Plan

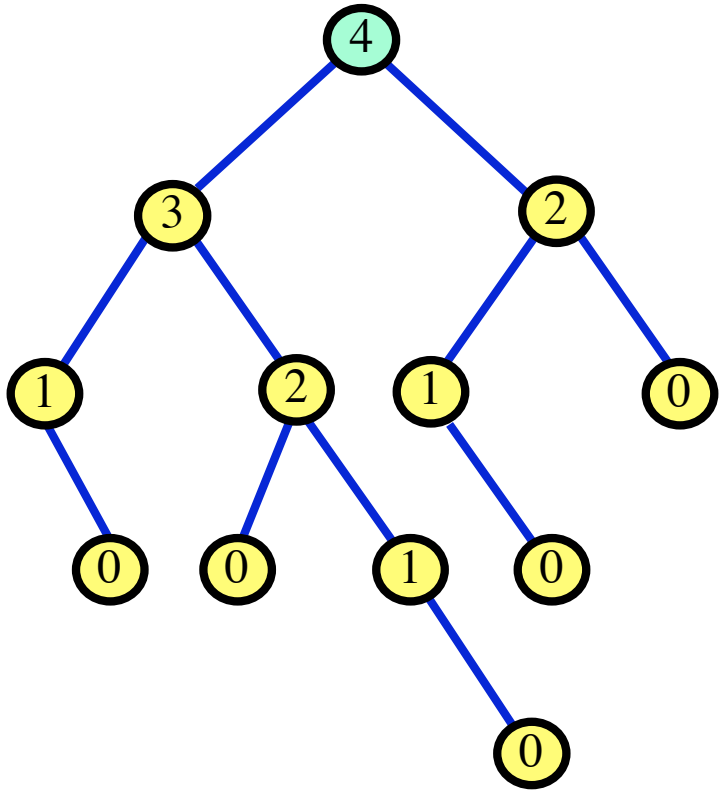
- Arbres équilibrés
  - Arbres AVL
  - Arbres a-b
- Quelques compléments de Java

Structures d'arbre

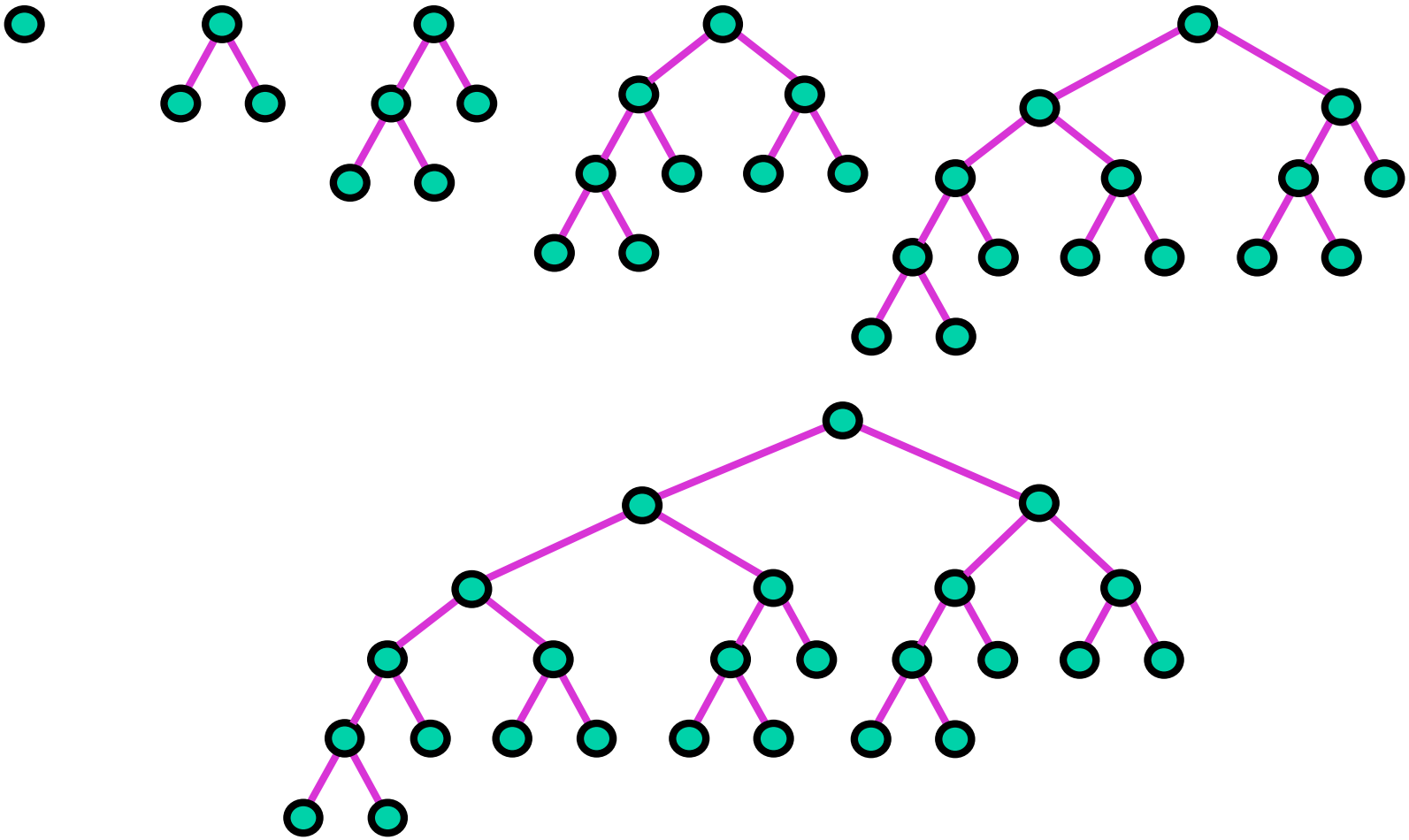
- Les structures d'arbre permettent de réaliser des opérations dynamiques, telles que recherche, prédécesseur, successeur, minimum, maximum, insertion, et suppression en temps proportionnel à la hauteur de l'arbre.
- Si l'arbre est équilibré, la hauteur est en O(log n) où n est le nombre de nœuds.
- Divers types d'arbres équilibrés sont utilisés : arbres AVL, 2-3, bicolores, b-arbres.

Arbres AVL : définition

Un arbre binaire est un arbre AVL (Adelson-Velskii et Landis) si, pour tout sommet, les hauteurs des sous-arbres gauche et droit diffèrent d'au plus 1.



Exemple : arbres de Fibonacci



Hauteur d'un arbre AVL

Propriété. Soit A un arbre AVL ayant n sommets et de hauteur h. Alors

$$\log_2(1+n) \leq 1+ h \leq \square \log_2(2 + n)$$

avec $\square \leq 1{,}44$.

Par exemple, si $n = 100000$, $17 \leq h \leq 25$.

On a toujours

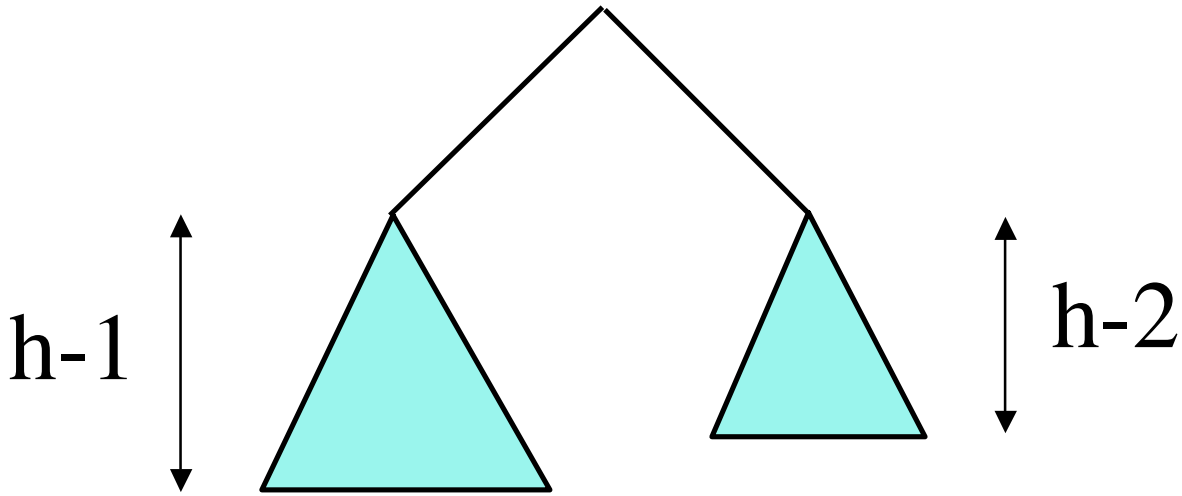
$$n \leq 2^{h+1} - 1$$

donc $\log_2(1+n) \leq 1+ h$.

Hauteur d'un arbre AVL (2)

Soit $N(h)$ le nombre *minimum* de sommets d'un arbre AVL de hauteur h. Alors

$$N(h) = 1 + N(h\text{-}1) + N(h\text{-}2).$$



Hauteur d'un arbre AVL (2)

La suite $F(h) = N(h)+1$ vérifie

$$F(0) = 2,\ F(1) = 3,\ F(h+2) = F(h+1) + F(h)$$

donc

$$F(h) = 1/\sqrt{5}\ (\varphi^{h+3} - \hat{\varphi}^{(h+3)})$$

où $\varphi = \frac{1+\sqrt{5}}{2}$ est le nombre d'or

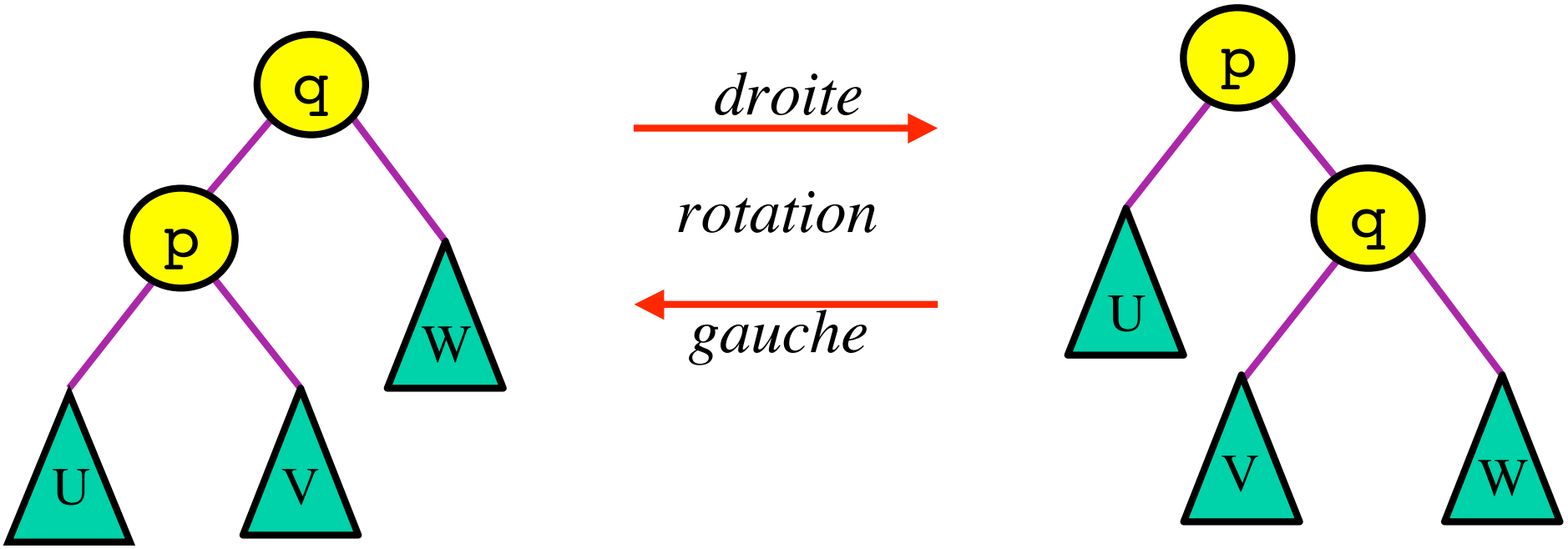
$$1 + n \geq F(h) > 1/\sqrt{5}\ (\varphi^{h+3} - 1)$$

$$h+3 < \log_2(2+n) / \log_2 \varphi + \log_2 \sqrt{5} \leq \lceil \log_2(2+n) \rceil + 2$$

