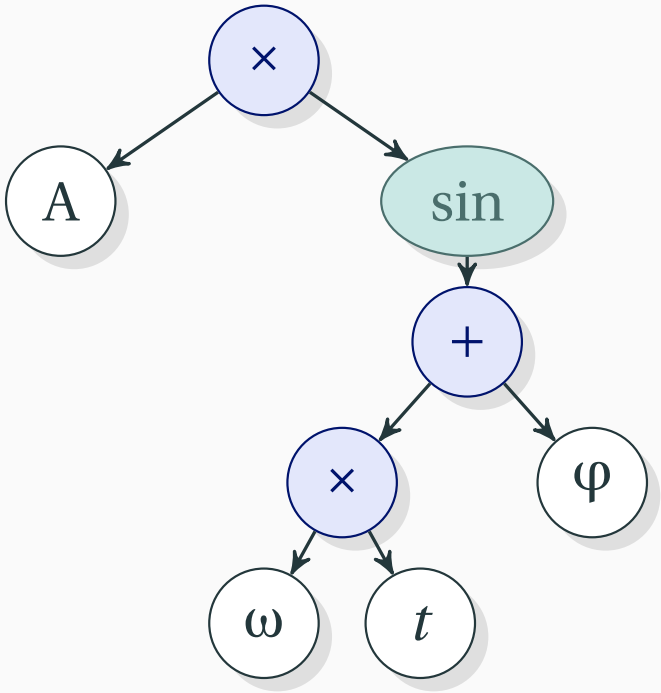
Introduction aux arbres

G. Dewaele

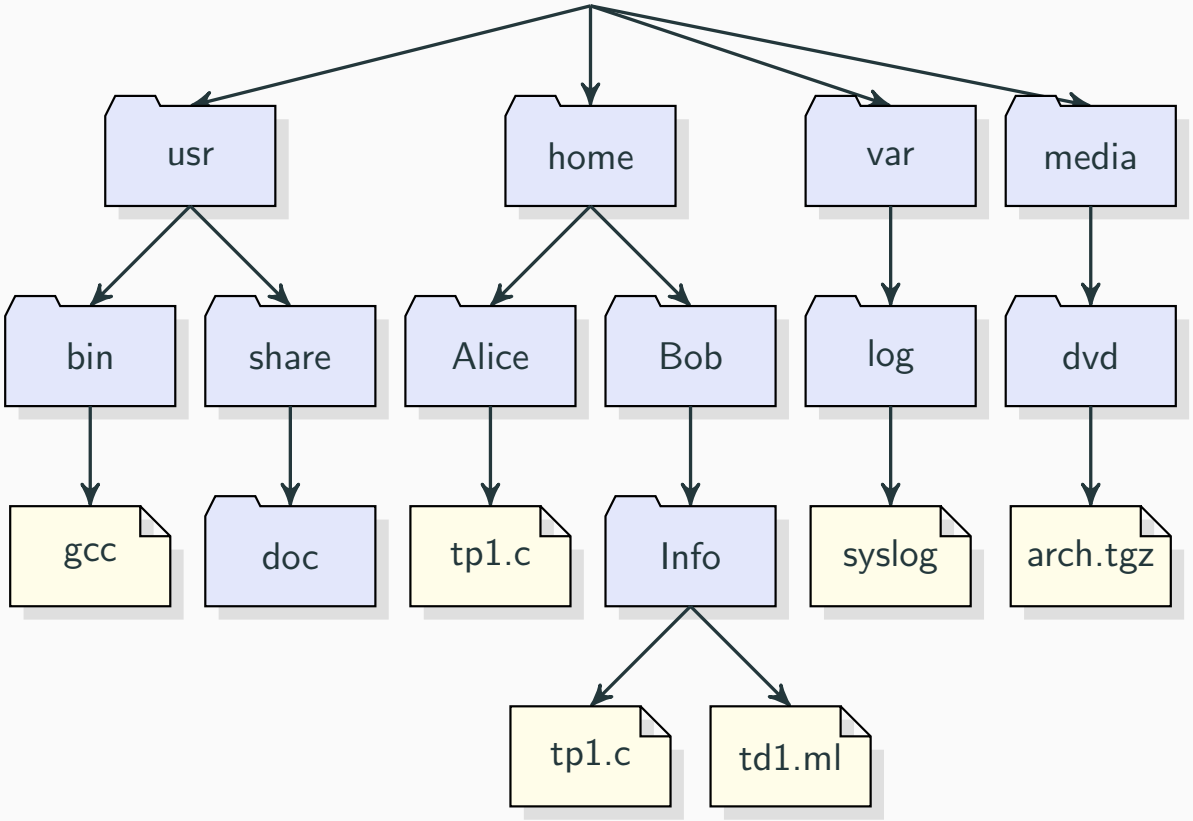
12 mars 2026

Lycée Louis-le-Grand

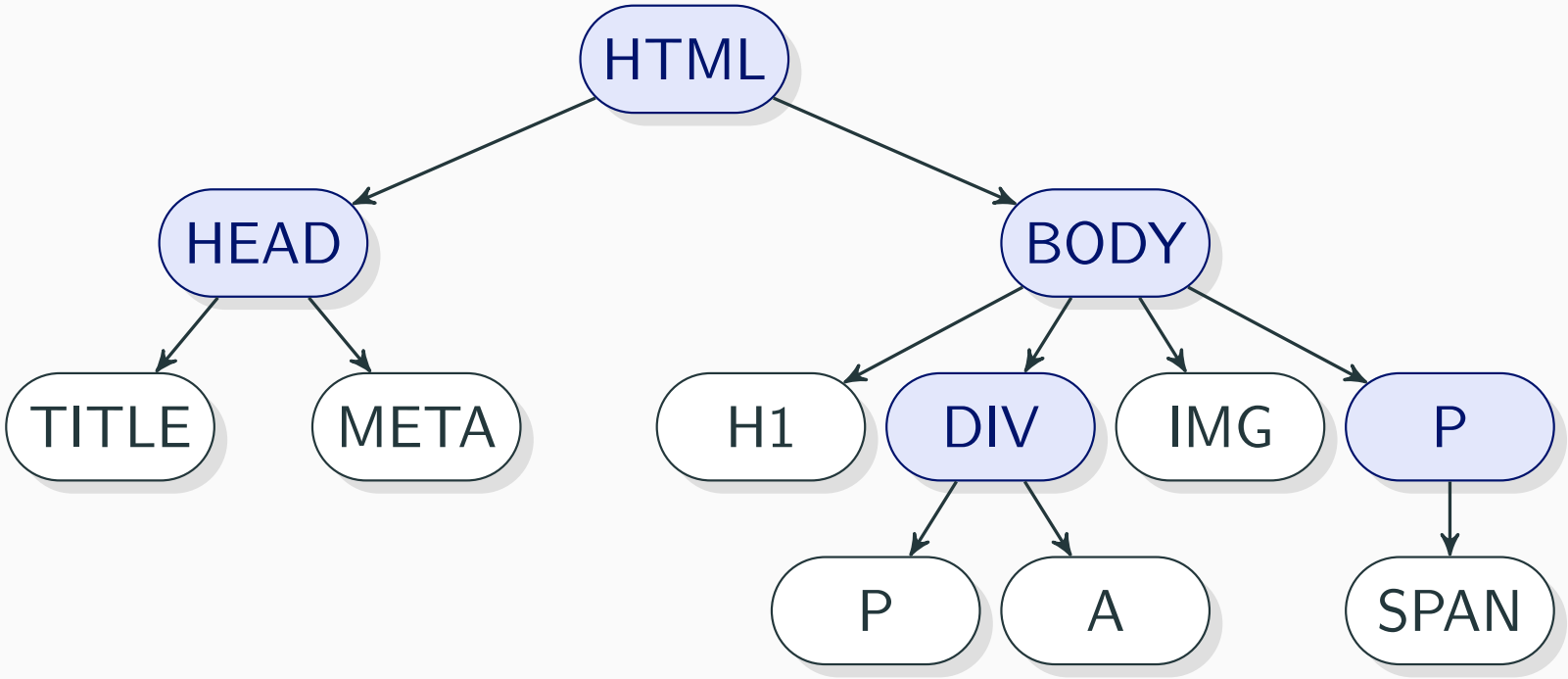
Arbres d'expressions



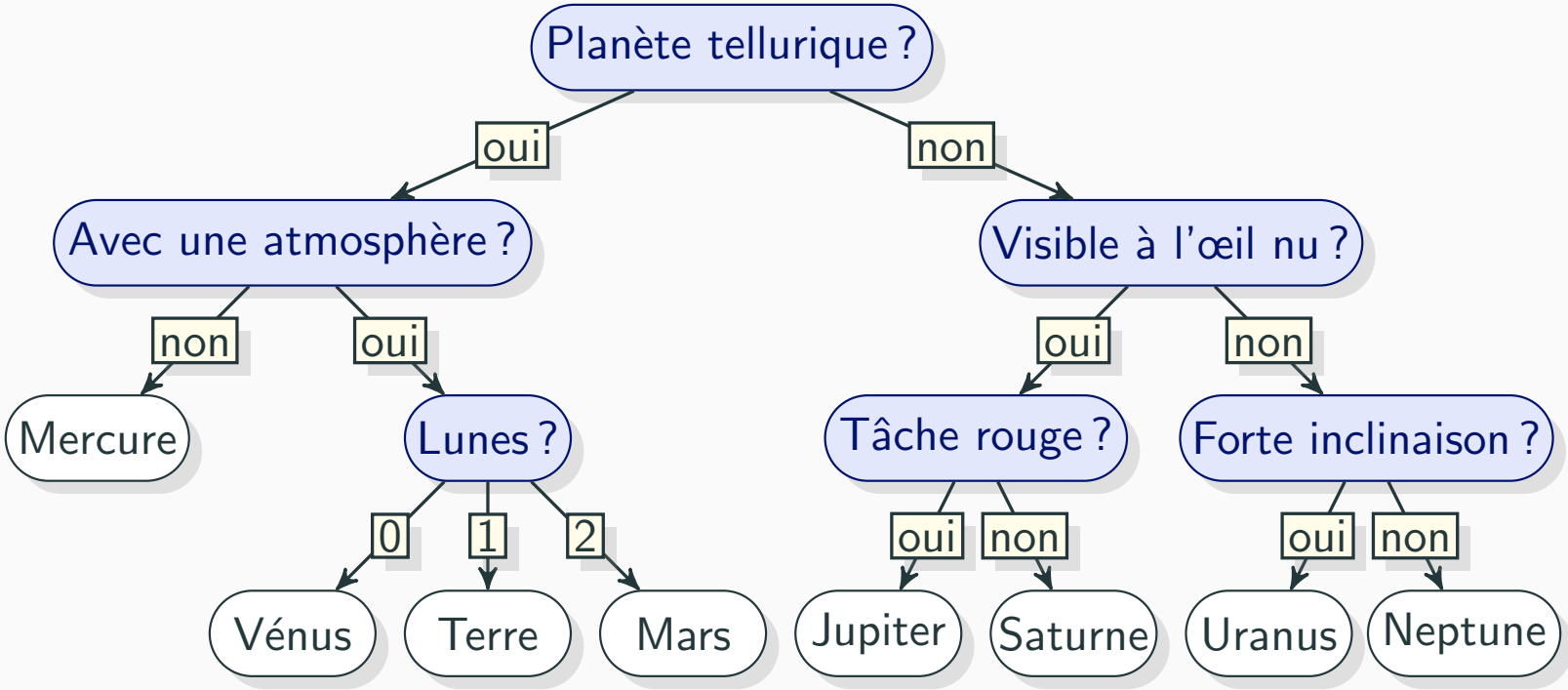
Arborescence de fichiers



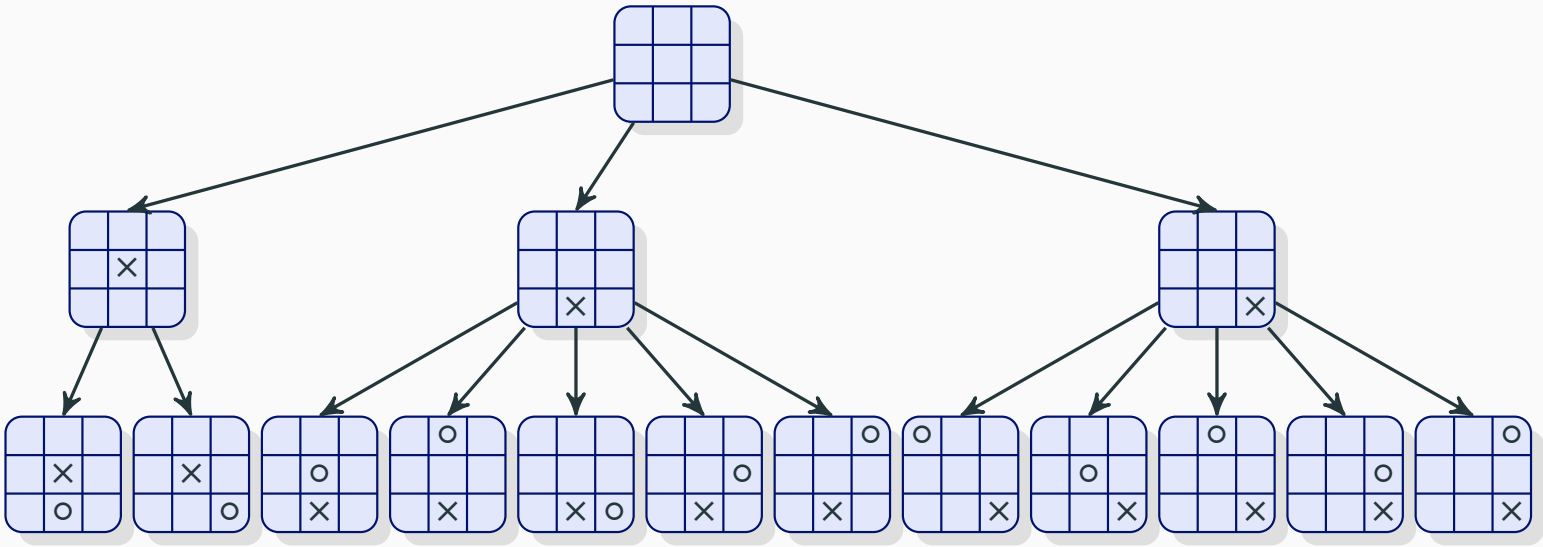
Documents structurés



Arbres de décision



Théorie des jeux



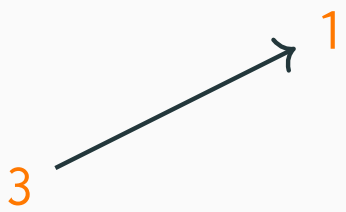
Rappels sur les listes chaînées

Nous avons étudié le principe des listes chaînées :

3

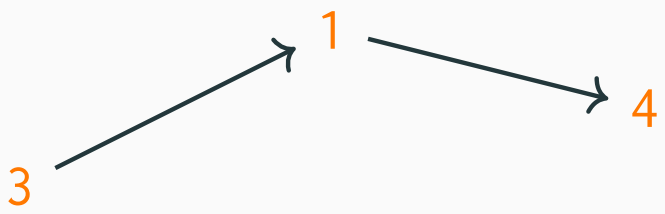
Rappels sur les listes chaînées

Nous avons étudié le principe des listes chaînées :



Rappels sur les listes chaînées

Nous avons étudié le principe des listes chaînées :



Rappels sur les listes chaînées

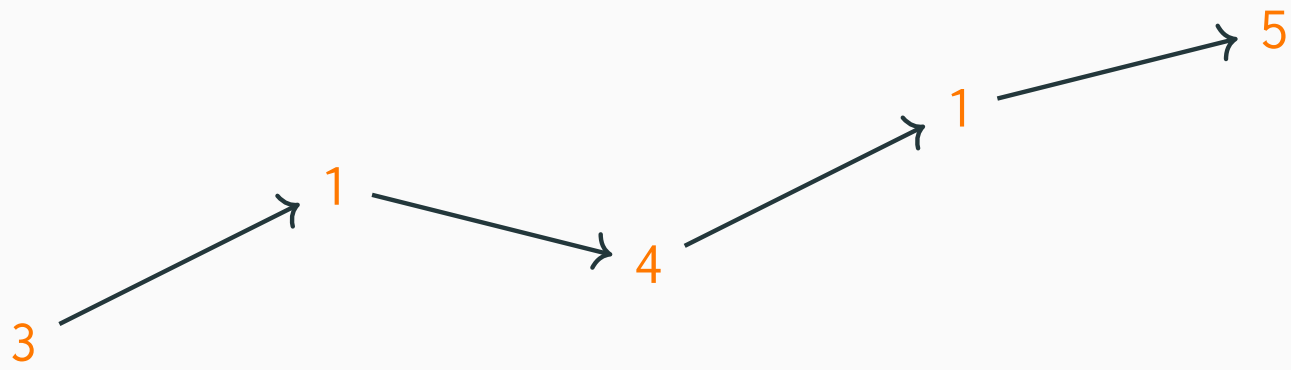
Nous avons étudié le principe des listes chaînées :

Rappels sur les listes chaînées

Nous avons étudié le principe des listes chaînées :

Rappels sur les listes chaînées

Nous avons étudié le principe des listes chaînées :



Elles peuvent être définies en Caml par :

```
type 'a chained_list =
  | Nil
  | Cell of { element : 'a; next : 'a chained_list }
```

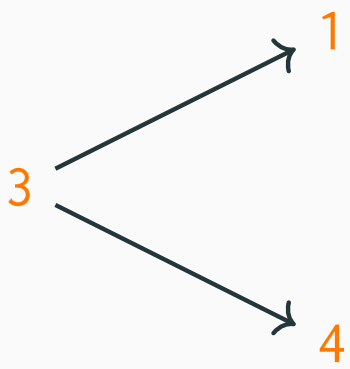
Les arbres

Pour un arbre, on peut avoir plusieurs suivants :

3

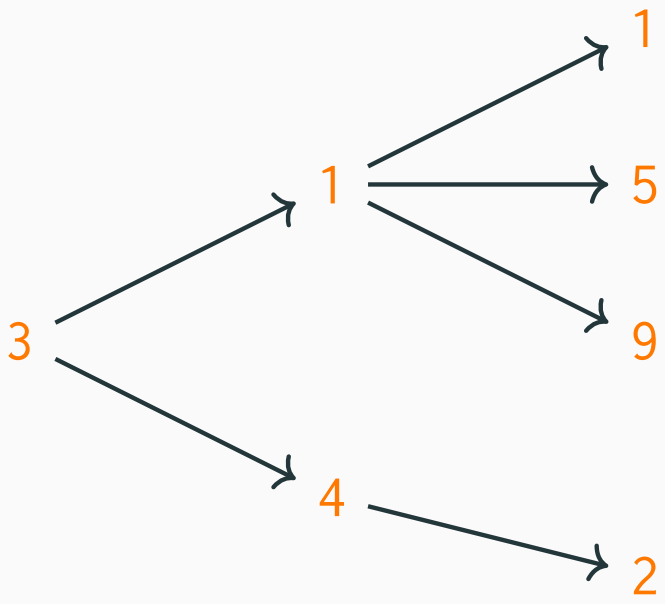
Les arbres

Pour un arbre, on peut avoir plusieurs suivants :



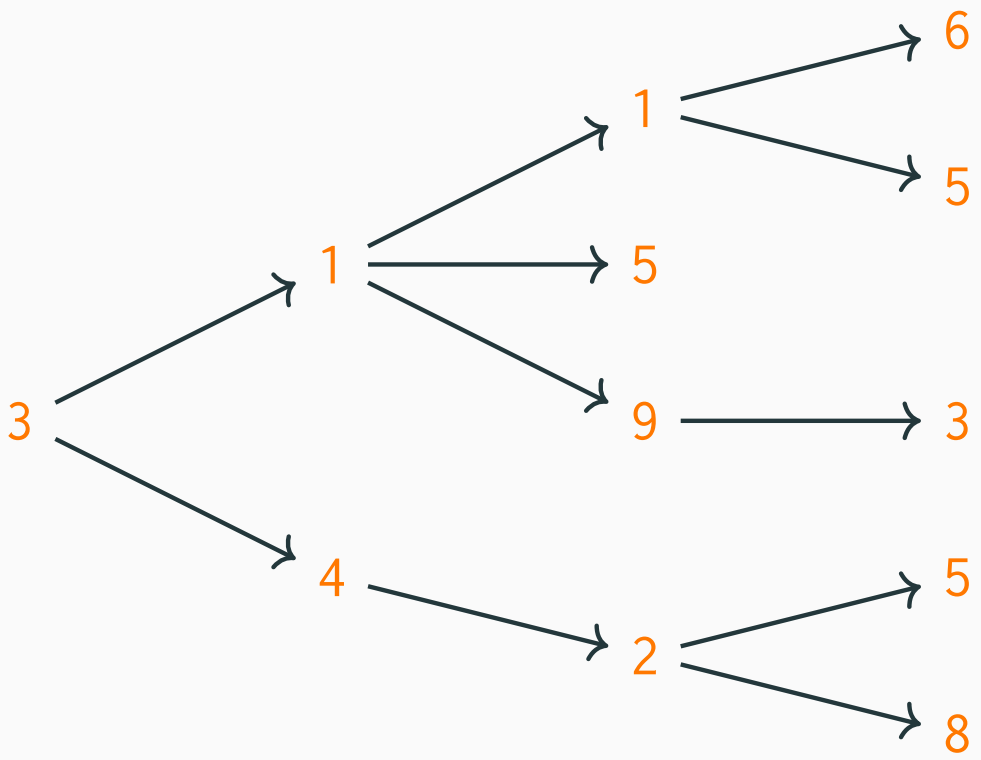
Les arbres

Pour un arbre, on peut avoir plusieurs suivants :



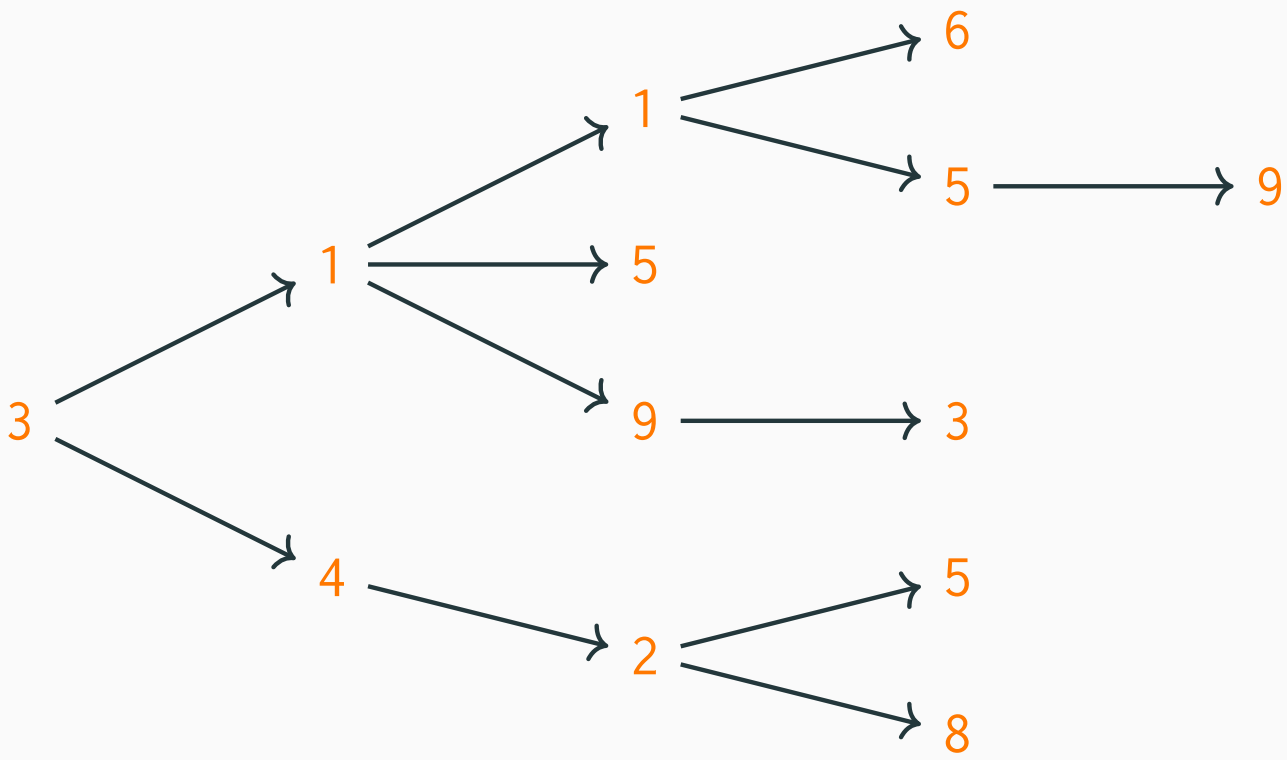
Les arbres

Pour un arbre, on peut avoir plusieurs suivants :



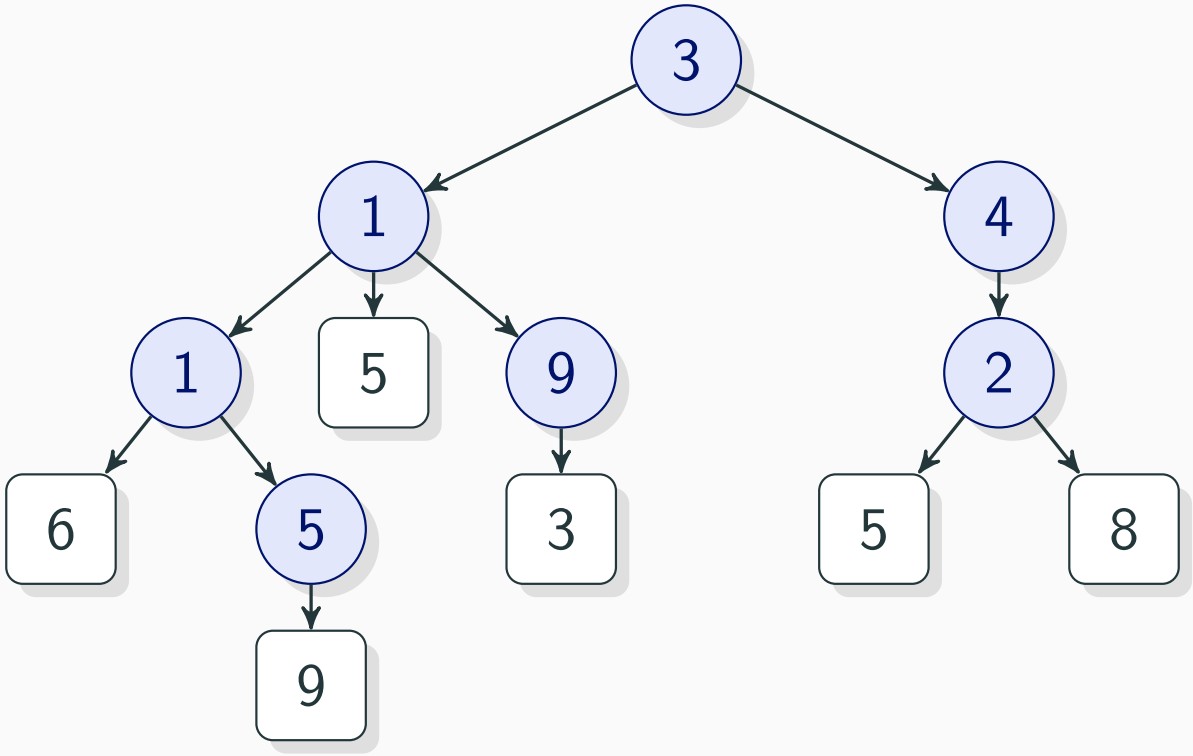
Les arbres

Pour un arbre, on peut avoir plusieurs suivants :



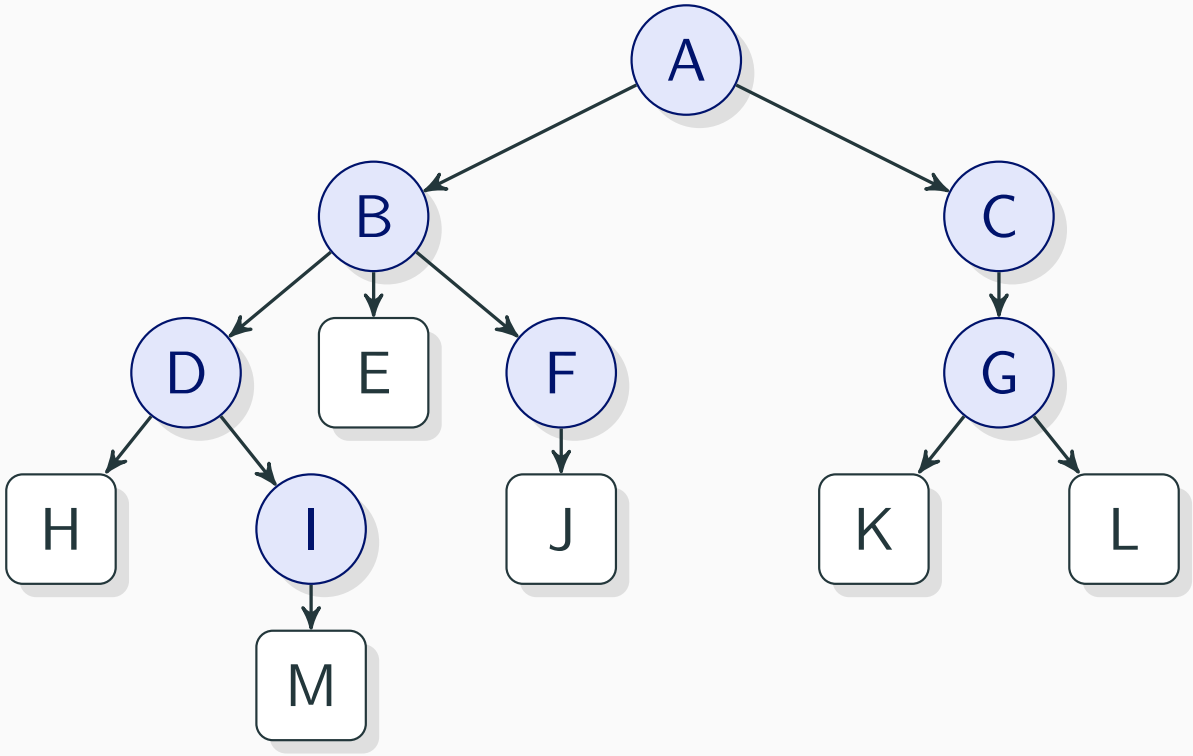
Les arbres

On représente fréquemment les arbres de haut en bas :



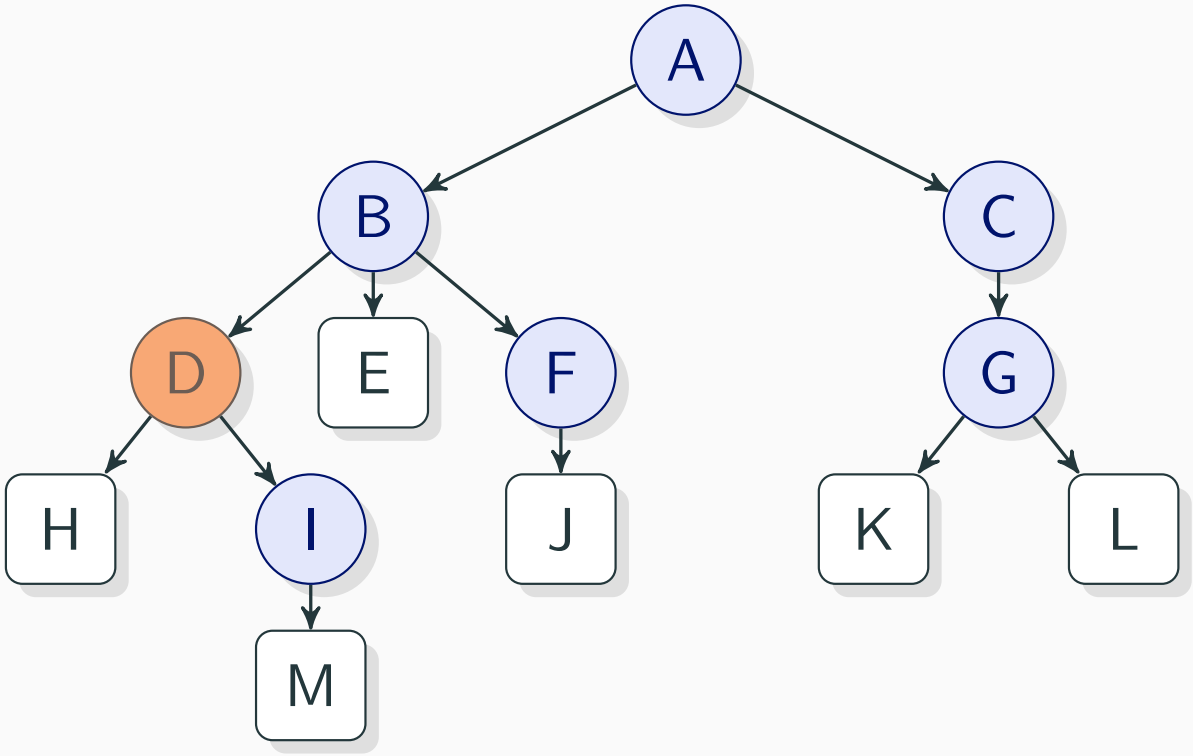
Les arbres

On représente fréquemment les arbres de haut en bas :



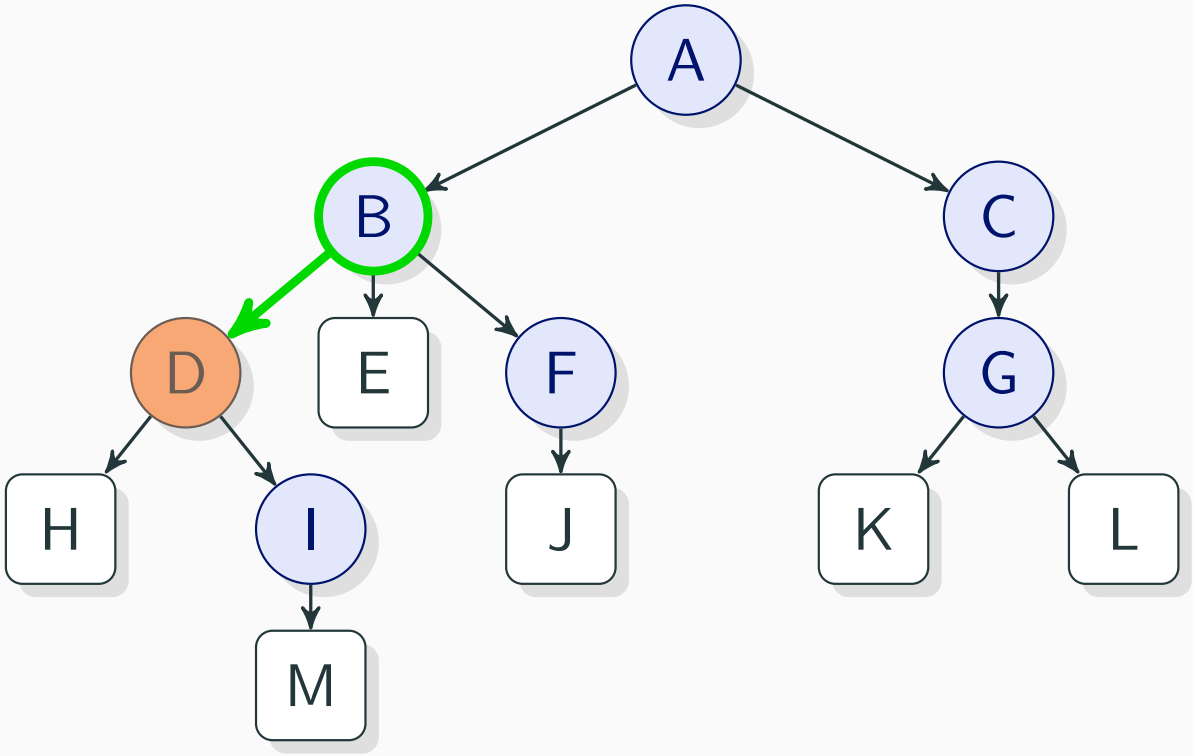
Les arbres

Une partie du vocabulaire est emprunté à la généalogie



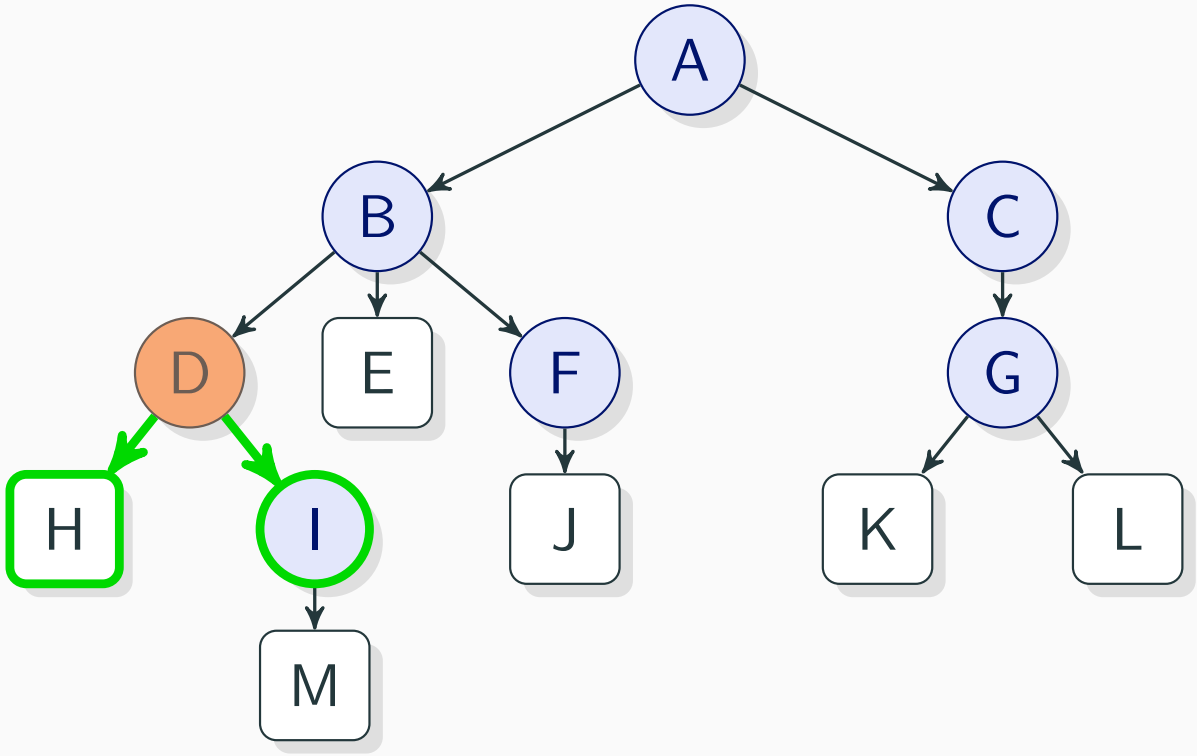
Les arbres

Une partie du vocabulaire est emprunté à la généalogie



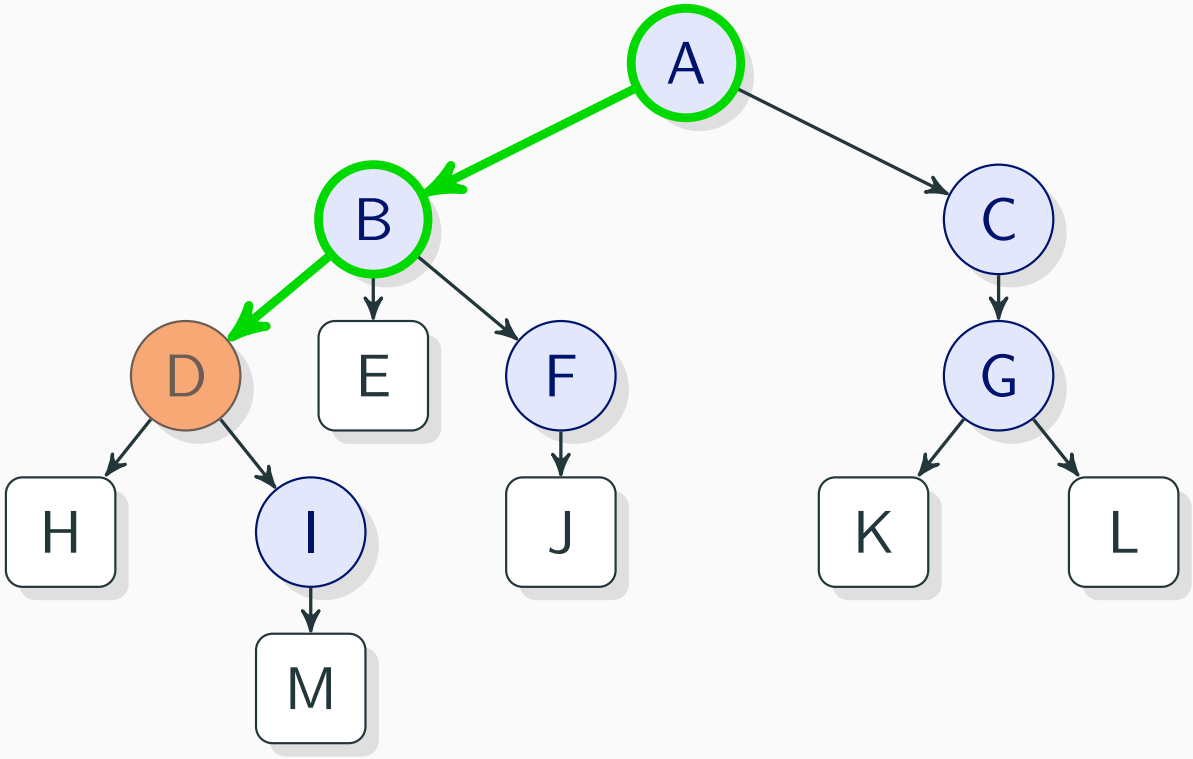
Les arbres

Une partie du vocabulaire est emprunté à la généalogie



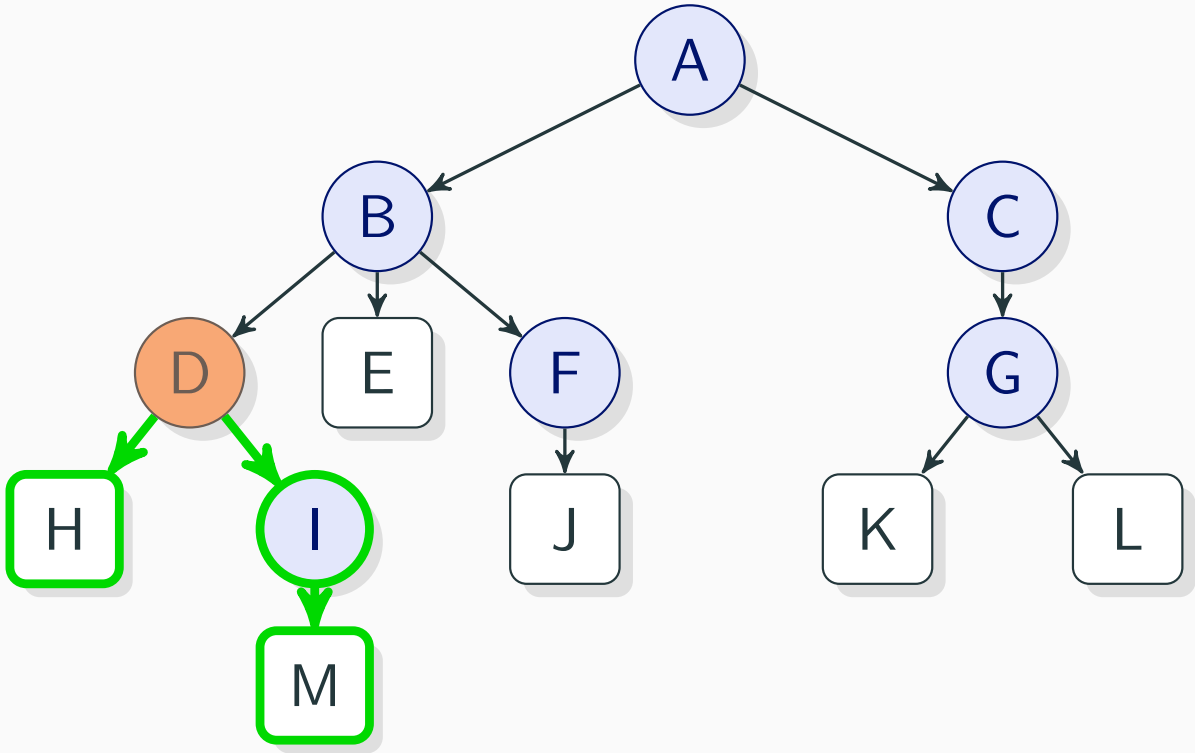
Les arbres

Une partie du vocabulaire est emprunté à la généalogie



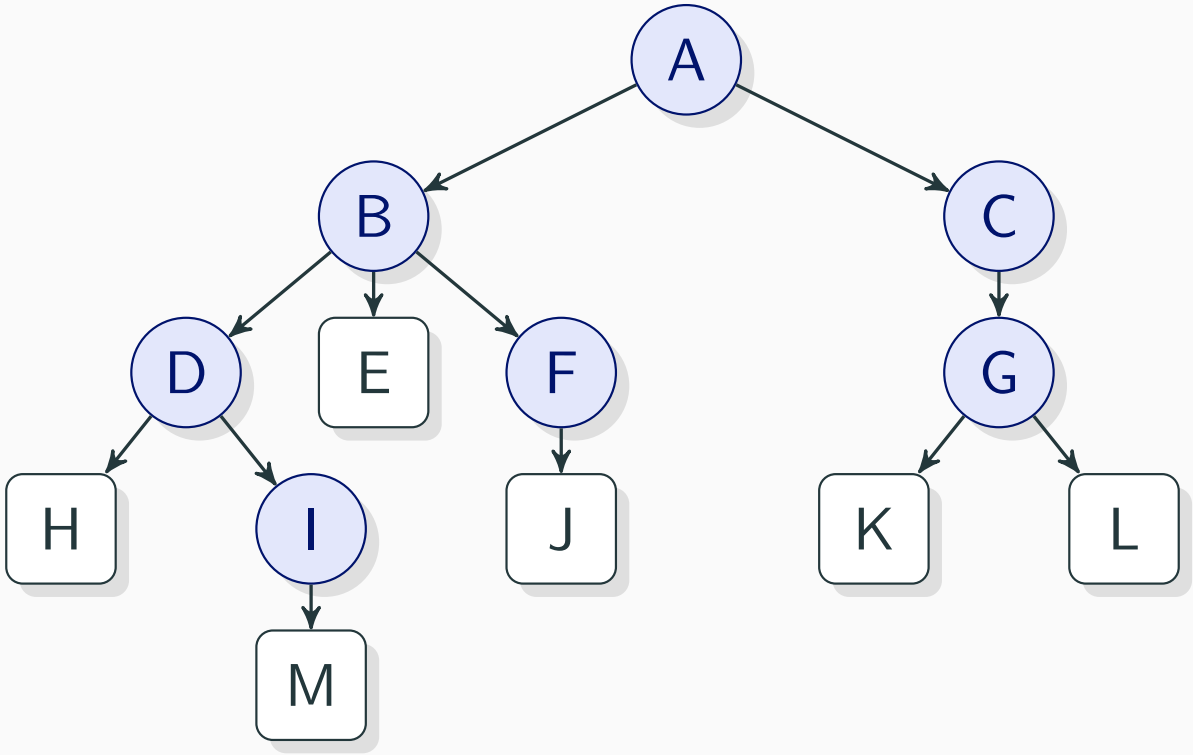
Les arbres

Une partie du vocabulaire est emprunté à la généalogie



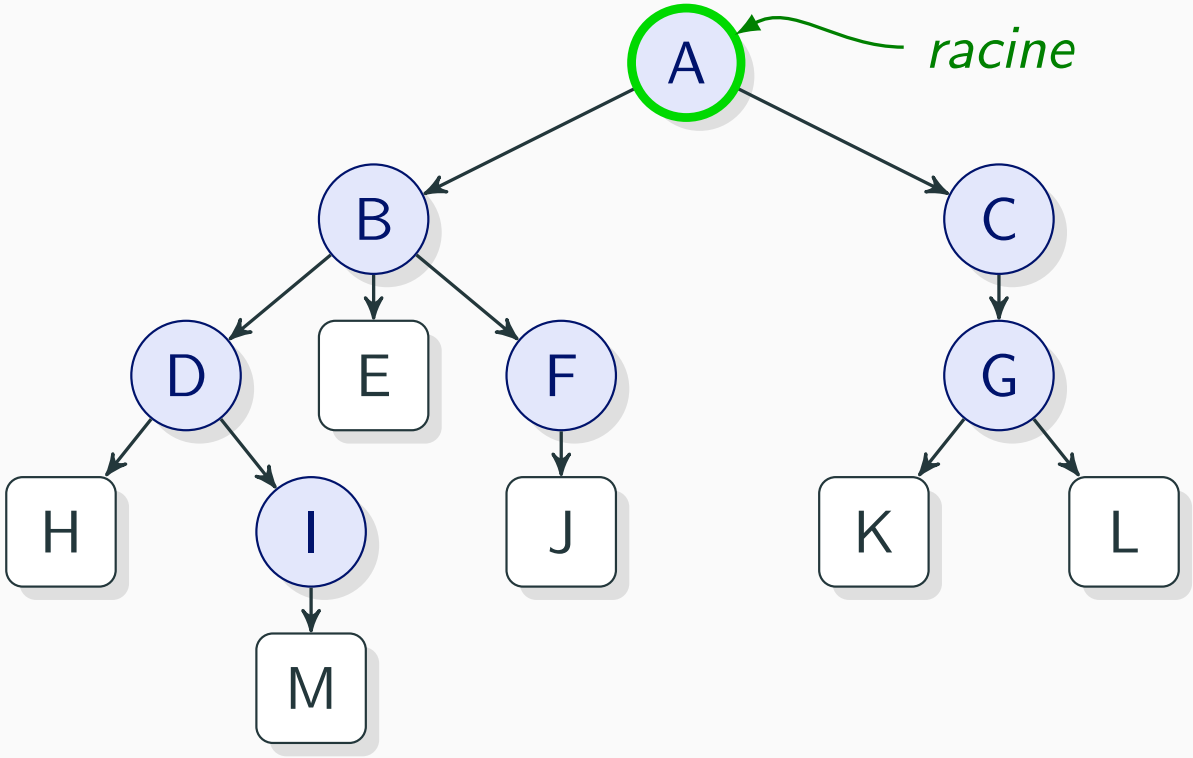
Les arbres

On utilise aussi la sémantique de la botanique



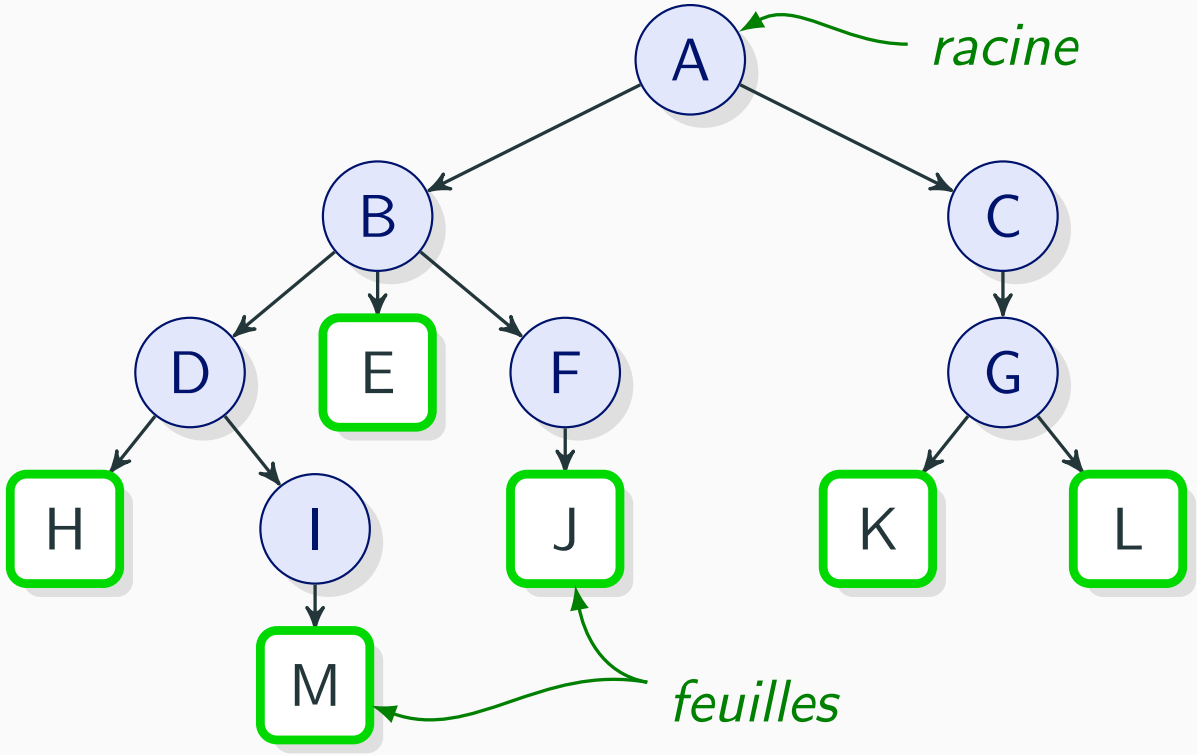
Les arbres

On utilise aussi la sémantique de la botanique



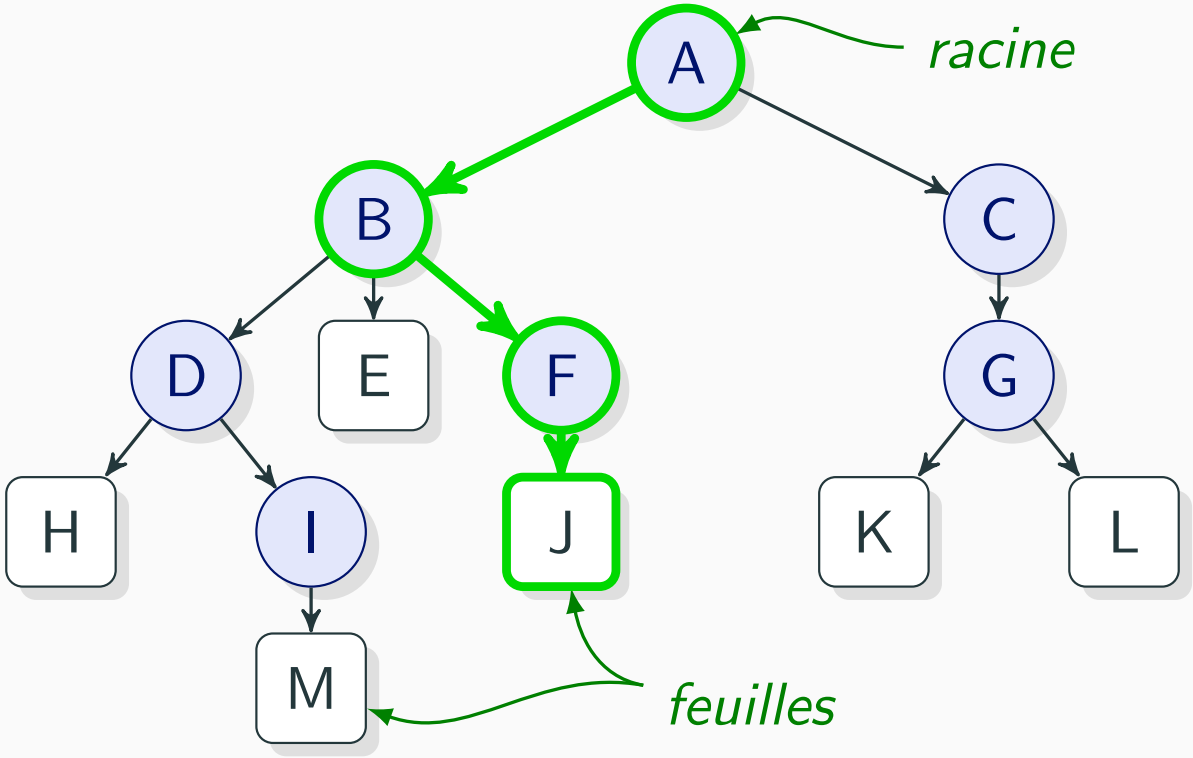
Les arbres

On utilise aussi la sémantique de la botanique



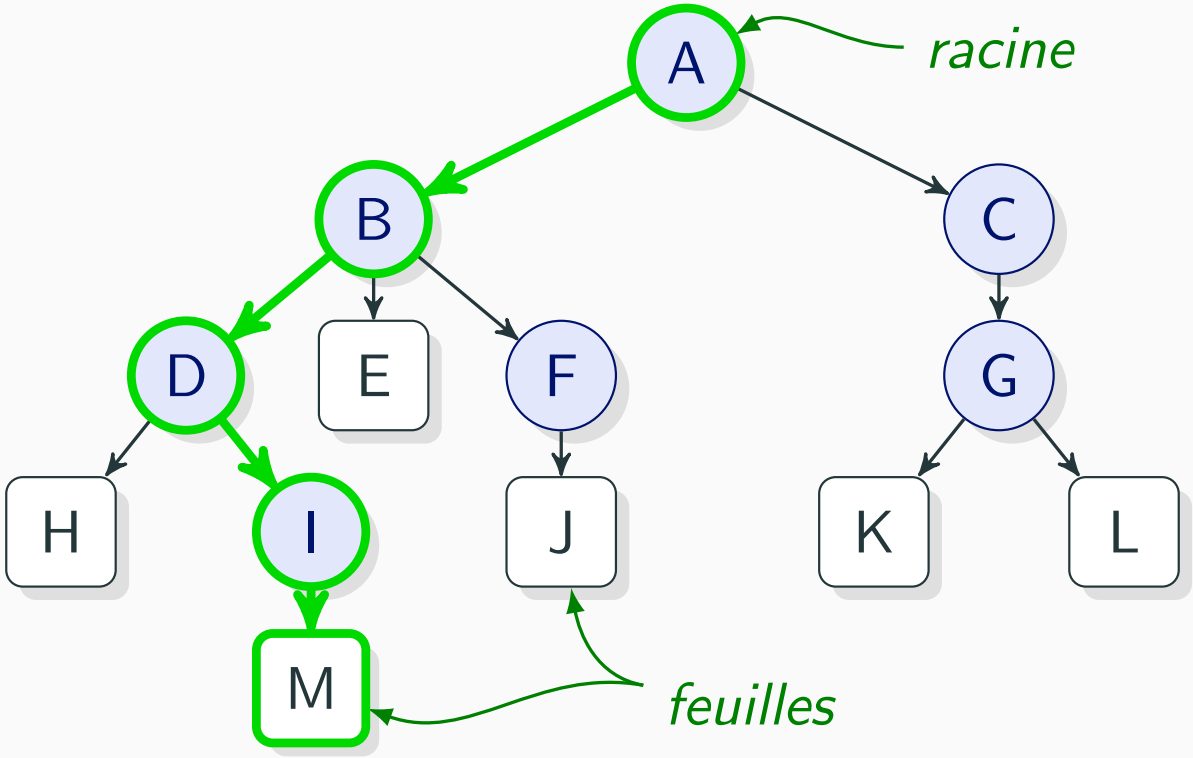
Les arbres

On utilise aussi la sémantique de la botanique



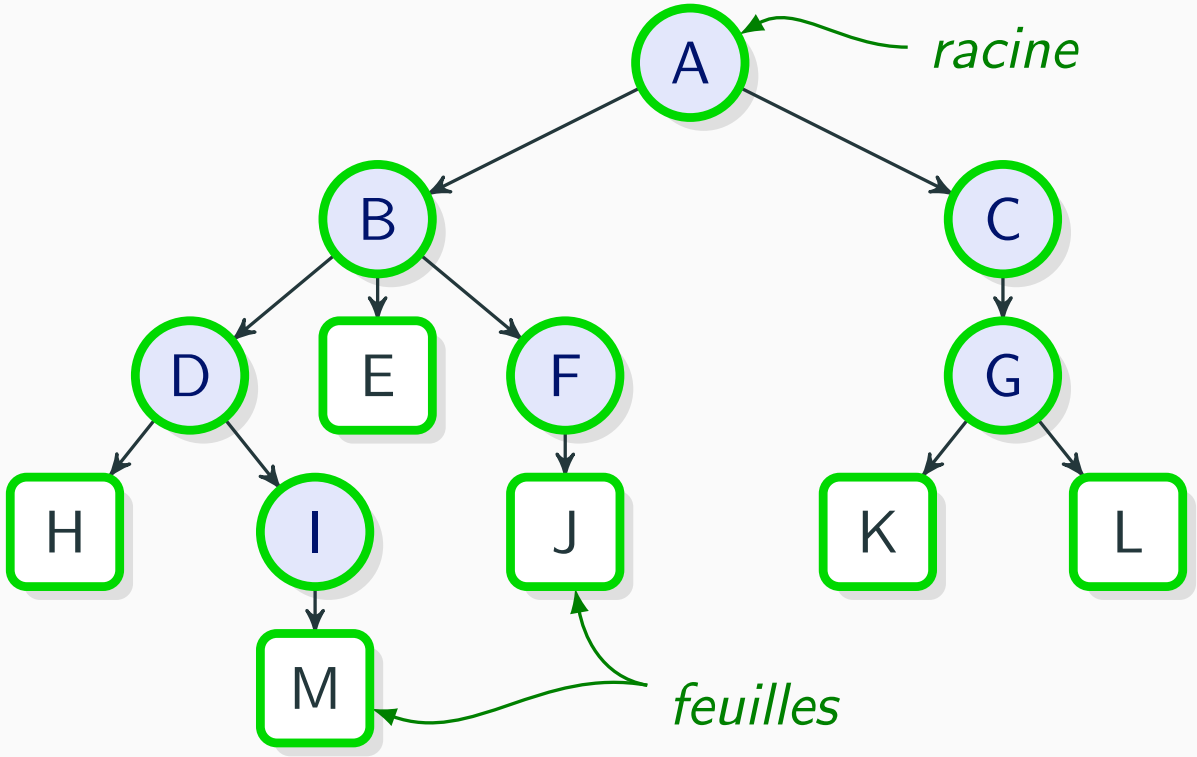
Les arbres

On utilise aussi la sémantique de la botanique



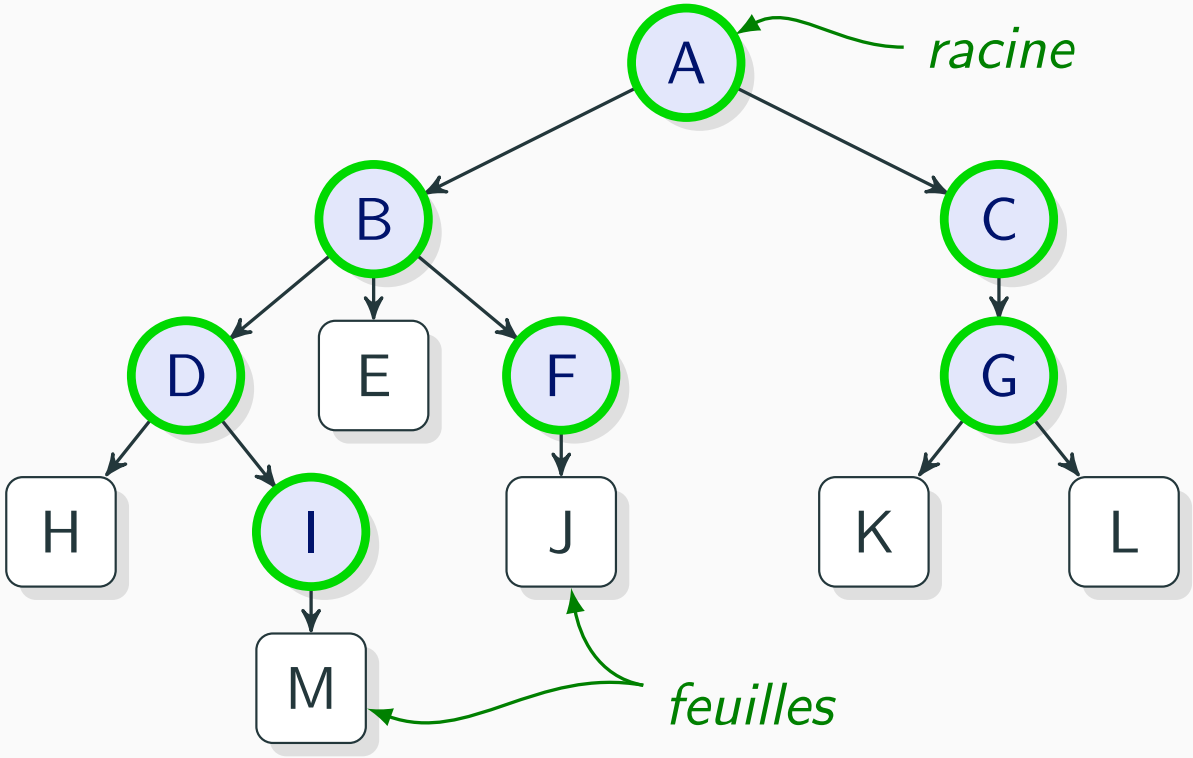
Les arbres

On utilise aussi la sémantique de la botanique



Les arbres

On utilise aussi la sémantique de la botanique



Les arbres

La descendance se fait toujours **de la racine vers les feuilles**

Les arbres

La descendance se fait toujours **de la racine vers les feuilles**

La racine est l'*unique* nœud sans parent

Les arbres

La descendance se fait toujours **de la racine vers les feuilles**

La racine est l'*unique* nœud sans parent

Les feuilles sont les nœuds sans enfants

Les arbres

La descendance se fait toujours **de la racine vers les feuilles**

- La racine est l'*unique* nœud sans parent
- Les feuilles sont les nœuds sans enfants

L'*arité* d'un nœud est son nombre d'enfants

Les arbres

La descendance se fait toujours **de la racine vers les feuilles**

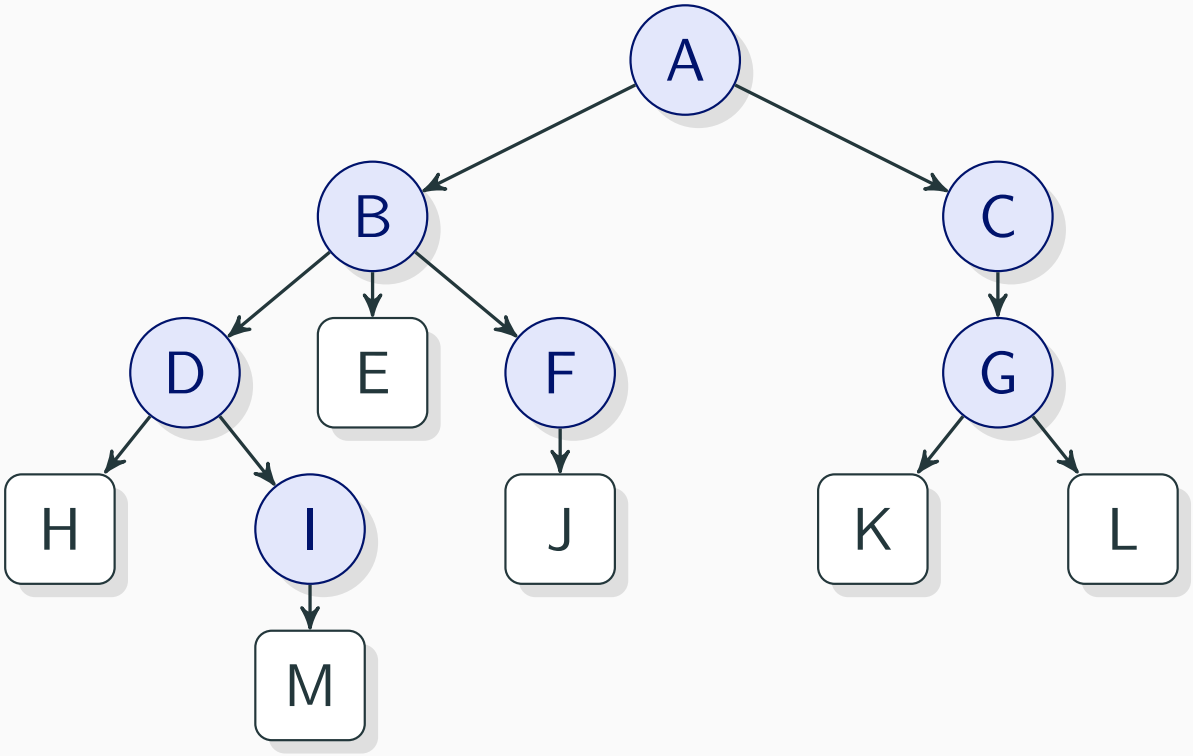
- La racine est l'*unique* nœud sans parent
- Les feuilles sont les nœuds sans enfants

