

# ARBRES GÉNÉRIQUES ET PRÉFIXES

---

À la fin de ce chapitre, je sais :

- ☞ convertir un arbre générique en arbre binaire
- ☞ expliquer ce qu'est un arbre préfixe
- ☞ expliquer le principe du codage d'Huffman
- ☞ implémenter un trie dans le cas où l'arbre est binaire

## A Arbres génériques

■ **Définition 1 — Arbre générique** . Un arbre générique est un arbre qui possède un nombre d'enfants quelconque.

En OCaml on peut définir un arbre générique comme suit :

```
type 'a gtree =  
  | Empty  
  | Node of 'a * 'a gtree list;;
```

Les fils de l'arbre sont contenus dans une **liste de sous-arbres**.

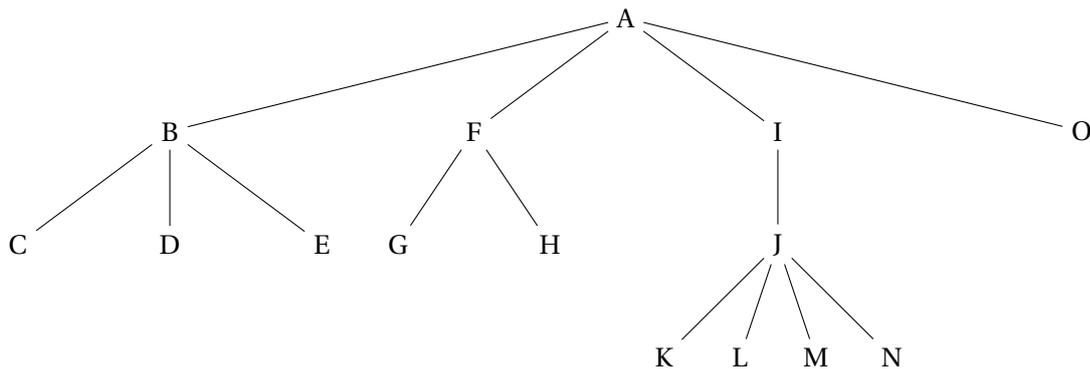


FIGURE 1 – Arbres générique

**R** Un arbre générique peut se parcourir par niveau grâce à un parcours en largeur. Il peut également se parcourir en profondeur dans les ordres préfixes, infixes et postfixes.

Sur l'arbre générique de la figure 1, un parcours par niveau résulte en ['A'; 'B'; 'F'; 'I'; 'O'; 'C'; 'D'; 'E'; 'G'; 'H'; 'J'; 'K'; 'L'; 'M'; 'N'].

De même, le parcours en profondeur préfixe résulte en ['A'; 'B'; 'C'; 'D'; 'E'; 'F'; 'G'; 'H'; 'I'; 'J'; 'K'; 'L'; 'M'; 'N'; 'O'].

### a Calculs sur un arbre générique

Pour calculer la hauteur d'un arbre générique, on doit évaluer chaque sous-arbre, c'est-à-dire traiter tous les sous-arbres présents dans la liste. Il est nécessaire de procéder par récursivité mutuelle ou bien en utilisant les fonctions `List.fold_left` et `max` comme le propose le code ci-dessous.

```
let rec h a =
  match a with
  | Empty -> -1
  | Node (_, []) -> 0
  | Node (_, fils) -> 1 + h_fils fils
  and h_fils fils =
    match fils with
    | [] -> 0
    | f::t -> max (h f) (h_fils t);;

(* Version folding *)
let rec hauteur a =
  match a with
  | Empty -> -1
  | Node (_, []) -> 0
  | Node (_, fils) -> 1 + List.fold_left (fun acc f -> max acc (hauteur f)) 0 fils
  ;;
```

### b Transformer un arbre générique en arbre binaire

Il est possible de transformer un arbre générique en arbre binaire en adoptant la convention suivante : le fils gauche est un fils du nœud, le fils droit est un frère de ce fils. Avec cette convention, l'arbre générique de la figure 1 est transformé en l'arbre binaire de la figure 2

