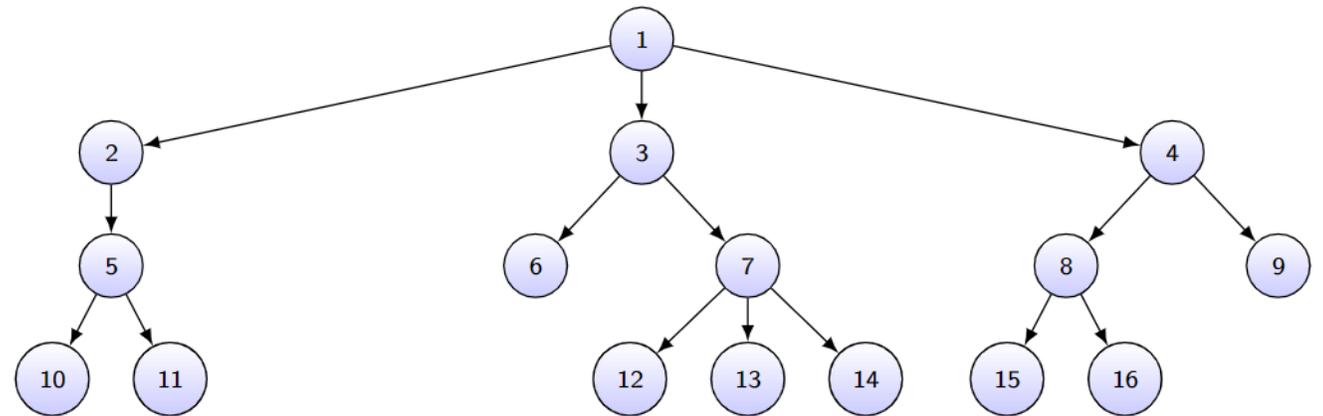


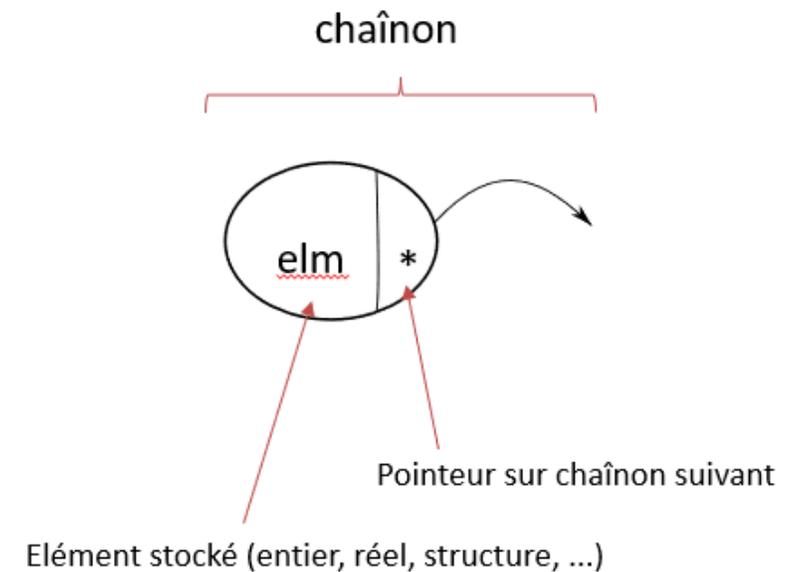
# INFORMATIQUE 3

## III. ARBRES



# Rappel: les listes chaînées

```
Structure Chainon :  
  elm : Element  
  suivant : pointeur sur structure Chainon
```

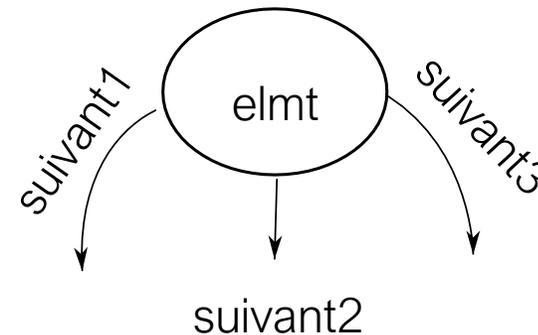


- Les chaînons de la liste pointent sur le chaînon suivant.
- Contrairement aux tableaux statiques, la taille des listes chaînées n'est pas limitée.

# Et si on rajoutais une dimension ?

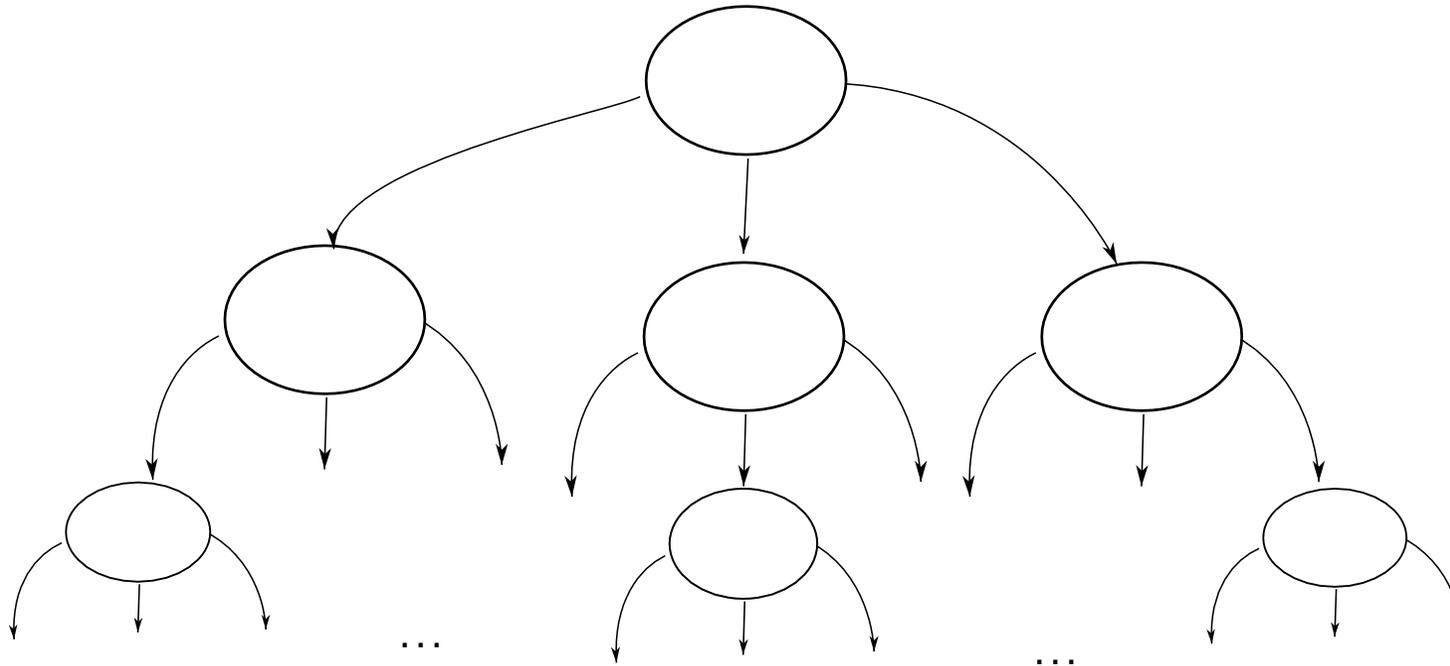
- Le principe des listes chaînées est utilisé pour de nombreux autres concepts!
- Exemple : une Chainon avec plusieurs « suivant »...

```
typedef struct chainon
{
    int elmt;
    struct chainon *suivant1;
    struct chainon *suivant2;
    struct chainon *suivant3;
} Chainon;
```



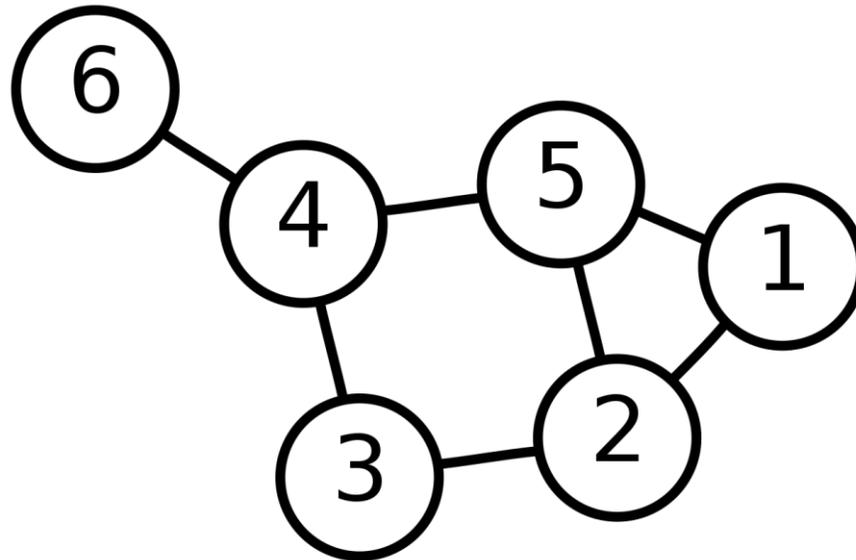
# Et si on rajoutais une dimension ?

- Le principe des listes chaînées est utilisé pour de nombreux autres concepts!
- Exemple : une Chainon avec plusieurs « suivant »... permet de construire un arbre !



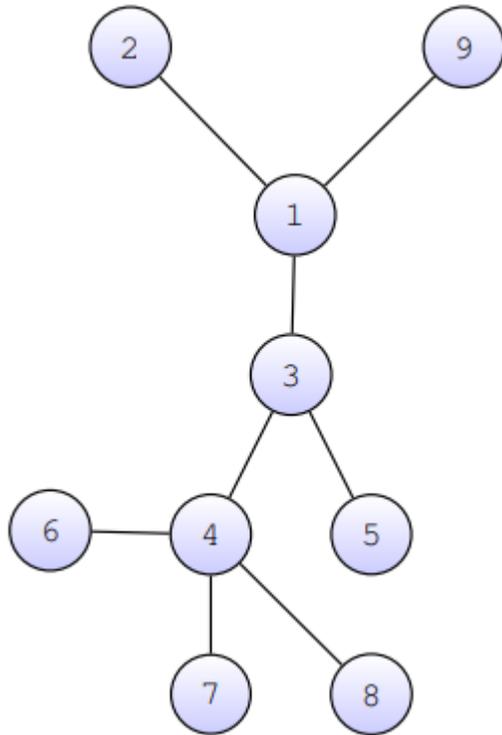
# Les graphes

- Les listes chaînées appartiennent à la famille des graphes.
- Un graphe est composé d'un ensemble d'éléments reliés entre eux.
- Il existe une très grande variété de types de graphes. La discipline qui les étudie est la **théorie des graphes**.

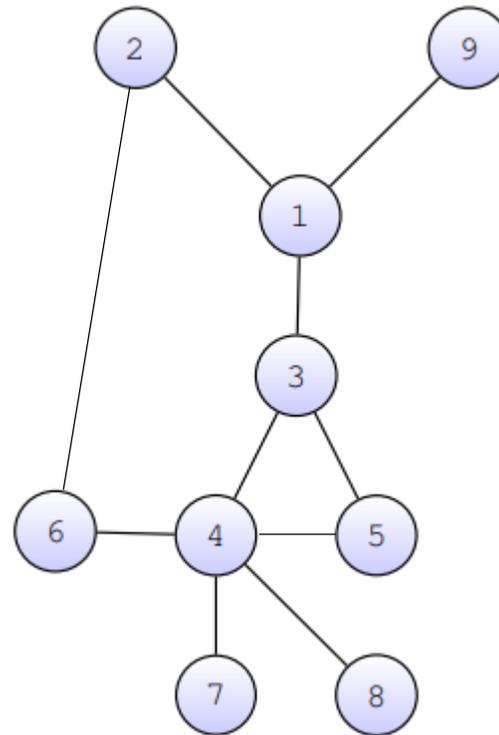


# Arbre : définition

- Un arbre est un **graphe** (orienté ou non), **connexe et sans cycle**.



Arbre



Graphe connexe avec cycle

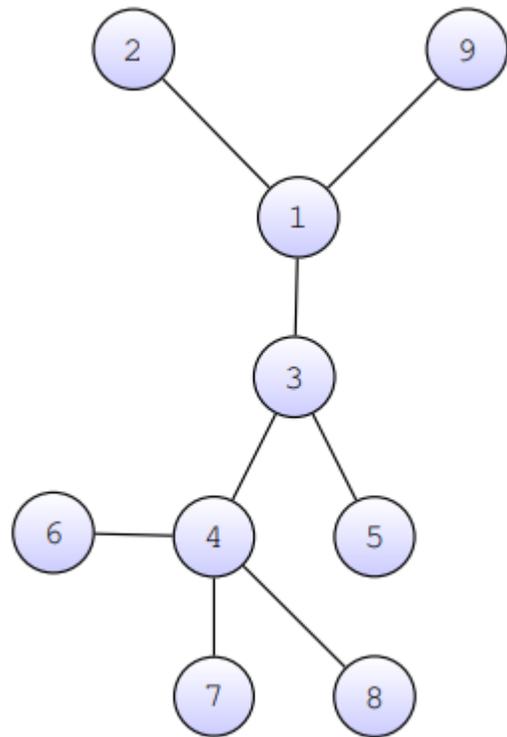
Orienté: il n'y a qu'un seul sens autorisé pour aller d'un élément à l'autre

Connexe : d'un seul tenant.  
On peut toujours trouver un chemin qui lie deux éléments entre eux

Sans cycle : pas de "rebouclage" entre les éléments

# Arbre enraciné

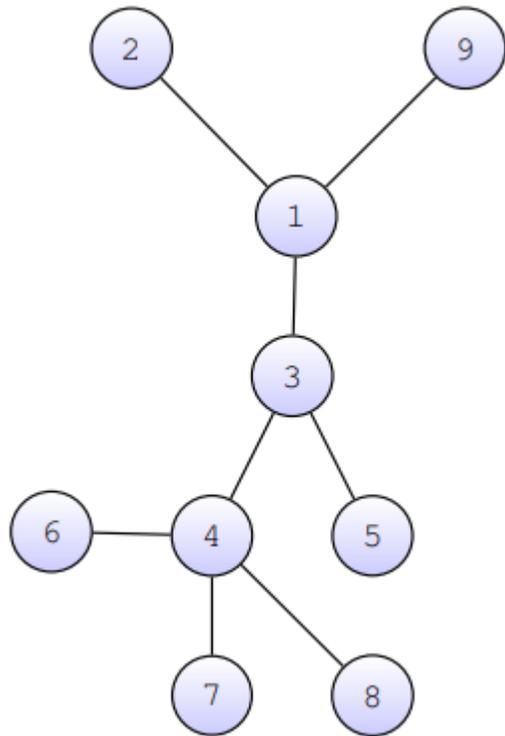
- **Arbre non-enraciné** : il n'y a pas d'ordre entre ses éléments



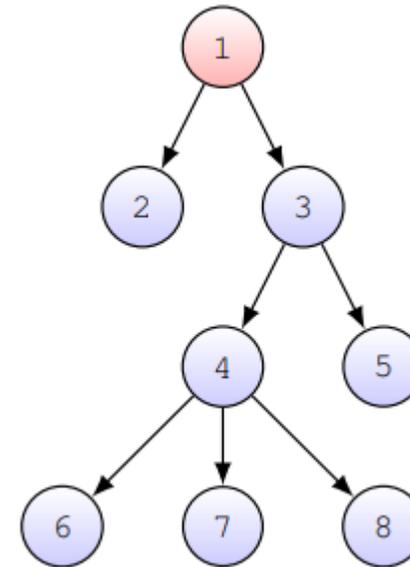
Arbre non enraciné

# Arbre enraciné

- **Arbre non-enraciné** : il n'y a pas d'ordre entre ses éléments
- **Arbre enraciné** : arbre hiérarchique dans lequel on peut établir des niveaux. Il est **orienté**.



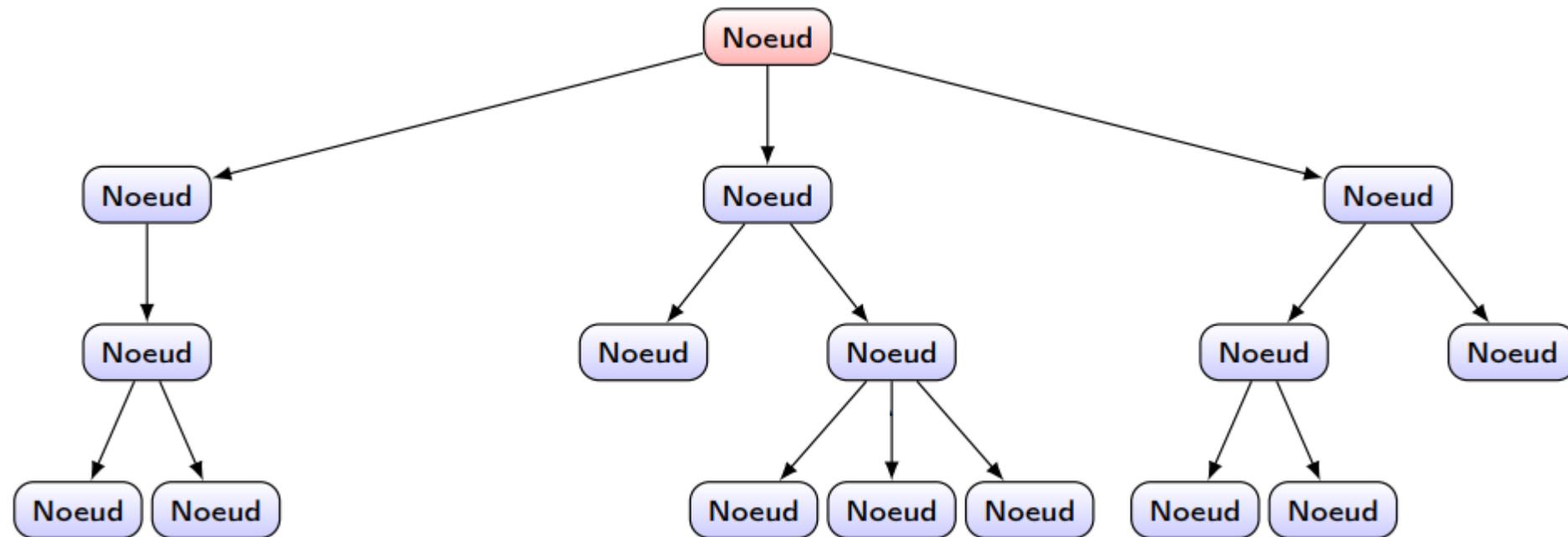
Arbre non enraciné



Arbre enraciné ou orienté

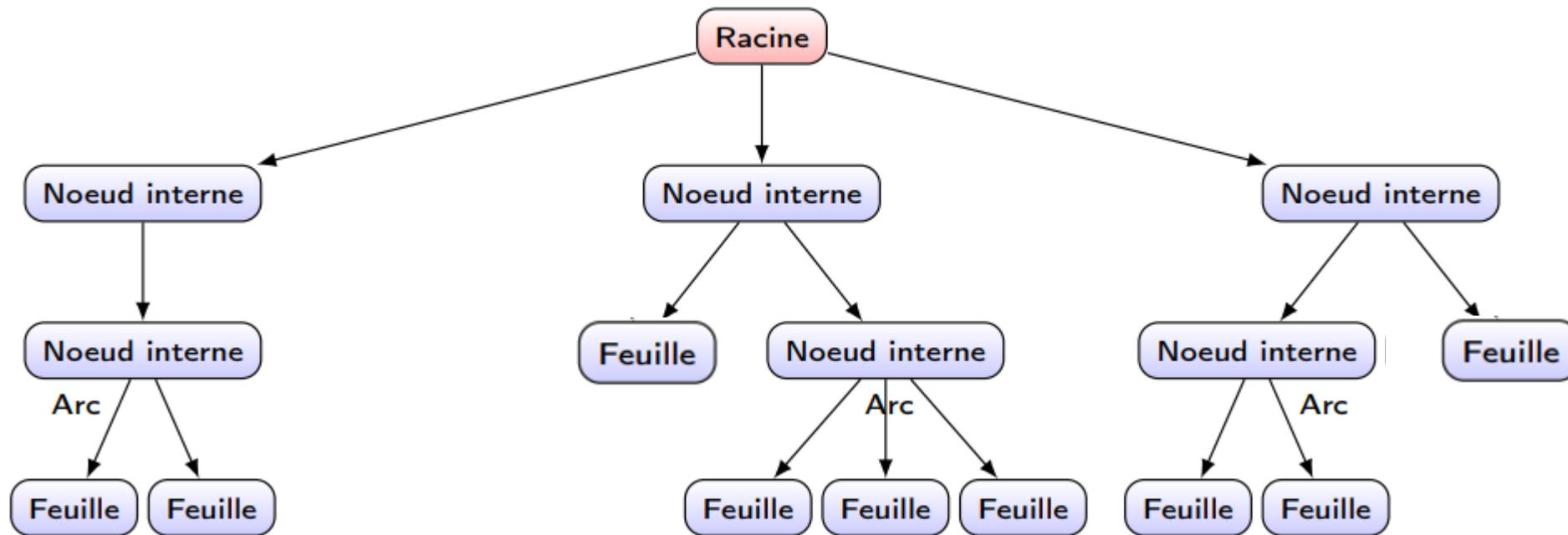
# Arbre : vocabulaire

- Un **nœud** est un élément de l'arbre.



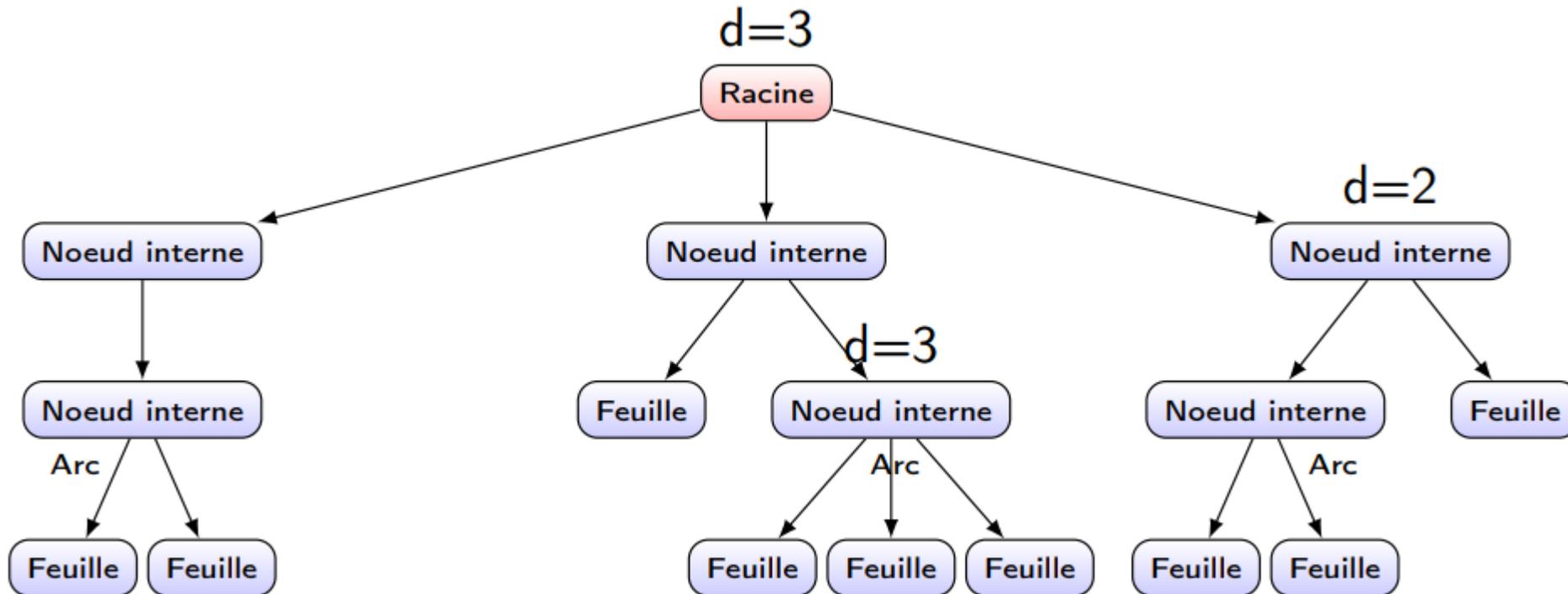
# Arbre : vocabulaire

- **Racine** : nœud au sommet de l'arbre.
- **Nœud externe ou feuille** : c'est un nœud qui n'a pas d'arc sortant, qui est au bout de l'arbre.
- **Nœud interne** : c'est un nœud qui n'est pas externe.



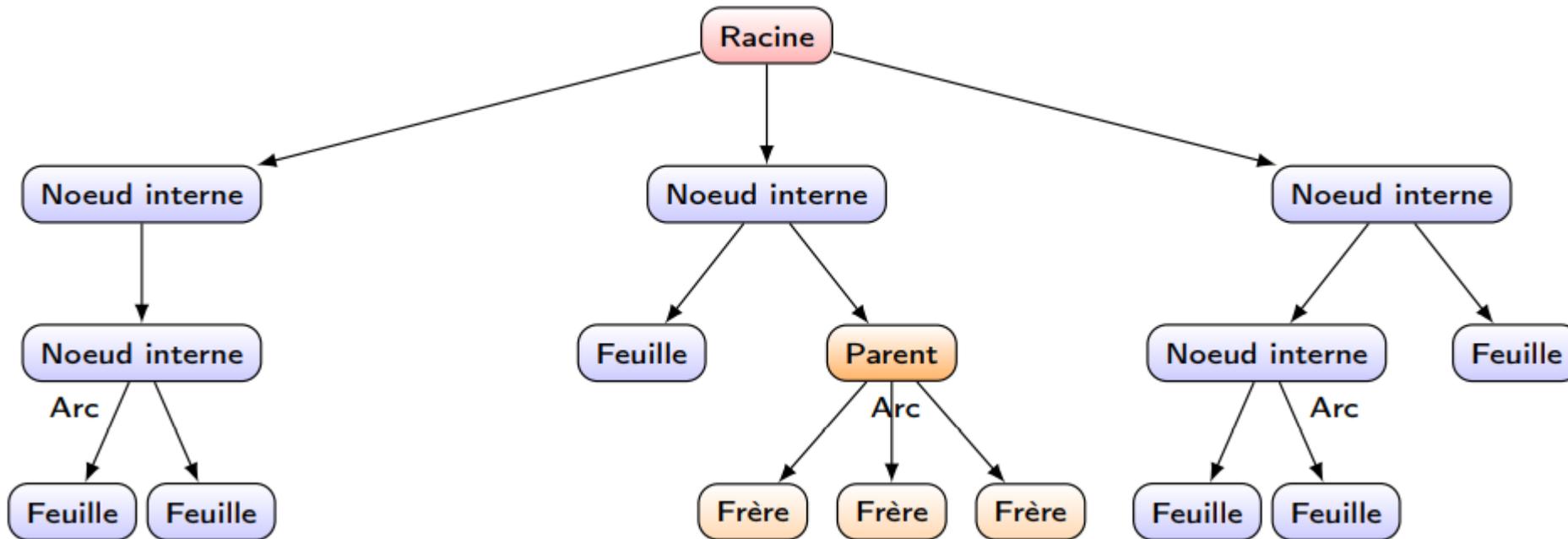
# Arbre : vocabulaire

- **Racine** : nœud au sommet de l'arbre.
- **Nœud externe ou feuille** : c'est un nœud qui n'a pas d'arc sortant, qui est au bout de l'arbre.
- **Nœud interne** : c'est un nœud qui n'est pas externe.
- **Degré** d'un nœud : le nombre d'arcs sortants d'un nœud, nombre de  **fils**.



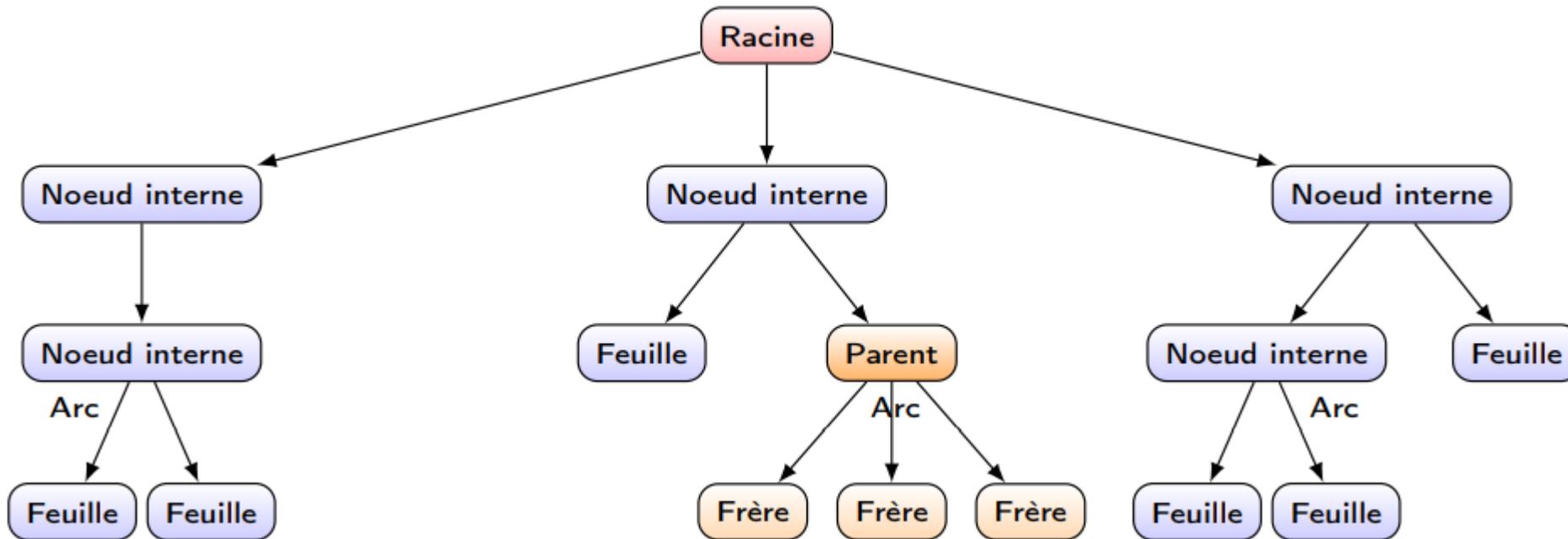
# Arbre : vocabulaire

- **Racine** : nœud au sommet de l'arbre.
- **Nœud externe ou feuille** : c'est un nœud qui n'a pas d'arc sortant, qui est au bout de l'arbre.
- **Nœud interne** : c'est un nœud qui n'est pas externe.
- **Degré** d'un nœud : le nombre d'arcs sortants d'un nœud, nombre de  **fils**.
- Nœuds **frères ou sœurs** : des nœuds qui ont le même parent.



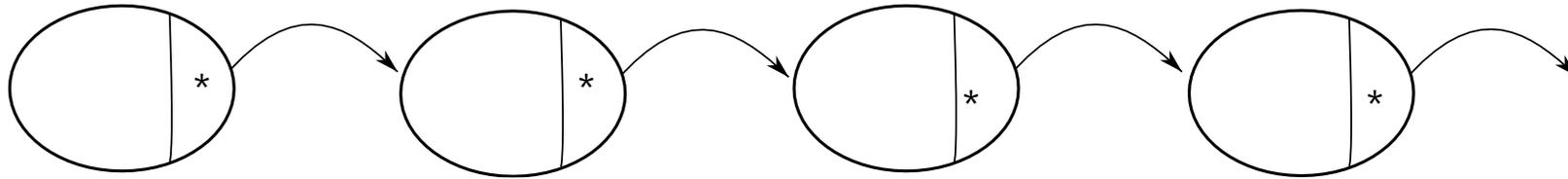
# Arbre : vocabulaire

- **Racine** : nœud au sommet de l'arbre.
- **Nœud externe ou feuille** : c'est un nœud qui n'a pas d'arc sortant, qui est au bout de l'arbre.
- **Nœud interne** : c'est un nœud qui n'est pas externe.
- **Degré** d'un nœud : le nombre d'arcs sortants d'un nœud, nombre de  **fils**.
- Nœuds **frères ou sœurs** : des nœuds qui ont le même parent.
- Un **sous-arbre** est l'arbre issu d'un nœud.



# Arbre : vocabulaire

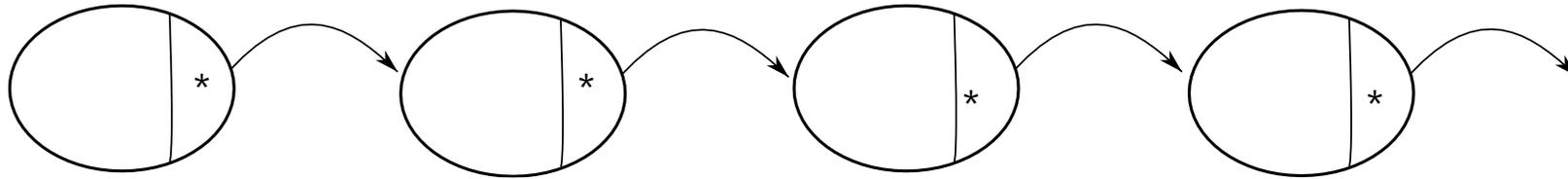
- Une liste chaînée est un arbre orienté (enraciné) où chaque noeud a un seul fils!



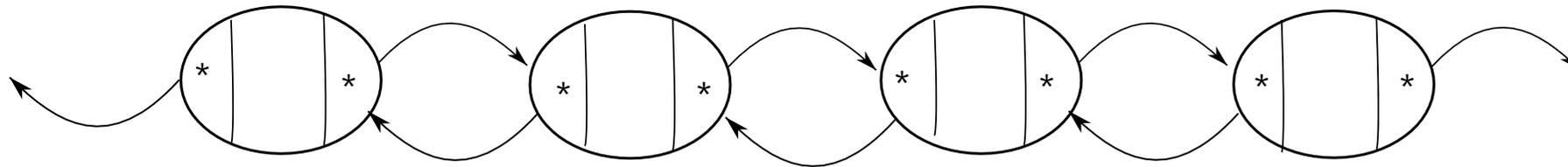
- Arbre non orienté où chaque noeud à un fils ??

# Arbre : vocabulaire

- Une liste chaînée est un arbre orienté (enraciné) où chaque nœud a un seul fils!



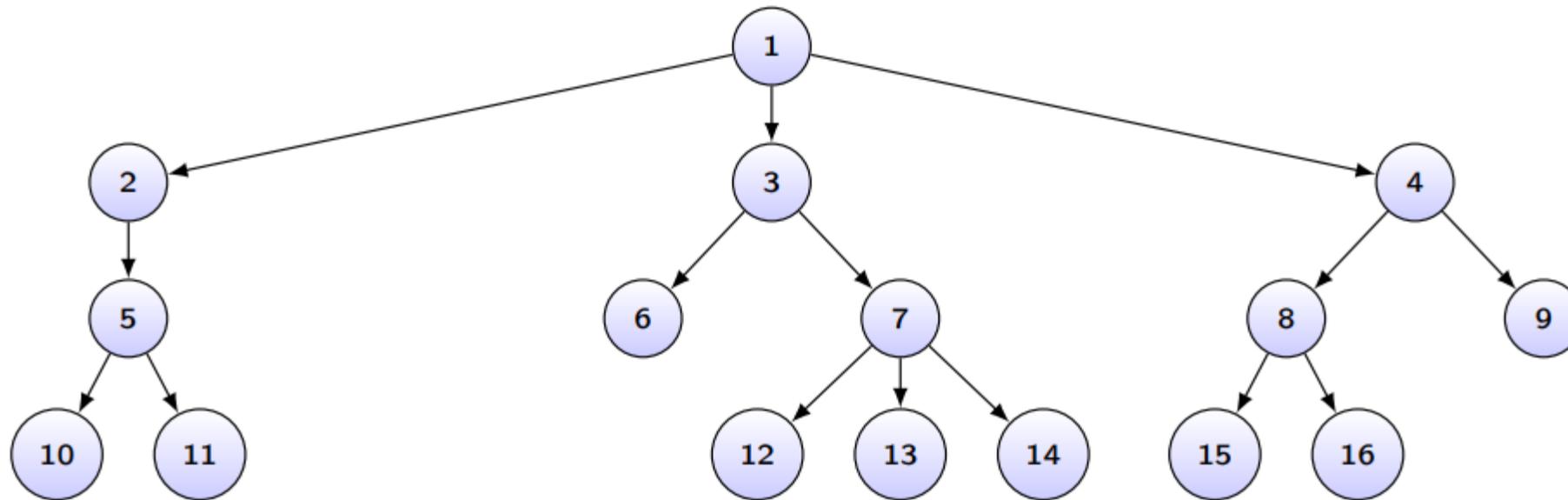
- Arbre non orienté où chaque nœud a un fils ??



