

# Le langage Racket

---

- Un langage de programmation fonctionnelle
- 10/11/2021

# Plan du cours

- 1. Arbres équilibrés
- 2. Exercices

# 1. Arbres équilibrés

Dans un arbre AVL, si un nœud possède un facteur d'équilibrage supérieur à 2 en valeur absolue, un rééquilibrage est nécessaire

L'insertion d'un nouvel élément suit 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<sup>46</sup>.

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

# 1. Arbres équilibrés

Dans un arbre AVL, si un nœud possède un facteur d'équilibrage supérieur à 2 en valeur absolue, un rééquilibrage est nécessaire

L'insertion d'un nouvel élément suit 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<sup>46</sup>.

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

# 1. Arbres équilibrés

Comment rééquilibrer un arbre?

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  $r'$ .

# 1. Arbres équilibrés

Comment rééquilibrer un arbre?

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.

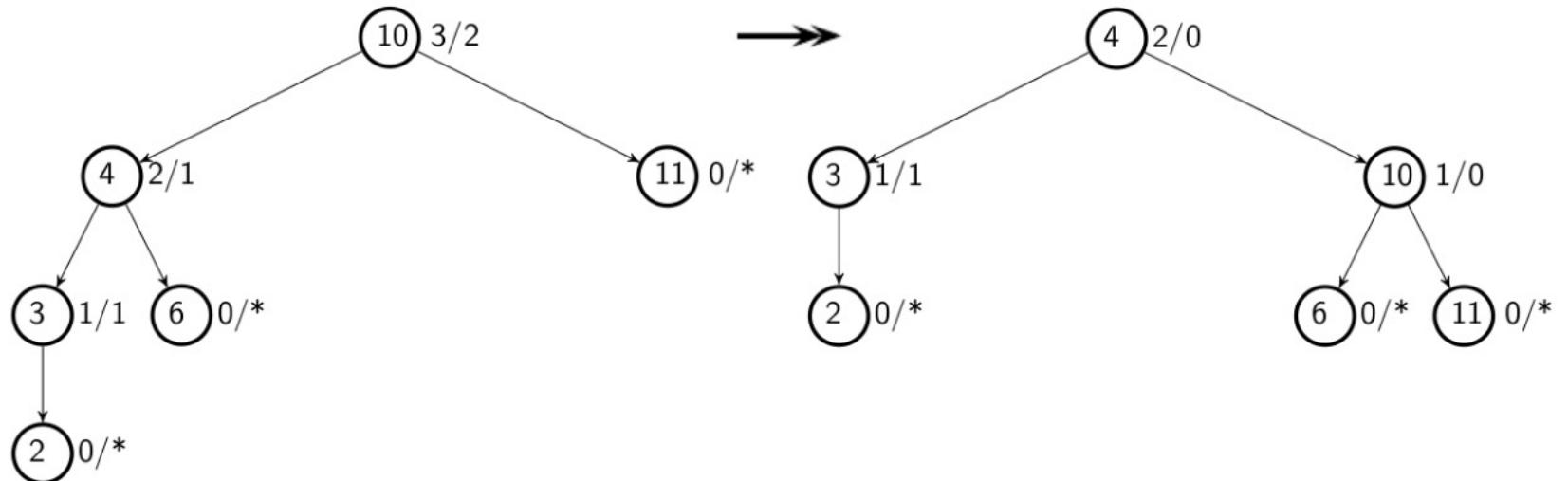


Fig. 6 – Rotation droite d'un arbre.

