Chapitre 4 : Structures de données

4.1.Spécification abstraite de données

- Le Type Abstrait Algébrique (TAA)
- Disposition pratique d'un TAA
- Le Type Abstrait de Donnée (TAD)
- Classification hiérarchique
  - le TAD liste linéaire
  - le TAD pile LIFO
  - le TAD file FIFO
- Exercices d'implantation de TAD

4.2. Implantation de TAD avec Unit en Delphi

- **Types abstraits de données et Unités en Delphi**

  Traduction générale TAD → Unit pascal
  Exemples de Traduction TAD → Unit pascal
  Variations sur les spécifications d'implantation
  Exemples d'implantation de la liste linéaire
  Exemples d'implantation de la pile LIFO
  Exemples d'implantation de la file FIFO

4.3. Structures d'arbres binaires

- 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

- Exemples et implémentation d'arbre
  - Arbre de dérivation
  - Arbre abstrait
  - Arbre lexicographique
  - Arbre d'héritage
  - Arbre de recherche

- Arbres binaires
- TAD d'arbre binaire
- Exemples et implémentation d'arbre
    - *tableau statique*
    - *variable dynamique*
    - *classe*
- Arbres binaires de recherche
- Arbres binaires partiellement ordonnés *(tas)*
- 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

- Exercices deux TAD implantés en Unit et en classes avec Delphi

4.1 : Spécification abstraite de donnée

Plan du chapitre:

Introduction

Le mécanisme de l'abstraction est l'un des **plus** important avec celui de la méthode structurée fonctionnelle en vue de la réalisation des programmes.

Notion d'abstraction en informatique

L'abstraction consiste à penser à un objet en termes d'actions que l'on peut effectuer sur lui, et non pas en termes de représentation et d'implantation de cet objet.

C'est cette attitude qui a conduit notre société aux grandes réalisations modernes. C'est un art de l'ingénieur et l'informatique est une science de l'ingénieur.

Dès le début de la programmation nous avons vu apparaître dans les langages de programmation les notions de **subroutine** puis de **procédure** et de **fonction** qui sont une abstraction d'encapsulation pour les familles d'instructions structurées:

- Les paramètres formels sont une **abstraction** fonctionnelle (comme des variables muettes en mathématiques),
- Les paramètres effectifs au moment de l'appel sont des **instanciations** de ces paramètres formels (une implantation particulière).

L'idée de considérer les **types** de données comme une abstraction date des années 80. On s'est en effet aperçu qu'il était nécessaire de s'abstraire de la représentation ainsi que pour l'abstraction fonctionnelle. On a vu apparaître depuis une vingtaine d'année un domaine de recherche : celui des **spécifications algébriques**. Cette recherche a donné naissance au concept de **Type Abstrait Algébrique** (TAA).

Selon ce point de vue une structure de donnée devient:

Une collection d'informations structurées et reliées entre elles selon un graphe relationnel établi grâce aux opérations effectuées sur ces données.

Nous spécifions d'une façon simple ces structures de données selon deux niveaux d'abstraction, du plus abstrait au plus concret.

Une spécification abstraite :

Description des propriétés générales et des opérations qui décrivent la structure de données.

Une spécification opérationnelle :

Description d'une forme d'implantation informatique choisie pour représenter et décrire la structure de donnée. Nous la divisons en deux étapes : la spécification opérationnelle concrète (choix d'une structure informatique classique) et la spécification opérationnelle d'implantation (choix de la description dans le langage de programmation).

Remarque :

Pour une spécification abstraite fixée nous pouvons définir plusieurs spécifications opérationnelles différentes.

1. La notion de TAD (Type Abstrait de Données)

Bien que nous situant au niveau débutant il nous est possible d'utiliser sans effort théorique et mental compliqué, une méthode de spécification semi-formalisée des données. Le " type abstrait de donnée " basé sur le type abstrait algébrique est une telle méthode.
Le lecteur ne souhaitant pas aborder le formalisme mathématique peut sans encombre pour la suite, sauter le paragraphe qui suit et ne retenir que le point de vue pratique de la syntaxe d'un TAA.

1.1 Le Type Abstrait Algébrique (TAA)

Dans ce paragraphe nous donnons quelques indications théoriques sur le support formel algébrique de la notion de TAA. *(notations de F.H.Raymond cf.Biblio)*

Notion d'algèbre formelle informatique

Soit (Fn) $n \in \mathbf{N}$ **,** une famille d'ensembles tels que :

$\mathbf{(}\exists i_0 \in \mathbf{N}\ (1 \leq i_0 \leq n) / Fi_0 \neq \varnothing \mathbf{)} \wedge \mathbf{(}\forall i, \forall j,\ i \neq j \Rightarrow F_i \cap F_j = \varnothing \mathbf{)}$
posons : $\mathbf{I} = \{ n \in \mathbf{N} / \mathbf{F}_n \neq \varnothing \}$

Vocabulaire :

Les symboles ou éléments de F_0 sont appelés symboles de constantes ou symboles fonctionnels 0-aire.

Les symboles de F_n (où $n \geq 1$ et $n \in \mathbf{I}$) sont appelés symboles fonctionnels n-aires.

Notation :

$$F = \bigcup_{n \in N} Fn = \bigcup_{n \in \mathbf{I}} Fn$$

soit F* l'ensemble des expressions sur F, **le couple (F^*,F) est appelé une algèbre abstraite**.

On définit pour tout symbole fonctionnel n-aire f, une application de F^*_n dans F^* notée "$\hat{f}$" de la façon suivante :

$\forall$ e, ($f \in F_n \rightarrow \hat{f}$)

et {$\hat{f}$: $F^{*n}_{\bullet} \rightarrow F^*$ telle que $(a_1,..., a_n) \rightarrow \hat{f}(a1,...,an) = f(a1,...,an)$ }

On dénote :

$F_n = \{ \hat{f} / f \in Fn \}$ et $\hat{F} = \bigcup_{n \in I} Fn$

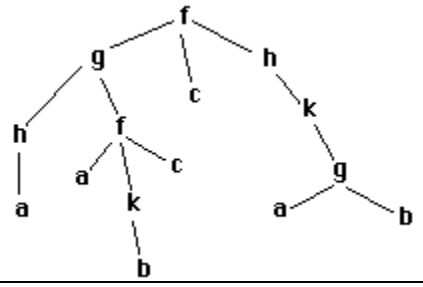
le couple $(F^*, \hat{F})$ est appelé une algèbre formelle informatique (AFI).

Les expressions de F^* construites à partir des fonctions $\hat{f}$ sur des symboles fonctionnels n-aires s'appellent des schémas fonctionnels.

Par définition, les schémas fonctionnels de F_0 sont appelés les constantes.

Exemple :
F0 = {a,b,c} // *les constantes*
F1 = {h,k} // *les symboles unaires*
F2 = {g} // *les symboles binaires*
F3 = {f} // *les symboles ternaires*

Soit le schéma fonctionnel : **fghafakbcchkgab**
(chemin aborescent abstrait non parenthésé)

Ce schéma peut aussi s'écrire avec un parenthèsage :	ou encore avec une représentation arborescente:
f [g(h(a),f(a,k(b),c)), c, h(k(g(a,b)))]	

Interprétation d'une algèbre formelle informatique

Soit une algèbre formelle informatique (AFI) : **(** $F^*, \hat{F}$ **)**

- on se donne un ensemble **X** tel que **X** $\neq \varnothing$,
- **X** est muni d'un ensemble d'opérations sur X noté Ω,

L'on construit une fonction ψ telle que :

ψ : **(** $F^*, \hat{F}$ **)** → **X** ayant les propriétés d'un homomorphisme

ψ est appelée fonction d'interprétation de l'AFI.
X est appelée l'univers de l'AFI.

Une AFI est alors un modèle abstrait pour toute une famille d'éléments fonctionnels, il suffit de changer le modèle d'interprétation pour implanter une structure de données spécifique.

Exemple :

F_0 = {x, y} une AFI
F_2 = {f, g }
$F = F_0 \cup F_2$

l'Univers : **X** = **R** *(*les nombres réels)

les opérateurs Ω = {+,*****} (addition et multiplication)

l'interprétation y : **(**$F^*, \hat{F}$**)** → **(R , Ω)**
définie comme suit :

ψ (f) : $\mathbf{R}^2$ → **R**
ψ (f) : (a,b) → ψ (f) [a,b] = a+b

ψ (g) : $\mathbf{R}^2$ → **R**
ψ (g) : (a,b) → ψ (g) [a,b] = a*****b

ψ (x) = a_0 (avec $a_0 \in$ **R** fixé interprétant la constante x)
ψ (y) = a_1 (avec $a_1 \in$ **R** fixé interprétant la constante y)

Soit le schéma fonctionnel **fxgyx,** son interprétation dans ce cas est la suivante :

ψ(fxgyx) = ψ(f (x,g(y,x))) = ψ(f) **(**ψ(x),ψ(g)[ψ(y),ψ(x)] **)**
= ψ(x) + ψ(g)[ψ(y),ψ(x)] ⇐ *propriété de ψ(f)*
= ψ(x) + ψ(y)*****ψ(x) ⇐ *propriété de ψ(g)*

Ce qui donne comme résultat : ψ (fxgyx) = a_0 + a_1 * a_0

A partir de la même AFI, il est possible de définir une autre fonction d'interprétation ψ'et un autre Univers **X**'.

par exemple :
l'Univers : **X = N** *(les entiers naturels)*
les opérateurs : Ω = **{ reste** , ≥ **}** *(le reste de la division euclidienne et la relation d'ordre)*

La fonction ψ'est définie comme suit :

ψ'(g) : $\mathbf{N}^2 \rightarrow \mathbf{N}$
ψ'(g) : (a,b) → ψ'(g)[a,b] = **reste**(a,b)

ψ'(f) : $\mathbf{N}^2 \rightarrow \mathbf{N}$
ψ'(f) : (a,b) → ψ'(f)[a,b] = a ≥ b ψ'(x) = n_0 (avec $n_0 \in$ **N** fixé)
ψ'(y) = n_1 (avec $n_1 \in$ **N fixé**)

On interprète alors le même schéma fonctionnel dans ce nouveau cas **fxgyx** :
ψ'(**fxgyx**) = $n_0 \geq$ **reste**(n_1, n_0)

Ceci n'est qu'une interprétation. cette interprétation reste encore une abstraction de plus bas niveau, le sens (sémantique d'exécution), s'il y en a un, sera donné lors de l'implantation de ces éléments. Nous allons définir un outil informatique se servant de ces notions d'AFI et d'interprétation, il s'agit du type abstrait algébrique.

Un TAA (type abstrait algébrique) est alors la donnée du triplet :
- une AFI
- un univers **X** et Ω
- une fonction d'interprétation ψ

la syntaxe du TAA est définie par l'**AFI** et l'ensemble **X**
la sémantique du TAA est définie par ψ et l'ensemble Ω

Notre objectif étant de rester pratique, nous arrêterons ici la description théorique des TAA (compléments cités dans la bibliographie pour le lecteur intéressé).

1.2 Disposition pratique d'un TAA

on écrira (exemple fictif):

Sorte : A, B, C ..... *les noms de types définis par le TAA, ce sont les types au sens des langages de programmation.*

Opérations :

f : A x B → B
g : A → C
x : → B *(notation pour les symboles de constantes de F0)*
y : → B *(notation pour les symboles de constantes de F0)*

Cette partie qui décrit la syntaxe du TAA s'appelle aussi **la signature du TAA** .

La sémantique est donnée par ψ , Ω sous la forme d'axiomes et de préconditions.

Le domaine d'une opération définie partiellement est défini par une précondition.

Un TAA réutilise des TAA déjà définis, sous forme de **hiérarchie**. Dans ce cas, la signature totale est la réunion des signatures de tous les TAA.

Si des opérateurs utilisent le même symbole, le problème de **surcharge** peut être résolu sans difficulté, parce que les opérateurs sont définis par leur ensembles de définitions.

SYNTAXE DE L'ECRITURE D'UN TYPE ABSTRAIT ALGEBRIQUE :

sorte : .....
utilise : .....
opérations :
préconditions :
..... **def_ssi** .....
axiomes :

FinTAA

Exemple d'écriture d'un TAA (les booléens) :

sorte : Booléens
opérations :
V : → Booléens
F : → Booléens
¬ : Booléens → Booléens
∧ : Booléens x Booléens → Booléens
∨ : Booléens x Booléens → Booléens
axiomes :

¬ (V) = F ; ¬ (F) = V
a ∧ V = a ; a ∧ F = F
a ∨ V = V ; a ∨ F = a

FinBooléens

1.3 Le Type Abstrait de Donnée (TAD)

Dans la suite du document les TAA ne seront pas utilisés entièrement, la partie axiomes étant occultée. Seules les parties opérations et préconditions sont étudiées en vue de leur implantation.
C'est cette restriction d'un TAA que nous appellerons un type abstrait de données (TAD). Nous allons fournir dans les paragraphes suivants quelques Types Abstrait de Données différents.

Nous écrirons ainsi par la suite un TAD selon la syntaxe suivante :

TAD Truc

utilise : .....
Champs : .....
opérations : .....
préconditions : .....

FinTAD Truc

Le TAD Booléens s'écrit à partir du TAA Booléens :

TAD Booléens
opérations :
V : → Booléens
F : → Booléens
¬ : Booléens → Booléens
∧ : Booléens x Booléens → Booléens
∨ : Booléens x Booléens → Booléens
FinTAD Booléen

Nous remarquons que cet outil permet de spécifier des structures de données d'une manière générale sans avoir la nécessité d'en connaître l'implantation, ce qui est une caractéristique de la notion d'abstraction.

1.4 Classification hiérarchique

Nous situons, dans notre approche pédagogique de la notion d'abstraction, les TAD au sommet de la hiérarchie informationnelle :

HIERARCHIE INFORMATIONNELLE

- **1°** TYPES ABSTRAITS (les TAA,...)
- **2°** CLASSES / OBJETS
- **3°** MODULES
- **4°** FAMILLES de PROCEDURES et FONCTIONS
- **5°** ROUTINES (procédures ou fonctions)
- **6°** INSTRUCTIONS STRUCTUREES ou COMPOSEES
- **7°** INSTRUCTIONS SIMPLES (langage évolué)
- **8°** MACRO-ASSEMBLEUR
- **9°** ASSEMBLEUR (langage symbolique)
- **10°** INSTRUCTIONS MACHINE (binaire)

Nous allons étudier dans la suite 3 exemples complets de TAD classiques : la **liste linéaire**, la pile **LIFO**1, la file **FIFO**2. Pour chacun de ces exemples, il sera fourni une spécification opérationnelle en pascal, puis plus loin en Delphi.

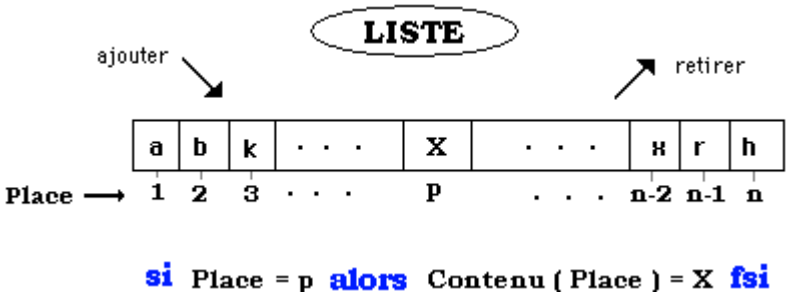
Exemples de types abstraits de données

1.5 Le TAD liste linéaire (spécifications abstraite et concrète)

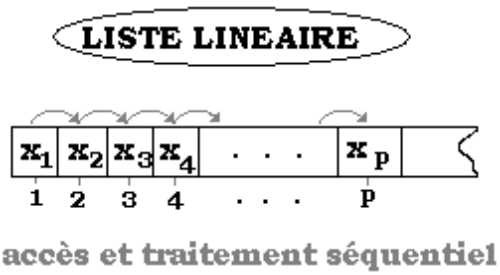
Spécification abstraite

Répertorions les fonctionnalités d'une liste en soulignant les verbes d'actions et les noms, à partir d'une description semi-formalisée:

- Il est possible dans une telle structure **d'ajouter** ou de **retirer** des éléments en n'importe quel point de la liste.
- **L'ordre** des éléments est primordial. Cet ordre est construit, non sur la valeur des éléments de la liste, mais sur les places (rangs)de ces éléments dans la liste.
- Le modèle mathématique choisi est la **suite finie** d'éléments de type T_0 : $(a_i)_{i \in I}$ où I est fini, totalement ordonné, $a_i \in T_0$
- Chaque place a un **contenu** de type T_0.
- Le nombre d'éléments d'une liste λ est appelé **longueur** de la liste. Si la liste est vide nous dirons que sa longueur est nulle (longueur = 0).
- On doit pouvoir effectuer au minimum (non exhaustif) les actions suivantes sur les éléments d'une liste λ : **accéder** à un élément de place fixée, **supprimer** un élément de place fixée, **insérer** un nouvel élément à une place fixée, etc ....



- C'est une structure de donnée séquentielle dans laquelle les données peuvent être traitées les unes à la suite des autres :



De cette description nous extrayons une spécification sous forme de TAD.

Ecriture syntaxique du TAD liste linéaire

TAD Liste
utilise : **N**, T_0, Place
Champs : $(a_1,.....,a_n)$ suite finie dans T_0
opérations :
liste_vide : $\rightarrow$ Liste
acces : Liste x N $\rightarrow$ Place
contenu : Place $\rightarrow T_0$
kème : Liste x **N** $\rightarrow T_0$
long : Liste $\rightarrow$ **N**
supprimer : Liste x **N** $\rightarrow$ Liste
inserer : Liste x **N** x $\rightarrow$ Liste
succ : Place $\rightarrow$ Place

préconditions :
acces(L,k) **def_ssi** $1 \leq k \leq$ long(L)
supprimer(L,k) **def_ssi** $1 \leq k \leq$ long(L)
inserer(L,k,e) **def_ssi** $1 \leq k \leq$ long(L) +1
kème(L,k) **def_ssi** $1 \leq k \leq$ long(L)

Fin-Liste

signification des opérations : (spécification abstraite)

acces(L,k) : opération générale d'accès à la position d'un élément de rang k de la liste L.
supprimer(L,k) : suppression de l'élément de rang k de la liste L.
inserer(L,k,e) : insérer l'élément e de T_0 , à la place de l'élément de rang k dans la liste L.
kième(L,k) : fournit l'élément de rang k de la liste.

spécification opérationnelle concrète

- La liste est représentée en mémoire par un ***tableau*** et un attribut de ***longueur***.
- Le kème élément de la liste est le kème élément du tableau.
- Le tableau est plus grand que la liste (il y a donc dans cette interprétation une contrainte sur la longueur. Notons Longmax cette valeur maximale de longueur de liste).

Il faut donc, afin de conserver la cohérence, ajouter deux préconditions au **TAD Liste** :

long(**L**) **def_ssi** long(**L**) $\leq$ Longmax
inserer(**L,k,e**) **def_ssi** $(1 \leq k \leq$ long(L) +1) $\wedge$ long(**L**) $\leq$ Longmax**)**

D'autre part la structure de tableau choisie permet un traitement itératif de l'opération kème (une autre spécification récursive de cet opérateur est possible dans une autre spécification opérationnelle de type dynamique).

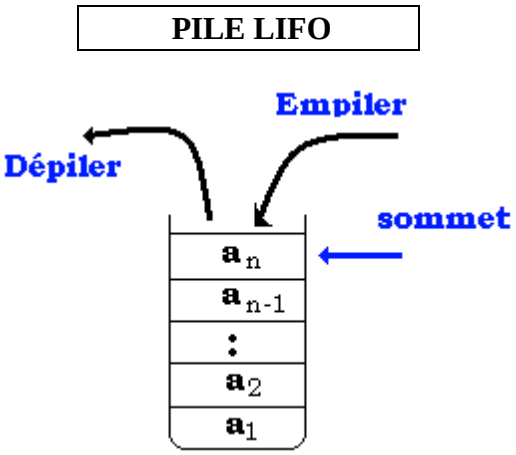
kème(L,k) = contenu(acces(L,k))

1.6 Le TAD pile LIFO (spécification abstraite et concrète)

Spécification abstraite

Répertorions les fonctionnalités d'une pile LIFO (Last In First Out) en soulignant les verbes d'actions et les noms, à partir d'une description semi-formalisée:

- C'est un modèle pour toute structure de données où l'on accumule des informations les unes après les autres, mais où l'on choisit de n'effectuer un traitement que sur le dernier élément entré. *Exemples* : pile de dossiers sur un bureau, pile d'assiettes, etc...
- Il est possible dans une telle structure **d'ajouter** ou de **retirer** des éléments uniquement au début de la pile.
- **L'ordre** des éléments est imposé par la pile. Cet ordre est construit non sur la valeur des éléments de la liste, mais sur les places (rangs)de ces éléments dans la liste. Cet ordre n'est pas accessible à l'utilisateur, c'est un élément privé.
- Le modèle mathématique choisi est la **suite finie** d'éléments de type T_0 :
  $(a_i)_{i\in I}$ où I est fini, totalement ordonné, $a_i \in T_0$
- La pile possède une place spéciale dénommée **sommet** qui identifie son premier élément et contient toujours le dernier élément entré.
- Le nombre d'éléments d'une pile LIFO P est appelé **profondeur** de la pile. Si la pile est vide nous dirons que sa profondeur est nulle (profondeur = 0).
- On doit pouvoir effectuer sur une pile LIFO au minimum (non exhaustif) les actions suivantes : voir si la pile **est vide**, **dépiler** un élément, **empiler** un élément, observer le **premier** élément sans le prendre, etc...



- C'est une structure de données séquentielle dans laquelle les données peuvent être traitées les unes à la suite des autres à partir du sommet

Ecriture syntaxique du TAD Pile LIFO

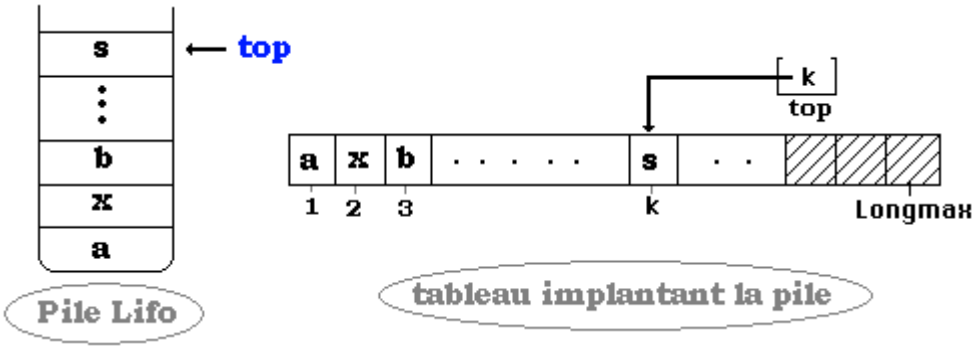
TAD PILIFO

utilise : T_0, Booléens
Champs : $(a_1,.....,a_n)$ suite finie dans T_0
opérations :
sommet : → PILIFO
Est_vide : PILIFO → Booléens
empiler : PILIFO x T_0 x sommet → PILIFO x sommet
dépiler : PILIFO x sommet → PILIFO x sommet x T_0
premier : PILIFO → T_0
préconditions :
dépiler(P) **def_ssi** est_vide(P) = **Faux**
premier(P) **def_ssi** est_vide(P) = **Faux**

FinTAD-PILIFO

spécification opérationnelle concrète

- La Pile est représentée en mémoire dans un *tableau*.
- Le *sommet* (noté **top**) de la pile est un *pointeur* sur la case du tableau contenant le début de pile. Les variations du contenu k de top se font au gré des empilements et dépilements.
- Le tableau est plus grand que la pile (il y a donc dans cette interprétation une contrainte sur la longueur, notons *Longmax* cette valeur maximale de profondeur de la pile).
- L'opérateur *empiler* : rajoute dans le tableau dans la case pointée par top un élément et top augmente d'une unité.
- L'opérateur *depiler* : renvoie l'élément pointé par **top** et diminue top d'une unité.
- L'opérateur *premier* fournit une copie du sommet pointé par top (la pile reste intacte).
- L'opérateur *Est_vide* teste si la pile est vide (vrai si elle est vide, faux sinon).