Liste doublement chaînée !

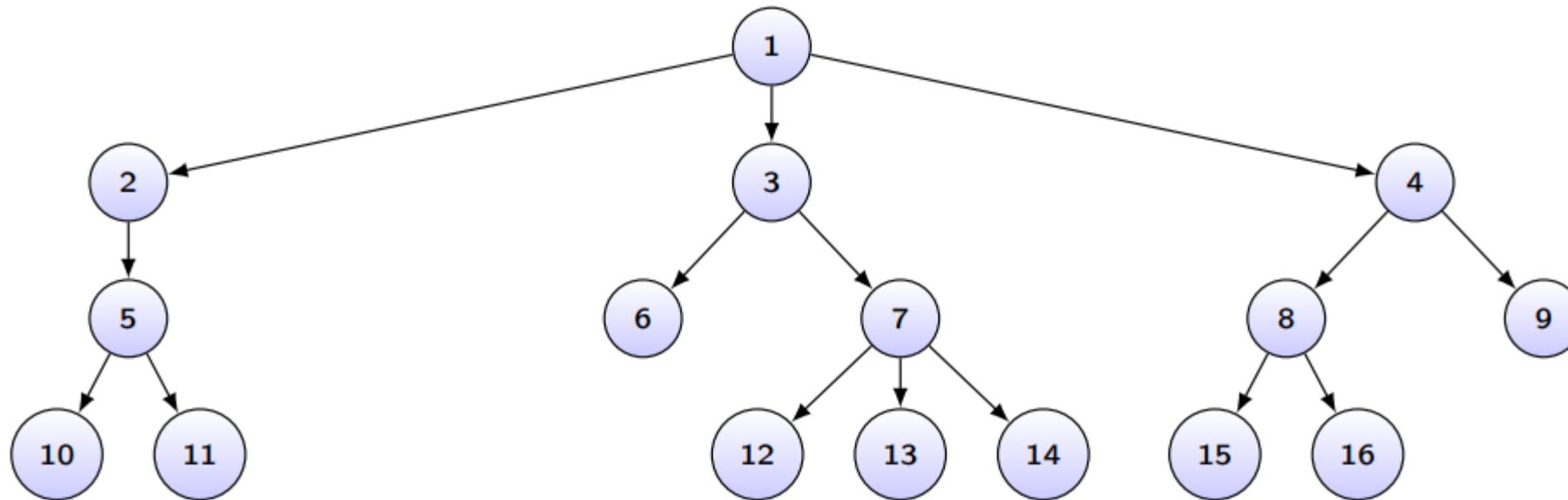
# Arbre : vocabulaire

- **Arbre numéroté** : chaque nœud est étiqueté par un entier positif
- **Arbre ordonné** : il existe un ordre entre les enfants et les nœuds internes. On dit que le  $i$  ème enfant est absent si aucun nœud n'est étiqueté par la valeur  $i$ .



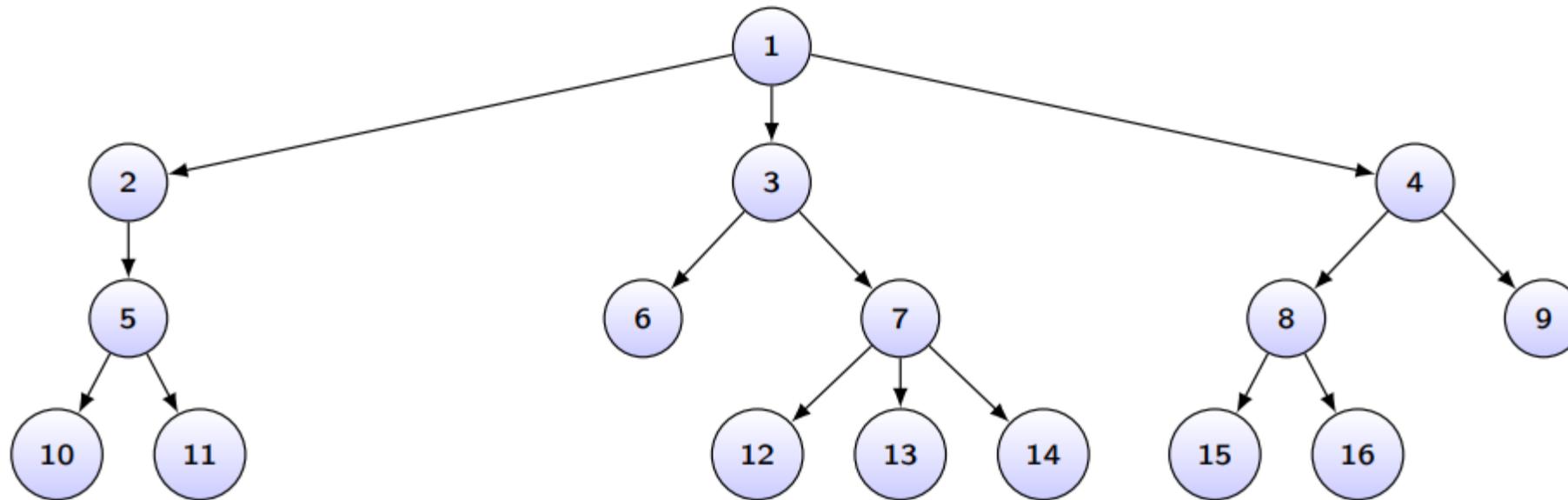
# Arbre : vocabulaire

- **Arbre numéroté** : chaque nœud est étiqueté par un entier positif
- **Arbre ordonné** : il existe un ordre entre les enfants et les nœuds internes. On dit que le  $i$  ème enfant est absent si aucun nœud n'est étiqueté par la valeur  $i$ .
- Un **arbre k-iaire** est un arbre dont les nœud ont au maximum  $k$  fils



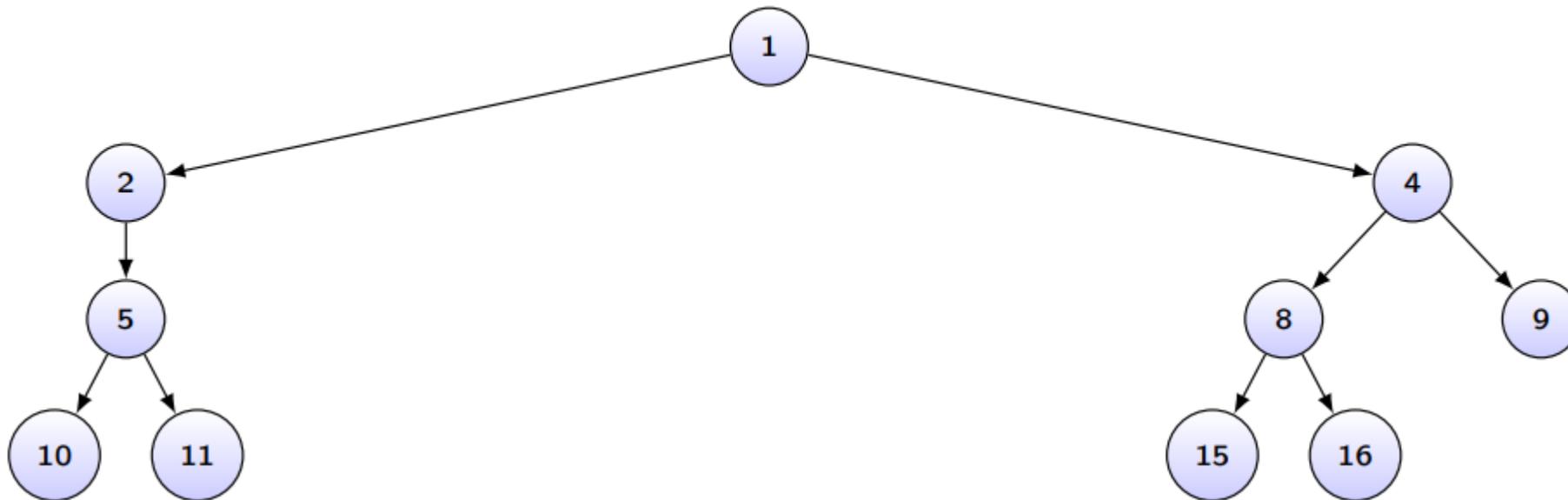
# Arbre : vocabulaire

- **Arbre numéroté** : chaque nœud est étiqueté par un entier positif
- **Arbre ordonné** : il existe un ordre entre les enfants et les nœuds internes. On dit que le  $i$  ème enfant est absent si aucun nœud n'est étiqueté par la valeur  $i$ .
- Un **arbre k-iaire** est un arbre dont les nœud ont au maximum  $k$  fils. Ici on a  $k=3$ .



# Arbre : vocabulaire

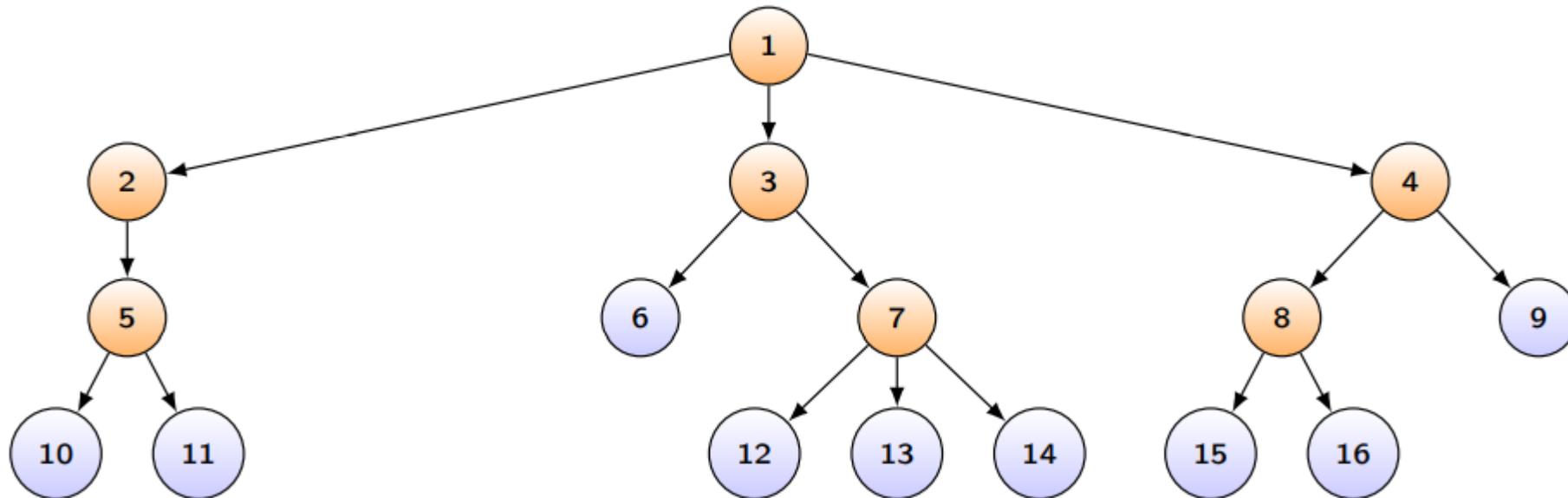
- **Arbre numéroté** : chaque nœud est étiqueté par un entier positif
- **Arbre ordonné** : il existe un ordre entre les enfants et les nœuds internes. On dit que le  $i$ ème enfant est absent si aucun nœud n'est étiqueté par la valeur  $i$ .
- Un **arbre binaire** est un arbre dont les nœuds ont au maximum 2 fils.



Exemple d'arbre binaire

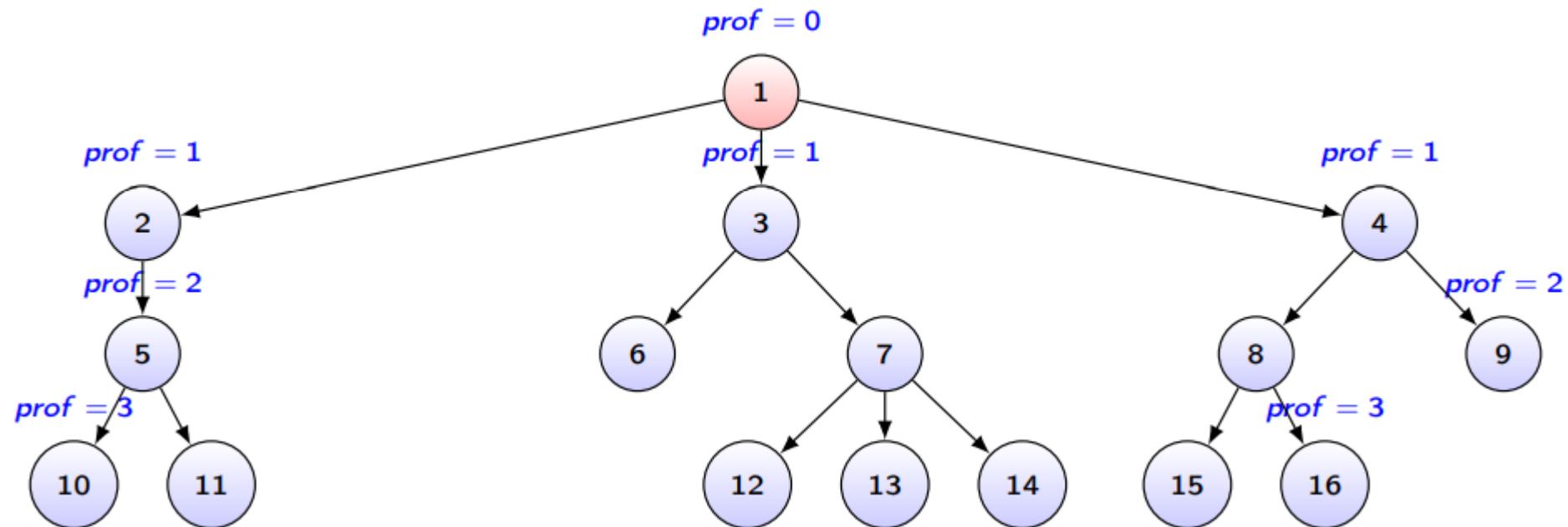
# Arbre : vocabulaire

- Taille d'un arbre: nombre de de noeuds internes de l'arbre



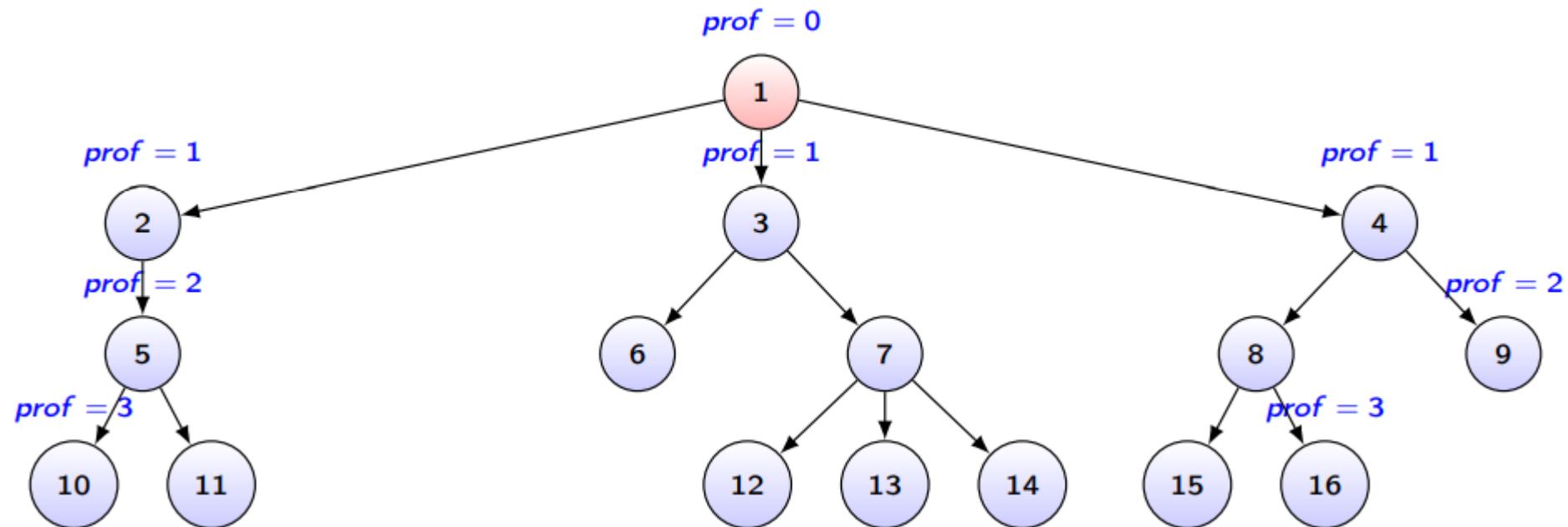
# Arbre : vocabulaire

- Taille d'un arbre: nombre de de nœuds internes de l'arbre
- Profondeur d'un nœud : c'est la distance du chemin qui relie la racine au nœud.



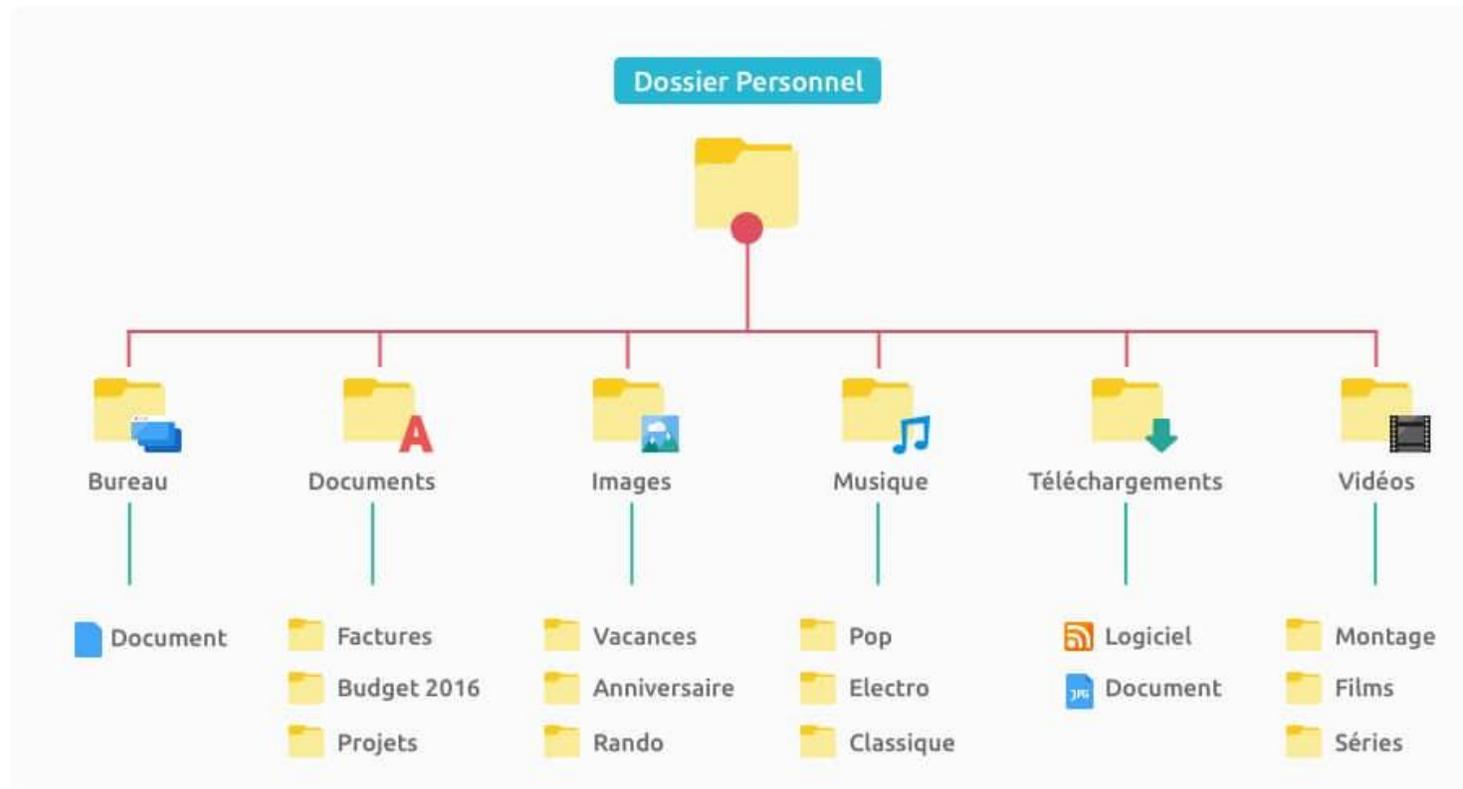
# Arbre : vocabulaire

- Taille d'un arbre: nombre de de nœuds internes de l'arbre
- Profondeur d'un nœud : c'est la distance du chemin qui relie la racine au nœud.
- Hauteur d'un arbre : profondeur maximale. Ici 3.



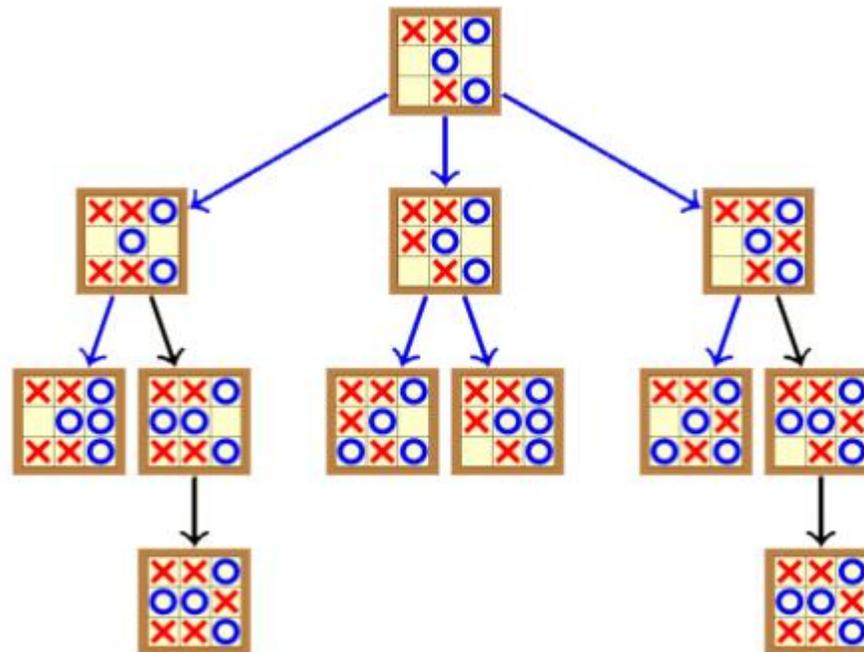
# Arbre : Utilité

- Les arbres sont souvent utilisés en informatique (compression, expressions arithmétiques, arbres de recherche, structures de données hiérarchiques etc... )
- Exemple : organisation en arbre des fichiers



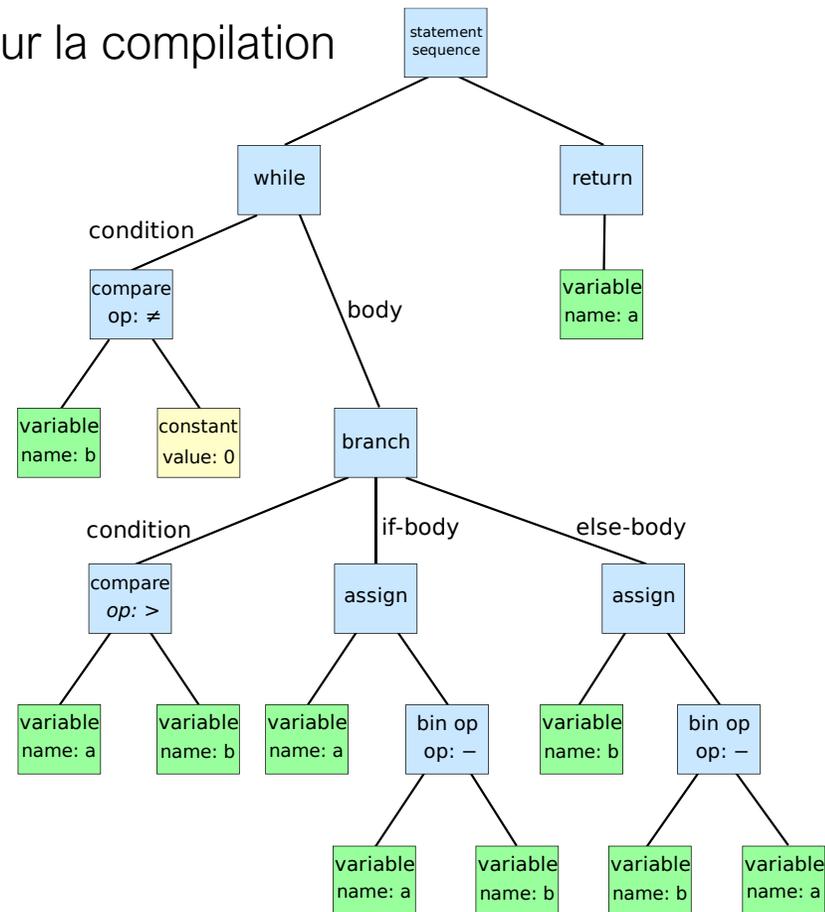
# Arbre : Utilité

- Les arbres sont souvent utilisés en informatique (compression, expressions arithmétiques, arbres de recherche, structures de données hiérarchiques etc... )
- Exemple : Théorie des jeux, intelligence artificielle classique



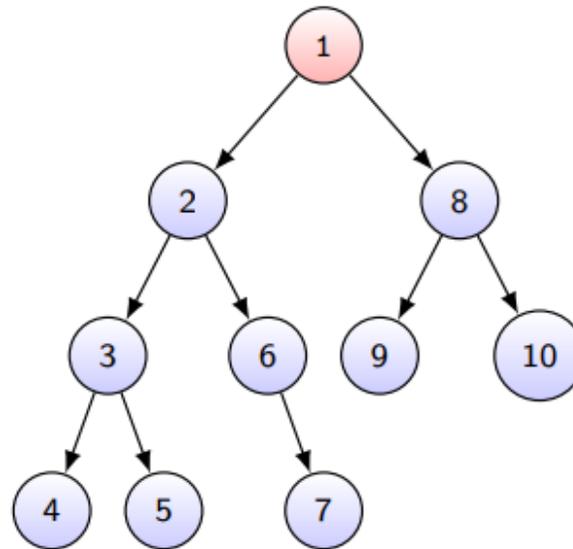
# Arbre : Utilité

- Les arbres sont souvent utilisés en informatique (compression, expressions arithmétiques, arbres de recherche, structures de données hiérarchiques etc... )
- Exemple : arbre syntaxique pour la compilation



# Implémentation d'un arbre

- Le cas des **arbres binaires** est le plus facile à étudier et à implémenter car l'on connaît par avance le nombre maximal de fils de chaque nœud.
- Nous nous concentrons sur ce cas particulier d'arbres pour la suite du cours.
- L'extension à des arbres k-aires se fait en augmentant le nombre de fils.



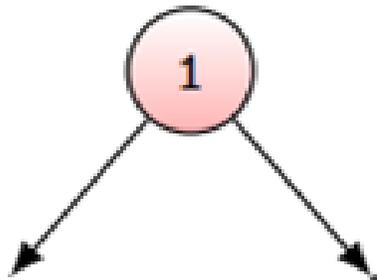
# Implémentation d'un arbre binaire

- Les nœuds d'un **arbre binaire** ont maximum deux fils.
- La structure d'un nœud de l'arbre peut donc être :

Structure Arbre :

```
  elm : Element
```

```
  fg, fd : pointeurs sur structure Arbre; // fils gauche et droit
```



# Implémentation d'un arbre binaire

- Création d'un nœud de l'arbre :

```
FONCTION creerArbre(r: Element) : pointeur sur Arbre
```

```
VARIABLE
```

```
    noeud : pointeur sur Arbre
```

```
DEBUT
```

```
    noeud ← reserverMemoire(Arbre)
```

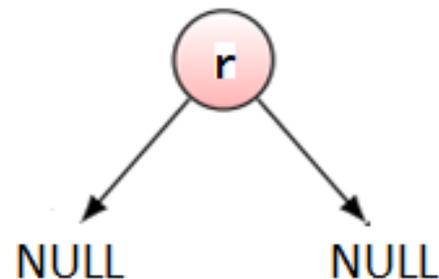
```
    elmt(noeud) ← r
```

```
    fg(noeud) ← NULL // Le noeud n'a pas de fils à sa création
```

```
    fd(noeud) ← NULL
```

```
    RETOURNER noeud
```

```
FIN
```



# Implémentation d'un arbre binaire

- Création d'un nœud de l'arbre :

```
FONCTION creerArbre(r: Element) : pointeur sur Arbre
```

```
VARIABLE
```

```
    noeud : pointeur sur Arbre
```

```
DEBUT
```

```
    noeud ← reserverMemoire(Arbre)
```

```
    elmt(noeud) ← r
```

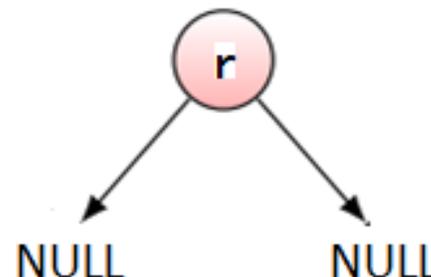
```
    fg(noeud) ← NULL // Le noeud n'a pas de fils à sa création
```

```
    fd(noeud) ← NULL
```

```
    RETOURNER noeud
```

```
FIN
```

Remarque : comme pour les listes chaînées, on va utiliser des pointeurs sur Arbre plutôt que des structures Arbre afin de pouvoir libérer l'espace mémoire à posteriori.



# Opérations sur un arbre binaire

- Vérifier sur un nœud de l'arbre est une feuille (nœud « au bout » de l'arbre)

FONCTION estFeuille(a : pointeur sur Arbre) : Boolean

DEBUT

SI a EGAL A NULL ALORS  
ERREUR()

FIN SI

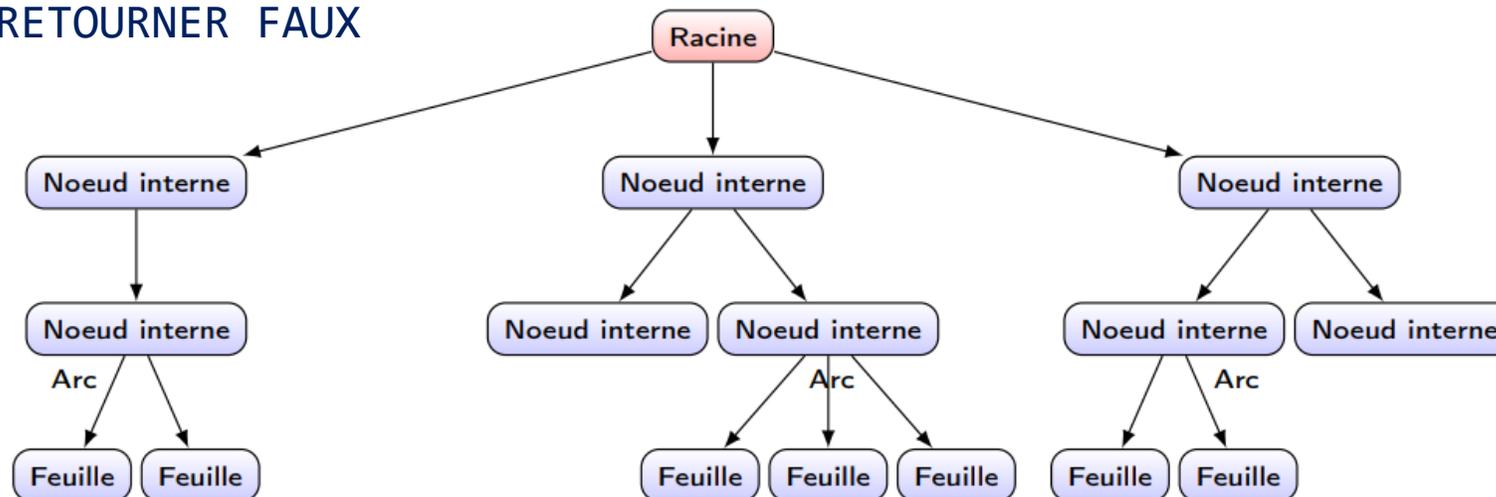
SI ??? ALORS  
RETOURNER VRAI

SINON

RETOURNER FAUX

FIN SI

FIN



# Opérations sur un arbre binaire

- Vérifier sur un nœud de l'arbre est une feuille (nœud « au bout » de l'arbre)

FONCTION estFeuille(a : pointeur sur Arbre) : Booleen

DEBUT

SI a EGAL A NULL ALORS  
ERREUR()

FIN SI

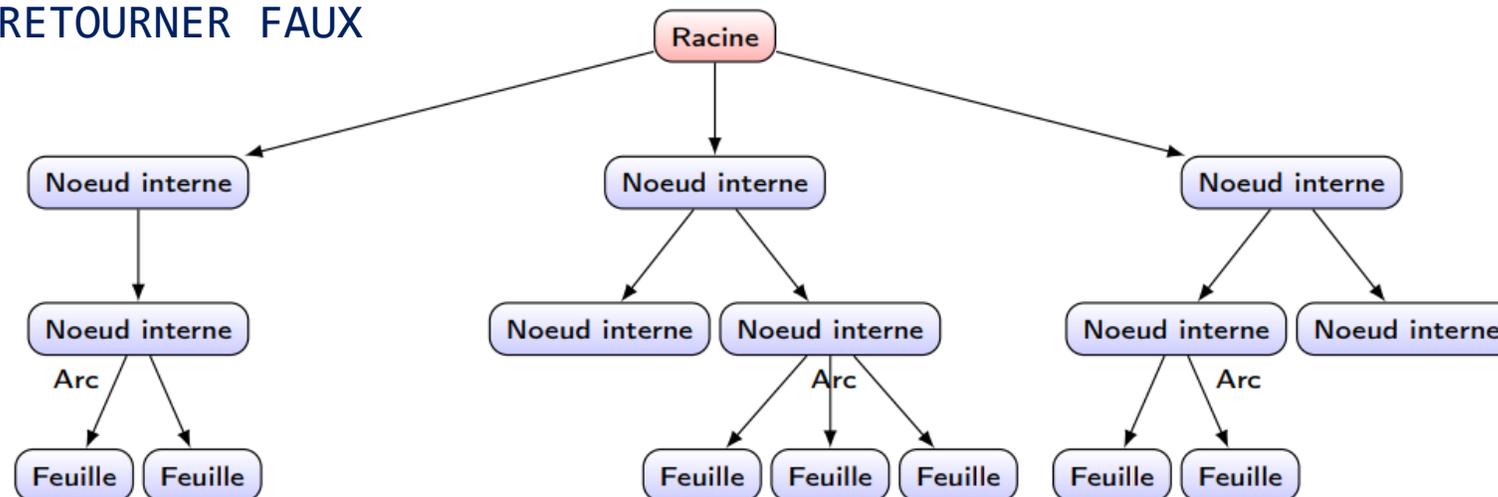
SI fd(a) EGAL A NULL ET fg(a) EGAL A NULL ALORS  
RETOURNER VRAI

