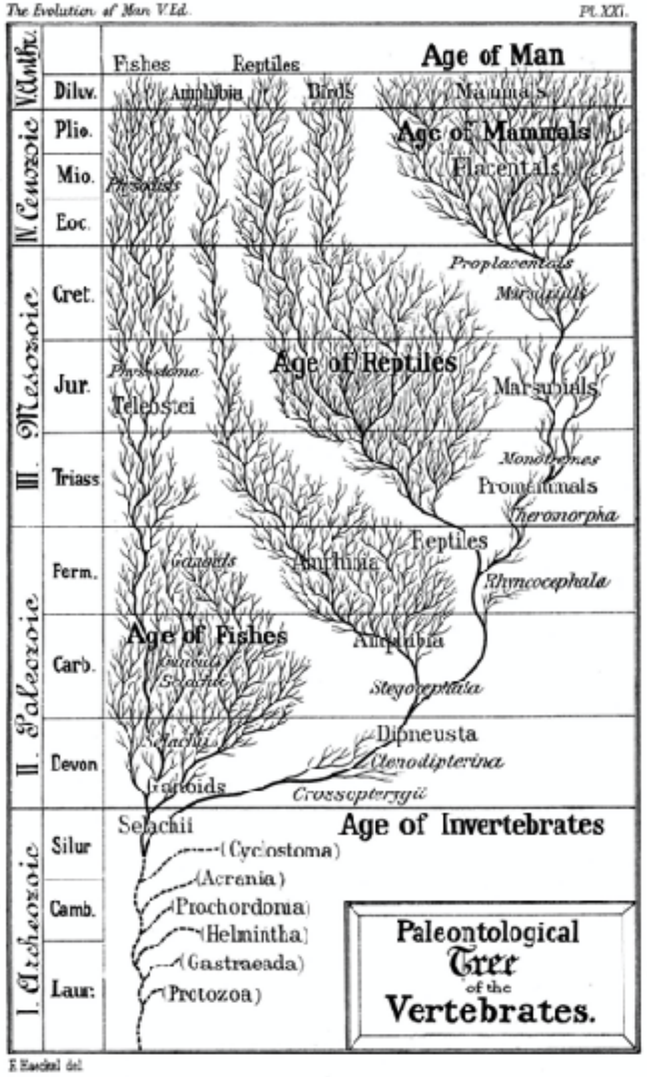
5.1. Arbres

HEIG VD

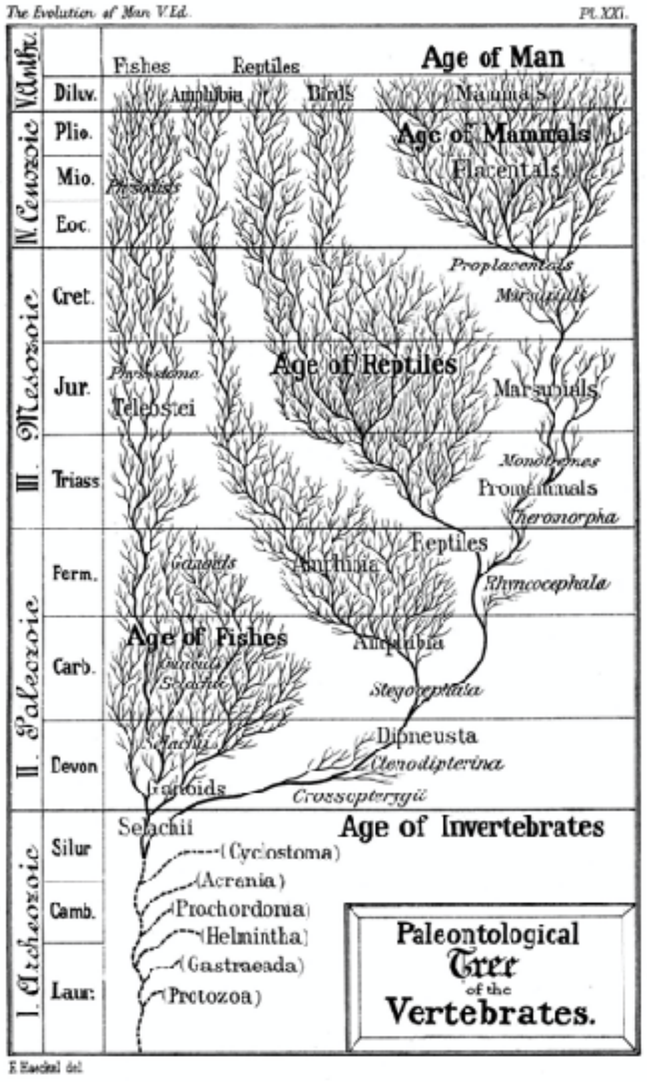
1 . Introduction

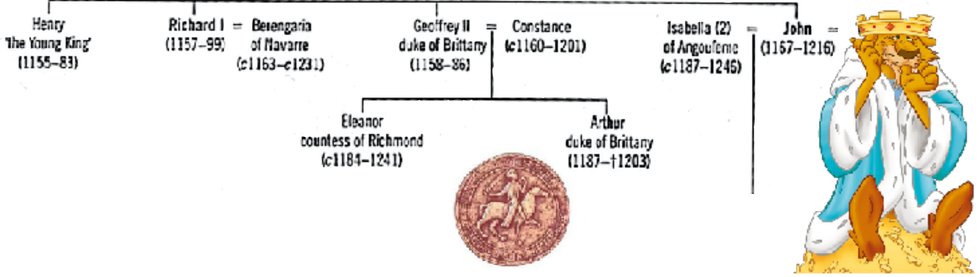
Exemples

Exemples

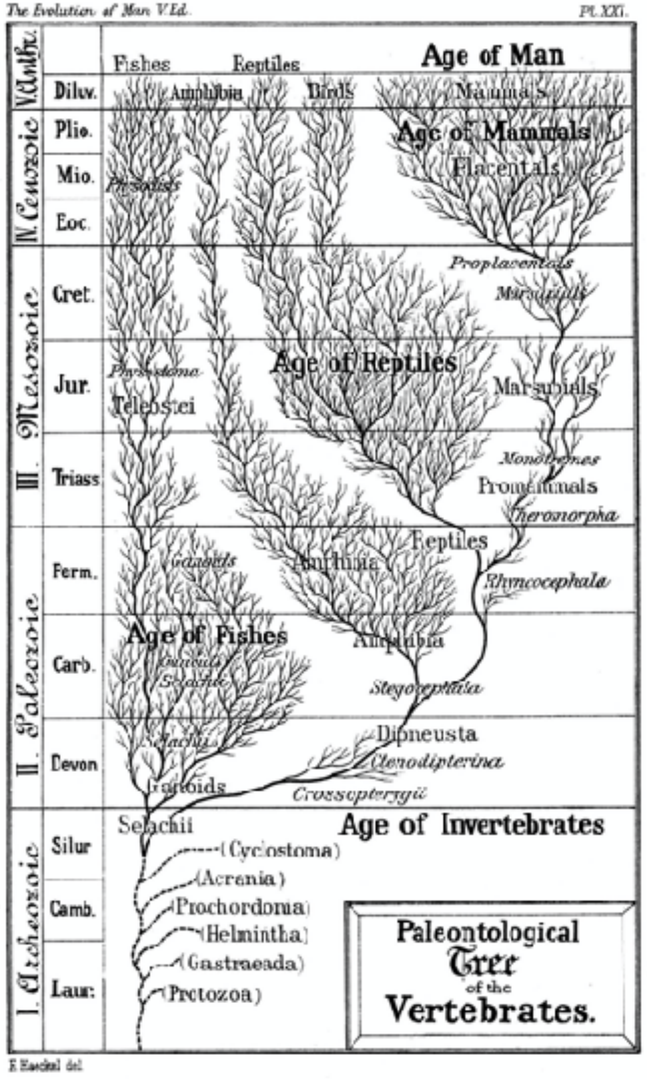

The Evolution of Man V.Ed.
Pl. XXI.
Fishes
Reptiles
Age of Man
Amphibia
Birds
Mamma's
Age of Mammals
Placentals
Proplacentals
Marsupials
Age of Reptiles
Teleostei
Monotremes
Promammals
Theromorpha
Reptiles
Rhyncocephala
Amphibia
Stegocephala
Dipneusta
Ctenodipterina
Crossopterygii
Selachii
Age of Fishes
Age of Invertebrates
(Cyclostoma)
(Acrania)
(Prochordonia)
(Helmintha)
(Gastraeada)
(Protozoa)
Paleontological Tree of the Vertebrates.
V. Cenozoic
IV. Cenozoic
III. Mesozoic
II. Paleozoic
I. Archeozoic
Diluv.
Plio.
Mio.
Eoc.
Cret.
Jur.
Triass
Ferm.
Carb.
Devon
Silur
Camb.
Laur.

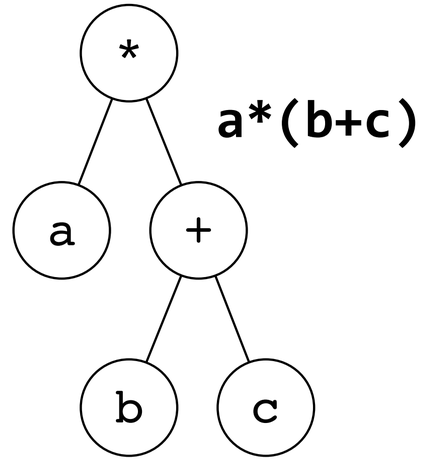
Exemples

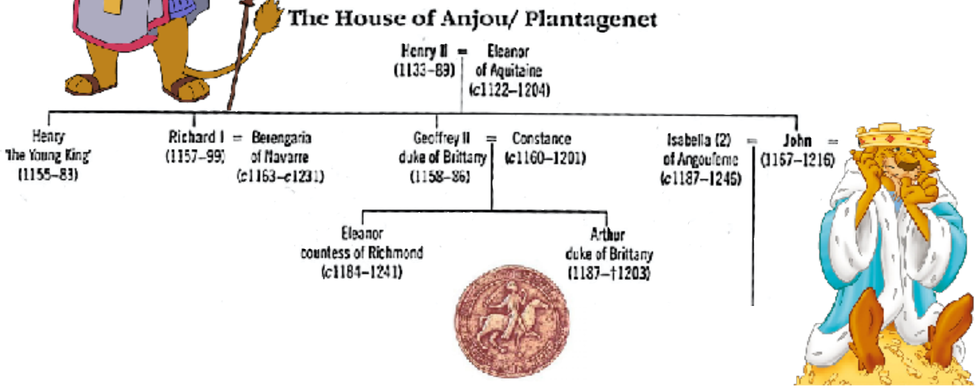

The Evolution of Man V.Ed.
Pl. XXI.
Age of Man
Fishes
Reptiles
Amphibia
Birds
Mammals
Age of Mammals
Placentals
Proplacentals
Marsupials
Age of Reptiles
Marsupials
Monotremes
Prototheria
Theromorpha
Reptiles
Rhyncocephala
Amphibia
Stegocephala
Age of Fishes
Dipneusta
Ctenodipterina
Crossopterygii
Selachii
Age of Invertebrates
(Cyclostoma)
(Acrania)
(Prochordonia)
(Helmintha)
(Gastraeada)
(Protozoa)
Paleontological Tree of the Vertebrates.
Diluv.
Plio.
Mio.
Eoc.
Cret.
Jur.
Triass
Perm.
Carb.
Devon
Silur
Camb.
Laur.
V. Anthr.
IV. Cenozoic
III. Mesozoic
II. Paleozoic
I. Archeozoic


The House of Anjou/ Plantagenet
Henry II (1133–89)
Eleanor of Aquitaine (c1122–1204)
Henry 'the Young King' (1155–83)
Richard I (1157–99)
Berengaria of Navarre (c1163–c1231)
Geoffrey II duke of Brittany (1158–86)
Constance (c1160–1201)
Isabella (2) of Angoulême (c1187–1246)
John (1167–1216)
Eleanor countess of Richmond (c1184–1241)
Arthur duke of Brittany (1187–†1203)

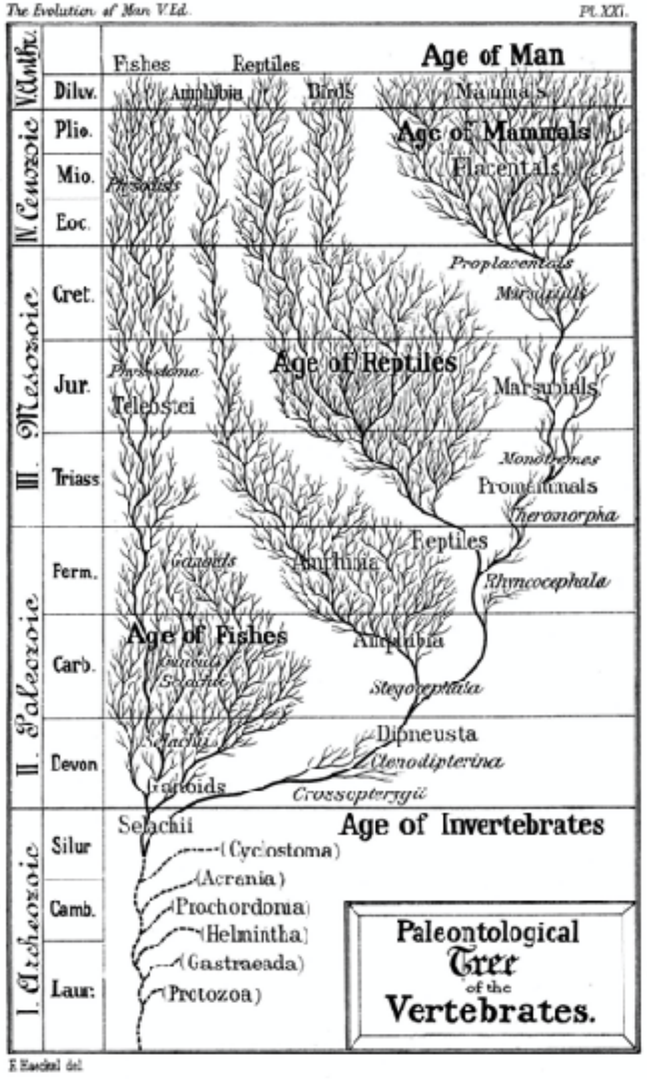
Exemples

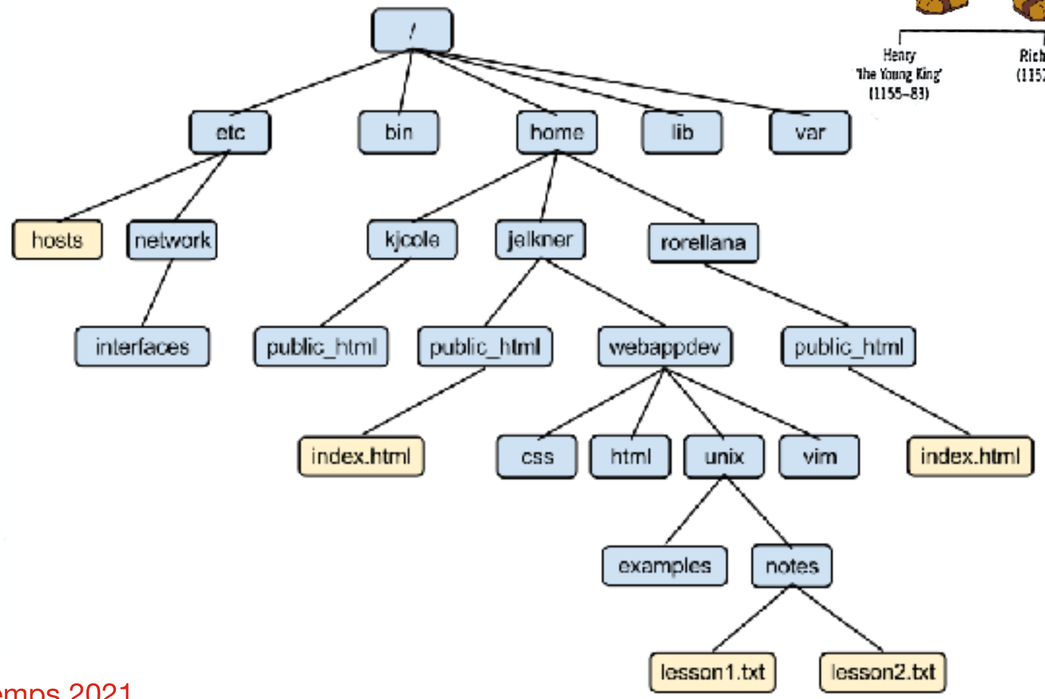

The Evolution of Man V.Ed.
Pl. XXI.
Age of Man
Fishes
Reptiles
Amphibia
Birds
Mamma's
Age of Mammals
Placentals
Proplacentals
Marsupials
Age of Reptiles
Teleostei
Monotremes
Promammals
Theromorpha
Reptiles
Rhyncocephala
Amphibia
Age of Fishes
Stegocephala
Dipneusta
Ctenodipterina
Crossopterygii
Selachii
Age of Invertebrates
(Cyclostoma)
(Acrania)
(Prochordonia)
(Helmintha)
(Gastraeada)
(Protozoa)
Paleontological Tree of the Vertebrates.
V. Cenozoic
IV. Mesozoic
III. Paleozoic
II. Archeozoic
I. Azoic
Diluv.
Plio.
Mio.
Eoc.
Cret.
Jur.
Triass
Perm.
Carb.
Devon
Silur
Camb.
Laur.

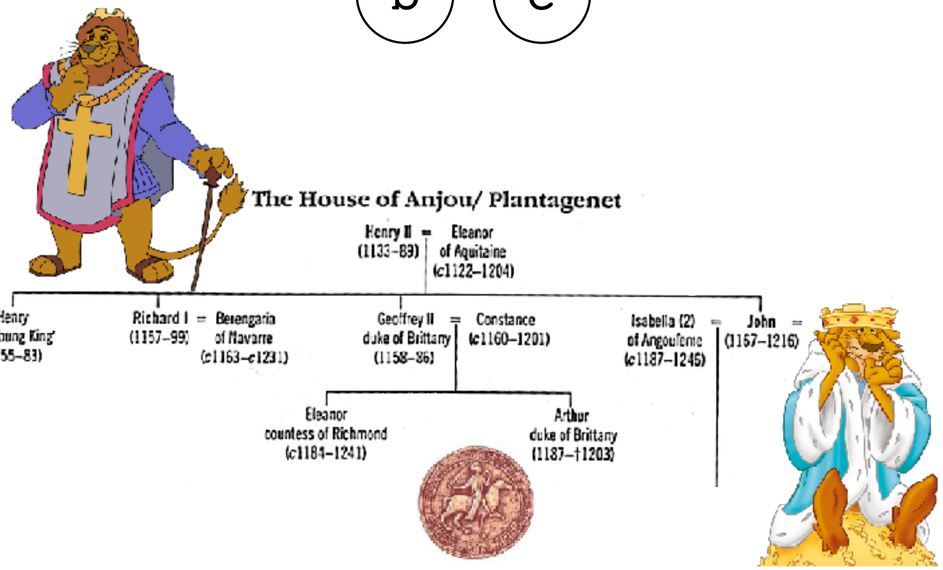

*
a*(b+c)
a
+
b
c

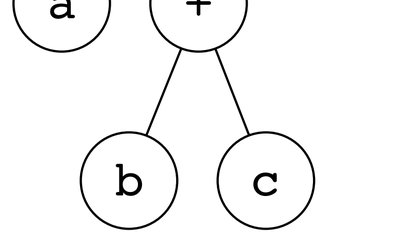

The House of Anjou/ Plantagenet
Henry II (1133–89)
Eleanor of Aquitaine (c1122–1204)
Henry 'the Young King' (1155–83)
Richard I (1157–99)
Berengaria of Navarre (c1163–c1231)
Geoffrey II duke of Brittany (1158–86)
Constance (c1160–1201)
Isabella (2) of Angoulême (c1187–1246)
John (1167–1216)
Eleanor countess of Richmond (c1184–1241)
Arthur duke of Brittany (1187–†1203)

Exemples


The Evolution of Man V.Ed.
Pl. XXI.
Age of Man
Fishes
Reptiles
Amphibia
Birds
Mamma's
Age of Mammals
Placentals
Proplacentals
Marsupials
Age of Reptiles
Monotremes
Prototheria
Theromorpha
Rhyncocephala
Ganoids
Amphibia
Age of Fishes
Stegocephala
Dipneusta
Ctenodipterina
Crossopterygii
Selachii
Age of Invertebrates
(Cyclostoma)
(Acrania)
(Prochordonia)
(Helmintha)
(Gastraeada)
(Protozoa)
Paleontological Tree of the Vertebrates.
Archæozoic
Paleozoic
Mesozoic
Cenozoic
Quatr.
Plio.
Mio.
Eoc.
Cret.
Jur.
Triass
Ferm.
Carb.
Devon
Silur
Camb.
Laur.

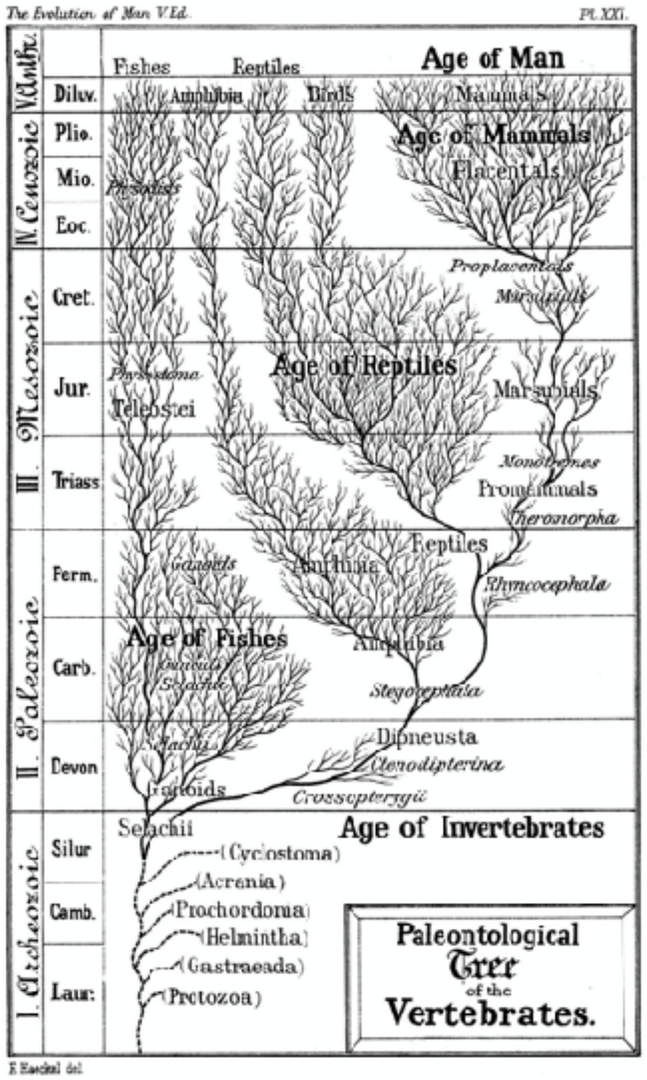

/
etc
bin
home
lib
var
hosts
network
kjcole
jelkner
rorellana
interfaces
public_html
public_html
webappdev
public_html
index.html
css
html
unix
vim
index.html
examples
notes
lesson1.txt
lesson2.txt

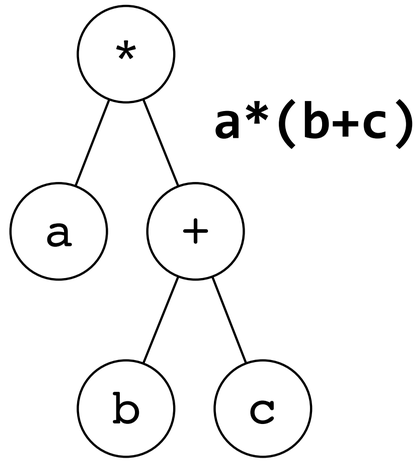

The House of Anjou/ Plantagenet
Henry II (1133–89) = Eleanor of Aquitaine (c1122–1204)
Henry 'the Young King' (1155–83)
Richard I (1157–99) = Berengaria of Navarre (c1163–c1231)
Geoffrey II duke of Brittany (1158–86) = Constance (c1160–1201)
Isabella (2) of Angoulême (c1187–1246)
John (1167–1216)
Eleanor countess of Richmond (c1184–1241)
Arthur duke of Brittany (1187–†1203)

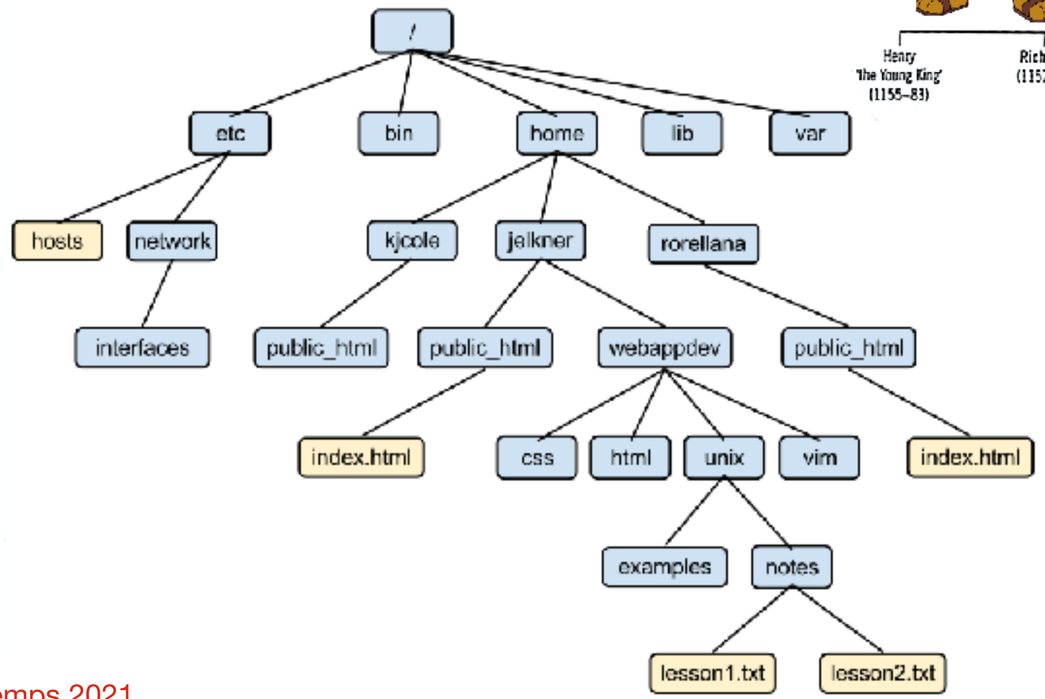

*
a*(b+c)
a
+
b
c

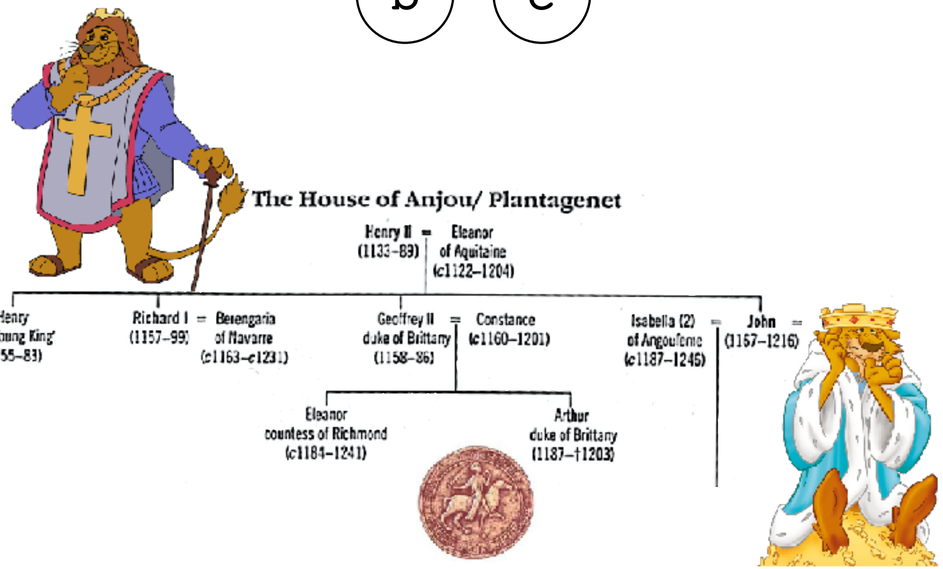
Exemples

set	Set (class template)
multiset	Multiple-key set (class template)
map	Map (class template)
multimap	Multiple-key map (class template)



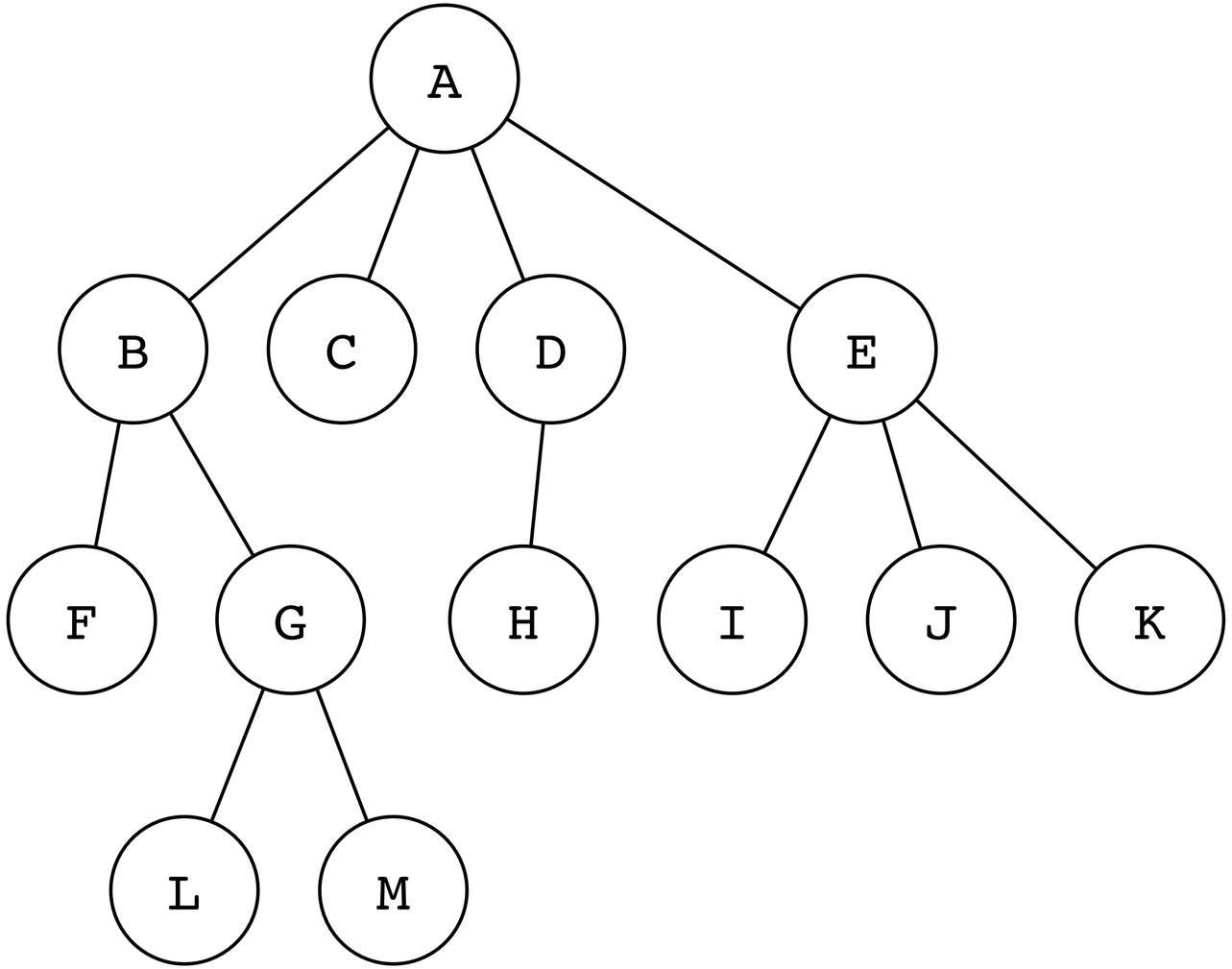






Qu’est-ce qu’un arbre?

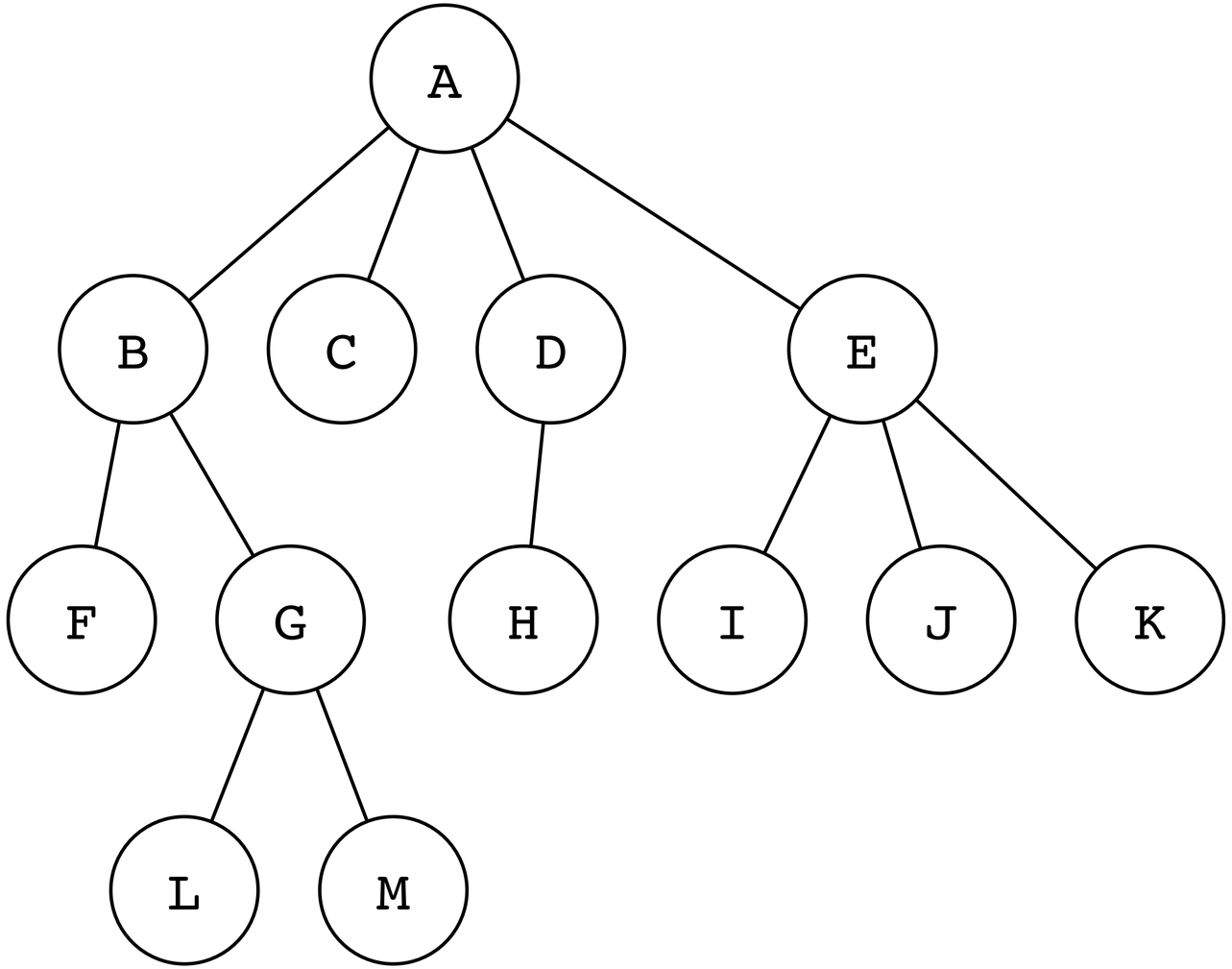
Extension du concept de liste avec plusieurs éléments suivants



Qu’est-ce qu’un arbre?

Extension du concept de liste avec plusieurs éléments suivants

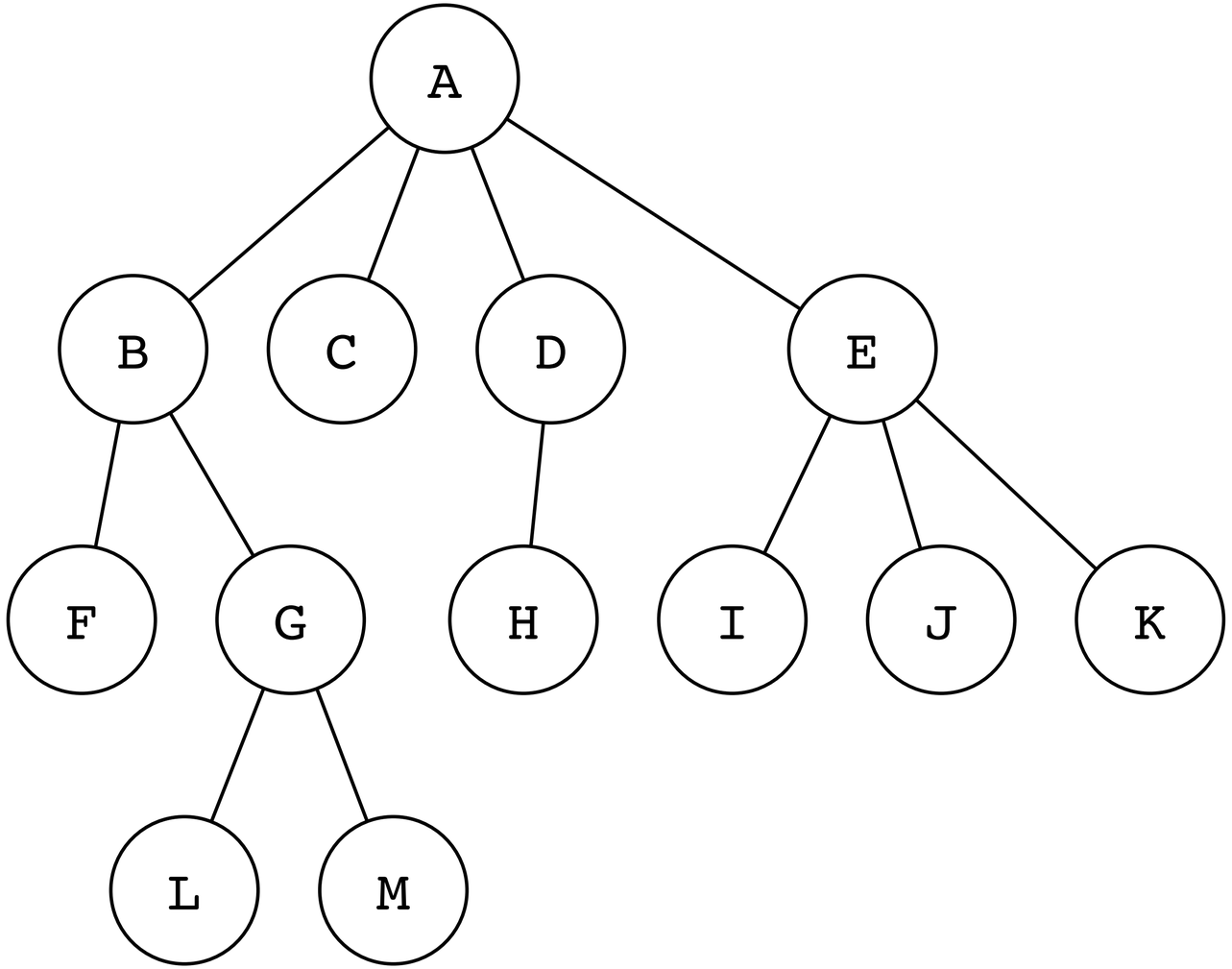
- Noeuds (maillons)



Qu'est-ce qu'un arbre?

Extension du concept de liste avec plusieurs éléments suivants

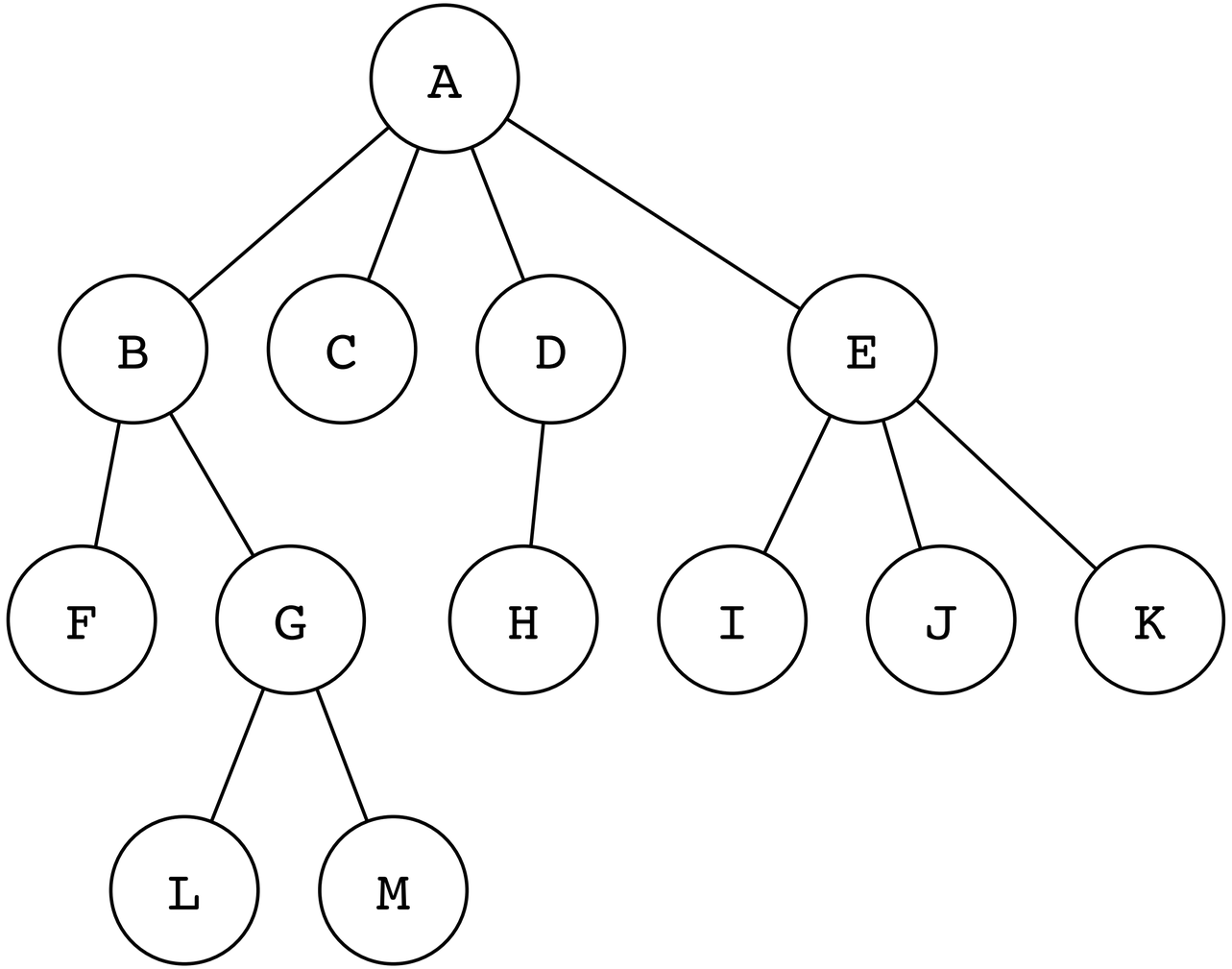
- Noeuds (maillons)
- Parent (précédent)



Qu'est-ce qu'un arbre?

Extension du concept de liste avec plusieurs éléments suivants

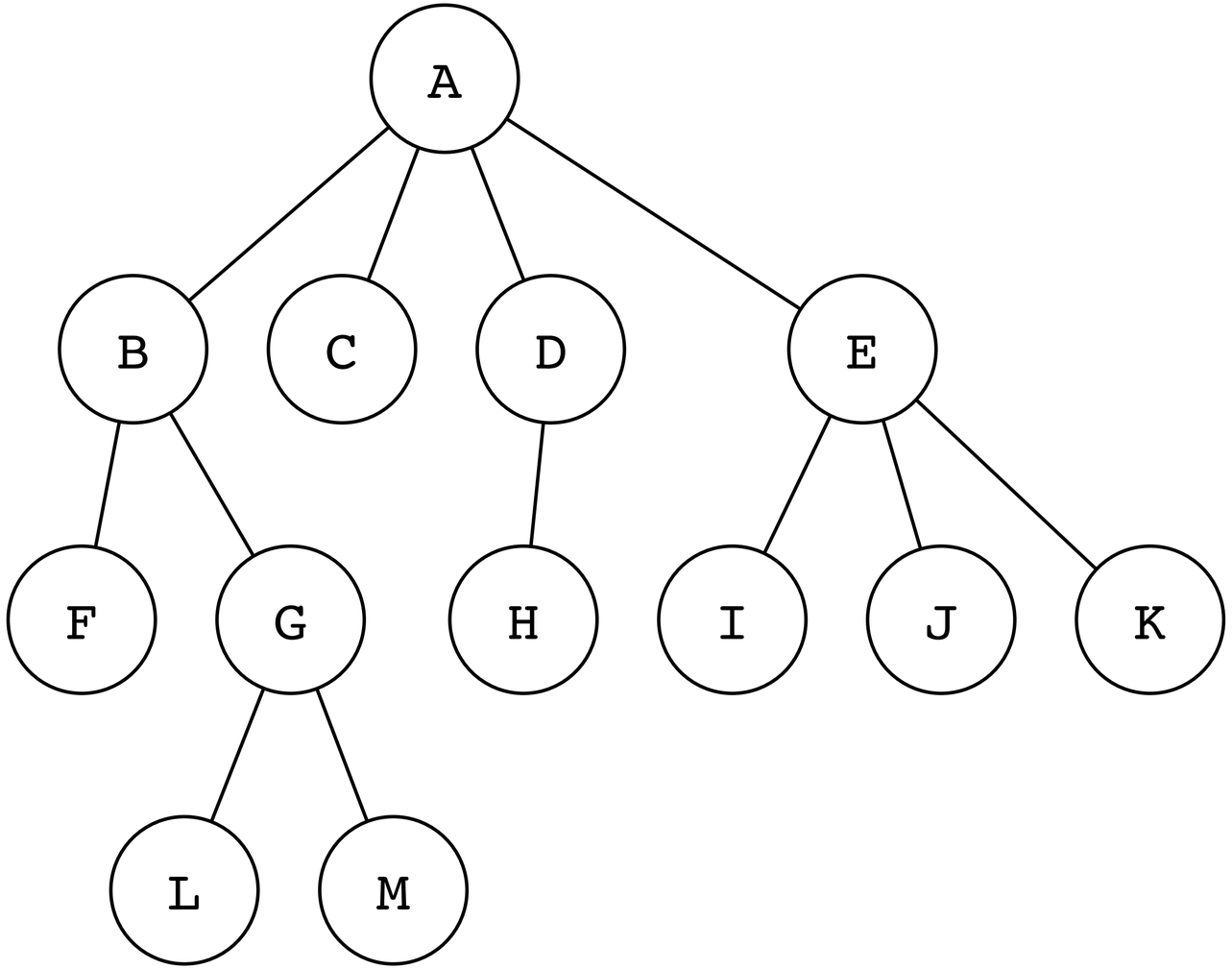
- Noeuds (maillons)
- Parent (précédent)
- Enfants (suivant)



Qu'est-ce qu'un arbre?

Extension du concept de liste avec plusieurs éléments suivants

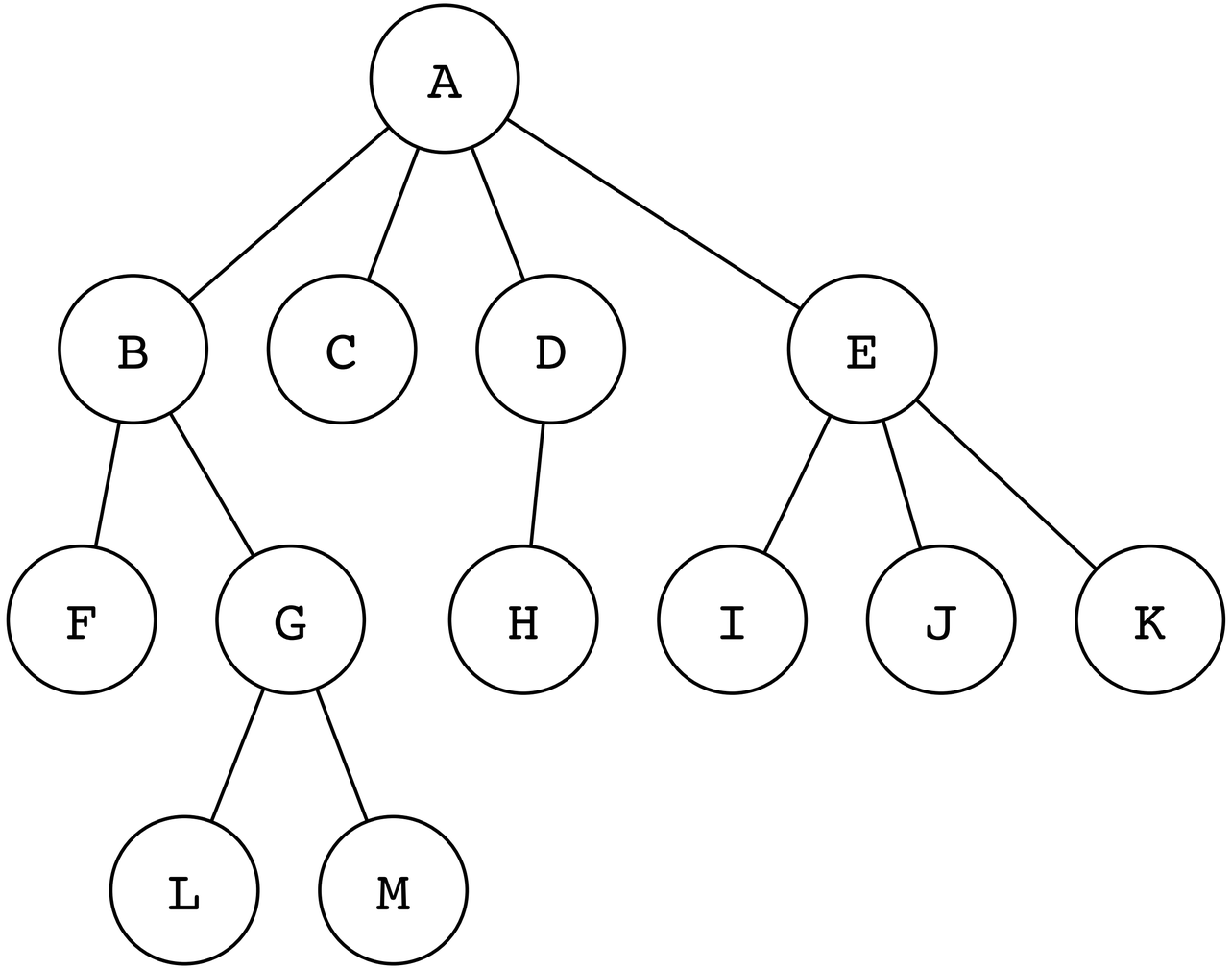
- Noeuds (maillons)
- Parent (précédent)
- Enfants (suivant)
- Racine (début)



Qu'est-ce qu'un arbre?

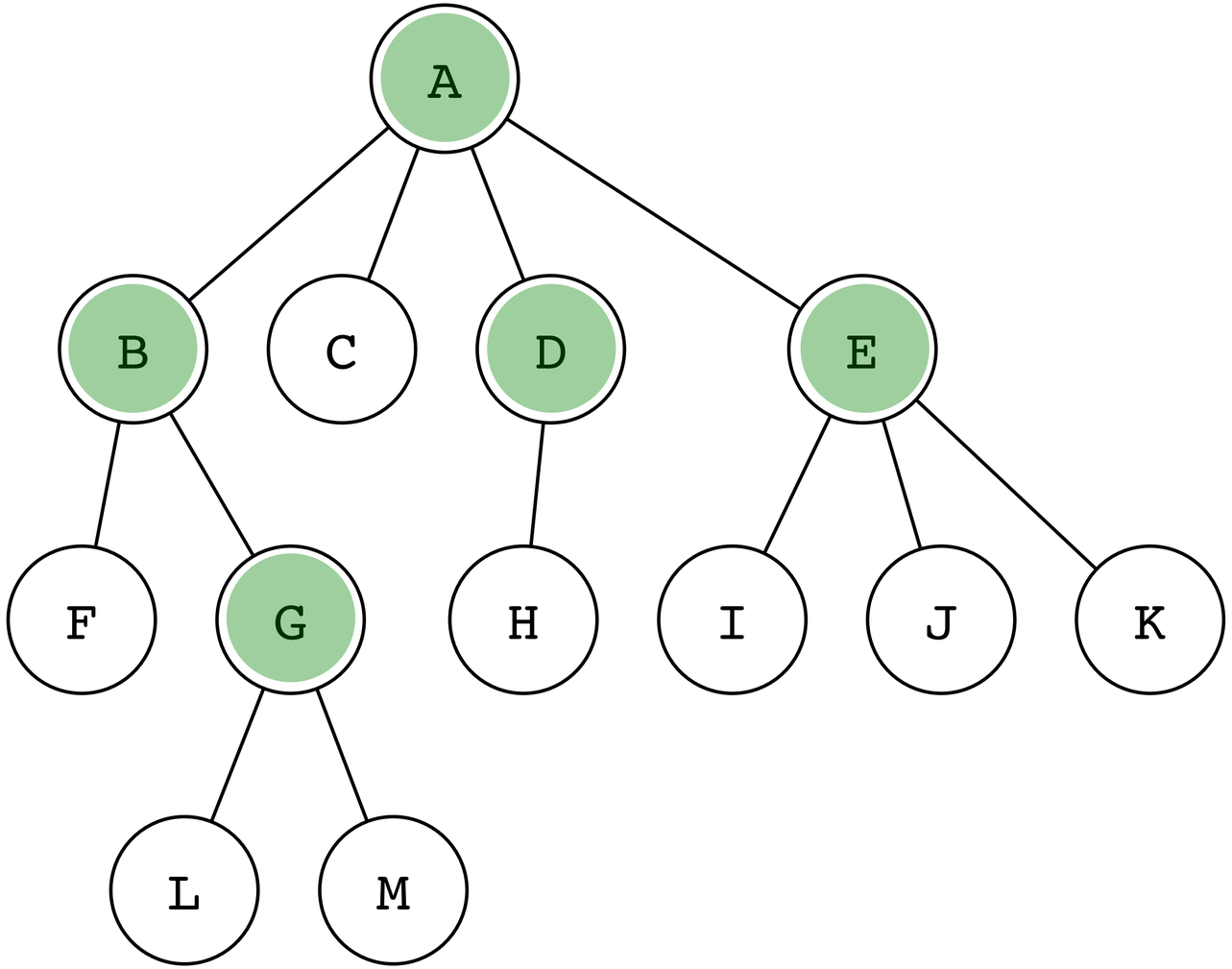
Extension du concept de liste avec plusieurs éléments suivants

- Noeuds (maillons)
- Parent (précédent)
- Enfants (suivant)
- Racine (début)
- Etiquettes



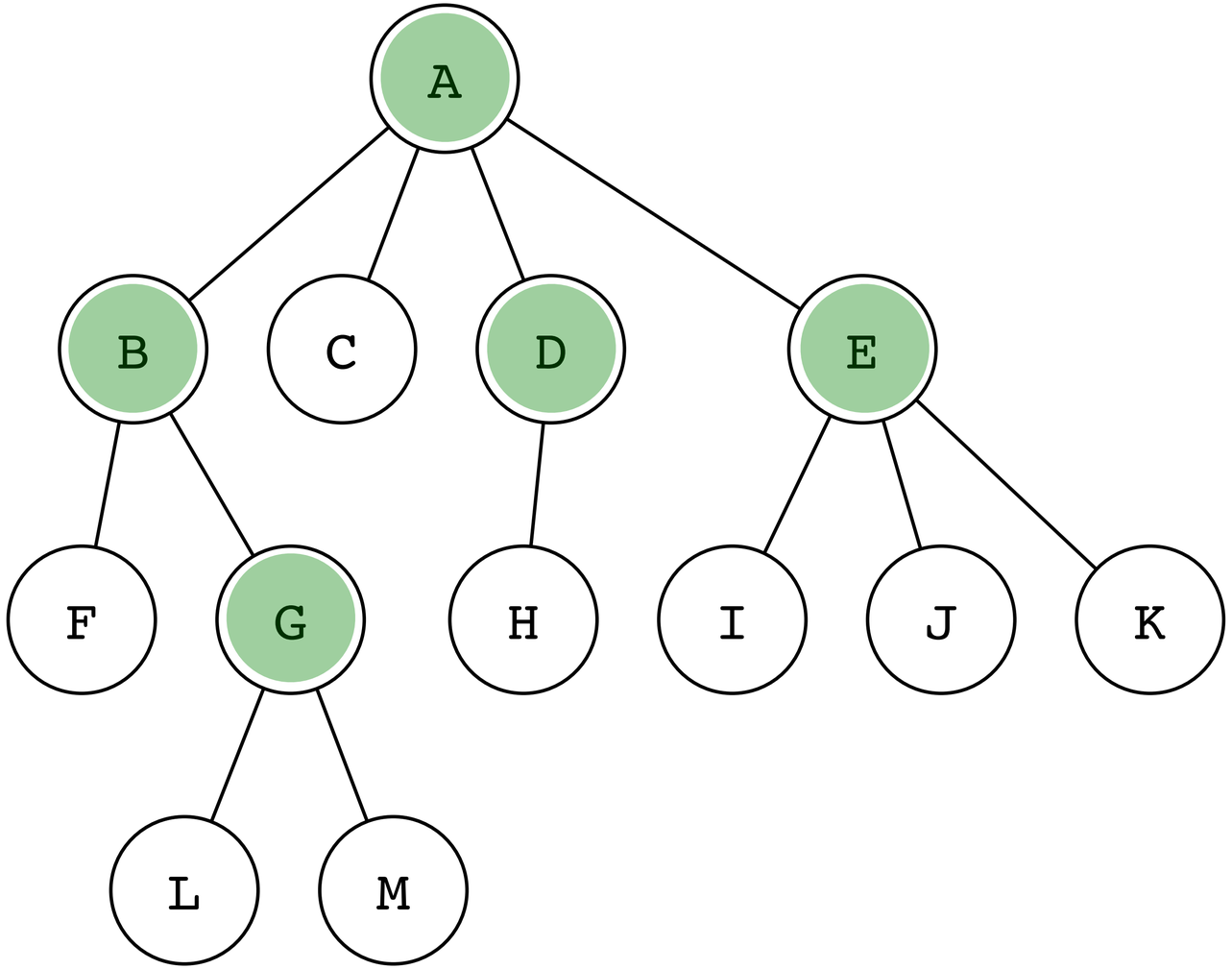
Nomenclature des noeuds

- Noeuds internes : ont un ou plusieurs enfants



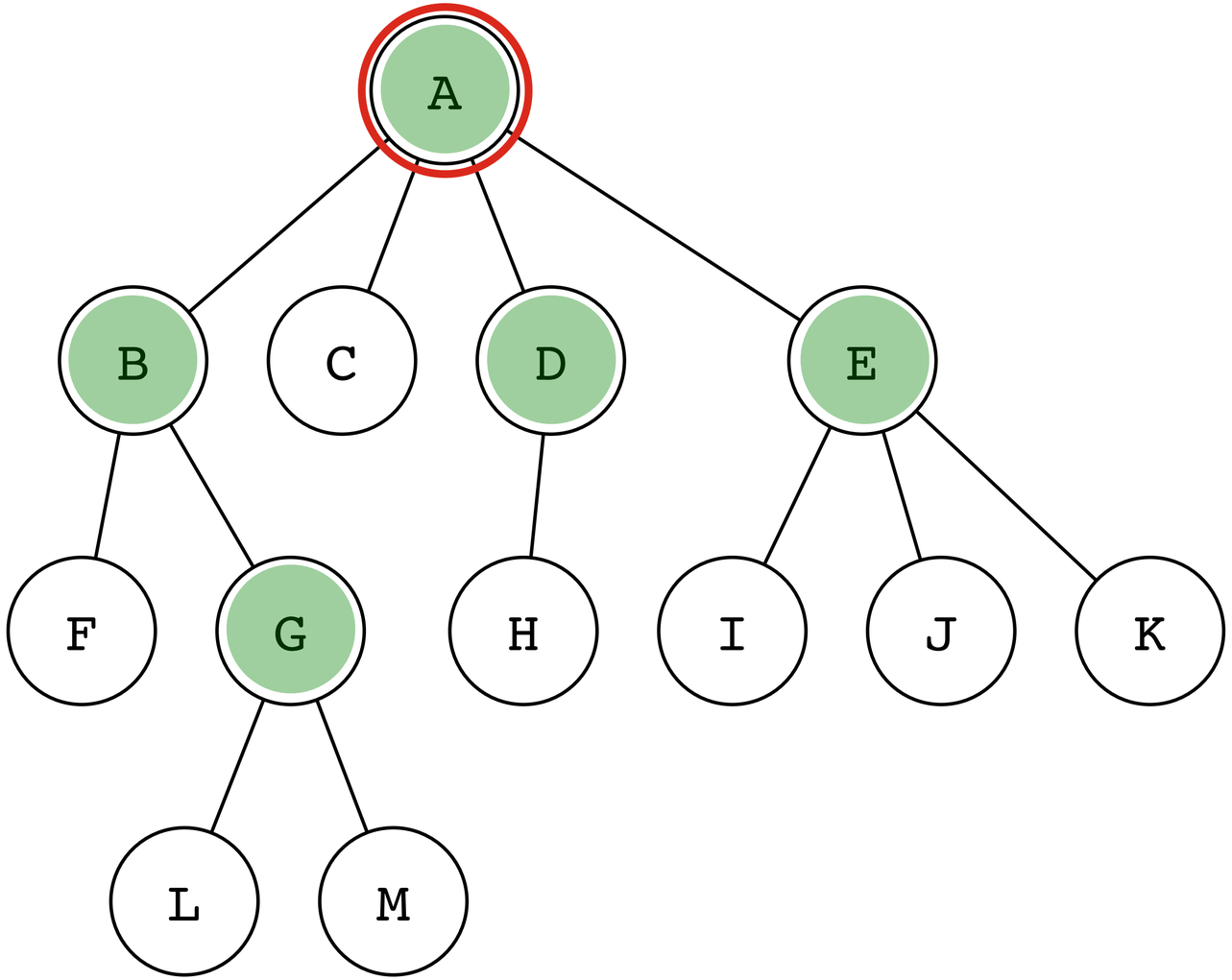
Nomenclature des noeuds

- Noeuds internes : ont un ou plusieurs enfants
- Feuilles : n'ont pas d'enfant



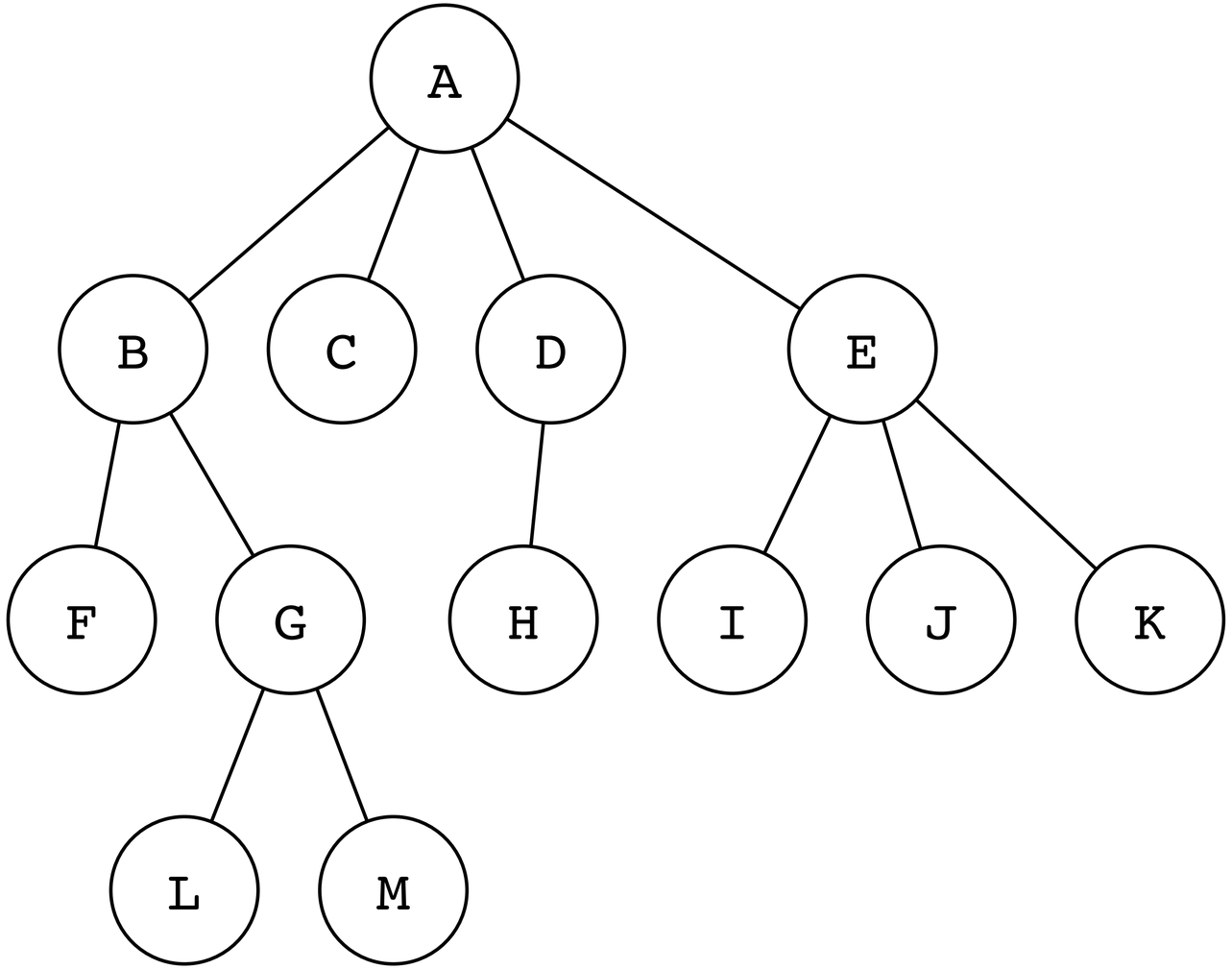
Nomenclature des noeuds

- Noeuds internes : ont un ou plusieurs enfants
- Feuilles : n'ont pas d'enfant
- Racine : n'a pas de parent



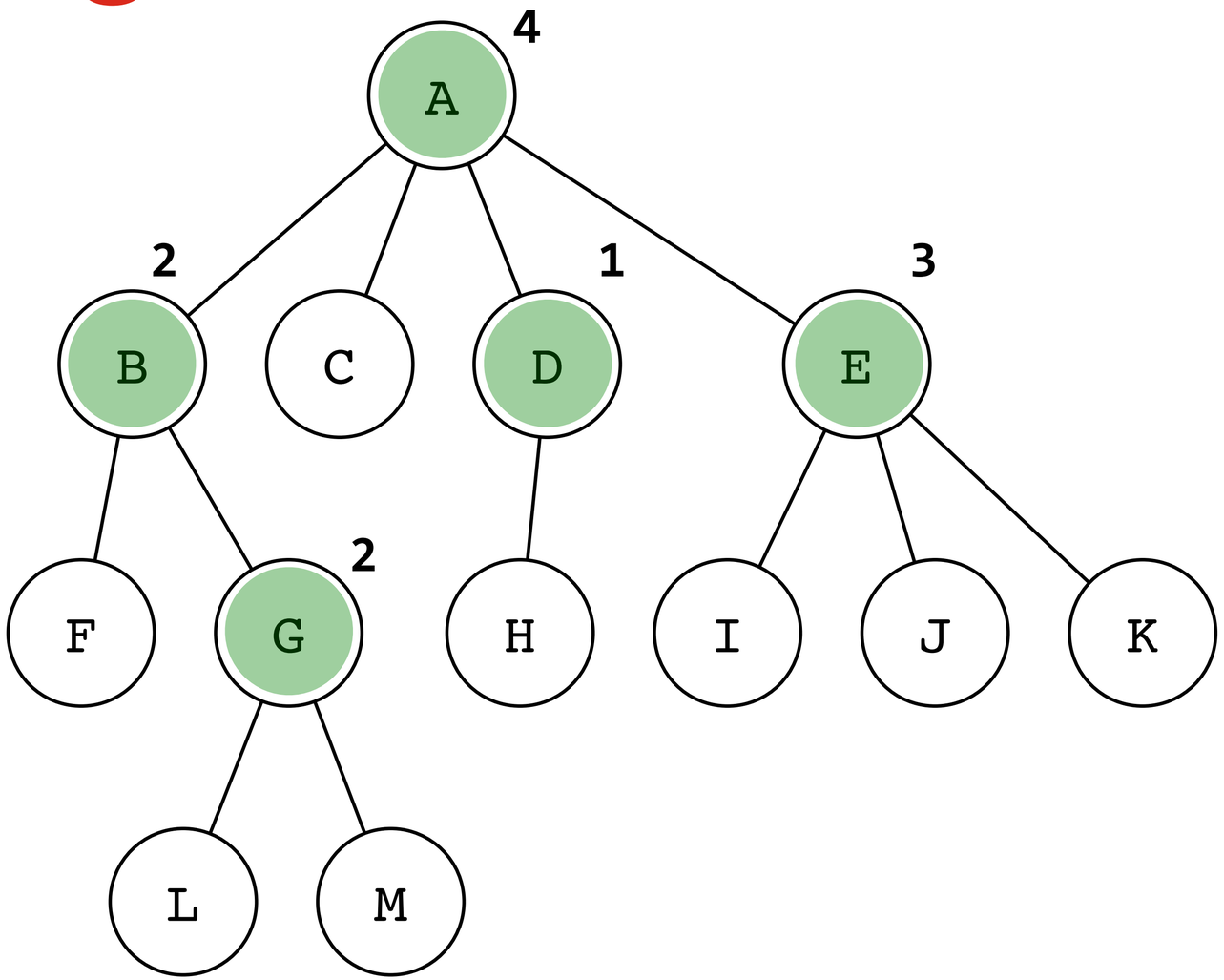
Degrés

- Degré d'un noeud : nombre d'enfants



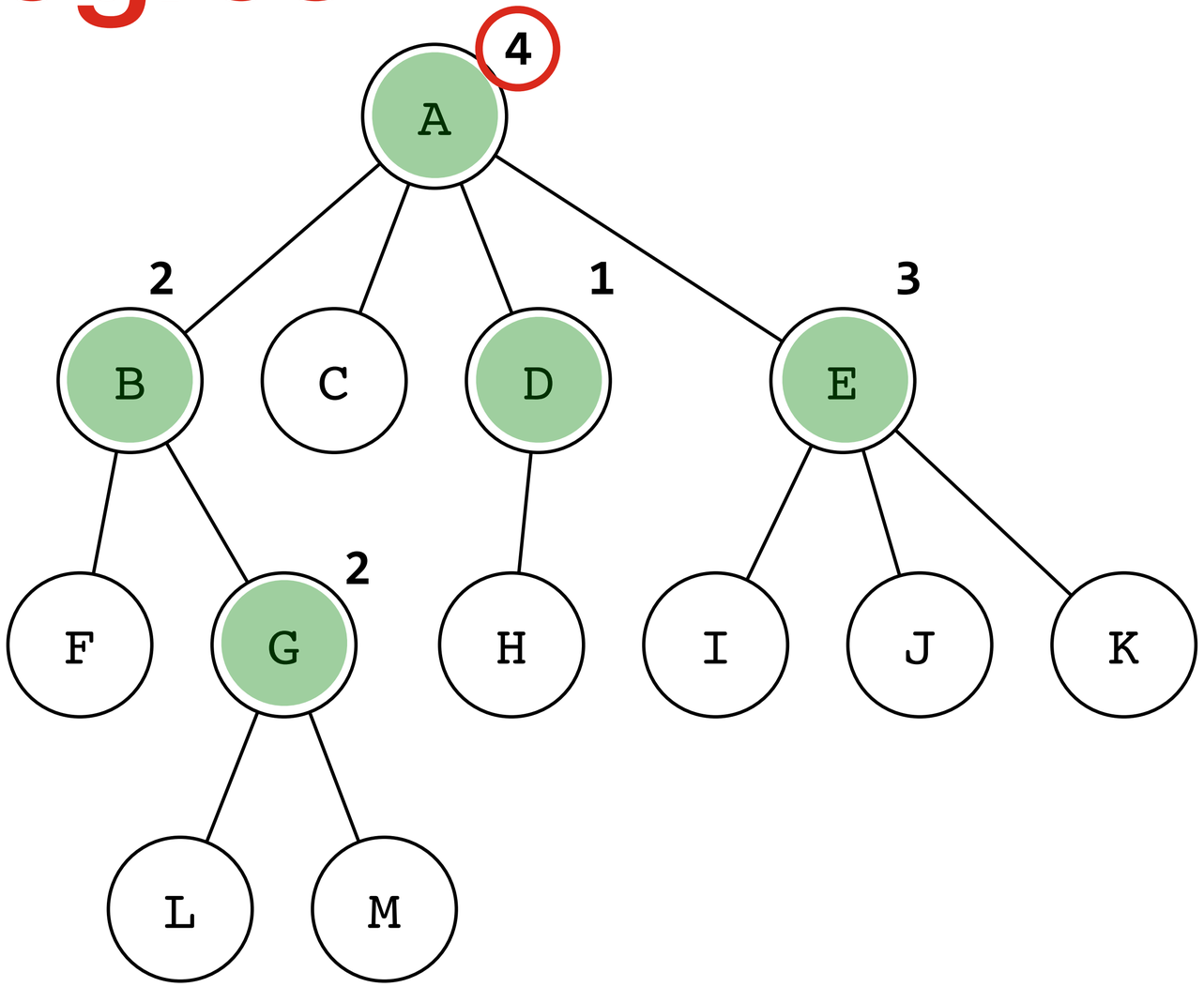
Degrés

- Degré d'un noeud : nombre d'enfants



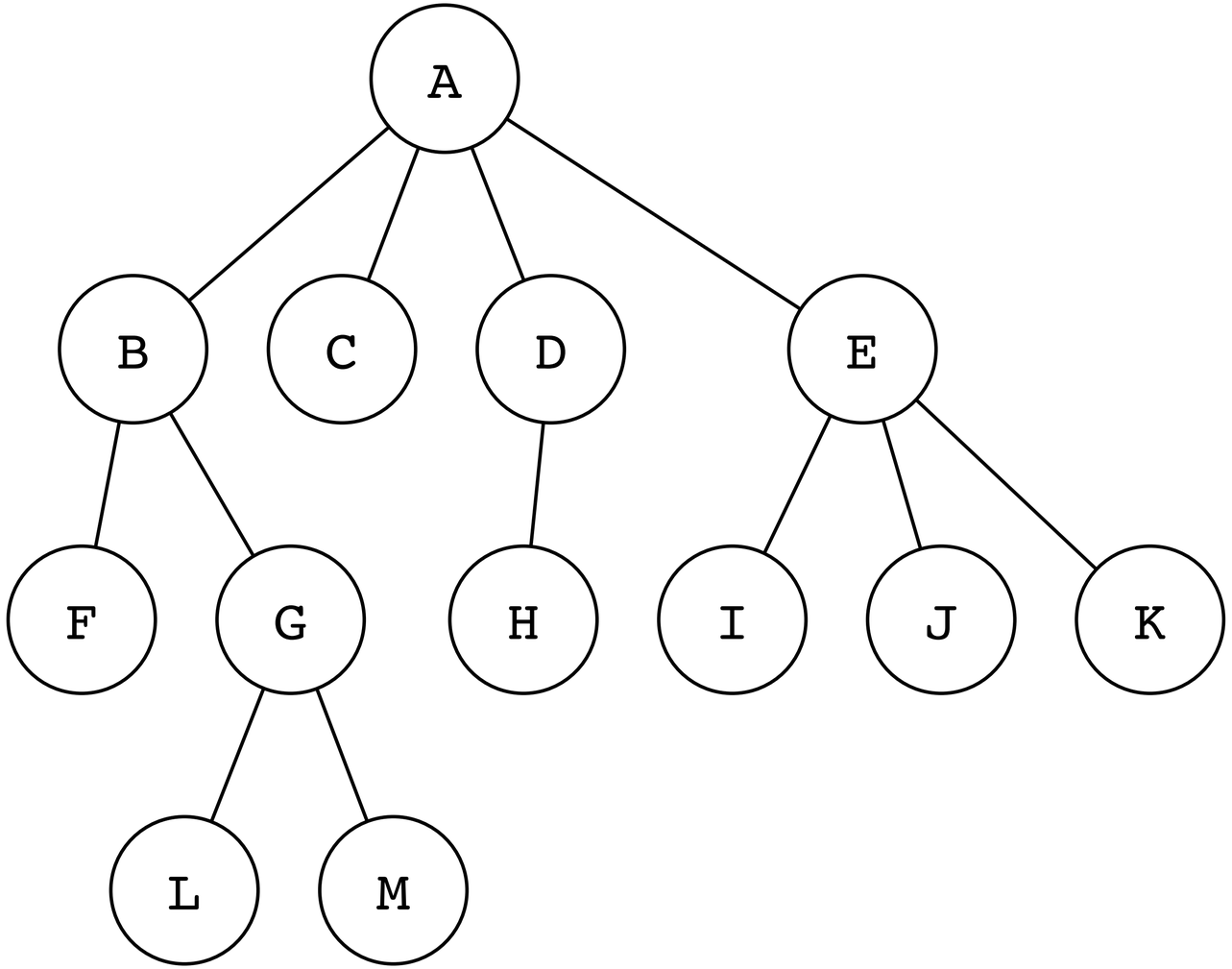
Degrés

- Degré d'un noeud : nombre d'enfants
- Degré d'un arbre : degré max de ses noeuds



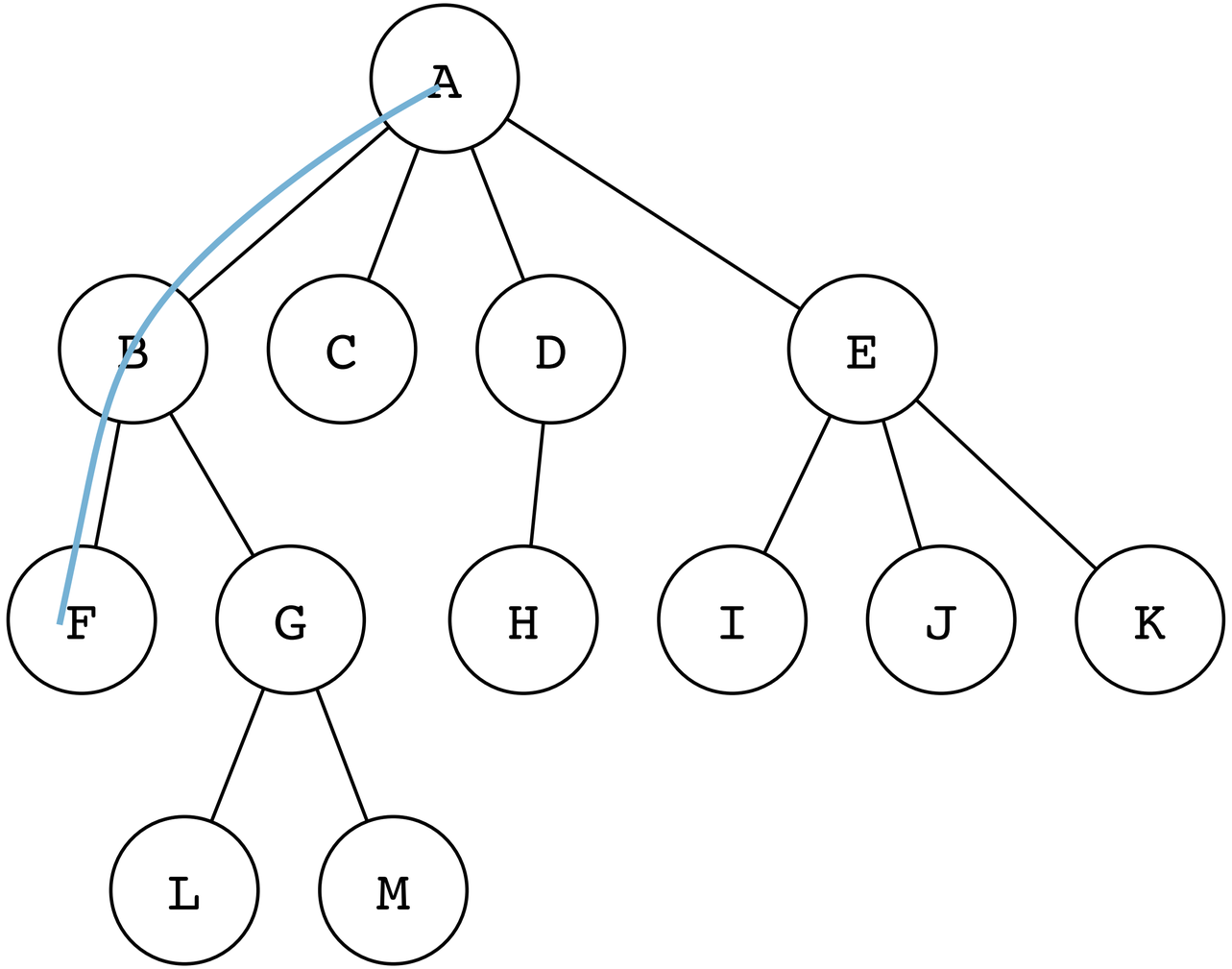
Chemins

- Chemin d'un noeud : suite des noeuds reliant la racine au noeud



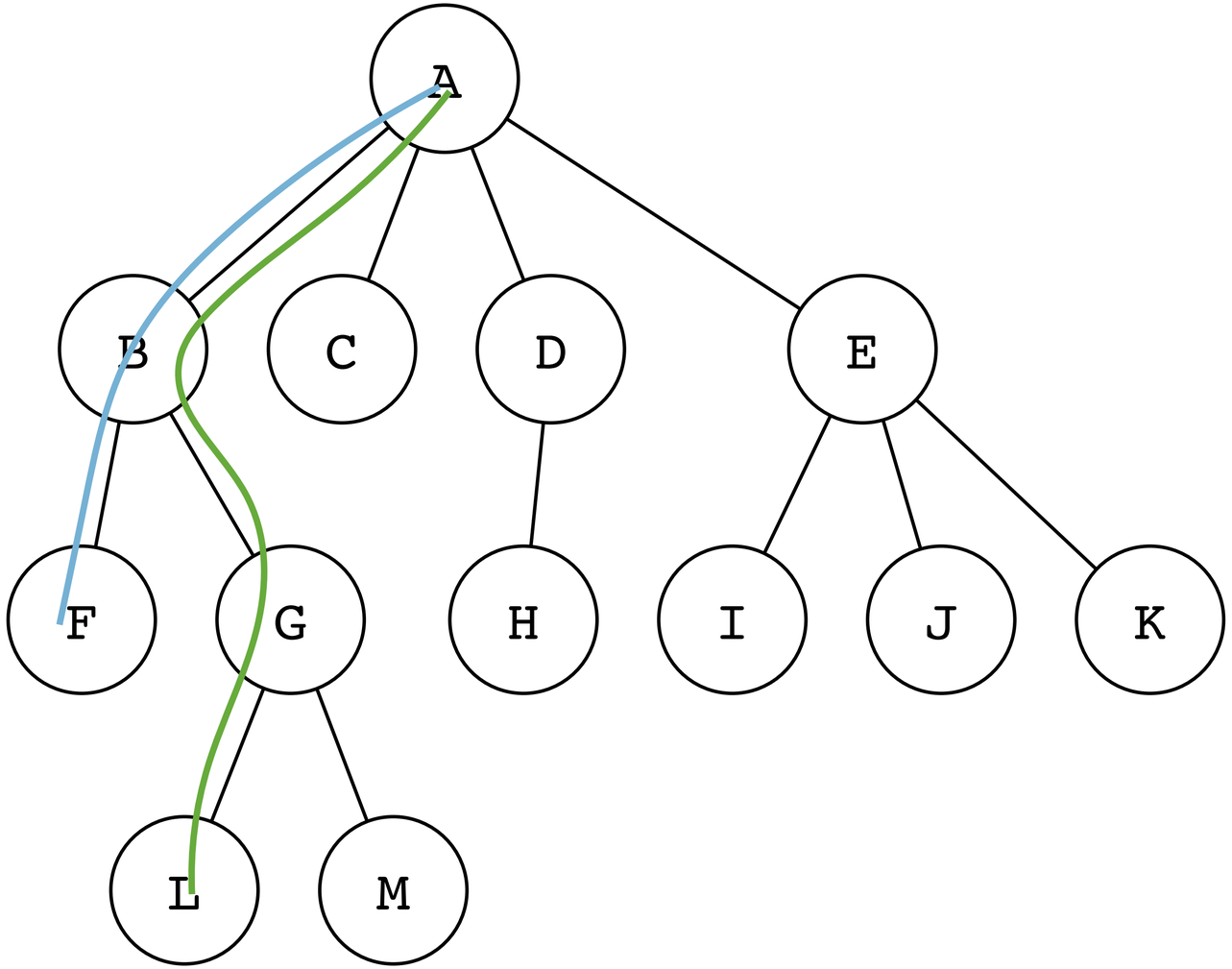
Chemins

- Chemin d'un noeud : suite des noeuds reliant la racine au noeud



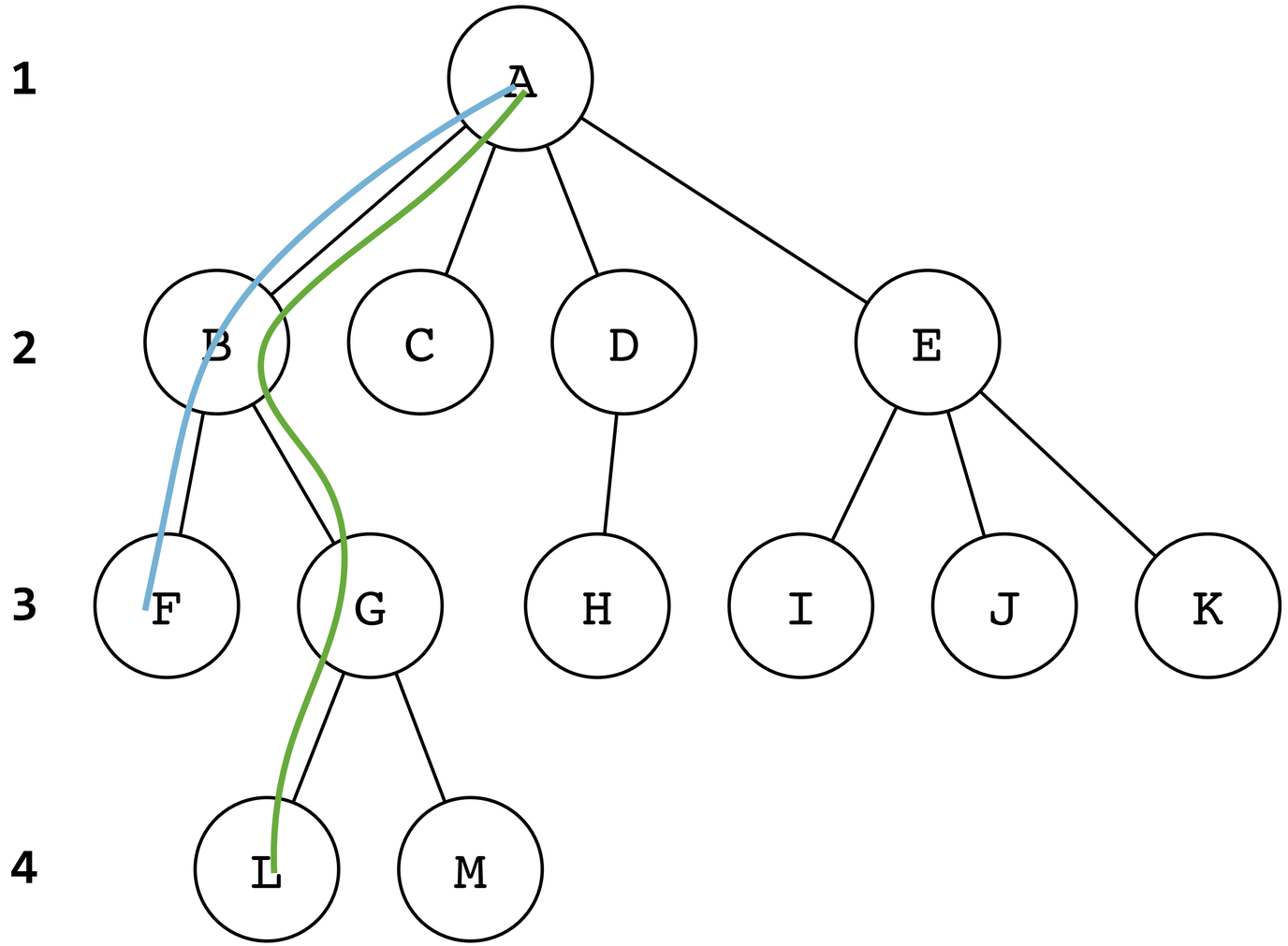
Chemins

- Chemin d'un noeud : suite des noeuds reliant la racine au noeud



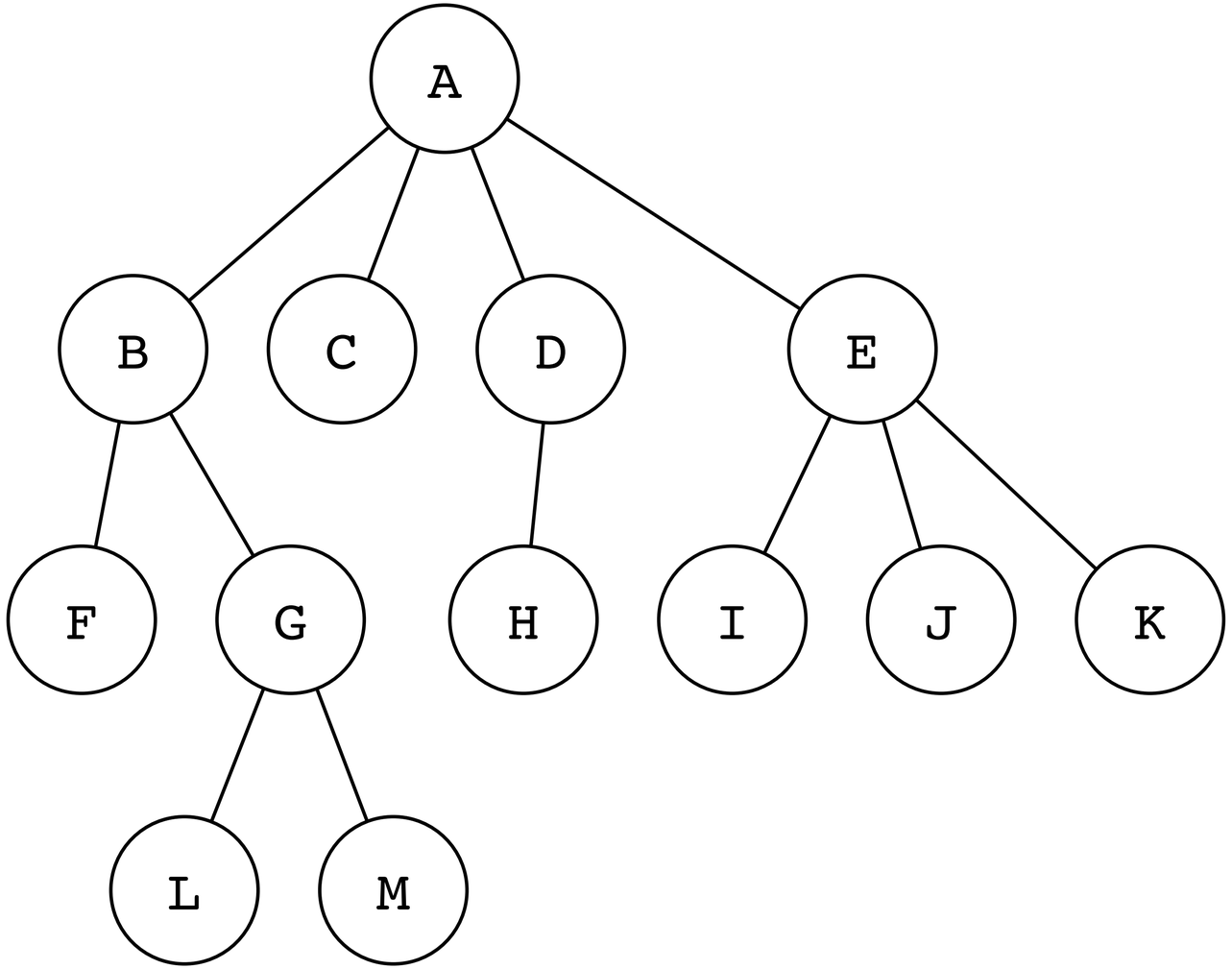
Chemins

- Chemin d'un noeud : suite des noeuds reliant la racine au noeud
- Niveau d'un noeud: longueur de son chemin



