Algorithmique

Chapitre 1.1 : Rééquilibrage des arbres binaires

Dans quel but ?

Arbre binaire → recherche dichotomique. Dichotomique signifie couper en **2**. Donc cela équivaut à une recherche binaire. On veut pouvoir donc faire une recherche binaire une fois l'arbre bien équilibré.

Dans un arbre binaire, après…	il reste…
→ Une opération	n / 2 clés
→ Deux opérations O(2)	n / 4 clés (on divise encore le nombre d'éléments /2)
→ Trois opérations O (3)	n / 8 clés
→ X opérations O(x)	n / 2^x = 1 clé, 2^x = n ➔ x = $\log_2 n$
	$O(\log_2 n) > O(n/2)$

Pour quoi utiliser le logarithme en base 2 ?

- Pour le software
- Pour le hardware

Ex : 25 en base 10 = ? en base 2 ➔ 011001

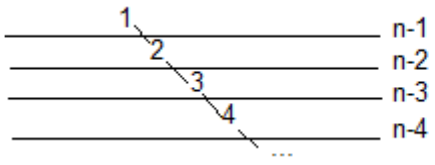
Il faut 5 à 6 digits pour représenter 25 : 5~6 = lg 25 (* lg = $\log^2 n$)

Soit la formule suivante (où « n » est le nombre à représenter)

$$\#DIGITS = [\log_2 n]$$

Insertion dans l'arbre

Les éléments à placer dans l'arbre sont triés. Comme on met les éléments plus petits que la racine à gauche tout partira à droite dans ce cas-ci :



on a "n" éléments
A chaque fois qu'on insère, il reste n - ... éléments
(voir dessin)

n = 4
On doit insérer 1 2 3 4

Cet arbre est *inutilisable* car il y a **DEGENERESCENCE de l'OCT**.

Rééquilibrage : On va procéder par étapes

1) Calculer la hauteur
2) Equilibre, stabilité, déséquilibre et instabilité
3) Rééquibrage d'un nœud
4) Insertion_Rééquilibrage
5) Hypothèse 0 / Def

Ces différentes étapes sont décrites page suivante.

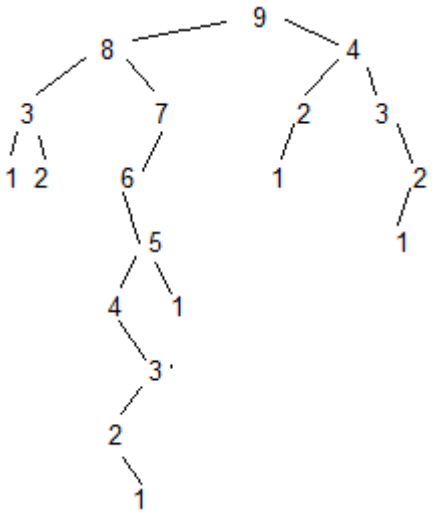
Première étape : Calculer la hauteur

La hauteur d'un arbre est la distance entre la racine et le point le plus éloigné.
Procédons par cas :

a) Dans le cas où *l'arbre est vide*

```
SI      arbin_vide (arbin)      // si l'arbre est vide
     ALORS      h= 0            // la hauteur vaut 0
     SINON      . . .
FIN SI
```

b) Dans le cas où *l'arbre contient des éléments*
Il faut prendre celui qui a la plus grande hauteur dans les sous-arbres. On caractérise la différence de hauteur entre les fils d'un nœud. Le déséquilibre d'un nœud est en fonction de la hauteur du nœud. (Exemple si le nœud de gauche est = 8 et celui de droite = 4)



On prend celui qui a **la plus grande hauteur** dans Les sous – arbres :

```
   h(arbin) = 1 +
max (h(gauche(arbin)),
     h(droite(arbin)))
```

```
// Dans le dessin on part du bas. La racine a une hauteur de 9. Son sous arbre de droite ne va pas valoir 3 car il y a plus de sous-arbres de son côté droit, donc il va valoir 4 !
```

L'algorithme de la hauteur donne au final :

```
PROCEDURE     hauteur(arbin)
SI arbin_vide(arbin)
     ALORS    h = 0 ;
     SINON    h = 1 + max(h(gauche(arbin)),h(droite(arbin))) ;
FIN SI
```

```
// Cette fonction est récursive et donc s'appelle plusieurs fois. Elle est donc très lourde à l'exécution. (Coût : O(2n))
```

Deuxième étape : Equilibre, stabilité, déséquilibre, Instabilité

On va devoir rééquilibrer un nœud.

Troisième étape : Rééquilibrer un nœud sur base des hypothèses

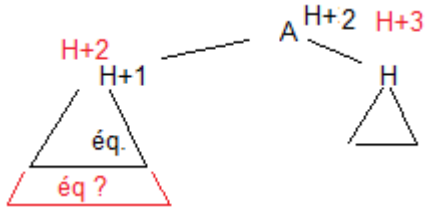
Hypothèse 1 : définir l'état interne des sous-arbres
1°) On sait rééquilibrer un nœud (Hypothèse 0 + Hypothèse 1)
2°) Le rééquilibrage possède une propriété très importante ➔ Procédure insertion_rééq.

Quatrième étape : Utilisation de la fonction « Ins_Rééq »

L'hypothèse 1 sera satisfaite !

Cinquième étape : H0/Def. sont suffisantes pour résoudre le problème.

