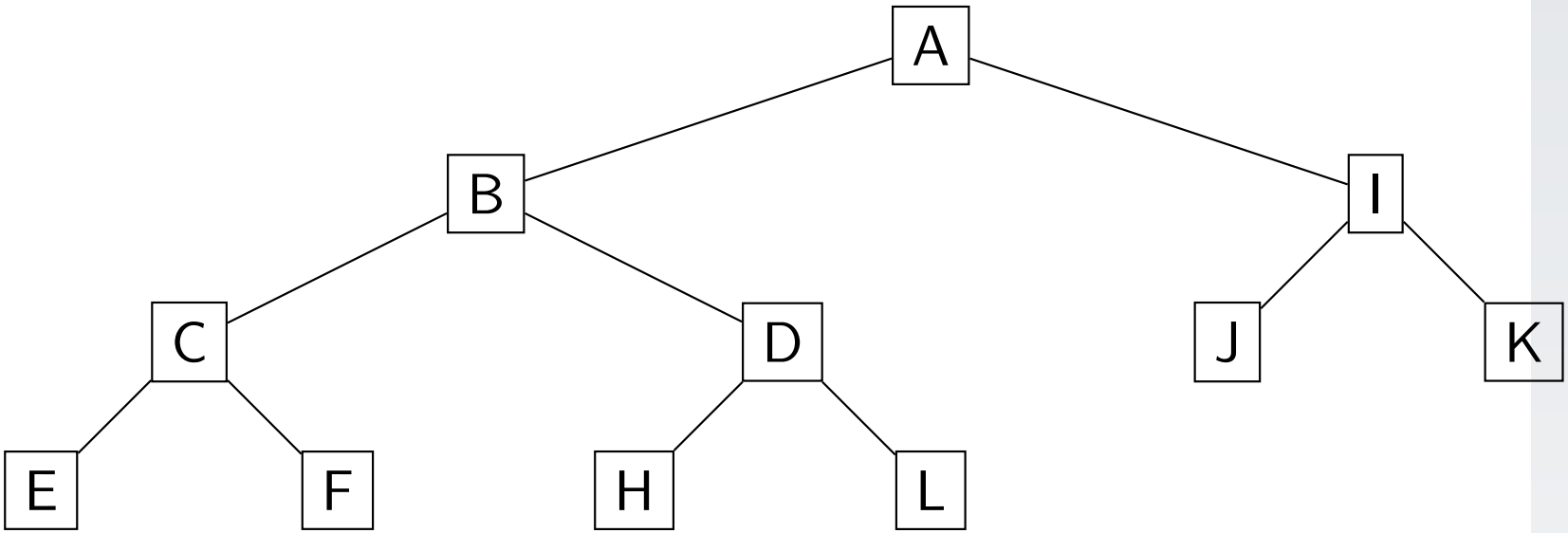

FIGURE 3 – Un arbre binaire complet

À retenir

Un arbre binaire est **équilibré** si pour chaque nœud interne, les *sous-arbres gauche et droite* ont une hauteur qui diffère au plus de 1.

