Partie 3: Arbres (applications)

Marie-Pierre Béal, Laurent Bulteau

Université Paris-Est Marne-la-Vallée

Plan

1 Algorithme de Huffman

2 Arbres binaires de recherche

3 Arbres AVL

Code

Définition

Un *code préfixe* est un ensemble de mots binaires non vides tel qu'aucun mot n'est le préfixe d'un autre.
On associe un symbole à chaque mot du code.

Exemple

$C = \{a : 1, b : 01, c : 001, d : 000\}$

Code

Définition

Un *code préfixe* est un ensemble de mots binaires non vides tel qu'aucun mot n'est le préfixe d'un autre.
On associe un symbole à chaque mot du code.

Exemple

$C = \{a : 1, b : 01, c : 001, d : 000\}$

Décodage : A chaque mot binaire correspond au plus une suite de symboles

01101100001

Code

Définition

Un *code préfixe* est un ensemble de mots binaires non vides tel qu'aucun mot n'est le préfixe d'un autre.
On associe un symbole à chaque mot du code.

Exemple

$C = \{a : 1, b : 01, c : 001, d : 000\}$

Décodage : A chaque mot binaire correspond au plus une suite de symboles

$$01101100001$$
$$\rightarrow 01|1|01|1|000|01$$

Code

Définition

Un *code préfixe* est un ensemble de mots binaires non vides tel qu'aucun mot n'est le préfixe d'un autre.
On associe un symbole à chaque mot du code.

Exemple

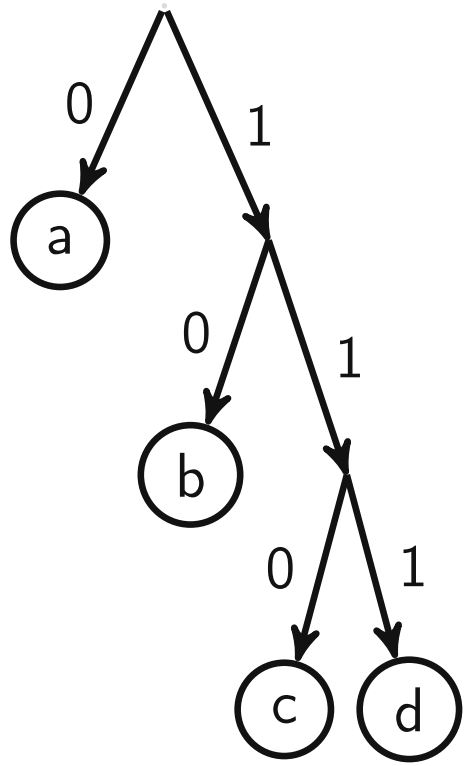
$C = \{a : 1, b : 01, c : 001, d : 000\}$

Décodage : A chaque mot binaire correspond au plus une suite de symboles

$$\begin{aligned} & 01101100001 \\ \rightarrow\ & 01|1|01|1|000|01 \\ \rightarrow\ & babadb \end{aligned}$$

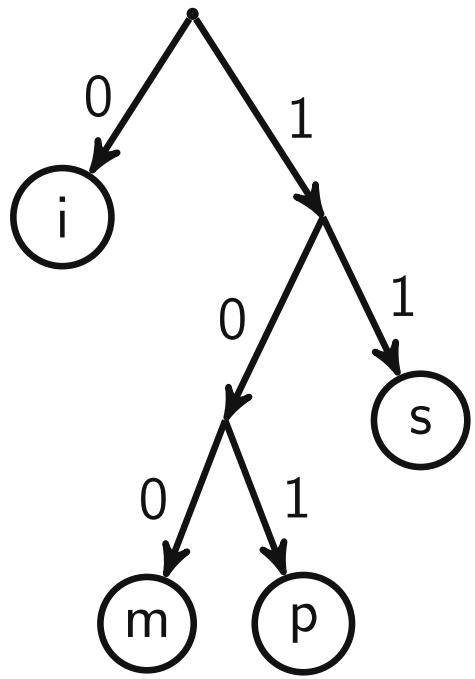
Bijection : code préfixe $\leftrightarrow$ arbre binaire

$$C = \{a : 0, b : 10, c : 110, d : 111\}$$



Exemple

1. Décoder le texte suivant en utilisant le code préfixe décrit par l'arbre ci-dessous.
   1 0 0 0 1 1 1 1 0 1 1 1 1 0 1 0 1 0

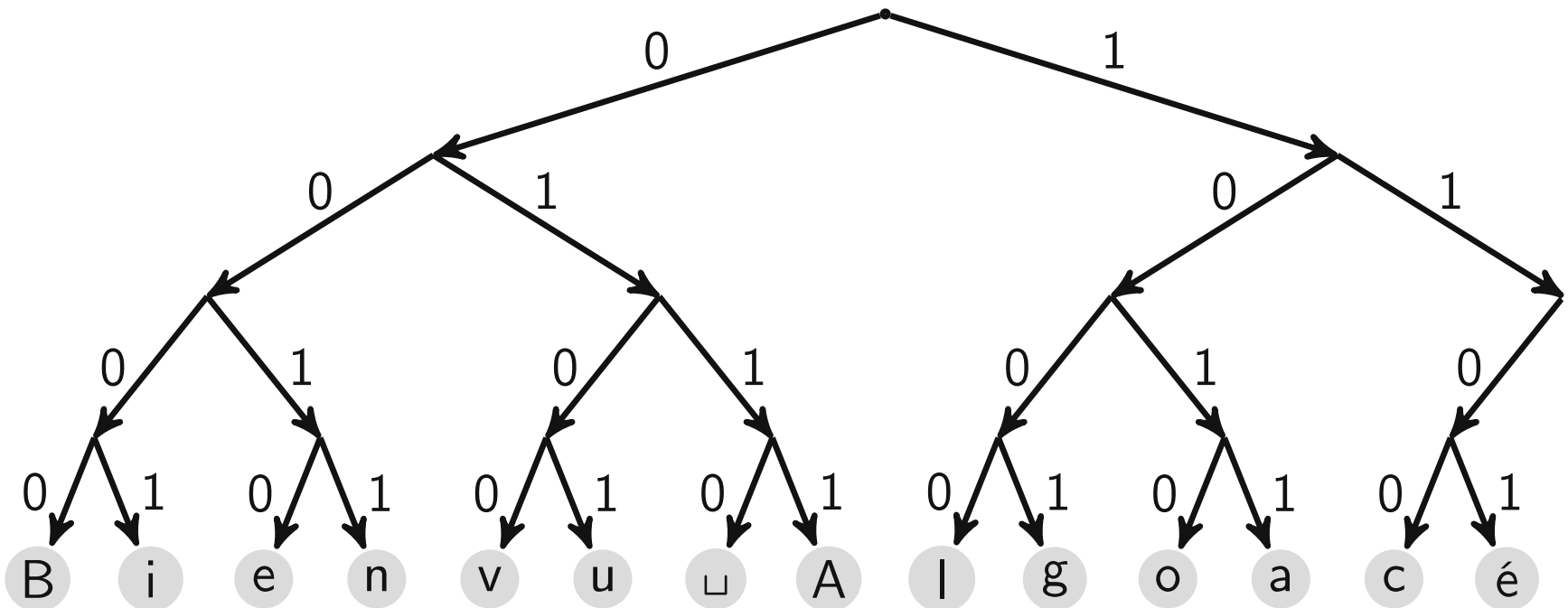


Codage de "Bienvenue en Algo Avancée"

Code ascii : $\{A : 01000001, B : 01000010, C : 01000011, \ldots\}$
Chaîne codée : 01000010 01100101 01100101...
Longueur : $25 * 8 = 200$ bits

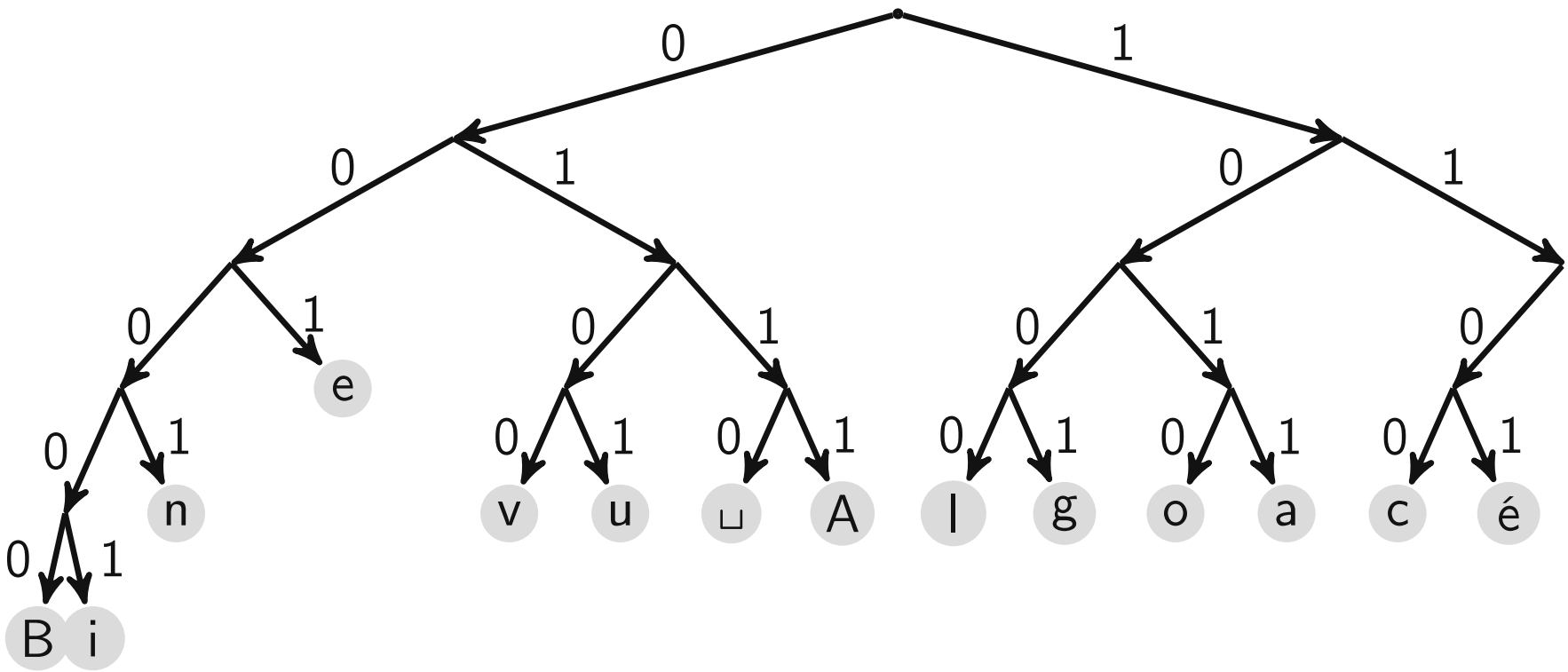
Codage de "Bienvenue en Algo Avancée"

