Algo Design

Cours 5-1

Recherche en table

et Arbres

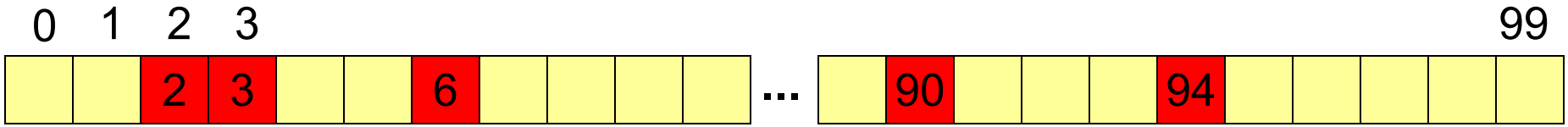
Recherche en table

- Problème : Trouver une information à laquelle est associée une « clé »
- Exemple : Annuaire, Bibliothèque, …
- **Fonctions :**
  - recherche,
  - insertion,
  - suppression.
- Quelle structure de données utiliser ?

Notations

- Les clés sont prises dans un ensemble isomorphe à [0,*m*-1] (*m* très grand)
- On souhaite stocker ***n*** paires de la forme [clé, information] (en général *n*<<*m*)
- Pour chaque méthode, on s'intéresse à
  - la complexité **spatiale** du stockage
  - la complexité **temporelle** de chaque opération (recherche, insertion, suppression)

Table à adressage direct

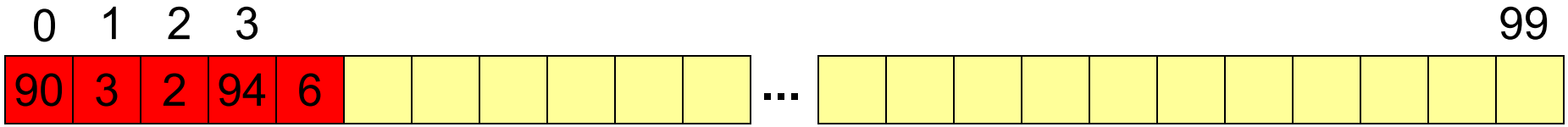


- On utilise un tableau *T* à *m* entrées
  - Stockage en Θ(m)
- **`Recherche`**(*clé*) : retourner *T* [*clé*]
  - complexité Θ(1)
- **`Insertion`**(*clé*,*info*) : *T* [*clé*]=*info*
  - complexité Θ(1)
- **`Suppression`**(*clé*) : *T* [*clé*]=*Nil*
  - complexité Θ(1)

Complexité spatiale

- Il y a 2.10^{11} mots de 8 lettres majuscules
- 1 Mega-octet = 10^6 octets
- 1 Giga-octet = 10^9 octets
- Conclusion :
  un complexité spatiale en $\Theta(m)$ est irréaliste en général

Recherche séquentielle

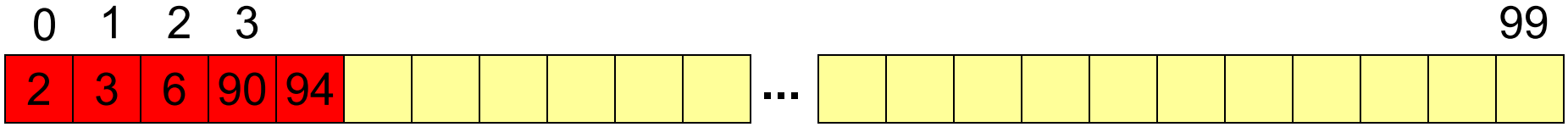


- On utilise un tableau *T* à *n* entrées
  - Stockage en Θ(n)
- **`Recherche`**(*clé*) : parcours de *T*
  - complexité Θ(n)
- **`Insertion`**(*clé*,*info*) : (recherche préalable ?)
  - complexité Θ(1)
- **`Suppression`**(*clé*) : recherche+décalage
  - complexité Θ(n)

Complexité temporelle

- Un dictionnaire contient de l'ordre de $\mathbf{10^7}$ mots.
- Si un test prend en moyenne un centième de seconde (0,01 s), une recherche complète prend 2 minutes.

Recherche dichotomique

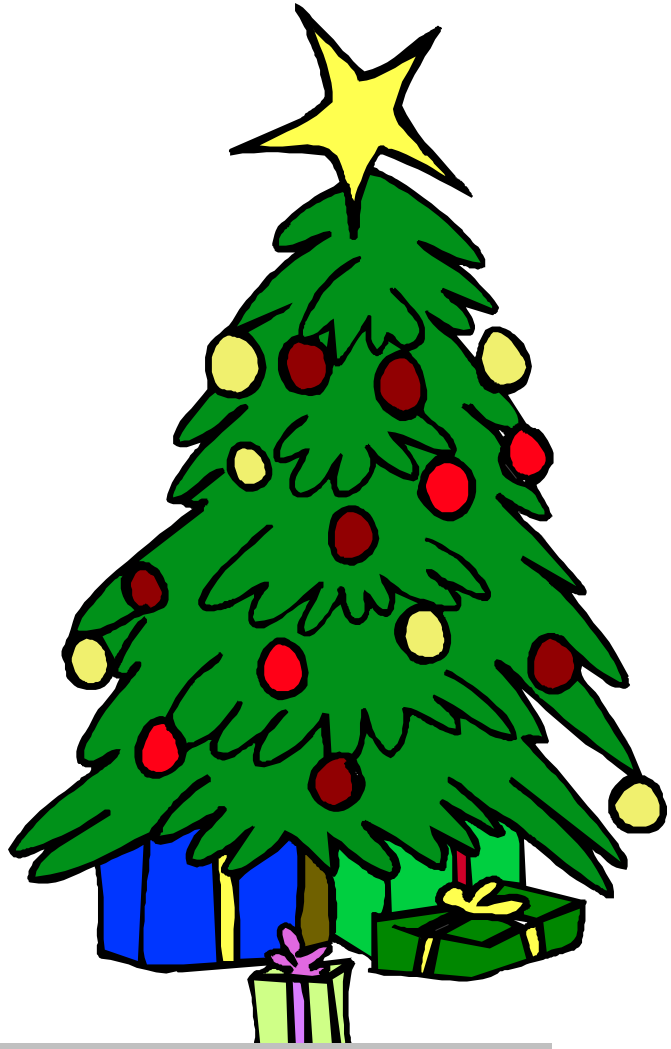


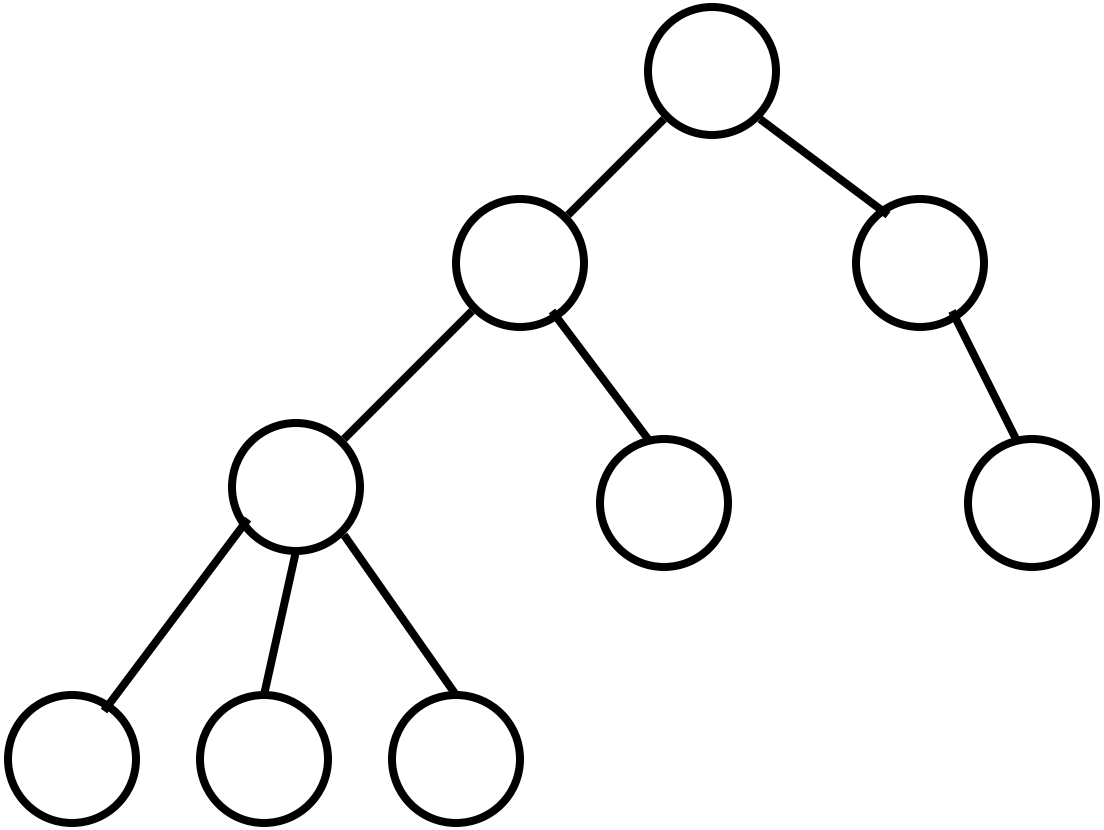
- On utilise un tableau *T* à *n* entrées **classées par ordre croissant**
  - Stockage en Θ(n)
- **`Recherche`**(*clé*) : parcours de *T*
  - complexité Θ(log(n))
- **`Insertion`**(*clé*,*info*) : recherche+décalage
  - complexité Θ(n)
- **`Suppression`**(*clé*) : recherche+décalage
  - complexité Θ(n)

Complexité temporelle

- Un dictionnaire contient de l'ordre de **10^7** mots.
- **$\log_2(10^7) = 24$**
- Si un test prend en moyenne un centième de seconde (0,01 s), une recherche complète prend 0,2 secondes ! *(accès ?)*
- Peut-on faire mieux ??? … parfois oui ...

