Livret - 8

Arbres binaires

Arbre, TAD d'arbre binaire, parcours.

Outil utilisé : classes génériques C#

RM di scala

Cours informatique programmation

Rm di Scala - http://www.discala.net

8 : Structures d'arbres binaires

Plan du chapitre:

1. Notions générales sur les arbres

1.1 Vocabulaire employé sur les arbres :

- Etiquette, racine, noeud, branche, feuille
- Hauteur, profondeur ou niveau d'un noeud
- Chemin d'un noeud, noeuds frères, parents, enfants, ancêtres
- Degré d'un noeud
- Hauteur ou profondeur d'un arbre
- Degré d'un arbre
- Taille d'un arbre

1.2 Exemples et implémentation d'arbre

- Arbre de dérivation
- Arbre abstrait
- Arbre lexicographique
- Arbre d'héritage
- Arbre de recherche

2. Arbres binaires

2.1 TAD d'arbre binaire
2.2 Exemples et implémentation d'arbre

- tableau statique
- variable dynamique
- classe

2.3 Arbres binaires de recherche
2.4 Arbres binaires partiellement ordonnés (tas)
2.5 Parcours en largeur et profondeur d'un arbre binaire

- Parcours d'un arbre
- Parcours en largeur
- Parcours préfixé
- Parcours postfixé
- Parcours infixé
- Illustration d'un parcours en profondeur complet
- Exercice

1. Notions générales sur les structures d'arbres

La structure d'arbre est très utilisée en informatique. Sur le fond on peut considérer un arbre comme une généralisation d'une liste car les listes peuvent être représentées par des arbres. La complexité des algorithmes d'insertion de suppression ou de recherche est généralement plus faible que dans le cas des listes (cas particulier des arbres équilibrés). Les mathématiciens voient les arbres eux-même comme des cas particuliers de graphes non orientés connexes et acycliques, donc contenant des sommets et des arcs :

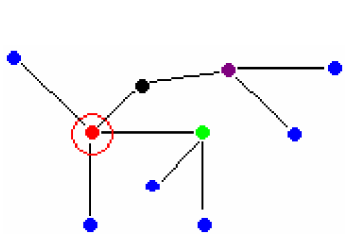


fig-1

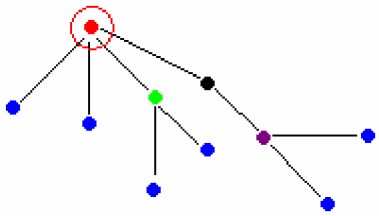


fig-2

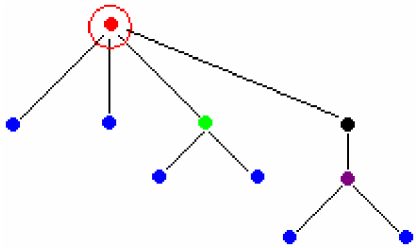


fig-3

Ci dessus 3 représentations graphiques de la même structure d'arbre : dans la figure fig-1 tous les sommets ont une disposition équivalente, dans la figure fig-2 et dans la figure fig-3 le sommet "**cerclé**" se distingue des autres.

Lorsqu'un sommet est distingué par rapport aux autres, on le dénomme **racine** et la même structure d'arbre s'appelle une **arborescence**, par abus de langage dans tout le reste du document nous utliserons le vocable **arbre** pour une **arborescence**.

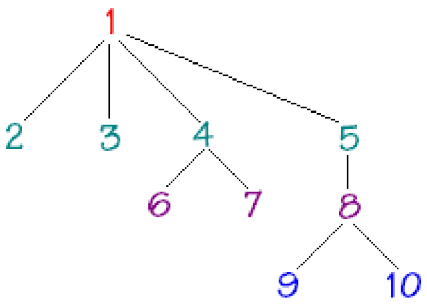
Enfin certains arbres particuliers nommés arbres binaires sont les plus utilisés en informatique et les plus simples à étudier. En outre il est toujours possible de "**binariser**" un arbre non binaire, ce qui nous permettra dans ce chapitre de n'étudier que les structures d'arbres binaires.

1.1 Vocabulaire employé sur les arbres

Etiquette

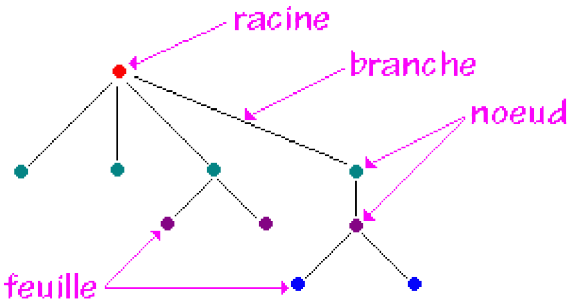
Un arbre dont tous les noeuds sont nommés est dit **étiqueté**. L'étiquette (ou nom du sommet) représente la "valeur" du noeud ou bien l'information associée au noeud.
Ci-dessous un arbre étiqueté dans les entiers entre 1 et 10 :

Ci-dessous un arbre étiqueté dans les entiers entre 1 et 10 :



Nous rappellons la terminologie de base sur les arbres:

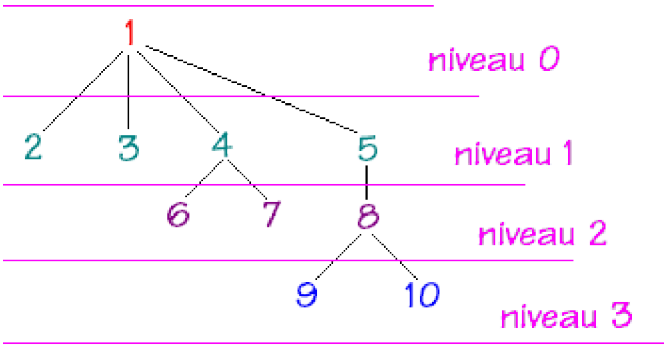
Racine , noeud, branche , feuille



Hauteur d'un noeud

Nous conviendrons de définir la **hauteur** (ou **profondeur** ou **niveau d'un noeud**) d'un noeud X comme égale **au nombre de noeuds à partir de la racine** pour aller jusqu'au noeud X.

En reprenant l'arbre précédant et en notant **h** la fonction hauteur d'un noeud :



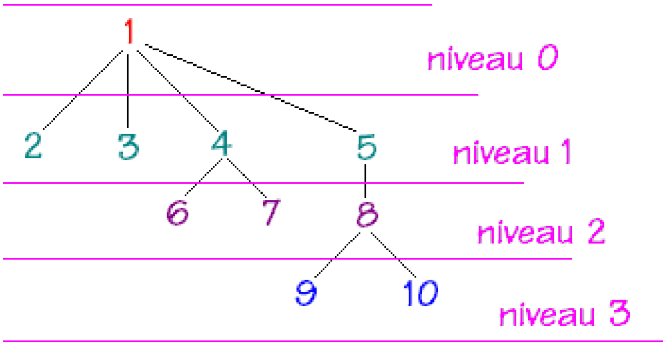
Pour atteindre le noeud étiqueté 9 , il faut parcourir le lien 1--5, puis 5--8, puis enfin 8--9 soient 4 noeuds donc 9 est de profondeur ou de hauteur égale à 4, soit **h(9) = 4**.
Pour atteindre le noeud étiqueté 7 , il faut parcourir le lien 1--4, et enfin 4--7, donc 7 est de profondeur ou de hauteur égale à 3, soit **h(7) = 3**.

Par définition la hauteur de la racine est égal à 1.
h(racine) =1 (*pour tout arbre non vide)*

(*Certains auteurs adoptent une autre convention pour calculer la hauteur d'un noeud: la racine a pour hauteur 0 et donc n'est pa comptée dans le nombre de noeuds, ce qui donne une hauteur inférieure d'une unité à notre définition*).

Chemin d'un noeud

On appelle chemin du noeud X la **suite des noeuds** par lesquels il faut passer pour aller de la racine vers le noeud X.



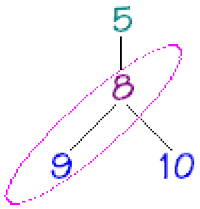
Chemin du noeud 10 = (1,5,8,10)
Chemin du noeud 9 = (1,5,8,9)
.....
Chemin du noeud 7 = (1,4,7)
Chemin du noeud 5 = (1,5)
Chemin du noeud 1 = (1)

Remarquons que la hauteur h d'un noeud X est égale au nombre de noeuds dans le chemin :

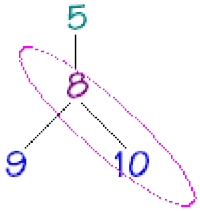
h(X) = NbrNoeud(Chemin(X)).

Le vocabulaire de lien entre noeuds de niveau différents et reliés entres eux est emprunté à la généalogie :

Parents, enfants

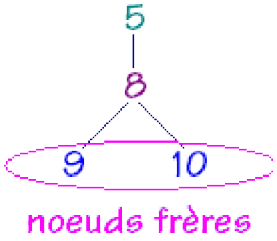


9 est l'enfant de 8
8 est le parent de 9



10 est l'enfant de 8
10 est le parent de 8

Noeuds frères, ancêtres



- **9 et 10 sont des frères**
- **5 est le parent de 8 et l'ancêtre de 9 et 10.**

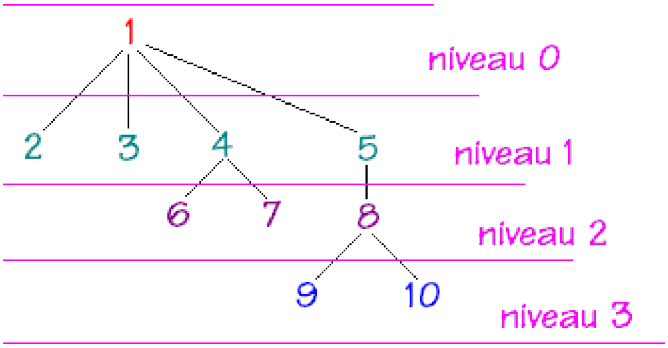
On parle aussi d'ascendant, de descendant ou de fils pour évoquer des relations entres les noeuds d'un même arbre reliés entre eux.

Nous pouvons définir récursivement la hauteur h d'un noeud X à partir de celle de son parent :

Hauteur d'un noeud (récursif)

h (racine) = 1;
h (X) = 1+ h (parent (X))

Reprenons l'arbre précédent en exemple :

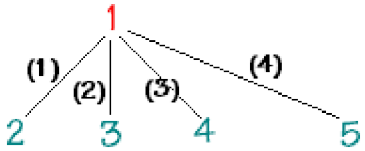


Calculons récursivement la hauteur du noeud 9, notée h(9) :
h(9) = 1+h(8)
h(8) = 1+h(5)
h(5) = 1+h(1)
h(1) = 1 = h(5)=2 = h(8)=3 = h(9)=4

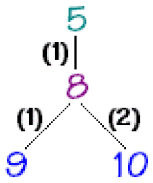
Degré d'un noeud

Par définition le **degré** d'un noeud est égal au **nombre de ses descendants** (enfants).

Soient les deux exemples ci-dessous extraits de l'arbre précédent :



Le noeud 1 est de degré 4, car il a 4 enfants



Le noeud 5 n'ayant qu'un enfant son degré est 1.
Le noeud 8 est de degré 2 car il a 2 enfants.

Remarquons
que lorsqu'un arbre a **tous ses noeuds de degré 1**, on le nomme **arbre dégénéré** et que c'est en fait une **liste**.

