Arbres rouge et noir

Irena.Rusu@univ-nantes.fr
LINA, bureau 123, 02.51.12.58.16

Temps d'exécution

opérations sur ensemble

implémentation	Ens_vide	Ajouter Enlever	Élément	Min
table	cst	$O(1)$*	$O(n)$	$O(n)$
table triée	cst	$O(n)$	$O(\log n)$	$O(1)$
liste chaînée	cst	$O(1)$*	$O(n)$	$O(n)$
arbre équilibré	cst	$O(\log n)$	$O(\log n)$	$O(\log n)$
arbre	cst	$O(\log n)$	$O(\log n)$	$O(\log n)$
table de hachage	$O(B)$	cst	cst	$O(B)$

en moyenne

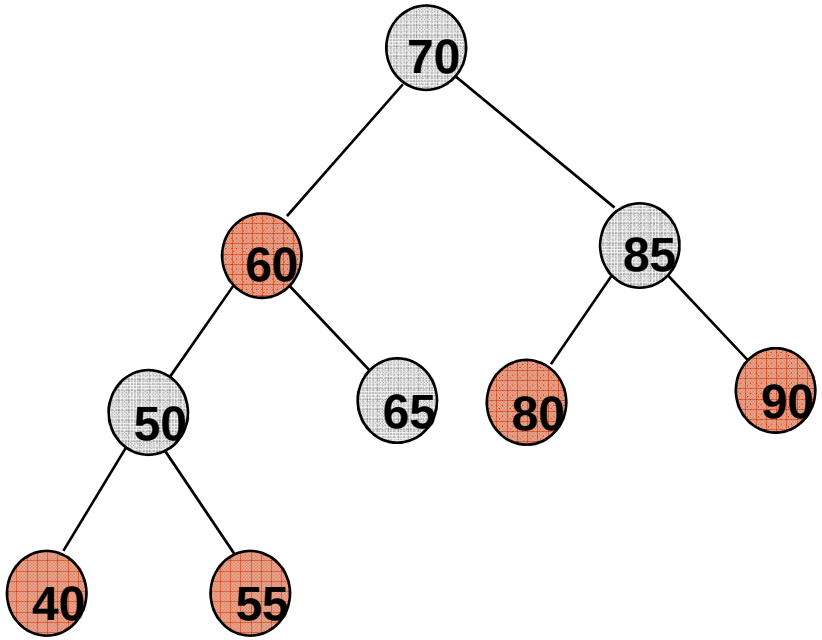
n nombre d 'éléments

$B > n$ taille de la table de hachage

**sans le test d 'appartenance*

Objectifs

- Recherche, insertion, suppression en O(log n), comme dans les AVLs
- Un seul rééquilibrage/opération d'insertion ou suppression
- Pas d'amélioration en O( ), mais un comportement différent (plus ou moins efficace) en fonction du type de données en entrée.



Les figures proviennent de ©http://www.cours.polymtl.ca/inf1101/

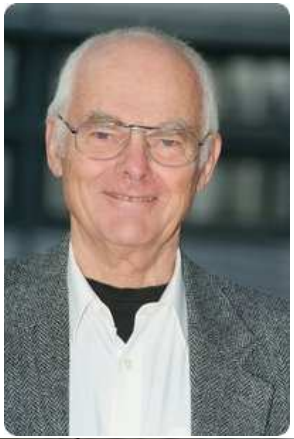
Sommaire

- Arbres rouge et noir
  - Equilibrage
  - Ajout d’un élément
  - Suppression d’un élément
  - Comparaison AVL vs. ARN

Sommaire

- Arbres rouge et noir
  - Equilibrage
  - Ajout d’un élément
  - Suppression d’un élément
  - Comparaison AVL vs. ARN

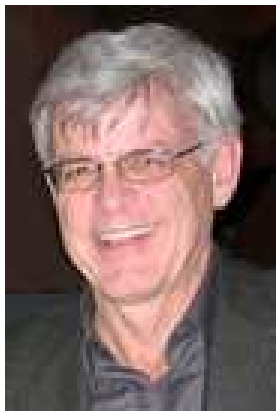
Arbres rouge et noir – Historique



- Auteur : Rudolf Bayer (1939 -)
- Publication en 1972 :

Symmetric Binary B-Trees: Data Structure and Maintenance Algorithms, *Acta informatica* 1: 290-306 (1972)

- Etude détaillée par Leonidas Guibas (1949 -) et Robert Sedgewick (1946 -)



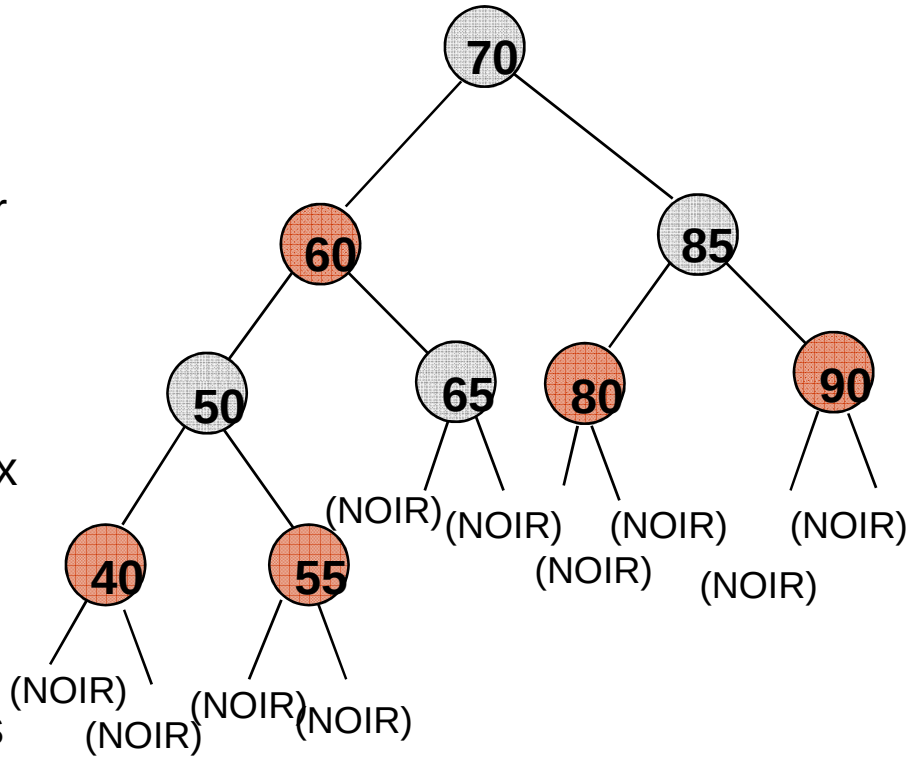
- Publication en 1978 :

A Dichromatic Framework for Balanced Trees, *FOCS,* 1978: 8-21

Arbres rouge et noir (ARN)

A arbre ARN ssi A est un ABR dont les nœuds NIL sont représentés, et en plus (si A non-vide)

- chaque nœud est soit rouge soit noir
- la racine est noire
- chaque noeud NIL est noir
- si un nœud est rouge, alors ses deux fils sont noirs
- tous les chemins reliant un nœud à un NIL ont le même nombre de nœuds noirs.



Arbres rouge et noir (ARN)

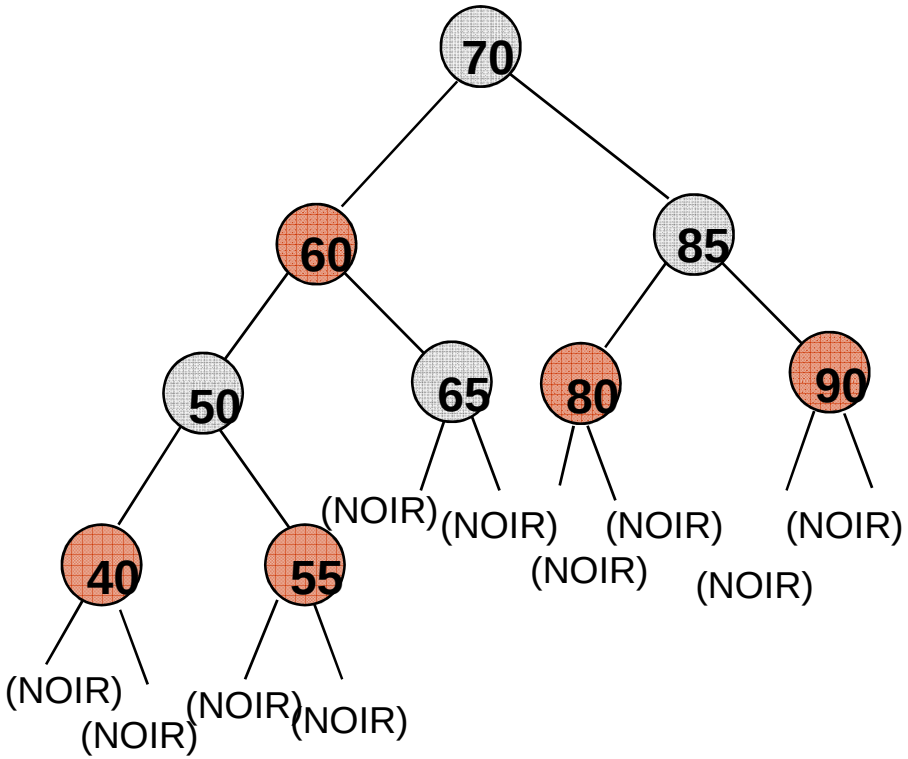
Hauteur noire

hn(x) : le nombre de nœuds noirs sur tout chemin de x à un NIL, sauf x.

hn(A)=hn(r), où $A=(r, A_g, A_d)$

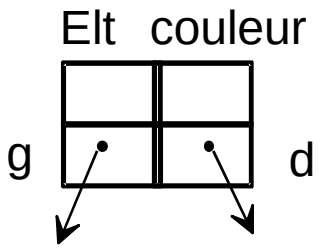
Remarque La longueur d'un chemin de la racine à une feuille (non-NIL) est au plus 2*hn(A), car :

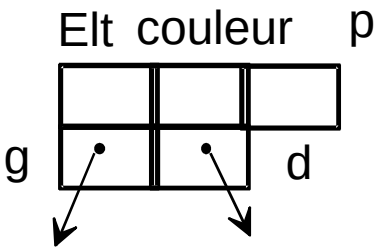
- la racine est noire
- il n'y a pas deux nœuds rouges consécutifs



Implémentation des ARNs

arn: structure, comme les AVL,
sauf 1) ajouter couleur : **et parfois même 2) ajouter parent p :**





Remarque. Lors de l'implémentation, un noeud NIL a les mêmes champs que les autres nœuds, et la couleur noire

Fonctions :

(arn) ARNajouter (element x, arn A) ;
/* rend l'arbre modifié*/

(arn) ARNenlever (element x, arn A) ;
/* rend l'arbre modifié */

Hauteur d'ARN

A **arbre ARN à *n* nœuds**

$$\log_2(n+1)-1 \leq h(A) \leq 2\log_2(n+1)$$

$\Rightarrow$ **Implémentation d'ensembles avec opérations :**

MIN (*A*), MAX (*A*)

AJOUTER (*x*, *A*)

ENLEVER (*x*, *A*)

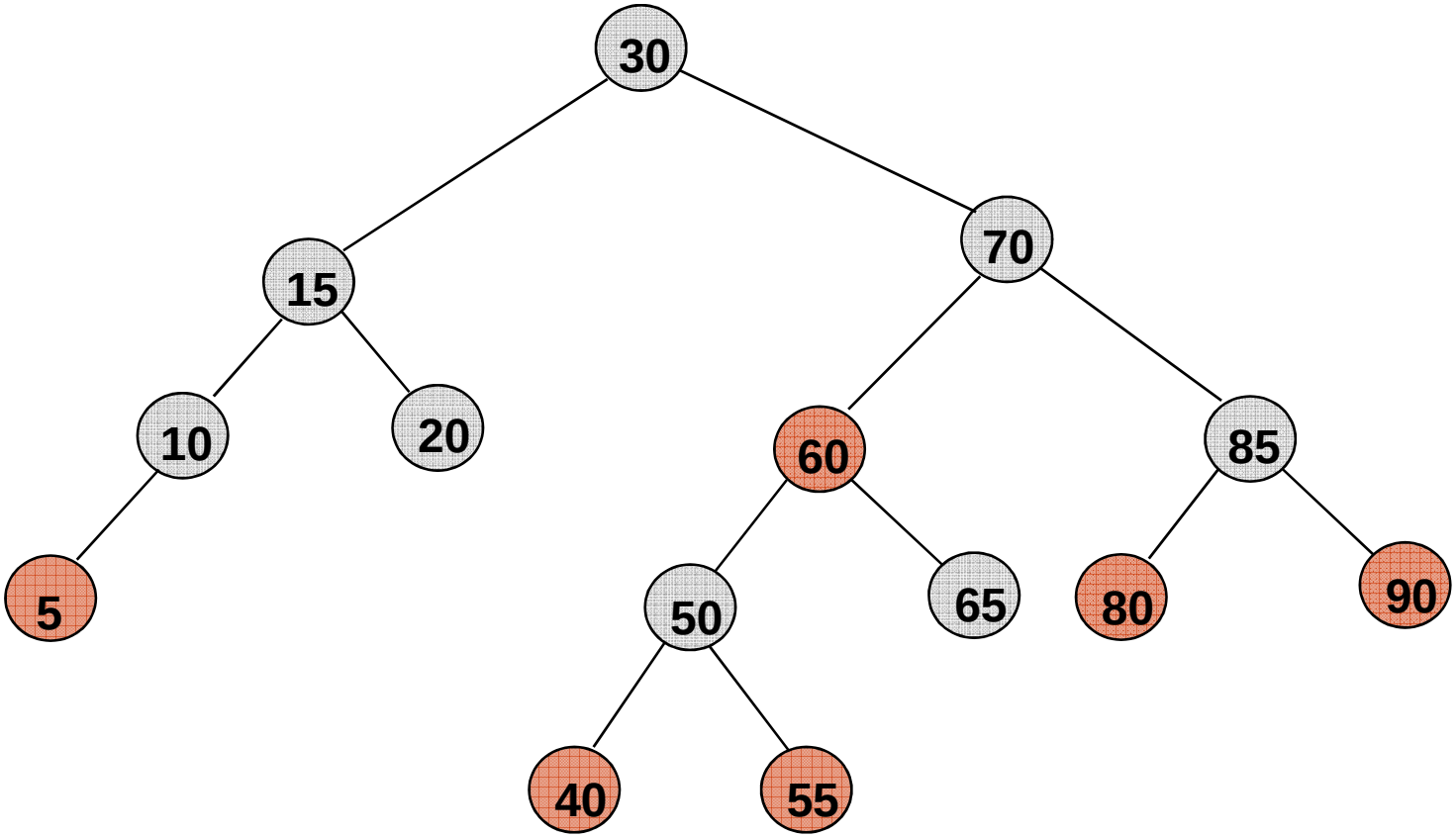
ELEMENT (*x*, *A*)