Code réduit : pour 14 symboles $\{B : 0000, i : 0001, e : 0010, \ldots\}$
Chaîne codée : 0000 0001 0010...
Longueur : $25 * 4 = 100$ bits (+ le code)



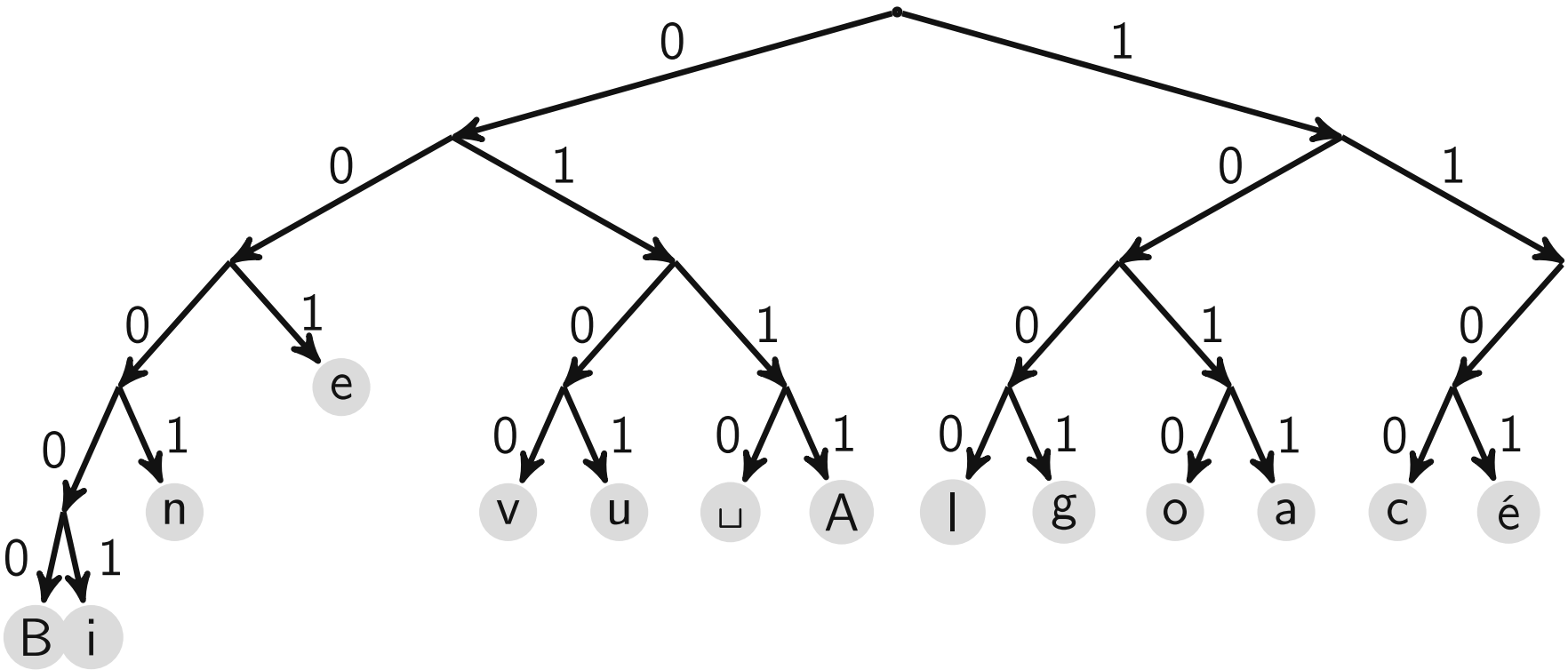
Codage de "Bienvenue en Algo Avancée"

Code réduit : pour 14 symboles $\{B : 0000, i : 0001, e : 0010, \ldots\}$
Chaîne codée : 0000 0001 0010...
Longueur : $25 * 4 = 100$ bits (+ le code)
Idée : on peut utiliser des codes de longueurs différentes pour des lettres plus ou moins fréquentes



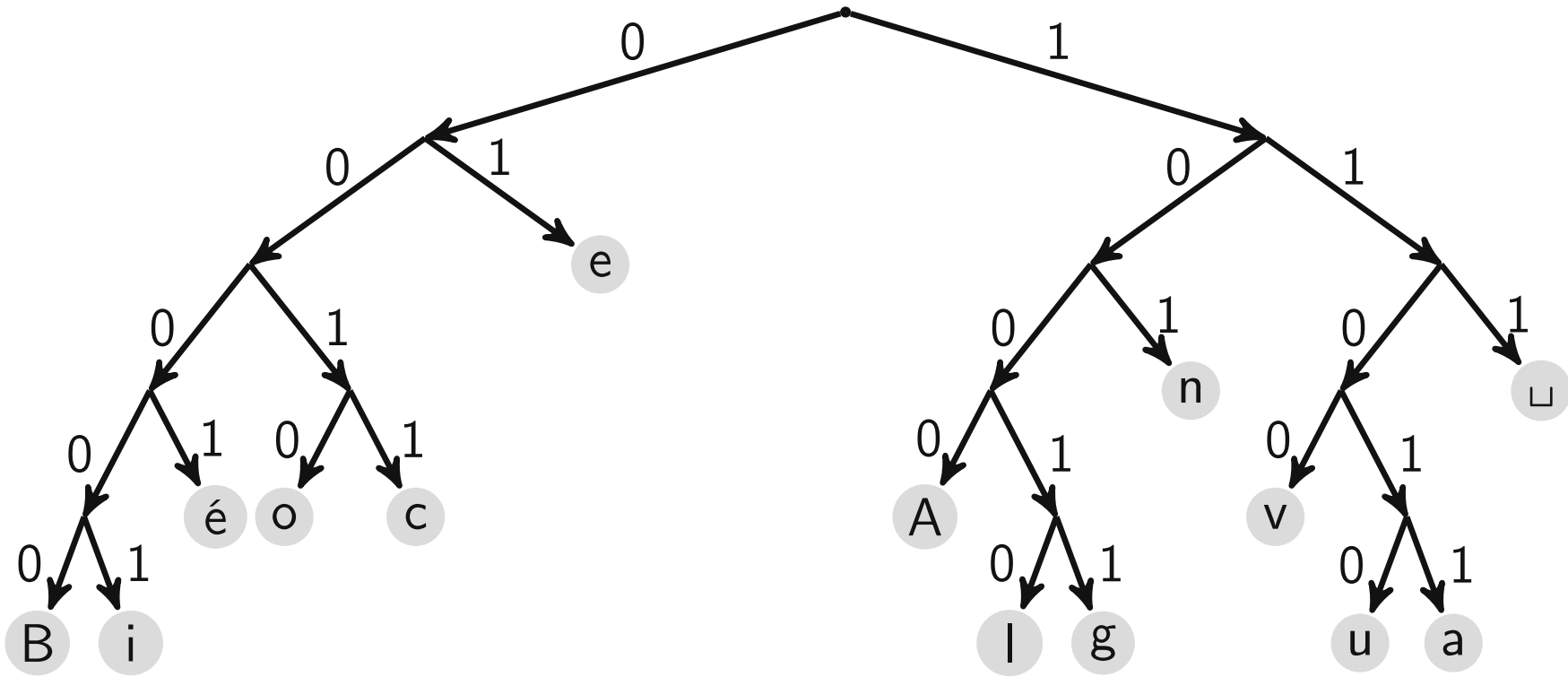
Codage de "Bienvenue en Algo Avancée"

Code réduit : pour 14 symboles $\{B : 00000, i : 00001, e : 001, \ldots\}$
Chaîne codée : 00000 00001 001...
Longueur : 97 bits (+ le code)
Idée : on peut utiliser des codes de longueurs différentes pour des lettres plus ou moins fréquentes



Codage de "Bienvenue en Algo Avancée"

Code de Huffman : le plus compact $\{B : 00000, i : 00001, e : 01, \ldots\}$
Chaîne codée : 00000 00001 01 101 1100...
Longueur : 89 bits (+ le code)



Algorithme de Huffman

Fréquences : $f : A \to \mathbb{N} \setminus \{0\}$
Arbre élémentaire (feuille) : $X_a = (x_a)$ pour chaque lettre $a \in A$

Huffman(Alphabet A, fonction f) : arbre T

1 $L = (X_a \mid a \in A)$ // L est une liste d'arbres
2 **while** $|L| > 1$
3 $g, d =$ arbres de L avec $f(g)$, $f(d)$ minimales
4 $x = newTree(' ', g, d)$
5 $f(x) = f(g) + f(d)$
6 remplacer g, d par x dans L
7 **return** l'unique arbre de L // Temps linéaire si f est trié