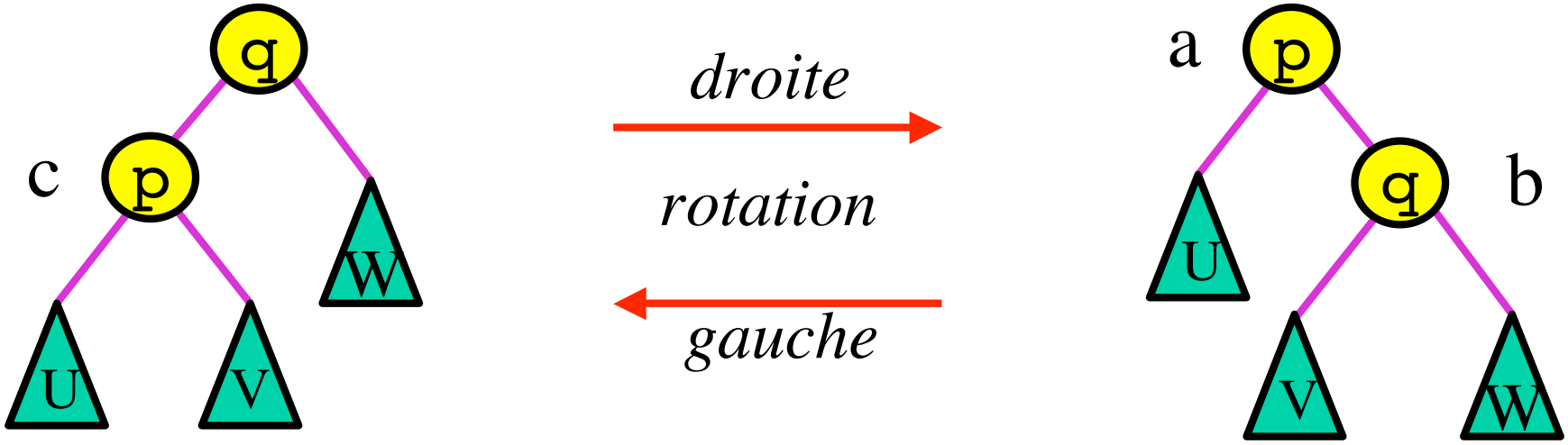
Rotations et équilibrage

Les rotations gauche et droite transforment un arbre

- Elles préservent l'ordre infixe
- Elles se réalisent en temps constant.

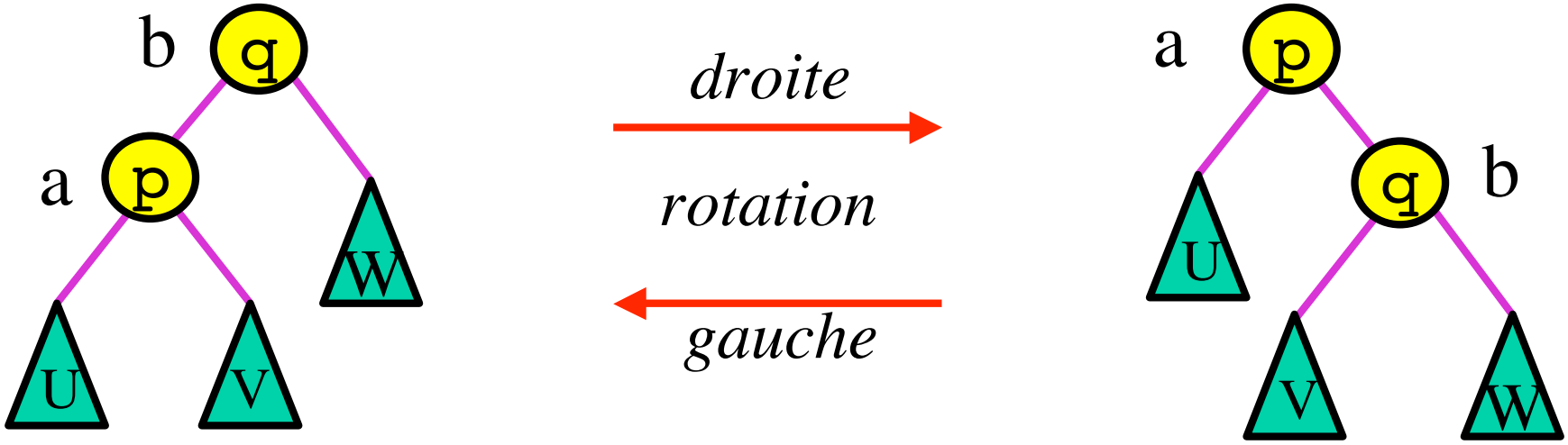


Implantation (non destructive)



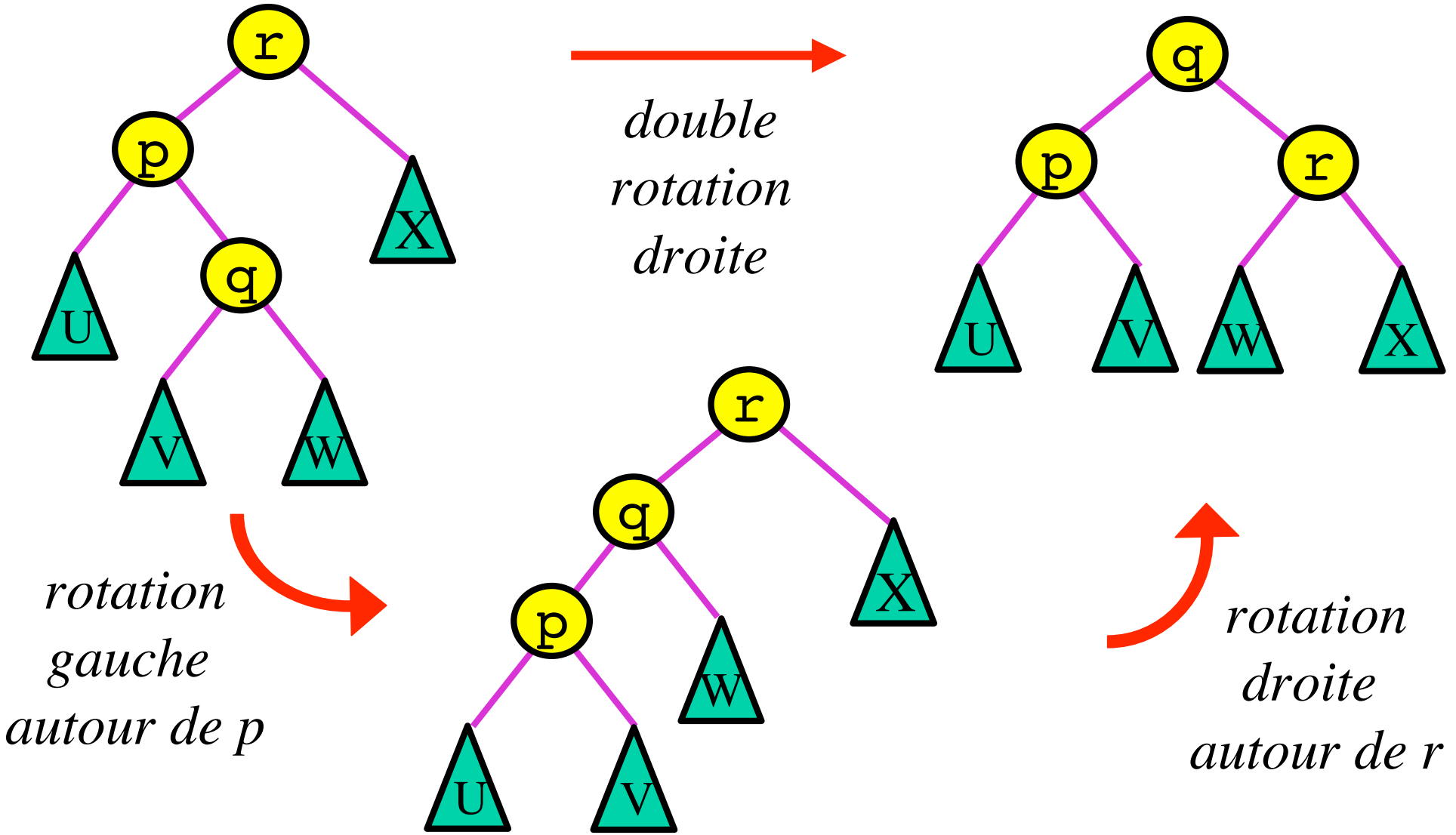

```
static Arbre rotationG(Arbre a)
{
  Arbre b = a.filsD;
  Arbre c = new Arbre (a.filsG,
    a.contenu, b.filsG);
  return new Arbre (c, b.contenu,
    b.filsD);
}
```

Implantation (destructive)



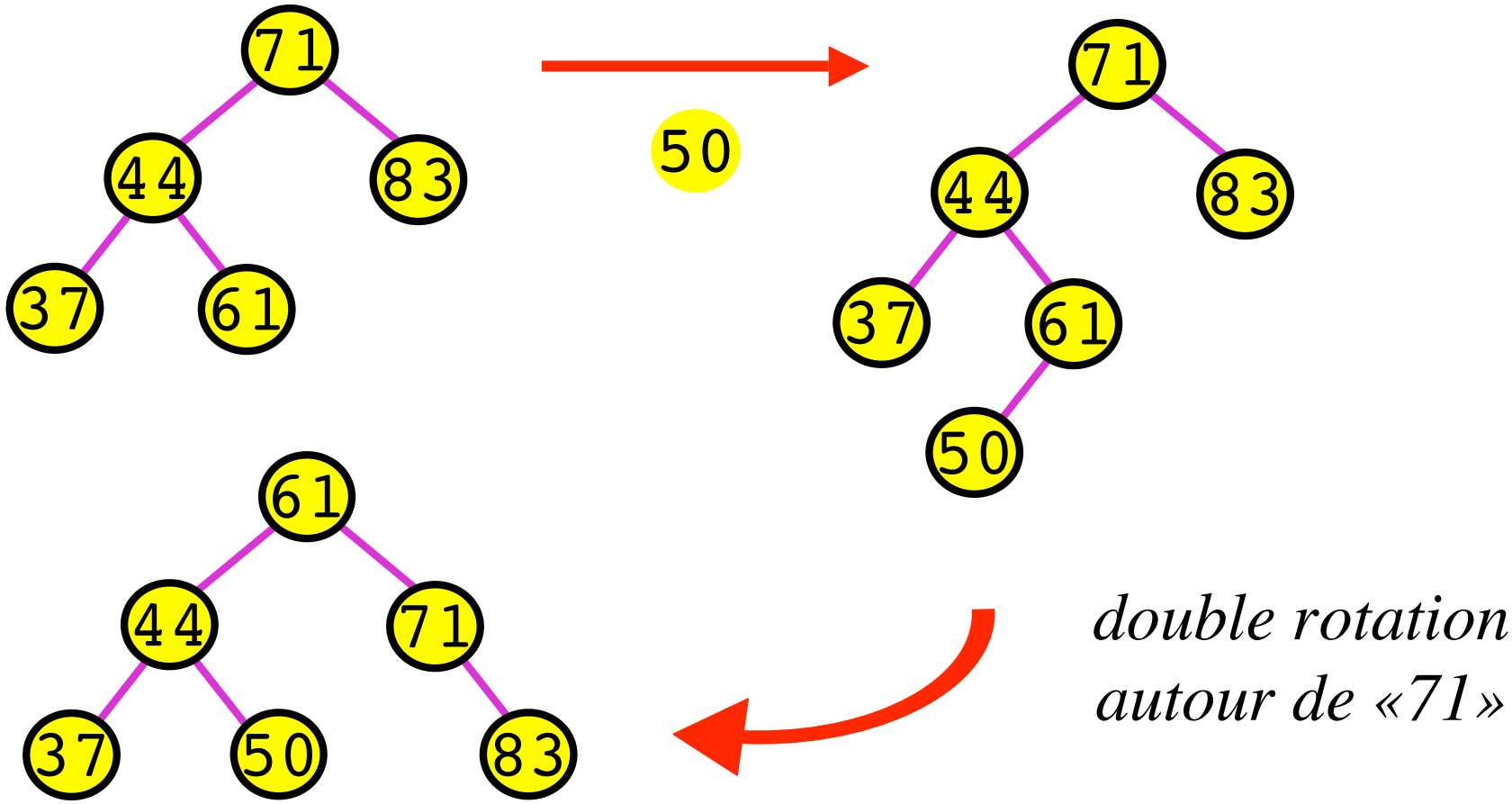

```
static Arbre rotationG(Arbre a)
{
  Arbre b = a.filsD;
  a.filsD = b.filsG;
  b.filsG = a;
  return b;
}
```