Hauteur d'un arbre

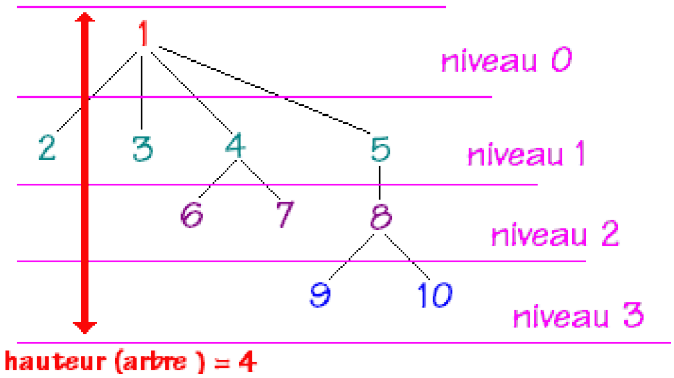
Par définition c'est le **nombre de noeuds du chemin le plus long** dans l'arbre.; on dit aussi profondeur de l'arbre.

La hauteur **h** d'un arbre correspond donc au nombre maximum de niveaux :

Hauteur d'un arbre

h (Arbre) = max { h (X) / ∀ X, X noeud de Arbre }
si Arbre = ∅ alors h(Arbre) = 0

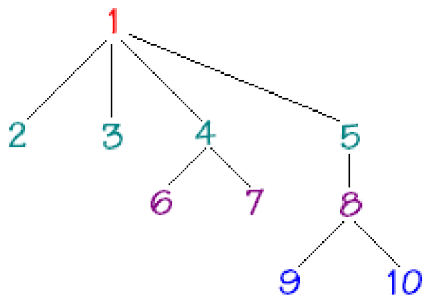
La hauteur de l'arbre ci-dessous :



Degré d'un arbre

Le degré d'un arbre est égal au **plus grand des degrés de ses nœuds :**
d°(Arbre) = max { d°(X) / ∀ X, X noeud de Arbre }

Soit à répertorier dans l'arbre ci-dessous le degré de chacun des noeuds :



d°(1) = 4 d°(2) = 0
d°(3) = 0 d°(4) = 2
d°(5) = 1 d°(6) = 0
d°(7) = 0 d°(8) = 2
d°(9) = 0 d°(10) = 0

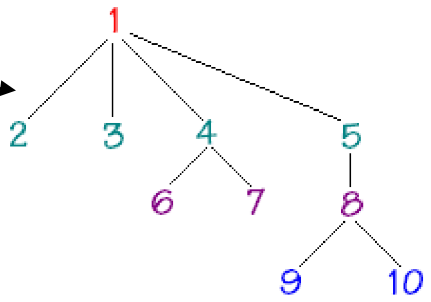
La valeur maximale est 4 , donc cet arbre est de degré 4.

Taille d'un arbre

On appelle **taille** d'un arbre le nombre total de noeuds de cet arbre :

taille(< r , fg , fd >) = 1 + taille(fg) + taille(fd)

L'arbre ci-contre a pour **taille** 10 (car il a 10 noeuds)



1.2 Exemples et implémentation d'arbre

Les structures de données arborescentes permettent de représenter de nombreux problèmes, nous proposons ci-après quelques exemples d'utilisations d'arbres dans des contextes différents.

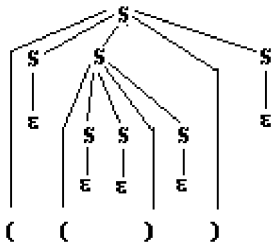
Arbre de dérivation d'un mot dans une grammaire

Exemple - 1
arbre d'analyse

Soit la grammaire G_2 : VN = {S}
V_T = {(,)}
Axiome : S
Règles
1 : S → (SS)S
2 : S → ε

Le langage $L(G_2)$ se dénomme langage des parenthèses bien formées.

Soit le mot (( )) de G_2 , voici un arbre de dérivation de (( )) dans G_2 :



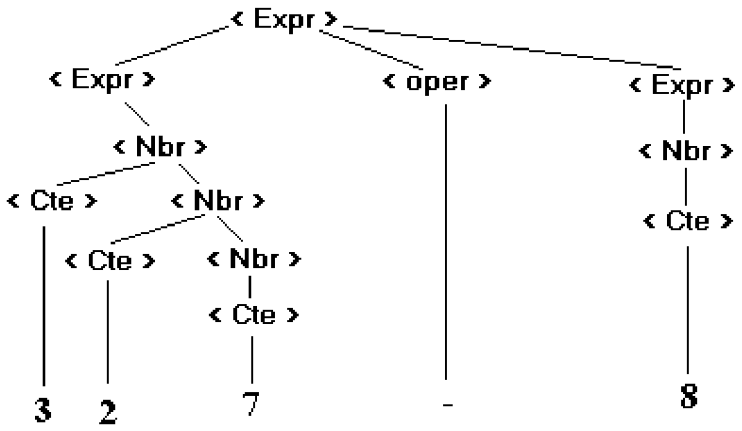
Exemple - 2
arbre abstrait

Soit la grammaire Gexp :

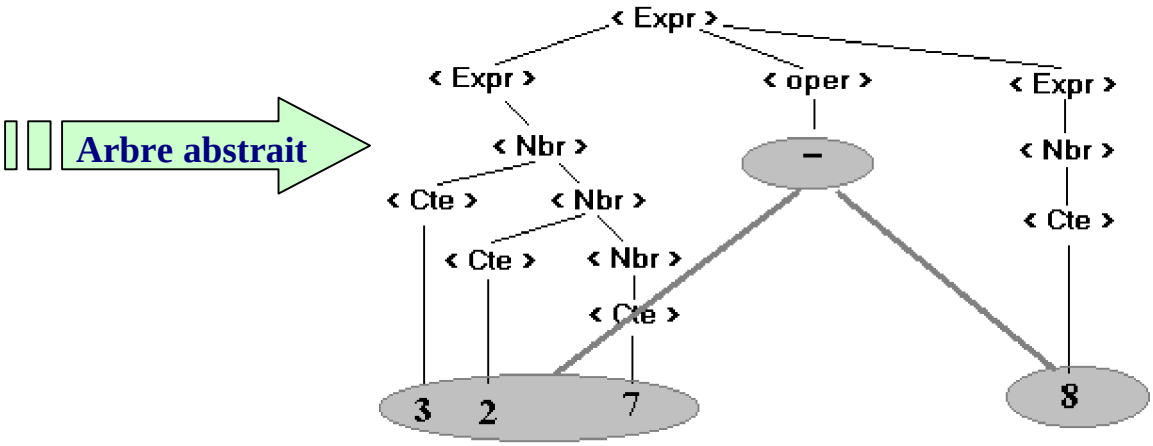
G_{exp} = (V_N,V_T,Axiome,Règles)
V_T = { **0**, ..., **9**, **+**,**-**, **/**, *****, **)**, **(**}
V_N = { ⟨ Expr ⟩, ⟨ Nbr ⟩, ⟨ Cte ⟩, ⟨ Oper ⟩}
Axiome : ⟨ Expr ⟩
Règles :
1 :⟨ Expr ⟩→⟨ Nbr ⟩ | **(**⟨Expr ⟩**)**| ⟨ Expr ⟩⟨ Oper ⟩⟨ Expr ⟩
2 :⟨ Nbr ⟩→⟨ Cte ⟩ | ⟨ Cte ⟩⟨ Nbr ⟩
3 :⟨ Cte ⟩→**0** | **1** |...| **9**
4 :⟨ Oper ⟩→**+** | **-** | ***** | **/**

soit : **327 - 8** un mot de $L(G_{exp})$

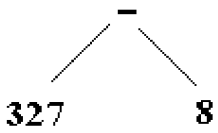
Soit son arbre de dérivation dans G_{exp} :



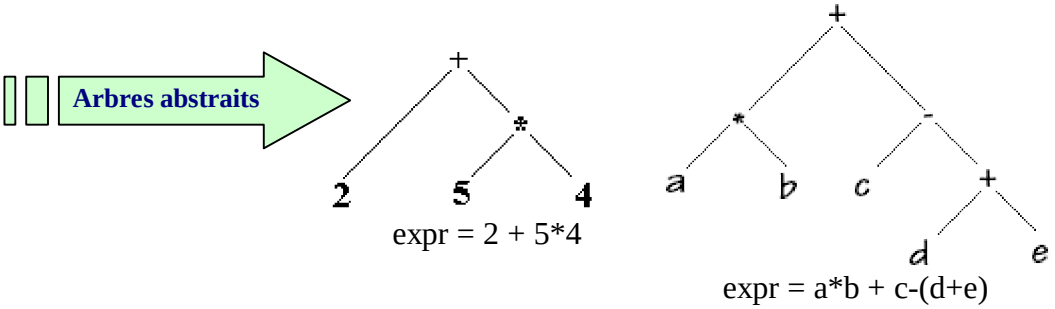
L'arbre obtenu ci-dessous en grisé à partir de l'arbre de dérivation s'appelle l'arbre abstrait du mot " 327-8 " :



On note ainsi cet arbre abstrait :

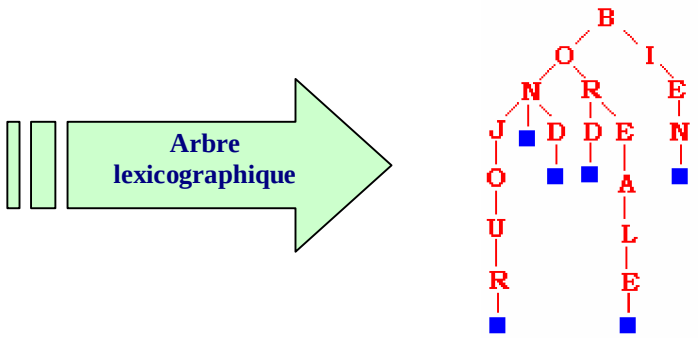


Voici d'autres abres abstraits d'expressions arithmétiques :

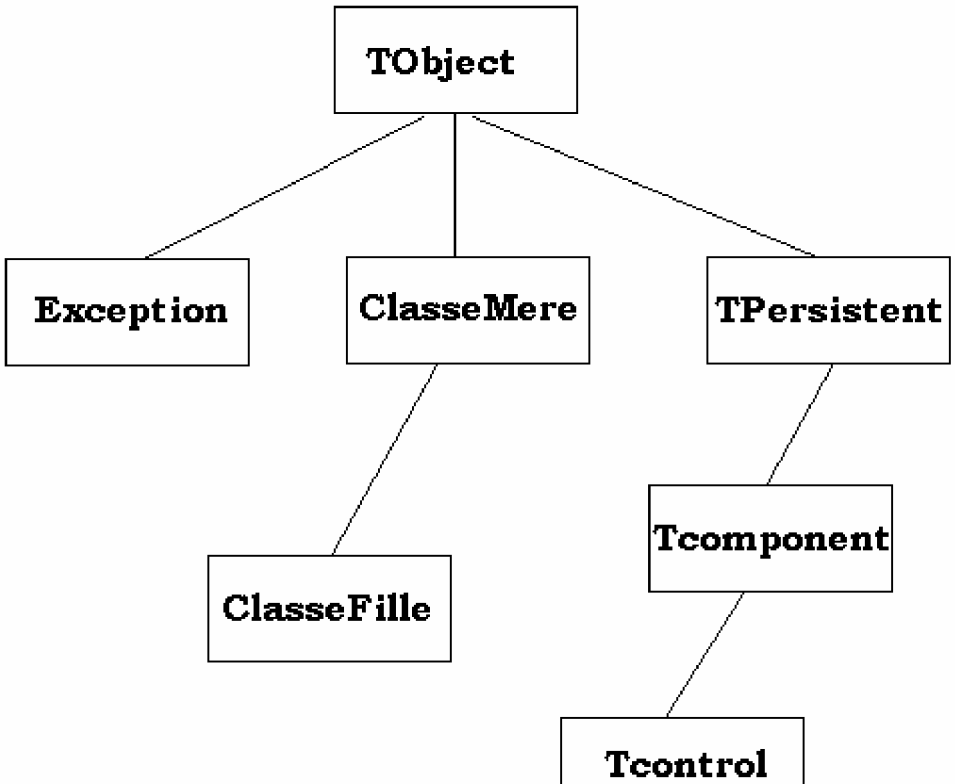


Arbre lexicographique

Rangement de mots par ordre lexical (alphabétique)
Soient les mots BON, BONJOUR, BORD, BOND, BOREALE, BIEN, il est possible de les ranger ainsi dans une structure d'arbre :