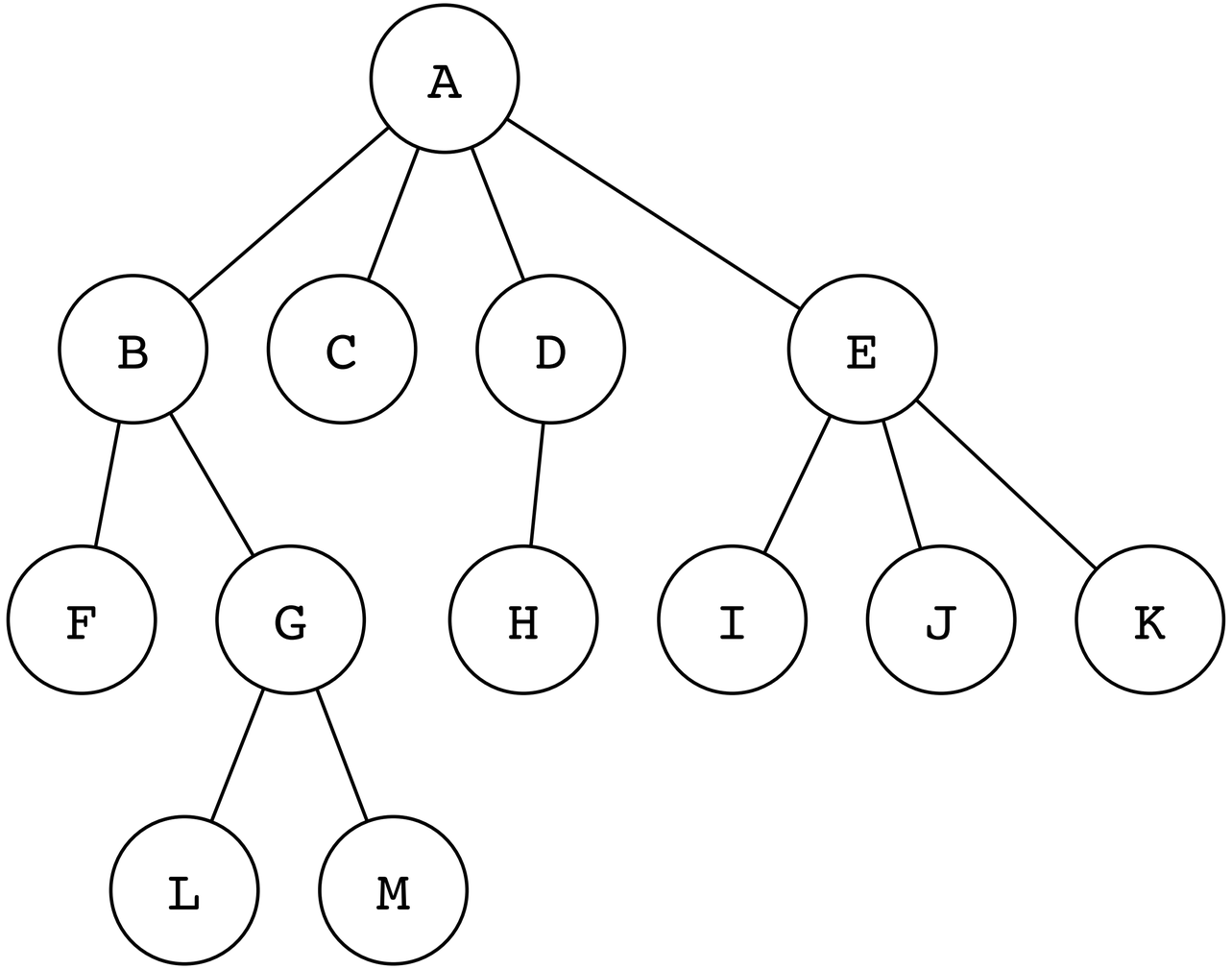
Hauteur et sous-arbres

- Hauteur d'un arbre: niveau maximum parmi ses noeuds



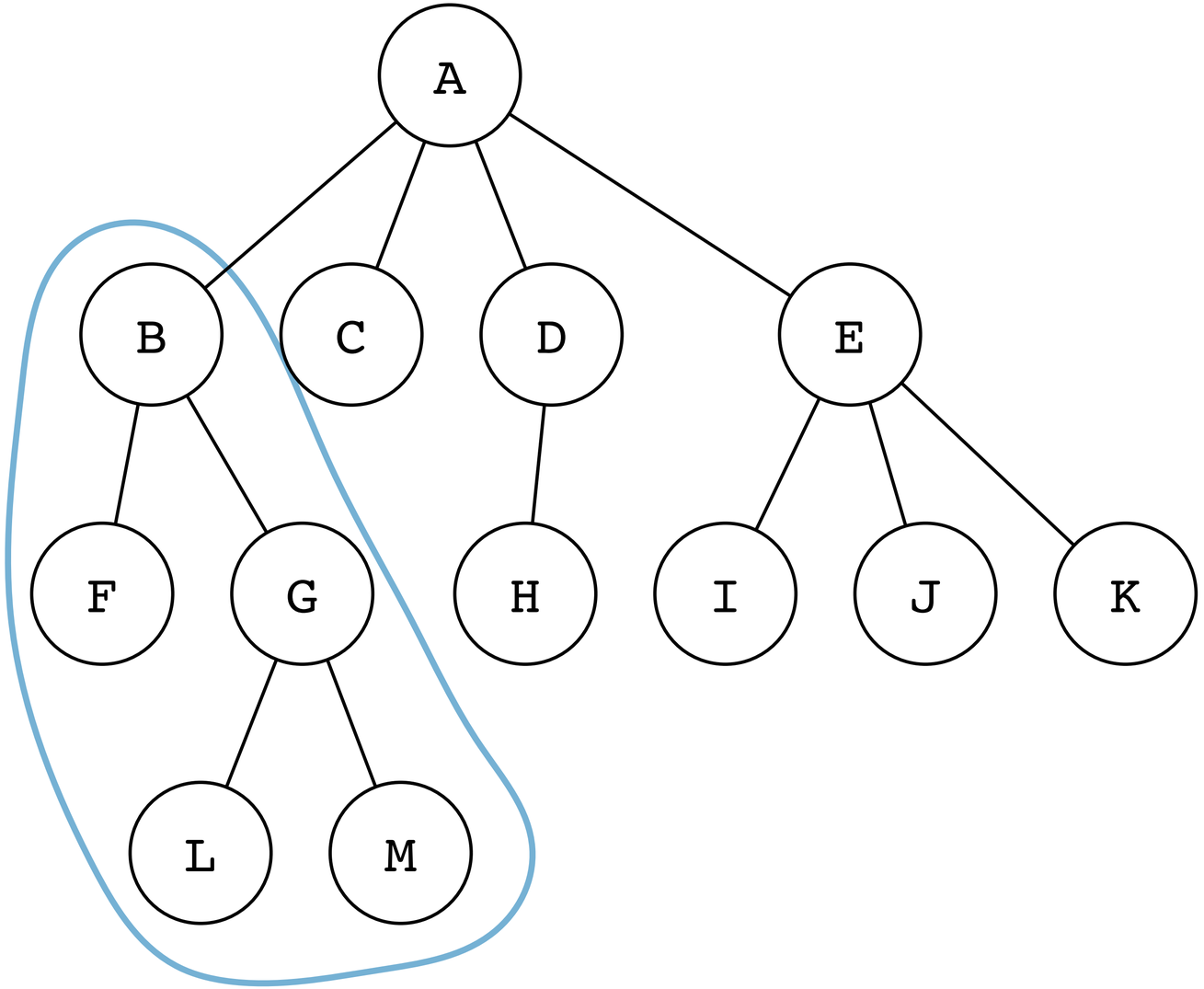
Hauteur et sous-arbres

- Hauteur d'un arbre: niveau maximum parmi ses noeuds
- Sous-arbre : arbre dont un des noeuds de l'arbre est la racine



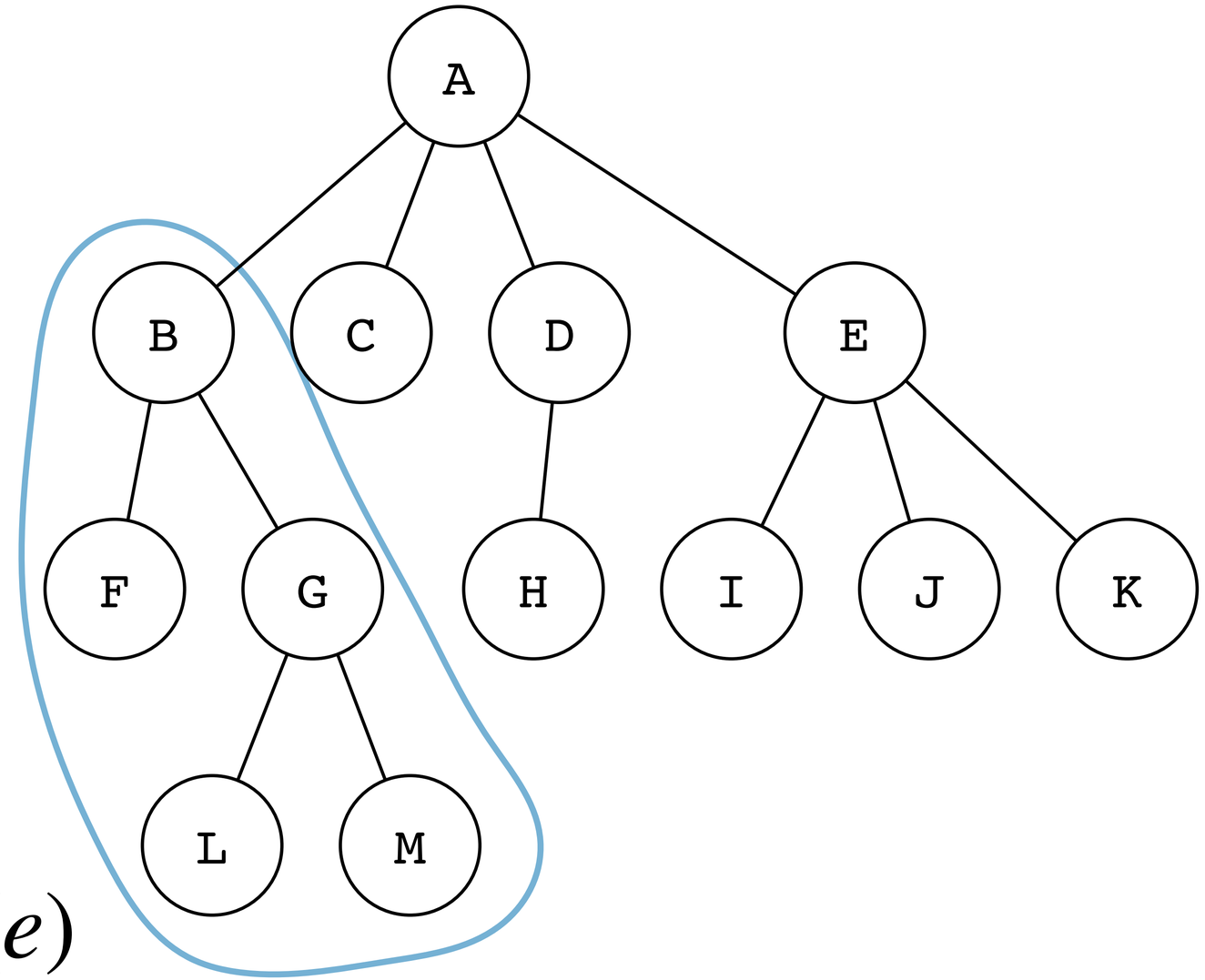
Hauteur et sous-arbres

- Hauteur d'un arbre: niveau maximum parmi ses noeuds
- Sous-arbre : arbre dont un des noeuds de l'arbre est la racine



Hauteur et sous-arbres

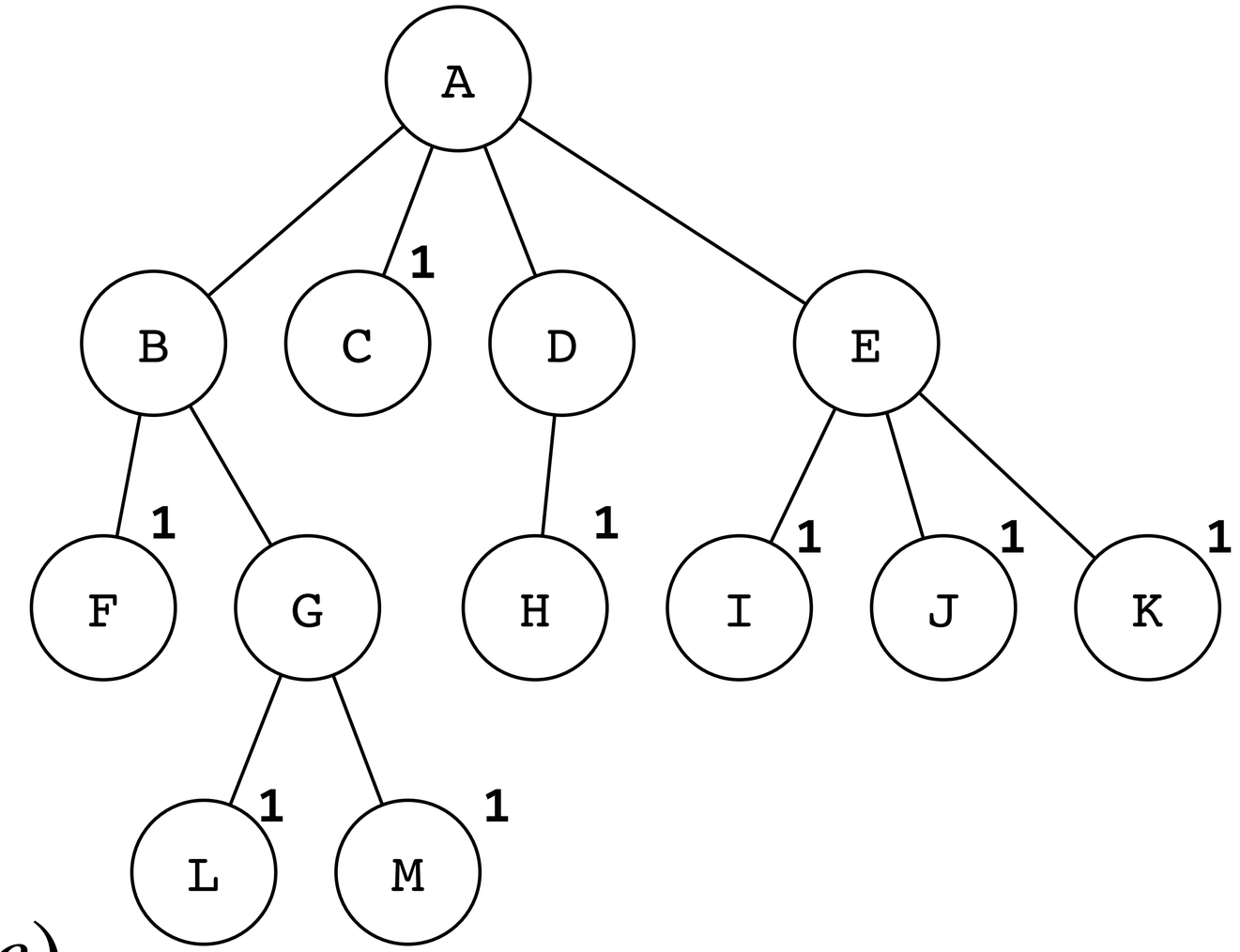
- Hauteur d'un arbre: niveau maximum parmi ses noeuds
- Sous-arbre : arbre dont un des noeuds de l'arbre est la racine



$$H(n) = 1 + \max_{e \in enfants(n))} H(e)$$

Hauteur et sous-arbres

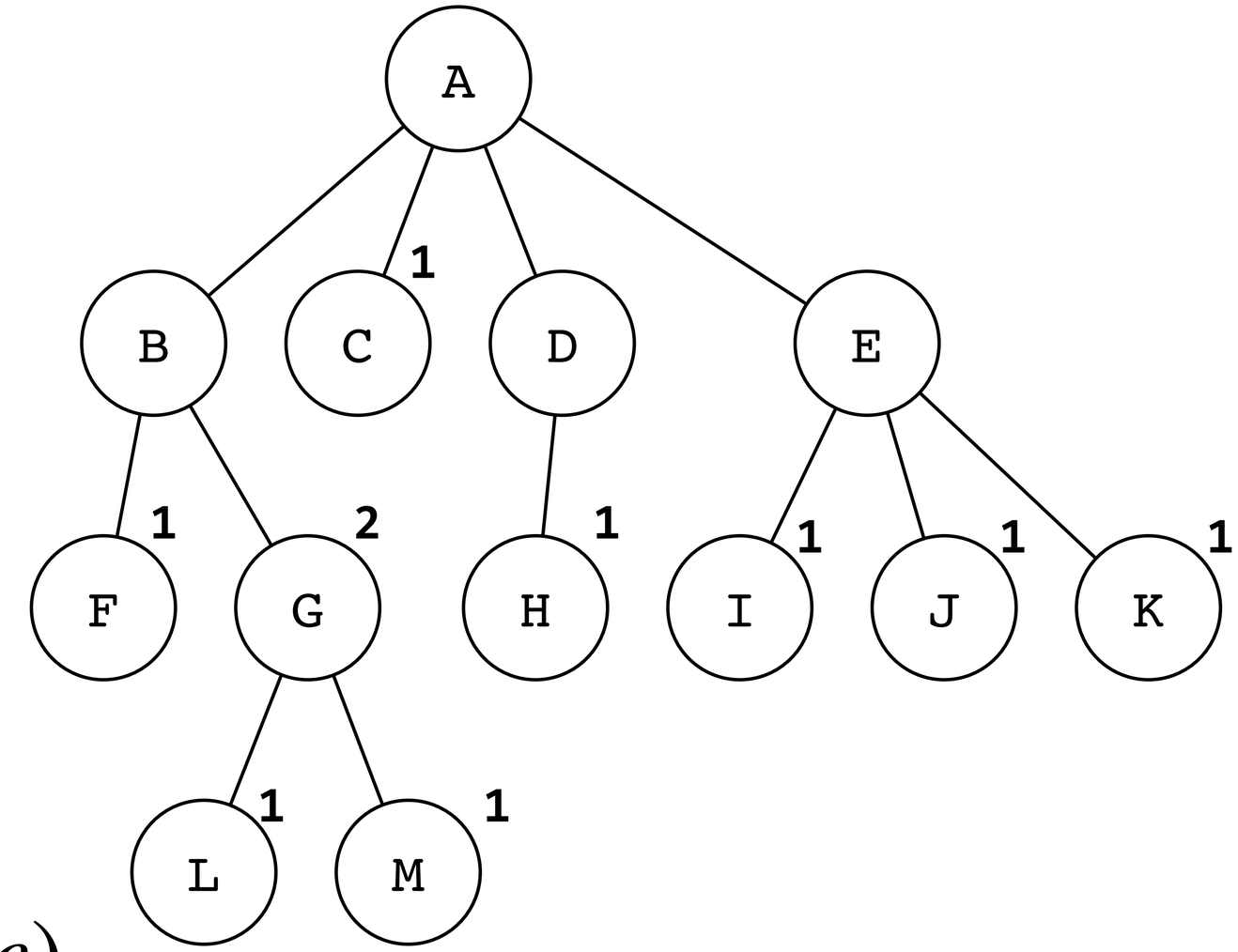
- Hauteur d'un arbre: niveau maximum parmi ses noeuds
- Sous-arbre : arbre dont un des noeuds de l'arbre est la racine



$$H(n) = 1 + \max_{e \in enfants(n))} H(e)$$

Hauteur et sous-arbres

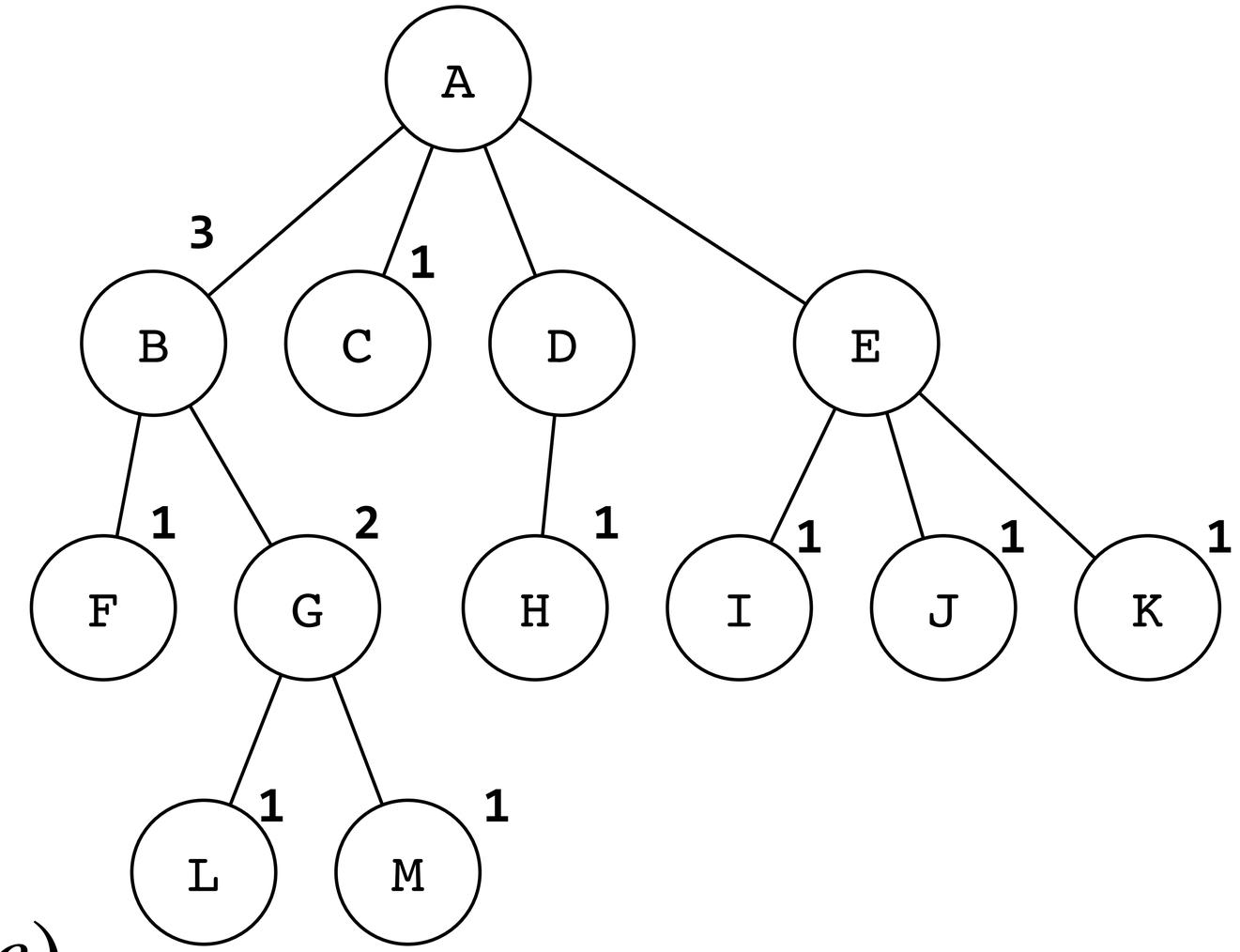
- Hauteur d'un arbre: niveau maximum parmi ses noeuds
- Sous-arbre : arbre dont un des noeuds de l'arbre est la racine



$$H(n) = 1 + \max_{e \in enfants(n))} H(e)$$

Hauteur et sous-arbres

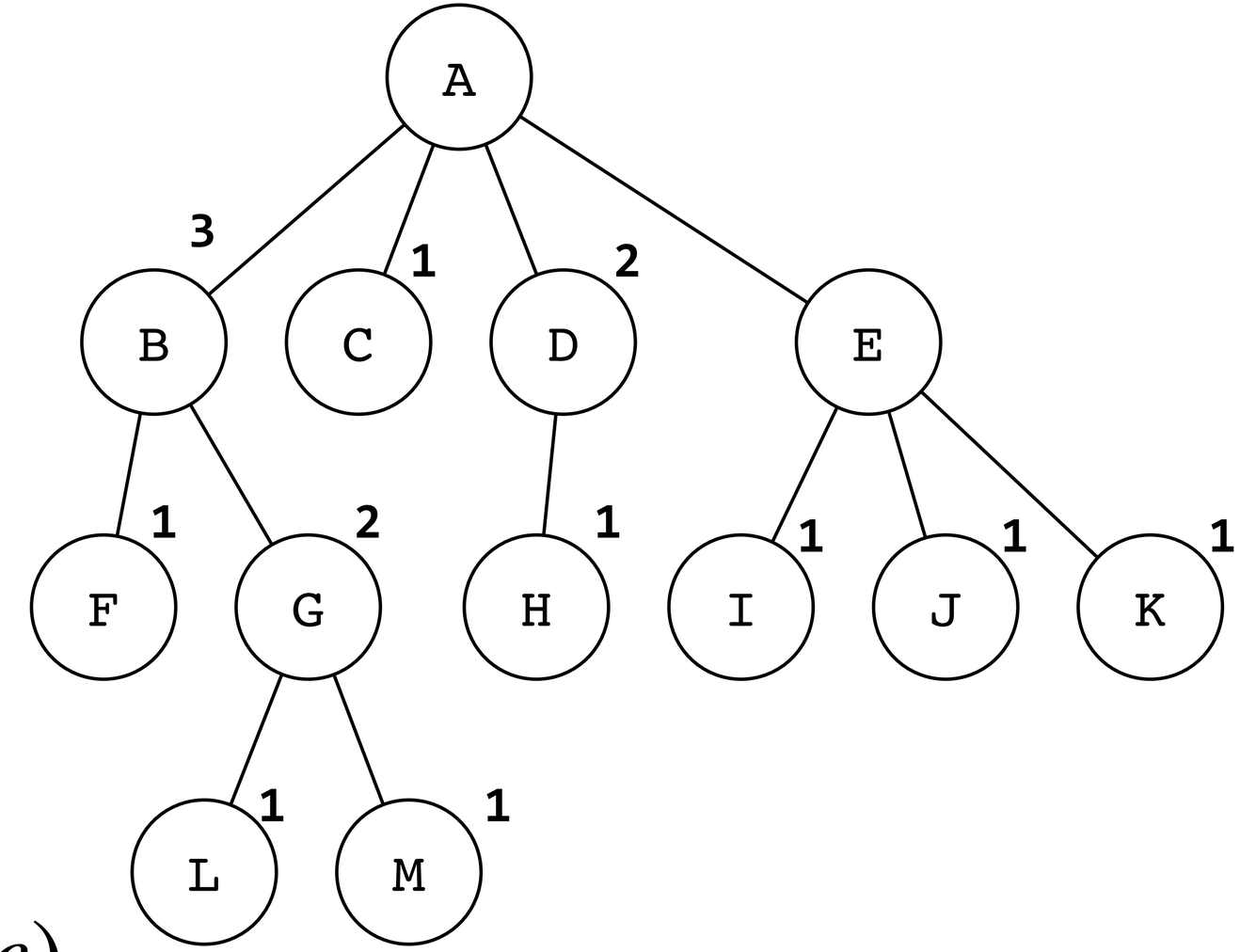
- Hauteur d'un arbre: niveau maximum parmi ses noeuds
- Sous-arbre : arbre dont un des noeuds de l'arbre est la racine



$$H(n) = 1 + \max_{e \in enfants(n))} H(e)$$

Hauteur et sous-arbres

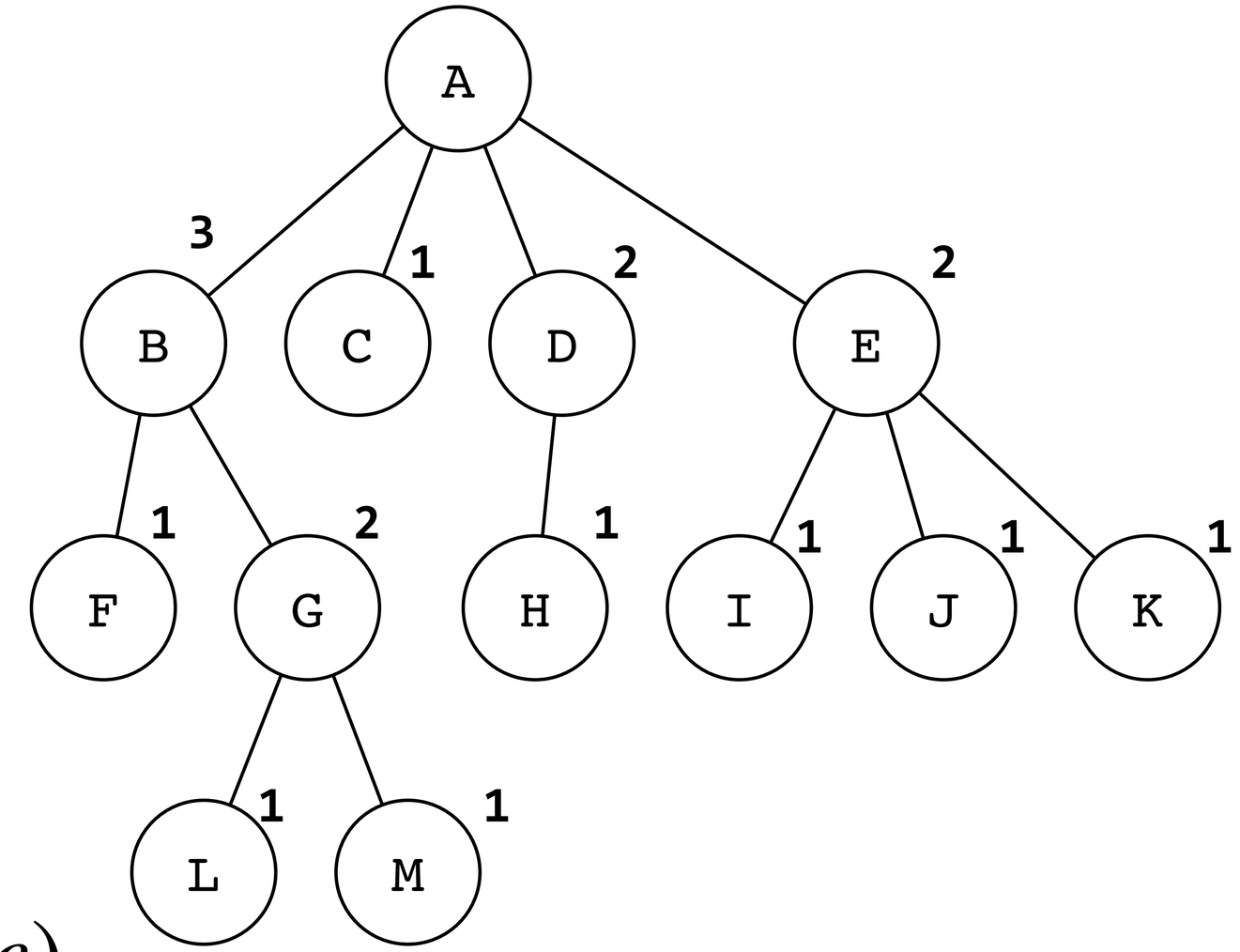
- Hauteur d'un arbre: niveau maximum parmi ses noeuds
- Sous-arbre : arbre dont un des noeuds de l'arbre est la racine



$$H(n) = 1 + \max_{e \in enfants(n))} H(e)$$

Hauteur et sous-arbres

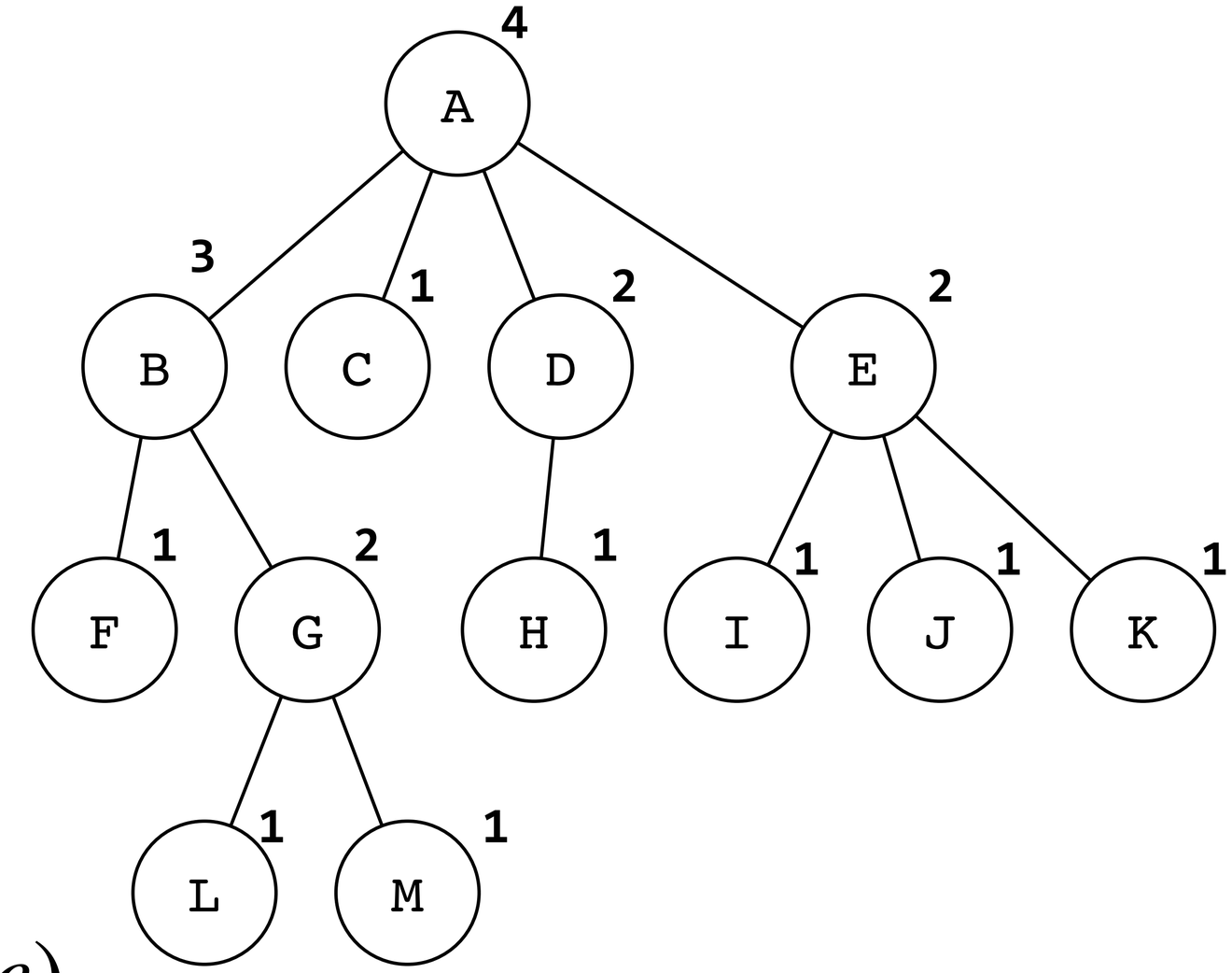
- Hauteur d'un arbre: niveau maximum parmi ses noeuds
- Sous-arbre : arbre dont un des noeuds de l'arbre est la racine



$$H(n) = 1 + \max_{e \in enfants(n))} H(e)$$

Hauteur et sous-arbres

- Hauteur d'un arbre: niveau maximum parmi ses noeuds
- Sous-arbre : arbre dont un des noeuds de l'arbre est la racine



$$H(n) = 1 + \max_{e \in enfants(n))} H(e)$$

Arbre vide

Arbre vide

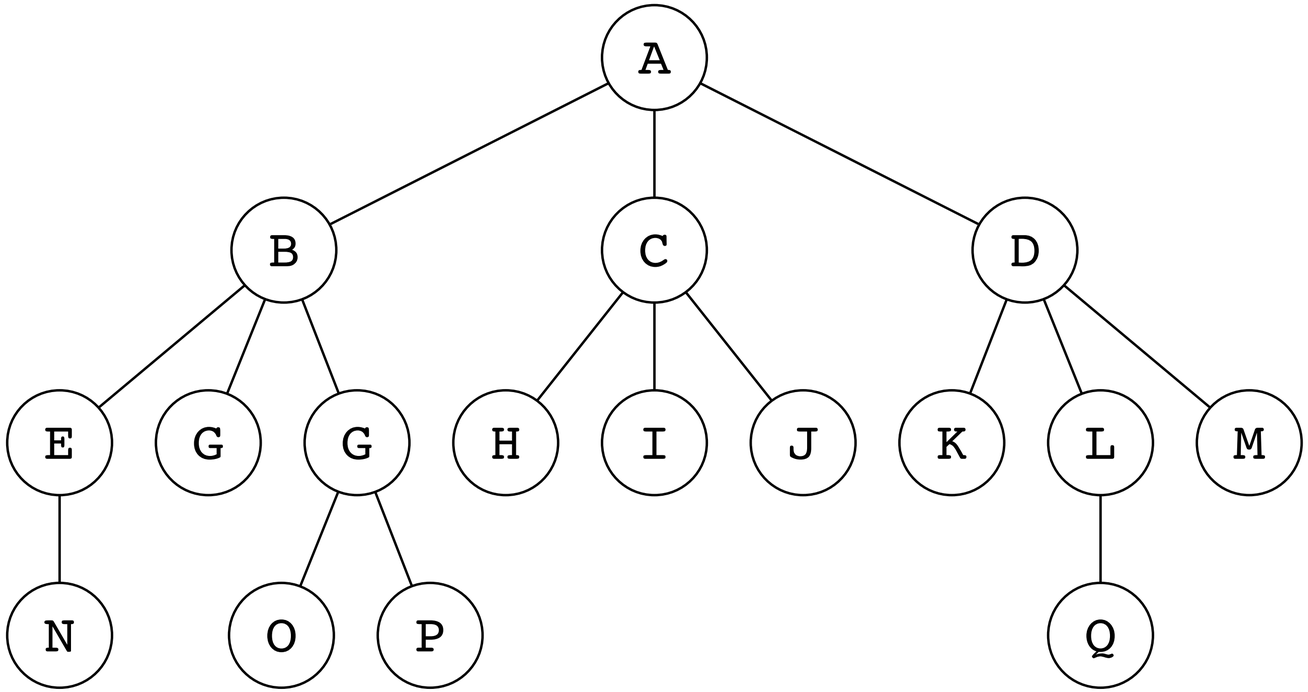
- Arbre sans aucun noeud

Arbre vide

- Arbre sans aucun noeud
- A une hauteur 0

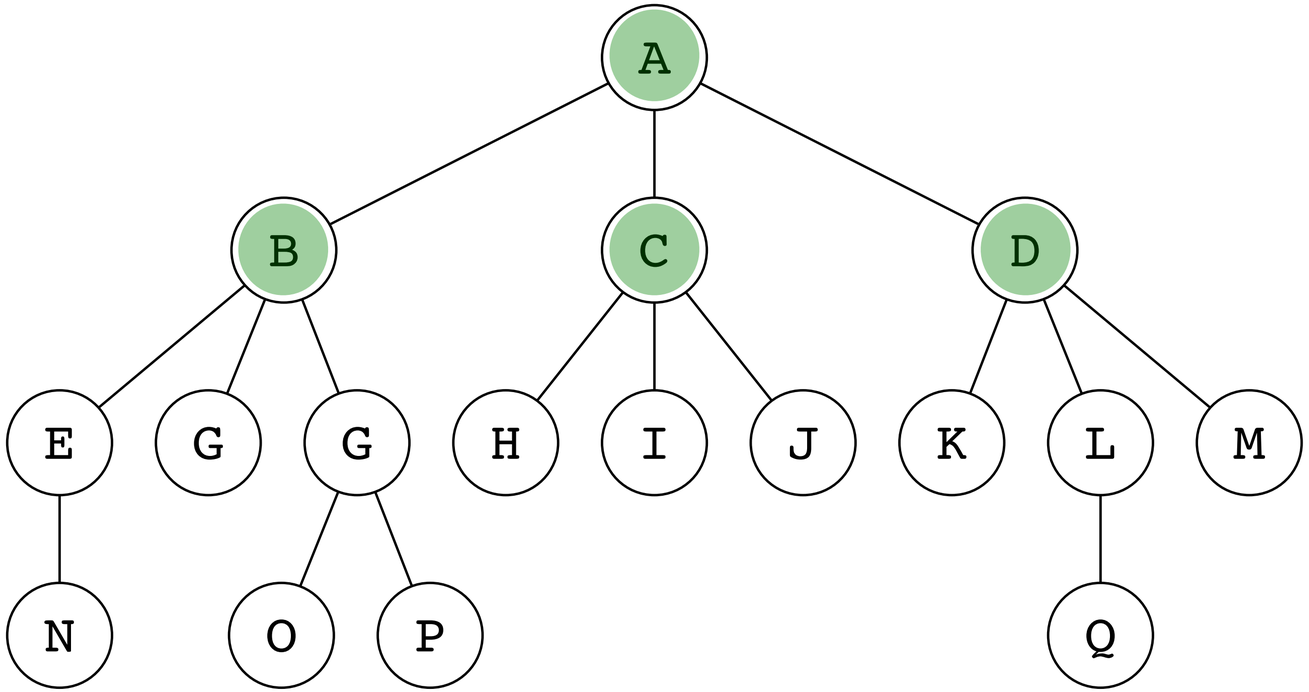
Arbre plein

- De hauteur h (ici 4)
- De degré d (ici 3)



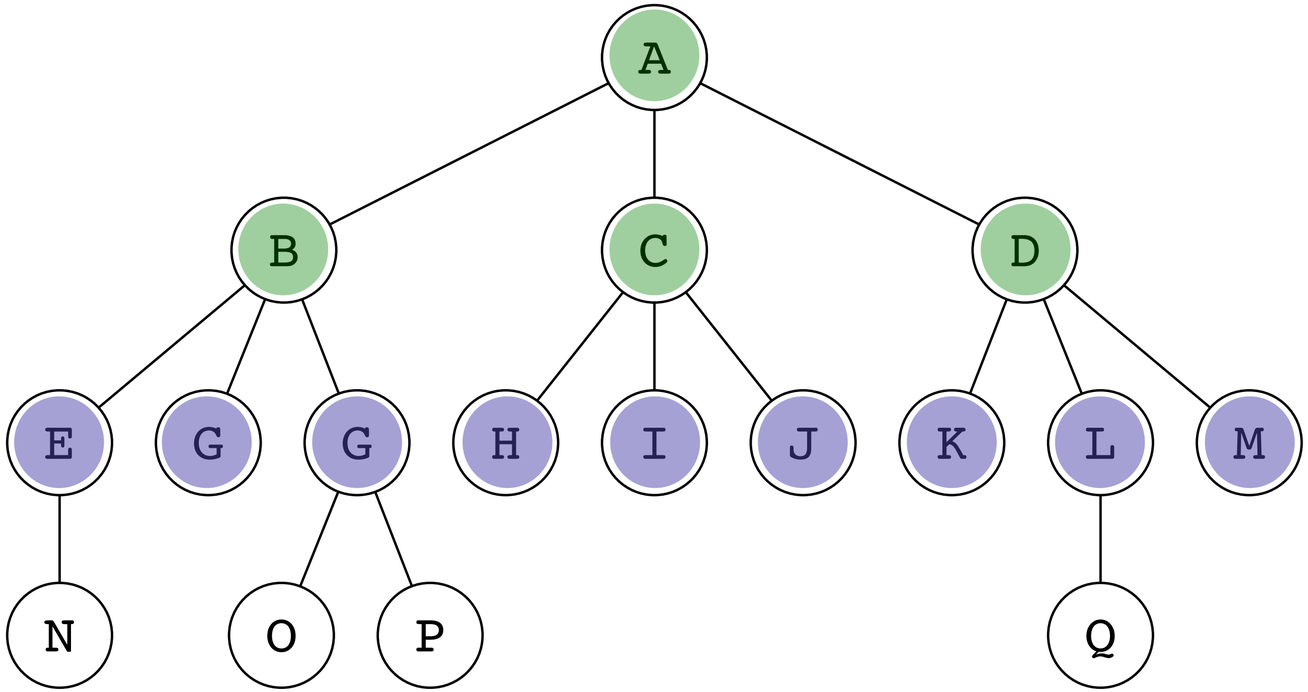
Arbre plein

- De hauteur h (ici 4)
- De degré d (ici 3)
- Tous les noeuds de niveau inférieur à h-1 sont de degré d



Arbre plein

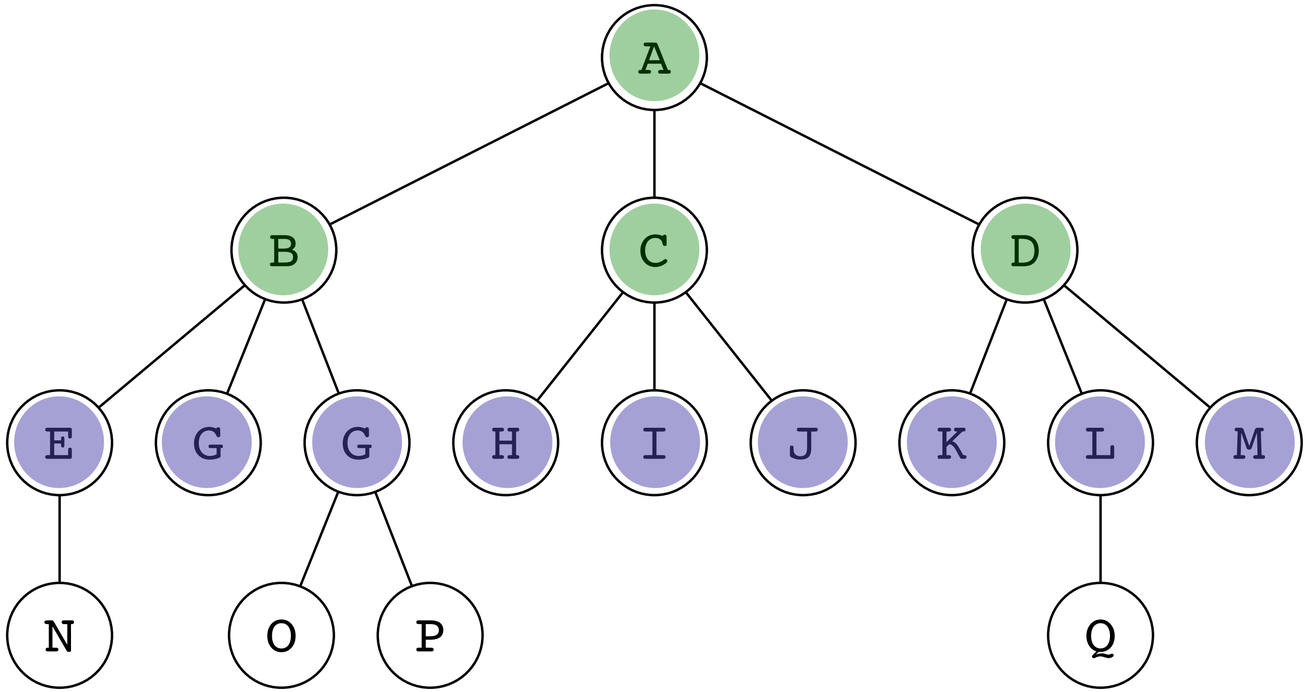
- De hauteur h (ici 4)
- De degré d (ici 3)
- Tous les noeuds de niveau inférieur à h-1 sont de degré d



- Les noeuds de niveau h-1 ont un degré quelconque

Arbre plein

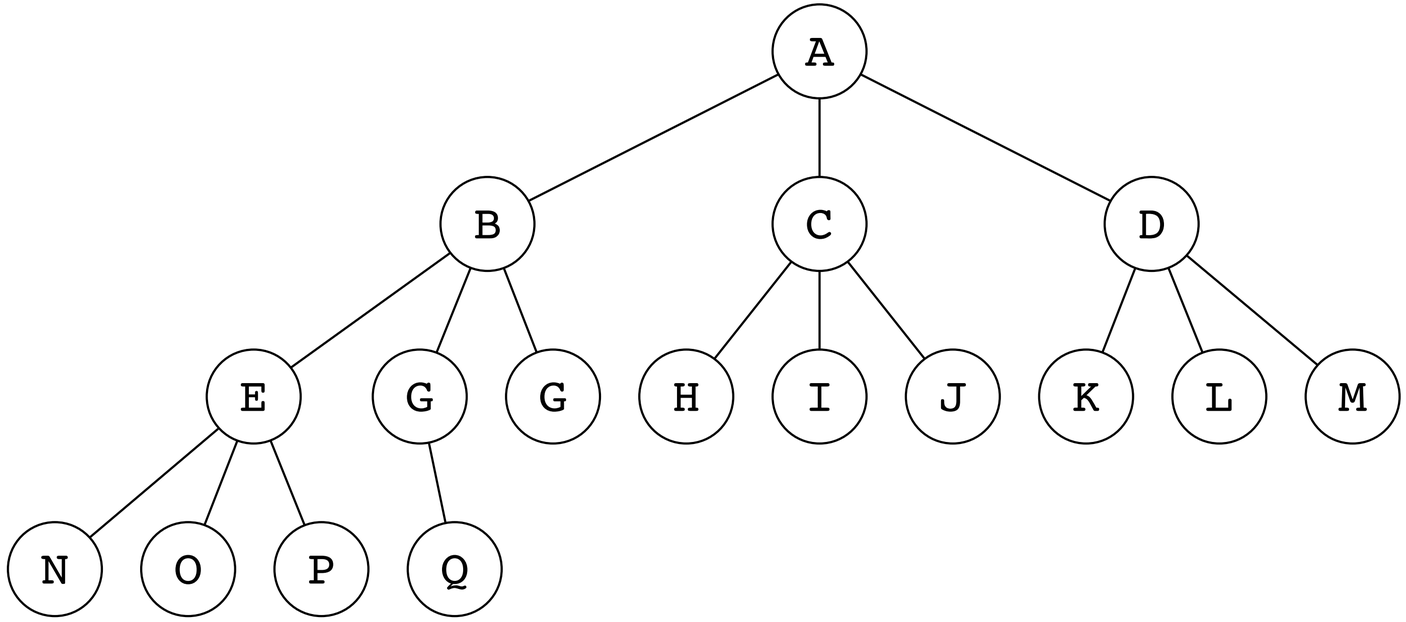
- De hauteur h (ici 4)
- De degré d (ici 3)
- Tous les noeuds de niveau inférieur à h-1 sont de degré d



- Les noeuds de niveau h-1 ont un degré quelconque
- Les noeuds de niveau h sont des feuilles

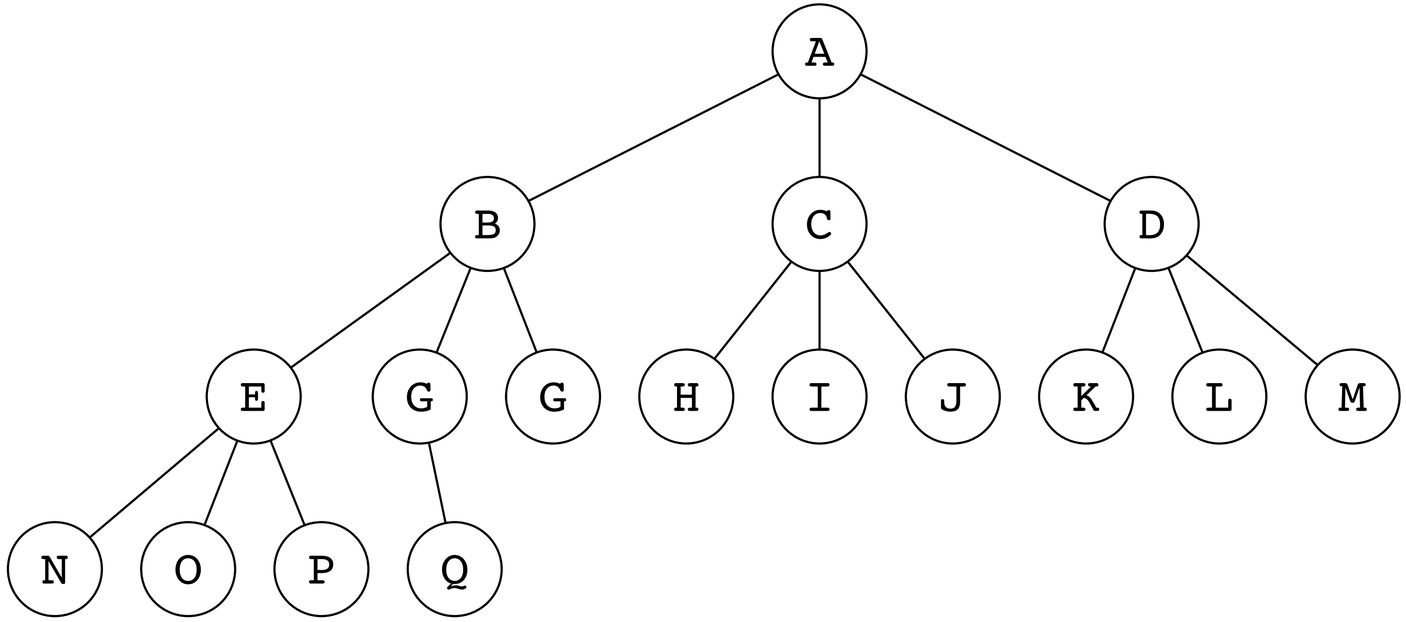
Arbre complet

- Arbre plein dont le dernier niveau est rempli par la gauche



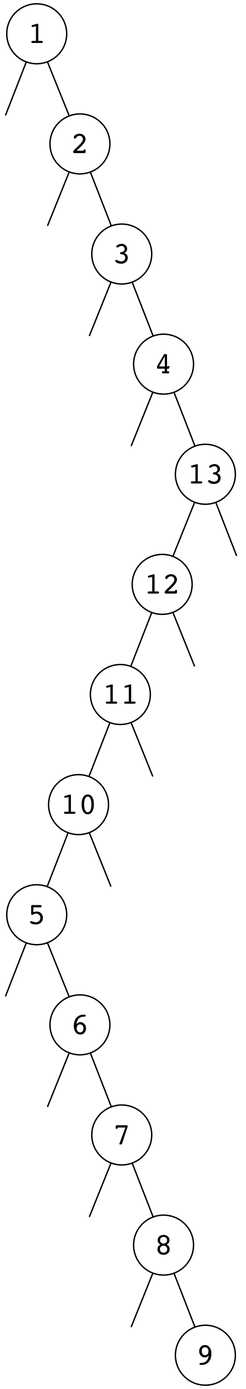
Arbre complet

- Arbre plein dont le dernier niveau est rempli par la gauche
- Au chapitre 4, un tas était un arbre binaire complet



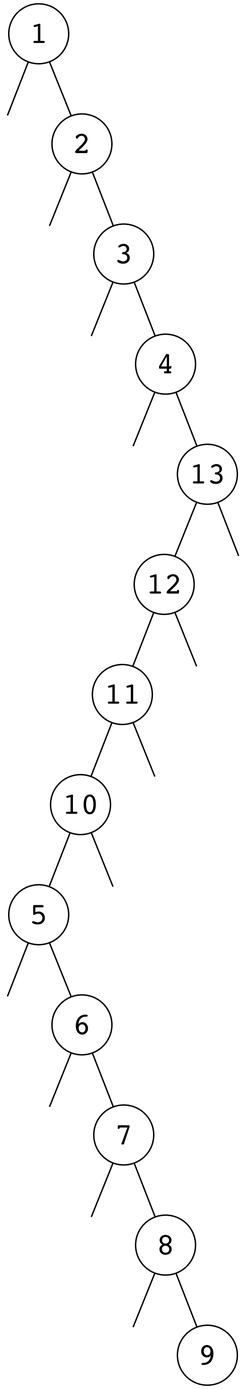
Arbre dégénéré

- Arbre de degré 1



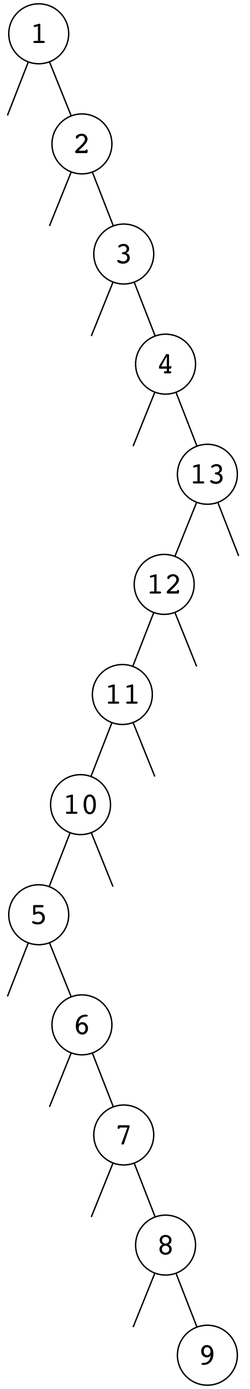
Arbre dégénéré

- Arbre de degré 1
- Topologiquement équivalent à une liste chainée



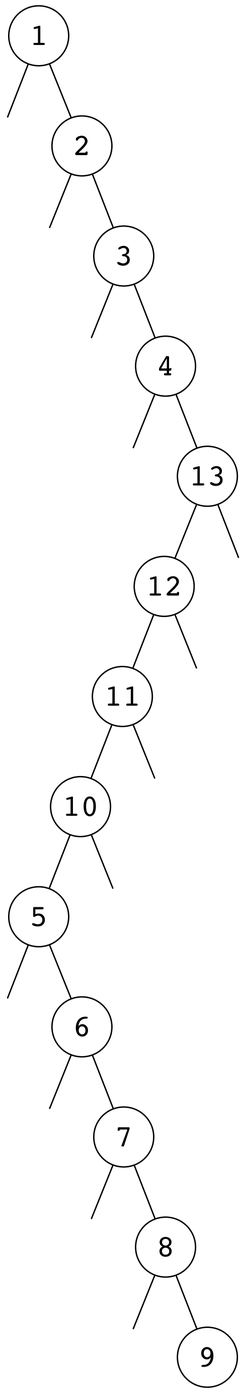
Arbre dégénéré

- Arbre de degré 1
- Topologiquement équivalent à une liste chainée
- Typiquement le pire cas pour les algorithmes dont la complexité dépend de la hauteur de l'arbre



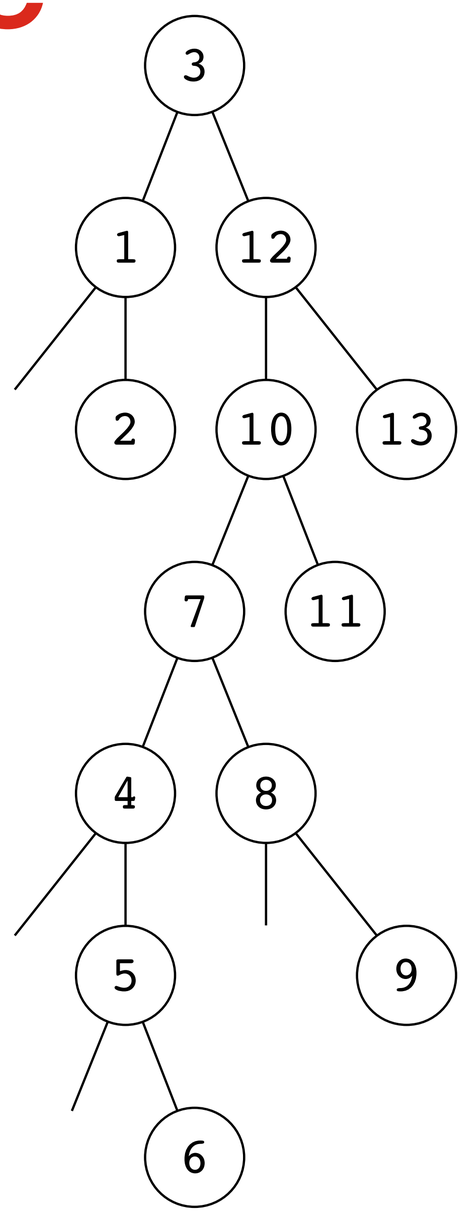
Arbre dégénéré

- Arbre de degré 1
- Topologiquement équivalent à une liste chainée
- Typiquement le pire cas pour les algorithmes dont la complexité dépend de la hauteur de l'arbre
- Parfois utile comme structure intermédiaire (équilibrage par linéarisation / arborisation)



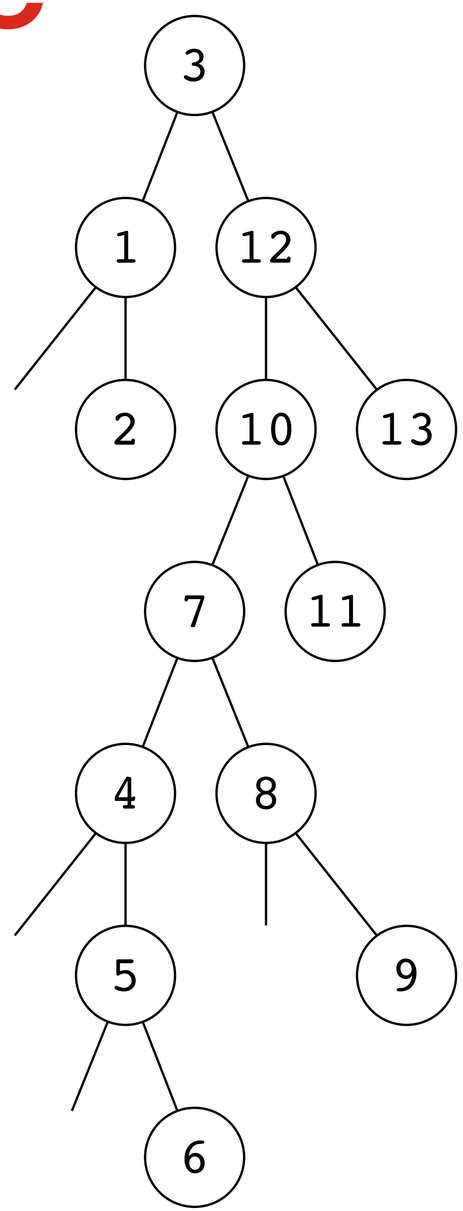
Arbre binaire

- Arbre de degré 2



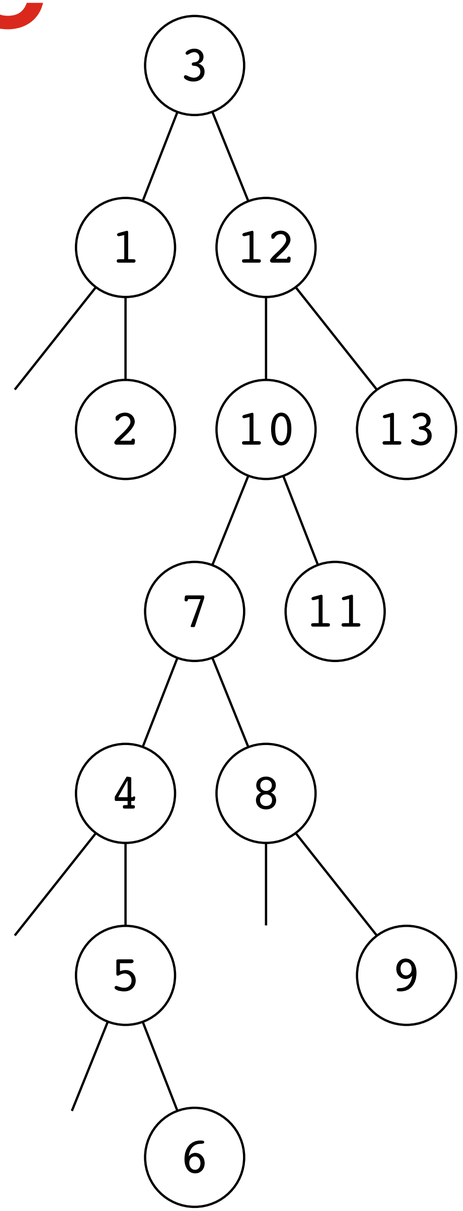
Arbre binaire

- Arbre de degré 2
- … ou plutôt ≤ 2

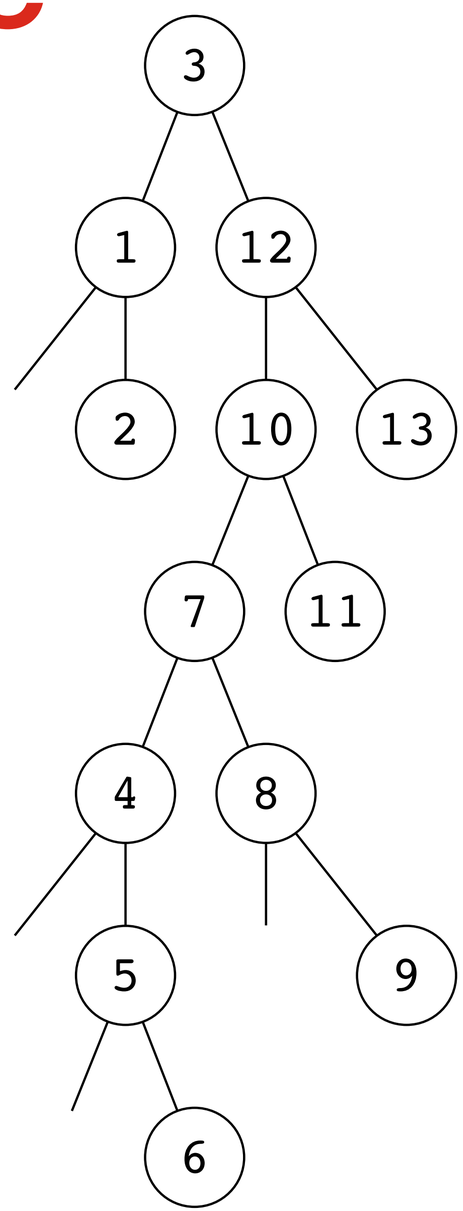


Arbre binaire

- Arbre de degré 2
- … ou plutôt ≤ 2
- Pour les noeuds de degré 1, peut distinguer si l'enfant est à gauche ou à droite



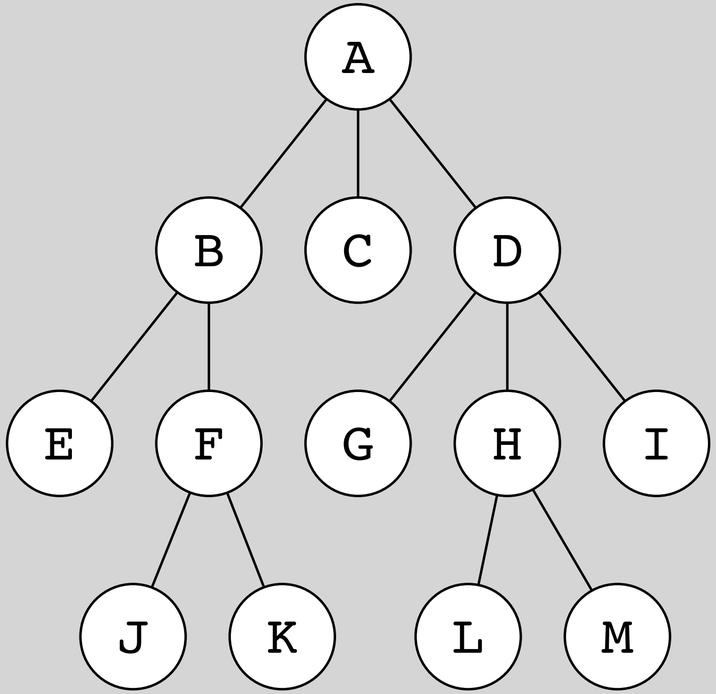
Arbre binaire



- Arbre de degré 2
- … ou plutôt ≤ 2
- Pour les noeuds de degré 1, peut distinguer si l'enfant est à gauche ou à droite
- Etudié spécifiquement par la suite

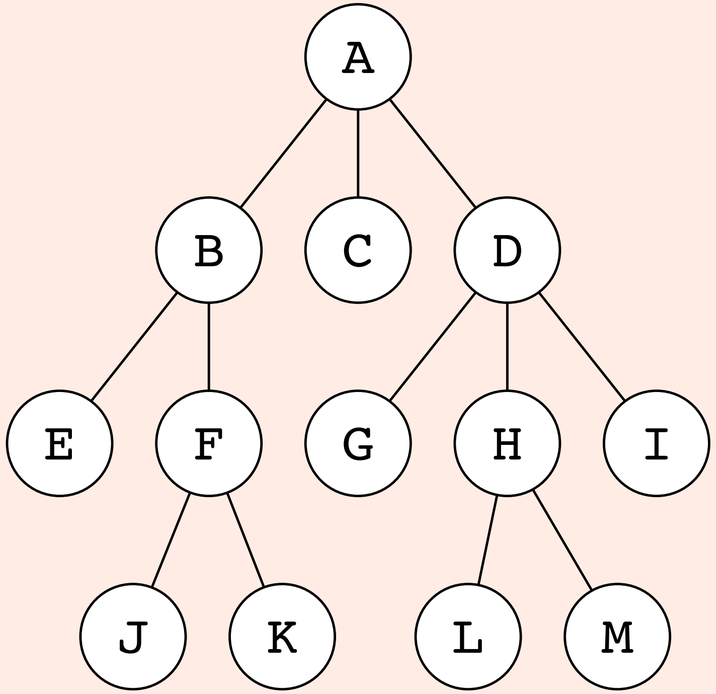
Exercices

- Quelle est le degré de cet arbre ?
- Quel est le chemin du noeud F ?
- Quel est le niveau du noeud M ?
- Quelle est la racine ?
- Quels sont les noeuds internes?
- Quelles sont les feuilles



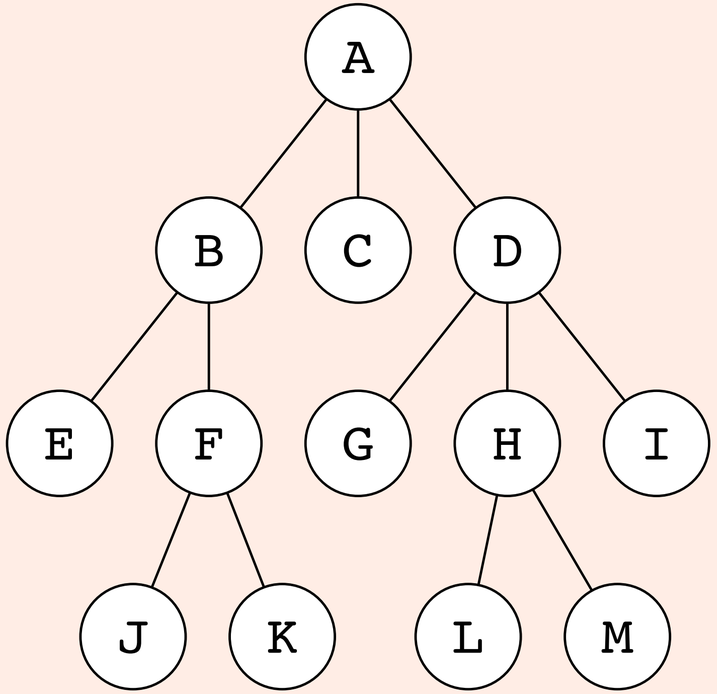
Solution

- Quelle est le degré de cet arbre ?
- Quel est le chemin du noeud F ?
- Quel est le niveau du noeud M ?
- Quelle est la racine ?
- Quels sont les noeuds internes?
- Quelles sont les feuilles



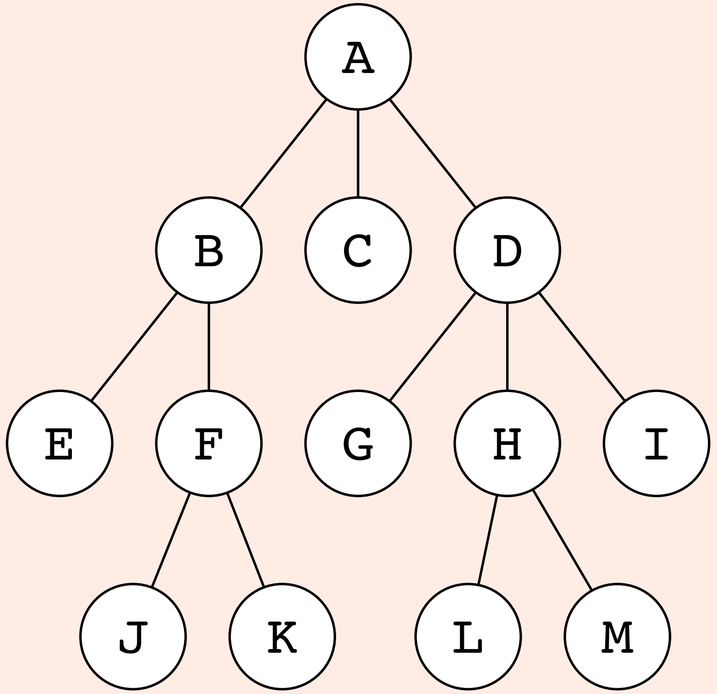
Solution

- Quelle est le degré de cet arbre ? 3
- Quel est le chemin du noeud F ?
- Quel est le niveau du noeud M ?
- Quelle est la racine ?
- Quels sont les noeuds internes?
- Quelles sont les feuilles



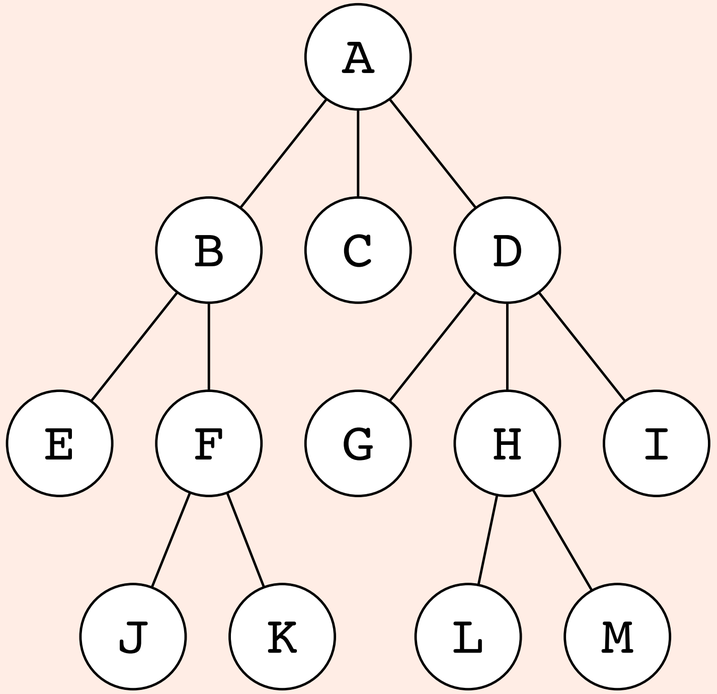
Solution

- Quelle est le degré de cet arbre ? 3
- Quel est le chemin du noeud F ? A B F
- Quel est le niveau du noeud M ?
- Quelle est la racine ?
- Quels sont les noeuds internes?
- Quelles sont les feuilles



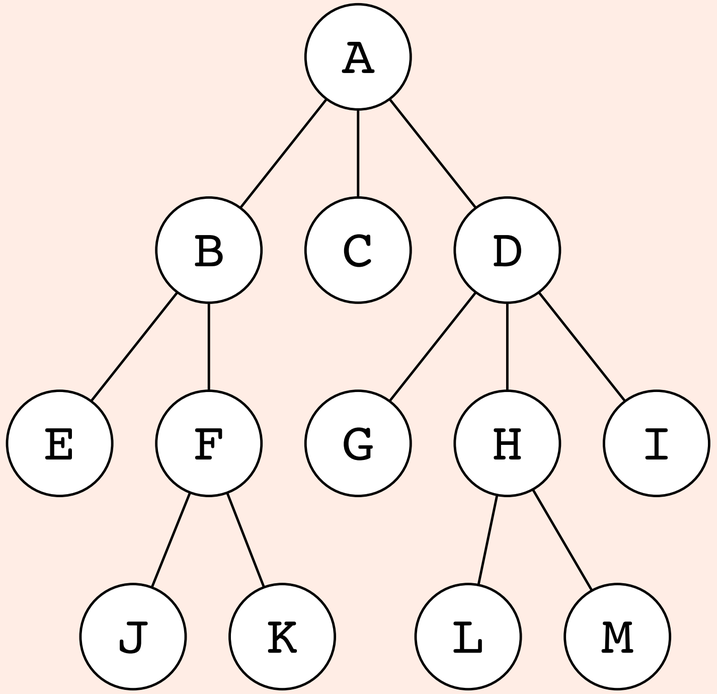
Solution

- Quelle est le degré de cet arbre ? 3
- Quel est le chemin du noeud F ? A B F
- Quel est le niveau du noeud M ? 4
- Quelle est la racine ?
- Quels sont les noeuds internes?
- Quelles sont les feuilles



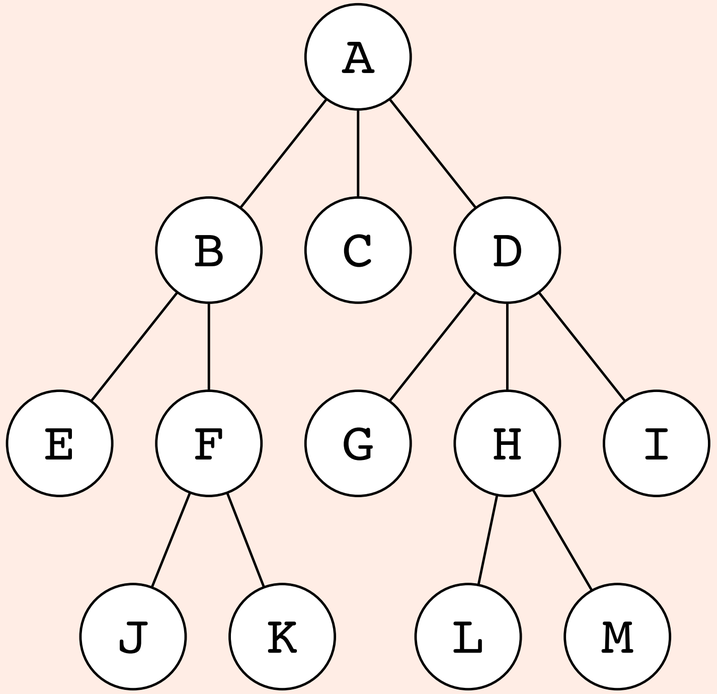
Solution

- Quelle est le degré de cet arbre ? 3
- Quel est le chemin du noeud F ? A B F
- Quel est le niveau du noeud M ? 4
- Quelle est la racine ? A
- Quels sont les noeuds internes?
- Quelles sont les feuilles



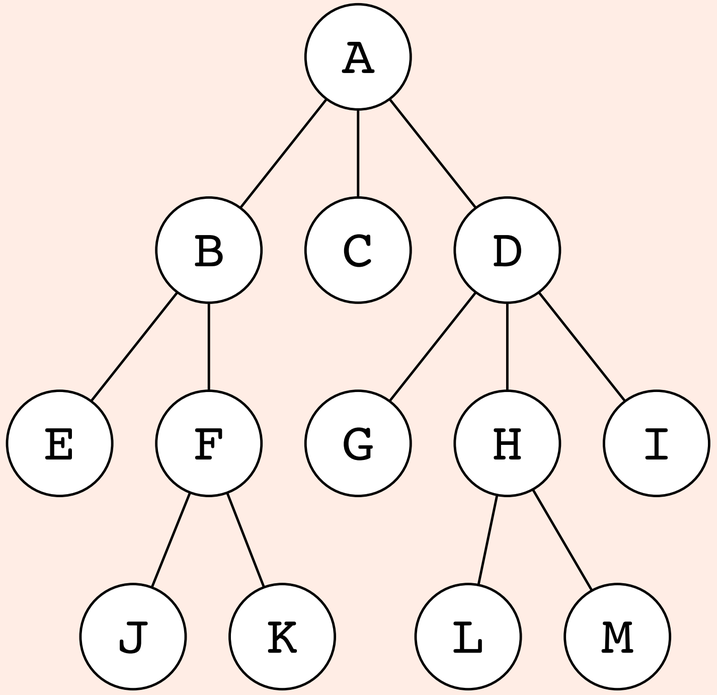
Solution

- Quelle est le degré de cet arbre ? 3
- Quel est le chemin du noeud F ? A B F
- Quel est le niveau du noeud M ? 4
- Quelle est la racine ? A
- Quels sont les noeuds internes? A B D F H
- Quelles sont les feuilles



Solution

- Quelle est le degré de cet arbre ? 3
- Quel est le chemin du noeud F ? A B F
- Quel est le niveau du noeud M ? 4
- Quelle est la racine ? A
- Quels sont les noeuds internes? A B D F H
- Quelles sont les feuilles C E G I J K L M



2. Structure

Mises en oeuvre

- Un noeud par élément, qui stocke son étiquette
- Des pointeurs vers les noeuds enfants
- Un pointeur vers le noeud parent ?

Mises en oeuvre

```
structure Noeud<T>
    T étiquette
    Tableau<Noeud*> enfants
```

- Un noeud par élément, qui stocke son étiquette
- Des pointeurs vers les noeuds enfants
- Un pointeur vers le noeud parent ?

