11 Arbres

Avec les tris, la notion d'arbre apparait.

11.1 Contexte

Les arbres[42] sont des listes à plusieurs éléments ; les arbres sont composés de nœuds et d'arcs ; les arcs relient les nœuds ; parmis les noeuds, on distingue la racine, les nœuds intermédiaires et les feuilles ; le premier nœud est appelé racine ; les nœuds reliés par un arc à la racine sont les nœuds-fils de la racine ; les nœuds ne possédant pas de fils sont des feuilles ; les nœuds qui ne sont pas des feuilles et qui ne sont pas la racine sont des nœuds intermédiaires ; les feuilles sont des nœuds terminaux.

Les arbres binaires sont des arbres où chaque nœud peut avoir au maximum deux nœuds-fils ; afin d'obtenir une représentation sans ambiguïté d'un arbre, on considérera le fils unique d'un nœud comme le sous-arbre gauche de ce nœud.

On appelle également facteur de branchement le nombre de nœuds-fils pour chaque nœud dans l'arbre ; dans un arbre binaire, le facteur de branchement est de 2.

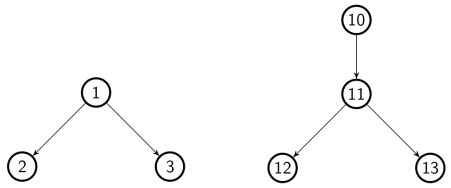

Fig. 2 – Arbres à 3 et 4 nœuds.

La figure 2 présente deux exemples arbres ; à gauche, on a un arbre à 3 nœuds ; 1 est la racine ; 2 et 3 sont des feuilles ; à droite, on a un arbre à 4 nœuds ; 10 est la racine ; 11 est un nœud intermédiaire ; 12 et 13 sont des feuilles.

La profondeur d'un nœud est la distance à la racine ; la racine est à profondeur zéro ; les nœuds-fils de la racine sont à profondeur un ; la hauteur d'un arbre est la profondeur maximale de ses nœuds.

Un arbre est complet si toutes les feuilles sont à la même profondeur ; dans ce cas, pour un arbre de hauteur h :
- le nombre de feuilles est 2^h
- le nombre de nœuds est $2^{h+1} - 1$

42. Un arbre est un graphe orienté sans cycle ; les nœuds d'un arbre peuvent posséder des nœuds-fils pour lesquels ce nœud sera leur nœud parent ; l'orientation du graphe est une conséquence de l'existance du nœud racine qui implique un sens d'orientation ; les arcs permettent *a priori* d'aller vers les nœuds-fils ; stocker la liaison vers le nœud parent n'est pas obligatoire ; l'absence de cycle implique qu'il n'existe qu'un seul chemin pour relier deux nœuds.

En utilisant les listes, six solutions de réprésentation interne sont possibles :

① Considérer les nœuds non-terminaux comme des listes de taille variable et les feuilles comme des éléments.

② Considérer les nœuds non-terminaux comme des listes de taille variable et les feuilles comme des listes à un élément.

③ Considérer les nœuds non-terminaux comme des listes de taille variable et les feuilles comme des listes de taille 3, avec un élément et deux listes vides.

④ Considérer les nœuds non-terminaux comme des listes de taille 3 (en les complétant si nécessaire avec des listes vides) et les feuilles comme des éléments.

⑤ Considérer les nœuds non-terminaux comme des listes de taille 3 (en les complétant si nécessaire avec des listes vides) et les feuilles comme des listes de taille 1.

⑥ Considérer les nœuds non-terminaux comme des listes de taille 3 (en les complétant si nécessaire avec des listes vides) et les feuilles comme des listes de taille 3.

Avec ①, les arbres présentés en figure 2 correspondent aux définitions suivantes :

```
1 '(1 2 3)
2 '(10 (11 12 13))
```

Avec ②, les arbres présentés en figure 2 correspondent aux définitions suivantes :

```
1 '(1 (2) (3))
2 '(10 (11 (12) (13)))
```

Avec ③, les arbres présentés en figure 2 correspondent aux définitions suivantes :

```
1 '(1 (2 () ()) (3 () ()))
2 '(10 (11 (12 () ()) (13 () ())))
```

Avec ④, les arbres présentés en figure 2 correspondent aux définitions suivantes :

```
1 '(1 2 3)
2 '(10 (11 12 13) ())
```

Avec ⑤, les arbres présentés en figure 2 correspondent aux définitions suivantes :

```
1 '(1 (2) (3))
2 '(10 (11 (12) (13)) ())
```

Avec ⑥, les arbres présentés en figure 2 correspondent aux définitions suivantes :

```
1 '(1 (2 () ()) (3 () ()))
2 '(10 (11 (12 () ()) (13 () ())) ())
```

Avec ⑥, on a des définitions structurellement plus lourdes et globalement plus homogènes ; tous les nœuds sont représentés par des listes de taille 3 ; on peut déduire du contenu des listes la nature des nœuds (*i.e.* si deuxième et troisième valeurs sont des listes vides, alors le nœud considéré est une feuille).

