IFT339
Structures de données

Thème 10 : Équilibre et Itération dans les arbres binaires de recherche

Aïda Ouangraoua
Département d'informatique

UNIVERSITÉ DE
SHERBROOKE

Équilibre dans les arbres

- Objectif : opération en O(log(n))
  - Possible seulement si la hauteur de l'arbre h = O(log(n))

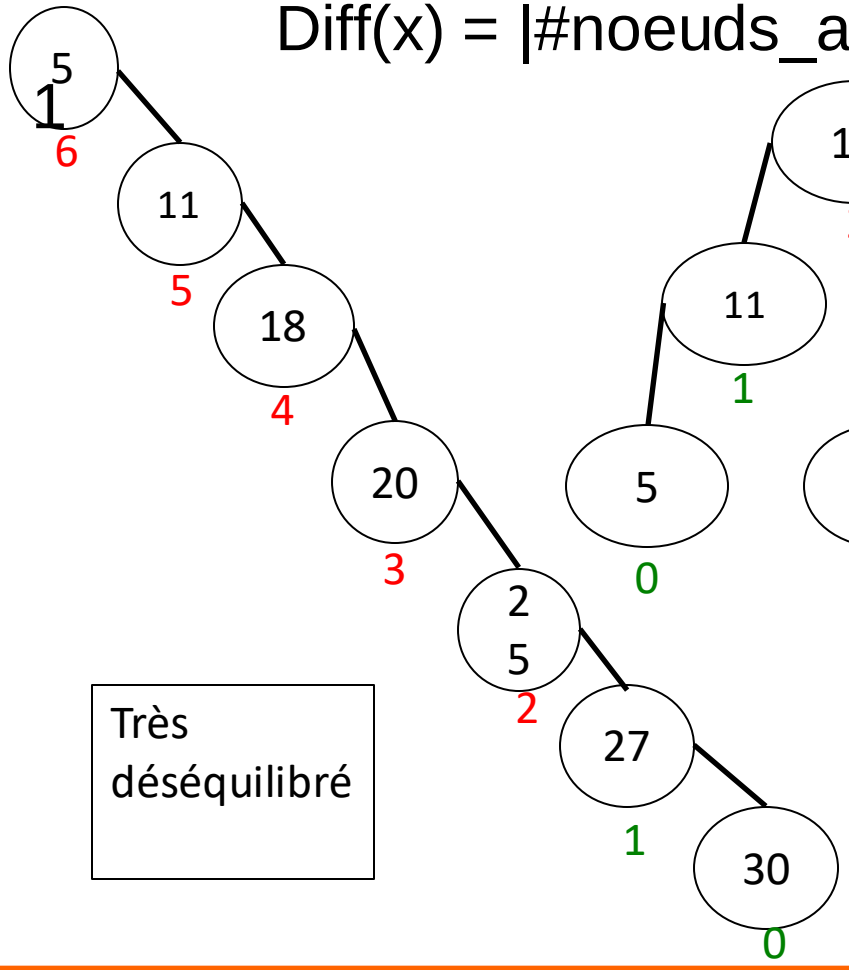
- Solution : arbre équilibré : pour chaque nœud x:

  Diff(x) = |#noeuds_a_gauche(x) - #noeuds_a_droite(x)| <= 1

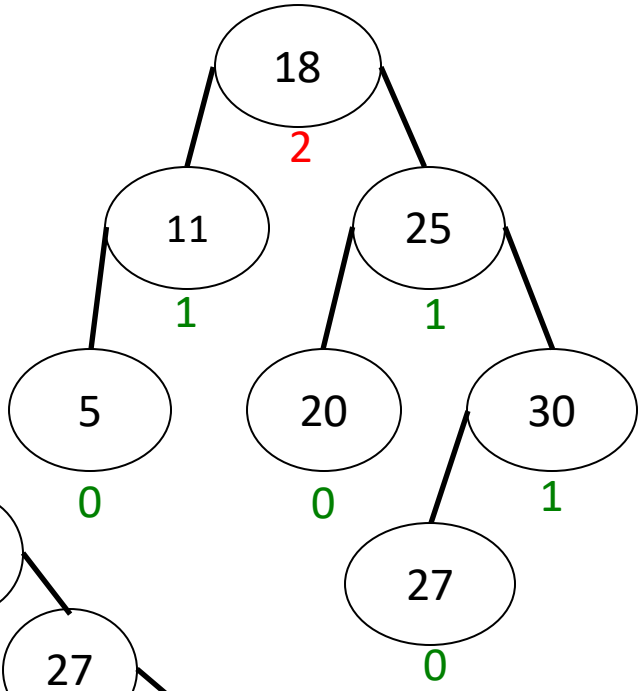
Équilibre dans les arbres

- Objectif : opération en O(log(n))
  - Possible seulement si la hauteur de l'arbre h = O(log(n))
- Solution : arbre équilibré : pour chaque nœud x:

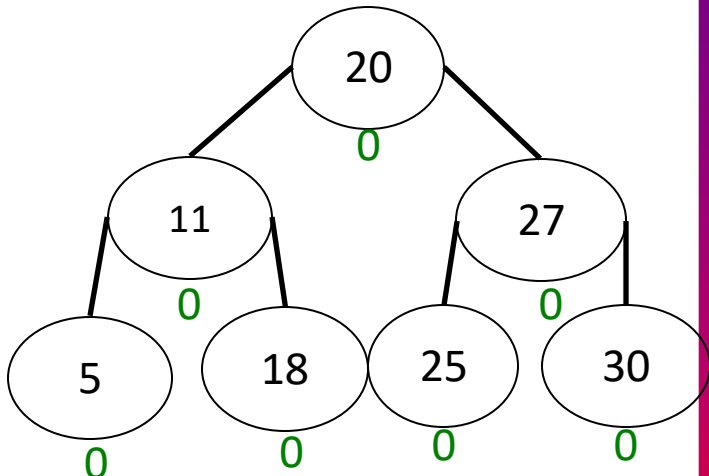
Diff(x) = |#noeuds_a_gauche(x) - #noeuds_a_droite(x)| <=



