Arbres binaires et Arbres binaires de recherche

Sommaire

Sommaire

Introduction

- On a vu jusqu'ici des structures de données *linéaires* (listes, piles, files) :
  - adaptées pour des objets pouvant être énumérés séquentiellement (nombres, chaînes de caractères, objets d'une classe supportant une opération de comparaison) ;
  - mais inefficaces pour des algorithmes d'accès à des éléments en position arbitraire.
- Certaines données s'organisent naturellement sous forme arborescente :
  - arborescence d'un système de fichiers [1] ;
  - le DOM (Document Object Model) en JavaScript d'une page html ;
  - le contenu d'un arbre généalogique ;
  - la structure hiérarchique des salariés d'une entreprise ;
  - arborescence liée à la preuve d'une formule de logique.
- Les *arbres binaires* sont une forme élémentaire d'arborescence, souvent utilisée pour l'efficacité de stockage/recherche de données qu'elle permet.

1. p.ex. sous unix/linux, windows, MacOS, iOS, Android, ...

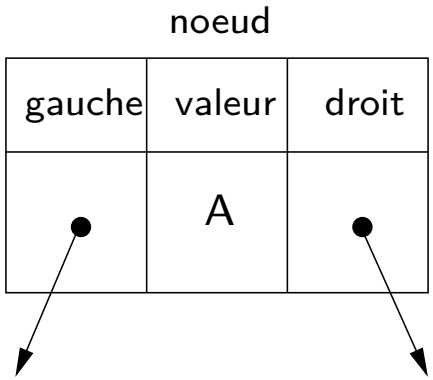
Ex. 1 Sous Debian, ouvrir un terminal. À l'aide de la commande `which tree`, vérifier que la commande `tree` est installée sur votre système. La commande `tree` permet d'afficher, sous forme d'un arbre, l'ensemble des répertoires et fichiers situés à un certain endroit du système de fichiers. Pour connaître le fonctionnement de cette commande et ce qu'elle fait, taper la commande `man tree` et lire (au moins le début de) la documentation affichée (taper `q` pour sortir).

1. Le répertoire `/etc` est celui qui contient les éléments de configuration importants du système. Vérifier, en tapant la commande `tree /etc`, que s'affiche, sous forme d'un arbre, le contenu de ce répertoire. Combien de sous-répertoires contient-il ? de fichiers ?
2. Par quelle commande peut-on afficher l'arbre des sous-répertoires du répertoire racine, en affichant seulement les sous-répertoires (pas les fichiers), et seulement ceux qui sont directement sous le niveau racine (on rappelle que le niveau racine est noté / sous UNIX) ? Repérer dans l'arbre affiché les répertoires importants.

Ex. 2 Une formule de logique peut être validée par la construction d'un arbre. Le site https://www.umsu.de/trees/ propose une application en ligne réalisant cette construction/vérification. On sait, par exemple, que l'implication mathématique $p \implies q$, notée $p \rightarrow q$ en logique, est équivalente à l'expression $\neg p \vee q$ (rappel : $\neg$ est l'opérateur de négation, $\vee$ est le ou logique). Traduit en notations de la logique, la formule $(p \rightarrow q) \leftrightarrow (\neg p \vee q)$ est donc vraie. En tapant cette formule dans la zone de saisie de la page web précédemment citée, faites construire l'arbre de validation de cette formule. Vérifier également que la formule $(p \rightarrow q) \leftrightarrow (\neg p \wedge q)$ est fausse.

Sommaire

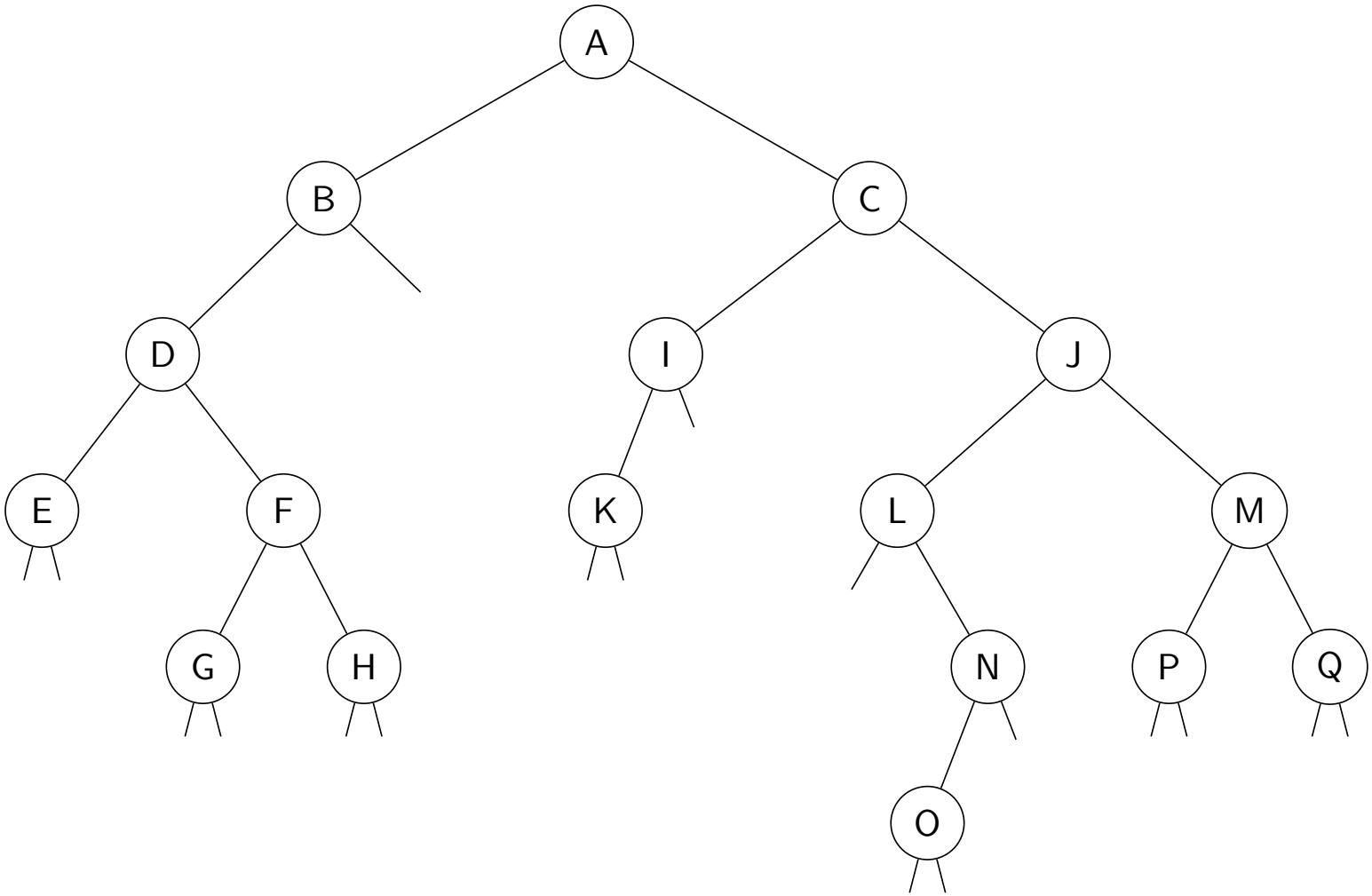
- Un *arbre binaire* est composé de *nœuds*. Chaque nœud est susceptible de contenir une donnée et d'être relié à deux nœuds qui seront appelés le *nœud gauche* et le *nœud droit*.



- Un nœud particulier de l'arbre, appelé *nœud racine*, est celui par lequel on accède à l'arbre. On identifiera un arbre à son nœud racine.
- Tout nœud de l'arbre (et tout arbre, sauf s'il est vide) est relié à deux arbres binaires, appelés *sous-arbre gauche* et *sous-arbre droit* ayant respectivement pour racine le nœud gauche et le nœud droit du nœud considéré.
- Il existe un arbre ne contenant aucun nœud, appelé *arbre vide*, qu'on notera parfois nil[2].
- Les nœuds d'un arbre binaire qui ont nil comme sous-arbre gauche et droit sont appelés *feuilles* de l'arbre. Ils sont aux extrémités des «branches» de l'arbre.

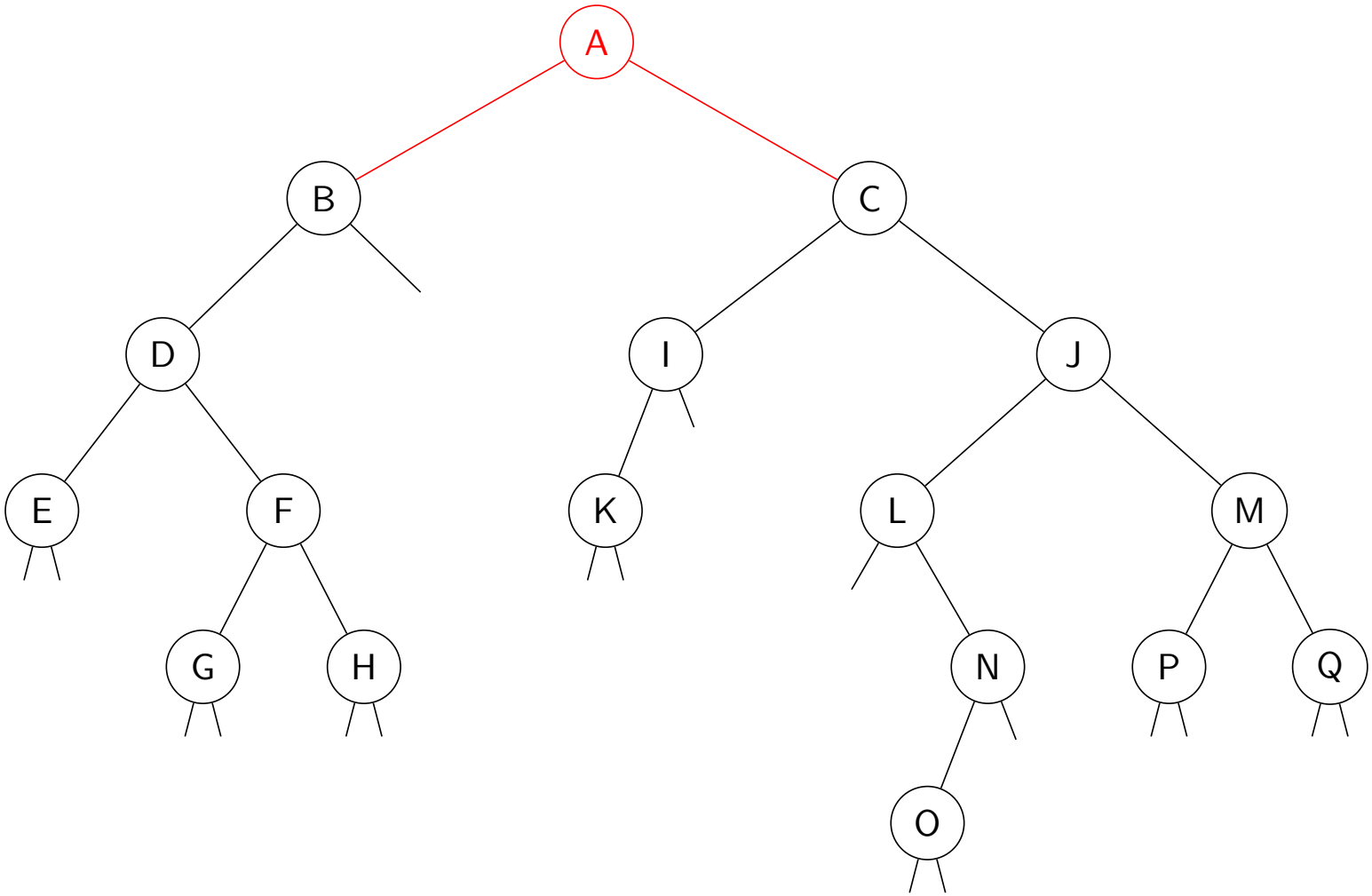
2. Abréviation du latin nihil, qui signifie rien.

Un arbre binaire contenant des lettres.



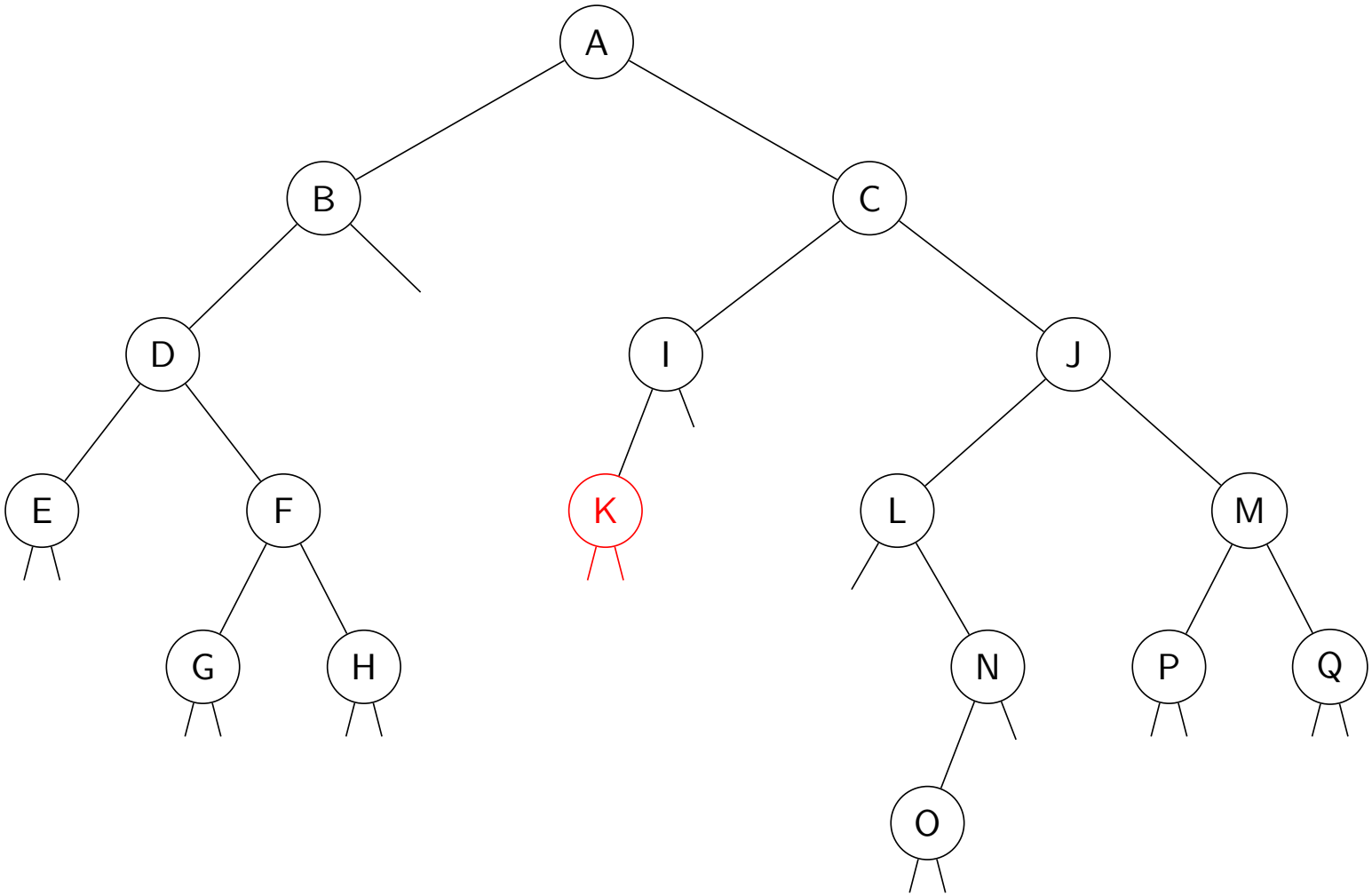
Un arbre

Un arbre binaire contenant des lettres.