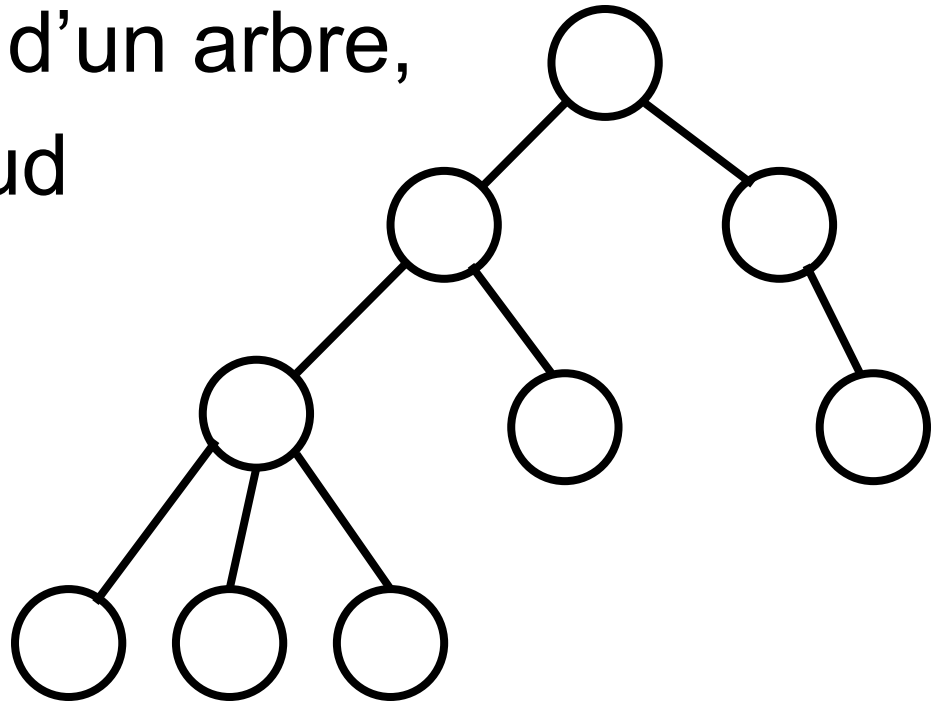
Les Arbres





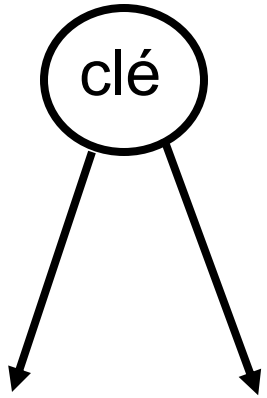
Les arbres : vocabulaire

- Nœud, fils, père, racine, feuille, nœud interne, ancêtre, descendant
- Sous-arbre, hauteur d'un arbre, profondeur d'un nœud
- degré d'un nœud, arité d'un arbre, arbre *k*-aire



Les arbres binaires

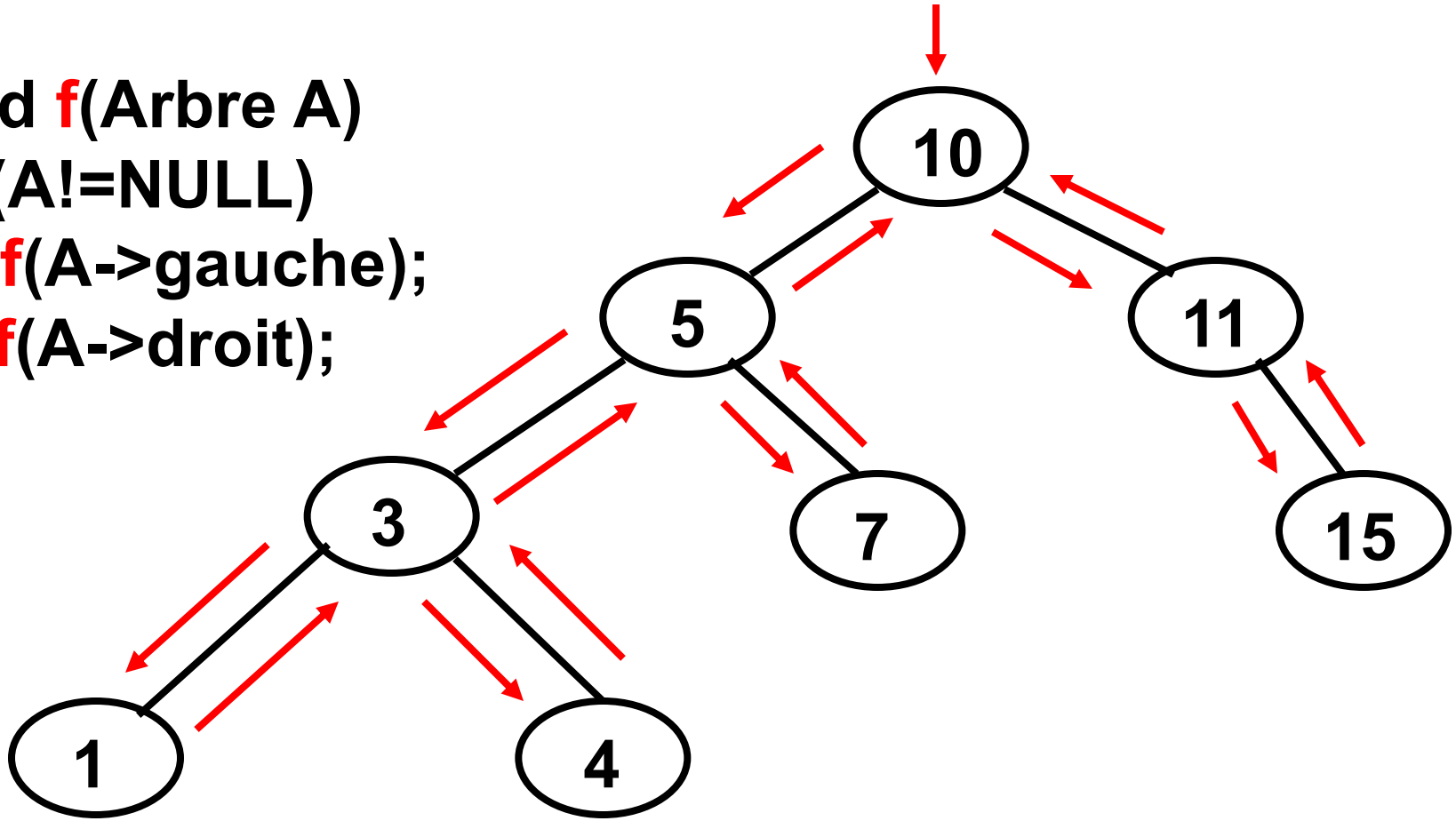
- Chaque nœud a (éventuellement) un fils gauche et/ou un fils droite
- Implémentation :




```
struct noeud {
   type_element cle;
   struct noeud *gauche;
   struct noeud *droit;
};
typedef struct noeud *arbre_binaire;
```

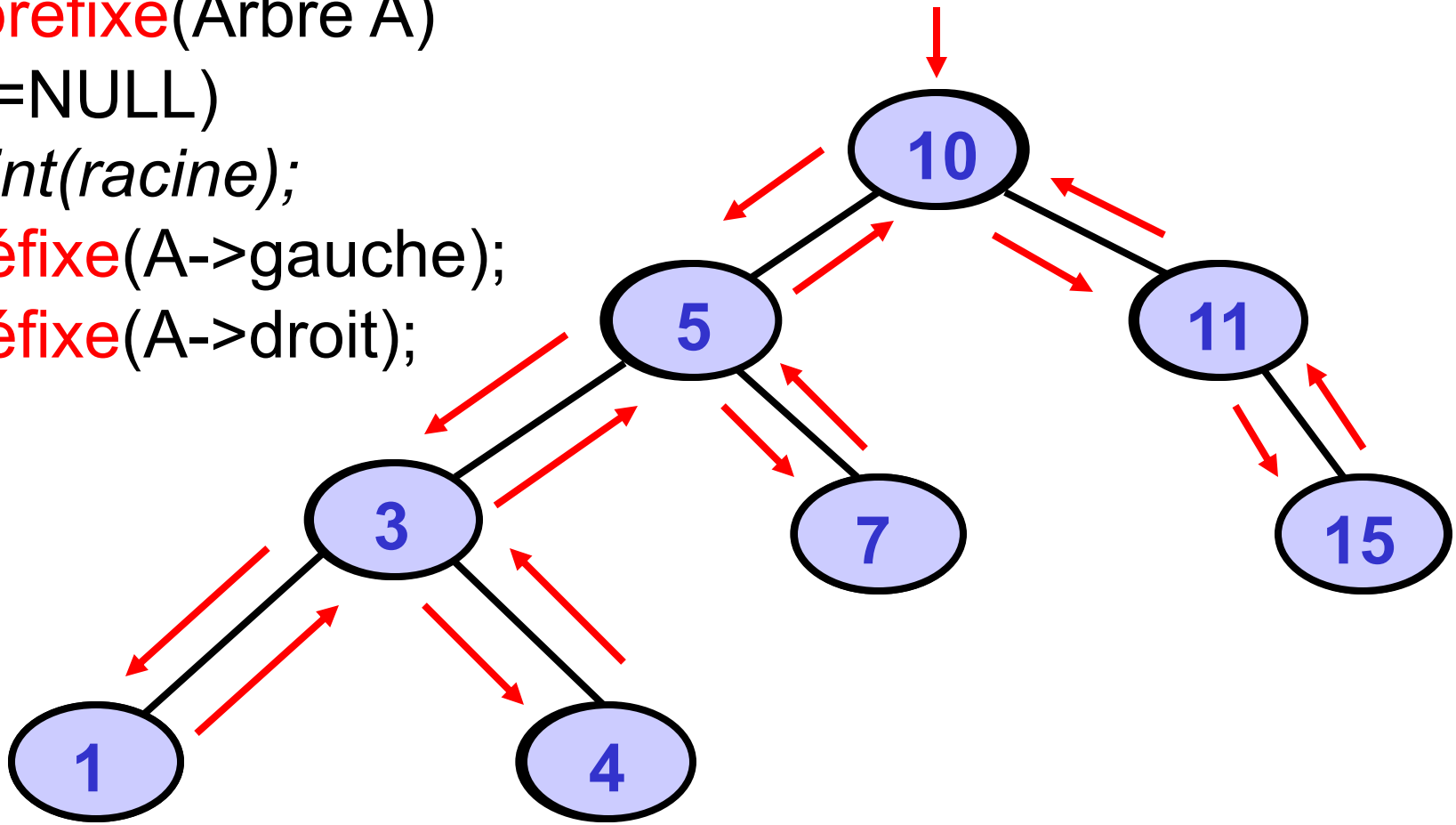
Parcours récursif d'arbre

```
void f(Arbre A)
{if (A!=NULL)
    {f(A->gauche);
     f(A->droit);
}}
```