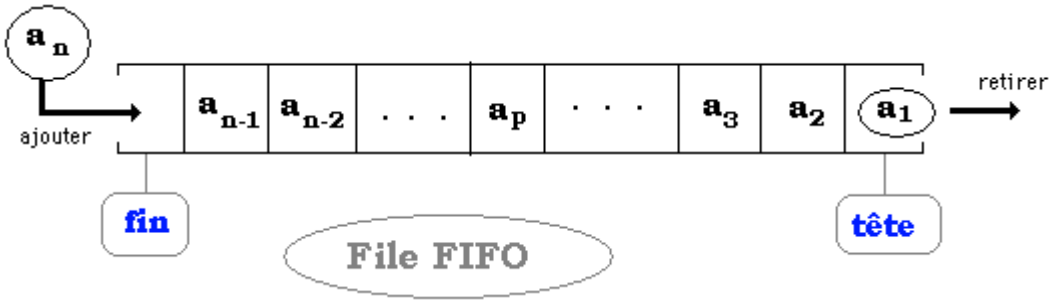


1.7 Le TAD file FIFO (spécification abstraite seule)

Spécification abstraite

Répertorions les fonctionnalités d'une file FIFO (First In First Out) en soulignant les verbes d'actions et les noms, à partir d'une description semi-formalisée:

- C'est un modèle pour toute structure de données où l'on accumule des informations les unes après les autres, mais où l'on choisit d'effectuer un traitement selon l'ordre d'arrivée des éléments, comme dans une file d'attente.
- *Exemples* :toutes les files d'attente, supermarchés, cantines , distributeurs de pièces, etc...
- Il est possible dans une telle structure **d'ajouter** des éléments à la fin de la file, ou de **retirer** des éléments uniquement au début de la file.
- **L'ordre** des éléments est imposé par la file. Cet ordre est construit non sur la valeur des éléments de la liste, mais sur les places (rangs)de ces éléments dans la liste. Cet ordre n'est pas accessible à l'utilisateur, c'est un élément privé.
- Le modèle mathématique choisi est la **suite finie** d'éléments de type T_0 :
  $(a_i)_{i\in I}$ où I est fini, totalement ordonné, $a_i \in T_0$
- La file possède deux places spéciales dénommées **tête** et **fin** qui identifient l'une son premier élément, l'autre le dernier élément entré.
- Le nombre d'éléments d'une file FIFO " F " est appelé **longueur** de la file ; si la file est vide nous dirons que sa longueur est nulle (longueur = 0).
- On doit pouvoir effectuer sur une file FIFO au minimum (non exhaustif) les actions suivantes : voir si la file **est vide**, **ajouter** un élément, **retirer** un élément, observer le premier élément sans le prendre, etc...



Ecriture syntaxique du TAD file FIFO

TAD FIFO
utilise : T_0, Booléens
Champs : $(a_1,.....,a_n)$ suite finie dans T_0
opérations :
tête : → FIFO
fin : → FIFO
Est_vide : FIFO → Booléens
ajouter : FIFO x T_0 x fin → PILIFO x fin
retirer : FIFO x tête → FIFO x tête x T_0
premier : FIFO → T_0

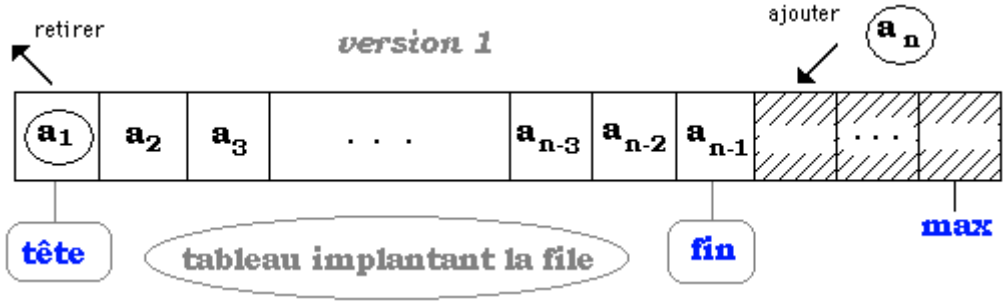
préconditions :
retirer(F) **def_ssi** est_vide(F) = **Faux**
premier(F) **def_ssi** est_vide(F) = **Faux**

FinTAD-FIFO

Spécification opérationnelle concrète

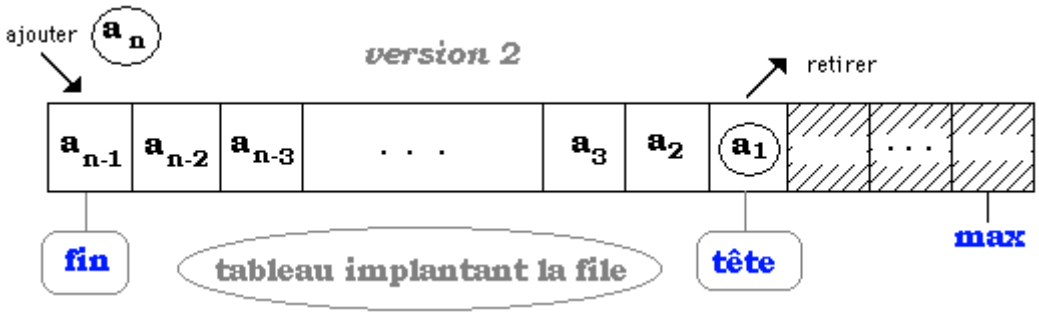
- La file est représentée en mémoire dans un *tableau*.
- La *tête* de la file est un pointeur sur la case du tableau contenant le début de la file. Les variations de la valeur de la tête se font au gré des ajouts et des retraits.
- La *fin* ne bouge pas , c'est le point d'entrée de la file.
- Le tableau est plus grand que la file (il y a donc dans cette interprétation une contrainte sur la longueur ; notons *max* cette valeur maximale de longueur de la file).
- L'opérateur *ajouter* : ajoute dans le tableau dans la case pointée par fin un élément et tête augmente d'une unité.
- L'opérateur *retirer* : renvoie l'élément pointé par tête et diminue tête d'une unité.
- L'opérateur *premier* fournit une copie de l'élément pointé par tête (la file reste intacte).
- L'opérateur *Est_vide* teste si la file est vide (vrai si elle est vide, faux sinon).

On peut ajouter après la dernière cellule pointée par l'élément fin comme le montre la figure ci-dessous :



dans ce cas retirer un élément en tête impliquera un décalage des données vers la gauche.

On peut aussi choisir d'ajouter à partir de la première cellule comme le montre la figure ci-dessous :



dans ce cas ajouter un élément en fin impliquera un décalage des données vers la droite.

4.2 : Types abstraits de données

Implantation avec Unit en Delphi

Plan du chapitre:

1. Types abstraits de données et Unités en Delphi

1. Types abstraits de données et Unités en Delphi

Dans cette partie nous adoptons un point de vue pratique dirigé par l'implémentation dans un langage accessible à un débutant des notions de type abstrait de donnée.

Nous allons proposer une écriture des TAD avec des unités Delphi (pascal version avec **Unit**) puis après avec des classes Delphi. Le langage Delphi en effet pour implanter des TAD, possède deux notions situées à deux niveaux d'abstraction différents :

- La notion d'unité (**Unit**) se situe au niveau 4 de la hiérarchie informationnelle citée auparavant : elle nous a déjà servi à implanter la notion de module. Une unité est donc une approximation relativement bonne pour le débutant de la notion de TAD.

- La notion classique de **classe**, contenue dans tout langage orienté objet, se situe au niveau 2 de cette hiérarchie **constitue une meilleure approche de la notion de TAD**.

En fait un TAD sera bien décrit par la partie **interface** d'une **unit** et se traduit presque immédiatement ; le travail de traduction des préconditions est à la charge du programmeur et se trouvera dans la partie *privée* de la unit (**implementation**)

1.1 Traduction générale TAD → Unit pascal

Nous proposons un tableau de correspondance pratique entre la signature d'un TAD et la partie interface d'une Unit :

syntaxe du TAD	*squelette de la unit associée*
TAD Truc	**Unit Truc ;**
utilise TAD_1 , TAD_2,...,TAD_n	**interface** **uses** TAD_1,...,TAD_n ;
champs **.....**	**const ....** **type ....** **var ....**
opérations Op1 : E x F → G Op2 : E x F x G → H x S	**procedure** Op1(x :E ;y :F ;**var** z :G) ; **procedure** Op2(x :E ;y :F ; z :G ; **var** t :H ; **var** u :S) ; **.....**
FinTAD-Truc	**end.**

Reste à la charge du programmeur l'écriture, dans la partie **implementation** de la **unit,** des blocs de code associés à chacune des procédures décrites dans la partie **interface** :

```
Unit Truc ;
interface
   uses TAD1,...,TADn ;
   const ....
   type ....
   var ....
   procedure Op1(x :E ;y :F ;var z :G) ;
   procedure Op2(x :E ;y :F ; z :G ; var t :H ; var u :S) ;
   .....

implementation
procedure Op1(x :E ;y :F ;var z :G) ;
begin
 ..... code
end ;

procedure Op2(x :E ;y :F ; z :G ; var t :H ;var u :S) ;
begin
 ..... code
end ;
etc...
end.
```

1.2 Exemples de Traduction TAD → Unit pascal

Le TAD Booléens implanté sous deux spécifications concrètes en pascal avec deux types scalaires différents.

Spécification opérationnelle concrète n°1

Les constantes du type Vrai, Faux sont représentées par deux constantes pascal dans un type intervalle sur les entiers.

```
Const   vrai=1 ; faux=0
type    Booleens=faux..vrai ;
```

Voici l'interface de la unit traduite et le TAD :

TAD : Booléens **Champs** : **Opérations** : Vrai : → Booléens Faux : → Booléens Et : Booléens x Booléens → Booléens Ou : Booléens x Booléens → Booléens Non : Booléens → Booléens **FINTAD**-Booléens	**Unit** Bool ; **interface** **Const** vrai=1 ; faux=0 **type** Booleens = faux..vrai ; **function** Et (x,y :Booleens) :Booleens ; **function** Ou (x,y :Booleens) :Booleens ; **function** Non(x :Booleens) :Booleens ;

Spécification opérationnelle concrète n°2

Les constantes du type Vrai, Faux sont représentées par deux identificateurs pascal dans un type énuméré.

type Booleens=(faux,vrai) ;

Voici l'interface de la unit traduite et le TAD :

TAD : Booléens **Champs** : **Opérations** : Vrai : → Booléens Faux : → Booléens Et : Booléens x Booléens → Booléens Ou : Booléens x Booléens → Booléens Non : Booléens → Booléens **FINTAD**-Booléens	**Unit** Bool ; **interface** **type** Booleens=(faux,vrai) ; **function** Et (x,y :Booleens) :Booleens ; **function** Ou (x,y :Booleens) :Booleens ; **function** Non(x :Booleens) :Booleens ;

Nous remarquons la similarité des deux spécifications concrètes :

Implantation avec des entiers	Implantation avec des énumérés
Unit Bool ; **interface** **Const** vrai=1 ; faux=0 **type** Booleens = faux..vrai ; **function** Et (x,y :Booleens) :Booleens ; **function** Ou (x,y :Booleens) :Booleens ; **function** Non(x :Booleens) :Booleens ;	**Unit** Bool ; **interface** **type** Booleens = (faux,vrai) ; **function** Et (x,y :Booleens) :Booleens ; **function** Ou (x,y :Booleens) :Booleens ; **function** Non(x :Booleens) :Booleens ;

1.3 Variations sur les spécifications d'implantation

Cet exercice ayant été proposé à un groupe d'étudiants, nous avons eu plusieurs genres d'implantation des opérations : et, ou, non. Nous exposons au lecteur ceux qui nous ont parus être les plus significatifs :

Implantation d'après spécification concrète n°1

Fonction Et	Fonction Et
function Et (x,y :Booleens) :Booleens ; **begin** Et := x * y **end** ;	**function** Et (x,y :Booleens) :Booleens ; **begin** **if** x=Faux **then** Et :=Faux **else** Et := Vrai **end** ;

Fonction Ou	Fonction Ou

function Ou (x,y :Booleens) :Booleens ;
begin
  Ou :=x+y - x*y
end ;
	function Ou (x,y :Booleens) :Booleens ;
	begin
	  if x=Vrai **then** Ou := Vrai
	  else Ou := Faux
	end ;

Fonction Non	Fonction Non

function Non(x :Booleens) :Booleens ;
begin
  Non := 1-x
end ;
	function Ou (x,y :Booleens) :Booleens ;
	begin
	  if x=Vrai **then** Ou := Vrai
	  else Ou := Faux
	end ;

Dans la colonne de gauche de chaque tableau, l'analyse des étudiants a été dirigée par le choix de la spécification concrète sur les entiers et sur un modèle semblable aux fonctions indicatrices des ensembles. Ils ont alors cherché des combinaisons simples d'opérateurs sur les entiers fournissant les valeurs adéquates.

Dans la colonne de droite de chaque tableau, l'analyse des étudiants a été dirigée dans ce cas par des considérations axiomatiques sur une algèbre de Boole. Ils se sont servis des propriétés d'absorbtion des éléments neutres de la loi " *ou* " et de la loi " *et* ". Il s'agit là d'une structure algébrique abstraite.

Influence de l'abstraction sur la réutilisation

A cette étape du travail nous avons demandé aux étudiants quel était, s'il y en avait un, le meilleur choix d'implantation quant à sa réutilisation pour l'implantation d'après la spécification concrète n°2.

Les étudiants ont compris que la version dirigée par les axiomes l'emportait sur la précédente, car sa qualité d'abstraction due à l'utilisation de l'axiomatique a permis de la réutiliser sans aucun changement dans la partie **implementation** de la unit associée à spécification concrète n°2 (en fait toute utilisation des axiomes d'algèbre de Boole produit la même efficacité).

Conclusion :

l'abstraction a permis ici une réutilisation totale et donc un gain de temps de programmation dans le cas où l'on souhaite changer quelle qu'en soit la raison, la spécification concrète.

Dans les exemples qui suivent, la notation $\cong$. indique la traduction en un squelette en langage pascal.

1.4 Exemples d'implantation de la liste linéaire

spécification proposée en pseudo-Pascal :

Liste ≅	**type** Liste=**record** t : **array[**1.. Longmax**] of** T_0; long : 0.. Longmax **end**;
liste_vide ≅	**var** L : Liste (avec L.long :=0)
acces ≅	**var** k : integer; L : liste (*adresse(L.t[k])*)
contenu ≅	**var** k : integer; L : liste *(val(adresse(L.t[k])))*
kème ≅	**var** k : integer; L : liste *(kème(L,k)* ≅ L.t[k])
long ≅	**var** L : liste *(long* ≅ L.long)
succ ≅	adresse(L.t[k])+1 *c-à-dire (L.t[k+1])*
supprimer ≅	**procedure** supprimer(**var** L : Liste ; k : 1.. Longmax); **begin** ..... **end;** *{supprimer}*
inserer ≅	**procedure** inserer(**var** L : Liste ; k : 1.. Longmax; x : T_0); **begin** ..... **end;** *{inserer}*

La précondition de l'opérateur **supprimer** peut être ici implantée par le test :

if k<=long(L) **then** ....

La précondition de l'opérateur **insérer** peut être ici implantée par le test :

if (long(L) < Longmax) **and** (k<=Long(L)+1) **then** .....

Les deux objets **acces** et **contenu** ne seront pas utilisés en pratique, car le tableau les implante automatiquement d'une manière transparente pour le programmeur.

Le reste du programme est laissé au soin du lecteur qui pourra ainsi se construire sur sa machine , une base de types en Pascal-Delphi de base.

Nous pouvons " **enrichir** " le TAD Liste en lui adjoignant deux opérateurs test et rechercher (rechercher un élément dans une liste). Ces adjonctions ne posent aucun problème. Il suffit pour cela de rajouter au TAD les lignes correspondantes :

opérations
Test : Liste x T_0 → Booléen
rechercher : Liste x T_0 → Place
précondition
rechercher(**L**,**e**) **def_ssi** Test(**L**,**e**) = **V**

Le lecteur construira à titre d'exercice l'implantation Pascal-Delphi de ces deux nouveaux opérateurs en étendant le programme déjà construit. Il pourra par exemple se baser sur le schéma de représentation Pascal suivant :

```
function Test(L : liste; e : T0):Boolean;
begin
{il s'agit de tester la présence ou non de e dans la liste L}
end;

procedure rechercher(L : liste ; x : T0; var rang : integer);
begin
  if Test(L,x) then
{il s'agit de fournir le rang de x dans la liste L}
  else
rang:=0
end;
```

1.5 Exemples d'implantation de la pile LIFO

Nous allons utiliser un **tableau** avec une case supplémentaire permettant d'indiquer que le fond de pile est atteint (la case 0 par exemple, qui ne contiendra jamais d'élément).

spécification proposée en pseudo-Pascal :

Pilifo ≅	**type** Pilifo=**record** t : **array[** 0.. Longmax **] of** T_0; sommet: 0.. Longmax **end**;
depiler ≅	**procedure** depiler (**var** Elt : T_0 ;**var** P : Pilifo) ;
empiler ≅	**procedure** empiler(Elt : T_0 ;**var** P : Pilifo) ;

premier ≅	**procedure** premier(**var** Elt : T_0 ; P : Pilifo) ; *(on pourra utiliser une function renvoyant un T_0, si le type T_0 s'y prête !)*
Est_vide ≅	**function** Est_vide(P : Pilifo) : boolean ;

Le contenu des procédures et des fonctions est laissé au lecteur à titre d'exercice.

Remarque :

Il est aussi possible de construire une spécification opérationnelle à l'aide du TAD Liste en remplaçant dans l'étude précédente le mot " tableau " par le mot " liste ". Il est vivement conseillé au lecteur d'écrire cet exercice en Delphi pour bien se convaincre de la différence entre les niveaux d'abstractions.

1.6 Exemples d'implantation de la file FIFO

Nous allons utiliser ici aussi un **tableau** avec une case supplémentaire permettant d'indiquer que la file est vide (la case 0 par exemple, qui ne contiendra jamais d'élément).

spécification proposée en pseudo-Pascal :

Fifo ≅	**type** Fifo=**record** t : **array[** 0.. Longmax **] of** T_0; sommet: 0.. Longmax **end**;
retirer ≅	**procedure** retirer(**var** Elt : T_0 ;**var** F : Fifo) ;
ajouter ≅	**procedure** ajouter(Elt : T_0 ;**var** F : Fifo) ;
premier ≅	**procedure** premier(**var** Elt : T_0 ; P : Fifo) ; *(on pourra utiliser une function renvoyant un T_0, si le type T_0 s'y prête !)*
Est_vide ≅	**function** Est_vide(P : Fifo) : boolean ;

Le contenu des procédures et des fonctions est laissé au lecteur à titre d'exercice.

Remarque :

Comme pour le TAD Pilifo, il est aussi possible de construire une spécification opérationnelle du TAD FIFO à l'aide du TAD Liste en remplaçant dans l'étude précédente le mot " tableau " par le mot " liste ".

