

# INTRODUCTION

CE POLYCOPIÉ VA ESSAYER ( ET C'EST PAS GAGNÉ )  
DE PRÉSENTER CE QUI EST VU EN ALGORITHMIQUE  
LORS DES DEUX ANNÉES DE CLASSES  
PRÉPARATOIRES À L'EPITA. SA DESTINATION  
PREMIÈRE EST DE SERVIR DE SUPPORT DE COURS  
AUX ÉTUDIANTS DES CLASSES PRÉPARATOIRES  
D'EPITA. MAIS, IL PEUT AUSSI ÊTRE UTILISÉ PAR  
TOUTES LES INSENSÉS DÉSIREUX DE S'INITIER AUX  
CONCEPTS ALGORITHMIQUES DE BASE, AINSI QUE  
PARTOUTS LES ÉTUDIANTS QUI INTÈGRENTE EPITA EN  
COURS ( 2<sup>ÈME</sup> ANNÉE DE CLASSE PRÉPARATOIRE OU  
1<sup>ÈRE</sup> ANNÉE D'INGÉNIERIE ), ET QUI VOUDRAIENT  
CONNAITRE LE CONTENU EXACT DES COURS  
DISPENSÉS EN ALGORITHMIQUE LORS DES ANNÉES  
PRÉCÉDENTES.  
ALORS BONNE LECTURE ET SURTOUT... RONNE  
CHANCE !



101010101010101010  
( Peyln nl' je pechep)

INTRODUCTION .....	2
CHAPITRE 1 .....	5
TYPE ALGEBRIQUE ABSTRAIT .....	5
Déclarations .....	5
Signature .....	5
Hiérarchie des types abstraits .....	6
Propriétés d'un type abstrait .....	7
CHAPITRE 2 .....	9
LES STRUCTURES SEQUENTIELLES .....	9
<i>Les listes linéaires</i> .....	9
Le type « Liste itérative » .....	9
Le type « Liste récursive » .....	10
Extension du type liste .....	11
Concaténation .....	11
La recherche d'un élément .....	12
<i>Représentation des Listes</i> .....	13
Représentation statique .....	13
Représentation dynamique .....	14
Variantes de représentation .....	14
Utilisation d'une sentinelle en tête .....	14
Liste circulaire .....	15
Liste doublement chaînée .....	15
<i>Les piles et les files</i> .....	16
<i>Les piles</i> .....	16
Représentation statique des piles .....	16
Représentation dynamique des piles .....	17
<i>Les files</i> .....	17
Représentation statique des files .....	18
Représentation dynamique des files .....	19
CHAPITRE 3 .....	20
LES STRUCTURES ARBORESCENTES .....	20
<i>Les arbres binaires</i> .....	20
Terminologie .....	21
Mesures sur les arbres binaires .....	22
Arbres binaires particuliers .....	23
Occurrence et numérotation hiérarchique .....	24
Représentation des arbres binaires .....	25
Représentation dynamique .....	25
Représentation statique .....	26
Parcours d'un arbre binaire .....	26
Algorithme de parcours profondeur main gauche .....	29
Parcours d'un arbre étiqueté représentant une expression arithmétique .....	30
<i>Les arbres généraux</i> .....	32
Représentation des arbres généraux .....	34
Représentation sous forme statique-dynamique .....	34
Représentation sous forme de N-uplet .....	35
Représentation sous forme d'arbre binaire .....	36
Parcours d'un arbre général .....	37
CHAPITRE 4 .....	40
LES GRAPHS .....	40
<i>Définitions</i> .....	41
<i>Terminologie</i> .....	42
<i>Types abstraits</i> .....	44
Types abstraits des graphes orientés .....	44
Types abstraits des graphes non orientés .....	45

<i>Représentation des graphes</i>	46
Représentation sous forme de matrice	46
Représentation sous forme de Liste	48
Cas où S est statique	48
<i>Parcours de Graphes</i>	50
Parcours en profondeur	50
Forêt couvrante associée au parcours en profondeur d'un graphe	52
Parcours en profondeur d'un graphe non orienté	53
Parcours en largeur	54
<b>CHAPITRE 5</b>	56
<b>ALGORITHMES DE RECHERCHE – MÉTHODES SIMPLES</b>	56
<i>Définitions</i>	56
<i>Recherche séquentielle</i>	56
Dans une liste non triée	56
Recherche auto-adaptative	57
Dans une liste triée	57
<i>Recherche dichotomique</i>	58
Recherche de la première occurrence d'un élément	60
<i>Recherche par interpolation</i>	61
<b>CHAPITRE 6</b>	63
<b>ALGORITHMES DE RECHERCHE – ARBRES BINAIRES DE RECHERCHE</b>	63
<i>Définition</i>	63
<i>Recherche d'un élément</i>	63
<i>Ajout d'un élément</i>	65
Ajout aux feuilles	65
Construction d'un ABR par ajout successif aux feuilles	67
Ajout à la racine	68
<i>Suppression d'un élément</i>	70
<i>Conclusion</i>	73
<b>CHAPITRE 7</b>	74
<b>ALGORITHMES DE RECHERCHE – ARBRES EQUILIBRES</b>	74
<i>Rotations</i>	74
<i>Arbres A.V.L.</i>	77
Définitions	77
Ajout dans un AVL	78
Principe général de rééquilibrage	78
Supprimer dans un AVL	82
<i>Arbres 2.3.4</i>	83
Définitions	84
Recherche dans un Arbre 2.3.4	84
Ajout dans un arbre 2.3.4	85
Ajout avec éclatements à la remontée	85
Ajout avec éclatements à la descente	87
Représentation des arbres 2.3.4	87
<i>Arbres bicolores</i>	88
Recherche dans un arbre bicolore	90
Ajout dans un arbre bicolore	90
<i>Conclusion</i>	95
<b>BIBLIOGRAPHIE</b>	96
<b>REMERCIEMENTS</b>	96

# Chapitre 1

## Type algébrique abstrait

La conception d'un algorithme peut se faire de deux manières (une alternative, quoi !). La plus intuitive est la **démarche ascendante** qui consiste à décrire un traitement en se plaçant au plus proche de la machine. C'est à dire que les types de données considérés seront ceux qui correspondent au langage de programmation utilisé. Leur implémentation se fera via les déclarations de types du langage et l'on utilisera alors les méthodes (procédures et fonctions) qui leur sont propres.