11.2 Arbres binaires

Afin de simplifier/clarifier l'utilisation des arbres, il est possible de définir des fonctions de manipulation des arbres dans une interface (nommée `bt1`) :

```
(define tree list)
(define (leaf e) e)
(define (leaf? e) (not(list? e)))
(define (not-leaf? e) (list? e))
(define (root T) (first T))
(define (L-subtree T) (first(rest T)))
(define (R-subtree T) (first(rest(rest T))))
(define (has-L-subtree? T) (>(length T) 1))
(define (has-R-subtree? T) (>(length T) 2))
```

Ici, comme présenté dans ⓵, les nœuds intermédiaires sont des listes de taille variable et les feuilles sont des éléments.

Notons que :

- La définition du prédicat `not-leaf?` évite `(not(leaf? e))` équivalent à `(not(not(list? e)))` qui produirait un test avec une double négation inutile.
- Les fonctions de test `has-L-subtree?` et `has-R-subtree?` utilisent la taille de la liste et présupposent donc que le sous-arbre gauche existe avant le sous-arbre droit.

Ainsi les arbres présentés en figure 2 correspondent maintenant à :

```
(tree 1 (leaf 2) (leaf 3))
(tree 10 (tree 11 (leaf 12) (leaf 13)))
```

```
'(1 2 3)
'(10 (11 12 13))
```

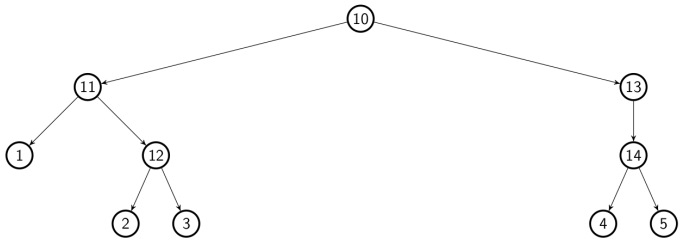

Fig. 3 – Arbres binaire avec 10 nœuds.

La figure 3 présente un arbre à 10 nœuds, dont les définitions suivantes sont équivalentes :

```
(tree 10 (tree 11 (leaf 1) (tree 12 (leaf 2) (leaf 3)))
         (tree 13 (tree 14 (leaf 4) (leaf 5))))
(tree 10 (tree 11 1 (tree 12 2 3)) (tree 13 (tree 14 4 5)))
'(10 (11 1 (12 2 3)) (13 (14 4 5)))
```

11.3 Parcours en profondeur

Le parcours en profondeur consiste à descendre le plus profond possible avant d'explorer la branche suivante ; dans les arbres étiquetés sur tous les nœuds, on différencie les parcours préfixe, infixe et postfixe :

- Dans le cas préfixe, il s'agit d'évaluer le nœud courant, puis le sous-arbre gauche, puis le sous-arbre droit ; un nœud est donc évalué avant ses sous-arbres gauche et droit.
- Dans le cas infixe, il s'agit d'évaluer le sous-arbre gauche, puis le nœud courant, puis le sous-arbre droit ; un nœud est donc évalué quand son sous-arbre droit est complètement évalué et est évalué ensuite son sous-arbre droit.
- Dans le cas postfixe, il s'agit d'évaluer le sous-arbre gauche, puis le sous-arbre droit, puis le nœud courant ; un nœud est donc évalué quand ses deux sous-arbres sont complètement évalués.

Parcours préfixe en profondeur avec l'interface `bt1`

La fonction `dfs-list` réalise la construction d'une liste des nœuds de l'arbre en suivant un parcours préfixe en profondeur et en utilisant les définitions de l'interface `bt1` :

```
1 (define (dfs-list T)
2   (if (leaf? T) (list T)
3     (if (has-R-subtree? T)
4       (append (list (root T)) (dfs-list (L-subtree T))
5                               (dfs-list (R-subtree T)))
6       (append (list (root T)) (dfs-list (L-subtree T)))
7     )))
```

Appliquée à l'arbre de la figure 3, la fonction `dfs-list` retourne la liste des nœuds correspondant au parcours préfixe.

```
1 (dfs-list '(10 (11 1 (12 2 3)) (13 (14 4 5))))
```

```
'(10 11 1 12 2 3 13 14 4 5)
```

