UNIVERSITÉ DE PAU ET DES PAYS DE L'ADOUR

Collège STEE

Département d'Informatique

B.P. 1155

64013 PAU CEDEX

Téléphone secrétariat : 05.59.40.79.64

Télécopie : 05.59.40.76.54

TYPE DE STRUCTURE D'ARBRE

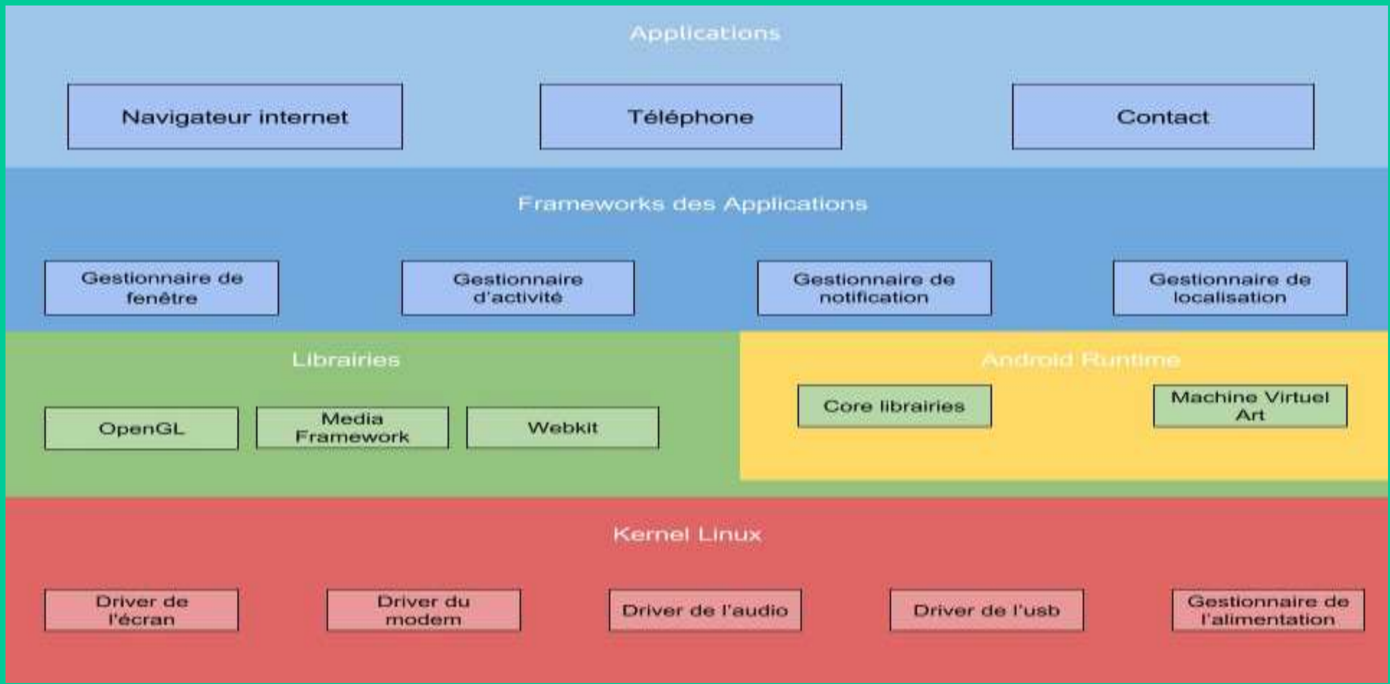
La structure d'arbre est l'une **des plus importantes** et des **plus spécifiques** de l'informatique.

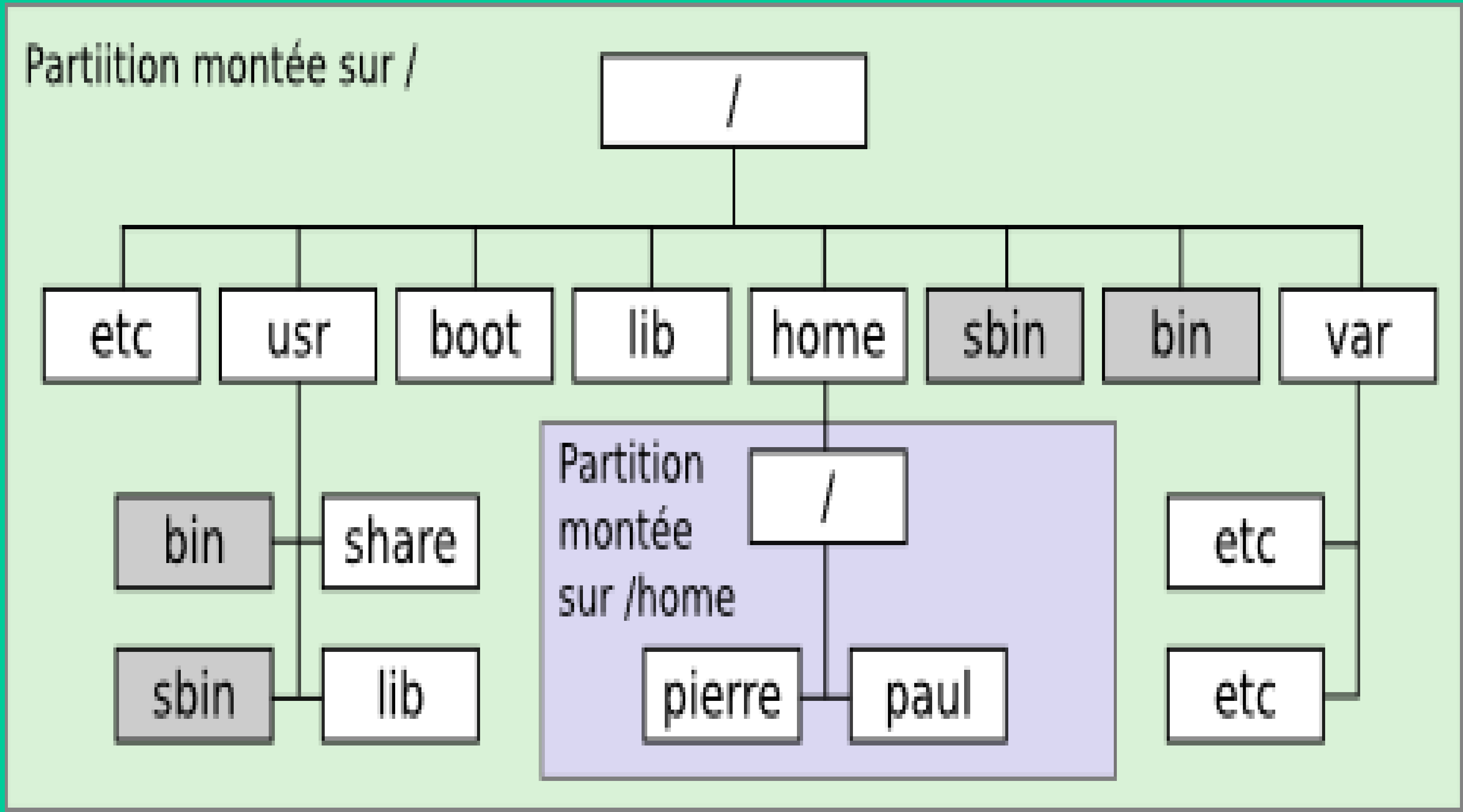
Pourquoi les arbres ?

L'arbre est une **structure de données** qui formalise par excellence le concept de **hiérarchie**.

Exemples

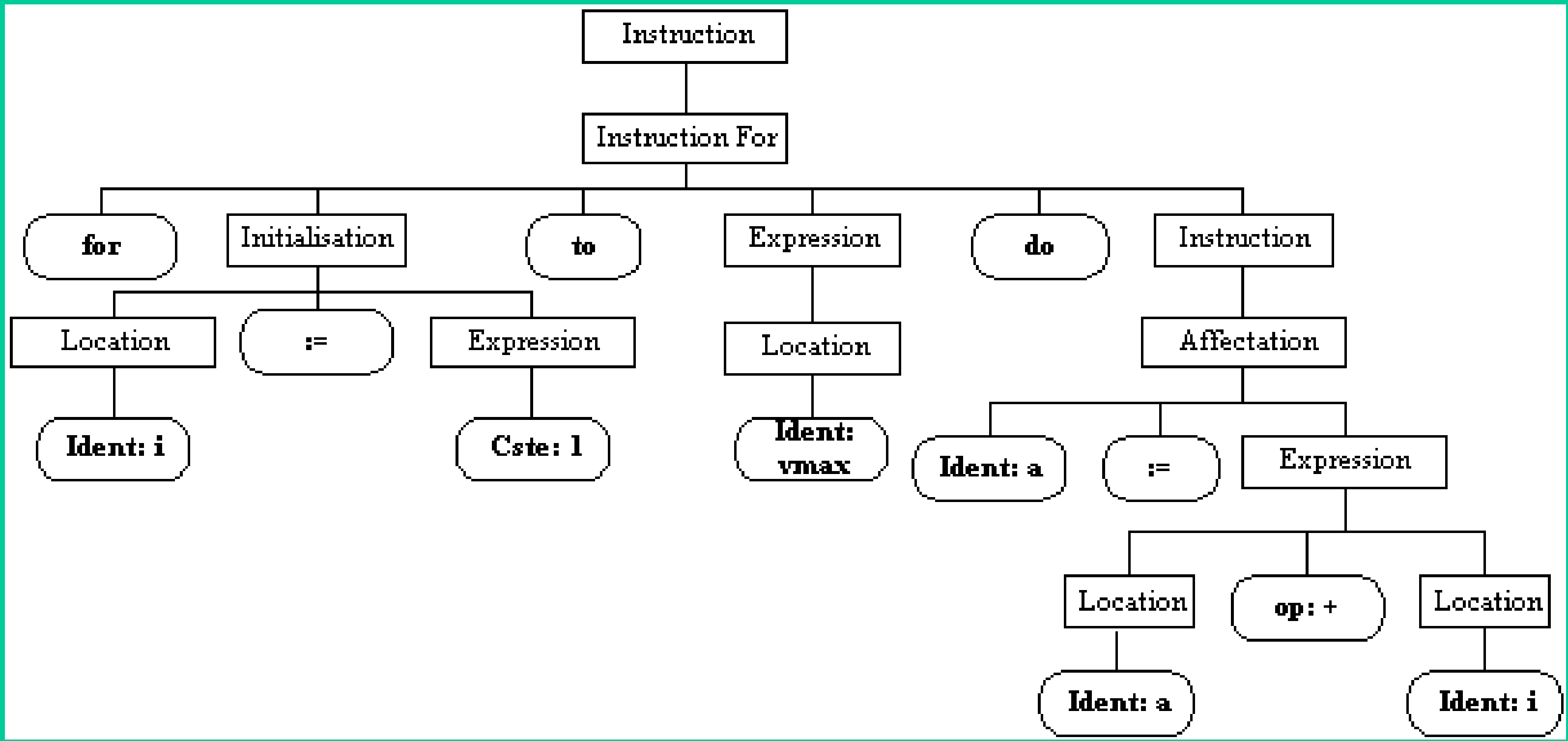
C'est sous forme d'**arbre** que sont organisés les **fichiers** dans les systèmes d'exploitation tels que UNIX.



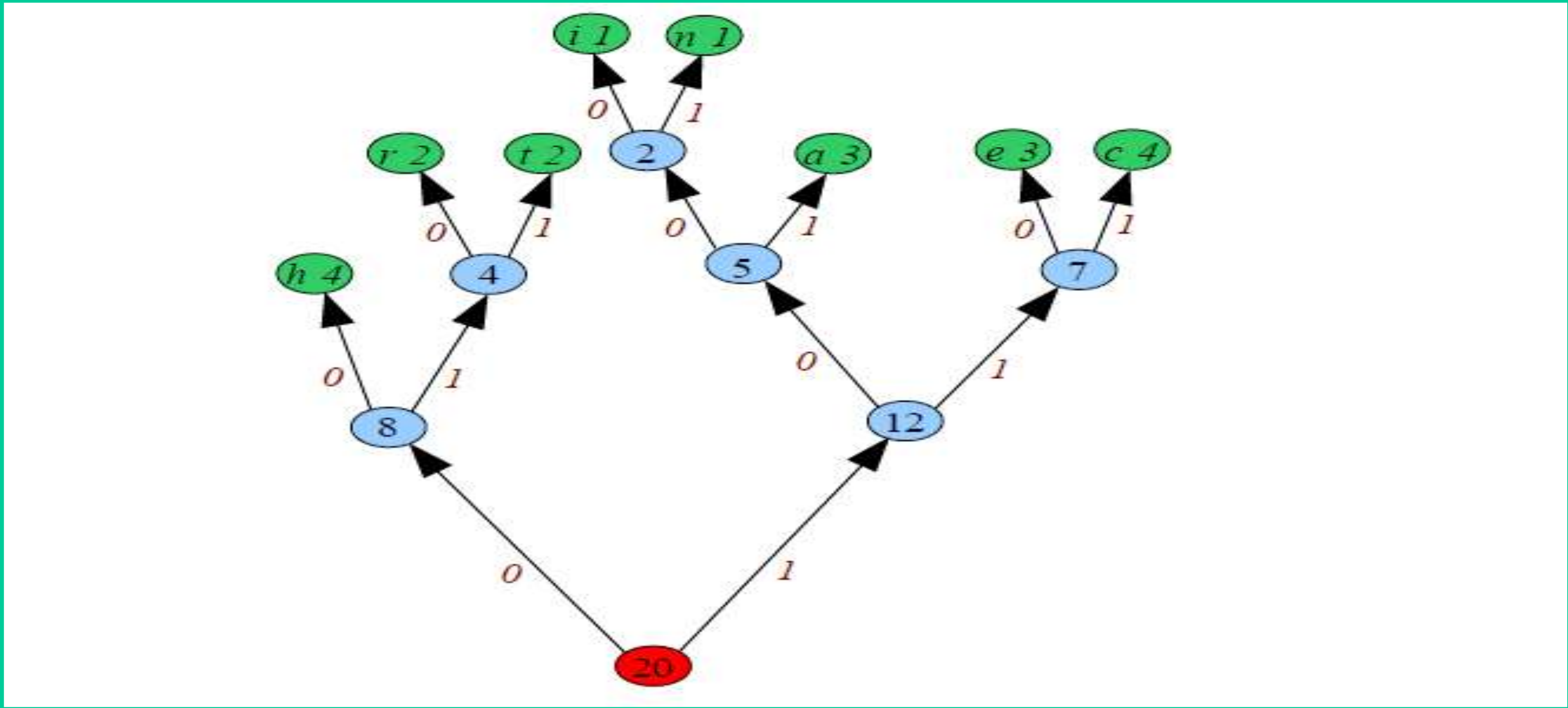


Partiition montée sur /
/
etc
usr
boot
lib
home
sbin
bin
var
bin
share
sbin
lib
Partition
montée
sur /home
/
pierre
paul
etc
etc

C’est également sous forme d’**arbre** que sont représentés les **instructions** analysées par un **compilateur**.



La **compression des données** utilise un codage d'arbre



Codage de la phrase: «recherche chat châtain »

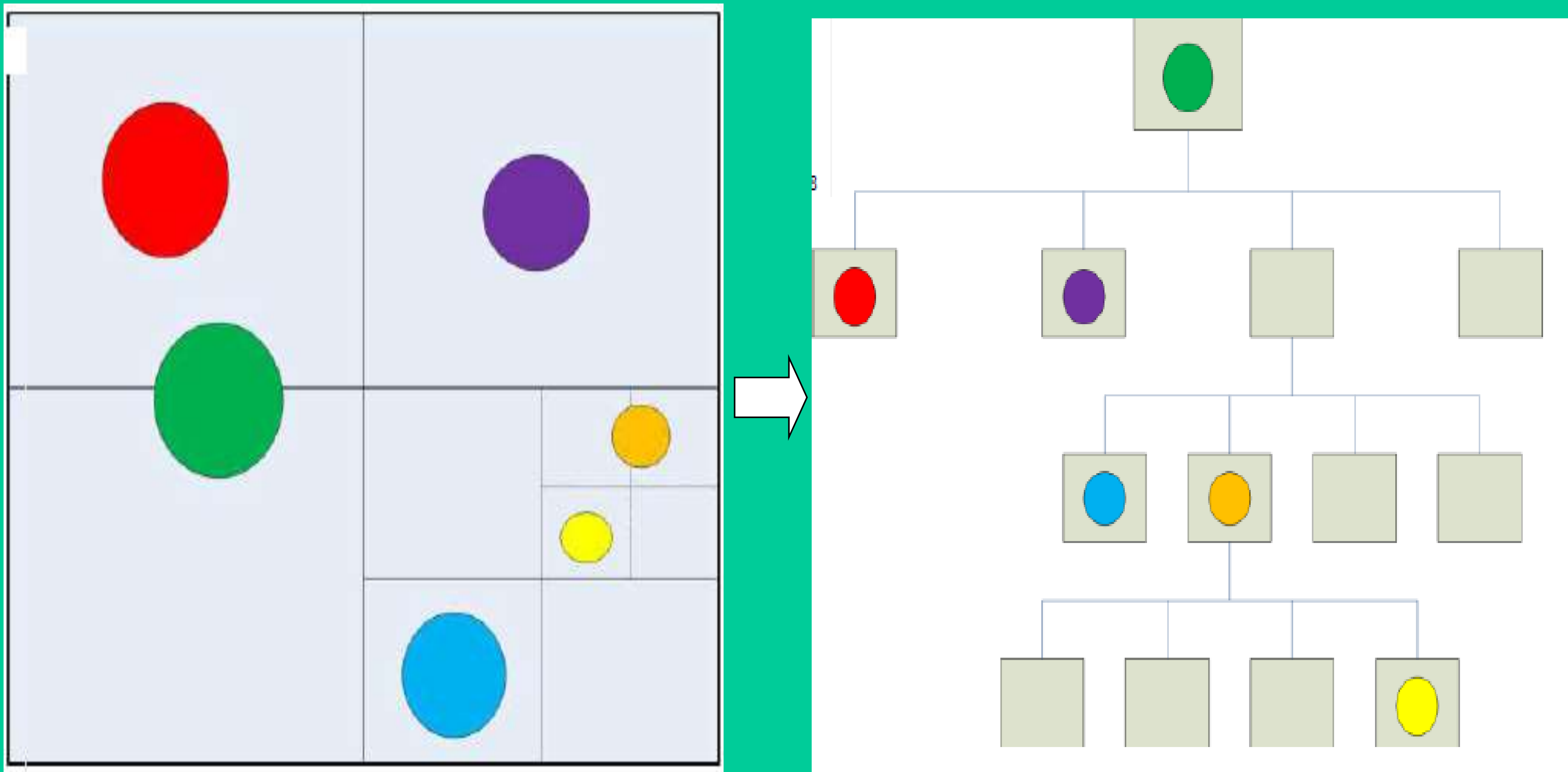
L’algorithme de **Huffman** permet le codage des 20 caractères (octets) par 58 bits est :

0101101110011001011100110111001010111110010101110110001001

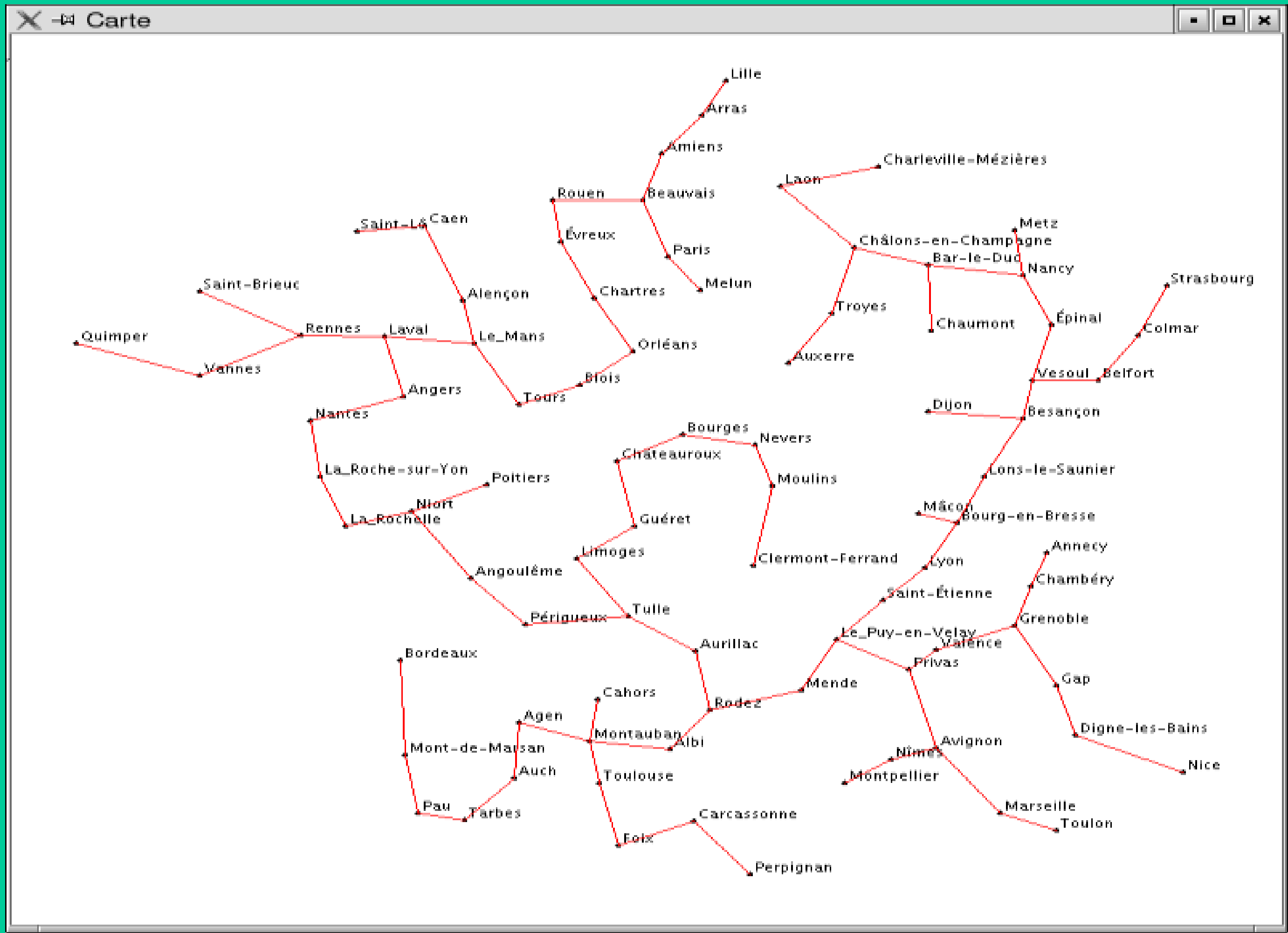
Le taux de compression est alors:

58/160 = **0,3625**

La **synthèse d’images** utilise la représentation d’arbre



La **cartographie** utilise un **arbre** pour représenter un pays ou région.



Qu'est-ce qu'un arbre ?

Un **arbre** est une **structure** :

- composée d'un **ensemble d'objets** appelés **nœuds**.
- où les **nœuds** sont disposés selon une **relation** de **hiérarchie**

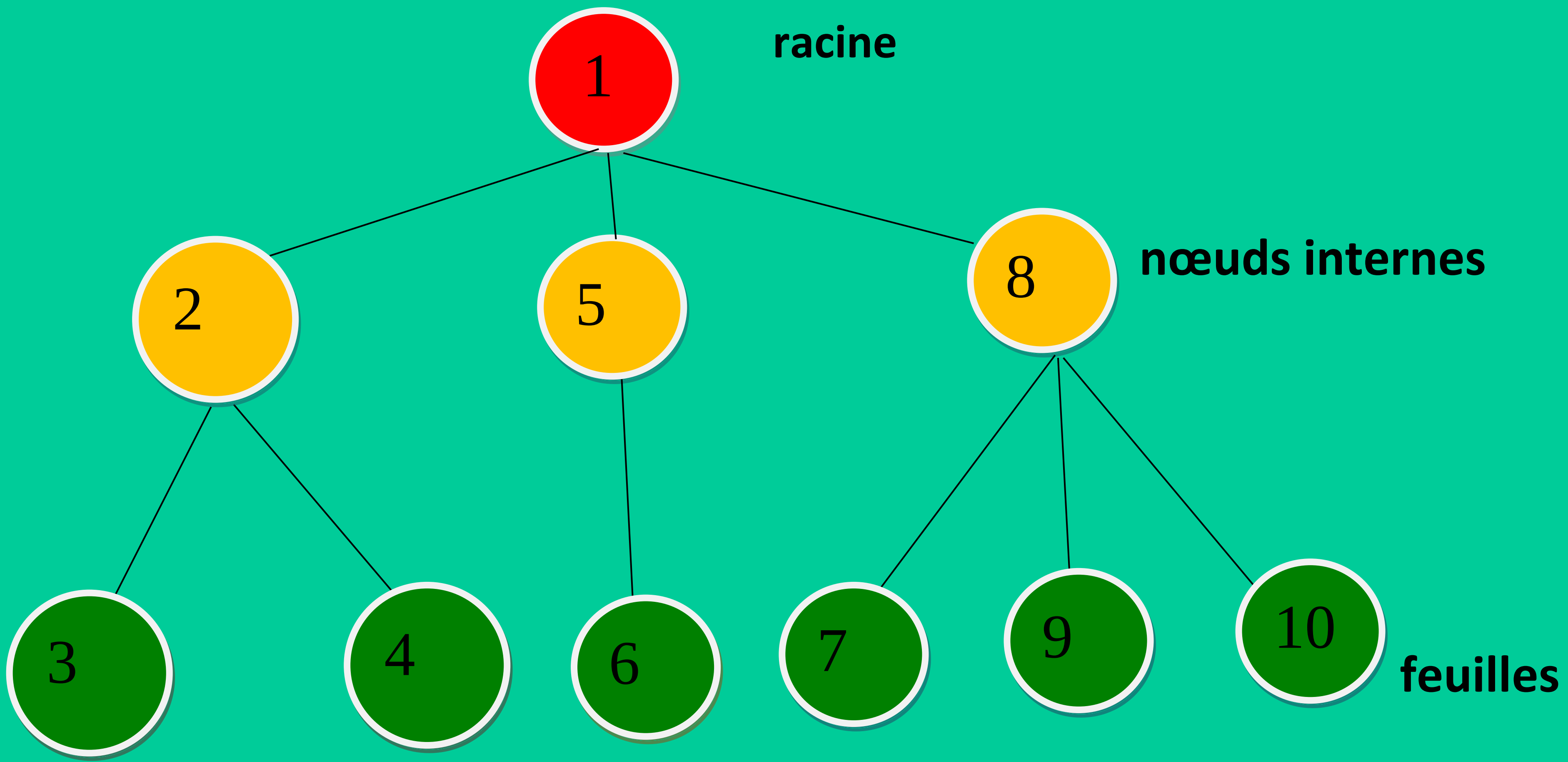
Dans un arbre, la **hiérarchie** distingue :

-un nœud **unique** appelé **racine**,

-des nœuds **terminaux** appelés **feuilles**,

-des nœuds **intermédiaires**: **nœuds internes**.

Représentation d’un arbre à 3 niveaux



I- Arbre binaire

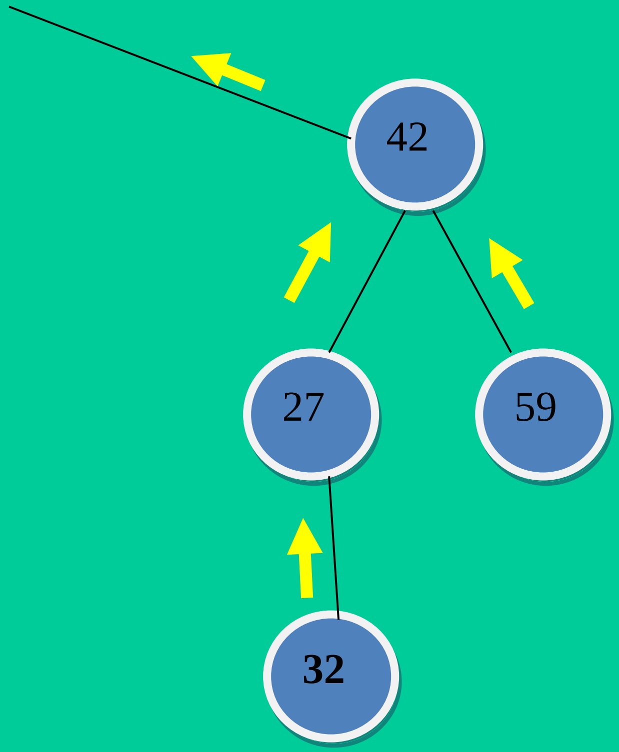
Un arbre binaire est un **arbre particulier**.