Un des problèmes est de ne pas pouvoir transcrire facilement l'algorithme vers un autre langage (du C vers le Pascal, ou réciproquement, je ne suis pas regardant !).

L'autre méthode, la **démarche descendante**, consiste à définir des types de données abstraits. C'est à dire que l'on va définir pour un type de donnée son nom, sa syntaxe d'utilisation (opérations, paramètres, etc.) et ses propriétés. Dès lors son utilisation simplifiera grandement l'écriture d'un algorithme dans la mesure où nous serons détachés (Algo lave plus blanc) de toutes les contingences propres à un langage de programmation et à une machine (implémentation des fonctions de manipulation, gestion de la mémoire, etc.).

Dans ce polycopié, les algorithmes seront développés selon la norme d'écriture algorithmique en vigueur dans les classes préparatoires d'**EPITA**.

Ceux qui seraient intéressés par la référence de cette norme peuvent la trouver sur le site de la **SDA** : [www.sda.epita.fr](http://www.sda.epita.fr) (*si le domaine est relayé, ce qui arrive parfois*).

### Déclarations

#### Signature

La **signature** d'un type abstrait est la composée d'un ou de plusieurs noms d'ensembles de valeurs (Entier, Arbre binaire, etc.), appelé(s) **sortes**, et d'un ensemble d'opérations caractérisant le ou les type(s) que l'on veut définir. Les **sortes** correspondent en finale aux types des langages de programmation.

*Exemple :*

```
SORTE Liste, Place
```

Les **opérations** sont définies par leur **nom** et leur **profil**, ce dernier décrivant à quelles **sortes** appartiennent les paramètres et le résultat de ces **opérations**.

*Exemple :*

```
OPERATIONS
```

```
Insérer :      Liste × Entier × Elément → Liste
```

Il est à noter que l'utilisation algorithmique d'une opération se fait de la manière suivante :

Le Nom de l'opération et entre parenthèses les arguments des sortes définies dans le profil. Suivant la déclaration suivante, avec *L* de sorte *Liste*, *i* de sorte *Entier* et *e* de sorte *Elément*, nous aurions :

```
Insérer(L,i,e)
```

Bien entendu, pour éviter une surcharge de parenthèses et lorsque l'on se réfère à des opérations classiques, le nom de l'opération peut en même temps donner la place des arguments à l'aide du caractère de soulignement « \_ ».

Exemple :

OPERATIONS

$_+$  : Entier  $\times$  Entier  $\rightarrow$  Entier  
 $_!$  : Entier  $\rightarrow$  Entier

Nous pourrions alors utiliser ces opérations de la manière suivante :

$x+y$        $x!$        $(x!) + y$        $(x+y)!$

Vous pouvez noter l'utilisation des parenthèses pour lever une éventuelle ambiguïté.  
Une opération dont le profil ne demande pas d'argument est une **constante**.

Exemple :

OPERATIONS

$0$  :  $\rightarrow$  Entier  
Vrai :  $\rightarrow$  Booléen

Pour finir sur la **signature**, voici devant vos yeux ébahis un exemple complet, celle du type **Booléen** :

SORTE Booléen

OPERATIONS

Vrai :  $\rightarrow$  Booléen  
Faux :  $\rightarrow$  Booléen  
 $\neg$  : Booléen  $\rightarrow$  Booléen  
 $\wedge$  : Booléen  $\times$  Booléen  $\rightarrow$  Booléen  
 $\vee$  : Booléen  $\times$  Booléen  $\rightarrow$  Booléen



## Hierarchie des types abstraits

On a la possibilité pour définir un type abstrait de réutiliser ceux précédemment définis. En effet, si un type possède des opérations manipulant des entiers, il est préférable de ne pas devoir redéfinir le type Entier.  
Par exemple, pour le type Vecteur, nous allons réutiliser les types Entier et Elément. Se dessine alors une hiérarchie de ces différents types, celui que l'on est en train de définir et ceux qui le sont déjà, que nous allons donc réutiliser.

SORTE Vecteur  
UTILISE Entier, Elément  
OPERATIONS

ième : Vecteur  $\times$  Entier  $\rightarrow$  Elément  
changer-ième : Vecteur  $\times$  Entier  $\times$  Elément  $\rightarrow$  Liste  
bornesup : Vecteur  $\rightarrow$  Entier  
borneinf : Vecteur  $\rightarrow$  Entier

Dans ce cas, la signature de la sorte Vecteur est l'union des signatures des sortes Entier et Elément à laquelle viennent s'ajouter les nouvelles opérations. Nous pourrons donc utiliser des opérations déjà définies comme l'addition sur les entiers, ce qui pour l'opération *ième* (par exemple) permettra d'utiliser en 2<sup>ème</sup> argument la somme de deux entiers.

On introduit alors une classification des sortes et des opérations :

- Une sorte est dite **définie** si elle est nouvelle
- Une sorte est dite **prédéfinie** si elle est utilisée
- Une opération est dite **interne** si elle renvoie un résultat de sorte définie
- Une opération est dite **observateur** si elle possède au moins un argument de sorte définie et si elle renvoie un résultat de sorte prédéfinie

Dans l'exemple précédent, Vecteur est une **sorte définie**, Entier et Elément sont des **sortes prédéfinies**. Changer-ième est une **opération interne** alors que ième, bornesup et borneinf sont des **observateurs**.

### Propriétés d'un type abstrait

L'idée (qu'elle est bonne !) est de donner un sens aux noms de la signature. C'est à dire que lorsque l'on évoquera un type de donnée, on mesurera immédiatement toutes ses possibilités et ses limites. Dans ce cas, et si l'on veut se détacher de toute contingence matérielle, on énonce les propriétés des opérations sous forme d'axiomes.

La définition d'un type abstrait est donc la composée d'une signature et d'un ensemble d'axiomes. Deux problèmes se posent lors de l'écriture de ces axiomes : la **consistance** et la **complétude**.

La **consistance** correspond au besoin de ne pas avoir d'axiomes contradictoires. C'est à dire deux axiomes rendant pour des arguments identiques des valeurs différentes.