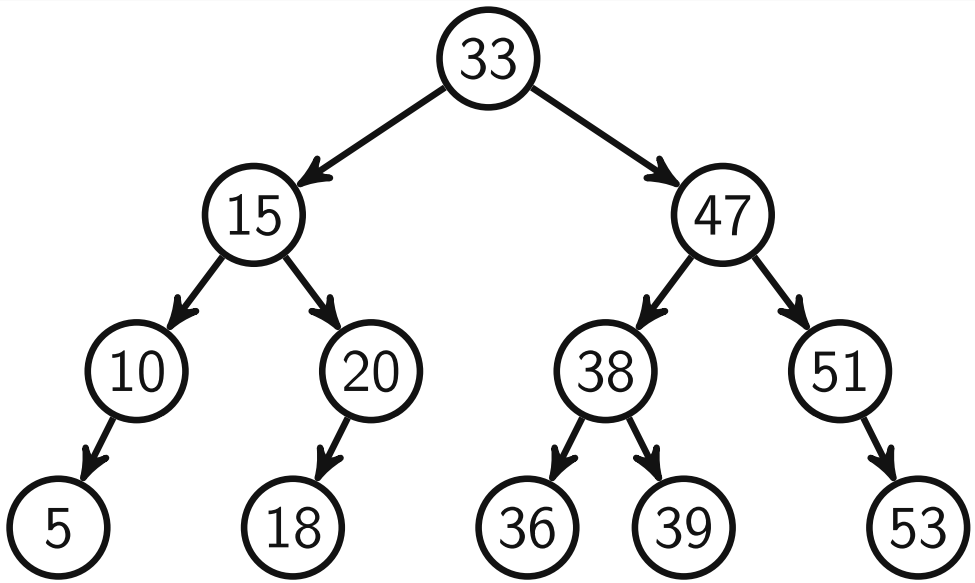
Exercice - algorithme de Huffman

2. Calculer le code de Huffman pour le texte "`abracadabra`", puis donner la séquence binaire correspondante.
   NB : plusieurs solutions sont possibles, mais elles ont toutes la même longueur.

Arbres binaires de recherche

Définition

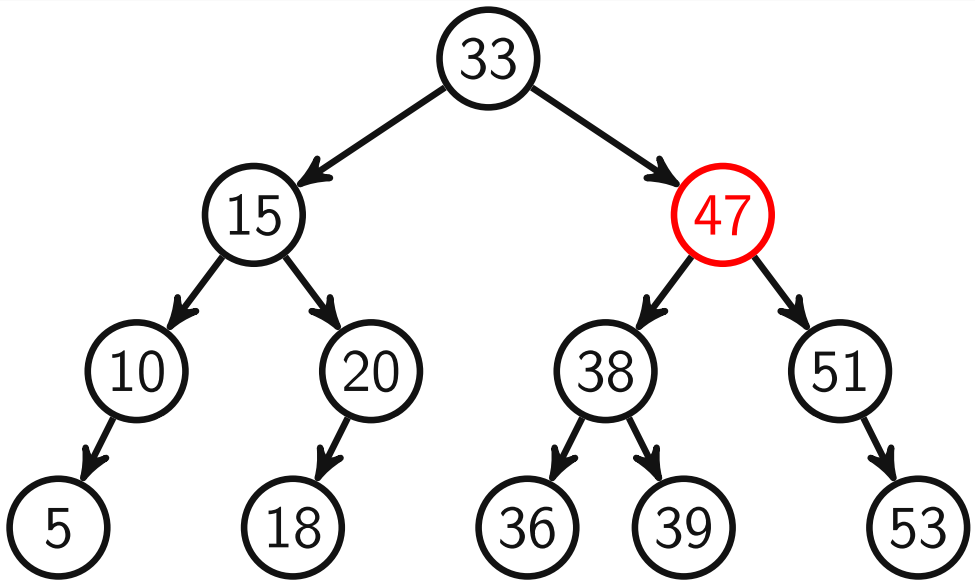
Un arbre binaire est un **arbre binaire de recherche** (ABR) si pour tout nœud, l'étiquette du nœud
– est supérieure ou égale aux étiquettes des nœuds du sous-arbre gauche
– est inférieure ou égale aux étiquettes des nœuds du sous-arbre droit.



Arbres binaires de recherche

Définition

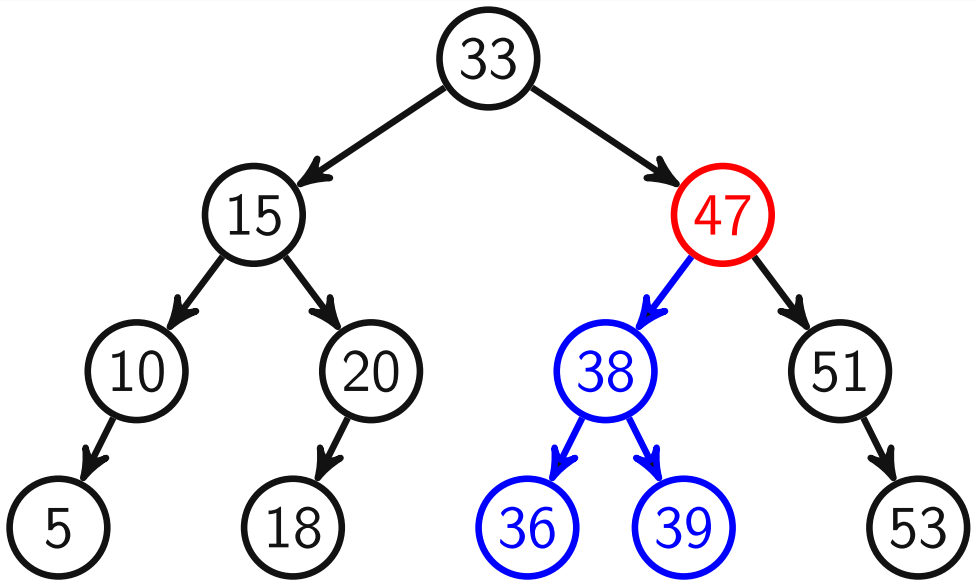
Un arbre binaire est un **arbre binaire de recherche** (ABR) si pour tout nœud, l'étiquette du nœud
– est supérieure ou égale aux étiquettes des nœuds du sous-arbre gauche
– est inférieure ou égale aux étiquettes des nœuds du sous-arbre droit.



Arbres binaires de recherche

Définition

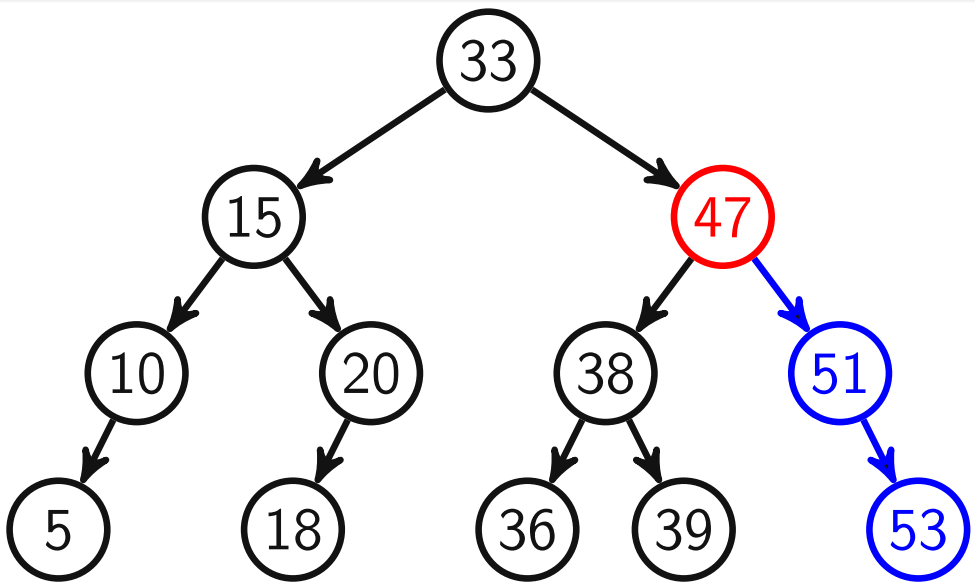
Un arbre binaire est un **arbre binaire de recherche** (ABR) si pour tout nœud, l'étiquette du nœud
– est supérieure ou égale aux étiquettes des nœuds du sous-arbre gauche
– est inférieure ou égale aux étiquettes des nœuds du sous-arbre droit.



Arbres binaires de recherche

Définition

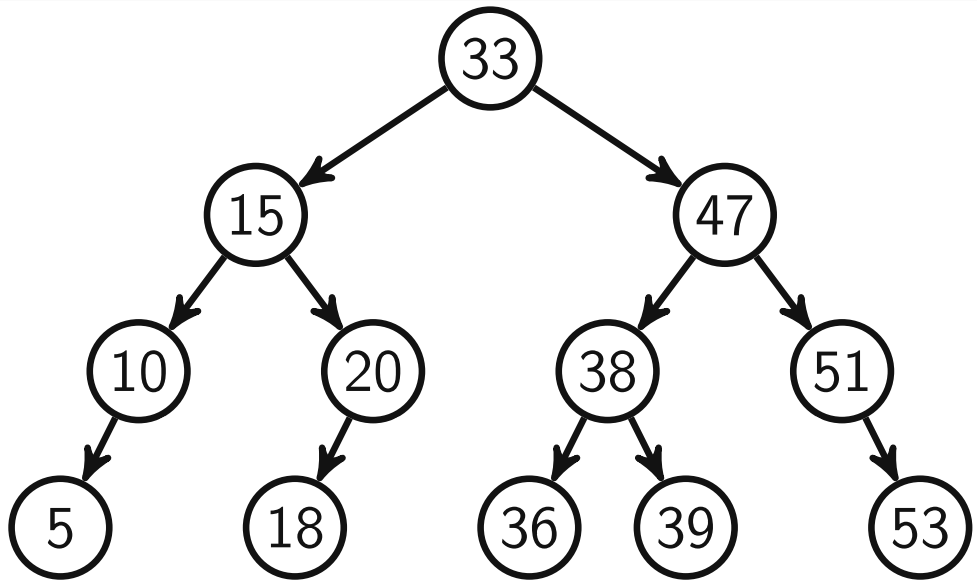
Un arbre binaire est un **arbre binaire de recherche** (ABR) si pour tout nœud, l'étiquette du nœud
– est supérieure ou égale aux étiquettes des nœuds du sous-arbre gauche
– est inférieure ou égale aux étiquettes des nœuds du sous-arbre droit.