Chaque nœud est en relation avec au maximum **trois** nœuds :

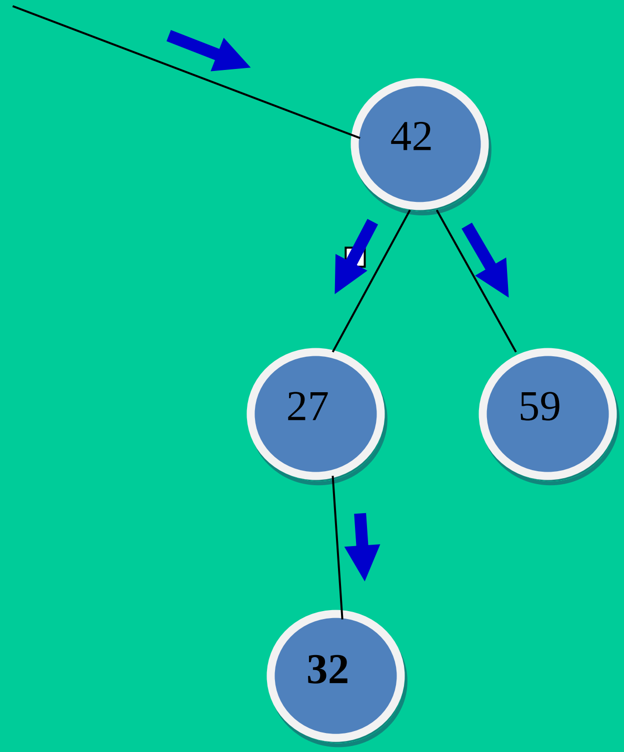
-1 nœud **père**: seule la **racine n’a pas de père**,

- au plus 2 nœuds **fils**: une **feuille n’a pas de fils**,

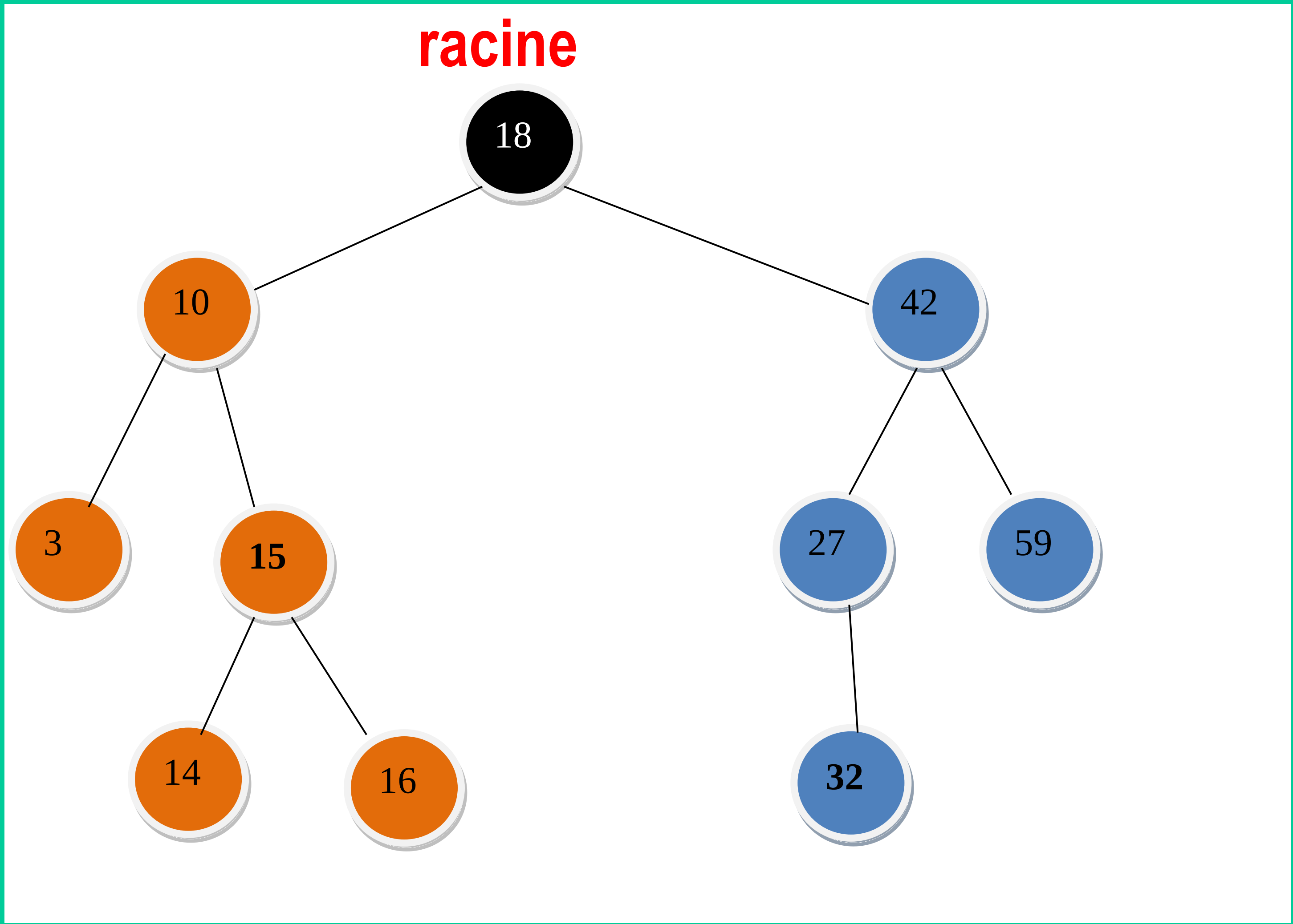
Relation **père**



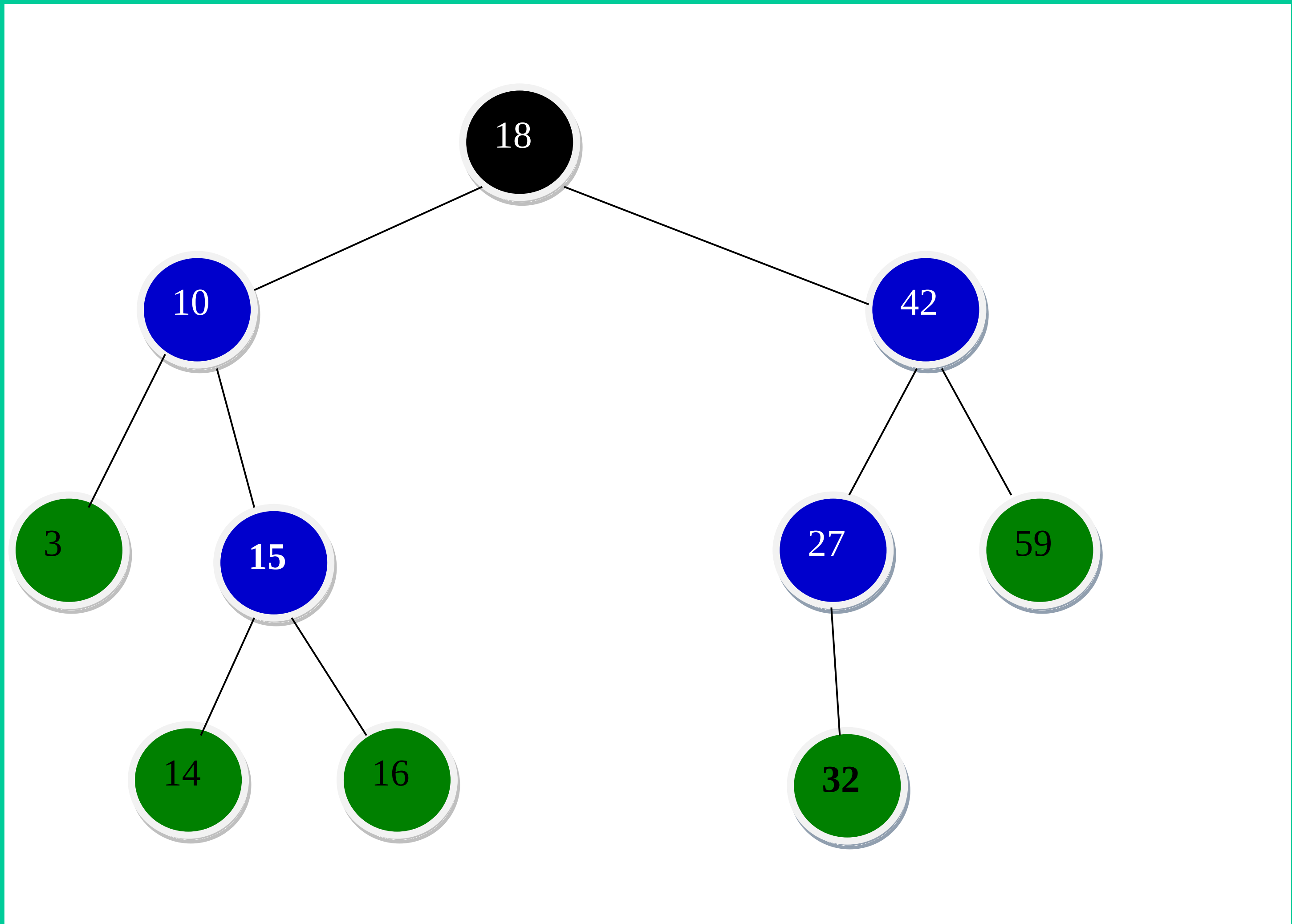
Relation **fils**



La **racine** est un nœud qui n'a **pas de père**



Les **feuilles** n'ont **aucun fils**



1- définition récursive

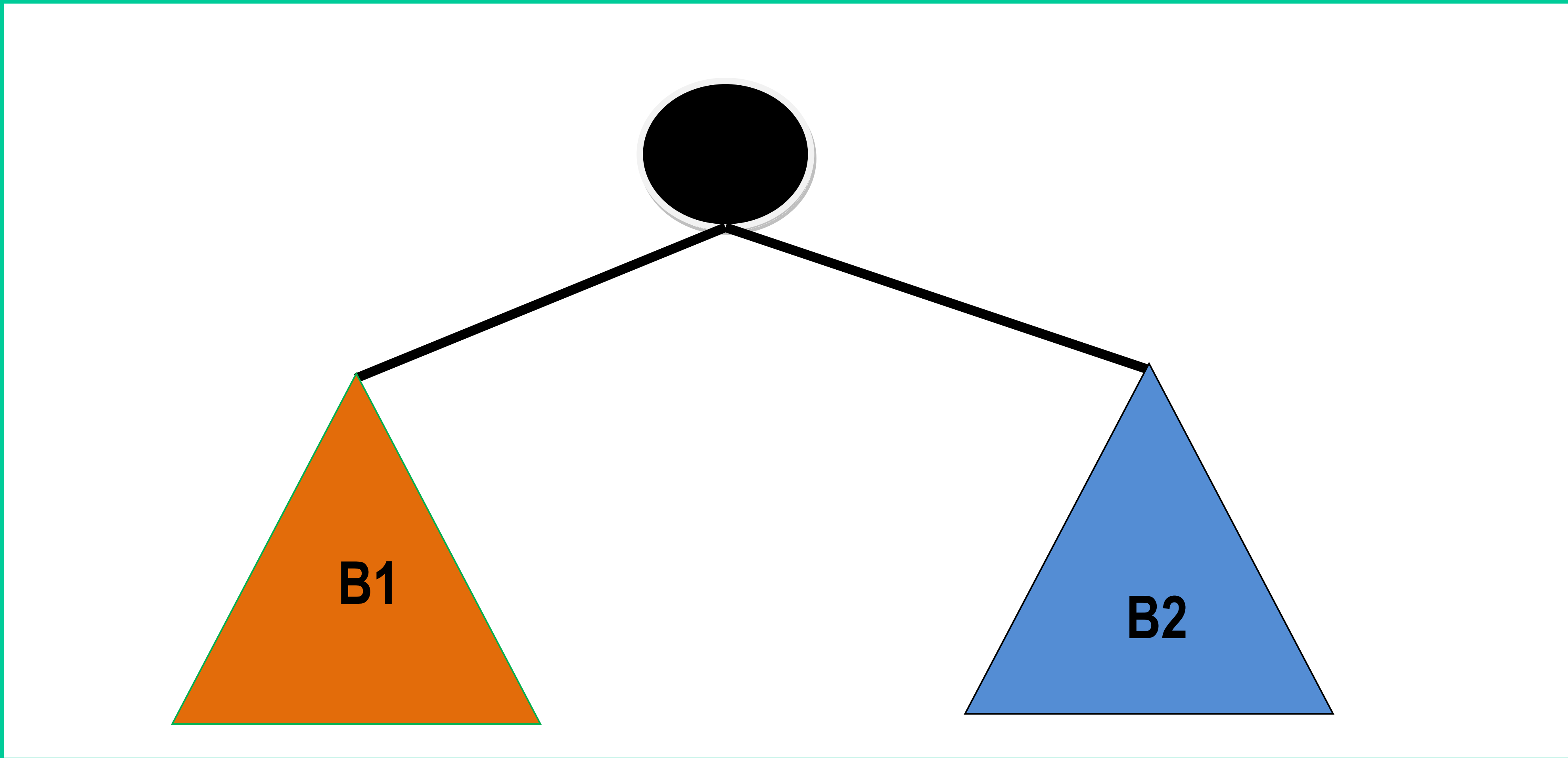
On note

$$\mathbf{B} = < \mathbf{o}, \mathbf{B_1}, \mathbf{B_2} >$$

un arbre binaire **B** où :

- **o** est sa **racine** de **B**,
- $\mathbf{B_1}$ est son **sous-arbre gauche**,
- $\mathbf{B_2}$ est son **sous-arbre droit**.

Illustration simple d'un arbre binaire

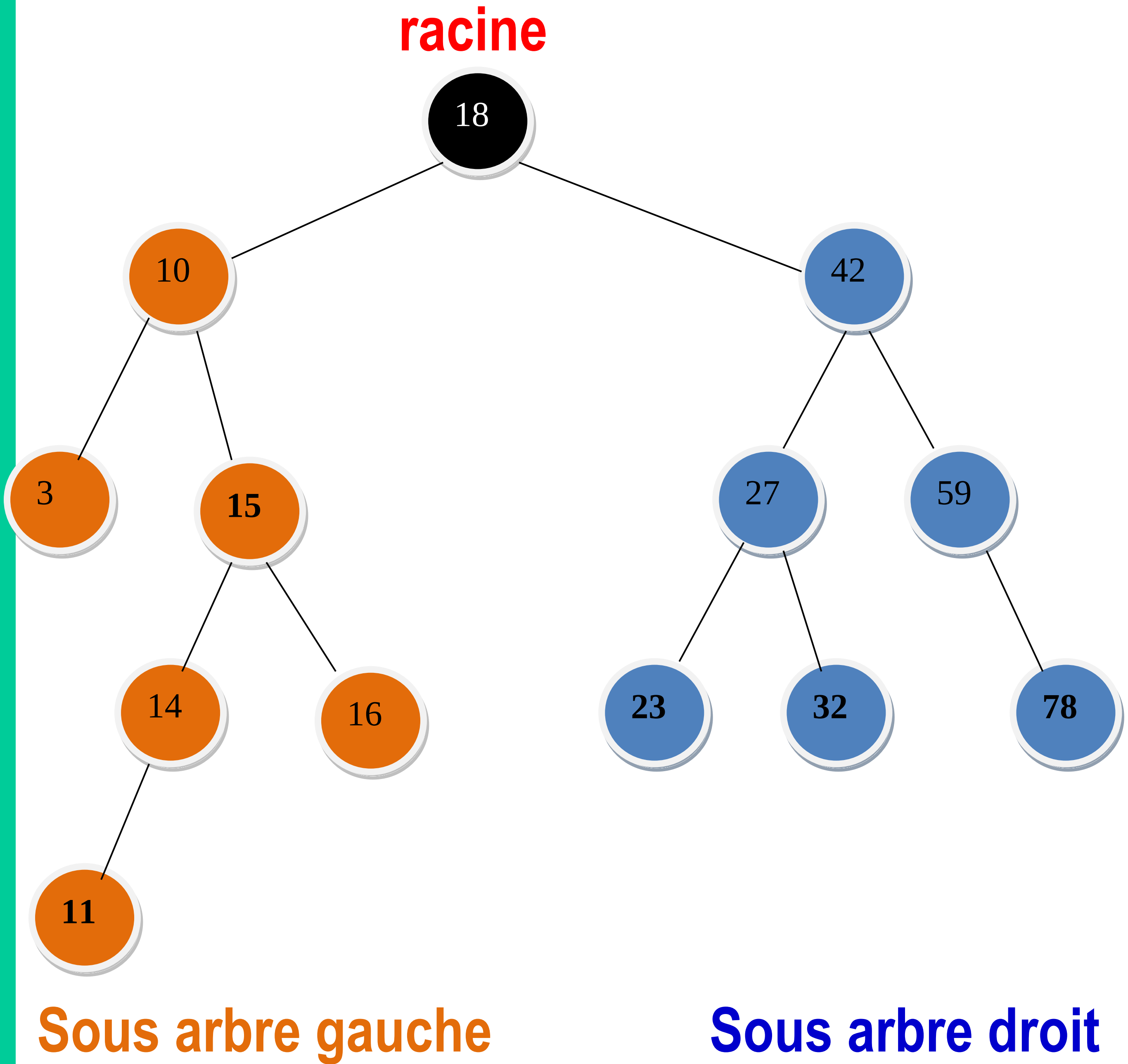


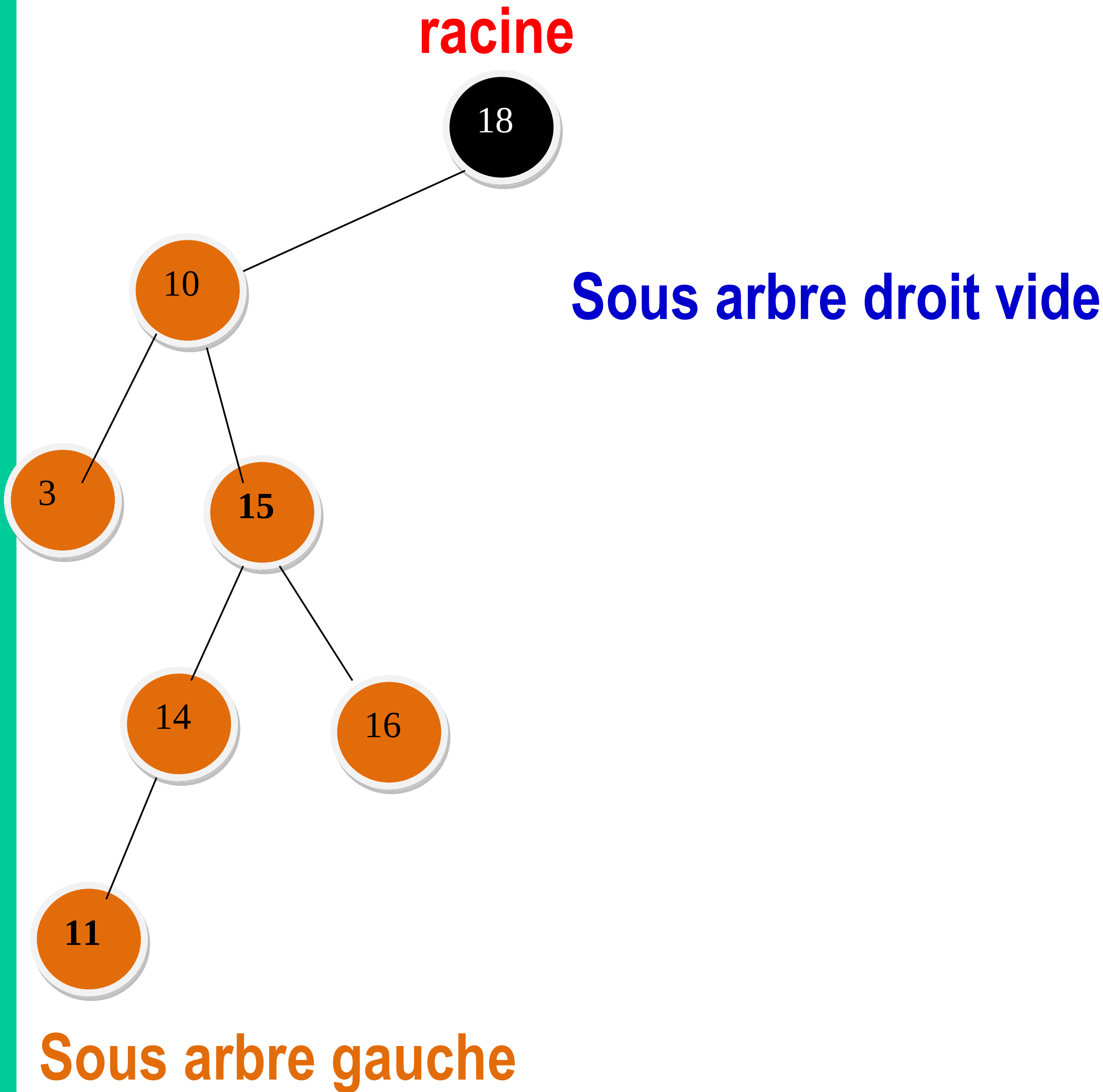
Un arbre binaire est :

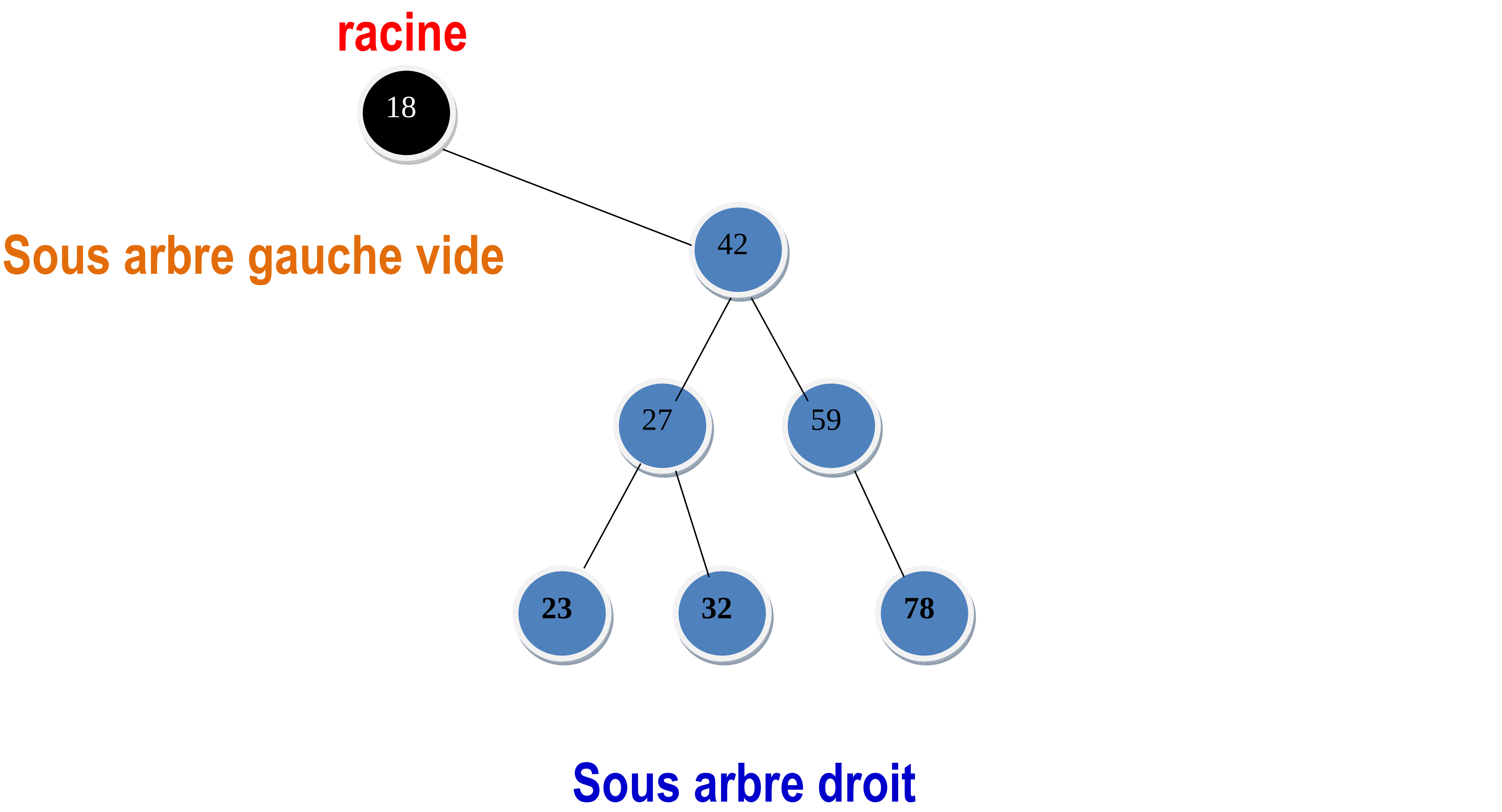
- soit **vide**,
- soit de la forme **$<o, B_1, B_2>$**

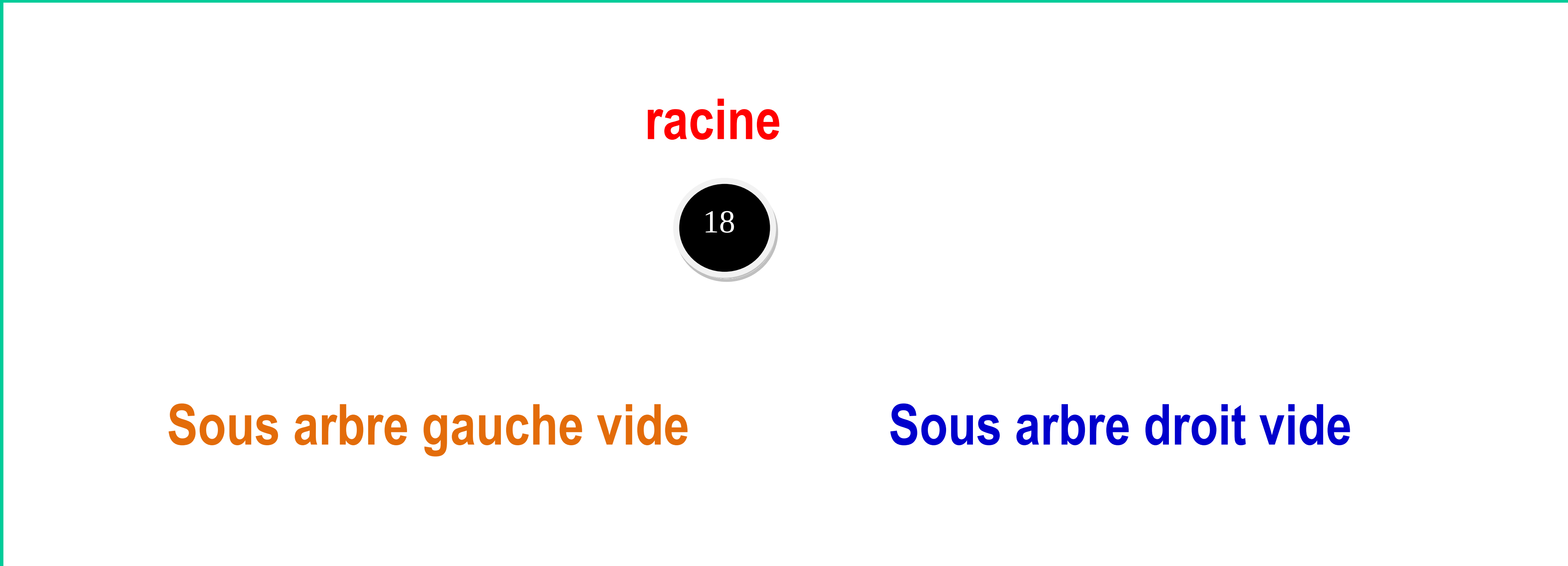
où :

- **B_1** et **B_2** sont des arbres binaires **disjoints**,
- **o**, le nœud **racine**.


racine
18
10
42
3
15
27
59
14
16
23
32
78
11
Sous arbre gauche
Sous arbre droit


racine
18
10
Sous arbre droit vide
3
15
14
16
11
Sous arbre gauche


racine
18
Sous arbre gauche vide
42
27
59
23
32
78
Sous arbre droit


racine
18
Sous arbre gauche vide
Sous arbre droit vide

2- Type abstrait arbre binaire

Une spécification **minimale** du type abstrait des arbres binaires peut être établie comme suit :

```
spec ARBRE0 [sort Noeud ] =
generated type Arbre[Noeud ] ::=
                arbreVide
                | construire(Noeud ; Arbre[Noeud ] ; Arbre[Noeud ] )
end
```

La spécification ARBRE0 peut être enrichie comme suit:

spec ARBRE [sort Noeud] =

ARBRE0[sort Noeud]

then

pred

estVide : Arbre[Noeud]

ops

gauche : Arbre[Noeud] ->? Arbre[Noeud]
droit : Arbre[Noeud] ->? Arbre[Noeud]
racine : Arbre[Noeud] ->? Noeud

forall B, B_1, B_2: Arbre[Noeud]; o: Noeud

. **def racine**(B) <=> not **estVide**(B)
. **def gauche**(B) <=> not **estVide**(B)
. **def droit**(B) <=> not **estVide**(B)

. **estVide**(arbreVide)
. not **estVide**(construire(o, B_1, B_2))

. **gauche**(construire(o, B_1, B_2)) = B_1
. **droit**(construire(o, B_1, B_2)) = B_2

. **racine**(construire(**o**, B_1, B_2)) = **o**

end

REMARQUE :

On peut ajouter l'opération **contenu** qui permet d'associer à chaque nœud une **information** de sorte ELEMENT.

contenu: ARBRE x NŒUD $\rightarrow$ ELEMENT

Un arbre dont les nœuds contiennent des éléments est dit arbre **étiqueté**.