Très déséquilibré



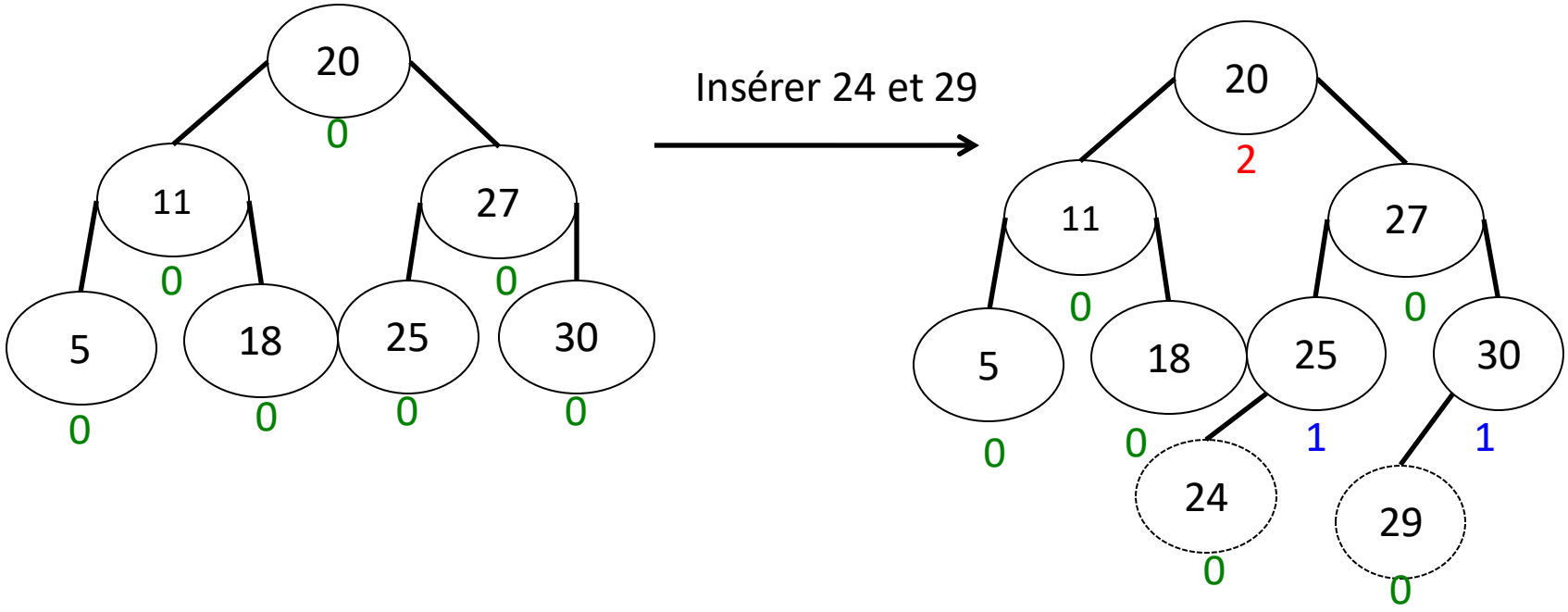
Partiellement déséquilibré



Totalement équilibré

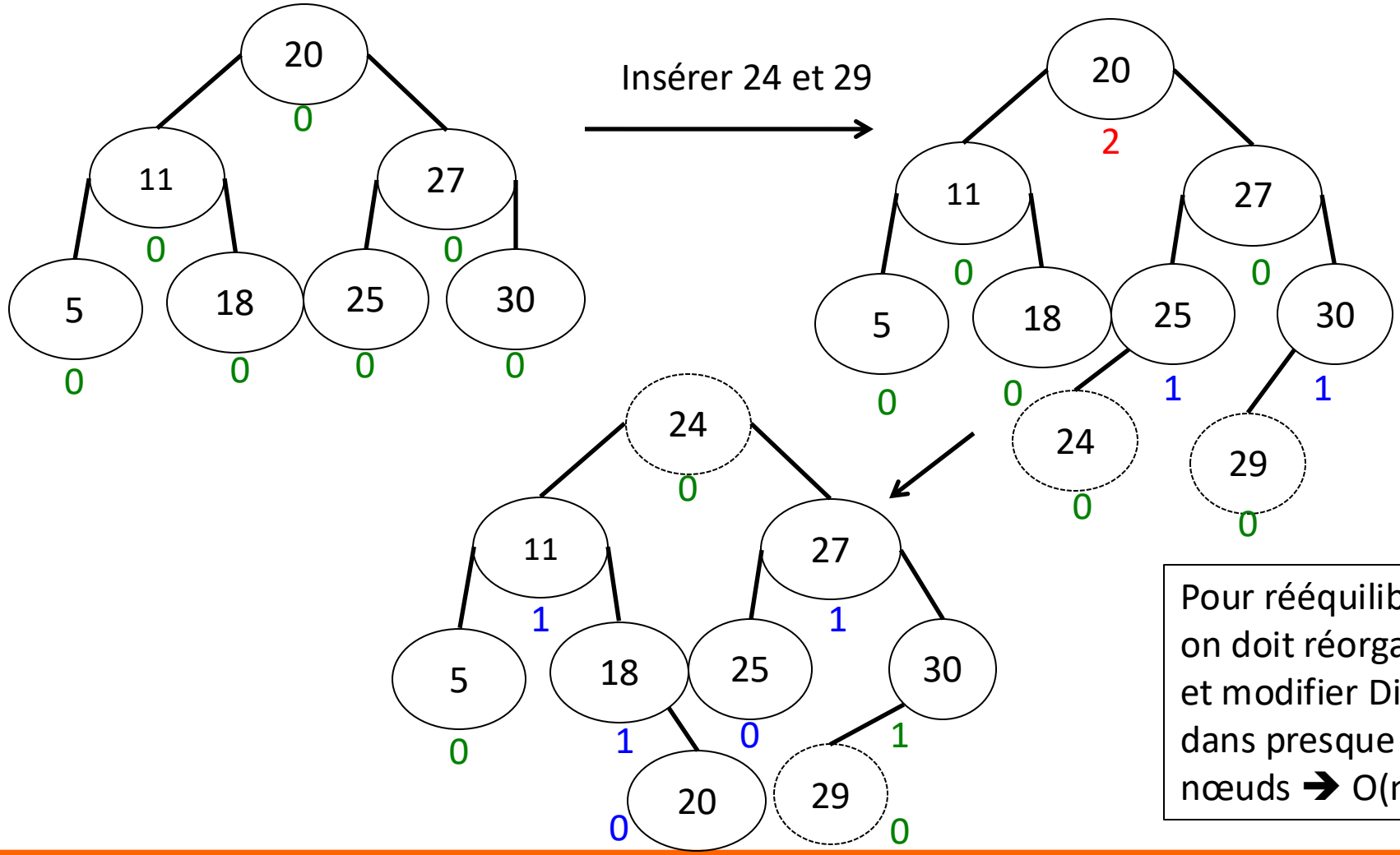
Équilibre dans les arbres

- Idéal : équilibre total
- Mais maintenir l'équilibre total nécessite une insertion en O(n)