Arbres binaires de recherche

Définition

Un arbre binaire est un **arbre binaire de recherche** (ABR) si pour tout nœud, l'étiquette du nœud
– est supérieure ou égale aux étiquettes des nœuds du sous-arbre gauche
– est inférieure ou égale aux étiquettes des nœuds du sous-arbre droit.



1. Comment parcourir un ABR pour afficher la liste des éléments en ordre croissant ?

Recherche d'un élément

```
int find(Arbre a, Element e) {
   if (a==NULL) return 0;
   if (a->elem == e) return 1;
   if (a->elem > e)
      return find(a->filsG, e);
   else
      return find(a->filsD, e);
}
```

Recherche d'un élément

```
int find(Arbre a, Element e) {
   if (a==NULL) return 0;
   if (a->elem == e) return 1;
   if (a->elem > e)
      return find(a->filsG, e);
   else
      return find(a->filsD, e);
}
```

2. Quelle est la complexité de cette fonction pour un ABR de n nœuds et de hauteur h ?

Recherche d'un élément

```
int find(Arbre a, Element e) {
   if (a==NULL) return 0;
   if (a->elem == e) return 1;
   if (a->elem > e)
      return find(a->filsG, e);
   else
      return find(a->filsD, e);
}
```

2. Quelle est la complexité de cette fonction pour un ABR de n nœuds et de hauteur h ?
3. Écrire une fonction permettant d'insérer un élément dans un arbre binaire.

```
void insert(Arbre* a, Element e);
```

Exercices

4. Calculer l'arbre obtenu en insérant successivement les entiers de 1 à 10 dans l'ordre suivant :
   5 8 1 3 4 6 10 7 9 2
5. Écrire des fonctions calculant le minimum et le maximum d'un arbre binaire de recherche. Quelles sont leurs complexités ?

```
Element minBST(Arbre a);
Element maxBST(Arbre a);
```

6. Écrire une fonction permettant de tester si un arbre binaire est un arbre binaire de recherche.

```
int isBST(Arbre a);
```

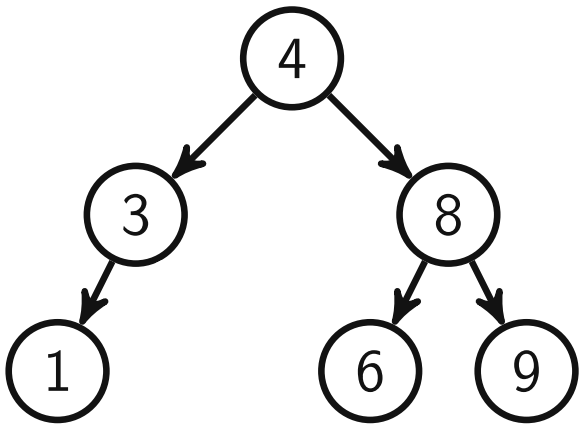
   - En utilisant les fonctions `minBST` et `maxBST`. Quelle est sa complexité ?
   - Peut-on améliorer la complexité en utilisant une autre méthode (penser à la question 1) ?

Retirer le minimum

```
Element popMinimum(Arbre* a) {
   Element min;
   if ((*a)->filsG == NULL) {
      min = (*a)->elem;
      *a = (*a)->filsD;
   } else
      min = popMinimum(& ((*a)->filsG));
   return min;
}
```

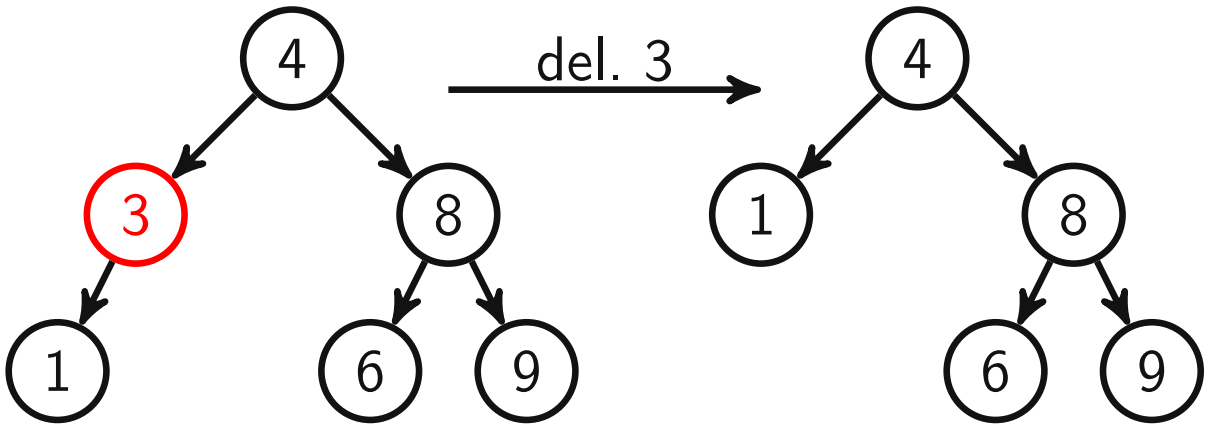
Retirer un élément quelconque

- On parcours l'arbre pour trouver le nœud contenant cet élément
- Si ce nœud a 0 ou 1 fils : on peut le retirer
- Sinon : on extrait le minimum du sous-arbre droit, qui vient remplacer l'élément retiré



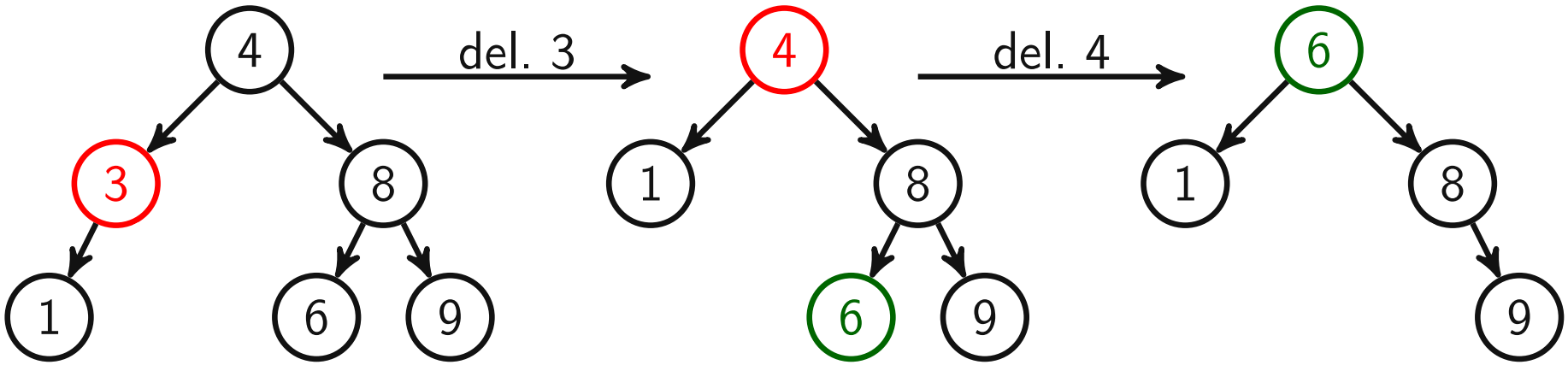
Retirer un élément quelconque

- On parcours l'arbre pour trouver le nœud contenant cet élément
- Si ce nœud a 0 ou 1 fils : on peut le retirer
- Sinon : on extrait le minimum du sous-arbre droit, qui vient remplacer l'élément retiré



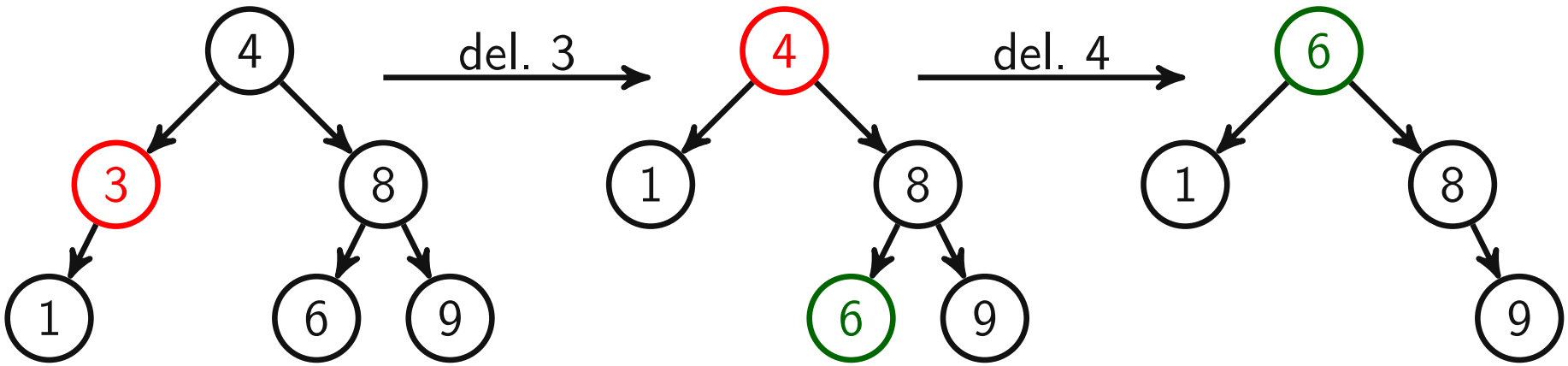
Retirer un élément quelconque

- On parcours l'arbre pour trouver le nœud contenant cet élément
- Si ce nœud a 0 ou 1 fils : on peut le retirer
- Sinon : on extrait le minimum du sous-arbre droit, qui vient remplacer l'élément retiré