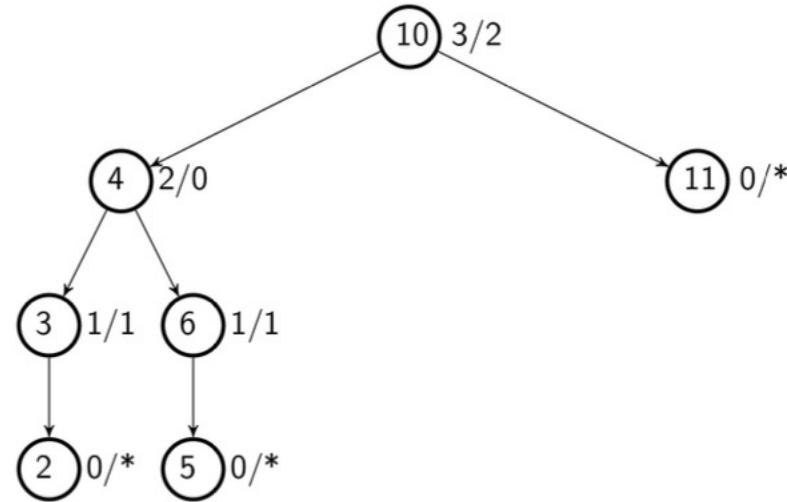


Fig. 7 – Exemple d’arbre nécessitant un rééquilibrage par rotation droite.

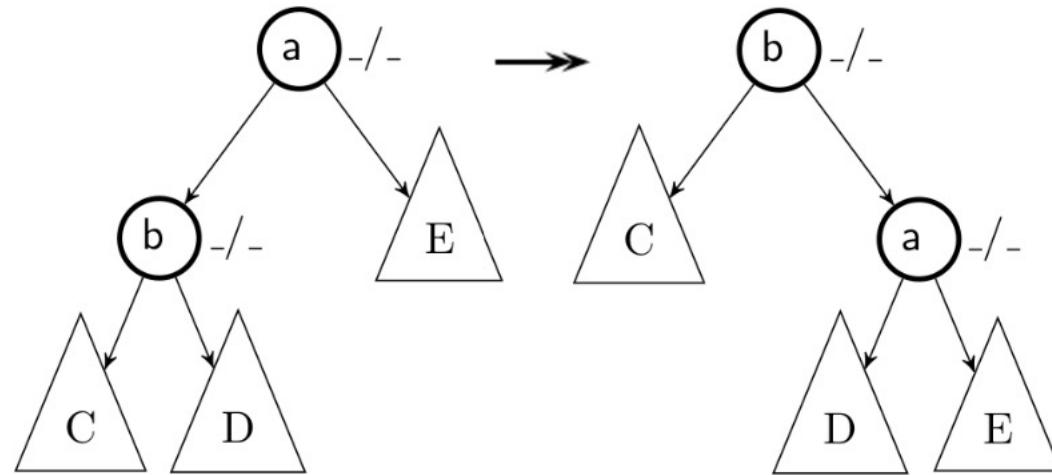


Fig. 11 – Rotation droite d'un arbre; les différentes valeurs de facteur d'équilibre des nœuds ( $a, b$ ) avant rotation sont  $(2, 1)$  et  $(2, 0)$ ; par rotation on a :  $(2, 1) \rightarrow (0, 0)$  et  $(2, 0) \rightarrow (1, -1)$ .

