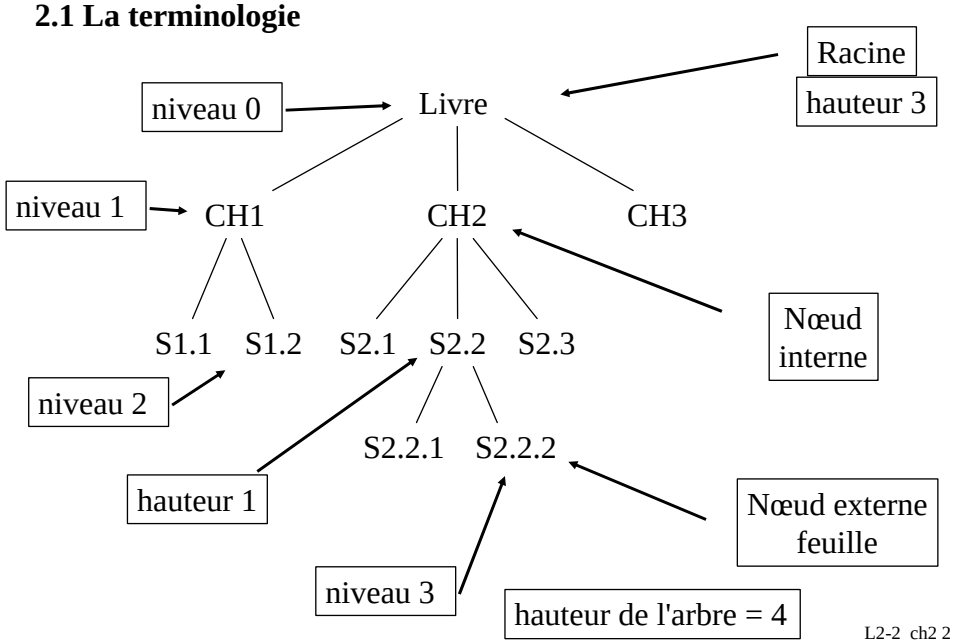
CH.2 ARBRES

- 2.1 La terminologie
- 2.2 La représentation des arbres
- 2.3 Les arbres binaires
- 2.4 Les codes de Huffman
- 2.5 Les arbres binaires de recherche
- 2.6 Les arbres AVL
- 2.7 Les arbres partiellement ordonnés

2.1 La terminologie



Terminologie en partie généalogique :
père, fils, frères, descendants, ascendants.
Existence d'un chemin unique entre un ascendant et chaque descendant.

Structure de données appropriée à la manipulation d'objets entre lesquels il existe une relation hiérarchique :

- Arbre généalogique
- Structure d'un livre
- Structure d'un système de fichiers

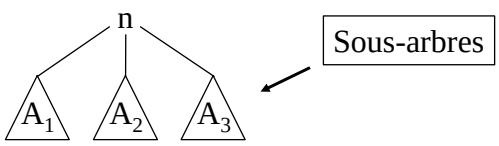
Structure de données indissociable de la récursivité :

- Structure de l'exécution d'un programme récursif
- Représentation compacte de données syntaxiques (arbres abstraits)
- Représentations propices à des démonstrations ou à des algorithmes (codes, tris)

Les noeuds des arbres contiennent des informations sous forme d'étiquettes :
noms, emplacements en mémoire, pointeurs vers des données.

Analogue aux listes chaînées, avec structure plane et plus seulement linéaire.

Définition récursive d'un arbre :
- Arbre vide
- Un seul noeud = racine
- Si on dispose d'un certain nombre d'arbres, on peut les "raccrocher" à un nouveau noeud, qui devient racine.



Ordre entre les sommets

Les frères sont ordonnés (âge, adresse,...) bien que souvent sans incidence sur le problème.

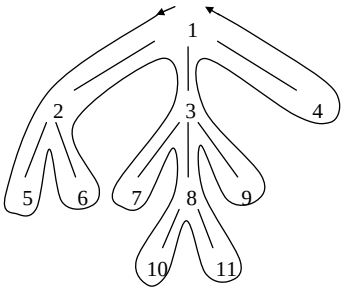
Parcours d'arbres

Problème : accéder à l'information contenue dans les noeuds d'un arbre (analogue au parcours d'une liste).

Parcours à partir de la racine en *profondeur*.

On peut prendre connaissance de l'information attachée à chaque noeud dans un ordre ou dans un autre :

- ordre *préfixe* : on lit l'information la première fois ;
- ordre *postfixe* : on lit cette information la dernière fois.



Parcours en profondeur

Ordre de visite des sommets

1 2 5 2 6 2 1 3 7 3 8 10 8 11 8 3 9 3 1 4 1

Ordre préfixe

1 2 5 2 **6** 2 1 **3 7** 3 **8 10** 8 **11** 8 3 **9** 3 1 **4** 1

Ordre postfixe

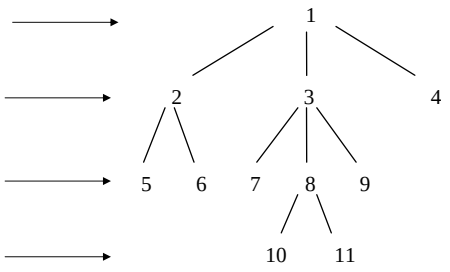
1 2 **5** 2 **6 2** 1 3 **7** 3 8 **10** 8 **11 8** 3 **9 3** 1 **4 1**

```
fonction PROFONDEUR(noeud n)
{
   écrire n ;
   pour chaque fils f de n de gauche à droite
   faire
   {
      PROFONDEUR(f); écrire n ;
   }
}
```

Exploration à partir d'un noeud *n*.

```
fonction PREFIXE(noeud n)
{
   écrire n ;
   pour chaque fils f de n de gauche à droite faire PREFIXE(f);
}


fonction POSTFIXE(noeud n)
{
   pour chaque fils f de n de gauche à droite faire POSTFIXE(f);
   écrire n ;
}
```


Parcours à partir de la racine en *largeur*.
On parcourt l'arbre niveau après niveau.