Une solution d'implantation de la liste linéaire en Unit Delphi

```
Unit UListchn;

 interface
const
  max_elt = 100;
type
  T0 = integer;
  liste =
  record
    suite: array[1..max_elt] of T0;
    long: 0..max_elt;
    init_ok:char;
  end;

  function longueur (L: liste): integer;
  procedure supprimer (var L: liste; k: integer);
  procedure inserer (var L: liste; k: integer; x: T0);
  function kieme (L: liste; n: integer): T0;
  function Test (L: liste; x: T0): boolean;
  procedure Rechercher (L: liste; x: T0; var place: integer);

implementation

procedure init_liste(var L:liste);
  {initialisation obligatoire}
  begin
    with L do
    begin
      long:=0;
      init_ok:='#'
    end
  end;

  function Est_vide(L:liste):boolean;
  begin
    if L.init_ok<>'#' then
    begin
      writeln('>>> Gestionnaire de Liste: Liste non
initialisée !! (erreur fatale)');
      halt
    end
    else
      if L.long=0 then
      begin
        Est_vide:=true;
        writeln('>>> Gestionnaire de Liste: Liste vide')
      end
      else
        est_vide:=false
  end;
```

```
function longueur (L: liste): integer;
  begin
    longueur := L.long
  end;

  procedure supprimer (var L: liste; k: integer);
  var
    n: 0..max_elt;
    i: 1..max_elt;
    begin
      if not Est_vide(L) then
        if k>1 then
        begin
          n := longueur(L);
          if (1<=k)and(k <= n) then
          begin
            for i := k to n - 1 do
              L.suite[i] := L.suite[i + 1];
            L.long := n - 1
          end
        end
        else
          l.long :=0;
    end;{supprimer}
```

```
procedure inserer (var L: liste; k: integer; x: T0);
 var
  n: 0..max_elt;
  i: 1..max_elt;
  begin
   if not Est_vide(L) then
   begin
    n := longueur(L);
    if (n < max_elt) and (k <= n + 1) then
    begin
     for i := n downto k do
      L.suite[i + 1] := L.suite[i];
     L.suite[k] := x
    end;
    L.long := L.long + 1
   end
   else
   begin
    L.suite[1]:=x;
    L.long :=1
   end
  end;

 function kieme (L: liste; n: integer): T0;
 begin
  if not Est_vide(L) then
  begin
   kieme := L.suite[n]
  end
 end;
```

```
function Test (L: liste; x: T0): boolean;
 {teste la presence ou non de x dans la liste L}
 var
  present:boolean;
  rang:integer;
  begin
   if not Est_vide(L) then
   begin
    Test_Recherche(L,x,present,rang);
    Test:=present
   end
  end;
```

```
procedure Test_Recherche(L:liste;x:T0;var
trouve:boolean;var rang:integer);
 var
  fini,present:boolean;
  i,n:integer;
  begin
   if not Est_vide(L) then
   begin
    fini := false;
    i := 1;
    n:=L.long;
    present := false;
    while not fini and not present do
    begin
     if i <= n then
      if L.suite[i] <> x then
       i := i + 1
      else
       present := true
     else
      fini := true
    end;
    if present then
    begin
     {valeur x trouvée a l'indice:i}
     trouve:=true;
     rang:=i
    end
    else
    begin
     {cette valeur x n'est pas dans le tableau}
     trouve:=false;
     {on n'affecte aucune valeur a rang !! }
    end
   end
  end;
```

```
procedure Rechercher (L: liste; x: T0; var place:
integer);
 var
  present:boolean;
  begin
   if not Est_vide(L) then
   begin
    Test_Recherche(L,x,present,place);
    if not present then
     place:=0
   end
  end;
end. { fin de la Unit UListchn }
```

Une solution d'implantation de la pile Lifo en Unit Delphi

```
Unit Upilifo;

 interface

const
  max_elt = 100;
  fond = 0;
type
  T0 = integer;
  pile =
  record
    suite: array[0..max_elt] of T0;
    sommet: 0..max_elt;
  end;

  function Est_Vide(P:pile):boolean;
  procedure Empiler(elt:T0;var P:pile);
  procedure Depiler(var elt:T0;var P:pile);
  function premier (P: pile): T0;

implementation

  function Est_Vide(P:pile):boolean;
  begin
   if P.sommet = fond then
     Est_Vide := true
   else
     Est_Vide := false
  end;

  procedure Empiler(elt:T0;var P:pile);
  begin
    P.sommet := P.sommet + 1;
    P.suite[P.sommet] := elt
  end;

  procedure Depiler(var elt:T0;var P:pile);
   begin
    if not Est_Vide(P) then
    begin
      elt := P.suite[P.sommet];
      P.sommet := P.sommet - 1
    end
    {la precondition est implantee ici par
     le test if not est_vide(p) then ...    }
   end;

   function premier (P: pile): T0;
   begin
    if not Est_Vide(P) then
      premier := P.suite[P.sommet]
    { la precondition est implantee ici par
      le test if not est_vide(p) then ...    }
   end;

  end. { fin de la Unit Upilifo }
```

4.3 : 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 :

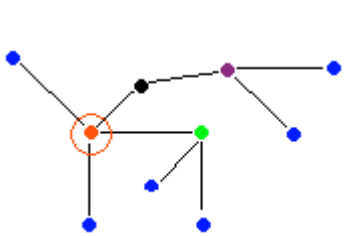


fig-1

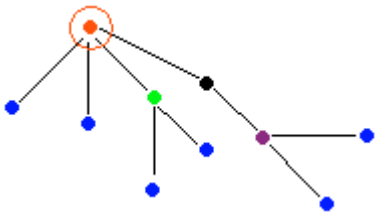


fig-2

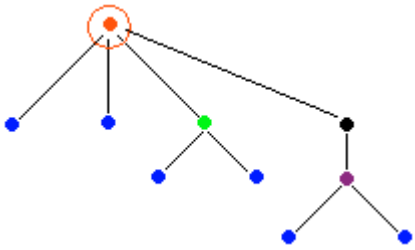


fig-3

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**.

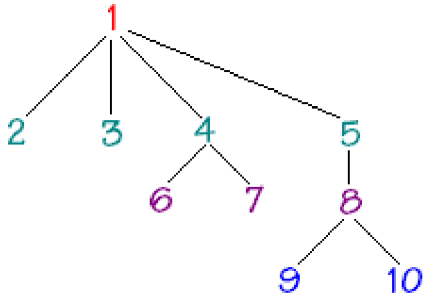
Enfin certains arbres particuliers nommés arbres binaires sont les plus utilisés en informatique et les plus simples à étudier. En outre il est toujours possible de "**binariser**" un arbre non binaire, ce qui nous permettra dans ce chapitre de n'étudier que les structures d'arbres binaires.

1.1 Vocabulaire employé sur les arbres

Etiquette

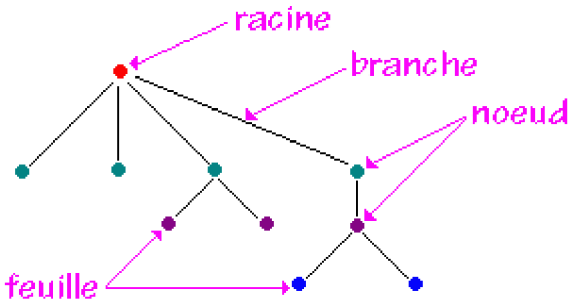
Un arbre dont tous les noeuds sont nommés est dit **étiqueté**. L'étiquette (ou nom du sommet) représente la "valeur" du noeud ou bien l'information associée au noeud.
Ci-dessous un arbre étiqueté dans les entiers entre 1 et 10 :

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



Nous rappellons la terminologie de base sur les arbres:

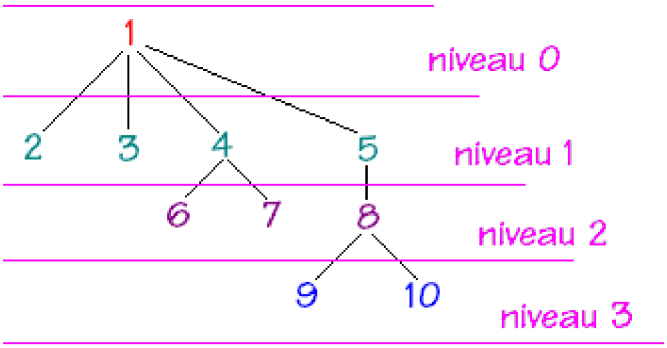
Racine , noeud, branche , feuille



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 :



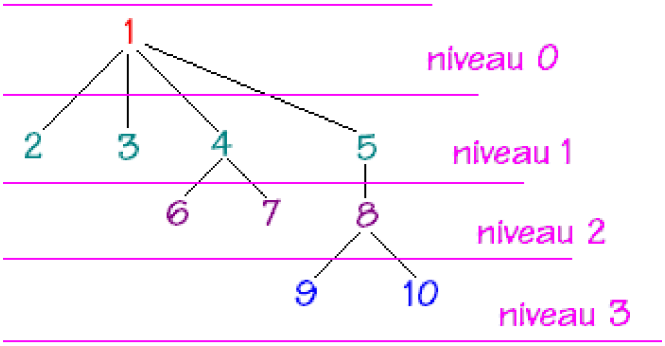
Pour atteindre le noeud étiqueté 9 , il faut parcourir le lien 1--5, puis 5--8, puis enfin 8--9 soient 4 noeuds donc 9 est de profondeur ou de hauteur égale à 4, soit **h(9) = 4**.
Pour atteindre le noeud étiqueté 7 , il faut parcourir le lien 1--4, et enfin 4--7, donc 7 est de profondeur ou de hauteur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



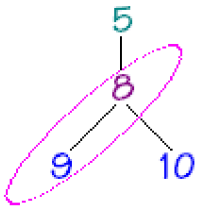
Chemin du noeud 10 = (1,5,8,10)
Chemin du noeud 9 = (1,5,8,9)
.....
Chemin du noeud 7 = (1,4,7)
Chemin du noeud 5 = (1,5)
Chemin du noeud 1 = (1)

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

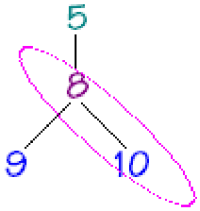
h(X) = NbrNoeud(Chemin(X)).

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

Parents, enfants

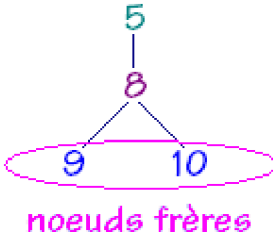


9 est l'enfant de 8
8 est le parent de 9



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.**

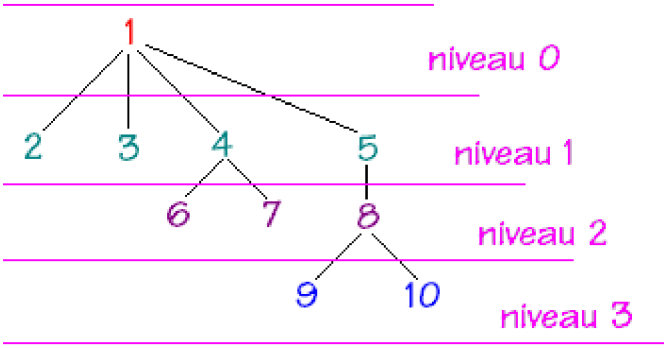
On parle aussi d'ascendant, de descendant ou de fils pour évoquer des relations entres les noeuds d'un même arbre reliés entre eux.

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 :

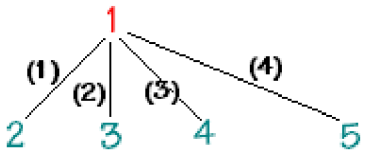


Calculons récursivement la hauteur du noeud 9, notée h(9) :
h(9) = 1+h(8)
h(8) = 1+h(5)
h(5) = 1+h(1)
h(1) = 1 = h(5)=2 = h(8)=3 = h(9)=4

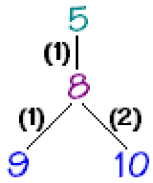
Degré d'un noeud

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

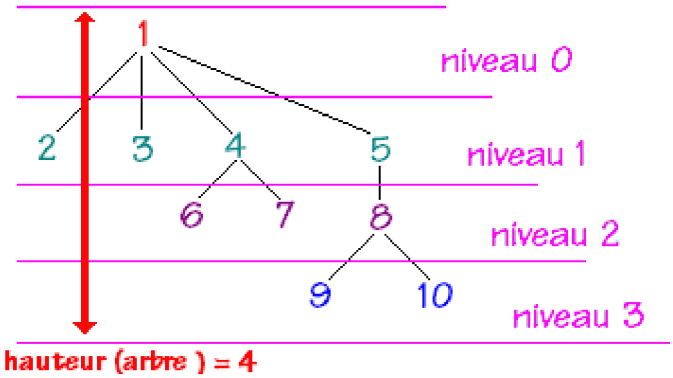
Par définition c'est le **nombre de noeuds du chemin le plus long** dans l'arbre.; on dit aussi profondeur de l'arbre.

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

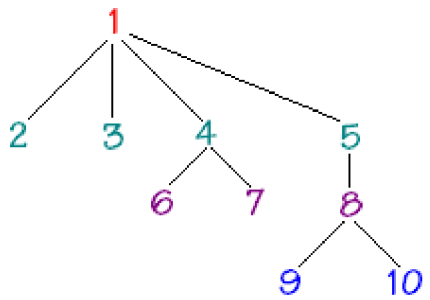
La hauteur de l'arbre ci-dessous :



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

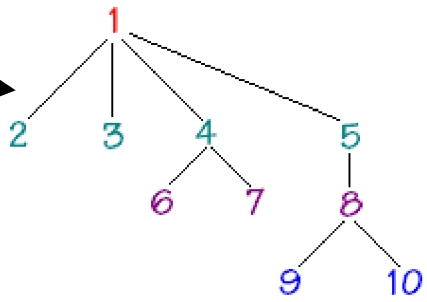
La valeur maximale est 4 , donc cet arbre est de degré 4.

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.

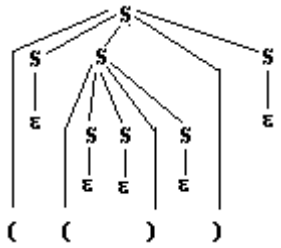
Arbre de dérivation d'un mot dans une grammaire

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 :



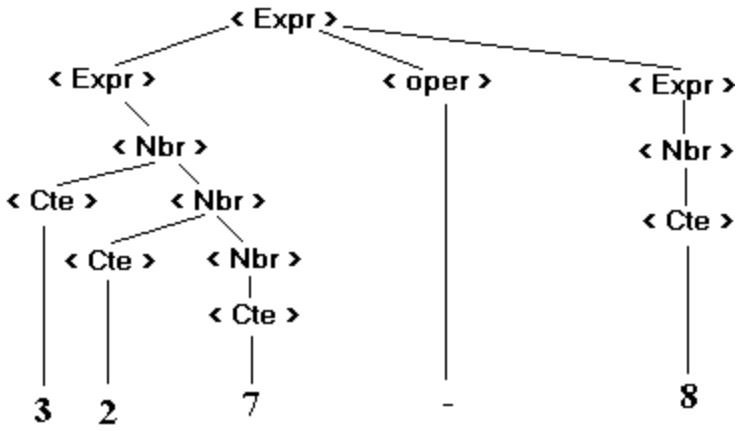
Exemple - 2
arbre abstrait

Soit la grammaire Gexp :

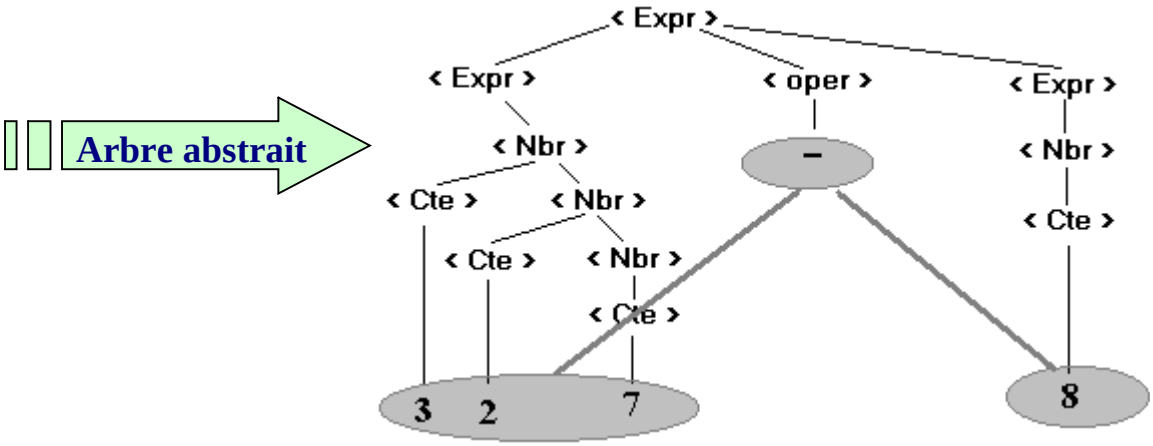
G_{exp} = (V_N,V_T,Axiome,Règles)
V_T = { **0**, ..., **9**, **+**,**-**, **/**, *****, **)**, **(**}
V_N = { ⟨ Expr ⟩, ⟨ Nbr ⟩, ⟨ Cte ⟩, ⟨ Oper ⟩}
Axiome : ⟨ Expr ⟩
Règles :
1 :⟨ Expr ⟩→⟨ Nbr ⟩ | **(**⟨Expr ⟩**)**| ⟨ Expr ⟩⟨ Oper ⟩⟨ Expr ⟩
2 :⟨ Nbr ⟩→⟨ Cte ⟩ | ⟨ Cte ⟩⟨ Nbr ⟩
3 :⟨ Cte ⟩→**0** | **1**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

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

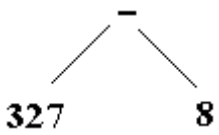
Soit son arbre de dérivation dans G_{exp} :



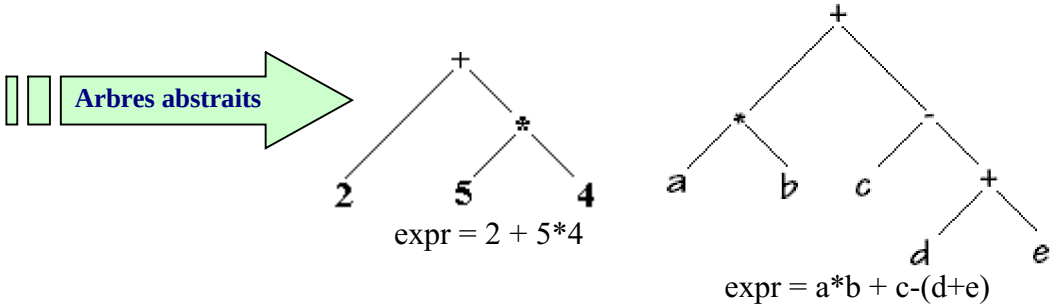
L'arbre obtenu ci-dessous en grisé à partir de l'arbre de dérivation s'appelle l'arbre abstrait du mot " 327-8 " :



On note ainsi cet arbre abstrait :

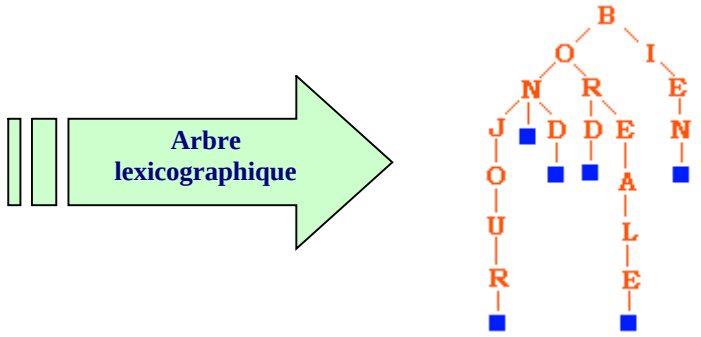


Voici d'autres abres abstraits d'expressions arithmétiques :

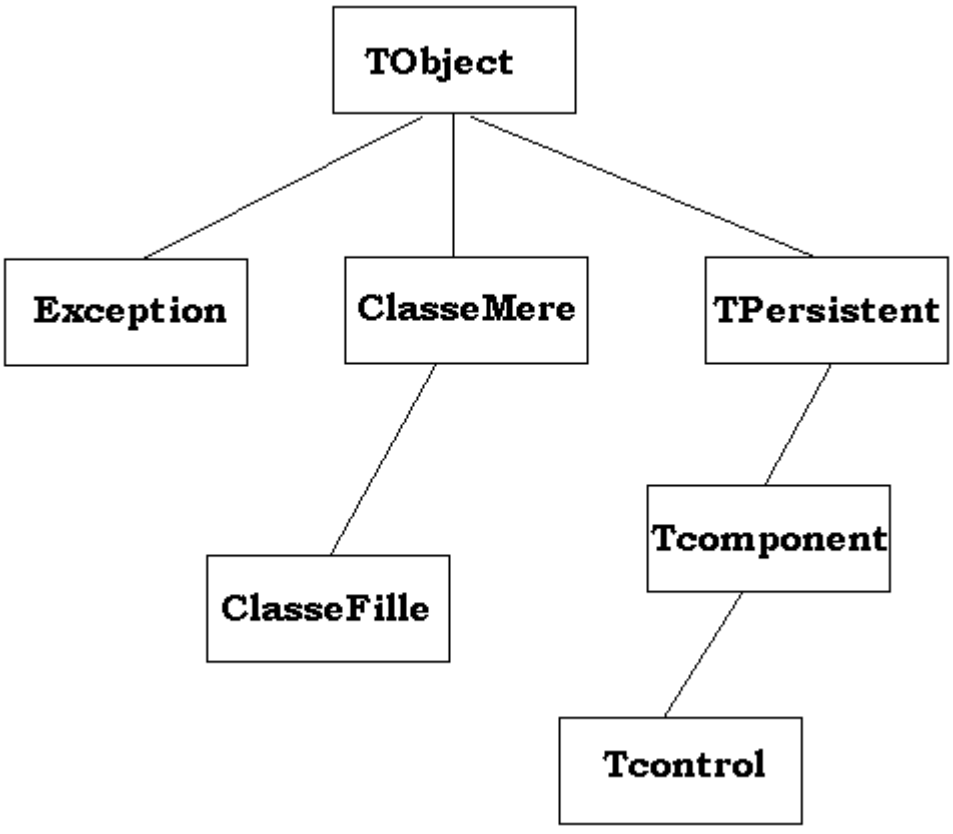


Arbre lexicographique

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 :



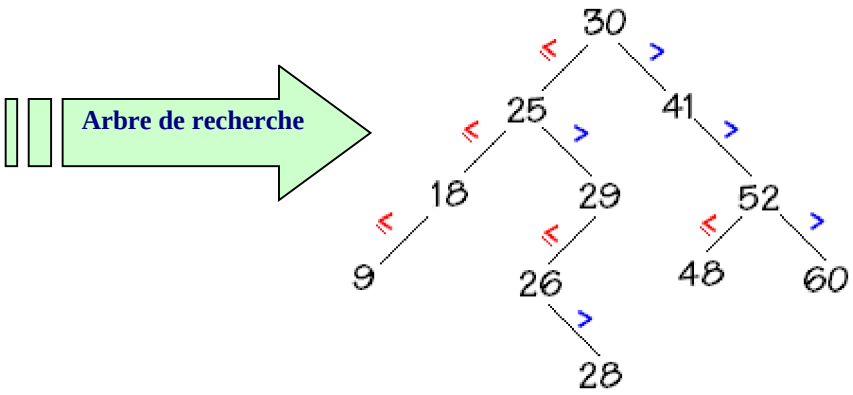
Arbre d'héritage en Delphi



Arbre de recherche

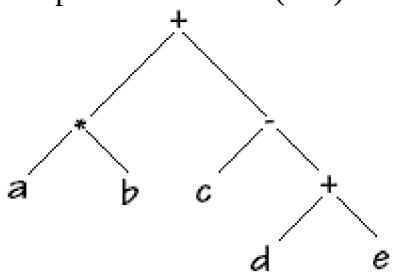
Voici à titre d'exemple que nous étudierons plus loin en détail, un arbre dont les noeuds sont de degré 2 au plus et qui est tel que pour chaque noeud la valeur de son enfant de gauche lui est inférieure ou égale, la valeur de son enfant de droite lui est strictement supérieure.

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

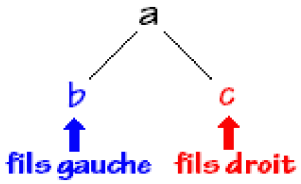


2 Les arbres binaires

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.