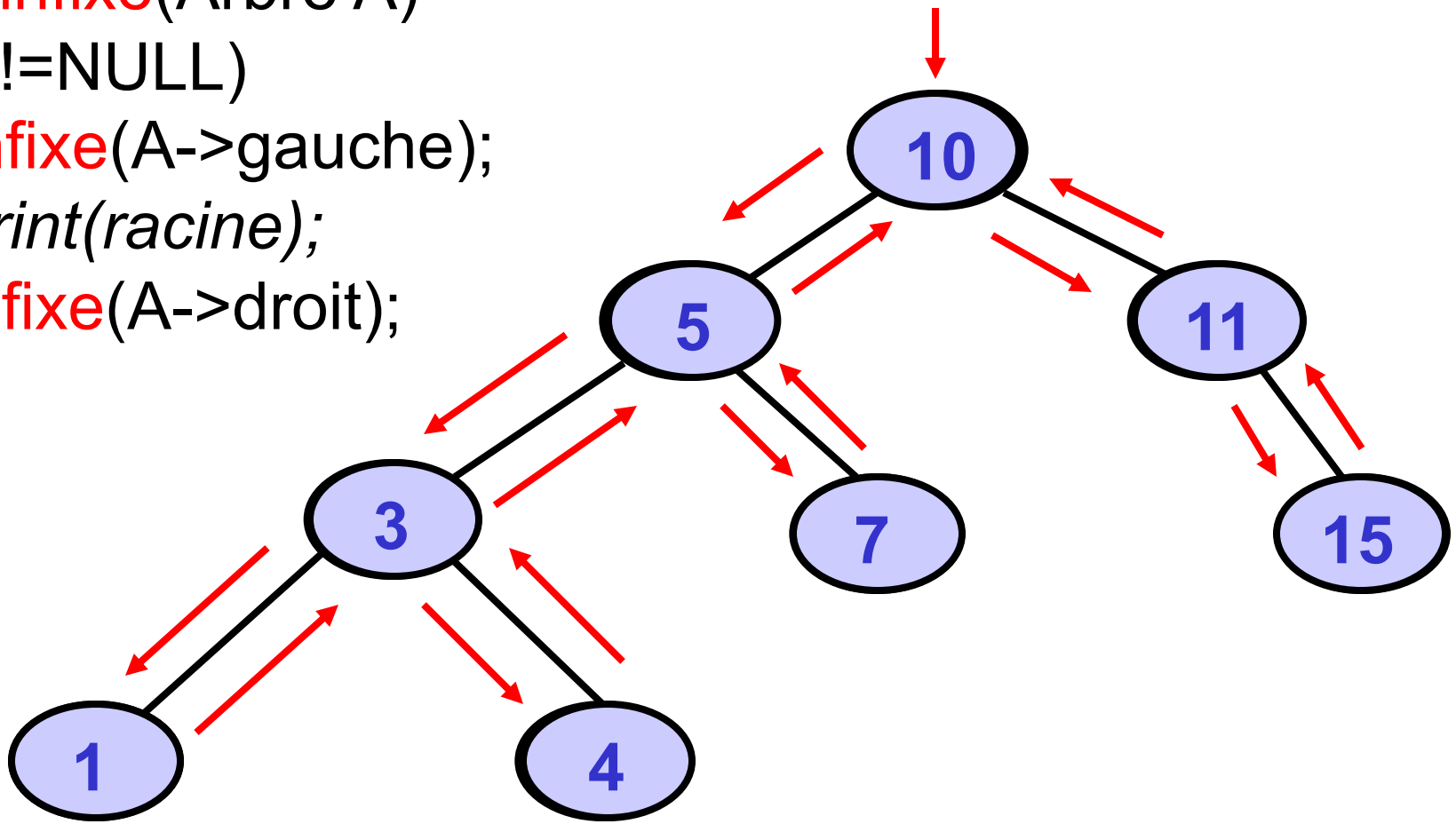
Parcours préfixé

```
void préfixe(Arbre A)
{if (A!=NULL)
    {print(racine);
     préfixe(A->gauche);
     préfixe(A->droit);
}}
```



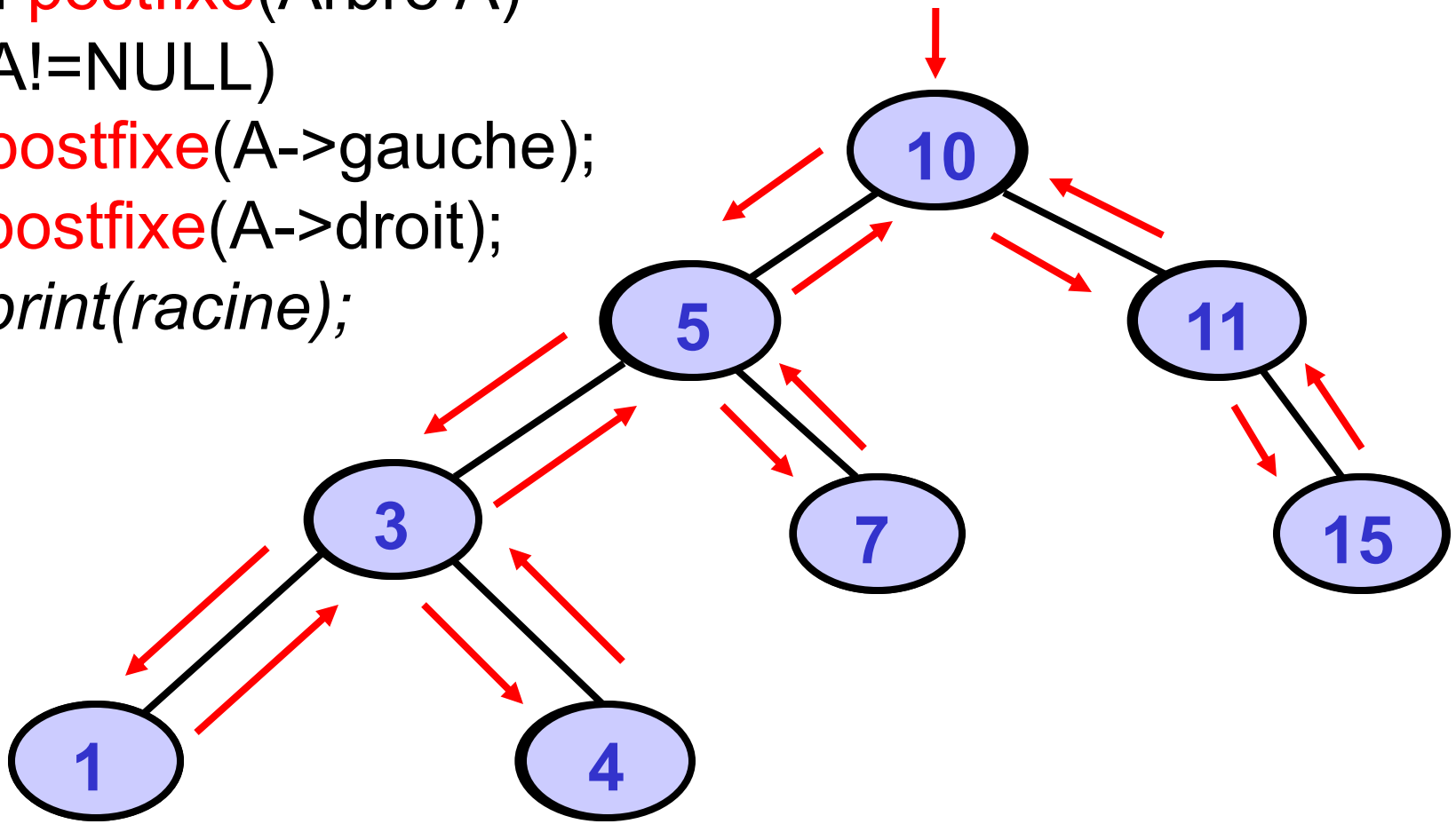
Parcours infixé

```
void infixe(Arbre A)
{if (A!=NULL)
    {infixe(A->gauche);
     print(racine);
     infixe(A->droit);
}}
```



Parcours postfixé

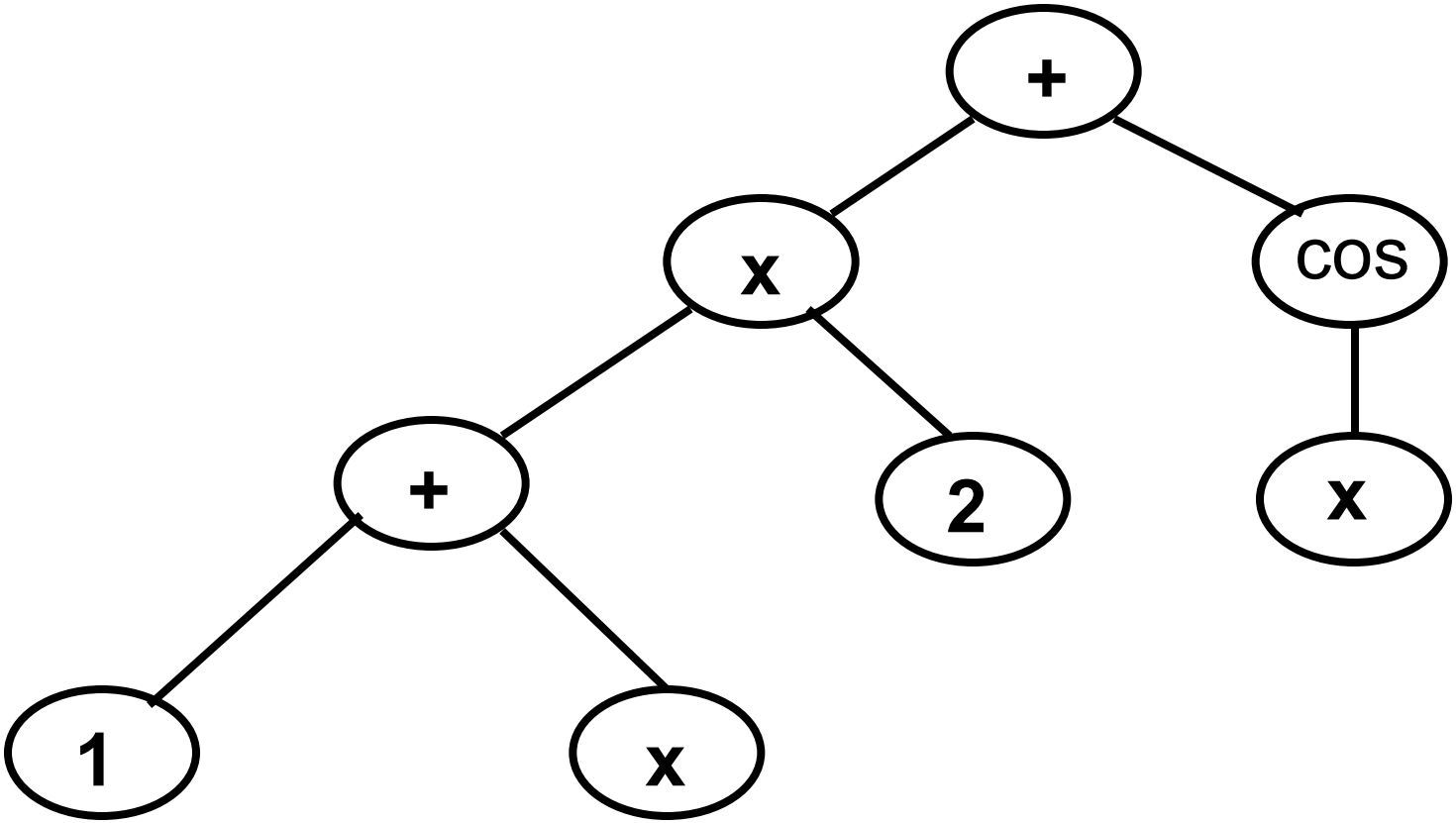
```
void postfixe(Arbre A)
{if (A!=NULL)
    {postfixe(A->gauche);
     postfixe(A->droit);
     print(racine);
}}
```



