Algorithmique des structures arborescentes

L2 Info et Math-info, 2017–18

Marc Zeitoun

15 février 2018

Moodle

- Nouveaux exercices sur Moodle.
- **Lire** les définitions (polycopié, diapos, **forum**).
- Pour certains exercices, **nombre limité** d'évaluations.

Plan

Plan

Les arbres binaires de recherche (ABR)

- Partant d'une liste **triée**, on obtient un **arbre particulier**.
- En **tout** nœud x :
  - si y est dans le sous-arbre **gauche** de x :

$$\text{étiquette}(y) \leqslant \text{étiquette}(x)$$

  - si y est dans le sous-arbre **droit** de x :

$$\text{étiquette}(y) \geqslant \text{étiquette}(x)$$

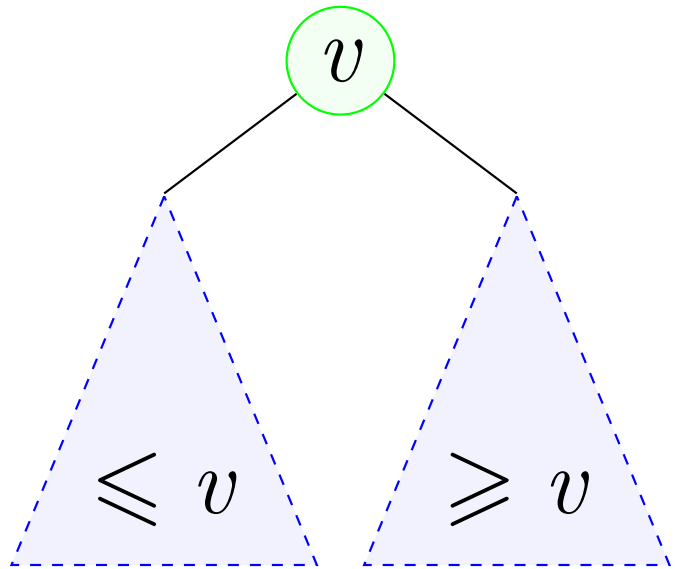
Cette propriété ne se vérifie pas uniquement « localement ».

ABR

La propriété d'être un arbre binaire de recherche n'est **pas locale**.

Pour **chaque nœud** v, elle implique :

- **tous les nœuds** du sous-arbre gauche de v,
- **tous les nœuds** du sous-arbre droit de v.

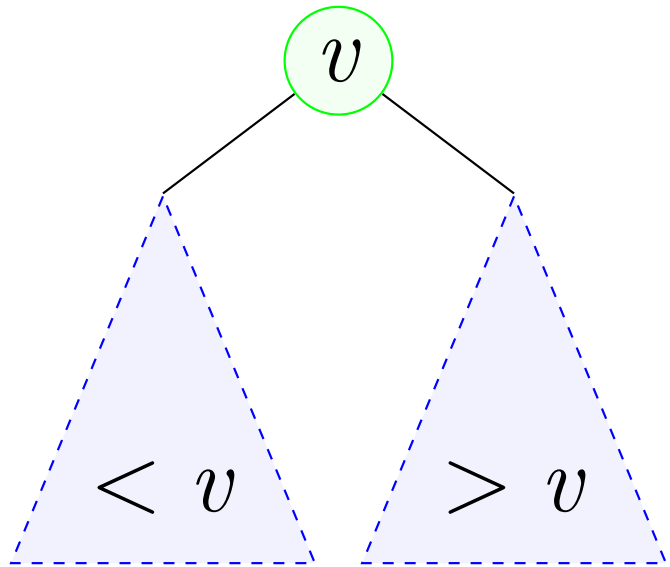


ABR

La propriété d'être un arbre binaire de recherche n'est **pas locale**.

Pour **chaque nœud** v, elle implique :

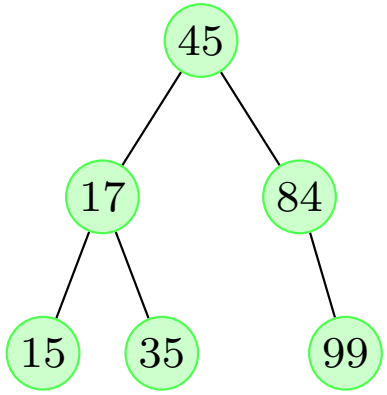
- **tous les nœuds** du sous-arbre gauche de v,
- **tous les nœuds** du sous-arbre droit de v.



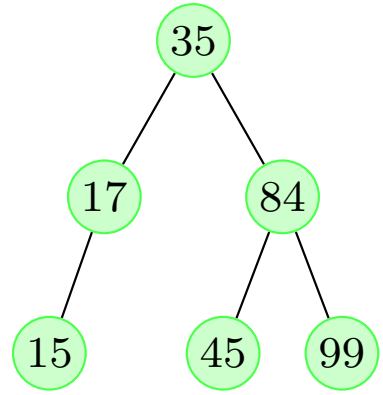
Dans ce cours, 2 nœuds auront toujours des clés différentes

ABR : exemples et non-exemples

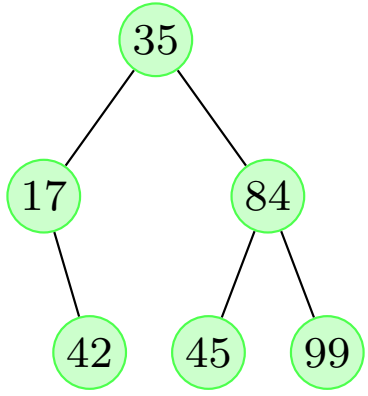
Les 2 premiers ABR représentent le même ensemble de clés.



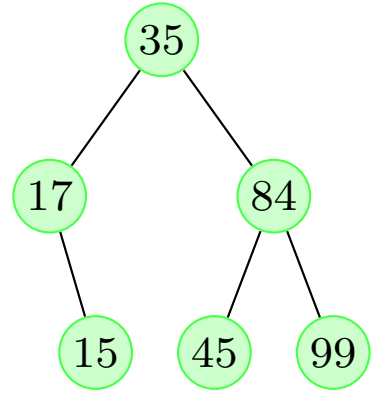
OUI



OUI



NON



NON

Représentation des multi-ensembles par ABR

- « Ensembles » pouvant contenir plusieurs fois la même valeur.
- Multi-ensemble : fonction f d'un ensemble E dans $\mathbb{N}$.

  Intuition $f(e) = 4$ signifie : e est *présent en* 4 *exemplaires*.
- Notation : $\{\!\{1\,,1\,,4\,,5\,,5\,,5\,,7\,,10\,,42\}\!\}$.

Représentation des multi-ensembles par ABR

- « Ensembles » pouvant contenir plusieurs fois la même valeur.
- Multi-ensemble : fonction f d'un ensemble E dans $\mathbb{N}$.

  Intuition $f(e) = 4$ signifie : e est *présent en* 4 *exemplaires*.
- Notation : $\{\!\{1, 1, 4, 5, 5, 5, 7, 10, 42\}\!\}$.
- L'ordre n'est pas important :

$$\{\!\{1, 1, 4, 5, 5, 5, 7, 10, 42\}\!\}$$
$$=$$
$$\{\!\{5, 5, 1, 10, 5, 4, 7, 42, 1\}\!\}$$

Représentation des multi-ensembles par ABR

- « Ensembles » pouvant contenir plusieurs fois la même valeur.
- Multi-ensemble : fonction f d'un ensemble E dans $\mathbb{N}$.

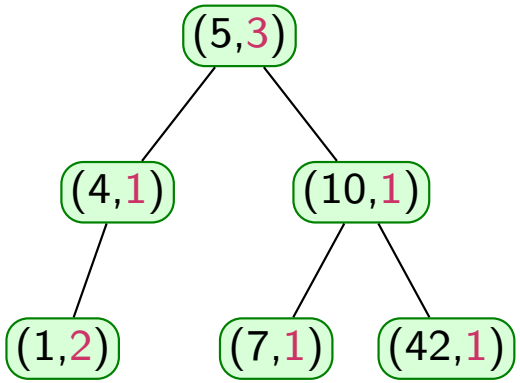
  Intuition $f(e) = 4$ signifie : e est *présent en* 4 *exemplaires*.
- Notation : $\{\!\{1, 1, 4, 5, 5, 5, 7, 10, 42\}\!\}$.
- L'ordre n'est pas important :

$$\{\!\{1, 1, 4, 5, 5, 5, 7, 10, 42\}\!\} = \{\!\{5, 5, 1, 10, 5, 4, 7, 42, 1\}\!\}$$

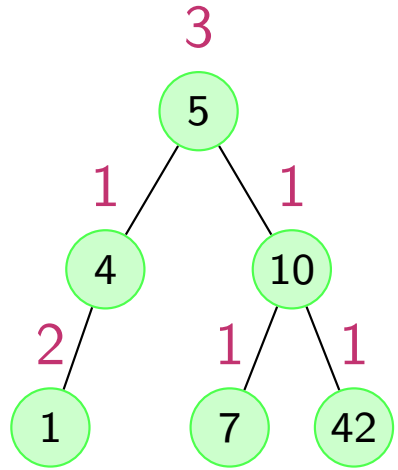
- On utilise les ABR pour représenter les multi-ensembles.
- Élément e en 4 exemplaires $\rightarrow$ nœud d'étiquette $(e, 4)$.

  $\Rightarrow$ Aucune clé n'apparaît dans plusieurs nœuds.

Notations



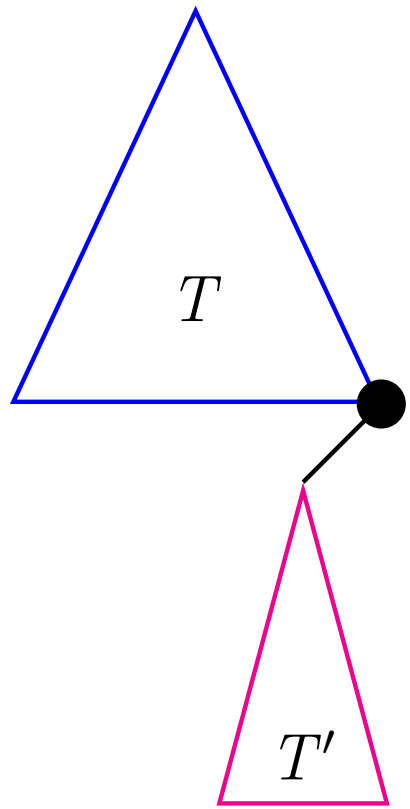
Couples avec clés et multiplicité



Multiplicité au dessus des nœuds

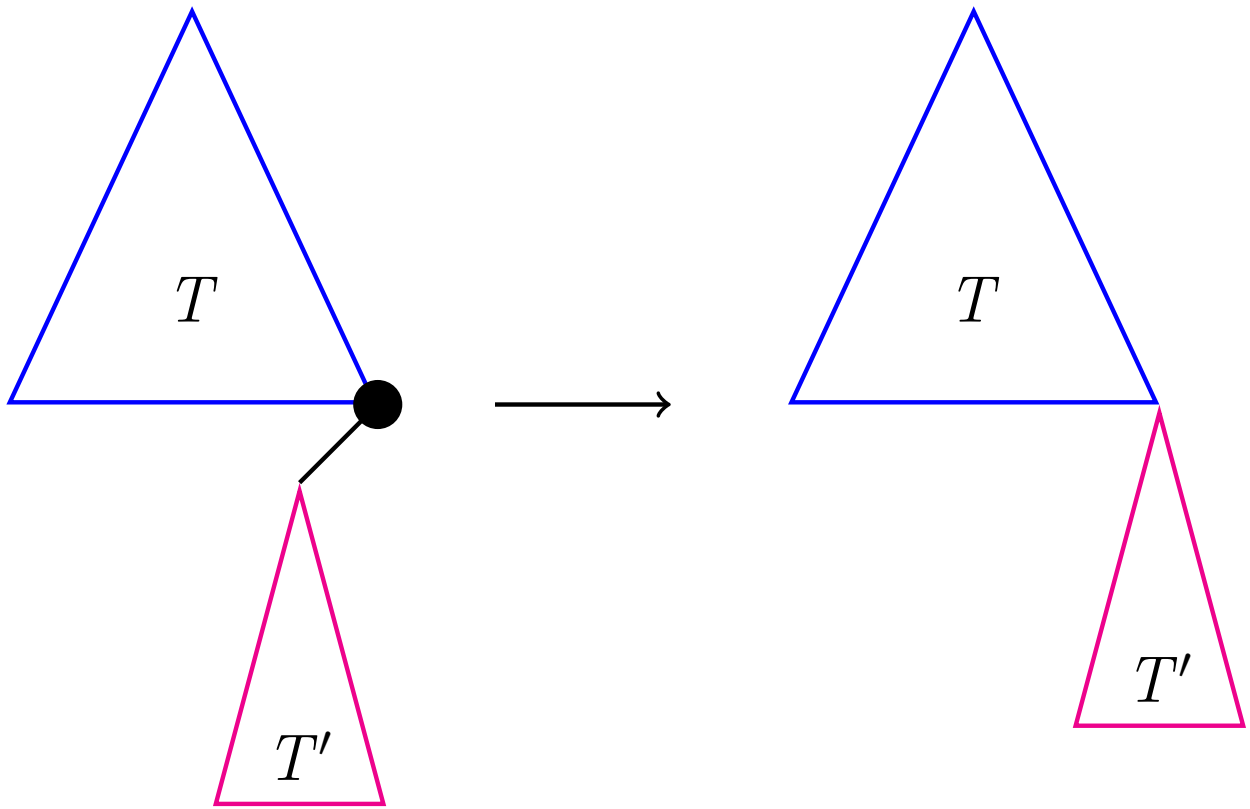
Élément maximum dans un ABR non vied

- Se trouve en parcourant l'arbre depuis la racine « vers la droite ».
- Le nœud portant la clé maximale n'a pas de fils droit.



Suppression de l'élément maximum dans un ABR

- En $O(1)$ une fois le nœud portant la clé maximale localisé.



Recherche d'un élément x

Utilise la propriété d'être un arbre binaire ***de recherche***.

- Si l'arbre est vide : x non présent dans l'arbre.
- Si $x =$ clé(racine) : trouvé.
- Si $x <$ clé(racine) : recherche récursive dans le sous-arbre gauche.
- Si $x >$ clé(racine) : recherche récursive dans le sous-arbre droit.