Équilibre dans les arbres

- Idéal : équilibre total
- Mais maintenir l'équilibre total nécessite une insertion en O(n)



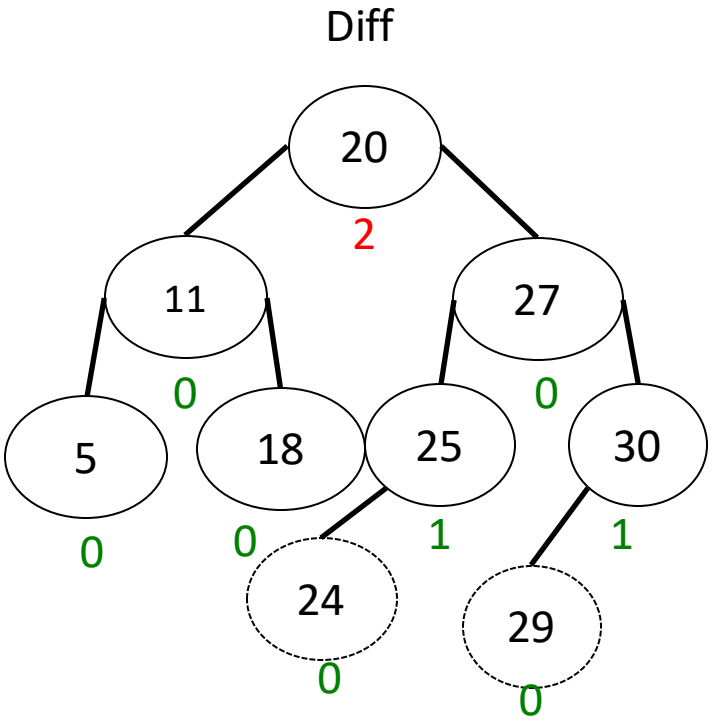
Pour rééquilibrer, on doit réorganiser et modifier Diff(x) dans presque tous les nœuds ➔ O(n)

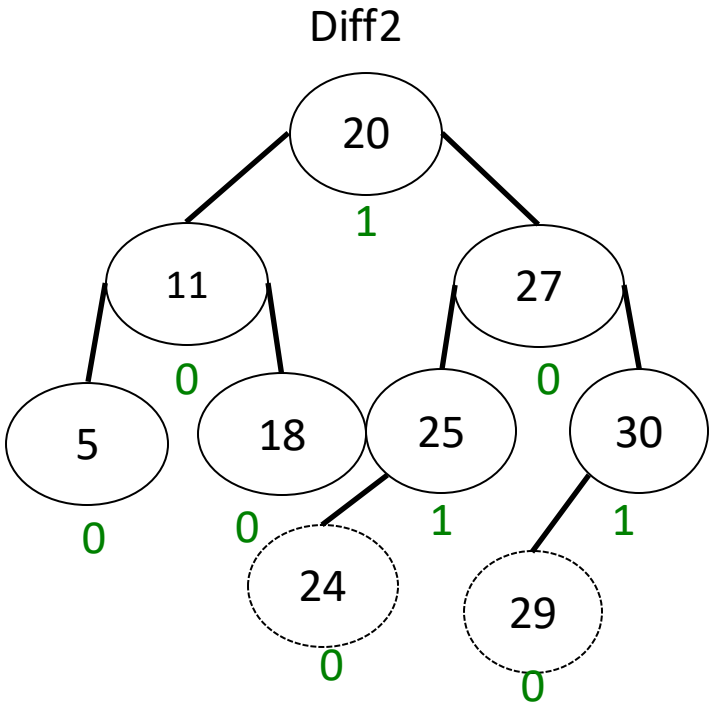
Équilibre dans les arbres

- Solution : équilibre partiel sur

Diff2(x) = |hauteur_a_gauche(x) - hauteur_a_droite(x)| <= 1

- Arbre AVL (Adelson-Veleski et Landis)