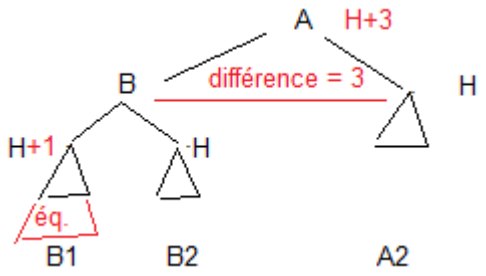
Exemple qui pose problème :



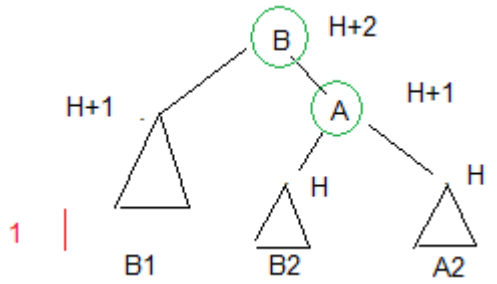
L'**hypothèse d'un arbre équilibré** est fort VIOLENTE !

```
// La hauteur H+2 pose problème.
```

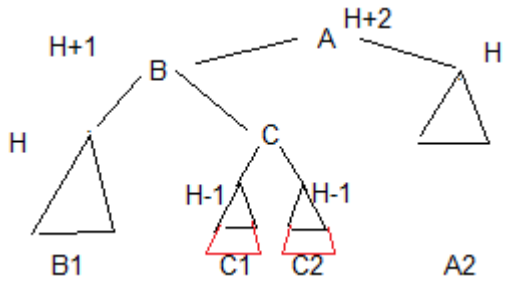
```
// Cas spécifiques
```



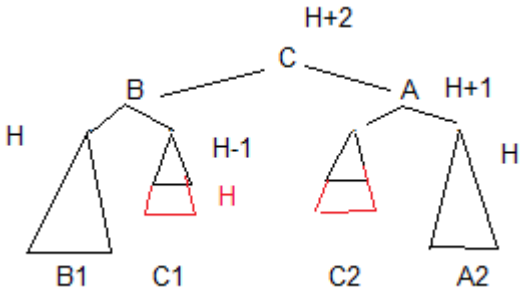
➔



On a 3 sous-arbres et 2 nœuds. Les positions relatives ne peuvent pas changer !



➔



Attention que C ne peut être mis ailleurs !

Cela amène à une propriété essentielle du rééquilibrage.
Le nœud qu'on rééquilibre est toujours le dernier nœud déséquilibré. On retrouve la hauteur initiale. **Le rééquilibrage restitue donc la hauteur initiale de l'arbre devenu déséquilibré** (On a conservé ici Hauteur + 2, et au premier exemple, la hauteur valait H+2 avant d'être déséquilibré !)

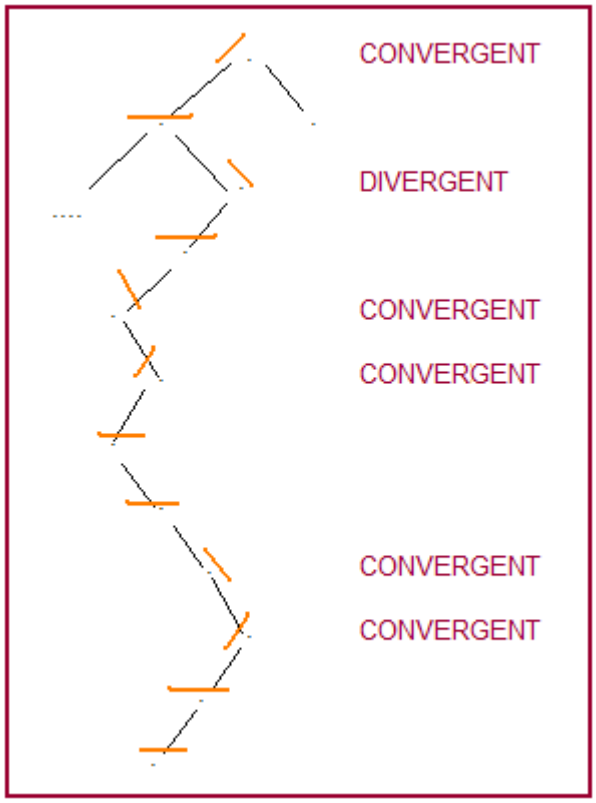
Chemin d'insertion
Légende : les barres dans l'arbre signifient qu'un nœud est stable. Leur direction indique si le nœud est convergent OU divergent (voir définitions plus loin)

On utilise une limite raisonnable[1] :

```
SI  | h(gauche(arbin)) – h(droite(arbin)) |    = 0
    ➔   arbin stable,  arbin = équilibré
SI  | h(gauche(arbin)) – h(droite(arbin)) |    = 1
    ➔   arbin instable, arbin = équilibré
SI  | h(gauche(arbin)) – h(droite(arbin)) |    >= 2
    ➔   arbin déséquilibré
```

[1] Ceci n'est pas un algorithme, par contre ceci est une NOTE DE BAS DE PAGE.

Rem : La hauteur des sous-arbres importe.



Un arbre complet contient $2^n - 1$ éléments.

