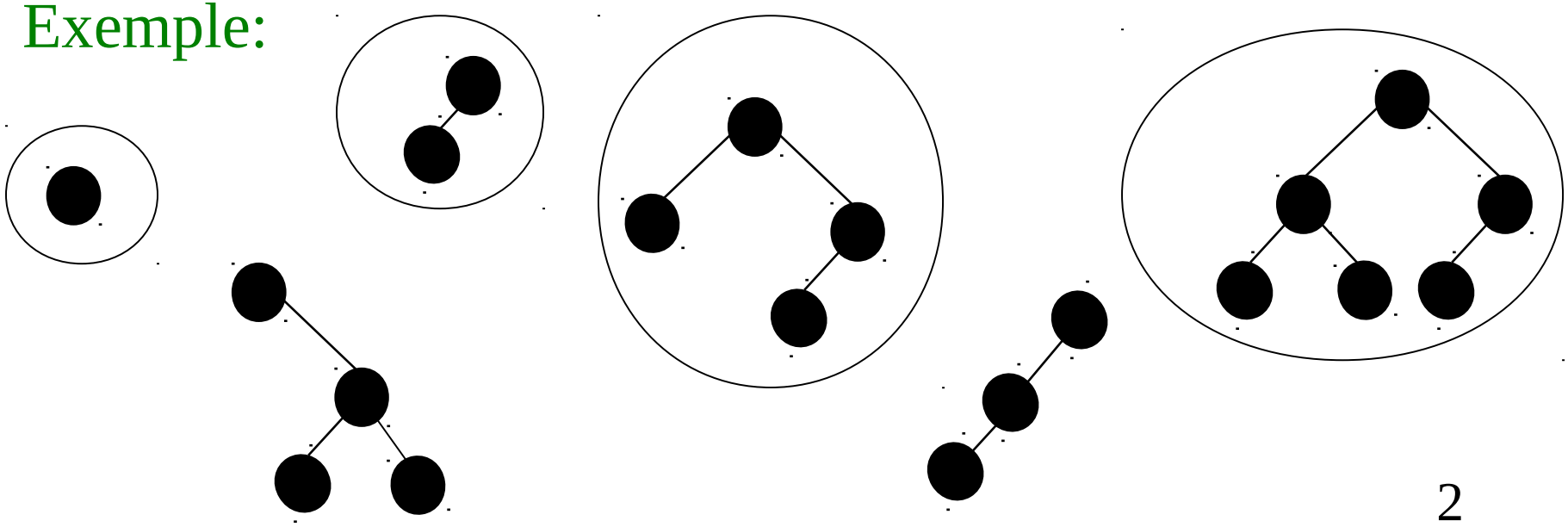
Arbre AVL

- **Arbre AVL** (Adelson-Velskii et Landis):
  - Le meilleur ABR permettant d'avoir à tout moment un arbre **raisonnablement équilibré**.
  - *Idée :* si l'insertion ou la suppression provoque un déséquilibre de l'arbre, rétablir un certain équilibre.
  - Toutes les opérations insertion, suppression, recherche, min, max… sur un arbre AVL avec N noeuds en O(log N) (en moyenne et dans le pire cas!)

Arbres AVL

Arbre AVL (propriété): c'est un ABR tel que la différence des hauteurs du sous-arbre gauche et droit de la racine est d'au plus 1 **et** les sous-arbres gauche et droit sont des AVL

Exemple:



Arbres AVL

Pour plus lisibilité, on remplace les clés associées aux nœuds en utilisant **/ (1), \(-1), -(0), // (2) et \\ (-2)** 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)

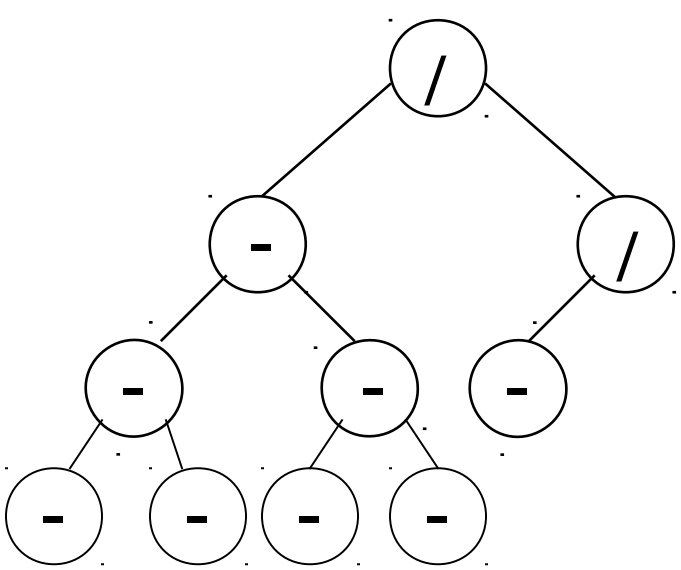
- : équilibré → h(D) = h(G)

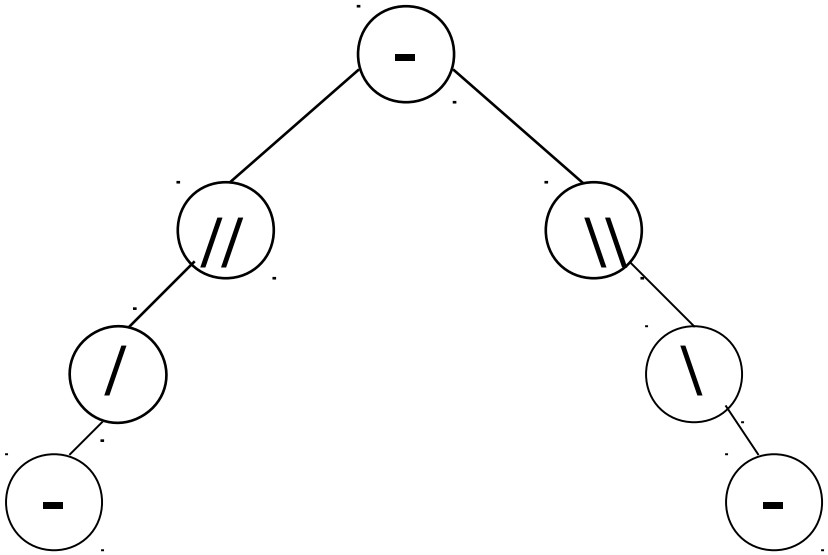
\\ : déséquilibre droit → h(D) > 1 + h(G)

// : déséquilibre gauche → h(G) > 1 + h(D)

Arbres AVL

Exemples :





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

Implémentation des Arbres AVL

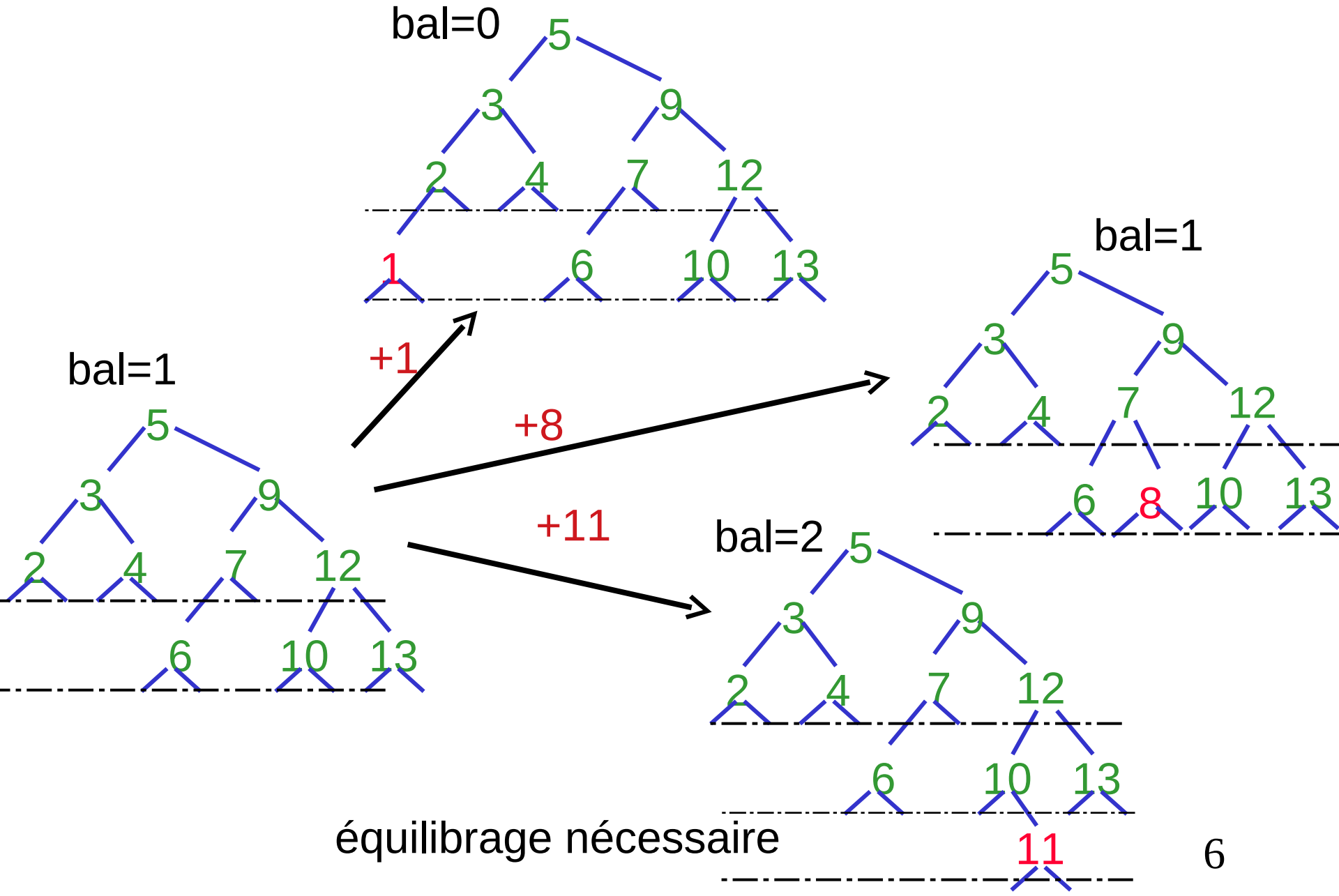
```
typedef struct n {
     int cle ;
     int bal ; /* balance comprise entre –1 et +1 */
     struct n *fgauche, *fdroite ;
} noeud ;
typedef noeud *Arbre ;
```

```
(arbre,entier) AJOUTER (int x, Arbre A) ;
```
/* rend l'arbre modifié et la différence de hauteur : 0 ou +1 */

```
(arbre,entier) ENLEVER (int x, Arbre A) ;
```
/* rend l'arbre modifié et la différence de hauteur : -1 ou 0 */

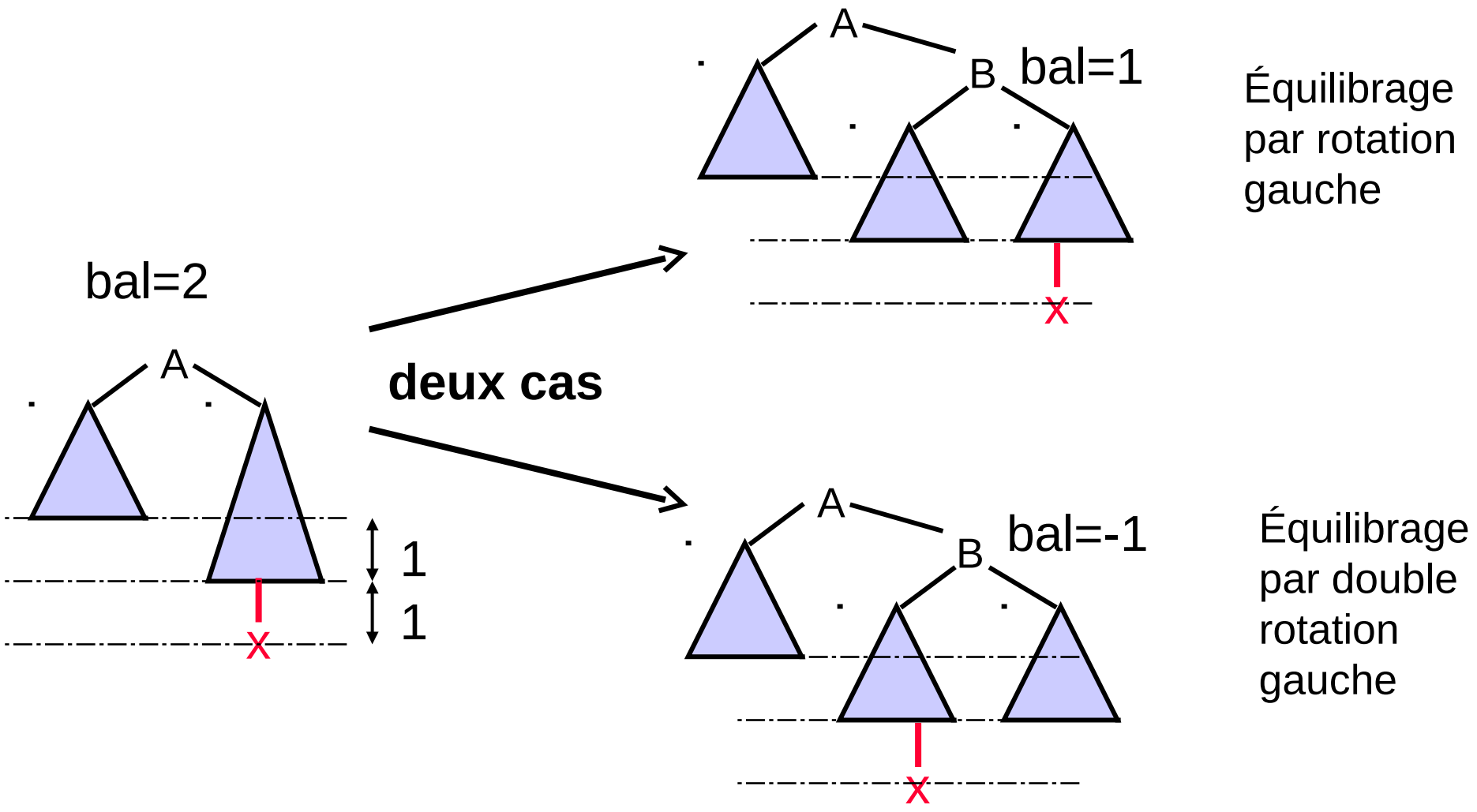
```
(arbre,entier) OTERMIN (Arbre A) ;
```
/* rend l'arbre modifié et la différence de hauteur : -1 ou 0 */