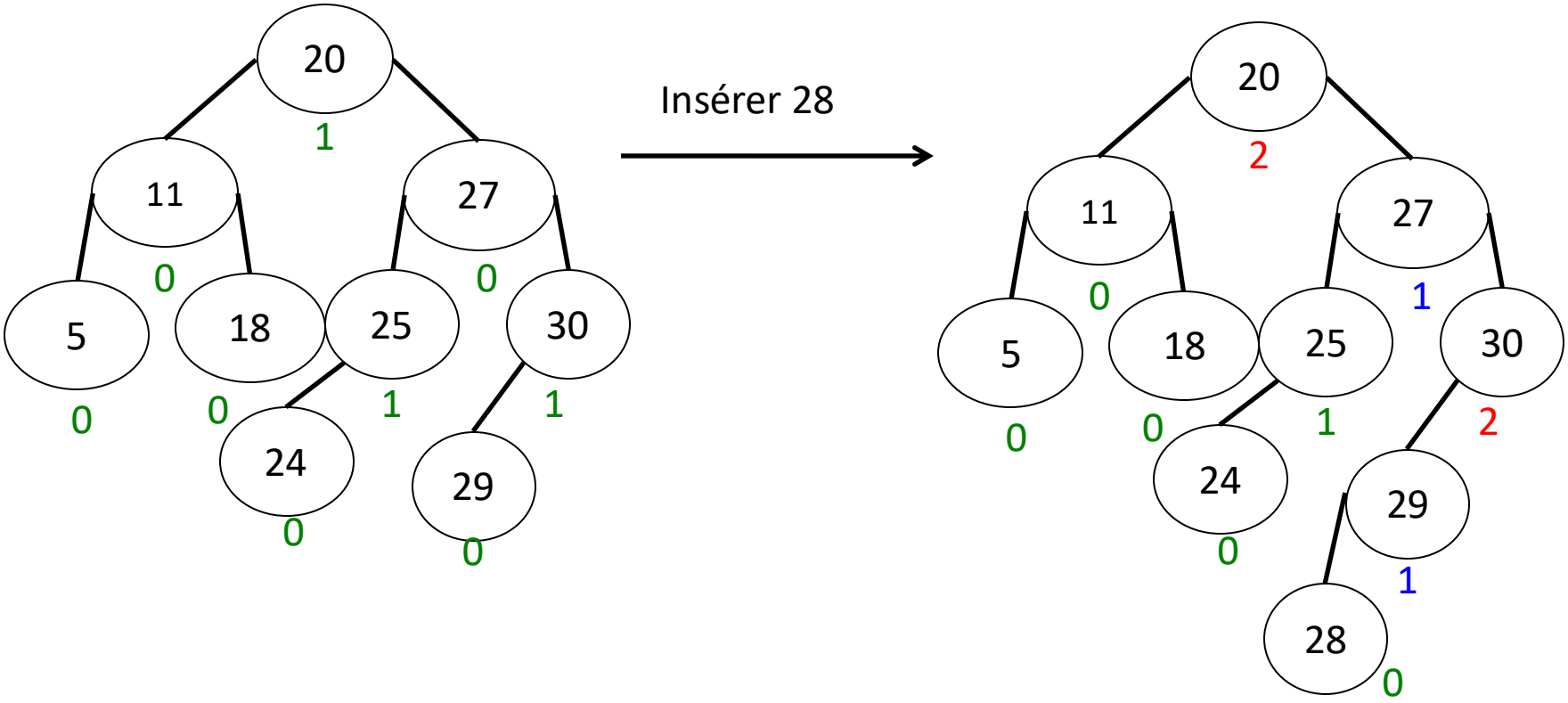




Arbre AVL

- ABR tel que pour tout nœud x,

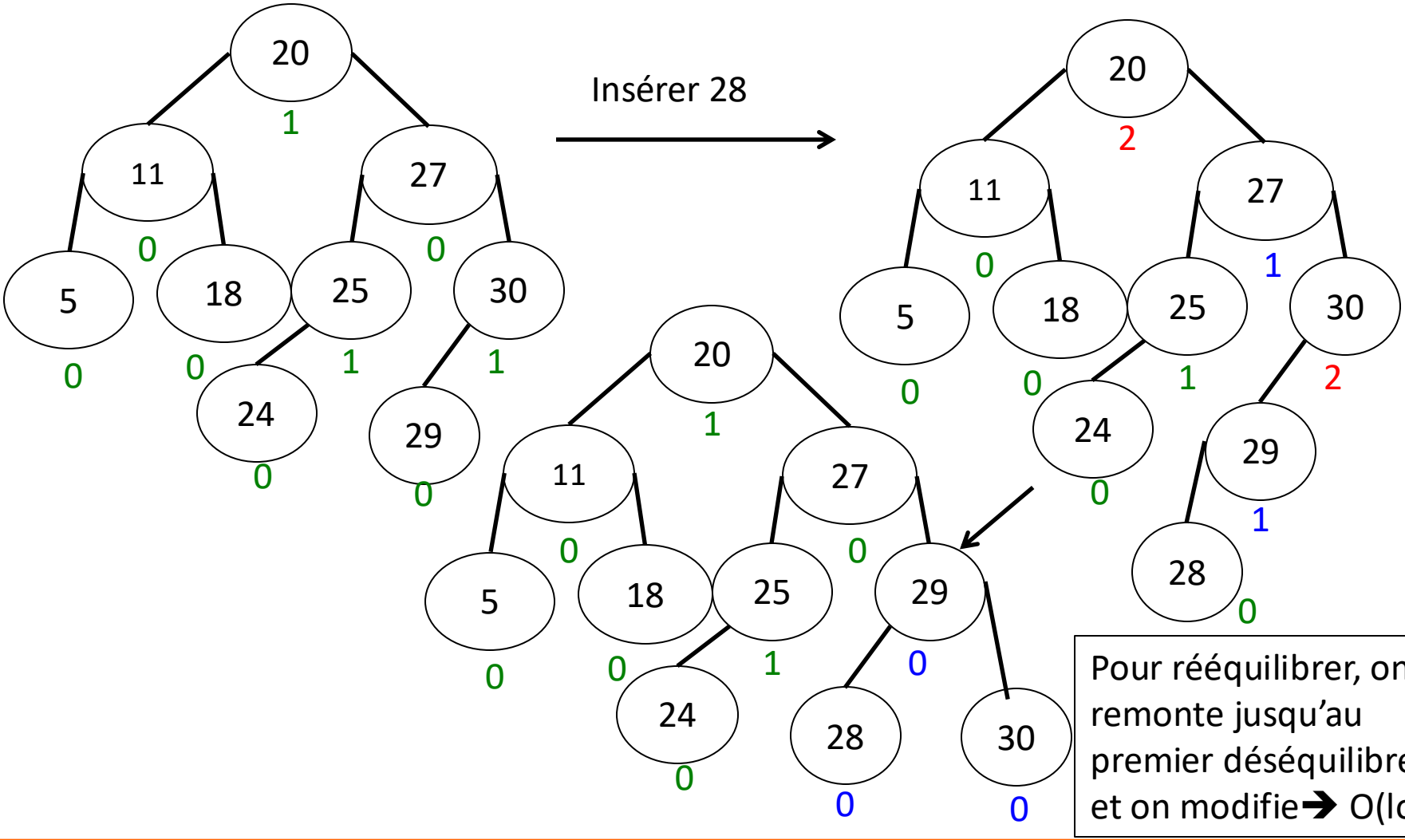
Diff2(x) = |hauteur_a_gauche(x) - hauteur_a_droite(x)| <= 1