Retirer un élément quelconque

- On parcours l'arbre pour trouver le nœud contenant cet élément
- Si ce nœud a 0 ou 1 fils : on peut le retirer
- Sinon : on extrait le minimum du sous-arbre droit, qui vient remplacer l'élément retiré



7 Écrire une fonction permettant de supprimer un élément dans un arbre binaire.

```
void delete(Arbre* a, Element e);
```

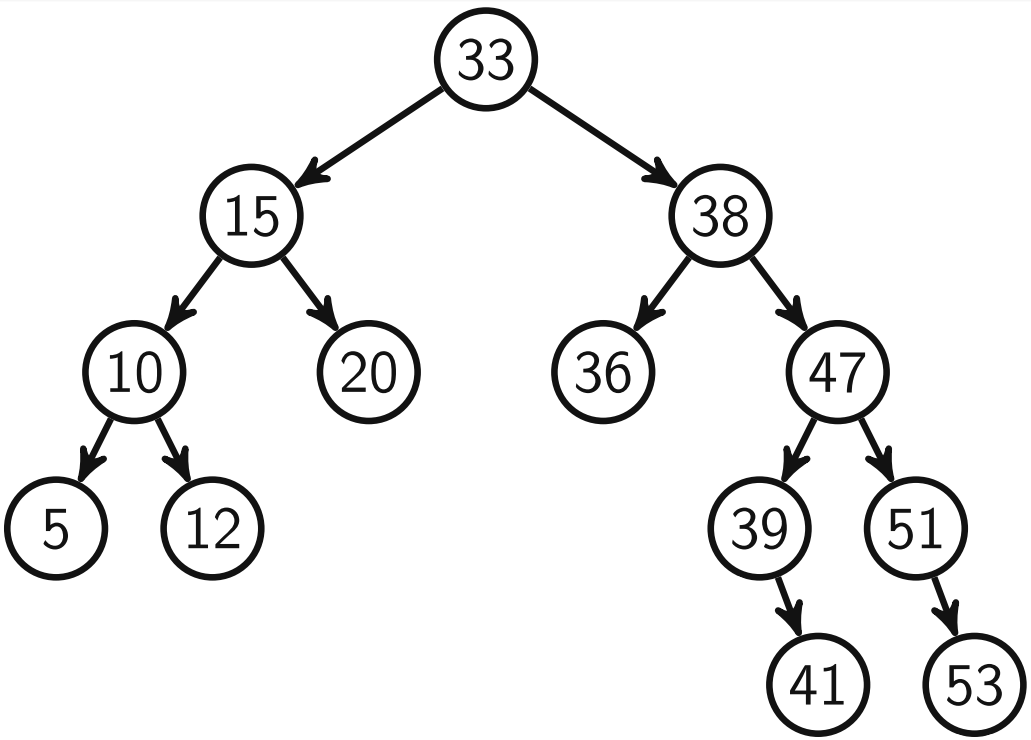
Arbres AVL

Définitions

La **balance** d'un nœud est :

hauteur du sous-abre droit − *hauteur du sous-arbre gauche*

Un **AVL** est un arbre binaire de recherche tel que la balance de chaque nœud est comprise entre -1 et +1



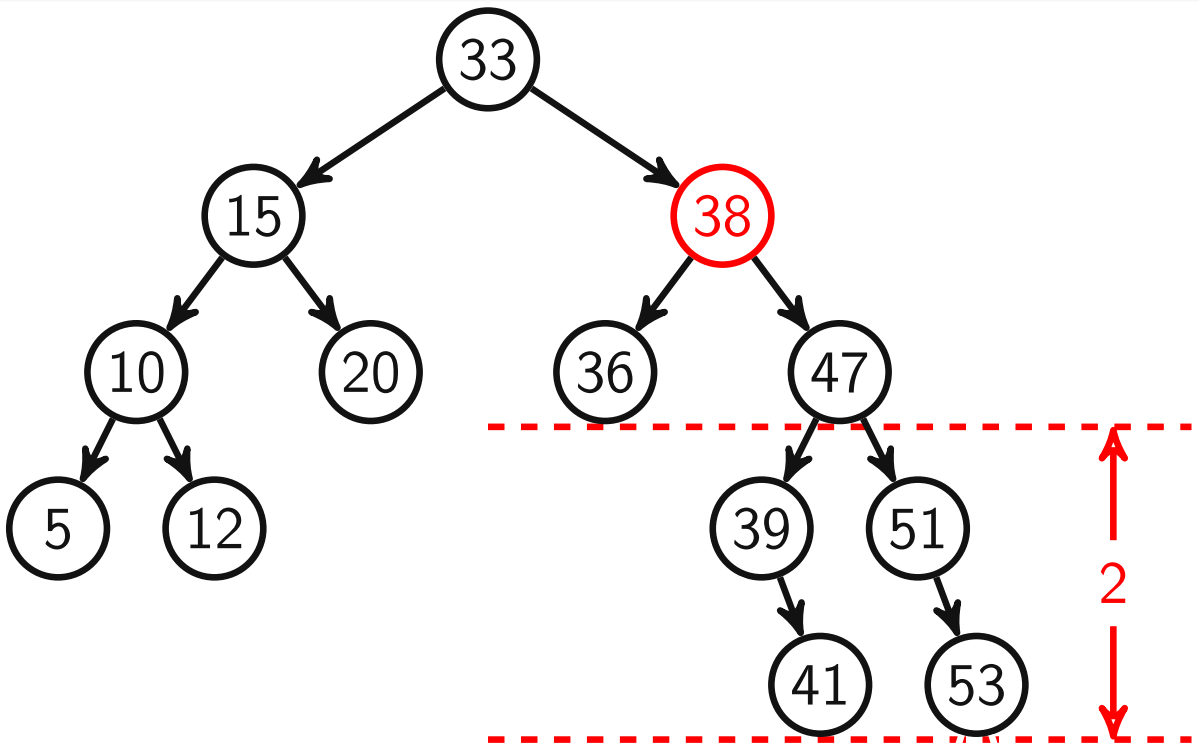
Arbres AVL

Définitions

La **balance** d'un nœud est :

hauteur du sous-abre droit − *hauteur du sous-arbre gauche*

Un **AVL** est un arbre binaire de recherche tel que la balance de chaque nœud est comprise entre -1 et +1



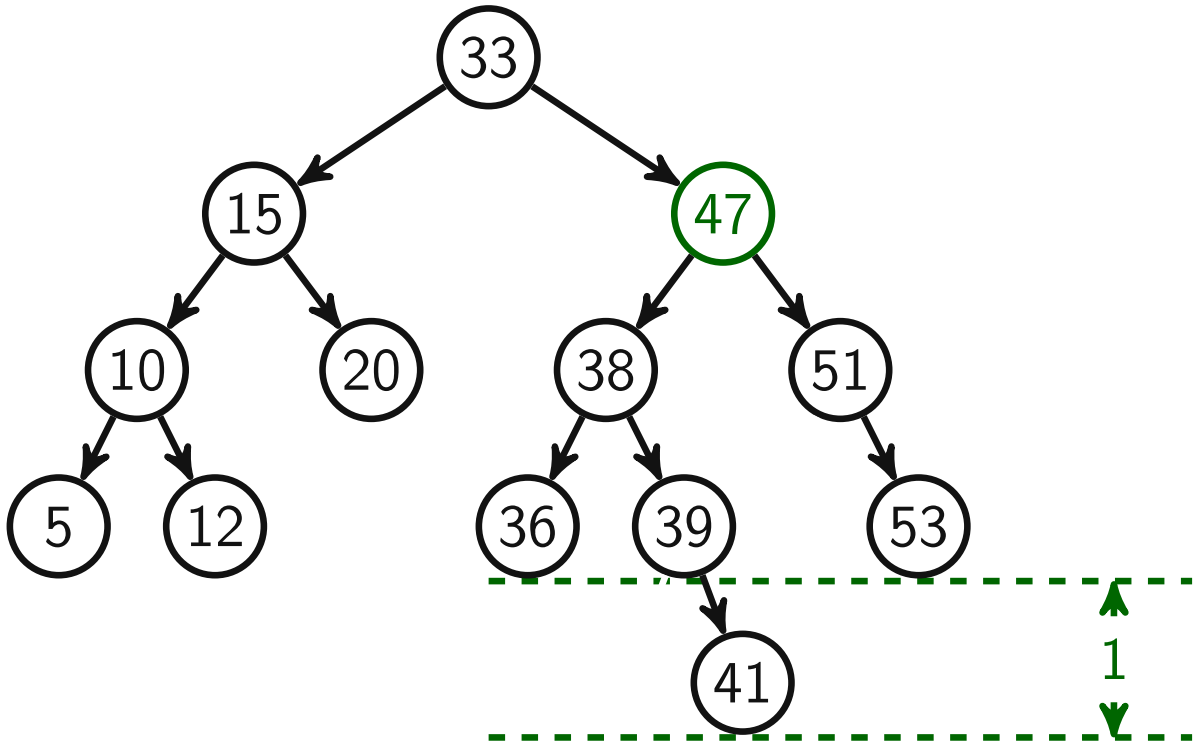
Arbres AVL

Définitions

La **balance** d'un nœud est :

hauteur du sous-abre droit − *hauteur du sous-arbre gauche*

Un **AVL** est un arbre binaire de recherche tel que la balance de chaque nœud est comprise entre -1 et +1