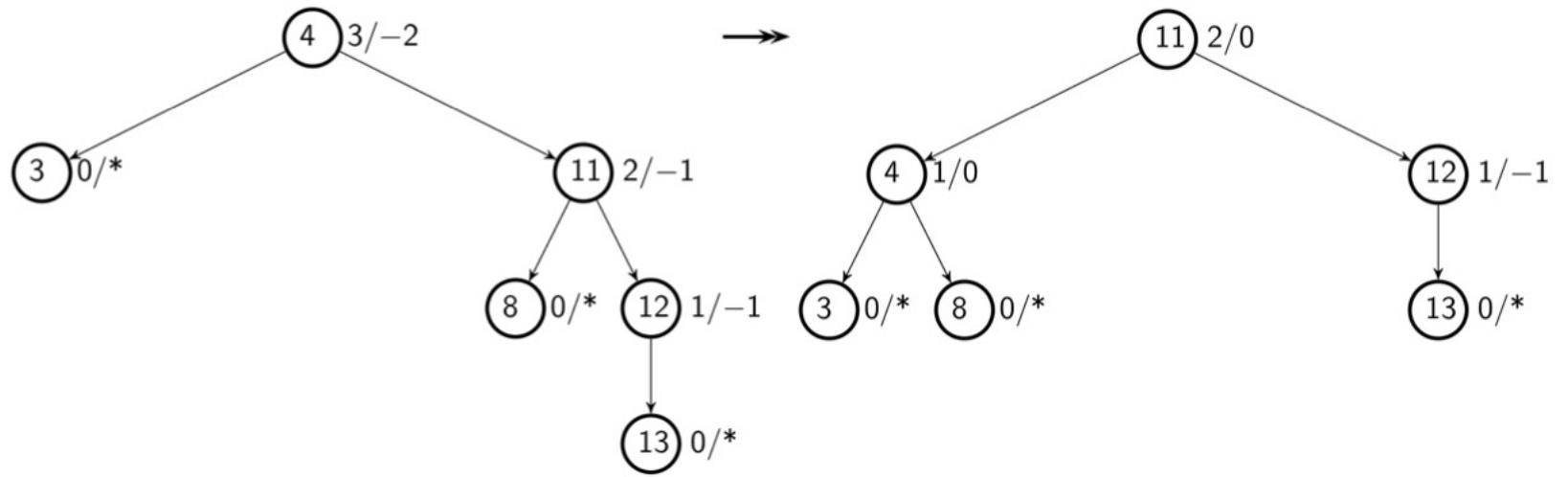


Fig. 9 – Rotation gauche d'un arbre.

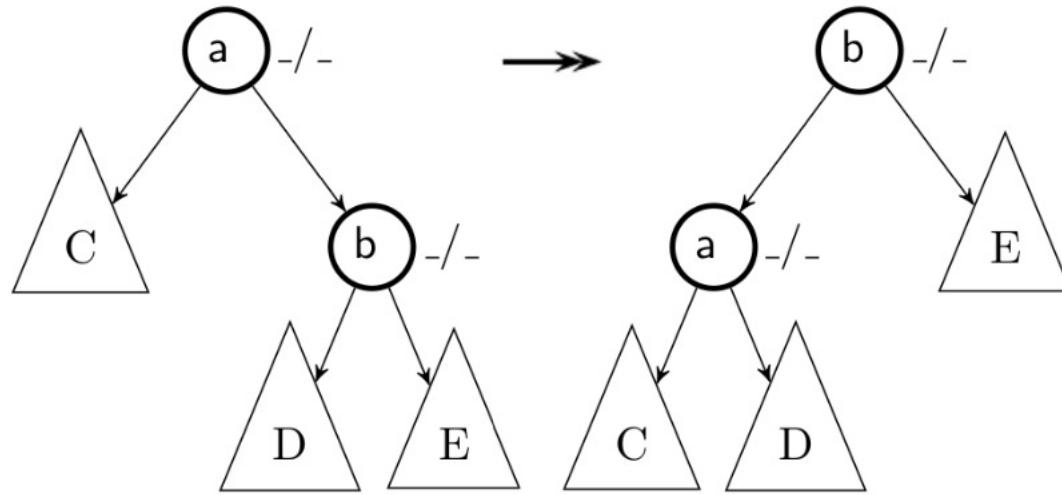


Fig. 12 – Rotation gauche d'un arbre ; les différentes valeurs de facteur d'équilibrage des nœuds ( $a, b$ ) avant rotation sont  $(-2, -1)$  et  $(-2, 0)$  ; par rotation on a :  $(-2, -1) \rightarrow (0, 0)$  et  $(-2, 0) \rightarrow (-1, 1)$ .

