# ARBRES

INF2031

# 1-généralités

# 1) Définition

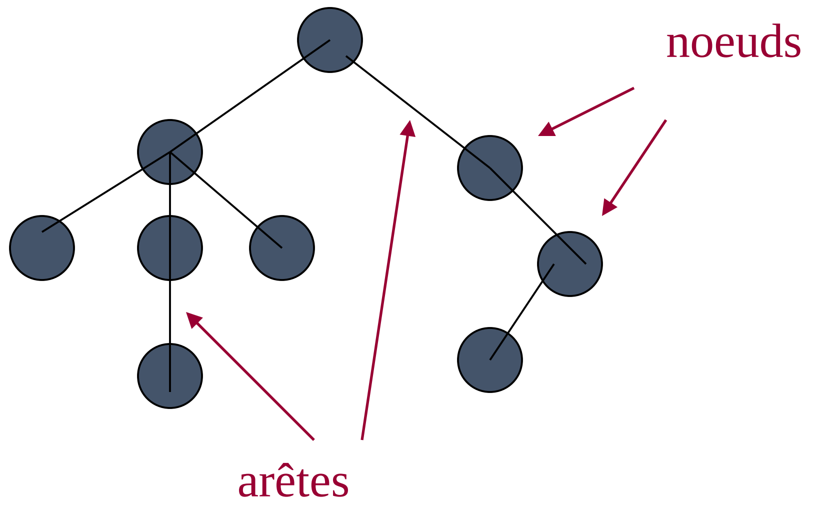
- Un **arbre** est une structure de données organisées de façon hiérarchique, à partir d'un nœud distingué appelé racine.
- Très importante en informatique!.
- Nous étudierons deux types d'arbres : **Arbre Binaires de Recherches** et **Arbres équilibrés**

# 2) Exemples

- système de fichiers UNIX/Windows, Arbres de tri, Arbre de décision, Arbre de jeux (i.e., Echecs )

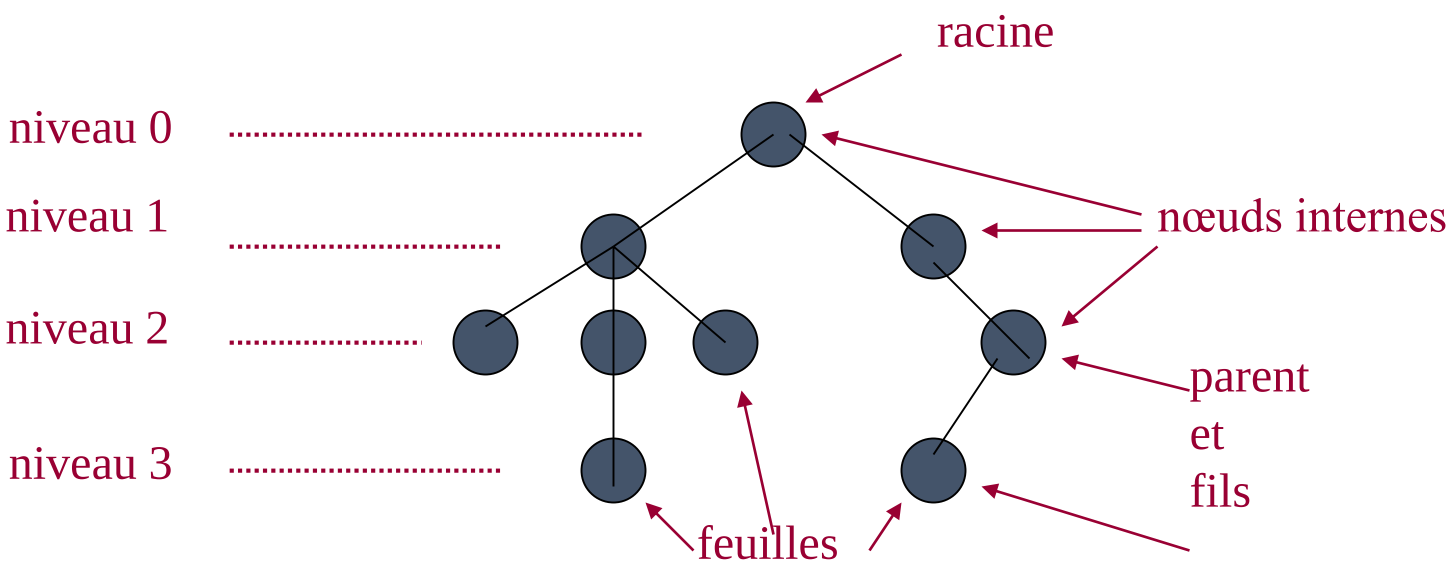
# 3) vocabulaire

- Un arbre est un ensemble de **Nœuds**, reliés par des **Arêtes**. Entre deux nœuds il existe toujours un seul chemin.



## …

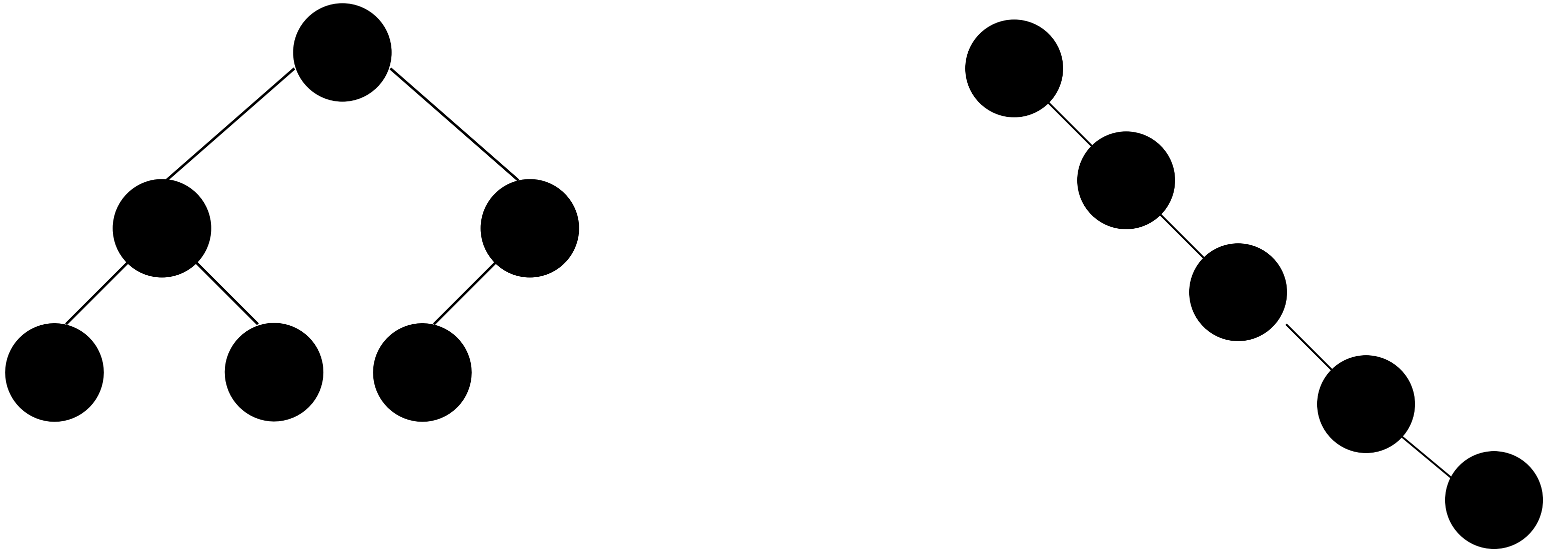
- Les arbres sont enracinés. Une fois la **racine** définit tous les nœuds admettent un **niveau**.
- Les arbres ont des noeuds **internes** et des **feuilles** (nœuds externes). Chaque noeud (à l'exception de la racine) a un **parent** et admet zéro ou plusieurs **fils**.



# 2 – arbres binaires

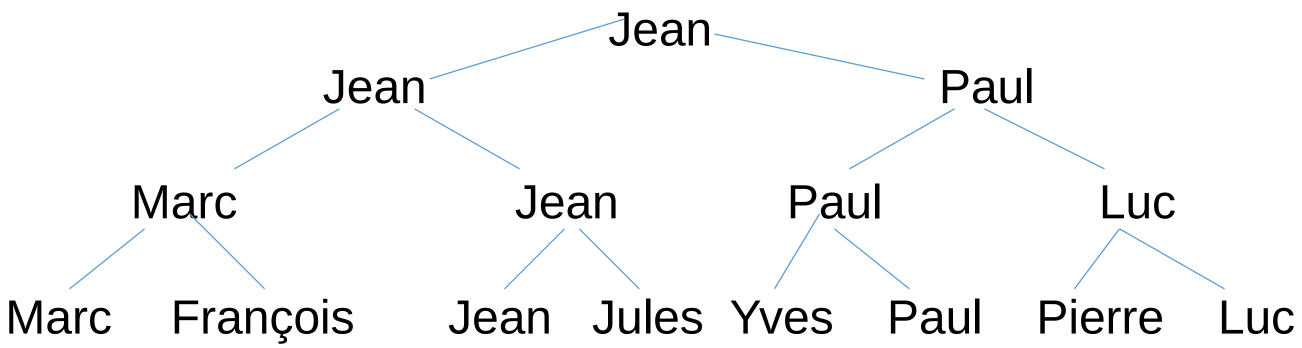
# 1) Définitions

- Un **Arbre Binaire** est un arbre où chaque nœud admet au plus 2 fils.

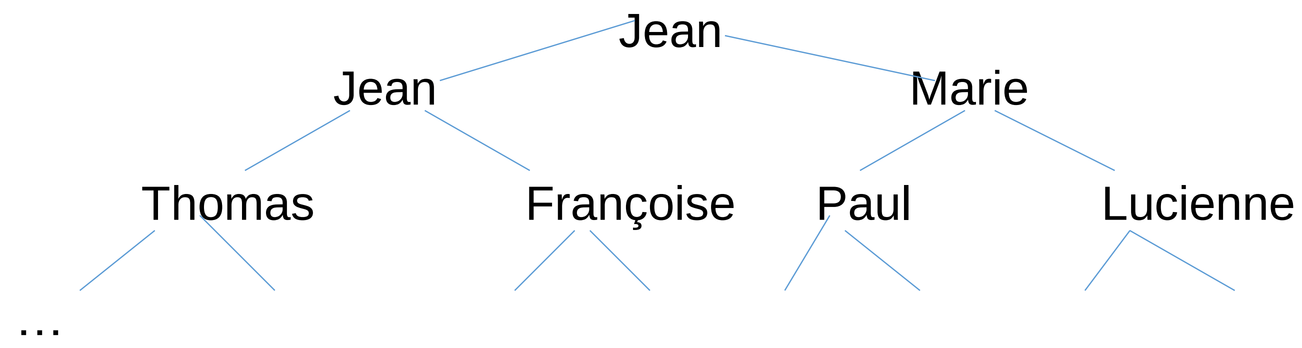


# 2) Exemples

- les résultats d'un tournoi de tennis : au premier tour Jean a battu Jules, Marc a battu François, Paul a battu Yves, et Luc a battu Pierre ; au deuxième tour Jean a battu Marc, et Paul a battu Luc ; et Jean a gagné en finale contre Paul.



