U.F.R SCIENCES ET TECHNIQUES
Département d'Informatique
B.P. 1155
64013 PAU CEDEX
Téléphone secrétariat : 05.59.40.79.64
Télécopie : 05.59.40.76.54

TYPE DE STRUCTURES D'ARBRE

I- ARBRE BINAIRE
II- MESURE SUR LES ARBRES
IIII- PARCOURS D'ARBRE BINAIRE
IV- ARBRE GENERAL
V- PARCOURS D'ARBRE

INTRODUCTION

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 la notion de **hiérarchie**.

Exemples

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

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

La **compression des données** utilise un codage d’arbre: algorithme de **Huffman**.

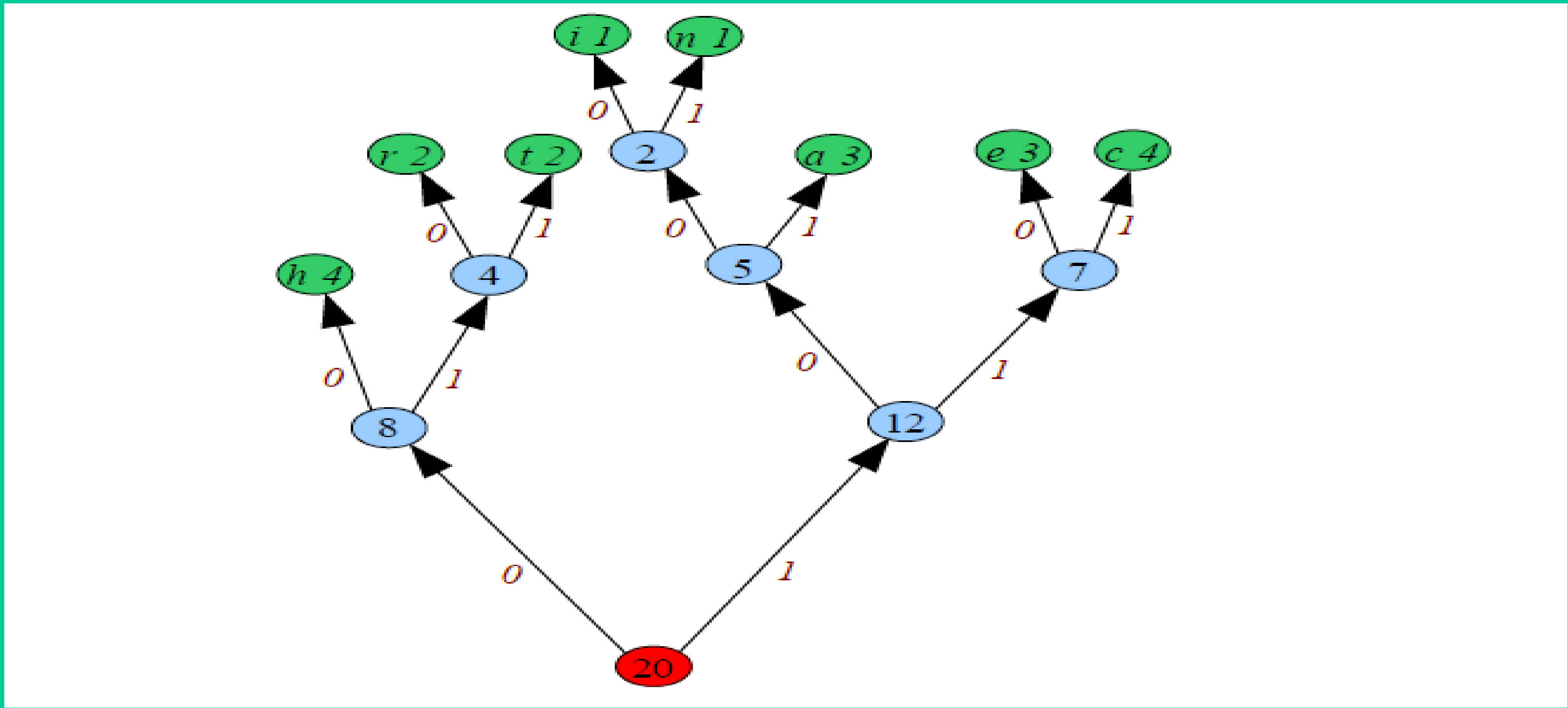
Il est en particulier utilisé pour une **compression** en :

- **mp3** (format d'un fichier son),
- **jpeg** (format d'une image)
- ou **mpeg** (format d'une video).

Exemple du codage de la phrase:

«recherche chat châtain »

par un arbre binaire :



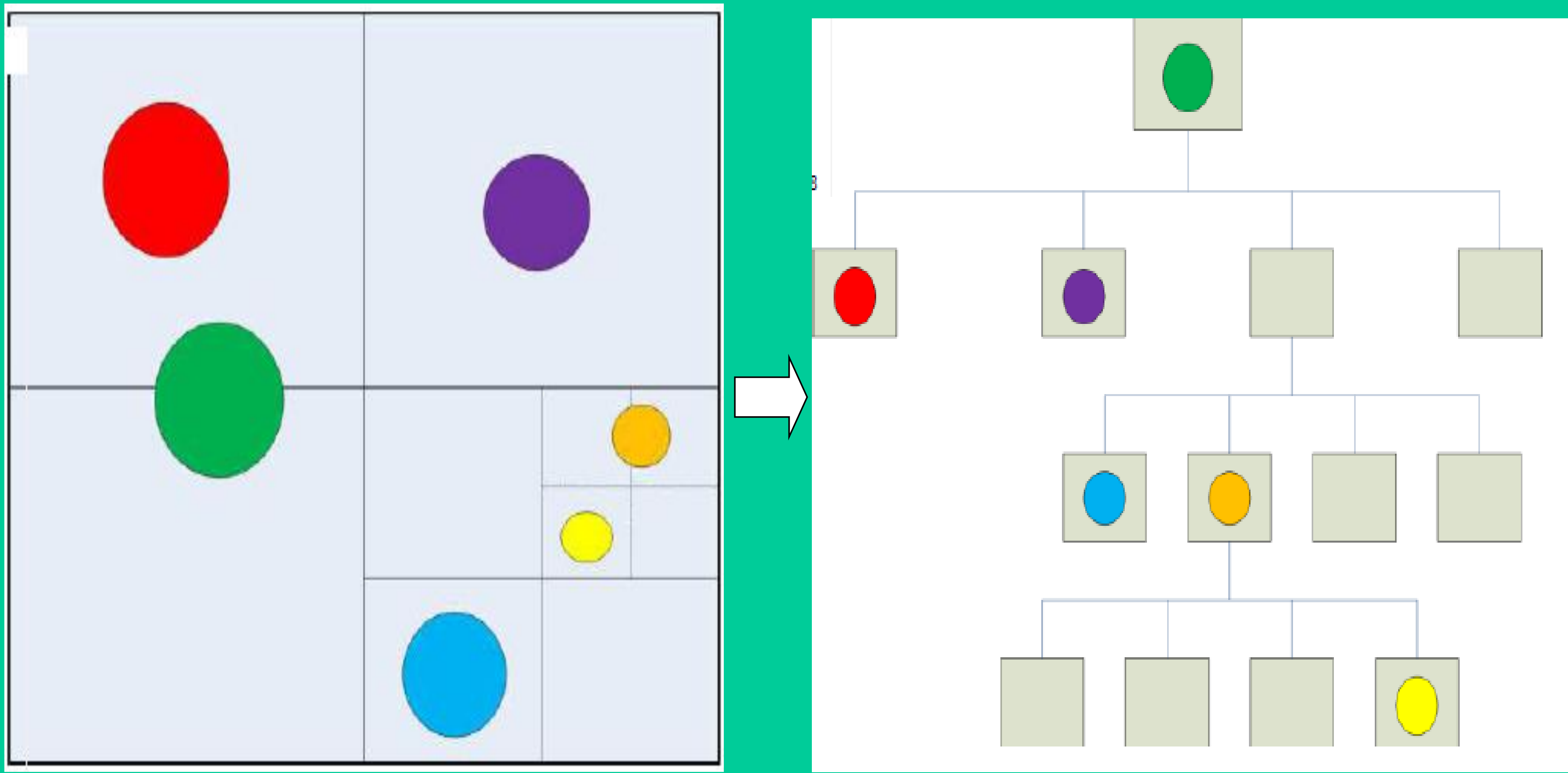
Le codage des 20 caractères (bytes) par 58 bits est :

0101101110011001011100110111001010111110010101110110001001

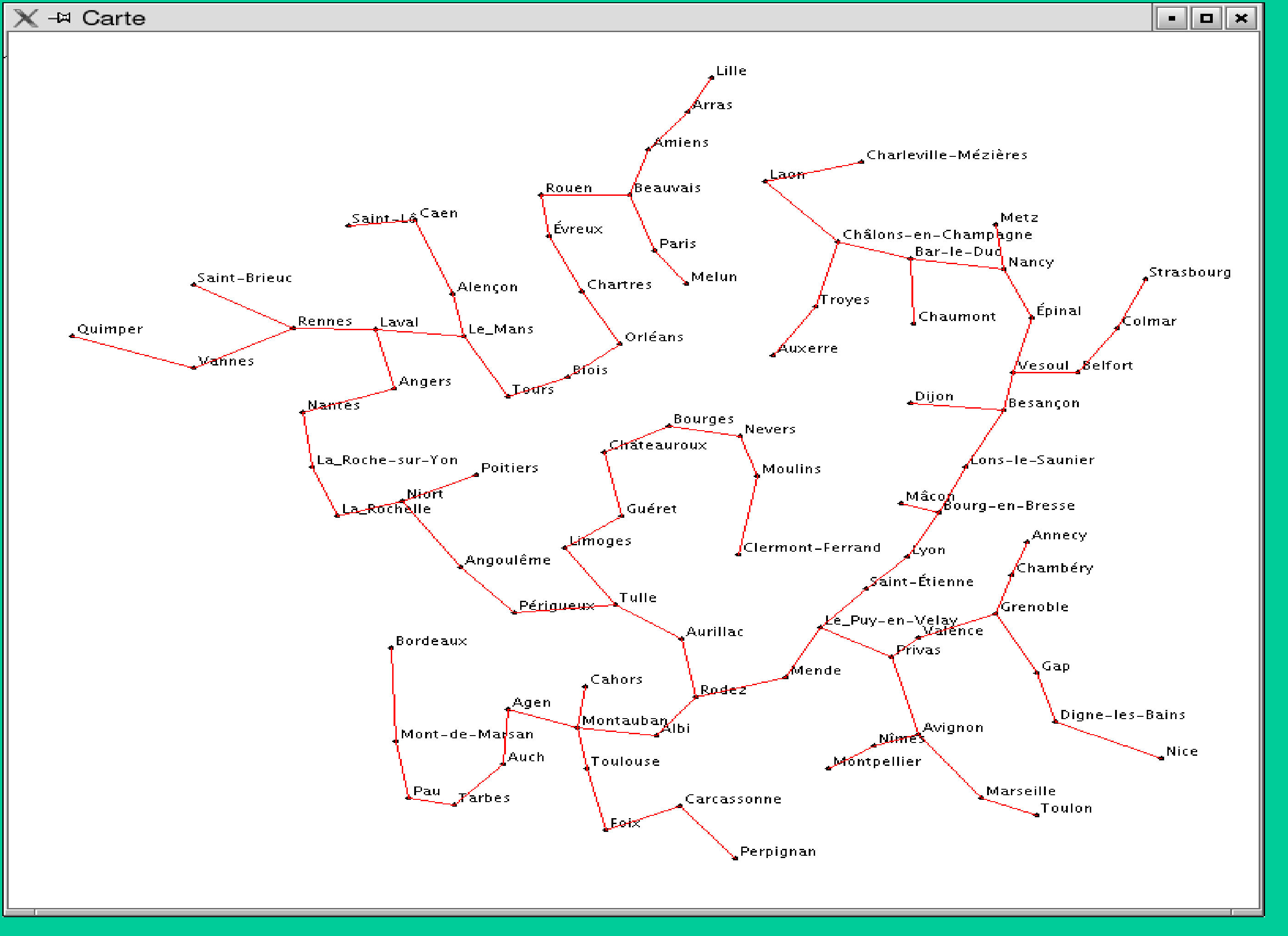
Le taux de compression est donc:

58/160 = **0,3625**

La synthèse d'images utilise la représentation d'arbre

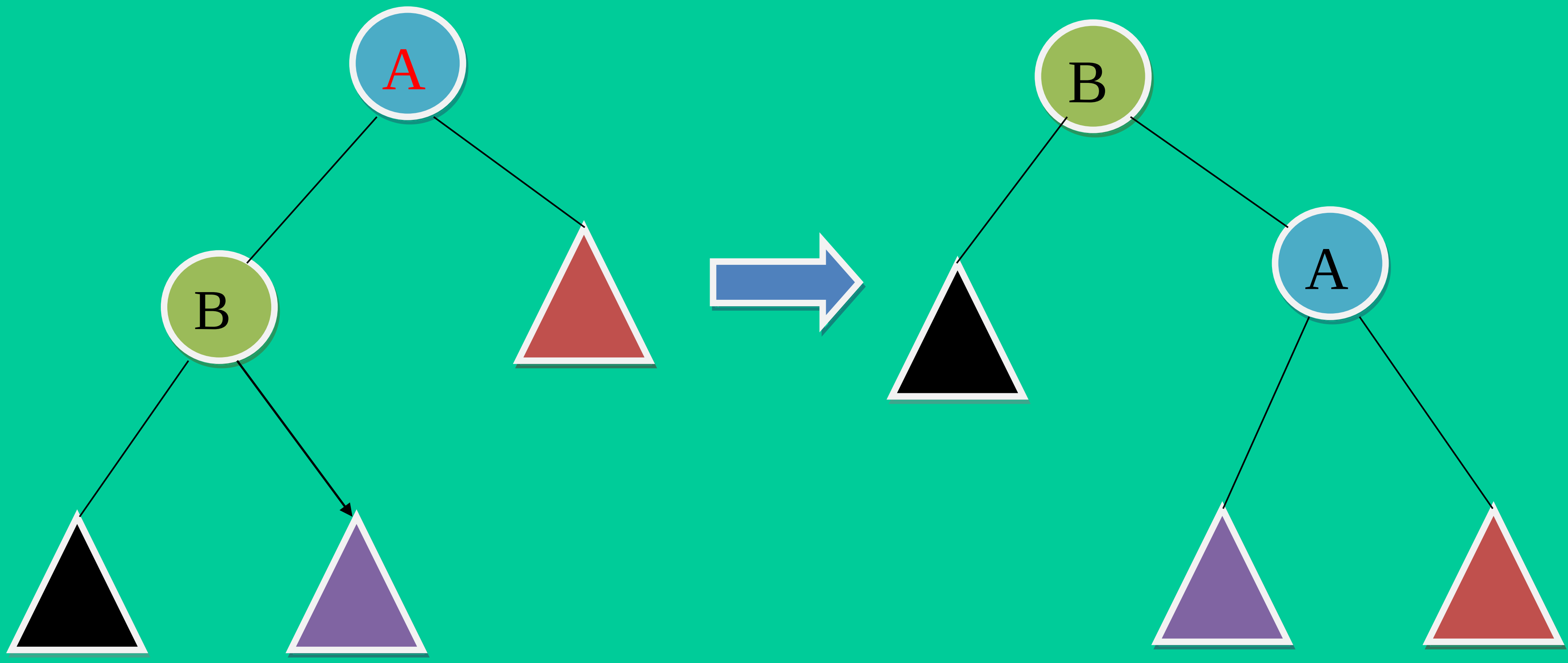


La **cartographie** utilise un **arbre** pour représenter un pays ou région.

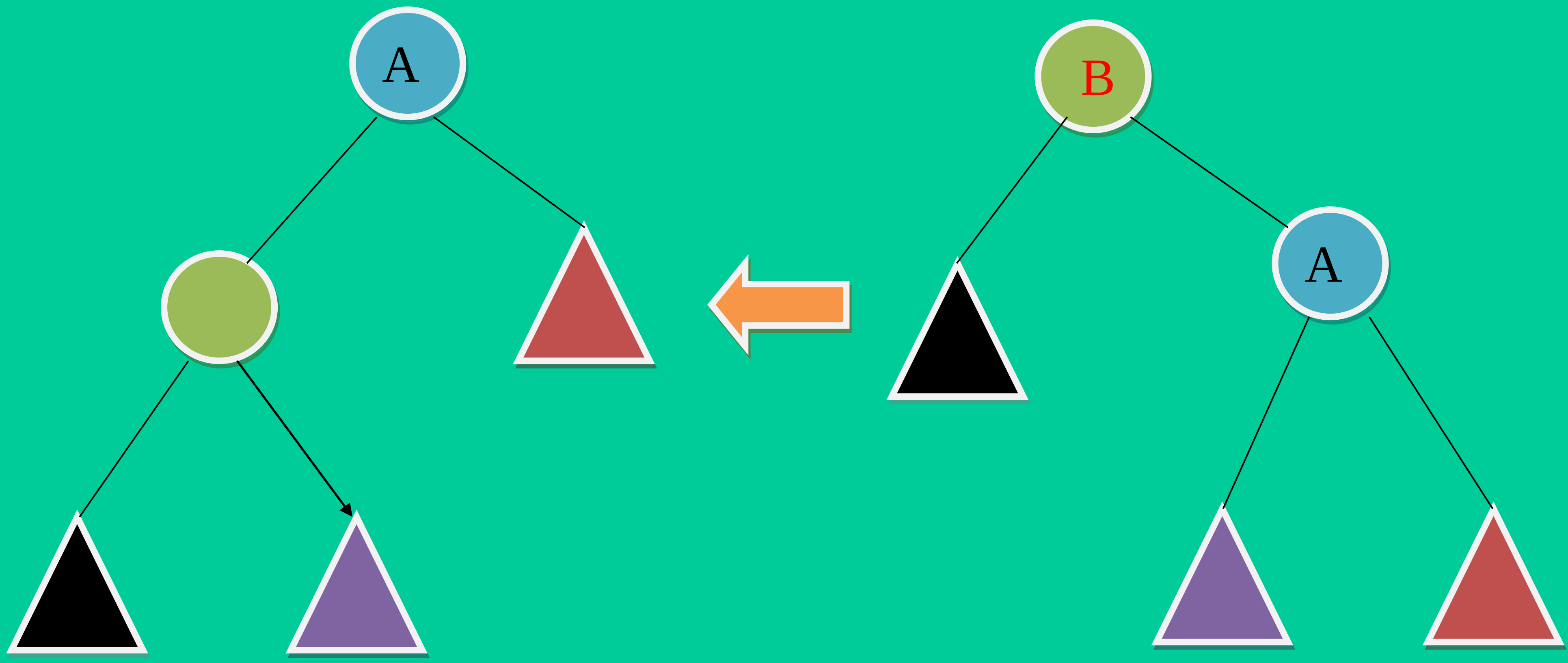


Transformation de la structure d'un arbre

Rotation à droite autour de A notée rd(A)



Rotation à gauche autour de B notée rg(B)



Qu’est-ce qu’un arbre ?

Un arbre est un ensemble de **nœuds**.