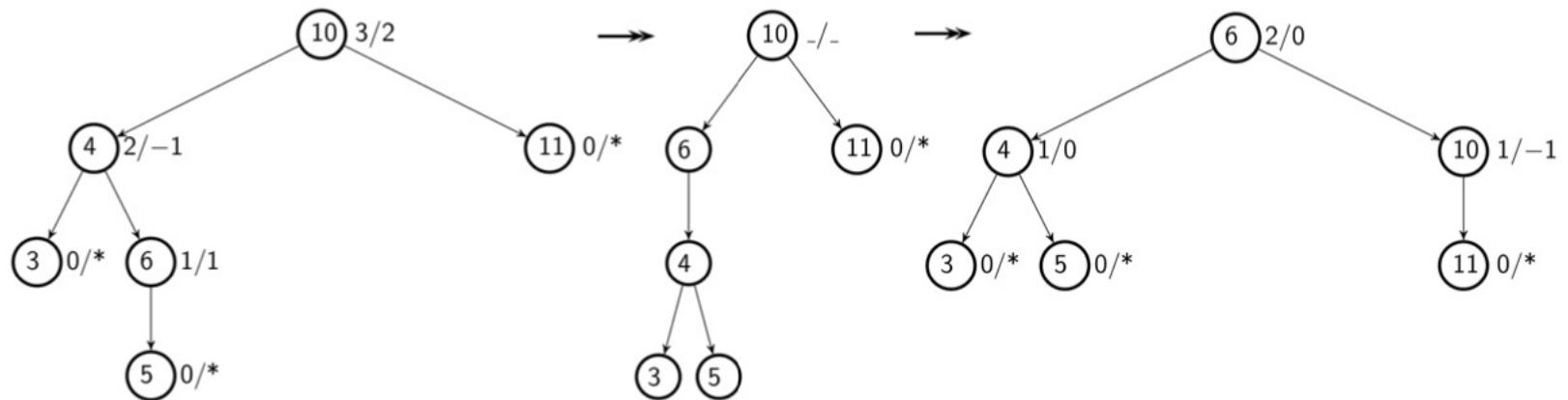


Fig. 10 – Rotation gauche droite d'un arbre.

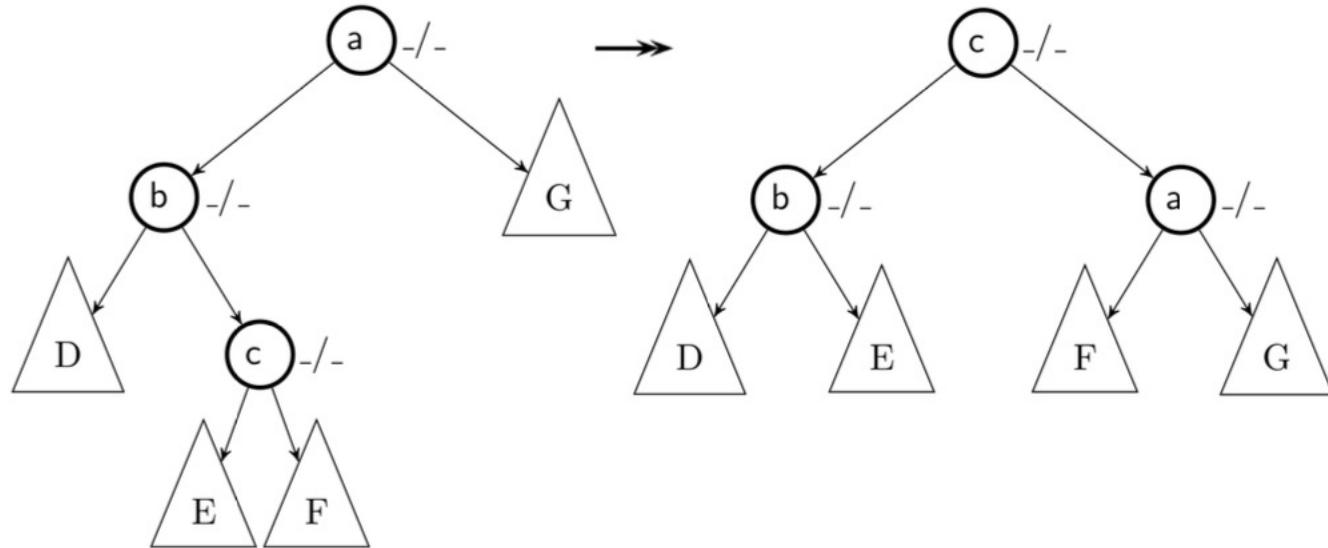


Fig. 13 – Rotation gauche droite d'un arbre ; les différentes valeurs de facteur d'équilibrage des nœuds ( $a, b, c$ ) avant rotation sont  $(2, -1, -1)$ ,  $(2, -1, 0)$  et  $(2, -1, 1)$  ; par rotation on a :  $(2, -1, -1) \rightarrow (1, 0, 0)$ ,  $(2, -1, 0) \rightarrow (0, 0, 0)$  et  $(2, -1, 1)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