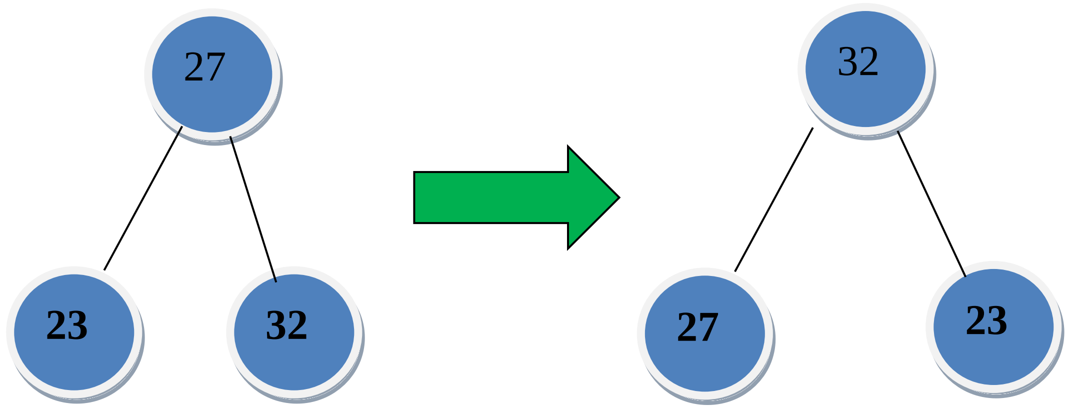
Si $B = < o, B_1, B_2 >$ est un arbre étiqueté tel que :

contenu$(B,o) = e$

alors on notera abusivement:

$B = < e, B_1, B_2 >$

Mais attention : un abus reste… un abus

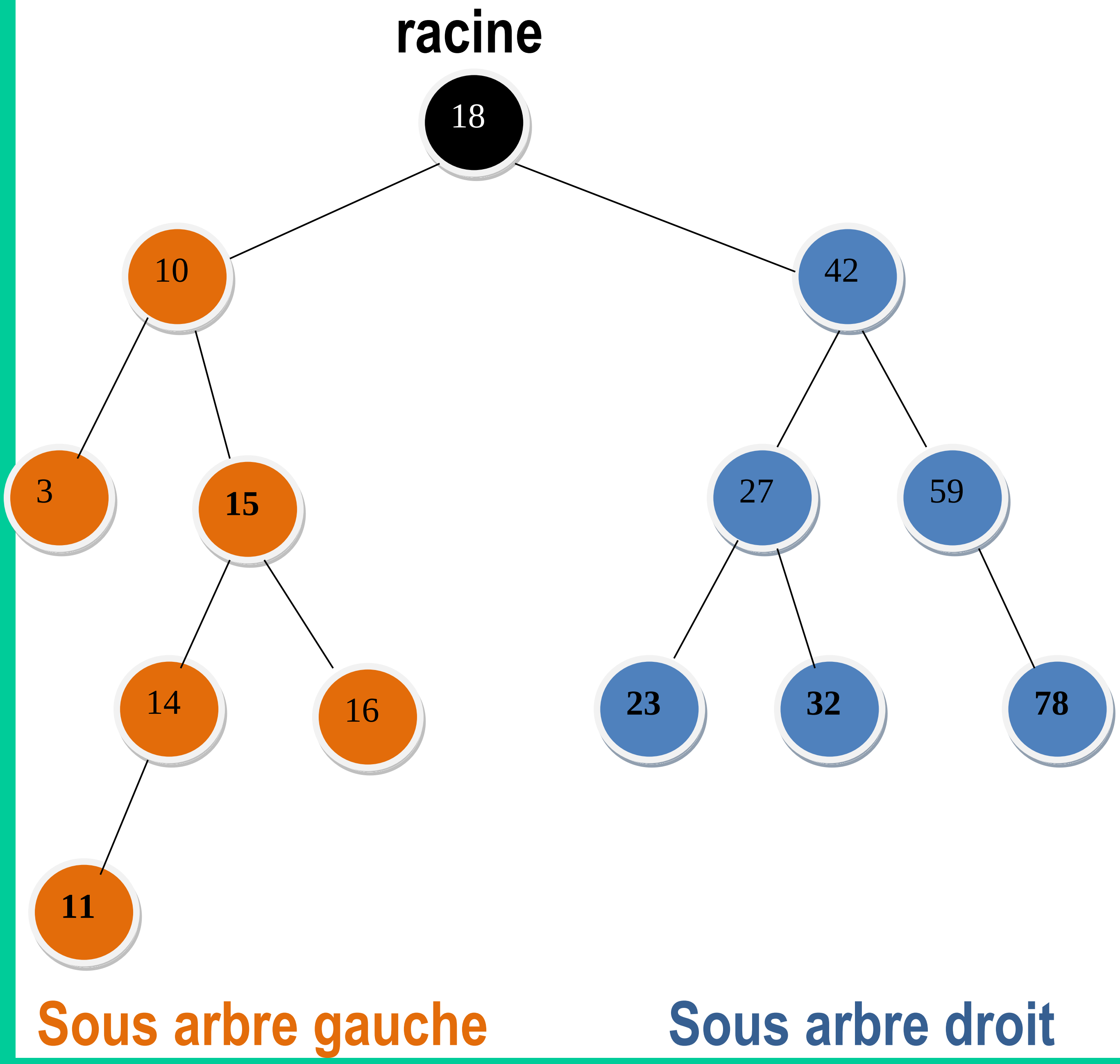


3- Vocabulaire des arbres

Relations entre arbres binaires

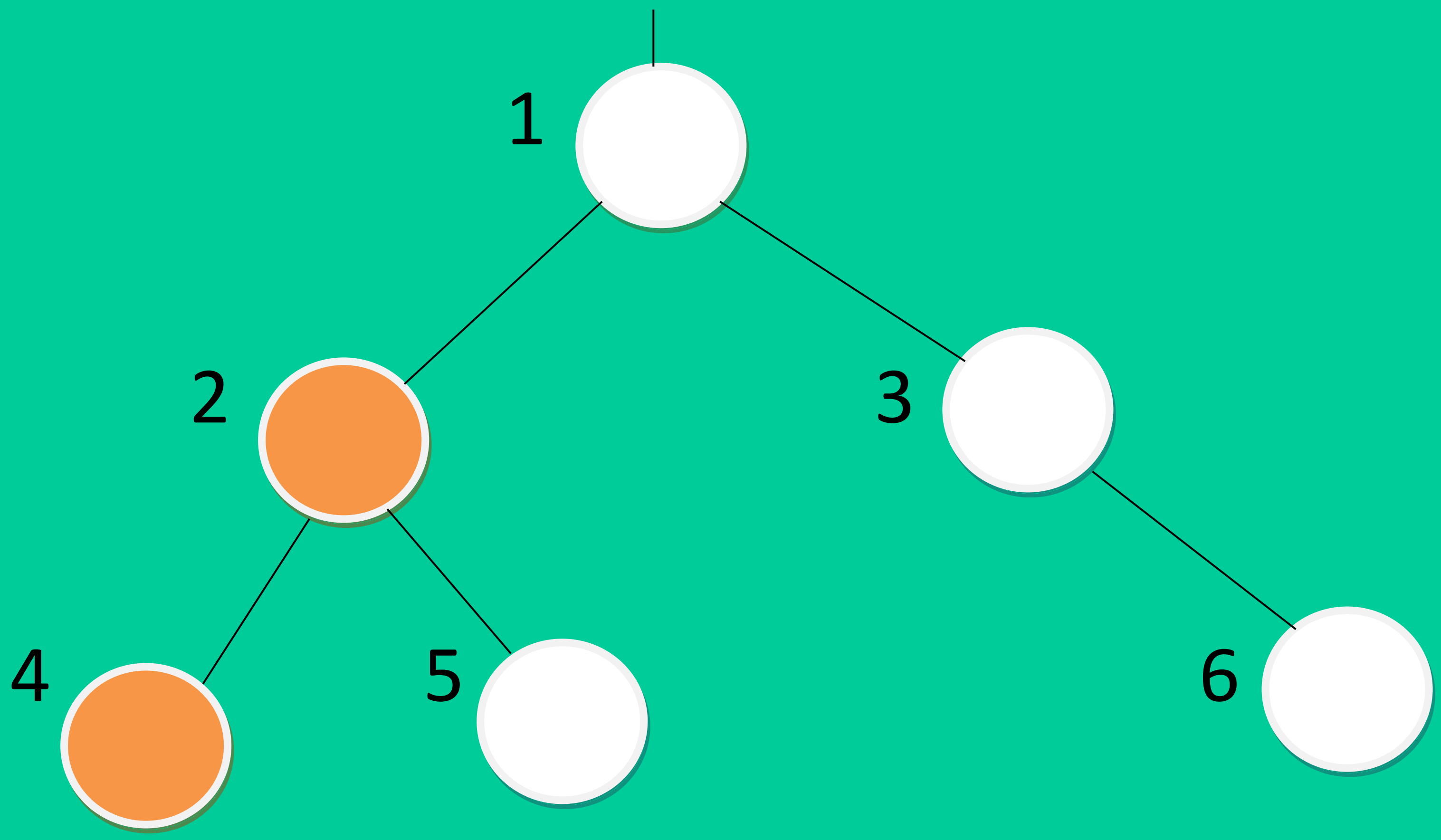
On dit que C est un **sous-arbre** de B si et seulement si:

- $C = B$
- ou $C = B_1$
- ou $C = B_2$
- ou C est **sous-arbre** de B_1 ou de B_2


racine
18
10
42
3
15
27
59
14
16
23
32
78
11
Sous arbre gauche
Sous arbre droit

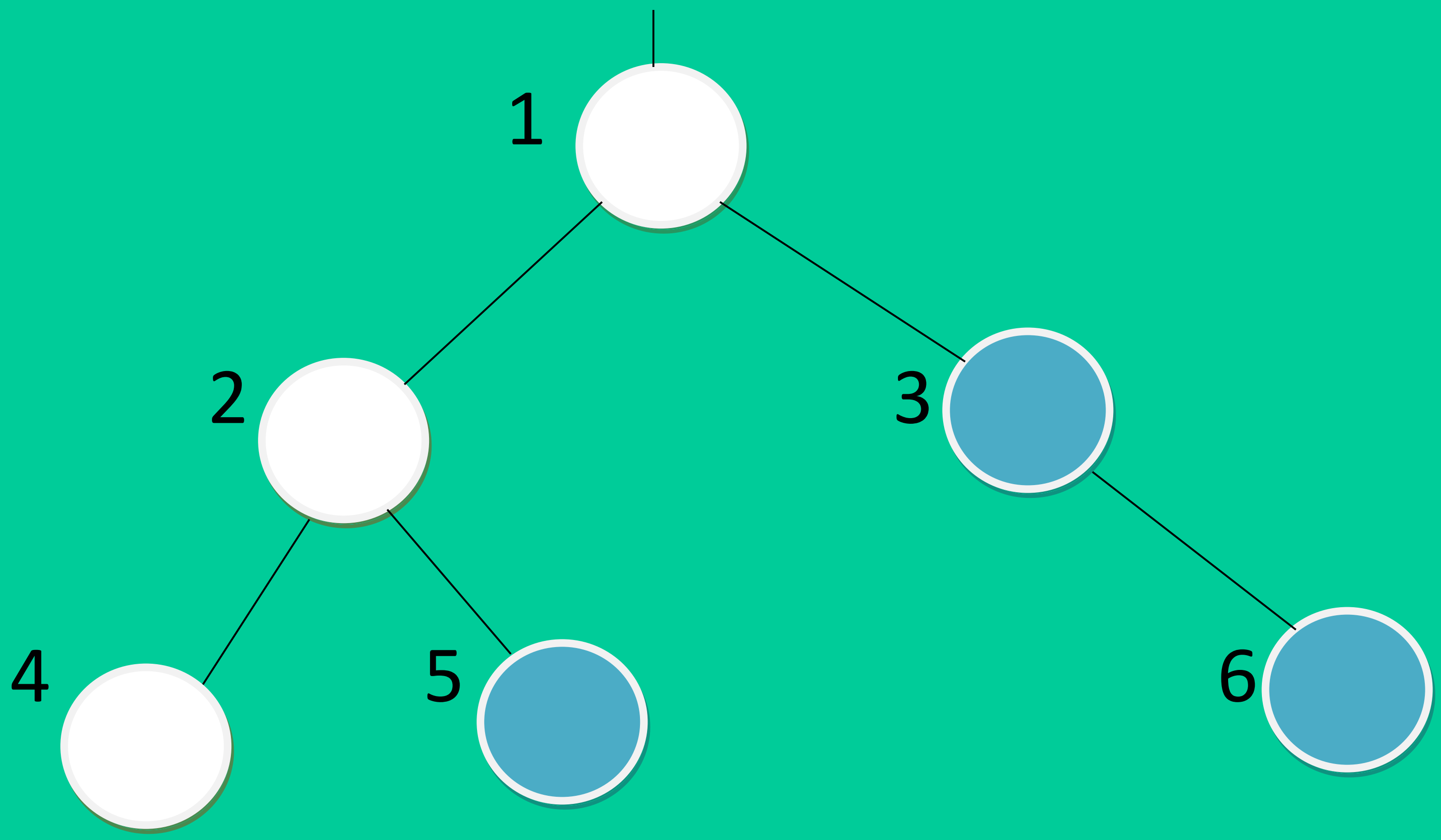
Relation entre nœuds

On appelle **fils gauche** d'un nœud, la **racine** de son sous arbre gauche.



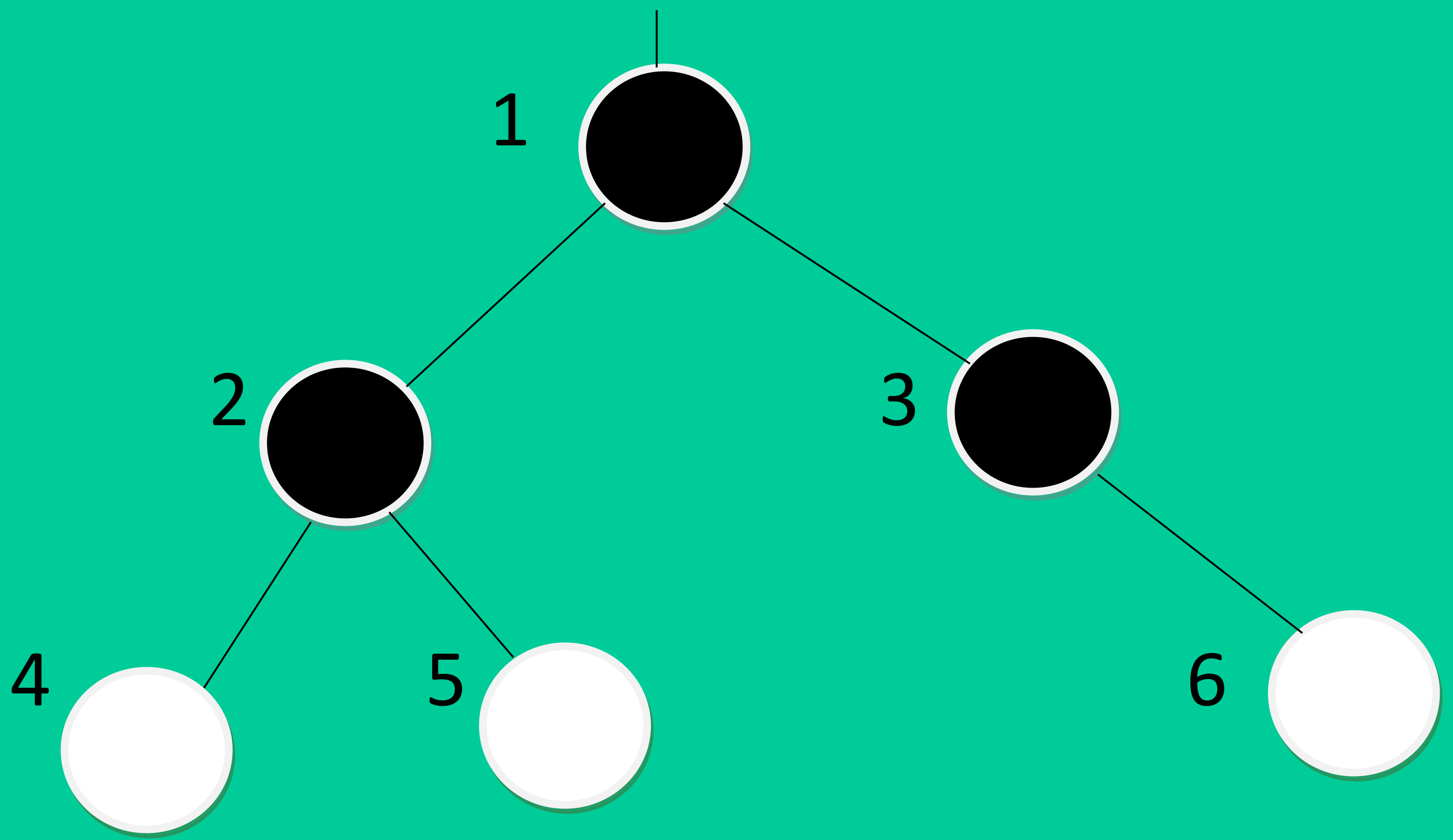
2 ←filsGauche(1) ; **4 ← filsGauche(2)**

On appelle **fils droit** d'un nœud, la **racine** de son sous arbre droit.



3 ← filsDroit (1) ; 5 ← filsDroit (2) ; 6← filsDroit (3) ;

Si un nœud **i** a pour fils un nœud **j**, on dit que **i** est le **<u>père</u>** de j :



père(1,2) ; père(1,3) ; père(2,4) ; père(2,5) ; père(3,6)

Important :

- Chaque nœud n'a qu'un **seul père**.

- Le nœud **a** est un **ascendant** du nœud **b** si et seulement si :

  - **a** est le **père** de **b**,
  - ou **a** est un **ascendant** du père de **b**.

La fonction :

est_ascendant? : NŒUD x NŒUD $\rightarrow$ BOOLEEN

est spécifiée comme suit:

estAscendant? (a, b :NOEUD) r : BOOLEEN

Pré : true

Post : r = (a = pere(b) $\vee$ estAscendant?(a, pere(b)))

Le nœud **a** est **descendant** de **b** si et seulement si:

- **a** est le fils de **b**,
- ou **a** est un **descendant** d'un fils de **b**.

La fonction **est_descendant?** est spécifiée comme suit :

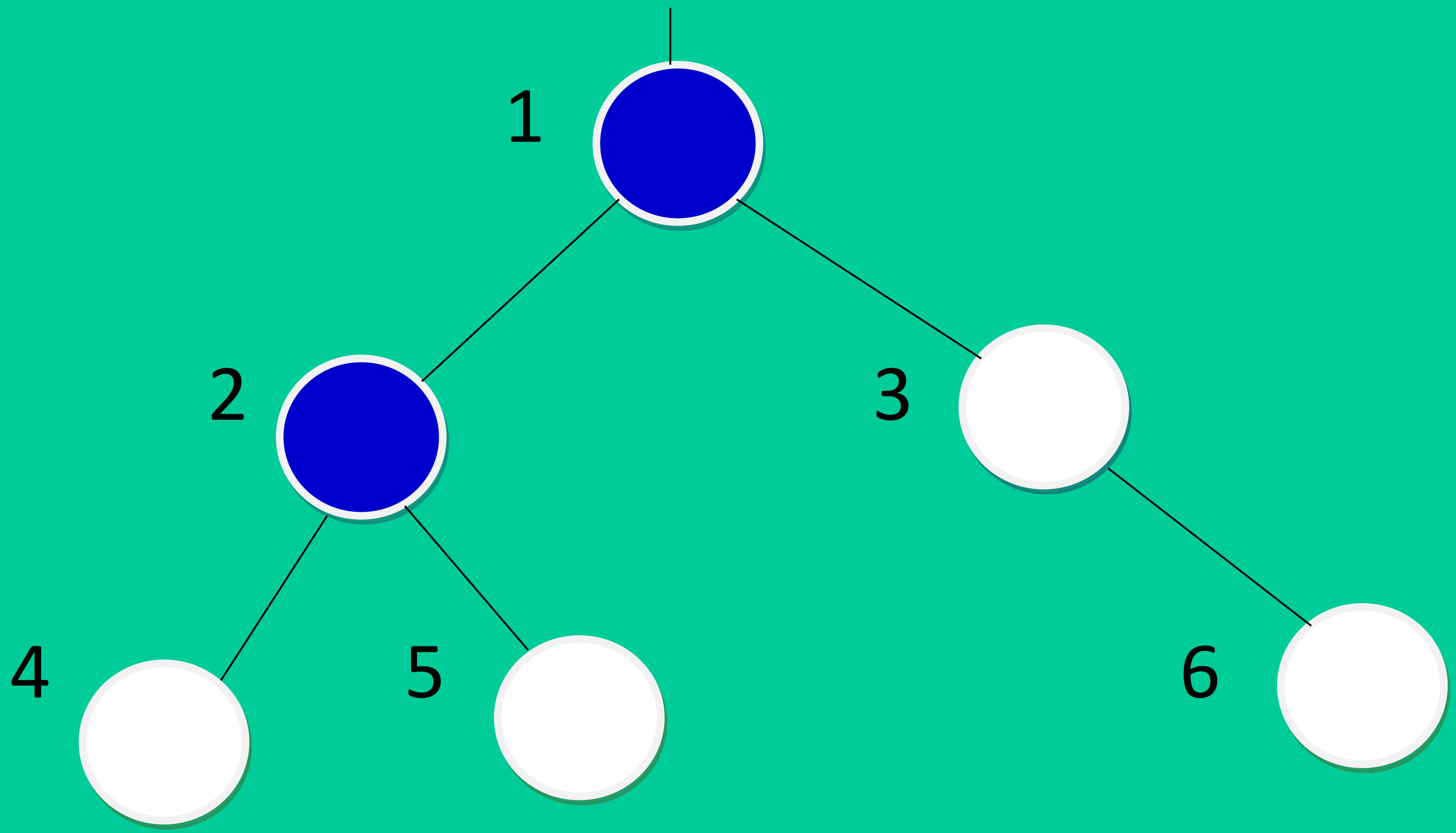
estDescendant? (a, b :NOEUD) r : BOOLEEN

Pré : true

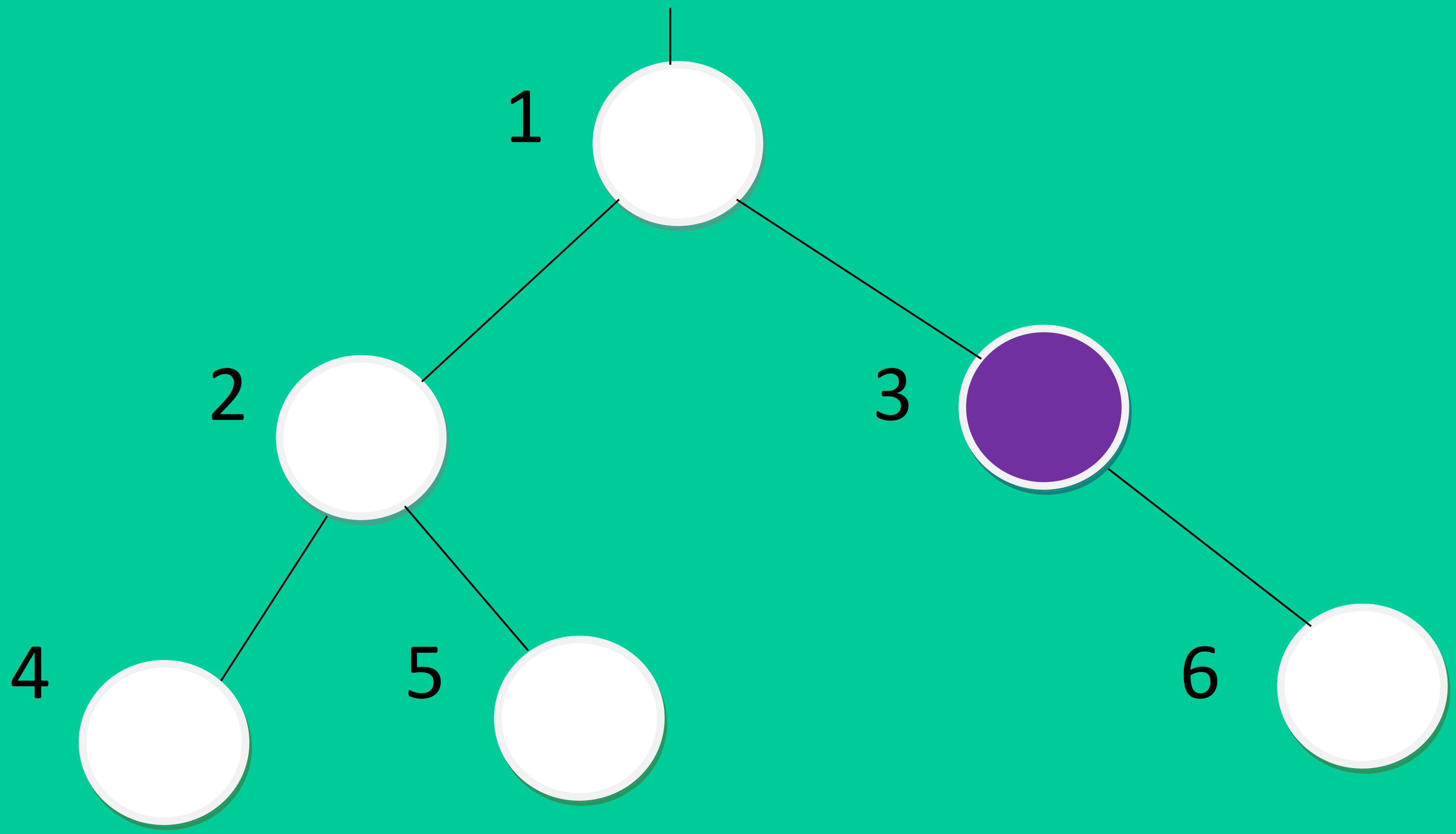
Post : r = (a = fils(b) ∨ estDescendant? (a,fils(b)))

Nœud interne et feuille d'un arbre

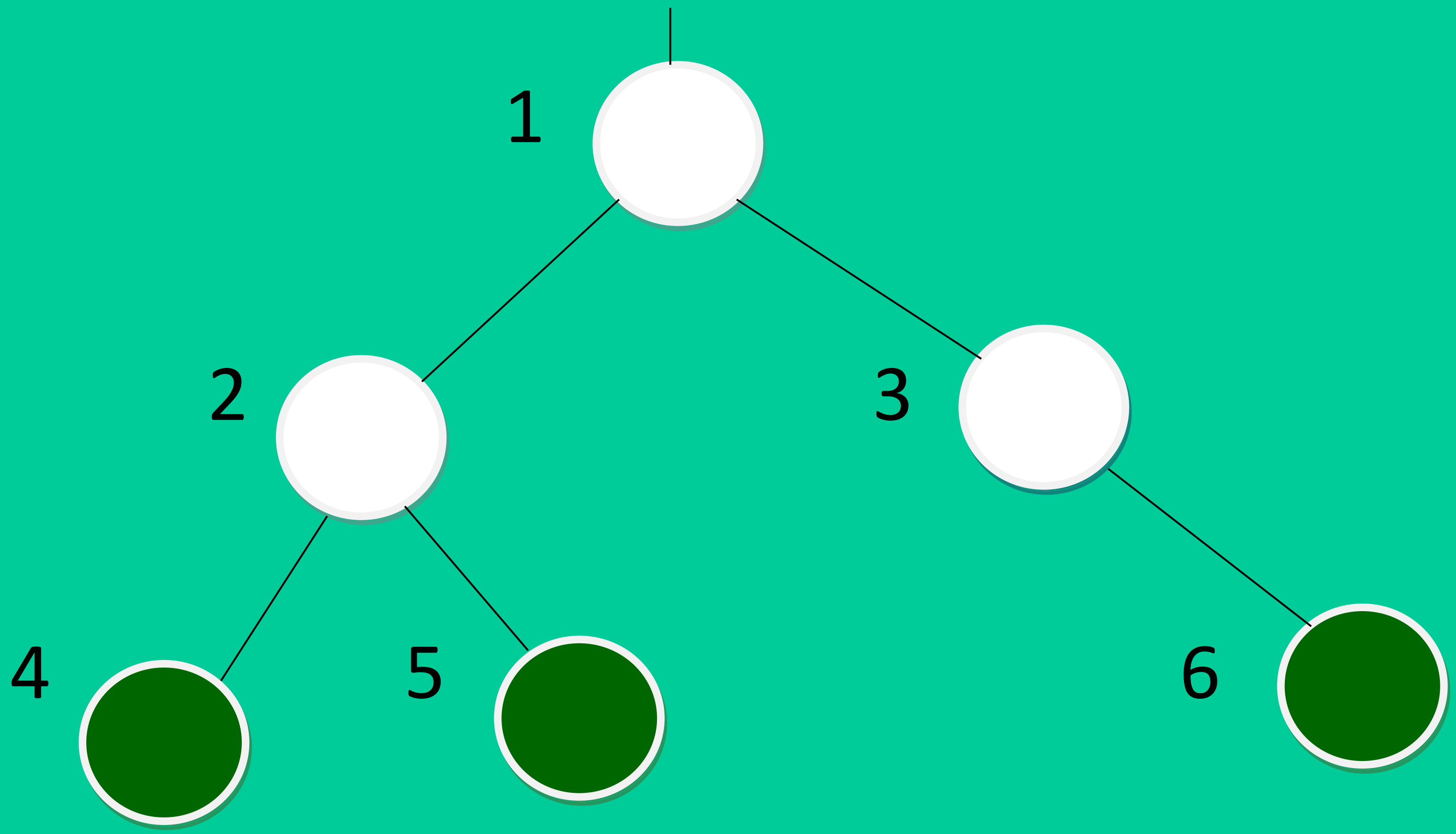
Tous les nœuds d'un arbre binaire ont **au plus deux** fils: un nœud qui a **deux** fils est appelé **nœud interne** ou **point double**.



Un nœud qui a **seulement un fils** est dit **point simple**.

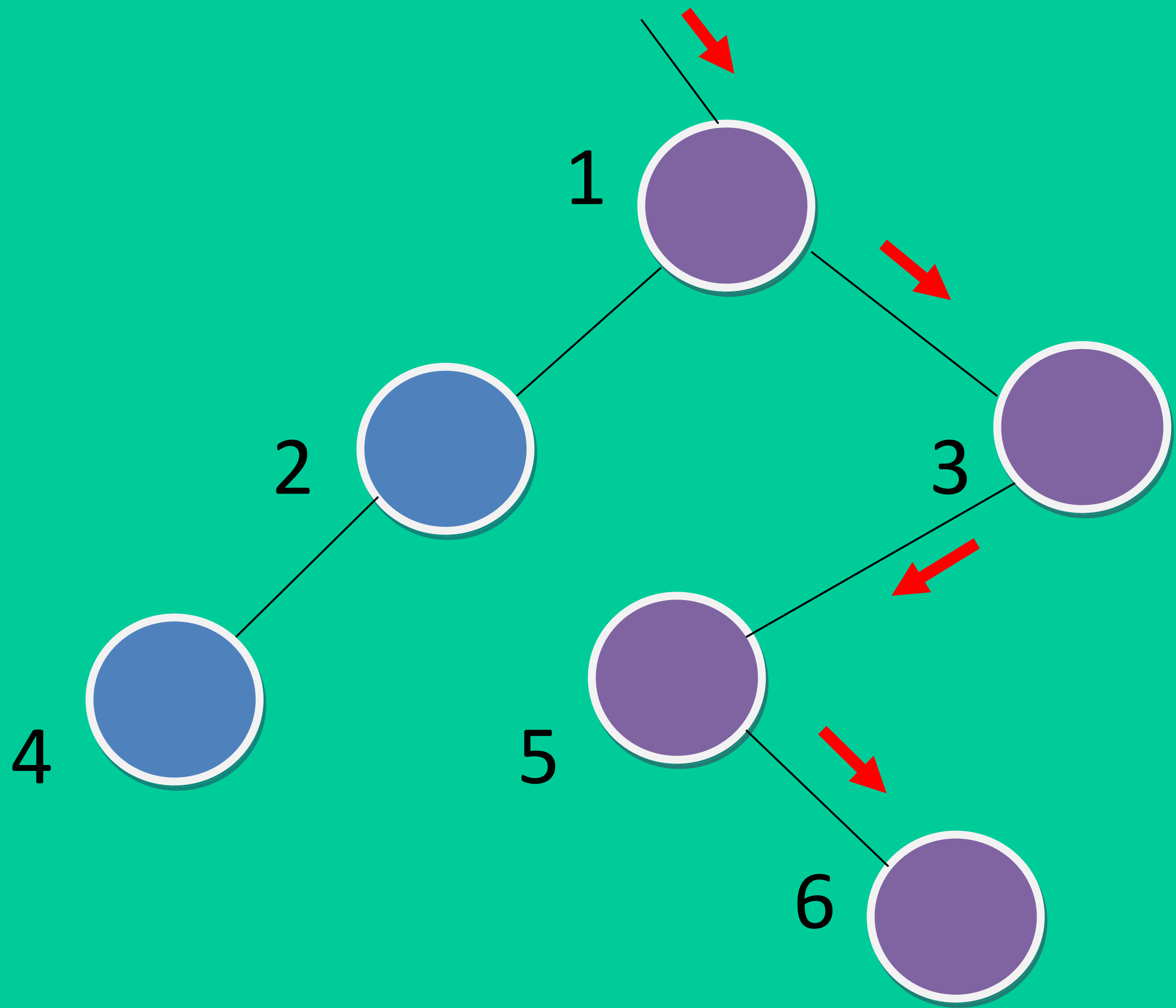


Un nœud **sans fils** est appelé **nœud externe** ou **feuille**.