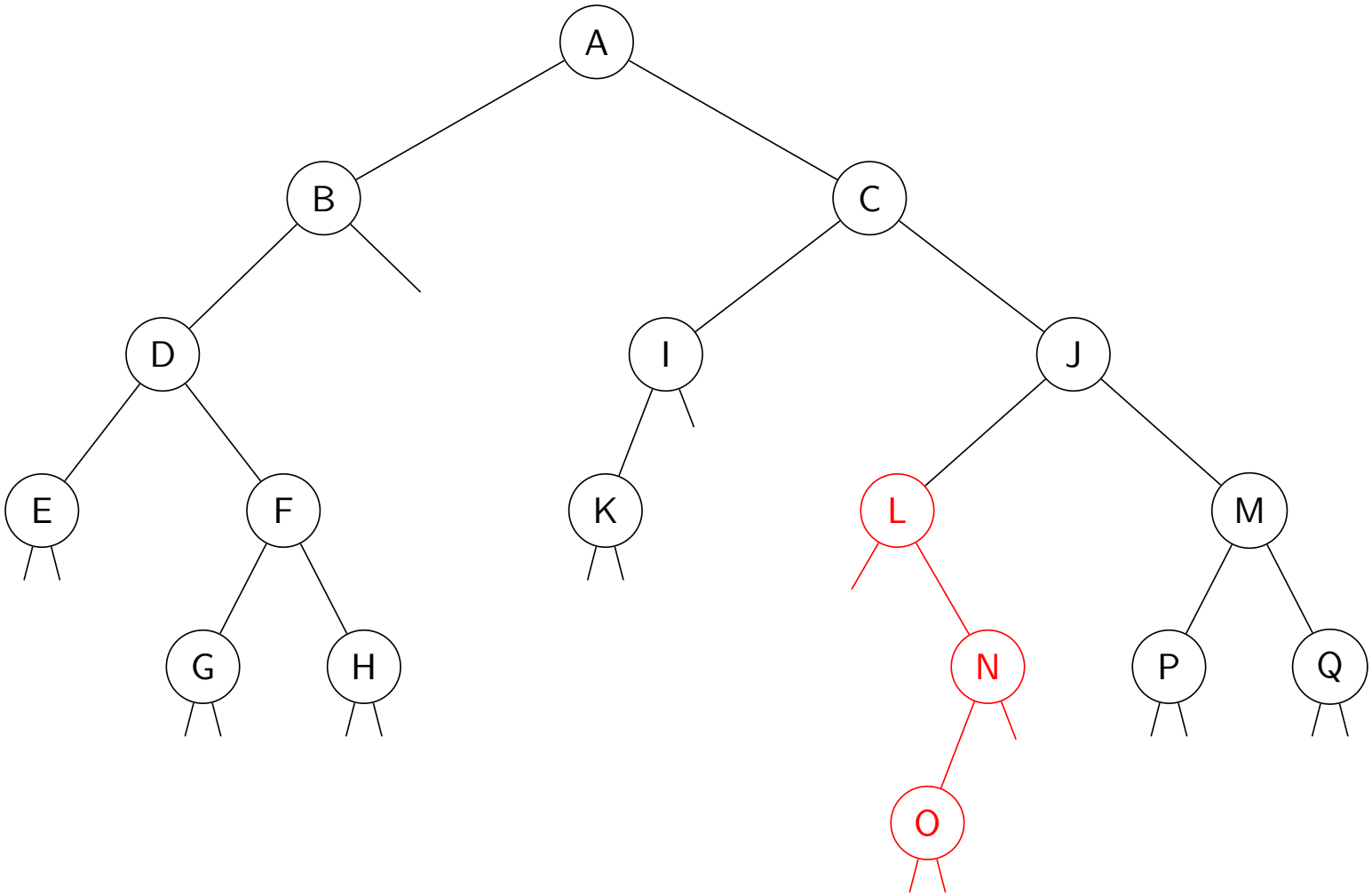
Son nœud racine

Un arbre binaire contenant des lettres.



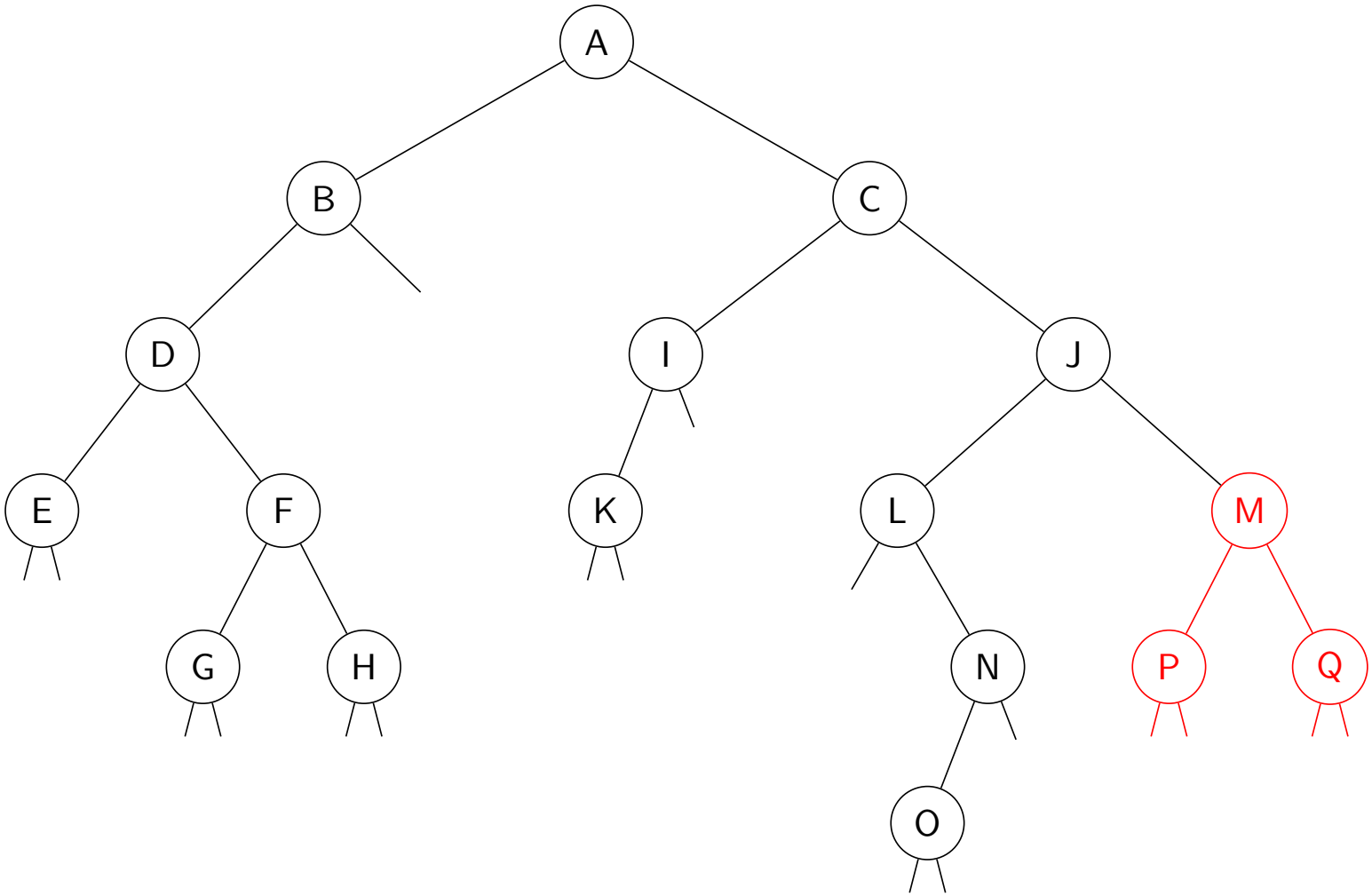
Un nœud feuille

Un arbre binaire contenant des lettres.



Le sous-arbre gauche du nœud J

Un arbre binaire contenant des lettres.



Le sous-arbre droit du nœud J

- On utilisera une classe `Noeud` pour construire des nœuds et une classe ... `Arbre` pour construire des arbres :

```
class Noeud:
    """un noeud d'un arbre binaire"""
    def __init__(self, g=None, v=None, d=None):
        self.gauche = g # (nœud racine du) sous-arbre gauche
        self.valeur = v # clé du noeud
        self.droit = d  # (nœud racine du) sous-arbre droit

class Arbre:
    """un arbre binaire"""
    def __init__(self, noeud = None):
        self.racine = noeud
```

Remarques :

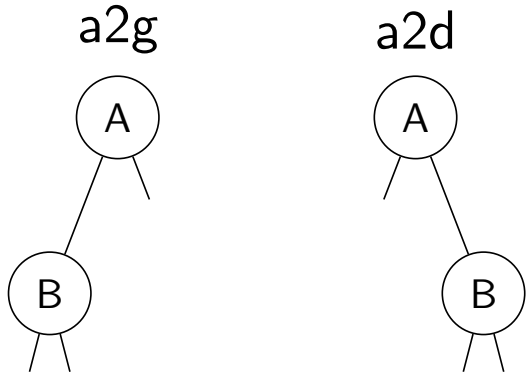
1. On étudiera ici des arbres binaires *homogènes*, c'est à dire dont les valeurs contenues dans les nœuds sont toutes du même type (`str`, `int`,...)
2. Noter le caractère intrinsèquement récursif de la définition d'un arbre binaire. Ceci se traduira dans l'écriture (elle aussi souvent naturellement récursive) des fonctions travaillant sur les arbres binaires.

3. Il serait possible de construire des arbres binaires autrement. Par exemple avec des listes ou des tuples, comme on le montre ci-dessous.

```
a2g = [[None, "B", None], "A", None]
a2d = (None, "A", (None, "B", None))
```

ou

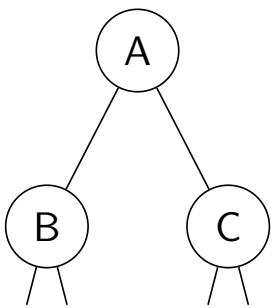
```
a2g = [[[], "B", []], "A", []]
a2d = ((), "A", ((), "B", ()))
```



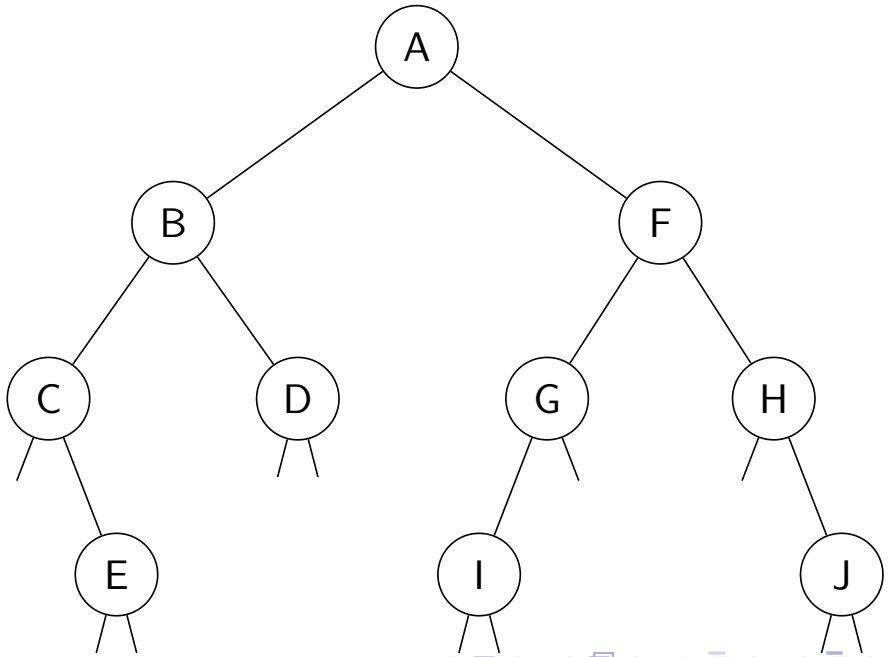
Ex. 3 Soit l'arbre `a10` (à 10 nœuds) dessiné page suivante. Quelle est la valeur de son nœud racine ? Donner un de ses nœuds feuilles. Donner le sous-arbre gauche du nœud contenant `B`, le sous-arbre droit du nœud contenant `A`.