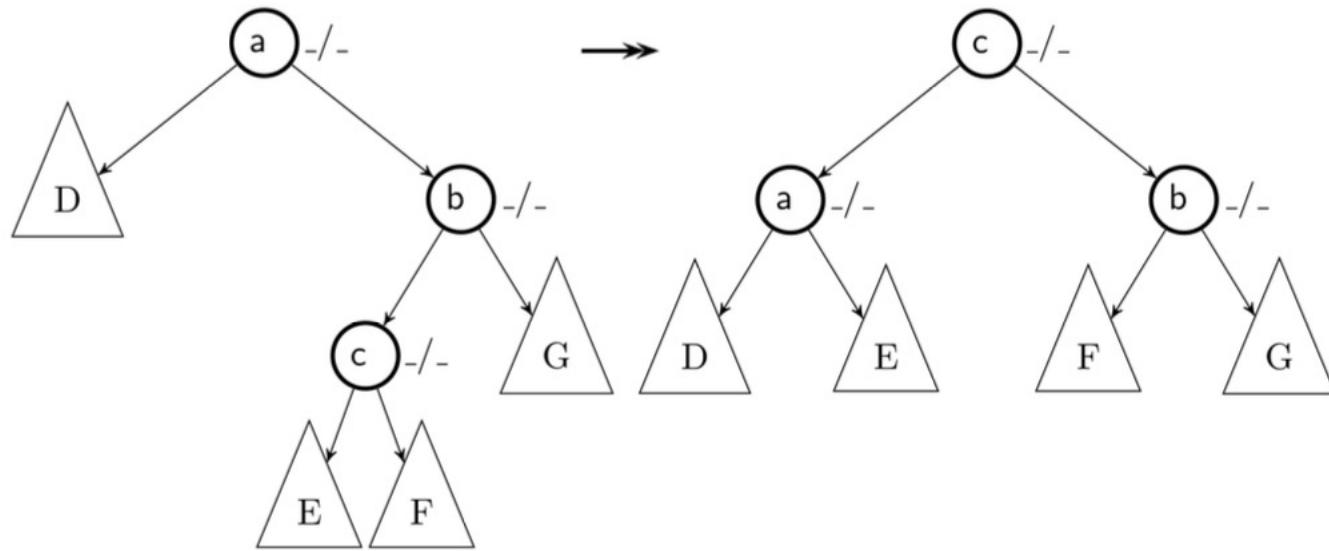


Fig. 14 – Rotation droite gauche d'un arbre ; les différentes valeurs de facteur d'équilibrage des noeuds ( $a, b, c$ ) avant rotation sont  $(-2, 1, -1)$ ,  $(-2, 1, 0)$  et  $(-2, 1, 1)$  ; par rotation on a :  $(-2, 1, -1) \rightarrow (1, 0, 0)$ ,  $(-2, 1, 0) \rightarrow (0, 0, 0)$  et  $(-2, 1,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

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)) ; D'après la construction de l'arbre
(define (L-subtree T) (first(rest(rest(rest T))))) ; D'après la construction de l'arbre
(define (R-subtree T) (first(rest(rest(rest(rest T)))))) ; D'après la construction de l'arbre
(define (leaf? T) (and (empty? (L-subtree T)) (empty? (R-subtree T)))) ; D'après la construction de l'arbre
(define (depth T) ; D'après la construction de l'arbre AVL
  (if (empty? T) -1
      (if (leaf? T) 0
          (first(rest T))))
)
(define (avltree val L R)
  (if (and (empty? L) (empty? R)) (leaf val) ; si les sous-arbres L et R sont vides
      (let ([dl (depth L)] ; dl la profondeur du sous-arbre gauche
           [dr (depth R)] ; dr la profondeur du sous-arbre droit
           )
        (list val (+ 1 (max dl dr)) (- dl dr) L R) ; val ne change pas, la profondeur augmente de 1
        )
    )
)
(define (equi T) (first (rest (rest T)))) ; Facteur d'équilibrage . D'après la construction de l'arbre

```

```

(define (equi T) (first (rest (rest T)))) ; Facteur d'équilibrage . D'aprè
(define (avltree5 val d e L R) (list val d e L R)) ; D'après la constructi
(define (root T) (first 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) ; insérer un élément val dans un arbre AVL T
  (if (empty? T) (leaf val)
    (let ([r (root T)])
      (if (< val r) ; si val est inférieur que le noeud courant.
          (if (has-L-subtree? T) ; alors on insère la valeur dans le sou
              (avltree r (add (L-subtree T) val) (R-subtree T))
              (avltree r (leaf val) (R-subtree T)) ; Si le sous-arbre ga
              )
          (if (has-R-subtree? T)
              (avltree r (L-subtree T) (add (R-subtree T) val)) ; Récipr
              (avltree r (L-subtree T) (leaf val)))
          )
      )
    )
  )
)