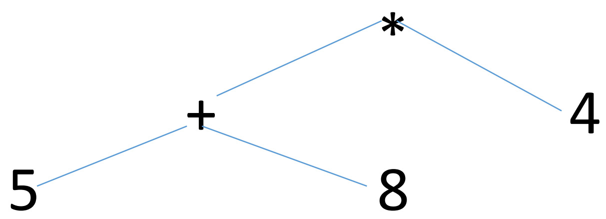
- Arbre généalogique des ascendants




→ Arbre des descendants ?

• une expression arithmétique dans laquelle tous les opérateurs sont binaires

(5 + 8) * 4



# 3) Vocabulaire

étant donné un arbre B = <o, B1, B2 > :

- 'o' est la racine de B.

- B1 est le **sous-arbre gauche** de B, et B2 est son **sous-arbre droit**.

- On dit que C est un **sous-arbre** de B si, et seulement si : C = B, ou C = B1, ou C = B2, ou C est un sous-arbre de B1, ou de B2.

•On appelle **fils gauche** (respectivement **fils droit**) d'un nœud la racine de son sous-arbre gauche (respectivement sous-arbre droit)

•On dit qu'il y a un **lien gauche** (respectivement droit) entre la racine et son fils gauche (respectivement fils droit).

•Si un nœud $n_i$ a pour) un nœud $n_j$, on dit que $n_i$ est le **père** de $n_j$ (chaque nœud n'a qu'un seul père).

•Deux nœuds qui ont le même père sont dits **frères**.

•Le nœud $n_i$ est un **ascendant** ou un **ancêtre** du nœud $n_j$ si, et seulement si, $n_i$ est le père de $n_j$, ou un ascendant du père de $n_j$ ;

• $n_i$ est un **descendant** de $n_j$ si, et seulement si $n_i$ est fils de $n_j$, ou $n_i$ est un descendant d'un fils de $n_j$.

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

Donc, un arbre binaire est :

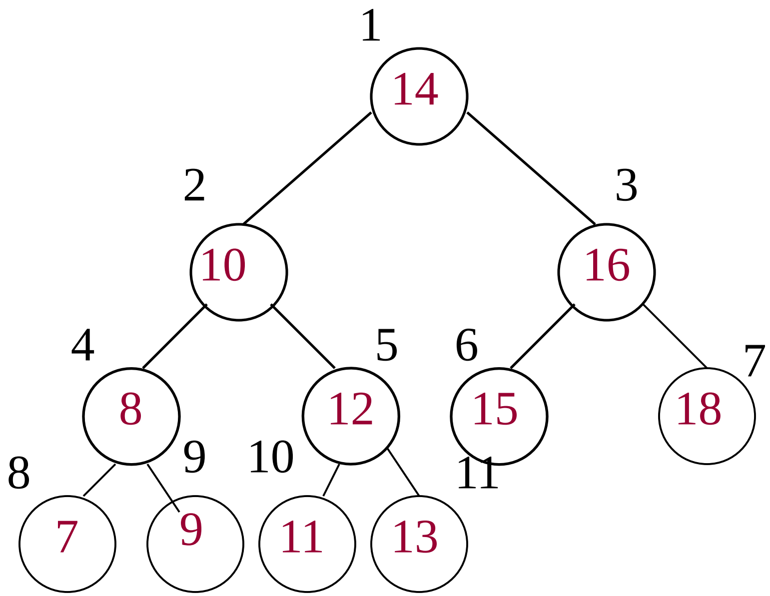
•soit vide

•soit constitué d'une information appelé clé (*key*) et d'au plus 2 arbres binaires

# 4) représentation par tableaux

- Un arbre binaire complet peut être représenté par un tableau A avec un accès en O(1) à chaque noeud:
  - Mémoriser les noeuds séquentiellement de la racine aux feuilles et de gauche vers la droite.
  - Fils gauche de A[i] est en A[2i]
  - Fils droit de A[i] est en A[2i + 1]
  - Parent de A[i] est en A[i/2]

# Arbres Binaires: représentation par tableau




| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| tab A: | 14 | 10 | 16 | 8 | 12 | 15 | 18 | 7 | 9 | 11 | 13 |

# 5) Modélisation

```
Type structure node

        entier key;
        Structure modifiable ls, rs   //left son, right son
Fin
```

Si a est un pointeur sur la racine de l'arbre
a=NULL correspond à un arbre vide
a->ls pointe sur le fils gauche de a
a->rs pointe sur le fils droit de a
a->key permet d'accéder au contenu du nœud.

# 3 – arbre binaires de recherches

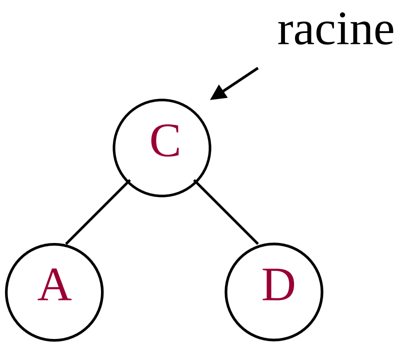
# 1) Définition

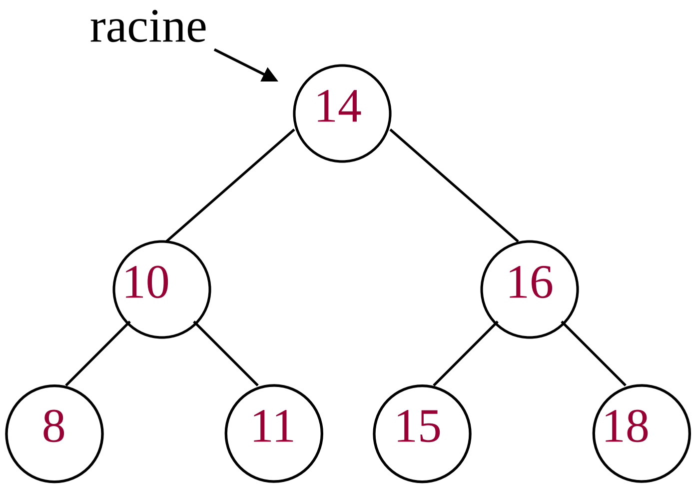
- Un **Arbre Binaire de Recherche (ABR)** est un arbre binaire avec les propriétés suivantes :
  - La clé associée à un noeud est *supérieur* aux clés des nœuds de son sous-arbre gauche
  - La clé associée à un noeud est *inférieur* aux clés des nœuds de son sous-arbre droit

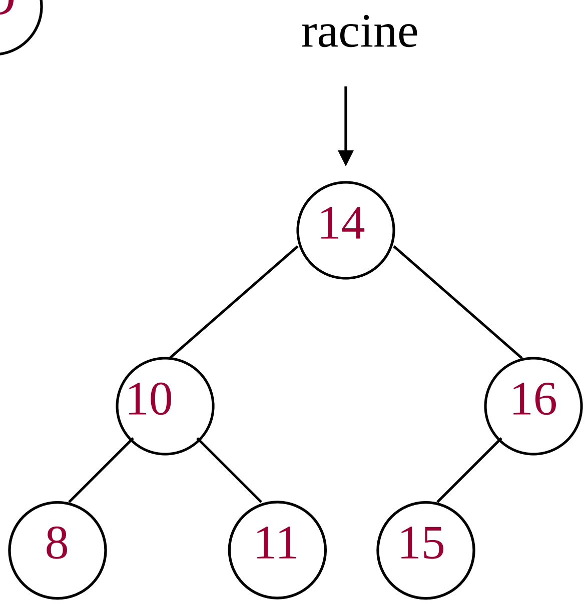
- Un ABR possède donc la **propriété** suivante :

**a->ls->key < a->key < a->rs->key**

# 2) Exemples



# 3) Recherche d'un élément dans un ABR

- Pas de notion de retour arrière, on ne parcourt qu'une branche de l'arbre car on sait si l'élément recherché est plus grand ou plus petit que la valeur contenue dans la racine.
- Donc, le parcours est naturellement dichotomique.

# a) Principe

```
SI l'arbre est vide ALORS fin et échec
SINON
   SI l'élément cherché = élément pointé ALORS fin et réussite
   SINON
         SI l'élément cherché < élément pointé ALORS
                  rechercher l'élément dans le sous arbre gauche
         SINON
                  rechercher l'élément dans le sous arbre droit
```

# b) Algorithme

```
entier rechercher(entier x, arbre a)
   entier ok
   SI (a = NULL) ALORS
      ok ← 0
   SINON
      SI (a->key = x) ALORS
           ok ← 1
      SINON
           SI (a->info > x) ALORS
                ok ← rechercher(x, a->ls)
           SINON
                ok ← rechercher(x, a->rs)
   retourner ok
fin
```

**!!!** Cet algorithme renvoie la première occurrence du terme cherché, c.à.d. qu'il renvoie vrai la première fois où il rencontre l'élément cherché, et il s’arrête.

Si on recherche la $n^{\text{ième}}$ apparition de l'élément dans l'arbre, il faut mettre un compteur.

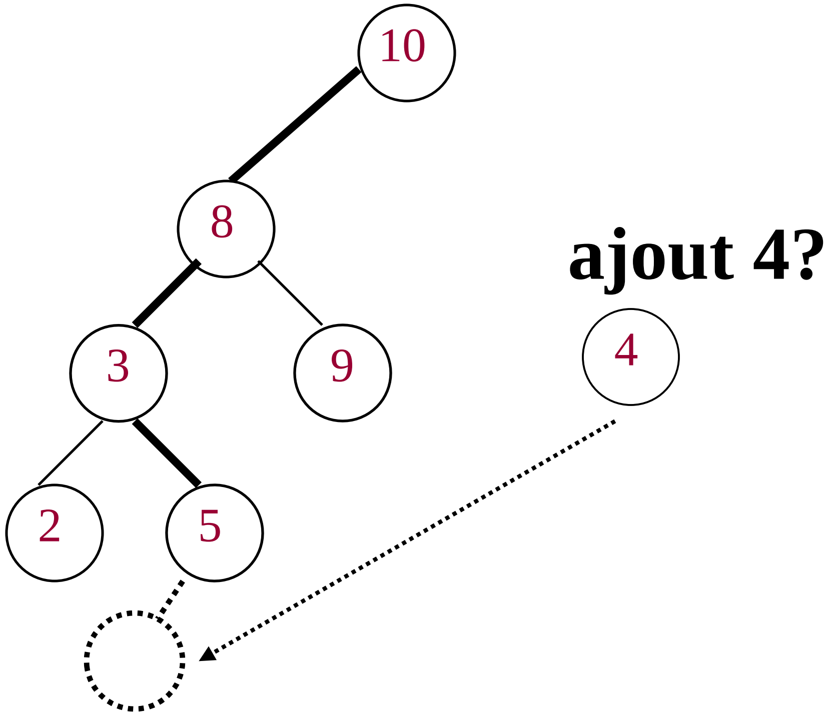
# 4) Ajout d'un élément dans un ABR

**Comment ajouter une clé?**

Déterminer la position d'insertion en utilisant la fonction **rechercher**.