L'*arité* d'un nœud est son nombre d'enfants

- Les feuilles sont les nœuds d'arité nulle

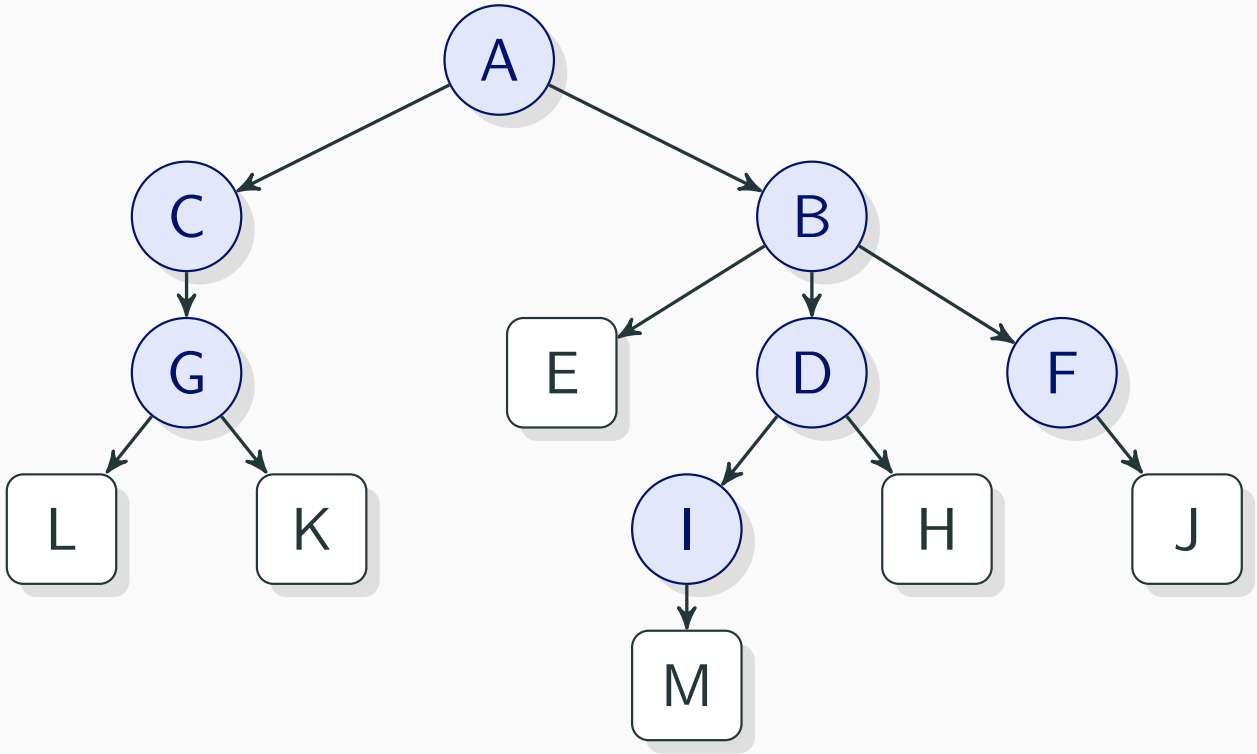
Les arbres

L'ordre des enfants peut, ou non, être significatif



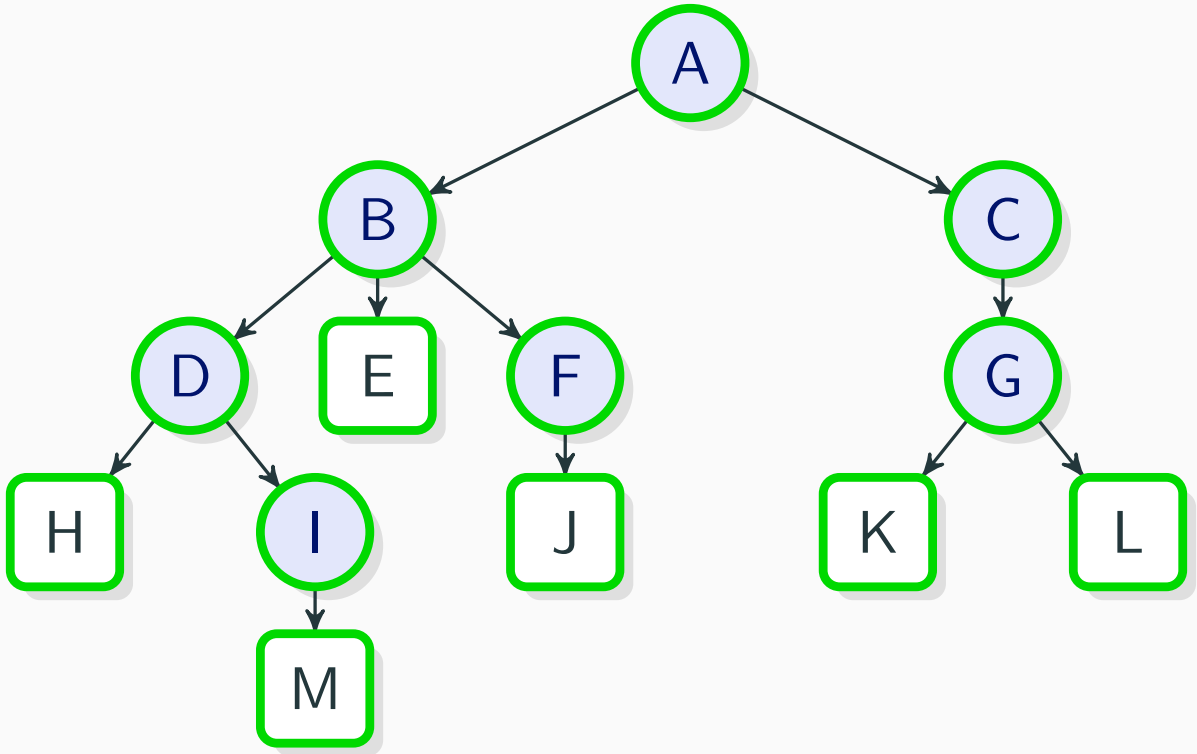
Les arbres

L'ordre des enfants peut, ou non, être significatif



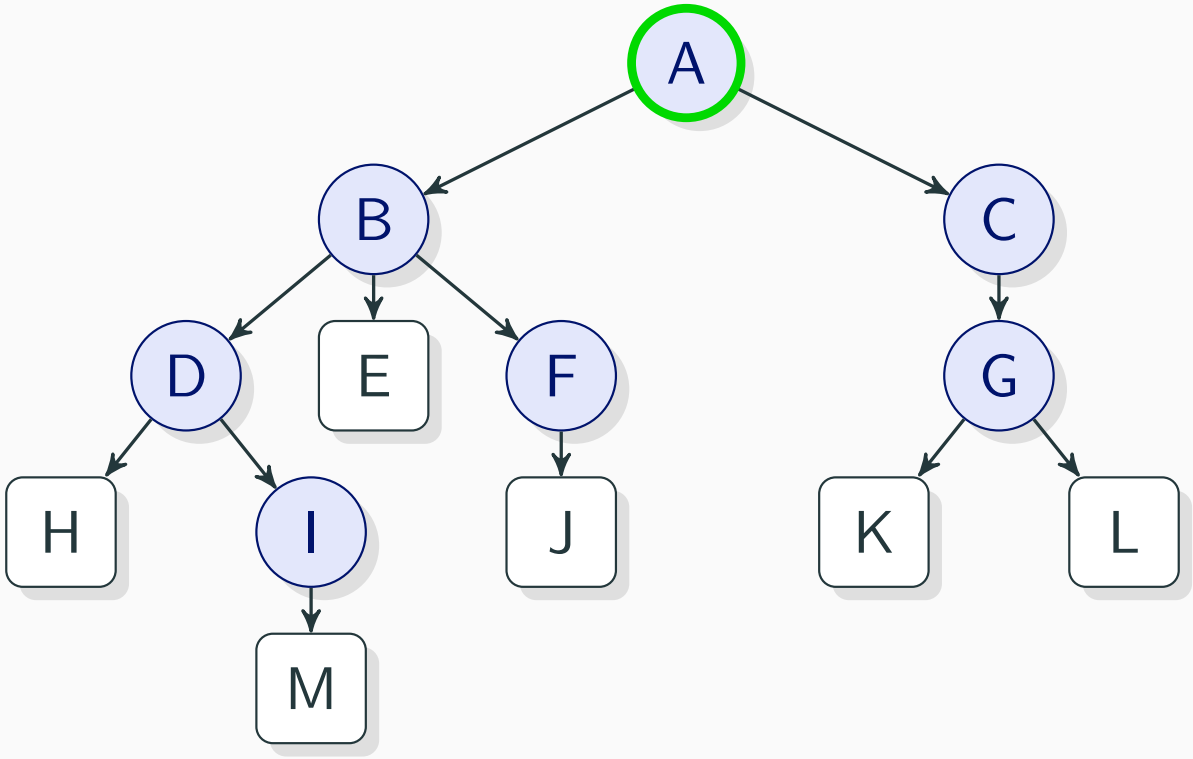
Vocabulaire des arbres

La *taille* d'un arbre est son nombre de nœuds :



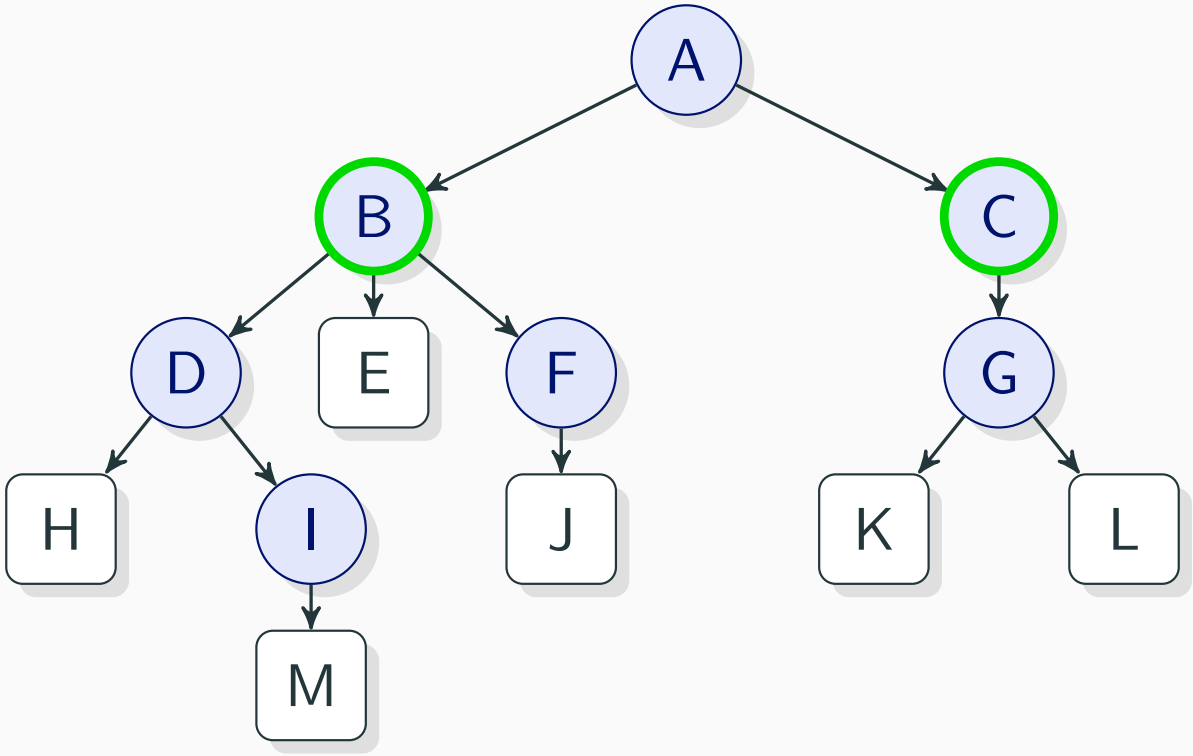
Vocabulaire des arbres

La *profondeur* d'un nœud est sa distance à la racine :



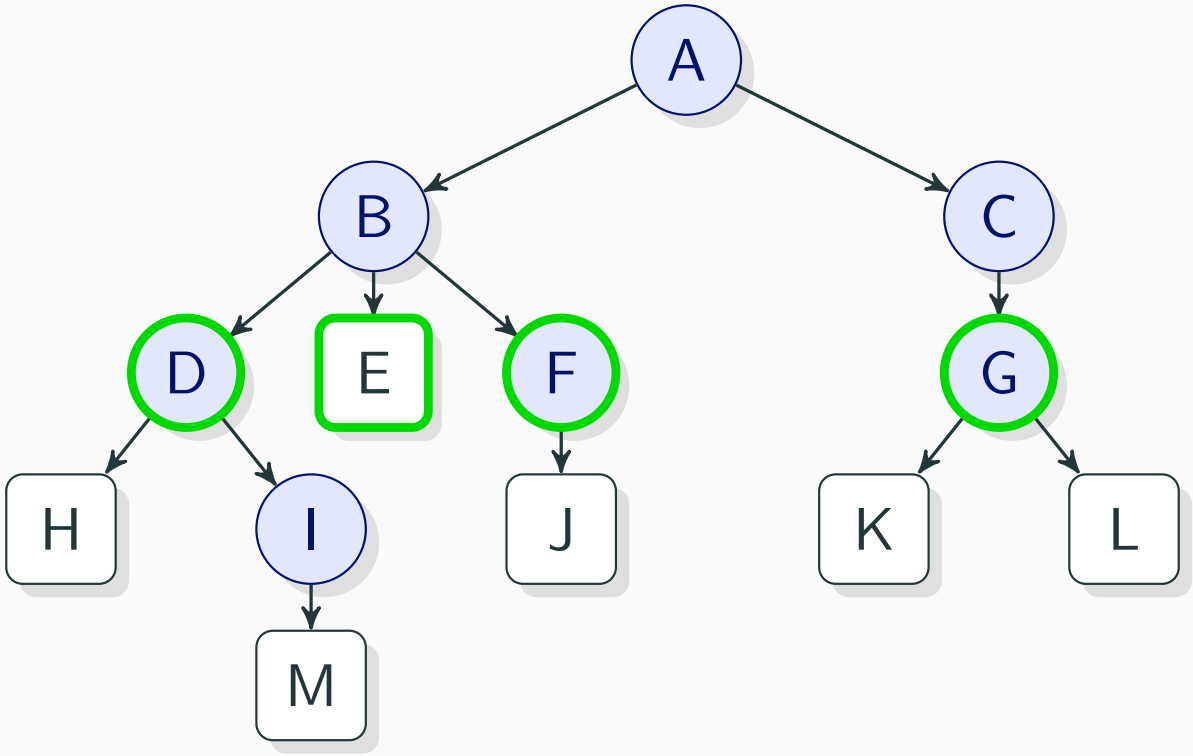
Vocabulaire des arbres

La *profondeur* d'un nœud est sa distance à la racine :



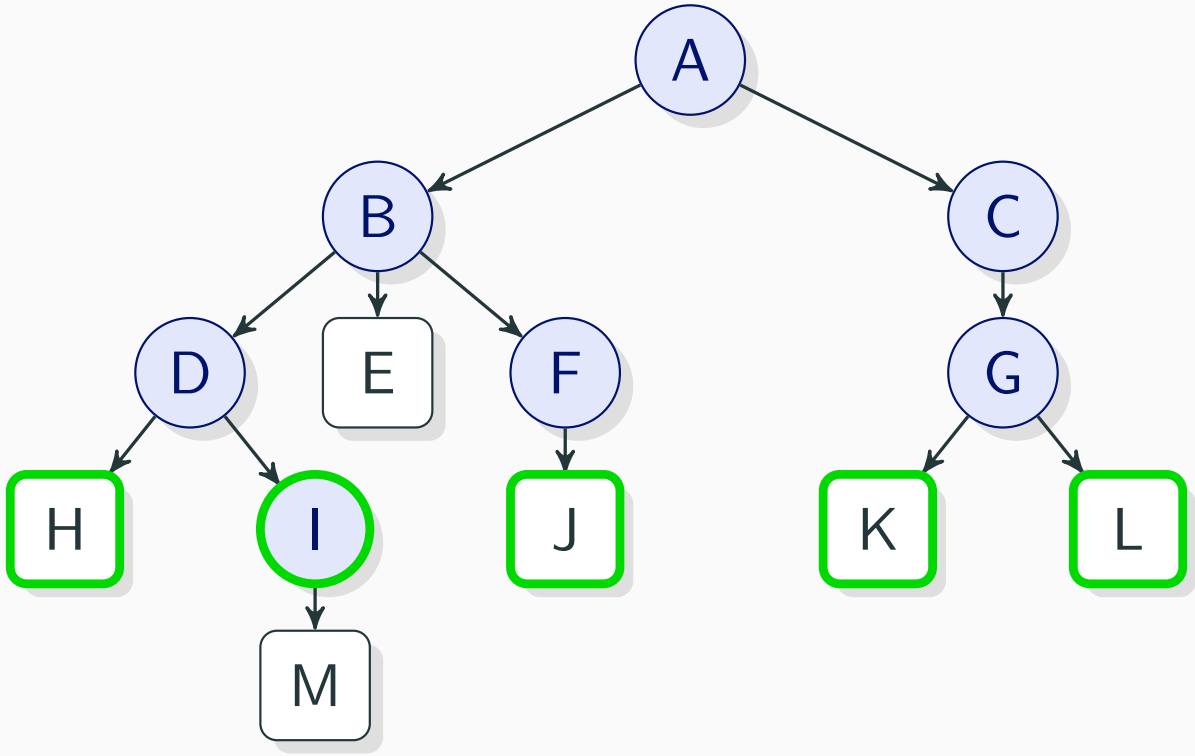
Vocabulaire des arbres

La *profondeur* d'un nœud est sa distance à la racine :



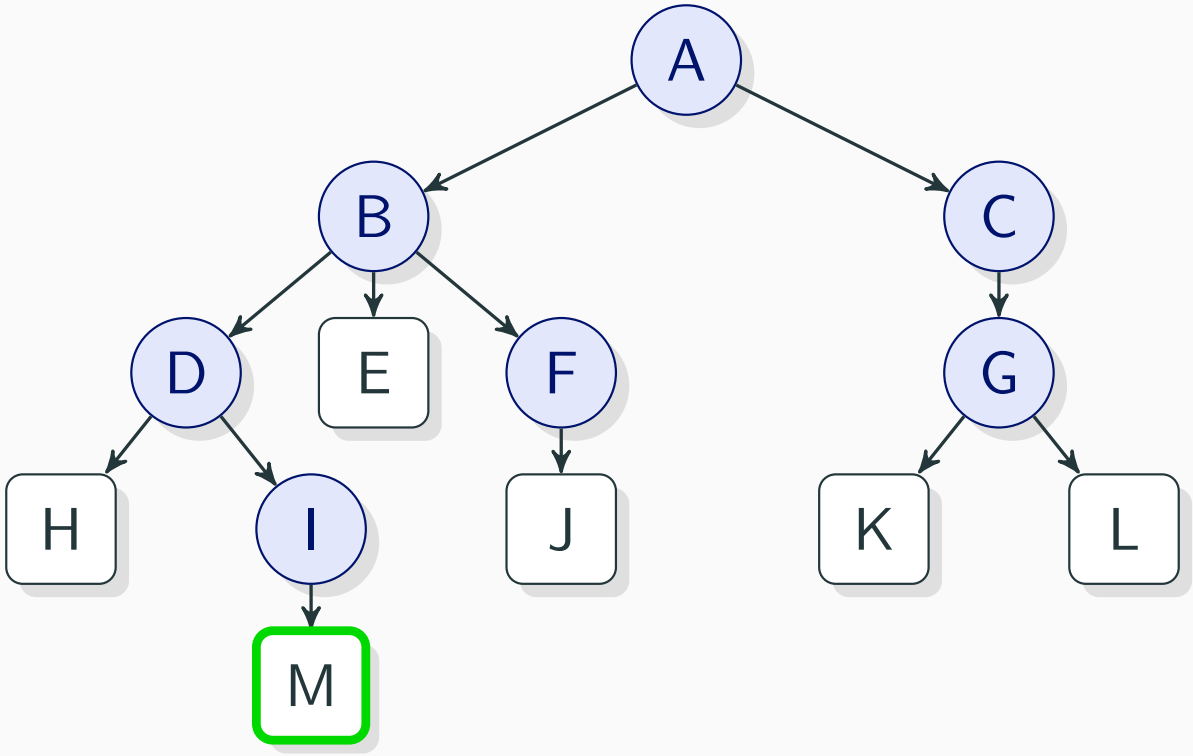
Vocabulaire des arbres

La *profondeur* d'un nœud est sa distance à la racine :



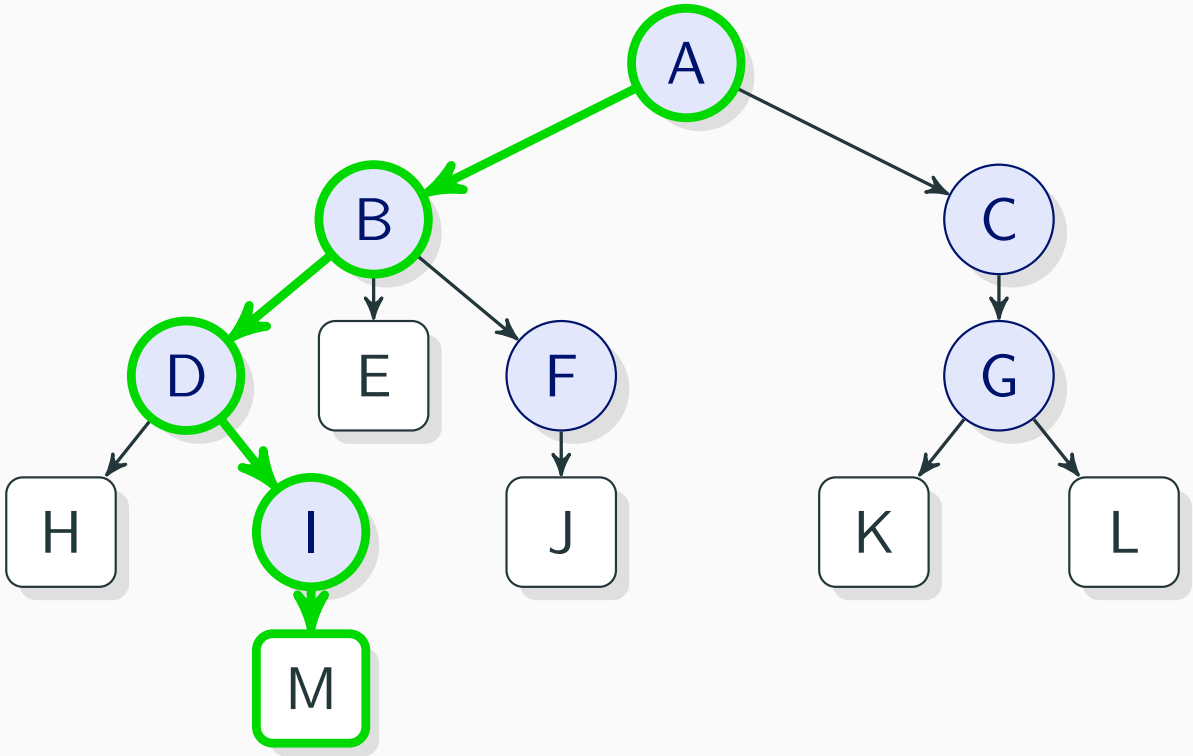
Vocabulaire des arbres

La *profondeur* d'un nœud est sa distance à la racine :



Vocabulaire des arbres

Sa *hauteur* est la profondeur maximale d'une feuille :



Vocabulaire des arbres

Le vocabulaire peut changer d'un ouvrage à l'autre

Attention à ne pas confondre taille et hauteur !

Nous dirons qu'un arbre est *parfait* si toutes les feuilles sont à la même profondeur

Premières propriétés

Théorème
Pour un arbre A *dont les nœuds ont une arité maximale* a*, on peut lier la taille* $|\mathrm{A}|$ *de l'arbre à sa hauteur* $\mathrm{h}(\mathrm{A})$ *par la relation :*

$$\mathrm{h}(\mathrm{A}) + 1 \leqslant |\mathrm{A}| \leqslant \frac{a^{\mathrm{h}(\mathrm{A})+1} - 1}{a - 1}$$

De façon similaire, on peut proposer un encadrement de sa hauteur :

$$\left\lfloor \log_a\big((a-1) \times |\mathrm{A}|\big) \right\rfloor \leqslant \mathrm{h}(\mathrm{A}) \leqslant |\mathrm{A}| - 1$$

Démonstration.
On encadre le nombre de nœuds de profondeur p par 1 et a^p.

On a alors

$$\sum_{p=0}^{\mathrm{h(A)}} 1 \leqslant |\mathrm{A}| \leqslant \sum_{p=0}^{\mathrm{h(A)}} a^p$$

ce qui conduit au premier encadrement.

Le second encadrement s'obtient en isolant $\mathrm{h(A)}$ dans les inégalités précédentes, en particulier, pour l'inégalité de gauche :

$$|\mathrm{A}| \leqslant \frac{a^{\mathrm{h(A)}+1} - 1}{a-1}$$

$$(a-1) \times |\mathrm{A}| + 1 \leqslant a^{\mathrm{h(A)}+1}$$

$$\log_a\big((a-1) \times |\mathrm{A}| + 1\big) \leqslant \mathrm{h(A)} + 1$$

$$\log_a\big((a-1) \times |\mathrm{A}|\big) < \mathrm{h(A)} + 1$$

$$\big\lfloor \log_a\big((a-1) \times |\mathrm{A}|\big) \big\rfloor < \mathrm{h(A)} + 1$$

Représentation en OCaml

Il n'existe aucun type « tout prêt »

Représentation en OCaml

Il n'existe aucun type « tout prêt »

Pour les listes :

```
type 'a chained_list =
  | Nil
  | Cell of { element : 'a; next : 'a chained_list }
```

Représentation en OCaml

Il n'existe aucun type « tout prêt »

Pour les listes :

```
type 'a chained_list =
  | Nil
  | Cell of { element : 'a; next : 'a chained_list }
```


Pour les arbres :

```
type 'a tree =
   { element : 'a; children : 'a tree list }
```

Représentation en OCaml

Il n'existe aucun type « tout prêt »

Pour les listes :

```
type 'a chained_list =
  | Nil
  | Cell of { element : 'a; next : 'a chained_list }
```

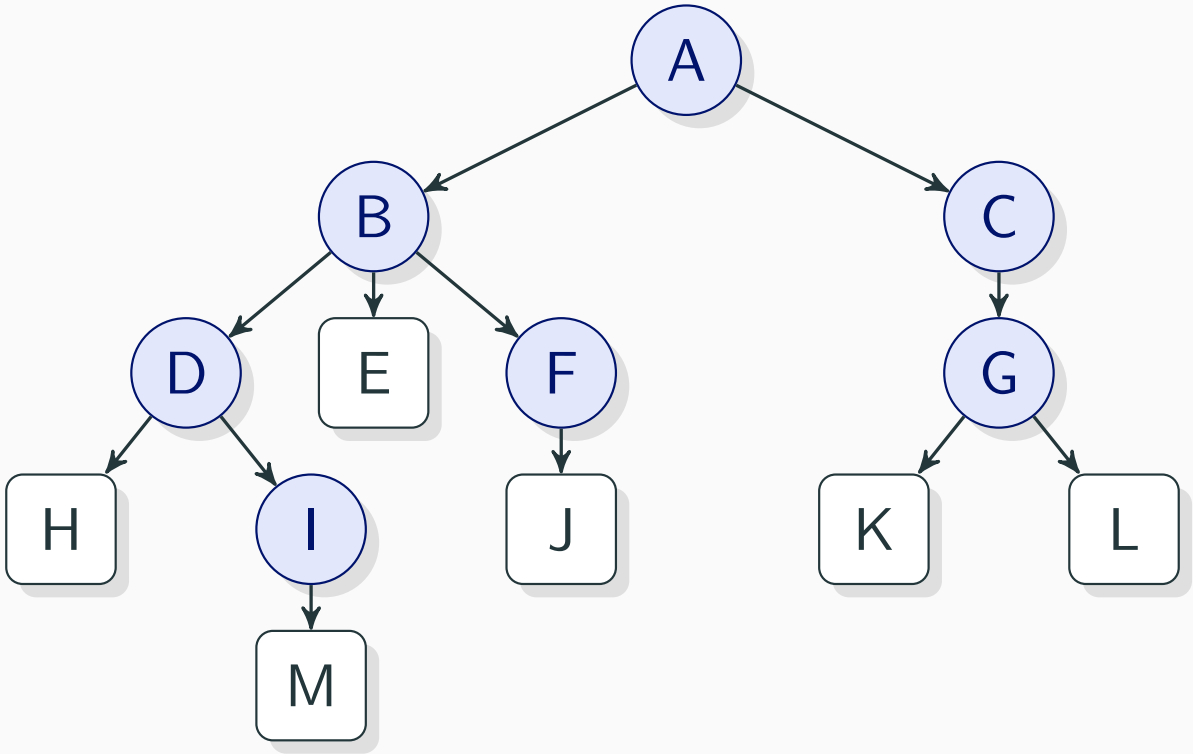
Pour les arbres :

```
type 'a tree =
   { element : 'a; children : 'a tree list }
```

Pas besoin de `Nil` ! Il suffit d'avoir `children = []`

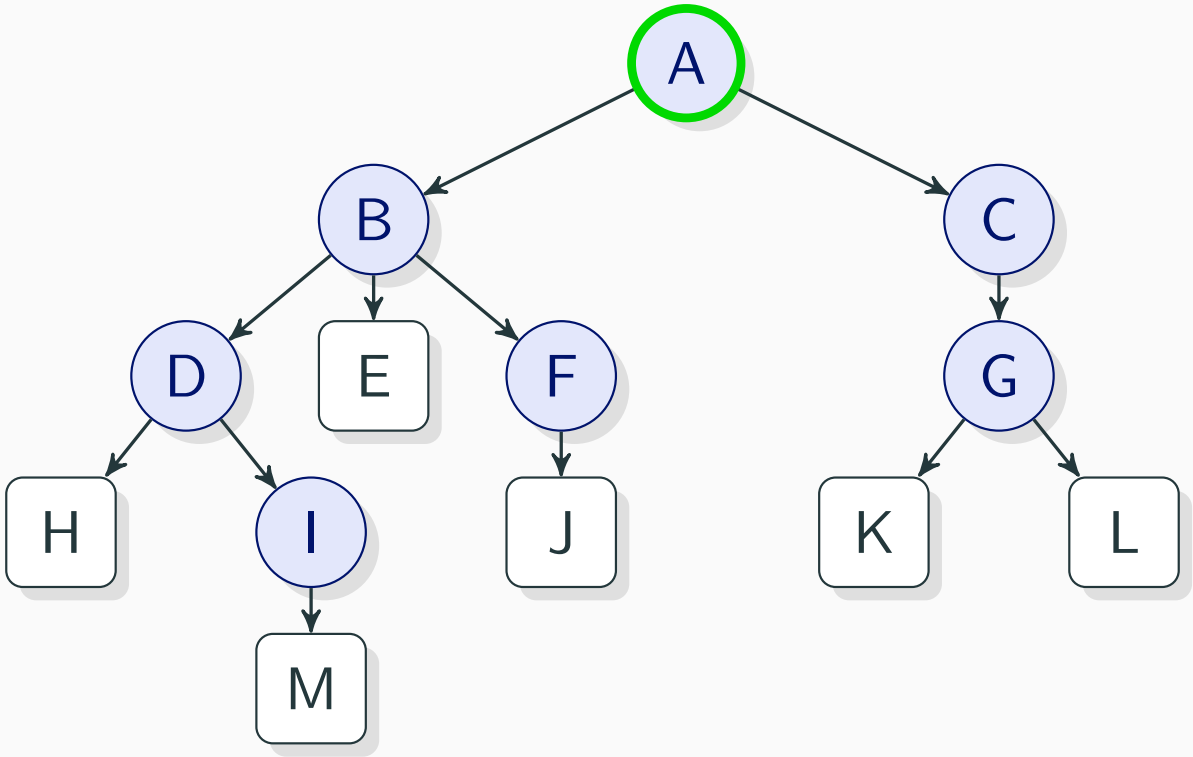
Représentation en OCaml

Un arbre est constitué d'une racine et de sous-arbres :



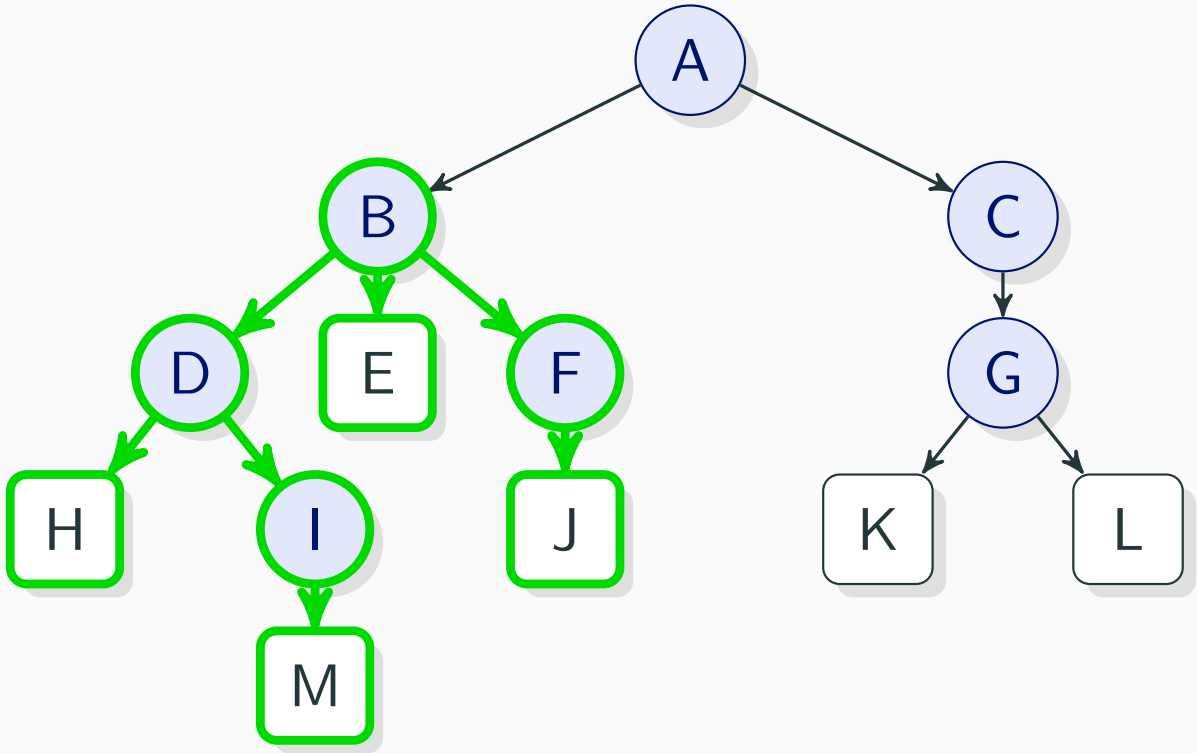
Représentation en OCaml

Un arbre est constitué d'une racine et de sous-arbres :



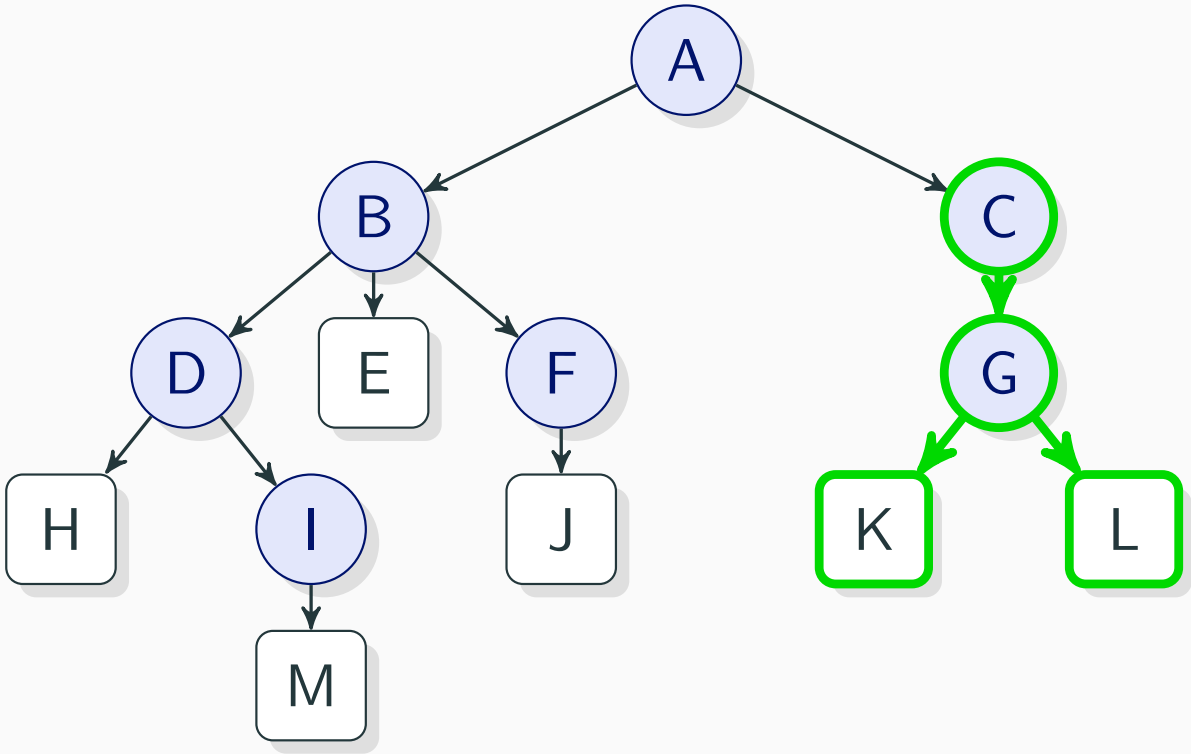
Représentation en OCaml

Un arbre est constitué d'une racine et de sous-arbres :



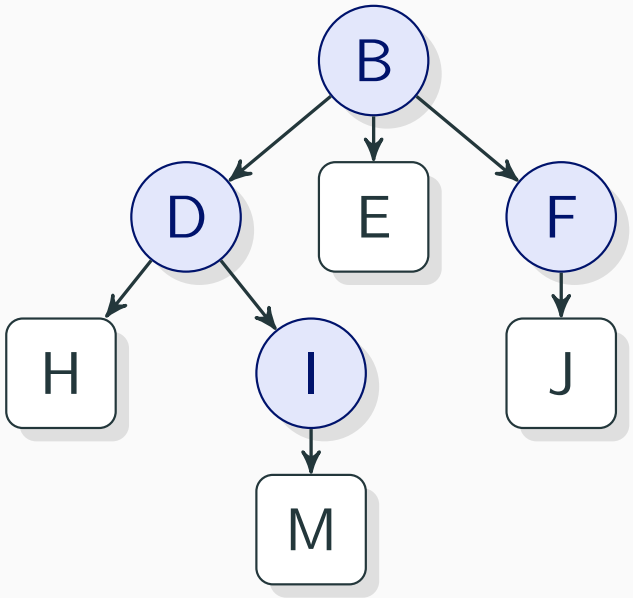
Représentation en OCaml

Un arbre est constitué d'une racine et de sous-arbres :



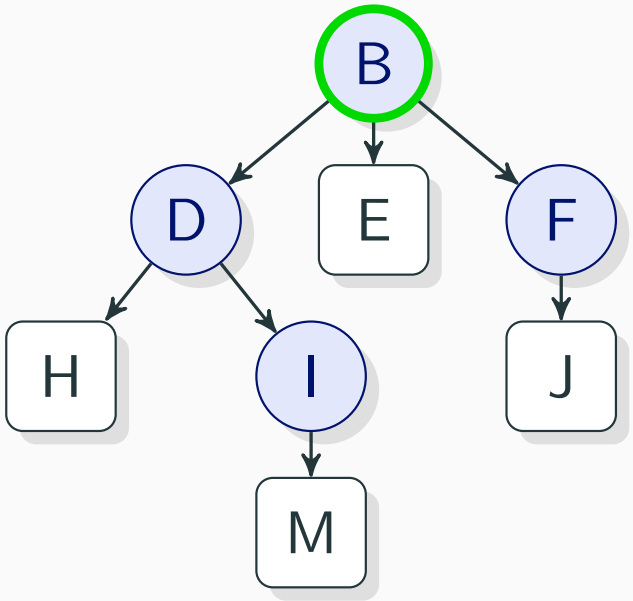
Représentation en OCaml

Un arbre est constitué d'une racine et de sous-arbres :



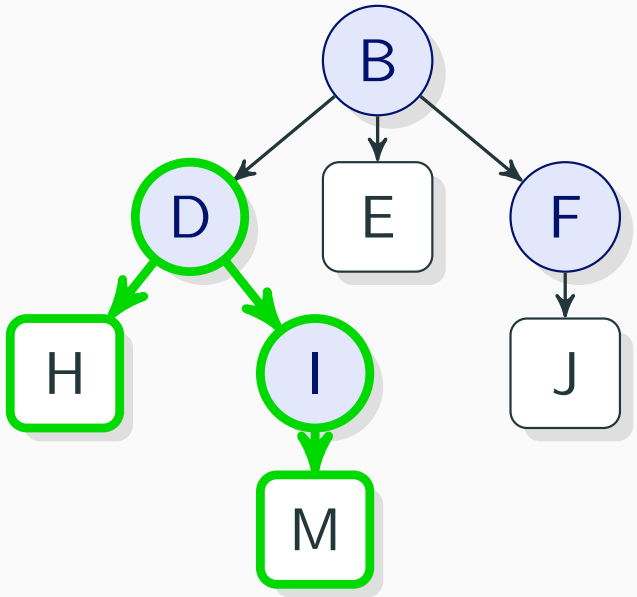
Représentation en OCaml

Un arbre est constitué d'une racine et de sous-arbres :



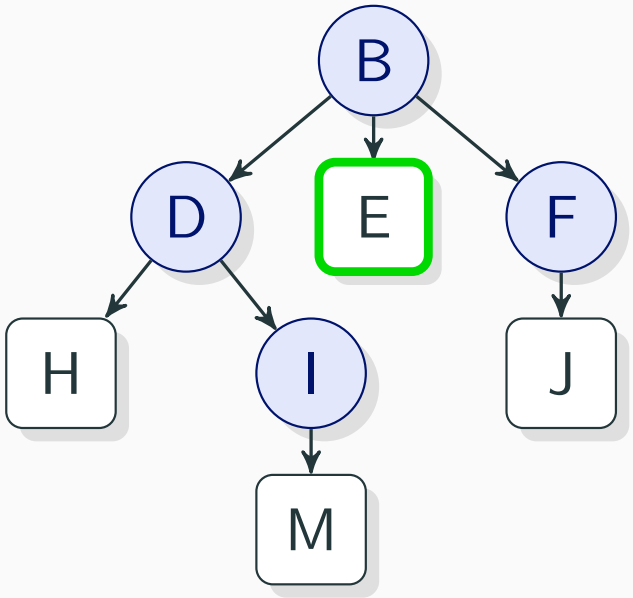
Représentation en OCaml

Un arbre est constitué d'une racine et de sous-arbres :



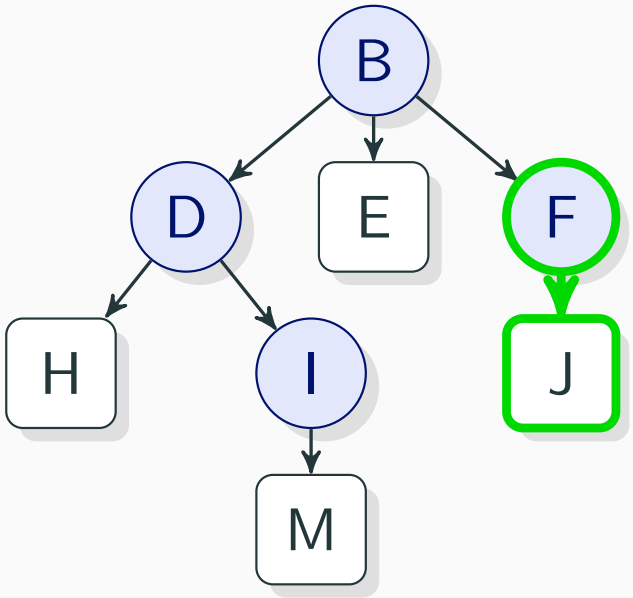
Représentation en OCaml

Un arbre est constitué d'une racine et de sous-arbres :



Représentation en OCaml

Un arbre est constitué d'une racine et de sous-arbres :



Représentation en OCaml

Un arbre est constitué d'une racine et de sous-arbres :

Représentation en OCaml

Un arbre est constitué d'une racine et de sous-arbres :

Représentation en OCaml

Un arbre est constitué d'une racine et de sous-arbres :

aucun sous-arbre

Représentation en OCaml

Pour le type :

```
type 'a tree = { element : 'a; children : 'a tree list }
```


L'arbre est défini par

```
# let ex = { element="A"; children=[
                { element="B"; children=[
                    { element="D"; children=[
                        { element="H"; children=[] } ;
                        { element="I"; children=[
                            { element="M"; children=[] } ] } ] } ;
                    { element="E"; children=[] };
                    { element="F"; children=[
                        { element="J"; children=[] } ] } ] } ;
                { element="C"; children=[
                    { element="G"; children=[
                        { element="K"; children=[] } ;
                        { element="L"; children=[] } ] } ] } ] };;
```

Représentation en OCaml

Pour le type :

```
type 'a tree = Node of 'a * 'a tree list
```


L'arbre est défini par

```
# let ex = Node("A", [
             Node("B", [
               Node("D", [
                 Node("H", [] ) ;
                 Node("I", [
                   Node("M", [] ) ] ) ] ) ;
               Node("E", [] ) ;
               Node("F", [
                 Node("J", [] ) ] ) ] ) ;
             Node("C", [
               Node("G", [
                 Node("K", [] ) ;
                 Node("L", [] ) ] ) ] ) ];;
```

Calcul de la taille

Le calcul de la taille se fait récursivement

Pour le type

```
type 'a tree = Node of 'a * 'a tree list
```

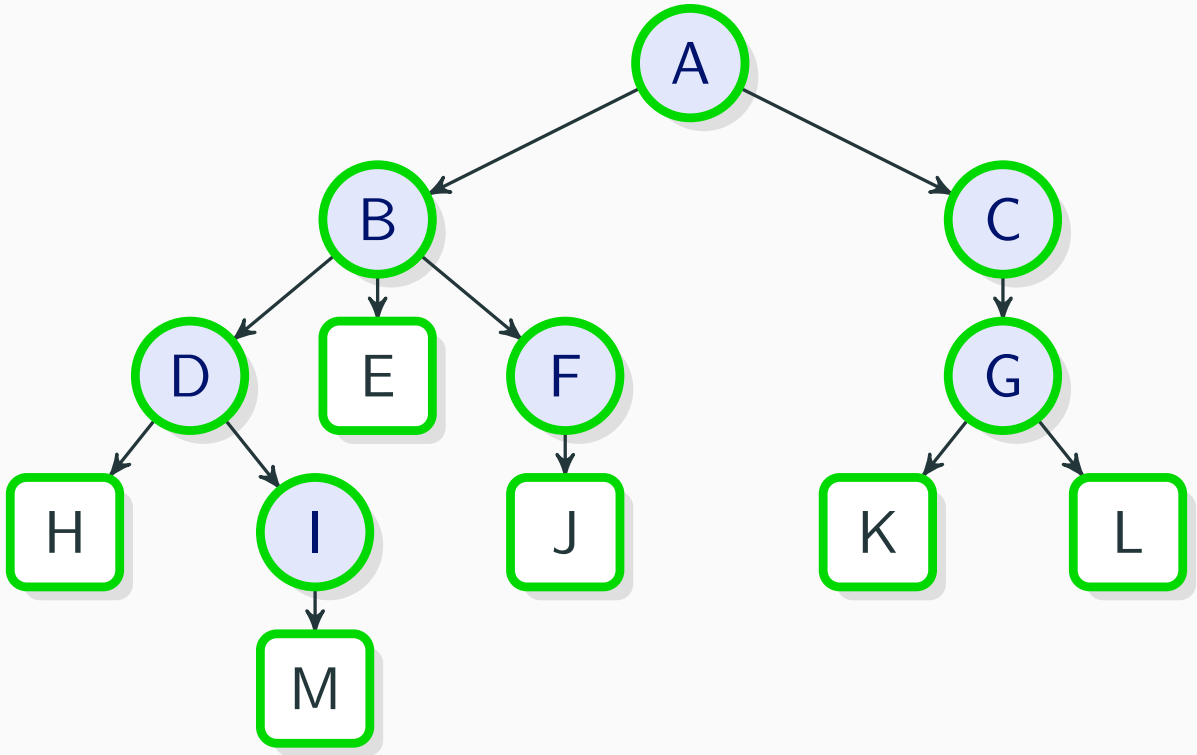

On écrira :

```
# let rec size = function
    Node (_, lst)
      -> let sum_list = List.fold_left (+) 0
           in 1 + sum_list (List.map size lst);;

val size : 'a tree -> int = <fun>
```

Calcul de la taille

On peut se passer de `List.map` et `List.fold_left`

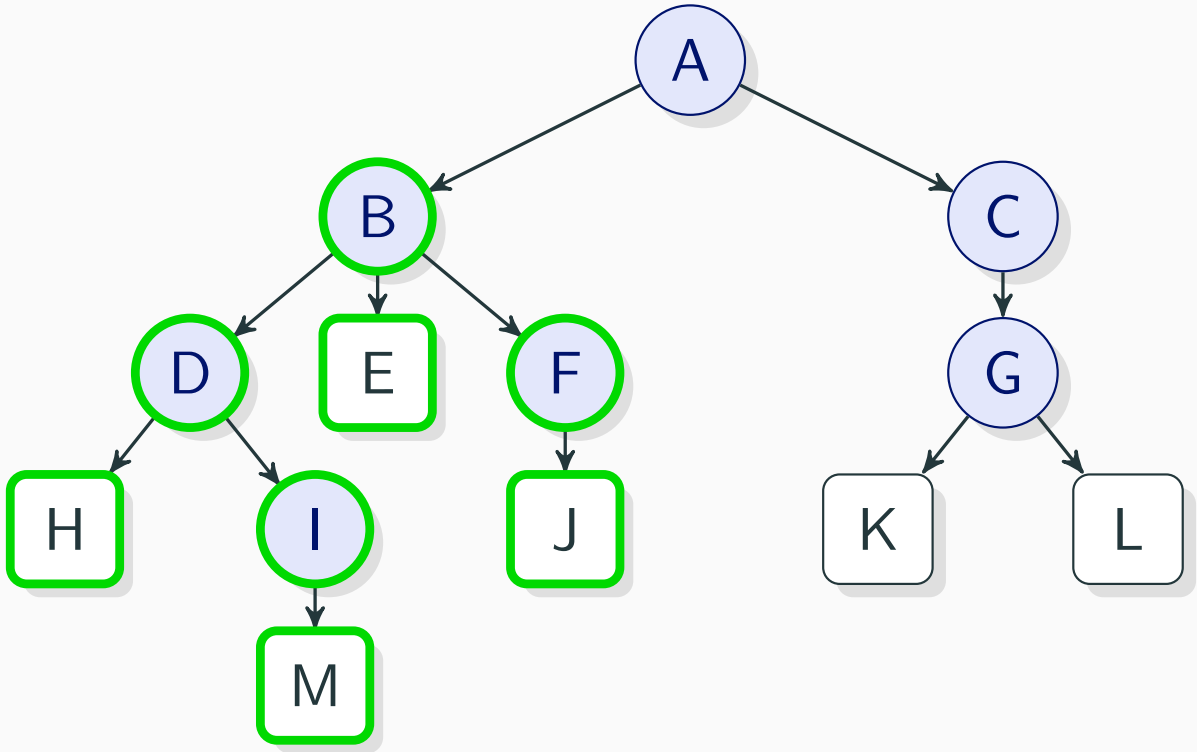
Pour cela, on « élague » les branches une à une :



Calcul de la taille

On peut se passer de `List.map` et `List.fold_left`

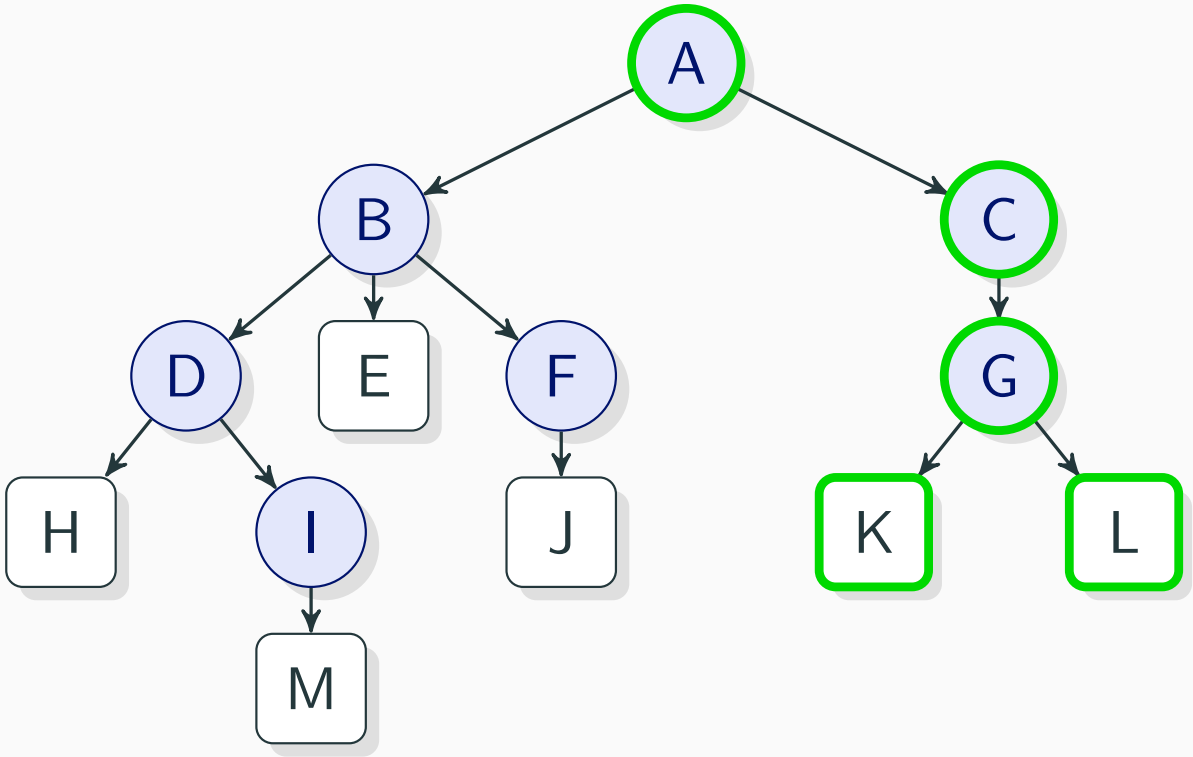
Pour cela, on « élague » les branches une à une :



Calcul de la taille

On peut se passer de `List.map` et `List.fold_left`

Pour cela, on « élague » les branches une à une :



Calcul de la taille

On filtre donc la liste des enfants :

```
# let rec size = function
    | Node (_, []) -> 1 (* feuille *)
    | Node (v, h::t)
      -> size h + size (Node (v, t));;

val size : 'a tree -> int = <fun>
```

Calcul de la hauteur

La hauteur se détermine également par récurrence


```
# let rec height = function
    | Node (_, []) -> 0
    | Node (_, lst)
        -> let max_list =
                  List.fold_left max (List.hd lst)
             in 1 + max_list (List.map height lst);;

val height : 'a tree -> int = <fun>
```

Calcul de la hauteur

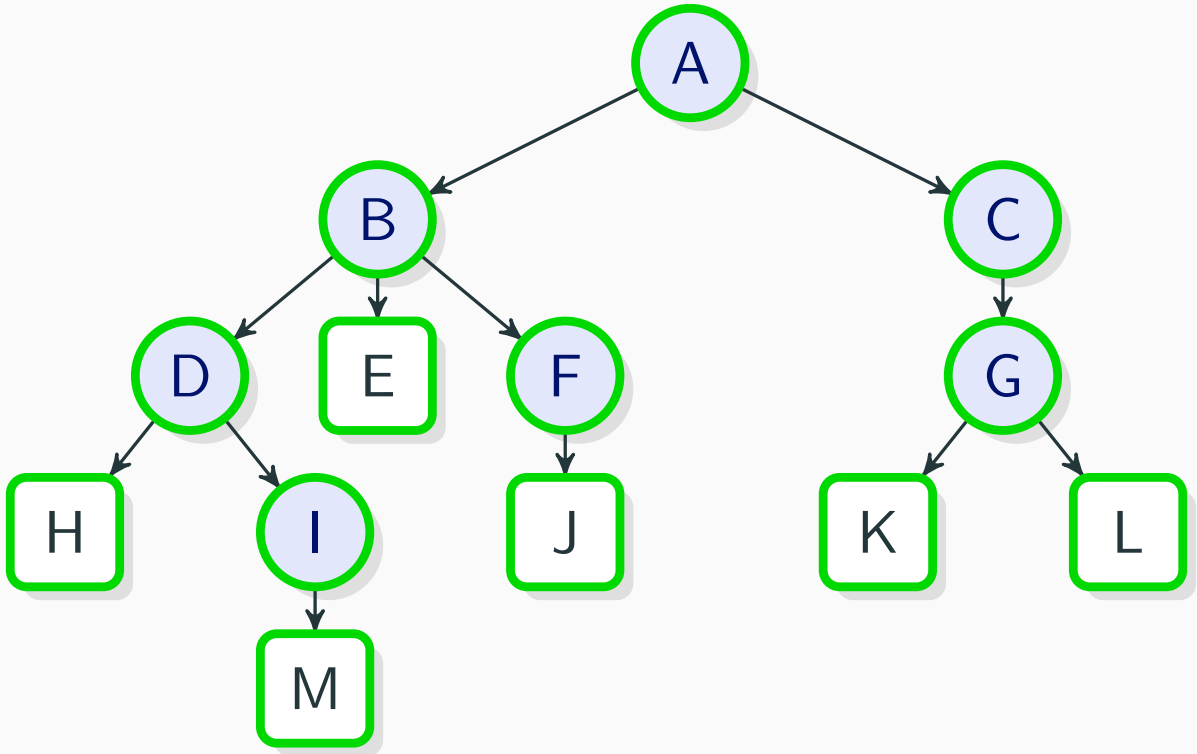
Ou, si l'on est astucieux

```
# let rec height = function
    Node (_, lst)
      -> 1 + List.fold_left max (-1)
                    (List.map height lst);;

val height : 'a tree -> int = <fun>
```

Dans ce cas, des explications sont nécessaires

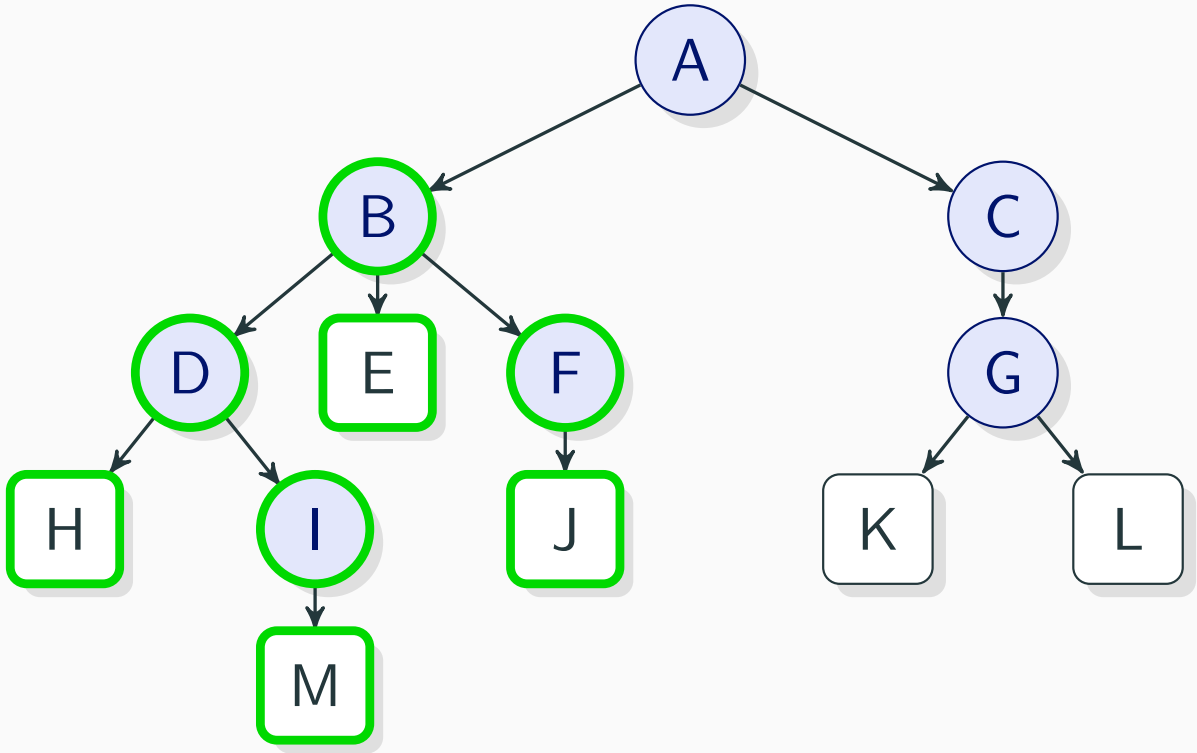
Calcul de la hauteur

Ou bien par élagage :



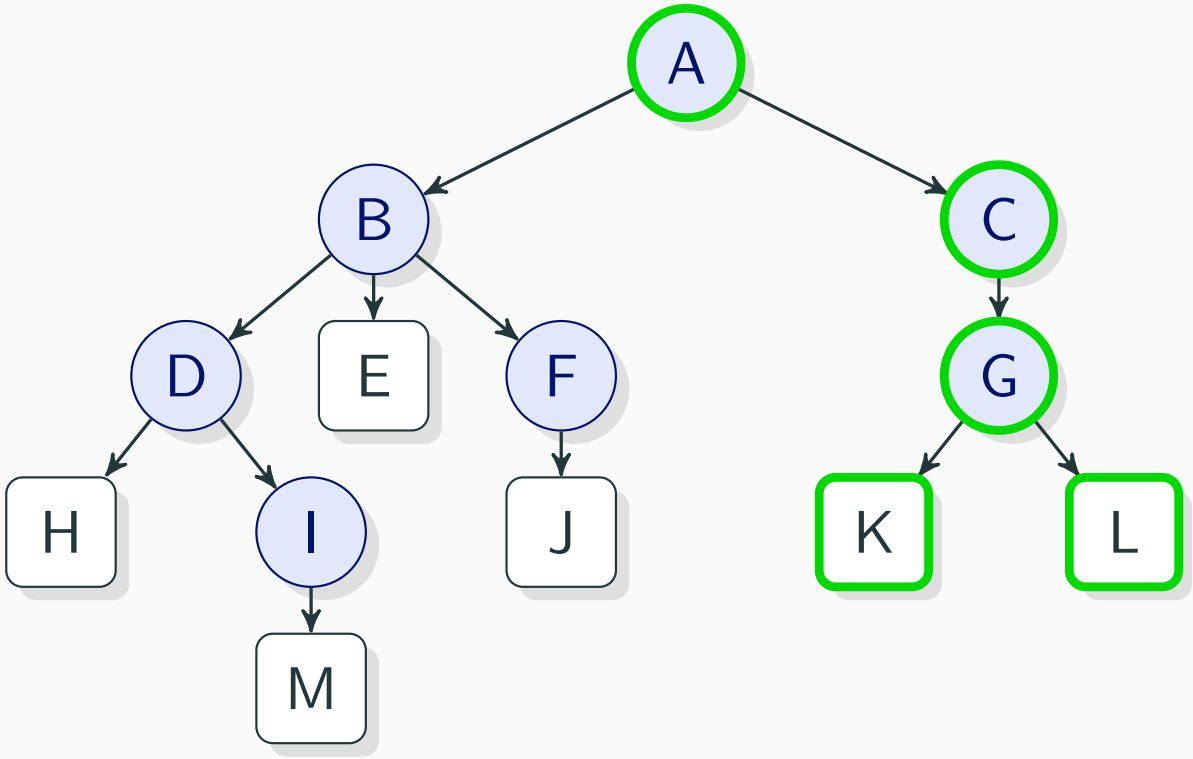
Calcul de la hauteur

Ou bien par élagage :



Calcul de la hauteur

Ou bien par élagage :



Calcul de la hauteur

Cela s'écrit :

```
# let rec height = function
    | Node (_, []) -> 0
    | Node (v, t::q) -> max (1 + height t)
                            (height (Node (v, q)));;

val height : 'a tree -> int = <fun>
```

Calcul du plus grand élément

Et pour trouver le plus grand élément :

```
# let rec largest = function
    Node (v, lst)
      -> List.fold_left max v
                  (List.map largest lst);;

val largest : 'a tree -> 'a = <fun>
```

Là encore, des explications sont indispensables

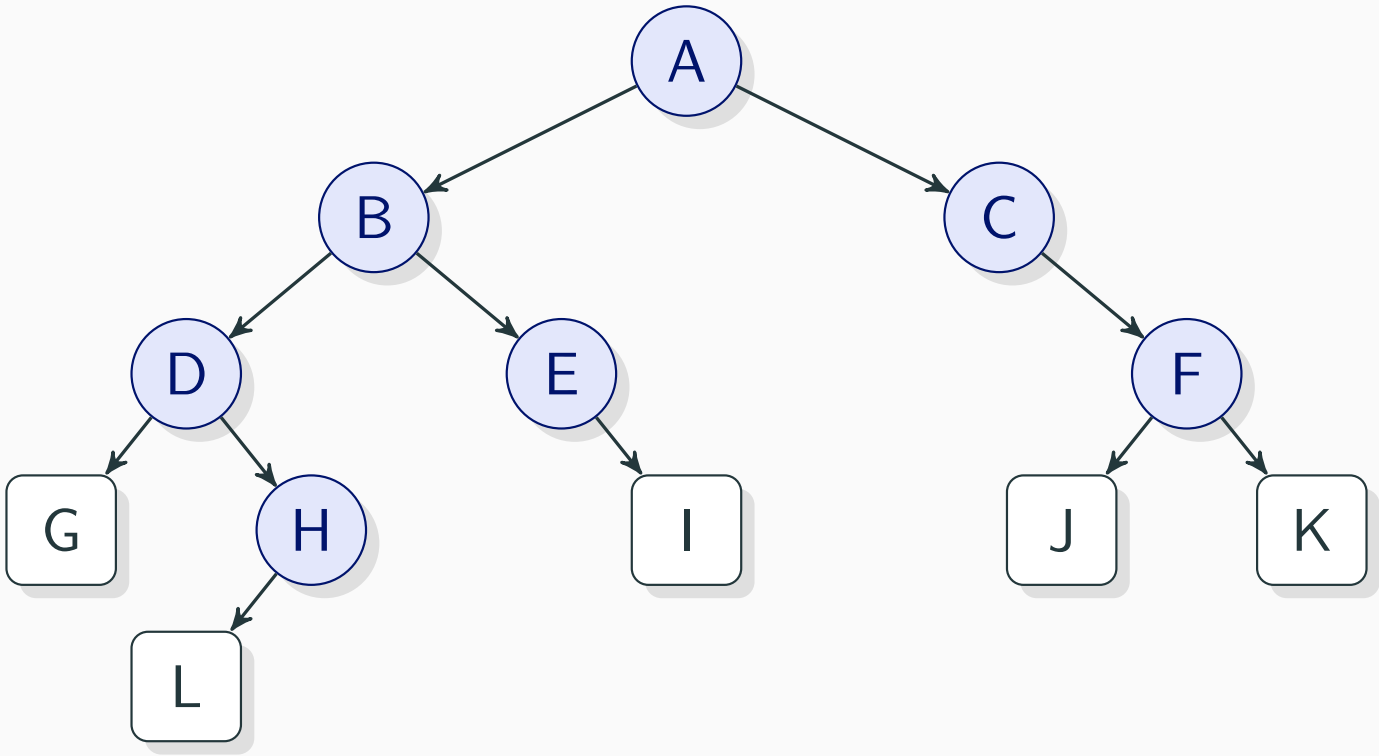
Calcul du plus grand élément

Ou bien par élagage :

```
# let rec largest = function
    | Node (v, []) -> v
    | Node (v, t::q) -> max (largest t)
                            (largest (Node (v,q)));;

val largest : 'a tree -> int = <fun>
```