(define (addl T L) ; Ajouter les éléments d'une liste L à une liste T
  (if (empty? L) T
    (addl (add T (first L)) (rest L))
  )
)

```

```

; Rotation à droite

(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
  )))

```

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

## Question 22

---

En utilisant l'interface `avltree`, écrire la fonction `rotate-L` qui réalise la rotation gauche d'un arbre.

Appeler (`rotate-L (addl empty '(4 3 11 8 12 13))`).

(*fonctions utiles : root, depth, L-subtree, R-subtree*)

```

(define (rotate-L T)
  (let* ([a (root T)]
        [b (root (R-subtree T))]
        [C (R-subtree (R-subtree T))])
    [D (L-subtree T)]
    [E (L-subtree (R-subtree T))])
  [aDE (avltree a D E)])
  (avltree5 b
    (+ 1 (max (depth C) (depth aDE))))
  0
  aDE
  C
)))
;(addl empty '(4 3 11 8 12 13))
;(rotate-L (addl empty '(4 3 11 8 12 13)))

```

## Question 23

---

En utilisant l'interface `avltree`, écrire la fonction `rotate-LR` qui réalise la rotation gauche droite d'un arbre.

Appeler (`rotate-LR (addl empty '(6 7 2 4 1 3 5))`).

(*fonctions utiles : root, depth, L-subtree, R-subtree*)

```

(define (rotate-LR T)
  (let* ([a (root T)]
        [L (L-subtree T)]
        [R (R-subtree T)]
        [A (rotate-L L)]
        [LL (avltree a A R)]
        [G (rotate-R LL)])
    (avltree5 (root G)
              (+ 1 (max (depth (L-subtree G)) (depth (R-subtree G)))))
              0
              (L-subtree G)
              (R-subtree G)
              )))
;(addl empty '(6 7 2 4 1 3 5))
;(rotate-LR (addl empty '(6 7 2 4 1 3 5)))

```

## Question 24

---

En utilisant l'interface `avltree`, écrire la fonction `rotate-RL` qui réalise la rotation droite gauche d'un arbre.

Appeler `(rotate-RL (addl empty '(2 1 6 7 4 3 5)))`.

*(fonctions utiles : root, depth, L-subtree, R-subtree)*

```

(define (rotate-RL T)
  (let* ([a (root T)]
        [R (R-subtree T)]
        [L (L-subtree T)]
        [A (rotate-R R)]
        [LL (avltree a L A)]
        [G (rotate-L LL)])
    (avltree5 (root G)
              (+ 1 (max (depth (L-subtree G)) (depth (R-subtree G)))))
              0
              (L-subtree G)
              (R-subtree G)
              )))
;(addl empty '(2 1 6 7 4 3 5))
;(rotate-RL (addl empty '(2 1 6 7 4 3 5)))

```

## Question 25

---

En utilisant l'interface `avltree`, écrire la fonction `reequi` qui réalise les rotations nécessaires à l'équilibrage d'un arbre.

```
(define (reequi T)
  (cond
    [(and (= (equi T) 2) (= (equi (L-subtree T)) 1)) (rotate-R T)]
    [(and (= (equi T) 2) (= (equi (L-subtree T)) 0)) (rotate-R T)]
    [(and (= (equi T) -2) (= (equi (R-subtree T)) -1)) (rotate-L T)]
    [(and (= (equi T) -2) (= (equi (R-subtree T)) 0)) (rotate-L T)]
    [(and (= (equi T) 2) (= (equi (L-subtree T)) -1)) (rotate-LR T)]
    [(and (= (equi T) -2) (= (equi (R-subtree T)) 1)) (rotate-RL T)]
    [else T]
  ))
```

## Question 5

---

En utilisant l'interface `bt1`, définir une fonction `nbn-at-d` qui retourne la largeur d'un arbre à la profondeur `d`.

Appeler `(nbn-at-d arbre1 2)` retourne 2.

Appeler `(nbn-at-d arbre1 3)` retourne 2.

*(fonctions utiles : leaf?, has-R-subtree?, L-subtree, R-subtree)*