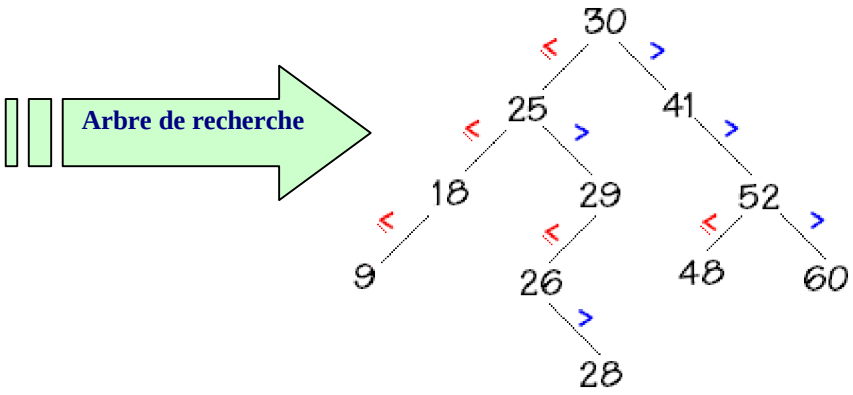
Arbre d'héritage en Delphi



Arbre de recherche

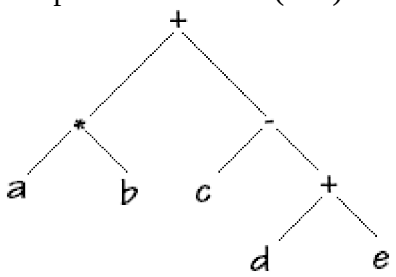
Voici à titre d'exemple que nous étudierons plus loin en détail, un arbre dont les noeuds sont de degré 2 au plus et qui est tel que pour chaque noeud la valeur de son enfant de gauche lui est inférieure ou égale, la valeur de son enfant de droite lui est strictement supérieure.

Ci-après un tel arbre ayant comme racine 30 et stockant des entiers selon cette répartition :

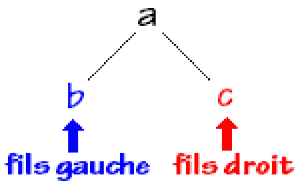


2 Les arbres binaires

Un arbre **binaire** est un arbre de degré 2 (dont les noeuds sont de degré 2 au plus).
L'arbre abstrait de l'expression a*b + c-(d+e) est un arbre binaire :



Vocabulaire :
Les descendants (enfants) d'un noeud sont lus de gauche à droite et sont appelés respectivement **fils gauche** (descendant gauche) et **fils droit** (descendant droit) de ce noeud.



Les arbres binaires sont utilisés dans de très nombreuses activités informatiques et comme nous l'avons déjà signalé il est toujours possible de reprrésenter un arbre général (de degré 2) par un arbre binaire en opérant une "binarisation".

Nous allons donc étudier dans la suite, le comportement de cette structure de donnée récursive.

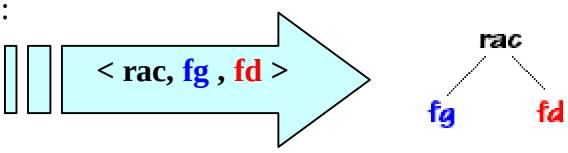
2.1 TAD d'arbre binaire

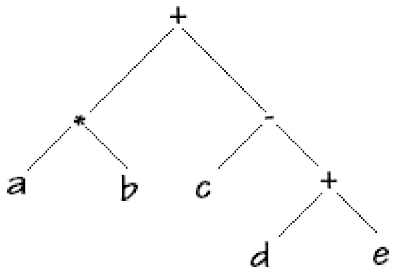
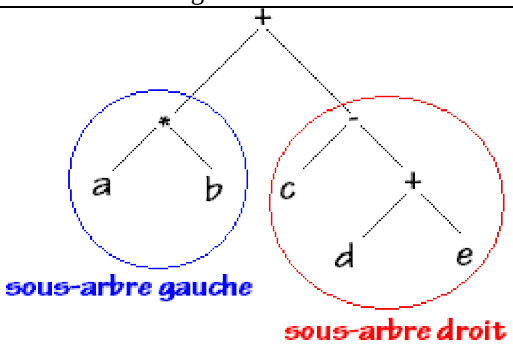
Afin d'assurer une cohérence avec les autres structures de données déjà vues (**liste, pile, file**) nous proposons de décrire une abstraction d'un arbre binaire avec un TAD. Soit la signature du TAD d'arbre binaire :

```
TAD ArbreBin
utilise : T0, Noeud, Booleens
opérations :
   ∅   : → ArbreBin
   Racine : ArbreBin → Noeud
   filsG : ArbreBin → ArbreBin
   filsD : ArbreBin → ArbreBin
   Constr : Noeud x ArbreBin x ArbreBin → ArbreBin
   Est_Vide  : ArbreBin → Booleens
   Info : Noeud → T0

préconditions :

   Racine(Arb) def_ssi Arb ≠ ∅
   filsG(Arb)  def_ssi Arb ≠ ∅
   filsD(Arb)  def_ssi Arb ≠ ∅
axiomes :

    ∀rac ∈ Noeud ,  ∀fg ∈ ArbreBin , ∀fd ∈ ArbreBin
    Racine(Constr(rac,fg,fd)) = rac
    filsG(Constr(rac,fg,fd))  = fg
    filsD(Constr(rac,fg,fd))  = fd
    Info(rac) ∈T0

FinTAD- ArbreBin
```

- T_0 est le type des données rangées dans l'arbre.
- L'opérateur **filsG( ) renvoie le sous-arbre gauche** de l'arbre binaire, l'opérateur **filsD( ) renvoie le sous-arbre droit** de l'arbre binaire, l'opérateur Info( ) permet de stocker des informations de type T_0 dans chaque noeud de l'arbre binaire.

Nous noterons **< rac, fg , fd >** avec conventions implicites un arbre binaire dessiné ci-dessous :



Exemple, soit l'arbre binaire A :	Les sous-arbres gauche et droit de l'arbre A :
A = 	

filsG(A) = < * , a , b >
filsD(A) = < - , c , < + , d , e > >

2.2 Exemples et implémentation d'arbre binaire étiqueté

Nous proposons de représenter un **arbre binaire étiqueté** selon deux spécifications différentes classiques :

1°) Une implantation fondée sur une structure de tableau en allocation de mémoire statique, nécessitant de connaître au préalable le nombre maximal de noeuds de l'arbre (ou encore sa taille).

2°) Une implantation fondée sur une structure d'allocation de mémoire dynamique implémentée soit par des pointeurs (variables dynamiques) soit par des références (objets) .

Implantation dans un tableau statique

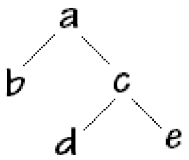
Spécification concrète
Un noeud est une structure statique contenant 3 éléments :

- l'information du noeud
- le fils gauche
- le fils droit

Pour un arbre binaire de taille = n, **chaque noeud de l'arbre binaire est stocké dans une cellule d'un tableau** de dimension 1 à n cellules. Donc chaque noeud est repéré dans le tableau par un indice (celui de la cellule le contenant).
Le champ fils gauche du noeud sera l'indice de la cellule contenant le descendant gauche, et le champ fils droit vaudra l'indice de la cellule contenant le descendant droit.

Exemple

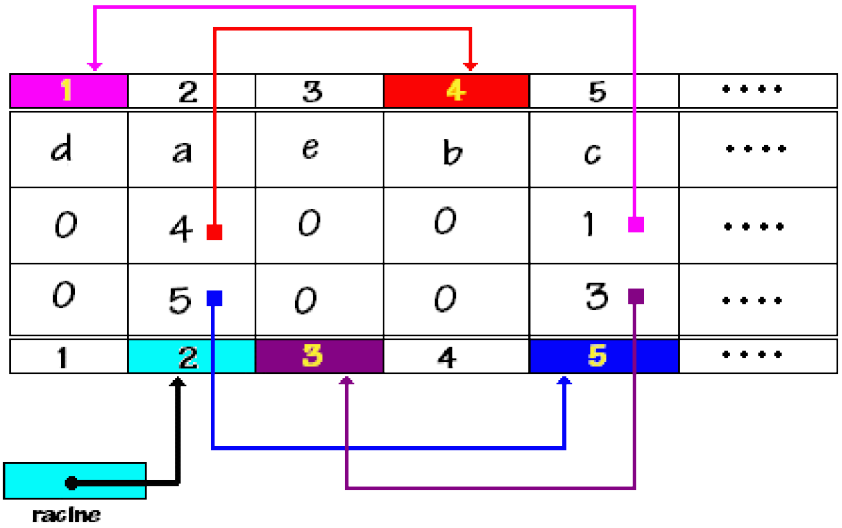
Soit l'arbre binaire ci-contre :



Selon l'implantation choisie, par hypothèse de départ, la racine <a, vers b, vers c >est contenue dans la cellule d'indice 2 du tableau, les autres noeuds sont supposés être rangés dans les cellules 1, 3,4,5 :

Nous avons :

racine = table[2]
table[1] = < d , 0 , 0 >
table[2] = < a , **4** , **5** >
table[3] = < e , 0 , 0 >
table[4] = < b , 0 , 0 >
table[5] = < c , **1** , **3** >



Explications :

table[2] = < a , **4** , **5** >signifie que le fils gauche de ce noeud est dans table[**4**] et son fils droit dans table[**5**]
table[5] = < c , **1** , **3** >signifie que le fils gauche de ce noeud est dans table[**1**] et son fils droit dans table[**3**]
table[1] = < d , 0 , 0 > signifie que ce noeud est une feuille
...etc

Implantation dynamique avec une classe

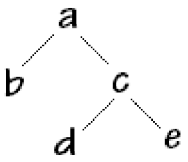
Spécification concrète

Le noeud est un objet contenant 3 éléments dont 2 sont eux-mêmes des objets de noeud :

- l'information du noeud
- une référence vers le fils gauche
- une référence vers le fils droit

Exemple

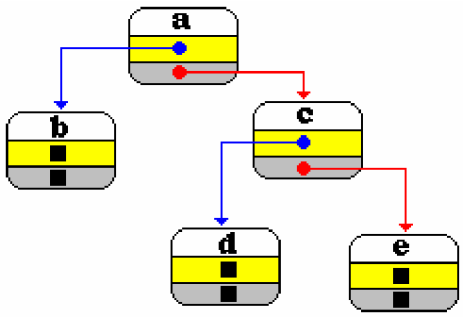
Soit l'arbre binaire ci-contre :



Selon l'implantation choisie, par hypothèse de départ, le premier objet appelé racine de l'arbre est **< a, ref vers b, ref vers c >**

Nous avons :

racine → < a, ref vers b, ref vers c >
ref vers b → < b, null, null >
ref vers c → < a, ref vers d, ref vers e >
ref vers d → < d, null, null >
ref vers e → < e, null, null >



Spécification d'implantation en C#

Nous livrons ci-dessous une écriture de la signature et l'implémentation minimale d'une classe d'arbre binaire nommée ArbreBin en C# :

```
interface IArbreBin<T0>
{
   T0 Info {    get;      set;       }
}
class ArbreBin<T0> : IArbreBin<T0>
{
   private T0 InfoLoc;
   private ArbreBin<T0> fg;
```

```
    private ArbreBin<T0> fd;

    public ArbreBin(T0 s) : this ( )       {
       InfoLoc = s;
    }
    public ArbreBin ( )
    {
       InfoLoc = default(T0);
       fg = default(ArbreBin<T0>);
       fd = default(ArbreBin<T0>);
    }
    public T0 Info
    {
       get   {  return InfoLoc;   }
       set    {  InfoLoc = value; }
    }

    public ArbreBin<T0> filsG
    {
       get   {  return this.fg;  }
       set   {  fg = new ArbreBin<T0>(value.Info); }
    }
    public ArbreBin<T0> filsD
    {
       get  { return this.fd;  }
       set  {  fd = new ArbreBin<T0>(value.Info); }
    }
}
```

2.3 Arbres binaires de recherche

- Nous avons étudié prédédement des algorithmes de recherche en table, en particulier la recherche dichotomique dans une table triée dont la recherche s'effectue en **O(log(n))** comparaisons.
- Toutefois lorsque le nombre des éléments varie (ajout ou suppression) ces ajouts ou suppressions peuvent nécessiter des temps en **O(n)**.
- En utilisant une liste chaînée qui approche bien la structure dynamique (plus gourmande en mémoire qu'un tableau) on aura en moyenne des temps de suppression ou de recherche au pire de l'ordre de **O(n)**. L'ajout en fin de liste ou en début de liste demandant un temps constant noté **O(1)**.