Ex. 4 À l'aide des classes `Noeud` et `Arbre` données précédemment, créer des variables de la classe `Arbre`, notées `av`, `a1`, `a2g`, `a2d`, `a3` et `a10` représentant respectivement : l'arbre binaire vide, l'arbre binaire ayant un nœud, les arbres binaires `a2g`, `a2d` déjà donnés ci-dessus, et les arbres binaires `a3` et `a10` représentés ci-dessous. Pour vérifier que les arbres construits sont les bons, on pourra utiliser la fonction `ecrit_fichier_dot(racine)` (fournie dans le code source à télécharger sur le site du cours de NSI), qui construit un fichier au format `.dot`, visualisable par exemple à l'aide de `xdot` ou de l'éditeur en ligne (graphviz visual editor) : ici.

a3



a10



N.B. : on doit bien remarquer que *tout* nœud est relié à deux sous-arbres (éventuellement vides) et qu'une feuille est reliée à deux *arbres* vides.

⌨ Ex. 5 Dessiner tous les arbres binaires ayant respectivement 3 et 4 nœuds.

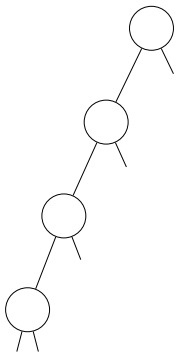
⌨ Ex. 6 Sachant qu'il existe un arbre binaire vide, un arbre binaire contenant un nœud, 2 arbres binaires contenant 2 nœuds, 5 arbres binaires contenant 3 nœuds et 14 arbres binaires contenant 4 nœuds, calculer le nombre d'arbres binaires contenant 5 nœuds. Rq : ce nombre s'appelle le nombre de Catalan d'ordre 5.

Sommaire

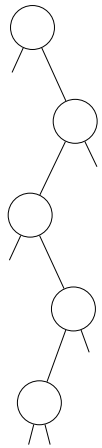
Hauteur et taille d'un arbre binaire

- Un arbre binaire est (entre autre) caractérisé de façon importante par sa *hauteur* et sa *taille*.
- La forme d'un arbre binaire est variable. Voici quelques arbres particuliers :

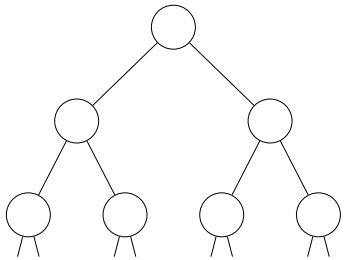
arbre peigne



arbre filiforme



arbre parfait



- La *hauteur* d'un arbre binaire est, par définition, le nombre maximal de nœuds par lesquels on passe en partant de la racine pour atteindre une feuille (en comptant la racine et la feuille). Par exemple, la hauteur de a3 est 2, celle de a10 est 4 (vérifiez-le).
- La *taille* d'un arbre est simplement, par définition, son nombre de nœuds.

- On comprend qu'il existe un lien entre la hauteur et la taille d'un arbre binaire. Par ailleurs, à taille fixée, la recherche d'une valeur dans d'un arbre binaire de petite hauteur sera plus efficace que dans un arbre de grande hauteur.
- Si la taille est N, la plus grande hauteur h est obtenue pour un arbre filiforme et on a alors $h = N$. Donc, dans le cas général, $h \leqslant N$.
- Si la hauteur est h, le nombre maximal de nœuds correspond à un arbre parfait de hauteur h. Or la taille N d'un tel arbre est

$$N = 1 + 2^1 + 2^2 + \cdots + 2^{h-1} = \frac{2^h - 1}{2 - 1} = 2^h - 1. \quad \text{Donc } N \leqslant 2^h - 1.$$

- On a donc obtenu, pour tout arbre binaire de taille N et de hauteur h :

$$\begin{array}{rcl} h & \leqslant & N \leqslant 2^h - 1 \\ \log_2(h) \leqslant \log_2(N) & \leqslant & \log_2(2^h - 1) \leqslant \log_2(2^h) = h \end{array}$$

Donc :

$$\boxed{\log_2(N) \leqslant h \leqslant N}$$

Ci-dessus, $\log_2$ est le logarithme en base deux, souvent utile en informatique et en électronique, qui vérifie pour tout $x > 0$, $\log_2(2^x) = x$. L'encadrement ci-dessus sera utile pour évaluer la meilleure complexité possible pour un algorithme de recherche (cf. les ABR plus loin).

Un algorithme, écrit en pseudo-code, du calcul récursif de la hauteur d'un arbre binaire dont le nœud racine est racine est :

```
Fonction HAUTEUR(racine)
    si racine = nil alors                                  # arbre vide
        renvoyer 0
    sinon
        renvoyer 1 + max(HAUTEUR(racine.gauche),HAUTEUR(racine.droite))
```

Ex. 7 Écrire en Python, en utilisant l'algorithme précédent, une fonction `hauteur(racine)` qui calcule et renvoie la hauteur d'un arbre binaire dont le nœud racine est `racine`. Tester votre fonction sur les arbres précédemment créés (cf. ex. 4). Rq : la fonction `max` est prédéfinie en Python.

Un algorithme, écrit en pseudo-code, de calcul récursif de la taille d'un arbre binaire est le suivant :

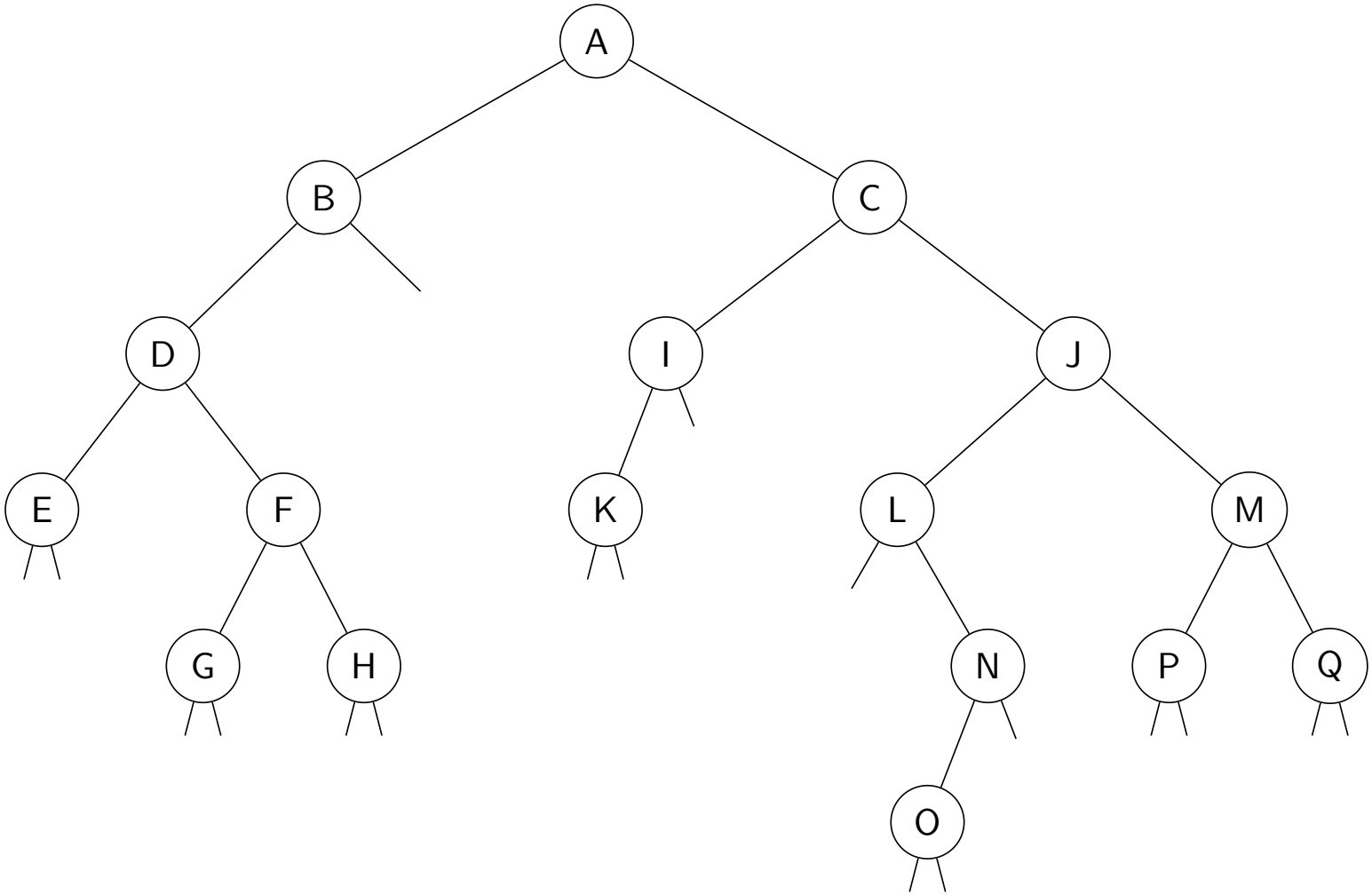
```
Fonction TAILLE(racine)
    si racine = nil alors                                  # arbre vide
        renvoyer 0
    sinon
        renvoyer 1 + TAILLE(racine.gauche) + TAILLE(racine.droite)
```

Ex. 8 Écrire en Python, en utilisant l'algorithme précédent, une fonction `taille(racine)` qui calcule et renvoie la taille d'un arbre binaire dont le nœud racine est `racine`. Tester votre fonction sur les arbres précédemment créés (cf. ex. 4).