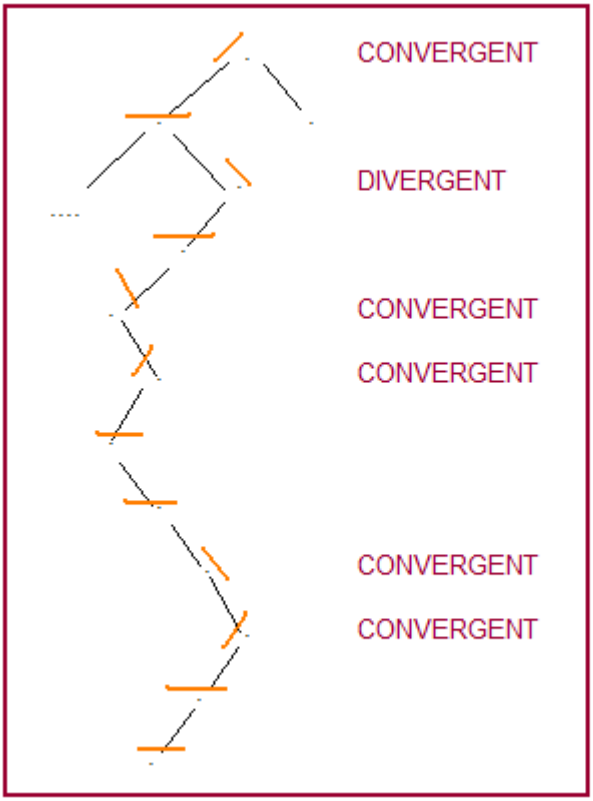
- Donc on accepte ici une différence de 1 entre la hauteur du sous-gauche et du sous-droit, sinon cela n'a aucun sens.
- Si la différence était = à 2, on aurait des arbres moins bien structurés.

Définitions :

Nœuds *convergents* : Vont dans la même direction

Nœuds *divergents* : Vont dans la direction opposée (penchent dans la direction inverse de celle où on va)

- Si on rencontre un convergent : on le rééquilibre
- Si on en rencontre un suivant plus bas, alors on garde le dernier et on ne rééquilibre pas les précédents.



Si on rencontre un divergent, on oublie le convergent déjà rencontré.

Si c'est un convergent, on le met de côté pour éventuellement rééquilibrer

Si ce n'est pas un divergent, est-ce l'autre ?

Si c'est stable, inutile de faire quelque chose

Si on arrive en dessous, insertion

Lors de l'appel du rééquilibrage si convergent, on fait des tests sur les hauteurs (appel de la fonction « hauteur »)

La hauteur est le **chemin le plus long**

Etant donné le **coût trop élevé de la fonction « HAUTEUR »,** on utilise un champ pour chaque nœud avec sa hauteur mise à jour à chaque fois, avant rééquilibrage si besoin

La fonction « **Consarbin** »

```
Fonction CONSARBIN (source, parbin_g, parbin_d, hauteur) : ARBIN
    // renvoie le pointeur du nœud
            Allocation de temp ;
            Temp->elem = source;
               Temp->gauche = parbin_g ;
               Temp->droite = parbin_d ;
               Temp->hauteur = hauteur ;
            Return(Temp) ;
```

Rappel de la fonction pour effectuer **un traitement** sur un arbre

```
PROC Fonction TRAITER_ARBIN(arbin)
    SI ( !arbin_vide(arbin))
        ALORS   TRAITER(racine(arbin)) ;
                TRAITER(gauche(arbin)) ;
                TRAITER(droite(arbin)) ;
        SINON   TRAITER(NULL) ;
    FIN SI          // 6 permutations possibles
```

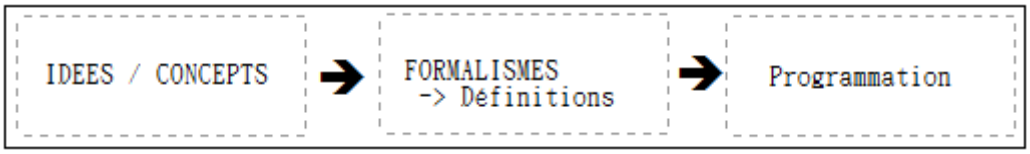
On en arrive maintenant à la fonction de **rééquilibrage**. *(DM = Donnée Modifiée)*

```
PROCEDURE INS_REEQ (DM : &pointeur vers arbin, DNM : élément source : CLE)
    ARBIN b, père_b, convergent, divergent, père_convergent ;
    SI   arbin_vide(arbin)
         ALORS     arbin = consarbin(source, NULL, NULL, 1) ;
         // On met la source à la racine, pas de sous-arbres encore.
         SINON     b = a ;
                   pere_b = convergent = divergent = père_convergent  = NULL ;
                   TANT QUE b != NULL FAIRE
                        SI source <= racine(b)
                             ALORS
                                  SI (h(gauche(b)) > h(droite(b))
                                       ALORS convergent = b
                                             Père_convergent = père_b
                                       SINON
                                            SI (h(droite(b)) > h(gauche(b))
                                                 ALORS     convergent = NULL ;
                                                           Divergent = b ;
                                            FIN SI
                                  FIN SI
                                  Père_b = b
                                  b = gauche(b)
                             SINON
                                  SI h (droite(b)) > h(gauche(b))
                                       ALORS convergent = b
                                             Père_convergent = père_b
                                       SINON
                                            SI h(droite(b)) < h(gauche(b))
                                                 ALORS     convergent = NULL ;
                                                           Divergent = b
                                            FIN SI
                                  FIN SI
                                  Père_b = b
                                  b = droite(b)
                        FIN SI    // b est vide, on va accrocher élément hors boucle
                   FIN TANT QUE  SI source <= r(père_b) ALORS psag@pere_b =
consarbin(source,NULL,NULL) SINON psad@pere_b = consarbin(source,NULL,NULL)
    FIN SI

    SI convergent != NULL                          // On a rencontré conv. Non annulé ?
         ALORS b = convergent ;
         SINON
              SI divergent != NULL      // rencontré divergent annulant convergent ?.
                   ALORS
                        SI source <= racine(divergent)  // quel coté était-on descendu
                             ALORS b = gauche(divergent)
                             SINON b = droite(divergent)
                        FIN SI
                   SINON
                   b = a
              FIN SI
     FIN SI
    FAIRE     hauteur++ de b                                          // maj hauteur
         SI source < racine(b)
              ALORS b = gauche(b)
              SINON b = droite(b)
    JUSQUE b = VIDE

    SI   convergent != NULL                        // on avait rencontré un convergent ?
         ALORS
              SI convergent = a                              // si convergent base arbre
                   ALORS     rééq(a)                         // rééquilibrer tout
                   SINON
                        SI source <= racine(père_convergent)
                             ALORS rééq(&sous_arbre_gauche@pere_convergent)
                             SINON rééq(&sous_arbre_droite@pere_convergent)
                        FIN SI
              FIN SI
    FIN SI
```

Programmation défensive
Permet d'être sûr du bon sens de ce qu'on a fait et d'avoir des informations de l'endroit par où on a accès ➔ cela permet le débugging.