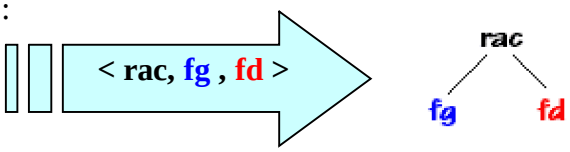
2.1 TAD d'arbre binaire

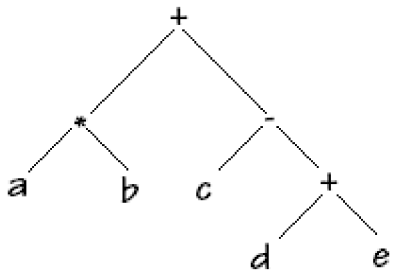
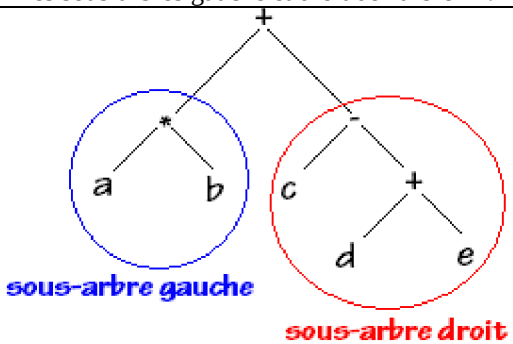
Afin d'assurer une cohérence avec les autres structures de données déjà vues (**liste, pile, file**) nous proposons de décrire une abstraction d'un arbre binaire avec un TAD. Soit la signature du TAD d'arbre binaire :

```
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-PILIFO
```

- 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

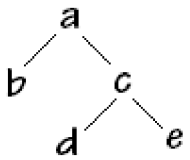
Spécification concrète
Un noeud est une structure statique contenant 3 éléments :

- 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

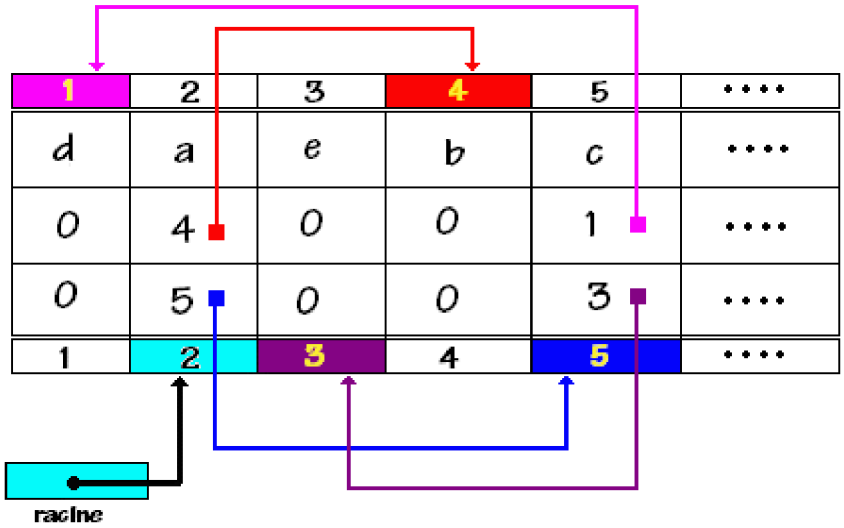
Soit l'arbre binaire ci-contre :



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

Spécification d'implantation en Delphi

Nous proposons d'utiliser les déclarations suivantes :

```
const
  taille = n;
type
  Noeud = record
              info : T0;
              filsG , filsD : 0..taille ;
          end;
 Tableau = Array[1..taille] of Noeud ;
 ArbrBin = record
              ptrRac : 0..taille;
              table : Tableau ;
          end;
Var
      Tree : ArbrBin ;
```

Explications :

Lorsque **Tree.ptrRac = 0** on dit que l'arbre est vide.
L'accès à la racine de l'arbre s'effectue ainsi : **Tree.table[ptrRac]**
L'accès à l'info de la racine de l'arbre s'effectue ainsi :
Tree.table[ptrRac].info
L'accès au fils gauche de la racine de l'arbre s'effectue ainsi :

```
var   ptr:0..taille ;

ptr := Tree.table[ptrRac].filsG;
Tree.table[ptr] ....
```

L'insertion ou la suppression d'un noeud dans l'arbre ainsi représenté s'effectue directement dans une cellule du tableau. Il faudra donc posséder une structure (de liste, de pile ou de file par exemple) permettant de connaître les cellules libres ou de ranger une cellule nouvellement libérée. Une telle structure se dénomme "**espace libre**".

L'insertion se fera dans la première cellule libre, l'espace libre diminuant d'une unité.
La suppression rajoutera une nouvelle cellule dans l'espace libre qui augmentera d'une unité.

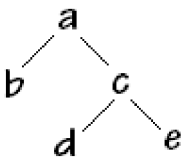
Implantation avec des variables dynamiques

Spécification concrète
Le noeud reste une structure statique contenant 3 éléments dont 2 sont dynamiques :

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

Exemple

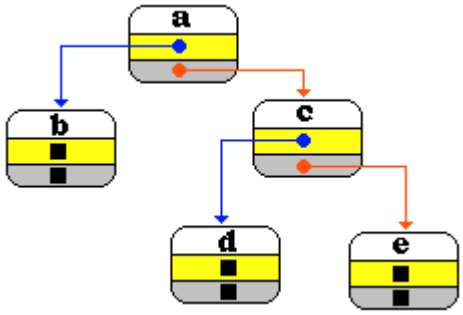
Soit l'arbre binaire ci-contre :



Selon l'implantation choisie, par hypothèse de départ, la référence vers la racine **pointe vers** la structure statique (le noeud) < a, ref vers b, ref vers c >

Nous avons :

ref 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 Delphi

Nous proposons d'utiliser les déclarations de variables dynamiques suivantes :

type ArbrBin = ^Noeud ; Noeud = **record** info : T0; filsG , filsD : ArbrBin ; **end**; **Var** Tree : ArbrBin ;	*Explications :* Lorsque **Tree = nil** on dit que l'arbre est vide. L'accès à la racine de l'arbre s'effectue ainsi : **Tree** L'accès à l'info de la racine de l'arbre s'effectue ainsi : **Tree^.info** L'accès au fils gauche de la racine de l'arbre s'effectue ainsi : **Tree^.filsG** L'accès au fils gauche de la racine de l'arbre s'effectue ainsi : **Tree^.filsD**

Nous noterons une simplification notable des écritures dans cette implantation par rapport à l'implantation dans un tableau statique. Ceci provient du fait que la **structure d'arbre est définie récursivement** et que la notion de variable dynamique permet une définition récursive donc plus proche de la structure.

Implantation avec une classe Delphi

Nous livrons ci-dessous une écriture de la signature et l'implémentation minimale d'une classe d'arbre binaire nommée TreeBin en Delphi (l'implémentation complète est à construire lors des exercices sur les classes) :

```
TreeBin = class
    public
      Info : string;
      filsG , filsD : TreeBin;
      constructor CreerTreeBin(s:string);overload;
      constructor CreerTreeBin(s:string; fg , fd : TreeBin);overload;
      destructor Liberer;
  end;
```

2.3 Arbres binaires de recherche

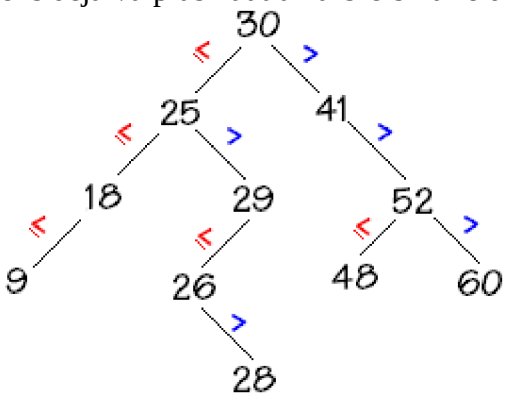
- Nous avons étudié prédédement des algorithmes de recherche en table, en particulier la recherche dichotomique dans une table triée dont la recherche s'effectue en **O(log(n))** comparaisons.
- Toutefois lorsque le nombre des éléments varie (ajout ou suppression) ces ajouts ou suppressions peuvent nécessiter 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.

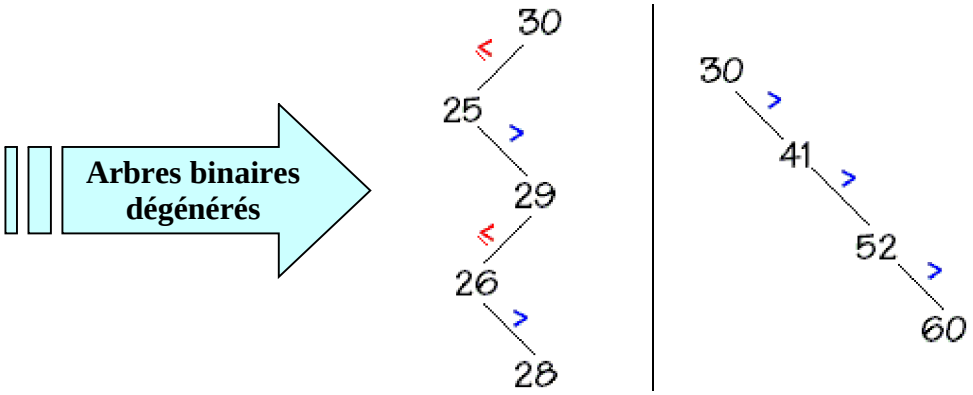
Un arbre binaire de recherche satisfait aux critères suivants :

- 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).

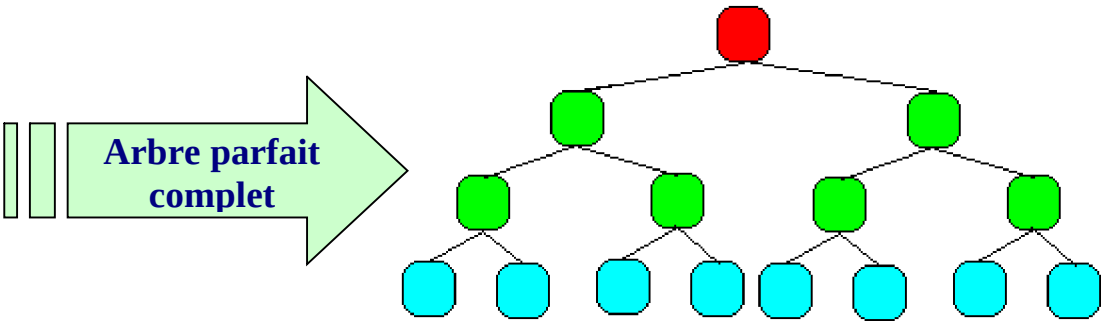
Il faudra donc utiliser une catégorie spéciale d'arbres binaires qui restent équilibrés (leurs feuilles sont sur 2 niveaux au plus) pour assurer une recherche au pire en O(log(n)).

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

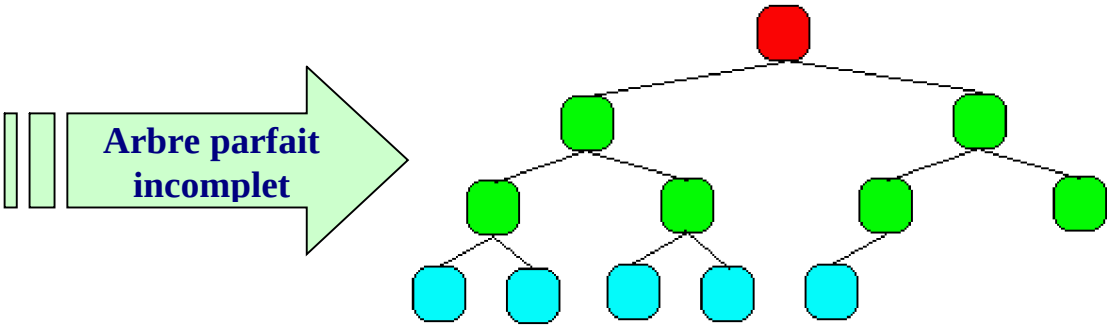
Arbre parfait

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.



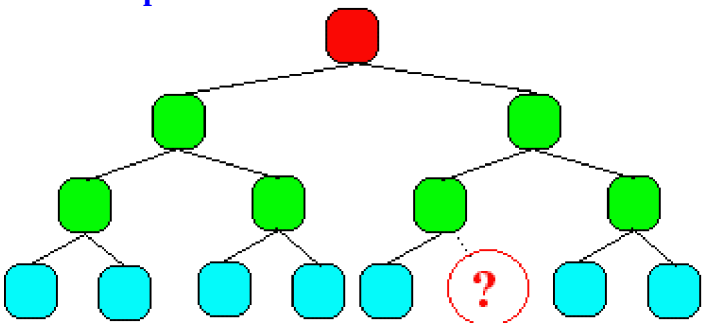
parfait complet : *le dernier niveau est complet car il contient tous les enfants*

un abre **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)*



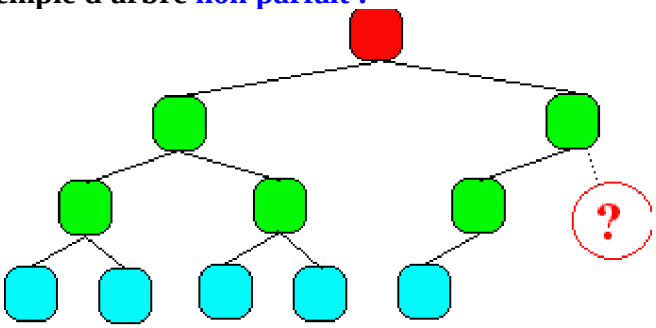
parfait icomplet: *le dernier niveau est incomplet car il manque 3 enfants à la droite*

Exemple d'arbre non parfait :



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

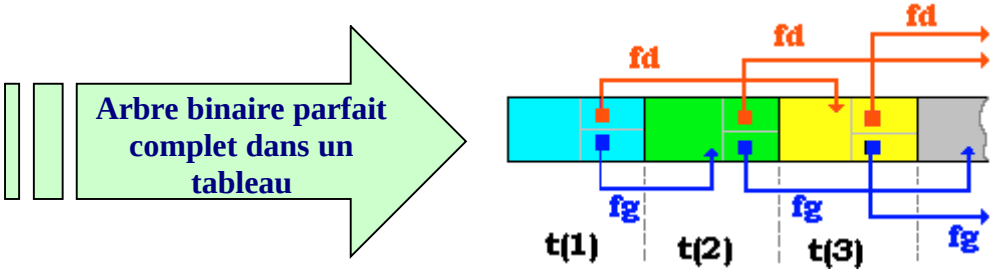
Autre exemple d'arbre non parfait :



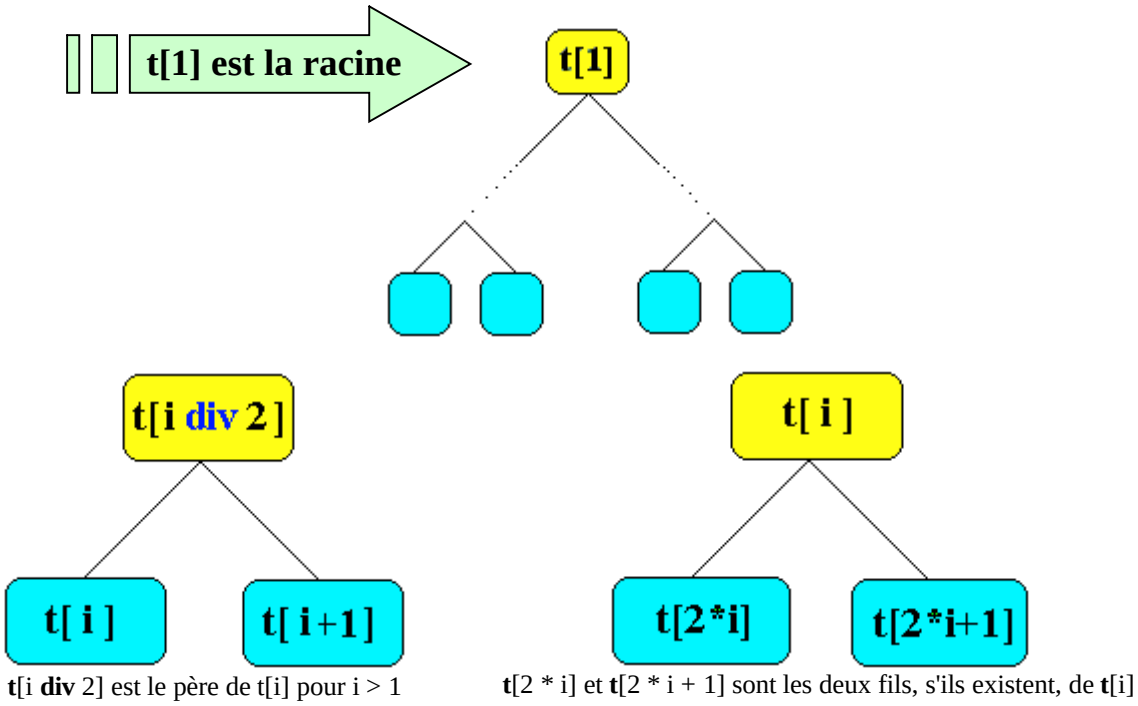
*(**non parfait :** les feuilles sont bien regroupées à gauche, mais il manque 1 enfant à l'avant dernier niveau)*

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 :

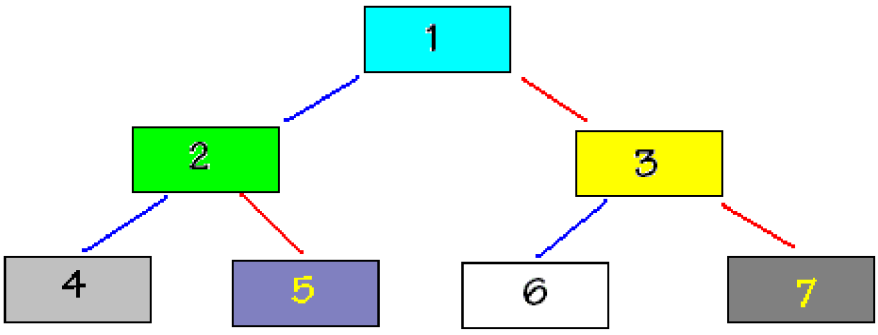


t[i **div** 2] est le père de t[i] pour i > 1

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.

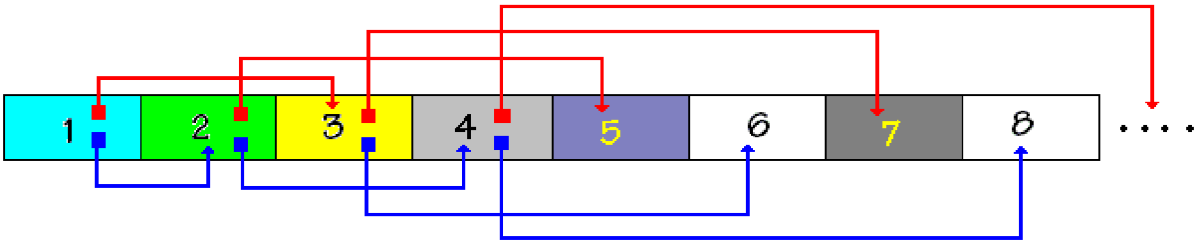
Exemple de rangement d'un tel arbre dans un tableau
(on a figuré l'indice de numérotation hiérarchique de chaque noeud dans le rectangle associé au noeud)



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).

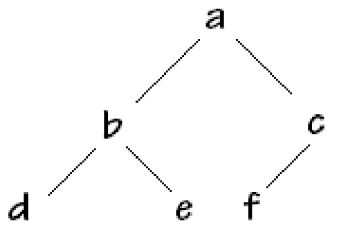
Dans cette disposition le passage d'un noeud de numéro **k** (indice dans le tableau) vers son fils gauche s'effectue par calcul d'indice, le fils gauche se trouvera dans la cellule d'index **2*k** du tableau, son fils droit se trouvant dans la cellule d'index **2*k + 1** du tableau. Ci-dessous l'arbre 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.

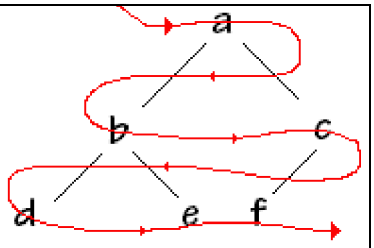


On voit par exemple, que par calcul on a bien le fils gauche du noeud d'indice 2 est dans la cellule d'index 2*2 = 4 et son fils droit se trouve dans la cellule d'index 2*2+1 = 5 ...

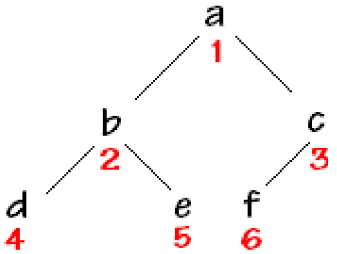
Exemple d'un arbre parfait étiqueté avec des caractères :



arbre parfait



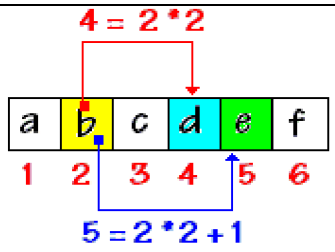
parcours hiérarchique



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**'.

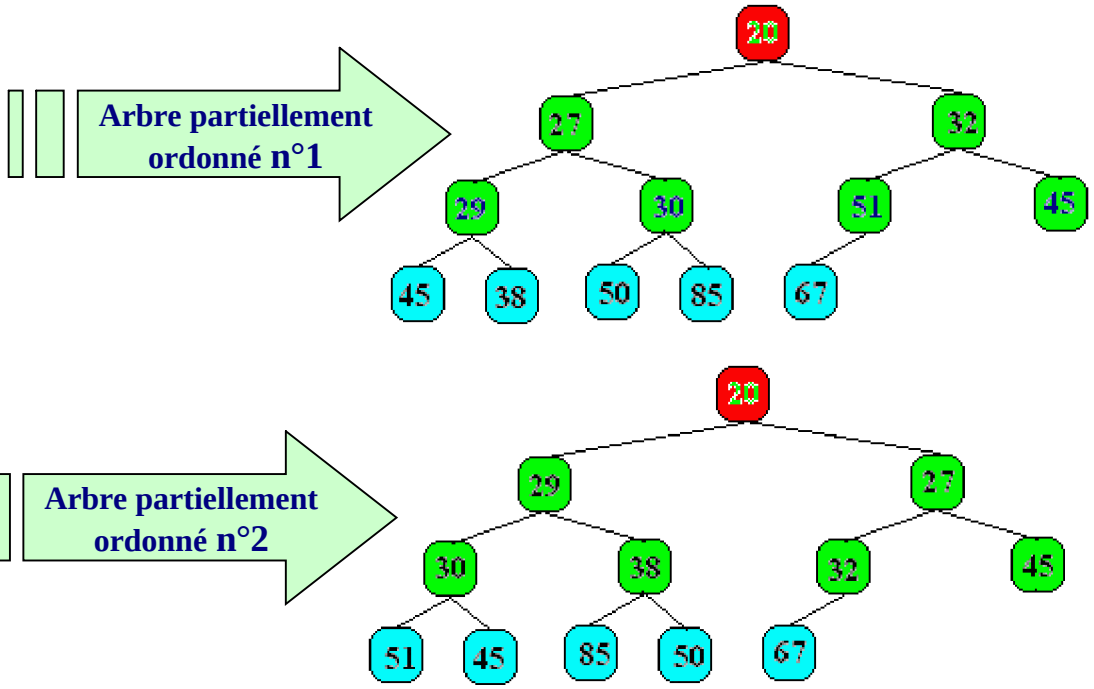


Arbre partiellement ordonné

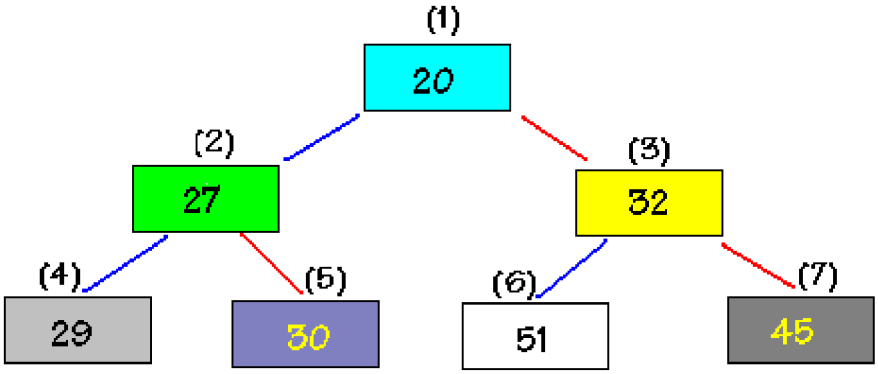
C'est un arbre étiqueté dont les valeurs des noeuds appartiennent à un ensemble muni d'une **relation d'ordre total** (les nombres entiers, réels etc... en sont des exemples) tel que pour un noeud donné tous ses **fils ont une valeur supérieure ou égale à celle de leur père**.