La **complétude** signifie que l'on a écrit suffisamment d'axiomes. En fait, ceux-ci doivent permettre de déduire une valeur pour toute application d'un observateur à une opération interne.

Bien évidemment, il existe des opérations qui ne sont pas décrites partout (personne n'est parfait). On les appelle des **fonctions partielles**. Dans ce cas, et avant de décrire les axiomes utilisant ces opérations, il faut préciser leur domaine de définition. Ce que l'on fait à l'aide d'une **précondition**.

Pour exemple, donnons la définition complète du type Vecteur précédemment évoqué.

**SORTE Vecteur**

**UTILISE** Entier, Elément, Booléen

**OPÉRATIONS**

vect : Entier x Entier → Vecteur  
changer-ième : Vecteur x Entier x Elément → Vecteur  
ième : Vecteur x Entier → Elément  
init : Vecteur x Entier → Booléen  
borneinf : Vecteur → Entier  
bornesup : Vecteur → Entier

**PRÉCONDITION**

ième(v,i) est-défini-ssi Borneinf(v) ≤ i ≤ bornesup(v) & init(v,i)=vrai

**AXIOMES**

```

borneinf(v) ≤ i < bornesup(v) => ième(changer-ième(v,i,e),i)=e

borneinf(v) ≤ i < bornesup(v) & borneinf(v) ≤ j ≤ bornesup(v) & i ≠ j
=> ième(changer-ième(v,i,e),j)=ième(v, j)

init(vect(i,j),k)=Faux

borneinf(v) ≤ i < bornesup(v) => init(changer-ième(v,i,e),i)=Vrai

borneinf(v) ≤ i < bornesup(v) & borneinf(v) ≤ j ≤ bornesup(v) & i ≠ j
=> init(changer-ième(v,i,e),j)=init(v,j)

borneinf(vect(i,j))=i

borneinf(changer-ième(v,i,e))=borneinf(v)

bornesup(vect(i,j))=j

bornesup(changer-ième(v,i,e))=bornesup(v)

```

**AVEC**

v	:	Vecteur
i, j, k	:	Entier
e	:	Elément

La génération spontanée n'existant pas en Algorithmique, nous avons rajouté l'opération **Vect** qui crée un Vecteur à partir de ses bornes. D'autre part, l'opération **ième** n'étant pas définie sur un indice auquel nous n'avons pas affecté d'élément (à l'aide de **changer-ième**), nous avons dû rajouter aussi une opération partielle **Init** dont le seul but est de permettre l'écriture d'une **précondition** sur **ième** (donner son domaine de définition).

Pour conclure, à l'aide de cet exemple, la définition du type vecteur est suffisamment complète pour les raisons suivantes :

- Tout vecteur est le fruit d'une opération **vect** et d'une série d'opérations **changer-ième**
- Les axiomes permettent de déduire le résultat de **Init**, **Bornesup** et **Borneinf** dans tous les cas.
- Les axiomes permettent de déduire le résultat de **ième** quand la précondition est satisfaite.

# Chapitre 2

## Les structures séquentielles

On présentera dans ce chapitre les *listes linéaires*, les *piles* et les *Filres*. Nous donnerons à chaque fois la définition du type abstrait agrémentée de quelques commentaires. Puis nous présenterons diverses représentations de ces types ainsi que quelques algorithmes de manipulation. Les algorithmes seront à chaque fois et dans la mesure du possible fournis de manière abstraite et concrète.

### Les listes linéaires

la *liste linéaire* est la forme la plus simple d'organisation de données que l'on puisse rencontrer. Celles-ci sont stockées les unes à la suite des autres dans des places et permettent divers traitements séquentiels. L'ordre des éléments dans une liste ne dépend pas des éléments eux-mêmes, mais de la place de ceux-ci dans la liste. Il y a plusieurs façons de décrire une liste, soit *itérativement*, soit *récursivement*. Nous allons envisager les deux et voir sur quels types de fonctionnement elles sont respectivement basées. En effet, il nous faut pouvoir créer des éléments, en modifier et en supprimer. De plus (et pour le même prix), nous y ajouterons des fonctionnalités propres aux listes comme la concaténation.

#### Le type « Liste itérative »

SORTE  
Liste, Place



UTILISE  
Entier, Élément

#### OPÉRATIONS

Liste-vide	:	$\rightarrow$ Liste
accès	:	Liste $\times$ Entier $\rightarrow$ Place
contenu	:	Place $\rightarrow$ Élément
ième	:	Liste $\times$ Entier $\rightarrow$ Élément
longueur	:	Liste $\rightarrow$ Entier
supprimer	:	Liste $\times$ Entier $\rightarrow$ Liste
insérer	:	Liste $\times$ Entier $\times$ Élément $\rightarrow$ Liste
succ	:	Place $\rightarrow$ Place

#### PRÉCONDITIONS

accès( $\lambda, k$ ) est-défini-ssi  $1 \leq k \leq$  longueur( $\lambda$ )  
supprimer( $\lambda, k$ ) est-défini-ssi  $1 \leq k \leq$  longueur( $\lambda$ )  
insérer( $\lambda, k, e$ ) est-défini-ssi  $1 \leq k \leq$  longueur( $\lambda$ ) + 1

#### AXIOMES

Longueur(liste-vide) = 0  
 $\lambda \neq$  liste-vide &  $1 \leq k \leq$  longueur( $\lambda$ )  
 $\Rightarrow$  longueur(supprimer( $\lambda, k$ )) = longueur( $\lambda$ ) - 1  
 $1 \leq k \leq$  longueur( $\lambda$ ) + 1  $\Rightarrow$  longueur(insérer( $\lambda, k, e$ )) = longueur( $\lambda$ ) + 1  
ième( $\lambda, k$ ) = contenu(accès( $\lambda, k$ ))  
 $\lambda \neq$  liste-vide &  $1 \leq k \leq$  longueur( $\lambda$ ) &  $1 \leq i < k$

```

=> ième(supprimer(λ, k), i) = ième(λ, i)
λ=liste-vide & 1 ≤ k ≤ longueur(λ) & k ≤ longueur(λ) - 1
=> ième(supprimer(λ, k), i) = ième(λ, i+1)
1 ≤ k ≤ longueur(λ) + 1 & 1 ≤ i < k => ième(insérer(λ, k, e), i) = ième(λ, i)
1 ≤ k ≤ longueur(λ) + 1 & k = i => ième(insérer(λ, k, e), i) = e
1 ≤ k ≤ longueur(λ) + 1 & k < i ≤ longueur(λ) + 1
=> ième(insérer(λ, k, e), i) = ième(λ, i-1)
λ=liste-vide & 1 ≤ k < longueur(λ) => succ(accès(λ, k)) = accès(λ, k+1)