Insertions dans un Arbre AVL



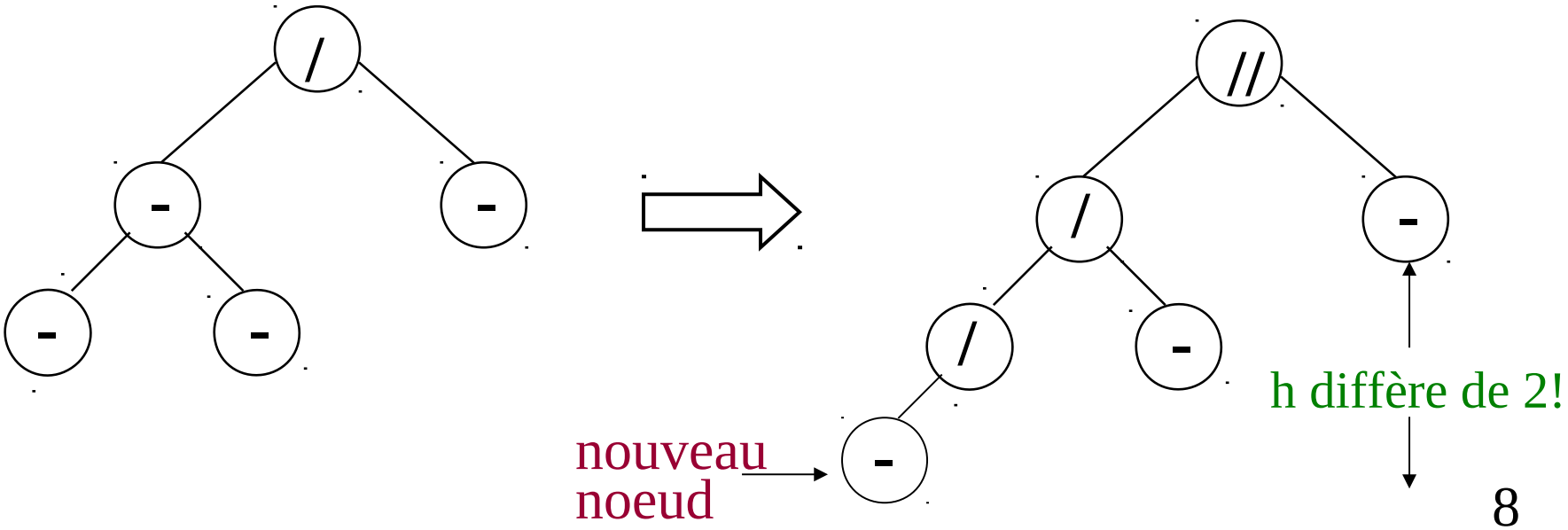
équilibrage nécessaire

Equilibrage après ajout dans un Arbre AVL



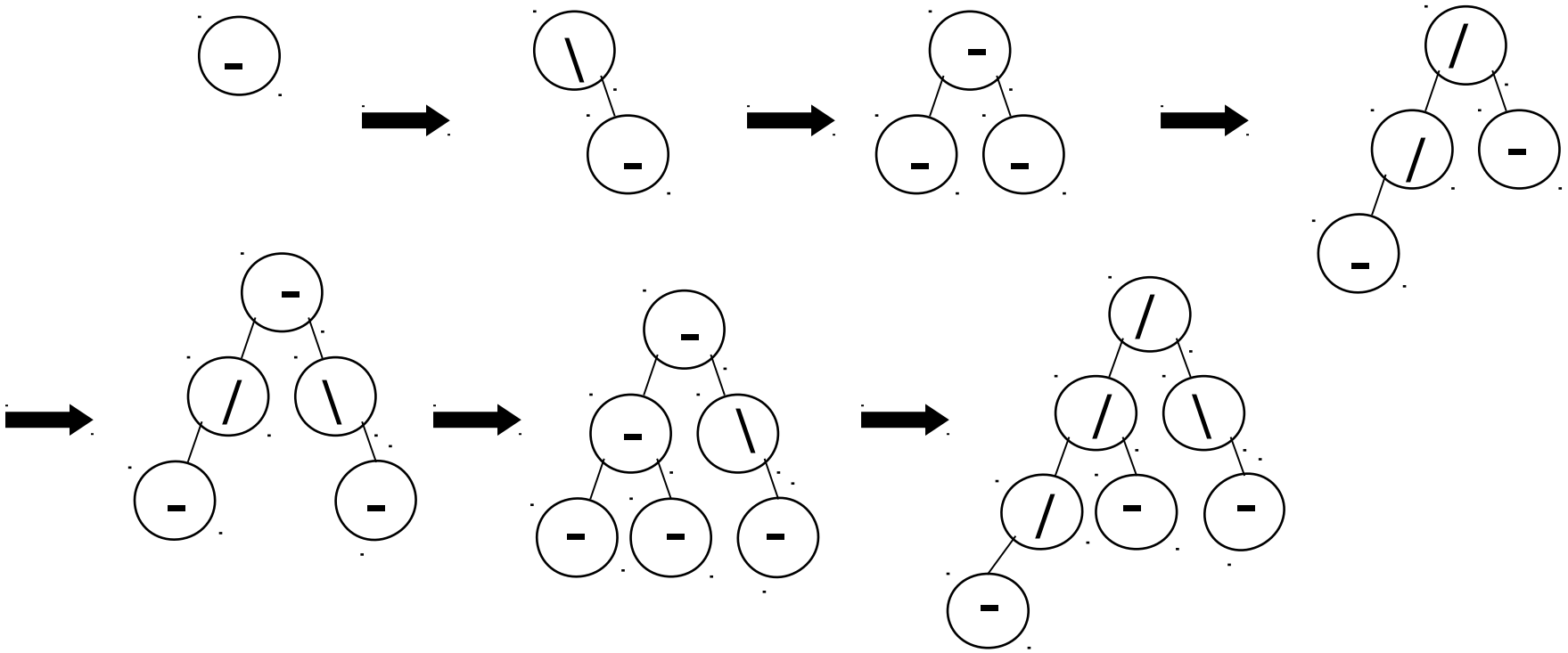
Arbres AVL

Un arbre AVL n’est pas un arbre plein.
Insertions et suppression sont effectuées de la même manière que pour les ABR. Après chaque opération, **on a besion de vérifier la propriété d’AVL!. Car l’arbre peut ne plus l’être!**

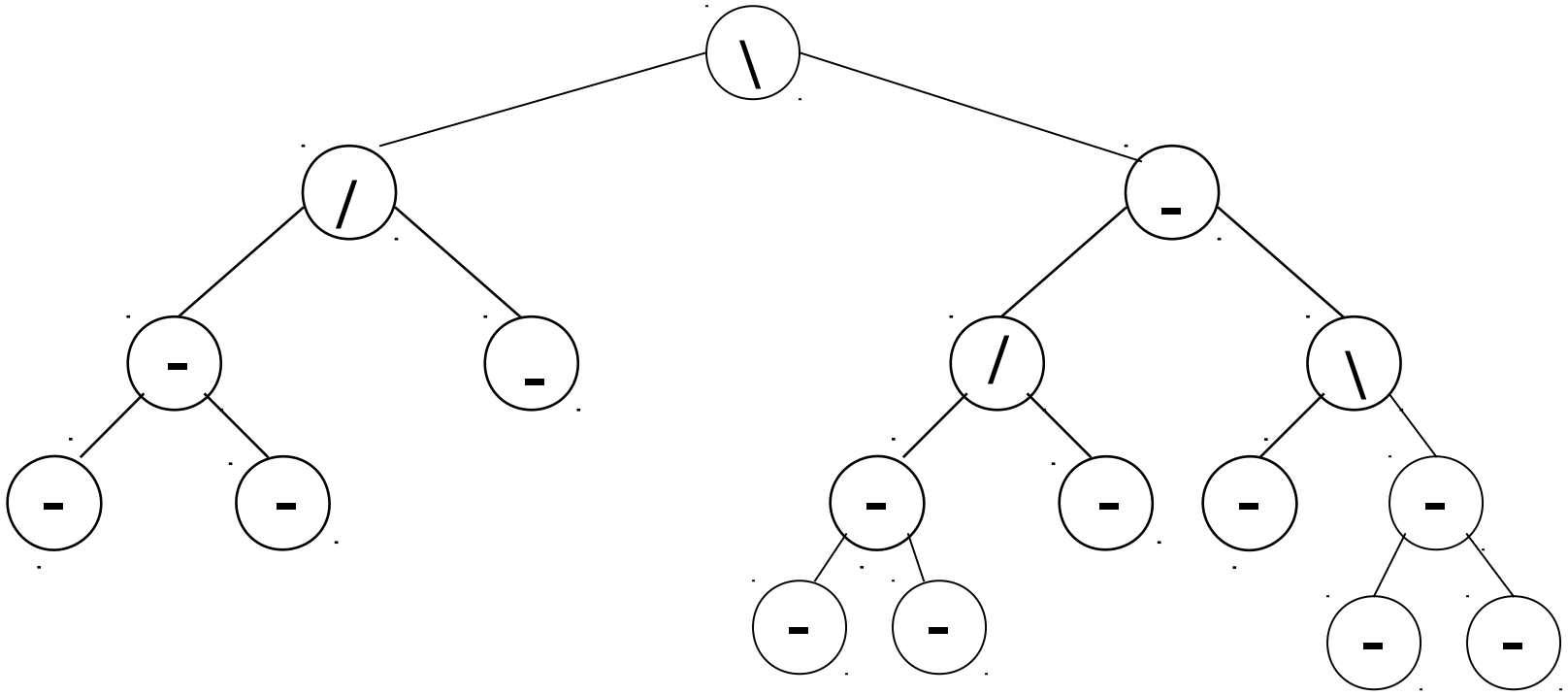


Arbres AVL

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

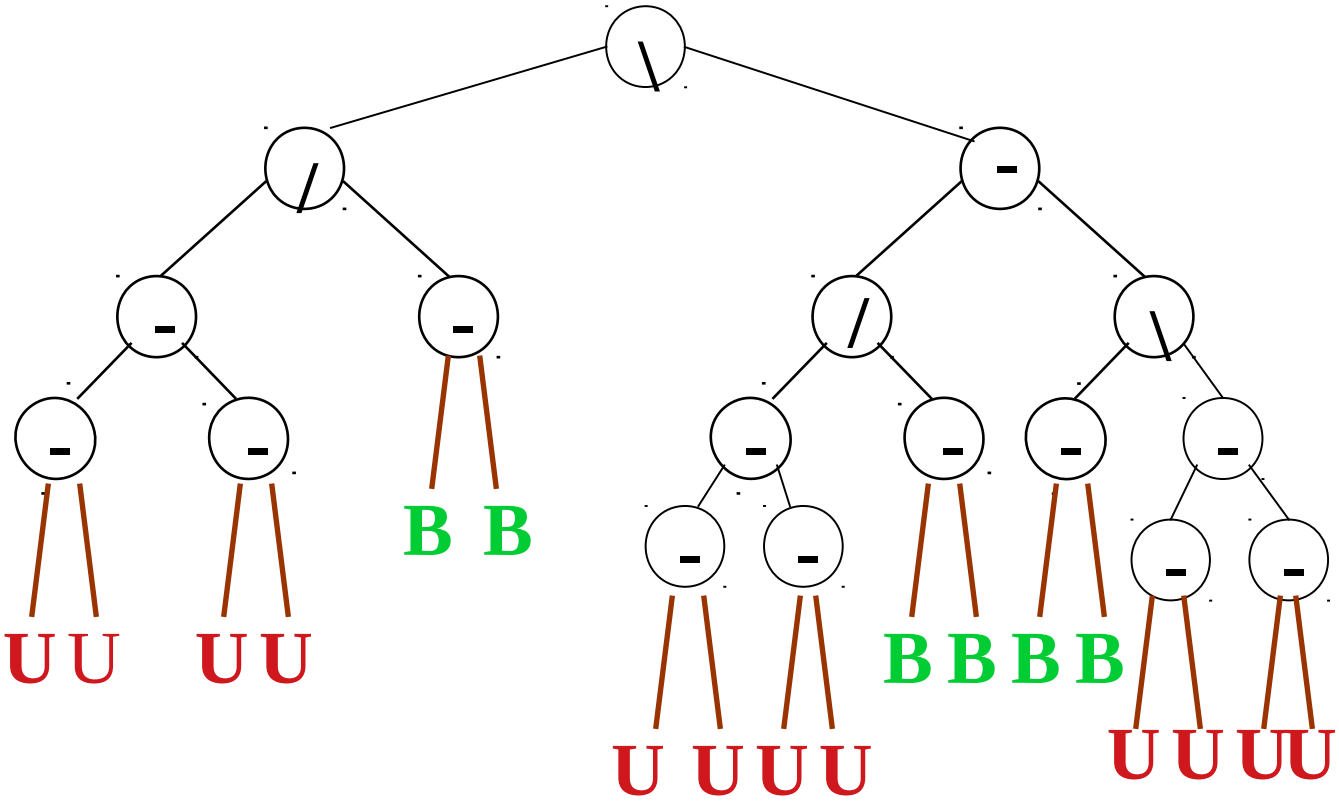
L'arbre devient déséquilibré si l'élément ajouté est le descendant gauche (droit) d'un nœud avec un léger déséquilibre gauche (droit). Alors la hauteur de ce sous-arbre augmente.