} **O(log(*n*))**
pire des cas

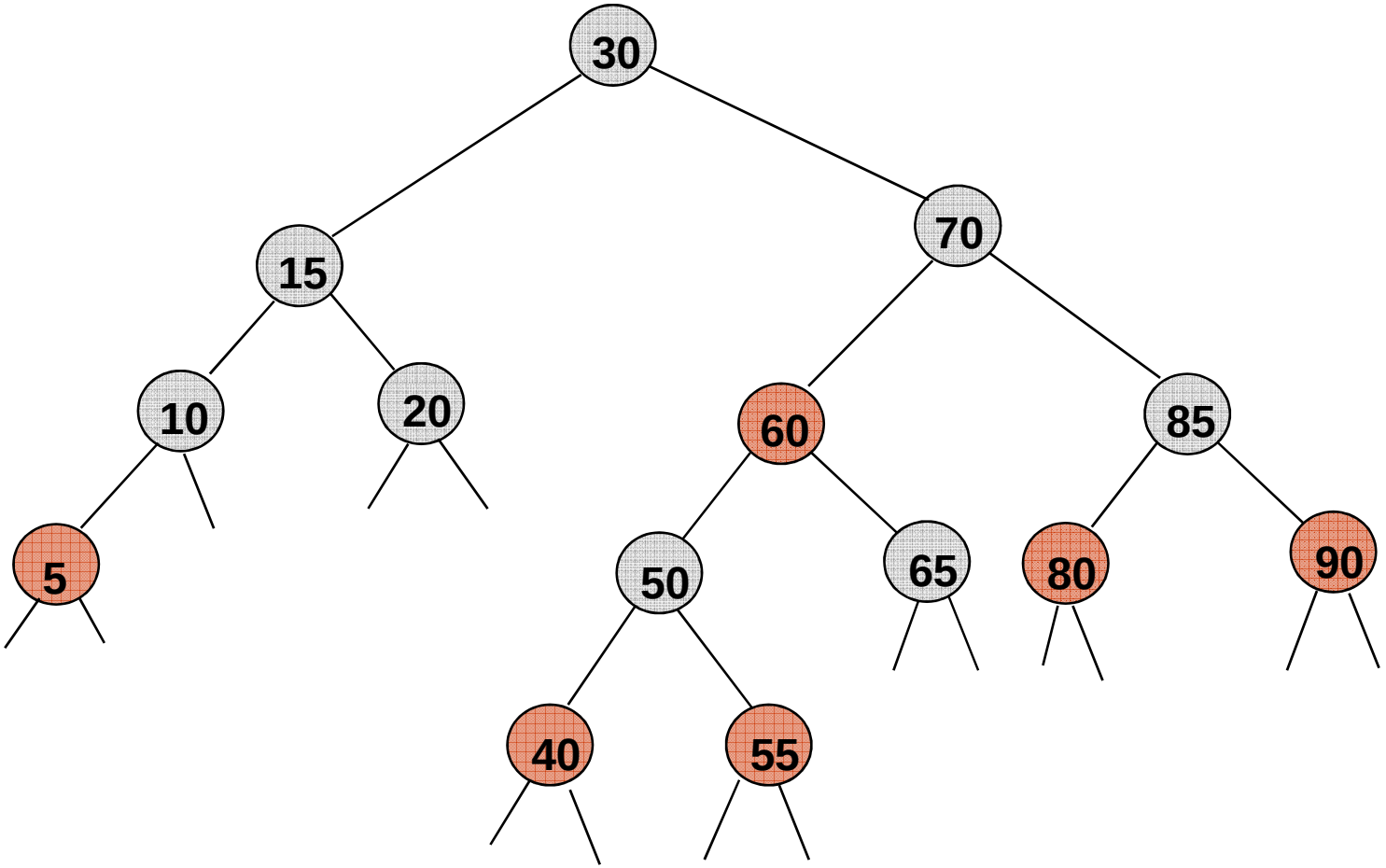
Sommaire

- Arbres rouge et noir
  - Equilibrage
  - Ajout d’un élément
  - Suppression d’un élément
  - Comparaison AVL vs. ARN

Effets des ajouts/suppressions

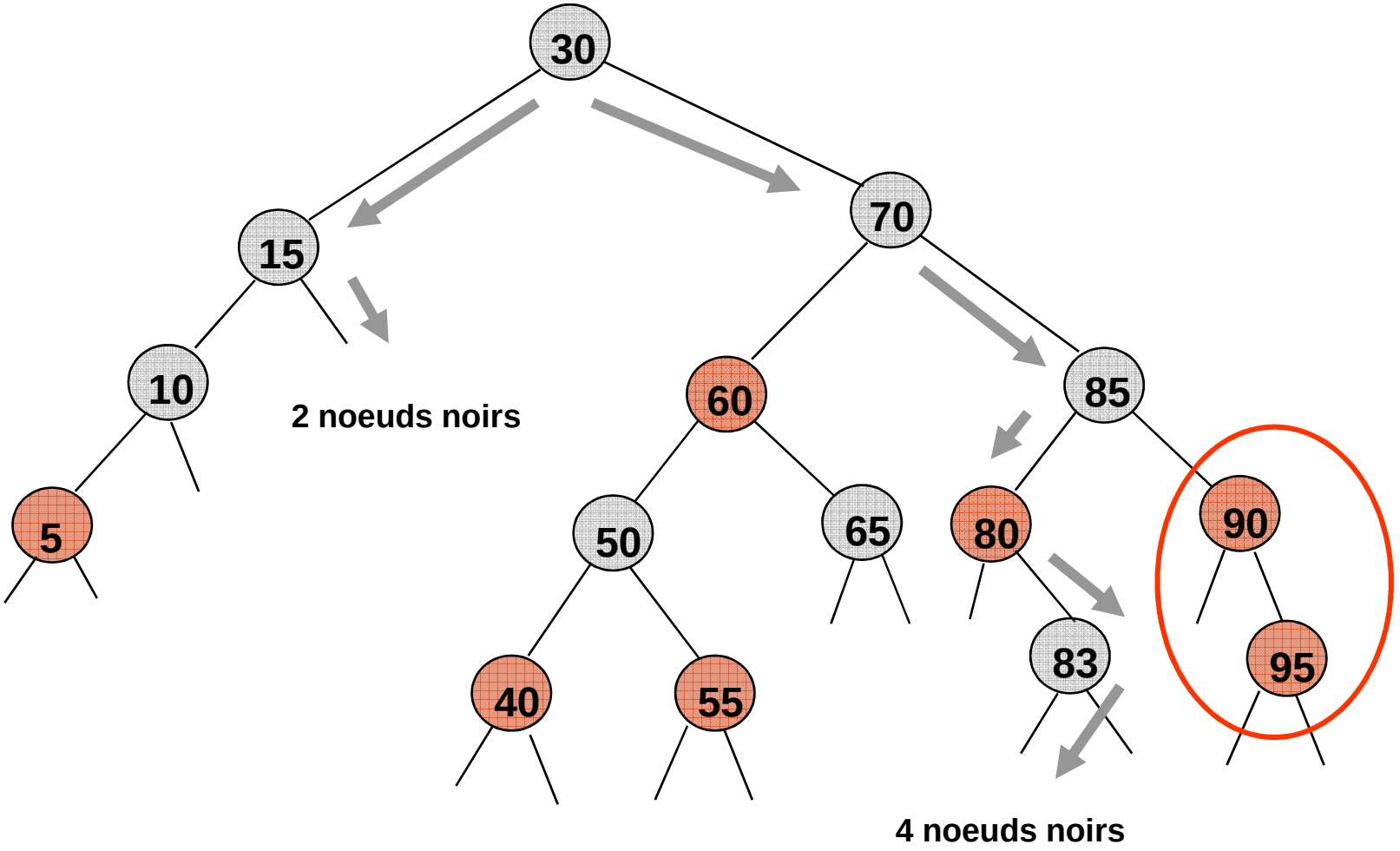


Effets des ajouts/suppressions



Note. Ne pas oublier les NIL, même s'ils ne sont pas dessinés

Suppression de 20, ajouts de 83 et 95

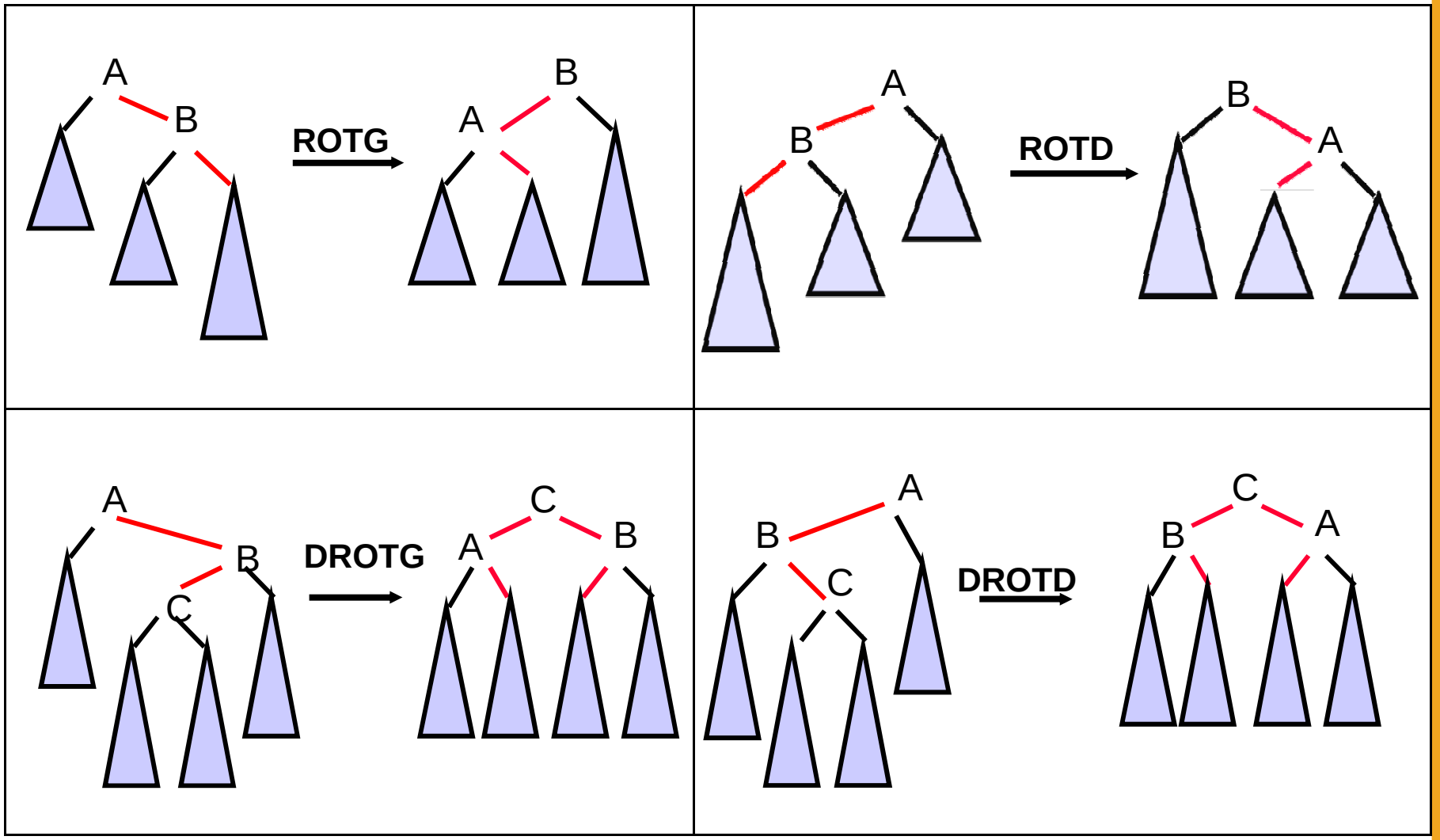


Problèmes et solutions

- **Problèmes :**
  - Pas assez de nœuds noirs entre la racine et NIL (suppression)
  - Trop de nœuds noirs entre la racine et NIL (ajout)
  - Des nœuds rouges consécutifs (suppression, ajout)

- **Solutions :**
  - Rotations gauche et droite
  - Re-coloriage de noeuds

Equilibrage : 4 solutions (à choisir convenablement)

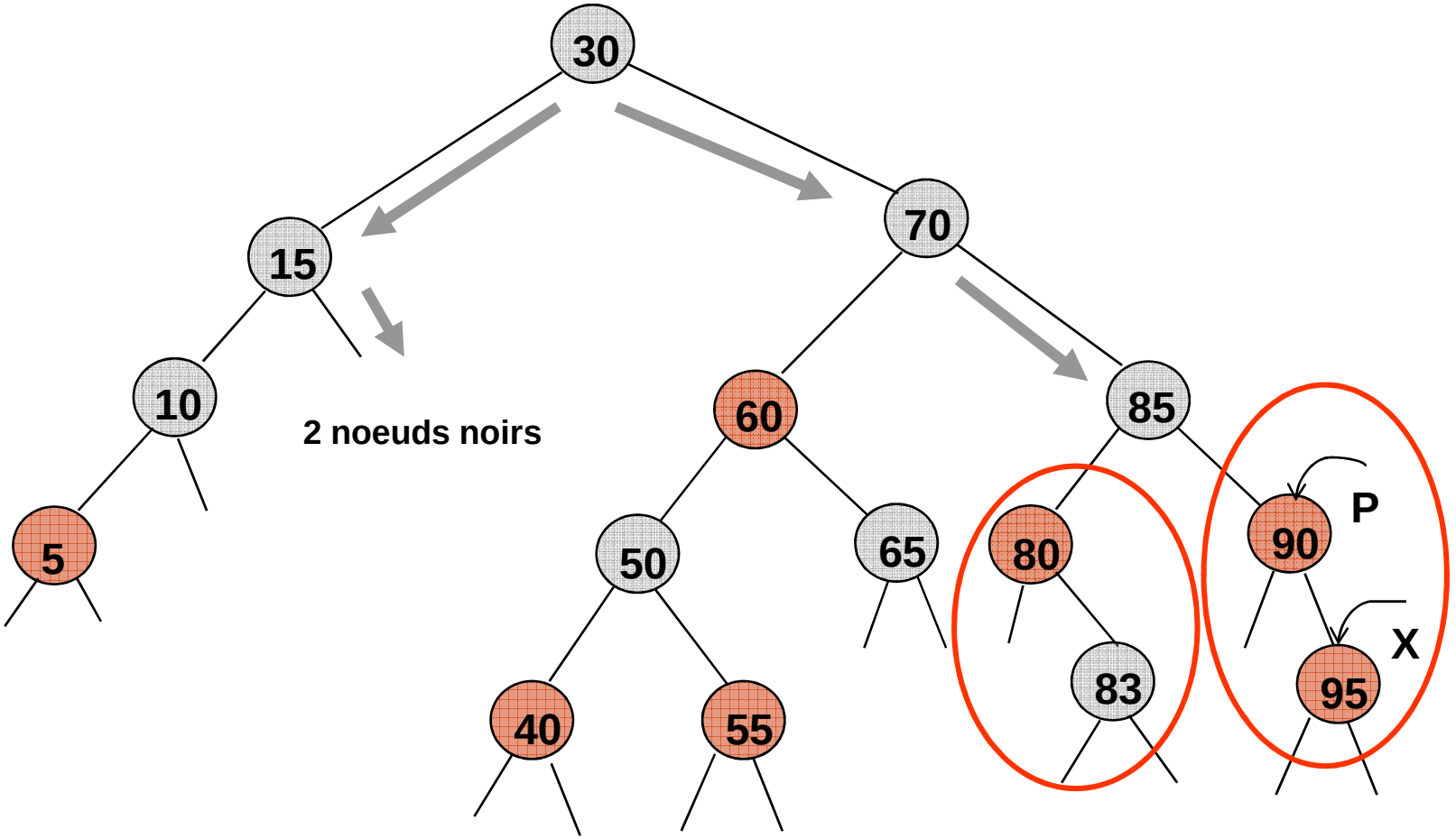


Note : Implémentation similaire aux AVLs (sauf type des nœuds, spécifique aux ARN)

Sommaire

- Arbres binaires équilibrés en hauteur (ou AVL)
  - Equilibrage
  - Ajout d'un élément
  - Suppression d'un élément
  - Comparaison AVL vs. ARN

Toujours ajouter un nœud rouge
(rappel : les NIL sont tous noirs)