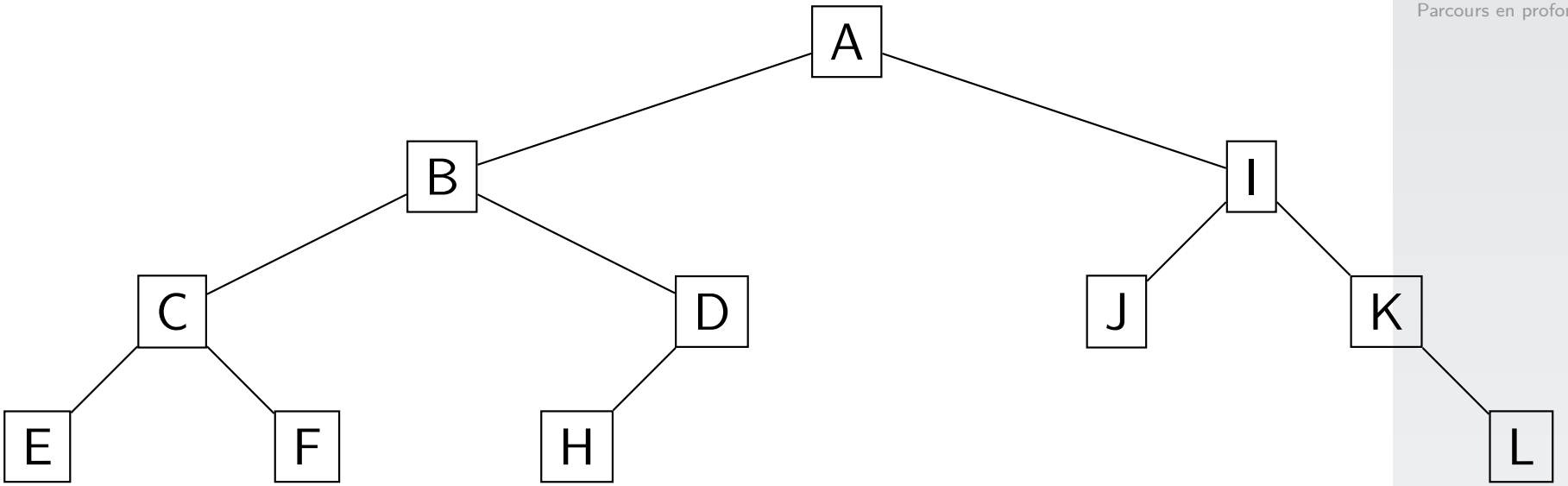

FIGURE 4 – Un arbre binaire équilibré non complet

À retenir

Un arbre binaire est **parfait** si tous les niveaux sont remplis.

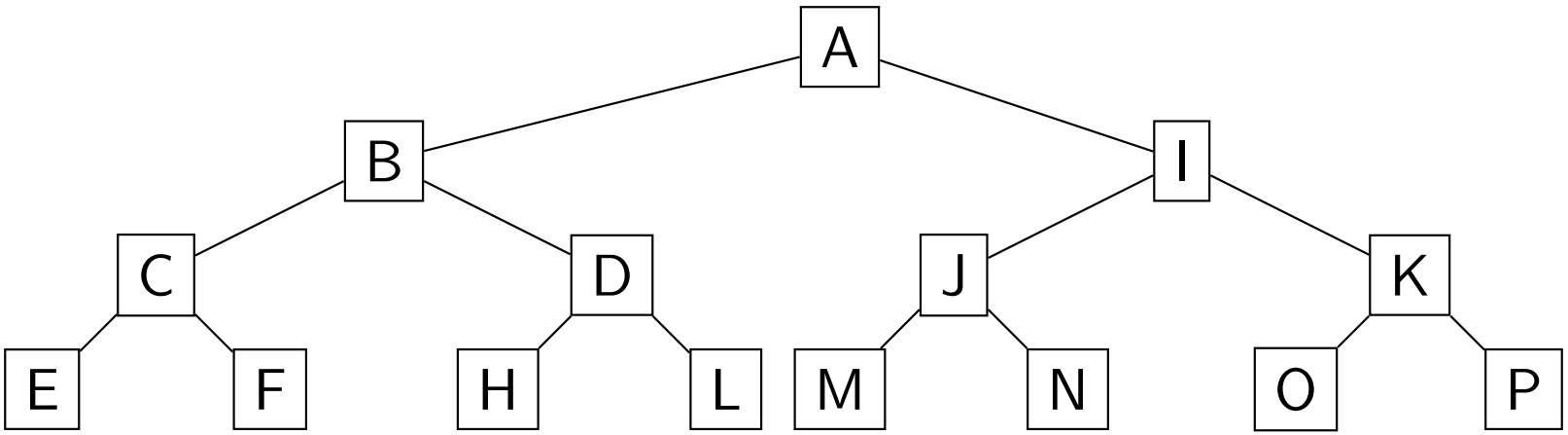

FIGURE 5 – Un arbre binaire parfait

Sommaire

Propriétés

À retenir

Dans un arbre binaire, la taille N et la hauteur h sont liées par les inégalités :