Dans les figures suivantes, on note :

U: nouveaux nœuds pouvant déséquilibrer l'arbre

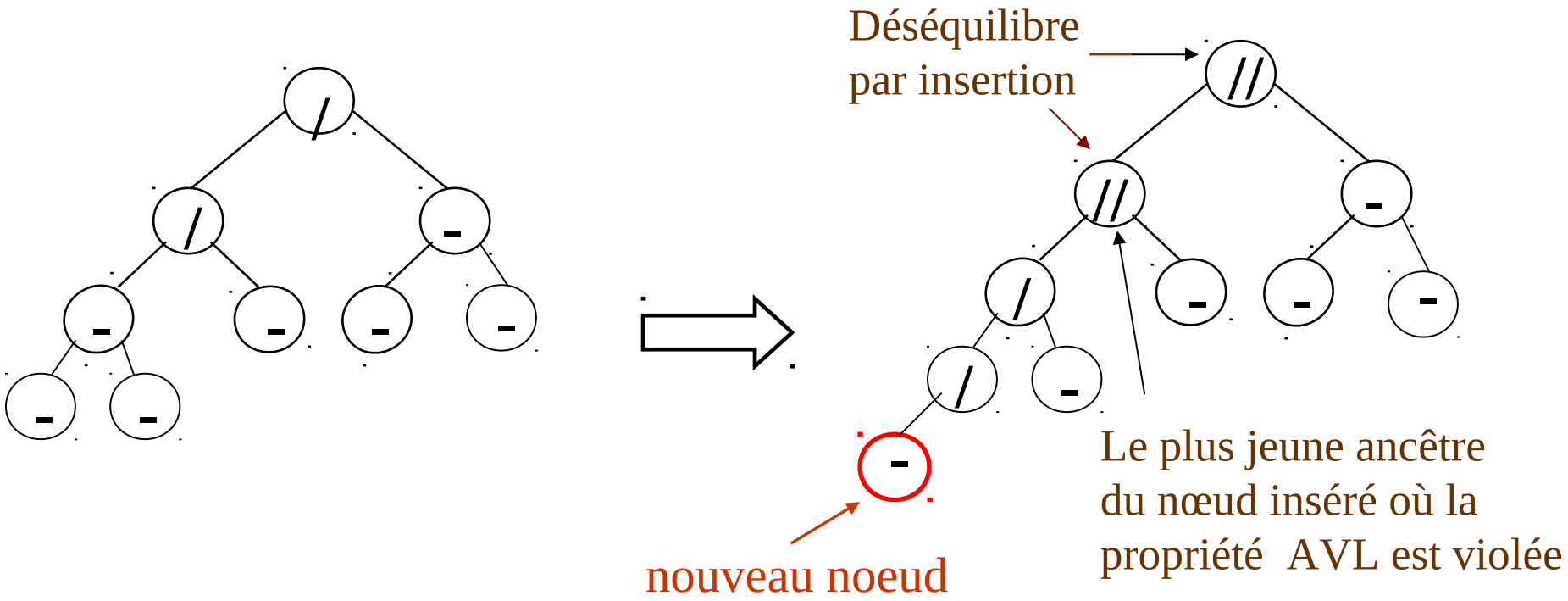
B: nouveaux nœuds laissant l'arbre équilibré

Arbres AVL: Insertion



Arbres AVL: Insertion

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

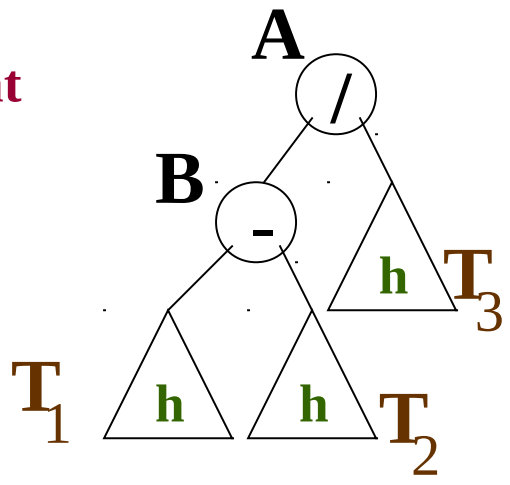


Arbres AVL: Insertion

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

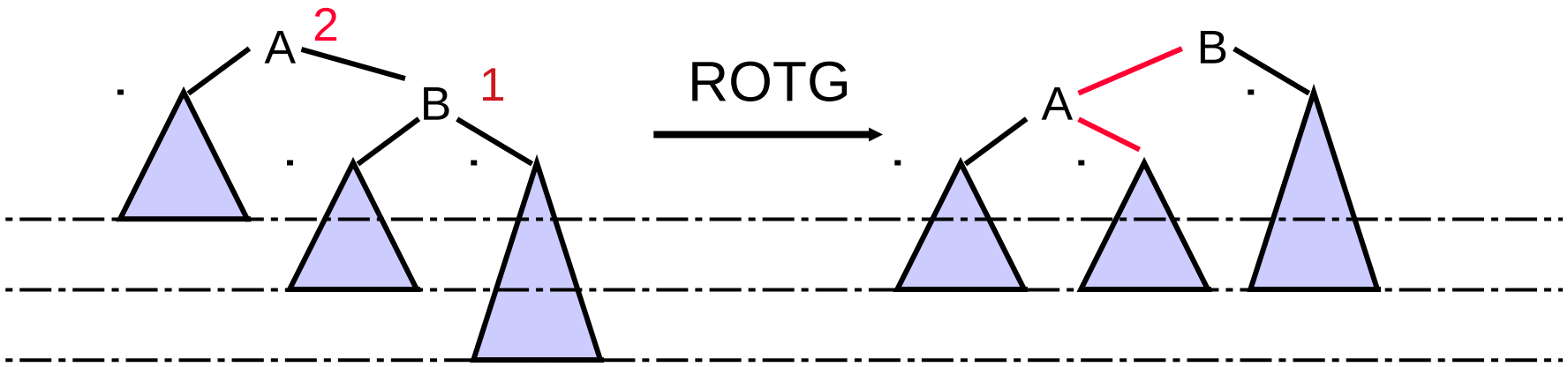


Le même raisonnement peut être utilisé si l 'arbre le plus haut est celui de **droite**

Arbres AVL: Rotation Gauche

$A = (r, A_g, A_d)$ avec A_g, A_d sont AVL

- bal(A) = 2
- bal (A_d) = 1



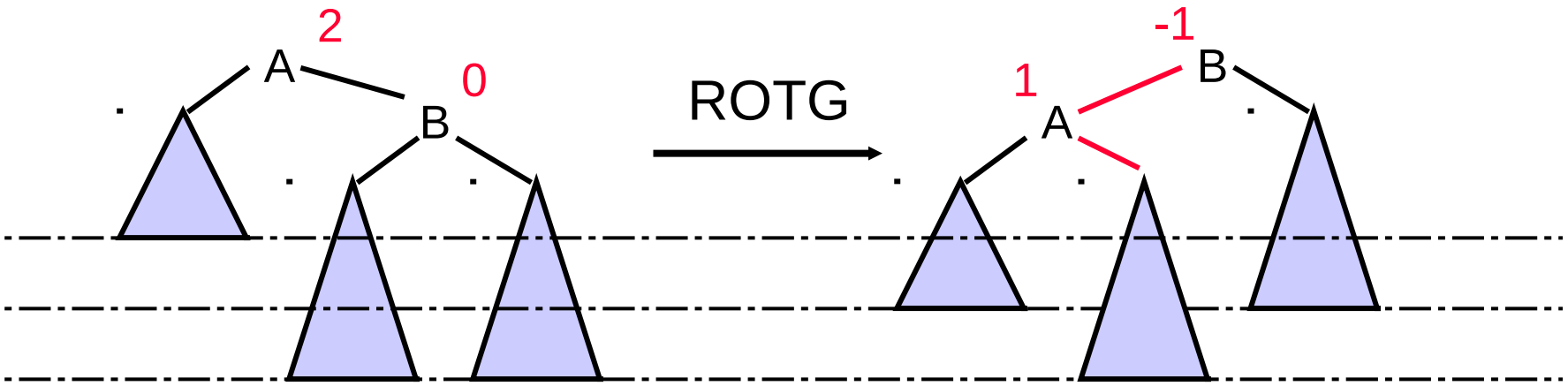
La rotation préserve l'ordre symétrique

Note : elle diminue la hauteur

Arbres AVL: Rotation Gauche (Suite)