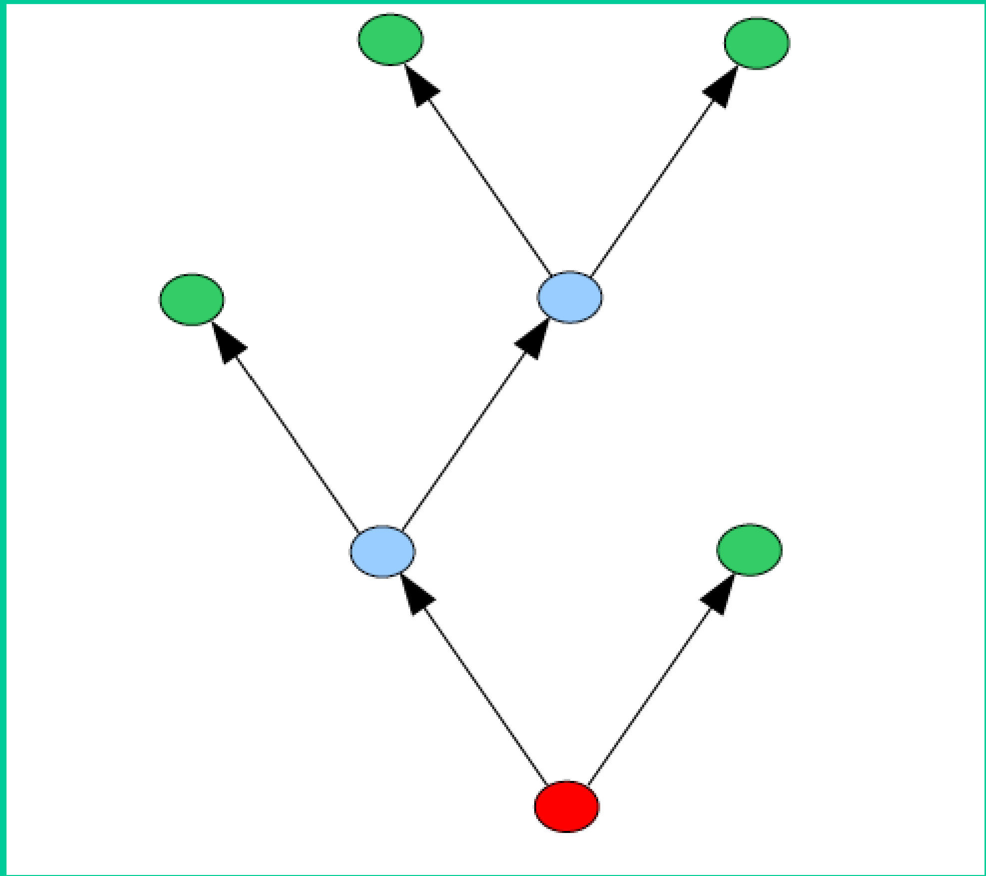
Les nœuds sont organisés de façon **hiérarchique**.

Cette **hiérarchie** distingue :

-un nœud **particulier** appelé **racine**,

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

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



I- Arbre binaire

1- définition récursive

Un arbre binaire est :

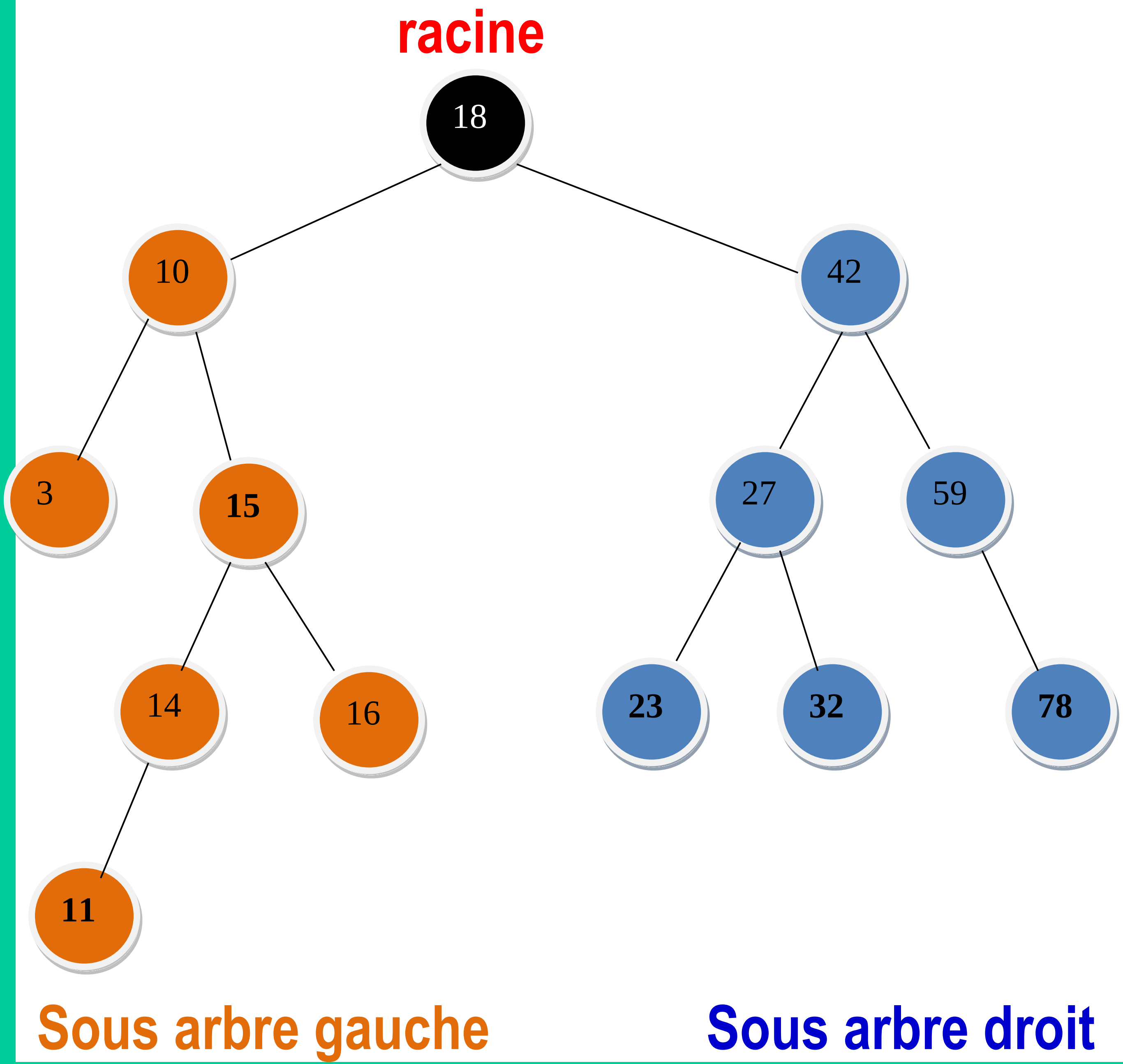
- soit **vide**,
- soit de la forme $<o, B_1, B_2>$

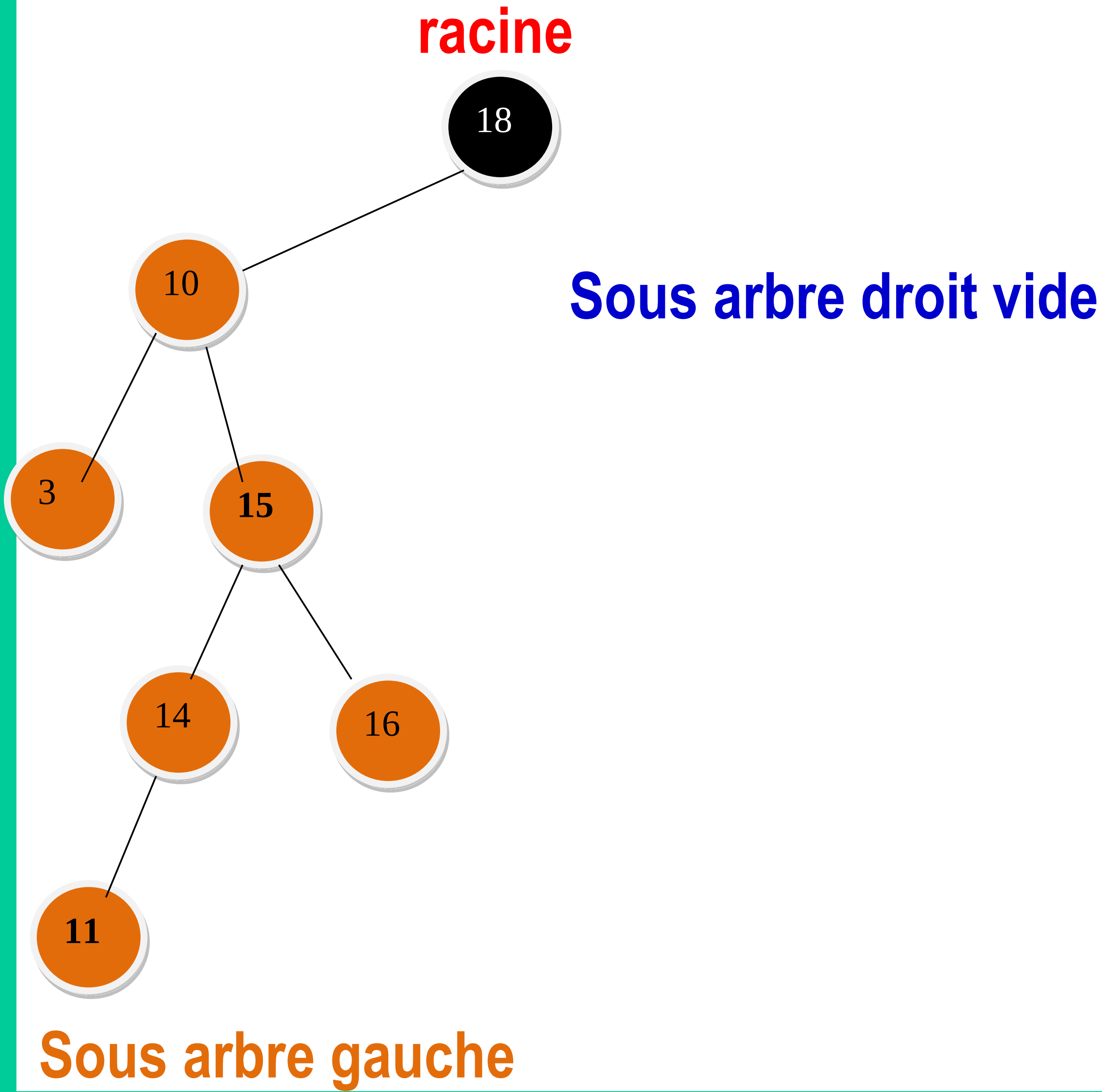
où :

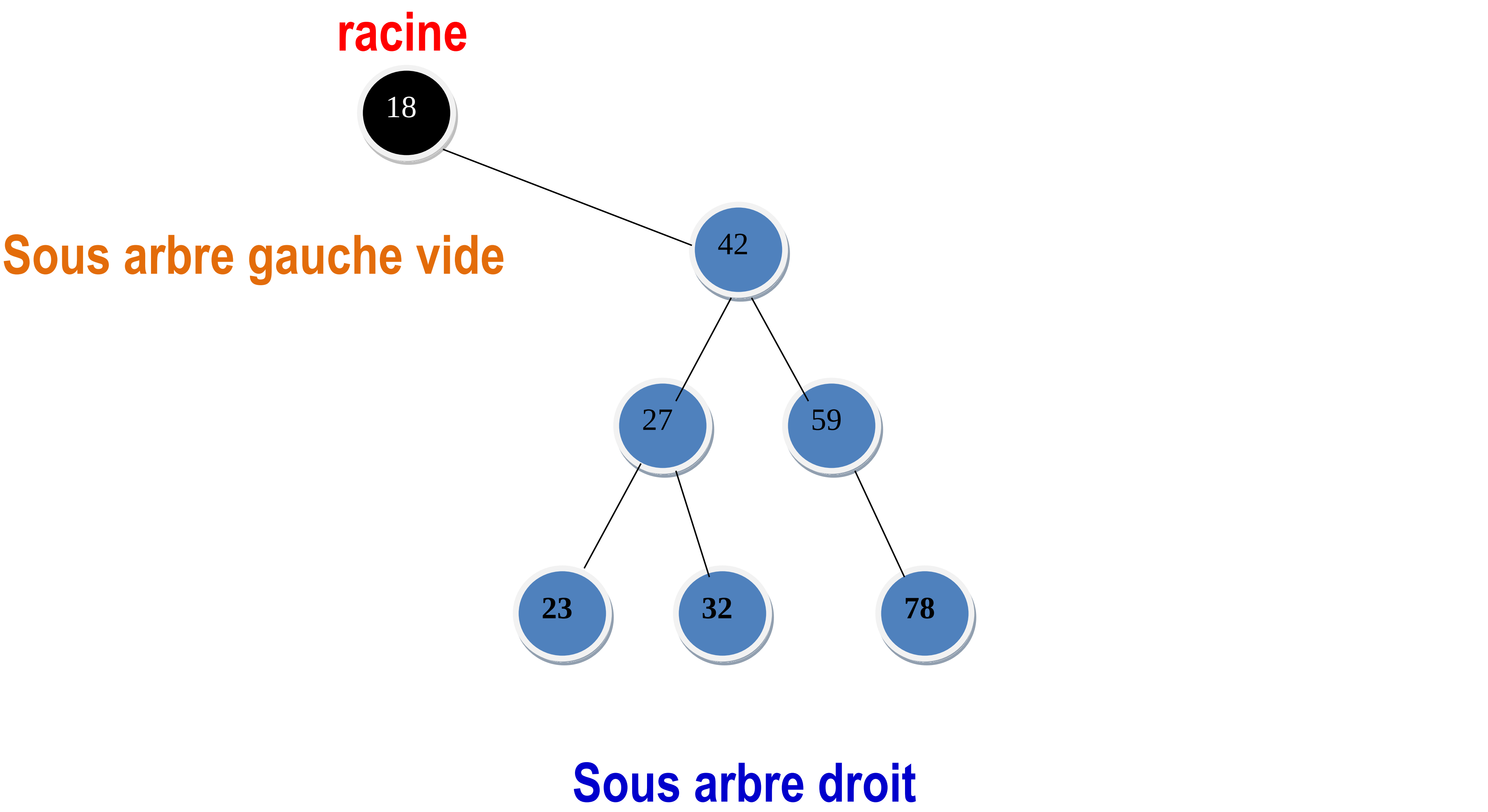
- B_1 et B_2 sont des arbres binaires **disjoints**,
- **o**, un nœud **racine**.

Etant donné un arbre binaire **B**= < **o**, **B**$_1$, **B**$_2$ > :

- **o** est la **racine** de **B**,
- **B**$_1$ est le **sous-arbre gauche** de la racine **o** de **B** ou simplement son sous-arbre gauche,
- **B**$_2$ est son **sous-arbre droit**.


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

Le type abstrait des **nœuds** peut être spécifié comme suit :

spec NŒUD[sort Element] =

generated type **Nœud** ::= **créer**(Element)

op

contenu : Noeud$\rightarrow$ Elem

$\forall$ n : Noeud; e: Element $\bullet$ **contenu**(créer(e)) = e

end

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) $\Leftrightarrow$ $\neg$ **estVide**(B)
- **def gauche**(B) $\Leftrightarrow$ $\neg$ **estVide**(B)
- **def droit**(B) $\Leftrightarrow$ $\neg$ **estVide**(B)

- **estVide**(arbreVide)
- $\neg$**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 :

$$\textbf{contenu}(B,o) = e$$

alors on notera abusivement:

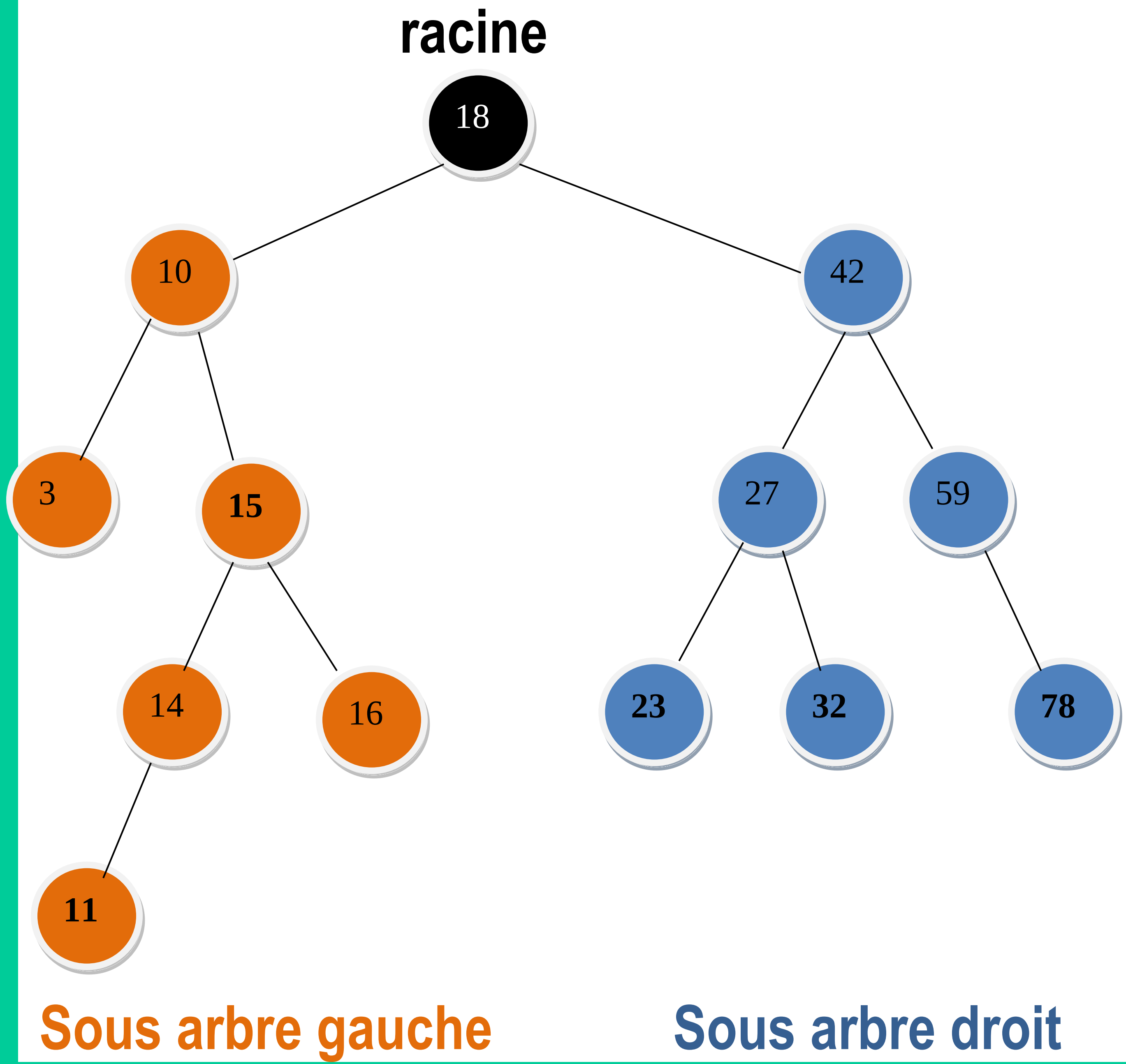
$$B = < e, B_1, B_2 >$$

3- Vocabulaire des arbres

Relations entre arbres binaires

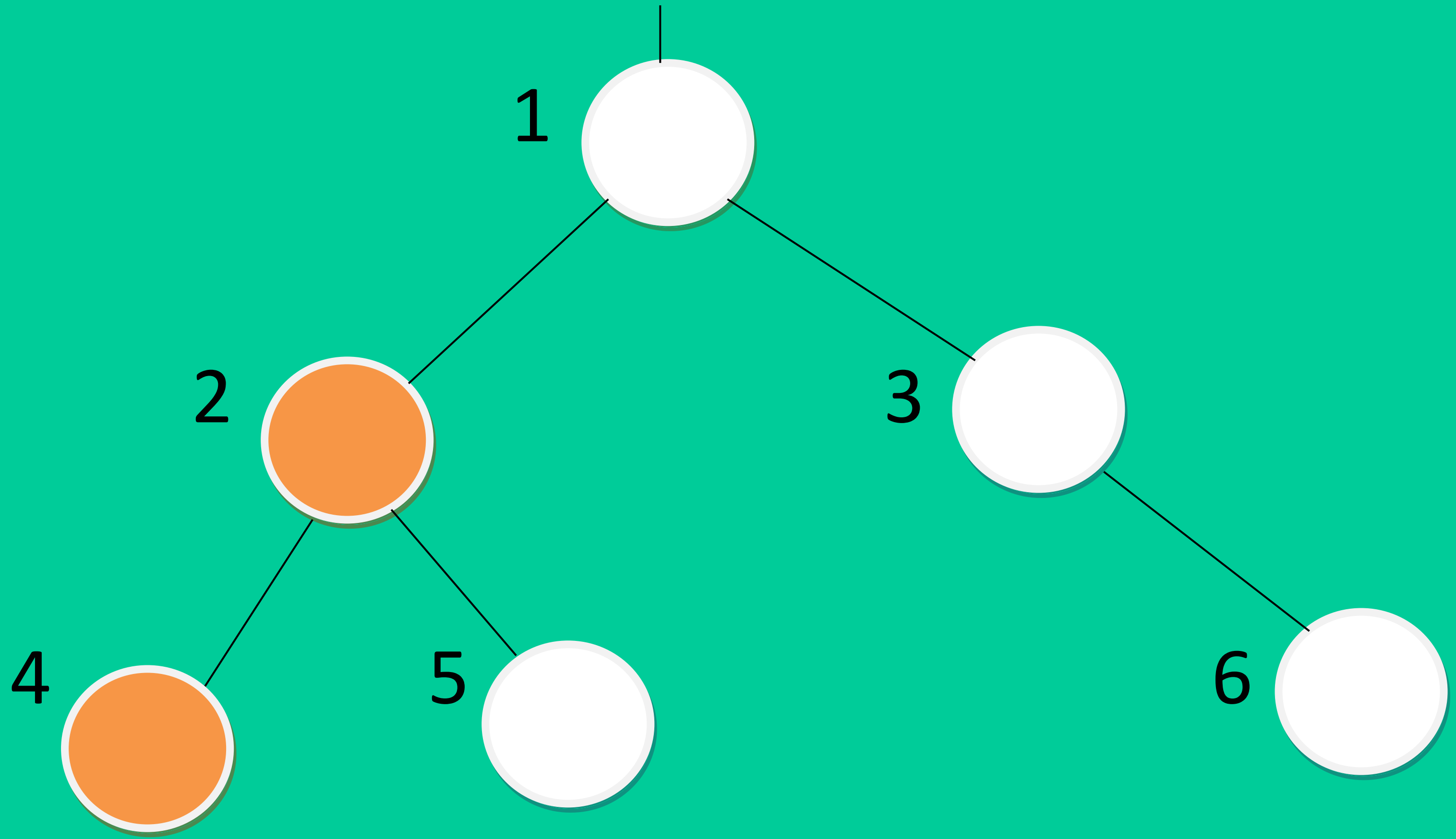
On dit que C est un **sous-arbre** de B si et seulement si:

- 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

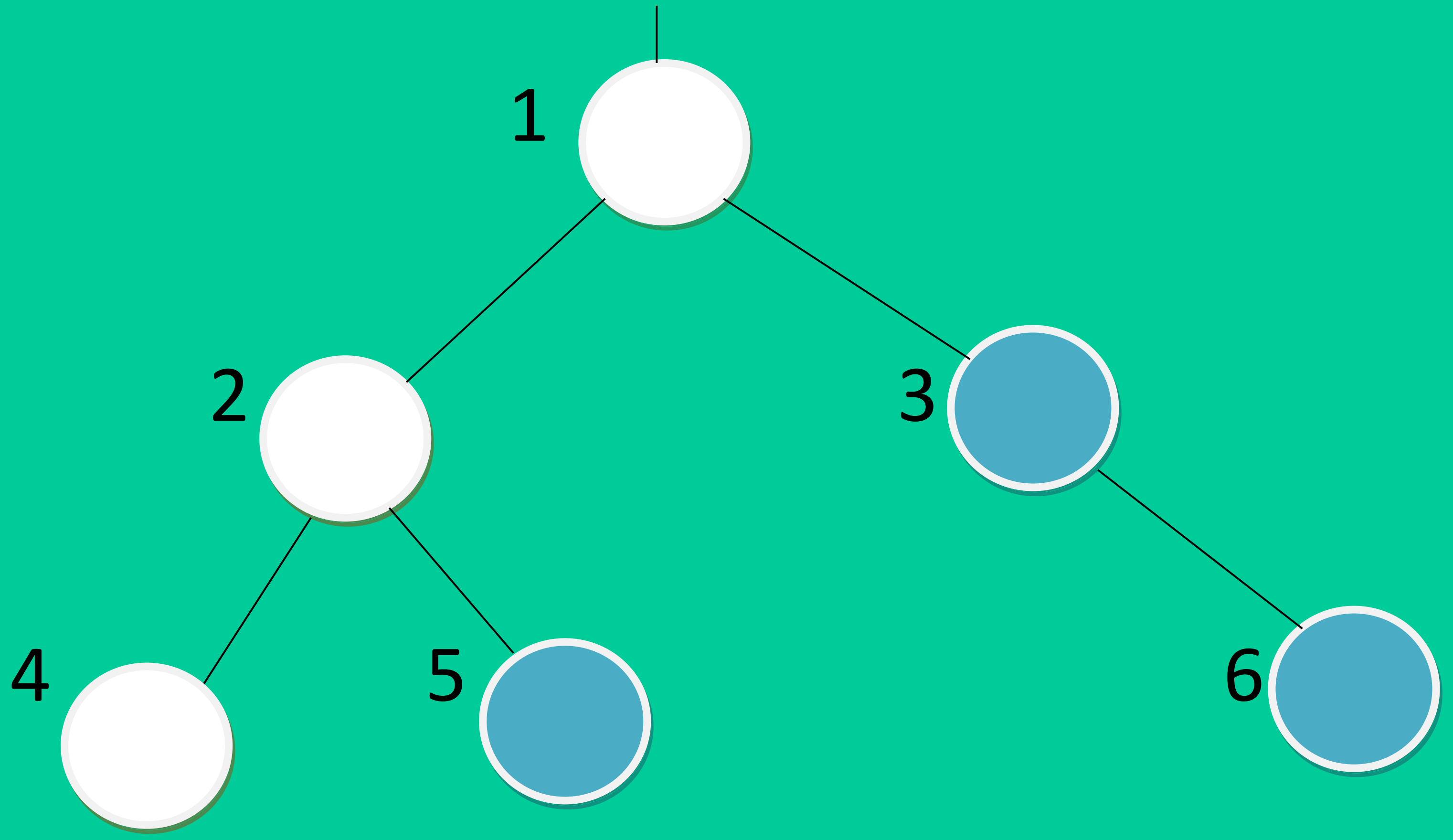
Relation entre nœuds

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



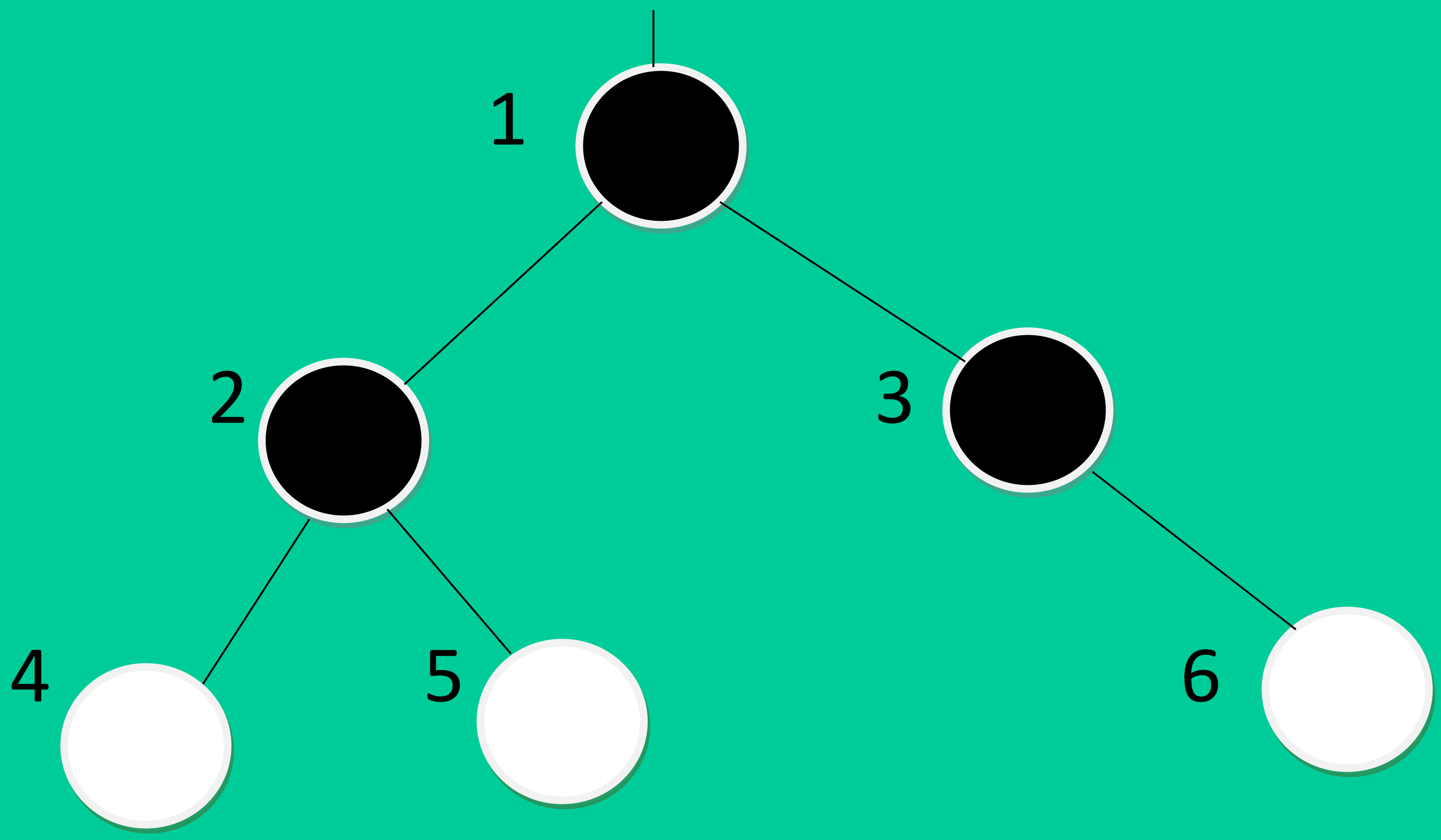
2 ←filsGauche(1) ;   4 ← filsGauche(2)

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)

<u>Important</u> :

- 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:

est-ascendant? (a, b :NOEUD) r : BOOLEEN

Pré : true

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

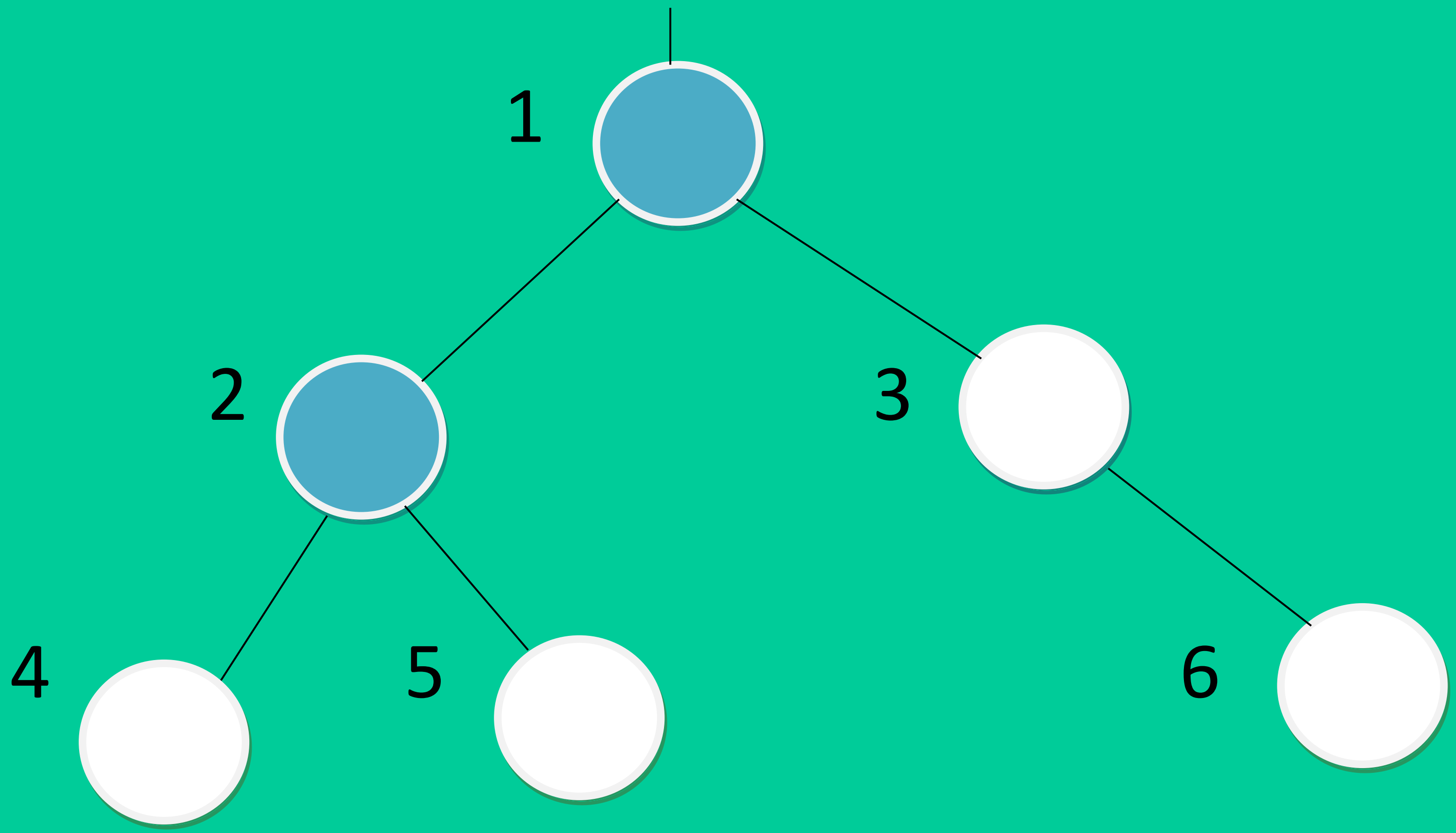
est-descendant? (a, b :NOEUD) r : BOOLEEN

Pré : true

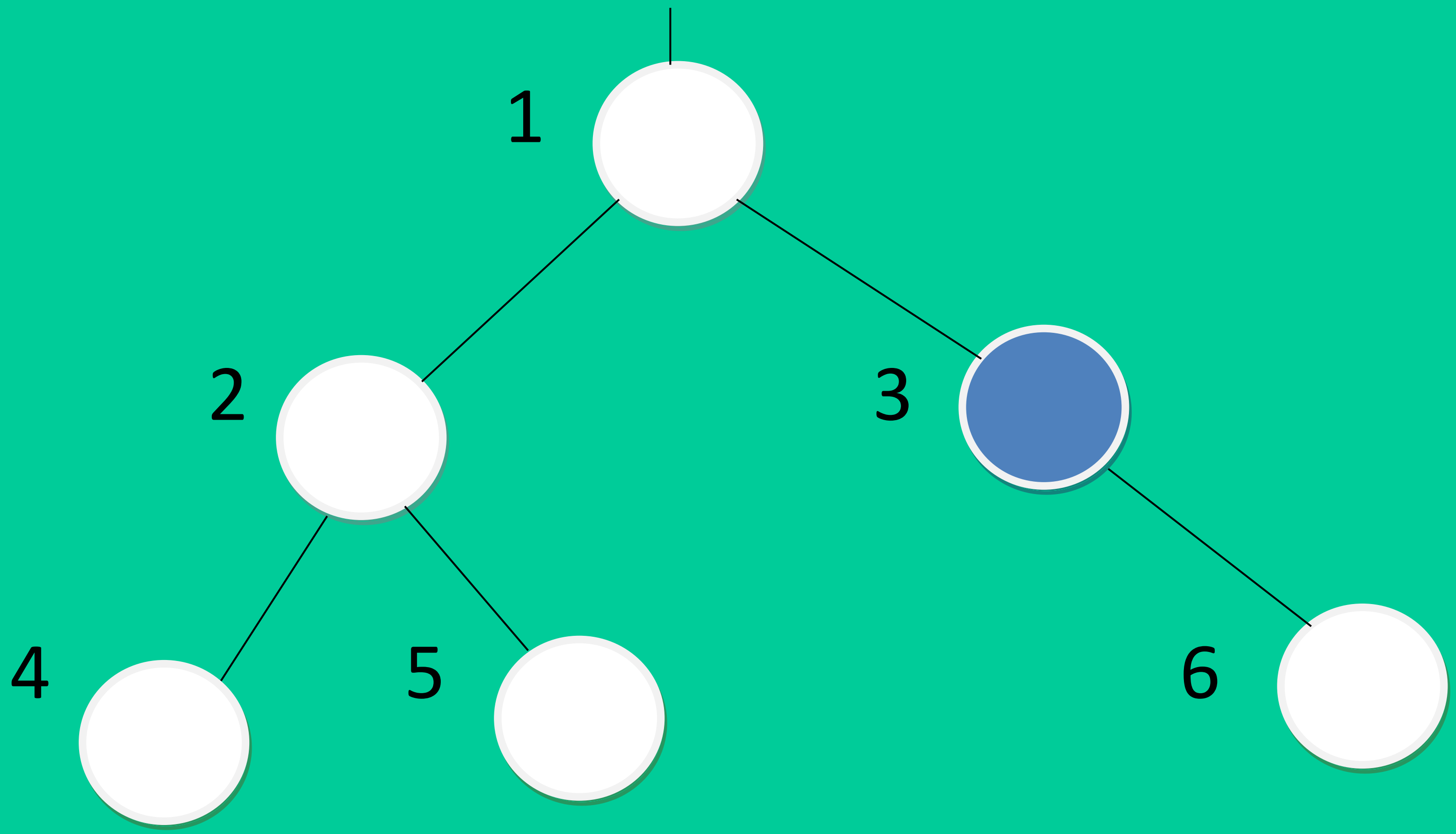
Post : r = (a = fils(b) $\vee$ est_descendant? (a,fils(b)))

Nœud interne et feuille d’un arbre

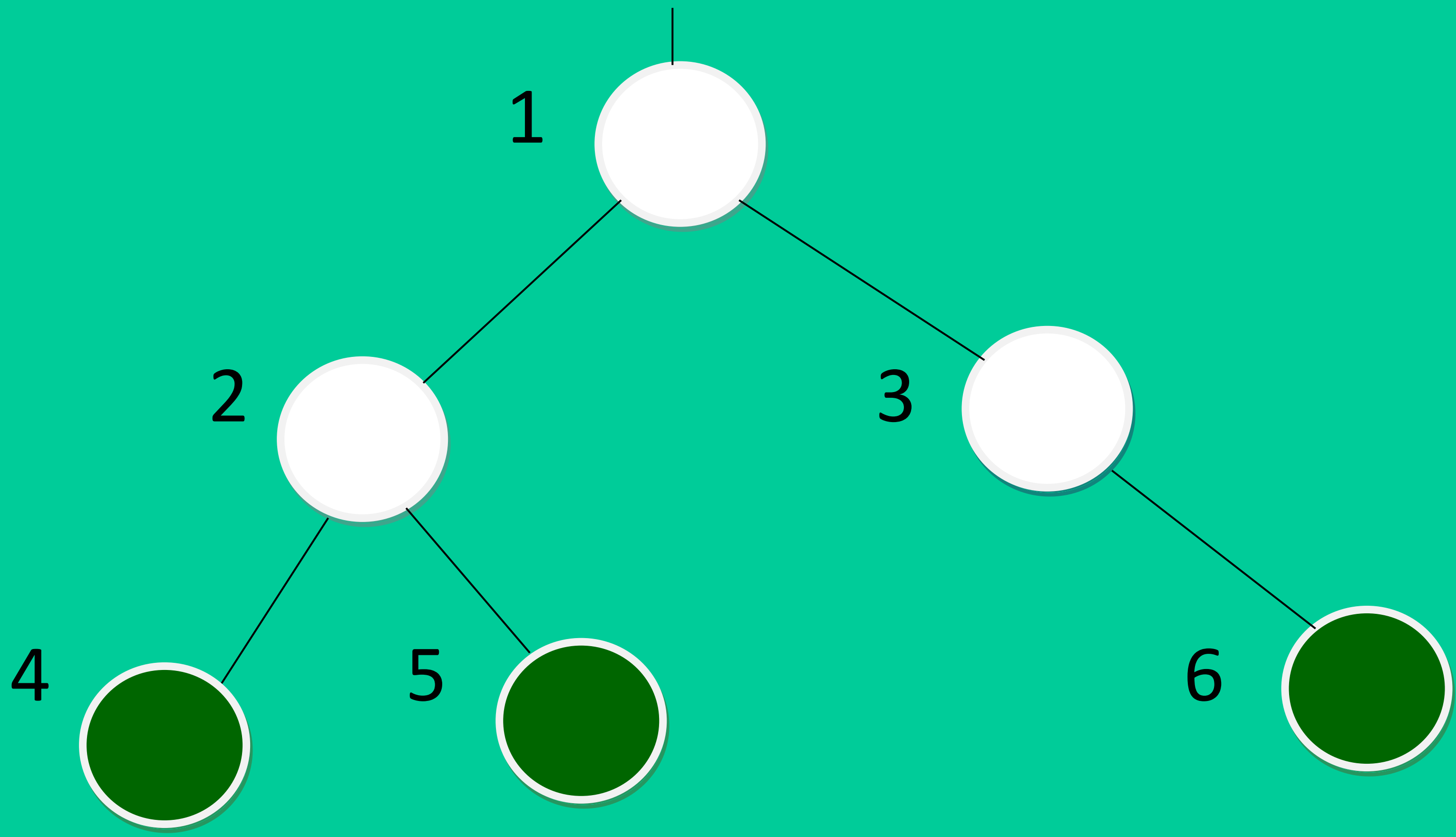
Tous les nœuds d’un arbre binaire ont **au plus deux** fils: un nœud qui a **deux** fils est appelé **nœud interne** ou **point double**.