Double rotation



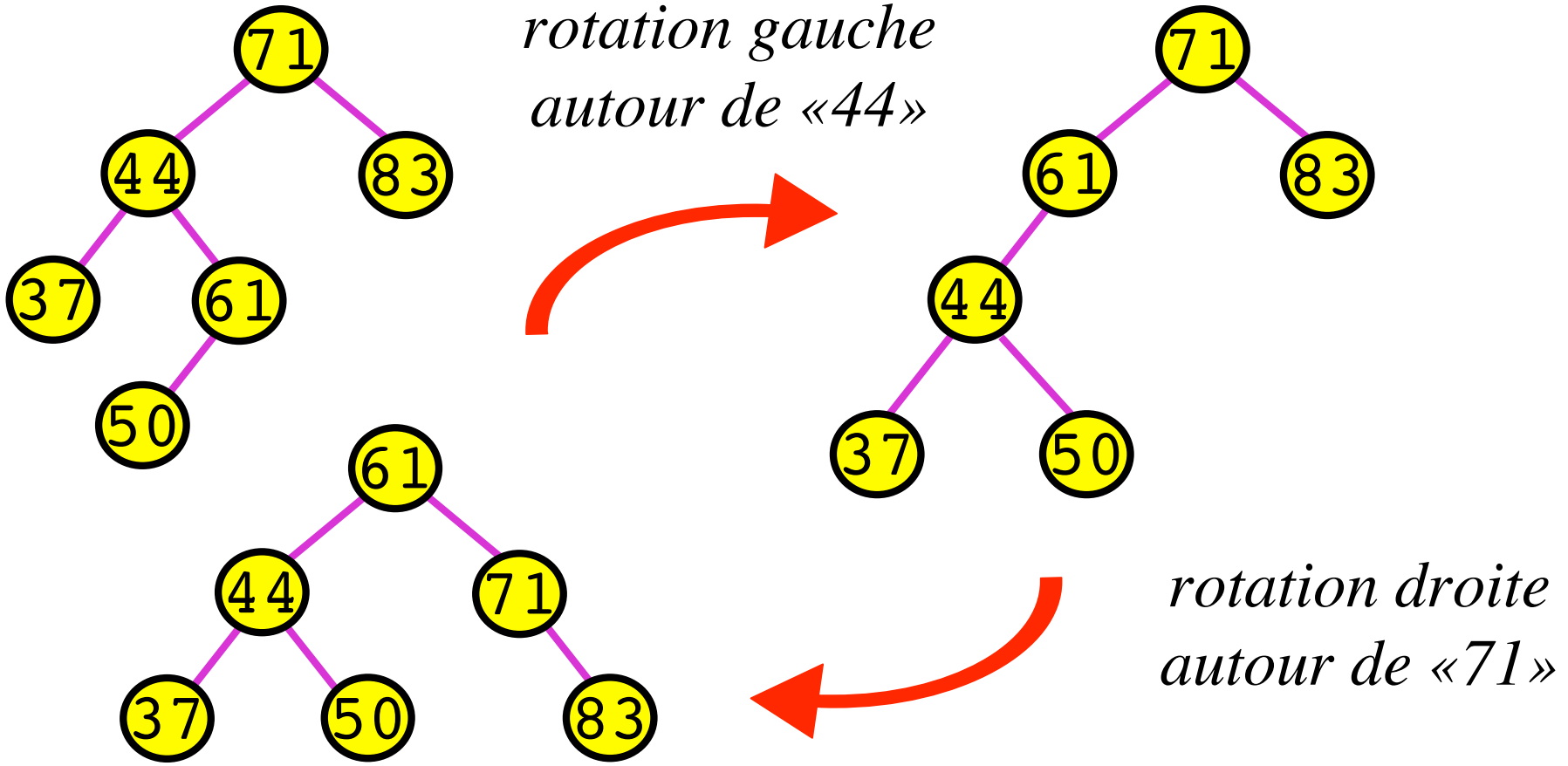
Insertion dans un arbre AVL

- Insertion ordinaire, suivie de rééquilibrage.



Insertion dans un arbre AVL

- Insertion ordinaire, suivie de rééquilibrage.



Propriétés

- Pour rééquilibrer un arbre AVL après une insertion, une seule rotation ou double rotation suffit.

- Pour rééquilibrer un arbre AVL après une suppression, il faut jusqu'à h rotations ou double rotations (h est la hauteur de l'arbre).

Algorithme. Soit A un arbre, G et D ses sous-arbres gauche et droit. On suppose que

$$\left| h(G) - h(D) \right| = 2$$

- Si $h(G) - h(D) = 2$, on fait une rotation droite.

Soient g et d les sous-arbres gauche et droit de G. Si $h(g) < h(d)$, on fait d'abord une rotation gauche de G.

- Si $h(G) - h(D) = -2$, opérations symétriques.

Implantation : la classe AVL

```
class Avl
{
  char contenu;
  int hauteur;
  Avl filsG, filsD;

  Avl(Avl g, char c, Avl d) {
    filsG = g;
    contenu = c;
    filsD = d;
    hauteur = 1 + Math.max(H(g), H(d));
  }
  ...
}
```

Implantation : calculs sur la hauteur

Toutes les méthodes sont dans la classe AVL.

```
static int H(Avl a)
{
  return (a == null) ? -1 : a.hauteur;
}

static void Avl calculerHauteur(Avl a)
{
  a.hauteur = 1 +
    Math.max(H(a.filsG), H(a.filsD));
}
```

Implantation des rotations

```
static Avl rotationG(Avl a)
{
  Avl b = a.filsD;
  Avl c =
    new Avl(a.filsG, a.contenu, b.filsG);
  return
    new Avl(c, b.contenu, b.filsD);
}
```

Implantation du rééquilibrage

```
static Avl equilibrer(Avl a)
{
  calculerHauteur(a);
  if (H(a.filsG) - H(a.filsD) == 2)
  {
    if (H(a.filsG.filsG) <
        H(a.filsG.filsD))
      a.filsG = rotationG(a.filsG);
    return rotationD(a);
  } // else
  if (H(a.filsG)- H(a.filsD) == -2){...}
  return a;
}
```

Implantation de l'insertion

```
static Avl inserer(int x, Avl a)
{
  if (a == null)
    return new Avl(null, x, null);
  if (x < a.contenu)
    a.filsG = inserer(x, a.filsG);
  else if (x > a.contenu)
    a.filsD = inserer(x, a.filsD);
  // seul changement :
  return equilibrer(a);
}
```

Pourquoi des b-arbres ?

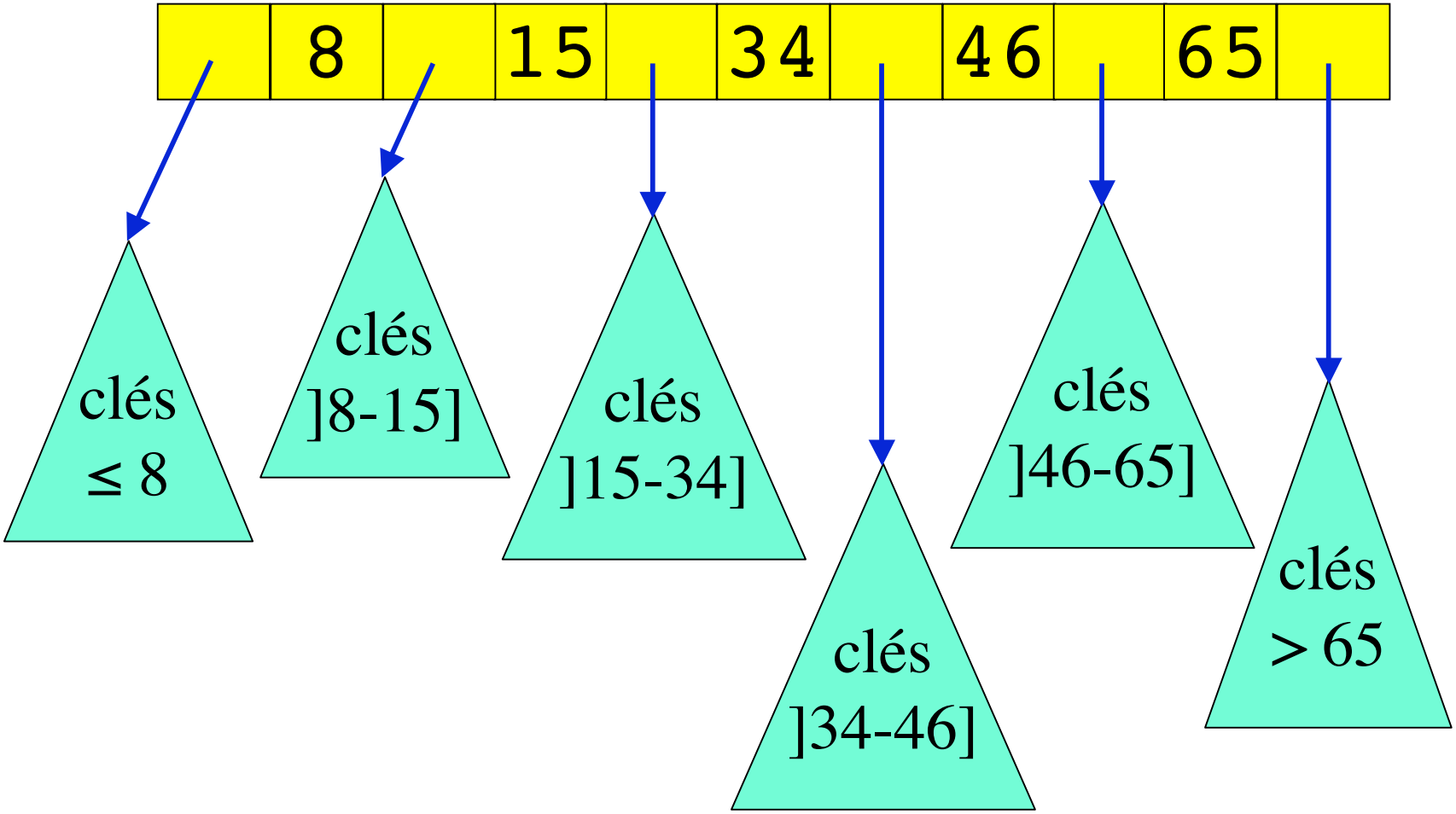
- On doit souvent manipuler de grands volumes de données.
- Si les données ne tiennent pas en mémoire vive, les problèmes d'accès au disque deviennent primordiaux.
- Un seul accès au disque prend environ autant de temps que 200 000 instructions.
- L'utilisation d'un b-arbre minimise le nombre d'accès au disque.

Information sur disque

- Un disque est divisé en pages (par exemple de taille 512, 2048, 4096 ou 8192 octets).
- La page est l'unité de transfert entre mémoire et disque.
- Les informations sont constituées d'enregistrements, accessibles par une clé.
- Une page contient plusieurs enregistrements.
- Problème : trouver la page contenant l'enregistrement de clé c.