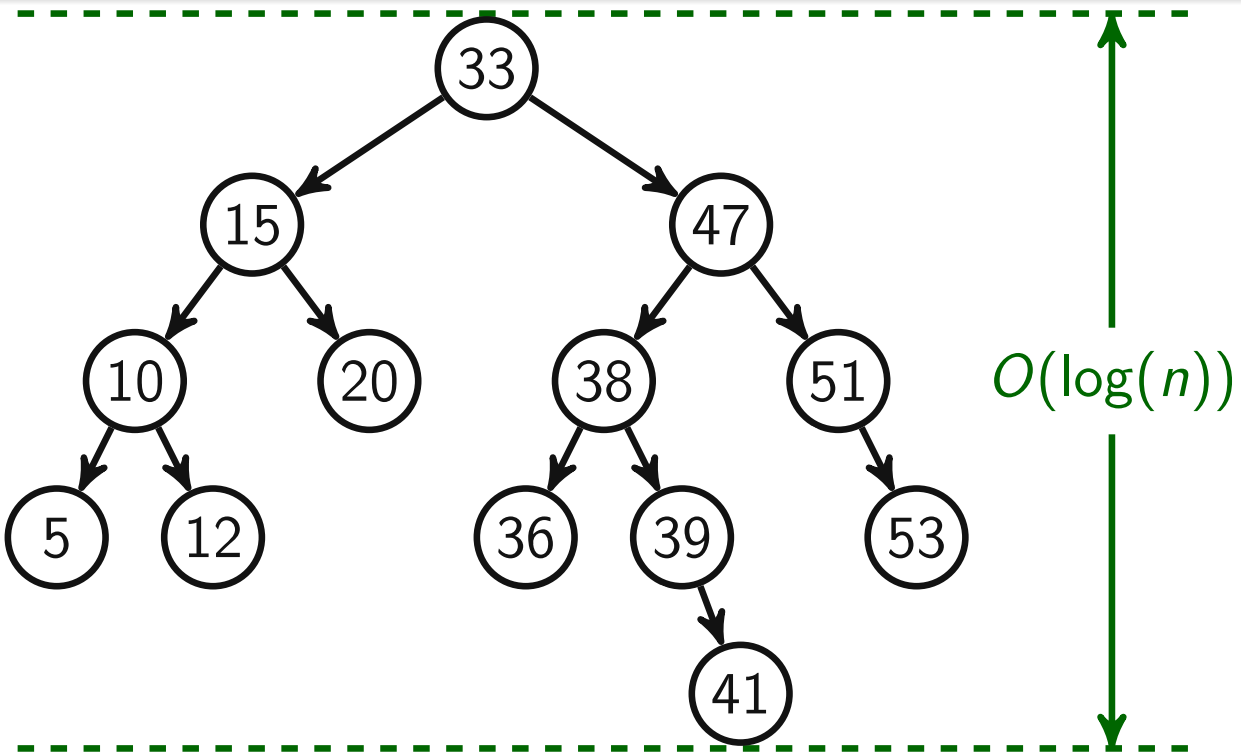
Arbres AVL

Définitions

La **balance** d'un nœud est :

hauteur du sous-abre droit − *hauteur du sous-arbre gauche*

Un **AVL** est un arbre binaire de recherche tel que la balance de chaque nœud est comprise entre -1 et +1

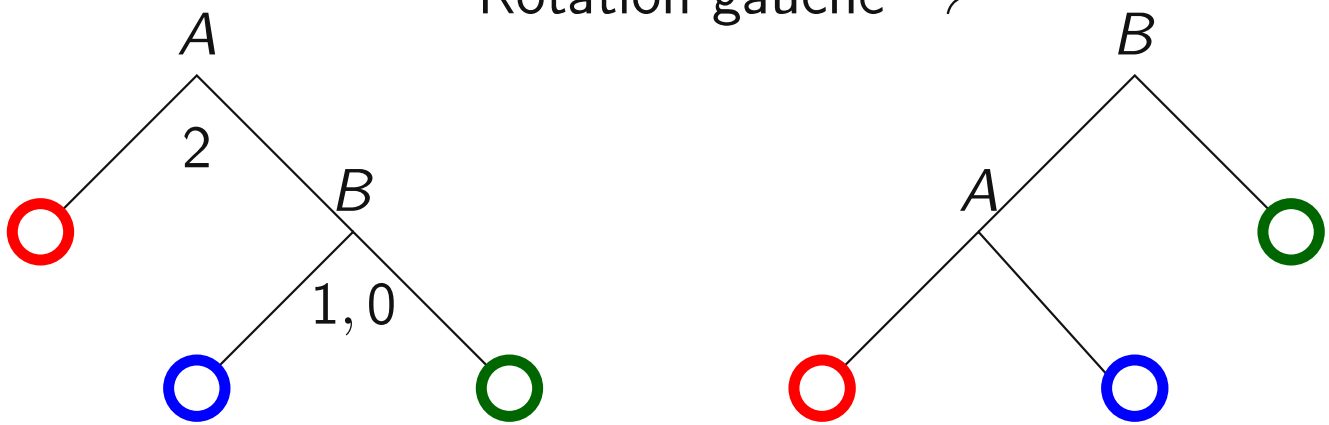


Arbres AVL – fonctionnement

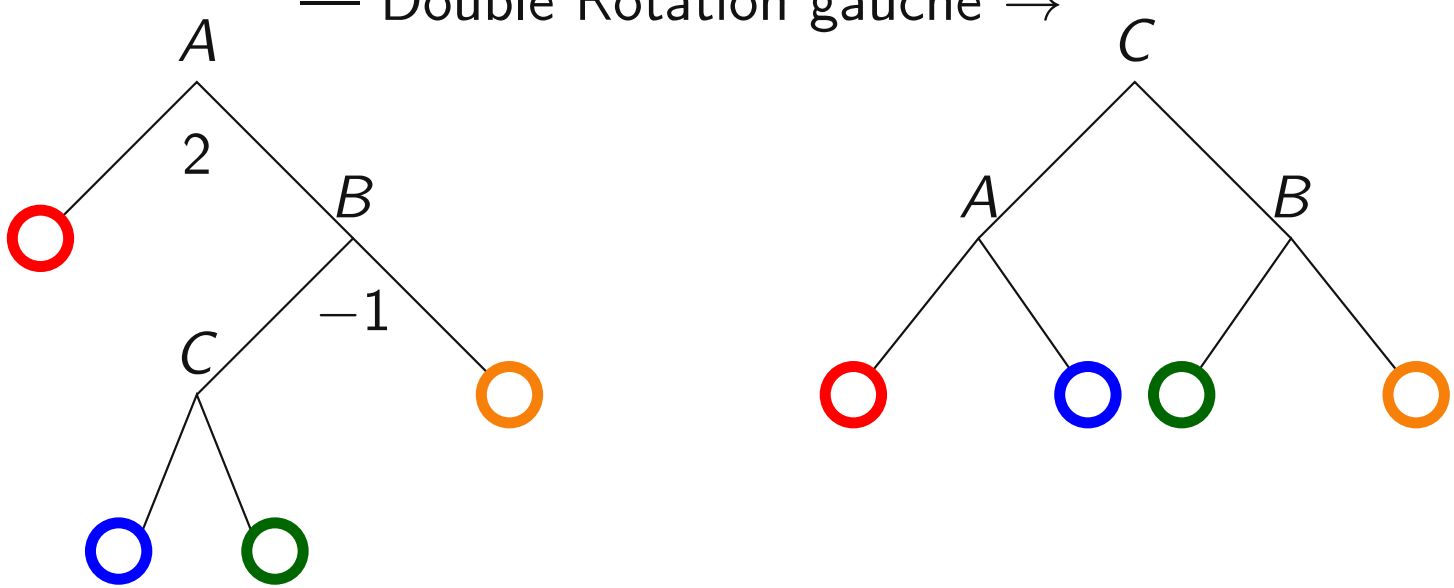
- Chaque nœud connait sa propre balance
- Insertion : comme dans les ABR, **mais :**
  - On met à jour les balances $\rightarrow$ `insert` renvoie 1 si l'hauteur de l'arbre à augmenté, 0 sinon.
  - Si la balance d'un nœud atteint -2 ou +2, on **réorganise localement** ce nœud et ses fils.
  - Plusieurs cas possibles : on parle de **rotations** ou **doubles rotations**, à **gauche** ou à **droite**.
  - Au maximum un seul rééquilibrage est nécessaire par insertion.
- Suppression : idem, mais plusieurs rééquilibrages peuvent être nécessaires.

