TP 4 Algorithmes et structures de données

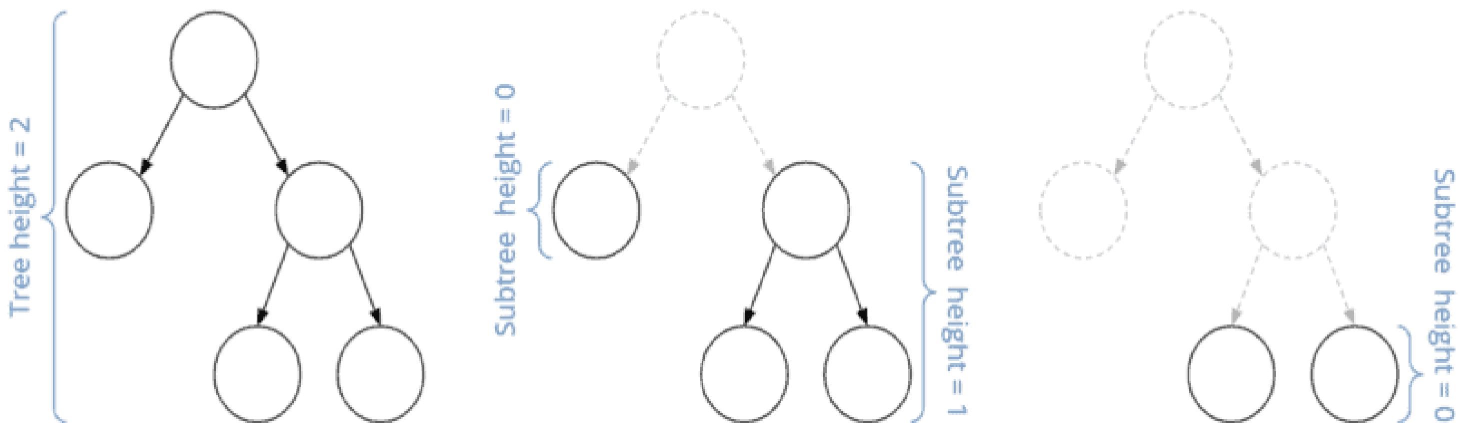
(préparation à l'agrégation)

0. Qu'est-ce qu'un arbre AVL ?

Un arbre AVL est un arbre binaire de recherche (ABR) dans lequel les hauteurs des sous-arbres gauche et droit de chaque nœud diffèrent d'au plus un. Après chaque opération d'insertion et de suppression, cet invariant est vérifié, c'est à dire, l'équilibre est rétabli grâce à des rotations si nécessaire.

Hauteur d'un arbre AVL

La hauteur d'un (sous) arbre est la distance entre la racine et la feuille la plus basse ; un (sous-)arbre constitué uniquement d'un nœud racine a une hauteur de 0.



Facteur d'équilibre de l'arbre AVL

Le facteur d'équilibre "BF" (de l'anglais *Balance Factor*) d'un nœud désigne la différence des hauteurs "H" du sous-arbre droit et gauche ("node.right" et "node.left") :

BF(nœud) = H(nœud.droite) - H(nœud.gauche)

La hauteur d'un sous-arbre inexistant est de -1 (une de moins que la hauteur d'un sous-arbre d'un seul nœud).

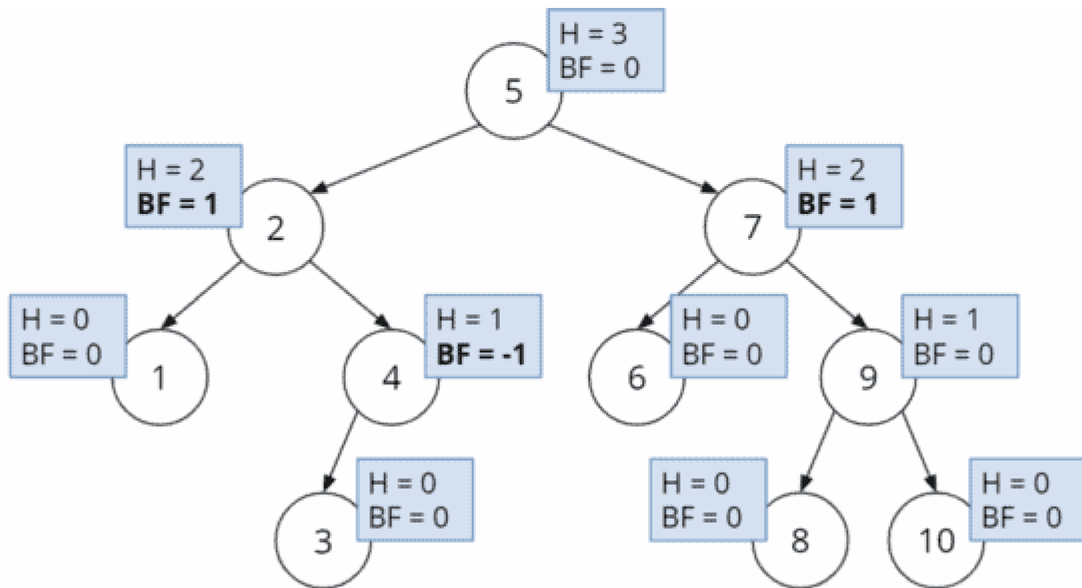
Il y a trois cas :