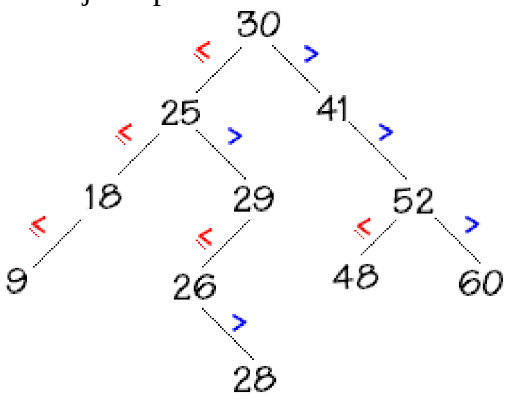
Les arbres binaires de recherche sont un bon compromis pour un temps **équilibré entre ajout, suppression et recherch**e.

Un arbre binaire de recherche satisfait aux critères suivants :

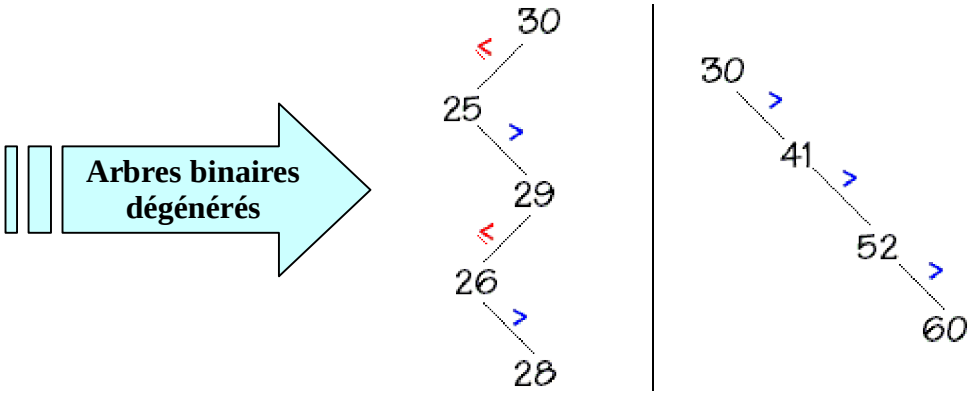
- L'ensemble des étiquettes est **totalement ordonné**.
- Une étiquette est dénommée **clef**.
- Les **clefs** de tous les noeuds du sous-arbre **gauche** d'un noeud X, sont **inférieures ou égales** à la clef de X.
- Les **clefs** de tous les noeuds du sous-arbre **droit** d'un noeud X, sont

supérieures à la clef de X.

Nous avons déjà vu plus haut un arbre binaire de recherche :



Prenons par exemple le noeud (25) son sous-arbre droit est bien composé de noeuds dont les clefs sont supérieures à 25 : (29,26,28). Le sous-arbre gauche du noeud (25) est bien composé de noeuds dont les clefs sont inférieures à 25 : (18,9).
On appelle arbre binaire dégénéré un arbre binaire dont le degré = 1, ci-dessous 2 arbres binaires de recherche dégénérés :



Nous remarquons dans les deux cas que nous avons affaire à une liste chaînée donc le nombre d'opération pour la suppression ou la recherche est au pire de l'ordre de O(n).

Il faudra donc utiliser une catégorie spéciale d'arbres binaires qui restent équilibrés (leurs feuilles sont sur 2 niveaux au plus) pour assurer une recherche au pire en O(log(n)).

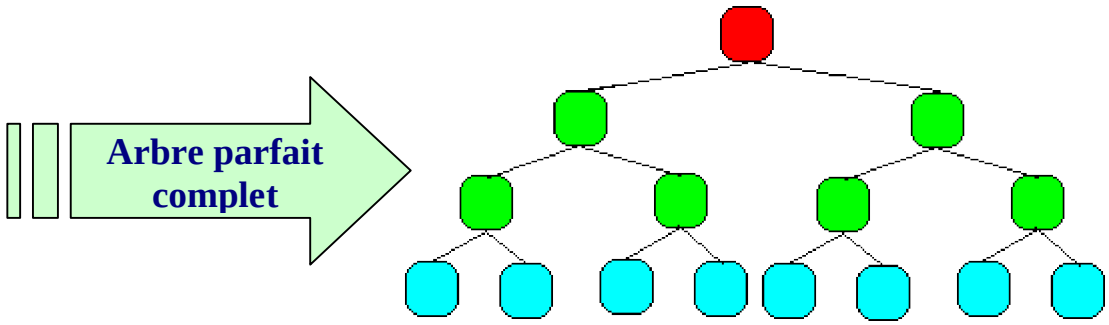
2.4 Arbres binaires partiellement ordonnés (tas)

Nous avons déjà évoqué la notion d'arbre parfait lors de l'étude du tri par tas, nous récapitulons ici les éléments essentiels le lecteur

Arbre parfait

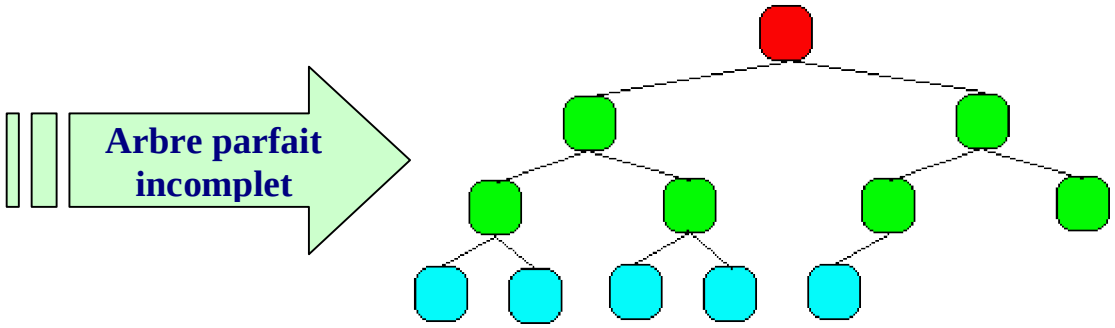
c'est un arbre binaire dont tous les noeuds de chaque niveau sont présents sauf éventuellement au dernier niveau où il peut manquer des noeuds (noeuds terminaux = feuilles), dans ce cas l'arbre parfait est un arbre binaire

incomplet et les feuilles du dernier niveau **doivent être regroupées à partir de la gauche** de l'arbre.



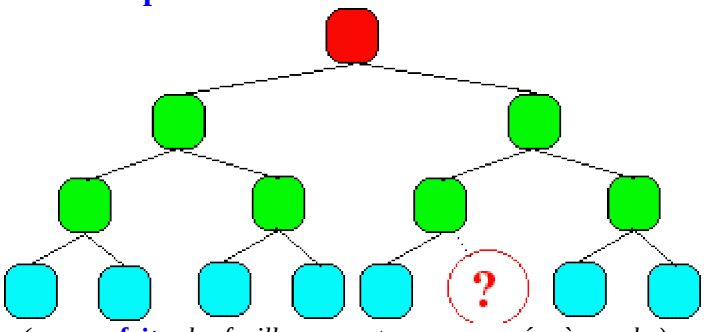
parfait complet : *le dernier niveau est complet car il contient tous les enfants*

un arbre **parfait** *peut être incomplet lorsque le dernier niveau de l'arbre est incomplet (dans le cas où manquent des feuilles à la droite du dernier niveau, les feuilles sont regroupées à gauche)*



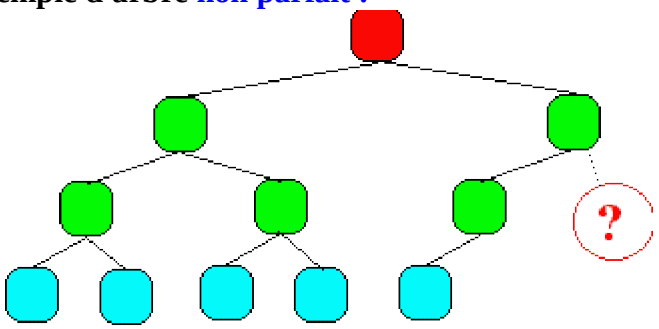
parfait incomplet: *le dernier niveau est incomplet car il manque 3 enfants à la droite*

Exemple d'arbre non parfait :



*(**non parfait :** les feuilles ne sont pas regroupées à gauche)*

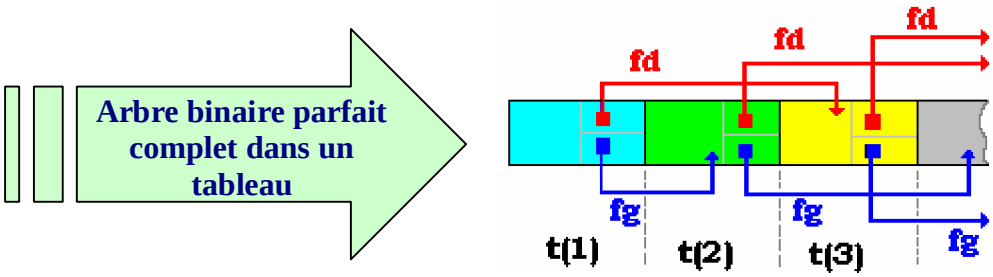
Autre exemple d'arbre non parfait :