## B Arbres préfixes

■ **Définition 2 — Arbre préfixe** . Soit  $\mathcal{E}$  un ensemble d'étiquettes et  $\mathcal{S}$  un ensemble de séquences sur ces étiquettes. L'arbre préfixe de  $\mathcal{S}$  est un arbre enraciné tel que :

1. chaque **arête** est étiquetée par une étiquette de  $\mathcal{E}$ ,
2. pour un nœud  $n$  et deux de ses fils  $g$  et  $d$ , l'arête  $(n, g)$  ne porte pas la même étiquette que l'arête  $(n, d)$ ,

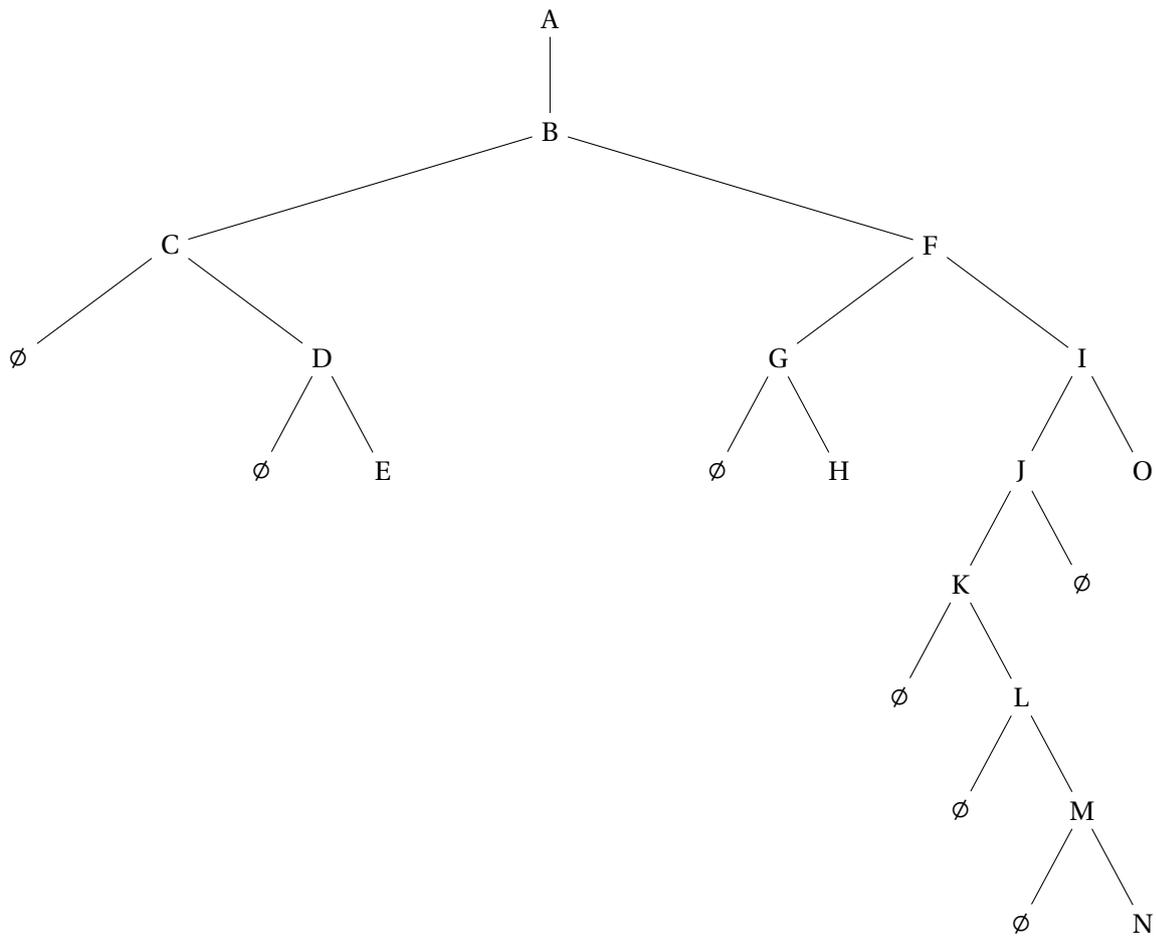


FIGURE 2 – Transformation de l'arbre générique de la figure 1 en arbre binaire

3. à chaque séquence  $s$  de l'ensemble  $S$  correspond une feuille  $f$  de l'arbre telle que la concaténation des étiquettes des arêtes sur le chemin de la racine à  $f$  est égale à  $s$ .
4. de même, chaque chemin de la racine à une feuille correspond à une séquence.