Arbre AVL

- ABR tel que pour tout nœud x,

  Diff2(x) = |hauteur_a_gauche(x) - hauteur_a_droite(x)| <= 1



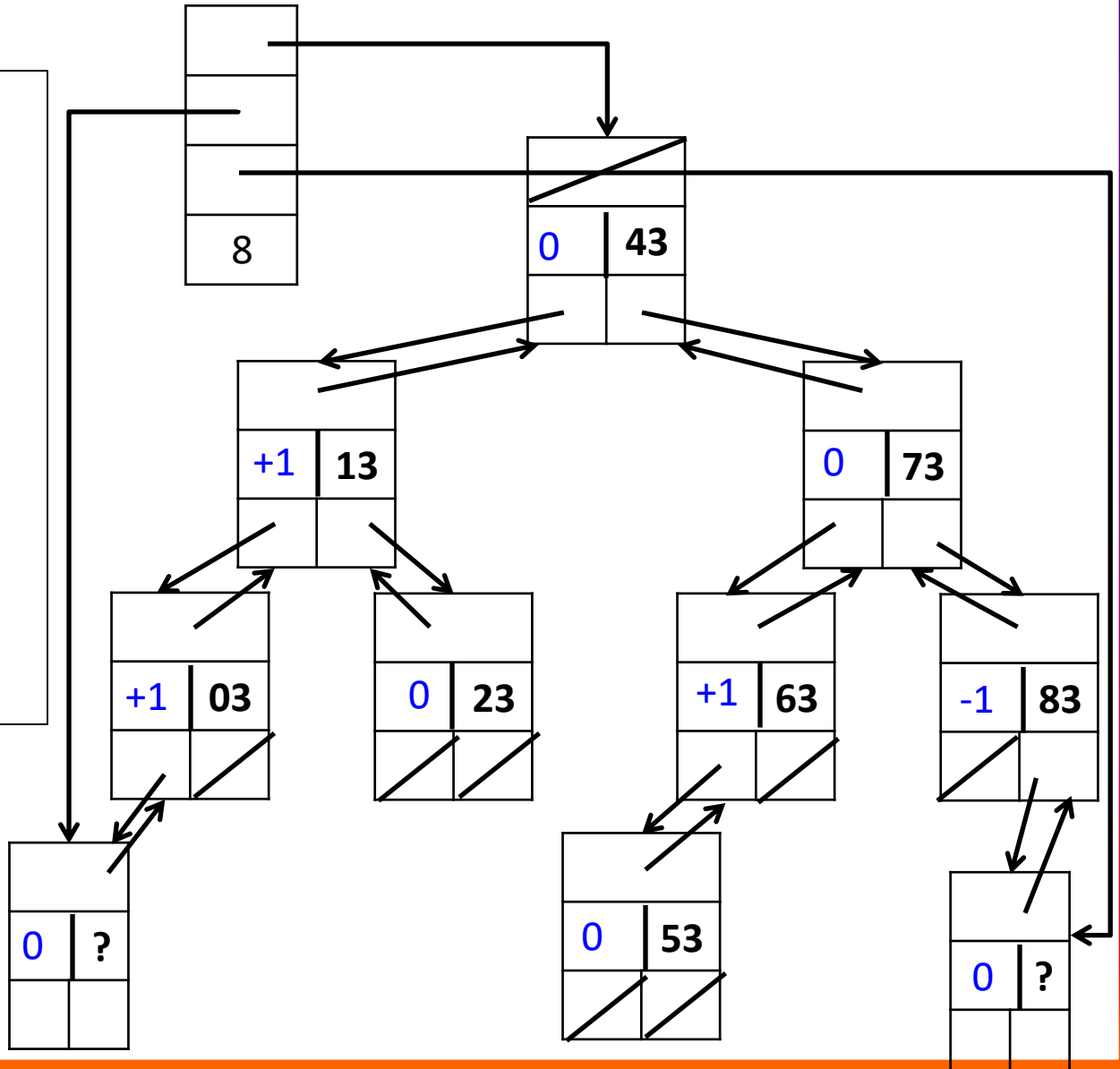
Pour rééquilibrer, on remonte jusqu'au premier déséquilibré et on modifie➔ O(log(n))

Arbre AVL

- Représentation

```
class noeud{
 private:
  type val;
  int indice //h_gauche –
h_droit
  nœud* parent;
  nœud* gauche;
  nœud* droit;
 public:
   noeud(type&);
   noeud(type&, nœud*,
          nœud*, nœud* );}
```

```
class avl{
 private:
  noeud
*racine,debut,fin;
  size_t dim;
 }
```



Arbre AVL : prototype des opérations

❑ **Modificateurs**

swap : avl& → Ø // échange le paramètre avec l'objet appelant
insert: noeud*, type& → noeud* //ajoute un élément à une position (privée)
insert: type& → noeud* //ajoute un élément (privée)
insert: type& → iterator //ajoute un élément
erase: noeud*→ noeud* //retire un élément à une position (privée)
erase: : type& → noeud* //retire un élément (privée)
erase: type& → iterator //retire un élément

❑ **Recherche/ localisation**

lower_bound: type& → noeud* //premier élément >= x (privée)
upper_bound: type& → noeud* //premier élément > x (privée)
search: type& → nœud* (privée)
search: nœud*, type& → nœud* (privée)
search: type& → iterator
previous: noeud*→ noeud* * (privée)
next: noeud*→ noeud* * (privée)