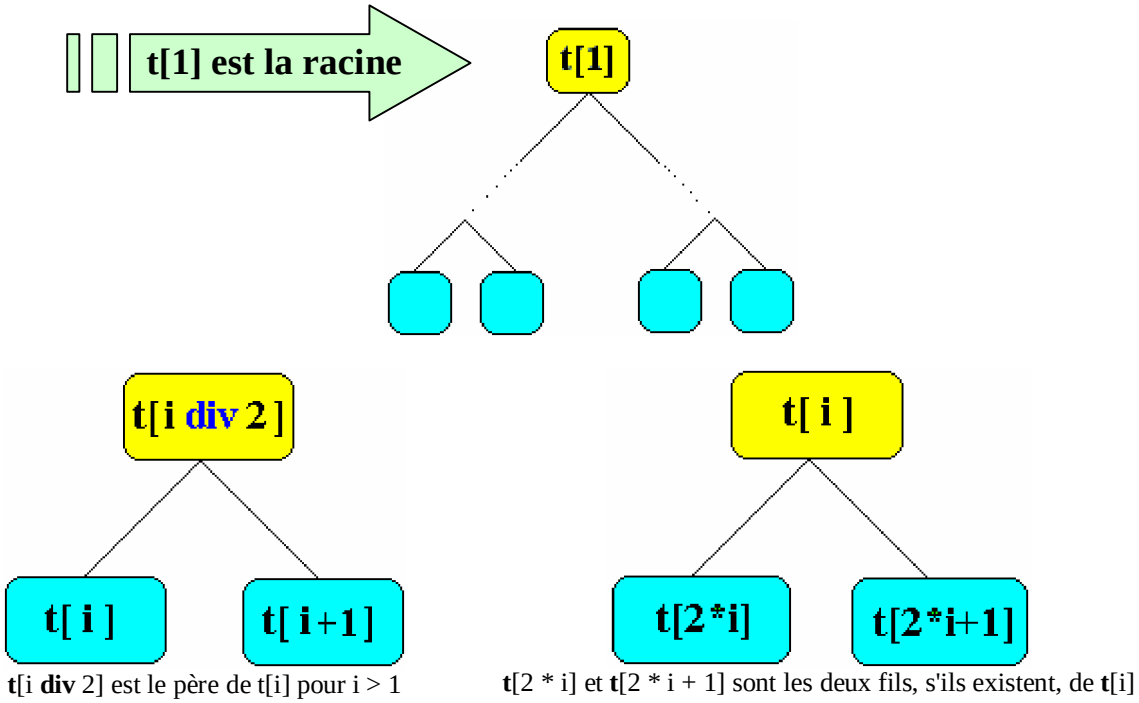
(**non parfait :** *les feuilles sont bien regroupées à gauche, mais il manque 1 enfant à l'avant dernier niveau)*

Un arbre binaire parfait se représente classiquement dans un tableau :

Les noeuds de l'arbre sont dans les cellules du tableau, il n'y a pas d'autre information dans une cellule du tableau, l'accès à la topologie arborescente est simulée à travers un calcul d'indice permettant de parcourir les cellules du tableau selon un certain 'ordre' de numérotation correspondant en fait à un **parcours hiérarchique** de l'arbre. En effet ce sont les numéros de ce parcours qui servent d'indice aux cellules du tableau nommé **t** ci-dessous :



Si **t** est ce tableau, nous avons les règles suivantes :

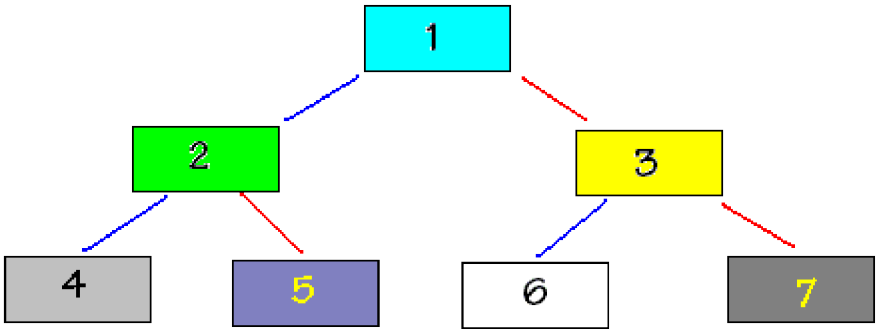


t[i **div** 2] est le père de t[i] pour i > 1

t[2 * i] et **t**[2 * i + 1] sont les deux fils, s'ils existent, de **t**[i]

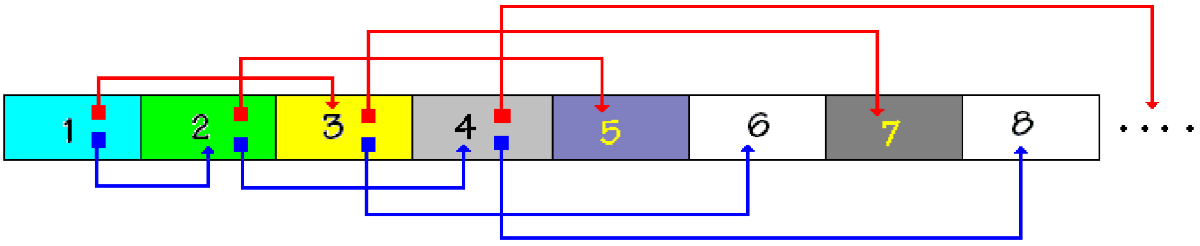
si **p** est le nombre de noeuds de l'arbre et si 2 * i = p, t[i] n'a qu'un fils, t[p].
si i est supérieur à p **div** 2, t[i] est une feuille.

Exemple de rangement d'un tel arbre dans un tableau
(on a figuré l'indice de numérotation hiérarchique de chaque noeud dans le rectangle associé au noeud)



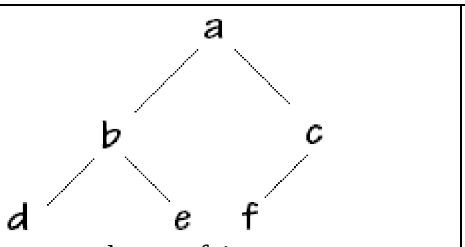
Cet arbre sera stocké dans un tableau en disposant séquentiellement et de façon contigüe les noeuds selon la numérotation hiérarchique (l'index de la cellule = le numéro hiérarchique du noeud).

Dans cette disposition le passage d'un noeud de numéro **k** (indice dans le tableau) vers son fils gauche s'effectue par calcul d'indice, le fils gauche se trouvera dans la cellule d'index **2*k** du tableau, son fils droit se trouvant dans la cellule d'index **2*k + 1** du tableau. Ci-dessous l'arbre précédent est stocké dans un tableau : le noeud d'indice hiérarchique 1 (la racine) dans la cellule d'index 1, le noeud d'indice hiérarchique 2 dans la cellule d'index 2, etc...
Le nombre qui figure dans la cellule (nombre qui vaut l'index de la cellule = le numéro hiérarchique du noeud) n'est mis là qu'à titre pédagogique afin de bien comprendre le mécanisme.

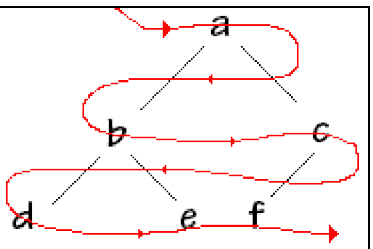


On voit par exemple, que par calcul on a bien le fils gauche du noeud d'indice 2 est dans la cellule d'index 2*2 = 4 et son fils droit se trouve dans la cellule d'index 2*2+1 = 5 ...

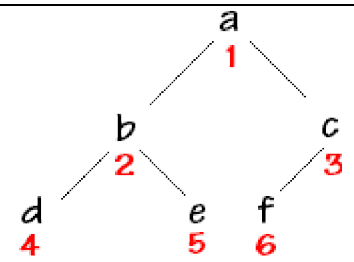
Exemple d'un arbre parfait étiqueté avec des caractères :



arbre parfait



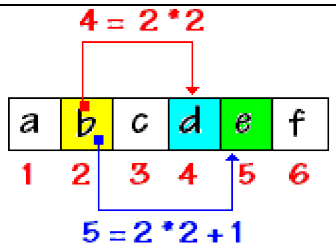
parcours hiérarchique



numérotation hiérarchique

rangement de l'arbre dans un tableau

Soit le noeud '**b**' de numéro hiérarchique 2 (donc rangé dans la cellule de rang 2 du tableau), son fils gauche est '**d**', son fils droit est '**e**'.



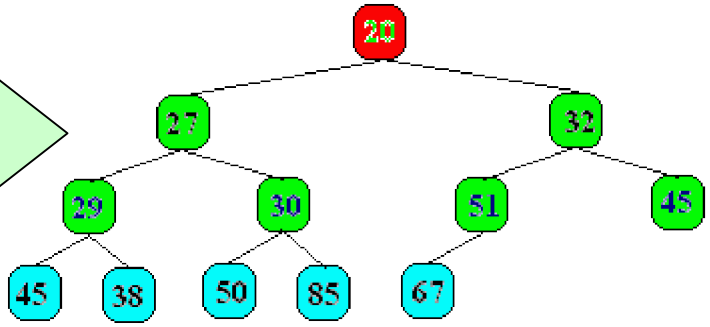
Arbre partiellement ordonné

C'est un arbre étiqueté dont les valeurs des noeuds appartiennent à un ensemble muni d'une **relation d'ordre total** (les nombres entiers, réels etc... en sont des exemples) tel que pour un noeud donné tous ses **fils ont une valeur supérieure ou égale à celle de leur père**.

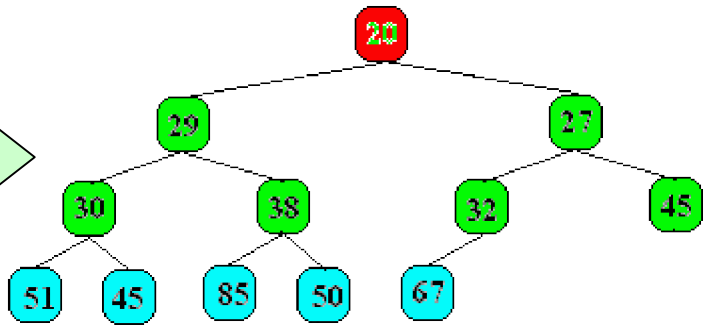
*Exemple de **deux** arbres partiellement ordonnés*

sur l'ensemble {20,27,29,30,32,38,45,45,50,51,67,85} d'entiers naturels :

Arbre partiellement ordonné n°1

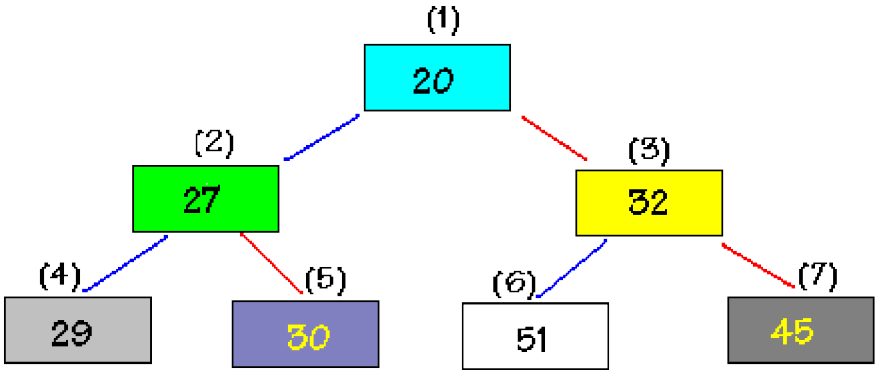


Arbre partiellement ordonné n°2

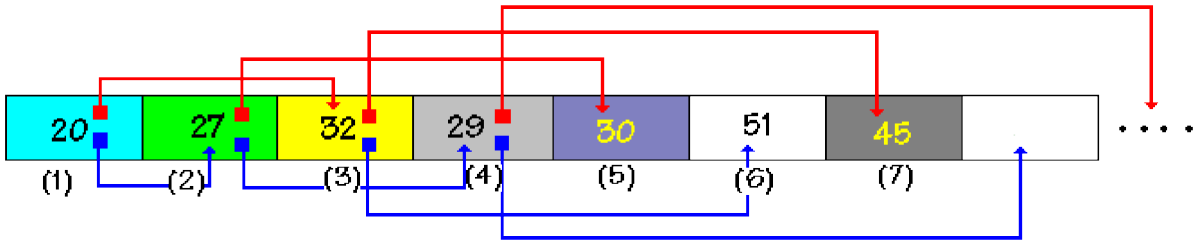


Nous remarquons que **la racine d'un tel arbre est toujours l'élément de l'ensemble possédant la valeur minimum** (le plus petit élément de l'ensemble), car la valeur de ce noeud par construction est inférieure à celle de ses fils et par transitivité de la relation d'ordre à celles

de ses descendants c'est le minimum. Si donc nous arrivons à ranger une liste d'éléments dans un tel arbre le minimum de cette liste est atteignable immédiatement comme racine de l'arbre. En reprenant l'exemple précédent sur 3 niveaux : *(entre parenthèses le numéro hiérarchique du noeud)*



Voici réellement ce qui est stocké dans le tableau : *(entre parenthèses l'index de la cellule contenant le noeud)*



Le tas

On appelle **tas** un tableau représentant un **arbre parfait partiellement ordonné**.