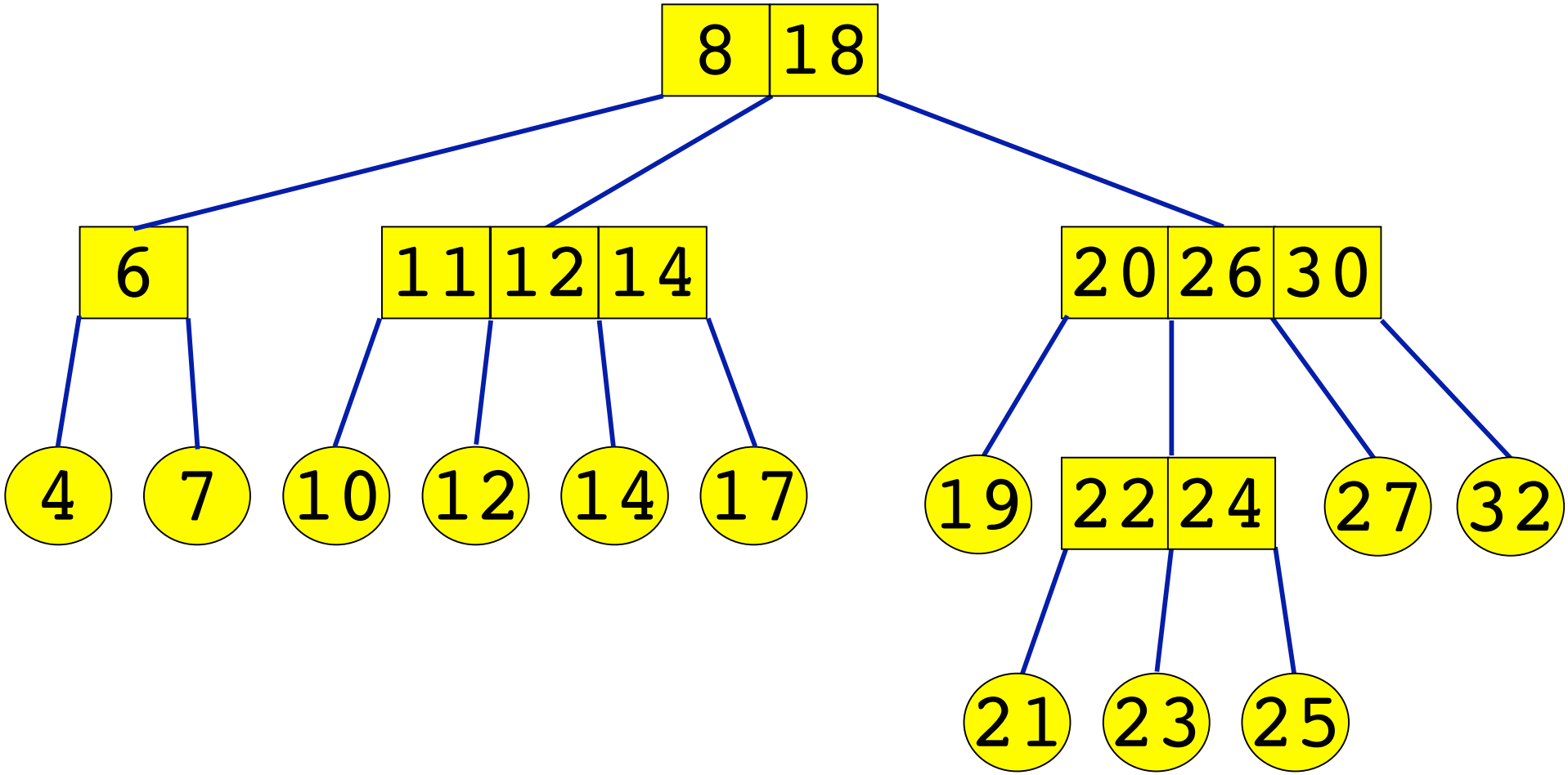
Arbres de recherche généralisés

Chaque noeud interne contient des balises



Arbres de recherche généralisés

Même principe que les arbres binaires de recherche

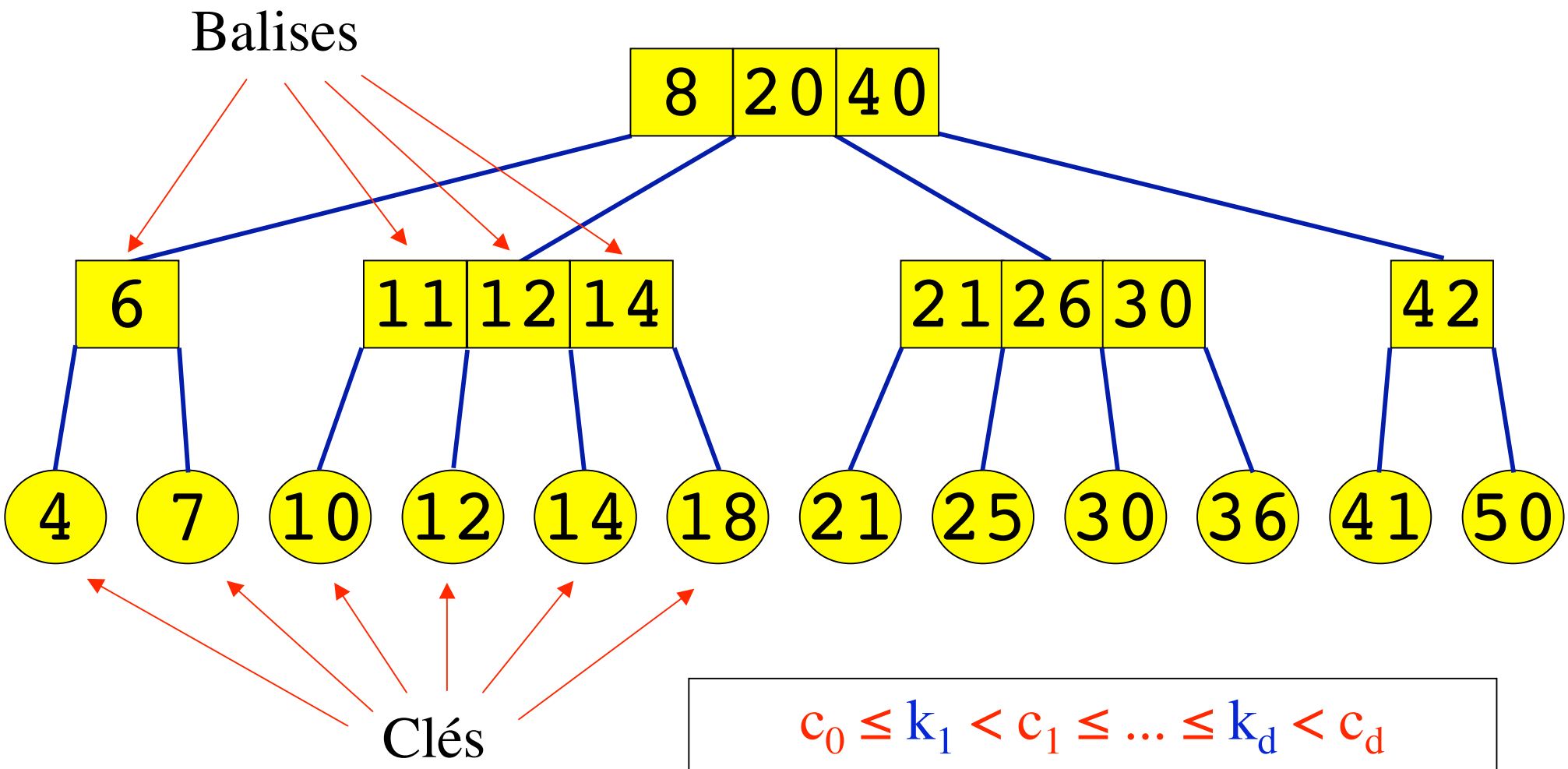


Définition d'un arbre a-b ($2a - 1 \leq b$)

- C'est un arbre de recherche généralisé.
- Les feuilles ont toutes la même profondeur.
- La racine a au moins 2 et au plus b fils (sauf si l'arbre est réduit à sa racine).
- Les autres nœuds internes ont au moins a et au plus b fils.

- Lorsque $b = 2a - 1$, un arbre a-b est appelé un b-arbre d'ordre a-1.
- En pratique, a peut être de l'ordre de plusieurs centaines.

Un arbre 2-4



$$c_0 \leq k_1 < c_1 \leq \ldots \leq k_d < c_d$$
$$10 \leq 11 < 12 \leq 12 < 14 \leq 14 < 18$$

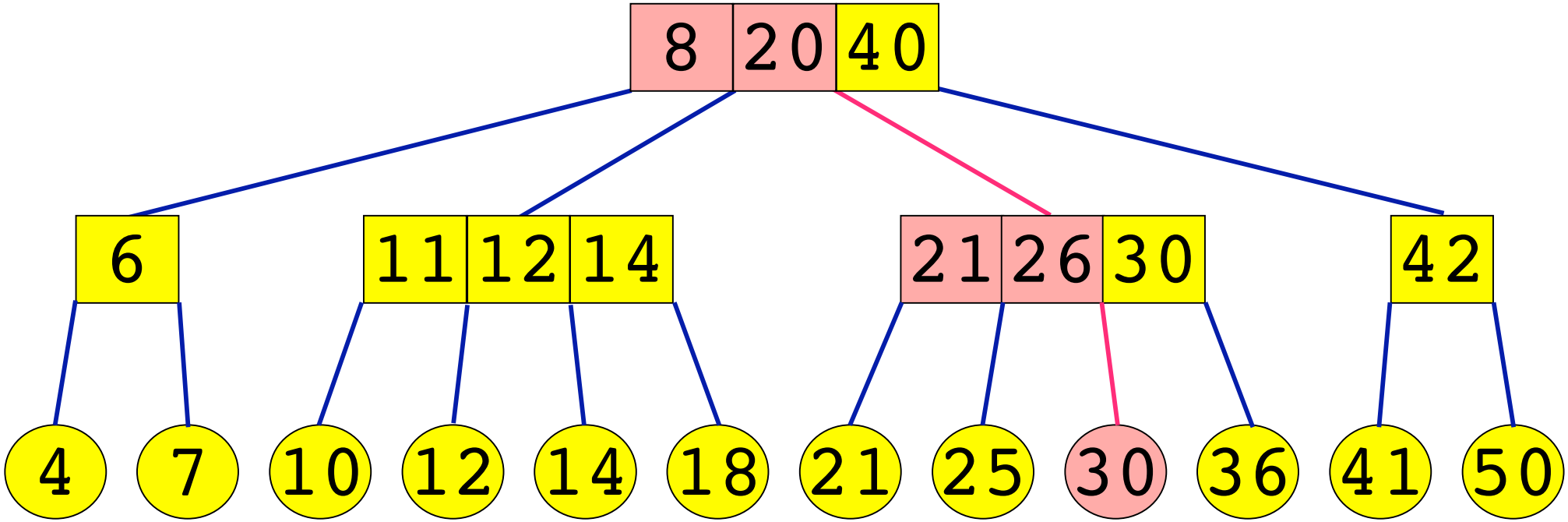
Taille d'un arbre a-b (2a - 1 ≤ b)

Proposition. Si un arbre a-b de hauteur h contient n feuilles, alors $2a^{h-1} \leq n \leq b^h$ et donc

$$\log n / \log b \leq h \leq 1 + \log(n/2) / \log a$$

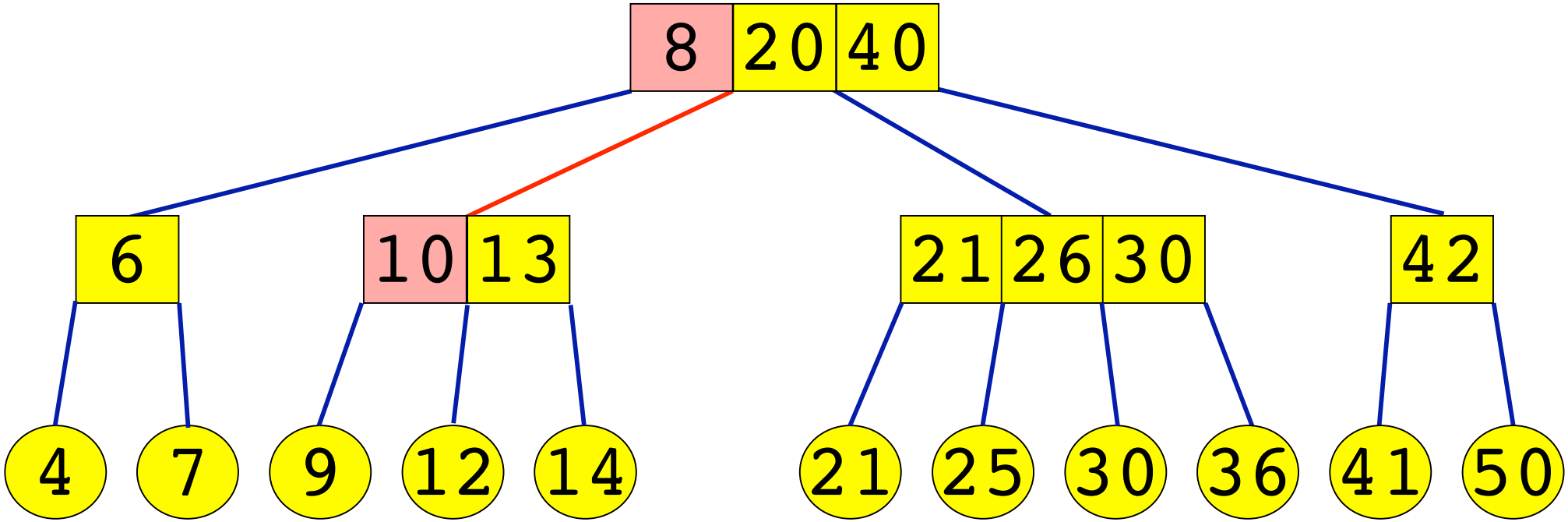
Preuve. Tout nœud a au plus b fils. Il y a donc au plus b^h feuilles. Tout nœud autre que la racine a au moins a fils, et la racine au moins 2. Au total, il y a au moins $2a^{h-1}$ feuilles.

