# Algorithmique des structures de données arborescentes
# Feuille d'exercices 5 (semaines 6 à 8)

**Exercice 5.1**

1. Rappeler la hauteur *minimale* d'un arbre binaire à $n$ nœuds, et à quelle condition elle est atteinte.
2. Rappeler quelle est la hauteur *maximale* d'un arbre binaire à $n$ nœuds, et à quelle condition cette valeur est atteinte.

Pour un arbre binaire de recherche de hauteur $h$, on a vu que les opérations de

- recherche d'un élément,
- insertion d'un élément,
- suppression d'un élément

peuvent se réaliser en temps $O(h)$ dans le cas le pire. On considère deux familles d'arbres garantissant une hauteur au maximum $\alpha \log(n)$ pour des arbres à $n$ nœuds (où la constante $\alpha > 0$ dépend de la famille d'arbres). Intuitivement, un arbre d'une famille d'arbres équilibrés a beaucoup de nœuds, un nombre exponentiel par rapport à sa hauteur.

La 1re famille est celle des arbres rouges et noirs. La 2e est celle des arbres AVL. L'objectif de cette feuille d'exercices est de comprendre comment insérer une nouvelle valeur dans de tels arbres (la suppression fera l'objet d'autres exercices).

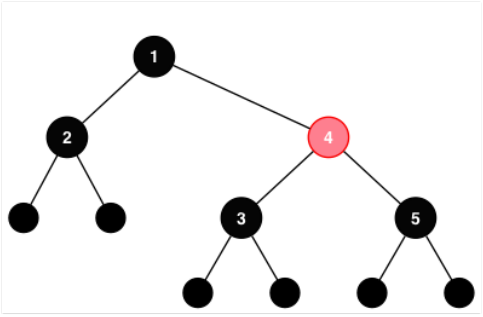
## 1 Arbres rouges et noirs

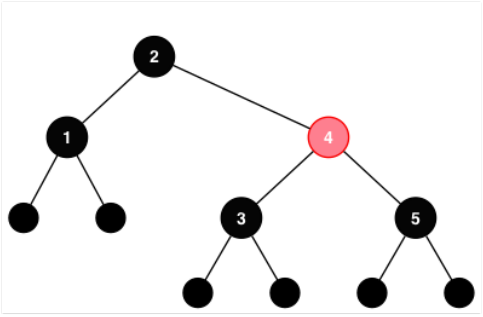
Un **arbre rouge et noir** est un *arbre binaire* ***de recherche*** tel que :

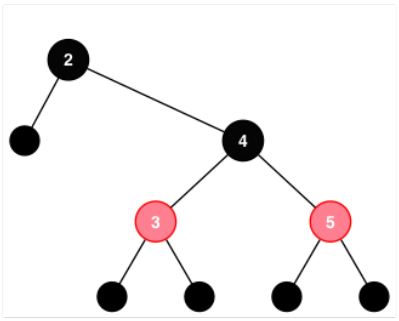
1. il est complet (pas de nœud d'arité 1),
2. seuls les nœuds internes portent des valeurs,
3. chaque nœud est soit rouge, soit noir,
4. la racine est noire,
5. les feuilles sont noires,
6. le père d'un nœud rouge est noir,
7. le nombre de nœuds noirs sur chaque branche est constant.

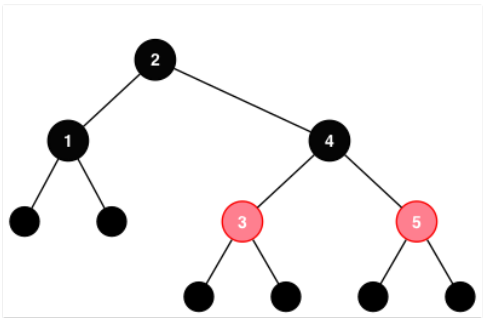
Si $t$ est un arbre rouge et noir, le nombre de nœuds noirs sur chaque branche est appelé la *hauteur noire de* $t$ et noté $hn(t)$.

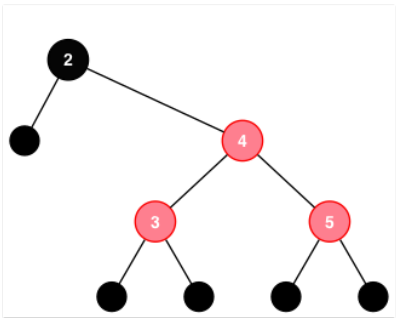
**Exercice 5.2** Parmi les arbres suivants, lesquels sont des arbres rouges et noirs ? Pour les autres, indiquer quelles propriétés sont manquantes (les multiplicités sont omises sur les figures).