$A = (r, A_g, A_d)$ avec A_g, A_d sont AVL

- bal(A) = 2
- bal (A_d) = 0



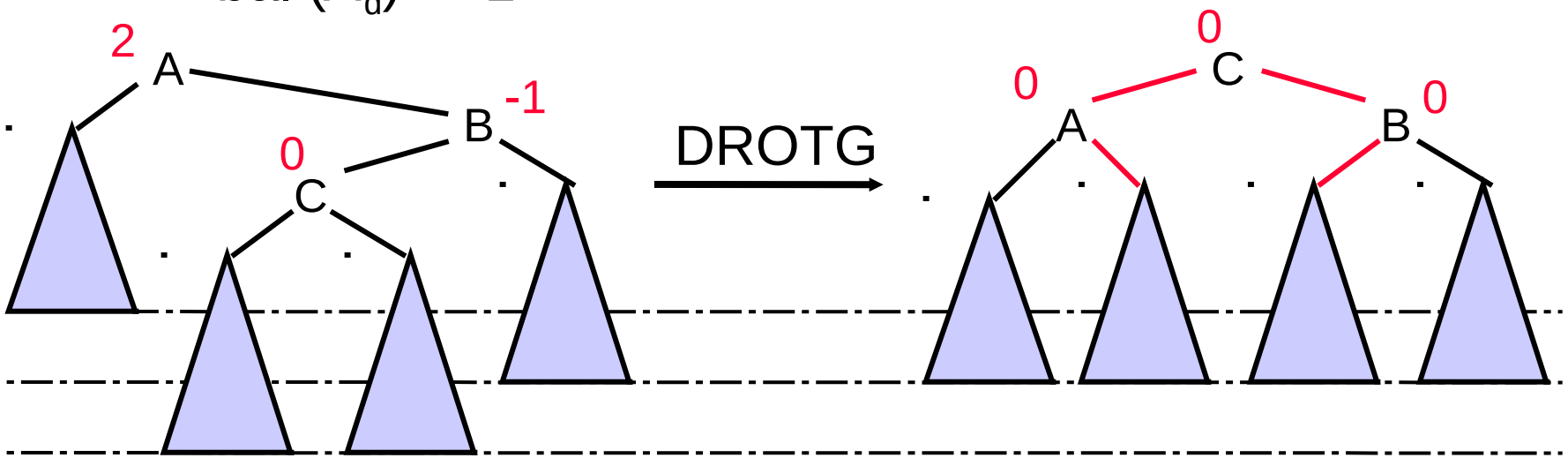
La rotation préserve l'ordre symétrique

Note : elle ne diminue pas la hauteur

Arbres AVL: Double rotation gauche

$A = (r, A_g, A_d)$ avec A_g, A_d sont AVL

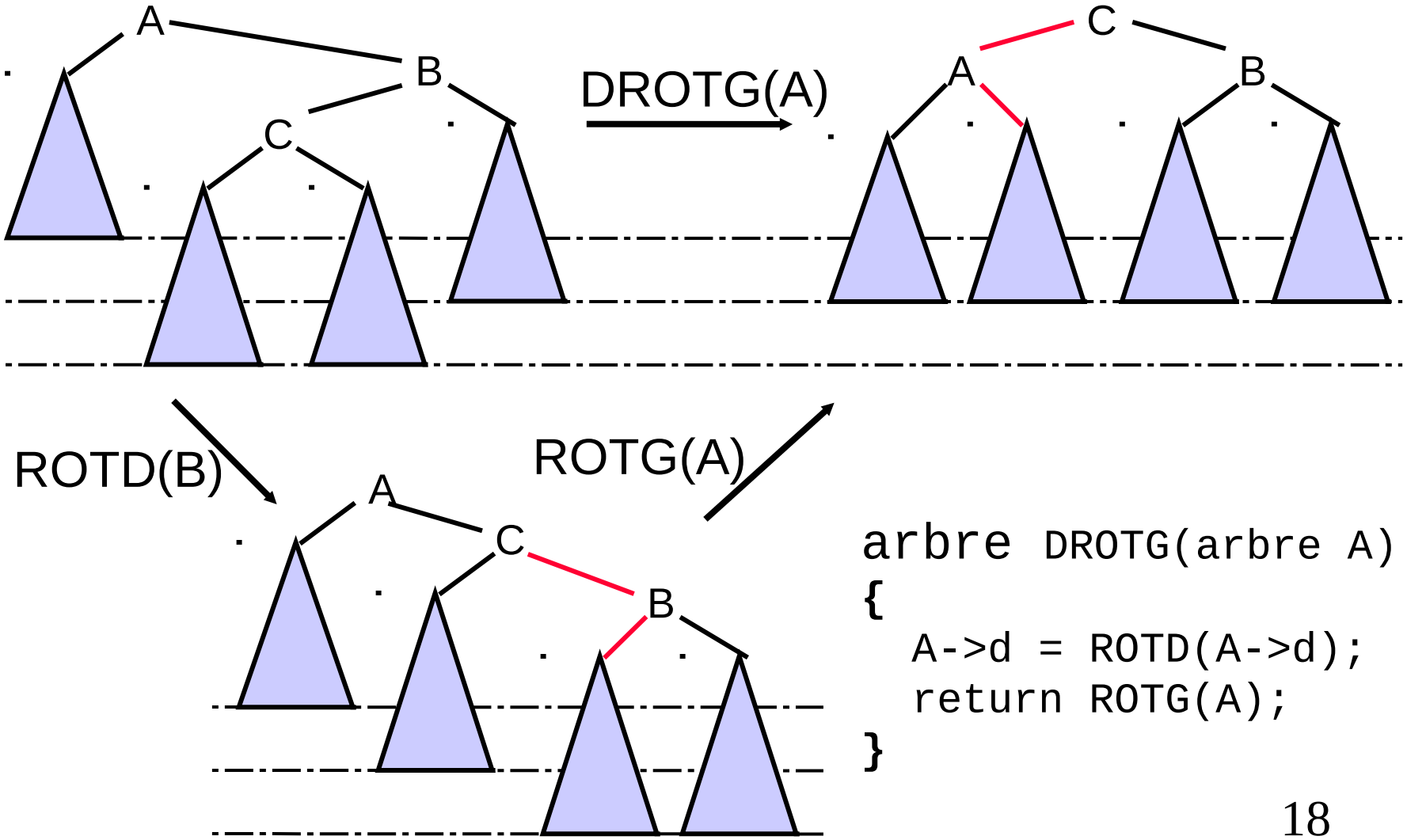
- bal(A) = 2
- bal (A_d) = -1



La rotation préserve l'ordre symétrique

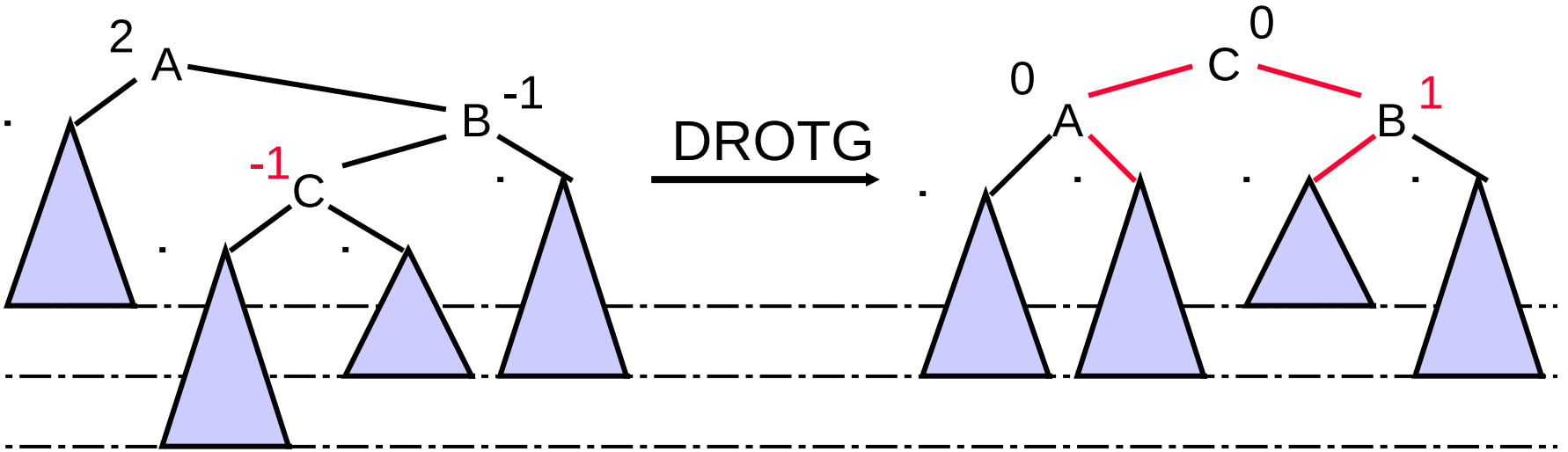
Note : elle diminue la hauteur

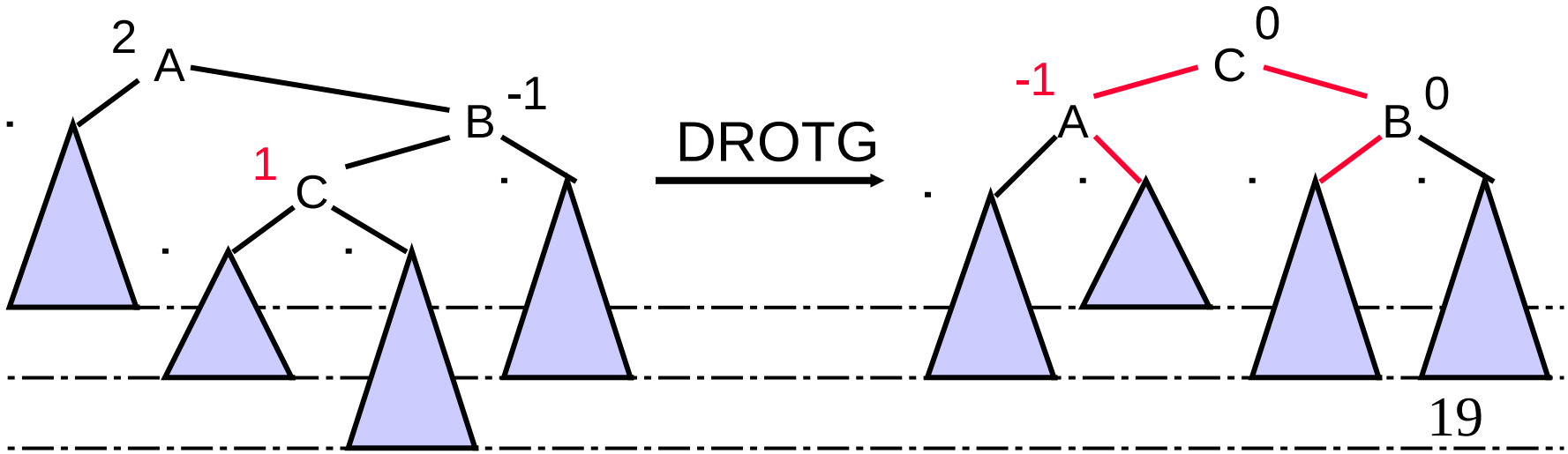
Arbres AVL: Double rotation gauche (suite)




```
arbre DROTG(arbre A)
{
  A->d = ROTD(A->d);
  return ROTG(A);
}
```

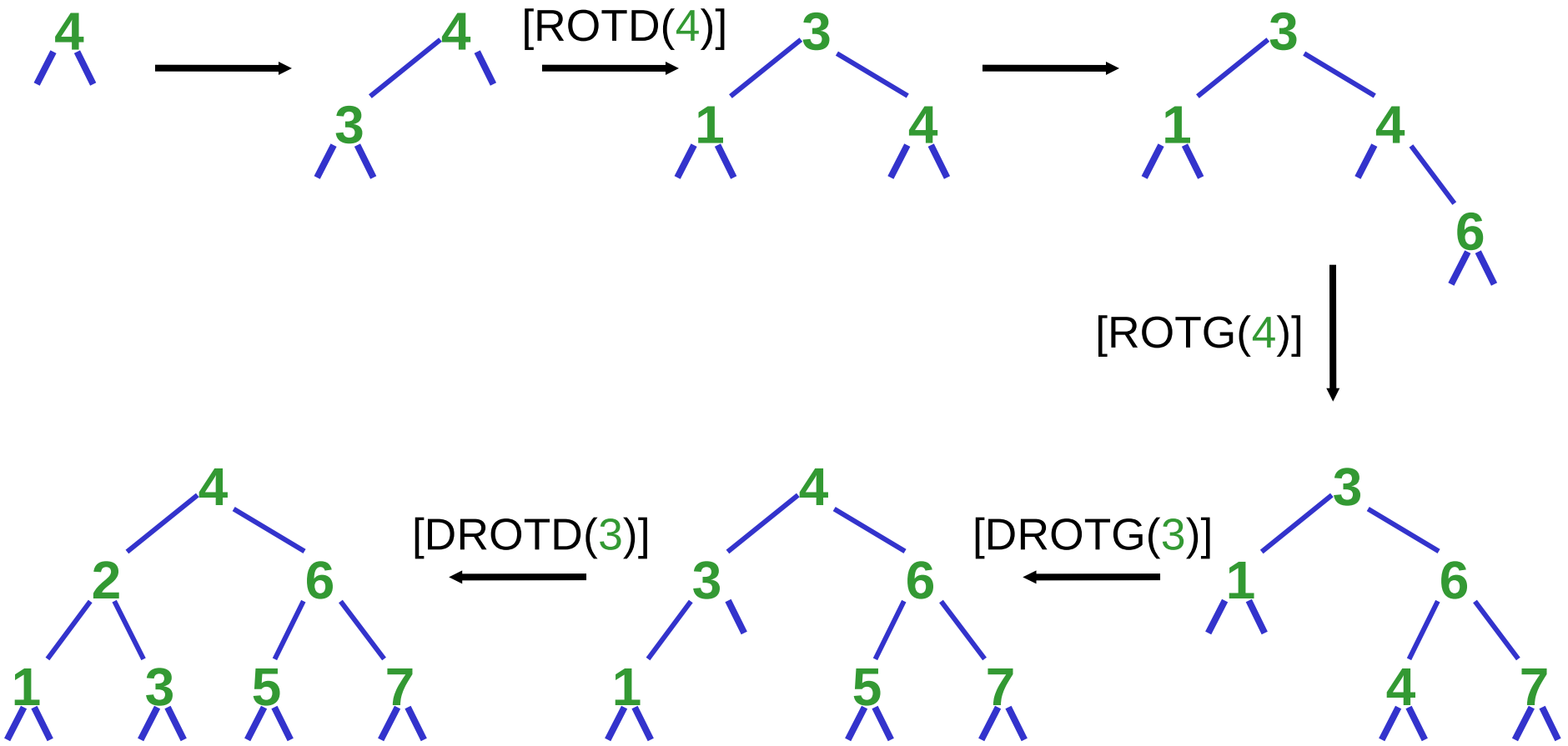
Arbres AVL: Double rotation gauche (suite)





Arbres AVL: Exemples de rotations

Ajouts successifs de **4**, **3**, **1**, **6**, **7**, **5**, **2** dans un arbre vide



Equilibrage

Entrée

A arbre tel que

A_g, A_d sont AVL

$-2 \leq bal(A) \leq 2$

Sortie

A arbre AVL

$[-1 \leq bal(A) \leq 1]$

```
arbre EQUILIBRER(arbre A)
{
  if (A->bal == 2)
      if (A->d->bal >= 0)
            return ROTG(A);
      else {
           A->d = ROTD(A->d);
           return ROTG(A);
          }
      else if (A->bal == -2)
            if (A->g->bal <= 0)
                  return ROTD(A);
            else { A->g = ROTG(A->g);
                  return ROTD(A);
            }
      else return A;
}
```

Arbres AVL: Insertion

- Nous avons défini un arbre **“équilibré”** et nous avons aussi montré comment insérer dans l’arbre en utilisant les algorithmes ABR de manière à maintenir l’équilibre (propriétés AVL) et la propriété ABR.

Arbre AVL: Insertion (Algorithme)

- Algorithme d ’adjonction dans un AVL
  Pour conserver la valeur du déséquilibre en chaque nœud de l ’arbre, on utilisera les déclarations suivantes.

```
typedef struct n
        {
         int val ;
         int bal ;
         struct n * fgauche, *fdroite ;
        } noeud;

typedef nœud *AVL;
```

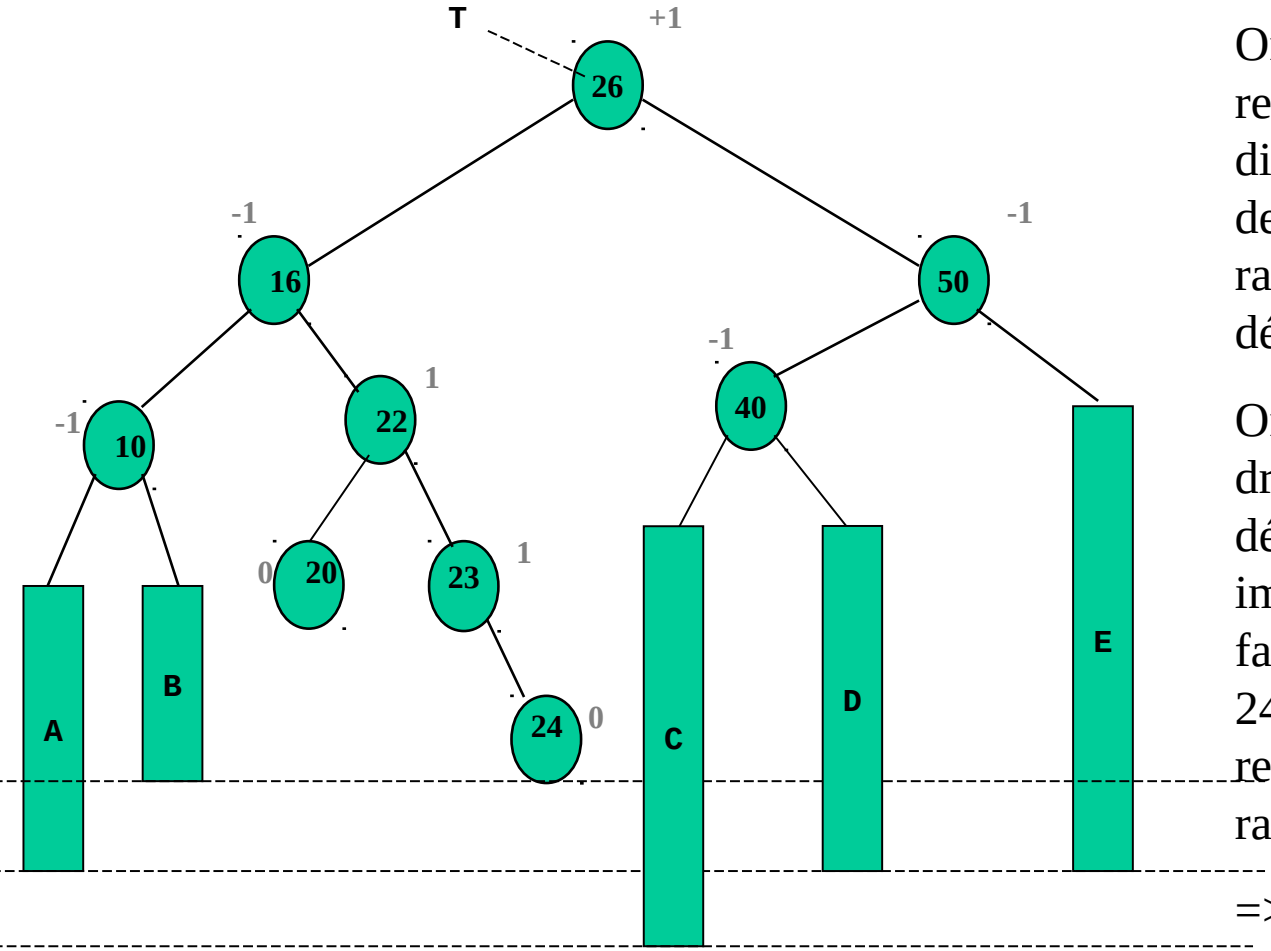
Principe :

Lors de la descente dans l ’arbre à la recherche de la place où l ’on doit ajouter x, on mémorise le dernier sous-arbre A pour lequel le déséquilibre est ± 1.

Après avoir ajouté x à la feuille Y, c ’est uniquement sur le chemin de A à Y qu ’il est nécessaire de modifier les valeurs du déséquilibre. Il faut ensuite faire le cas échéant, un rééquilibrage en A.

Arbre AVL: suppression

- La réorganisation de l ’arbre peut nécessiter plusieurs rotations successives.



On veut supprimer 26, on le remplace par 24, cette suppr diminue la hauteur du sous-arbre de racine 22 et le sous-arbre de racine 16 est alors trop déséquilibré.

