IN103
Résolution de problèmes algorithmiques
une approche fondée sur les structures de données

Alexandre Chapoutot

Année académique 2023-2024

Plan du cours

Une nouvelle organisation des données

Jusqu'à présent, des structures de données linéaires ont été étudiées :

- tableau : taille fixe, accès directe
- liste chaînée : taille variable, accès indirect
- pile : comportement LIFO
- file : comportement FIFO
- ensemble : unicité des éléments

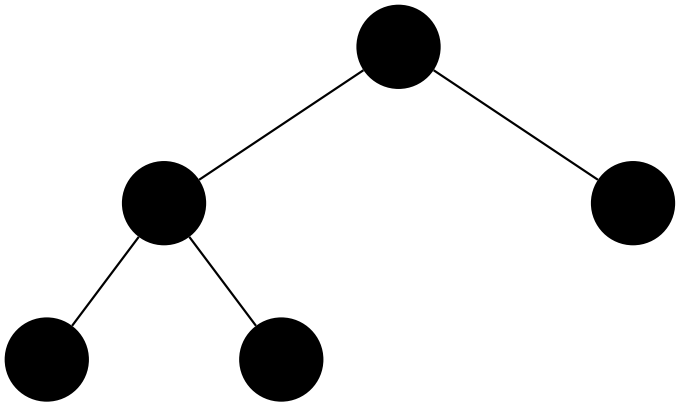
L'organisation des données dans ces structures est sans relation mathématique particulière.

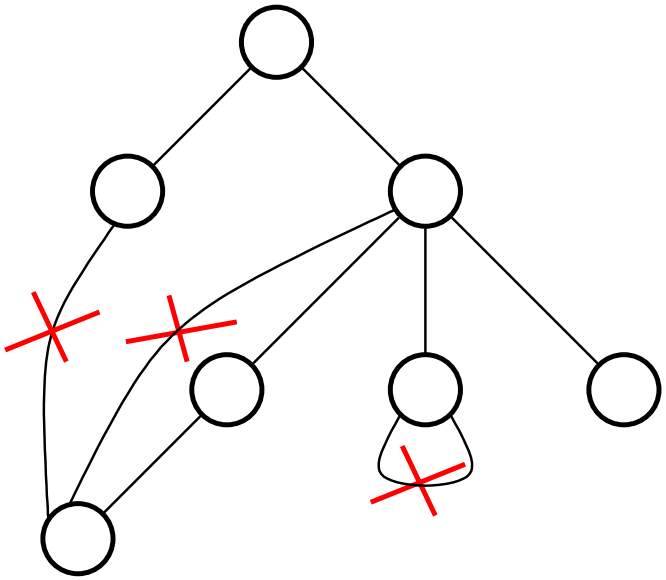
Conséquence, la recherche dans une liste chaînée a une complexité $\mathcal{O}(n)$.

Les arbres sont une structure de données associées à une relation d'ordre.

Arbres

Structure arborescente





- Structure arborescente avec une relation parent/enfant
- Des **nœuds**, des **arcs**
- mais pas de cycle, pas de raccourci (saut de générations)

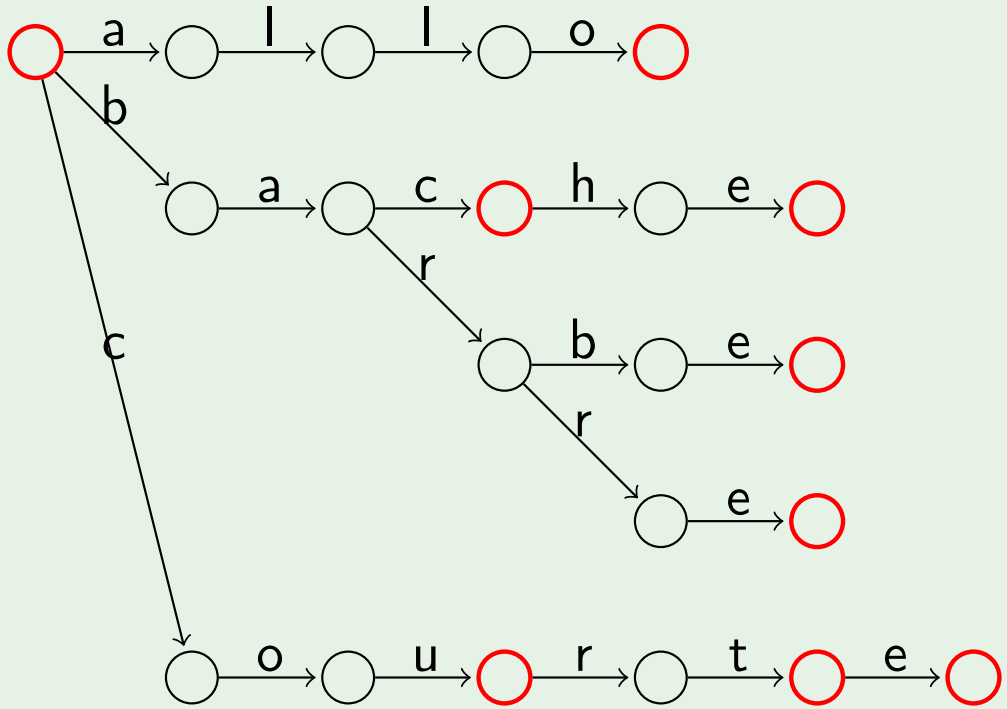
Exemple : dictionnaire

Dictionnaire représenté par un **arbre préfixe** (en anglais *trie*)

- Permet le partage de préfixes communs,
- Préfixes partagés d'où nécessité de marquer les fins de mots

Exemple

Les mots { allo, bac, bache, barbe, barre, cou, court, courte } sont représentés par



Exemple : système de fichiers (simple)

```
/
|-- bin
|   |-- alsaunmute
|   |-- arch
|   |-- awk
|   `-- zcat
|-- boot
|   |-- config-2.6.40.6-0.fc15.i686
|   |-- grub
|   |   |-- device.map
|   |   |-- e2fs_stage1_5
|   |   |-- ufs2_stage1_5
|   |   |-- vstafs_stage1_5
|   |   `-- xfs_stage1_5
|   |-- initramfs-2.6.40.6-0.fc15.i686.img
|   `-- vmlinuz-2.6.43.2-6.fc15.i686
|-- cgroup
|-- dev
|   |-- autofs
```

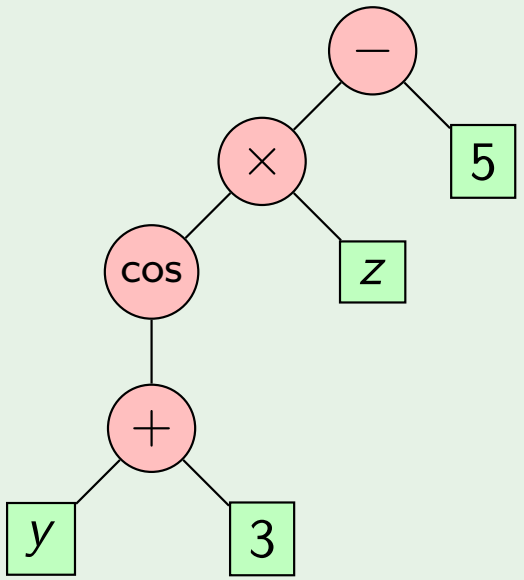
- Racine : /
- Nœuds internes : répertoires.
- Feuilles : fichiers.
- (On ne considère pas les « raccourcis » – ou « liens »).

Exemple : arbre de syntaxe

Expressions arithmétiques :

- Constantes.
- Opérateurs binaires entre **expressions arithmétiques**.
- Appel de fonction avec des **expressions arithmétiques** en arguments.
- Variables.

Exemple : $\cos(y + 3) \times z - 5$

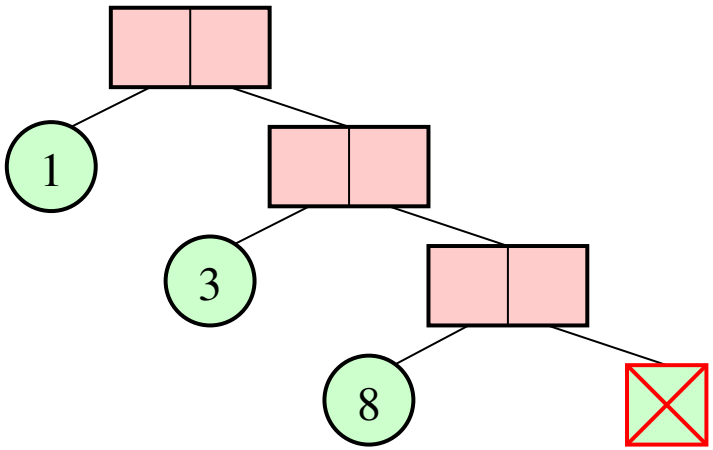


Permet de calculer

- évaluation
- dérivation
- simplification
- compilation, ...