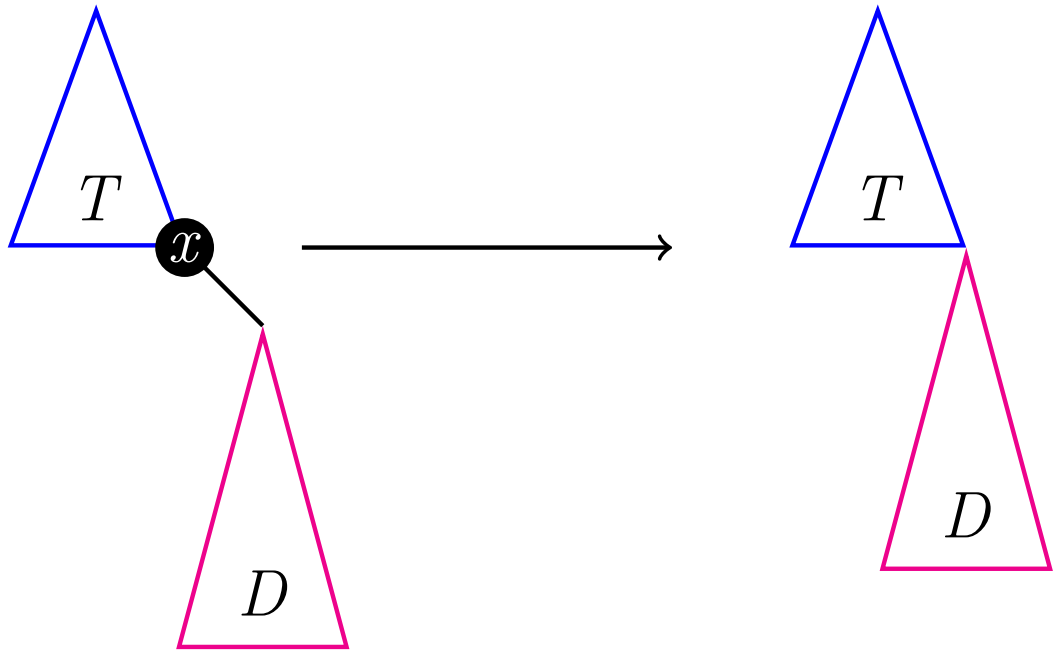
Insertion d'un élément

- Deux algorithmes classiques : par création d'une nouvelle feuille ou par ajout d'une nouvelle racine.
- **Dans ce cours** : par création d'une nouvelle feuille.
- Principe : naviguer jusqu'à
  - soit trouver un nœud portant la clé à insérer (et incrémenter sa multiplicité),
  - soit arriver jusqu'à un sous-arbre vide, et créer une feuille à cet emplacement.

Suppression d'un élément x de multiplicité 1

Principe :

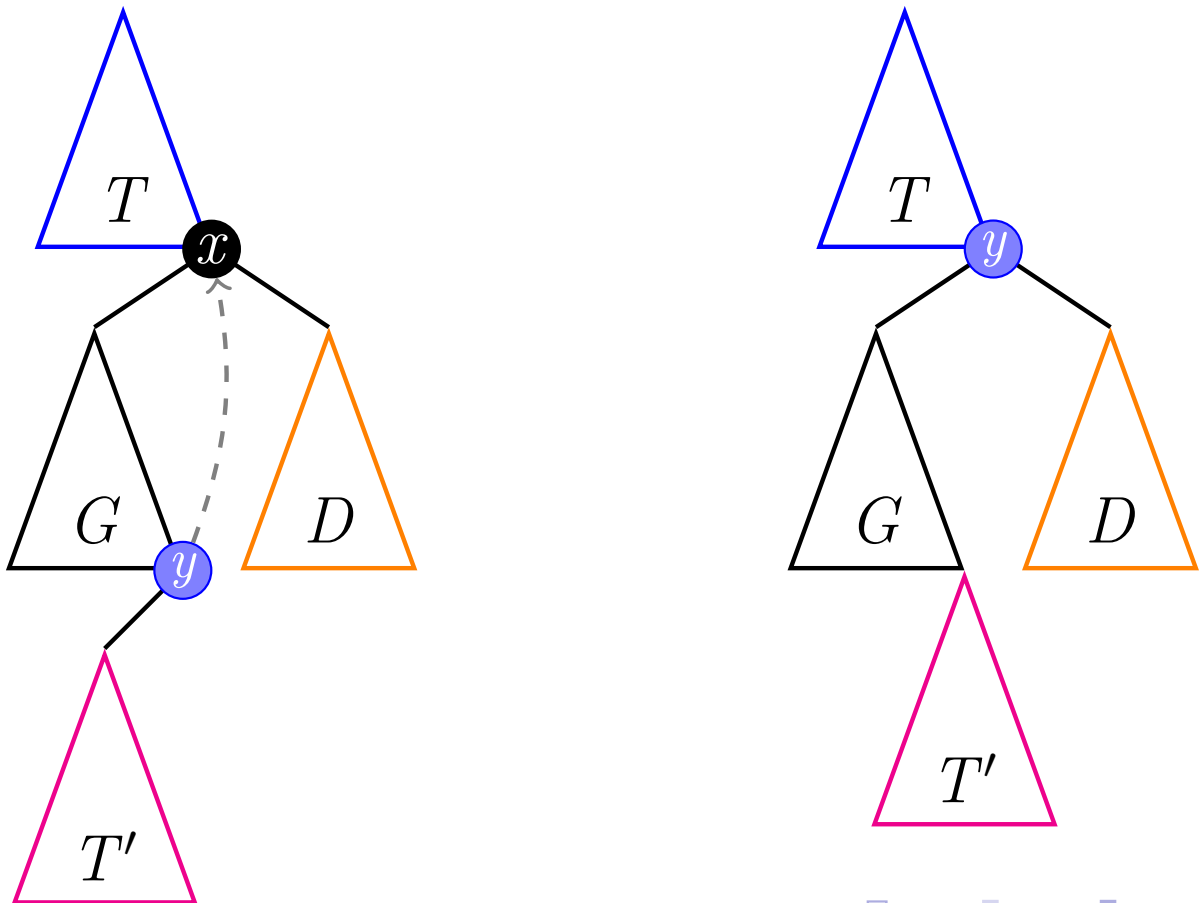
- Rechercher le nœud portant la clé x à supprimer.
- 1er cas : son sous-arbre gauche est vide :



Suppression d'un élément x de multiplicité 1

Principe :

- 2e cas : son sous-arbre gauche est **non** vide : échange avec le maximum y de son sous-arbre gauche, facile à supprimer. **Conserve** la propriété ABR.



Plan

ABR : Complexité recherche/insertion/suppression

- Complexité de recherche/insertion/suppression dans un ABR :

ABR : Complexité recherche/insertion/suppression

- Complexité de recherche/insertion/suppression dans un ABR : $O(h)$ où h est la **hauteur** de l'arbre.

ABR : Complexité recherche/insertion/suppression

- Complexité de recherche/insertion/suppression dans un ABR :

$$O(h) \text{ où } h \text{ est la } \textbf{hauteur} \text{ de l'arbre.}$$

Si n est le nombre de nœuds,

- Au pire, $h =$

ABR : Complexité recherche/insertion/suppression

- Complexité de recherche/insertion/suppression dans un ABR :

$$O(h) \text{ où } h \text{ est la } \textbf{hauteur} \text{ de l'arbre.}$$

Si n est le nombre de nœuds,

- Au pire, $h = n - 1$.
- Au mieux, $h =$

ABR : Complexité recherche/insertion/suppression

- Complexité de recherche/insertion/suppression dans un ABR :

$$O(h) \text{ où } h \text{ est la } \textbf{hauteur} \text{ de l'arbre.}$$

Si n est le nombre de nœuds,

- Au pire, $h = n - 1$.
- Au mieux, $h = \log_2(n+1) - 1$.

Arbres équilibrés

- Maintenir dans les ABR

$$h = \alpha \log(n)$$

garantirait une complexité $O(\log(n))$ **pour ces opérations**.

Arbres équilibrés

- Maintenir dans les ABR

$$h = \alpha \log(n)$$

garantirait une complexité $O(\log(n))$ **pour ces opérations**.

Nouvel objectif

Maintenir $h = \alpha \log(n)$ pour un nombre $\alpha > 0$.

Plan

Les arbres rouges et noirs

Un **arbre rouge et noir** est un ABR tel que :

1. il est **plein**

Les arbres rouges et noirs

Un **arbre rouge et noir** est un ABR tel que :

1. il est **plein** (pas de nœud d'arité 1),

Les arbres rouges et noirs

Un **arbre rouge et noir** est un ABR tel que :

1. il est **plein** (pas de nœud d'arité 1),
2. seuls les nœuds internes portent des valeurs,

Les arbres rouges et noirs

Un **arbre rouge et noir** est un ABR tel que :

1. il est **plein** (pas de nœud d'arité 1),
2. seuls les nœuds internes portent des valeurs,
3. chaque nœud est soit rouge, soit noir,

Les arbres rouges et noirs

Un **arbre rouge et noir** est un ABR tel que :

1. il est **plein** (pas de nœud d'arité 1),
2. seuls les nœuds internes portent des valeurs,
3. chaque nœud est soit rouge, soit noir,
4. la racine est noire,

Les arbres rouges et noirs

Un **arbre rouge et noir** est un ABR tel que :

1. il est **plein** (pas de nœud d'arité 1),
2. seuls les nœuds internes portent des valeurs,
3. chaque nœud est soit rouge, soit noir,
4. la racine est noire,
5. les feuilles sont noires,

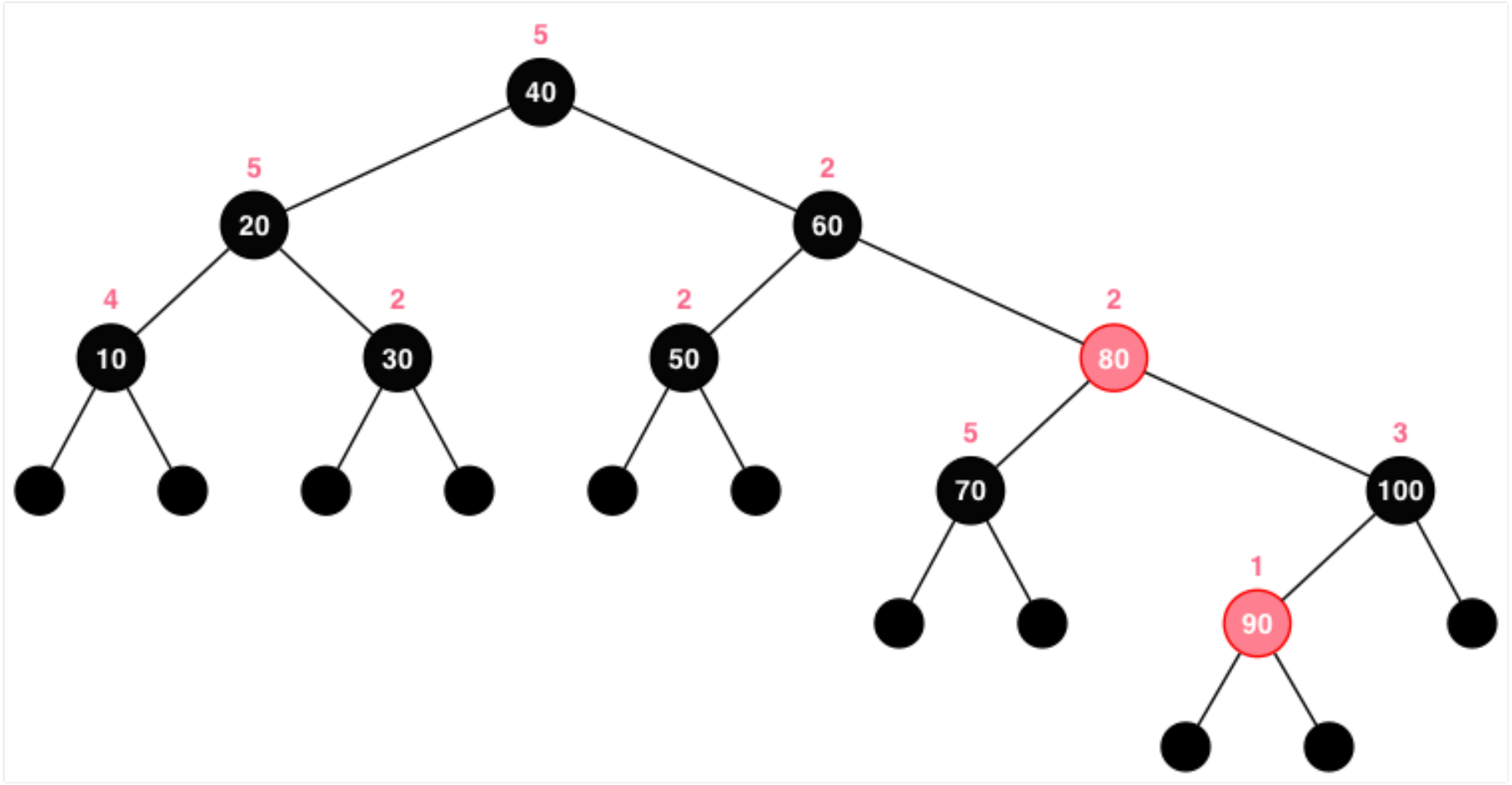
Les arbres rouges et noirs

Un **arbre rouge et noir** est un ABR tel que :