```

AVEC

λ	:	Liste
i, k	:	Entier
e	:	Elément

Nous ne redonnerons plus par la suite la terminologie des éléments utilisés dans ce type (ça ira bien comme ça !). Alors dans le type abstrait qui précède, nous avons :

- Liste et Place comme sortes définies*
- Entier et Elément comme sortes prédéfinies*
- Accès et Succ comme opérations internes de Place*
- Contenu comme observateur de Place*
- Liste-vide, Supprimer et Insérer comme opérations internes de Liste*
- Accès et Longueur comme observateurs de Liste*

Ces opérations n'étant pas définies partout, il y bien sur des préconditions. La définition axiomatique affecte aux diverses fonctions les rôles suivants :

- Liste-Vide* crée une liste sans éléments (une sorte de « constructeur »)
- Accès* permet d'atteindre une place référencée par son rang (le k<sup>ème</sup>)
- Contenu* permet d'obtenir l'élément d'une Place
- Longueur* donne le nombre d'éléments d'une Liste
- Supprimer* permet de détruire un élément (le K<sup>ème</sup>)
- Insérer* permet de créer un élément (en l'insérant devant la K<sup>ème</sup> Place)
- Succ* permet de passer à la place suivante

Le type « Liste récursive »

$$L = \emptyset + E \times L$$

SORTIE

Liste, Place

UTILISE

Elément

OPÉRATIONS

liste-vide	:	→ Liste
tête	:	Liste → Place
fin	:	Liste → Liste
cons	:	Elément x Liste → Liste
premier	:	Liste → Elément
contenu	:	Place → Elément

succ : Place → Place

**PRECONDITIONS**

tête( $\lambda$ ) est-défini-ssi  $\lambda \neq$  liste-vide  
fin( $\lambda$ ) est-défini-ssi  $\lambda \neq$  liste-vide  
premier( $\lambda$ ) est-défini-ssi  $\lambda \neq$  liste-vide

**AXIOMES**

$\lambda \neq$  liste-vide => premier( $\lambda$ ) = contenu(tête(( $\lambda$ )))  
fin(cons(e,  $\lambda$ )) =  $\lambda$   
premier(cons(e,  $\lambda$ )) = e  
 $\lambda \neq$  liste-vide => succ(tête( $\lambda$ )) = tête(fin( $\lambda$ ))

**AVEC**

$\lambda$  : Liste  
e : Elément

Cette présentation correspond à une autre forme d'implémentation des listes linéaires. En fait l'opération de base n'est plus l'*accès* à la K<sup>ème</sup> place, mais l'opération *Tête* qui renvoie la première place d'une liste et l'opération succ qui permet d'atteindre les autres places séquentiellement. Les autres opérations découlant de ce fonctionnement.

Le type abstrait Liste récursif permet bien évidemment de décrire plus facilement des traitements récursifs. Cela dit, cela revient au même puisque toutes les opérations d'un type peuvent être définies en fonction de celles de l'autre type.

*Exemple :*

- tête( $\lambda$ )=accès( $\lambda$ , 1)
- fin( $\lambda$ )=supprimer( $\lambda$ , 1)
- cons(e,  $\lambda$ )=insérer( $\lambda$ , 1, e)

## Extension du type liste

Bien entendu, nous avons souvent besoin d'opérations complémentaires sur les listes comme la **concaténation** de deux listes ou la **recherche d'un élément** dans une liste. Dans ce cas, et ceci est valable pour n'importe quel type défini, on déclare ce que l'on appelle des **extensions** au type. Il est alors inutile de représenter le type abstrait qui est supposé connu (Oui, c'est vrai ! Ca va bien comme ça !). Pour ces deux opérations supplémentaires, nous présenterons le profil (Le bon, hein Tintin !) et les axiomes pour une liste itérative et pour une liste récursive.

### Concaténation

La concaténation de deux listes est l'opération qui permet de les rassembler en les mettant bout à bout. Les éléments de chacune conservent leur place d'origine au sein de leur propre liste, la deuxième liste étant accrochée à la suite de la première.

**OPERATIONS**

concaténer : Liste × Liste → Liste

**AXIOMES (Liste itérative)**

```
longueur(concaténer(λ, λ')) = longueur(λ) + longueur(λ')
1 ≤ i ≤ longueur(λ) => ième(concaténer(λ, λ'), i) = ième(λ, i)
longueur(λ) + 1 ≤ i ≤ longueur(λ) + longueur(λ')
=> ième(concaténer(λ, λ'), i) = ième(λ', i - longueur(λ))
```

**AXIOMES (Liste récursive)**

```
concaténer(liste_vide, λ) = λ
concaténer(cons(e, λ), λ') = cons(e, concaténer(λ, λ'))
```

**AVEC**

λ, λ' : Liste  
e : Elément

**La recherche d'un élément**

La recherche consiste à trouver un élément dans une liste et à retourner sa place si celui-ci est présent. Dans ce cas (J'aime bien cette locution. Et de plus, je fais ce que je veux ! C'est clair Junior !?), le problème est que la recherche n'est pas définie pour un élément non présent. Il faut donc une *précondition* sur la recherche que l'on décrira à l'aide d'une opération auxiliaire *est-présent*.