Parcours en largeur

1 / 2 3 4 / 5 6 7 8 9 / 10 11

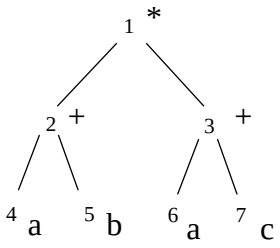
Pas de programme récursif.
Structure appropriée = file (FIFO) de structures (nœud, entier).
Le nombre sera la hauteur du nœud. On peut soit garder le noeud, soit un pointeur vers celui-ci (selon l'implémentation choisie.)

```
fonction LARGEUR(arbre A)
{
   file F ; noeud n, f ; entier h ;
   ENFILER(F, (racine de A, 0));
   jusqu'à ce que (F est vide) faire
   {
      (n, h) = DEFILER(F) ;
      écrire("hauteur de", n, "=", h) ;
      pour chaque fils f de n de gauche à droite
      faire ENFILER(F, (f, h + 1)) ;
   }
}
```

Temps de parcours linéaire.

Exemple : expressions arithmétiques, arbres abstraits

Expression (a + b) * (a + c) représentée par l'arbre étiqueté :



Arbre abstrait

Ordre préfixe : * + a b + a c

Ordre postfixe : a b + a c + *

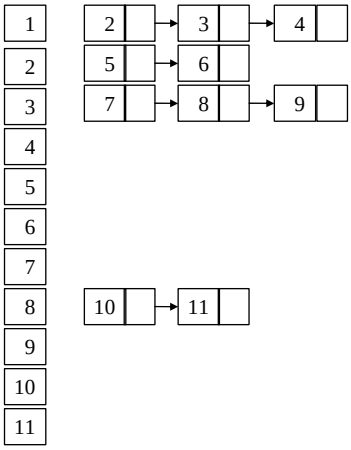
Les noeuds internes ont pour étiquettes les opérateurs, les noeuds externes sont étiquetés par des identificateurs.

La connaissance du nombre d'opérandes de chaque opérateur permet de reconstituer l'arbre (donc l'expression) à partir de l'ordre préfixe ou de l'ordre postfixe (mais pas de l'ordre infixe).
D'où les noms de forme préfixe et postfixe d'une expression arithmétique.

2.2 La représentation des arbres

Deux représentations utiles :

• Liste des fils :



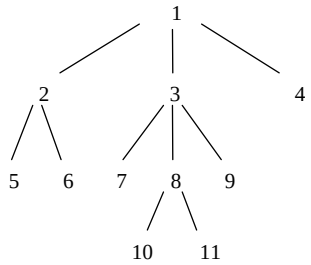


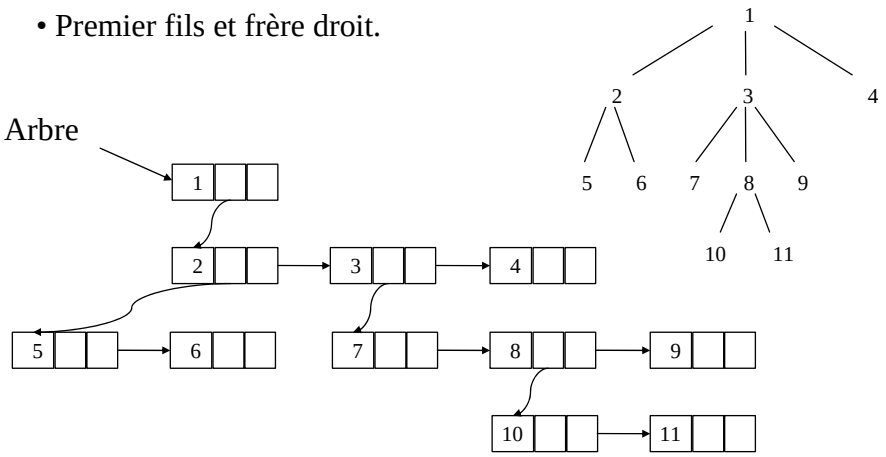
Tableau de listes chaînées.
Les étiquettes sont gérées par un autre tableau.

Pour construire un arbre à partir de sous-arbres, il faut renuméroter les noeuds.

Exemple de mise en oeuvre de l'ordre préfixe

```
typedef struct cellule           /* noeud et pointeur */
{     int numero;
      struct cellule *suivant;
} *LISTE;

LISTE arbre[TAILLE];             /* tableau de listes */

void prefixe(int);

void predecrit(LISTE l)          /* parcourt les sous-arbres */
{     if (l != NULL)
      {     prefixe(l->numero);
            predecrit(l->suivant);
      }
}

void prefixe(int noeud)          /* fonction principale */
{     printf("%d ", noeud);
      predecrit(arbre[noeud]);
}
```

• Premier fils et frère droit.



Pas de nécessité de numéroter les noeuds : les cellules peuvent contenir les étiquettes (ou un pointeur vers celles-ci).

Exemple de mise en oeuvre de l'ordre préfixe

```
typedef struct cellule           /* étiquette et pointeur */
{     int numero;
      struct cellule *fils;
      struct cellule *frere;
} CELLULE, *LISTE;

LISTE arbre;                     /* pointeur vers CELLULE */

void prefixe(LISTE l)
{     if (l != NULL)
      {     printf("%d ", l->numero);
            prefixe(l->fils);
            prefixe(l->frere);
      }
}
```

Ordre postfixe : attention à l'ordre des appels récursifs !

```
void postfixe(LISTE l)
{  if (l != NULL)
   {  postfixe(l->fils);
      printf("%d ", l->numero);
      postfixe(l->frere);
   }
}
```

On a souvent besoin de remonter d'un noeud vers son père.