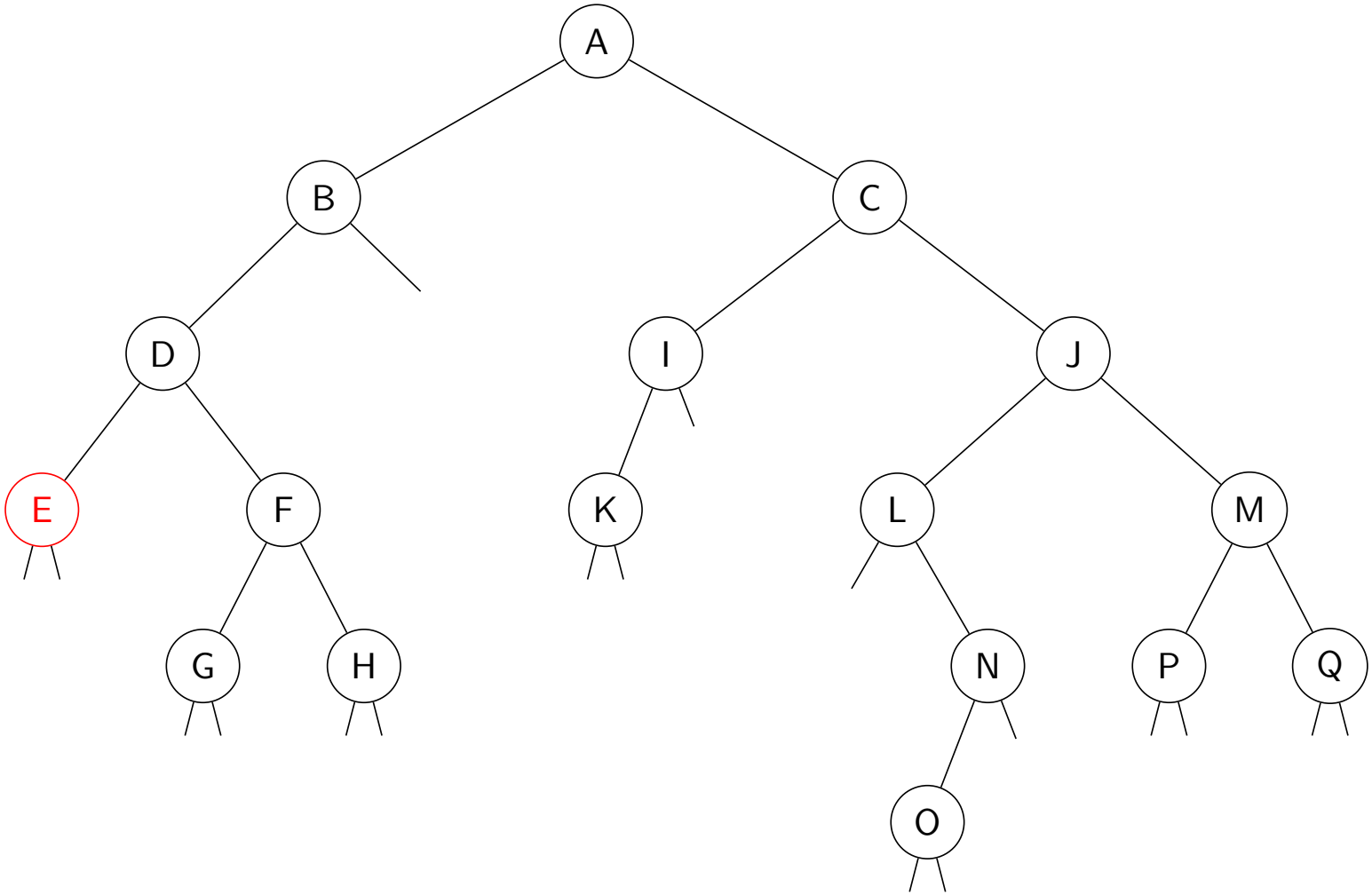
Sommaire

Parcours d'un arbre binaire

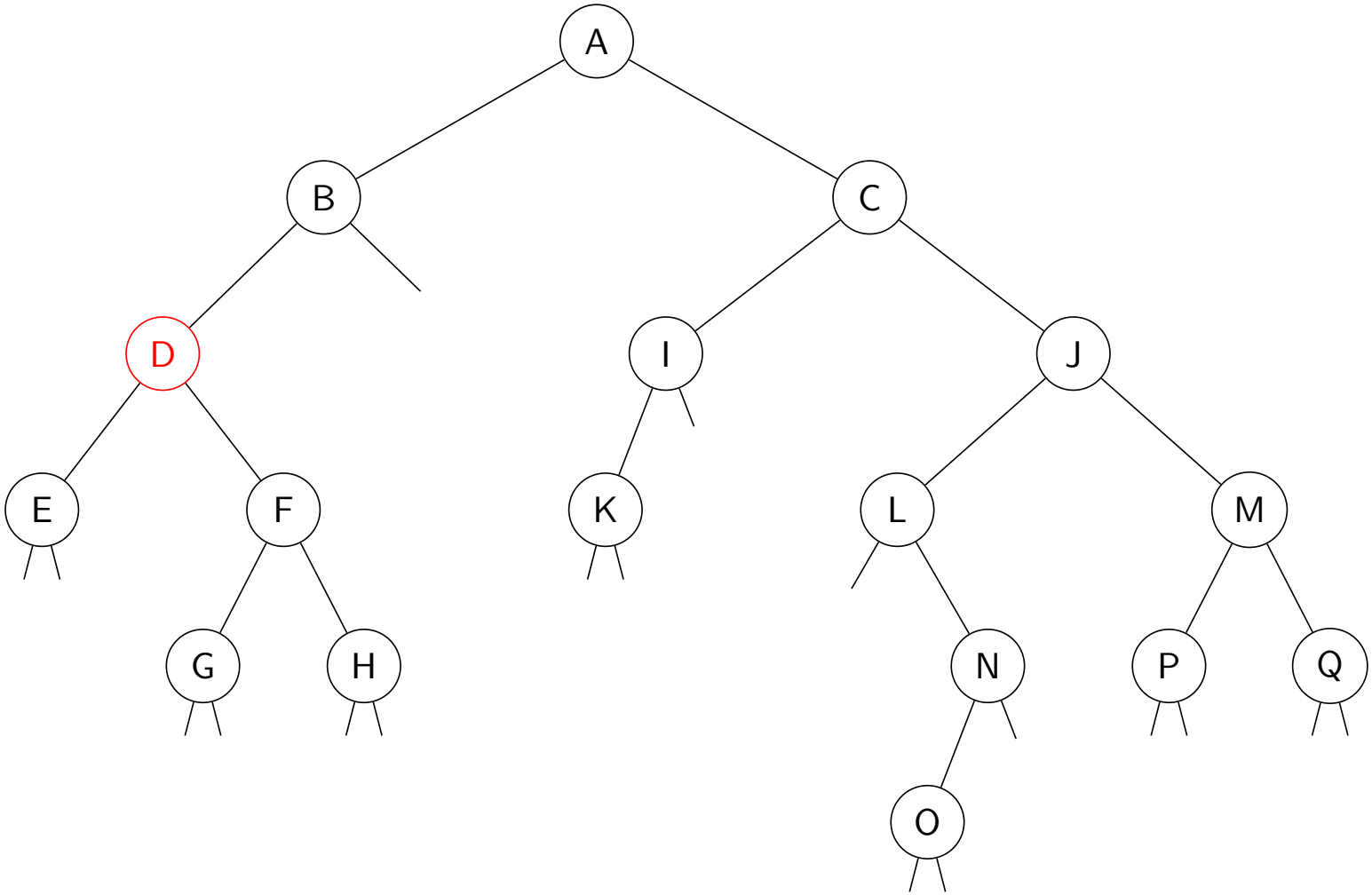
- On peut avoir besoin (par exemple pour afficher son contenu) d'énumérer les nœuds d'un arbre binaire. On distingue plusieurs types d'algorithmes qui réalisent ce qu'on appelle un *parcours* d'arbre :
  - parcours *en profondeur* :
    - infixe
    - préfixe
    - suffixe
  - parcours *en largeur*
- Les parcours en profondeur se distinguent par l'*ordre* dans lequel on énumère les nœuds.
- Pour un parcours *infixe* (d'un arbre non vide), on réalise récursivement :
  - le parcours infixe du sous-arbre gauche ;
  - l'affichage du contenu de sa racine ;
  - le parcours infixe du sous-arbre droit.
- Voici l'exemple du parcours infixe d'un arbre donné précédemment.



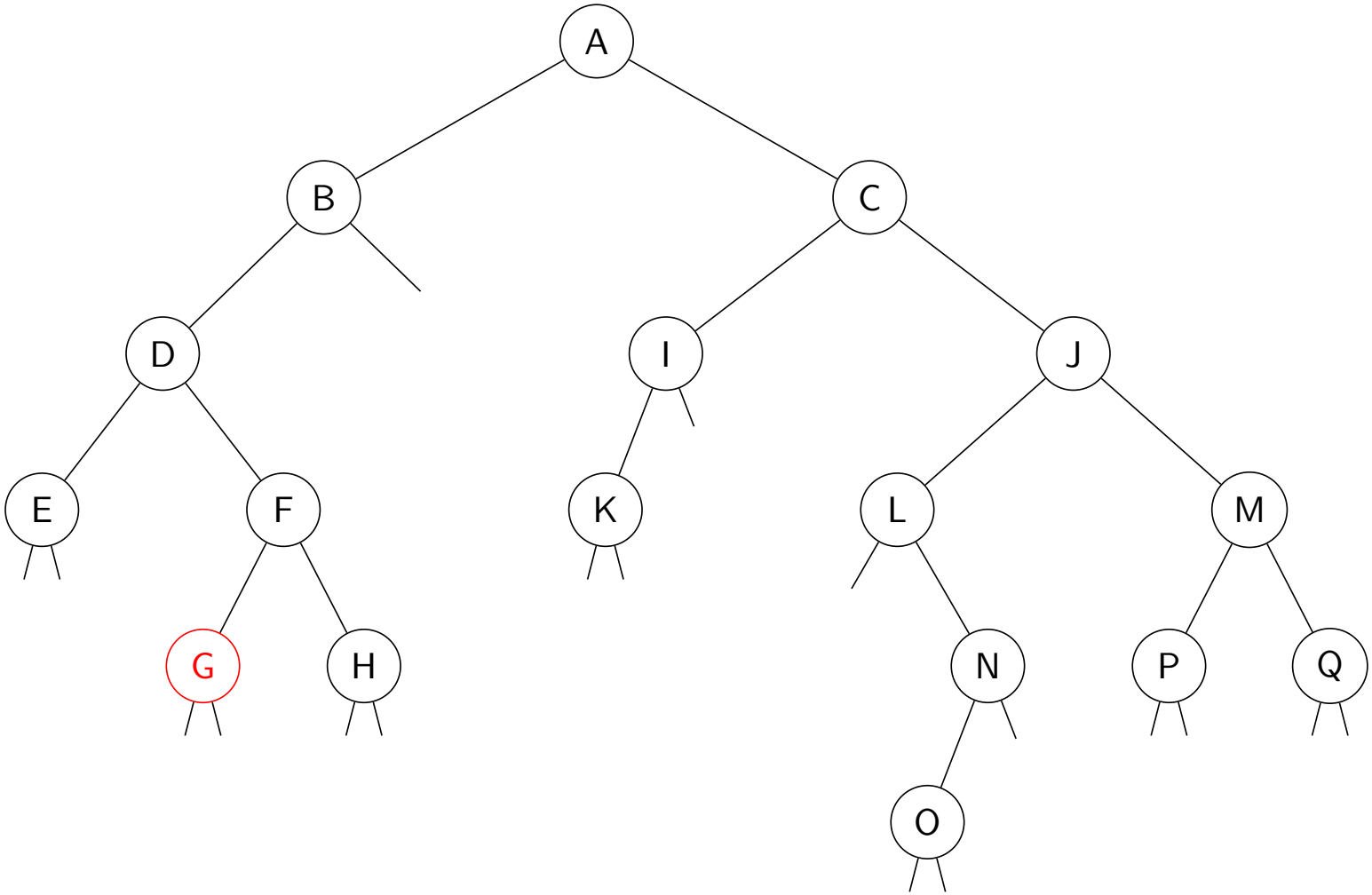
Parcours infixe :