Recherche dans un arbre a-b

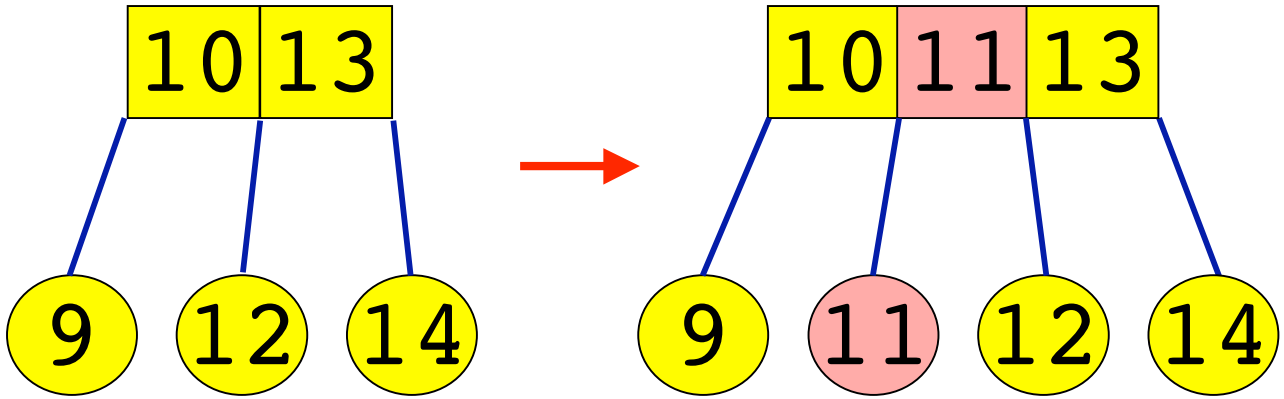


Recherche de 30

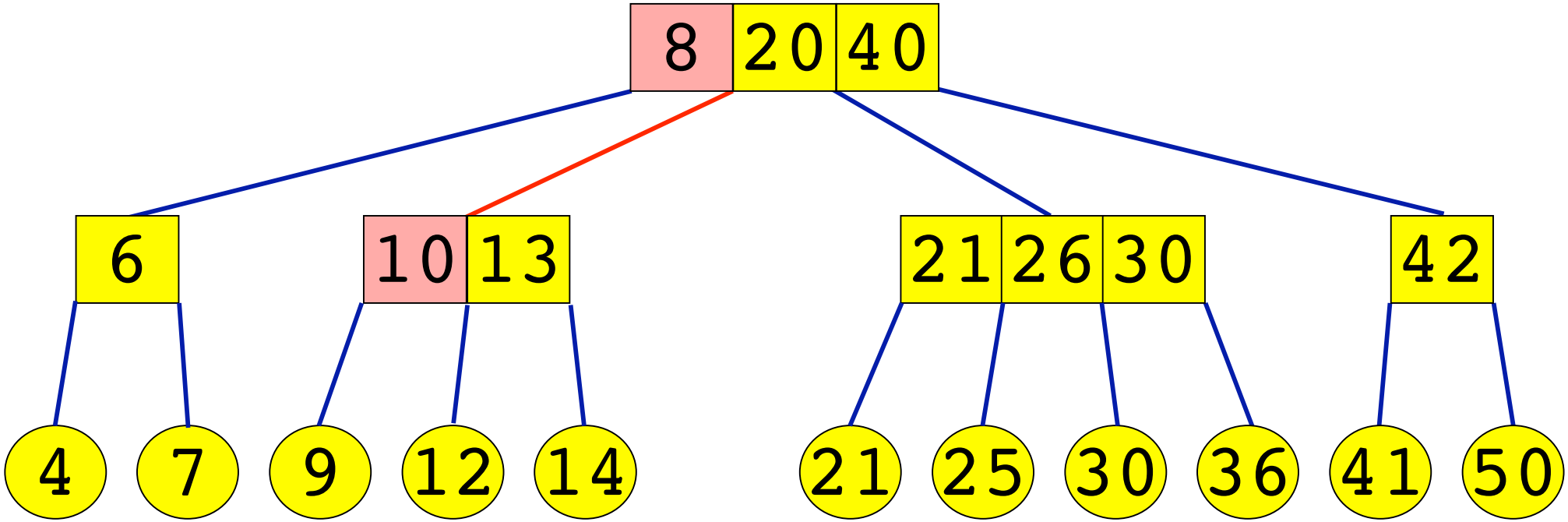
Insertion dans un arbre a-b (1)



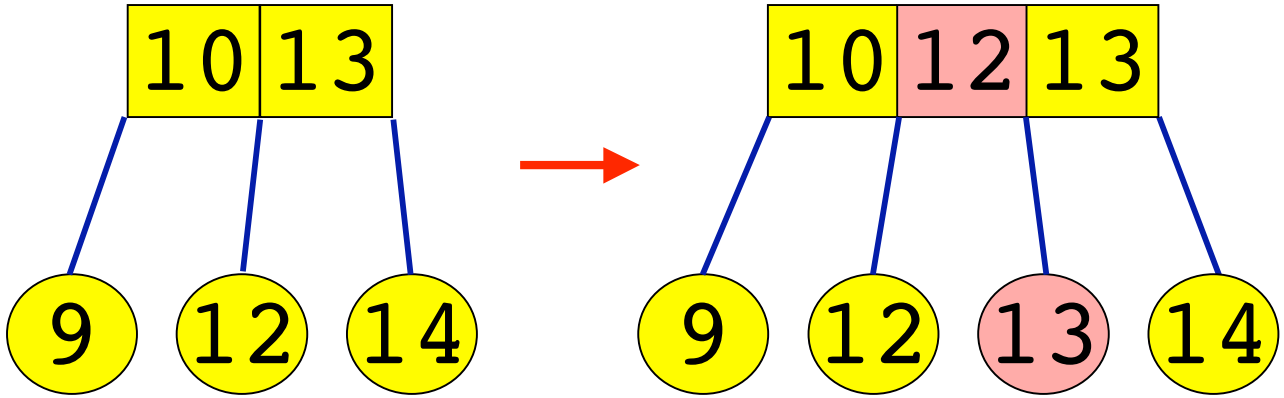
Insertion de 11:
la balise est
min(11, 12)



Insertion dans un arbre a-b (2)

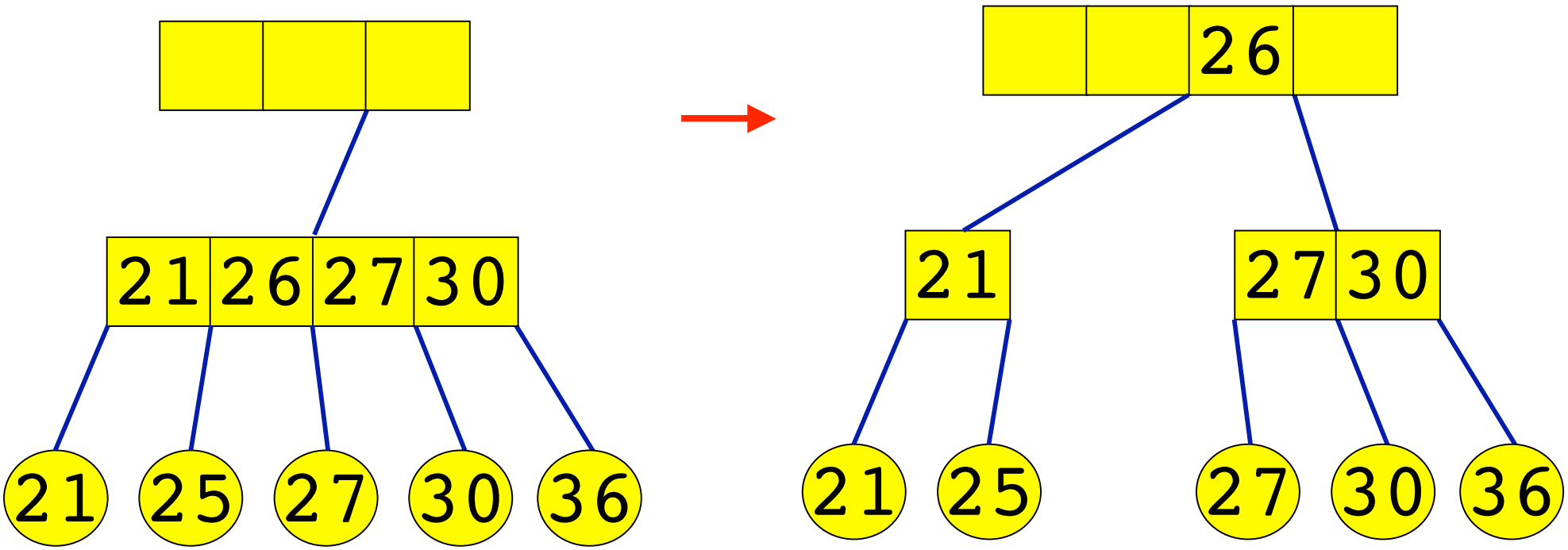


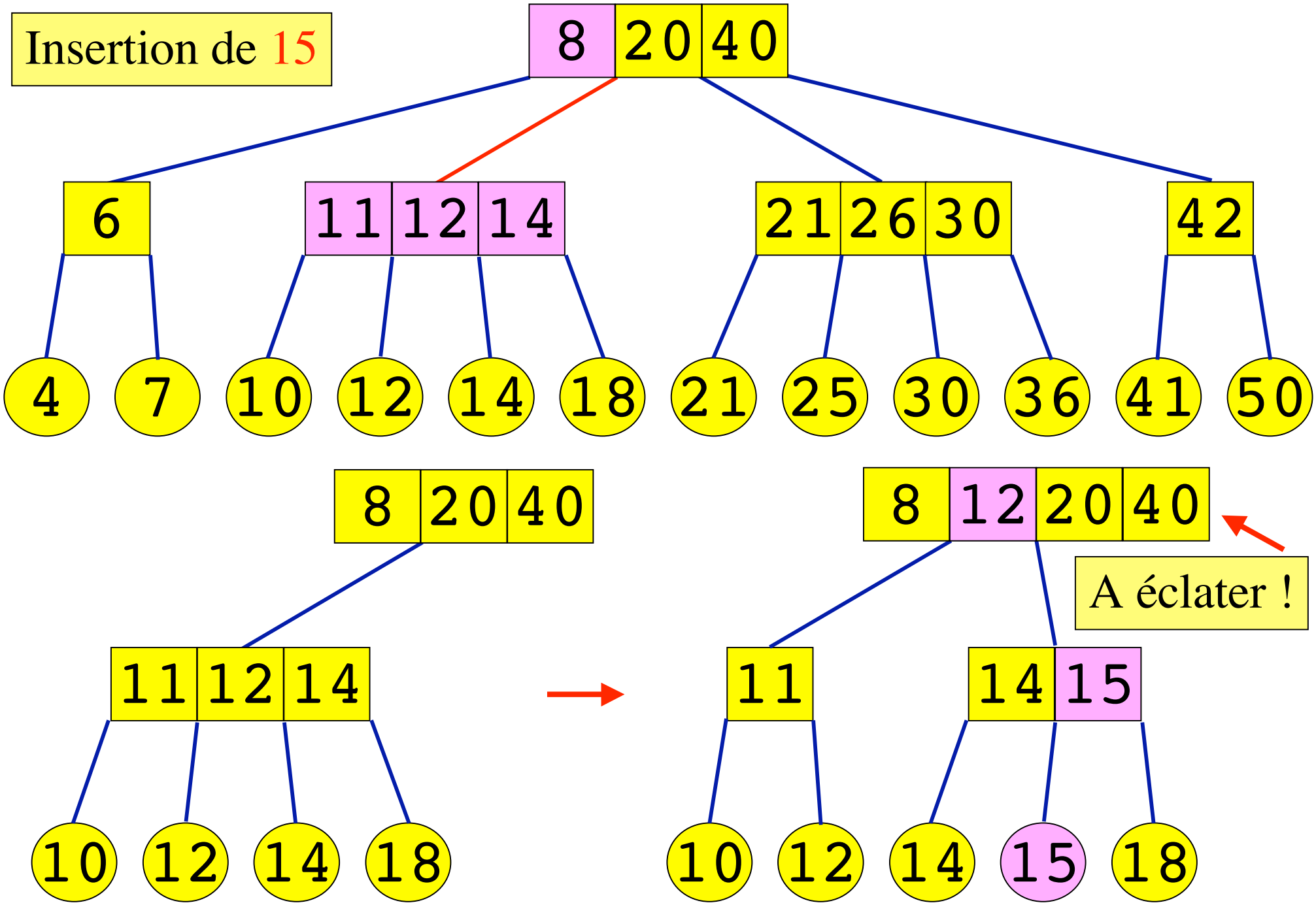
Insertion de 13 : la balise est min(12, 13)