Parcours infixe en profondeur avec l'interface `bt1`

La fonction `dfs-list2` réalise la construction d'une liste des nœuds de l'arbre en suivant un parcours infixe en profondeur et en utilisant les définitions de l'interface `bt1` :

```
(define (dfs-list2 T)
  (if (leaf? T) (list T)
    (if (has-R-subtree? T)
      (append (dfs-list2 (L-subtree T)) (list (root T))
              (dfs-list2 (R-subtree T)))
      (append (dfs-list2 (L-subtree T)) (list (root T)))
    )))
```

Appliquée à l'arbre de la figure 3, la fonction `dfs-list2` retourne la liste des nœuds correspondant au parcours infixe.

```
(dfs-list2 '(10 (11 1 (12 2 3)) (13 (14 4 5))))
```

```
'(1 11 2 12 3 10 4 14 5 13)
```

Parcours postfixe en profondeur avec l'interface `bt1`

La fonction `dfs-list3` réalise la construction d'une liste des nœuds de l'arbre en suivant un parcours postfixe en profondeur et en utilisant les définitions de l'interface `bt1` :

```
(define (dfs-list3 T)
  (if (leaf? T) (list T)
    (if (has-R-subtree? T)
      (append (dfs-list3 (L-subtree T)) (dfs-list3 (R-subtree T))
              (list (root T)))
      (append (dfs-list3 (L-subtree T)) (list (root T)))
    )))
```

Appliquée à l'arbre de la figure 3, la fonction `dfs-list3` retourne la liste des nœuds correspondant au parcours postfixe.

```
(dfs-list3 '(10 (11 1 (12 2 3)) (13 (14 4 5))))
```

```
'(1 2 3 12 11 4 5 14 13 10)
```

11.4 Parcours en largeur

Partant de la notion de niveau dans un arbre, qui caractérise l'ensemble des nœuds à une profondeur fixée, le parcours en largeur consiste à descendre niveau par niveau et d'énumérer les nœuds de chaque niveau ; un niveau inférieur (*i.e.* à profondeur `+1`) est exploré après évaluation complète du niveau courant.

On peut réaliser un parcours en profondeur en utilisant deux listes :
- Une première liste des nœuds courants nommée $\mathcal{R}$
- Une seconde liste des fils des nœuds courants nommée $\mathcal{L}$
- Initialement $\mathcal{R} \leftarrow \{\texttt{root}(T)\}$ et $\mathcal{L} \leftarrow \{\emptyset\}$
- A chaque itération, $\mathcal{L}$ reçoit les fils de $\mathcal{R}$
- Entre chaque itération, $\mathcal{R} \leftarrow \mathcal{L}$
- On arrête quand $\mathcal{L} = \{\emptyset\}$

Appliqué à la figure 3, on a :
- A la 1ère itération : $\mathcal{R} = \{10\}$ et $\mathcal{L} = \{11, 13\}$
- A la 2ème itération : $\mathcal{R} = \{11, 13\}$ et $\mathcal{L} = \{1, 12, 14\}$
- A la 3ème itération : $\mathcal{R} = \{1, 12, 14\}$ et $\mathcal{L} = \{2, 3, 4, 5\}$
- A la 4ème itération : $\mathcal{R} = \{2, 3, 4, 5\}$ et $\mathcal{L} = \{\emptyset\}$

En utilisant `bt1`, on définit une fonction `root-nodes` qui pour une liste d'arbres `Lin` retourne la liste des nœuds root de `Lin` et une fonction `subtrees` qui pour une liste d'arbres `Lin` retourne la liste des sous-arbres de `Lin`.

```
(define (root-nodes Lin)
  (if (empty? Lin) empty
    (let ([i (first Lin)])
    (if (leaf? i)
      (cons i (root-nodes (rest Lin)))
      (cons (root i) (root-nodes (rest Lin)))
    ))))
(define (subtrees Lin)
  (if (empty? Lin) empty
     (let ([i (first Lin)])
     (if (leaf? i)
        (subtrees (rest Lin))
        (if (has-R-subtree? i)
           (cons (L-subtree i) (cons (R-subtree i)
              (subtrees (rest Lin))))
           (cons (L-subtree i) (subtrees (rest Lin)))
        )))
  ))
```

Ce qui permet d'obtenir :

```
1 (root-nodes '((10 (11 1 (12 2 3)) (13 (14 4 5)))))
2 (root-nodes '((11 1 (12 2 3)) (13 (14 4 5))))
```

```
'(10)
'(11 13)
```

La première liste (ligne 1) contient l'arbre de la figure 3 dont la racine est 10. La deuxième liste (ligne 2) contient les deux sous-arbres de 10, dont les racines sont 11 et 13.