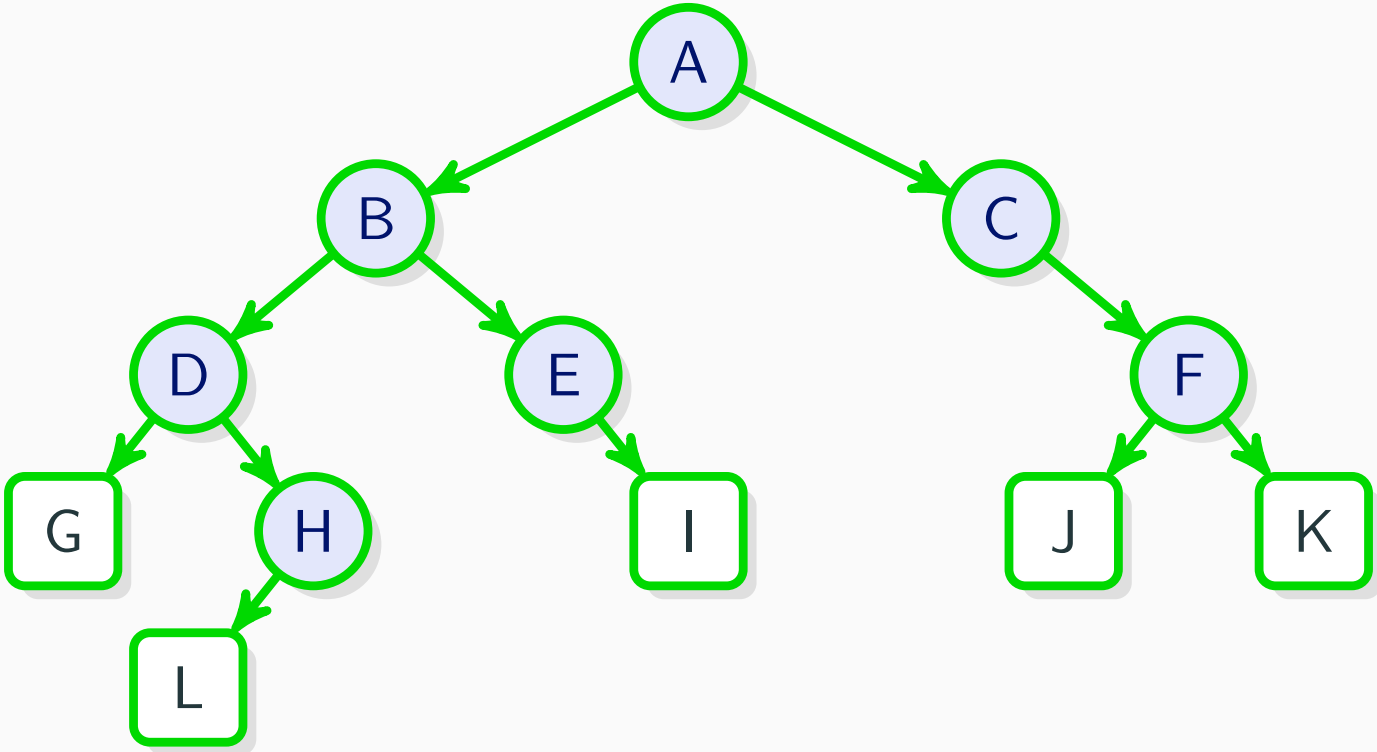
Parcours d'un arbre

Une fonction récursive visite les nœuds un par un



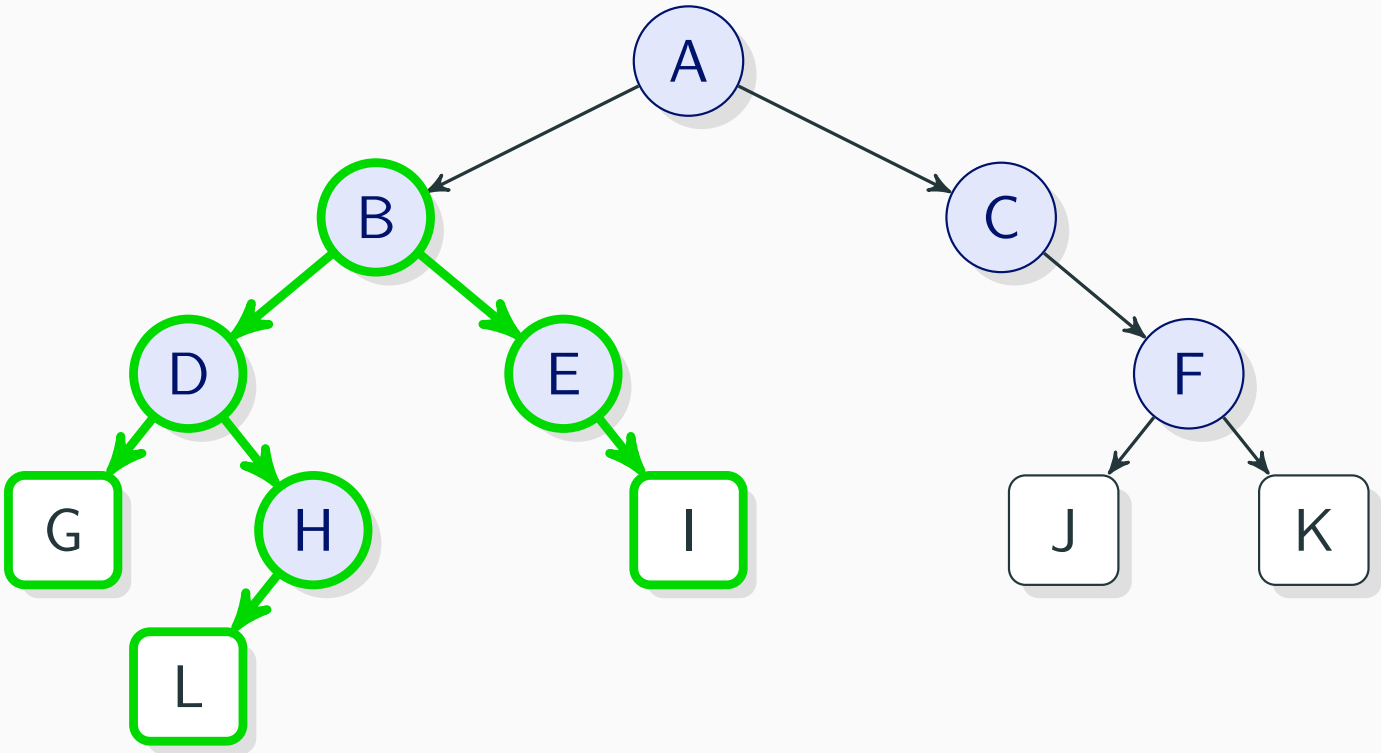
Parcours d'un arbre

Une fonction récursive visite les nœuds un par un



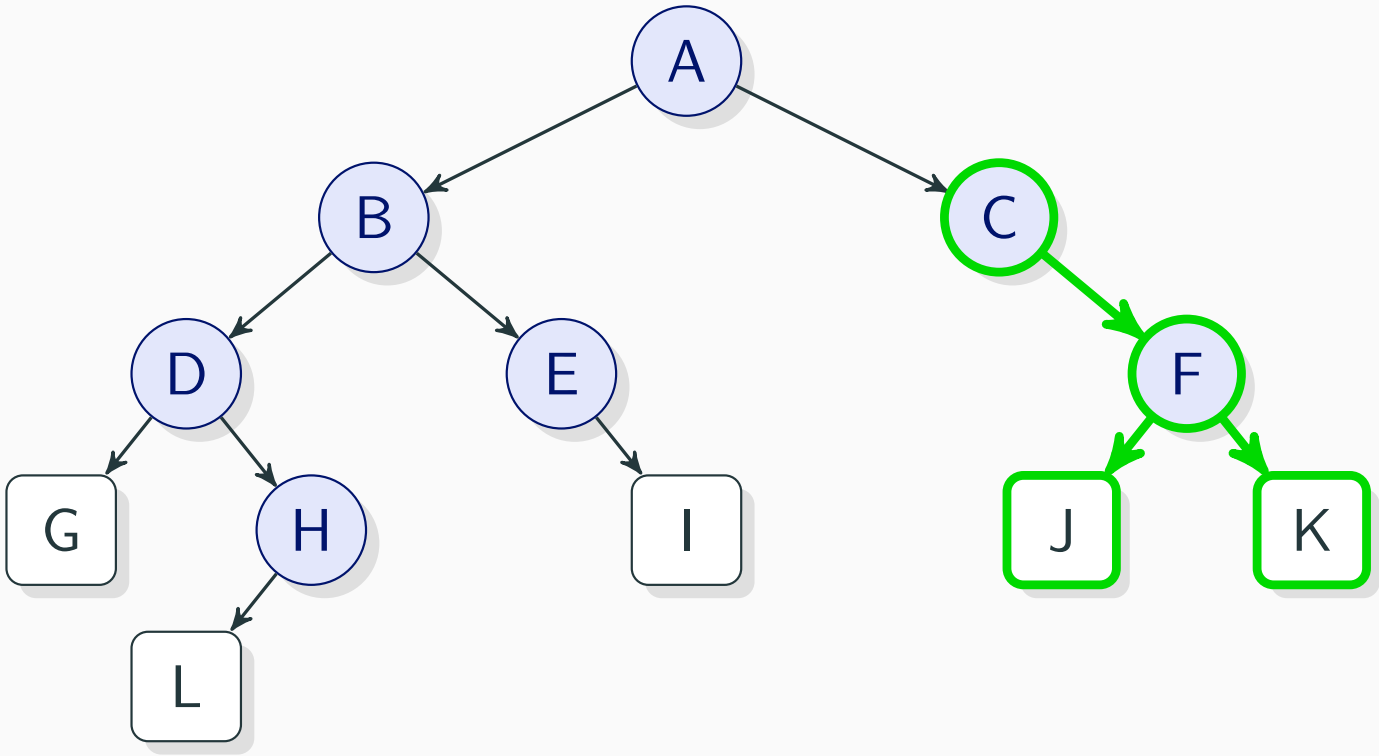
Parcours d'un arbre

Une fonction récursive visite les nœuds un par un



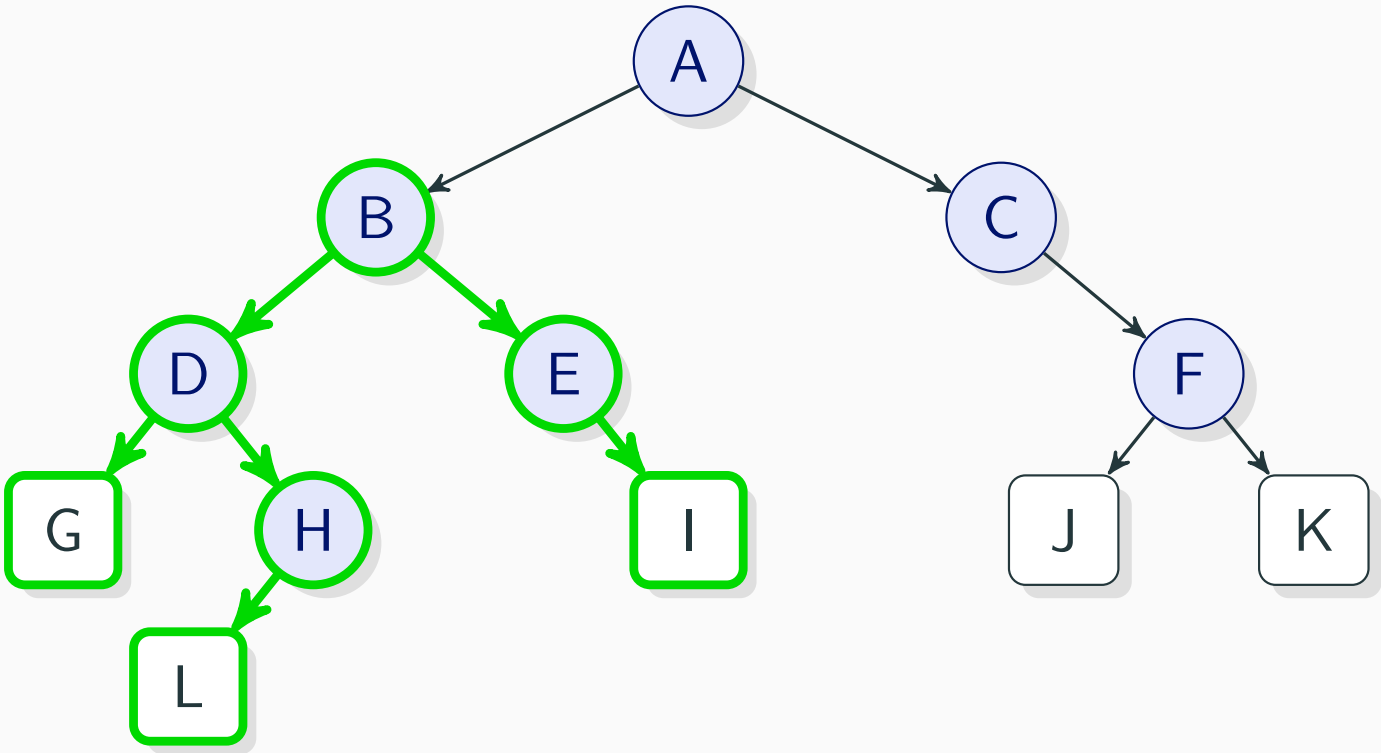
Parcours d'un arbre

Une fonction récursive visite les nœuds un par un



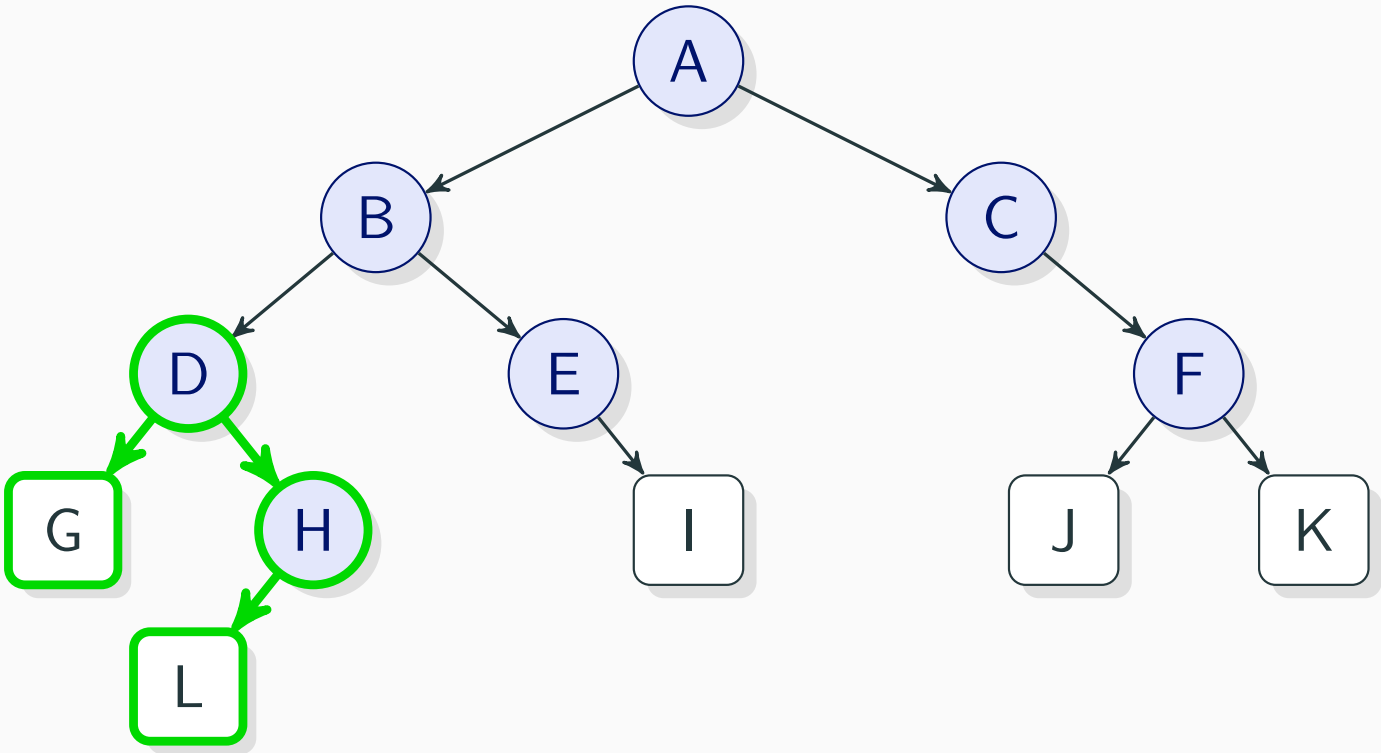
Parcours d'un arbre

Une fonction récursive visite les nœuds un par un



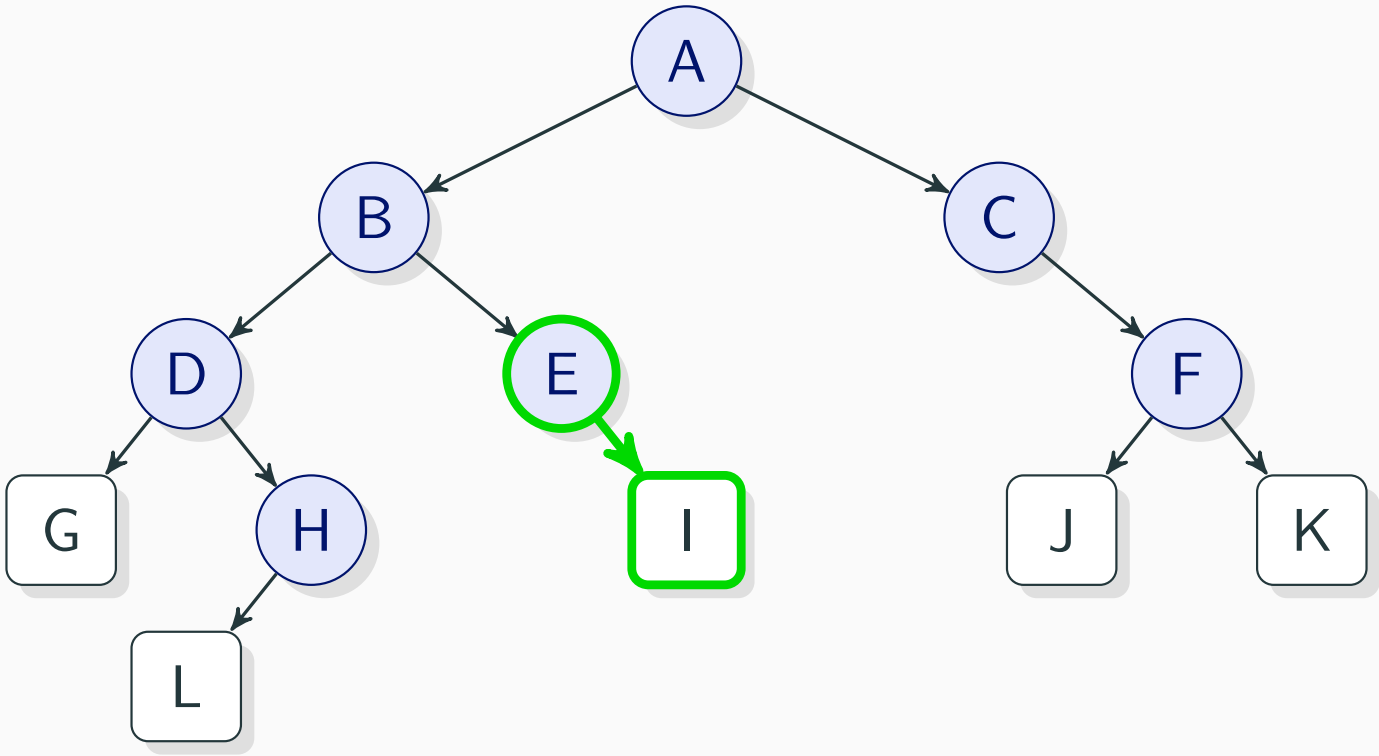
Parcours d'un arbre

Une fonction récursive visite les nœuds un par un



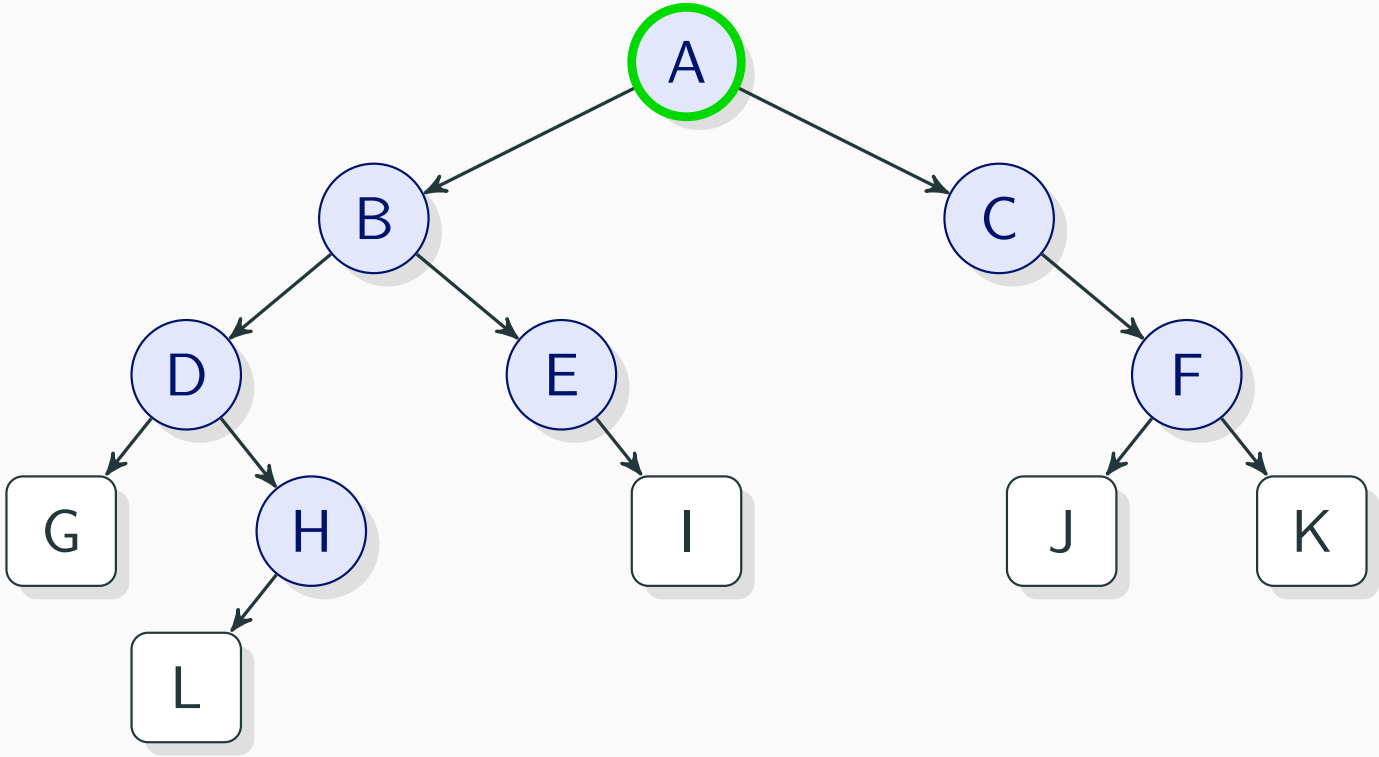
Parcours d'un arbre

Une fonction récursive visite les nœuds un par un



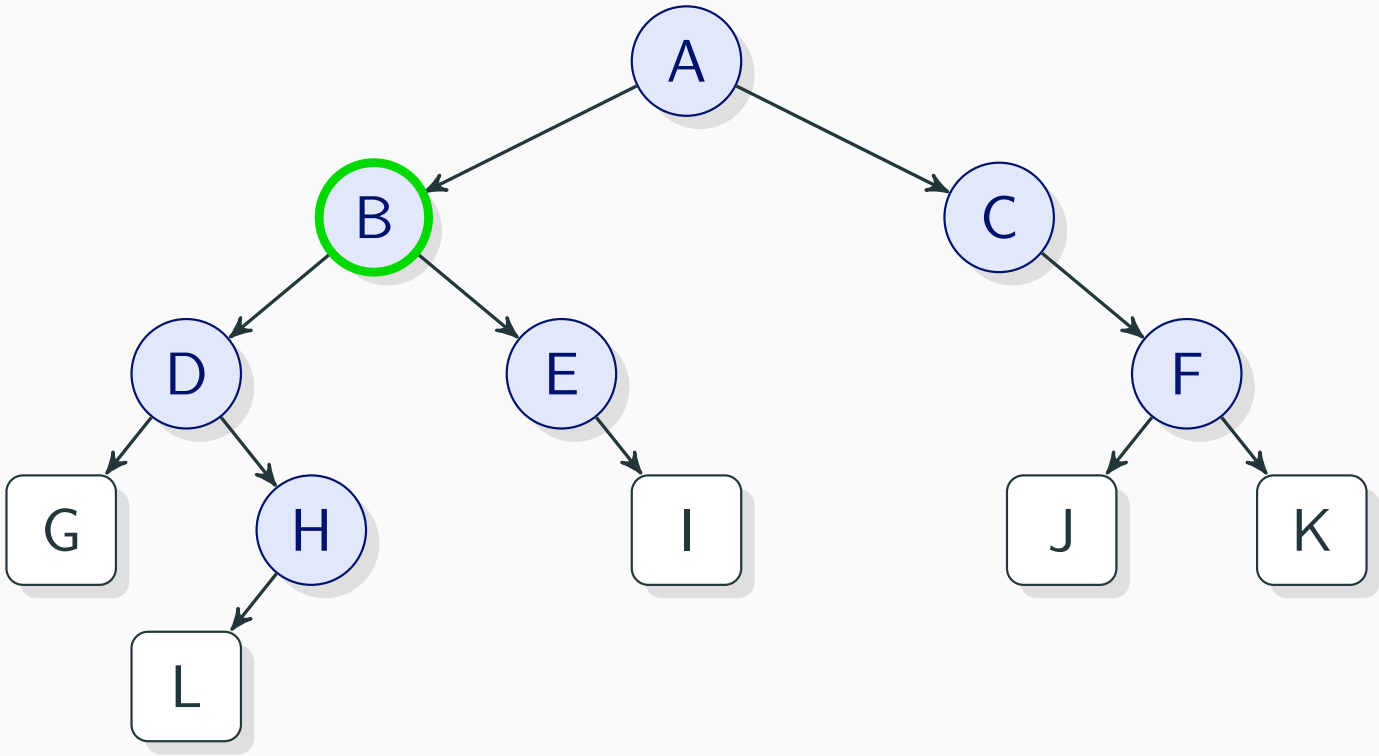
Parcours d'un arbre

Une fonction récursive visite les nœuds un par un



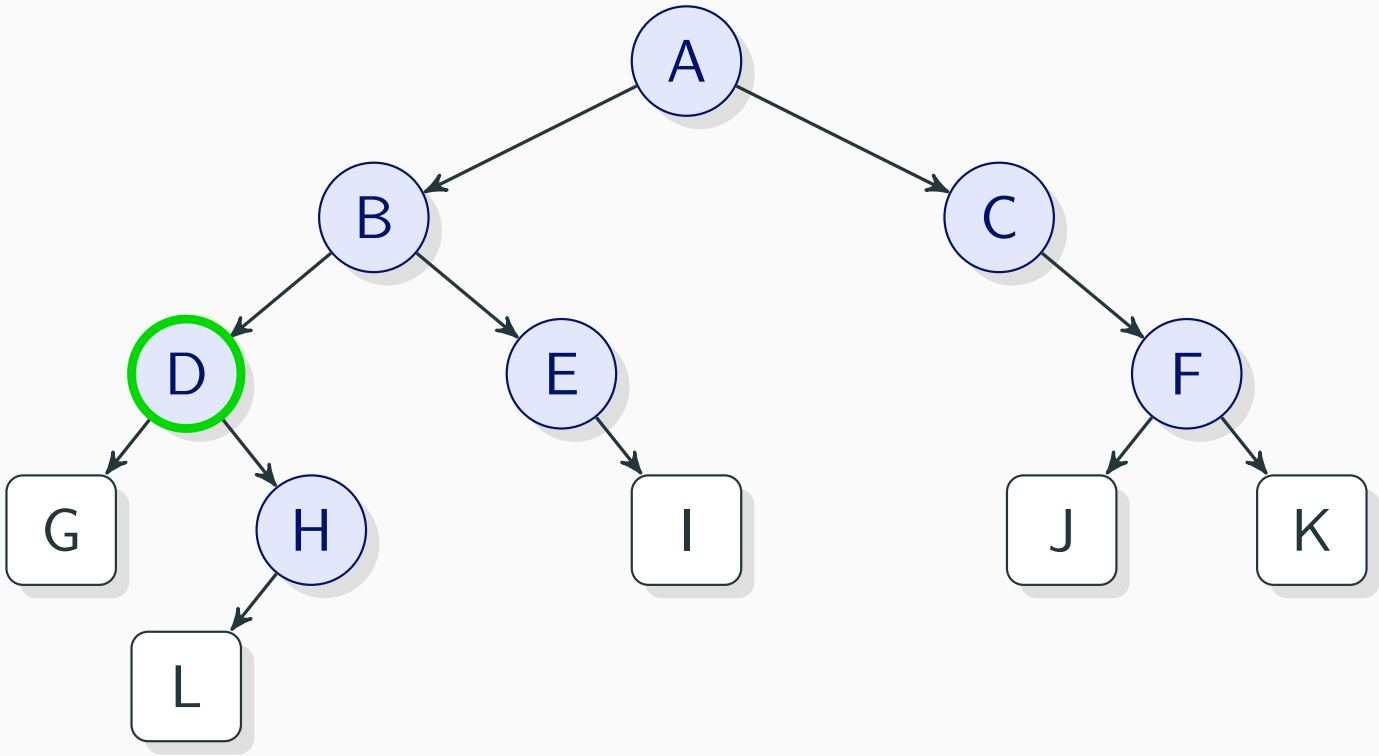
Parcours d'un arbre

Une fonction récursive visite les nœuds un par un



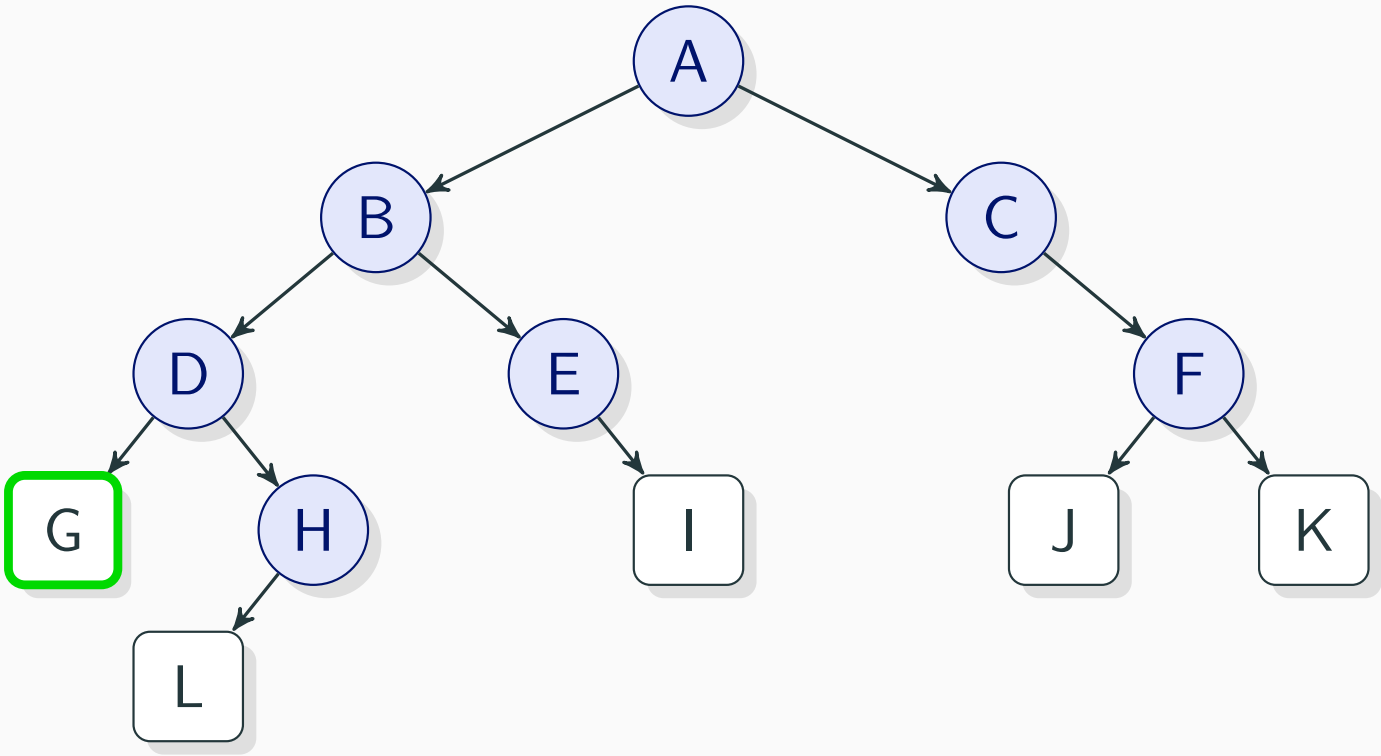
Parcours d'un arbre

Une fonction récursive visite les nœuds un par un



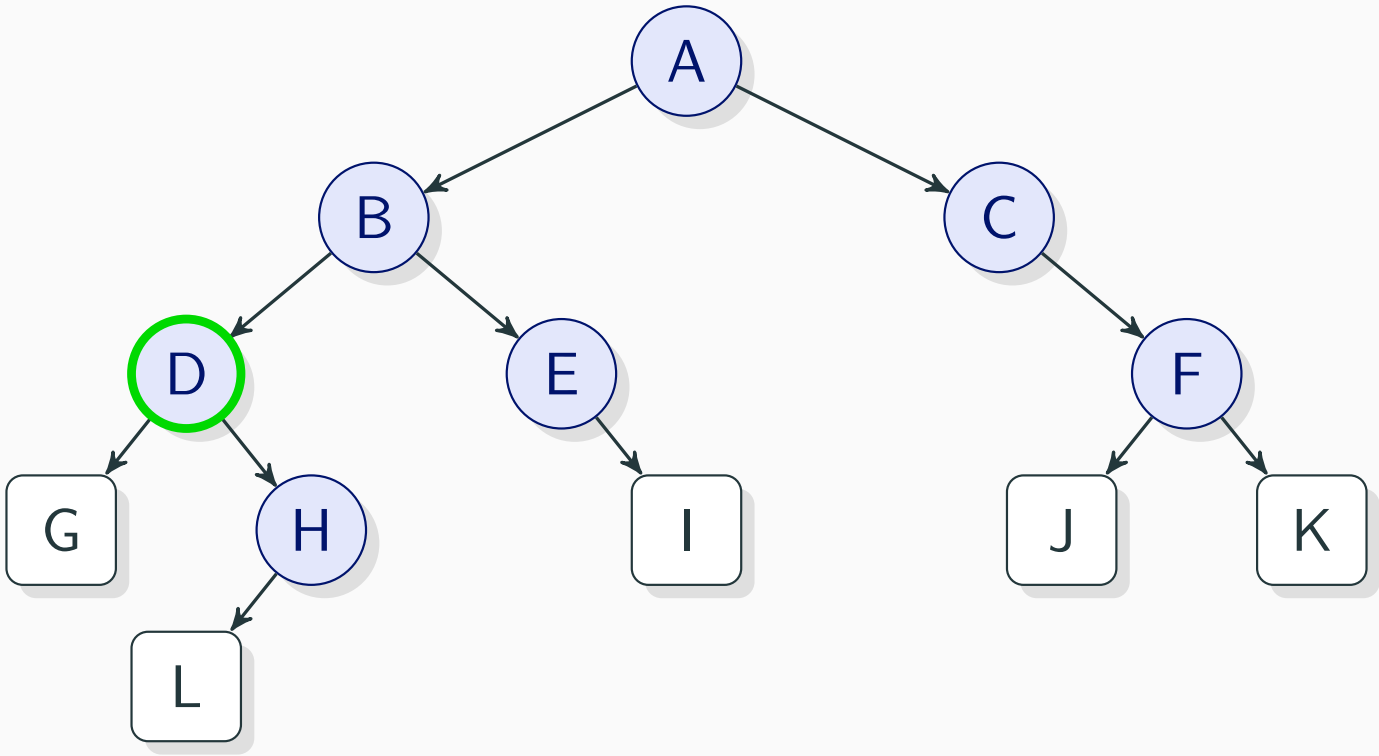
Parcours d'un arbre

Une fonction récursive visite les nœuds un par un



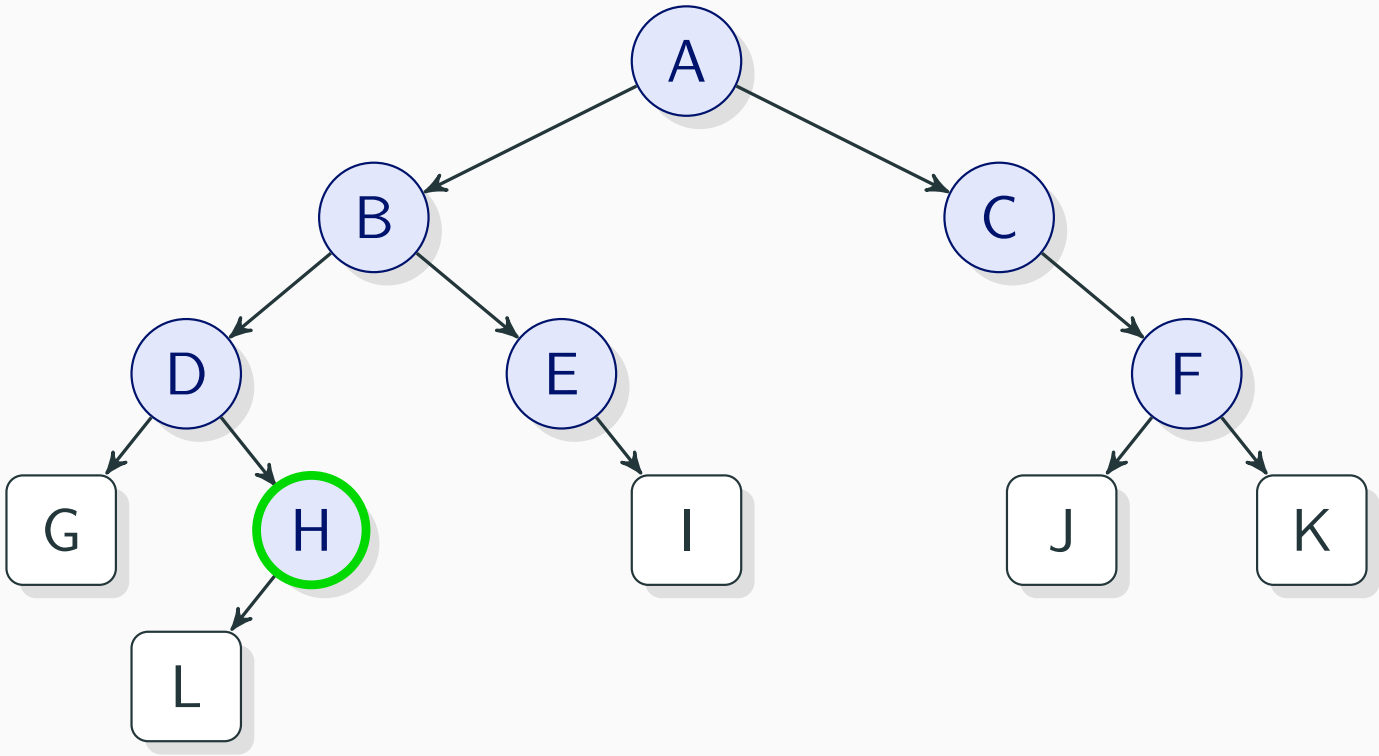
Parcours d'un arbre

Une fonction récursive visite les nœuds un par un



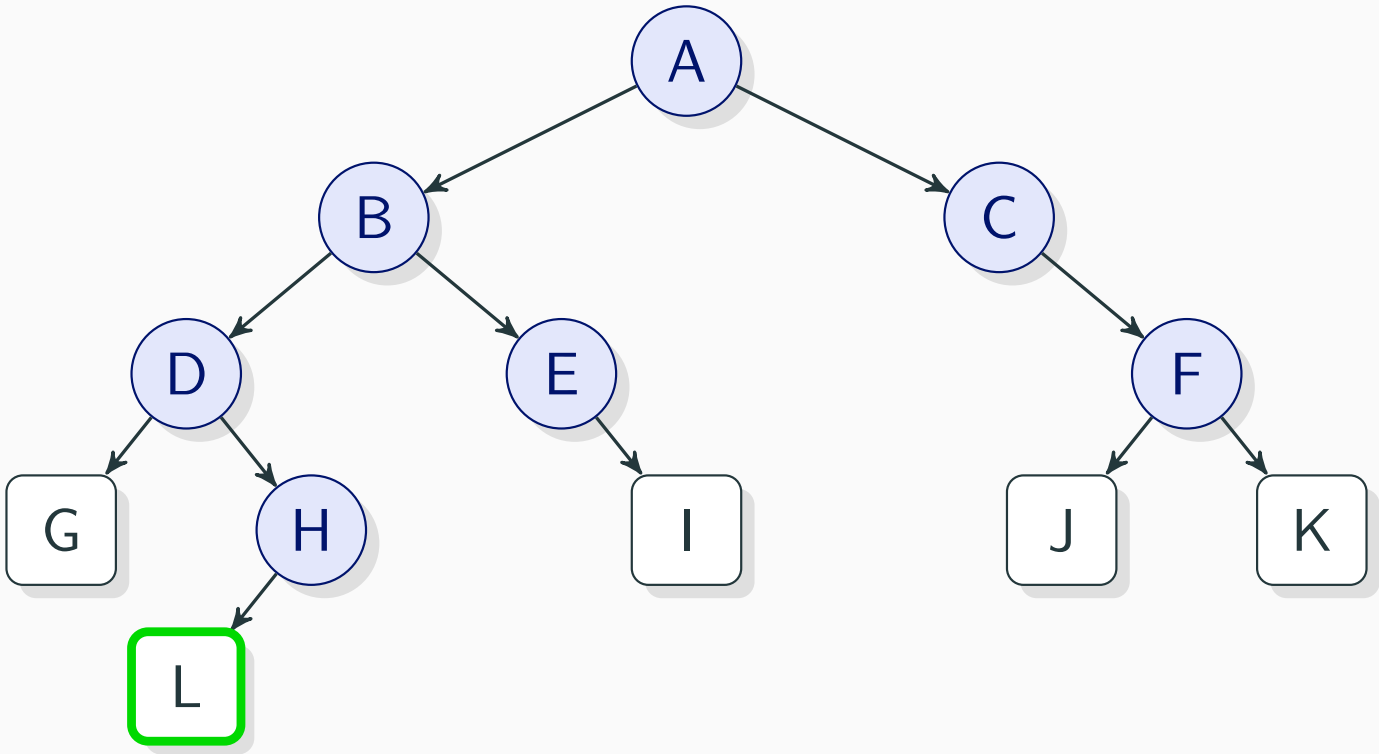
Parcours d'un arbre

Une fonction récursive visite les nœuds un par un



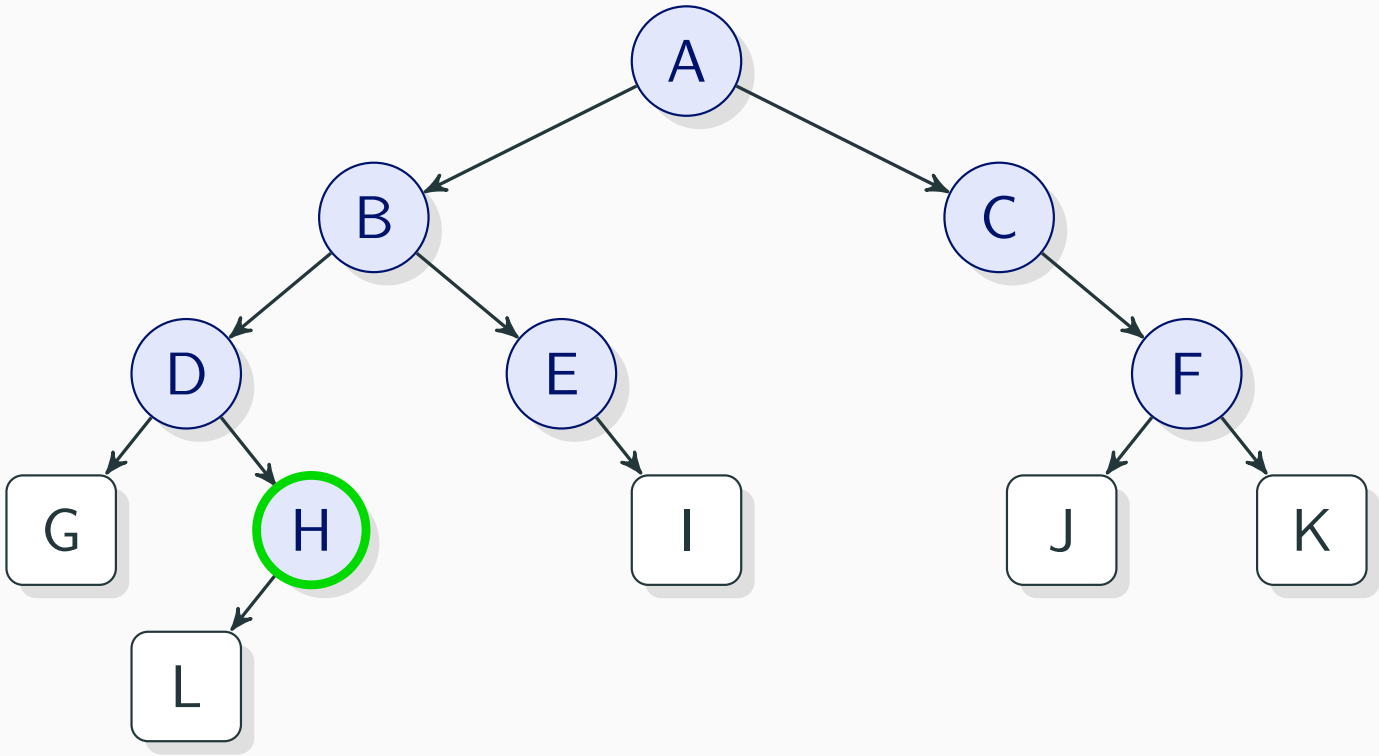
Parcours d'un arbre

Une fonction récursive visite les nœuds un par un



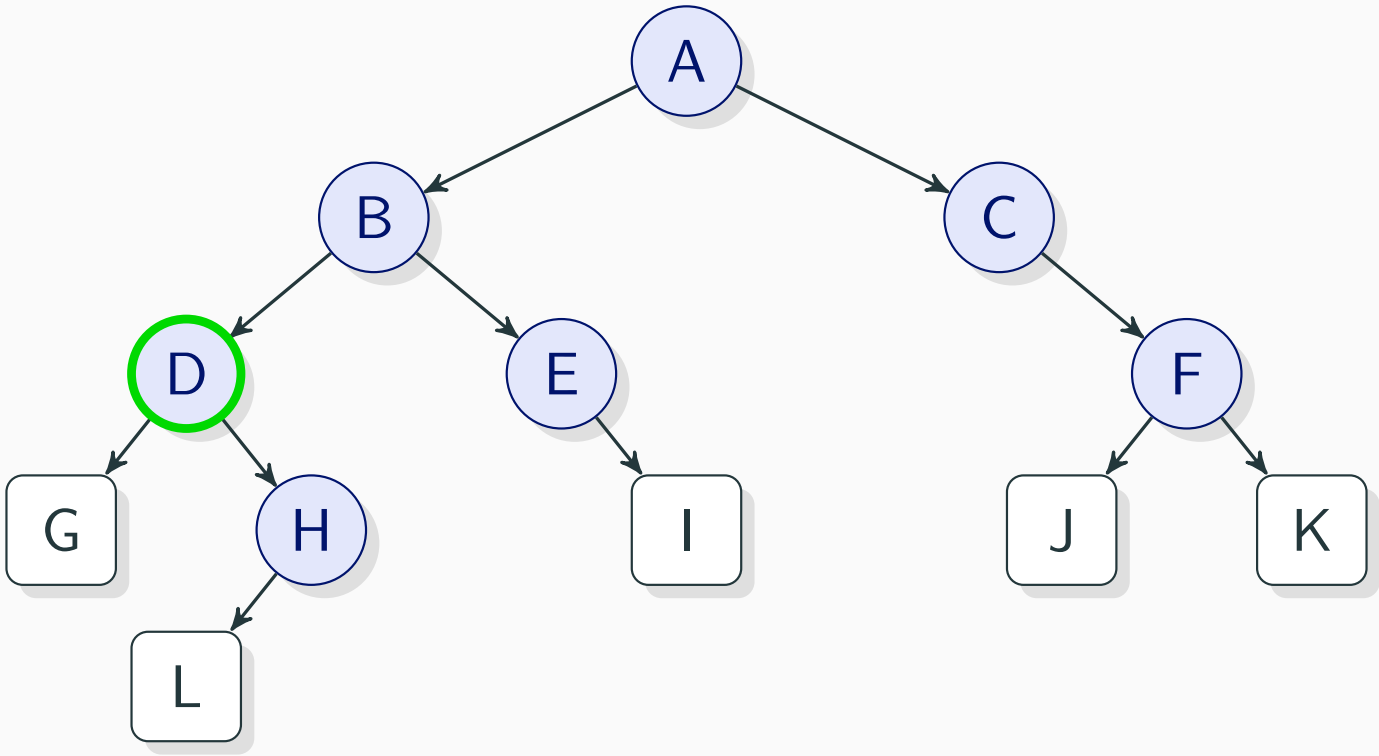
Parcours d'un arbre

Une fonction récursive visite les nœuds un par un



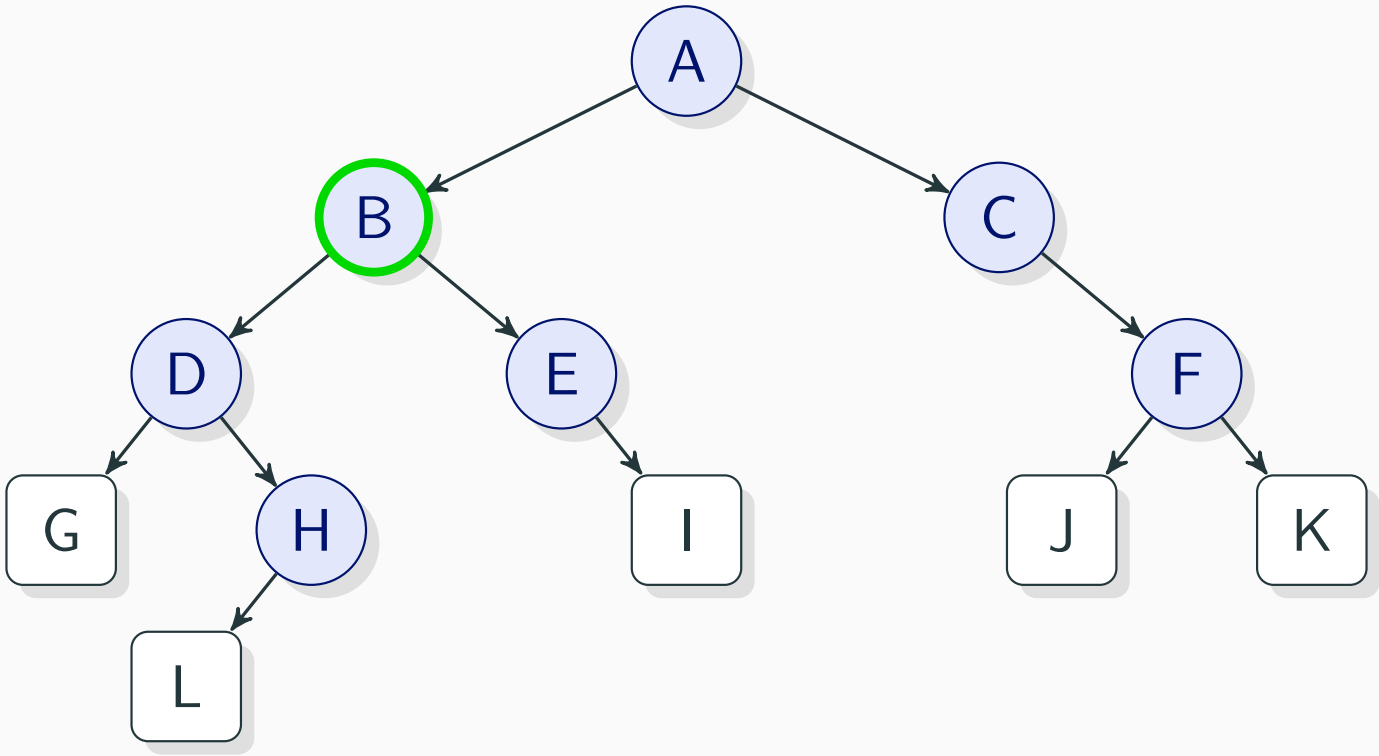
Parcours d'un arbre

Une fonction récursive visite les nœuds un par un



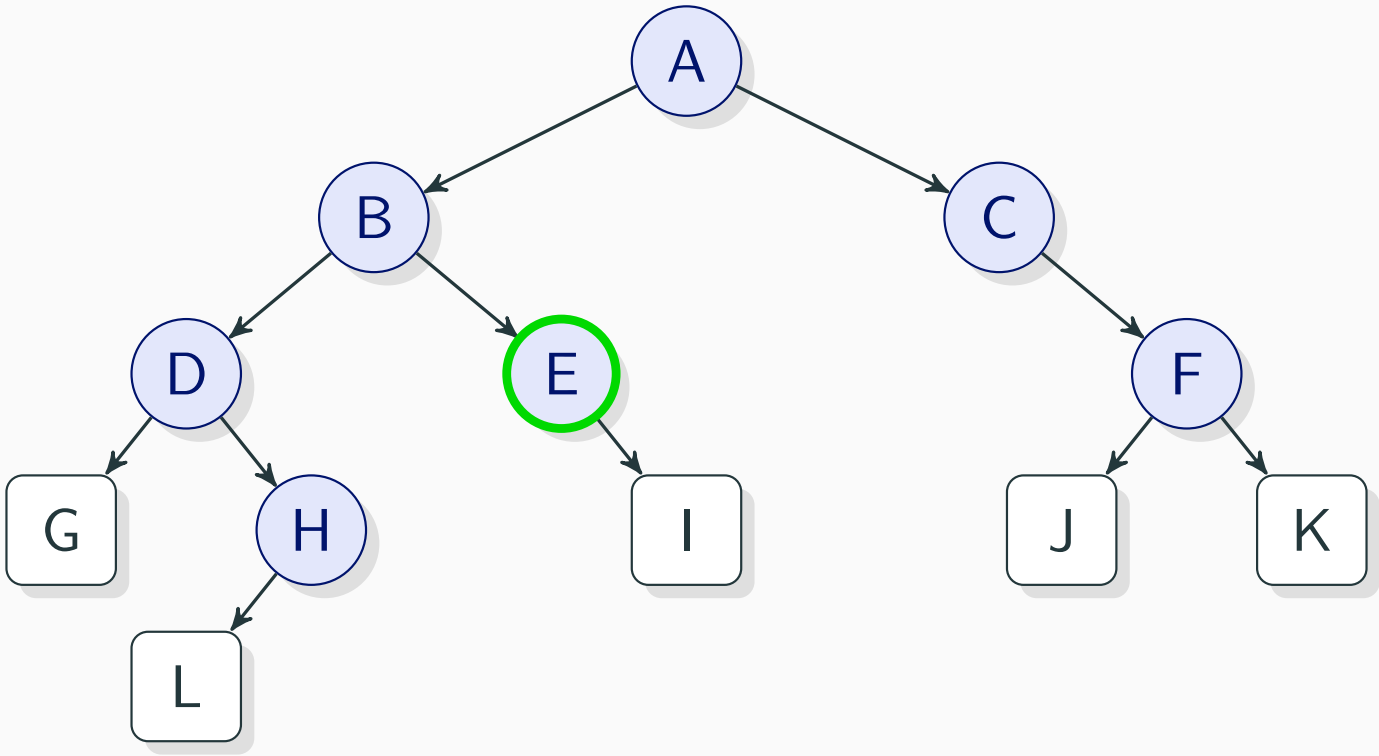
Parcours d'un arbre

Une fonction récursive visite les nœuds un par un



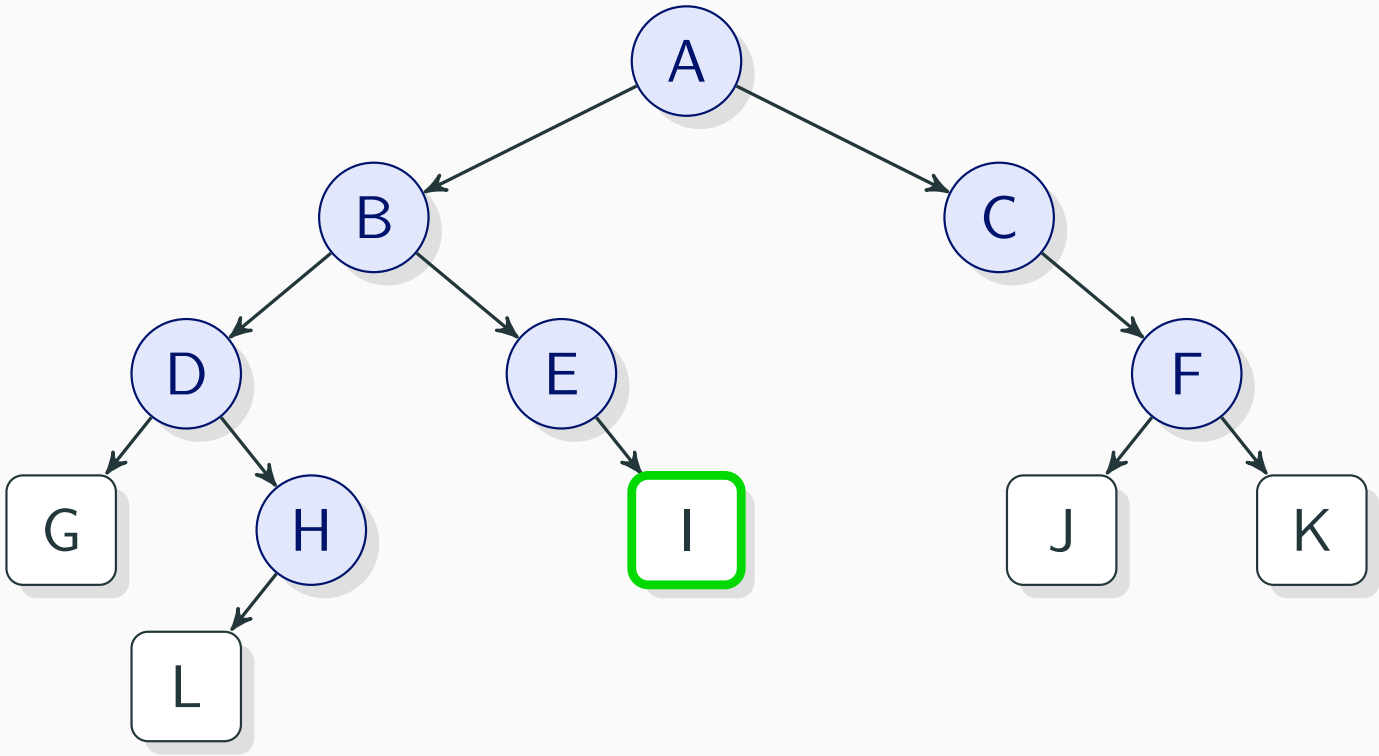
Parcours d'un arbre

Une fonction récursive visite les nœuds un par un



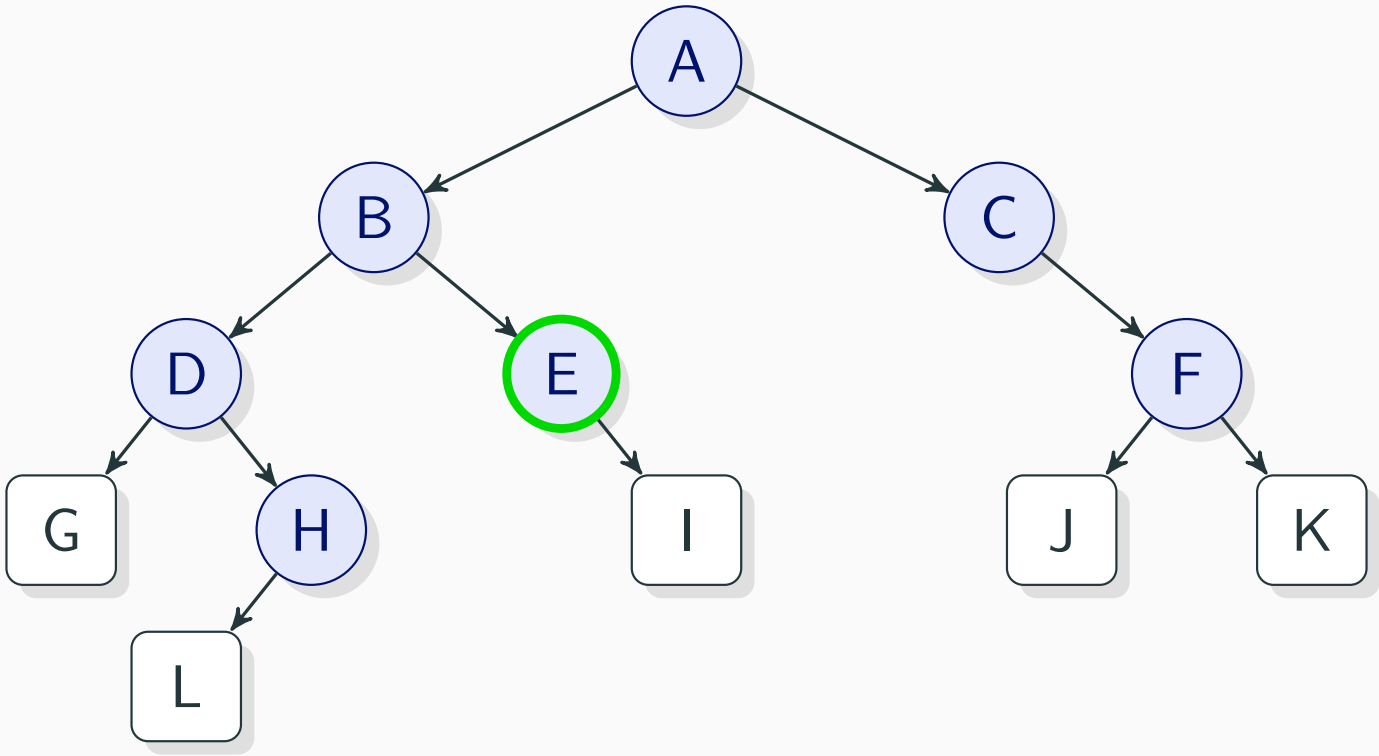
Parcours d'un arbre

Une fonction récursive visite les nœuds un par un



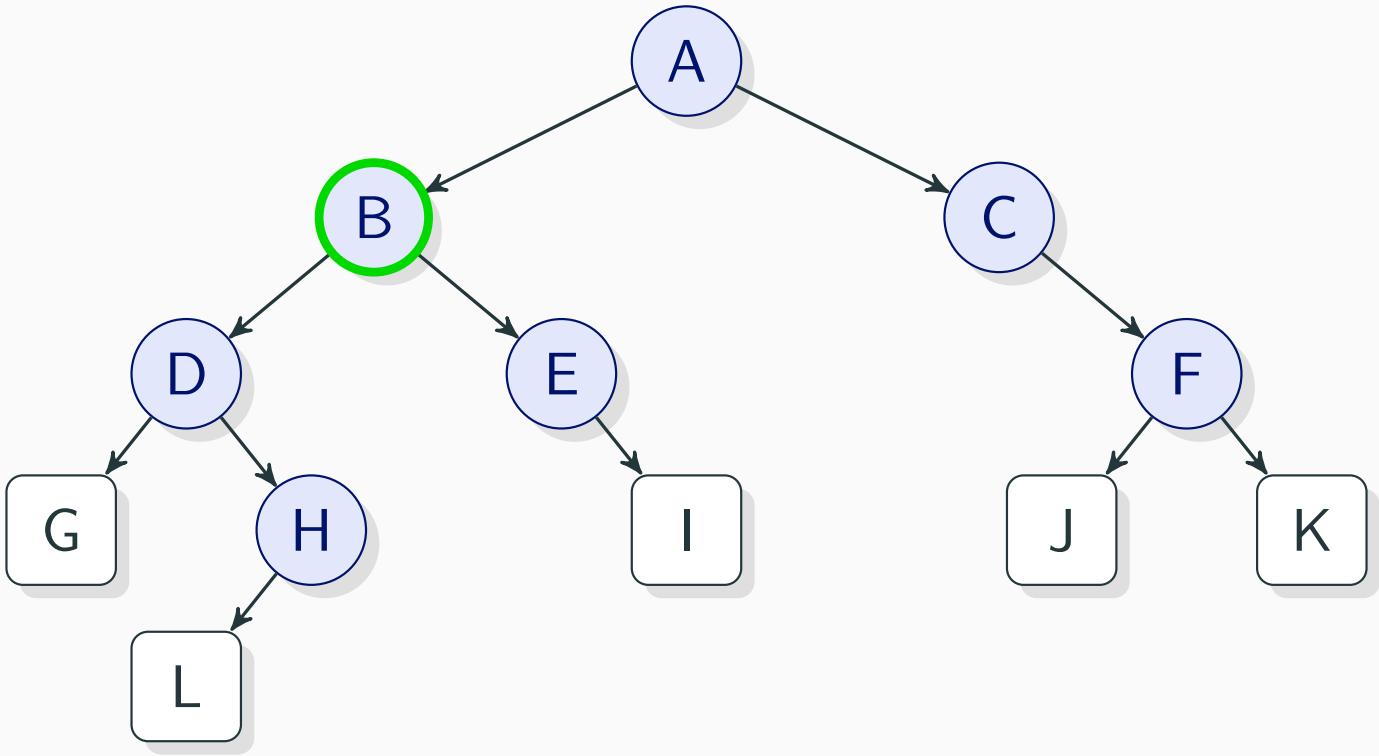
Parcours d'un arbre

Une fonction récursive visite les nœuds un par un



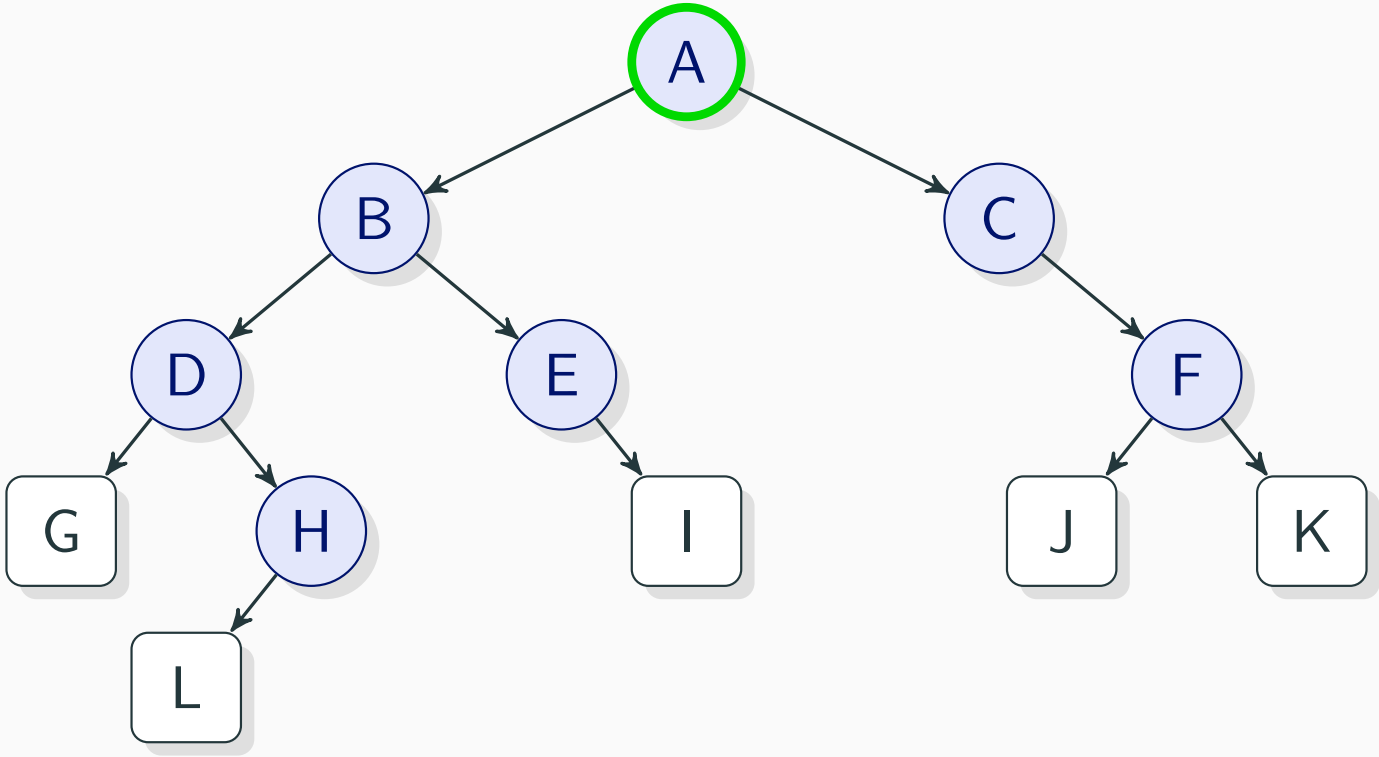
Parcours d'un arbre

Une fonction récursive visite les nœuds un par un



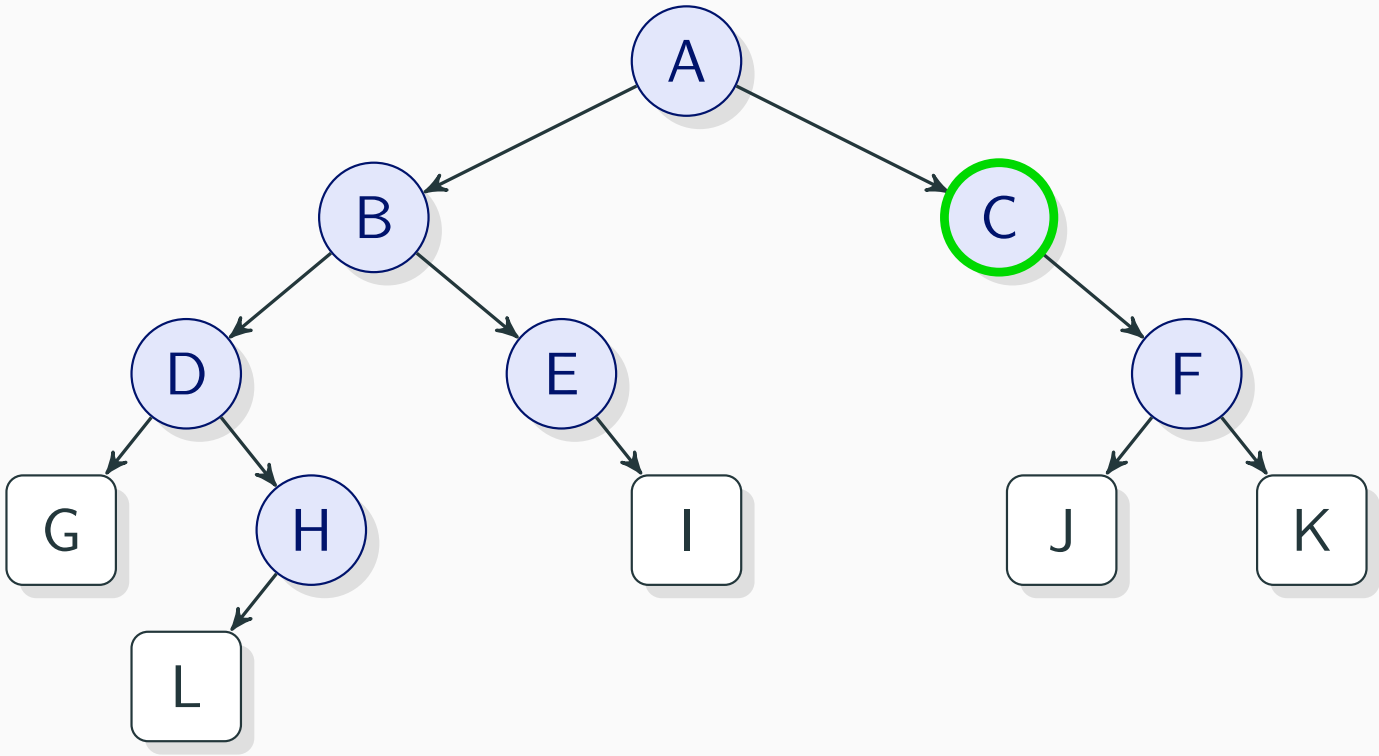
Parcours d'un arbre

Une fonction récursive visite les nœuds un par un



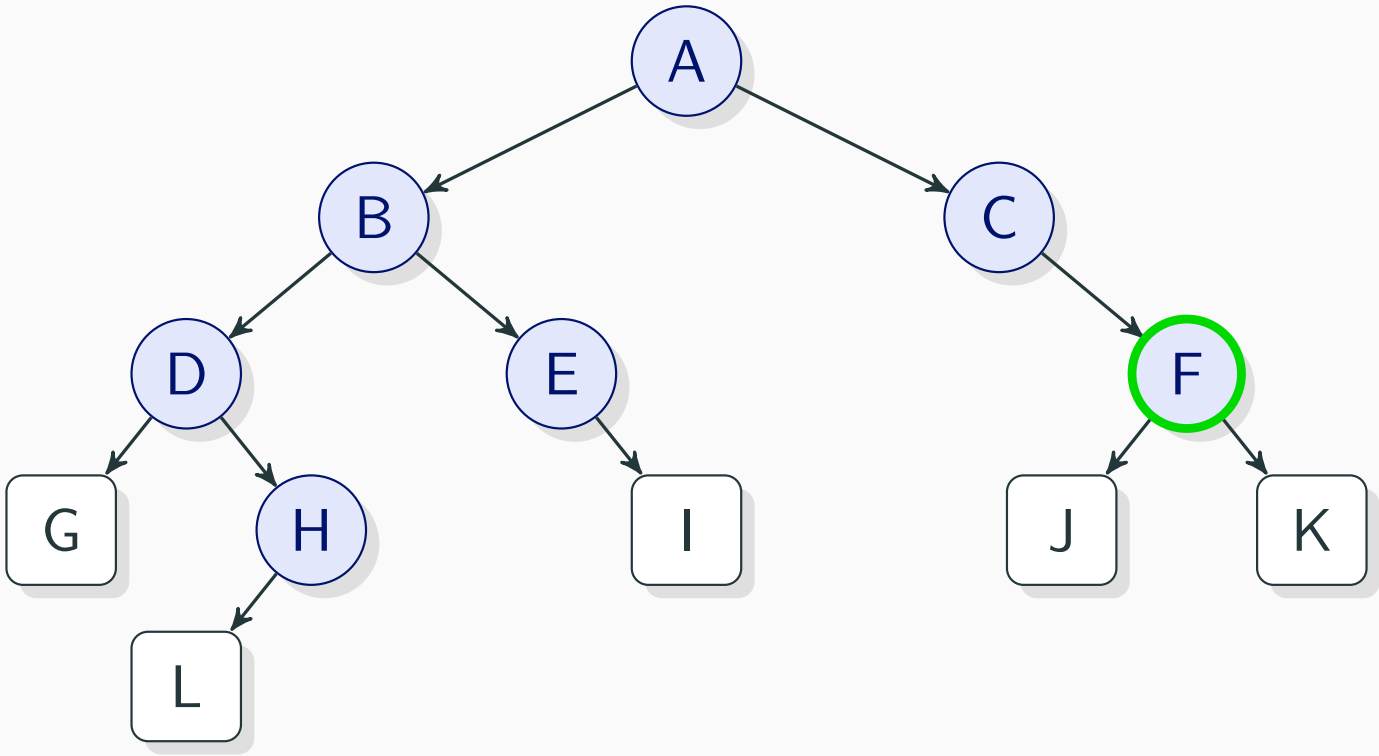
Parcours d'un arbre

Une fonction récursive visite les nœuds un par un



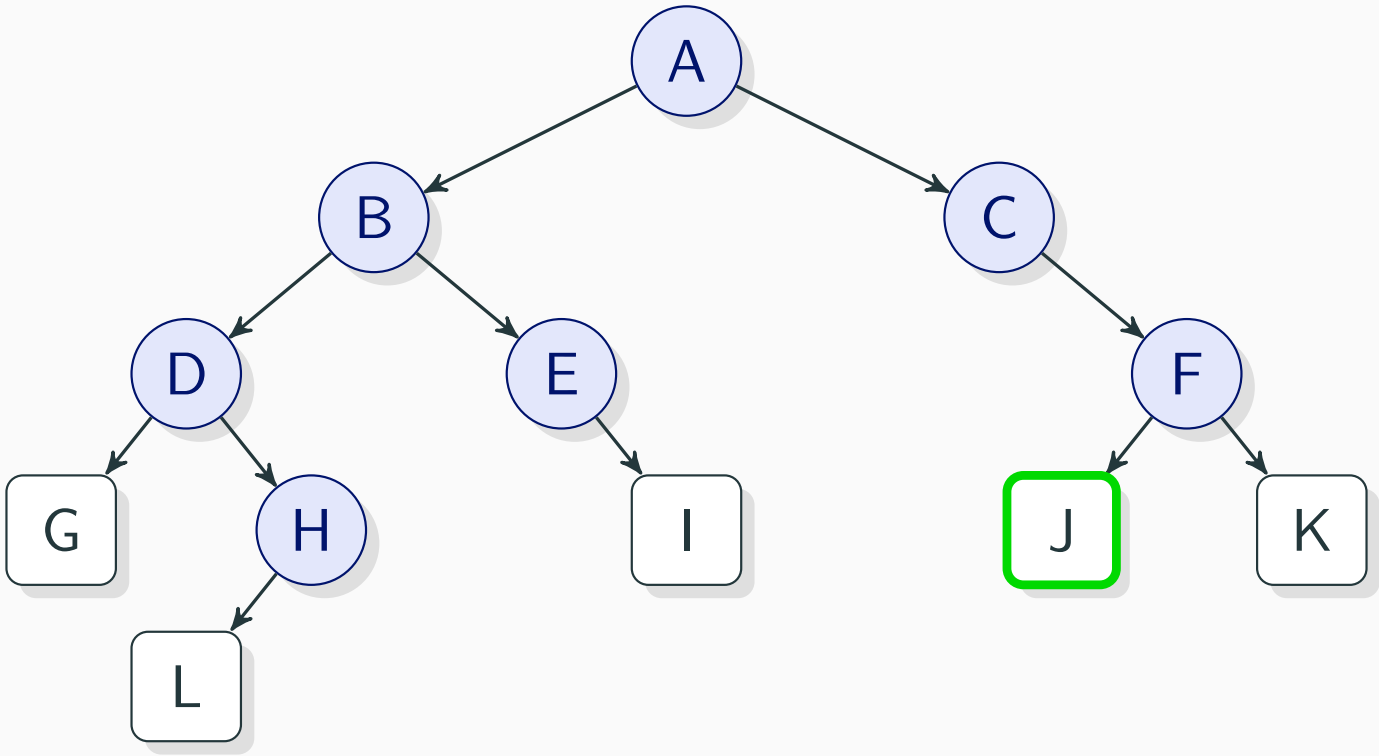
Parcours d'un arbre

Une fonction récursive visite les nœuds un par un



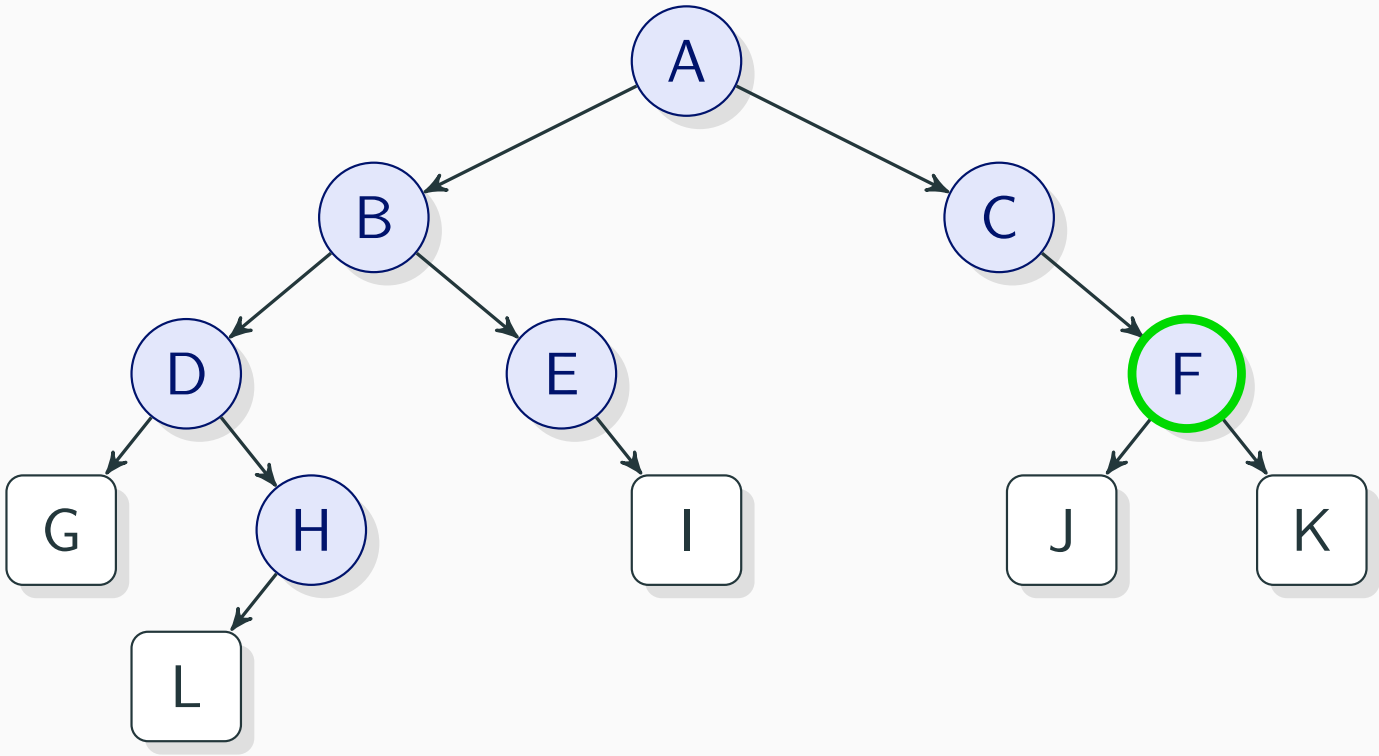
Parcours d'un arbre

Une fonction récursive visite les nœuds un par un