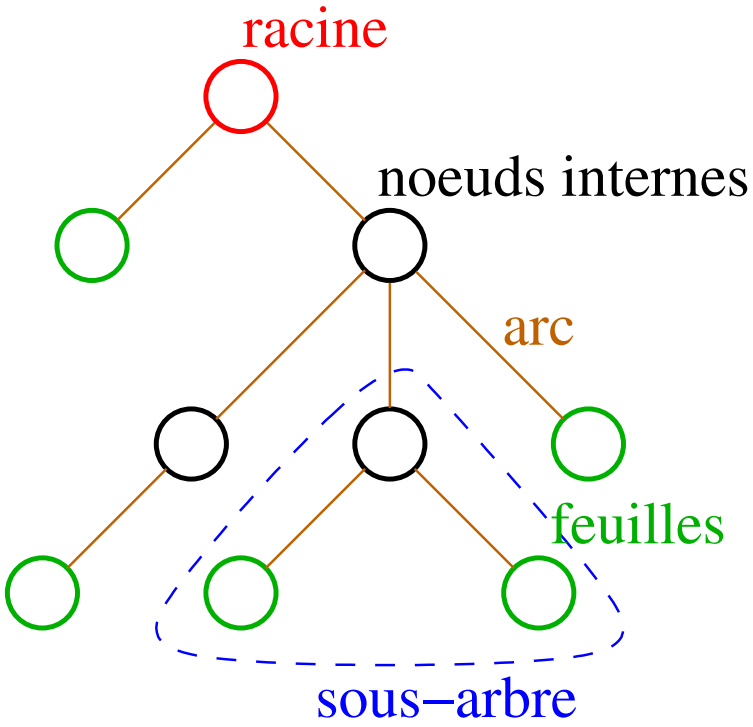
Exemple : listes simplement chaînées

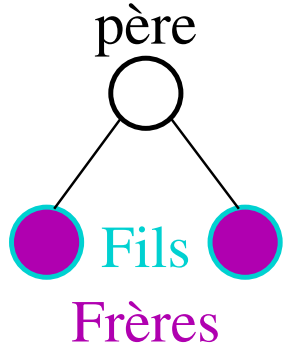
- Une liste : [1, 3, 8]
- Rappel : type inductif
  `list` ::=
  `Nil` | `List of int` $*$ `list`
- Longueur de la liste = hauteur de l'arbre



Arbres : vocabulaire - 1

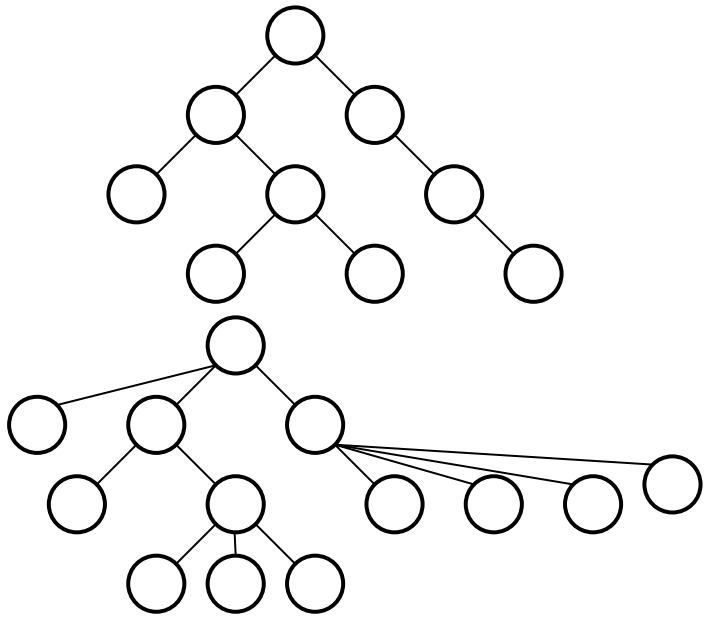
- Nœud (*node*) :
  - **Racine** (*root*)
  - **Nœud interne**
  - **Feuille** (*leaf*)
- **Arc** (*edge*)
- **Sous-arbre** (*subtree*)
- Parenté :
  - **Père** (*parent*)
  - **Fils** (*child*)
  - **Frère** (*sibling*)





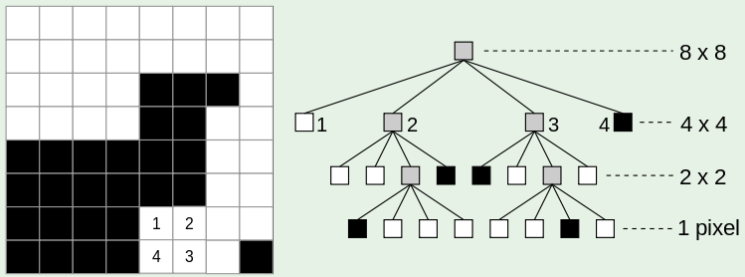
Arbres : vocabulaire - 2

- **Degré** d'un nœud : nombre de fils
- **Arité** d'un arbre : nombre **max** de fils
  - Pas forcément homogène sur tous les nœuds de l'arbre
- **Arbre binaire** : arité 2
- **Quadtree** : arité 4
- **Octree** : arité 8
- Autrement arbre n-aire



Compression d'images avec quadtree [a]

a. Crédit `https://fr.wikipedia.org/wiki/Quadtree`



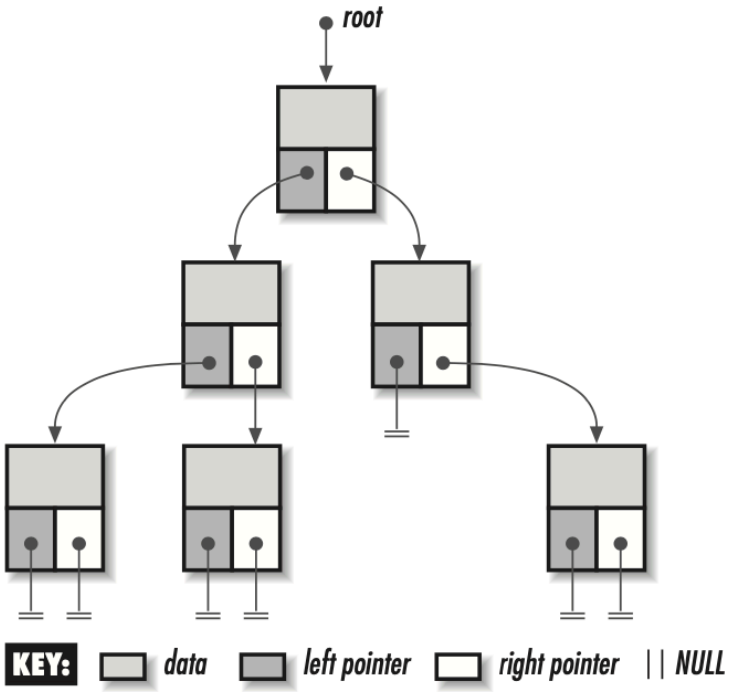
Arbres binaires

Arbres binaires : implémentation

Type inductif utilisé dans les langages fonctionnels (Ocaml, Haskel, ...)

- `tree ::= Empty | Node of tree * elem * tree`

Comme pour les listes, les pointeurs sont utilisés pour faire un chaînage [1]



Définition d'un nœud

```
typedef struct integer_bitreenode_ {
  int data;
  struct integer_bitreenode_ *left;
  struct integer_bitreenode_ *right;
} integer_bitreenode_t;
```

Définition d'un arbre

```
typedef struct integer_bitree_ {
  int size;
  integer_bitreenode_t *root;
} integer_bitree_t;
```

1. Crédit image : O'Reilly 'Mastering Algorithms with C'

API des arbres binaires – 1

API Création/Destruction

```
void integer_bitree_init(integer_bitree_t*);

void integer_bitree_destroy(integer_bitree_t*);
```

API Accesseurs

```
int integer_bitree_size(integer_bitree_t*);

integer_bitreenode_t* integer_bitree_root(integer_bitree_t*);

int integer_bitree_data(integer_bitreenode_t*);

integer_bitreenode_t* integer_bitree_left(integer_bitreenode_t*);

integer_bitreenode_t* integer_bitree_right(integer_bitreenode_t*);
```

API des arbres binaires – 2

API Prédicats

```
bool integer_bitree_is_eob(integer_bitreenode_t*);

bool integer_bitree_is_leaf(integer_bitreenode_t*);
```