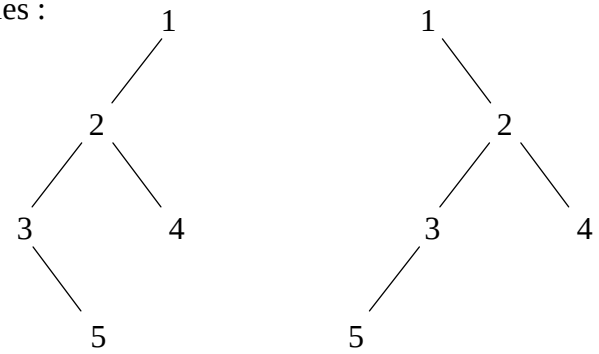
Dans le cas des listes de fils, on retrouve le père par recherche dans les liste (ou fonction précalculée).

Dans le cas premier fils-frère droit, il vaut mieux introduire un pointeur vers le père (*cf.* listes doublement chaînées).

2.3 Les arbres binaires

Chaque noeud a un fils gauche, un fils droit, les deux ou aucun.

Exemples :



Deux arbres binaires différents

Propriétés :
- un arbre binaire de hauteur n a au plus $2^{n+1} – 1$ noeuds dont 2^n noeuds externes.
- le nombre de feuilles est égal à un de plus que le nombre de noeuds de degré 2.

Arbre binaire complet : aucun noeud de degré 1 ; il a k noeuds internes et $k + 1$ noeuds externes

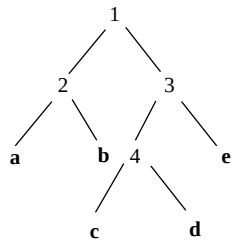
Arbre binaire entier (plein) : il a exactement 2^i noeuds au niveau i, pour tout i

Le type arbre binaire (fait apparaître la construction récursive) :

```
typedef struct noeud *ARBRE_BINAIRE;
typedef struct noeud          /* étiquette et pointeur */
{  int numero;
   ARBRE_BINAIRE fils_gauche;
   ARBRE_BINAIRE fils_droit;
} NOEUD;
```


2.4 Les codes de Huffman

Arbres et codes de longueur variable : codes préfixes, codes de Huffman



a codé 00 b codé 01 c codé 100
d codé 101 e codé 11
Les noeuds internes sont sans signification.

Code de longueur variable, ayant la propriété du *préfixe*, qui permet un décodage non ambigu en une seule lecture :

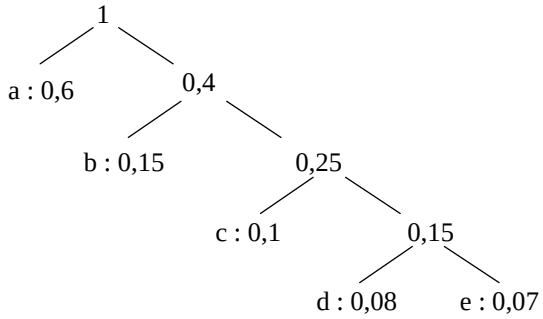
Ici, le mot abacedb est codé 0001001001110101.
Le décodage se fait en parcourant l'arbre binaire.

Lorsque les lettres apparaissent avec des fréquences différentes, on cherche à coder les lettres fréquentes par des mots courts et les lettres rares par des mots longs.

Supposons que les fréquences sont :
a : 60%, b : 15%, c : 10%, d : 8% et e : 7%.
Un codage de longueur fixe (trois bits/lettres) donne donc, en moyenne 3 *n* symboles pour un mot de *n* lettres.
Le code précédent donne
(0,6 x 2 + 0,15 x 2 + 0,1 x 3 + 0,08 x 3 + 0,07 x 2) x *n*,
soit en moyenne 2,18 *n* symboles pour le même mot.

L'algorithme de Huffman permet d'obtenir le meilleur code possible compte tenu des fréquences d'apparition.

On l'obtient en construisant l'arbre par les feuilles correspondant aux lettres les moins fréquentes et en plaçant à chaque noeud la somme des fréquences des fils a : 60%, b : 15%, c : 10%, d : 8% et e : 7% :



a codé 0 b codé 10
c codé 110 d codé 1110
e codé 1111

Ce code donne (0,6 x 1 + 0,15 x 2 + 0,1 x 3 + 0,08 x 4 + 0,07 x 4) x *n*, soit en moyenne 1,8 *n* symboles pour le même mot.

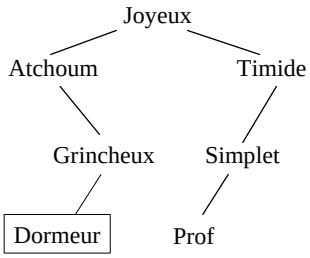
C'est le mieux qu'on puisse faire en codant chaque caractère.

2.5 Les arbres binaires de recherche

Dictionnaire : ensemble de données ordonnées (p. ex. mots avec l'ordre du dictionnaire).
On veut un modèle permettant une recherche rapide et assez facile à actualiser (suppression-insertion). Une liste chaînée permet d'effectuer toutes ces opérations. Si on a n mots, le temps des opérations de recherche, effacement et insertion sont $O(n)$.

Un *arbre binaire de recherche* (ABR) a ses noeuds étiquetés par les mots du dictionnaire. La propriété de structure d'un tel arbre est la suivante :
Si x est un noeud, y un noeud de son sous-arbre gauche et z un noeud de son sous-arbre droit, alors étiq(y) < étiq(x) < étiq(z).

Par exemple :



La recherche de w se fait facilement de manière récursive :

ou bien w est à la racine,
ou bien w est plus petit : on le cherche dans le sous arbre gauche,
ou bien w est plus grand : on le cherche dans le sous arbre droit.

L'insertion se fait de même :

ou bien l'arbre est vide : on crée un noeud unique étiqueté w,
ou bien w est plus petit que la racine : on l'insère dans le sous arbre gauche,
ou bien w est plus grand que la racine : on l'insère dans le sous arbre droit.