SINON

RETOURNER FAUX

FIN SI

FIN



# Opérations sur un arbre binaire

- Vérifier si le fils droit ou gauche d'un nœud existe :

```
FONCTION existeFilsDroit(a : pointeur sur
Arbre) : Booleen
DEBUT
    SI a EGAL A NULL ALORS
        ERREUR()
    SINON SI ??? ALORS
        RETOURNER FAUX
    SINON
        RETOURNER VRAI
    FIN SI
FIN
```

```
FONCTION existeFilsGauche(a : pointeur sur
Arbre) : Booleen
DEBUT
    SI a EGAL A NULL ALORS
        ERREUR()
    SINON SI ??? ALORS
        RETOURNER FAUX
    SINON
        RETOURNER VRAI
    FIN SI
FIN
```

# Opérations sur un arbre binaire

- Vérifier si le fils droit ou gauche d'un nœud existe :

```
FONCTION existeFilsDroit(a : pointeur sur
Arbre) : Booleen
DEBUT
    SI a EGAL A NULL ALORS
        ERREUR()
    SINON SI fd(a) EGAL A NULL ALORS
        RETOURNER FAUX
    SINON
        RETOURNER VRAI
    FIN SI
FIN
```

```
FONCTION existeFilsGauche(a : pointeur sur
Arbre) : Booleen
DEBUT
    SI a EGAL A NULL ALORS
        ERREUR()
    SINON SI fg(a) EGAL A NULL ALORS
        RETOURNER FAUX
    SINON
        RETOURNER VRAI
    FIN SI
FIN
```

# Opérations sur un arbre binaire

- Retourner l'adresse du fils droit ou gauche d'un arbre:

```
FONCTION filsDroit(a: pointeur sur Arbre) :  
pointeur sur Arbre  
DEBUT  
    SI a EGAL A NULL ALORS  
        ERREUR()  
    SINON SI existeFilsDroit(a) ALORS  
        RETOURNER fd(a)  
    SINON  
        RETOURNER NULL  
    FIN SI  
FIN
```

```
FONCTION filsGauche(a: pointeur sur Arbre) :  
pointeur sur Arbre  
DEBUT  
    SI a EGAL A NULL ALORS  
        ERREUR()  
    SINON SI existeFilsGauche(a) ALORS  
        RETOURNER fg(a)  
    SINON  
        RETOURNER NULL  
    FIN SI  
FIN
```

# Opérations sur un arbre binaire

- Modifier l'élément d'un nœud

```
PROCEDURE modifierElement(a: pointeur sur Arbre; r: Element) :  
pointeur sur Arbre
```

```
DEBUT
```

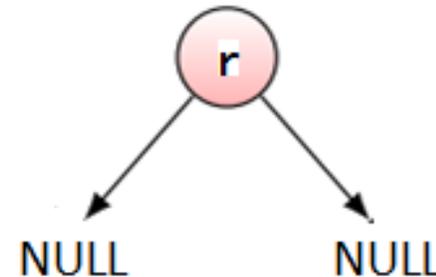
```
    SI a EGAL A NULL ALORS  
        ERREUR()
```

```
    SINON
```

```
        elmt(a) ← r
```

```
    FIN SI
```

```
FIN
```



# Opérations sur un arbre binaire

- Ajouter un fils droit ou gauche au nœud :

```
FONCTION ajouterFilsGauche(a: pointeur sur  
Arbre; r: Element) : Booléen
```

```
DEBUT
```

```
  SI a EGAL A NULL ALORS
```

```
    ???
```

```
  SINON SI ??? ALORS
```

```
    fg(a) ← creerArbre(r)
```

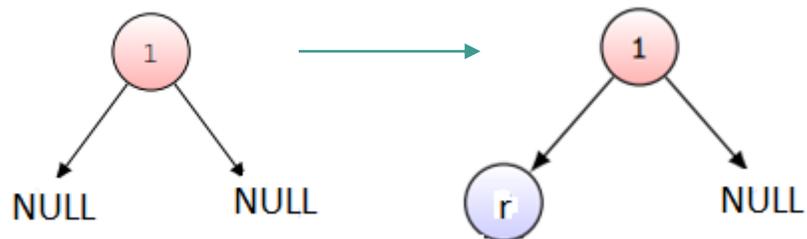
```
    RETOURNER VRAI
```

```
  SINON
```

```
    RETOURNER FAUX
```

```
  FIN SI
```

```
FIN
```



```
FONCTION ajouterFilsDroit(a: pointeur sur  
Arbre; r: Element) : Booléen
```

```
DEBUT
```

```
  SI a EGAL A NULL ALORS
```

```
    ???
```

```
  SINON SI ??? ALORS
```

```
    fd(a) ← creerArbre(r)
```

```
    RETOURNER VRAI
```

```
  SINON
```

```
    RETOURNER FAUX
```

```
  FIN SI
```

```
FIN
```



# Opérations sur un arbre binaire

- Ajouter un fils droit ou gauche au nœud :

FONCTION ajouterFilsGauche(a: pointeur sur  
Arbre; r: Element) : Booléen

DEBUT

SI a EGAL A NULL

    a ← creerArbre(r)

    RETOURNER VRAI

SINON SI ??? ALORS

    fg(a) ← creerArbre(r)

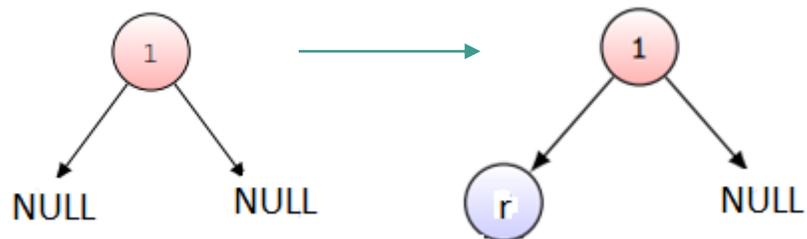
    RETOURNER VRAI

SINON

    RETOURNER FAUX

FIN SI

FIN



FONCTION ajouterFilsDroit(a: pointeur sur  
Arbre; r: Element) : Booléen

DEBUT

SI a EGAL A NULL ALORS

    a ← creerArbre(r)

    RETOURNER VRAI

SINON SI ??? ALORS

    fd(a) ← creerArbre(r)

    RETOURNER VRAI

SINON

    RETOURNER FAUX

FIN SI

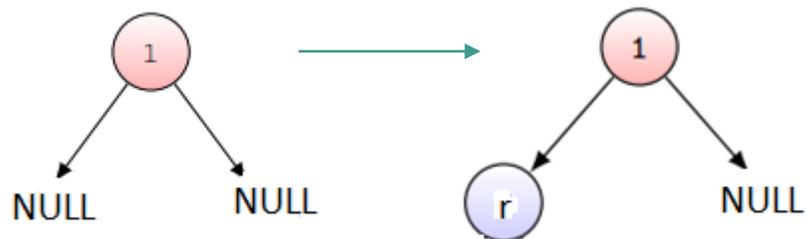
FIN



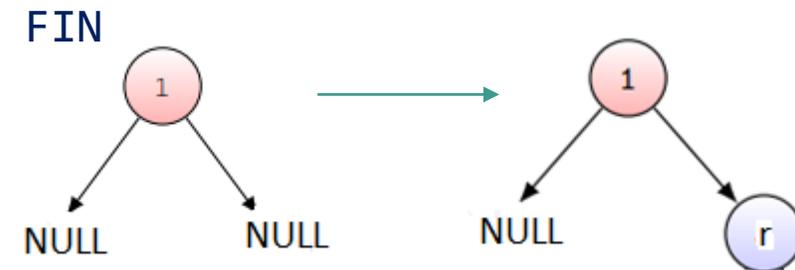
# Opérations sur un arbre binaire

- Ajouter un fils droit ou gauche au nœud :

```
FONCTION ajouterFilsGauche(a: pointeur sur
Arbre; r: Element) : Booléen
DEBUT
  SI a EGAL A NULL
    a ← creerArbre(r)
    RETOURNER VRAI
  SINON SI NON existeFilsGauche(a) ALORS
    fg(a) ← creerArbre(r)
    RETOURNER VRAI
  SINON
    RETOURNER FAUX
  FIN SI
FIN
```



```
FONCTION ajouterFilsDroit(a: pointeur sur
Arbre; r: Element) : Booléen
DEBUT
  SI a EGAL A NULL ALORS
    a ← creerArbre(r)
    RETOURNER VRAI
  SINON SI NON existeFilsDroit(a) ALORS
    fd(a) ← creerArbre(r)
    RETOURNER VRAI
  SINON
    RETOURNER FAUX
  FIN SI
FIN
```



# Opérations sur un arbre binaire

- Supprimer fils gauche/ fils droit d'un nœud:

```
PROCEDURE supprimerFilsDroit(a: pointeur sur Arbre)
```

```
DEBUT
```

```
  // pointeur NULL ?
```

```
  SI a EGAL A NULL ALORS
```

```
    ERREUR()
```

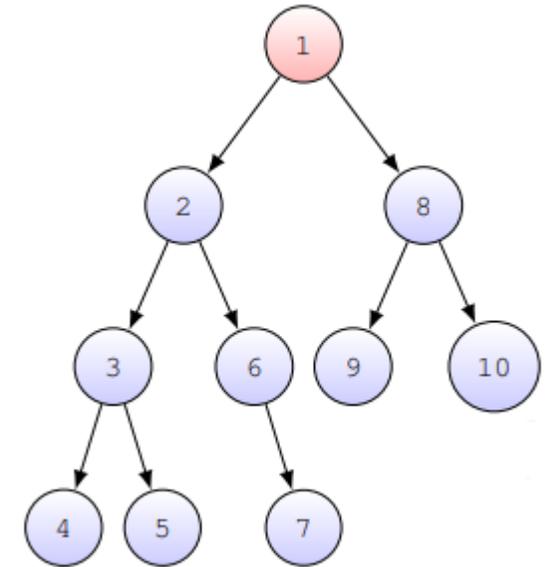
```
  // Existence du fils droit ?
```

```
  SINON SI existeFilsDroit(a) ALORS
```

```
    libererMemoire(fd(a))
```

```
  FIN SI
```

```
FIN
```



# Opérations sur un arbre binaire

- Supprimer fils gauche/ fils droit d'un nœud:

```
PROCEDURE supprimerFilsDroit(a: pointeur sur Arbre)
```

```
DEBUT
```

```
  // pointeur NULL ?
```

```
  SI a EGAL A NULL ALORS
```

```
    ERREUR()
```

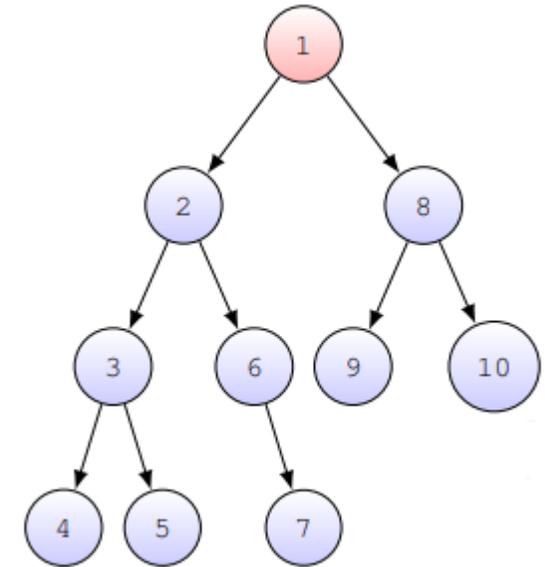
```
  // Existence du fils droit ?
```

```
  SINON SI existeFilsDroit(a) ALORS
```

```
    libererMemoire(fd(a))
```

```
  FIN SI
```

```
FIN
```



la libération de mémoire correspond à celle allouée, donc uniquement au nœud, pas ses descendants !

→ Fuite mémoire RAM

→ Il faut également supprimer tous les descendants

# Opérations sur un arbre binaire

- Supprimer fils gauche/ fils droit d'un nœud:

```
PROCEDURE supprimerFilsDroit(a: pointeur sur Arbre)
DEBUT
    // pointeur NULL ?
    SI a EGAL A NULL ALORS
        ERREUR()
    // Existence du fils droit ?
    SINON SI existeFilsDroit(a) ALORS
        // noeud à supprimer possède un fils droit ?
        SI existeFilsGauche(fd(a)) ALORS
            ???
        FIN SI
        // noeud à supprimer possède un fils gauche ?
        SI existeFilsDroit(fd(a)) ALORS
            ???
        FIN SI
        libererMemoire(fd(a))
    FIN SI
FIN
```

# Opérations sur un arbre binaire

- Supprimer fils gauche/ fils droit d'un nœud:

```
PROCEDURE supprimerFilsDroit(a: pointeur sur Arbre)
```

```
DEBUT
```

```
  // pointeur NULL ?
```

```
  SI a EGAL A NULL ALORS
```

```
    ERREUR()
```

```
  // Existence du fils droit ?
```

```
  SINON SI existeFilsDroit(a) ALORS
```

```
    // noeud à supprimer possède un fils droit ?
```

```
    SI existeFilsGauche(fd(a)) ALORS
```

```
      supprimerFilsGauche(fd(a)) // appel récursif procédure «miroir»
```

```
    FIN SI
```

```
    // noeud à supprimer possède un fils gauche ?
```

```
    SI existeFilsDroit(fd(a)) ALORS
```

```
      supprimerFilsDroit(fd(a)) // appel récursif
```

```
    FIN SI
```

```
    libererMemoire(fd(a))
```

```
  FIN SI
```

```
FIN
```

# Opérations sur un arbre binaire

- Supprimer fils gauche/ fils droit d'un nœud:

```
PROCEDURE supprimerFilsDroit(a: pointeur sur Arbre)
DEBUT
  SI a EGAL A NULL ALORS
    ERREUR()
  SINON SI existeFilsDroit(a) ALORS
    SI existeFilsGauche(fd(a)) ALORS
      supprimerFilsGauche(fd(a))
    FIN SI
    SI existeFilsDroit(fd(a)) ALORS
      supprimerFilsDroit(fd(a))
    FIN SI
    libererMemoire(fd(a))
  FIN SI
FIN
```

```
PROCEDURE supprimerFilsGauche(a: pointeur sur Arbre)
DEBUT
  SI a EGAL A NULL ALORS
    ERREUR()
  SINON SI existeFilsGauche(a) ALORS
    SI existeFilsGauche(fg(a)) ALORS
      supprimerFilsGauche(fg(a))
    FIN SI
    SI existeFilsDroit(fg(a)) ALORS
      supprimerFilsDroit(fg(a))
    FIN SI
    libererMemoire(fg(a))
  FIN SI
FIN
```

# Opérations sur un arbre binaire

- Supprimer fils gauche/ fils droit d'un nœud: illustration

```
PROCEDURE supprimerFilsGauche(a: pointeur sur Arbre)
```

```
→ DEBUT
```

```
    SI a EGAL A NULL ALORS
```

```
        ERREUR()
```

```
    SINON SI existeFilsGauche(a) ALORS
```

```
        SI existeFilsGauche(fg(a)) ALORS
```

```
            supprimerFilsGauche(fg(a))
```

```
        FIN SI
```

```
        SI existeFilsDroit(fg(a)) ALORS
```

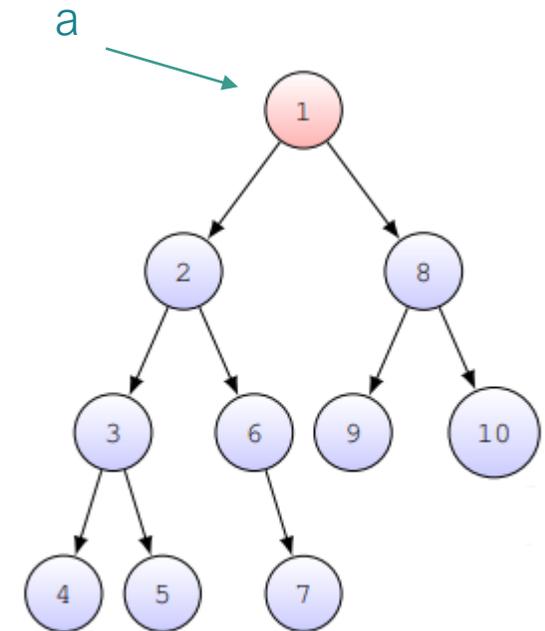
```
            supprimerFilsDroit(fg(a))
```

```
        FIN SI
```

```
        libererMemoire(fg(a))
```

```
    FIN SI
```

```
FIN
```



# Opérations sur un arbre binaire

- Supprimer fils gauche/ fils droit d'un nœud: illustration

```
PROCEDURE supprimerFilsGauche(a: pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a EGAL A NULL ALORS
```

```
    ERREUR()
```

```
  → SINON SI existeFilsGauche(a) ALORS  
    SI existeFilsGauche(fg(a)) ALORS  
      supprimerFilsGauche(fg(a))
```

```
    FIN SI
```

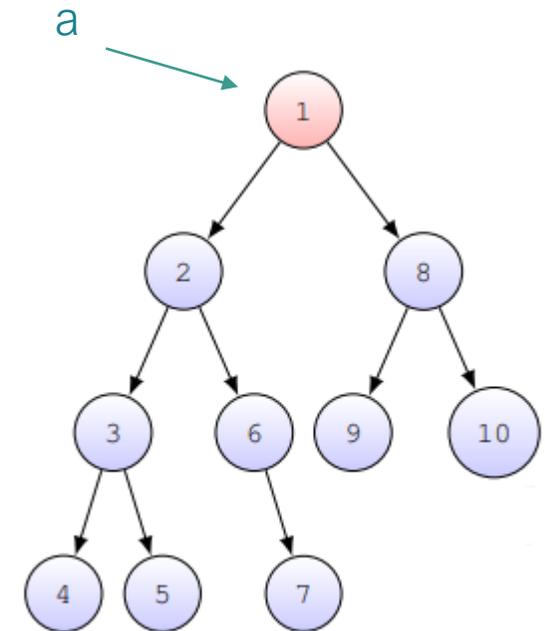
```
    SI existeFilsDroit(fg(a)) ALORS  
      supprimerFilsDroit(fg(a))
```

```
    FIN SI
```

```
    libererMemoire(fg(a))
```

```
  FIN SI
```

```
FIN
```



# Opérations sur un arbre binaire

- Supprimer fils gauche/ fils droit d'un nœud: illustration

```
PROCEDURE supprimerFilsGauche(a: pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a EGAL A NULL ALORS
```

```
    ERREUR()
```

```
  SINON SI existeFilsGauche(a) ALORS
```

```
    → SI existeFilsGauche(fg(a)) ALORS  
      supprimerFilsGauche(fg(a))
```

```
    FIN SI
```

```
  SI existeFilsDroit(fg(a)) ALORS
```

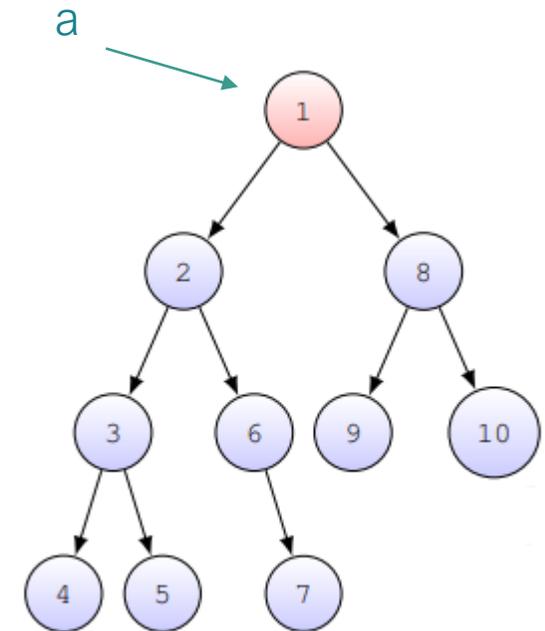
```
    supprimerFilsDroit(fg(a))
```

```
  FIN SI
```

```
  libererMemoire(fg(a))
```

```
FIN SI
```

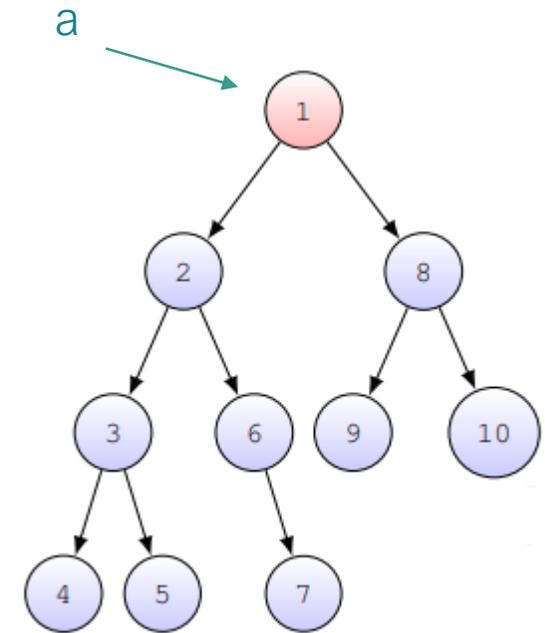
```
FIN
```



# Opérations sur un arbre binaire

- Supprimer fils gauche/ fils droit d'un nœud: illustration

```
PROCEDURE supprimerFilsGauche(a: pointeur sur Arbre)
DEBUT
  SI a EGAL A NULL ALORS
    ERREUR()
  SINON SI existeFilsGauche(a) ALORS
    SI existeFilsGauche(fg(a)) ALORS
      → supprimerFilsGauche(fg(a))
    FIN SI
    SI existeFilsDroit(fg(a)) ALORS
      supprimerFilsDroit(fg(a))
    FIN SI
    libererMemoire(fg(a))
  FIN SI
FIN
```



# Opérations sur un arbre binaire

- Supprimer fils gauche/ fils droit d'un nœud: illustration

```
PROCEDURE supprimerFilsGauche(a: pointeur sur Arbre)
```

```
→ DEBUT
```

```
  SI a EGAL A NULL ALORS
```

```
    ERREUR()
```

```
  SINON SI existeFilsGauche(a) ALORS
```

```
    SI existeFilsGauche(fg(a)) ALORS
```

```
      →      supprimerFilsGauche(fg(a))
```

```
    FIN SI
```

```
    SI existeFilsDroit(fg(a)) ALORS
```

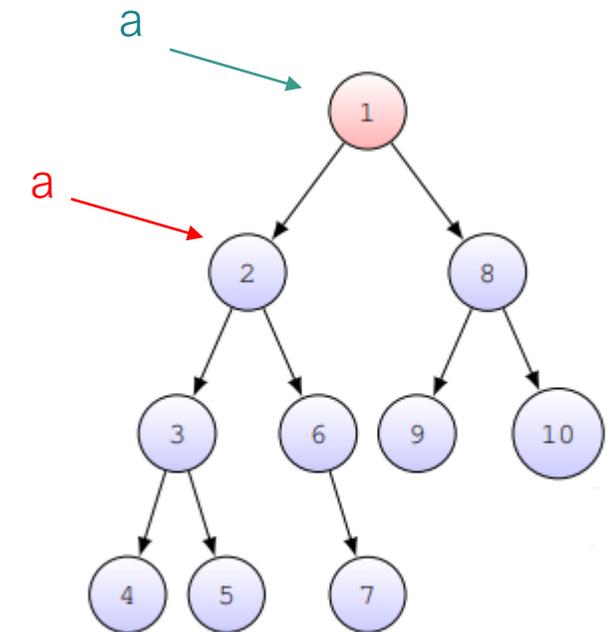
```
      supprimerFilsDroit(fg(a))
```

```
    FIN SI
```

```
    libererMemoire(fg(a))
```

```
  FIN SI
```

```
FIN
```



# Opérations sur un arbre binaire

- Supprimer fils gauche/ fils droit d'un nœud: illustration

```
PROCEDURE supprimerFilsGauche(a: pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a EGAL A NULL ALORS
```

```
    ERREUR()
```

```
  → SINON SI existeFilsGauche(a) ALORS
```

```
    SI existeFilsGauche(fg(a)) ALORS
```

```
      → supprimerFilsGauche(fg(a))
```

```
    FIN SI
```

```
    SI existeFilsDroit(fg(a)) ALORS
```

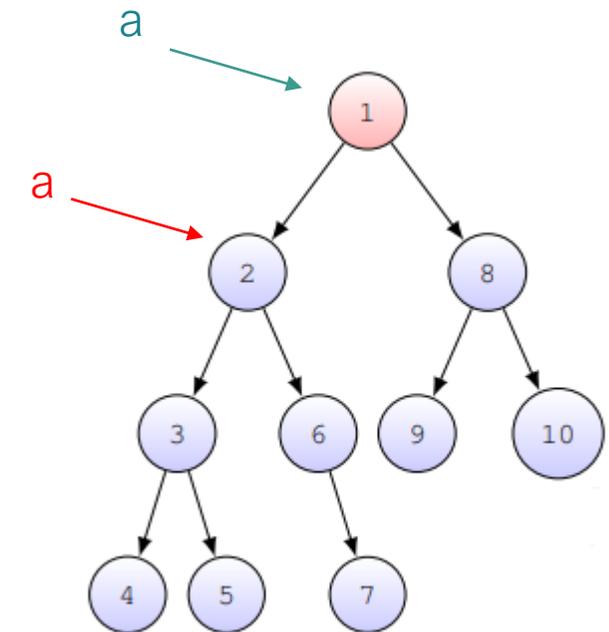
```
      supprimerFilsDroit(fg(a))
```

```
    FIN SI
```

```
    libererMemoire(fg(a))
```

```
  FIN SI
```

```
FIN
```



# Opérations sur un arbre binaire

- Supprimer fils gauche/ fils droit d'un nœud: illustration

```
PROCEDURE supprimerFilsGauche(a: pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a EGAL A NULL ALORS
```

```
    ERREUR()
```

```
  SINON SI existeFilsGauche(a) ALORS
```

```
    → SI existeFilsGauche(fg(a)) ALORS
```

```
      → supprimerFilsGauche(fg(a))
```

```
    FIN SI
```

```
    SI existeFilsDroit(fg(a)) ALORS
```

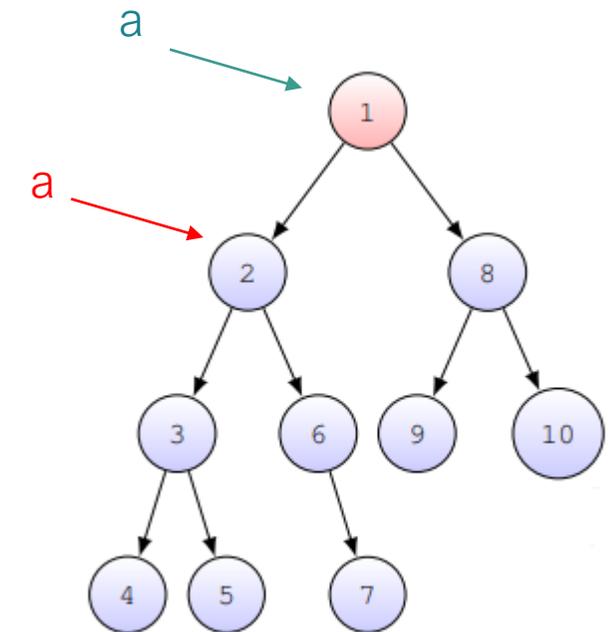
```
      supprimerFilsDroit(fg(a))
```

```
    FIN SI
```

```
    libererMemoire(fg(a))
```

```
  FIN SI
```

```
FIN
```



# Opérations sur un arbre binaire

- Supprimer fils gauche/ fils droit d'un nœud: illustration

```
PROCEDURE supprimerFilsGauche(a: pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a EGAL A NULL ALORS
```

```
    ERREUR()
```

```
  SINON SI existeFilsGauche(a) ALORS
```

```
    SI existeFilsGauche(fg(a)) ALORS
```

```
        supprimerFilsGauche(fg(a))
```

```
    FIN SI
```

```
    SI existeFilsDroit(fg(a)) ALORS
```

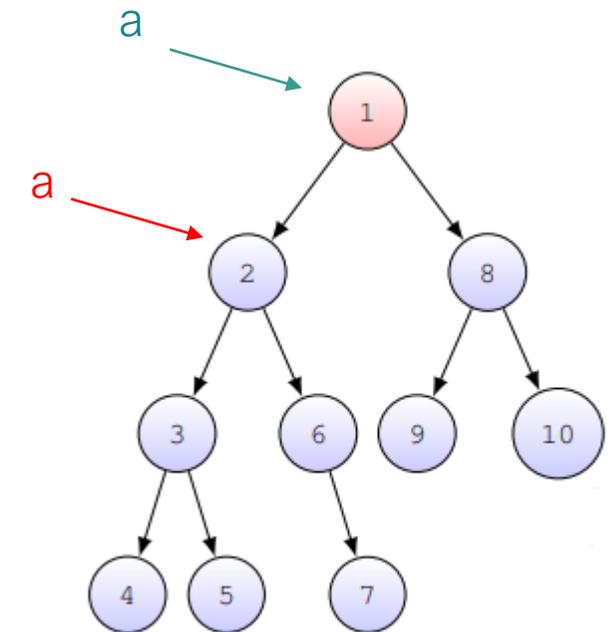
```
      supprimerFilsDroit(fg(a))
```

```
    FIN SI
```

```
    libererMemoire(fg(a))
```

```
  FIN SI
```

```
FIN
```



# Opérations sur un arbre binaire

- Supprimer fils gauche/ fils droit d'un nœud: illustration

```
PROCEDURE supprimerFilsGauche(a: pointeur sur Arbre)
```

```
→ DEBUT
```

```
  SI a EGAL A NULL ALORS
```

```
    ERREUR()
```

```
  SINON SI existeFilsGauche(a) ALORS
```

```
    SI existeFilsGauche(fg(a)) ALORS
```

```
      → supprimerFilsGauche(fg(a))
```

```
    FIN SI
```

```
    SI existeFilsDroit(fg(a)) ALORS
```

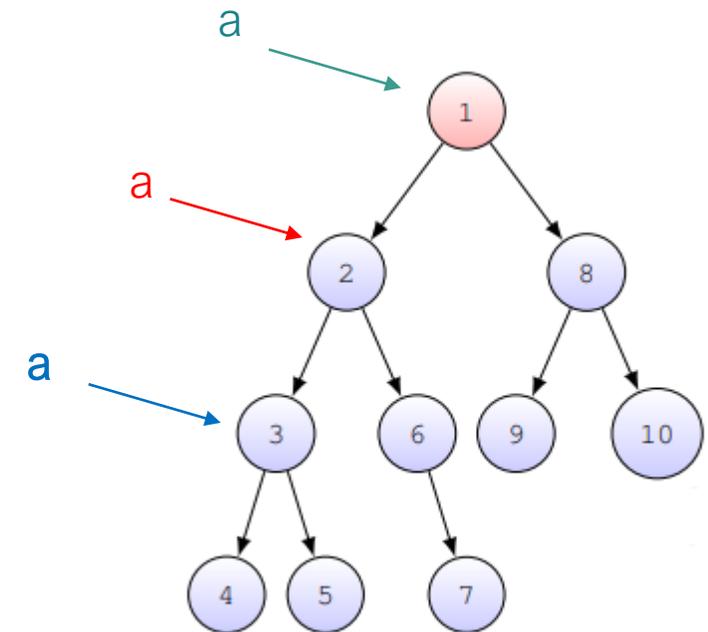
```
      supprimerFilsDroit(fg(a))
```

```
    FIN SI
```

```
    libererMemoire(fg(a))
```

```
  FIN SI
```

```
FIN
```



# Opérations sur un arbre binaire

- Supprimer fils gauche/ fils droit d'un nœud: illustration

```
PROCEDURE supprimerFilsGauche(a: pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a EGAL A NULL ALORS
```

```
    ERREUR()
```

```
  → SINON SI existeFilsGauche(a) ALORS
```

```
    SI existeFilsGauche(fg(a)) ALORS
```

```
      → supprimerFilsGauche(fg(a))
```

```
    FIN SI
```

```
    SI existeFilsDroit(fg(a)) ALORS
```

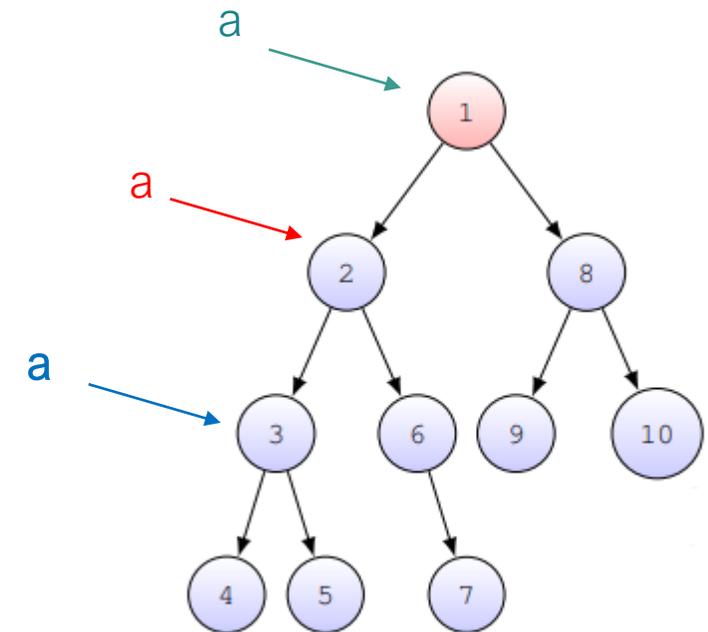
```
      supprimerFilsDroit(fg(a))
```

```
    FIN SI
```

```
    libererMemoire(fg(a))
```

```
  FIN SI
```

```
FIN
```



# Opérations sur un arbre binaire

- Supprimer fils gauche/ fils droit d'un nœud: illustration

```
PROCEDURE supprimerFilsGauche(a: pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a EGAL A NULL ALORS
```

```
    ERREUR()
```

```
  SINON SI existeFilsGauche(a) ALORS
```

```
    → SI existeFilsGauche(fg(a)) ALORS
```

```
      → supprimerFilsGauche(fg(a))
```

```
    FIN SI
```

```
    SI existeFilsDroit(fg(a)) ALORS
```

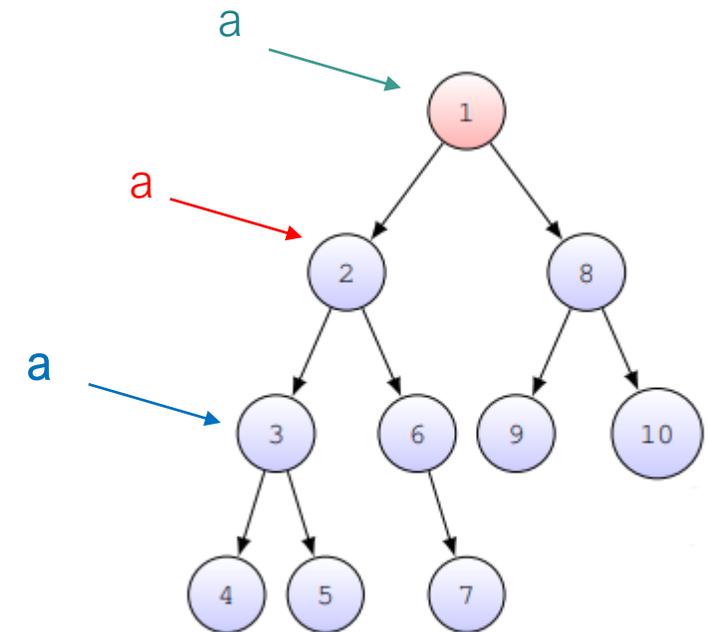
```
      supprimerFilsDroit(fg(a))
```

```
    FIN SI
```

```
    libererMemoire(fg(a))
```

```
  FIN SI
```

```
FIN
```



# Opérations sur un arbre binaire

- Supprimer fils gauche/ fils droit d'un nœud: illustration

```
PROCEDURE supprimerFilsGauche(a: pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a EGAL A NULL ALORS
```

```
    ERREUR()
```

```
  SINON SI existeFilsGauche(a) ALORS
```

```
    SI existeFilsGauche(fg(a)) ALORS
```

```
        supprimerFilsGauche(fg(a))
```

```
    FIN SI
```

```
     SI existeFilsDroit(fg(a)) ALORS
```