Isomorphisme : programmation fonctionnelle basée sur l'isomorphisme ➔ fonctions essentielles deviennent des fonctions récursives !

Chapitre 1.2 : Concepts, formalisme et programmation



Idées et concepts :
Concepts émis au sujet …

- des structures de données (Structure Dynamique de Données : SDD)

Exemple :
- Pile
- File ➔ Contiennent des collections de données
- Liste liée

- des propriétés (juger sur la qualité et la quantité)

Exemple :
- Propriété de hauteur d'un arbre est la notion de chemin le plug long de la racine à la feuille la plus éloignée

- des traitements de la structure

Exemple :
- Parcours générique d'un arbre binaire (TRAITER)

Formalismes et définitions :
- Arbin : soit vide, soit non-vide
- Arbin non-vide : possède [racine, sous-arbre de gauche, sous-arbre de droite]
- Hauteur : si arbin vide, 0
- Hauteur : si arbin non-vide, 1 + MAX(h(g(a)),h(d(a)))
- Traitement : si vide, traiter le vide
- Traitement : si non vide, traitement spécifique

SDD (algèbre syntaxique) Propriétés Traitements

Programmation
Programmation des structures, des propriétés et des traitements en code !

Chapitre 1.3 : Algorithmes de base

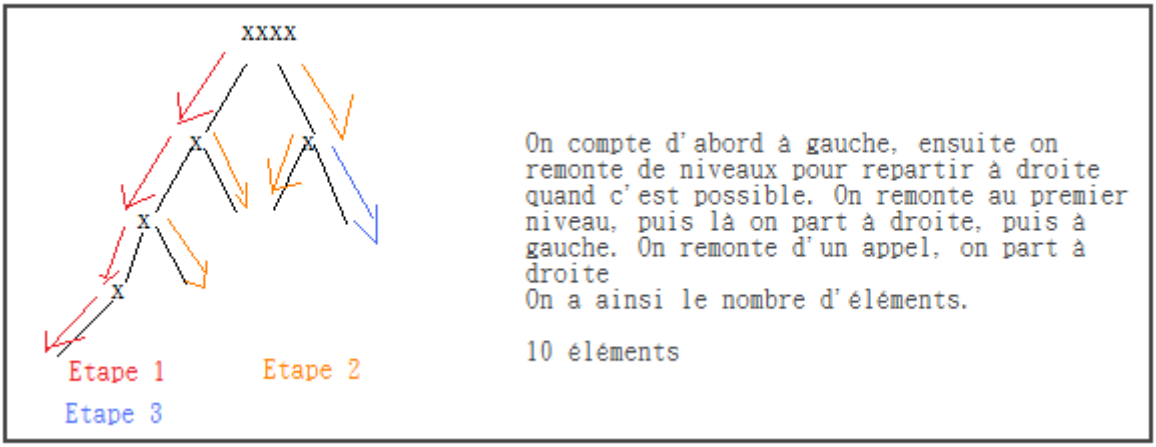
```
// Fonction initialise l'arbre
à NULL en renvoyant "vide"
FONCTION CREER_ARBIN : ARBIN                                    { = vide }
DEBUT
CREER_ARBIN <-- vide                                  // On retourne NULL
FIN

// La fonction ARBIN_VIDE prend comme
paramètre un arbre binaire
FONCTION ARBIN_VIDE(DNM : a : ARBIN) : BOOLEEN    { arbre vide ? }
DEBUT
ARBIN_VIDE <-- (a = vide)
FIN                    // On retourne 1 s'il est = à VIDE (NULL)


// La fonction renvoie un ELEMENT
de la racine
FONCTION r(DNM : a : ARBIN) : ELEMENT                  {lire racine}
DEBUT
SI a = vide
  ALORS FCERR('fonction r()','arbin vide',1)
  SINON r <-- info@a
FIN           // On retourne l'information de la racine de a

// La fonction renvoie le sous-arbre de gauche
FONCTION g(DNM : a : ARBIN) : ARBIN                    {gauche de a}
DEBUT
SI a = vide
  ALORS FCERR('fonction g()','arbin vide',1)
  SINON g <-- psag@a                    // si vide, fonction ERR
FIN                   // On retourne l'arbre de gauche de "a"


// La fonction renvoie le sous-arbre de droite
FONCTION d(DNM : a : ARBIN) : ARBIN                    {droite de a}
DEBUT
SI a = vide
  ALORS FCERR('fonction d()','arbin vide',1)
  SINON d <-- psad@a
FIN                    // retourner le sous-arbre droit de "a"
```

Chapitre 1.4 : Algorithmes supplémentaires

1) Nb_elem :

```
FONCTION nb_elem(arbin) // Récursive
SI arbin_vide(a)
    ALORS return 0
    SINON   nb = 1 + nb_elem(sous-arbre g) + nb_elem(sous-arbre d) ;
```

```
FONCTION PARCOURS_COMPTE(a)
SI NONVIDE(a)                              // Parcours GRD
    nb_elem = PARCOURS_COMPTE(g(a)) //on compte à gauche
    nb_elem = nb_elem + 1          //on compte la racine
    nb_elem = PARCOURS_COMPTE(d(a)) //on compte à droite
FIN SI
Return nb_elem //Retourne le nombre d'éléments
```