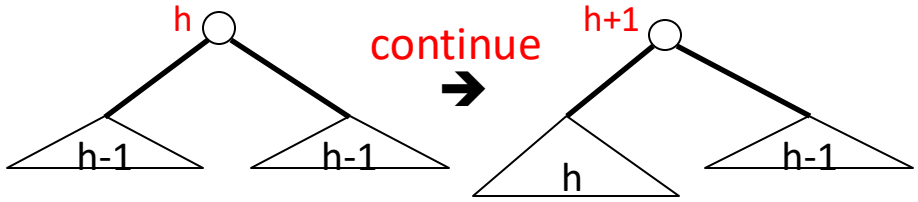
Identiques aux algorithmes pour ABR

AVL : insert

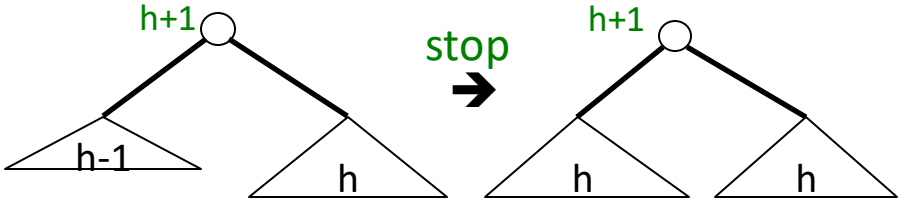
- Après insertion d'une feuille, on remonte vers la racine et pour tout nœud x rencontré:

  Si on remonte de la gauche
  indice(x) = indice(x) + 1
  Sinon
  indice(x) = indice(x) – 1

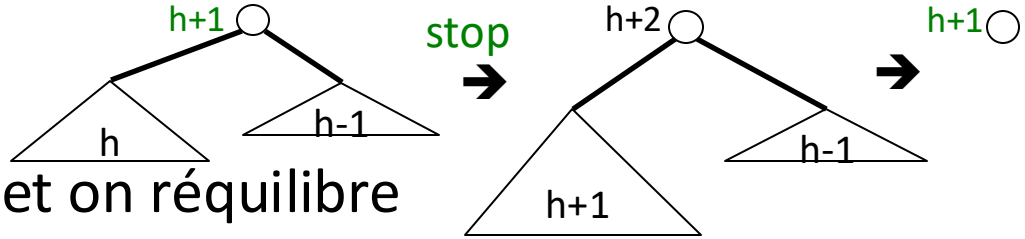
  Si indice(x) == +1 ou -1
  on continue



  Si indice(x) == 0
  on sort de la boucle



  Si indice(x) == -2 ou +2
  on sort de la boucle et on réquilibre



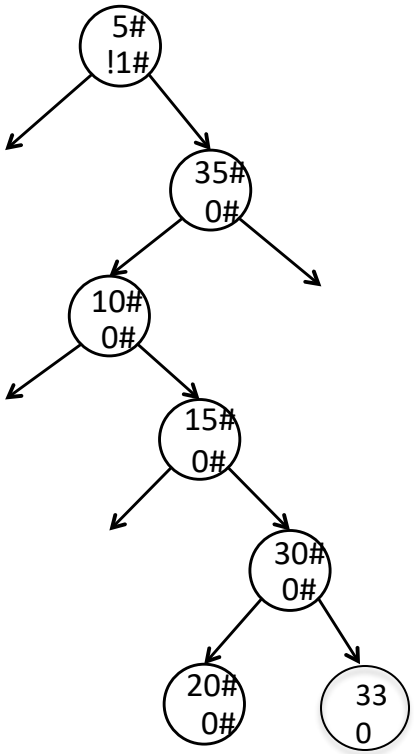
AVL : insert

- noeud* insert (nœud* n, type x)

On insère toujours une feuille

```
        nœud* nouveau = ABR:insert(n,x);
        nœud* no = nouveau;
        nœud* p = no -> parent;
        if(p -> gauche == no) p -> indice += 1; //allonger à gauche
         else p -> indice -= 1; //allonger à droite
        while(p != racine and (p -> indice == -1 or p -> indice == +1
                no = p;
                p = p -> parent ;
                if(p -> gauche == no) p -> indice += 1; //allonger à
gauche
                  else p -> indice -= 1; //allonger à droite
         }
         if(p -> indice == -2 or  p -> indice == +2 )
                //reequilibrer en faisant une rotation gauche si indice
== -2
```

Ajouter 18, puis 34

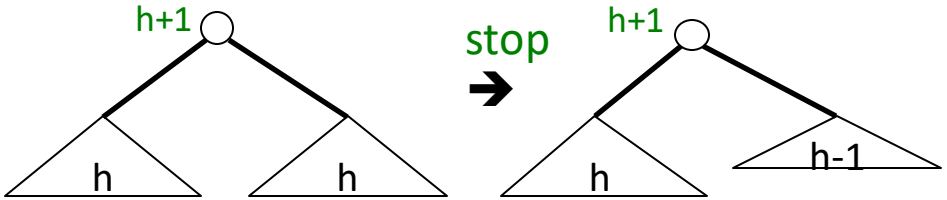


AVL : erase

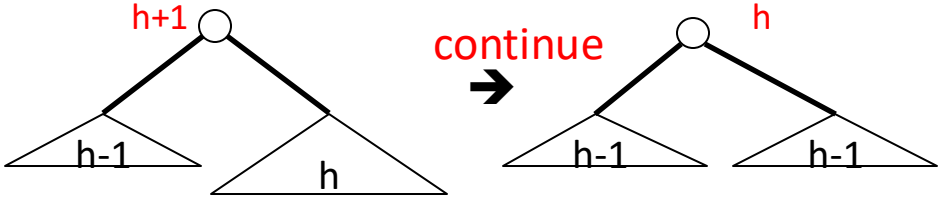
- Après suppression d'une feuille, on remonte vers la racine et pour tout nœud x rencontré:

  Si on remonte de la gauche
  indice(x) = indice(x) - 1
  Sinon
  indice(x) = indice(x) + 1

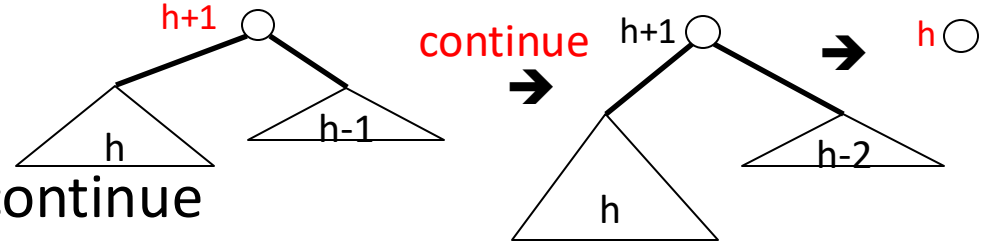
  Si indice(x) == +1 ou -1
  on sort de la boucle