Dérivation formelle

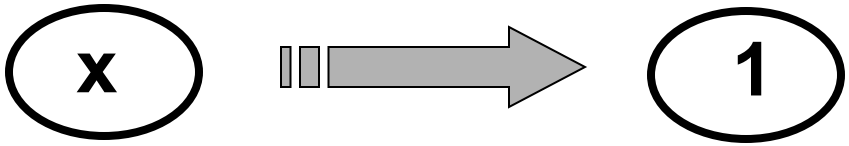
- Représentation de toute expression arithmétique sous forme d'arbre de syntaxe abstraite
- Dérivation = ensemble de règles de transformation appliquée aux arbres de syntaxe abstraite
- Simplification = idem (en plus difficile)

(x+1).2+cos(x)



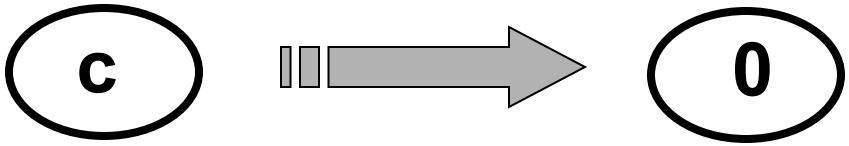
Règles de dérivation

- D(x)=1



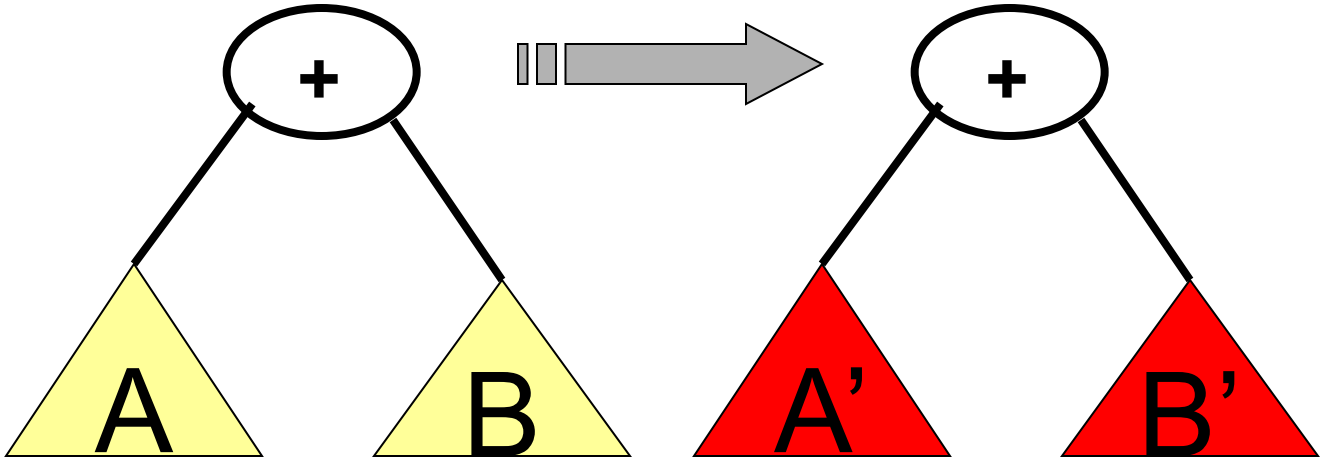
Règles de dérivation

- D(x)=1
- D(c)=0



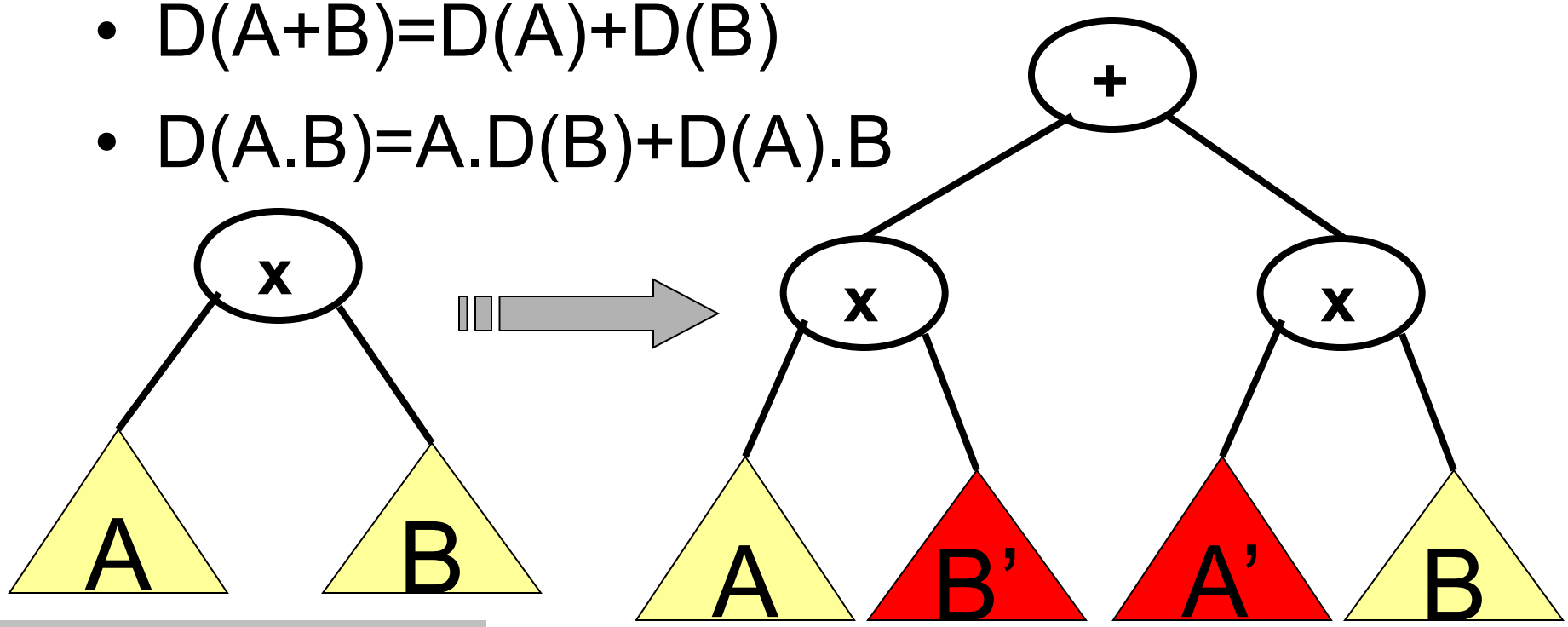
Règles de dérivation

- D(x)=1
- D(c)=0
- D(A+B)=D(A)+D(B)



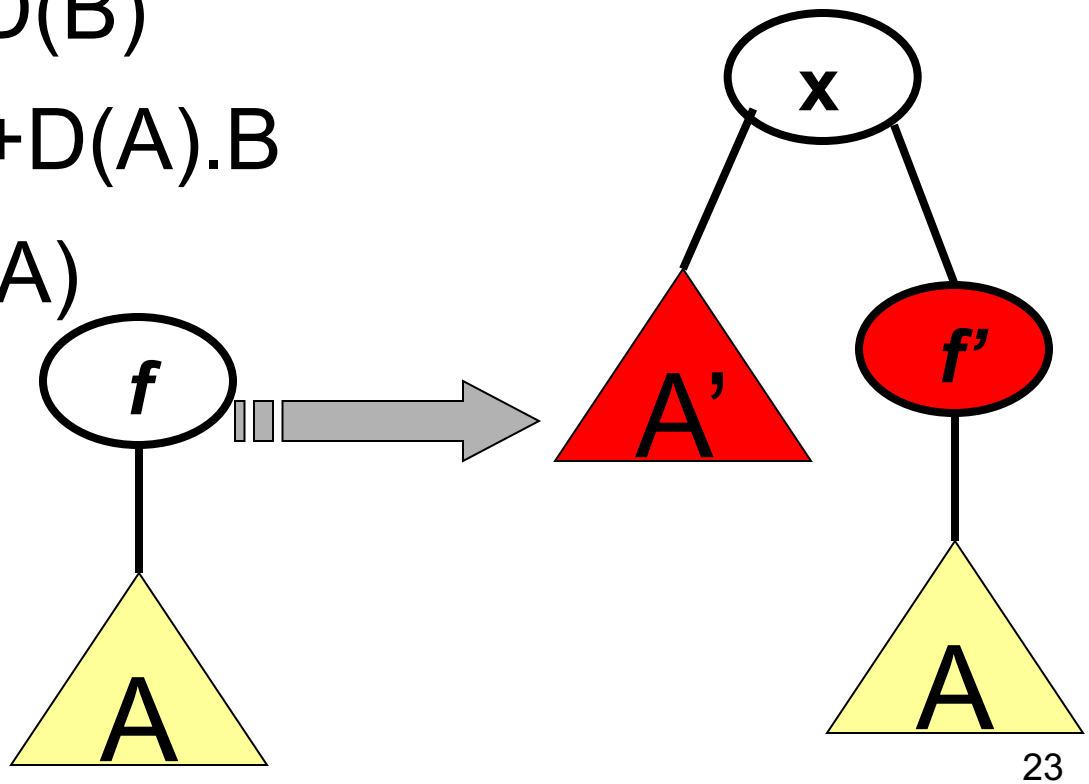
Règles de dérivation

- D(x)=1
- D(c)=0
- D(A+B)=D(A)+D(B)
- D(A.B)=A.D(B)+D(A).B



Règles de dérivation

- D(x)=1
- D(c)=0
- D(A+B)=D(A)+D(B)
- D(A.B)=A.D(B)+D(A).B
- D(f(A))=D(A).f’(A)



Règles de dérivation

- D(x)=1
- D(c)=0
- D(A+B)=D(A)+D(B)
- D(A.B)=A.D(B)+D(A).B
- D(f(A))=D(A).f’(A)
- Ces règles s’appliquent **récursivement** aux arbres de syntaxe abstraite

Règles de simplification

- Il n'y a pas une liste close de règles…
- Problème de définition de la forme simplifiée