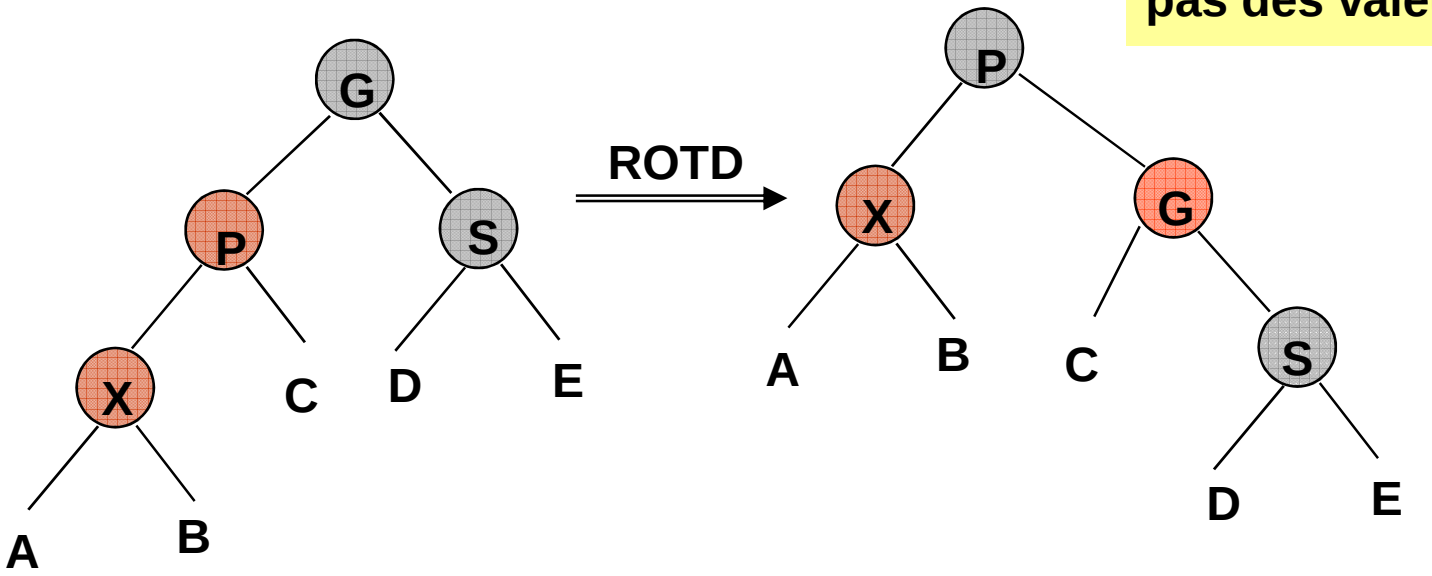


83 sera colorié en rouge, et ce sera le même cas que pour 95

Résoudre le cas X, P rouges

- **Cas 1** : le frère S de P est noir

Note. **Ici et par la suite, les lettres sont des pointeurs, pas des valeurs**



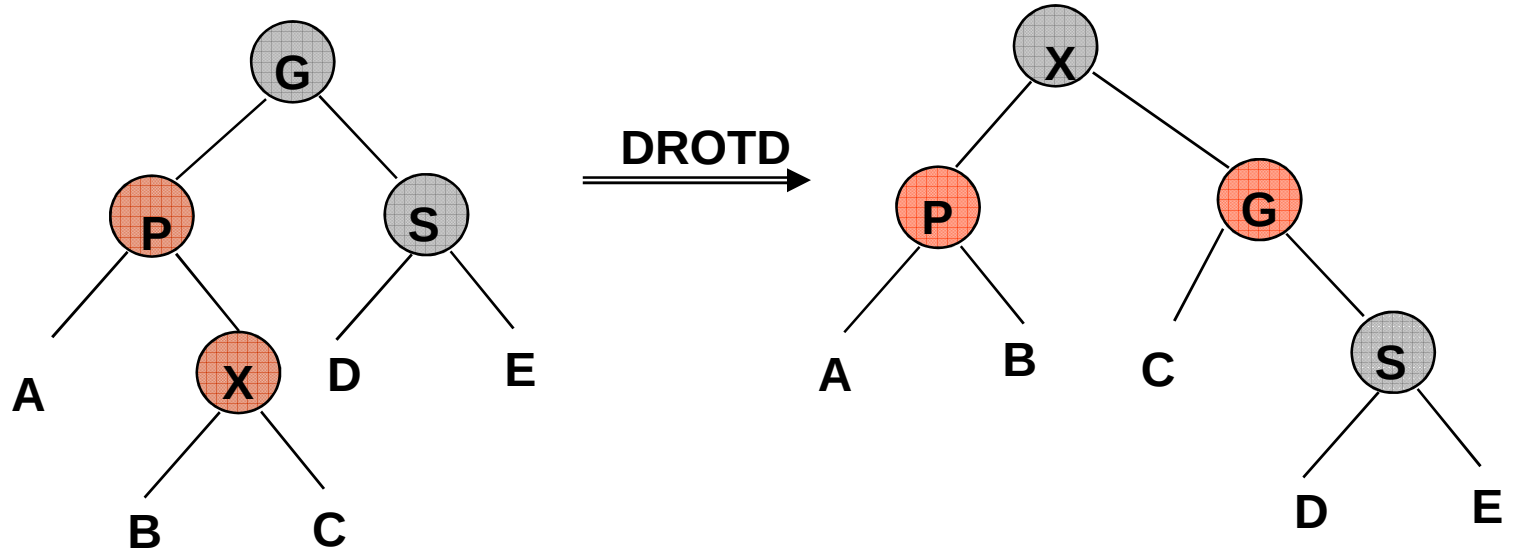
Si X, P sont tous les deux des fils gauches, alors rotation droite

Recoloriage : les deux racines G, P inversent leurs couleurs

Observer que le nombre de nœuds noirs sur chaque chemin est gardé

Résoudre le cas X, P rouges

- **Cas 1** : le frère S de P est noir

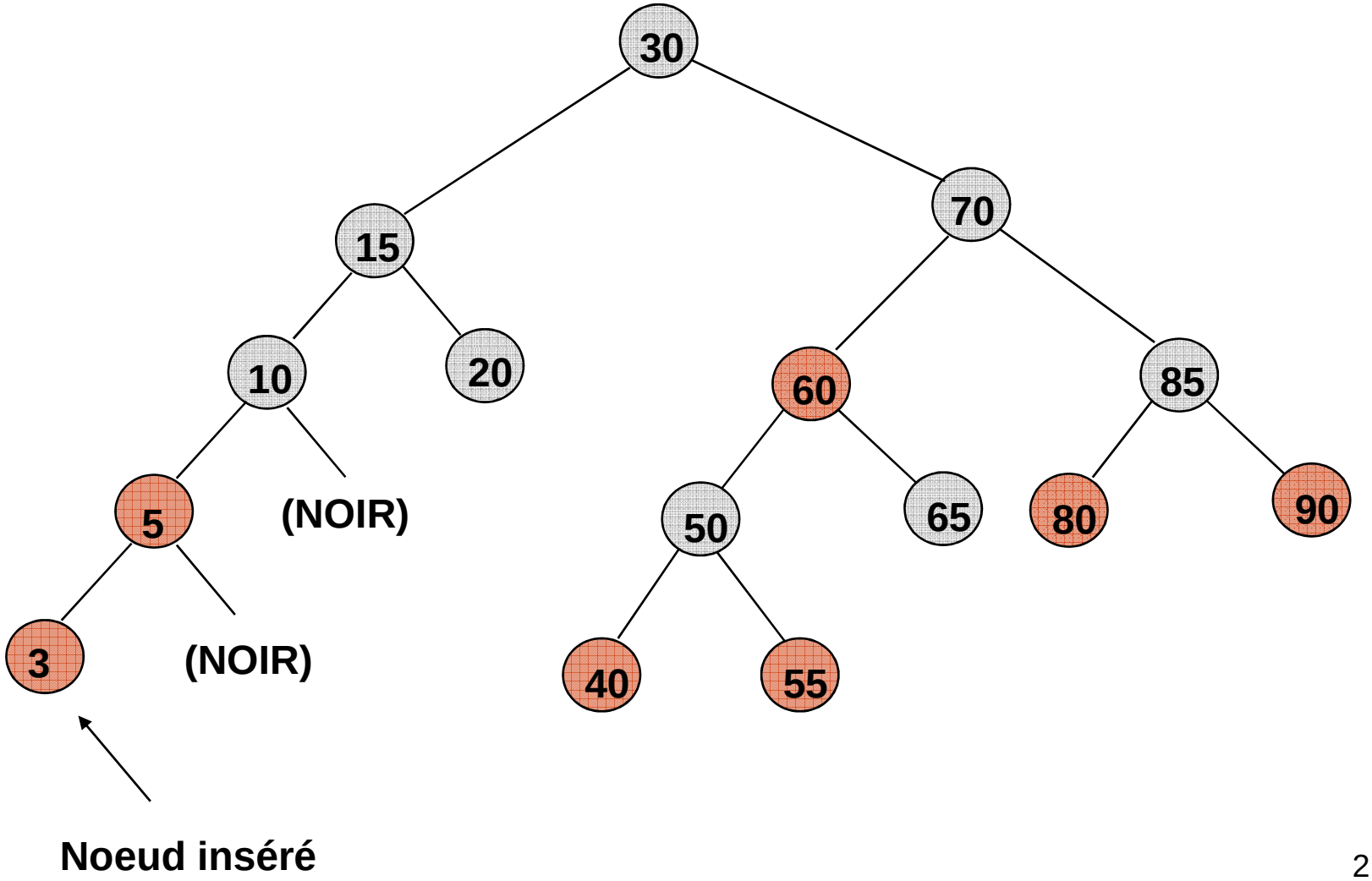


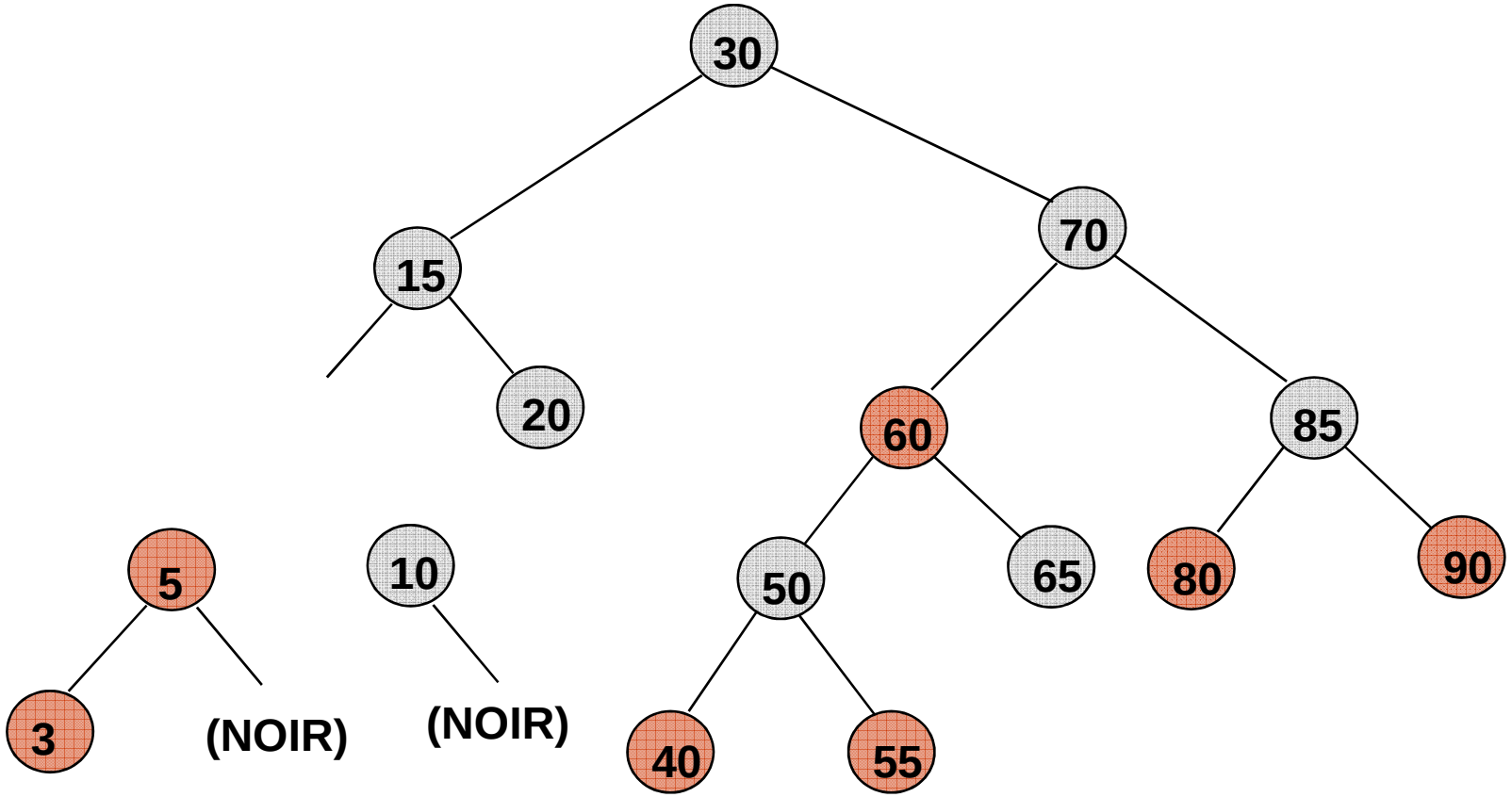
Si P est fils gauche, et X fils droit, alors double rotation

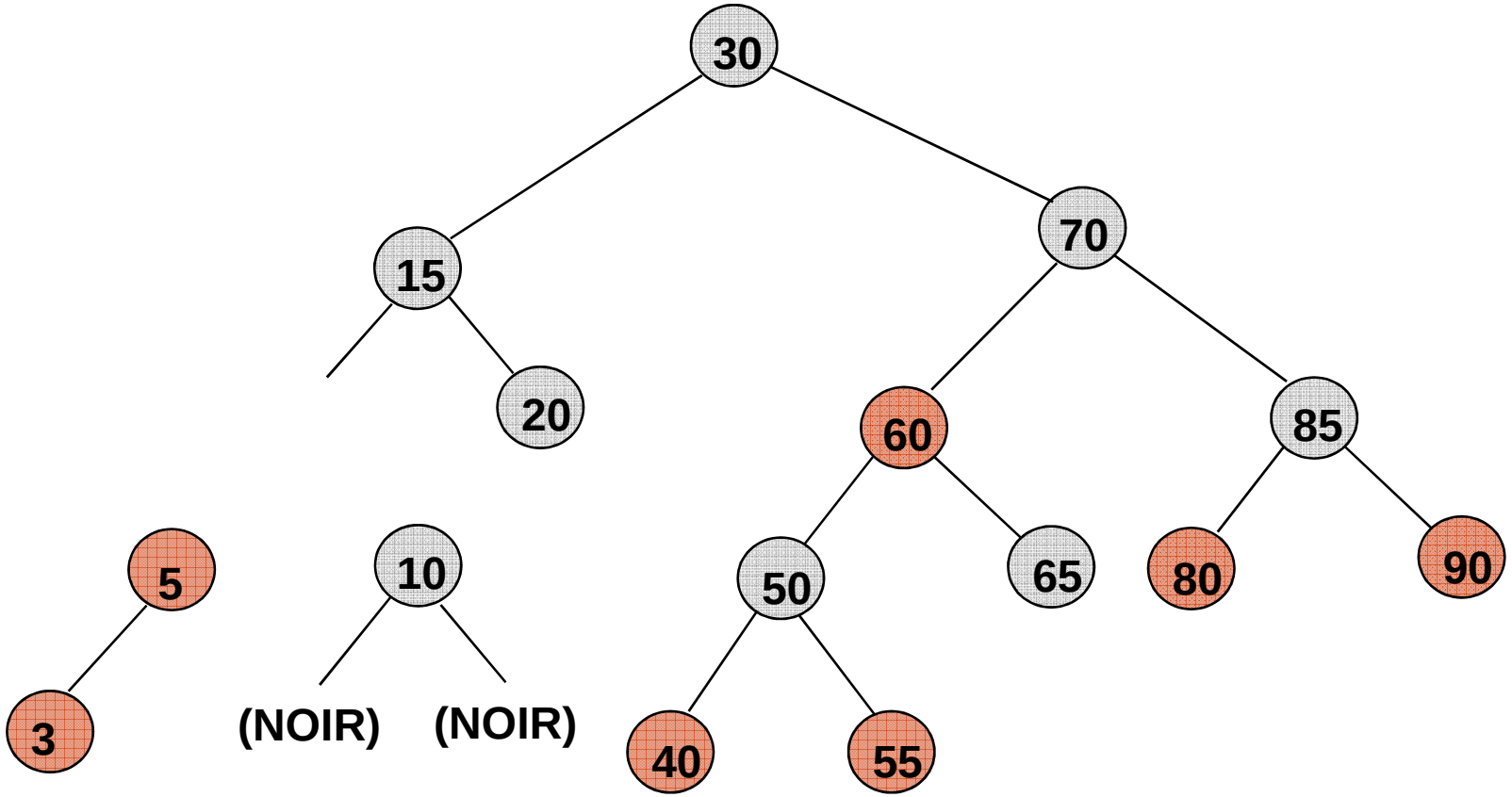
Inverser les couleurs des deux racines G et X