- Si le facteur d'équilibre est < 0, le nœud est dit *lourd à gauche* .
- Si le facteur d'équilibre est > 0, le nœud est dit *lourd à droite* .
- Un facteur d'équilibre de 0 représente un nœud *équilibré* .

Dans un arbre AVL, le facteur d'équilibre à chaque nœud est -1, 0 ou 1.

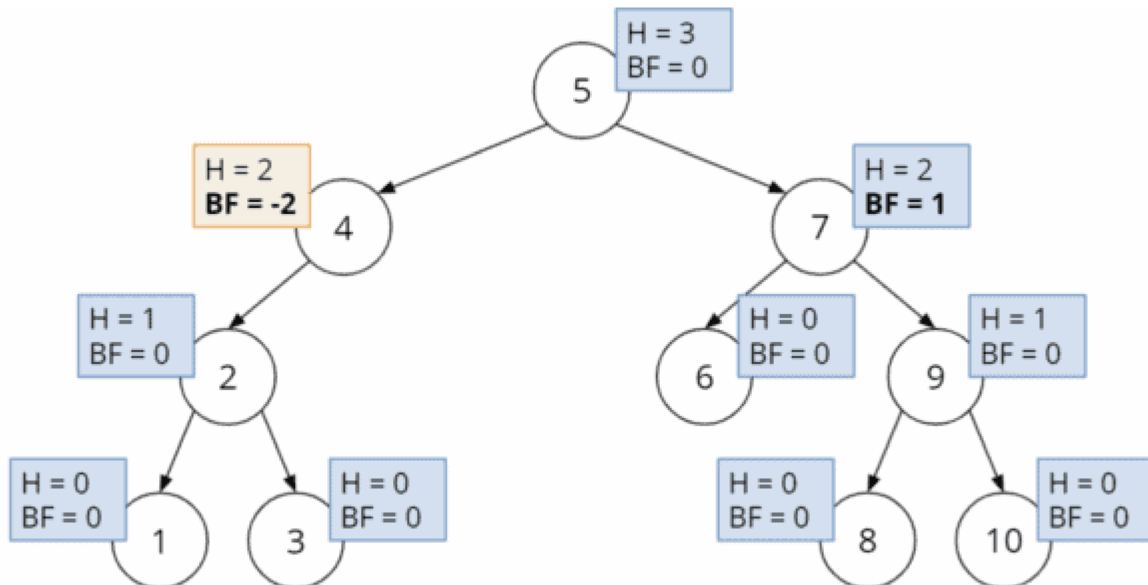
Exemple d'arbre AVL

L'exemple suivant montre un arbre AVL avec une hauteur et un facteur d'équilibre spécifiés à chaque nœud :



Les nœuds 2 et 7 dans cet exemple sont lourds à droite, le nœud 4 est lourd à gauche. Tous les autres nœuds sont équilibrés.

L'arbre suivant, cependant, *n'est pas* un arbre AVL puisque le critère AVL ($-1 \leq BF \leq 1$) n'est pas rempli au nœud 4. Son sous-arbre gauche a une hauteur de 1 et le sous-arbre droit vide a une hauteur de - 1. La différence entre eux est de -2.



1. Implémentation d'un ABR classique

Il y a mille manières d'implémenter un ABR. Je vous ai fourni une implémentation de 100 lignes sur une feuille écrites. Il faut l'étudier mais le « copier – coller » ne va pas vous aider ; la chose utile est de lire et comprendre le code. J'essaie de l'expliquer. On utilise une structure *node* ainsi :

```
#include<stdlib.h>
#include<stdio.h>
#define ROOT_CHILD 1
#define LEFT_CHILD 2
#define RIGHT_CHILD 3
typedef struct node_{
    int val;
    struct node_* left;
    struct node_* right;
    struct node_* up;           //pas obligatoire mais utile
    int  type;                  //pas obligatoire mais utile pour savoir si le nœud
                                //est un fils à gauche, à droite, ou la racine
} node;

node * abr = NULL;
```