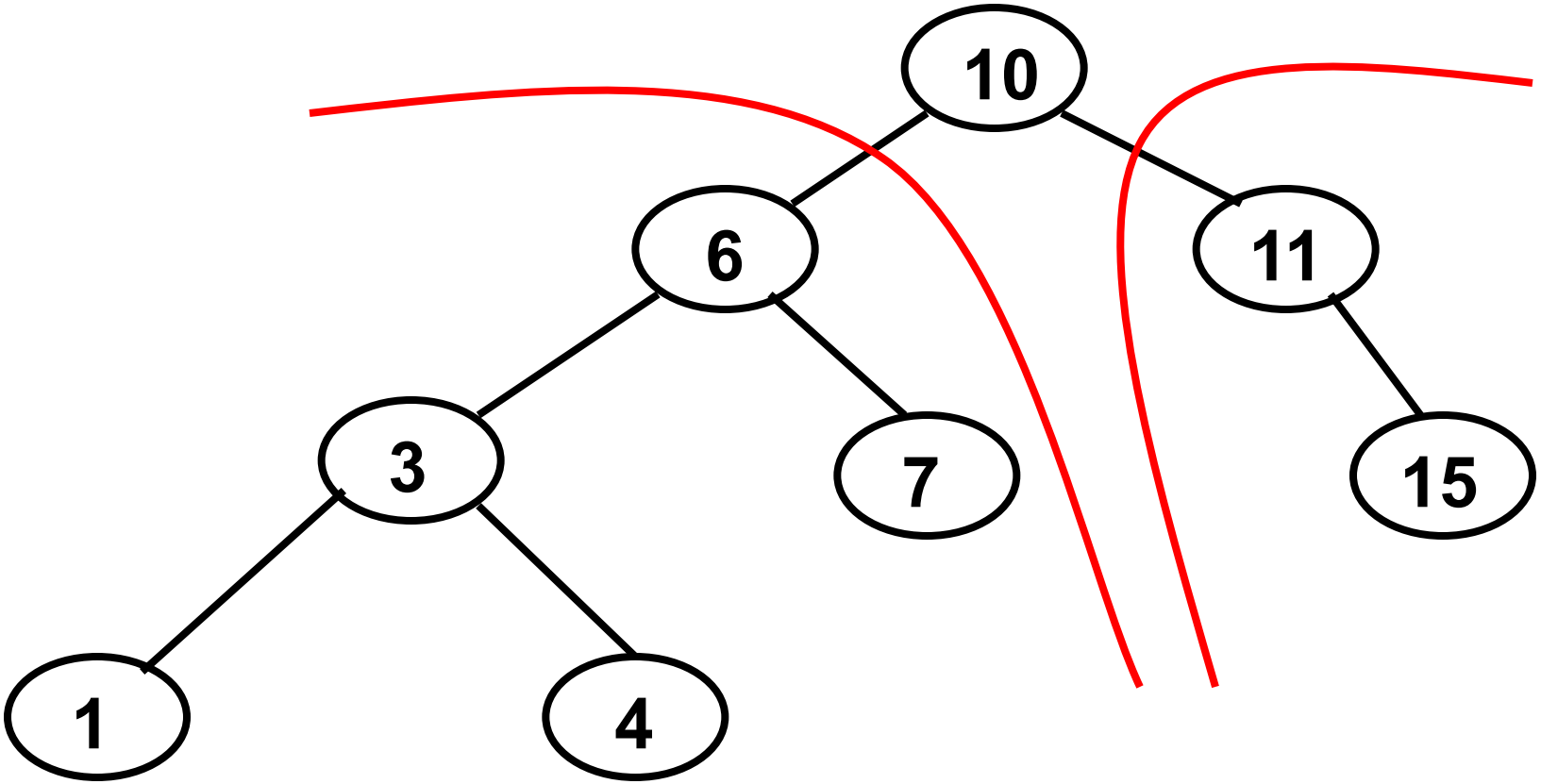
$$(a+b)(a-b) = a^2-b^2$$

- On peut quand même appliquer des règles simples :
  - S(0+A)=A
  - S(1.A)=A
  - …

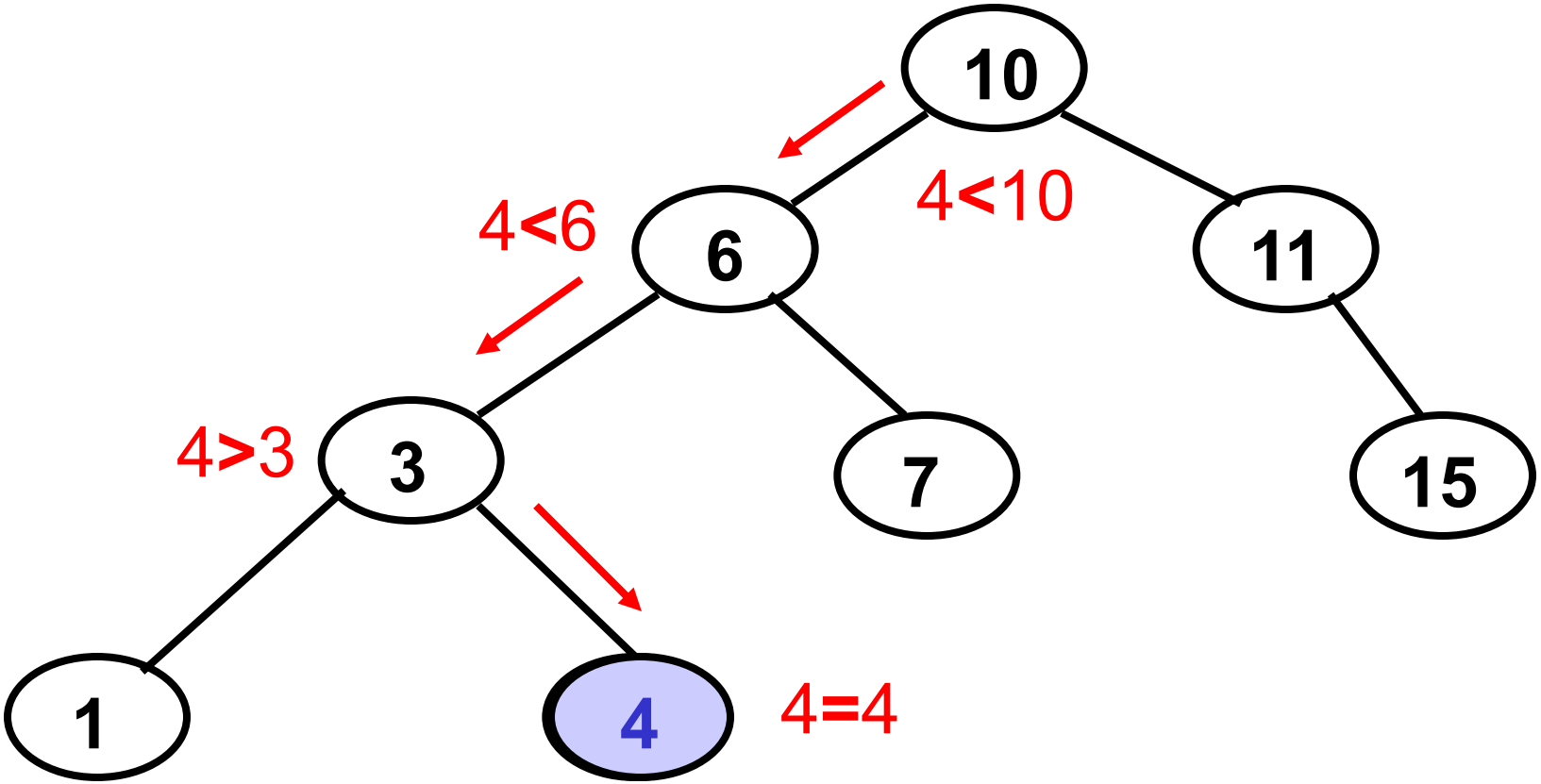
Arbre binaire de recherche

- Structure d'arbre binaire
- Chaque nœud contient une clé
- Toute clé d'un nœud de l'arbre est
  - supérieure à celles des descendants de son fils gauche
  - inférieure à celles des descendants de son fils droit