Mises en oeuvre

- Un noeud par élément, qui stocke son étiquette
- Des pointeurs vers les noeuds enfants
- Un pointeur vers le noeud parent ?

```
structure Noeud<T>
    T étiquette
    Tableau<Noeud*> enfants
```

```
structure Noeud<T>
    T étiquette
    Noeud* parent
    Tableau<Noeud*> enfants
```

Mises en oeuvre

- Un noeud par élément, qui stocke son étiquette
- Des pointeurs vers les noeuds enfants
- Un pointeur vers le noeud parent ?

```
structure Noeud<T>
    T étiquette
    Tableau<Noeud*> enfants
```

```
structure Noeud<T>
    T étiquette
    Noeud* parent
    Tableau<Noeud*> enfants
```

```
structure Noeud<T>
    T étiquette
    Liste<Noeud*> enfants
```

Mises en oeuvre

- Un noeud par élément, qui stocke son étiquette
- Des pointeurs vers les noeuds enfants
- Un pointeur vers le noeud parent ?

```
structure Noeud<T>
    T étiquette
    Tableau<Noeud*> enfants
```

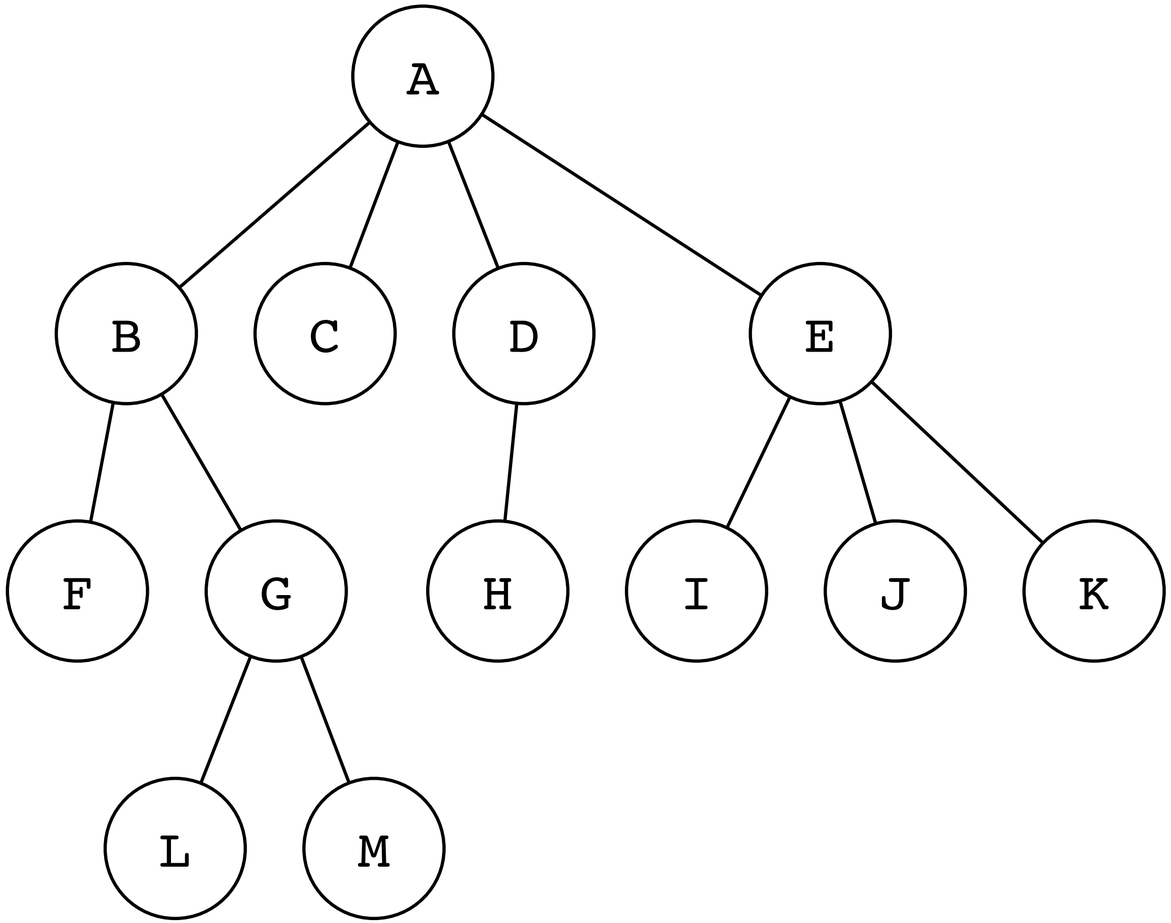
```
structure Noeud<T>
    T étiquette
    Noeud* parent
    Tableau<Noeud*> enfants
```

```
structure Noeud<T>
    T étiquette
    Liste<Noeud*> enfants
```

```
structure Noeud<T>
    T étiquette
    Noeud* parent
    Liste<Noeud*> enfants
```

Sans utiliser de structures linéaires

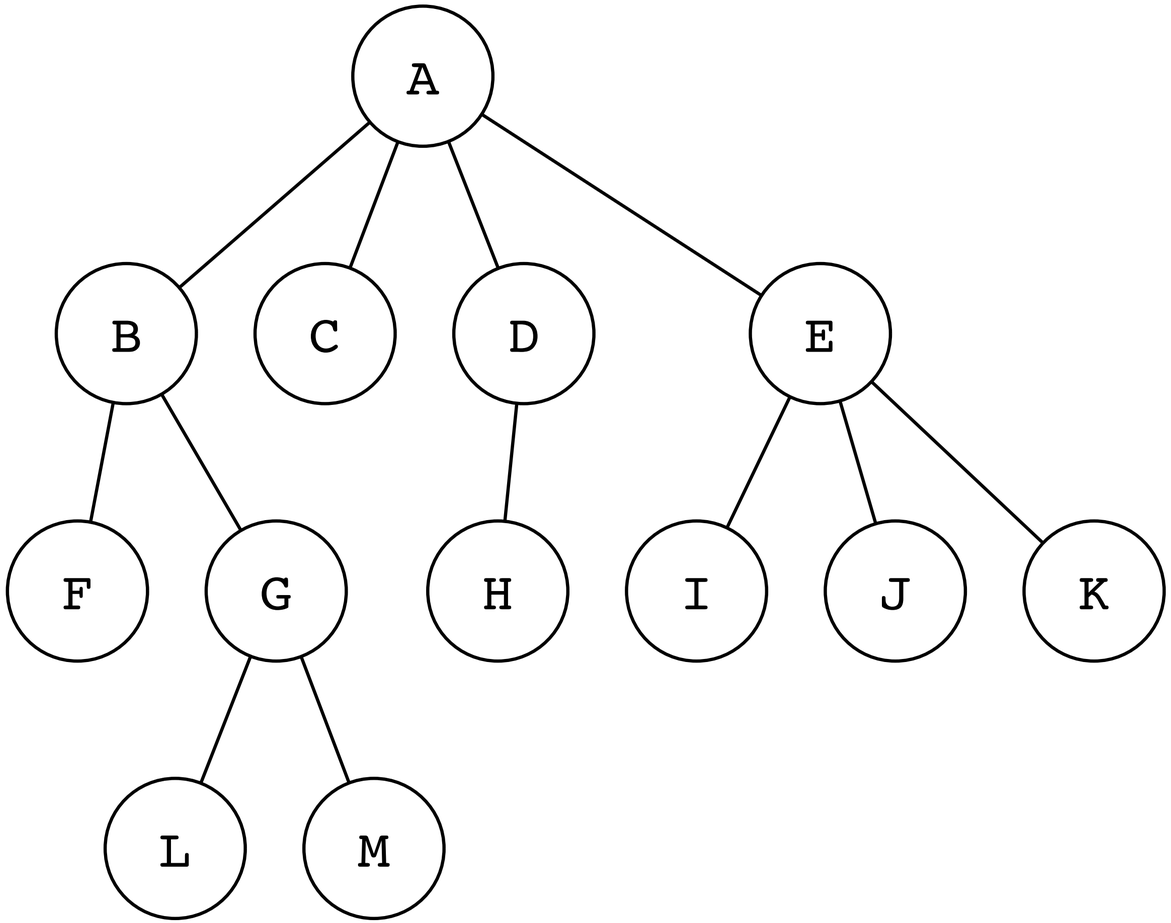
```
structure Noeud<T>
    T étiquette
```



Sans utiliser de structures linéaires

- Un lien descendant vers l'ainé des enfants

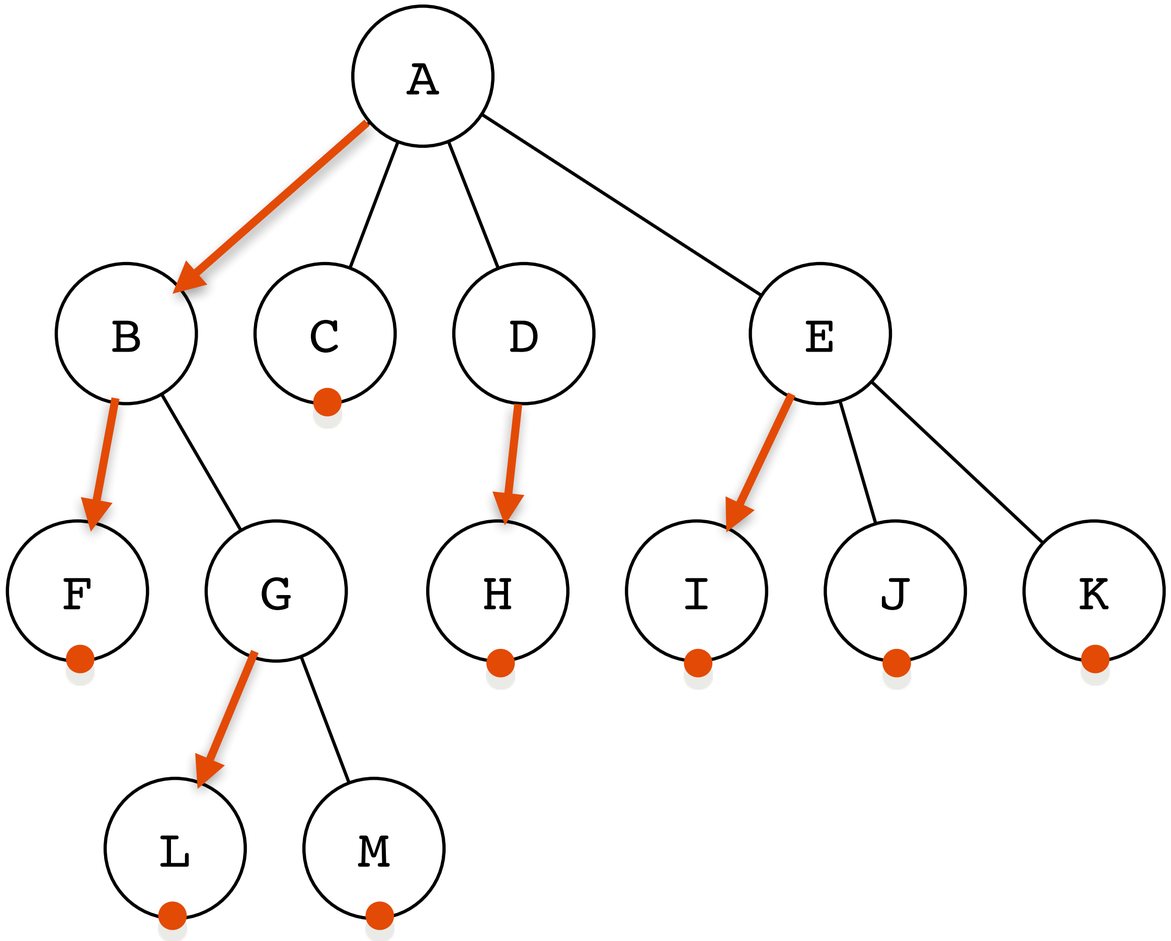
```
structure Noeud<T>
    T étiquette
    Noeud* ainé
```



Sans utiliser de structures linéaires

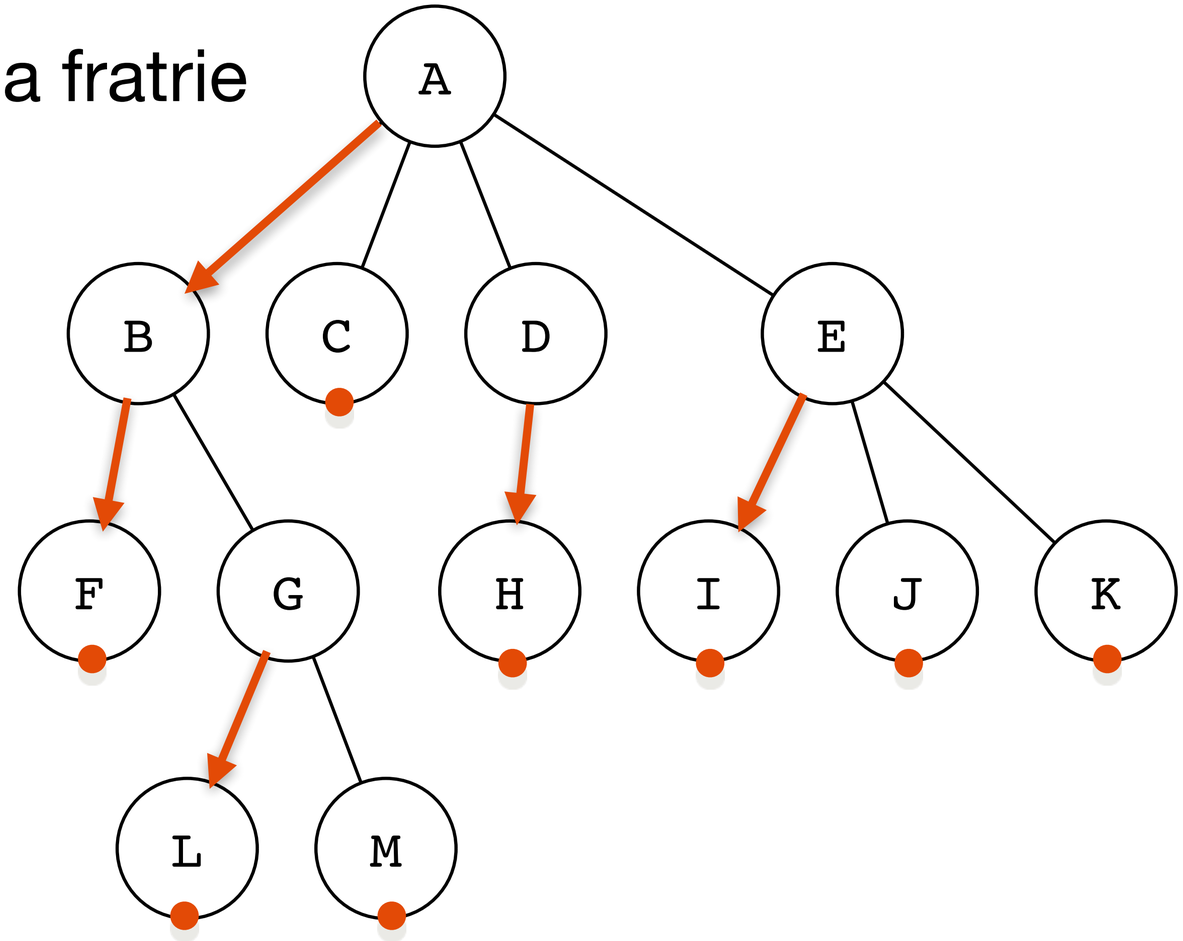
- Un lien descendant vers l'ainé des enfants

```
structure Noeud<T>
    T étiquette
    Noeud* ainé
```



Sans utiliser de structures linéaires

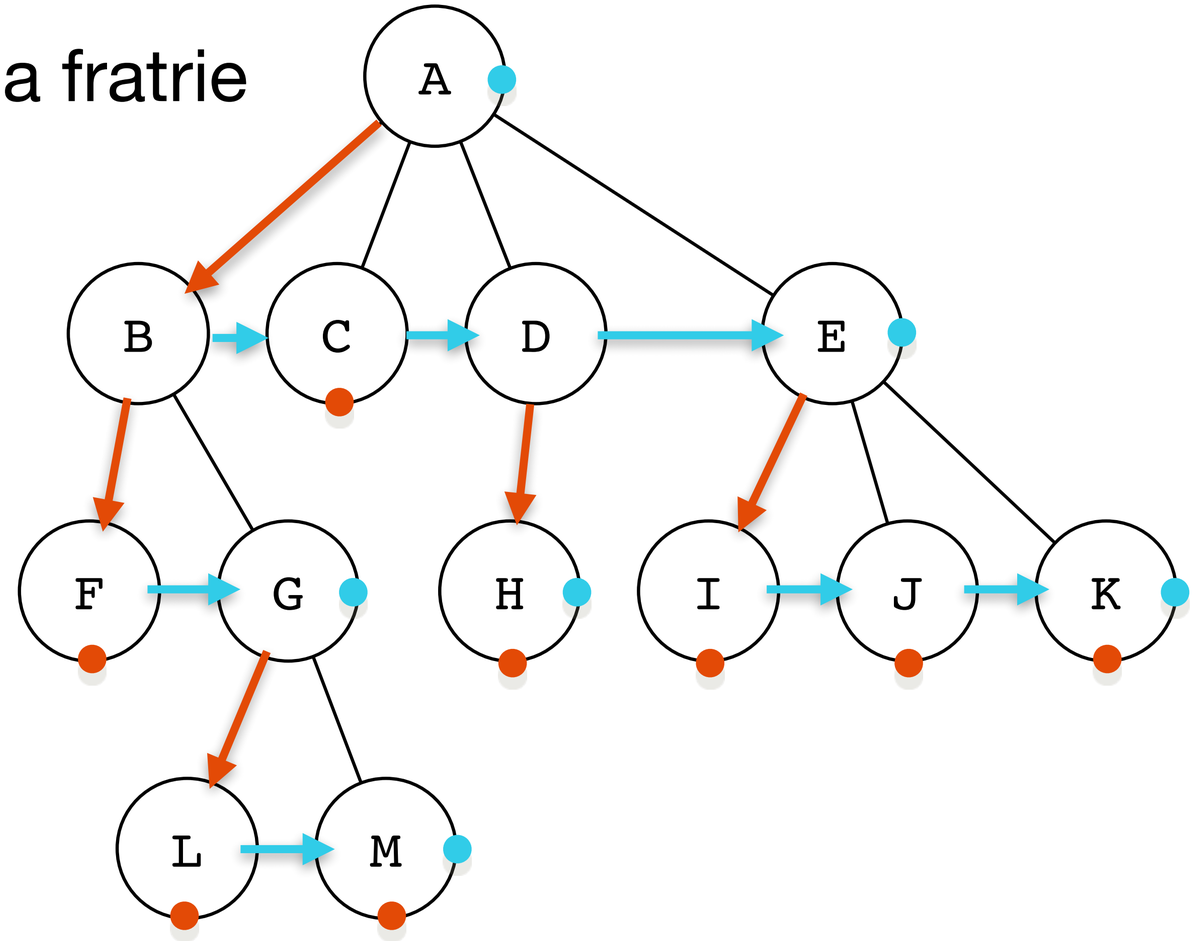
- Un lien descendant vers l'ainé des enfants
- Un lien horizontal vers le puiné dans la fratrie




```
structure Noeud<T>
    T étiquette
    Noeud* ainé
    Noeud* puiné
```

Sans utiliser de structures linéaires

- Un lien descendant vers l'ainé des enfants
- Un lien horizontal vers le puiné dans la fratrie

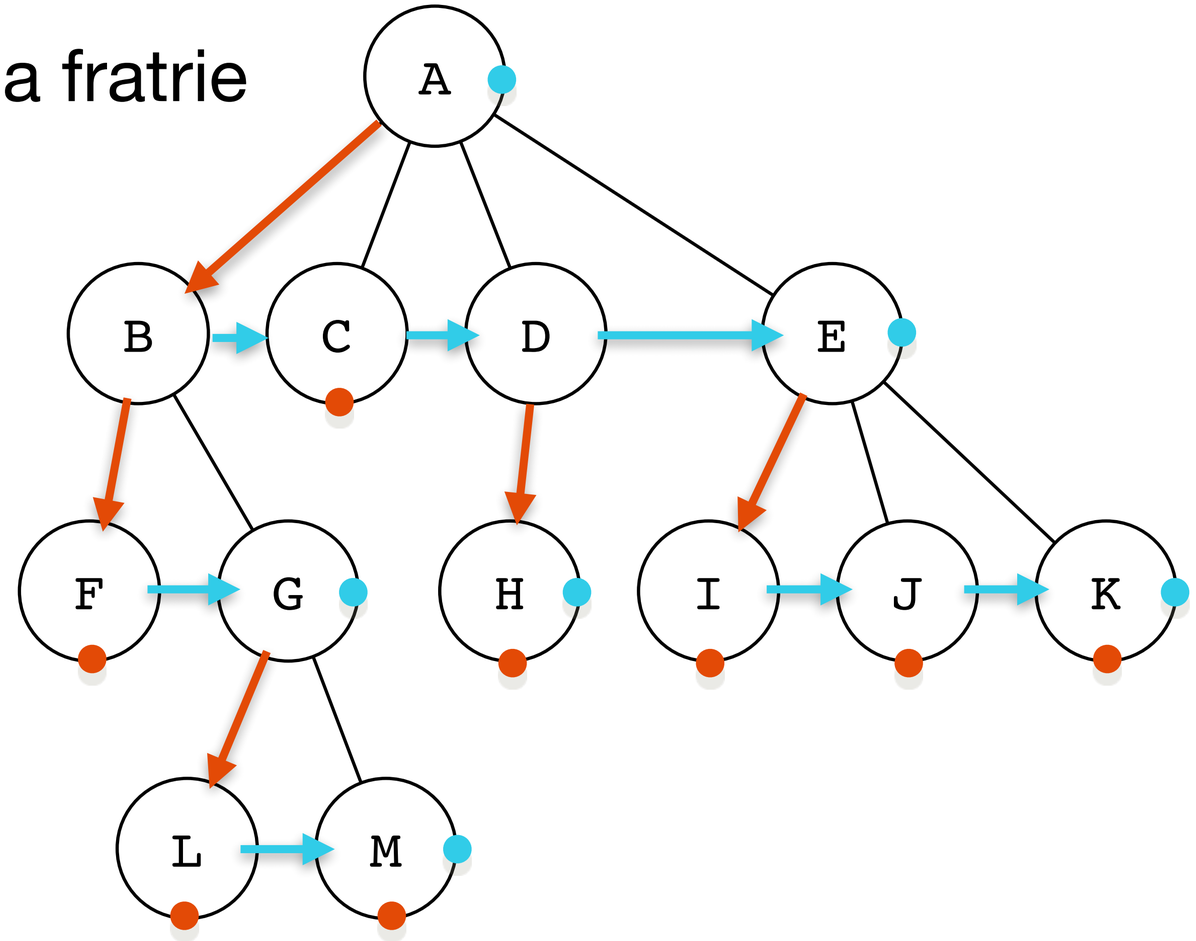



```
structure Noeud<T>
    T étiquette
    Noeud* ainé
    Noeud* puiné
```

Sans utiliser de structures linéaires

- Un lien descendant vers l'ainé des enfants
- Un lien horizontal vers le puiné dans la fratrie
- Un lien montant vers le parent

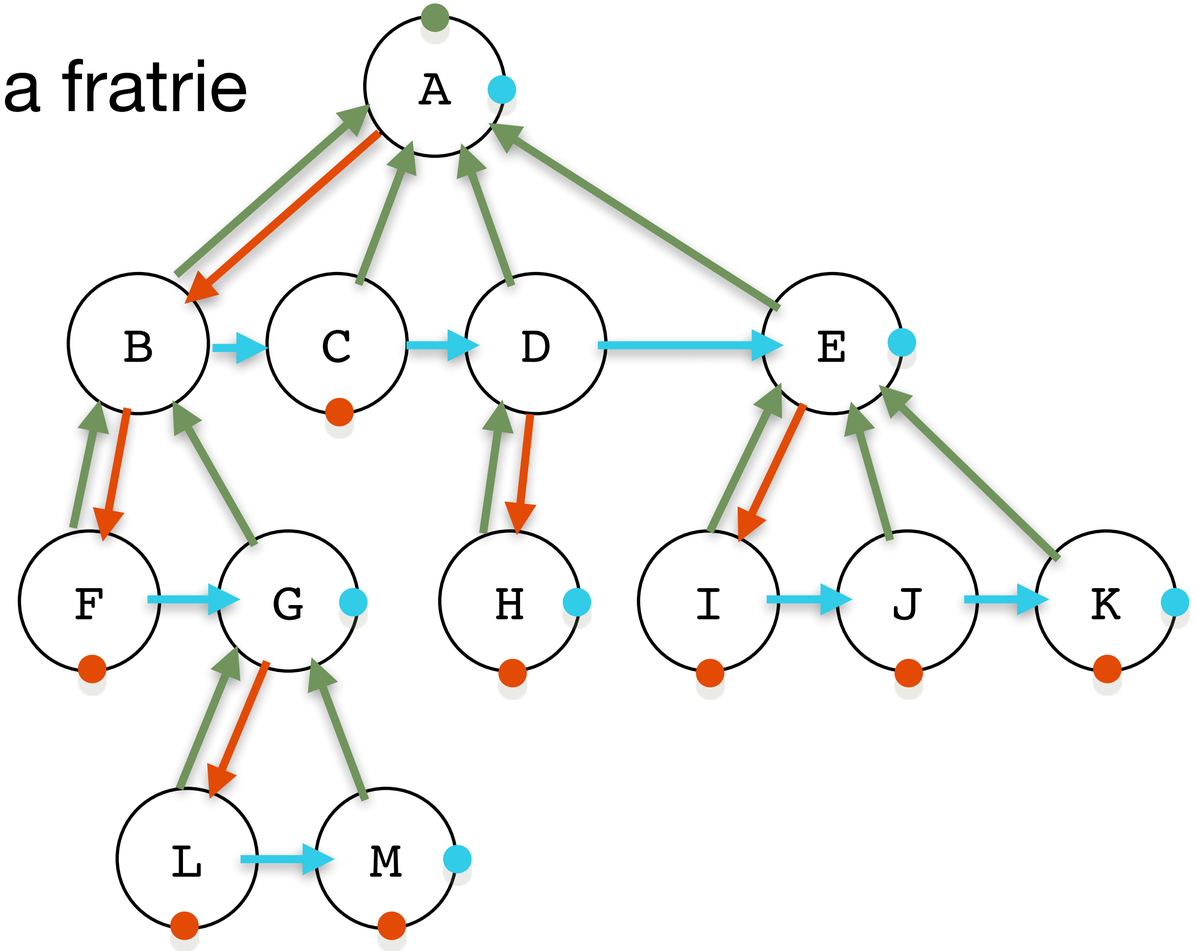
```
structure Noeud<T>
    T étiquette
    Noeud* ainé
    Noeud* puiné
    Noeud* parent
```



Sans utiliser de structures linéaires

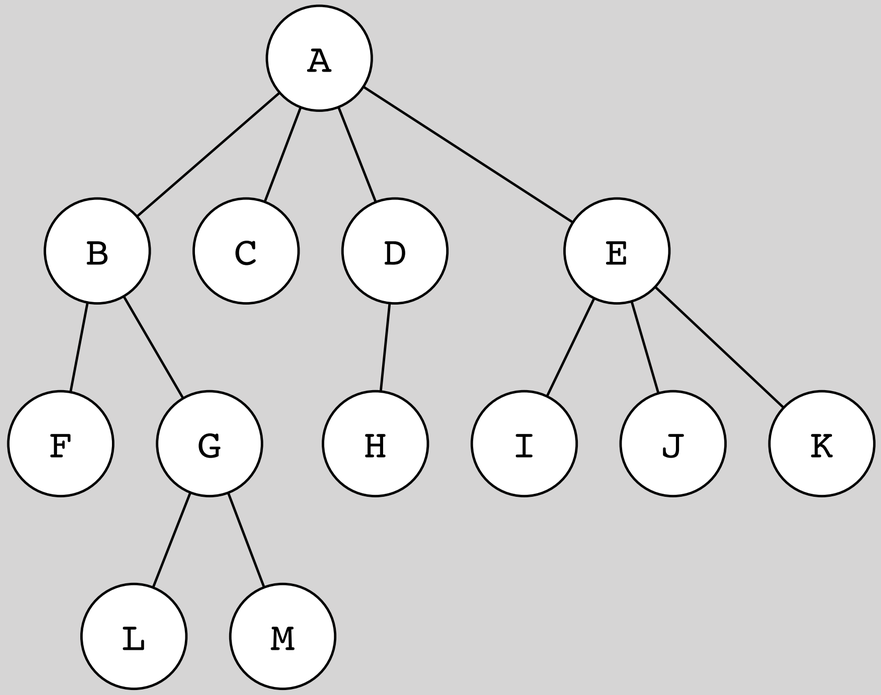
- Un lien descendant vers l'ainé des enfants
- Un lien horizontal vers le puiné dans la fratrie
- Un lien montant vers le parent

```
structure Noeud<T>
    T étiquette
    Noeud* ainé
    Noeud* puiné
    Noeud* parent
```



Exercice

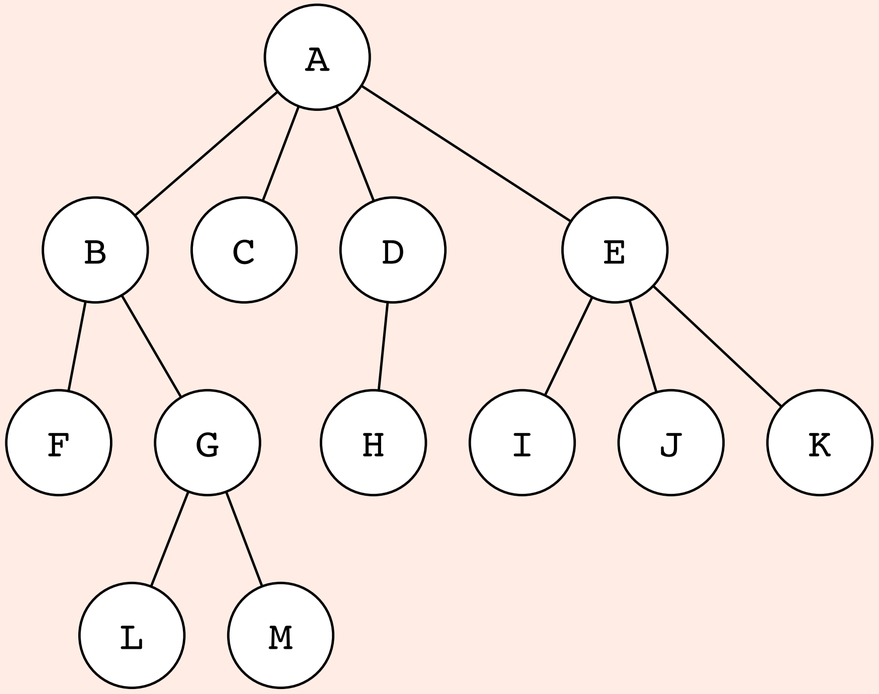
Quelle est le nombre minimum de pointeurs et/ou `size_t` utilisés par les structures ci-contre pour stocker l'arbre ci-dessous?




```
template <typename T> struct Node1 {
    T etiquette;
    Node1* parent;
    array<Node1*,DEGRE> enfants;
};

template <typename T> struct Node2 {
    T etiquette;
    Node2* parent;
    vector<Node2*> enfants;
};

template <typename T> struct Node3 {
    T etiquette;
    Node3* parent;
    list<Node3*> enfants;
};

template <typename T> struct Node4 {
    T etiquette;
    Node4* parent;
    forward_list<Node4*> enfants;
};

template <typename T> struct Node5 {
    T etiquette;
    Node5* parent;
    Node5* ainé;
    Node5* puiné;
};
```

Exercice

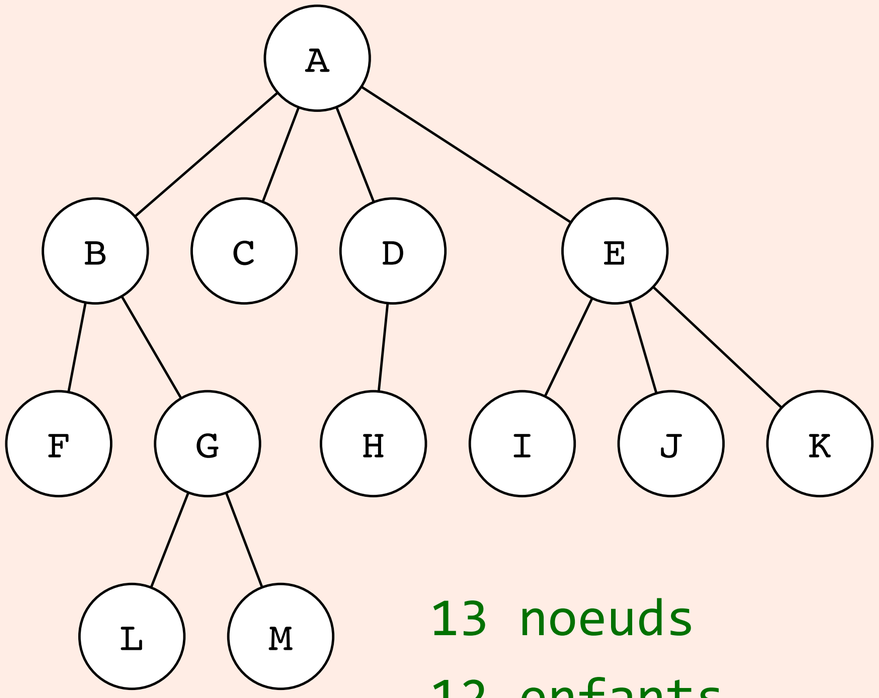
Quelle est le nombre minimum de pointeurs et/ou `size_t` utilisés par les structures ci-contre pour stocker l'arbre ci-dessous?




```
template <typename T> struct Node1 {
    T etiquette;
    Node1* parent;
    array<Node1*,DEGRE> enfants;
};

template <typename T> struct Node2 {
    T etiquette;
    Node2* parent;
    vector<Node2*> enfants;
};

template <typename T> struct Node3 {
    T etiquette;
    Node3* parent;
    list<Node3*> enfants;
};

template <typename T> struct Node4 {
    T etiquette;
    Node4* parent;
    forward_list<Node4*> enfants;
};

template <typename T> struct Node5 {
    T etiquette;
    Node5* parent;
    Node5* ainé;
    Node5* puiné;
};
```

Exercice

Quelle est le nombre minimum de pointeurs et/ou `size_t` utilisés par les structures ci-contre pour stocker l'arbre ci-dessous?



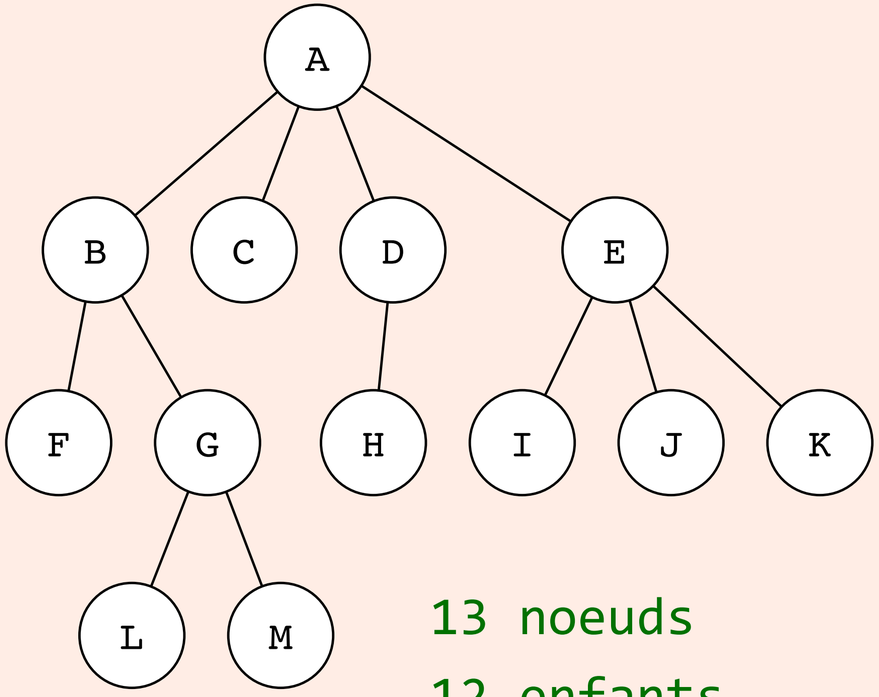
13 noeuds

12 enfants

```
template <typename T> struct Node1 {
    T etiquette;
    Node1* parent;
    array<Node1*,DEGRE> enfants;
};

template <typename T> struct Node2 {
    T etiquette;
    Node2* parent;
    vector<Node2*> enfants;
};

template <typename T> struct Node3 {
    T etiquette;
    Node3* parent;
    list<Node3*> enfants;
};

template <typename T> struct Node4 {
    T etiquette;
    Node4* parent;
    forward_list<Node4*> enfants;
};

template <typename T> struct Node5 {
    T etiquette;
    Node5* parent;
    Node5* ainé;
    Node5* puiné;
};
```

Exercice

Quelle est le nombre minimum de pointeurs et/ou `size_t` utilisés par les structures ci-contre pour stocker l'arbre ci-dessous?



13 noeuds

12 enfants

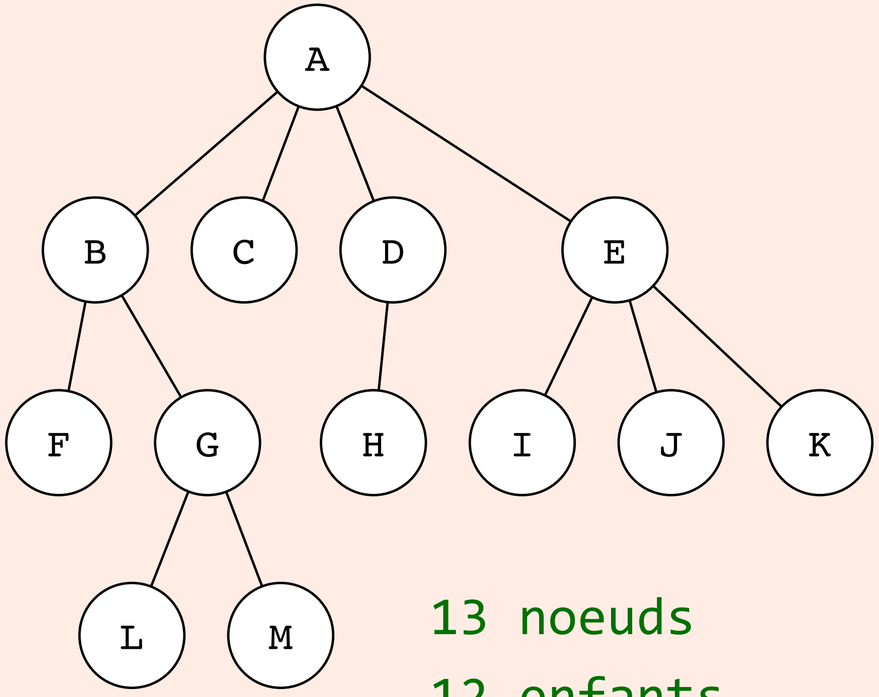
```cpp
template <typename T> struct Node1 {
    T etiquette;
    Node1* parent;
    array<Node1*,DEGRE> enfants;
};

template <typename T> struct Node2 {
    T etiquette;
    Node2* parent;
    vector<Node2*> enfants;
};

template <typename T> struct Node3 {
    T etiquette;
    Node3* parent;
    list<Node3*> enfants;
};

template <typename T> struct Node4 {
    T etiquette;
    Node4* parent;
    forward_list<Node4*> enfants;
};

template <typename T> struct Node5 {
    T etiquette;
    Node5* parent;
    Node5* ainé;
    Node5* puiné;
};
```

Degré 4

13 × 5 = 65

Exercice

Quelle est le nombre minimum de pointeurs et/ou `size_t` utilisés par les structures ci-contre pour stocker l'arbre ci-dessous?



13 noeuds

12 enfants

```cpp
template <typename T> struct Node1 {
    T etiquette;
    Node1* parent;
    array<Node1*,DEGRE> enfants;
};

template <typename T> struct Node2 {
    T etiquette;
    Node2* parent;
    vector<Node2*> enfants;
};

template <typename T> struct Node3 {
    T etiquette;
    Node3* parent;
    list<Node3*> enfants;
};

template <typename T> struct Node4 {
    T etiquette;
    Node4* parent;
    forward_list<Node4*> enfants;
};

template <typename T> struct Node5 {
    T etiquette;
    Node5* parent;
    Node5* ainé;
    Node5* puiné;
};
```

Degré 4

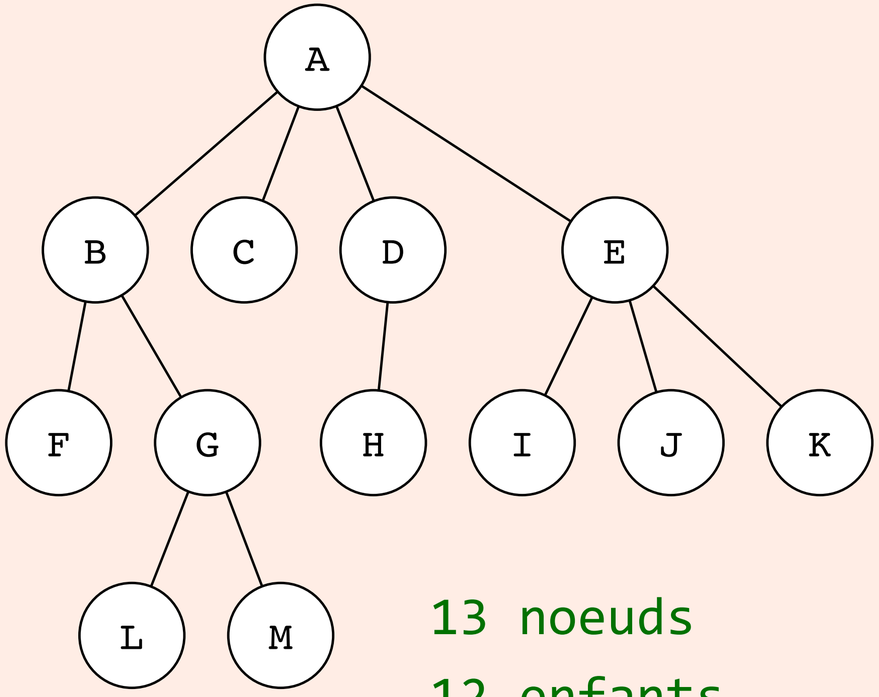
13 × 5 = 65

3+n par vector de n ptr

13 × 4 + 12 = 64

Exercice

Quelle est le nombre minimum de pointeurs et/ou `size_t` utilisés par les structures ci-contre pour stocker l'arbre ci-dessous?



13 noeuds

12 enfants

```cpp
template <typename T> struct Node1 {
    T etiquette;
    Node1* parent;
    array<Node1*,DEGRE> enfants;
};

template <typename T> struct Node2 {
    T etiquette;
    Node2* parent;
    vector<Node2*> enfants;
};

template <typename T> struct Node3 {
    T etiquette;
    Node3* parent;
    list<Node3*> enfants;
};

template <typename T> struct Node4 {
    T etiquette;
    Node4* parent;
    forward_list<Node4*> enfants;
};

template <typename T> struct Node5 {
    T etiquette;
    Node5* parent;
    Node5* ainé;
    Node5* puiné;
};
```

Degré 4

13 × 5 = 65

3+n par vector de n ptr

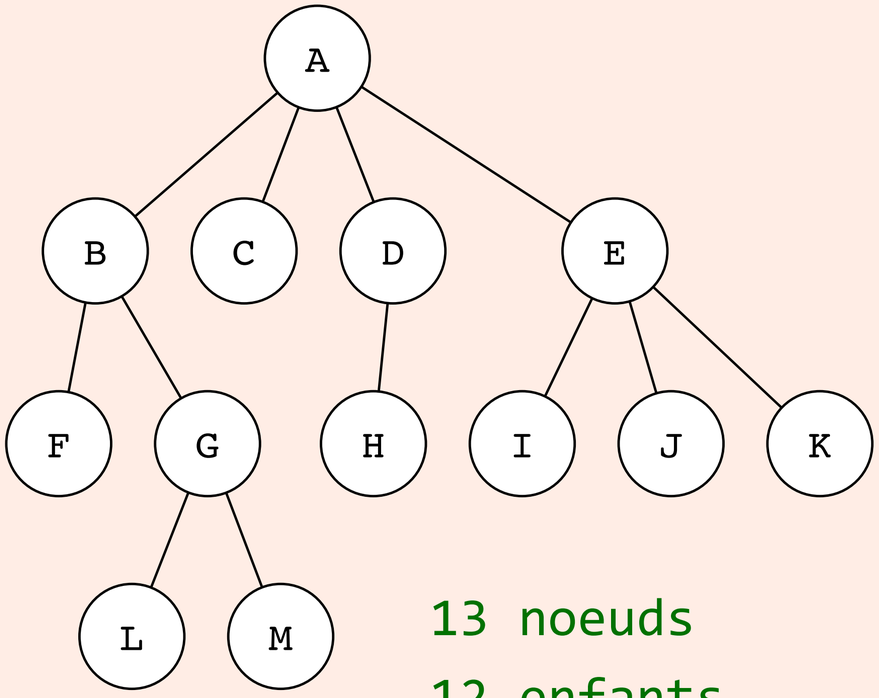
13 × 4 + 12 = 64

3+3n ptr / liste de n ptr

13 × 4 + 12 × 3 = 88

Exercice

Quelle est le nombre minimum de pointeurs et/ou `size_t` utilisés par les structures ci-contre pour stocker l'arbre ci-dessous?



13 noeuds

12 enfants

```cpp
template <typename T> struct Node1 {
    T etiquette;
    Node1* parent;
    array<Node1*,DEGRE> enfants;
};

template <typename T> struct Node2 {
    T etiquette;
    Node2* parent;
    vector<Node2*> enfants;
};

template <typename T> struct Node3 {
    T etiquette;
    Node3* parent;
    list<Node3*> enfants;
};

template <typename T> struct Node4 {
    T etiquette;
    Node4* parent;
    forward_list<Node4*> enfants;
};

template <typename T> struct Node5 {
    T etiquette;
    Node5* parent;
    Node5* ainé;
    Node5* puiné;
};
```

Degré 4

13 × 5 = 65

3+n par vector de n ptr

13 × 4 + 12 = 64

3+3n ptr / liste de n ptr

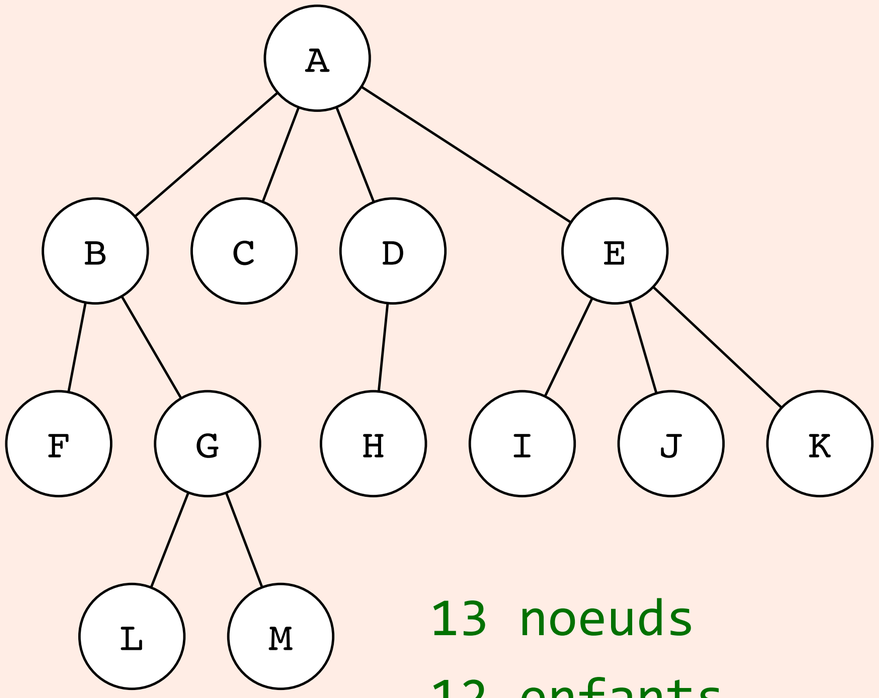
13 × 4 + 12 × 3 = 88

1+2n ptr / liste de n ptr

13 × 2 + 12 × 2 = 50

Exercice

Quelle est le nombre minimum de pointeurs et/ou `size_t` utilisés par les structures ci-contre pour stocker l'arbre ci-dessous?



13 noeuds

12 enfants

```cpp
template <typename T> struct Node1 {
    T etiquette;
    Node1* parent;
    array<Node1*,DEGRE> enfants;
};

template <typename T> struct Node2 {
    T etiquette;
    Node2* parent;
    vector<Node2*> enfants;
};

template <typename T> struct Node3 {
    T etiquette;
    Node3* parent;
    list<Node3*> enfants;
};

template <typename T> struct Node4 {
    T etiquette;
    Node4* parent;
    forward_list<Node4*> enfants;
};

template <typename T> struct Node5 {
    T etiquette;
    Node5* parent;
    Node5* ainé;
    Node5* puiné;
};
```

Degré 4

13 × 5 = 65

3+n par vector de n ptr

13 × 4 + 12 = 64

3+3n ptr / liste de n ptr

13 × 4 + 12 × 3 = 88

1+2n ptr / liste de n ptr

13 × 2 + 12 × 2 = 50

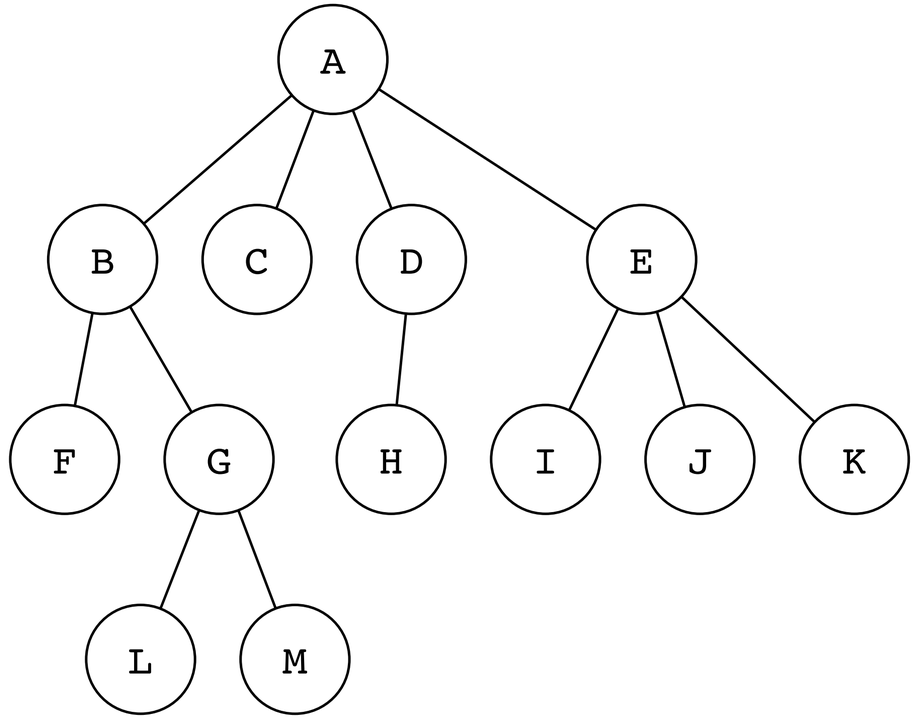
3 ptr par élément

13 × 3 = 39

3. Parcours

Parcours en profondeur

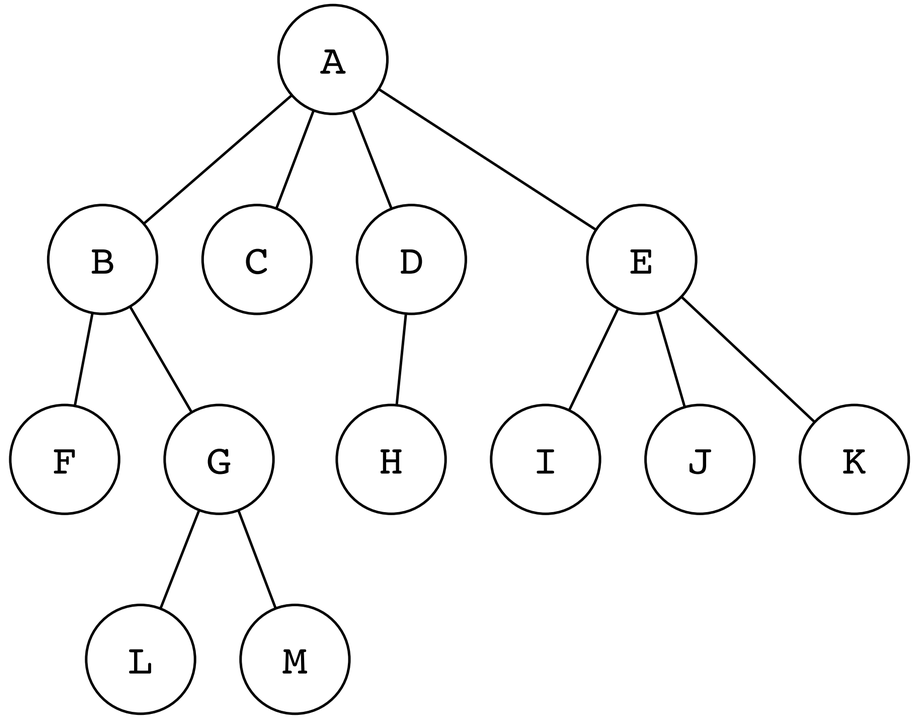
- Comment **visiter** tous les noeuds d'un arbre ?
- **Récursivement :** visiter la racine, puis parcourir tous les sous-arbres dont ses enfants sont la racine
- **Cas trivial :** ne rien faire pour un arbre vide



Parcours en profondeur

- Comment **visiter** tous les noeuds d'un arbre ?
- **Récursivement :** visiter la racine, puis parcourir tous les sous-arbres dont ses enfants sont la racine
- **Cas trivial :** ne rien faire pour un arbre vide

```
fonction profondeur (r, fn)
  si r != ∅
    fn(r)
    pour tout enfant e de r
      profondeur(e, fn)
```



Primogéniture

- King Albert II (born 1934)
  - **King Philippe** (b. 1960)
    - **(1)** Princess Elisabeth, Duchess of Brabant (b. 2001)[1]
    - **(2)** Prince Gabriel (b. 2003)[1]
    - **(3)** Prince Emmanuel (b. 2005)[1]
    - **(4)** Princess Eléonore (b. 2008)[1]
  - **(5)** Princess Astrid, Archduchess of Austria-Este (b. 1962)[1]
    - **(6)** Prince Amedeo, Archduke of Austria-Este (b. 1986)[1]
      - **(7)** Archduchess Anna Astrid of Austria-Este (b. 2016)
      - **(8)** Archduke Maximilian of Austria-Este (b. 2019)
    - **(9)** Princess Maria Laura, Archduchess of Austria-Este (b. 1988)[1]
    - **(10)** Prince Joachim, Archduke of Austria-Este (b. 1991)[1]
    - **(11)** Princess Luisa Maria, Archduchess of Austria-Este (b. 1995)[1]
    - **(12)** Princess Laetitia Maria, Archduchess of Austria-Este (b. 2003)[1]
  - **(13)** Prince Laurent (b. 1963)[1]
    - **(14)** Princess Louise (b. 2004)[1]
    - **(15)** Prince Nicolas (b. 2005)[1]
    - **(16)** Prince Aymeric (b. 2005)[1]

Qui succèdera à Philippe, roi des Belges ?

Primogéniture

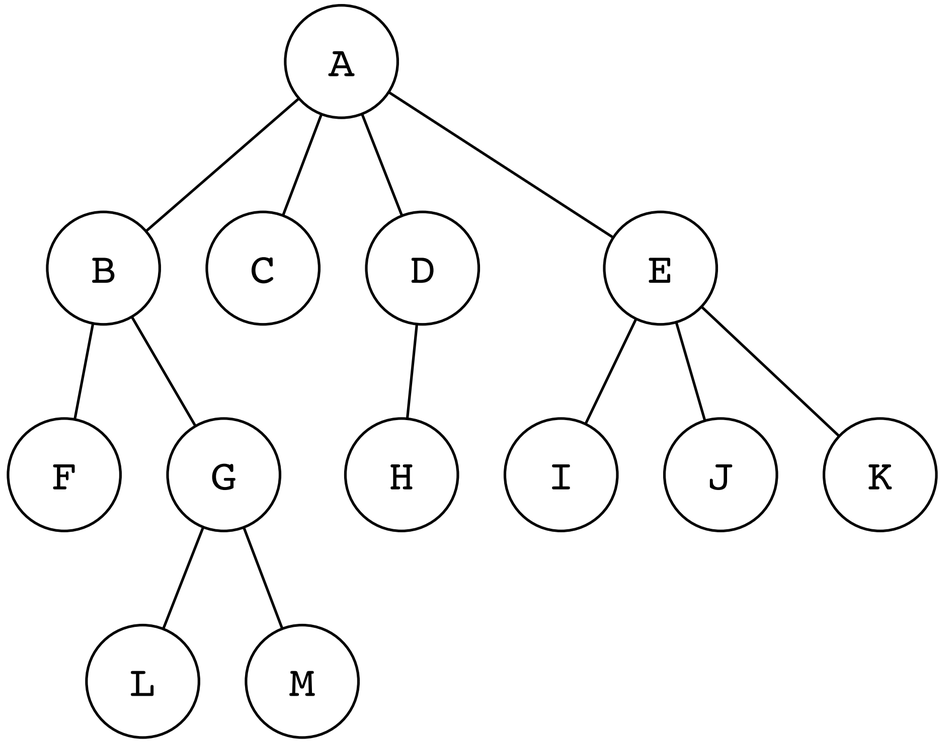
- King Albert II (born 1934)
  - **King Philippe** (b. 1960)
    - **(1)** Princess Elisabeth, Duchess of Brabant (b. 2001)[1]
    - **(2)** Prince Gabriel (b. 2003)[1]
    - **(3)** Prince Emmanuel (b. 2005)[1]
    - **(4)** Princess Eléonore (b. 2008)[1]
  - **(5)** Princess Astrid, Archduchess of Austria-Este (b. 1962)[1]
    - **(6)** Prince Amedeo, Archduke of Austria-Este (b. 1986)[1]
      - **(7)** Archduchess Anna Astrid of Austria-Este (b. 2016)
      - **(8)** Archduke Maximilian of Austria-Este (b. 2019)
    - **(9)** Princess Maria Laura, Archduchess of Austria-Este (b. 1988)[1]
    - **(10)** Prince Joachim, Archduke of Austria-Este (b. 1991)[1]
    - **(11)** Princess Luisa Maria, Archduchess of Austria-Este (b. 1995)[1]
    - **(12)** Princess Laetitia Maria, Archduchess of Austria-Este (b. 2003)[1]
  - **(13)** Prince Laurent (b. 1963)[1]
    - **(14)** Princess Louise (b. 2004)[1]
    - **(15)** Prince Nicolas (b. 2005)[1]
    - **(16)** Prince Aymeric (b. 2005)[1]

Qui succèdera à Philippe, roi des Belges ?

Parcours post-ordonné

```
fonction préordre (r, fn)
  si r != ∅
    fn(r)
    pour tout enfant e de r
      profondeur(e, fn)
```

- Revenons sur le parcours en profondeur
- Pourquoi visiter la racine **avant** les enfants et pas **après**?
- Utile par exemple pour détruire la structure

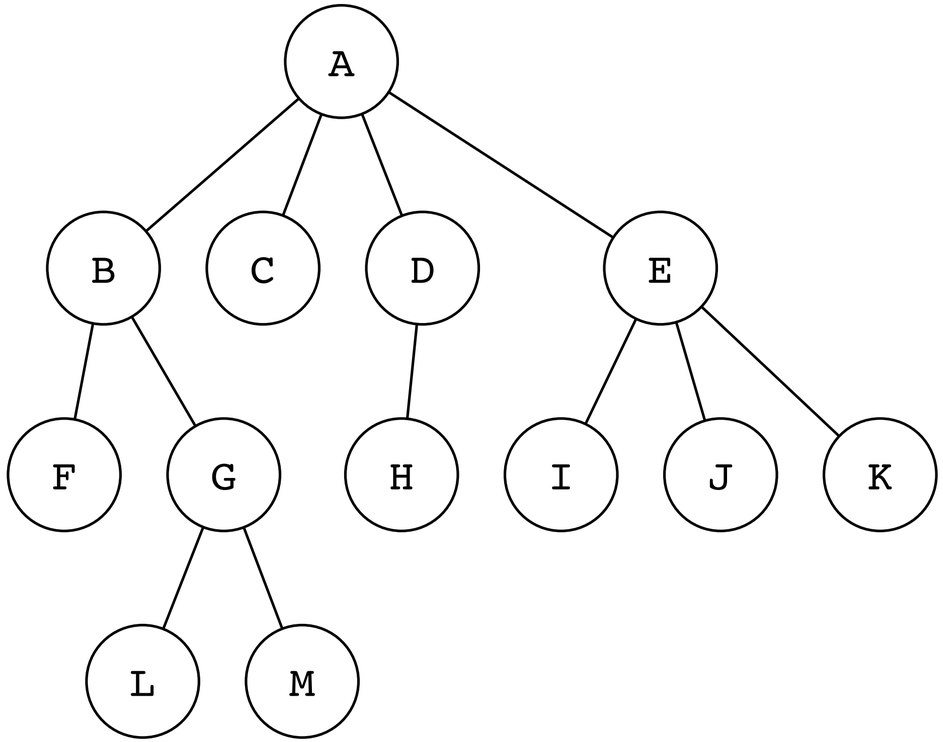


Parcours post-ordonné

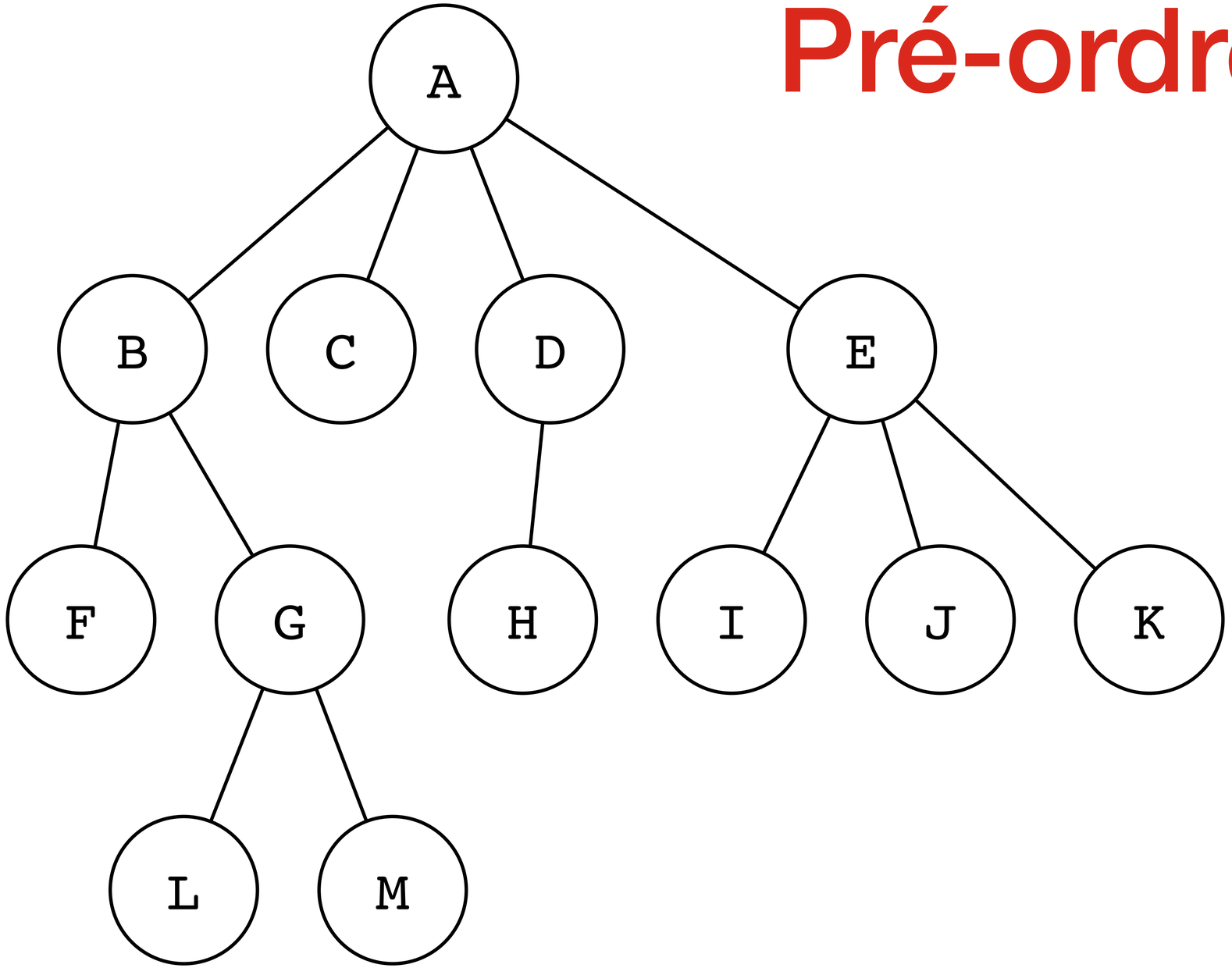
```
fonction préordre (r, fn)
  si r != ∅
    fn(r)
    pour tout enfant e de r
      profondeur(e, fn)
```

```
fonction postordre (r, fn)
  si r != ∅
    pour tout enfant e de r
      postordre(e, fn)
    fn(r)
```

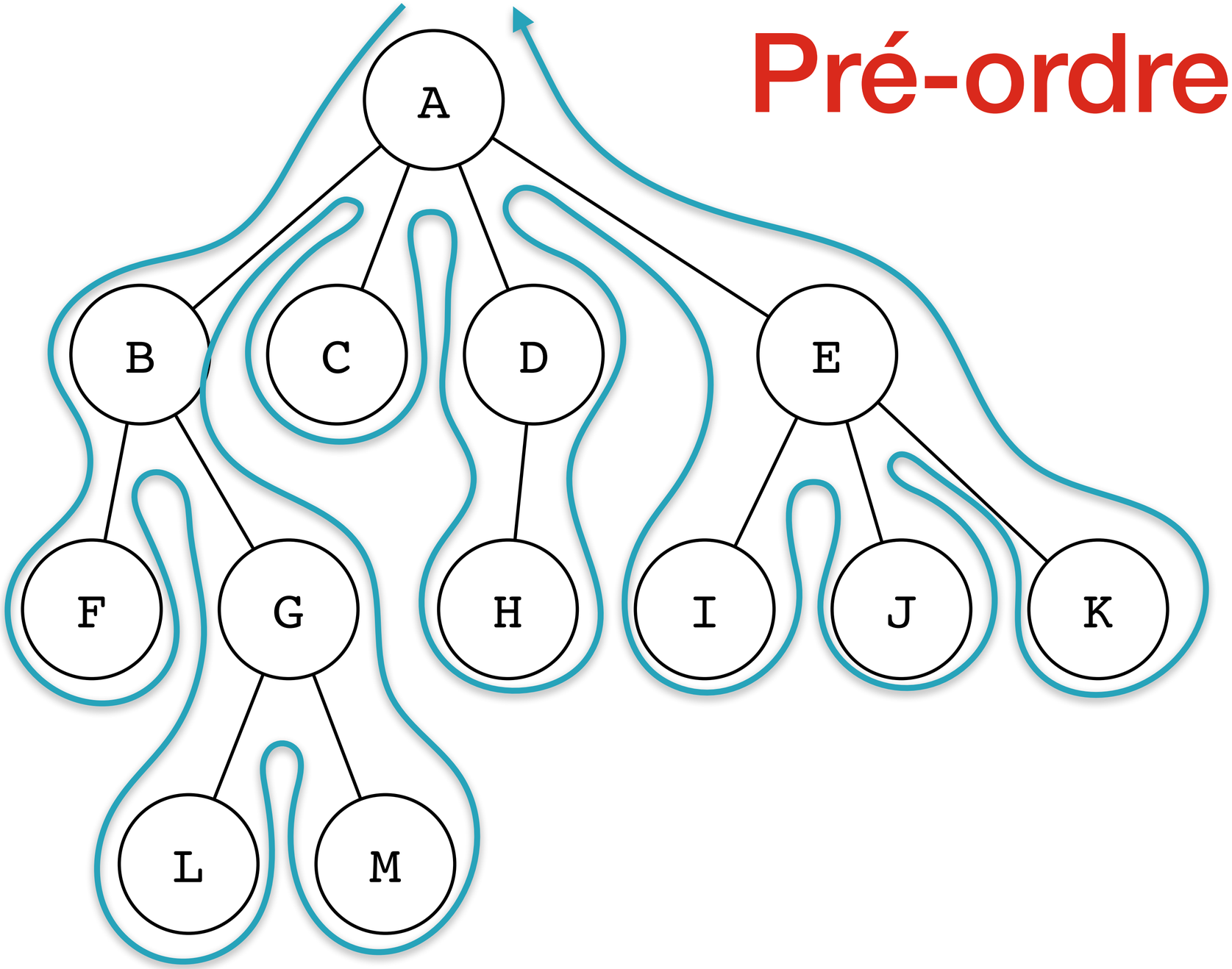
- Revenons sur le parcours en profondeur
- Pourquoi visiter la racine **avant** les enfants et pas **après**?
- Utile par exemple pour détruire la structure