**OPERATIONS**

```
rechercher : Liste × Elément → Place
est-présent : Liste × Elément → Booléen
```

**PRÉCONDITIONS**

```
rechercher(λ, e) est-défini -> est-présent(λ, e) = vrai
```

**AXIOMES (Liste itérative)**

```
est-présent(liste-vide, e) = faux
e = e' => est-présent(insérer(λ, i, e), e') = vrai
e ≠ e' => est-présent(insérer(λ, i, e), e') = est-présent(λ, e)

est-présent(λ, e) = vrai => contenu(rechercher(λ, e)) = e
```

**AXIOMES (Liste récursive)**

```
est-présent(liste-vide, e) = faux
e = e' => est-présent(cons(e, λ), e') = vrai
e ≠ e' => est-présent(cons(e, λ), e') = est-présent(λ, e)

est-présent(λ, e) = vrai => contenu(rechercher(λ, e)) = e
```

**AVEC**

$\lambda, \lambda'$  : Liste  
 $e, e'$  : Elément

## Représentation des Listes

Les listes comme tous les autres types de données sont représentables de différentes manières. Les deux plus importantes sont la *représentation statique* (à l'aide de tableau) et la *représentation dynamique* (à l'aide de pointeurs et d'enregistrements). Bien sûr, pour des types de données plus élaborés, il sera possible de représenter ceux-ci par des hybrides des deux (statique et dynamique). De plus, il sera éventuellement possible d'avoir plusieurs représentations statiques et plusieurs dynamiques (C'est chouette ! Non ?).

### Représentation statique

Exemple de déclaration algorithmique :

**Constante**

Nbmax = 20

**Type**

Element	= ....	/* Définition du type des éléments */
T_VectNbmaxEntier	= Nbmax Element	

**Variable**

T_VectNbmaxEntier	Liste
Entier	Longueur

Ce qui correspondrait à la structure suivante :

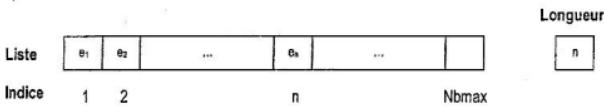


Figure 1. Représentation statique des listes

Le problème posé par les tableaux est la nécessité d'un surdimensionnement. En effet, ils sont statiques. Donc pour être sûr que votre liste de données puisse y être représentée, vous êtes tenus de donner au tableau une taille supérieure à celle de la liste (pour d'éventuels ajouts). Dès lors, vous devez savoir où s'arrêtent vos données dans ce tableau. C'est l'utilité de la variable longueur qui contiendra toujours la taille de votre liste. Pour accéder à une donnée il suffit alors de préciser le nom du tableau et le rang de celle-ci. C'est très simple (un enfant de 5 ans comprendrait. Enfin, je crois...).

Par contre, pour insérer ou supprimer une donnée, vous devrez décaler dans un sens ou dans l'autre tous les éléments se trouvant entre celle-ci et la fin de votre liste.

Enfin, cette représentation est très mal adaptée aux listes récursives.

## Représentation dynamique

Exemple de déclaration algorithmique :

```
Type
    Element = ...
    T_Penreg = ↑ T_Enreg /* type pointeur sur T_Enreg */

    T_Enreg = Enregistrement
        Element Elt
        T_Penreg Lien
    Fin Enregistrement T_Enreg

Variable
    T_Penreg Liste
```

Ce qui correspondrait à la structure suivante :

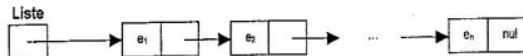


Figure 2. Représentation dynamique des listes

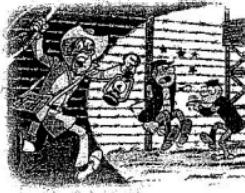
Le pointeur nul représente la fin de liste (liste-vide). Cette représentation utilise à priori plus de place que la précédente dans la mesure où l'on doit stocker la valeur des pointeurs. Mais le nombre d'éléments est toujours celui de la liste, ni plus ni moins. Contrairement à l'implémentation statique pour laquelle il faut surdimensionner le tableau, celle-ci ne nécessite pas de compteur du nombre d'éléments (Longueur). Un inconvénient majeur est de ne pas pouvoir accéder au K<sup>me</sup> élément directement. Par contre, il est facile de concaténer deux chaînes, d'ajouter ou de supprimer un élément sans avoir à tout décaler. Elle est de plus très bien adaptée aux traitements récursifs.

## Variante de représentation

### Utilisation d'une sentinelle en tête

Une possibilité est de ne pas utiliser un pointeur sur l'enregistrement, mais directement un enregistrement pour générer la tête de liste. L'avantage est de ne pas avoir besoin de traitement particulier en insertion devant le premier élément. Dans ce cas, l'élément de l'enregistrement de tête n'est pas utilisé et la déclaration de variables est :

```
Variable
    T_Enreg Liste
```



Ce qui donne :

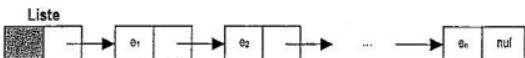


Figure 3. Représentation dynamique d'une liste avec élément de tête.

### Liste circulaire

On peut aussi utiliser des listes circulaires. Dans ce cas, le dernier pointeur n'est pas nul, mais il pointe sur le premier élément de la liste. Pour cela, le pointeur principal de liste référence le dernier élément et pas le premier. Dans ce cas, pour obtenir l'élément de tête, il suffit d'avancer d'un lien. Notons que si la liste n'est composée que d'un élément, celui-ci pointe sur lui-même (Ah Ouais! C'est Géniaaaalll!). La déclaration est alors la même que pour la représentation dynamique de base et cela correspond à la structure suivante :



Figure 4. Représentation dynamique d'une liste circulaire.

### Liste doublement chaînée

Le problème des représentations précédentes est de ne pouvoir aller que dans un sens. En effet, les listes étant généralement ordonnées, il peut être intéressant de revenir sur l'élément précédent, or cette possibilité n'existe pas. Pour y arriver, il suffit de rajouter un lien en sens inverse. Dans ce cas, il faut posséder non seulement un pointeur de tête, mais aussi un pointeur de queue. La déclaration devient :

#### Type

```
Element    = ....          /* Définition du type des éléments */
T_Penreg  = ↑ T_Enreg    /* type pointeur sur T_Enreg */
```

```
T_Enreg   = Enregistrement
           Element Elt
           T_Penreg Suivant, Precedent
Fin Enregistrement T_Enreg
```

#### Variable

```
T_Penreg Premier, Dernier
```

Ce qui correspond à la structure suivante :

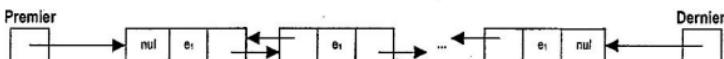


Figure 5. Représentation dynamique d'une liste doublement chaînée.