On utilise le type OCaml suivant pour représenter les arbres rouges et noirs :

```
type 'a rb = Leaf | Red of 'a * 'a rb * 'a rb | Black of 'a * 'a rb * 'a rb
```

Les feuilles sont notées simplement `Leaf` puisqu'elles ne portent pas de valeur (et leur couleur est implicitement noire). Un nœud rouge est construit grâce au constructeur `Red` et un nœud noir grâce au constructeur `Black`. Attention, certaines valeurs de type `'a rb` ne représentent pas un arbre rouge et noir.

**Exercice 5.3**

1. Écrire une fonction `min_max_black_height` qui, étant donné un arbre `t` de type `'a rb`, renvoie un couple `(m,m')`, où
   - `m` est le nombre *minimal* de nœuds noirs trouvés sur les branches de `t`,
   - `m'` est le nombre *maximal* de nœuds noirs trouvés sur les branches de `t`.
2. On suppose déjà écrite une fonction `is_bst` qui teste si un arbre de type `(int * 'a) tree` est un arbre binaire de recherche (voir 2e exercice de la feuille 4), où le type `'a tree` est défini par :

   ```
   type 'a tree = Empty | Bin of 'a * 'a tree * 'a tree
   ```

   Écrire une fonction `is_rb : (int * 'a) rb -> bool` testant si un arbre de type `(int * 'a) rb` est un arbre rouge et noir. On demande que la complexité de cette fonction soit $O(n)$, où $n$ est le nombre de nœuds de l'arbre (en supposant que la fonction `is_bst` a elle aussi une complexité $O(n)$).
3. Justifier la complexité $O(n)$ en évaluant le nombre d'appels récursifs et le temps consommé lors de chaque appel.

**Exercice 5.4 — Les arbres rouges et noirs sont équilibrés.** Soit $t$ un arbre rouge et noir de hauteur $h(t)$, de hauteur noire $hn(t)$, ayant $n(t)$ nœuds et $i(t)$ nœuds internes.

1. En utilisant le fait qu'un arbre rouge et noir est complet, rappeler la relation entre $n(t)$ et $i(t)$.
2. Montrer que $hn(t) - 1 \leqslant h(t) \leqslant 2(hn(t) - 1)$.
3. En utilisant le fait que l'arbre est complet, montrer que $n(t) \geqslant 2^{hn(t)} - 1$.
4. Montrer que $h(t) \leqslant 2\log_2(i(t) + 1)$.

## 1.1 Insertion dans un arbre rouge et noir

Un algorithme d'insertion d'une nouvelle clé dans les arbres rouges et noirs est le suivant :

- On insère la nouvelle clé en remplaçant une feuille noire par un nœud rouge portant cette valeur (ce nœud a comme fils deux feuilles noires). Pour cette étape, on utilise l'algorithme vu pour les arbres binaires de recherche.
- Cette étape préserve les propriétés des arbres binaires de recherche, sauf peut-être :
  - si on a inséré dans un arbre ne portant aucune valeur, la racine est rouge au lieu d'être noire,
  - après insertion, il peut y avoir deux nœuds rouges consécutifs (un nœud et son père).

La première propriété est facile à corriger. Pour la seconde, on peut se trouver dans une des 4 configurations suivantes :

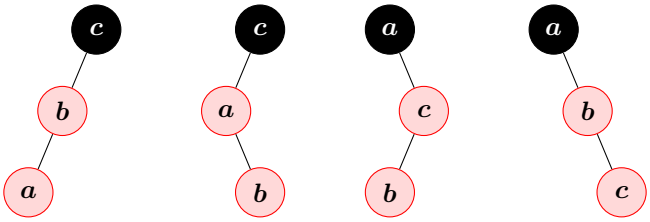
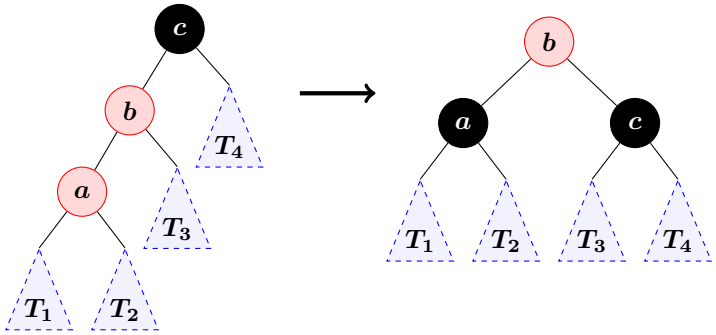