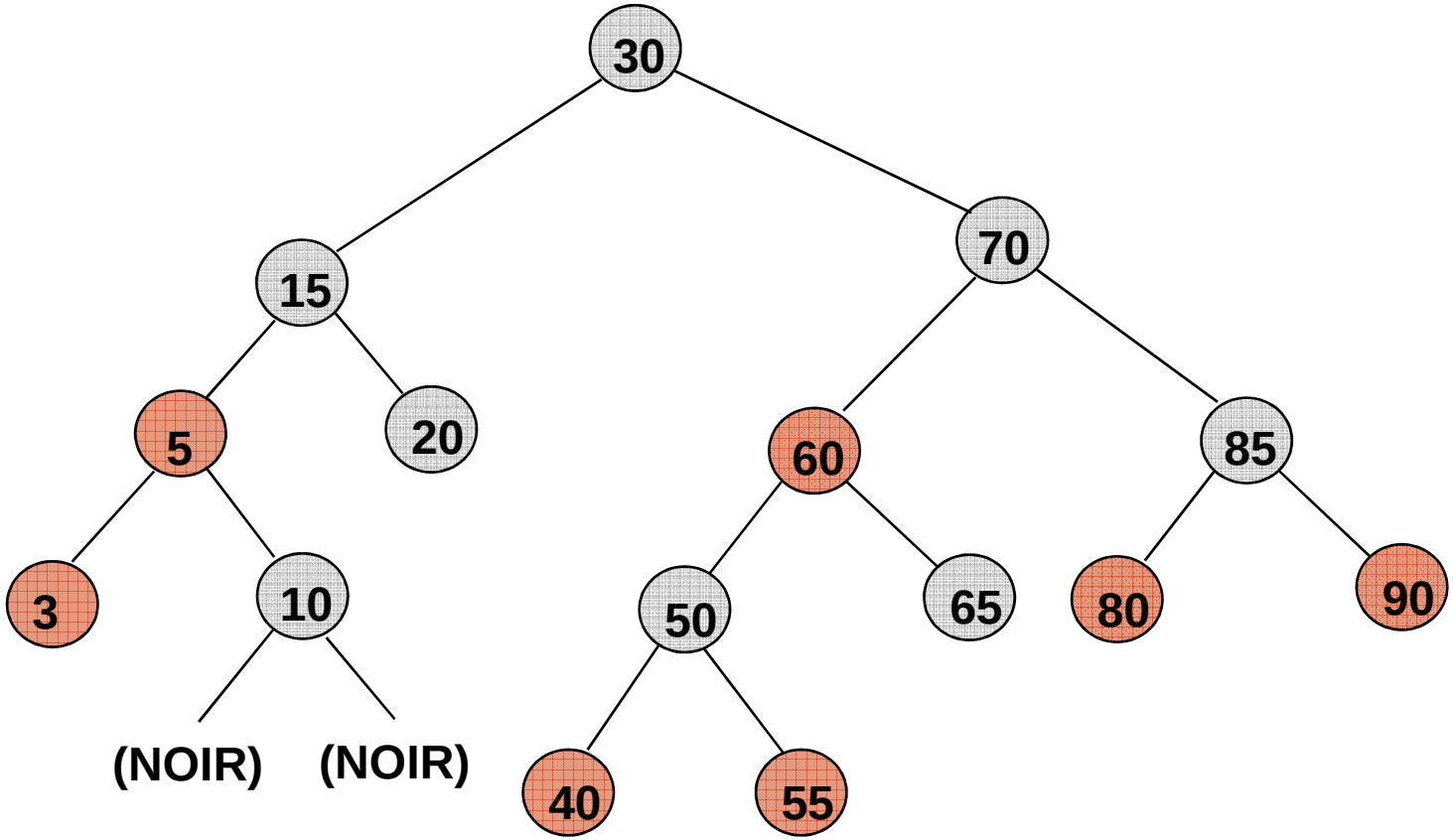
Observer que le nombre de nœuds noirs sur chaque chemin est gardé