Bien sûr, il est toujours possible de construire d'autres structures comme par exemple; une liste circulaire doublement chaînée. Enfin pour terminer sur les variantes possibles des listes, citons la simulation de pointeurs dans un tableau. Dans ce cas, les éléments du tableau sont des enregistrements contenant deux champs; l'élément et un entier contenant l'indice de l'élément suivant. Tout est possible ou presque, mais utiliser ce genre de structure n'a aucun intérêt dans la mesure où elle présente tous les inconvénients du statique et du dynamique réunis (un peu comme le Side-Car).

## Les piles et les files

### Les piles

Les piles sont des structures *LIFO* (Last In First Out). C'est à dire que les entrées et les sorties s'effectuent du même coté. L'image la plus simple que l'on puisse donner est la pile d'assiette où, si l'on est totalement terminé (ce qui pour certains étudiants n'est pas gagné), les entrées et les sorties se font au même endroit.

On appelle ce dernier le **sommet** de la pile.

Le type abstrait d'une pile est le suivant :

```
SORTIE
    Pile

UTILISE
    Booléen, Elément

OPÉRATIONS
    Pile-vide      :   → Pile
    empiler        :   Pile × Elément → Pile
    dépiler        :   Pile → Pile
    sommet         :   Pile → Elément
    est-vide       :   Pile → Booléen

PRÉCONDITIONS
    dépiler(p) est-défini-ssi est-vide(p) = Faux
    sommet(p) est-défini-ssi est-vide(p) = Faux

AXIOMES
    dépiler(empiler(p,e)) = p
    sommet(empiler(p,e)) = e
    est-vide(pile-vide) = Vrai
    est-vide(empiler(p,e)) = Faux

AVEC
    p      :   Pile
    e      :   Elément
```



### Représentation statique des piles

Nous avons besoin d'un tableau pour ranger les éléments et d'un entier **Sommet** qui nous permette de savoir en permanence où se situe celui-ci. Ce qui donne:

**Constante**  
Nbmax = 20

**Type**

Element	= ....	<i>/* Définition du type des éléments */</i>
T_Pile	= Nbmax Element	

**Variable**

T_Pile	Pile
Entier	Sommet

L'implémentation correspond alors à celle de la *Figure 1*.

### Représentation dynamique des piles

Dans ce cas, les éléments de la pile sont chaînés entre eux, et le pointeur représente le sommet de celle-ci. Il est à noter un lien de précédence sur les éléments. Ce qui donne :

**Type**

Element	= ....	<i>/* Définition du type des éléments */</i>
T_Ppile	= ↑ T_Pile	<i>/* type pointeur sur T_Pile */</i>

T_Pile	= Enregistrement
	Element Elt
	T_Ppile Precedent
	Fin Enregistrement T_Pile

**Variable**

T_Ppile	Pile
---------	------

L'implémentation correspond alors à celle de la *Figure 2*.

### Les files

Les files sont des structures **FIFO** (First In First Out). C'est à dire que les entrées et les sorties s'effectuent à chaque extrémité de la liste. L'image la plus simple que l'on puisse donner est la file d'attente où (en l'absence de tout resquilleur) la première personne arrivée dans la file sera la première à en sortir. Nous avons donc besoin dans ce cas là de maîtriser la position de l'entrée et celle de la sortie. On référence alors la *Tête* et la *Queue* de la file. Le type abstrait d'une pile est le suivant :

```

SORTIE
File

UTILISE
Booléen, Élément

OPÉATIONS
file-vide   : → File
ajouter      : File × Élément → File
retirer      : File → File

```

premier : File → Elément  
est-vide : File → Booléen

**PRECONDITIONS**

premier(f) est-défini-ssi est-vide(f)=Faux  
retirer(f) est-défini-ssi est-vide(f)=Faux

**AXIOMES**

est-vide(f)=Vrai => premier(ajouter(f,e))=e  
est-vide(f)=Faux => premier(ajouter(f,e))=premier(f)  
  
est-vide(f)=Vrai => retirer(ajouter(f,e))=file-vide  
est-vide(f)=Faux => retirer(ajouter(f,e))=ajouter(retirer(f),e)  
  
est-vide(file-vide)=Vrai  
est-vide(ajouter(f,e))=Faux

**AVEC**

f : File  
e : Elément

## Représentation statique des files

Nous avons besoin d'un tableau pour ranger les éléments et de deux entiers *Tete* et *Queue* qui nous permettent de savoir en permanence où se situe le début et la fin de la file. Ce qui donne:

**Constante**

Nbmax = 8

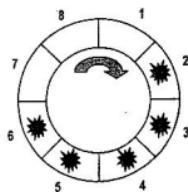
**Type**

Element = .... /\* Définition du type des éléments \*/  
T\_File = Nbmax Element

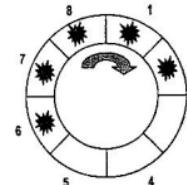
**Variable**

T\_File File  
Entier Tete, Queue

Nous pourrions pour visualiser l'implémentation utiliser la *Figure 1*, mais en fait pour illustrer les débordements, il vaut mieux représenter cette file de façon circulaire, ce qui donne pour deux cas différents:



Tête : 2 Queue : 6



Tête : 6 Queue : 2

Figure 6. Représentation circulaire statique d'une file

Pour ces exemples, nous avons fixé Nbmax à 8. En fait les valeurs de Tête et de Queue avancent d'un rang à chaque fois, exception faite de la bascule de 8 à 1. En fait ce débordement est géré de façon extrêmement simple, il suffit d'utiliser un modulo Nbmax, soit 8 dans ce cas.