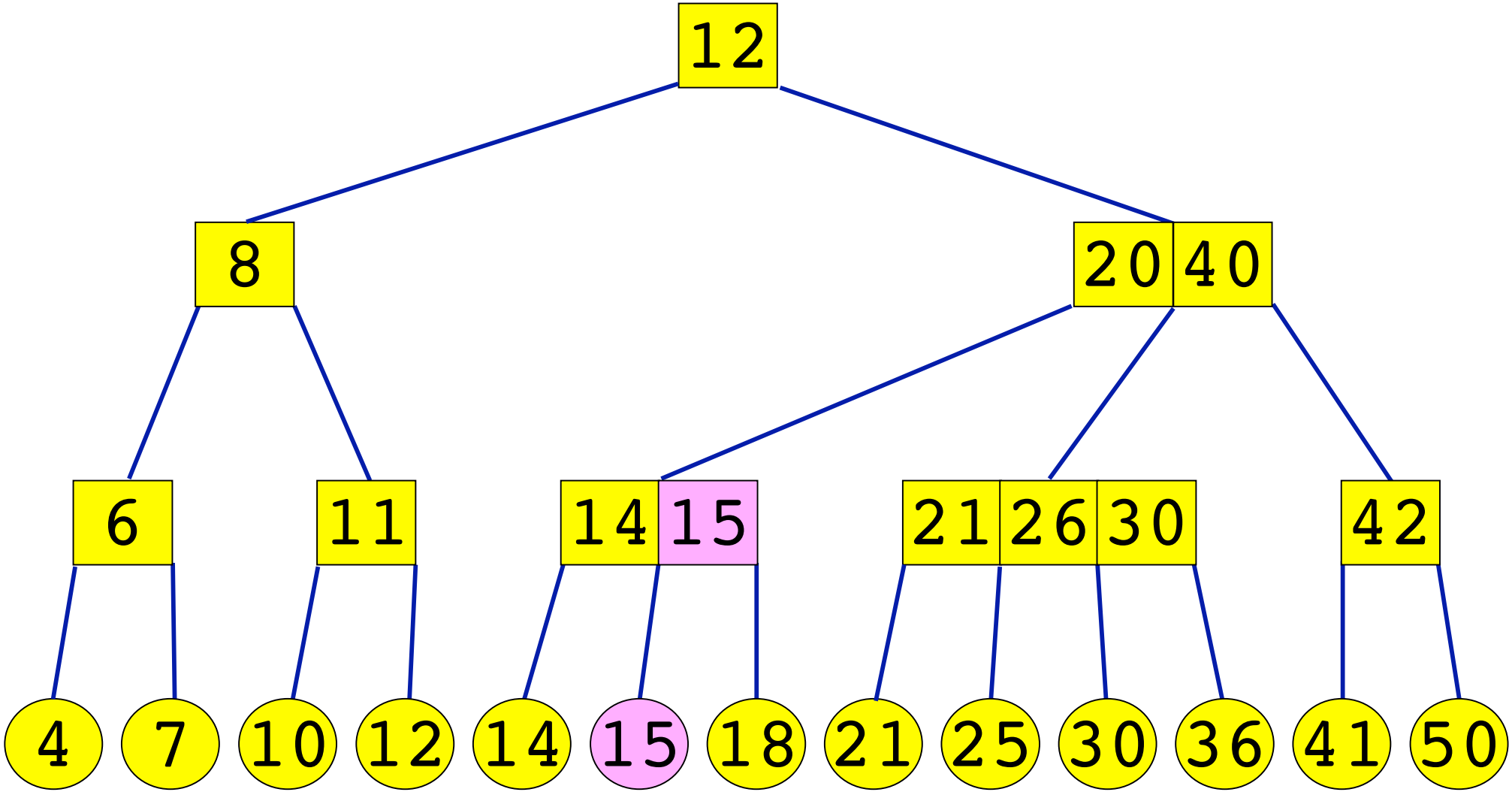
Eclatement d'un nœud

L'insertion d'un élément peut produire un nœud ayant plus de b fils. Il faut alors éclater le nœud.



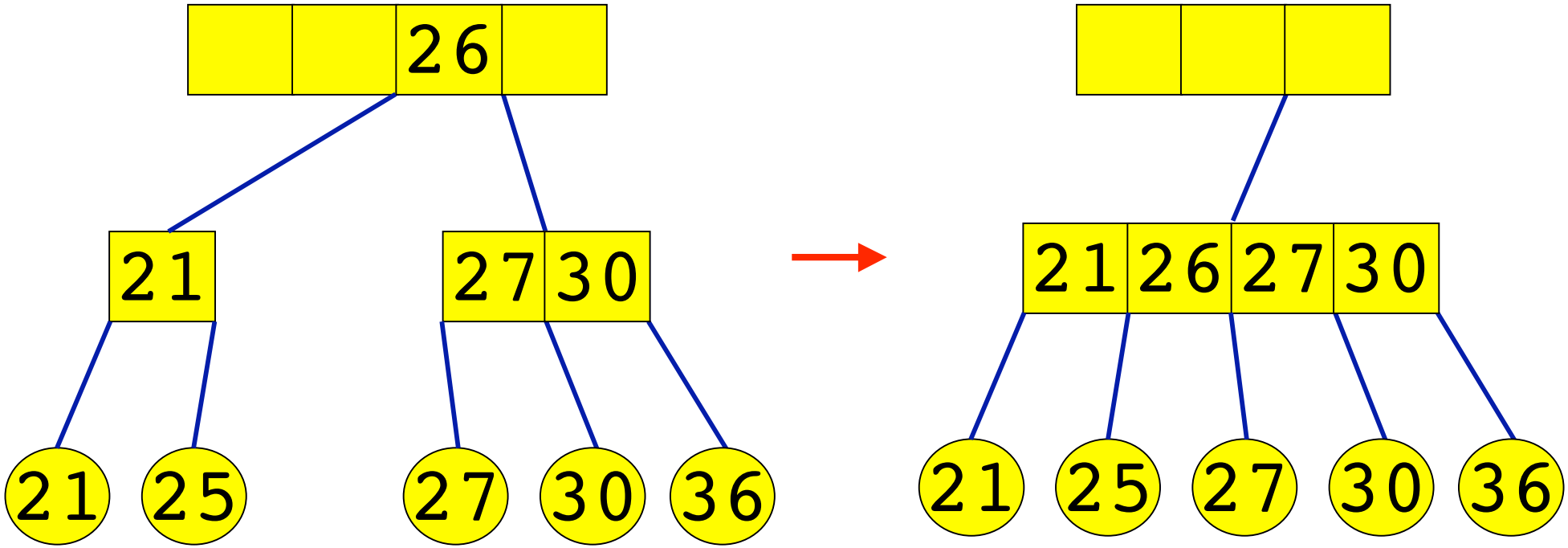

Insertion de 15
8 20 40
6
11 12 14
21 26 30
42
4 7 10 12 14 18 21 25 30 36 41 50
8 20 40
11 12 14
10 12 14 18
8 12 20 40
A éclater !
11
14 15
10 12 14 15 18

Après insertion de 15

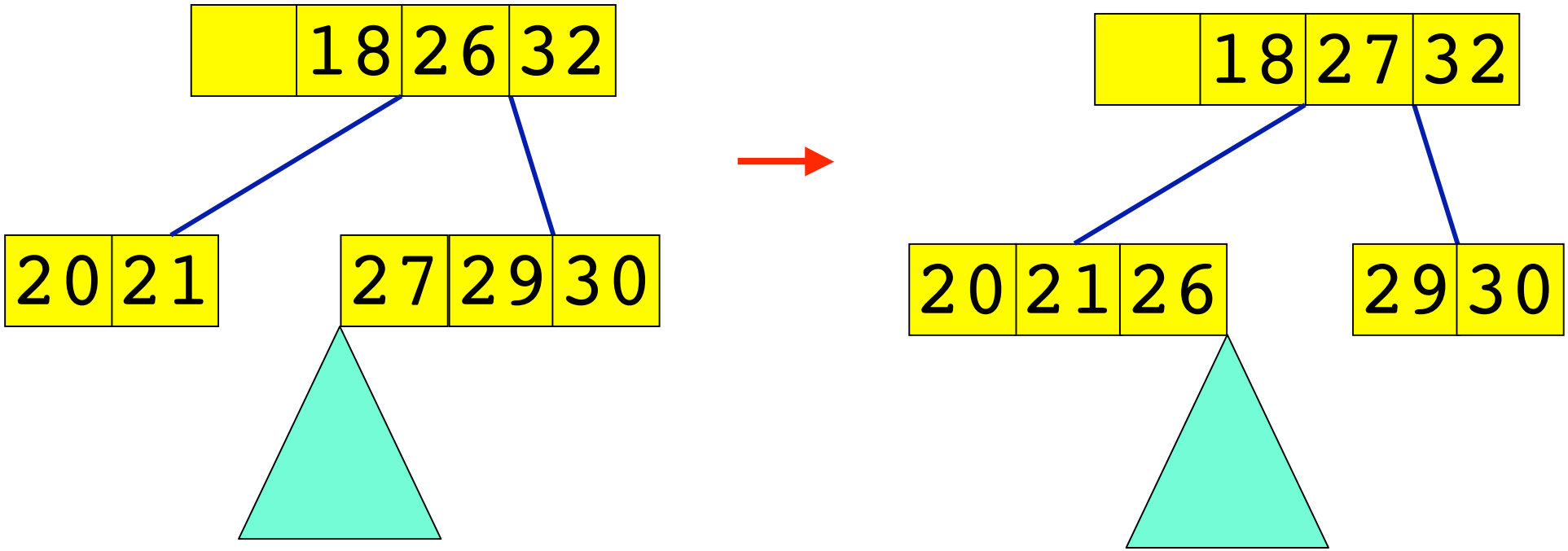


Fusion de deux nœuds

C'est l'opération inverse de l'éclatement.



Partage de deux nœuds



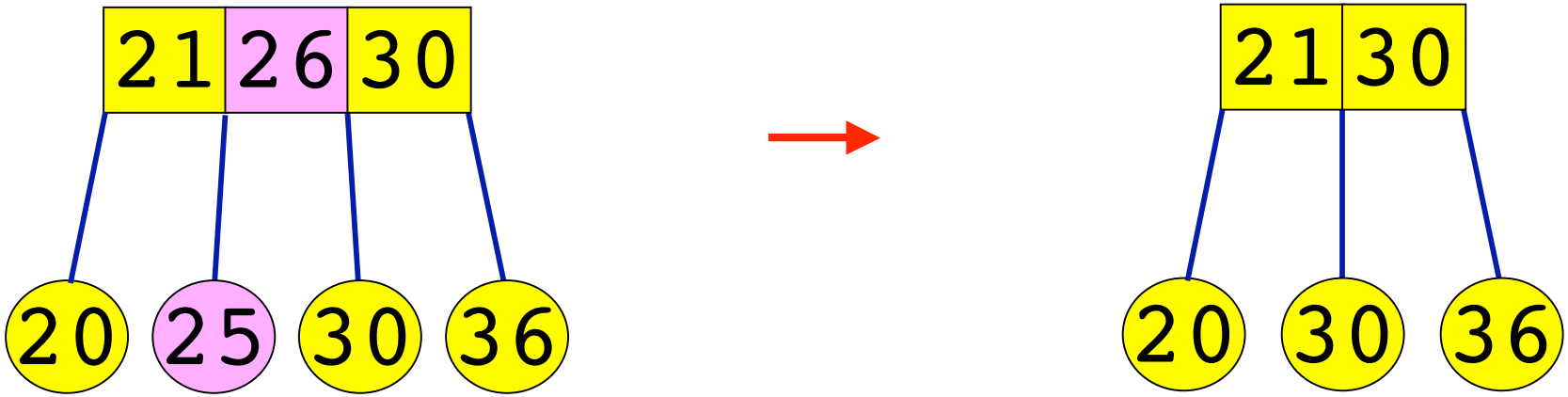
Suppression d'une feuille (1)

Algorithme.

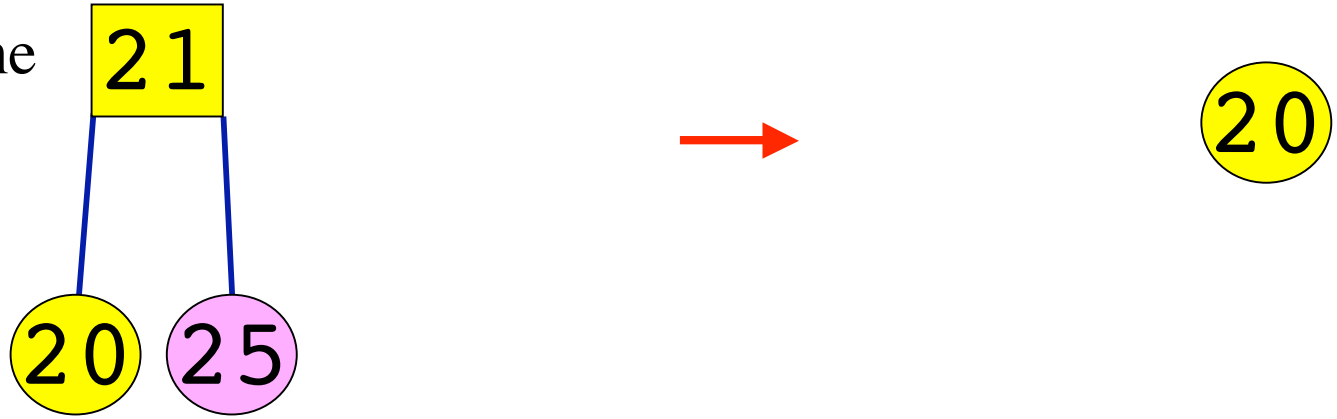
- Supprimer la feuille, puis les balises figurant sur le chemin de la feuille à la racine.
- Si les nœuds ainsi modifiés ont toujours a fils, l'arbre est encore a-b.
- Si un nœud possède seulement a -1 fils, examiner ses *frères* adjacents.
  - Si l'un de ces frères possède au moins a + 1 fils, il suffit de faire un partage avec ce frère.
  - Sinon, les frères adjacents ont a fils, la fusion avec l'un deux produit un nœud ayant $2a - 1 \leq b$ fils.