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

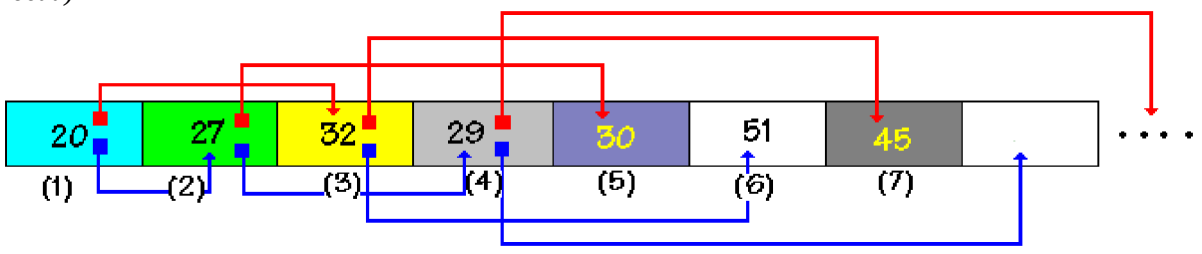
sur l'ensemble {20,27,29,30,32,38,45,45,50,51,67,85} d'entiers naturels :



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

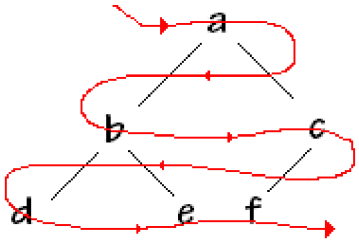
Objectif : les arbres sont des structures de données. Les informations sont contenues dans les noeuds de l'arbre, afin de construire des algorithmes effectuant des opérations sur ces informations (ajout, suppression, modification,...) il nous faut pouvoir examiner tous les noeuds d'un arbre. Examinons 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

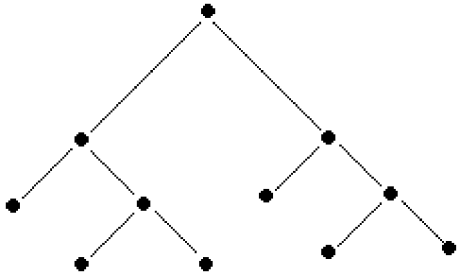
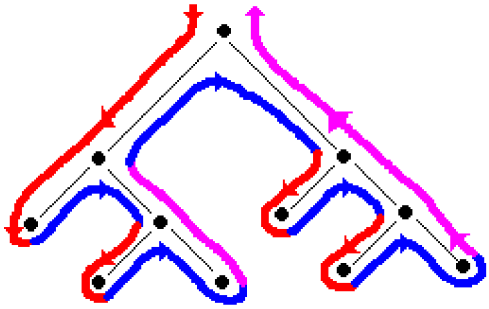
La stratégie consiste à **descendre** le plus profondément soit **jusqu'aux feuilles** d'un noeud de l'arbre, puis lorsque toutes les feuilles du noeud ont été visitées, l'algorithme "**remonte**" au noeud plus haut dont les feuilles n'ont pas encore été visitées.

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

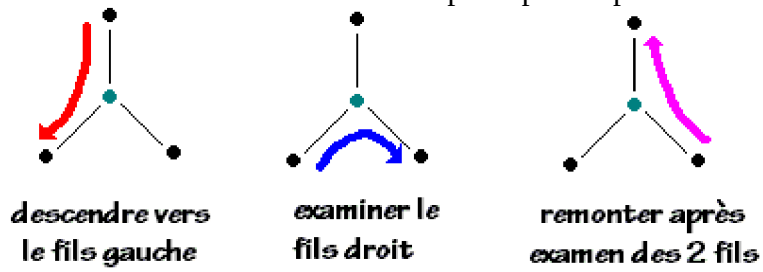
Parcours en profondeur par la gauche

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**".

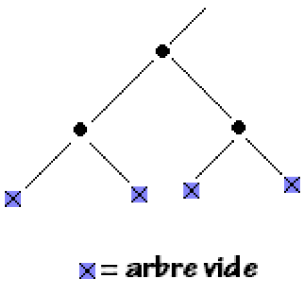
Schémas montrant le principe du parcours exhaustif en "**profondeur par la gauche**" :

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).

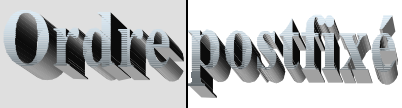
En fait on n'utilise en pratique que trois variantes de cet algorithme, celles qui constituent des parcours ordonnés de l'arbre en fonction de l'application du traitement de l'information située aux noeuds. Chacun de ces 3 parcours définissent un ordre implicite (préfixé, infixé, postfixé) sur l'affichage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

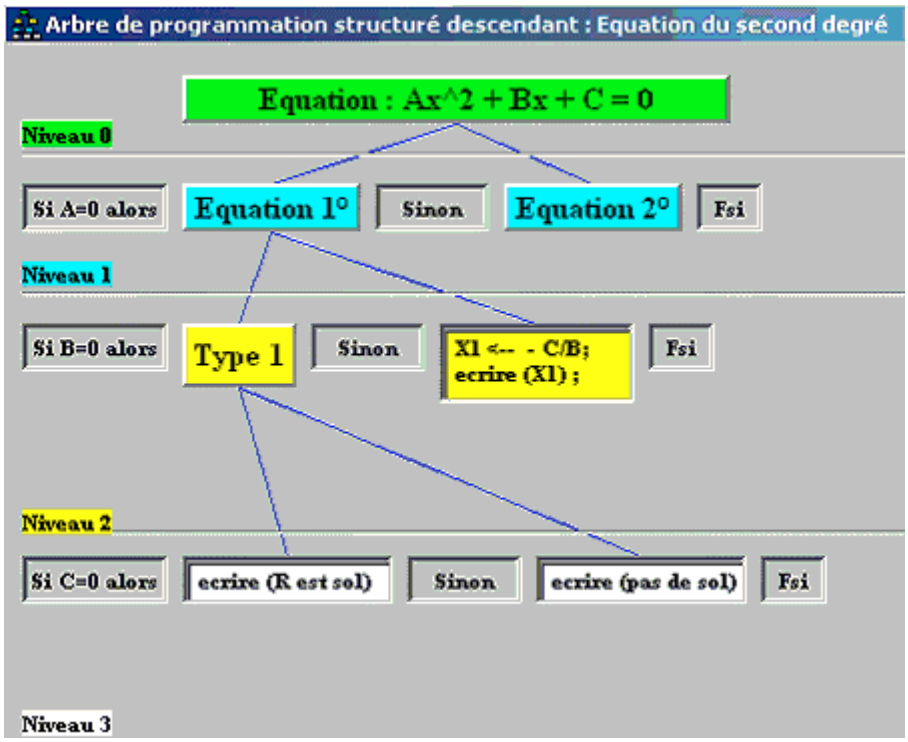
```
parcourir ( Arbre )
si Arbre  ≠ ∅  alors
   parcourir ( Arbre.filsG) ;
   Traiter-2 (info(Arbre.Racine)) ;
   parcourir ( Arbre.filsD ) ;
Fsi
```

Illustration prtaique d'un parcours général en profondeur

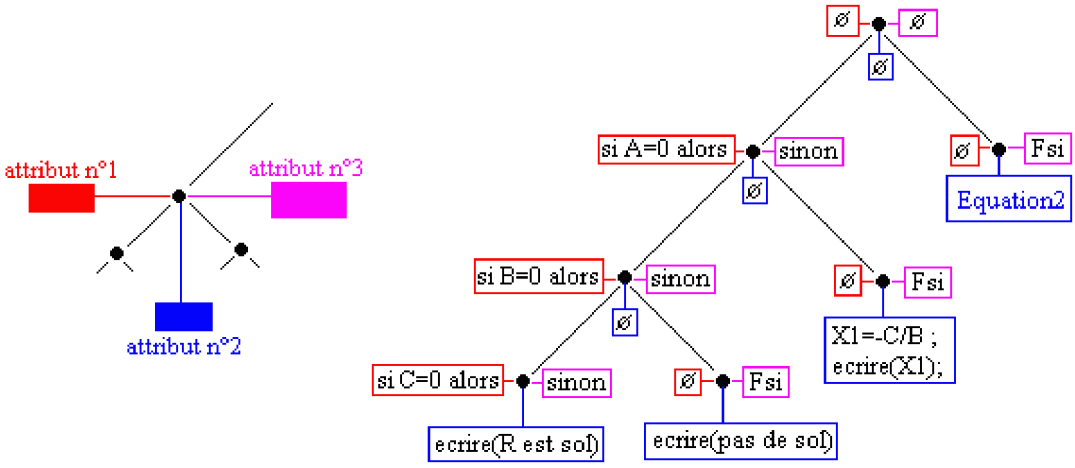
Le lecteur trouvera plus loin des exemples de parcours selon l'un des 3 ordres infixé, préfixé, postfixé, nous proposons ici un exemple didactique de parcours général avec les 3 traitements.

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) :



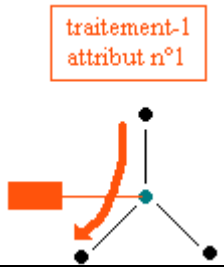
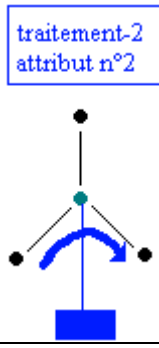
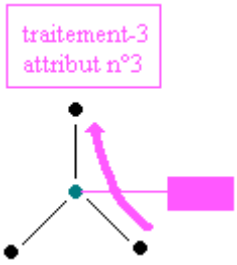
Ci-dessous une représentation schématique de la branche gauche de l'arbre de programmation précédent :



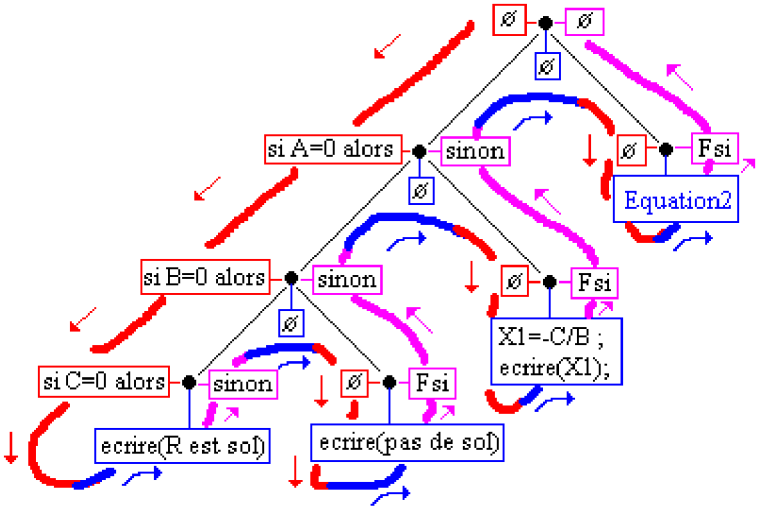
Nous avons établi un modèle d'arbre (binaire ici) où les informations au noeud sont au nombre de 3 (nous les nommerons attribut n°1, attribut n°2 et attribut n°3). Chaque attribut est une **chaîne de caractères**, vide s'il y a lieu.

Nous noterons ainsi un attribut contenant une chaîne vide : ∅

Traitement des attributs pour produire le texte

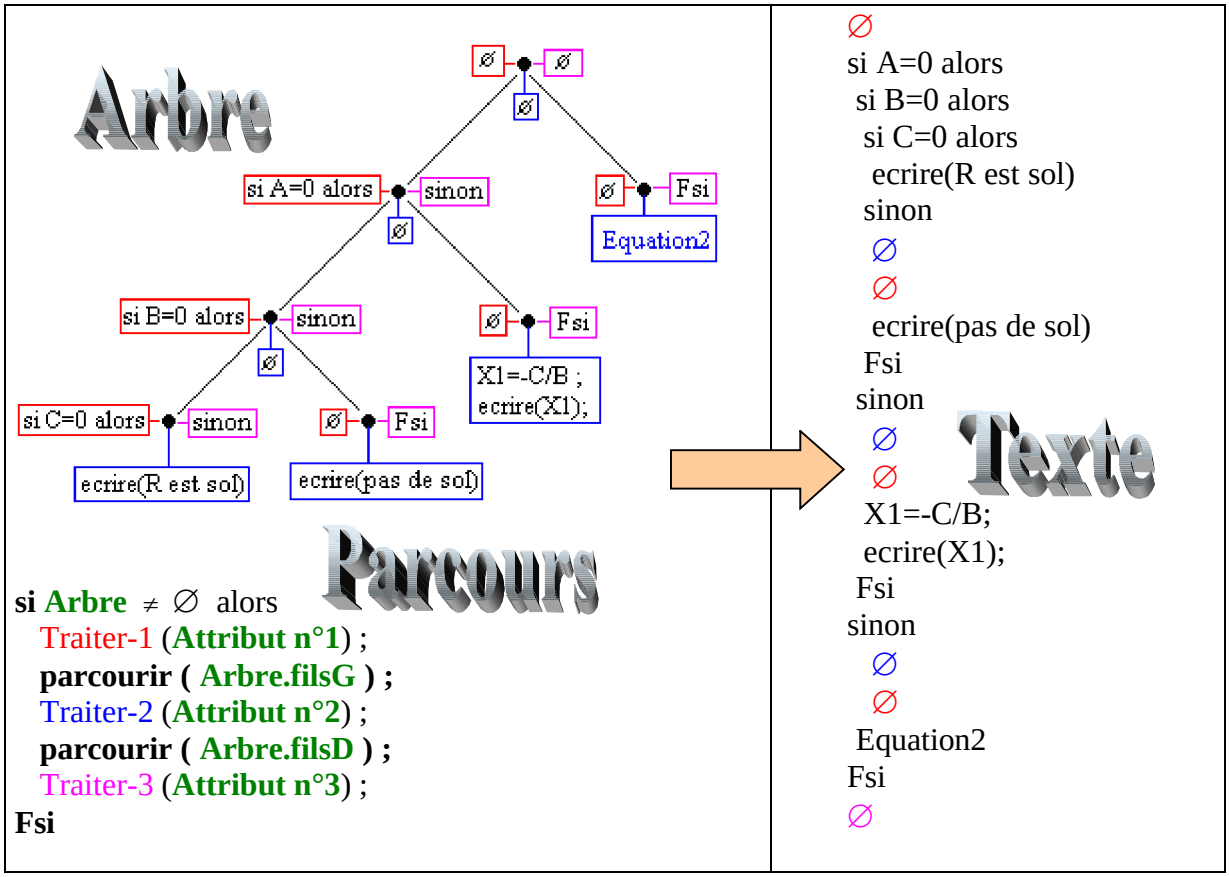
	Traiter-1 (**Arbre.Racine.Attribut n°1**) consiste à écrire le contenu de l'Attribut n°1 : **si** Attribut n°1 non vide **alors** ecrire(Attribut n°1) **Fsi**
	Traiter-2 (**Arbre.Racine.Attribut n°2**) consiste à écrire le contenu de l'Attribut n°2 : **si** Attribut n°2 non vide **alors** ecrire(Attribut 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 :



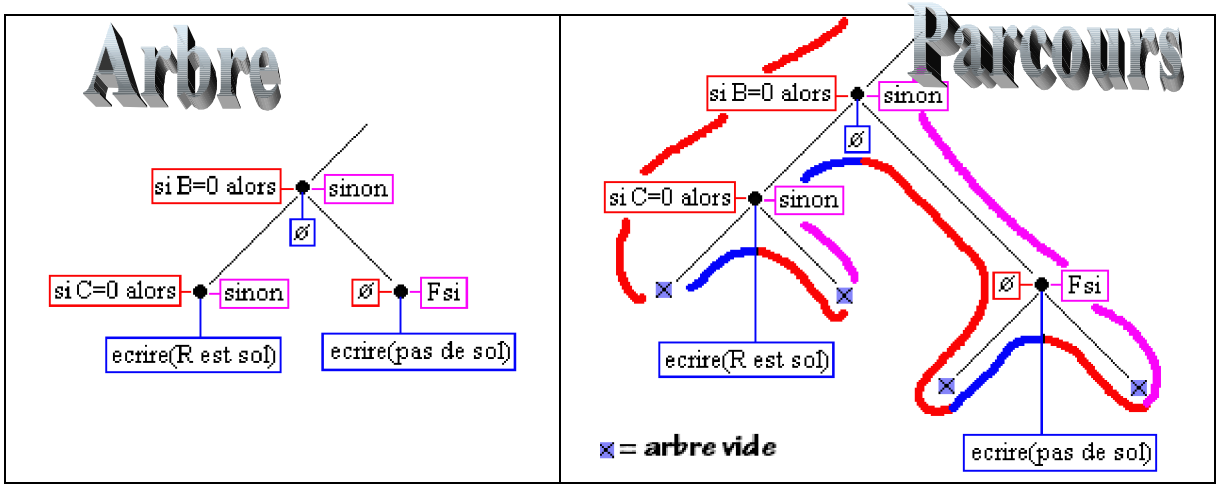
parcourir (Arbre)

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

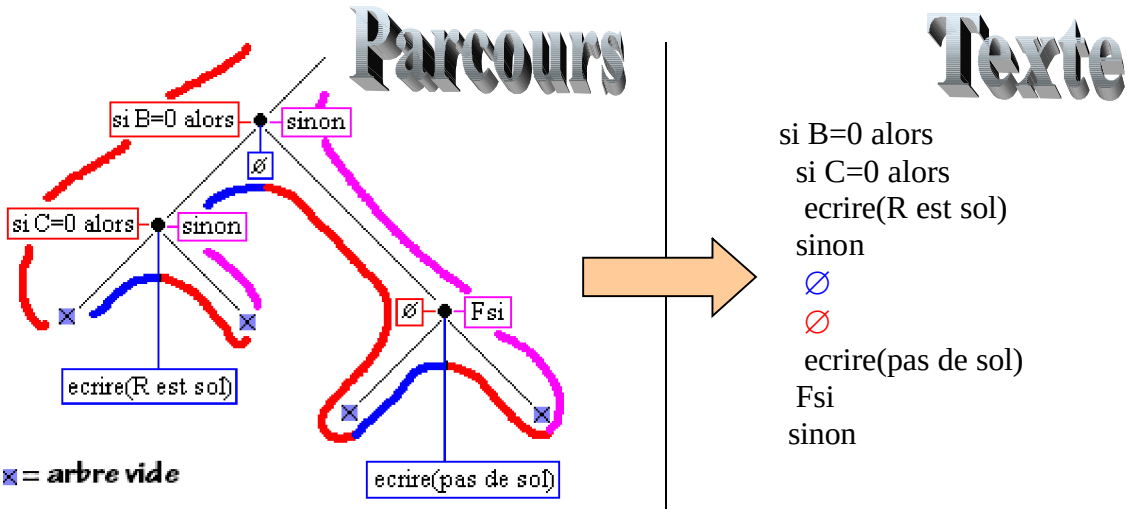


Rappelons que le symbole ∅ représente la chaîne vide il est uniquement mis dans le texe dans le but de permettre le suivi du parcours de l'arbre.