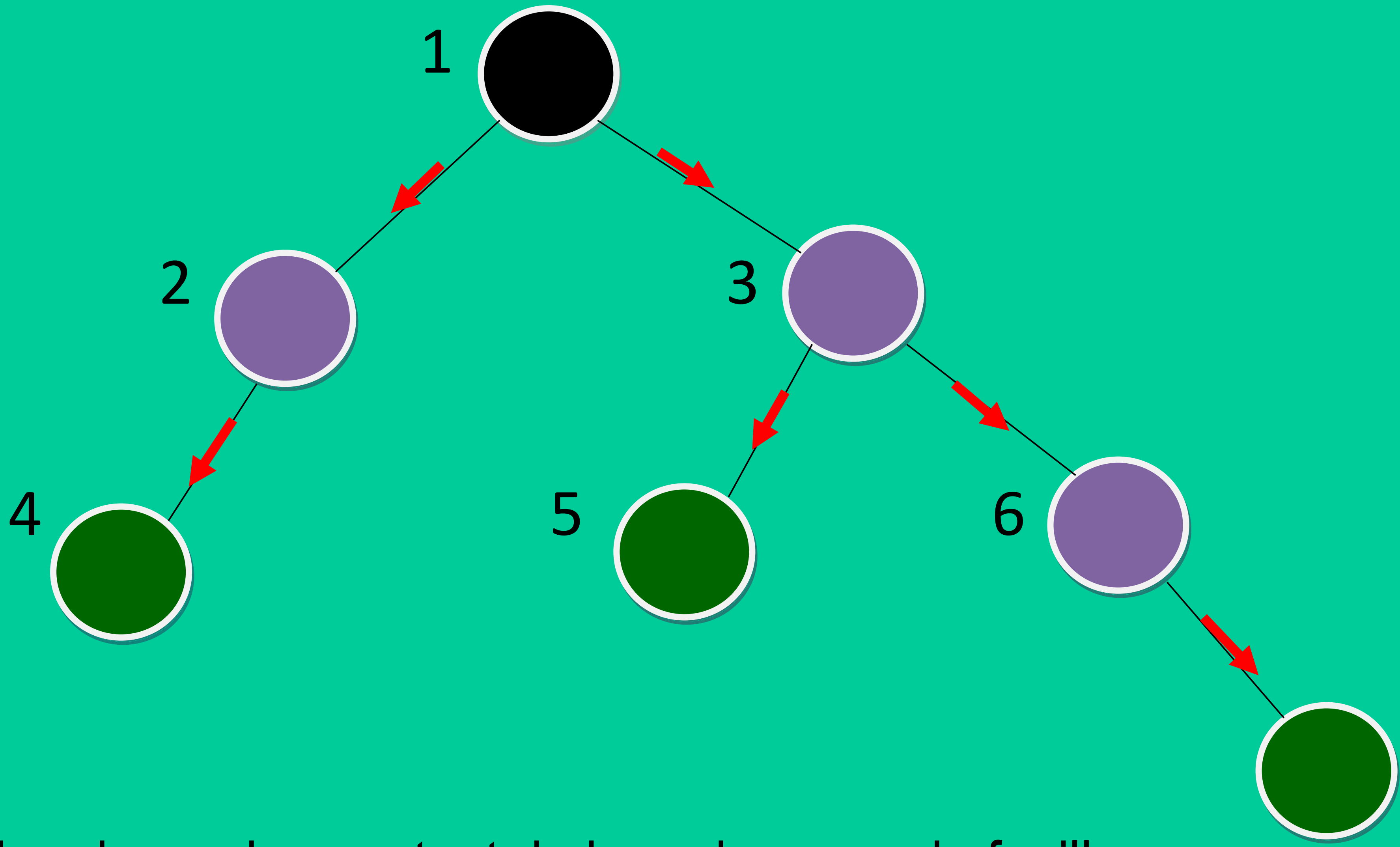


Chemin et branche d'un arbre

Un **chemin** est une **suite** de nœuds consécutifs.

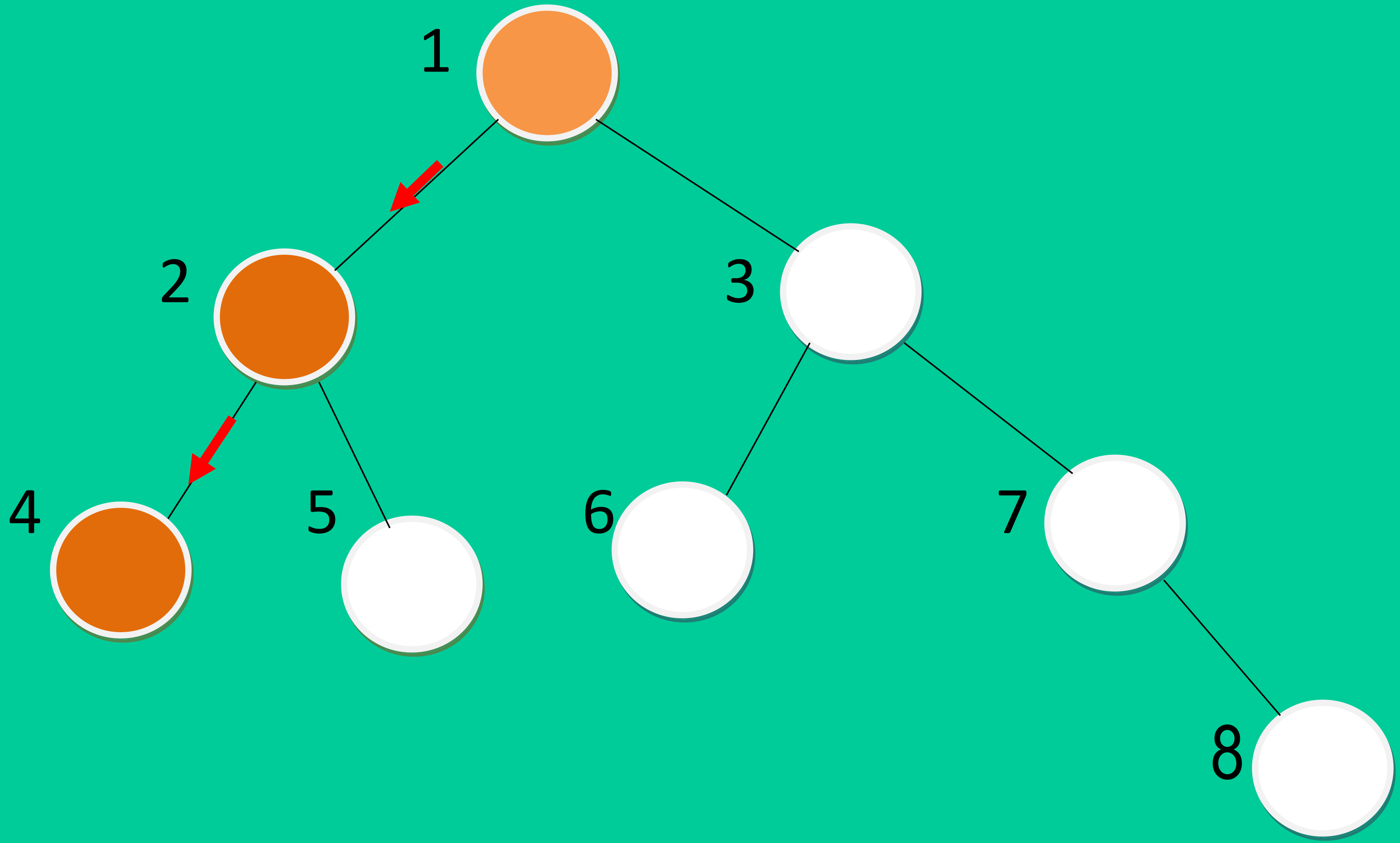


Une **branche** est un **chemin** de la **racine** à une **feuille** de B.

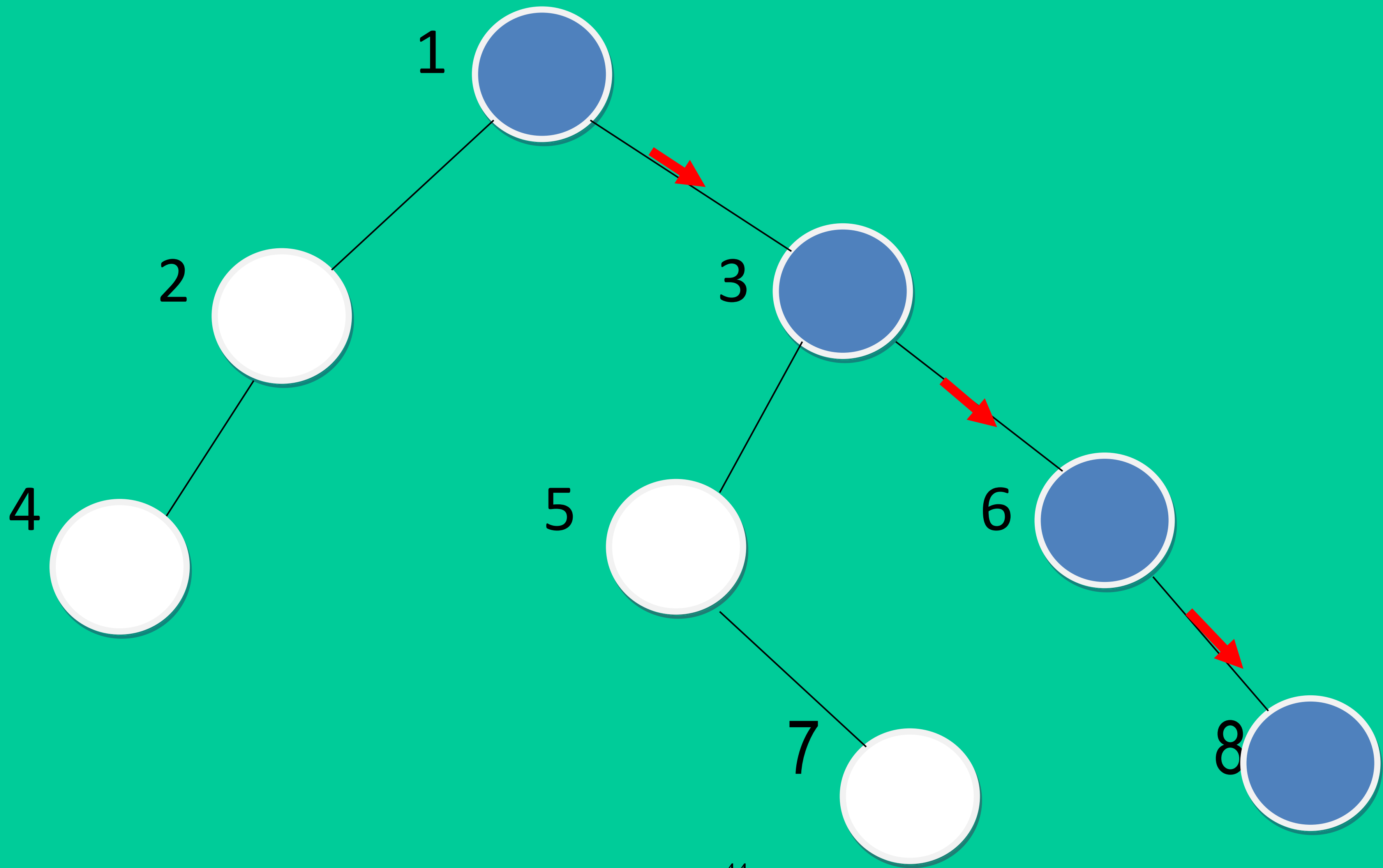


Un arbre a donc autant de branches que de feuilles.

On appelle **bord gauche** la branche partant de la **racine** en ne suivant que des fils gauches.



On appelle **bord droit** la branche partant de la racine en ne suivant que des fils droits.



II- MESURES SUR LES ARBRES

- Taille
- Niveau
- Hauteur
- Cheminement interne/externe
- Profondeur moyenne

1- Taille d'un arbre

La **taille** d'un arbre est le nombre de ses nœuds.

On définit l'opération :

taille : ARBRE $\rightarrow$ ENTIER

à l'aide des deux axiomes suivants:

taille(**arbreVide**) = 0

taille(**construire**(o, B_1,B_2))=**taille**(B_1)+**taille**(B_2)+1

2- Niveau d'un noeud

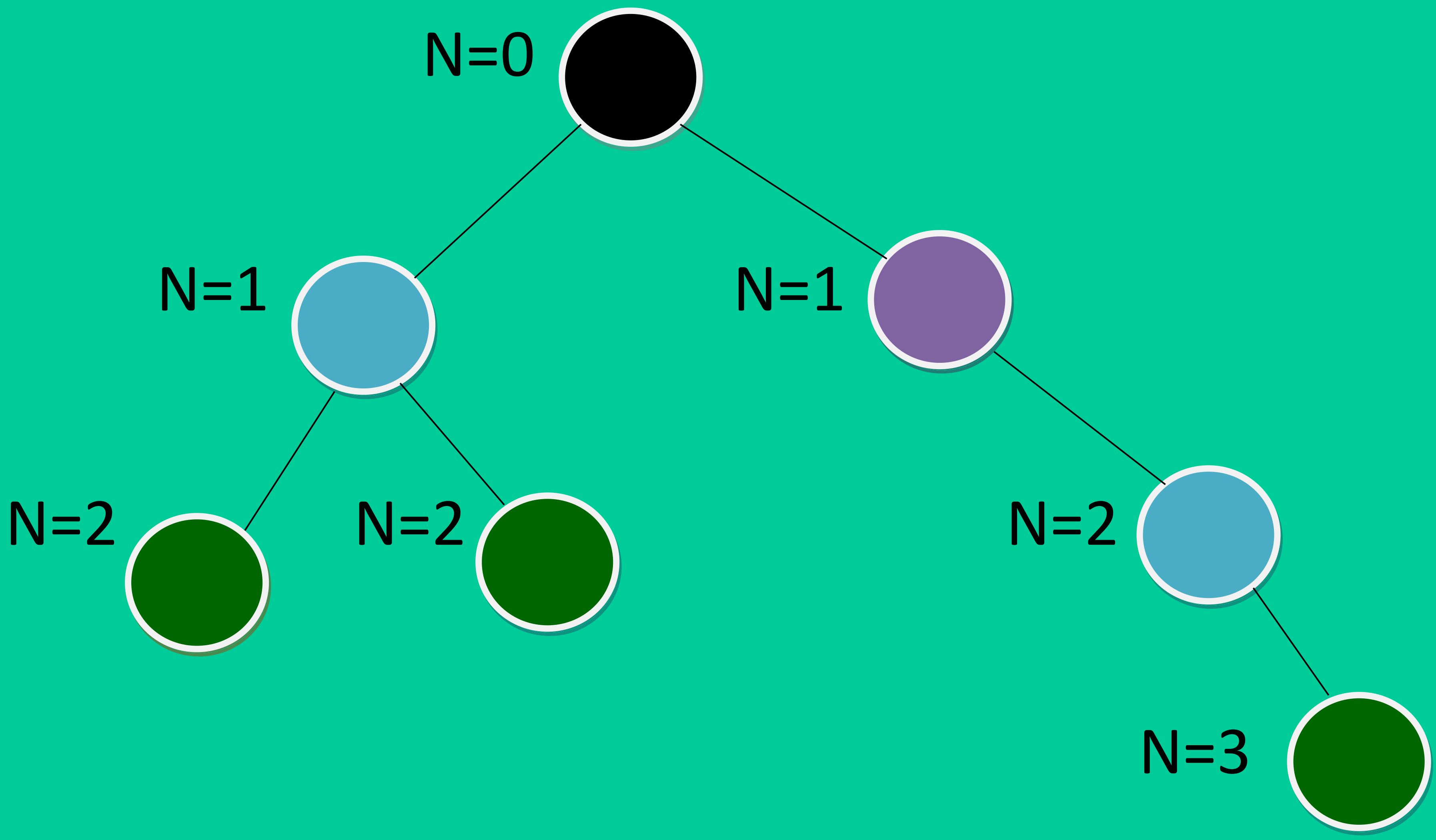
Le **niveau** d'un nœud **x** d'un arbre B est une fonction définie récursivement de la façon suivante :

niveau : NŒUD $\rightarrow$ ENTIER

avec les axiomes suivants:

niveau(racine(B)) = 0

$\forall$x : nœud de B • **niveau**(x) = **niveau**(**pere**(x))+1



N mesure le niveau du nœud.

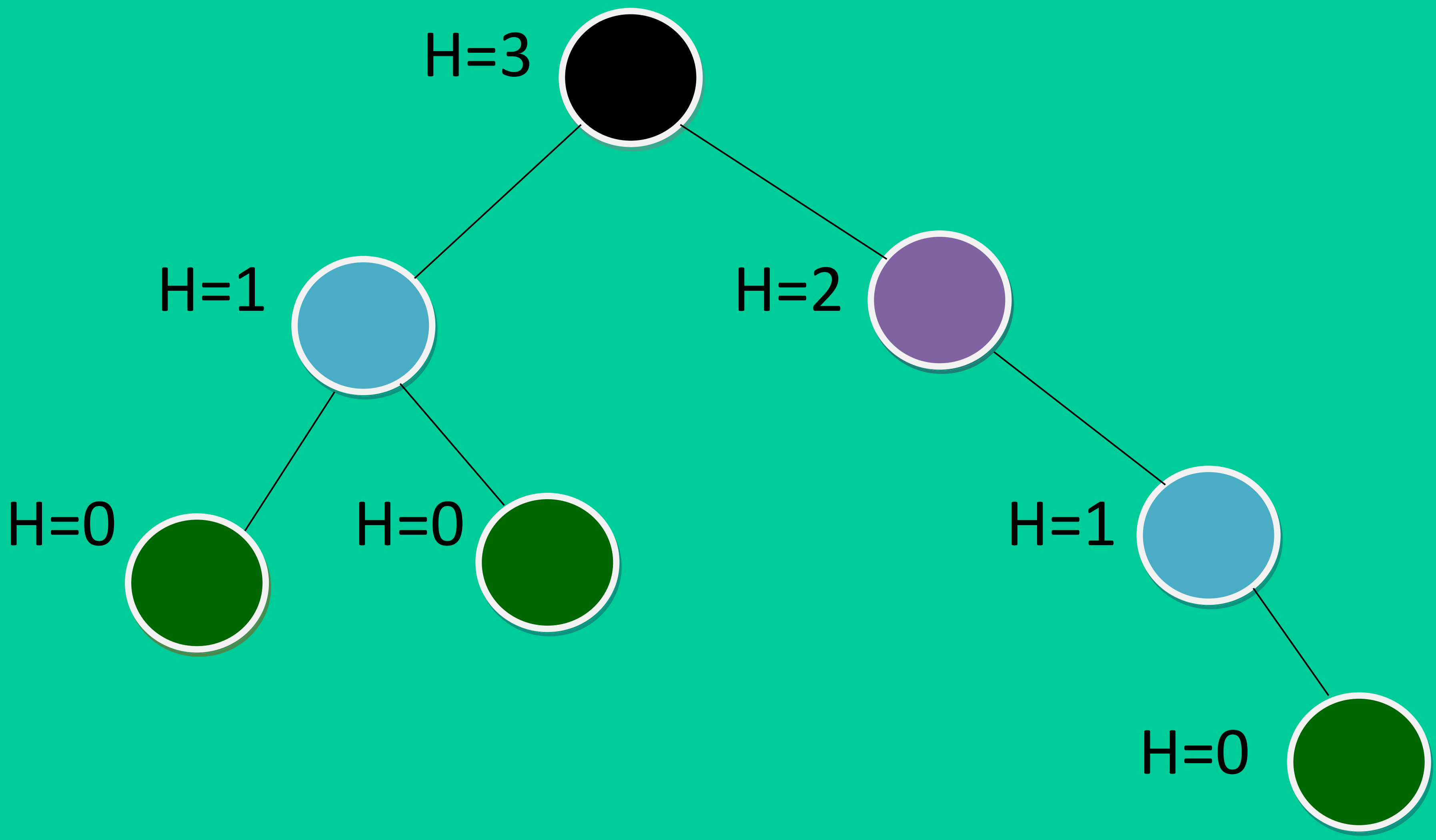
Hauteur d'un arbre

La **hauteur** d'un arbre B, notée h(B), est définie comme suit:

h(**arbreVide**) = **-1**
h(A) = **1** + Max [h(**gauche**(A)), h(**droit**(A))]

Par abus de langage, on note :

h(A) = h(**racine**(A))



H mesure la hauteur d'un nœud

Bornes optimales

Soit un arbre binaire, non vide, de hauteur **H** et de taille **n**, on a :

$$[\log_2 \mathbf{n}] \leq H \leq \mathbf{n-1}.$$

Relation importante lorsque la **complexité** de l'algorithme est en **O**(H).

III- PARCOURS D'UN ARBRE BINAIRE

Parcourir un arbre consiste à atteindre:

- systématiquement,
- dans un certain **ordre**,

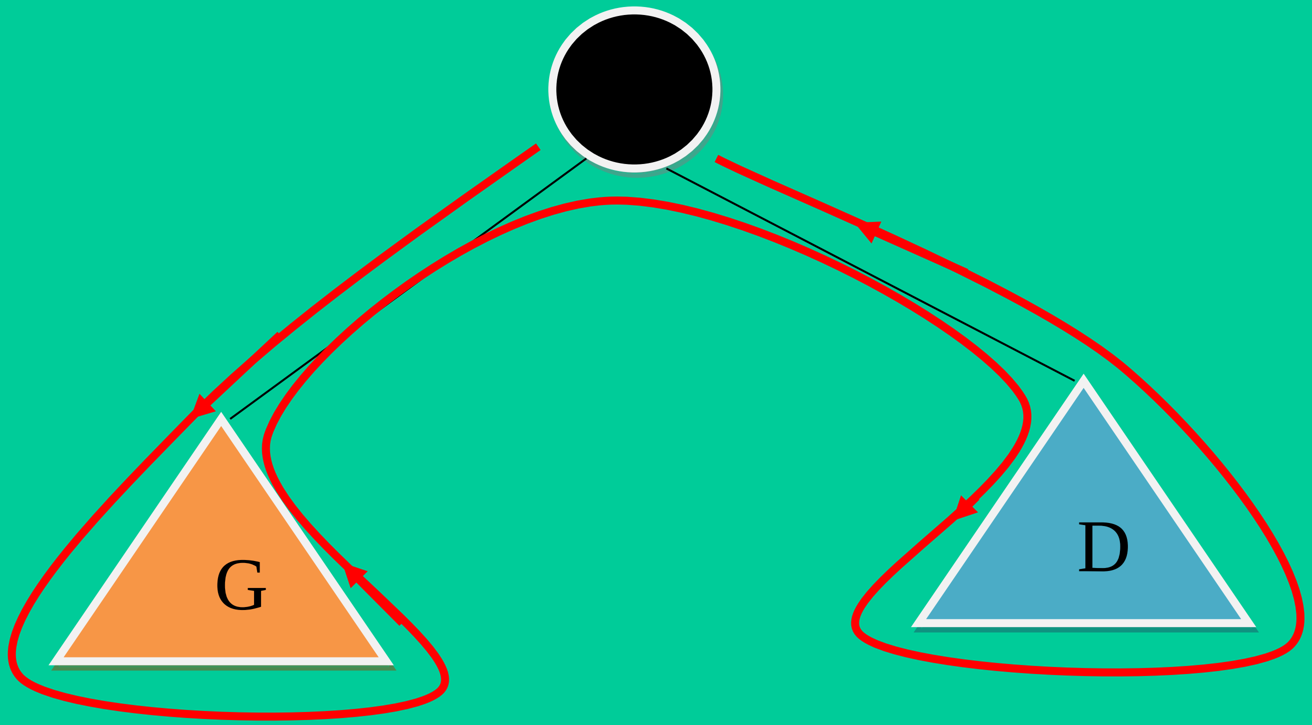
chacun des nœuds de l'arbre pour y effectuer le même traitement.

1 – Parcours en profondeur

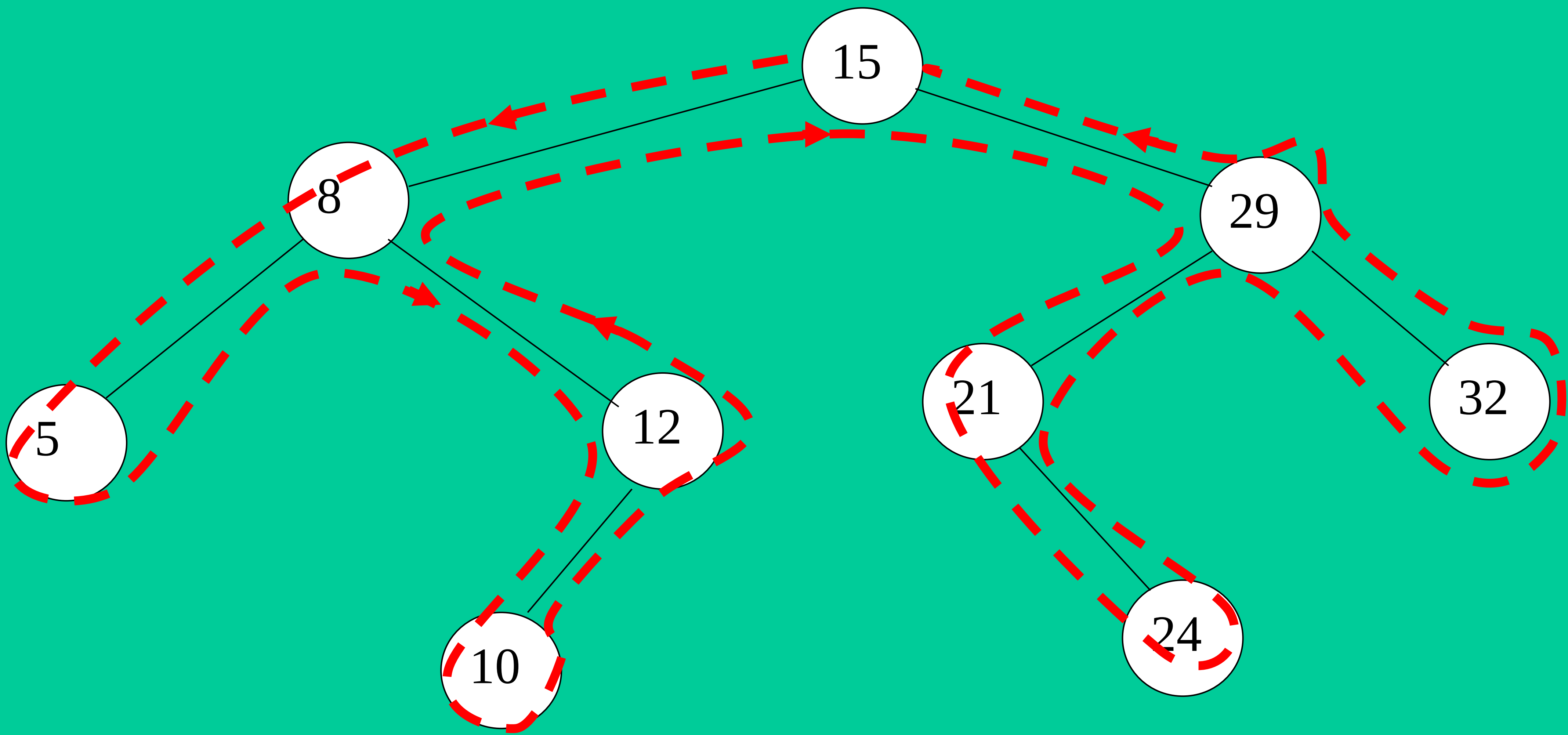
L'opération consiste à tourner autour de l'arbre en suivant le chemin qui:

- part à gauche de la racine,
- va toujours le plus à gauche possible en suivant l'arbre.