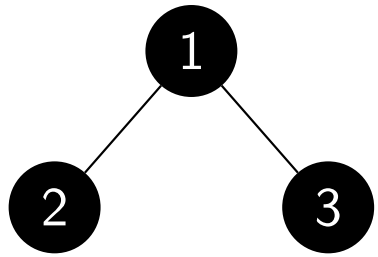
API Insertion/Suppression

```
int integer_bitree_ins_left(integer_bitree_t*,
                            integer_bitreenode_t*, int);

int integer_bitree_ins_right(integer_bitree_t*,
                             integer_bitreenode_t *, int);

void integer_bitree_rem_left(integer_bitree_t*,
                             integer_bitreenode_t*);

void integer_bitree_rem_right(integer_bitree_t*,
                              integer_bitreenode_t*);
```

Exemple : un petit arbre binaire

Code source

```
int main () {
  int code;
  integer_bitreenode_t* root;
  integer_bitree_t tree;
  integer_bitree_init (&tree);

  code = integer_bitree_ins_left (&tree, NULL, 1);
  printf ("Root takes value 1\n");

  root = integer_bitree_root (&tree);
  code = integer_bitree_ins_left (&tree, root, 2);
  printf ("Root->left takes value 2\n");
  code = integer_bitree_ins_right (&tree, root, 3);
  printf ("Root->right takes value 3\n");

  integer_bitree_destroy (&tree);

  return EXIT_SUCCESS;
}
```



Complexité des opérations sur les arbres binaires

Fonction	**Complexité**	**Commentaire**
Initialisation	$\mathcal{O}(1)$	Mise de la mémoire à zéro
Destruction	$\mathcal{O}(n)$	Parcourir tous les nœuds pour supprimer les éléments
Accesseurs	$\mathcal{O}(1)$	
Insertion	$\mathcal{O}(1)$	**Hyp :** on sait où faire l'insertion
Suppression	$\mathcal{O}(n)$	**Hyp :** on sait où faire la suppression mais il faut éliminer tous les nœuds du sous-arbre.
Recherche	$\mathcal{O}(n)$	les valeurs sont stockées sans ordre particulier, il faut donc tout parcourir.

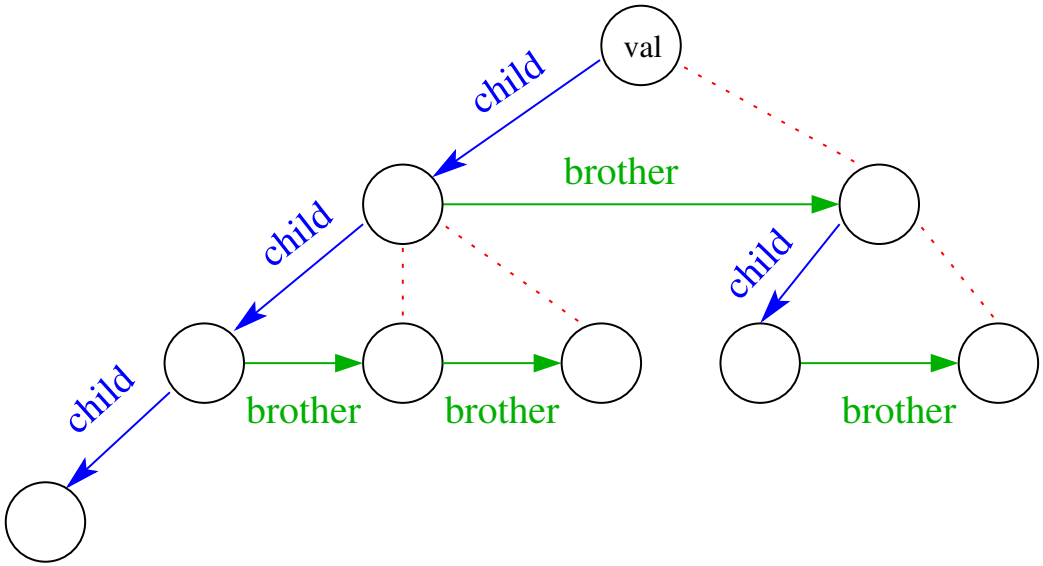
Remarque

La structure d'arbre binaire permet de construire un arbre facilement mais n'est pas pensée pour chercher des éléments, cf plus loin.

Arbre n-aire

Cas général, un nombre quelconque de fils, un nœud est

- une donnée.
- une **liste chaînée** de tous les fils (frères).
- le parent a un **pointeur** sur son premier fils.



Code source

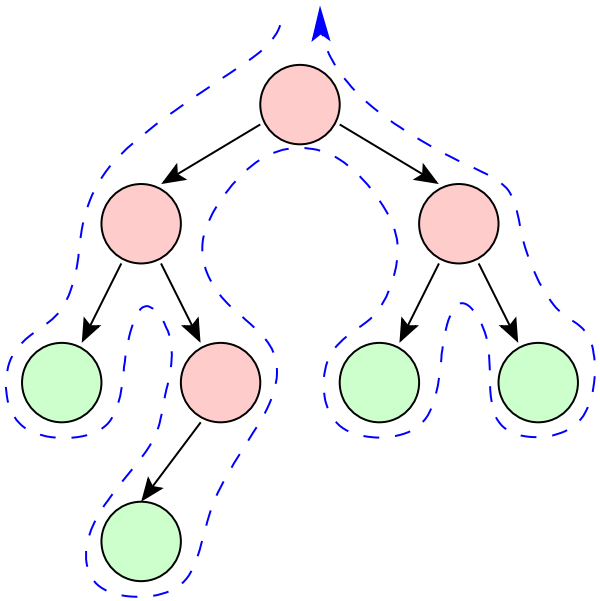
```
struct node {
  int data;
  struct node* child;
  struct node* brothers;
};
```

Remarque : on peut aussi utiliser un tableau pour stocker les pointeurs des enfants mais cela nécessite de fixer l'arité de l'arbre au départ.

Parcours d'arbres

Arbres binaires : parcours en profondeur

- En Anglais : *Depth First Search* (DFS).
- Descente **au plus profond** en commençant par le fils gauche (ou droit).
- Parcours récursif :
  - À chaque nœud, on applique la descente sur le fils gauche.
  - Une fois le sous-arbre gauche exploré, on explore le fils droit.
  - Une fois le sous-arbre droit exploré, on remonte et applique le parcours.
  - Fin : fin de l'exploration du sous-arbre droit.



Function

```
dfs(integer_bitreenode_t* n) :
    if n != NULL then
        dfs(n->left);
        dfs(n->right);
    end
```

Remarque : pseudo-code du parcours sans traitement !

Parcours en profondeur

Le traitement de la valeur du nœud courant peut se faire à des moments différents durant le parcours

- **Ordre préfixe** : traitement, puis visite gauche, visite droit
- **Ordre infixe** : visite gauche, traitement, visite, droit
- **Ordre postfixe** : visite gauche, visite droit, traitement

```
Function dfs(integer_bitreenode_t* n) :
    if n != NULL then
        do_something(n->data)
        dfs(n->left)

        dfs(n->right)

    end
```

Parcours en profondeur

Le traitement de la valeur du nœud courant peut se faire à des moments différents durant le parcours

- **Ordre préfixe** : traitement, puis visite gauche, visite droit
- **Ordre infixe** : visite gauche, traitement, visite, droit
- **Ordre postfixe** : visite gauche, visite droit, traitement

Function `dfs(integer_bitreenode_t* n)` **:**

```
    if n != NULL then
        dfs(n->left)
        do_something(n->data)
        dfs(n->right)
    end
```

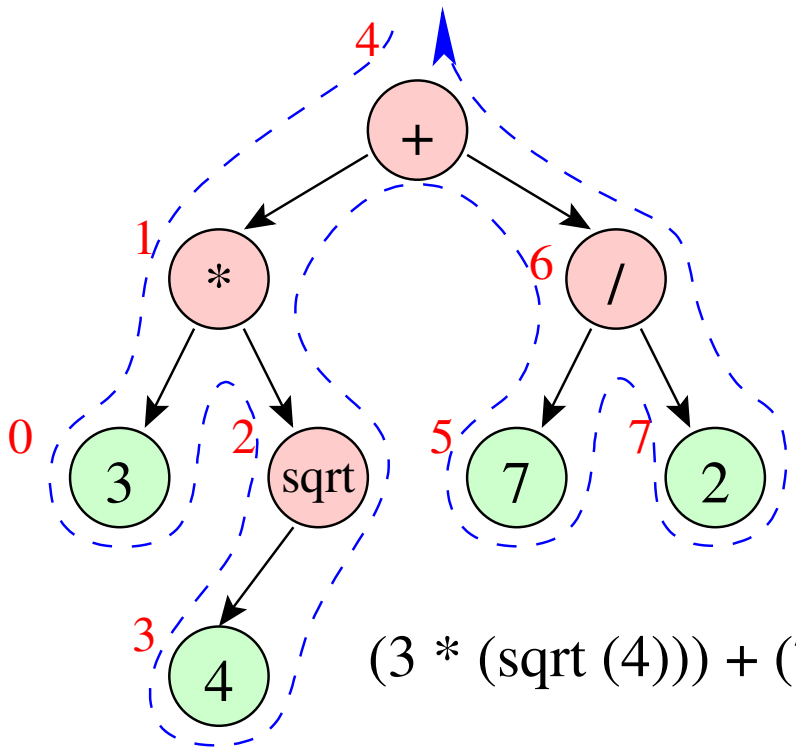
Parcours en profondeur

Le traitement de la valeur du nœud courant peut se faire à des moments différents durant le parcours