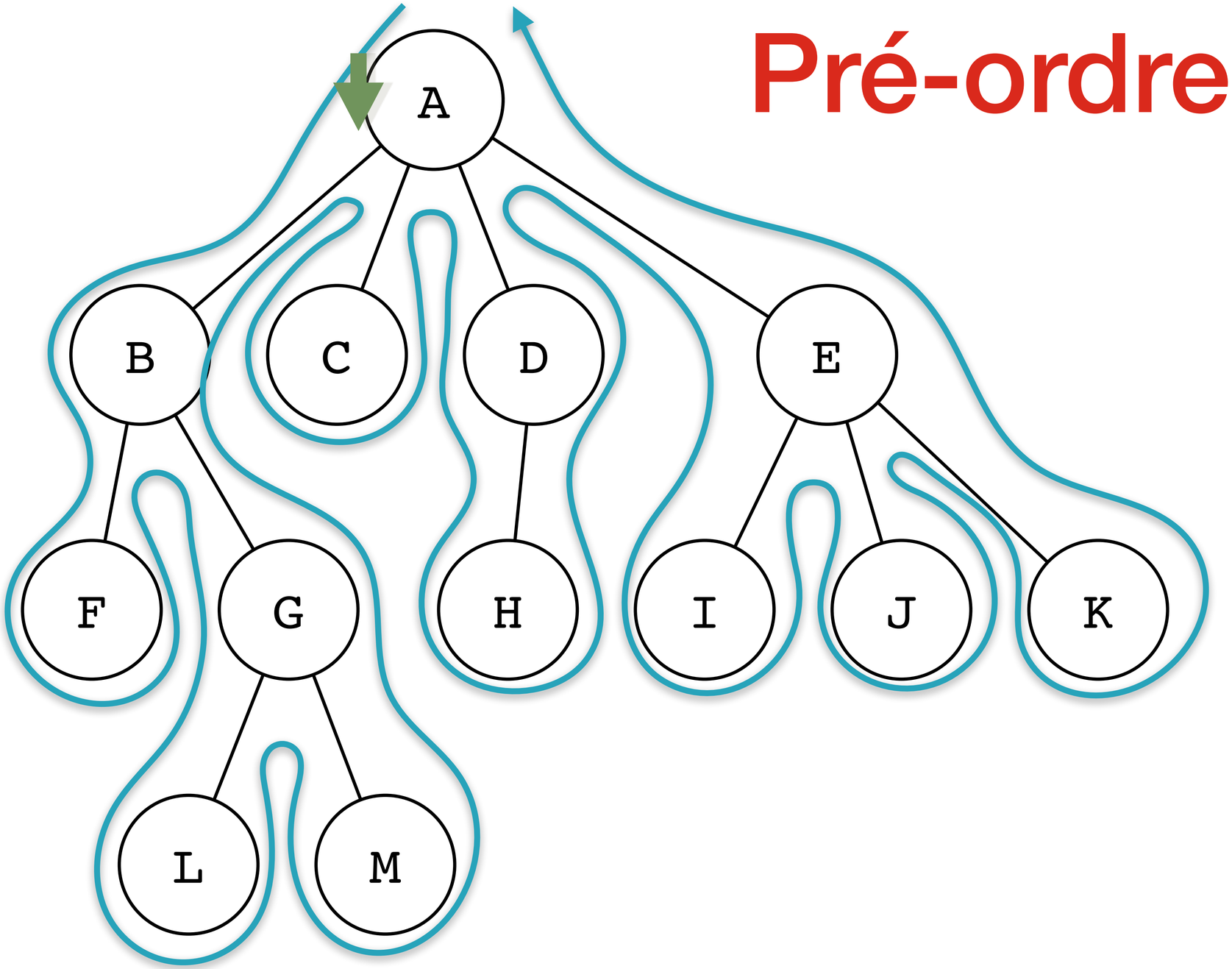
Pré-ordre



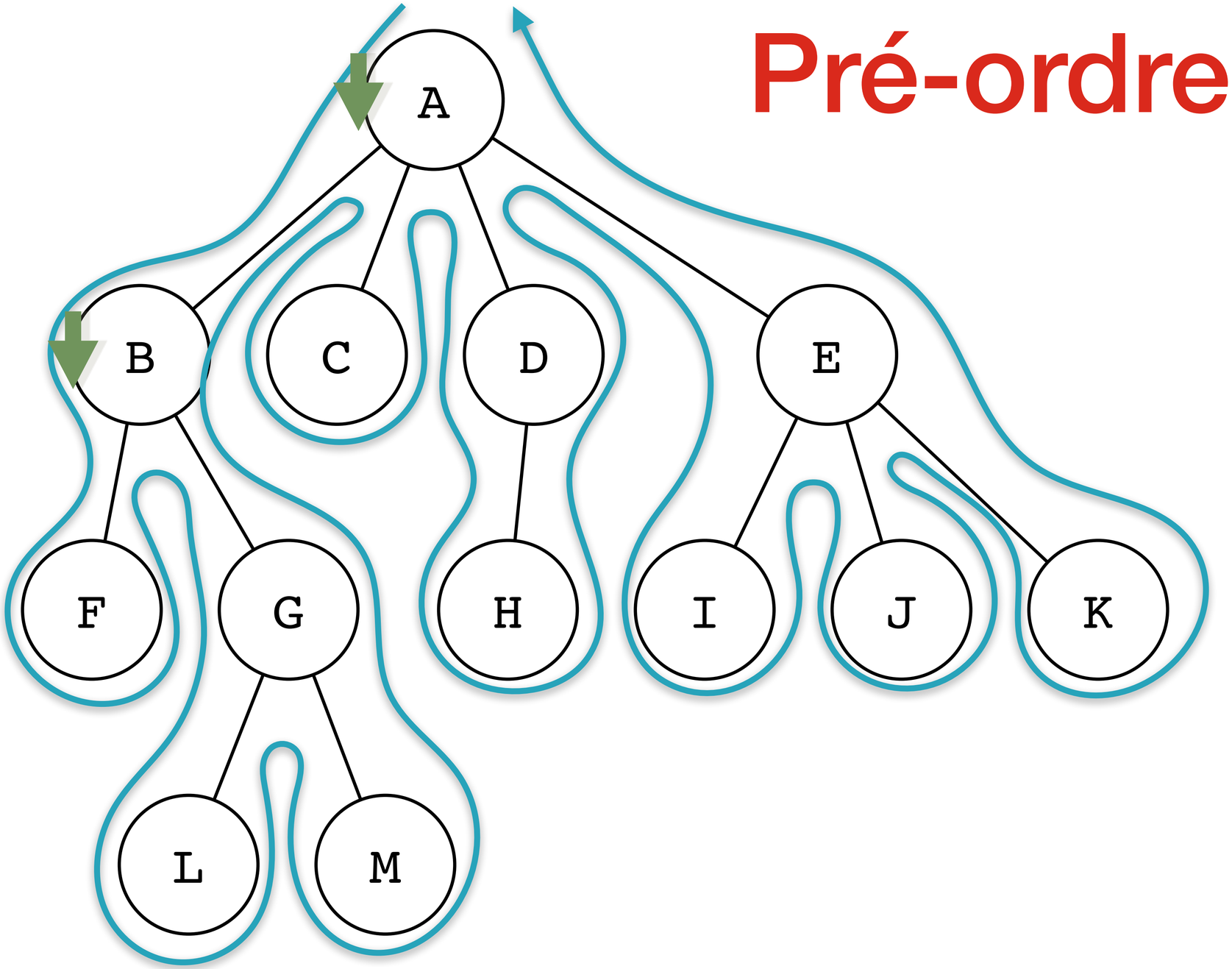
Pré-ordre



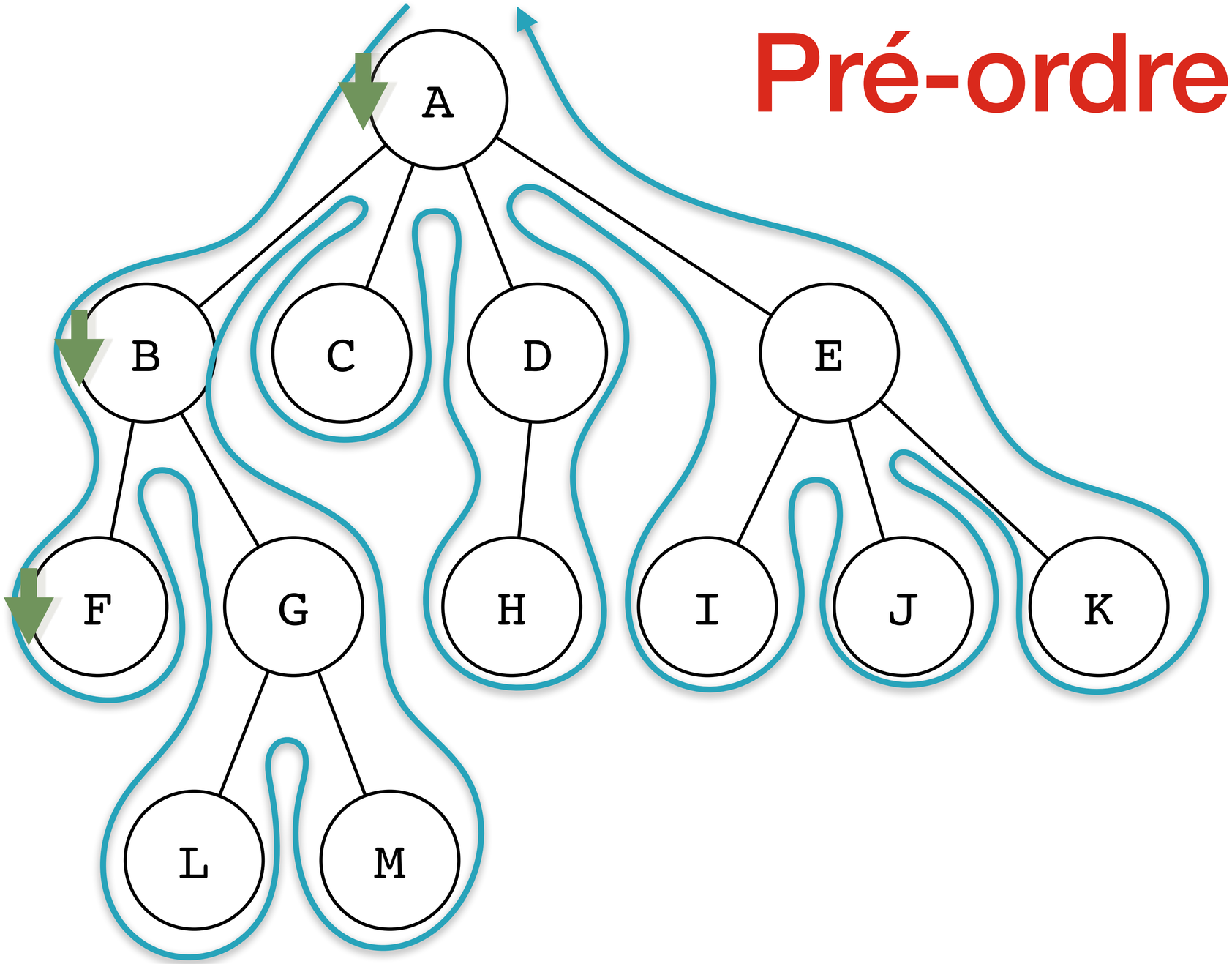
Pré-ordre



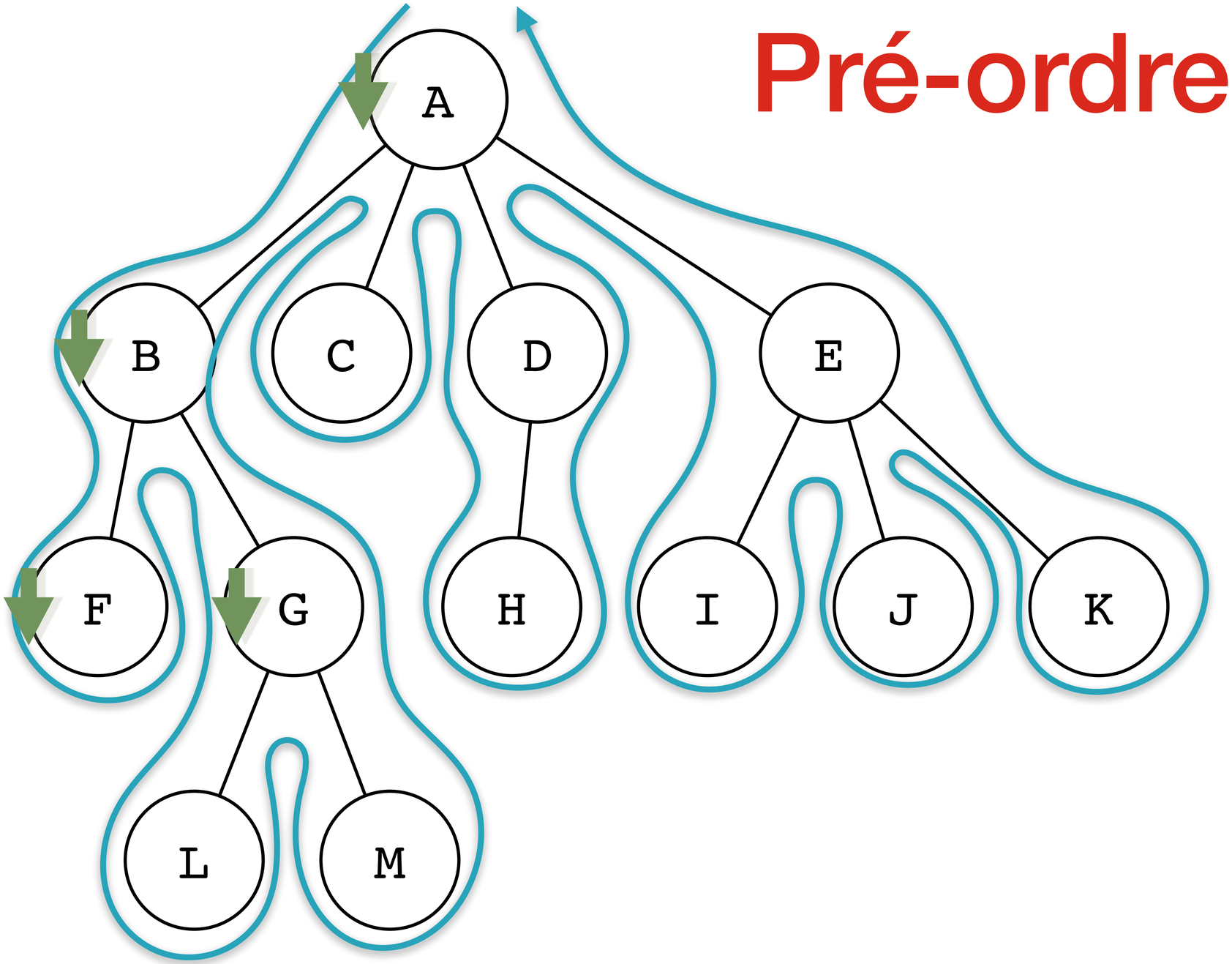
Pré-ordre



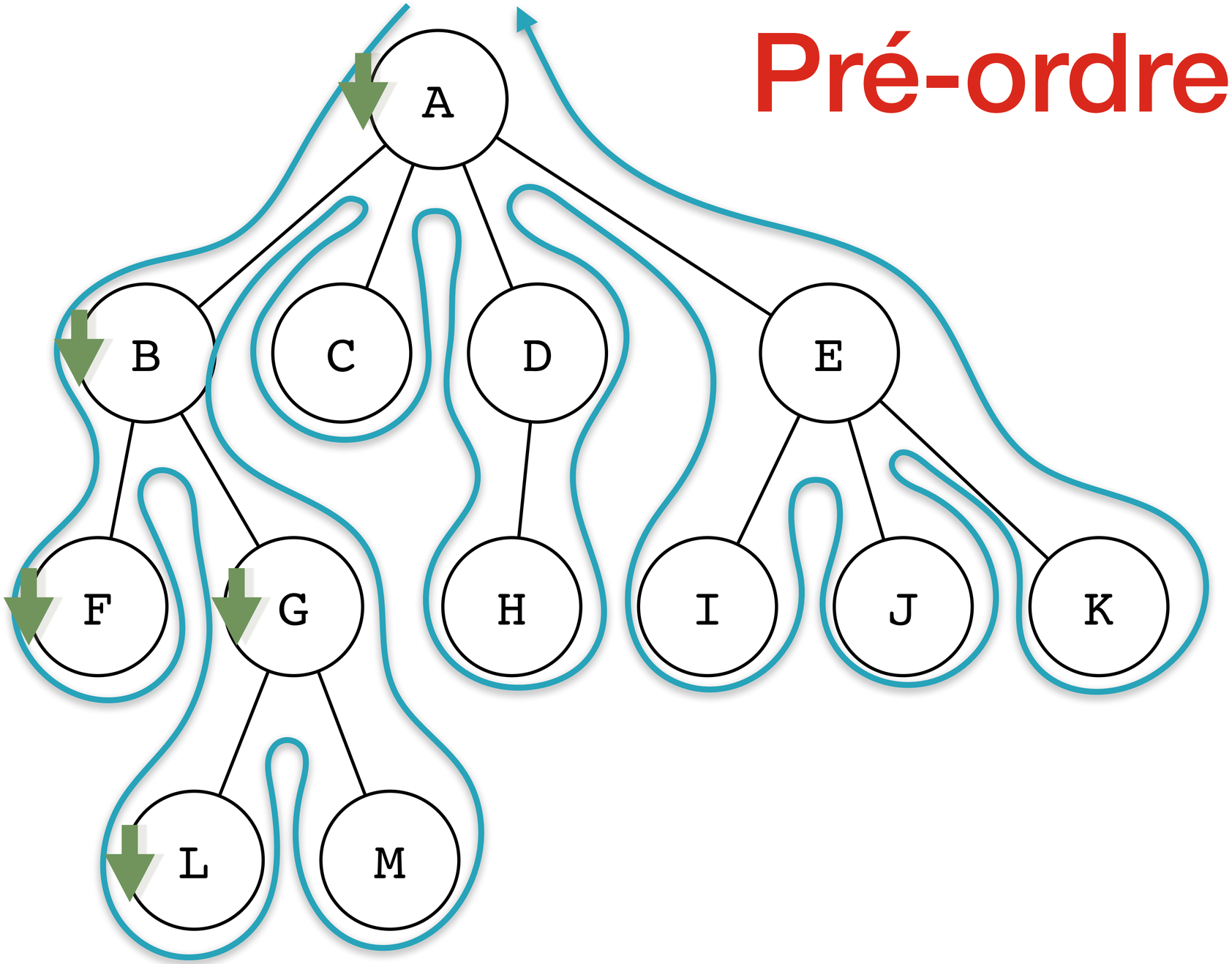
Pré-ordre



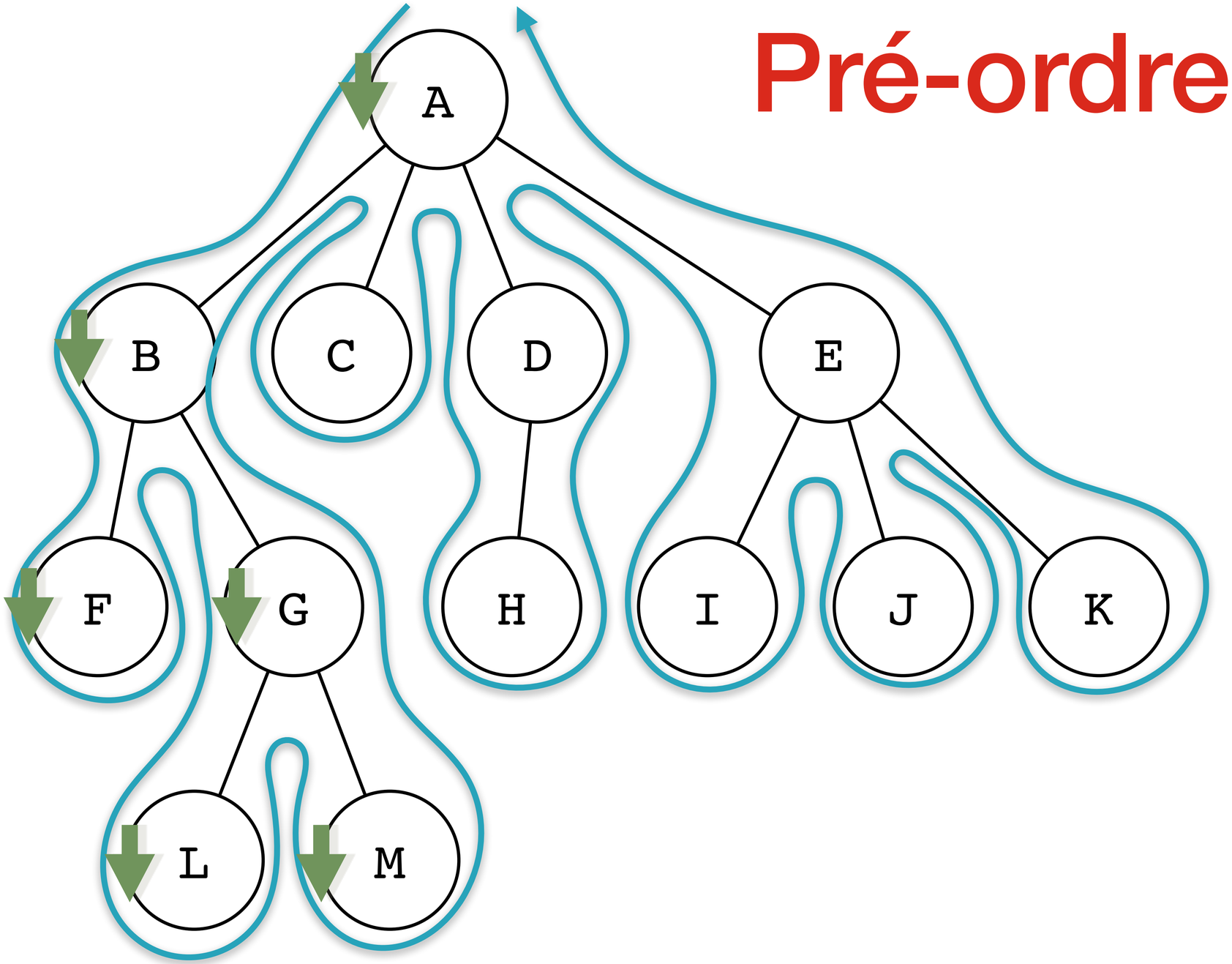
Pré-ordre



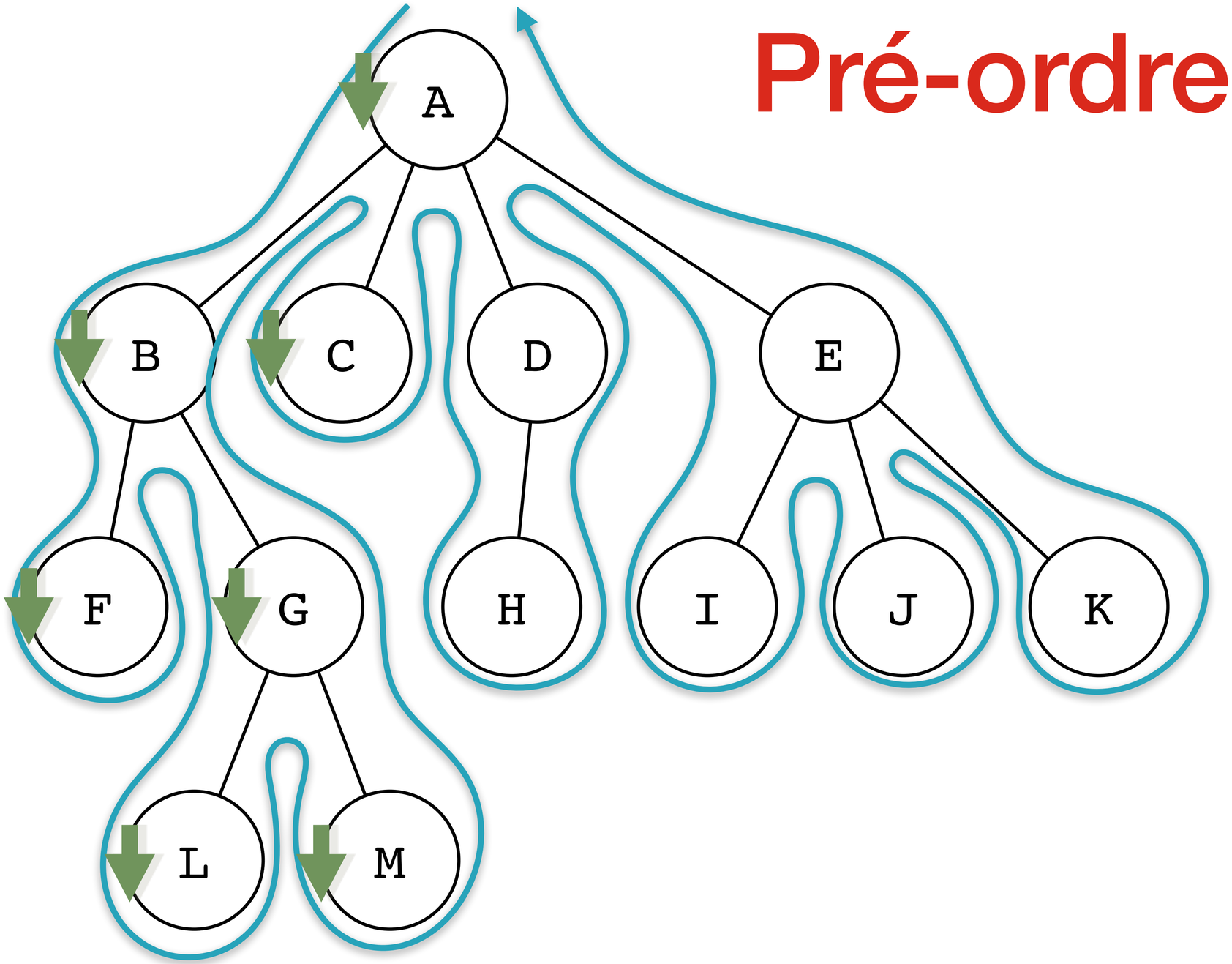
Pré-ordre



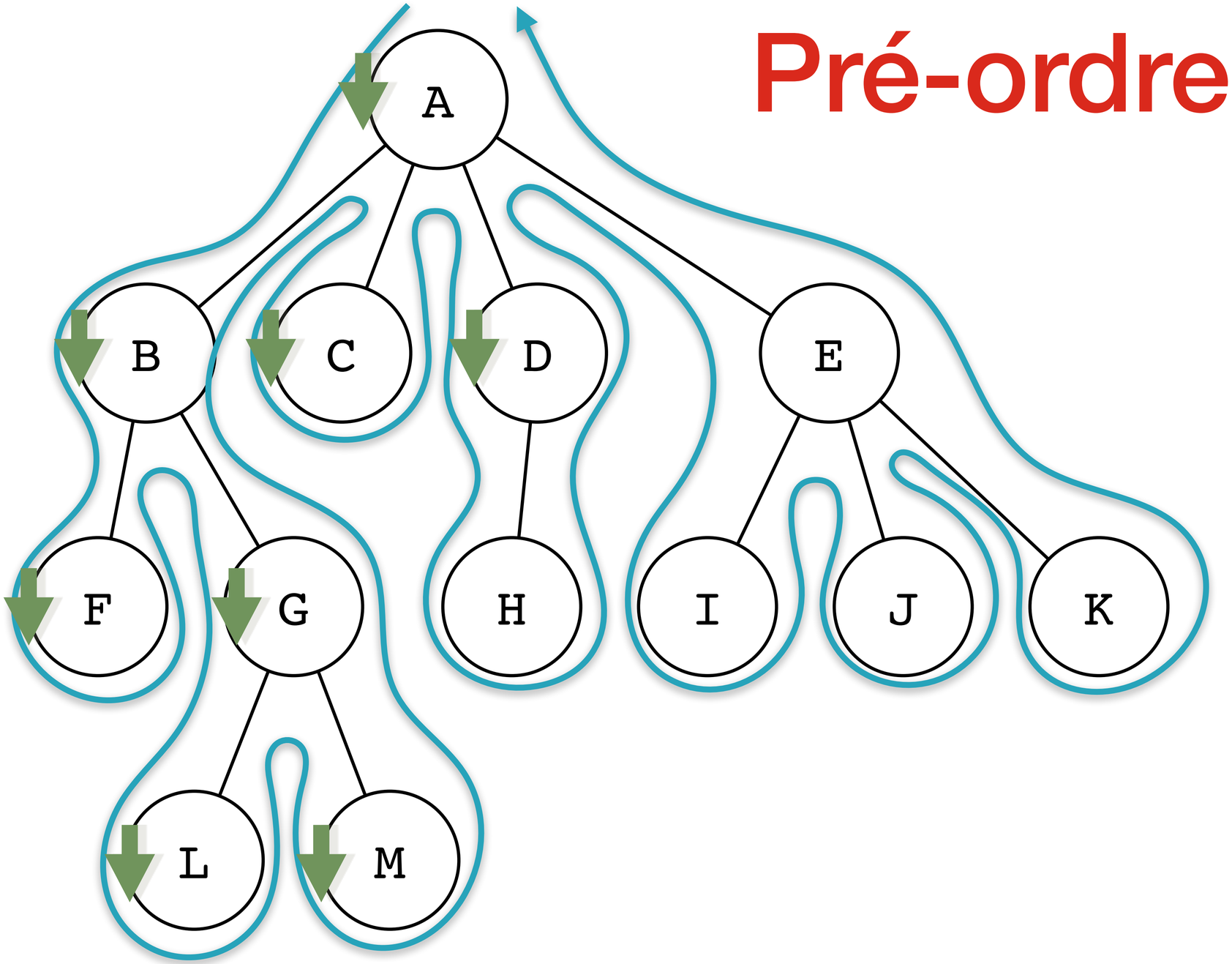
Pré-ordre



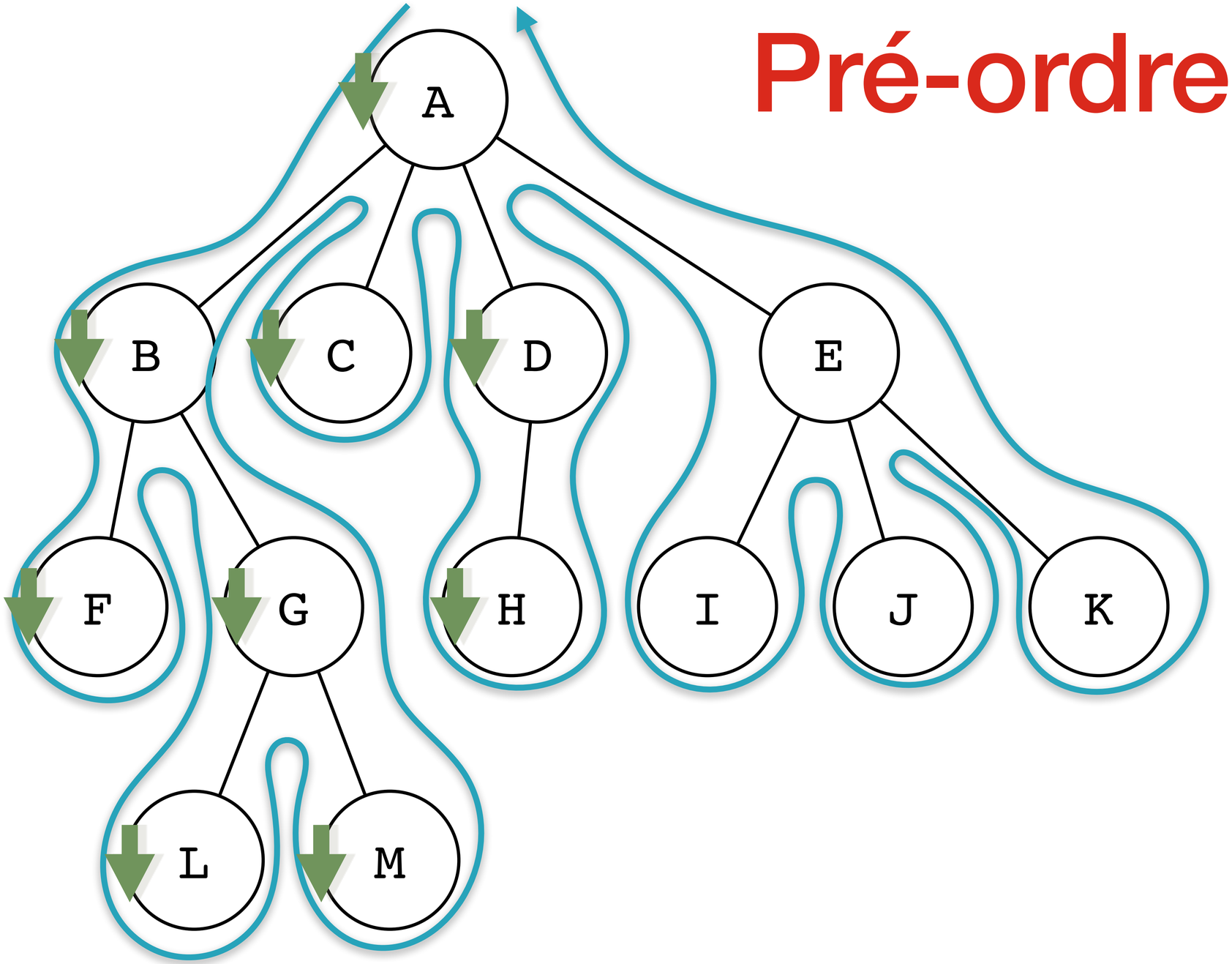
Pré-ordre



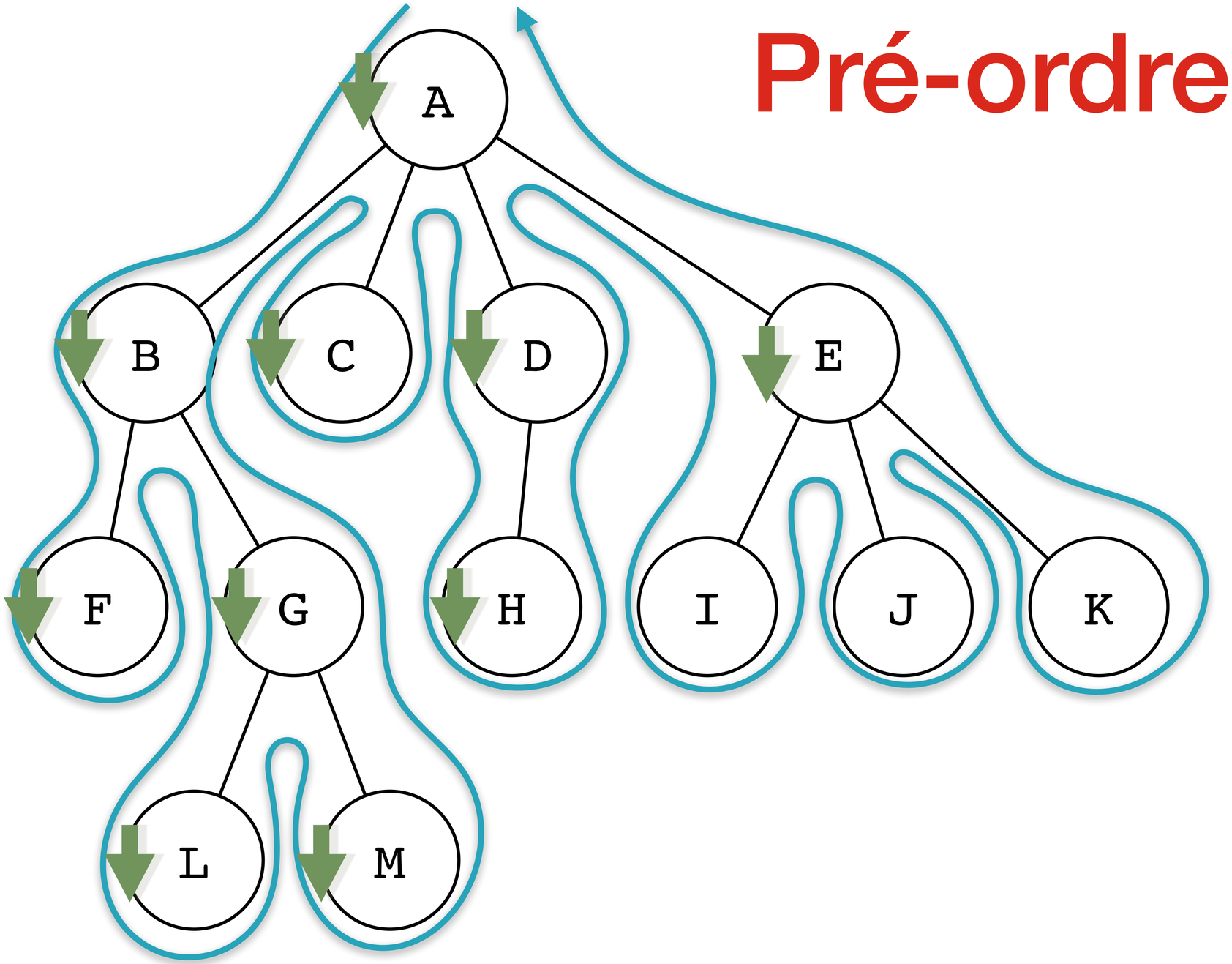
Pré-ordre



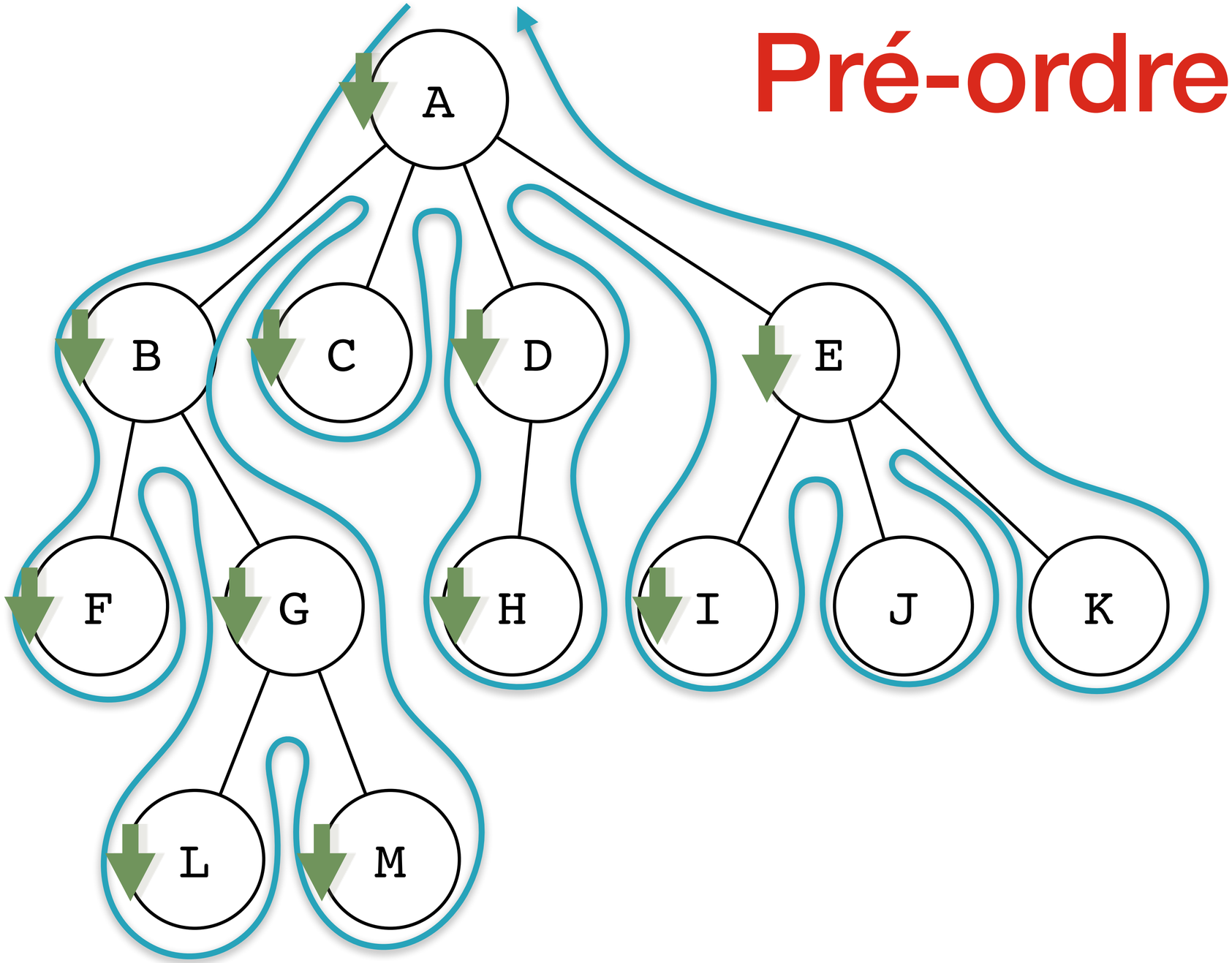
Pré-ordre



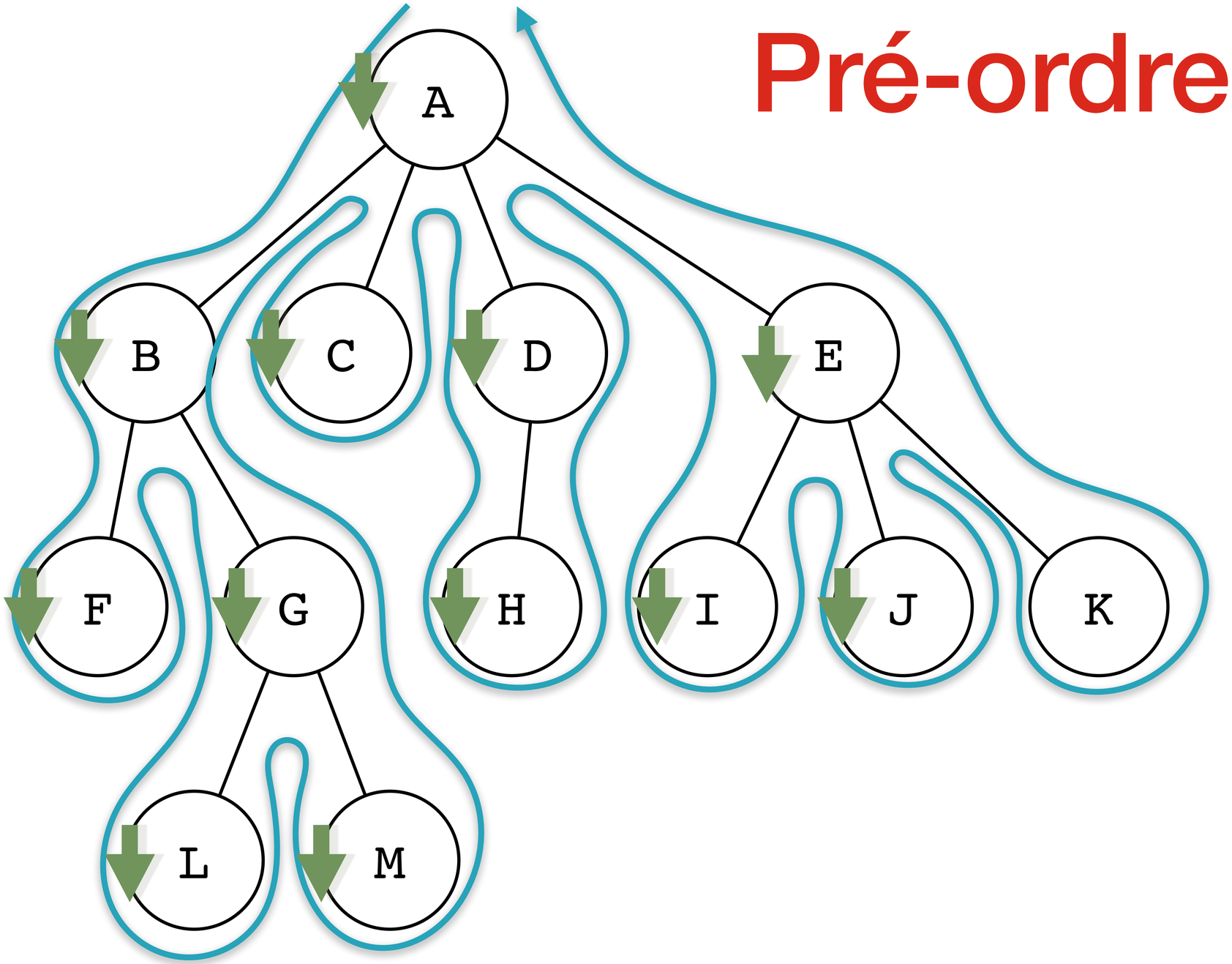
Pré-ordre



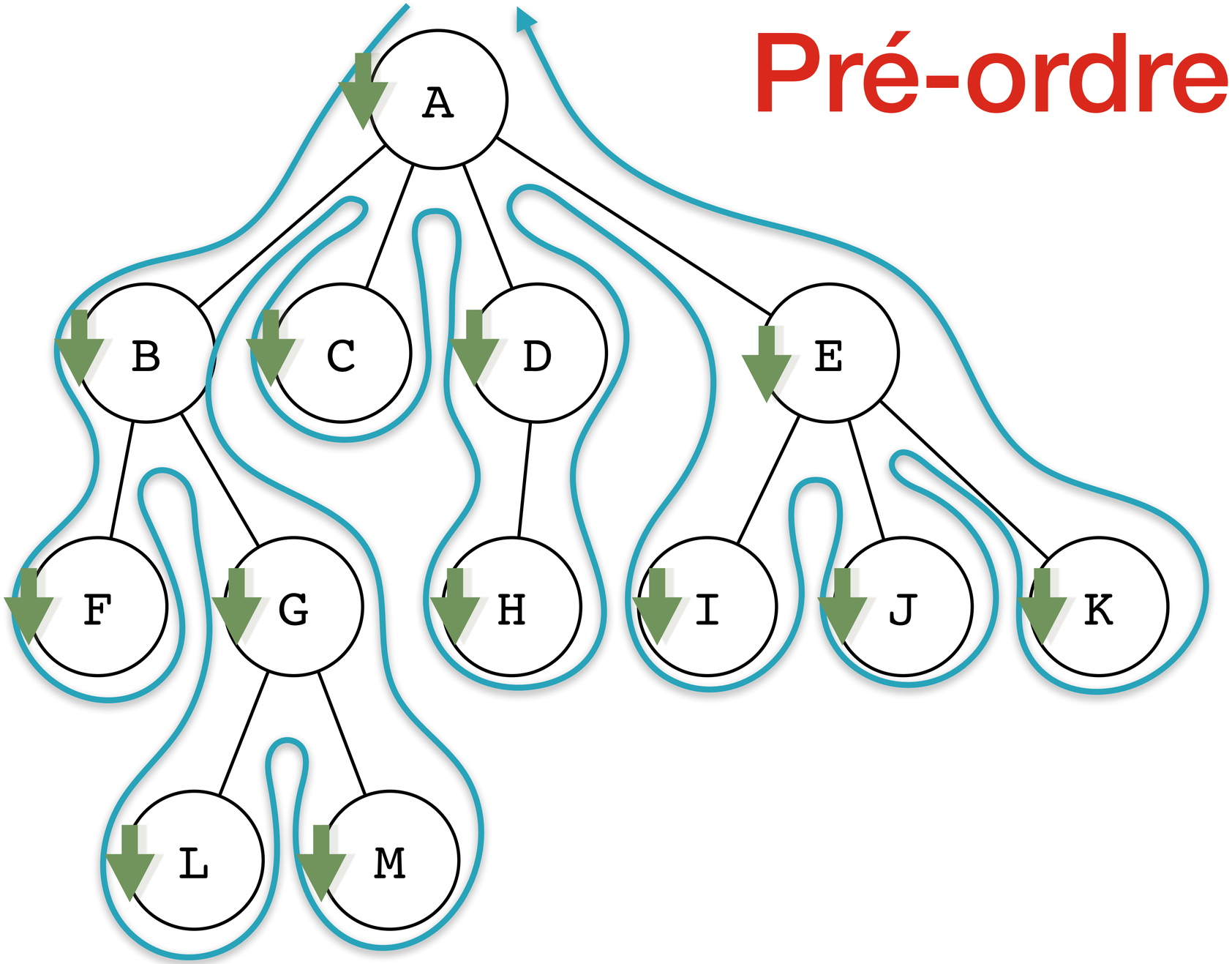
Pré-ordre



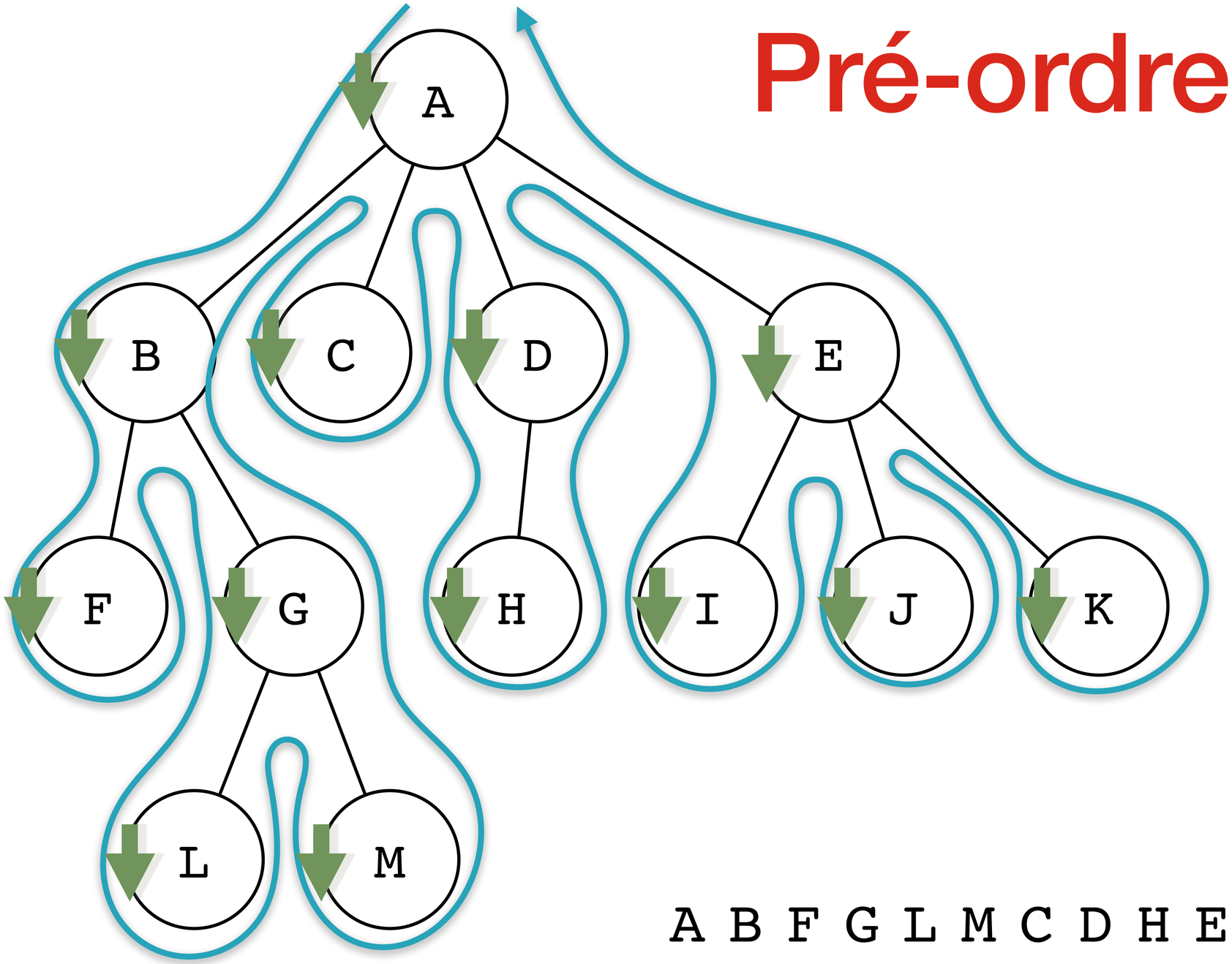
Pré-ordre



Pré-ordre

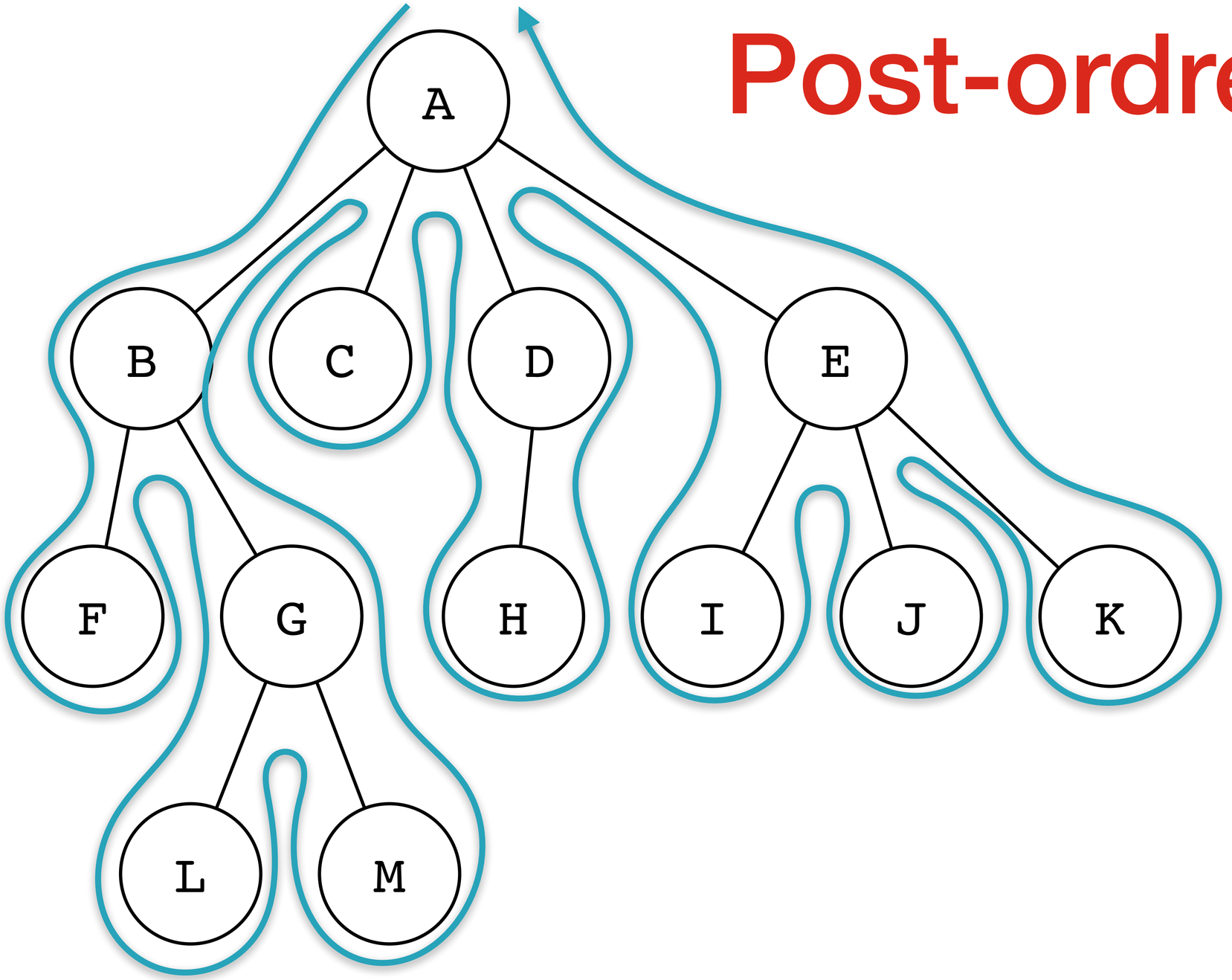


Pré-ordre

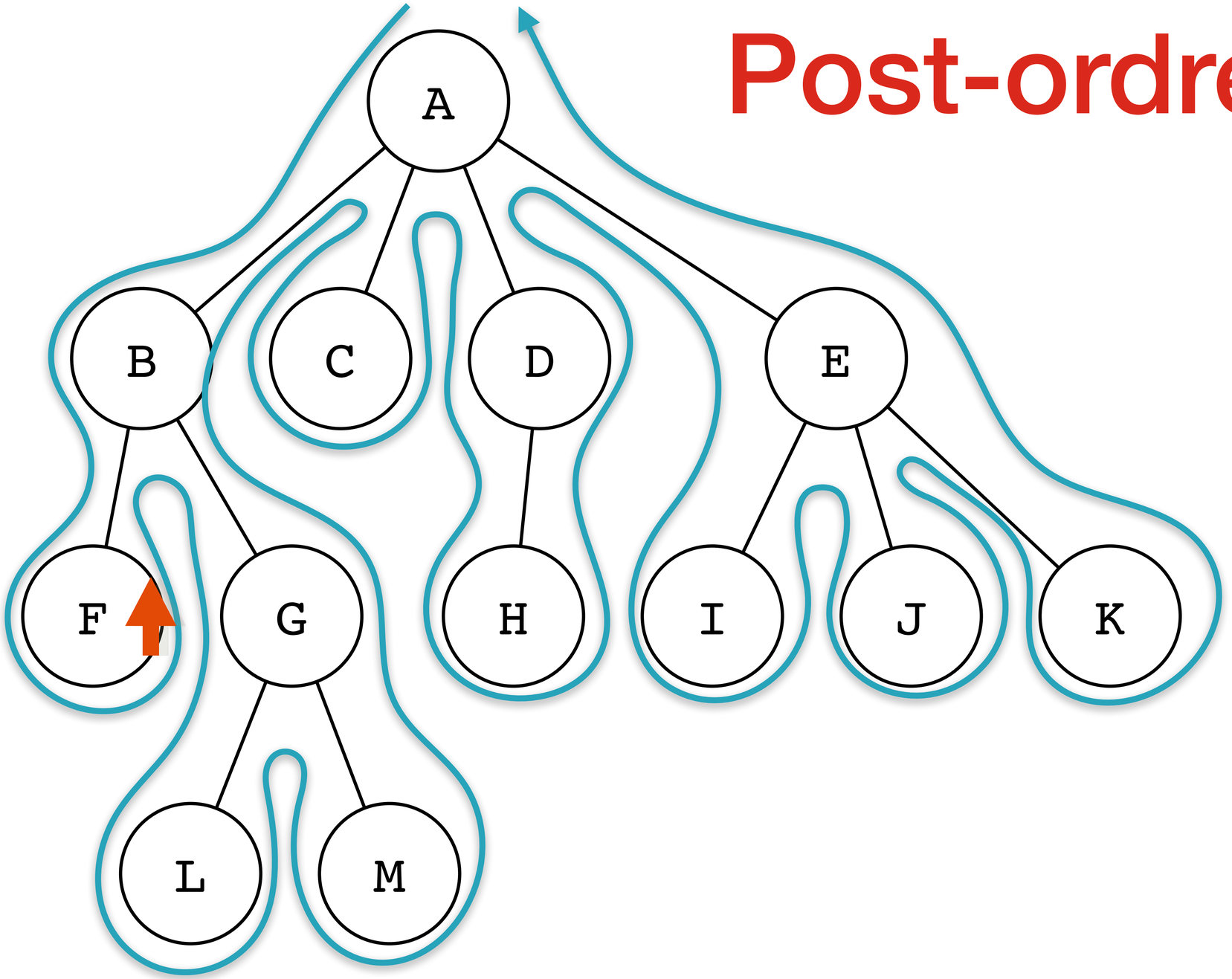


A B F G L M C D H E I J K

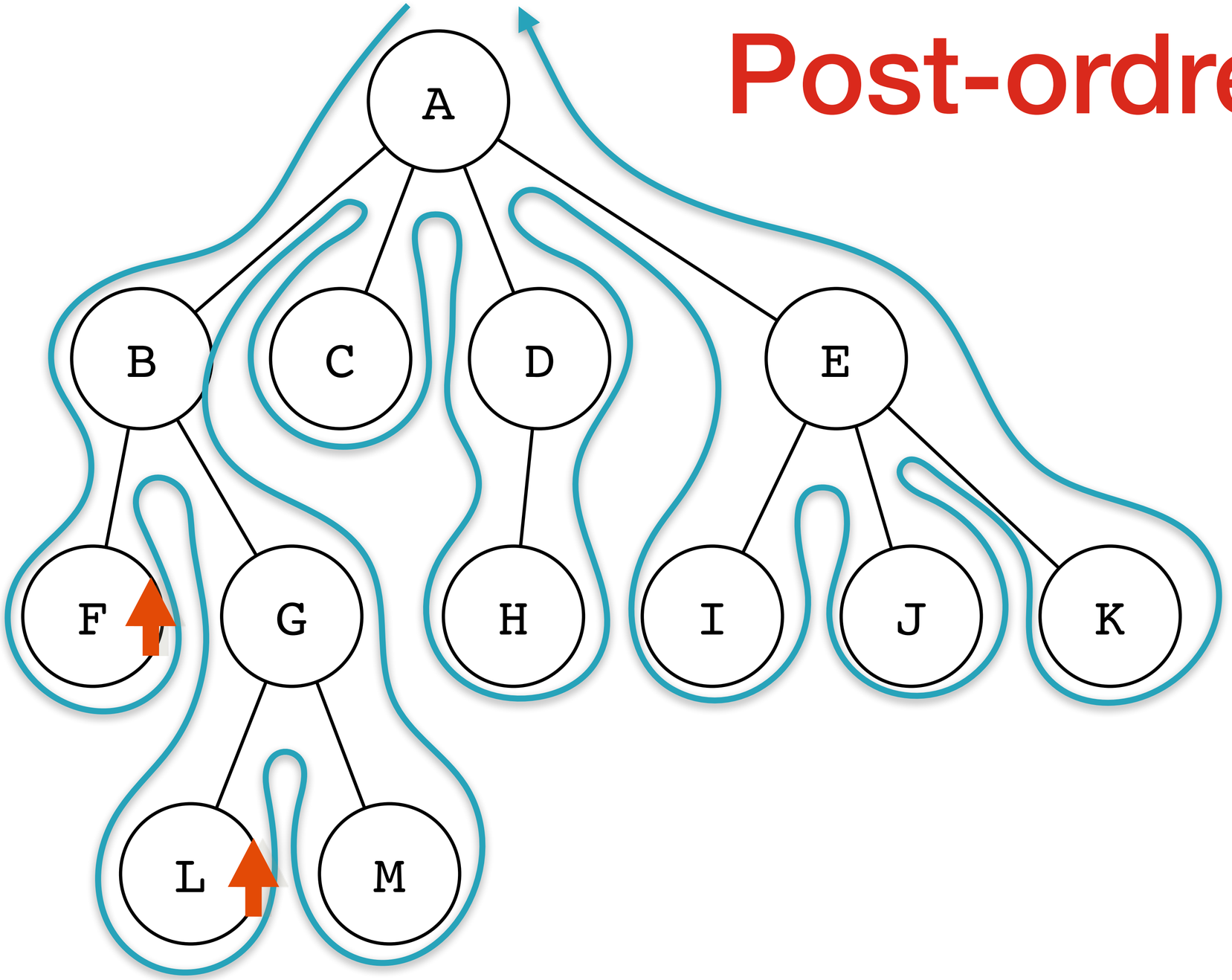
Post-ordre



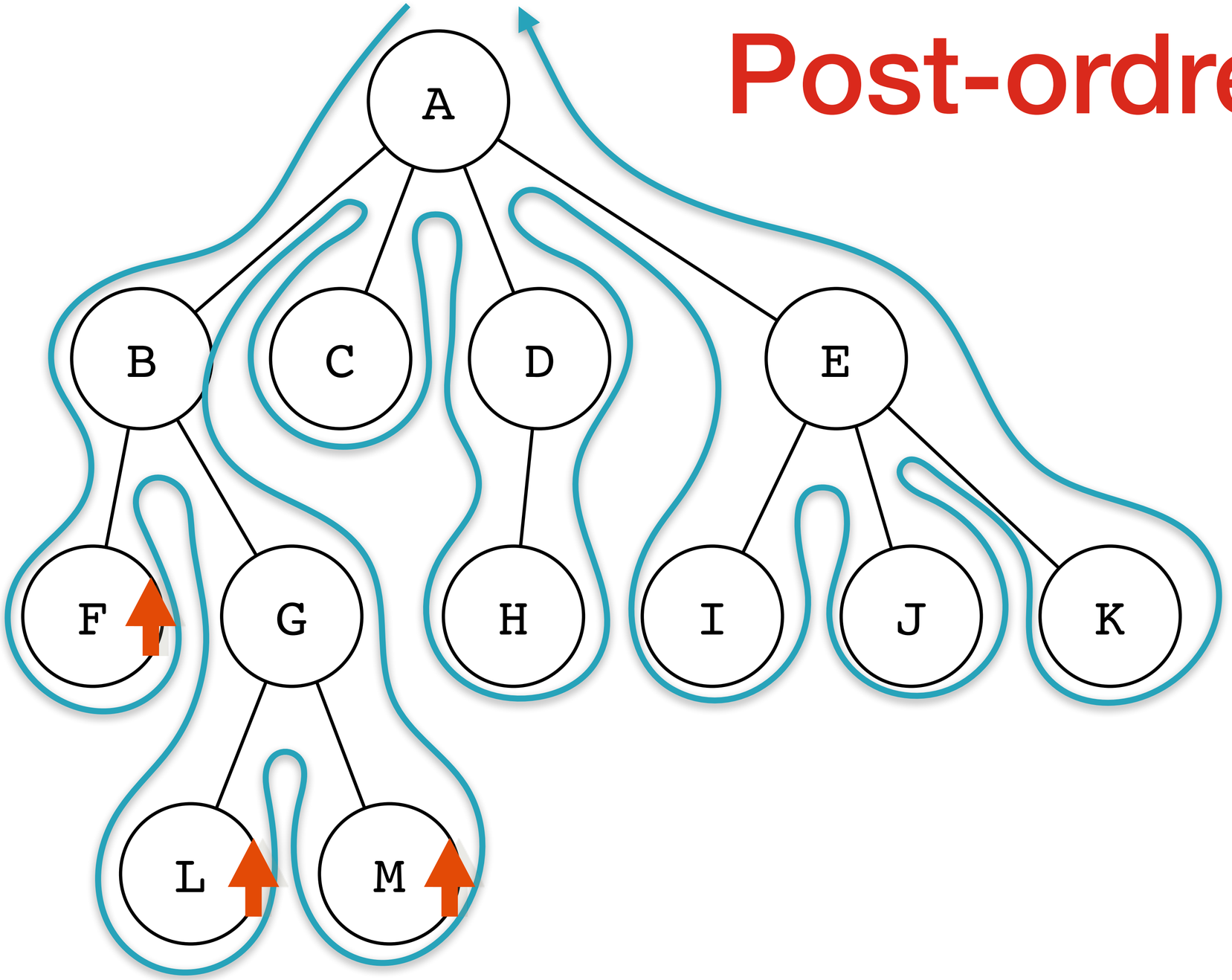
Post-ordre



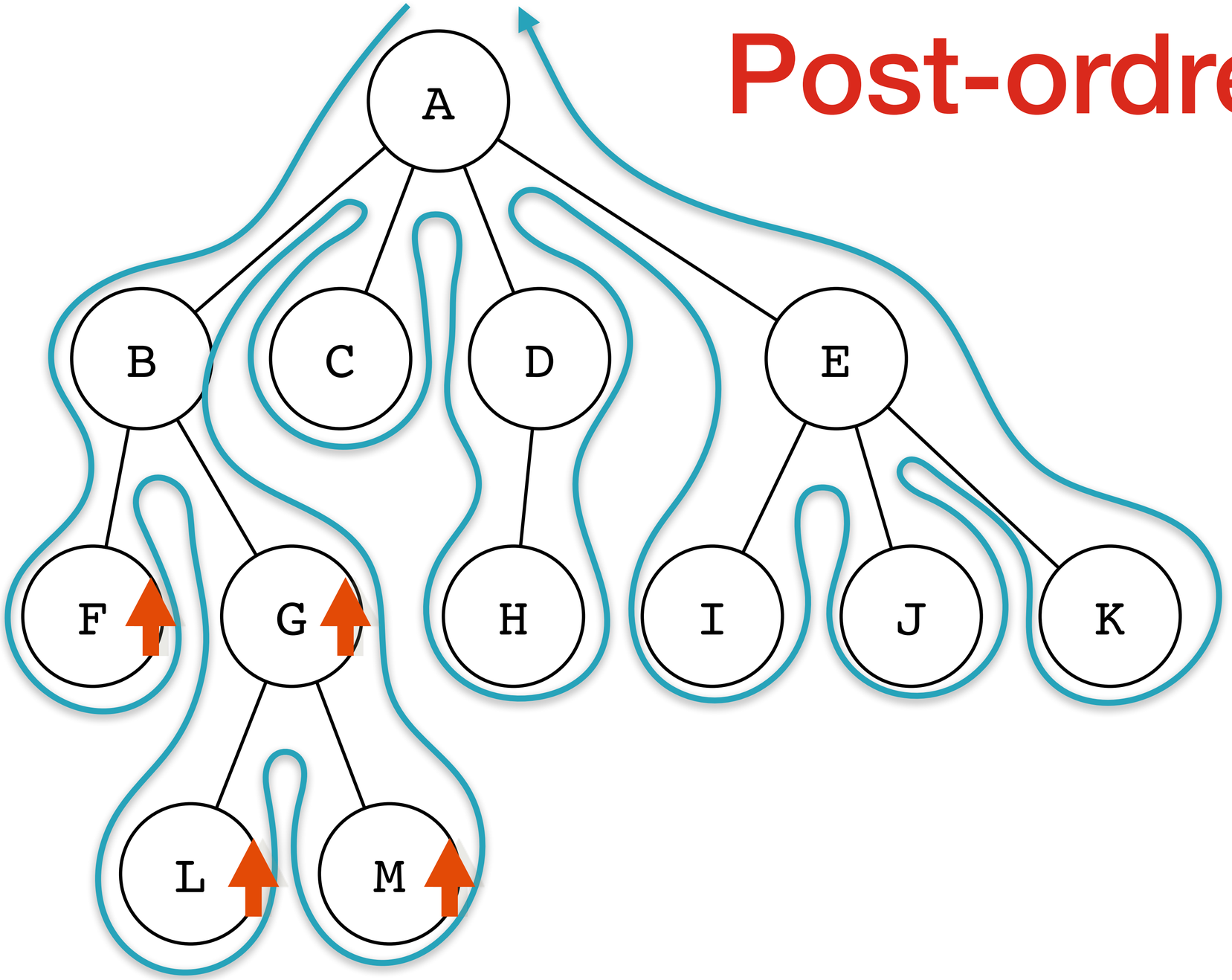
Post-ordre



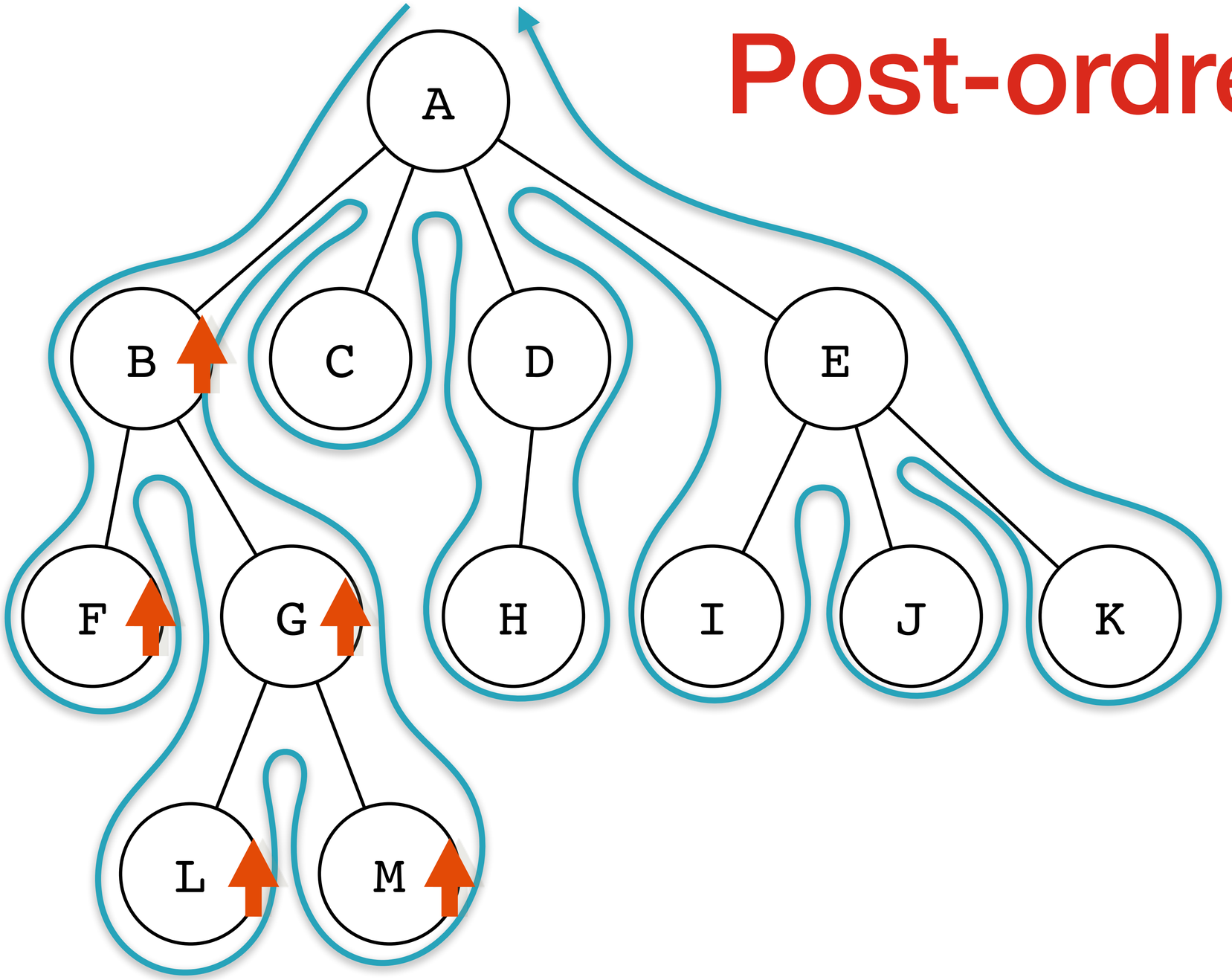
Post-ordre



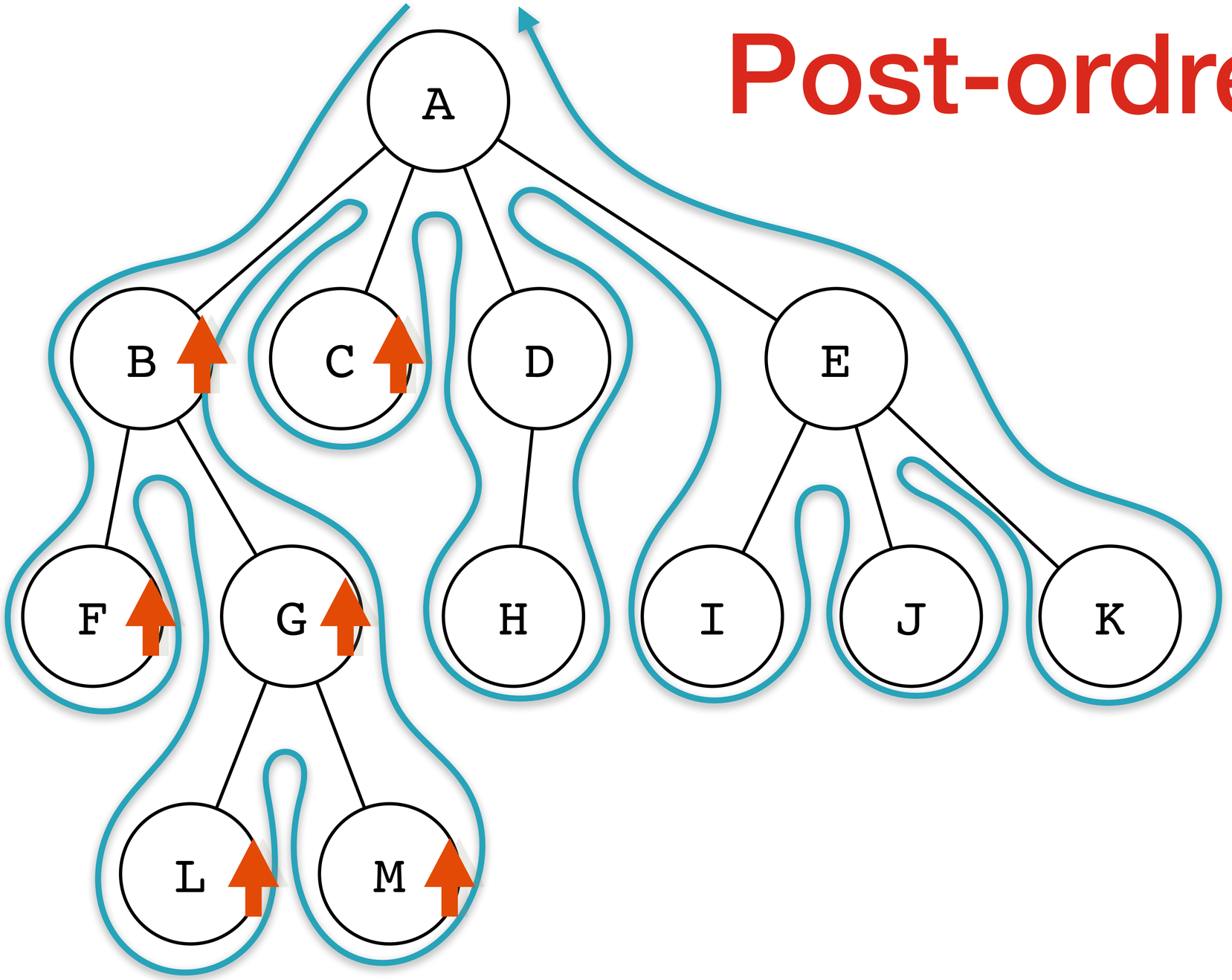
Post-ordre



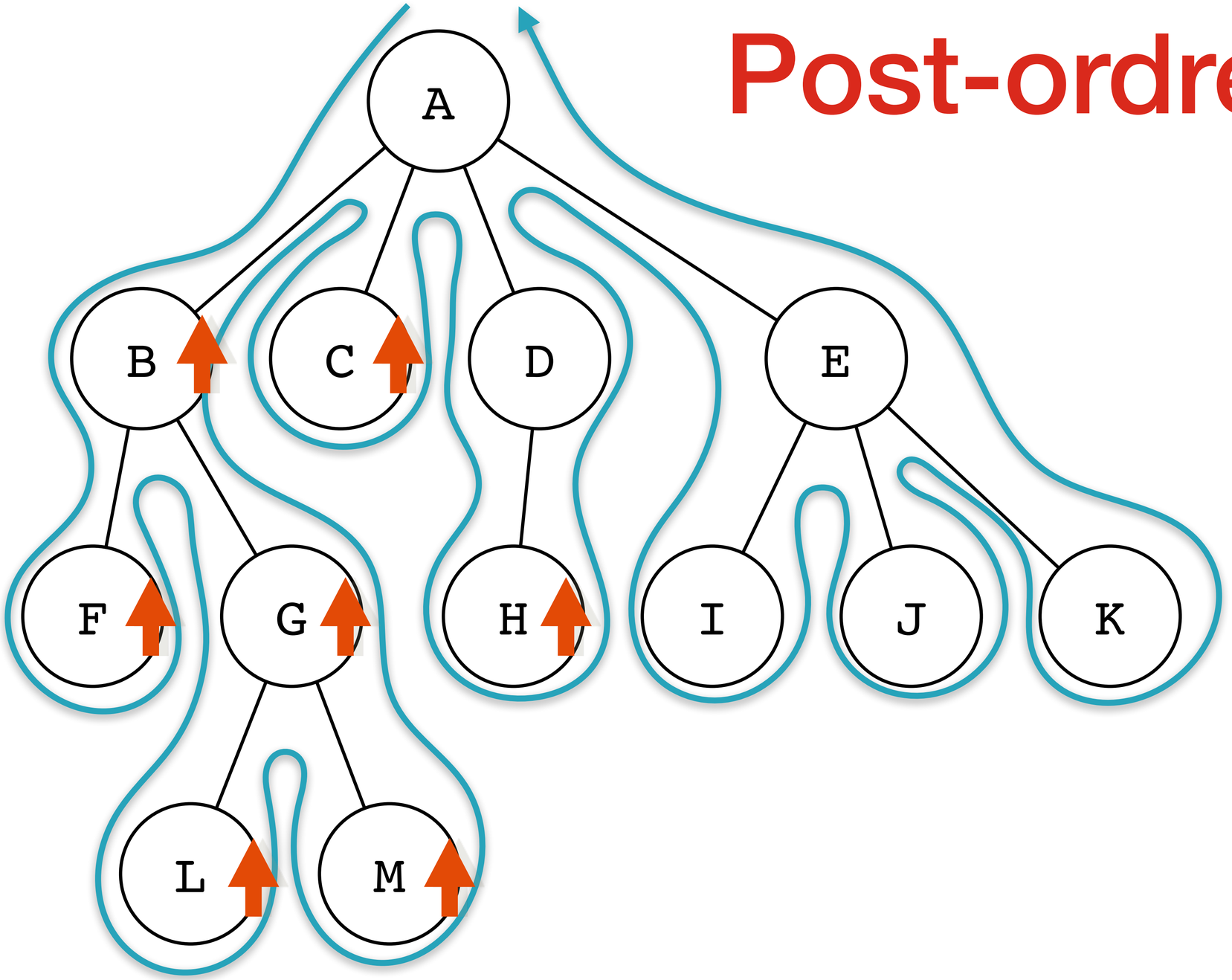
Post-ordre



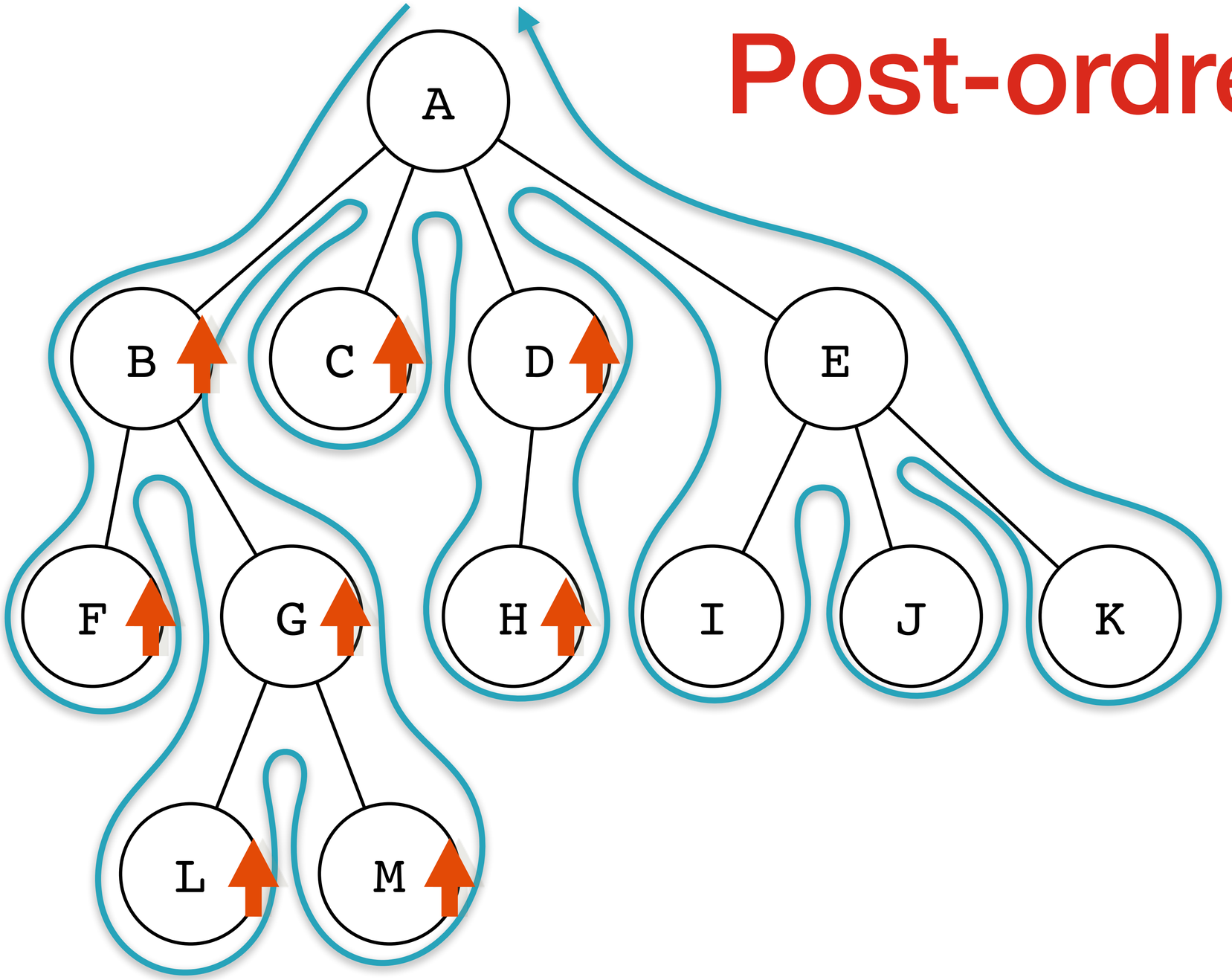
Post-ordre



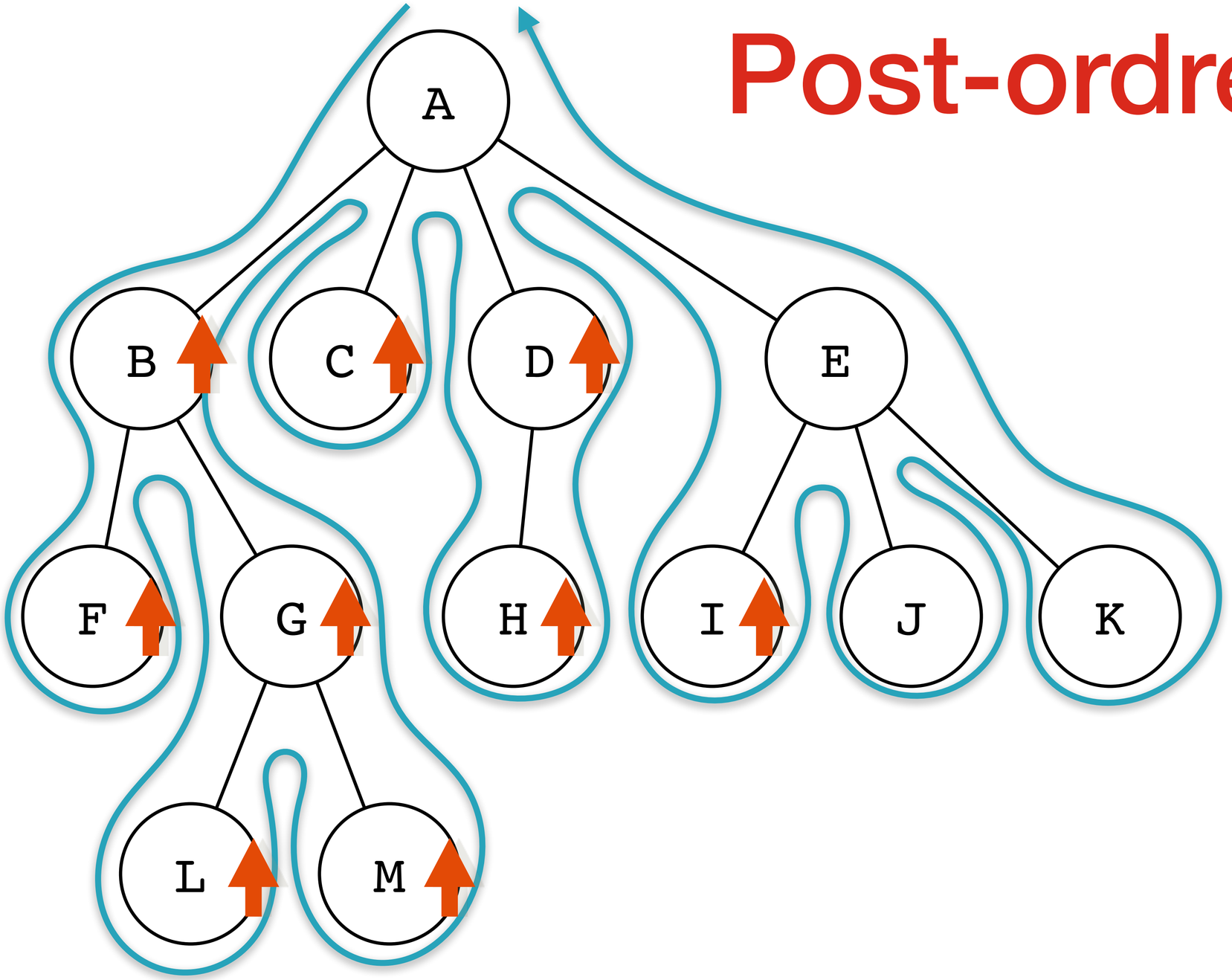
Post-ordre



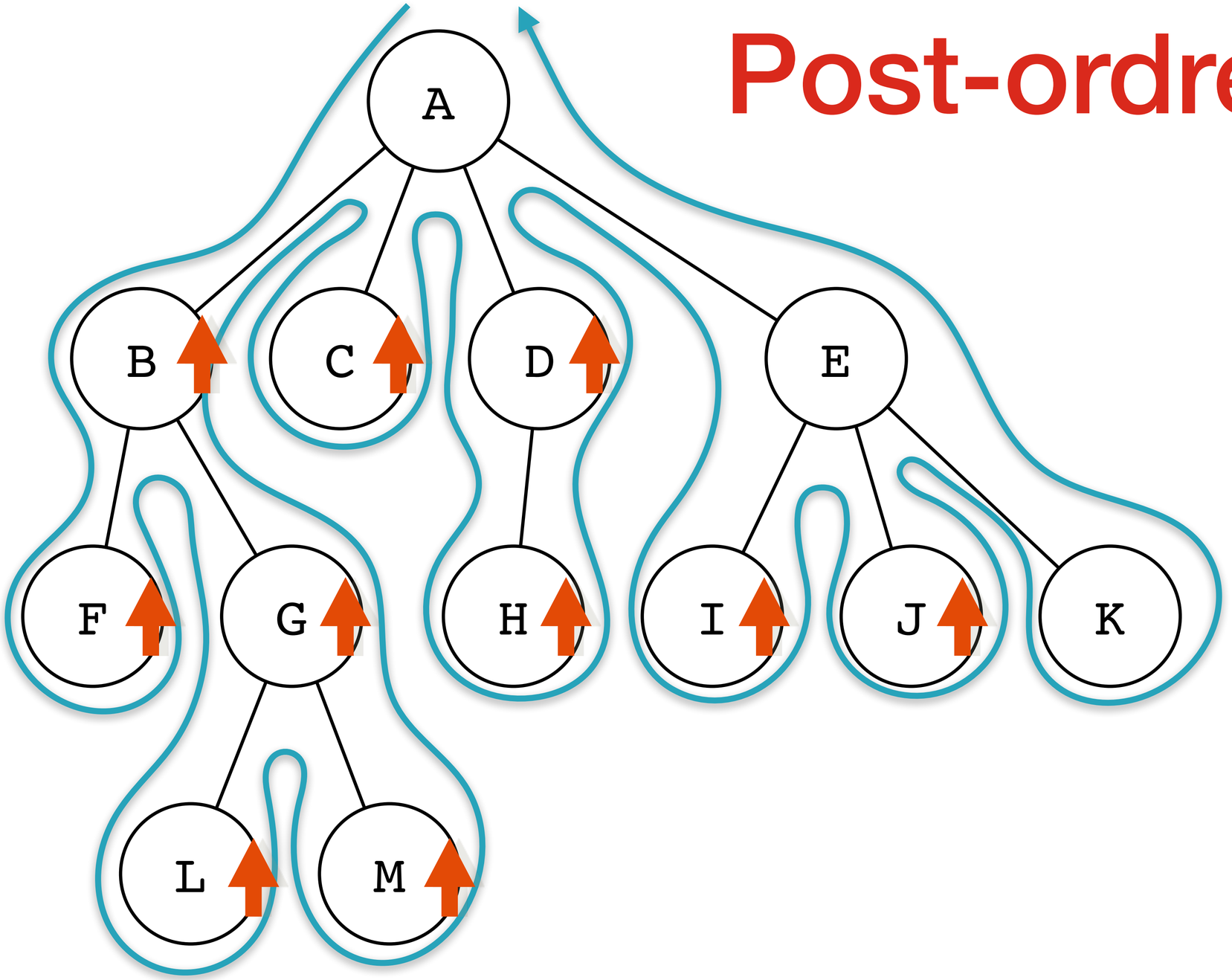
Post-ordre



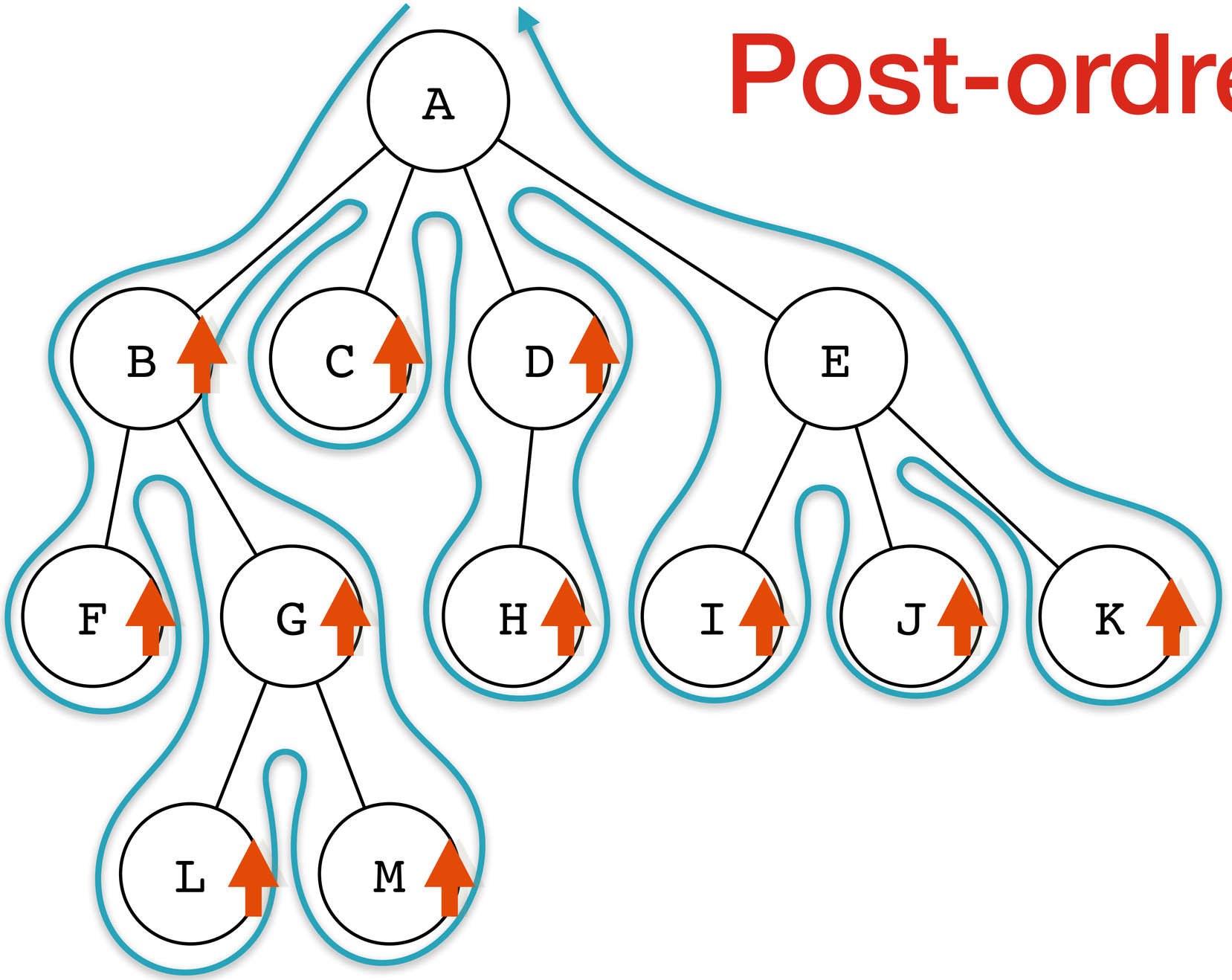
Post-ordre



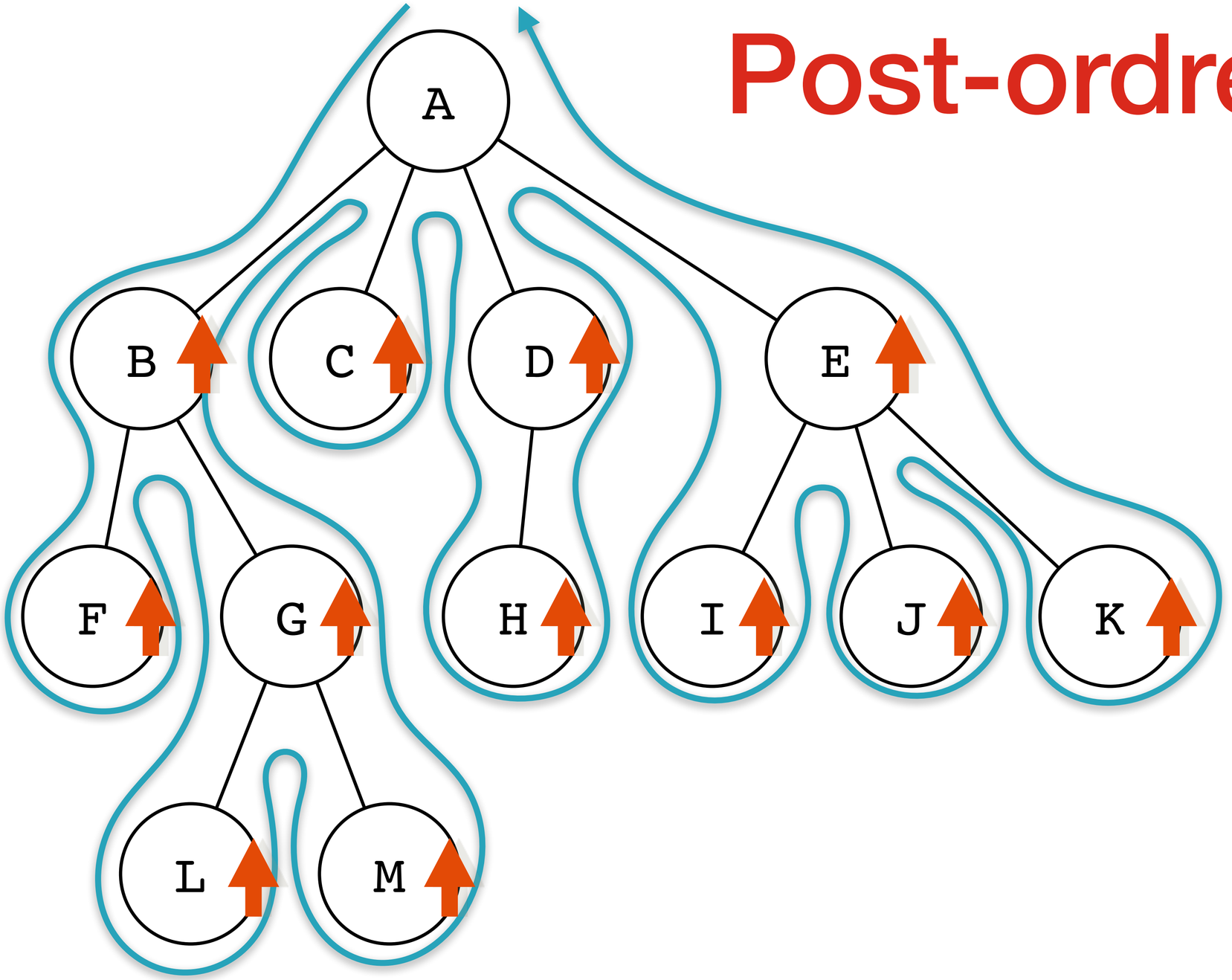
Post-ordre



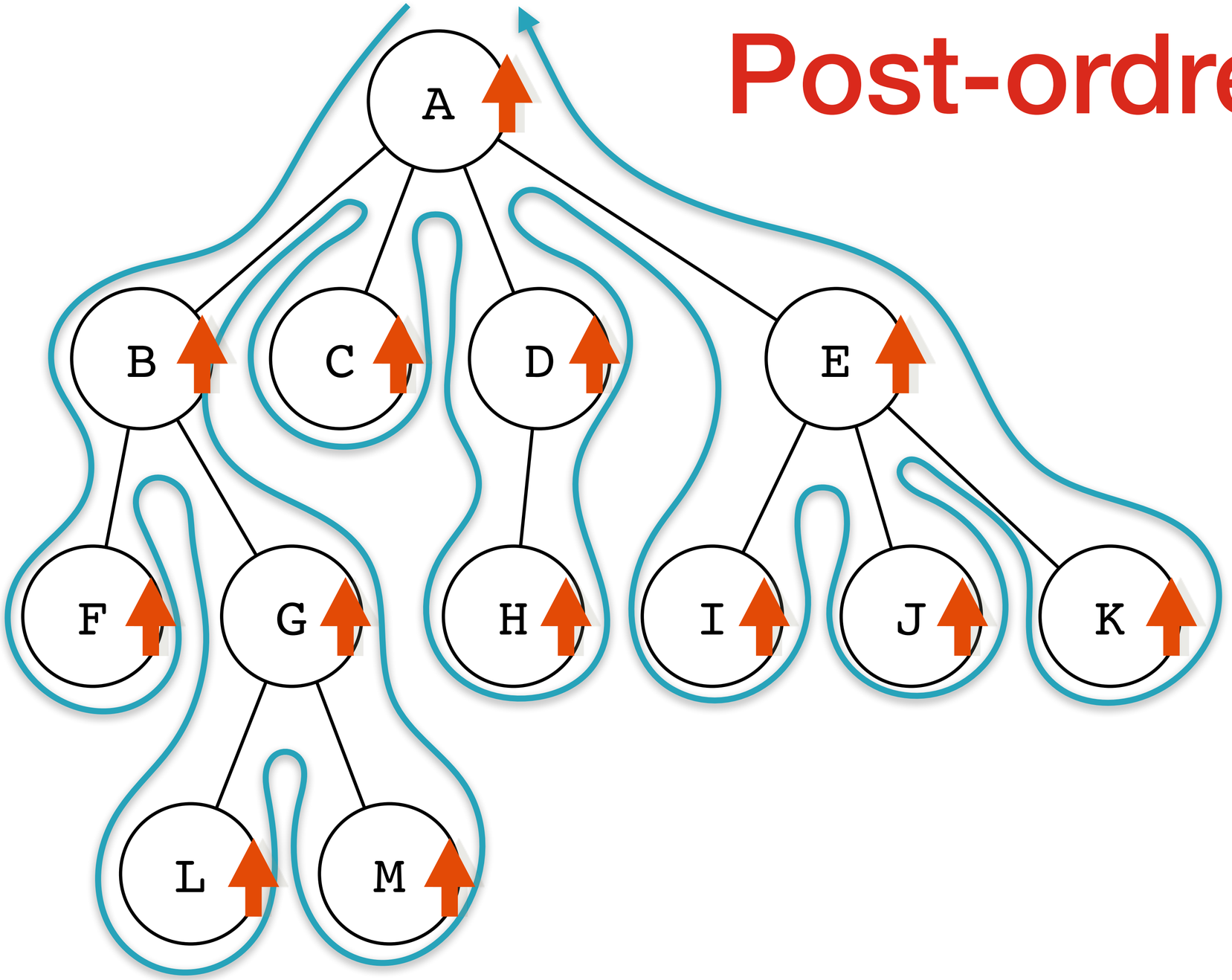
Post-ordre



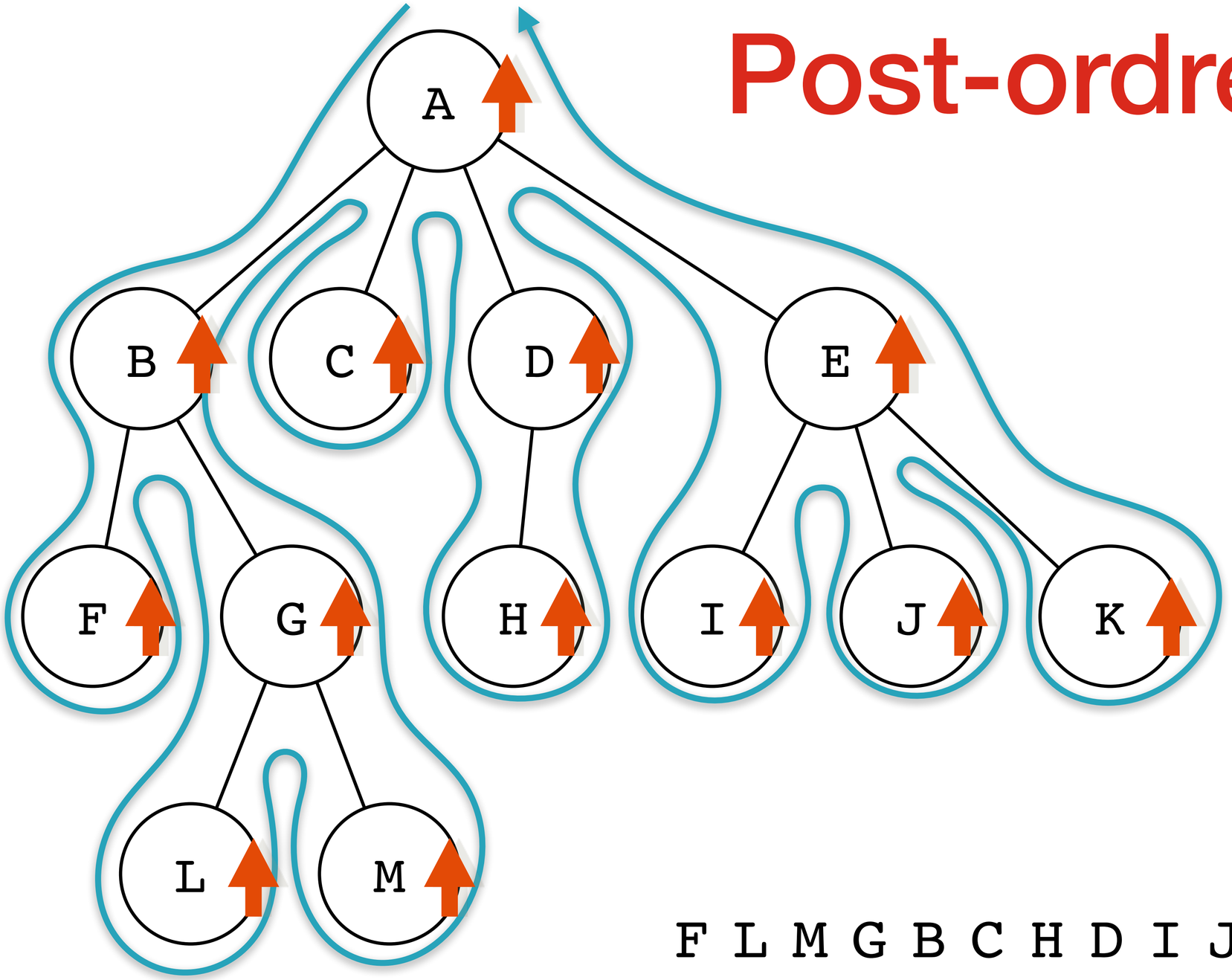
Post-ordre



Post-ordre



Post-ordre



F L M G B C H D I J K E A

Applications

- Copier en pré-ordre

```
fonction copier (r)
  si r != ∅
    r2 ← nouveau noeud d'étiquette r.étiquette
    pour tout enfant e de r
      e2 ← copier(e)
      Ajouter e2 aux enfants de r2
  retourner r2
```

- Détruire en post-ordre

Applications

- Copier en pré-ordre

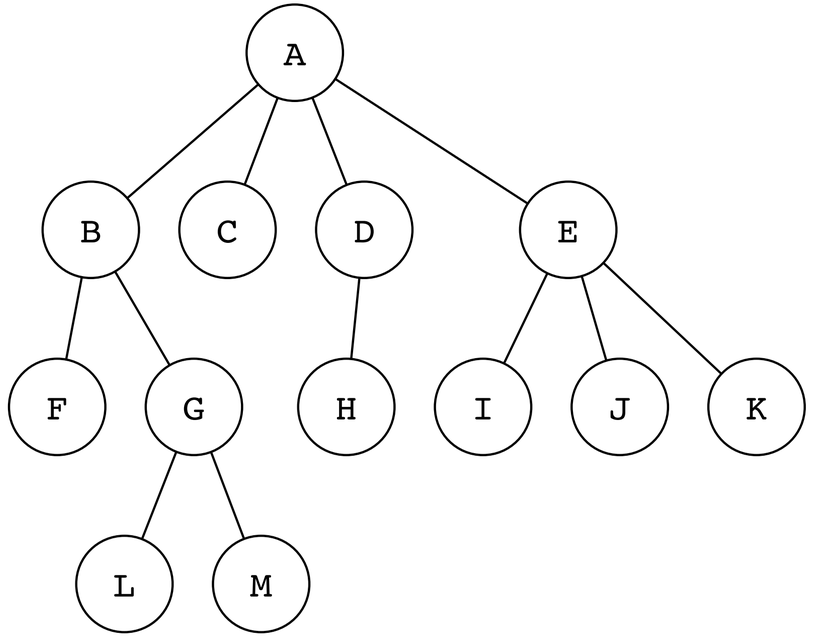
```
fonction copier (r)
  si r != ∅
    r2 ← nouveau noeud d'étiquette r.étiquette
    pour tout enfant e de r
      e2 ← copier(e)
      Ajouter e2 aux enfants de r2
  retourner r2
```

- Détruire en post-ordre

```
fonction détruire (ref r)
  si r != ∅
    pour tout enfant e de r
      détruire(e)
    effacer le noeud r
```

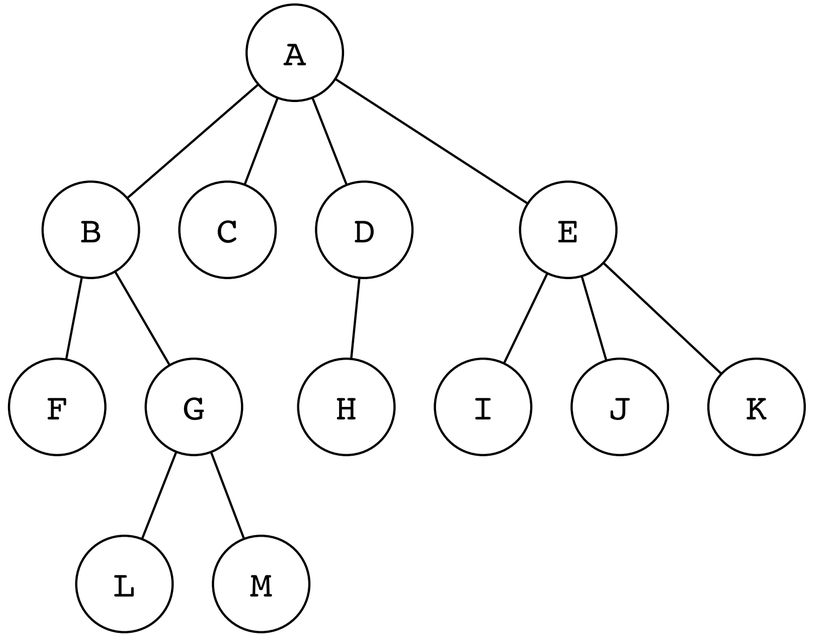
Parcours en largeur

- Visiter les noeuds d'un arbre niveau par niveau
- Comment savoir que le noeud F suit le noeud E?
- S'en souvenir depuis que l'on a visité son parent B
- Utiliser une file FIFO



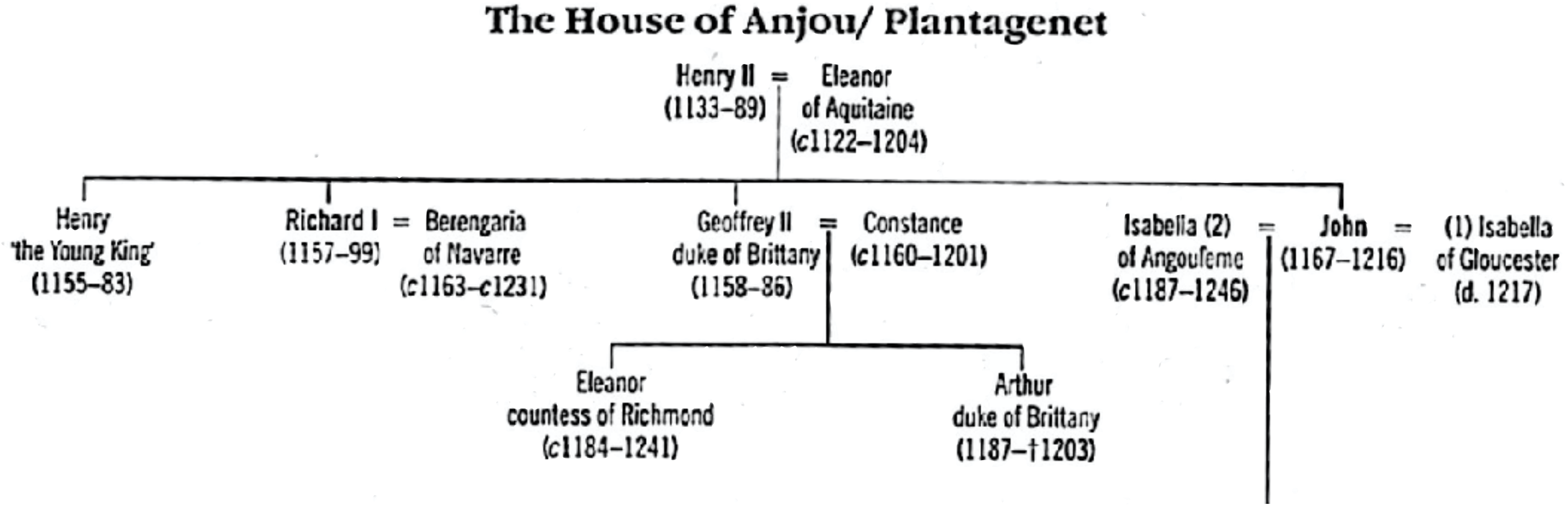
Parcours en largeur

- Visiter les noeuds d'un arbre niveau par niveau
- Comment savoir que le noeud F suit le noeud E?
- S'en souvenir depuis que l'on a visité son parent B
- Utiliser une file FIFO

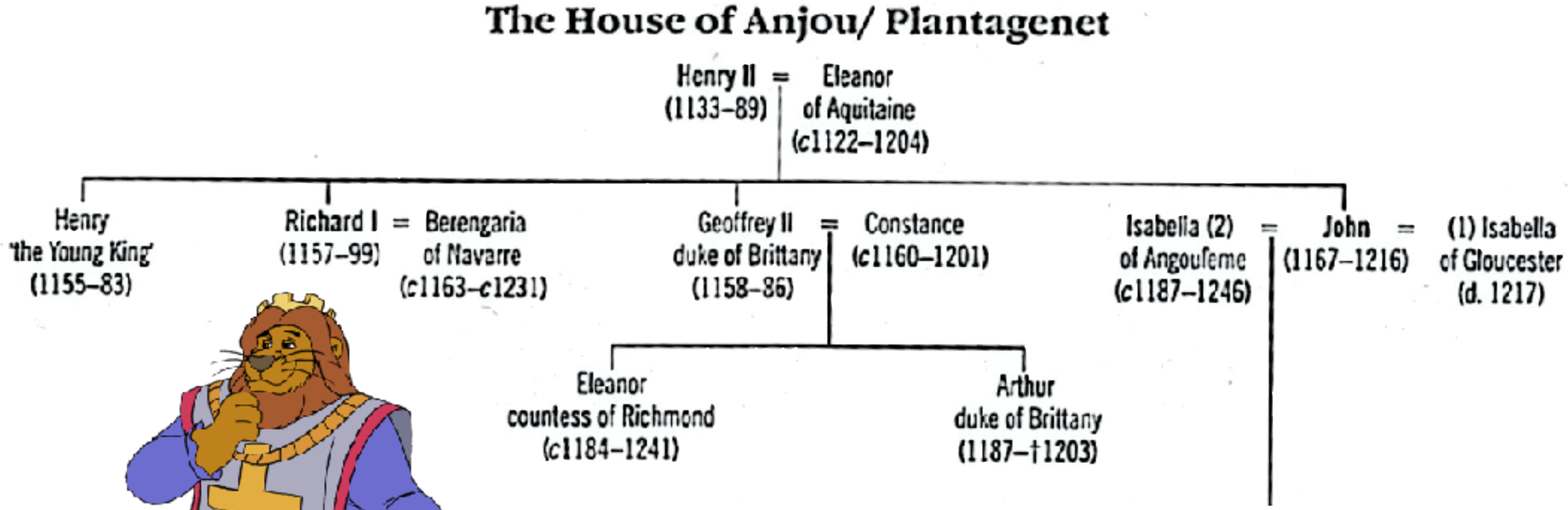



```
fonction largeur (r, fn)
  Initialiser File Q
  Q.push(r)
  tant que pas Q.vide()
    r = Q.front(); Q.pop()
    fn(r)
    pour tout enfant e de r
      Q.push(e)
```

Succession de Richard Coeur de Lion



Succession de Richard Coeur de Lion



Succession de Richard Coeur de Lion

The House of Anjou/ Plantagenet

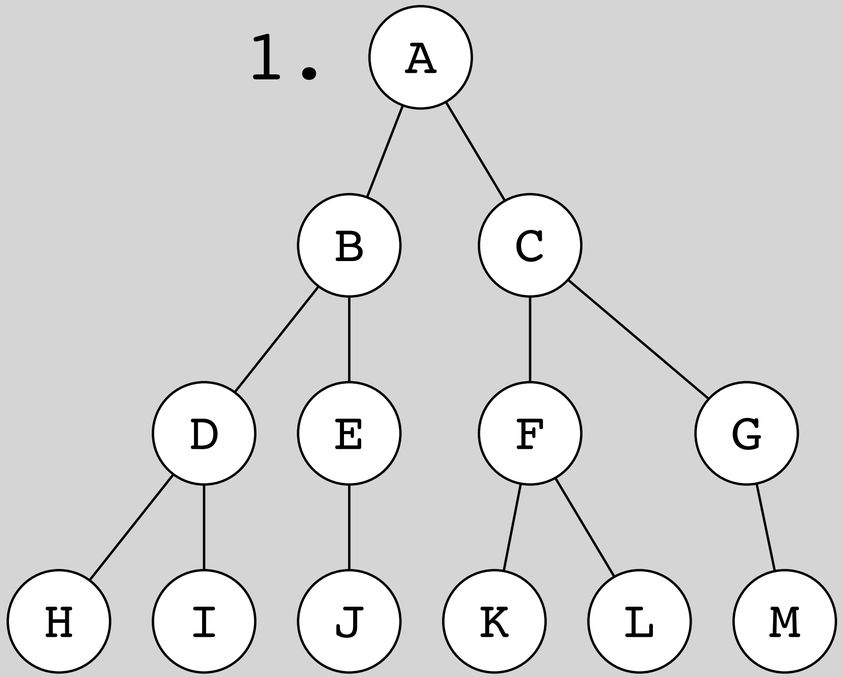
- Henry II (1133–89) = Eleanor of Aquitaine (c1122–1204)
  - Henry 'the Young King' (1155–83)
  - Richard I (1157–99) = Berengaria of Navarre (c1163–c1231)
  - Geoffrey II duke of Brittany (1158–86) = Constance (c1160–1201)
    - Eleanor countess of Richmond (c1184–1241)
    - Arthur duke of Brittany (1187–†1203)
  - Isabella (2) of Angouleme (c1187–1246) = John (1167–1216)
  - (1) Isabella of Gloucester (d. 1217)

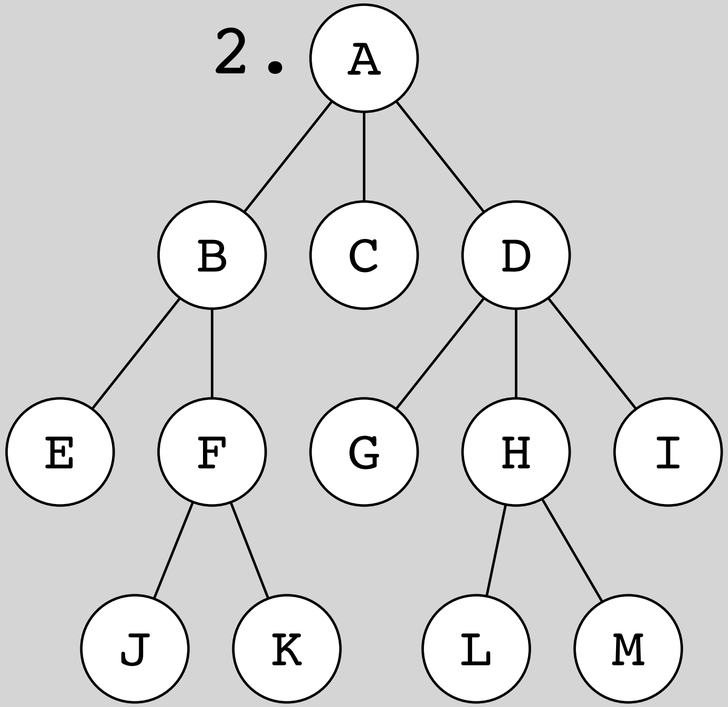
Succession de Richard Coeur de Lion

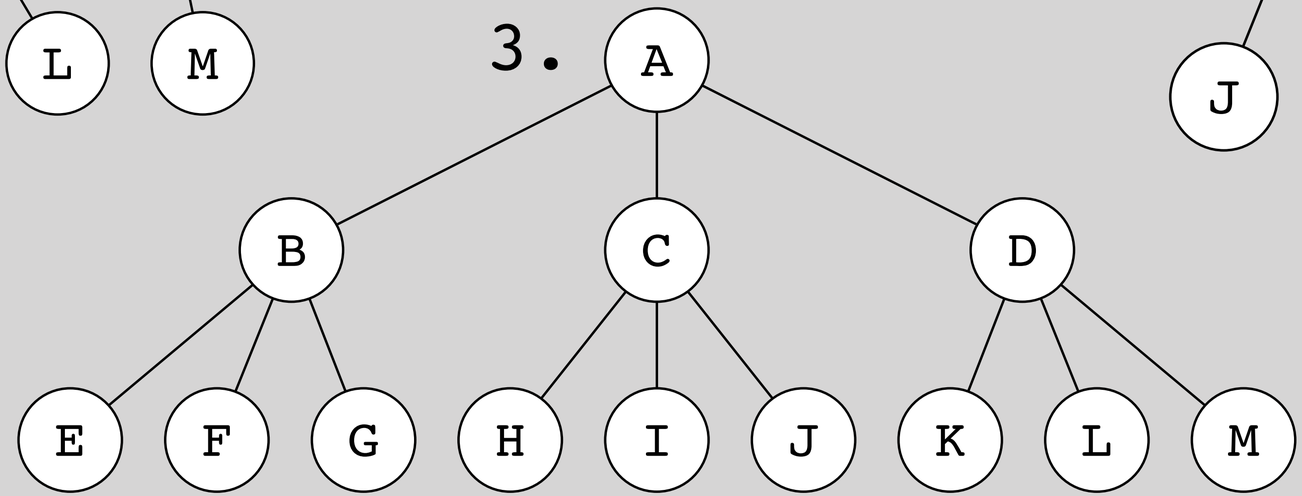
The House of Anjou/ Plantagenet

- Henry II (1133–89) = Eleanor of Aquitaine (c1122–1204)
  - Henry 'the Young King' (1155–83)
  - Richard I (1157–99) = Berengaria of Navarre (c1163–c1231)
  - Geoffrey II duke of Brittany (1158–86) = Constance (c1160–1201)
    - Eleanor countess of Richmond (c1184–1241)
    - Arthur duke of Brittany (1187–†1203)
  - Isabella (2) of Angouleme (c1187–1246) = John (1167–1216) = (1) Isabella of Gloucester (d. 1217)

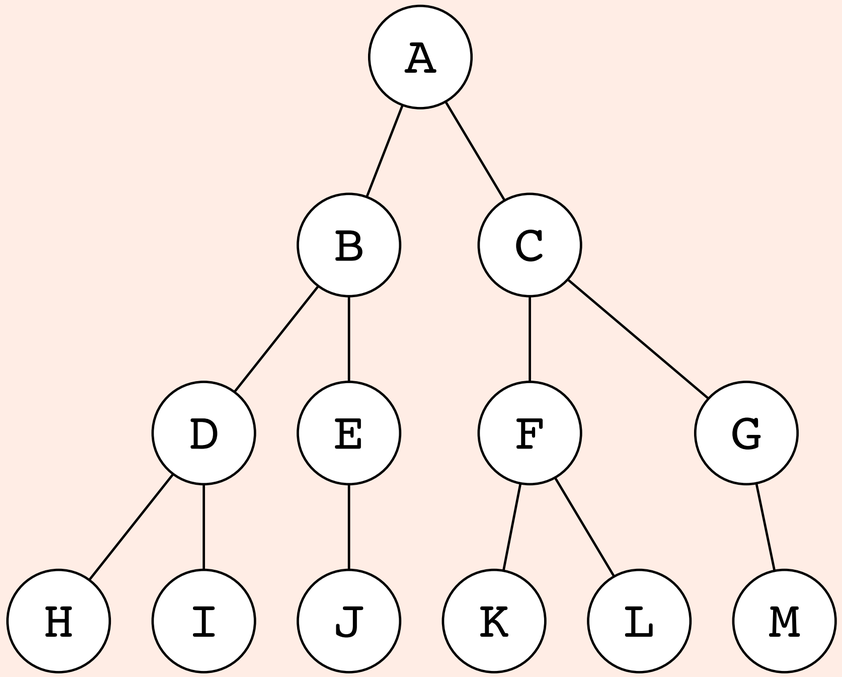
Effectuez les 3 parcours pour chaque arbre