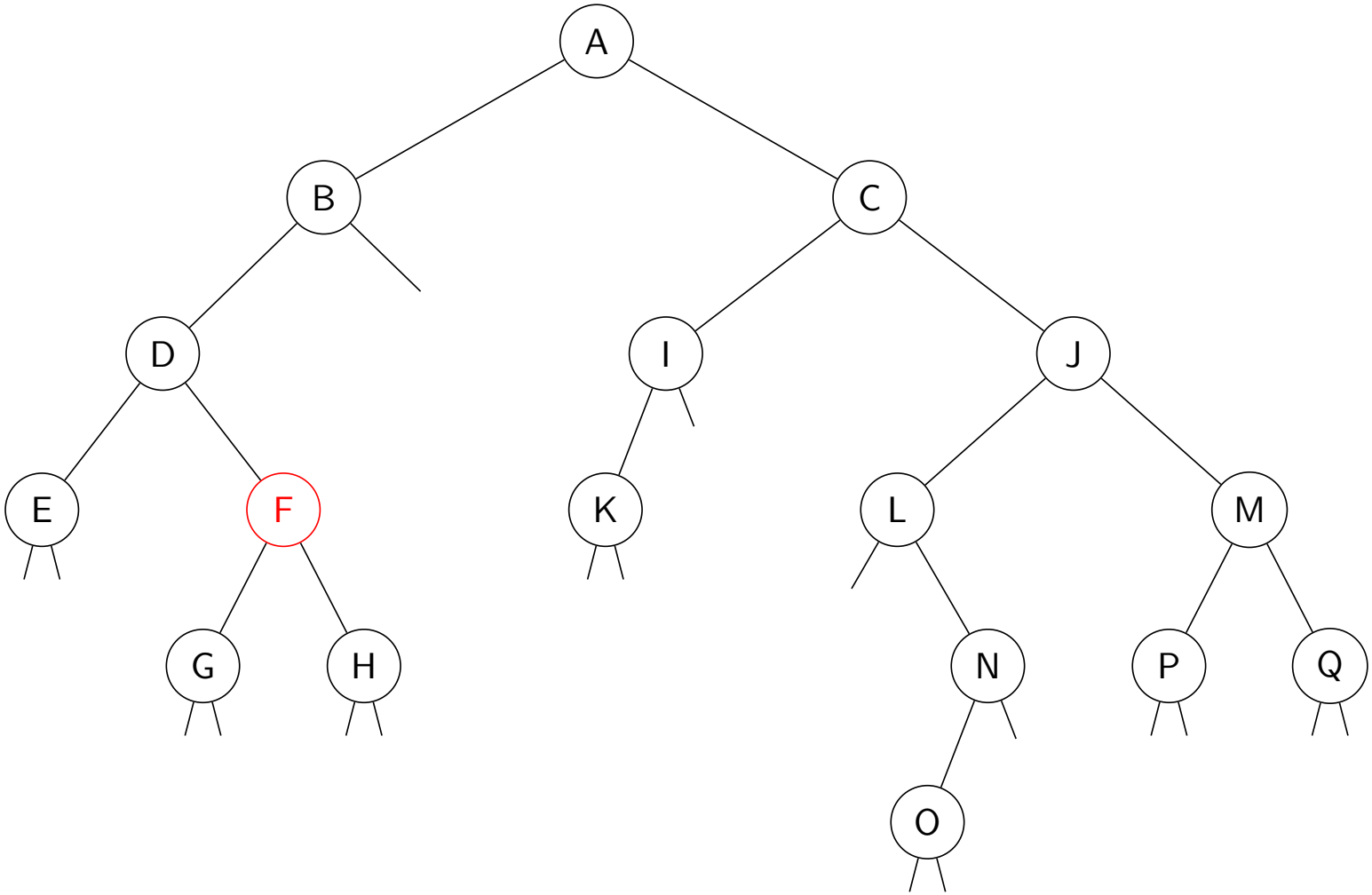
Parcours infixe : E



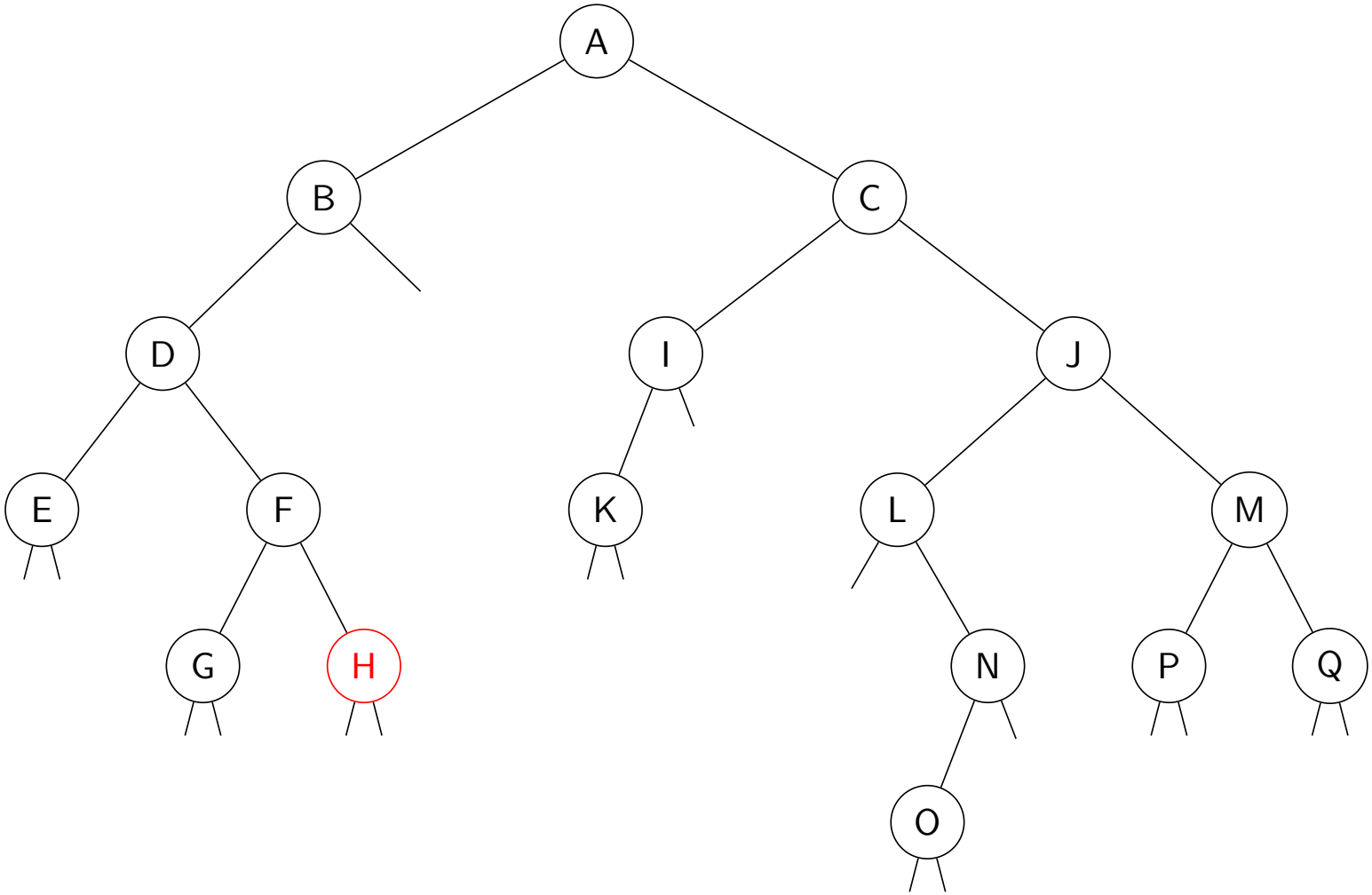
Parcours infixe : E D



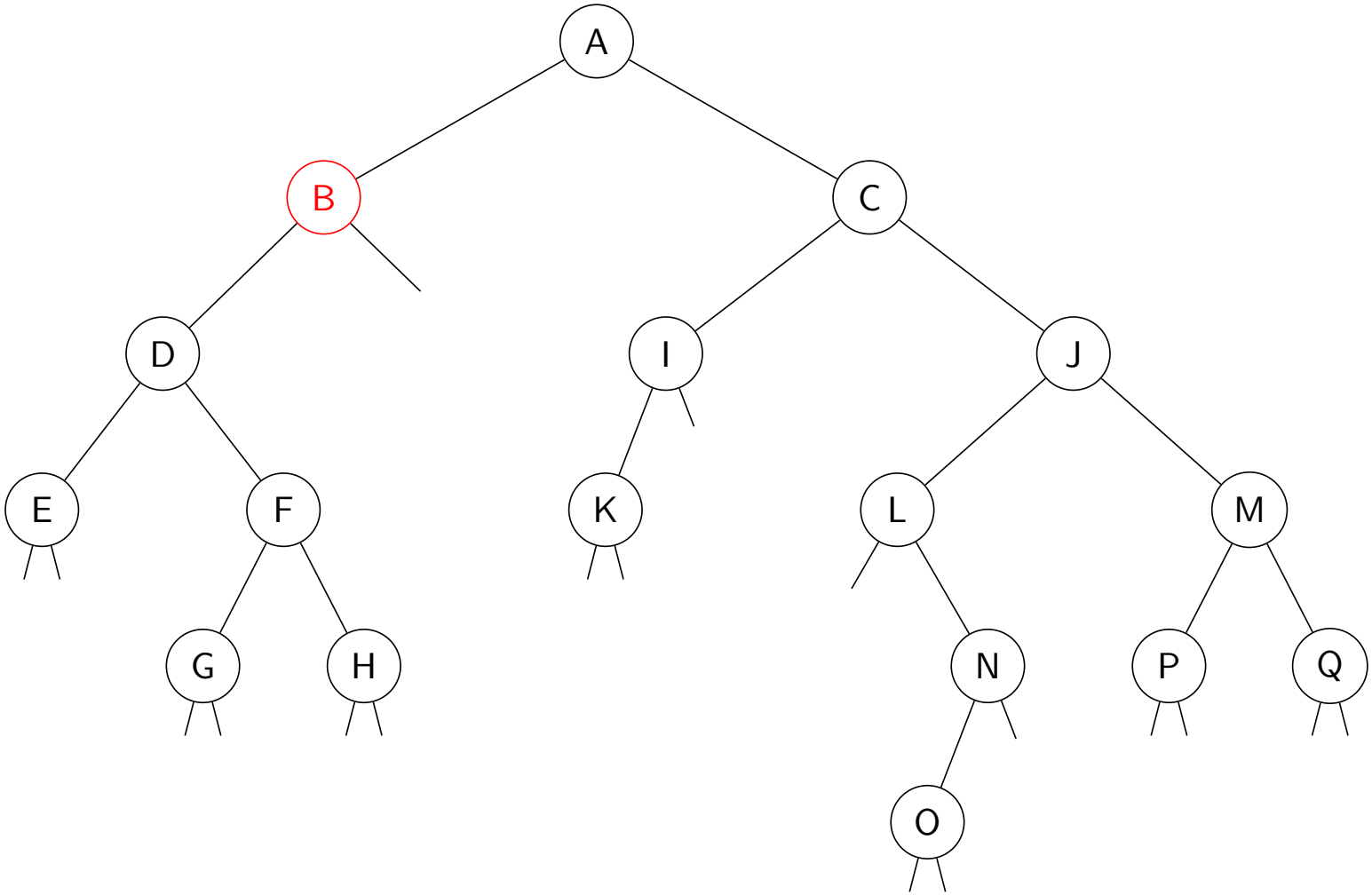
Parcours infixe : E D G



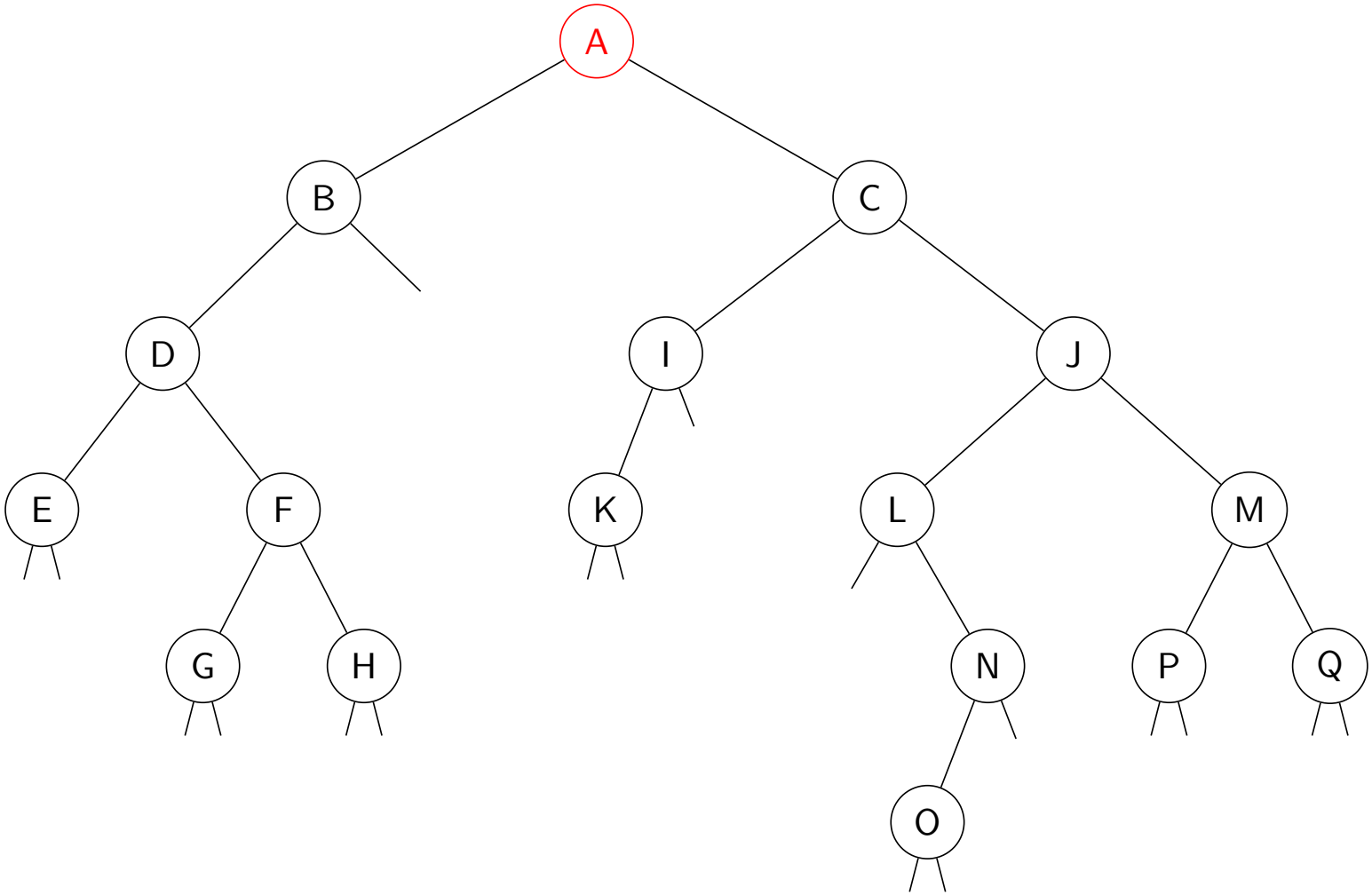
Parcours infixe : E D G F



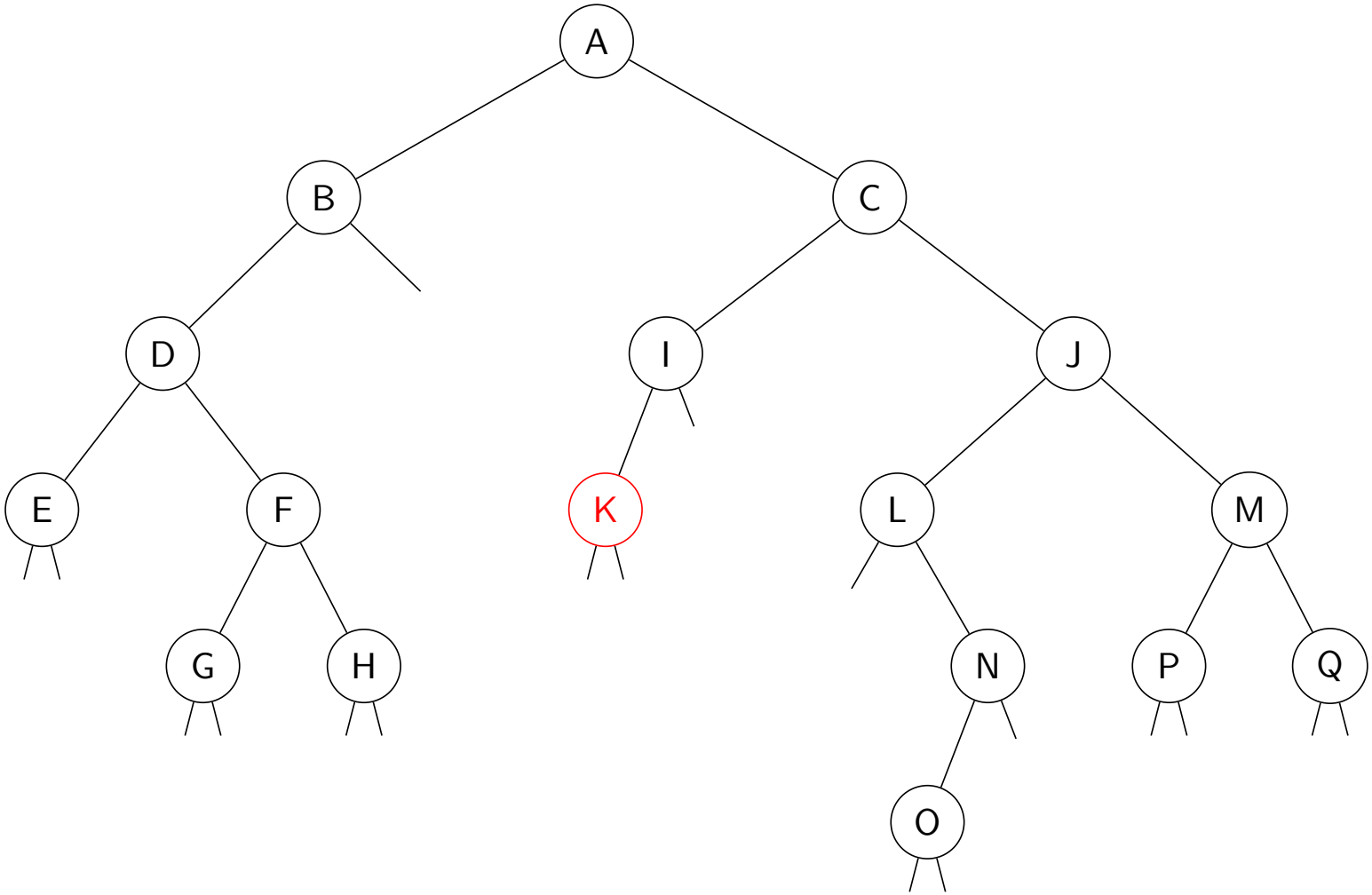
Parcours infixe : E D G F H