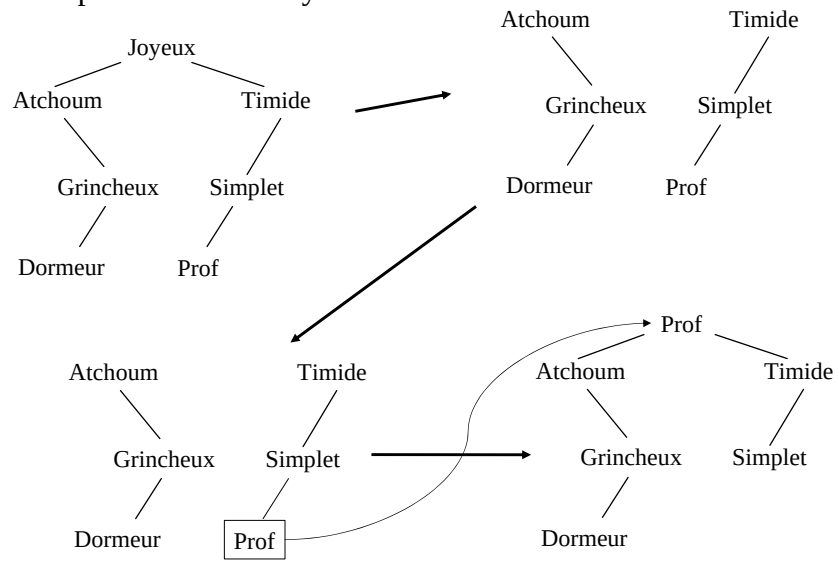
L'effacement est plus délicat :

si *w* est à la racine et si celle-ci n'a pas de fils, on obtient l'arbre vide,
si *w* est à la racine et si celle-ci n'a qu'un fils, on efface la racine,
si *w* est à la racine et si celle-ci a deux fils, on efface la racine ;
il reste deux sous arbres ; on enlève la plus petite étiquette *x* du sous arbre droit, qu'on place comme nouvelle racine.

Il reste à trouver *x* et à l'enlever ; pour enlever la plus petite étiquette d'un arbre non vide :

si la racine n'a pas de fils gauche, *x* est l'étiquette de la racine et on enlève la racine,
si la racine a un fils gauche, on cherche et on enlève *x* la plus petite étiquette du sous-arbre gauche.

Exemple : on enlève Joyeux



Avantage au niveau de la complexité : les opérations sur les mots nécessitent un nombre de manipulations égal à la hauteur k de l'arbre plus un. Si celui-ci est équilibré et complet, le nombre de noeuds (donc d'étiquettes) vaut $2^{k+1} - 1$. La hauteur est donc de l'ordre de $O(\log n)$, si n est le nombre d'étiquettes, à comparer à $O(n)$ si on traite le dictionnaire comme une liste chaînée.

Pour avoir un arbre aussi équilibré que possible, il faut insérer les étiquettes dans le désordre (ordre aléatoire). On peut aussi "corriger" les déséquilibres quand ils apparaissent : arbres AVL.

L'insertion dans l'ordre provoque en effet un arbre binaire dans lequel les noeuds n'ont qu'un fils droit, équivalent à une liste chaînée.

2.6 Les arbres AVL

Adelson-Velskii et Landis :
structure d'ABR équilibré et algorithmes permettant de maintenir cet équilibre.

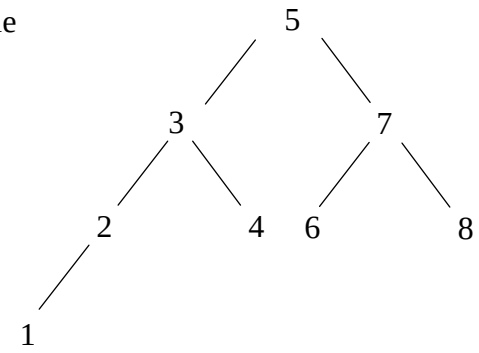
$h_d(a)$ = hauteur du sous-arbre droit du noeud a ;
$h_g(a)$ = hauteur du sous-arbre gauche du noeud a ;

BAL(a) = $h_d(a) - h_g(a)$ = *facteur d'équilibrage* du noeud a.

Arbre AVL : ABR et,
pour tout noeud, le facteur d'équilibrage vaut 0, + 1 ou – 1.

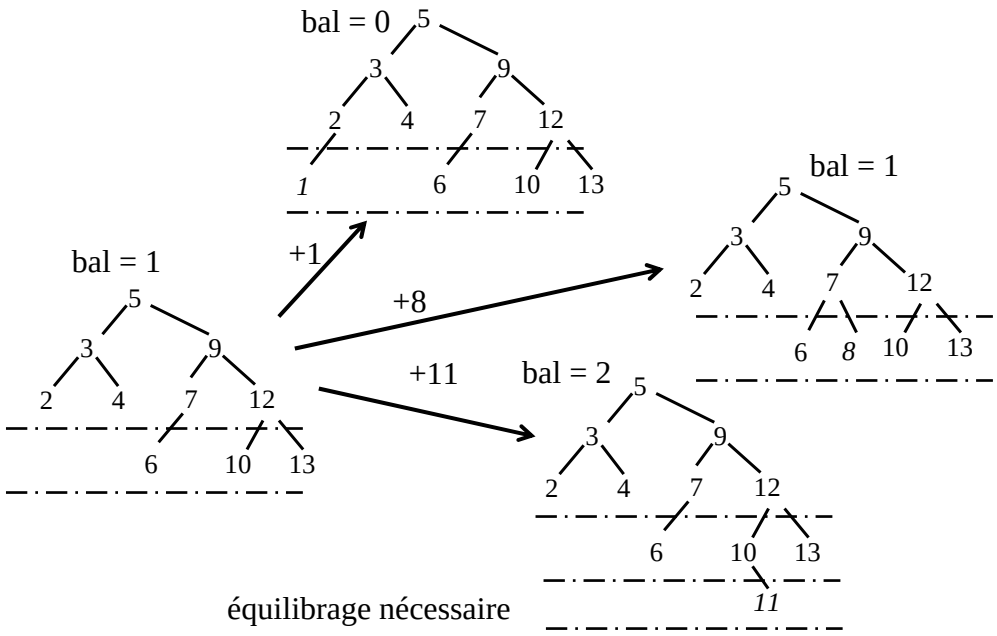
Ou récursivement : si x est la racine, y son fils gauche et z son fils droit,
étiq(y) < étiq(x) < étiq(z) ;
BAL(x) = 0, + 1 ou – 1 ;
les deux sous-arbres de la racine sont des AVL.