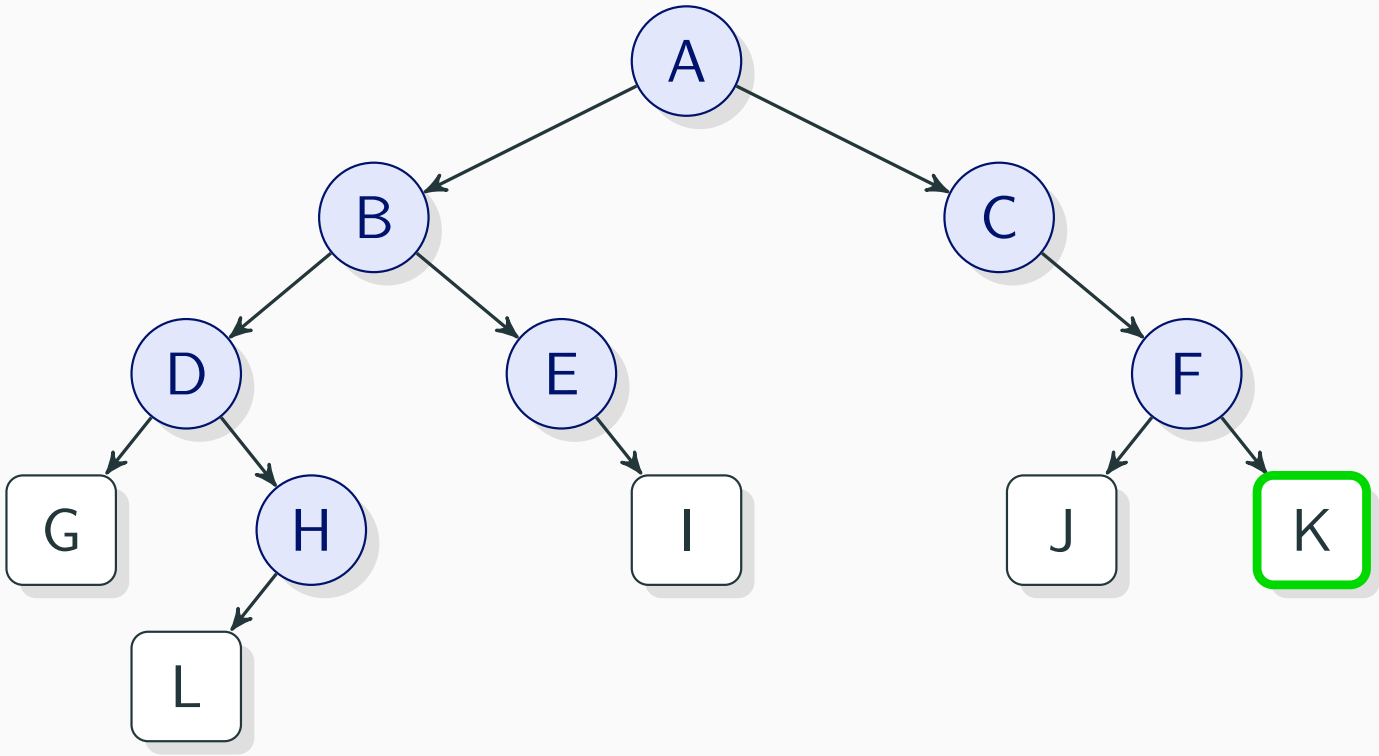
Parcours d'un arbre

Une fonction récursive visite les nœuds un par un



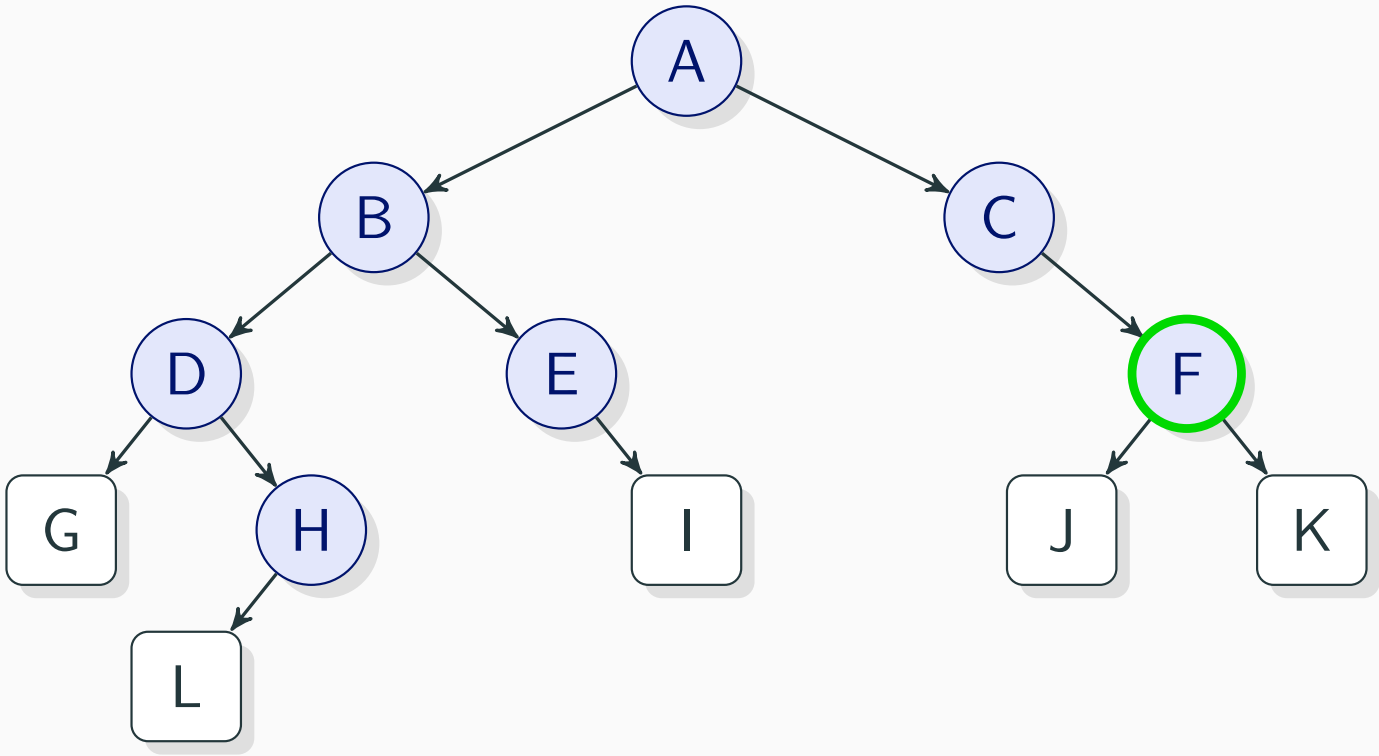
Parcours d'un arbre

Une fonction récursive visite les nœuds un par un



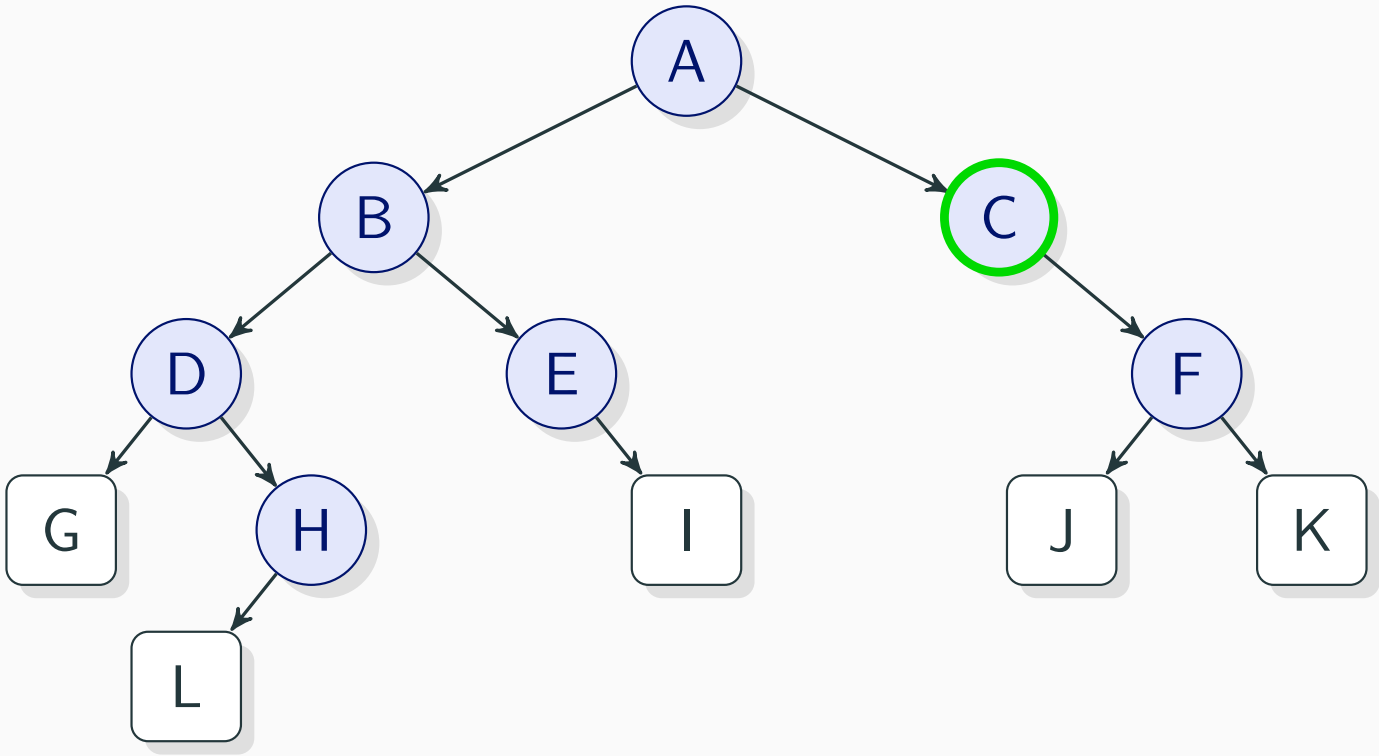
Parcours d'un arbre

Une fonction récursive visite les nœuds un par un



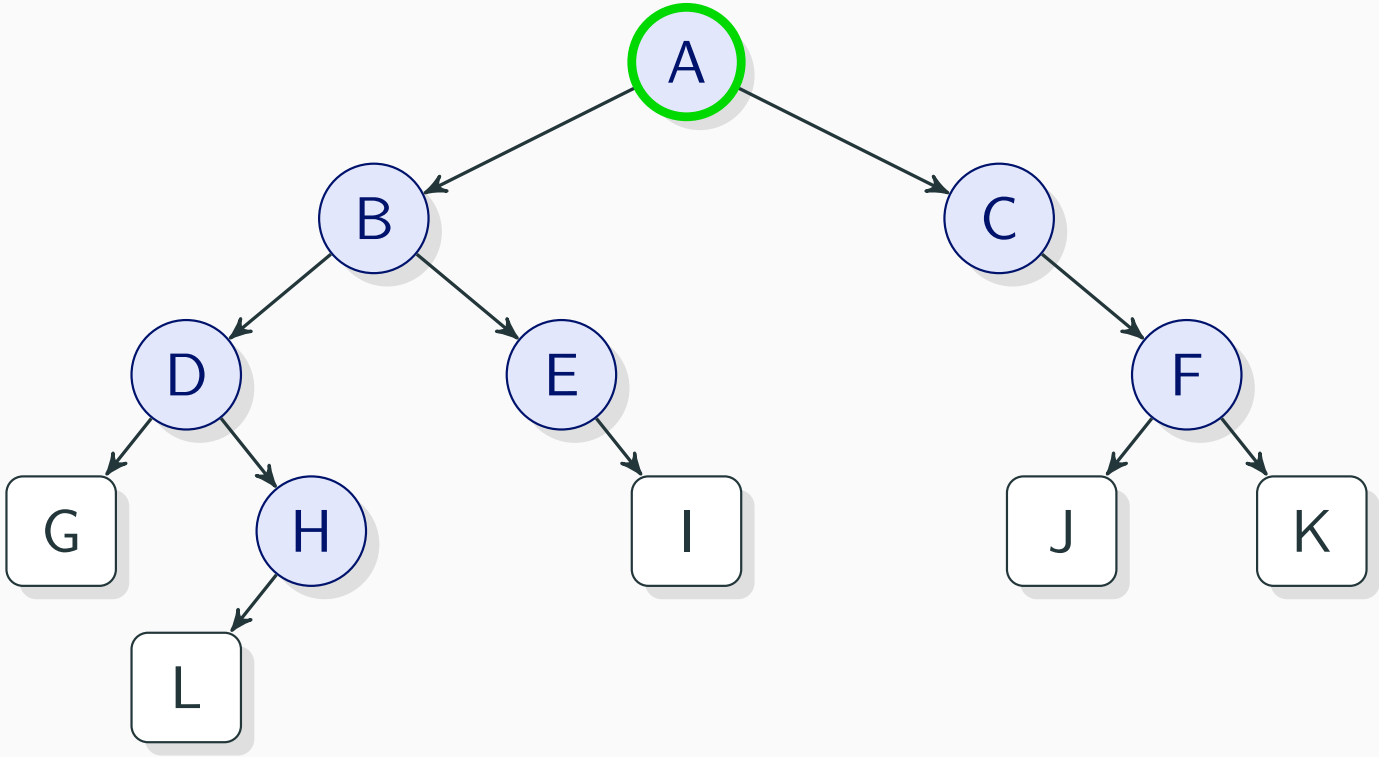
Parcours d'un arbre

Une fonction récursive visite les nœuds un par un



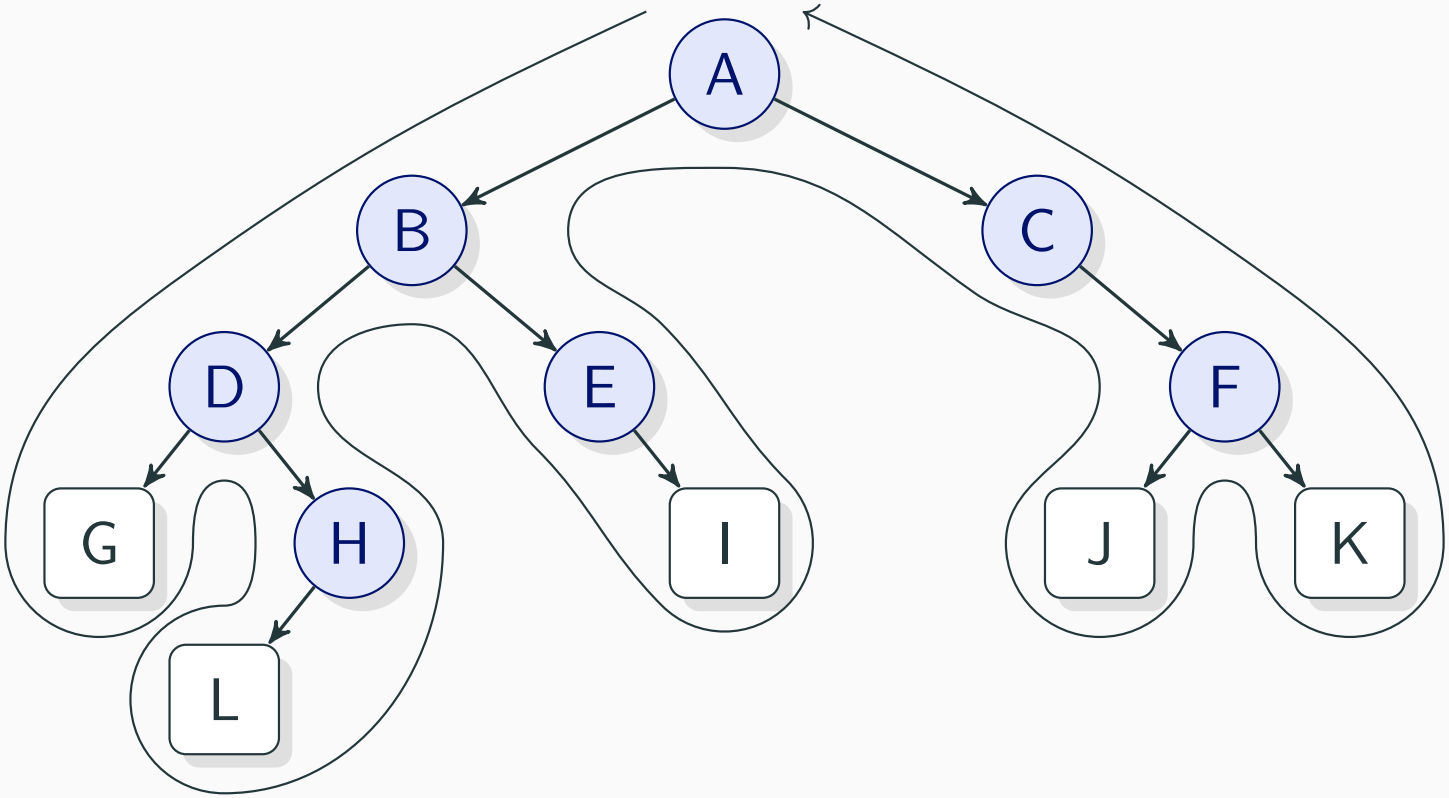
Parcours d'un arbre

Une fonction récursive visite les nœuds un par un



Parcours d'un arbre

Une fonction récursive visite les nœuds un par un



C'est un parcours *en profondeur*

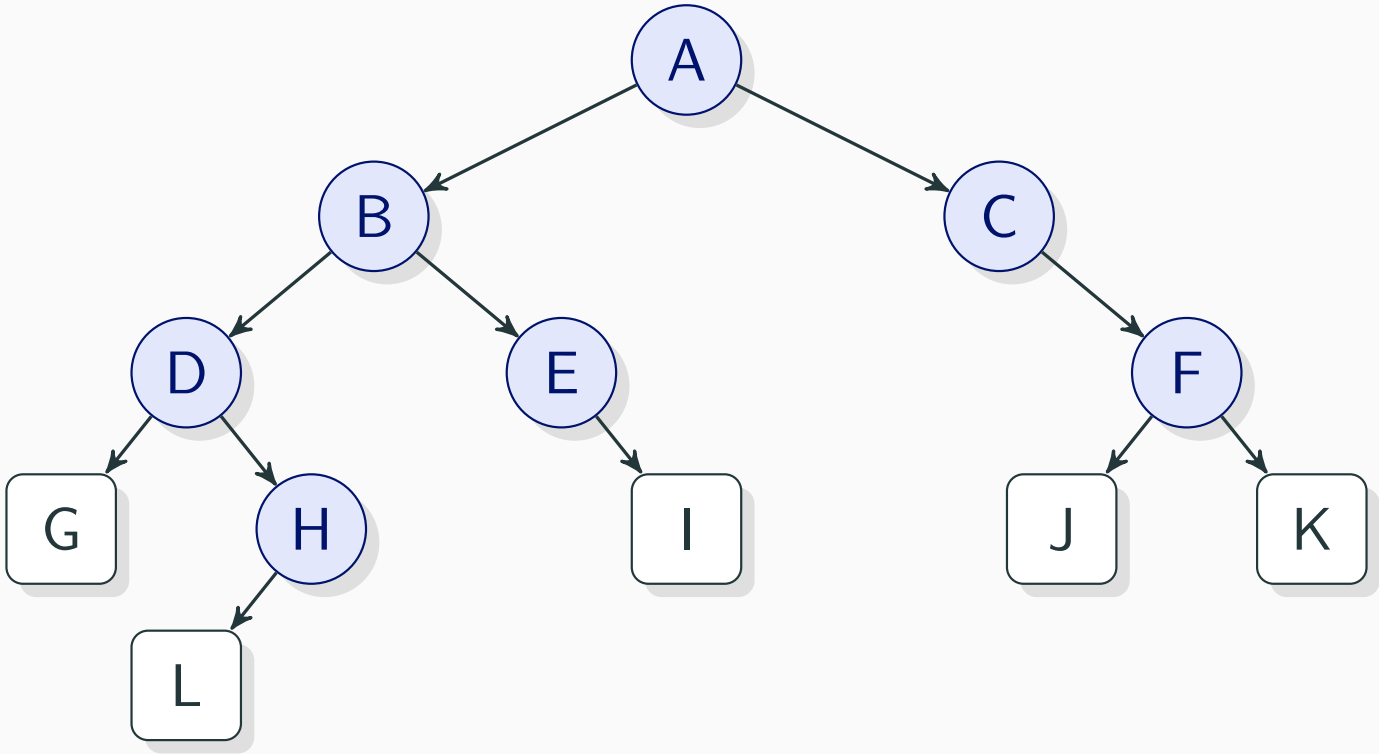
Parcours en profondeur (DFS)


```
# let rec dfs f = function
    Node (e, lst)
      -> f e;
         List.iter (dfs f) lst;;

val dfs : ('a -> unit) -> 'a tree -> unit = <fun>
```

Parcours en largeur (BFS)

Il existe un autre type de parcours, en *largeur* (hiérarchique)

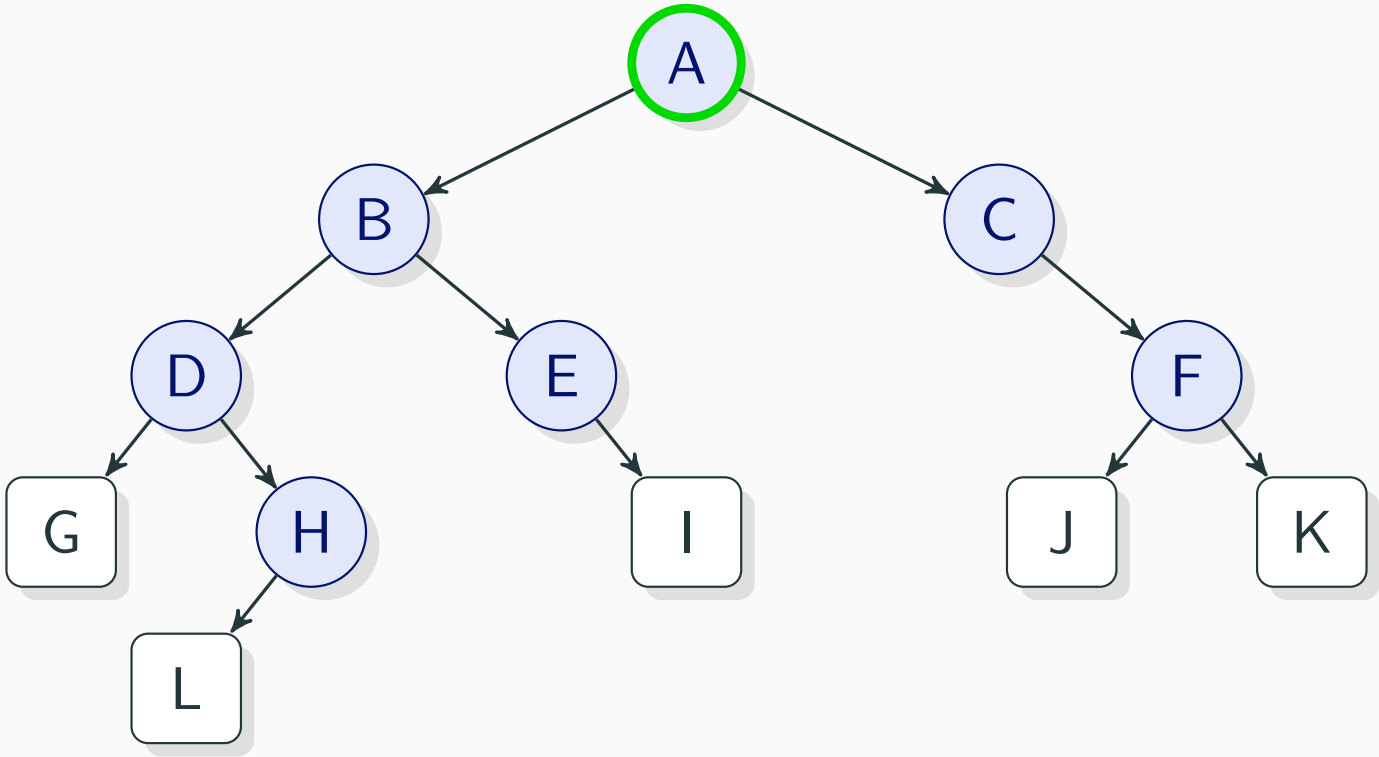
Les nœuds sont visités par profondeur croissante :



Parcours en largeur (BFS)

Il existe un autre type de parcours, en *largeur* (hiérarchique)

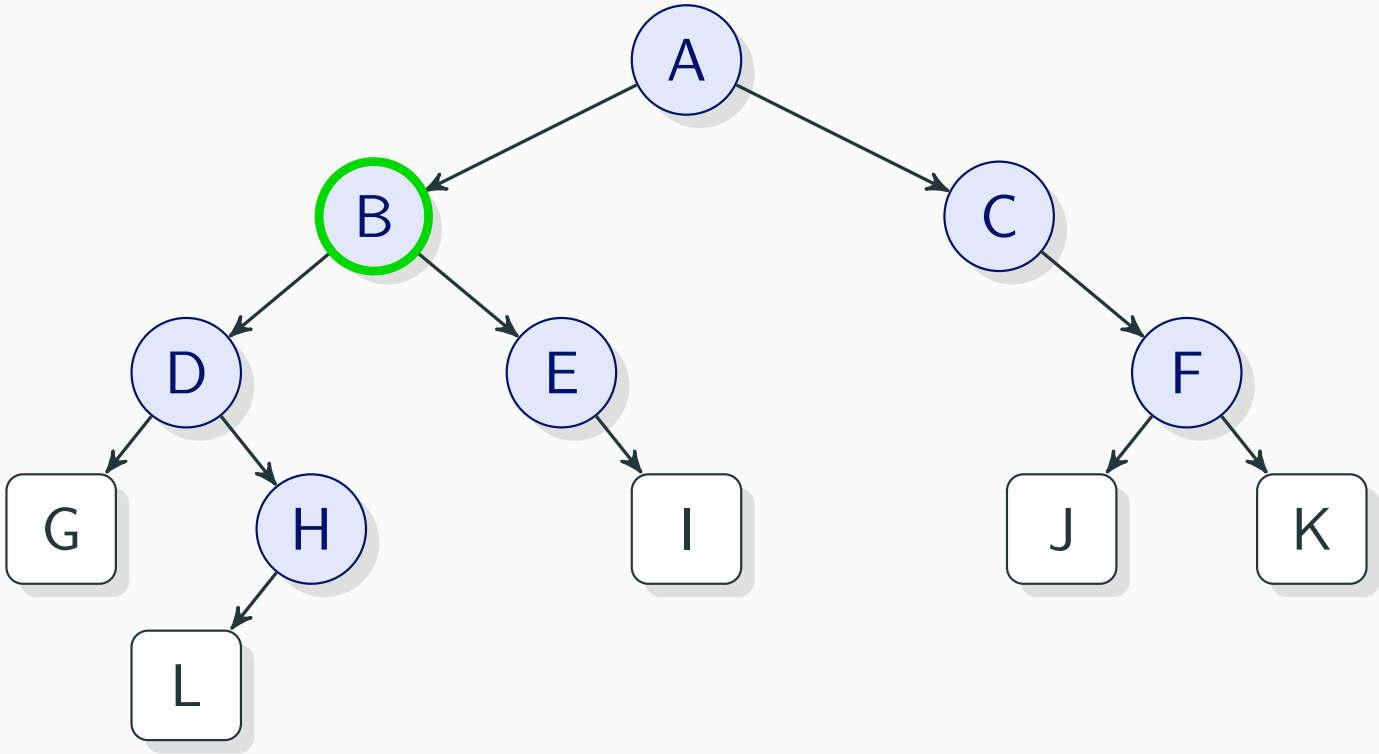
Les nœuds sont visités par profondeur croissante :



Parcours en largeur (BFS)

Il existe un autre type de parcours, en *largeur* (hiérarchique)

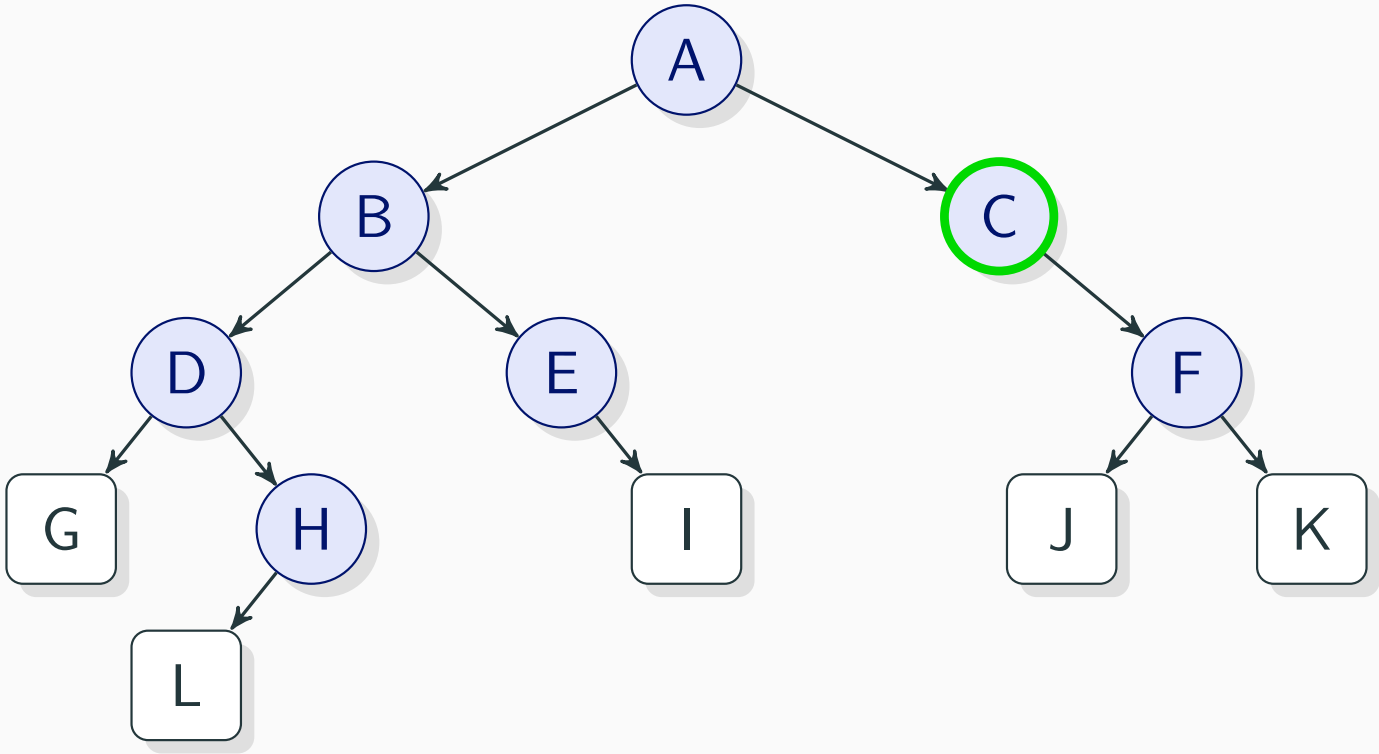
Les nœuds sont visités par profondeur croissante :



Parcours en largeur (BFS)

Il existe un autre type de parcours, en *largeur* (hiérarchique)

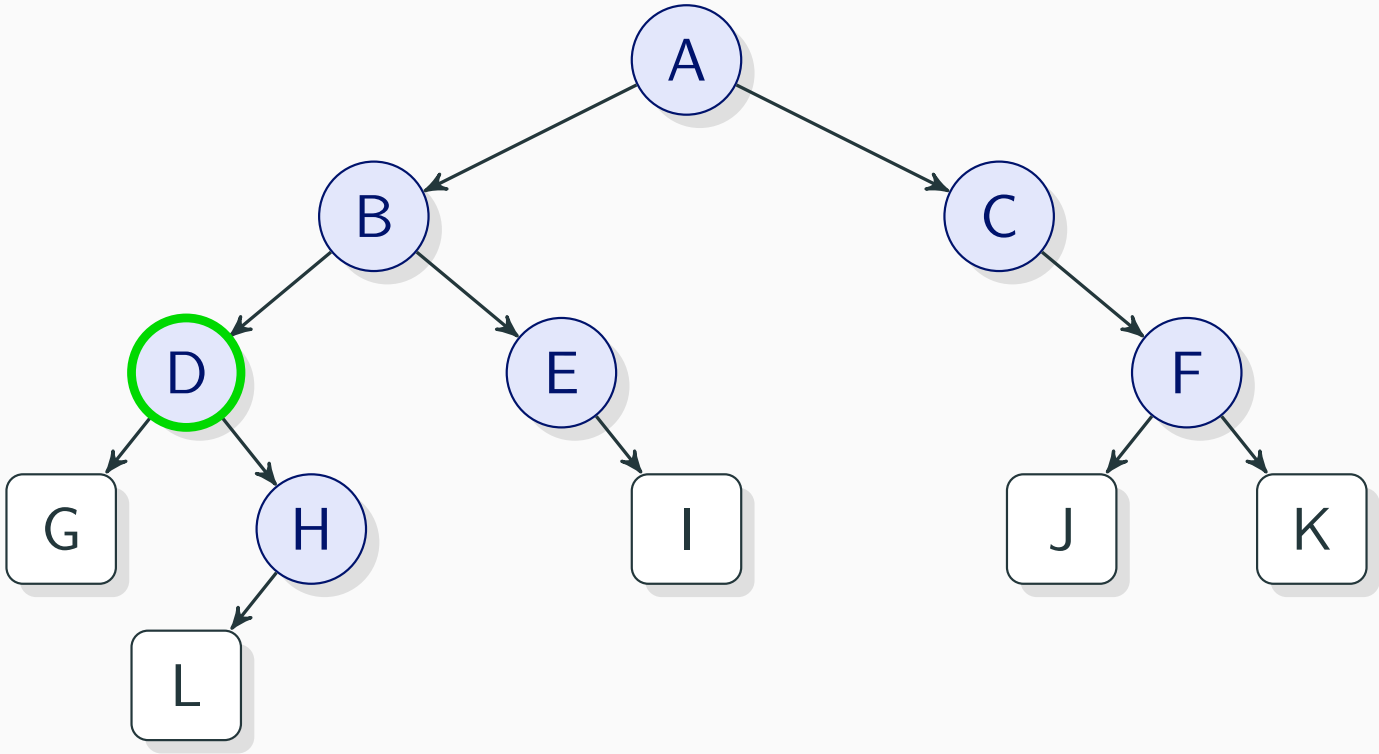
Les nœuds sont visités par profondeur croissante :



Parcours en largeur (BFS)

Il existe un autre type de parcours, en *largeur* (hiérarchique)

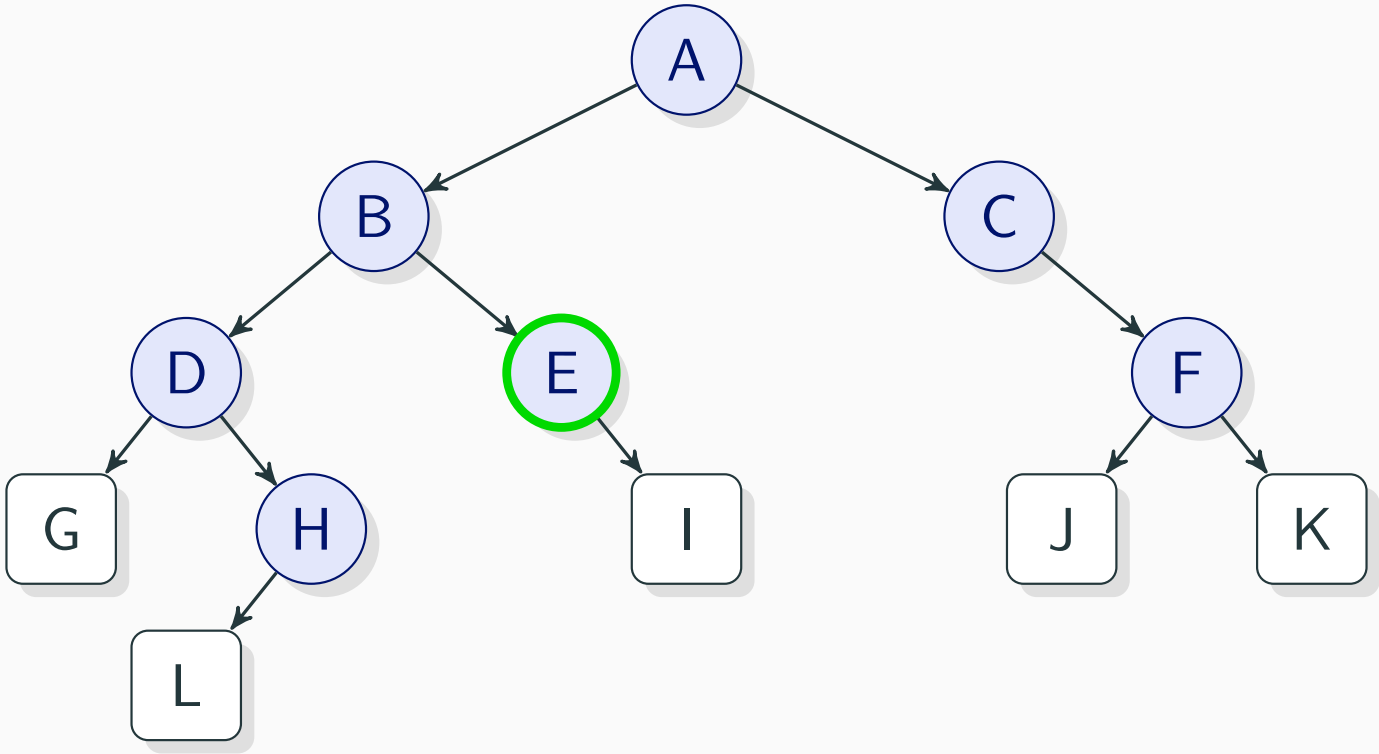
Les nœuds sont visités par profondeur croissante :



Parcours en largeur (BFS)

Il existe un autre type de parcours, en *largeur* (hiérarchique)

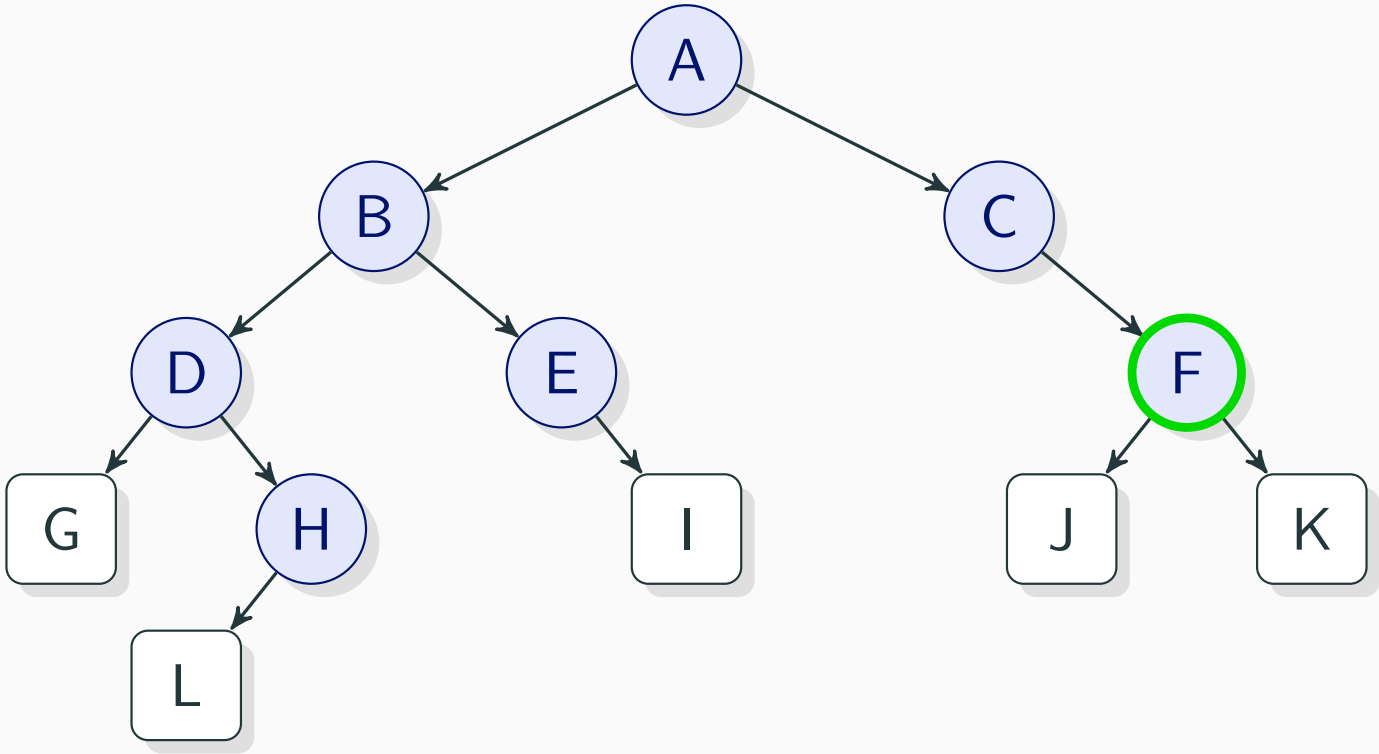
Les nœuds sont visités par profondeur croissante :



Parcours en largeur (BFS)

Il existe un autre type de parcours, en *largeur* (hiérarchique)

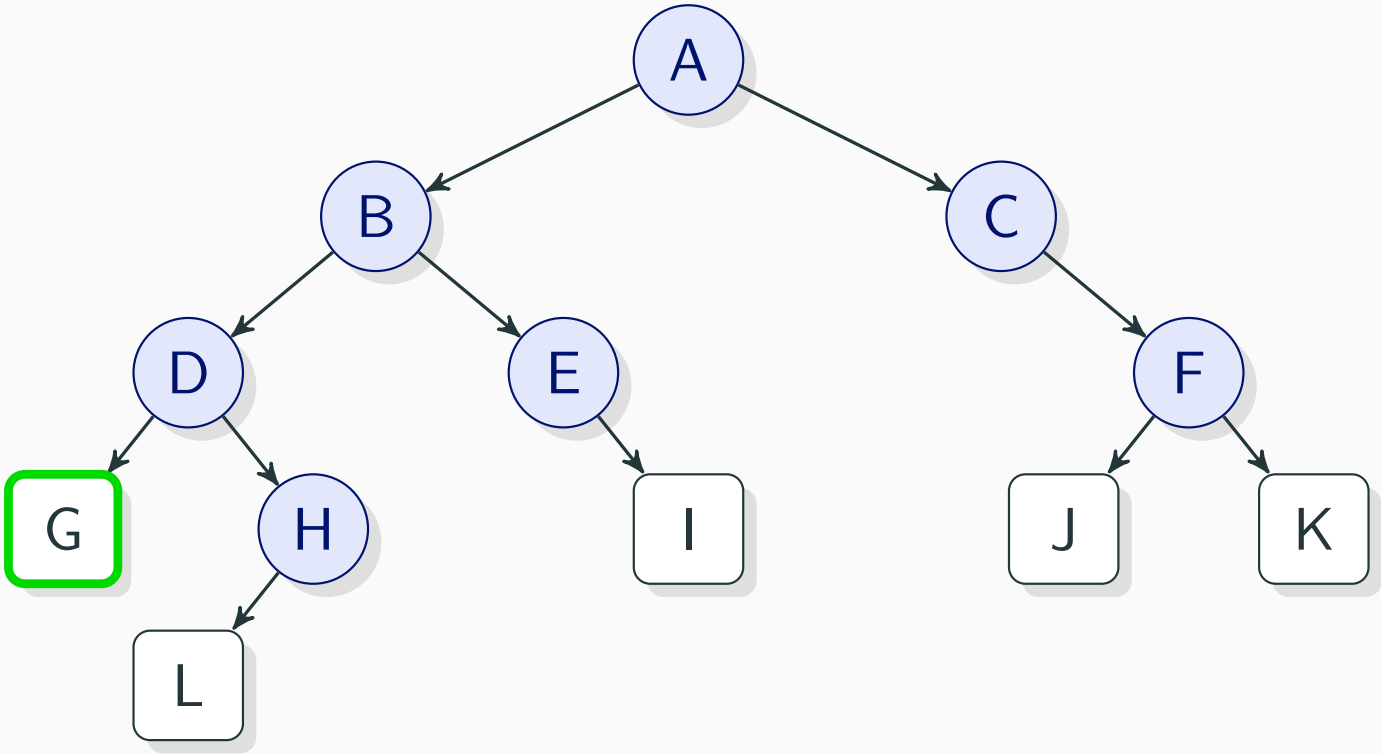
Les nœuds sont visités par profondeur croissante :



Parcours en largeur (BFS)

Il existe un autre type de parcours, en *largeur* (hiérarchique)

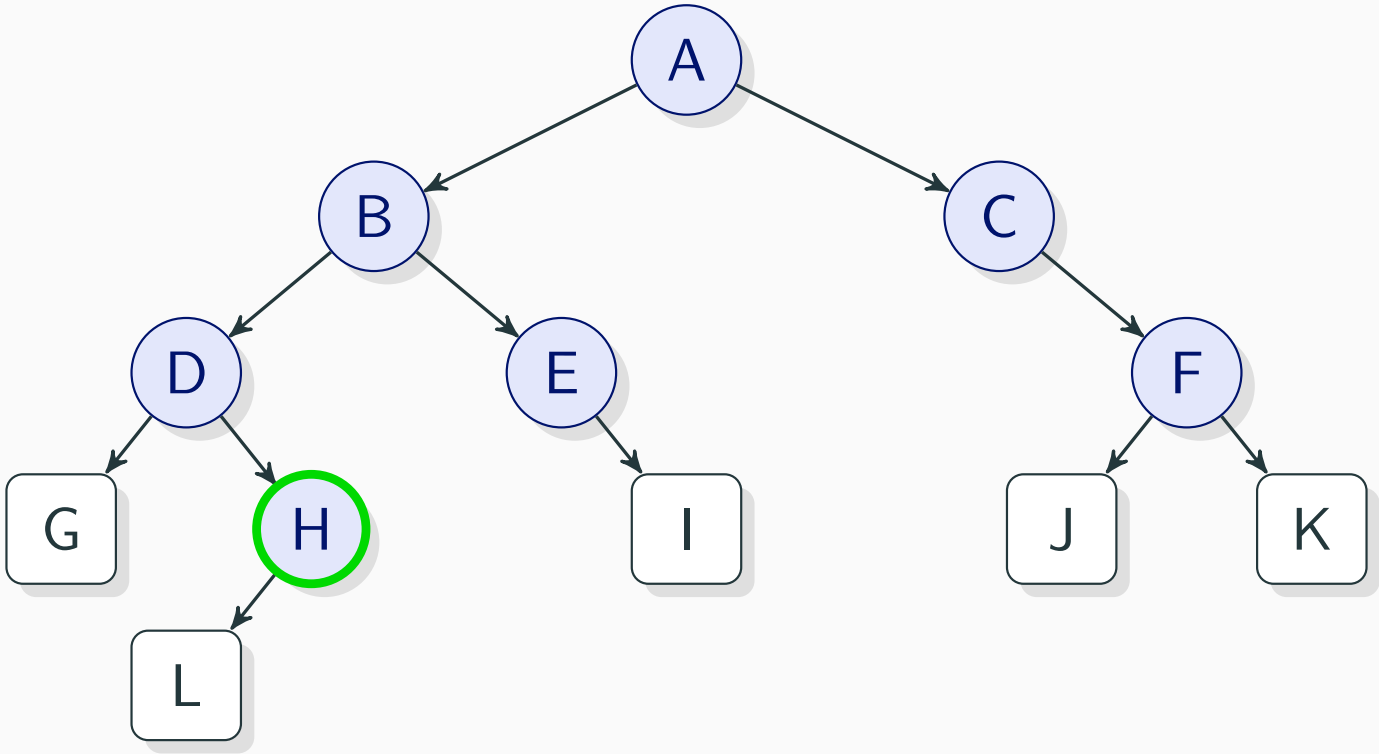
Les nœuds sont visités par profondeur croissante :



Parcours en largeur (BFS)

Il existe un autre type de parcours, en *largeur* (hiérarchique)

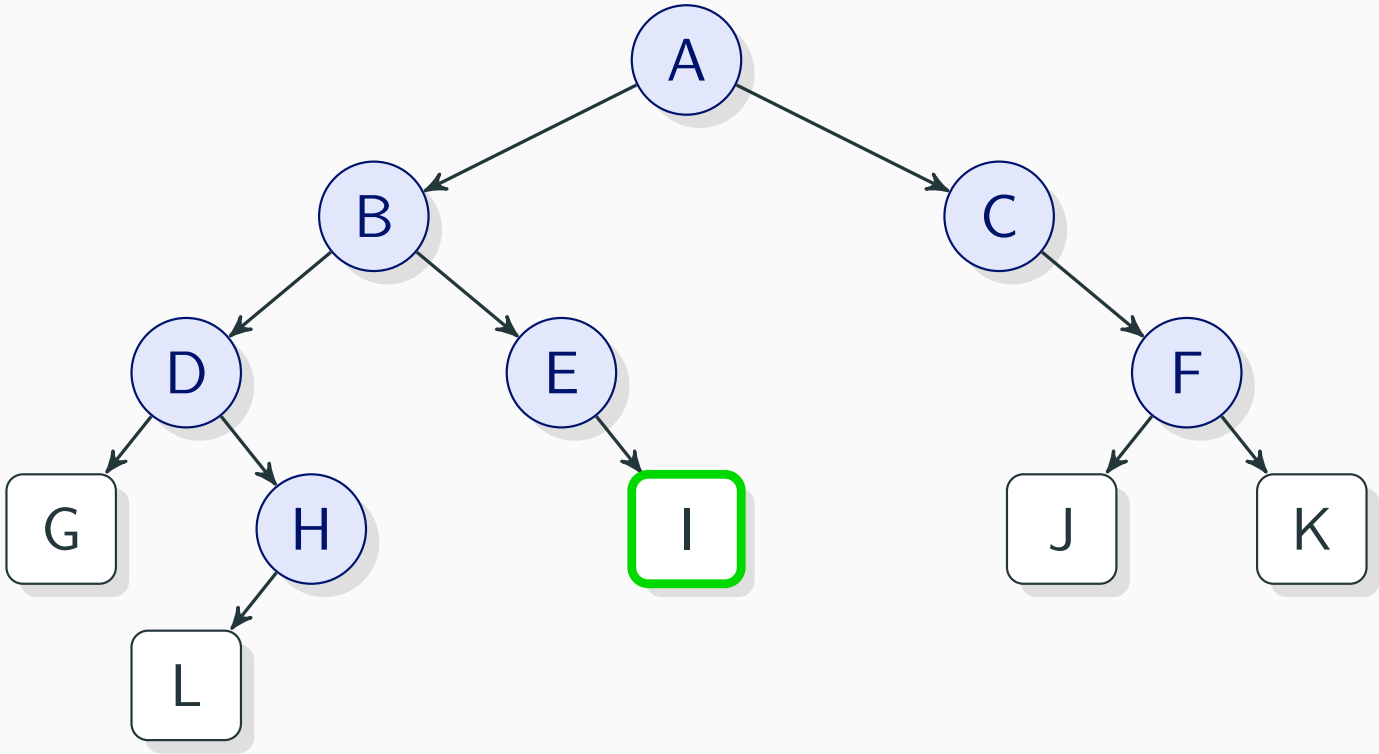
Les nœuds sont visités par profondeur croissante :



Parcours en largeur (BFS)

Il existe un autre type de parcours, en *largeur* (hiérarchique)

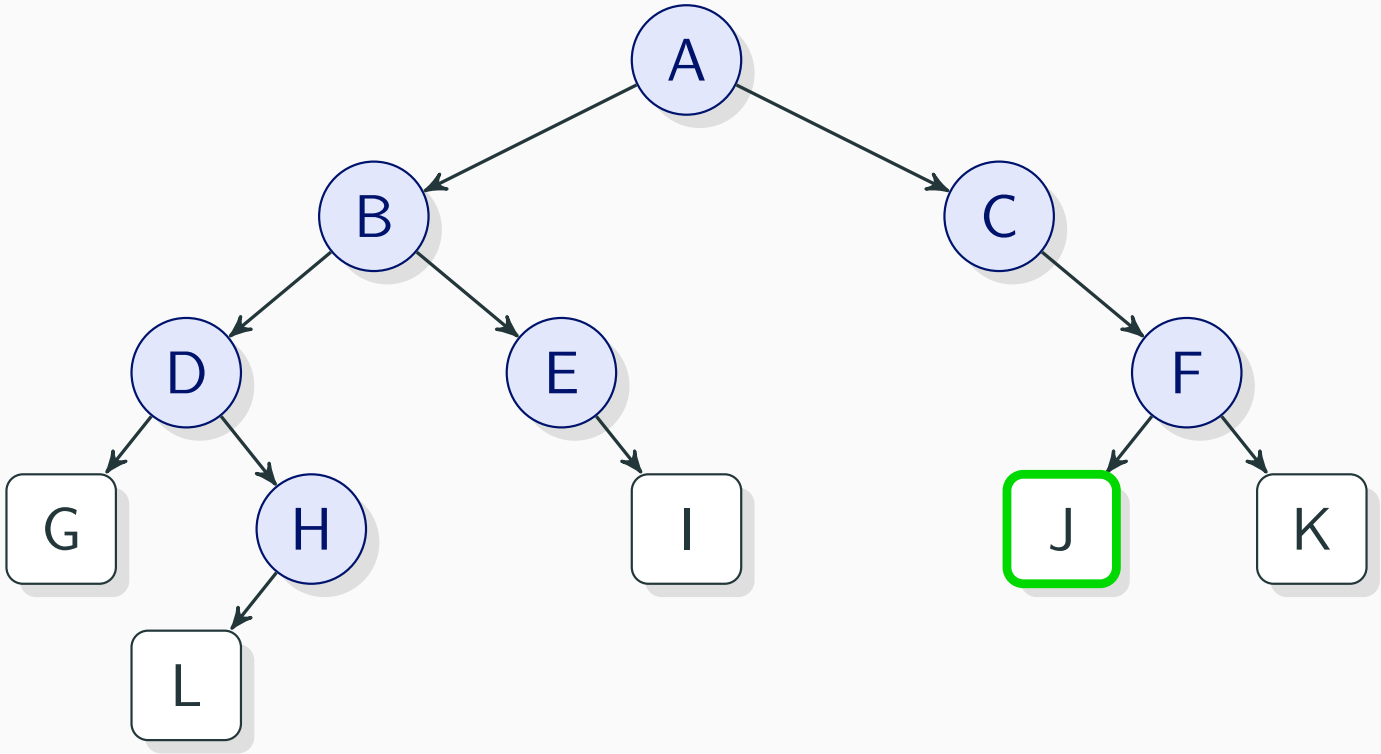
Les nœuds sont visités par profondeur croissante :



Parcours en largeur (BFS)

Il existe un autre type de parcours, en *largeur* (hiérarchique)