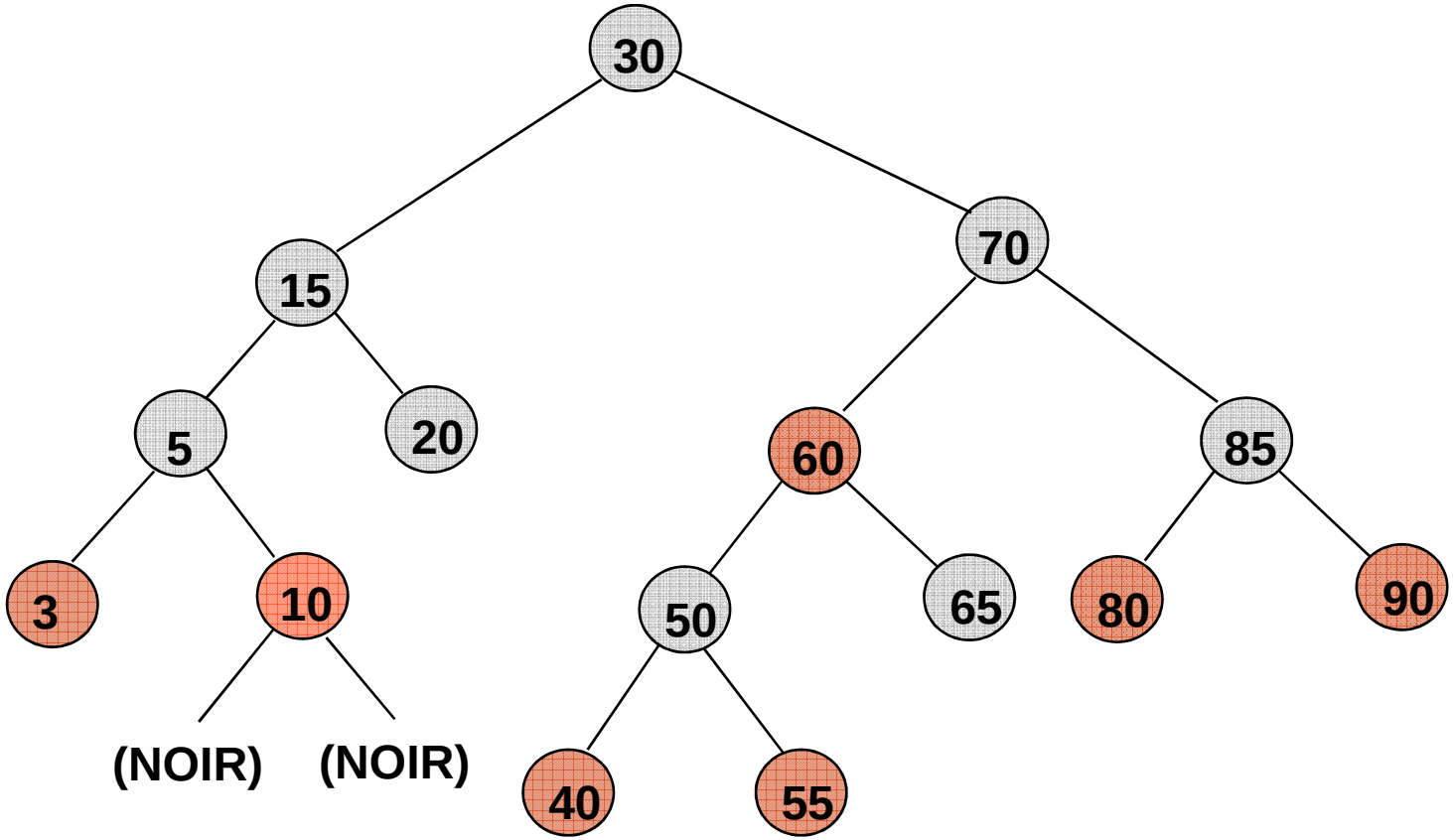
Exemple

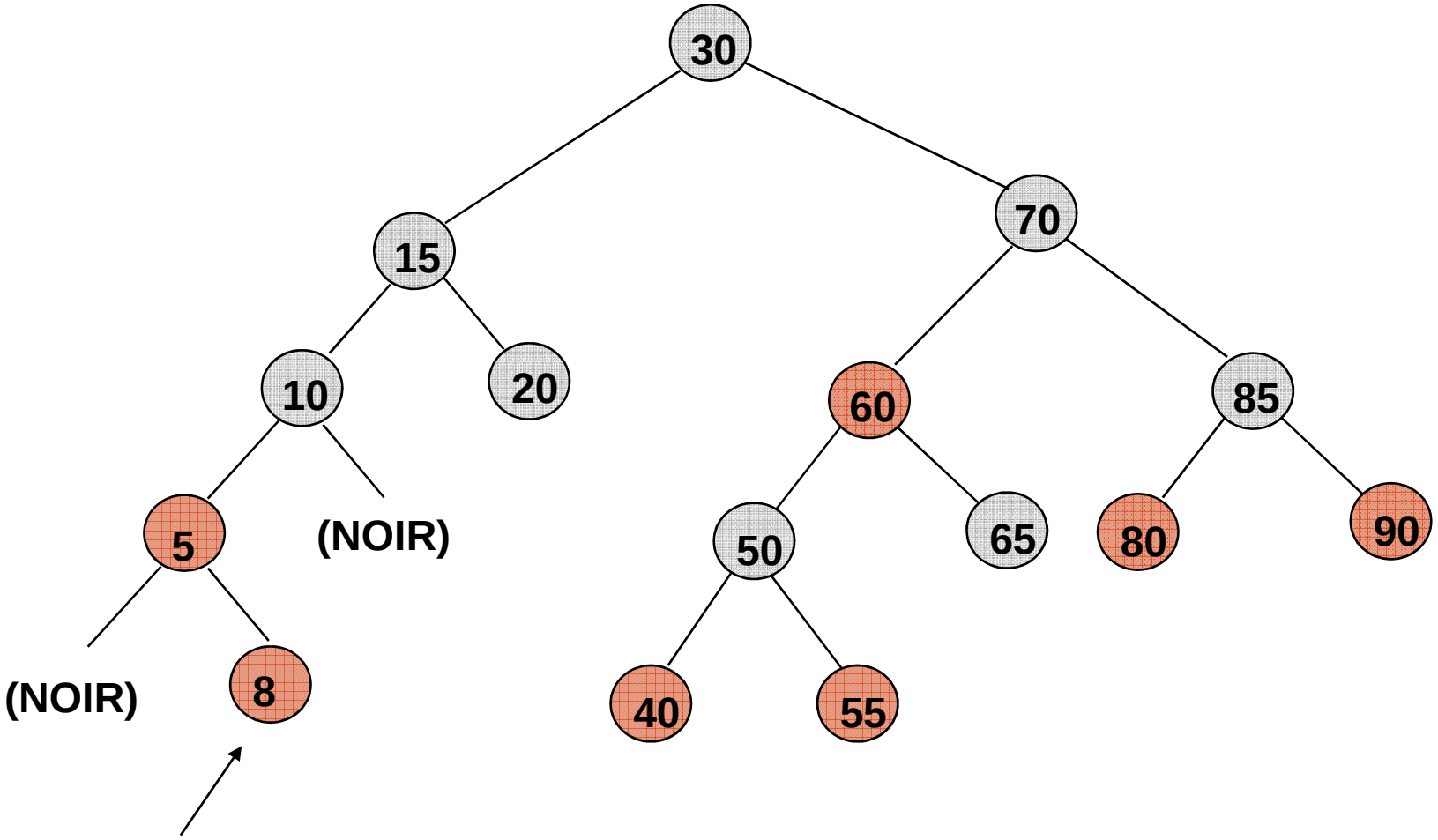


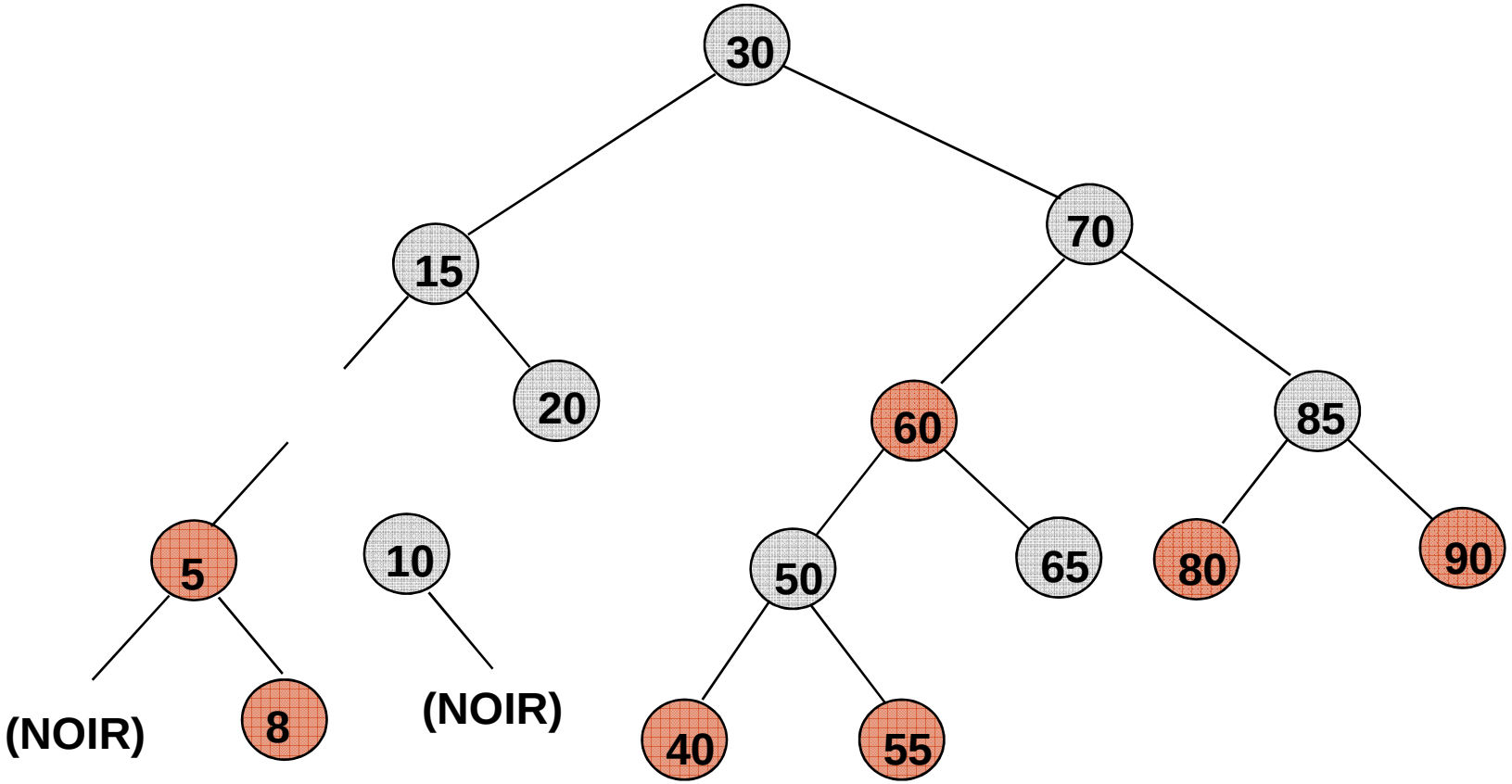

30
15
70
20
60
85
5
10
50
65
80
90
3
(NOIR)
(NOIR)
40
55

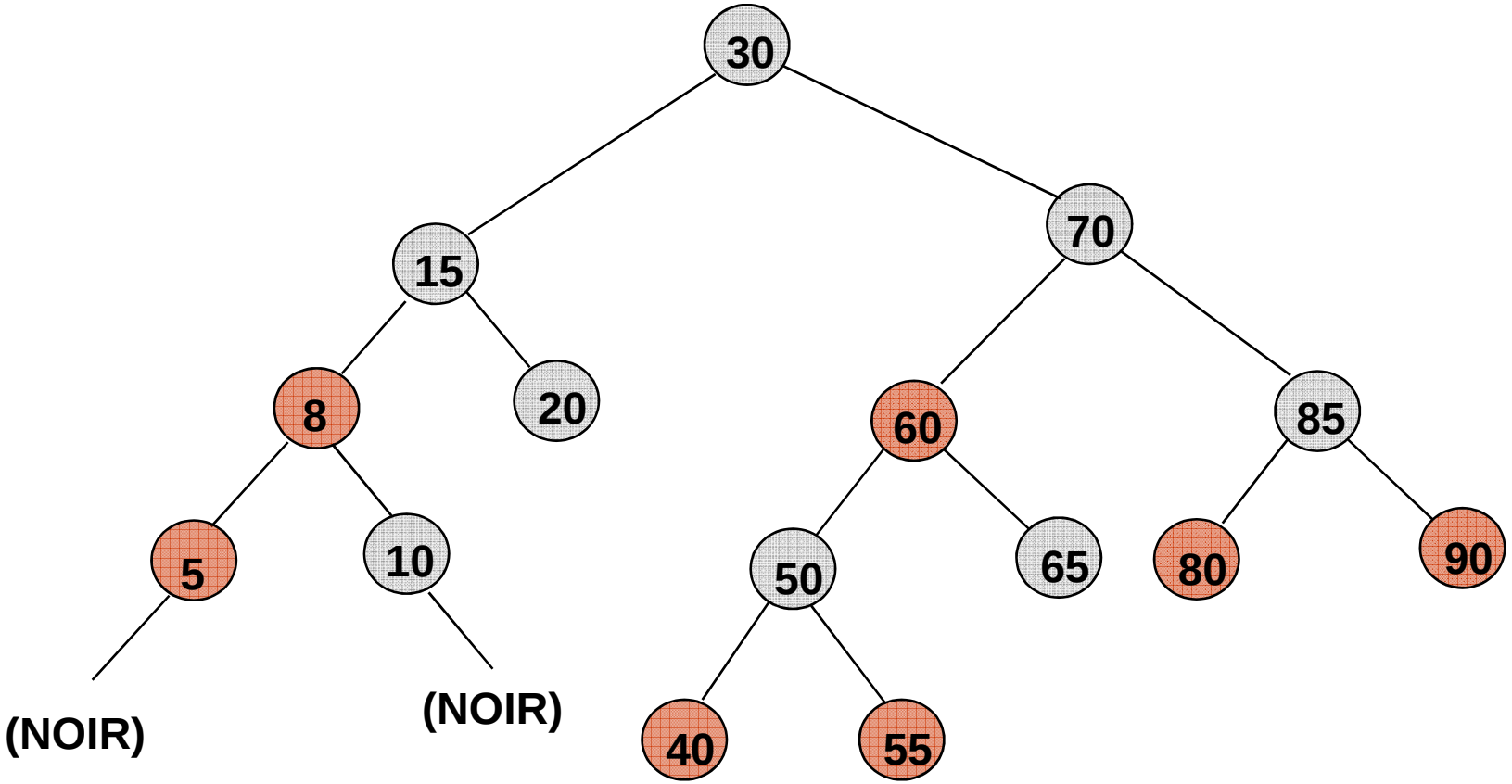

30
15
70
20
60
85
5
10
50
65
80
90
3
(NOIR)
(NOIR)
40
55

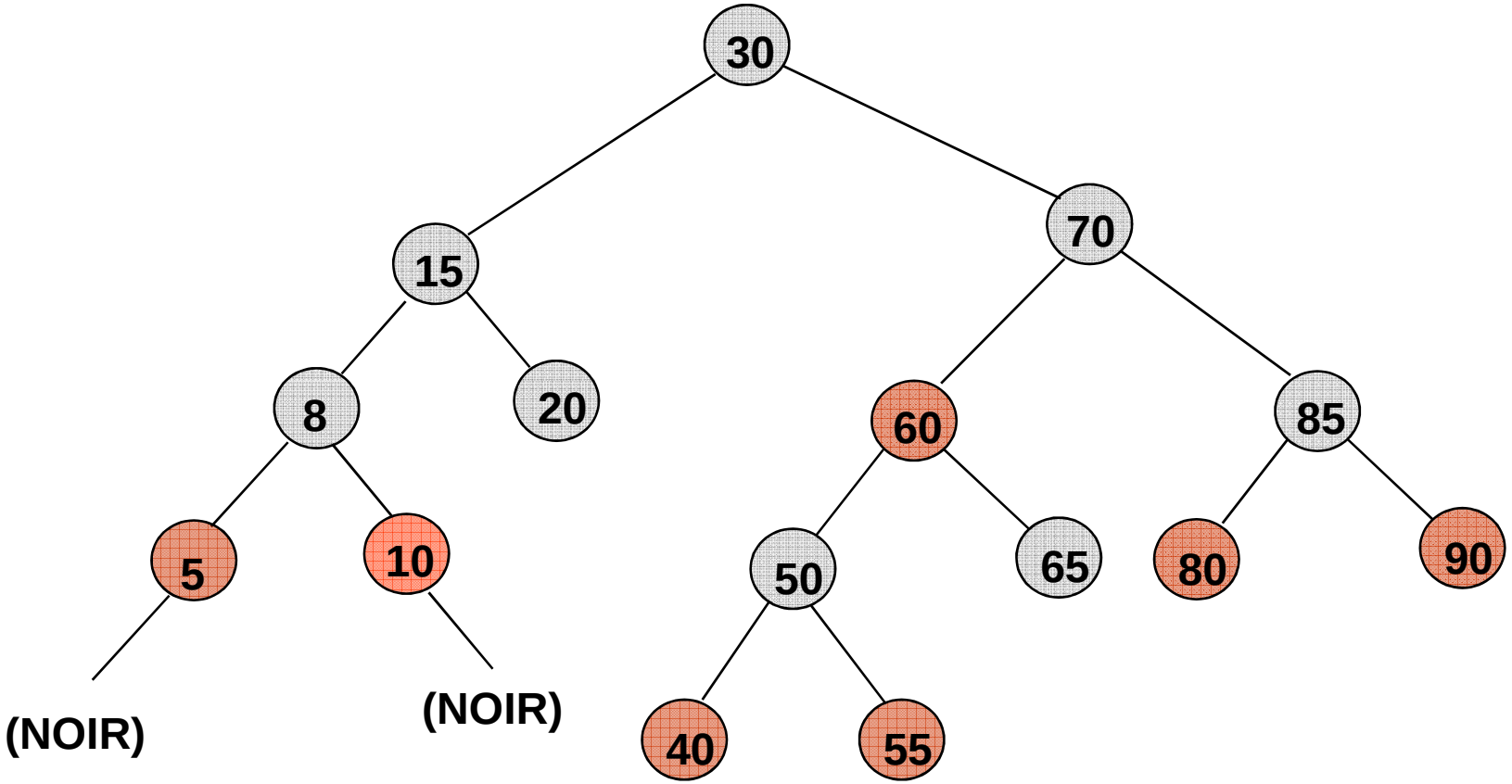

30
15
70
5
20
60
85
3
10
50
65
80
90
(NOIR)
(NOIR)
40
55


30
15
70
5
20
60
85
3
10
50
65
80
90
(NOIR)
(NOIR)
40
55


30
15
70
10
20
60
85
5
(NOIR)
50
65
80
90
(NOIR)
8
40
55
Noeud inséré

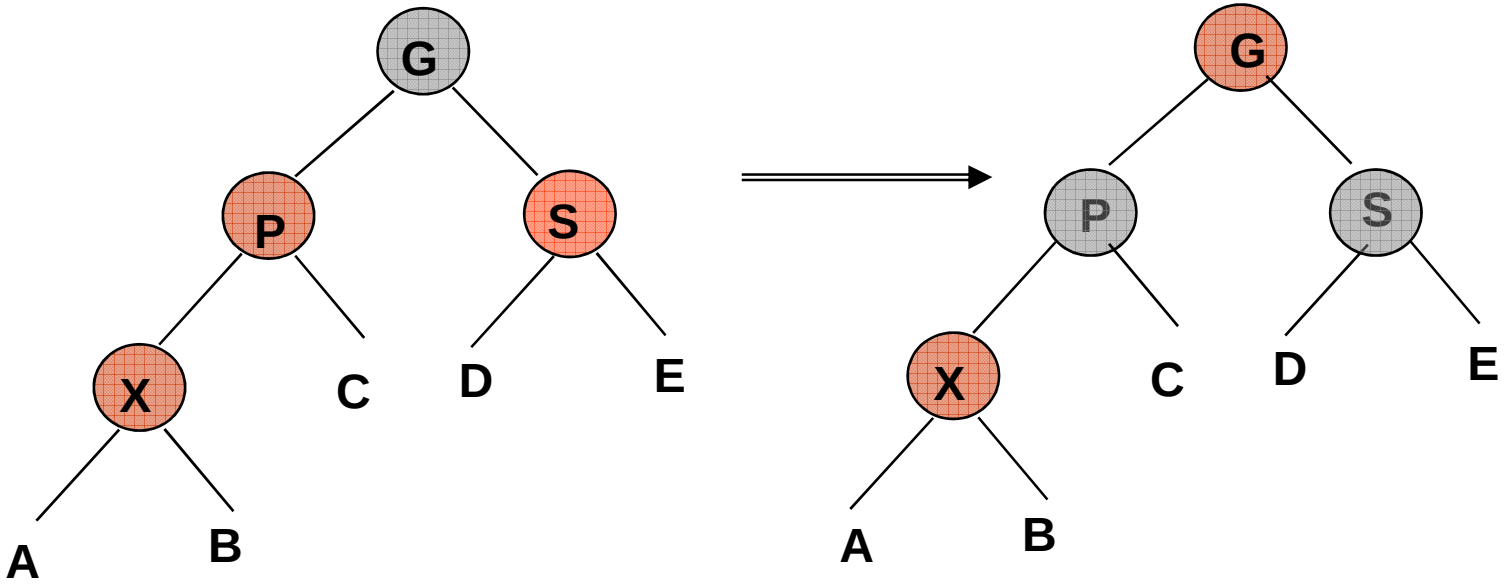

30
15
70
20
60
85
5
10
50
65
80
90
(NOIR)
8
(NOIR)
40
55


30
15
70
8
20
60
85
5
10
50
65
80
90
(NOIR)
(NOIR)
40
55


30
15
70
8
20
60
85
5
10
50
65
80
90
(NOIR)
(NOIR)
40
55

Résoudre le cas X, P rouges

- **Cas 2** : le frère S de P est rouge



La re-coloration met la couleur rouge au sommet et – si le père de G était rouge – remonte le problème vers le haut.

Algorithme dit d'insertion « bottom-up »

```
arn ARNajouter(element x, arn A)
(A,X) ← ajouter(x,A) // insertion comme dans ABR, X pointeur sur x
si (X.p ≠ arbre vide) faire
    X.couleur ← ROUGE
    tant que ((X.p).couleur = ROUGE) faire
        si (X.p est le fils gauche de son père) alors
            S ← X.p.p.d ;
            si (S.couleur = NOIR) alors  TraiterCas1(S)
            sinon   TraiterCas2(S)    finsi
        sinon (idem avec permutation de « droite » et « gauche ») finsi
    fin tantque
finsi
A.couleur ← NOIR
```

Remontée vers la racine : pour ou contre ?

- POUR :
  - La propagation vers la racine se terminera forcément à un moment donné, puisque la racine est noire.
- CONTRE :
  - La propagation peut nécessiter autant de rotations que la hauteur de l'arbre, et on n'atteint pas l'objectif 1 rotation/opération.
  - Elle nécessite l'ajout du pointeur parent, qui prend de la place dans la structure de données.

Temps pour un ajout « bottom-up »

Temps d'une rotation : constant

Note : ARNajouter exécute au plus h(A) rotations
car on remonte d'une feuille vers la racine

A arbre ARN à *n* nœuds

Temps total d'un ajout : $O(h(A)) = O(\log n)$

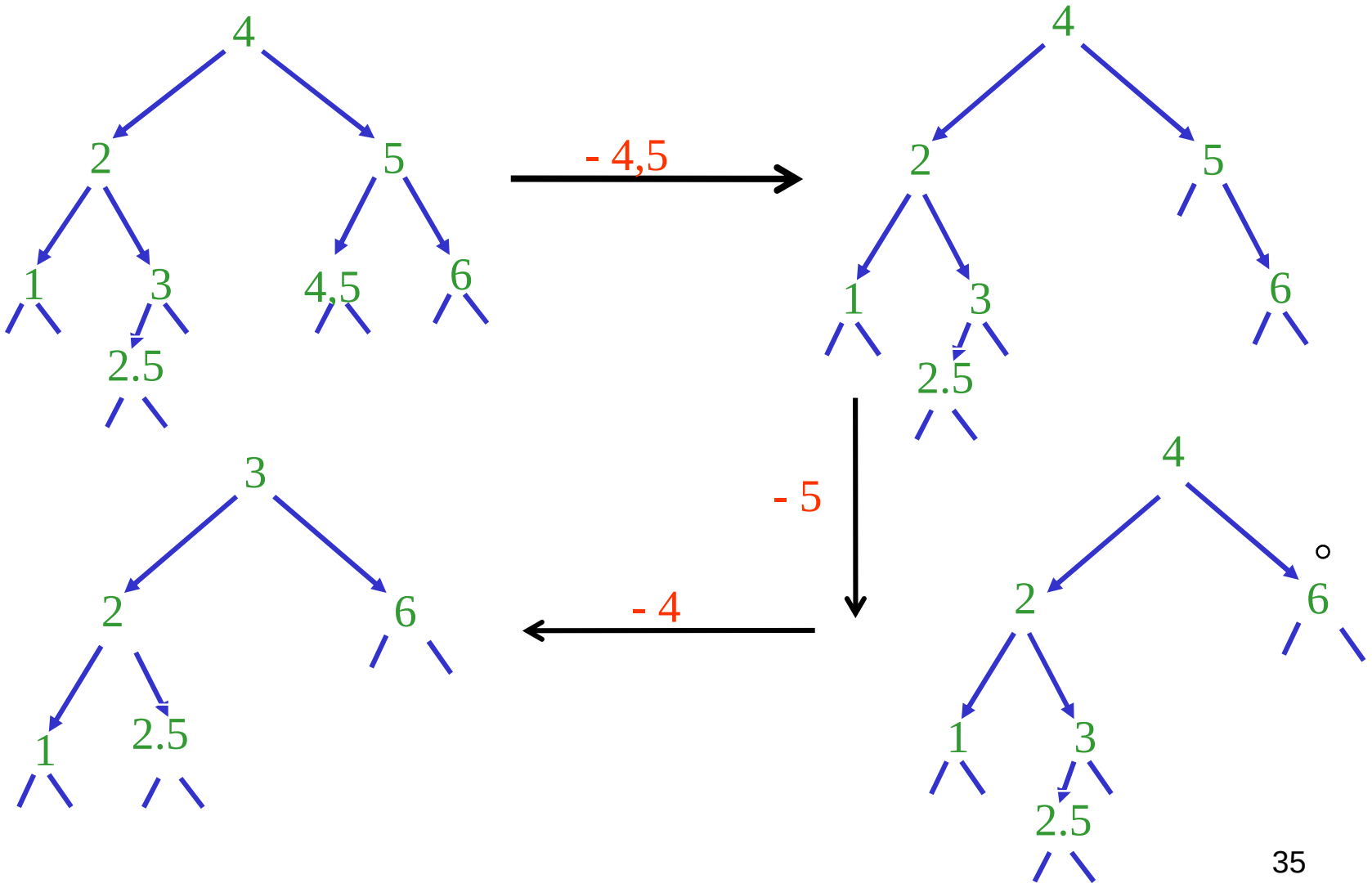
car une seule branche de l'arbre est examinée.

Complexité bonne, mais n'améliore en rien les AVLs

Sommaire

- Arbres binaires équilibrés en hauteur (ou AVL)
  - Equilibrage
  - Ajout d’un élément
  - Suppression d’un élément
  - Comparaison AVL vs. ARN