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

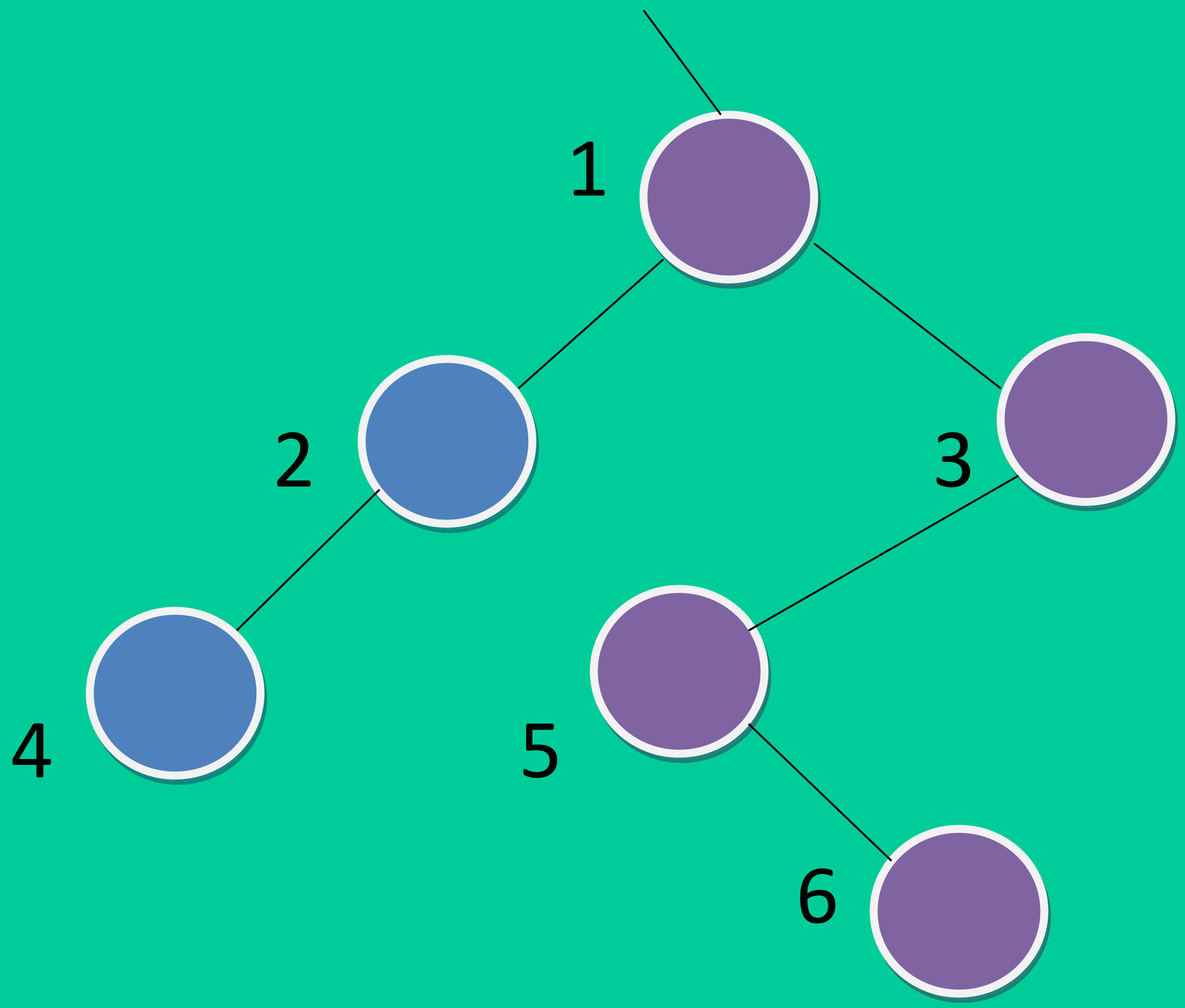


Un nœud **sans fils** est appelé **nœud externe** ou **feuille**.

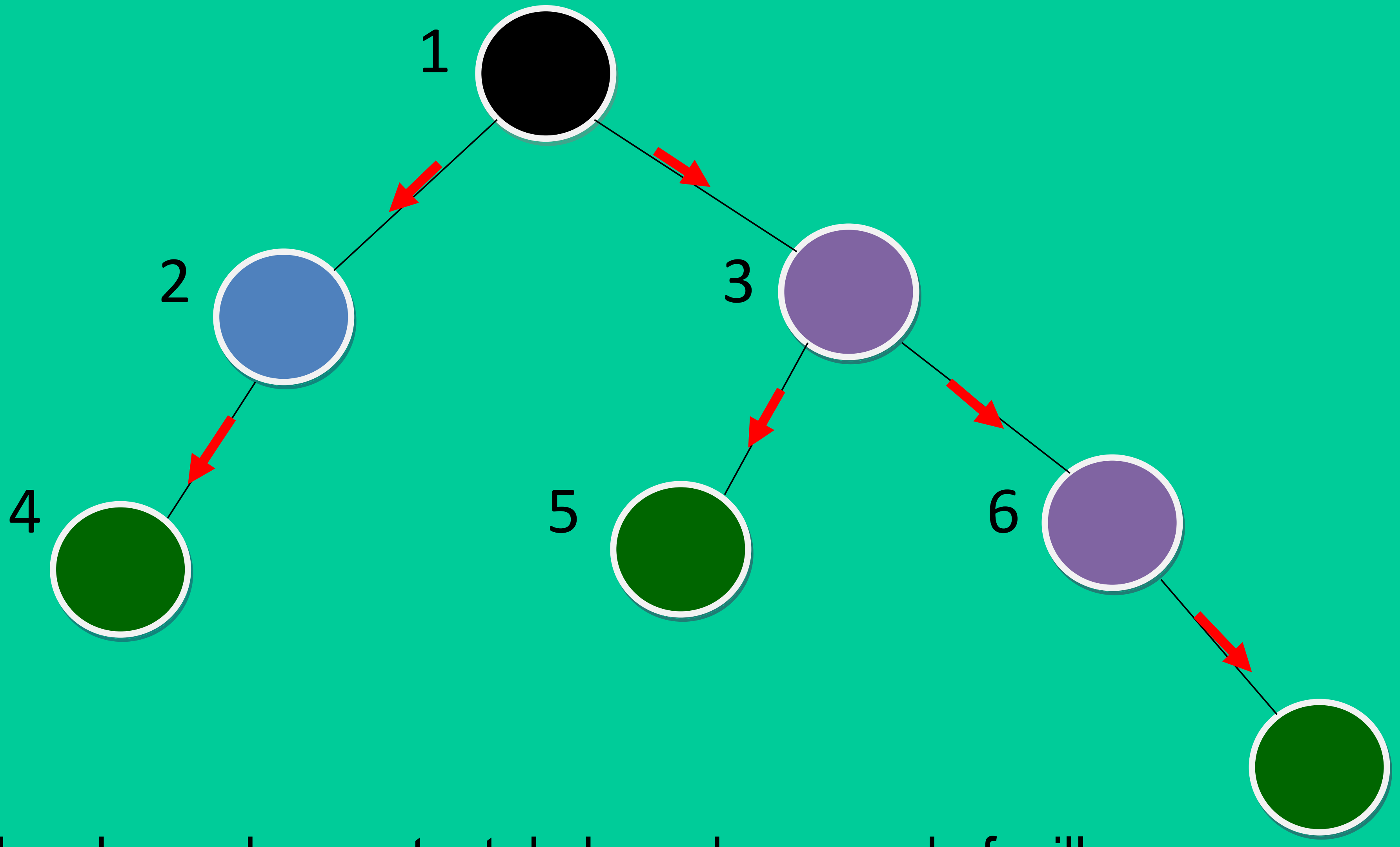


Chemin et branche d’un arbre

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

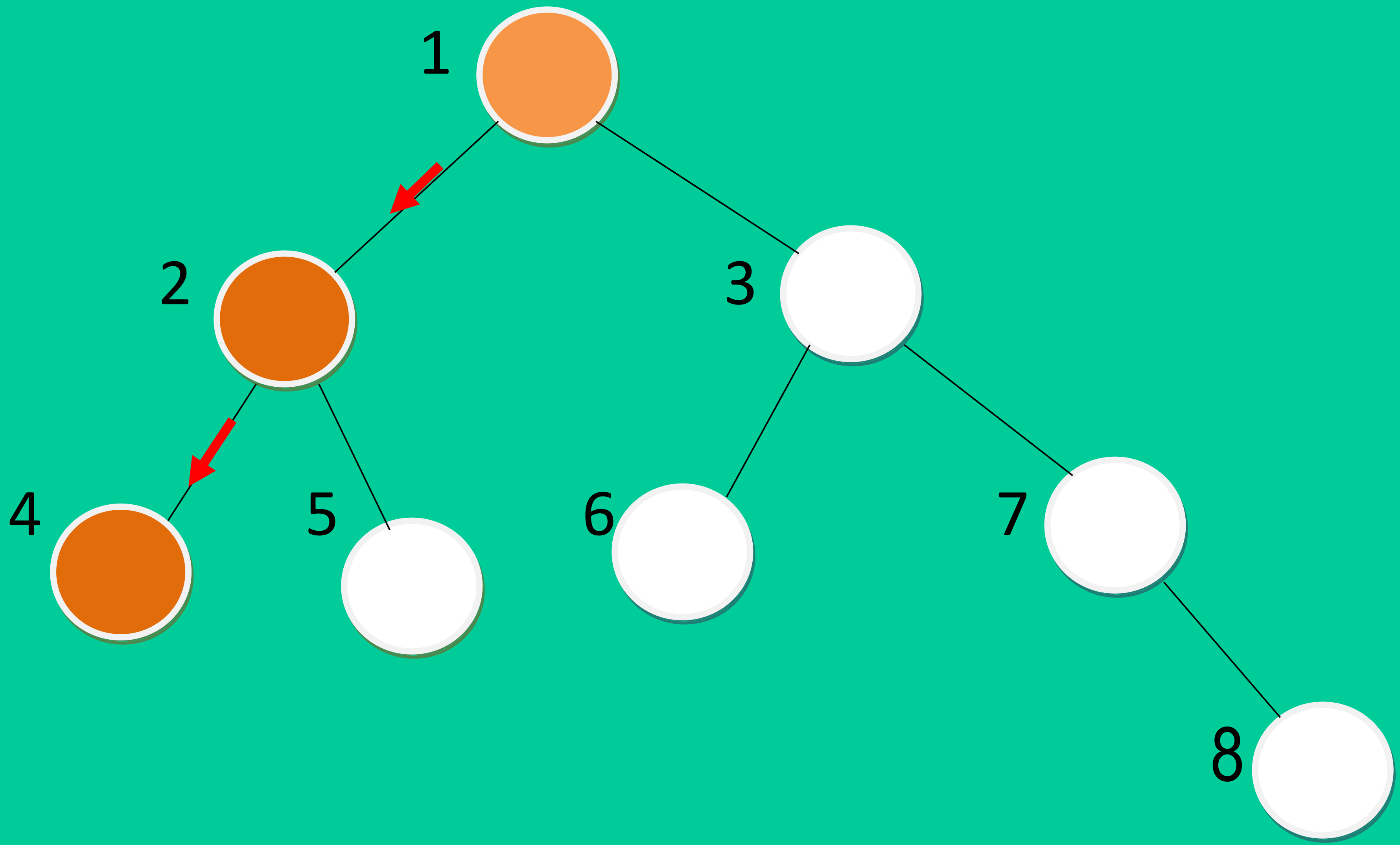


Une **branche** est un **chemin** de la **racine** à une **feuille** de B.

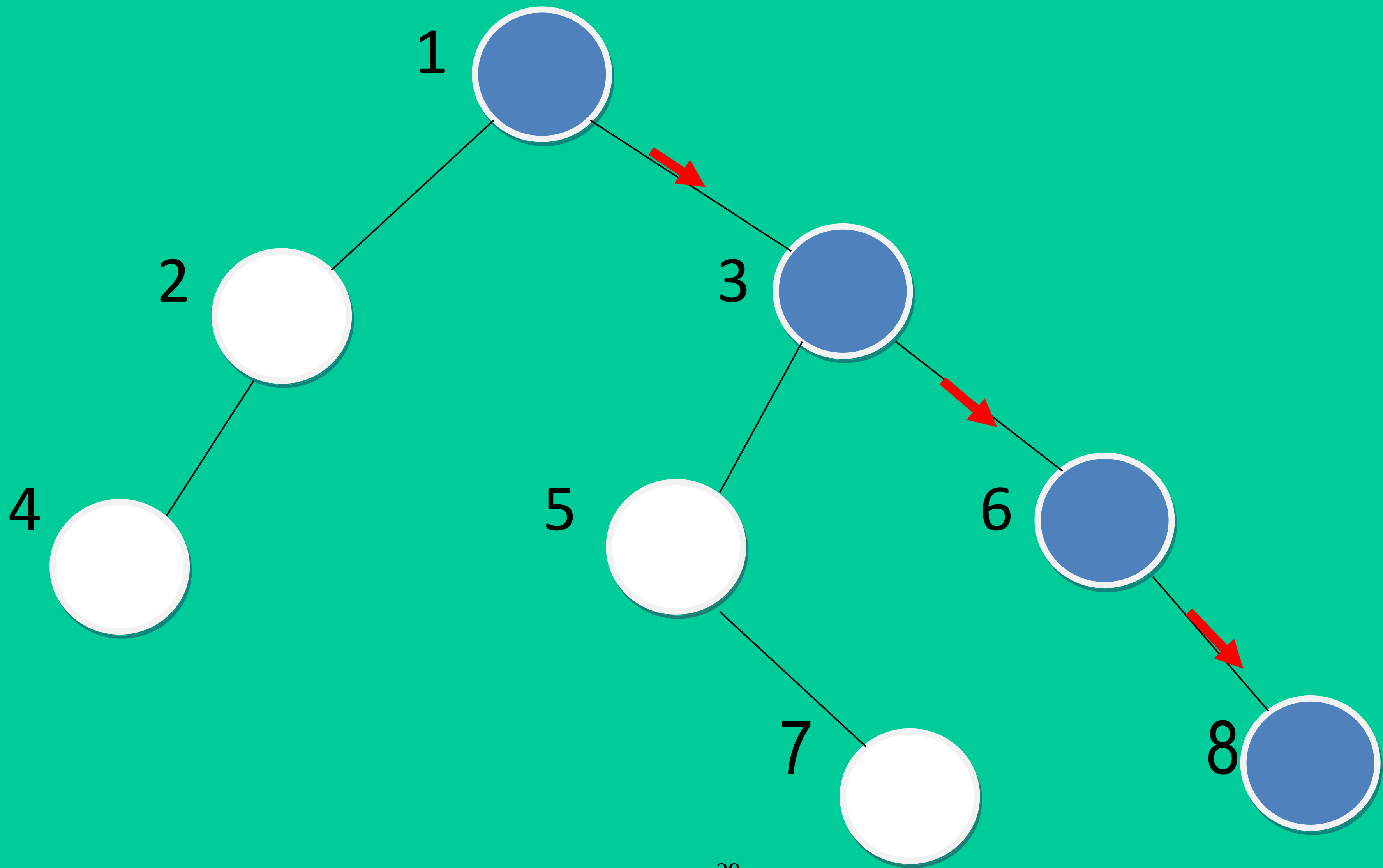


Un arbre a donc autant de branches que de feuilles.

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

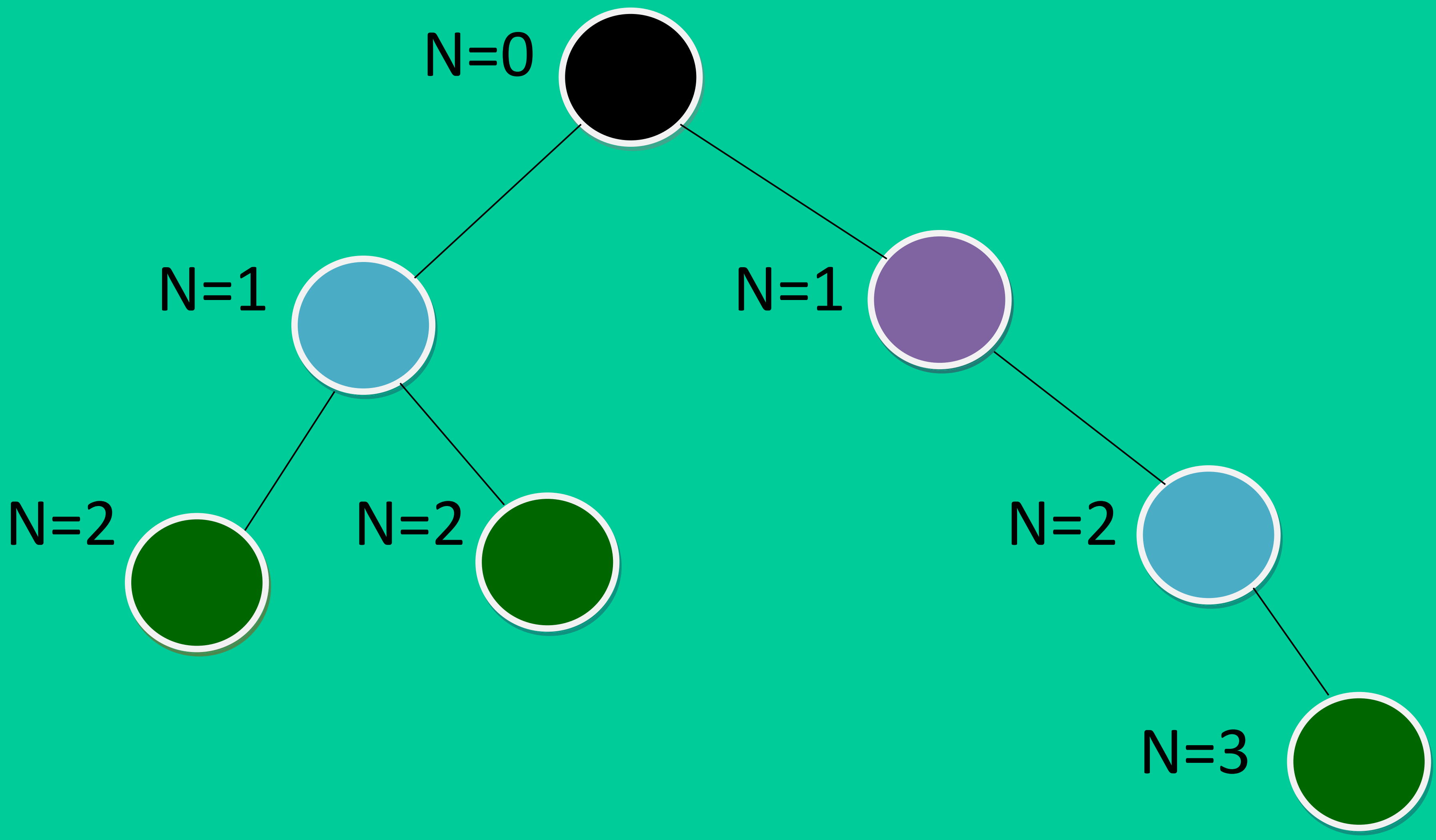
Le **niveau** d’un nœud **x** d’un arbre B est défini récursivement de la façon suivante :

niveau : NŒUD $\rightarrow$ ENTIER

avec les axiomes suivants:

niveau(racine(B)) = 0

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



N mesure le niveau du nœud.