Illustration du principe de parcours

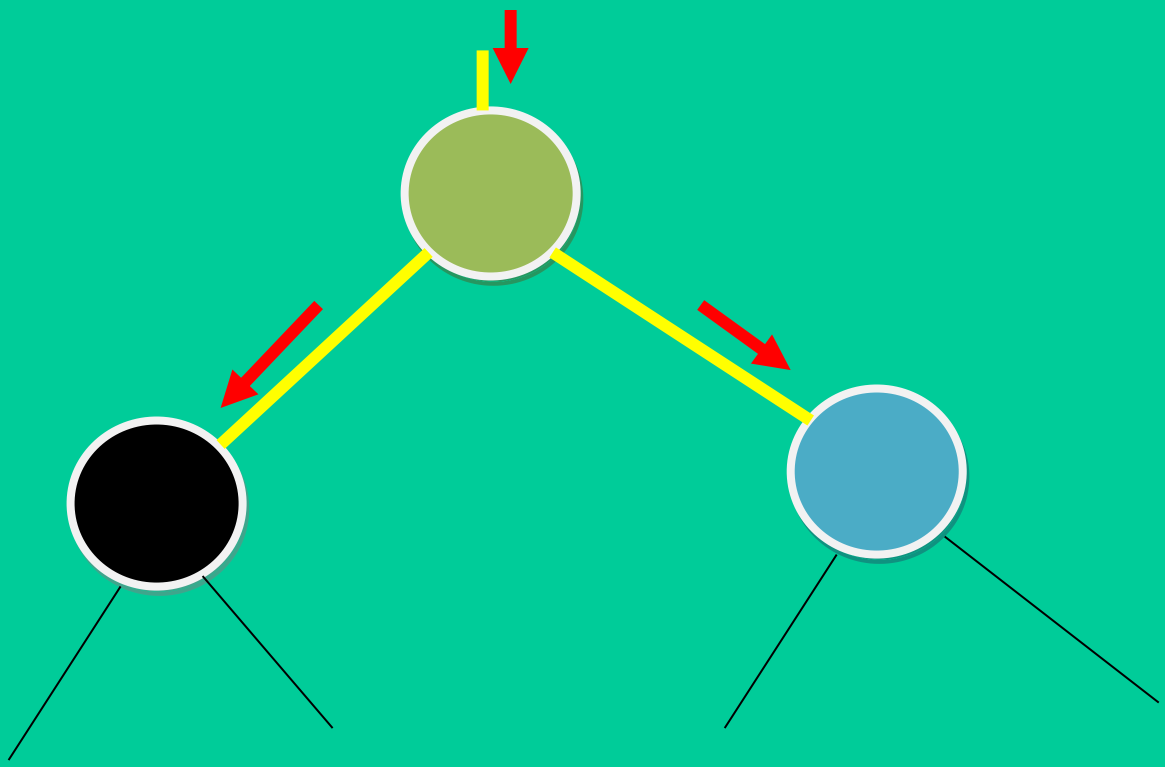


Exemple de parcours en profondeur



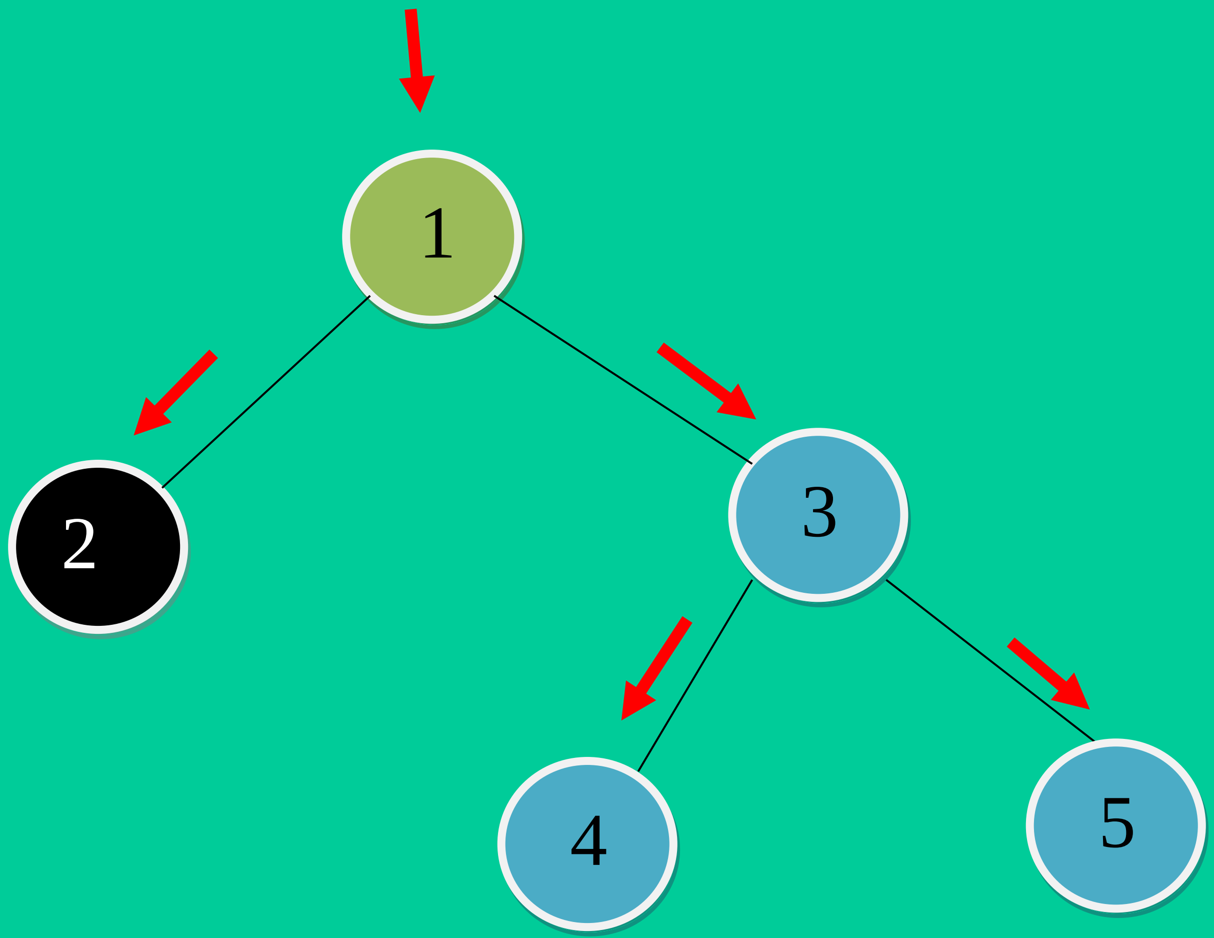
Dans ce parcours, chaque nœud de l'arbre est rencontré **trois** fois:

a)- d'abord à la **«descente»**,

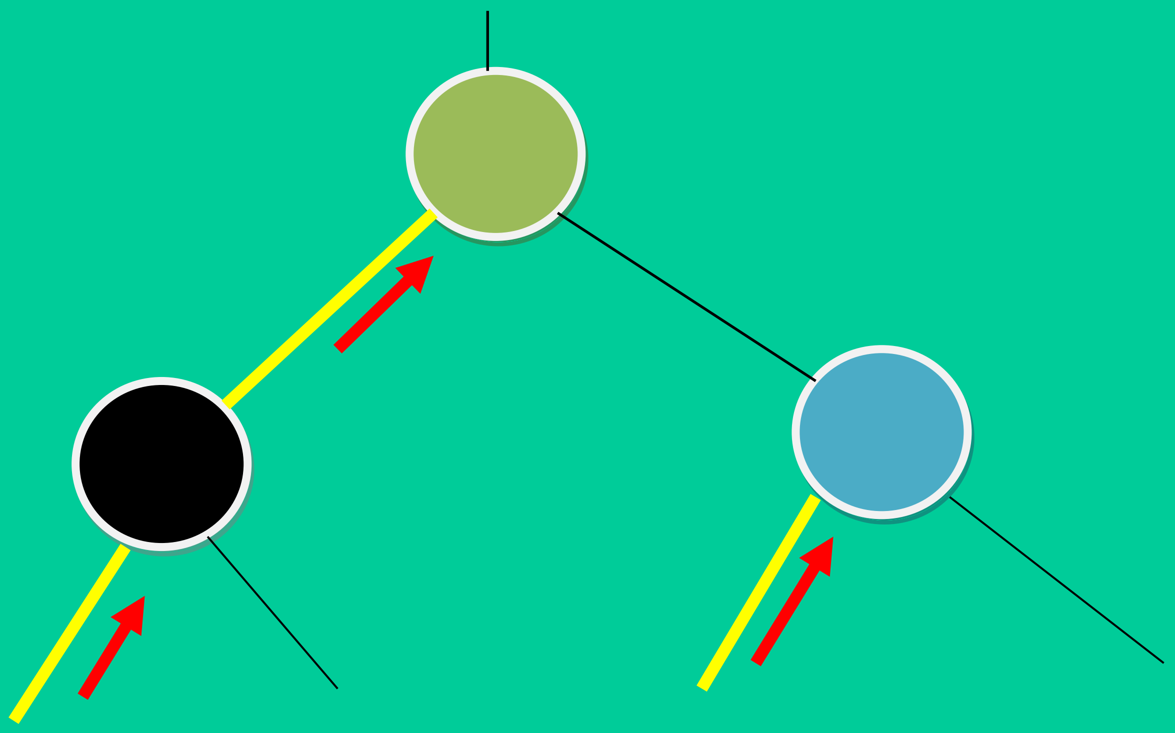


Il induit un parcours de l'arbre en **ordre préfixe**.

Exemple d'ordre préfixe : 1,2,3,4,5

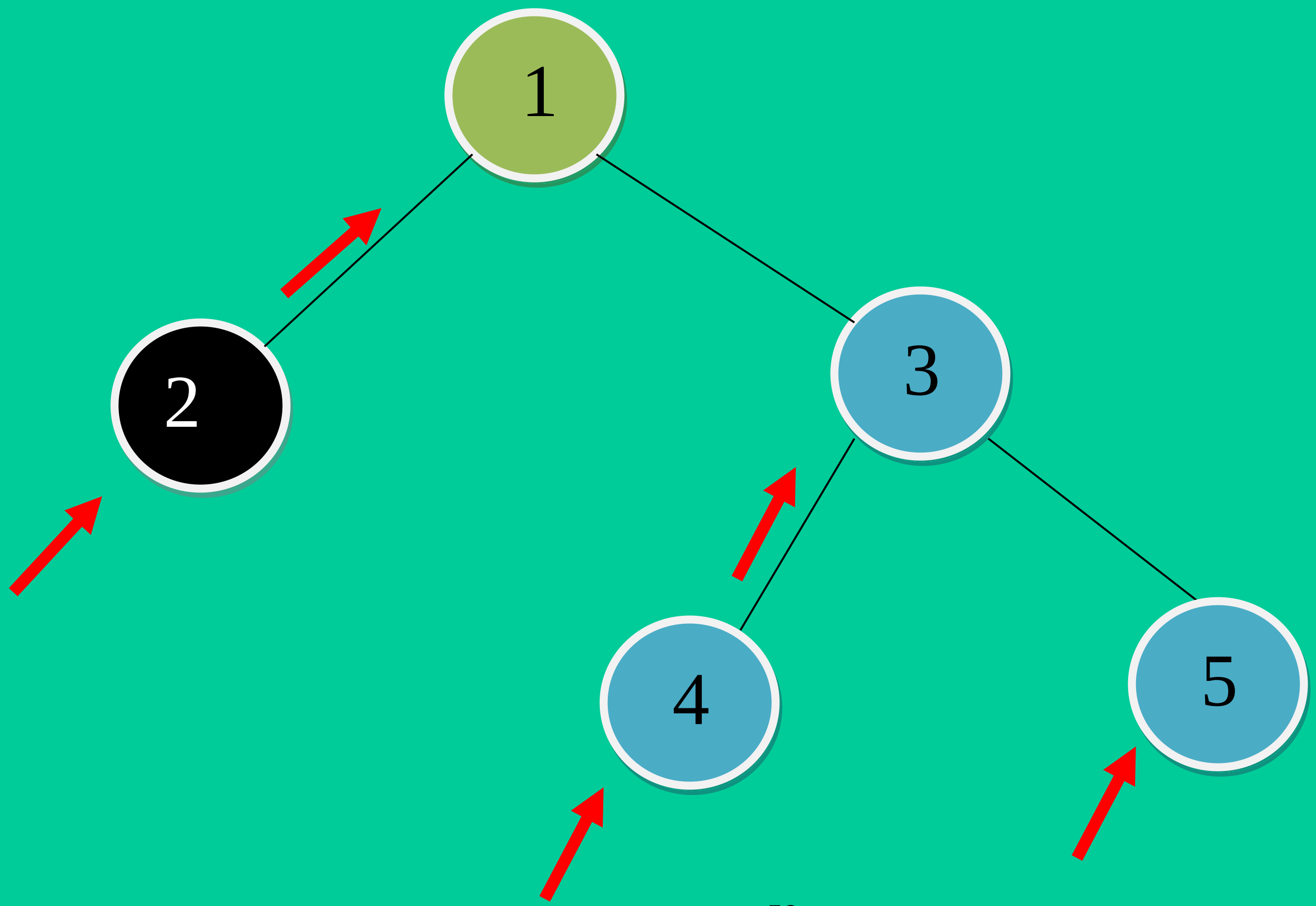


b)- puis en « **montée gauche**»: après le parcours de son sous arbre gauche,

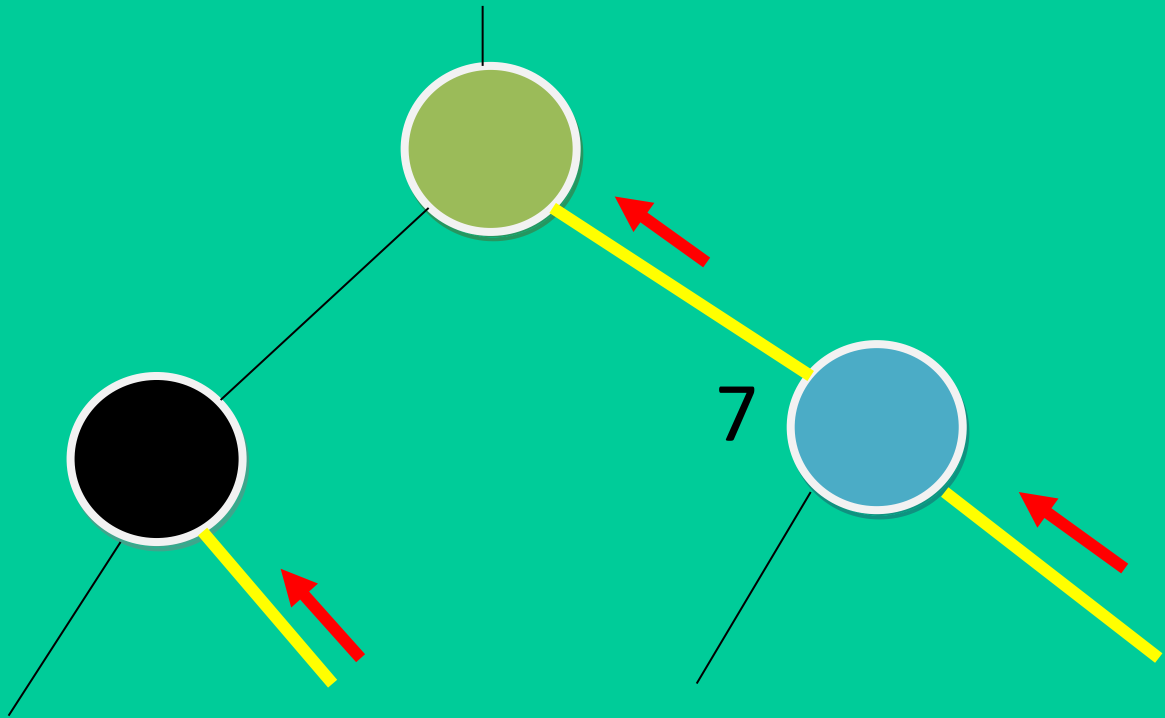


Il induit un parcours de l'arbre en **ordre infixe**.

Exemple d'ordre infixe : 2,1,4,3,5

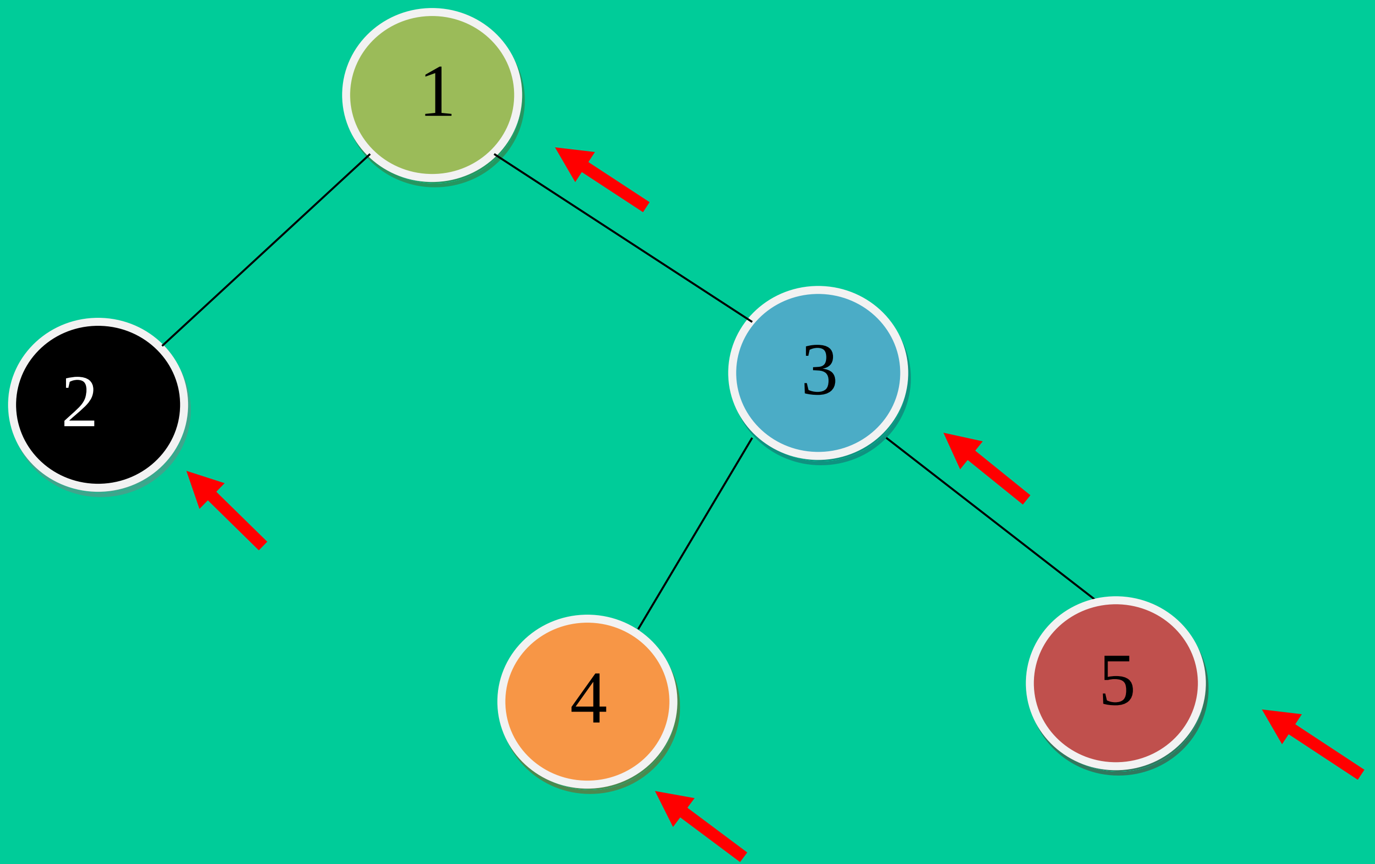


c)- et enfin en «**montée droite**»: après le parcours de son sous arbre droit.



Il induit un parcours de l'arbre en **ordre suffixe** .

Exemple d'ordre suffixe : 2,4,5,3,1



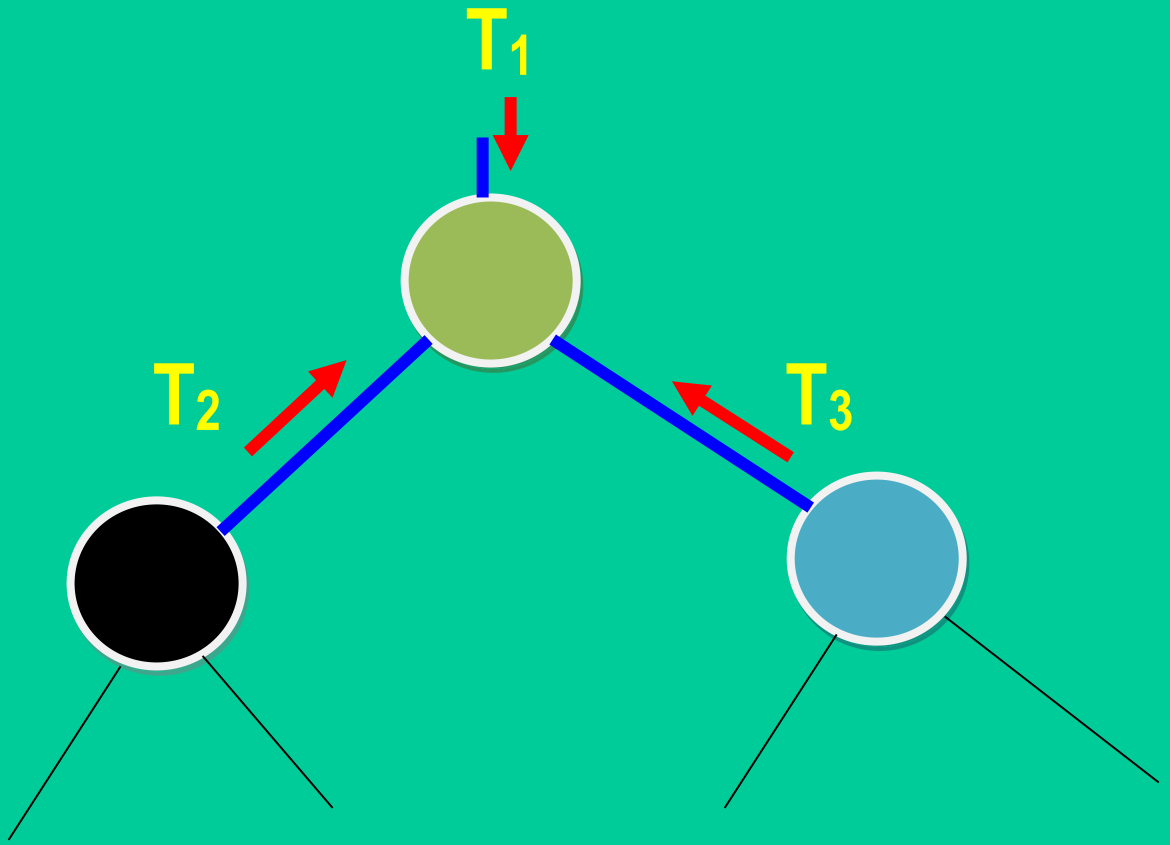
2 – Algorithme de parcours

On peut faire correspondre, en général, à chacune des **3** rencontres du nœud un traitement différent.

Notons T_1, T_2 et T_3 ces traitements.

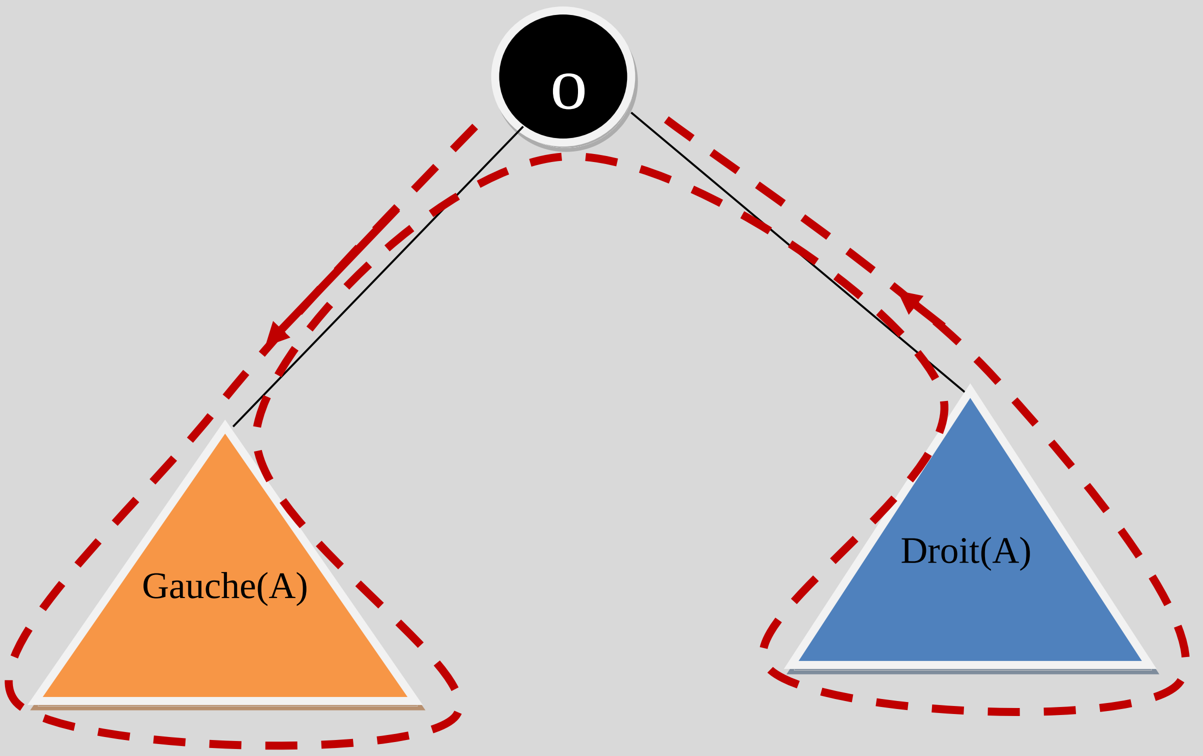
- T_1 : traitement en ordre **préfixe**,
- T_2 : traitement en ordre **infixe**
- T_3 : traitement en ordre **suffixe**

Par ailleurs, envisageons pour l'arbre vide un traitement spécial noté T_0.



Les 3 traitements T_1 , T_2 , et T_3

Illustration de l'algorithme de parcours



Parcours de l'arbre A

Procédure de parcours

```
profondeur(ARBRE A)
/* Parcours en profondeur « main gauche» d'un arbre binaire A */
    {
    if( A = = arbreVide() )
            T0 ( ) ; /* traitement spécial d'un arbre vide */
    else
                {
                T1( ) ; /* traitement en ordre préfixe */
                profondeur (gauche(A)) ;
                T2 () ; /* traitement en ordre infixe */
                profondeur (droit(A)) ;
                T3 () ; /* traitement en ordre suffixe */
                }
    }
```

IV-Arbre Binaire de Recherche (ABR)

Un arbre binaire **B** est un **Arbre Binaire de Recherche** (ABR) si:

- soit **B** est vide,
- soit **B** est non vide et, si **v** est la valeur associée à sa racine, on a:
  - toute valeur **x** associée à son sous arbre **gauche** est telle que:

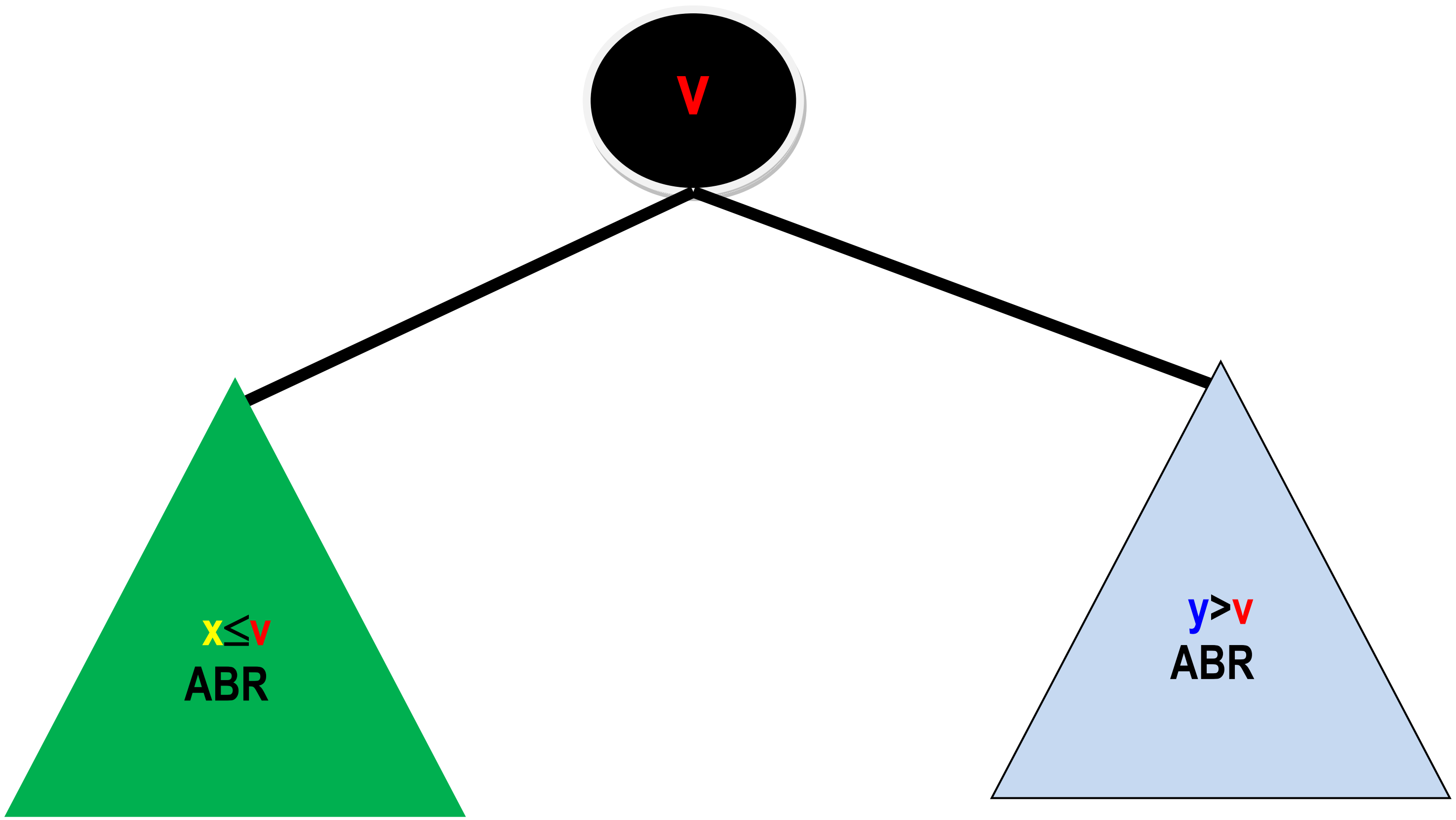
    $$x \leq v$$

  - toute valeur y associée à son sous arbre **droit** est telle que:

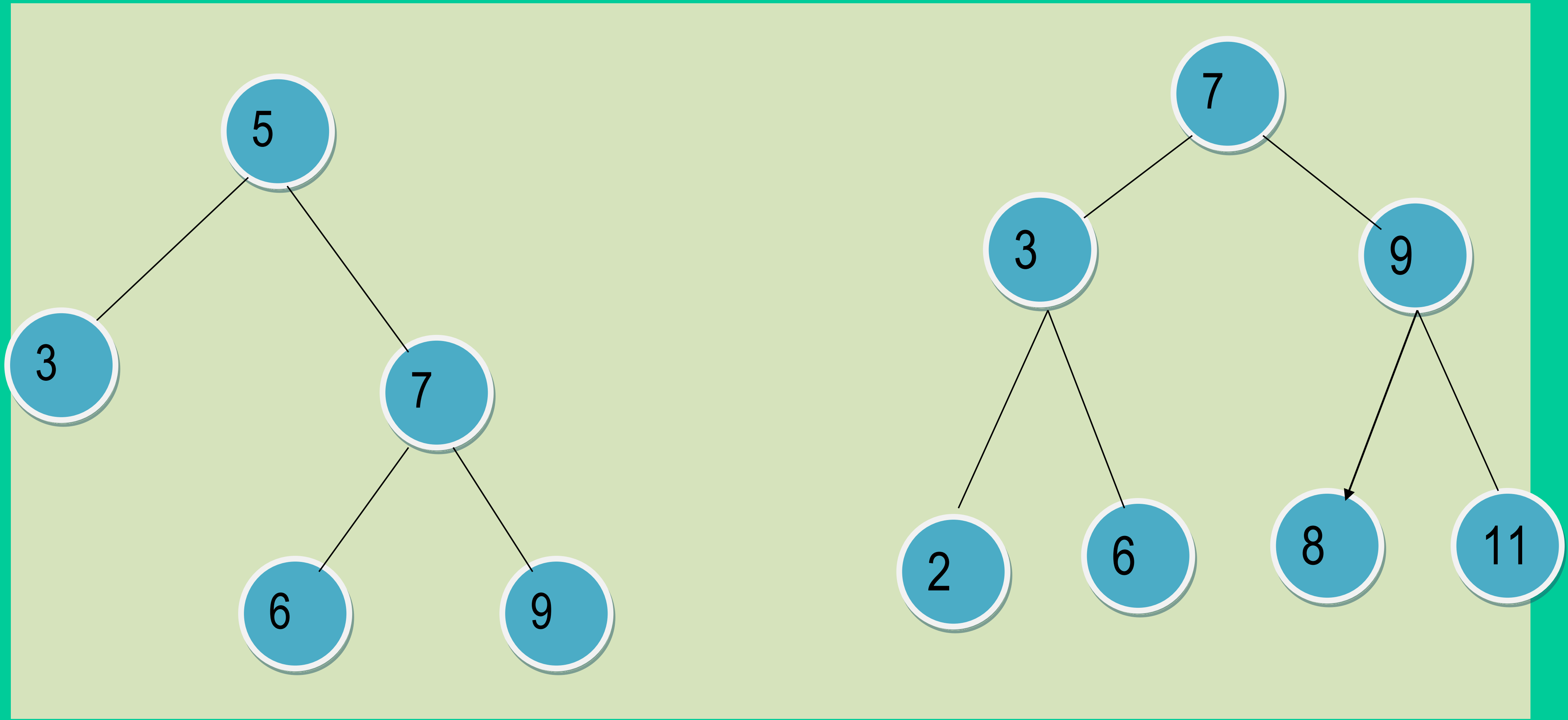
    $$y > v$$

  - tout sous arbre de **B** est un ABR.