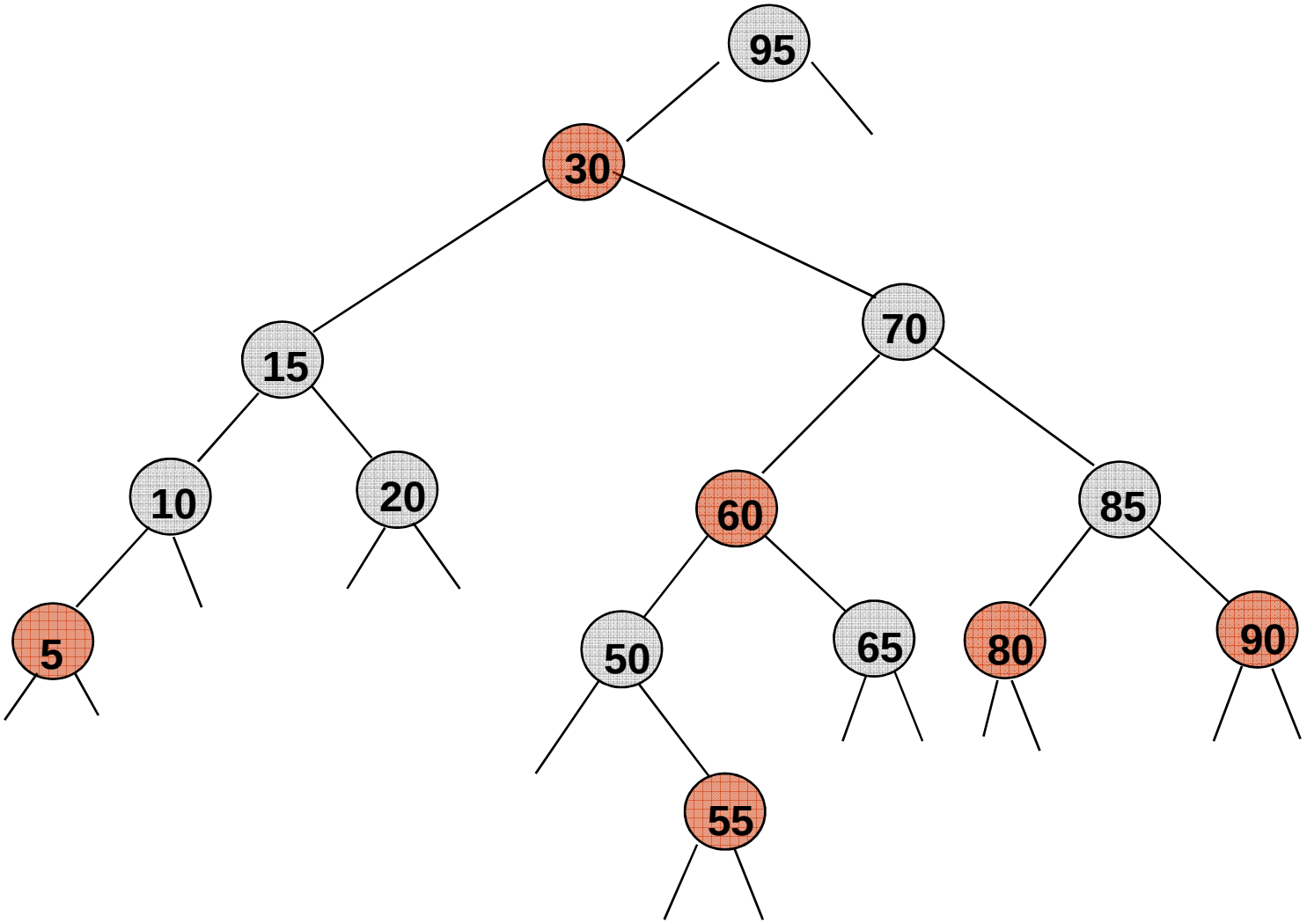
Suppression d'un élément - Rappels



Remarques

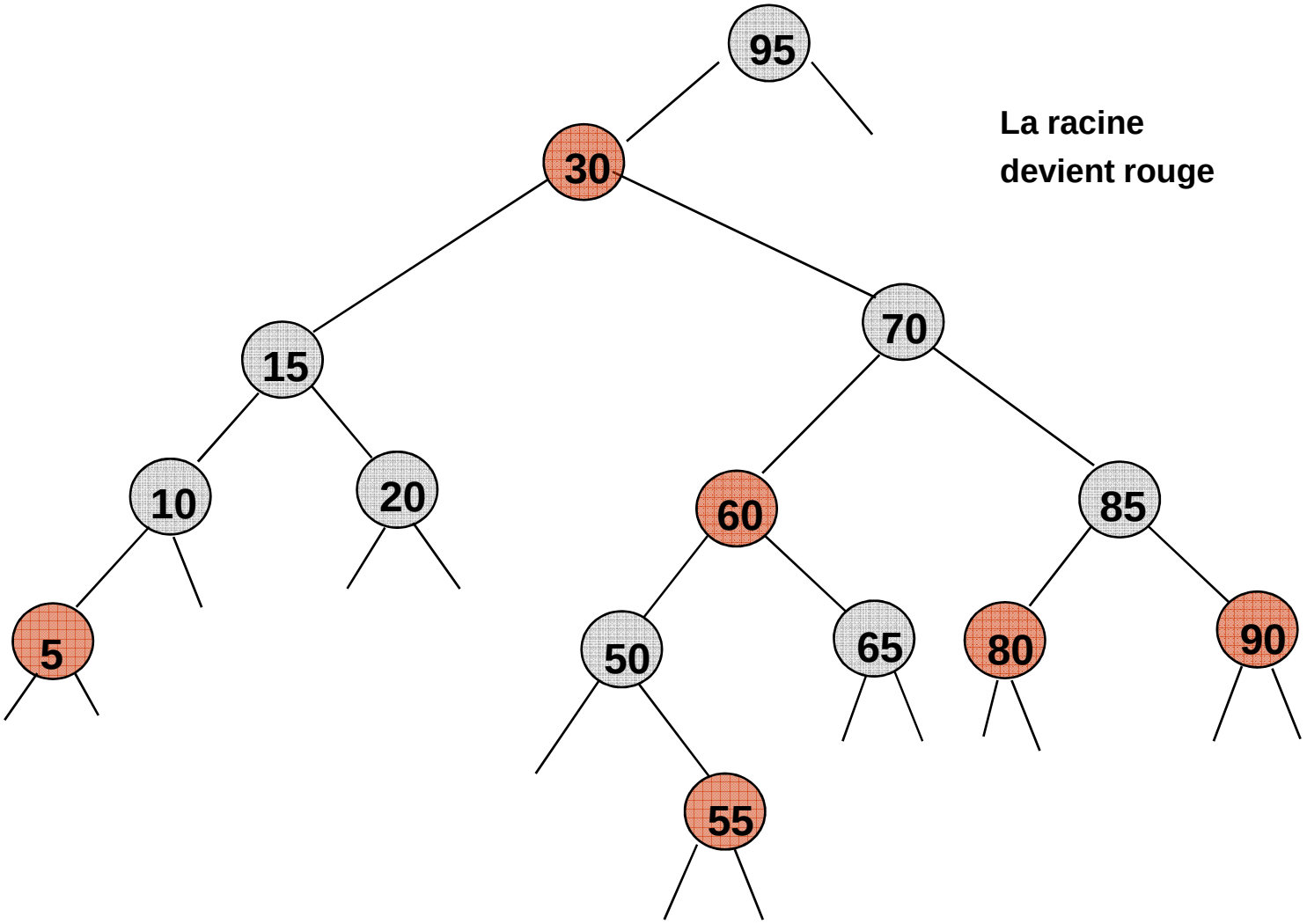
- Les seuls nœuds effectivement supprimés ont le degré d'au plus 1
- Dans un ANR, si le nœud supprimé est rouge, ça ne pose pas de problème.
  - Son père est noir
  - Son unique fils (s'il existe) est noir
  - Après suppression, toutes les propriétés de l'ARN sont gardées.
- Dans un ARN, si le nœud supprimé est noir, nous avons
  - Des cas immédiats : re-coloriage
  - Des cas difficiles : rotations et re-coloriage

Effets des suppressions



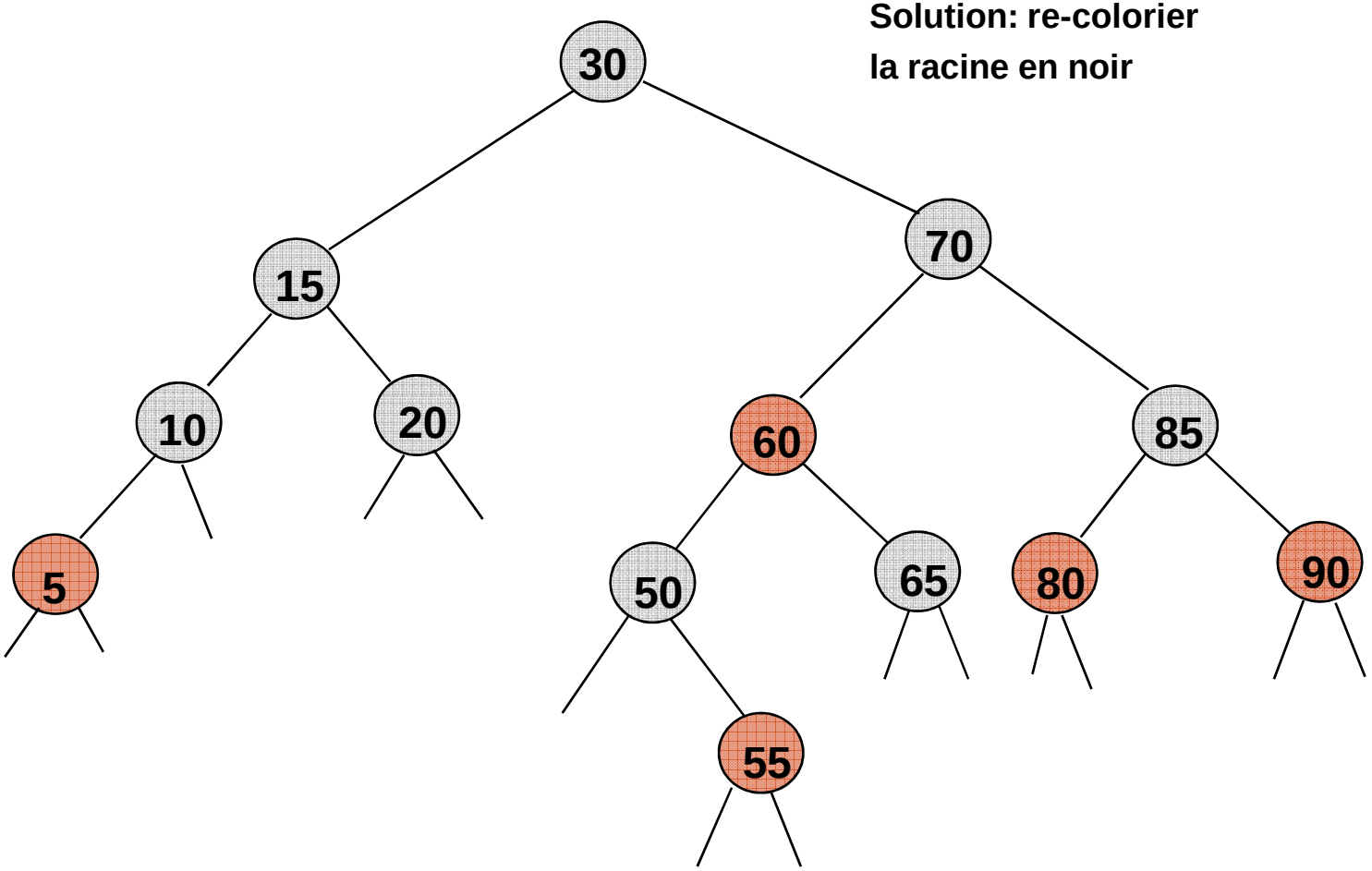
Note. Ne pas oublier les NIL, même s'ils ne sont pas dessinés

Cas facile: suppression de 95 (qui est noir)

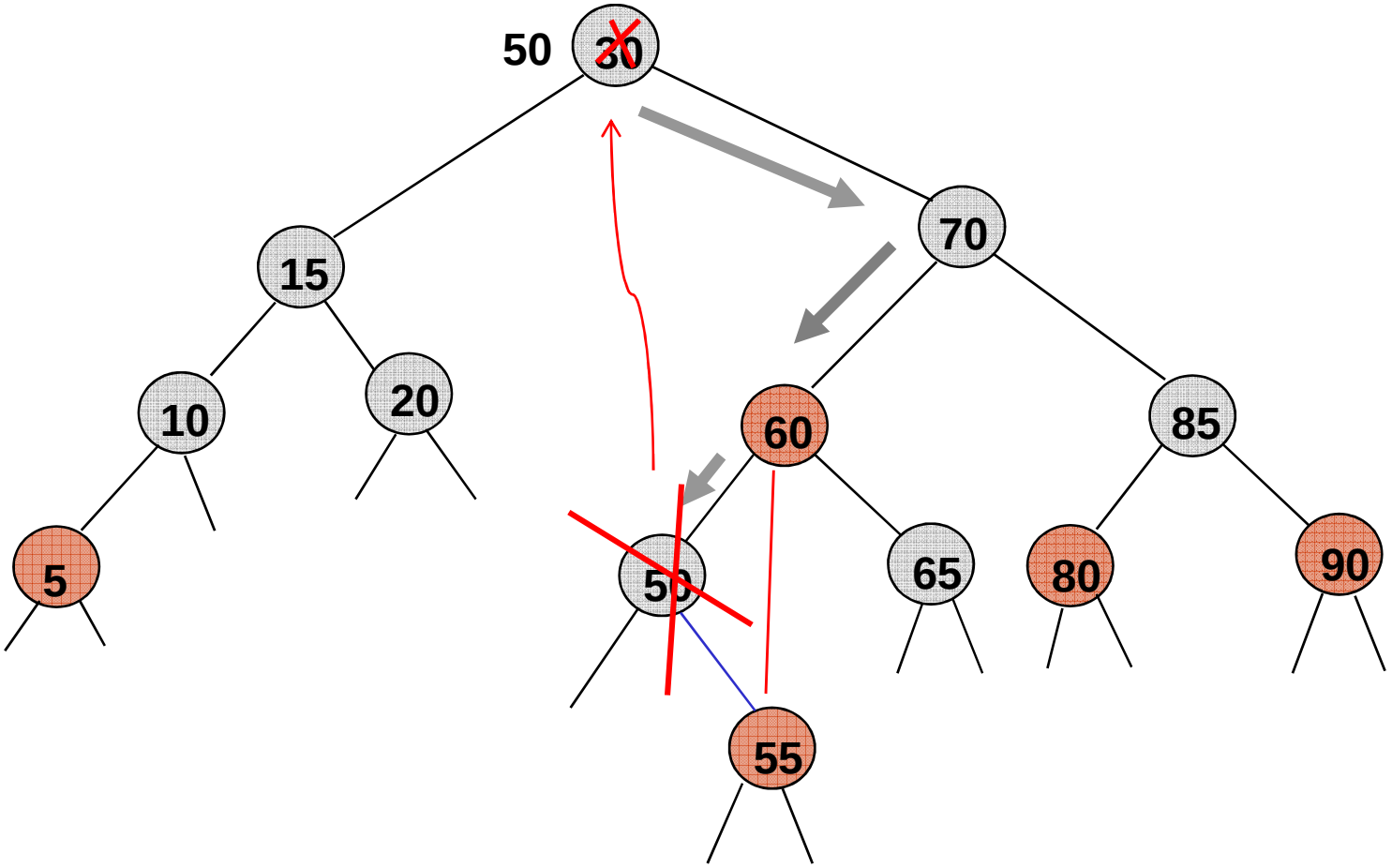


La racine devient rouge

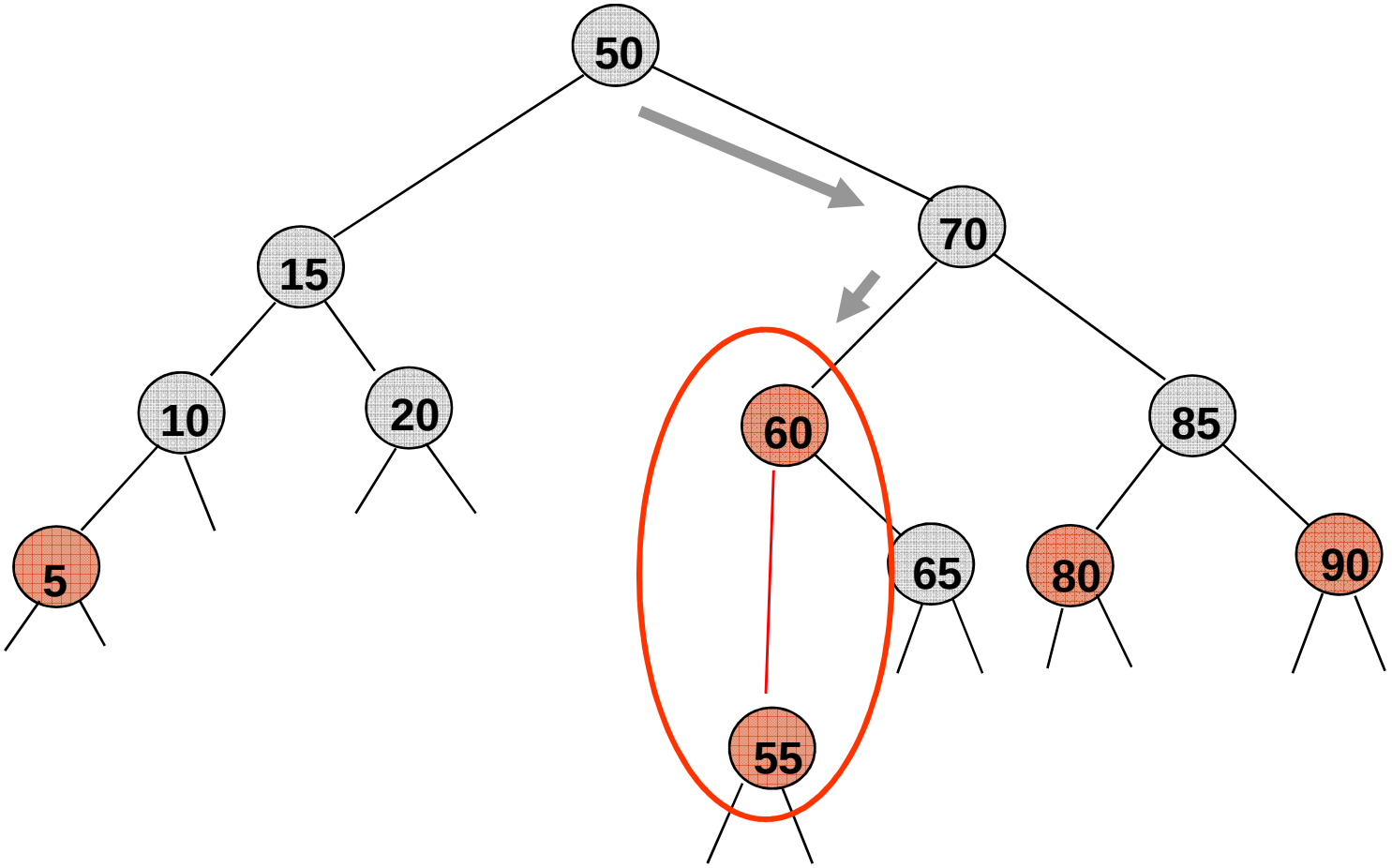
Cas facile : suppression de 95 (qui est noir)



Cas facile : suppression de 30 (qui est noir)