2) Maj. champ hauteur

```
SI NONVIDE(ar)
    //La hauteur reprise est la + grande de ses fils pour 1 arbre non vide
    ALORS    ar->hauteur = MAX(HAUTEUR_MAJ(d(ar)), HAUTEUR_MAJ(g(ar)))+1
    //La hauteur est de 0 pour un arbre vide
    SINON     ar->hauteur = 0
FIN SI
Return (ar->hauteur) ;
```

3) GOOD_H

Vérifier si les champs « hauteur » sont bons

```
SI NONVIDE(ar)
    ALORS good_h ← (a->hauteur == MAX(HAUTEUR(g(ar)), HAUTEUR(d(ar)))+1
                    AND GOOD_H(g(ar)) AND GOOD_H(d(ar)))
    //Un arbre vide n'a pas de champ hauteur enregistré
    SINON     good_h ← VRAI
FIN SI
```

4) IS_AVL

Définition de la propriété qui dit si l'arbre est équilibré : Il l'est si la différence entre la hauteur de ses 2 fils est < 2, et que c'est aussi le cas de ses fils, dans le cas d'un arbre vide c'est forcément vrai

```
SI NONVIDE(a)
       ALORS is_avl ← ( | HAUTEUR(g(a)) - HAUTEUR(d(a)) | < 2
                AND IS_AVL(g(a)) AND IS_AVL(d(a)))
       SINON is_avl ← VRAI
FIN SI
//Retourne is_avl (val. booléenne)
```

5) IS_LEX

Dit si l'arbre est lexicographique (tous les plus petits ou égaux à gauche, tous les plus grands à droite). Si vide, il est lexicographique. Sinon tous les éléments à gauche sont les plus petits.

```
min ← moins l'infini
max ← plus l'infini
FONCTION IS_LEX(a, min, max) :BOOL
SI arbin_vide(a)          //Si a est vide > considéré comme lexicographique
    ALORS     is_lex ← VRAI
    SINON   max = a->val                          // le maximum est la racine
          is_lex_g ← is_lex(g(a), min, max)     //on parcoure la gauche
          is_lex_r ← (a->val >= g(a)->val AND a->val < d(a)->val
               AND a->val < max AND a->val >= min) // true si conditions ok
           min ← a->val                          //le minimum est la racine
          is_lex_d ← is_lex(d(a), min, max)     //on parcoure la droite
        Is_lex ← (is_lex_r AND is_lex_g AND is_lex_d)
FIN SI
 //Retourne is_lex (valeur booléenne)
```

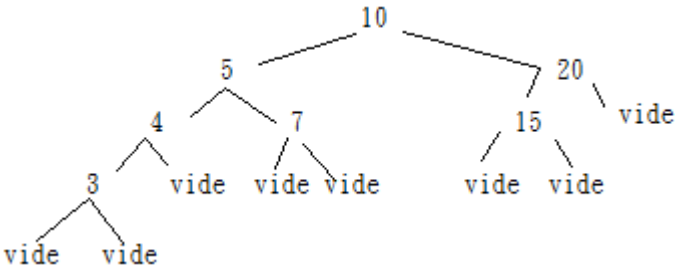
On a ajouté un MIN et un MAX. Au fur et à mesure on met de côté l'un des deux pour isoler la plage de valeurs acceptables.

6) GOOD_AVL

```
min ← moins l'infini
max ← plus l'infini
FONCTION GOOD_AVL(a, min, max) :BOOL
SI arbin_vide(a)      //Si a est vide il peut être considéré comme correct
    ALORS    good_avl ← VRAI
    SINON    max ← a→val                        // le maximum est la racine
          good_avl_g ← good_avl(g(a), min, max) //on parcoure la gauche
          good_avl_r ← (a->val >= g(a)->val AND a->val < d(a)->val
               AND a->val < max AND a->val >= min
               AND |HAUTEUR(g(a)) - HAUTEUR(d(a))|< 2
               AND a->hauteur == MAX(HAUTEUR(g(a)), HAUTEUR(d(a)))+1)
             min ← a->val                          //le minimum est la racine
          good_avl_d ← good_avl(d(a), min, max) //on parcoure la droite
          good_avl ← (good_avl_r AND good_avl_g AND good_avl_d)
FIN SI
//Retourne good_avl (valeur booléenne)
```

7) Arbin_to_file & Arbin_from_file

Permettent de sauvegarder et charger des éléments d'un arbre équilibré en hauteur sur/depuis un disque. OCT = O(n) où n = #ELEMENTS