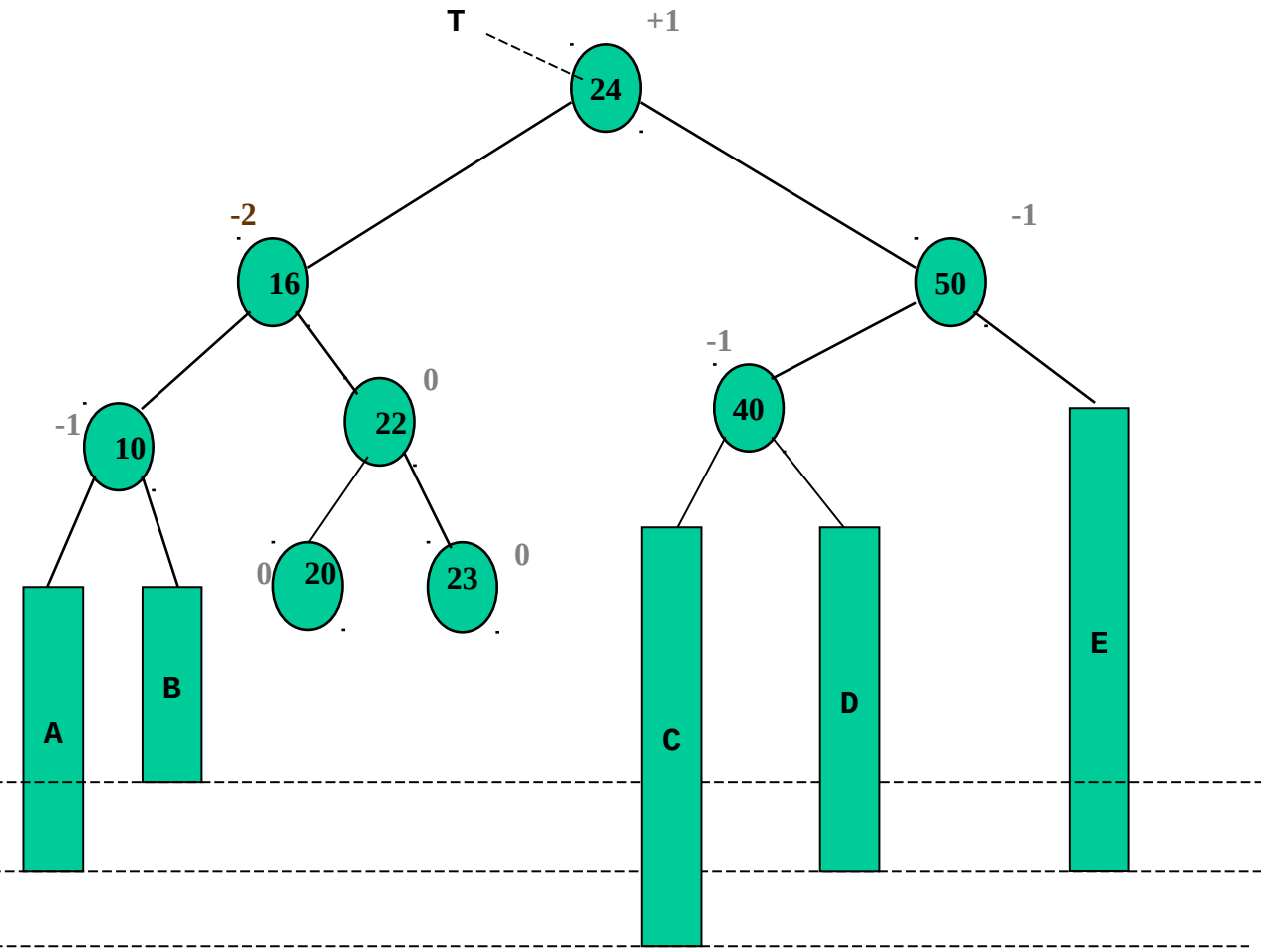
On le réorganise par une rotation droite, mais cela accentue le déséquilibre du niveau immédiatement supérieur et il faut faire une rotation droite gauche en 24. Les rotations peuvent ainsi remonter en cascade jusqu ’à la racine de l ’arbre.

=> 1.5 log2 n rotations.

=> la suppression est en O(log2 n)

Arbre AVL: suppression

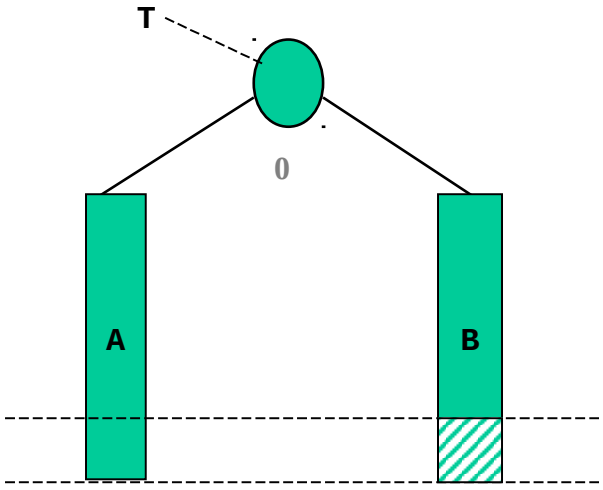
- L'arbre n'est plus un AVL



On distingue différents cas…

Arbre AVL: suppression

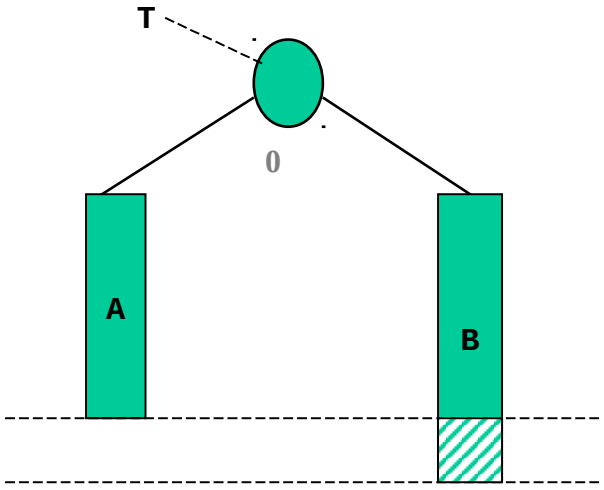
- **Cas I:**



• Rien à faire, car la hauteur de l'arbre n'a pas été modifié

• Avec -1 même situation

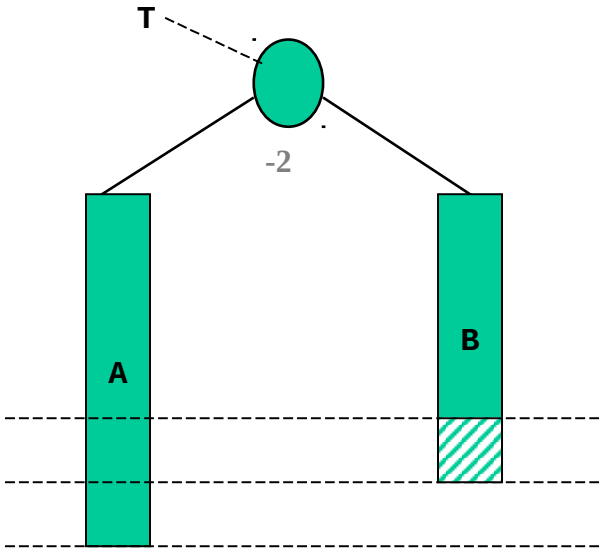
Cas II:



• Ici la hauteur du sous-arbre va évoluer(+1) localement : aucun déséquilibre n 'est apparu, au contraire, le sous arbre devient équilibré. Des déséquilibres peuvent apparaître plus haut!

Arbre AVL: suppression

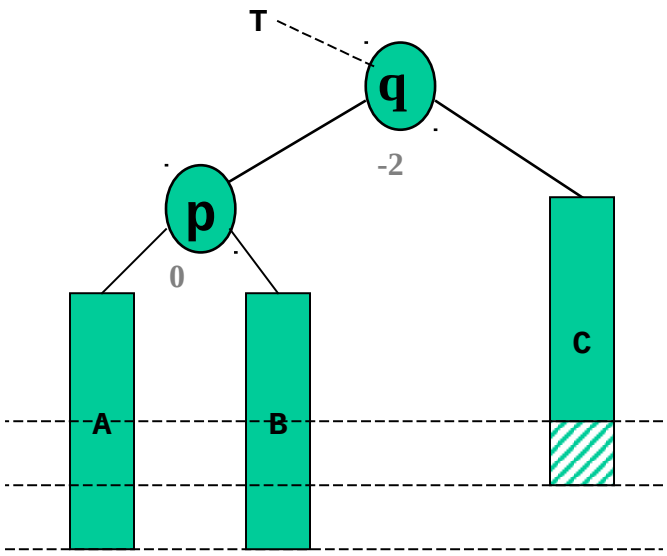
- **Cas III:**



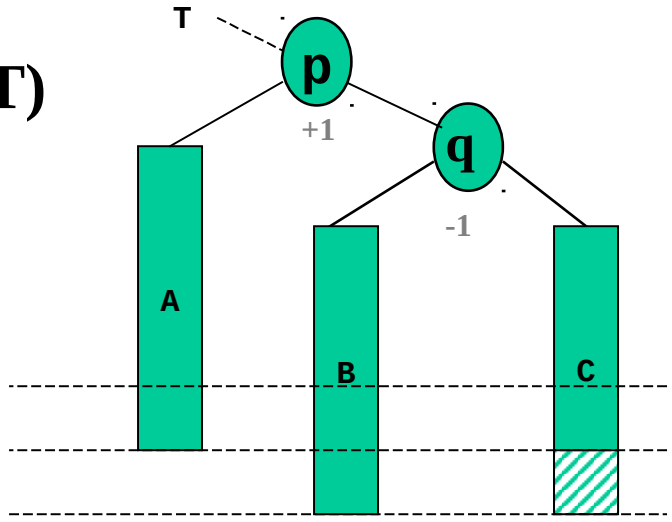
Ici la hauteur du sous-arbre A n'a pas évolué mais le déséquilibre est passé à -2 : il faut intervenir; on distingue les différents cas de figure qui sont liées au fils gauche :

Arbre AVL: suppression

- **Cas III.1:**



RD(T)

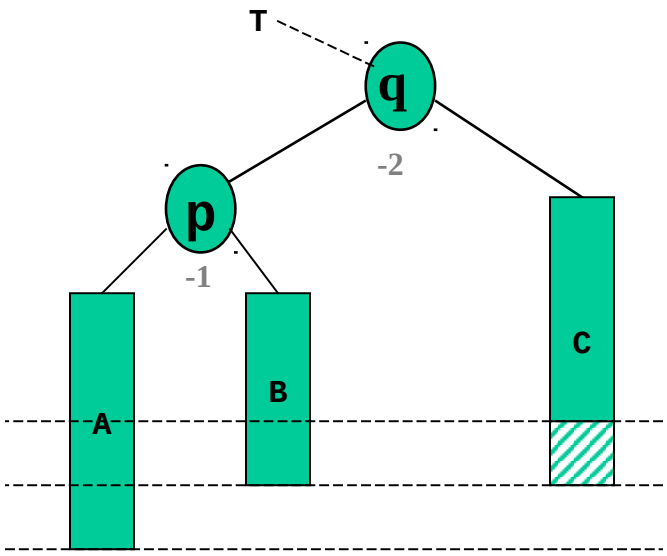


Il y a arrêt du traitement ici, puisque :

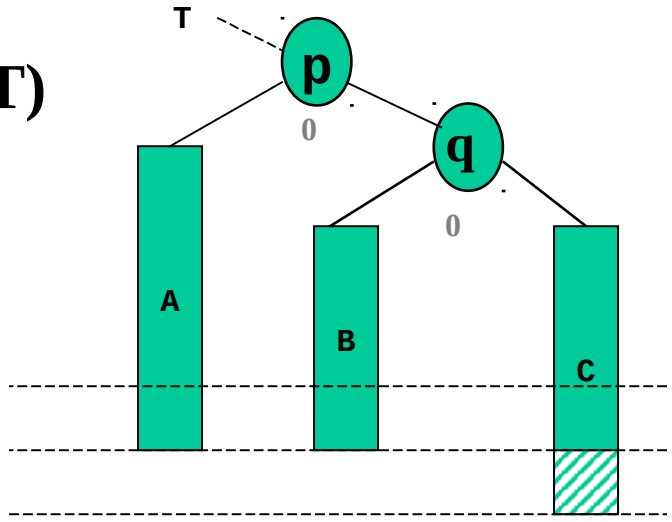
- le sous-arbre est équilibré
- sa hauteur n'a pas été modifiée (il est donc inutile de propager le résultat vers le haut)

Arbre AVL: suppression

- **Cas III.2:**



RD(T)



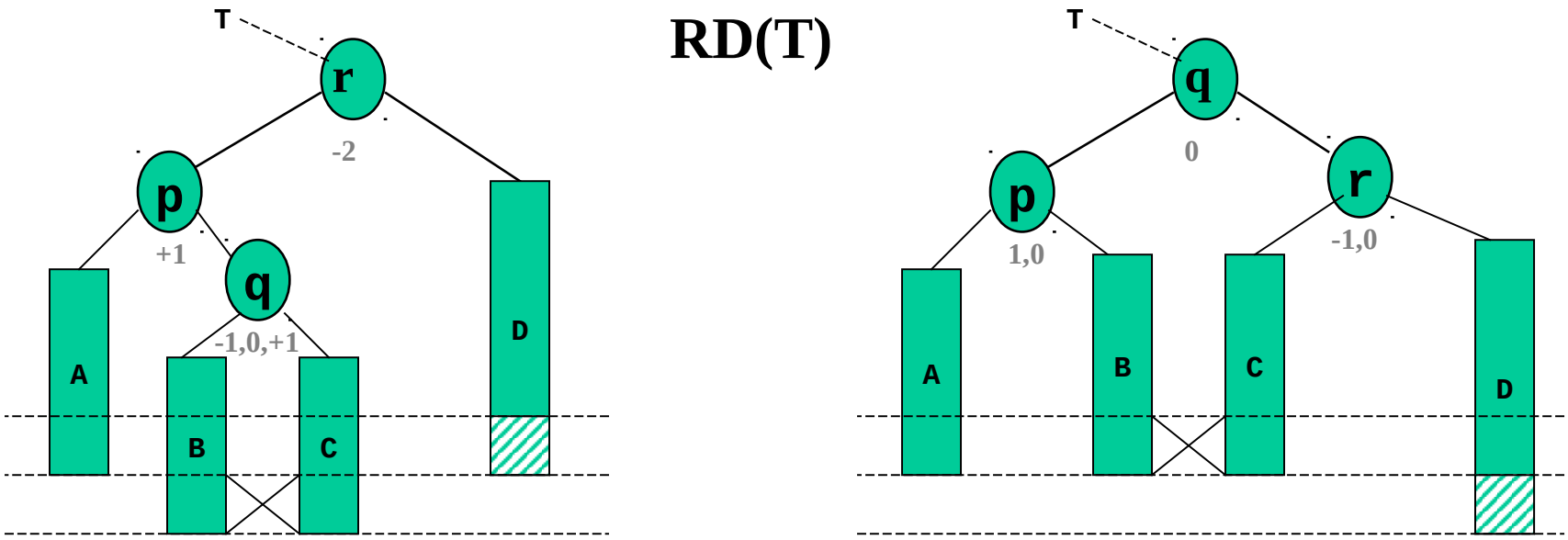
Le sous-arbre est rééquilibré, mais la hauteur a été modifiée

il faut remonter l ’information au dessus de T pour procéder éventuellement à des rééquilibrage

=> appliquer le même principe que celui qui vient d’être appliqué en considérant I,II et les différents cas de III.