$$h + 1 \leqslant N \leqslant 2^{h+1} - 1$$

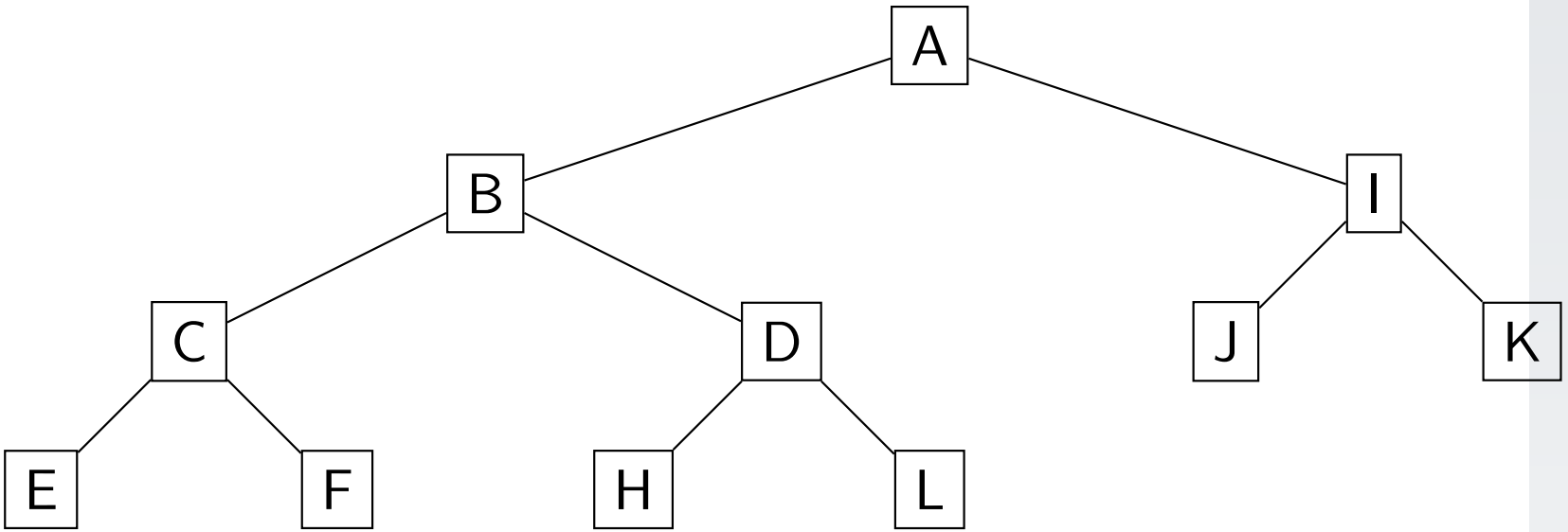

FIGURE 6 – Un arbre binaire complet

Démonstration

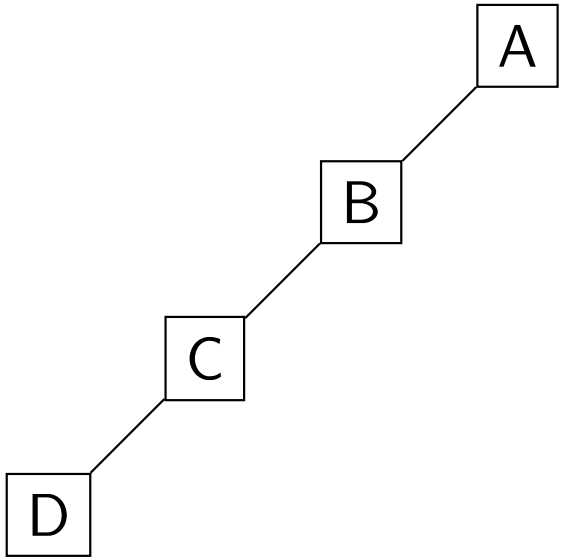

FIGURE 7 – Un arbre binaire minimal

$$h + 1 \leqslant N$$

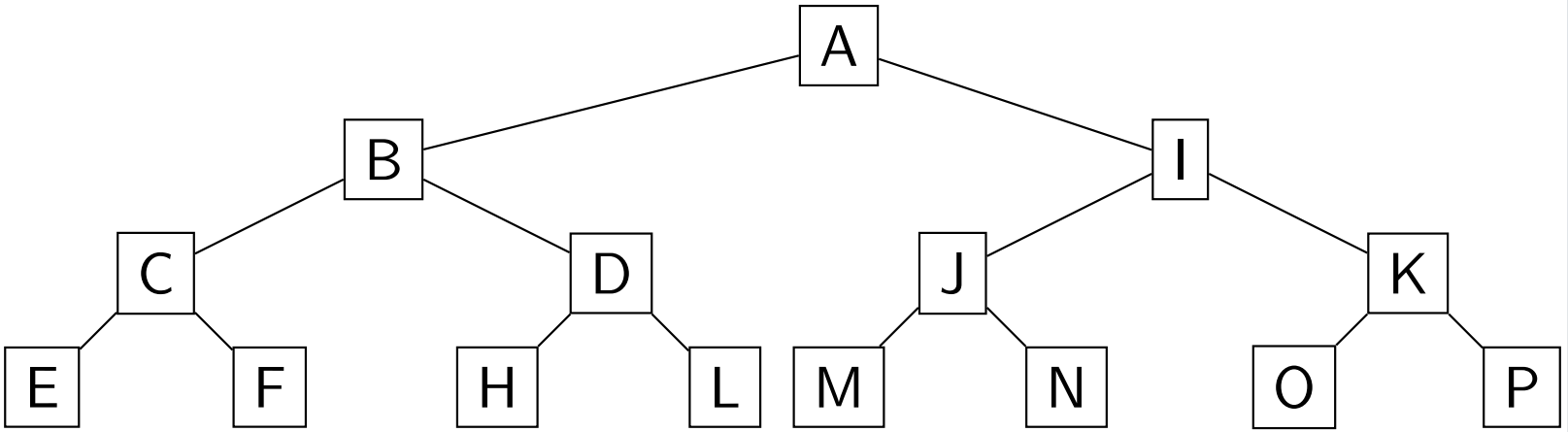

FIGURE 8 – Un arbre binaire parfait

- niveau 0 : $2^0 = 1$ nœuds
- niveau 1 : $2^1 = 2$ nœuds
- ...
- niveau h : 2^h nœuds

On double le nombre de nœuds à chaque niveau.

Somme des premiers termes d'une suite géométrique :

$$\sum_{k=0}^{h} 2^k = u_0 \times \frac{1 - q^{h+1}}{1 - q}$$

$$\sum_{k=0}^{h} 2^k = 1 \times \frac{1 - 2^{h+1}}{1 - 2}$$

$$\sum_{k=0}^{h} 2^k = 2^{h+1} - 1$$

La taille maximale est inférieure ou égale à $2^{h+1} - 1$

À retenir

L'inégalité est vérifiée :

$$h + 1 \leqslant N \leqslant 2^{h+1} - 1$$

Remarque

Si on définit la hauteur comme le nombre de nœuds maximum entre la racine et une feuille, on a :

$$h \leqslant N \leqslant 2^{h} - 1$$

Sommaire

Représentation d'une expression mathématique

Une expression mathématique applique une *opération* sur deux *opérandes*. Un arbre binaire permet donc de représenter n'importe quelle opération.