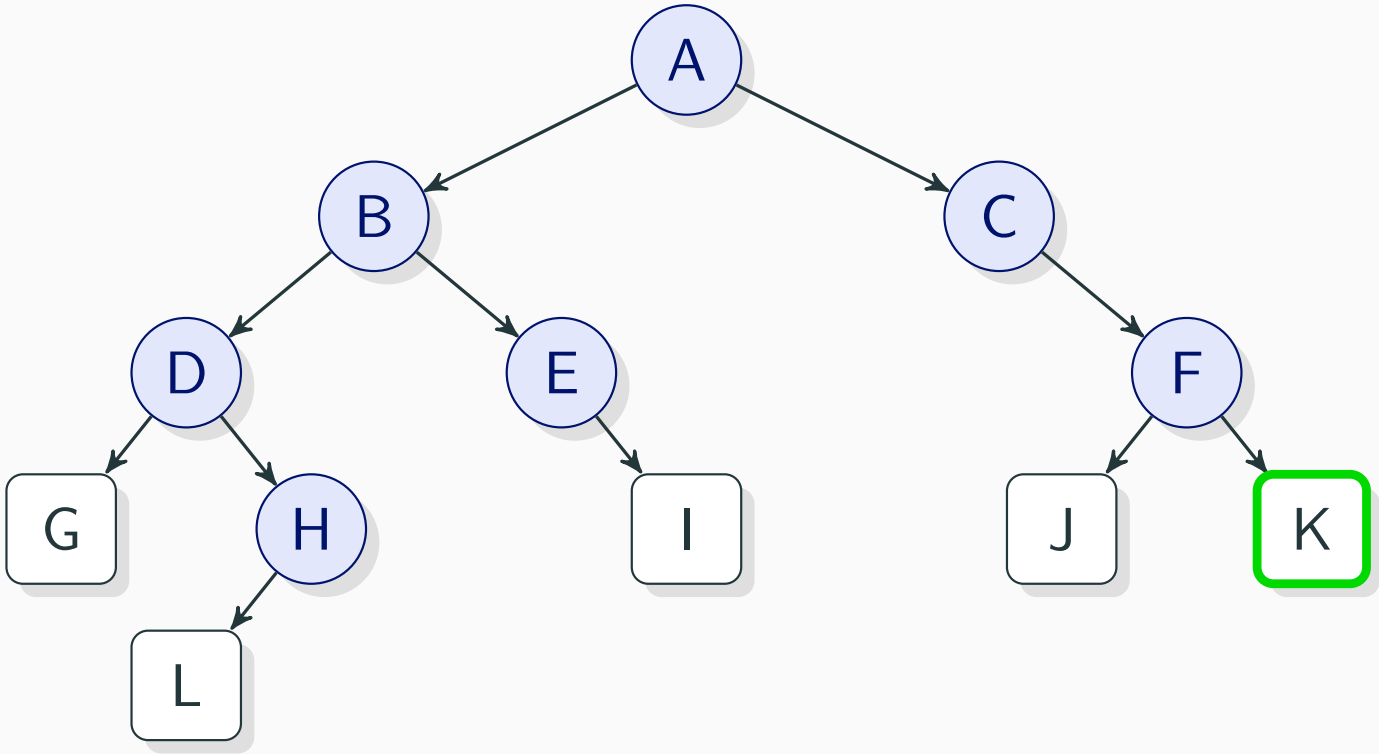
Les nœuds sont visités par profondeur croissante :



Parcours en largeur (BFS)

Il existe un autre type de parcours, en *largeur* (hiérarchique)

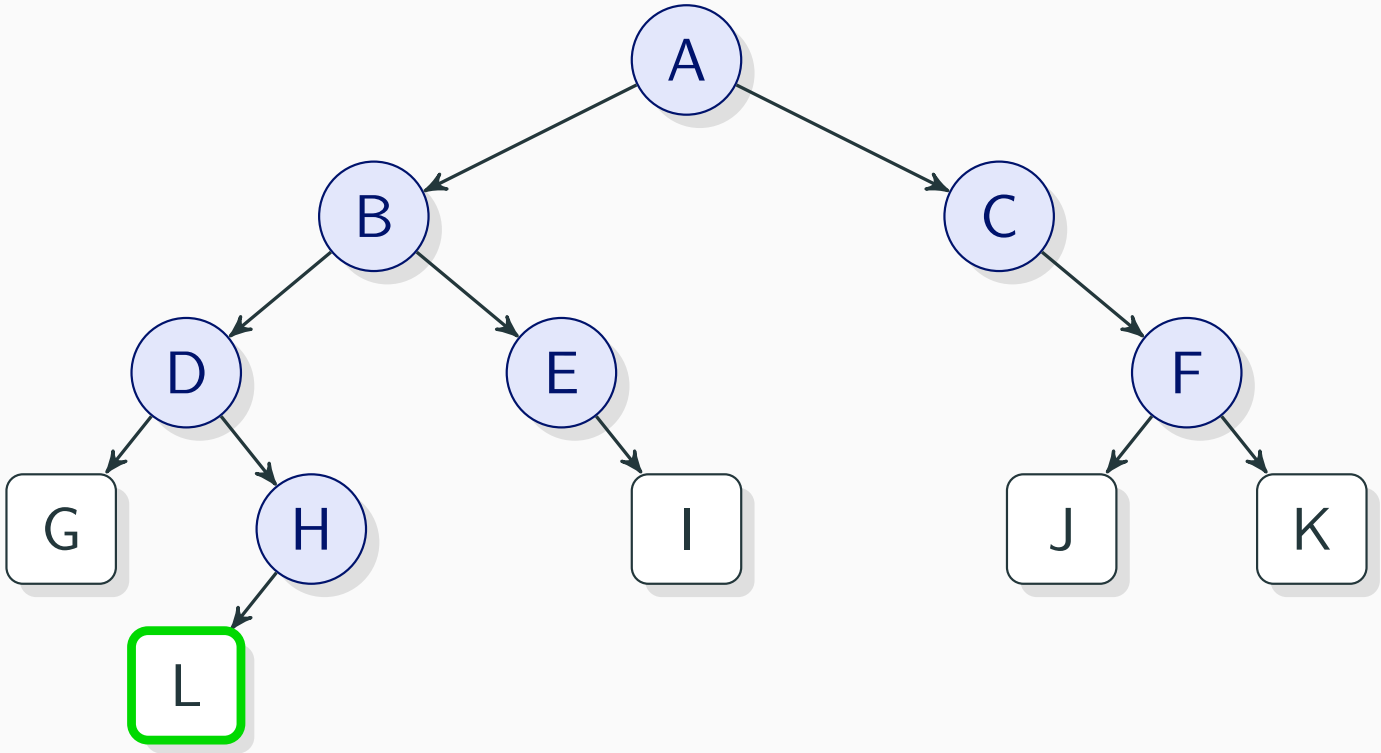
Les nœuds sont visités par profondeur croissante :



Parcours en largeur (BFS)

Il existe un autre type de parcours, en *largeur* (hiérarchique)

Les nœuds sont visités par profondeur croissante :



Parcours en largeur (BFS)


```
# let bfs f tree =
    let queue = Queue.create () in
      Queue.add tree queue;
      while not (Queue.is_empty queue) do
        match Queue.pop queue with
          | Node (e, lst)
              -> f e;
                 List.iter
                   (fun ch -> Queue.push ch queue)
                   lst
      done;;

val bfs : ('a -> unit) -> 'a tree -> unit = <fun>
```

Arbres binaires stricts

On impose très souvent des contraintes sur l'arité

Arbres binaires stricts

On impose très souvent des contraintes sur l'arité

Dans un *arbre binaire strict*, les nœuds internes sont d'arité 2

Généralement, on distingue un *enfant gauche* et un *enfant droit*

Arbres binaires stricts

On impose très souvent des contraintes sur l'arité

Dans un *arbre binaire strict*, les nœuds internes sont d'arité 2

Généralement, on distingue un *enfant gauche* et un *enfant droit*

On utilise généralement un type spécifique qui garantit la contrainte

Représentation des arbres binaires stricts

On peut les représenter par :

```
type 'a tree =
  | Node of 'a * 'a tree * 'a tree
  | Leaf of 'a
```

Représentation des arbres binaires stricts

On peut les représenter par :

```
type 'a tree =
  | Node of 'a * 'a tree * 'a tree
  | Leaf of 'a
```

Ou bien :

```
type 'a tree =
  | Node of 'a tree * 'a * 'a tree
  | Leaf of 'a
```

Représentation des arbres binaires stricts

On peut avoir un type distinct pour les feuilles :

```
type ('a, 'b) tree =
  | Node of 'a * ('a, 'b) tree * ('a, 'b) tree
  | Leaf of 'b
```

Utile par exemple pour un arbre d'expression mathématique :

- 'a désigne un opérateur binaire
- 'b désigne un opérande (valeur)

Calcul de la taille

Pour un arbre binaire strict défini par

```
type ('a, 'b) tree =
  | Node  of 'a * ('a, 'b) tree * ('a, 'b) tree
  | Leaf of 'b
```


La taille peut se calculer de la façon suivante :

Calcul de la taille

Pour un arbre binaire strict défini par

```
type ('a, 'b) tree =
  | Node  of 'a * ('a, 'b) tree * ('a, 'b) tree
  | Leaf of 'b
```


La taille peut se calculer de la façon suivante :

```
# let rec size = function
    | Node (_, lchild, rchild)
        -> 1 + size lchild + size rchild
    | Leaf _ -> 1;;

val size : ('a, 'b) tree -> int = <fun>
```

Calcul de la hauteur

Pour un arbre binaire strict défini par

```
type ('a, 'b) tree =
  | Node of 'a * ('a, 'b) tree * ('a, 'b) tree
  | Lead of 'b
```


La hauteur peut se calculer de la façon suivante :

Calcul de la hauteur

Pour un arbre binaire strict défini par

```
type ('a, 'b) tree =
  | Node of 'a * ('a, 'b) tree * ('a, 'b) tree
  | Lead of 'b
```

La hauteur peut se calculer de la façon suivante :

```
# let rec height = function
    | Node (_, lchild, rchild)
        -> 1 + max (height lchild) (height rchild)
    | Leaf _ -> 0;;

val height : ('a, 'b) tree -> int = <fun>
```

Calcul du nombre de feuilles

Pour un arbre binaire strict défini par

```
type ('a, 'b) tree =
  | Node of 'a * ('a, 'b) tree * ('a, 'b) tree
  | Leaf of 'b
```


Pour obtenir le nombre de feuilles, on aurait :

Calcul du nombre de feuilles

Pour un arbre binaire strict défini par

```
type ('a, 'b) tree =
  | Node of 'a * ('a, 'b) tree * ('a, 'b) tree
  | Leaf of 'b
```

Pour obtenir le nombre de feuilles, on aurait :

```
# let rec nb_leaves = function
    | Node (_, lchild, rchild)
        -> nb_leaves lchild + nb_leaves rchild
    | Leaf _ -> 1;;

val nb_leaves : ('a, 'b) tree -> int = <fun>
```

Calcul du nombre de nœuds internes

Pour un arbre binaire strict défini par

```
type ('a, 'b) tree =
  | Node of 'a * ('a, 'b) tree * ('a, 'b) tree
  | Leaf of 'b
```


Pour obtenir le nombre de nœuds *internes* :

Calcul du nombre de nœuds internes

Pour un arbre binaire strict défini par

```
type ('a, 'b) tree =
  | Node of 'a * ('a, 'b) tree * ('a, 'b) tree
  | Leaf of 'b
```

Pour obtenir le nombre de nœuds *internes* :

```
# let rec nb_nodes = function
    | Node (_, lchild, rchild)
        -> 1 + (nb_nodes lchild) + (nb_nodes rchild)
    | Leaf _ -> 0;;

val nb_nodes : ('a, 'b) tree -> int = <fun>
```

Détermination de la plus grande feuille

Pour un arbre binaire strict défini par

```
type ('a, 'b) tree =
  | Node of 'a * ('a, 'b) tree * ('a, 'b) tree
  | Leaf of 'b
```


Pour obtenir la plus grande feuille :

Détermination de la plus grande feuille

Pour un arbre binaire strict défini par

```
type ('a, 'b) tree =
  | Node of 'a * ('a, 'b) tree * ('a, 'b) tree
  | Leaf of 'b
```


Pour obtenir la plus grande feuille :

```
# let rec max_leaf = function
    | Node (_, lchild, rchild)
        -> max (max_leaf lchild) (max_leaf rchild)
    | Leaf v -> v;;

val max_leaf : ('a, 'b) tree -> 'b = <fun>
```

Preuves sur les arbres

Théorème

Si un arbre binaire strict possède n nœuds et f feuilles

alors $f = n + 1$

Preuves sur les arbres

Théorème

Si un arbre binaire strict possède n nœuds et f feuilles

alors $f = n + 1$

Vrai pour un arbre réduit à une feuille $(h = 0)$

car $n = 0$ et $f = 1$

Preuves sur les arbres

Théorème

Si un arbre binaire strict possède n nœuds et f feuilles
alors $f = n + 1$

Vrai pour un arbre réduit à une feuille ($h = 0$)

car $n = 0$ et $f = 1$

Récurrence forte sur la hauteur h (un peu lourd)

Définition par induction structurelle

Pour construire l'ensemble $\mathscr{A}^{*2}_{\mathrm{S}\,(\mathscr{E},\mathscr{F})}$ des arbres binaires stricts non vides étiquetés par $\mathscr{E}$ et $\mathscr{F}$:

- $\forall\, y \in \mathscr{F}$, la feuille $\big(y\big)$ est un arbre binaire strict ;
- $\forall\, x \in \mathscr{E}$ et $\forall \big(\mathrm{F}_g\,, \mathrm{F}_d\big) \in \mathscr{A}^{*2}_{\mathrm{S}\,(\mathscr{E},\mathscr{F})}$, l'arbre $\big(\mathrm{F}_g\,, x, \mathrm{F}_d\big)$ est un arbre binaire strict.

Preuves par induction structurelle

Théorème
Soit A *une assertion définie sur l'ensemble* $\mathscr{A}^{*}_{\mathrm{S}\ (\mathscr{E},\mathscr{F})}$ *des arbres binaires stricts non vides étiquetés par* $\mathscr{E}$ *et* $\mathscr{F}$*. Si*

- $\forall\, y \in \mathscr{F}$ *l'assertion* A *est vraie pour une feuille* (y) *;*
- $\forall\, x \in \mathscr{E}$ *et* $\forall (\mathrm{F}_g, \mathrm{F}_d) \in \mathscr{A}^{*\,2}_{\mathrm{S}\ (\mathscr{E},\mathscr{F})}$*, l'implication* $\big(\mathrm{A}(\mathrm{F}_g)$ *et* $\mathrm{A}(\mathrm{F}_d)\big) \Longrightarrow \mathrm{A}(\mathrm{F}_g, x, \mathrm{F}_d)$ *est vraie*

alors l'assertion $\mathrm{A}(\mathrm{A})$ *est vraie pour tout arbre* $\mathrm{A} \in \mathscr{A}^{*}_{\mathrm{S}\ (\mathscr{E},\mathscr{F})}$*.*

Preuves par induction structurelle

Théorème

Si un arbre binaire strict possède n nœuds et f feuilles
alors $f = n + 1$

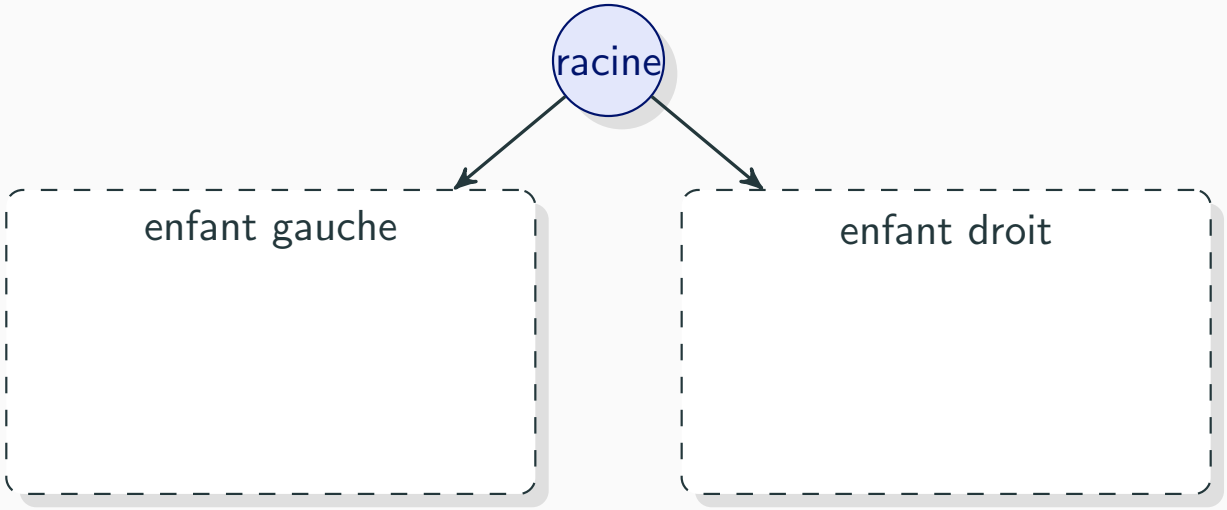
Pour un arbre de hauteur $h > 0$:

Preuves par induction structurelle

Théorème

Si un arbre binaire strict possède n nœuds et f feuilles
alors $f = n + 1$

Pour un arbre de hauteur $h > 0$:

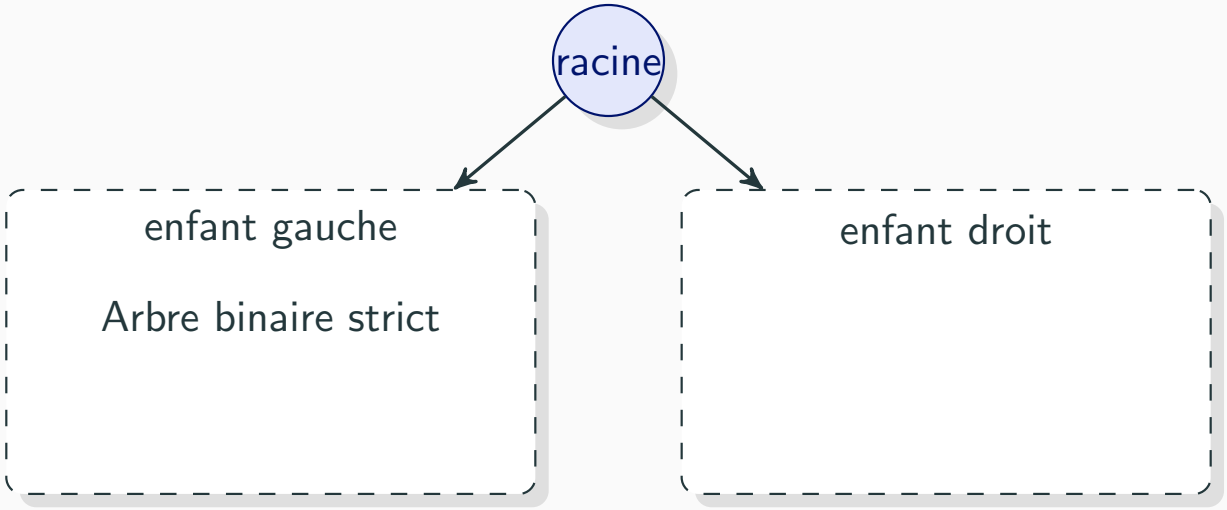


Preuves par induction structurelle

Théorème

Si un arbre binaire strict possède n nœuds et f feuilles
alors $f = n + 1$

Pour un arbre de hauteur $h > 0$:

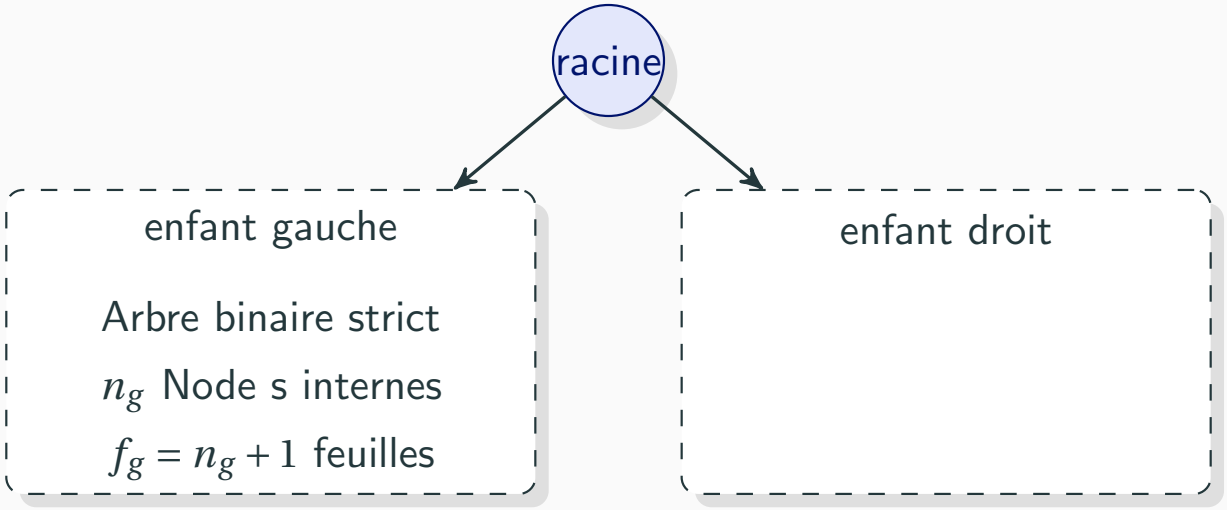


Preuves par induction structurelle

Théorème

Si un arbre binaire strict possède n nœuds et f feuilles
alors $f = n + 1$

Pour un arbre de hauteur $h > 0$:



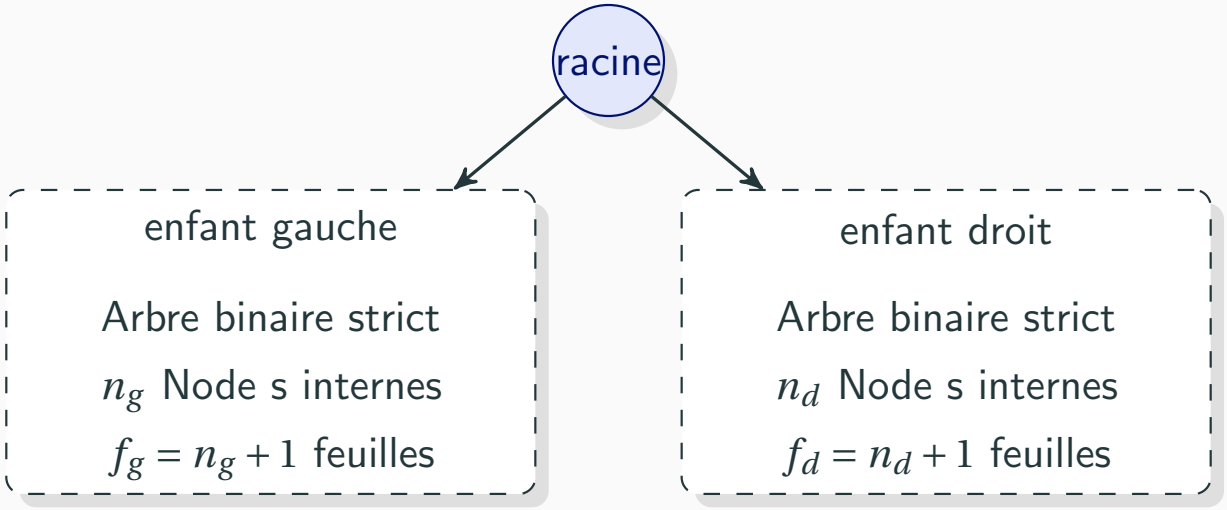
Preuves par induction structurelle

Théorème

Si un arbre binaire strict possède n nœuds et f feuilles

alors $f = n + 1$

Pour un arbre de hauteur $h > 0$:



Preuves par induction structurelle

Théorème

Si un arbre binaire strict possède n nœuds et f feuilles
alors $f = n + 1$

Au total, $\begin{cases} n = 1 + n_g + n_d \\ f = f_g + f_d \end{cases}$

Preuves par induction structurelle

Théorème

Si un arbre binaire strict possède n nœuds et f feuilles
alors $f = n + 1$

Au total, $\begin{cases} n = 1 + n_g + n_d \\ f = f_g + f_d \end{cases}$

Puisque $f_g = n_g + 1$ et $f_d = n_d + 1$, on a bien $f = n + 1$

La propriété est donc vraie pour tout arbre

Preuves par induction structurelle

Théorème

Si un arbre binaire strict possède n nœuds et f feuilles
alors $f = n + 1$

Parfois, on peut aller plus vite :

Preuves par induction structurelle

Théorème

Si un arbre binaire strict possède n nœuds et f feuilles
alors $f = n + 1$

Parfois, on peut aller plus vite :

- chaque nœud excepté la racine a un unique parent

Preuves par induction structurelle

Théorème

Si un arbre binaire strict possède n nœuds et f feuilles
alors $f = n + 1$

Parfois, on peut aller plus vite :

- chaque nœud excepté la racine a un unique parent
- il y a donc $n + f - 1$ liens de parenté

Preuves par induction structurelle

Théorème

Si un arbre binaire strict possède n nœuds et f feuilles
alors $f = n + 1$

Parfois, on peut aller plus vite :

- chaque nœud excepté la racine a un unique parent
- il y a donc $n + f - 1$ liens de parenté
- chaque nœud interne a deux enfants

Preuves par induction structurelle

Théorème

Si un arbre binaire strict possède n nœuds et f feuilles
alors $f = n + 1$

Parfois, on peut aller plus vite :

- chaque nœud excepté la racine a un unique parent
- il y a donc $n + f - 1$ liens de parenté
- chaque nœud interne a deux enfants
- donc $2n = n + f - 1$, soit $f = n + 1$

Arbres binaires (ou binaires-unaires)

Dans un *arbre binaire*, les nœuds internes sont d'arité 1 ou 2

Généralement, on distingue un *enfant gauche* et un *enfant droit*

Un Node peut avoir :

- deux enfants
- uniquement un enfant gauche
- uniquement un enfant droit
- aucun enfant (feuille)

Représentation des arbres binaires

On pourrait définir le type suivant :

```
type ('a, 'b) tree =
  | Node  of 'a * ('a, 'b) tree * ('a, 'b) tree
  | LNode of 'a * ('a, 'b) tree
  | RNode of 'a * ('a, 'b) tree
  | Leaf  of 'b
```


Inconvénient : quatre cas à filtrer dans les fonctions !

Représentation des arbres binaires

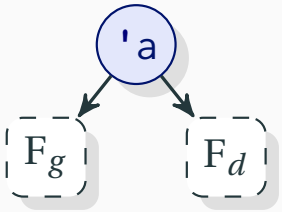
On peut réintroduire un « `Nil` » indiquant une absence

```
type ('a, 'b) tree =
  | Node of 'a * ('a, 'b) tree * ('a, 'b) tree
  | Leaf of 'b
  | Nil
```

Représentation des arbres binaires

On peut réintroduire un « `Nil` » indiquant une absence

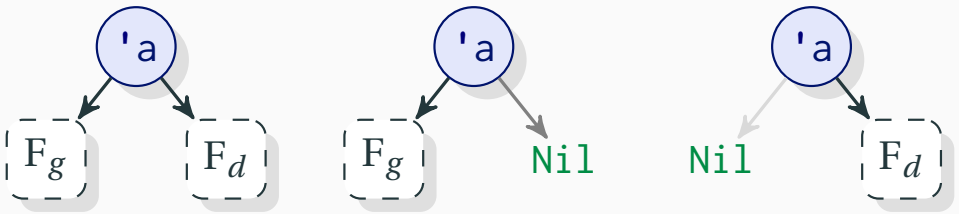
```
type ('a, 'b) tree =
  | Node of 'a * ('a, 'b) tree * ('a, 'b) tree
  | Leaf of 'b
  | Nil
```



Représentation des arbres binaires

On peut réintroduire un « `Nil` » indiquant une absence

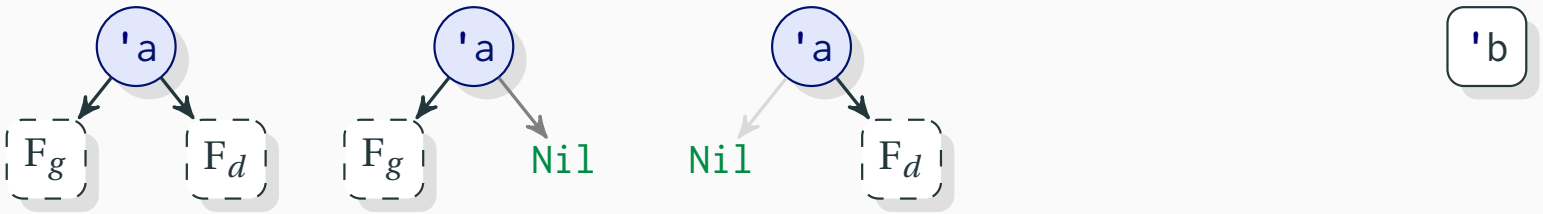
```
type ('a, 'b) tree =
  | Node of 'a * ('a, 'b) tree * ('a, 'b) tree
  | Leaf of 'b
  | Nil
```



Représentation des arbres binaires

On peut réintroduire un « `Nil` » indiquant une absence

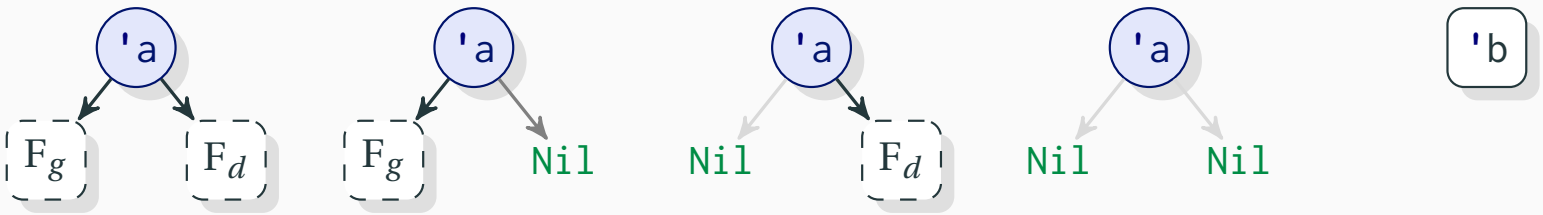
```
type ('a, 'b) tree =
  | Node of 'a * ('a, 'b) tree * ('a, 'b) tree
  | Leaf of 'b
  | Nil
```



Représentation des arbres binaires

On peut réintroduire un « `Nil` » indiquant une absence

```
type ('a, 'b) tree =
  | Node of 'a * ('a, 'b) tree * ('a, 'b) tree
  | Leaf of 'b
  | Nil
```



Il existe deux façons de représenter une feuille, c'est gênant !

Représentation des arbres binaires

Généralement, on utilisera le type suivant :

```
type 'a tree =
  | Node of 'a * 'a tree * 'a tree
  | Nil
```


Les feuilles sont des `Node (..., Nil, Nil)`

Représentation des arbres binaires

Généralement, on utilisera le type suivant :

```
type 'a tree =
  | Node of 'a * 'a tree * 'a tree
  | Nil
```


Les feuilles sont des `Node (..., Nil, Nil)`

On perd la possibilité de d'avoir des types distincts pour les feuilles

Représentation des arbres binaires

Généralement, on utilisera le type suivant :

```
type 'a tree =
  | Node of 'a * 'a tree * 'a tree
  | Nil
```


Les feuilles sont des `Node (..., Nil, Nil)`

On perd la possibilité de d'avoir des types distincts pour les feuilles

Un arbre peut être *vide* (`Nil`)

Manipulation des arbres binaires

On les traite par filtrage :

```
let foo = function                                          
  | Nil -> ...                          (* arbre vide    *)
  | Node (e, Nil, Nil) -> ...           (* feuille       *)
  | Node (e, lchild, Nil) -> ...        (* enfant gauche *)
  | Node (e, Nil, rchild) -> ...        (* enfant droit  *)
  | Node (e, lchild, rchild) -> ... (* deux enfants  *)
```

Certains cas peuvent être fusionnés/décomposés

Calcul du plus grand élément

On examine chaque cas :


```
# let rec max_tree = function
    | Nil -> failwith "Arbre vide"
    | Node (e, Nil, Nil) -> v
    | Node (e, lchild, Nil) -> max v (max_tree lchild)
    | Node (e, Nil, rchild) -> max v (max_tree rchild)
    | Node (e, lchild, rchild)
        -> max v (max (max_tree lchild)
                      (max_tree rchild))

val max_tree : 'a tree -> 'a = <fun>
```

Calcul du nombre de feuilles

Pour déterminer le nombre de feuilles, trois cas à considérer :


```
# let rec nb_leaves = function
    | Nil  -> 0                        (* arbre vide    *)
    | Node (_, Nil,   Nil) -> 1        (* feuille       *)
    | Node (_, lchild, rchild)         (* noeud interne *)
        -> (nb_leaves lchild) + (nb_leaves rchild);;

val nb_leaves : 'a tree -> int = <fun>
```

Calcul du nombre de feuilles

Pour déterminer le nombre de feuilles, trois cas à considérer :


```
# let rec nb_leaves = function
    | Nil  -> 0                        (* arbre vide    *)
    | Node (_, Nil,   Nil) -> 1        (* feuille       *)
    | Node (_, lchild, rchild)         (* noeud interne *)
        -> (nb_leaves lchild) + (nb_leaves rchild);;

val nb_leaves : 'a tree -> int = <fun>
```



```
# let rec nb_leaves = function
    | Nil  -> 0
    | Node (_, lchild, rchild)
        -> max 1 ((nb_leaves lchild) + (nb_leaves rchild));;

val nb_leaves : 'a tree -> int = <fun>
```

Définition par induction structurelle

- l'arbre vide `Nil` est un arbre binaire ;
- $\forall x \in \mathscr{E}$ et $\forall (\mathrm{F}_g, \mathrm{F}_d) \in \mathscr{A}_{\mathrm{B}\mathscr{E}}^2$, l'arbre $(\mathrm{F}_g, x, \mathrm{F}_d)$ est un arbre binaire.

Théorème
Soit A *une assertion définie sur l'ensemble* $\mathscr{A}_{\mathrm{B}\mathscr{E}}$ *des arbres binaires étiquetés par* $\mathscr{E}$*. Si*

- *l'assertion* A *est vraie pour un arbre vide ;*
- $\forall x \in \mathscr{E}$ *et* $\forall (\mathrm{F}_g, \mathrm{F}_d) \in \mathscr{A}_{\mathrm{B}\mathscr{E}}^2$*, l'implication* $(\mathrm{A}(\mathrm{F}_g)$ *et* $\mathrm{A}(\mathrm{F}_d)) \Longrightarrow \mathrm{A}(\mathrm{F}_g, x, \mathrm{F}_d)$ *est vraie*

alors l'assertion A *est vraie pour tout arbre* $\mathrm{A} \in \mathscr{A}_{\mathrm{B}\mathscr{E}}$*.*

Preuves sur les arbres

Théorème
Pour un arbre binaire $\mathrm{A} \in \mathscr{A}_{\mathrm{B}}$ *quelconque, on a*

$$\mathrm{h}(\mathrm{A}) + 1 \leqslant |\mathrm{A}| \leqslant 2^{\mathrm{h}(\mathrm{A})+1} - 1.$$

Preuves sur les arbres

Théorème
Pour tout arbre binaire $\mathrm{A} \in \mathscr{A}_{\mathrm{B}}$*, le nombre de feuilles* f *et le nombre de nœuds internes* n *vérifient* $f \leqslant n+1$*.*

Parfois il faut traiter les feuilles à part !

Propriétés par induction structurelle

Fonctions de $\mathscr{A}_{\mathrm{B}\mathscr{E}} \to \mathscr{V}$

On choisit $\alpha \in \mathscr{V}$ et $\phi : \mathscr{V} \times \mathscr{E} \times \mathscr{V} \mapsto \mathscr{V}$

- $f(\texttt{Nil}) = \alpha$;
- $\forall x \in \mathscr{E}$ et $\forall \mathrm{F}_g, \mathrm{F}_d \in \mathscr{A}_{\mathrm{B}\mathscr{E}}^2,\ f\big(\mathrm{F}_g, x, \mathrm{F}_d\big) = \phi\big(f\big(\mathrm{F}_g\big), x, f\big(\mathrm{F}_d\big)\big)$.

Calcul de la taille

- $|\texttt{Nil}| = 0$;
- si $\mathrm{A} = (\mathrm{F}_g, x, \mathrm{F}_d)$, alors $|\mathrm{A}| = 1 + |\mathrm{F}_g| + |\mathrm{F}_d|$.

```
# let rec size = function                                    
    | Nil  -> 0                          (* arbre vide    *)
    | Node (_, lchild, rchild)    (* Node          *)
        -> 1 + (size lchild) + (size rchild);;

val size : 'a tree -> int = <fun>
```

Calcul de la hauteur

- $\mathrm{h}(\texttt{Nil}) = -1$;
- si $\mathrm{A} = \left(\mathrm{F}_g, x, \mathrm{F}_d\right)$, alors $\mathrm{h}(\mathrm{A}) = 1 + \max\left(\mathrm{h}\left(\mathrm{F}_g\right), \mathrm{h}\left(\mathrm{F}_d\right)\right)$.

Dans ce cas :

```
# let rec height = function
    | Nil  -> -1                    (* arbre vide    *)
    | Node (_, lchild, rchild)      (* Node          *)
        -> 1 + max (height lchild) (height rchild);;

val height : 'a tree -> int = <fun>
```

Calcul de la hauteur

Parfois, on doit traiter à part des cas particuliers, par exemple les feuilles :

```
# let rec nb_leaves = function
    | Nil -> 0                          (* arbre vide *)
    | Node (_, Nil, Nil) -> 1           (* feuille    *)
    | Node (_, lchild, rchild)
        -> nb_leaves lchild + nb_leaves rchild;;

val nb_leaves : 'a tree -> int = <fun>
```

Parcours d'un arbre

Attention, dans une fonction telle que :

```
# let rec hauteur = function                                    
    | Nil  -> -1                          (* arbre vide    *)
    | Node (_, lchild, rchild)    (* Node          *)
        -> 1 + max (hauteur lchild) (hauteur rchild);;

val hauteur : 'a tree -> int = <fun>
```

On a bien un parcours en profondeur

Parcours d'un arbre

Attention, dans une fonction telle que :

```
# let rec hauteur = function                                    
    | Nil  -> -1                          (* arbre vide    *)
    | Node (_, lchild, rchild)    (* Node          *)
        -> 1 + max (hauteur lchild) (hauteur rchild);;

val hauteur : 'a tree -> int = <fun>
```

On a bien un parcours en profondeur

Mais on ne sait pas si l'enfant gauche est traité avant le droit !

Parcours d'un arbre

Attention, dans une fonction telle que :

```
# let rec hauteur = function                                    
    | Nil  -> -1                          (* arbre vide    *)
    | Node (_, lchild, rchild)    (* Node          *)
        -> 1 + max (hauteur lchild) (hauteur rchild);;

val hauteur : 'a tree -> int = <fun>
```

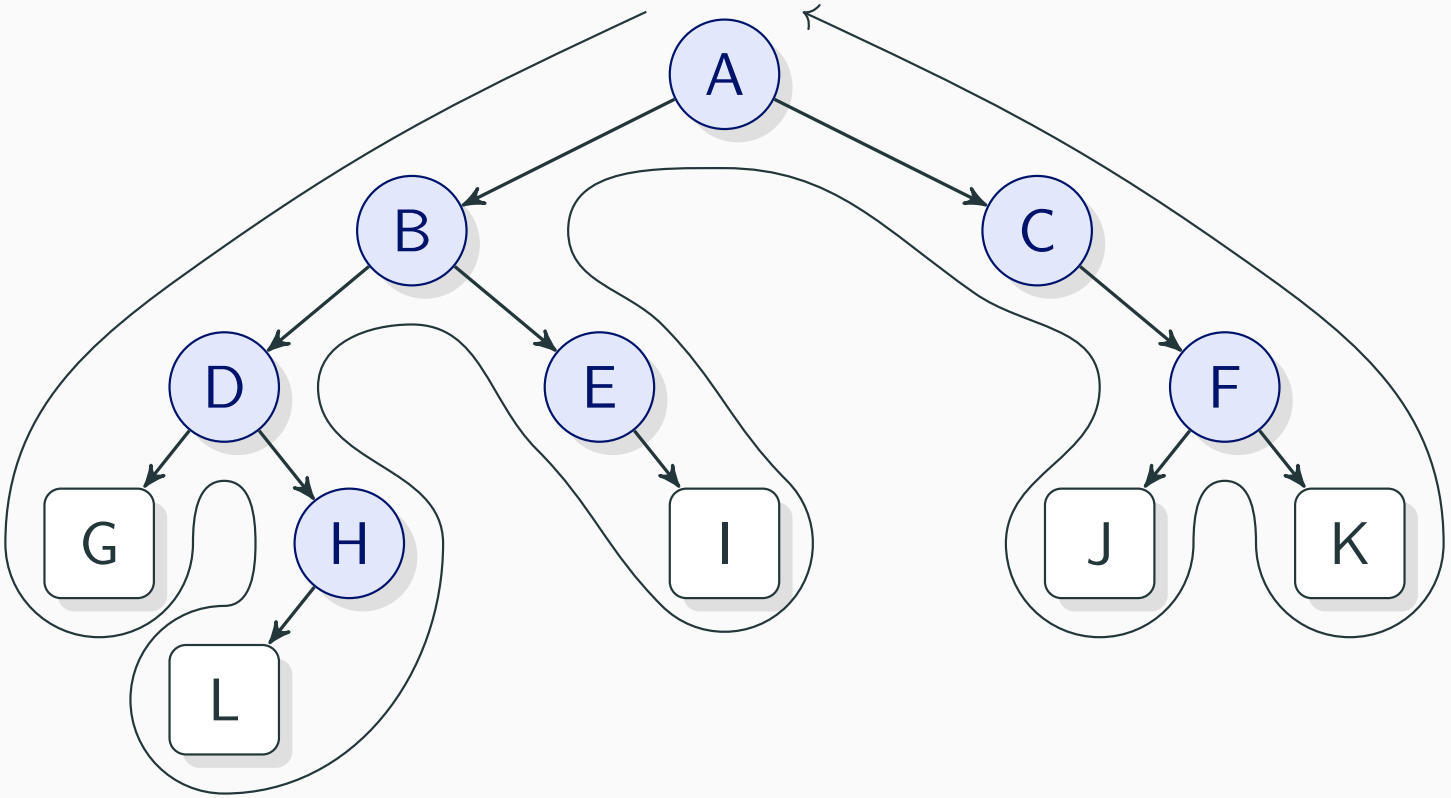
On a bien un parcours en profondeur

Mais on ne sait pas si l'enfant gauche est traité avant le droit !

En revanche, un sous-arbre est traité en *intégralité* avant l'autre

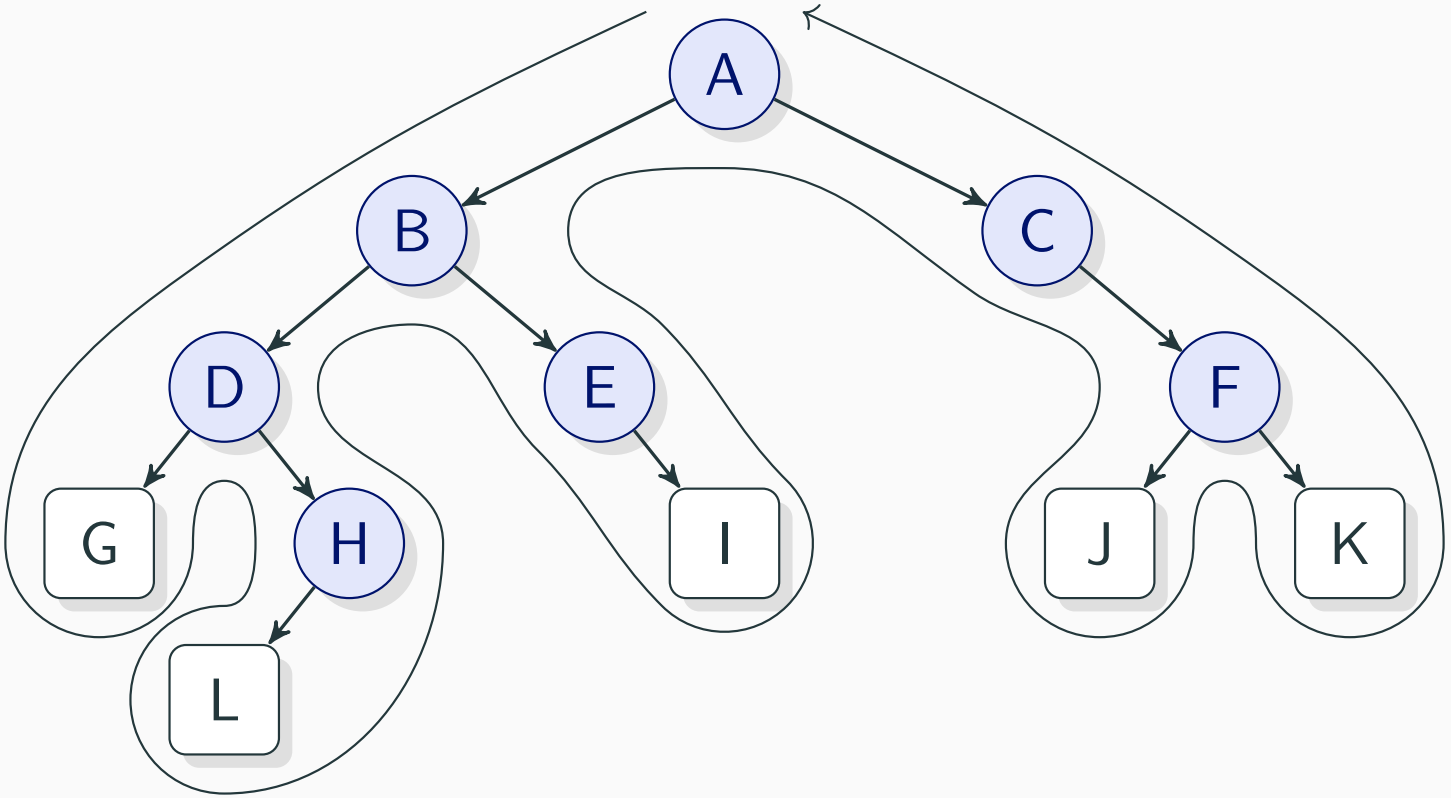
Parcours d'un arbre

Dans un parcours en profondeur d'un arbre *binaire* :



Parcours d'un arbre

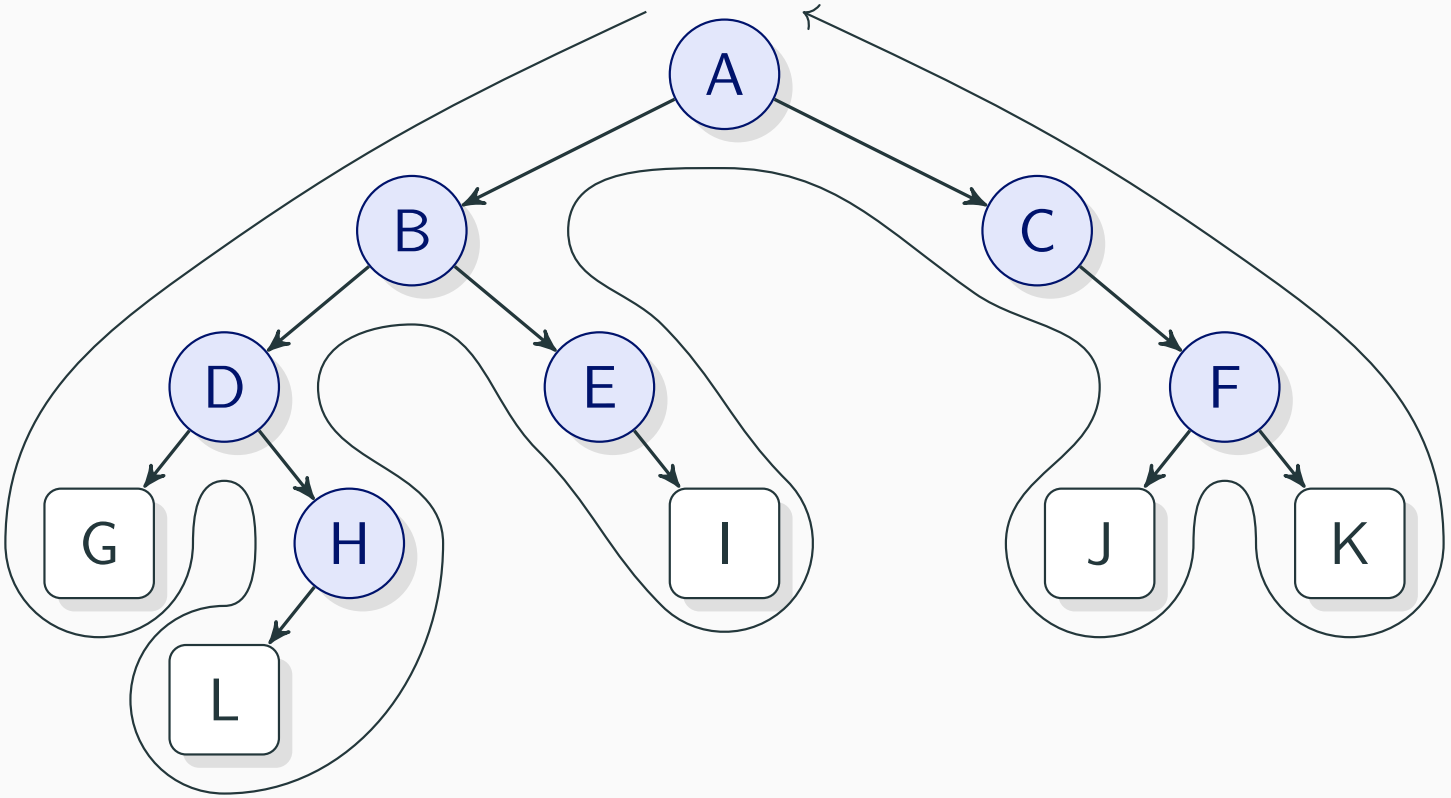
Dans un parcours en profondeur d'un arbre *binaire* :



- les feuilles sont visitées 1 fois

Parcours d'un arbre

Dans un parcours en profondeur d'un arbre *binaire* :



- les feuilles sont visitées 1 fois
- les nœuds 2 fois (un enfant) ou 3 fois (deux enfants)

Parcours préfixe

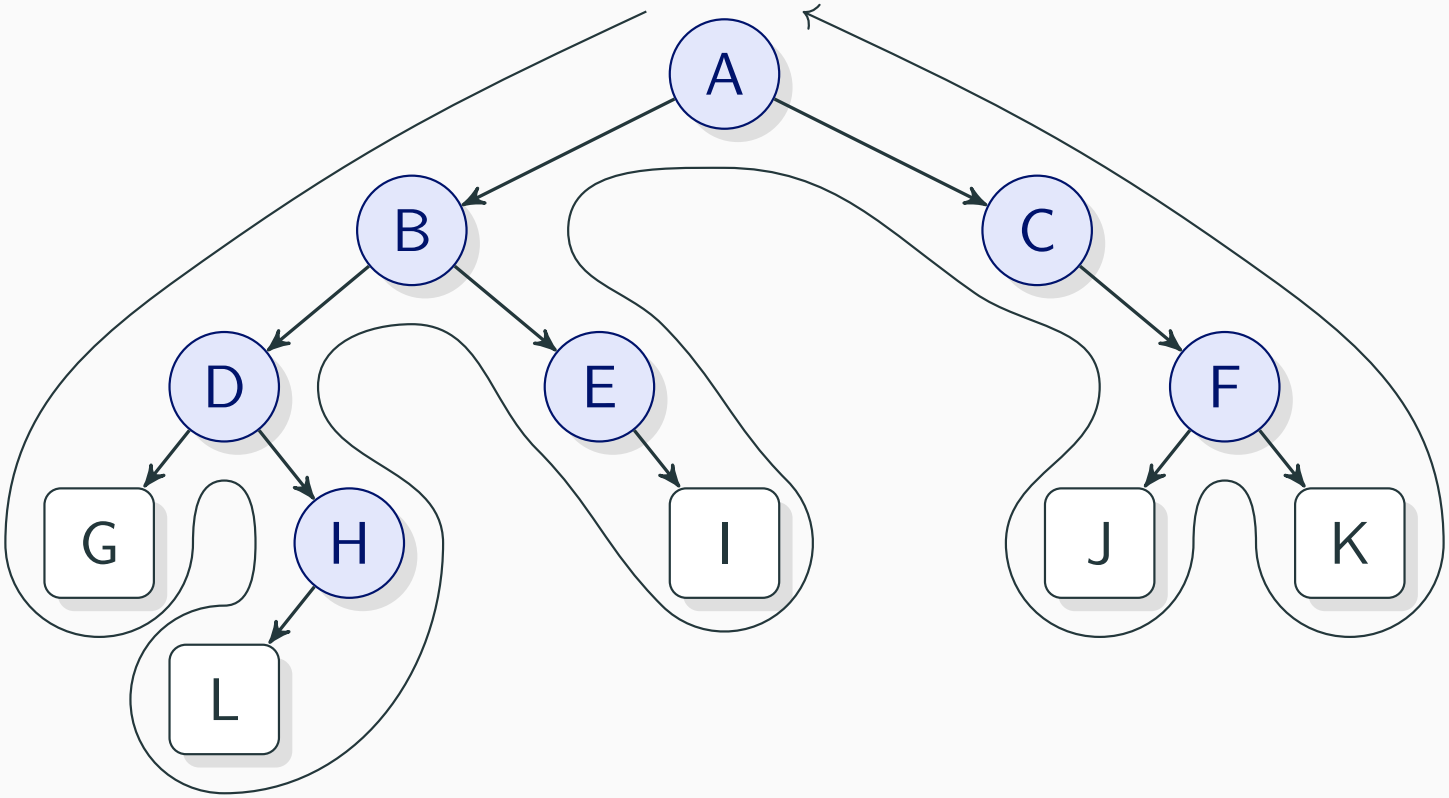
Dans un parcours en profondeur *préfixe*,

un nœud est traité *avant* ses enfants (première visite)


```
# let rec dfs_prefix f = function
    | Nil -> ()
    | Node (e, lchild, rchild)
        -> f e;
           dfs_prefix f lchild;
           dfs_prefix f rchild;;

val dfs_prefix : ('a -> 'b) -> 'a arbre -> unit = <fun>
```

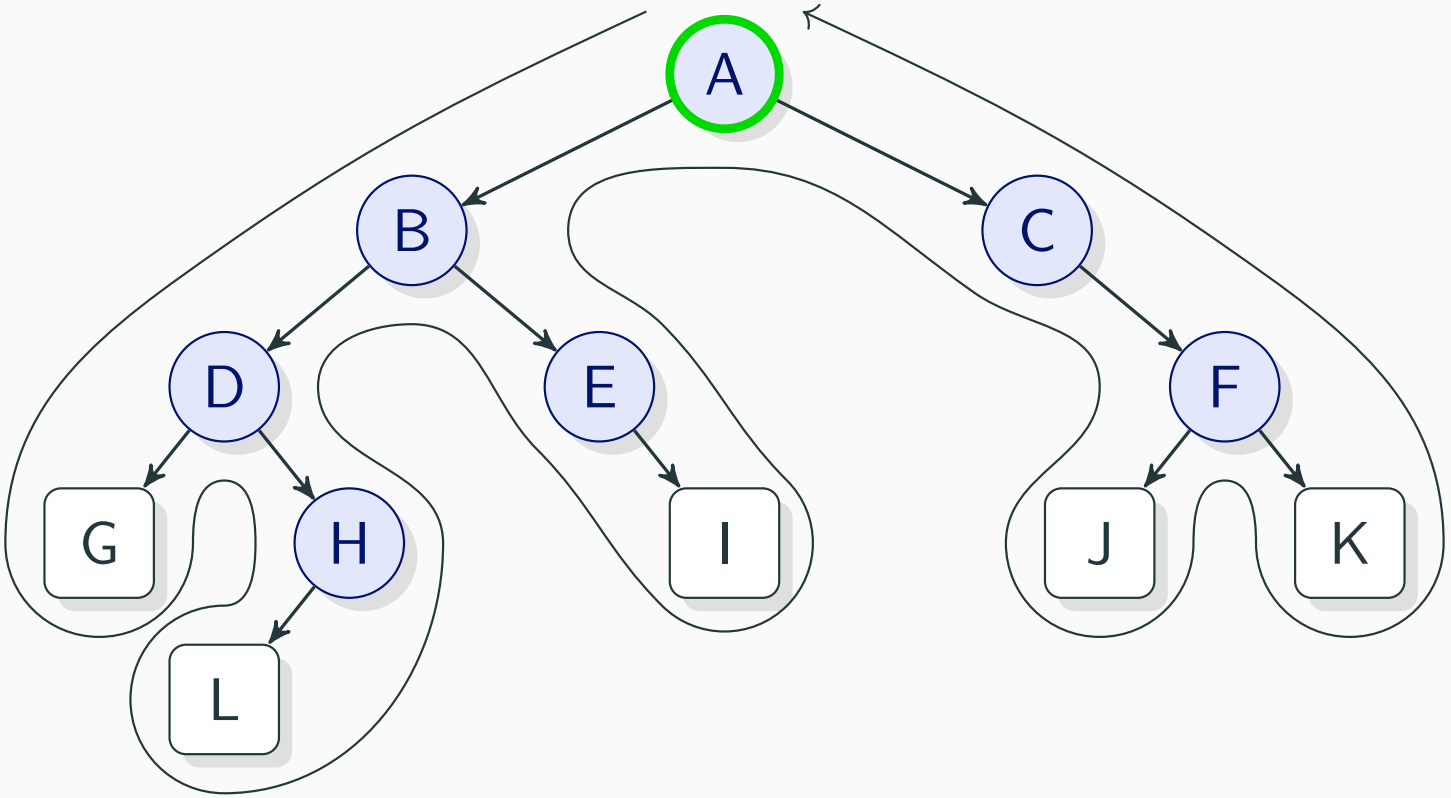
Parcours préfixe

L'exécution de `dfs_prefix` se déroule de la façon suivante :



Parcours préfixe

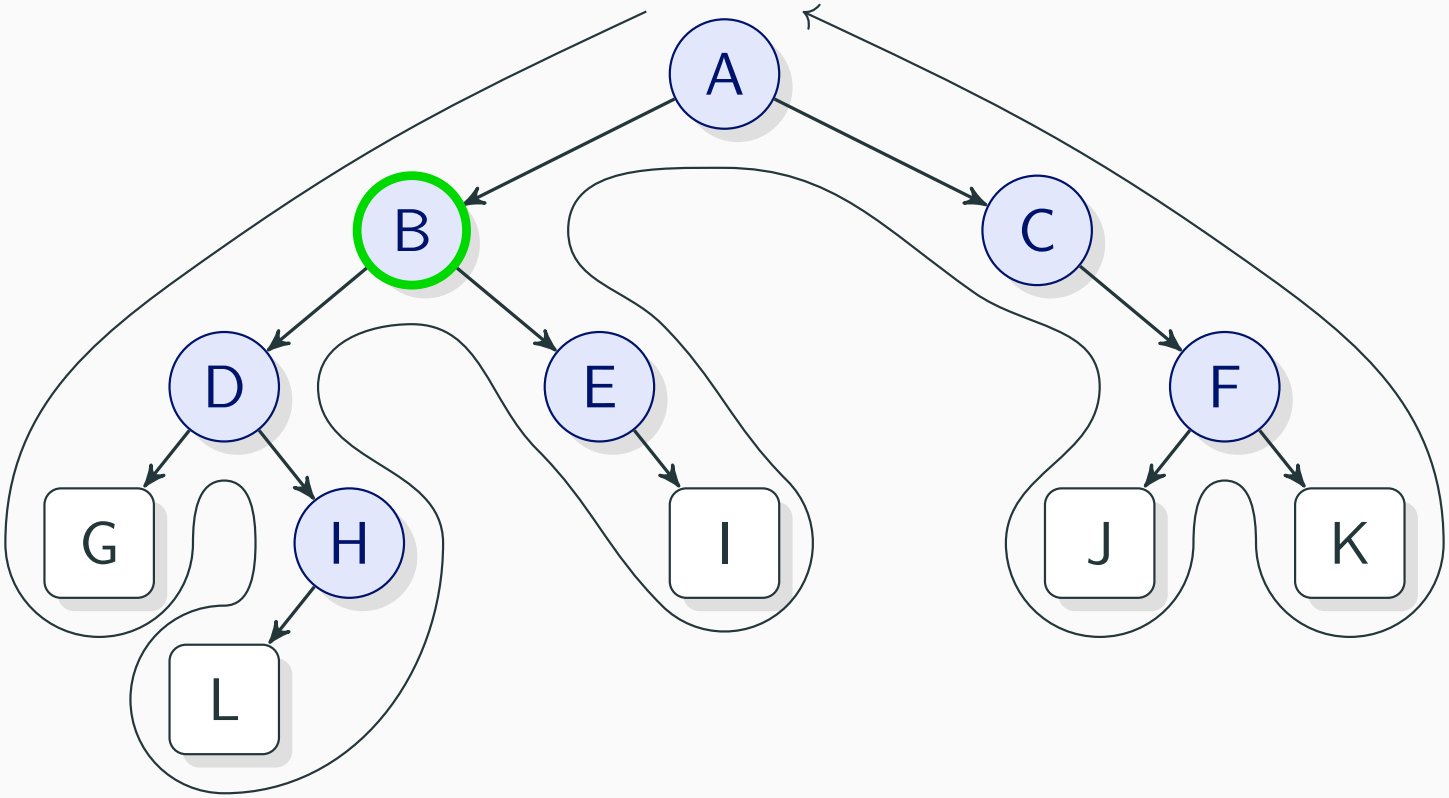
L'exécution de `dfs_prefix` se déroule de la façon suivante :



A

Parcours préfixe

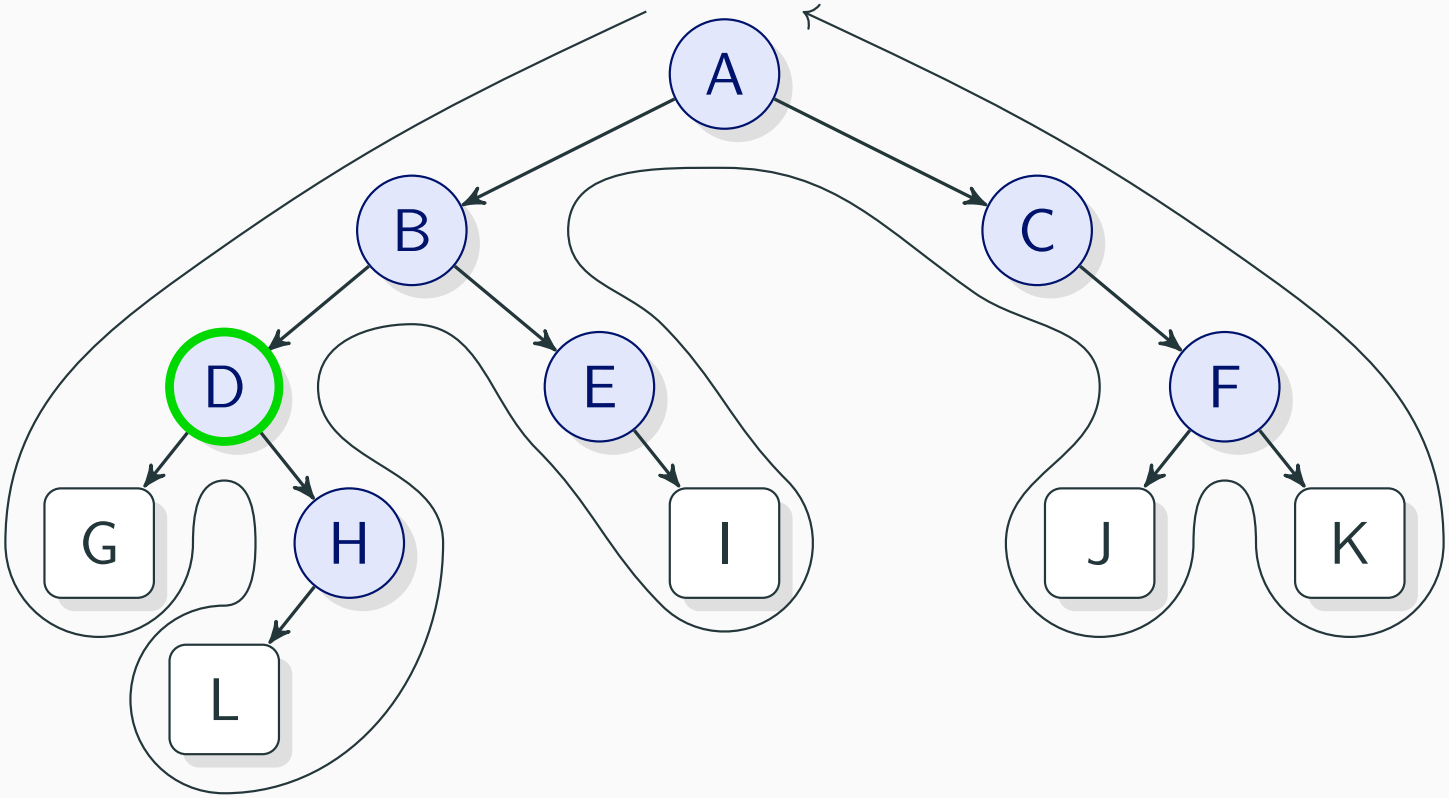
L'exécution de `dfs_prefix` se déroule de la façon suivante :



AB

Parcours préfixe

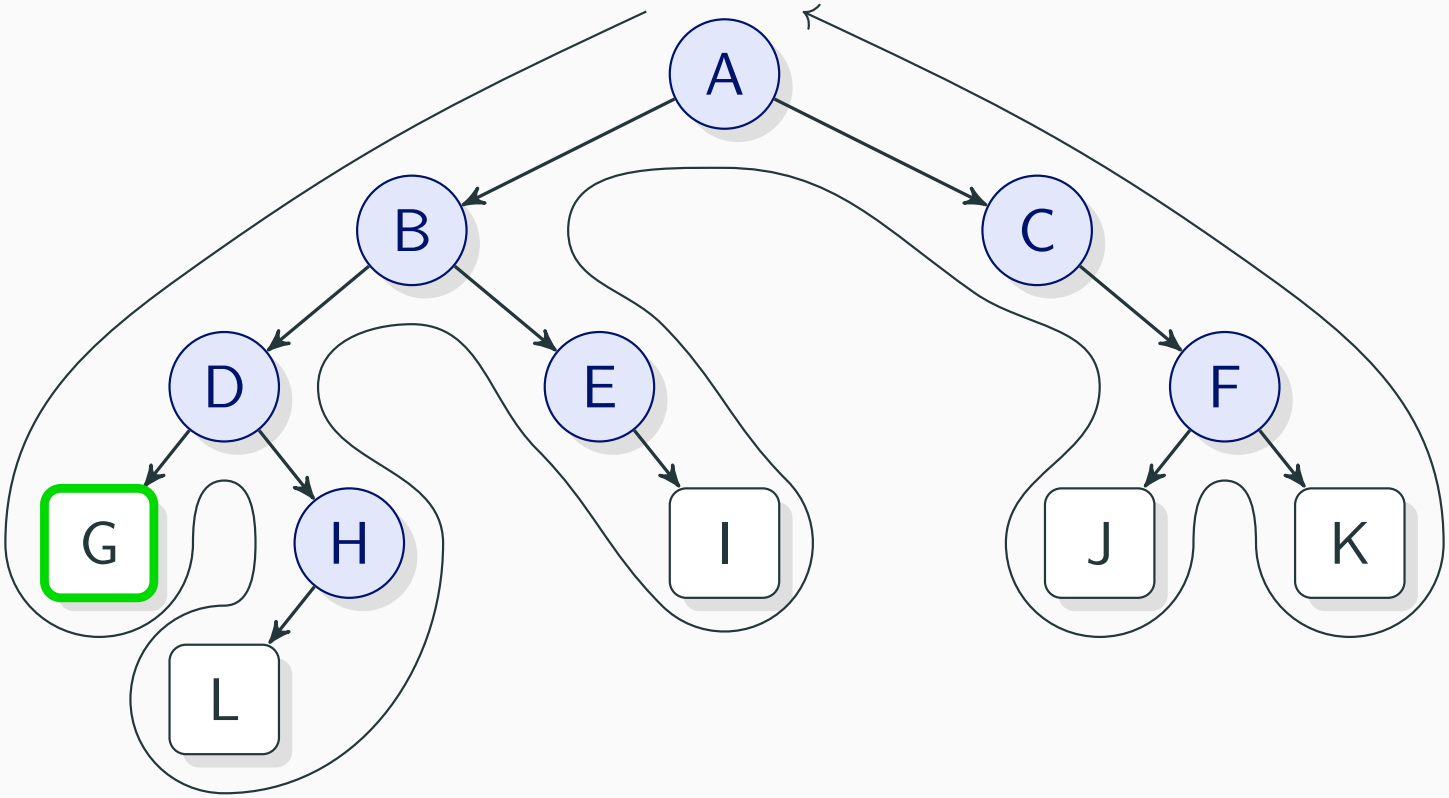
L'exécution de `dfs_prefix` se déroule de la façon suivante :