Arbre AVL: suppression

- **Cas III.3:**



Le sous-arbre est rééquilibré, mais sa hauteur a diminué de 1

=> remonté de l'information comme en III.2

Arbre AVL: suppression

Principe de l ’algorithme :

- Réorganisation de l’arbre de la feuille supprimée jusqu’à la racine de l’arbre avec éventuellement des rotations (on s’arrête pour les cas I ou III.1)
  - recalcul des déséquilibres occasionnés par la suppression et éventuelle exécutions des rotations nécessaires du fait de nouveaux déséquilibres
  - la propagation continue tant que l’arbre demeure déséquilibré lors de la remontée

Au pire cas le nombre de rotations est $\log_2 n$

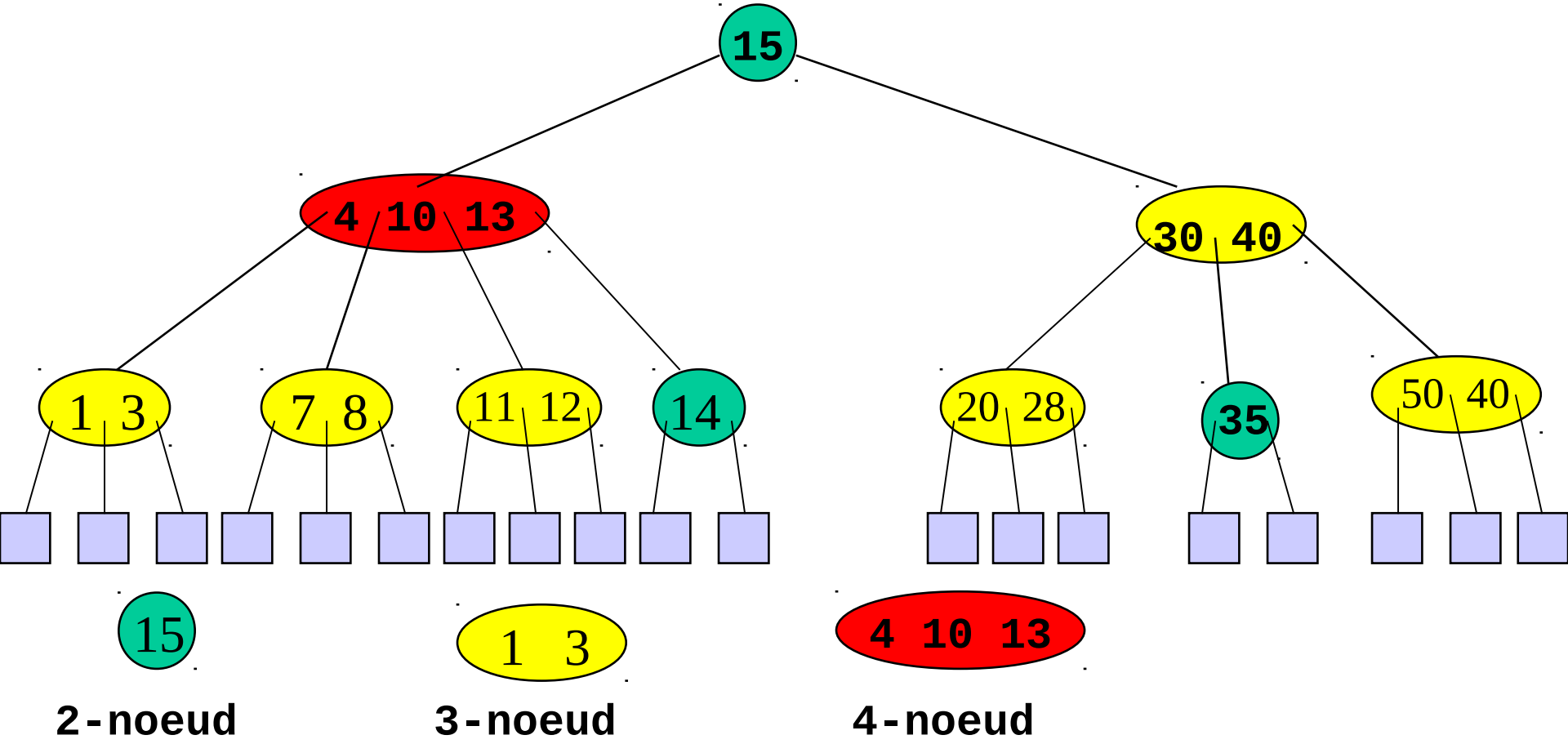
Arbres 2.3.4

- Pour éviter les cas d ’arbres de recherche dégénérés, on peut aussi faire varier le nombre de directions de recherche à partir d ’un nœud.

- **Définition générale** : Un arbre de recherche est un arbre général étiqueté dont chaque nœud contient un k-uplet d’éléments distincts et ordonnées (k=1,2,3…). Un nœud contenant les éléments $x_1 < x_2 \ldots < x_k$ à k+1 sous-arbres, tels que :

  - tous les éléments du premier sous-arbre sont inférieurs ou égaux à x_1
  - tous les éléments du ième sous-arbre (i=2,…,k) sont strictement supérieurs à x_{i-1} et inférieurs ou égaux à x_i
  - tous les éléments du $(k+1)^{ième}$ sous-arbre sont strictement supérieurs à x_k

Arbres 2.3.4



Définition : Un **arbre 2.3.4** est un arbre de recherche dont les nœuds sont de trois types, 2-nœud, 3-nœud, 4-nœud, et dont toutes les feuilles sont situées au même niveau

Arbres 2.3.4

- la hauteur reste logarithmique par rapport au nombre de nœuds
  - ∃ algos de rééquilibrages qui maintiennent un arbre 2.3.4 après ajout ou suppr en effectuant une suite de rotations sur chemin de la racine à une feuille
- O(log n) pour recherche, ajout et suppression
- implantation efficace sous la forme d'arbres binaires de recherches bicolores

Arbres 2.3.4

Propriété : La hauteur h(n) d ’un arbre 2.3.4 contenant n élément est en $\theta(\log n)$. Plus précisément, on a l’encadrement suivant :

$$\log_4(n+1) \leq h(n)+1 \leq \log_2(n+1)$$

Preuve : on considère les arbres 2.3.4 extrémaux:

- contenant le moins d’éléments : (que des 2-nœuds)
  - puisque toutes les feuilles sont au même niveau,le nombre de nœuds : $2^0+ 2^1+... 2^h= 2^{h+1}-1$
- contenant le plus d’éléments : (que des 4-nœuds)

  $4^0+ 4^1+... 4^h= 4^{h+1}-1$

  on en déduit qu ’un arbre 2.3.4 contenant n nœuds et de hauteur h(n) :

  $2^{h(n)+1}-1 <= n <= 4^{h(n)+1}-1$

  d’où l’on tire l ’encadrement de la hauteur

Arbres 2.3.4

Algorithme de recherche (principe)

Soit M un arbre 2.3.4, Soit x un élément à rechercher dans M.

- x est comparé avec le(s) éléments x_1,.., x_i ($i \in [1..3]$) de la racine de M
  - si $\exists\ j \in [1..i]$ tq. X= xj alors trouvé
  - si $x \leq x_1$ => recherche dans le premier sous-arbre de M
  - si $x_j < x < x_{j+1}$ ($j \in [1..i-1]$) => recherche dans la (j+1)ième sous-arbre de M
  - si $x > x_j$ => recherche dans le dernier sous-arbre de M
- si la recherche se termine sur une feuille qui ne contient pas x => $x \notin M$

Exercice Implanter cet algorithme,

- on utilisera les déclarations suivantes :

```
typedef struct s
      {
        int n /* nombre de pointeurs */
        struct s * sArbre[4]; /* sous arbres */
        int elements[3]; /* les éléments */
      } nœud;
typedef nœud * Arbre234;
```

-

Arbres 2.3.4 : Recherche

```
typedef struct s {
                    int n; /* nombre de pointeurs */
                    struct n * sArbre; /* sous arbres */
                    int *elements; /* les éléments */
          } nœud;
typedef nœud * Arbre234;
int  RechercheCle  (int x; Arbre234 A )
{  int pos ;
   if (A ==NULL )
                    return 0;
        else {
                pos = position (x, A) ;
                if ( (x == A->elements [ pos ] ) )
                    return 1;
                  else
                    return RechercheCle (x, A->sArbre [ pos ] ) ;
             }
}

int position (int x; Arbre234 A)
/* plus grand pos tel que A->elements[pos] ≤ x */
{
        int pos, trouve =0;
        for (pos=0; pos<A->n && !trouve;  pos++){
                    trouve = (x<= A->elements[pos]);
        }
        return pos;
}
```

Arbres 2.3.4

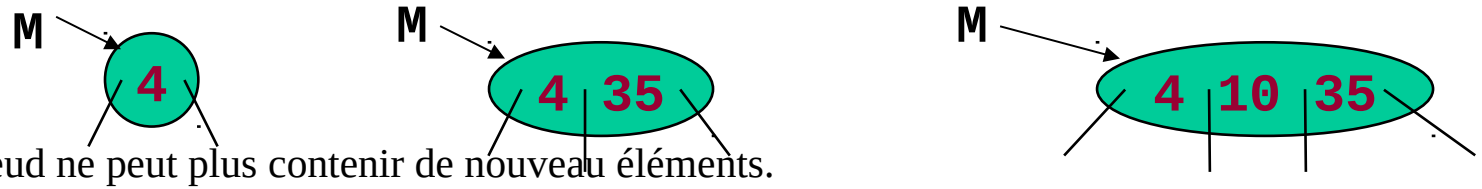
Ajout d ’un élément

L’ajout d’un nouvel élément ne pose problème que si la feuille qui doit le recevoir contient déjà 3 éléments.

Dans ce cas, il faut ajouter un nouveau nœud à l ’arbre et le réorganiser.

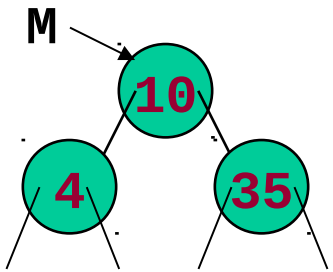
Adjonction avec éclatement en remonté :

On ajoute successivement 4, 35, 10, 13, 3, 30, 15, 12, 7, 40, 20, 11, 6



Ce nœud ne peut plus contenir de nouveau éléments.

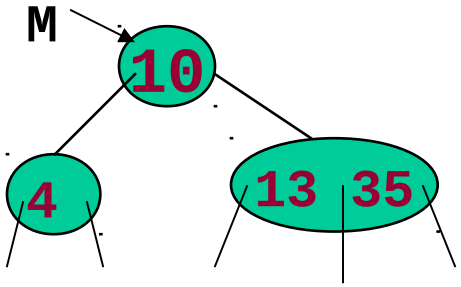
On remarque que du point de vue recherche, M est équivalent à l ’arbre binaire :

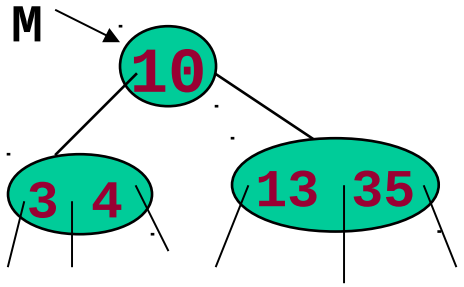


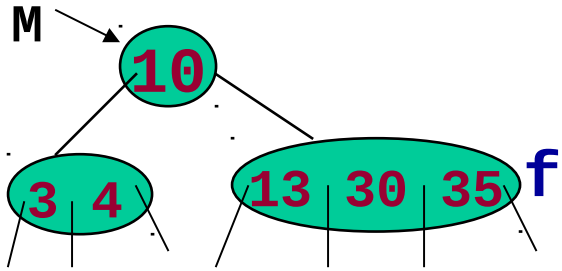
Cet arbre est un arbre 2.3.4, on peut y ajouter de nouveaux éléments :

13, puis 3, puis 30

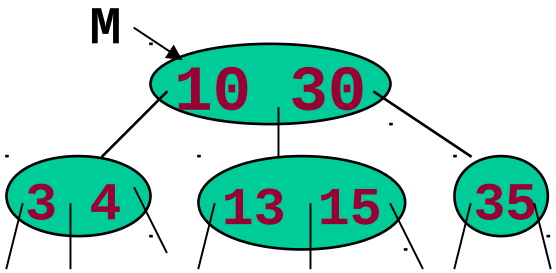
Arbres 2.3.4

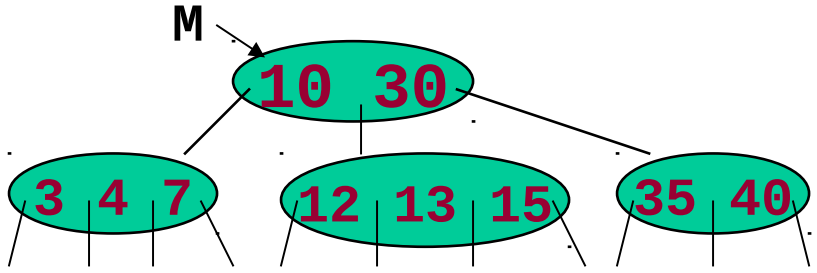






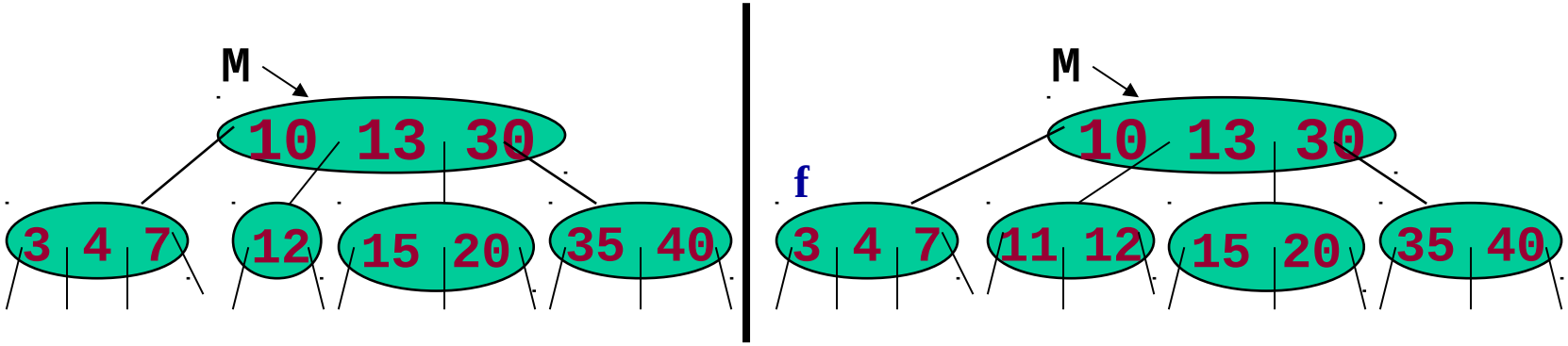
L ’ajout de 15 provoque l ’éclatement de la feuille f en deux 2-nœuds contenant respectivement l
plus petit et le plus grand élément de f. L’élément médian 30 doit être ajouté au nœud père de
f, il y a alors de la place pour 15 dans le même nœud que 13 qui devient alors un 3-noeud





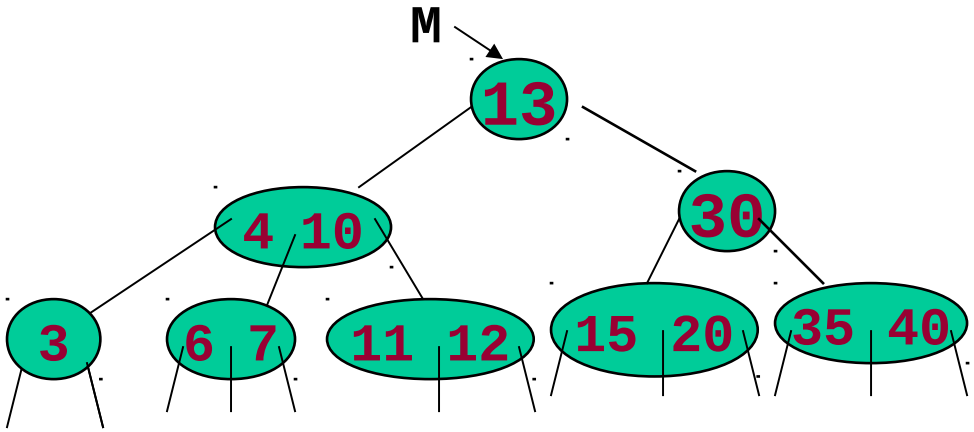
Arbres 2.3.4

L ’ajout de 20 entraîne un éclatement de la feuille contenant 12, 13 et 15



L ’adjonction de 6 provoque l ’éclatement de la feuille f, la remonté de 4 fait éclater à son tour la racine de l ’arbre en 2-noeuds

=> les éclatements peuvent remonter en cascade sur toute la hauteur de l ’arbre.