Exemple



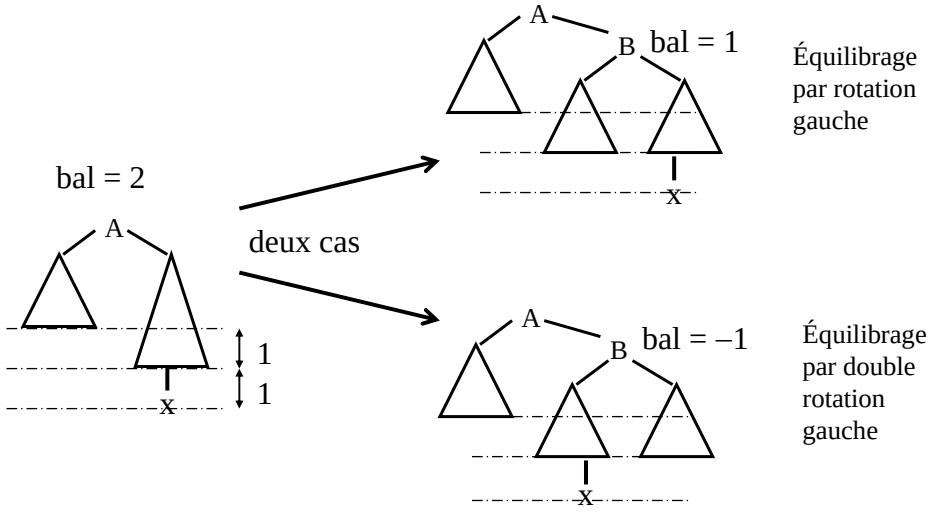
Le nombre de noeuds d'un arbre AVL de hauteur k vérifie :
$m(k) = m(k-1) + m(k-2) - 1$, donc
$m(k) = F_{k+2} - 1$ où F_k est le k-ième nombre de Fibonacci.
Le nombre de noeuds n d'un arbre AVL de hauteur k vérifie par passage aux logarithmes $\log(n) \sim \log(\varphi)\, k$, ou encore $k \sim 1{,}45 \log(n)$.

L2-2 ch2 27



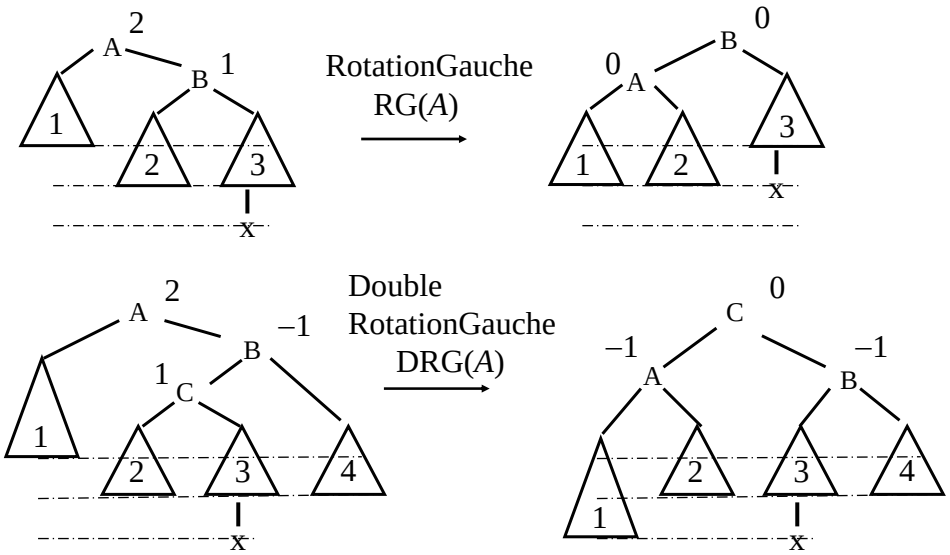
L2-2 ch2 28

L'insertion modifie le facteur d'équilibrage de ses ancêtres de 0 ou ± 1. On cherche (en remontant) le premier ancêtre dont le bal est ± 2. Son fils ne peut pas être à 0. Deux cas :



L2-2 ch2 29

Rééquilibrage après ajout.



L2-2 ch2 30

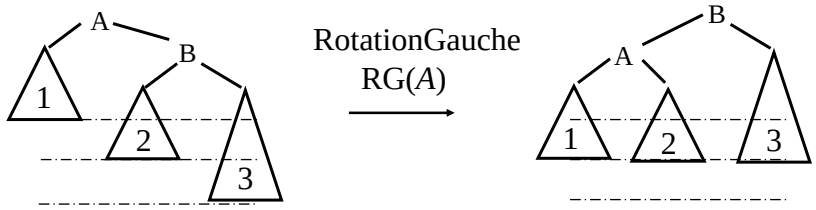
On peut définir de façon analogue des rotations droites et des doubles rotations droites.

Enlever un noeud dans un ABR aboutit toujours à l'enlèvement d'une feuille. Un tel enlèvement peut aboutir à un arbre déséquilibré. Dans ce cas, on doit rétablir l'équilibre par des rotations ou des doubles rotations.