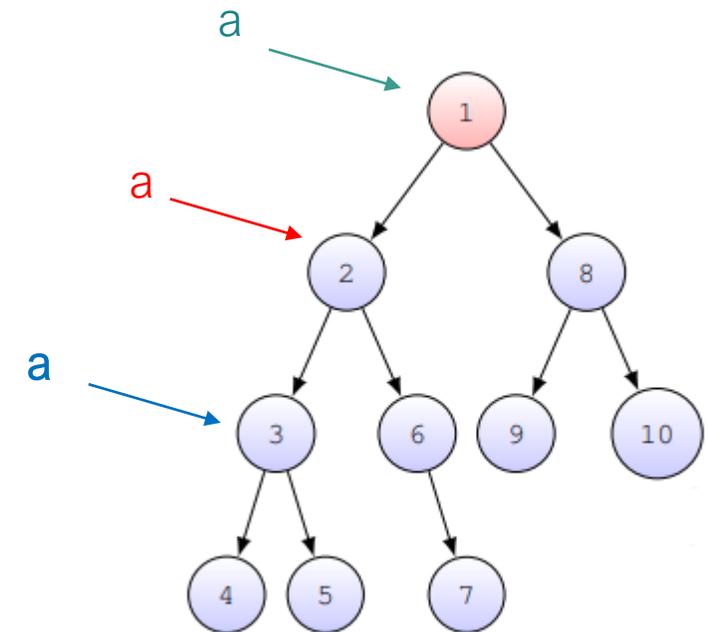
```
      supprimerFilsDroit(fg(a))
```

```
    FIN SI
```

```
    libererMemoire(fg(a))
```

```
  FIN SI
```

```
FIN
```



# Opérations sur un arbre binaire

- Supprimer fils gauche/ fils droit d'un nœud: illustration

```
PROCEDURE supprimerFilsGauche(a: pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a EGAL A NULL ALORS
```

```
    ERREUR()
```

```
  SINON SI existeFilsGauche(a) ALORS
```

```
    SI existeFilsGauche(fg(a)) ALORS
```

```
      →      →      supprimerFilsGauche(fg(a))
```

```
    FIN SI
```

```
    SI existeFilsDroit(fg(a)) ALORS
```

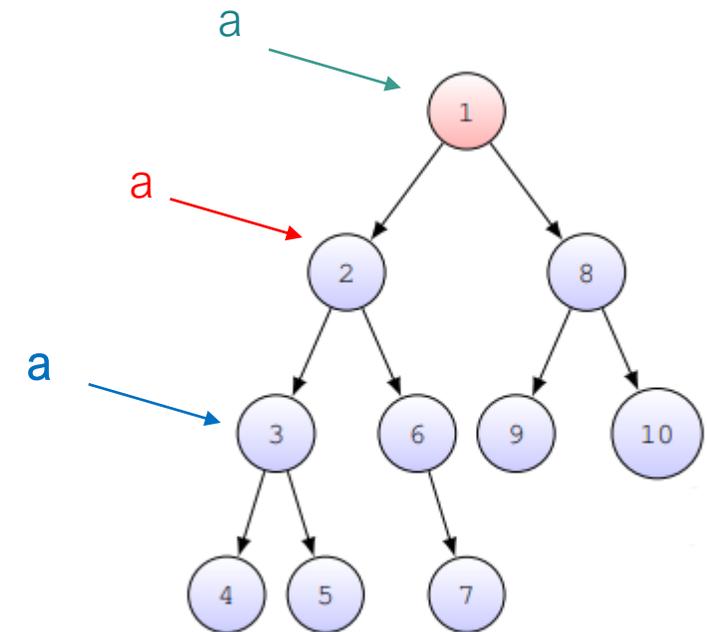
```
      supprimerFilsDroit(fg(a))
```

```
    FIN SI
```

```
      →      libererMemoire(fg(a))
```

```
  FIN SI
```

```
FIN
```



# Opérations sur un arbre binaire

- Supprimer fils gauche/ fils droit d'un nœud: illustration

```
PROCEDURE supprimerFilsGauche(a: pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a EGAL A NULL ALORS
```

```
    ERREUR()
```

```
  SINON SI existeFilsGauche(a) ALORS
```

```
    SI existeFilsGauche(fg(a)) ALORS
```

```
      → →      supprimerFilsGauche(fg(a))
```

```
    FIN SI
```

```
    SI existeFilsDroit(fg(a)) ALORS
```

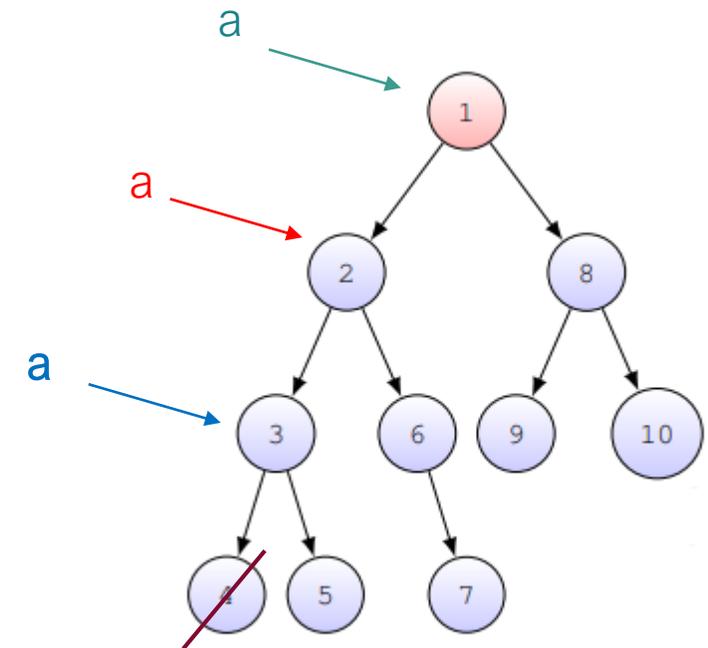
```
      supprimerFilsDroit(fg(a))
```

```
    FIN SI
```

```
    libererMemoire(fg(a))
```

```
  → FIN SI
```

```
FIN
```



# Opérations sur un arbre binaire

- Supprimer fils gauche/ fils droit d'un nœud: illustration

```
PROCEDURE supprimerFilsGauche(a: pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a EGAL A NULL ALORS
```

```
    ERREUR()
```

```
  SINON SI existeFilsGauche(a) ALORS
```

```
    SI existeFilsGauche(fg(a)) ALORS
```

```
      → supprimerFilsGauche(fg(a))
```

```
    FIN SI
```

```
    SI existeFilsDroit(fg(a)) ALORS
```

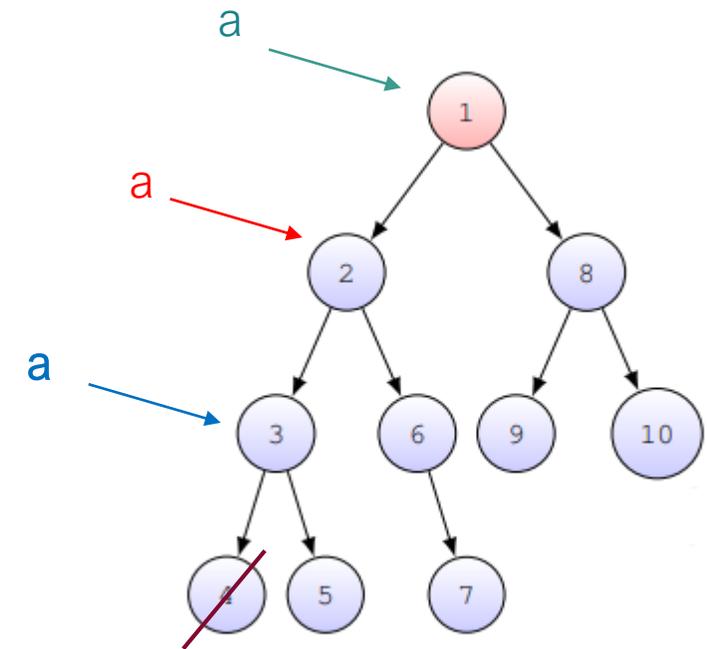
```
      supprimerFilsDroit(fg(a))
```

```
    FIN SI
```

```
    libererMemoire(fg(a))
```

```
  FIN SI
```

```
→ FIN
```



# Opérations sur un arbre binaire

- Supprimer fils gauche/ fils droit d'un nœud: illustration

```
PROCEDURE supprimerFilsGauche(a: pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a EGAL A NULL ALORS
```

```
    ERREUR()
```

```
  SINON SI existeFilsGauche(a) ALORS
```

```
    SI existeFilsGauche(fg(a)) ALORS
```

```
      →      supprimerFilsGauche(fg(a))
```

```
    → FIN SI
```

```
    SI existeFilsDroit(fg(a)) ALORS
```

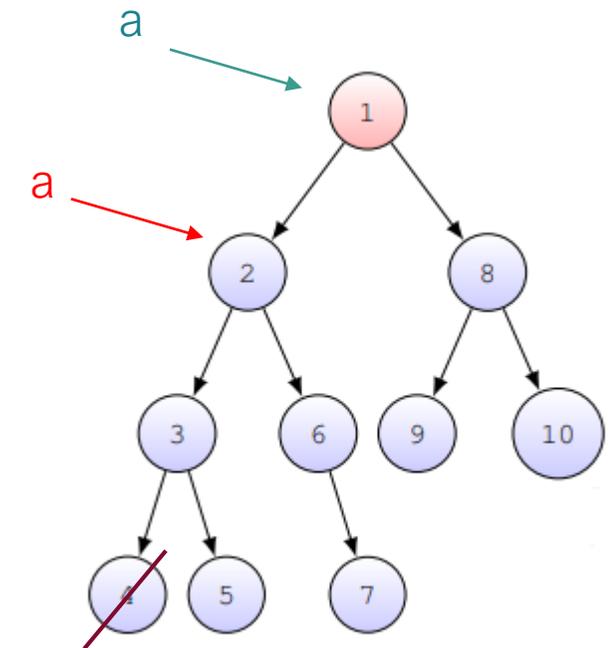
```
      supprimerFilsDroit(fg(a))
```

```
    FIN SI
```

```
    libererMemoire(fg(a))
```

```
  FIN SI
```

```
FIN
```



# Opérations sur un arbre binaire

- Supprimer fils gauche/ fils droit d'un nœud: illustration

```
PROCEDURE supprimerFilsGauche(a: pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a EGAL A NULL ALORS
```

```
    ERREUR()
```

```
  SINON SI existeFilsGauche(a) ALORS
```

```
    SI existeFilsGauche(fg(a)) ALORS
```

```
      → supprimerFilsGauche(fg(a))
```

```
    FIN SI
```

```
    → SI existeFilsDroit(fg(a)) ALORS
```

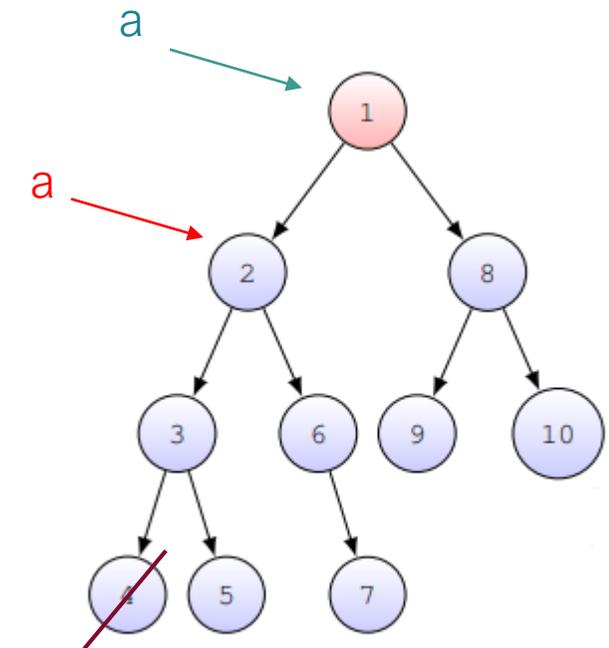
```
      supprimerFilsDroit(fg(a))
```

```
    FIN SI
```

```
    libererMemoire(fg(a))
```

```
  FIN SI
```

```
FIN
```



# Opérations sur un arbre binaire

- Supprimer fils gauche/ fils droit d'un nœud: illustration

```
PROCEDURE supprimerFilsGauche(a: pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a EGAL A NULL ALORS
```

```
    ERREUR()
```

```
  SINON SI existeFilsGauche(a) ALORS
```

```
    SI existeFilsGauche(fg(a)) ALORS
```

```
      →      supprimerFilsGauche(fg(a))
```

```
    FIN SI
```

```
    SI existeFilsDroit(fg(a)) ALORS
```

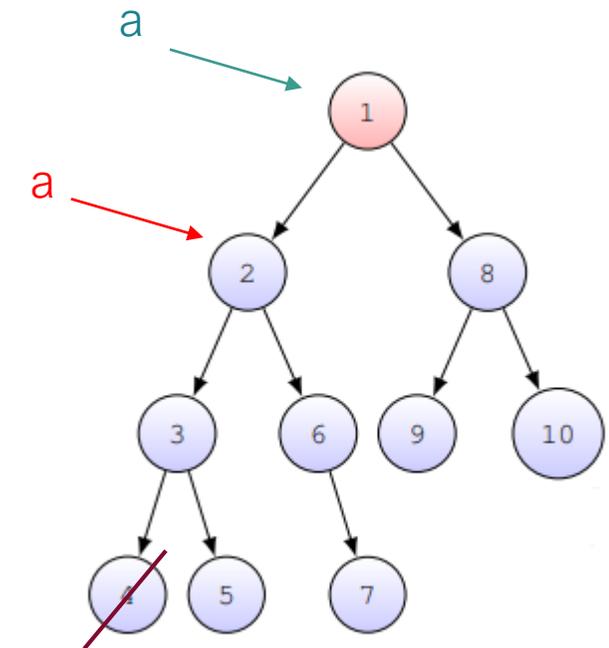
```
      →      supprimerFilsDroit(fg(a))
```

```
    FIN SI
```

```
    libererMemoire(fg(a))
```

```
  FIN SI
```

```
FIN
```



# Opérations sur un arbre binaire

- Supprimer fils gauche/ fils droit d'un nœud: illustration

```
PROCEDURE supprimerFilsGauche(a: pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a EGAL A NULL ALORS
```

```
    ERREUR()
```

```
  SINON SI existeFilsGauche(a) ALORS
```

```
    SI existeFilsGauche(fg(a)) ALORS
```

```
      → supprimerFilsGauche(fg(a))
```

```
    FIN SI
```

```
    SI existeFilsDroit(fg(a)) ALORS
```

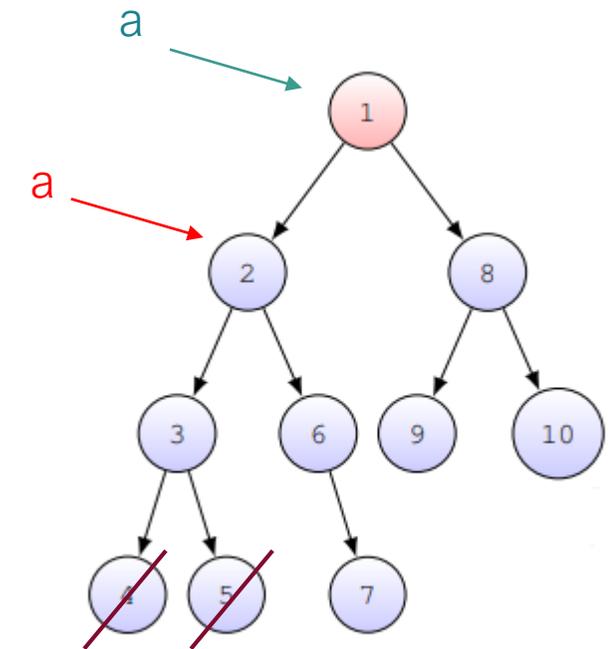
```
      supprimerFilsDroit(fg(a))
```

```
    → FIN SI
```

```
    libererMemoire(fg(a))
```

```
  FIN SI
```

```
FIN
```



# Opérations sur un arbre binaire

- Supprimer fils gauche/ fils droit d'un nœud: illustration

```
PROCEDURE supprimerFilsGauche(a: pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a EGAL A NULL ALORS
```

```
    ERREUR()
```

```
  SINON SI existeFilsGauche(a) ALORS
```

```
    SI existeFilsGauche(fg(a)) ALORS
```

```
      → supprimerFilsGauche(fg(a))
```

```
    FIN SI
```

```
    SI existeFilsDroit(fg(a)) ALORS
```

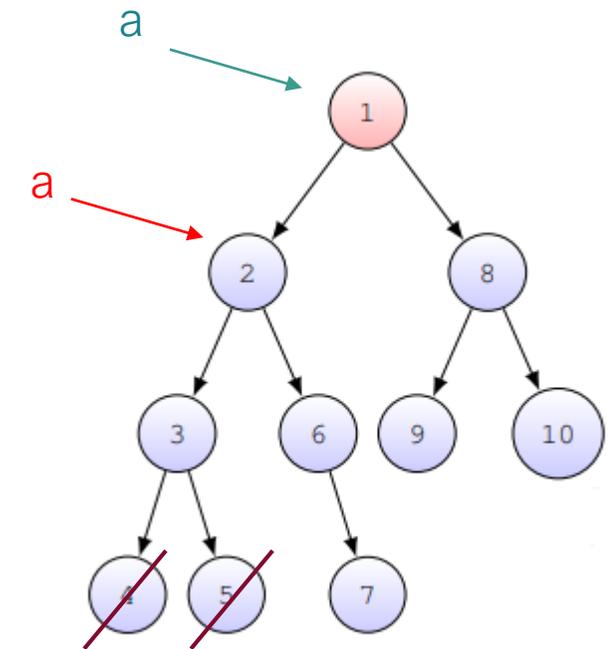
```
      supprimerFilsDroit(fg(a))
```

```
    FIN SI
```

```
      → libererMemoire(fg(a))
```

```
  FIN SI
```

```
FIN
```



# Opérations sur un arbre binaire

- Supprimer fils gauche/ fils droit d'un nœud: illustration

```
PROCEDURE supprimerFilsGauche(a: pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a EGAL A NULL ALORS
```

```
    ERREUR()
```

```
  SINON SI existeFilsGauche(a) ALORS
```

```
    SI existeFilsGauche(fg(a)) ALORS
```

```
      → supprimerFilsGauche(fg(a))
```

```
    FIN SI
```

```
    SI existeFilsDroit(fg(a)) ALORS
```

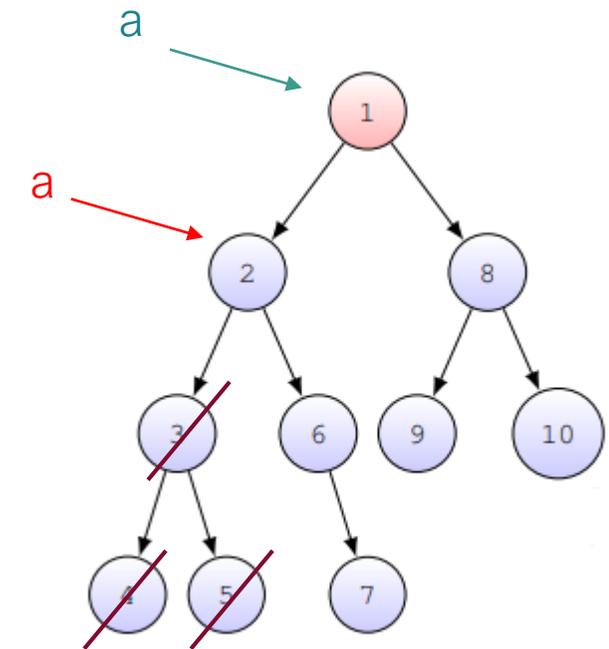
```
      supprimerFilsDroit(fg(a))
```

```
    FIN SI
```

```
    libererMemoire(fg(a))
```

```
  → FIN SI
```

```
FIN
```



# Opérations sur un arbre binaire

- Supprimer fils gauche/ fils droit d'un nœud: illustration

```
PROCEDURE supprimerFilsGauche(a: pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a EGAL A NULL ALORS
```

```
    ERREUR()
```

```
  SINON SI existeFilsGauche(a) ALORS
```

```
    SI existeFilsGauche(fg(a)) ALORS
```

```
      →      supprimerFilsGauche(fg(a))
```

```
    FIN SI
```

```
    SI existeFilsDroit(fg(a)) ALORS
```

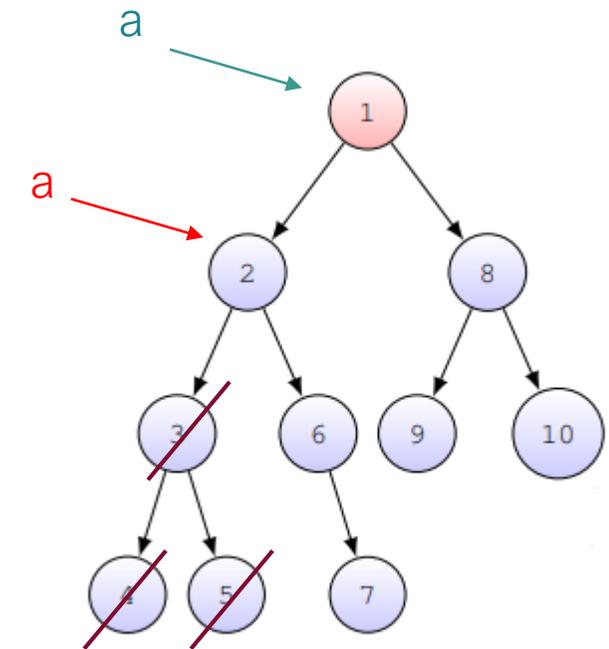
```
      supprimerFilsDroit(fg(a))
```

```
    FIN SI
```

```
    libererMemoire(fg(a))
```

```
  FIN SI
```

```
→ FIN
```



# Opérations sur un arbre binaire

- Supprimer fils gauche/ fils droit d'un nœud: illustration

```
PROCEDURE supprimerFilsGauche(a: pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a EGAL A NULL ALORS
```

```
    ERREUR()
```

```
  SINON SI existeFilsGauche(a) ALORS
```

```
    SI existeFilsGauche(fg(a)) ALORS
```

```
      supprimerFilsGauche(fg(a))
```

```
    → FIN SI
```

```
    SI existeFilsDroit(fg(a)) ALORS
```

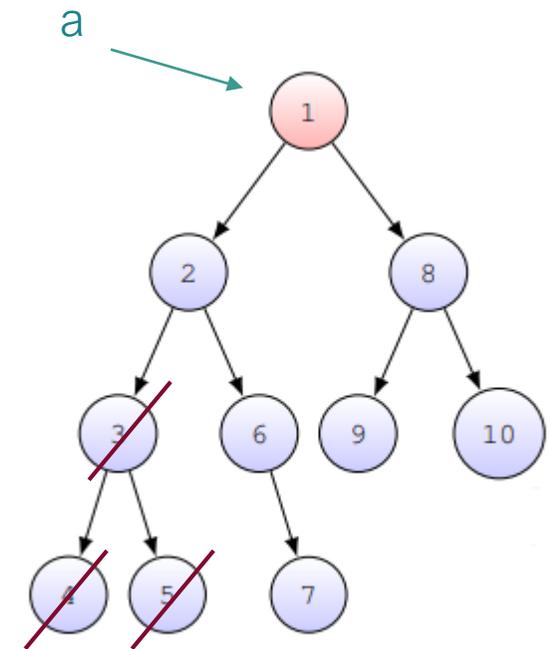
```
      supprimerFilsDroit(fg(a))
```

```
    FIN SI
```

```
    libererMemoire(fg(a))
```

```
  FIN SI
```

```
FIN
```



# Opérations sur un arbre binaire

- Supprimer fils gauche/ fils droit d'un nœud: illustration

```
PROCEDURE supprimerFilsGauche(a: pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a EGAL A NULL ALORS
```

```
    ERREUR()
```

```
  SINON SI existeFilsGauche(a) ALORS
```

```
    SI existeFilsGauche(fg(a)) ALORS
```

```
      supprimerFilsGauche(fg(a))
```

```
    FIN SI
```

```
  → SI existeFilsDroit(fg(a)) ALORS
```

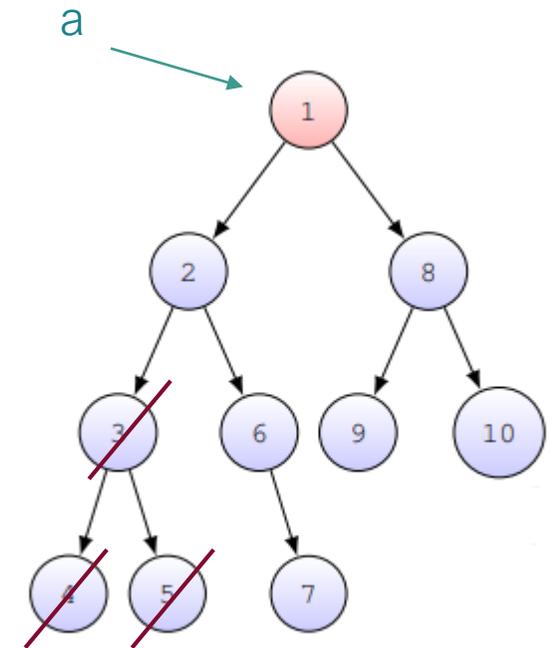
```
    supprimerFilsDroit(fg(a))
```

```
  FIN SI
```

```
  libererMemoire(fg(a))
```

```
FIN SI
```

```
FIN
```



# Opérations sur un arbre binaire

- Supprimer fils gauche/ fils droit d'un nœud: illustration

```
PROCEDURE supprimerFilsGauche(a: pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a EGAL A NULL ALORS
```

```
    ERREUR()
```

```
  SINON SI existeFilsGauche(a) ALORS
```

```
    SI existeFilsGauche(fg(a)) ALORS
```

```
      supprimerFilsGauche(fg(a))
```

```
    FIN SI
```

```
  SI existeFilsDroit(fg(a)) ALORS
```

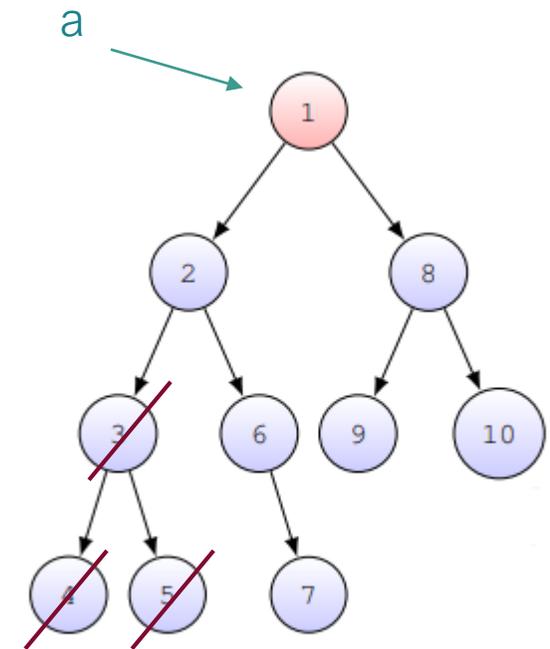
```
    → supprimerFilsDroit(fg(a))
```

```
  FIN SI
```

```
  libererMemoire(fg(a))
```

```
FIN SI
```

```
FIN
```



# Opérations sur un arbre binaire

- Supprimer fils gauche/ fils droit d'un nœud: illustration

```
PROCEDURE supprimerFilsGauche(a: pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a EGAL A NULL ALORS
```

```
    ERREUR()
```

```
  SINON SI existeFilsGauche(a) ALORS
```

```
    SI existeFilsGauche(fg(a)) ALORS
```

```
      supprimerFilsGauche(fg(a))
```

```
    FIN SI
```

```
  SI existeFilsDroit(fg(a)) ALORS
```

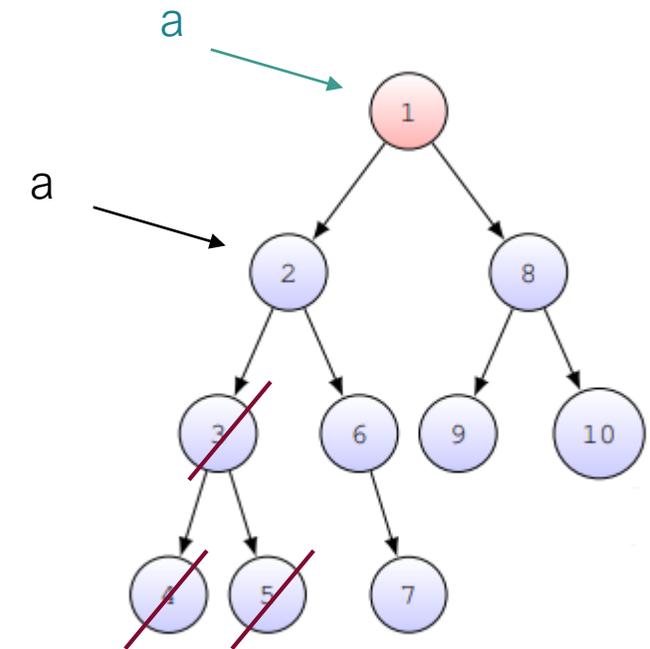
```
    ➡ supprimerFilsDroit(fg(a))
```

```
  FIN SI
```

```
  libererMemoire(fg(a))
```

```
FIN SI
```

```
FIN
```



# Opérations sur un arbre binaire

- Supprimer fils gauche/ fils droit d'un nœud: illustration

```
PROCEDURE supprimerFilsGauche(a: pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a EGAL A NULL ALORS
```

```
    ERREUR()
```

```
  SINON SI existeFilsGauche(a) ALORS
```

```
    SI existeFilsGauche(fg(a)) ALORS
```

```
      supprimerFilsGauche(fg(a))
```

```
    FIN SI
```

```
    SI existeFilsDroit(fg(a)) ALORS
```

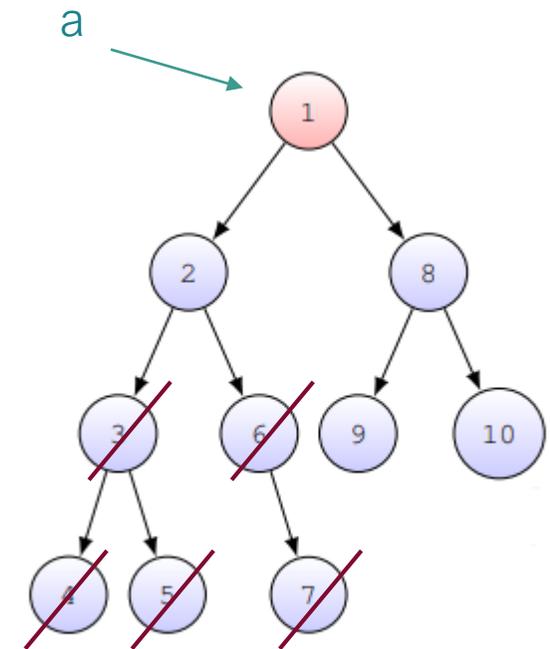
```
      supprimerFilsDroit(fg(a))
```

```
    → FIN SI
```

```
      libererMemoire(fg(a))
```

```
  FIN SI
```

```
FIN
```



# Opérations sur un arbre binaire

- Supprimer fils gauche/ fils droit d'un nœud: illustration

```
PROCEDURE supprimerFilsGauche(a: pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a EGAL A NULL ALORS
```

```
    ERREUR()
```

```
  SINON SI existeFilsGauche(a) ALORS
```

```
    SI existeFilsGauche(fg(a)) ALORS
```

```
      supprimerFilsGauche(fg(a))
```

```
    FIN SI
```

```
    SI existeFilsDroit(fg(a)) ALORS
```

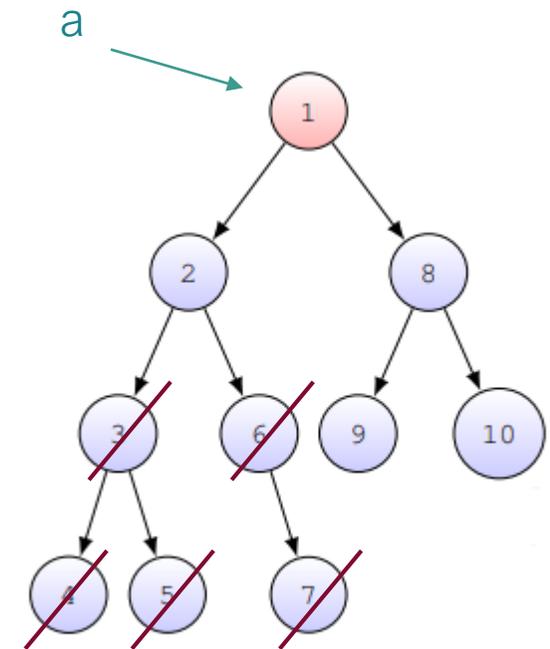
```
      supprimerFilsDroit(fg(a))
```

```
    FIN SI
```

```
      → libererMemoire(fg(a))
```

```
  FIN SI
```

```
FIN
```



# Opérations sur un arbre binaire

- Supprimer fils gauche/ fils droit d'un nœud: illustration

```
PROCEDURE supprimerFilsGauche(a: pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a EGAL A NULL ALORS
```

```
    ERREUR()
```

```
  SINON SI existeFilsGauche(a) ALORS
```

```
    SI existeFilsGauche(fg(a)) ALORS
```

```
      supprimerFilsGauche(fg(a))
```

```
    FIN SI
```

```
    SI existeFilsDroit(fg(a)) ALORS
```

```
      supprimerFilsDroit(fg(a))
```

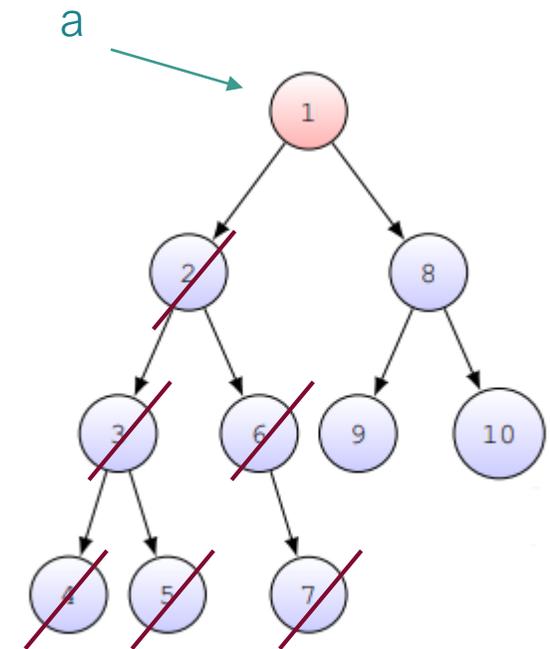
```
    FIN SI
```

```
    libererMemoire(fg(a))
```



```
  FIN SI
```

```
FIN
```



# Opérations sur un arbre binaire

- Supprimer fils gauche/ fils droit d'un nœud: illustration

```
PROCEDURE supprimerFilsGauche(a: pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a EGAL A NULL ALORS
```

```
    ERREUR()
```

```
  SINON SI existeFilsGauche(a) ALORS
```

```
    SI existeFilsGauche(fg(a)) ALORS
```

```
      supprimerFilsGauche(fg(a))
```

```
    FIN SI
```

```
    SI existeFilsDroit(fg(a)) ALORS
```

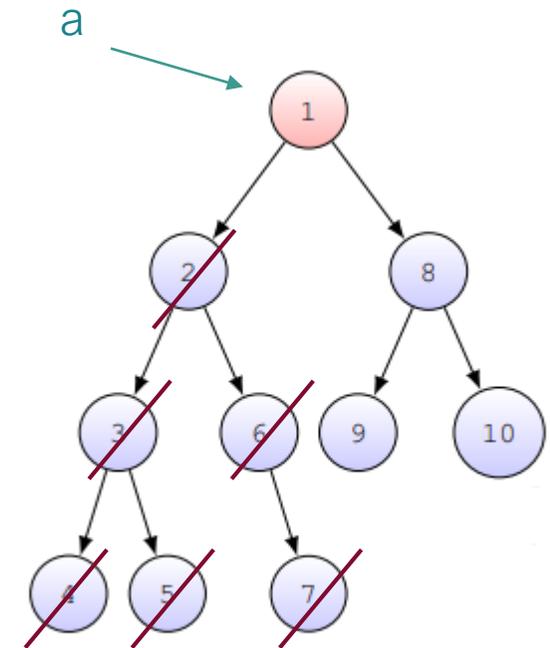
```
      supprimerFilsDroit(fg(a))
```

```
    FIN SI
```

```
    libererMemoire(fg(a))
```

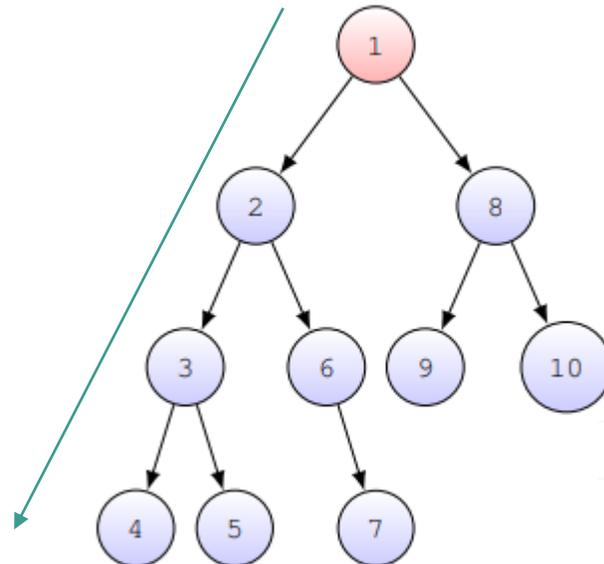
```
  FIN SI
```

```
→ FIN
```



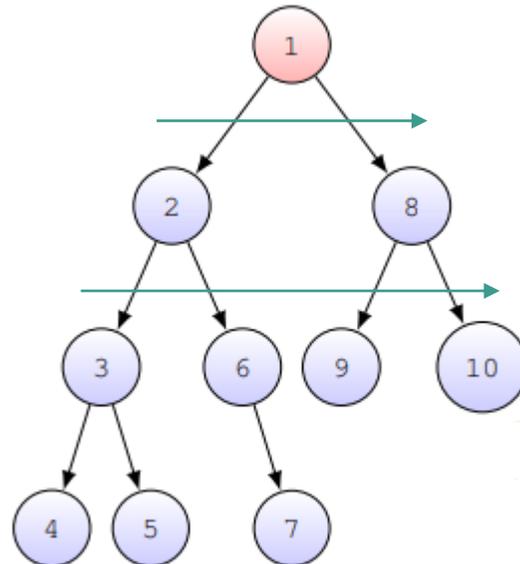
# Parcours d'un arbre

- Un algorithme de parcours d'un arbre permet de passer par tous ses nœuds. On en distingue deux types:
  - **Les parcours en longueur** : lorsque, systématiquement, si l'arbre n'est pas vide, le parcours de l'un des deux sous-arbres est terminé avant que ne commence celui de l'autre. On explore jusqu'au bout une branche pour passer à la suivante (on va vers les fils avant d'aller vers les frères).