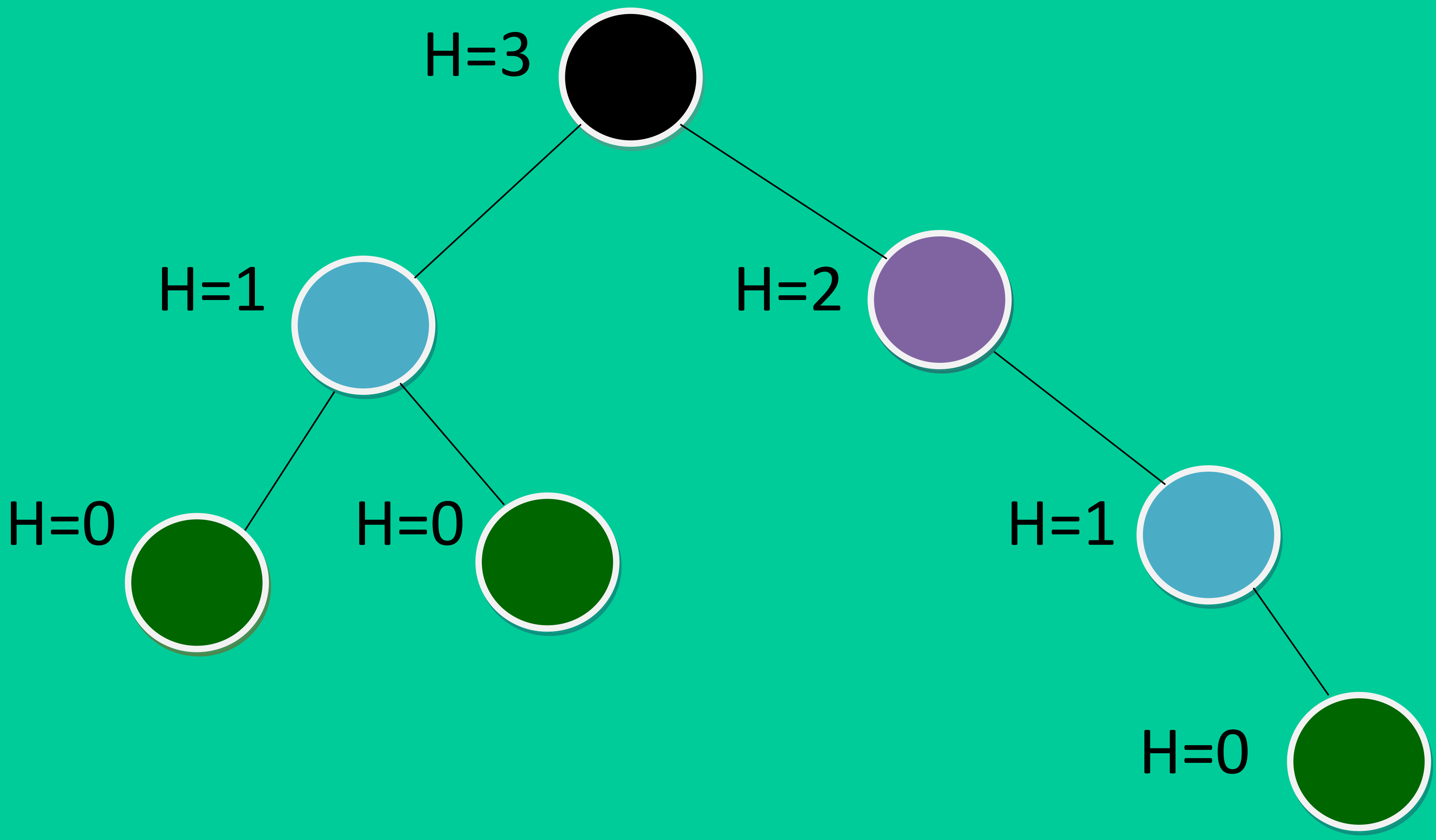
Hauteur d’un arbre

La **hauteur** ou **profondeur** 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 :

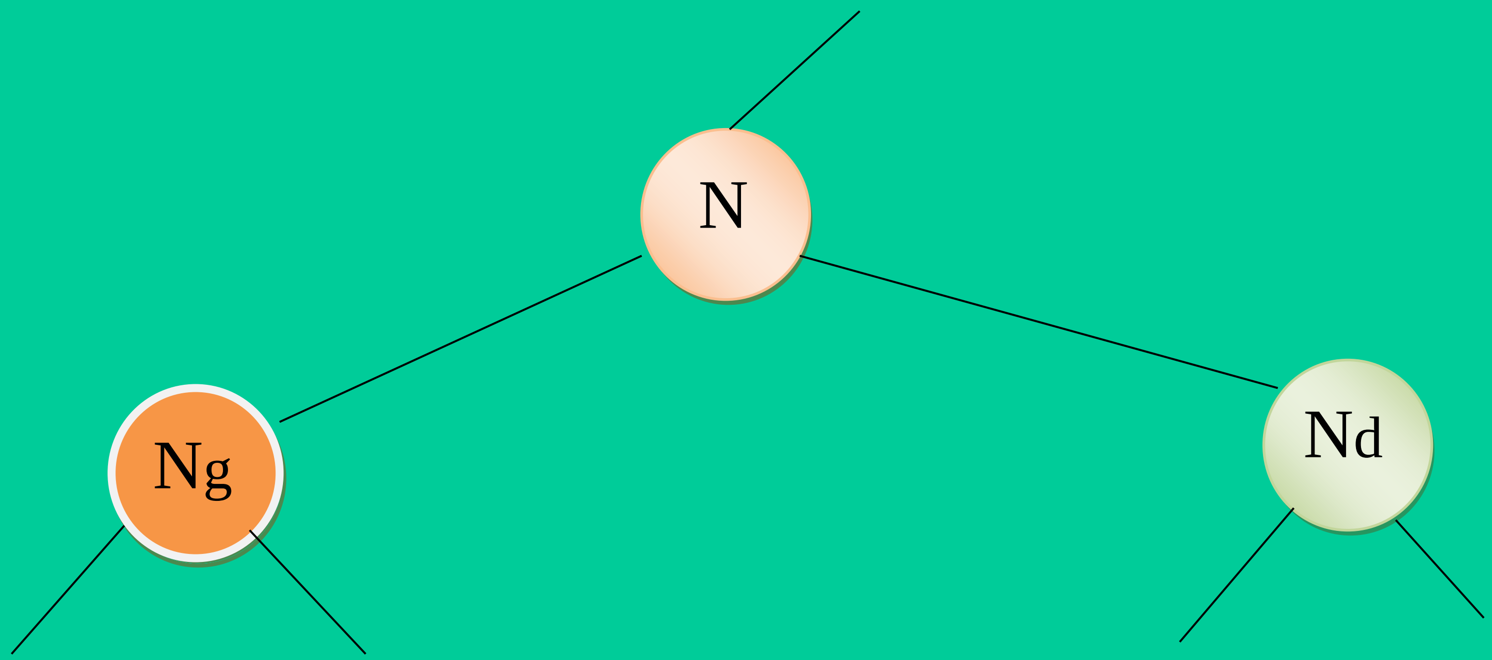
$$[\log_2 \mathbf{n}] \leq H \leq \mathbf{n-1}.$$

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

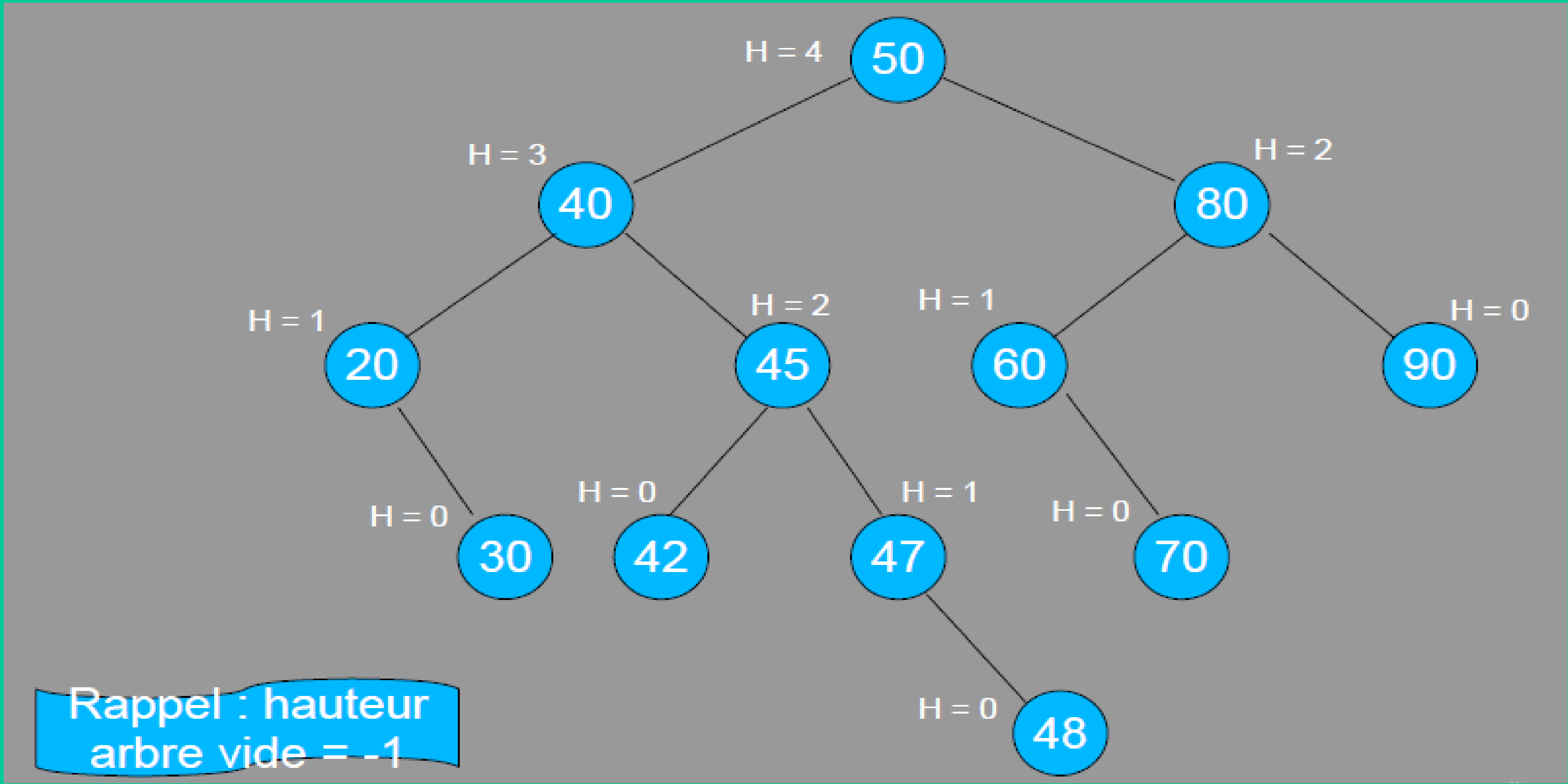
Les arbres H-équilibrés ou AVL

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

$$\forall N \in S \quad \bullet \quad | H(Ng) - H(Nd) | \leq 1$$



Exemple d'arbre H-équilibré ou AVL



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 :

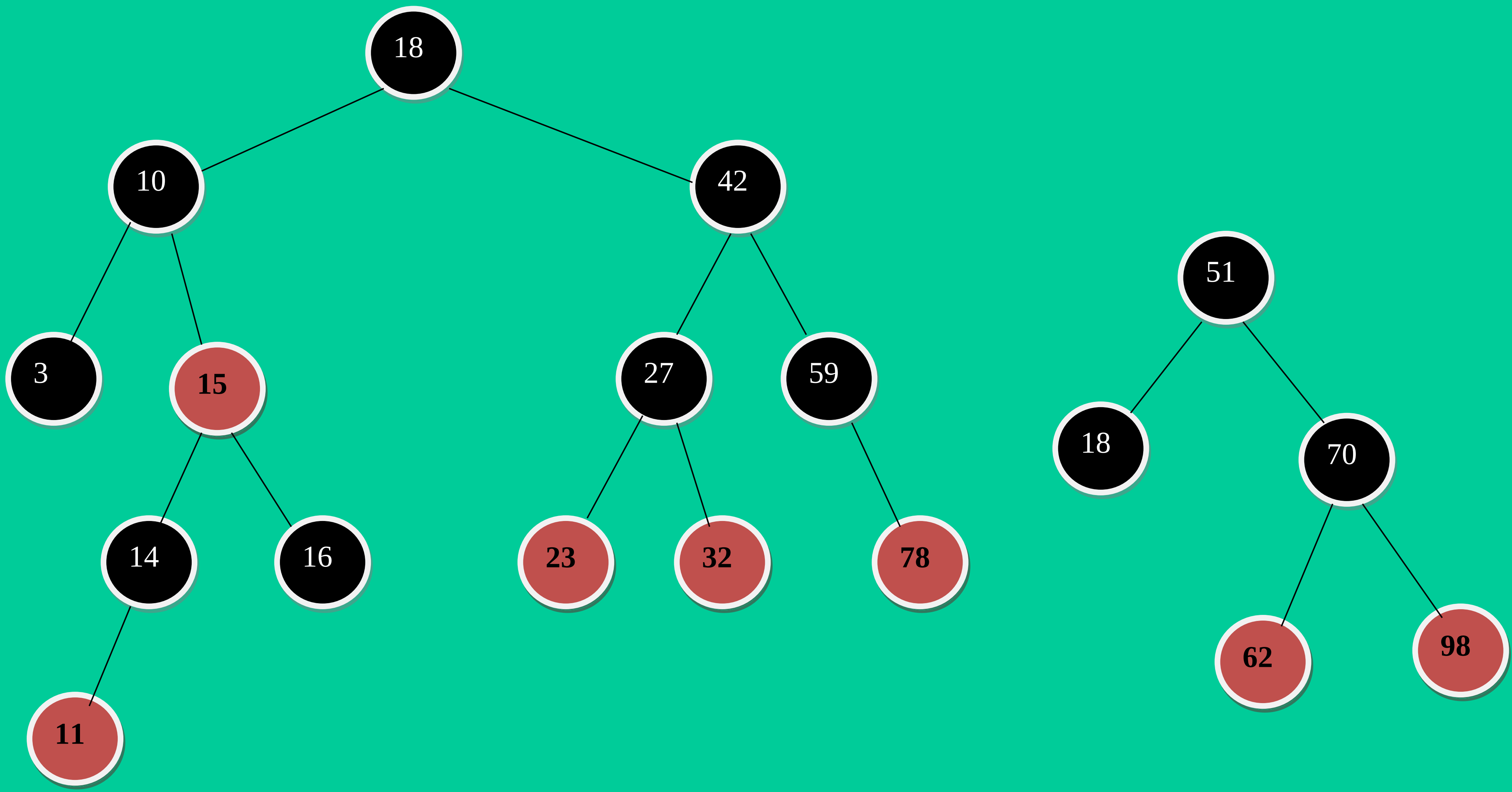
1. chaque nœud est soit rouge, soit noir;

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** :

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

III- PARCOURS D'UN ARBRE BINAIRE

Parcourir un arbre consiste à atteindre:

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

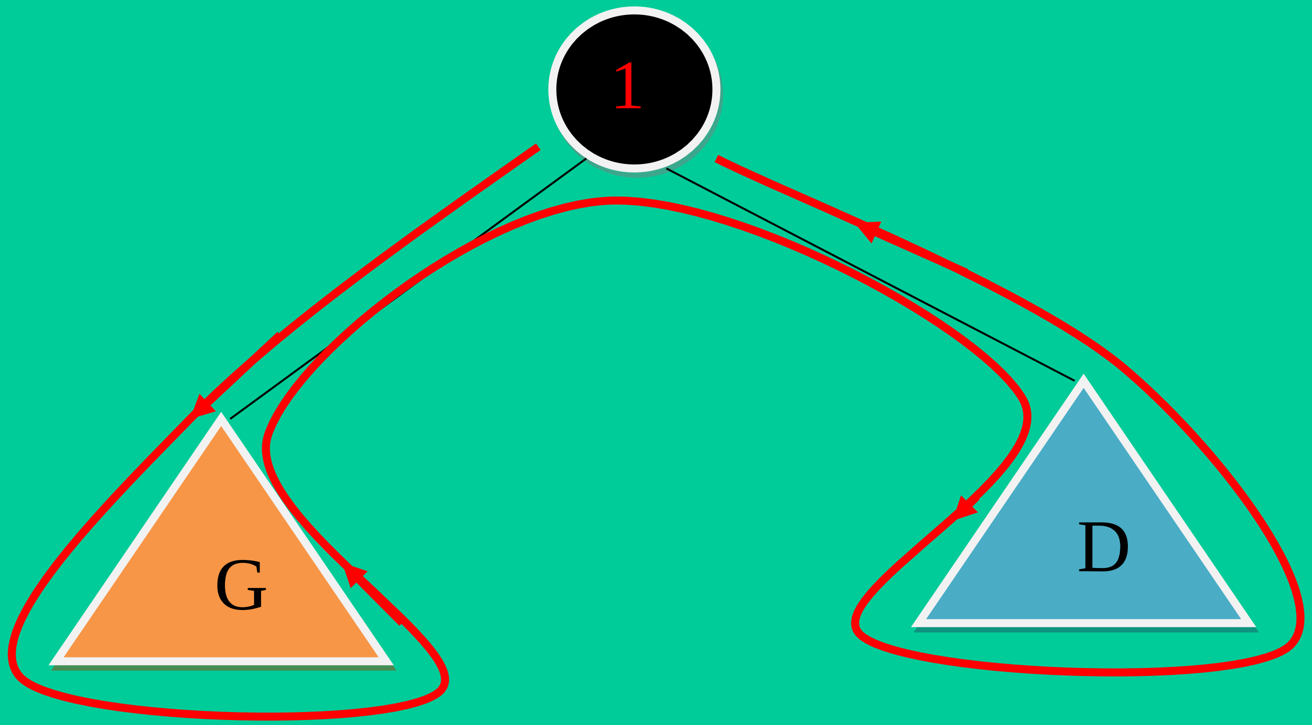
chacun des nœuds de l'arbre pour y effectuer le même traitement.