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** :

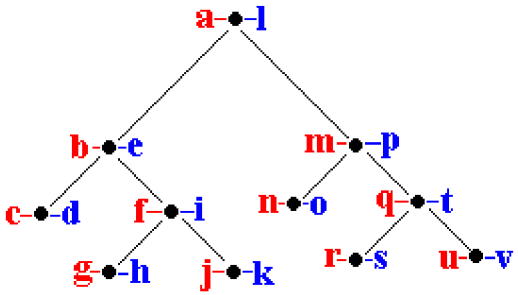


Le parcours partiel ci-haut produit le texte algorithmique suivant (le symbole ∅ est encore écrit pour la compréhension de la traversée) :



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
```

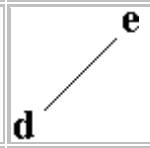
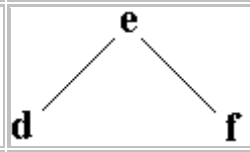
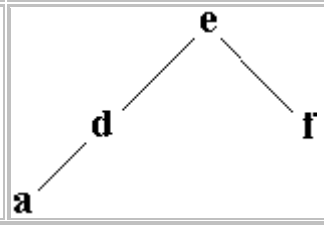
2.6 Insertion, suppression, recherche dans un arbre binaire de recherche

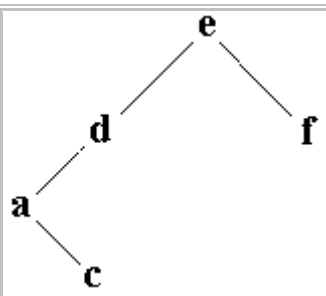
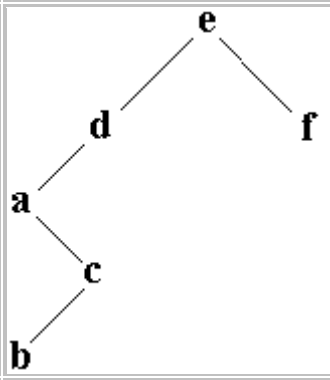
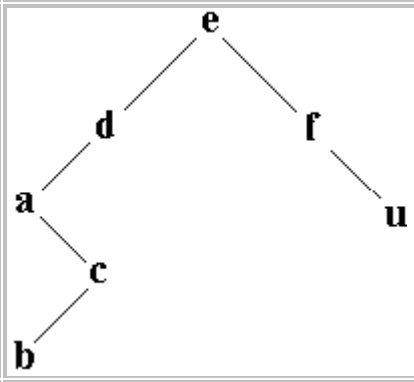
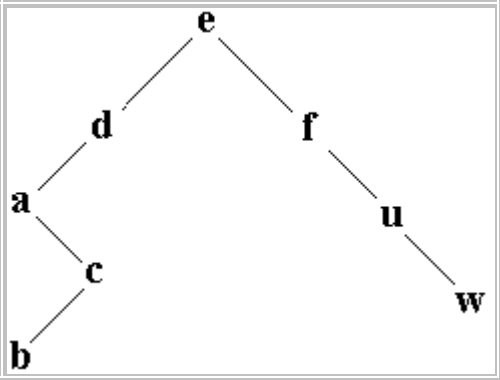
Algorithme d'insertion dans un arbre binaire de recherche

*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

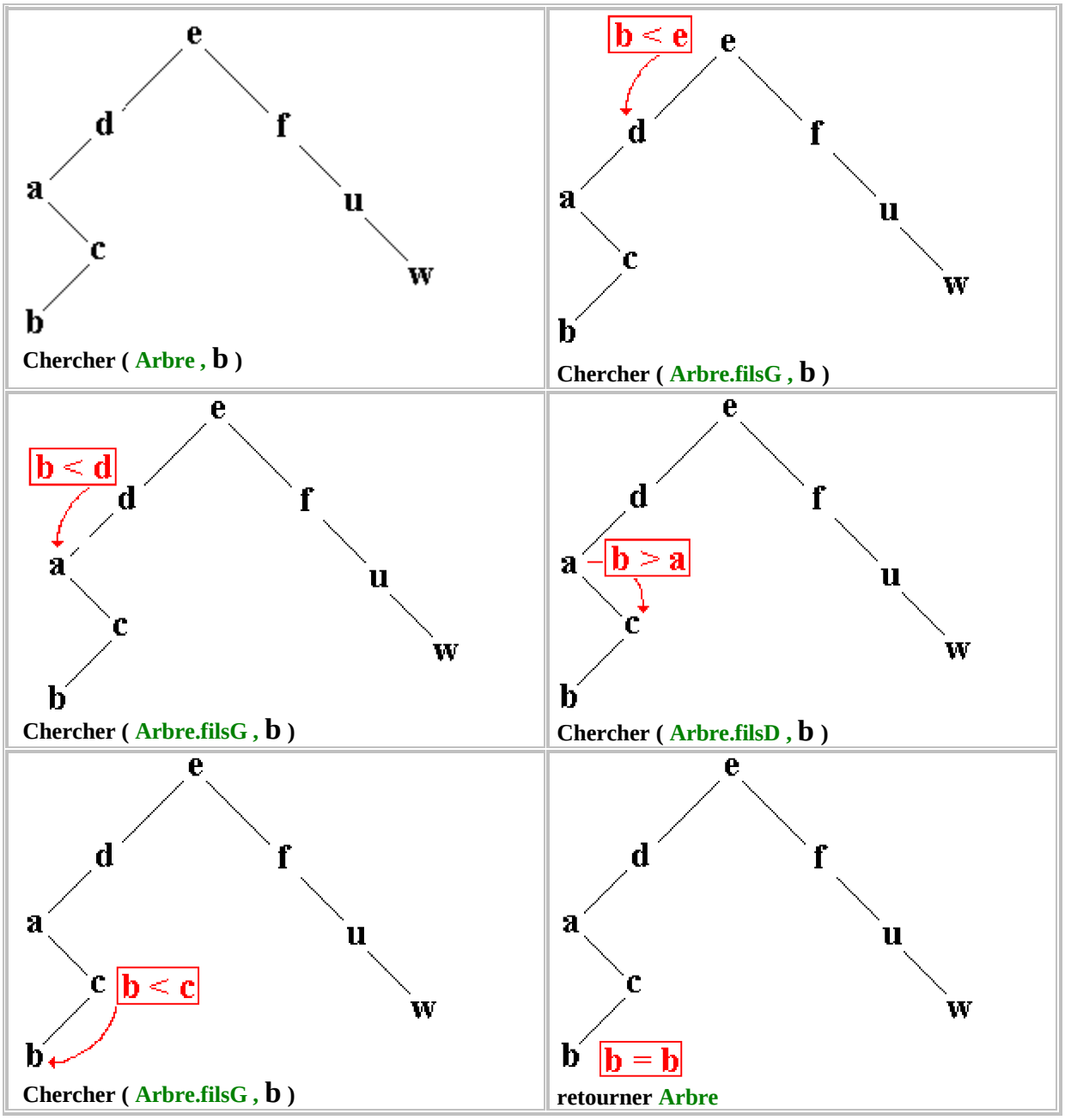
*chercher l'élément Elt dans l'arbre **Arbre :***

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
```

Exercices chapitre 4

Ex-1 : On défini un nombre rationnel (fraction) comme un couple de deux entiers : le numérateur et le dénominateur. On donne ci-dessous un TAD minimal de rationnel et l'on demande de l'implanter en **Unit** Delphi.

TAD : rationnel
Utilise : **Z** *//ensemble des entiers relatifs.*
Champs : (Num , Denom) ∈ **Z x Z***
Opérations :

Num :	———> rationnel	
Denom :	———> rationnel	
Reduire :	rationnel	———> rationnel
Addratio :	rationnel x rationnel	———> rationnel
Divratio :	rationnel x rationnel	———> rationnel
Mulratio :	rationnel x rationnel	———> rationnel
AffectQ :	rationnel	———> rationnel
OpposeQ :	rationnel	———> rationnel

Préconditions :
Divratio (x,y) **defssi** y.Num ≠ 0

Finrationnel

Ex-2 : On défini un nombre complexe à coefficient entiers relatifs comme un couple de deux entiers relatifs : la partie réelle et la partie imaginaire. On donne ci-dessous un TAD minimal de nombre et l'on demande de l'implanter en **Unit** Delphi.

TAD complexe
Utilise : **Z**
Champs : (part_reel , part_imag) ∈ **Z x Z***
Opérations :

part_reel :	———> **Z**	
part_imag :	———> **Z**	
Charger :	**Z x Z**	———> complexe
AffectC :	complexe	———> complexe
plus :	complexe x complexe	———> complexe
moins:	complexe x complexe	———> complexe
mult :	complexe x complexe	———> complexe
OpposeC :	complexe	———> complexe

Préconditions :
aucune

Fincomplexe

Ex-3 : On défini un nombre complexe à coefficient rationnel comme un couple de deux rationnels : la partie réelle et la partie imaginaire. On demande de construire un TAD minimal de nombre complexe utilisant le TAD rationnel et de l'implanter en **Unit** Delphi.

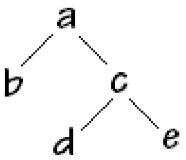
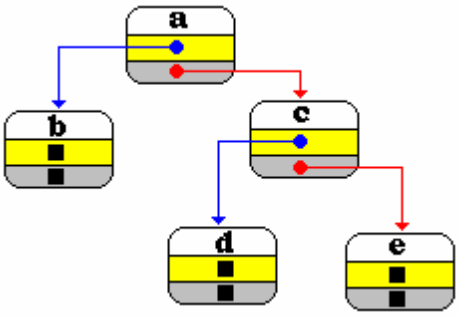
Ex-4 : En reprenant l'énoncé de l'exercice Ex-1 TAD rationnel, on implante sous forme de **classe** Delphi ce TAD et on compare son implantation en **Unit**.

Ex-5 : En reprenant l'énoncé de l'exercice Ex-3 TAD de nombre complexe à coefficients rationnels, on implante sous forme de **classe** Delphi ce TAD et on compare son implantation en **Unit**.

Ex-6 : Implantations des 4 algorithmes de parcours d'un arbre binaire étudiés dans le cours :

Il est demandé d'implanter en Delphi chacun des trois parcours en profondeur par la gauche définissant un ordre implicite (préfixé, infixé, postfixé) sur l'affichage et le traitement des données de type **string** contenues dans un arbre binaire, et le parcours en largeur avec une file Fifo.

Il est demandé d'écrire en Delphi console, l'implantation de la structure d'arbre binaire et des algorithmes de parcours avec des variables dynamiques de type **parbre = ^arbre** où arbre est un type **record** à définir selon le schéma fournit par l'exemple ci-dessous :

L'arbre :	Est implanté par les pointeurs **parbre** suivants :
	

Le traitement au nœud consistera à afficher la donnée de type string.

Puis uniquement lorsque vous aurez lu les chapitre sur les classes et la programmation objet, vous reprendrez l'implantation de la structure d'arbre binaire et des algorithmes de parcours avec une classe **TreeBin** selon le modèle ci-après :

```
TreeBin = class
    private
      FInfo : string ;
      procedure FreeRecur( x :TreeBin ) ;
      procedure PreOrdre ( x : TreeBin ; var res : string);
      procedure PostOrdre ( x : TreeBin ; var res : string);
    public
      filsG , filsD : TreeBin ;
      constructor Create(s:string; fg , fd : TreeBin);
      destructor Liberer;
      function Prefixe : string ;
      function Postfixe : string ;
      property Info:string read FInfo write FInfo;
 end;
```

Ex-7 : Il est demandé d'implanter en Delphi les 3 algorithmes d'insertion, de suppression et de recherche dans un arbre binaire de recherche ; comme dans l'exercice précédent vous proposerez une version avec variables dynamiques et une version avec une classe **TreeBinRech** héritant de la classe **TreeBin** définie à l'exercice précédent.

Ex-1 TAD rationnel - solution en Unit Delphi

Spécifications opérationnelles

TAD : rationnel
Utilise : **Z** *//ensemble des entiers relatifs.*
Champs : (Num , Denom) ∈ **Z x Z***
Opérations :

Num :	———> rationnel	
Denom :	———> rationnel	
Reduire :	rationnel	———> rationnel
Addratio :	rationnel x rationnel	———> rationnel
Divratio :	rationnel x rationnel	———> rationnel
Mulratio :	rationnel x rationnel	———> rationnel
AffectQ :	rationnel	———> rationnel
OpposeQ :	rationnel	———> rationnel

Préconditions :

Divratio (x,y) **defssi** y.Num ≠ 0

Finrationnel

Reduire : rendre le rationnel irréductible en calculant le pgcd du numérateur et du dénominateur, puis diviser les deux termes par ce pgcd.
Addratio : addition de deux nombres rationnels par la recherche du plus petit commun multiple des deux dénominateurs et mise de chacun des deux rationnels au même dénominateur.
Mulratio : multiplication de deux nombres rationnels, par le produit des deux dénominateurs et le produit des deux numérateurs.
Divratio : division de deux nombres rationnels, par le produit du premier par l'inverse du second.
AffectQ : affectation classique d'un rationnel dans un autre rationnel.
OpposeQ : renvoie l'opposé d'un rationnel dans un autre rationnel

Spécifications d'implantation

La structure de données choisie est le type record permettant de stocker les champs numérateur et dénominateur d'un nombre rationnel :

```
unit Uratio;  {unité de rationnels spécification classique ZxZ/R }
interface
    type      rationnel = record
                              num: integer;
                              denom: integer
                          end;
procedure reduire (var r: rationnel);
procedure addratio (a, b: rationnel; var s: rationnel);
procedure divratio (a, b: rationnel; var s: rationnel);
procedure mulratio (a, b: rationnel; var s: rationnel);
procedure affectQ(var s: rationnel; b: rationnel);
procedure opposeQ(x:rationnel;var s:rationnel);
```

Code de la Unit : Uratio

```
unit Uratio; {unité de rationnels spécification classique ZxZ/R}

interface

 type
  rationnel =
  record
   num: integer;
   denom: integer
  end;
  procedure reduire (var r: rationnel);
  procedure addratio (a, b: rationnel; var s: rationnel);
  procedure divratio (a, b: rationnel; var s: rationnel);
  procedure mulratio (a, b: rationnel; var s: rationnel);
  procedure affectQ(var s: rationnel; b: rationnel);
  procedure opposeQ(x:rationnel;var s:rationnel);

implementation

procedure maxswap (var a: integer; var b: integer);
var
   t: integer;
   begin
    if a < b then
    begin
      t := a;
      a := b;
      b := t
    end;
   end;
```

---------- SPECIFICATIONS ----------
maxswap : N x N ---> N x N
met le plus grand des deux entiers a et b dans a,
et le plus petit dans b.
local: t
paramètre-Entrée: a, b
paramètre-Sortie: a, b

```
function pgcd (a, b: integer): integer;
  var
   r: integer;
   begin
    maxswap(a, b);
    if a*b=0 then
      pgcd:=1
    else
    begin
      repeat
        r := a mod b;
        a := b;
        b := r
      until
       r = 0;
       pgcd := a
    end
   end;
```

---------- SPECIFICATIONS ----------
pgcd : N° x N° ---> N
renvoie le pgcd de deux entiers non nuls
pgcd=1, si l'un des 2 au moins est nul.
local: r
paramètre-Entrée: a, b
paramètre-Sortie: pgcd
utilise: maxswap

```
function ppcm (a, b: integer): integer;
  var
    k, p: integer;
    begin
      maxswap(a, b);
      if a*b=0 then
        ppcm:=0
      else
      begin
        k := 1;
        p := b;
        while (k <= a) and (p mod a <> 0) do
        begin
          p := b * k;
          k := k + 1
        end;
        ppcm := p
      end
    end;
```

---------- SPECIFICATIONS ----------
ppcm : N° x N° ---> N
renvoie le ppcm de deux entiers non nuls
renvoie 0, si l'un des 2 au moins est nul.
local: k, p
paramètre-Entrée: a, b
paramètre-Sortie: ppcm
utilise: maxswap

```
procedure reduire (var r: rationnel);
  var     pg: integer;
    begin
      if r.denom=0 then
        halt
      else
      begin
        pg := pgcd(r.num, r.denom);
        if pg <> 1 then
        begin
          r.num := r.num div pg;
          r.denom := r.denom div pg
        end;
        if (r.num>0) and (r.denom>0)
          or (r.num<0) and (r.denom<0) then
        begin {positif}
          r.num:=abs(r.num);
          r.denom:=abs(r.denom);
        end
        else
        begin{négatif}
          r.num:=-abs(r.num);
          r.denom:=abs(r.denom);
        end
      end
    end;
```

---------- SPECIFICATIONS ----------
reduire : Q ---> Q
rend un rationnel non nul irréductible
arrête l'exécution, si le dénominateur est nul.
Le signe est détenu par le numérateur.
local: pg
paramètre-Entrée: r
paramètre-Sortie: r
utilise: pgcd

```
procedure affectQ( var s: rationnel; b: rationnel);
  begin
    s.num:=b.num;
    s.denom:=b.denom
  end;
```

```
procedure opposeQ(x:rationnel;var s:rationnel);
 begin
  s.num:=-x.num;
  s.denom:=x.denom
 end;
```

```
procedure addratio (a, b: rationnel; var s: rationnel);
 var
  divcom, coeff_a, coeff_b: integer;
  begin
   reduire(a);
   reduire(b);
   divcom := ppcm(a.denom, b.denom);
   coeff_a := divcom div a.denom;
   coeff_b := divcom div b.denom;
   s.num := a.num * coeff_a + b.num * coeff_b;
   s.denom := divcom;
   reduire(s);
  end;
```

---------- SPECIFICATIONS ----------
addratio : Q x Q ---> Q
donne dans s la somme des deux rationnels non nuls, s=a+b
a et b sont rendus irréductibles avant le calcul.
Le résultat s est rendu irréductible après calcul.
local: divcom, coeff_a, coeff_b
paramètre-Entrée: a, b
paramètre-Sortie: s
utilise: reduire,ppcm

```
procedure divratio (a, b: rationnel; var s: rationnel);
 begin
  reduire(a);
  reduire(b);
  if b.num=0 then
   halt
  else
  begin
   s.num := a.num * b.denom;
   s.denom := a.denom * b.num;
   reduire(s)
  end
 end;
```

---------- SPECIFICATIONS ----------
divratio : Q x Q ---> Q
donne dans s le rapport des deux rationnels non nuls, s=a/b
a et b sont rendus irréductibles avant le calcul.
Le résultat s est rendu irréductible après calcul.
paramètre-Entrée: a, b
paramètre-Sortie: s
utilise: reduire

```
procedure mulratio (a, b: rationnel; var s: rationnel);
 begin
  reduire(a);
  reduire(b);
  s.num := a.num * b.num;
  s.denom := a.denom * b.denom;
  reduire(s)
 end;

end.
```

---------- SPECIFICATIONS ----------
multratio : Q x Q ---> Q
donne dans s le produit des deux rationnels non nuls, s=a*b
a et b sont rendus irréductibles avant le calcul.
Le résultat s est rendu irréductible après calcul.
paramètre-Entrée: a, b
paramètre-Sortie: s
utilise: reduire

Utilisation de la unit

```
program essaiRatio;  {début programme de test de la unit Uratio de nombres rationnels  }
 uses Uratio;
 var    r1, r2, r3, r4, r5 : rationnel;
  begin    { exemple de calcul sur les rationnels non nuls à continuer}
   r1.num :=18;     r1.denom := 15;     r2.num := 7;     r2.denom := 12;
   addratio(r1, r2, r3);
   writeln('18/15 + 7/12 = ', r3.num, '/', r3.denom);
   mulratio(r1, r2, r4);
   writeln('18/15 * 7/12 = ', r4.num, '/', r4.denom);  …..
```

Ex-2 TAD nombre complexe à coeff entiers - solution en Unit Delphi

Spécifications opérationnelles

TAD complexe
Utilise : **Z**
Champs : (part_reel , part_imag) ∈ **Z x Z***
Opérations :

part_reel :	———> **Z**
part_imag :	———> **Z**
Charger :	**Z x Z** ———> complexe
AffectC :	complexe ———> complexe
plus :	complexe x complexe ———> complexe
moins:	complexe x complexe ———> complexe
mult :	complexe x complexe ———> complexe
OpposeC :	complexe ———> complexe

Préconditions :
aucune
Fincomplexe

Charger : remplit les deux champs part_reel et part_imag d'un nombre complexe.
AffectC : affectation classique d'un complexe dans un autre.
plus : addition de 2 nombres complexes spécif. mathématique classique :
z_1=x+iy et z_2=x'+iy' =>z_1+z_2=(x+x')+(y+y')i.
moins : soustraction de 2 nombres complexes spécif. mathématique classique :
z_1=x+iy et z_2=x'+iy' =>z_1-z_2=(x-x')+(y-y')i.
mult : multiplication de 2 nombres complexes spécif mathématique classique :
z_1=x+iy et z_2=x'+iy' =>z_1+z_2=(x.x'- y.y')+(x.y'+x'.y)i.
OpposeC : pour un nombre complexe x+iy cet opérateur renvoie –x –iy

Spécifications d'implantation

La structure de données choisie est aussi ici le type **record** permettant de stocker les champs partie réelle et partie imaginaire d'un nombre complexe :

```
unit  UComplx1; {unit de calcul sur les nombres complexes à coefficients entiers}
interface
  type    complex = record
                      part_reel: integer;
                      part_imag: integer
                    end;

  procedure Charger (x, y: integer; var z: complex);
  procedure affectC (var z: complex;y: complex );
  procedure plus (x, y: complex; var z: complex);
  procedure moins (x, y: complex; var z: complex);
  procedure mult (x, y: complex; var z: complex);
  procedure OpposeC (x:complex; var z:complex);
```

Code de la Unit : UComplx1

```
unit UComplx1;{unit de calcul sur les nombres complexes à coefficients entiers }

interface

 type
  complex =    record
    part_reel: integer;
    part_imag: integer
  end;

  procedure Charger (x, y: integer; var z: complex);
  procedure affectC (var z: complex;y: complex );
  procedure plus (x, y: complex; var z: complex);
  procedure moins (x, y: complex; var z: complex);
  procedure mult (x, y: complex; var z: complex);
  procedure OpposeC(x:complex;var z:complex);

implementation

procedure Charger (x, y: integer; var z: complex);
 begin
   z.part_reel := x;
   z.part_imag := y
 end;

 procedure affectC (var z: complex;y: complex );
 begin
   z.part_reel := y.part_reel;
   z.part_imag := y.part_imag
 end;

 procedure plus (x, y: complex; var z: complex);
 begin
   z.part_reel := x.part_reel + y.part_reel;
   z.part_imag := x.part_imag + y.part_imag
 end;

procedure moins (x, y: complex; var z: complex);
begin
  z.part_reel := x.part_reel - y.part_reel;
   z.part_imag := x.part_imag - y.part_imag
 end;

 procedure mult (x, y: complex; var z: complex);
 begin
   z.part_reel := x.part_reel * y.part_reel
                    - x.part_imag * y.part_imag;
   z.part_imag := x.part_reel * y.part_imag
                   + y.part_reel * x.part_imag
 end;

 procedure OpposeC(x:complex;var z:complex);
 begin
   z.part_reel := -x.part_reel;
   z.part_imag := x.part_imag
 end;

end.
```

Utilisation de la unit

```
program essai_complexe1;
 {test de l'utilisation de la unit Ucomplx1
  de nombres complexes à coeff. Entiers   }
 uses
   Ucomplx1;

 var
   u, z, z1, z2: complex;

procedure ecrit (v: string; z: complex);
begin
   writeln(v, ': ', z.part_reel : 3,
                '+ ', z.part_imag : 3, '.i')
end;

begin
    writeln('_________');
    Charger(2, -3, z1);
    ecrit('z1', z1);
    Charger(8, 113, z2);
    ecrit('z2', z2);
    affectC(z, z1);
    ecrit('z', z);
    plus(z1, z2, u);
    ecrit('u=z1+z2', u);
    moins(z1, z2, u);
    ecrit('u=z1-z2', u);
    Charger(1, 0, z);
    mult(z1, z2, u);
    ecrit('u=z1*z2', u);
    mult(z, z2, u);
    ecrit('u=1*z2', u);
end.
```

Le TAD complexe utilisant Le TAD rationnel

TAD complexe
Utilise : **rationnel**
Champs : (part_reel , part_imag) ∈ **rationnel x rationnel**
Opérations :

part_reel :	———> **rationnel**	
part_imag :	———> **rationnel**	
Charger :	**rationnel x rationnel**	———> complexe
AffectC :	complexe	———> complexe
plus :	complexe x complexe	———> complexe
moins:	complexe x complexe	———> complexe
mult :	complexe x complexe	———> complexe
OpposeC :	complexe	———> complexe

Préconditions :
aucune
Fincomplexe

Spécifications opérationnelles
(identiques à l'exercice précédent)

Charger : remplit les deux champs part_reel et part_imag d'un nombre complexe.
AffectC : affectation classique d'un complexe dans un autre.
plus : addition de 2 nombres complexes spécif. mathématique classique :
z_1=x+iy et z_2=x'+iy' =>z_1+z_2=(x+x')+(y+y')i.
moins : soustraction de 2 nombres complexes spécif. mathématique classique :
z_1=x+iy et z_2=x'+iy' =>z_1-z_2=(x-x')+(y-y')i.
mult : multiplication de 2 nombres complexes spécif mathématique classique :
z_1=x+iy et z_2=x'+iy' =>z_1+z_2=(x.x'- y.y')+(x.y'+x'.y)i.
OpposeC : pour un nombre complexe x+iy cet opérateur renvoie –x –iy

Spécifications d'implantation

```
unit UComplx2; {unit de nombres complexes à
  coefficients rationnels utilisant
  la unit de rationnels déjà construite Uratio }

interface
uses Uratio;
  type
    complex = record
       part_reel: rationnel;
       part_imag: rationnel
     end;
```

```
procedure Charger (x, y: integer; var z: complex);
procedure affectC (var z: complex;y: complex );
procedure plus (x, y: complex; var z: complex);
procedure moins (x, y: complex; var z: complex);
procedure mult (x, y: complex; var z: complex);
procedure OpposeC(x:complex; var z:complex);
```