Cet arbre est vide. Un point de départ pour démarrer c’est d’étendre le code suivant:

```
node* new_node(int value, node * up){
    node* tmp  = (node*)calloc(1, sizeof(node));
    tmp -> val = value;
    tmp -> left   = NULL;
    tmp -> right  = NULL;
    tmp -> up     = up;
    tmp  -> type  = LEFT_CHILD ; //ou RIGHT_NODE à compléter
    return tmp;
}
```

Si l’arbre contient beaucoup de nœuds, vous pouvez accéder à diverses nœuds grâce à des opérations comme abr->left->left ou abr->right->left, etc.

Vous pouvez avoir besoin de fonctions comme

```
node* go_left(node* r){
    if(r->left!=NULL)
        r = r->left;
    return r;
}

node* go_right(node* r){
    if(r->right!=NULL)
        r = r->right;
    return r;
}
node* go_left_to_the_end(node* r){
    while(r->left!=NULL)
        r = r->left;
    return r;
}

node* go_right_to_the_end(node* r){
    while(r->right!=NULL)
        r = r->right;
    return r;
}
```

Il faut d'abord implémenter un ABR classique et vérifier s'il fonctionne. On doit être capable d'ajouter des valeurs et des les afficher dans le bon ordre. La suppression sera implémentée plus tard. Une des fonctions les plus difficiles à coder est de trouver la valeur suivante dans le bon ordre. Voici une solution; n'hésitez pas à demander des explications.

```
node* next_val(node* r){
    if(r->right != NULL)                       //if right child exists
        return go_left_to_the_end(r->right);   //reach the left-most elem on right branch
    if(r->type == LEFT_CHILD)          //if no right child but we are on a left node
        return r->up;                 //we can go up and that is sufficient
    //We get here: we are on right branch
    //Move up until we get into a node that is a left branch or root
    while(r->type == RIGHT_CHILD){
        r = r-> up;
    }
    return r-> up;
}
```

Voici un exemple de fonction main (n'hésitez pas pour des explications):

```
abr = new_node(7,NULL,ROOT_CHILD);
add(abr,8);
add(abr,4);
add(abr,6);
add(abr,5);
add(abr,9);
print_tree(abr,0);
node*iterator = go_left_to_the_end(abr);
while(iterator!=NULL){
    printf("%d ",iterator->val);
    iterator = next_val(iterator);
}
```

2. Implémentation d'un arbre AVL : 4 types de rotations

On suppose que vous avez implémenté l'ABR classique (que insertion et affichage). Calculer les hauteurs de chaque nœud. S'il existe un nœud *nœud* tel que *BF(nœud) = H(nœud.droite) – H(nœud.gauche)* est supérieur à 1 ou inférieur à -1, il faut rééquilibrer l'arbre. Nous distinguons la rotation à droite et à gauche.

Rotation à droite

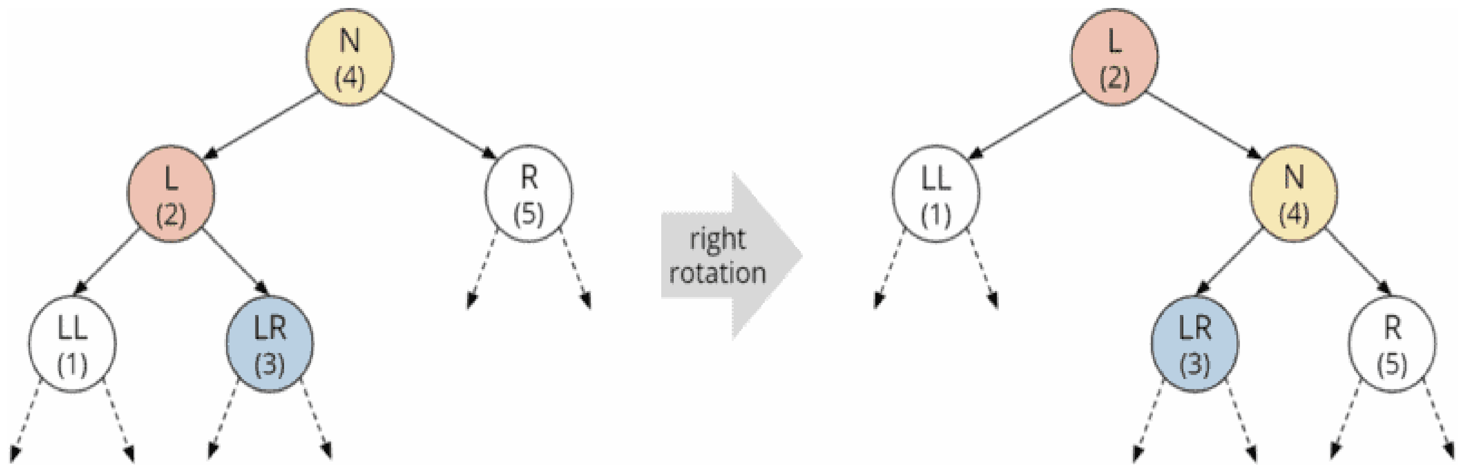
L'image suivante montre une rotation à droite. Le (sous) arbre affiché contient les nœuds suivants :