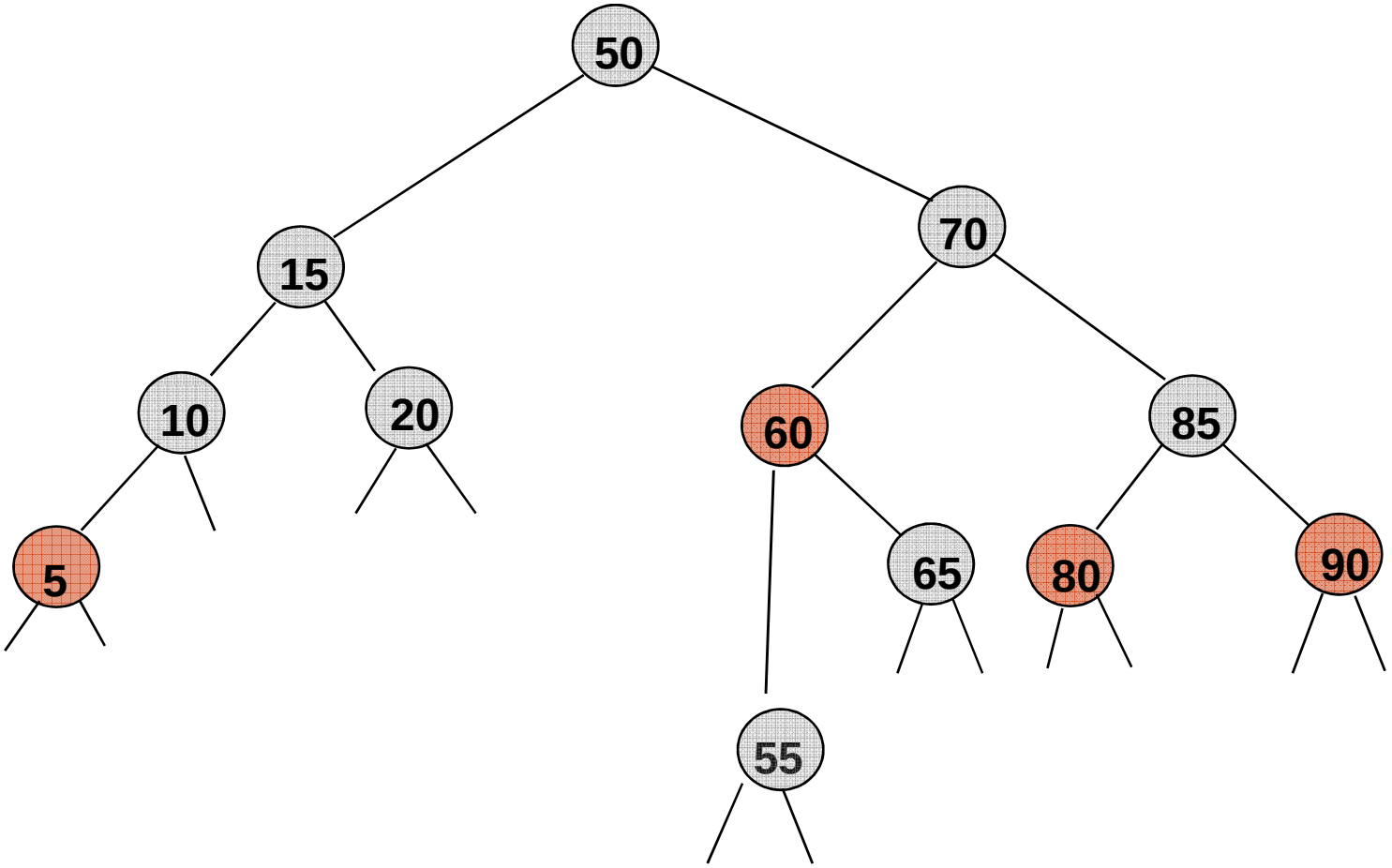


Cas facile : suppression de 30 (qui est noir)



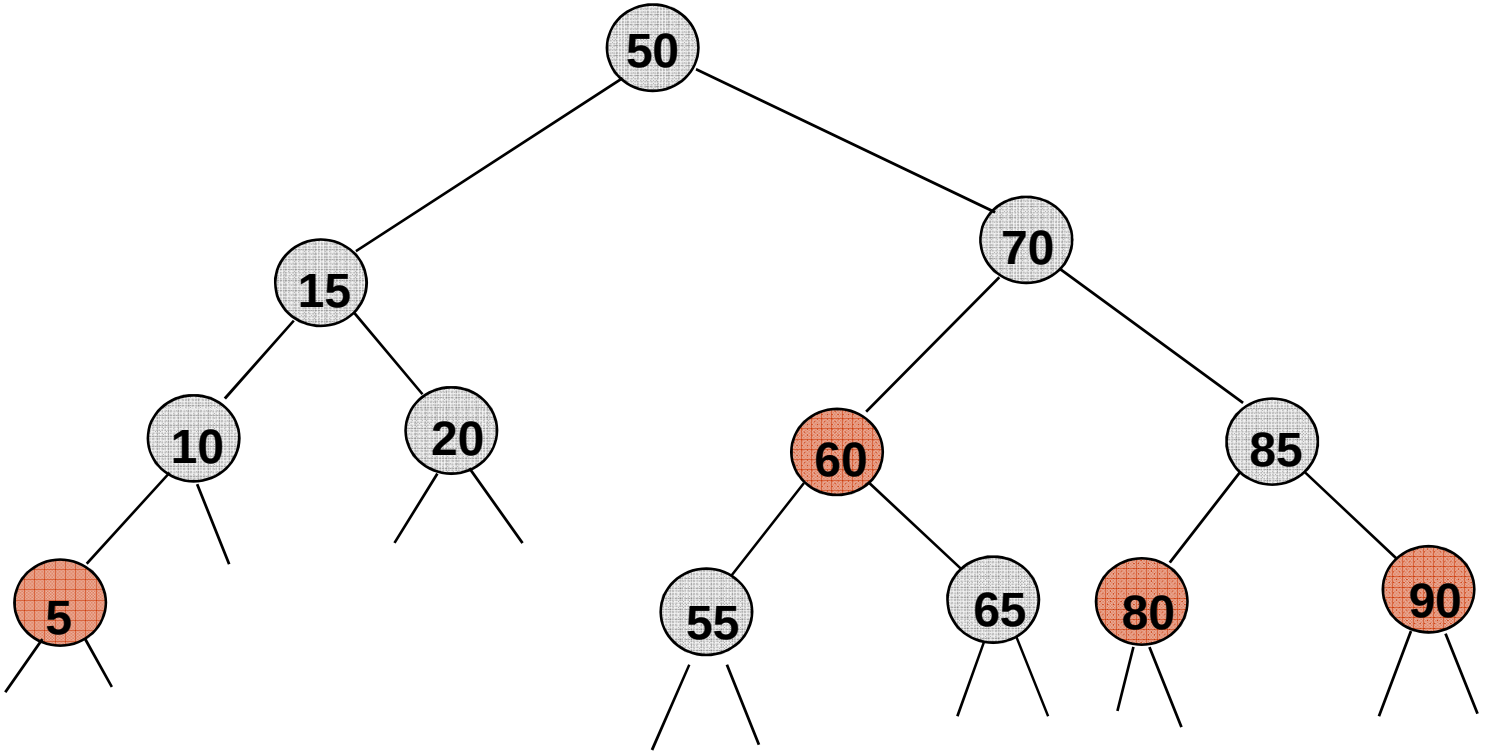
2 noeuds rouges

Cas facile : suppression de 30 (qui est noir)

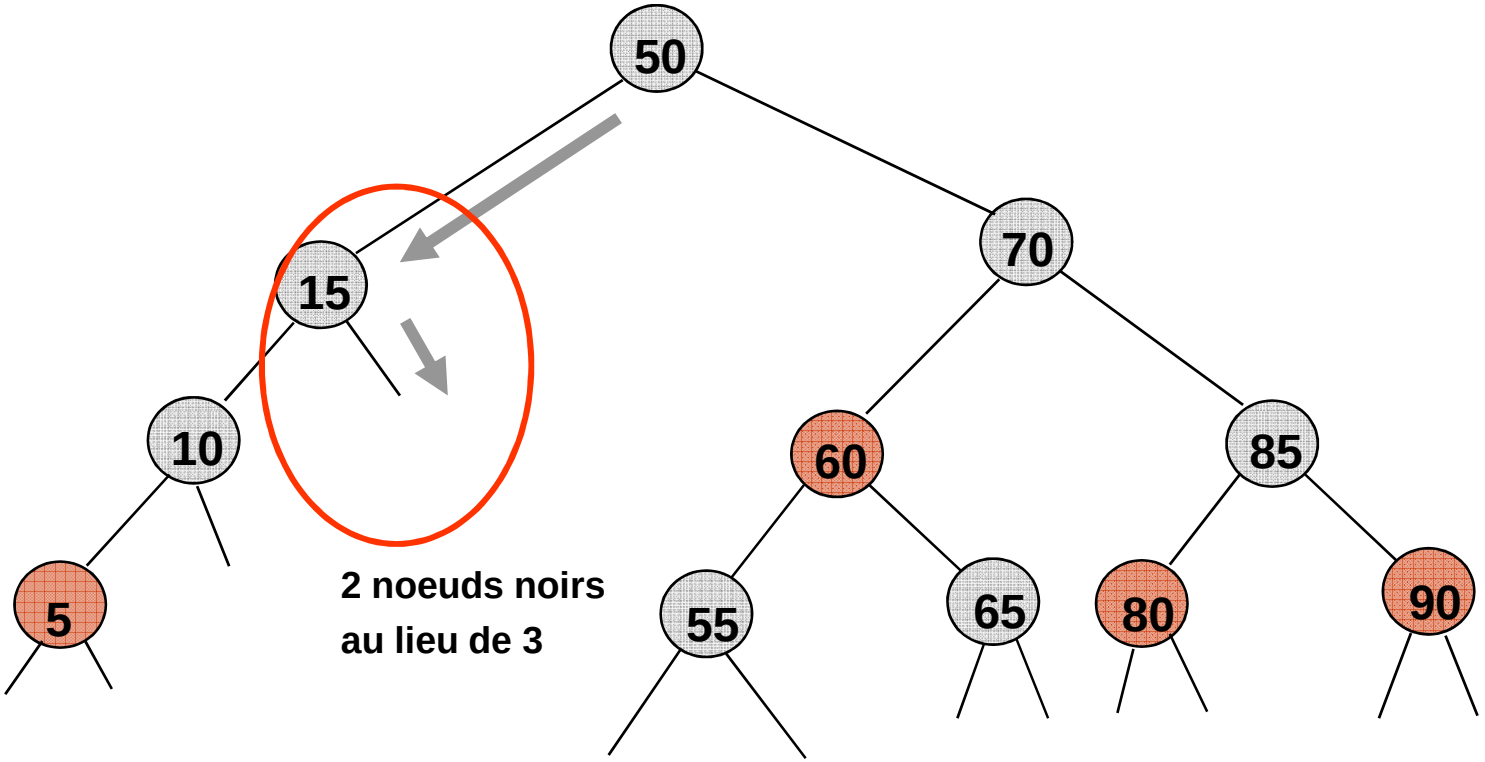


Re-colorier le nœud en noir

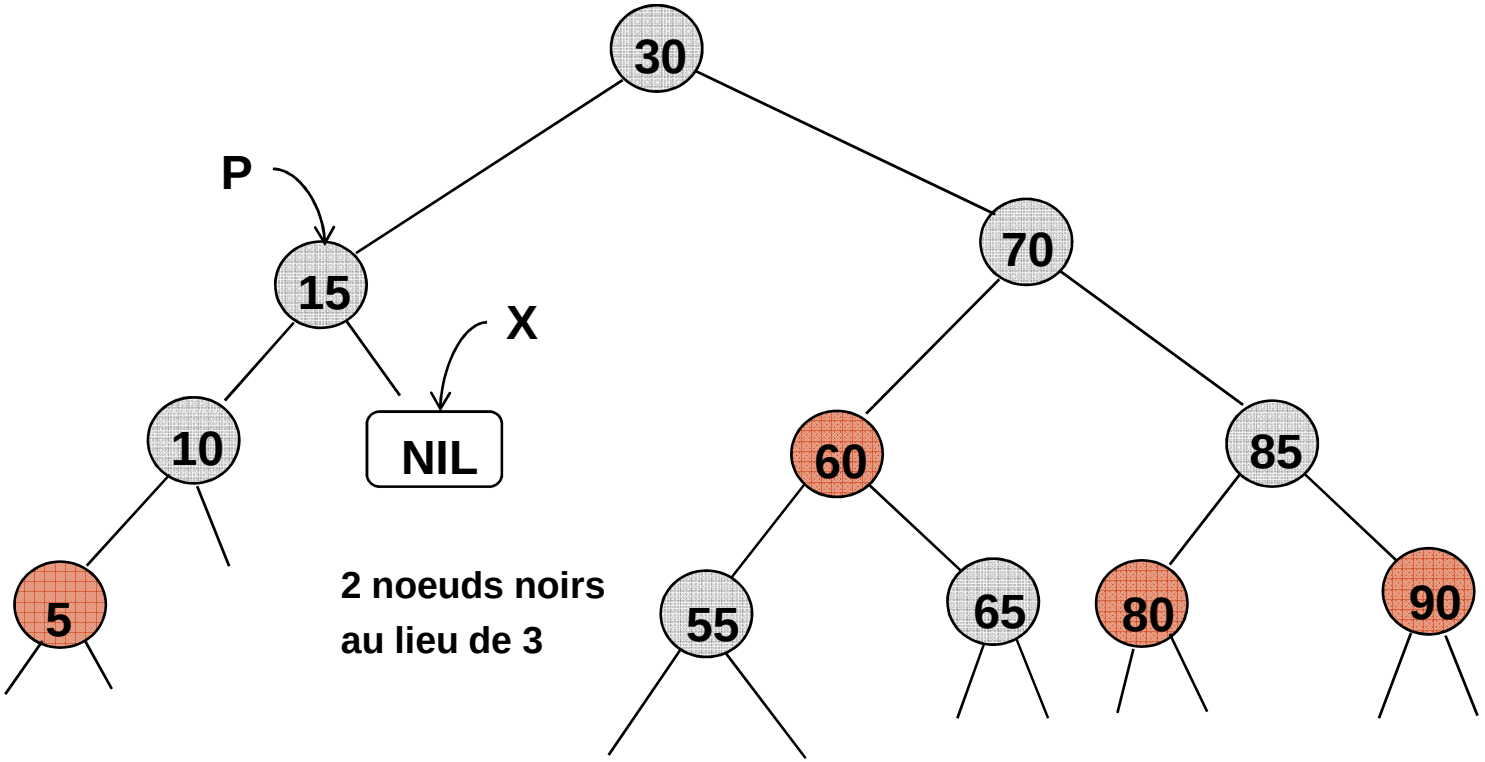
Cas difficile : suppression de 20 (qui est noir)



Cas difficile : suppression de 20 (qui est noir)



Cas difficile : suppression de 20 (qui est noir)

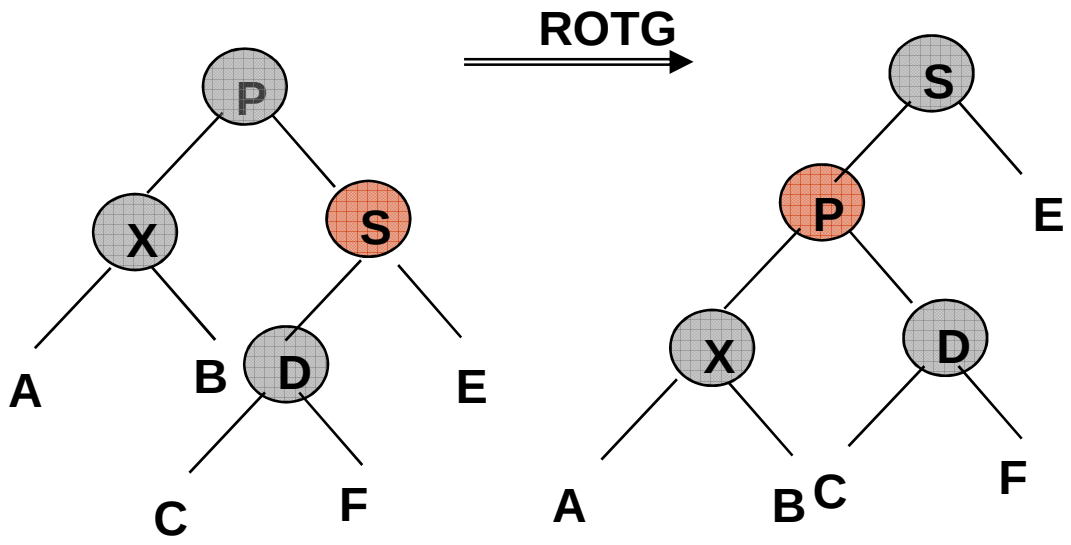


Remarques : 1 nœud a été supprimé (valeur 20), de couleur c=NOIR

1 nœud X a changé de père P (le nœud NIL, qui est représenté comme un vrai nœud, de couleur noire)