Code de la Unit : Ucomplx2

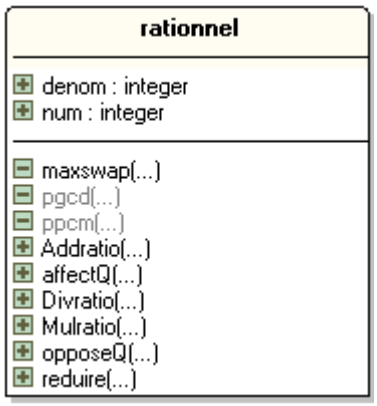
```
unit UComplx2;
{unit de nombres complexes à coefficients rationnels utilisant la unit de rationnels déjà construite Uratio }
interface
 uses Uratio;
 type
  complex =    record
    part_reel: rationnel;
    part_imag: rationnel
  end;

  procedure Charger (x, y: rationnel; var z: complex);
  procedure affectC (var z: complex;y: complex );
  procedure plus (x, y: complex; var z: complex);
  procedure moins (x, y: complex; var z: complex);
  procedure mult (x, y: complex; var z: complex);
  procedure OpposeC(x:complex;var z:complex);

implementation

procedure OpposeC ( x:complex; var z:complex); (* fournit l'opposé de x dans z *)
  begin
    OpposeQ(x.part_reel,z.part_reel);
    OpposeQ(x.part_imag,z.part_imag);
  end;

 procedure Charger (x, y : rationnel; var z : complex);
 begin
  reduire(x);
  reduire(y);
  affectQ(z.part_reel,x);
  affectQ(z.part_imag,y);
 end;

 procedure affectC (var z: complex;y: complex );
 begin
  Charger(y.part_reel,y.part_imag,z)
 end;

 procedure plus (x, y: complex; var z: complex);
 var r1,r2:rationnel;
 begin
  addratio(x.part_reel,y.part_reel,r1);
  addratio(x.part_imag, y.part_imag,r2);
  Charger (r1,r2,z)
 end;

 procedure moins (x, y: complex; var z: complex);
 var z1:complex;
 begin
  OpposeC(y,z1);
  plus(x,z1,z)
 end;

 procedure mult (x, y: complex; var z: complex);
 var r1,r2,r3,r4:rationnel;
 begin
  {z.part_reel := x.part_reel * y.part_reel - x.part_imag * y.part_imag;}
```

```
  mulratio(x.part_reel,y.part_reel,r1);
  mulratio(x.part_imag, y.part_imag,r2);
  OpposeQ(r2,r3);
  addratio(r1,r3,r4);

  {z.part_imag := x.part_reel * y.part_imag + y.part_reel * x.part_imag}
  mulratio(x.part_reel,y.part_imag,r1);
  mulratio(x.part_imag, y.part_reel,r2);
  addratio(r1,r2,r3);
  Charger(r4,r3,z)
 end;

end.
```

Utilisation de la unit Ucomplx2

```
program essai_complexe2;
 {programme de test et d'utilisation de la unit Ucomplx2  de nombres complexes à coeff. rationnels  }
 uses
  Uratio,Ucomplx2 ;
 procedure ecrit (v: string; z: complex);
 begin
  writeln(v, ': (', z.part_reel.num,'/', z.part_reel.denom, ')+ (', z.part_imag.num,'/', z.part_imag.denom, ').i')
 end;
 var   r1,r2,r3:rationnel;
       u, z, z1, z2: complex;
  begin
    writeln('_____');
    {//// les chargements dépendent du type des données ////}
    r1.num :=18;
    r1.denom := 15;
    r2.num := 7;
    r2.denom := 12;
    Charger(r1, r2, z1);
    ecrit('z1', z1);
    r1.num :=35;
    r1.denom := 25;
    r2.num := 11;
    r2.denom := 6;
    Charger(r1, r2, z2);
    ecrit('z2', z2);
    {//// les appels d'opérateurs sont identiques à ceux de l'exercice précédent: ////}
    affectC(z, z1);
    ecrit('z=z1', z);
    plus(z1, z2, u);
    ecrit('u=z1+z2', u);
    moins(z1, z2, u);
    ecrit('u=z1-z2', u);
    r1.num :=1;
    r1.denom := 1;
    r2.num := 0;
    r2.denom := 1;
    Charger(r1, r2, z1);
    mult(z1, z2, u);
    ecrit('u=1*z2', u);
    mult(z, z2, u);
    ecrit('u=z1*z2', u);
  end.
```

Ex-4 TAD nombre rationnel - solution en classe Delphi

Le TAD	La classe
TAD : rationnel **Utilise** : **Z** *//ensemble des entiers relatifs.* **Champs** : (Num , Denom) ∈ **Z x Z*** **Opérations** : Num : ———> rationnel Denom : ———> rationnel Reduire : rationnel ———> rationnel Addratio : rationnel × rationnel ———> rationnel Divratio : rationnel × rationnel ———> rationnel Mulratio : rationnel × rationnel ———> rationnel AffectQ : rationnel ———> rationnel OpposeQ : rationnel ———> rationnel **Préconditions** : Divratio (x,y) **defssi** y.Num ≠ **Finrationnel**	

En Delphi

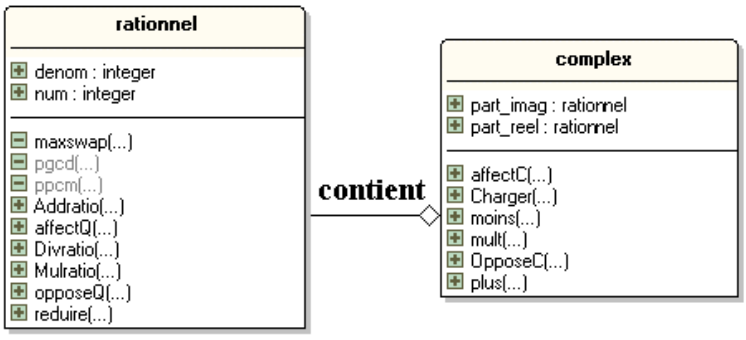
Classe

```
unit UClasseratio;
 interface
 type
  rationnel = class
   private
      function pgcd (a, b: integer): integer;
      function ppcm (a, b: integer): integer;
      procedure maxswap (var a: integer;
                         var b: integer);
   public
      num: integer;
      denom: integer;
      procedure reduire ;
      procedure Addratio (a,s: rationnel);
      procedure Divratio (a,d: rationnel);
      procedure Mulratio (a,m: rationnel);
      procedure affectQ (s: rationnel);
      procedure opposeQ (s:rationnel);
  end;

implementation
 procedure rationnel.maxswap (var a: integer;
                              var b: integer); …
 function rationnel.pgcd (a, b: integer): integer; …
 function rationnel.ppcm (a, b: integer): integer; …
 procedure rationnel.reduire ; …
 procedure rationnel.Addratio (a,s: rationnel); …
 procedure rationnel.Divratio (a,d: rationnel); …
 procedure rationnel.Mulratio (a,m: rationnel); …
 procedure rationnel.affectQ (s: rationnel); …
 procedure rationnel.opposeQ (s: rationnel); …
```

Unit

```
unit Unitratio;
interface
 type
  rationnel =  record
    num: integer;
    denom: integer
  end;

  procedure reduire (var r: rationnel);
  procedure Addratio (a, b: rationnel; var s: rationnel);
  procedure Divratio (a, b: rationnel; var s: rationnel);
  procedure Mulratio (a, b: rationnel; var s: rationnel);
  procedure affectQ (var s: rationnel; b: rationnel);
  procedure opposeQ (x:rationnel;var s:rationnel);

implementation

 procedure maxswap (var a: integer; var b: integer); …
 function pgcd (a, b: integer): integer; …
 function ppcm (a, b: integer): integer; …
 procedure reduire (var r: rationnel); …
 procedure Addratio (a, b: rationnel; var s: rationnel); …
 procedure Divratio (a, b: rationnel; var s: rationnel); …
 procedure Mulratio (a, b: rationnel; var s: rationnel); …
 procedure affectQ (var s: rationnel; b: rationnel); …
 procedure opposeQ (x: rationnel;var s: rationnel); …
```

Ex-5 TAD nombre complexe - solution en classe Delphi

Le TAD	La classe complex utilise la classe rationnel
TAD complex **Utilise** : **Z** **Champs** : (part_reel , part_imag) ∈ **Z x Z*** **Opérations** : part_reel : → **Z** part_imag : → **Z** Charger : **Z** × **Z** → complex AffectC : complex → complex plus : complex × complex → complex moins: complex × complex → complex mult : complex × complex → complex OpposeC : complex → complex **Fincomplexe**	

En Delphi

unit UClasseComplexe2; Classe

```
 interface
 uses UClasseratio;
 type
   complex = class
 private
 //pas d'éléments privés pour l'instant
 public
   part_reel: rationnel;
   part_imag: rationnel;
   procedure Charger (x, y: rationnel);
   procedure affectC (z: complex);
   procedure plus (x,z: complex);
   procedure moins (x,z: complex);
   procedure mult (x,z: complex);
   procedure OpposeC(z: complex);
end;

implementation
 procedure complex.Charger (x, y: rationnel); …
 procedure complex.affectC (z: complex); …
 procedure complex.plus (x,z:complex); …
 procedure complex.moins (x,z: complex); …
 procedure complex.mult (x,z: complex); …
 procedure complex.OpposeC(z:complex); …
end.
```

unit Ucomplx2; Unit

```
interface
 uses Uratio;
 type
   complex =
   record
     part_reel: rationnel;
     part_imag: rationnel
   end;

   procedure Charger (x, y: rationnel; var z: complex);
   procedure affectC (var z: complex;y: complex );
   procedure plus (x, y: complex; var z: complex);
   procedure moins (x, y: complex; var z: complex);
   procedure mult (x, y: complex; var z: complex);
   procedure OpposeC(x:complex;var z:complex);


implementation
 procedure Charger (x, y: rationnel; var z: complex); …
 procedure affectC (var z: complex;y: complex ); …
 procedure plus (x, y: complex; var z: complex); …
 procedure moins (x, y: complex; var z: complex); …
 procedure mult (x, y: complex; var z: complex); …
 procedure OpposeC(x:complex;var z:complex); …
end.
```

Ex-6 : Solution des 4 algorithmes de parcours d'un arbre

*Nous implantons en Delphi les 4 algorithmes dont 3 sont sous forme de procédures récursives. Nous supposons que les informations stockées dans un noeud sont du type chaîne de caractère (**string**), le traitement consistera ici à écrire le contenu de la string d'un noeud lorsqu'il est parcouru. La structure de données d'arbre est représentée par*

Implantations des données avec variables dynamique et classe

Nous proposons parallèlement les deux implantations demandées que le lecteur testera sur sa machine en fonction de son avancement dans le cours.

Implantation en Delphi avec des variables dynamiques :

```
type
   parbre = ^arbre;
   arbre = record
              info : string ;
              filsG, filsD: parbre
           end;
```

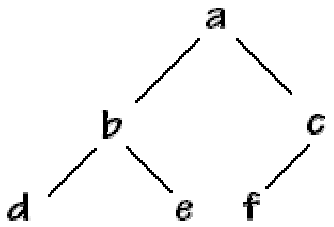
Implantation en Delphi avec une classe :

```
interface
// dans cette classe tous les champ sont publics afin de simplifier l'écriture
type
  TreeBin = class
    private
      procedure FreeRecur( x :TreeBin ) ;
    public
      Info : string;
      filsG , filsD : TreeBin;
      constructor CreerTreeBin(s:string);overload;
      constructor CreerTreeBin(s:string; fg , fd : TreeBin);overload;
      destructor Liberer;
  end;

implementation
{----------   Méthodes privé ----------}
procedure TreeBin.FreeRecur ( x : TreeBin ) ;
 // parcours postfixe pour détruire les objets de l'arbre x
begin
  FreeRecur( x.filsG );
  FreeRecur( x.filsD );
  x.Free
end;

{----------   Méthodes public ----------}
constructor TreeBin.CreerTreeBin(s:string);
 begin
    self.info := s;
    self.filsG := nil;
    self.filsD := nil;
 end;

 constructor TreeBin.CreerTreeBin(s : string ; fg, fd : TreeBin);
 begin
    self.info := s;
    self.filsG := fg;
```

```
    self.filsD := fd;
 end;

destructor TreeBin.Liberer;
 // la destruction de tout l'arbre :
begin
   FreeRecur (self) ;
   self := nil;
end;

end.
```

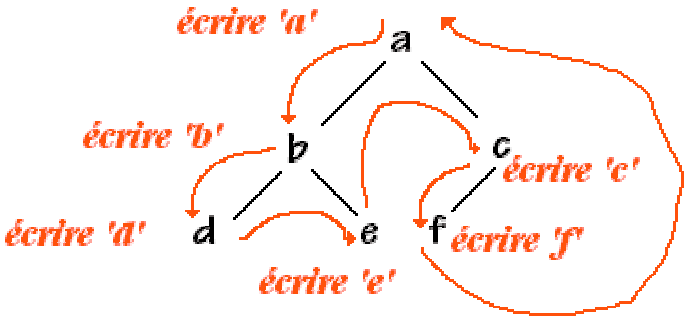
Nous prenons comme exemple sur lequel appliquer les 4 algorithmes, l'arbre binaire X suivant (chaque noeud a une info de type caractère stocké dans une string) :

X =



Implantation de l'algorithme de parcours en pré-ordre :

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



La procédure écrira successivement : **abdecf**

Préordre en Delphi avec les variables dynamiques :

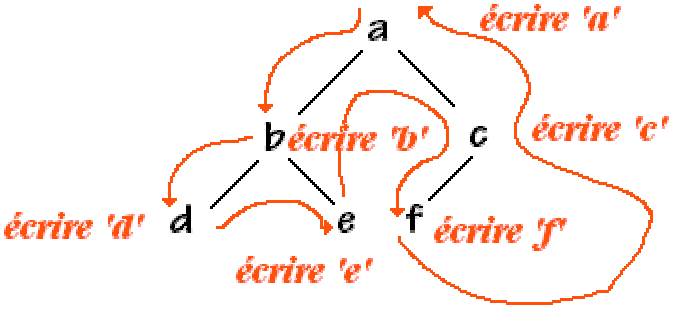
```
procedure prefixe (f : parbre);
 begin
  if f <> nil then
   with f^ do
    begin
     write(info);
     prefixe( filsG );
     prefixe( filsD )
    end
 end;
```

Préordre en Delphi avec la classe :

```
interface
type
  TreeBin = class
    private
      procedure FreeRecur( x :TreeBin ) ;
      procedure PreOrdre ( x : TreeBin);
    public
      Info : string;
      filsG , filsD : TreeBin;
      constructor CreerTreeBin(s:string);overload;
      constructor CreerTreeBin(s:string; fg , fd : TreeBin);overload;
      destructor Liberer;
      procedure Prefixe;
   end;
implementation
{------------------   Méthodes privé ----------------------}
procedure TreeBin.PreOrdre ( x : TreeBin ) ;
 // parcours préfixé d'un arbre x
begin
 if x<>nil then
 begin
   write(x.info);
   PreOrdre( x.filsG );
   PreOrdre( x.filsD )
 end
end;
//autres méthodes .....

{------------------   Méthodes public ----------------------}
procedure Prefixe;
// parcours préfixé de l'objet
begin
  PreOrdre( self );
end;
//autres méthodes .....

end.
```

Implantation de l'algorithme en post-ordre :

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



La procédure écrira successivement: **debfca**

Postordre en Delphi avec les variables dynamiques :

```
procedure postfixe (f : parbre);
 begin
  if f <> nil then
   with f^ do
    begin
     postfixe( filsG );
     postfixe( filsD );
     write(info)
    end
 end;
```

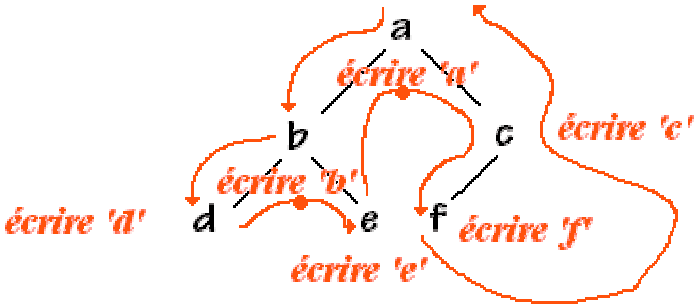
Postordre en Delphi avec la classe :

```
interface
type
  TreeBin = class
    private
      procedure FreeRecur( x :TreeBin ) ;
      procedure PreOrdre ( x : TreeBin);
      procedure PostOrdre ( x : TreeBin);
    public
      Info : string;
      filsG , filsD : TreeBin;
      constructor CreerTreeBin(s:string);overload;
      constructor CreerTreeBin(s:string; fg , fd : TreeBin);overload;
      destructor Liberer;
      procedure Prefixe;
      procedure Postfixe;
   end;
implementation
{------------------   Méthodes privé ------------------------}
procedure TreeBin.PostOrdre ( x : TreeBin ) ;
 // parcours postfixé d'un arbre x
begin
 if x<>nil then
 begin   PostOrdre( x.filsG );
   PostOrdre( x.filsD );
   write(x.info);
 end
end;
//autres méthodes .....

{------------------   Méthodes public ------------------------}
procedure Postfixe;
// parcours préfixé de l'objet
begin
  PostOrdre( self );
end;
//autres méthodes .....

end.
```

Implantation de l'algorithme en ordre symétrique :

```
parcourir ( Arbre )
```

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



La procédure écrira successivement : **dbeafc**

Ordre infixé en Delphi avec les variables dynamiques :

```
procedure infixe (f : parbre);
 begin
  if f <> nil then
   with f^ do
    begin
     infixe( filsG );
     write(info);
     infixe( filsD )
    end
 end;
```

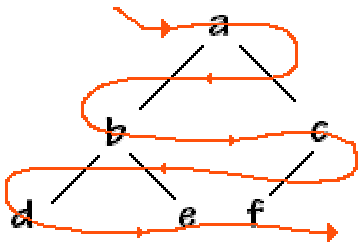
Ordre infixé en Delphi avec la classe :

```
interface
type
  TreeBin = class
    private
      procedure FreeRecur( x :TreeBin ) ;
      procedure PreOrdre ( x : TreeBin);
      procedure PostOrdre ( x : TreeBin);
      procedure InfixeOrdre ( x : TreeBin);
    public
      Info : string;
      filsG , filsD : TreeBin;
      constructor CreerTreeBin(s:string);overload;
      constructor CreerTreeBin(s:string; fg , fd : TreeBin);overload;
      destructor Liberer;
      procedure Prefixe;
      procedure Postfixe;
      procedure Infixe;
  end;
implementation
{--------------------  Méthodes privé ------------------------}
procedure TreeBin.InfixeOrdre ( x : TreeBin ) ;
 // parcours infixé d'un arbre x
begin
 if x<>nil then
 begin
```

```
   InfixeOrdre( x.filsG );
   write(x.info);
   InfixeOrdre( x.filsD );
 end
end;
//autres méthodes ......

{------------------   Méthodes public ----------------------}
procedure Infixe;
// parcours préfixé de l'objet
begin
  InfixeOrdre( self );
end;
//autres méthodes ......

end.
```

Algorithme de parcours en largeur (hiérarchique)

```
Largeur ( Arbre )
si Arbre ≠ ∅  alors
   ajouter Arbre.racine 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
```



La procédure écrira successivement : **abcdef**

Implantation en Delphi avec les variables dynamiques et un TList :

```
type
   parbre = ^arbre;
   arbre = record
              info : string ;
              filsG, filsD: parbre
         end;
```

Nous utilisons un objet Delphi de classe TList pour implanter notre Fifo.