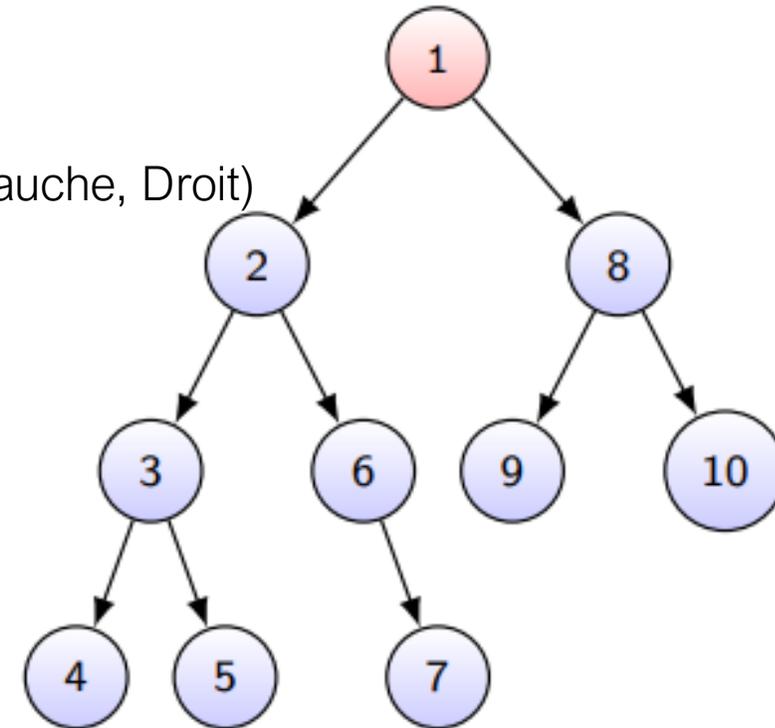
# Parcours d'un arbre

- Un algorithme de parcours d'un arbre permet de passer par tous ses nœuds. On en distingue deux types:
  - **Les parcours en longueur**
  - **Les parcours en largeur** : On explore les nœuds un par un sur un même niveau. On passe ensuite sur le niveau suivant, et ainsi de suite (on va vers les frères avant de parcourir les fils).



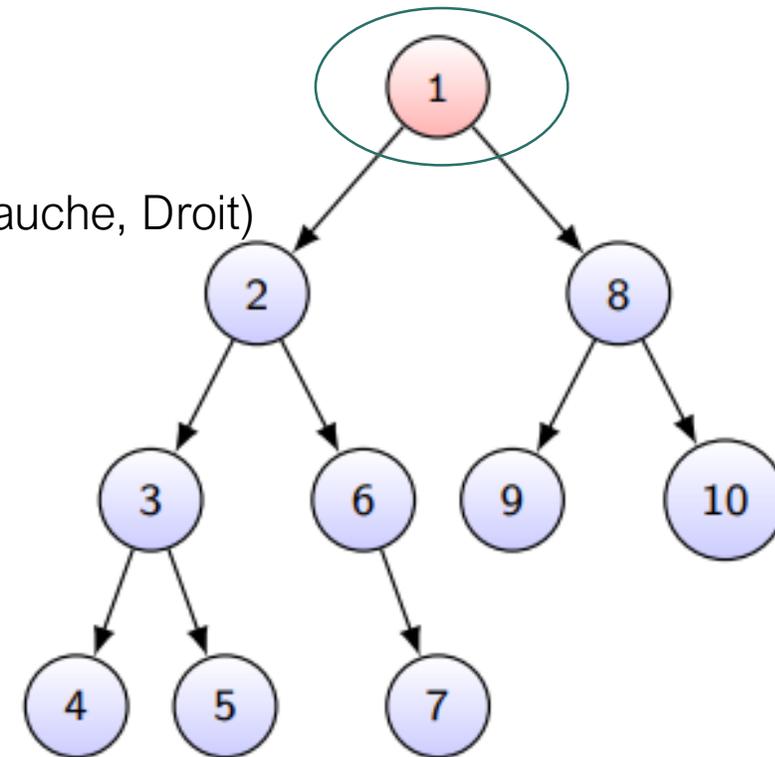
# Parcours en longueur

- Type **PREFIXE** : on visite chaque nœud *avant* de visiter ses fils.
  - 1. On visite la racine de l'arbre
  - 2. Parcours préfixe des sous-arbre de gauche à droite
- Dans un arbre binaire, on l'appelle parcours R G D (Racine, Gauche, Droit)
- **Parcours :**



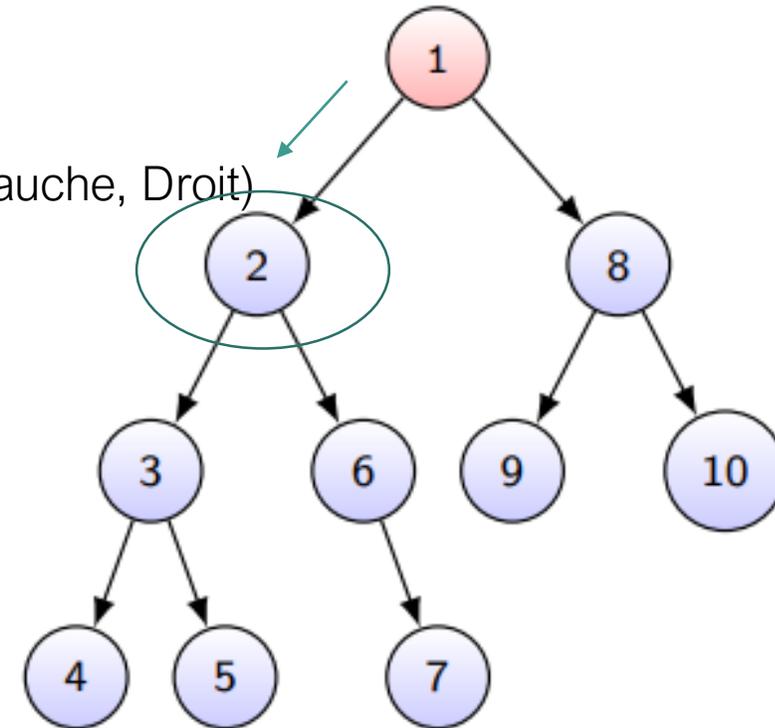
# Parcours en longueur

- Type **PREFIXE** : on visite chaque nœud *avant* de visiter ses fils.
  - 1. On visite la racine de l'arbre
  - 2. Parcours préfixe des sous-arbre de gauche à droite
- Dans un arbre binaire, on l'appelle parcours R G D (Racine, Gauche, Droit)
- **Parcours** : 1



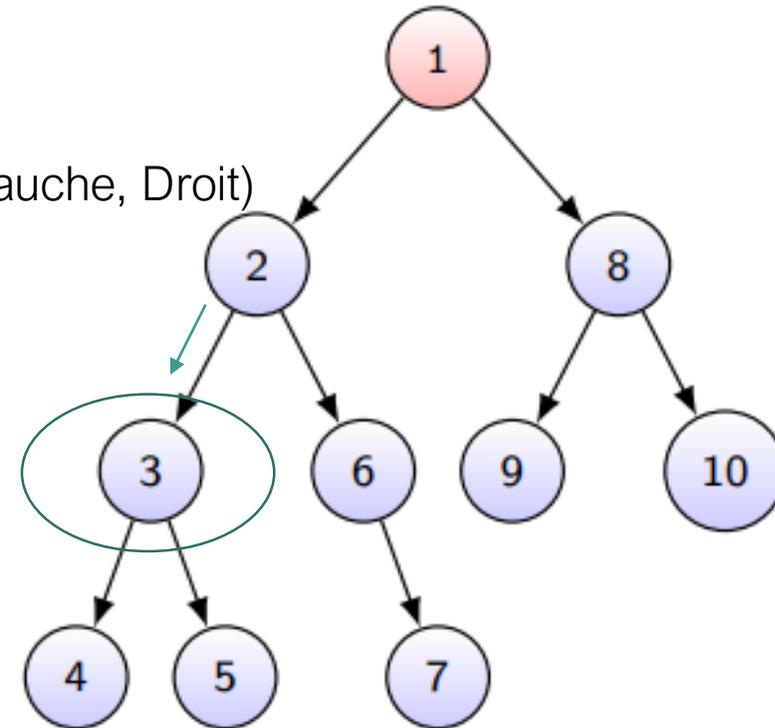
# Parcours en longueur

- Type **PREFIXE** : on visite chaque nœud *avant* de visiter ses fils.
  - 1. On visite la racine de l'arbre
  - 2. Parcours préfixe des sous-arbre de gauche à droite
- Dans un arbre binaire, on l'appelle parcours R G D (Racine, Gauche, Droit)
- **Parcours** : 1, 2



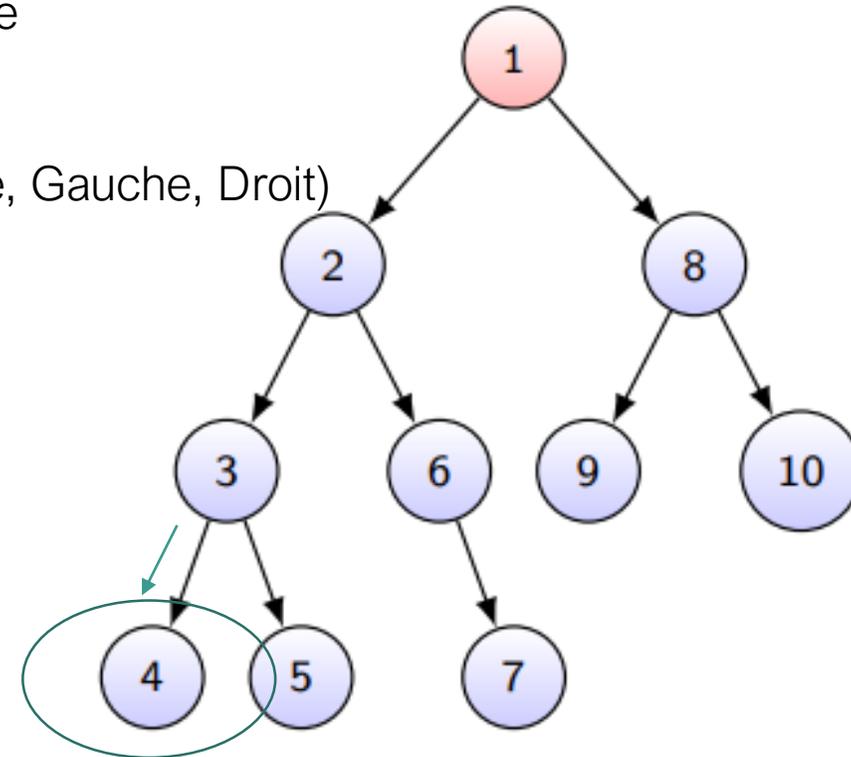
# Parcours en longueur

- Type **PREFIXE** : on visite chaque nœud *avant* de visiter ses fils.
  - 1. On visite la racine de l'arbre
  - 2. Parcours préfixe des sous-arbre de gauche à droite
- Dans un arbre binaire, on l'appelle parcours R G D (Racine, Gauche, Droit)
- **Parcours** : 1, 2, 3



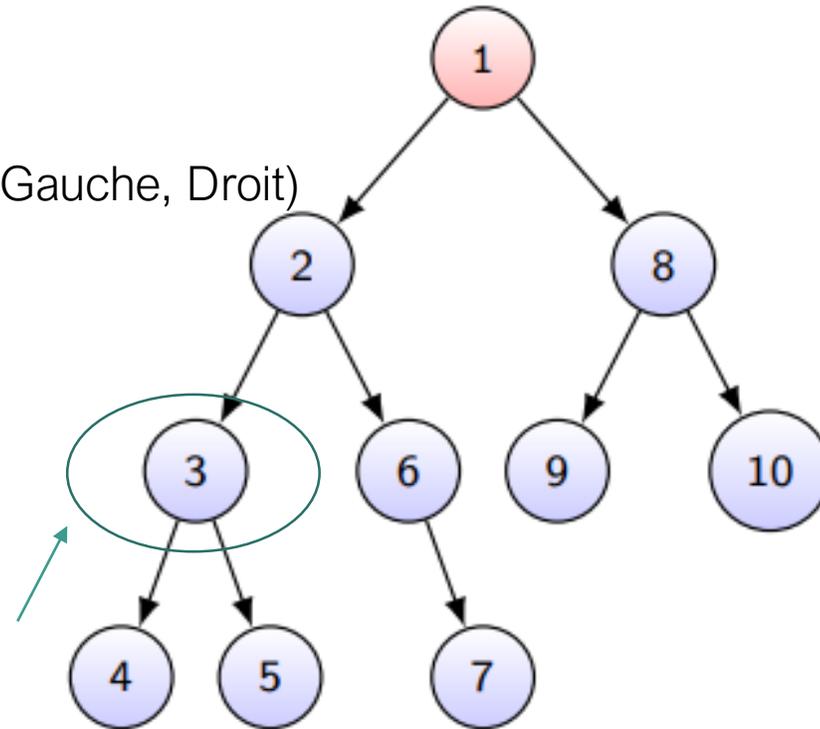
# Parcours en longueur

- Type **PREFIXE** : on visite chaque nœud *avant* de visiter ses fils.
  - 1. On visite la racine de l'arbre
  - 2. Parcours préfixe des sous-arbre de gauche à droite
- Dans un arbre binaire, on l'appelle parcours R G D (Racine, Gauche, Droit)
- **Parcours** : 1, 2, 3, 4



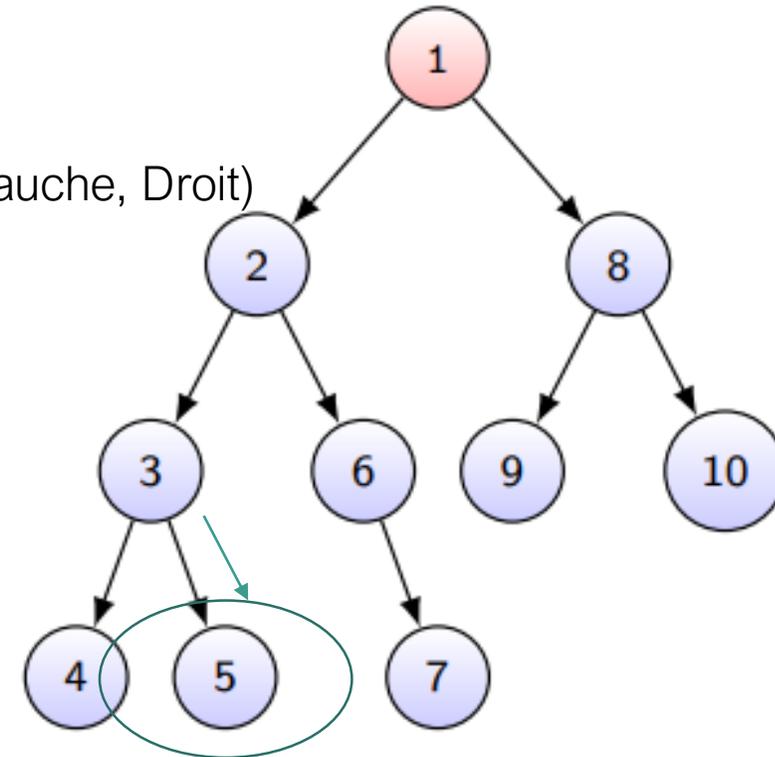
# Parcours en longueur

- Type **PREFIXE** : on visite chaque nœud *avant* de visiter ses fils.
  - 1. On visite la racine de l'arbre
  - 2. Parcours préfixe des sous-arbre de gauche à droite
- Dans un arbre binaire, on l'appelle parcours R G D (Racine, Gauche, Droit)
- **Parcours** : 1, 2, 3, 4



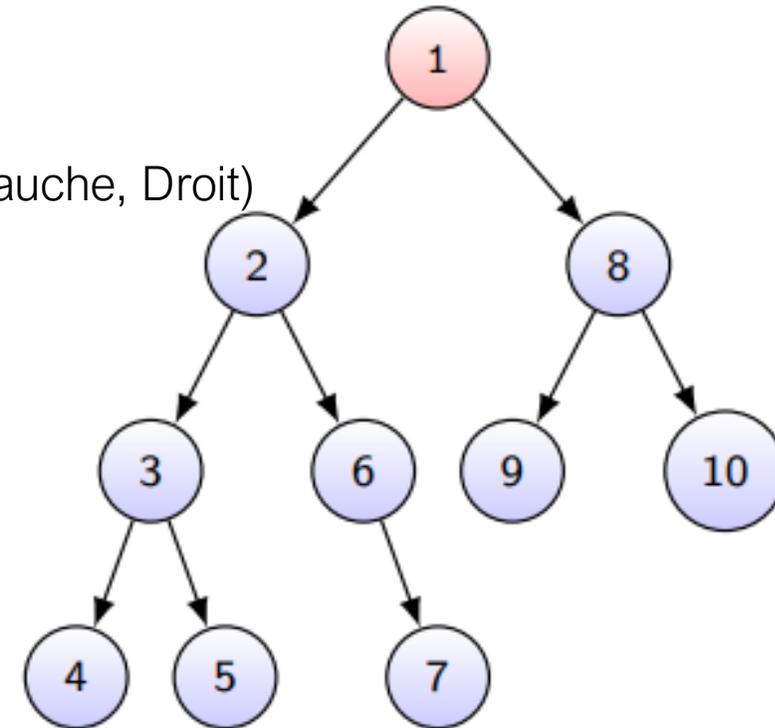
# Parcours en longueur

- Type **PREFIXE** : on visite chaque nœud *avant* de visiter ses fils.
  - 1. On visite la racine de l'arbre
  - 2. Parcours préfixe des sous-arbre de gauche à droite
- Dans un arbre binaire, on l'appelle parcours R G D (Racine, Gauche, Droit)
- **Parcours** : 1, 2, 3, 4, 5



# Parcours en longueur

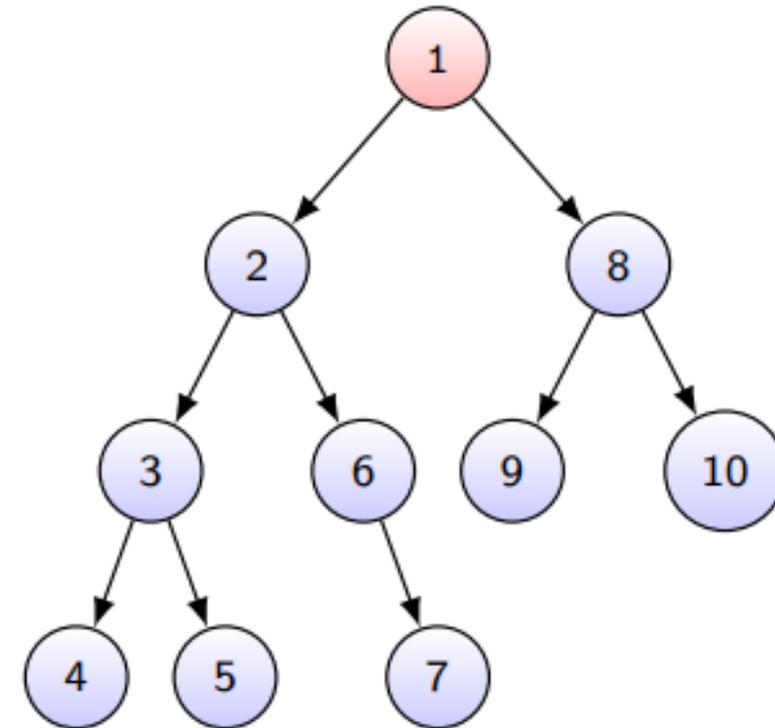
- Type **PREFIXE** : on visite chaque nœud *avant* de visiter ses fils.
  - 1. On visite la racine de l'arbre
  - 2. Parcours préfixe des sous-arbre de gauche à droite
- Dans un arbre binaire, on l'appelle parcours R G D (Racine, Gauche, Droit)
- **Parcours** : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10



# Parcours en longueur

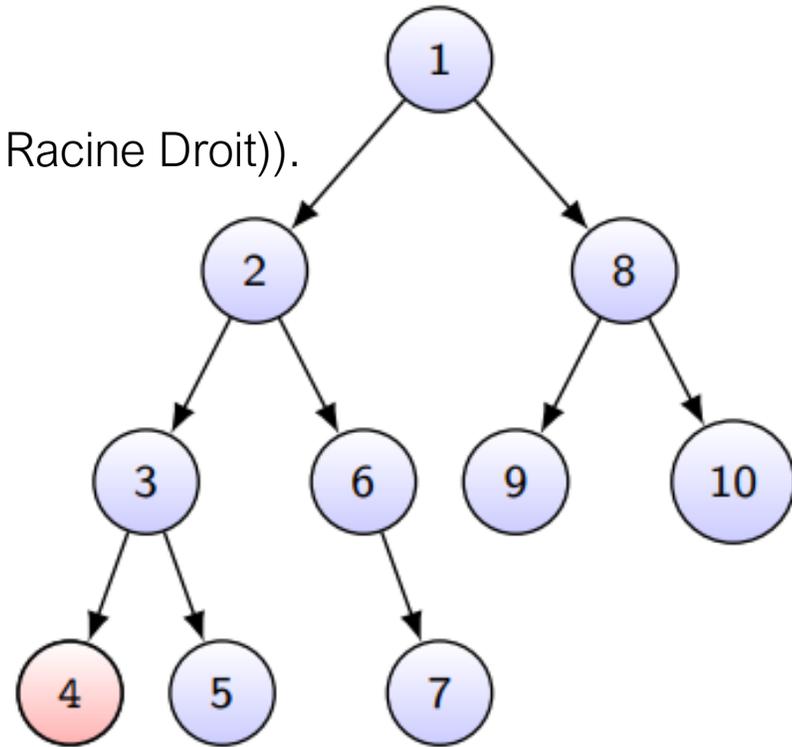
- Type **PREFIXE** : on visite chaque nœud *avant* de visiter ses fils.
  - 1. On visite la racine de l'arbre
  - 2. Parcours préfixe des sous-arbre de gauche à droite
- Algorithme :

```
PROCEDURE parcoursPrefixe(a : pointeur sur Arbre)
DEBUT
    SI a DIFFERENT DE NULL ALORS
        traiter( elmt(a) )
        parcoursPrefixe( fg(a) )
        parcoursPrefixe( fd(a) )
    FIN SI
FIN
```



# Parcours en longueur

- **Type INFIXE** : parcours en « triangle » (dans un arbre binaire seulement) : on visite chaque nœud après son fils à gauche et avant son fils à droite.
  - Parcours infixe du sous-arbre gauche
  - Visite de la racine
  - Parcours infixe du sous-arbre droit
- Dans un arbre binaire, on l'appelle parcours G R D (Gauche Racine Droit)).
- **Parcours** : 4, 3, 5, 2, 6, 7, 1, 9, 8, 10



# Parcours en longueur

- Algorithme :

```
PROCEDURE parcoursInfixe(a : pointeur sur Arbre)
```

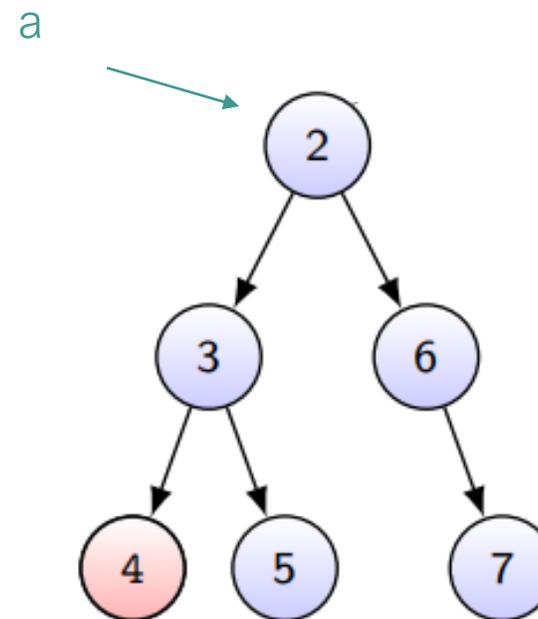
```
→ DEBUT
```

```
  SI a DIFFERENT DE NULL ALORS  
    parcoursInfixe(fg(a))  
    traiter(elm(a))  
    parcoursInfixe(fd(a))
```

```
  FIN SI
```

```
FIN
```

Parcours :



# Parcours en longueur

- Algorithme :

```
PROCEDURE parcoursInfixe(a : pointeur sur Arbre)
```

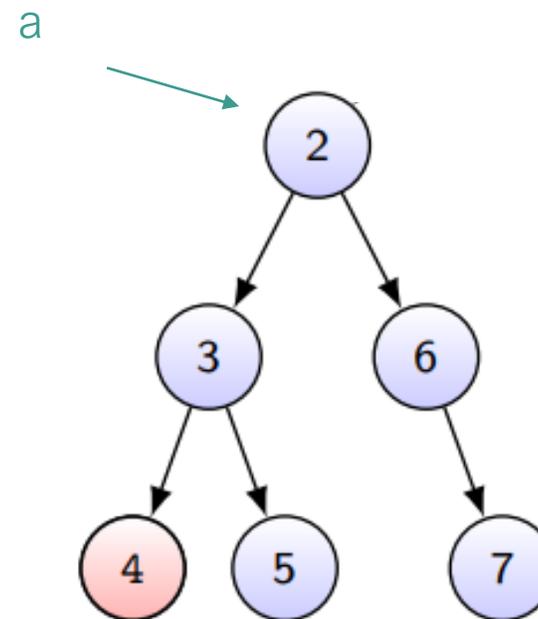
```
DEBUT
```

```
  → SI a DIFFERENT DE NULL ALORS  
    parcoursInfixe(fg(a))  
    traiter(elm(a))  
    parcoursInfixe(fd(a))
```

```
  FIN SI
```

```
FIN
```

Parcours :



# Parcours en longueur

- Algorithme :

```
PROCEDURE parcoursInfixe(a : pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a DIFFERENT DE NULL ALORS
```

```
    ➔ parcoursInfixe(fg(a))
```

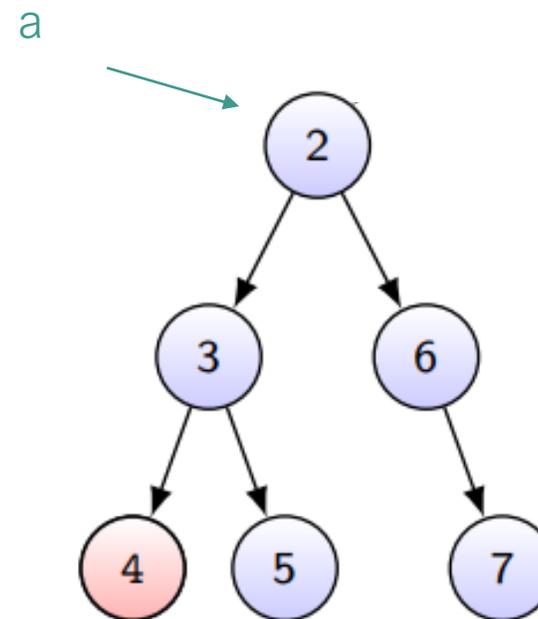
```
    traiter(elm(a))
```

```
    parcoursInfixe(fd(a))
```

```
  FIN SI
```

```
FIN
```

Parcours :



# Parcours en longueur

- Algorithme :

```
PROCEDURE parcoursInfixe(a : pointeur sur Arbre)
```

```
→ DEBUT
```

```
  SI a DIFFERENT DE NULL ALORS
```

```
    → parcoursInfixe(fg(a))
```

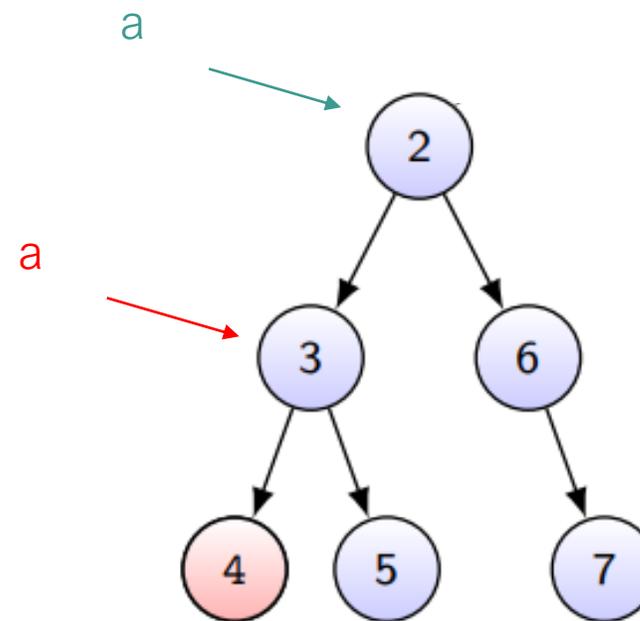
```
    traiter(elm(a))
```

```
    parcoursInfixe(fd(a))
```

```
  FIN SI
```

```
FIN
```

Parcours :



# Parcours en longueur

- Algorithme :

```
PROCEDURE parcoursInfixe(a : pointeur sur Arbre)
```

```
DEBUT
```

```
  → SI a DIFFERENT DE NULL ALORS
```

```
    → parcoursInfixe(fg(a))
```

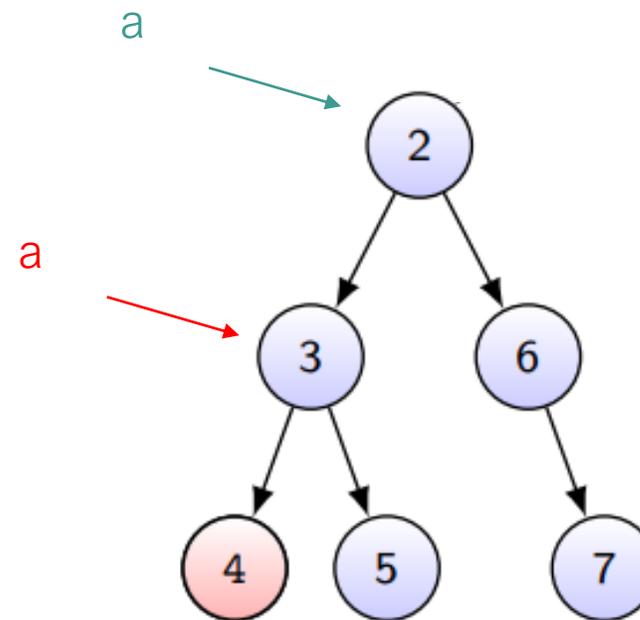
```
      traiter(elm(a))
```

```
      parcoursInfixe(fd(a))
```

```
  FIN SI
```

```
FIN
```

Parcours :



# Parcours en longueur

- Algorithme :

```
PROCEDURE parcoursInfixe(a : pointeur sur Arbre)
```

```
DEBUT
```

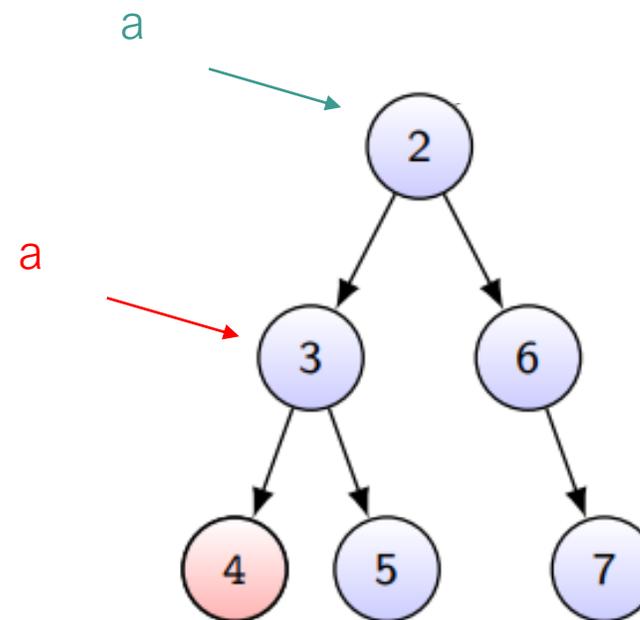
```
  SI a DIFFERENT DE NULL ALORS
```

```
    → parcoursInfixe(fg(a))  
    traiter(elm(a))  
    → parcoursInfixe(fd(a))
```

```
  FIN SI
```

```
FIN
```

Parcours :



# Parcours en longueur

- Algorithme :

PROCEDURE parcoursInfixe(a : pointeur sur Arbre)