1 – Parcours en profondeur à «main gauche»

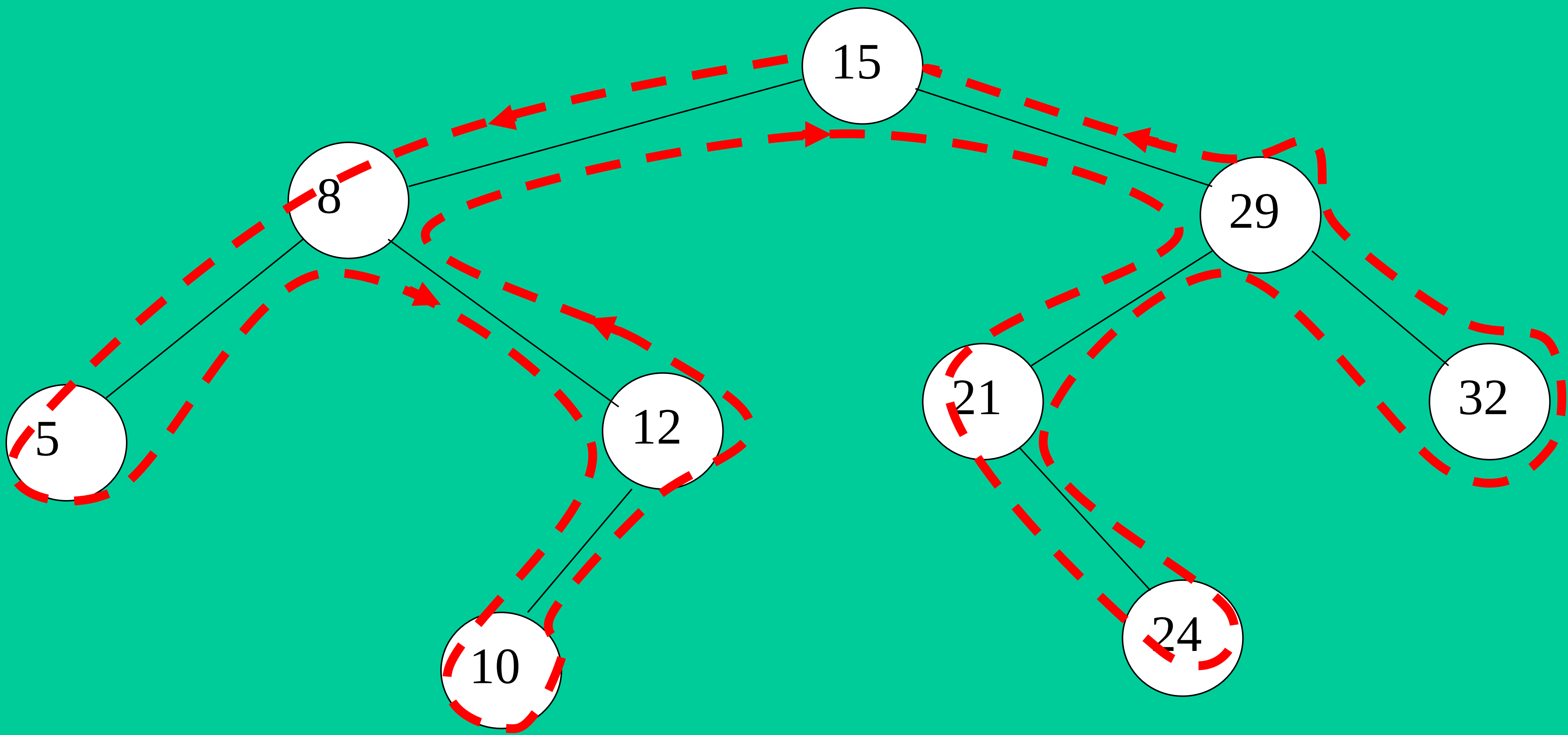
L'opération consiste à tourner autour de l'arbre en suivant le chemin qui:

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

Principe de parcours main gauche

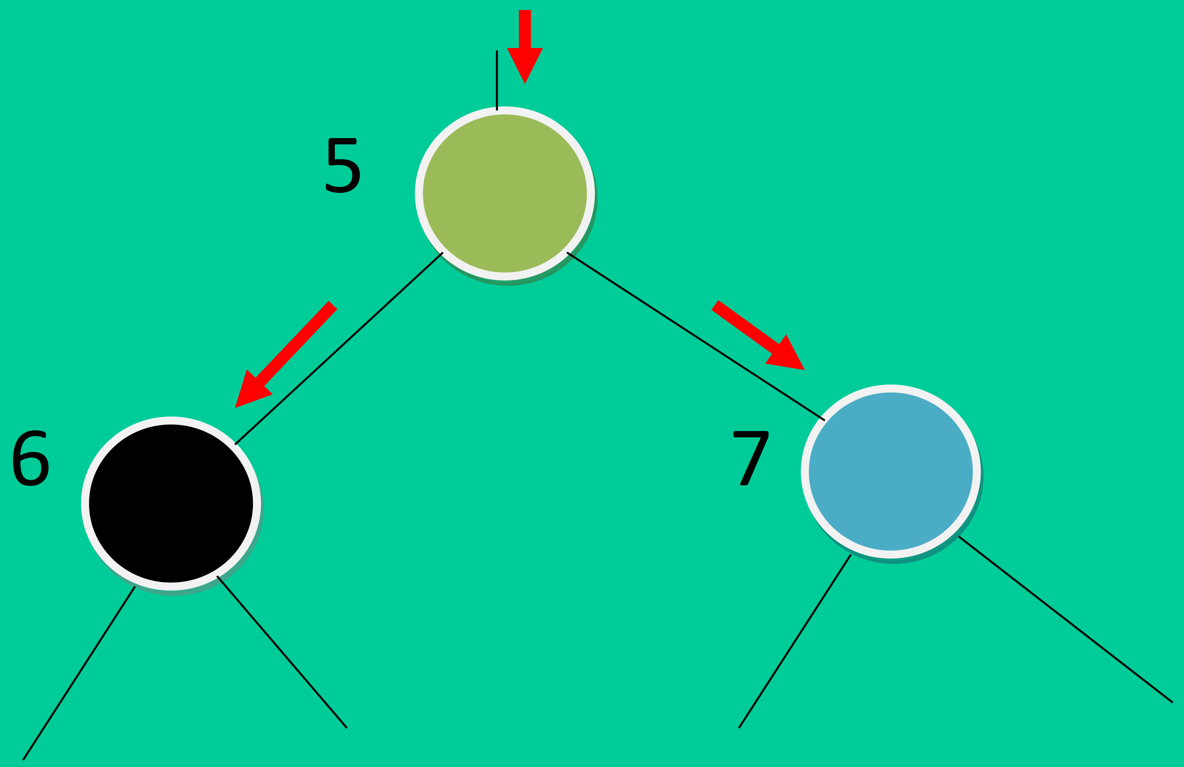


Exemple de parcours main gauche



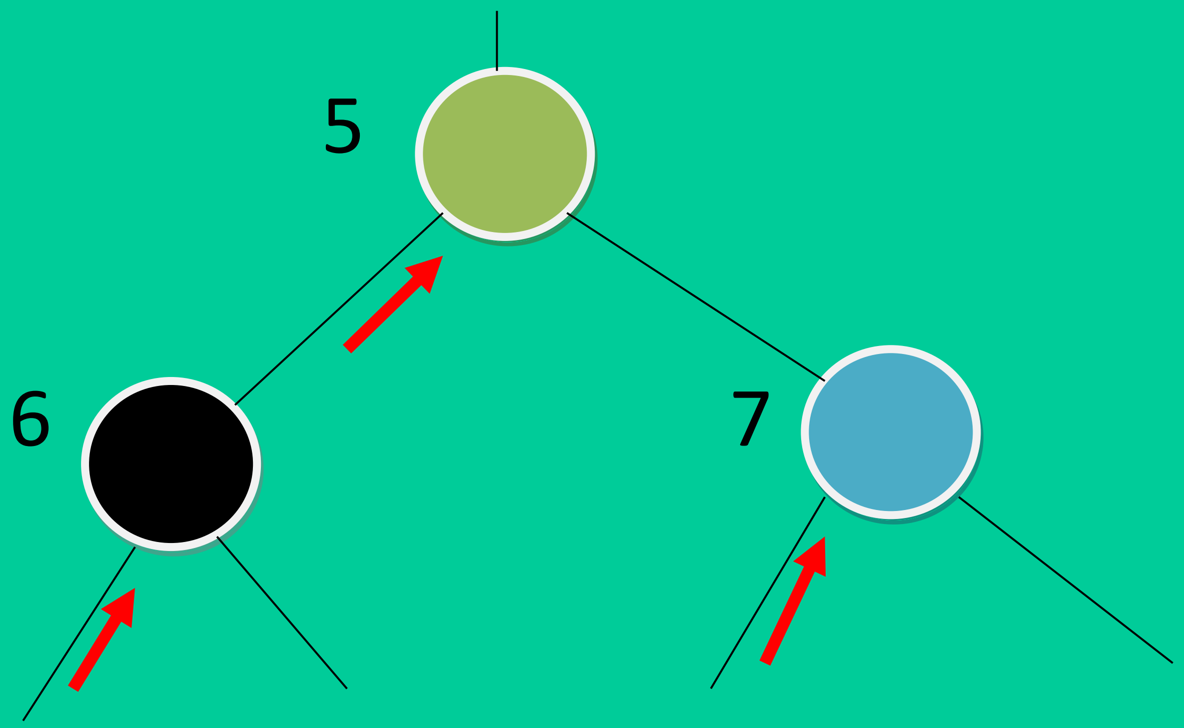
Dans ce parcours, chaque nœud de l’arbre est rencontré **trois** fois:

a)- d’abord à la **«descente»**,



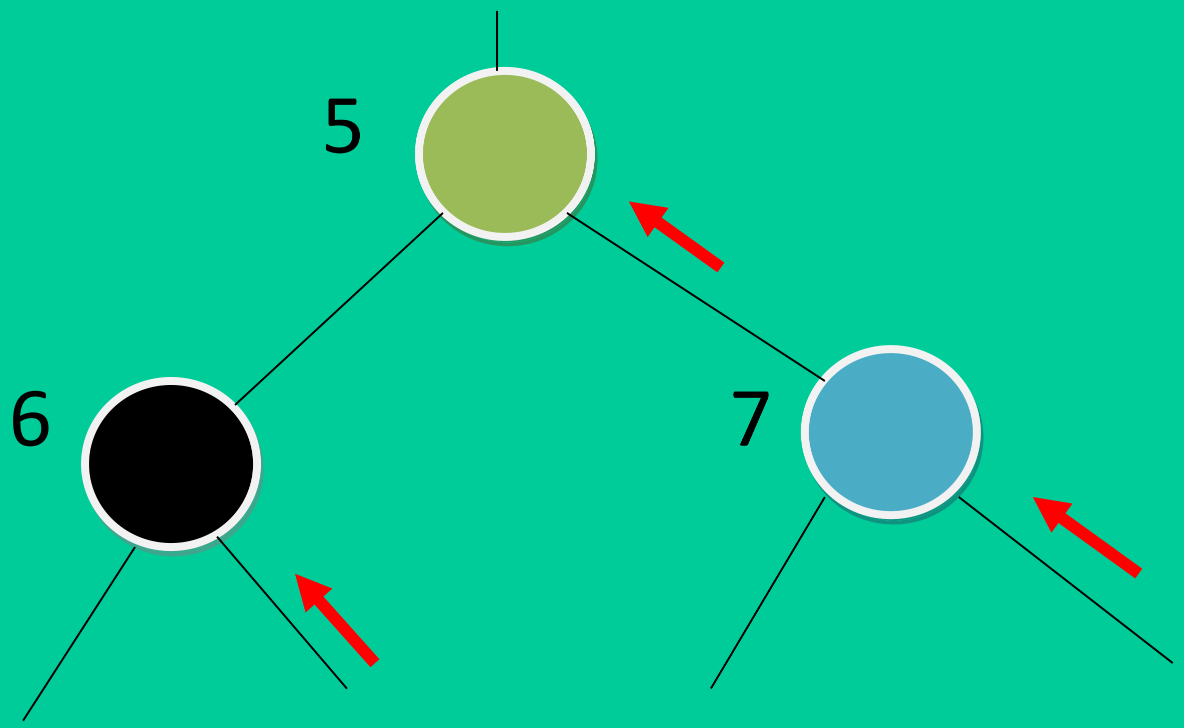
Il induit un parcours de l’arbre en **ordre préfixe**.

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



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

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



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

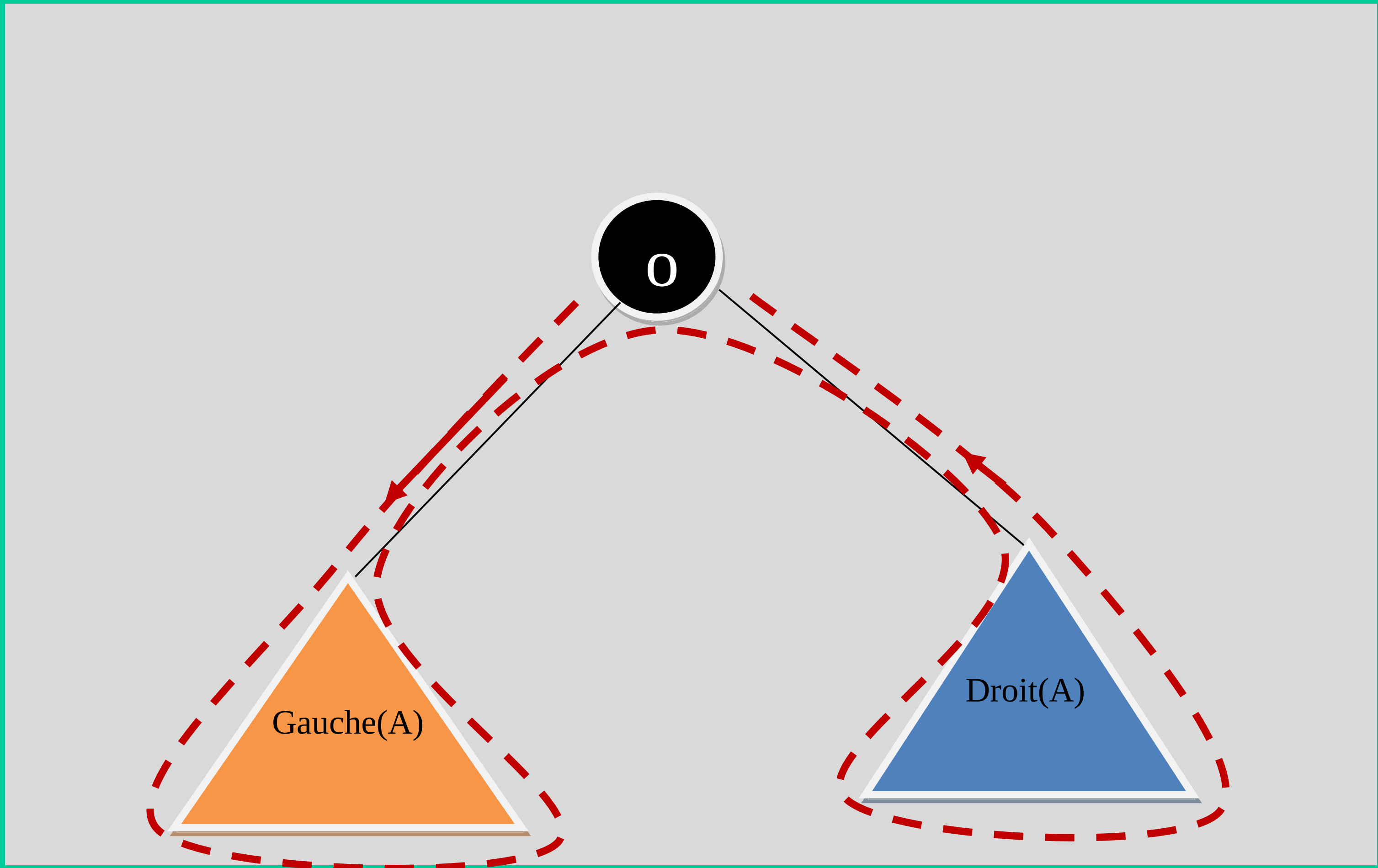
2 – Algorithme de parcours

On peut faire correspondre à chacune des rencontres du nœud un traitement particulier.

Notons T_1, T_2 et T_3 ces traitements.

- T_1 : traitement à la «descente»,
- T_2 : traitement en «montée gauche»,
- T_3 : traitement en «montée droite».

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



Arbre A :parcours en profondeur «main gauche»

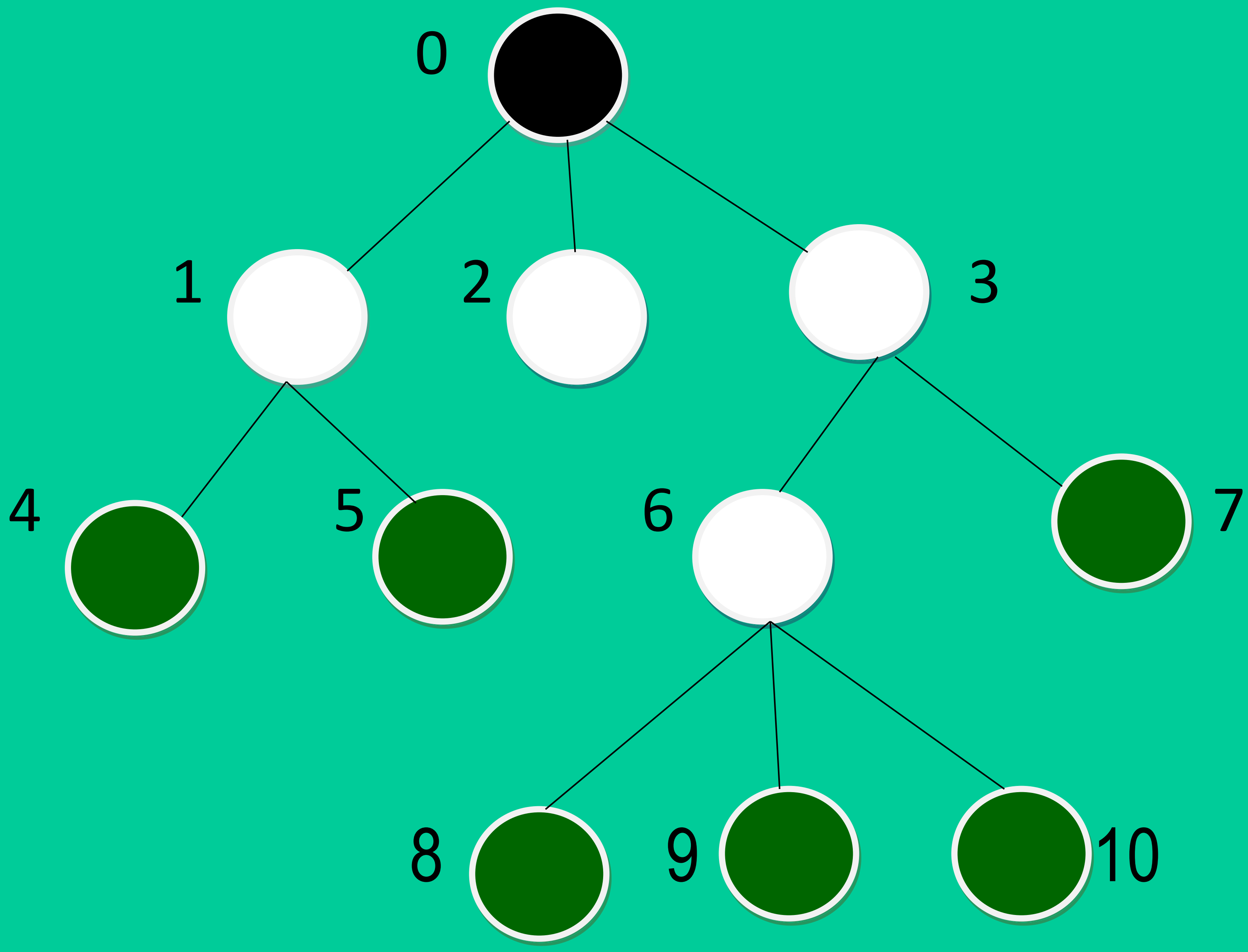
```
profondeur(ARBRE B) /* Parcours en profondeur « main gauche» d'un arbre binaire B
*/
    {
    if( B = = arbreVide() )
        T0 ( ) ; /* traitement spécial d'un arbre vide */
        else
            {
            T1( ) ; /* traitement à la descente */
            profondeur (gauche(B)) ;
            T2 () ; /* traitement en «montée gauche» */
            profondeur (droit(B)) ;
            T3 () ; /* traitement en «montée droite» */
            }
    }
```

IV- ARBRE GENERAL

Pourquoi l’arbre général ?

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

- Dans certains cas le nombre de fils de chaque nœud n’est plus limité à deux.