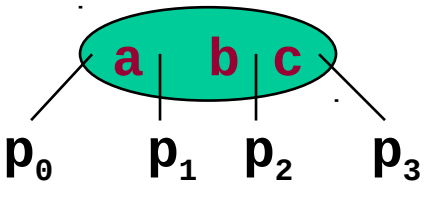


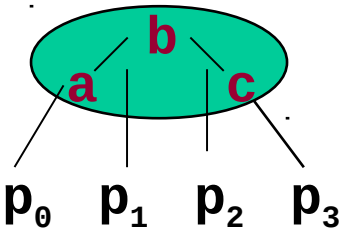
Arbres 2.3.4

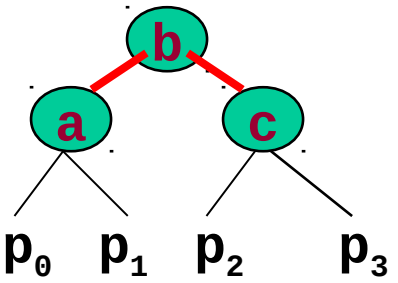
- **Pour éviter des éclatements de bas en haut**, il suffit de travailler sur des arbres qui ne contiennent jamais deux 4-nœuds qui se suivent. Dans ce cas toute adjonction provoque au plus un éclatement.
- **Ceci peut être réalisé** en éclatant les 4-nœuds à la descente : Lors de la recherche de la place de l ’élément à ajouter, on parcourt un chemin à partir de la racine jusqu ’à une feuille; seuls les 4-nœuds de ce chemin risquent d ’éclater suite à l ’adjonction.
- On prévient ce risque en les faisant éclater, au fur et à mesure de leur rencontre avant de réaliser l ’adjonction. (Ceci provoque parfois des éclatements inutiles)

Une représentation des arbres 2.3.4 : les arbres bicolores

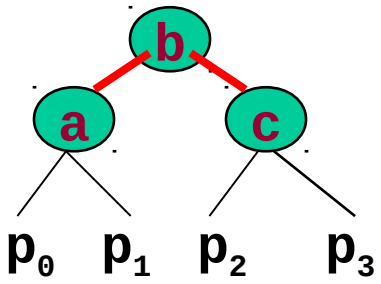
- Définition : Un arbre bicolore est un arbre binaire de recherche (ABR) dont les nœuds portent une information supplémentaire (**rouge** et **noir**).
- Les 4-nœuds et 3-nœuds sont transformés en arbre binaire de recherche.

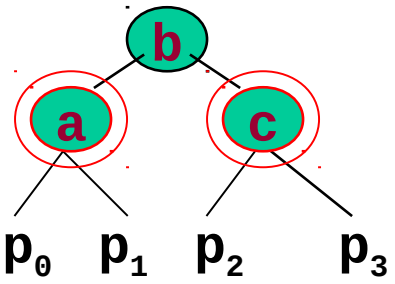






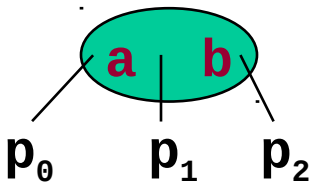
- Double trait si le lien appartient à un nœud de l ’arbre 2.3.4 (relie des nœuds jumeaux)
- On peut aussi représenter par un double cercle les nœuds vers lesquels « pointent » des doubles traits



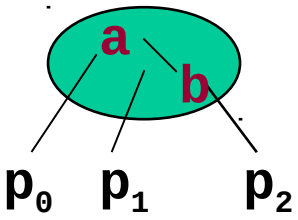


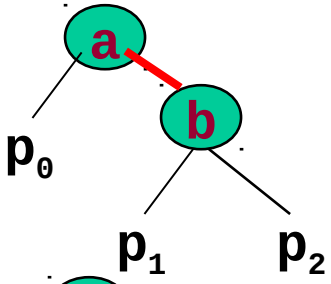
Une représentation des arbres 2.3.4 : les arbres bicolores

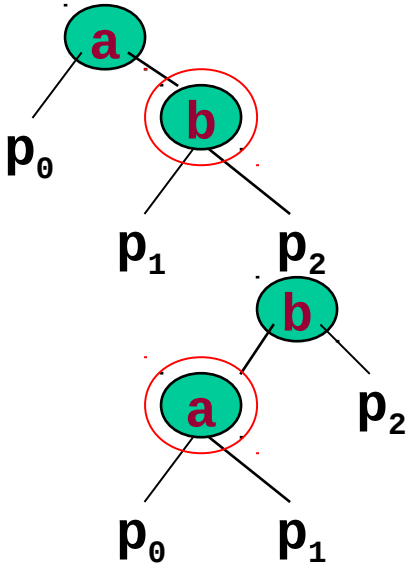
- Pour les 3-nœuds, il existe 2 transformations possibles

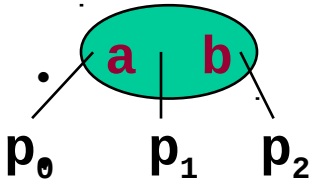


Penché à droite

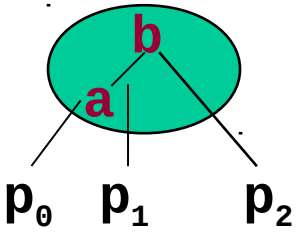


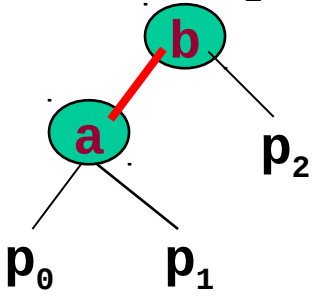






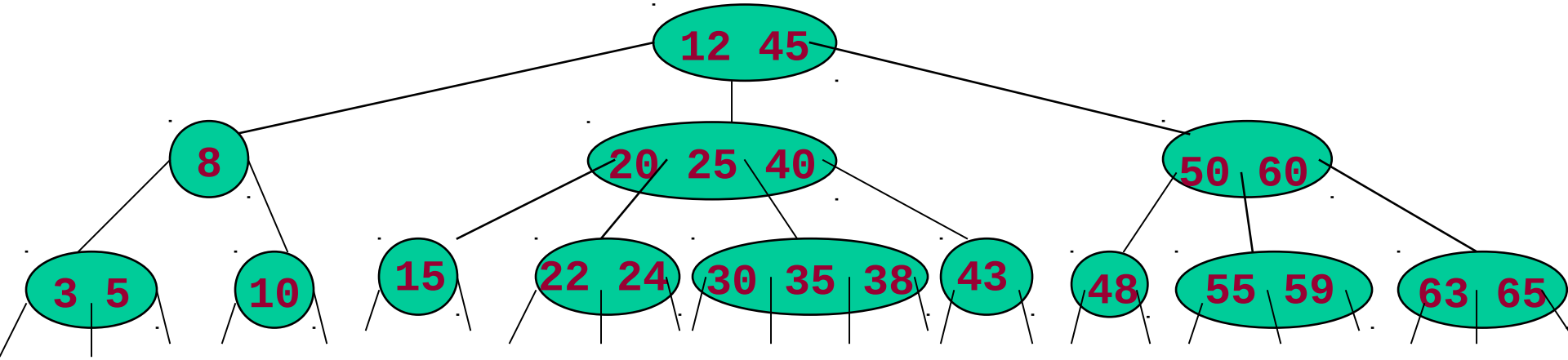
Penché à gauche



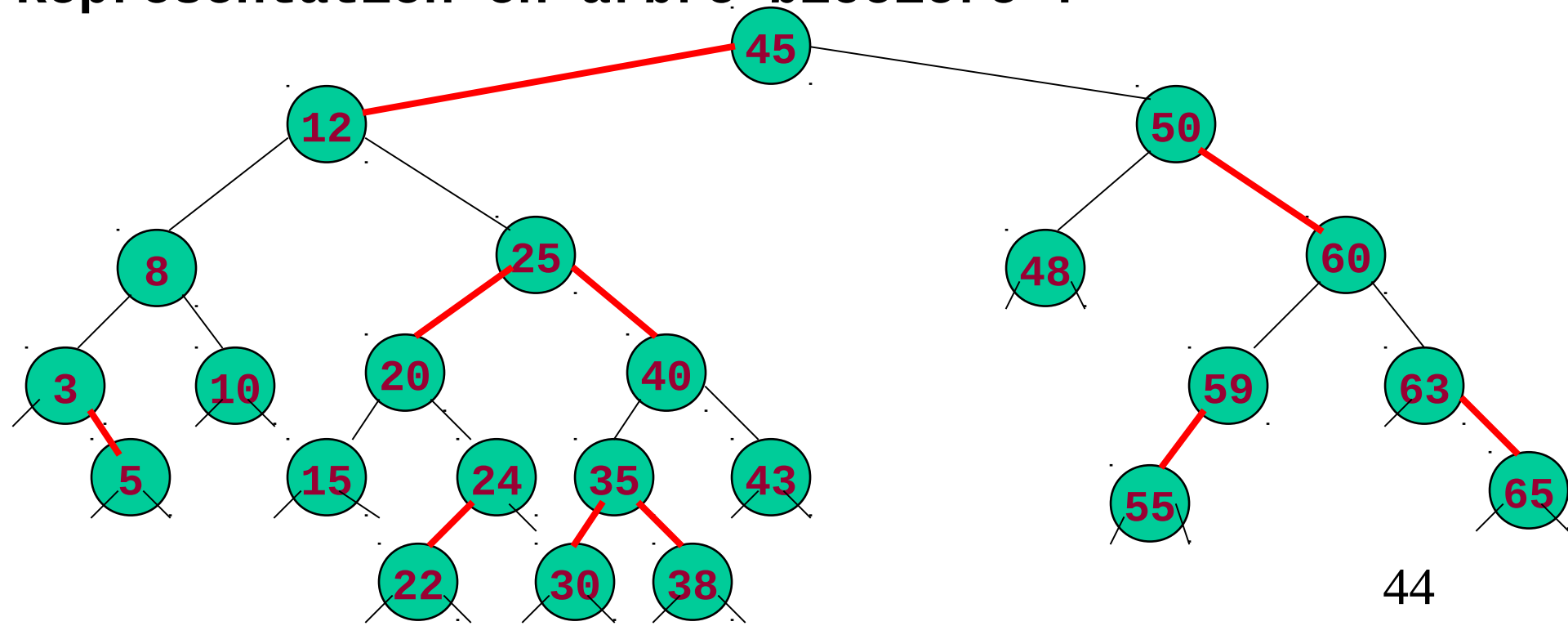


- Les deux représentations pourront exister à la suite de rotations
- La hauteur de l'arbre bicolore obtenu par ces transformations est au plus 2*la hauteur de l'arbre 2.3.4 initial, augmentée de 1.
- => Tout arbre bicolore associé à un arbre 2.3.4 contenant n éléments a une hauteur de l'ordre O(log n)

Arbres 2.3.4 /Arbres bicolores

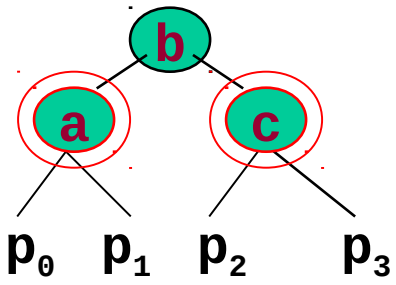


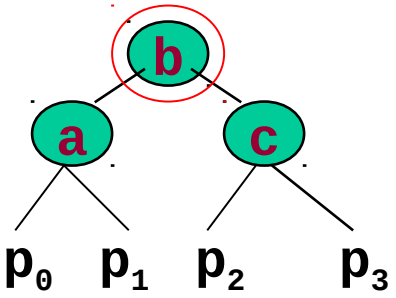
Représentation en arbre bicolore ?