➔ DEBUT

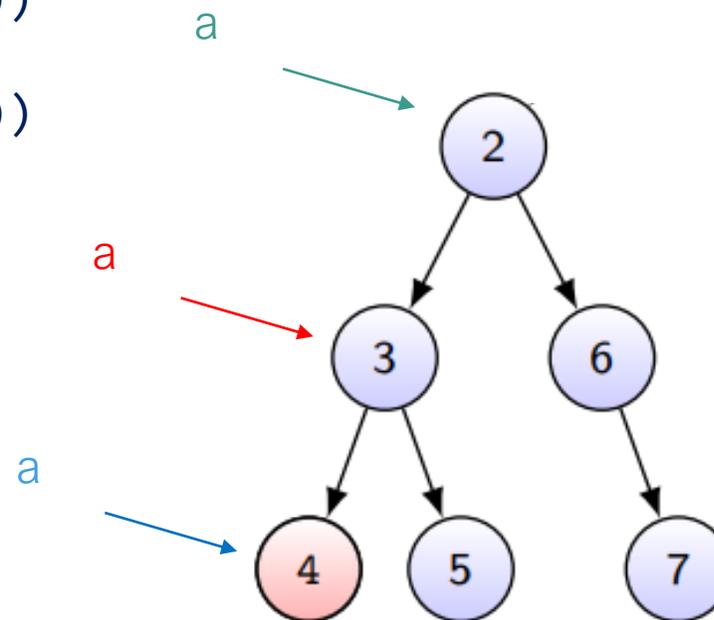
SI a DIFFERENT DE NULL ALORS

➔ ➔ parcoursInfixe(fg(a))  
traiter(elm(a))  
parcoursInfixe(fd(a))

FIN SI

FIN

Parcours :



# Parcours en longueur

- Algorithme :

```
PROCEDURE parcoursInfixe(a : pointeur sur Arbre)
```

```
DEBUT
```

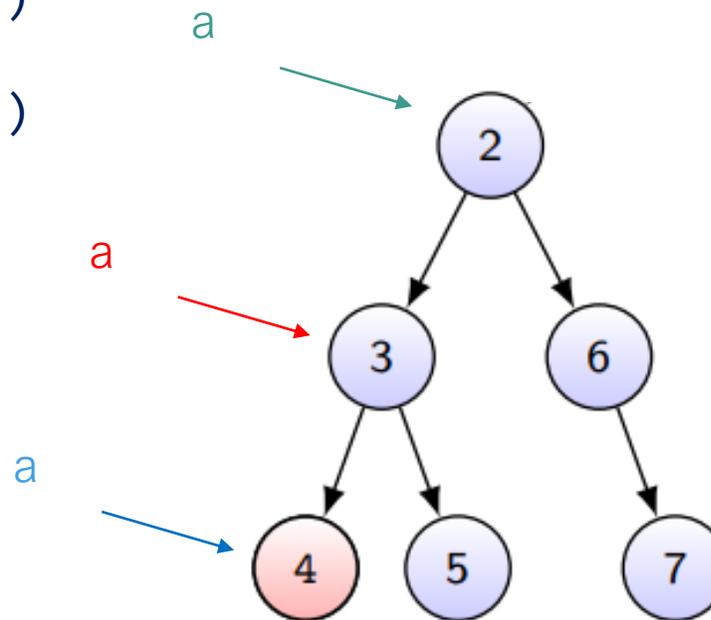
```
  → SI a DIFFERENT DE NULL ALORS
```

```
    → parcoursInfixe(fg(a))  
    traiter(elm(a))  
    parcoursInfixe(fd(a))
```

```
  FIN SI
```

```
FIN
```

Parcours :



# Parcours en longueur

- Algorithme :

```
PROCEDURE parcoursInfixe(a : pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a DIFFERENT DE NULL ALORS
```

```
    → → → parcoursInfixe(fg(a))
```

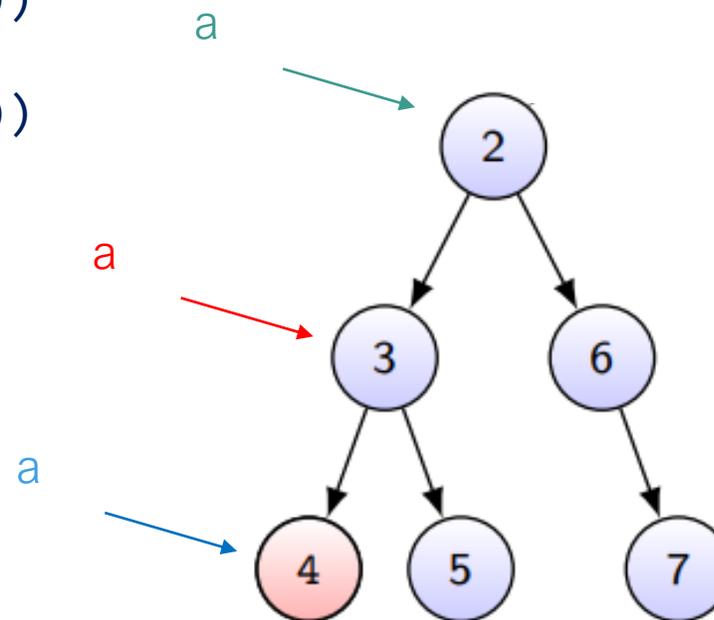
```
    traiter(elm(a))
```

```
    parcoursInfixe(fd(a))
```

```
  FIN SI
```

```
FIN
```

Parcours :



# Parcours en longueur

- Algorithme :

PROCEDURE parcoursInfixe(a : pointeur sur Arbre)

➔ DEBUT

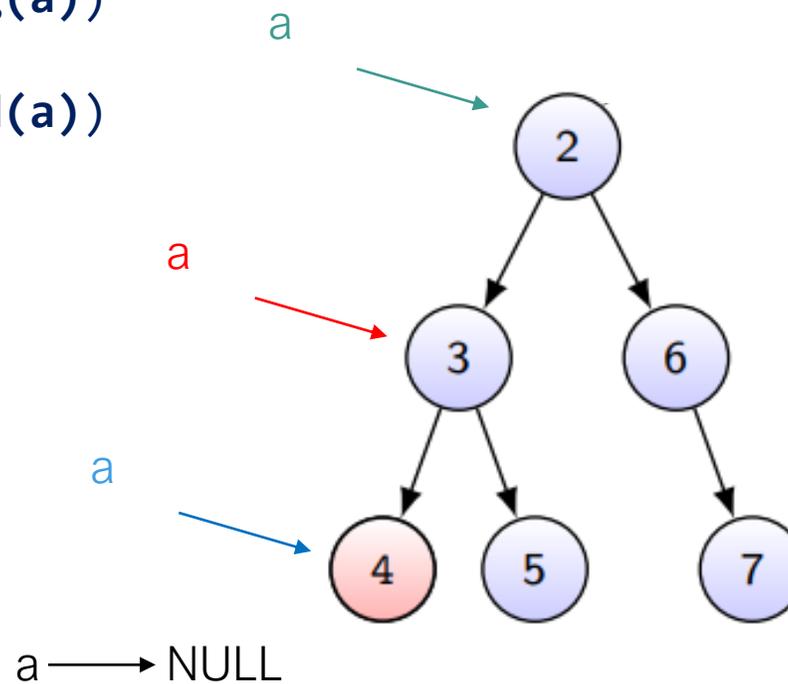
SI a DIFFERENT DE NULL ALORS

➔ ➔ ➔ parcoursInfixe(fg(a))  
traiter(elm(a))  
parcoursInfixe(fd(a))

FIN SI

FIN

Parcours :



# Parcours en longueur

- Algorithme :

```
PROCEDURE parcoursInfixe(a : pointeur sur Arbre)
```

```
DEBUT
```

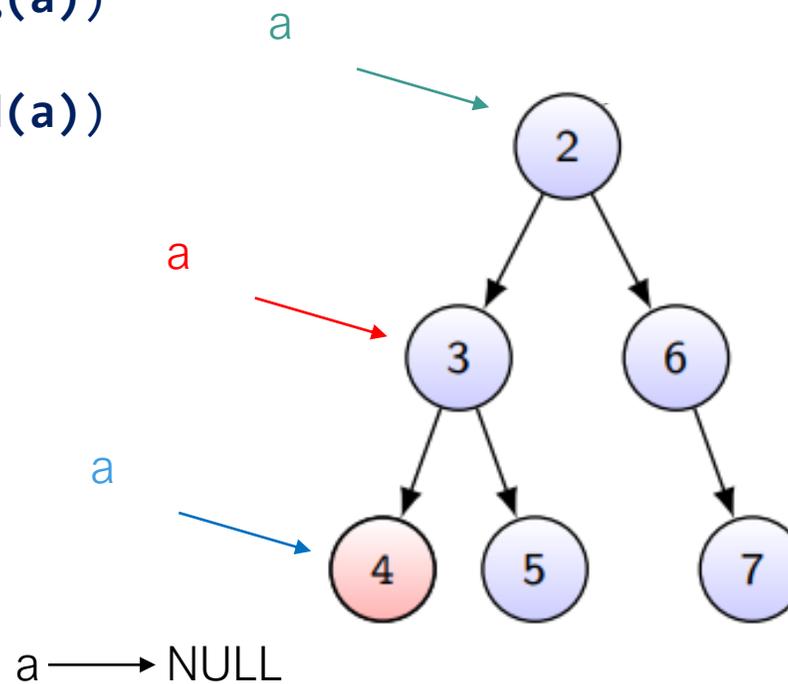
```
  ──▶ SI a DIFFERENT DE NULL ALORS
```

```
    ─▶ ─▶ ─▶ parcoursInfixe(fg(a))  
          traiter(elm(a))  
          parcoursInfixe(fd(a))
```

```
  FIN SI
```

```
FIN
```

Parcours :



# Parcours en longueur

- Algorithme :

```
PROCEDURE parcoursInfixe(a : pointeur sur Arbre)
```

```
DEBUT
```

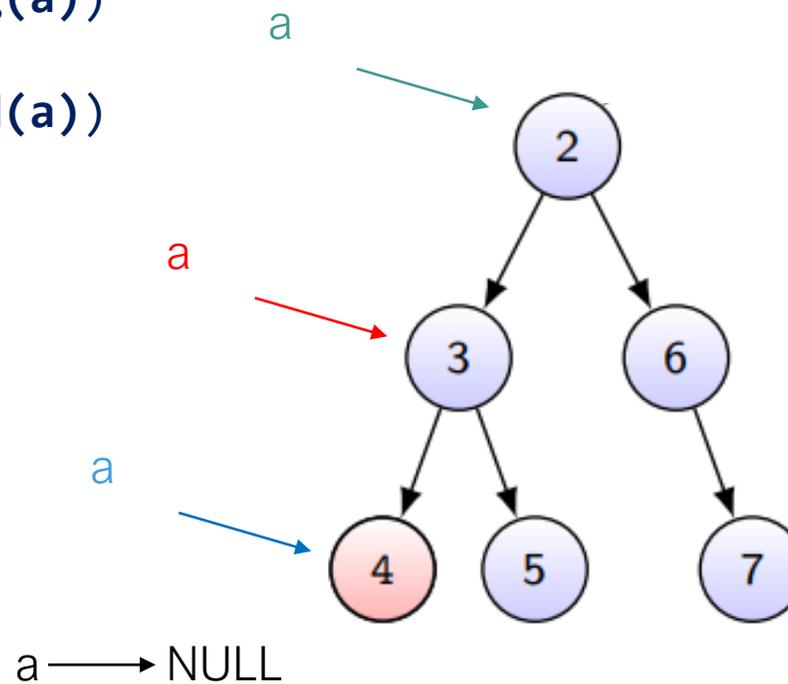
```
  SI a DIFFERENT DE NULL ALORS
```

```
    → → → parcoursInfixe(fg(a))  
           traiter(elm(a))  
           parcoursInfixe(fd(a))
```

```
  → FIN SI
```

```
FIN
```

Parcours :



# Parcours en longueur

- Algorithme :

```
PROCEDURE parcoursInfixe(a : pointeur sur Arbre)
```

```
DEBUT
```

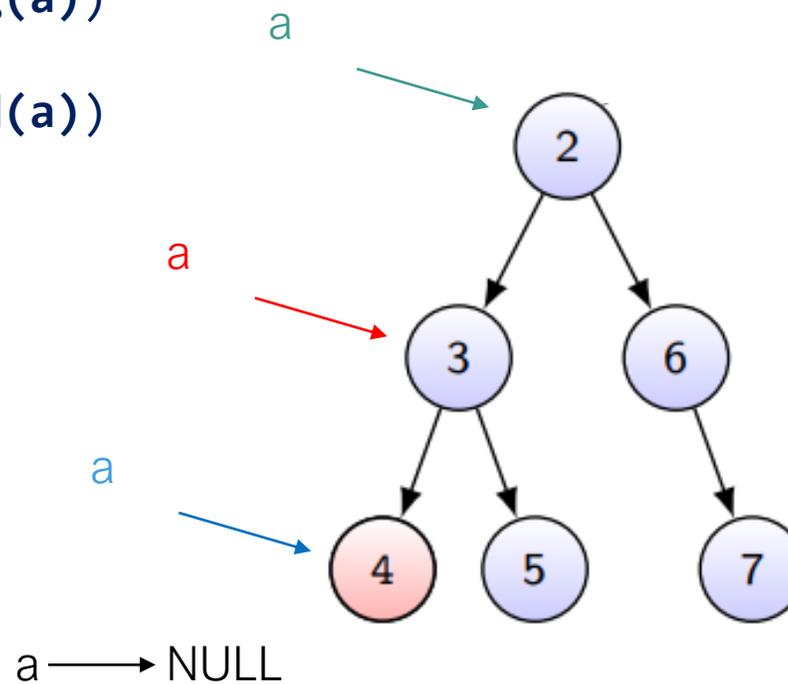
```
  SI a DIFFERENT DE NULL ALORS
```

```
    → → → parcoursInfixe(fg(a))  
          traiter(elm(a))  
          parcoursInfixe(fd(a))
```

```
  FIN SI
```

```
→ FIN
```

Parcours :



# Parcours en longueur

- Algorithme :

```
PROCEDURE parcoursInfixe(a : pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a DIFFERENT DE NULL ALORS
```

```
    → parcoursInfixe(fg(a))
```

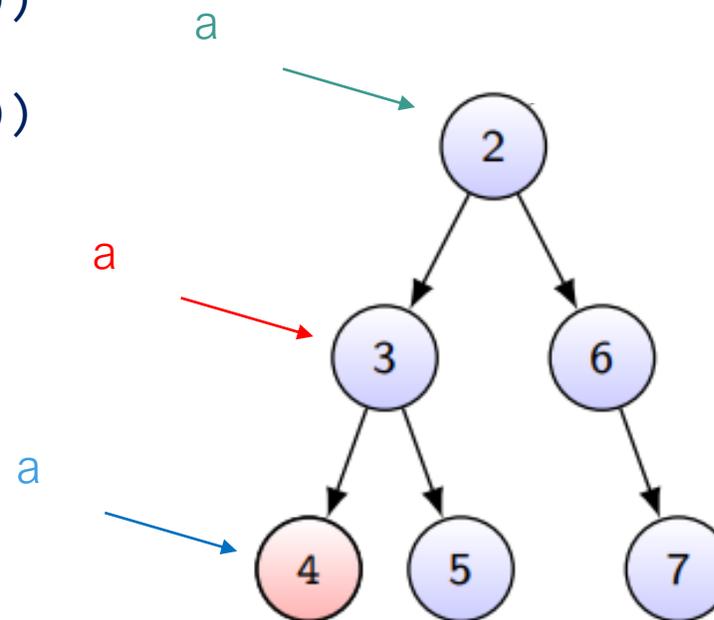
```
    → traiter(elm(a))
```

```
    → parcoursInfixe(fd(a))
```

```
  FIN SI
```

```
FIN
```

Parcours :



# Parcours en longueur

- Algorithme :

```
PROCEDURE parcoursInfixe(a : pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a DIFFERENT DE NULL ALORS
```

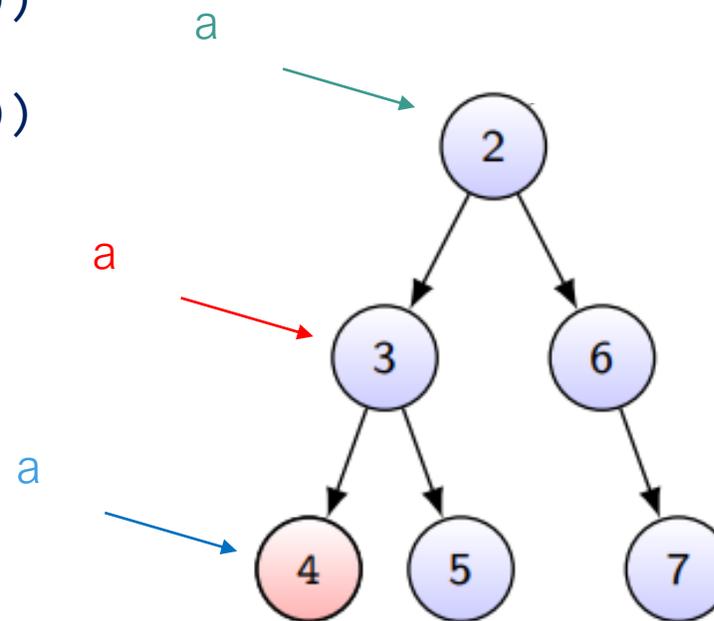
```
    → parcoursInfixe(fg(a))  
    traiter(elm(a))
```

```
    → parcoursInfixe(fd(a))
```

```
  FIN SI
```

```
FIN
```

Parcours : 4



# Parcours en longueur

- Algorithme :

PROCEDURE parcoursInfixe(a : pointeur sur Arbre)

➔ DEBUT

SI a DIFFERENT DE NULL ALORS

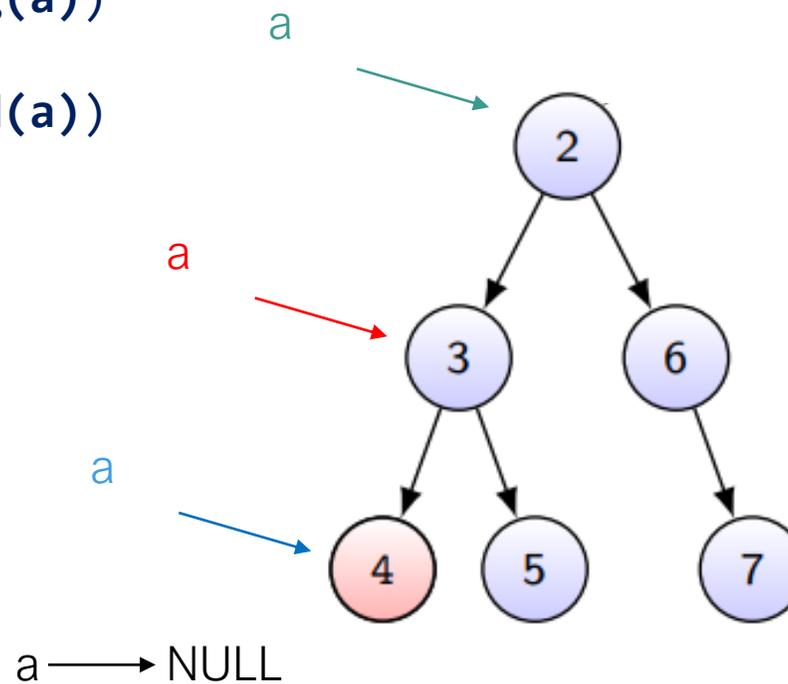
➔➔ parcoursInfixe(fg(a))  
traiter(elm(a))

➔➔➔ parcoursInfixe(fd(a))

FIN SI

FIN

Parcours : 4



# Parcours en longueur

- Algorithme :

```
PROCEDURE parcoursInfixe(a : pointeur sur Arbre)
```

```
DEBUT
```

```
  ──▶ SI a DIFFERENT DE NULL ALORS
```

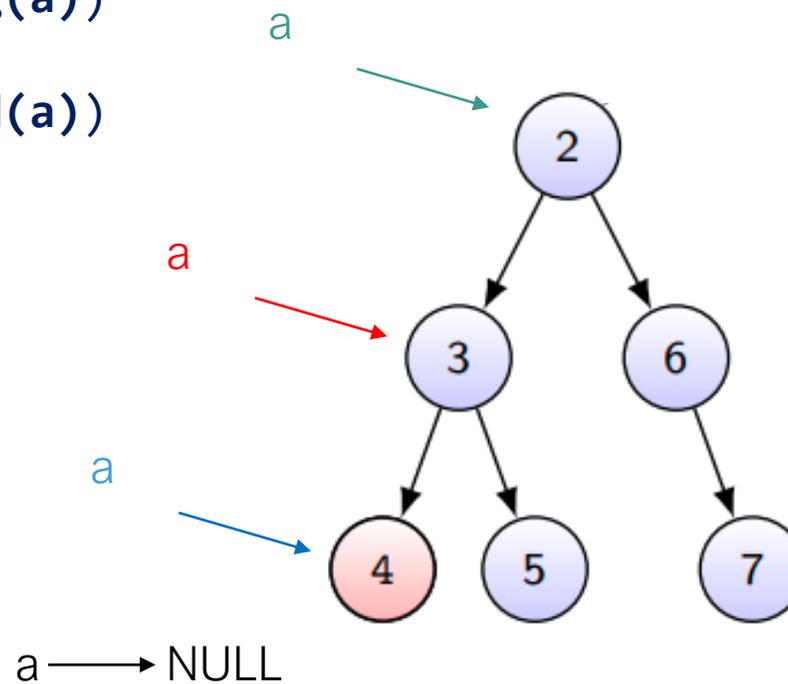
```
    ──▶ parcoursInfixe(fg(a))  
       traiter(elmt(a))
```

```
    ──▶ parcoursInfixe(fd(a))
```

```
  FIN SI
```

```
FIN
```

Parcours : 4



# Parcours en longueur

- Algorithme :

```
PROCEDURE parcoursInfixe(a : pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a DIFFERENT DE NULL ALORS
```

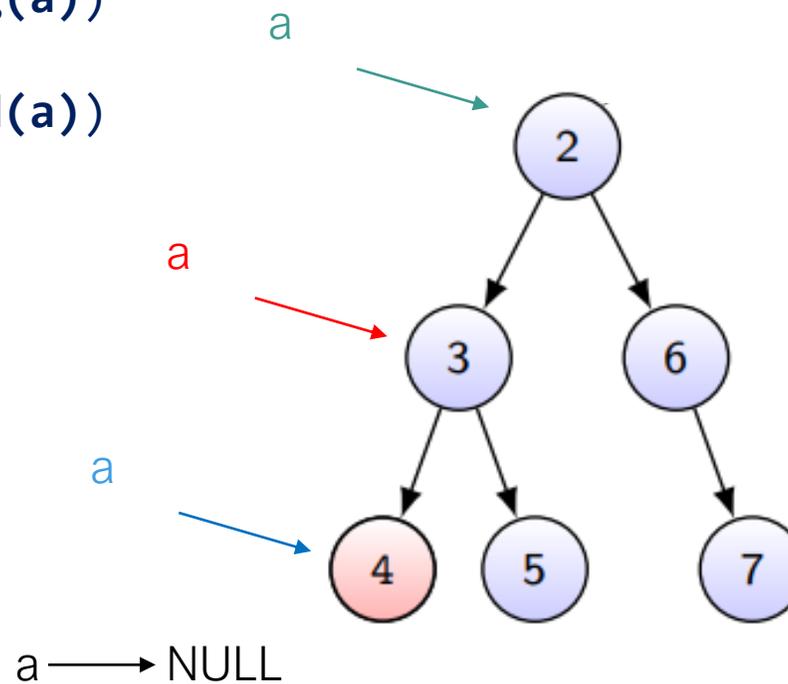
```
    → parcoursInfixe(fg(a))  
    traiter(elm(a))
```

```
    → parcoursInfixe(fd(a))
```

```
  → FIN SI
```

```
FIN
```

Parcours : 4



# Parcours en longueur

- Algorithme :

```
PROCEDURE parcoursInfixe(a : pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a DIFFERENT DE NULL ALORS
```

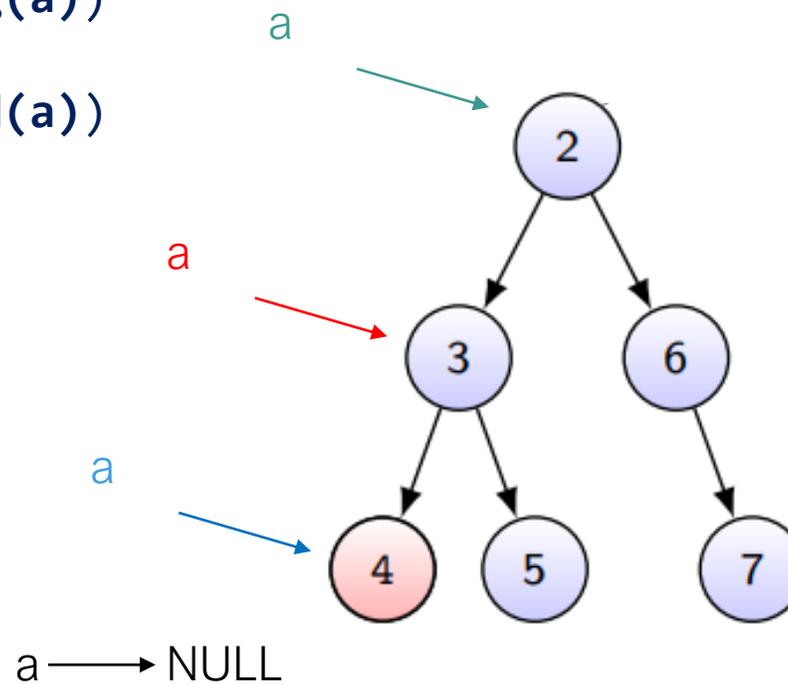
```
    → parcoursInfixe(fg(a))  
    traiter(elm(a))
```

```
    → parcoursInfixe(fd(a))
```

```
  FIN SI
```

```
→ FIN
```

Parcours : 4



# Parcours en longueur

- Algorithme :

```
PROCEDURE parcoursInfixe(a : pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a DIFFERENT DE NULL ALORS
```

```
    → parcoursInfixe(fg(a))
```

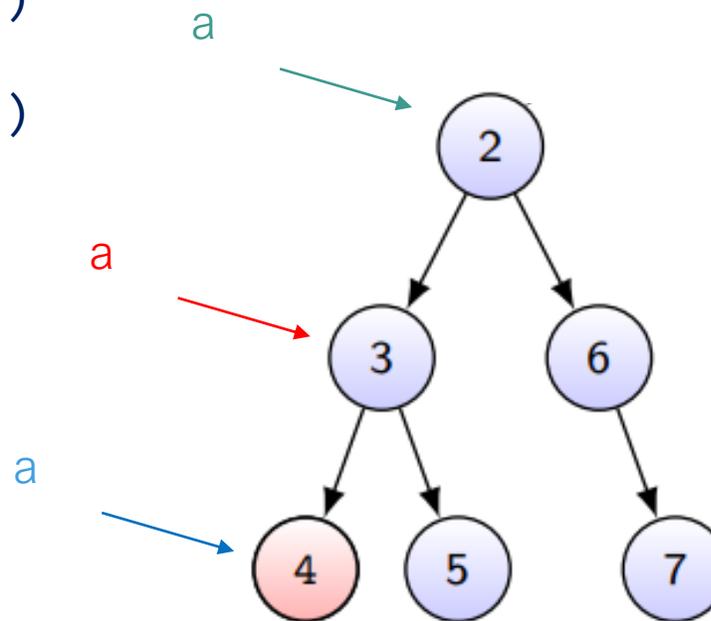
```
    traiter(elm(a))
```

```
    parcoursInfixe(fd(a))
```

```
  → FIN SI
```

```
FIN
```

Parcours : 4



# Parcours en longueur

- Algorithme :

```
PROCEDURE parcoursInfixe(a : pointeur sur Arbre)
```

```
DEBUT
```

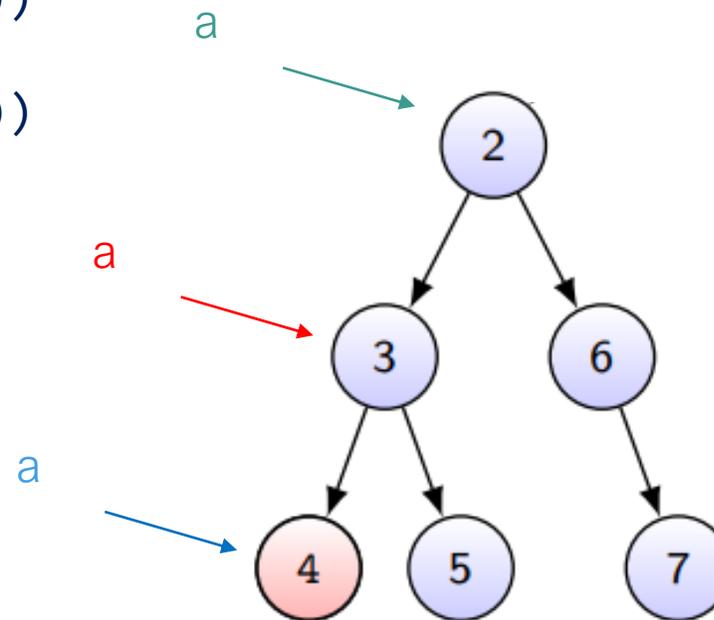
```
  SI a DIFFERENT DE NULL ALORS
```

```
    → parcoursInfixe(fg(a))  
    traiter(elm(a))  
    → parcoursInfixe(fd(a))
```

```
  FIN SI
```

```
→ FIN
```

Parcours : 4



# Parcours en longueur

- Algorithme :

```
PROCEDURE parcoursInfixe(a : pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a DIFFERENT DE NULL ALORS
```

```
    → parcoursInfixe(fg(a))
```

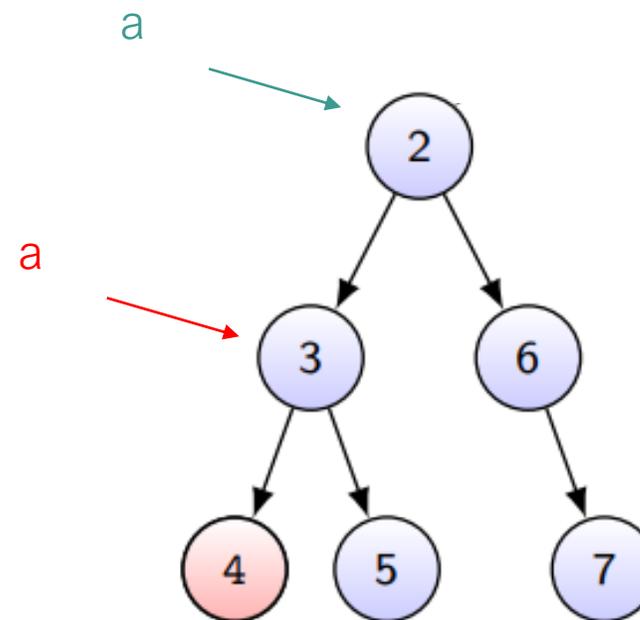
```
    → traiter(elmt(a))
```

```
    parcoursInfixe(fd(a))
```

```
  FIN SI
```

```
FIN
```

Parcours : 4



# Parcours en longueur

- Algorithme :

```
PROCEDURE parcoursInfixe(a : pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a DIFFERENT DE NULL ALORS
```

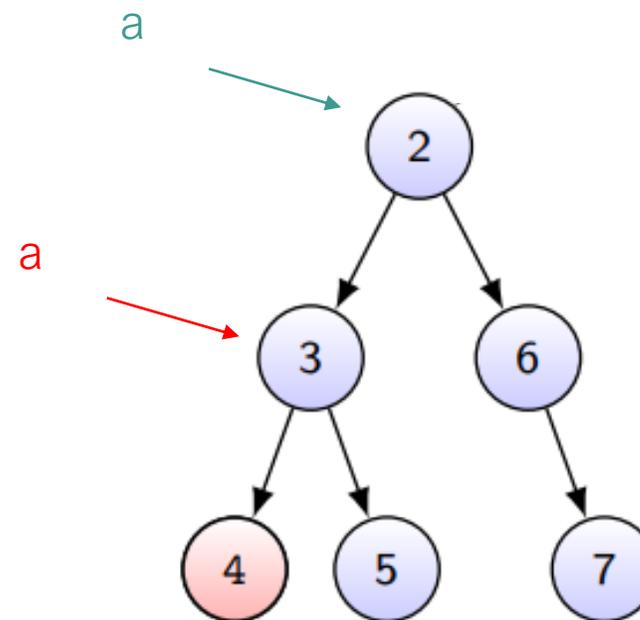
```
    → parcoursInfixe(fg(a))  
    traiter(elm(a))
```

```
    → parcoursInfixe(fd(a))
```

```
  FIN SI
```

```
FIN
```

Parcours : 4, 3



# Parcours en longueur

- Algorithme :

PROCEDURE parcoursInfixe(a : pointeur sur Arbre)

➔ DEBUT

SI a DIFFERENT DE NULL ALORS

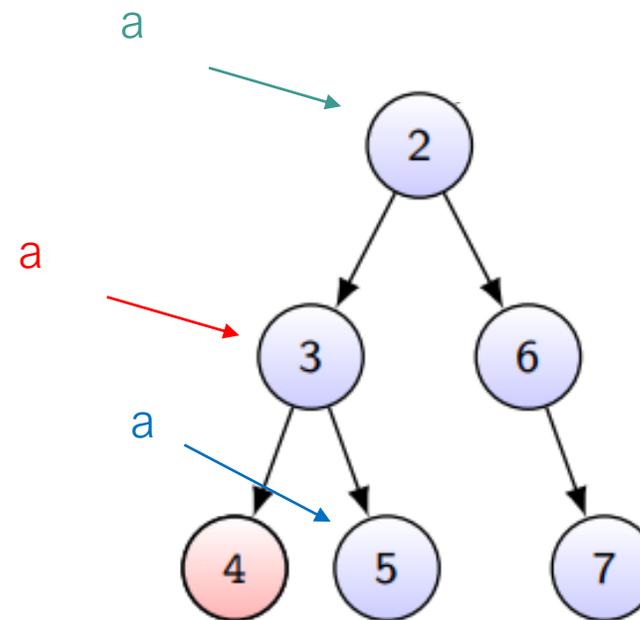
➔ parcoursInfixe(fg(a))  
traiter(elm(a))

➔ parcoursInfixe(fd(a))

FIN SI

FIN

Parcours : 4, 3



# Parcours en longueur

- Algorithme :

```
PROCEDURE parcoursInfixe(a : pointeur sur Arbre)
```

```
DEBUT
```

```
  → SI a DIFFERENT DE NULL ALORS
```

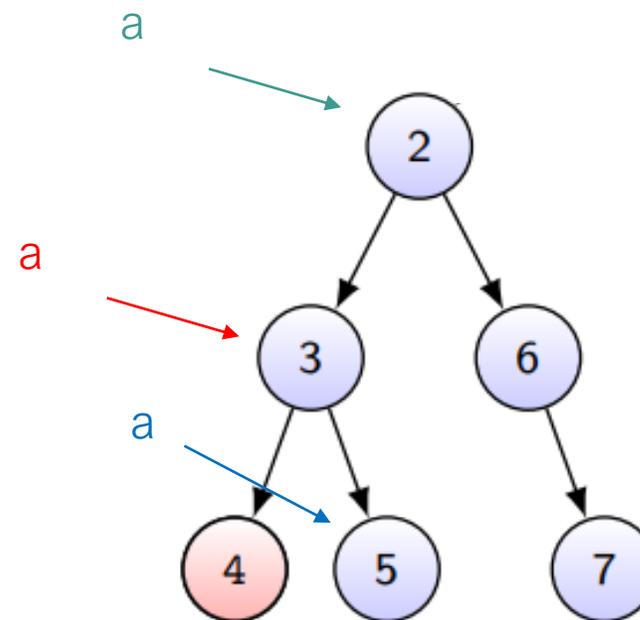
```
    → parcoursInfixe(fg(a))  
    traiter(elm(a))
```

```
    → parcoursInfixe(fd(a))
```

```
  FIN SI
```

```
FIN
```