FIG. 5.1 : Quatre configurations avec 2 nœuds rouges consécutifs ($a < b < c$)

Chacune de ces 4 configurations correspond à un sous-arbre interdit, car il ne satisfait pas la condition 6 de la définition des arbres rouges et noirs. Dans chaque cas, on applique une transformation locale de ce sous-arbre qui se réalise en temps $O(1)$. Par exemple, dans la première configuration, on réorganise le sous-arbre de la façon suivante :

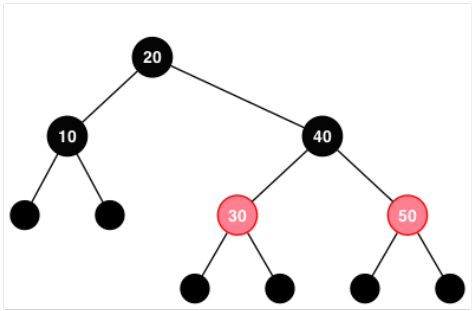



Intuitivement, cette réorganisation fait remonter le problème d'avoir 2 nœuds rouges consécutifs vers la racine (il faudrait donc la répéter « vers le haut » dans un algorithme itératif).

**Exercice 5.5**

1. Vérifier que la hauteur noire *de l'arbre de départ* n'est pas modifiée par la réorganisation ci-dessus.
2. Vérifier que si *l'arbre de départ* est un ABR, il en est de même de l'arbre produit.
3. Pour chacune des trois autres configurations de la FIG. 5.1, dessiner des sous-arbres $T_1, T_2, T_3$ et $T_4$ aux emplacements manquants, avec $clés(T_1) < a < clés(T_2) < b < clés(T_3) < c < clés(T_4)$ et proposer une réorganisation de l'arbre supprimant les nœuds rouges consécutifs du sous-arbre traité, tout en conservant les faits que :
   - toutes les branches ont le même nombre de nœuds noirs,
   - l'arbre produit est un ABR.

**Exercice 5.6** On considère l'arbre rouge et noir suivant :




Quels arbres rouges et noirs obtient-on après avoir inséré :

1. la clé 55, puis la clé 45 ?
2. la clé 45, puis la clé 55 ?
3. la clé 45, puis la clé 35 ?

**Exercice 5.7** Écrire une fonction `rb_balance` qui prend en paramètre un arbre de type `'a rb` et qui renvoie l'arbre donné en exercice 5, question 3 si son argument correspond à l'un des 4 motifs de la Fig. 5.1, et renvoie son argument inchangé sinon.

**Exercice 5.8** Écrire une fonction d'insertion `rb_insert` de type `('a * int) rb -> 'a -> ('a * int) rb`. La seconde composante des nœuds est la multiplicité.

## 2 Arbres AVL

Les arbres AVL imposent une autre contrainte sur les arbres binaires de recherche, et cette contrainte a aussi pour conséquence qu'un AVL a une hauteur logarithmique par rapport à son nombre de nœuds. Le terme AVL provient des initiales des chercheurs ayant introduit ces arbres, Georgii Adelson-Velsky et Evguenii Landis.

On appelle **équilibre** d'un nœud la différence entre la hauteur de son sous-arbre gauche et de son sous-arbre droit. Un arbre binaire de recherche est appelé un *AVL* si l'équilibre de ***tout*** nœud de l'arbre est soit $-1$, soit 0, soit 1. D'après l'exercice 1, feuille 3, il existe $\beta > 0$ telle que tout AVL à $n$ nœuds a une hauteur au maximum $\beta \cdot \log_2(n)$ : les arbres AVL forment une famille d'arbres équilibrés.

L'algorithme usuel d'insertion dans un AVL se réalise en 2 phases :

- on insère d'abord la valeur comme dans un ABR classique, en remplaçant un sous-arbre vide par une nouvelle feuille,
- si la propriété d'être un AVL a été détruite par l'insertion, on rééquilibre l'arbre en le réorganisant.

Pour cela, on considère le premier nœud sur la branche menant de la feuille insérée à la racine dont l'équilibre est différent de $-1$, 0 ou 1. On l'appelle « nœud témoin » et on note $c$ son étiquette dans les figures suivantes.