Arbres 2.3.4 /Arbres bicolores

- On simule l ’adjonction avec éclatement à la descente dans un arbre 2.3.4
- Eclater un 4-nœud revient à inverser les couleurs des éléments de ce nœud





- Ceci peut cependant faire apparaître deux nœuds rouges consécutifs, ce qui doit être évité si l ’on veut conserver la propriété de hauteur logarithmique

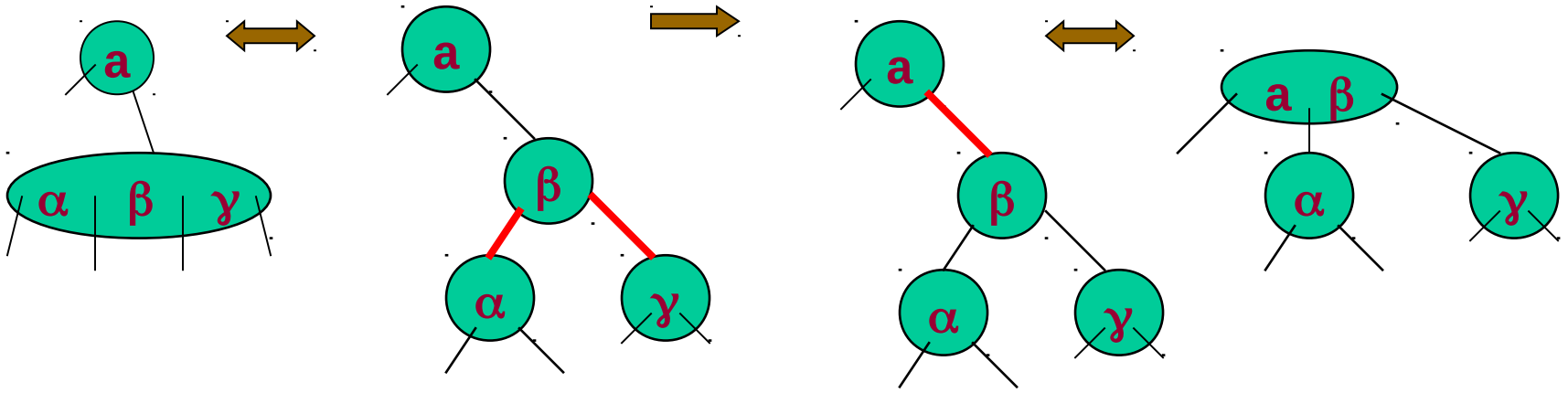
=> utilisation de transformations locales : rotations

Arbres 2.3.4 /Arbres bicolores

Plusieurs situations possibles

1) Le 4-nœud à éclater est attaché à un 2-nœud

=> une simple inversion de couleur suffit



Arbres 2.3.4 /Arbres bicolores

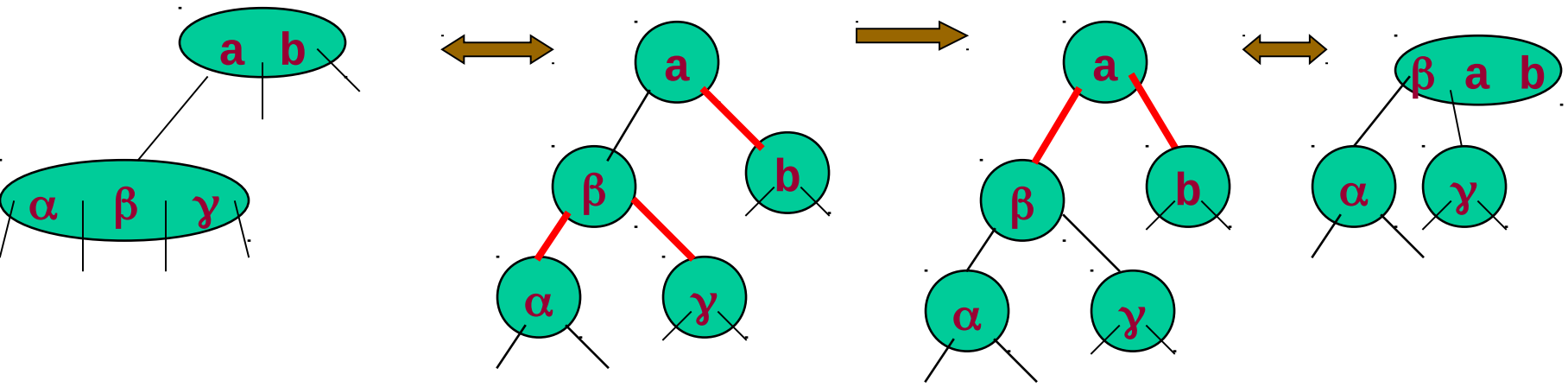
2) Le 4-nœud , E, à éclater est attaché à un 3-nœud :

3 cas lorsque le 3-nœud est penché à droite

3 cas lorsque le 3-nœud est penché à gauche

a) E est premier fils du 3-nœud

=> on inverse les couleurs des éléments du 4-noeud

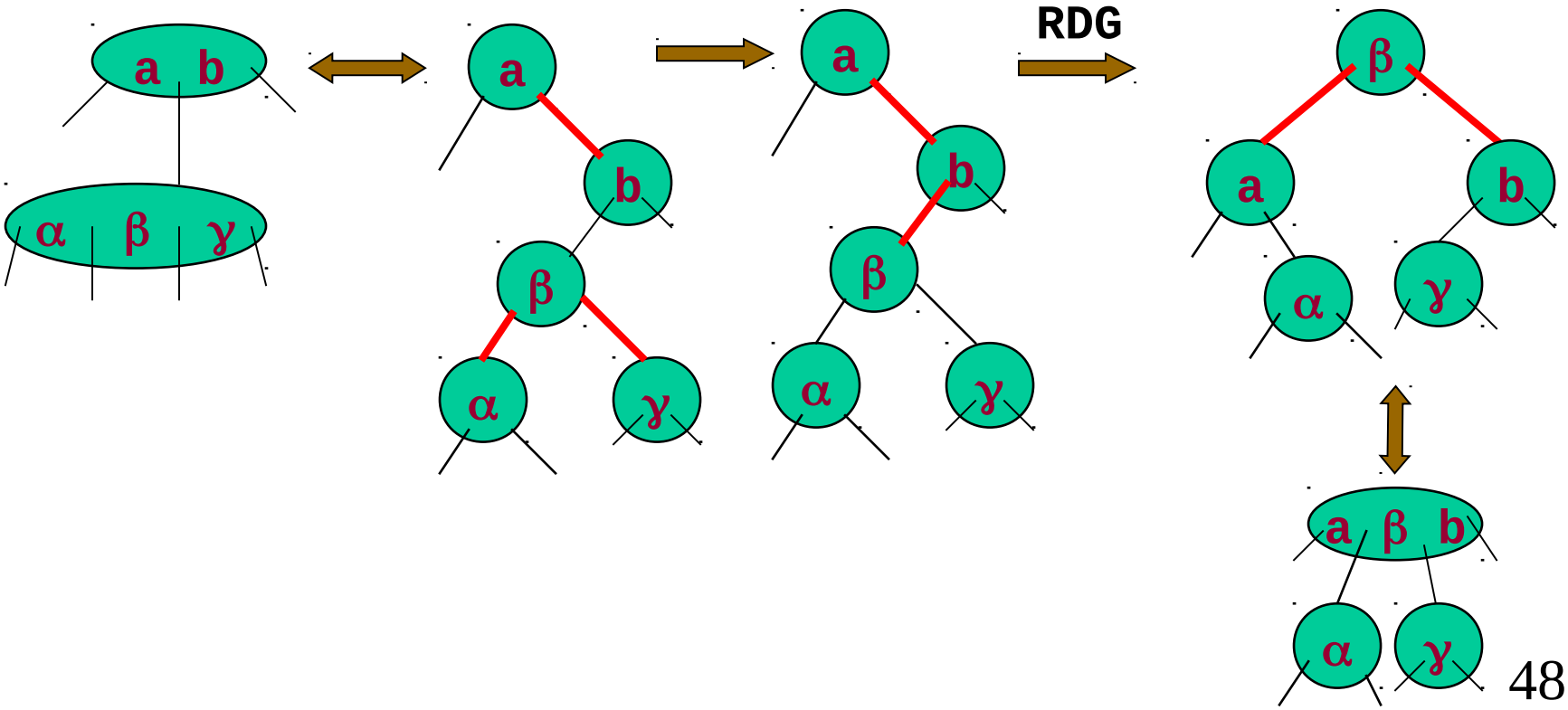


Arbres 2.3.4 /Arbres bicolores

b) E est second fils du 3-nœud

=> une inversion de couleurs entraîne une mauvaise disposition des éléments jumeaux a, β et b

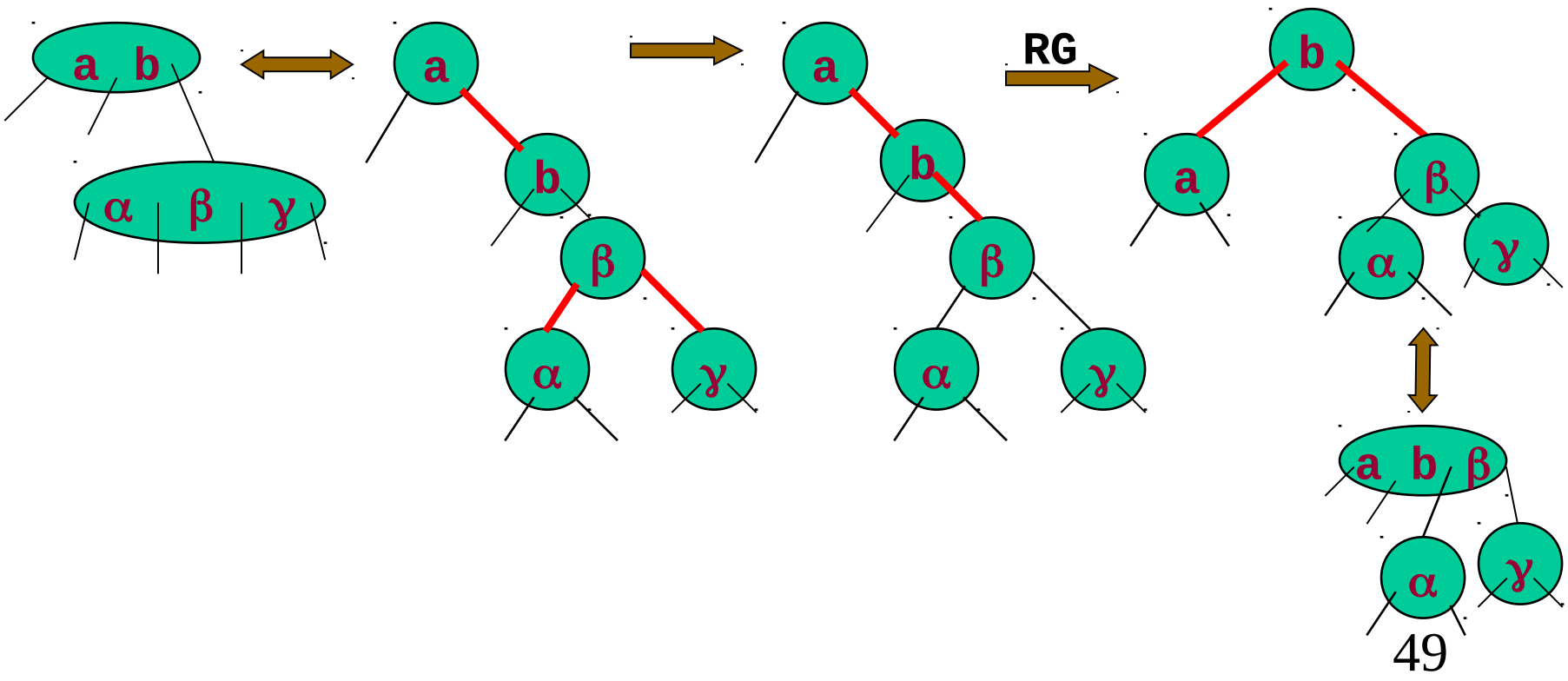
=> rotation droite-gauche au niveau du nœud contenant a



Arbres 2.3.4 /Arbres bicolores

c) E est troisième fils du 3-nœud

=> rotation gauche au niveau du nœud contenant a



Arbres 2.3.4 /Arbres bicolores

Algorithme d'adjonction :

- Descendre à partir de la racine à la recherche de la feuille où insérer l'élément
- Sur le chemin, lorsqu'un nœud A a ses deux fils rouges on inverse les couleurs de A et de ses fils : si de plus le père de A est lui aussi rouge, on fait une rotation au niveau du grand-père de A avant de poursuivre la descente.
- L'adjonction d 'un nouvel élément se fait toujours dans une feuille qui devient un nœud rouge, puisque le nouvel élément est ajouté en tant que jumeau; dans le cas où le père du nœud ajouté est rouge, mais n'a pas de frère rouge, il faut effectuer une rotation comme en b)

Arbres 2.3.4 /Arbres bicolores

Exercice :

Écrire l'algorithme d'adjonction en utilisant le type suivant :

```
typedef enum{blanc, rouge} couleur;
typedef struct bn{
    couleur coul;
    int val;
    struct bn * fg, *fd;
} Bnoeud;
typedef Bnoeud * Bicolore;
```

Recherche externe

- Grandes collections d ’éléments stockés sur mémoire secondaire paginée.
- Le nombre d ’accès à la mémoire secondaire est prépondérant

  => minimiser le nombre de transferts de pages
- B-Arbres :
  - généralisation des arbres 2.3.4

Recherche externe : B-arbres

Un **B-arbre** d ’ordre m est

- un arbre binaire de recherche formé de nœuds qui peuvent chacun contenir jusqu’à $2m$ éléments;
- chaque nœud est dans une page différente du support externe et
- la complexité des algos de recherche, adjonction et suppression d ’un élément parmi n, compté en nombre d ’accès à la mémoire secondaire, est en $O(\log_m n)$

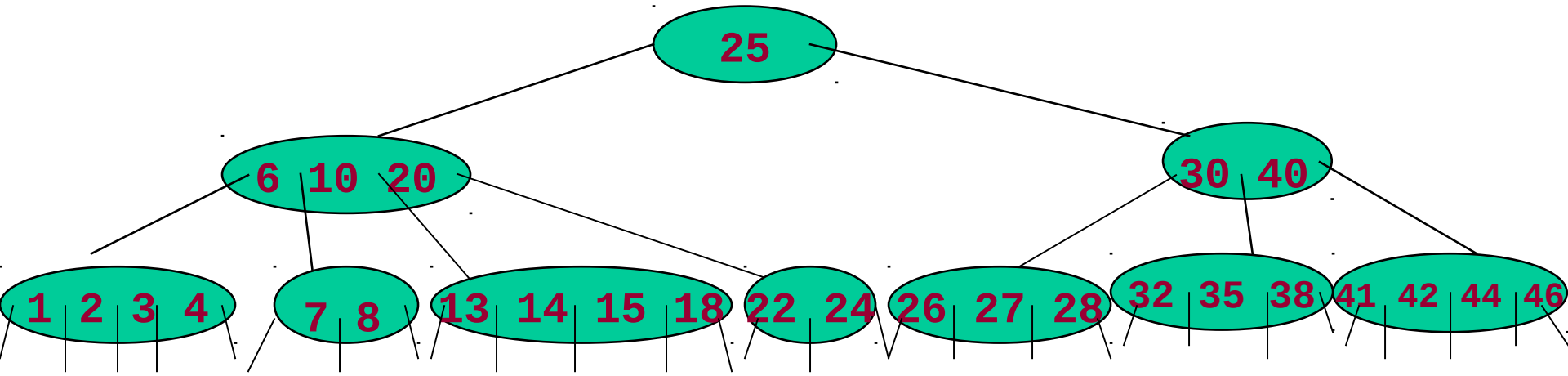
B-arbres

Un **B-arbre** d ’ordre m est un arbre binaire de recherche dont

- toutes les feuilles sont située au même niveau
- tous les nœuds, sauf la racine, sont des k-nœuds avec k $\in$ [m+1..2m+1]
- la racine est un k-nœud avec k $\in$[2..2m+1]

B-arbres

Exemple



Exemple de B-arbre d’ordre 2

B-arbres

Dans la réalité m est choisi de manière à ce qu'un nœud corresponde au contenu d'une page mémoire.

Exemple :

- **m=250** => un arbre de hauteur 2 peut contenir plus de 125 millions d 'éléments
  - 500 éléments à la racine
  - 500 *501 (environ $25*10^4$) éléments dans les nœuds de profondeur 1
  - 501^2 *500 (environ $125*10^6$) éléments dans les nœuds de profondeur 2

=> Dans les B-arbres, les niveaux les plus bas de l 'arbre contiennent la quasi-totalité des éléments

Algorithmes de recherche, d'adjonction et de suppression

=> généralisation des algorithmes pour les arbres 2.3.4