Résoudre l'insuffisance de nœuds noirs sur le chemin vers X (qui est – dans les cas difficiles -noir)

- **Cas 1** : le frère S de X est rouge



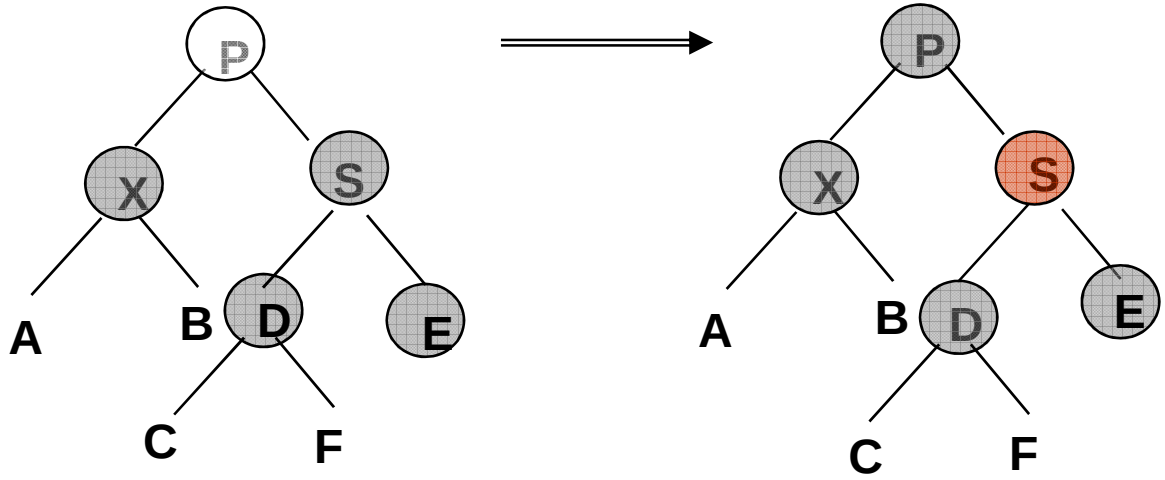
Note. Ici et par la suite, les lettres sont des pointeurs, pas des valeurs

Rotation gauche et re-coloriage, en inversant les couleurs de S, P

Ainsi, ce cas se réduit au cas où le frère de X est noir

Résoudre l'insuffisance de nœuds noirs sur le chemin vers X (qui est – dans les cas difficiles -noir)

- **Cas 2** : le frère S de X est noir, et ses fils sont noirs



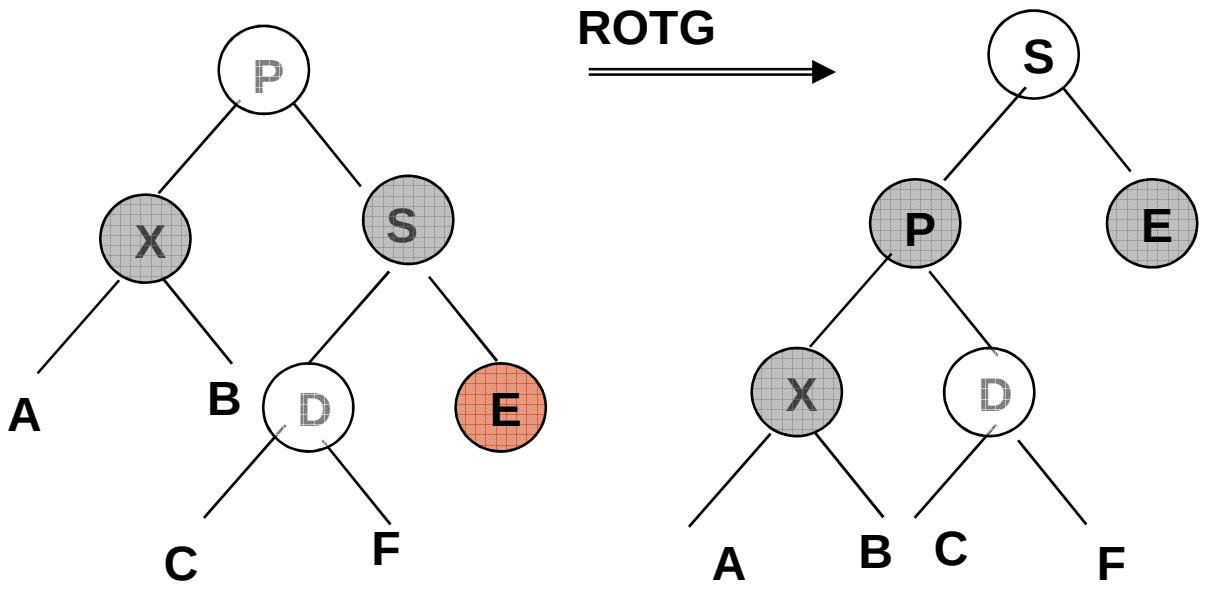
P devient noir, S devient rouge

Quelle que soit la couleur initiale de P, les chemins de P à X, D, E ont maintenant le même nombre de sommets noirs (2, comme X avant)

Donc localement OK mais globalement (vers la racine), P peut poser le même problème → problème déplacé vers la racine

Résoudre l'insuffisance de nœuds noirs sur le chemin vers X (qui est – dans les cas difficiles -noir)

- **Cas 3** : le frère S de X est noir, et au moins un de ses fils est rouge

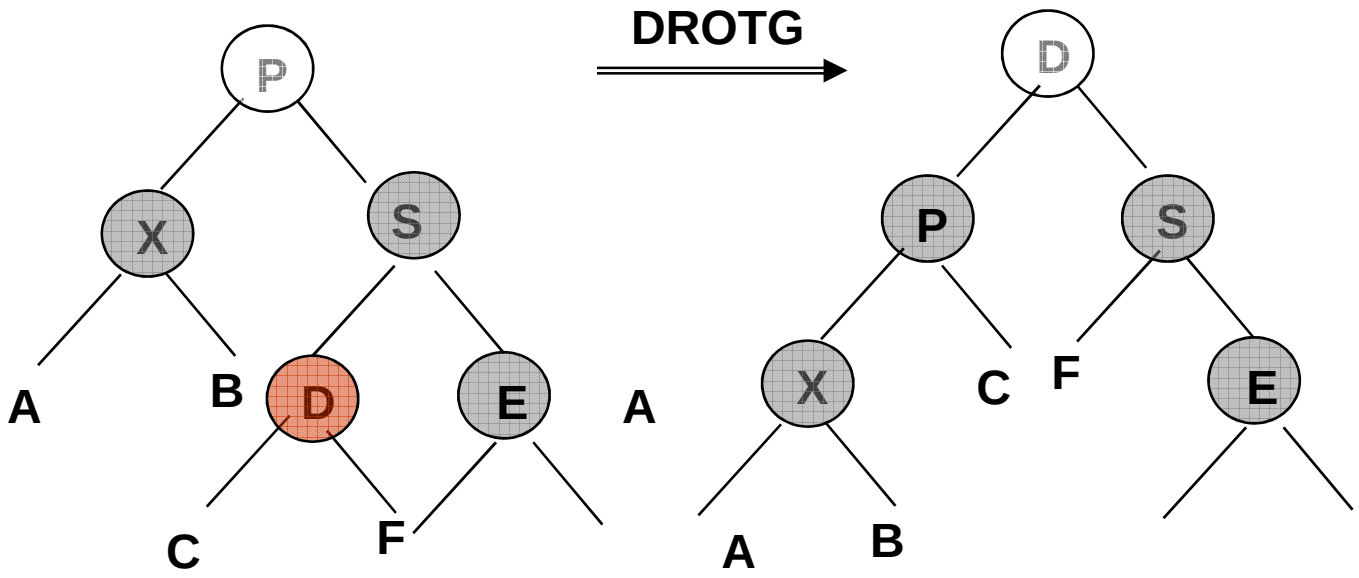


Si E est rouge : rotation gauche et re-coloriage

- La nouvelle racine prend la couleur de l'ancienne
- Ses deux fils deviennent noirs

Résoudre l'insuffisance de nœuds noirs sur le chemin vers X (qui est – dans les cas difficiles -noir)

- **Cas 3** : le frère S de X est noir, et au moins un de ses fils est rouge



Si E est noir : double rotation gauche et le re-coloriage

- La nouvelle racine prend la couleur de l'ancienne
- Ses deux fils deviennent noirs

Suppression de l'élément x

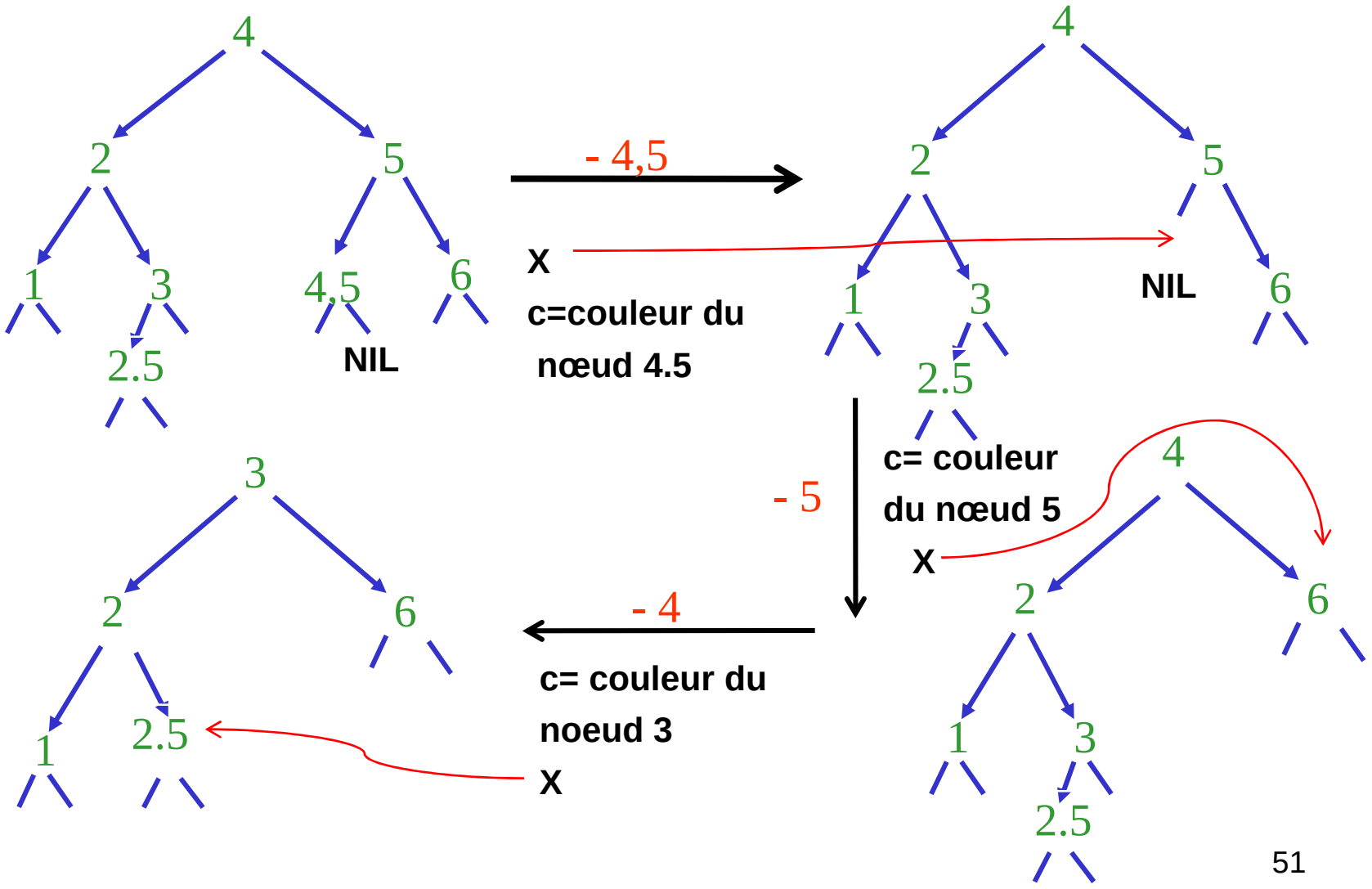
Besoin de réécrire la fonction Enlever pour les ABR t. q.

(A,X,c) ← Enlever(x,A) signifie :

- suppression de x comme dans un ABR
- A : l'arbre résultant
- X : pointe sur l'unique nœud dont le père a changé (même si c'est un nœud NIL)
- c: couleur du nœud qui a été enlevé à cette occasion (celui qui contenait x, ou celui qui contenait la valeur qui remplace x)

Note. Nous n'écrivons pas cette fonction ici.

Suppression d'un élément dans un ABR



Algorithme de suppression dit « bottom-up »

```
arn ARNenlever(element x, arn A)
(A,X,c) ← Enlever(x,A)
si (c=NOIR)  alors
   tant que ((X ≠ A) et X.couleur = NOIR) faire
      si (X est le fils gauche de son père X.p) alors
         S ← X.p.d
         si (S.couleur = ROUGE) alors  TraiterCas1(S)
         si (S.g.couleur = NOIR et S.d.couleur = NOIR) alors /
                  TraiterCas2(S)
         sinon
                  TraiterCas3(S)
      sinon (idem avec « droite » et « gauche » échangés)
X.couleur ← NOIR
```

Temps pour une suppression

Note : ARNenlever peut exécuter
une rotation sur chaque ancêtre du nœud supprimé

A arbre ARN à *n* nœuds

Temps total d'une suppression : $O(h(A)) = O(\log n)$

car ARNenlever examine une seule

branche de l'arbre (et temps d'une rotation constant)

Complexité bonne, mais n'améliore en rien les AVLs

Sommaire

- Arbres binaires équilibrés en hauteur (ou AVL)
  - Equilibrage
  - Ajout d'un élément
  - Suppression d'un élément
  - Comparaison AVL vs. ARN

Comparaison AVL vs. ARN

- **La théorie :**
  - La hauteur de l'ARN peut être plus grande que celle de l'AVL correspondant → AVL légèrement gagnant
  - Sur un ARN, avec une approche top-down (non vue ici, mais possible), pas de rééquilibrage successif vers la racine → ARN légèrement gagnant
  - ARN plus facile à implémenter comme une structure de données « à mémoire » (*persistent data structure*), très utilisée par exemple en programmation fonctionnelle
- **La pratique :**
  - Les résultats des tests dépendent des jeux de test, de l'implémentation, de l'utilisation ou non d'améliorations ponctuelles, de l'endroit où la structure est stockée sur la machine
  - Ils montrent un comportement comparable

Jeux de données « réels »

- **1000 tests**
- **Linux**
- **500 MB RAM**

test set	representation	time (seconds) BST	AVL	RB
Mozilla	plain	4.49	4.81	5.32
	parents	15.67	3.65	3.78
	threads	16.77	3.93	3.95
	right threads	16.91	4.07	4.20
	linked list	16.31	3.64	4.35
VMware	plain	208.00*	8.72	10.59
	parents	447.40*	6.31	7.32
	threads	445.80*	6.91	8.51
	right threads	446.40*	6.88	8.59
	linked list	472.00*	7.35	8.60
Squid	plain	7.34	4.41	4.67
	parents	12.52	3.69	3.80
	threads	13.44	3.92	4.18
	right threads	14.46	4.17	4.27
	linked list	13.13	4.02	4.19
random	plain	2.83	2.81	2.86
	parents	1.63	1.67	1.64
	threads	1.64	1.74	1.68
	right threads	1.92	1.96	1.93
	linked list	1.46	1.54	1.51

Source: B. Plaff, Performance analysis of BSTs in System Software, 2002.

Jeux de données « réels »

- **1000 tests**
- **Linux**
- **500 MB RAM**

test set	representation	BST	AVL	RB
Mozilla	parents	11.12	3.25	3.17
	threads	11.91	3.52	3.30
VMware	parents	331.00*	5.42	6.35
	threads	325.00*	5.96	7.42
Squid	parents	11.22	3.54	3.62
	threads	12.10	3.78	3.99
random	parents	1.60	1.64	1.62
	threads	1.61	1.71	1.65

Avec une amélioration locale pour la suppression (dans tous les arbres et avec les représentations indiquées)

Source: B. Plaff, Performance analysis of BSTs in System Software, 2002**.**

Conclusion

- Connaître les deux, c'est pouvoir manipuler les deux
- Quand on a le choix (et le temps !) :
  - Chercher (si possible) les tests correspondant au type de données à manipuler
  - Faire soi-même des tests