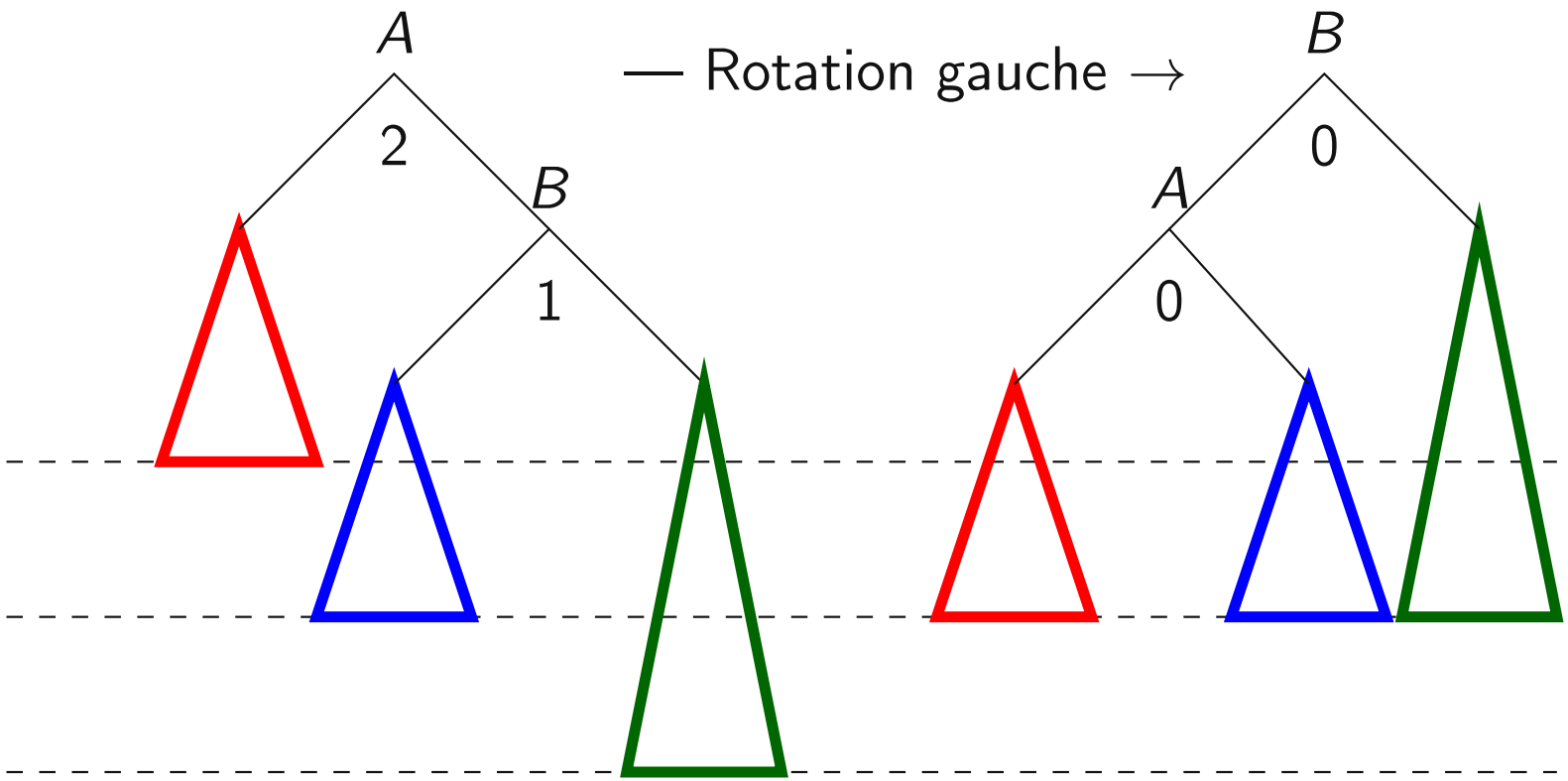
Rotations d'un AVL

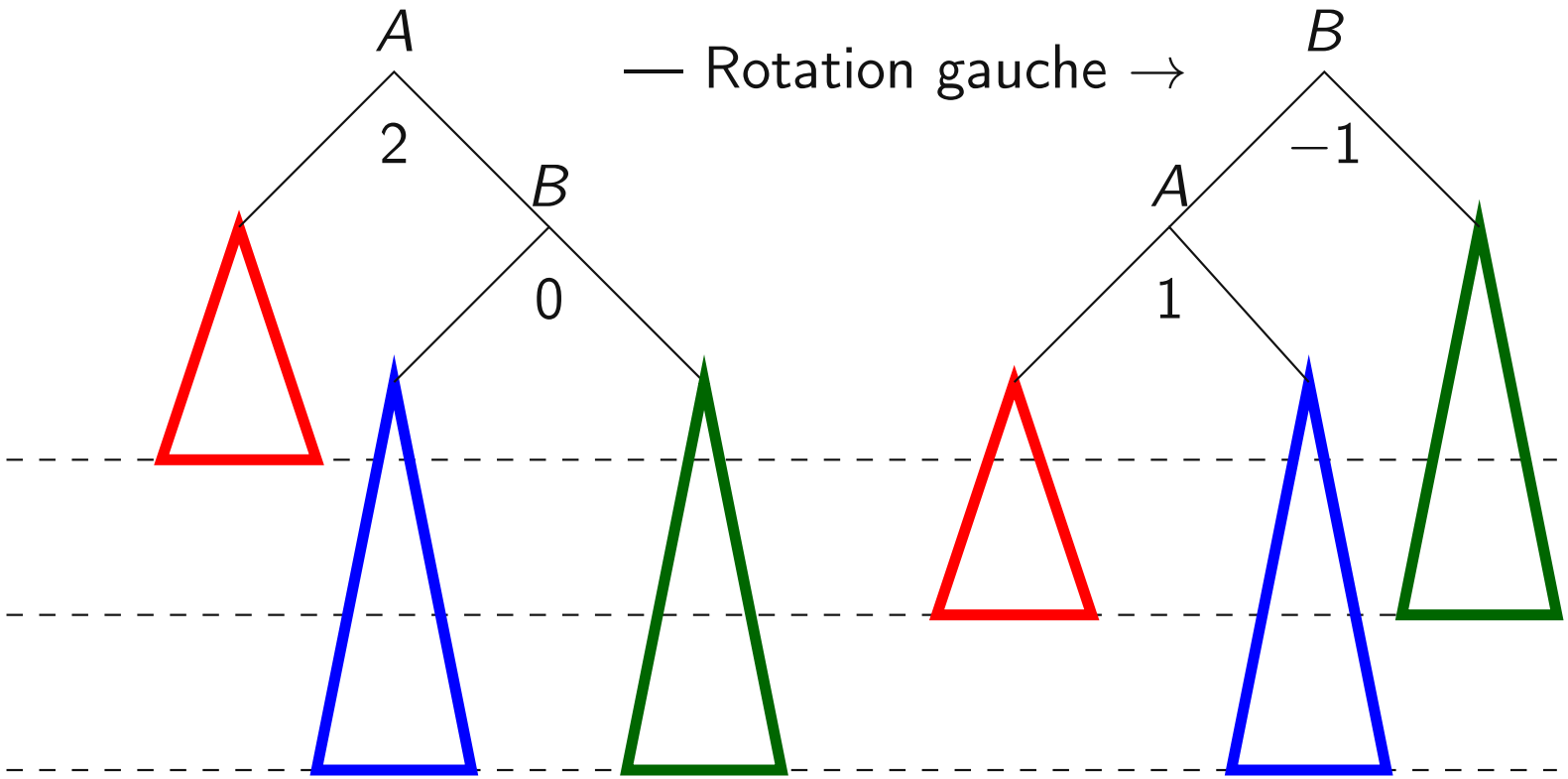
— Rotation gauche →

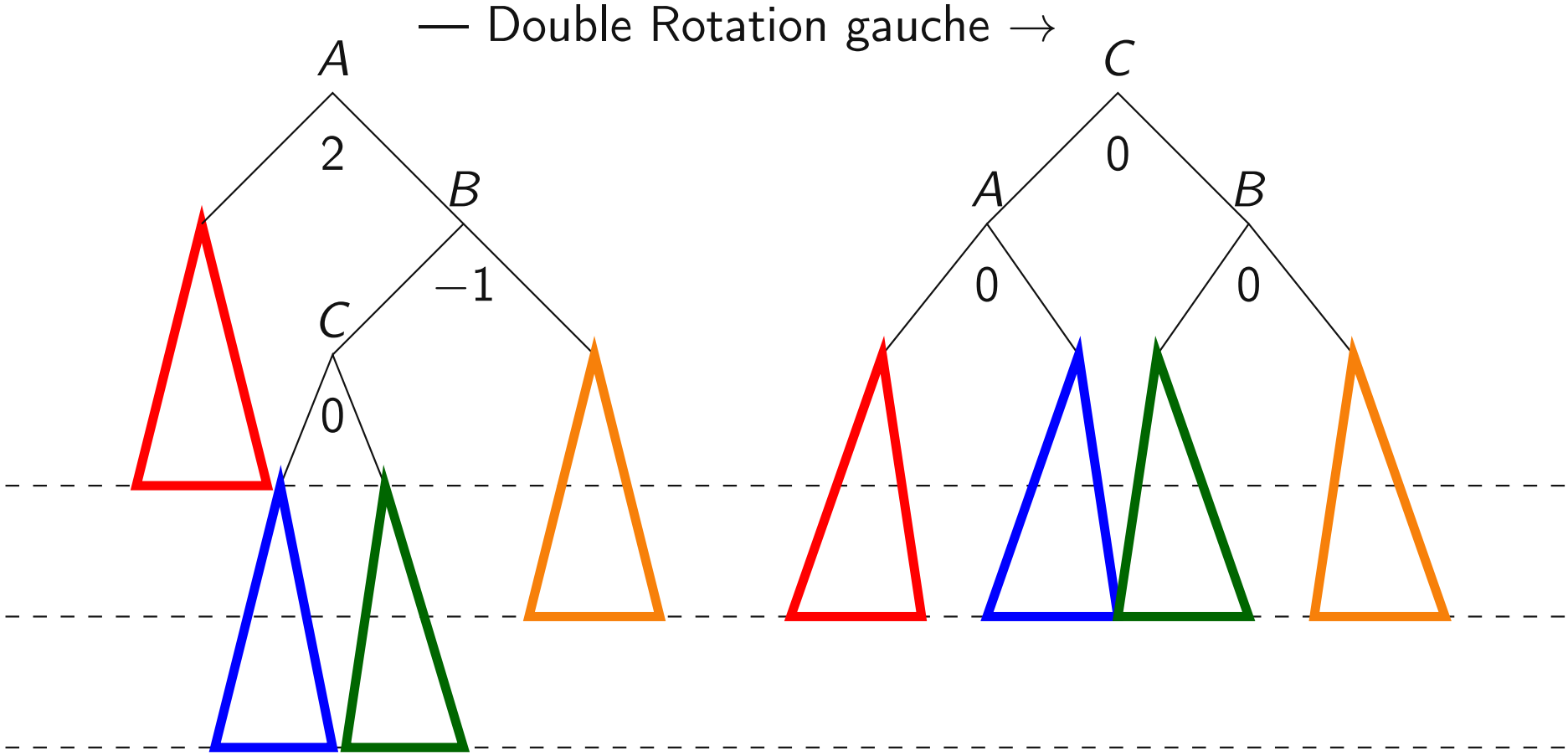


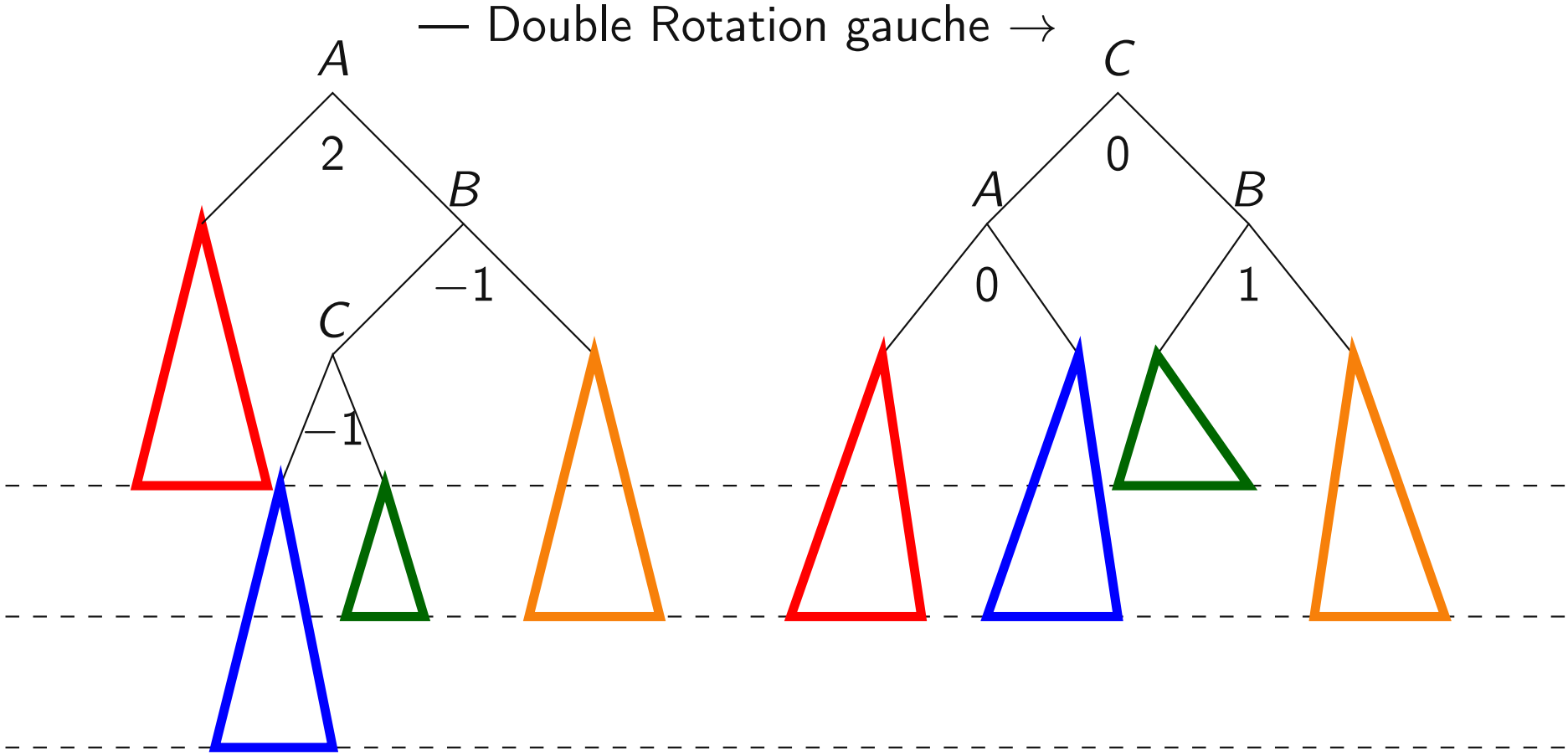
— Double Rotation gauche →

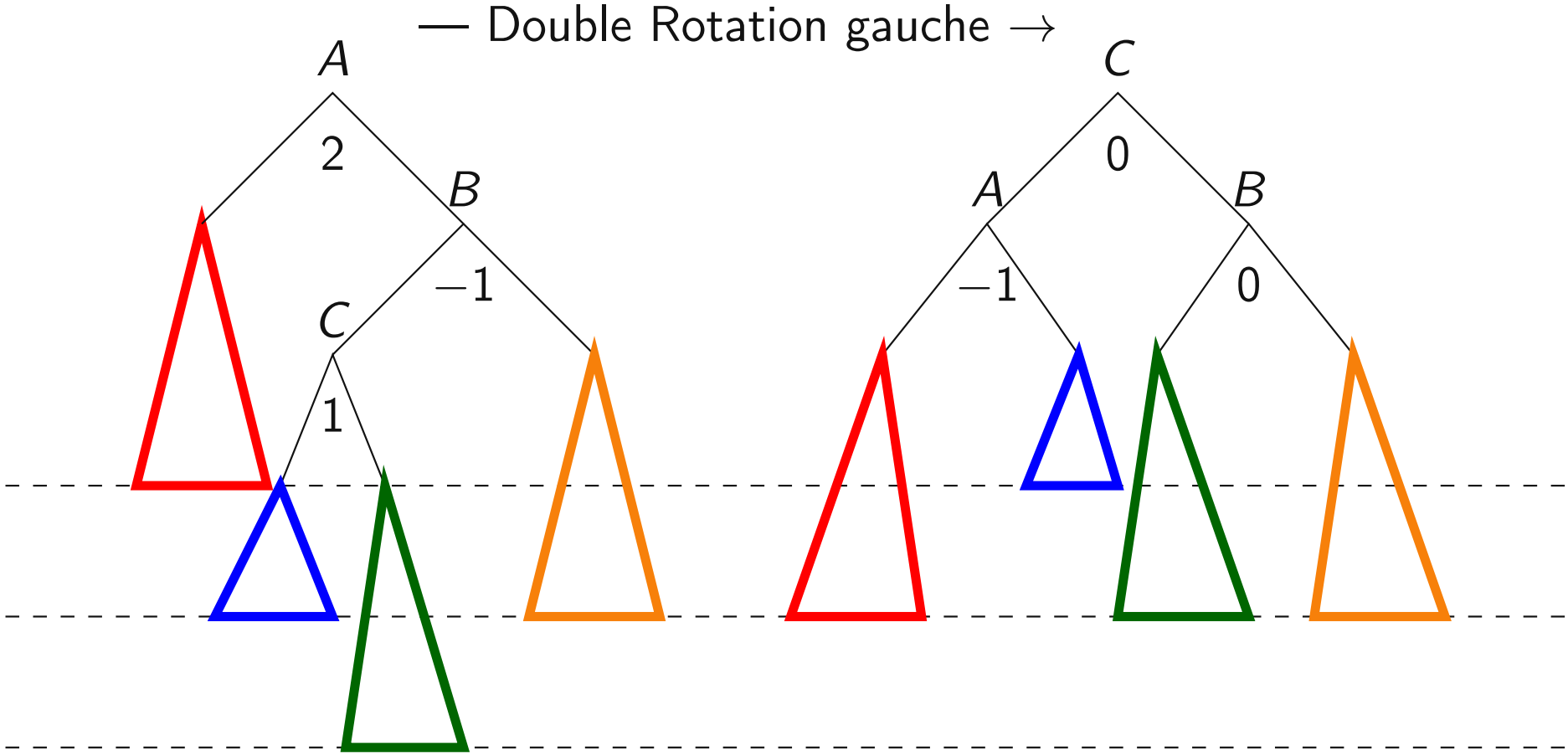



A
2
B
1
— Rotation gauche →
B
0
A
0


A
2
B
0
— Rotation gauche →
B
−1
A
1


— Double Rotation gauche →
A
2
B