- **Ordre préfixe** : traitement, puis visite gauche, visite droit
- **Ordre infixe** : visite gauche, traitement, visite, droit
- **Ordre postfixe** : visite gauche, visite droit, traitement

```
Function dfs(integer_bitreenode_t* n) :
    if n != NULL then
        dfs(n->left)

        dfs(n->right)
        do_something(n->data)
    end
```

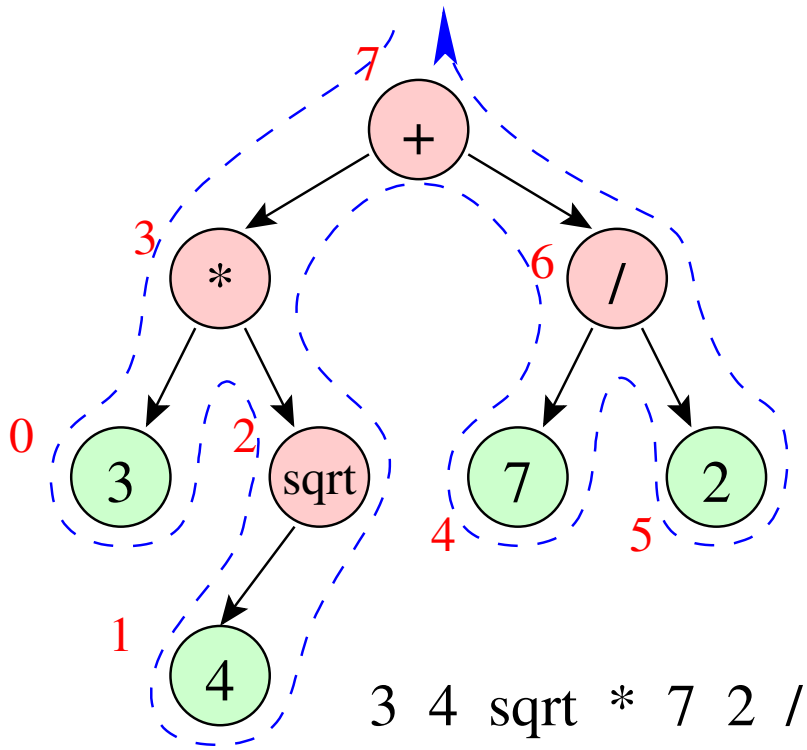
Exemple : parcours infixe



(3 * (sqrt (4))) + (7 / 2) (= 9.5)

Remarque en rouge l'ordre de traitement des nœuds

Exemple : parcours postfixe



Remarque en rouge l'ordre de traitement des nœuds

Implémentation des parcours d'arbres dans `libin103`

Trois fonctions sont accessibles dans le fichier `bitreealg.h`

Prototypes

```
int integer_preorder(integer_bitreenode_t*, integer_list_t*);

int integer_inorder(integer_bitreenode_t*, integer_list_t*);

int integer_postorder(integer_bitreenode_t*, integer_list_t*);
```

- Le premier argument est le nœud de l'arbre à partir duquel il faut effectuer le parcours.
- Le second argument est une liste des valeurs des nœuds visités dans l'ordre choisi.

Remarque

Pour effectuer un traitement sur les nœuds, il faut coder une version spécialisée du parcours voulu.

Complexité des fonctions de parcours

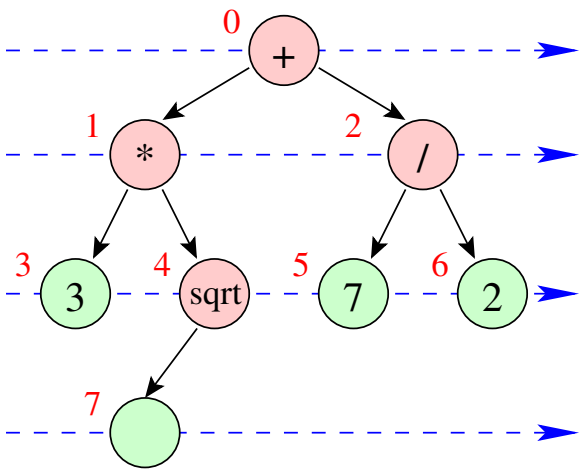
Fonction	**Complexité**	**Commentaire**
Préfixe	$\mathcal{O}(n)$	Il faut parcourir tous les nœuds les éléments
Infixe	$\mathcal{O}(n)$	Il faut parcourir tous les nœuds les éléments
Postfixe	$\mathcal{O}(n)$	Il faut parcourir tous les nœuds les éléments

Remarque

Les parcours sont exhaustifs d'où le coût linéaire.

Arbres binaires : parcours en largeur

- En Anglais : *Breadth First Search* (BFS).
- Parcours « par niveaux » (croissants), visite tous les nœuds de « même profondeur ».
- Utilisation d'une structure de données **file** :
  On extrait un nœud, on le traite, on insère tous ses fils.



Remarque en rouge l'ordre de traitement des nœuds

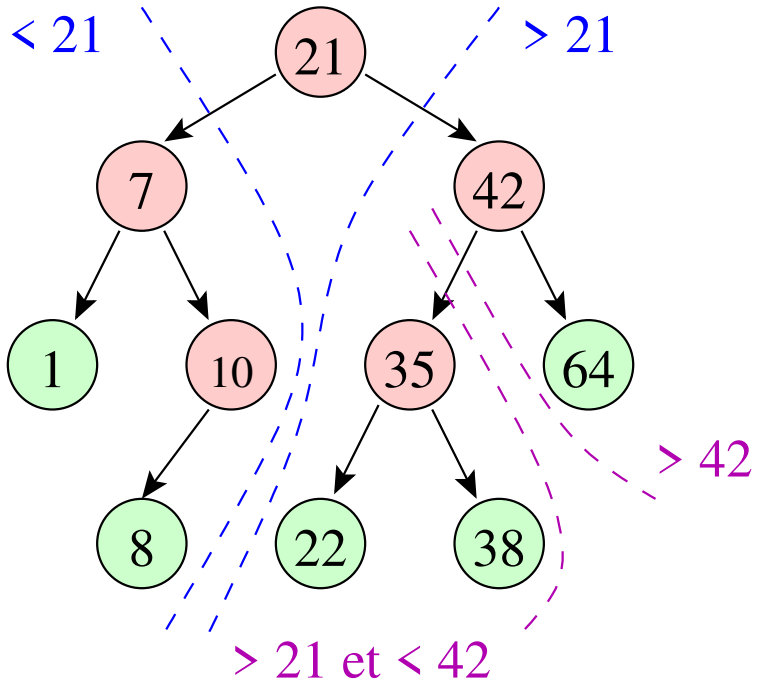
```
Function bfs(integer_bitreenode_t* n) :
    generic_queue_enqueue(q, n);
    while generic_queue_size(q) > 0 do
        generic_queue_dequeue (q, tmp);
        do_something(tmp->data);
        if tmp->left != NULL then
            generic_queue_enqueue(q, n->left);
        end
        if tmp->right != NULL then
            generic_queue_enqueue(q, n->right);
        end
    end
```

Arbres binaires de recherche

Arbres Binaires de Recherche (ABR)

Structure d'arbre binaire (*Binary Search Tree* (BST)) telle que :