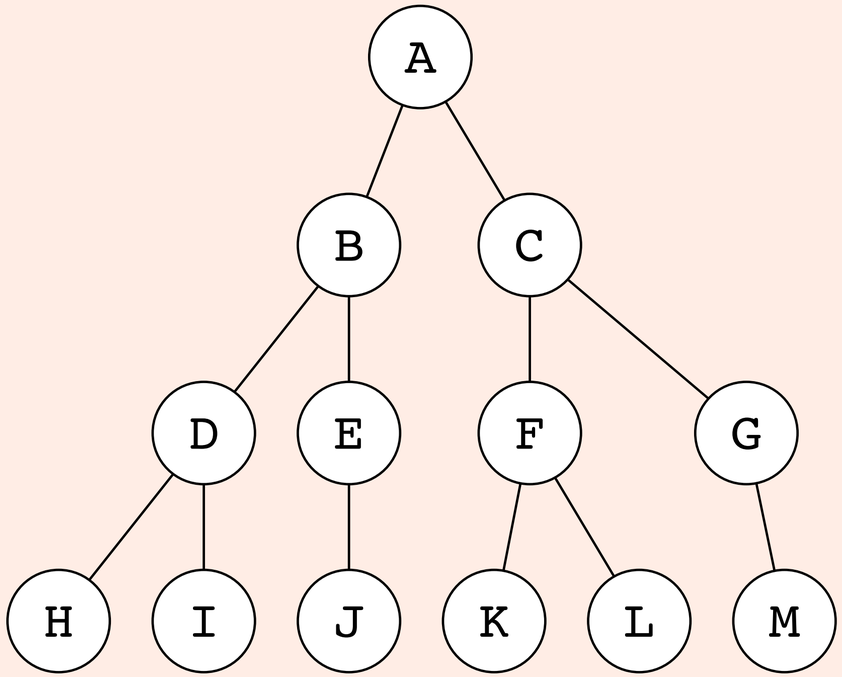




Parcours (1)

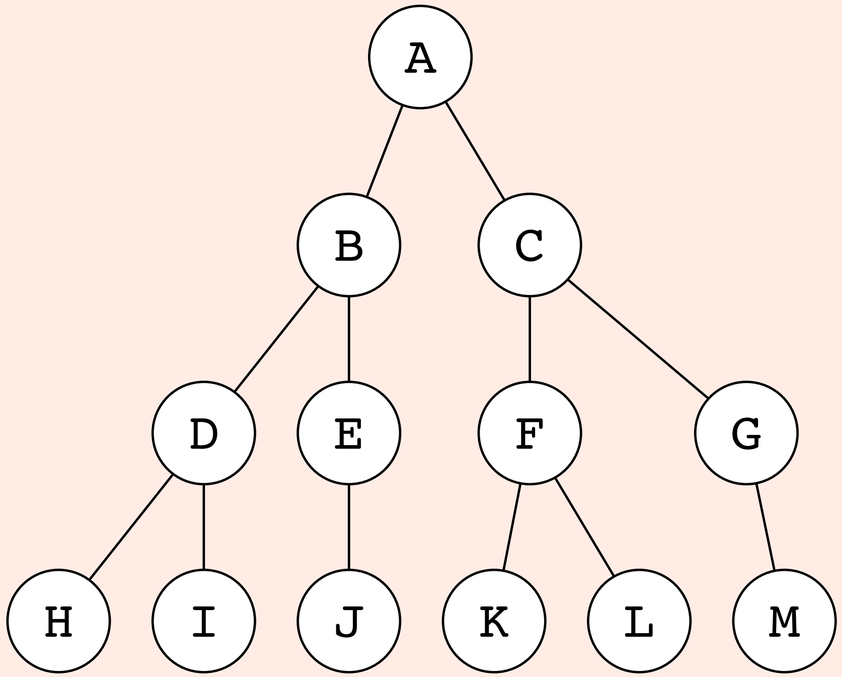


Parcours (1)



Pré: ABDHIEJCFKLGM

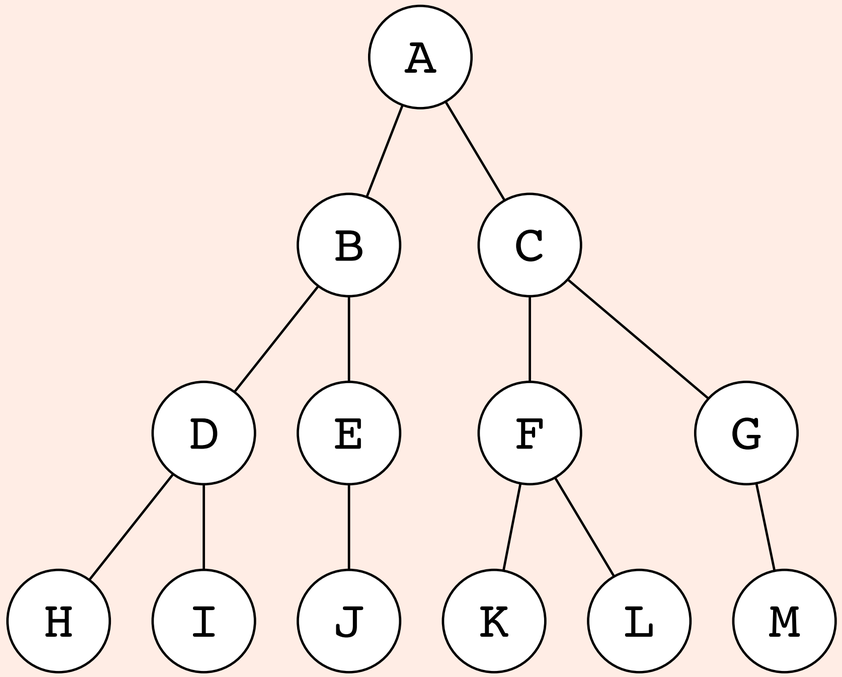
Parcours (1)



Pré: ABDHIEJCFKLGM

Post: HIDJEBKLFMGCA

Parcours (1)

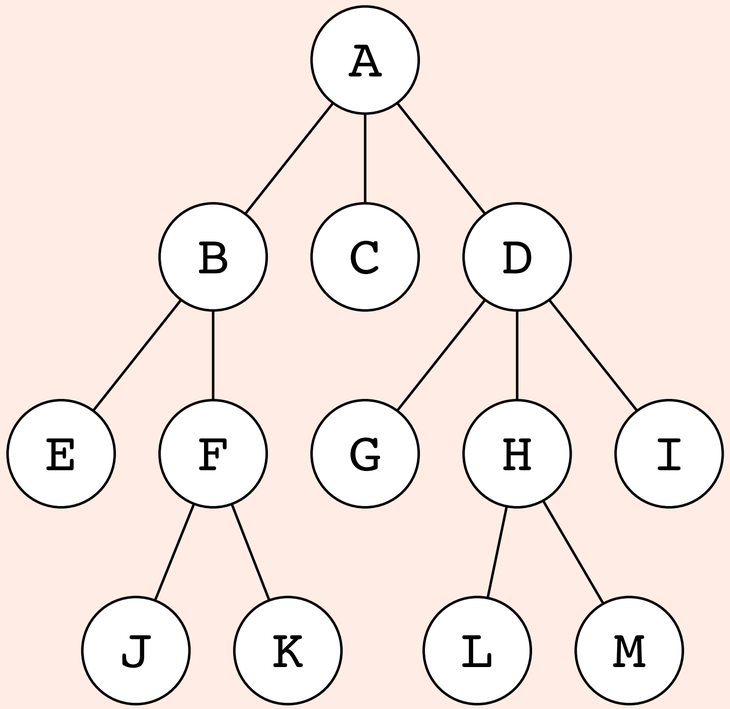


Pré: ABDHIEJCFKLGM

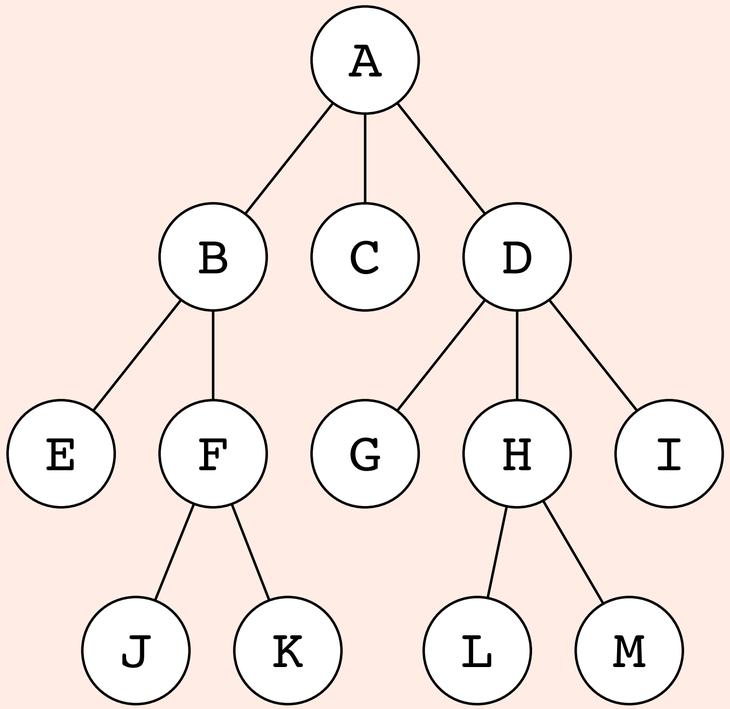
Post: HIDJEBKLFMGCA

Largeur: ABCDEFGHIJKLM

Parcours (2)

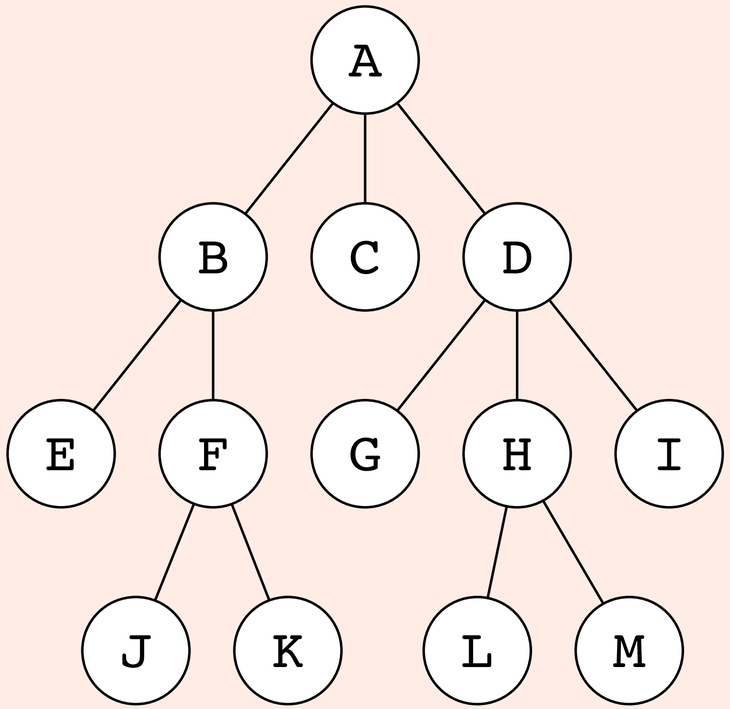


Parcours (2)



Pré: ABEFJKCDGHLMI

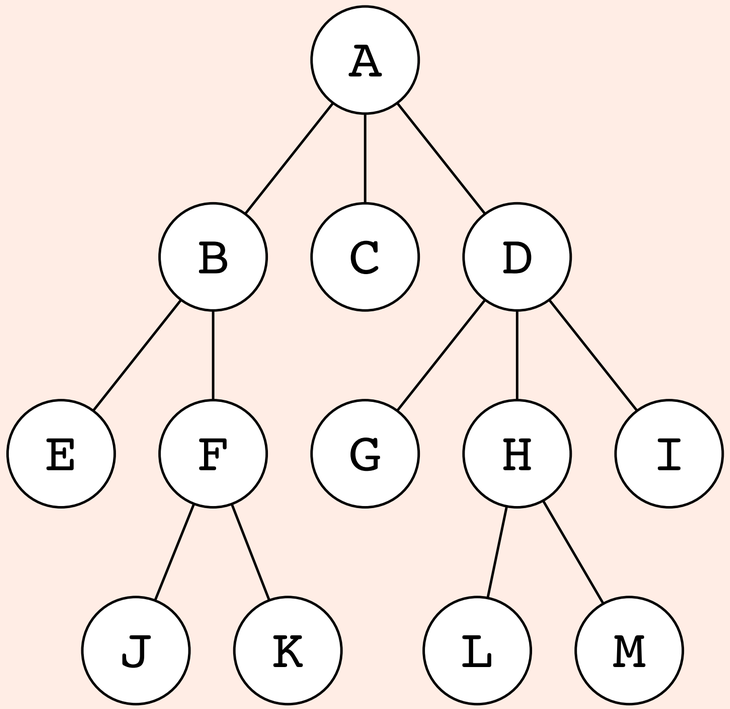
Parcours (2)



Pré: ABEFJKCDGHLMI

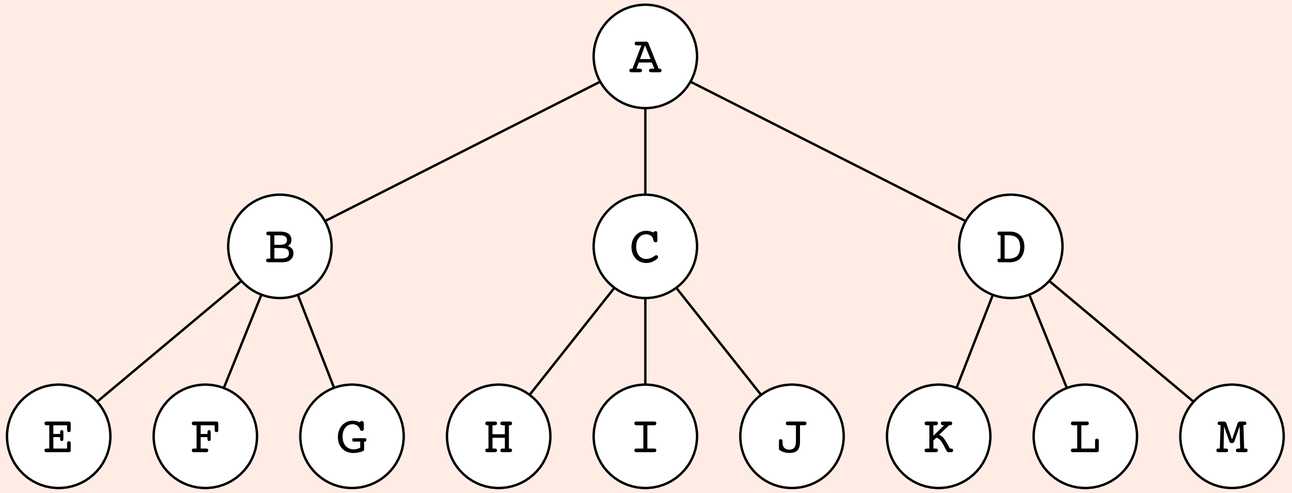
Post: EJKFBCGLMHIDA

Parcours (2)

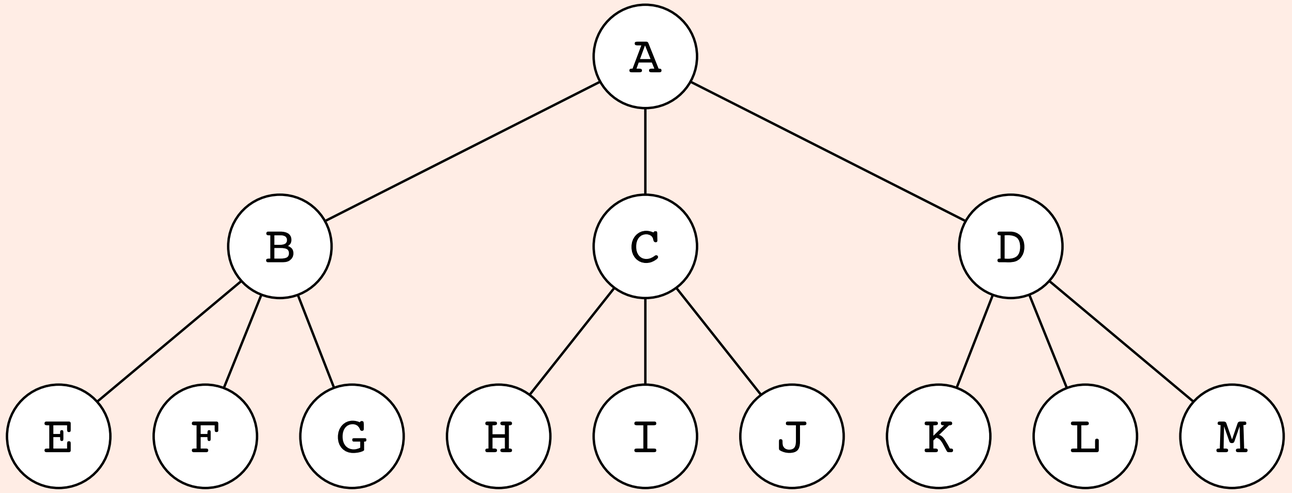



```
Pré:      ABEFJKCDGHLMI

Post:     EJKFBCGLMHIDA

Largeur:  ABCDEFGHIJKLM
```

Parcours (3)

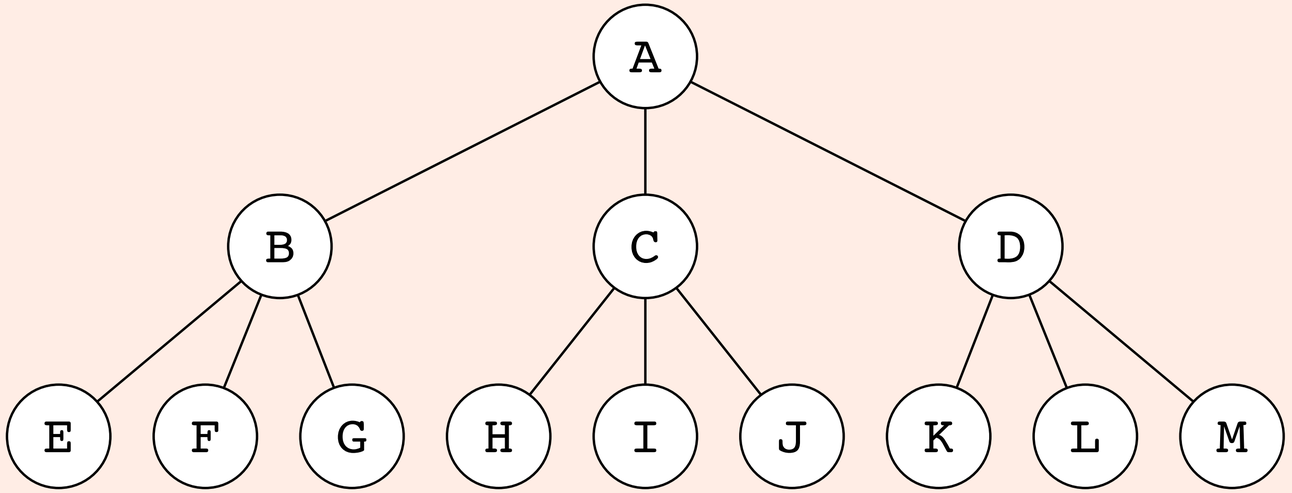


Parcours (3)



Pré: ABEFGCHIJDKLM

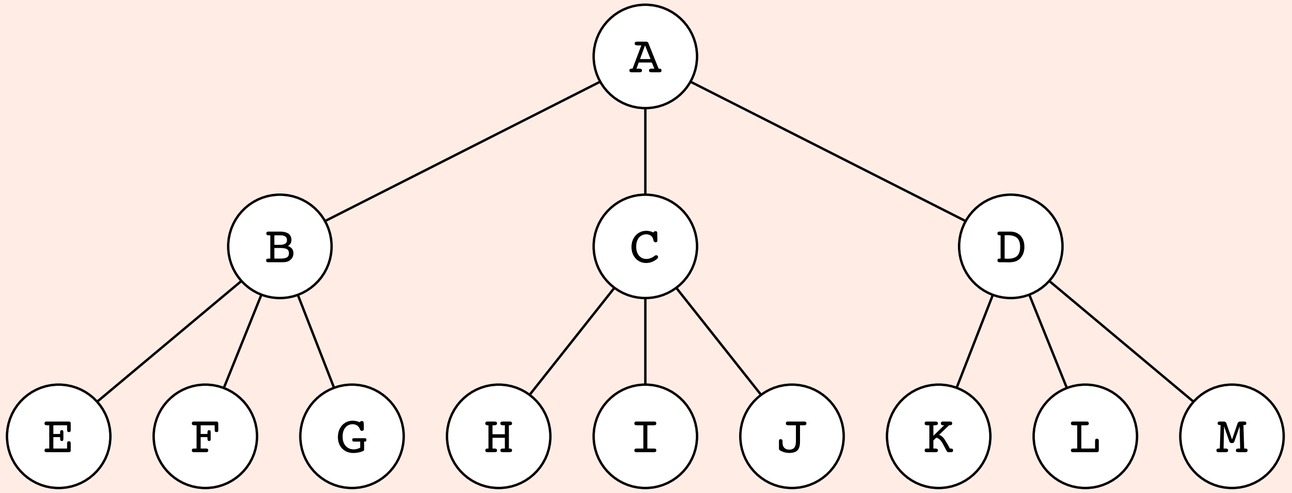
Parcours (3)



Pré: ABEFGCHIJDKLM

Post: EFGBHIJCKLMDA

Parcours (3)



Pré: ABEFGCHIJDKLM

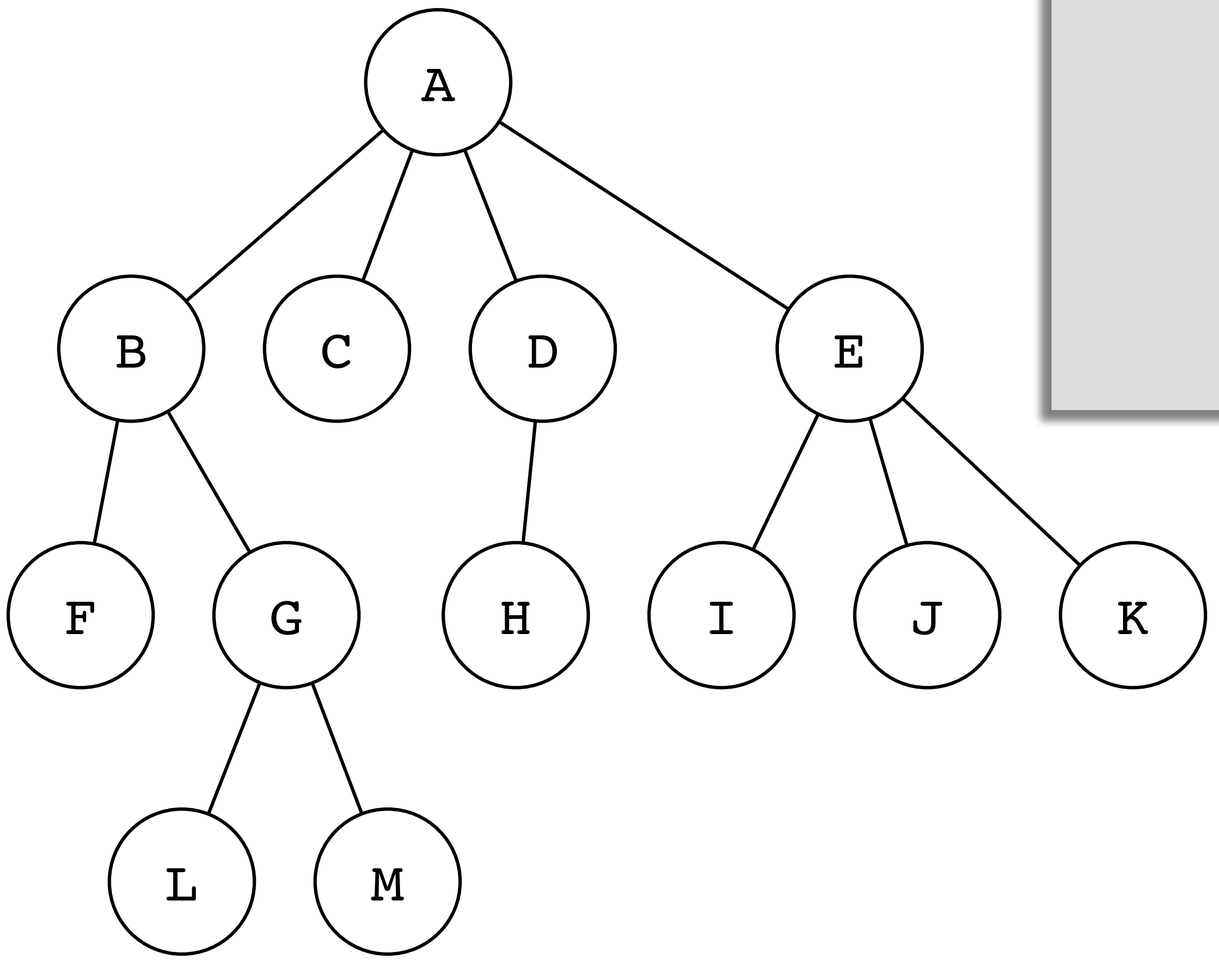
Post: EFGBHIJCKLMDA

Largeur: ABCDEFGHIJKLM

4. Représentations linéaires

Représentation indentée (1)

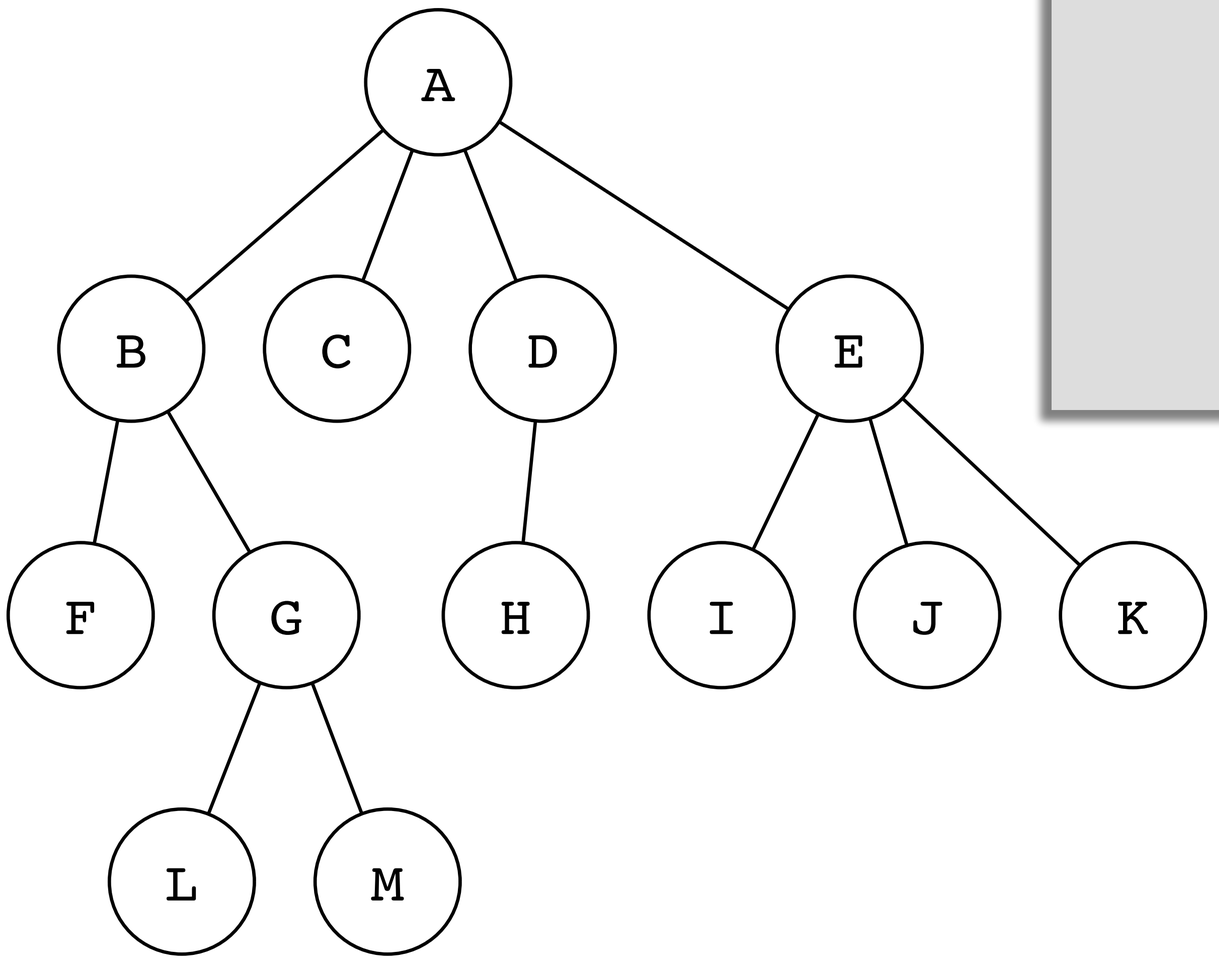
```
A
    B
        F
        G
            L
            M
    C
    D
        H
    E
        I
        J
        K
```




```
fonction indenter(r, n)
```

Représentation indentée (1)

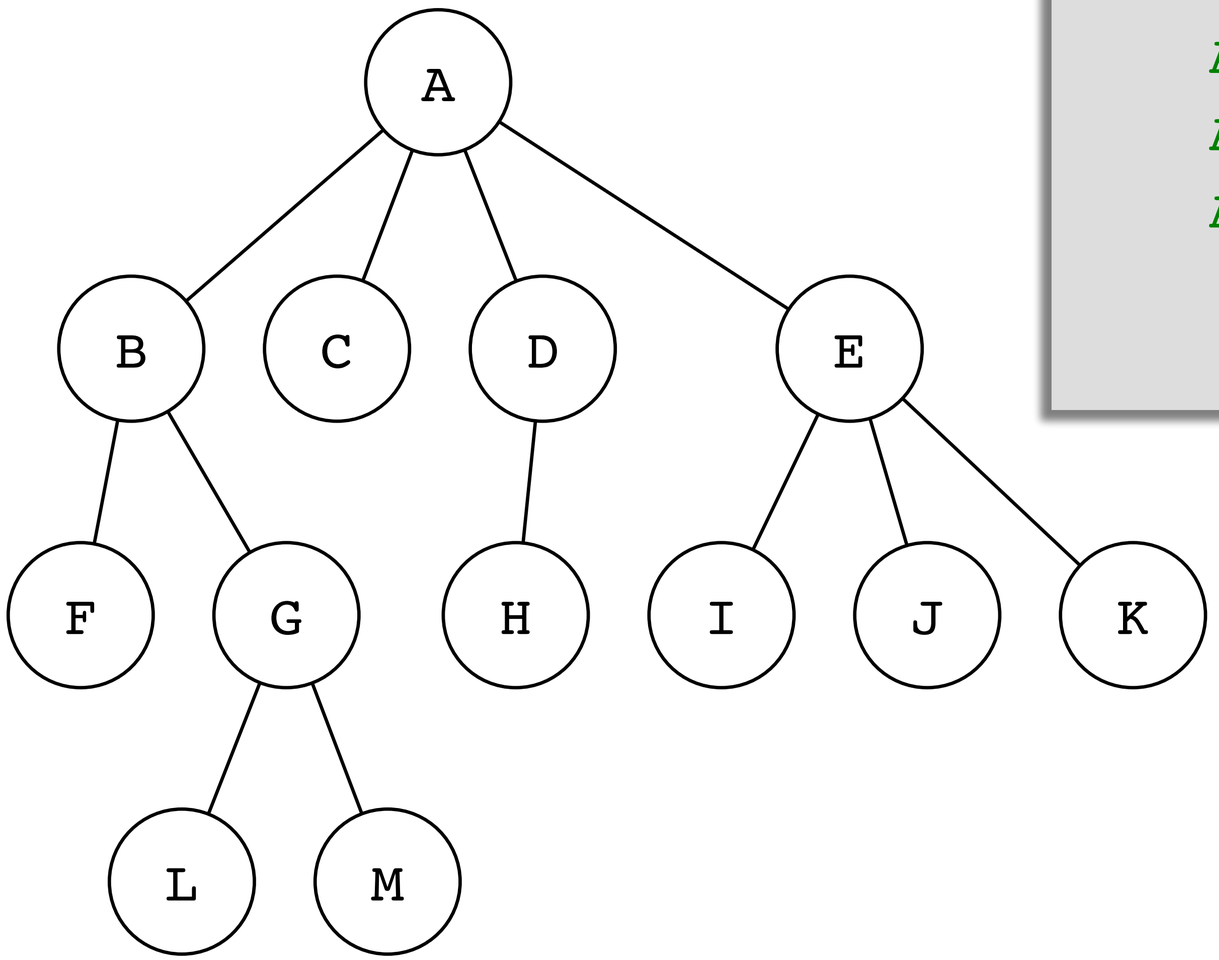
```
A
    B
        F
        G
            L
            M
    C
    D
        H
    E
        I
        J
        K
```




```
fonction indenter(r, n)
  si r != ∅
```

Représentation indentée (1)

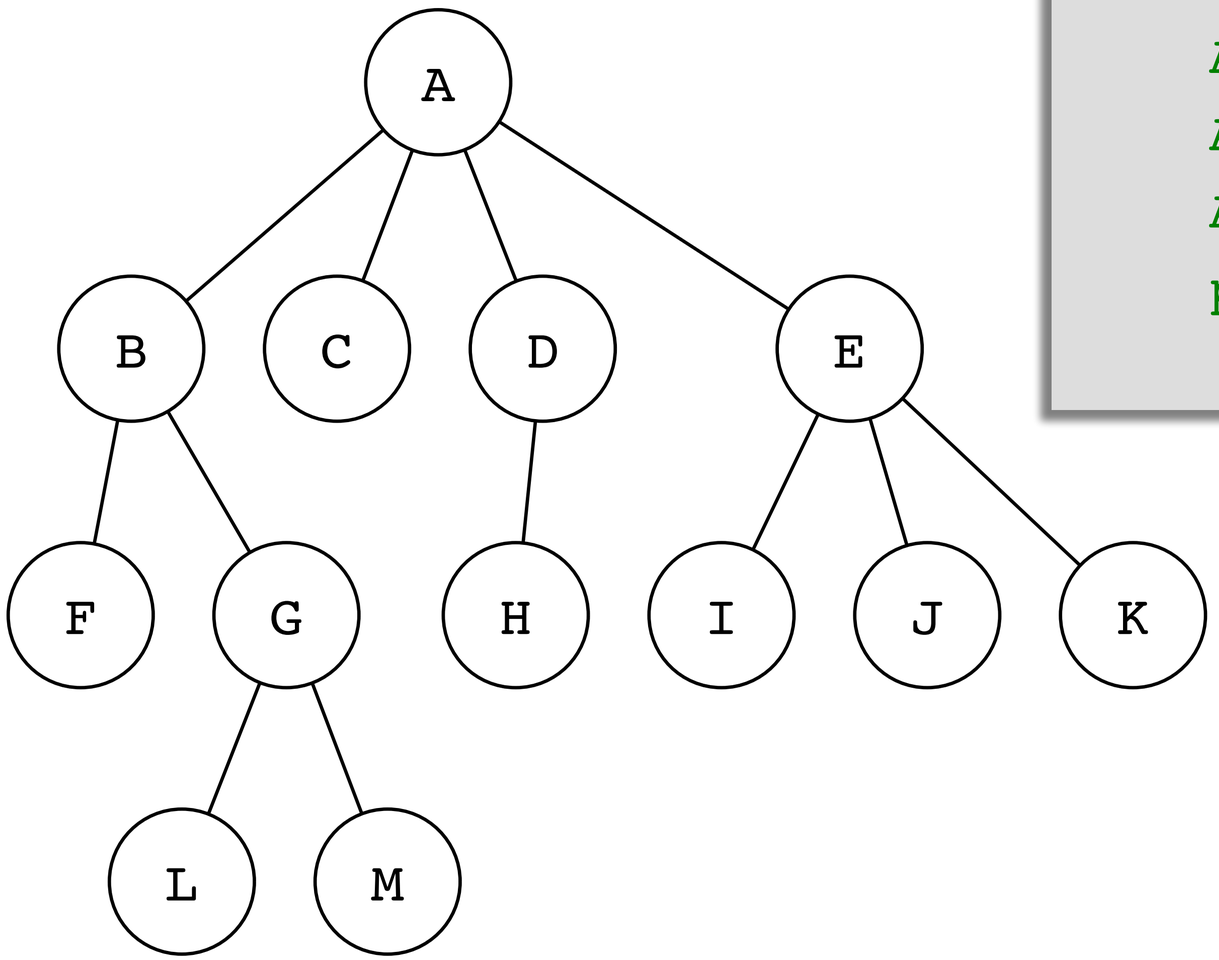
```
A
    B
        F
        G
            L
            M
    C
    D
        H
    E
        I
        J
        K
```




```
fonction indenter(r, n)
  si r != ∅
    Afficher n blancs
    Afficher r.etiquette
    Afficher saut de ligne
```

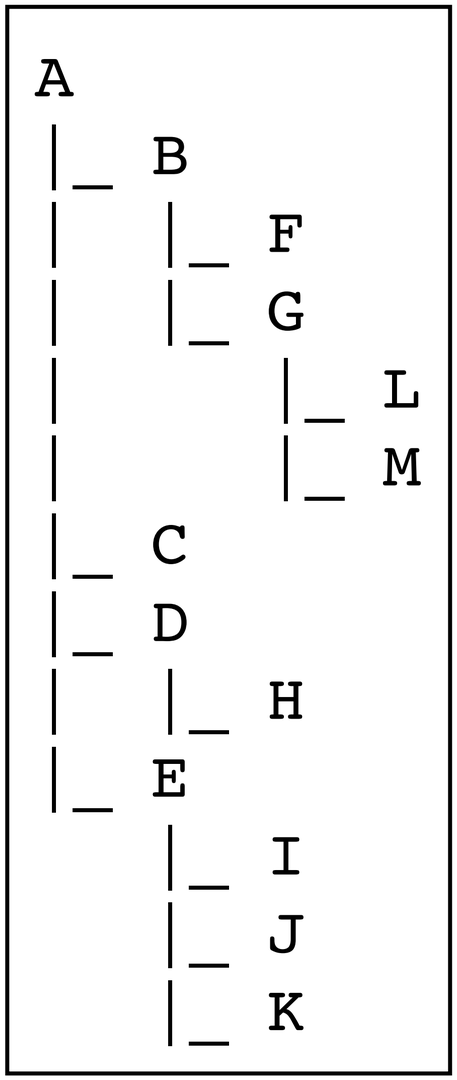
Représentation indentée (1)

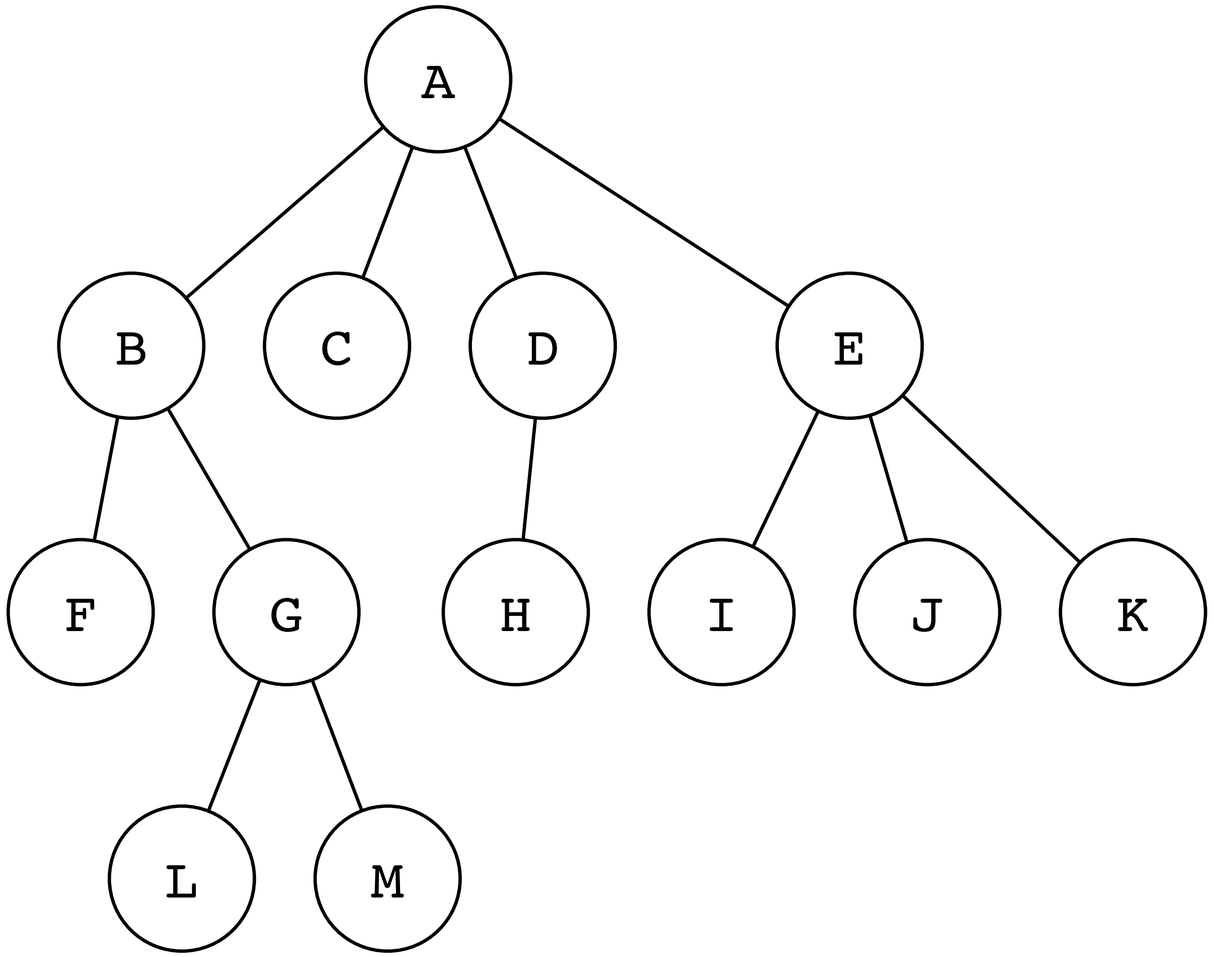
```
A
    B
        F
        G
            L
            M
    C
    D
        H
    E
        I
        J
        K
```




```
fonction indenter(r, n)
  si r != ∅
    Afficher n blancs
    Afficher r.etiquette
    Afficher saut de ligne
    pour tout enfant e de r
        indenter(r, n+1)
```

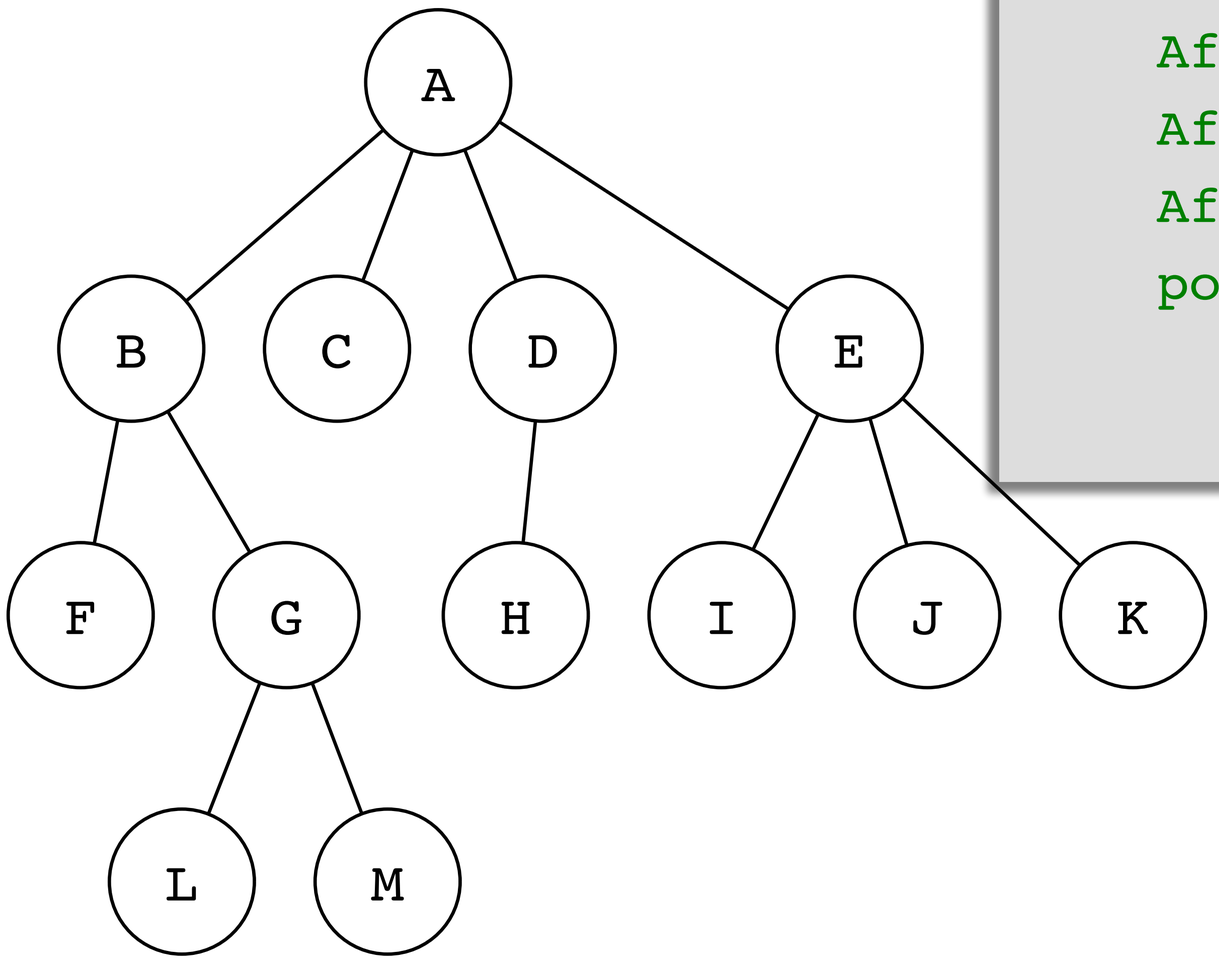
Représentation indentée (2)





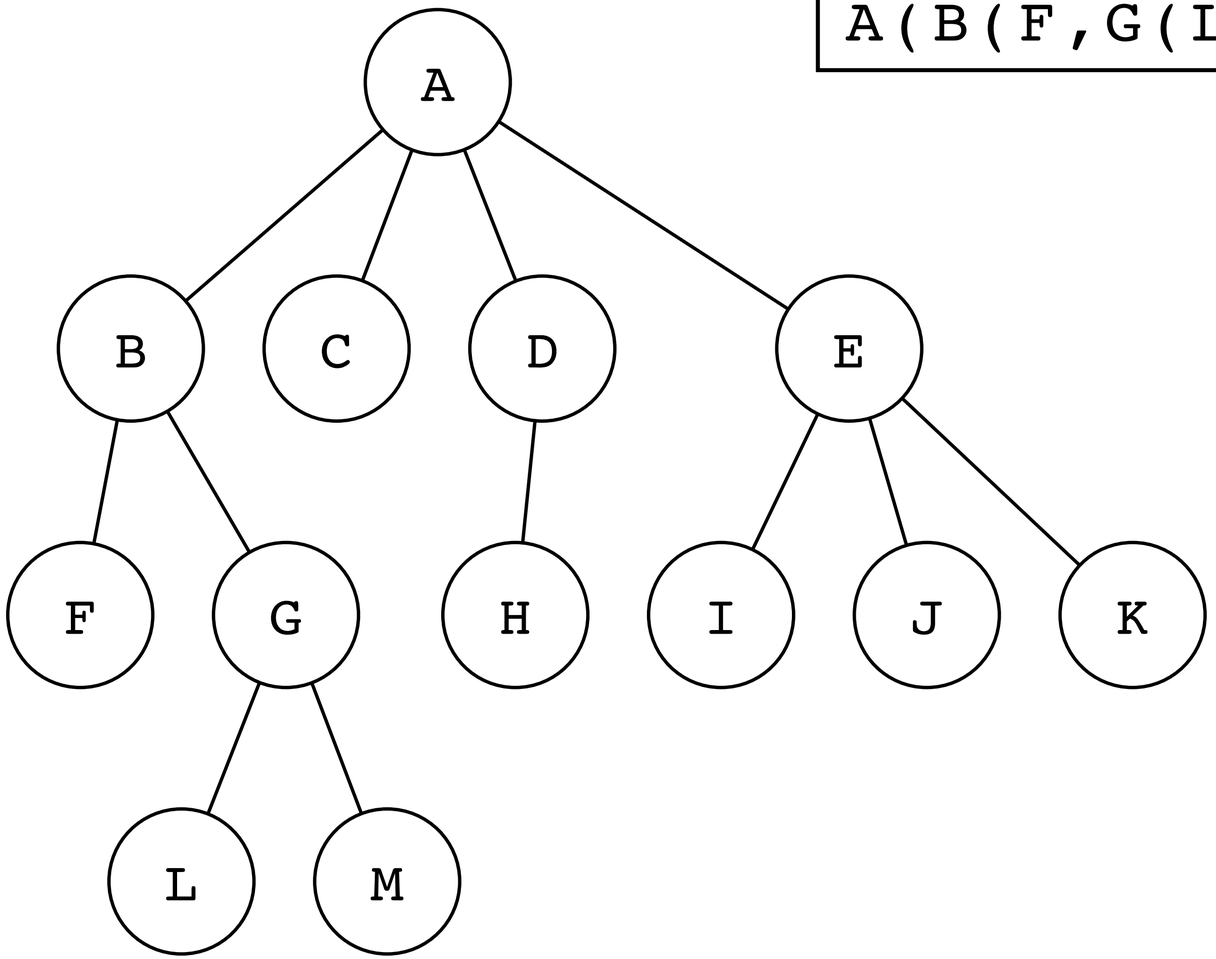
Représentation indentée (2)

```
A
 |_ B
 |   |_ F
 |   |_ G
 |       |_ L
 |       |_ M
 |_ C
 |_ D
 |   |_ H
 |_ E
     |_ I
     |_ J
     |_ K
```




```
fonction indenter(r, préfixe)
  si r != ∅
    Afficher préfixe
    Afficher r.etiquette
    Afficher saut de ligne
    pour tout enfant e de r
        pe ← préfixe modifié
        indenter(r,pe)
```

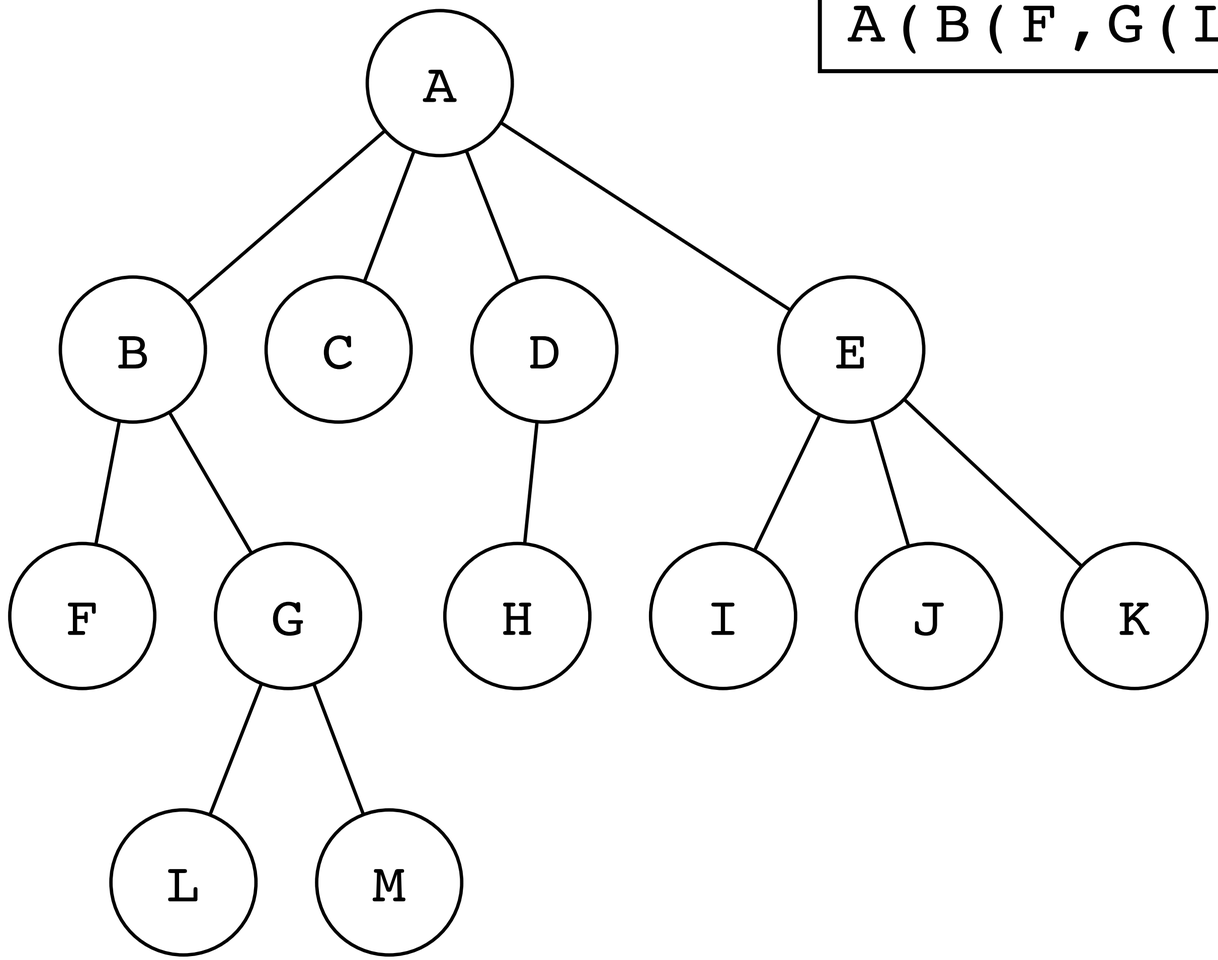
Listes imbriquées




```
A(B(F,G(L,M)),C,D(H),E(I,J,K))
```

```
fonction imbriquer(r)
```

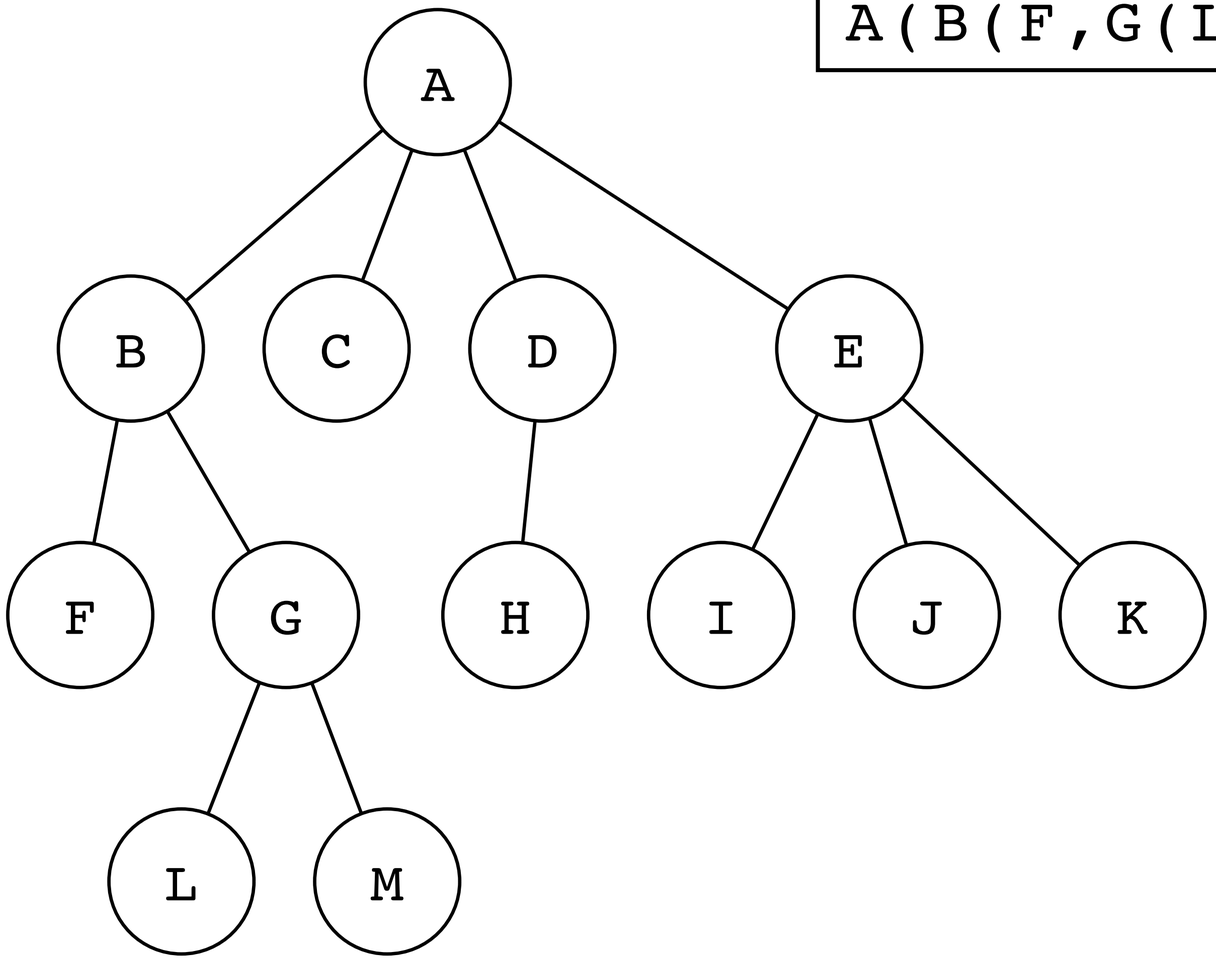
Listes imbriquées




```
A(B(F,G(L,M)),C,D(H),E(I,J,K))
```

```
fonction imbriquer(r)
  si r != ∅
```

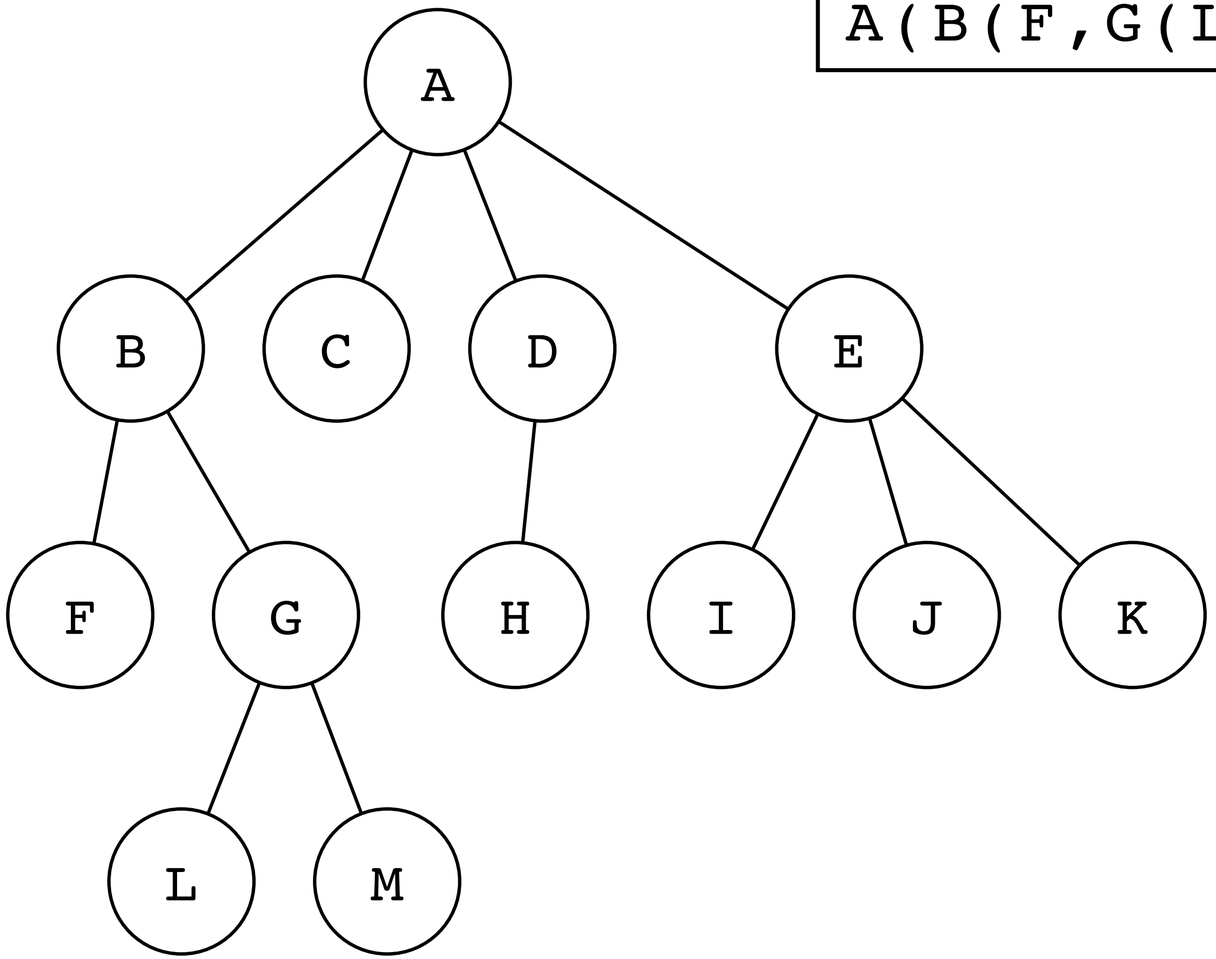
Listes imbriquées




```
A(B(F,G(L,M)),C,D(H),E(I,J,K))
```

```
fonction imbriquer(r)
  si r != ∅
    Afficher r.etiquette
```

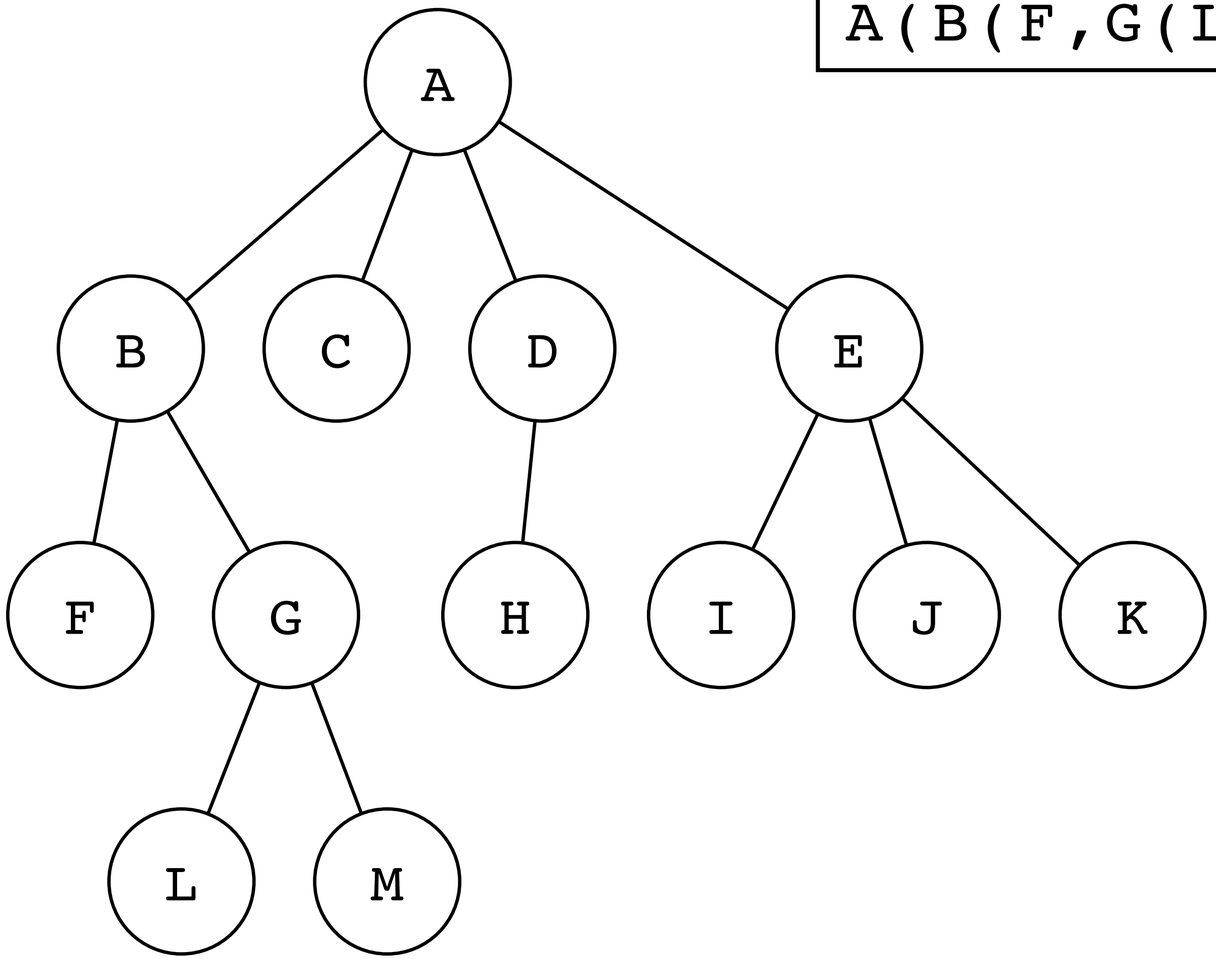
Listes imbriquées




```
A(B(F,G(L,M)),C,D(H),E(I,J,K))
```

```
fonction imbriquer(r)
  si r != ∅
    Afficher r.etiquette
    si degré(r) > 0
```

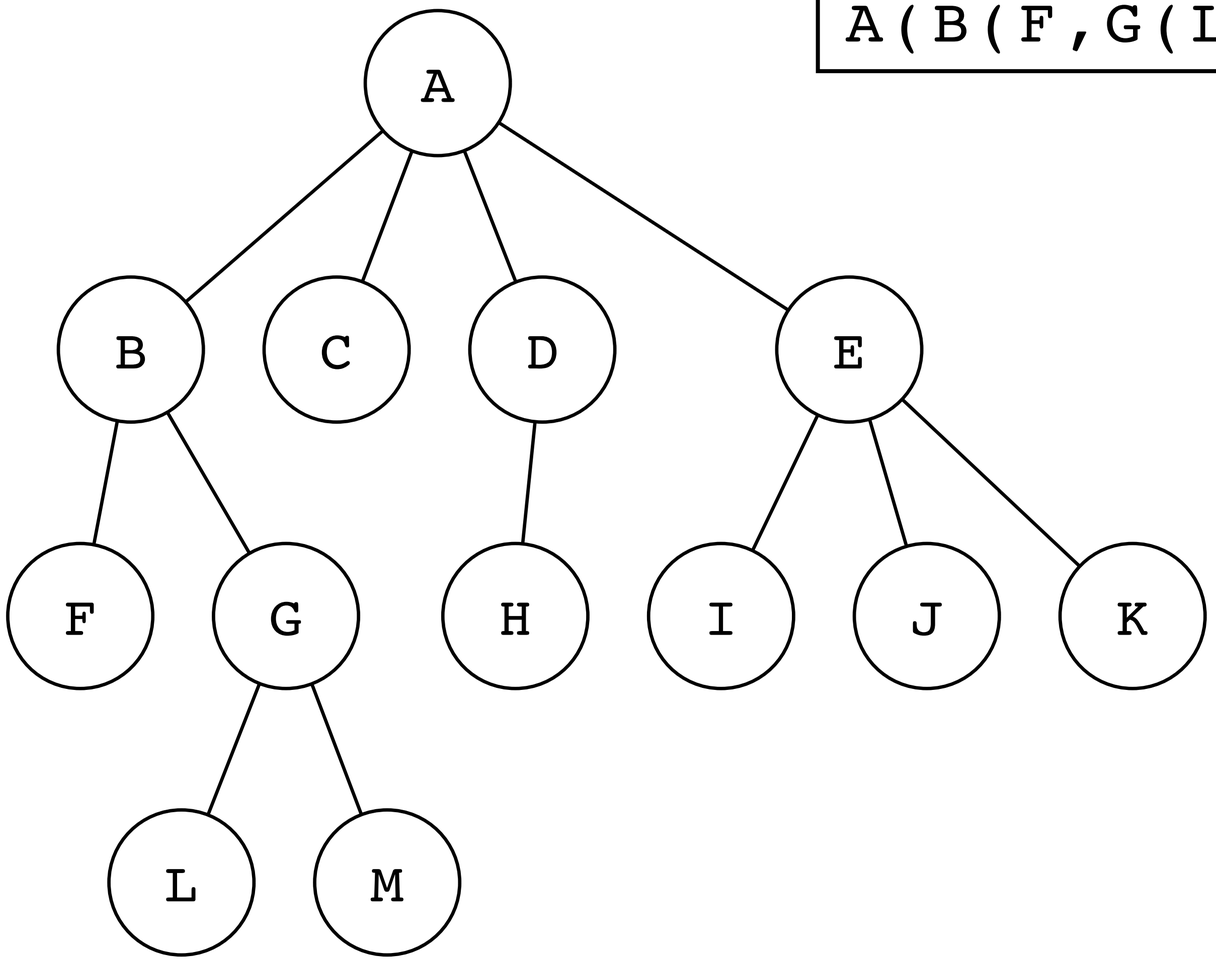
Listes imbriquées




```
A(B(F,G(L,M)),C,D(H),E(I,J,K))
```

```
fonction imbriquer(r)
  si r != ∅
    Afficher r.etiquette
    si degré(r) > 0
      Afficher "("
```

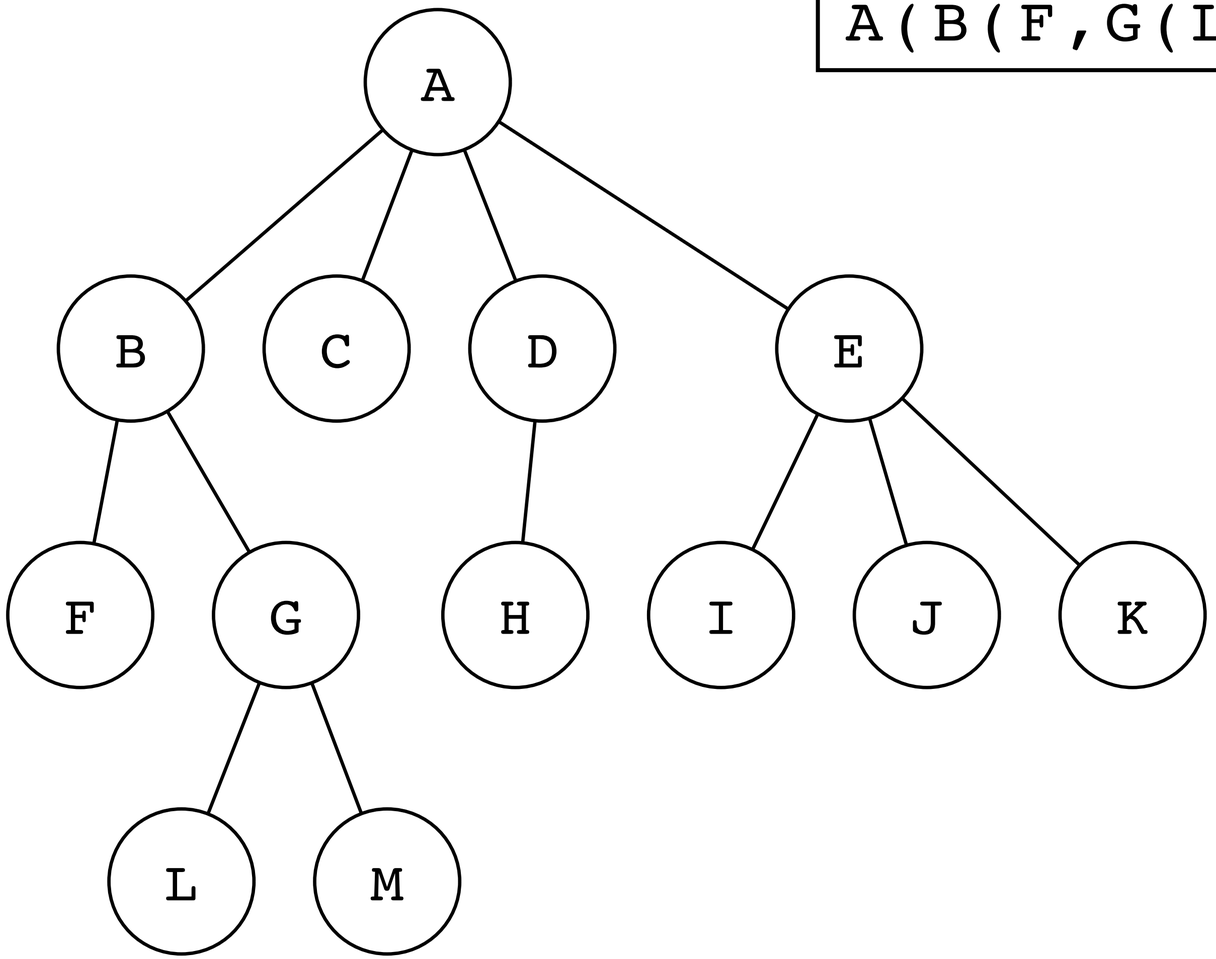
Listes imbriquées




```
A(B(F,G(L,M)),C,D(H),E(I,J,K))
```

```
fonction imbriquer(r)
  si r != ∅
    Afficher r.etiquette
    si degré(r) > 0
      Afficher "("
      pour tout enfant e de r
        imbriquer(e)
```

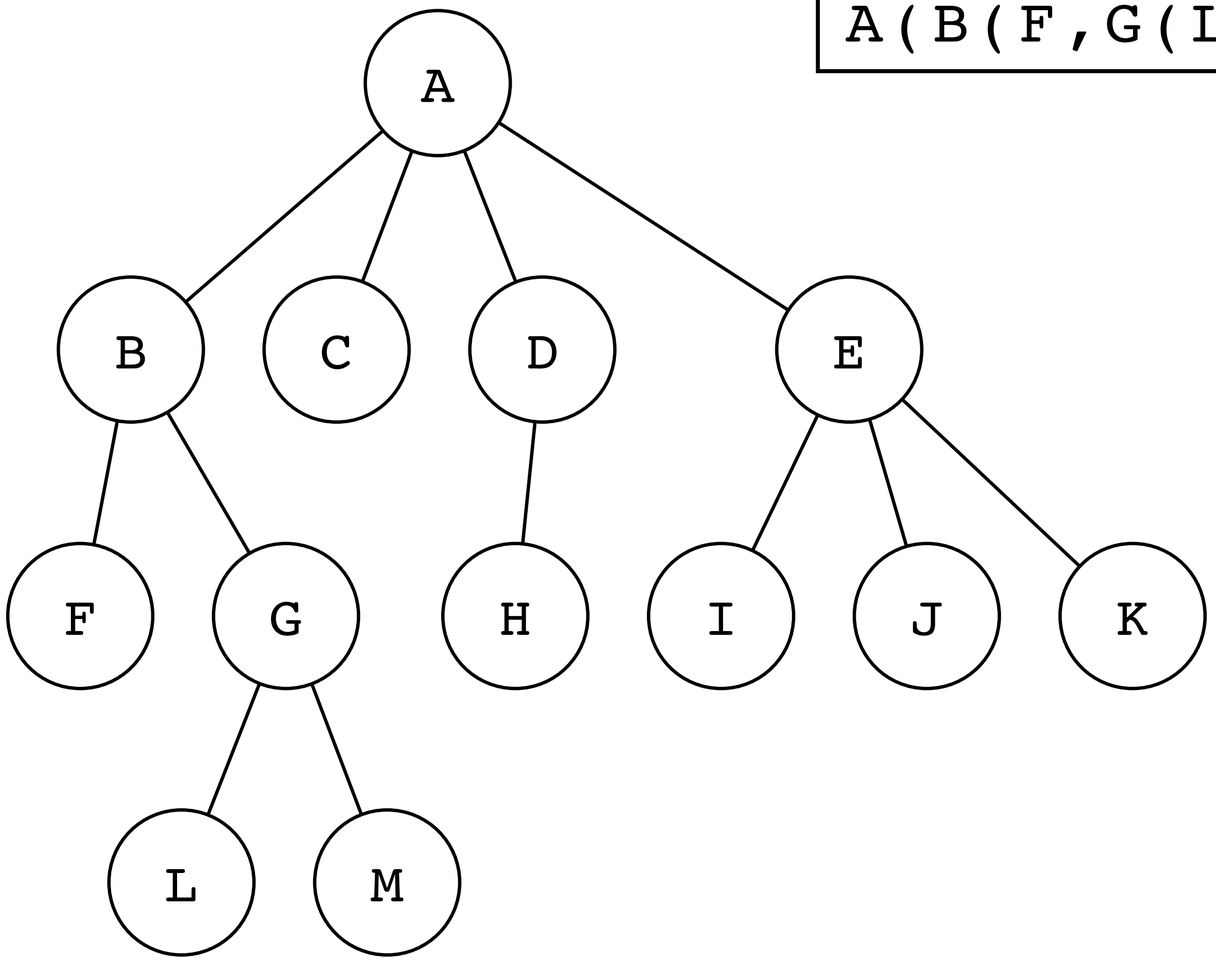
Listes imbriquées




```
A(B(F,G(L,M)),C,D(H),E(I,J,K))
```

```
fonction imbriquer(r)
  si r != ∅
    Afficher r.etiquette
    si degré(r) > 0
      Afficher "("
      pour tout enfant e de r
        imbriquer(e)
        si e n'est pas le dernier
          Afficher ","
```

Listes imbriquées




```
A(B(F,G(L,M)),C,D(H),E(I,J,K))
```

```
fonction imbriquer(r)
  si r != ∅
    Afficher r.etiquette
    si degré(r) > 0
      Afficher "("
      pour tout enfant e de r
        imbriquer(e)
        si e n'est pas le dernier
          Afficher ","
      Afficher ")"
```

Arbre depuis un liste imbriquée

```
void from_string(std::istringstream &s, Noeud *parent, Noeud *&out) {
```

Arbre depuis un liste imbriquée

```
void from_string(std::istringstream &s, Noeud *parent, Noeud *&out) {
```

```
struct Noeud {
    std::string etiquette;
    Noeud *parent;
    Noeud *aine;
    Noeud *puine;
};
```

Arbre depuis un liste imbriquée

```
void from_string(std::istringstream &s, Noeud *parent, Noeud *&out) {
```

```
struct Noeud {
    std::string etiquette;
    Noeud *parent;
    Noeud *aine;
    Noeud *puine;
};
```

```
Arbre::Arbre(std::string const &imbriquees) {
    std::istringstream s(imbriquees);
    from_string(s, nullptr, racine);
}
```

Arbre depuis un liste imbriquée

```
void from_string(std::istringstream &s, Noeud *parent, Noeud *&out) {
    out = new Noeud{read_label(s),
                    parent, nullptr, nullptr};
```

```
struct Noeud {
    std::string etiquette;
    Noeud *parent;
    Noeud *aine;
    Noeud *puine;
};
```

```
Arbre::Arbre(std::string const &imbriquees) {
    std::istringstream s(imbriquees);
    from_string(s, nullptr, racine);
}
```

Arbre depuis un liste imbriquée

```
void from_string(std::istringstream &s, Noeud *parent, Noeud *&out) {
    out = new Noeud{read_label(s),
                    parent, nullptr, nullptr};

    Noeud *n = out;
    char c;
    while (s.get(c)) {
```

```
struct Noeud {
    std::string etiquette;
    Noeud *parent;
    Noeud *aine;
    Noeud *puine;
};
```

```
Arbre::Arbre(std::string const &imbriquees) {
    std::istringstream s(imbriquees);
    from_string(s, nullptr, racine);
}
```

Arbre depuis un liste imbriquée

```
void from_string(std::istringstream &s, Noeud *parent, Noeud *&out) {
    out = new Noeud{read_label(s),
                    parent, nullptr, nullptr};

    Noeud *n = out;
    char c;
    while (s.get(c)) {
        switch (c) {
            case ',':
                n->puine = new Noeud{read_label(s),
                                     parent, nullptr, nullptr};
                n = n->puine;
                break;
```

```
struct Noeud {
    std::string etiquette;
    Noeud *parent;
    Noeud *aine;
    Noeud *puine;
};
```

```
Arbre::Arbre(std::string const &imbriquees) {
    std::istringstream s(imbriquees);
    from_string(s, nullptr, racine);
}
```

Arbre depuis un liste imbriquée

```
void from_string(std::istringstream &s, Noeud *parent, Noeud *&out) {
    out = new Noeud{read_label(s),
                    parent, nullptr, nullptr};

    Noeud *n = out;
    char c;
    while (s.get(c)) {
        switch (c) {
            case ',':
                n->puine = new Noeud{read_label(s),
                                     parent, nullptr, nullptr};
                n = n->puine;
                break;
            case '(':
                from_string(s, n, n->aine);
                break;
```

```
struct Noeud {
    std::string etiquette;
    Noeud *parent;
    Noeud *aine;
    Noeud *puine;
};
```

```
Arbre::Arbre(std::string const &imbriquees) {
    std::istringstream s(imbriquees);
    from_string(s, nullptr, racine);
}
```

Arbre depuis un liste imbriquée

```
void from_string(std::istringstream &s, Noeud *parent, Noeud *&out) {
    out = new Noeud{read_label(s),
                    parent, nullptr, nullptr};

    Noeud *n = out;
    char c;
    while (s.get(c)) {
        switch (c) {
            case ',':
                n->puine = new Noeud{read_label(s),
                                     parent, nullptr, nullptr};
                n = n->puine;
                break;
            case '(':
                from_string(s, n, n->aine);
                break;
            case ')':
                return;
```

```
struct Noeud {
    std::string etiquette;
    Noeud *parent;
    Noeud *aine;
    Noeud *puine;
};
```

```
Arbre::Arbre(std::string const &imbriquees) {
    std::istringstream s(imbriquees);
    from_string(s, nullptr, racine);
}
```

Arbre depuis un liste imbriquée

```
void from_string(std::istringstream &s, Noeud *parent, Noeud *&out) {
    out = new Noeud{read_label(s),
                    parent, nullptr, nullptr};

    Noeud *n = out;
    char c;
    while (s.get(c)) {
        switch (c) {
            case ',':
                n->puine = new Noeud{read_label(s),
                                     parent, nullptr, nullptr};
                n = n->puine;
                break;
            case '(':
                from_string(s, n, n->aine);
                break;
            case ')':
                return;
            default :
                throw bad_expression();
        }
    }
}
```

```
struct Noeud {
    std::string etiquette;
    Noeud *parent;
    Noeud *aine;
    Noeud *puine;
};
```

```
Arbre::Arbre(std::string const &imbriquees) {
    std::istringstream s(imbriquees);
    from_string(s, nullptr, racine);
}
```

Exercices

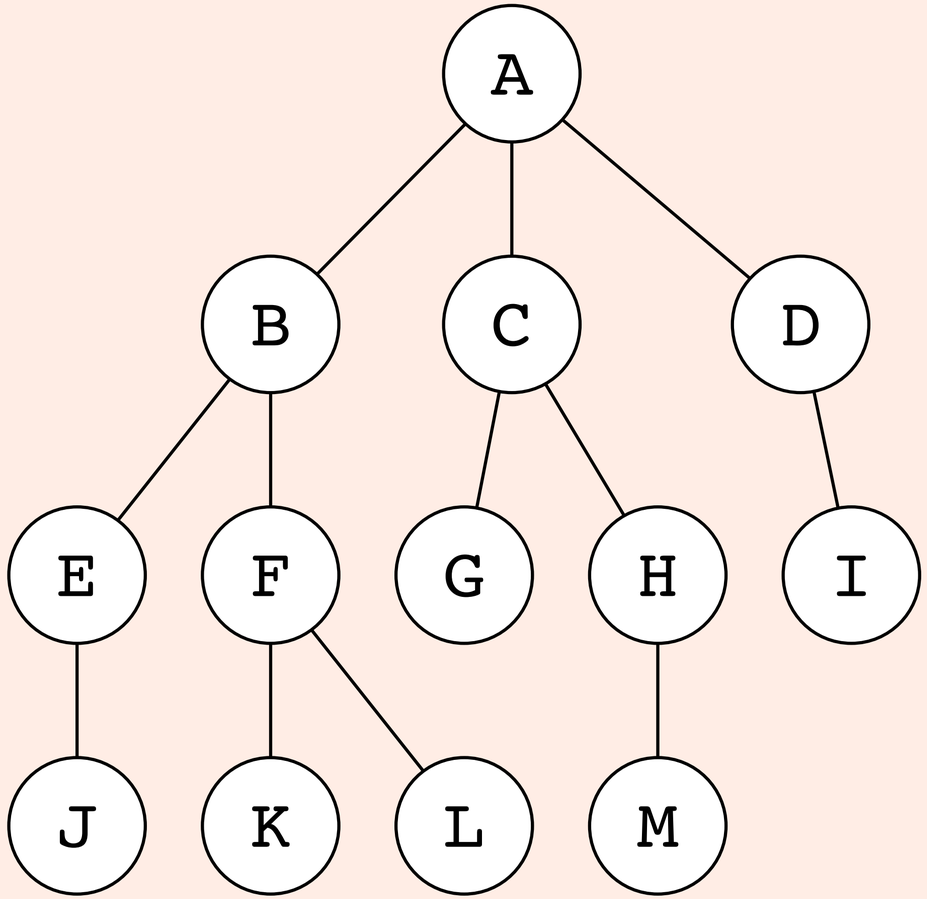
- Dessinez les graphes représentés par les listes suivantes

```
A(B(E(J),F(K,L)),C(G,H(M)),D(I))
```

```
A(B(D(L),F(I),G(H,K)),C(E(M),J))
```

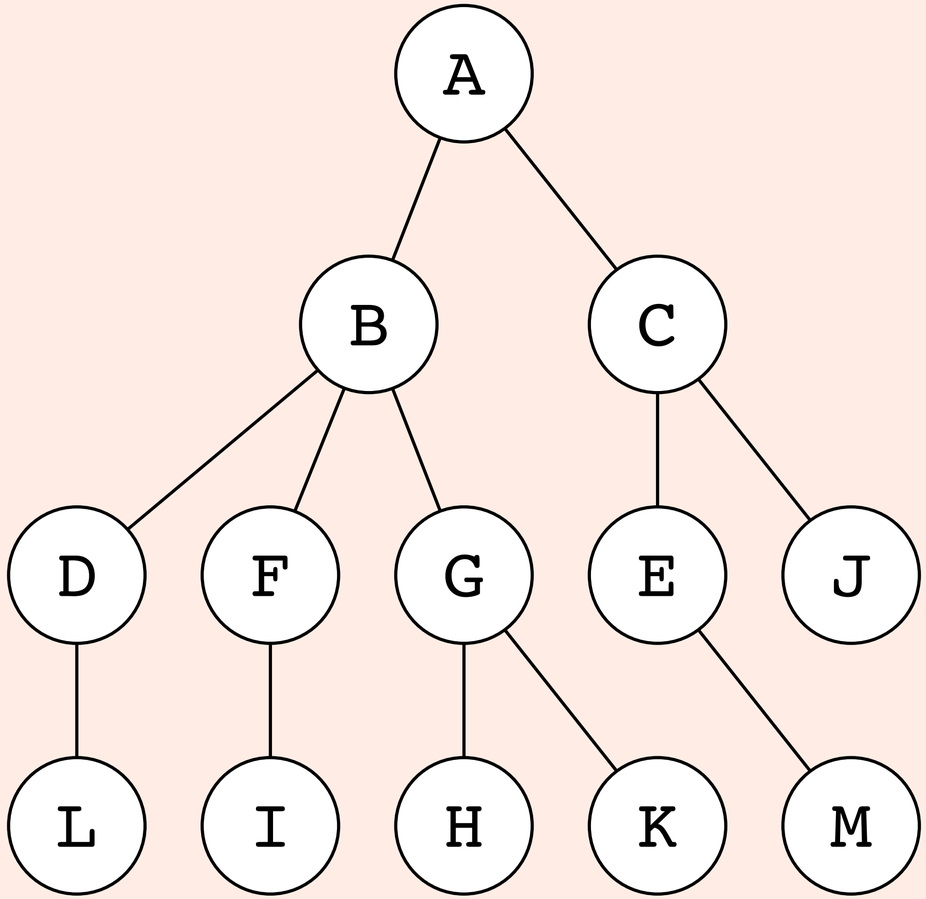
Solution (1)

```
A(B(E(J),F(K,L)),C(G,H(M)),D(I))
```



Solution (2)

```
A(B(D(L),F(I),G(H,K)),C(E(M),J))
```



HEIG VD

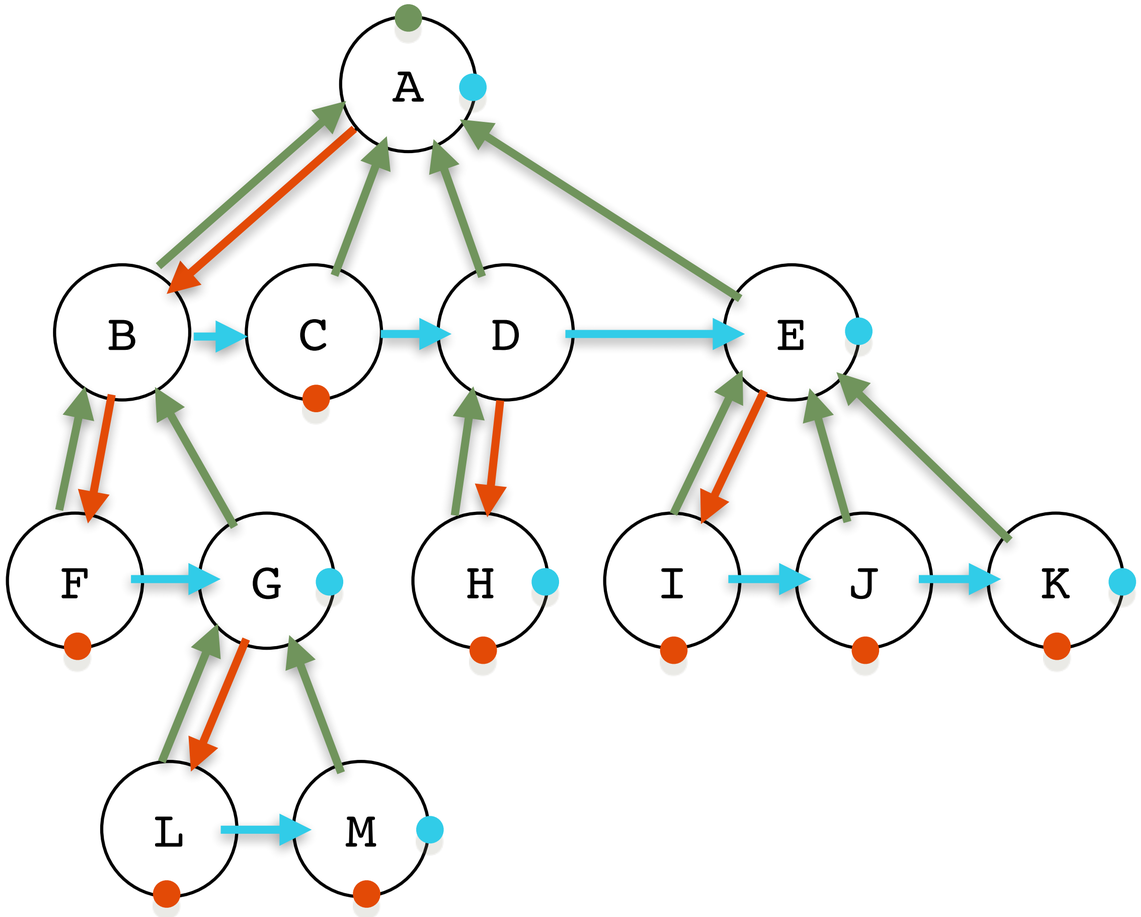
Itération sur un arbre

Itération

- Itérer = parcourir avec `début`, `suivant`, `fin`

```
fonction profondeur (r, fn)
n ← début(r)
tant que n != fin(n)
   fn(n)
   n ← suivant(n)
```

```
structure Noeud<T>
    T étiquette
    Noeud* parent
    Noeud* ainé
    Noeud* puiné
```