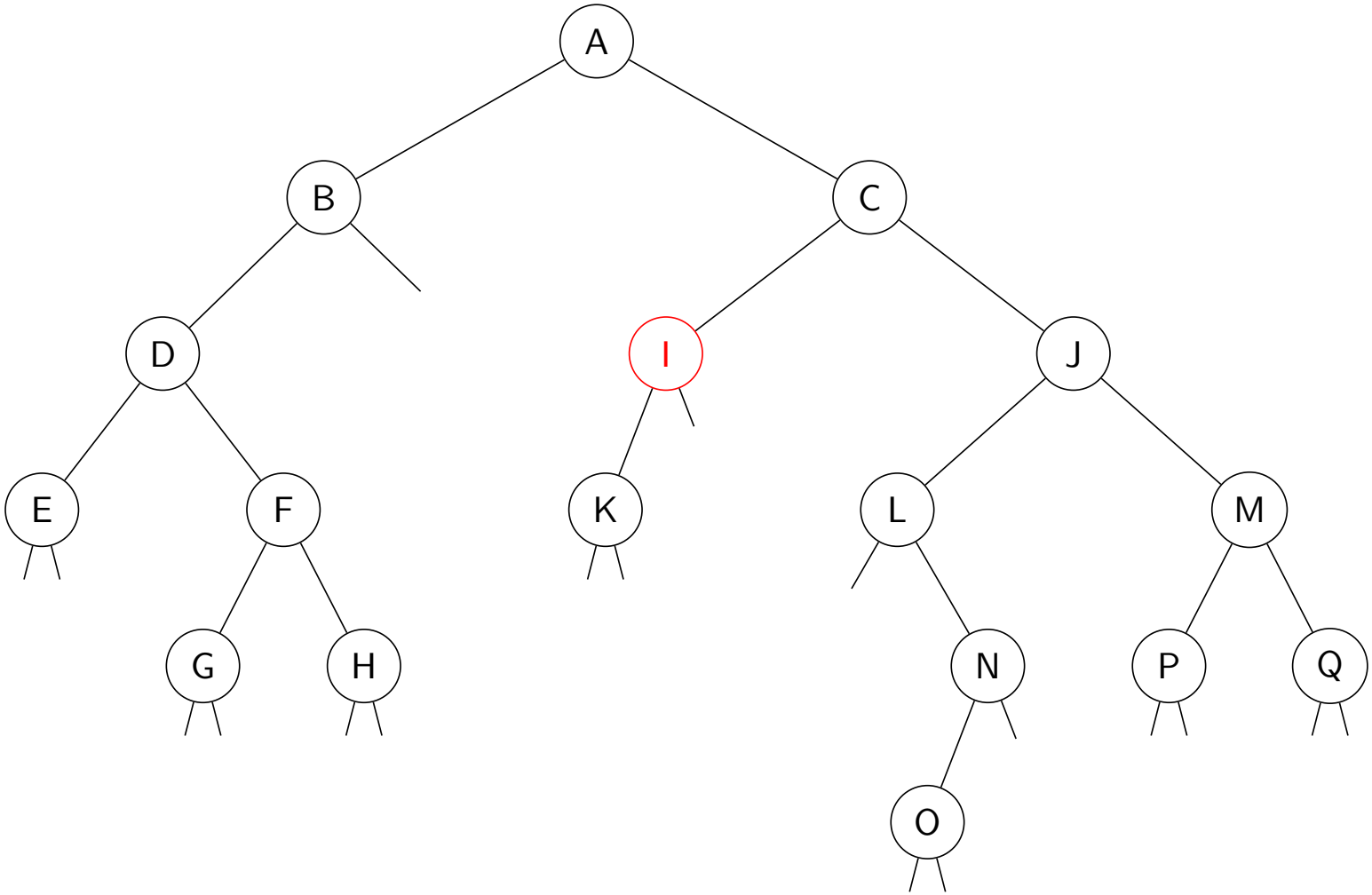
Parcours infixe : E D G F H B



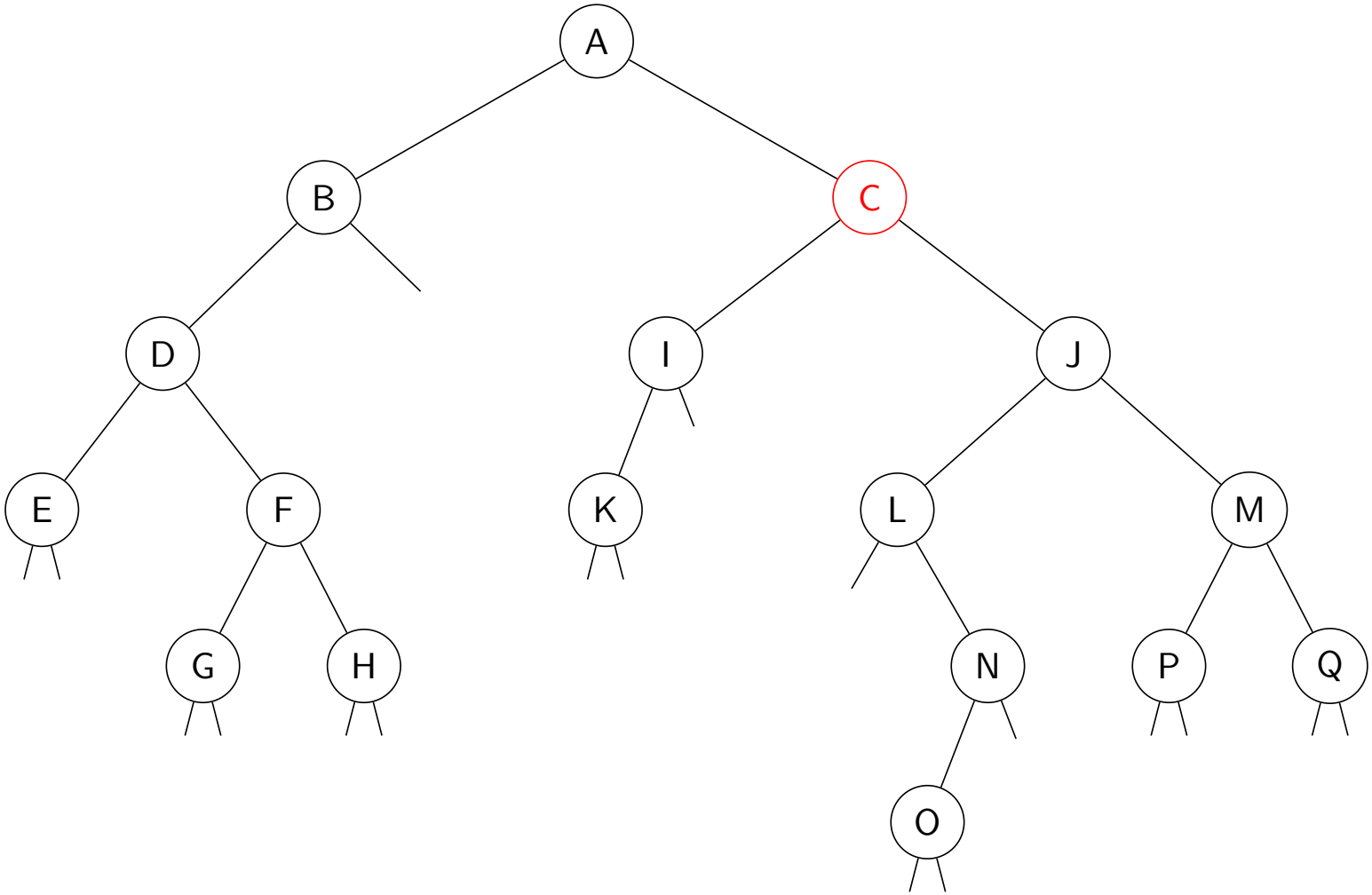
Parcours infixe : E D G F H B A



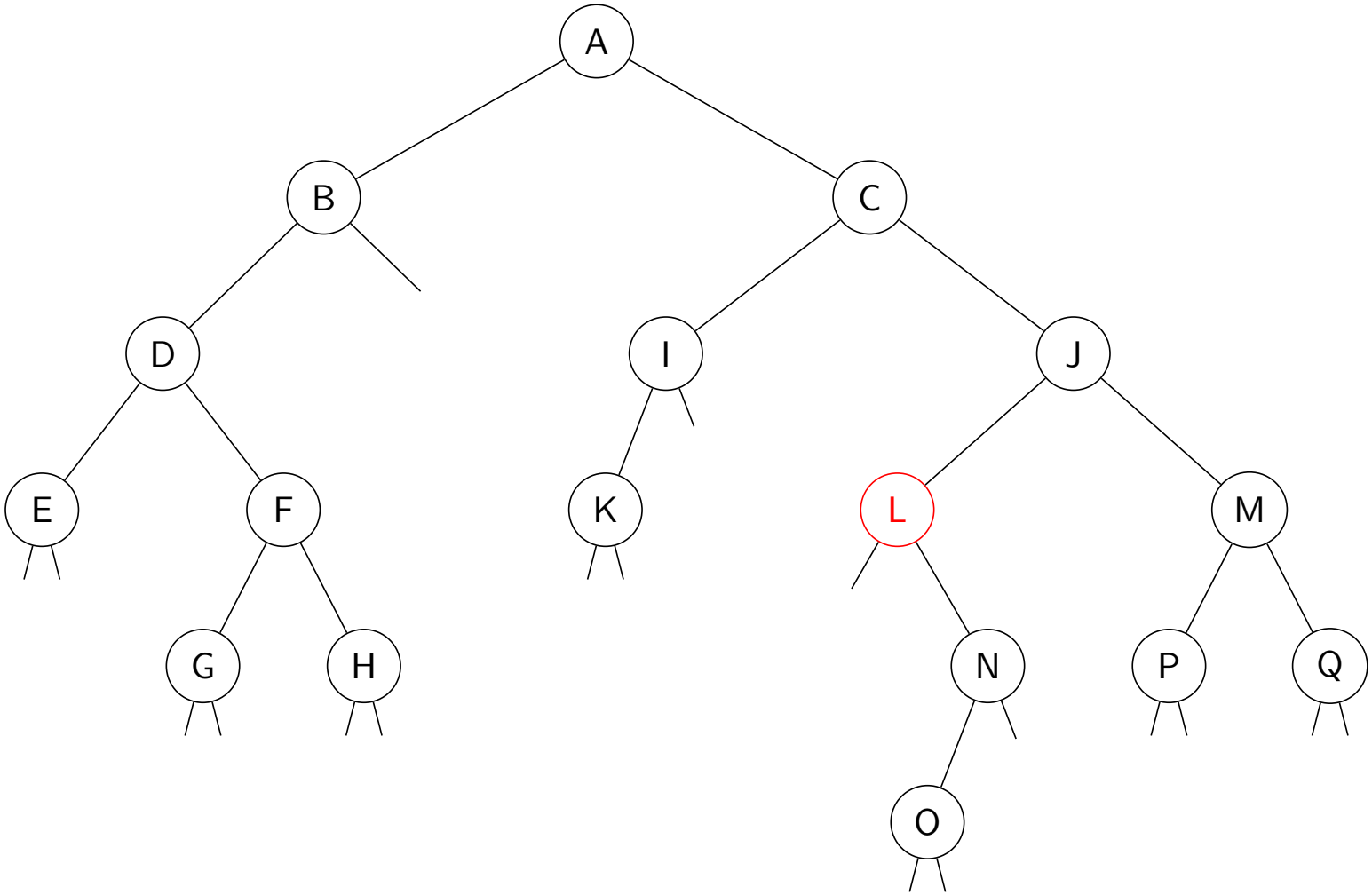
Parcours infixe : E D G F H B A K



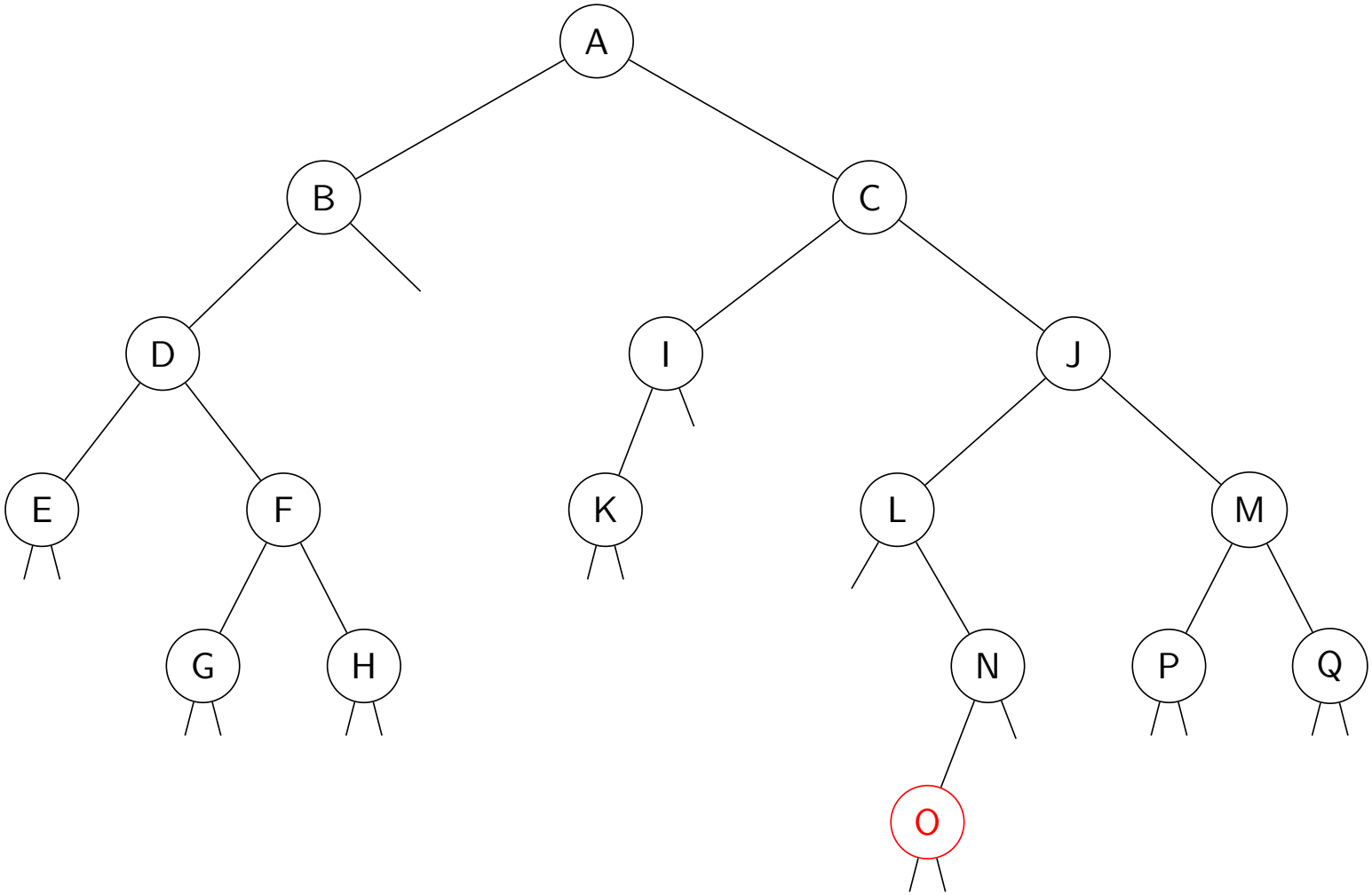
Parcours infixe : E D G F H B A K I