Exemples de suppression (1)

Suppression de 25

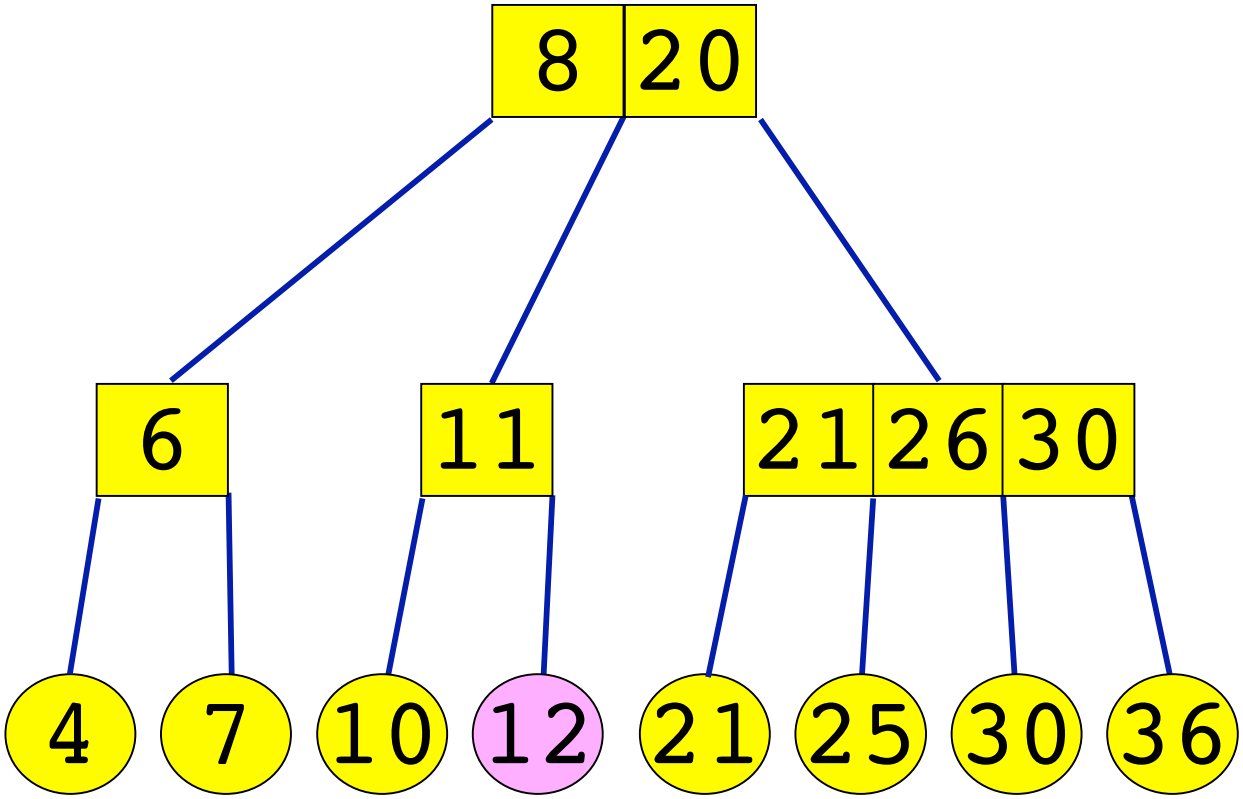


racine



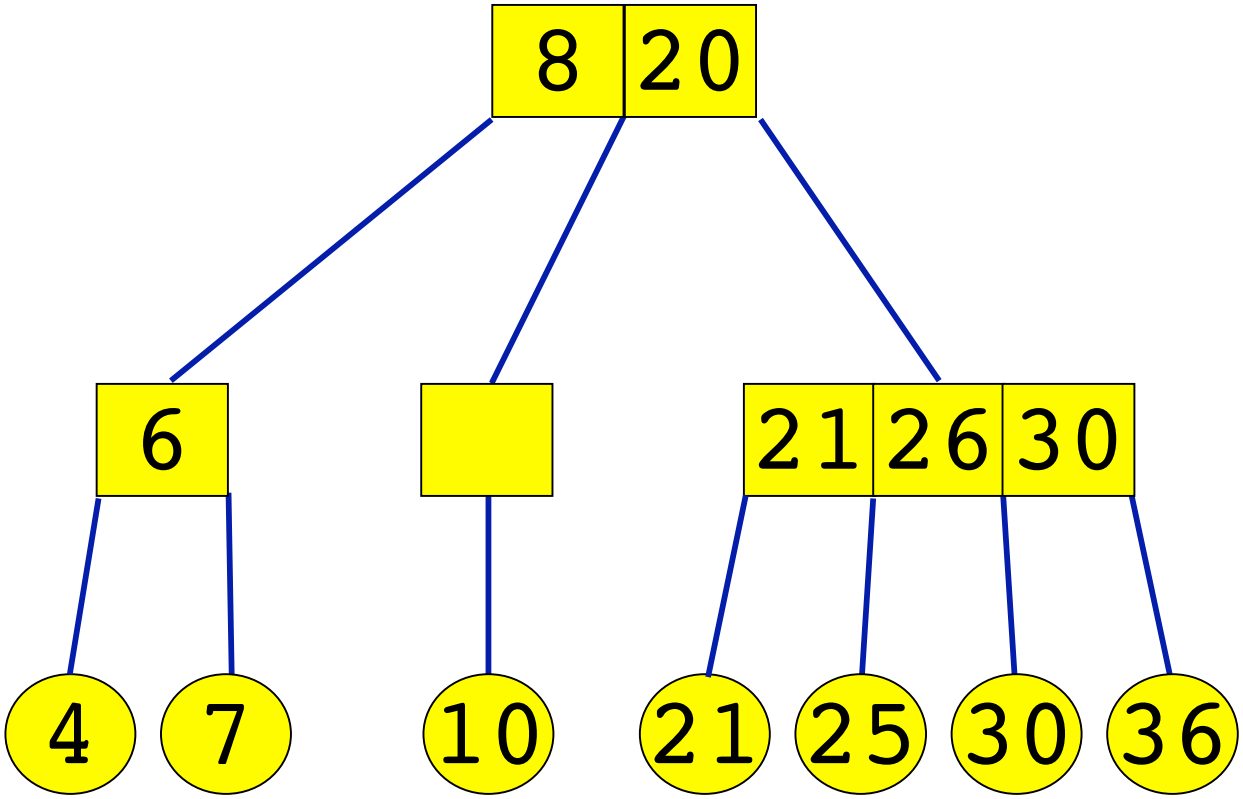
Suppression de 12 (1)

Suppression de 12



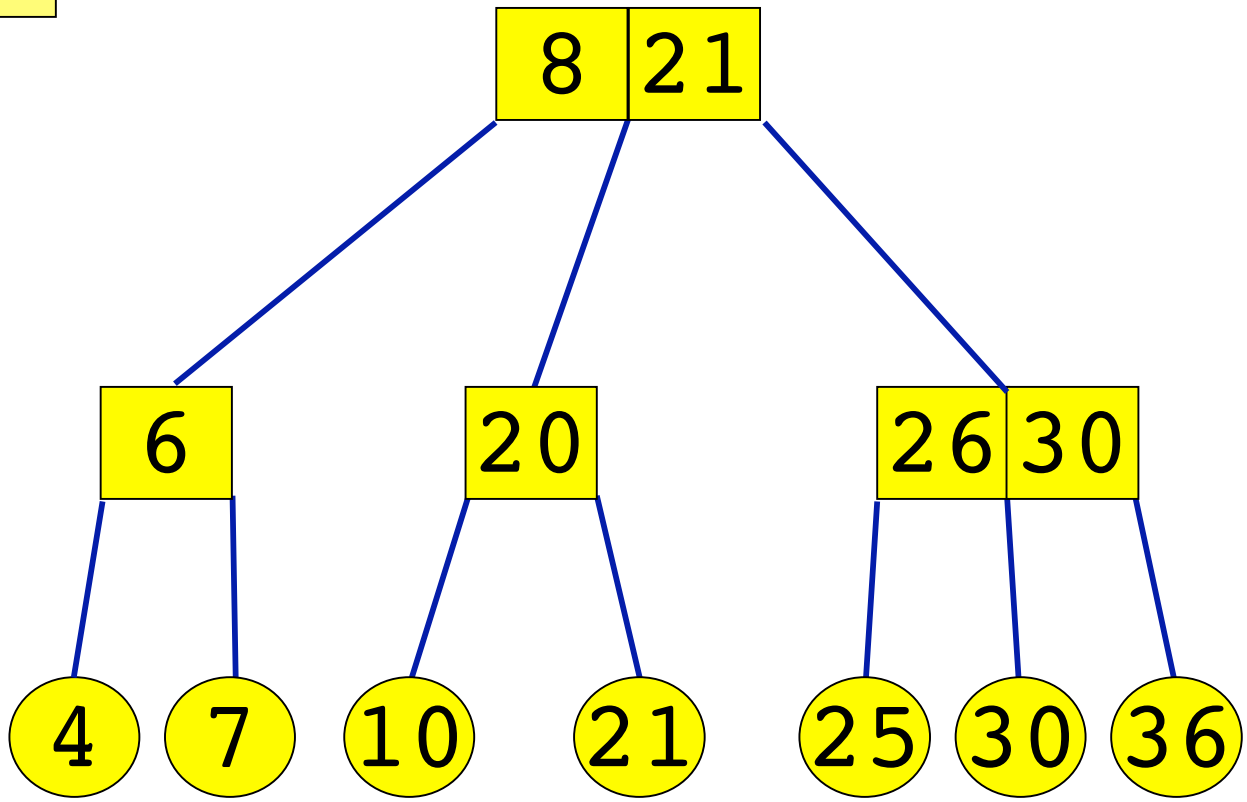
Suppression de 12 (2)

L'arbre n'est plus 2-4



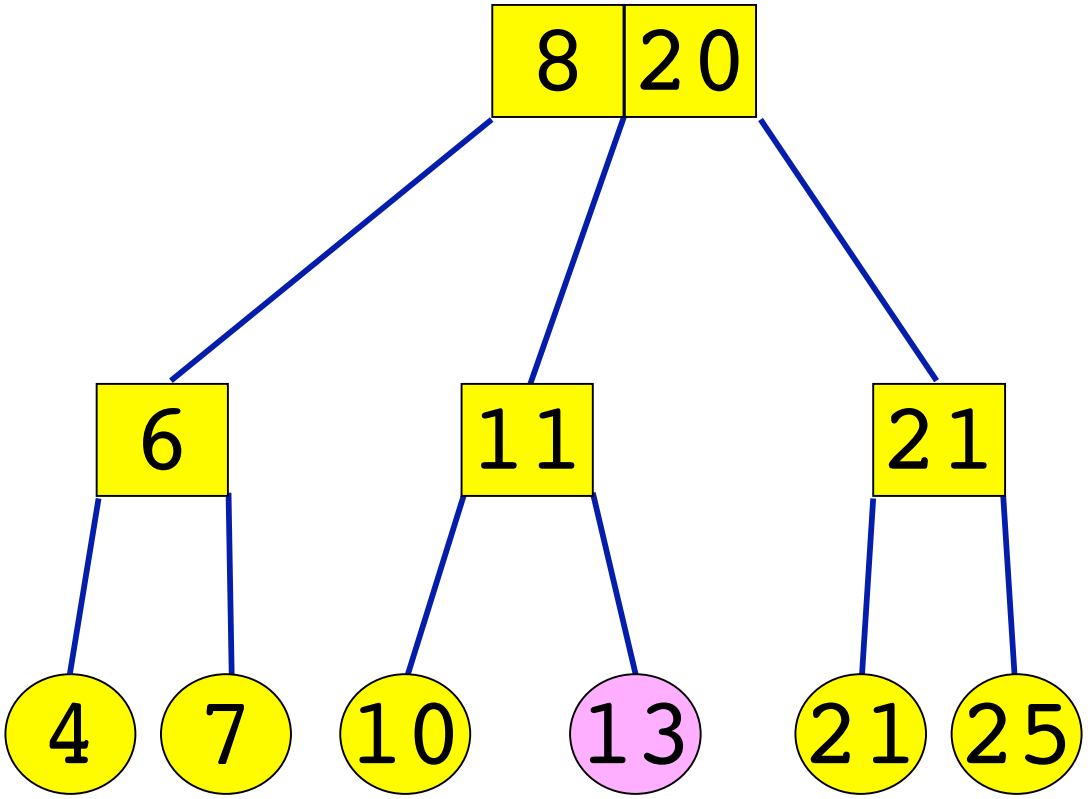
Suppression de 12 (3)

Après partage avec le frère droit.