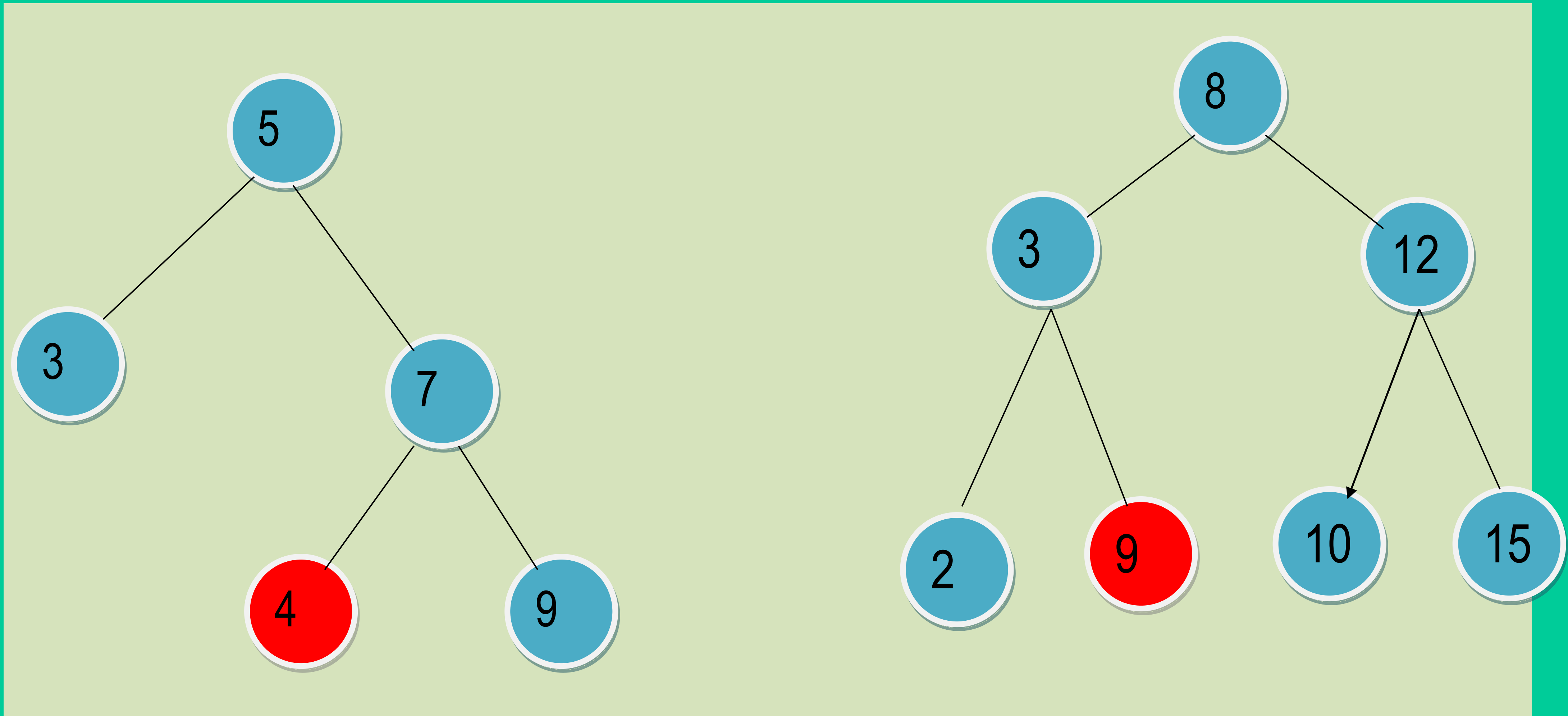
Illustration simple d’un ABR



Exemples d'arbres ABR



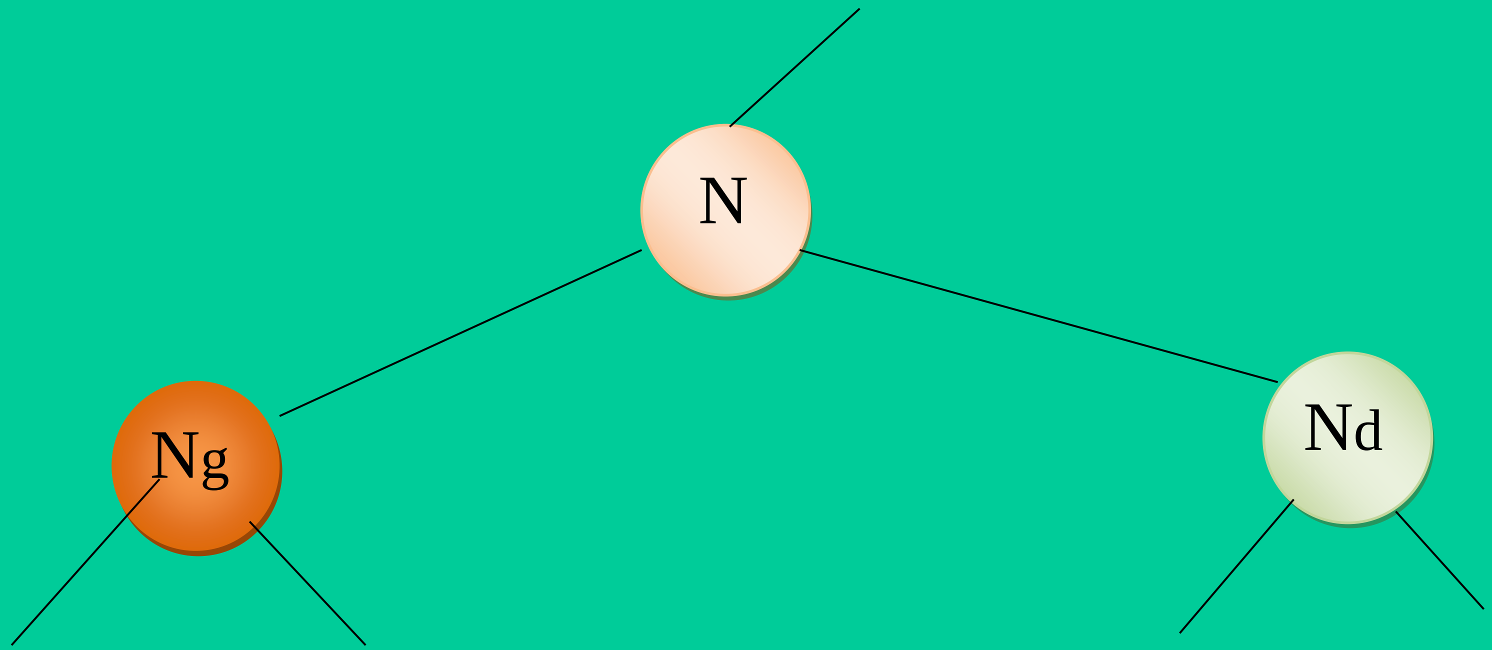
Exemples d'arbres non ABR



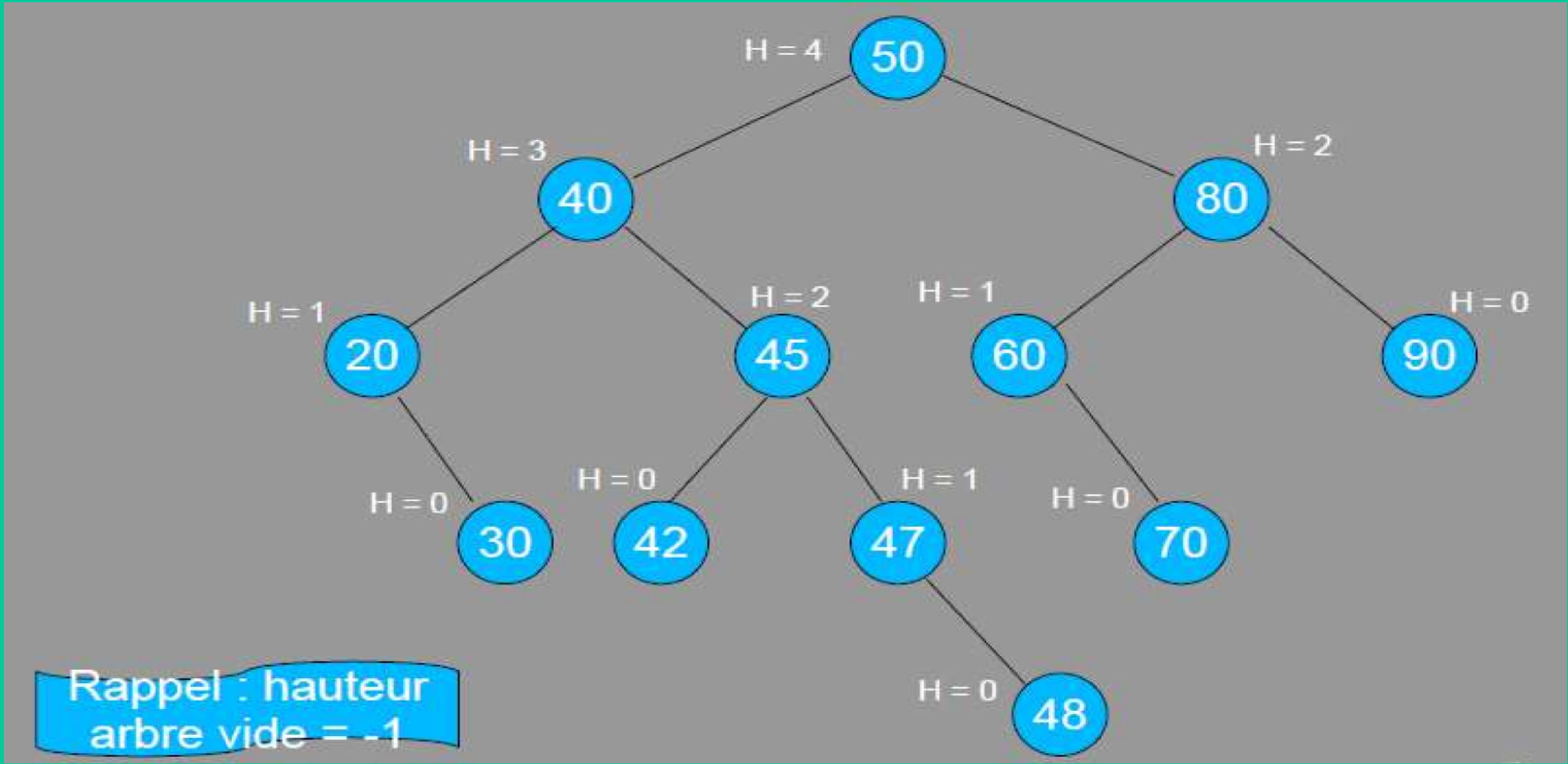
V- Les arbres H-équilibrés ou AVL

Les arbres AVL sont tels que pour tout nœud N, on a :

$$\forall N \in S \quad \bullet \quad | H(N_g) - H(N_d) | \leq 1$$



Exemple d'arbre AVL



Propriété 1 : sur la hauteur.

Soit A un arbre AVL ayant n sommets et de hauteur H.

Alors

$$\log_2(1+n) \leq 1+H \leq \alpha \log_2(2+n)$$

avec $\alpha \leq 1,44$.

Pour donner une idée de la relation entre la taille n et hauteur H d'un arbre AVL :

$n = 10^5$ alors $17 \leq H \leq 25$.

Propriété 2 : sur la taille

Soit $t_m(H)$ la taille minimale d'un arbre AVL de hauteur H.

Alors

$$t_m(H) \quad = \quad 1 + t_m\,(H\text{-}1) + t_m\,(H\text{-}2).$$

Constats sur la hauteur

-La **hauteur** d'un arbre est une **métrique** très importante: elle est un indice de **performance** d'un programme**.**

-Plus l'arbre aura une **hauteur élevée**, plus l'algorithme mettra de **temps à s'exécuter**.

Quel est le problème ?

Comment **maintenir** un arbre relativement équilibré au fur et à mesure des **insertions** et **suppression** ?

Une solution pourrait être de passer par un **équilibrage** de l'arbre à chaque insertion, à l'aide des **rotations**.

Ces rotations sont des **transformations de base** centrées sur un nœud.

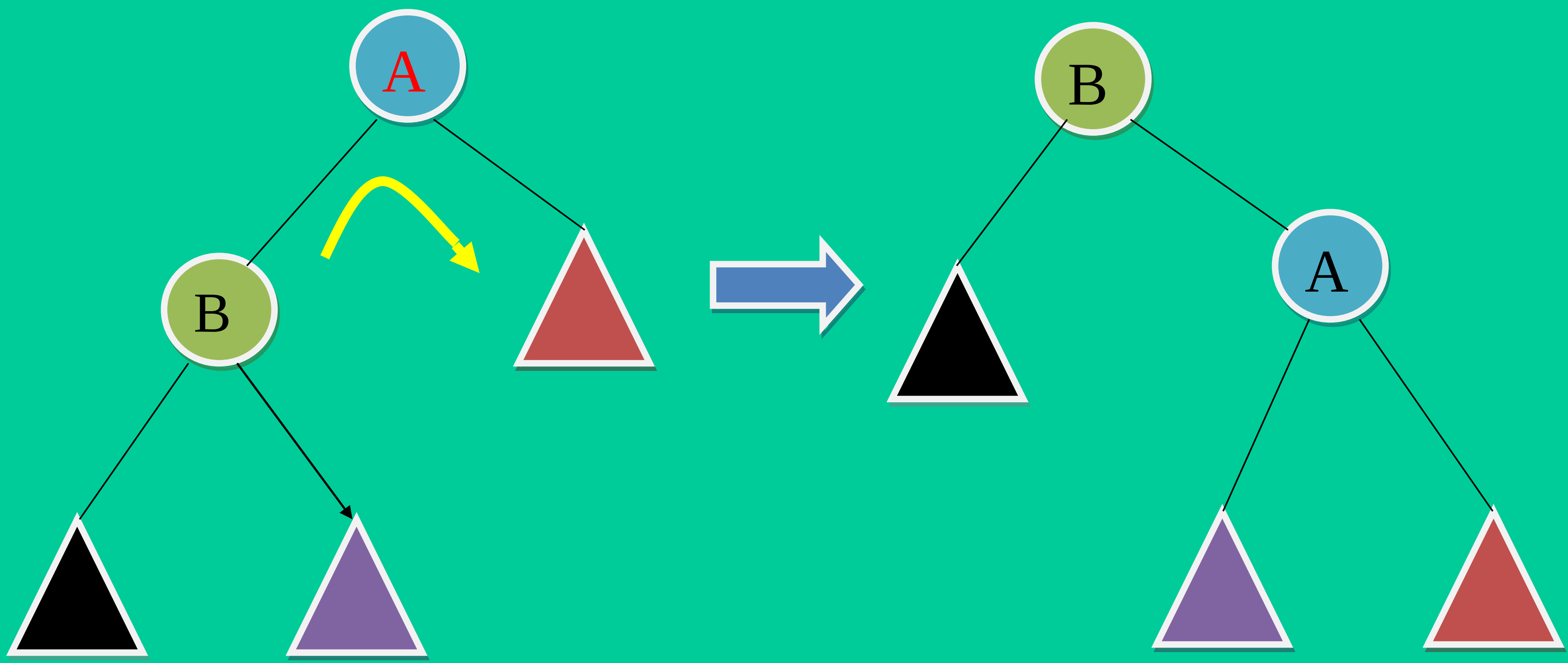
Il existe deux sortes de rotations :

-rotation gauche, notée **rg( )**

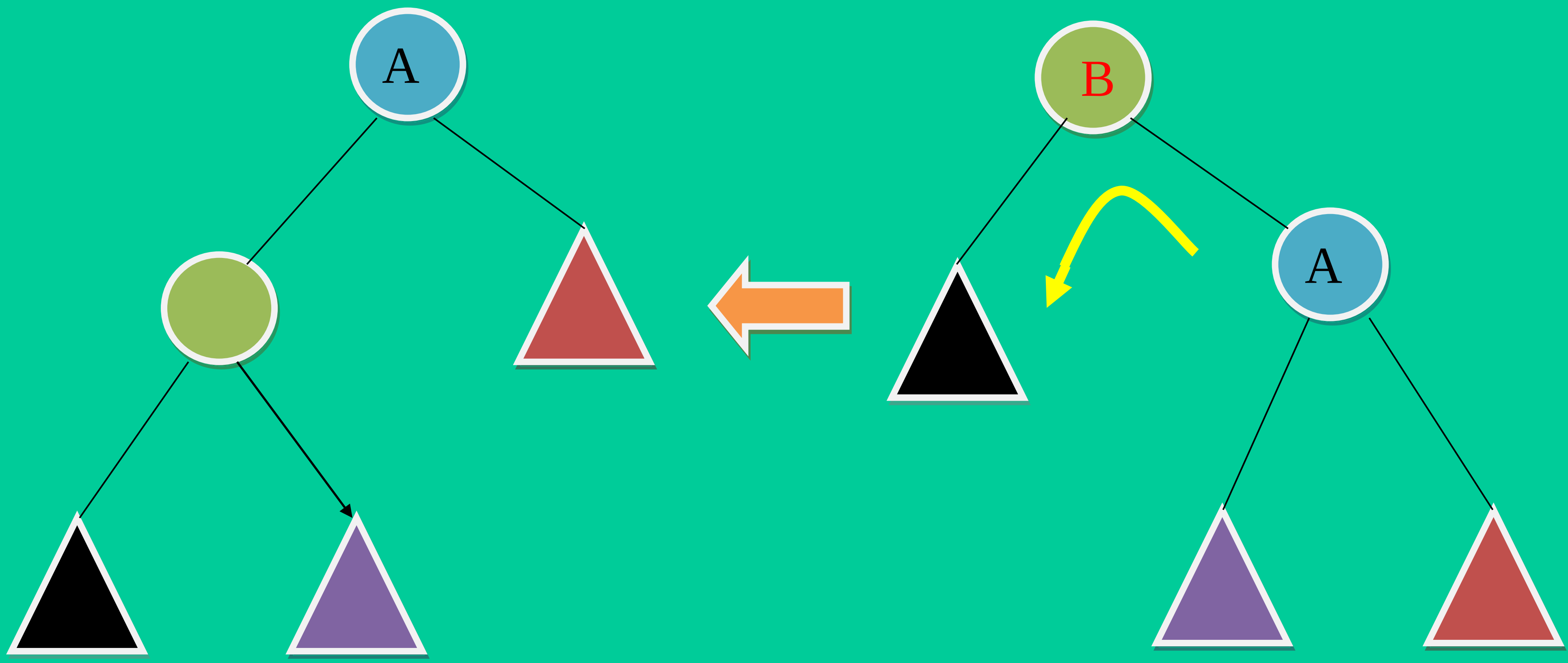
-rotation droite, notée **rd( )**

Ces deux rotations sont symétriques.

Rotation à droite centrée sur A notée rd(A)



Rotation à gauche centrée sur B notée rg(B)



VII-Arbres rouge-noir

Inventés par Bayer en 1972.

Étudiés en détail par Guibas et Sedgewick en 1978.

Un arbre binaire de recherche est «rouge-noir» s’il vérifie les 5 propriétés suivantes :

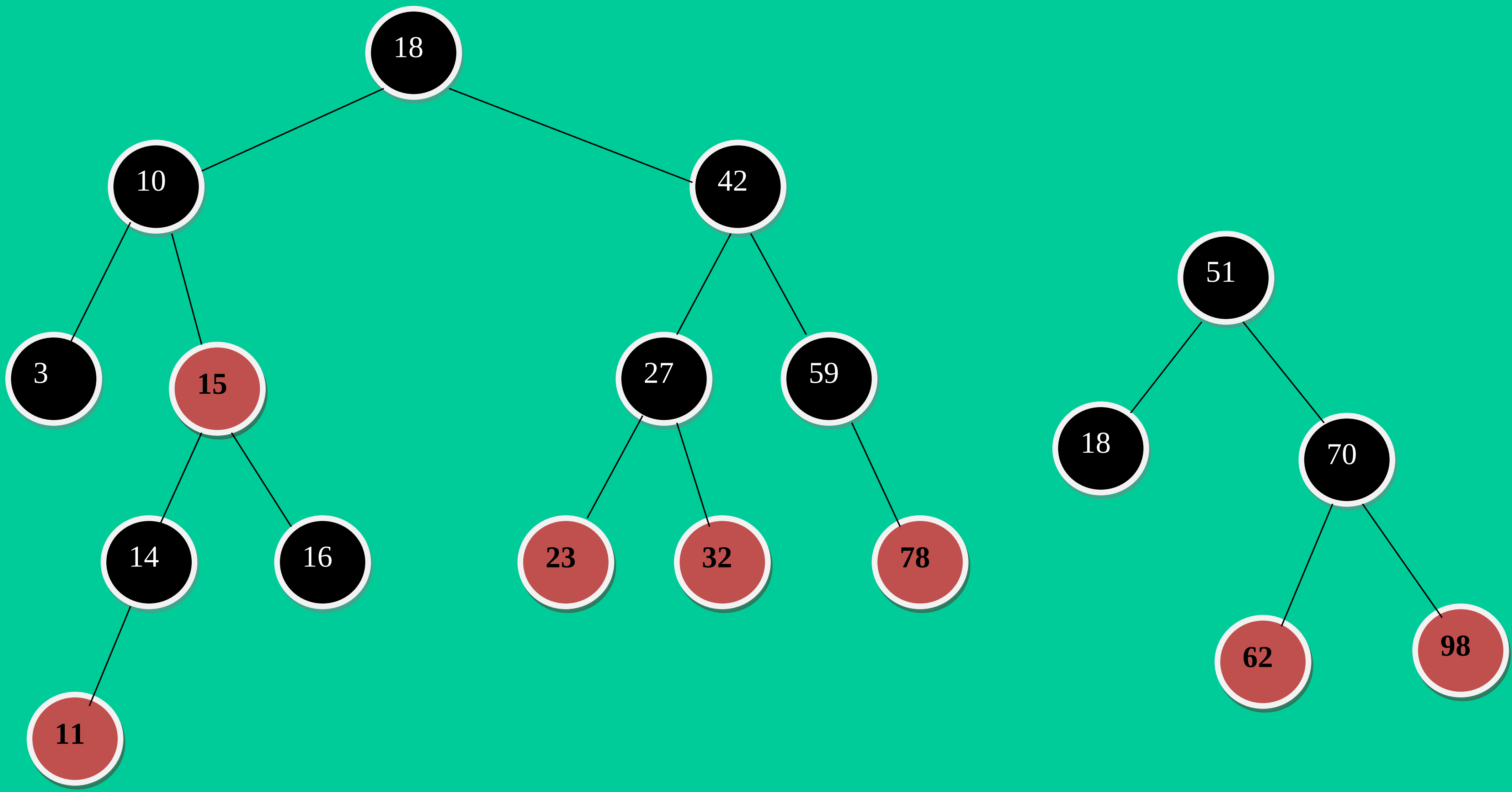
1. chaque nœud est soit rouge, soit noir;

2. la racine est noire ;

3. chaque sous-arbre vide est noir ;

4. si un nœud est rouge, alors ses deux fils sont noirs;

5. tous les chemins reliant un nœud à une feuille contiennent le même nombre de nœuds noirs.


18
10
42
3
15
27
59
14
16
23
32
78
11
51
18
70
62
98

Remarques importantes

Pour de nombreuses applications une solution plus **systématique** consiste à passer par des arbres:

- **H-équilibrés** ou arbres AVL(**A**delson-**V**elsky et **L**andis):

$$\mathbf{H} \leq \log_2(n)$$

- ou les arbres **rouge-noir** :

$$\mathbf{H} \leq 2 \times \log_2(n + 1).$$

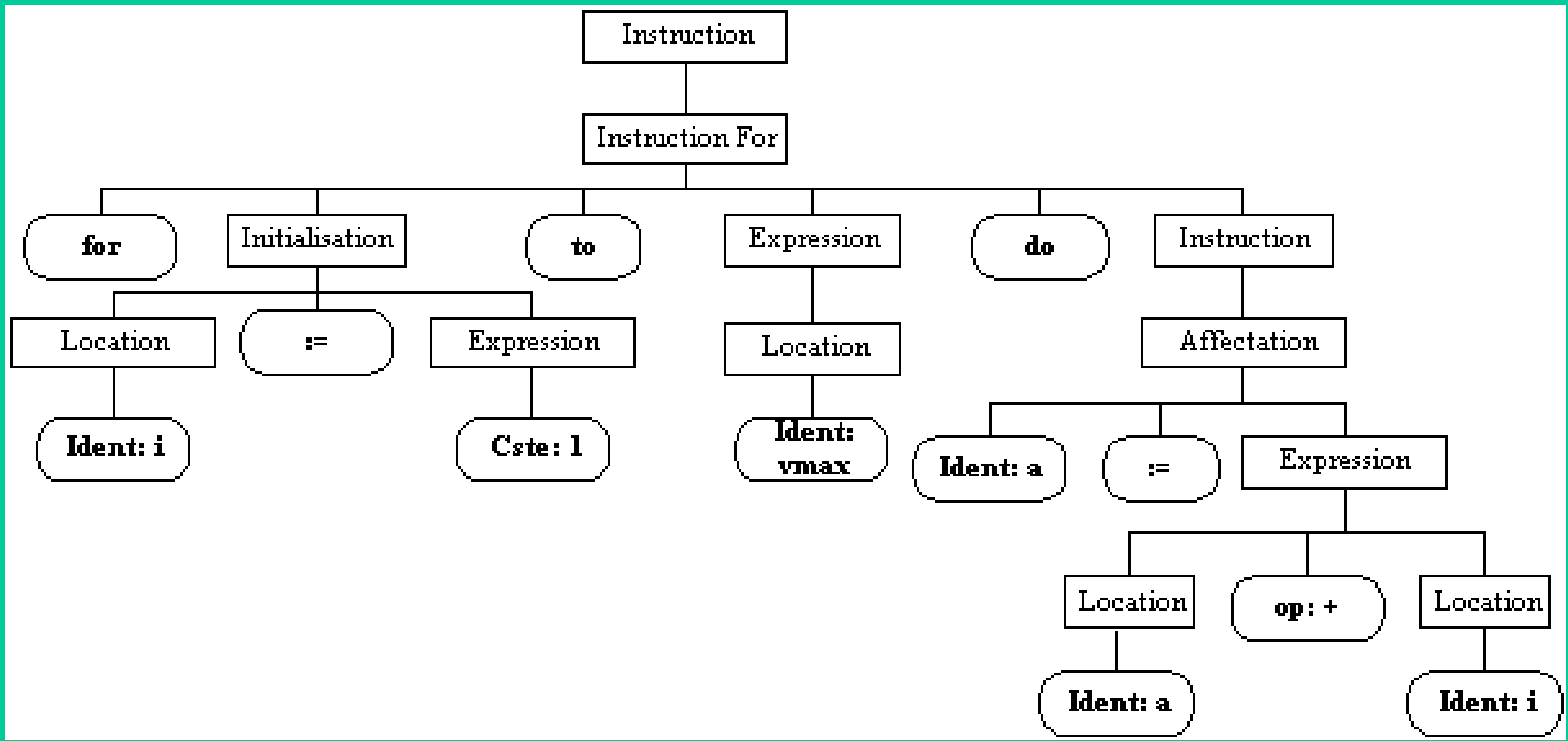
V- ARBRE GENERAL

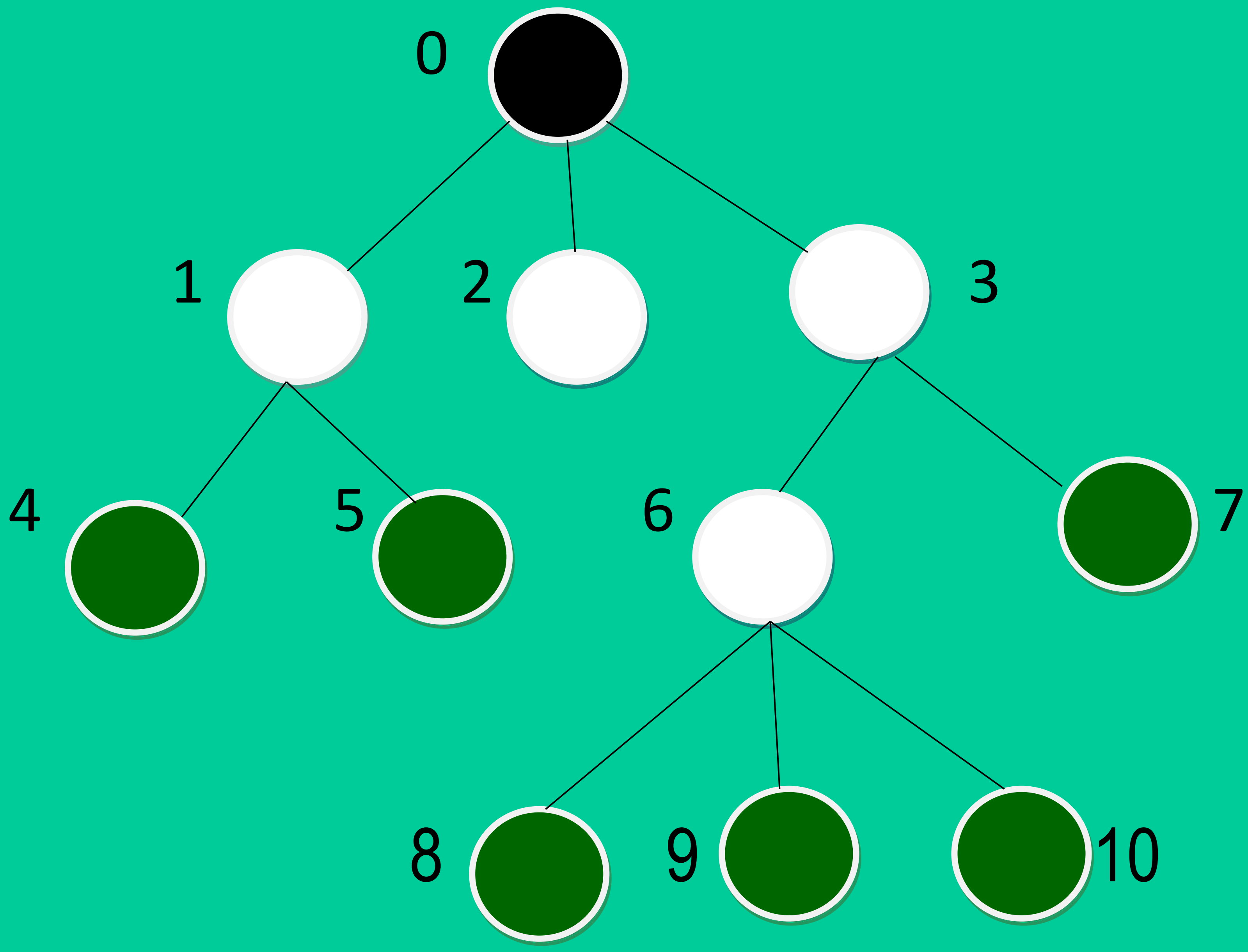
Pourquoi l'arbre général ?