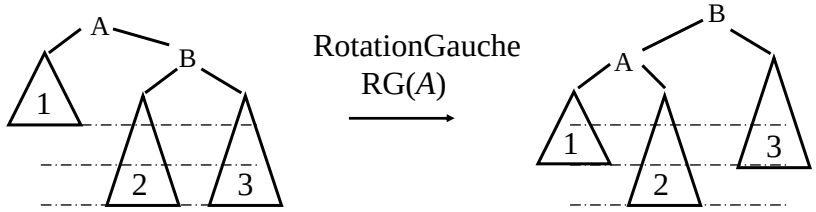
La situation déséquilibrée après enlèvement peut être des types suivants.

Si le noeud enlevé était à gauche de *A*. Selon la cause du déséquilibre.

Cas 1-a

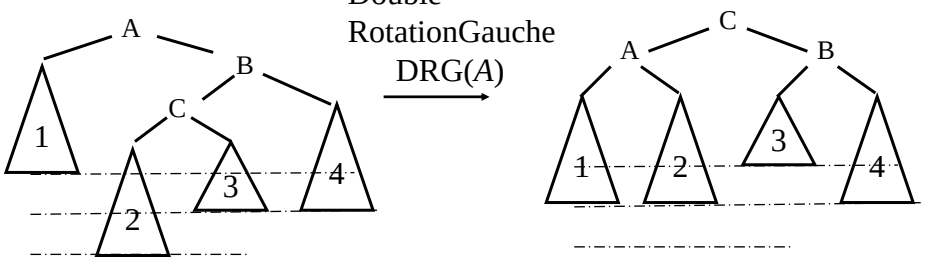


Cas 1-b



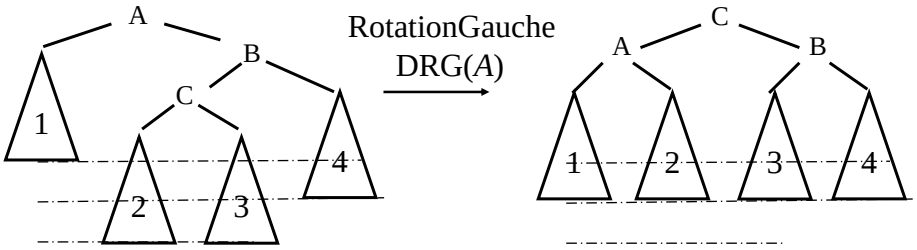
Cas 2-a

Double RotationGauche DRG(*A*)



Cas 2-b

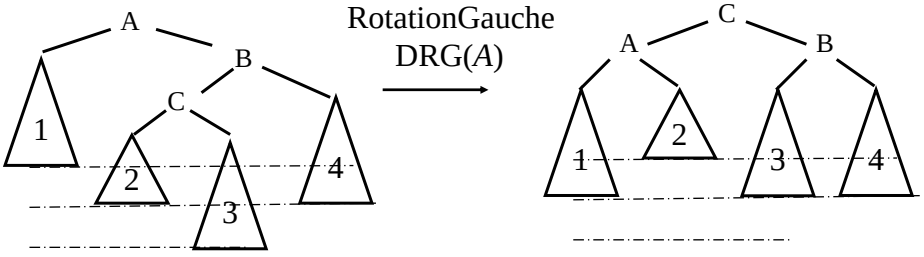
Double RotationGauche DRG(*A*)



L2-2 ch2 33

Cas 2-c

Double RotationGauche DRG(*A*)



Les rotations et doubles rotations droites se traitent de la même façon.

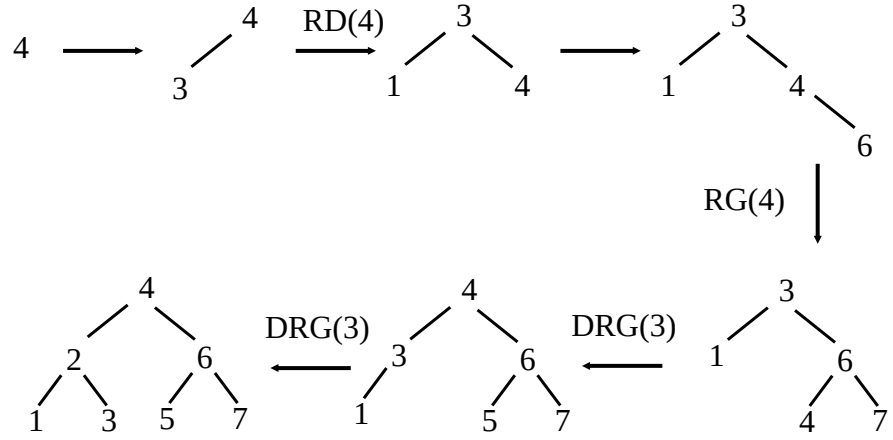
En fait, les divers sous-cas n'ont pas besoin d'être distingués dans la réalisation.

Ces cas permettent aussi de traiter l'insertion.