Ajouter la nouvelle clé si la recherche échoue.

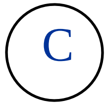
Exemple:



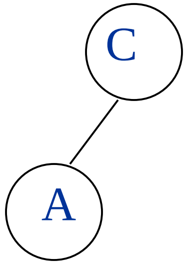
# Construction d’un ABR

**Exemple:** ajouter **C A B L M** **(dans l’ordre!)**

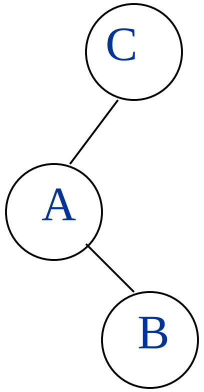
**1) Ajouter C**



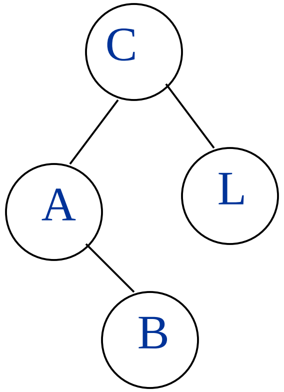

**2) ajouter A**



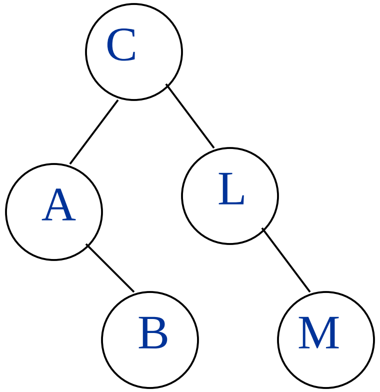

**3) ajouter B**




**4) Ajouter L**




**5) Ajouter M**

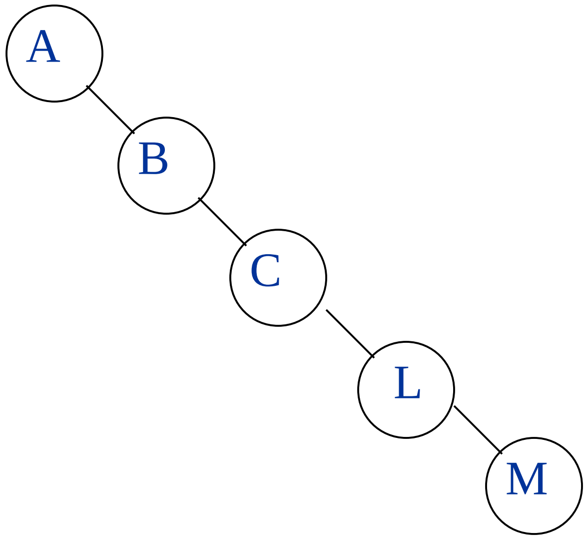


# Construction d’un ABR

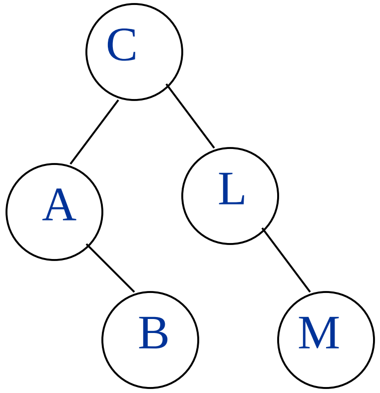
L’ABR est-il unique pour une séquence de lettres A B C L M ?

**NON!** différentes séquences donnent différents ABR

**Ajout de : A B C L M**




**Ajout de : C A B L M**



# Principe

Un ajout d'élément dans un ABOH se fait systématiquement aux feuilles :

1) SI arbre est vide ALORS création et ajout

SINON trouver la feuille

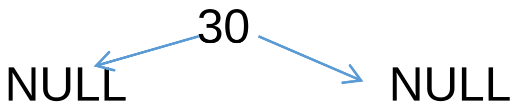
Trouver la feuille : parcourir l'arbre et rechercher la position de l'élément:

c.à.d. comparer l'élément à ajouter à la valeur contenue dans la racine :

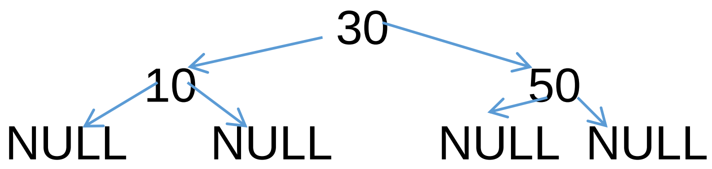
SI la valeur contenue dans la racine > élément à ajouter ALORS ajout dans le sag, donc retour en 1) avec le sag.

SINON ajout dans le sad, donc retour en 1) avec le sad

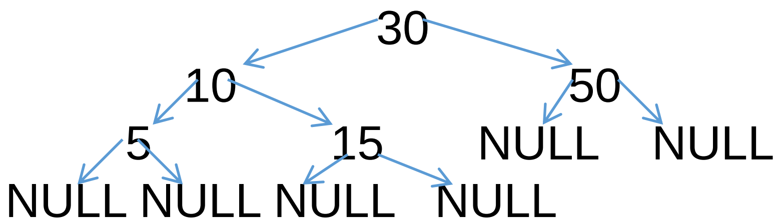
Exemple :



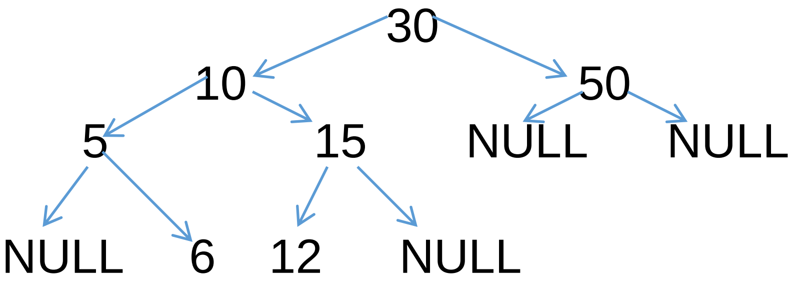

On veut ajouter 10 et 50




On veut ajouter 5 et 15




On veut ajouter 6 et 12



# Algorithme (en récursif)

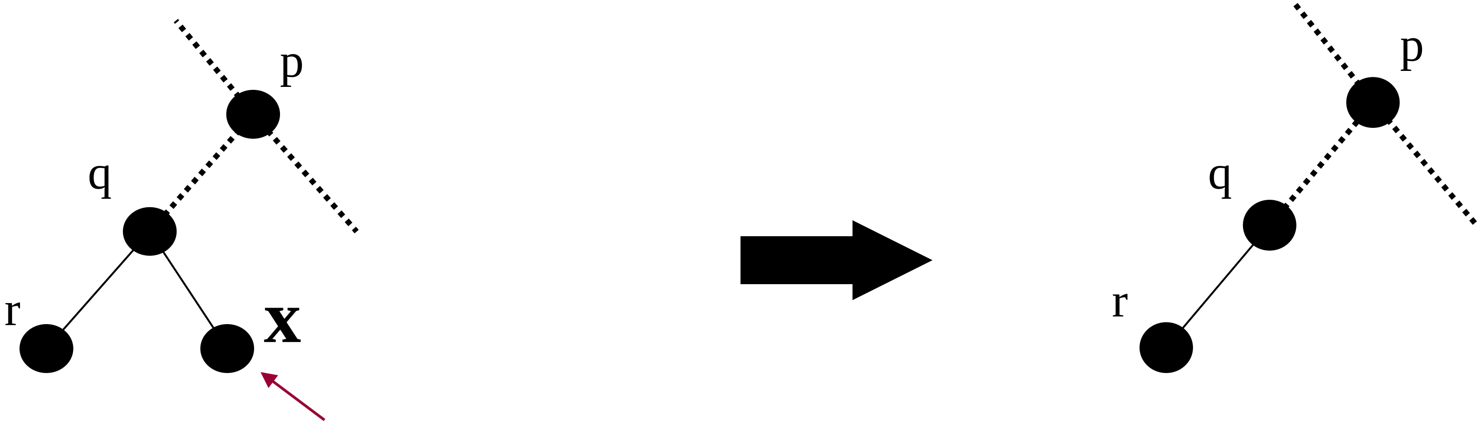
```
Arbre modifiable ajouter(entier x, arbre modifiable a)
    SI (a = NULL) ALORS
            allouer(a)
            a->key← x
            a->ls ← NULL
            a->rs ← NULL
    SINON
            SI (a->key < x) ALORS
                        a->ls <- ajout(x, a->ls)
            SINON
                        a->rs <- ajout(x, a->rs)
            FINSI
    FINSI
    retourner a
FIN
```

# ABR : supprimer un élément

Pour supprimer un nœud contenant x,

rechercher x, une fois trouvé appliquer l'un des trois cas suivants:

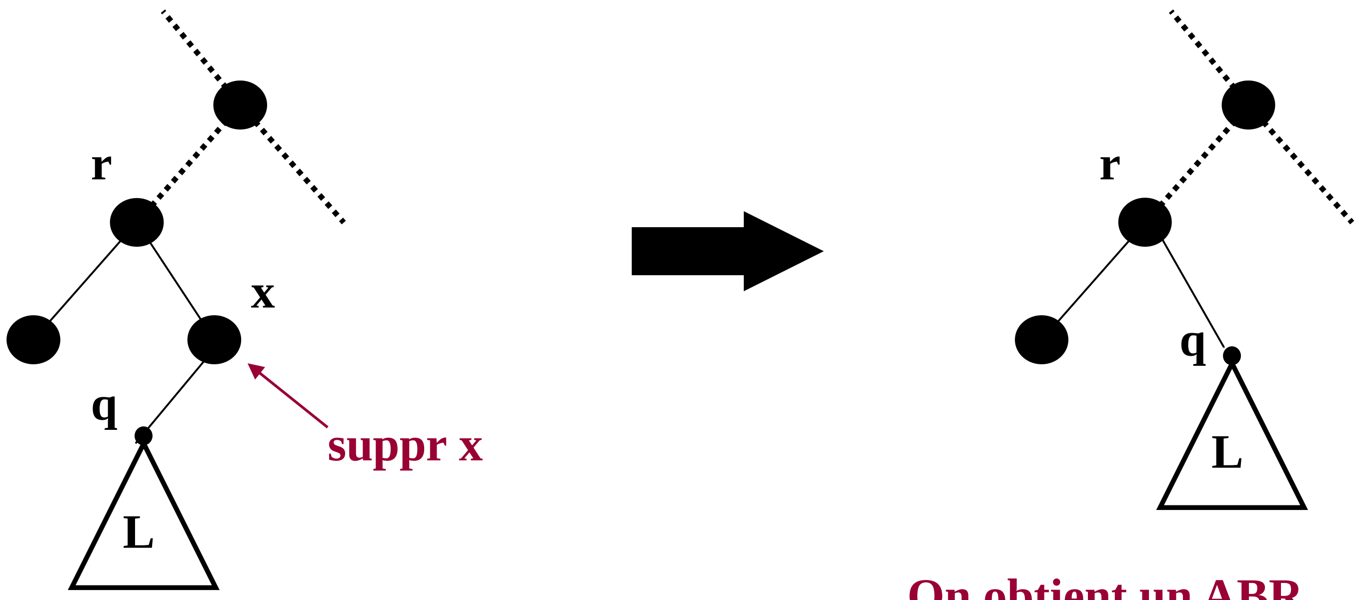
## CAS A: x est une feuille




**supprimer x** **On obtient un ABR**

# ABR : supprimer un élément

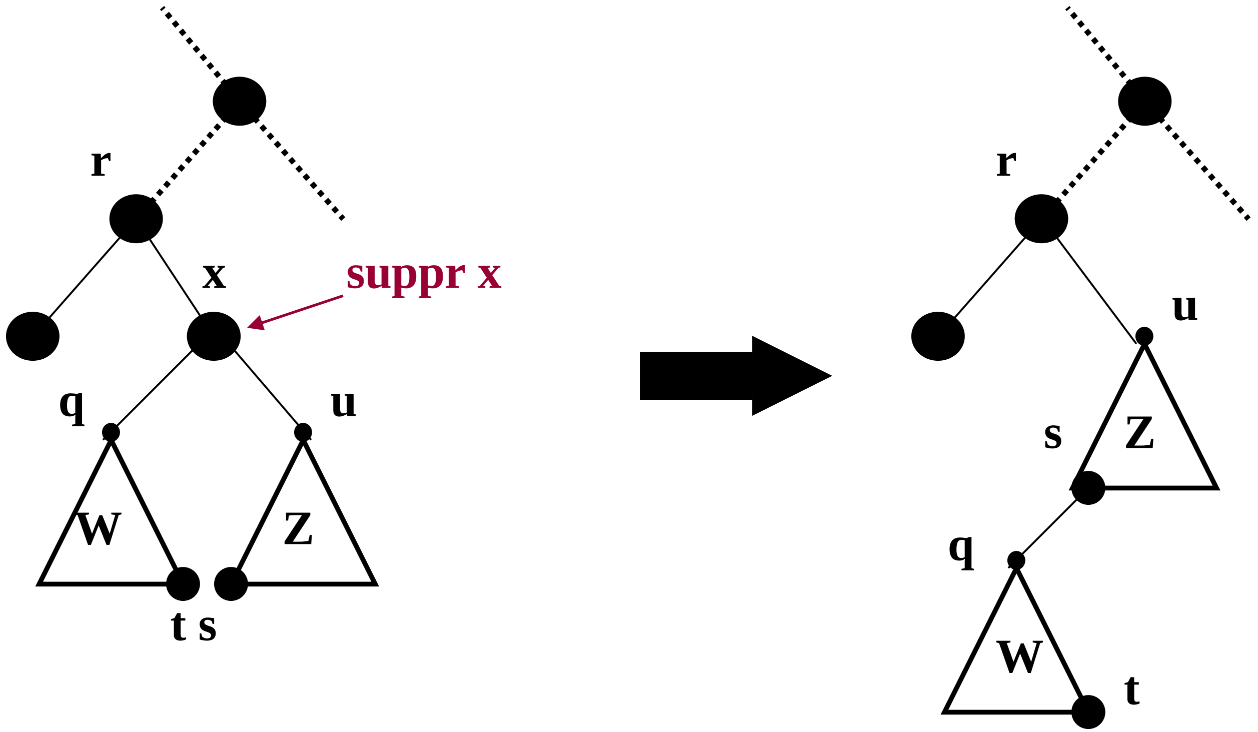
## <u>Cas B: x est un nœud interne avec un seul sous-arbre</u>




On obtient un ABR

# ABR : supprimer un élément

## Cas C: x est un nœud interne avec 2  sous-arbres




propriété ABR est conservé

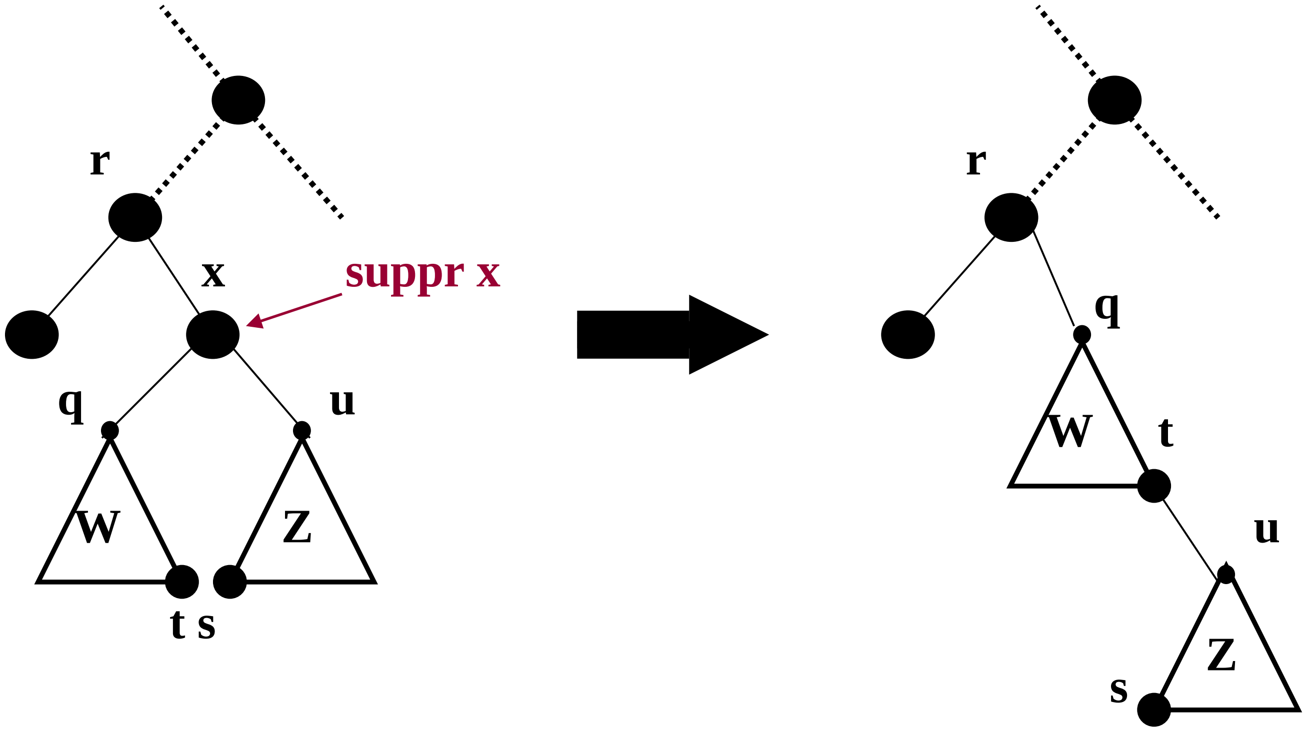
$$q < x < u$$

$\Rightarrow$ q est inférieur au plus petit élément de Z

$\Rightarrow$ r est supérieur au plus grand élément de W

# ABR : supprimer un élément

## Cas C suite: … ou encore comme suit




$$q < x < u$$

⇒ q est inférieur au plus petit élément de Z

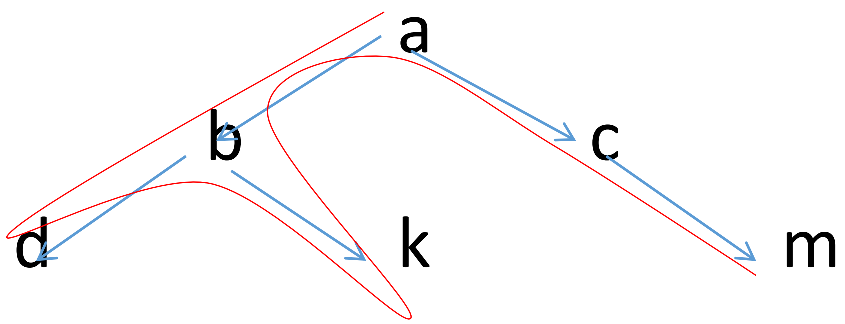
⇒ r est supérieur au plus grand élément de W

propriété ABR est conservé

# 4 – parcours

# 1) Parcours en profondeur

examiner complètement un chemin et passer au chemin suivant tant qu'il en reste : **DFS** (Depth First Search)

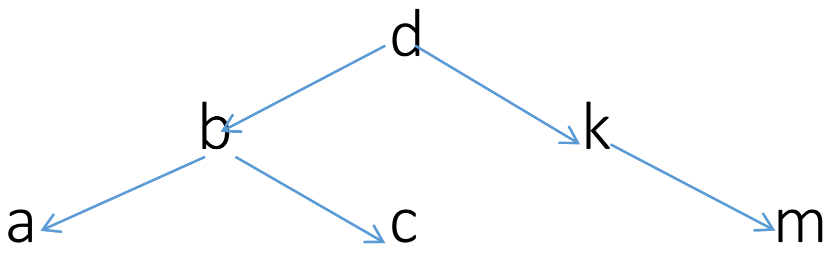


# a) Parcours : PréOrdre (préfixe)

Préordre : d b a c k m

PréOrdre est décrit récursivement :

- Visiter la racine
- si elle est non vide
  - Visiter le sous-arbre gauche en PréOrdre
  - Visiter le sous-arbre droit en PréOrdre