On obtient également :

```
1 (subtrees '((10 (11 1 (12 2 3)) (13 (14 4 5)))))
2 (subtrees '((11 1 (12 2 3)) (13 (14 4 5))))
```

```
'((11 1 (12 2 3)) (13 (14 4 5)))
'(1 (12 2 3) (14 4 5))
```

La première liste (ligne 1) contient l'arbre de la figure 3 dont les sous-arbres sont :

- `(11 1 (12 2 3))`
- `(13 (14 4 5))`

La deuxième liste (ligne 2) contient les deux sous-arbres de 10, dont les sous-arbres sont :

- `1`
- `(12 2 3)`
- `(14 4 5)`

Avec les fonctions `root-nodes` et `subtrees`, on obtient la fonction `bfs` pour un parcours en largeur :

```
1 (define (bfs Ln)
2   (define (f Ln R)
3     (if (empty? Ln)  R
4        (f (subtrees Ln) (append R (root-nodes Ln)))
5     ))
6   (f Ln empty))
7 (bfs '((10 (11 1 (12 2 3)) (13 (14 4 5)))))
```

```
'(10 11 13 1 12 14 2 3 4 5)
```

La fonction `bfs` utilise une fonction auxiliaire `f` pour initialiser $\mathcal{R}$ à la liste vide ; appliquée à l'arbre de la figure 3, `bfs` retourne une énumération des nœuds à profondeur croissante.

11.5 Arbres de recherche

Dans les arbres binaires de recherche, les nœuds sont classés selon la valeur des étiquettes ; pour un nœud `n` de valeur `v`, les nœuds de valeur inférieure à `v` seront classés dans les sous-arbres gauches de `n` et les nœuds de valeur supérieure à `v` seront classés dans les sous-arbres droits de `n` ; la position des nœuds dans l'arbre dépend donc de l'ordre d'ajout dans l'arbre ; la figure 4 représente un arbre dont les éléments sont ajoutés dans l'ordre défini par le liste L1 égale à `'(6 3 7 8 5 2)` ; la figure 5 représente un arbre dont les éléments sont ajoutés dans l'ordre défini par le liste L2 égale à `'(7 6 8 3 2 5)`.

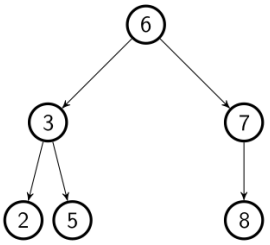

Fig. 4 – `arbre de L1`.

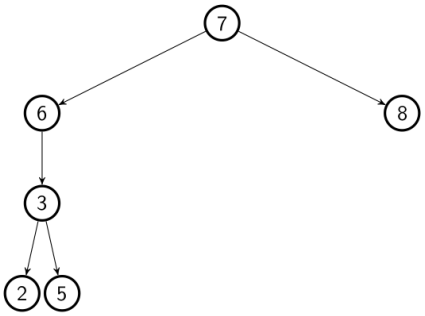

Fig. 5 – `arbre de L2`.

En utilisant la représentation ⓰, il est possible de définir des fonctions de manipulation des arbres binaires de recherche dans une interface (nommée `bt6`) :

```
(define (tree val L R) (list val L R))
(define (leaf val) (list val empty empty))
(define (root T) (first T))
(define (L-subtree T) (first(rest T)))
(define (R-subtree T) (first(rest(rest T))))
(define (has-L-subtree? T) (not (empty? (L-subtree T))))
(define (has-R-subtree? T) (not (empty? (R-subtree T))))
(define (add T val)
  (if (empty? T) (leaf val)
    (let ([r (root T)])
      (if (< val r)
        (if (has-L-subtree? T)
          (tree r (add (L-subtree T) val) (R-subtree T))
          (tree r (leaf val) (R-subtree T)))
        (if (has-R-subtree? T)
          (tree r (L-subtree T) (add (R-subtree T) val))
          (tree r (L-subtree T) (leaf val)))
      ))))
(define (addl T L)
  (if (empty? L) T
    (addl (add T (first L)) (rest L))
  ))
```

L'interface `bt6` représente les arbres avec des listes de taille 3 ; les feuilles sont également des listes de taille 3, avec une liste vide pour sous-arbre gauche et une liste vide pour sous-arbre droit ; la fonction `leaf` crée un arbre avec une racine et des sous-arbres vides ; la fonction `add` permet d'ajouter des éléments dans un arbre et la fonction `addl` permet d'ajouter une liste de valeurs.