Parcours : 4, 3



# Parcours en longueur

- Algorithme :

```
PROCEDURE parcoursInfixe(a : pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a DIFFERENT DE NULL ALORS
```

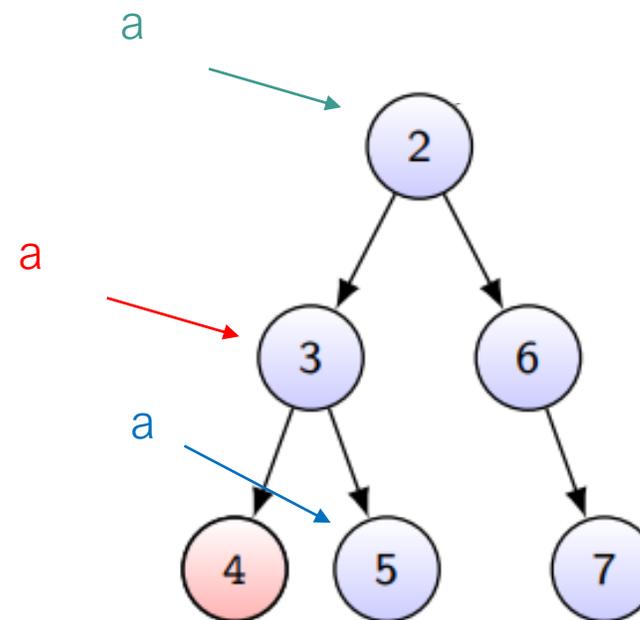
```
    → parcoursInfixe(fg(a))  
    traiter(elm(a))
```

```
    → parcoursInfixe(fd(a))
```

```
  FIN SI
```

```
FIN
```

Parcours : 4, 3



# Parcours en longueur

- Algorithme :

```
PROCEDURE parcoursInfixe(a : pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a DIFFERENT DE NULL ALORS
```

```
    → parcoursInfixe(fg(a))
```

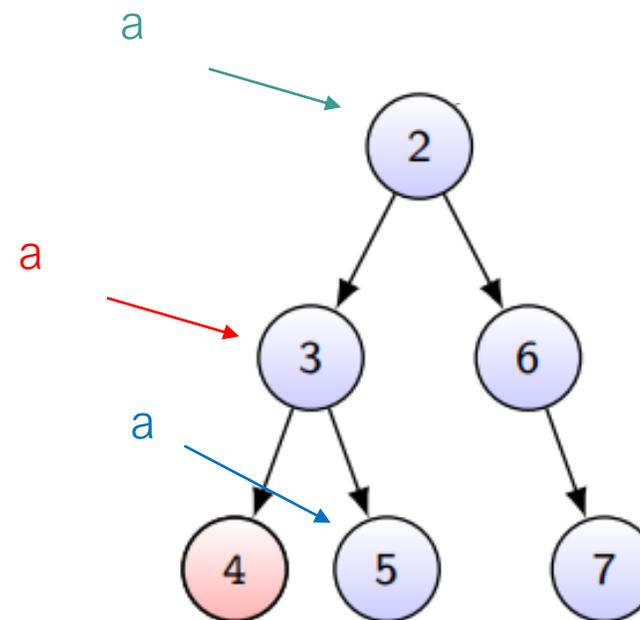
```
    → traiter(elm(a))
```

```
    → parcoursInfixe(fd(a))
```

```
  FIN SI
```

```
FIN
```

Parcours : 4, 3



# Parcours en longueur

- Algorithme :

```
PROCEDURE parcoursInfixe(a : pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a DIFFERENT DE NULL ALORS
```

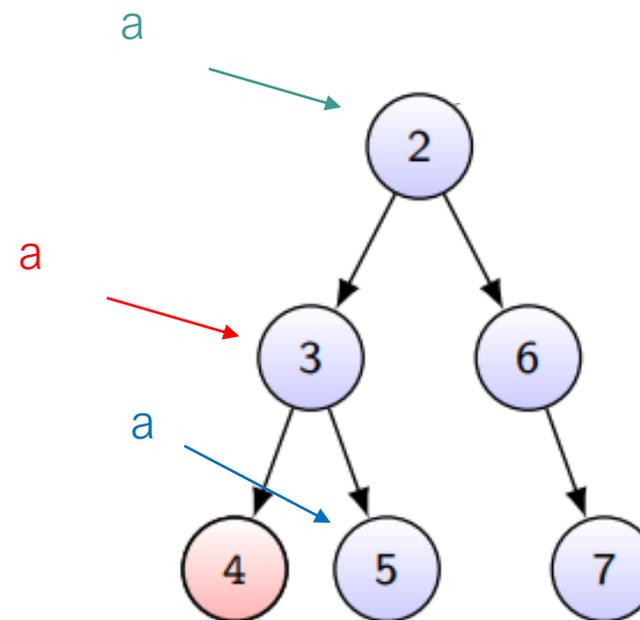
```
    → parcoursInfixe(fg(a))  
    traiter(elm(a))
```

```
    → → parcoursInfixe(fd(a))
```

```
  FIN SI
```

```
FIN
```

Parcours : 4, 3, 5



# Parcours en longueur

- Algorithme :

```
PROCEDURE parcoursInfixe(a : pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a DIFFERENT DE NULL ALORS
```

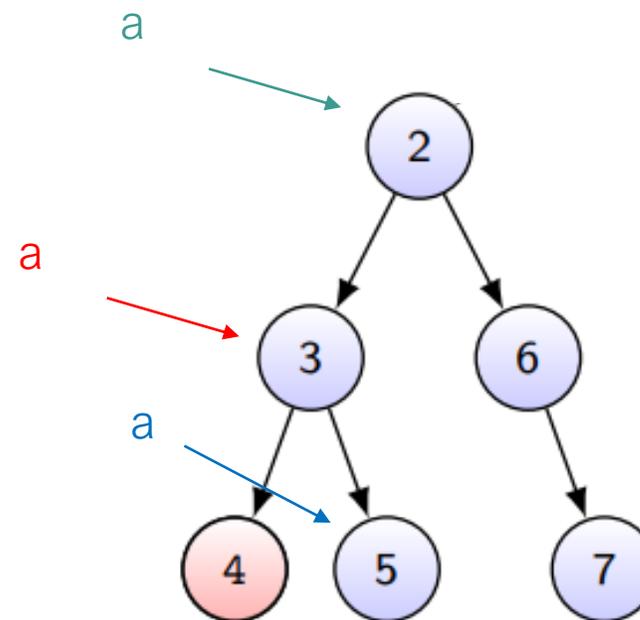
```
    → parcoursInfixe(fg(a))  
    traiter(elm(a))
```

```
    → parcoursInfixe(fd(a))
```

```
  → FIN SI
```

```
FIN
```

Parcours : 4, 3, 5



# Parcours en longueur

- Algorithme :

```
PROCEDURE parcoursInfixe(a : pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a DIFFERENT DE NULL ALORS
```

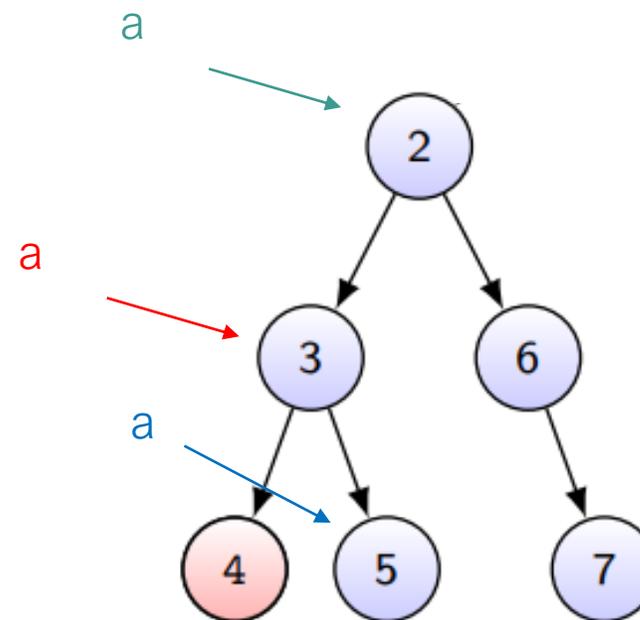
```
    → parcoursInfixe(fg(a))  
    traiter(elm(a))
```

```
    → parcoursInfixe(fd(a))
```

```
  FIN SI
```

```
→ FIN
```

Parcours : 4, 3, 5



# Parcours en longueur

- Algorithme :

```
PROCEDURE parcoursInfixe(a : pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a DIFFERENT DE NULL ALORS
```

```
    ➔ parcoursInfixe(fg(a))
```

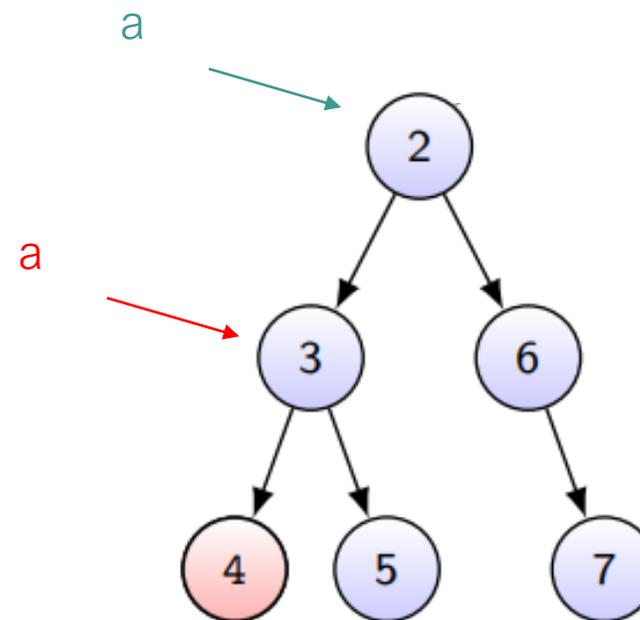
```
    traiter(elm(a))
```

```
    parcoursInfixe(fd(a))
```

```
  ➔ FIN SI
```

```
FIN
```

Parcours : 4, 3, 5



# Parcours en longueur

- Algorithme :

```
PROCEDURE parcoursInfixe(a : pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a DIFFERENT DE NULL ALORS
```

```
    → parcoursInfixe(fg(a))
```

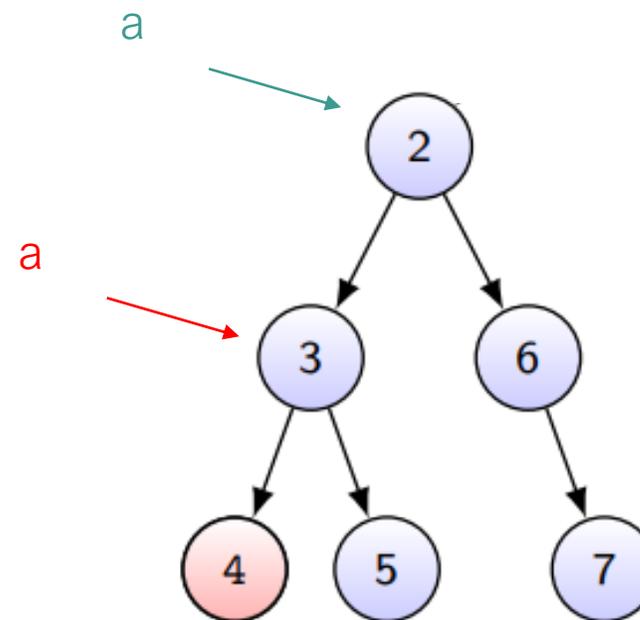
```
    traiter(elm(a))
```

```
    parcoursInfixe(fd(a))
```

```
  FIN SI
```

```
→ FIN
```

Parcours : 4, 3, 5



# Parcours en longueur

- Algorithme :

```
PROCEDURE parcoursInfixe(a : pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a DIFFERENT DE NULL ALORS
```

```
    parcoursInfixe(fg(a))
```

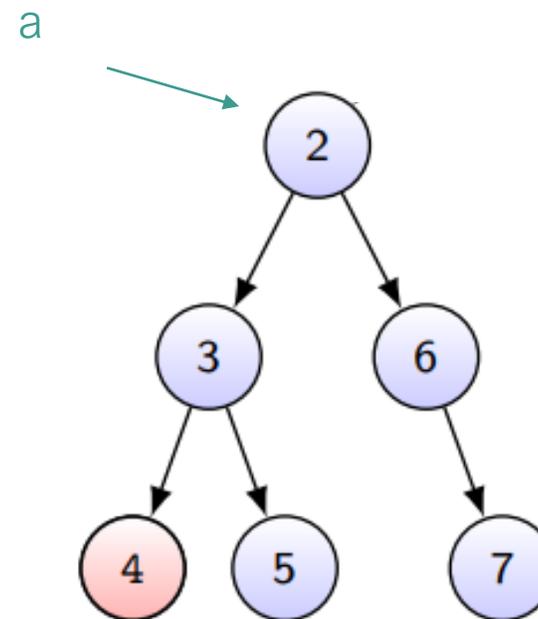
```
    → traiter(elm(a))
```

```
    parcoursInfixe(fd(a))
```

```
  FIN SI
```

```
FIN
```

Parcours : 4, 3, 5



# Parcours en longueur

- Algorithme :

```
PROCEDURE parcoursInfixe(a : pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a DIFFERENT DE NULL ALORS
```

```
    parcoursInfixe(fg(a))
```

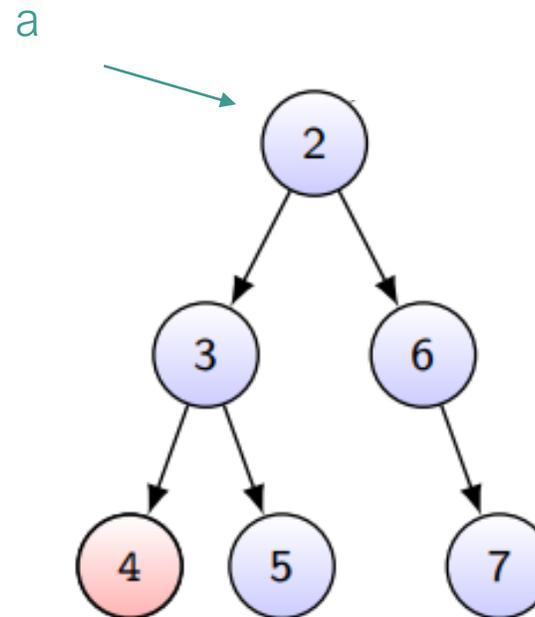
```
    traiter(elm(a))
```

```
    → parcoursInfixe(fd(a))
```

```
  FIN SI
```

```
FIN
```

Parcours : 4, 3, 5, 2



# Parcours en longueur

- Algorithme :

```
PROCEDURE parcoursInfixe(a : pointeur sur Arbre)
```

```
→ DEBUT
```

```
    SI a DIFFERENT DE NULL ALORS
```

```
        parcoursInfixe(fg(a))
```

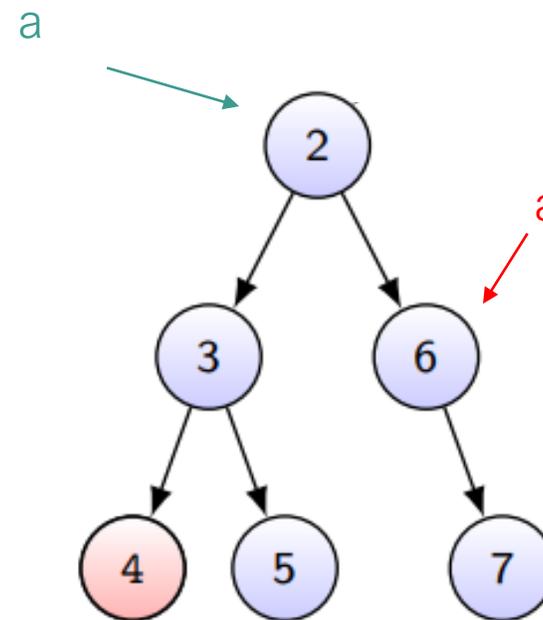
```
        traiter(elm(a))
```

```
        → parcoursInfixe(fd(a))
```

```
    FIN SI
```

```
FIN
```

Parcours : 4, 3, 5, 2



# Parcours en longueur

- Algorithme :

```
PROCEDURE parcoursInfixe(a : pointeur sur Arbre)
```

```
DEBUT
```

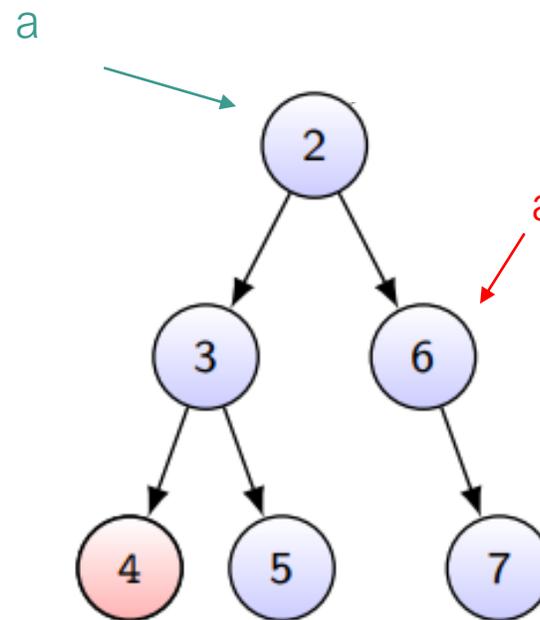
```
  → SI a DIFFERENT DE NULL ALORS  
    parcoursInfixe(fg(a))  
    traiter(elm(a))
```

```
  → parcoursInfixe(fd(a))
```

```
FIN SI
```

```
FIN
```

Parcours : 4, 3, 5, 2



# Parcours en longueur

- Algorithme :

```
PROCEDURE parcoursInfixe(a : pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a DIFFERENT DE NULL ALORS
```

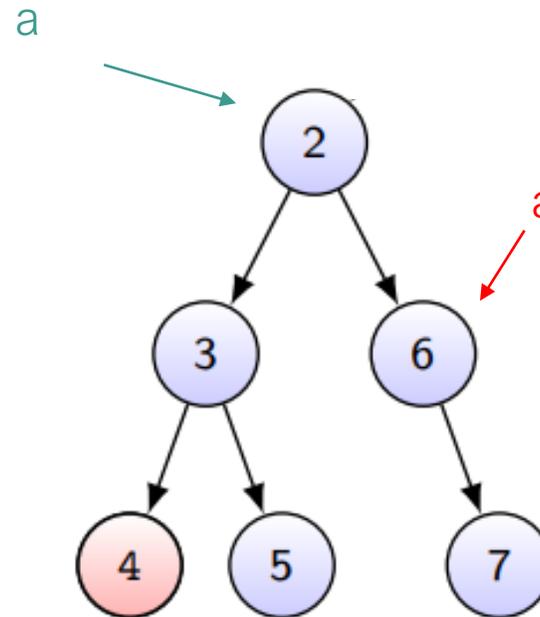
```
    → parcoursInfixe(fg(a))  
    traiter(elm(a))
```

```
    → parcoursInfixe(fd(a))
```

```
  FIN SI
```

```
FIN
```

Parcours : 4, 3, 5, 2



# Parcours en longueur

- Algorithme :

PROCEDURE parcoursInfixe(a : pointeur sur Arbre)

➔ DEBUT

SI a DIFFERENT DE NULL ALORS

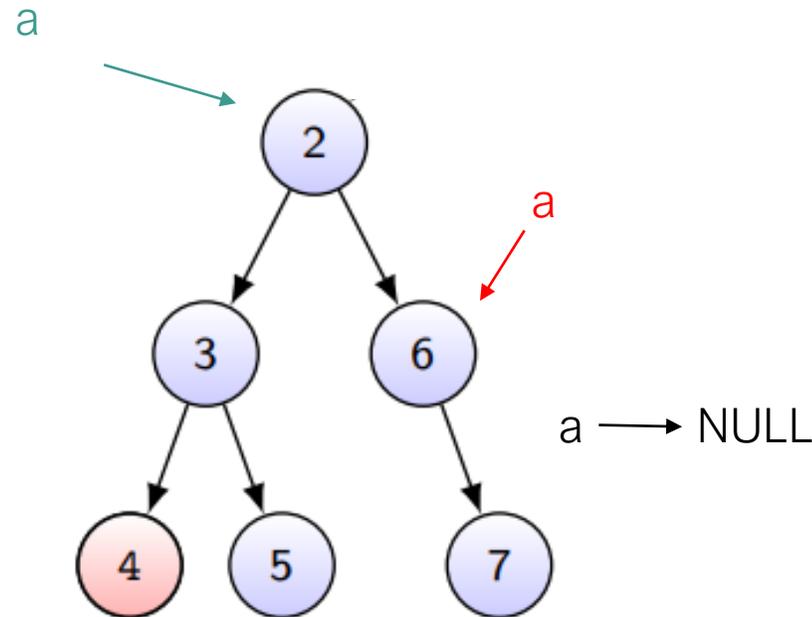
➔ parcoursInfixe(fg(a))  
traiter(elm(a))

➔ parcoursInfixe(fd(a))

FIN SI

FIN

Parcours : 4, 3, 5, 2



# Parcours en longueur

- Algorithme :

```
PROCEDURE parcoursInfixe(a : pointeur sur Arbre)
```

```
DEBUT
```

```
  ──▶ SI a DIFFERENT DE NULL ALORS
```

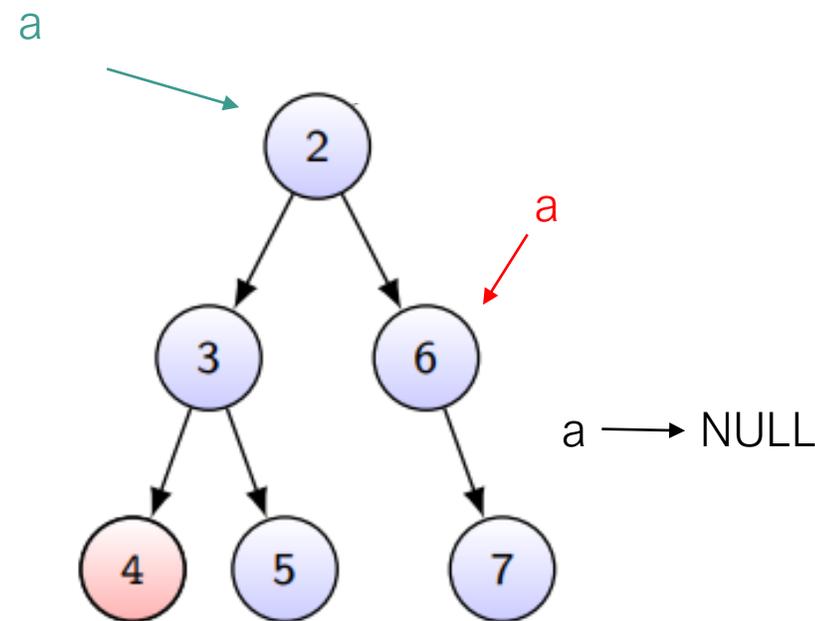
```
    ──▶ parcoursInfixe(fg(a))  
       traiter(elm(a))
```

```
    ──▶ parcoursInfixe(fd(a))
```

```
  FIN SI
```

```
FIN
```

Parcours : 4, 3, 5, 2



# Parcours en longueur

- Algorithme :

```
PROCEDURE parcoursInfixe(a : pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a DIFFERENT DE NULL ALORS
```

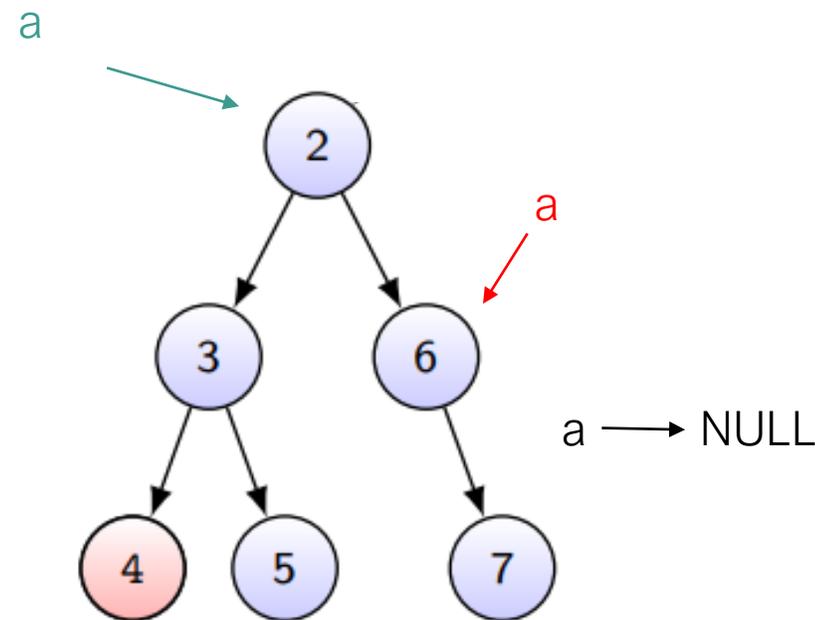
```
    → parcoursInfixe(fg(a))  
    traiter(elm(a))
```

```
    → parcoursInfixe(fd(a))
```

```
  → FIN SI
```

```
FIN
```

Parcours : 4, 3, 5, 2



# Parcours en longueur

- Algorithme :

```
PROCEDURE parcoursInfixe(a : pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a DIFFERENT DE NULL ALORS
```

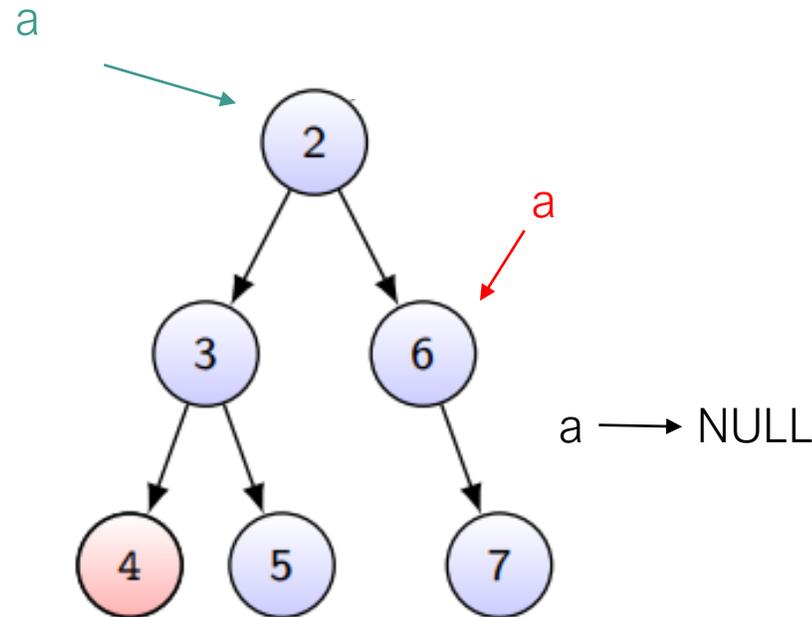
```
    → parcoursInfixe(fg(a))  
    traiter(elm(a))
```

```
    → parcoursInfixe(fd(a))
```

```
  FIN SI
```

```
→ FIN
```

Parcours : 4, 3, 5, 2



# Parcours en longueur

- Algorithme :

```
PROCEDURE parcoursInfixe(a : pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a DIFFERENT DE NULL ALORS
```

```
    parcoursInfixe(fg(a))
```

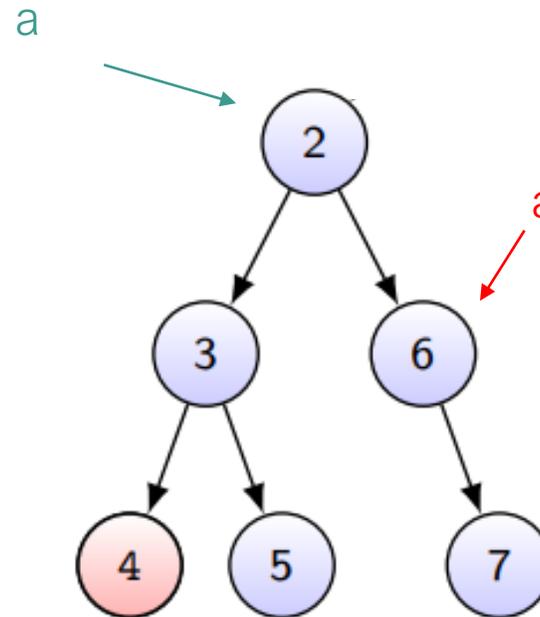
```
    → traiter(elm(a))
```

```
    → parcoursInfixe(fd(a))
```

```
  FIN SI
```

```
FIN
```

Parcours : 4, 3, 5, 2





# Parcours en longueur

- Algorithme :

PROCEDURE parcoursInfixe(a : pointeur sur Arbre)

➔ DEBUT

SI a DIFFERENT DE NULL ALORS

parcoursInfixe(fg(a))

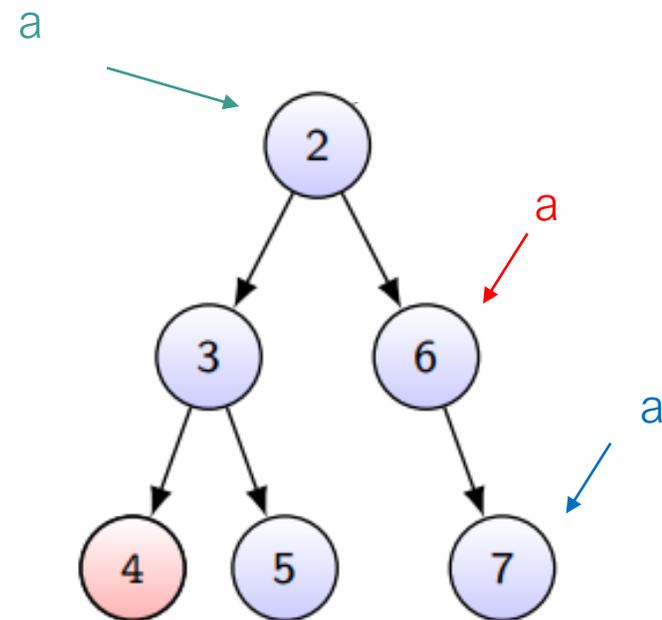
traiter(elm(a))

➔ ➔ parcoursInfixe(fd(a))

FIN SI

FIN

Parcours : 4, 3, 5, 2, 6



# Parcours en longueur

- Algorithme :

```
PROCEDURE parcoursInfixe(a : pointeur sur Arbre)
```

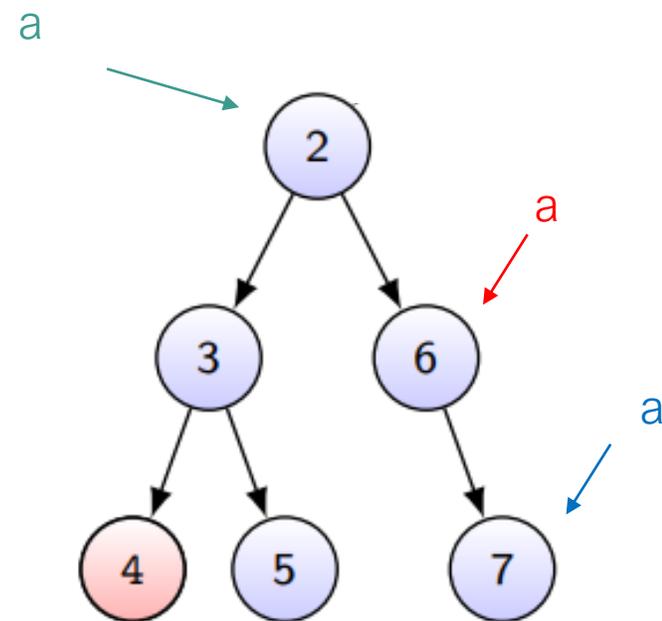
```
DEBUT
```

```
  → SI a DIFFERENT DE NULL ALORS  
    parcoursInfixe(fg(a))  
    traiter(elm(a))
```

```
  → → parcoursInfixe(fd(a))  
FIN SI
```

```
FIN
```

Parcours : 4, 3, 5, 2, 6



# Parcours en longueur

- Algorithme :

```
PROCEDURE parcoursInfixe(a : pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a DIFFERENT DE NULL ALORS
```

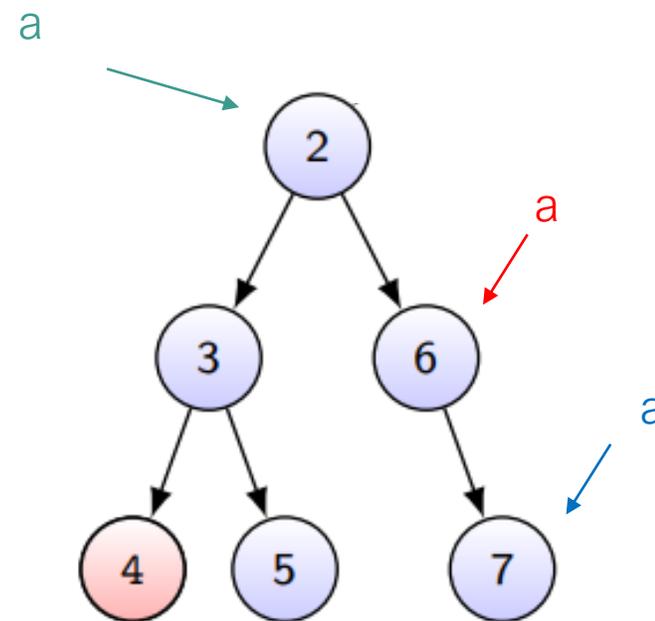
```
    ➡ parcoursInfixe(fg(a))  
    traiter(elm(a))
```

```
    ➡➡ parcoursInfixe(fd(a))
```

```
  FIN SI
```

```
FIN
```

Parcours : 4, 3, 5, 2, 6



# Parcours en longueur

- Algorithme :

```
PROCEDURE parcoursInfixe(a : pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a DIFFERENT DE NULL ALORS
```

```
    parcoursInfixe(fg(a))
```

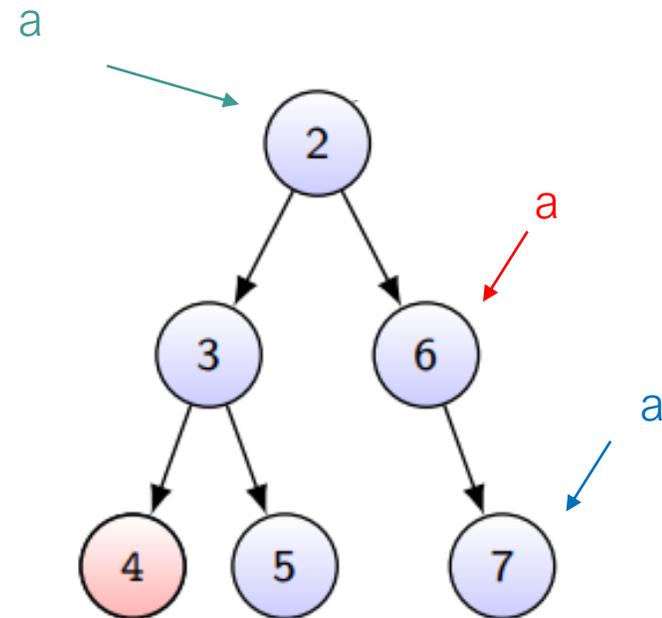
```
    ➡ traiter(elm(a))
```

```
    ➡➡ parcoursInfixe(fd(a))
```

```
  FIN SI
```

```
FIN
```

Parcours : 4, 3, 5, 2, 6



# Parcours en longueur

- Algorithme :

```
PROCEDURE parcoursInfixe(a : pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a DIFFERENT DE NULL ALORS
```

```
    parcoursInfixe(fg(a))
```