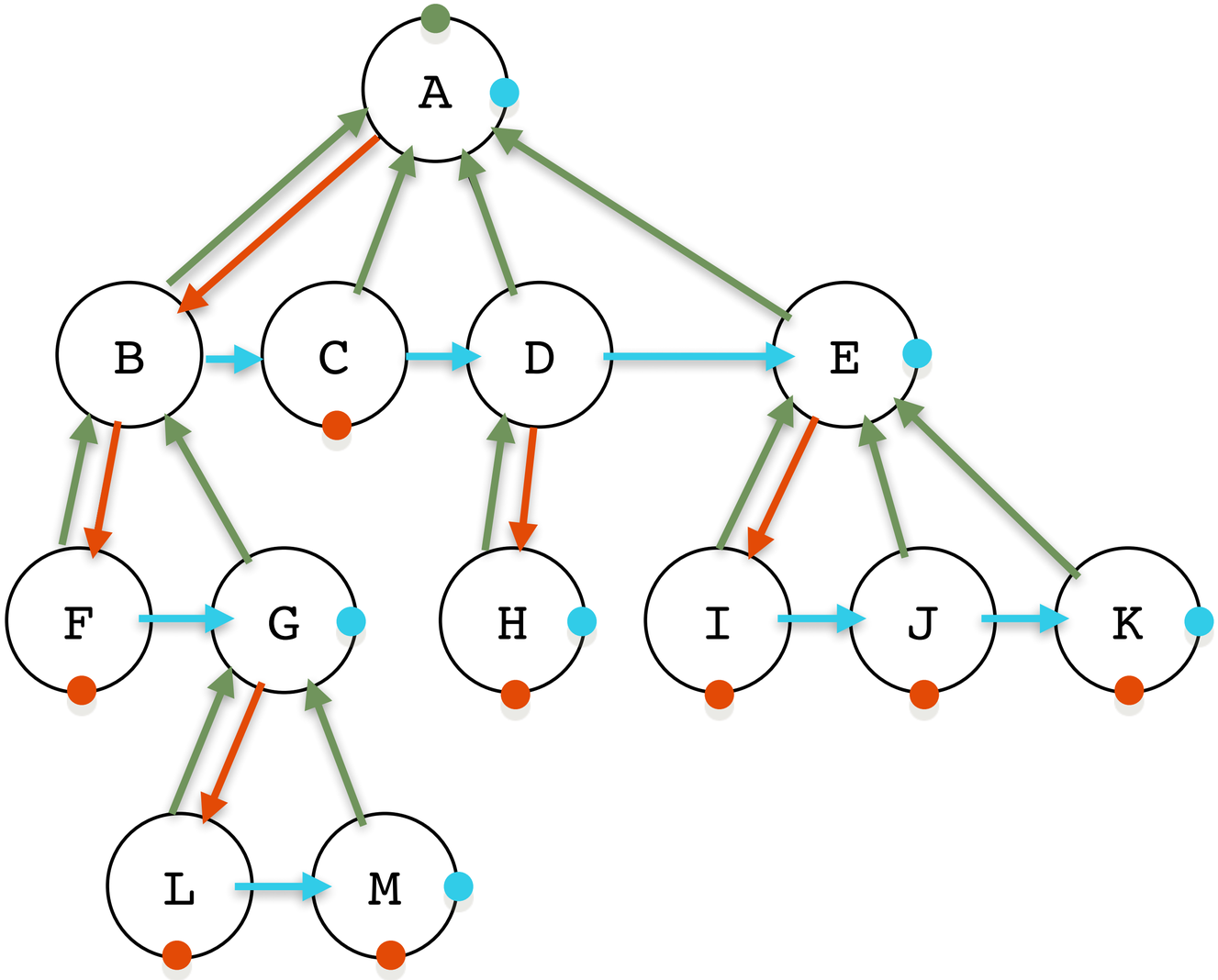


Itération en pré-ordre

```
fonction début(r)
```

```
fonction fin(r)
```

```
fonction suivant(r)
```

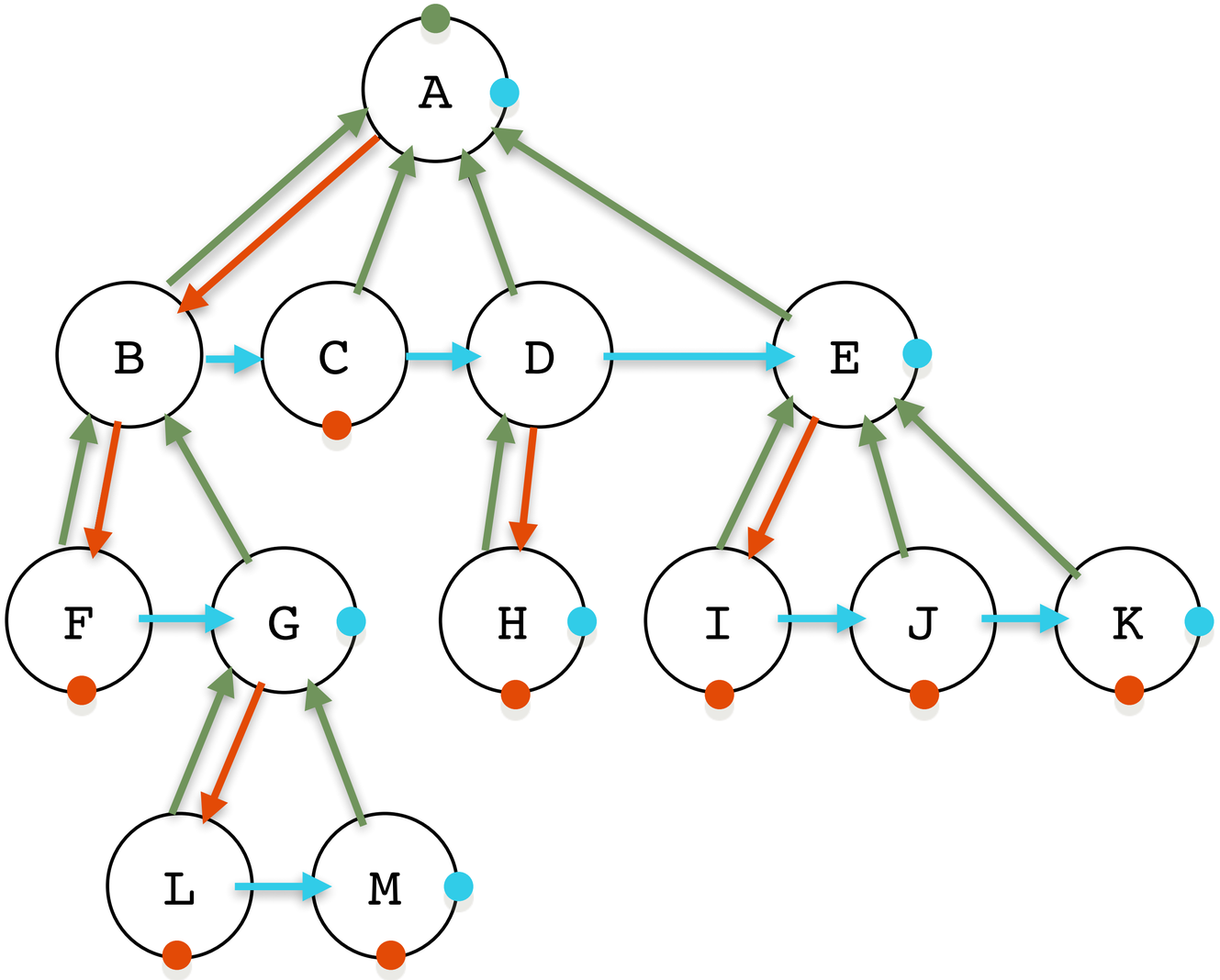


Itération en pré-ordre

```
fonction début(r)
  retourner r
```

```
fonction fin(r)
```

```
fonction suivant(r)
```

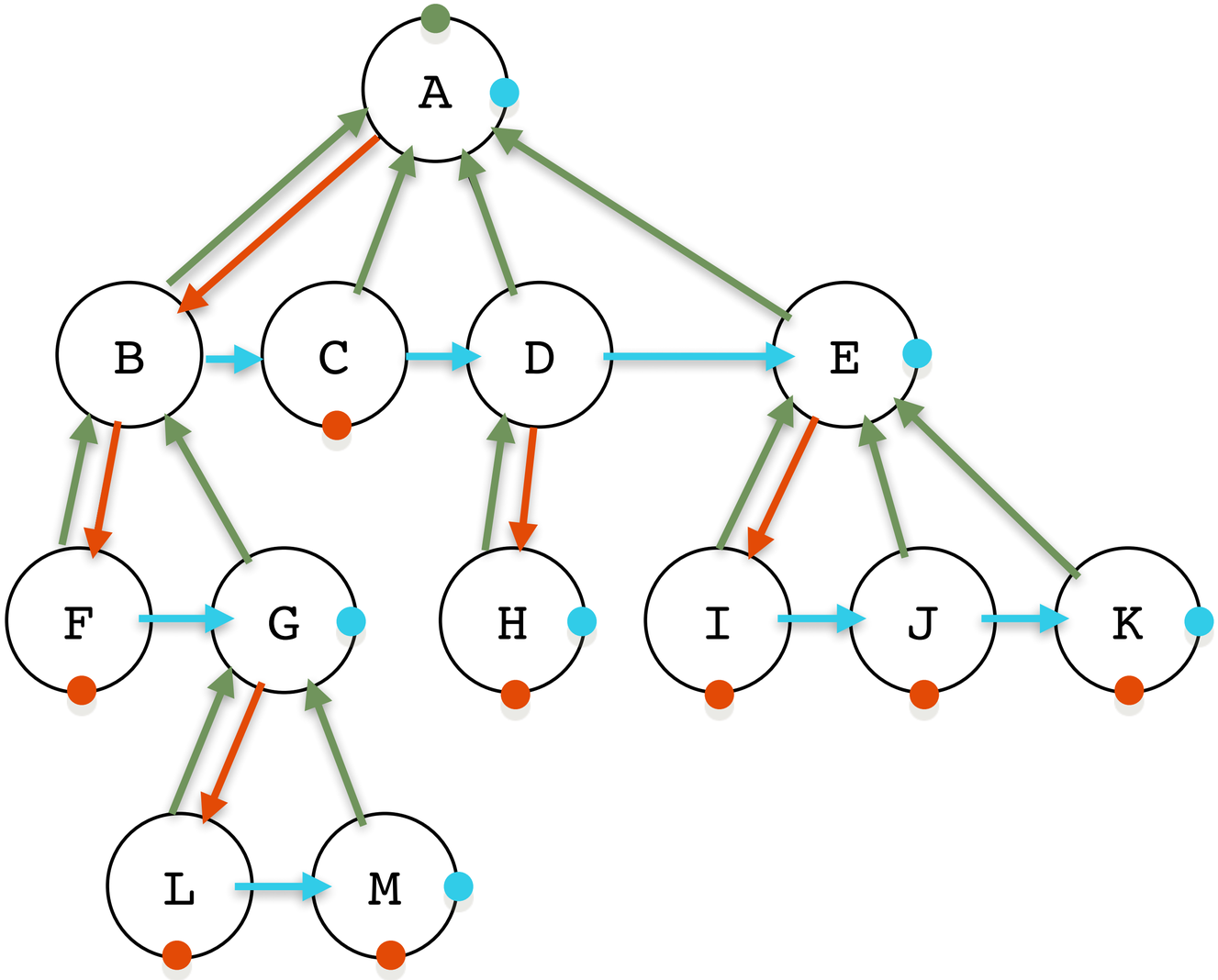


Itération en pré-ordre

```
fonction début(r)
  retourner r
```

```
fonction fin(r)
  retourner ∅
```

```
fonction suivant(r)
```

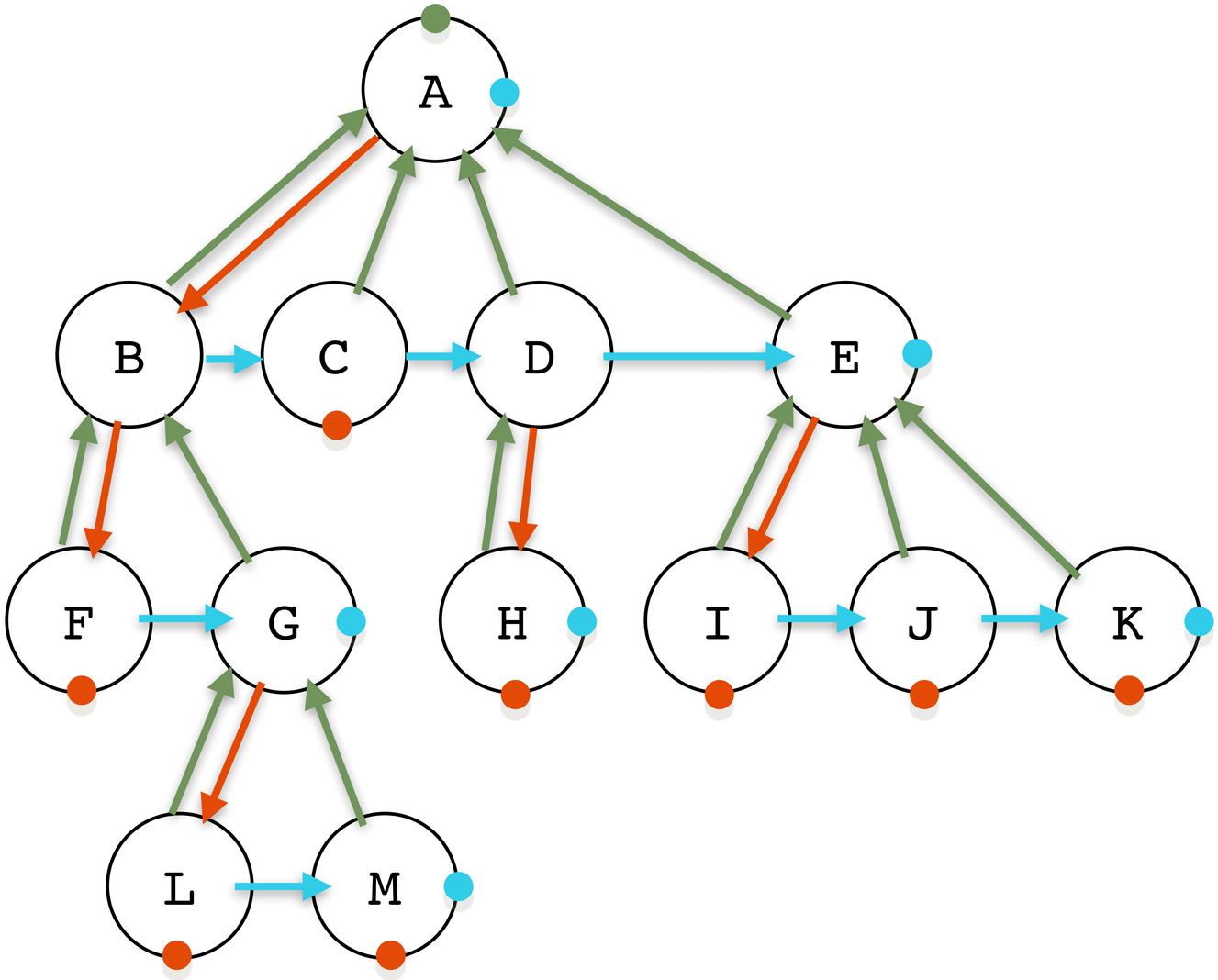


Itération en pré-ordre

```
fonction début(r)
  retourner r
```

```
fonction fin(r)
  retourner ∅
```

```
fonction suivant(r)
  si r.ainé != ∅
    retourner r.ainé
```

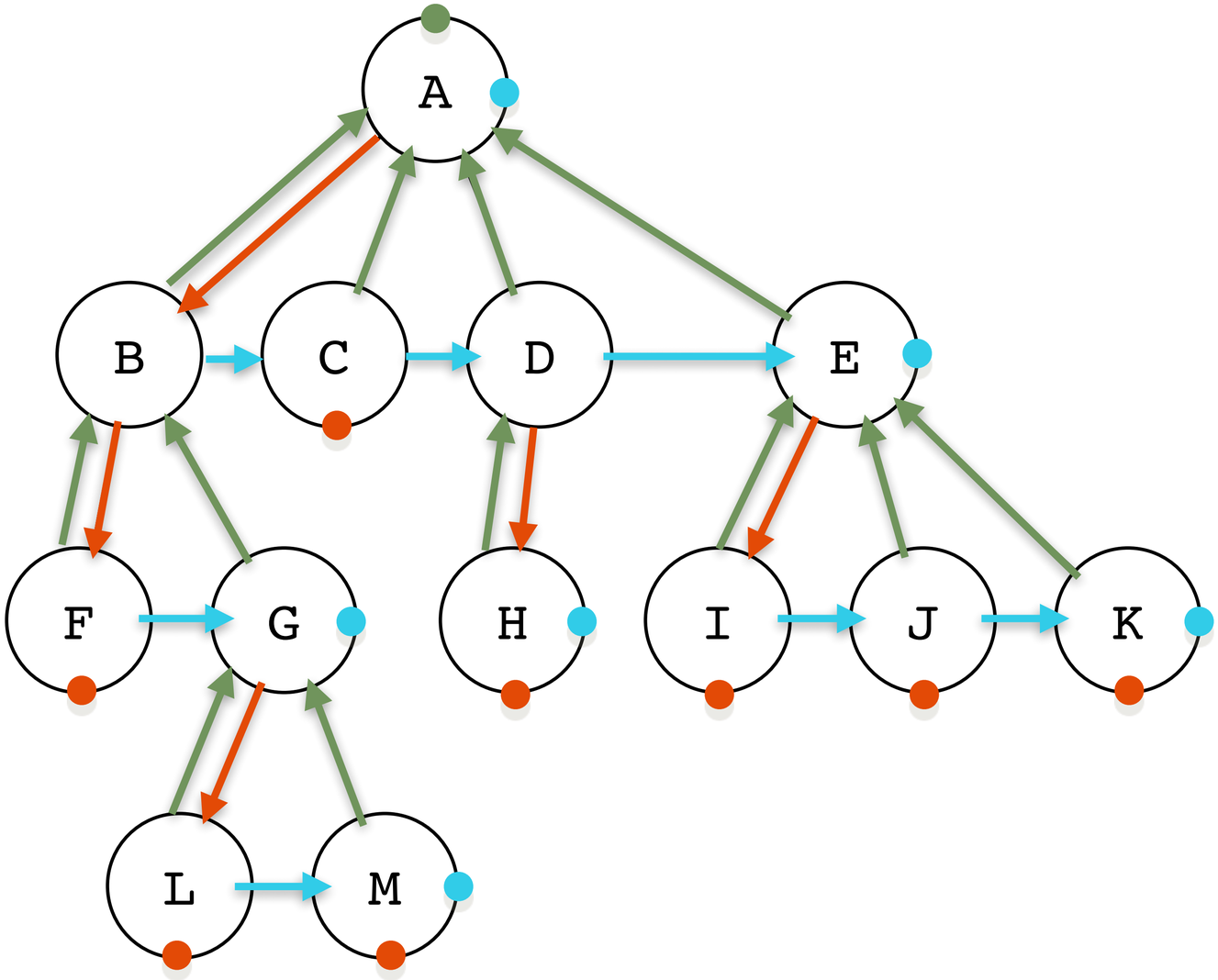


Itération en pré-ordre

```
fonction début(r)
  retourner r
```

```
fonction fin(r)
  retourner ∅
```

```
fonction suivant(r)
  si r.ainé != ∅
    retourner r.ainé
  sinon
    tant que r.puiné == ∅
          et r.parent != ∅
      r ← r.parent
    retourner r.puiné
```

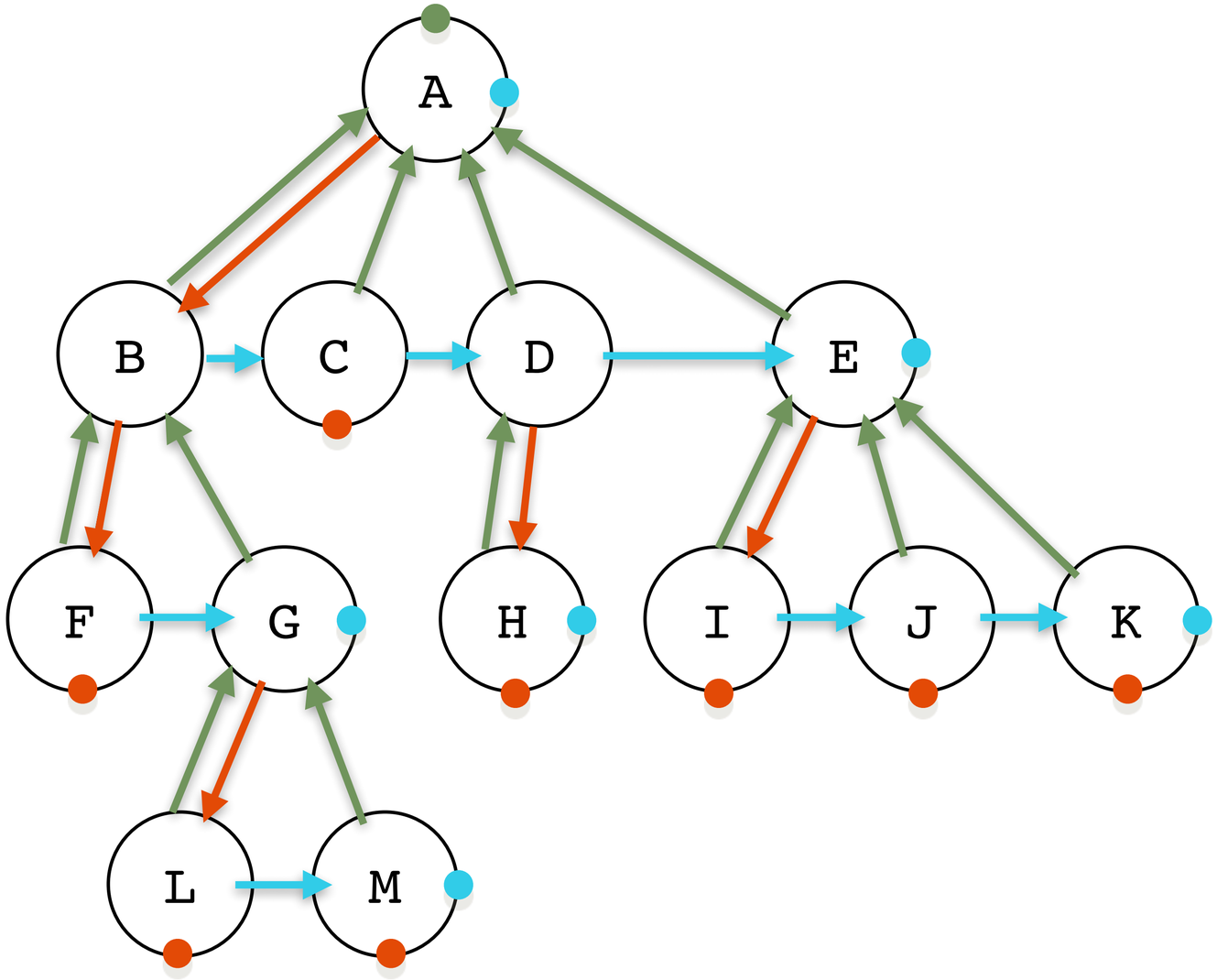


Itération en post-ordre

```
fonction début (r)
```

```
fonction fin(r)
```

```
fonction suivant(r)
```

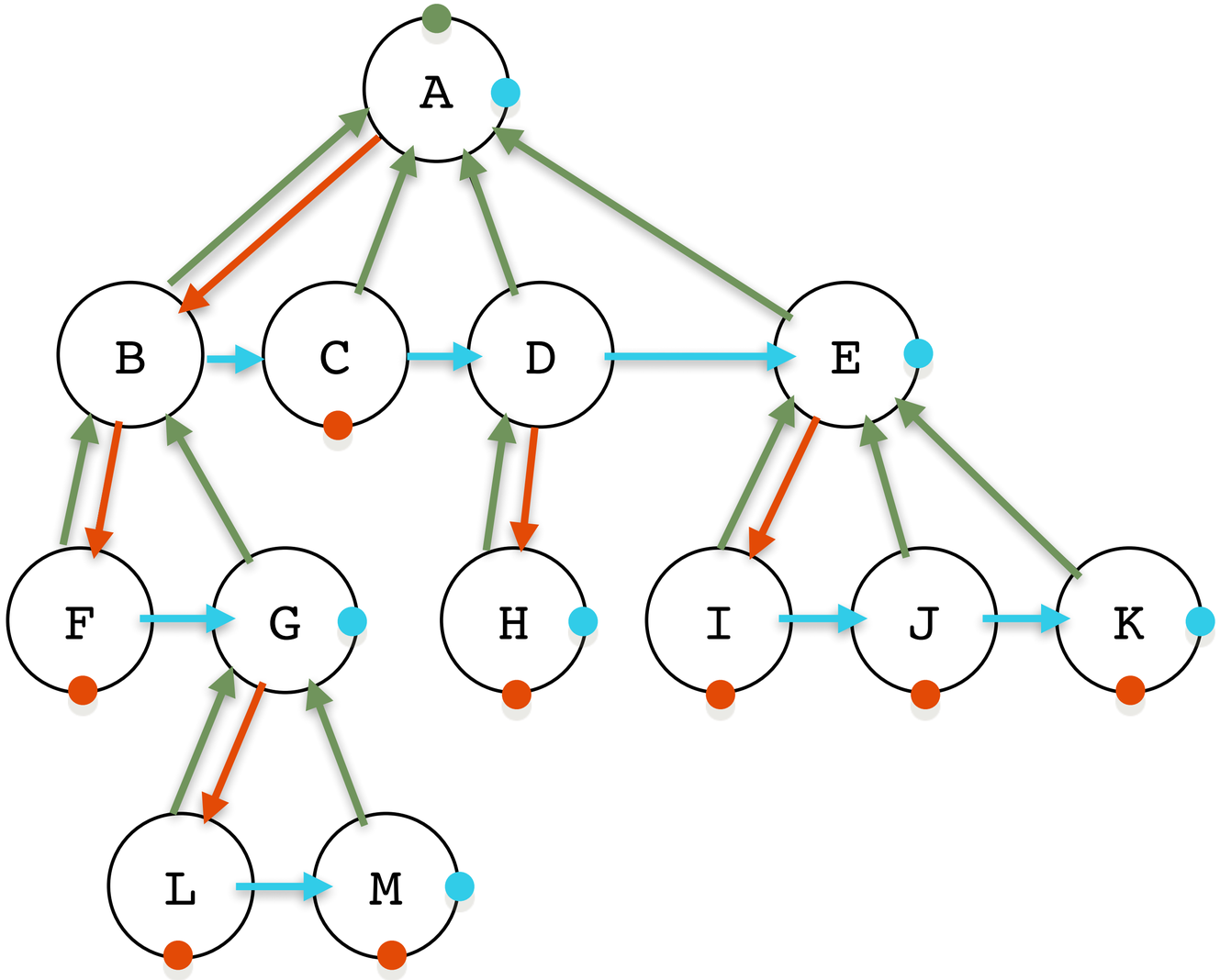


Itération en post-ordre

```
fonction début (r)
  si r == ∅, retourner ∅
  tant que r.ainé != ∅
     r ← r.ainé
  retourner r
```

```
fonction fin(r)
```

```
fonction suivant(r)
```

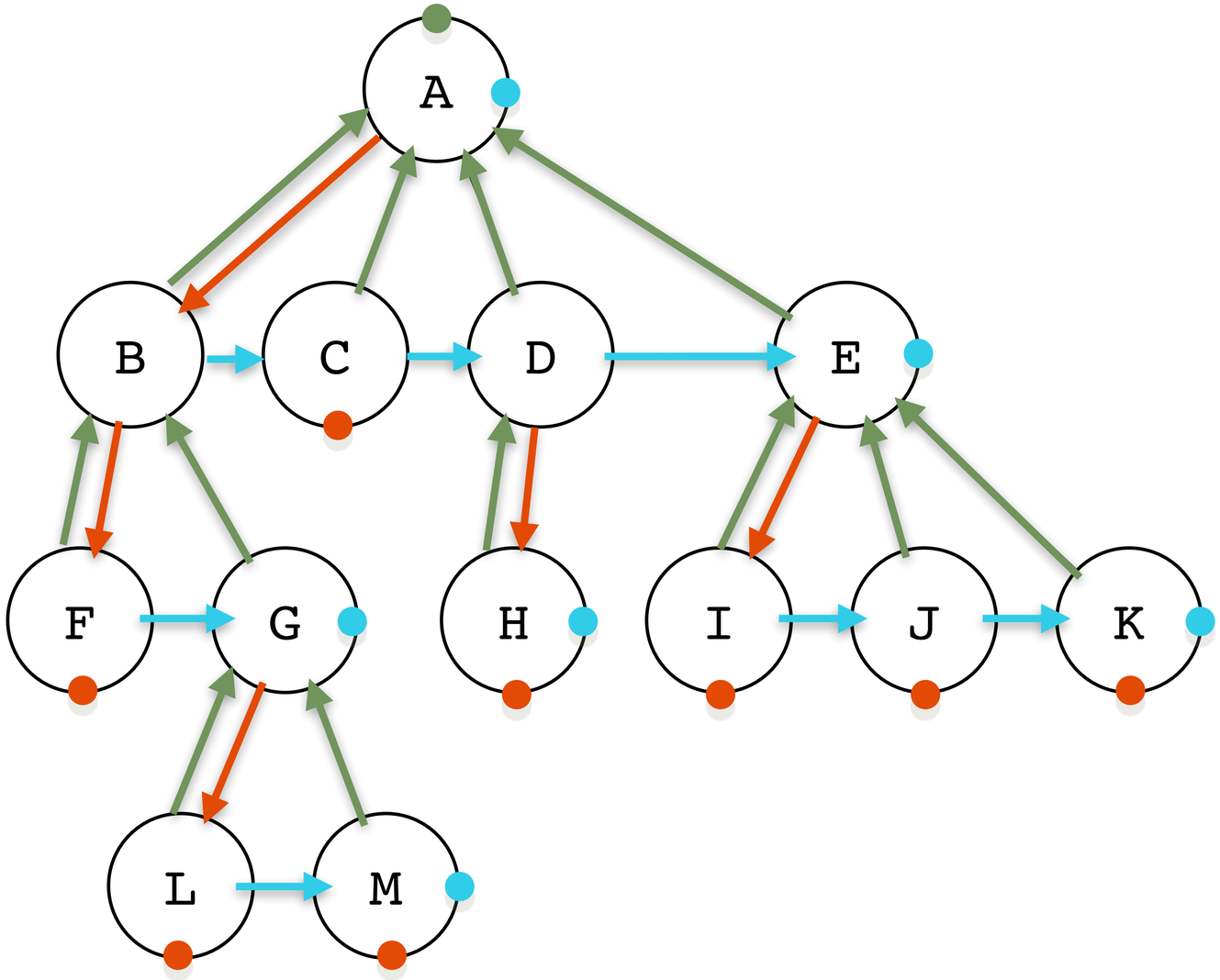


Itération en post-ordre

```
fonction début (r)
  si r == ∅, retourner ∅
  tant que r.ainé != ∅
     r ← r.ainé
  retourner r
```

```
fonction fin(r)
  retourner ∅
```

```
fonction suivant(r)
```

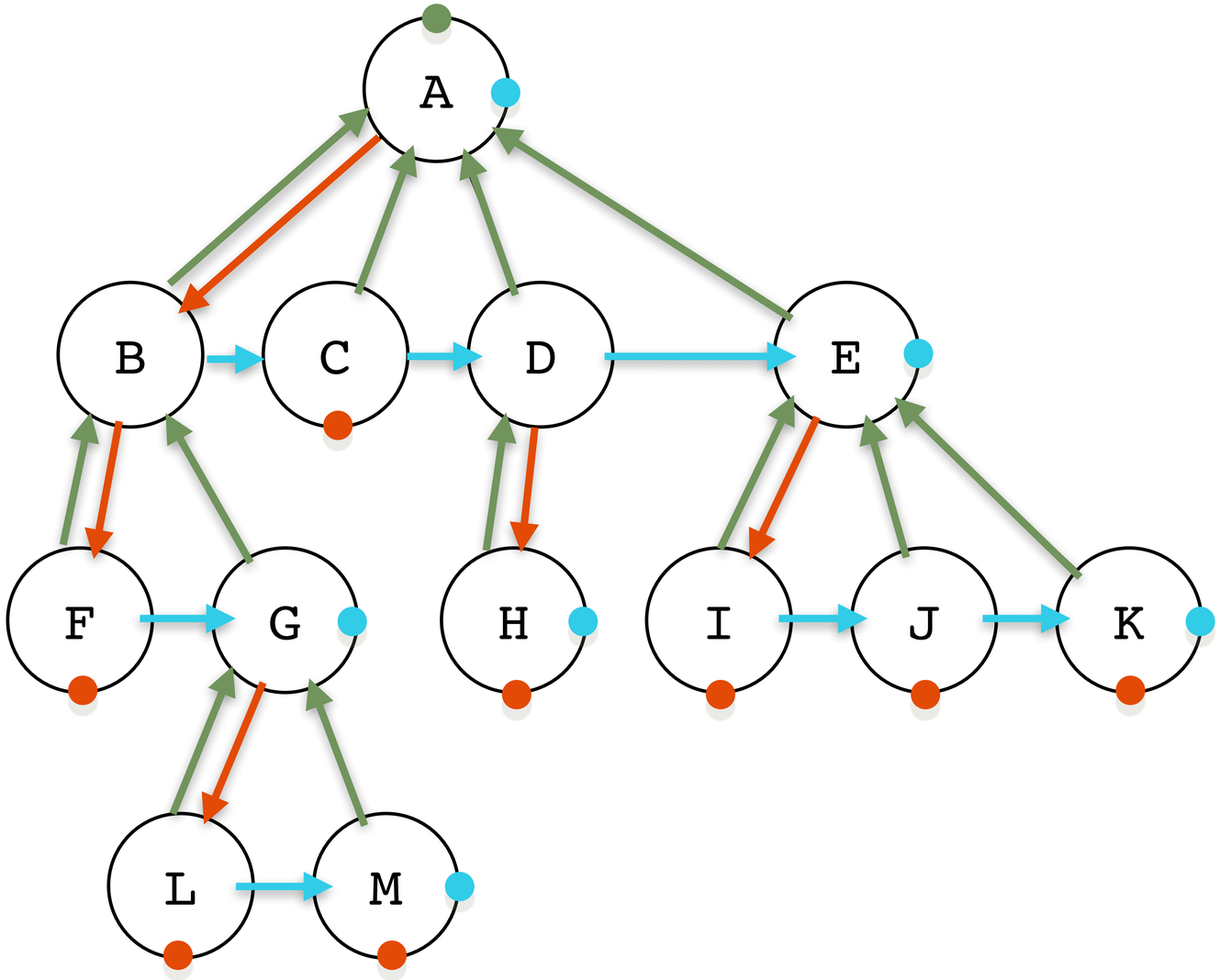


Itération en post-ordre

```
fonction début (r)
  si r == ∅, retourner ∅
  tant que r.ainé != ∅
     r ← r.ainé
  retourner r
```

```
fonction fin(r)
  retourner ∅
```

```
fonction suivant(r)
  si r.puiné != ∅
    retourner début(r.puiné)
```

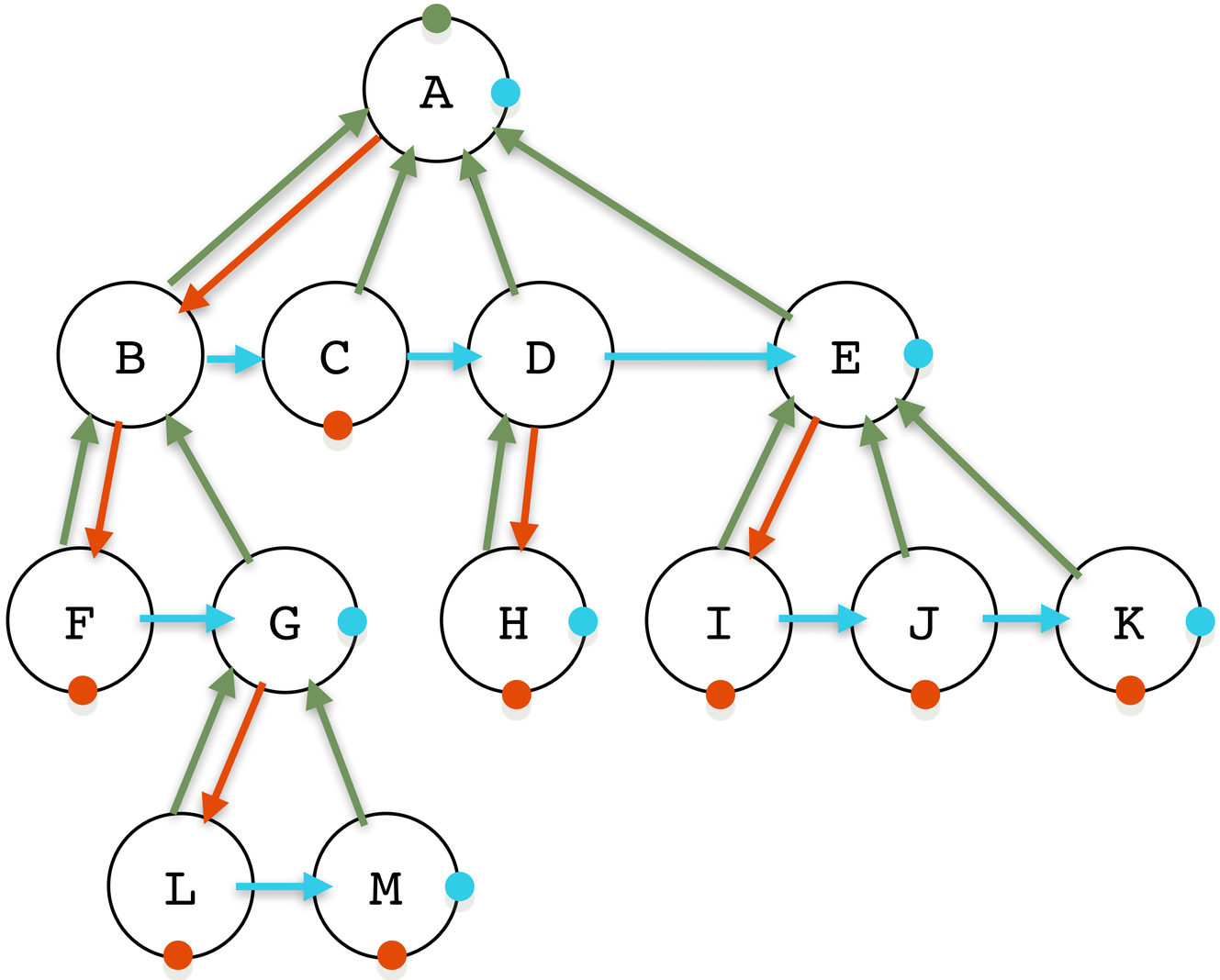


Itération en post-ordre

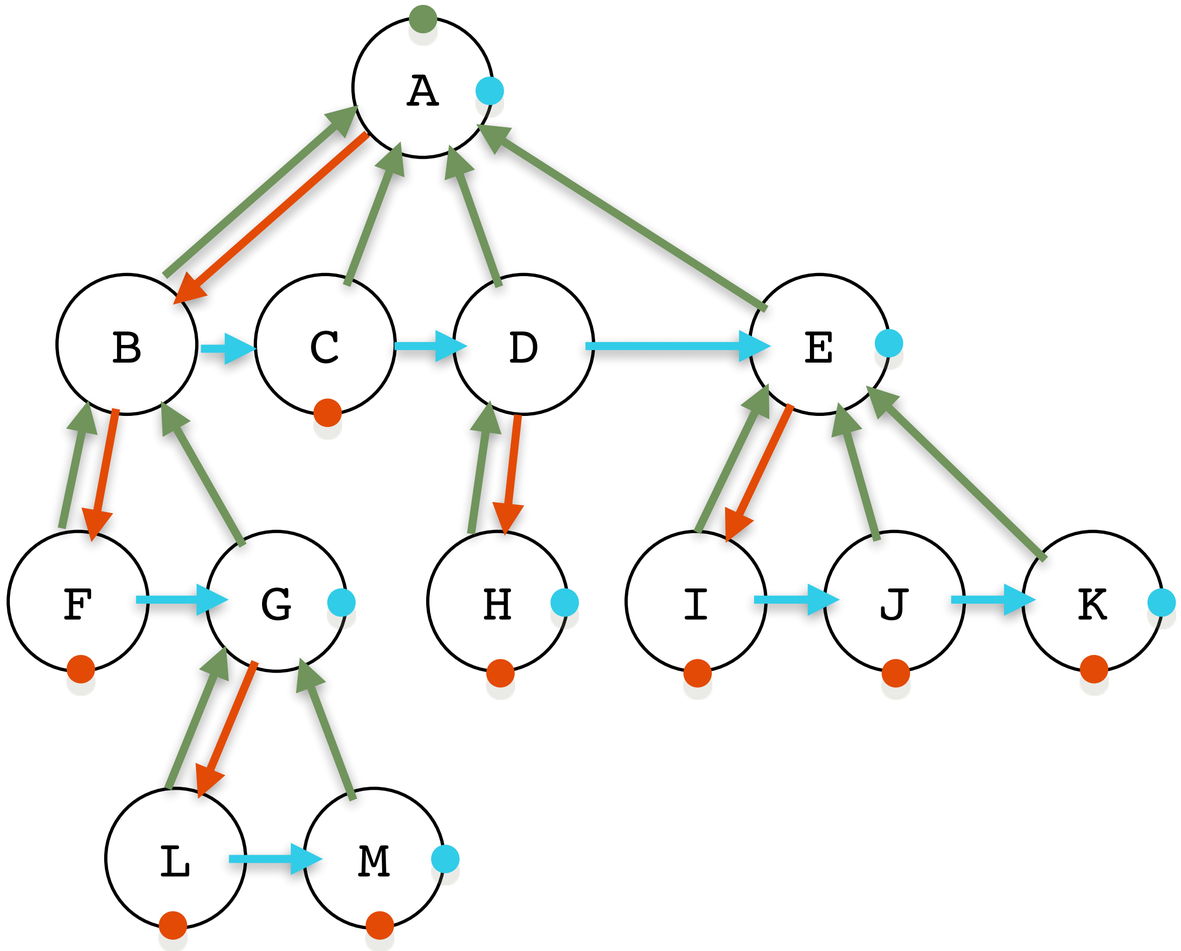
```
fonction début (r)
  si r == ∅, retourner ∅
  tant que r.ainé != ∅
     r ← r.ainé
  retourner r
```

```
fonction fin(r)
  retourner ∅
```

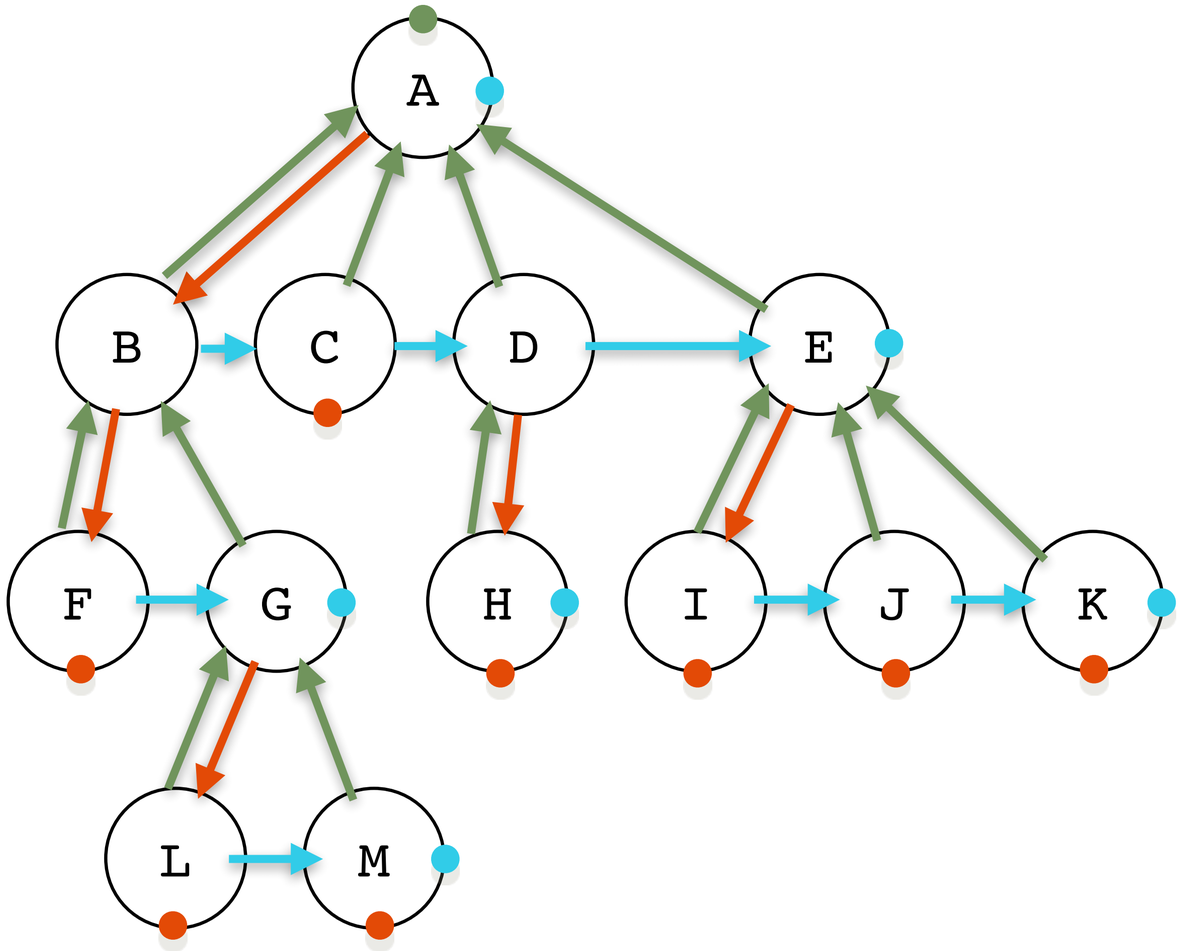
```
fonction suivant(r)
  si r.puiné != ∅
    retourner début(r.puiné)
  sinon
    retourner r.parent
```



Traçons ces 2 parcours

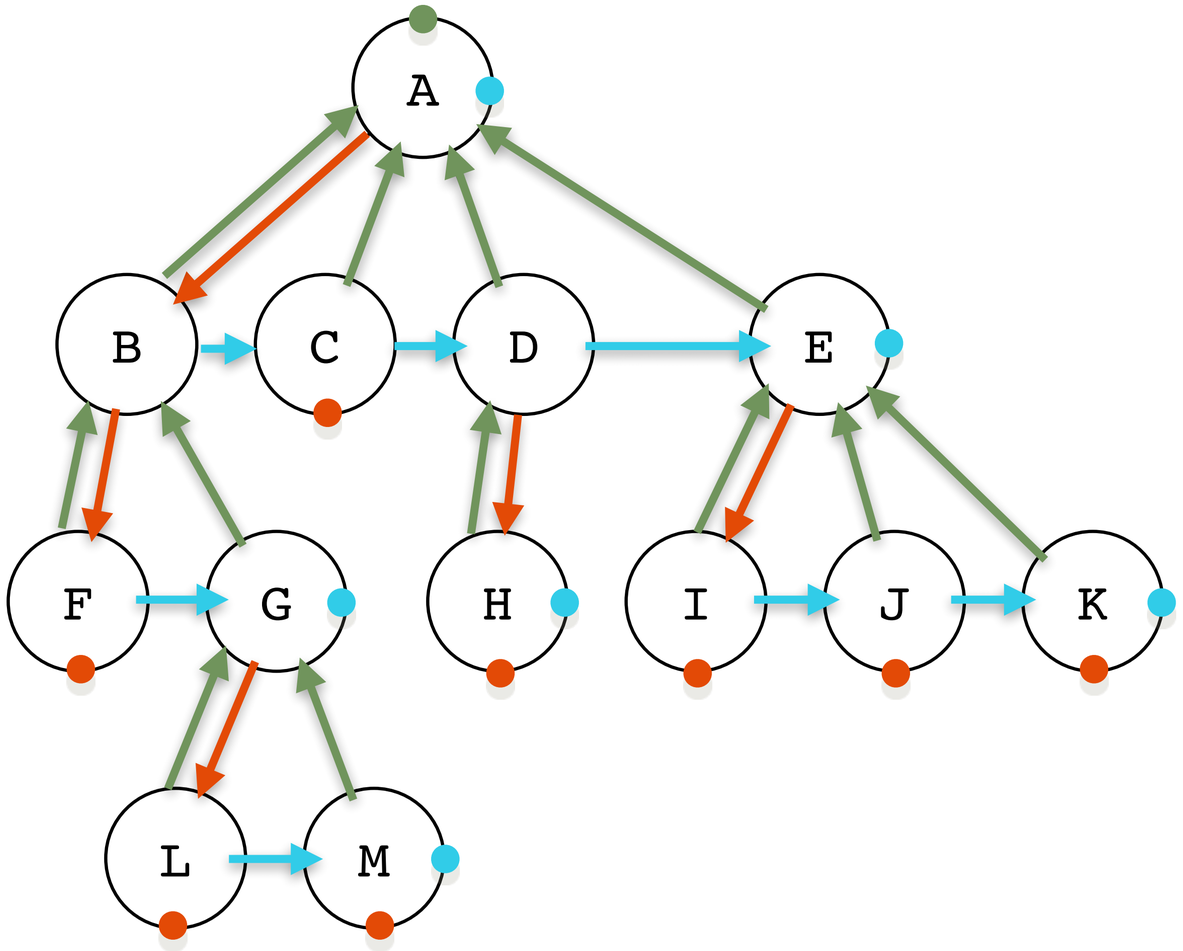


Traçons ces 2 parcours



Pré: A-B-F-G-L-M-G-B-C-D-H-D-E-I-J-K-E-A

Traçons ces 2 parcours



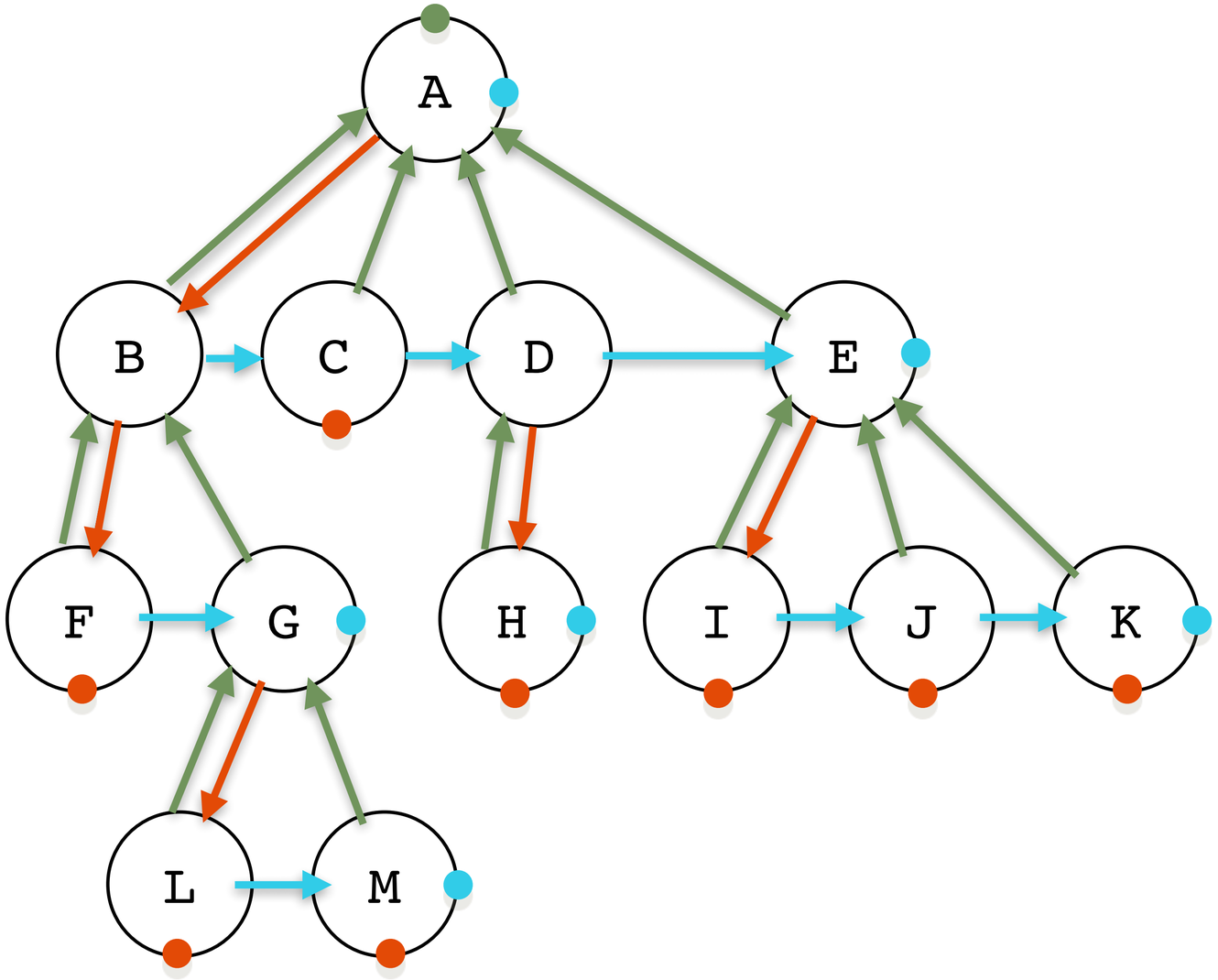
Pré: A-B-F-G-L-M-G-B-C-D-H-D-E-I-J-K-E-A

Post: A-B-F-G-L-M-G-B-C-D-H-D-E-I-J-K-E-A

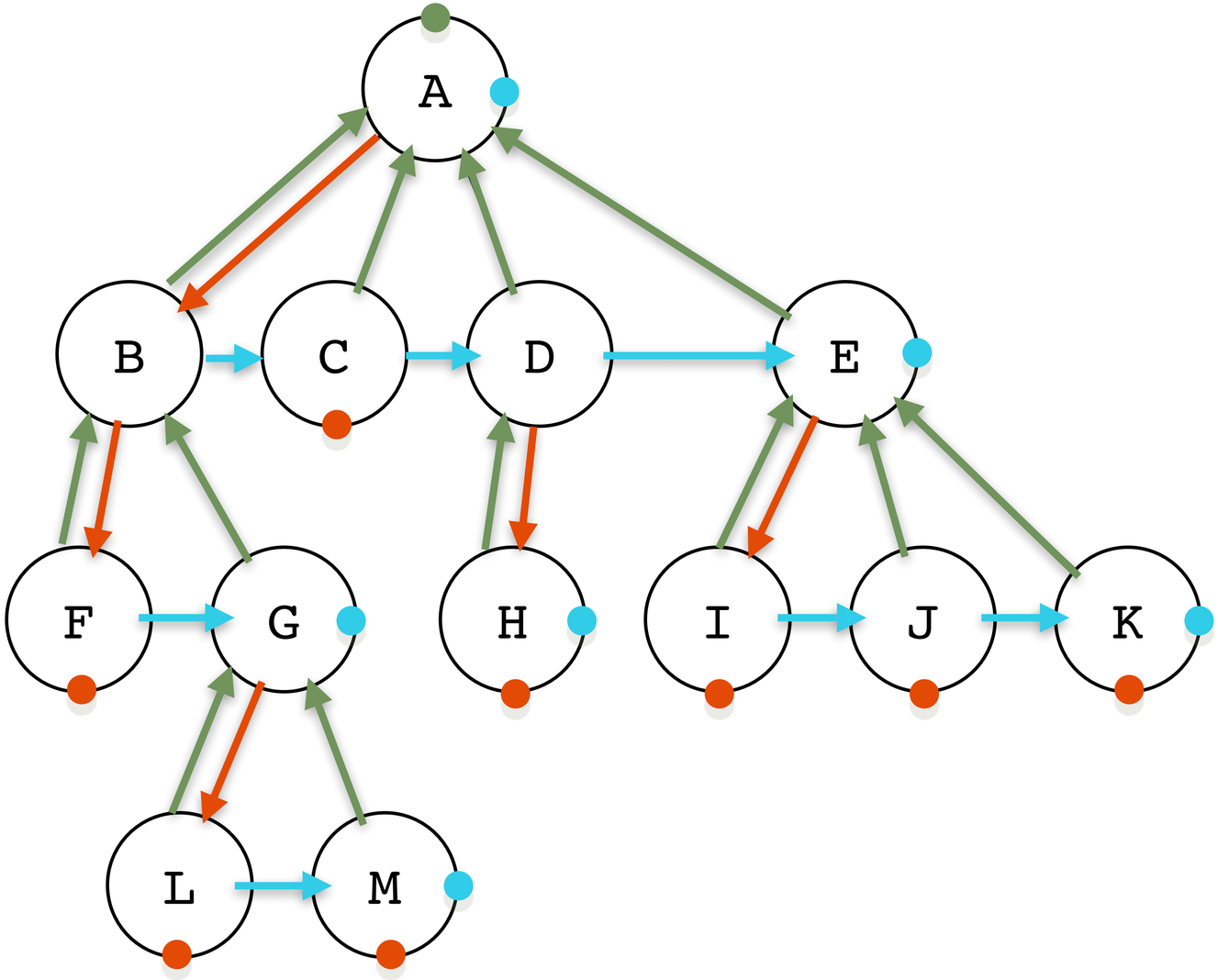
Itération en largeur

Comment aller du noeud K au noeud L en itérant, donc en ne suivant que des liens disponibles ci-contre ?

K-E-A-B-F-G-L



Itération en largeur (2)



Conclusion : on n’itère pas en largeur

Itération en largeur (2)

A-B-C-D-E-A-B-F-G-

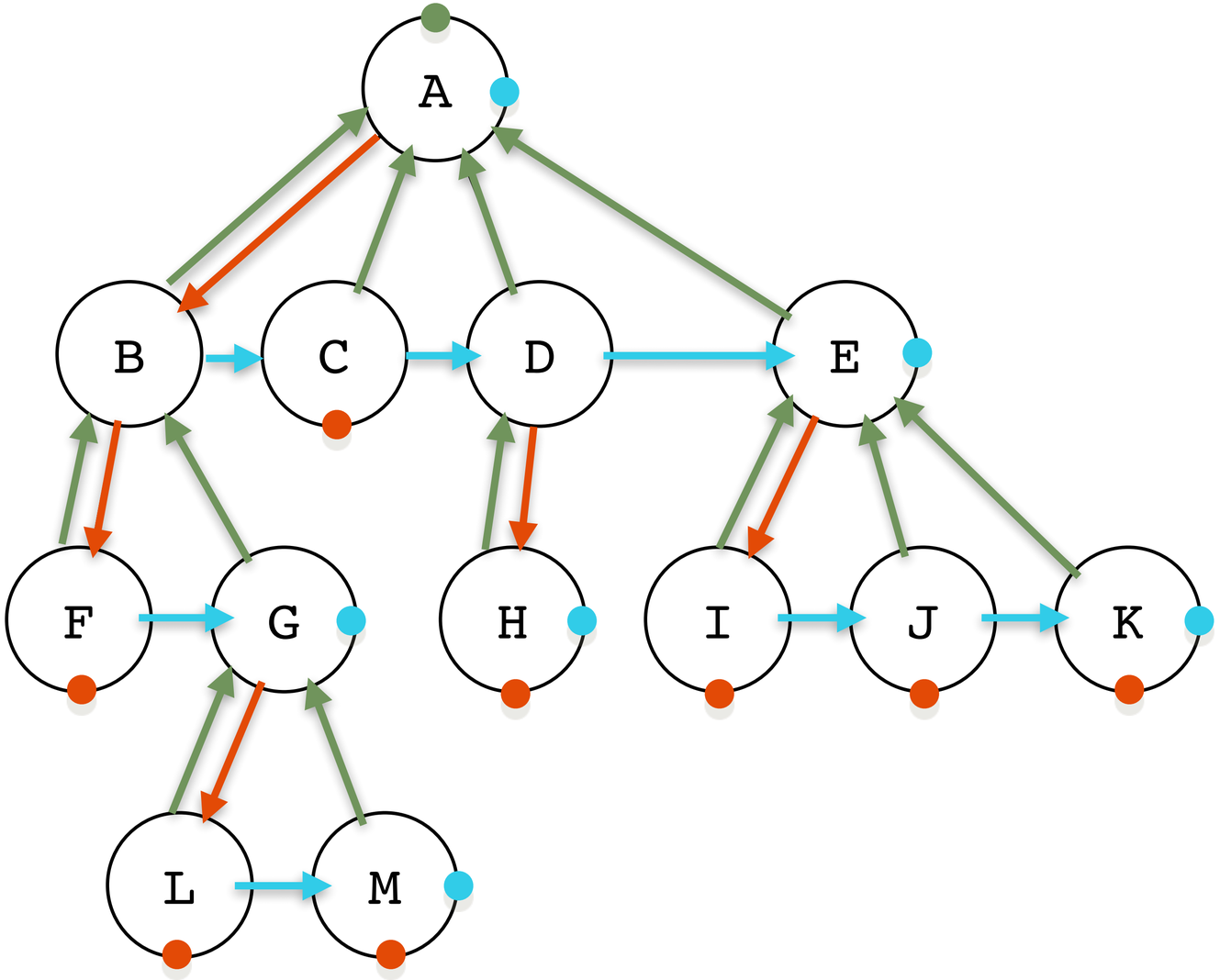
B-C-D-H-D-E-I-J-K-

E-A-B-F-G-L-M-G-B-

C-D-H-D-E-I-J-K-E-

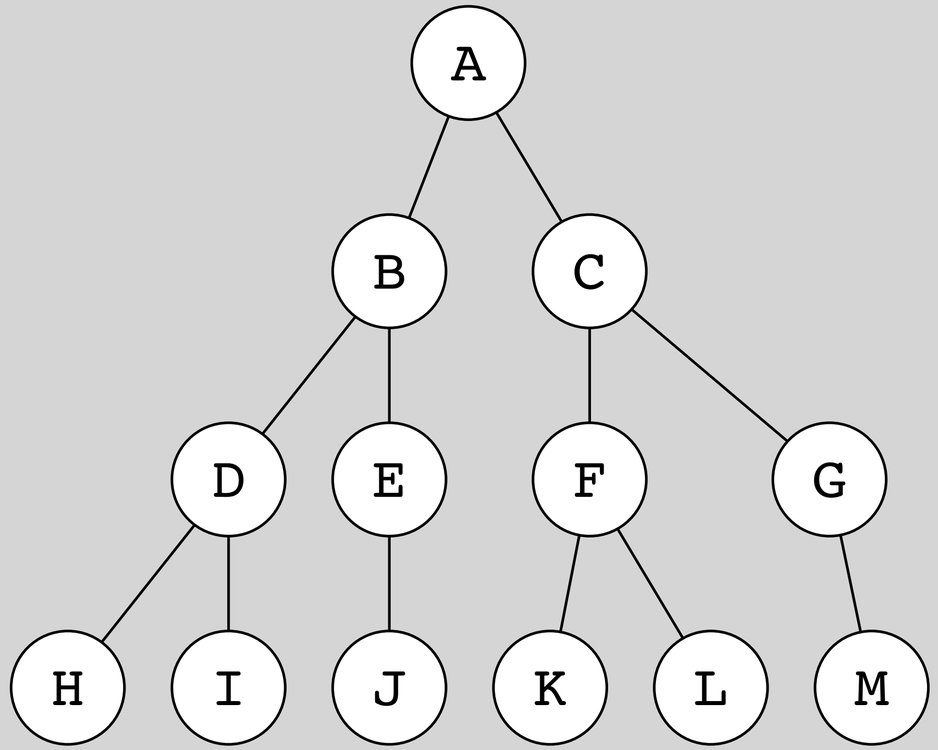
A-B-F-G-L-M-G-B-C-

D-H-D-E-I-J-K-E-A



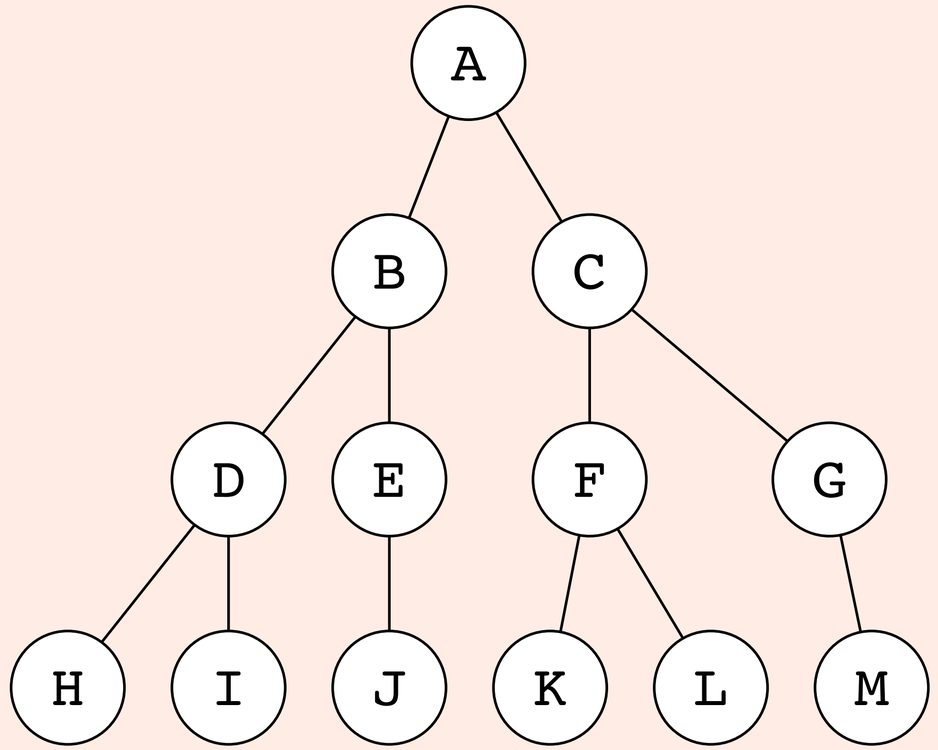
Conclusion : on n'itère pas en largeur

Exercices



- Quel noeud suit J en pré-ordre ?
- Quel noeud suit K en pré-ordre ?
- Quel noeud suit J en post-ordre ?
- Quel noeud suit B en post-ordre ?
- Tracez les noeuds par lesquels passe l'itération en pré-ordre sur cet arbre

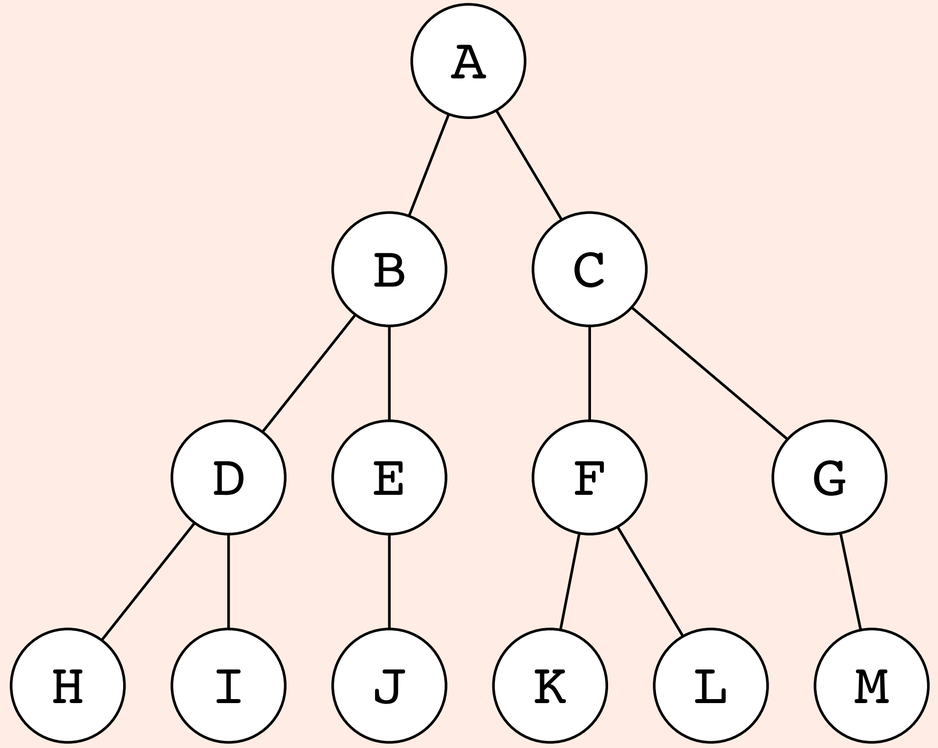
Solutions



- Quel noeud suit J en pré-ordre ?
- Quel noeud suit K en pré-ordre ?
- Quel noeud suit J en post-ordre ?
- Quel noeud suit B en post-ordre ?
- Tracez les noeuds par lesquels passe l'itération en pré-ordre sur cet arbre

Solutions

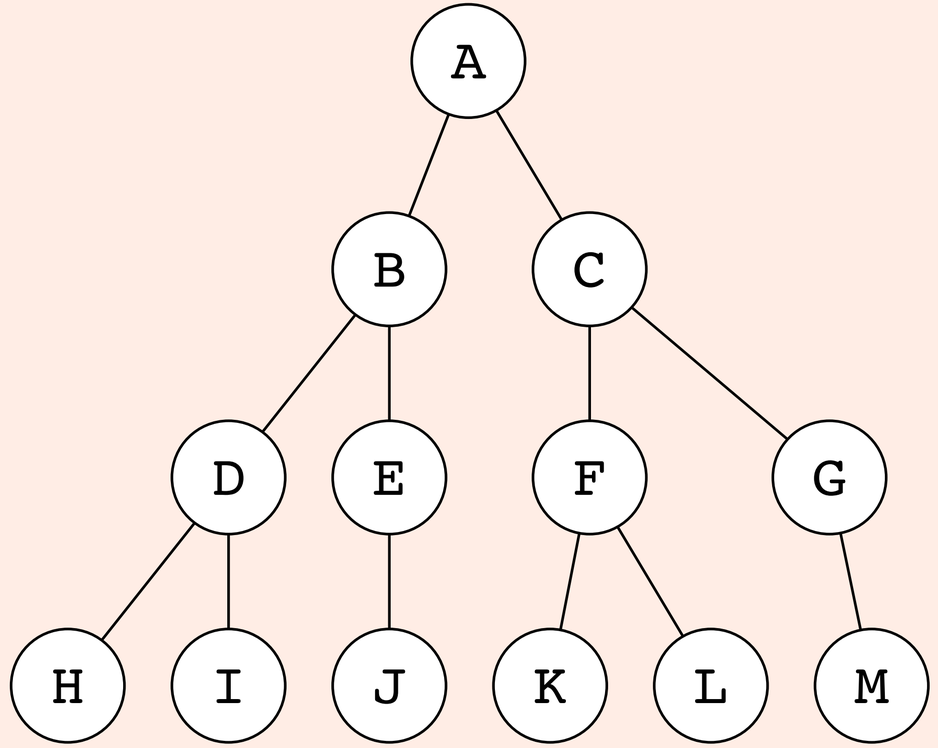
- Quel noeud suit J en pré-ordre ? C
- Quel noeud suit K en pré-ordre ?
- Quel noeud suit J en post-ordre ?
- Quel noeud suit B en post-ordre ?



- Tracez les noeuds par lesquels passe l'itération en pré-ordre sur cet arbre

Solutions

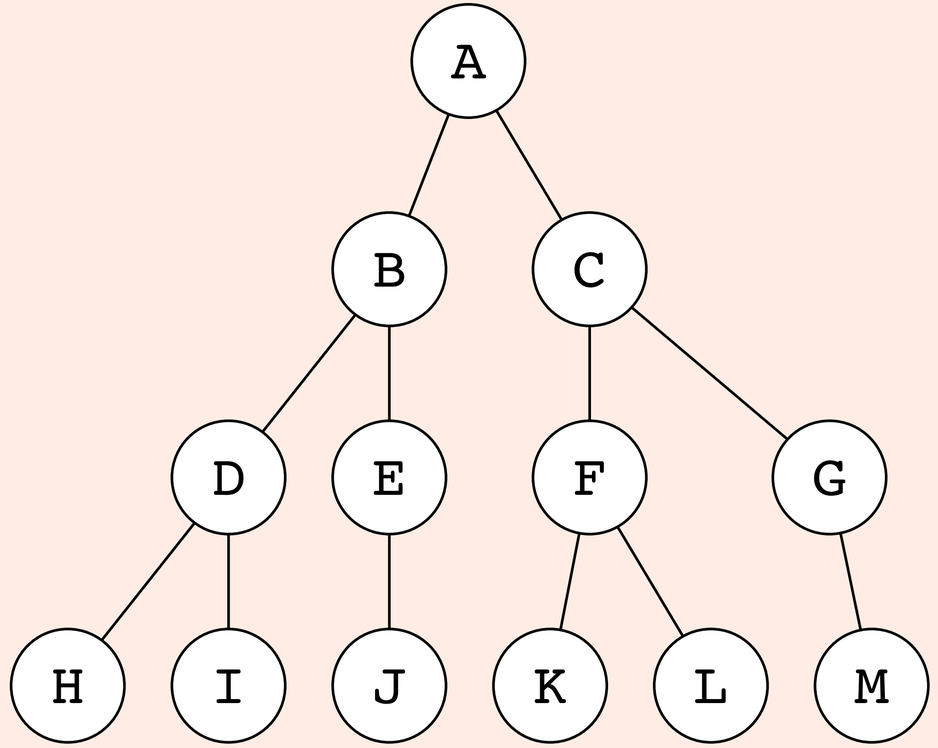
- Quel noeud suit J en pré-ordre ? C
- Quel noeud suit K en pré-ordre ? L
- Quel noeud suit J en post-ordre ?
- Quel noeud suit B en post-ordre ?



- Tracez les noeuds par lesquels passe l'itération en pré-ordre sur cet arbre

Solutions

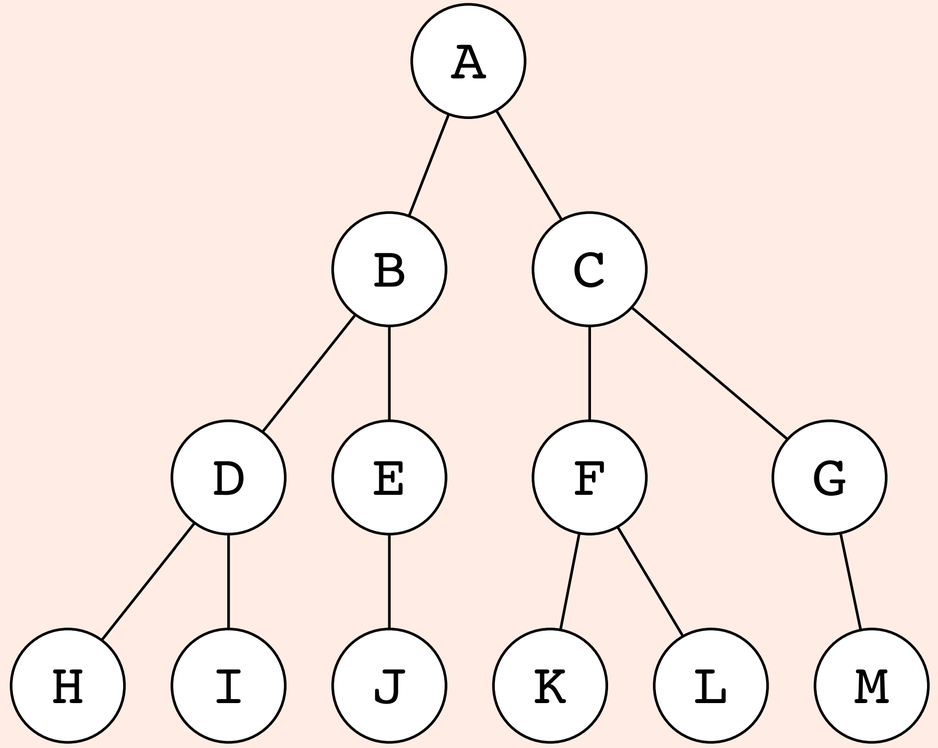
- Quel noeud suit J en pré-ordre ? C
- Quel noeud suit K en pré-ordre ? L
- Quel noeud suit J en post-ordre ? E
- Quel noeud suit B en post-ordre ?



- Tracez les noeuds par lesquels passe l'itération en pré-ordre sur cet arbre

Solutions

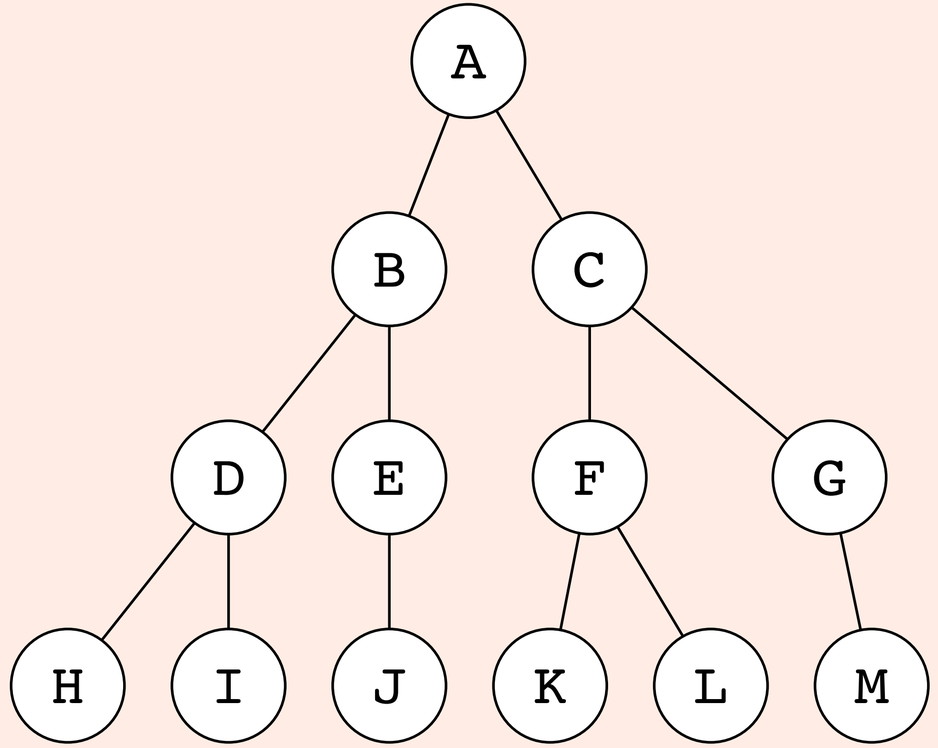
- Quel noeud suit J en pré-ordre ? C
- Quel noeud suit K en pré-ordre ? L
- Quel noeud suit J en post-ordre ? E
- Quel noeud suit B en post-ordre ? K



- Tracez les noeuds par lesquels passe l'itération en pré-ordre sur cet arbre

Solutions

- Quel noeud suit J en pré-ordre ? C
- Quel noeud suit K en pré-ordre ? L
- Quel noeud suit J en post-ordre ? E
- Quel noeud suit B en post-ordre ? K

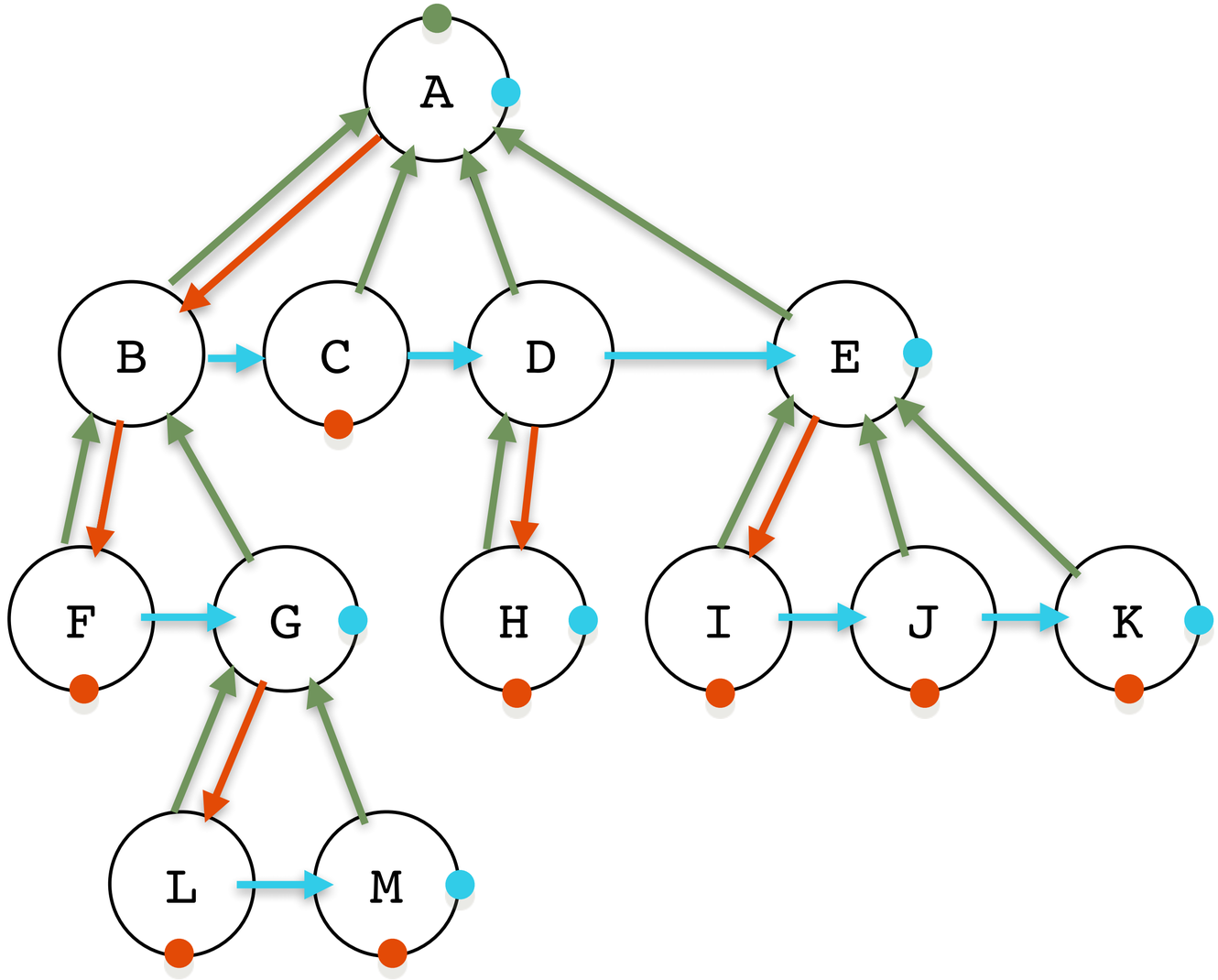


- Tracez les noeuds par lesquels passe l'itération en pré-ordre sur cet arbre A-B-D-H-I-D-E-J-E-B-C-F-K-L-F-G-M-G-C-A

6. et encore…

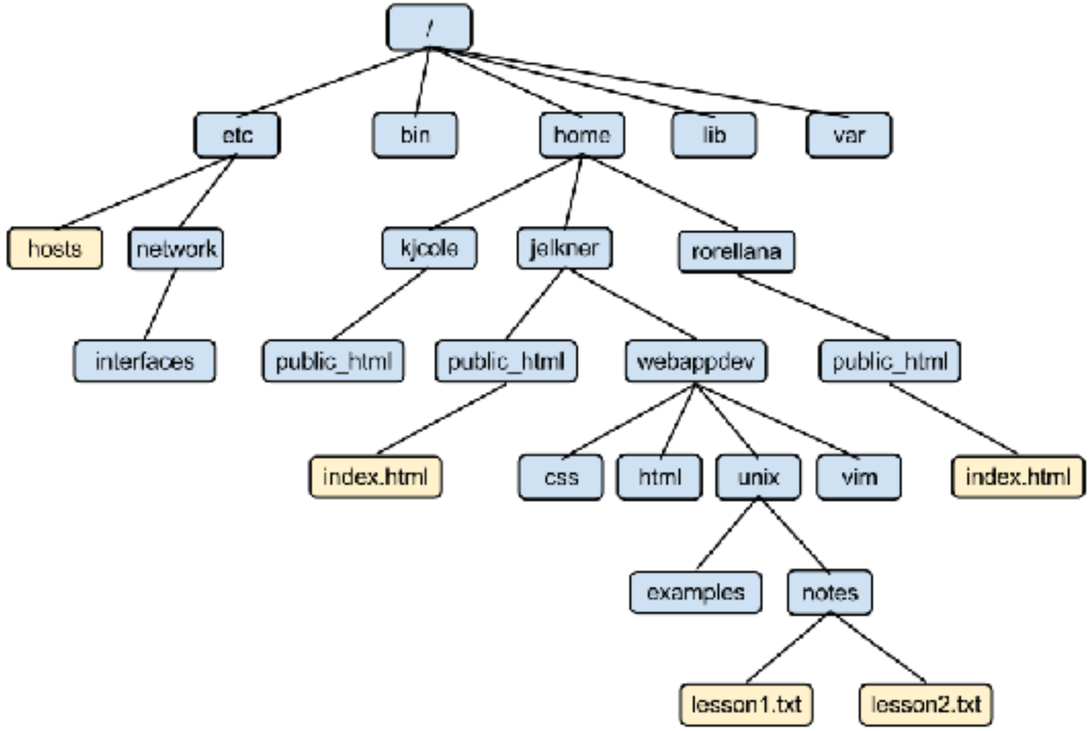
Insertion / suppression

- Dépendent fortement de la structure de Noeud utilisée
- En cas de suppression, que fait-on des enfants du noeud supprimé ?