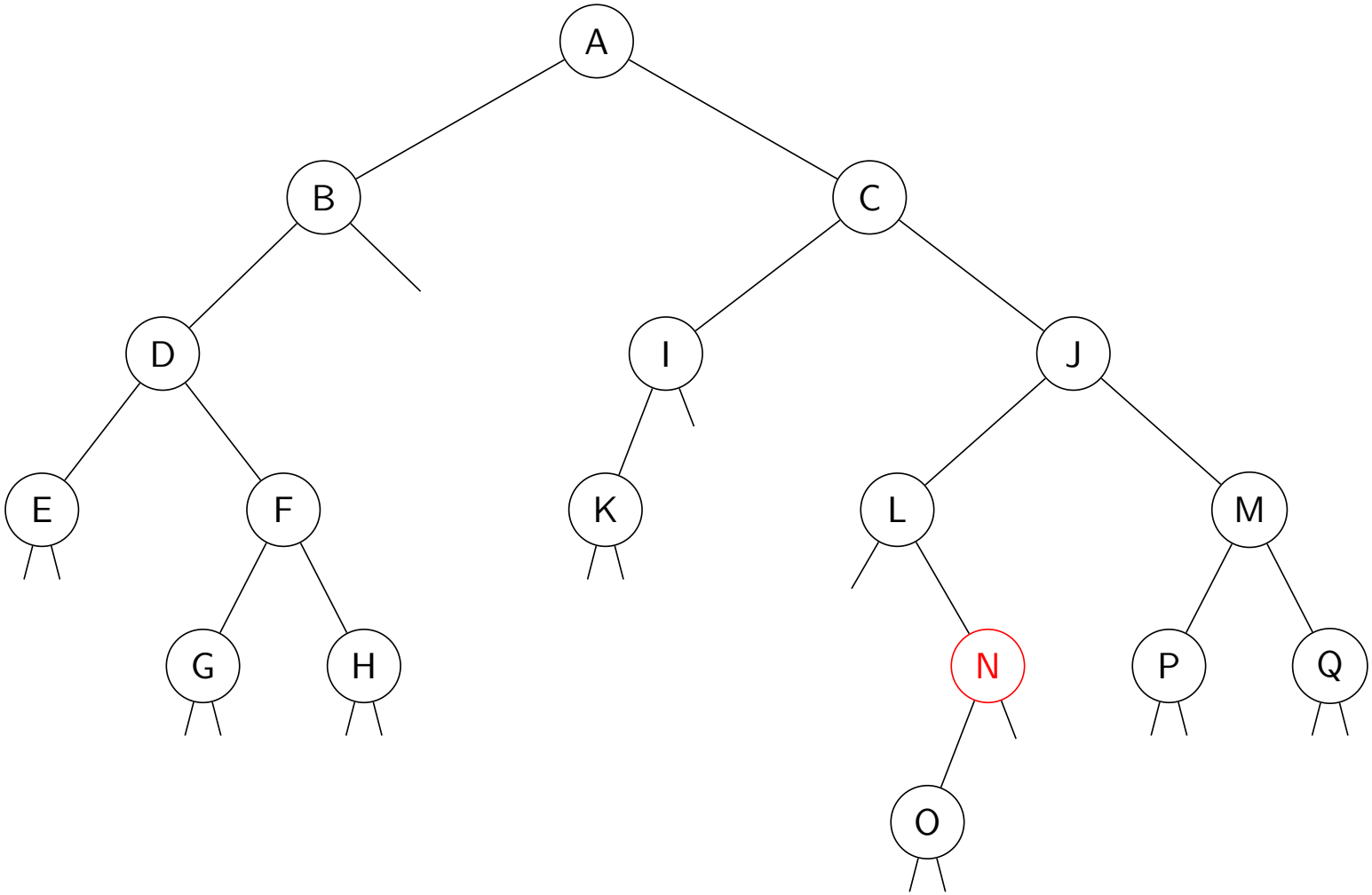
Parcours infixe : E D G F H B A K I C



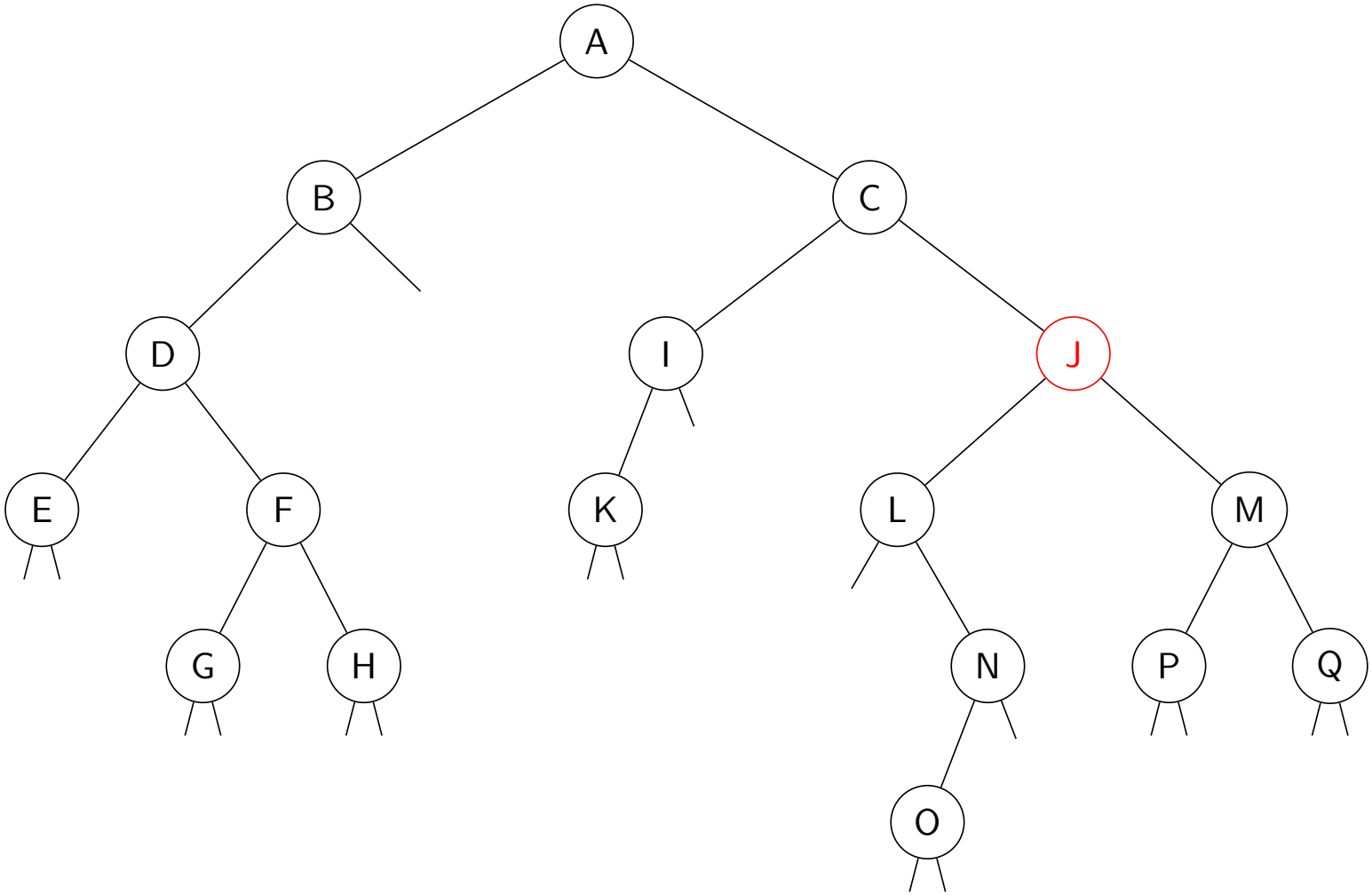
Parcours infixe : E D G F H B A K I C L



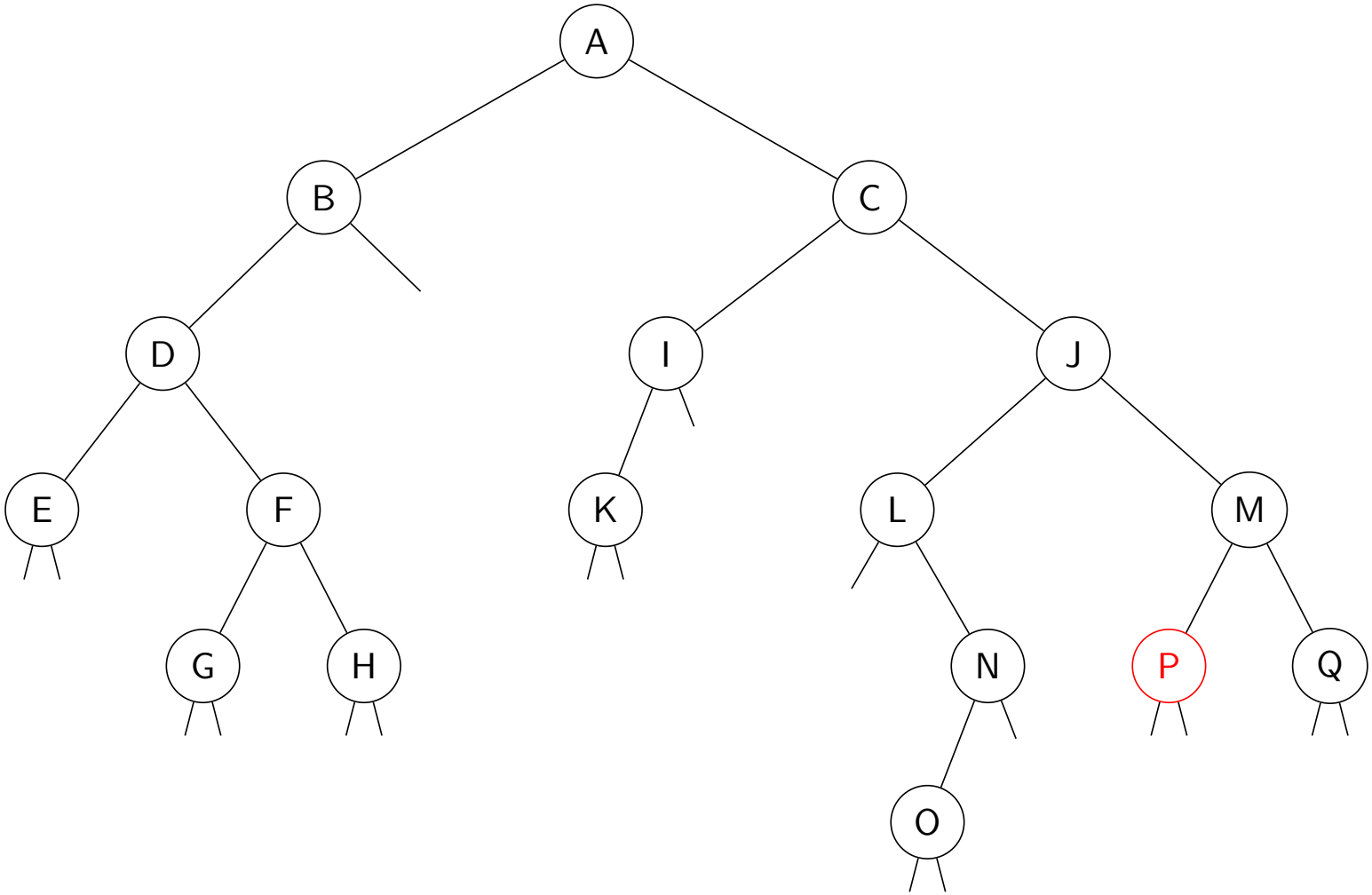
Parcours infixe : E D G F H B A K I C L O



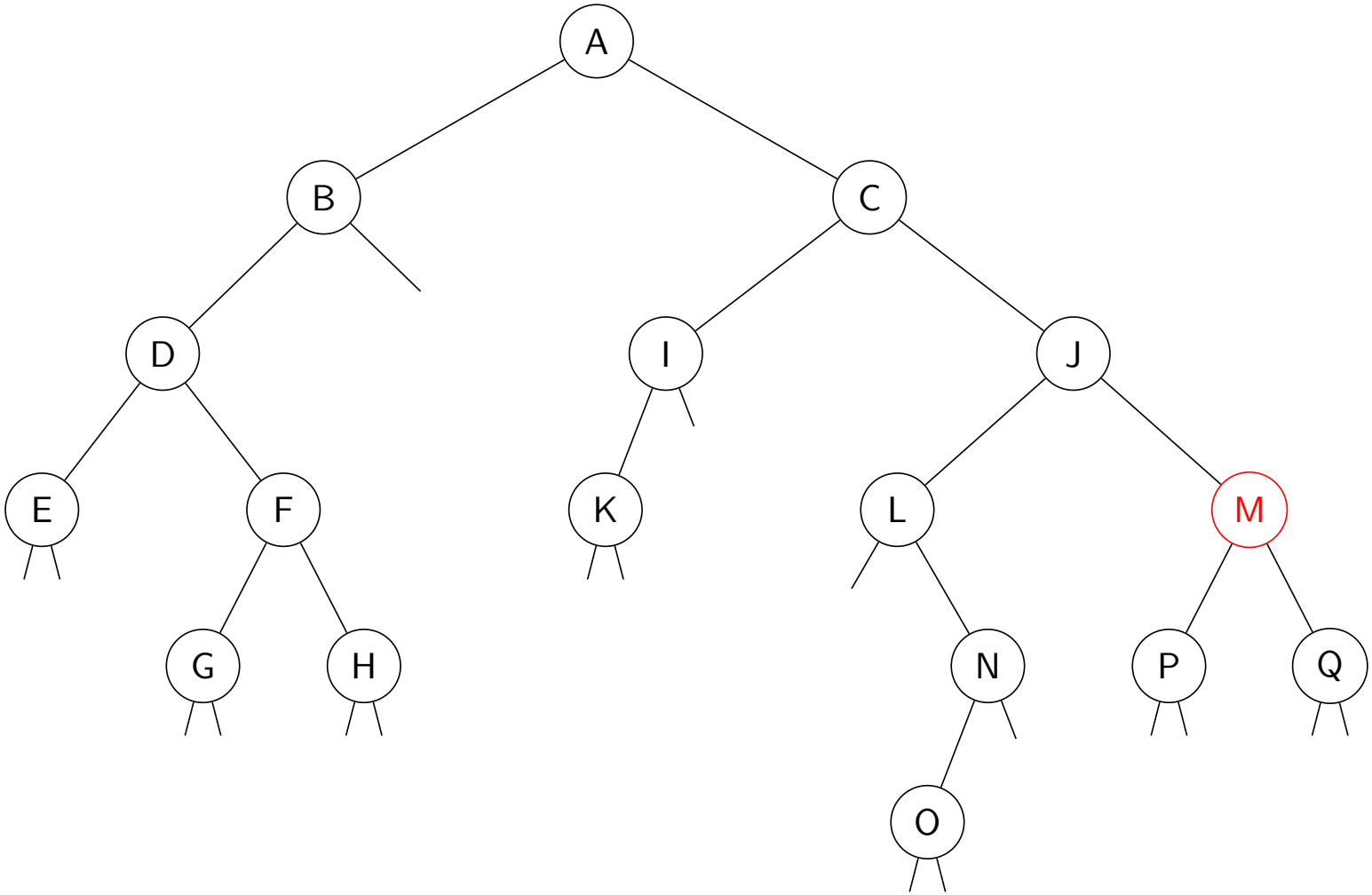
Parcours infixe : E D G F H B A K I C L O N