L'intérêt d'utiliser un arbre parfait complet ou incomplet réside dans le fait que le tableau est toujours **compacté**, les cellules vides s'il y en a se situent à la fin du tableau.
Le fait d'être partiellement ordonné sur les valeurs permet d'avoir immédiatement un **extremum** à la racine.

2.5 Parcours d'un arbre binaire

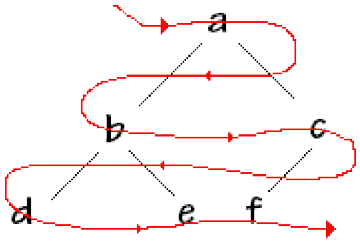
Objectif : les arbres sont des structures de données. Les informations sont contenues dans les noeuds de l'arbre, afin de construire des algorithmes effectuant des opérations sur ces informations (ajout, suppression, modification,...) il nous faut pouvoir examiner tous les noeuds d'un arbre. Examinons les différents moyens de parcourir ou de traverser chaque noeud de l'arbre et d'appliquer un traitement à la donnée rattachée à chaque noeud.

Parcours d'un arbre

L'opération qui consiste à **retrouver** systématiquement tous les noeuds d'un arbre et d'y appliquer un **même traitement** se dénomme **parcours** de l'arbre.

Parcours en largeur ou hiérarchique

Un algorithme classique consiste à **explorer** chaque noeud d'un niveau donné de **gauche à droite**, puis de passer au niveau suivant. On dénomme cette stratégie le parcours en largeur de l'arbre.



Parcours en profondeur

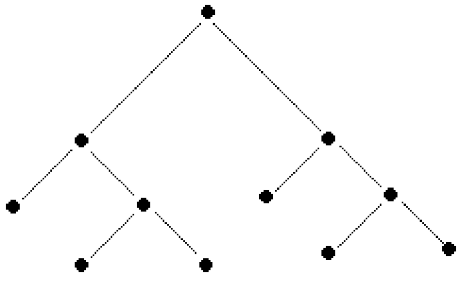
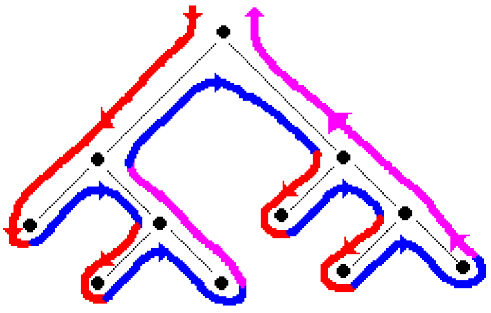
La stratégie consiste à **descendre** le plus profondément soit **jusqu'aux feuilles** d'un noeud de l'arbre, puis lorsque toutes les feuilles du noeud ont été visitées, l'algorithme "**remonte**" au noeud plus haut dont les feuilles n'ont pas encore été visitées.

Notons que ce parcours peut s'effectuer systématiquement en commençant par le fils gauche, puis en examinant le fils droit ou bien l'inverse.

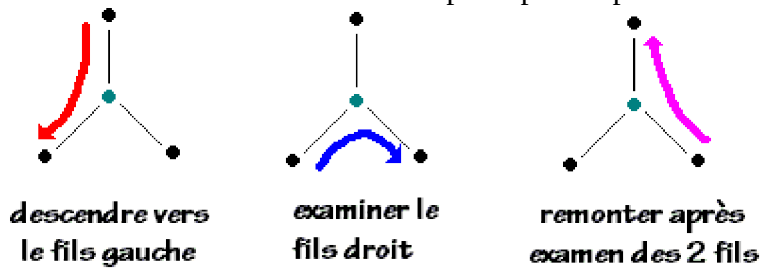
Parcours en profondeur par la gauche

Traditionnellement c'est l'exploration **fils gauche, puis ensuite fils droit** qui est retenue on dit alors que l'on traverse l'arbre en "**profondeur par la gauche**".

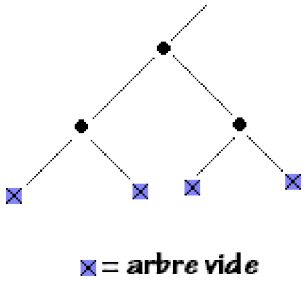
Schémas montrant le principe du parcours exhaustif en "**profondeur par la gauche**" :

Soit l'arbre binaire suivant:	Appliquons la méthode de parcours proposée :
	

Chaque noeud a bien été examiné selon les principes du parcours en profondeur :



En fait pour ne pas surcharger les schémas arborescents, nous omettons de dessiner à la fin de chaque noeud de type feuille les deux **noeuds enfants vides** qui permettent de reconnaître que le parent est une feuille :



Lorsque la compréhension nécessitera leur dessin nous conseillons au lecteur de faire figurer explicitement dans son schéma arborescent les noeuds vides au bout de chaque feuille.

Nous proposons maintenant, de donner une description en langage algorithmique LDFA du parcours en profondeur d'un arbre binaire sous forme récursive.

Algorithme général récursif de parcours en profondeur par la gauche

```
parcourir ( Arbre )
si Arbre ≠ ∅  alors
   Traiter-1 (info(Arbre.Racine)) ;
   parcourir ( Arbre.filsG ) ;
   Traiter-2 (info(Arbre.Racine)) ;
   parcourir ( Arbre.filsD ) ;
   Traiter-3 (info(Arbre.Racine)) ;
Fsi
```

Les différents traitements Traiter-1 ,Traiter-2 et Traiter-3 consistent à traiter l'information située dans le noeud actuellement traversé soit lorsque l'on descend vers le fils gauche (Traiter-1), soit en allant examiner le fils droit (Traiter-2), soit lors de la remonté après examen des 2 fils (Traiter-3).

En fait on n'utilise en pratique que trois variantes de cet algorithme, celles qui constituent des parcours ordonnés de l'arbre en fonction de l'application du traitement de l'information située aux noeuds. Chacun de ces 3 parcours définissent un ordre implicite (préfixé, infixé, postfixé) sur l'affichage et le traitement des données contenues dans l'arbre.

Algorithme de parcours en pré-ordre :

```
parcourir ( Arbre )
si Arbre ≠ ∅ alors
   Traiter-1 (info(Arbre.Racine)) ;
   parcourir ( Arbre.filsG ) ;
   parcourir ( Arbre.filsD ) ;
Fsi
```

Algorithme de parcours en post-ordre :

```
parcourir ( Arbre )
si Arbre ≠ ∅ alors
   parcourir ( Arbre.filsG ) ;
   parcourir ( Arbre.filsD ) ;
   Traiter-3 (info(Arbre.Racine)) ;
Fsi
```

Algorithme de parcours en ordre symétrique :

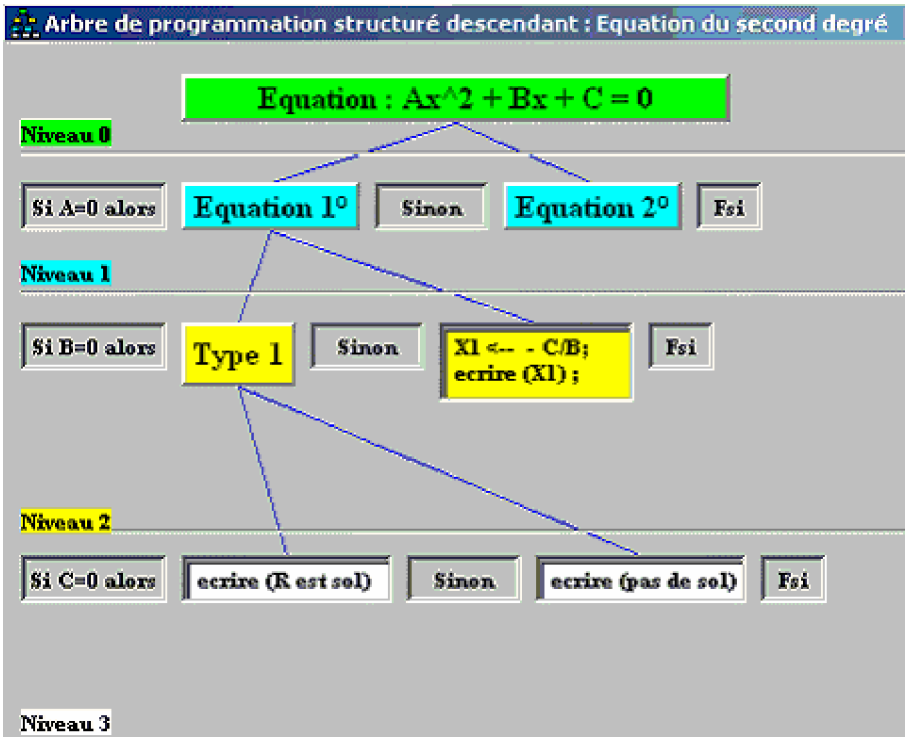
```
parcourir ( Arbre )
si Arbre ≠ ∅ alors
   parcourir ( Arbre.filsG) ;
   Traiter-2 (info(Arbre.Racine)) ;
   parcourir ( Arbre.filsD ) ;
Fsi
```

Illustration pratique d'un parcours général en profondeur

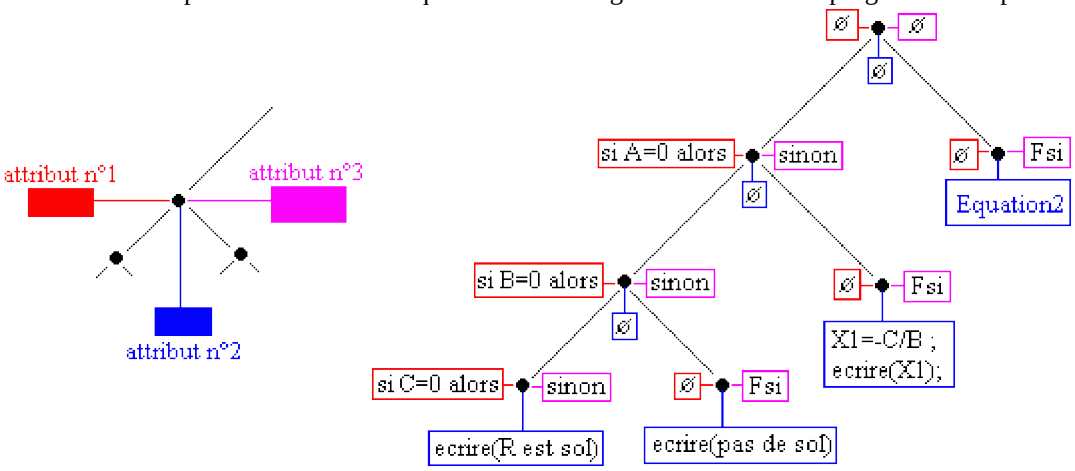
Le lecteur trouvera plus loin des exemples de parcours selon l'un des 3 ordres infixé, préfixé, postfixé, nous proposons ici un exemple didactique de parcours général avec les 3 traitements.