Fichier [contenu]: 10 5 4 3 vide vide vide 7 vide vide 20 15 vide vide vide

Explications

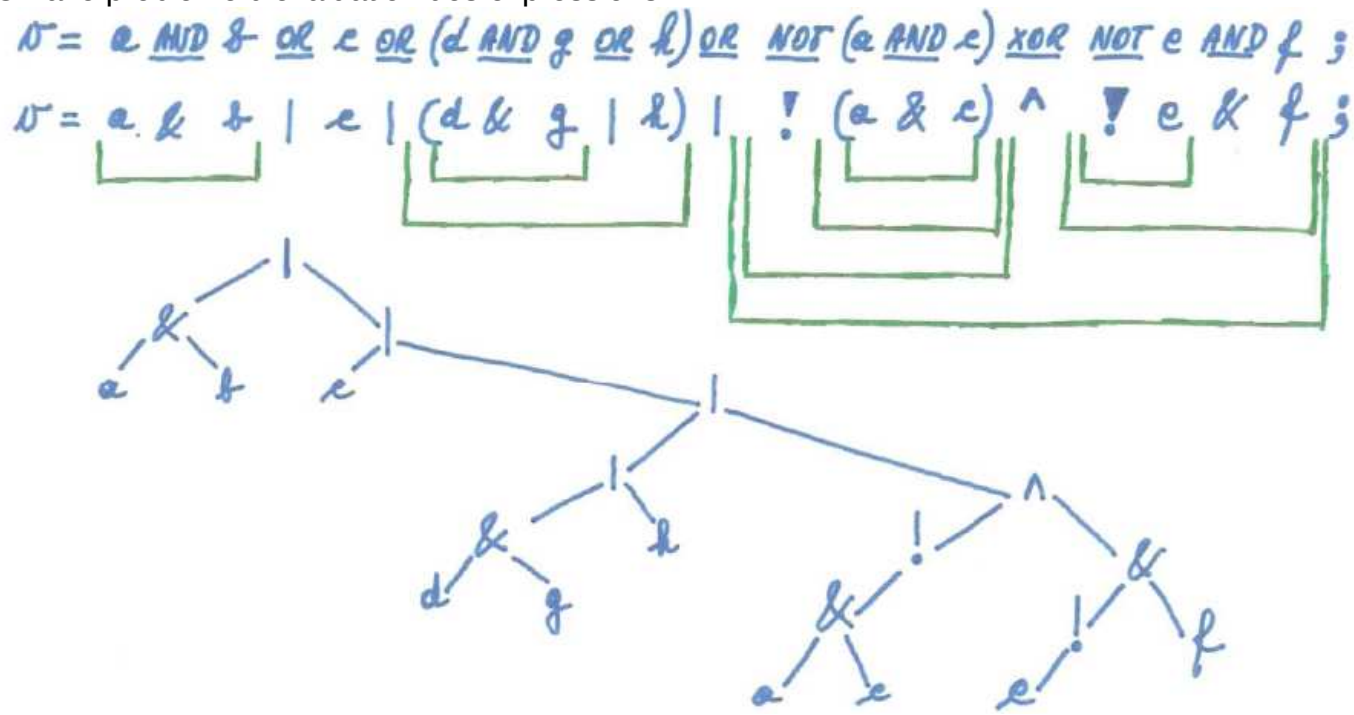
Racine = 10
Sous-arbre gauche = 5
Sous-arbre gauche de 5 = 4
Sous-arbre gauche de 4 = 3
Sous-arbre gauche de 3 = Vide
Sous-arbre droite de 3 = vide
Sous-arbre droite de 4 = vide
…

```
FCT Arbin_to_file[atf](arbin)
SI racine = NULL
     ALORS write(vide)
     SINON     write (racine(a))
               Arbin_to_file(gauche(a))
               Arbin_to_file(droite(a))
FIN SI
```

```
FCT File_to_arbin[fta](f)
S = read(f)
SI s = vide
  ALORS   return NULL ;
  SINON   return
               consarbin(s,fta(f),fta(f),x)
FIN SI
```

Chapitre 2 : Compilateur

On a le problème d'évaluation des expressions.



Le compilateur lit caractère par caractère. On va faire la partie réelle du compilateur : le *scanner !* Il remplace les chaînes de caractère (switch, while, fonctions) pour les reconnaître. Pour les reconnaître, on doit récupérer les variables inventées ! ➔ Classification des identificateurs.

« Table des symboles »
Distinguer les mots réservés des mots de l'utilisateur !

Ecriture simple :
Les 8 premières lettres de l'alphabet sont des identificateurs. Les & sont considérés comme des AND,…

Ce qu'on fait :
- Lire un symbole
- Pouvoir dire ce qu'on fait avec
- Lire le caractère d'après.

NOT est un opérateur unaire, on évalue toujours sa gauche, NOT sur valeur puis renvoyer le résultat. (**bt = binary tree**)

```
FCT eval_bt (bt) : BOOLEAN                    // eval_bt retourne un booléen
IF      bt = empty      THEN  WRITE ("erreur dans eval_bt")
                              exit()
else IF v(bt) = identificator THEN return (value(racine(bt)))
else IF v(bt) = '!'           THEN return (NOT eval_bt(left(bt)))
else IF v(bt) = '&'           THEN return(eval_bt(left(bt)) AND eval_bt(right(bt)))
else IF v(bt) = '^'           THEN return(eval_bt(left(bt)) XOR eval_bt(right(bt)))
else IF v(bt) = '|'           THEN return(eval_bt(left(bt)) OR eval_bt(right(bt)))
else                          WRITE ("identificateur non-attendu")
                              exit();
```

```
expr  △ terme "|"  terme ...
                          Une expression peut être décomposée en termes séparés
            ou            par | ou un terme séparé par | d'une expression
        terme ["|" expr ]...
```

```
FONCTION expr_to_ar (symbole) : ARBIN
   Arbre = term_to_ar (symbole)
       SI symbole != "|"
           ALORS return (arbre)
           SINON   symbole = getnextSymbol():
               return (consarbin("|",arbre,expr_to_ar(symbole));
```