- *N* : le nœud où un déséquilibre a été détecté
- *L* : le nœud enfant gauche de *N* (L = *Left*)
- *LL* : le nœud enfant gauche de *L (LL = Left Left)*
- *LR* : le nœud enfant droit de *L (LR = Left Right)*
- *R* : le nœud enfant droit de *N*

Sous chaque lettre, j'ai donné un exemple de valeur de nœud entre parenthèses. Cela montre clairement que la séquence suivante s'applique avant la rotation :

LL (1) < L (2) < LR (3) < N (4) < R (5)

Pendant la rotation, le nœud *L* se déplace vers la racine et la racine précédente *N* devient l'enfant droit de *L* . L'ancien enfant droit de *L* , *LR* devient le nouvel enfant gauche de *N* . Les deux nœuds restants, *LL* et *R* restent inchangés par rapport à leur nœud parent.



Le code C n’est pas si compliqué qu’on peut le penser. Voici la trame du code C à écrire:

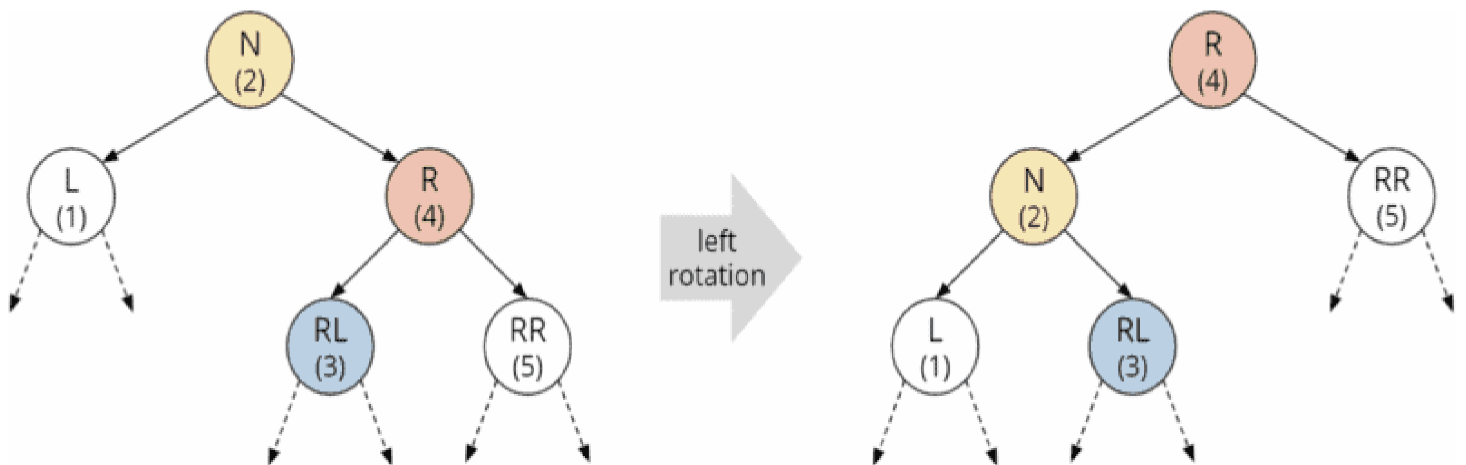
```
node* rotateRight(node*n) {
  node*leftChild = n->left;

  n->left = leftChild->right;
  leftChild->right = n;

  return leftChild;
}
```

Il suffit d’appeler n = rotateRight(n) sur le nœuds n que vous voulez équilibrer.

Rotation à gauche

La rotation à gauche fonctionne de la même manière :

Le nœud *R* devient la racine ; la racine précédente *N* devient l'enfant gauche de *R* . L'enfant gauche précédent de *R* , *RL* devient le nouvel enfant droit de *N* . Les positions relatives des nœuds *RR* et *L* ne changent pas.