**R** Un arbre préfixe garantit qu'une séquence ne peut pas être le préfixe d'une autre. L'avantage en termes de codage est considérable : le code ainsi généré est uniquement déchiffrable, il ne peut pas y avoir d'ambiguïté.

La figure 3 représente un arbre préfixe de chaînes de caractères : les chaînes sont les séquences de l'arbre.

La figure 4 représente un arbre d'Huffman binaire. L'ensemble des étiquettes est  $\{0,1\}$ , les nœuds sont étiquetés par les occurrences et les feuilles par le symbole correspondant à la sé-



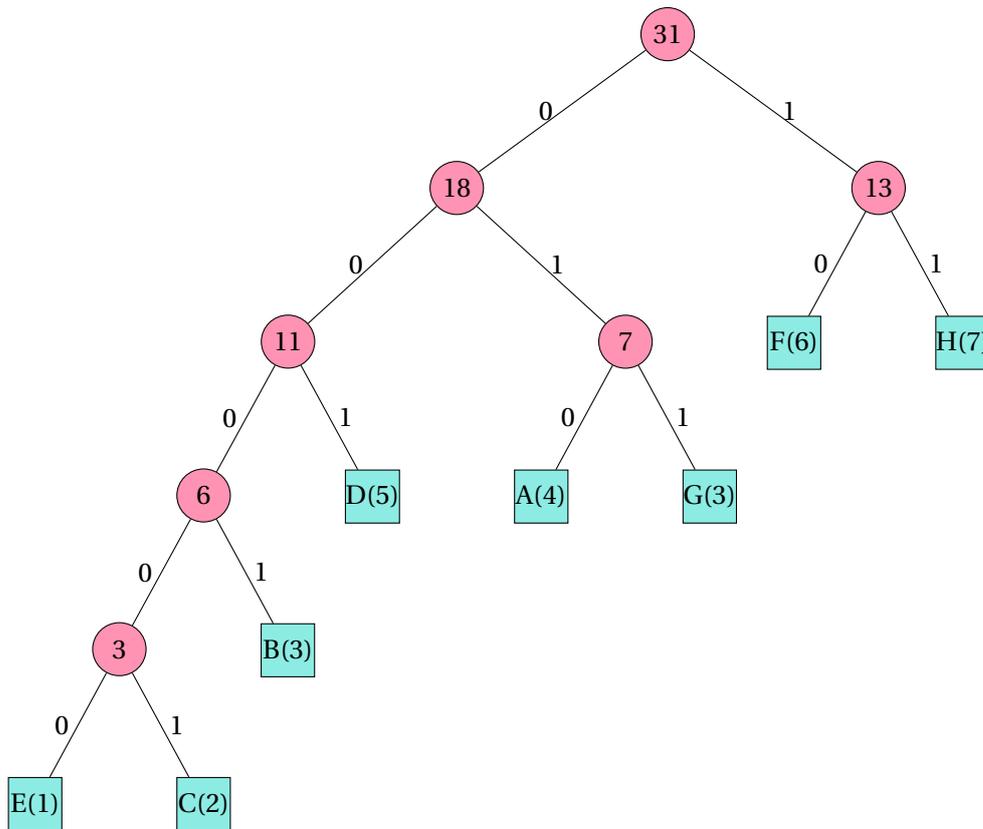


FIGURE 4 – Arbre d’Huffmann permettant de coder des caractères en binaire

**Algorithme 1** Construction d’un arbre préfixe associé à un code de Huffmann

---

```

1: Fonction HUFFMANN( $S, O$ )                                     ▷  $S$  est un ensemble de symboles
2:    $F \leftarrow$  file de priorités                               ▷  $O$  les occurrences de chaque de symboles
3:   pour chaque couple  $(s, o)$  de  $S \times O$  répéter
4:     ENFILER( $F, (FEUILLE(s), o)$ )
5:   tant que LONGUEUR( $F$ ) > 1 répéter
6:      $g, o_g \leftarrow$  DÉFILER( $F$ )
7:      $d, o_d \leftarrow$  DÉFILER( $F$ )
8:      $o \leftarrow o_g + o_d$                                        ▷ Addition des occurrences des sous-arbres
9:      $s \leftarrow$  SYMBOLE( $g$ ) + SYMBOLE( $d$ )                         ▷ Concaténation des symboles
10:    parent  $\leftarrow$  NŒUD( $(s, o), g, d$ )                             ▷ Création du nouveau nœud
11:    ENFILER( $F, (parent, o)$ )
12:  renvoyer DÉFILER( $F$ )

```

---

---

**Algorithme 2** Compresser à l'aide d'un arbre de Huffman
 

---

```

1: Fonction H_COMPRESS( $s, r$ )                                ▷  $s$  un symbole source,  $r$  la racine l'arbre
2:   si  $s$  est présent dans SYMBOLE(GAUCHE( $r$ )) alors
3:     renvoyer CONCAT(0, H_COMPRESS( $s, GAUCHE(r)$ ))
4:   si  $s$  est présent dans SYMBOLE(DROIT( $r$ )) alors
5:     renvoyer CONCAT(1, H_COMPRESS( $s, DROITE(r)$ ))
6:   renvoyer  $\epsilon$                                           ▷ On retourne le mot vide  $\epsilon$ 

```

---



---

**Algorithme 3** Décompresser à l'aide d'un arbre de Huffman
 

---

```

1: Fonction H_DECOMPRESS( $m, r$ )                                ▷  $m$  un mot codé,  $r$  la racine l'arbre
2:    $n \leftarrow r$ 