On peut retrouver les arbres des figures 4 et 5 en utilisant `add` ou `addl` :

```
(add(add(add(add(add(add empty 6) 3) 7) 8) 5) 2)
(addl empty '(6 3 7 8 5 2))
(add(add(add(add(add(add empty 7) 6) 8) 3) 2) 5)
(addl empty '(7 6 8 3 2 5))
```

```
'(6 (3 (2 () ()) (5 () ())) (7 () (8 () ())))
'(6 (3 (2 () ()) (5 () ())) (7 () (8 () ())))
'(7 (6 (3 (2 () ()) (5 () ())) ()) (8 () ()))
'(7 (6 (3 (2 () ()) (5 () ())) ()) (8 () ()))
```

11.6 Arbres équilibrés

Un arbre est équilibré quand ses sous-arbres ont des petites différences de hauteurs ; cette différence de hauteurs (sous-arbre gauche moins sous-arbre droit) est appelée facteur d'équilibrage ; l'utilisation des arbres équilibrés permet de garantir des opérations de recherche d'un élément en temps optimal [43] ; les opérations d'ajout et de suppression d'un élément sont cependant plus coûteuse en temps car elles peuvent impliquer une transformation de l'arbre (appelée rééquilibrage) ; dans un arbre AVL [44], le facteur d'équilibrage est inférieur strict à 2 en valeur absolue [45] ; si un nœud possède un facteur d'équilibrage supérieur à 2 en valeur absolue, alors un rééquilibrage est nécessaire ; l'insertion d'un nouvel élément suit donc les étapes suivantes :

- Insertion du nouvel élément dans l'arbre.
- Mise à jour des facteurs d'équilibrage des nœuds parents de l'élément ajouté.
- Si le facteur d'équilibrage d'un nœud est au-delà des valeurs admises, appliquer un rééquilibrage [46].

Les opérations de rééquilibrage possibles sont la rotation droite, la rotation gauche, la rotation gauche-droite et la rotation droite-gauche.

43. Dans un arbre équilibré à n nœuds, la recherche d'un élément est de $O(log\ n)$.

44. Les arbres de recherche automatiquement équilibrés ont été presentés par Adelson-Velskii et Landis en 1962.

45. Pour une hauteur plus grande dans le sous-arbre gauche, un nœud possède une différence de hauteur positive ; pour une hauteur plus grande dans le sous-arbre droit, un nœud possède une différence de hauteur négative ; pour un nœud n, pour h la fonction de hauteur d'un arbre, avec $G(n)$ le sous-arbre gauche du nœud n et $D(n)$ le sous-arbre droit du nœud n, un arbre AVL vérifie $|h(G(n)) - h(D(n))| < 2$.

46. Lors d'un ajout dans un arbre équilibré, les facteurs d'équilibrage des nœuds parent du nœud ajouté sont à mettre à jour ; regarder les facteurs d'équilibrage de ses nœuds est donc suffisant pour détecter un déséquilibre.

Si un nœud possède un facteur d'équilibrage de 2 et son sous-arbre gauche un facteur d'équilibrage de 1, alors une rotation droite est nécessaire ; pour r la racine avant rotation, r' la racine après rotation, $g(n)$ le fils gauche du nœud n, $d(n)$ le fils droit du nœud n, une rotation droite correspond aux étapes suivantes :

— r perd son fils gauche $g(r)$.
— $g(r)$ perd son fils droit $d(g(r))$.
— $g(r)$ devient la nouvelle racine r'.
— r devient $d(r')$ le fils droit de r'.
— $d(g(r))$ devient $g(d(r'))$ le fils gauche du fils droit de r'.

Le calcul d'un facteur d'équilibrage est défini comme suit :

— Si c'est une feuille, il vaut `*`,
— Si c'est un nœud avec un sous-arbre gauche feuille et sans sous-arbre droit, il vaut `1`,
— Sinon il vaut facteur de sous-arbre gauche moins facteur de sous-arbre droit.

La figure 6 présente un exemple de rotation droite à la racine ; pour chaque nœud, on note à sa droite, la hauteur de l'arbre correspondant, suivie du facteur d'équilibrage ; hauteur et facteur d'équilibrage sont séparés par un slash (noté `/`) ; les feuilles possèdent des hauteurs nulles et pas de facteur d'équilibrage (notés `0/*`) ; avant rotation, le nœud de valeur 4 a une hauteur de 2 et un facteur d'équilibrage de 1, et la racine est le nœud de valeur 10, qui a une hauteur de 3 et un facteur d'équilibrage de 2, qui implique un rééquilibrage ; les nœuds de valeur 10 et 4 sont déclencheurs de ce rééquilibrage ; après rééquilibrage, leur valeur d'équilibrage est nulle dans ce cas ; selon les cas, les valeurs d'équilibrage après rééquilibrage sont de -1, 0 ou 1.