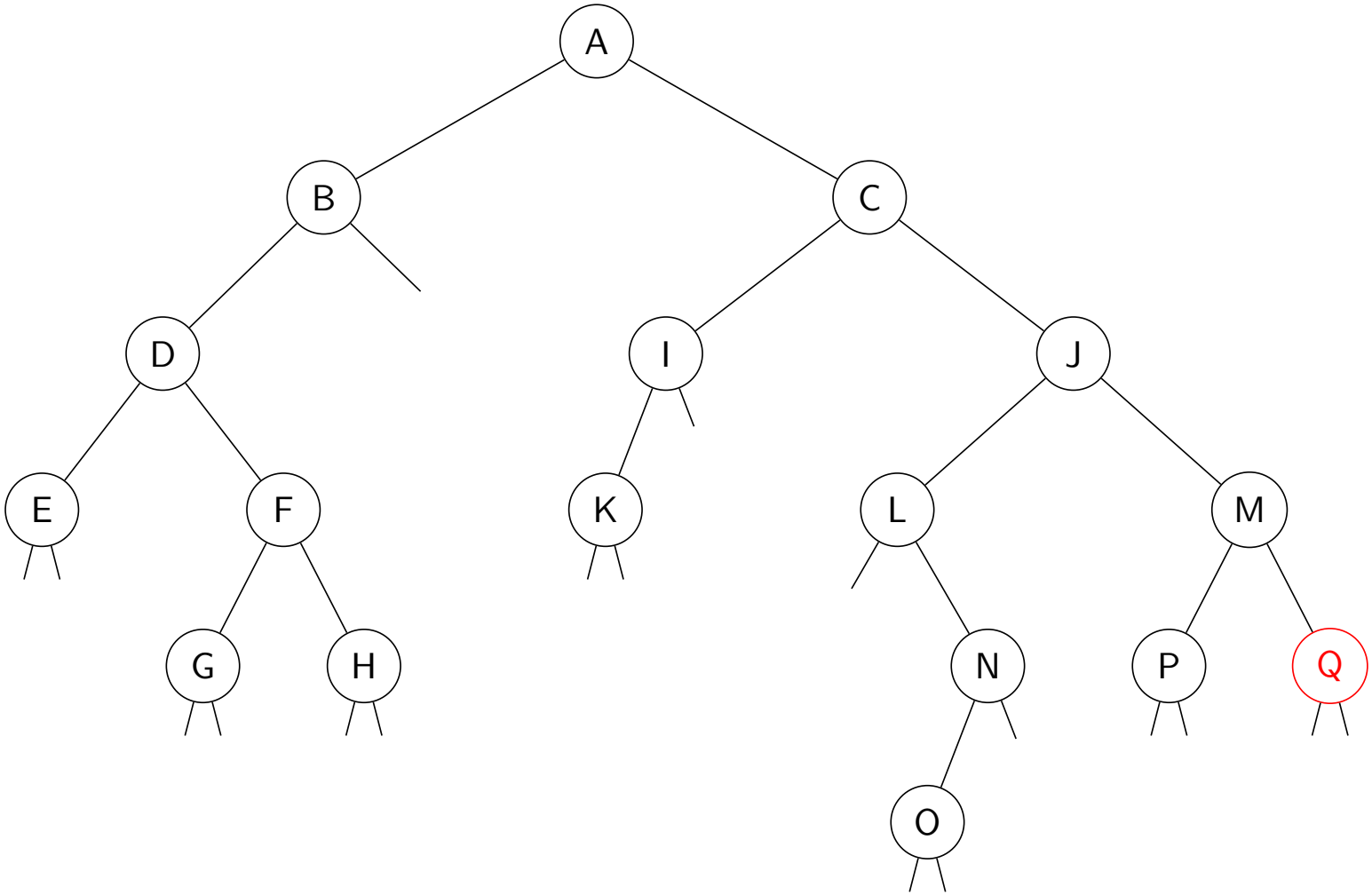
Parcours infixe : E D G F H B A K I C L O N J



Parcours infixe : E D G F H B A K I C L O N J P



Parcours infixe : E D G F H B A K I C L O N J P M



Parcours infixe : E D G F H B A K I C L O N J P M Q

Ex. 9 Avec un parcours infixe, dans quel ordre sont énumérés les nœuds des arbres `a2g`, `a2d`, `a3` et `a10` ?

Voici un algorithme, écrit en pseudo-code, d'une fonction qui réalise le parcours (et l'affichage) dans l'ordre infixe d'un arbre binaire de racine racine :

```
Fonction AFFICHER_INFIXE(racine)
    si racine ≠ nil alors                          # arbre non-vide
        AFFICHER_INFIXE(racine.gauche)
        afficher racine.valeur
        AFFICHER_INFIXE(racine.droit)
```

Ex. 10 En vous aidant de l'algorithme ci-dessus, écrire en Python une fonction `afficher_infixe(racine)` qui parcours et affiche dans l'ordre infixe les nœuds de l'arbre binaire dont la racine est le nœud `racine`. Tester votre fonction sur les arbres précédemment créés.

On donne ci-dessous les algorithmes récursifs des parcours préfixe et suffixe d'un arbre de nœud racine racine :

```
Fonction AFFICHER_PREFIXE(racine)
    si racine ≠ nil alors                          # arbre non-vide
        afficher racine.valeur
        AFFICHER_PREFIXE(racine.gauche)
        AFFICHER_PREFIXE(racine.droit)
```

```
Fonction AFFICHER_SUFFIXE(racine)
    si racine ≠ nil alors                          # arbre non-vide
        AFFICHER_SUFFIXE(racine.gauche)
        AFFICHER_SUFFIXE(racine.droit)
        afficher racine.valeur
```

Ex. 11 Écrire en Python les fonctions `afficher_prefixe(racine)` et `afficher_suffixe(racine)` qui réalisent respectivement l'affichage des nœuds correspondant au parcours préfixe et suffixe de l'arbre dont le nœud racine est `racine`. Tester ces fonctions sur les arbres déjà créés.

On peut se demander si l'ordre d'affichage des nœuds associé à un parcours donné (par exemple préfixe) caractérise de façon unique l'arbre binaire qui le produit [3]. Il n'en est rien, comme le montre l'exemple des arbres `a2g` et `a2d`, qui produisent le même affichage préfixe : `A B`.

En revanche, si on réalise un affichage préfixe qui insère la position des arbres vides rencontrés (à l'aide d'un mot clef, comme `nil`), il devient possible de coder de façon unique chaque arbre. Par exemple, `a2g` et `a2d` ont respectivement pour affichage préfixe : `A B nil nil nil` et `A nil B nil nil`.

- Le parcours *en largeur* d'un arbre binaire peut être décrit comme un parcours «de gauche à droite sur un même niveau» et «de haut en bas», c'est à dire du niveau le plus haut (celui du nœud racine) vers les niveaux les plus bas (auxquels appartiennent les nœuds feuilles).
- Sur l'arbre de la page 7 ce parcours affiche :

`A B C D I J E F K L M G H N P Q O`

Ex. 12 Quel est l'affichage en largeur de l'arbre `a10` ?

3. Dans l'idée d'obtenir ainsi un codage possible de l'arbre binaire.

- Écrit en pseudo-code, un algorithme de parcours en largeur d'un arbre binaire est le suivant :

```
Fonction AFFICHER_LARGEUR(racine)
    Créer une file f vide
    ajouter racine à f
    tant que f n'est pas vide faire
        lire le nœud n en tête de f
        si n ≠ nil alors
            afficher n.valeur
            si n.gauche ≠ nil alors
                ajouter n.gauche dans f
            si n.droit ≠ nil alors
                ajouter n.droit dans f
```

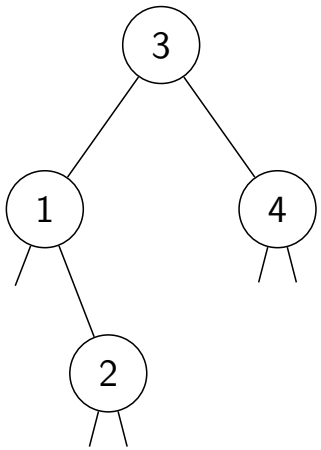
Ex. 13 Écrire en Python, en utilisant la classe `File` (fournie dans le fichier source disponible sur le site web du cours), un algorithme de parcours en largeur d'un arbre binaire. Tester votre algorithme sur les différents arbres déjà créés.

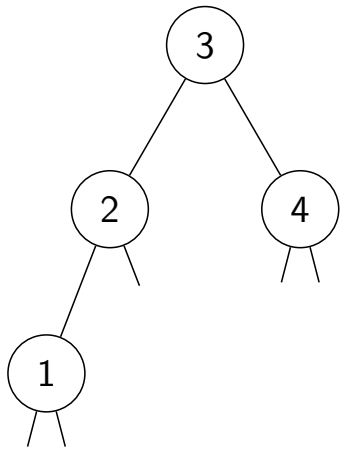
Sommaire

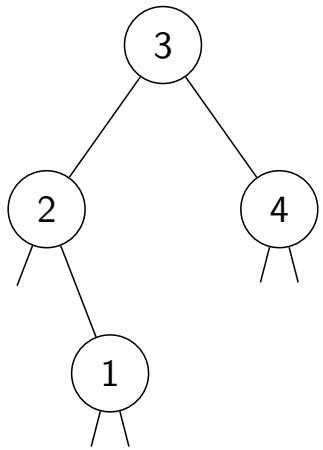
Arbres binaires de recherche

- Les arbres binaires de recherche (en abrégé des ABR) sont des arbres binaires :
  1. dont les valeurs des nœuds sont d'un type comparable (par exemple un type numérique ou chaîne de caractère);
  2. tels que, pour tout nœud, toutes les valeurs contenues dans son sous-arbre arbre gauche (respectivement dans son sous-arbre droit) sont plus petites (respecivement plus grandes) que celle du nœud.

Ex. 14 On donne ci-dessous trois arbres binaires. Lesquels sont des ABR ? Peut-on avoir deux ABR stockant le même ensemble de valeurs ? Donner tous les ABR ayant trois nœuds.







- On remarque que le parcours infixe d'un ABR affiche nécessairement les valeurs classées dans l'ordre croissant [4]
- Certaines opérations sont facilitées pour un ABR, par exemple la recherche d'une valeur.
- On donne ci-dessous l'algorithme, en pseudo-code, d'une fonction appartient(racine, v) qui renvoie vrai si v est stockée dans un des nœuds de l'ABR racine et faux sinon.

```
Fonction APPARTIENT(racine, v)
    si racine = nil alors
        renvoyer faux
    sinon
        si v < racine.valeur alors
            renvoyer APPARTIENT(racine.gauche, v)
        sinon si v = racine.valeur alors
            renvoyer vrai
        sinon
            renvoyer APPARTIENT(racine.droit, v)
```

4. D'ailleurs la réciproque est vraie : si l'affichage infixe d'un arbre est dans l'ordre croissant, c'est un ABR.

Ex. 15 Écrire en Python une fonction `appartient(racine, v)` qui renvoie vrai si la valeur `v` est dans l'arbre `racine` et faux sinon. Tester cette fonction sur un ABR préalablement créé «à la main».

On donne ci-dessous un algorithme, en langage semi-formel, d'une fonction ajouter(racine, v) qui renvoie le nœud racine d'un *nouvel* ABR contenant v et les valeurs déjà présentes.

```
Fonction AJOUTER(racine, v)
    si racine = nil alors
        renvoyer Noeud(nil, v, nil)
    si v < racine.valeur alors
        renvoyer Noeud(AJOUTER(racine.gauche, v), racine.valeur, ra-
        cine.droit)
    sinon
        renvoyer Noeud(racine.gauche, racine.valeur,AJOUTER(racine.droit,
        v))
```

Ex. 16 Écrire l'algorithme précédent en Python et le tester pour construire des ABR.

- La complexité de la recherche dans un ABR est directement liée à la hauteur de celui-ci. On a vu précédemment que la hauteur h est majorée par la taille N de l'arbre.
- Bien sûr, on cherche à ce que l'ABR utilisé soit de la hauteur la plus petite possible. Il existe des techniques permettant, lors de la création de l'ABR, de garder la propriété d'ABR tout en ayant un arbre le plus «équilibré» possible.
- Pratiquement, on arrive dans certains cas à une majoration de la forme

$$h \leqslant C.\log_2(N) \quad \text{où } C \simeq 1,44$$

qui montre que la complexité de la recherche dans les arbres obtenus est logarithmique, donc très efficace.

Ex. 17 Dans un ABR, où se trouve le plus petit élément ? En déduire le code en Python d'une fonction `minimum(racine)` qui renvoie le plus petit élément de l'ABR `racine`. Si l'arbre est vide, on renverra `None`.

Ex. 18

1. Écrire une fonction `remplir(racine, t)` qui ajoute tous les éléments de l'arbre binaire `racine` dans le tableau `t`, dans l'ordre infixe. On ajoutera les éléments `x` dans `t` avec la fonction `t.append(x)`.
2. Créer une fonction `lister(racine)` qui renvoie un tableau contenant tous les éléments de l'ABR `racine` classés par ordre croissant.
3. Déduire de ce qui précède l'écriture d'une fonction `trier(t)` qui renvoie un tableau contenant les éléments du tableau `t` classés par ordre croissant.