−1
C
0
C
0
A
0
B
0

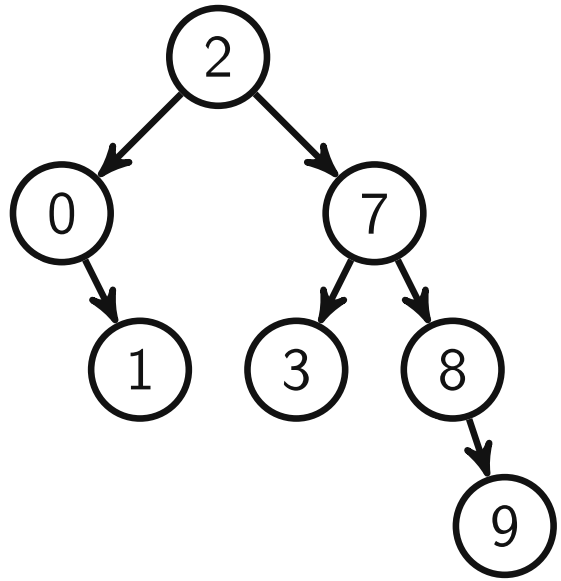

— Double Rotation gauche →
A
2
B
−1
C
−1
C
0
A
0
B
1


— Double Rotation gauche →
A
2
B
−1
C
1
C
0
A
−1
B
0

Exercice 3 – Arbres AVL

1. Calculer la balance de chaque nœud de l'arbre ci-dessous

Calculer les arbres obtenus successivement après les opérations suivantes :



2. Insertion de 5

3. Insertion de 6

4. Insertion de 4