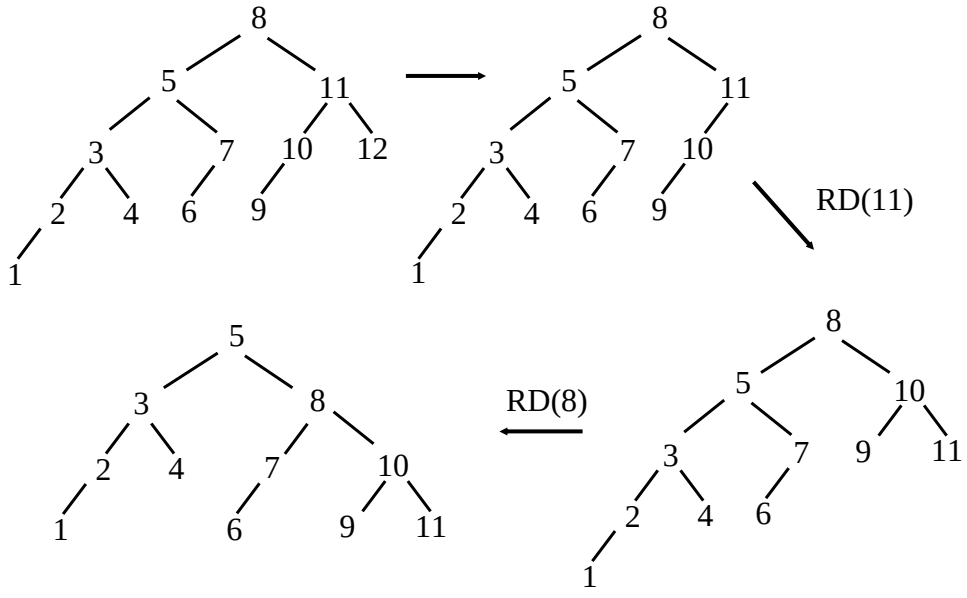
La double rotation gauche peut s'écrire avec des rotations simples : DRG(*A*) = RD(*B*) suivi de RG(*A*)

L2-2 ch2 34

Exemple : insertions successives de 4 3 1 6 7 5 2



Exemple : effacement de 12



Analyse du temps nécessaire

L'insertion provoque une seule opération de rééquilibrage ; il faut insérer ($O(\log n)$), puis trouver le déséquilibre en remontant vers la racine ($O(\log n)$) enfin corriger ($O(1)$) ; bilan ($O(\log n)$).
L'effacement nécessite la recherche du minimum sur la branche gauche ($O(\log n)$), puis remonter vers la racine pour corriger le premier déséquilibre, puis d'autres déséquilibres éventuels, chacun en ($O(1)$), pour un maximum de $O(\log n)$ corrections ; bilan ($O(\log n)$).

Les arbres AVL représentent donc une implémentation d'AVL qui assure, sans surcoût excessif, que la hauteur reste bien en ($O(\log n)$).

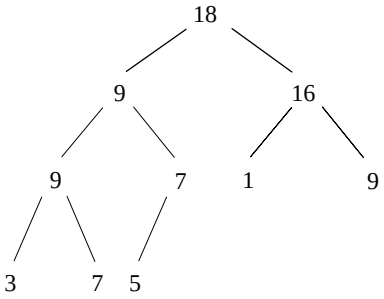
2.7 Les arbres partiellement ordonnés

Problème : des tâches sont affectées de niveaux de priorité. Comment insérer une nouvelle tâche dans la file des tâches existantes et comment trouver la tâche de priorité la plus élevée pour l'enlever de la file.

Modèle approprié : *arbre partiellement ordonné* et sa mise en oeuvre : *tas* ("*heap*"). On peut utiliser cette structure pour trier des données : c'est le *tri par tas* ("*heapsort*").

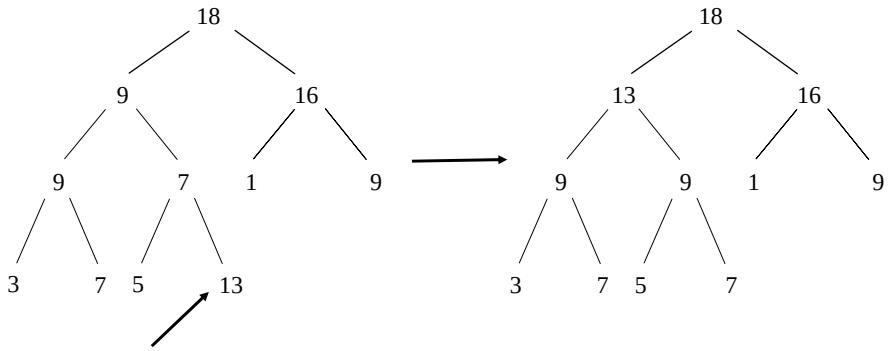
Arbre partiellement ordonné équilibré (APO) : arbre binaire dont l'étiquette de chaque noeud est supérieure ou égale à l'étiquette de ses fils (APO), tel que tous les niveaux sont complet sauf éventuellement le plus bas, dont les noeuds sont le plus à gauche possible.

APO équilibré



Opérations nécessaires : insertion d'une étiquette,
effacement de l'étiquette de la racine.

Insertion de l'étiquette 13 :

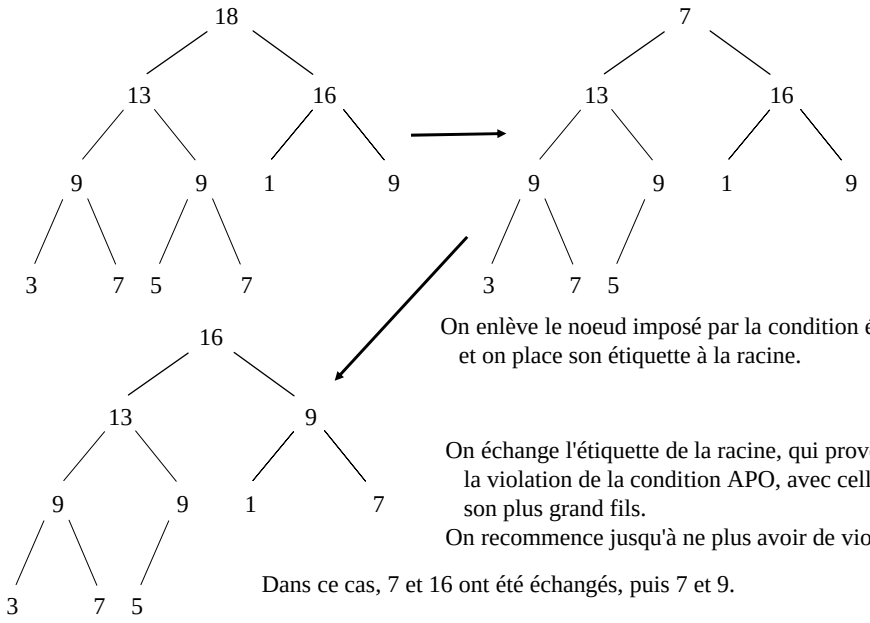


Nouveau noeud, imposé par la condition d'APO équilibré

On échange l'étiquette qui provoque la violation de la condition APO avec celle de son père.
On recommence jusqu'à ne plus avoir de violation.