Voici la trame du code C à écrire:

```
node* rotateLeft(node* n) {
```

```
  node* rightChild = n->right;

  n->right = rightChild->left;
  rightChild->left = n;

  return rightChild;
}
```

Comment savoir quelle rotation appliquer?

Si, à un nœud, nous déterminons que l'invariant AVL n'est plus satisfait (c'est-à-dire que le facteur d'équilibre est inférieur à -1 ou supérieur à +1), nous devons rééquilibrer. L'invariant AVL d'un nœud *nœud* est *BF(nœud) = H(nœud.droite) – H(nœud.gauche)*

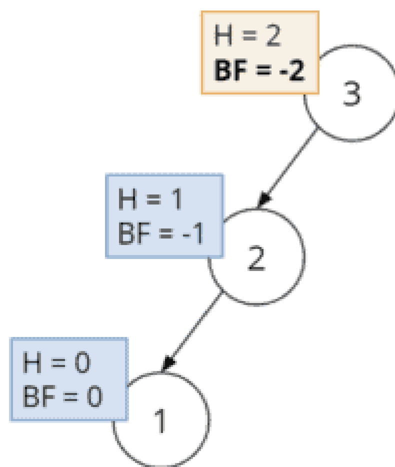
Nous différencions quatre cas :

- Équilibrer un nœud lourd à gauche :
  - Rotation à droite
  - Rotation gauche-droite
- Équilibrer un nœud lourd à droite :
  - Rotation à gauche
  - Rotation droite-gauche

Dans les sections qui suivent, je décris les quatre cas à l'aide de divers exemples.

Rééquilibrage par rotation à droite

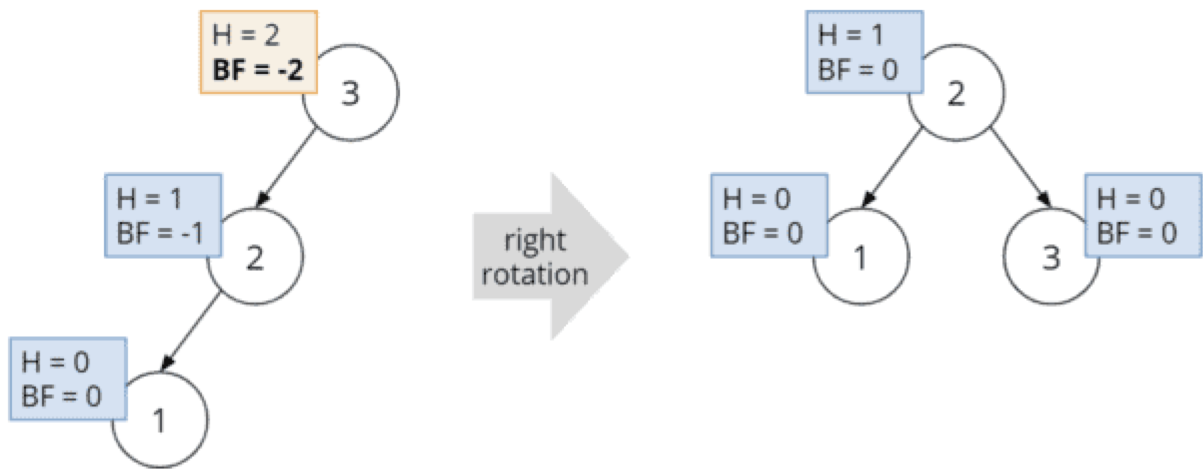
Nous insérons les nœuds 3, 2 et 1 dans un arbre vide. Sans rééquilibrage, l'arbre ressemble alors à ceci :



Nous examinons le facteur d'équilibre à partir du dernier nœud inséré 1 vers le haut :

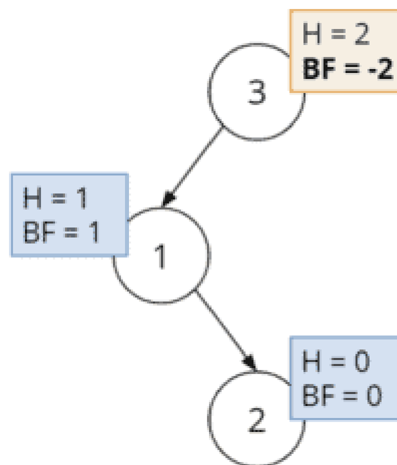
- Le facteur d'équilibre au nœud 1 est 0.
- Le facteur d'équilibre au nœud 2 est de -1 ; le nœud 2 est donc lourd à gauche. Cependant, l'invariant AVL ($-1 \leq BF \leq 1$) est toujours rempli.
- Le facteur d'équilibre au nœud 3 est de -2 ; l'invariant AVL n'est plus rempli à ce nœud.