  Si indice(x) == 0
  on continue



  Si indice(x) == -2 ou +2
  on réquilibre et on continue



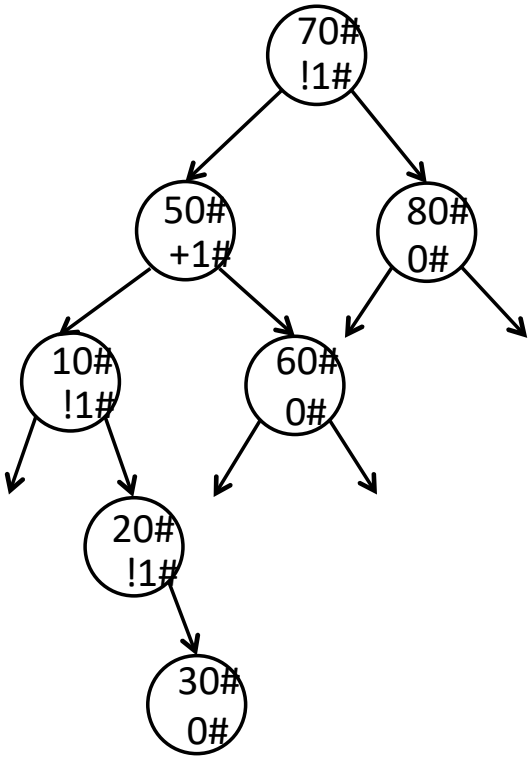
AVL : erase

On supprime toujours une feuille

- noeud* erase (nœud* n)

```
        //nœud* m, a_supprimer: la feuille supprimer
        nœud* p = m -> parent;
        if(p -> gauche == m) p -> indice -= 1; //raccourcir à gauche
       else p -> indice += 1; //raccourcir à droite
        while(p != racine and (p -> indice != -1 or p -> indice != +1)
              if(p -> indice == -2 or  p -> indice == +2 )
                    //reequilibrer en faisant une rotation gauche
                    // ou droite
               no = p;
               p = p -> parent ;
               if(p -> gauche == m) p -> indice -= 1; //raccourcir à
gauche
                  else p -> indice += 1; //raccourcir à droite
        }
```