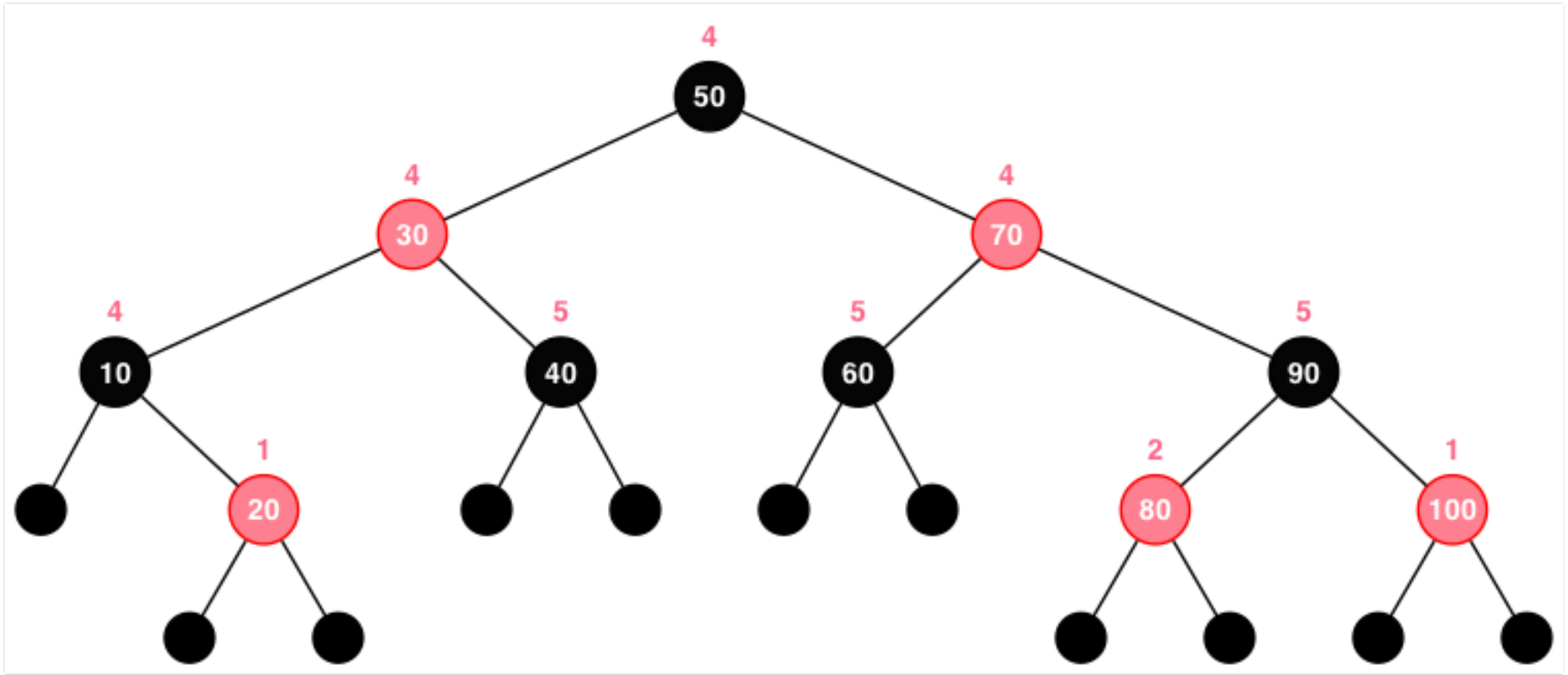
1. il est **plein** (pas de nœud d'arité 1),
2. seuls les nœuds internes portent des valeurs,
3. chaque nœud est soit rouge, soit noir,
4. la racine est noire,
5. les feuilles sont noires,
6. le père d'un nœud rouge est noir.
7. le nombre de nœuds noirs sur chaque branche est **constant**.

Hauteur noire = nombre de nœuds noirs sur chaque branche.

Arbres rouges et noirs : exemple 1



Arbres rouges et noirs : exemple 2



Plan

Les arbres rouges et noirs sont équilibrés

Preuve 1 : sans récurrence. Soit t un arbre rouge et noir.

- $hn =$ hauteur noire de t, $h =$ hauteur, $n =$ nombre de nœuds.
- L'arbre et plein et toutes les branches ont au moins hn nœuds.

  Donc tous les niveaux sont complets jusqu'à profondeur $hn - 1$
- Donc il y a au minimum $2^{hn} - 1$ nœuds : $\boldsymbol{hn \leqslant \log_2(n+1).}$
- Une branche a au maximum hn nœuds noirs et $hn - 1$ nœuds rouges : $\bullet - \bullet - \bullet - \bullet - \cdots - \bullet - \bullet - \bullet$.
- Donc $h \leqslant 2(hn - 1)$ et en utilisant l'inégalité ci-dessus :

$$h \leqslant 2(\log_2(n+1) - 1).$$

Plan

Insertion dans un arbre rouge-noir

Un algorithme d'insertion dans un arbre rouge-noir :

- On insère comme dans un ABR,
- Si création d'un nouveau nœud,
  - il remplace l'une des anciennes feuilles noires,
  - on le colore en **rouge**,
  - ses deux fils sont des feuilles (noires).

- **Problème potentiel**

Insertion dans un arbre rouge-noir

Un algorithme d'insertion dans un arbre rouge-noir :

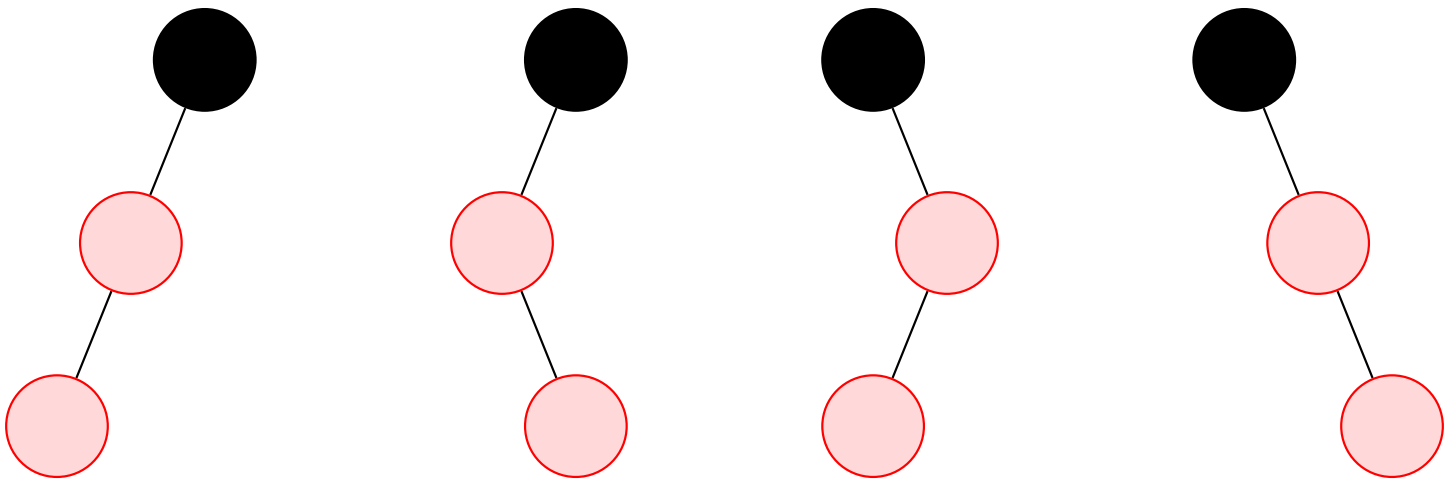
- On insère comme dans un ABR,
- Si création d'un nouveau nœud,
  - il remplace l'une des anciennes feuilles noires,
  - on le colore en **rouge**,
  - ses deux fils sont des feuilles (noires).

- **Problème potentiel**
  - l'arbre obtenu peut avoir 2 nœuds rouges père-fils.

Insertion dans un arbre rouge-noir

Réorganisation de l'arbre pour

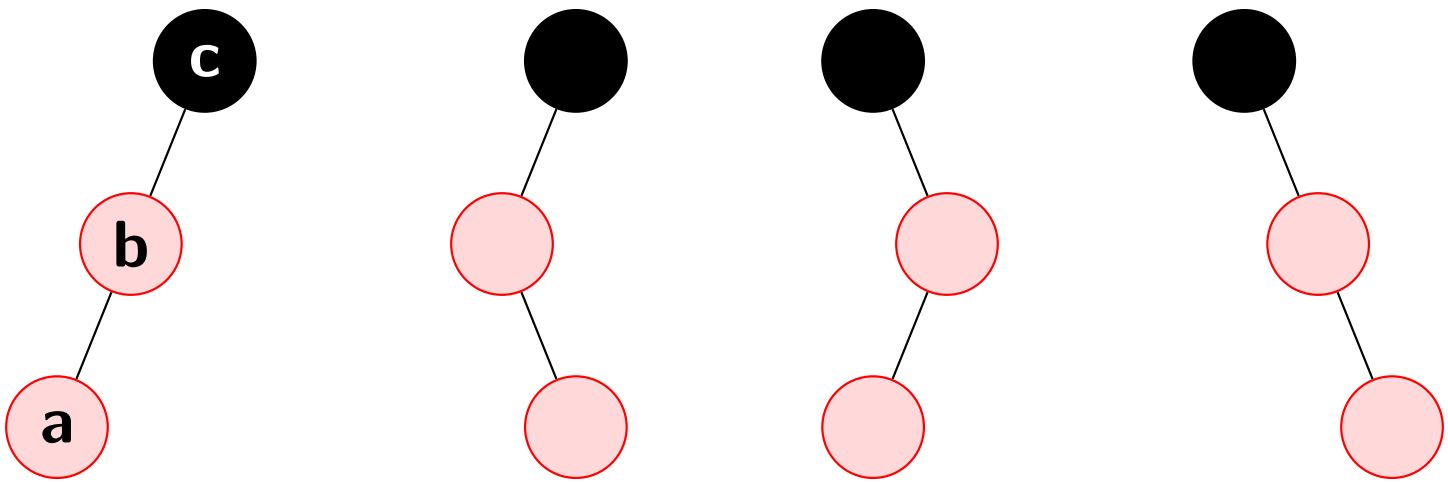
- éviter 2 nœuds rouges consécutifs.
- conserver les autres propriétés des arbres rouges et noirs.
- Après insertion : 4 cas possibles avec $\mathbf{a} < \mathbf{b} < \mathbf{c}$:



Insertion dans un arbre rouge-noir

Réorganisation de l'arbre pour

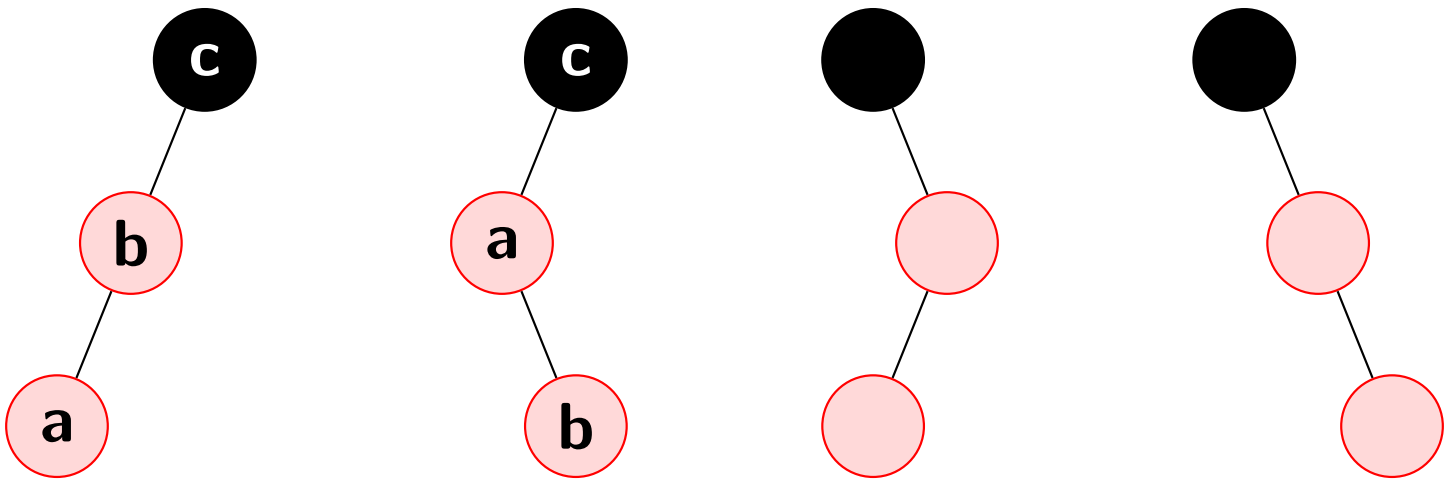
- éviter 2 nœuds rouges consécutifs.
- conserver les autres propriétés des arbres rouges et noirs.
- Après insertion : 4 cas possibles avec $\mathbf{a} < \mathbf{b} < \mathbf{c}$:



Insertion dans un arbre rouge-noir

Réorganisation de l'arbre pour

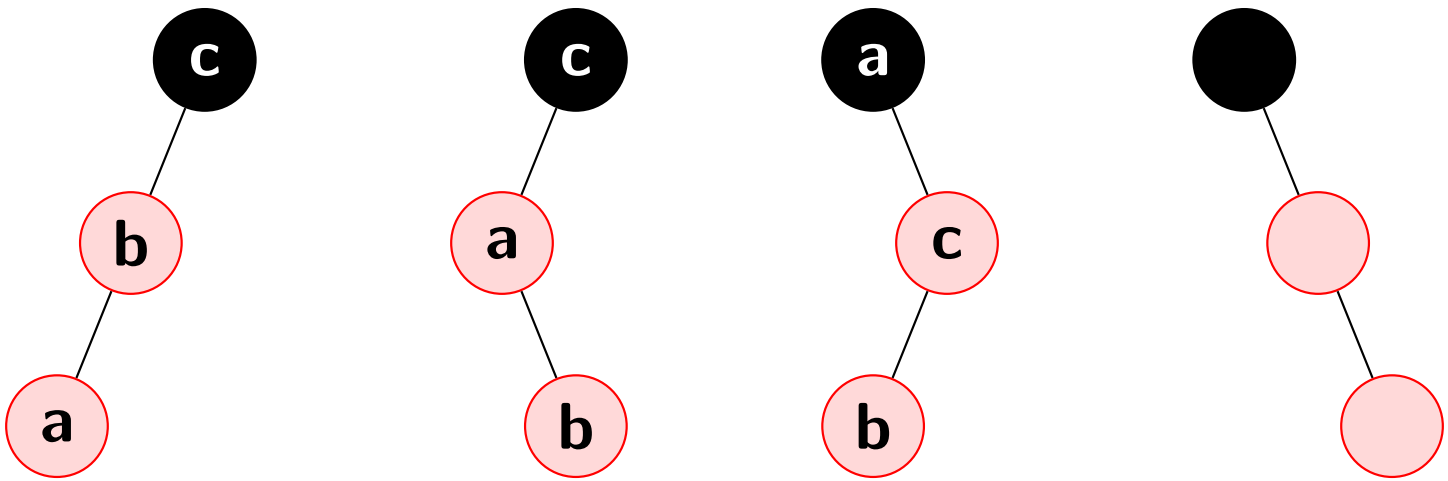
- éviter 2 nœuds rouges consécutifs.
- conserver les autres propriétés des arbres rouges et noirs.
- Après insertion : 4 cas possibles avec $\mathbf{a} < \mathbf{b} < \mathbf{c}$:



Insertion dans un arbre rouge-noir

Réorganisation de l'arbre pour

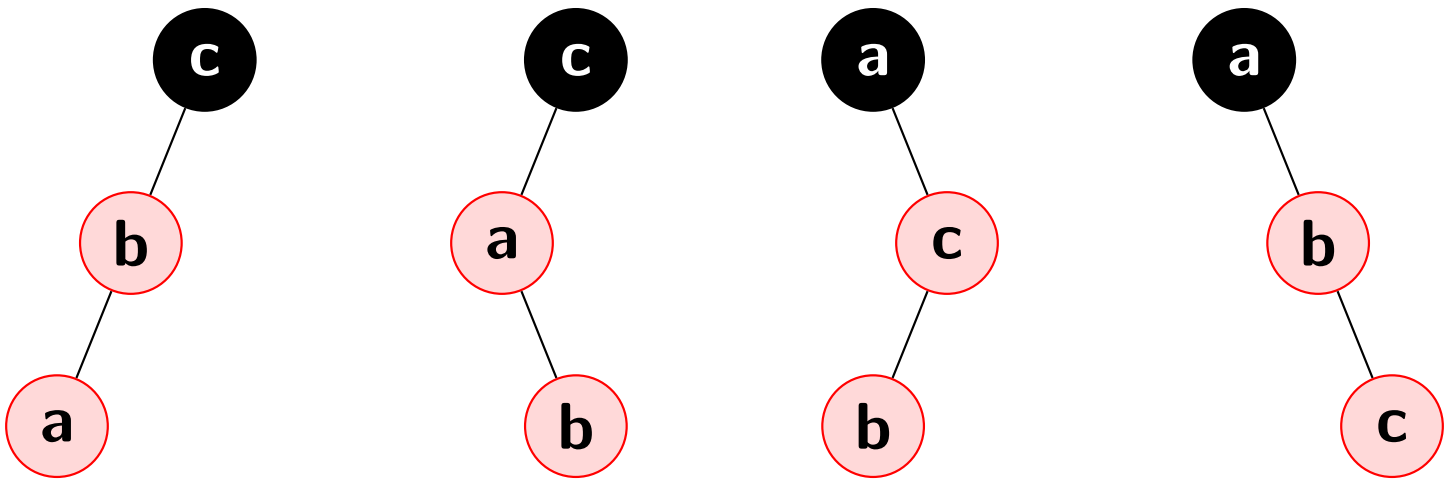
- éviter 2 nœuds rouges consécutifs.
- conserver les autres propriétés des arbres rouges et noirs.
- Après insertion : 4 cas possibles avec $\mathbf{a} < \mathbf{b} < \mathbf{c}$:



Insertion dans un arbre rouge-noir

Réorganisation de l'arbre pour

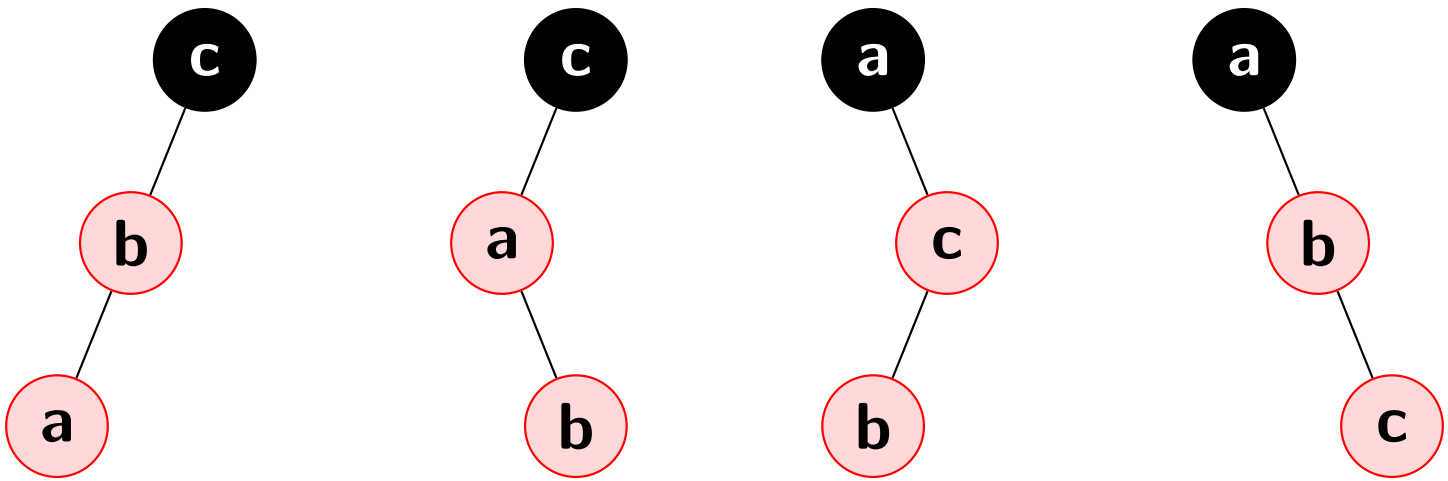
- éviter 2 nœuds rouges consécutifs.
- conserver les autres propriétés des arbres rouges et noirs.
- Après insertion : 4 cas possibles avec $\mathbf{a} < \mathbf{b} < \mathbf{c}$:



Insertion dans un arbre rouge-noir

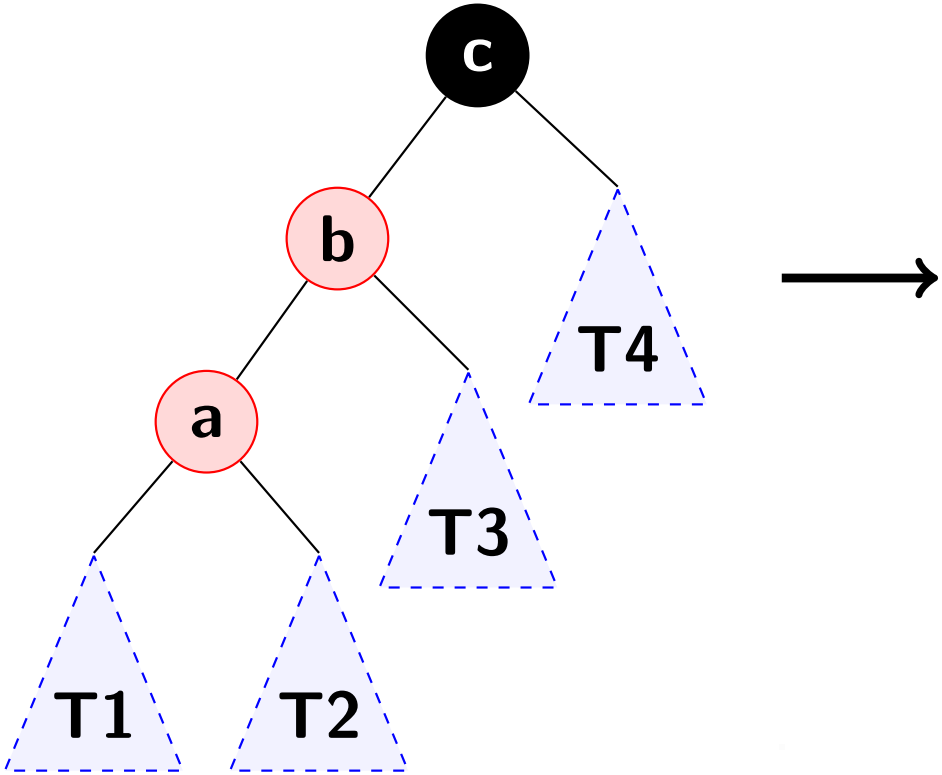
Réorganisation de l'arbre pour

- éviter 2 nœuds rouges consécutifs.
- conserver les autres propriétés des arbres rouges et noirs.
- Après insertion : 4 cas possibles avec $\mathbf{a} < \mathbf{b} < \mathbf{c}$:



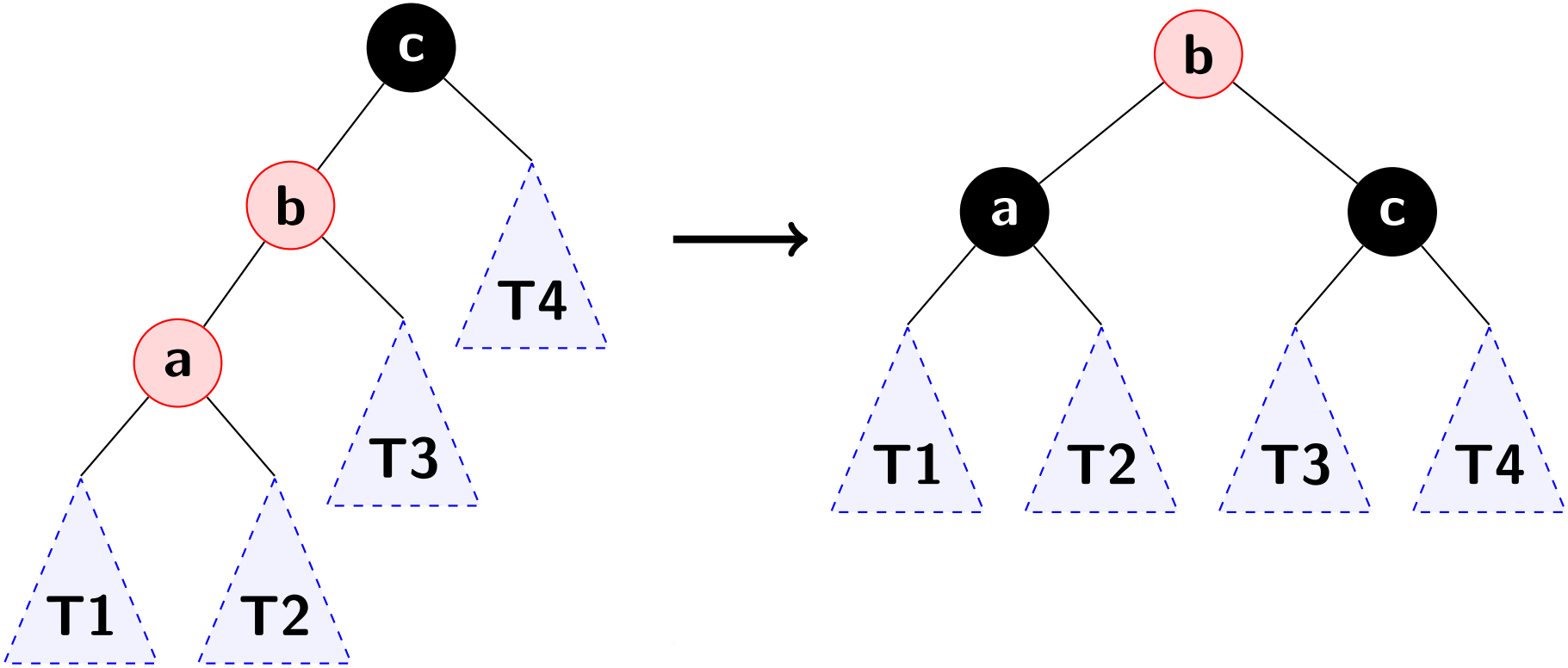
On applique une **transformation** dans chacun des 4 cas.

Insertion dans un arbre rouge-noir : cas 1



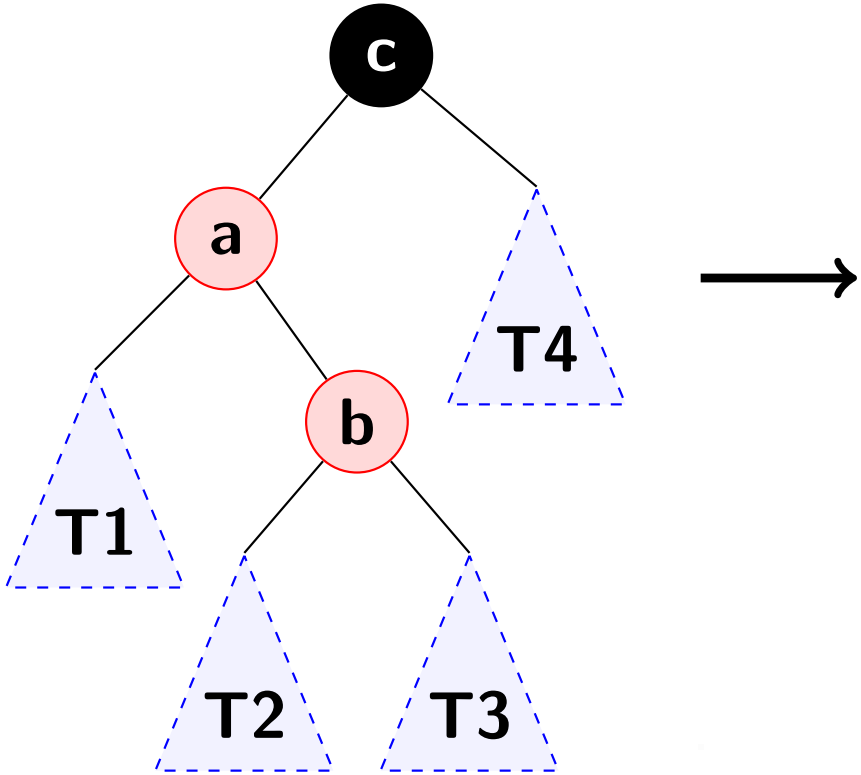
$$\textbf{cles(T1)} < \textbf{a} < \textbf{cles(T2)} < \textbf{b} < \textbf{cles(T3)} < \textbf{c} < \textbf{cles(T4)}$$