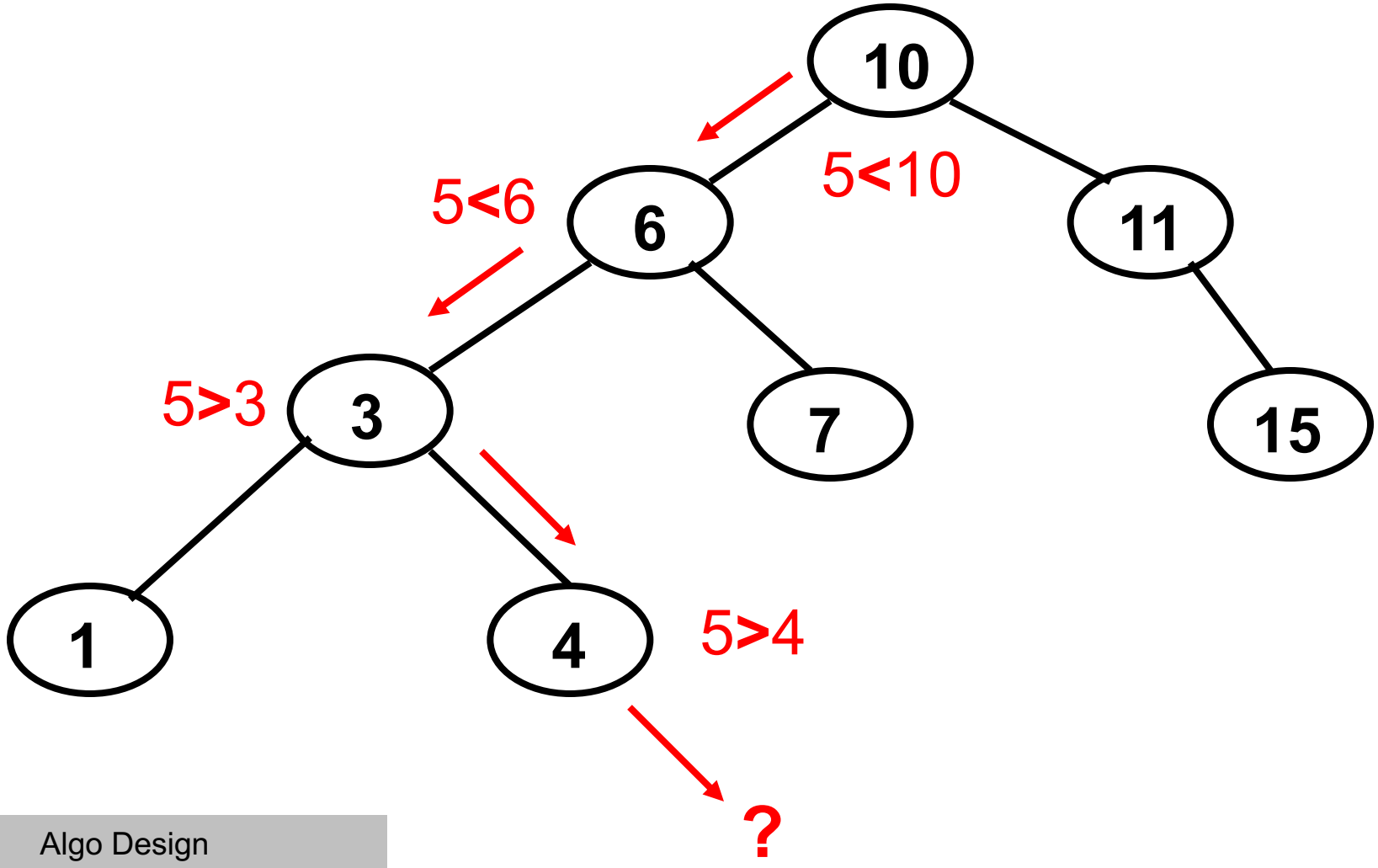
Exemple d'ABR



Recherche de 4



Recherche de 5



Recherche dans un ABR

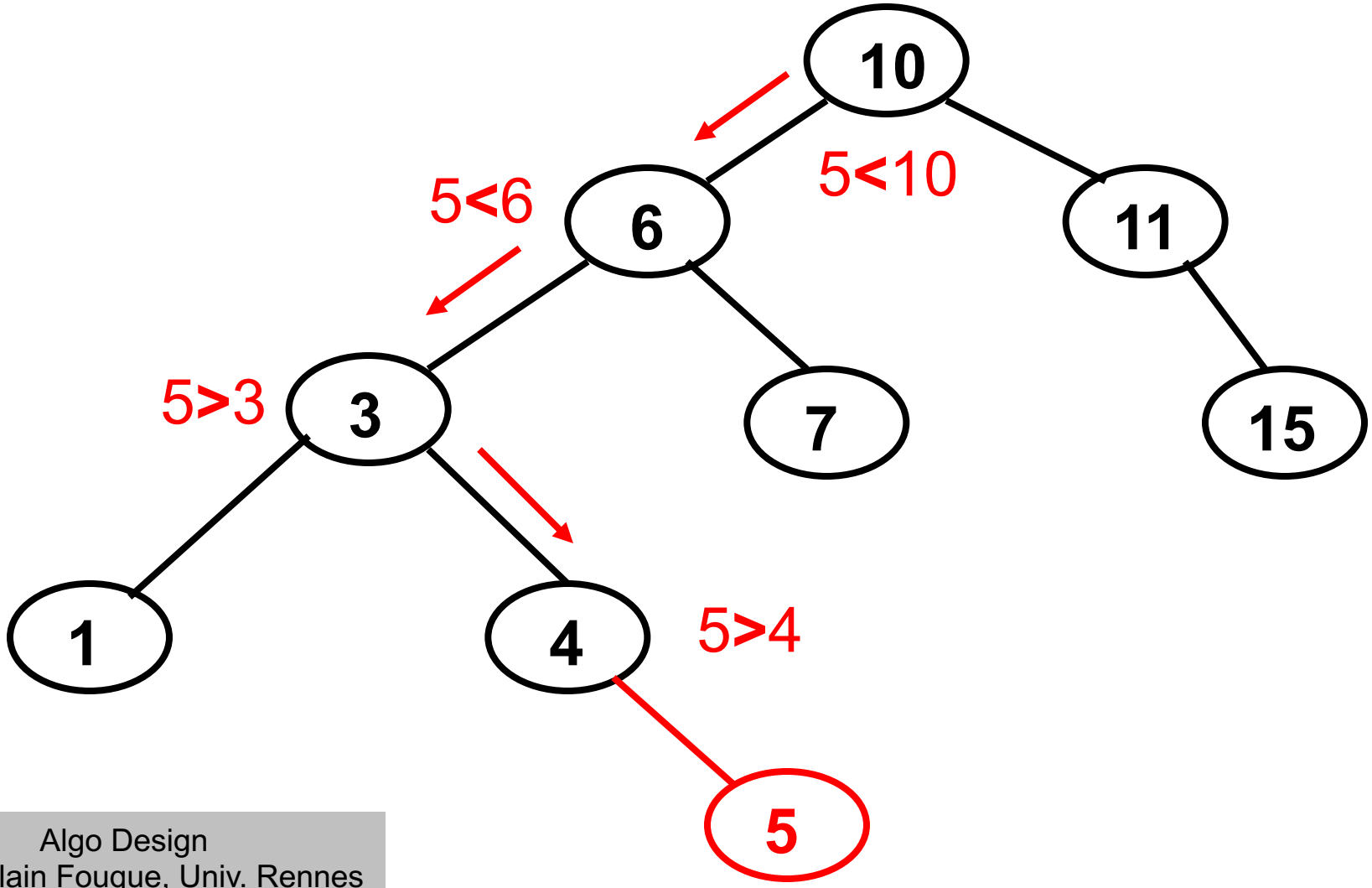
- Recherche de v dans l'ABR A
  - si A==Vide, retourner Echec
  - si v==Clé(A), retourner l'information associée
  - si v<Clé(A)
    - retourner Recherche(Gauche(A),v)
  - sinon
    - retourner Recherche(Droite(A),v)

Recherche dans un ABR

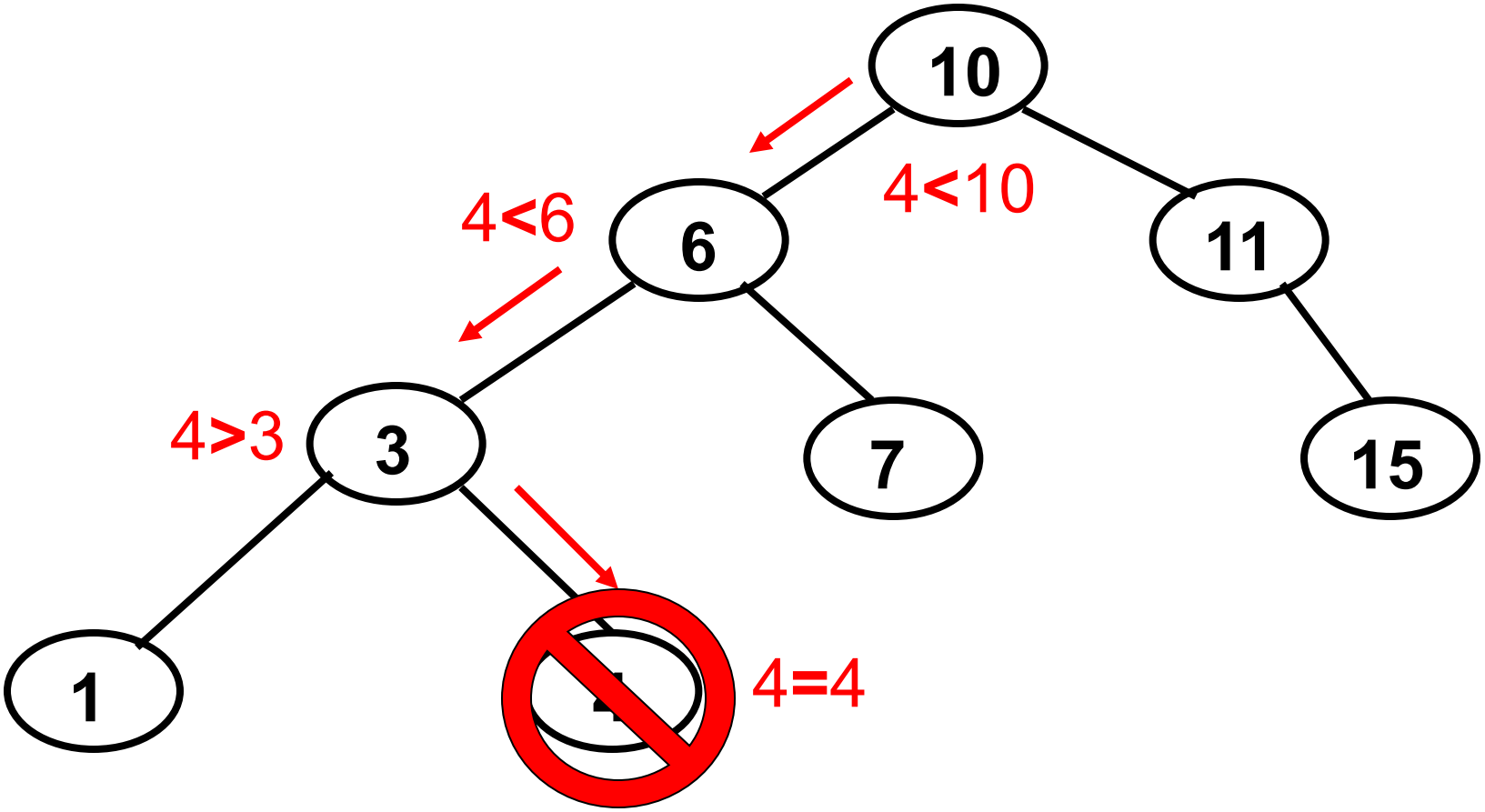
```
typedef struct noeud{
  int cle;
  struct noeud *gauche;
  struct noeud *droit;
}*ABR;
```

```
int recherche(ABR A, int v)
{ if (A==NULL)
    return 0;
  if (A->cle==v)
    return 1;
  if (A->cle<v)
    return recherche(A->droit,v);
  else
    return recherche(A->gauche,v);}
```