```
1 (define (nbn-at-d T d)
2   (if (= d 0) 1
3     (if (leaf? T) 0
4       (if (has-R-subtree? T)
5         (+ (nbn-at-d (L-subtree T) (- d 1))
6             (nbn-at-d (R-subtree T) (- d 1)))
7         (nbn-at-d (L-subtree T) (- d 1)))
8       )))
9 (nbn-at-d '(1 (2 3 (4 5 6))) 3)
10 (nbn-at-d '(1 (2 3 (4 5 6))) 4)
```

```
2  
2
```

## Question 8

---

En utilisant l'interface bt1, définir un prédictat **isst-root?** qui retourne vrai si la racine de l'arbre est supérieure à tous les éléments du sous-arbre gauche et inférieure à tous les éléments du sous-arbre droit.

Appeler (isst-root? '(2 1 4)) retourne #t.

Appeler (isst-root? '(1 2 3)) retourne #f.

Appeler (isst-root? '(4 (2 1 3) (6 5))) retourne #t.

(fonctions utiles : leaf?, root, chk?, flatten, has-R-subtree?, L-subtree, R-subtree)

```
1  (define (chk? v ope L)
2      (if (empty? L) #t
3          (if (ope v (first L)) (chk? v ope (rest L)) #f)
4      )))
5  (chk? 0 < '(1 2 3 4 5))
6  (chk? 6 >= '(1 3 7 5 2))
```

```
#t
#f
```

## Solution 8

---

```
1 (define (isst-root? T)
2   (if (leaf? T) #t
3     (if (has-R-subtree? T)
4       (and (chk? (root T) < (flatten (R-subtree T)))
5             (chk? (root T) >= (flatten (L-subtree T))))
6         (chk? (root T) >= (flatten (L-subtree T)))
7       )))
8 (isst-root? '(2 1 4))
9 (isst-root? '(1 2 3))
10 (isst-root? '(4 (2 1 3) (6 5)))
```

```
#t
#f
#t
```

## Question 9

---

En utilisant l'interface `bt1`, définir un prédictat `isst?` qui retourne vrai si l'arbre est un arbre de recherche.

Appeler `(isst? '(4 (2 1 3) (6 5)))` retourne #t.

Appeler `(isst? '(4 (1 2 3) (6 5)))` retourne #f.

*(fonctions utiles : leaf?, isst-root?, has-R-subtree?, L-subtree, R-subtree)*

```
1 (define (isst? T)
2   (if (leaf? T) #t
3       (if (not (isst-root? T)) #f
4           (if (has-R-subtree? T)
5               (and (isst? (R-subtree T)) (isst? (L-subtree T)))
6                   (isst? (L-subtree T))
7               )))
8   (isst? '(4 (2 1 3) (6 5)))
9   (isst? '(4 (1 2 3) (6 5))))
```

```
#t
#f
```

## Question 12

---

En utilisant l'interface `bt6`, définir une fonction `nb-nodes` permettant de connaitre le nombre de nœuds d'un arbre de recherche.

Appeler `(nb-nodes (tree 1 empty (leaf 2)))` retourne 2.

Appeler `(nb-nodes (tree 2 (leaf 1) empty))` retourne 2.

Appeler `(nb-nodes (addl empty '(7 6 8 3 2 5)))` retourne 6.

*(fonctions utiles : leaf?, has-LR-subtree?, has-L-subtree?, has-R-subtree?, L-subtree, R-subtree)*

## Solution 12

---

Première solution :

```
1 (define (nb-nodes T)
2   (if (leaf? T) 1
3     (if (has-LR-subtree? T)
4       (+ 1 (nb-nodes (L-subtree T)) (nb-nodes (R-subtree T)))
5       (if (has-L-subtree? T)
6         (+ 1 (nb-nodes (L-subtree T))))
7         (+ 1 (nb-nodes (R-subtree T))))
8       )))
9 (nb-nodes (tree 1 empty (leaf 2)))
10 (nb-nodes (tree 2 (leaf 1) empty))
11 (nb-nodes (addl empty '(7 6 8 3 2 5)))
```

2

2

6

Deuxième solution (exploitant la représentation interne des arbres dans `bt6`) :