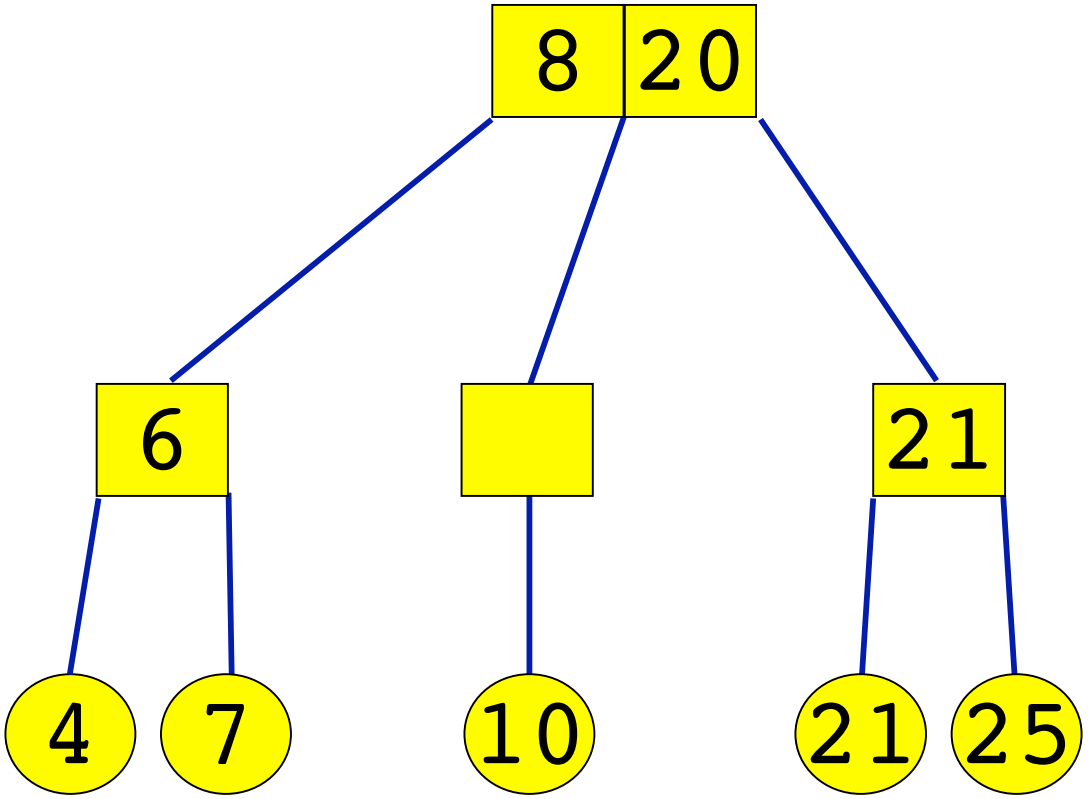
Suppression de 13 (1)

Suppression de 13



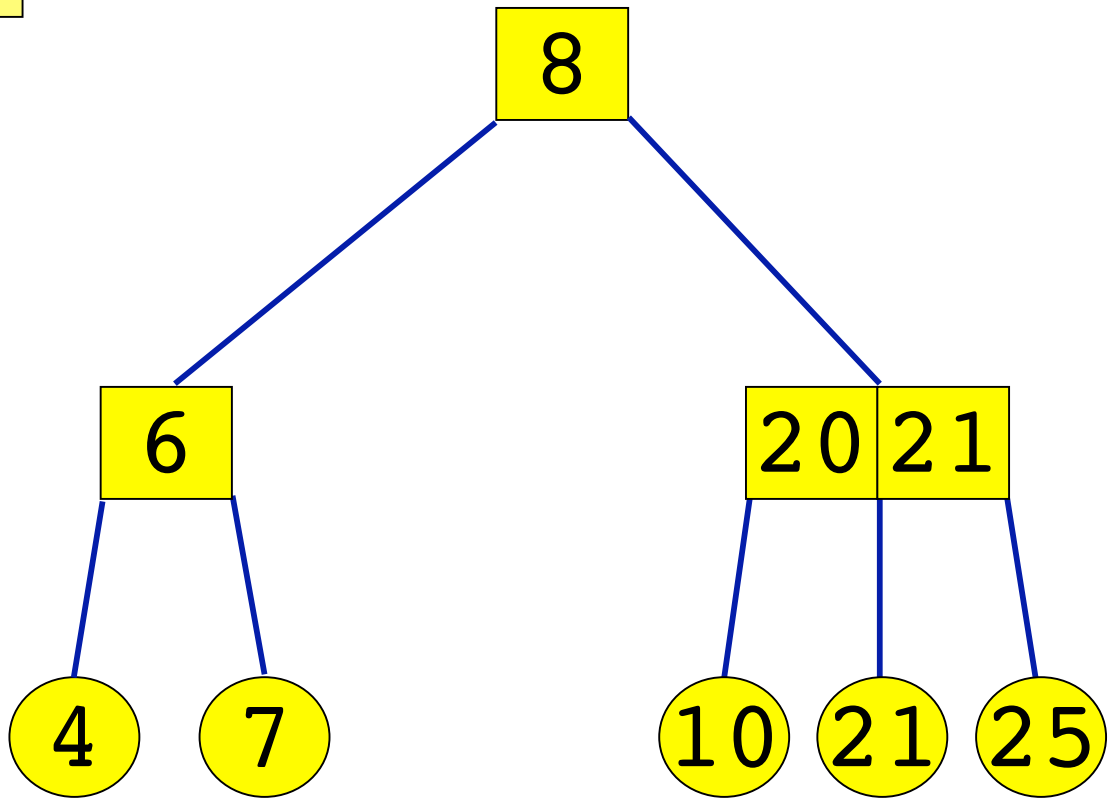
Suppression de 13 (2)

L'arbre n'est plus 2-4



Suppression de 13 (3)

Après fusion avec le frère droit.



`final`

La déclaration d'une variable `final` doit toujours être accompagnée d'une initialisation.

Une *méthode* `final` ne peut être surchargée.

Une *variable* `final` ne peut être modifiée.

Attention ! Si la variable est un tableau, donc une référence, cette référence ne change plus, mais les *valeurs* du tableau peuvent être modifiées.

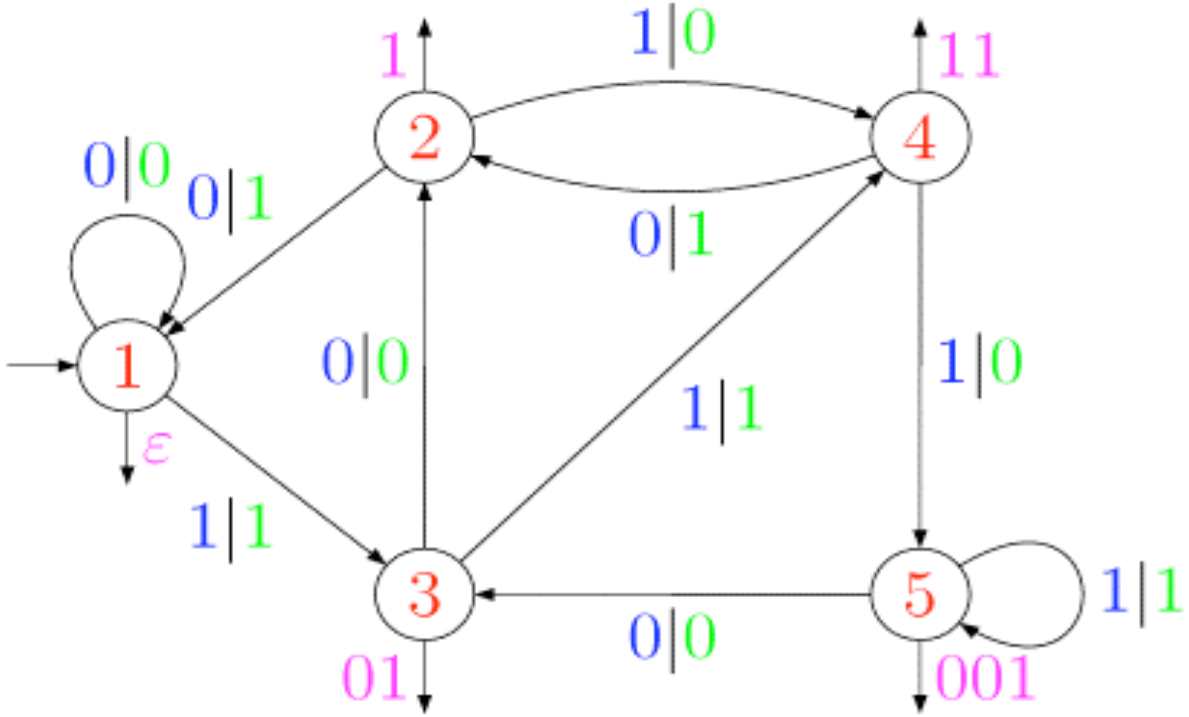
Une *classe* `final` ne peut pas avoir de classe dérivée. Les méthodes d'une *classe* `final` sont implicitement `final`.

final

```
class Test
{
  static final int n = 100;
  static final int[] a = {1, 2, 3};

  public static void main(String args[])
  {
    a[0] = 5; // OK
    n = 9; // Erreur ! Variable finale
  }
}
```

Spécial pub. La multiplication par 5

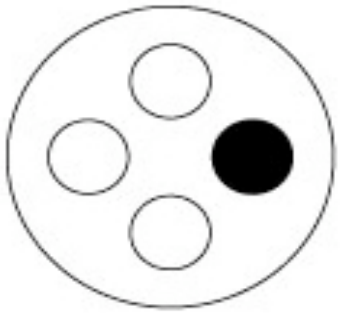


13 x 5 = 65
Entrée : 1011
Sortie : 1000001

Cet automate réalise la multiplication par 5 en binaire inversé.

Un petit casse-tête (1)

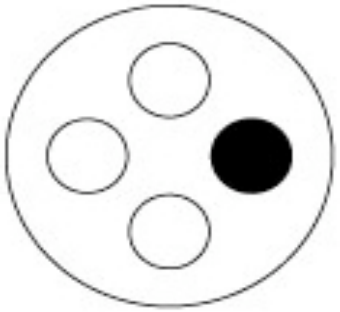
Un joueur a les yeux bandés. Face à lui, un plateau circulaire sur lequel sont disposés en carré quatre jetons, blancs d'un côté et noirs de l'autre. La configuration de départ est inconnue du joueur.
Le but du jeu est d'avoir les quatre jetons du côté blanc.



Configurations possibles,
à une rotation près.

Règles du jeu

Le joueur peut retourner autant de jetons qu'il le souhaite, mais sans les déplacer. A chaque tour, le maître de jeu annonce si la configuration obtenue est gagnante ou pas, puis effectue une rotation du plateau de 0, 1, 2 ou 3 quarts de tours.
Peut-on gagner à coup sûr ? En combien de coups ?



La méthode `toString` (1)

Méthode de la classe `java.lang.Object`.

```
public String toString()
```

Elle retourne une chaîne de caractères formée par le nom de la classe, suivi de @, du codage du type puis de la référence

B	byte	J	long
C	char	L*nom_de_classe*	classe
D	double	S	short
F	float	Z	boolean
I	int		

Précédé de [dans le cas d'un tableau

La méthode `toString` (2)

```
public static void main(String args[])
{
  byte[] T = {5, 2, 6};
  System.out.println(T);
}
```

```
>[B@f96da1
[ --> Tableau
B --> de bytes
@ --> Adresse, suivi de l'adresse en
héxadécimal
```

La méthode `toString` (2)

Elle est parfois surchargée. Par exemple, dans

`java.lang.Class`,

elle donne seulement le nom de la classe. Dans

`java.lang.String`

elle retourne la chaîne de caractères.

```
public static void main(String[]args) {
   String p = "bonjour";
   System.out.println(p);
} // affichera : bonjour
```

Les secrets de la classe `String`

- La classe `String` est `final`.
- Une chaîne de caractères renvoie toujours à la même instance de la classe `String`.
- Un tas de méthodes utiles :

```
int length()
int charAt(int index)
int compareTo(string s)
String replace(char c1, char c2)
String toLowerCase()
String toUpperCase()
String valueOf(int i)
```

Autres méthodes utiles de la classe `String`

```
boolean startsWith(string prefix)
boolean endsWith(string suffix)
int indexOf(int ch, int from)
int indexOf(string facteur)
int lastIndexOf(int ch, int from)
int lastIndexOf(string facteur)
String substring(int i, int j)
int indexOf(int c, int fromIndex)
int indexOf(string facteur)
String substring(int i, int j)
String concat(string s), etc.
```

La classe String

```
class Test
{
  static final int[] a = {1, 2, 3};
  static final int[] b = {1, 2, 3};

  public static void main(String args[])
  {
    String s = "bonjour";
    String t = "bonjour";
    System.out.println(a == b); // false
    System.out.println(s == t); // true
  }
}
```