Dans ce cas, il faut effectuer une rotation à droite autour du nœud 3 :

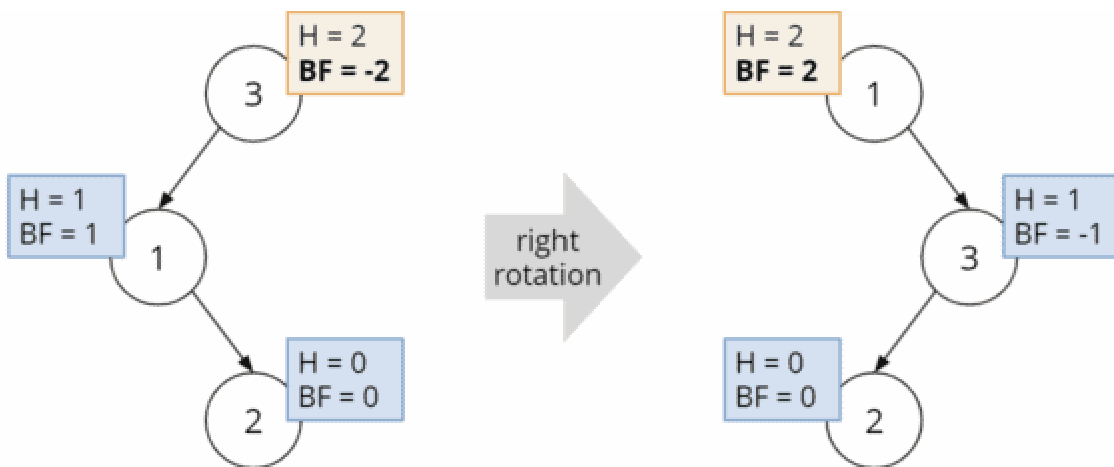


Rééquilibrage par rotation gauche-droite

Nous avons également une racine lourde à gauche dans l'exemple suivant, mais la situation semble un peu différente. Cette fois, nous insérons les nœuds dans l'ordre 3, 1, 2 :



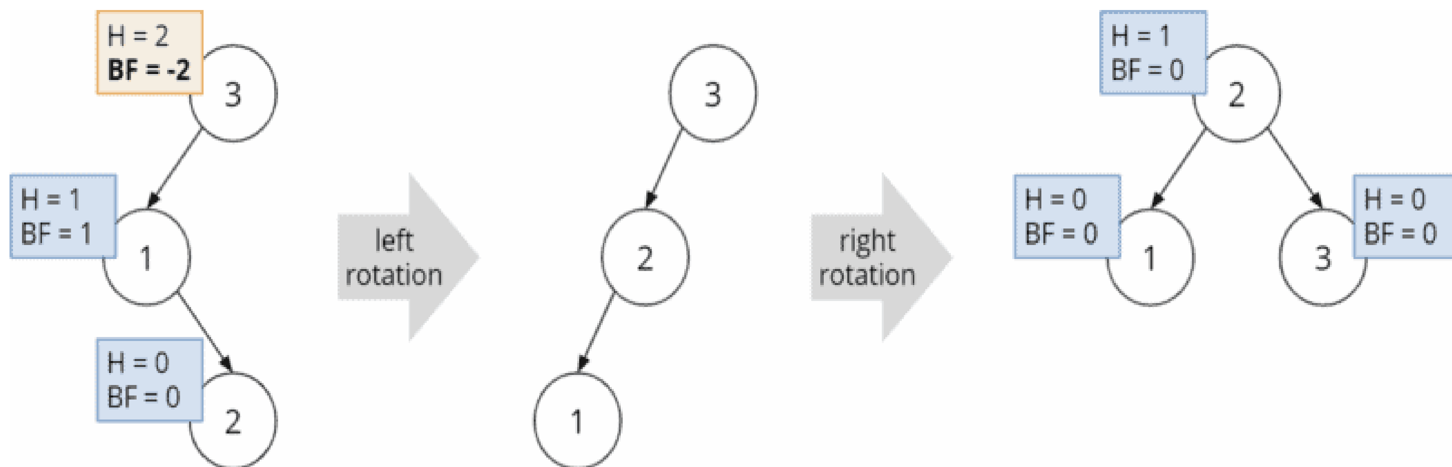
On remarque que le critère AVL n'est pas rempli à la racine (ayant un facteur d'équilibre de -2). Si nous effectuions maintenant – comme dans l'exemple précédent – une rotation à droite, l'arbre ressemblerait alors à ceci :



L'enfant droit du nœud 1 – le nœud 2 – est devenu l'enfant gauche du nœud 3. Au lieu d'une racine lourde à gauche avec BF -2, nous avons maintenant une racine lourde à droite avec BF +2. Nous avons raté la cible.