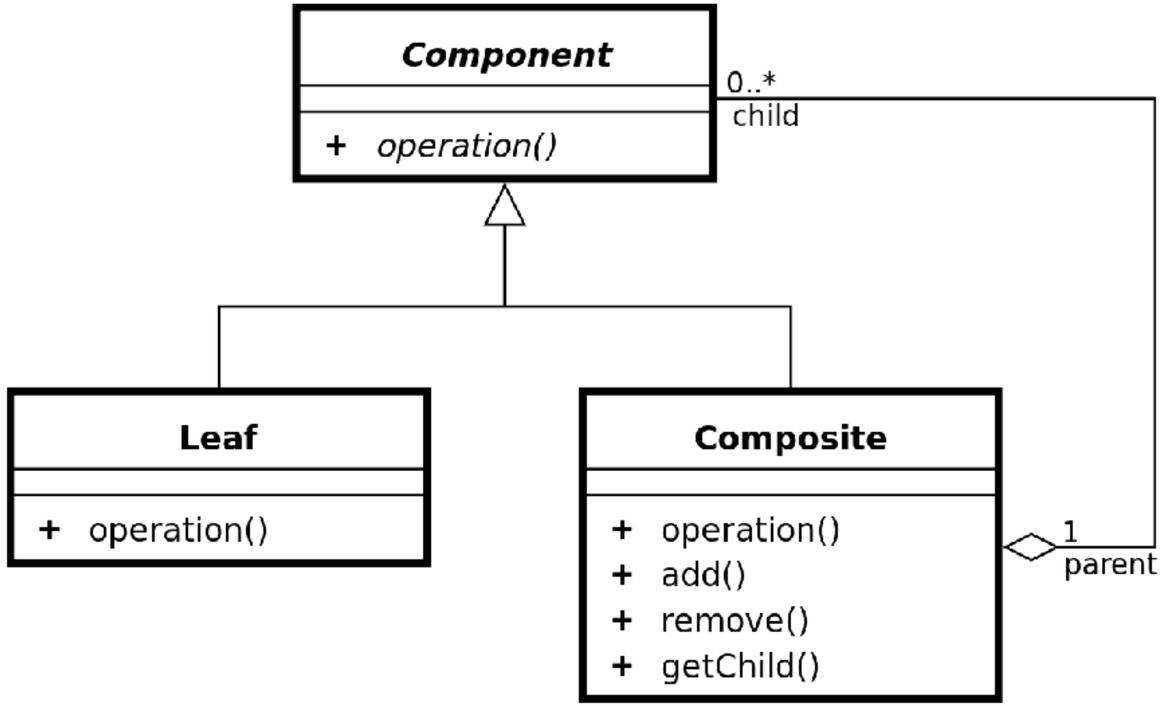
Noeuds de types différents

- Dans l'arbre d'un système de fichier, on a 2 types de noeuds
  - Répertoires
  - Fichiers



Composite Design Pattern

- En deuxième année
  - Programmation Orientée Objet
  - Modèles de Conception Réutilisables
  - Génie Logiciel

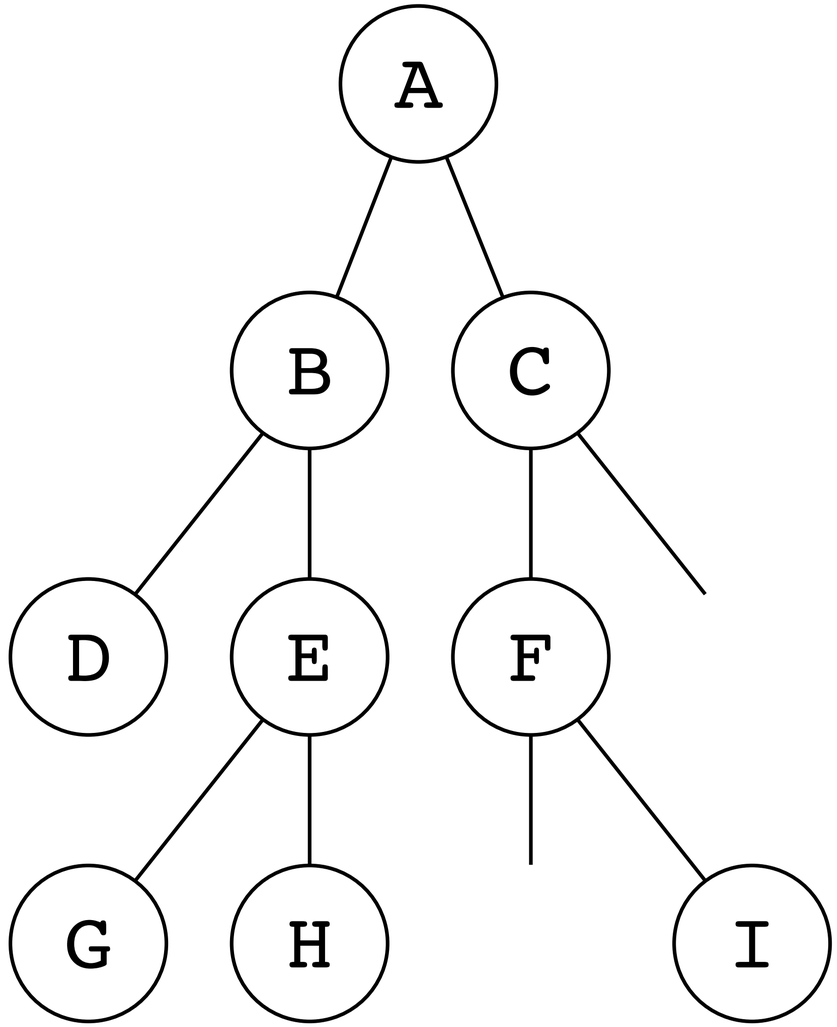


HEIG VD

7. Arbres binaires

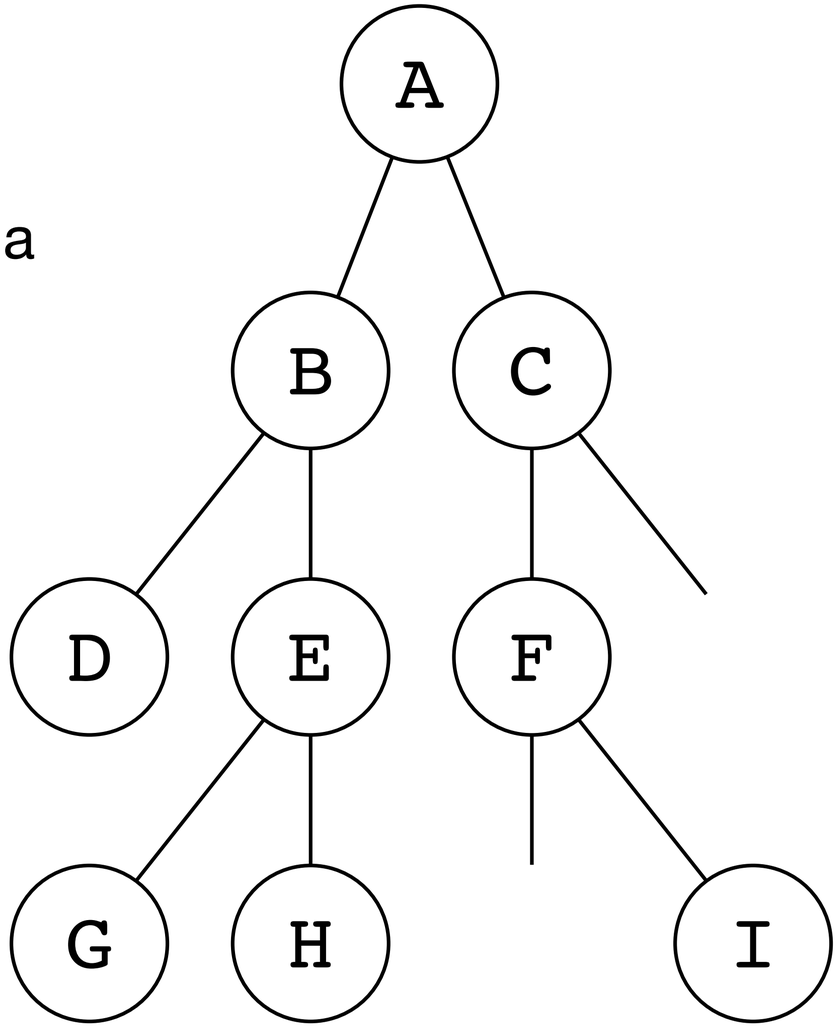
Binaires, pas quelconques

- Degré 2, mais aussi …



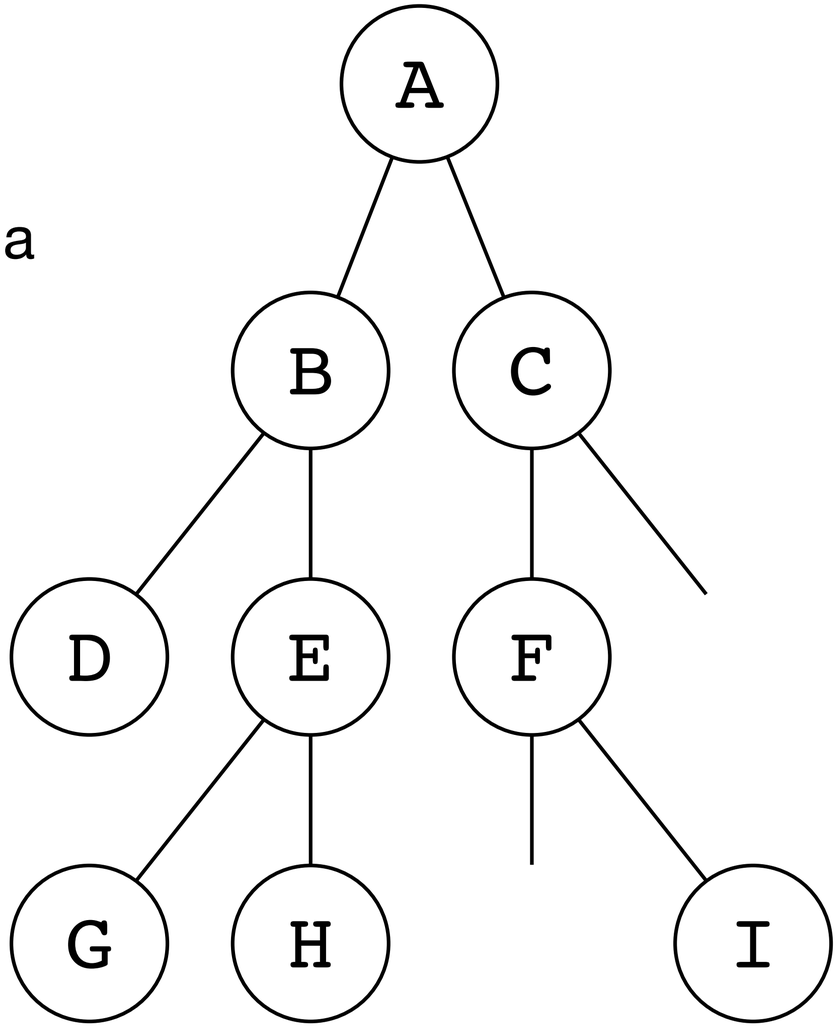
Binaires, pas quelconques

- Degré 2, mais aussi …
  - … enfants explicitement gauche ou droit, donc il y a 2 sortes de noeuds de degré 1



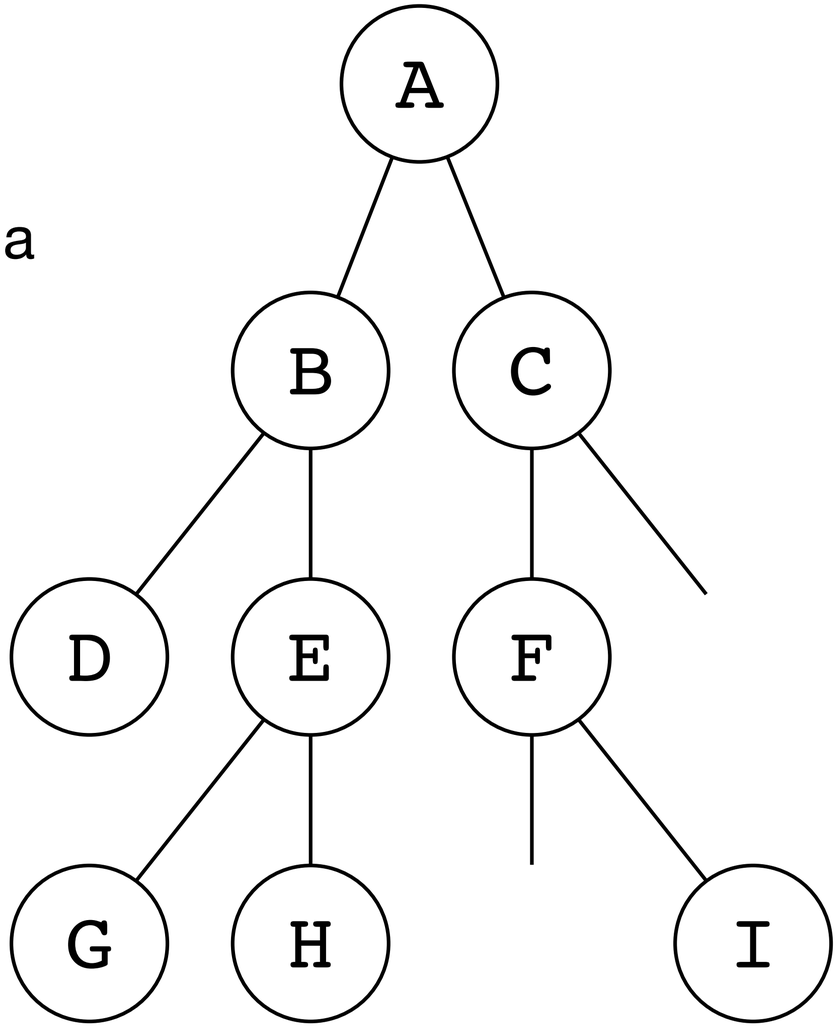
Binaires, pas quelconques

- Degré 2, mais aussi …
  - … enfants explicitement gauche ou droit, donc il y a 2 sortes de noeuds de degré 1
- Exemples



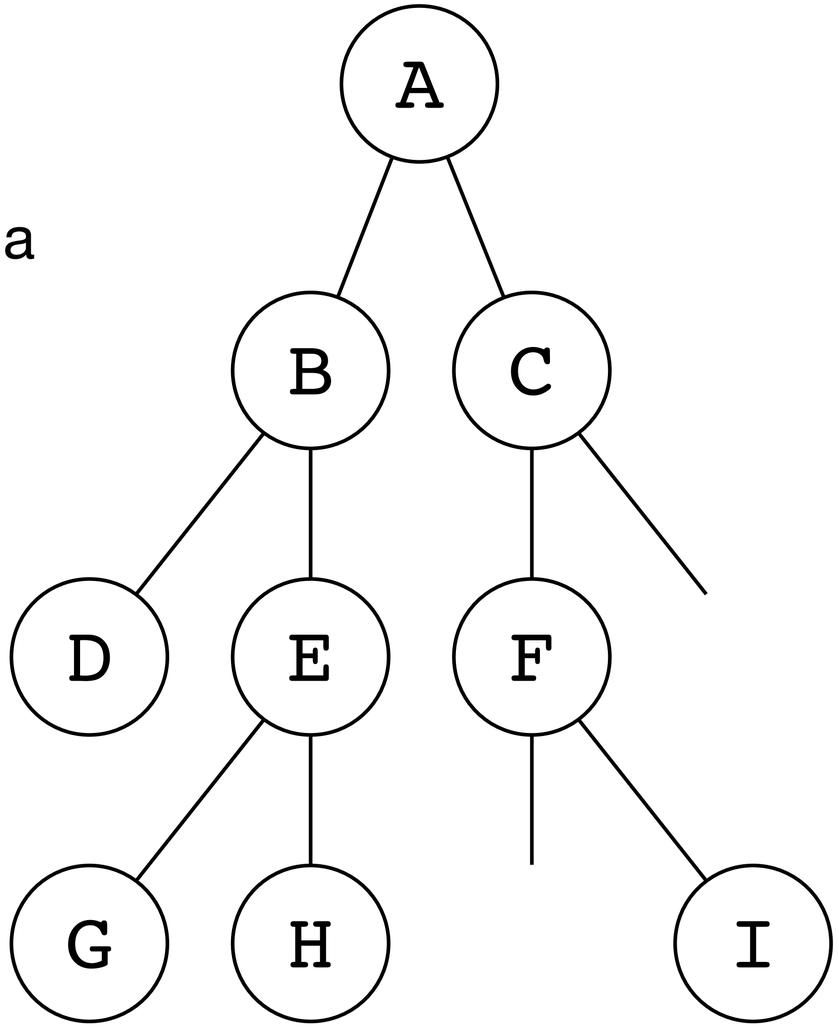
Binaires, pas quelconques

- Degré 2, mais aussi …
  - … enfants explicitement gauche ou droit, donc il y a 2 sortes de noeuds de degré 1
- Exemples
  - Tas (arbre binaire complet),



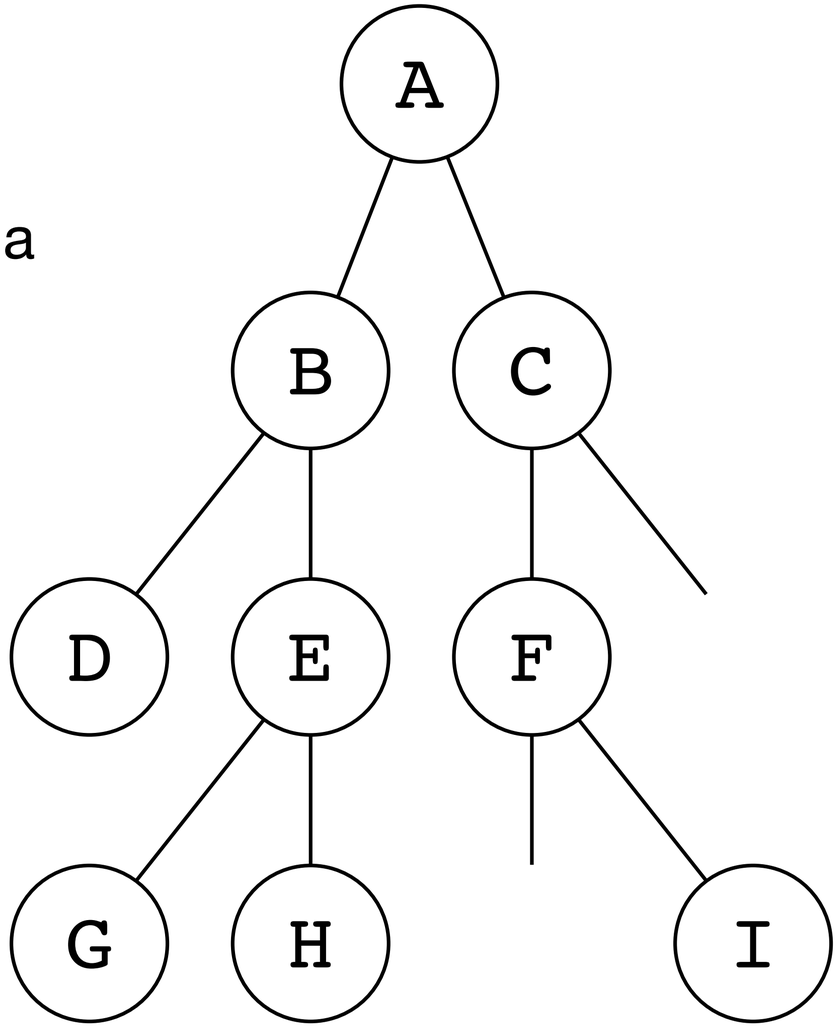
Binaires, pas quelconques

- Degré 2, mais aussi …
  - … enfants explicitement gauche ou droit, donc il y a 2 sortes de noeuds de degré 1
- Exemples
  - Tas (arbre binaire complet),
  - arbre binaires de recherche (ASD1),



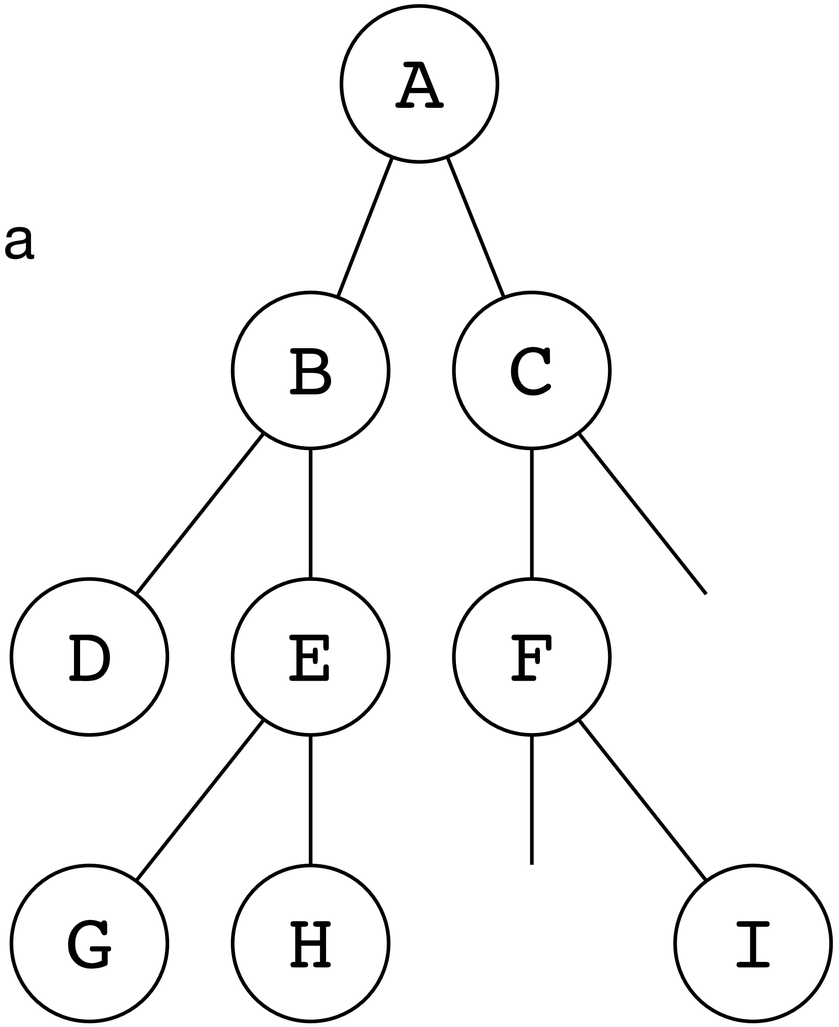
Binaires, pas quelconques

- Degré 2, mais aussi …
  - … enfants explicitement gauche ou droit, donc il y a 2 sortes de noeuds de degré 1
- Exemples
  - Tas (arbre binaire complet),
  - arbre binaires de recherche (ASD1),
  - arbres AVL (ASD2),



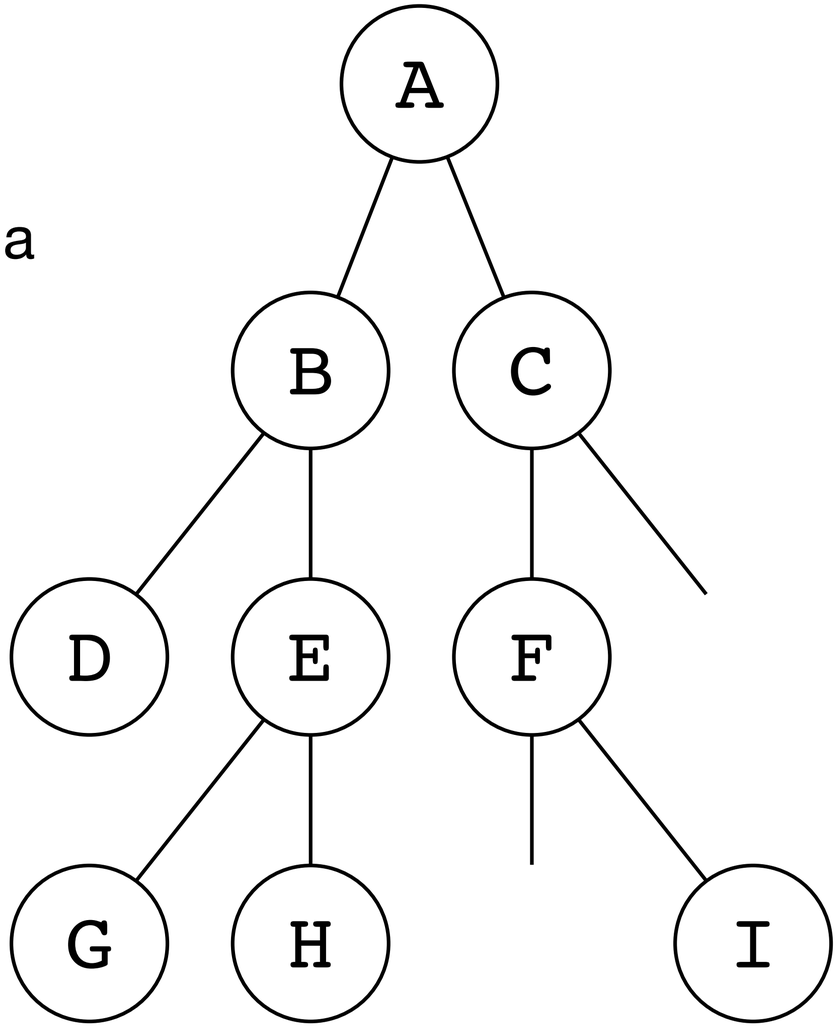
Binaires, pas quelconques

- Degré 2, mais aussi …
  - … enfants explicitement gauche ou droit, donc il y a 2 sortes de noeuds de degré 1
- Exemples
  - Tas (arbre binaire complet),
  - arbre binaires de recherche (ASD1),
  - arbres AVL (ASD2),
  - arbres rouge-noir (STL),



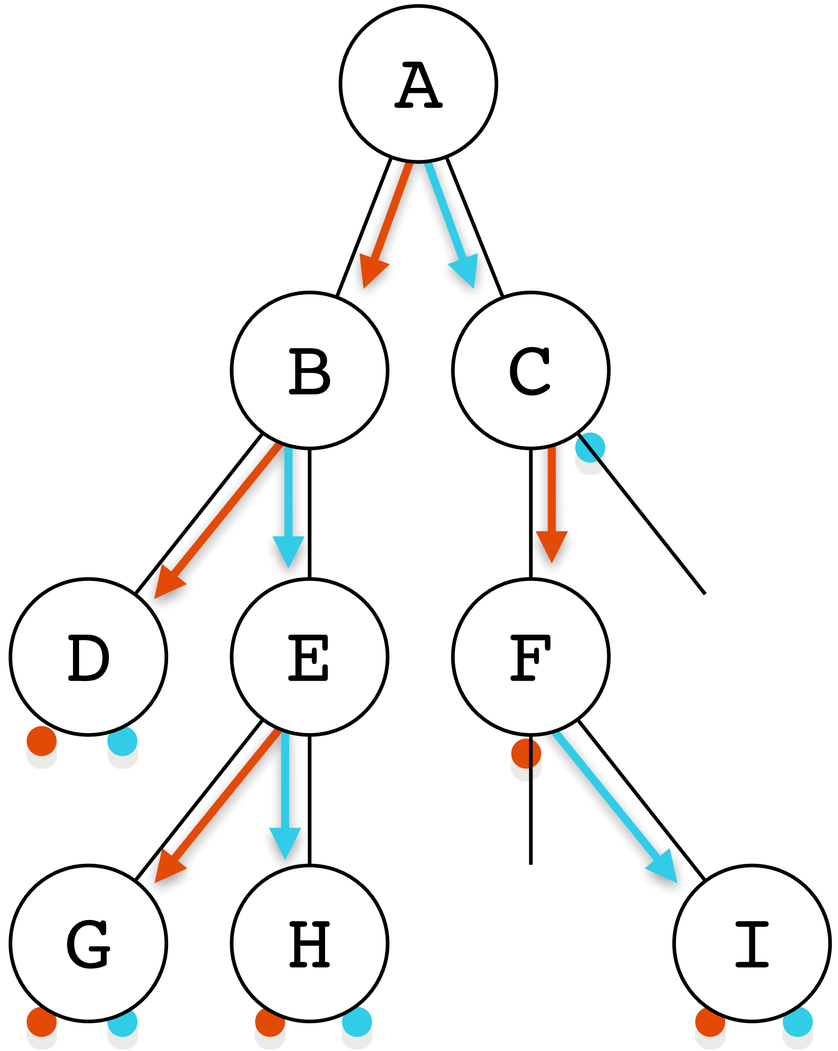
Binaires, pas quelconques

- Degré 2, mais aussi …
  - … enfants explicitement gauche ou droit, donc il y a 2 sortes de noeuds de degré 1
- Exemples
  - Tas (arbre binaire complet),
  - arbre binaires de recherche (ASD1),
  - arbres AVL (ASD2),
  - arbres rouge-noir (STL),
  - arbres d'Andersson, …



Représentation

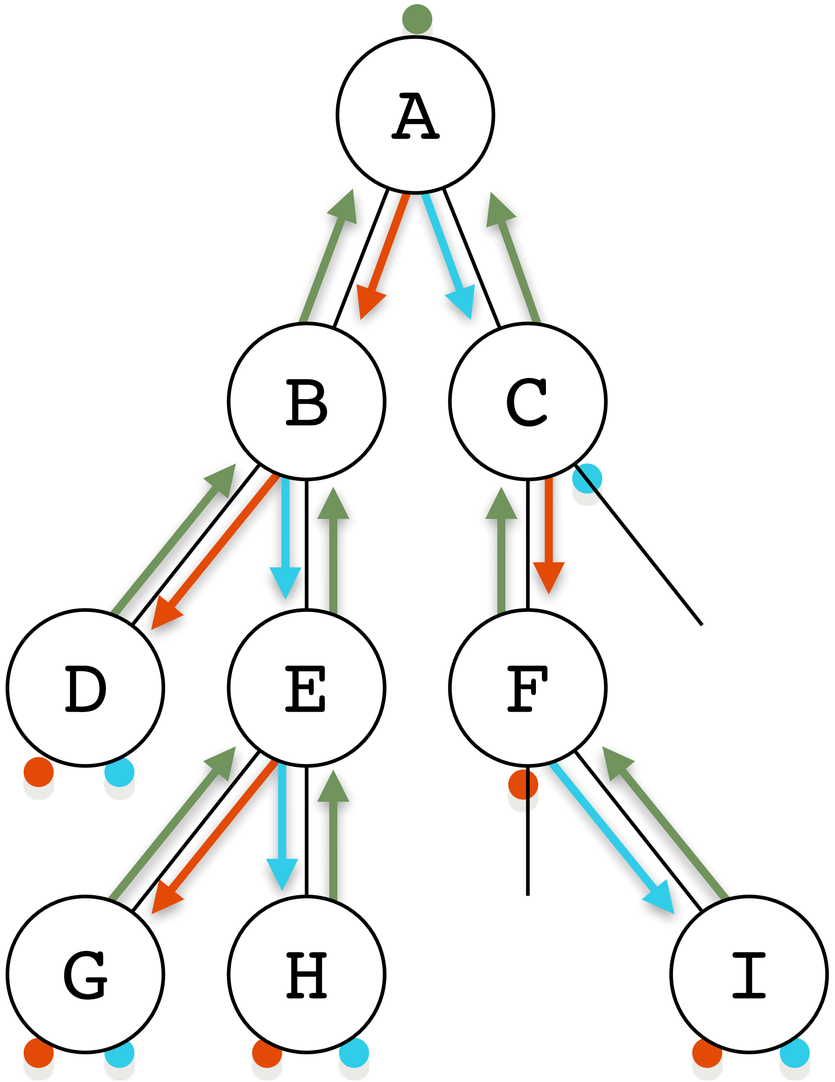
```
structure Noeud<T>
    T étiquette
    Noeud* gauche
    Noeud* droit
```



- Liens explicites vers les enfants gauche et droit

Représentation

```
structure Noeud<T>
    T étiquette
    Noeud* gauche
    Noeud* droit
    Noeud* parent
```



- Liens explicites vers les enfants gauche et droit
- Lien optionnel vers le parent

Représentations en C++

```
template <typename T>
struct Node
{
    T label;
    Node* parent;
    Node* left;
    Node* right;
};
```

Représentations en C++

```
template <typename T>
struct Node
{
    T label;
    Node* parent;
    Node* left;
    Node* right;
};
```

```
template <typename T>
struct Node
{
    T label;
    Node* parent;
    array<Node*,2> children;
};
```

Représentations en C++

```
template <typename T>
struct Node
{
    T label;
    Node* parent;
    Node* left;
    Node* right;
};
```

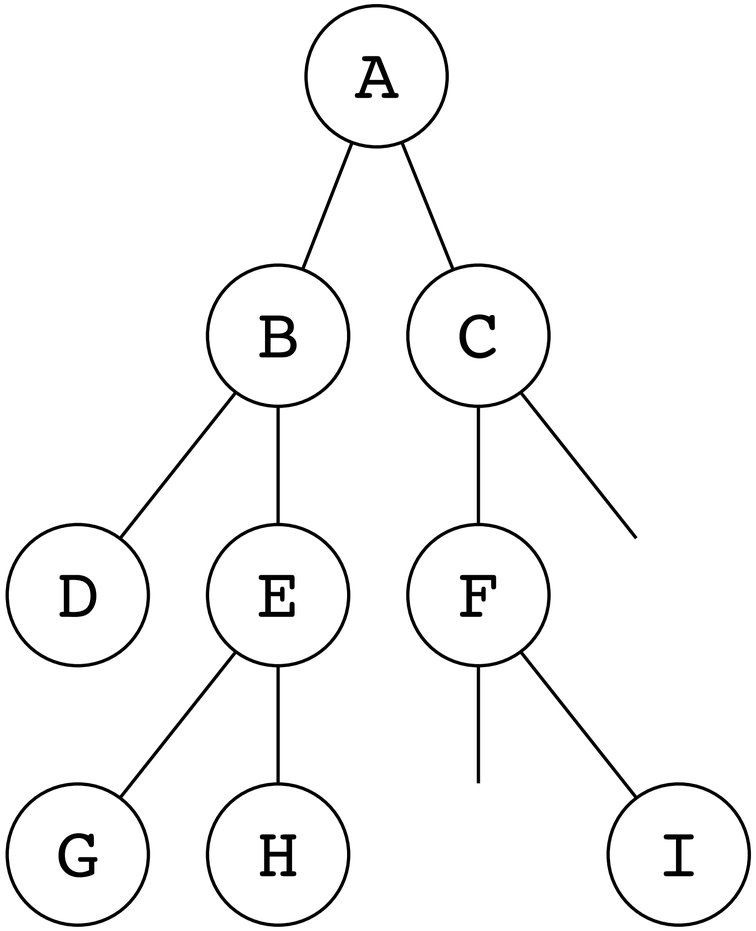
```
template <typename T>
struct Node
{
    T label;
    Node* parent;
    array<Node*,2> children;
};
```

```
template <typename T> struct Node
{
    T label;
    Node* parent;
    array<Node*,2> children;
    inline Node*& left()  { return children[0]; }
    inline Node*& right() { return children[1]; }
};
```

Parcours en profondeur

```
fonction pré-ordre (r, fn)
  si r != ∅
    fn(r)
    pré-ordre(r.gauche, fn)
    pré-ordre(r.droit,  fn)
```

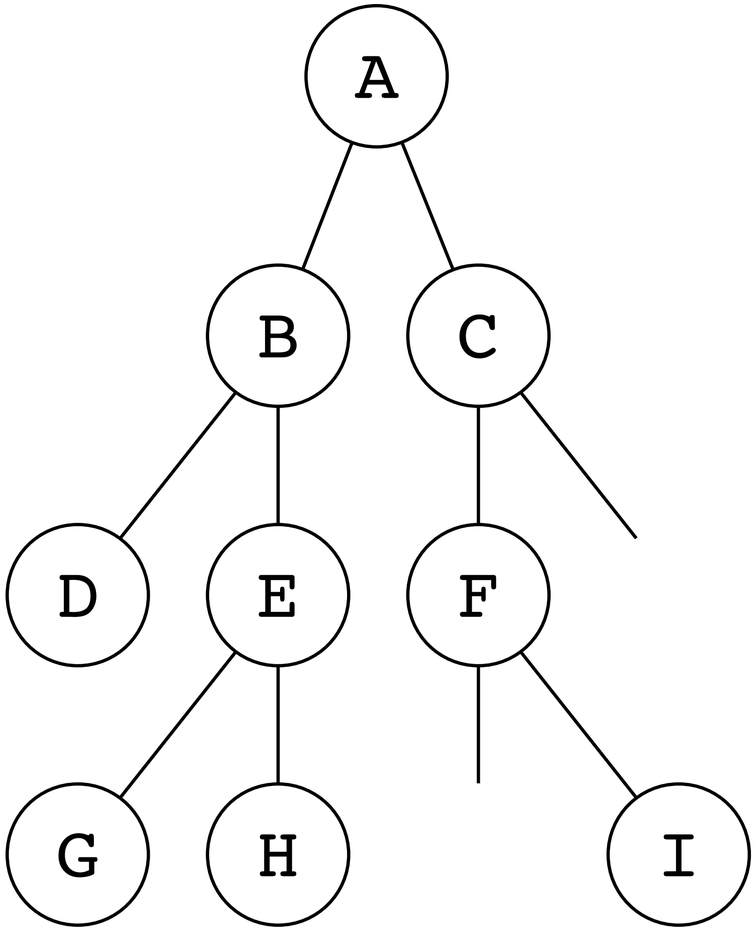
```
fonction post-ordre (r, fn)
  si r != ∅
    post-ordre(r.gauche, fn)
    post-ordre(r.droit,  fn)
    fn(r)
```



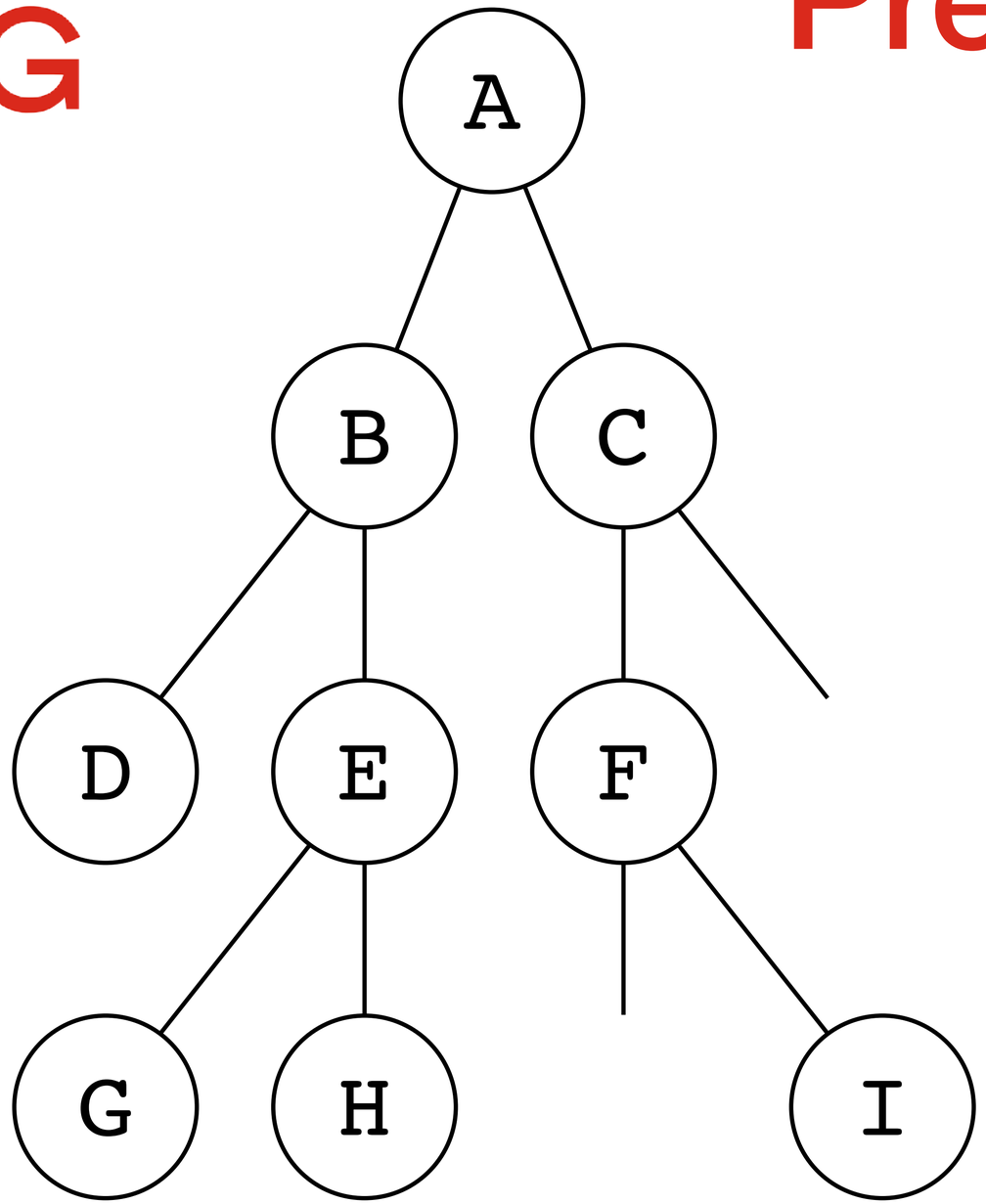
Parcours symétrique

- Traitement de la racine entre les deux appels récursifs

```
fonction symétrique (r, fn)
  si r != ∅
    symétrique(r.gauche, fn)
    fn(r)
    symétrique(r.droit,  fn)
```



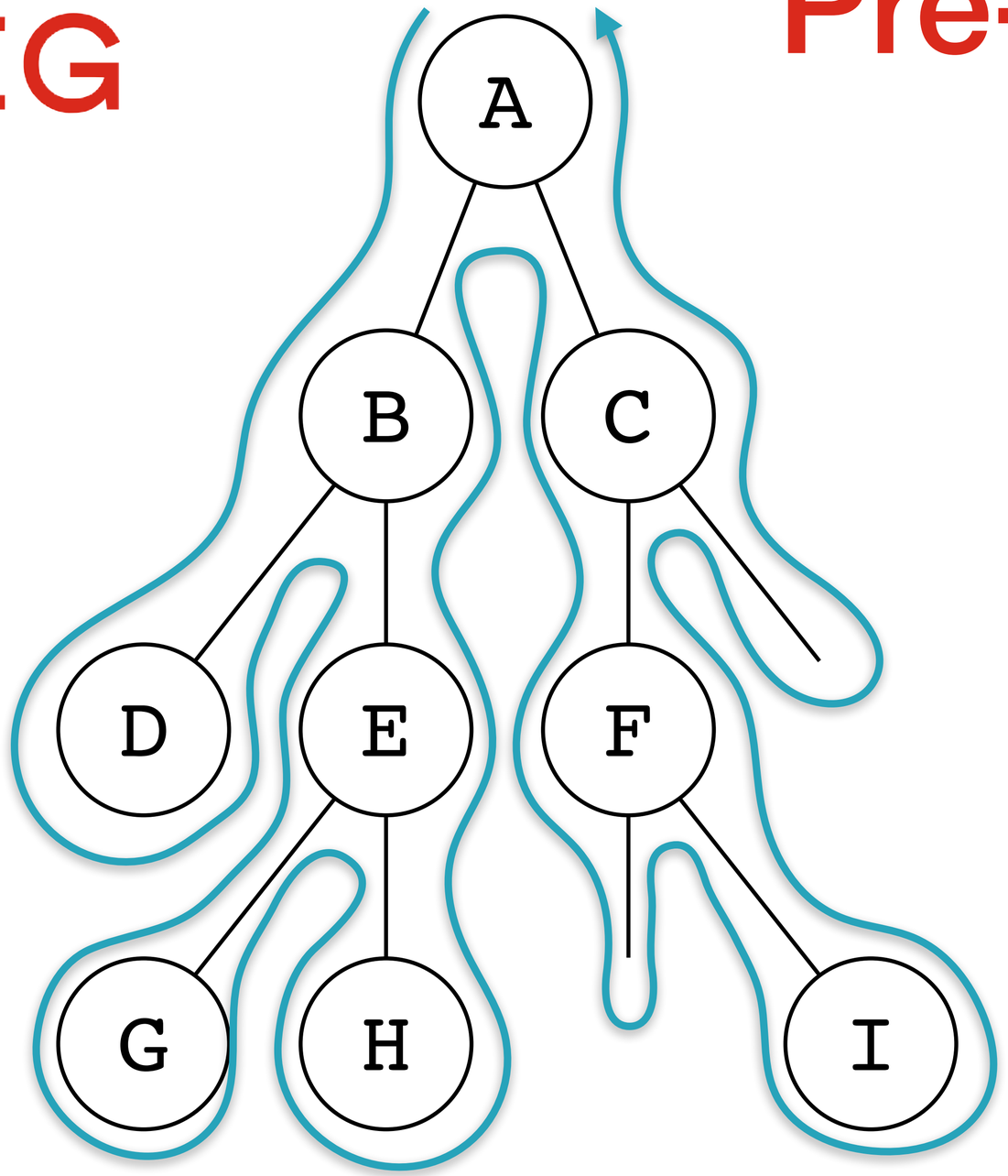
Pré-ordre




```
fonction pré-ordre (r, fn)
  si r != ∅
    fn(r)
    pré-ordre(r.gauche, fn)
    pré-ordre(r.droit,  fn)
```

- Racine
- Puis sous-arbre gauche
- Puis sous-arbre droit

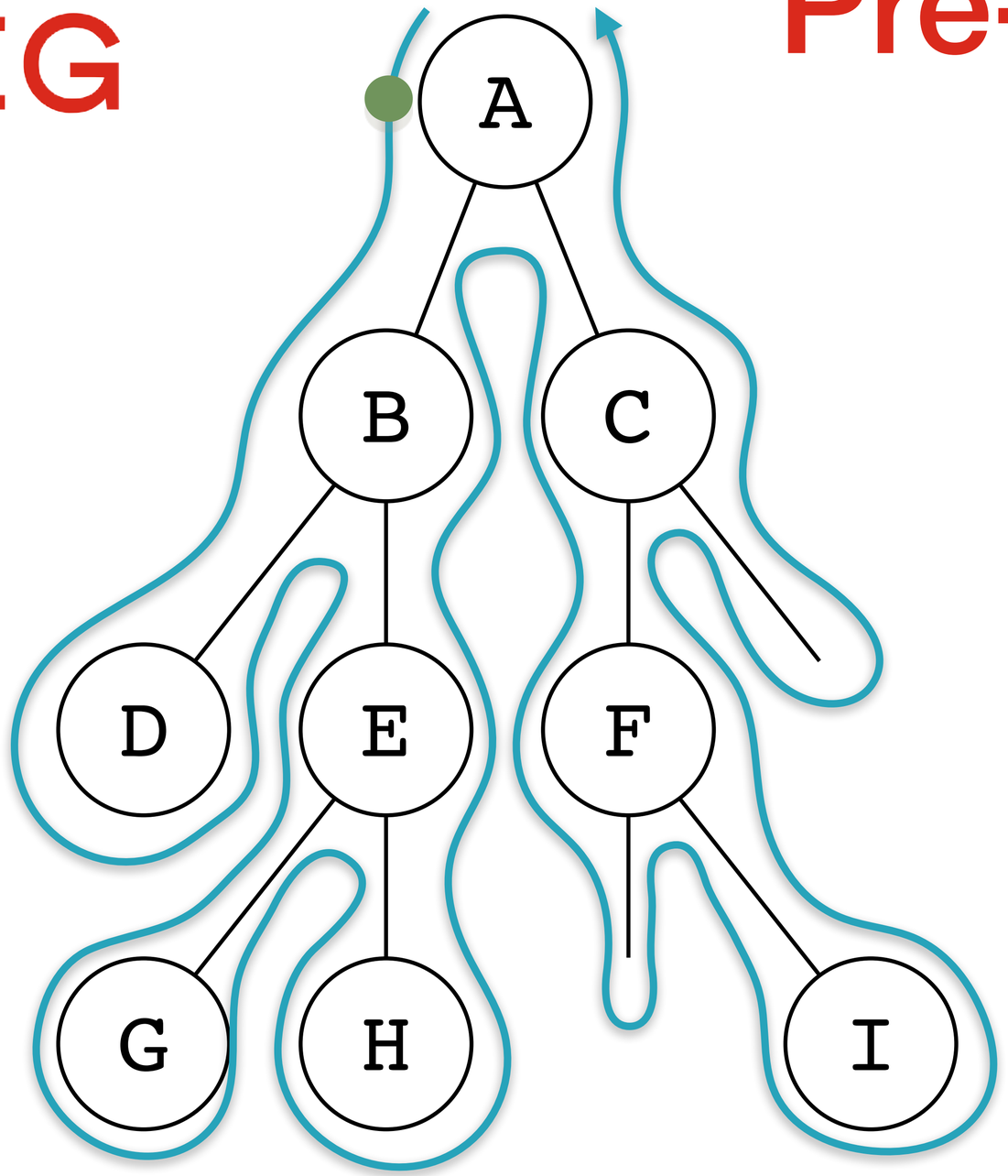
Pré-ordre




```
fonction pré-ordre (r, fn)
  si r != ∅
    fn(r)
    pré-ordre(r.gauche, fn)
    pré-ordre(r.droit,  fn)
```

- Racine
- Puis sous-arbre gauche
- Puis sous-arbre droit

Pré-ordre

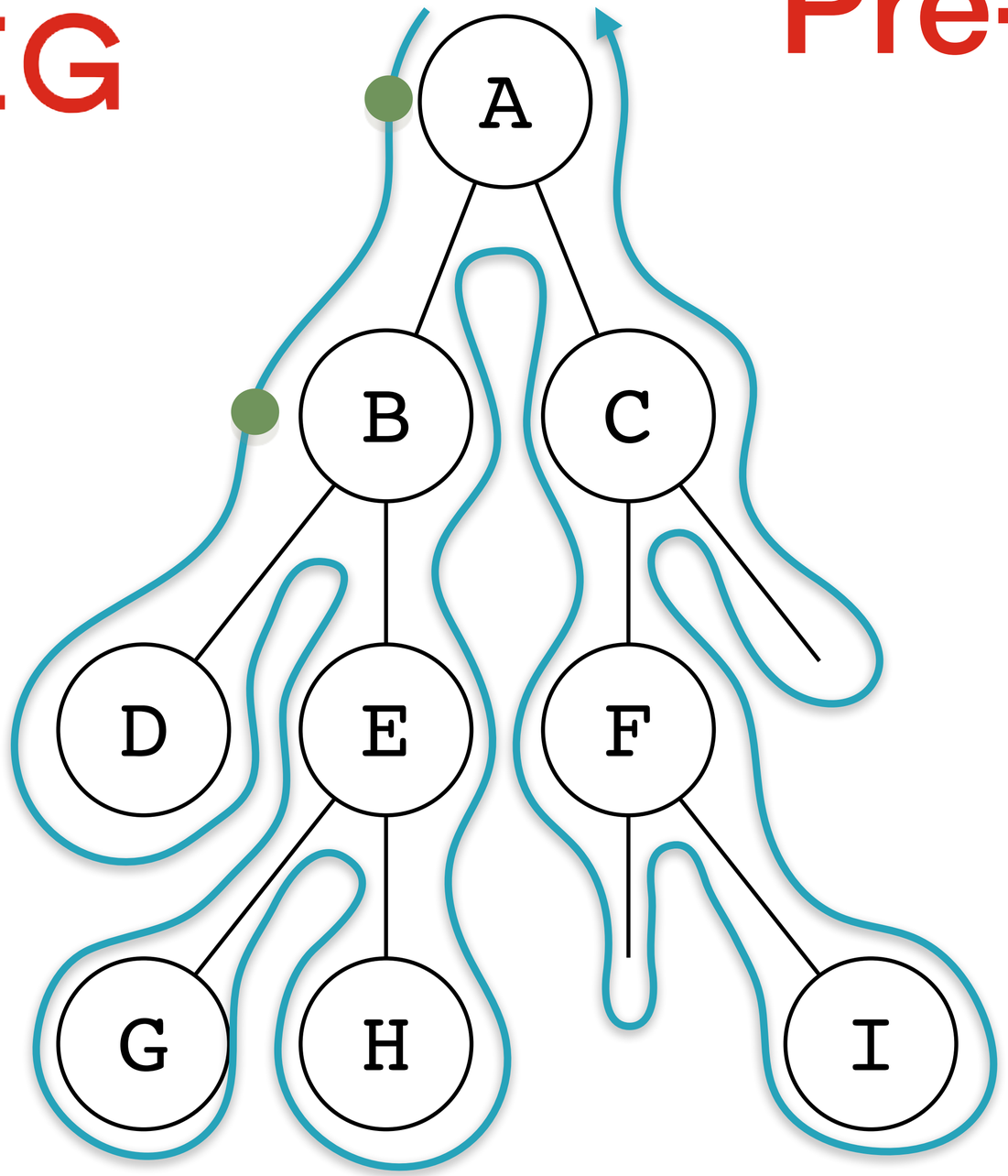



```
fonction pré-ordre (r, fn)
  si r != ∅
    fn(r)
    pré-ordre(r.gauche, fn)
    pré-ordre(r.droit,  fn)
```

- Racine
- Puis sous-arbre gauche
- Puis sous-arbre droit

```
A
```

Pré-ordre

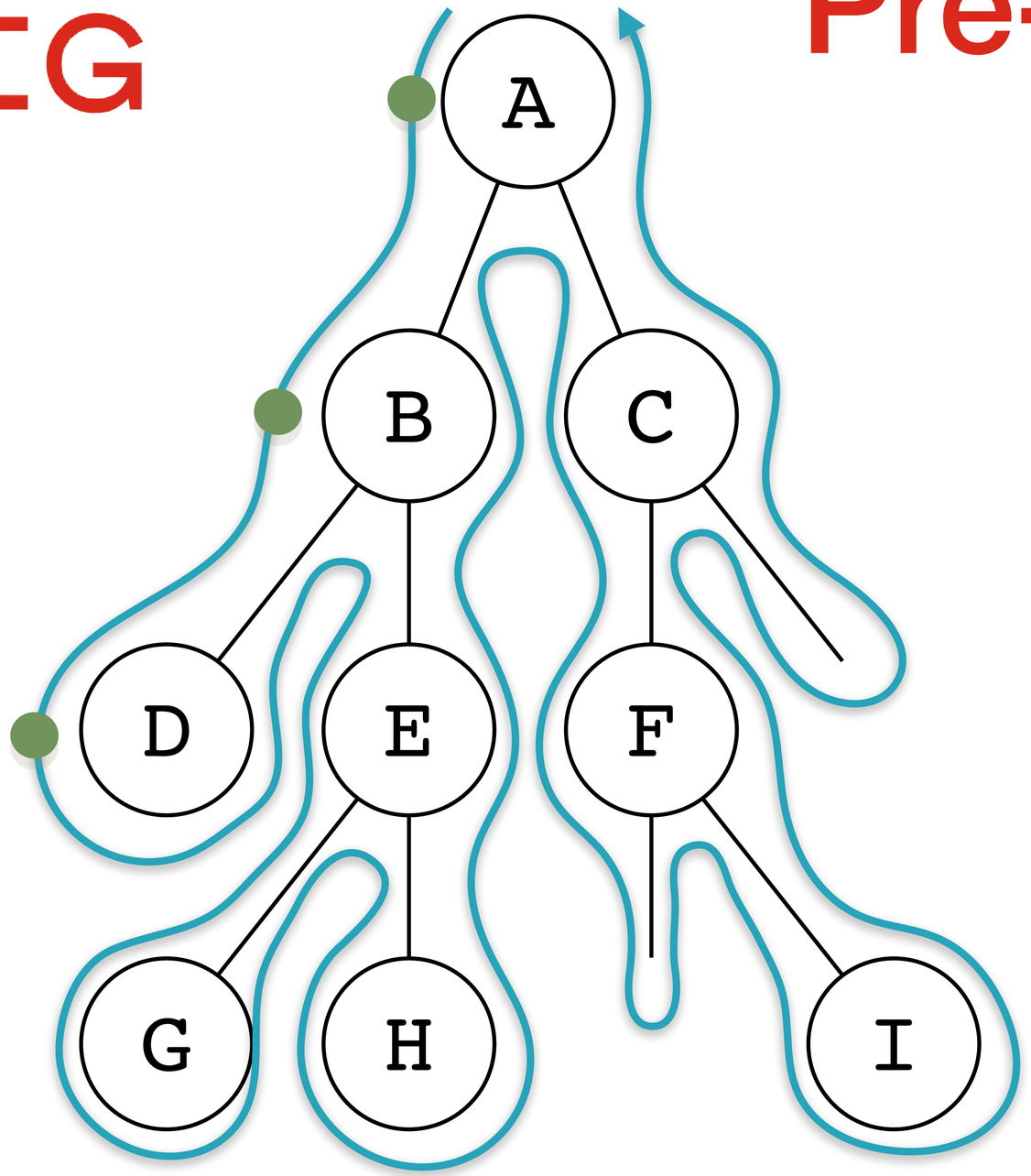



```
fonction pré-ordre (r, fn)
  si r != ∅
    fn(r)
    pré-ordre(r.gauche, fn)
    pré-ordre(r.droit,  fn)
```

- Racine
- Puis sous-arbre gauche
- Puis sous-arbre droit

```
A-B
```

Pré-ordre

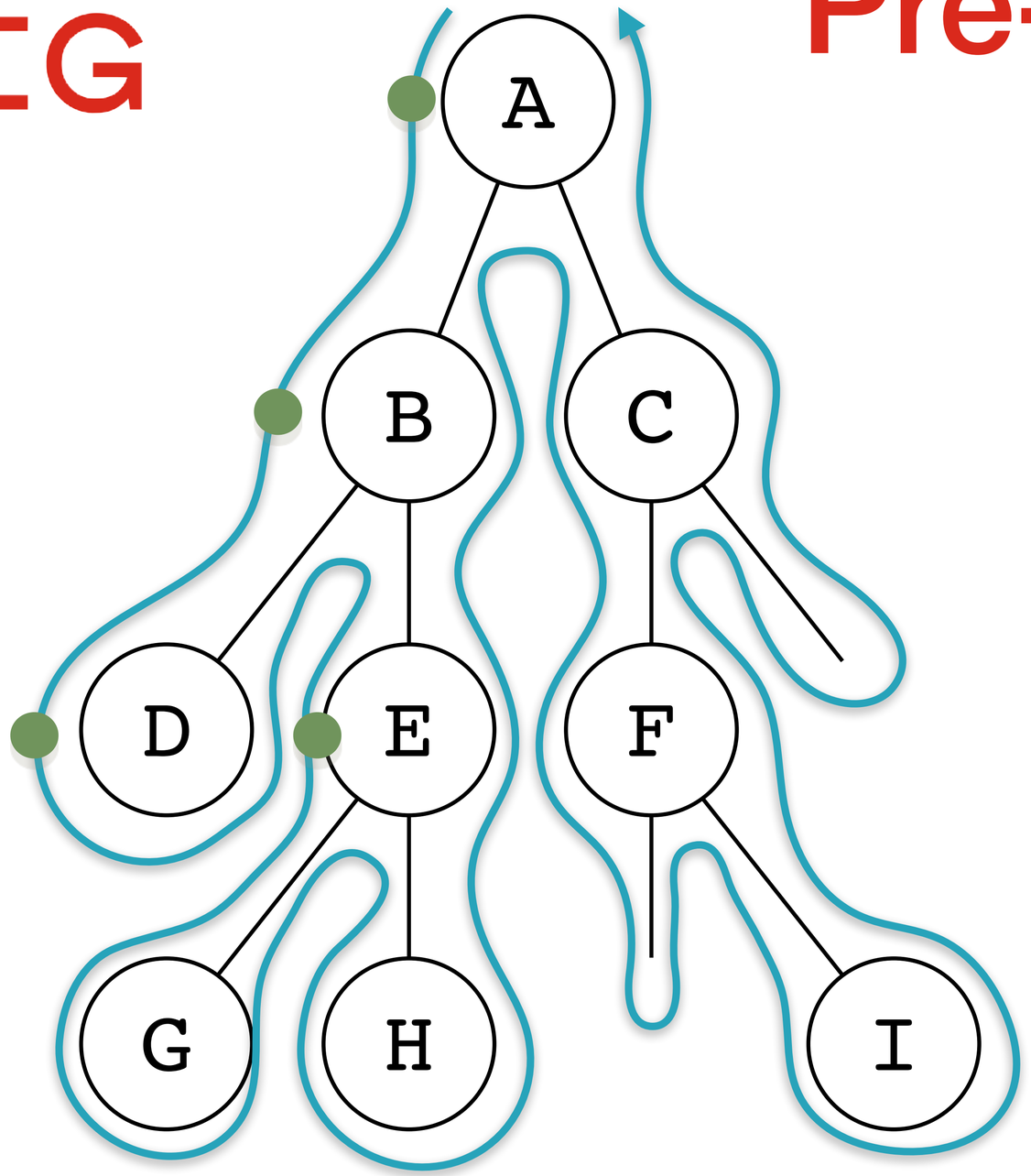



```
fonction pré-ordre (r, fn)
  si r != ∅
    fn(r)
    pré-ordre(r.gauche, fn)
    pré-ordre(r.droit,  fn)
```

- Racine
- Puis sous-arbre gauche
- Puis sous-arbre droit

`A-B-D`

Pré-ordre

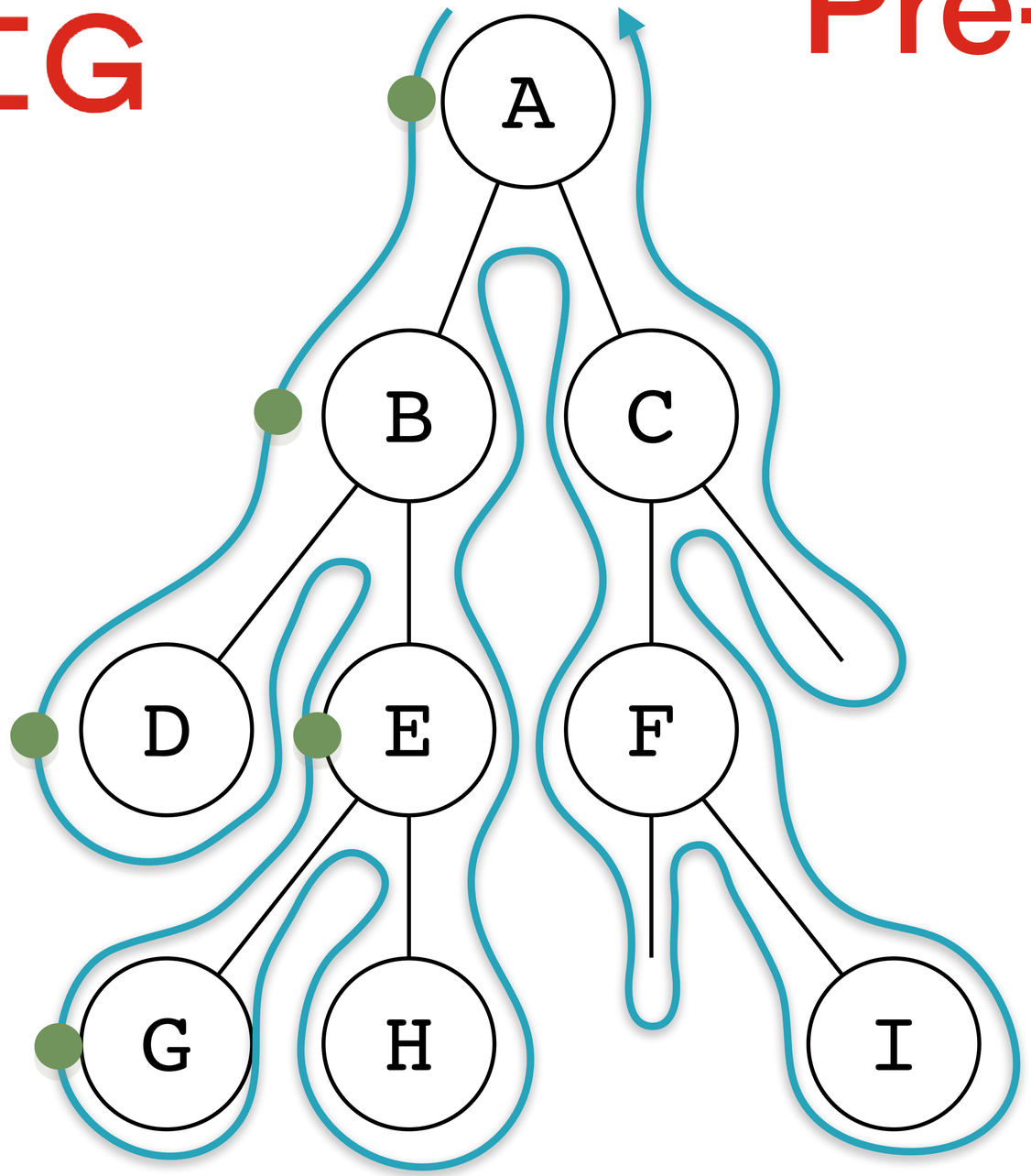



```
fonction pré-ordre (r, fn)
  si r != ∅
    fn(r)
    pré-ordre(r.gauche, fn)
    pré-ordre(r.droit,  fn)
```

- Racine
- Puis sous-arbre gauche
- Puis sous-arbre droit

`A-B-D-E`

Pré-ordre

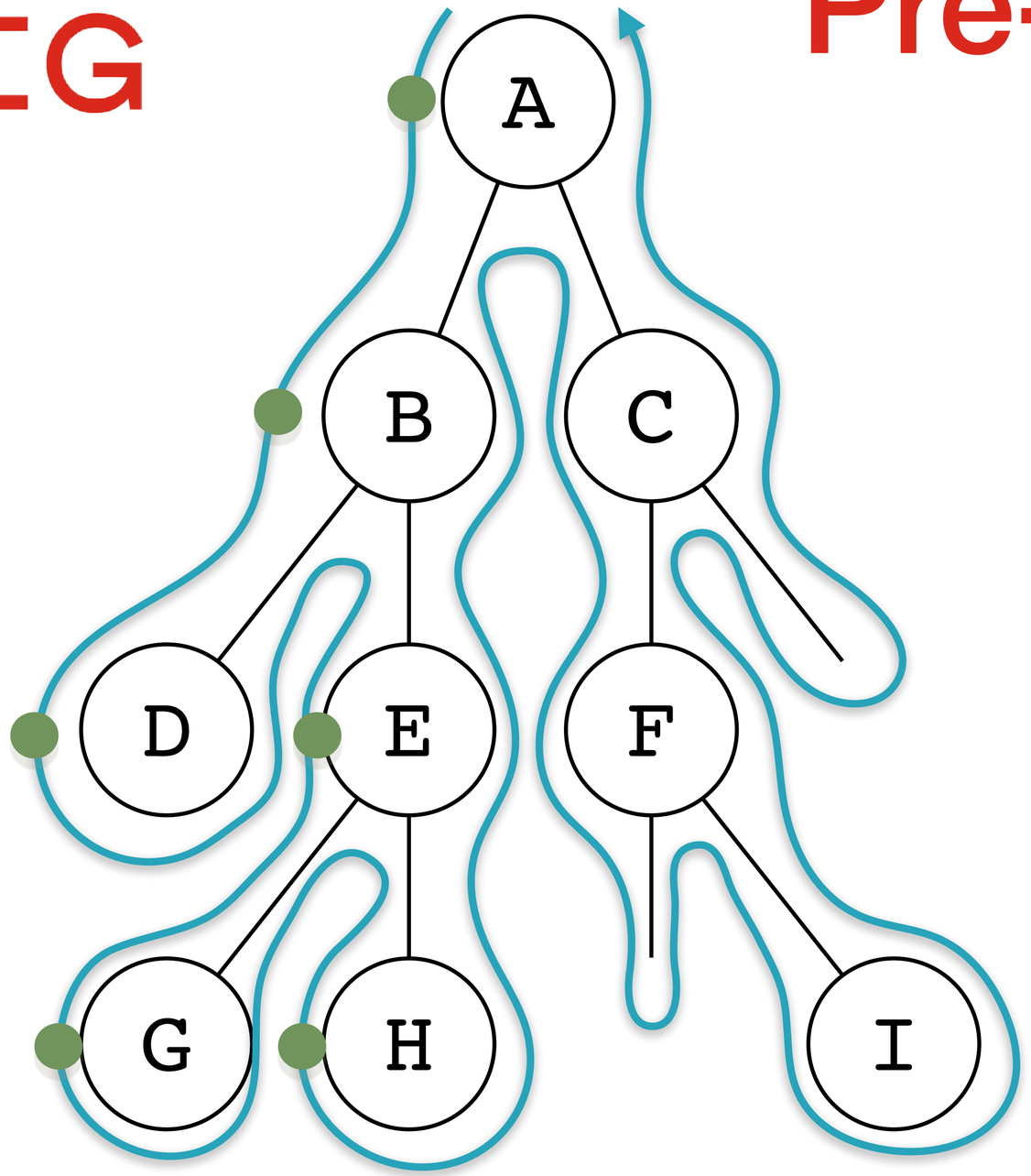



```
fonction pré-ordre (r, fn)
  si r != ∅
    fn(r)
    pré-ordre(r.gauche, fn)
    pré-ordre(r.droit,  fn)
```

- Racine
- Puis sous-arbre gauche
- Puis sous-arbre droit

A-B-D-E-G

Pré-ordre

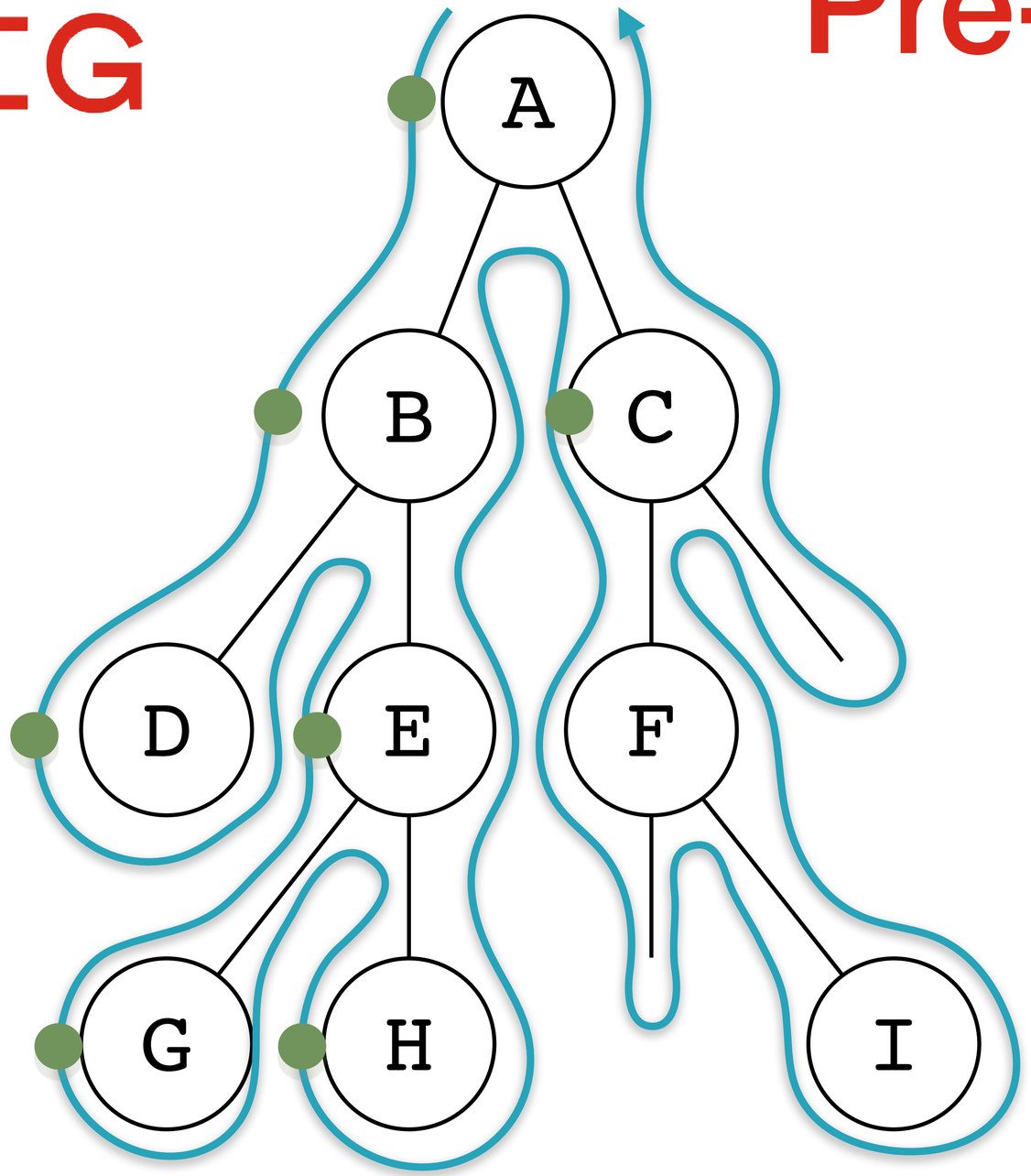



```
fonction pré-ordre (r, fn)
  si r != ∅
    fn(r)
    pré-ordre(r.gauche, fn)
    pré-ordre(r.droit,  fn)
```

- Racine
- Puis sous-arbre gauche
- Puis sous-arbre droit

A-B-D-E-G-H

Pré-ordre

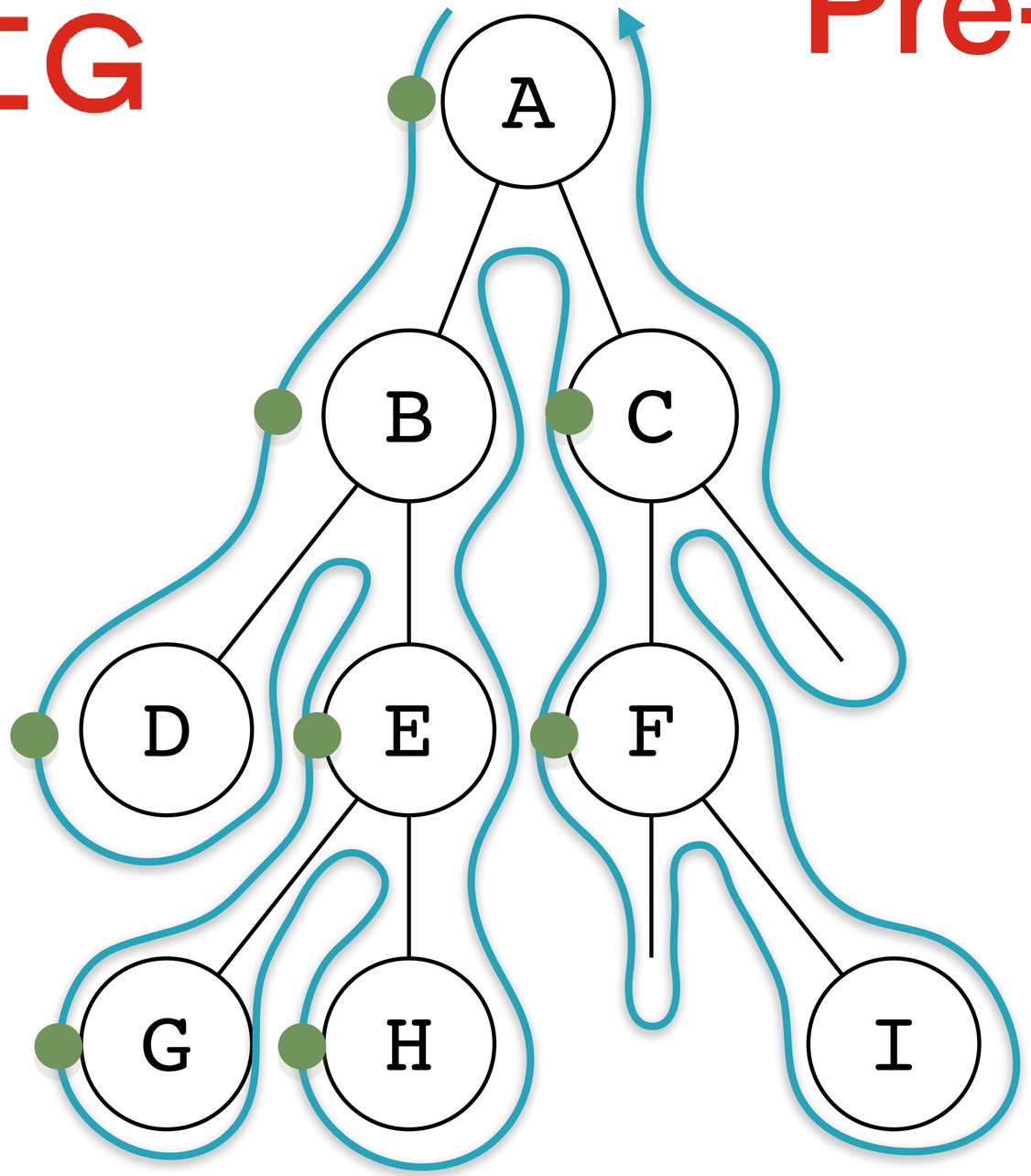



```
fonction pré-ordre (r, fn)
  si r != ∅
    fn(r)
    pré-ordre(r.gauche, fn)
    pré-ordre(r.droit,  fn)
```

- Racine
- Puis sous-arbre gauche
- Puis sous-arbre droit

A-B-D-E-G-H-C

Pré-ordre

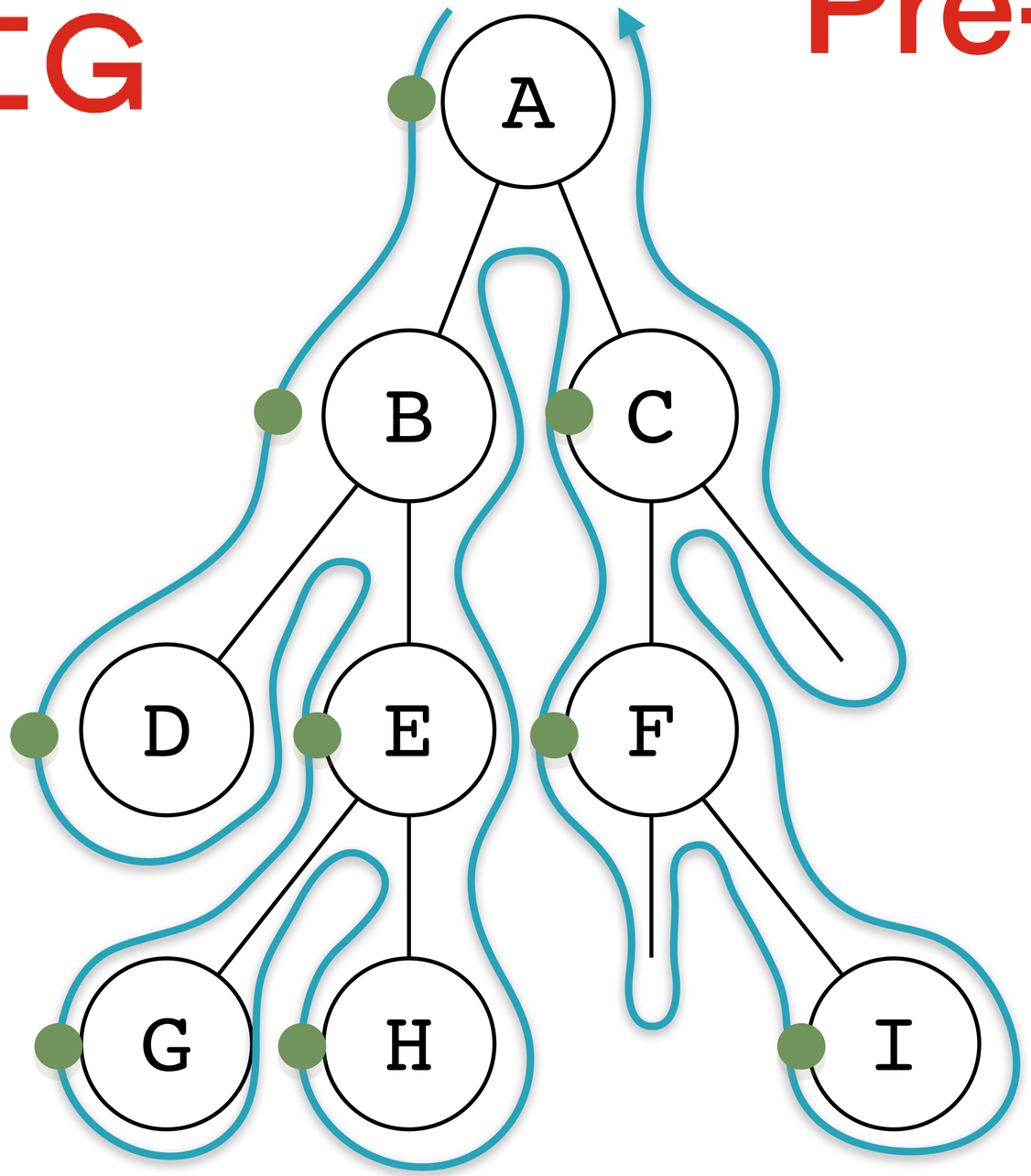



```
fonction pré-ordre (r, fn)
  si r != ∅
    fn(r)
    pré-ordre(r.gauche, fn)
    pré-ordre(r.droit,  fn)
```

- Racine
- Puis sous-arbre gauche
- Puis sous-arbre droit

A-B-D-E-G-H-C-F

Pré-ordre

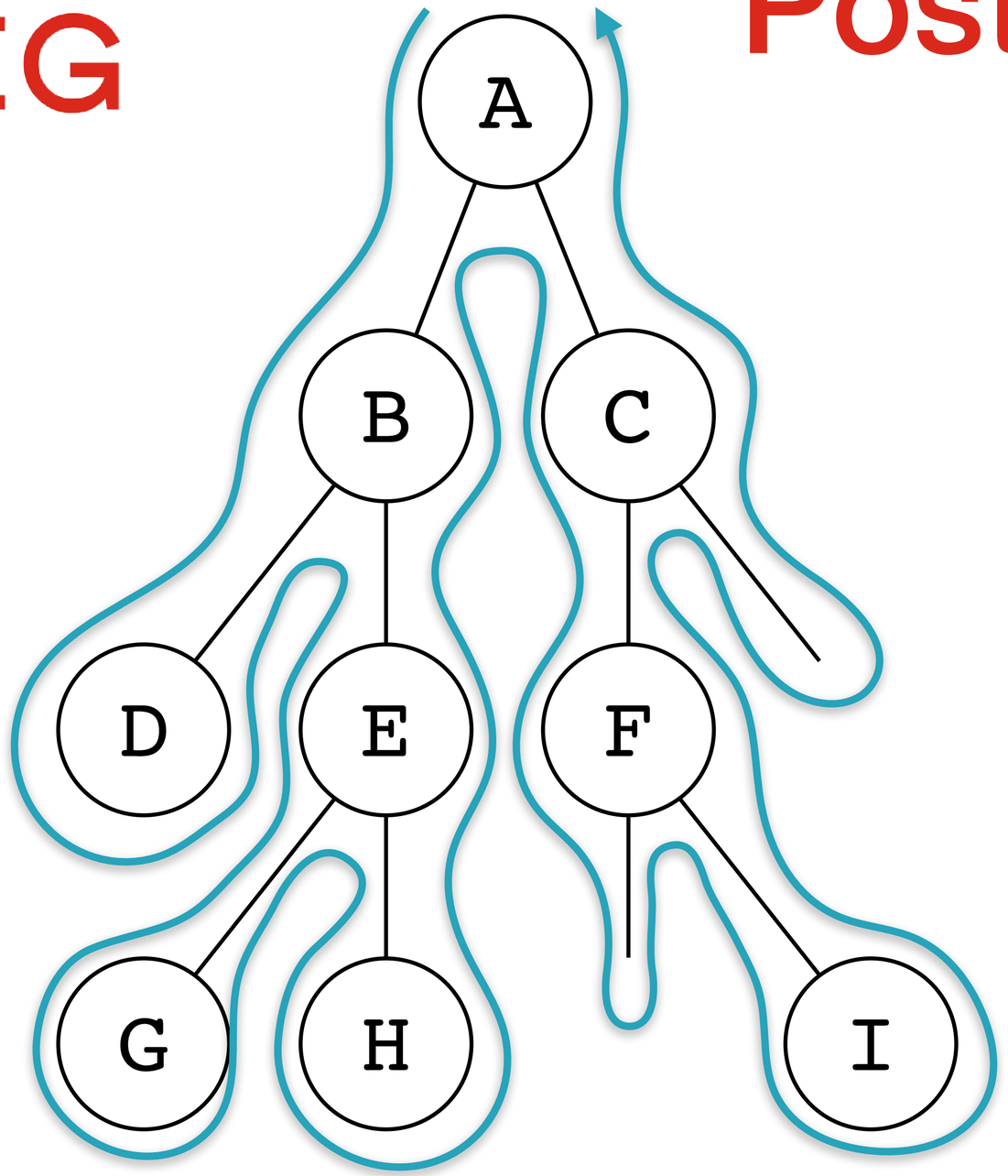



```
fonction pré-ordre (r, fn)
  si r != ∅
    fn(r)
    pré-ordre(r.gauche, fn)
    pré-ordre(r.droit,  fn)
```

- Racine
- Puis sous-arbre gauche
- Puis sous-arbre droit

A-B-D-E-G-H-C-F-I

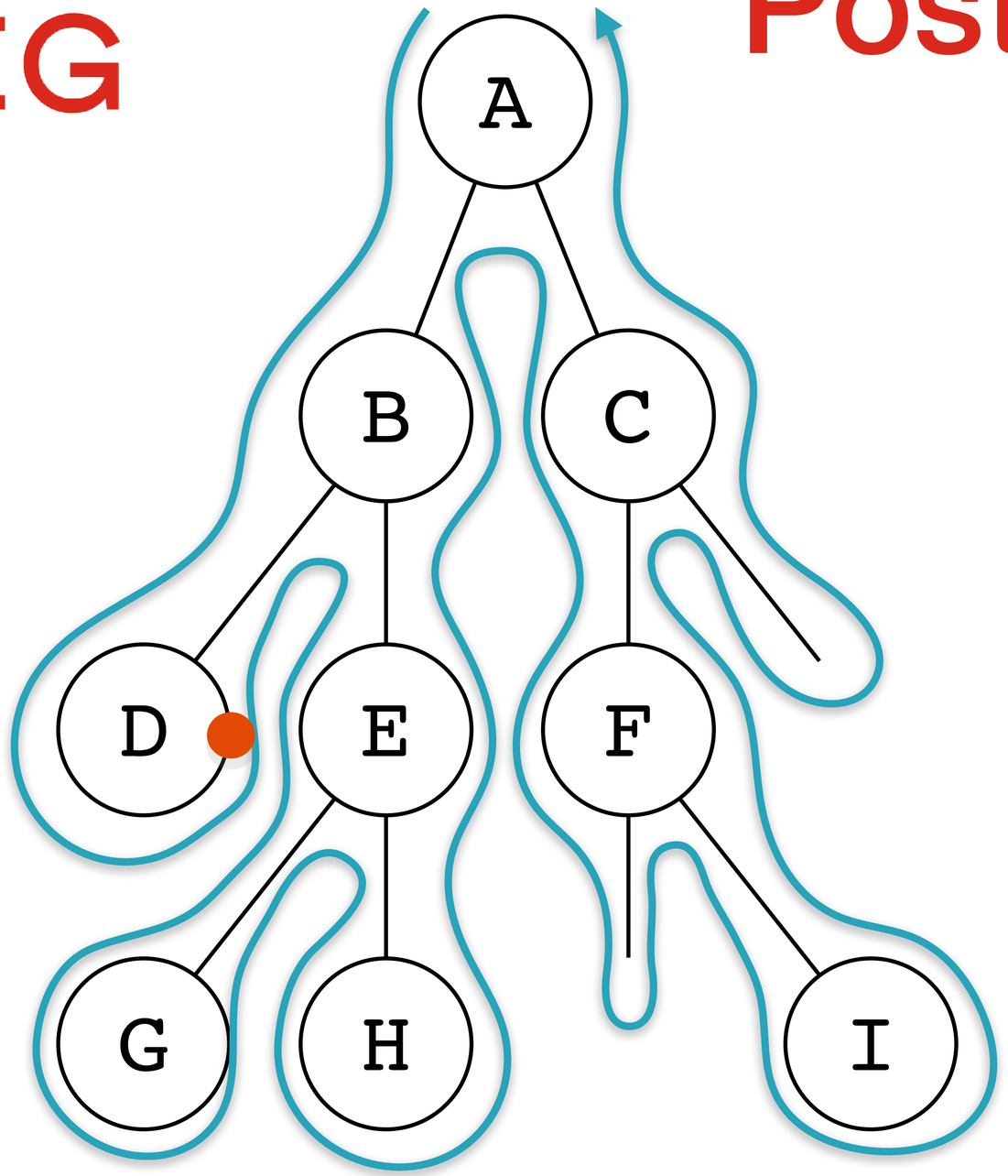
Post-ordre




```
fonction post-ordre (r, fn)
  si r != ∅
    post-ordre(r.gauche, fn)
    post-ordre(r.droit,  fn)
    fn(r)
```

- Sous-arbre gauche
- Puis sous-arbre droit
- Puis racine

Post-ordre

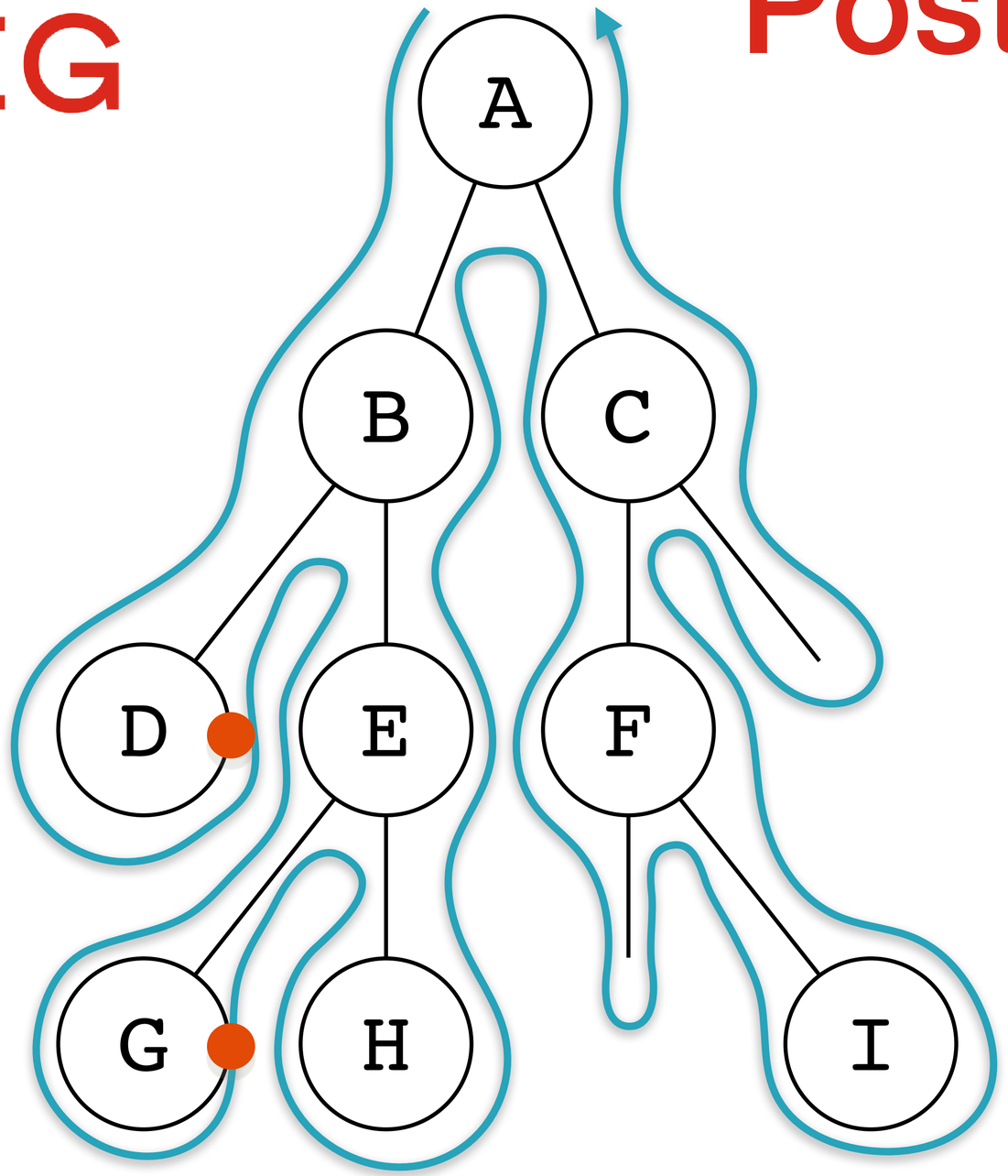



```
fonction post-ordre (r, fn)
  si r != ∅
    post-ordre(r.gauche, fn)
    post-ordre(r.droit,  fn)
    fn(r)
```

- Sous-arbre gauche
- Puis sous-arbre droit
- Puis racine

```
D
```

Post-ordre

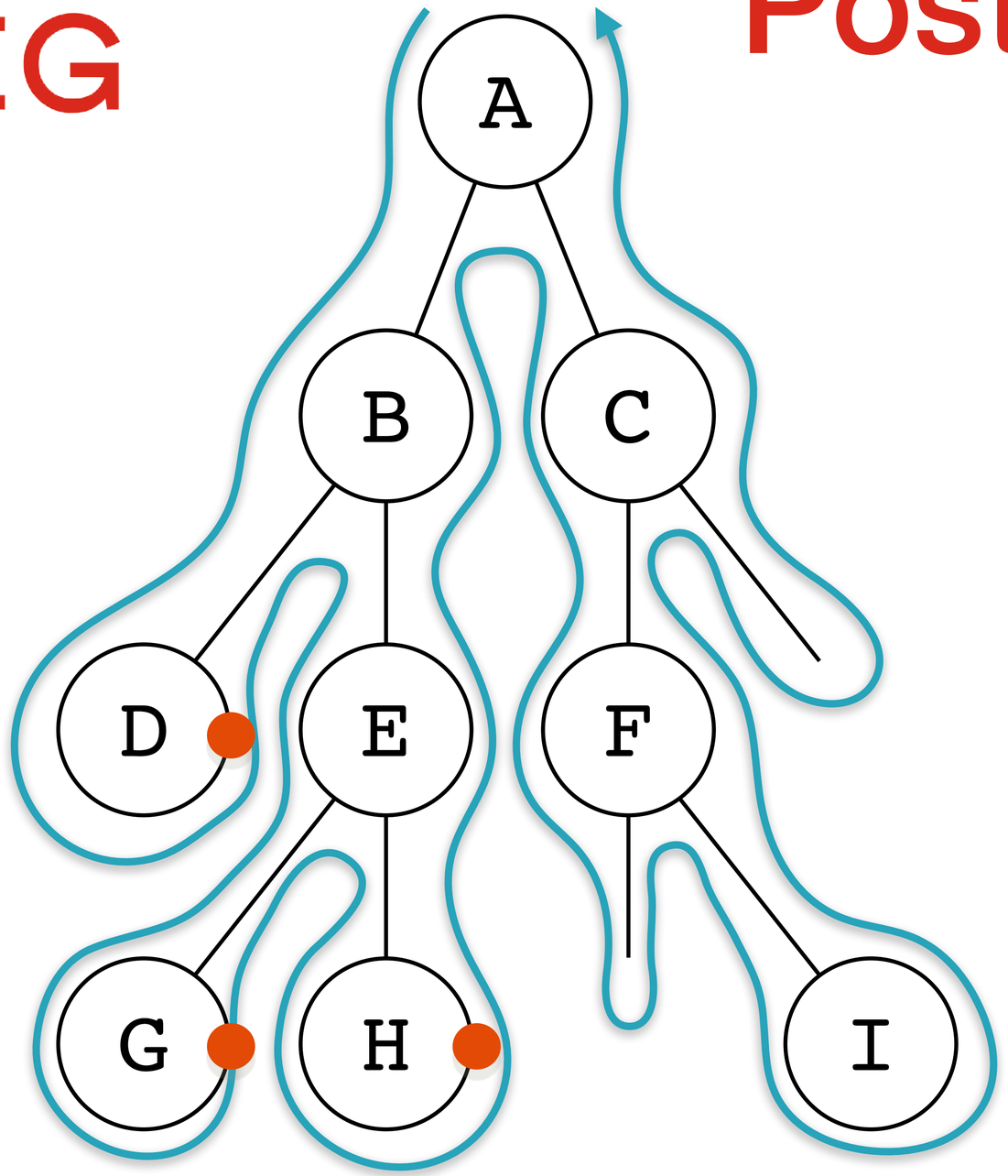



```
fonction post-ordre (r, fn)
  si r != ∅
    post-ordre(r.gauche, fn)
    post-ordre(r.droit,  fn)
    fn(r)
```

- Sous-arbre gauche
- Puis sous-arbre droit
- Puis racine

`D-G`

Post-ordre

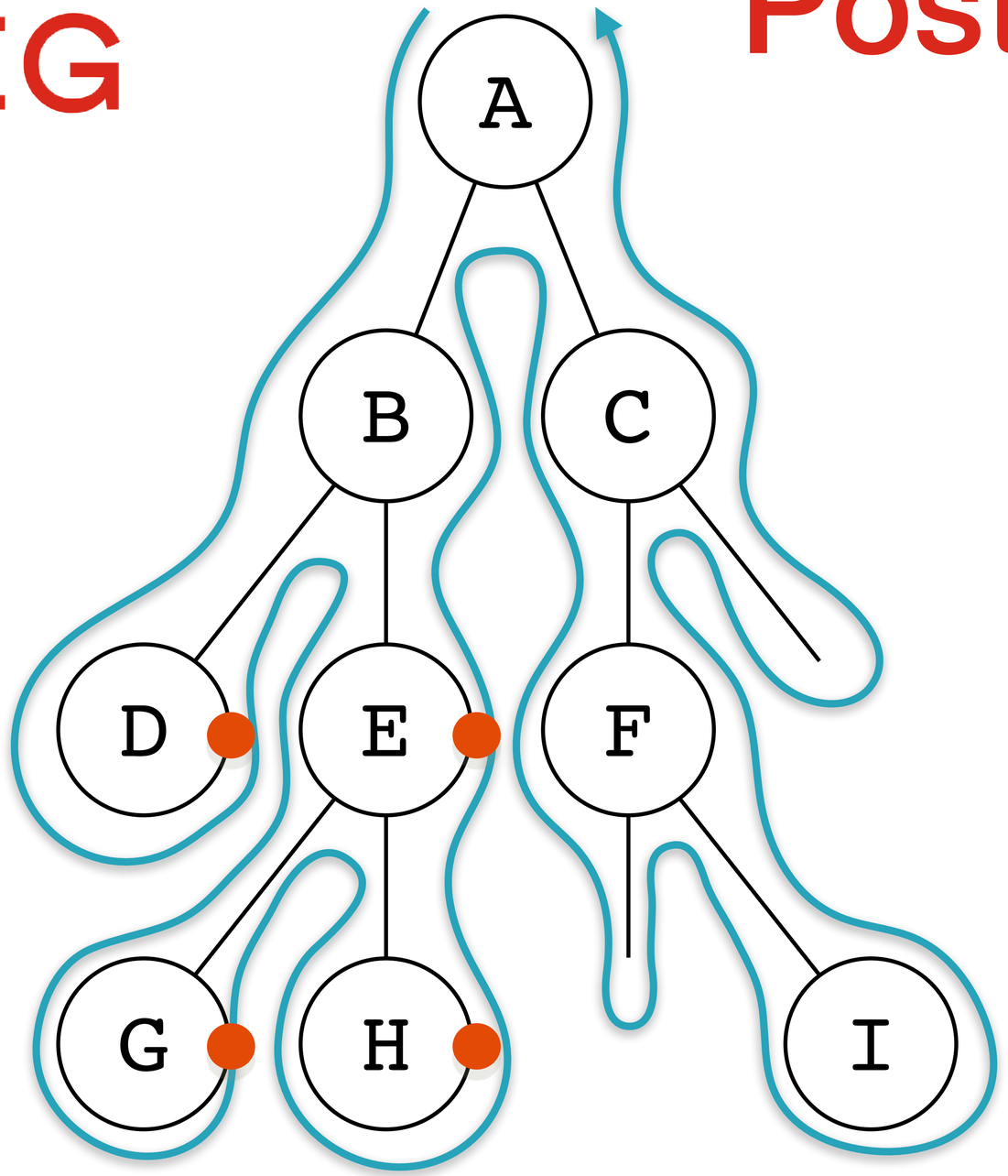



```
fonction post-ordre (r, fn)
  si r != ∅
    post-ordre(r.gauche, fn)
    post-ordre(r.droit,  fn)
    fn(r)
```