Insertion dans un arbre rouge-noir : cas 1



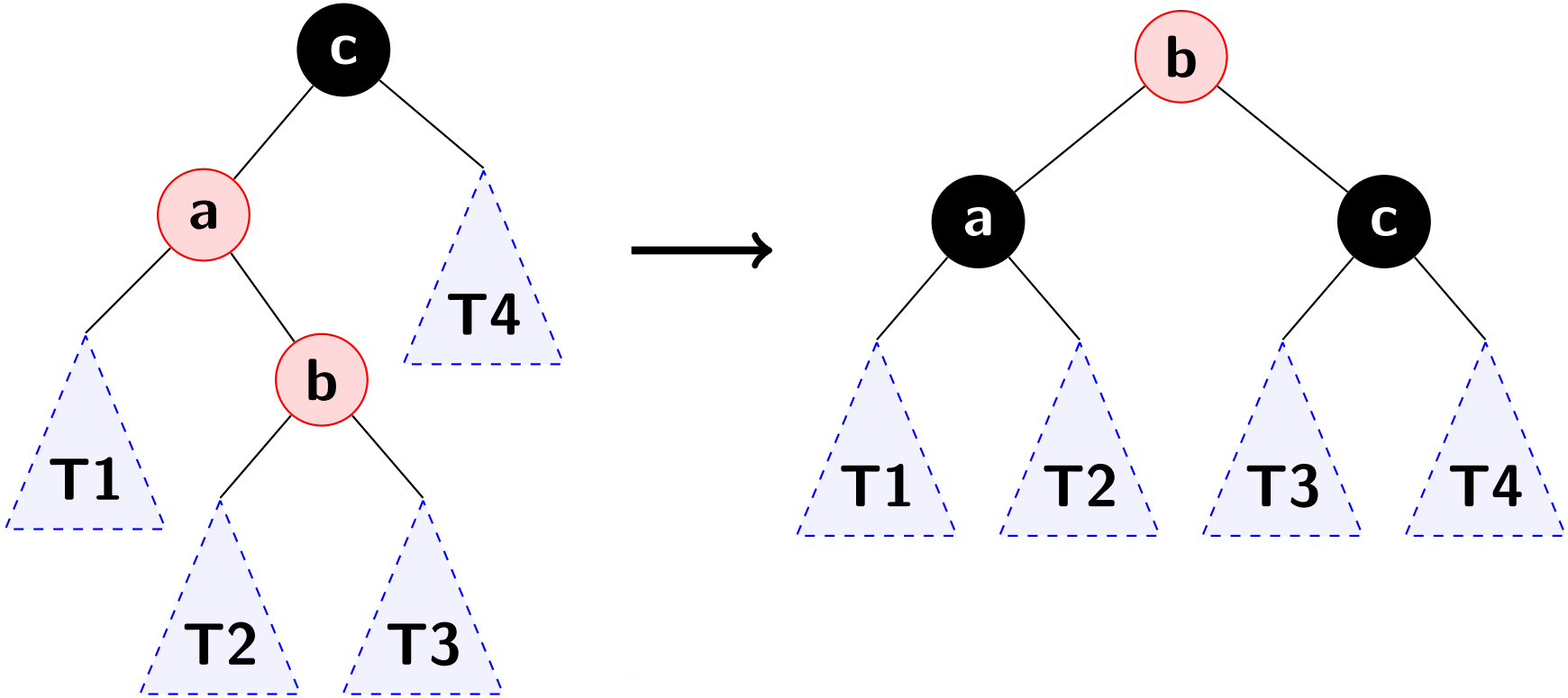
$$\mathbf{cles(T1) < a < cles(T2) < b < cles(T3) < c < cles(T4)}$$

Insertion dans un arbre rouge-noir : cas 2



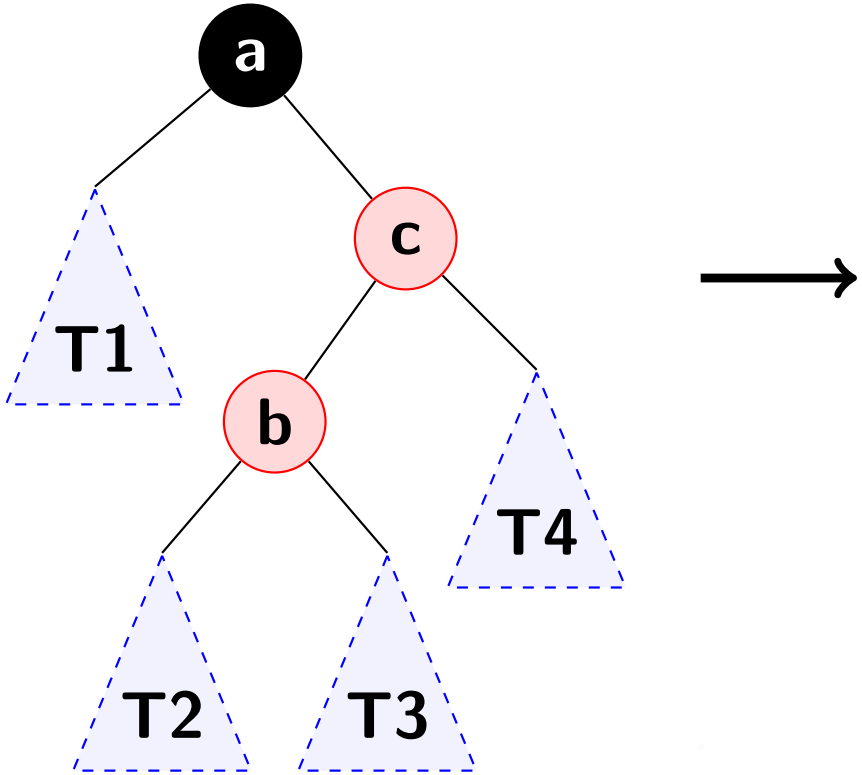
$$\mathbf{cles(T1) < a < cles(T2) < b < cles(T3) < c < cles(T4)}$$

Insertion dans un arbre rouge-noir : cas 2



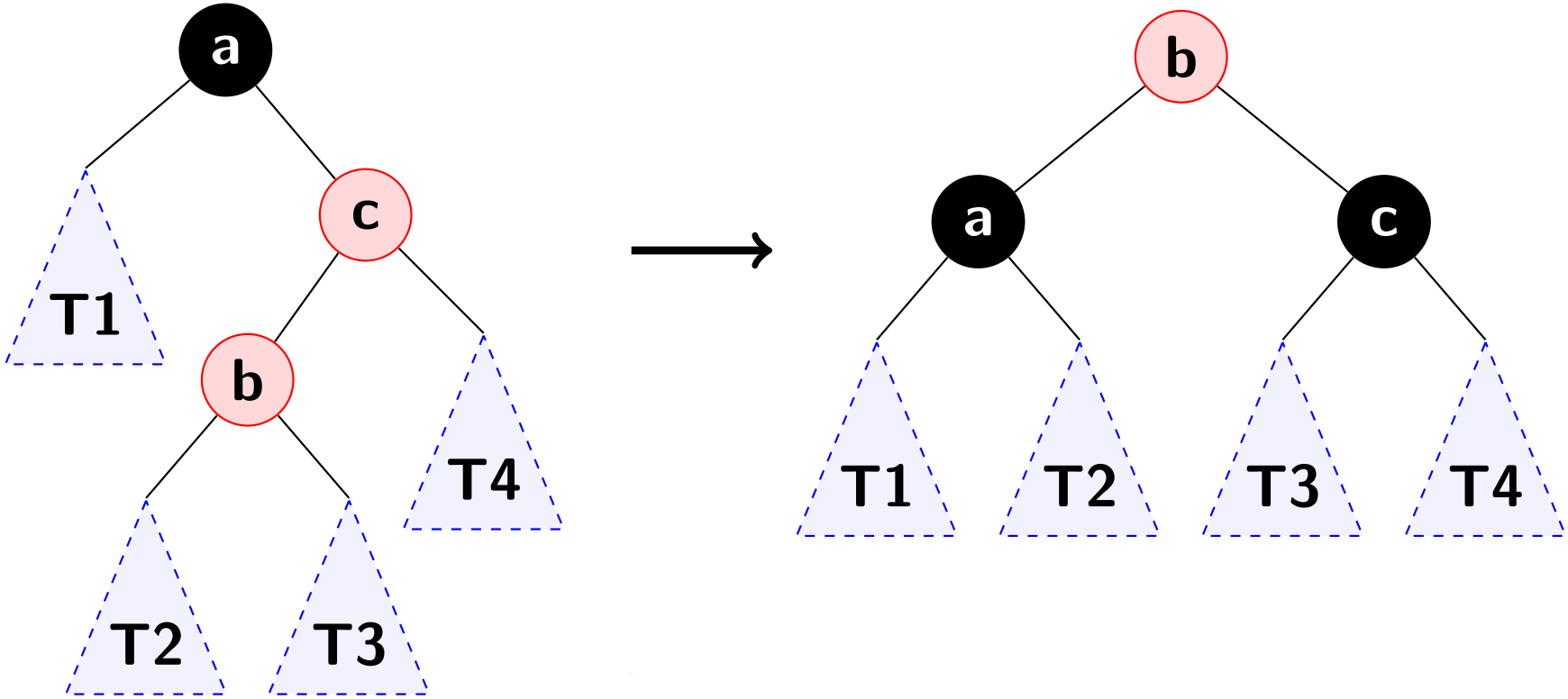
$$\mathbf{cles(T1)} < \mathbf{a} < \mathbf{cles(T2)} < \mathbf{b} < \mathbf{cles(T3)} < \mathbf{c} < \mathbf{cles(T4)}$$

Insertion dans un arbre rouge-noir : cas 3



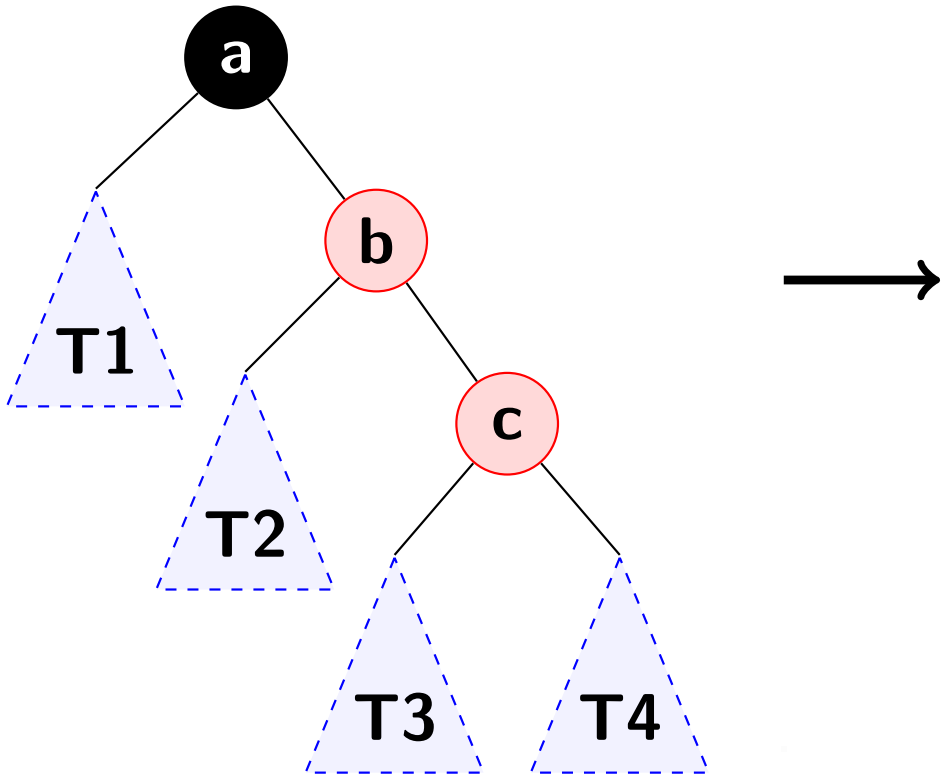
$$\textbf{cles(T1)} < \textbf{a} < \textbf{cles(T2)} < \textbf{b} < \textbf{cles(T3)} < \textbf{c} < \textbf{cles(T4)}$$

Insertion dans un arbre rouge-noir : cas 3



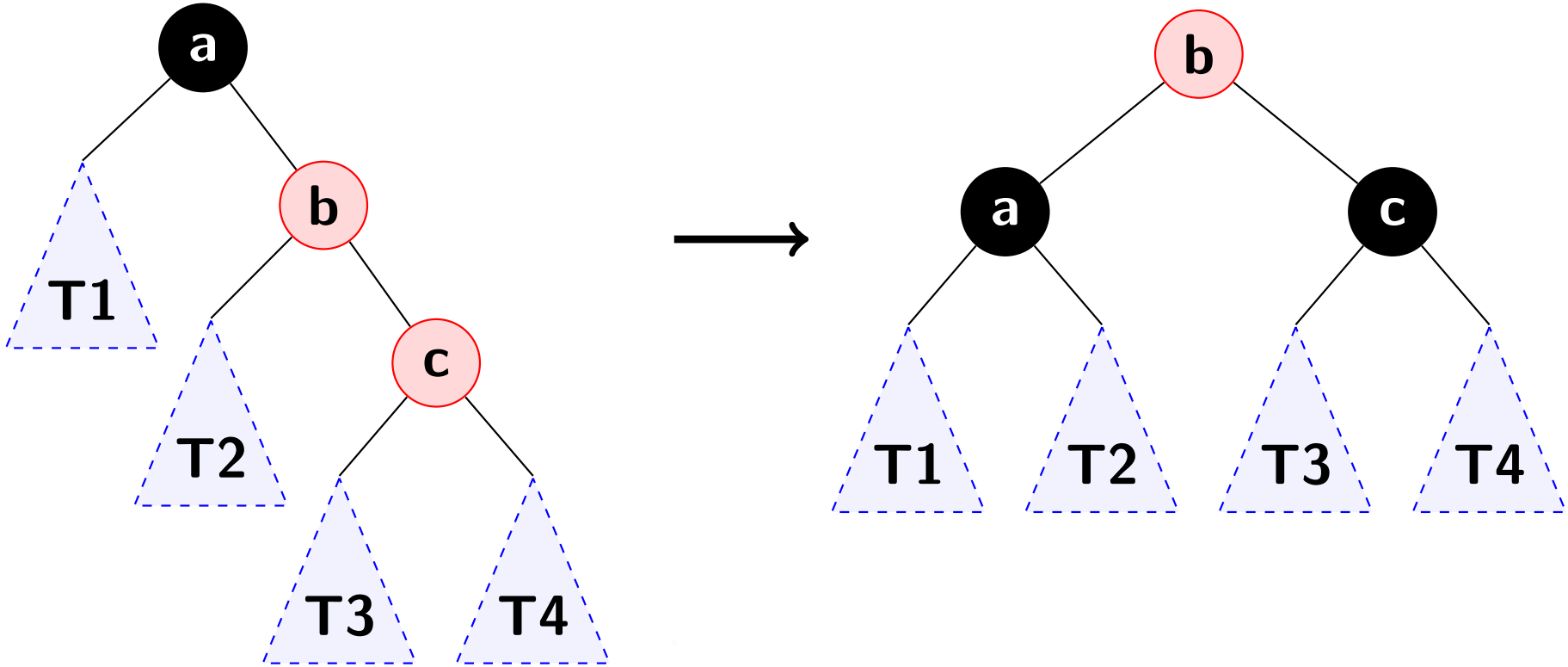
$$\text{cles(T1)} < \text{a} < \text{cles(T2)} < \text{b} < \text{cles(T3)} < \text{c} < \text{cles(T4)}$$