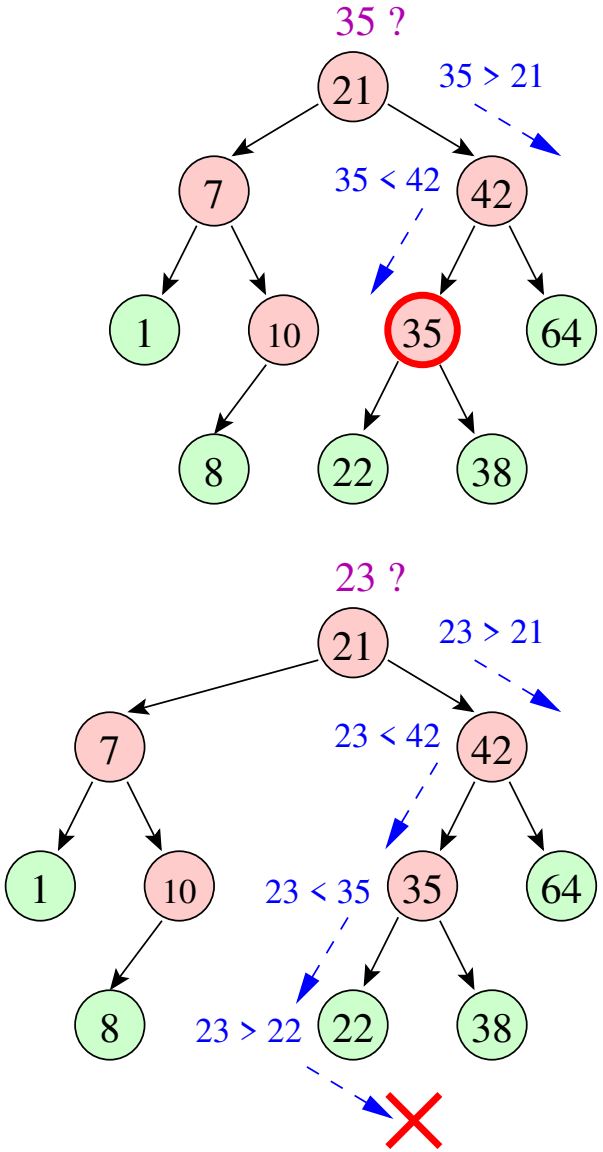
- Chaque nœud contient une **clef**.
- La clef d'un nœud est **supérieure** à celle de son fils **gauche**.
  Inductivement supérieur à celles du sous-arbre gauche.
- La clef d'un nœud est **inférieure** à celle de son fils **droit**.
  Inductivement inférieur à celles du sous-arbre droit.



Intérêts des ABR

- Stockage en $\mathcal{O}(n)$
- Recherche en $\mathcal{O}(h)$ (h hauteur de l'arbre).
- Insertion en $\mathcal{O}(h)$
- Suppression en $\mathcal{O}(h)$
- En moyenne $h = \mathcal{O}(\log(n))$ mais dans le pire cas $h = n$:
  $\rightarrow$ intérêt d'avoir des arbres « équilibrés » (cf plus loin)

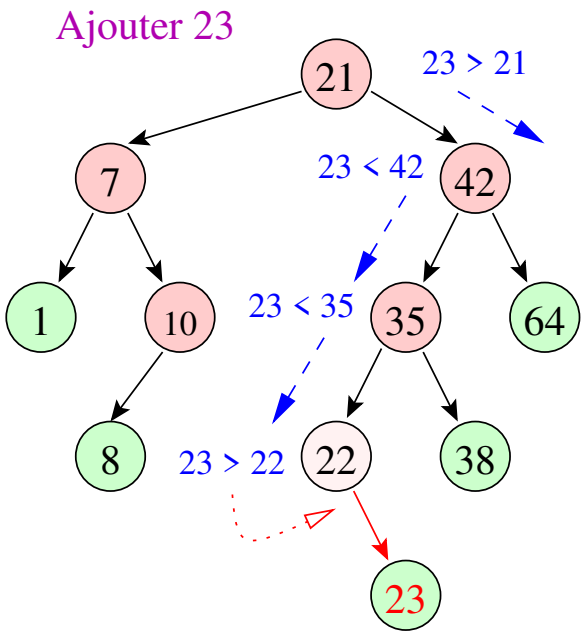
ABR : opération de recherche



Pseudo-code

```
Function lookfor(abr_t* n, int v) :
    if n == NULL then
        return false;
    end
    if v == n->data then
        return true;
    end
    if v < n->data then
        return lookfor(n->left, v);
    end
    if v > n->data then
        return lookfor(n->right, v);
    end
```

ABR : opération d'insertion



On cherche où insérer puis on créer un nouveau nœud.

Pseudo-code

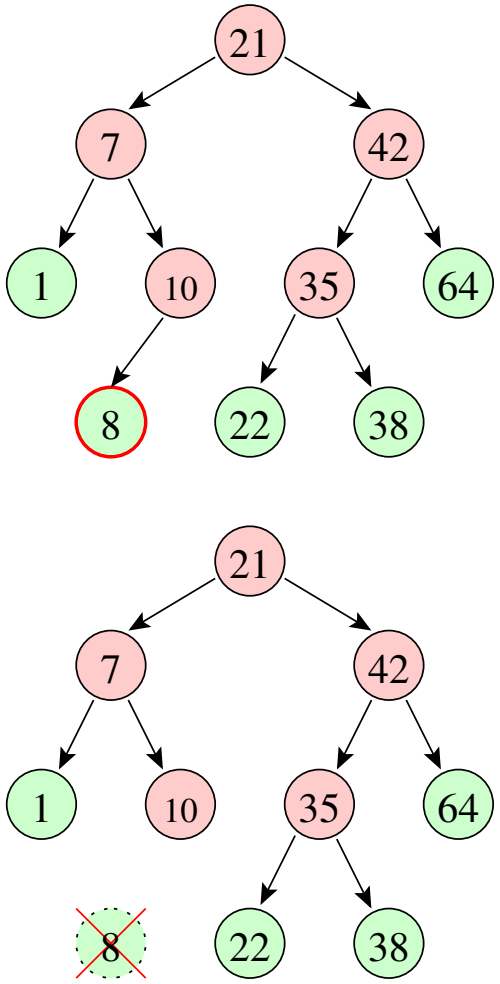
```
Function insert(abr_t* n, int v) :
    if v < n->data then
        if n->left != NULL then
            insert(n->left, v);
        else
            n->left = node(v);
        end
    else if v > n->data then
        if n->right != NULL then
            insert(n->right, v);
        else
            n->right = node(v);
        end
    else
        /* Do nothing                    */
    end
```

ABR : opération de suppression

On commence par rechercher le nœud à supprimer dans l'arbre, 3 cas dont 2 simples à gérer

- Suppression d'une feuille ($\rightarrow$ facile).
- Suppression d'un nœud avec 1 seul fils ($\rightarrow$ facile).
- Suppression d'un nœud avec 2 fils ($\rightarrow$ plus compliqué).

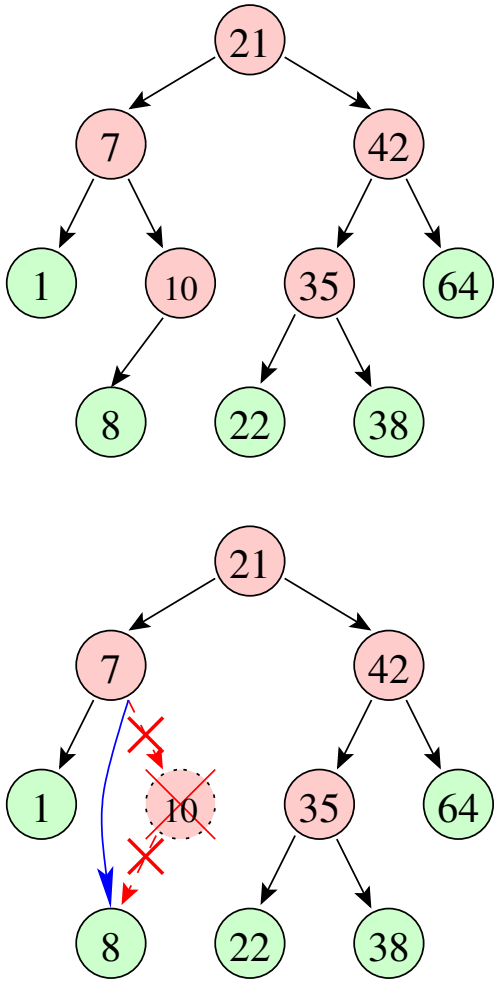
ABR : opération de suppression (feuille)



Suppression du nœud 8 . . .

- On cherche le nœud dans l'arbre
- Mettre le pointeur gauche du nœud 10 à `NULL`.
- Libérer la mémoire occupée par le nœud 8.