### Représentation dynamique des files

Dans ce cas, les éléments de la file sont chaînés entre eux, et les pointeurs *Tête* et *Queue* représentent les deux extrémités de celle-ci. Ce qui donne :

```
Type
Element = ....          /* Définition du type des éléments */
T_Pfile = ↑ T_file      /* type pointeur sur T_File */

T_File = Enregistrement
Element Elt
T_Pfile Suivant
Fin Enregistrement T_File
Variable
T_Pfile Tete, Queue
```

Nous pourrions alors utiliser cette structure de deux façons, une classique et une circulaire, ce qui donnerait l'aspect suivant :

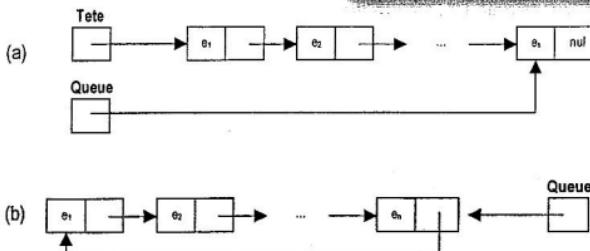


Figure 7. Représentation dynamique d'une file

On peut remarquer que dans le cas d'une représentation circulaire (Figure 7(b)), le pointeur de Tête n'a plus aucune utilité. Il suffit de suivre le lien « Suivant » à partir du dernier pour déterminer le premier élément. Il est, d'autre part, possible d'utiliser le système de sentinelle de la Figure 3, à savoir prendre un élément complet pour les deux pointeurs (Tête et Queue).

# Chapitre 3

## Les structures arborescentes

Un arbre est constitué d'éléments appelés nœuds organisés de façon hiérarchique. L'élément de base est la racine (Incroyable !). En informatique, comme dans la vraie vie, les arbres se rencontrent un peu partout : construction de répertoire, système d'exploitation, compilateur, etc.

Du à leur propriété structurelle récursive, les manipulations et les déclarations des arbres se font naturellement sous une forme récursive (un chêne, un chêne, un chêne, etc.). Ce qui ne veut pas dire que la forme itérative ne leur soit pas applicable. Les arbres se rencontrent à la base sous deux formes : les arbres binaires et les arbres généraux, la première étant une restriction de la deuxième.

### Les arbres binaires

La définition récursive d'un arbre binaire est  $B = \langle o, B1, B2 \rangle$  où  $o$  est le nœud racine, et  $B1$  et  $B2$  sont deux arbres binaires disjoints. Un arbre binaire vide est représenté par  $\emptyset$ .

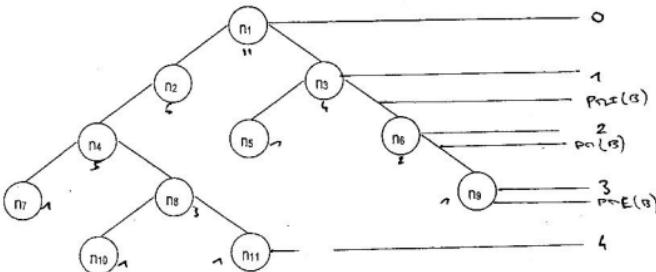


Figure 1. Représentation graphique d'un arbre binaire

Pour cet exemple, nous avons associé à chaque nœud un numéro pour pouvoir les distinguer les uns des autres. Une chose importante à remarquer est la *non symétrie* d'un arbre binaire.

Le type abstrait d'un arbre binaire est le suivant :

```
SORTIE
ArbreBinaire

UTILISE
Noeud, Elément

OPERATIONS
Arbre-vide   :   → ArbreBinaire
<_,_,_>      :   Nœud × ArbreBinaire × ArbreBinaire → ArbreBinaire
racine       :   ArbreBinaire → Nœud
```

$g$  : ArbreBinaire  $\rightarrow$  ArbreBinaire  
 $d$  : ArbreBinaire  $\rightarrow$  ArbreBinaire  
 contenu : Nœud  $\rightarrow$  Élément

#### PRECONDITIONS

$\text{racine}(B)$  est-défini-ssi  $B \neq \text{Arbre-vide}$   
 $g(B)$  est-défini-ssi  $B \neq \text{Arbre-vide}$   
 $d(B)$  est-défini-ssi  $B \neq \text{Arbre-vide}$

#### AXIOMES

$\text{racine } (<\!\!o, B_1, B_2\!\!>) = o$   
 $g(<\!\!o, B_1, B_2\!\!>) = B_1$   
 $d(<\!\!o, B_1, B_2\!\!>) = B_2$

#### AVEC

$\circ$  : Nœud  
 $B, B_1, B_2$  : ArbreBinaire



## Terminologie

La manipulation des arbres binaires nécessite l'utilisation d'un vocabulaire approprié dont voici les principaux éléments :

Soit un arbre binaire  $B = <\!\!o, B_1, B_2\!\!>$

- $o$  est la **racine** de  $B$
- $B_1$  est le **sous-arbre gauche** de  $B$
- $B_2$  est le **sous-arbre droit** de  $B$
- La racine du sous-arbre gauche (sous-arbre droit) d'un nœud est son **fils gauche** (**fils droit**)
- Le **lien** entre un nœud et son fils gauche (fils droit) est appelé **lien gauche** (**lien droit**)
- Un nœud  $n_j$  (seulement quand il approche de l'écurie) ayant pour fils gauche (fils droit) un nœud  $n_j$  est appelé **père** de  $n_j$
- Deux nœuds de même père sont dits **frères**
- Un nœud  $n_j$  est appelé **ascendant** (**descendant**) d'un nœud  $n_i$  si, et seulement si,  $n_j$  est le père (fils) de  $n_i$  ou un **ascendant** (**descendant**) du père (fils) de  $n_i$
- Les nœuds d'un arbre binaire ont au plus deux fils
- Un nœud ayant deux fils est appelé **nœud interne** ou **point double**
- Un nœud n'ayant qu'un fils gauche (fils droit) est appelé **point simple à gauche** (**point simple à droite**), ou **nœud interne au sens large**
- Un nœud n'ayant pas de fils est appelé **nœud externe** ou **feuille**
- Tout chemin allant de la racine de  $B$  à une feuille de  $B$  est appelé **branche** de  $B$
- Un arbre binaire possède autant de branches que de feuilles
- Le chemin obtenu en partant de la racine et ne suivant que des liens gauches (liens droits) est appelé **bord gauche** de  $B$  (**bord droit** de  $B$ )

Remarques diverses : Contrairement à la vraie vie, les arbres binaires ne présentent pas de variétés à feuillage persistant ou caducque. Quand je veux supprimer une feuille, je la fais. Mais le plus fort, c'est que j'en crée une (de feuille) quand je veux. On oubliera aussi les notions familiale de cousinage et autres.