Nous allons voir comment utiliser une telle structure arborescente afin de restituer du texte algorithmique linéaire en effectuant un parcours en profondeur.

Voici ce que nous donne une analyse descendante du problème de résolution de l'équation du second degré (nous avons fait figurer uniquement la branche gauche de l'arbre de programmation) :



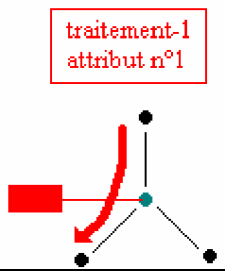
Ci-dessous une représentation schématique de la branche gauche de l'arbre de programmation précédent :



Nous avons établi un modèle d'arbre (binaire ici) où les informations au noeud sont au nombre de 3 (nous les nommerons attribut n°1, attribut n°2 et attribut n°3). Chaque attribut est une **chaîne de caractères**, vide s'il y a lieu.

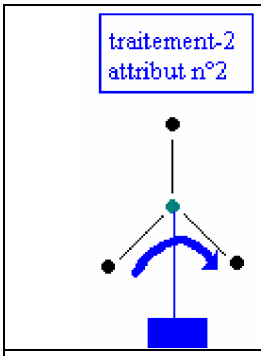
Nous noterons ainsi un attribut contenant une chaîne vide : ∅

Traitement des attributs pour produire le texte



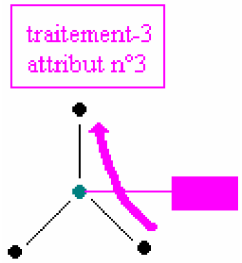
Traiter-1 (**Arbre.Racine.Attribut n°1**) consiste à écrire le contenu de l'Attribut n°1 :

si Attribut n°1 non vide **alors**
ecrire(Attribut n°1)
Fsi



Traiter-2 (**Arbre.Racine.Attribut n°2**) consiste à écrire le contenu de l'Attribut n°2 :

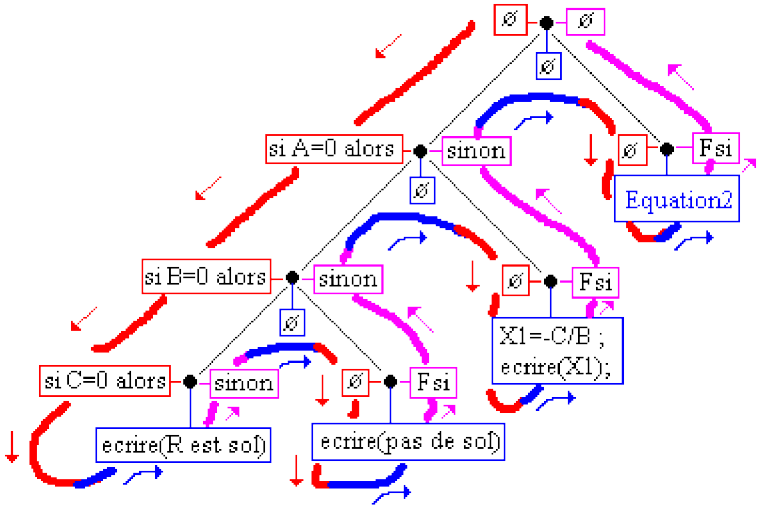
si Attribut n°2 non vide **alors**
ecrire(Attribut n°2)
Fsi



Traiter-3 (**Arbre.Racine.Attribut n°3**) consiste à écrire le contenu de l'Attribut n°3 :

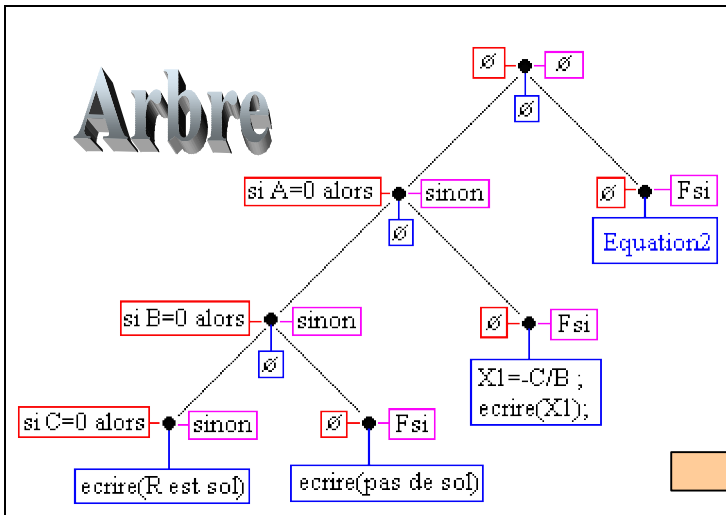
si Attribut n°3 non vide **alors**
ecrire(Attribut n°3)
Fsi

Parcours en profondeur de l'arbre de programmation de l'équation du second degré :



parcourir (Arbre)

si Arbre ≠ ∅ alors
Traiter-1 (**Attribut n°1**) ;
parcourir (Arbre.filsG) ;
Traiter-2 (**Attribut n°2**) ;
parcourir (Arbre.filsD) ;
Traiter-3 (**Attribut n°3**) ;
Fsi



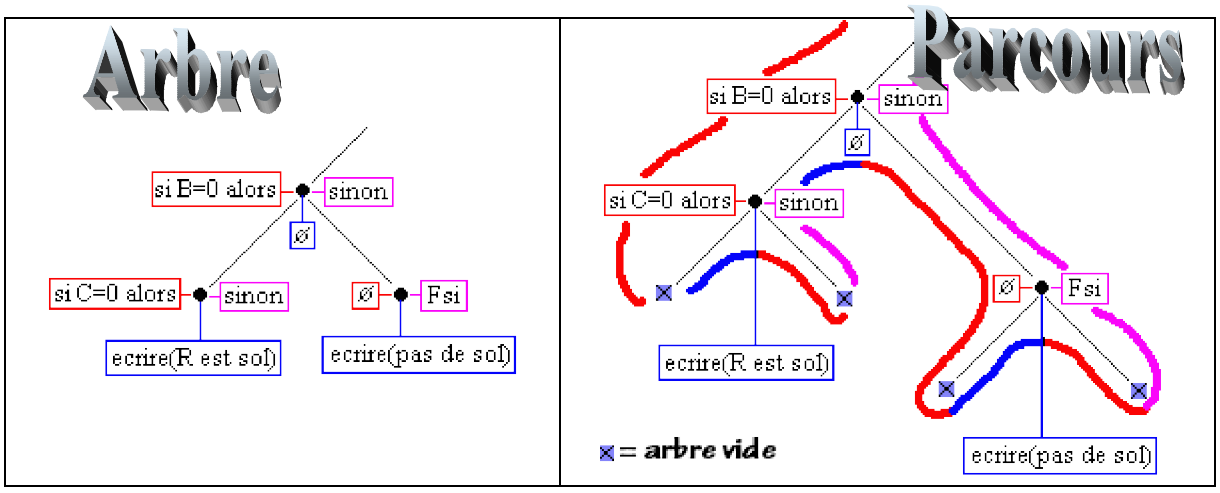
∅
si A=0 alors
si B=0 alors
si C=0 alors
ecrire(R est sol)
sinon
∅
∅
ecrire(pas de sol)
Fsi
sinon
∅
∅
X1=-C/B;
ecrire(X1);

Texte

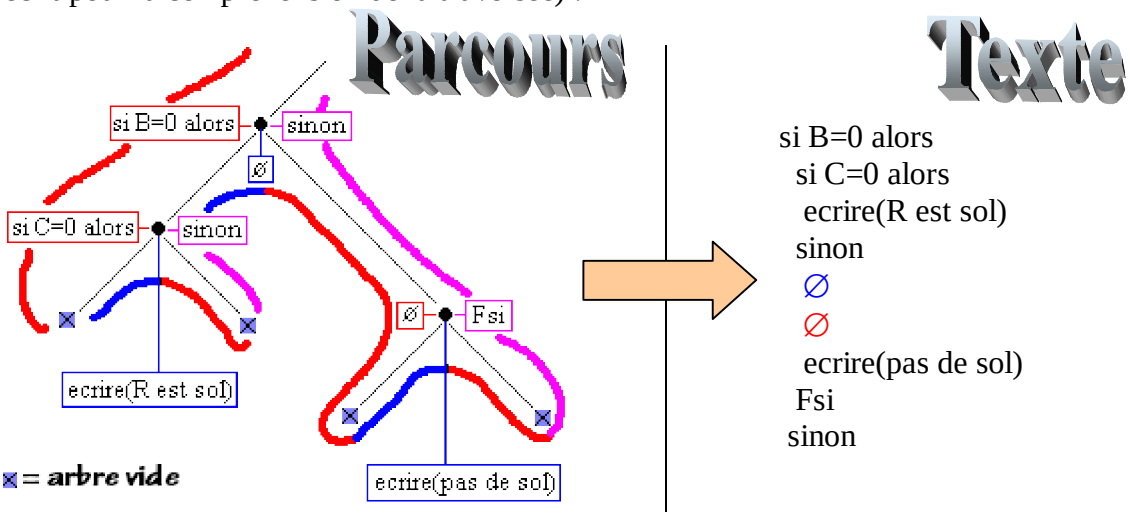
Parcours	
si Arbre ≠ ∅ alors Traiter-1 (**Attribut n°1**) ; **parcourir (Arbre.filsG) ;** Traiter-2 (**Attribut n°2**) ; **parcourir (Arbre.filsD) ;** Traiter-3 (**Attribut n°3**) ; **Fsi**	Fsi sinon ∅ ∅ Equation2 Fsi ∅

Rappelons que le symbole ∅ représente la chaîne vide il est uniquement mis dans le texe dans le but de permettre le suivi du parcours de l'arbre.