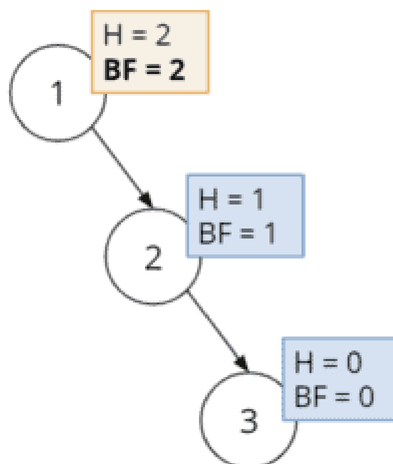
Que pouvons-nous faire à la place ?

La procédure correcte dans ce cas (l'enfant gauche de la racine est lourd à droite) est ce qu'on appelle une rotation gauche-droite. Tout d'abord, nous tournons vers la gauche autour du nœud 1 puis vers la droite autour du nœud 3 :

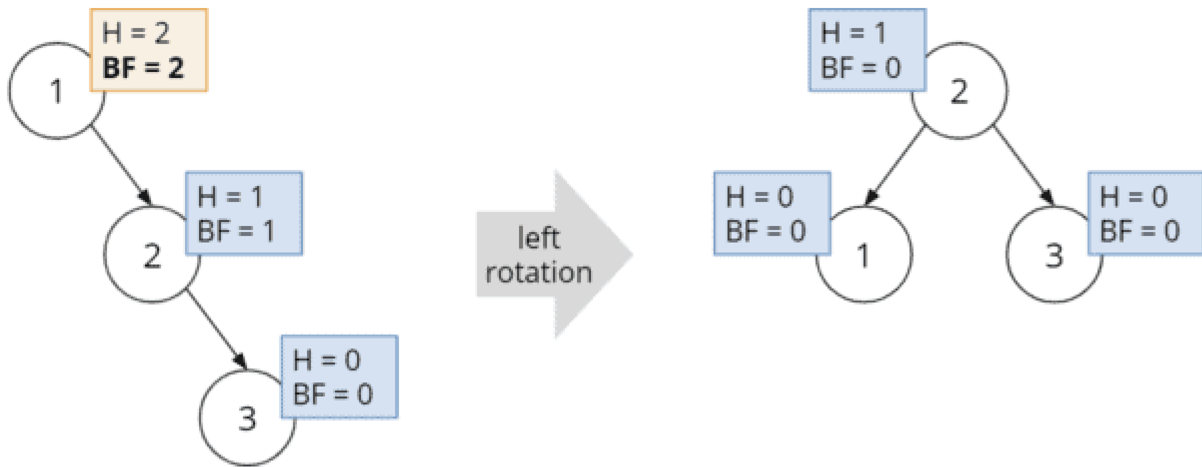


Rééquilibrage par rotation à gauche

Pour les nœuds lourds à droite, nous procédons de la même manière. Nous insérons d'abord les nœuds dans l'ordre 1, 2, 3 et obtenons l'arbre déséquilibré suivant :

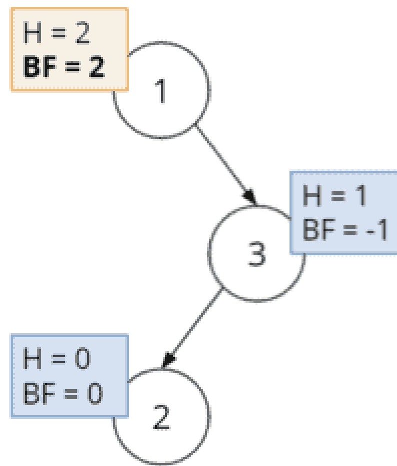


Le facteur d'équilibre de la racine est de +2. On peut rétablir l'équilibre par une simple rotation vers la gauche :