Insertion dans un arbre rouge-noir : cas 4



$$\mathbf{cles(T1) < a < cles(T2) < b < cles(T3) < c < cles(T4)}$$

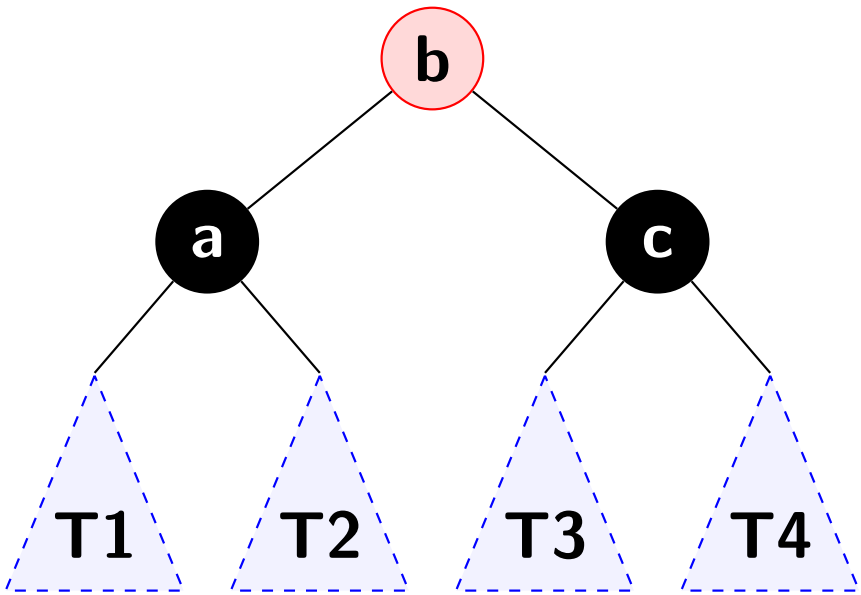
Insertion dans un arbre rouge-noir : cas 4



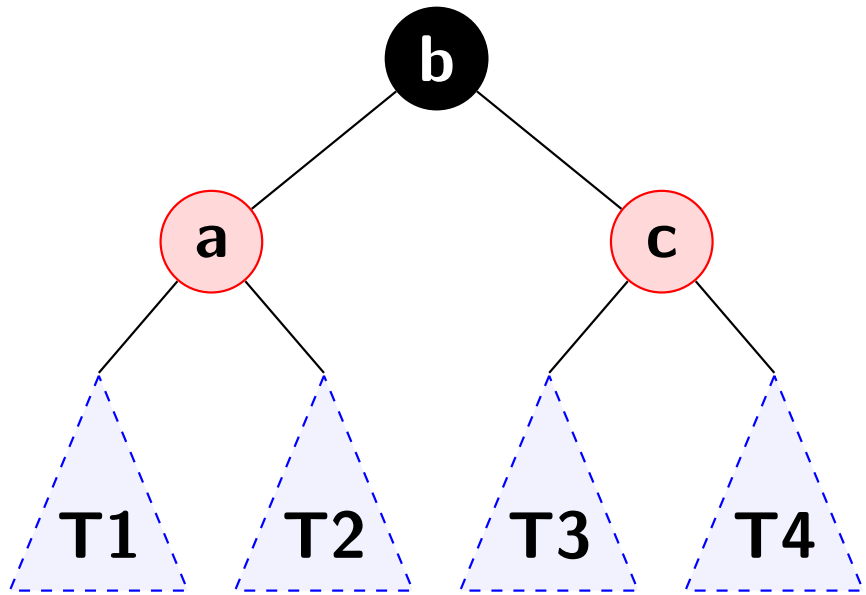
$$\mathbf{cles(T1) < a < cles(T2) < b < cles(T3) < c < cles(T4)}$$

Question...

Au lieu de



Peut-on choisir



? ? ? ?

Insertion et rééquilibrage

- La transformation crée un nœud rouge à la racine du sous-arbre sur lequel on travaille.
- $\Rightarrow$ Risque de créer à nouveau 2 nœuds rouges consécutifs.
- $\Rightarrow$ On doit ré-équilibrer récursivement.
- Si on arrive à la racine, on peut la colorier en noir.

  Augmente la hauteur noire de 1 sans changer le fait qu'elle est constante.

Suppression dans les arbres rouges-noirs

- Plus compliquée.
- On supprime d'abord comme dans un ABR normal.
- Problème quand on supprime un nœud noir : le nombre de nœuds noirs est maintenant inférieur.
- **Une solution** :
  - créer un nœud « double noir »,
  - le faire remonter, potentiellement jusqu'à la racine, par rotations.
  - le transformer en nœud noir s'il est à la racine.
  - la remontée du nœud double noir peut conduire à créer un nœud noir négatif.

Arbres rouges et noirs : complexité insertion, suppression

- La phase d'insertion sans ré-équilibrage prend un temps $O(\log(n))$
- Chaque ré-équilibrage local demande un temps $O(1)$
- Au pire $O(h) = O(\log(n))$ rééquilibrages chacuns coûtant $O(1)$ pour la suppression.
- **En tout** : $O(\log(n))$.

Plan

Les AVL

Les AVL sont une autre sorte d'arbres binaires de recherche équilibrés.

- **Équilibre** d'un nœud $=$

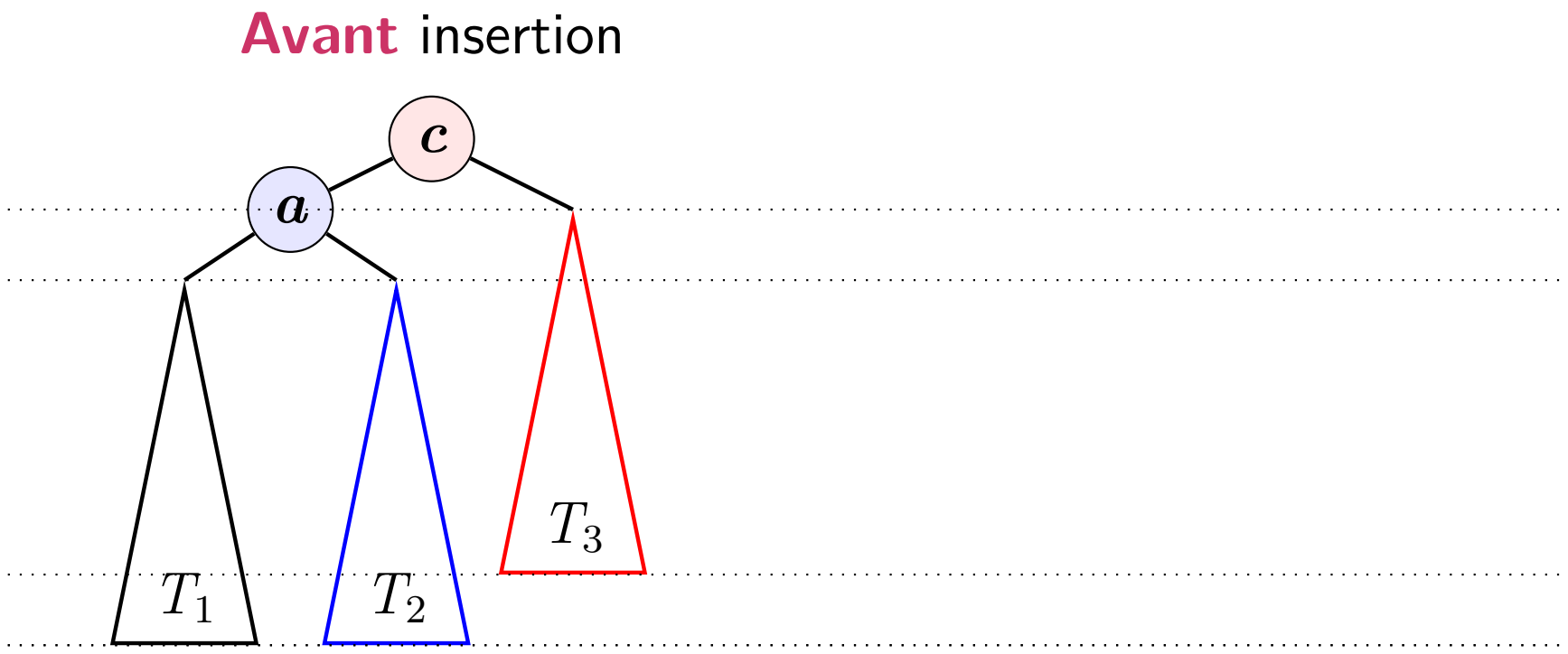
  hauteur(sous-arbre gauche) $-$ hauteur(sous-arbre droit).
- **AVL** $=$ ABR dont chaque nœud a un équilibre $-1\,, 0$ ou 1.

Insertion

- Insertion : on commence par insérer comme dans un **ABR**.
- Si création d'un nœud x, l'arbre n'est peut-être plus un **AVL**.
  Un des nœuds a pu passer d'équilibre ± 1 à ± 2.
- $c = 1^{\text{er}}$ nœud sur la branche de x à la racine d'équilibre ± 2.
- On doit **réparer** l'AVL pour rétablir les propriétés.

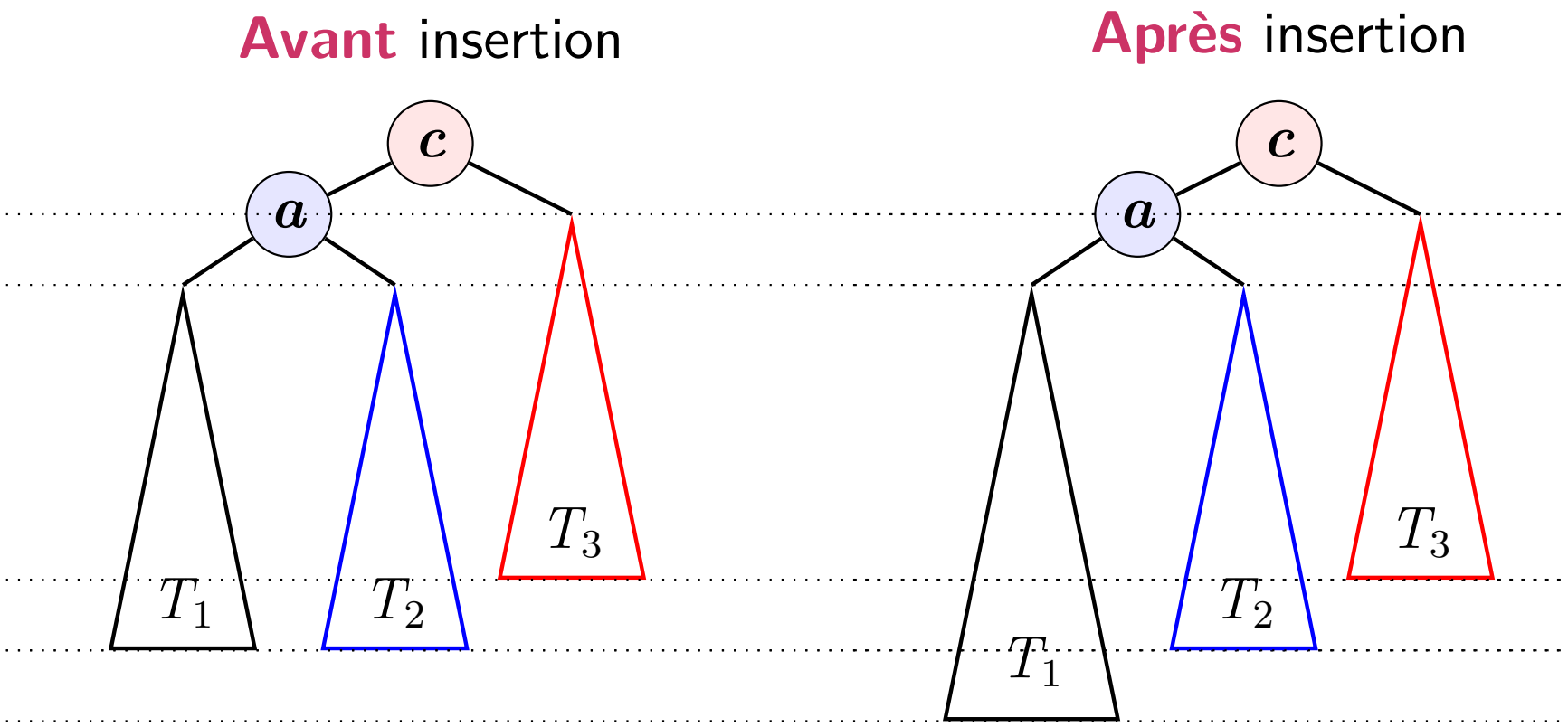
AVL : équilibrage, cas 1a

- Insertion dans le sous-arbre gauche du sous-arbre gauche (GG).



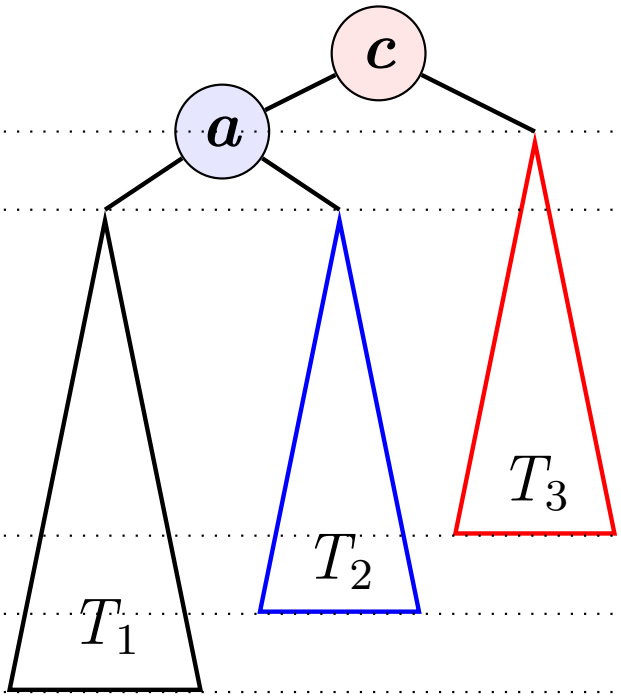
AVL : équilibrage, cas 1a

- Insertion dans le sous-arbre gauche du sous-arbre gauche (GG).