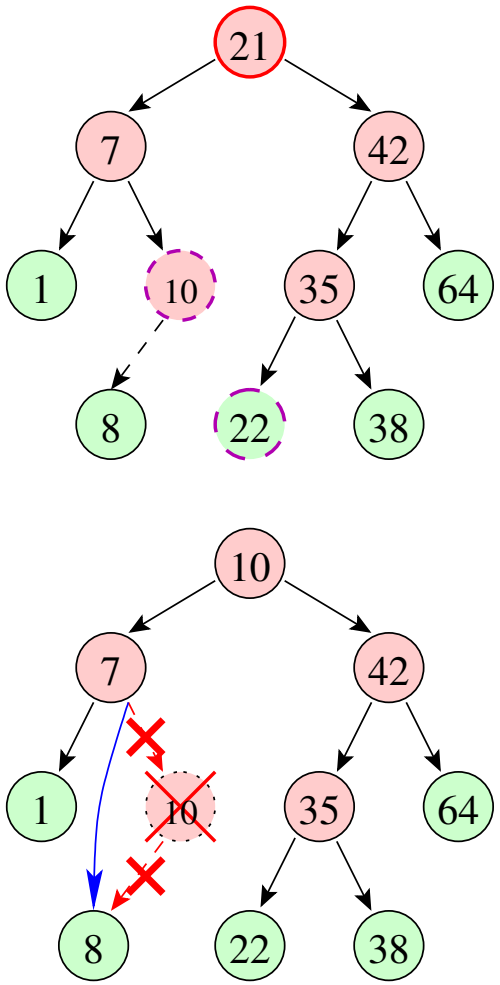
ABR : opération de suppression (1 fils)



Suppression du nœud 10 :

- On cherche le nœud dans l'arbre
- Faire pointer son père (7) vers son fils (8).
- Libérer la mémoire occupée par le nœud 10.

ABR : opération de suppression (2 fils)



Suppression du nœud 21 :

- On cherche le nœud dans l'arbre
- Remplacement par le minimum du fils droit ou le max du fils gauche.

  Successeur ou prédécesseur le plus proche.
- Choisir un candidat.
- On recopie sa clef et ses données pour remplacer le nœud 21.
- Ensuite, supprimer le nœud 10

  10 étant « le max », il n'a pas de fils droit,

  $\Rightarrow$ tombe dans un des 2 cas précédents.

Arbres binaires de recherche équilibrés

Importance des arbres équilibrés

Rappel

La complexité des opérations sur les ABR est majoritairement dans la classe $\mathcal{O}(h)$ avec h la hauteur de l'arbre.
Et dans le pire cas, $h = n$, avec n le nombre d'éléments.

Il existe plusieurs types d'ABR qui incorporent des opérations d'équilibrage

- les AVL (du nom des auteurs Adel'son, Vel'skii and Landis)
- les arbres rouges-noirs

La bibliothèque `libin103` met en œuvre les arbres AVL.

Arbres AVL

Les AVL sont des ABR dont

- la hauteur du sous-arbre gauche et la hauteur du sous-arbre droit ne diffère pas plus de 1.
- Nœuds comporte un indicateur d'équilibre (champs `factor`).

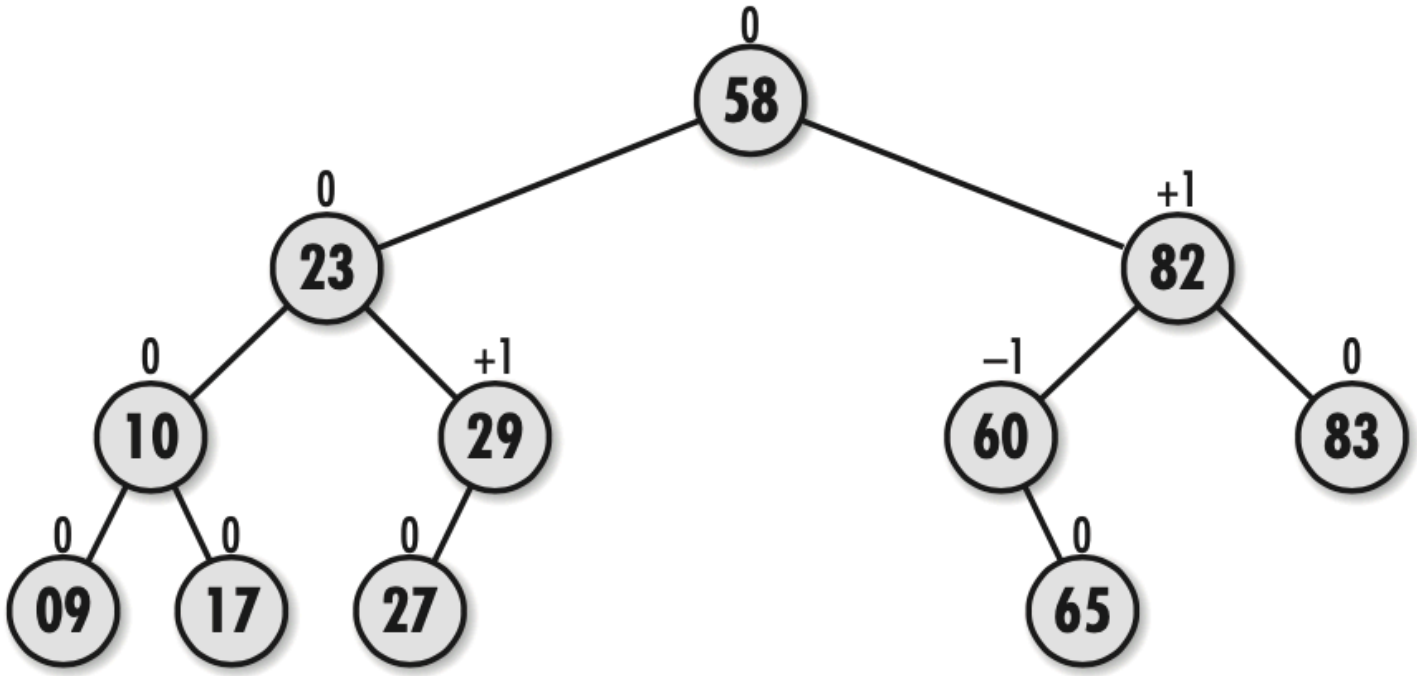
Définition d'un nœud

```
typedef struct integer_avlnode_ {
  int data;
  int hidden;
  int factor;
} integer_avlnode_t;
```

Définition d'un arbre

```
typedef generic_bitree_t integer_bistree_t;
```

Exemple : AVL avec indicateur d'équilibre



Implémentation des indicateur d'équilibre

```
#define          integer_AVL_LFT_HEAVY          1
#define          integer_AVL_BALANCED           0
#define          integer_AVL_RGT_HEAVY         -1
```

API des AVL – 1

API Création/Destruction

```
void integer_bistree_init(integer_bistree_t*);

void integer_bistree_destroy(integer_bistree_t*);
```

API Accesseurs / Prédicats

```
bool integer_bistree_lookup(integer_bistree_t*, int);

int integer_bistree_size(integer_bistree_t*);
```

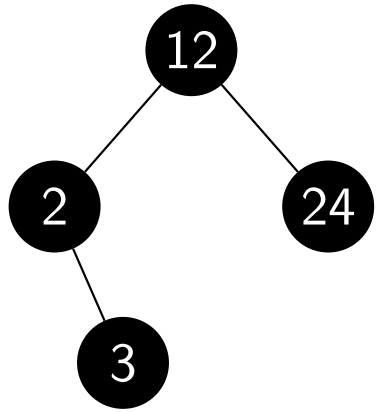
API Insertion/Suppression

```
int integer_bistree_insert(integer_bistree_t*, int);

int integer_bistree_remove(integer_bistree_t*, int);
```

Exemple : un petit AVL

Code source

```
int main (){

  int size = 4;
  int tab[] = {12, 2, 24, 3};

  integer_bistree_t avl;
  integer_bistree_init (&avl);

  for (int i = 0; i < size; i++) {
    integer_bistree_insert (&avl, tab[i]);
  }

  integer_bistree_destroy(&avl);

  return EXIT_SUCCESS;
}
```



Complexité des opérations sur les arbres binaires

Fonction	**Complexité**	**Commentaire**
Initialisation	$\mathcal{O}(1)$	Mise de la mémoire à zéro
Destruction	$\mathcal{O}(n)$	Parcourir tous les nœuds pour supprimer les éléments
Taille	$\mathcal{O}(1)$	
Insertion	$\mathcal{O}(\log(n))$	Il faut chercher où insérer
Suppression	$\mathcal{O}(\log(n))$	Il faut chercher où supprimer
Recherche	$\mathcal{O}(\log(n))$	L'équilibrage garantit une hauteur minimale

Remarque

La structure d'AVL est pensée pour stocker et faire des opérations de recherche

Remarque

L'implémentation des AVL dans la `libin103` utilise la technique nommée `lazy removal`. Elle ne supprime pas les données mais les cache (cf champs `hidden`).