- Chaque nœud d'un arbre **binaire** a **au plus deux** nœuds fils.

- Mais, dans certains cas le nombre de fils de chaque nœud **n'est plus limité** à deux.

C’est, par exemple, le cas d’un **compilateur où** les **instructions** sont représentées sous forme d’un **arbre** général




0
1
2
3
4
5
6
7
8
9
10

On introduit alors une structure **arborescente** plus large.

Cette structure est appelée **arbre planaire général** ou plus simplement **arbre**.

Cette structure est elle-même un **cas particulier** d'une structure plus générale: le **graphe**

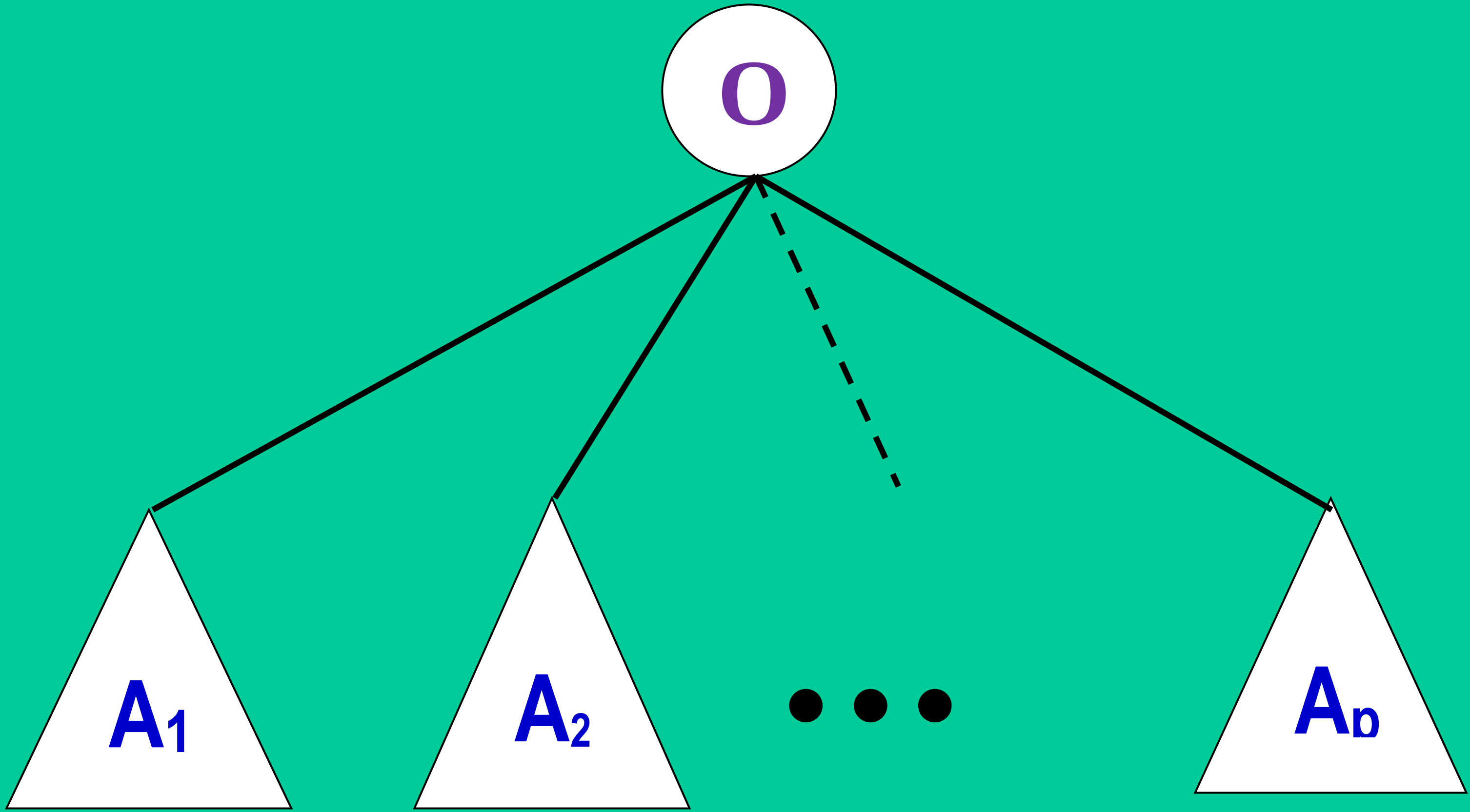
1- Définition

Un **arbre A** est :

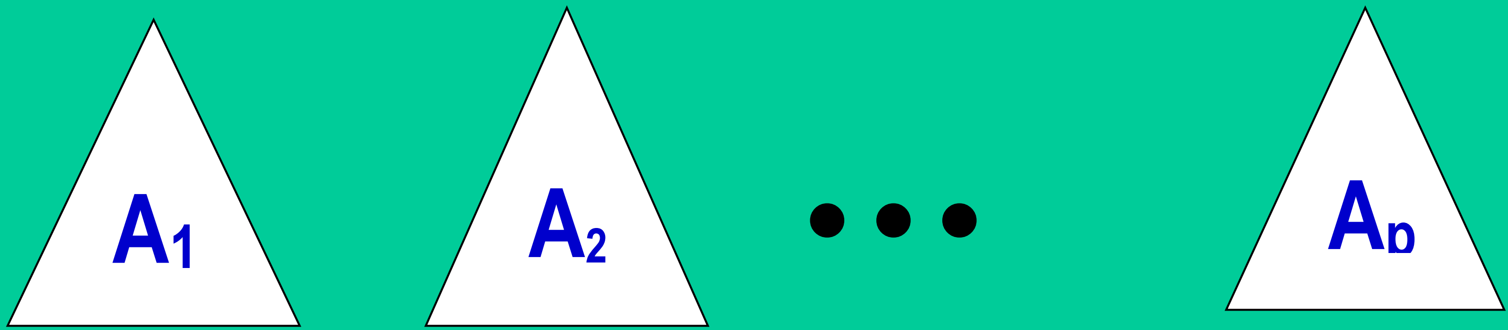
- -soit **vide,**
- -soit la donnée :
  - -d'une **racine**: **O**,
  - -et d'une **liste finie d'arbres disjoints** : [A_1 ,…,A_p]

On note :

$$A = < O,\ A_1 ,\ldots, A_p >$$



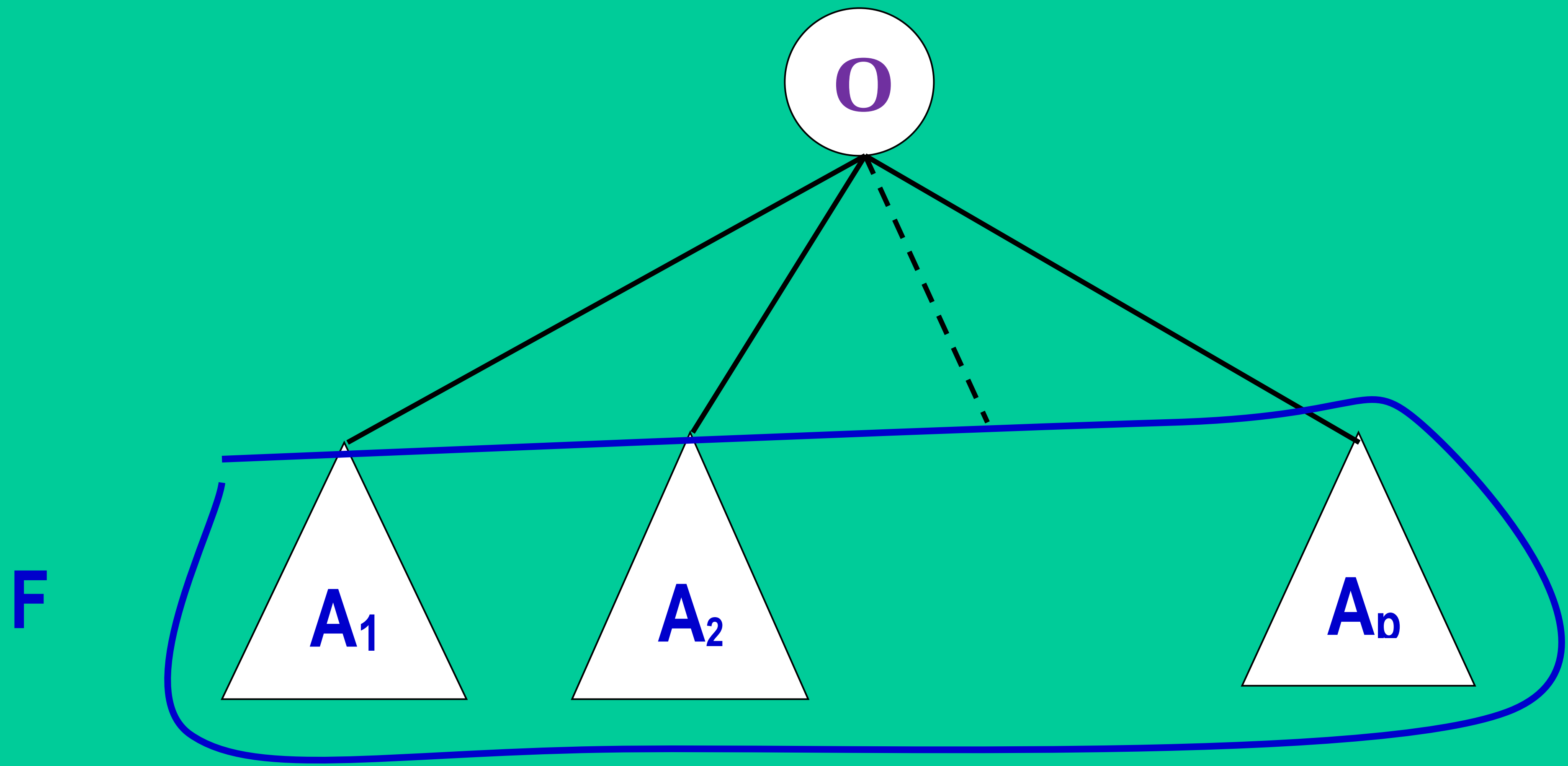
Une **liste** finie d'arbres **disjoints** est appelée une **forêt**.



Si F désigne une forêt, on note:

$$F = [\, A_1, \ldots, A_p \,]$$

Un arbre est donc obtenu en ajoutant un **nœud racine** **O** à une **forêt** **F**.



On note :

$$A =< o, F >$$

pour signifier qu’un **arbre** est **construit** à partir d’une **forêt**

et :

$$F= [A_1 ,\ldots,A_p]$$

pour signifier qu’une **forêt** est **construite** à partir d’une liste d’**arbres**.

Donc il faut spécifier les types abstraits des **arbres** et des **forêts** **conjointement**: dans une **même** spécification

Pourquoi ? :

- un **arbre** est construit à partir d'une **forêt**,
- mais une **forêt** est construite à partir d'**arbres**

Quelles opérations sur les arbres ?

-Pour construire un **arbre vide** :

arbreVide

-Pour construire un arbre partant d'une racine **o** et d'une forêt **F :**

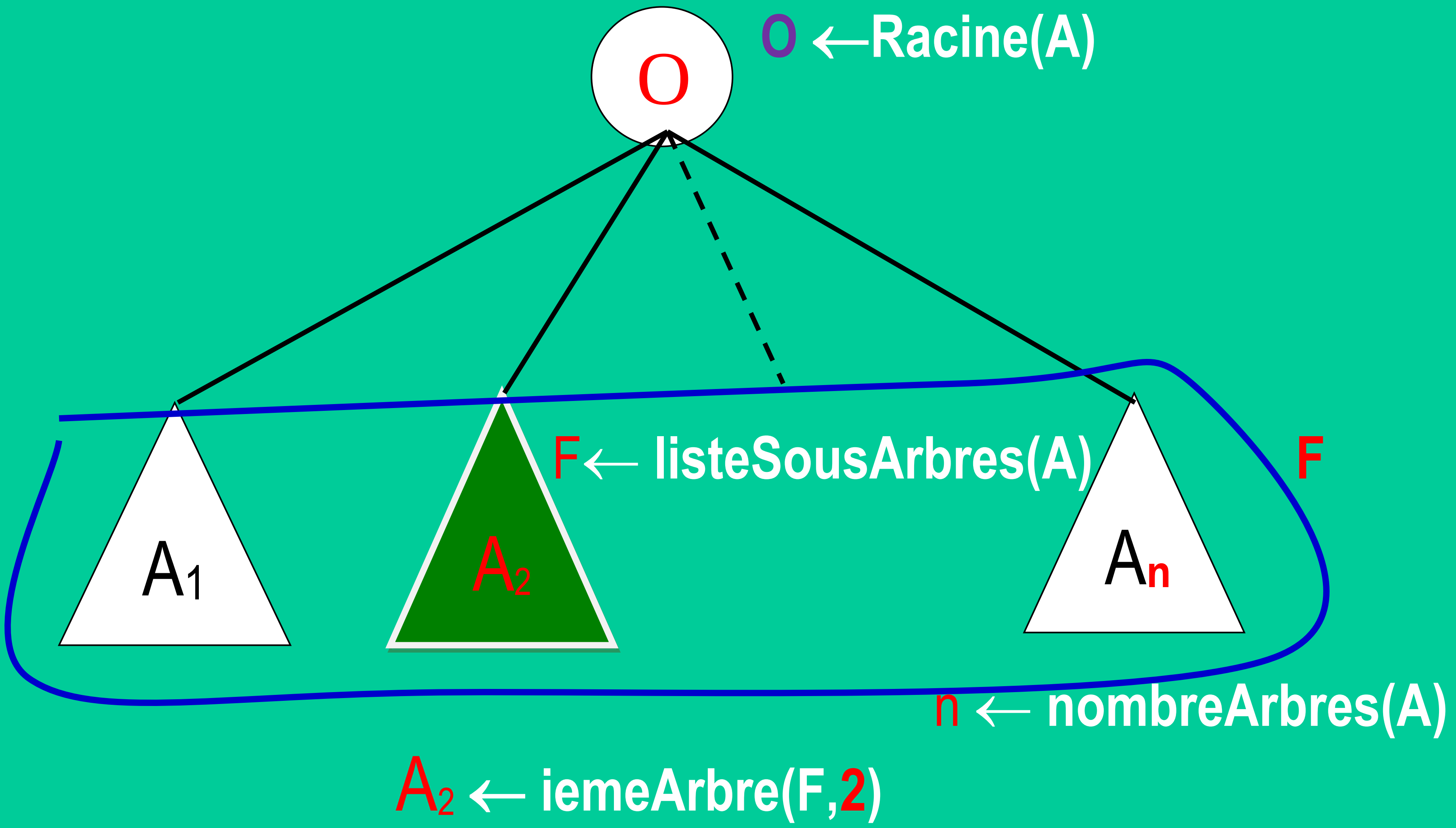
construire(0, F)

-Pour construire une forêt vide :

foretVide

-Pour construire une forêt en «plantant» un arbre **A** au rang **i** dans une forêt **F** **:**

planter(A, i ,F)


O ← Racine(A)
O
A1
A2
An
F← listeSousArbres(A)
F
n ← nombreArbres(A)
A2 ← iemeArbre(F,2)

2- Type abstrait arbre

Une spécification du type abstrait arbre peut être décrite comme suit :

```
spec ARBRE0 [sort Elem] given Int =
generated types
        Arbre[Elem] ::= arbreVide | construire(Elem; Foret[Arbre[Elem]])  ;
        Foret[Arbre[Elem]] ::= foretVide |
                              |planter(Arbre[Elem]; Int; Foret[Arbre[Elem]])
 end
```

```
spec ARBRE [sort Elem]  given Int =

        ARBRE0 [sort Elem]

then

    preds

        estArbreVide:     Arbre[Elem];
        estForetVide:     Foret[Arbre[Elem]]

    ops

        listeSousArbres : Arbre[Elem] →? Foret[Arbre[Elem]];
        racine :          Arbre[Elem] →? Elem;
        nombreArbres : Foret[Arbre[Elem]]→  Int
        iemeArbre:        Foret[Arbre[Elem]]    x Int →? Arbre;
```

$\forall$ A: Arbre[Elem]; F: Foret[Arbre[Elem]]; o: Elem

****Domaines de définition**

- **def** racine(A) $\Leftrightarrow$ $\neg$estArbreVide(A)
- **def** listeSousArbres(A) $\Leftrightarrow$ $\neg$estArbreVide(A)
- **def** iemeArbre(F, i, A) $\Leftrightarrow$ $\neg$estForetVide(F) $\wedge$ $1 \leq i \leq$ nombreArbres (F)

**pour tester si arbre ou forêt vide

- estArbre Vide(arbreVide)
- ¬estArbreVide(construire(o, F))

- estForetVide(foretVide)
- ¬estForetVide(planter(A,i,F))

**pour décrire un arbre

- racine(construire(o, F)) = o
- listeSousArbres(construire(o, F)) = F

**pour décrire une forêt

- **nombreArbres**(**foretVide**) = 0
- **nombreArbres**(**planter**(A,i,F)) = **nombreArbres**(F)+1

  - $0 < k \wedge k < i \Rightarrow$ **iemeArbre**(**planter**(A,i,F), k) = **iemeArbre**(F,k)
  - $k = i \Rightarrow$ **iemeArbre** (**planter**(A,i,F), k) = A
  - $i < k \wedge k \leq$ **nombreArbres**(A) +1 $\Rightarrow$
    iemeArbre(**planter**(A,i,F), k) = **iemeArbre**(F,k-1)

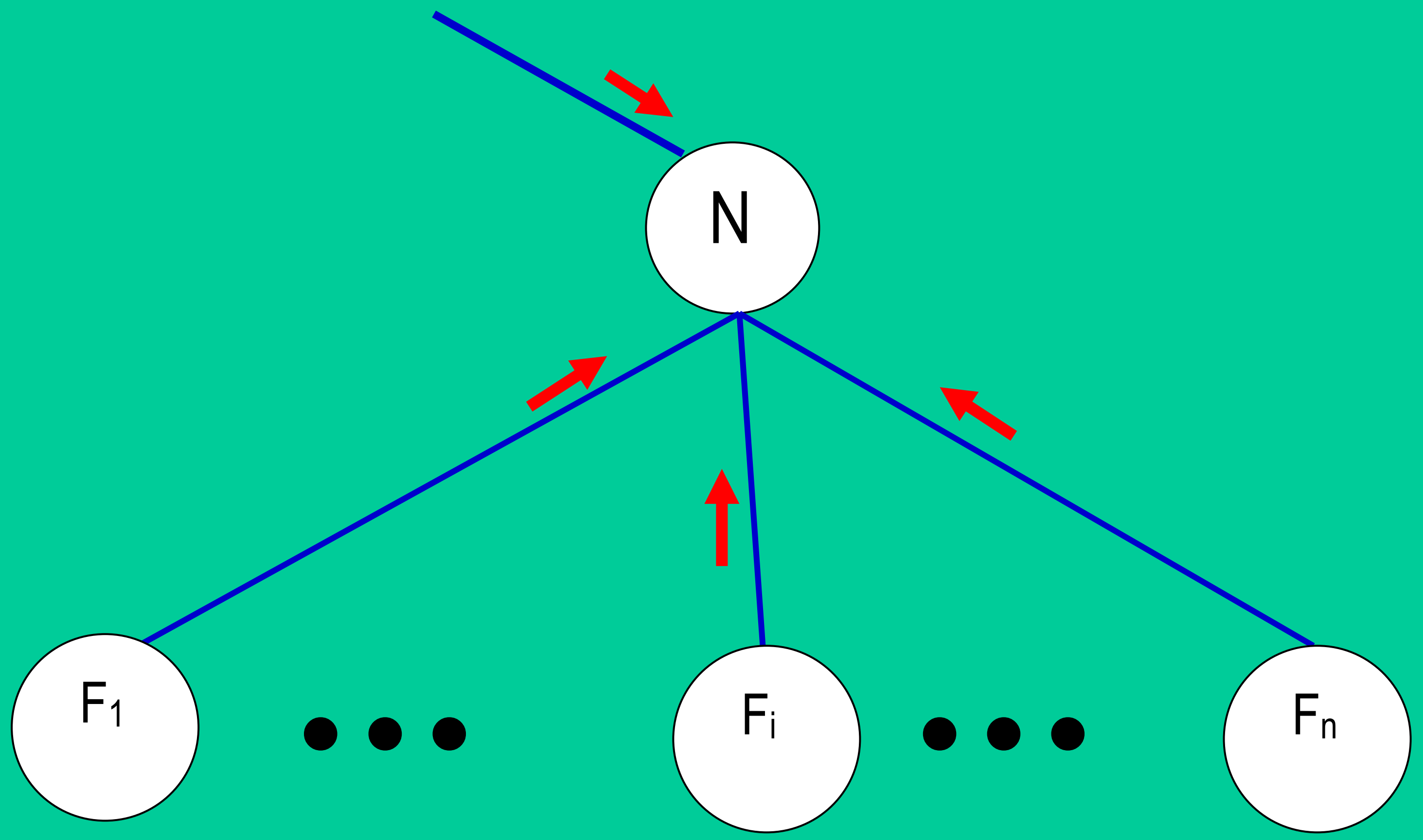
end

V- PARCOURS D'ARBRE

On peut généraliser le parcours en profondeur aux arbres généraux.

Dans ce parcours, chaque nœud **N** de l'arbre est rencontré «une fois de plus que son nombre de *fils* ».

On peut faire correspondre à chacun de ces passages un certain traitement du nœud rencontré.



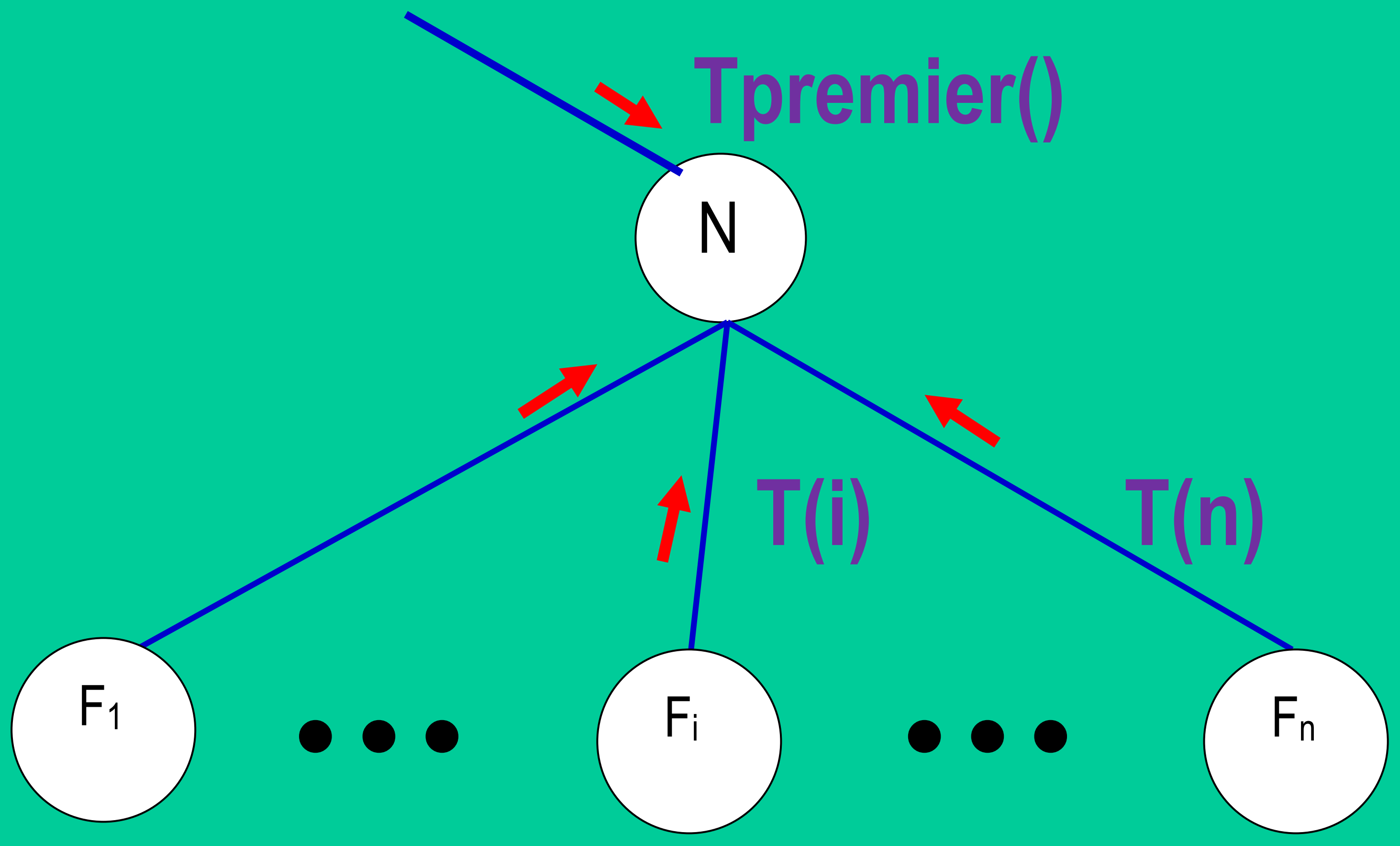
On note, si le nombre de fils est **n** :

Tpremier : le premier le traitement sur un nœud,

T(i) : le traitement après la visite du fils d'ordre i,

T(n) : le dernier traitement après la visite du fils d'ordre n,

Terminal : le traitement particulier des nœuds qui n'ont pas de fils : les **feuilles**.