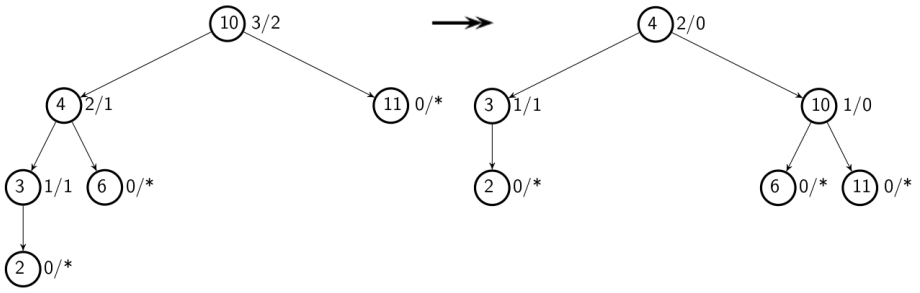

Fig. 6 – Rotation droite d'un arbre.

La figure 7 présente un autre cas possible d'arbre nécessitant un rééquilibrage par rotation droite ; la figure 8 présente un arbre avec des valeurs négatives de facteur d'équilibrage.

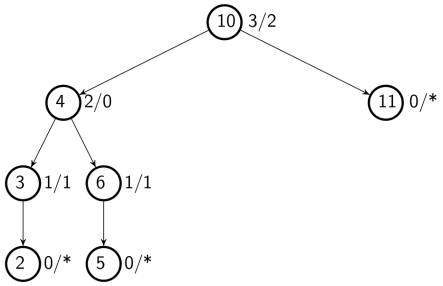

Fig. 7 – Exemple d'arbre nécessitant un rééquilibrage par rotation droite.

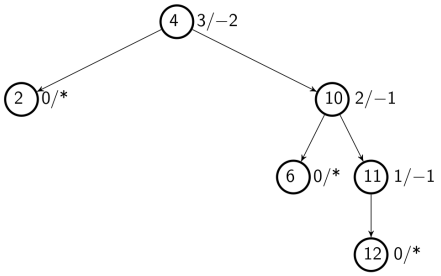

Fig. 8 – Exemple d'arbre avec des facteurs d'équilibrage négatifs.

Par symétrie, si un nœud possède un facteur d'équilibrage de -2 et son sous-arbre droit un facteur d'équilibrage de 1, alors une rotation gauche est nécessaire ; dans ce cas :

— r perd son fils droit $d(r)$.
— $d(r)$ perd son fils gauche $g(d(r))$.
— $d(r)$ devient la nouvelle racine r'.
— r devient $g(r')$ le fils gauche de r'.
— $g(d(r))$ devient $d(g(r'))$ le fils droit du fils gauche de r'.

La figure 9 présente un exemple de rotation gauche à la racine ; avant rotation, le nœud de valeur 11 a une hauteur de 2 et un facteur d'équilibrage de -1, et la racine est le nœud de valeur 4, qui a une hauteur de 3 et un facteur d'équilibrage de -2, qui implique un rééquilibrage ; les nœuds de valeur 11 et 4 sont déclencheurs de ce rééquilibrage ; après rééquilibrage, leur valeur d'équilibrage est nulle.

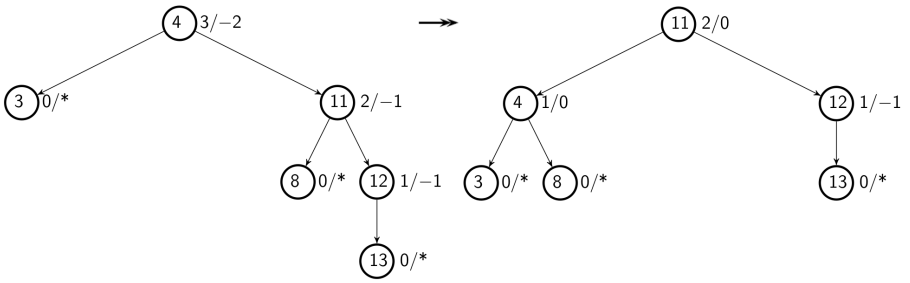

Fig. 9 – Rotation gauche d'un arbre.

Selon les situations, le rééquilibrage nécessite une opération (comme vu précédemment en figure 6) ou deux opérations ; la figure 10 présente un exemple de rotation gauche droite (*i.e.* gauche sur le fils gauche puis droite sur la racine) ; les figures 11, 12, 14 et 13 résument respectivement l'ensemble des situations de rééquilibrage par rotation droite, rotation gauche, rotation gauche droite et rotation droite gauche ; les sous-arbres sont représentés par des triangles.