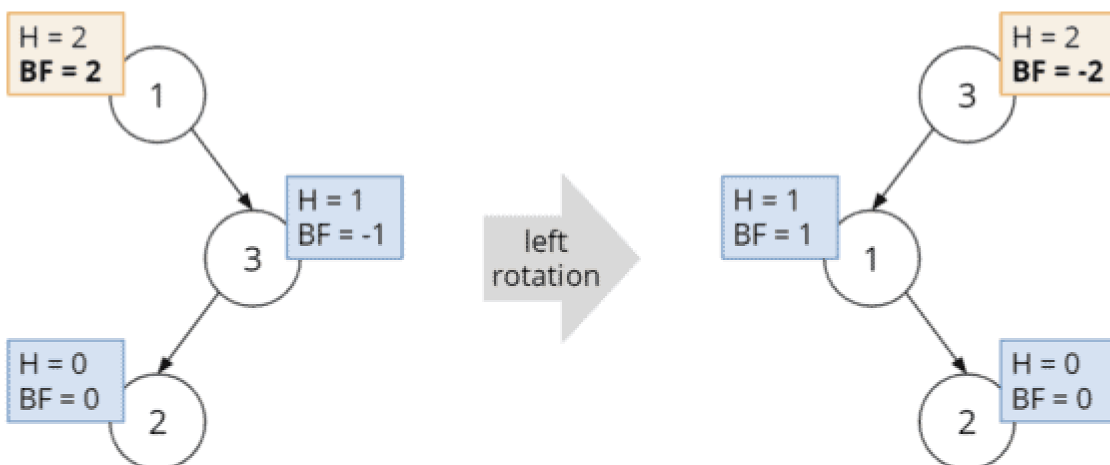


Rééquilibrage par rotation droite-gauche

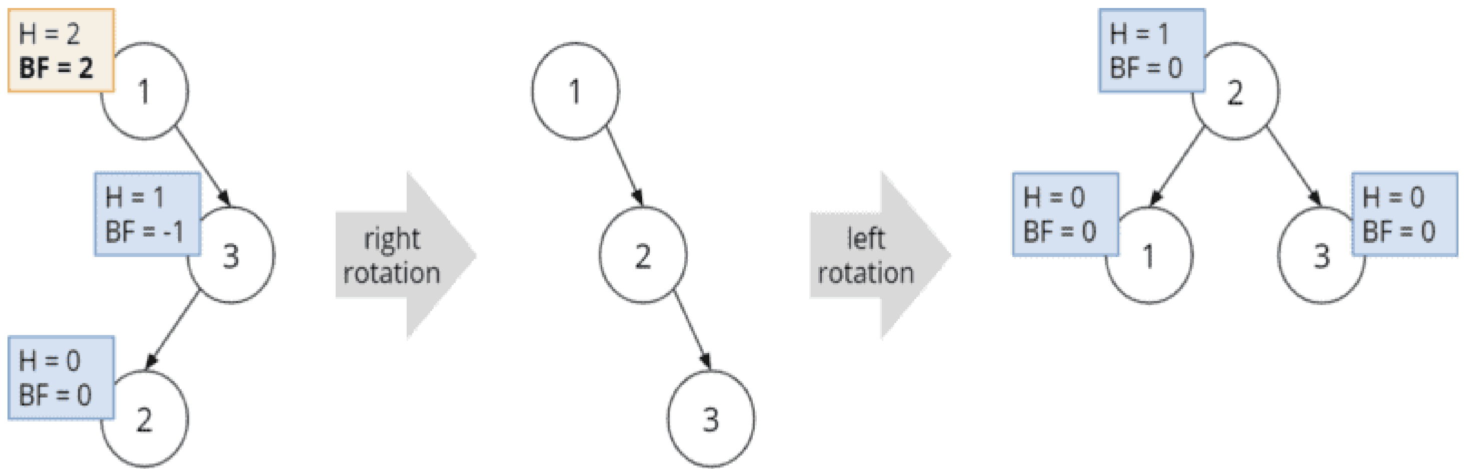
Le quatrième et dernier exemple montre un arbre AVL avec les nœuds insérés dans l'ordre 1, 3, 2 :



Le facteur d'équilibre de la racine est à nouveau de +2. Mais avec une rotation à gauche comme dans l'exemple précédent, ce qui suit se produirait :



De manière analogue au deuxième cas, la procédure correcte dans ce cas (l'enfant droit de la racine est lourd à gauche) est une rotation droite-gauche. On tourne vers la droite autour du nœud 3 puis vers la gauche autour du nœud 1 :



3. Implémentation des suppressions AVL

Écrire la fonction qui permet de supprimer un nœud. Après chaque suppression, la forme de l'arbre peut changer. Donc il faut recalculer toutes les hauteurs des sous-arbre et appliquer les rotations précédentes.

4. Vérifier l'efficacité

Écrire une fonction main qui permet d'ajouter n=1000000 de valeurs aléatoires dans un arbre. Écrire une fonction qui permet de chercher une valeur dans l'arbre. Appelez cette fonction n fois à chaque fois avec une valeur aléatoire. Comparer le temps de final de calcul de l'arbre ABR par rapport à l'arbre AVL.