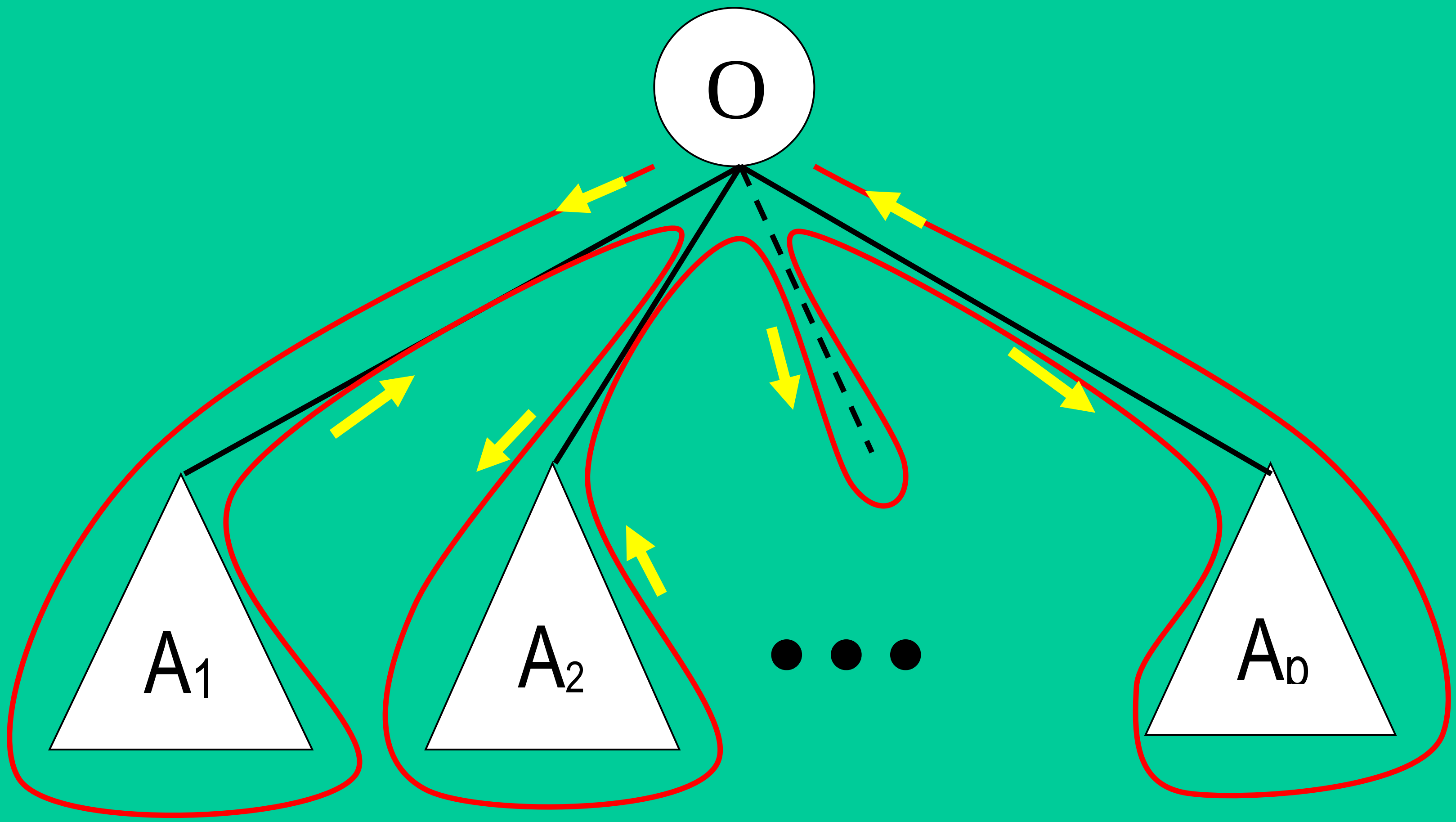

Tpremier()
N
T(i)
T(n)
F1
Fi
Fn

1- Parcours en profondeur

Le parcours en profondeur «main gauche» consiste à visiter tous les nœuds en **tournant autour** de l'arbre.

Il suit le chemin :

- qui part à **gauche** de la racine,
- et va toujours le **plus à gauche possible** en suivant l'arbre.


O
A1
A2
Ap

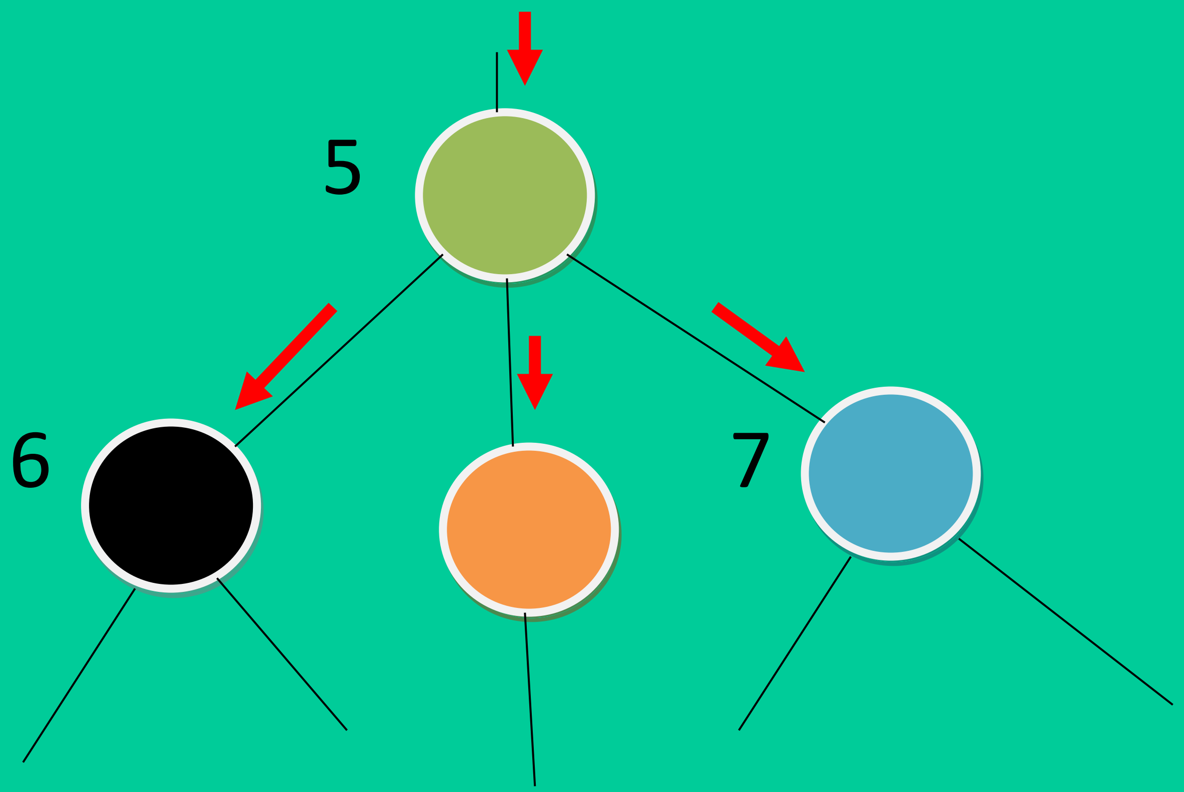
Ordre d’exploration d’un arbre

Deux ordres naturels d’exploration des arbres sont inclus dans le parcours en profondeur:

- l’ordre **préfixe**,
- l’ordre **suffixe**.

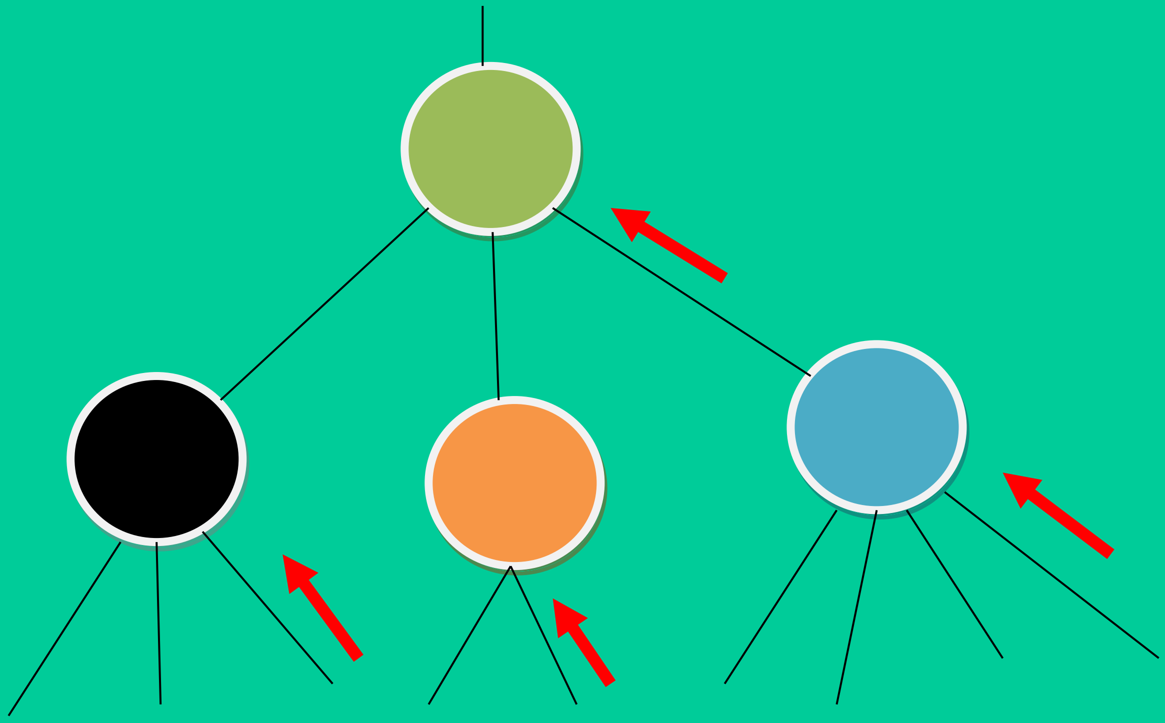
1-l’ordre préfixe:

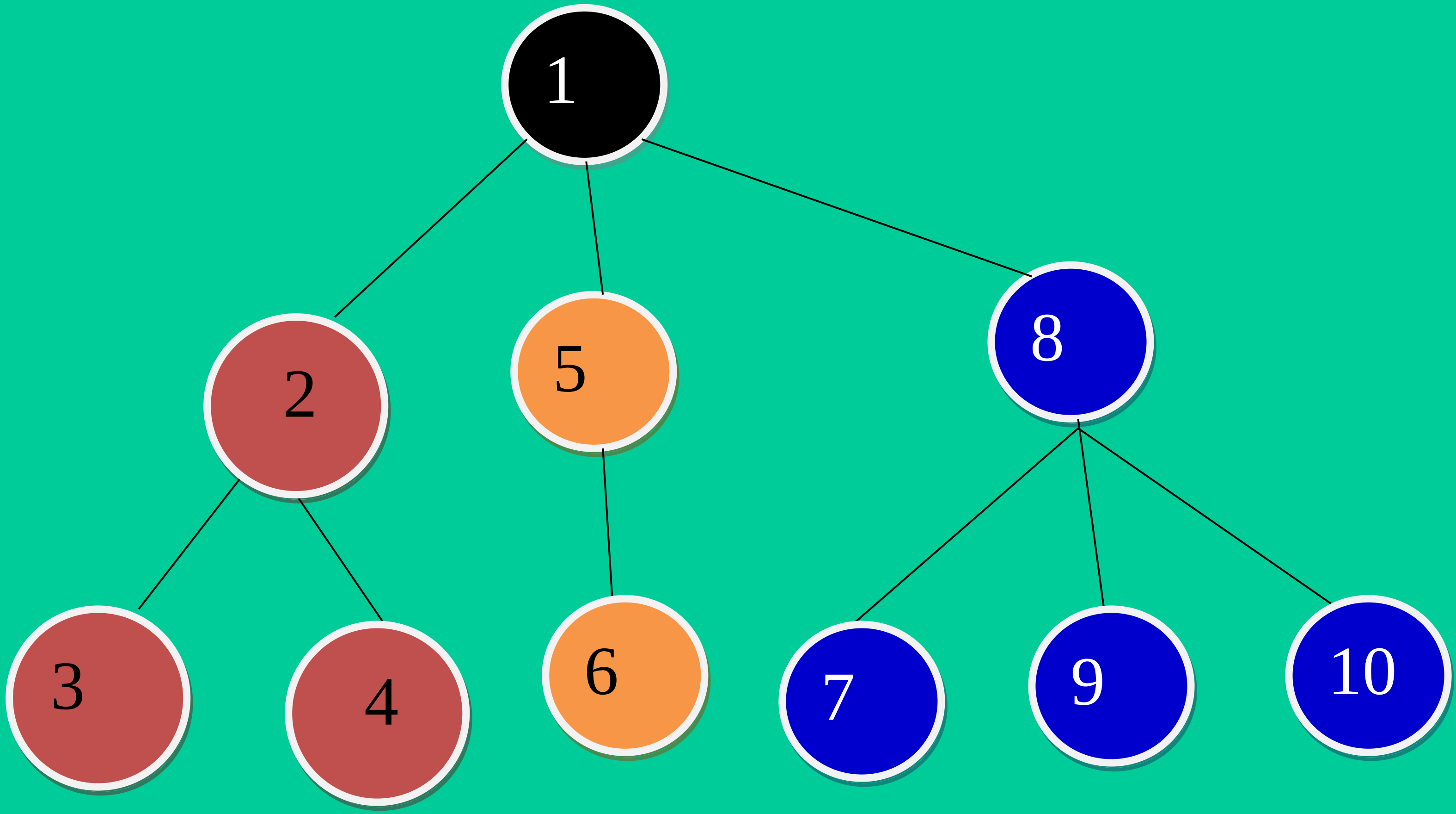
Chaque nœud n’est pris en compte que lors du **premier passage**.



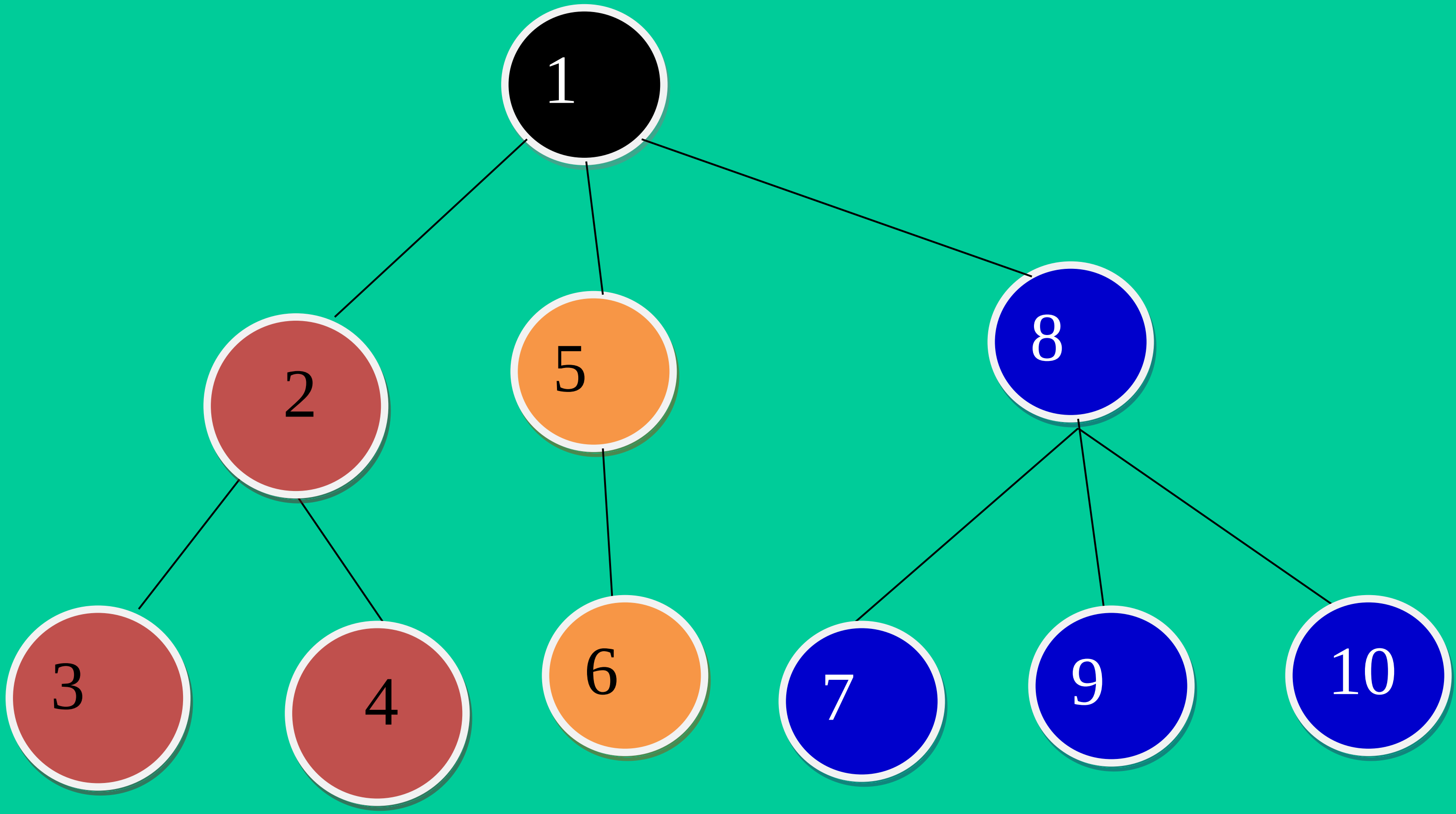
2-l’ordre suffixe:

Chaque nœud n’est pris en compte que lors du **dernier passage**.



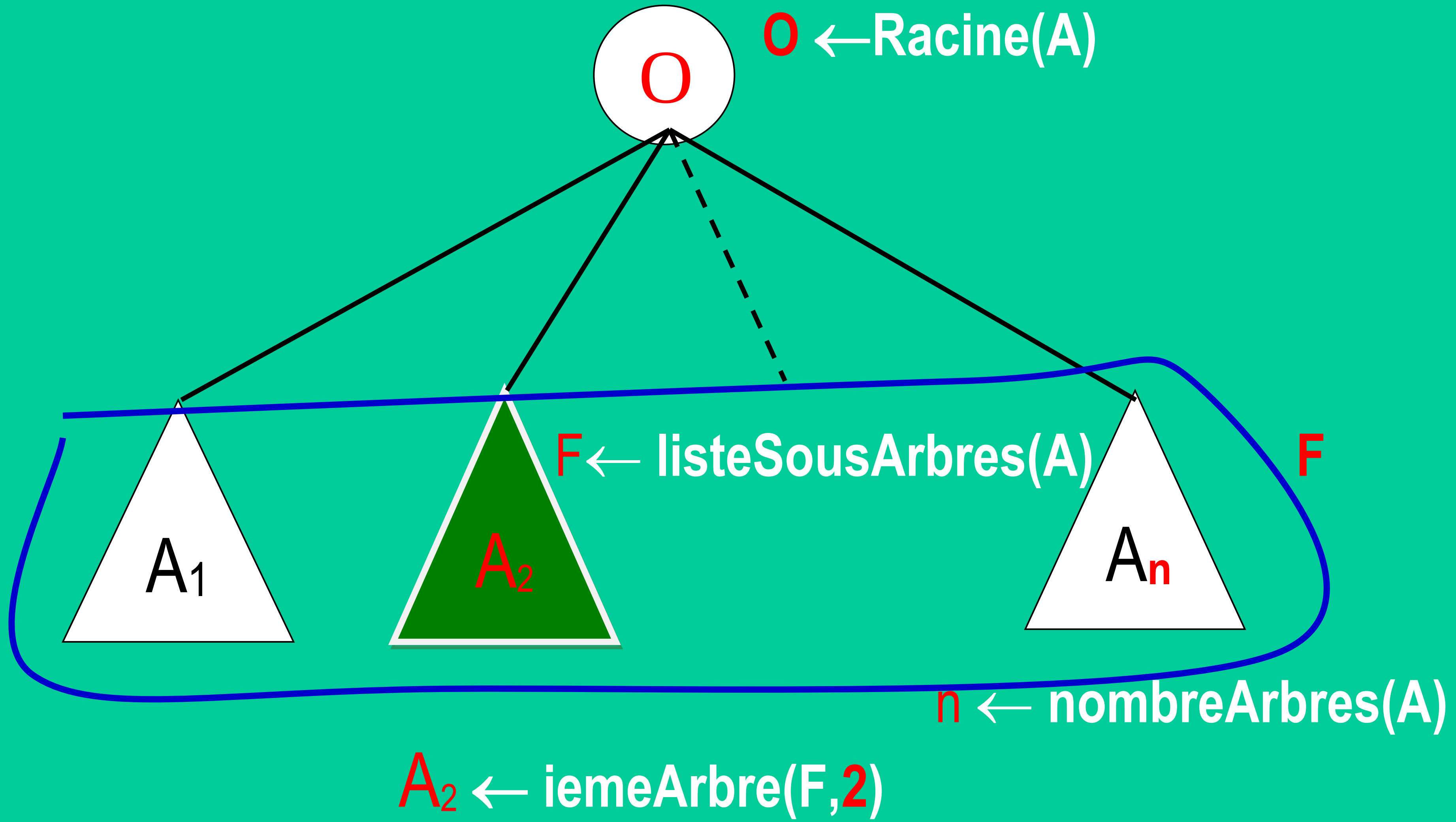


Parcours préfixe : **1 2 3 4 5 6 8 7 9 10**



Parcours suffixe: 3 4 2 6 5 7 9 10 8 1

Rappel :



O
O ← Racine(A)
F ← listeSousArbres(A)
F
A_1
A_2
A_n
n ← nombreArbres(A)
A_2 ← iemeArbre(F,2)

Procédure de parcours en profondeur

```
profondeur(ARBRE A)
  {
     int i , n;
      n = nombreArbres(listeSousArbres(A)) ;   /* n : nombre de fils  */
      if( estForetVide(listeSousArbres(A) )
            /* traitement spécial de nœud n'ayant pas de fils */
            Terminal( ) ;
```

```
else
 {

  /* traitement avant de visiter les n  fils */
  Tpremier() ;

 for(i=1 ; i<= n; i++)
                 {
                 profondeur(iemeArbre(listeSousArbres(A),i));
                 T(i) ; /* traitement après la visite du fils de rang i */
                 }
 }
}
```

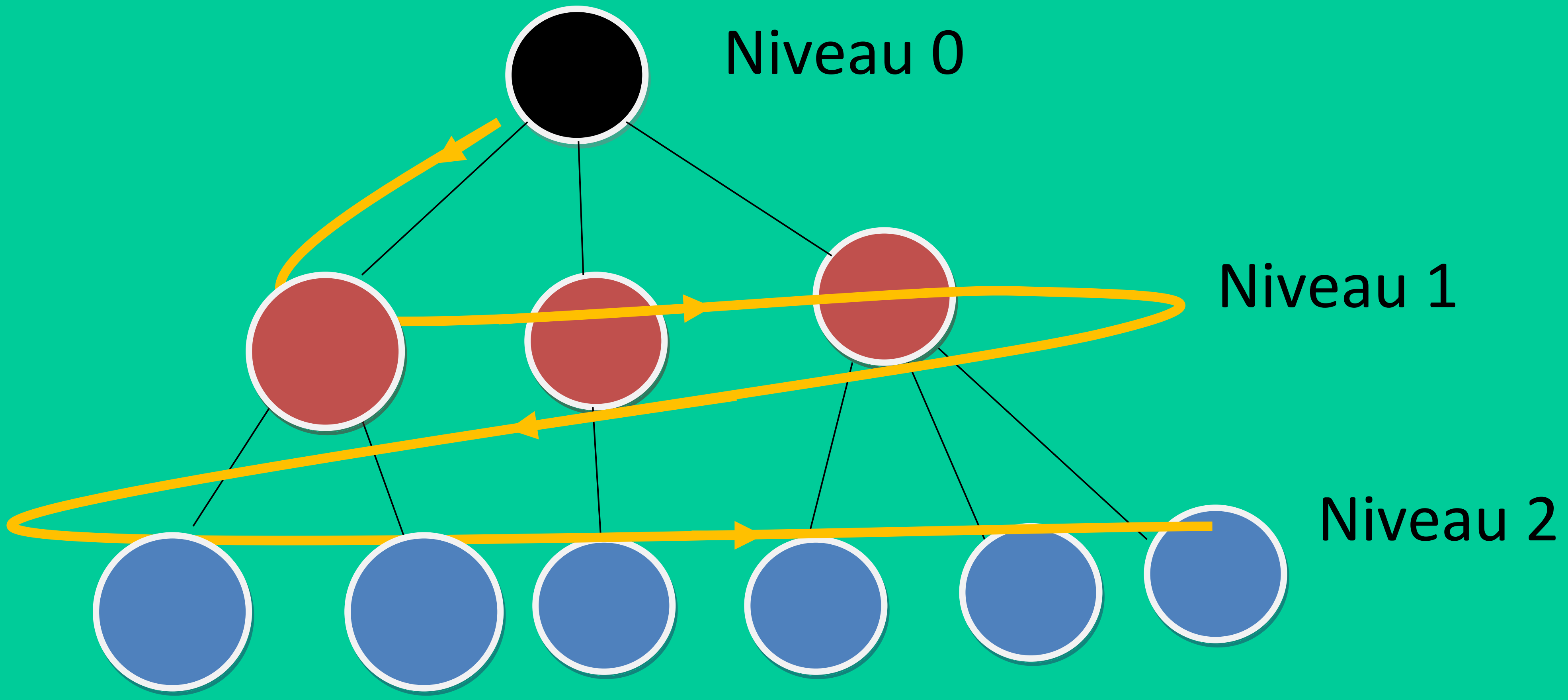
2- Parcours d'arbre en largeur

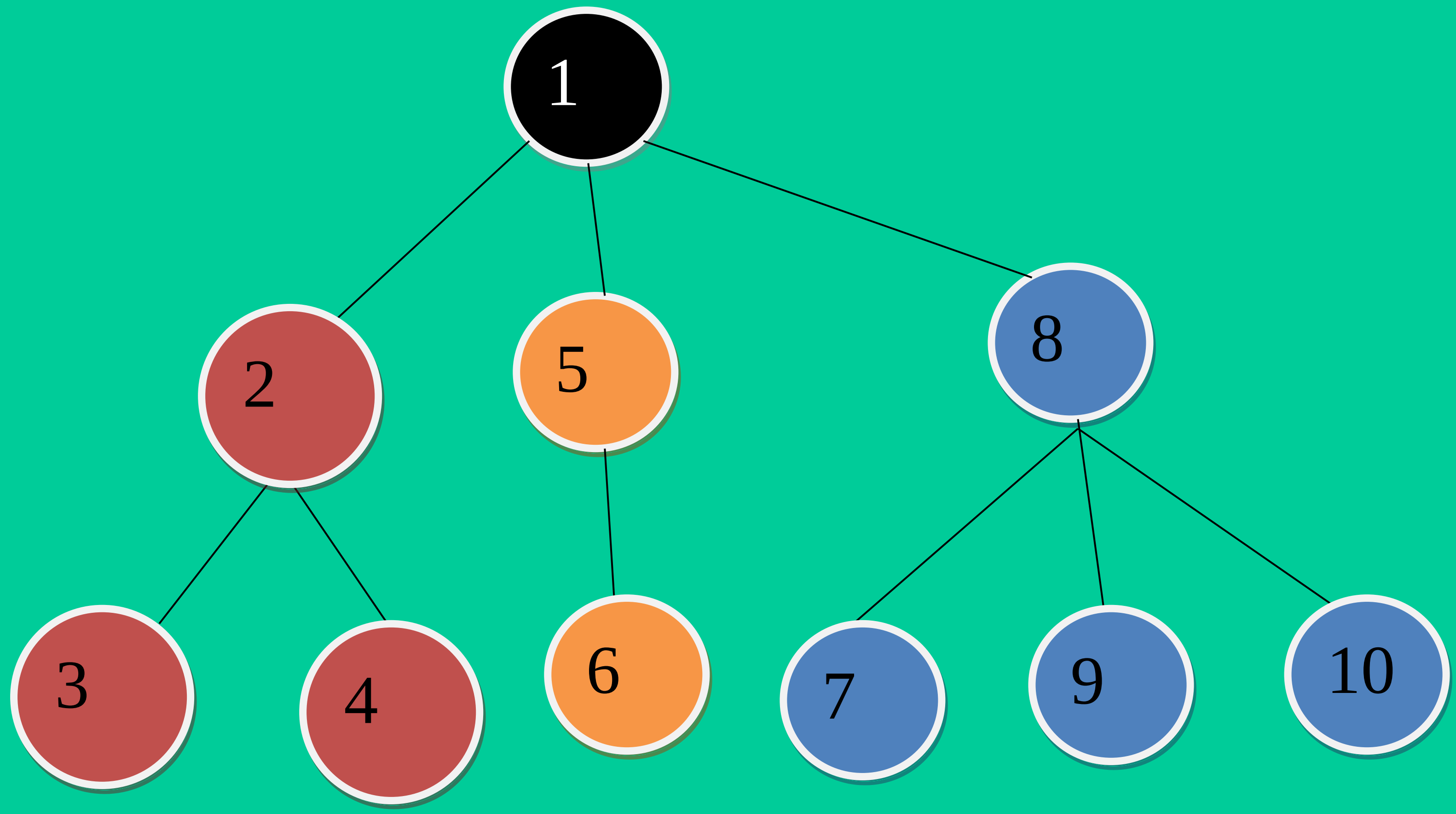
Un parcours en largeur permet de visiter les nœuds «**niveau par niveau**».

On commence par visiter le nœud de niveau 0: la racine de l'arbre.

Ensuite, on visite, de gauche à droite :

- tous les nœuds de **niveau** 1,
- puis ceux de niveau 2,
- ainsi de suite jusqu'au dernier niveau.


Niveau 0
Niveau 1
Niveau 2


1
2
5
8
3
4
6
7
9
10

1

~~1~~ 2 5 8

~~1~~ ~~2~~ 5 8 3 4

~~1 2~~ ~~5~~ 8 3 4 6

~~1 2~~ ~~5~~ ~~8~~ 3 4 6 7 9 10

~~1 2~~ ~~5~~ ~~8 3~~ 4 6 7 9 10

~~1 2~~ ~~5~~ ~~8 3 4~~ 6 7 9 10

~~1 2~~ ~~5~~ ~~8 3 4~~ ~~6~~ 7 9 10

~~1 2~~ ~~5~~ ~~8 3 4~~ ~~6~~ ~~7~~ 9 10

~~1 2~~ ~~5~~ ~~8 3 4~~ ~~6~~ ~~7~~ ~~9~~ 10

~~1 2~~ ~~5~~ ~~8 3 4~~ ~~6~~ ~~7~~ ~~9~~ ~~10~~

Procédure de parcours en largeur

```
Largeur(Arbre A)
   {
    int i;  Arbre A0; Ai; Foret  F;

/*  file : file pour ranger les racines des sous- arbres qui ont été visitées*/

    FILE file;
    file = fileVide() ;
    A0 = A;
/* visiter : fonction qui permet de visiter la racine d'un arbre */
    visiter(A0)  ;
    /* au départ, on visite la racine de arbre*/
    file = enfiler(file, A0) ;
```

```
while( ! estVide(file))
        {
         A0 = premier(file) ;
         file = defiler(file) ;
/* visiter les racines des sous_arbres  de A  */
        F= listeSousArbres(A0) ;
        for(i=1 ; i<= nombreArbres(F); i++)
                {
                 Ai= iemeArbre(F,i) ;
                visiter(Ai));
                 file = enfiler(file, Ai) ; /*  mise en file du sous arbre i de A */
                 }
          }
}
```