# Algorithme

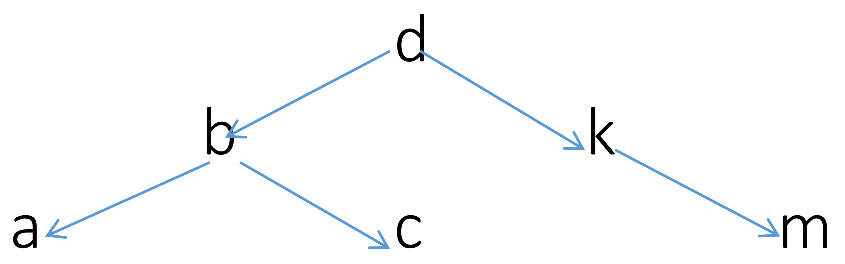
```
Vide pre_ordre(node modifiable a)

{
  SI (a ≠ NULL) alors
  {
        ecrire a->key
        pre_ordre(a->ls)
        pre_ordre(a->rs)
  }
}
```

# b) Parcours : InOrdre (infixe)

Inordre : a b c d k m

InOrdre est décrit récursivement :

- si la racine est non vide
  - Visiter le sous-arbre gauche en InOrdre
  - Visiter la racine
  - Visiter le sous-arbre droit en InOrdre



# Algorithme

```
Vide in_ordre(node modifiable a)

{
  SI (a ≠ NULL) ALORS
  {
        in_ordre(a->ls)
        ecrire a->key
        in_ordre(a->rs)
  }
}
```

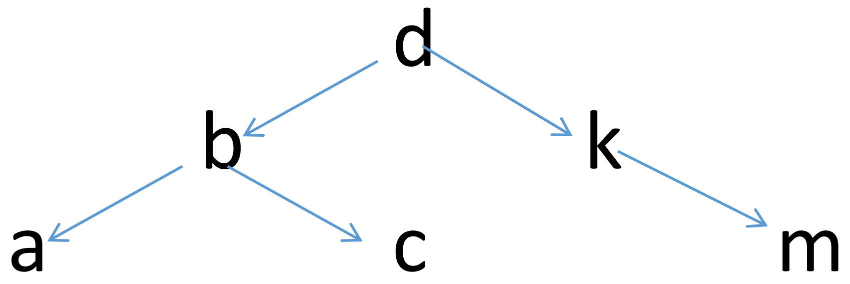
Remarque : *si l'arbre est un ABR, cet algorithme affiche les clés triées dans l'ordre croissant*

# c) Parcours : PostOrdre (postfixe)

c) Postordre : a c b m k d

SI l'arbre n'est pas vide :

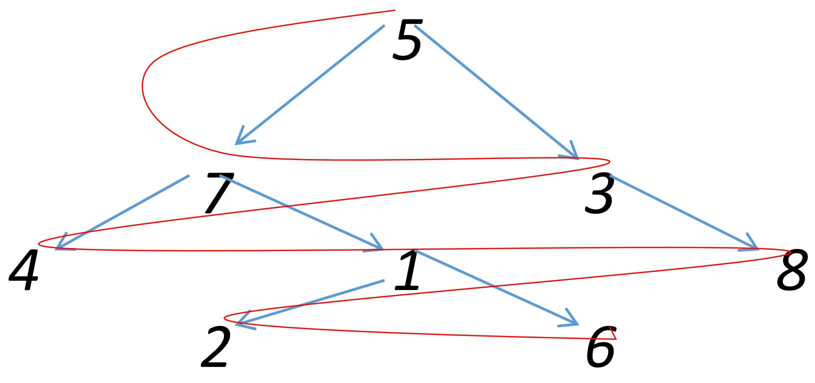
- visiter en postordre le sous arbre gauche
- visiter en postordre le sous arbre droit
- visiter la racine



# 2) Parcours en largeur

examiner tout un niveau (profondeur hiérarchique) passant au niveau du dessous tant qu'il en reste : **BFS** (Breadth first search)

*Problème : pas de lien entre fils. Cela doit être traité itérativement (couche par couche).*

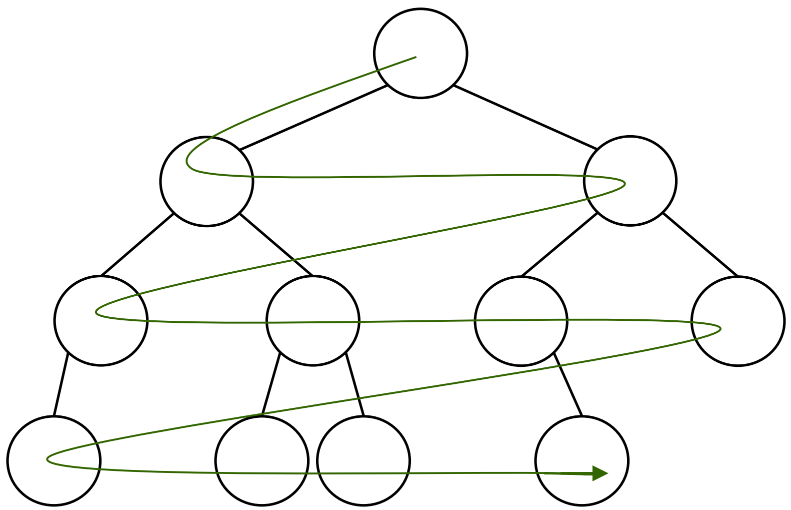


# Parcours: levelOrdre

LevelOrdre visite les noeuds niveau par niveau depuis la racine:
Peut être décrit facilement en utilisant une File **:**

- au départ, on commence avec une file vide, dans laquelle on ajoute la racine

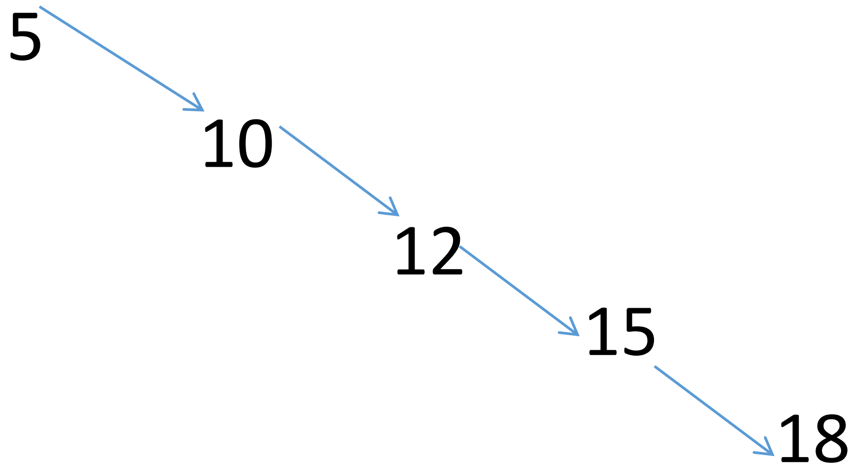
- tant que la file n'est pas vide, on enlève le premier noeud de la file, on le parcourt et on ajoute tous ses enfants dans la file.

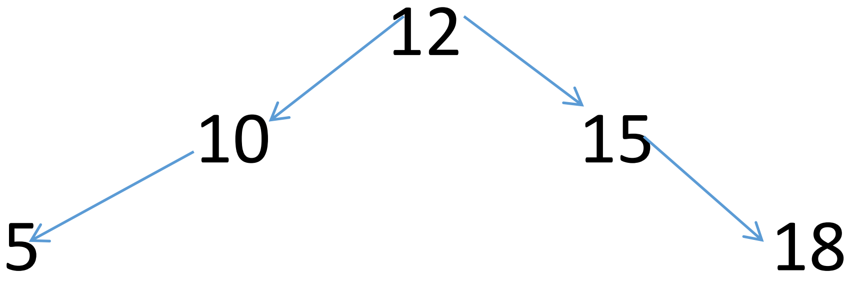


# 5 – Arbres équilibrés

# 1) Equilibre des arbres binaires

Déséquilibre possible d'un ABOH.



# Conséquences

Si un arbre est déséquilibré, ses performances sont mauvaises (instables).

Les performances dépendent de l’ordre d’insertion des informations.

La recherche n'est plus dichotomique dans un arbre déséquilibré.

# 2) Définition d'un arbre équilibré

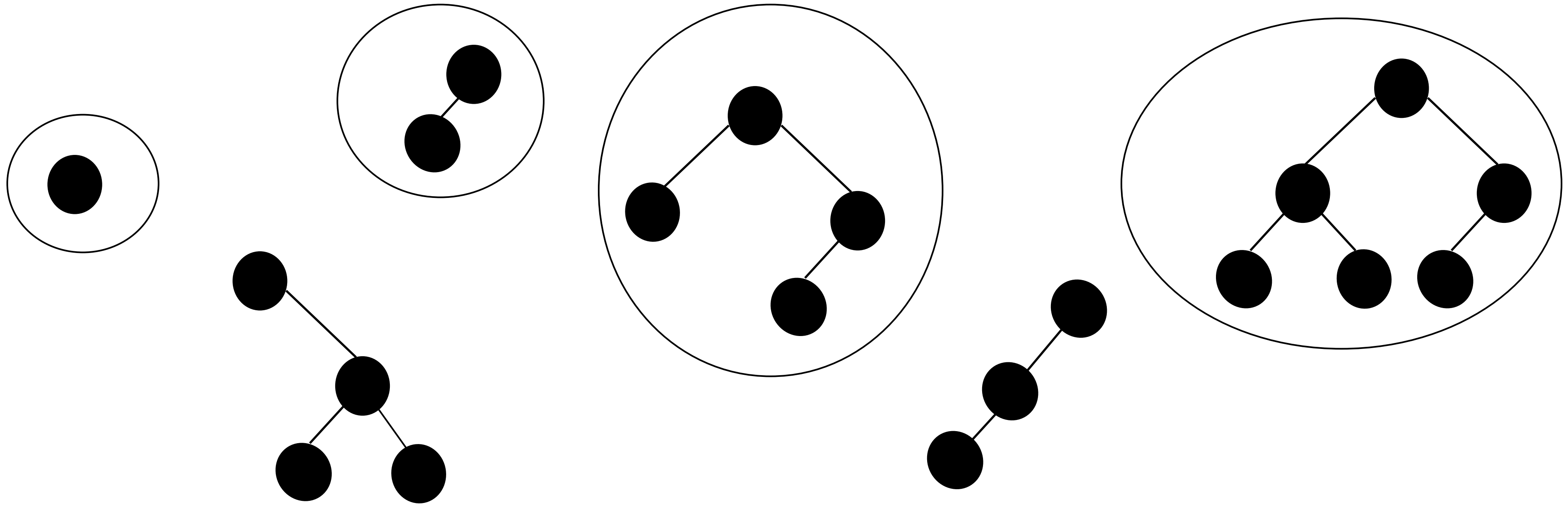
**Arbre AVL** (Adelson-Velskii et Landis):

Pour tout nœud de l'arbre, la valeur absolue de la différence entre le nombre des nœuds du sad et le nombre des nœuds du sag est inférieure ou égale à 1.