Ici, 13 et 7 ont été échangés, puis 13 et 9.

Suppression de l'étiquette 18



On enlève le noeud imposé par la condition équilibré et on place son étiquette à la racine.

On échange l'étiquette de la racine, qui provoque la violation de la condition APO, avec celle de son plus grand fils.
On recommence jusqu'à ne plus avoir de violation.

Dans ce cas, 7 et 16 ont été échangés, puis 7 et 9.

Mise en oeuvre des APO par la structure de données tas.

Les étiquettes des *n* noeuds sont stockées dans un tableau *A* de taille *n*.
A[1] contient l'étiquette de la racine, *A*[2] et *A*[3] les étiquettes de ses deux fils. En général, *A*[2**i*] et *A*[2**i*+1] contiennent les étiquettes des deux fils du noeud dont l'étiquette est *A*[*i*].
Le contenu du tas est en fait la liste des étiquettes niveau par niveau.

Les opérations d'insertion et d'effacement du maximum se font facilement dans le tas par échange des étiquettes.

Exemple

1	2	3	4	5	6	7	8	9	10
18	9	16	9	7	1	9	3	7	5

Insertion de 13

1	2	3	4	5	6	7	8	9	10	11
18	9	16	9	7	1	9	3	7	5	13

On compare $A[11]$ et $A[11 / 2]$, qu'on échange, et on itère.

1	2	3	4	5	6	7	8	9	10	11
18	13	16	9	9	1	9	3	7	5	7

Suppression du maximum 18

1	2	3	4	5	6	7	8	9	10	11
18	13	16	9	9	1	9	3	7	5	7

On retire $A[1]$ qu'on remplace par $A[11]$.

1	2	3	4	5	6	7	8	9	10	11
7	13	16	9	9	1	9	3	7	5	18

On échange $A[1]$ et le max de $A[2]$ et $A[3]$, et on itère.

1	2	3	4	5	6	7	8	9	10	11
16	13	9	9	9	1	7	3	7	5	18

On a à faire deux types d'échanges d'étiquettes :
un *échange montant*, vers la racine, servira pour l'insertion.

```
fonction echange_montant(tableau A, entier i)
{
   si ((i > 1) et (A[i] > A[i/2])) alors
   {
       echanger(A,i,i/2);
       echange_montant(A, i/2);
   }
}

fonction insere(tableau A, entier k, entier n)
{
   A[n] = k ;
   echange_montant(A, n);
}
```

On suppose que le tableau A a été alloué avec une taille suffisante.
L'entier n est la taille du tas après insertion.

Autre type d'échange :
un *échange descendant*, depuis la racine, pour la suppression du maximum.

```
fonction echange_descendant(tableau A, entier i, entier n)
{
   fils = 2*i ;
   si ((2*i < n) et (A[2*i] < A[2*i+1])) alors
   {
      fils = 2*i+1;
   }
   si ((2*i <= n) et (A[i] < A[fils])) alors
      {
         echanger(A,fils, i);
         echange_descendant(A, fils, n);
      }
   }
}
```

On suppose ici que le tas occupe les *n* premières positions du tableau *A*.

L'extraction du maximum d'un tas de taille *n*, déplacé à la position *n*.

```
fonction extraction_maximum(tableau A, entier n)
{
   echanger(A, n, 1);
   echange_descendant(A, 1, n-1);
}
```

Si la file de priorité est déjà constituée en tas, on peut trier celui-ci :

```
fonction trier_tas(tableau A, entier n)
{
   pour (i = n, i > 1, i--) faire
   {
      extraction_maximum(A, i);
   }
}
```

Complexité des opérations sur les tas :
Un APO équilibré ayant n noeuds est de hauteur $\lfloor \log_2(n) \rfloor$.
L'insertion nécessite donc un nombre de comparaisons $O(\log(n))$.
Il en est de même de l'extraction du maximum.

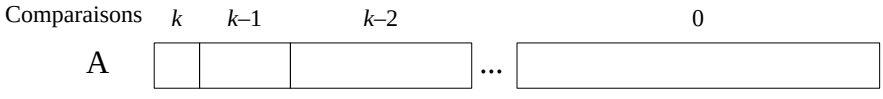
Par conséquent, le tri d'un tas requiert un nombre de comparaisons $O(n \log n)$.
La constitution d'un tas par insertions successives demande aussi un nombre de comparaisons $O(n \log n)$.

En fait, on peut ramener le temps de constitution d'un tas à $O(n)$ en effectuant une fabrication astucieuse de l'APO :
On place les étiquettes sans se soucier de l'ordre, puis on élimine les violations de la condition APO par échange descendant, des feuilles vers la racine.

Constitution d'un tas sur place. Le tableau A est de longueur n.

```
fonction constitution_tas(tableau A, entier n)
{
   pour (i = n/2, i >= 1, i--) faire
   {
      echange_descendant(A, i, n);
   }
}
```

Les violations de la condition APO sont corrigées aux niveaux inférieurs d'abord. À la fin, aucune violation ne subsiste : on a un tas.



Pour un tableau de taille n = $2^{k+1} - 1$. Le nombre total de comparaisons est donc $k+2(k-1)+2^2(k-2)+...+2^{k-1}1 = 2^{k+1} - k - 2 < n$

À démontrer !