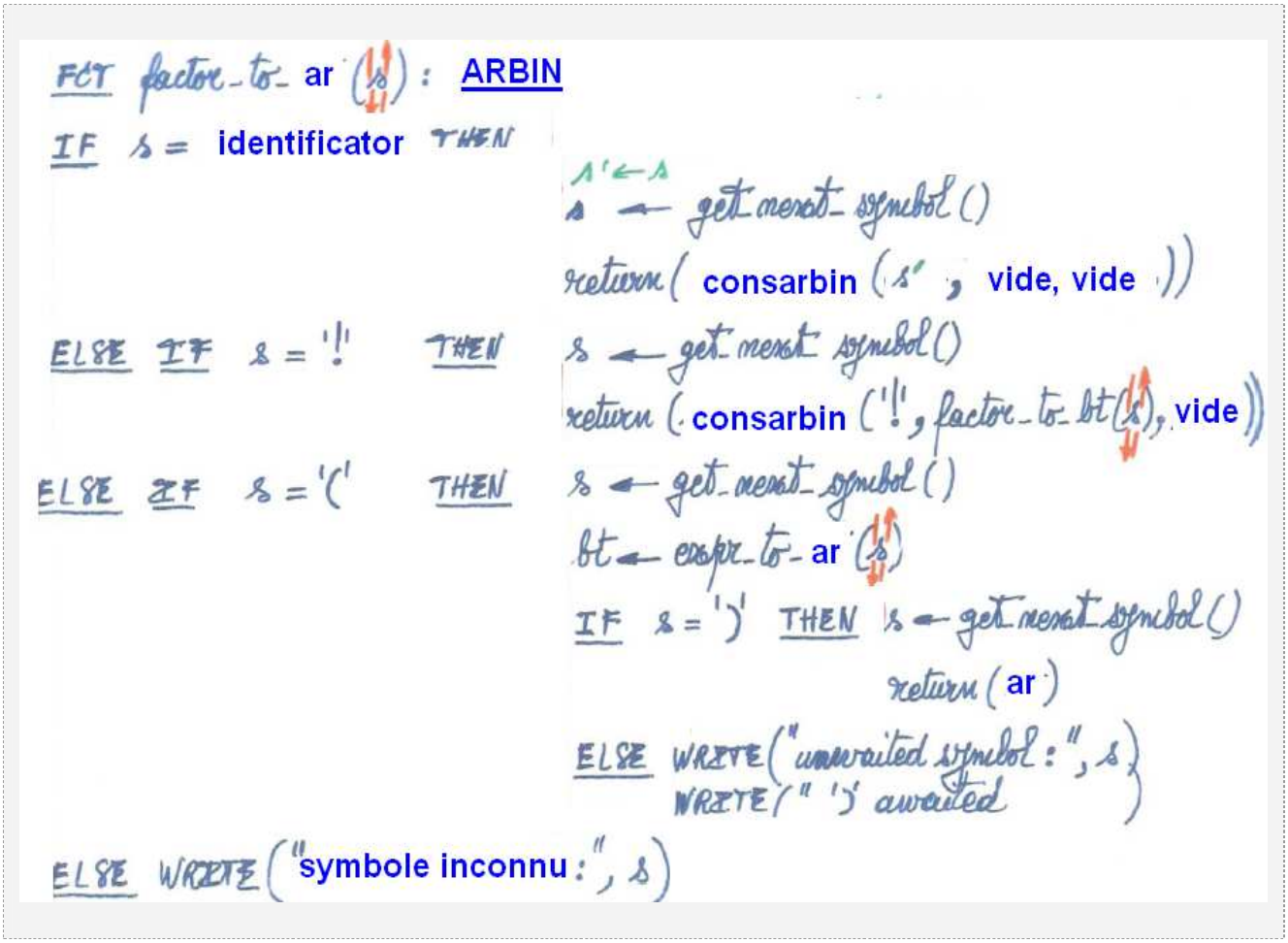
```
terme △ alter "^" alter "^"
        ou
terme △ alter ["^" terme]
```

```
FONCTION terme_to_ar (symbole) : ARBIN
   Arbre = alter_to_ar (symbole)
       SI symbole != "^"
           ALORS return (arbre)
           SINON   symbole = getnextSymbol():
               return (consarbin("^",arbre,terme_to_ar(symbole));
```

alter △ factor "&" factor "&" factor
ou alors
alter △ factor ["&" alter]

```
FONCTION alter_to_ar (symbole) : ARBIN
   Arbre = factor_to_ar (symbole)
       SI symbole != “&”
           ALORS return (arbre)
           SINON   symbole = getnextSymbol():
               return (consarbin(“&”,arbre,alter_to_ar(symbole));
```

factor △ id "!" factor | "(" expr ")"



Compilateur

1) Etendre les expressions booléennes aux expressions arithmétiques

Expr_rel Δ expr_arit op_rel expr_arit …
Expr_arit Δ id_arit op_arit id_arit …
Liste des opérateurs arithmétiques :>, <, ≥, ≤, ≤, =
*Liste des opérateurs relationnels : +, -, *, /, -, (,)*
(comme dans les autres exemples du cours, S envoie le symbole à la fonction et renvoie le suivant)

```
FCT expr_rel_to_ar(S) : ARBIN
ar ← expr_arit_to_ar(S)
SI S != '>' AND S != '<' AND S !=' ≥' AND S!=' ≤' AND S!='='
    ALORS renvoyer(ar)
    SINON op_rel ← S
          S ← GetNextSympbol()
          renvoyer(consarbin(op_rel, ar, expr_rel_to_ar(S))
FIN SI

FCT expr_arit_to_ar(S) : ARBIN
ar ← factor_arit_to_ar(S)
SI S != '-' AND S != '+'
    ALORS renvoyer(ar)
    SINON op_ari ← S
          S ← GetNextSympbol()
          renvoyer(consarbin(op_ari, ar, expr_arit_to_ar(S))
FIN SI

FCT factor_arit_to_ar(S) : ARBIN
ar ← terme_arit_to_ar(S)
SI S != '*' AND S != '/'
    ALORS renvoyer(ar)
    SINON op_ari ← S
          S ← GetNextSympbol()
          renvoyer(consarbin(ap_ari, ar, factor_arit_to_ar(S))
FIN SI

FCT terme_arit_to_ar(S) : ARBIN
SI S = id_arit
ALORS       S' ← S
            S ← GetNextSymbol()
            Return(consarbin(S', NULL,NULL))
SINON SI S = '-' ALORS S ← GetNextSymbol()
                       Return(consarbin('-', terme_arit_to_ar(S),NULL)
SINON SI S = '(' ALORS S ← GetNextSymbol()
                 ar ← exprreltoar(S)
                 SI S=')'     ALORS S ← GetNextSymbol()
                              Return(ar)
                               SINON ECRIRE("Erreur, ) manque")
SINON ECRIRE (« Erreur, symbole inconnu… », S)
FIN SI
```