**Exercice 5.9** Quelles sont les valeurs possibles de l'équilibre du nœud témoin ?

Pour rééquilibrer l'arbre, on peut supposer (par symétrie), que l'équilibre du nœud témoin est 2.

**Exercice 5.10** Vérifier qu'après insertion, si l'arbre n'est plus un AVL et que l'équilibre du nœud témoin est 2, on est dans l'un des 3 cas suivants : la configuration de gauche de la Fig. 5.2, celle de gauche de la Fig. 5.3, et celle de la configuration de gauche de la Fig. 5.3 en inversant les arbres $T_2$ et $T_3$.

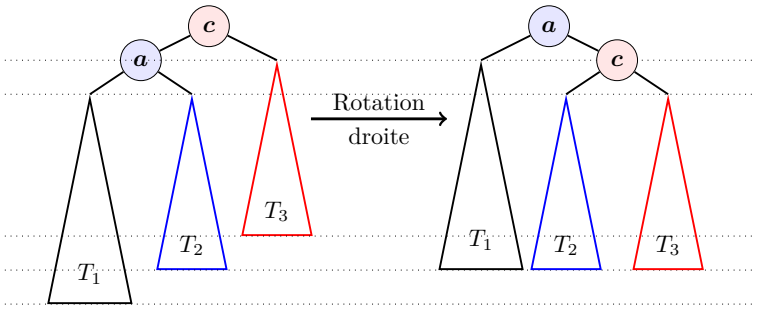



Fig. 5.2 : Insertion dans un AVL : cas 1

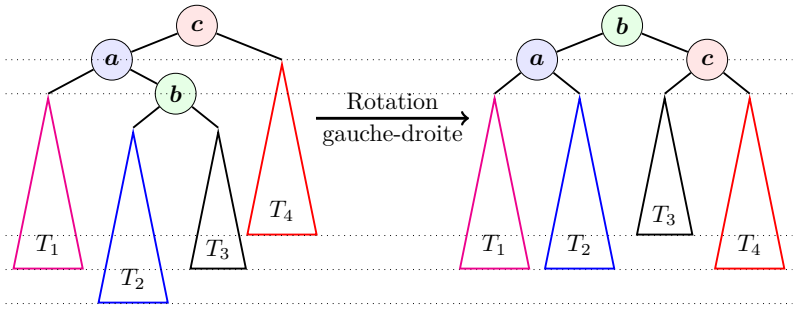



Fig. 5.3 : Insertion dans un AVL : cas 2

**Exercice 5.11**

1. En appliquant les transformations proposées ci-dessus et leurs symétriques, quels AVL obtient-on en insérant successivement à partir de l'arbre vide les valeurs 1, 2, 3, 4, 5 ? Les valeurs 1, 5, 2, 4, 3 ?
2. Vérifier qu'après une insertion dans un AVL, *une seule* réorganisation suffit pour retrouver la propriété d'être un AVL (contrairement aux arbres rouges et noirs, il n'est pas besoin de recommencer la réorganisation à une profondeur inférieure, c'est-à-dire plus près de la racine).

Pour écrire de façon efficace les algorithmes sur les arbres AVL, on mémorise dans chaque nœud la hauteur du sous-arbre en ce nœud. Ainsi, un arbre AVL sera représenté par une valeur du type `(('a * int) * int) tree` : la valeur de type `'a` est la clé du nœud, couplée à sa multiplicité. La seconde valeur entière est la hauteur de l'arbre. Par exemple, `Bin(((10,1),0), Empty, Empty)` est un AVL de hauteur 0 à une clé (10, de multiplicité 1).

**Exercice 5.12**

1. Écrire une fonction `compute_heights : ('a * int) tree -> (('a * int) * int) tree` qui renvoie l'arbre dans lequel cette composante de hauteur a été calculée et ajoutée.
2. Écrire une fonction `is_avl : ('a * int) tree -> bool` testant en temps $O(n)$ si un arbre binaire est un arbre AVL, où $n$ est le nombre de nœuds de l'arbre à tester (à nouveau, on suppose déjà écrite la fonction `is_bst`).

**Exercice 5.13**

1. Écrire une fonction `avl_fix` qui réalise le rééquilibrage d'un AVL selon les schémas ci-dessus.
2. Écrire une fonction `avl_insert` telle que `avl_insert t x` renvoie l'AVL obtenu depuis `t` en y insérant la clé `x`.