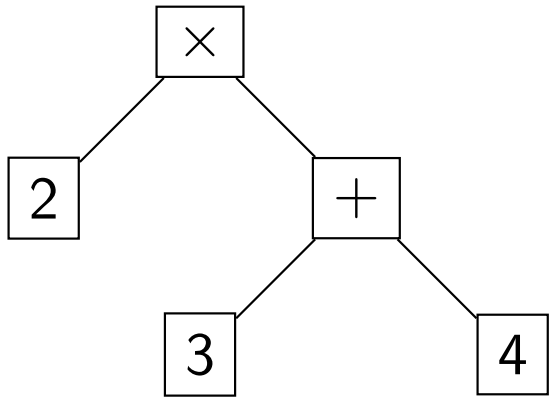

FIGURE 9 – $2 \times (3 + 4)$

Sommaire

Représentation en Python

```
1  class Noeud:
2
3  def __init__(self, v, g, d):
4      self.valeur = v
5      self.gauche = g
6      self.droite = d
```

Code 1 – Représentation d'un nœud

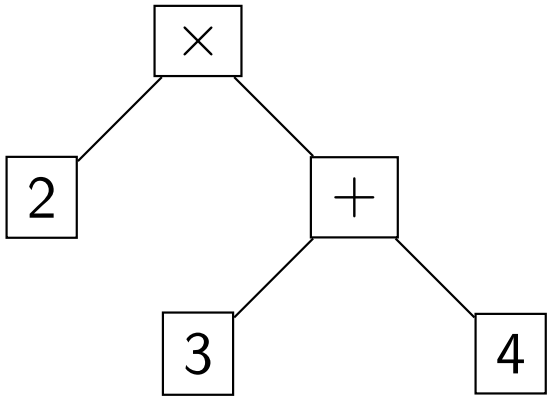

FIGURE 10 – $2 \times (3 + 4)$

Activité 1 : En utilisant la classe `Noeud` écrire la variable `arbre` qui représente l'expression 10.

Correction

```
1  arbre = Noeud("×",
2                Noeud(2, None, None),
3                Noeud("+",
4                      Noeud(3, None, None),
5                      Noeud(4, None, None)))
```

Pour calculer la taille d'un arbre, il faut :

- calculer récursivement la taille du fils gauche,
- calculer récursivement la taille du fils droit,
- ajouter 1 (pour le nœud en cours).

Activité 2 :

1. Dans l'algorithme de calcul de la taille, quel est le cas limite ?
2. Écrire la fonction récursive `taille(a: Noeud)` → `int` qui renvoie la taille de l'arbre.

Correction

```python
def taille(a: Noeud) -> int:
    """
    renvoie le nombre de noeuds de l'arbre a
    """
    if a is None:
        return 0
    else:
        return 1 + taille(a.gauche) + taille(a.droite)
```

Code 2 – Taille de l'arbre

Activité 3 :

1. Écrire la fonction récursive **`hauteur(a: Noeud)`** $\rightarrow$ **`int`** qui renvoie la hauteur de l'arbre binaire. On utilisera la fonction Python **`max`** pour comparer la taille des fils gauche et droit.
2. Étudier la complexité des fonctions **`taille`** et **`hauteur`**.

Correction


```
1  def hauteur(a: Noeud) -> int:
2      """
3      hauteur max de l'arbre a
4      """
5      if a is None:
6          return 0
7      else:
8          return 1 + max(hauteur(a.gauche), hauteur(a.droite))
```

Code 3 – Hauteur de l'arbre

Dans les deux fonctions on parcourt une et une seule fois chaque nœud. La complexité est **linéaire**.

Sommaire

Parcours en profondeur

Dans un arbre binaire, on commence par parcourir le sous-arbre gauche avant celui de droite.