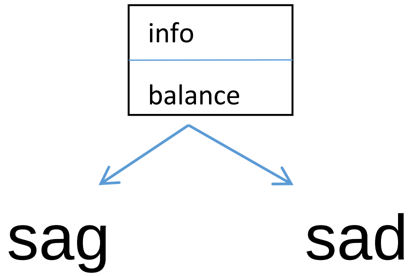
$|n_g - n_d| \leq 1$

*Idée :* si l'insertion ou la suppression provoque un déséquilibre de l'arbre, rétablir l'équilibre.

# Exemples



# 3) Algorithmique




balance :

-1 : hsag = hsad + 1
0 : hsad = hsag
+1 : hasd = hsag + 1

## a) Compter le nombre de noeuds dans un arbre binaire

<u>Principe</u>

SI l'arbre est vide retourner 0.

SINON

compter le nombre de nœuds du sag

compter le nombre de nœuds du sad

retourner $1 + n_{sag} + n_{sad}$

## b) Calculer la hauteur d’un arbre binaire

<u>Principe</u>

SI l'arbre est vide retourner 0.
SINON
    calculer la hauteur du sag
    calculer la hauteur du sad
    SI hg > hd retourner 1 + hg
    SINON retourner 1 + hd

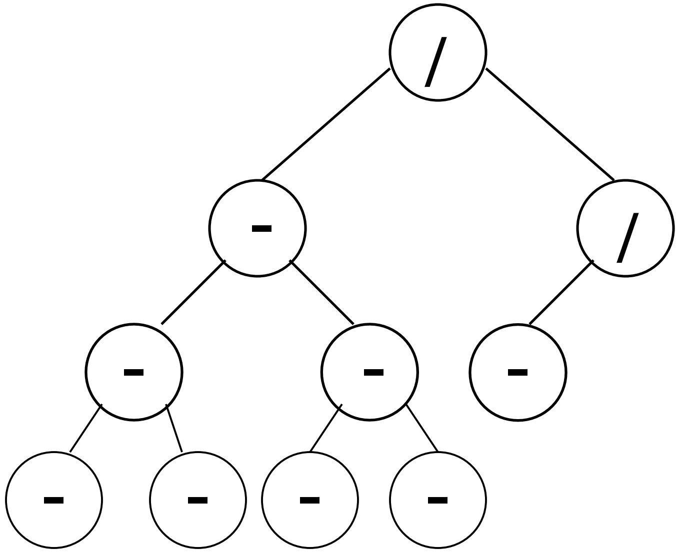
# 4) Arbres AVL

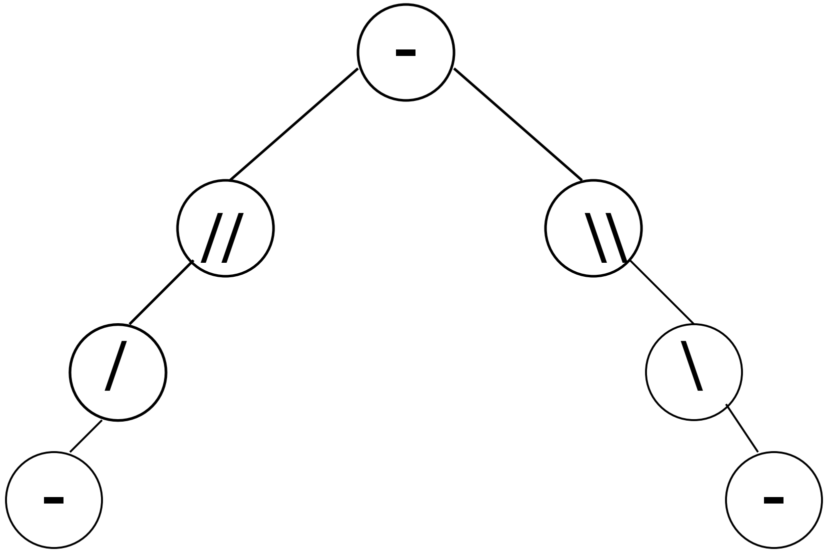
Pour plus lisibilité , remplaçer les clés associées aux nœuds en utilisant /, \, -, // **et** \\ pour représenter le facteur d'équilibre d'un nœud :

| | | |
|---|---|---|
| / : léger déséquilibre à gauche | → | h(G) = 1 + h(D) |
| \ : léger déséquilibre à droite | → | h(D) = 1 + h(G) |
| - : équilibré | → | h(D) = h(G) |
| \\ : déséquilibre droit | → | h(D) > 1 + h(G) |
| // : déséquilibre gauche | → | h(G) > 1 + h(D) |

# Arbres AVL

Exemples :

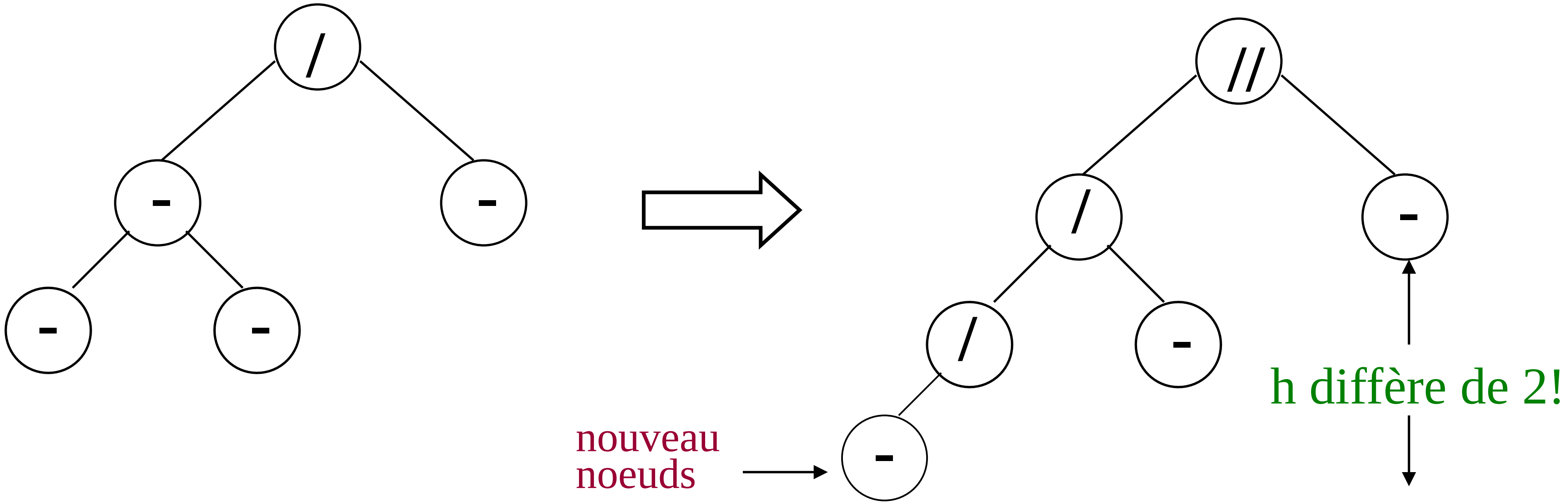



Les clés ne sont pas montré.
On suppose qu'elles satisfassent la propriété ABR

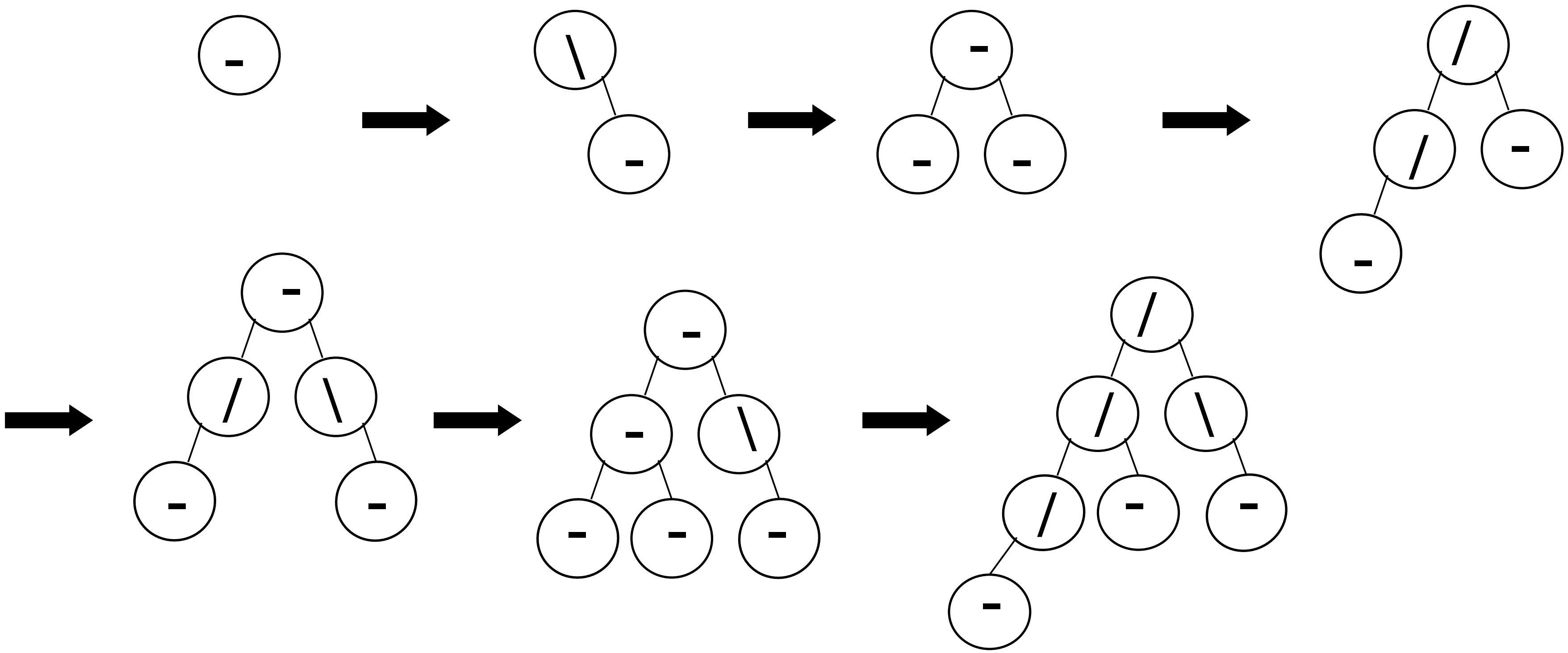
# Arbres AVL

Après chaque opération, *on a besoin de vérifier la propriété d'AVL!. Car l'arbre peut ne plus l'être!*