Rotation

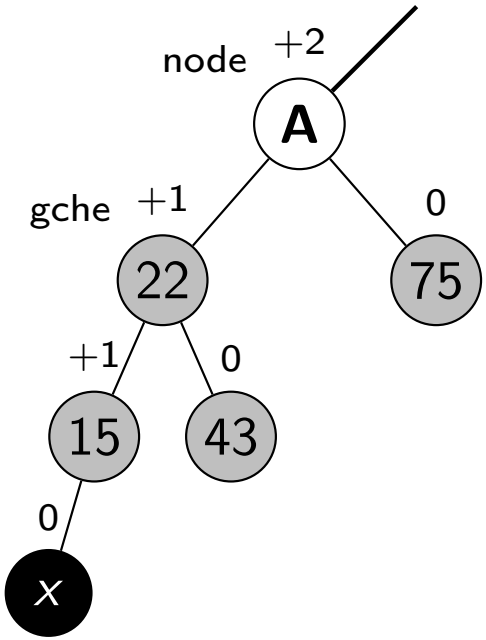
Pour maintenir un arbre équilibré les AVL effectuent des modifications locales de l'arbre après insertion : **Rotation**

Plusieurs cas sont envisagés :

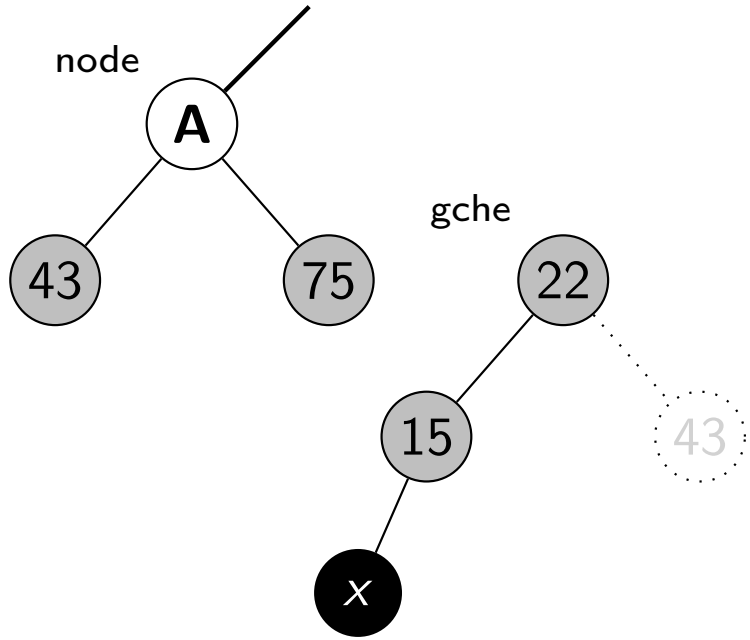
- **Rotation LL** (left-left)
  insertion d'un nouvel élément comme fils gauche du fils gauche
  $\Rightarrow$ Rotation à droite
- **Rotation LR** (left-right)
  insertion d'un nouvel élément comme fils droit du fils gauche
  $\Rightarrow$rotation à droite
- **Rotation RR** (right-right)
  insertion d'un nouvel élément comme fils droit du fils droit
  $\Rightarrow$rotation à gauche
- **Roration RL** (right-left)
  insertion d'un nouvel élément comme fils gauche du fils droit
  $\Rightarrow$rotation à gauche

Rotation : left-left (LL)

Étape 1 : insertion de x

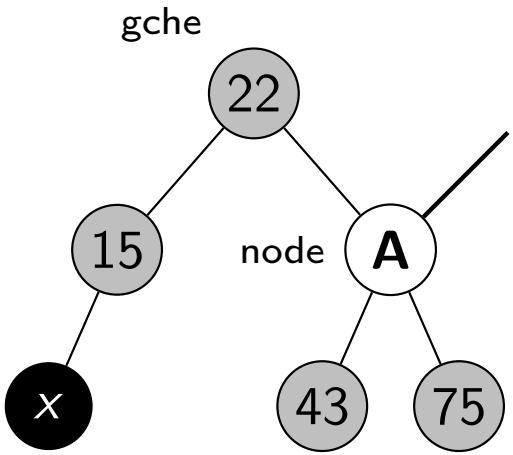


Étape 2 : `node->left = gche->right`

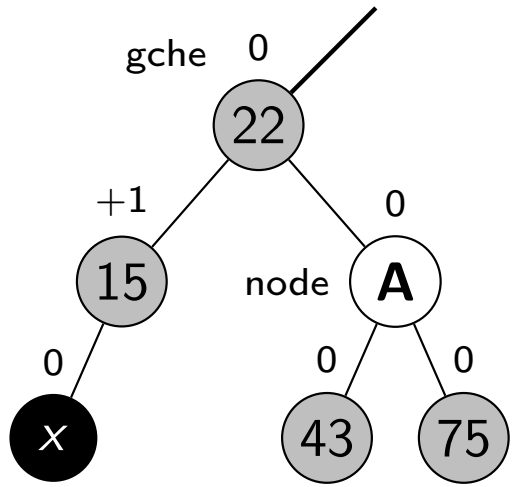


Rotation : left-left (LL)

Étape 3 : `gche->right = A`

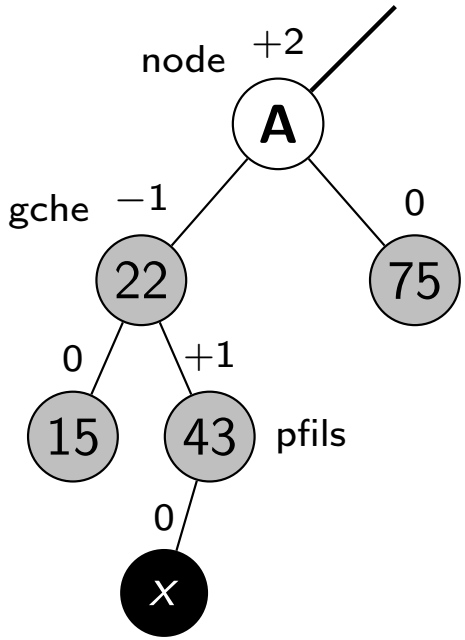


Étape 4 : Mise à jour du pointeur parent de A et des poids

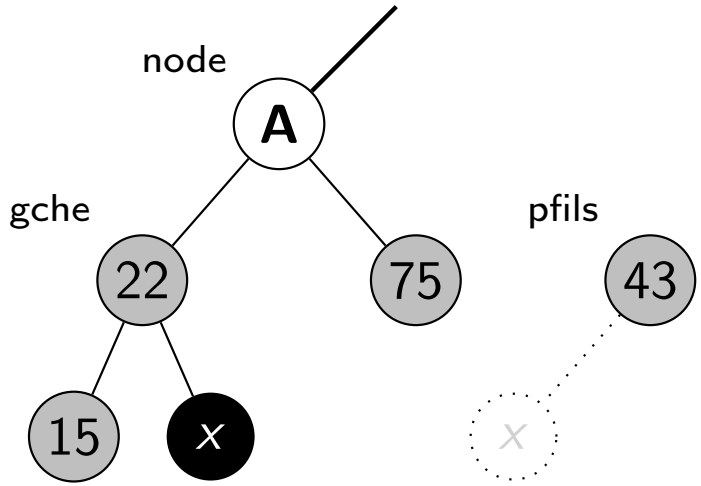


Rotation : left-right (LR)

Étape 1 : insertion de x

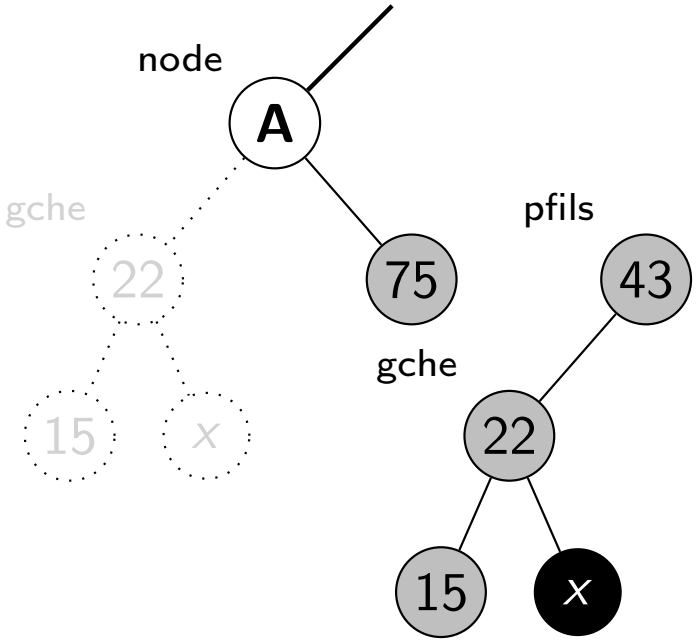


Étape 2 : `gche->right = pfils->left`

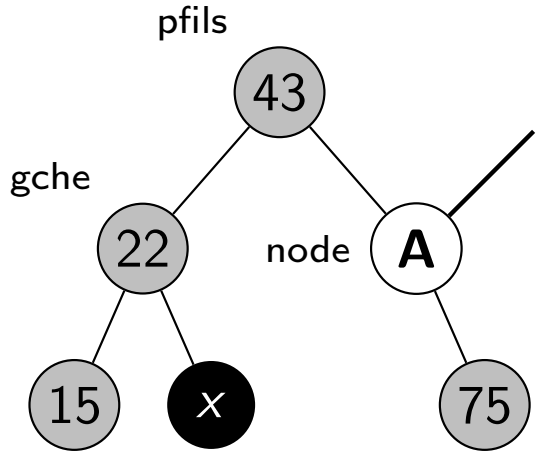


Rotation : left-right (LR)