ABD

Parcours préfixe

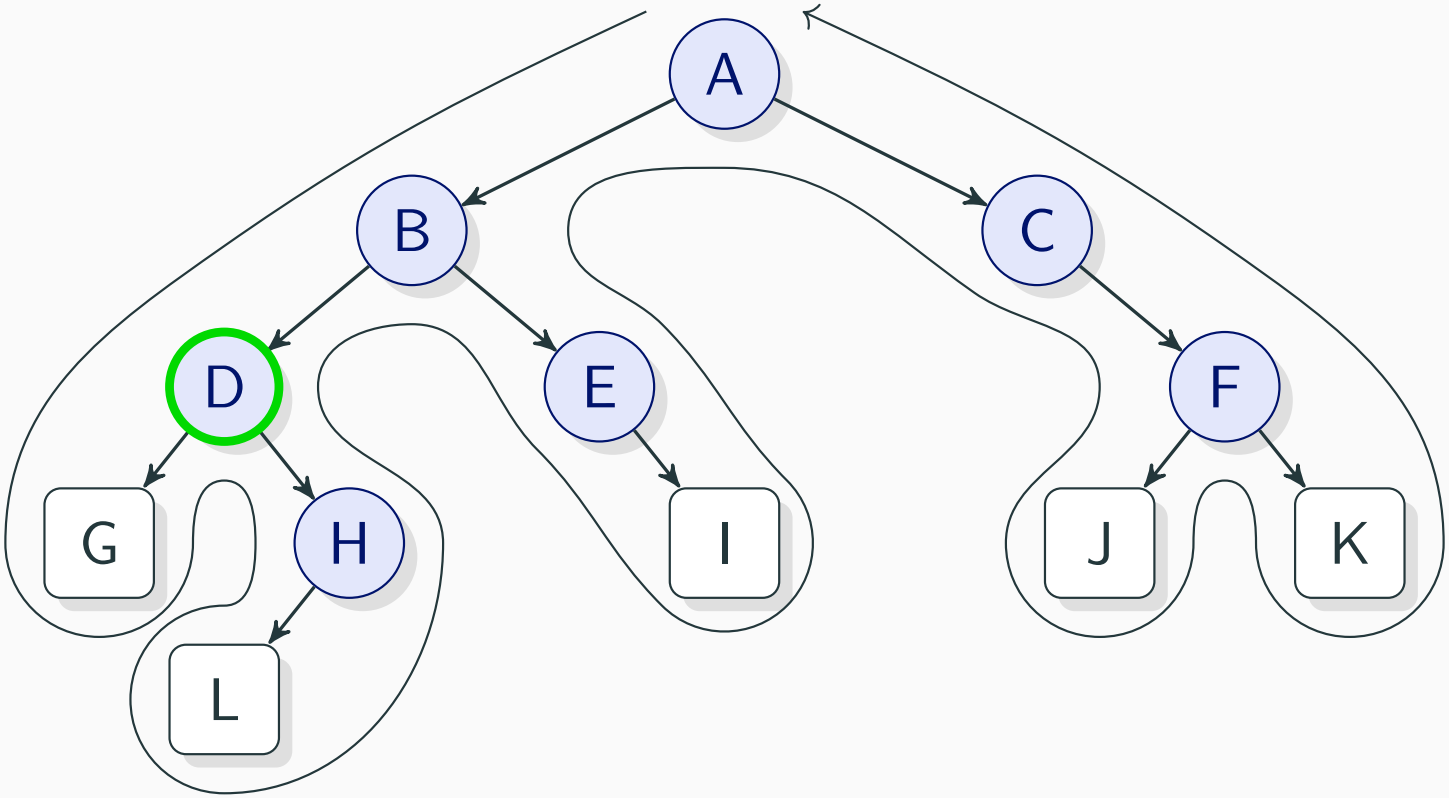
L'exécution de `dfs_prefix` se déroule de la façon suivante :



ABDG

Parcours préfixe

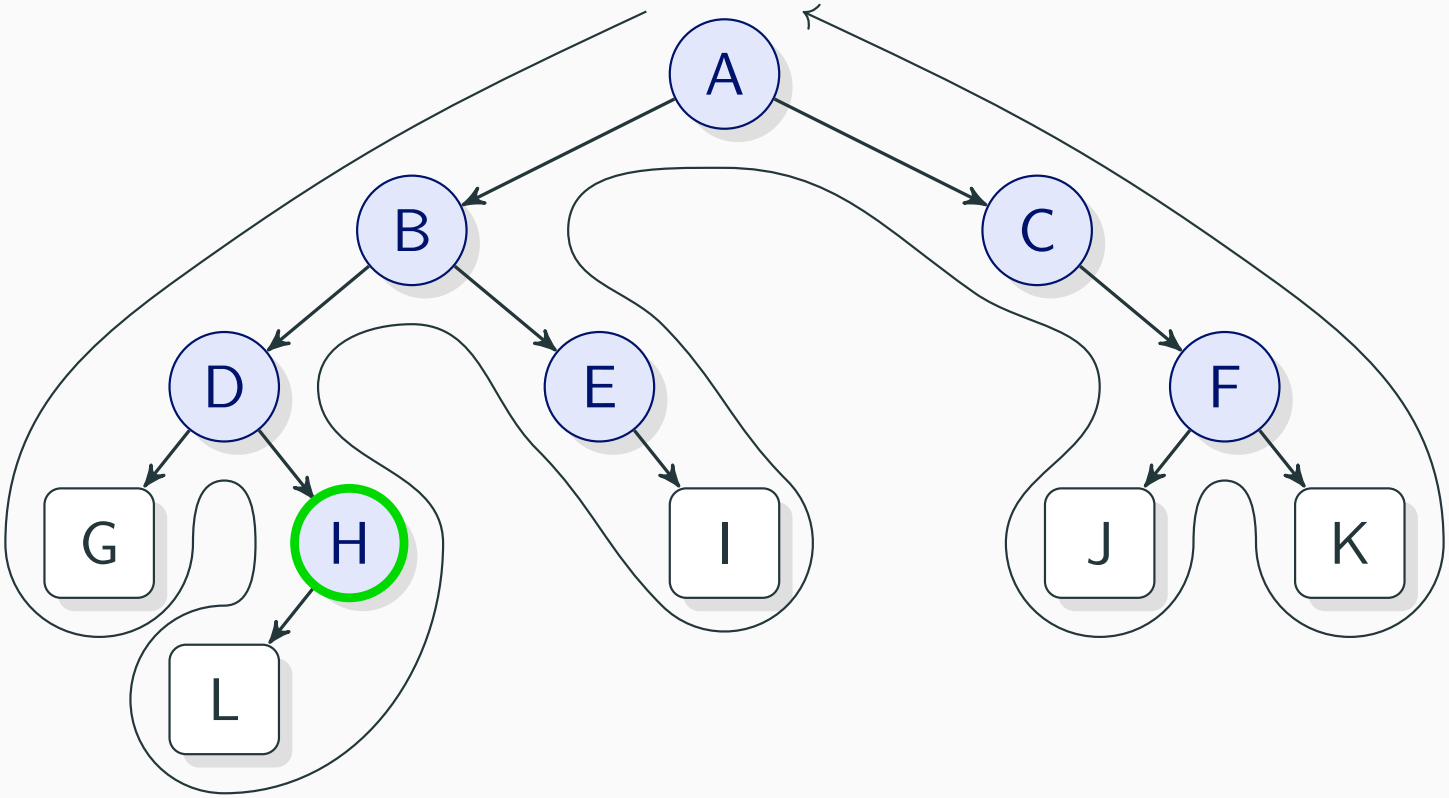
L'exécution de `dfs_prefix` se déroule de la façon suivante :



ABDG

Parcours préfixe

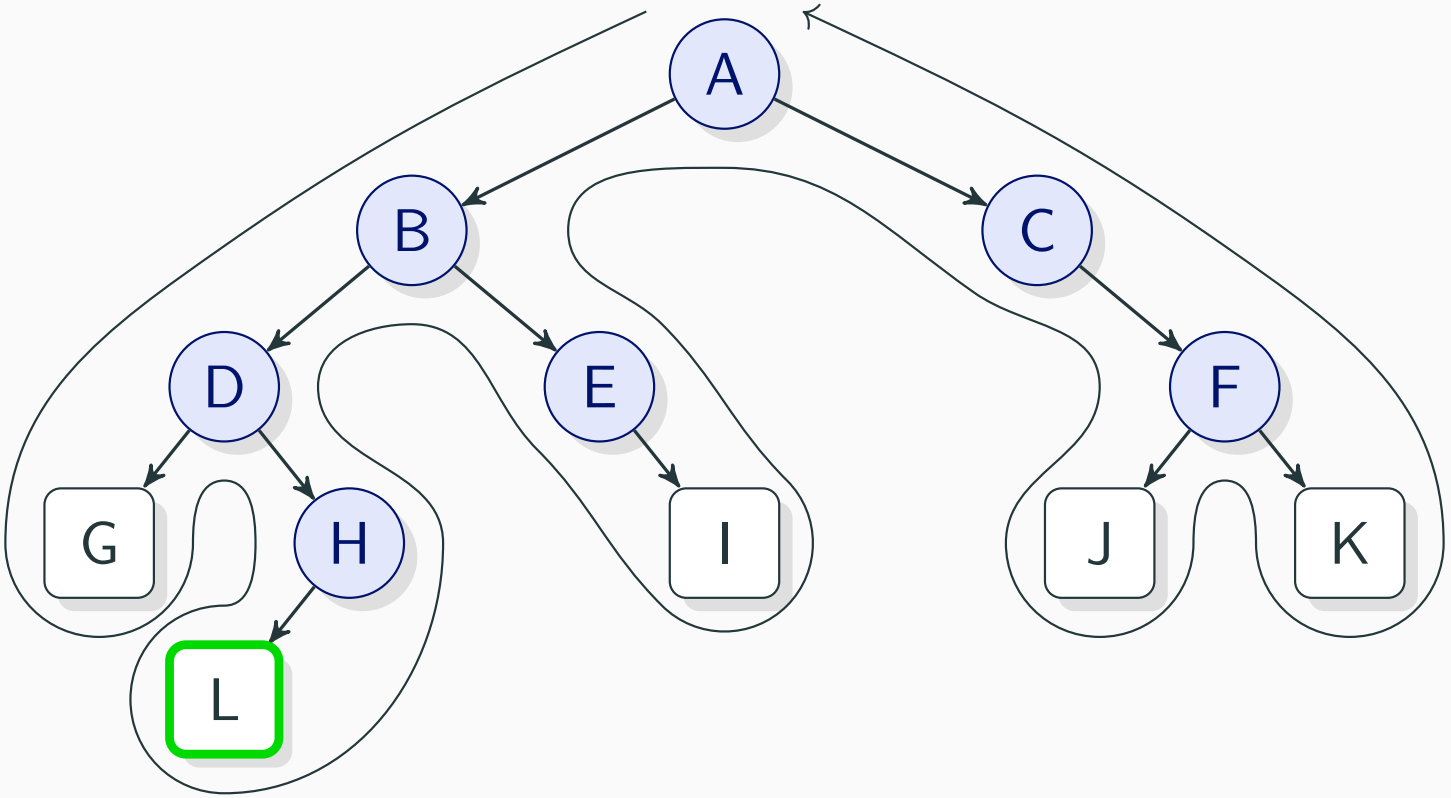
L'exécution de `dfs_prefix` se déroule de la façon suivante :



ABDGH

Parcours préfixe

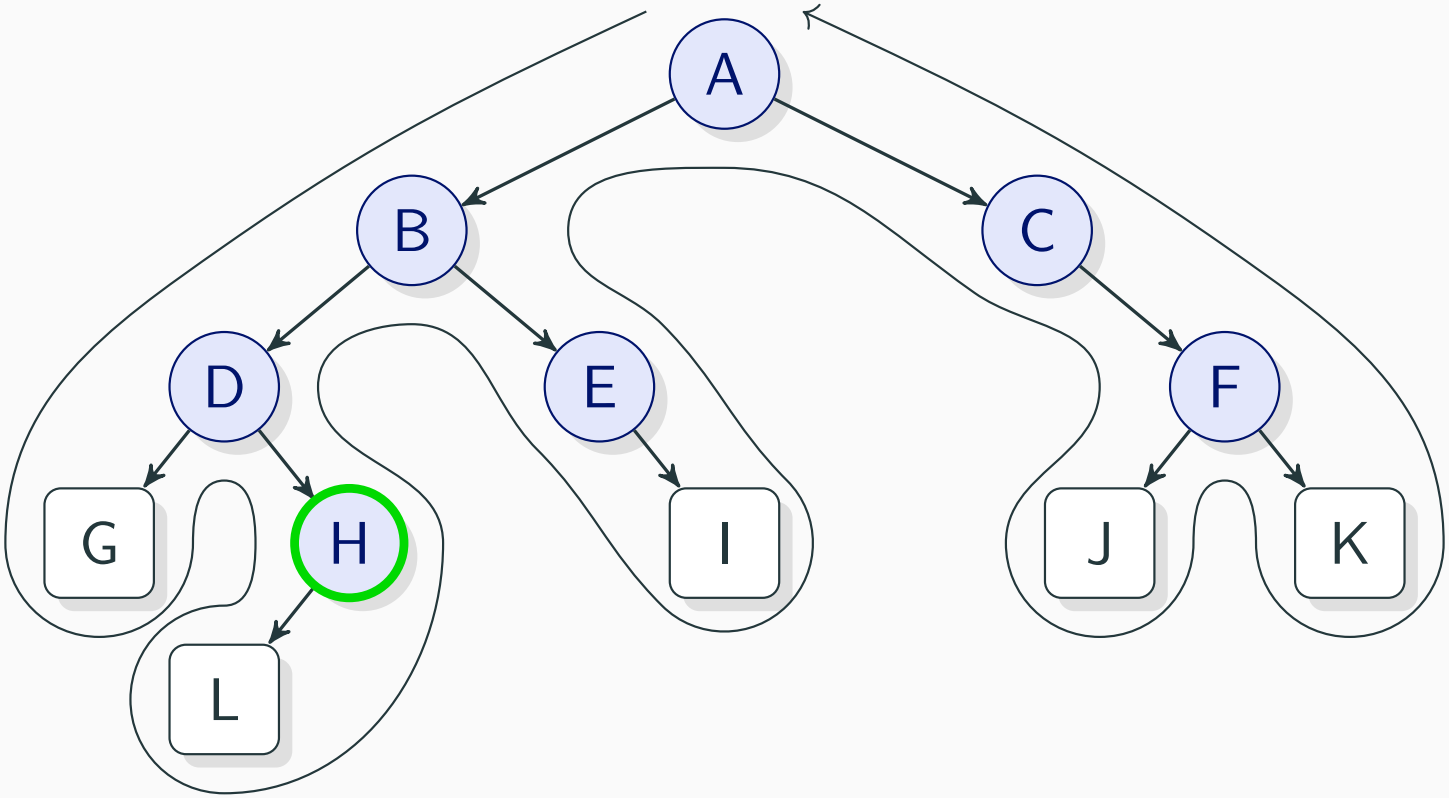
L'exécution de `dfs_prefix` se déroule de la façon suivante :



ABDGHL

Parcours préfixe

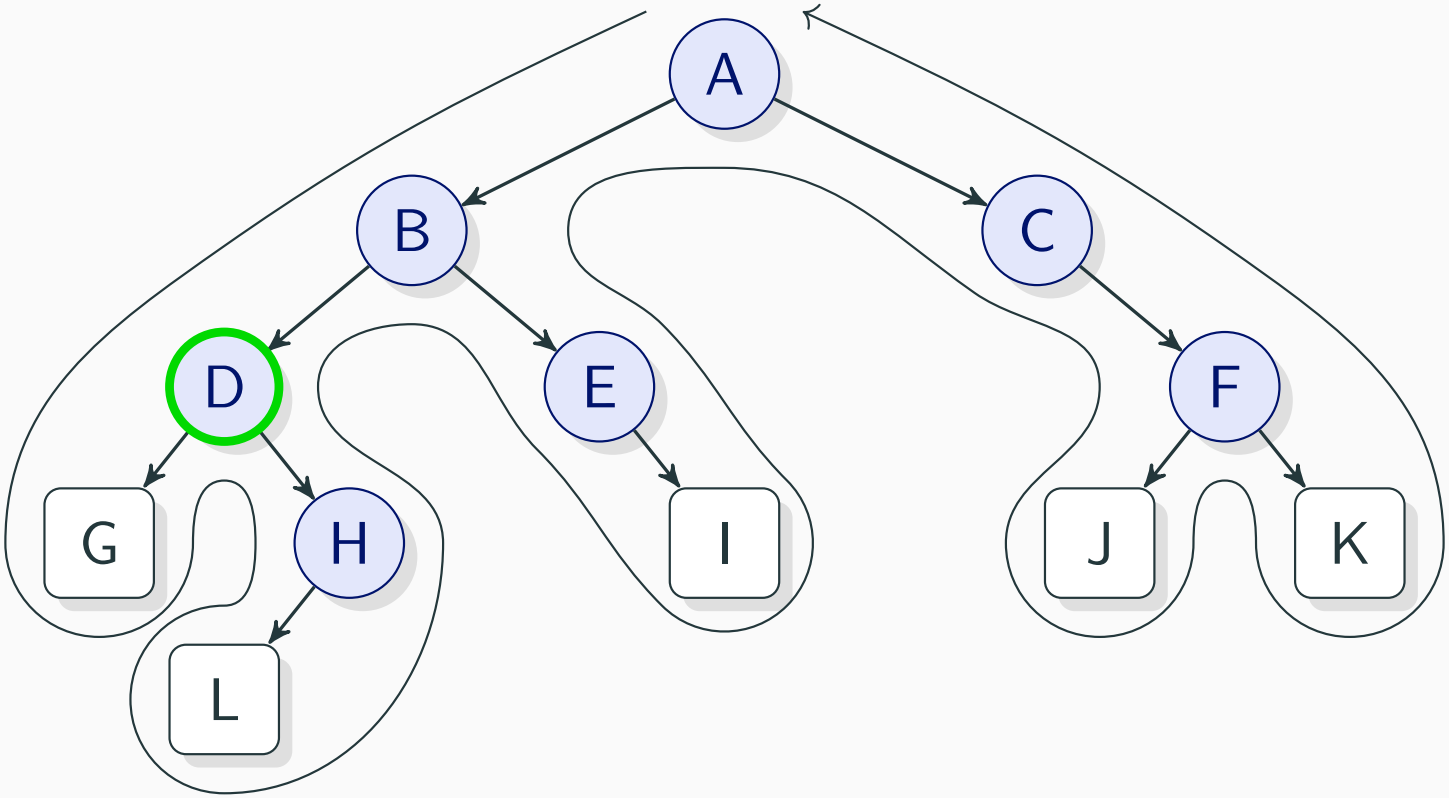
L'exécution de `dfs_prefix` se déroule de la façon suivante :



ABDGHL

Parcours préfixe

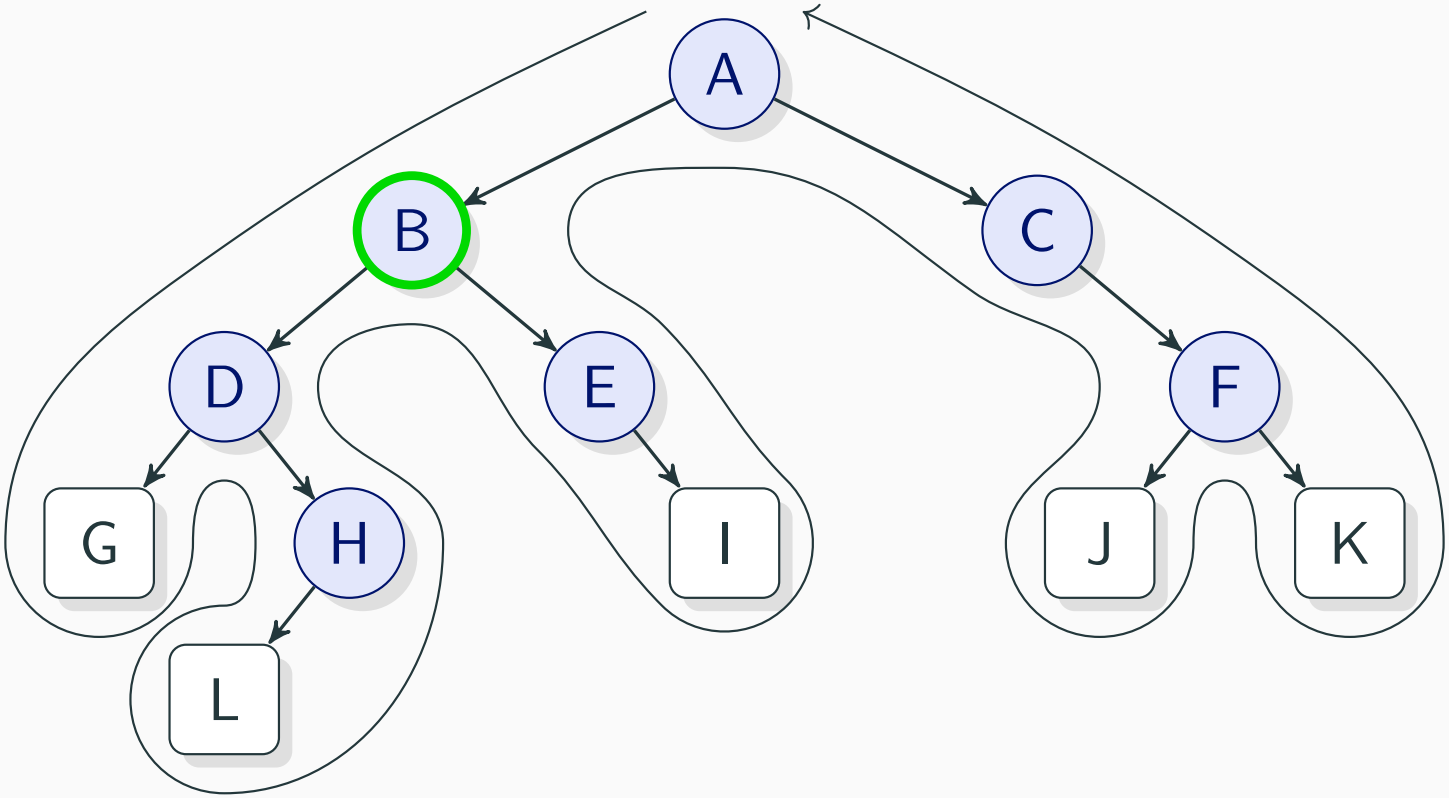
L'exécution de `dfs_prefix` se déroule de la façon suivante :



ABDGHL

Parcours préfixe

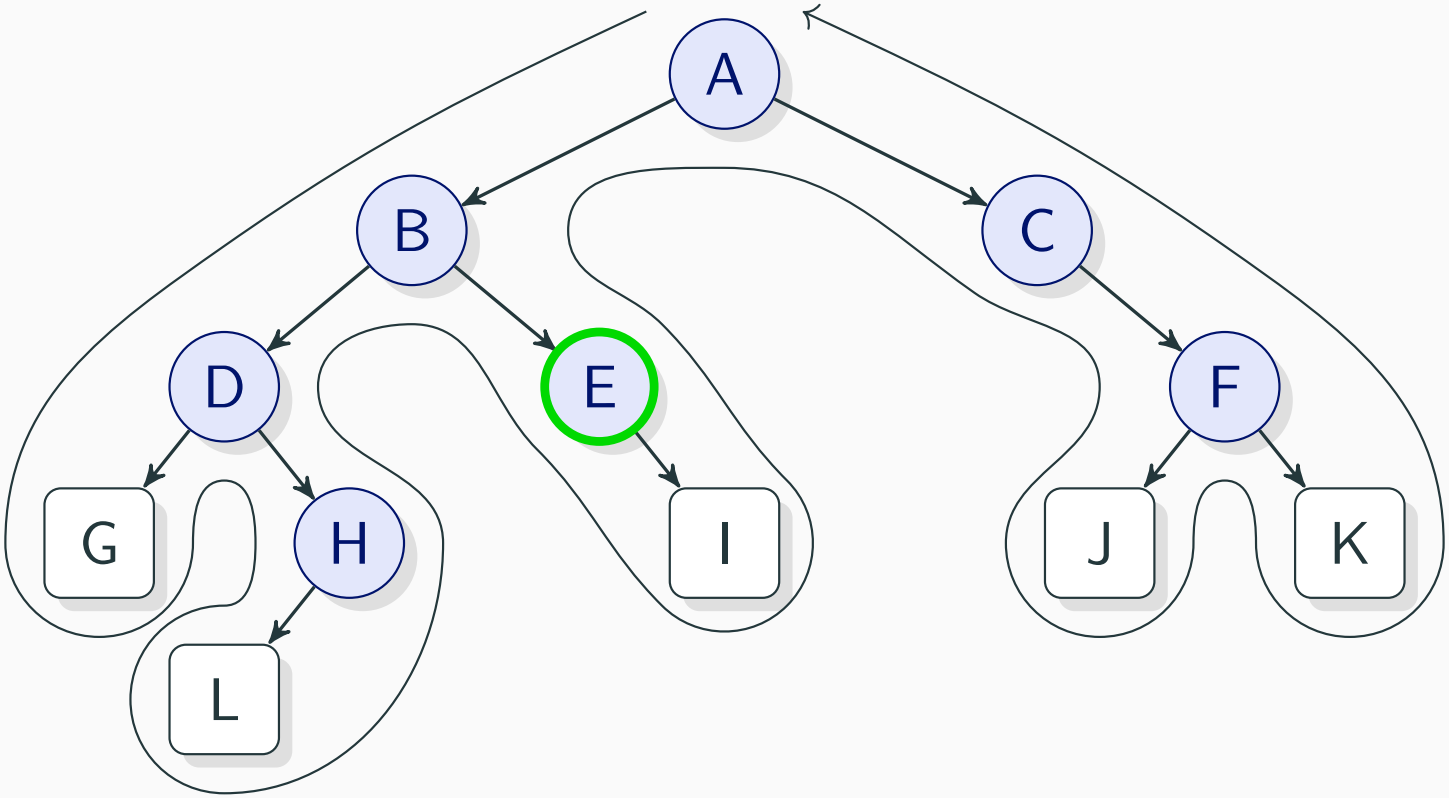
L'exécution de `dfs_prefix` se déroule de la façon suivante :



ABDGHL

Parcours préfixe

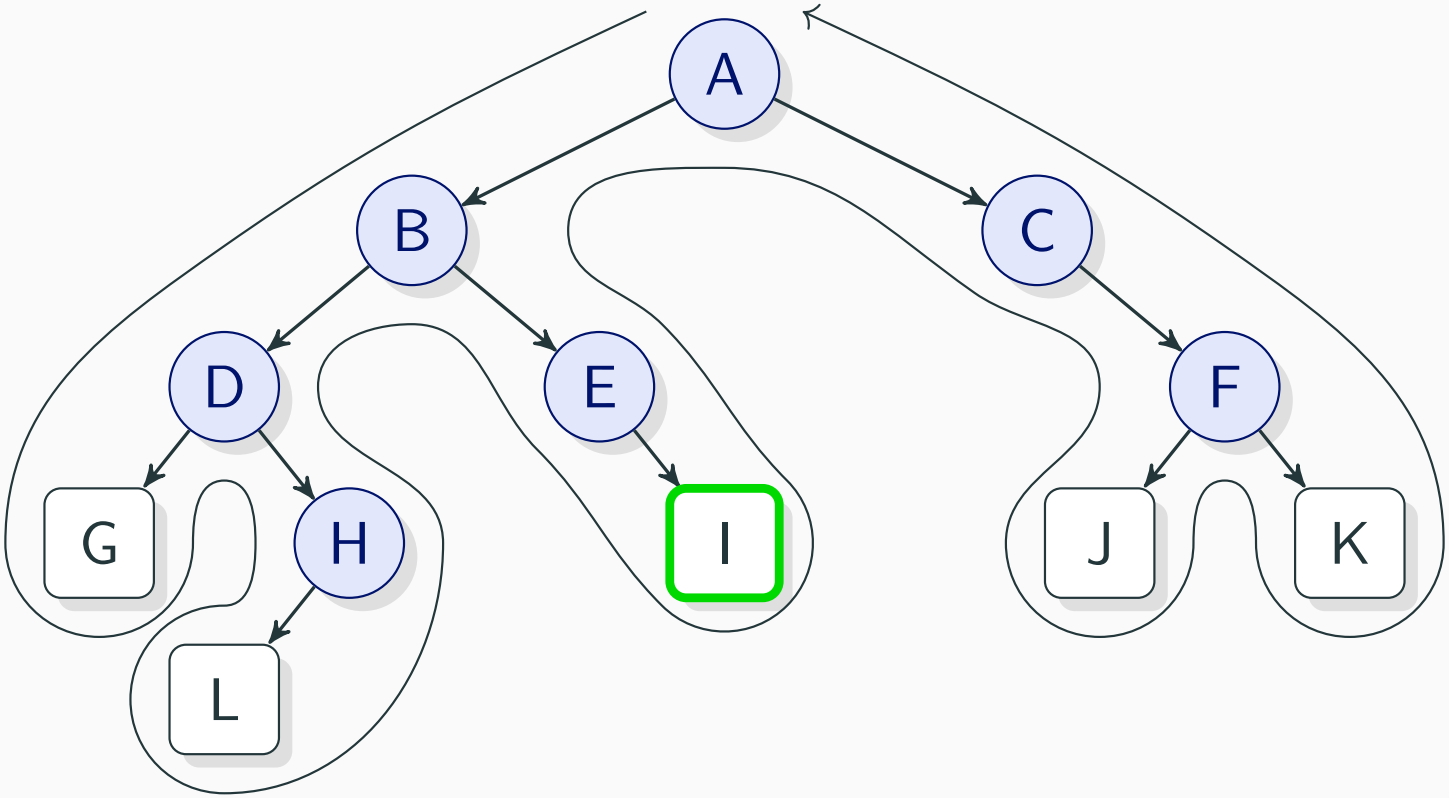
L'exécution de `dfs_prefix` se déroule de la façon suivante :



ABDGHLE

Parcours préfixe

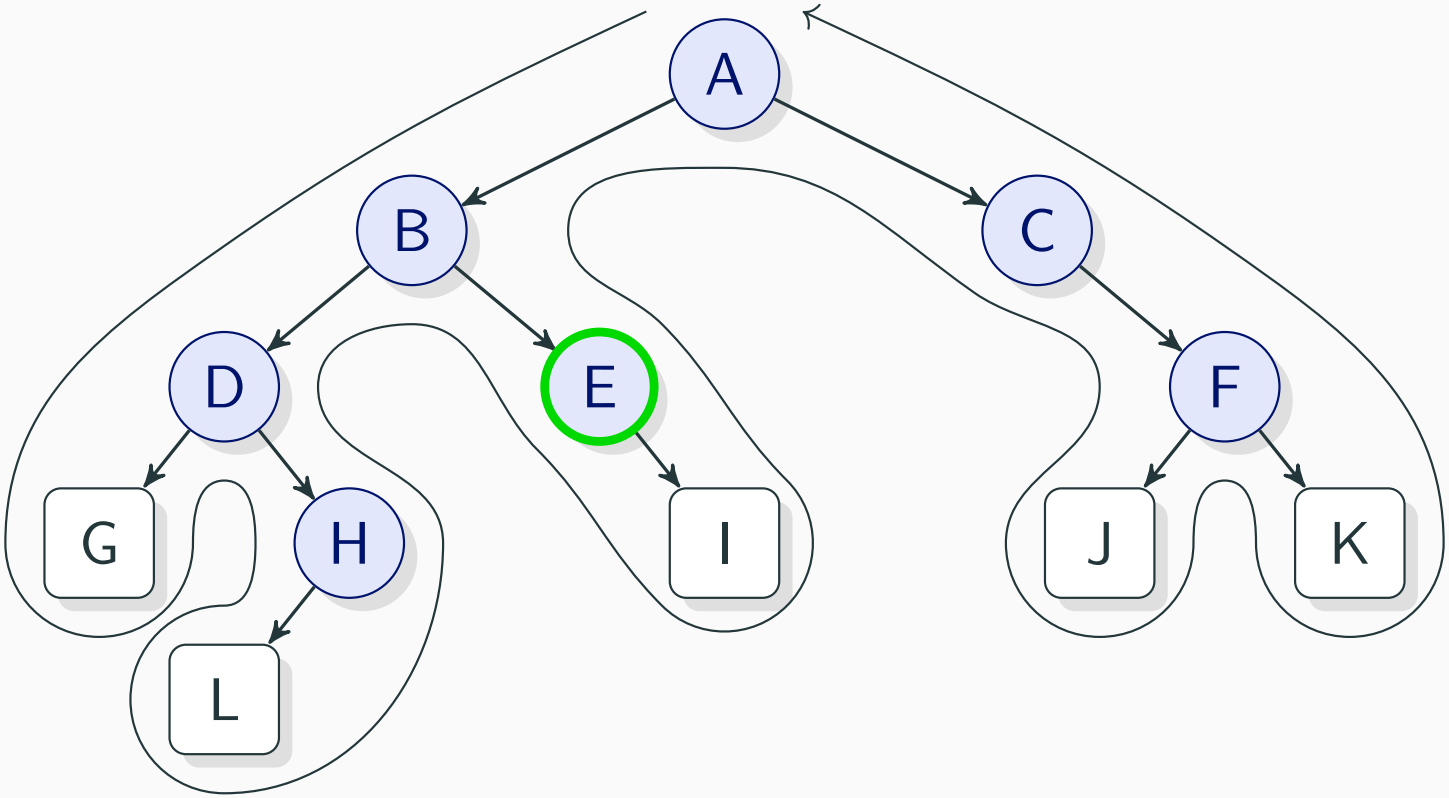
L'exécution de `dfs_prefix` se déroule de la façon suivante :



ABDGHLEI

Parcours préfixe

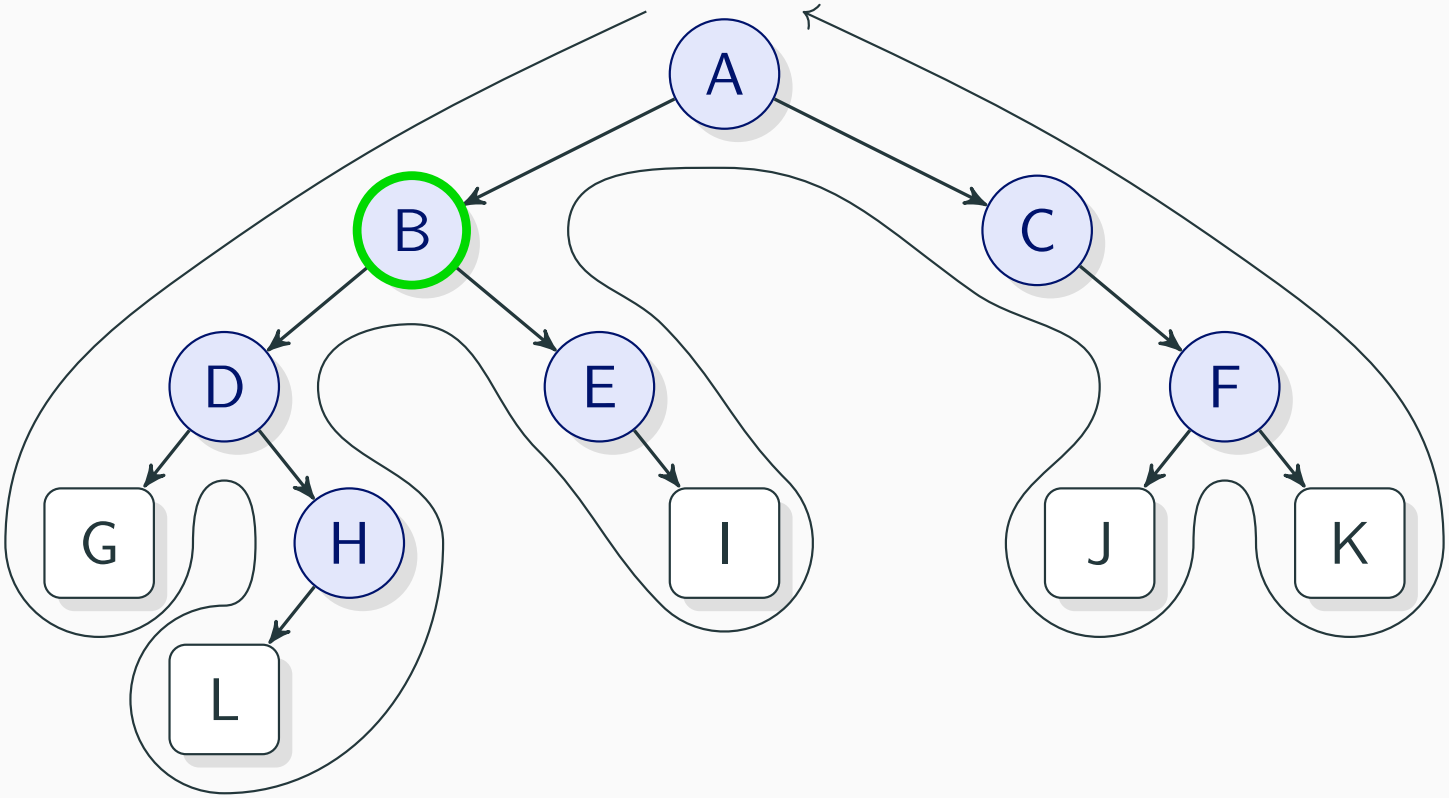
L'exécution de `dfs_prefix` se déroule de la façon suivante :



ABDGHLEI

Parcours préfixe

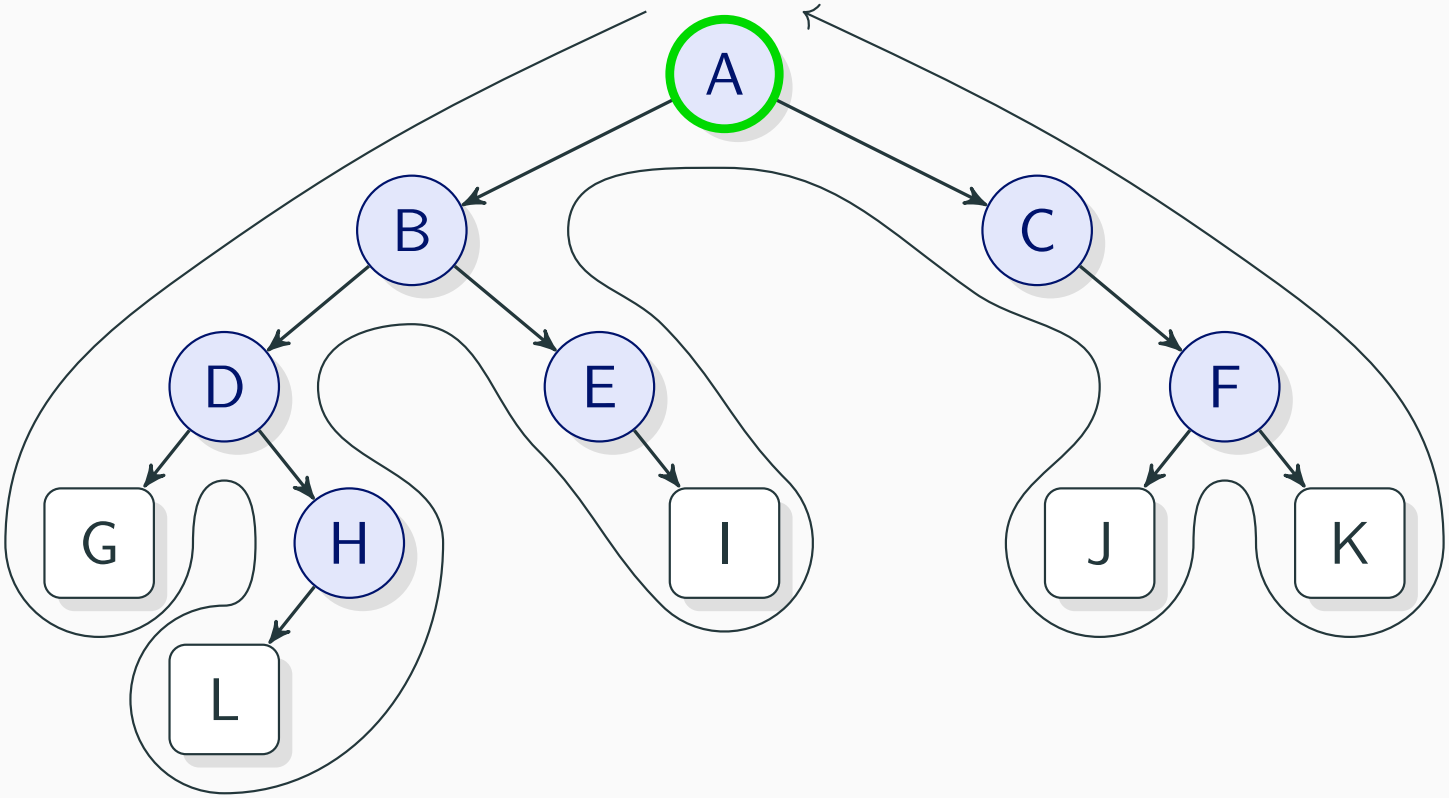
L'exécution de `dfs_prefix` se déroule de la façon suivante :



ABDGHLEI

Parcours préfixe

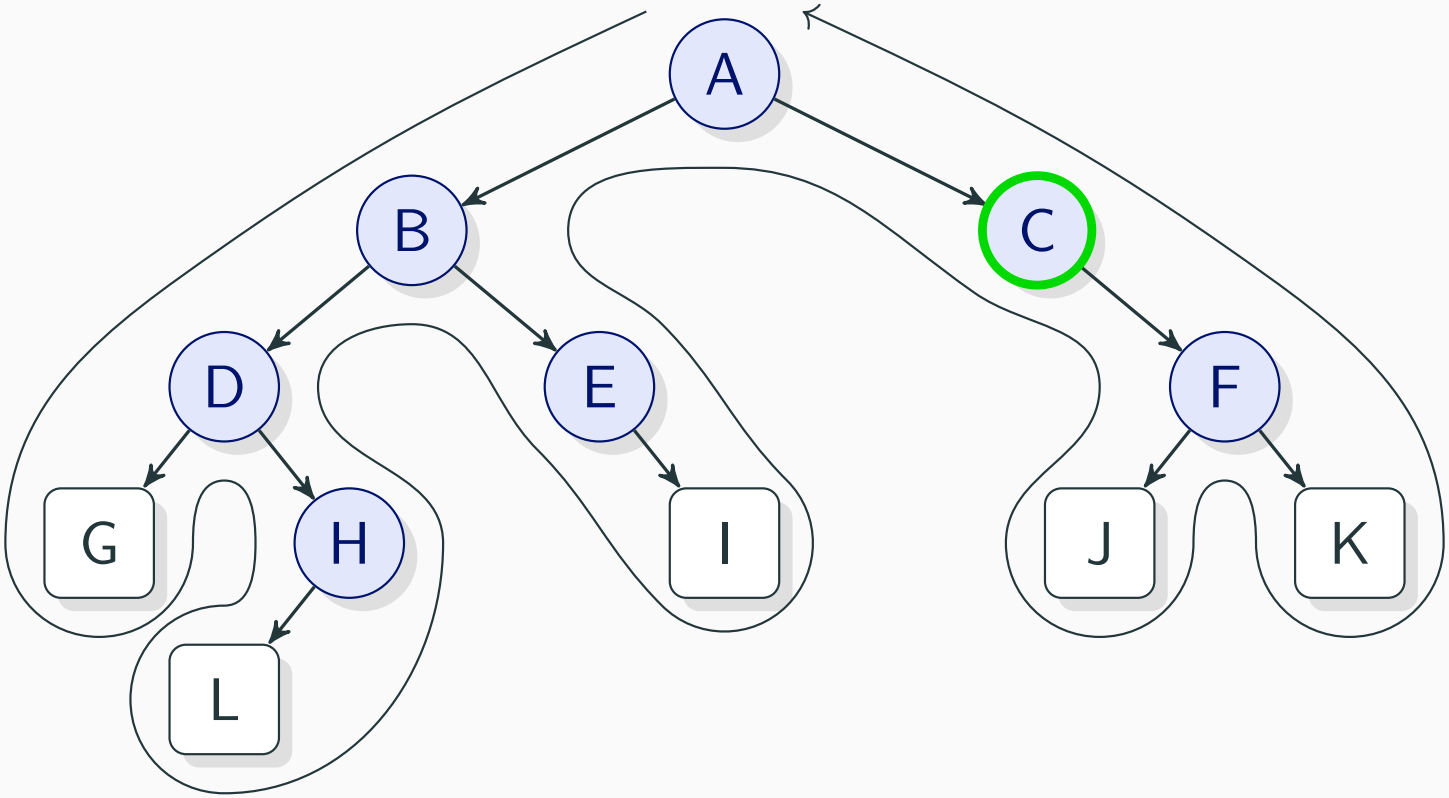
L'exécution de `dfs_prefix` se déroule de la façon suivante :



ABDGHLEI

Parcours préfixe

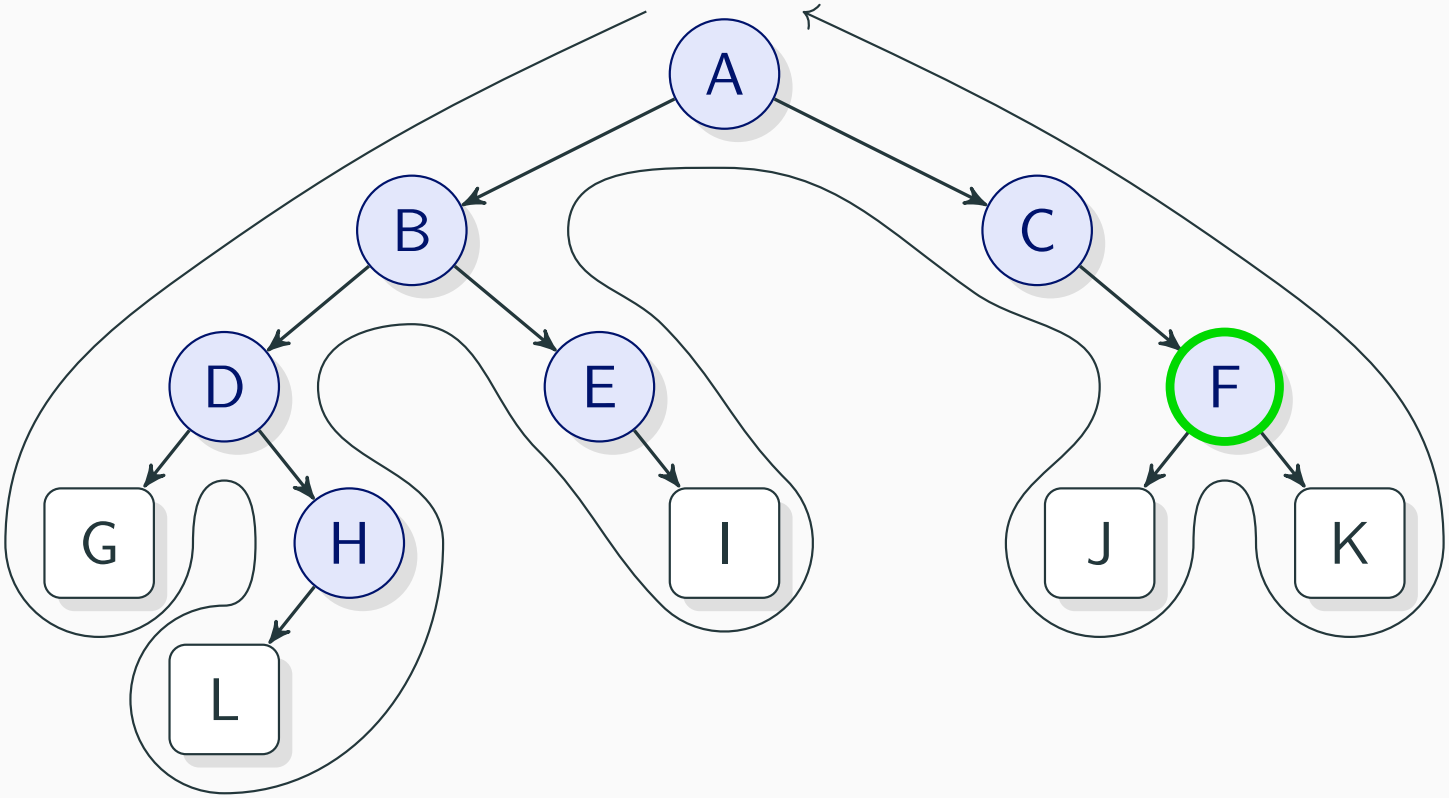
L'exécution de `dfs_prefix` se déroule de la façon suivante :



ABDGHLEIC

Parcours préfixe

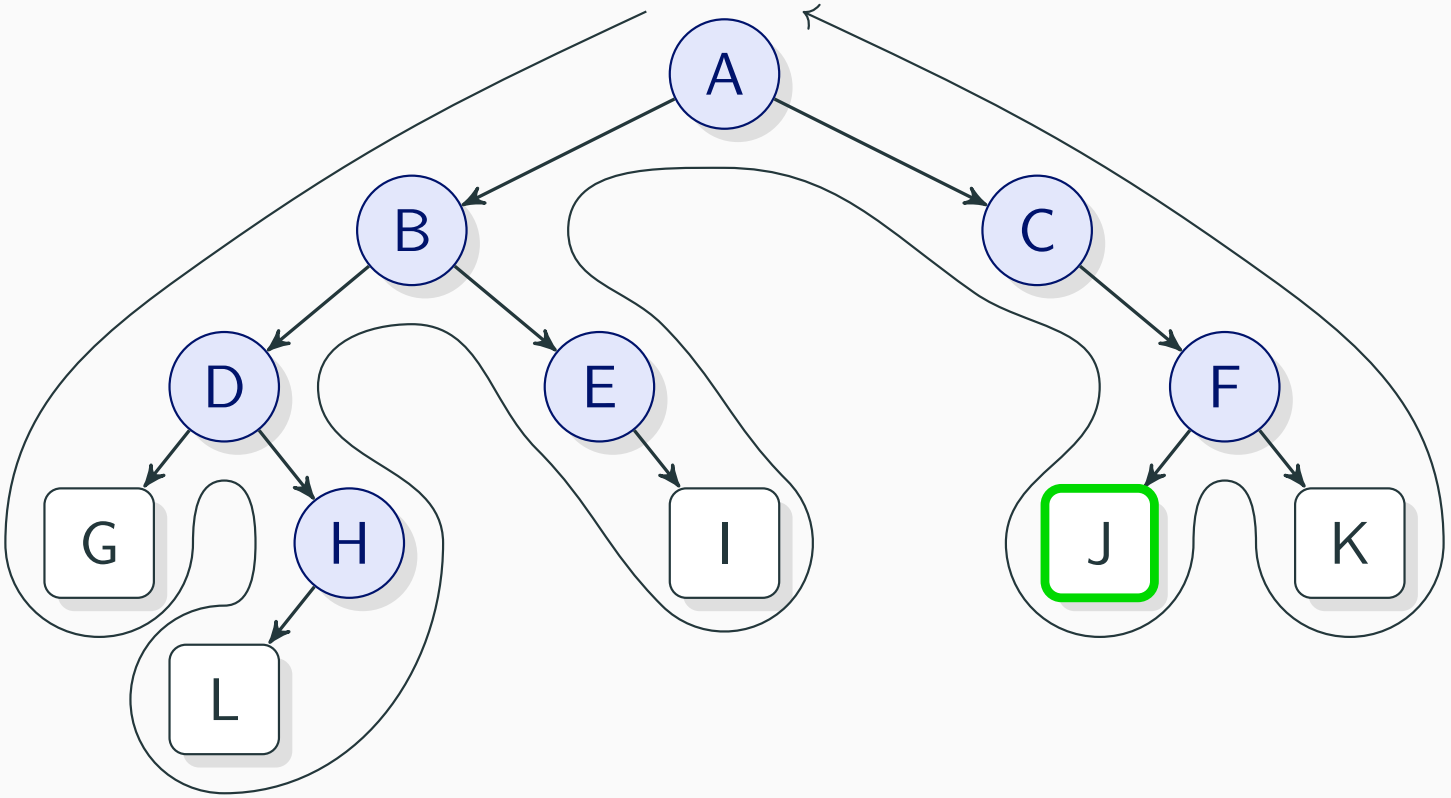
L'exécution de `dfs_prefix` se déroule de la façon suivante :



ABDGHLEICF

Parcours préfixe

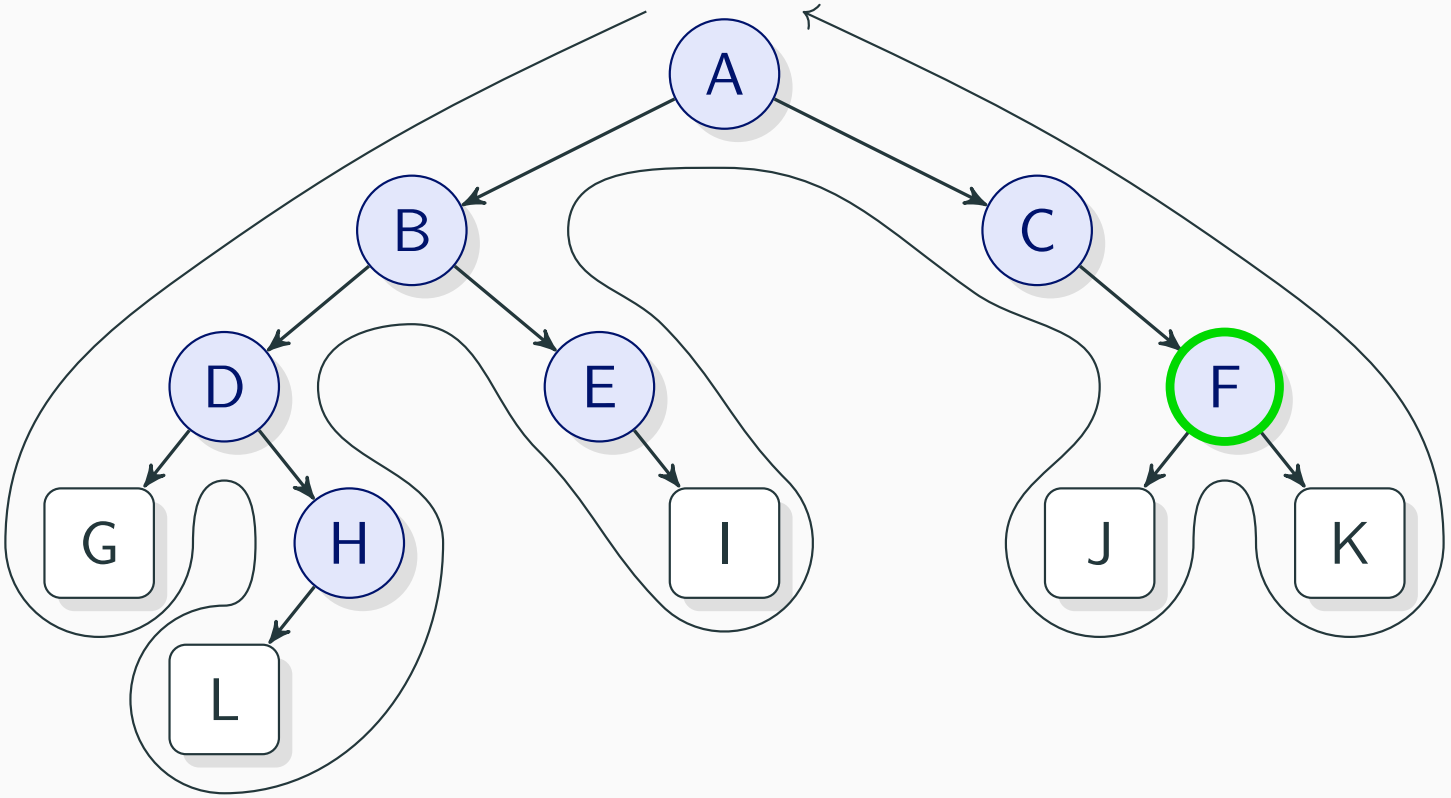
L'exécution de `dfs_prefix` se déroule de la façon suivante :



ABDGHLEICFJ

Parcours préfixe

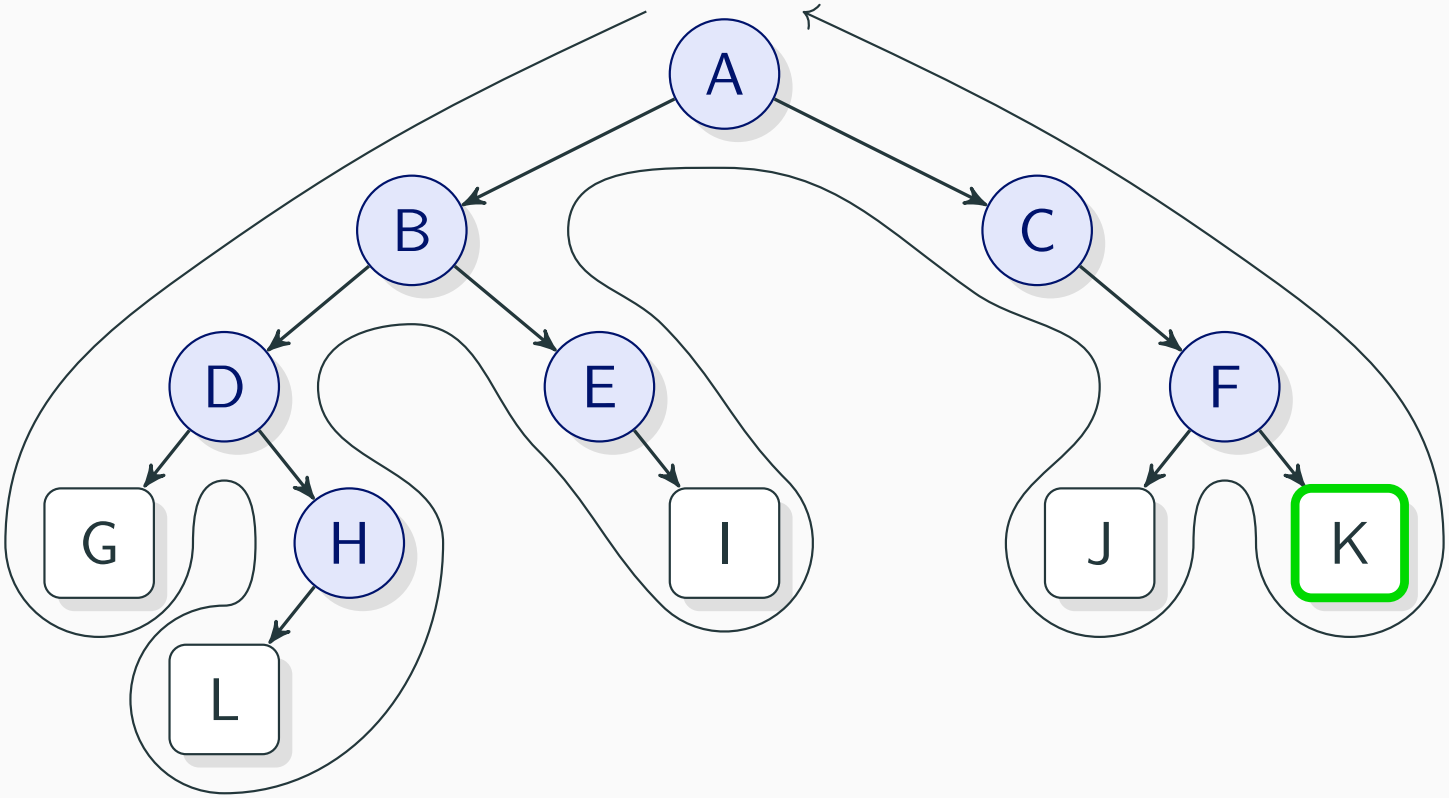
L'exécution de `dfs_prefix` se déroule de la façon suivante :



ABDGHLEICFJ

Parcours préfixe

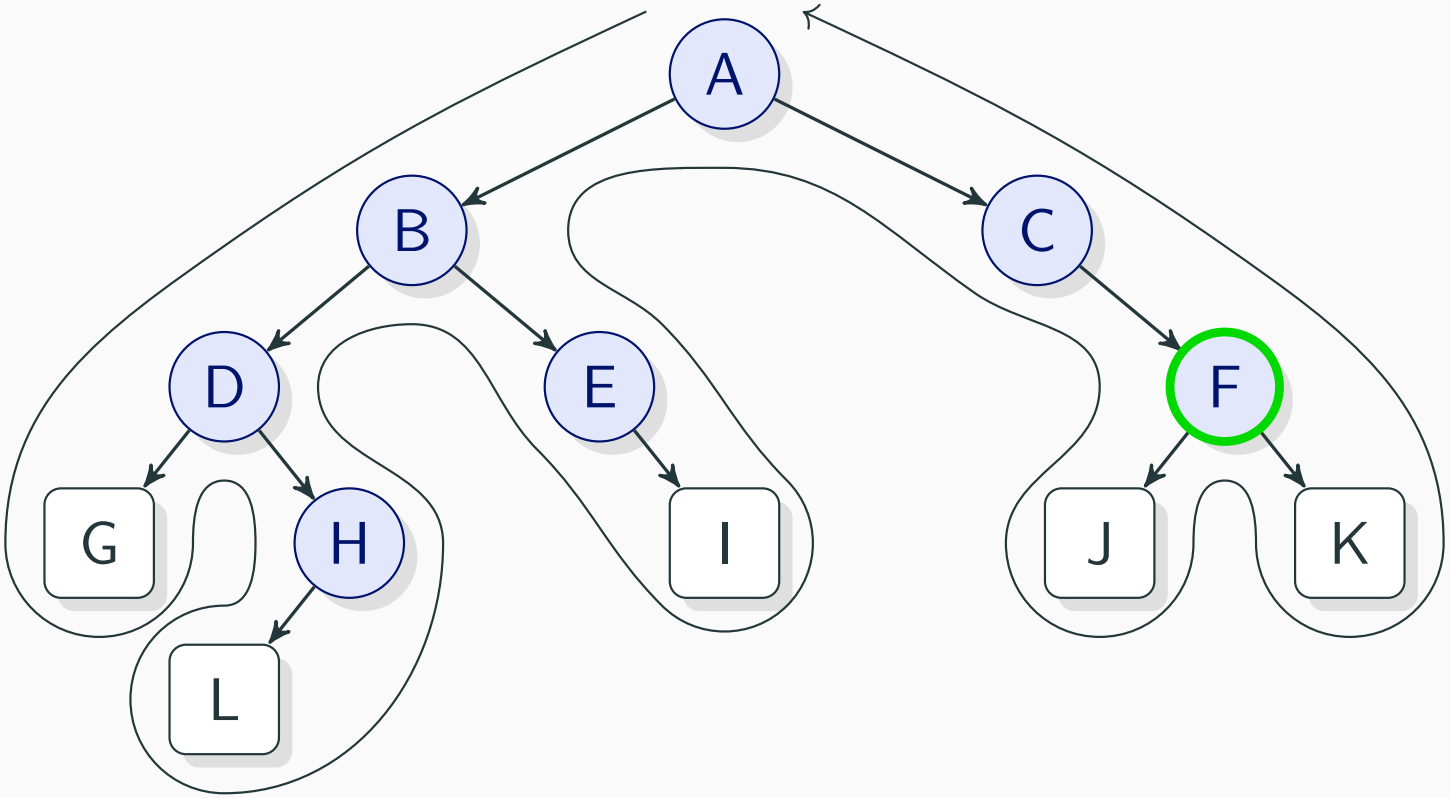
L'exécution de `dfs_prefix` se déroule de la façon suivante :



ABDGHLEICFJK

Parcours préfixe

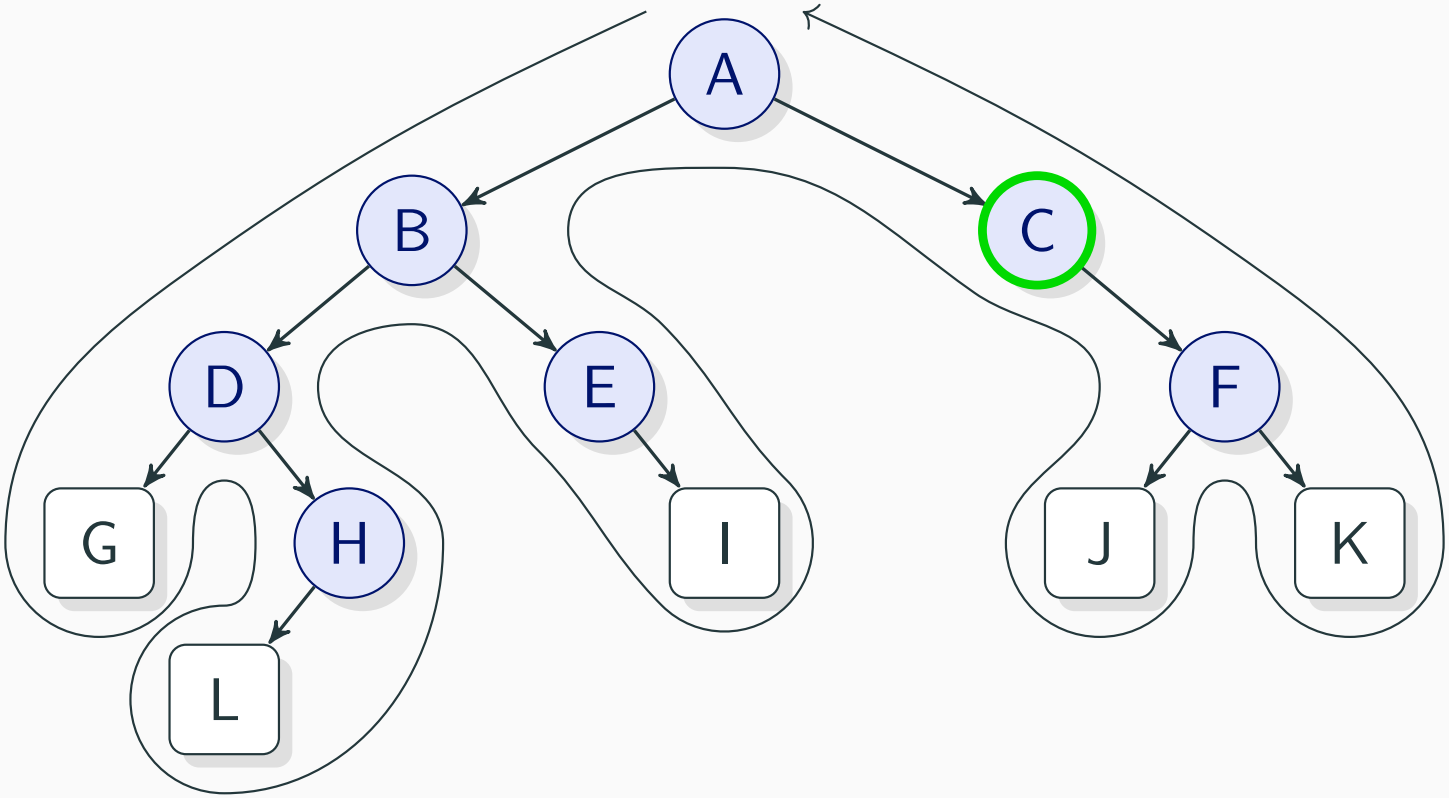
L'exécution de `dfs_prefix` se déroule de la façon suivante :



ABDGHLEICFJK

Parcours préfixe

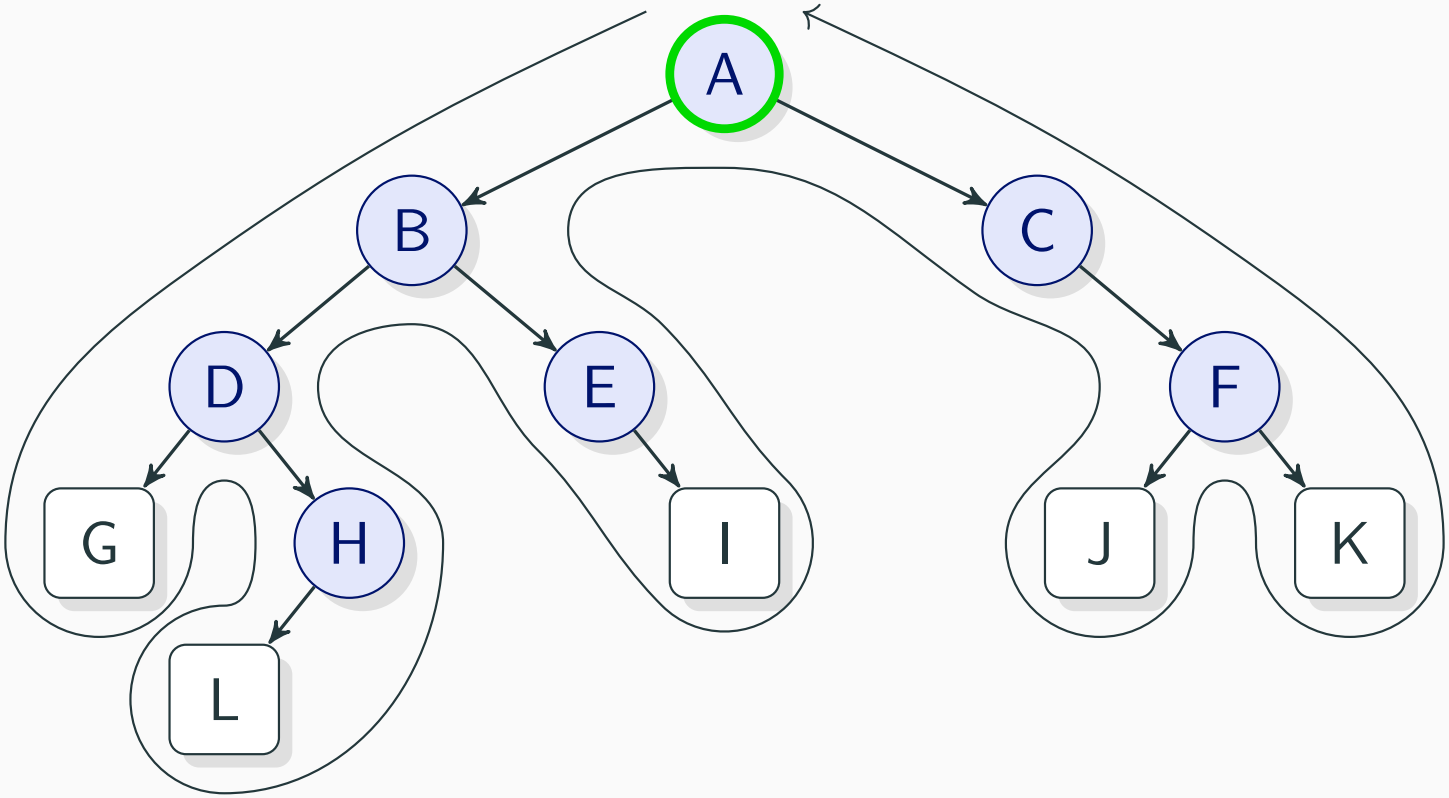
L'exécution de `dfs_prefix` se déroule de la façon suivante :