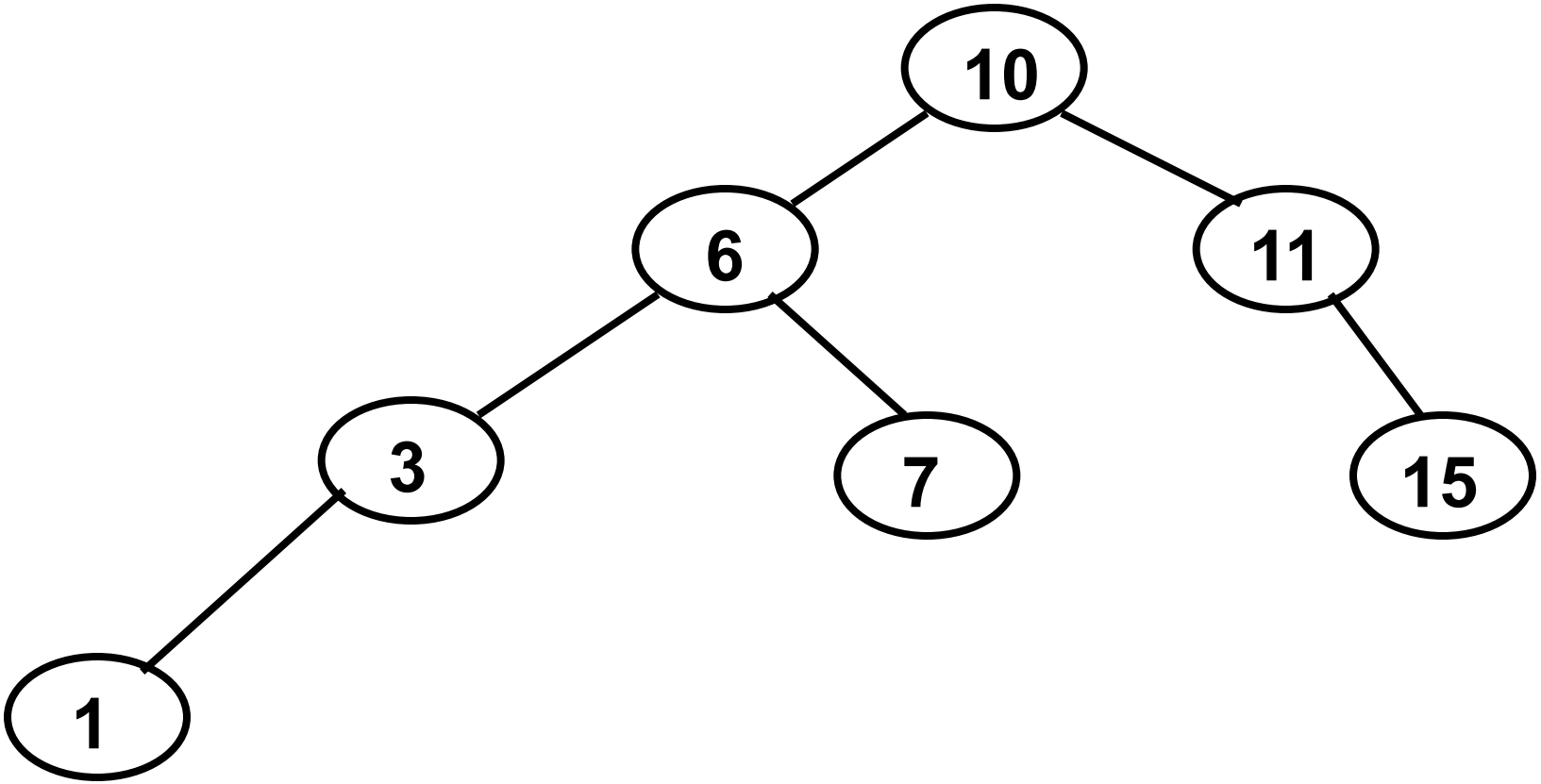
Insertion de 5



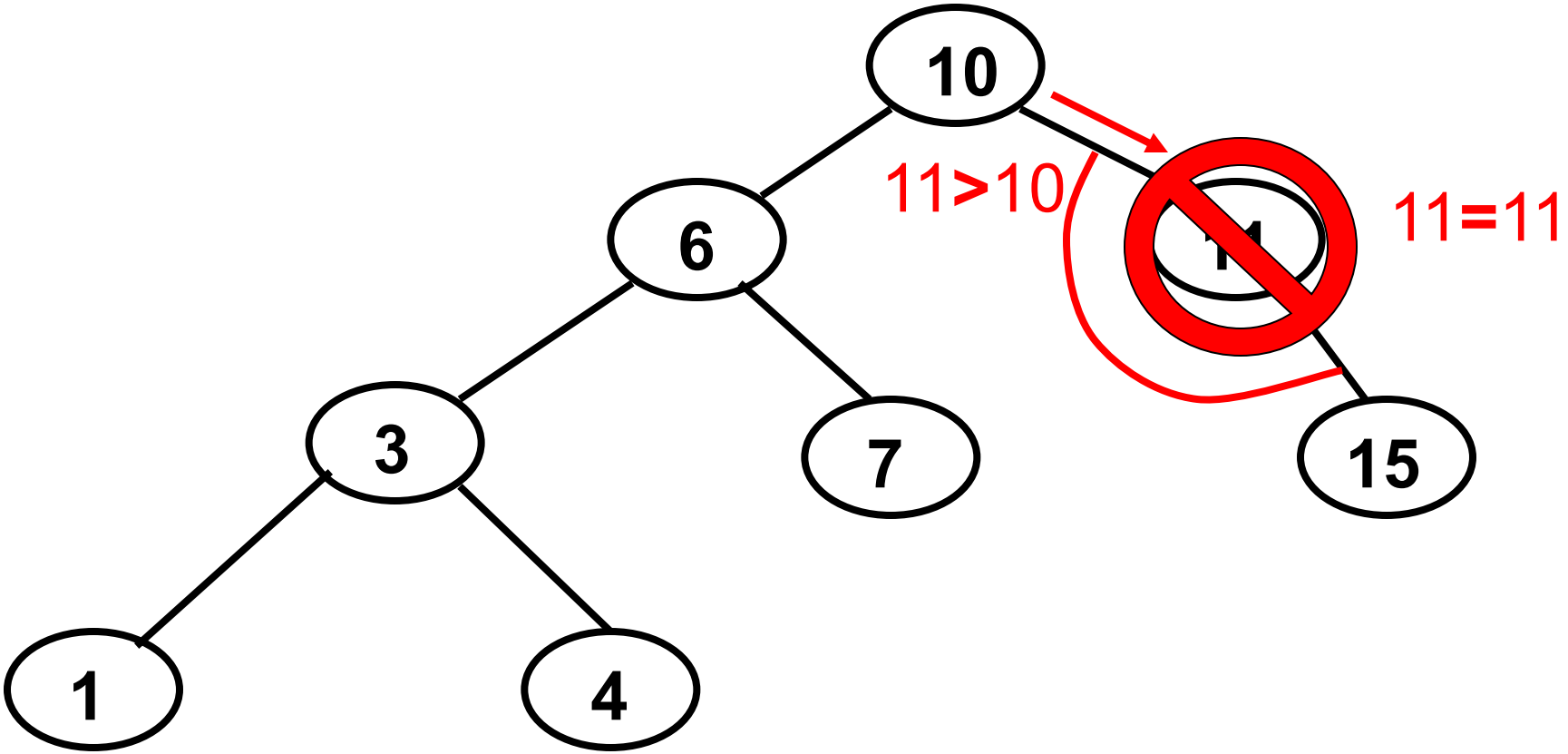
Suppression de 4



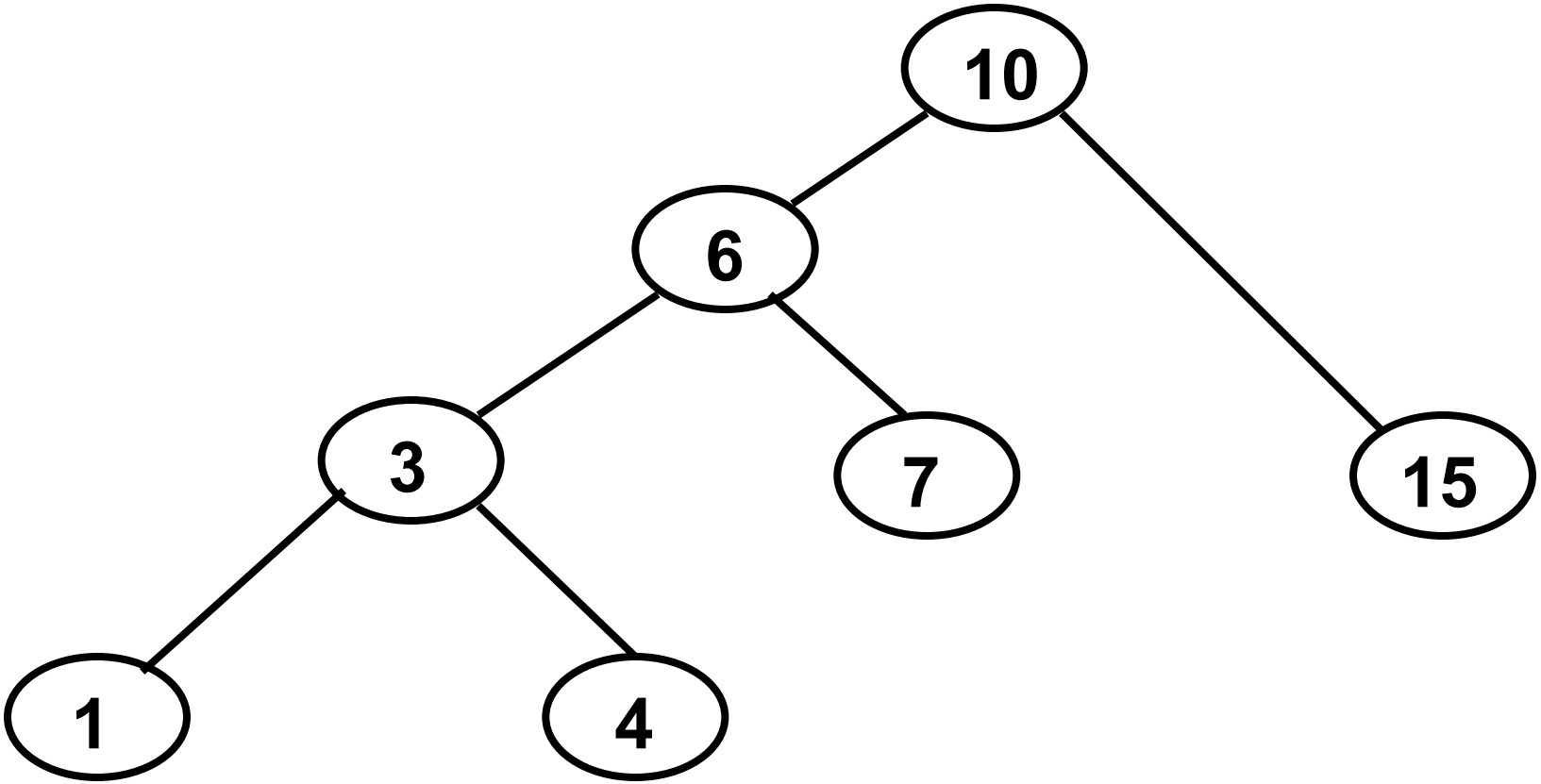
Suppression de 4



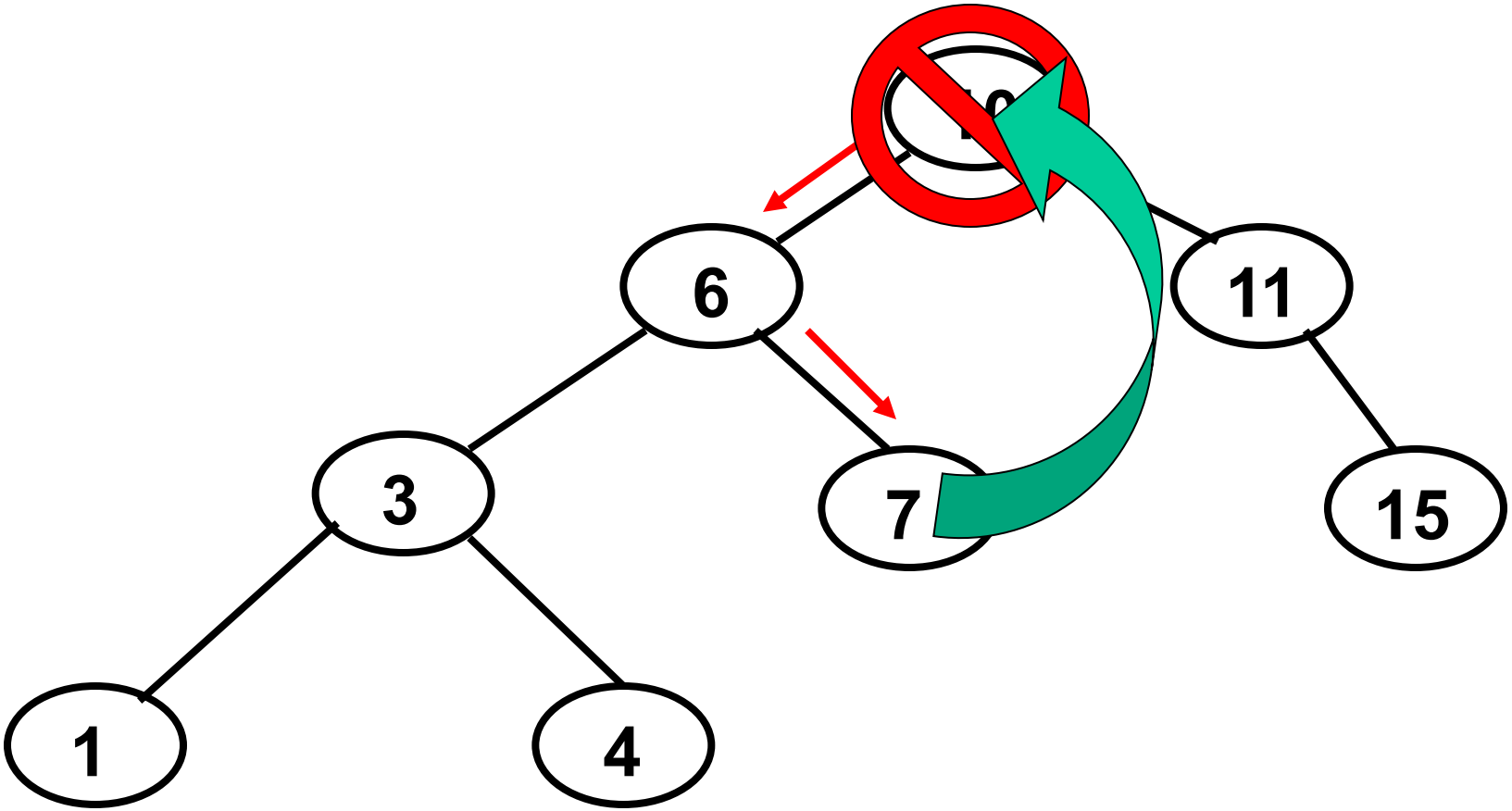
Suppression de 11



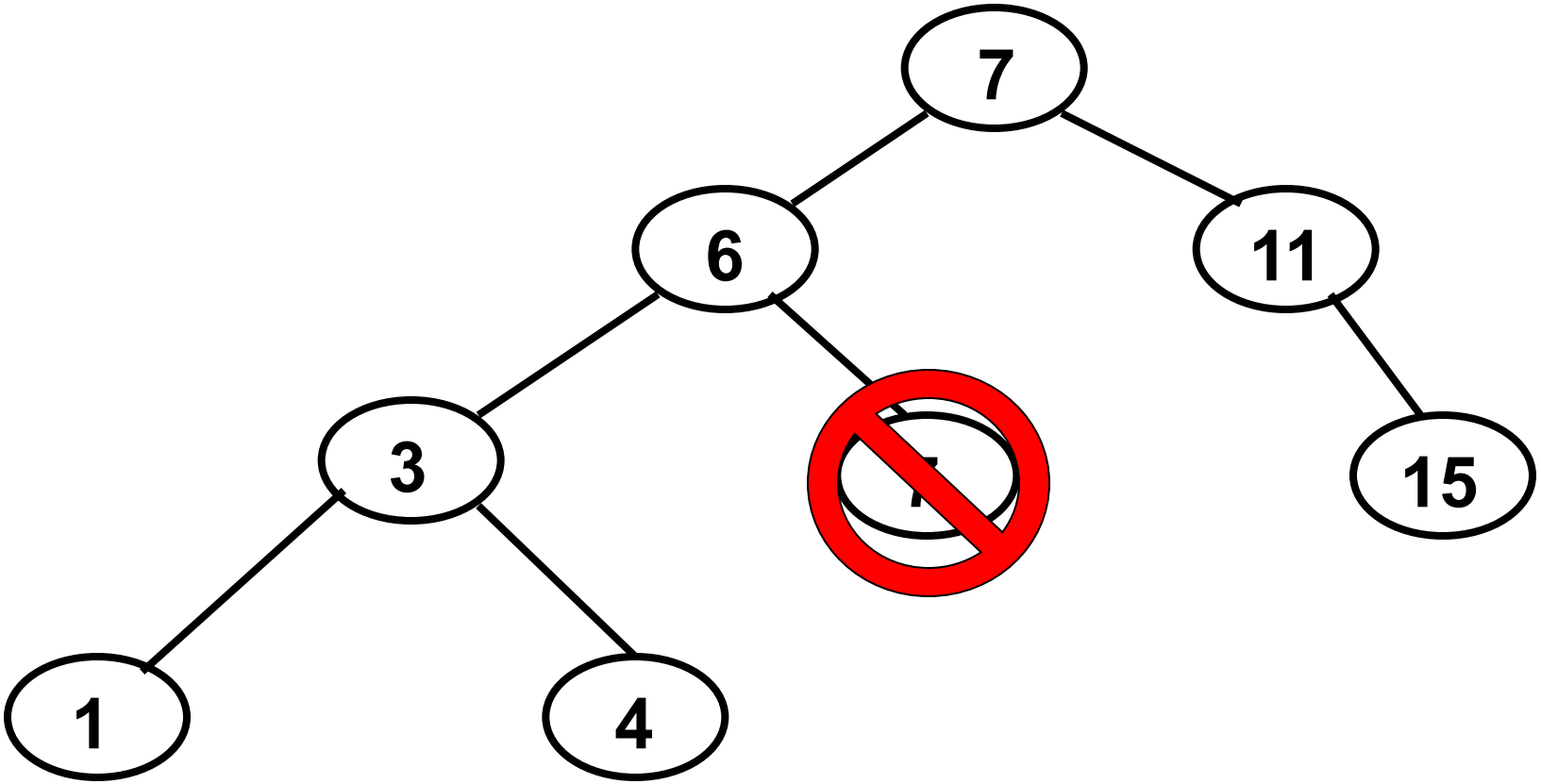
Suppression de 11



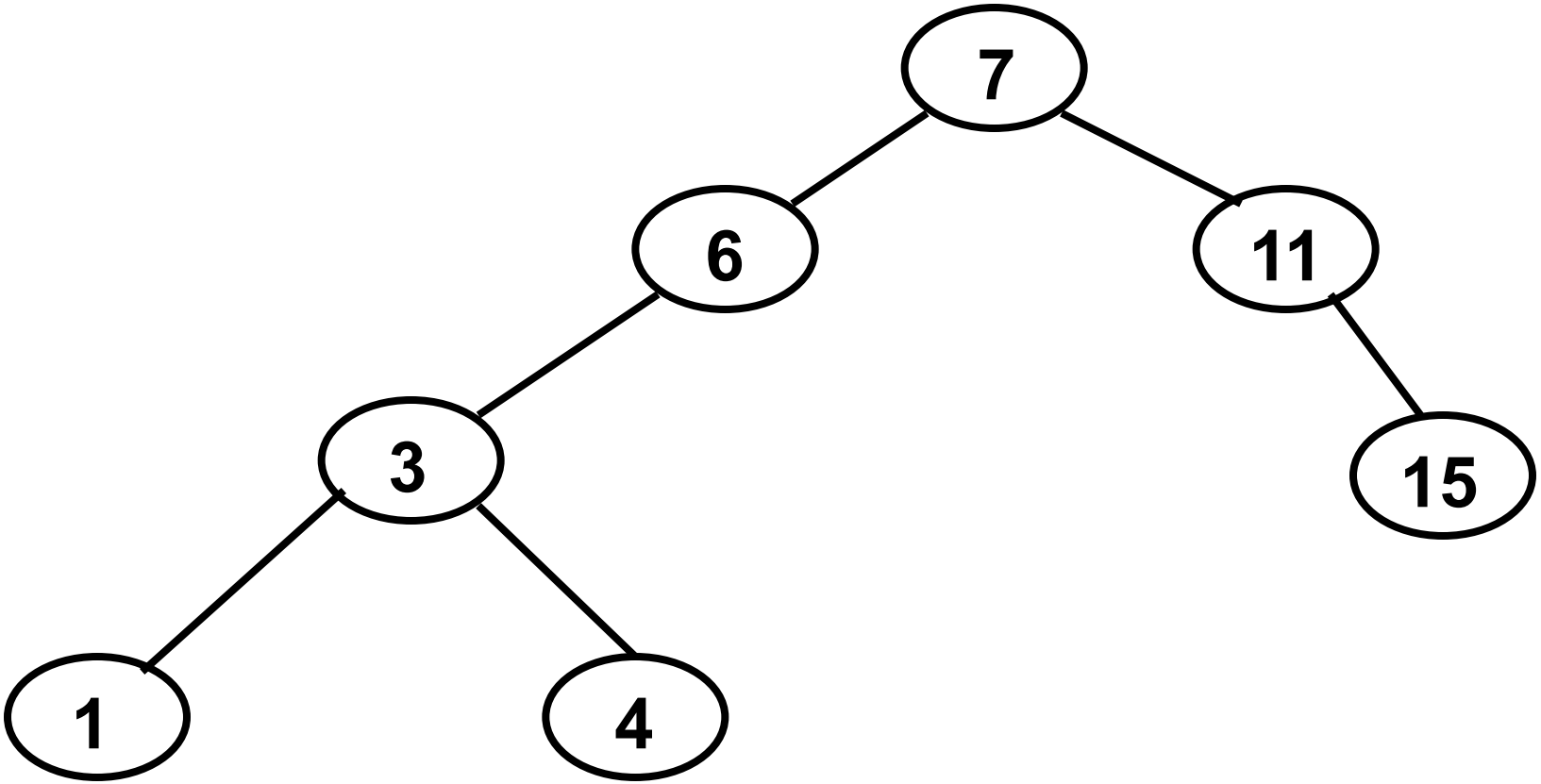
Suppression de 10



Suppression de 10



Suppression de 10



Complexité des ABR

- stockage en $\Theta(n)$
- recherche en $\Theta(\log(n))$
- insertion en $\Theta(\log(n))$
- suppression en $\Theta(\log(n))$
- **Solution au problème de la recherche en table**
- Dans le pire cas, les complexités sont mauvaises → structure d'arbre **équilibré**