3:    $d \leftarrow \epsilon$                                           ▷ Le mot vide
4:   pour chaque bit  $b$  de  $m$  répéter
5:     si  $n$  est une feuille alors
6:        $d \leftarrow$  CONCAT( $d, SYMBOLE(n)$ )
7:        $n \leftarrow r$                                           ▷ On a trouvé un symbole, on en cherche un autre
8:     sinon si  $b = 0$  alors
9:        $n \leftarrow GAUCHE(n)$ 
10:    sinon si  $b = 1$  alors
11:       $n \leftarrow DROIT(n)$ 
12:    sinon
13:      renvoyer  $\epsilon$                                           ▷ Ce symbole binaire n'existe pas, erreur
14:  renvoyer  $d$ 

```

---

**semi-adaptatif** si les probabilités associées aux éléments à compresser sont calculées et stockées (ou transmises) avec les données compressées. Les rendements sont meilleurs mais il faut stocker ou transmettre les probabilités.

**adaptatif** l'arbre est construit dynamiquement au fur et à mesure de la réception des symboles à coder. La complexité de l'algorithme de codage est plus grande mais le stockage ou la transmission de l'arbre n'est plus nécessaire. Les algorithmes FGK [**knuth\_dynamic\_1985**] et V [**vitter\_design\_1987**] permettent de réaliser cet arbre de Huffman dynamique.

Les codes de Huffman sont optimaux mais ils présentent certains inconvénients :

- la version non adaptative de ces codes ne peut pas être synchrone à la transmission des données,
- ils sont très sensibles aux erreurs de transmission. Si on fait par exemple l'erreur de décoder un mot par une autre de longueur différente suite à une erreur de transmission alors cette erreur se propage naturellement à la suite du décodage : on interprètera mal les symboles suivants.

## D Compression à dictionnaire

Issus des travaux initiaux de Jacob Ziv, Abraham Lempel et Terry Welch [**ziv\_universal\_1977**, **ziv\_compression\_1978**, **welch\_technique\_1984**], ces codes ne se fondent pas sur les statistiques de la source mais consistent à **associer à des séquences de symboles une entrée dans un dictionnaire**. Le code est constitué par les clefs du dictionnaire. La position des clefs est la valeur associée à la clef dans le dictionnaire. L'opération de décompression consiste à sélectionner les valeurs associées aux clefs du dictionnaire. On n'a donc pas besoin de connaître la source a priori pour utiliser ces algorithmes. De plus, on peut montrer qu'ils sont asymptotiquement optimaux.