ABDGHLEICFJK

Parcours préfixe

L'exécution de `dfs_prefix` se déroule de la façon suivante :



ABDGHLEICFJK

Parcours suffixe

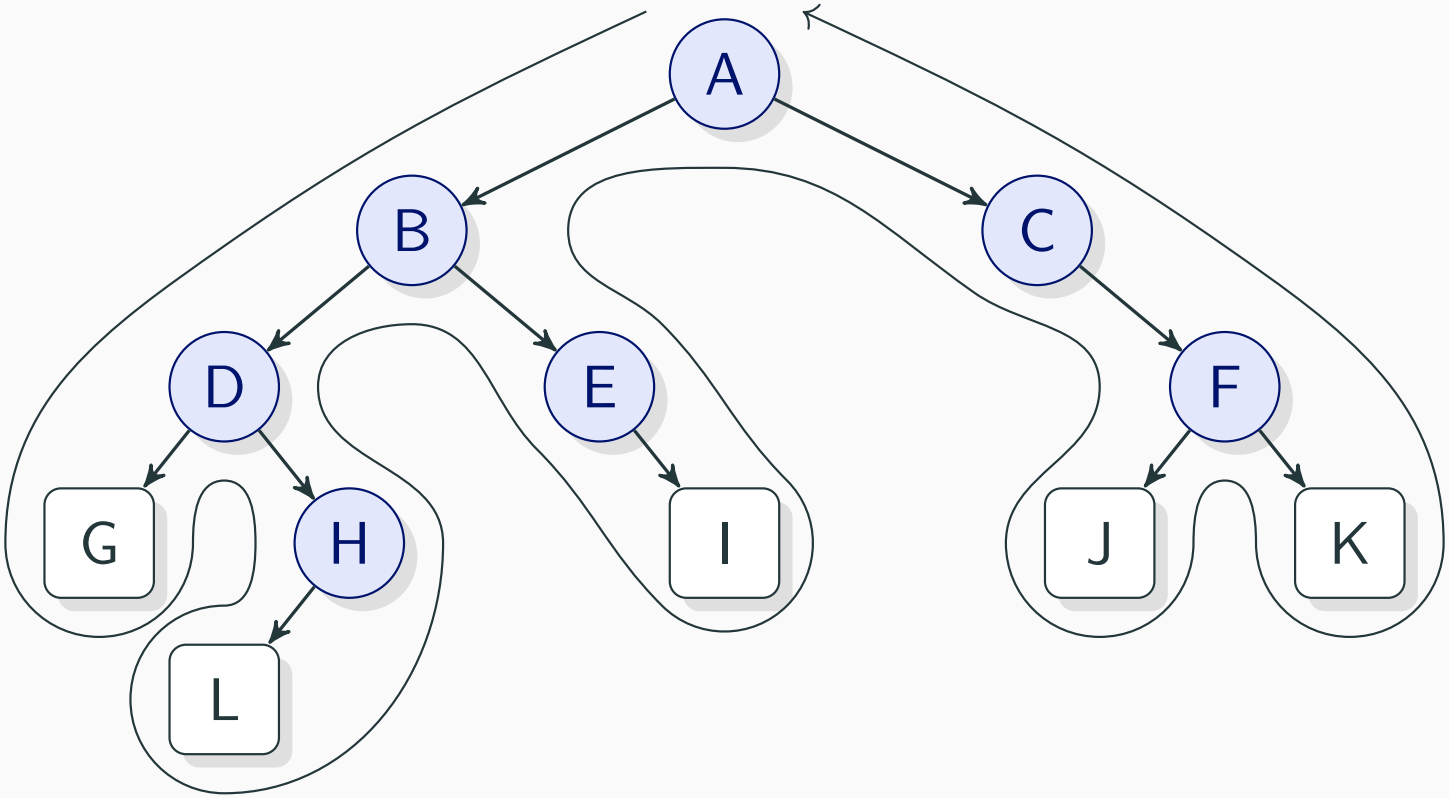
Dans un parcours en profondeur *suffixe*,

un nœud est traité *après* ses enfants (dernière visite)


```
# let rec dfs_suffix f = function
    | Nil -> ()
    | Node (e, lchild, rchild)
        -> dfs_suffix f lchild;
           dfs_suffix f rchild;
           f e;

val dfs_suffix : ('a -> 'b) -> 'a arbre -> unit = <fun>
```

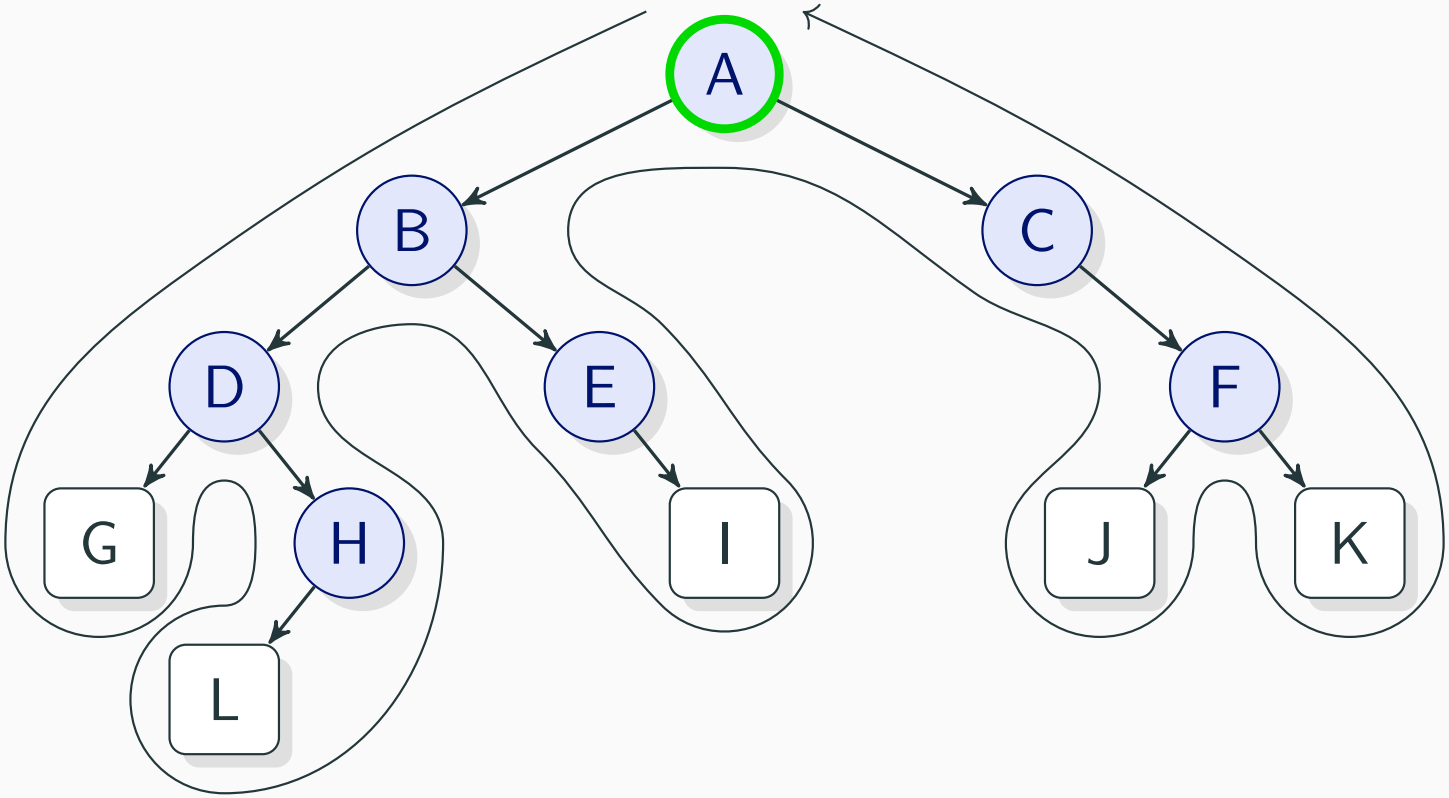
Parcours suffixe

L'exécution de `dfs_suffix` se déroule de la façon suivante :



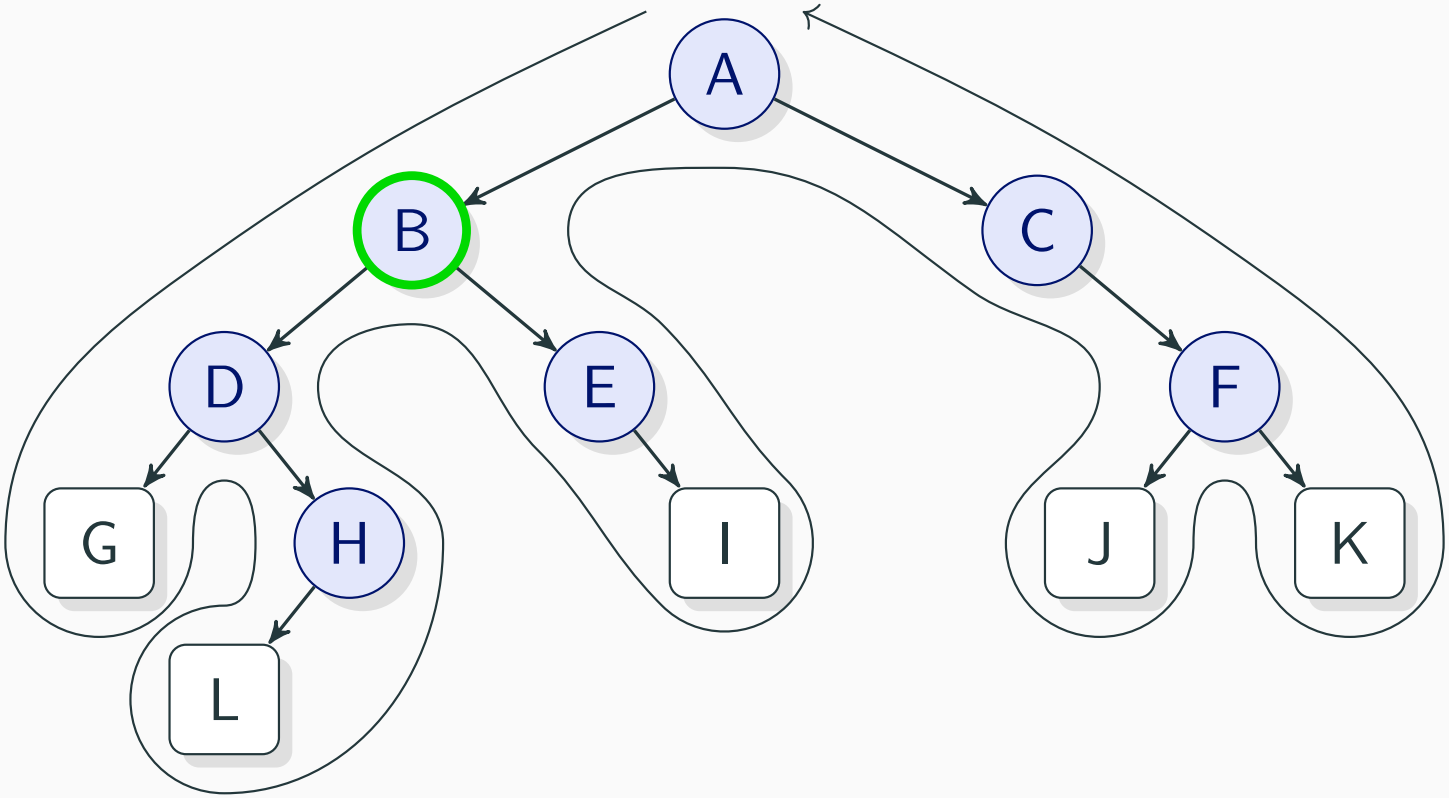
Parcours suffixe

L'exécution de `dfs_suffix` se déroule de la façon suivante :



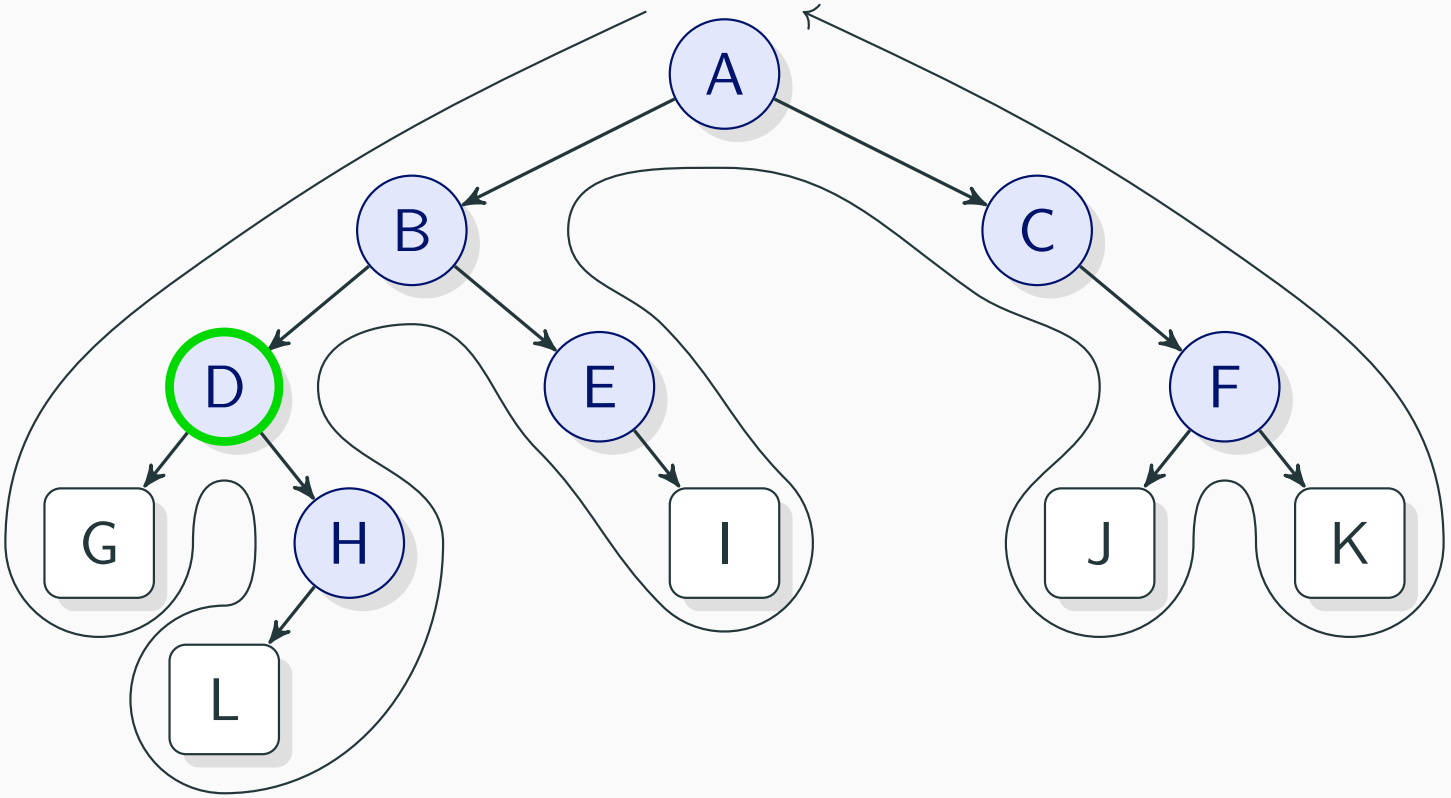
Parcours suffixe

L'exécution de `dfs_suffix` se déroule de la façon suivante :



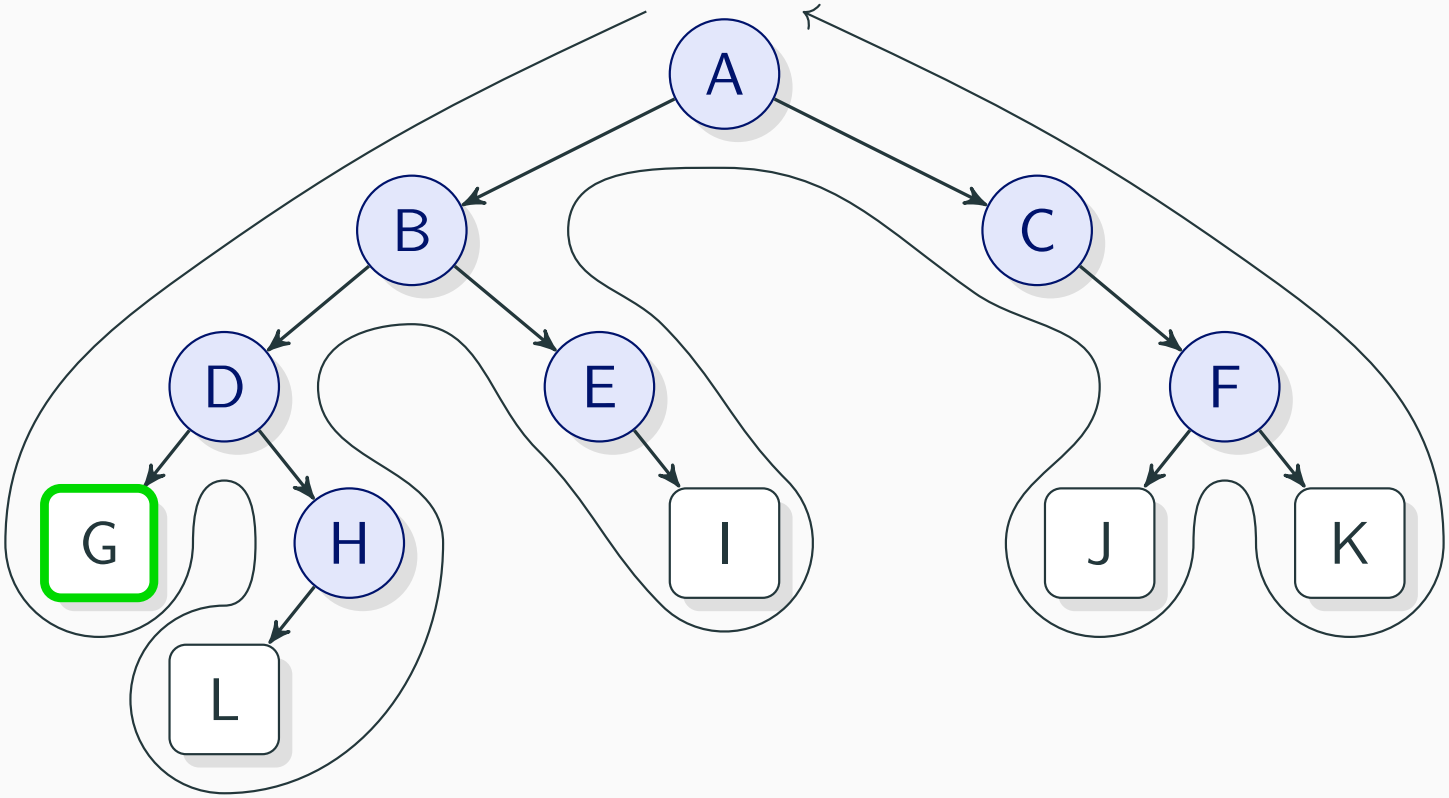
Parcours suffixe

L'exécution de `dfs_suffix` se déroule de la façon suivante :



Parcours suffixe

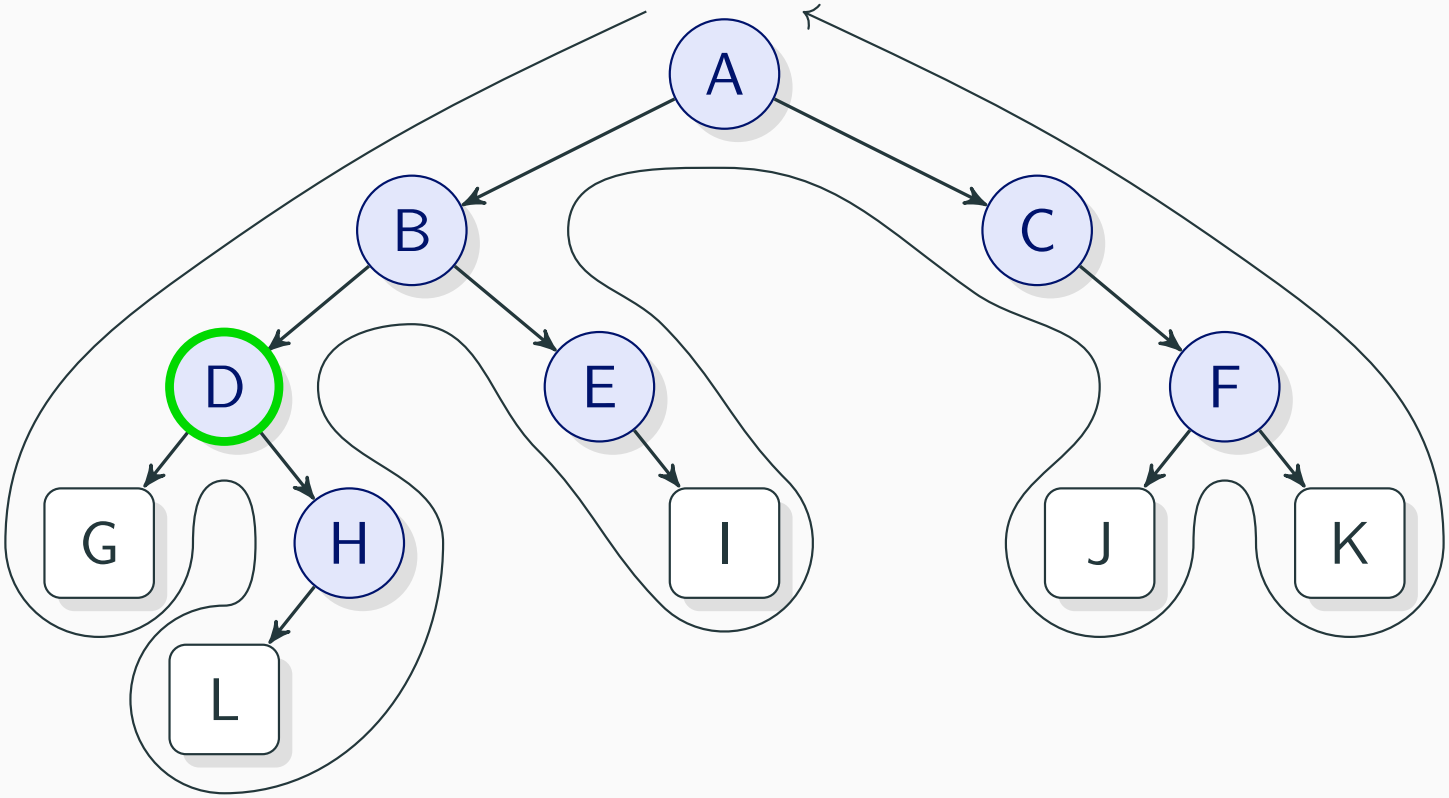
L'exécution de `dfs_suffix` se déroule de la façon suivante :



G

Parcours suffixe

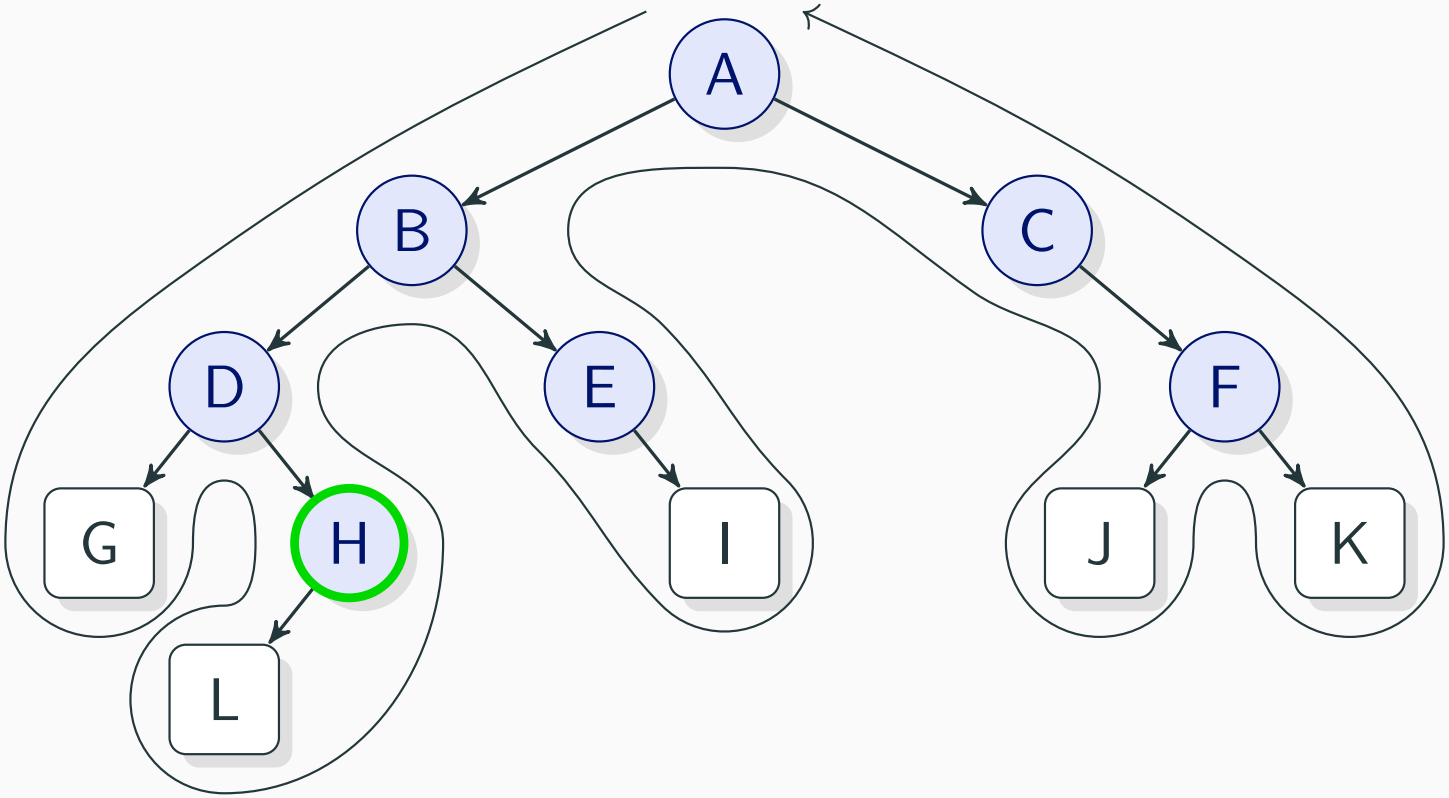
L'exécution de `dfs_suffix` se déroule de la façon suivante :



G

Parcours suffixe

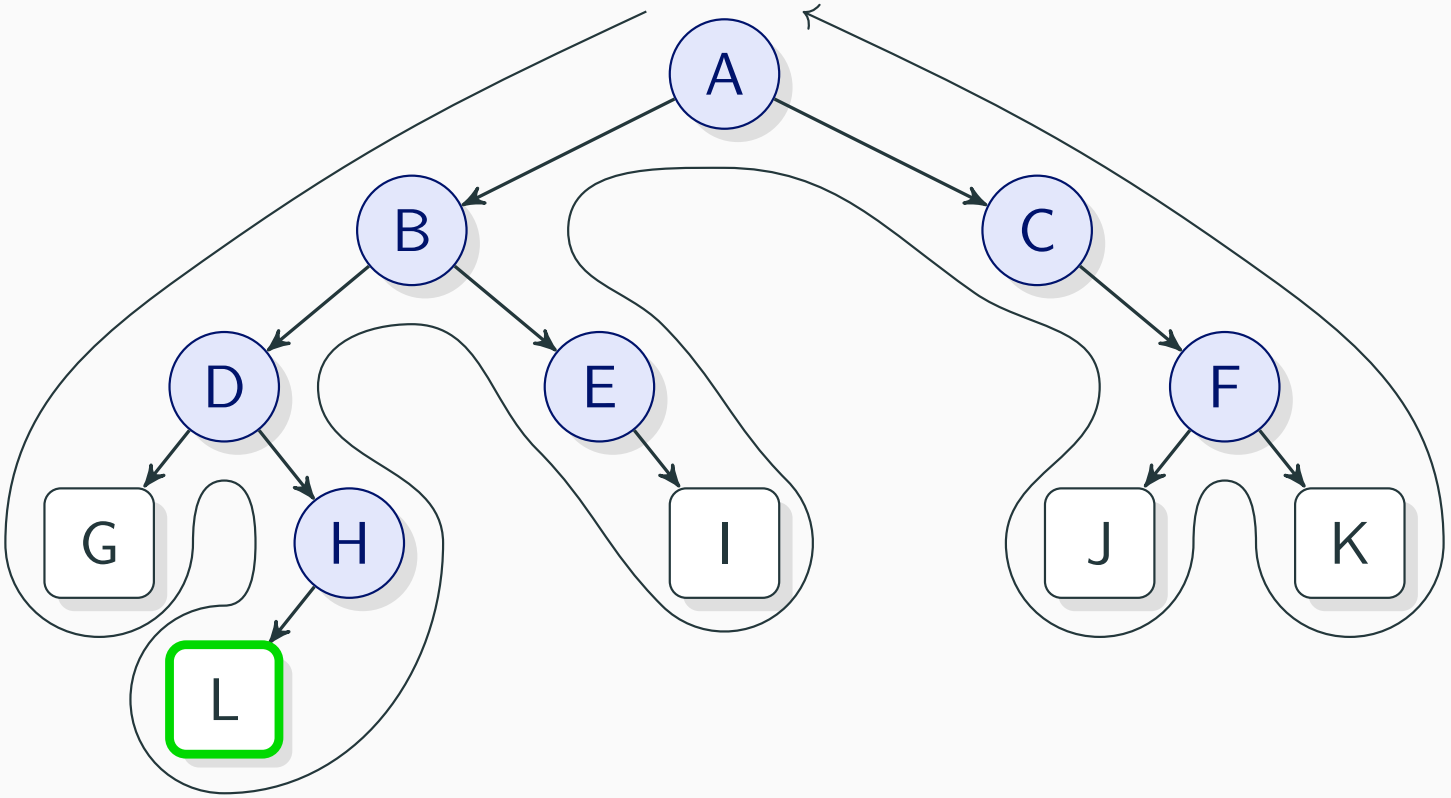
L'exécution de `dfs_suffix` se déroule de la façon suivante :



G

Parcours suffixe

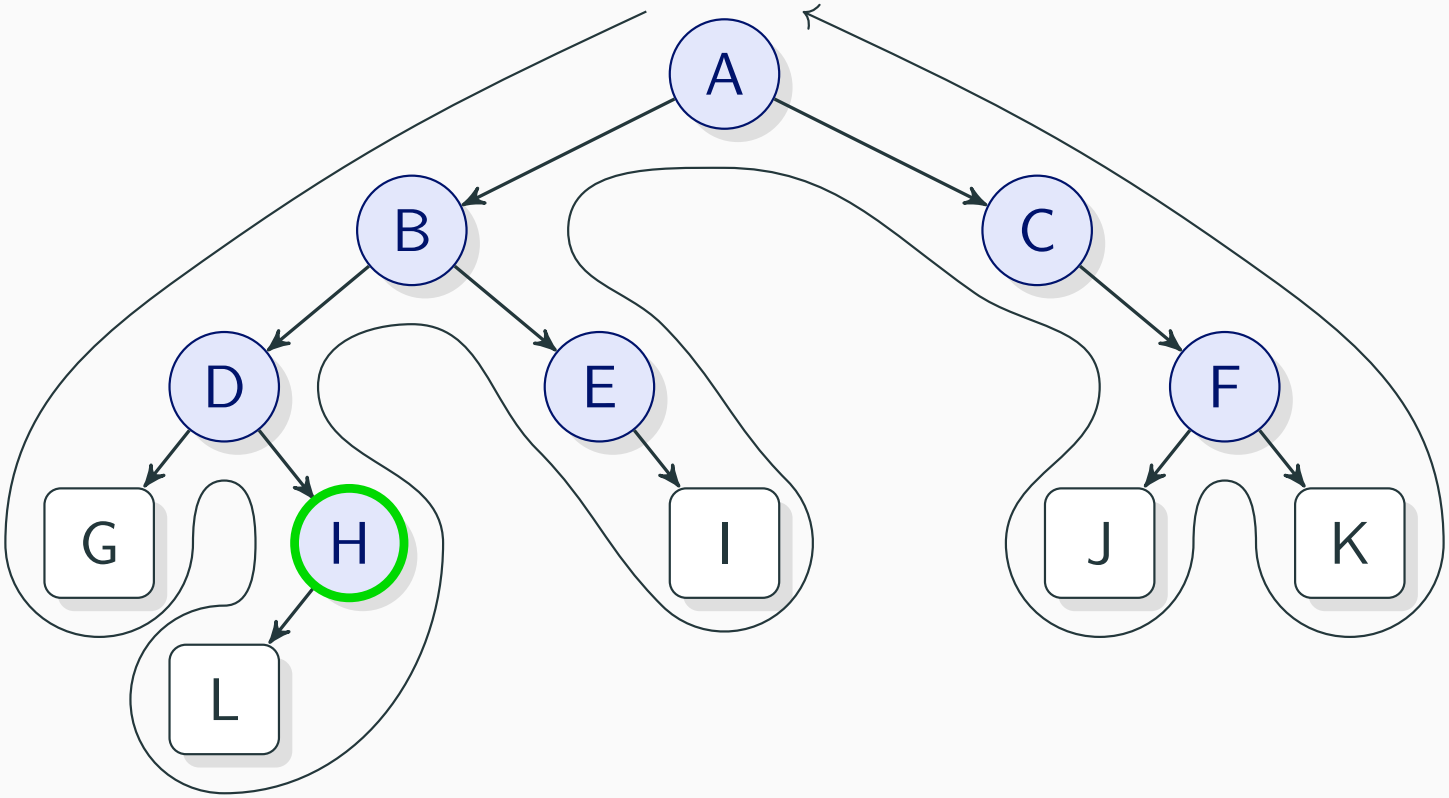
L'exécution de `dfs_suffix` se déroule de la façon suivante :



GL

Parcours suffixe

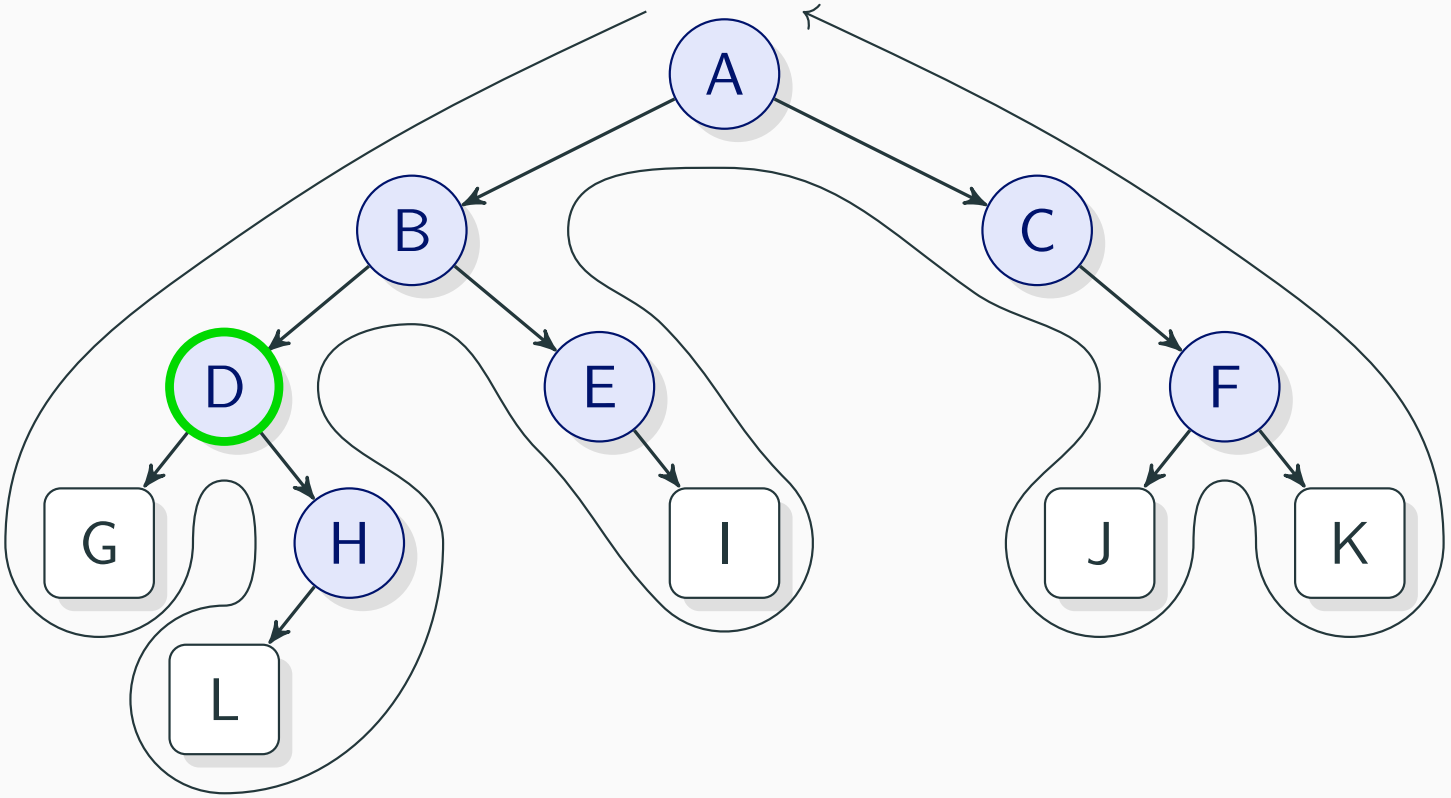
L'exécution de `dfs_suffix` se déroule de la façon suivante :



GLH

Parcours suffixe

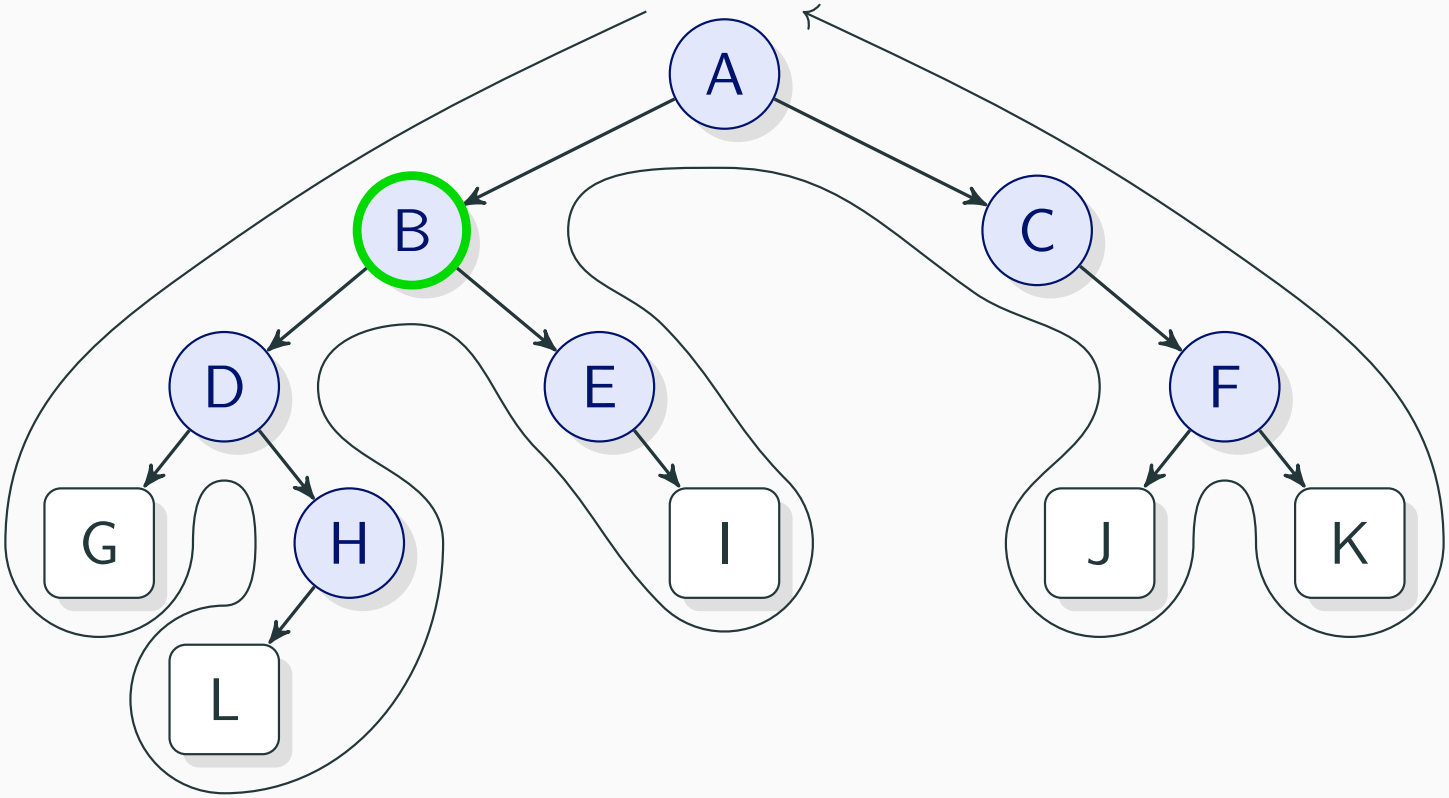
L'exécution de `dfs_suffix` se déroule de la façon suivante :



GLHD

Parcours suffixe

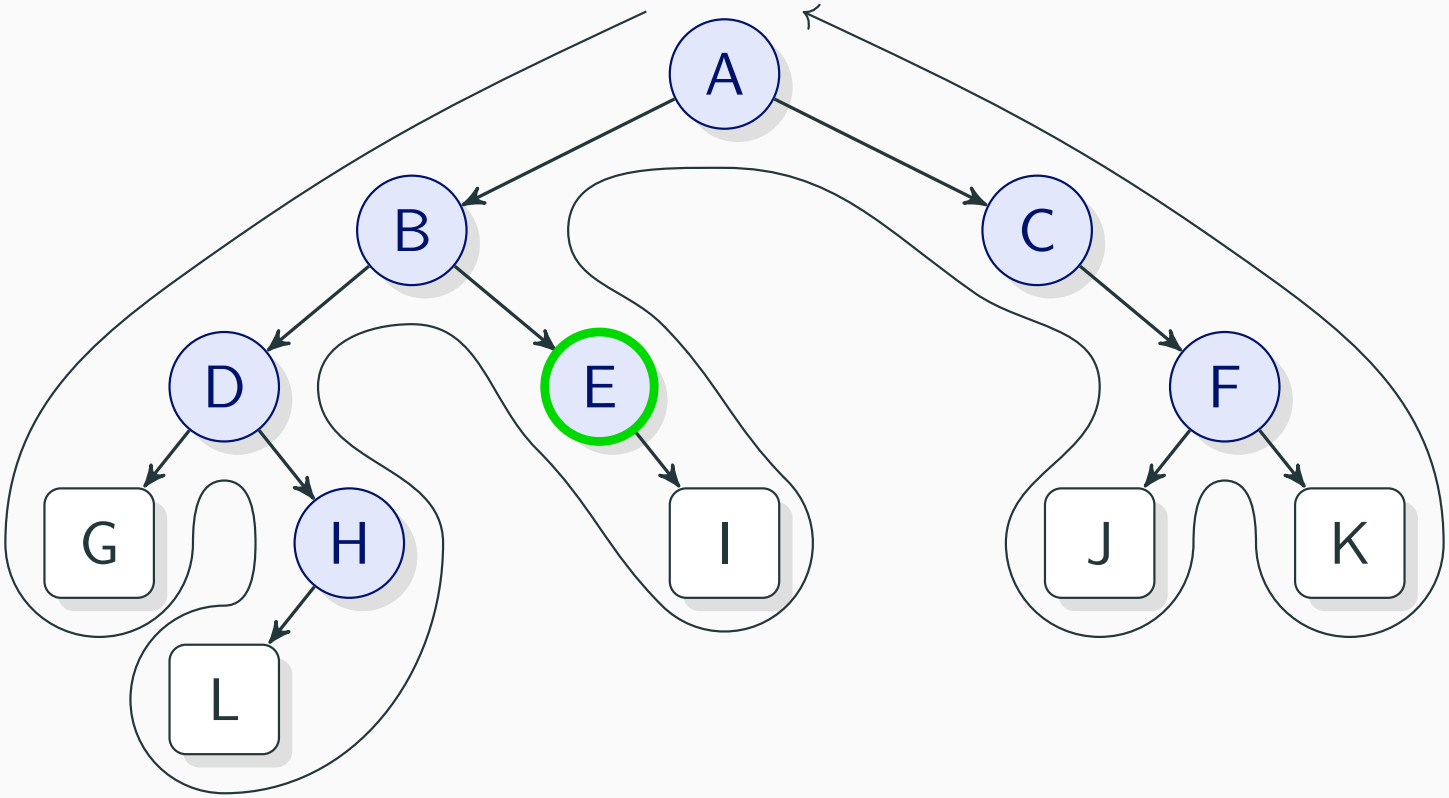
L'exécution de `dfs_suffix` se déroule de la façon suivante :



GLHD

Parcours suffixe

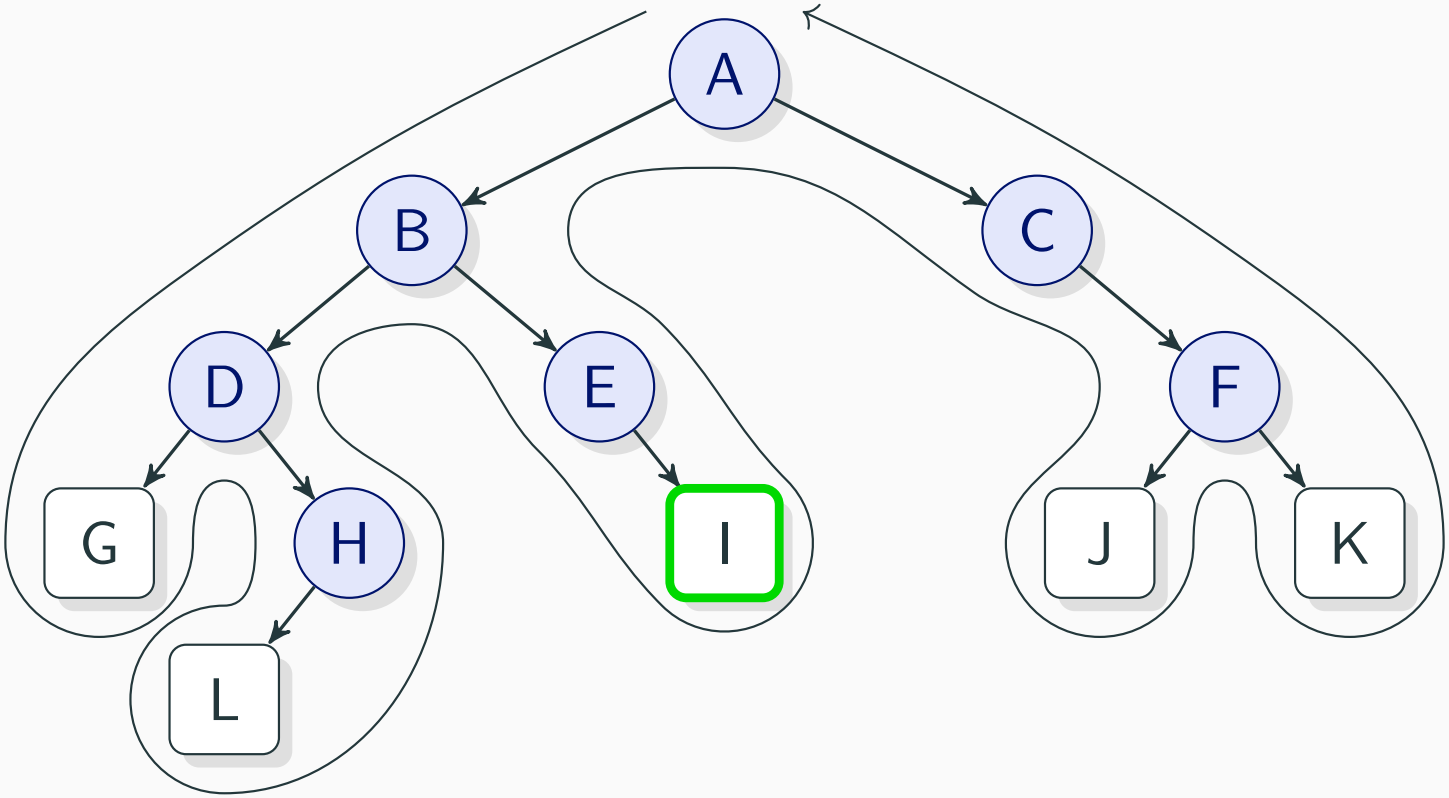
L'exécution de `dfs_suffix` se déroule de la façon suivante :



GLHD

Parcours suffixe

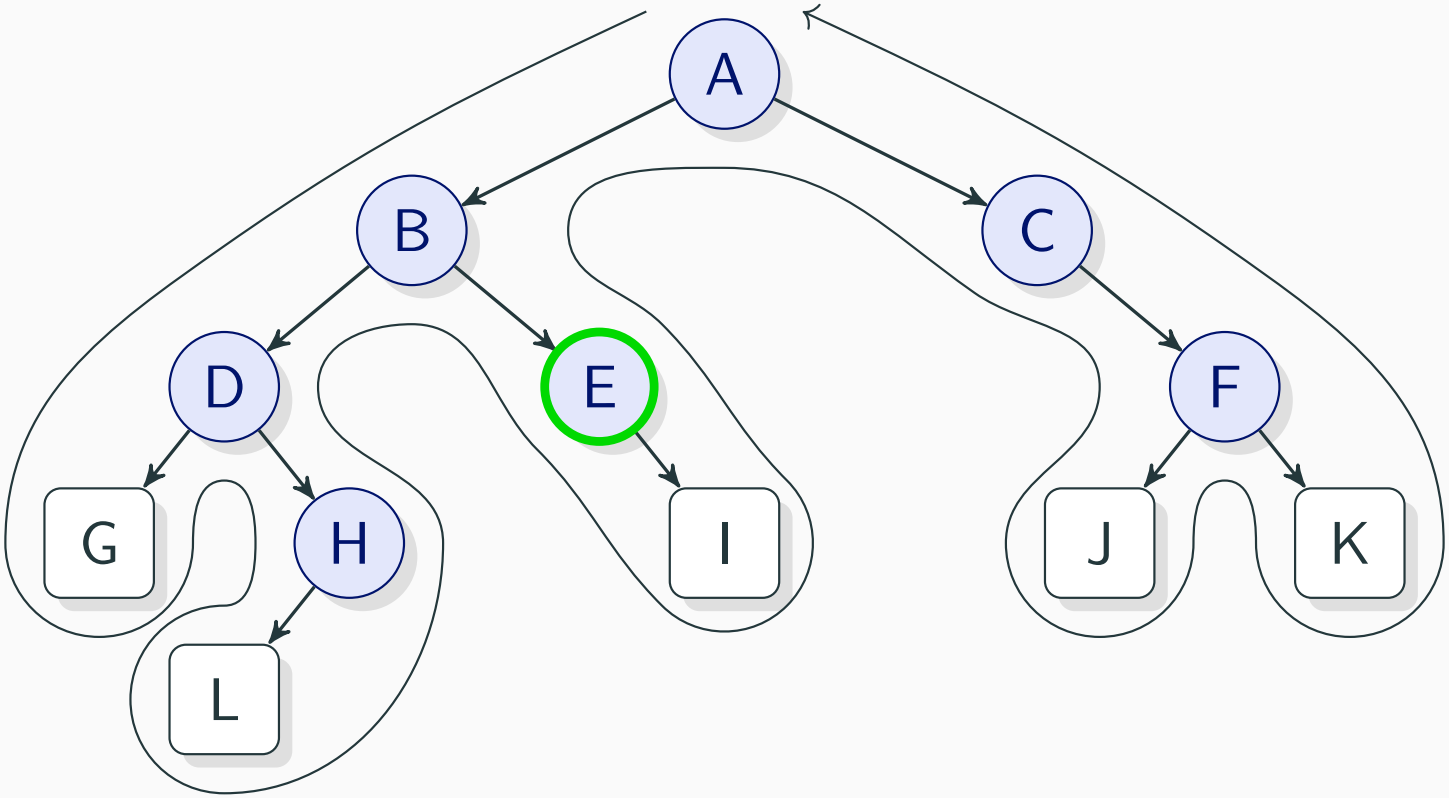
L'exécution de `dfs_suffix` se déroule de la façon suivante :



GLHDI

Parcours suffixe

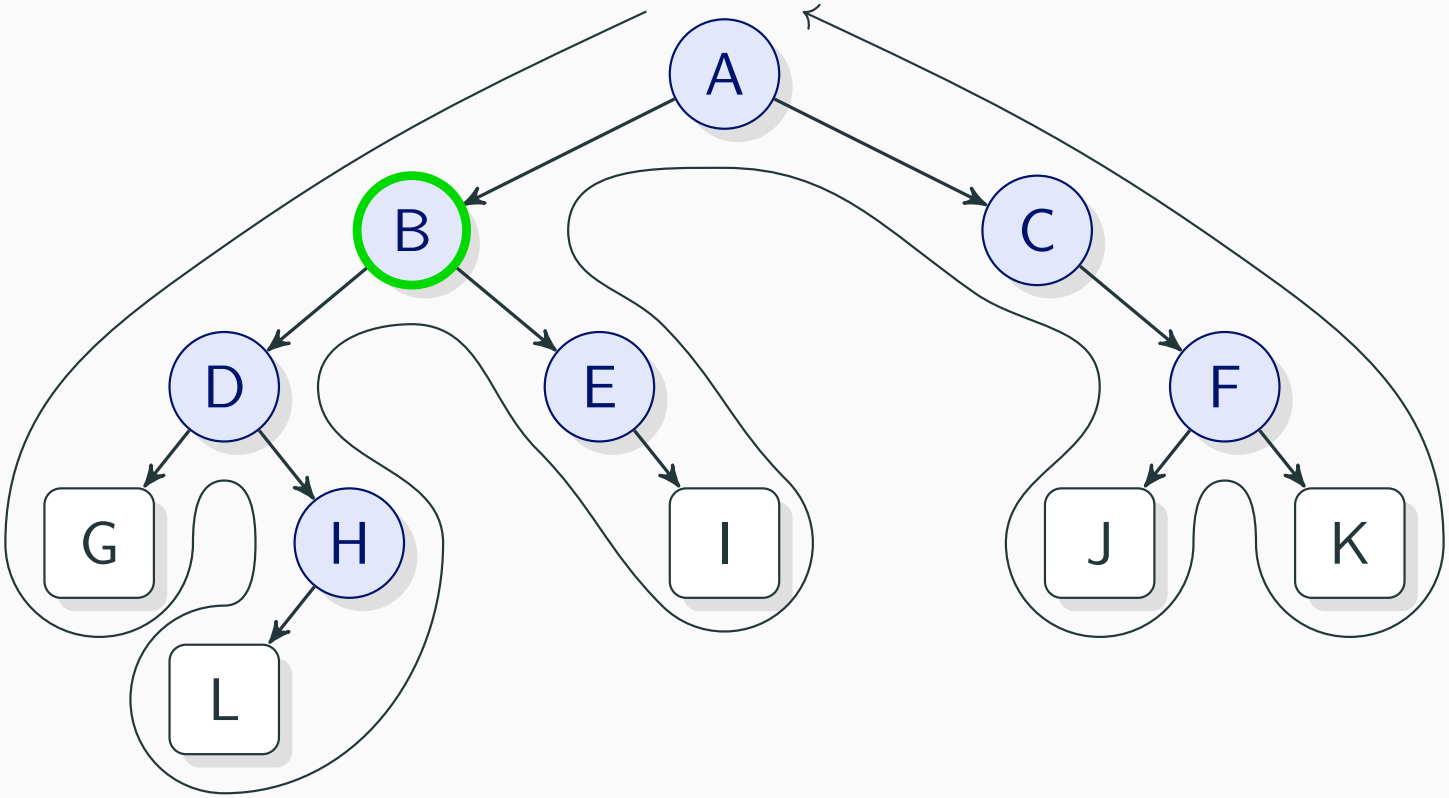
L'exécution de `dfs_suffix` se déroule de la façon suivante :



GLHDIE

Parcours suffixe

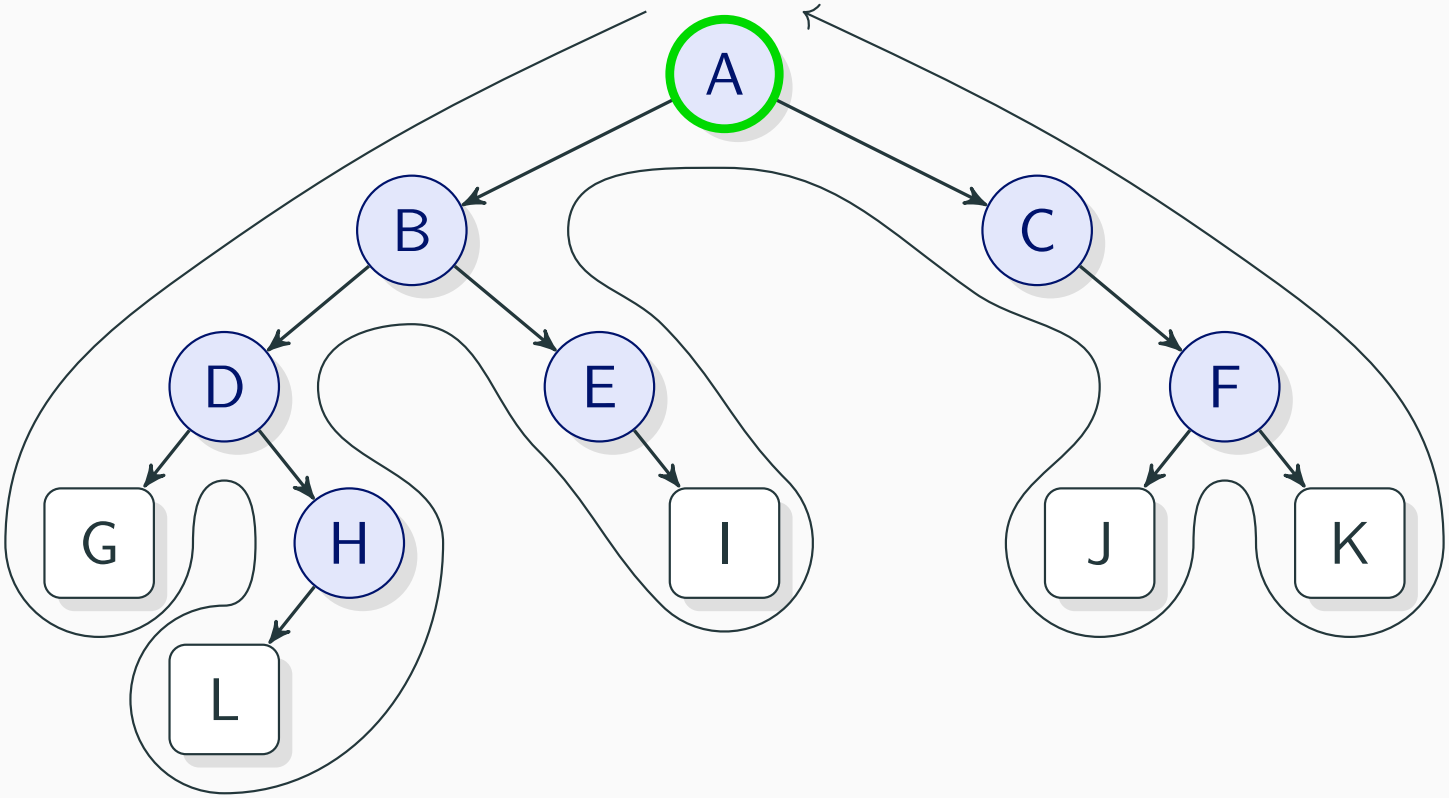
L'exécution de `dfs_suffix` se déroule de la façon suivante :



GLHDIEB

Parcours suffixe

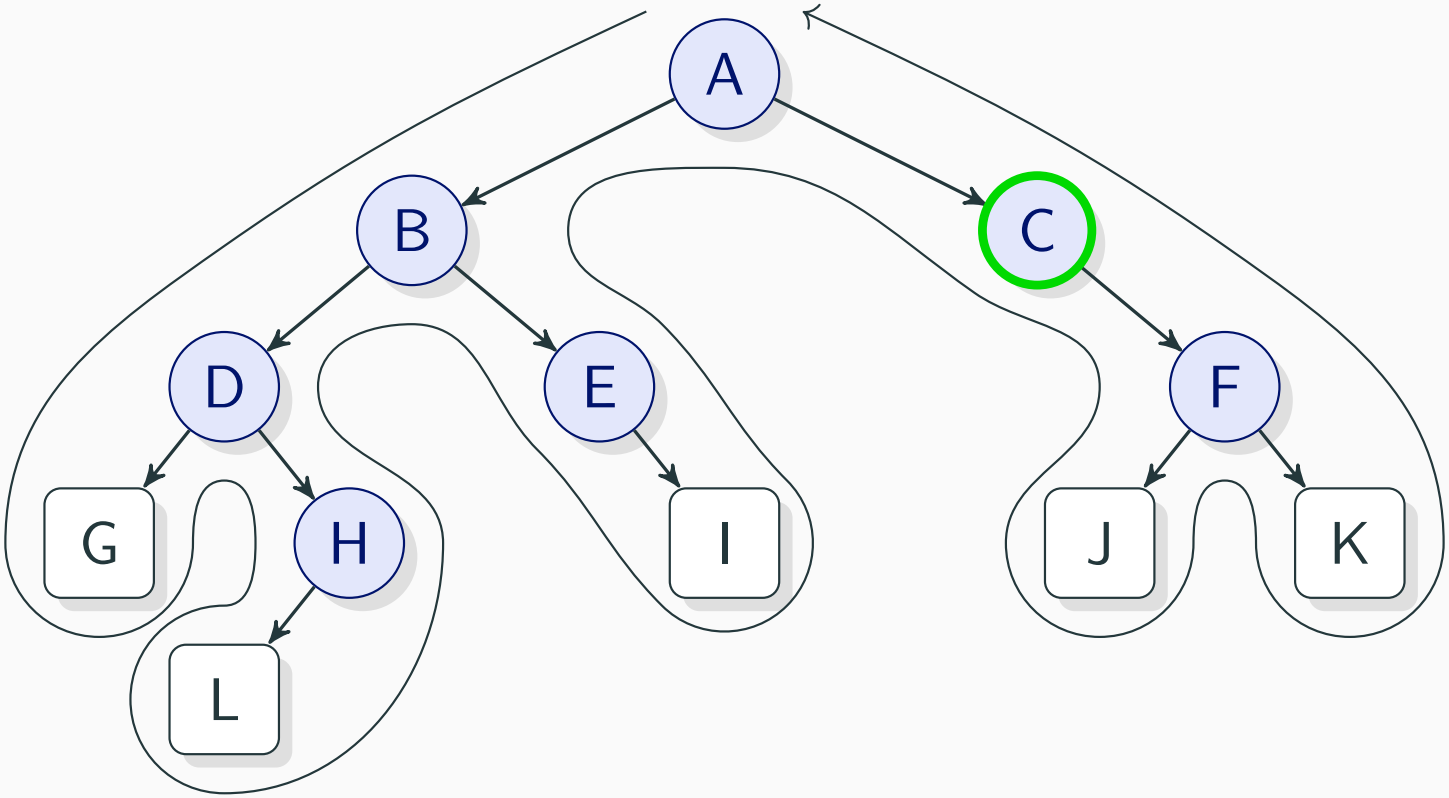
L'exécution de `dfs_suffix` se déroule de la façon suivante :



GLHDIEB

Parcours suffixe

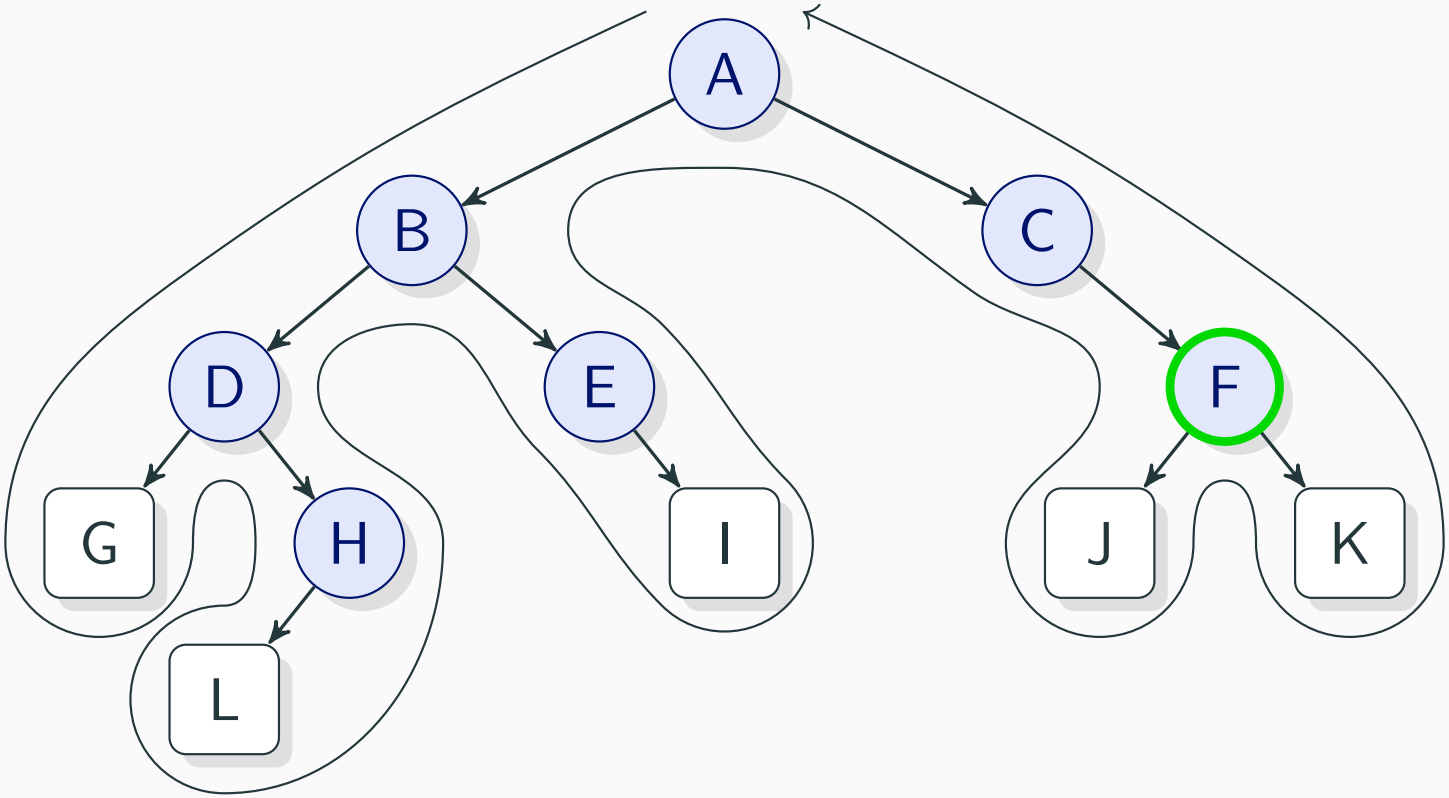
L'exécution de `dfs_suffix` se déroule de la façon suivante :