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

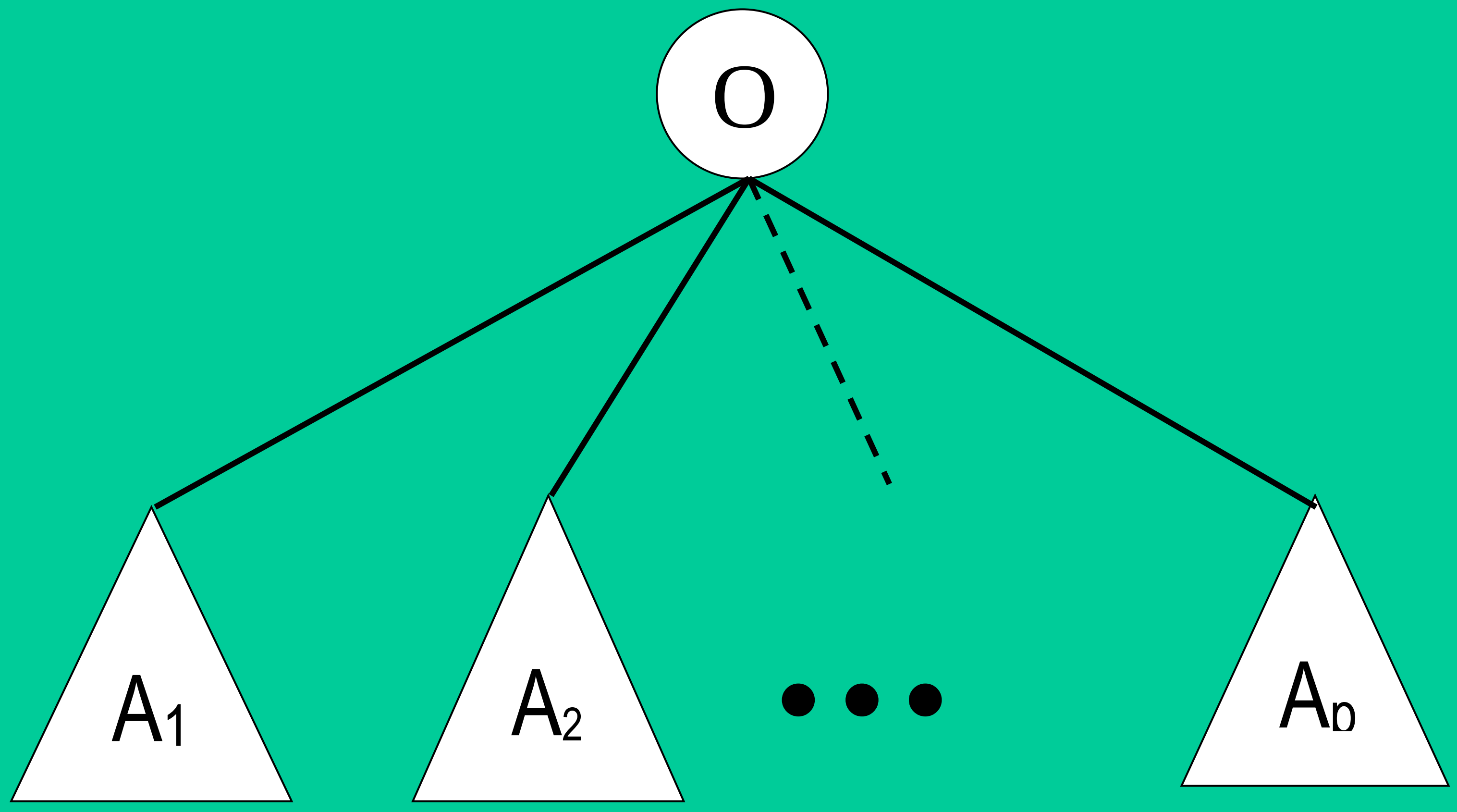
Un **arbre A** est la donnée :

-d'une **racine**: O,

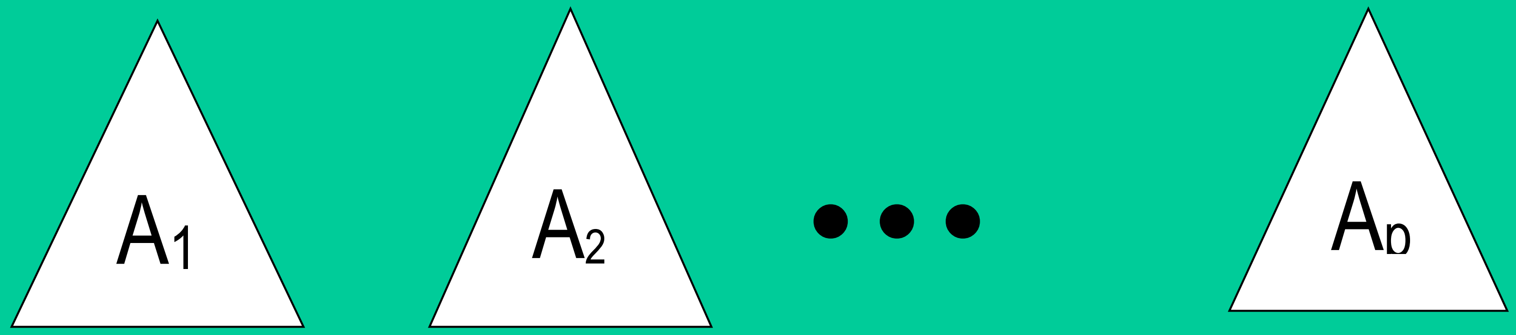
-et d'une **liste finie d'arbres disjoints** : [A_1 ,…,A_p]

On note :

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



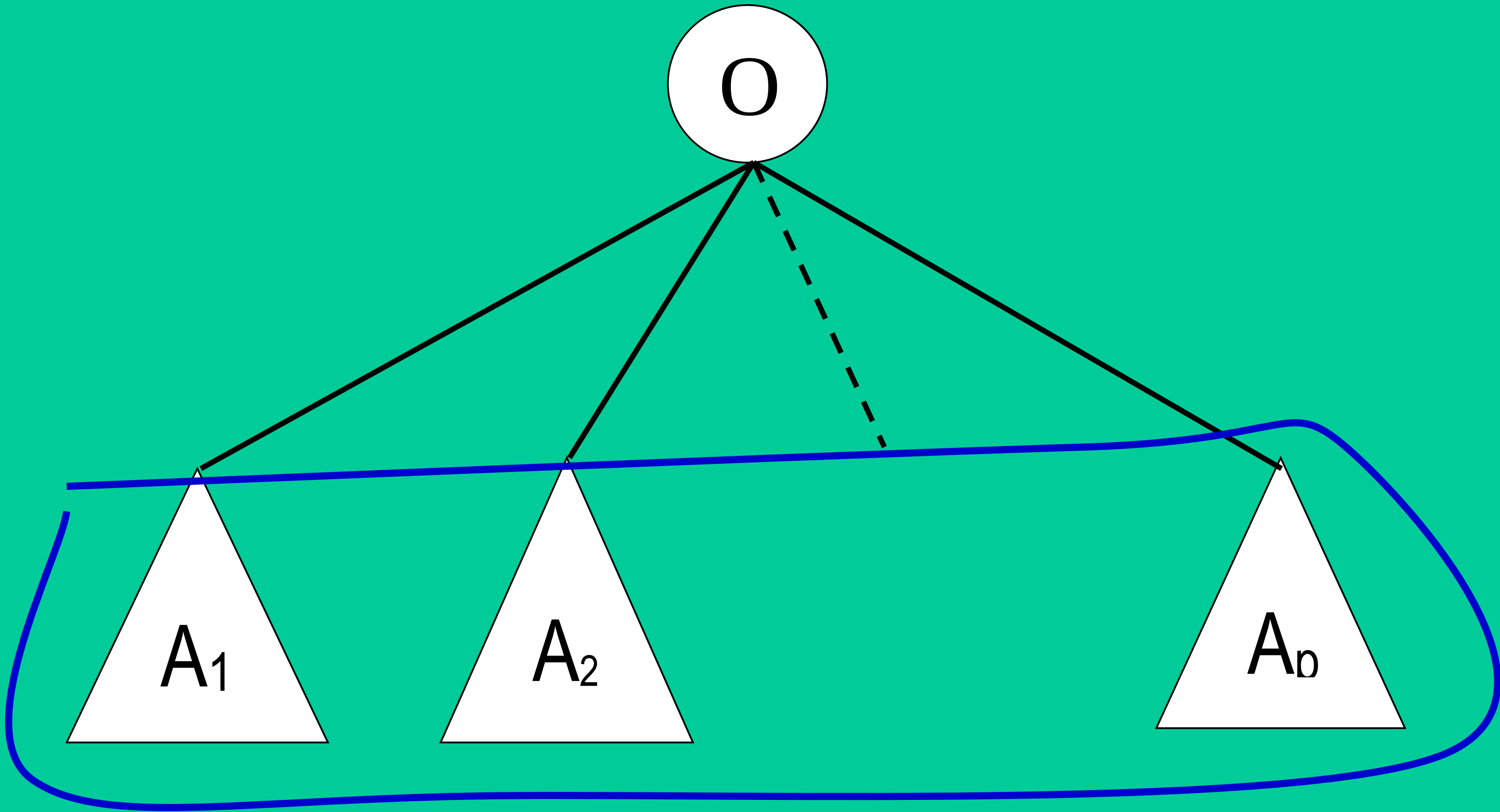
Une **liste** finie d'arbres **disjoints** est appelée une **forêt**.



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 :

$$A = < o, F >$$

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

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

- **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)

- **estArbre Vide**(**arbreVide**)
- $\neg$**estArbreVide**(**construire**(o, F))

- **estForetVide**(**foretVide**)
- $\neg$**estForetVide**(**planter**(A,i,F))

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

- **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)

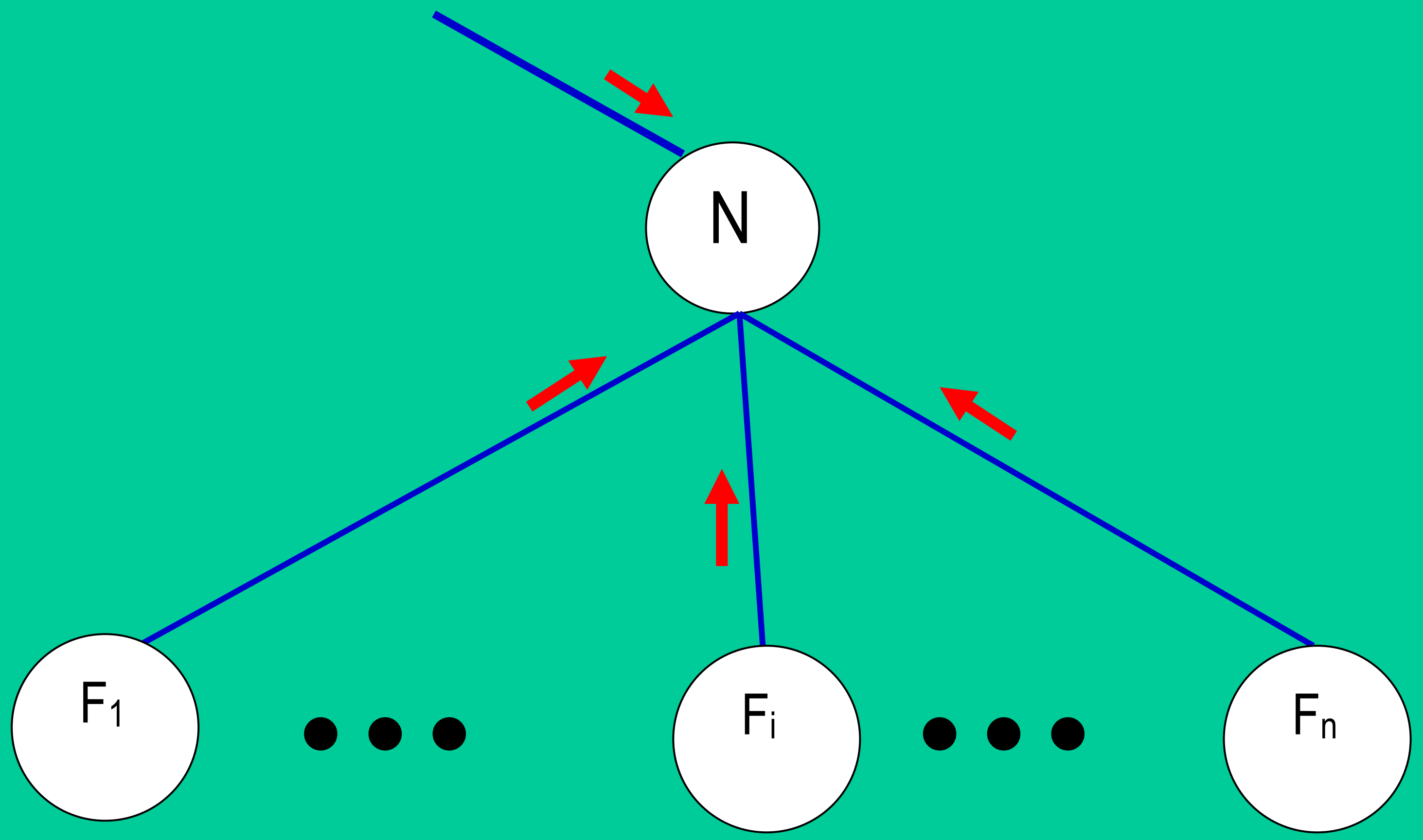
end

V- PARCOURS D'ARBRE

On peut généraliser le parcours en profondeur «main gauche » 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.



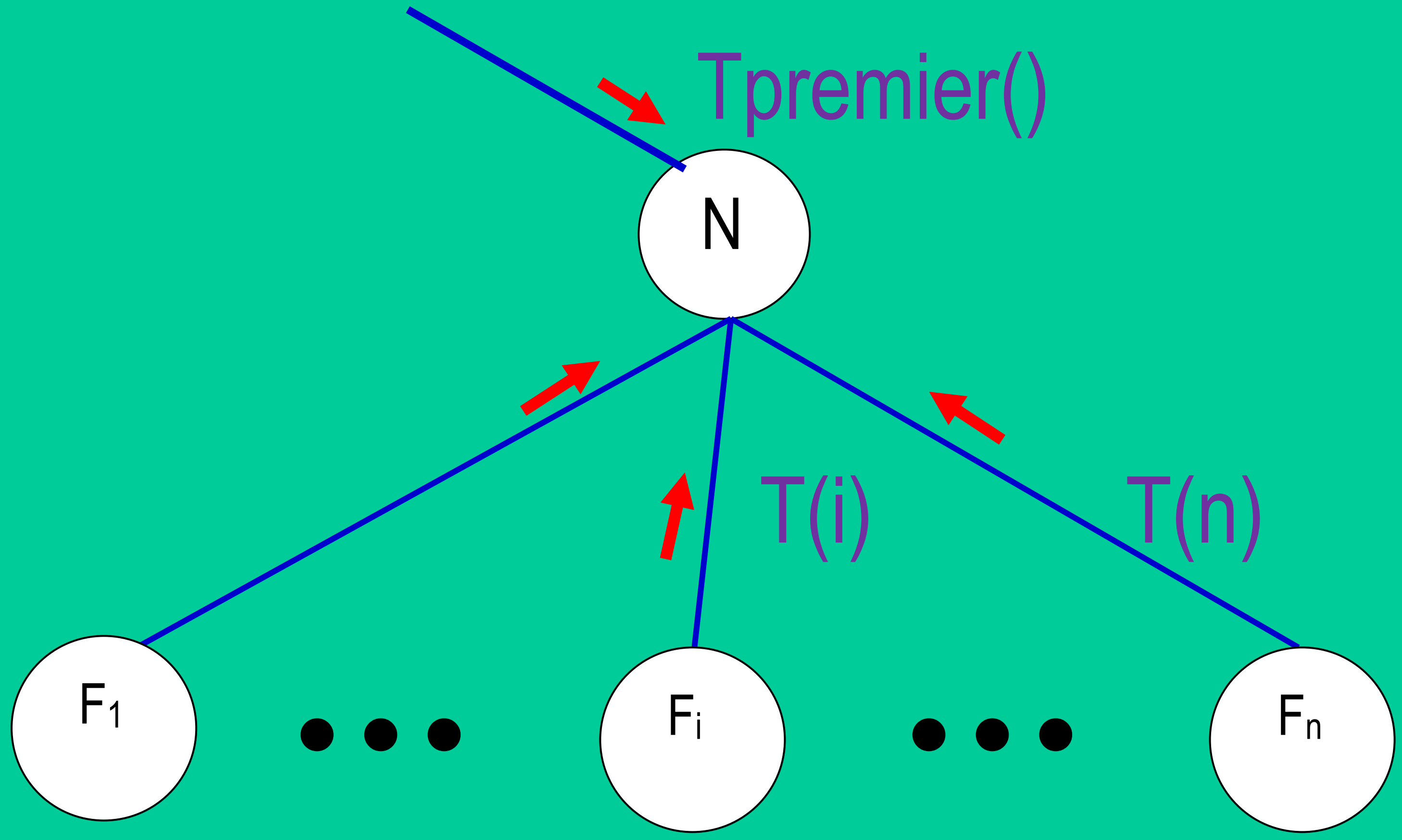
On note, si le nombre de fils est **n** :

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**.