AVL : équilibrage, cas 1a

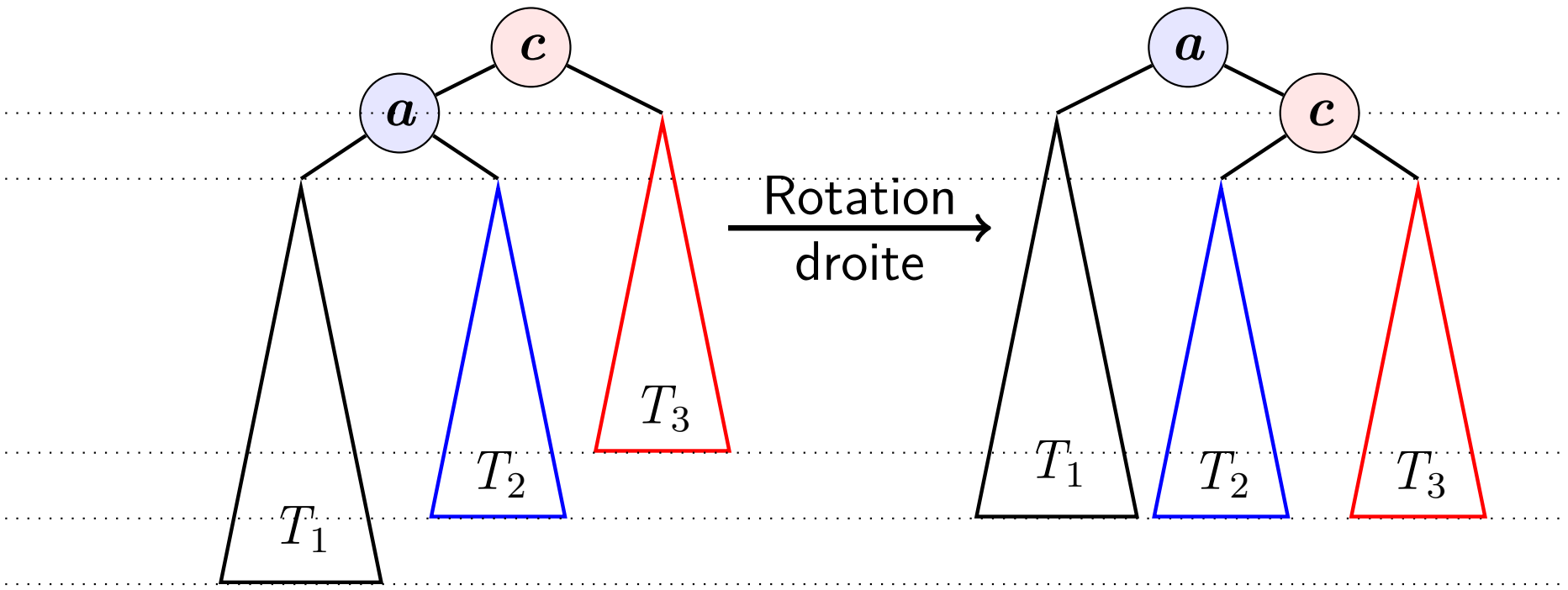
- hauteur(GG) = hauteur(D) + 1



$$\textit{clés}(T_1) < a < \textit{clés}(T_2) < c < \textit{clés}(T_3)$$

AVL : équilibrage, cas 1a

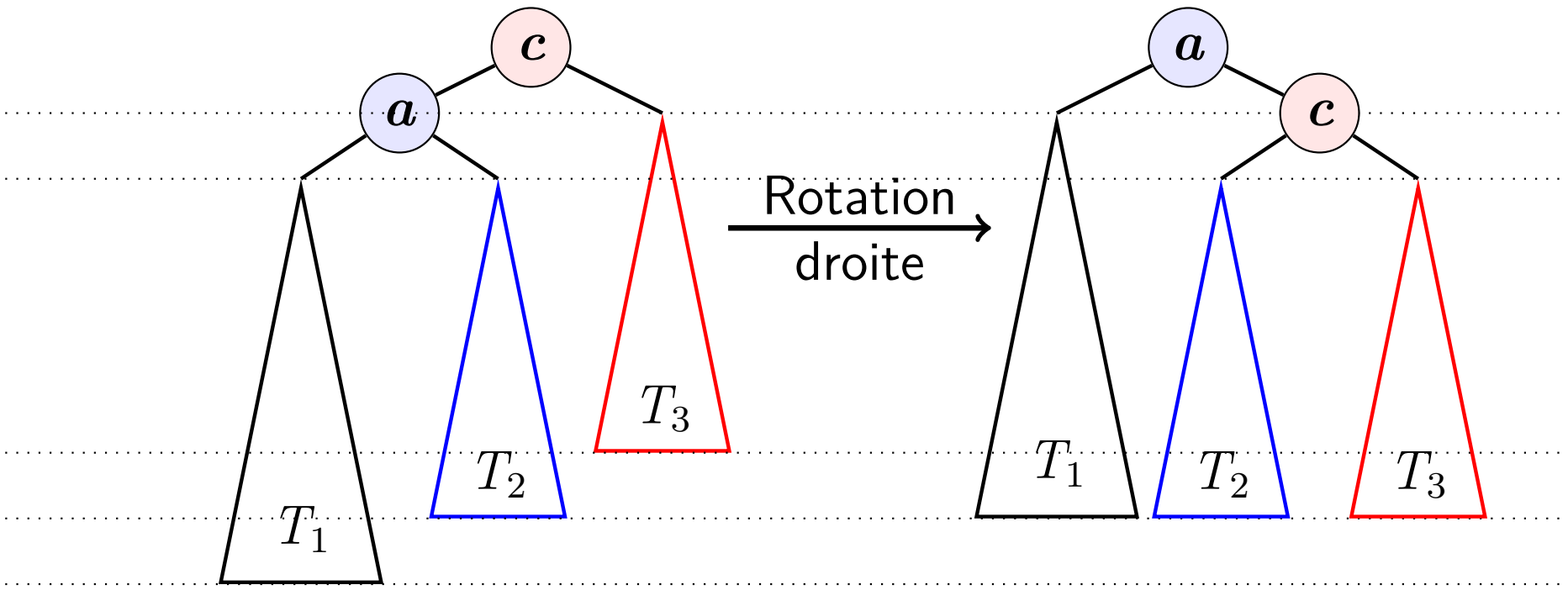
- hauteur(GG) = hauteur(D) + 1



$$clés(T_1) < a < clés(T_2) < c < clés(T_3)$$

AVL : équilibrage, cas 1a

- hauteur(GG) = hauteur(D) + 1

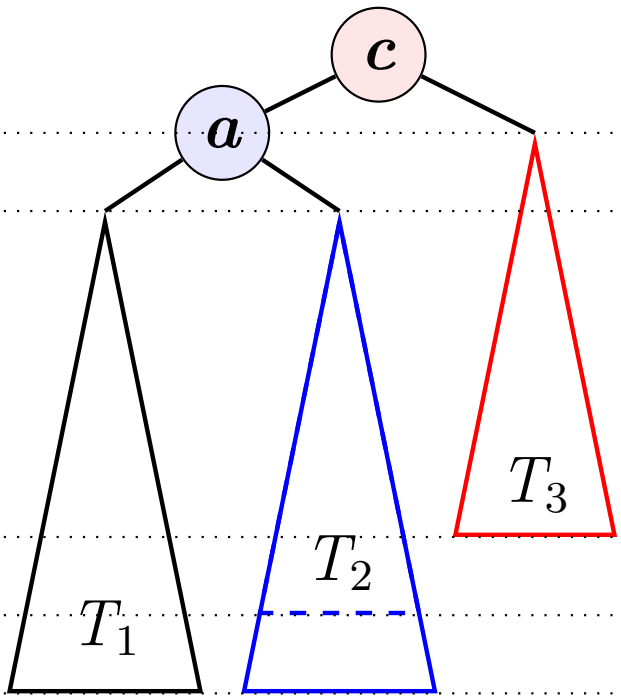


$$clés(T_1) < a < clés(T_2) < c < clés(T_3)$$

On retrouve la hauteur avant insertion $\rightsquigarrow$ terminé !

AVL : équilibrage, cas 1a

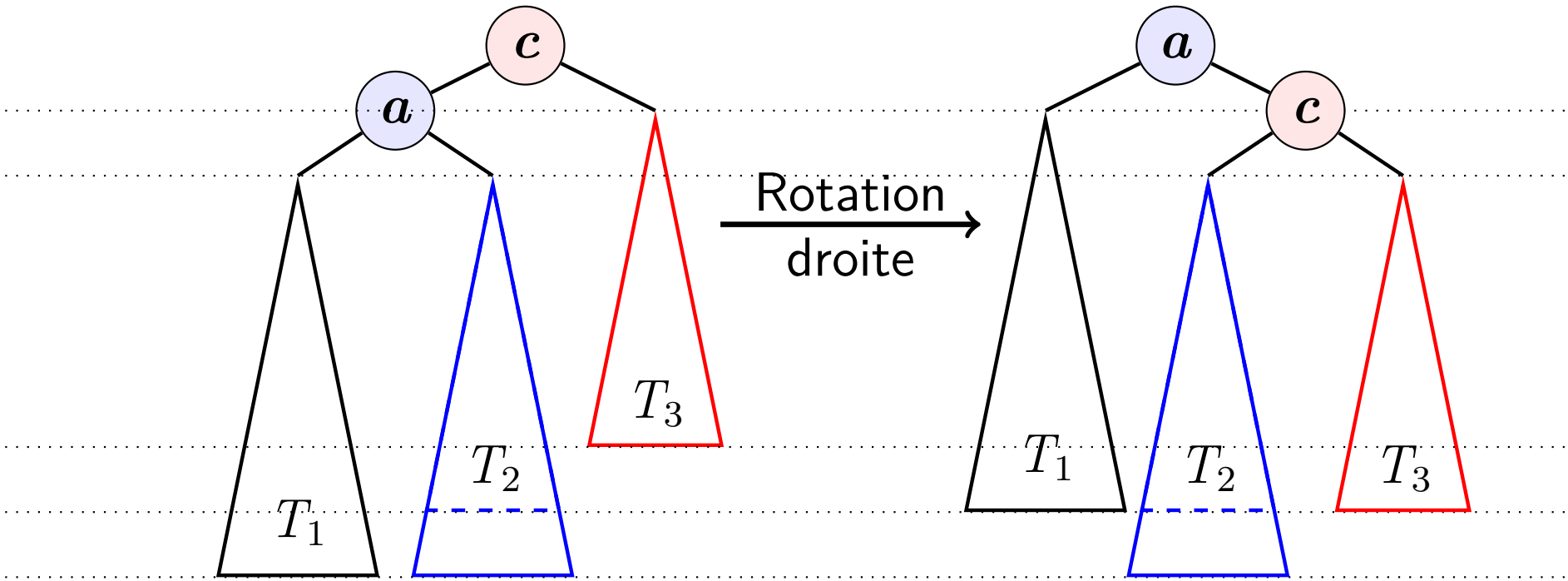
- **Remarque** Fonctionne même si T_2 est plus haut.
  Utile dans le cas de la suppression.



$$\textit{clés}(T_1) < a < \textit{clés}(T_2) < c < \textit{clés}(T_3)$$

AVL : équilibrage, cas 1a

- **Remarque** Fonctionne même si T_2 est plus haut.
  Utile dans le cas de la suppression.



$$\textbf{clés}(T_1) < a < \textbf{clés}(T_2) < c < \textbf{clés}(T_3)$$

AVL : équilibrage, cas 1b

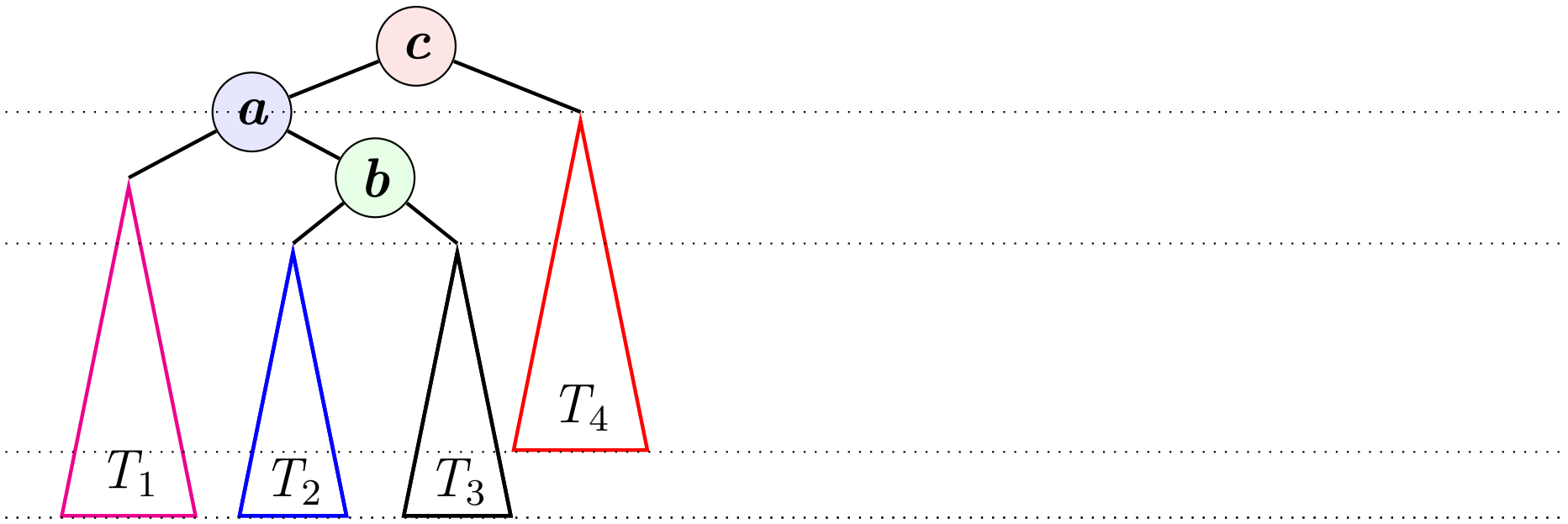
- $\text{hauteur}(\text{DD}) = \text{hauteur}(\text{G}) + 1$

Symétrique du cas 1a) : rotation gauche.

AVL : équilibrage, cas 2a

- Insertion dans le sous-arbre droit du sous-arbre gauche (GD). La hauteur de T_2 ou celle de T_3 a augmenté.