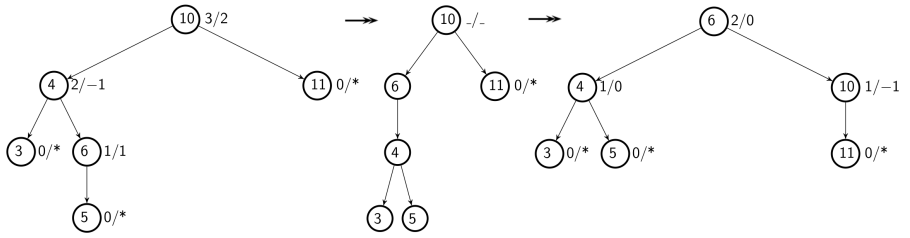

Fig. 10 – Rotation gauche droite d'un arbre.

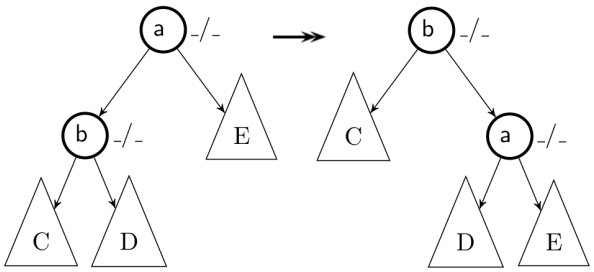

Fig. 11 – Rotation droite d'un arbre ; les différentes valeurs de facteur d'équilibrage des nœuds (a, b) avant rotation sont $(2, 1)$ et $(2, 0)$; par rotation on a : $(2, 1) \to (0, 0)$ et $(2, 0) \to (1, -1)$.

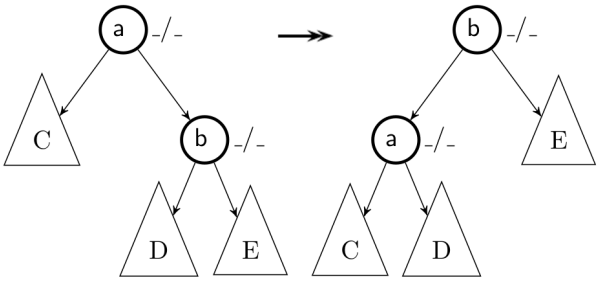

Fig. 12 – Rotation gauche d'un arbre ; les différentes valeurs de facteur d'équilibrage des nœuds (a, b) avant rotation sont $(-2, -1)$ et $(-2, 0)$; par rotation on a : $(-2, -1) \to (0, 0)$ et $(-2, 0) \to (-1, 1)$.

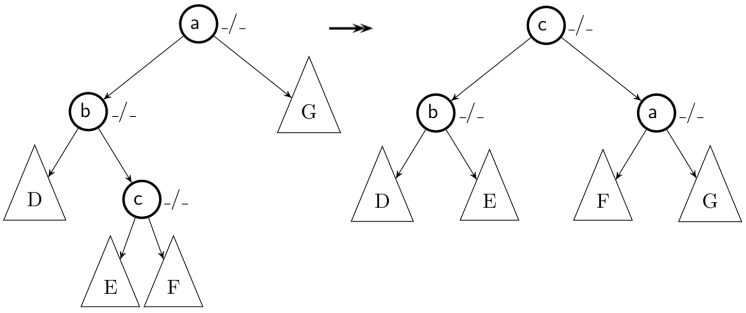

Fig. 13 – Rotation gauche droite d'un arbre ; les différentes valeurs de facteur d'équilibrage des nœuds (a, b, c) avant rotation sont $(2, -1, -1)$, $(2, -1, 0)$ et $(2, -1, 1)$; par rotation on a : $(2, -1, -1) \to (1, 0, 0)$, $(2, -1, 0) \to (0, 0, 0)$ et $(2, -1, 1) \to (0, -1, 0)$.