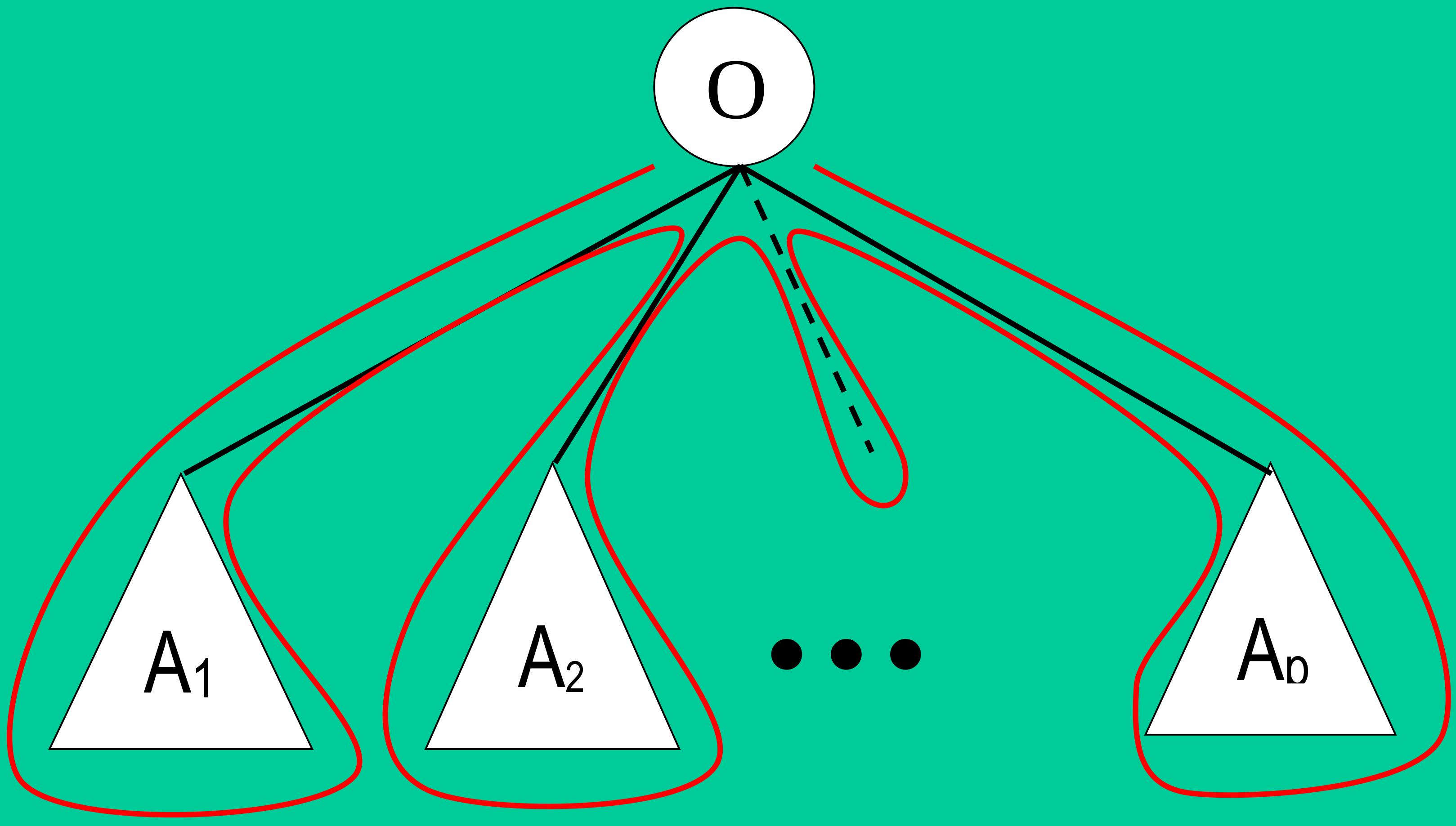

Tpremier()
N
T(i)
T(n)
F_1
F_i
F_n

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
An

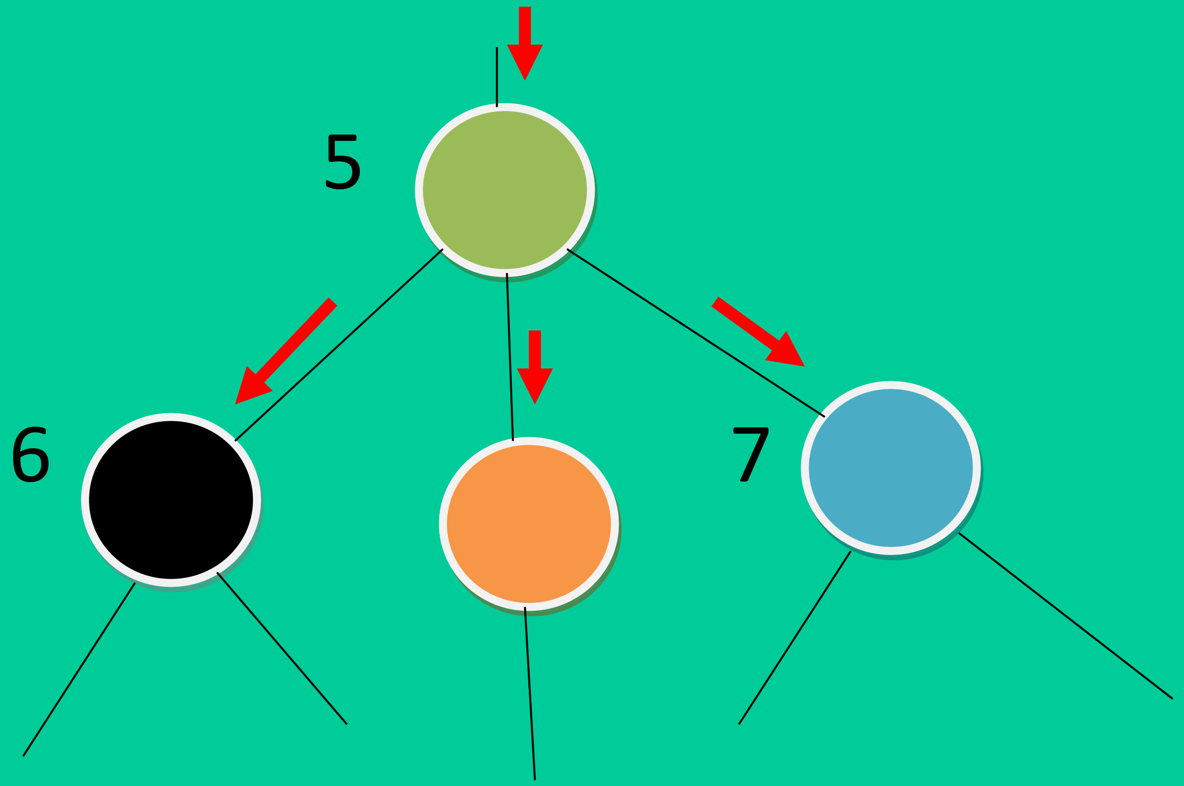
Ordre d’exploration d’un arbre

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

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

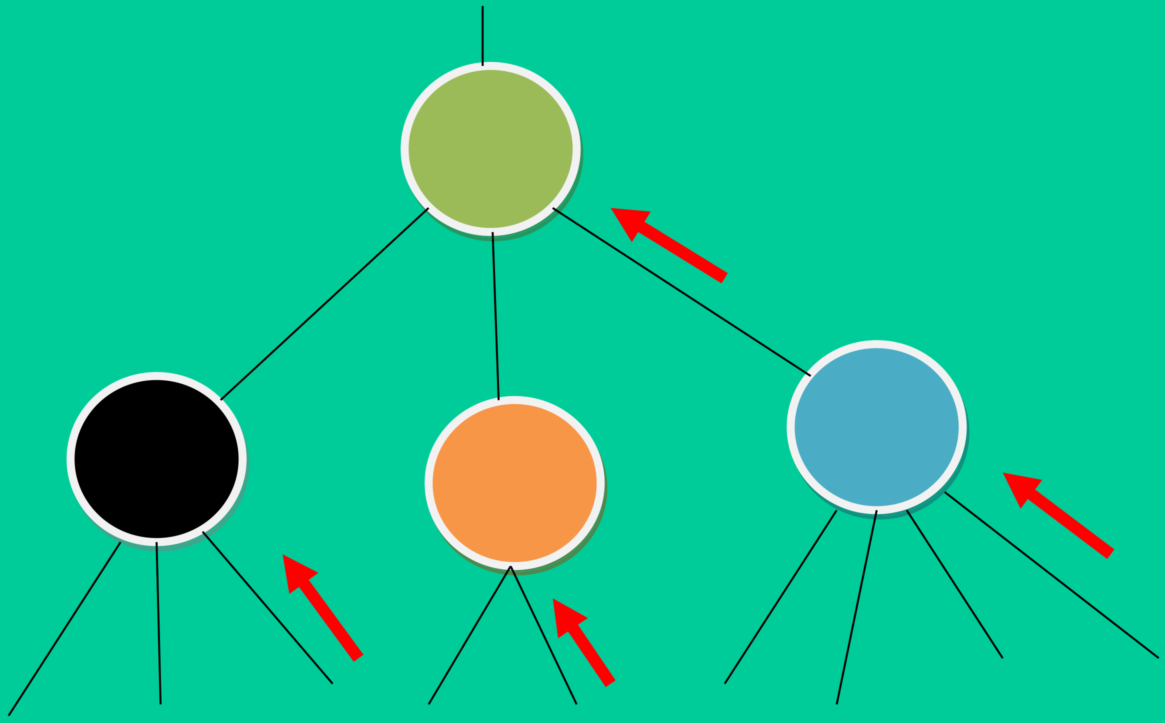
1-l’ordre préfixe:

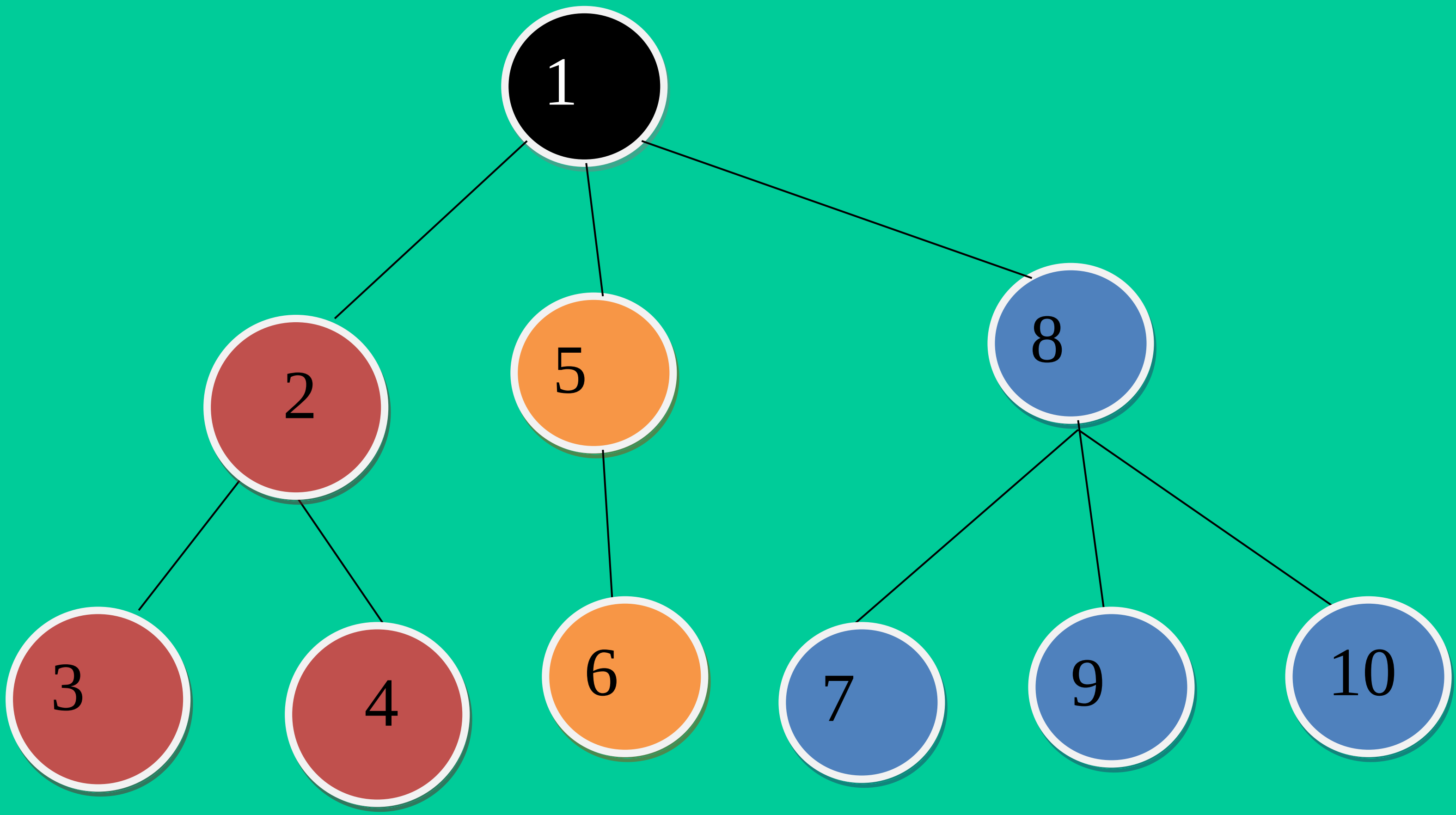
Chaque nœud n’est pris en compte que lors du **premier passage**.



2-l'ordre suffixe:

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

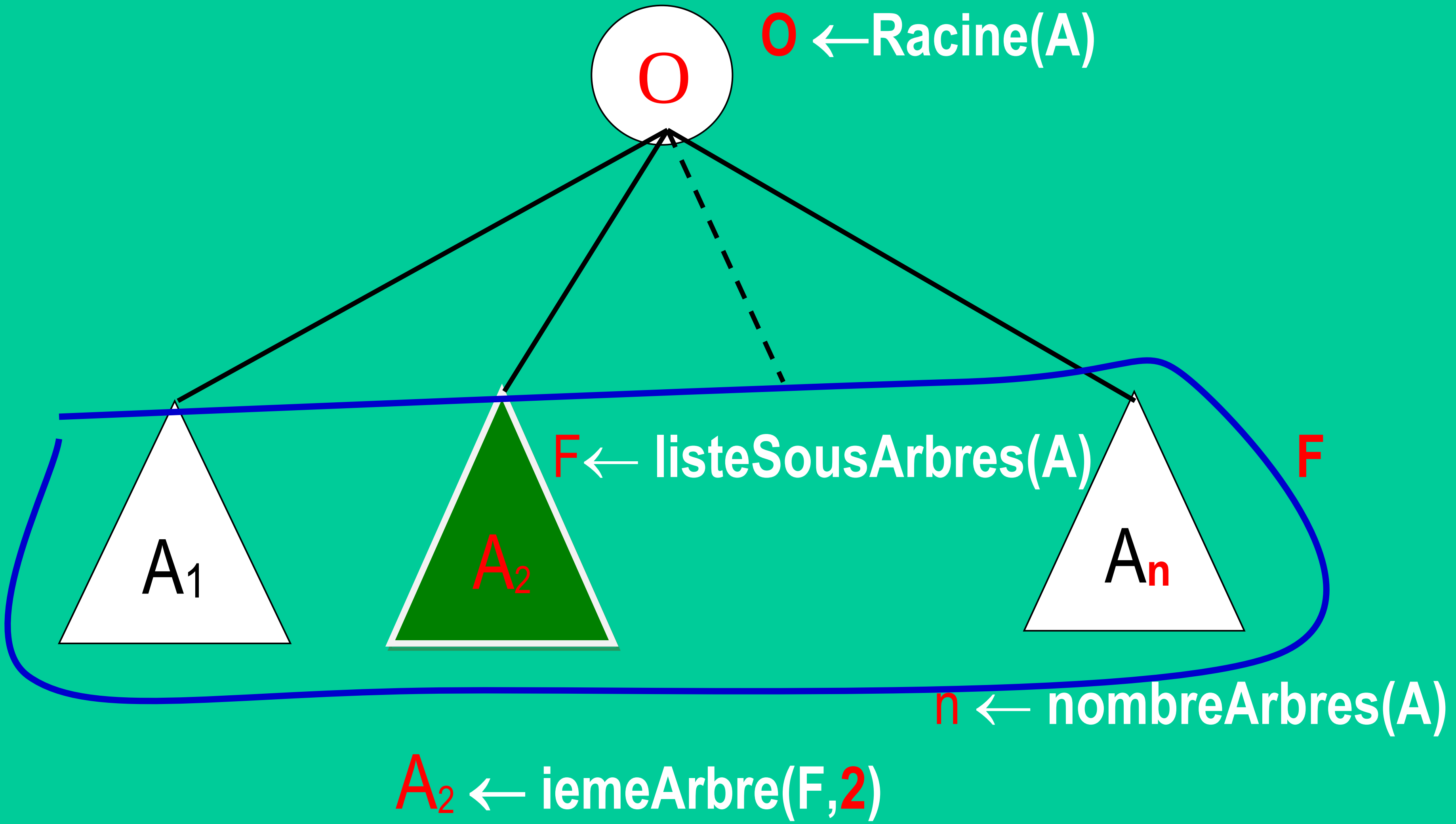




Ordre préfixe : **1 2 3 4 5 6 8 7 9 10**

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

Rappel :


O
O ←Racine(A)
F← listeSousArbres(A)
F
A1
A2
An
n ← nombreArbres(A)
A2 ← 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 */
          }
 }
}
```

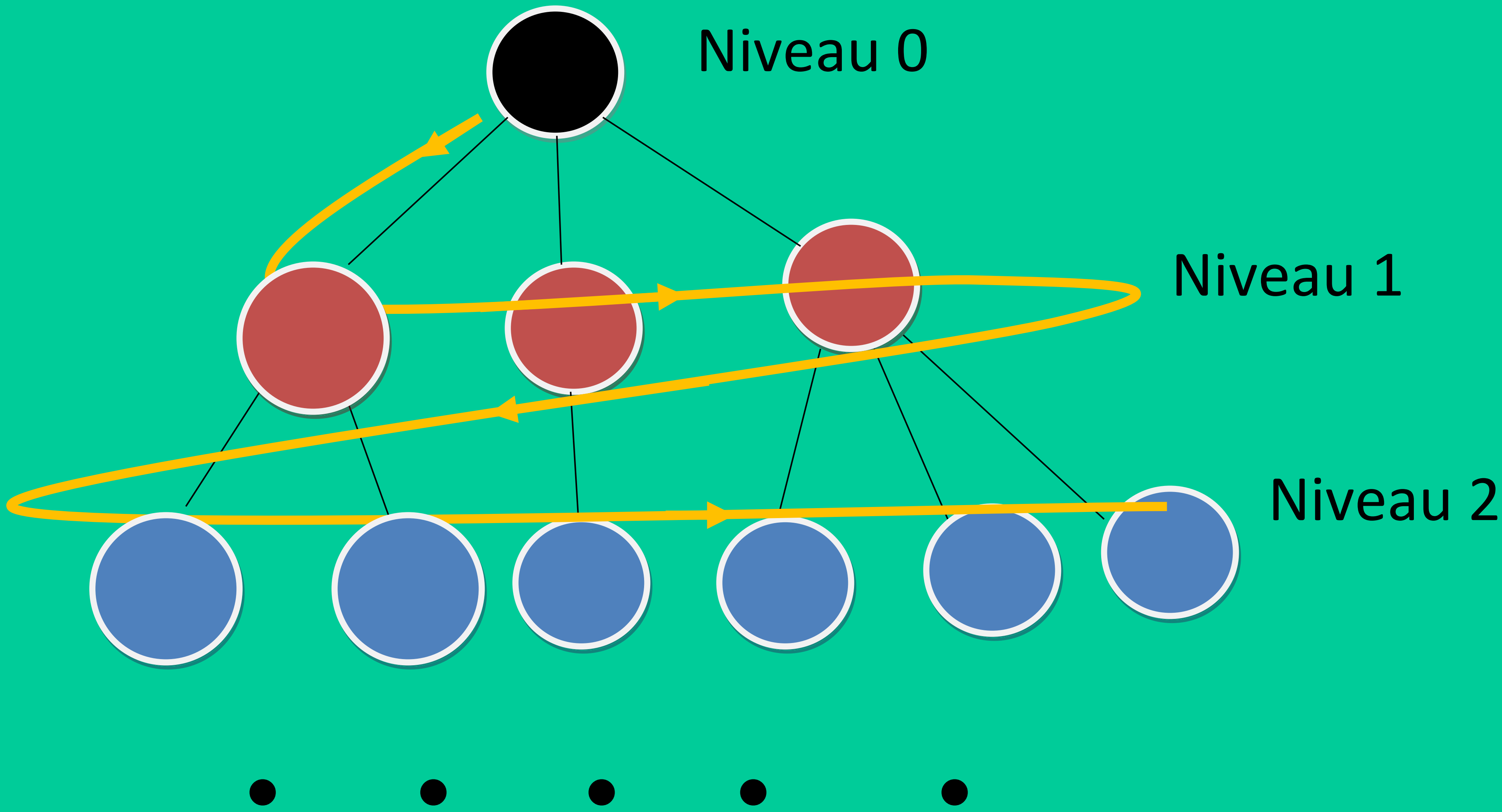
2- Parcours d’arbre en largeur

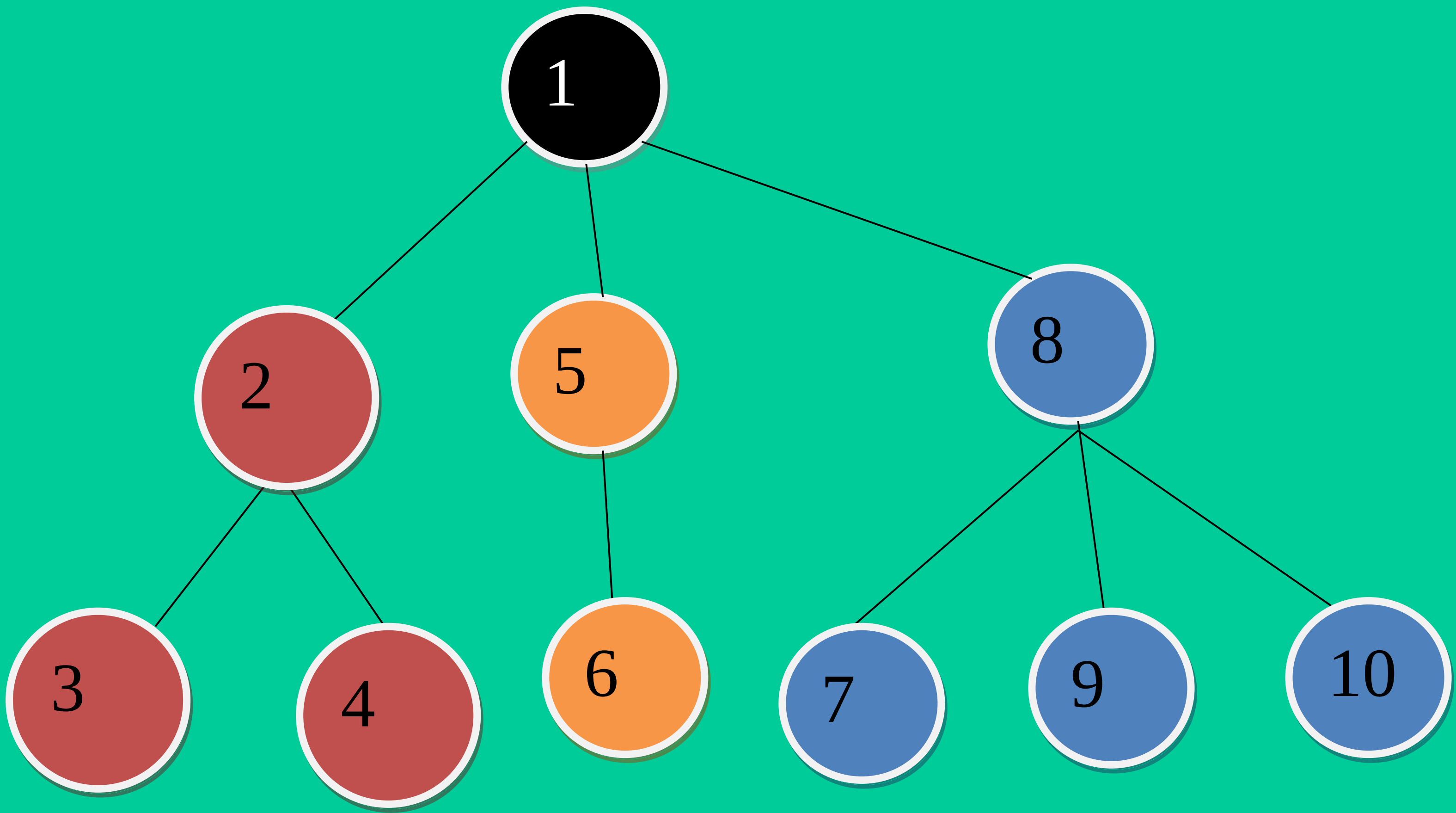
Un parcours en largeur permet de visiter les nœuds «**niveau par niveau**».

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 */
                 }
         }
}
```