GLHDIEB

Parcours suffixe

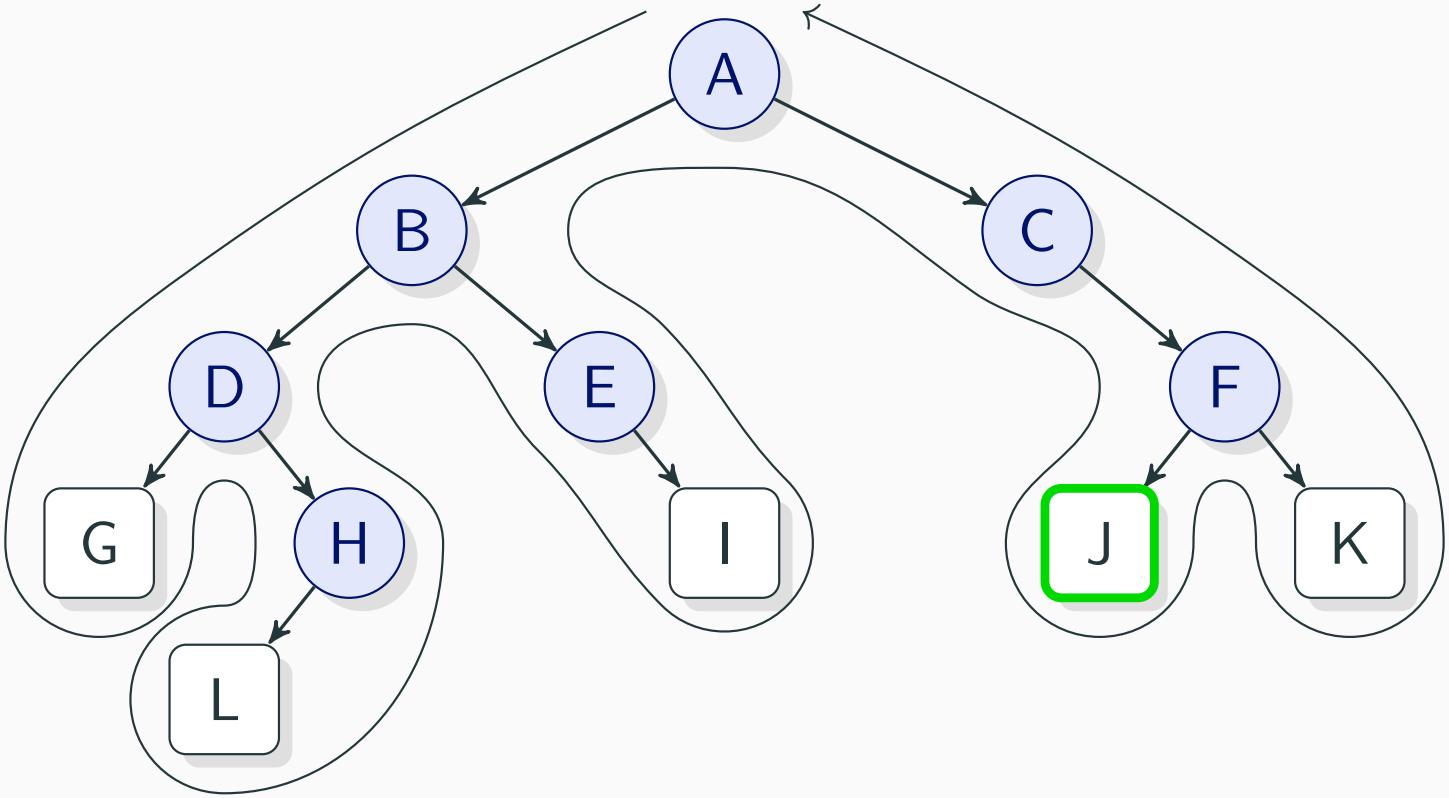
L'exécution de `dfs_suffix` se déroule de la façon suivante :



GLHDIEB

Parcours suffixe

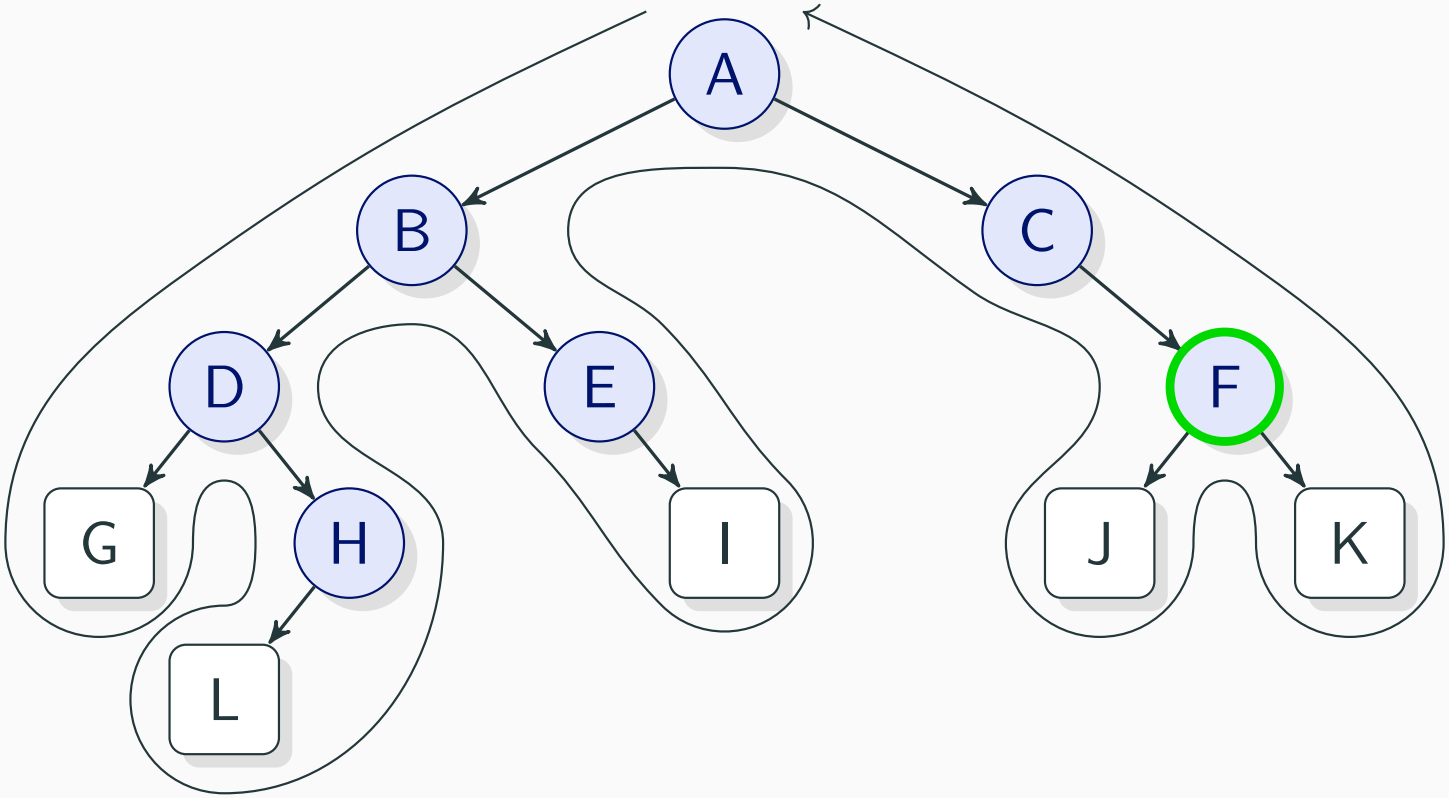
L'exécution de `dfs_suffix` se déroule de la façon suivante :



GLHDIEBJ

Parcours suffixe

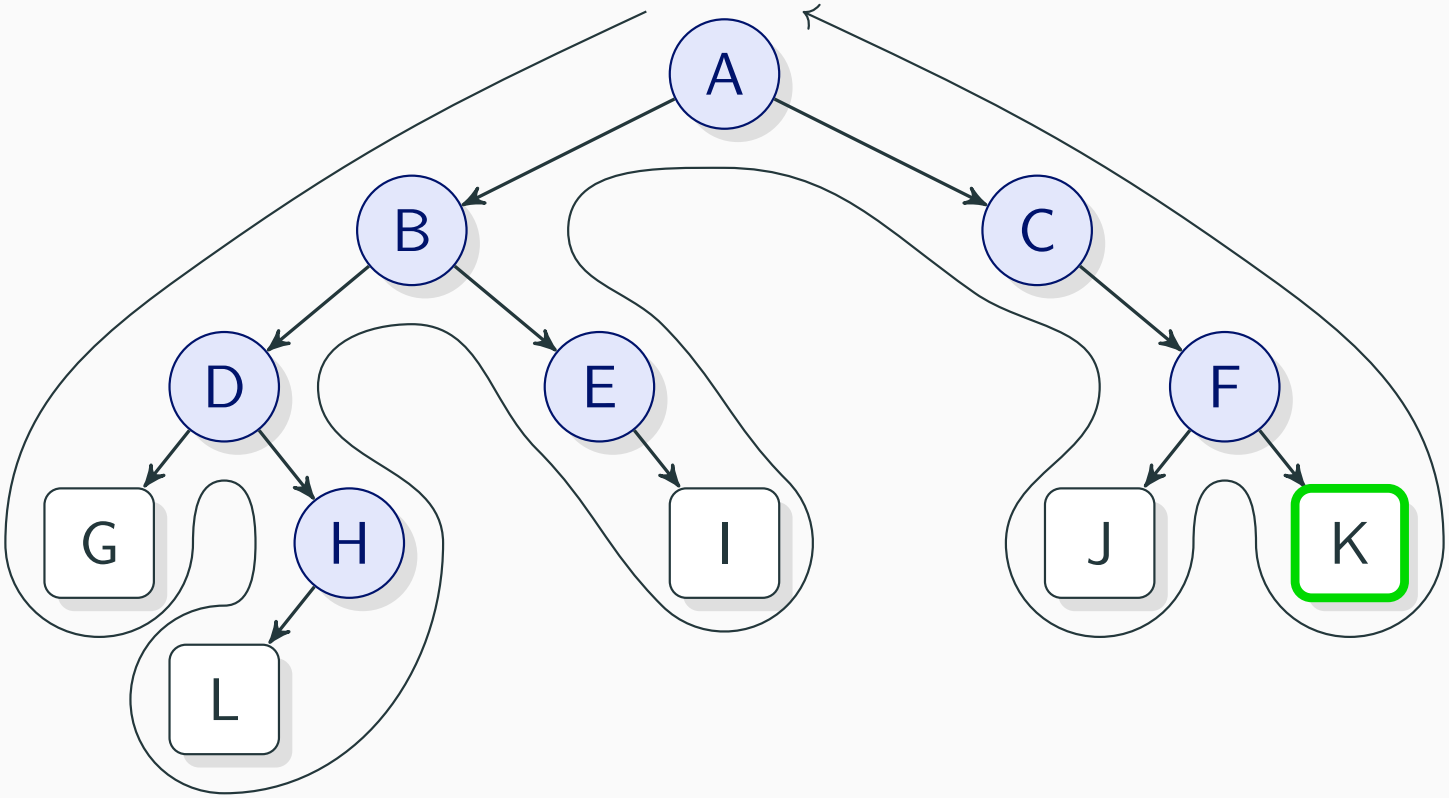
L'exécution de `dfs_suffix` se déroule de la façon suivante :



GLHDIEBJ

Parcours suffixe

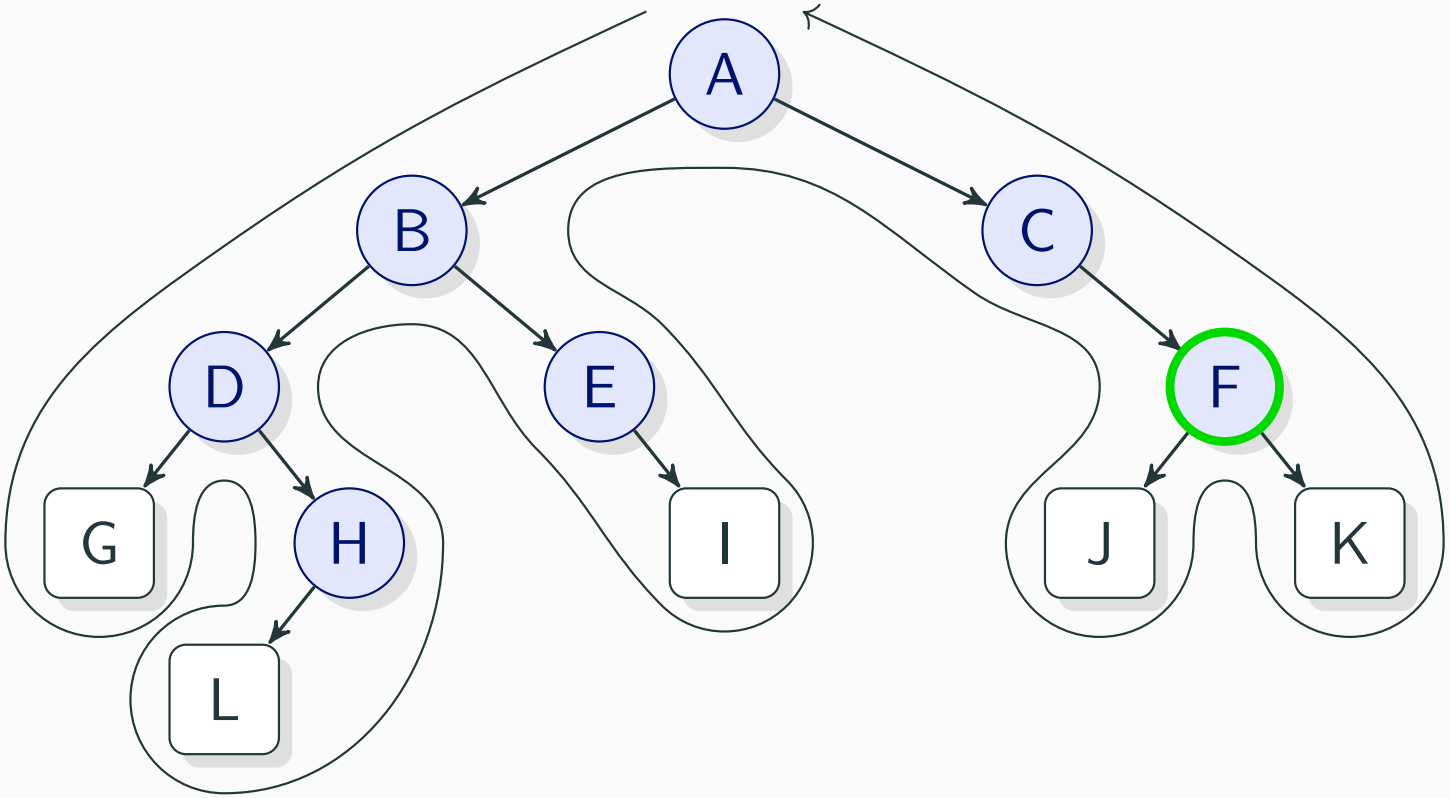
L'exécution de `dfs_suffix` se déroule de la façon suivante :



GLHDIEBJK

Parcours suffixe

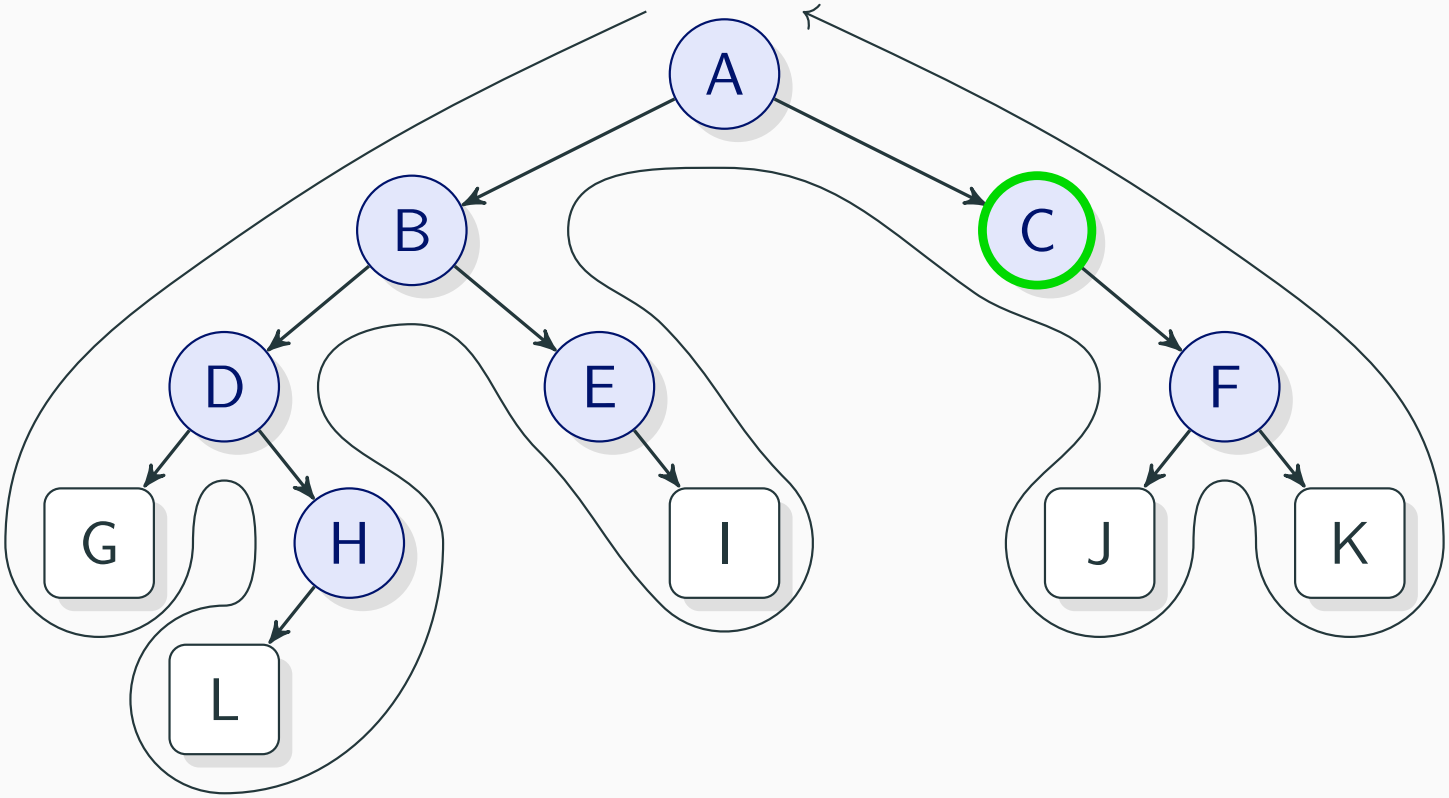
L'exécution de `dfs_suffix` se déroule de la façon suivante :



GLHDIEBJKF

Parcours suffixe

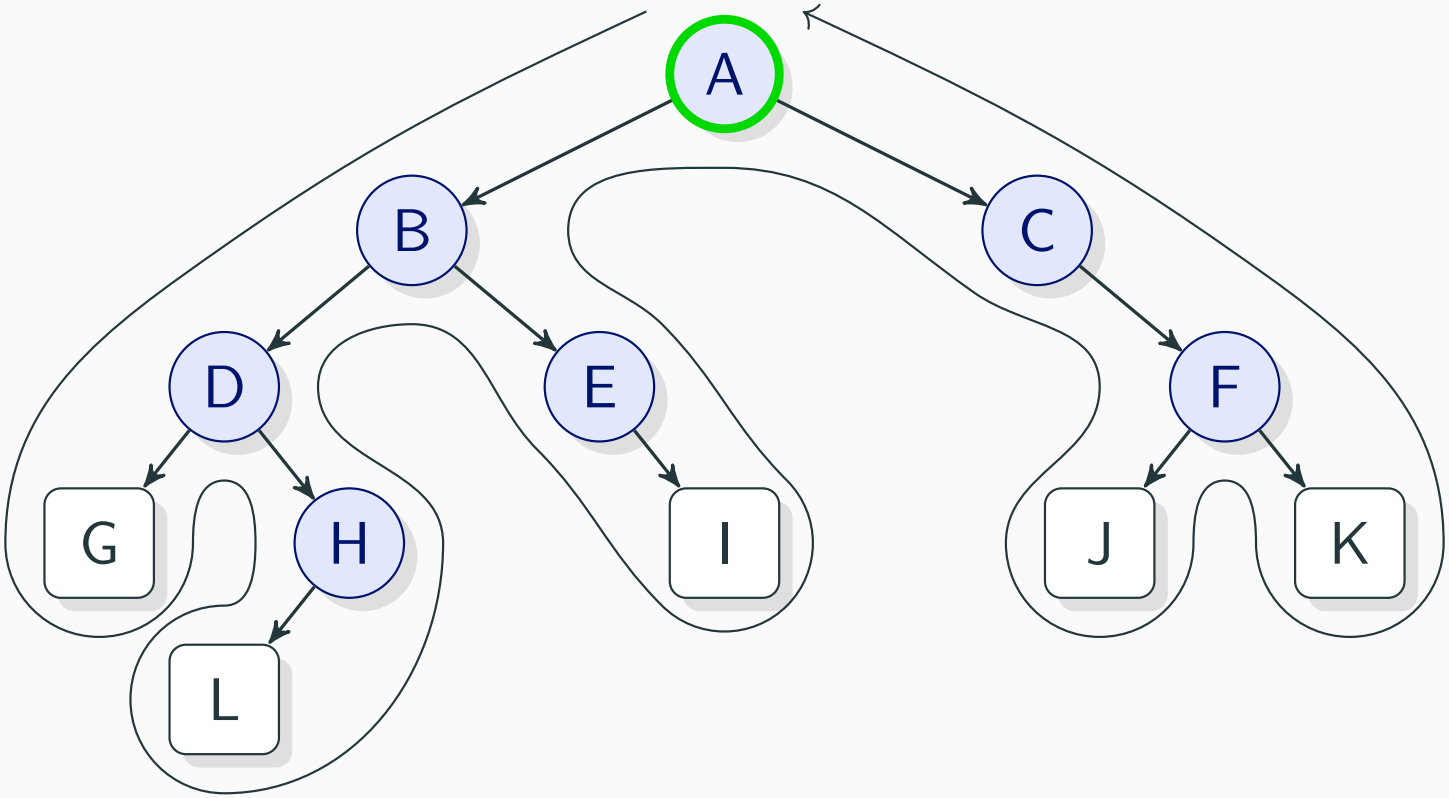
L'exécution de `dfs_suffix` se déroule de la façon suivante :



GLHDIEBJKFC

Parcours suffixe

L'exécution de `dfs_suffix` se déroule de la façon suivante :



GLHDIEBJKFCA

Parcours infixe

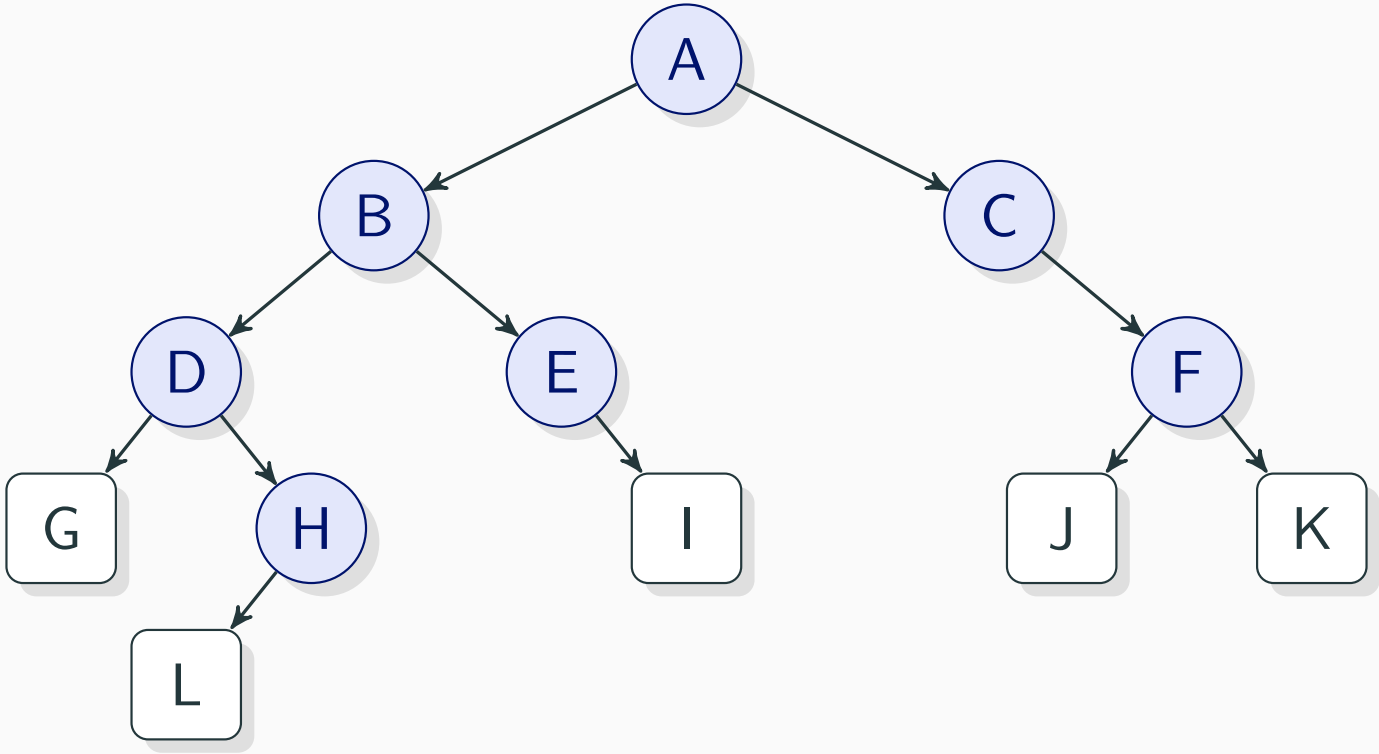
Dans un parcours en profondeur *infixe*,

un nœud est traité *entre* ses enfants


```
# let rec dfs_infix f = function
    | Nil -> ()
    | Node (e, lchild, rchild)
        -> dfs_infix f lchild;
           f e;
           dfs_infix f rchild;;

val dfs_infix : ('a -> 'b) -> 'a arbre -> unit = <fun>
```

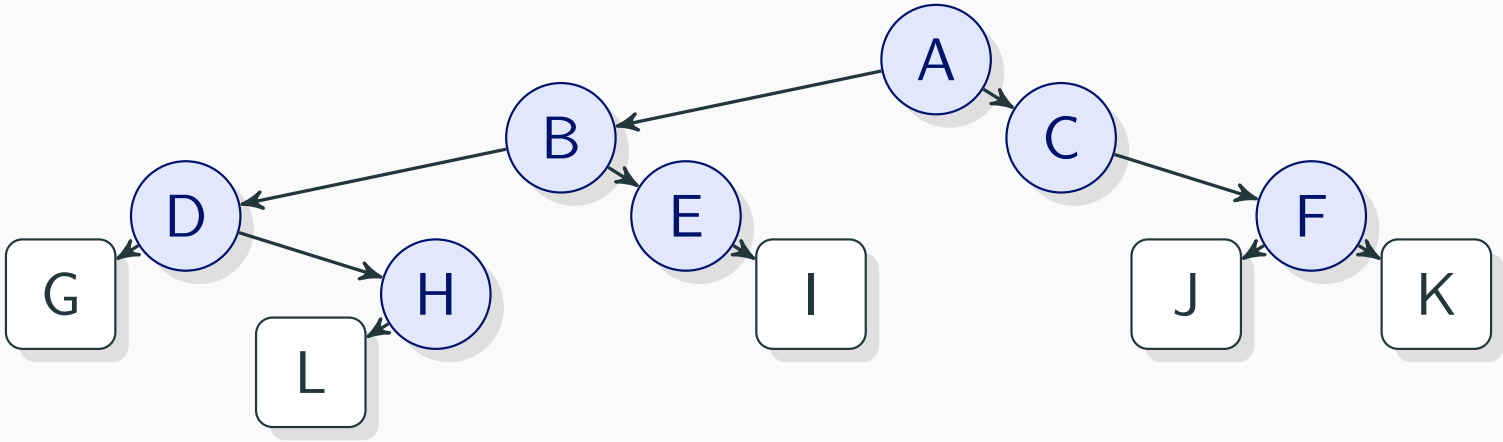
Parcours infixe

L'exécution de `dfs_infix` se déroule de la façon suivante :



Parcours infixe

L'exécution de `dfs_infix` se déroule de la façon suivante :



Le résultat correspond à un « tassement » de l'arbre

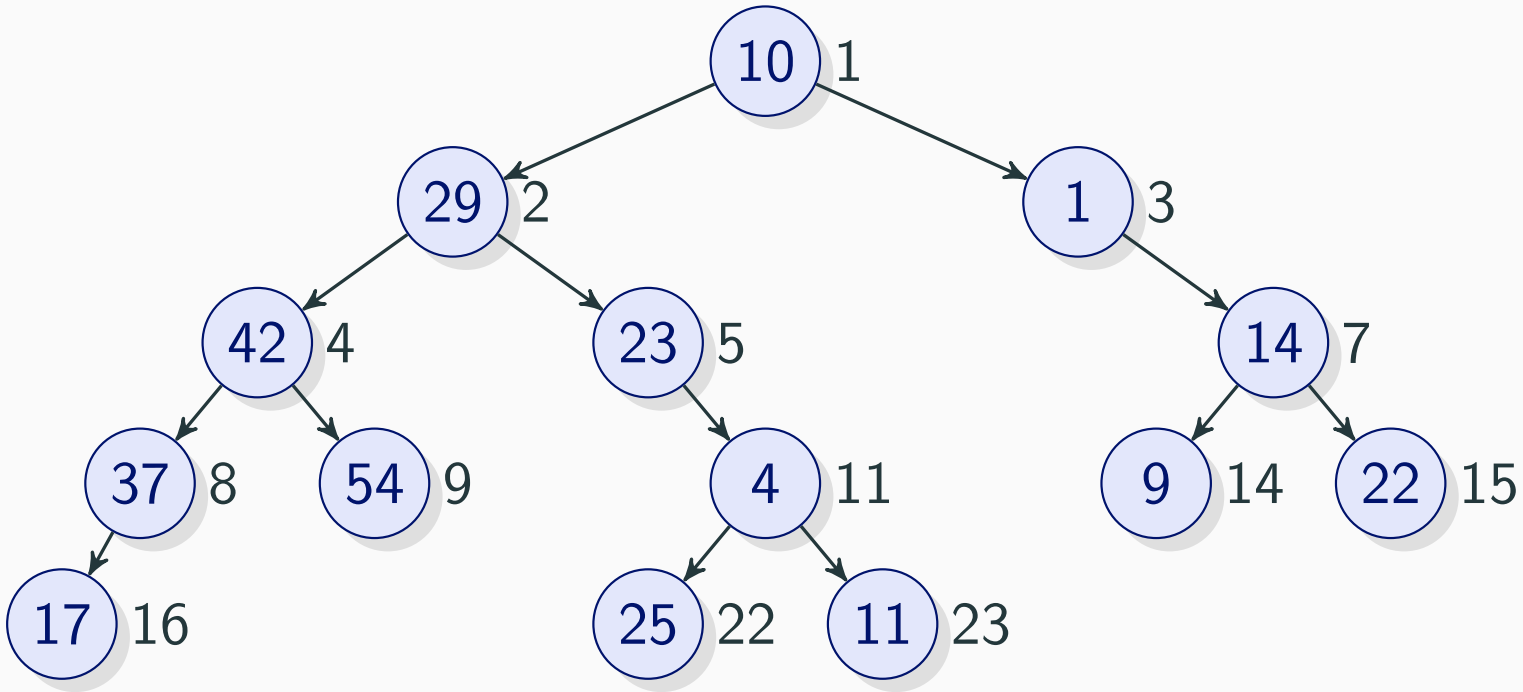
GDLHBEIACJFK

Numérotation de Sosa-Stradonitz

- la racine est numérotée 1 ;
- si un nœud est numéroté k :
  - son enfant gauche est numéroté $2k$
  - son enfant droit est numéroté $2k+1$
  - (son père portera donc le numéro $\lfloor k/2 \rfloor$).

Numérotation de Sosa-Stradonitz

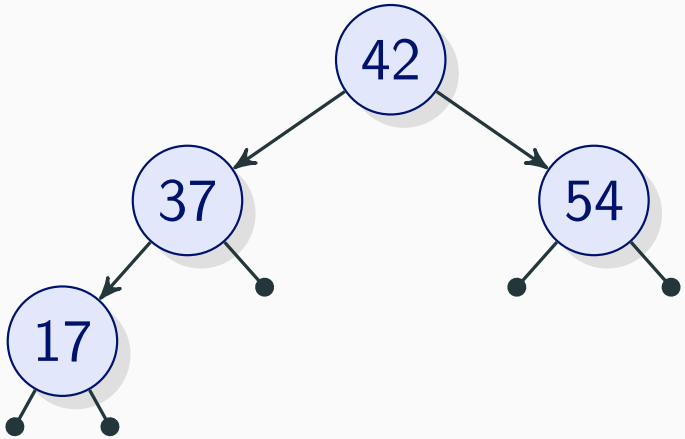
- la racine est numérotée 1 ;
- si un nœud est numéroté k :
  - son enfant gauche est numéroté $2k$
  - son enfant droit est numéroté $2k+1$
  - (son père portera donc le numéro $\lfloor k/2 \rfloor$).

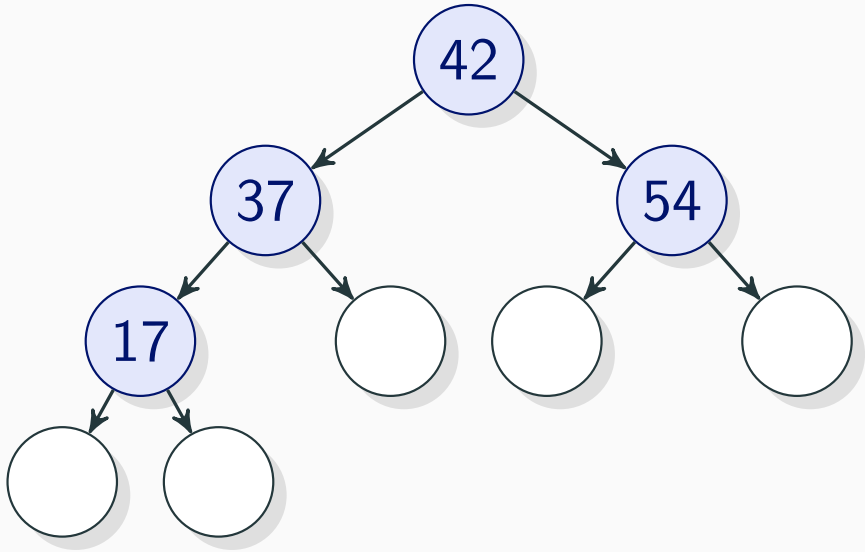


Numérotation de Sosa-Stradonitz

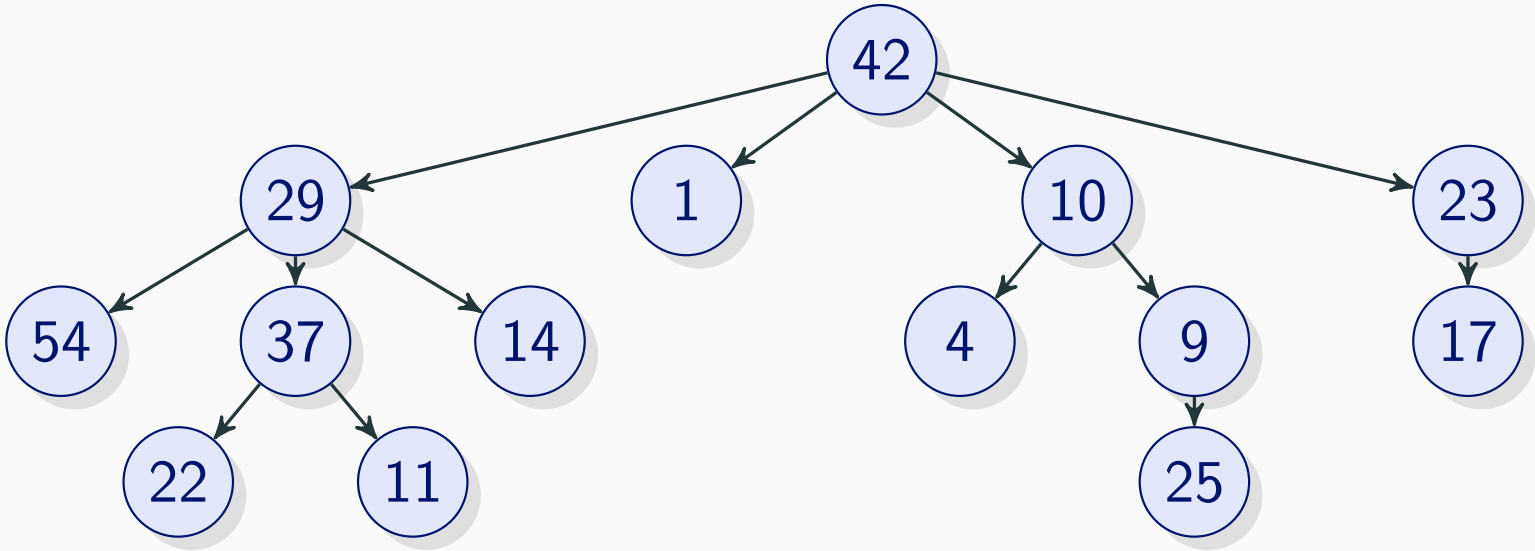
- numéro distinct pour chaque nœud
- l'écriture binaire de $k \rightarrow$ chemin menant de la racine au nœud
- n'importe quel entier strictement positif désigne bien une position
- profondeur $p \iff k \in [\![2^p \,..\, 2^{p+1} - 1]\!]$
- nœud numéroté $k \implies$ profondeur $\lfloor \log_2(k) \rfloor$
- numéros croissants $\rightarrow$ parcours hiérarchique de l'arbre

Lien binaires / binaires stricts

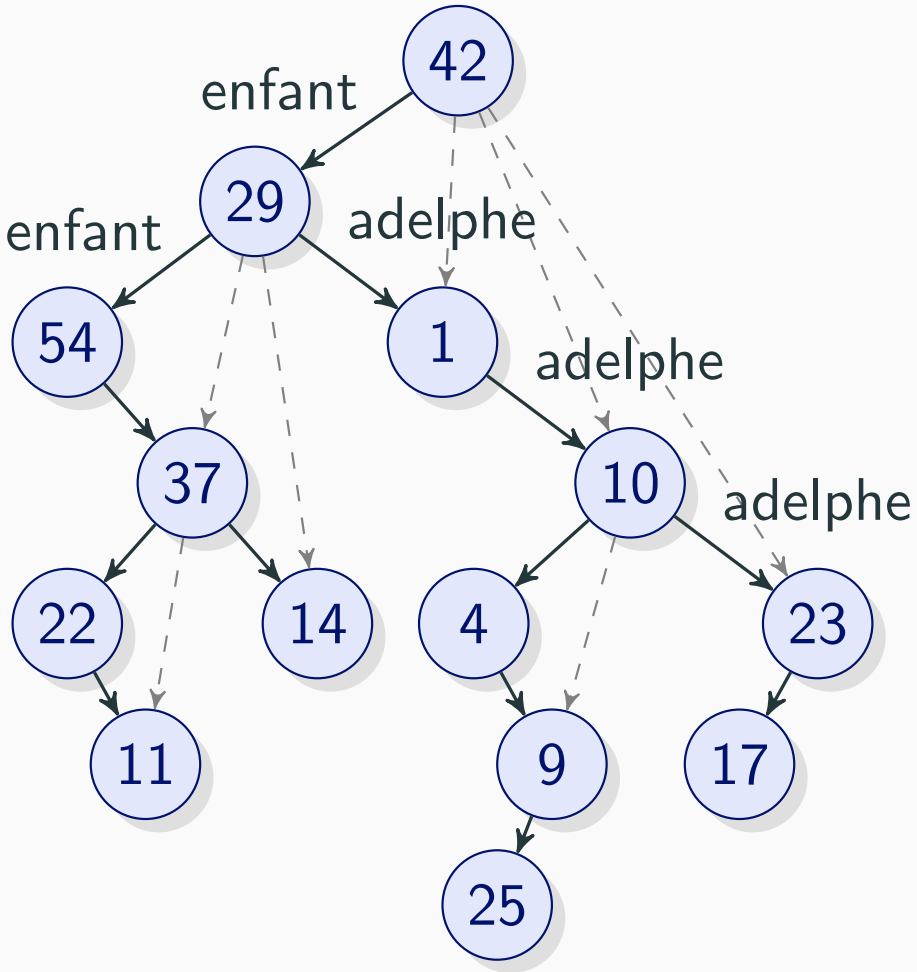




Lien avec les arbres quelconques



Lien avec les arbres quelconques



Arbres fils-frère (enfant-adelphe)

Pour calculer la hauteur :

Arbres fils-frère (enfant-adelphe)

Pour calculer la hauteur :

```
# let rec real_height = function
    | Nil -> -1
    | Node (fchild, _, fsibling)
        -> max (1 + real_height fchild)
               (real_height fsibling);;

val real_height : 'a tree -> int = <fun>
```

Arbres fils-frère (enfant-adelphe)

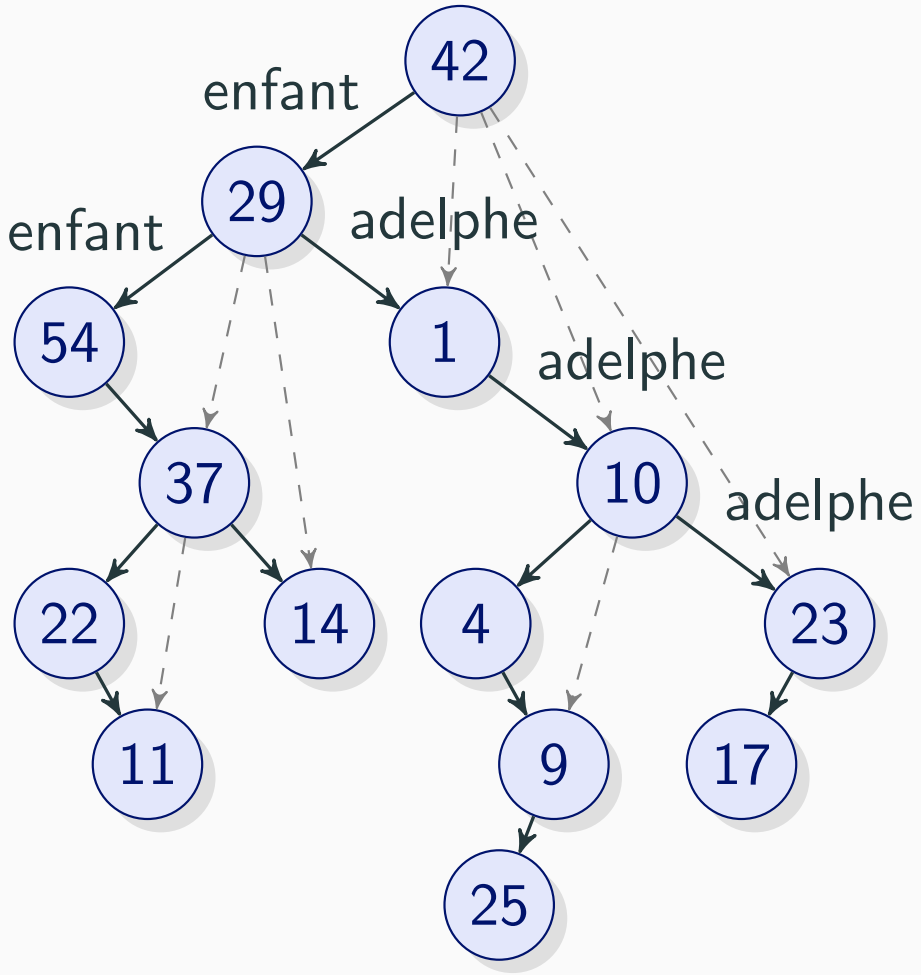
Pour l'arité de la racine :

Arbres fils-frère (enfant-adelphe)

Pour l'arité de la racine :


```
# let real_root_arity = function
    | Nil -> failwith "Empty tree"
    | Node (Nil, _, _) -> 0
    | Node (fchild, _, _)
        -> let rec count_siblings = function
              | Nil -> 0
              | Node (_, _, fsibling)
                  -> 1 + count_siblings fsibling
           in count_siblings fchild;;

val real_root_arity : 'a tree -> int = <fun>
```

Parcours dans les arbres fils-frère (enfant-adelphe)



Parcours dans les arbres fils-frère (enfant-adelphe)


```
# let real_bfs f tree =
    let queue = Queue.create () in
      Queue.add tree queue;
      while not (Queue.is_empty queue) do
        let rec visit_children = function
          | Nil -> ();
          | Node (fchild, e, fsibling)
              -> f e;
                 Queue.push fchild queue;
                 visit_children fsibling
        in visit_children (Queue.pop queue)
      done;;

val real_bfs : ('a -> unit) -> 'a tree -> unit = <fun>
```

Retour sur la complexité

```
# let rec size = function
    | Nil -> 0
    | Node (_, lchild, rchild)
        -> 1 + size lchild + size rchild;;

val size : 'a tree -> int = <fun>
```

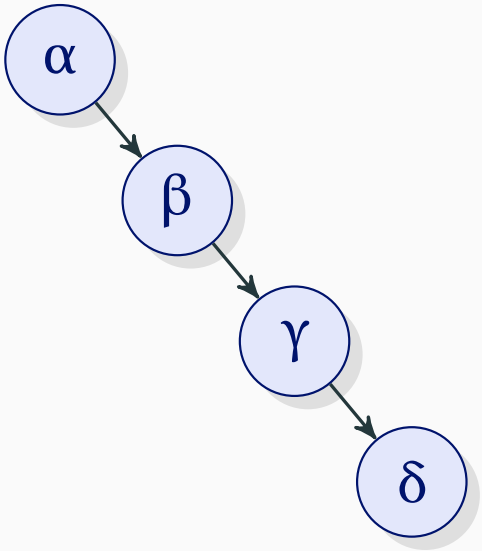
Retour sur la complexité

```
# let rec leftmost = function
    | Nil -> failwith "Empty"
    | Node (e, Nil, _) -> e
    | Node (_, lchild, _) -> leftmost lchild;;

val leftmost : 'a tree -> 'a = <fun>
```

Retour sur la complexité


```
# let rec list_of_tree = function
    | Nil -> []
    | Node (e, lchild, rchild)
        -> list_of_tree lchild
             @ e :: list_of_tree rchild;;

val list_of_tree : 'a tree -> 'a list = <fun>
```

Retour sur la complexité



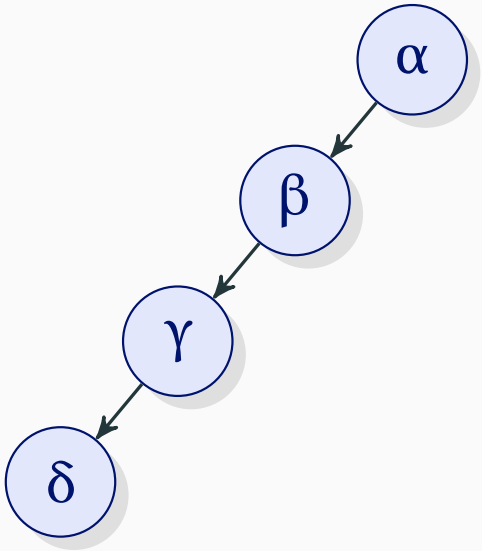
`[]@(α::[β;γ;δ])` 2 opérations élémentaires

`[]@(β::[γ;δ])` 2 opérations élémentaires

`[]@(γ::[δ])` 2 opérations élémentaires

`[]@(δ::[])` 2 opérations élémentaires

Retour sur la complexité



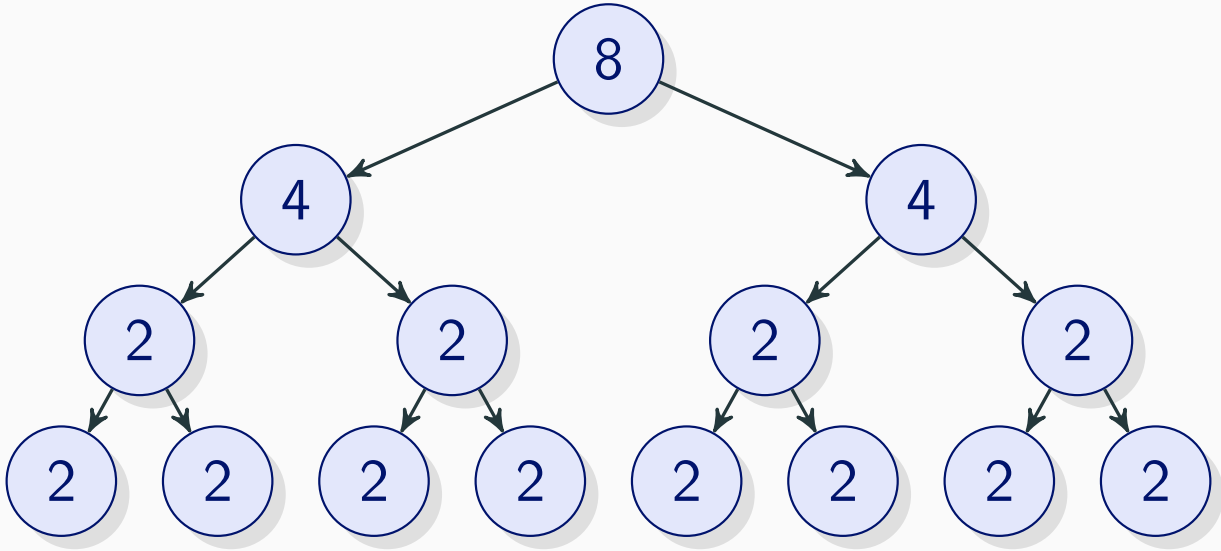
`[δ;γ;β]@(α::[])` 4 opérations élémentaires

`[δ;γ]@(β::[])` 3 opérations élémentaires

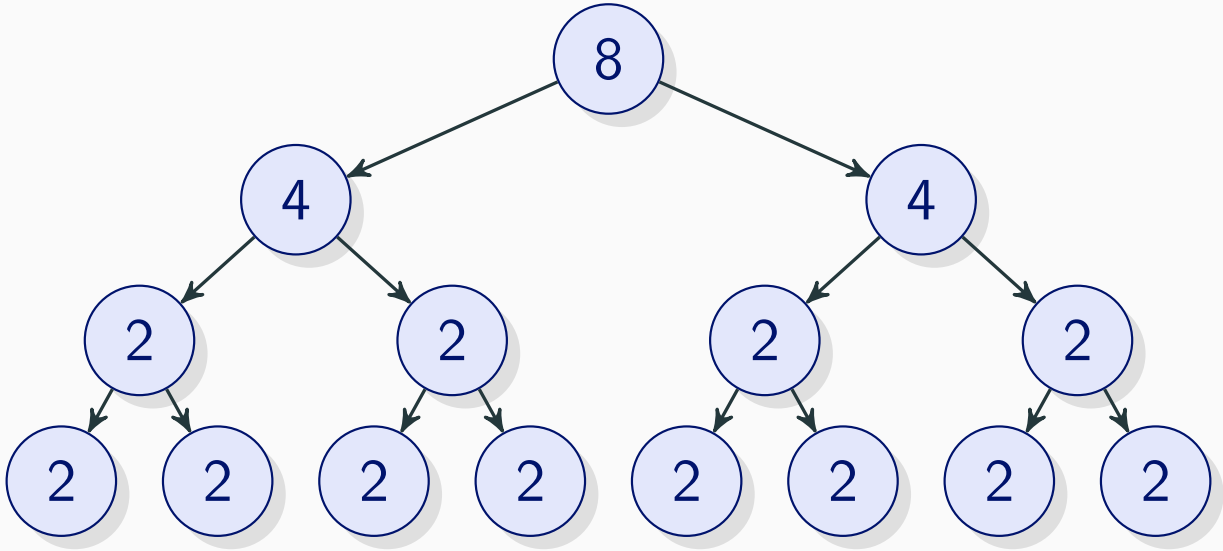
`[δ]@(γ::[])` 2 opérations élémentaires

`[]@(δ::[])` 2 opérations élémentaires

Retour sur la complexité



Retour sur la complexité



$$2\times 2^h + \sum_{p=1}^{h} 2^{h-p}\times 2^p = 2^h + \sum_{p=0}^{h} 2^{h-p}\times 2^p = 2^h + \sum_{p=0}^{h} 2^h = 2^h + (h+1)\times 2^h$$

$$= (h+2)\times 2^h$$

Retour sur la complexité


```
# let list_of_tree t =
    let rec to_list lst = function
      | Nil -> lst
      | Node (e, lchild, rchild)
          -> to_list (e::(to_list lst rchild)) lchild
    in to_list [] t;;

val list_of_tree : 'a tree -> 'a list = <fun>
```

Et en C ?


```c
struct tree {
    int value;
    struct tree* lchild;
    struct tree* rchild;
}

typedef struct tree Tree;
```

Un pointeur `NULL` replace l'objet `Nil`

Et en C ?

```c
int size(Tree* ptr_tree) {
    if (ptr_tree==NULL) { return 0; }
    return 1 + size(ptr_tree->lchild)
             + size(ptr_tree->rchild);
}
```

Et en C ?


```c
int height(Tree* ptr_tree) {
    if (ptr_tree==NULL) { return -1; }
    int hlc = height(ptr_tree->lchild);
    int hrc = height(ptr_tree->rchild);
    if (hlc>hrc) {
        return hlc;
    } else {
        return hrc;
    }
}
```

Mutabilité des arbres

```c
void incr_leftmost_node(Tree* ptr_tree) {
    if (ptr_tree==NULL) { return; }
    if (ptr_tree->lchild != NULL) {
        incr_leftmost_node(ptr_tree->lchild);
    } else {
        ptr_tree->val = ptr_tree->val + 1;
    }
}
```

Mutabilité des arbres

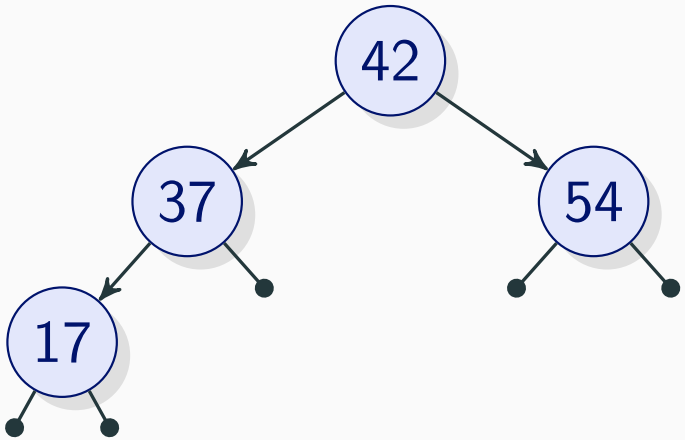
```c
void mirror(Tree* ptr_tree) {
    if (ptr_tree==NULL) { return; }
    Tree* ptr_tmp = ptr_tree->lchild;
    ptr_tree->lchild = ptr_tree->rchild;
    ptr_tree->rchild = ptr_tmp;
    mirror(ptr_tree->lchild);
    mirror(ptr_tree->rchild);
}
```

Construction

```c
Tree* new_node(int value, Tree* lchild, Tree* rchild) {
    Tree* ptr_tree = (Tree*)malloc(sizeof(Tree));
    if (ptr_tree==NULL) { return NULL; }
    ptr_tree->value = value;
    ptr_tree->lchild = lchild;
    ptr_tree->rchild = rchild;
    return ptr_tree;
}
```

Construction

```
Tree* ptr_tree = new_node(42, new_node(37,
                          new_node(17, NULL, NULL), NULL),
                          new_node(54, NULL, NULL));
```



Destruction

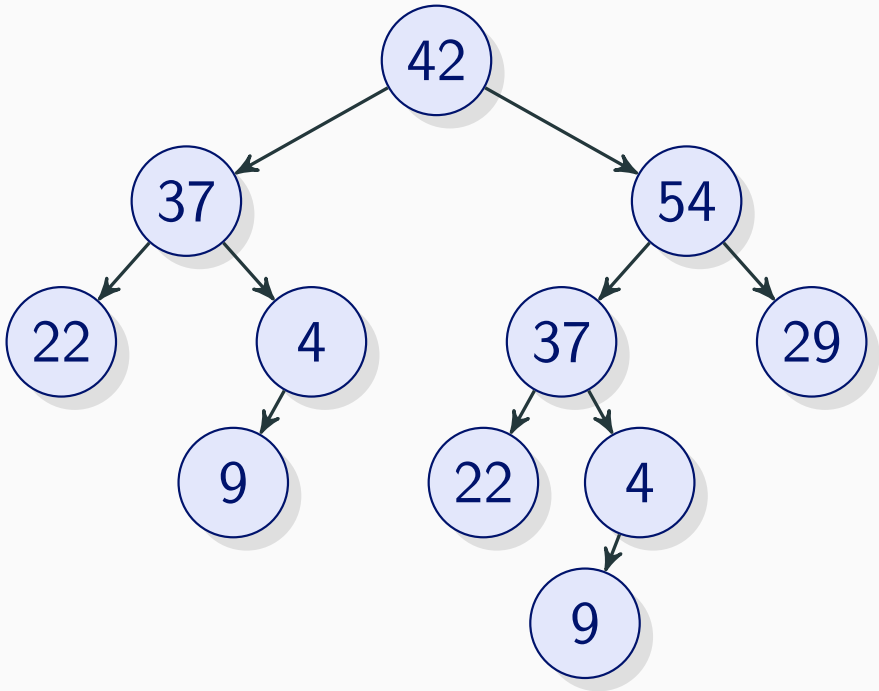
```c
void delete_tree(Tree* ptr_tree) {
    if (ptr_tree == NULL) { return; }
    delete_tree(ptr_tree->lchild);
    delete_tree(ptr_tree->rchild);
    free(tree);
}
```

```c
void delete_left_subtree(Tree* ptr_tree) {
    if (ptr_tree == NULL) { return; }
    delete_tree(ptr_tree->lchild);
    ptr_tree->lchild = NULL;
}
```

Destruction


```c
Tree* delete_leaves(Tree* ptr_tree) {
    if (ptr_tree == NULL) { return; }
    if (ptr_tree->lchild == NULL
          && ptr_tree->rchild == NULL) {
        free(ptr_tree);
        return NULL;
    } else {
        ptr_tree->lchild = delete_leaves(ptr_tree->lchild);
        ptr_tree->rchild = delete_leaves(ptr_tree->rchild);
        return ptr_tree;
    }
}
```

Considérations sur le stockage

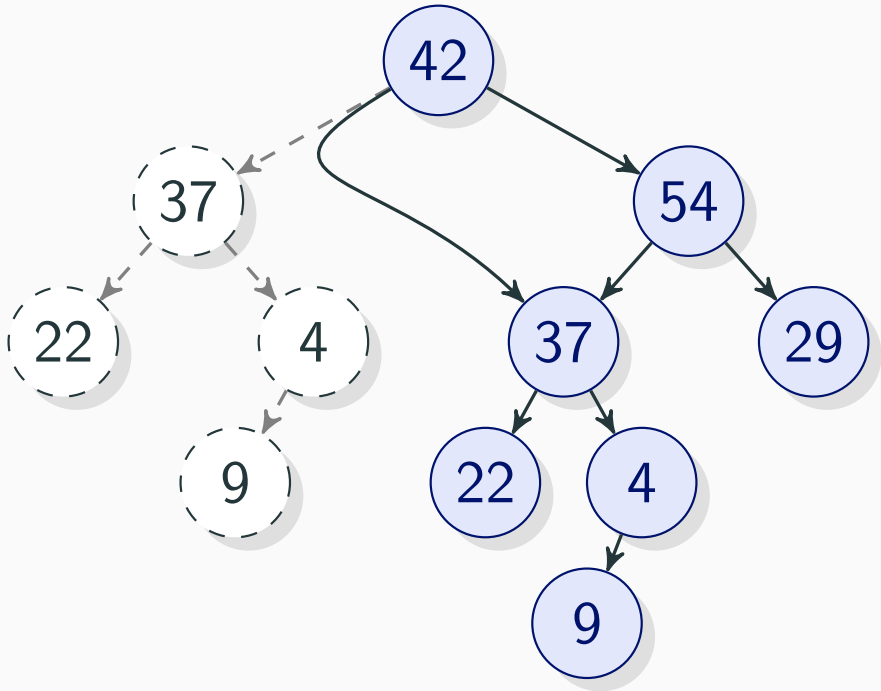


Considérations sur le stockage

```
let tree =
  let subtree = Node(37, Node(22, Nil, Nil),
                         Node(4, Node(9, Nil, Nil), Nil)
  in Node(42, subtree, Node(54, subtree, Node(29, Nil, Nil));
```

```
Tree* ptr_tree = new_node(37, new_node(22, NULL, NULL),
                    new_node(4, new_node(9, NULL, NULL),
                                NULL));
ptr_tree = new_node(42, ptr_tree, new_node(54, ptr_tree,
                                    new_node(29, NULL,
                                                 NULL)));
```

Considérations sur le stockage

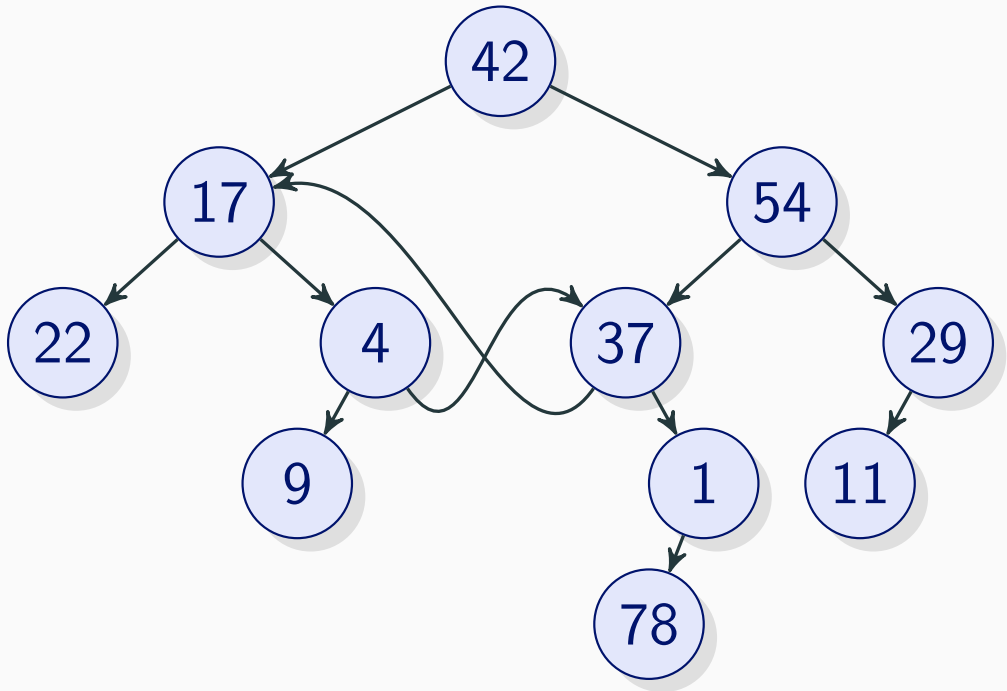


Considérations sur le stockage

```c
Tree* copy(Tree* ptr_tree) {
    if (ptr_tree == NULL) { return NULL; }
    return new_node(ptr_tree->value,
                    copy(ptr_tree->lchild),
                    copy(ptr_tree->rchild));
}
```

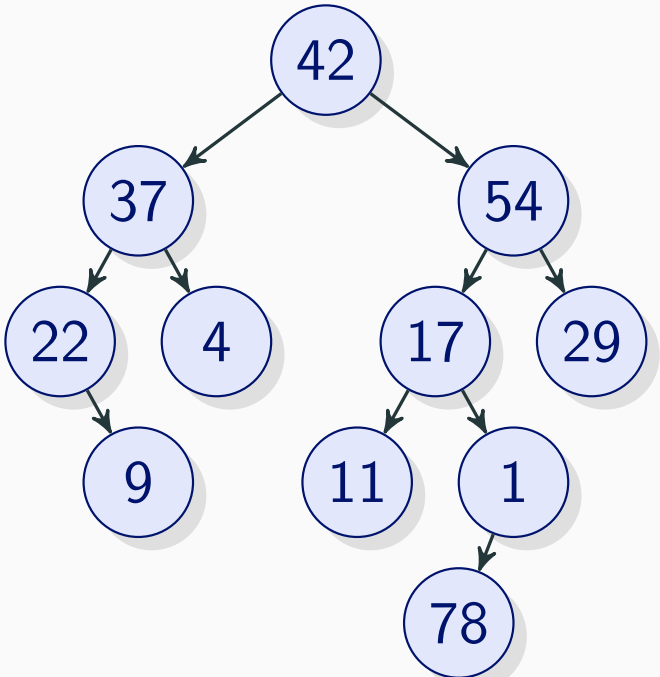
Considérations sur le stockage

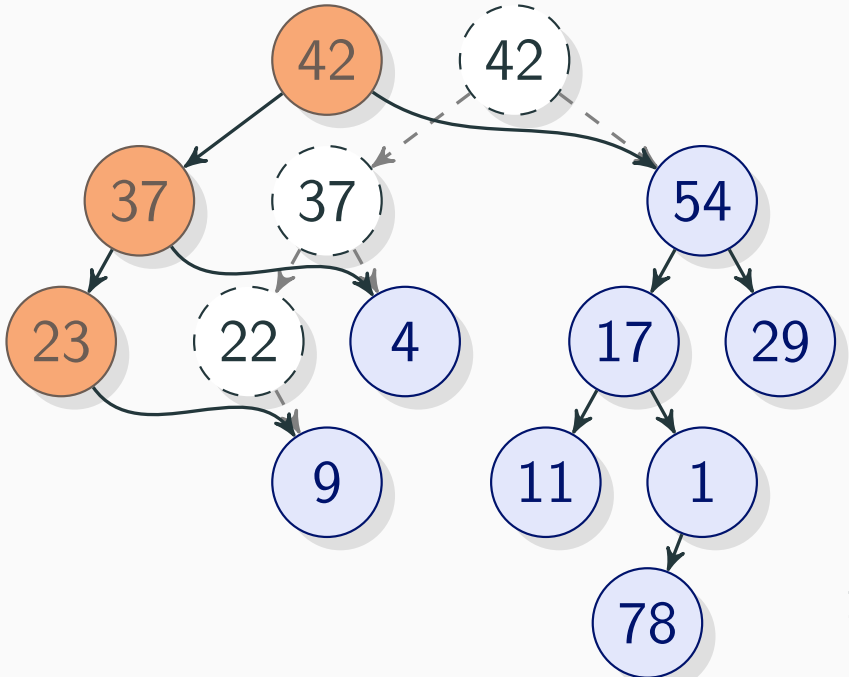
```
let tree =
  let rec subtree_A = Node(17, Node(22, Nil, Nil),
            Node(4, Node(9, Nil, Nil), subtree_B))
  and subtree_B = Node(37, subtree_A,
                    Node(1, Node(78, Nil, Nil), Nil))
  in Node(42, subtree_A, Node(54, subtree_B,
        Node(29, Node(11, Nil, Nil), Nil)))
```



Mutabilité et OCaml

```
# let rec incr_leftmost_node = function
    | Nil -> failwith "Arbre vide"
    | Node (e, Nil, rchild) -> Node (e+1, Nil, rchild)
    | Node (e, lchild, rchild)
        -> Node (e, incr_leftmost_node lchild, rchild);;

val incr_leftmost_node : int tree -> int tree = <fun>
```





Mutabilité et OCaml

```
type 'a tree = Nil
             | Node of { mutable value : 'a;
                         mutable lchild : 'a tree;
                         mutable rchild : 'a tree };;
```

Mutabilité et OCaml

```
# let rec incr_leftmost_node = function
    | Nil -> failwith "Arbre vide"
    | Node n when n.lchild=Nil -> n.value <- n.value+1
    | Node n -> incr_leftmost_node n.lchild;;

val incr_leftmost_node : int tree -> unit = <fun>
```

Mutabilité et OCaml


```
# let rec mirror = function
    | Nil -> ()
    | Node n -> let tmp = n.rchild in
                n.rchild <- n.lchild;
                n.lchild <- tmp;
                mirror n.lchild;
                mirror n.rchild;;

val mirror : 'a tree -> unit = <fun>
```

Mutabilité et OCaml

```
# let rec delete_leaves = function
  | Nil -> Nil
  | Node { value=_; lchild=Nil, rchild=Nil } -> Nil
  | Node n -> n.lchild <- delete_leaves n.lchild;
              n.rchild <- delete_leaves n.rchild;
              Node n;

val delete_leaves : 'a tree -> 'a tree = <fun>
```