- Sous-arbre gauche
- Puis sous-arbre droit
- Puis racine

`D-G-H`

Post-ordre

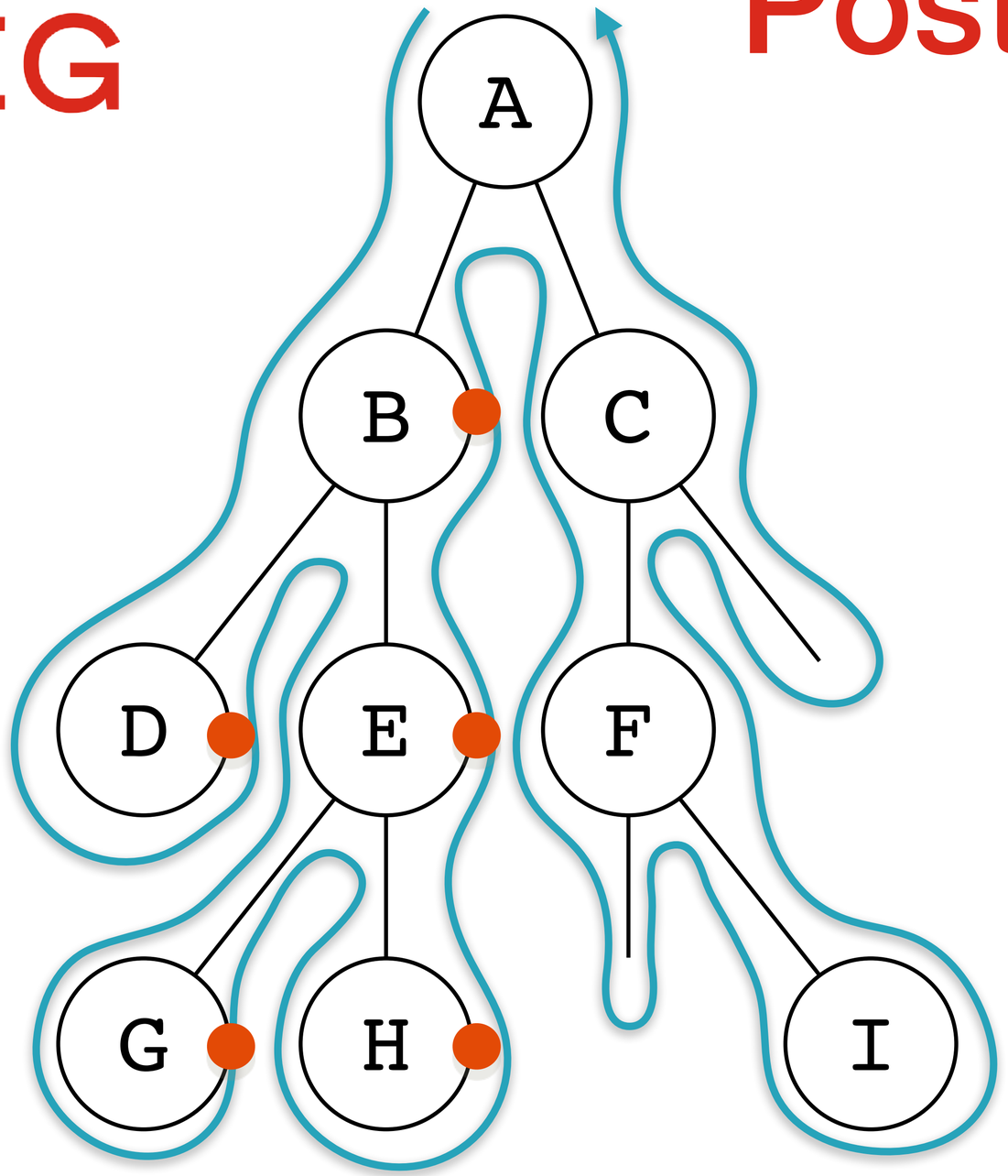



```
fonction post-ordre (r, fn)
  si r != ∅
    post-ordre(r.gauche, fn)
    post-ordre(r.droit,  fn)
    fn(r)
```

- Sous-arbre gauche
- Puis sous-arbre droit
- Puis racine

`D-G-H-E`

Post-ordre

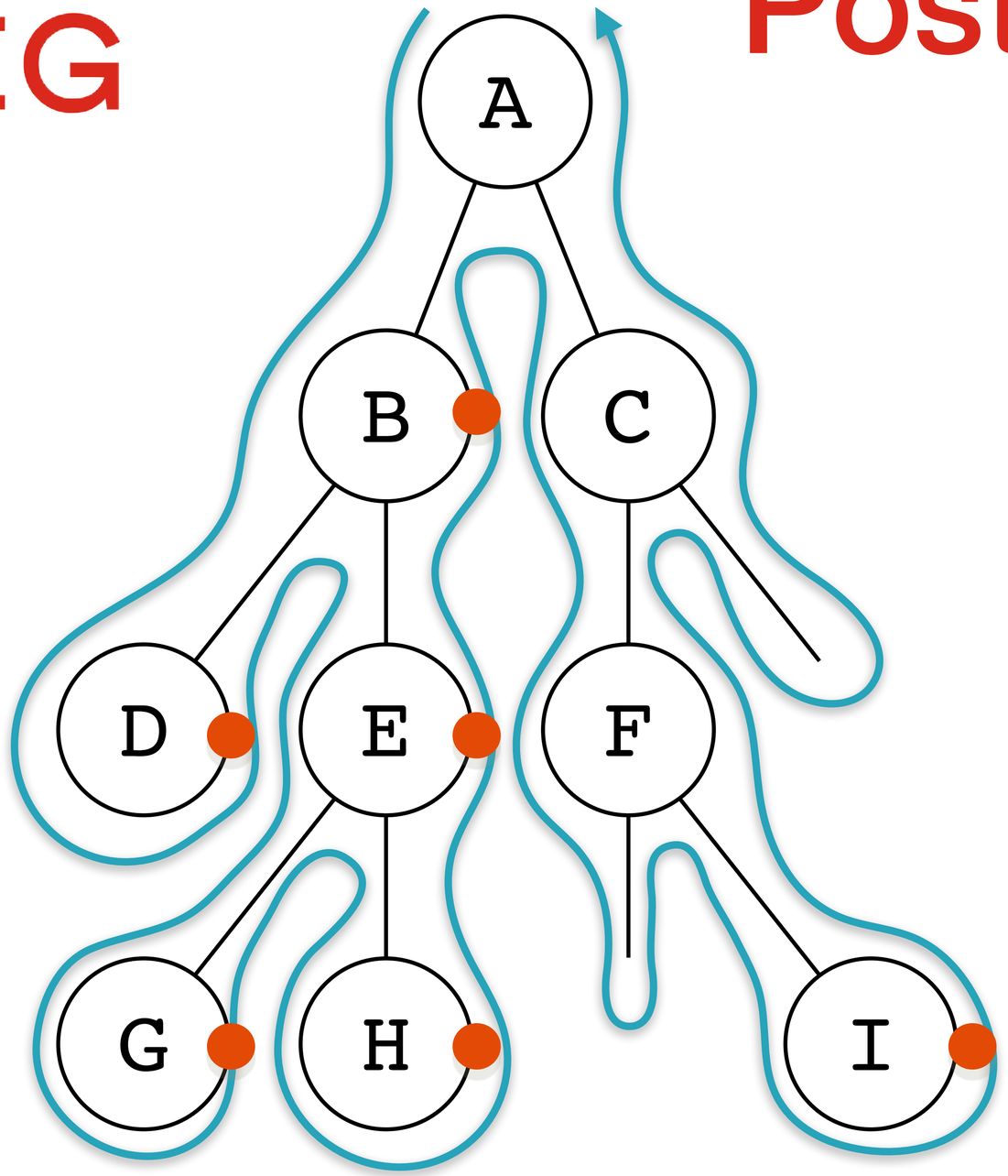



```
fonction post-ordre (r, fn)
  si r != ∅
    post-ordre(r.gauche, fn)
    post-ordre(r.droit,  fn)
    fn(r)
```

- Sous-arbre gauche
- Puis sous-arbre droit
- Puis racine

`D-G-H-E-B`

Post-ordre

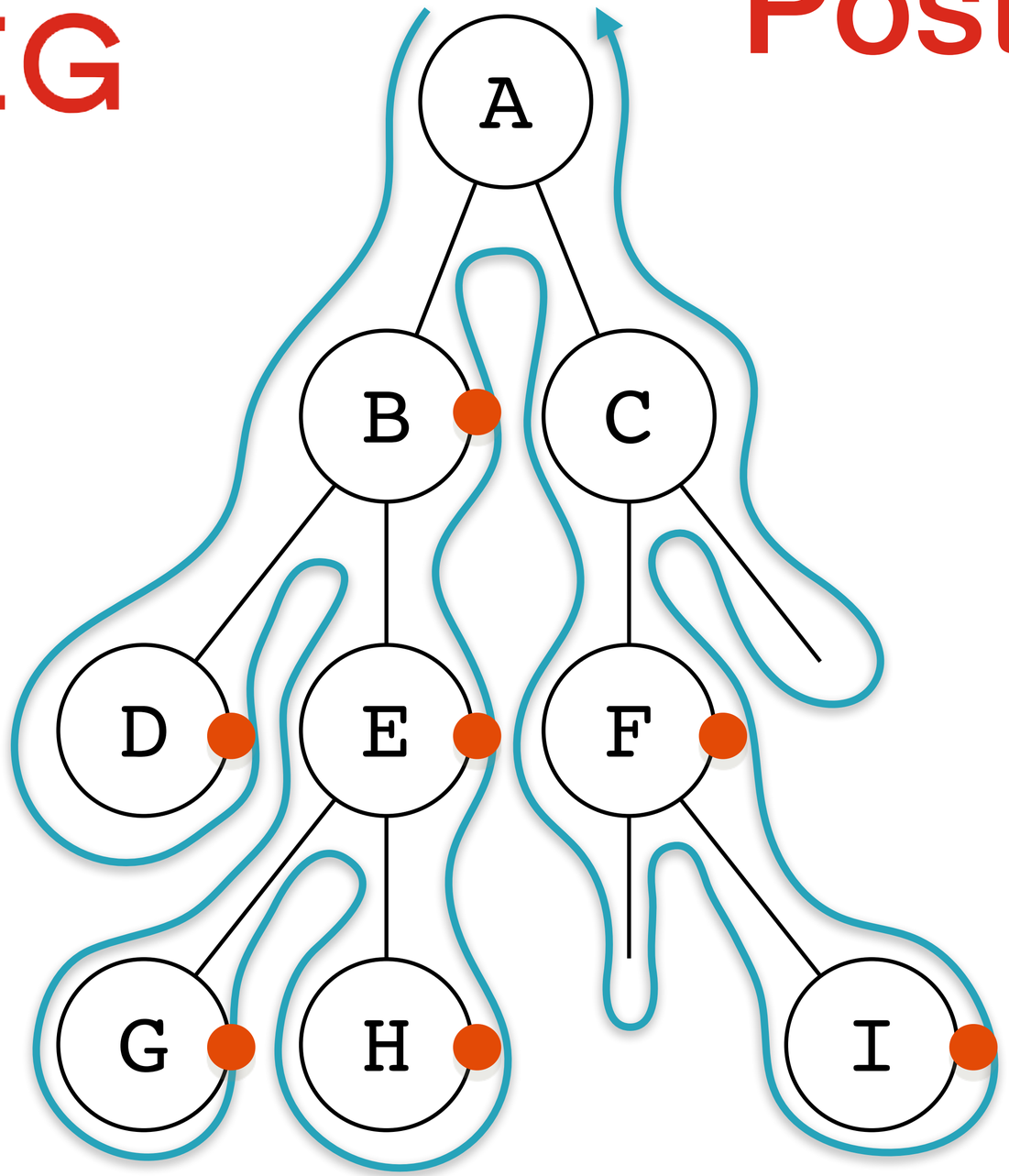



```
fonction post-ordre (r, fn)
  si r != ∅
    post-ordre(r.gauche, fn)
    post-ordre(r.droit,  fn)
    fn(r)
```

- Sous-arbre gauche
- Puis sous-arbre droit
- Puis racine

`D-G-H-E-B-I`

Post-ordre

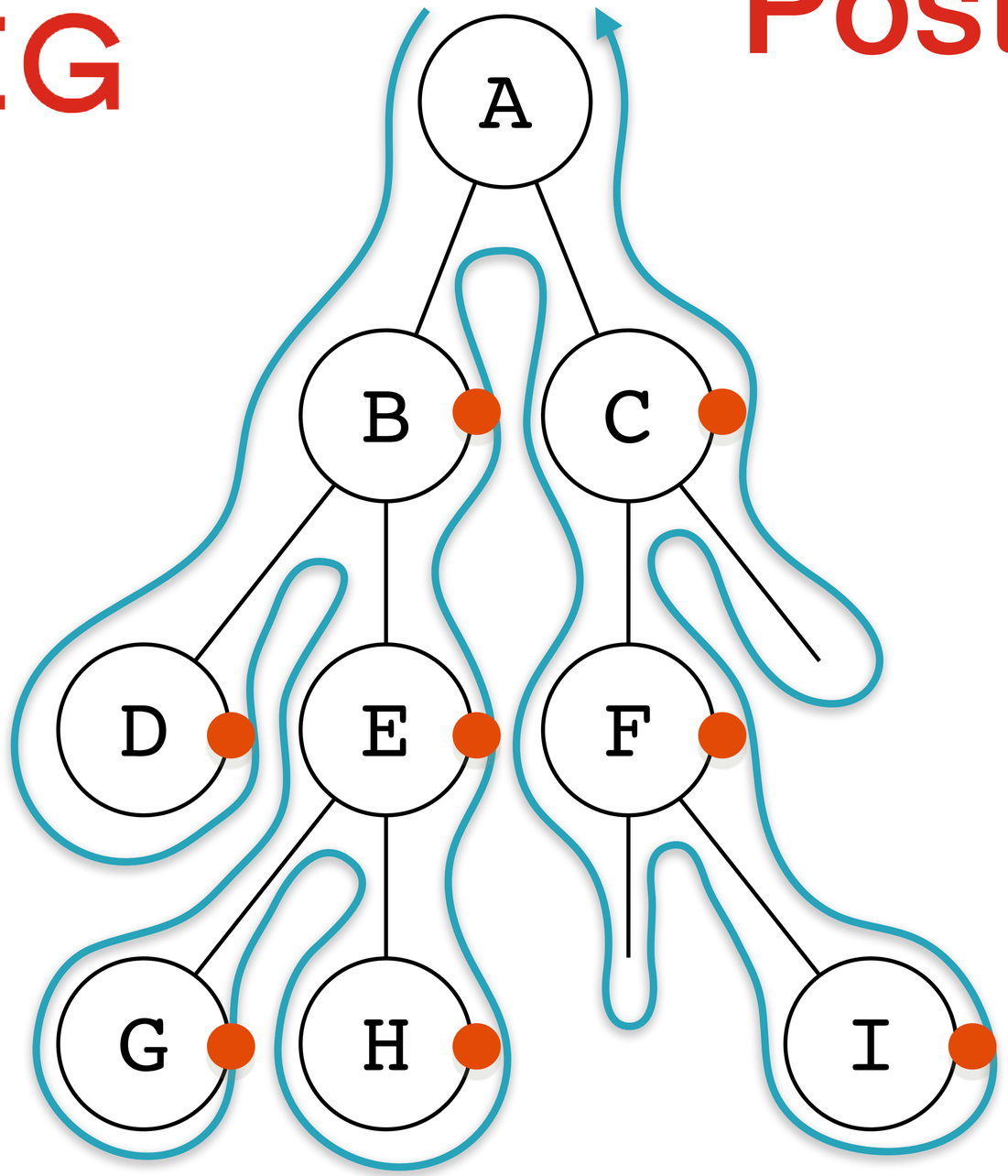



```
fonction post-ordre (r, fn)
  si r != ∅
    post-ordre(r.gauche, fn)
    post-ordre(r.droit,  fn)
    fn(r)
```

- Sous-arbre gauche
- Puis sous-arbre droit
- Puis racine

`D-G-H-E-B-I-F`

Post-ordre

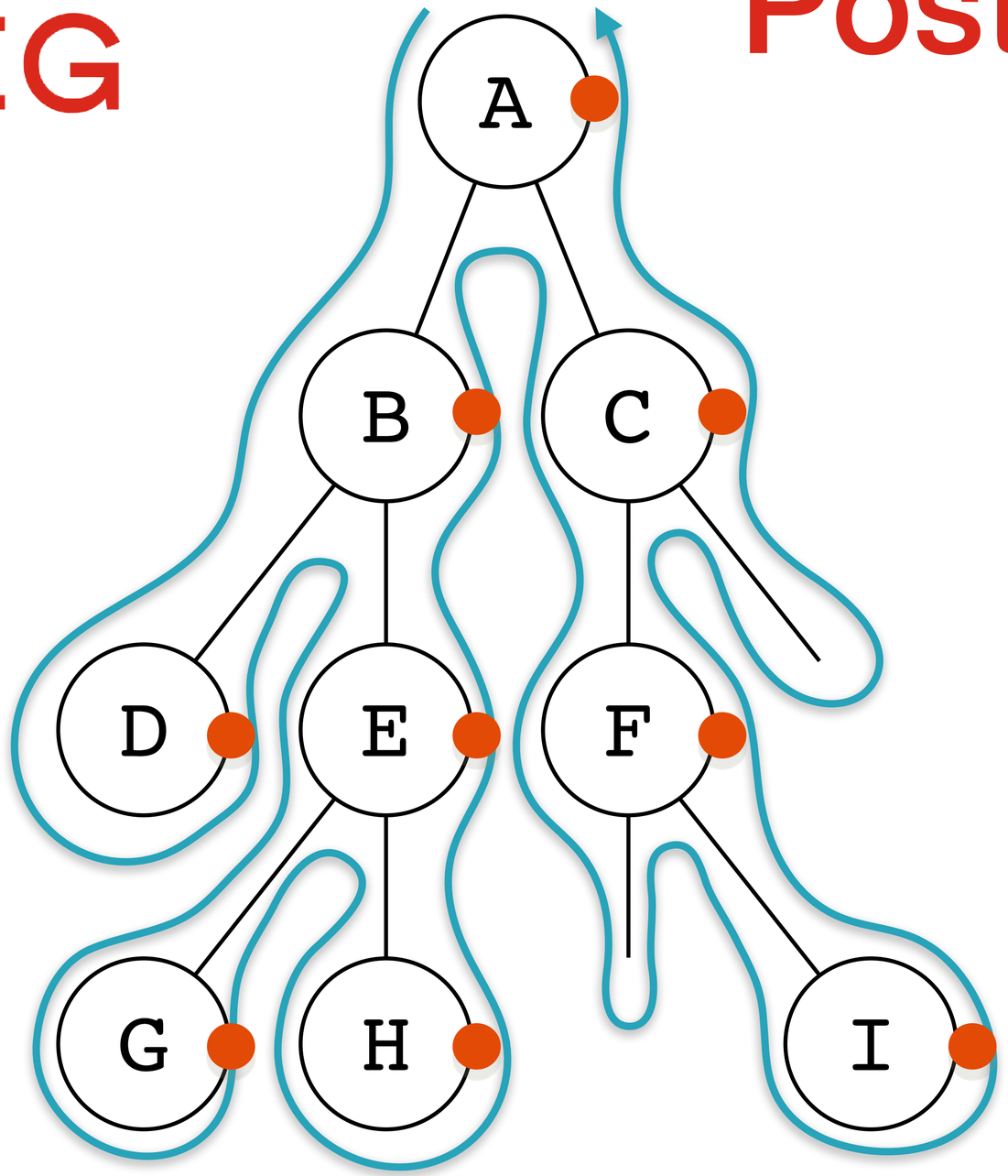



```
fonction post-ordre (r, fn)
  si r != ∅
    post-ordre(r.gauche, fn)
    post-ordre(r.droit,  fn)
    fn(r)
```

- Sous-arbre gauche
- Puis sous-arbre droit
- Puis racine

`D-G-H-E-B-I-F-C`

Post-ordre

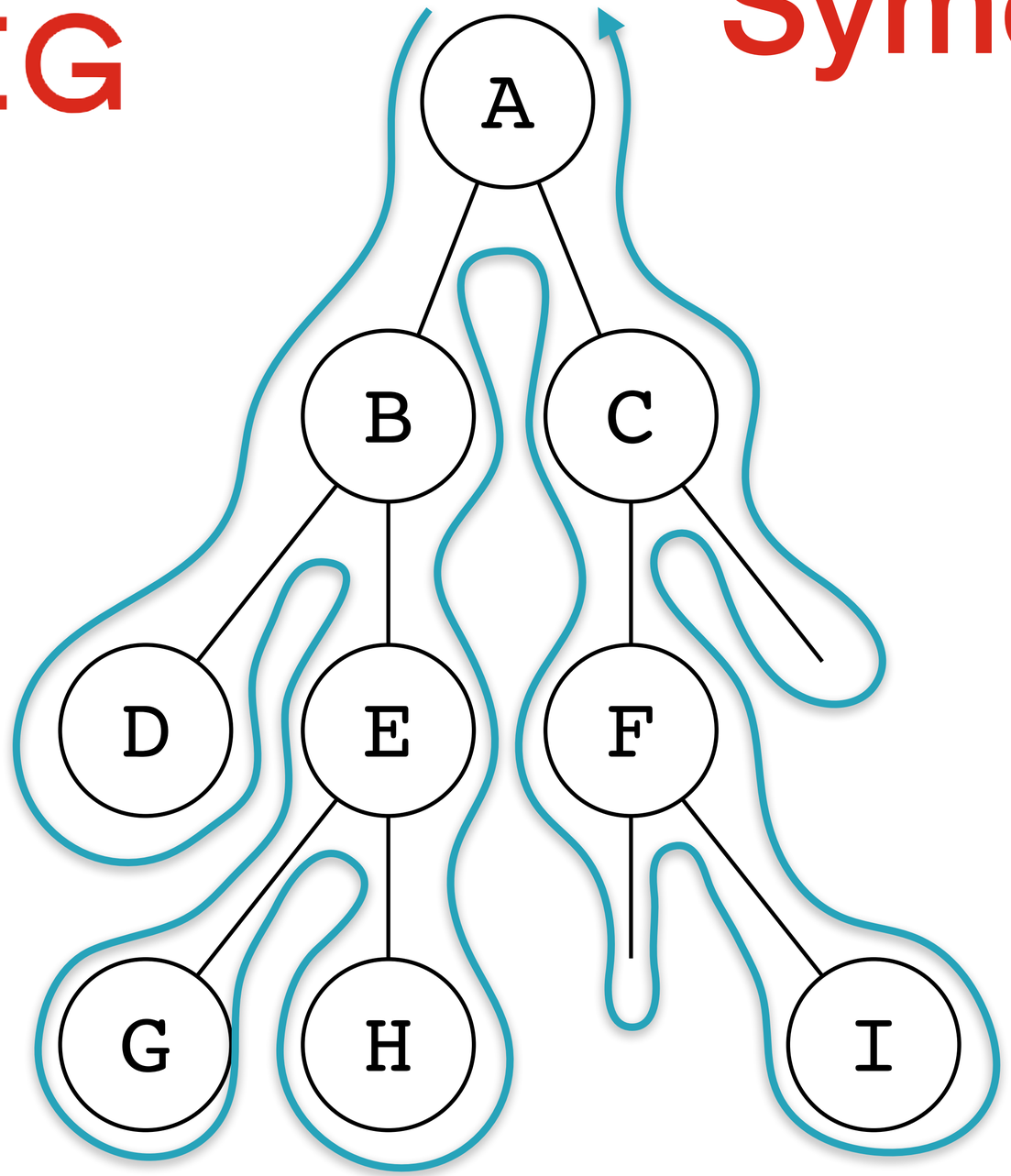



```
fonction post-ordre (r, fn)
  si r != ∅
    post-ordre(r.gauche, fn)
    post-ordre(r.droit,  fn)
    fn(r)
```

- Sous-arbre gauche
- Puis sous-arbre droit
- Puis racine

D-G-H-E-B-I-F-C-A

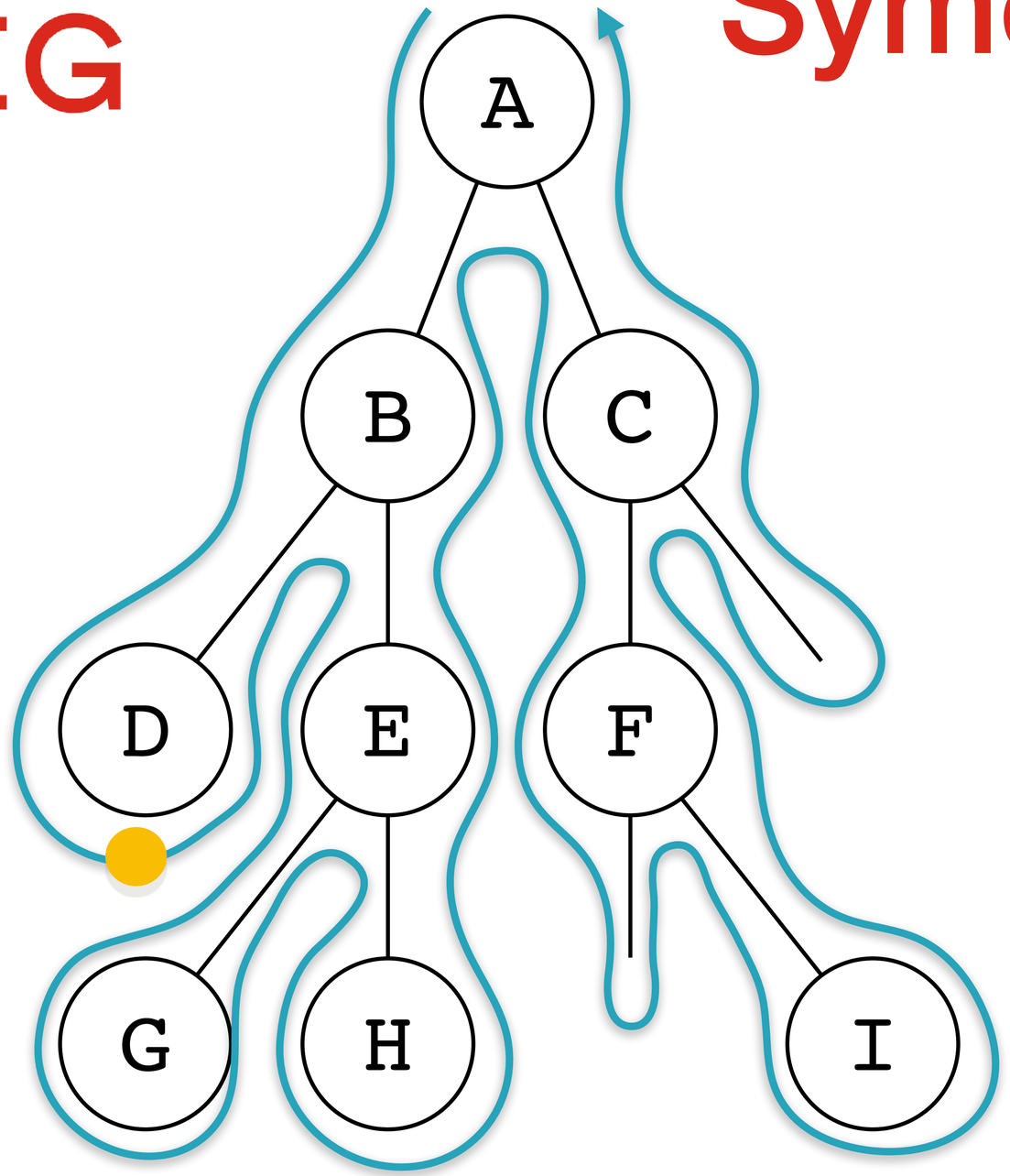
Symétrique




```
fonction symétrique (r, fn)
  si r != ∅
     symétrique(r.gauche, fn)
     fn(r)
     symétrique(r.droit,  fn)
```

- Sous-arbre gauche
- Puis racine
- Puis sous-arbre droit

Symétrique

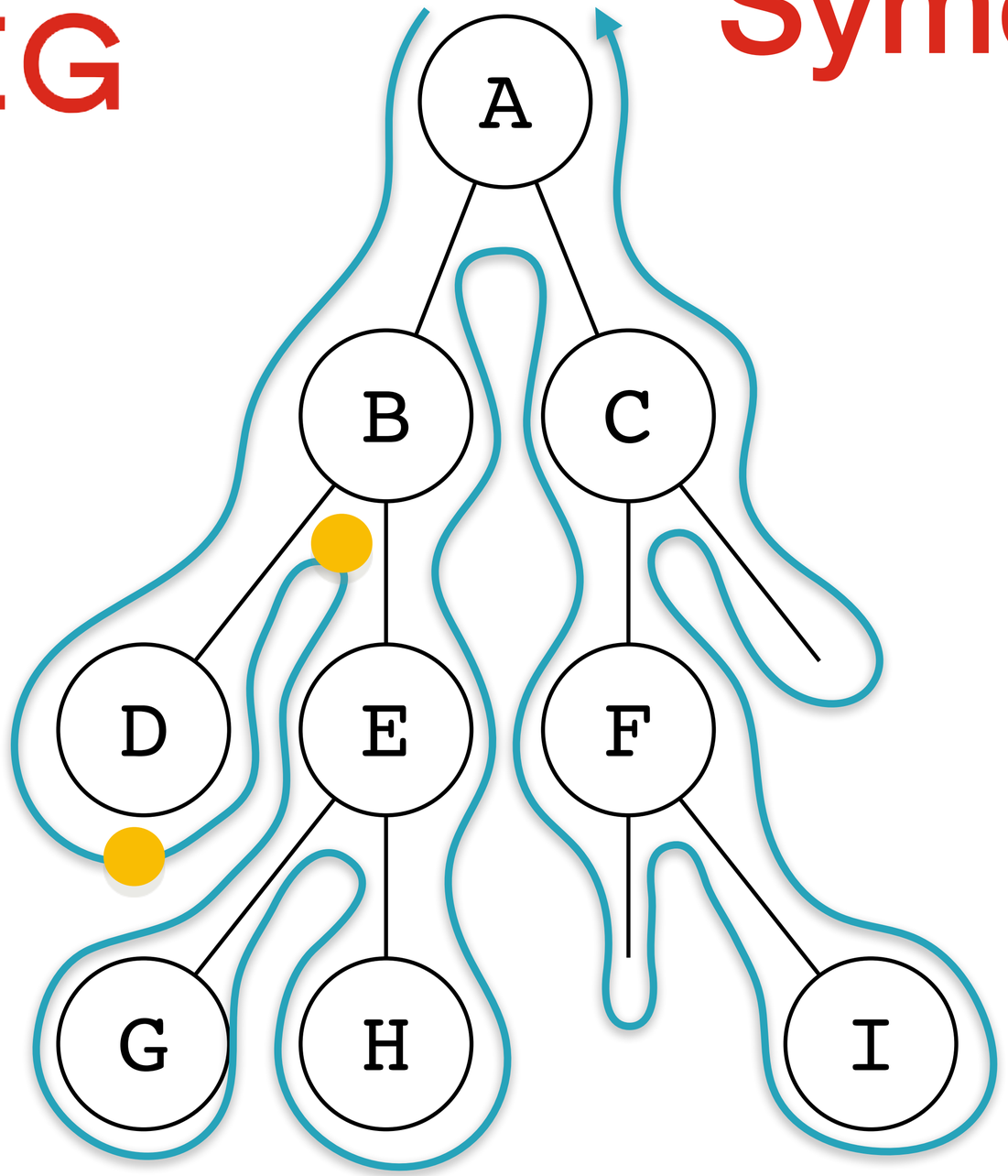



```
fonction symétrique (r, fn)
  si r != ∅
     symétrique(r.gauche, fn)
     fn(r)
     symétrique(r.droit,  fn)
```

- Sous-arbre gauche
- Puis racine
- Puis sous-arbre droit

```
D
```

Symétrique

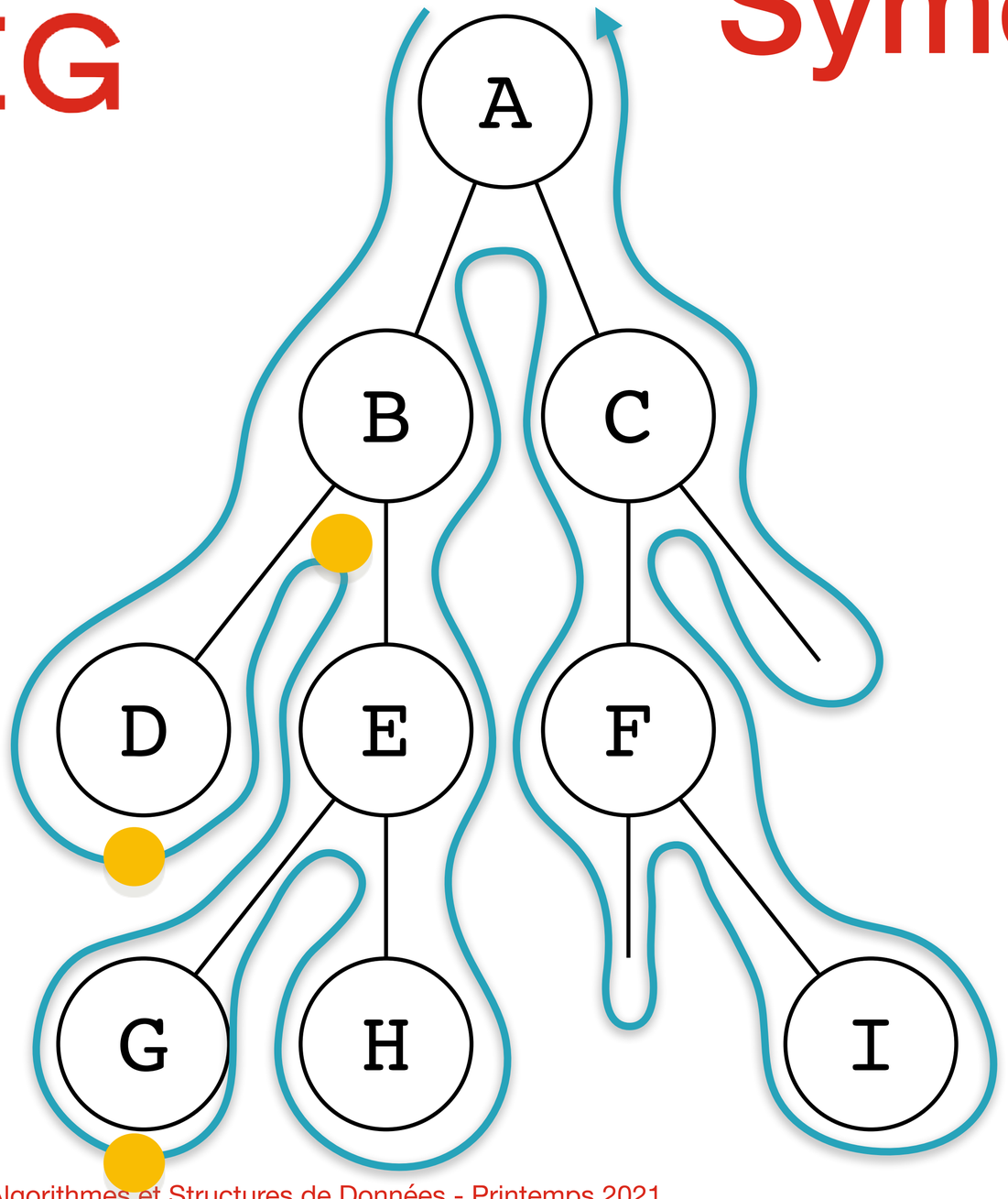



```
fonction symétrique (r, fn)
  si r != ∅
     symétrique(r.gauche, fn)
     fn(r)
     symétrique(r.droit,  fn)
```

- Sous-arbre gauche
- Puis racine
- Puis sous-arbre droit

`D-B`

Symétrique

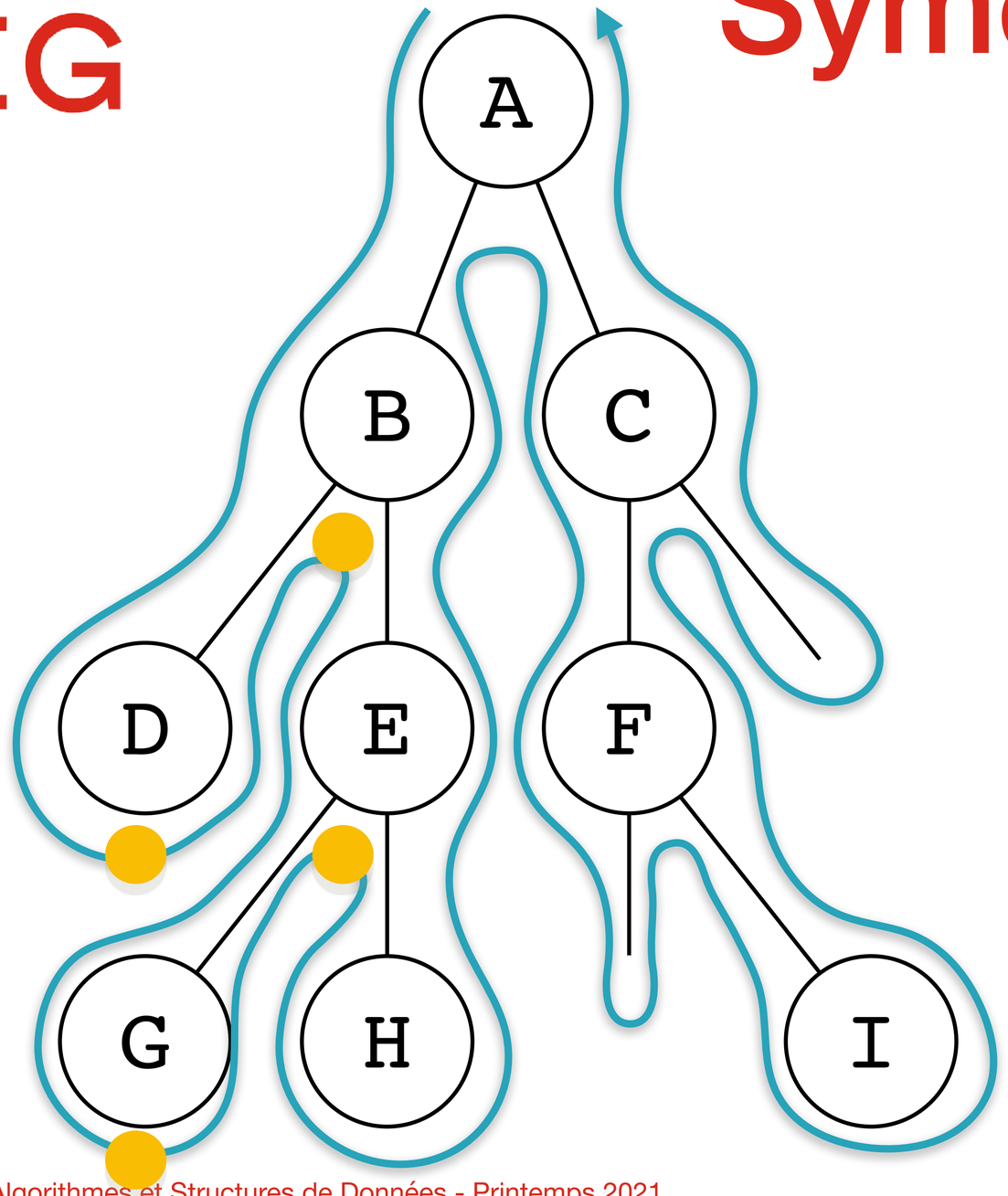



```
fonction symétrique (r, fn)
  si r != ∅
     symétrique(r.gauche, fn)
     fn(r)
     symétrique(r.droit,  fn)
```

- Sous-arbre gauche
- Puis racine
- Puis sous-arbre droit

`D-B-C`

Symétrique

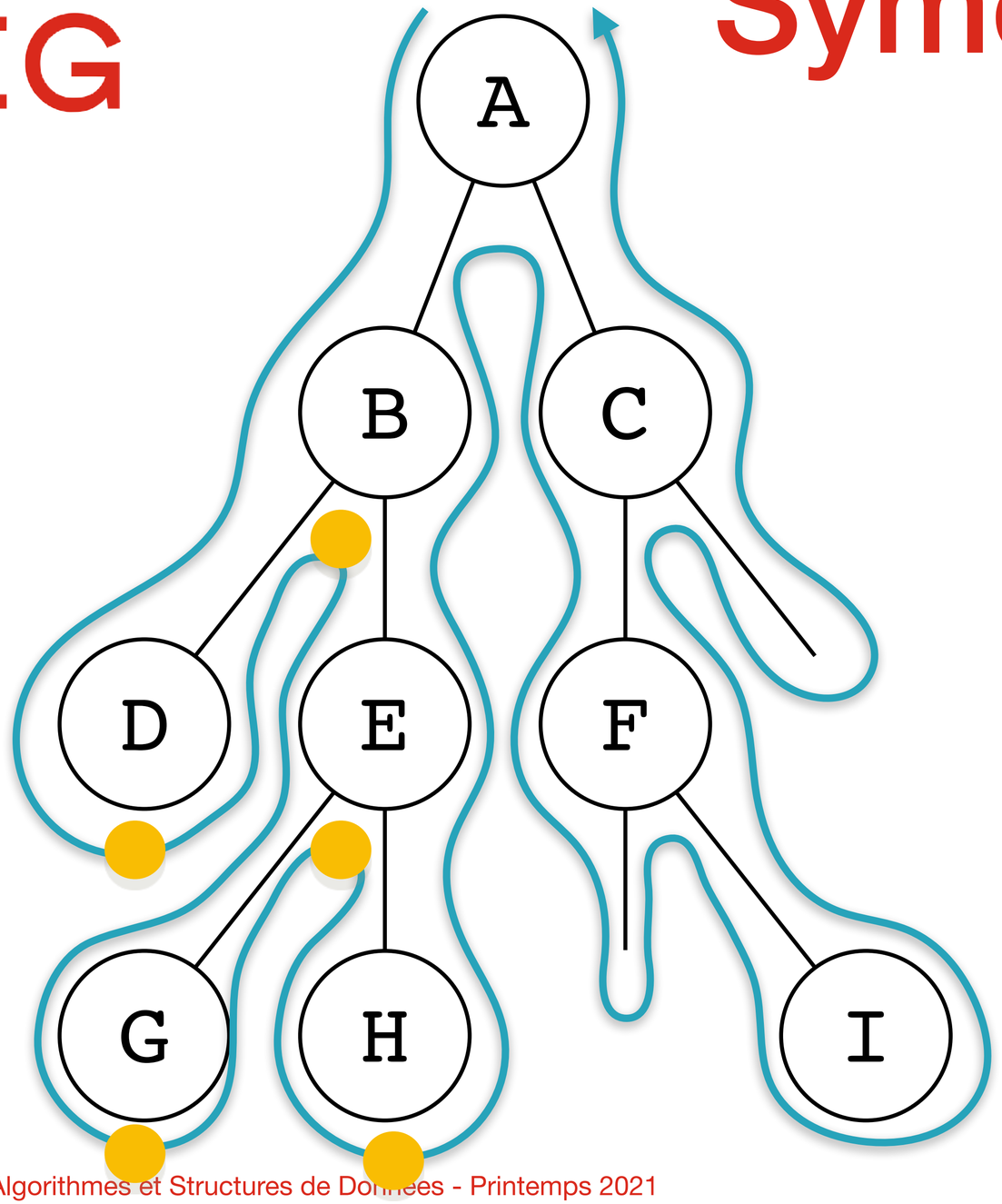



```
fonction symétrique (r, fn)
  si r != ∅
     symétrique(r.gauche, fn)
     fn(r)
     symétrique(r.droit,  fn)
```

- Sous-arbre gauche
- Puis racine
- Puis sous-arbre droit

`D-B-C-E`

Symétrique

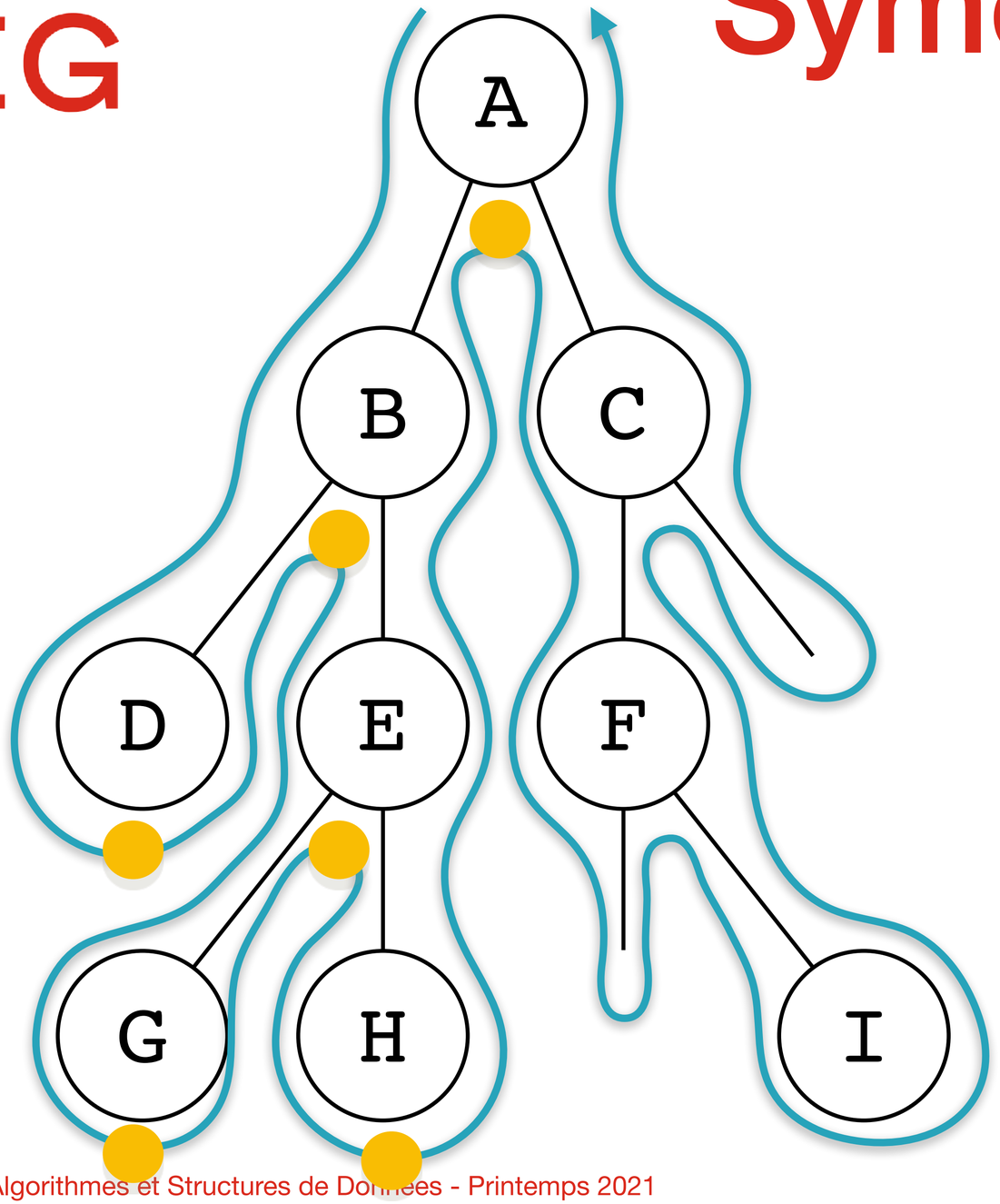



```
fonction symétrique (r, fn)
  si r != ∅
     symétrique(r.gauche, fn)
     fn(r)
     symétrique(r.droit,  fn)
```

- Sous-arbre gauche
- Puis racine
- Puis sous-arbre droit

`D-B-C-E-H`

Symétrique

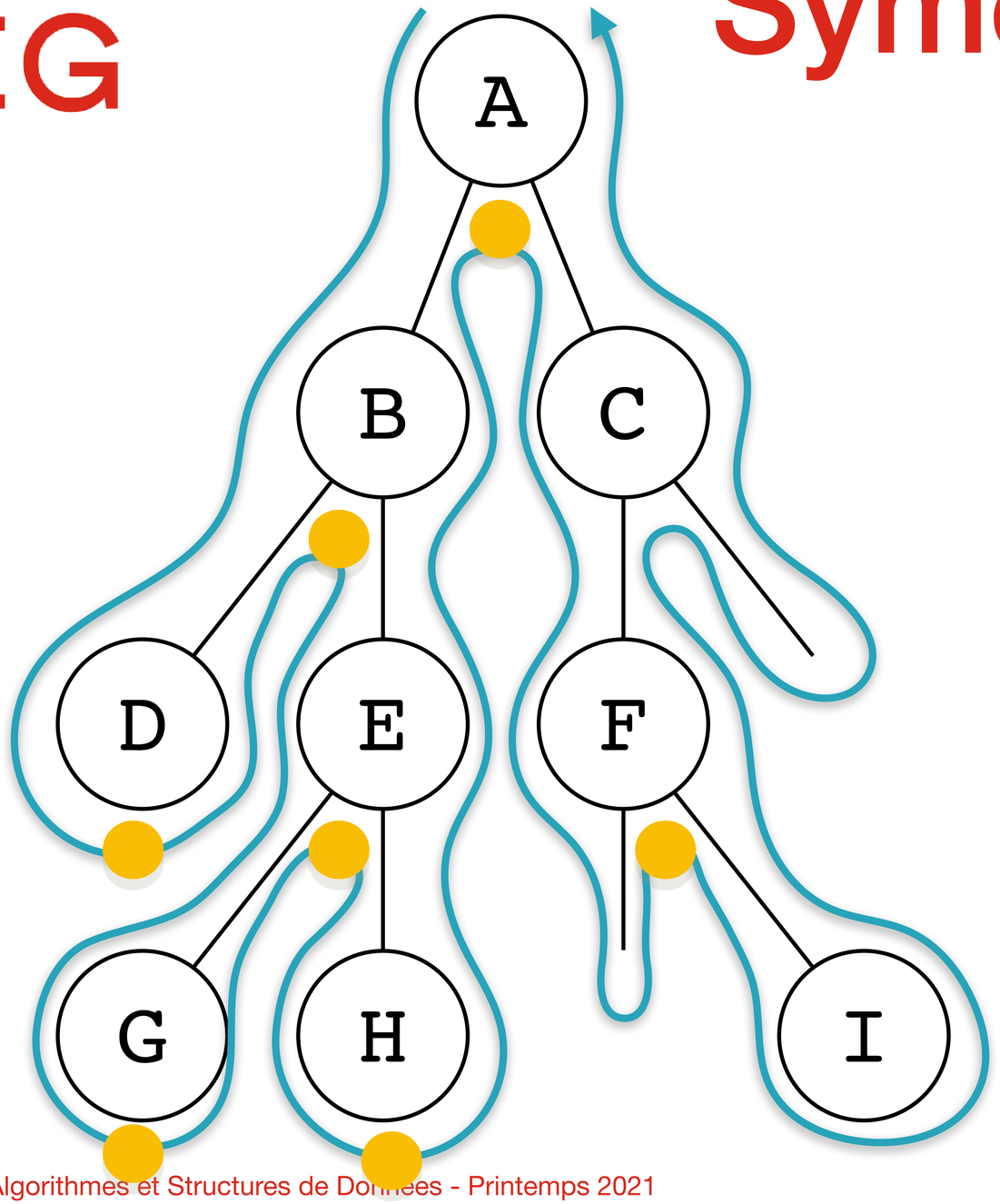



```
fonction symétrique (r, fn)
  si r != ∅
     symétrique(r.gauche, fn)
     fn(r)
     symétrique(r.droit,  fn)
```

- Sous-arbre gauche
- Puis racine
- Puis sous-arbre droit

`D-B-C-E-H-A`

Symétrique

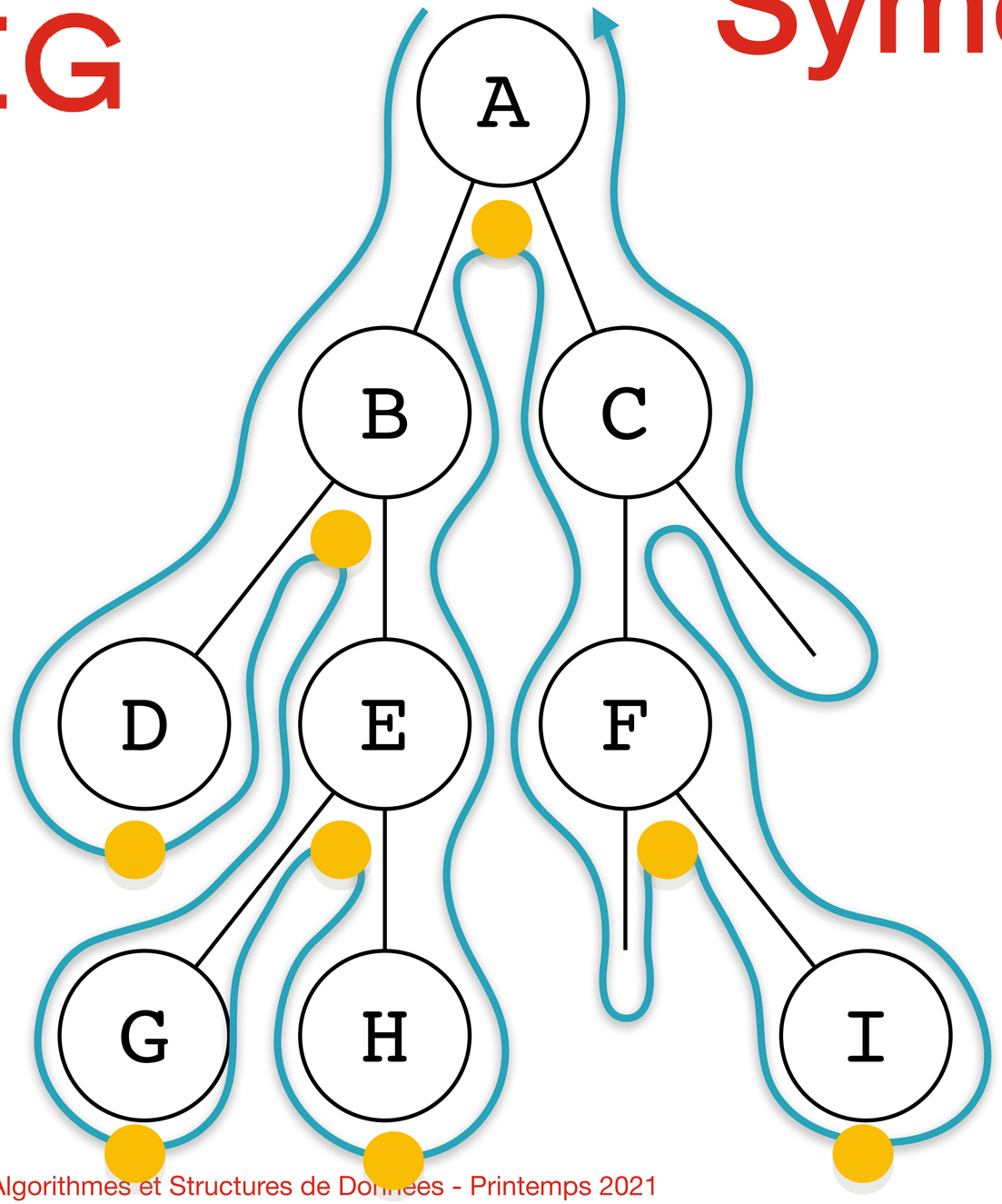



```
fonction symétrique (r, fn)
  si r != ∅
    symétrique(r.gauche, fn)
    fn(r)
    symétrique(r.droit,  fn)
```

- Sous-arbre gauche
- Puis racine
- Puis sous-arbre droit

`D-B-C-E-H-A-F`

Symétrique

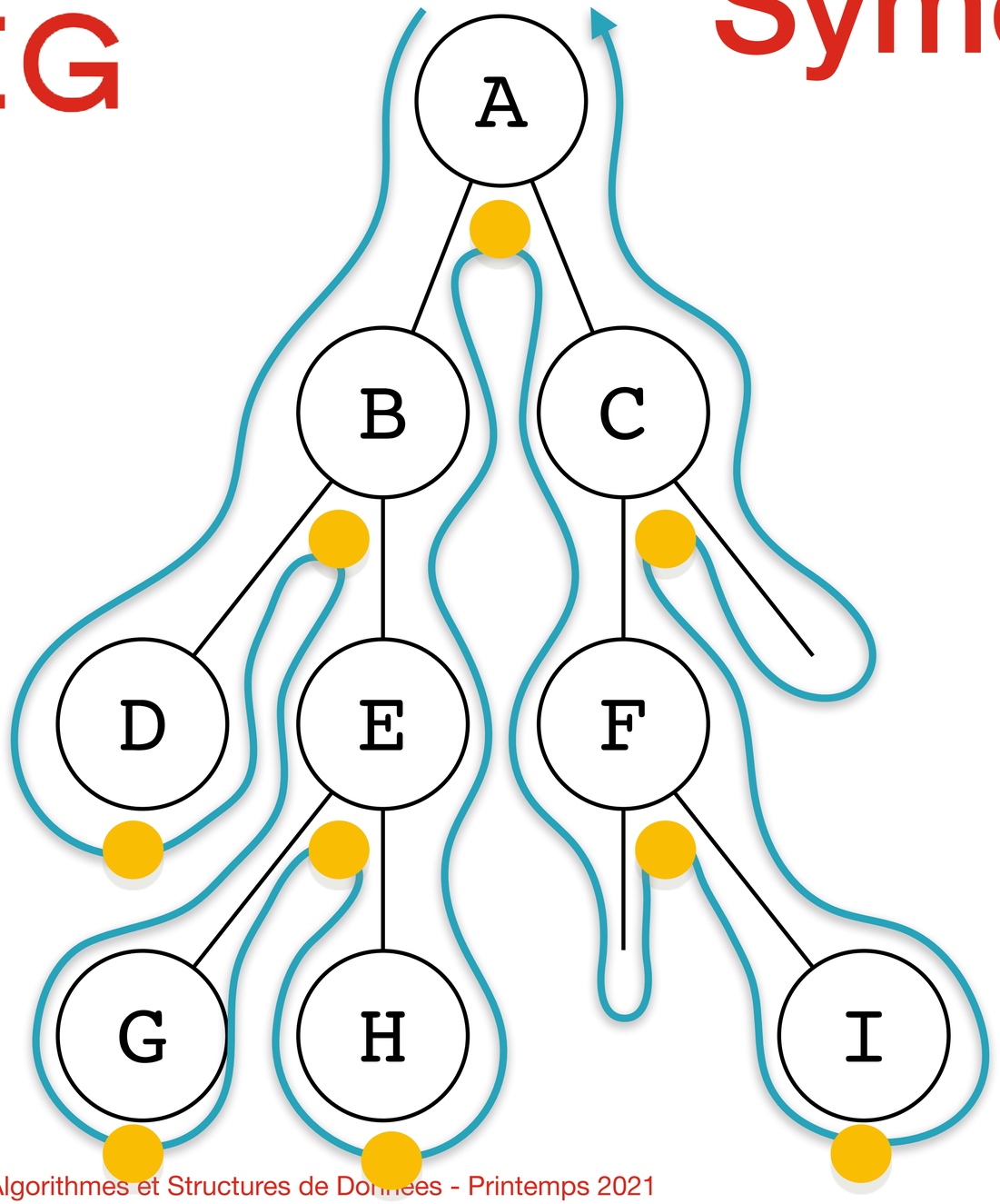



```
fonction symétrique (r, fn)
  si r != ∅
    symétrique(r.gauche, fn)
    fn(r)
    symétrique(r.droit,  fn)
```

- Sous-arbre gauche
- Puis racine
- Puis sous-arbre droit

`D-B-C-E-H-A-F-I`

Symétrique



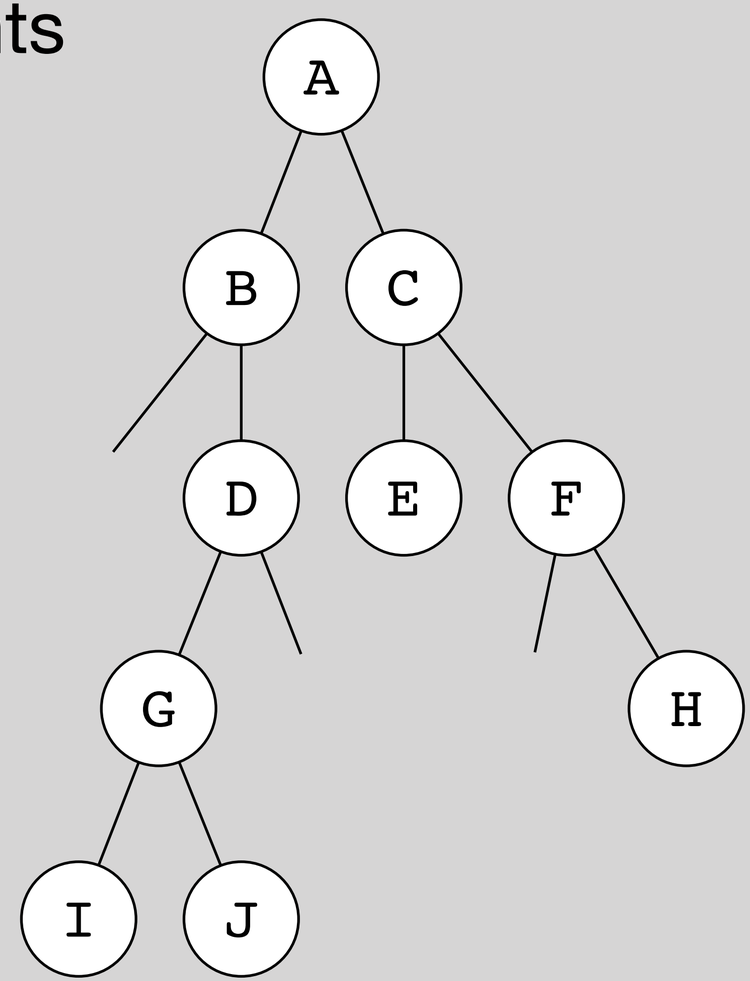

```
fonction symétrique (r, fn)
  si r != ∅
    symétrique(r.gauche, fn)
    fn(r)
    symétrique(r.droit,  fn)
```

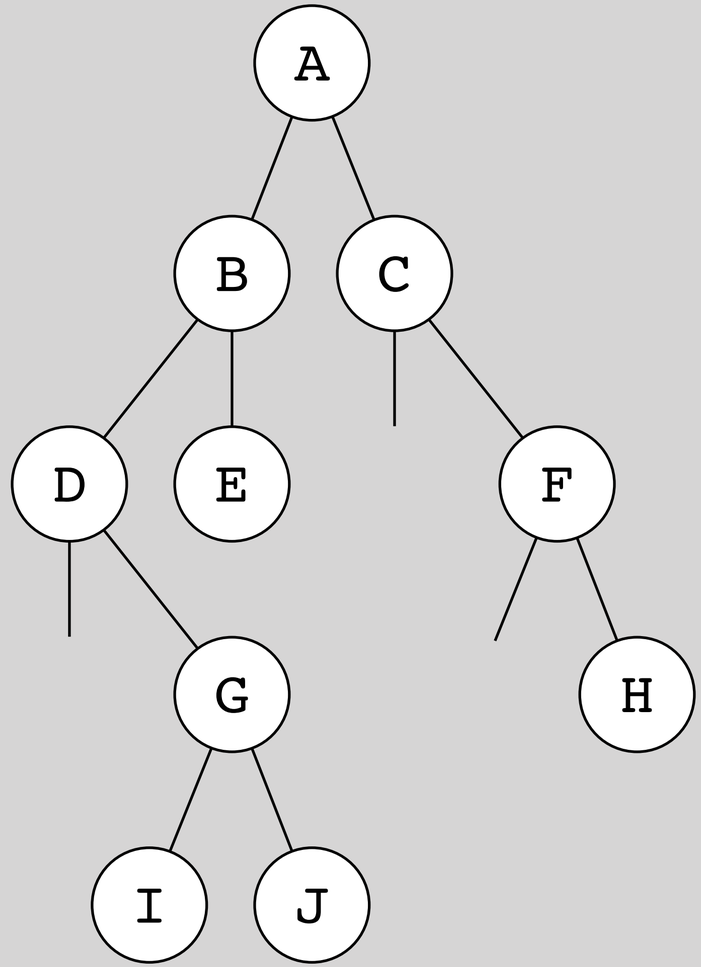
- Sous-arbre gauche
- Puis racine
- Puis sous-arbre droit

D-B-C-E-H-A-F-I-C

Exercices 1 & 2

- Effectuez les parcours pré-ordonné, symétrique et post-ordonnés des arbres suivants





Solution 1

- Pré-ordonné

  `A-B-D-G-I-J-C-E-F-H`

- Symétrique

  `B-I-G-J-D-A-E-C-F-H`

- Post-ordonné

  `I-J-G-D-B-E-H-F-C-A`

Solution 2

- Pré-ordonné

  `A-B-D-G-I-J-E-C-F-H`

- Symétrique

  `D-I-G-J-B-E-A-C-F-H`

- Post-ordonné

  `I-J-G-D-E-B-H-F-C-A`

Arbre à partir de 2 parcours

- Pré-ordonné:

```
A-B-D-E-C-F-G
```

- Symétrique:

```
D-B-E-A-G-F-C
```

Arbre à partir de 2 parcours

- Pré-ordonné:

  A-B-D-E-C-F-G

- Symétrique:

  D-B-E-A-G-F-C

- Le parcours pré- ou post-ordonné permet de connaître la racine

Arbre à partir de 2 parcours

- Pré-ordonné:

A-B-D-E-C-F-G

- Symétrique:

D-B-E-A-G-F-C

- Le parcours pré- ou post-ordonné permet de connaître la racine
- Connaissant la racine A,

Arbre à partir de 2 parcours

- Pré-ordonné:

A-B-D-E-C-F-G

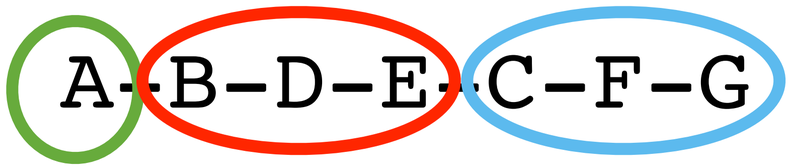
- Symétrique:

D-B-E-A-G-F-C

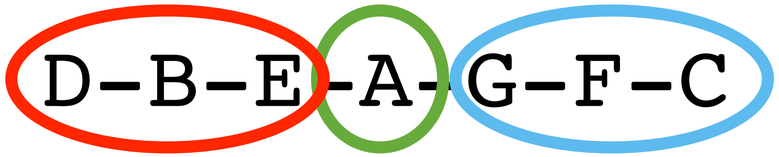
- Le parcours pré- ou post-ordonné permet de connaître la racine
- Connaissant la racine A,
- … le parcours symétrique définit les sous-arbres gauche

Arbre à partir de 2 parcours

- Pré-ordonné:



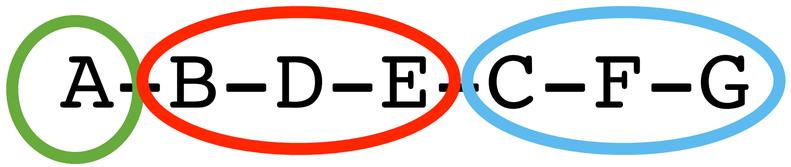
- Symétrique:



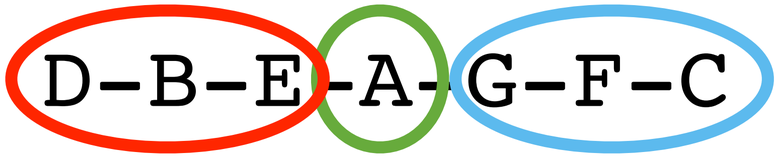
- Le parcours pré- ou post-ordonné permet de connaître la racine
- Connaissant la racine A,
- … le parcours symétrique définit les sous-arbres gauche
- … et droit

Arbre à partir de 2 parcours

- Pré-ordonné:



- Symétrique:



- Le parcours pré- ou post-ordonné permet de connaître la racine
- Connaissant la racine A,
- … le parcours symétrique définit les sous-arbres gauche
- … et droit
- Et récursivement

Exercices 3 & 4

- Dessiner l'arbre binaire dont on connait les parcours pré-ordonné et symétrique

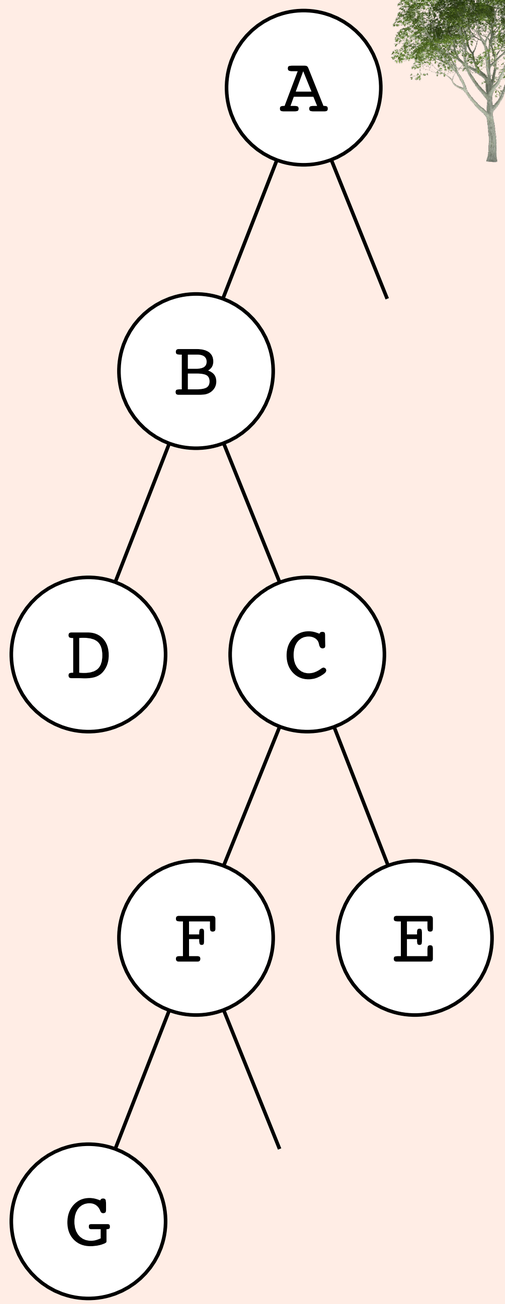
```
Pré-ordonné: A-B-D-C-F-G-E
Symétrique:  D-B-G-F-C-E-A
```

- Dessiner l'arbre binaire dont on connait les parcours symétrique et post-ordonné

```
Symétrique:   E-B-D-A-F-G-C
Post-ordonné: E-D-B-G-F-C-A
```

Solution 3

```
Pré-ordonné: A-B-D-C-F-G-E
Symétrique:  D-B-G-F-C-E-A
```

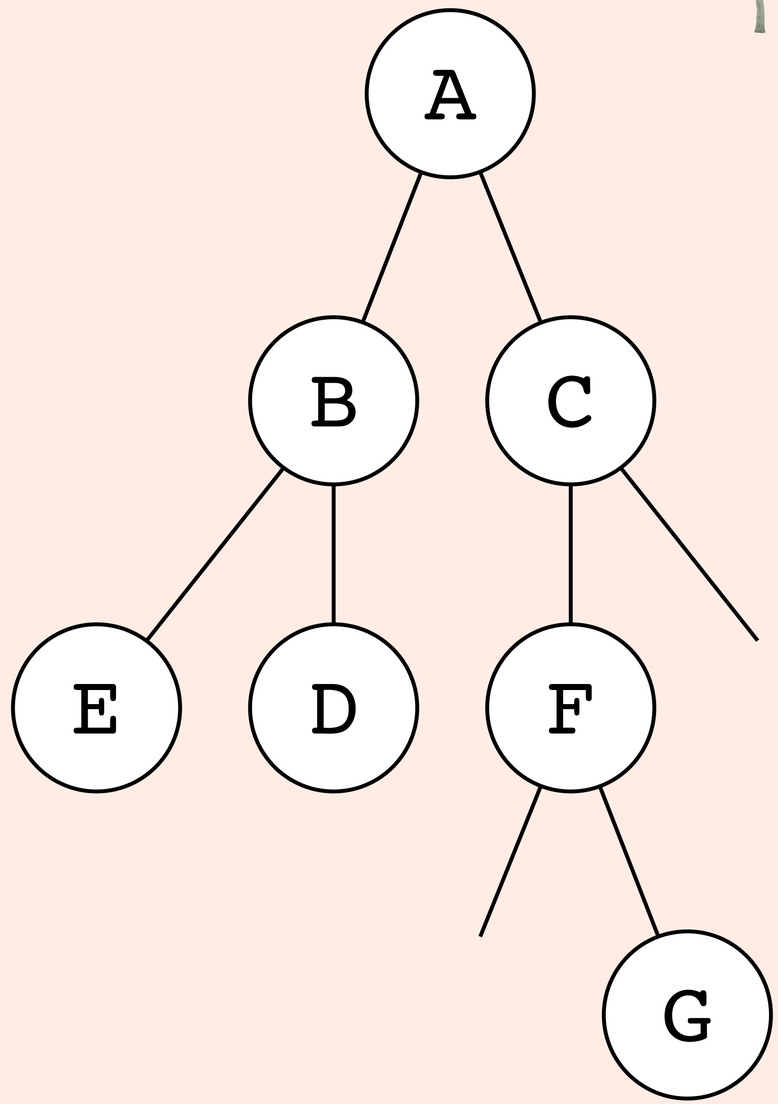


Méthode :

- Le parcours pré-ordonné nous donne la racine : A
- Le parcours symétrique nous indique que le sous-arbre gauche est DBGFCE et que le droit est vide
- Pour le sous-arbre DBGFCE, le pré-ordonné nous donne la racine B et puis le symétrique les sous-arbre gauche (D) et droit (GFCE).
- Pour le sous-arbre GFCE, le pré-ordonné nous donne la racine C et puis le symétrique les sous-arbre gauche (GF) et droit (E)
- Pour le sous-arbre GF, le pré-ordonné nous donne la racine F et puis le symétrique nous indique que G est à sa gauche

Solution 4

```
Symétrique:   E-B-D-A-F-G-C
Post-ordonné: E-D-B-G-F-C-A
```



Méthode :

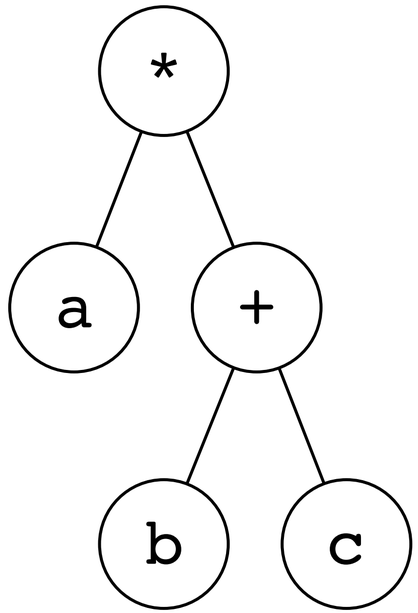
- Le parcours post-ordonné nous donne la racine : A
- Le parcours symétrique nous donne les sous-arbres gauche (EBD) et droit (FGC)
- Pour le sous-arbre EBD, le post-ordonné nous donne la racine B, puis le symétrique nous dit que E est à gauche et D à droite
- Pour le sous-arbre FGC, le post-ordonné nous donne la racine C, puis le symétrique que FC sont à gauche et que le sous-arbre droit est vide
- Pour le sous-arbre FC, le post-ordonné nous dit que F est la racine. Le symétrique nous dit que G est à sa droite

HEIG VD

8. Expressions arithmétiques

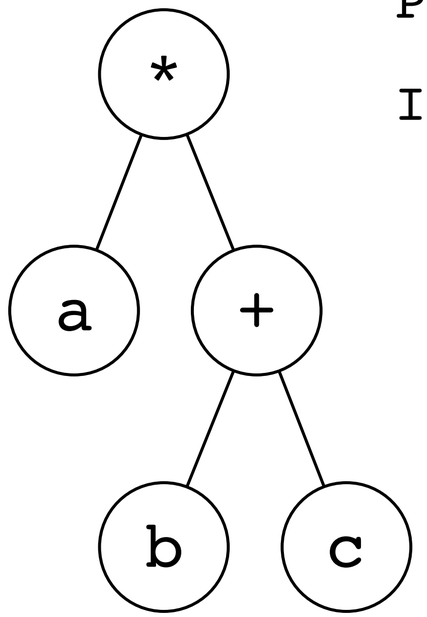
Expressions arithmétiques

- Arbre binaire avec 2 types de noeuds
  - Feuille = valeurs
  - Noeuds interne = opérateurs binaires tels que +, -, *, /
- Pas de noeud de degré 1



Expressions arithmétiques

- Arbre binaire avec 2 types de noeuds
  - Feuille = valeurs
  - Noeuds interne = opérateurs binaires tels que +, -, *, /
- Pas de noeud de degré 1




```
Préfixe:    * a + b c
Postfixe:   a b c + *
Infixe:     (a*(b+c))
```

Notation préfixe

- Parcours pré-ordonné de l'arbre

```
fonction préfixe (r)
  si r != ∅
    afficher r.etiquette
    préfixe(r.gauche)
    préfixe(r.droit)
```

Notation préfixe

- Parcours pré-ordonné de l'arbre
- Pas besoin de parenthèses si l'étiquette permet de connaitre le degré du noeud

```
fonction préfixe (r)
  si r != ∅
    afficher r.etiquette
    préfixe(r.gauche)
    préfixe(r.droit)
```

Notation préfixe

- Parcours pré-ordonné de l'arbre
- Pas besoin de parenthèses si l'étiquette permet de connaitre le degré du noeud
- Utiliser des listes imbriquées sinon, comme dans le langage lisp

```
fonction préfixe (r)
  si r != ∅
    afficher r.etiquette
    préfixe(r.gauche)
    préfixe(r.droit)
```

Notation préfixe

- Parcours pré-ordonné de l'arbre
- Pas besoin de parenthèses si l'étiquette permet de connaitre le degré du noeud
- Utiliser des listes imbriquées sinon, comme dans le langage lisp
- La fonction inverse a la même structure

```
fonction préfixe (r)
  si r != ∅
    afficher r.etiquette
    préfixe(r.gauche)
    préfixe(r.droit)
```

```
fonction arbre_depuis_préfixe (flux)
  e ← lire(flux)
  r ← nouveau noeud d'étiquette e
  si e est un opérateur
    r.gauche ← arbre_depuis_préfixe (flux)
    r.droit  ← arbre_depuis_préfixe (flux)
  retourner r
```

Notation postfixe

- Parcours post-ordonné de l'arbre

```
fonction postfixe (r)
  si r != ∅
    postfixe(r.gauche)
    postfixe(r.droit)
    afficher r.etiquette
```

Notation postfixe

- Parcours post-ordonné de l'arbre
- Permet d'évaluer l'expression, les opérandes étant connues avant l'opérateur

```
fonction postfixe (r)
  si r != ∅
    postfixe(r.gauche)
    postfixe(r.droit)
    afficher r.etiquette
```

Notation postfixe

- Parcours post-ordonné de l'arbre
- Permet d'évaluer l'expression, les opérandes étant connues avant l'opérateur
- Notation utilisées dans la calculatrices sous le nom de RPN

```
fonction postfixe (r)
  si r != ∅
    postfixe(r.gauche)
    postfixe(r.droit)
    afficher r.etiquette
```

Notation postfixe

- Parcours post-ordonné de l'arbre
- Permet d'évaluer l'expression, les opérandes étant connues avant l'opérateur
- Notation utilisées dans la calculatrices sous le nom de RPN
- Pour la fonction inverse, le plus simple consiste à lire le flux à l'envers

```
fonction postfixe (r)
  si r != ∅
    postfixe(r.gauche)
    postfixe(r.droit)
    afficher r.etiquette
```

```
fonction arbre_depuis_postfixe (flux)
  e ← lire(flux inversé)
  r ← nouveau noeud d'étiquette e
  si e est un opérateur
    r.droit ← arbre_depuis_postfixe (flux)
    r.gauche ← arbre_depuis_postfixe (flux)
  retourner r
```

Notation infixe

- Parcours symétrique de l'arbre

```
fonction infixe (r)
  si r est un noeud interne
    afficher parenthèse ouvrante (
    infixe(r.gauche)
    afficher r.etiquette
    infixe(r.droit)
    afficher parenthèse fermante )
  sinon (r est une feuille)
    afficher r.etiquette
```

Notation infixe

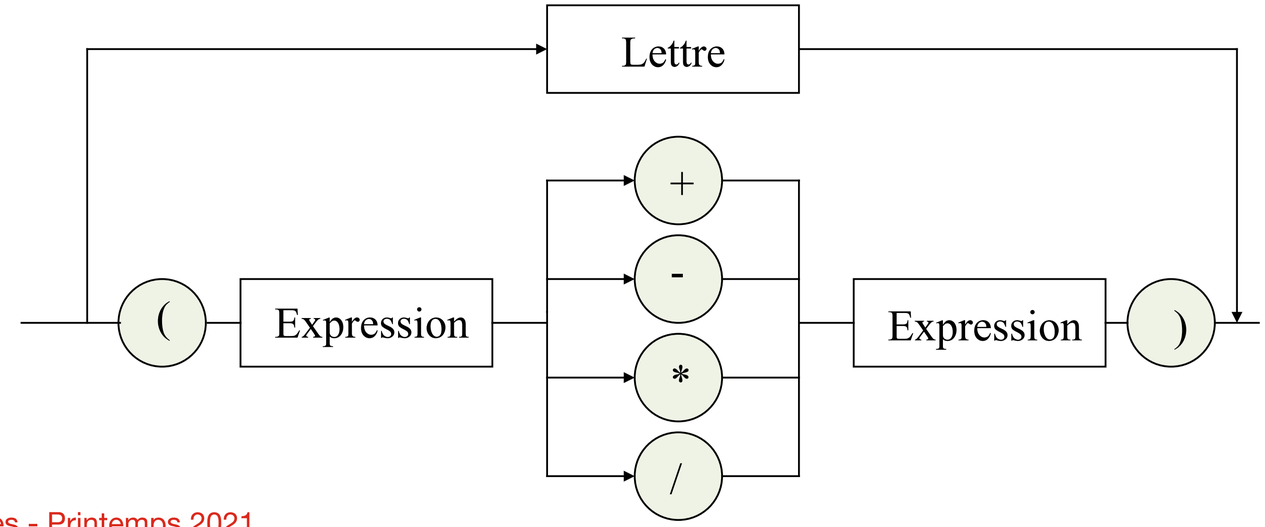
- Parcours symétrique de l'arbre
- Besoin de parenthèses pour lever les ambiguïtés

```
fonction infixe (r)
  si r est un noeud interne
    afficher parenthèse ouvrante (
    infixe(r.gauche)
    afficher r.etiquette
    infixe(r.droit)
    afficher parenthèse fermante )
  sinon (r est une feuille)
    afficher r.etiquette
```

Notation infixe

- Parcours symétrique de l'arbre
- Besoin de parenthèses pour lever les ambiguïtés
- Diagramme syntaxique

```
fonction infixe (r)
  si r est un noeud interne
    afficher parenthèse ouvrante (
    infixe(r.gauche)
    afficher r.etiquette
    infixe(r.droit)
    afficher parenthèse fermante )
  sinon (r est une feuille)
    afficher r.etiquette
```



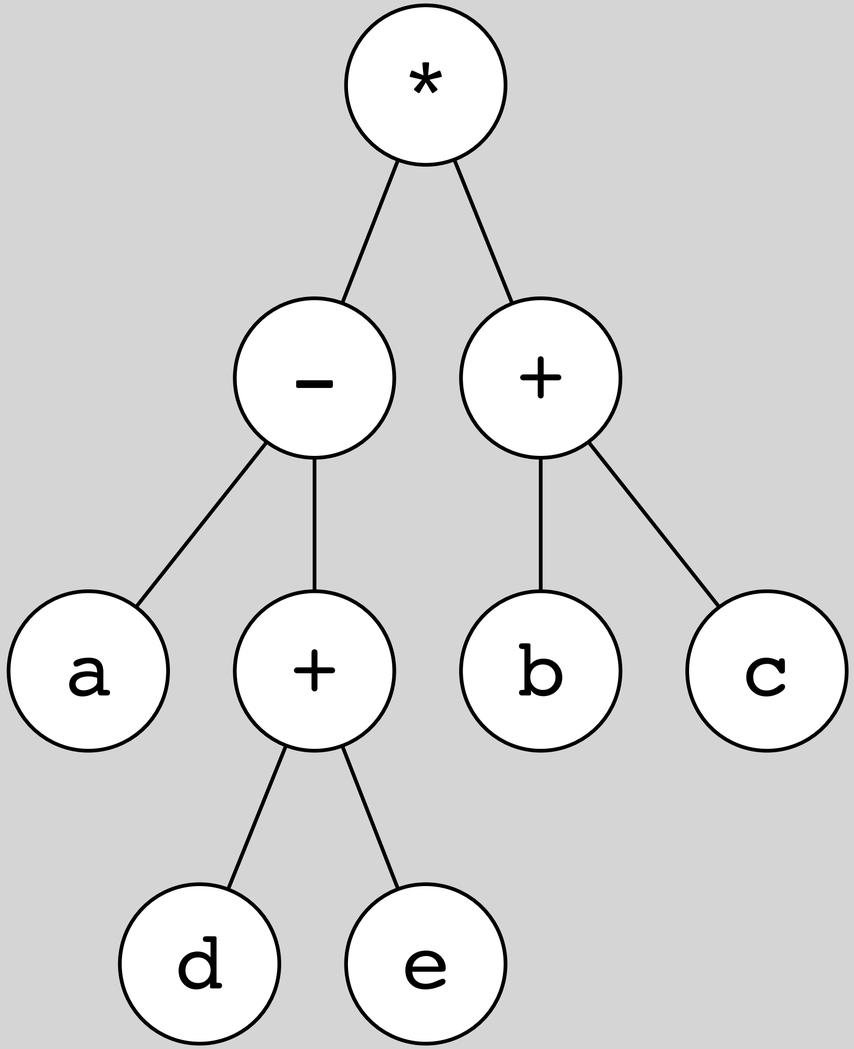
Arbre depuis expression infixe

- Même structure de code que la fonction inverse
- Les parenthèses ouvrantes distinguent noeuds internes et feuilles
- Les parenthèses fermantes sont inutiles.

```
fonction arbre_depuis_infixe (flux)
  e ← lire(flux)
  si e est une parenthèse ouvrante (
    rg ← arbre_depuis_infixe(flux)
    e  ← lire(flux) (e est un opérateur)
    r  ← nouveau noeud d'étiquette e
    r.gauche ← rg
    r.droite ← arbre_depuis_infixe(flux)
    lire(flux) (parenthèse fermante)
  sinon (r est une feuille)
    r ← nouveau noeud d'étiquette e
  retourner r
```

Exercice 1

- Soit l'expression mathématique modélisée par l'arbre ci-contre. Donnez-en les notations
  - préfixe
  - post-fixe
  - et infixe



Solutions 1

- Notation préfixe :

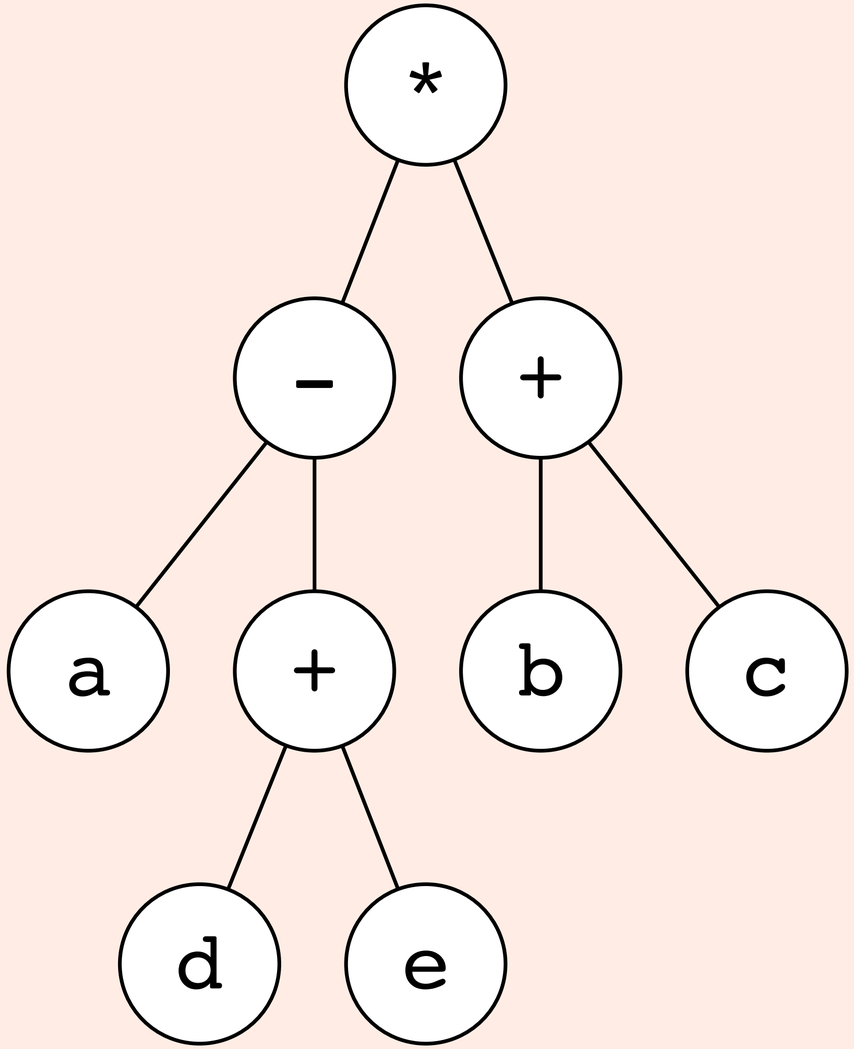
```
* - a + d e + b c
```

- Notation postfixe :

```
a d e + - b c + *
```

- Notation infixe :

```
((a-(d+e))*(b+c))
```



Exercice 2

- Ecrire la version post fixe de l'expression infixe

  `(((a+b)*(c-d))/(a-b))`

- Ecrire la version infixe de l'expression préfixe

  `+ a - + b c * d e`

- Ecrire la version préfixe de l'expression postfixe

  `a b + c d e - * /`

Solutions 2

1. Infixe: (((a+b)*(c-d))/(a-b))

   Postfixe: a b + c d - * a b - /

2. Préfixe: + a - + b c * d e

   Infixe: (a+((b+c)-(d*e)))

3. Postfixe: a b + c d e - * /

   Préfixe: / + a b * c - d e