Enlever 50

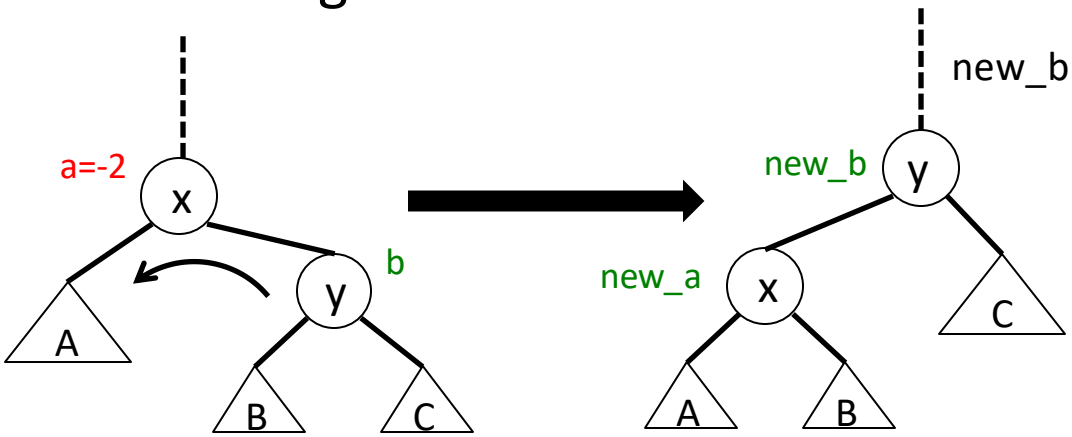


Ré-équilibrage par rotation

- Rotation vers la gauche

new_a = a + max(0,-b) + 1

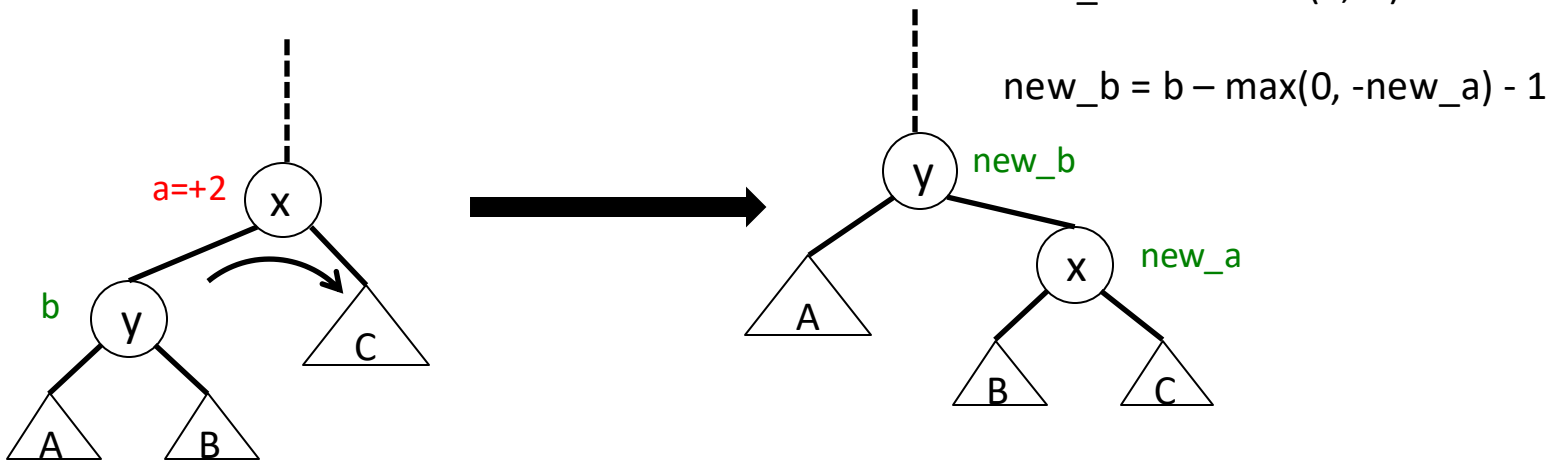
new_b = b + new_a + max(0, -new_a) + 1



- Rotation vers la droite

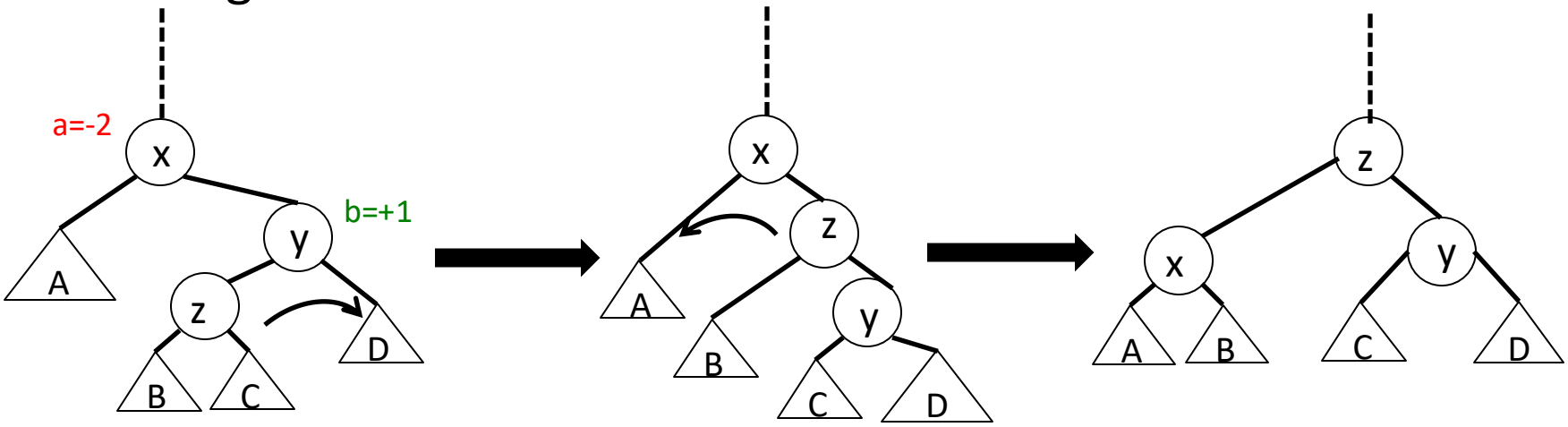
new_a = -b – max(0,-b) – 1 + a

new_b = b – max(0, -new_a) - 1

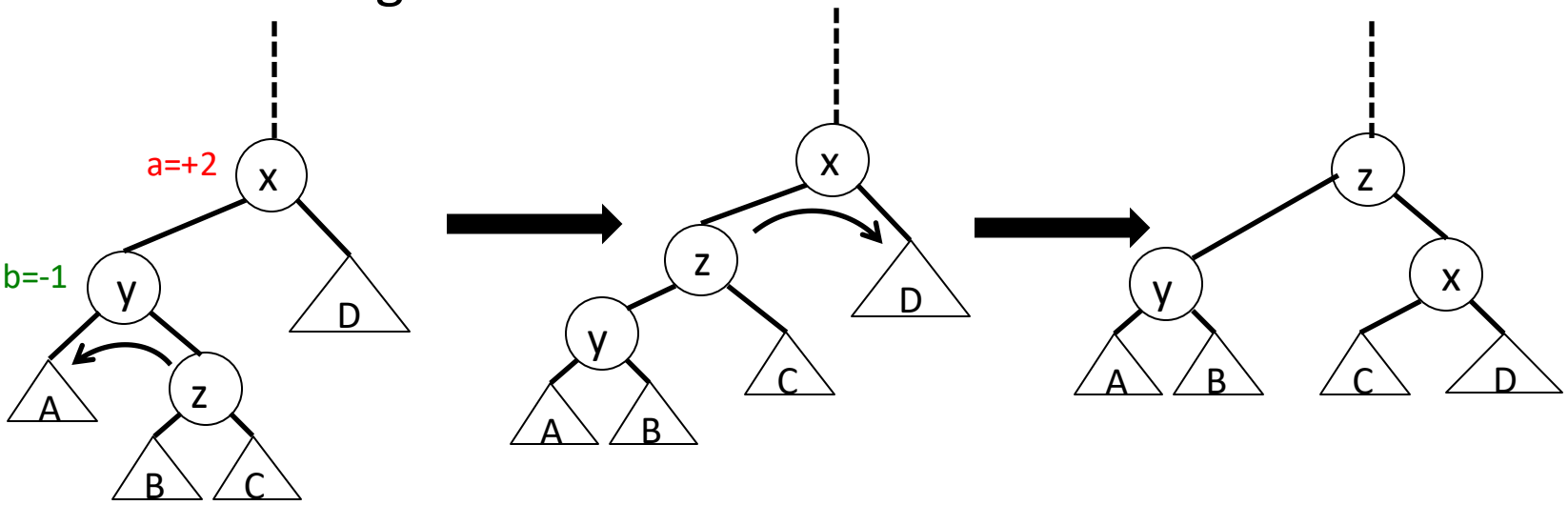


Ré-équilibrage par double rotation
quand le poids se trouve au centre

- Rotation gauche-droite



- Rotation droite-gauche



Itération en O(n) : next(), previous en O(1)

- noeud* previous (nœud* n)

```
        nœud* p;
        if(n->gauche != nullptr){  //cas (1) : a un enfant gauche; on doit descendre
                p = n->gauche;
                while(p->droit != nullptr)
                        p = p->droit;}
        else{    //cas(2) : pas d'enfant gauche; on doit remonter
                prevp = n; p = n->parent;
                while(p != nullptr and p->droit != prevp){
                        prevp = p;
                        p = p->parent;}}
        return p;
```

- O(1) car dans un ABR complet, on (n/2)+1 feuilles et n/2 nœud internes

Map (Fonction)

- Utilise une représentation sous forme d'AVL
- Élément = paire (type clé constant, type valeur modifiables)
- Ordre des éléments = ordre des clés