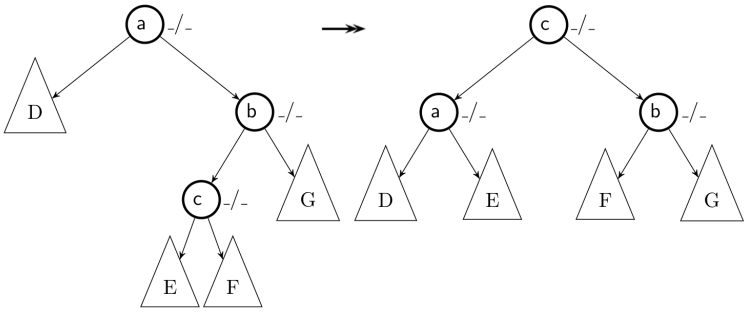

Fig. 14 – Rotation droite gauche d'un arbre ; les différentes valeurs de facteur d'équilibrage des nœuds (a, b, c) avant rotation sont $(-2, 1, -1)$, $(-2, 1, 0)$ et $(-2, 1, 1)$; par rotation on a : $(-2, 1, -1) \to (1, 0, 0)$, $(-2, 1, 0) \to (0, 0, 0)$ et $(-2, 1, 1) \to (0, -1, 0)$.

Il s'agit donc d'observer le facteur d'équilibrage à chaque ajout d'une nouvelle feuille, pouvant déséquilibrer l'arbre et impliquer une opération de rotation.

L'interface `avltree`, définie ci-dessous, représente les arbres AVL avec des listes de taille 5 ; un nœud possède une valeur, une profondeur, un facteur d'équilibrage, un sous-arbre droit et sous-arbre gauche ; la fonction `(avltree val L R)` construit l'arbre avec les valeurs de profondeur et de facteur d'équilibrage mais sans faire l'opération de rééquilibrage.

```
(define (leaf e) (list e 0 0 empty empty))
(define (avltree val L R)
 (if (and (empty? L) (empty? R)) (leaf val)
   (let ([dl (depth L)]
         [dr (depth R)])
     (list val (+ 1 (max dl dr)) (- dl dr) L R)
   )))
(define (equi T) (first (rest (rest T))))
(define (avltree5 val d e L R) (list val d e L R))
(define (root T) (first T))
(define (L-subtree T) (first(rest(rest(rest T)))))
(define (R-subtree T) (first(rest(rest(rest(rest T))))))
(define (has-L-subtree? T) (not(empty? (L-subtree T))))
(define (has-R-subtree? T) (not(empty? (R-subtree T))))
(define (leaf? T) (and (empty? (L-subtree T)) (empty? (R-subtree T))))
(define (depth T)
  (if (empty? T) -1
    (if (leaf? T) 0
      (first(rest T)))))
(define (add T val)
  (if (empty? T) (leaf val)
    (let ([r (root T)])
      (if (< val r)
        (if (has-L-subtree? T)
          (avltree r (add (L-subtree T) val) (R-subtree T))
          (avltree r (leaf val) (R-subtree T)))
        (if (has-R-subtree? T)
          (avltree r (L-subtree T) (add (R-subtree T) val))
          (avltree r (L-subtree T) (leaf val)))
      ))))
(define (addl T L)
  (if (empty? L) T
    (addl (add T (first L)) (rest L))
  ))
```

En utilisant la fonction `addl` on peut définir l'arbre de la figure 6.

```
(addl empty '(10 4 3 6 2 11))
```

```
'(10 3 2 (4 2 1 (3 1 1 (2 0 0 () ()) ())
                (6 0 0 () ()))
         (11 0 0 () ()))
```

Et enfin, on définit la rotation droite, conformément à la figure 11 pour les facteurs d'équilibrage des nœuds (a, b) variant par rotation selon $(2, 1) \rightarrow (0, 0)$, comme suit :

```
(define (rotate-R T)
  (let* ([a (root T)]
         [b (root (L-subtree T))]
         [C (L-subtree (L-subtree T))]
         [D (R-subtree (L-subtree T))]
         [E (R-subtree T)]
         [aDE (avltree a D E)])
    (avltree5 b
              (+ 1 (max (depth C) (depth aDE)))
              0
              C
              aDE
    )))
```

Ce qui permet d'obtenir le résultat suivant :

```
(rotate-R (addl empty '(10 4 3 6 2 11)))
```

```
'(4 2 0 (3 1 1 (2 0 0 () ()) ())
        (10 1 0 (6 0 0 () ()) (11 0 0 () ())) )
```

Pour finaliser l'interface `avltree`, le lecteur pourra :

- Définir les différentes fonctions de rotation conformément aux figures 11, 12, 13 et 14.
- Définir une fonction `reequi` qui teste les différents cas [47] de déséquilibre et appelle la fonction de rééquilibrage associée.
- Ajouter un appel à `reequi` pour l'ajout de chaque nœud.

47. Dans le cas de l'ajout, le test des valeurs d'équilibrage des nœuds parent du nœud ajouté est suffisant pour la détection du déséquilibre ; dans le cas de la suppression, le test des nœuds parent du nœud supprimé est suffisant pour la détection du déséquilibre.