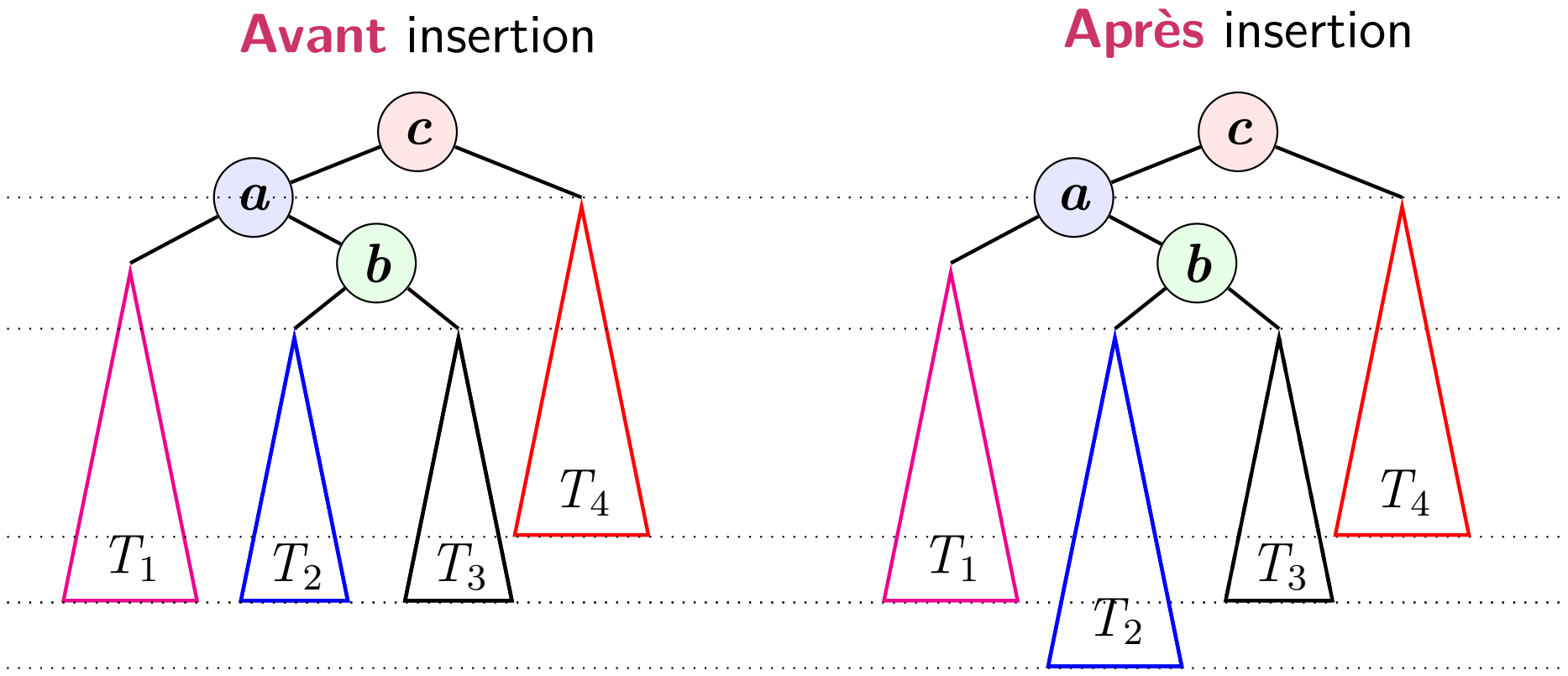
Avant insertion



$$\textit{clés}(T_1) < a < \textit{clés}(T_2) < b < \textit{clés}(T_3) < c < \textit{clés}(T_4)$$

AVL : équilibrage, cas 2a

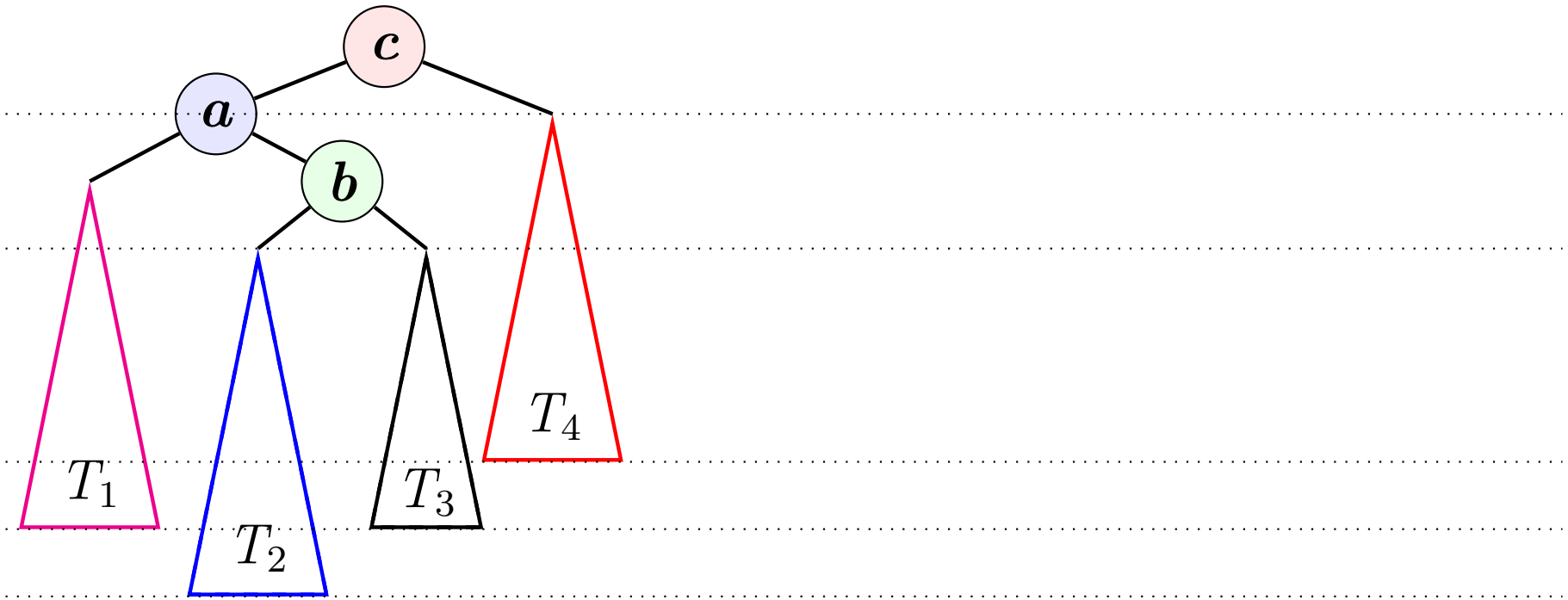
- Insertion dans le sous-arbre droit du sous-arbre gauche (GD). La hauteur de T_2 ou celle de T_3 a augmenté.



$$\textbf{clés}(T_1) < a < \textbf{clés}(T_2) < b < \textbf{clés}(T_3) < c < \textbf{clés}(T_4)$$

AVL : équilibrage, cas 2a

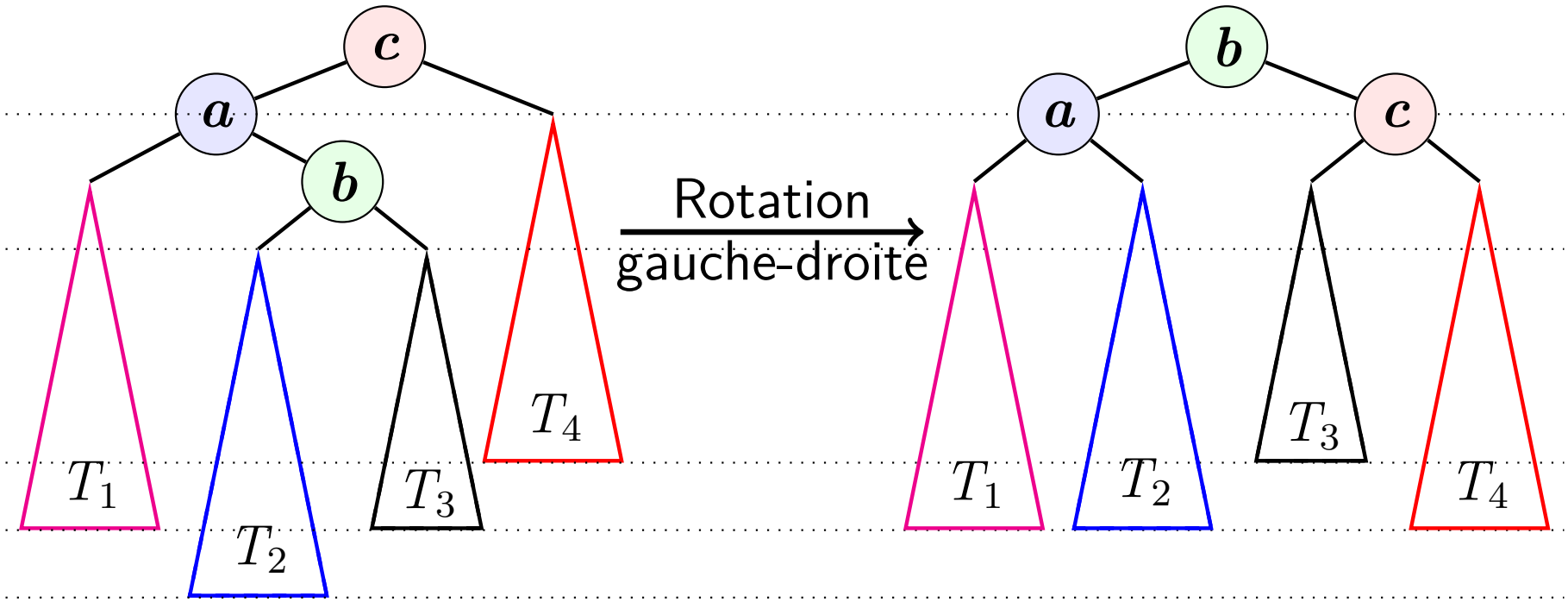
- hauteur(GD) = hauteur(D) + 1



$$\textbf{clés}(T_1) < a < \textbf{clés}(T_2) < b < \textbf{clés}(T_3) < c < \textbf{clés}(T_4)$$

AVL : équilibrage, cas 2a

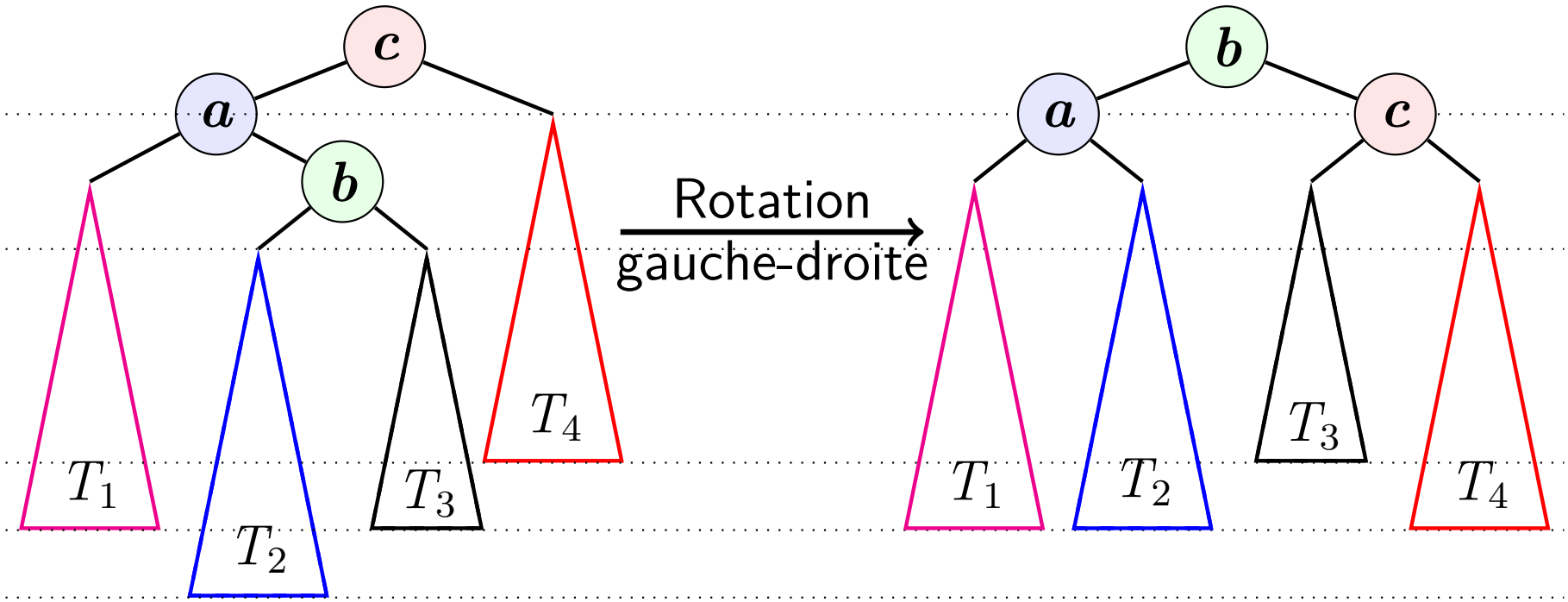
- hauteur(GD) = hauteur(D) + 1



$$clés(T_1) < a < clés(T_2) < b < clés(T_3) < c < clés(T_4)$$

AVL : équilibrage, cas 2a

- hauteur(GD) = hauteur(D) + 1



$$clés(T_1) < a < clés(T_2) < b < clés(T_3) < c < clés(T_4)$$

On retrouve la hauteur avant insertion $\rightsquigarrow$ terminé !

AVL : équilibrage, cas 2b

- $\text{hauteur}(\text{DG}) = \text{hauteur}(\text{D}) + 1$

Symétrique du cas 2a) : rotation droite-gauche.

Suppression dans les arbres AVL

- La suppression se gère de la même façon que l'insertion.
  - Suppression comme dans les arbres binaires de recherche.
  - Ré-équilibrage pour retrouver la propriété "AVL".
- Un ré-équilibrage peut déséquilibrer un nœud plus haut

$\rightsquigarrow$ peut nécessiter plusieurs équilibrages, au pire jusqu'à la racine.

Arbres AVL : complexité insertion, suppression

- La phase d'insertion prend un temps $O(\log(n))$
- Chaque ré-équilibrage local demande un temps $O(1)$
  - 1 rééquilibrage pour l'insertion.
  - Au pire $O(h) = O(\log(n))$ rééquilibrages chacuns coûtant $O(1)$ pour la suppression.

  En tout : $O(\log(n))$.

Récapitulatif pour les ABR équilibrés

- Les ABR équilibrés (rouges & noirs, AVL) permettent des opérations en $O(\log(n))$, où n est le nombre de nœuds.
- Permettent aussi, contrairement aux tables de hachage :
  - de trier les éléments en temps linéaire,
  - d'avoir accès en temps logarithmique au minimum, maximum.
  - d'avoir une complexité garantie.