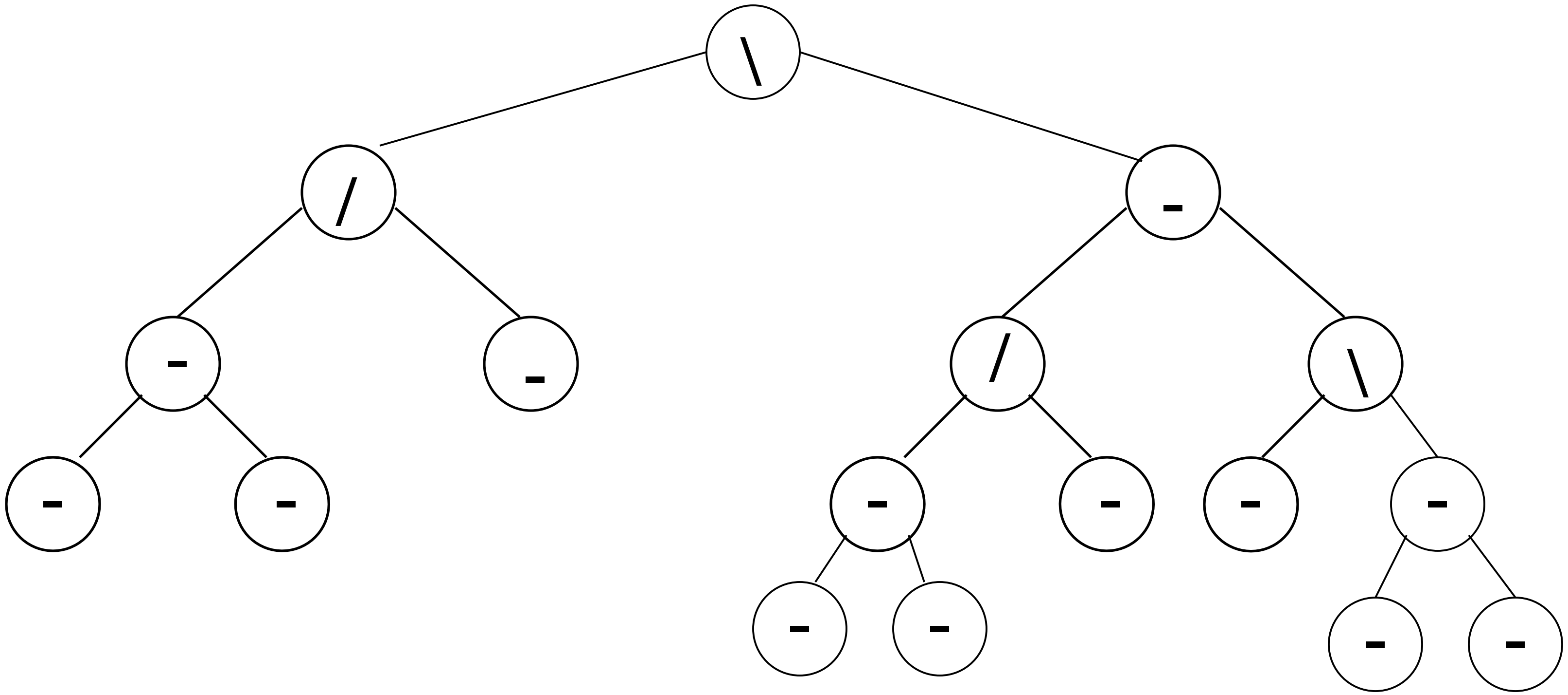


# Arbres AVL

Après une insertion, si l'arbre est un AVL alors on ne fait rien.
Comme sur l'exemple ci-dessous :



# Arbres AVL



Quand une insertion provoque le déséquilibre de l'arbre?

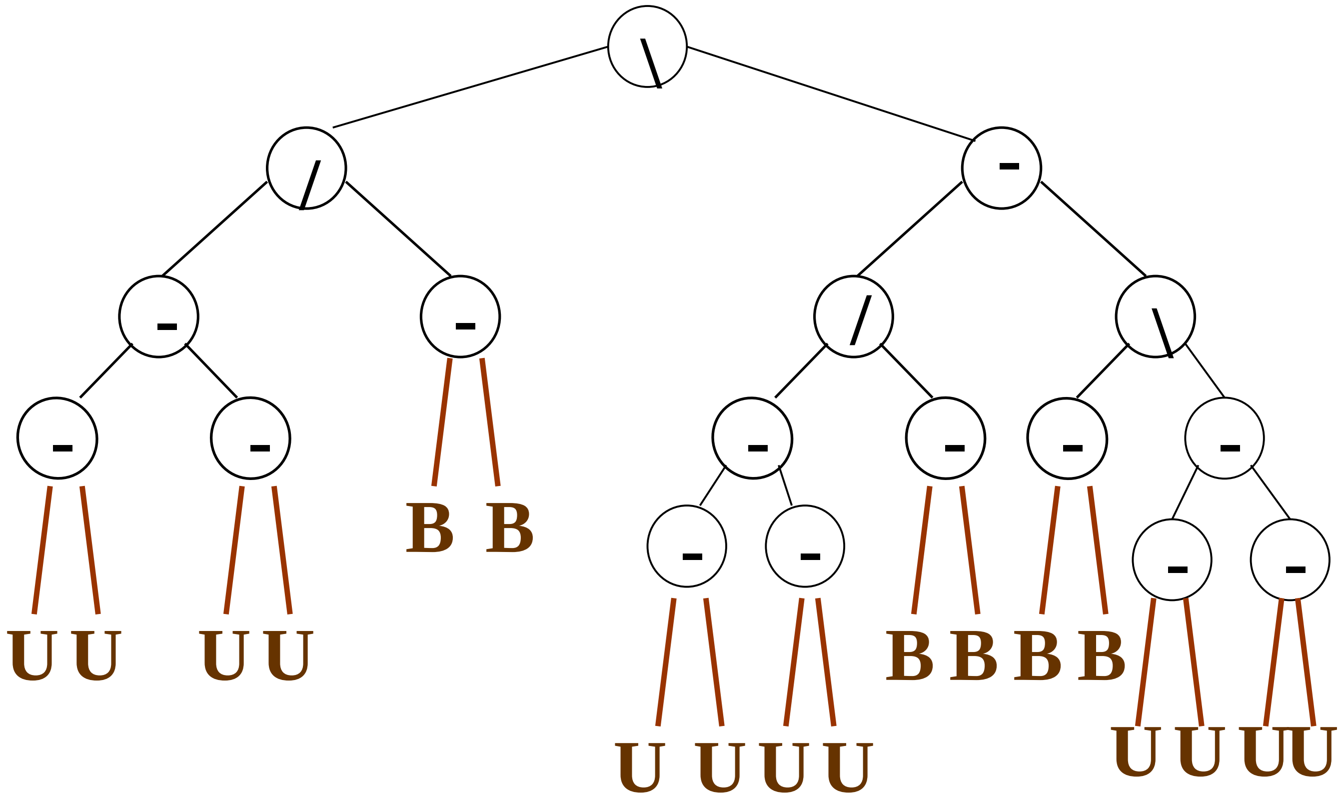
# Arbres AVL : insertion d'un noeud

L'arbre devient déséquilibré si l'élément ajouté est le descendant gauche (droit) d'un nœud avec un léger déséquilibre gauche (droit). Alors la hauteur de ce sous-arbre augmente.

Dans les figure suivantes, on note :

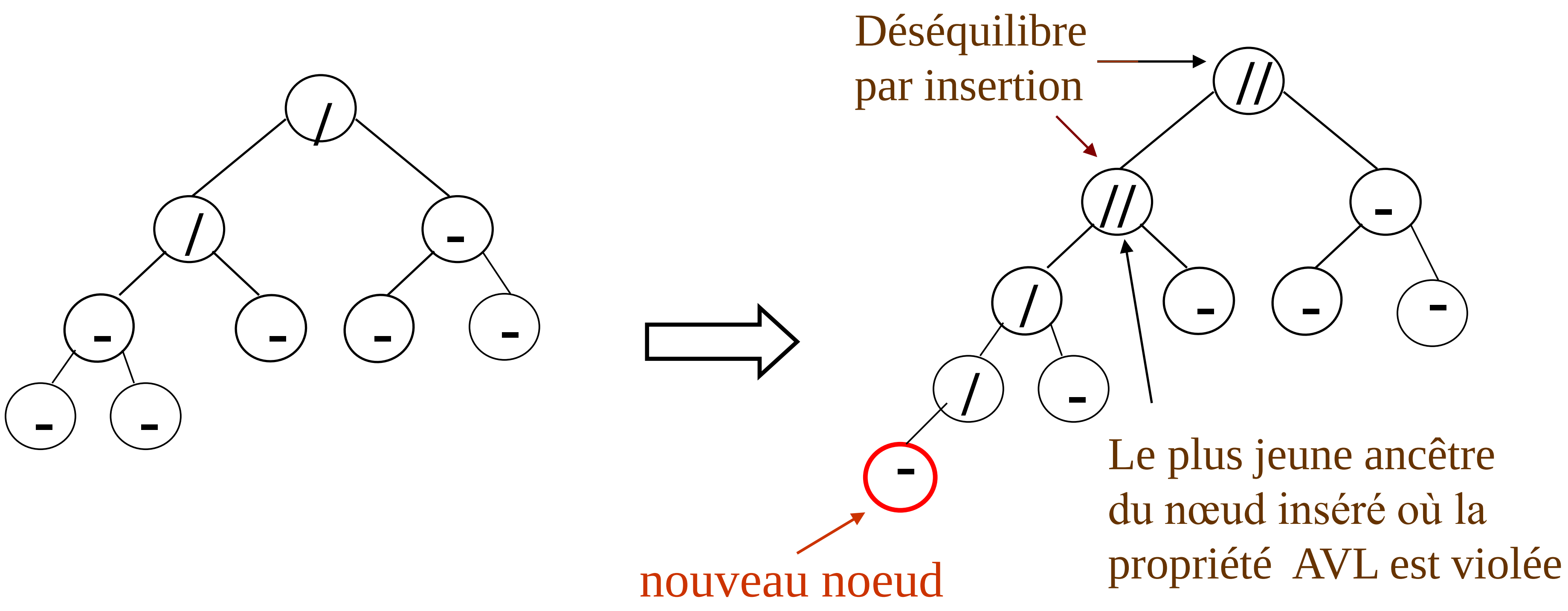
U: *nouveaux nœuds pouvant déséquilibrer l'arbre*
B: *nouveaux laissant l'arbre équilibré*

# Arbres AVL: Insertion



# Arbres AVL: Insertion

Noter que l'insertion d'un nœud peut provoquer des déséquilibre sur plusieurs nœuds.