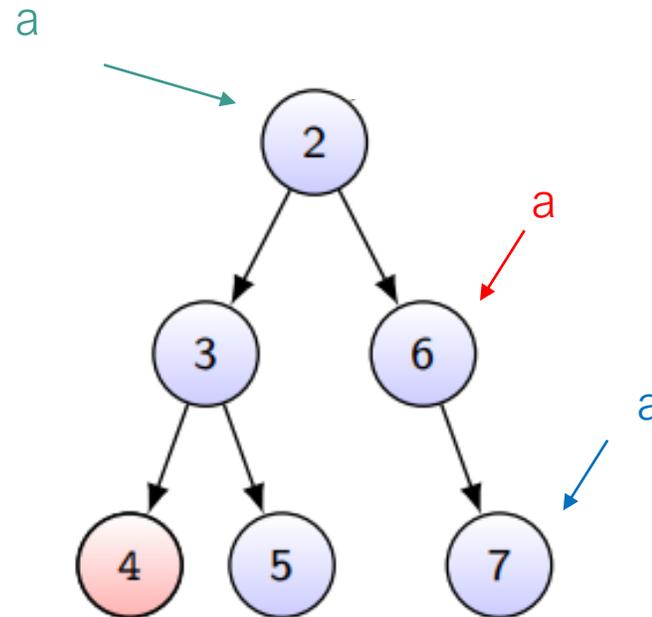
```
    traiter(elm(a))
```

```
    → → → parcoursInfixe(fd(a))
```

```
  FIN SI
```

```
FIN
```

Parcours : 4, 3, 5, 2, 6, 7



# Parcours en longueur

- Algorithme :

```
PROCEDURE parcoursInfixe(a : pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a DIFFERENT DE NULL ALORS
```

```
    parcoursInfixe(fg(a))
```

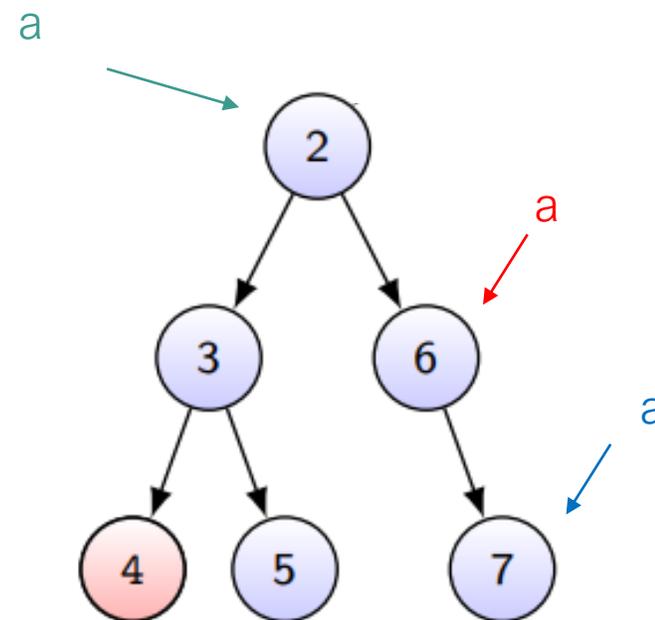
```
    traiter(elm(a))
```

```
      parcoursInfixe(fd(a))
```

```
   FIN SI
```

```
FIN
```

Parcours : 4, 3, 5, 2, 6, 7



# Parcours en longueur

- Algorithme :

```
PROCEDURE parcoursInfixe(a : pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a DIFFERENT DE NULL ALORS
```

```
    parcoursInfixe(fg(a))
```

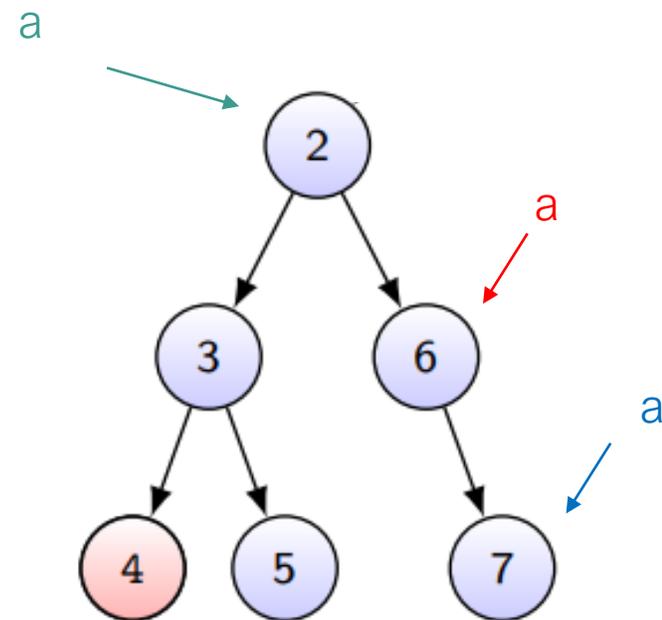
```
    traiter(elm(a))
```

```
      parcoursInfixe(fd(a))
```

```
  FIN SI
```

```
 FIN
```

Parcours : 4, 3, 5, 2, 6, 7



# Parcours en longueur

- Algorithme :

```
PROCEDURE parcoursInfixe(a : pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a DIFFERENT DE NULL ALORS
```

```
    parcoursInfixe(fg(a))
```

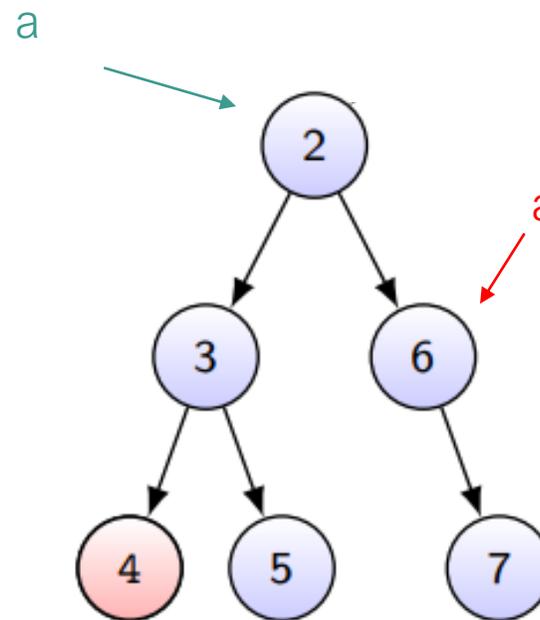
```
    traiter(elm(a))
```

```
    → parcoursInfixe(fd(a))
```

```
  → FIN SI
```

```
FIN
```

Parcours : 4, 3, 5, 2, 6, 7



# Parcours en longueur

- Algorithme :

```
PROCEDURE parcoursInfixe(a : pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a DIFFERENT DE NULL ALORS
```

```
    parcoursInfixe(fg(a))
```

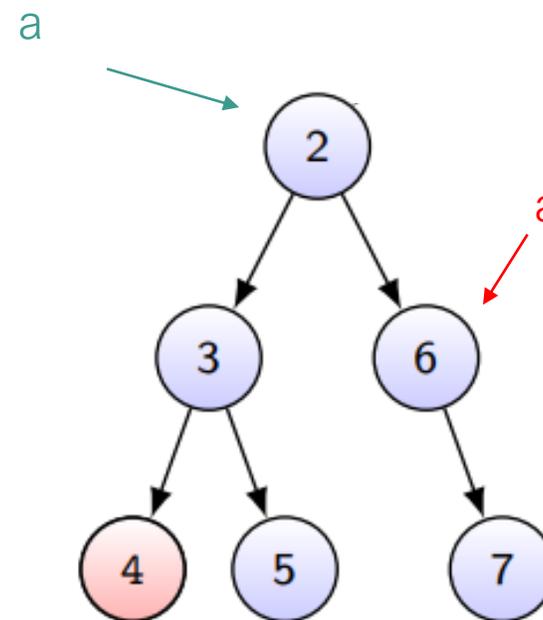
```
    traiter(elm(a))
```

```
    → parcoursInfixe(fd(a))
```

```
  FIN SI
```

```
→ FIN
```

Parcours : 4, 3, 5, 2, 6, 7



# Parcours en longueur

- Algorithme :

```
PROCEDURE parcoursInfixe(a : pointeur sur Arbre)
```

```
DEBUT
```

```
    SI a DIFFERENT DE NULL ALORS
```

```
        parcoursInfixe(fg(a))
```

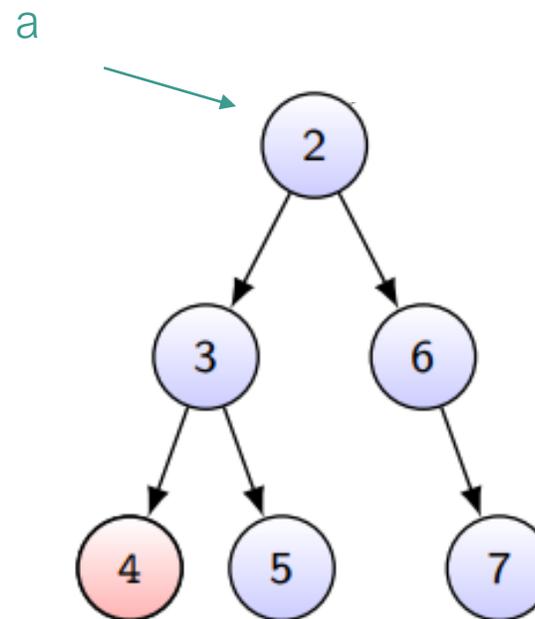
```
        traiter(elm(a))
```

```
        parcoursInfixe(fd(a))
```

```
    → FIN SI
```

```
FIN
```

Parcours : 4, 3, 5, 2, 6, 7



# Parcours en longueur

- Algorithme :

```
PROCEDURE parcoursInfixe(a : pointeur sur Arbre)
```

```
DEBUT
```

```
  SI a DIFFERENT DE NULL ALORS
```

```
    parcoursInfixe(fg(a))
```

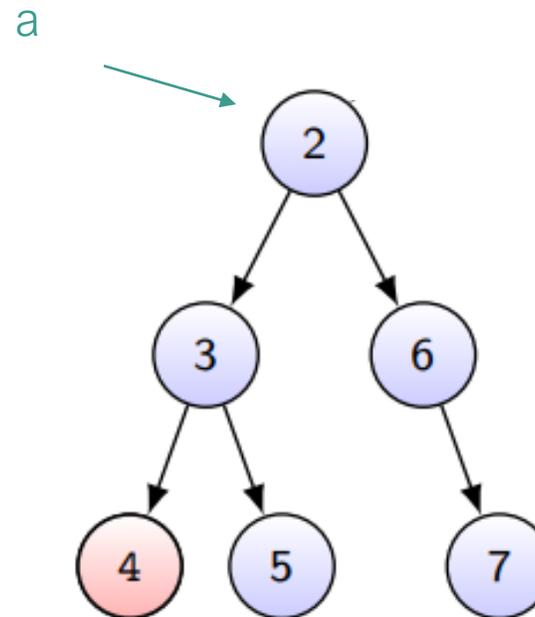
```
    traiter(elm(a))
```

```
    parcoursInfixe(fd(a))
```

```
  FIN SI
```

```
→ FIN
```

Parcours : 4, 3, 5, 2, 6, 7

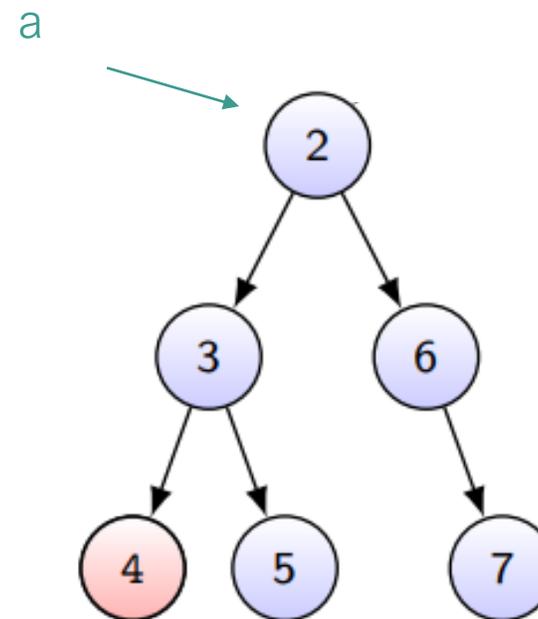


# Parcours en longueur

- Algorithme :

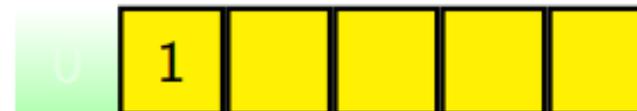
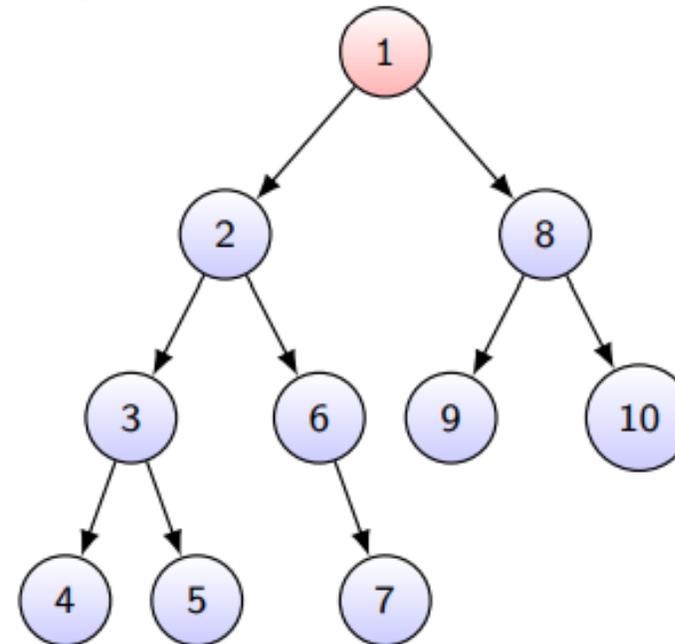
```
PROCEDURE parcoursInfixe(a : pointeur sur Arbre)
DEBUT
    SI a DIFFERENT DE NULL ALORS
        parcoursInfixe(fg(a))
        traiter(elm(a))
        parcoursInfixe(fd(a))
    FIN SI
FIN
```

Parcours : 4, 3, 5, 2, 6, 7



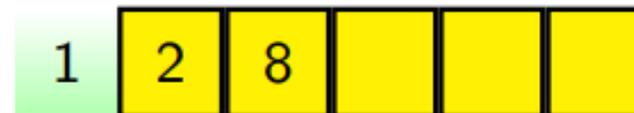
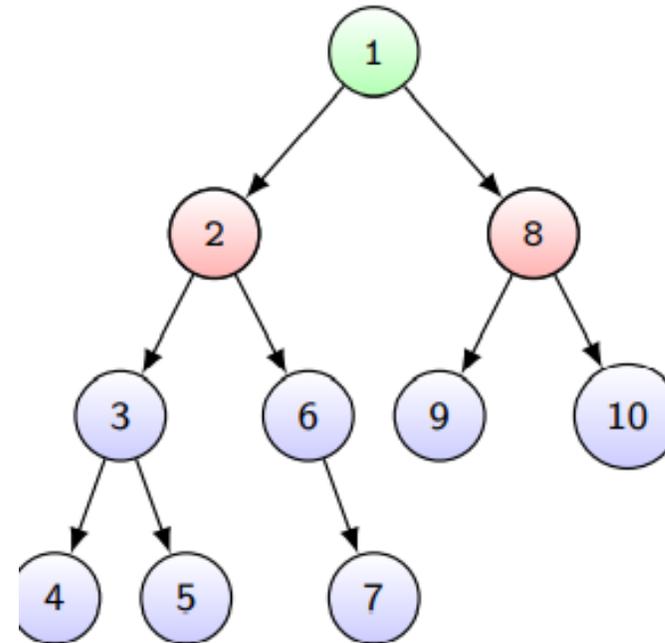
# Parcours en largeur

- **Parcours en largeur** : On explore les nœuds un par un sur un même niveau. On passe ensuite sur le niveau suivant, et ainsi de suite (on va vers les frères avant de parcourir les fils).
- Le parcours en largeur se réalise à l'aide d'une file de la façon suivante :
  - Mettre la racine dans la file
  - Tant que la file n'est pas vide :
    1. Récupérer en tête de file le nœud à traiter
    2. Traiter le nœud
    3. Ajouter ses fils dans la file
- Parcours:



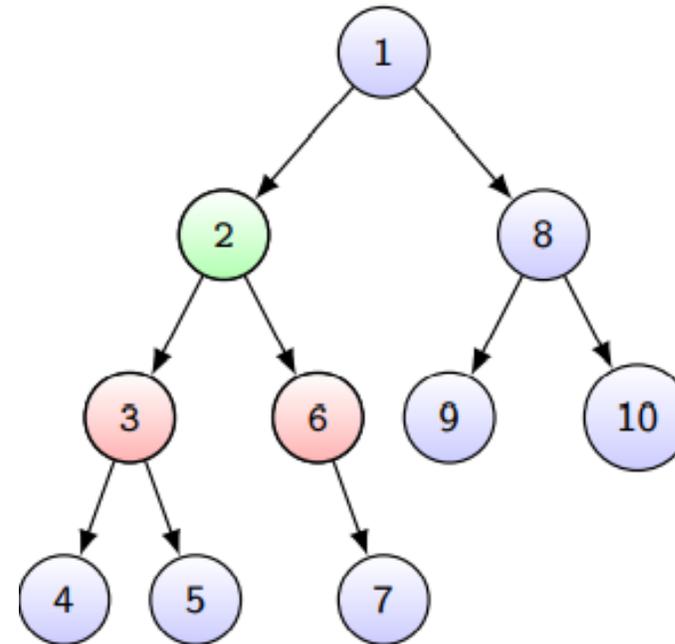
# Parcours en largeur

- **Parcours en largeur** : On explore les nœuds un par un sur un même niveau. On passe ensuite sur le niveau suivant, et ainsi de suite (on va vers les frères avant de parcourir les fils).
- Le parcours en largeur se réalise à l'aide d'une file de la façon suivante :
  - Mettre la racine dans la file
  - Tant que la file n'est pas vide :
    1. Récupérer en tête de file le nœud à traiter
    2. Traiter le nœud
    3. Ajouter ses fils dans la file
- Parcours: 1,



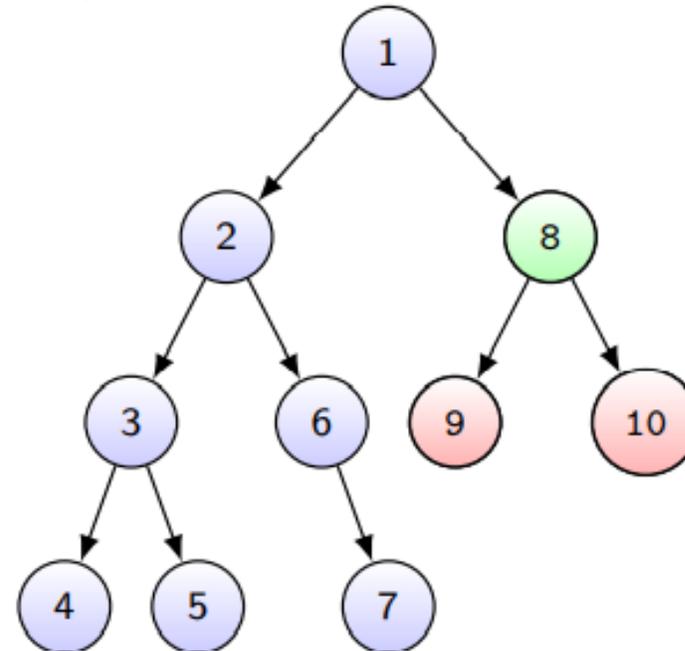
# Parcours en largeur

- **Parcours en largeur** : On explore les nœuds un par un sur un même niveau. On passe ensuite sur le niveau suivant, et ainsi de suite (on va vers les frères avant de parcourir les fils).
- Le parcours en largeur se réalise à l'aide d'une file de la façon suivante :
  - Mettre la racine dans la file
  - Tant que la file n'est pas vide :
    1. Récupérer en tête de file le nœud à traiter
    2. Traiter le nœud
    3. Ajouter ses fils dans la file
- Parcours: 1, 2,



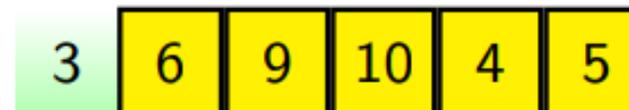
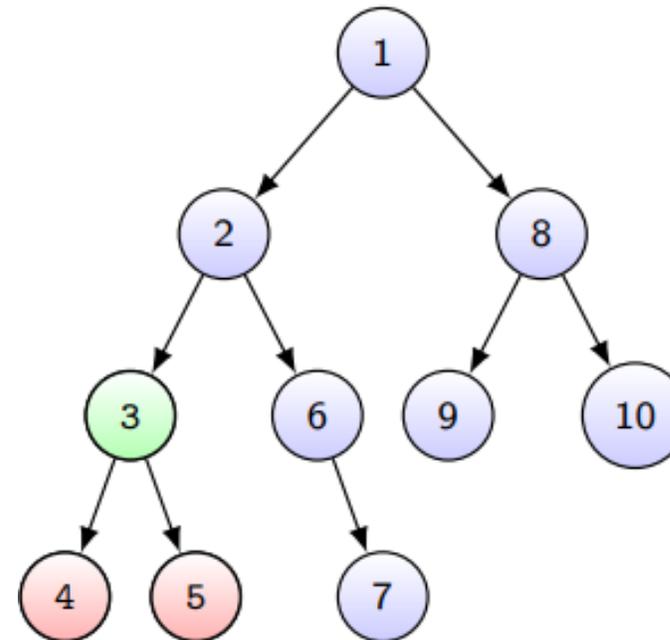
# Parcours en largeur

- **Parcours en largeur** : On explore les nœuds un par un sur un même niveau. On passe ensuite sur le niveau suivant, et ainsi de suite (on va vers les frères avant de parcourir les fils).
- Le parcours en largeur se réalise à l'aide d'une file de la façon suivante :
  - Mettre la racine dans la file
  - Tant que la file n'est pas vide :
    1. Récupérer en tête de file le nœud à traiter
    2. Traiter le nœud
    3. Ajouter ses fils dans la file
- Parcours: 1, 2, 8



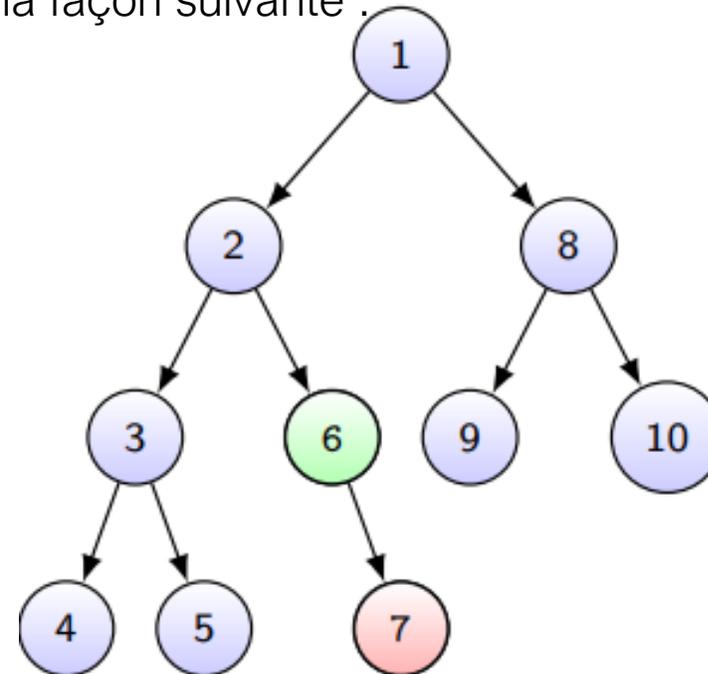
# Parcours en largeur

- **Parcours en largeur** : On explore les nœuds un par un sur un même niveau. On passe ensuite sur le niveau suivant, et ainsi de suite (on va vers les frères avant de parcourir les fils).
- Le parcours en largeur se réalise à l'aide d'une file de la façon suivante :
  - Mettre la racine dans la file
  - Tant que la file n'est pas vide :
    1. Récupérer en tête de file le nœud à traiter
    2. Traiter le nœud
    3. Ajouter ses fils dans la file
- Parcours: 1, 2, 8, 3



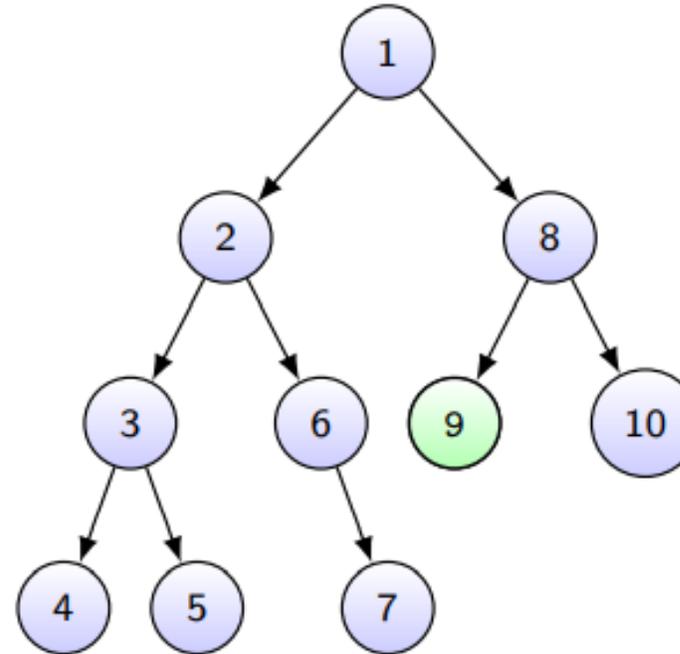
# Parcours en largeur

- **Parcours en largeur** : On explore les nœuds un par un sur un même niveau. On passe ensuite sur le niveau suivant, et ainsi de suite (on va vers les frères avant de parcourir les fils).
- Le parcours en largeur se réalise à l'aide d'une file de la façon suivante :
  - Mettre la racine dans la file
  - Tant que la file n'est pas vide :
    1. Récupérer en tête de file le nœud à traiter
    2. Traiter le nœud
    3. Ajouter ses fils dans la file
- Parcours: 1, 2, 8, 3, 6



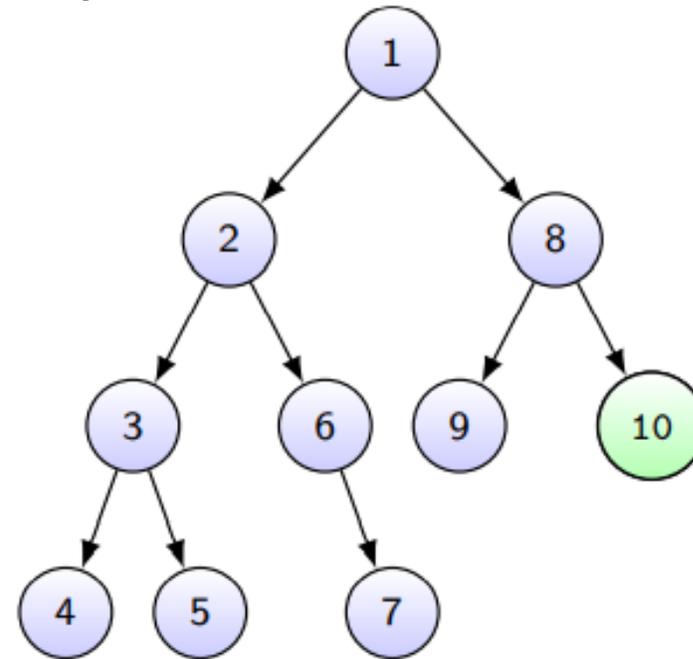
# Parcours en largeur

- **Parcours en largeur** : On explore les nœuds un par un sur un même niveau. On passe ensuite sur le niveau suivant, et ainsi de suite (on va vers les frères avant de parcourir les fils).
- Le parcours en largeur se réalise à l'aide d'une file de la façon suivante :
  - Mettre la racine dans la file
  - Tant que la file n'est pas vide :
    1. Récupérer en tête de file le nœud à traiter
    2. Traiter le nœud
    3. Ajouter ses fils dans la file
- Parcours: 1, 2, 8, 3, 6, 9



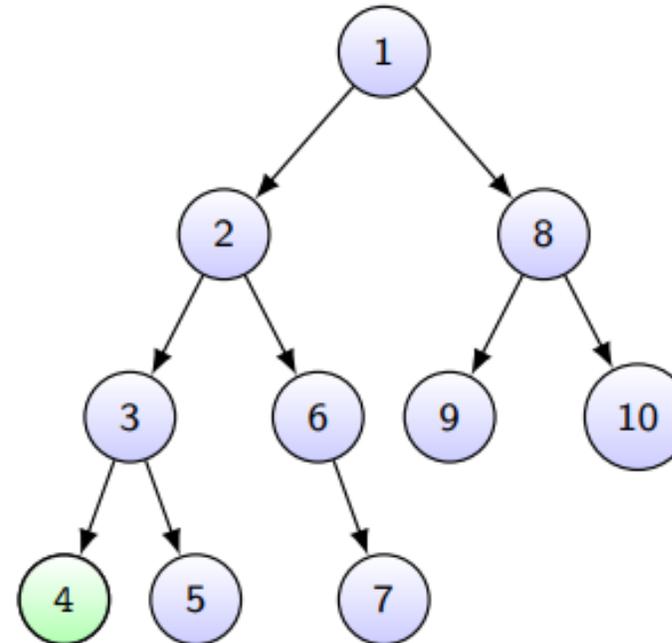
# Parcours en largeur

- **Parcours en largeur** : On explore les nœuds un par un sur un même niveau. On passe ensuite sur le niveau suivant, et ainsi de suite (on va vers les frères avant de parcourir les fils).
- Le parcours en largeur se réalise à l'aide d'une file de la façon suivante :
  - Mettre la racine dans la file
  - Tant que la file n'est pas vide :
    1. Récupérer en tête de file le nœud à traiter
    2. Traiter le nœud
    3. Ajouter ses fils dans la file
- Parcours: 1, 2, 8, 3, 6, 9, 10



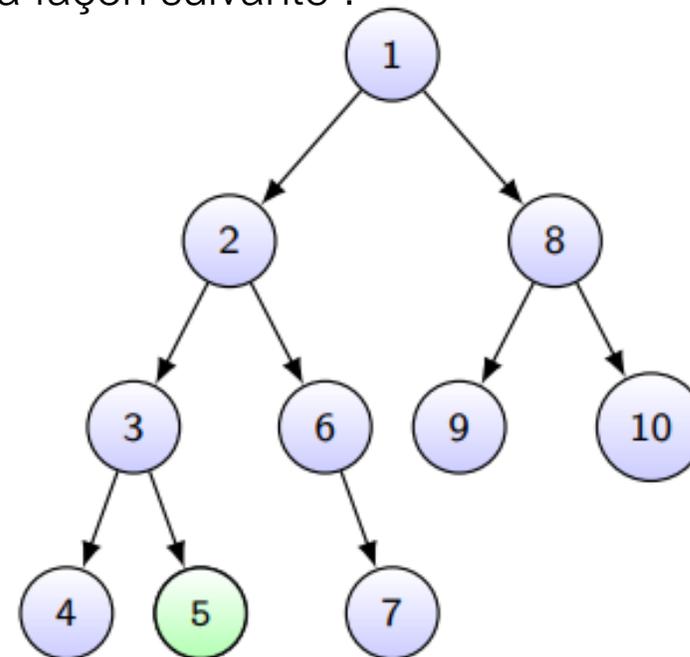
# Parcours en largeur

- **Parcours en largeur** : On explore les nœuds un par un sur un même niveau. On passe ensuite sur le niveau suivant, et ainsi de suite (on va vers les frères avant de parcourir les fils).
- Le parcours en largeur se réalise à l'aide d'une file de la façon suivante :
  - Mettre la racine dans la file
  - Tant que la file n'est pas vide :
    1. Récupérer en tête de file le nœud à traiter
    2. Traiter le nœud
    3. Ajouter ses fils dans la file
- Parcours: 1, 2, 8, 3, 6, 9, 10, 4



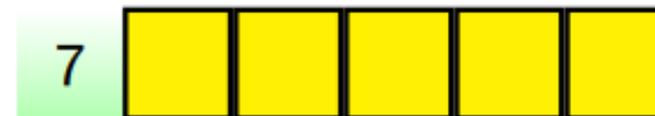
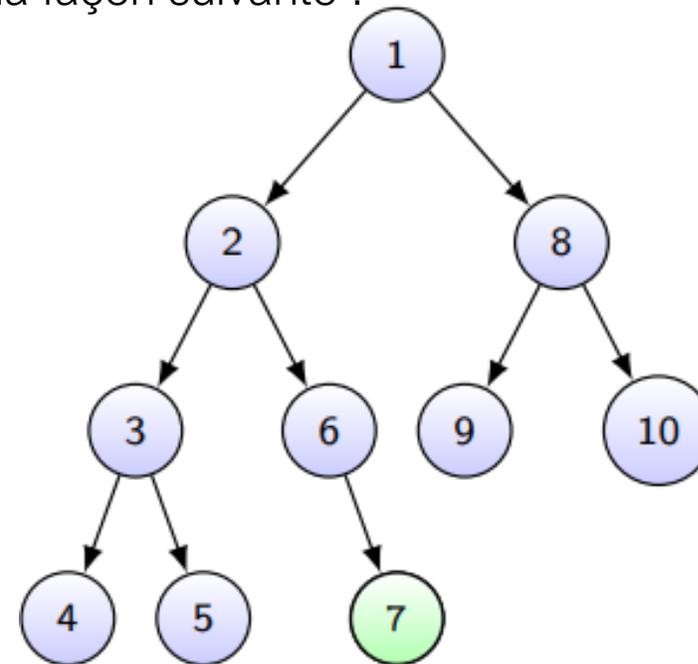
# Parcours en largeur

- **Parcours en largeur** : On explore les nœuds un par un sur un même niveau. On passe ensuite sur le niveau suivant, et ainsi de suite (on va vers les frères avant de parcourir les fils).
- Le parcours en largeur se réalise à l'aide d'une file de la façon suivante :
  - Mettre la racine dans la file
  - Tant que la file n'est pas vide :
    1. Récupérer en tête de file le nœud à traiter
    2. Traiter le nœud
    3. Ajouter ses fils dans la file
- Parcours: 1, 2, 8, 3, 6, 9, 10, 4, 5



# Parcours en largeur

- **Parcours en largeur** : On explore les nœuds un par un sur un même niveau. On passe ensuite sur le niveau suivant, et ainsi de suite (on va vers les frères avant de parcourir les fils).
- Le parcours en largeur se réalise à l'aide d'une file de la façon suivante :
  - Mettre la racine dans la file
  - Tant que la file n'est pas vide :
    1. Récupérer en tête de file le nœud à traiter
    2. Traiter le nœud
    3. Ajouter ses fils dans la file
- Parcours: 1, 2, 8, 3, 6, 9, 10, 4, 5, 7



# Parcours en largeur

- Le parcours en largeur se réalise à l'aide d'une file de la façon suivante :

- Mettre la racine dans la file
- Tant que la file n'est pas vide :
  - Récupérer en tête de file le nœud à traiter
  - Traiter le nœud
  - Ajouter ses fils dans la file

PROCEDURE `parcoursLargeur(a : pointeur sur Arbre)`

VARIABLE

`noeud : pointeur sur Arbre`

`f : File de pointeurs sur Arbre`

DEBUT

SI (`nonVide(a)`) ALORS

`creerFile(f)`

`enfiler(f, a)`

TANT QUE NON `fileVide(f)` FAIRE

`noeud ← defiler(f)`

`traiter(noeud)`

SI (`existeFilsGauche(noeud)`) `enfiler(f, fg(noeud))`

SI (`existeFilsDroit (noeud)`) `enfiler(f, fd(noeud))`

FIN TANT QUE

FIN SI

FIN