1') Créer un mix des expressions booléennes et expressions arithmétiques comme en C

2) Réécrire littéralement l'expression à partir de l'arbre binaire

```
AFFICHER_EXPRESSION(a)
SI NONVIDE(a)
   ALORS    AFFICHER_EXPRESSION(g(a)) //on descend dans le sous-arbre de
gauche
        Afficher r(a) //on affiche la racine
        //si on veut on peut aussi vérifier la valeur et afficher « OR »,
« AND », etc
        AFFICHER_EXPRESSION(d(a)) //on descend dans le sous-arbre de droite
   SINON // rien
FIN SI
```

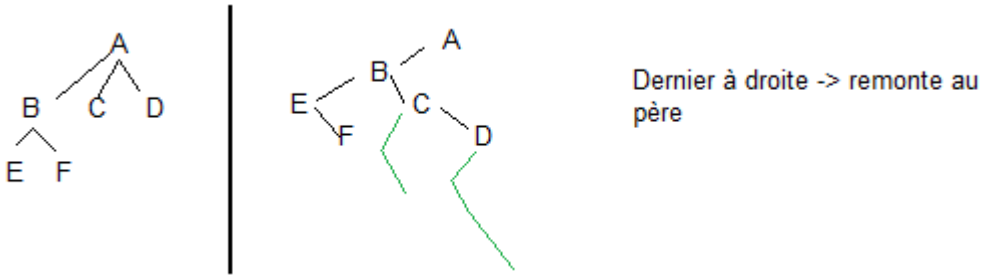
Chapitre 3 : Théorie des graphes

Discipline mathématique

```
          Algos/Graphes
          ----------------        =       recherche opérationnelle
               Simplex
```

Arbin enfilés

Arbres généalogiques en informatique



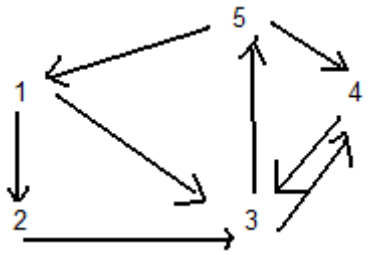
Cycles ➔ graphes

On peut avoir :

- Des états capteurs
- Des entrées sorties multiples
- Des étiquettes (ex : carte routière maritime aérienne, réseau électrique)

OCT

Avec les graphes on arrive facilement à des OCT de $O(n^4)$ ou $O(n^5)$. On parle d'algorithmes **POLYNOMIAUX**. Les puissances de n sont celles qu'on trouve dans les polynômes
MAIS si n = 1000 alors 1000^5 = 1000 Téra opérations !



Matrice	1	2	3	4	5
1	0	1	1	0	0
2	0	0	1	0	0
3	0	0	0	1	1
4	0	0	1	0	0
5	1	0	0	1	0

➔ Existe-t-il un chemin de longueur L = 1 de i à j
➔ :A[i,j] est vrai ou faux ?

➔ Existe-t-il un chemin de L = ~~2~~ (3) de i à j ?
OUI s'il existe une valeur telle que
A[i,k] et A[k,~~j~~l] = VRAI (et A[l, j] = VRAI

```
EXISTE ← FAUX
POUR k = 1 -> n PPD +1 FAIRE
    EXISTE ← existe OU A[i,k] ET A[k,j]
```

➔ Existe-t-il un chemin de longueur entre i et j ?
Regarder le chemin de toutes les longueurs.

? Somme des OCT : $O(n^{n-1})$ | $O(n^{n+1})$

Algorithme de Warshall

Dit s'il y a un chemin pour OCT = $O(n^3)$. Algorithme utilisable pour départager les couples de points accessibles ou non (s'il y a un chemin de 999 il le trouve en $O(n^3)$).
Algorithme très court !

1) Fictivité

Exponentiation indienne ➔ Si on doit calculer A^25, on décompose l'exposant en ses puissances de 2 et on retient les + élevées.
➔ A^8 A^16 A^1 = A^25

Comment calculer A^16 dans ce cas ?...
On a 6 multiplications de matrices pour arriver à A^16.
⇨ 6 * $O(n^3)$

2) Warshall

```
PROC WARSHALL(DNM A[1….n 1-n], RES F[1…n 1….n] copie de A)
POUR k = 1 → n          1          2

      POUR i = 1 → n

            SI F[i,k] ALORS          1'          2'
                  // De i vers k ? si chemin de i à 1, longueur = 1
                       POUR j = 1 → n

                                    F [i,j]← F[i,j] OU F[k,j]     1''     2''
                                    // On ne perd pas le résultat d'avant dans
la matrice et on fait un ou booléen. Si de 1 à j, il y a un chemin de longueur
1, on l'inclut.
1'''  2'''
```