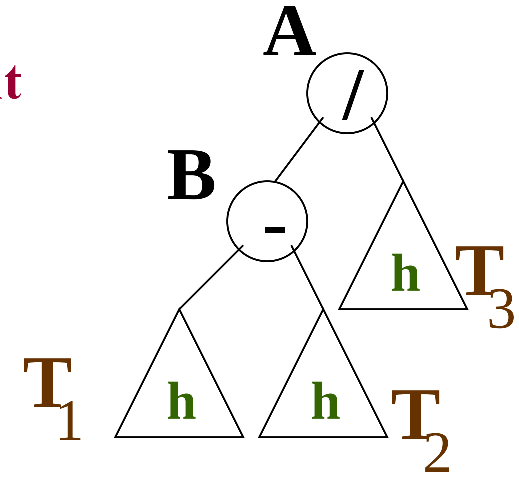


# Arbres AVL: Insertion

Supposons que le sous-arbre le plus  haut est celui de **gauche** et qu’un nœud est inséré pour augmenter la hauteur de ce sous-arbre.
L’arbre obtenu est déséquilibré
Rétablir un arbre AVL en utilisant des  **rotations**

=>  Soit **A** le plus jeune ancêtre  où apparaît le déséquilibre

**Dans l’arbre AVL, avant l’insertion, $T_1$, $T_2$ et $T_3$ ont une hauteur h**



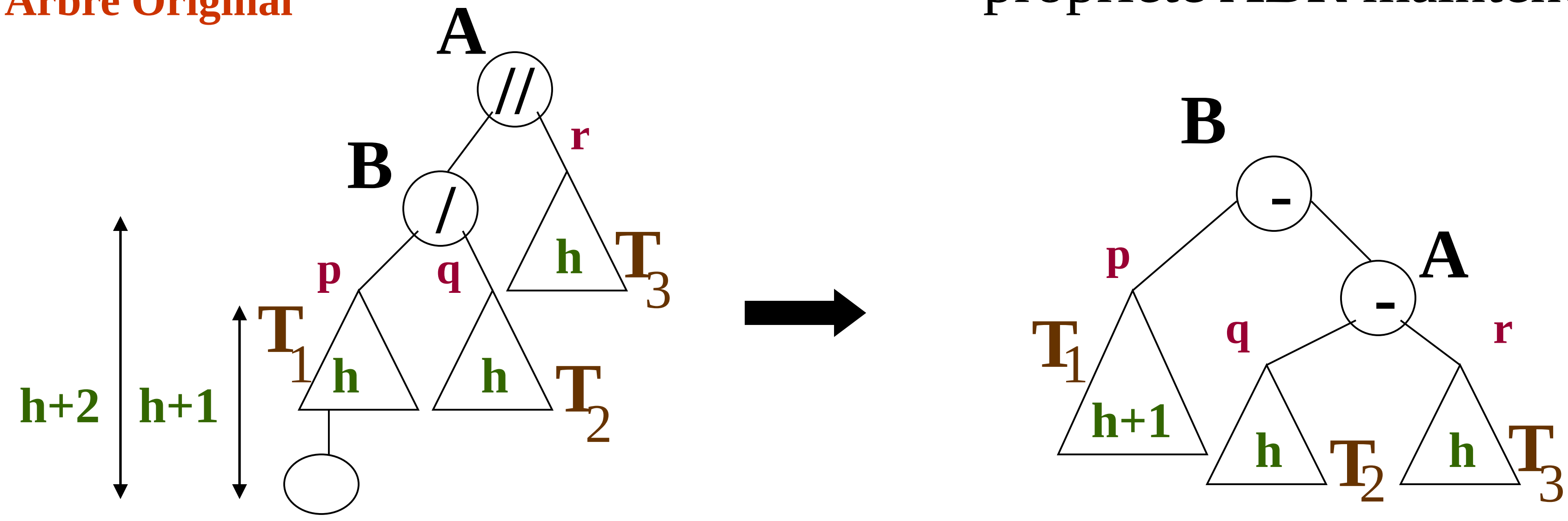

Le même raisonnement peut être utilisé si l ’arbre le plus haut est celui de **droite**

# Arbres AVL: Insertion

**<u>Cas I:</u> un nouveau nœud est inséré dans $T_1$**

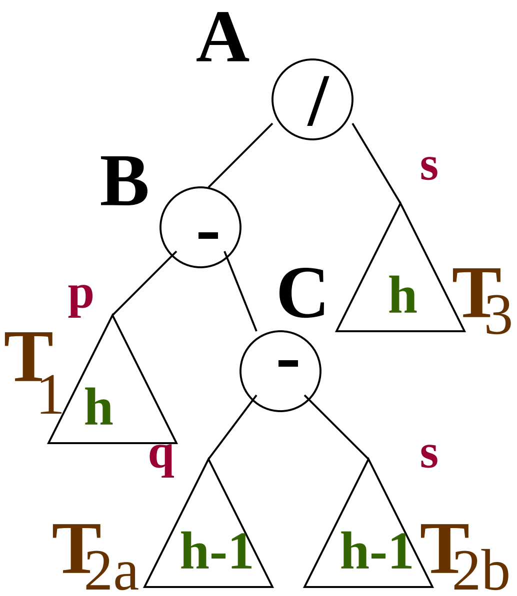
Rééquilibrer par **rotation droite:**
P < B < q < A < r =>
=> propriété ABR maintenue!

**Arbre Original**



# Arbres AVL: Insertion

## <u>Cas II:</u> nouveau noeud inséré dans $T_2$

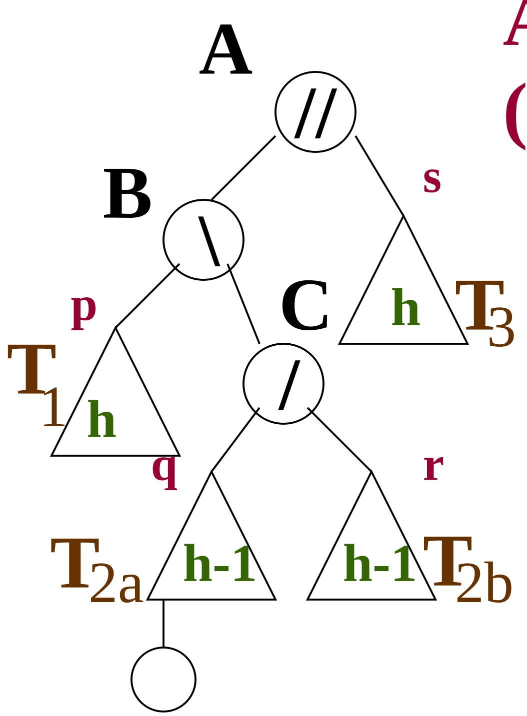



On a 3 cas a considérer :

1- nouveau nœud en C
2- nouveau nœud ajouté à $T_{2a}$
3- nouveau nœud ajouté à $T_{2b}$

Les 3 cas sont similaires.
On considérera le cas 2.

# Arbres AVL: Insertion




**Cas II - T2a :**

**Rééquilibrage de l'arbre AVL avec une double rotation (gauche sur B et ensuite droite sur A)**

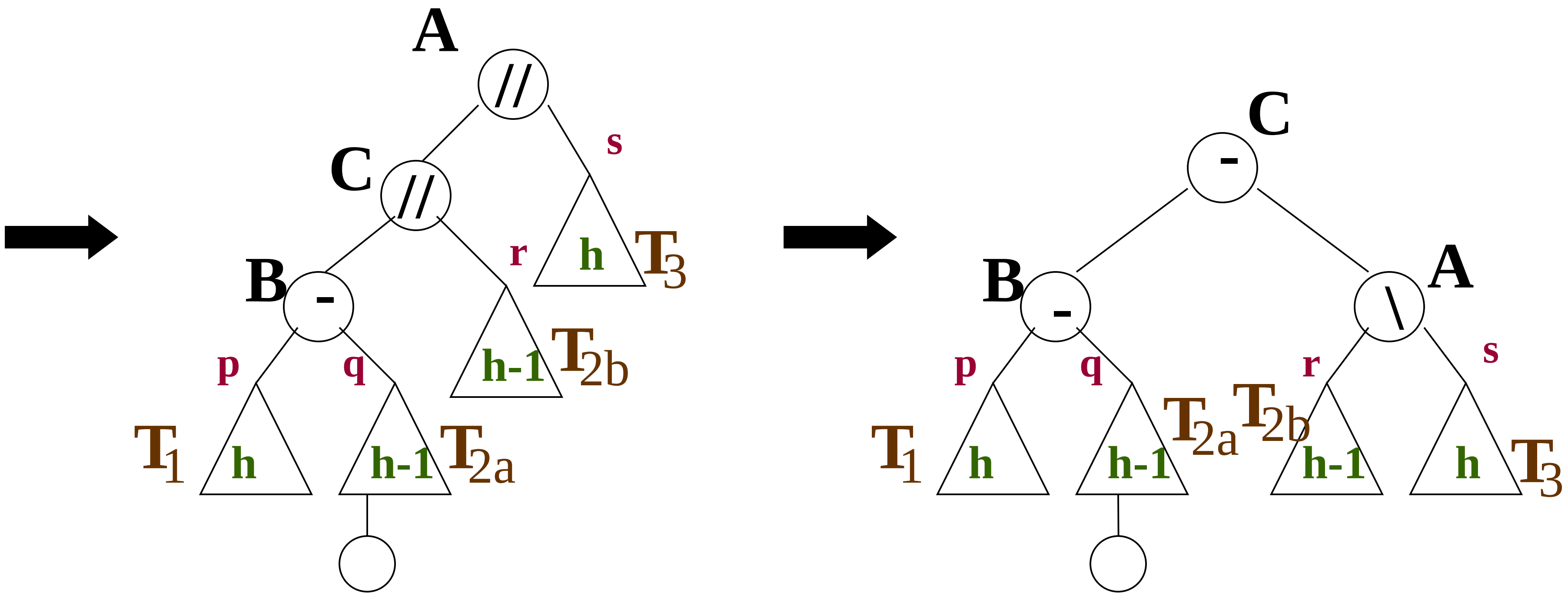
**Cas II - T2b :**

**Insertion en T2b => rotation droite sur B et ensuite gauche sur A**

# Arbres AVL: Insertion

**Cas II - T2a :**

**Rotation gauche sur B** → **Rotation droite sur A**




La propriété ABR est maintenue!