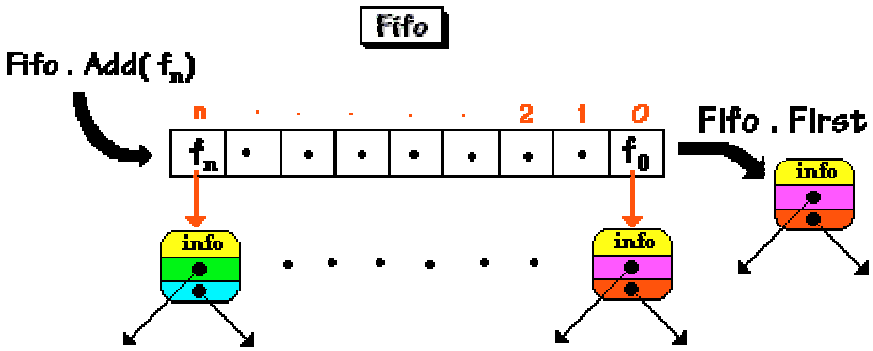
Un **TList** stocke un tableau de pointeurs (Pointer en Delphi). Un objet **TList** est souvent utilisé pour gérer une liste d'objets. **TList** introduit des propriétés et méthodes permettant d'ajouter ou de supprimer des objets de la liste. En particulier afin d'implanter une file de type Fifo, nous utiliserons les membres suivants du **TList**.

méthodes

function Add(x : Pointer): Integer; *Ajoute un nouvel élément en fin de liste et renvoie son rang.*
function First : Pointer; *Renvoie le premier élément du tableau (celui de rang 0).*

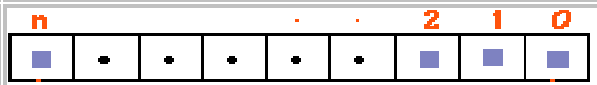
```
 procedure Delete( Index : Integer); Supprime l'élément d'indice spécifié par le paramètre Index.

attributs
 property Count : Integer; Indique le nombre d'entrées utilisées de la liste.
```

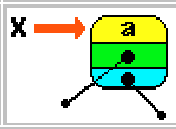
Le **TList** ne contiendra pas les noeuds de l'arbre eux-mêmes mais les pointeurs vers les noeuds (type **parbre** ici) :

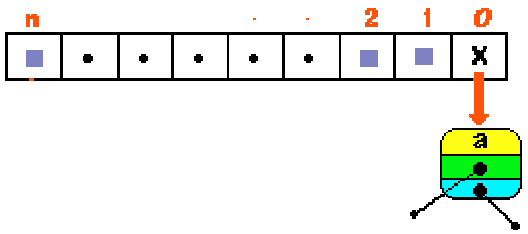
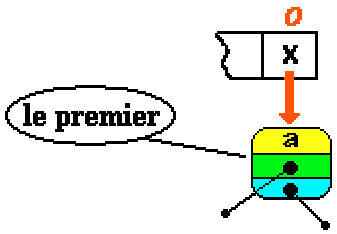
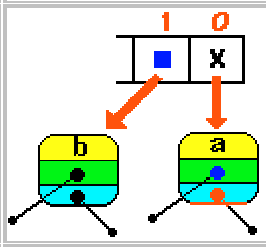
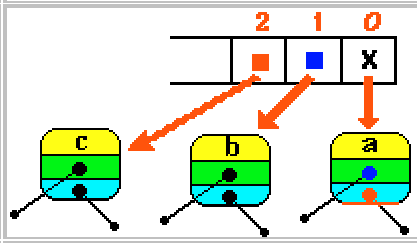
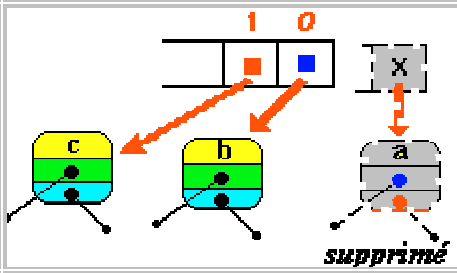
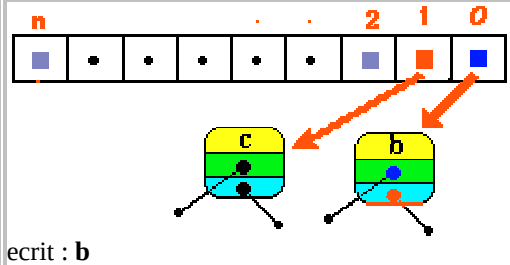



```
procedure Largeur ( x : parbre);
 var Fifo : TList;
 begin
  if f <> nil then
  begin
   Fifo:=TList.Create;  // crée la Fifo
   Fifo.Add( x ); // ajoute la racine x dans Fifo
   while Fifo.Count<>0 do
   begin
    write(parbre(Fifo.First)^.info); // traitement du premier
    if parbre(Fifo.First)^.filsG <> nil then
        Fifo.Add(parbre(Fifo.First)^.filsG); // ajoute le fils gauche du premier dans Fifo
    if parbre(Fifo.First)^.filsD <> nil then
        Fifo.Add(parbre(Fifo.First)^.filsD); // ajoute le fils droit du premier dans Fifo
    Fifo.delete(0); // supprime l'élément de rang 0 (le premier)
   end;
   Fifo.Free ; // supprime la Fifo
  end
 end;
```

On applique la procédure Largeur à l'arbre X afin de parcourir et d'écrire hiérarchiquement les caractères de chaque noeud. Ci-dessous le début d'un suivi d'exécution de la procédure Largeur :

Instruction exécutée	Action sur le TList Fifo
Fifo:=TList.Create;	

// ajouter la racine x dans Fifo
Fifo.Add(x);
	

	
Début de la boucle while : *// traitement du premier* write(parbre(Fifo.First)^.info)	 ecrit : **a**
// ajoute le fils gauche du premier dans Fifo Fifo.Add(parbre(Fifo.First)^.filsG)	
// ajoute le fils droit du premier dans Fifo Fifo.Add(parbre(Fifo.First)^.filsD)	
// supprime l'élément de rang 0 (le premier) Fifo.delete(0);	
Reprise de la boucle while : *// traitement du premier* write(parbre(Fifo.First)^.info) etc	 ecrit : **b**

Programmes Delphi complets

Ci-dessous un programme complet en Delphi (avec variables dynamiques) à exécuter tel quel avec Delphi en mode console :

```
program Tree;
 {les arbres sont parcourus sous les 3 formes
  in, post et pré-fixées
 }
 {$APPTYPE CONSOLE}
 uses   sysutils;
 type
  parbre = ^arbre;
  arbre =
  record
   val: char;
   g, d: parbre
  end;
  var
   rac,arb1,arb2,arb3,arb4,arb5: parbre;
```

```
procedure construit (var rac:parbre;
filsg,filsd:parbre;elt:string);
 // construit un arbre
 begin
  if rac=nil then
  begin
   new(rac);
   with rac^ do
   begin
    val := elt;
    g := filsg;
    d := filsd
   end;
  end
end;{construit}
```

```
procedure edite (f: parbre);
 {infixe avec parentheses}
 begin
  if f <> nil then
   with f^ do
   begin
    write('(');
    edite(g);
    write(val);
    edite(d);
    write(')')
   end
 end;{edite}
```

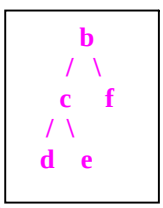
```
procedure postfixe (f: parbre);
 begin
  if f <> nil then
   with f^ do
   begin
    postfixe(g);
    postfixe(d);
    write(val);
   end
 end;{postfixe}
```

```
procedure prefixe (f: parbre);
 begin
  if f <> nil then
   with f^ do
   begin
    write(val);
    prefixe(g);
    prefixe(d)
   end
 end;{prefixe}
```

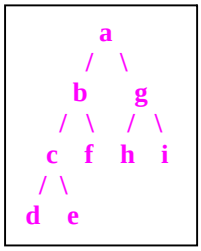
```
procedure infixe (f: parbre);
 begin
  if f <> nil then
   with f^ do
   begin
    infixe(g);
    write(val);
    infixe(d);
   end
 end;{infixe}
```

```
procedure Largeur ( f : parbre);
 var Fifo : TList;
 begin
  if f <> nil then
  begin
   Fifo:=TList.Create;  // crée la Fifo
   Fifo.Add( f ); // ajoute la racine f dans Fifo
   while fifo.count<>0 do
   begin
    write(parbre(Fifo.First)^.info); // traitement du
premier
    if parbre(Fifo.First)^.filsG <> nil then
     Fifo.Add(parbre(Fifo.First)^.filsG); // ajoute le fils
```

```
begin {prog-principal}

 arb1:=nil;
 arb2:=nil;
 arb3:=nil;
 arb4:=nil;
 arb5:=nil;
 construit(arb1,nil,nil,'d');
 construit(arb2,nil,nil,'e');
 construit(arb3,arb1,arb2,'c');
 construit(arb4,nil,nil,'f');
 construit(arb5,arb3,arb4,'b');
 {---------------------------------  }
```

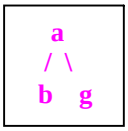


```
gauche du premier dans Fifo
   if parbre(Fifo.First)^.filsD <> nil then
    Fifo.Add(parbre(Fifo.First)^.filsD); // ajoute le fils
droit du premier dans Fifo
   Fifo.delete(0); // supprime l'élément de rang 0 (le
premier)
  end;
  Fifo.Free ; // supprime la Fifo
 end
end;{Largeur}
```




```
arb1:=nil;
arb2:=nil;
arb3:=nil;
construit(arb1,nil,nil,'h');
construit(arb2,nil,nil,'i');
construit(arb3,arb1,arb2,'g');
```


```
{--------------------------------}
rac:=nil;
construit(rac,arb5,arb3,'a');
{--------------------------------}
```




```
writeln('lecture parenthésée:');
edite(rac);   writeln;
writeln('lecture notation infixée:');
infixe(rac);   writeln;  //dcebfahgj
writeln('lecture notation postfixée:');
postfixe(rac);   writeln; //decfbhjga
writeln('lecture notation préfixée:');
prefixe(rac);   writeln; //abcdefghj
writeln('lecture hiérarchique:');
Largeur(rac);   writeln //abgcfhjde
end.
```

Ci-dessous une unit complète en Delphi de la classe TreeBin :

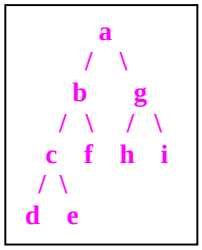
```
unit UTreeBin;

 interface
  uses Classes;
  type
     TreeBin = class
 private
     FInfo : string;
     procedure FreeRecur ( x :TreeBin ) ;
     procedure PreOrdre ( x : TreeBin ; var res :string);
     procedure PostOrdre ( x : TreeBin ; var res :string);
     procedure SymOrdre ( x : TreeBin ; var res :string);
     procedure Hierarchie (x: TreeBin ; var res:string);
 public
      filsG , filsD : TreeBin;
      constructor Create(s:string; fg , fd : TreeBin);
      destructor Liberer;
      function Prefixe : string;
      function Postfixe : string;
      function Infixe : string;
      function Largeur : string;
      property Info : string read FInfo write FInfo;
end;

implementation
{--------------------  Méthodes privé ------------------------}
 procedure TreeBin.FreeRecur ( x : TreeBin ) ;
 // parcours postfixe pour détruire les objets de l'arbre x
 begin
   FreeRecur( x.filsG );
   FreeRecur( x.filsD );
   x.Free
 end;
```

```
procedure TreeBin.PostOrdre (x: TreeBin ;var res:string);
// parcours postfixé d'un arbre x
begin
 if x<>nil then
 begin
   PostOrdre( x.filsG ,res );
   PostOrdre( x.filsD ,res );
   res:=res+x.FInfo
 end
end;

procedure TreeBin.PreOrdre (x: TreeBin ;var res:string);
begin
 if x<>nil then
 begin
  res:=res+x.FInfo;
  PreOrdre( x.filsG ,res );
  PreOrdre( x.filsD ,res );
 end
end;

procedure TreeBin.SymOrdre ( x : TreeBin ;var res:string);
begin
 if x<>nil then
 begin
  SymOrdre ( x.filsG ,res );
  res:=res+x.FInfo;
  SymOrdre ( x.filsD ,res );
 end
end;

procedure TreeBin.Hierarchie (x: TreeBin ;var res:string);
var Fifo : TList;
 begin
 if x <> nil then
 begin
  Fifo:=TList.Create;  // crée la Fifo
  Fifo.Add( x ); // ajoute la racine f dans Fifo
  while fifo.count<>0 do
  begin
    res:=res+ TreeBin (Fifo.First).info;
    if TreeBin (Fifo.First).filsG <> nil then
       Fifo.Add(TreeBin (Fifo.First).filsG); // ajoute le fils gauche du premier dans Fifo
    if TreeBin (Fifo.First).filsD <> nil then
      Fifo.Add(TreeBin (Fifo.First).filsD); // ajoute le fils droit du premier dans Fifo
    Fifo.delete(0); // supprime l'élément de rang 0 (le premier)
  end;
  Fifo.Free ; // supprime la Fifo
 end
end;{ Hierarchie }

{--------------------  Méthodes public -------------------------}
constructor TreeBin.Create(s : string ; fg, fd : TreeBin);
begin
 self.Finfo := s;
 self.filsG := fg;
 self.filsD := fd;
end;
```

```
destructor TreeBin.Liberer;
// la destruction de tout l'arbre :
begin
 FreeRecur (self) ;
 self := nil;
end;

function TreeBin.Postfixe:string;
var parcours:string;
begin
 parcours:='';
 PostOrdre( self , parcours );
 result:=parcours
end;

function TreeBin.Prefixe : string;
var parcours:string;
begin
 parcours:='';
 PreOrdre( self , parcours );
 result:=parcours
end;

function TreeBin.Infixe : string;
 var parcours:string;
 begin
  parcours:='';
  SymOrdre ( self , parcours );
  result:=parcours
 end;

function TreeBin.Largeur : string;
 var parcours:string;
 begin
  parcours:='';
  Hierarchie ( self , parcours );
  result:=parcours
 end;

end.
```

Ci-après le programme d'utilisation de la unit sur l'arbre ci-dessous :

```
program PrTreeBin;
  {$APPTYPE CONSOLE}
 uses
  SysUtils,
  UTreeBin in 'UTreeBin.pas';
 var racine,rac0,rac1,fg,fd : TreeBin;

  begin
   fg:=TreeBin.Create('d',nil,nil);
   fd:=TreeBin.Create('e',nil,nil);
   rac0:=TreeBin.Create('c',fg,fd);
   fd:=TreeBin.Create('f',nil,nil);
   rac1:=TreeBin.Create('b',rac0,fd);
   fg:=TreeBin.Create('h',nil,nil);
   fd:=TreeBin.Create('j',nil,nil);
   rac0:=TreeBin.Create('g',fg,fd);
   racine:=TreeBin.Create('a',rac1,rac0);
```



```
  writeln('lecture notation infixee:');
  writeln(racine.infixe);
  writeln('lecture notation postfixee:');
  writeln(racine.postfixe);
  writeln('lecture notation prefixee:');
  writeln(racine.prefixe);
  writeln('lecture hierarchique:');
  writeln(racine.Largeur);
  readln
 end.
```

Ex-7 : Solution des l'insertion, de la suppression et de la recherche dans une arbre binaire de recherche

Insertion – procédure Placer en Delphi avec les variables dynamiques :

```
 procedure placer (var arbre:parbre ; elt:string);
 {remplissage récursif de l'arbre binaire de recherche
 par adjonctions  successives aux feuilles de l'arbre }
 begin
  if arbre = nil then
  begin
   new(arbre);
   arbre^.info:=elt;
   arbre^.filsG :=nil;
   arbre^.filsD :=nil
  end
  else
  if elt <= arbre^.info then placer (arbre^.filsG ,elt)
  else placer (arbre^.filsD , elt);
 end;
```

procédure Placer en Delphi avec la classe TreeBinRech héritant de TreeBin (ex-6):

```
 interface

 type
    TreeBinRech = class ( TreeBin )
       private
          procedure placerArbre (var arbre:TreeBinRech ; elt : string);
       public
          procedure Placer ( elt : string);
    end;

 implementation
 {--------------------   Méthodes privé ------------------------}
 procedure TreeBinRech.placerArbre (var arbre:TreeBinRech ; elt:string);
 {remplissage récursif de l'arbre binaire de recherche
 par adjonctions  successives aux feuilles de l'arbre }
 begin
  if not Assigned(arbre) then
     arbre:=TreeBinRech.CreerTreeBin( elt )
  else
  if elt <= arbre.info then placerArbre (TreeBinRech(arbre.filsG) ,elt)
   else placerArbre (TreeBinRech(arbre.filsD) , elt);
 end;

 {--------------------   Méthodes public ------------------------}
```

```
 procedure TreeBinRech.Placer ( elt : string);
 begin
    placerArbre ( self, elt )
 end;

end.
```

Procédure Chercher en Delphi avec les variables dynamiques :

```
function Chercher( arbre:parbre; elt:string) : parbre;
begin
 if arbre = nil then
 begin
     result := nil;
     writeln('élément non trouvé')
 end
 else
  if elt = arbre^.info then result := arbre //l'élément est dans ce noeud
  else
    if elt < arbre^.info then
       result := Chercher(arbre^.filsG , elt) //on cherche à gauche
   else
      result := Chercher(arbre^.filsD , elt) //on cherche à droite
end;
```

procédure Chercher en Delphi avec la classe TreeBinRech :

```
interface

type
  TreeBinRech = class ( TreeBin )
     private
        procedure placerArbre (var arbre:TreeBinRech ; elt : string);
        function ChercherArbre( arbre:TreeBinRech; elt:string) : TreeBinRech;
     public
        procedure Placer ( elt : string);
        procedure Chercher ( elt : string);
   end;

implementation
{------------------   Méthodes privé ----------------------}
function TreeBinRech.ChercherArbre( arbre:TreeBinRech; elt:string) : TreeBinRech;
begin
 if not Assigned(arbre) then
 begin
     result := nil;
     writeln('élément non trouvé')
 end
 else
  if elt = arbre.info then result := arbre //l'élément est dans ce noeud
  else
    if elt < arbre.info then
       result := ChercherArbre(TreeBinRech(arbre.filsG) , elt) //on cherche à gauche
   else
      result := ChercherArbre(TreeBinRech(arbre.filsD) , elt) //on cherche à droite
end;

{------------------   Méthodes public ----------------------}
```

```
 procedure TreeBinRech.Chercher ( elt : string);
 begin
    ChercherArbre( self , elt )
 end;

end.
```

procédure Supprimer avec les variables dynamiques :

```
function PlusGrand ( arbre  : parbre ) : parbre;
begin
 if arbre^.filsD = nil then
     result := arbre
 else
   result := PlusGrand( arbre^.filsD ) //on descend à droite
end;

procedure Supprimer (var arbre:parbre; elt:string ) ;
var Node,Loc:parbre;
begin
  if arbre <> nil then
    if elt < arbre^.info  then
       Supprimer (arbre^.filsG, elt )
   else
    if elt > arbre^.info  then
       Supprimer (arbre^.filsD, elt )
    else // elt = arbre^.info
    if arbre^.filsG = nil  then
    begin
       Loc:=arbre;
       arbre := arbre^.filsD;
       dispose(Loc)
    end
    else
      if arbre^.filsD = nil then
      begin
        Loc:=arbre;
        arbre := arbre^.filsG;
        dispose(Loc)
      end
      else
      begin
        Node := PlusGrand ( arbre^.filsG );
        Loc:=Node;
        arbre^.info := Node^.info;
        arbre^.filsG :=Node^.filsG;
        dispose(Loc)
      end
end;
```

procédure Supprimer en Delphi avec la classe TreeBinRech :

```
interface
type
   TreeBinRech = class ( TreeBin )
      private
         procedure placerArbre (var arbre:TreeBinRech ; elt : string);
         function ChercherArbre( arbre:TreeBinRech; elt:string) : TreeBinRech;
         function SupprimerElt( arbre:TreeBinRech; elt:string) : TreeBinRech;
```

```
        function PlusGrand(arbre: TreeBinRech): TreeBinRech;
      public
        procedure Placer ( elt : string);
        procedure Chercher ( elt : string);
        procedure Supprimer ( elt : string);
    end;
implementation
{------------------   Méthodes privé ------------------------}
function TreeBinRech.PlusGrand(arbre: TreeBinRech): TreeBinRech;
 begin
  if not Assigned(Arbre.filsD) then
   result:=Arbre //c'est le pus grand élément
  else
   result:=PlusGrand (TreeBinRech(Arbre.filsD))
 end;

function TreeBinRech.SupprimerElt ( arbre:TreeBinRech; elt:string) : TreeBinRech;
var Noeud:TreeBinRech;
 begin
  if  not Assigned(Arbre) then
   writeln('element ',Elt,' non trouve dans l''arbre')
  else
   if Elt  < Arbre.Info  then
    SupprimerElt (TreeBinRech(Arbre.filsG), Elt ) //on cherche à gauche
   else
    if Elt  >  Arbre.Info  then
     SupprimerElt ( TreeBinRech(Arbre.filsD), Elt ) //on cherche à droite
    else //l'élément est dans ce noeud
    begin
     if Arbre.filsG = nil then //sous-arbre gauche vide
     begin
      Noeud :=Arbre;
      Arbre := TreeBinRech(Arbre.filsD); //remplacer arbre par son sous-arbre droit
     end
     else
      if Arbre.filsD = nil then //sous-arbre droit vide
      begin
       Noeud :=Arbre;
       Arbre :=TreeBinRech(Arbre.filsG); //remplacer arbre par son sous-arbre gauche
      end
      else //le noeud a deux descendants
      begin
       Noeud := PlusGrand( TreeBinRech(Arbre.filsG )); //Node = le max du fils gauche
       Arbre.Info:= Noeud.Info; //remplacer etiquette
       Arbre.filsG := Noeud.filsG;
      end ;
     Noeud.Free; //on supprime ce noeudil;
     writeln('element ',Elt,' supprime !')
    end
  end;
  {------------------   Méthodes public ------------------------}
   procedure TreeBinRech.Supprimer ( elt : string);
   begin
     SupprimerElt ( self , elt )
   end;

 end.
```

Programme Delphi complet

Ci-dessous un programme complet en Delphi (avec variables dynamiques) à exécuter tel quel avec Delphi en mode console; il est conseillé au lecteur de ré-écrire ce programme en utilisant la classe TreeBinRech décrite ci-haut :

```
program arbre_binaire_Rech;
{remplissage d'un arbre binaire de recherche
aussi dénommé arbre binaire ordonné horizontalement  }

{$APPTYPE CONSOLE}

uses   sysutils;
type
   pointeur = ^noeud;
   noeud = record
                info:string;
                filsGauche:pointeur;
                filsDroit:pointeur
           end;
var
   racine,tree:pointeur;

procedure placer_arbre(var arbre:pointeur; elt:string);
{remplissage récursif de l'arbre binaire de recherche}
begin
 if arbre=nil then
 begin
  new(arbre);
  arbre^.info:=elt;
  arbre^.filsGauche:=nil;
  arbre^.filsDroit:=nil
 end
 else
 { - 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)
 }
 if elt<=arbre^.info then placer_arbre(arbre^.filsGauche,elt)
 else placer_arbre(arbre^.filsDroit,elt);
end;


procedure infixe(branche:pointeur);
{lecture symétrique de l'arbre binaire}
begin
  if branche<>nil then
  begin
   infixe(branche^.filsGauche);
   write(branche^.info,' ');
   infixe(branche^.filsDroit);
  end
end;

function ChercherArbre( arbre:pointeur; elt:string):pointeur;
begin
```

```
 if arbre=nil then
 begin
   ChercherArbre:=nil;
   writeln('élément: '+elt+' non trouvé.')
 end
 else
  if elt = arbre^.info then ChercherArbre:=arbre
  else
   if elt < arbre^.info then ChercherArbre:=ChercherArbre(arbre^.filsGauche,elt)
   else ChercherArbre:=ChercherArbre(arbre^.filsDroit,elt)
end;

function PlusGrand ( arbre  : pointeur ) : pointeur;
// renvoie le plus grand élément de l'arbre
begin
 if arbre^.filsDroit = nil then  result := arbre
 else  result := PlusGrand ( arbre^.filsDroit ) //on descend à droite
end;

procedure Supprimer (var arbre:pointeur; elt:string ) ;
var Node,Loc:pointeur;
begin
  if arbre <> nil then
    if elt < arbre^.info  then
       Supprimer (arbre^.filsGauche, elt )
   else
    if elt > arbre^.info  then
       Supprimer (arbre^.filsDroit, elt )
    else // elt = arbre^.info
    if arbre^.filsGauche = nil  then
    begin
       Loc:=arbre;
       arbre := arbre^.filsDroit;
       dispose(Loc)
    end
    else
       if arbre^.filsDroit = nil then
       begin
         Loc:=arbre;
         arbre := arbre^.filsGauche;
         dispose(Loc)
      end
      else
      begin
        Node := PlusGrand ( arbre^.filsGauche );
        Loc:=Node;
        arbre^.info := Node^.info;
        arbre^.filsGauche :=Node^.filsGauche;
        dispose(Loc)
      end
end;

begin
  racine:=nil;
  {soit la liste entrée : e d f a c b u w }
  placer_arbre(racine,'e');
  placer_arbre(racine,'d');
  placer_arbre(racine,'f');
  placer_arbre(racine,'a');
  placer_arbre(racine,'c');
  placer_arbre(racine,'b');
  placer_arbre(racine,'u');
```

```
  placer_arbre(racine,'w');
  {on peut aussi entrer 8 éléments au clavier
  for i:=1 to 8 do
  begin
    readln(entree);
    placer_arbre(racine,entree);
  end; }
  supprimer(racine, 'b');
  writeln('parcours infixé (ou symétrique):');
  infixe(racine);
  writeln;
  tree:=ChercherArbre(racine,'c');
  if tree<>nil then writeln( 'recherche de "c" : ok');
  tree:=ChercherArbre(racine,'g');
  if tree<>nil then writeln( 'recherche de "g" : ok');
  { Notons que le parcours infixé produit une liste des
    éléments info, classée par ordre croissant
  }
end.
```