Parcours préfixe :

```
1 parcours préfixe(arbre)
2     affiche(valeur)
3     parcours préfixe(sous-arbre gauche)
4     parcours préfixe(sous-arbre droit)
```

Parcours préfixe :

```
1 parcours préfixe(arbre)
2     affiche(valeur)
3     parcours préfixe(sous-arbre gauche)
4     parcours préfixe(sous-arbre droit)
```

Parcours infixe :

```
1 parcours infixe(arbre)
2     parcours infixe(sous-arbre gauche)
3     affiche(valeur)
4     parcours infixe(sous-arbre droit)
```

Parcours préfixe :

```
1 parcours préfixe(arbre)
2     affiche(valeur)
3     parcours préfixe(sous-arbre gauche)
4     parcours préfixe(sous-arbre droit)
```

Parcours infixe :

```
1 parcours infixe(arbre)
2     parcours infixe(sous-arbre gauche)
3     affiche(valeur)
4     parcours infixe(sous-arbre droit)
```

Parcours postfixe (ou suffixe) :

```
1 parcours postfixe(arbre)
2     parcours postfixe(sous-arbre gauche)
3     parcours postfixe(sous-arbre droit)
4     affiche(valeur)
```

Activité 4 :

1. Écrire les trois fonctions *récursives* de parcours qui affichent (`print`) directement la valeur du nœud traversé.
2. Adapter ces fonctions pour renvoyer un tableau ordonné des nœuds traversés.
3. Quel parcours implémente la notation polonaise inverse ?
4. Étudier la complexité des fonctions de parcours.

Correction

```
1  def prefixe(a: Noeud) -> None:
2      if a is not None:
3          print(a.valeur, end=" ")
4          prefixe(a.gauche)
5          prefixe(a.droite)
```

```
1  prefixe(arbre)
```

```
1  '×' 2 '+' 3 4
```

Correction

```
1  def infixe(a: Noeud) -> None:
2      if a is not None:
3          infixe(a.gauche)
4          print(a.valeur, end=" ")
5          infixe(a.droite)
6
7
8  def postfixe(a: Noeud) -> None:
9      if a is not None:
10         postfixe(a.gauche)
11         postfixe(a.droite)
12         print(a.valeur, end=" ")
```

```python
def prefixe_tab(a: Noeud, parcours: list) -> None:
    if a is not None:
        parcours.append(a.valeur)
        prefixe_tab(a.gauche, parcours)
        prefixe_tab(a.droite, parcours)
```

Remarque

Le tableau passé en paramètre est une **référence** à un tableau du programme principal

```python
tab_prefixe = []
prefixe_tab(arbre, tab_prefixe)
print("préfixe ", tab_prefixe)
```

Code 4 – Appel

```
1  def infixe_tab(a: Noeud, parcours: list) -> None:
2      if a is not None:
3          infixe_tab(a.gauche, parcours)
4          parcours.append(a.valeur)
5          infixe_tab(a.droite, parcours)
6
7
8
9  def postfixe_tab(a: Noeud, parcours: list) -> None:
10     if a is not None:
11         postfixe_tab(a.gauche, parcours)
12         postfixe_tab(a.droite, parcours)
13         parcours.append(a.valeur)
```

```
1 # notation polonaise
2 préfixe  ['×', 2, '+', 3, 4]
3
4 # notation usuelle
5 infixe  [2, '×', 3, '+', 4]
6
7 # notation polonaise inverse
8 postfixe  [2, 3, 4, '+', '×']
```

Remarque

Dans la notation usuelle les parenthèses sont obligatoires pour éviter les ambiguïtés.

```python
def prefixe_tab2(a: Noeud) -> list:
    if a is not None:
        # concatène les tableaux construits
        return [a.valeur] + \
               prefixe_tab2(a.gauche) + \
               prefixe_tab2(a.droite)
    else:
        return []
```

Code 5 – Version avec concaténation

Remarque

En fin de parcours, il faut renvoyer un tableau vide pour concaténer des structures de même type.

À retenir

Un parcours en profondeur (ou en largeur) visite une et une seule fois chaque nœud. La complexité en temps est **linéaire**.