```
1 (define (nb-nodes T) (length (flatten T)))
2 (nb-nodes (tree 1 empty (leaf 2)))
3 (nb-nodes (tree 2 (leaf 1) empty))
4 (nb-nodes (addl empty '(7 6 8 3 2 5)))
```

```
2
2
6
```

## Question 13

---

En utilisant l'interface `bt6`, définir une fonction `nb-leaves` permettant de connaître le nombre de feuilles d'un arbre de recherche.

Appeler `(nb-leaves (tree 1 empty (leaf 2)))` retourne 1.

Appeler `(nb-leaves (tree 2 (leaf 1) empty))` retourne 1.

Appeler `(nb-leaves (addl empty '(7 6 8 3 2 5)))` retourne 3.

*(fonctions utiles : leaf?, has-LR-subtree?, has-L-subtree?, has-R-subtree?, L-subtree, R-subtree)*

## Solution 13

---

```
1 (define (nb-leaves T)
2   (if (leaf? T) 1
3     (if (has-LR-subtree? T)
4       (+ (nb-leaves (L-subtree T)) (nb-leaves (R-subtree T)))
5         (if (has-L-subtree? T)
6           (nb-leaves (L-subtree T))
7           (nb-leaves (R-subtree T))
8         )))
9   (nb-leaves (tree 1 empty (leaf 2))))
10  (nb-leaves (tree 2 (leaf 1) empty)))
11  (nb-leaves (addl empty '(7 6 8 3 2 5))))
```

```
1
1
3
```

## Question 14

---

En utilisant l'interface `bt6`, définir une fonction `h` permettant de connaitre la hauteur d'un arbre de recherche.

Appeler `(h (addl empty '(6 3 7 8 5 2)))` retourne 2.

Appeler `(h (addl empty '(7 6 8 3 2 5)))` retourne 3.

*(fonctions utiles : leaf?, has-LR-subtree?, has-L-subtree?, has-R-subtree, L-subtree, R-subtree)*

## Solution 14

---

```
1 (define (h T)
2   (if (leaf? T) 0
3     (if (has-LR-subtree? T)
4       (+ 1 (max (h (L-subtree T))
5                  (h (R-subtree T))))
6       (if (has-L-subtree? T)
7         (+ 1 (h (L-subtree T))))
8         (+ 1 (h (R-subtree T)))
9         ))))
10  (h (addl empty '(6 3 7 8 5 2)))
11  (h (addl empty '(7 6 8 3 2 5)))
```

2

3

## Question 18

---

En utilisant l'interface `bt6`, écrire un prédictat `isin?` qui vérifie si une valeur `v` est dans un arbre.

Appeler `(isin? (addl empty '(6 3 7 8 5 2)) 2)` retourne `#t`.

Appeler `(isin? (addl empty '(6 3 7 8 5 2)) 1)` retourne `#f`.

*(fonctions utiles : root, leaf?, has-L-subtree?, has-R-subtree?, L-subtree, R-subtree)*

## Solution 18

---

```
1 (define (isin? T e)
2   (if (= (root T) e) #t
3     (if (leaf? T) #f
4       (if (< e (root T))
5         (if (has-L-subtree? T)
6           (isin? (L-subtree T) e)
7             #f)
8           (if (has-R-subtree? T)
9             (isin? (R-subtree T) e)
10            #f)
11      ))))
12 (isin? (addl empty '(6 3 7 8 5 2)) 2)
13 (isin? (addl empty '(6 3 7 8 5 2)) 1)
```

```
#t
#f
```

## Question 21

---

En utilisant l'interface `bt6`, écrire une fonction `min-tree` qui retourne le minimum des valeurs d'un arbre.

Appeler `(min-tree (addl empty '(6 3 7 8 5 2)))` retourne 2.

Appeler `(min-tree (addl empty '(7 6 8 3 2 5)))` retourne 2.

(fonctions utiles : `root`, `leaf?`, `has-L-subtree?`, `L-subtree`)

## Solution 21

---

```
1 (define (min-tree T)
2   (if (has-L-subtree? T)
3     (min-tree (L-subtree T))
4     (root T)
5   )))
6 (min-tree (addl empty '(6 3 7 8 5 2)))
7 (min-tree (addl empty '(7 6 8 3 2 5)))
```

2

2