Pour bien comprendre le parcours aux feuilles de l'arbre précédent, nous avons fait figurer ci-dessous sur un exemple, les **noeuds vides** de chaque feuille et le **parcours complet associé** :

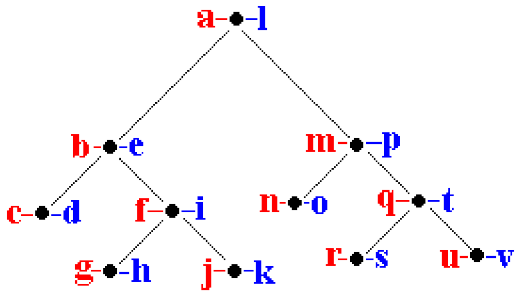


Le parcours partiel ci-haut produit le texte algorithmique suivant (le symbole ∅ est encore écrit pour la compréhension de la traversée) :



Exercice

Soit l'arbre ci-contre possédant 2 attributs par noeuds (un symbole de type caractère)



On propose le traitement en profondeur de l'arbre comme suit : L'attribut de gauche est écrit en descendant, l'attribut de droite est écrit en remontant, il n'y a pas d'attribut ni de traitement lors de l'examen du fils droit en venant du fils gauche. écrire la chaîne de caractère obtenue par le parcours ainsi défini.	**Réponse** : **abcdfghjkiemnoqrsuvtpl**

Terminons cette revue des descriptions algorithmiques des différents parcours classiques d'arbre binaire avec le parcours en largeur (*Cet algorithme nécessite l'utilisation d'une file du type Fifo dans laquelle l'on stocke les nœuds*)**.**

Algorithme de parcours en largeur

```
Largeur ( Arbre )

si Arbre ≠ ∅  alors
   ajouter racine de l'Arbre dans Fifo;
   tantque Fifo ≠ ∅ faire
      prendre premier de Fifo;
      traiter premier de Fifo;
      ajouter filsG de premier de Fifo dans Fifo;
      ajouter filsD de premier de Fifo dans Fifo;
   ftant
Fsi
```

2.6 Insertion, suppression, recherche dans un arbre binaire de recherche

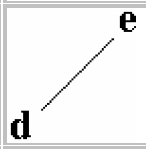
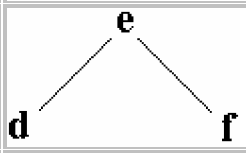
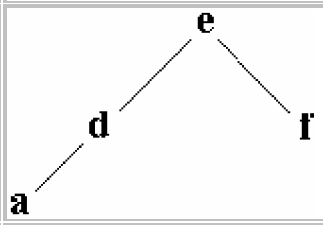
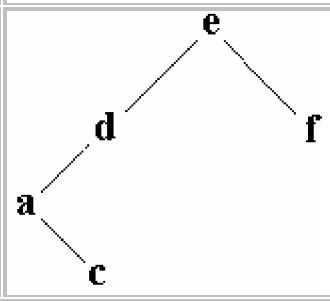
Algorithme d'insertion dans un arbre binaire de recherche

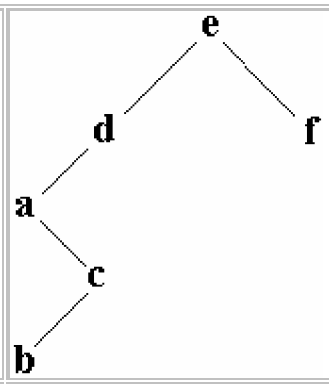
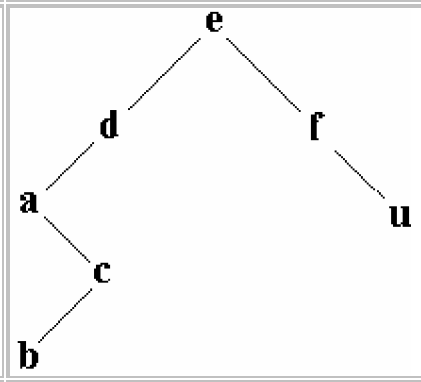
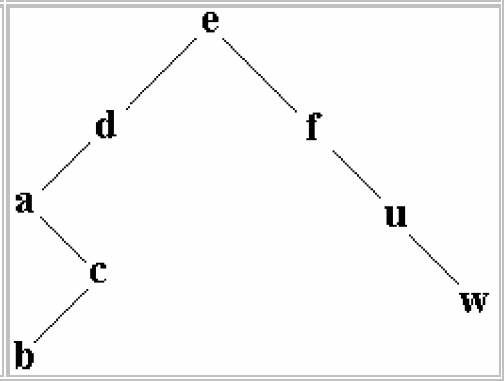
*placer l'élément Elt dans l'arbre **Arbre** par adjonctions successives aux feuilles*

```
placer ( Arbre Elt )

si Arbre = ∅ alors
   creer un nouveau noeud contenant Elt ;
   Arbre.Racine = ce nouveau noeud
sinon
{ - tous les éléments "info" de tous les noeuds du sous-arbre de gauche
```

```
    sont inférieurs ou égaux à l'élément "info" du noeud en cours (arbre)
  - tous les éléments "info" de tous les noeuds du sous-arbre de droite
    sont supérieurs à l'élément "info" du noeud en cours (arbre)
 }
  si clef ( Elt ) ≤ clef ( Arbre.Racine ) alors
    placer ( Arbre.filsG Elt )
  sinon
    placer ( Arbre.filsD Elt )
 Fsi
```

Soit par exemple la liste de caractères alphabétiques : **e d f a c b u w** *, que nous rangeons dans cet ordre d'entrée dans un arbre binaire de recherche. Ci-dessous le suivi de l'algorithme de placements successifs de chaque caractère de cette liste dans un arbre de recherche:*

Insertions successives des éléments	Arbre de recherche obtenu
placer (racine , '**e**') *e est la racine de l'arbre.*	
placer (racine , '**d**') *d < e donc fils gauche de e.*	
placer (racine , '**f**') *f > e donc fils droit de e.*	
placer (racine , '**a**') *a < e donc à gauche, a < d donc fils gauche de d.*	
placer (racine , '**c**') *c < e donc à gauche, c < d donc à gauche, c > a donc fils droit de a.*	

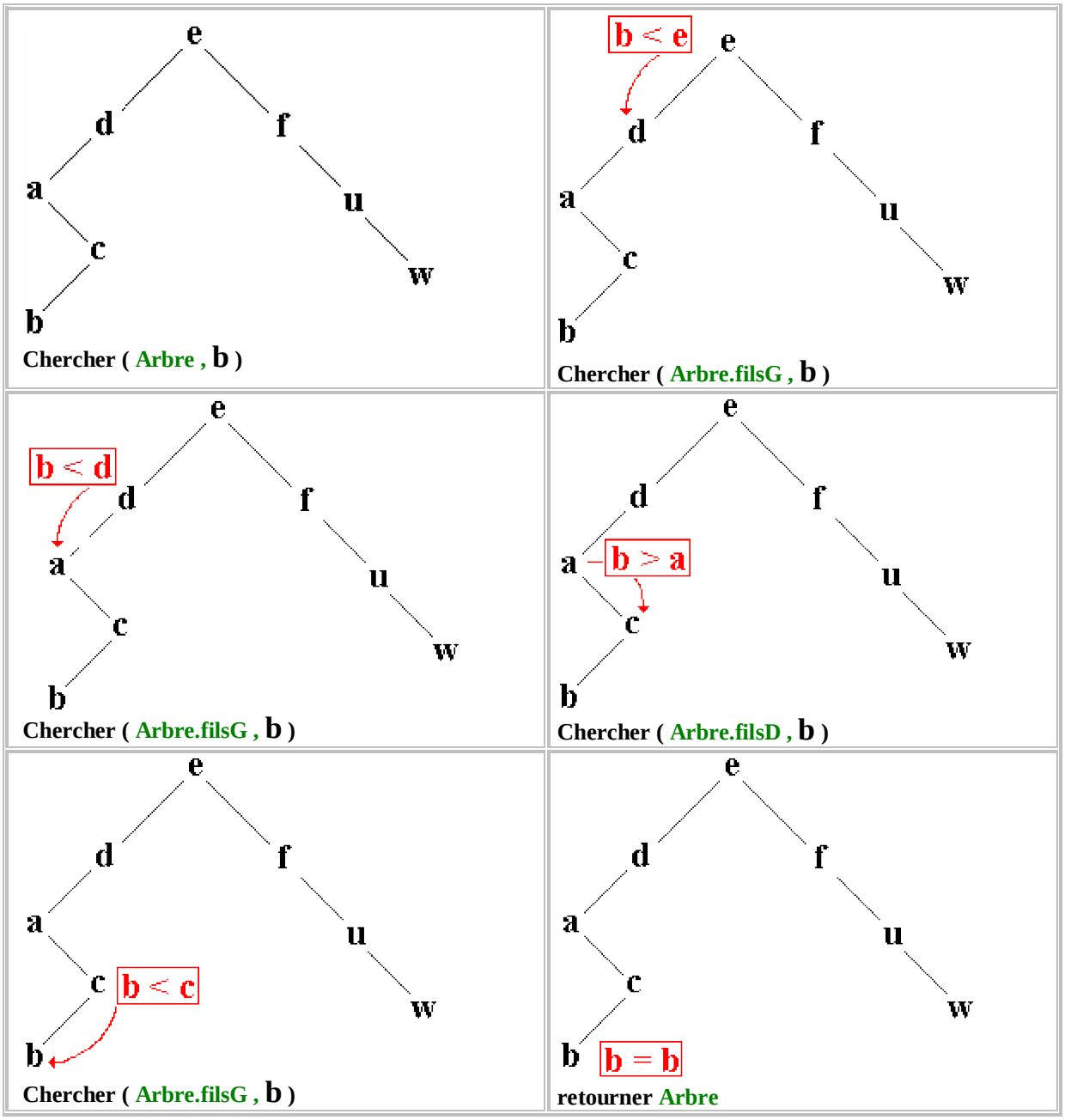
placer (racine , '**b**') *b < e donc à gauche, b < d donc à gauche, b > a donc à droite de a, b < c donc fils gauche de c.*	
placer (racine , '**u**') *u > e donc à droite de e, u > f donc fils droit de f.*	
placer (racine , '**w**') *w > e donc à droite de e, w > f donc à droite de f, w > u donc fils droit de u.*	

Algorithme de recherche dans un arbre binaire de recherche

*chercher l'élément Elt dans l'arbre **Arbre :***

```
Chercher ( Arbre Elt ) : Arbre

si Arbre = ∅ alors
  Afficher Elt non trouvé dans l'arbre;
sinon
  si clef ( Elt )  <  clef ( Arbre.Racine ) alors
    Chercher ( Arbre.filsG Elt ) //on cherche à gauche
  sinon
     si clef ( Elt )  >  clef ( Arbre.Racine ) alors
       Chercher ( Arbre.filsD Elt ) //on cherche à droite
     sinon retourner Arbre.Racine  //l'élément est dans ce noeud
     Fsi
  Fsi
Fsi
```

Ci-dessous le suivi de l'algorithme de recherche du caractère **b** *dans l'arbre précédent :*



Algorithme de suppression dans un arbre binaire de recherche

Afin de pouvoir supprimer un élément dans un arbre binaire de recherche, il est nécessaire de pouvoir d'abord le localiser, ensuite supprimer le noeud ainsi trouvé et éventuellement procéder à la réorganisation de l'arbre de recherche.

Nous supposons que notre arbre binaire de recherche ne possède que des éléments tous distincts (pas de redondance).

```
supprimer l'élément Elt dans l'arbre Arbre :
Supprimer ( Arbre Elt ) : Arbre
 local Node : Noeud

si Arbre = ∅ alors
  Afficher Elt non trouvé dans l'arbre;
sinon
  si clef ( Elt )  < clef ( Arbre.Racine )   alors
    Supprimer ( Arbre.filsG Elt ) //on cherche à gauche
  sinon
     si clef ( Elt )  >  clef ( Arbre.Racine ) alors
       Supprimer ( Arbre.filsD Elt ) //on cherche à droite
     sinon //l'élément est dans ce noeud
      si Arbre.filsG = ∅ alors //sous-arbre gauche vide
        Arbre ←Arbre.filsD //remplacer arbre par son sous-arbre droit
      sinon
       si Arbre.filsD = ∅ alors  //sous-arbre droit vide
        Arbre ←Arbre.filsG //remplacer arbre par son sous-arbre gauche
       sinon //le noeud a deux descendants
          Node ←PlusGrand( Arbre.filsG ); //Node = le max du fils gauche
          clef ( Arbre.Racine ) ← clef ( Node ); //remplacer etiquette
          détruire ( Node ) //on élimine ce noeud
       Fsi
      Fsi
     Fsi
   Fsi
Fsi
```

Cet algorithme utilise l'algorithme récursif **PlusGrand** *de recherche du plus grand élément dans l'arbre* ***Arbre :***

//par construction il suffit de descendre systématiquement toujours le plus à droite

```
PlusGrand ( Arbre ) : Arbre
si Arbre.filsD = ∅ alors
  retourner Arbre.Racine //c'est le plus grand élément
sinon
  PlusGrand ( Arbre.filsD  )
Fsi
```