Étape 3 : `pfils->left = gche`

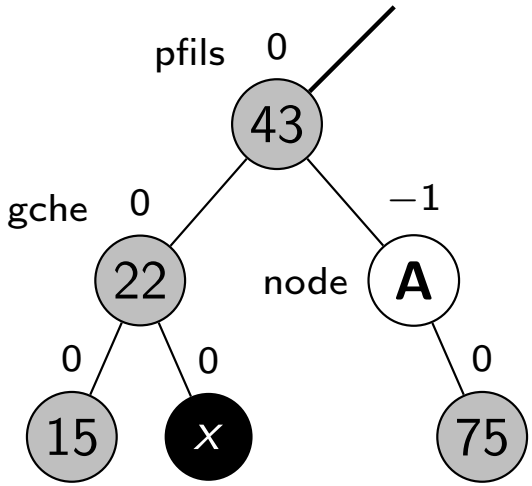


Étape 4 : `pfils->right = A`



Rotation : left-right (LR)

Étape 5 : Mise à jour du pointeur parent de A et des poids



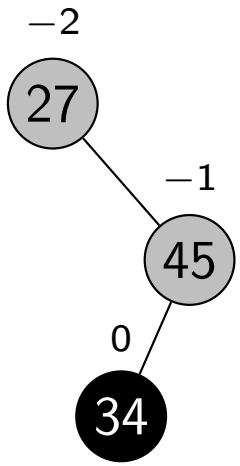
Remarque

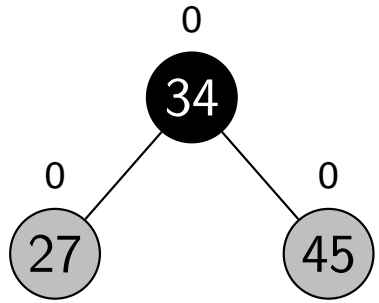
Mise à jour des poids des nœuds `A`, `pfils` et `gche` dépend de la position d'insertion de `x`

- x est insérer comme fils droit de pfils (déséquilibre sur `A`, facteur -1)
- x est insérer comme pfils (équilibré)
- x est insérer comme fils gauche de pfils (déséquilibre sur `gche`, facteur $+1$)

Exemple : construction d'un AVL – 1

Insertions consécutives de 27, 45, 34





Commentaire

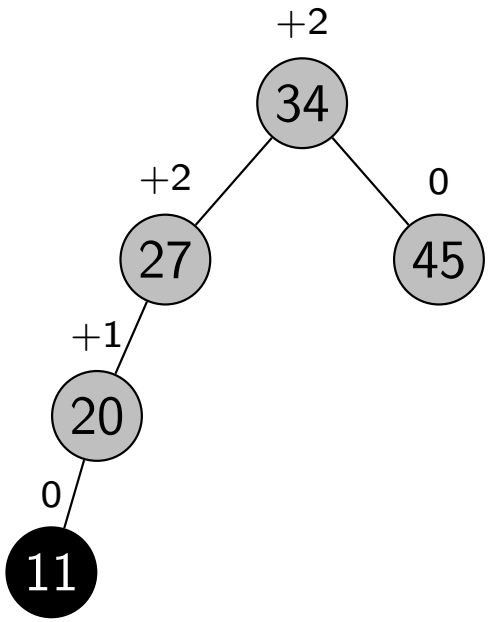
L'insertion de 34 créé un déséquilibre qui déclanche une rotation RL.

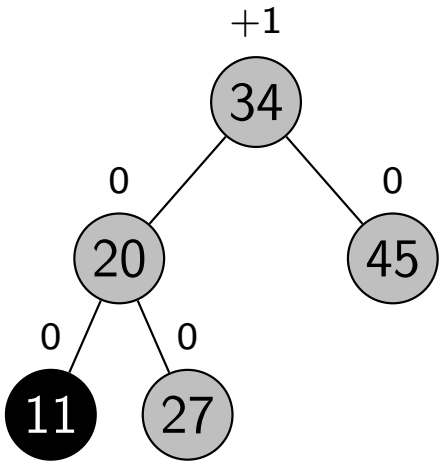
Commentaire

Après rotation RL

Exemple : construction d'un AVL – 2

Puis insertions consécutives de 20 et 11





Commentaire

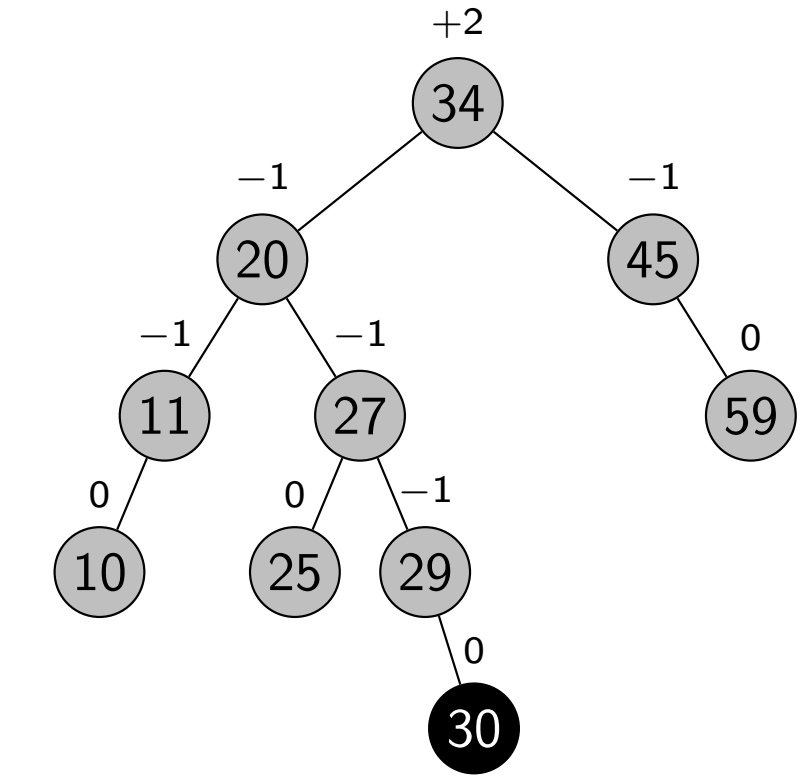
L'insertion de 11 créé un déséquilibre qui déclanche une rotation LL.

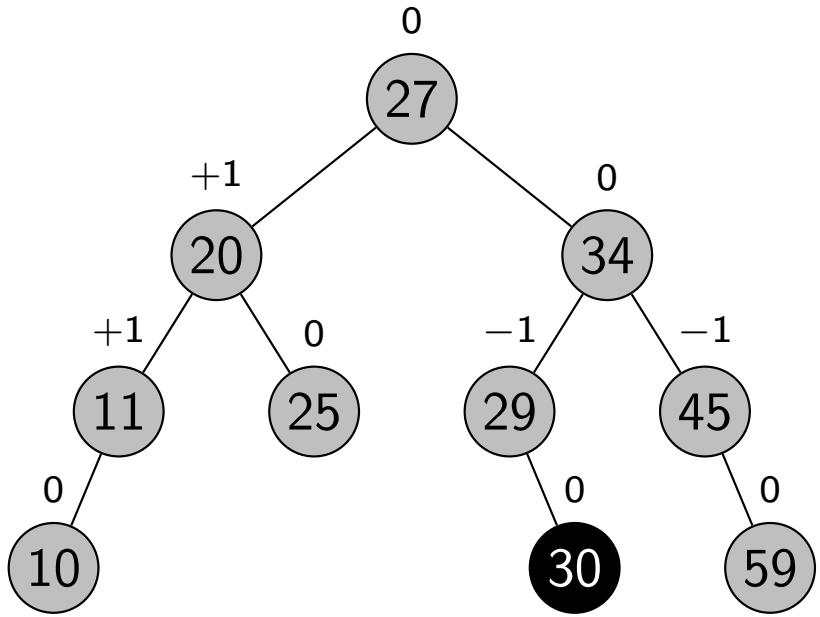
Commentaire

Après rotation LL

Exemple : construction d'un AVL – 3

Puis insertions consécutives de 59, 10, 25, 29 et 30





Commentaire

L'insertion de 30 créé un déséquilibre qui déclanche une rotation LR.

Commentaire

Après rotation LR