## Mesures sur les arbres binaires

Nous allons voir maintenant des opérations sur les arbres qui vont nous permettre de prendre diverses mesures et de pouvoir alors déterminer la complexité des différents algorithmes qui leur sont appliqués (aux arbres évidemment, de quoi on parle là ?). Ces opérations sont les suivantes :

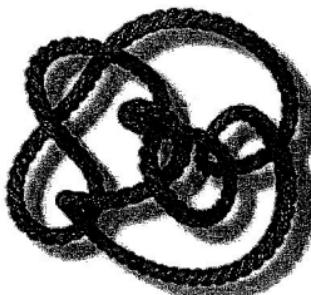
- La **taille** d'un arbre correspond au nombres de ses nœuds, elle est définie par :  
 $T(\text{arbre-vide})=0$   
 $T(B=\langle\alpha, B_1, B_2\rangle)=1+T(B_1)+T(B_2)$
- La **hauteur, profondeur ou niveau** d'un nœud est définie comme suit : soit un nœud  $x$  d'un arbre  $B$ , on a :  
 $H(x)=0$  si  $x$  est la racine de  $B$   
 $H(x)=1+H(y)$  si  $y$  est le père de  $x$
- La **hauteur, profondeur** d'un arbre  $B$  est définie par :  
 $H(B)=\max\{H(x) ; x \text{ nœud de } B\}$
- La **longueur de cheminement** d'un arbre  $B$  est définie par :  
 $LC(B)=\sum H(x) ; x \text{ nœuds de } B$
- La **longueur de cheminement externe** d'un arbre  $B$  est définie par :  
 $LCE(B)=\sum f_i ; f_i \text{ feuilles de } B$
- La **longueur de cheminement interne** d'un arbre  $B$  est définie par :  
 $LCI(B)=\sum H(x) ; x \text{ nœuds internes de } B$

On a alors la relation :  $LC(B)=LCE(B)+LCI(B)$

- La **profondeur moyenne interne** d'un arbre  $B$  est définie par :  
 $PMI(B)=LCI(B)/Nbi$  ;  $Nbi$  nombre de nœuds internes de  $B$
- La **profondeur moyenne externe** d'un arbre  $B$  est définie par :  
 $PME(B)=LCE(B)/Nbf$  ;  $Nbf$  nombre de feuilles de  $B$
- La **profondeur moyenne** d'un arbre  $B$  est définie par :  
 $PM(B)=LC(B)/T(B)$

**Caractéristiques des nœuds (appliquée à l'exemple de la Figure 1) :**

- $n_1$  est la racine de  $B$
- $n_2$  est son fils gauche et  $n_3$  son fils droit
- $n_5$  et  $n_6$  sont frères
- $n_1, n_3, n_4$  et  $n_8$  sont des points doubles
- $n_2$  est un point simple à gauche
- $n_6$  est un point simple à droite
- $n_5, n_7, n_9, n_{10}$  et  $n_{11}$  sont des feuilles
- $(n_1, n_2, n_4, n_7)$  est le bord gauche
- $(n_1, n_3, n_6, n_9)$  est le bord droit
- $(n_1, n_2, n_4, n_8, n_{10})$  est une branche de  $B$
- Hauteurs :
  - $H(n_1)=0$
  - $H(n_2)=H(n_3)=1$



- $H(n_4)=H(n_5)=H(n_6)=2$
- $H(n_7)=H(n_8)=H(n_9)=3$
- $H(n_{10})=H(n_{11})=4$

#### Caractéristiques de l'arbre :

- Hauteur :
  - $H(B)=4$
- Taille :
  - $T(B)=11$
- Longueur de cheminement :
  - $LC(B)=25$
  - $LCI(B)=9$
  - $LCE(B)=16$
- Profondeur moyenne :
  - $PM(B)=25/11 \approx 2,272$
  - ~~PM(B)~~ $=9/6 \approx 1,5$
  - ~~PM(B)~~ $=16/5 \approx 3,2$

#### Arbres binaires particuliers

Il existe des formes particulières d'arbres binaires qu'il faut connaître dans la mesure où leur spécificité permet de modifier les algorithmes sur les arbres, voire d'utiliser des algorithmes propres à leur structure. Ces arbres présentés en figure 2 sont :

- Dégénérés ou filiformes
- Complets
- Parfaits
- Localement complets

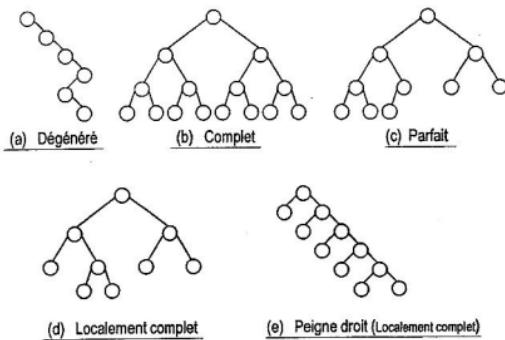


Figure 2. Arbres binaires particuliers

- Les arbres dégénérés *Figure 2 (a)* ne sont constitués que de points simples à gauche ou à droite.
- Les arbres complets *Figure 2 (b)* voient tous leurs niveaux remplis
- Les arbres parfaits *Figure 2 (c)* voient tous leurs niveaux remplis excepté le dernier qui a toutes ses feuilles ramenées complètement à gauche.
- Les arbres localement complets *Figure 2 (d)* ne sont constitués que de points doubles et de feuilles. On peut constater sur la *Figure 2 (e)* une particularité des arbres localement complets, le peigne droit dont tous les fils gauches sont des feuilles. Il existe bien sûr le peigne gauche.

### Occurrence et numérotation hiérarchique

Une façon de décrire un arbre binaire est de lui associer un mot formé de 0 et de 1. Ce mot est appelé occurrence du nœud. Par définition, la racine d'un arbre est noté  $\epsilon$  et si un nœud a pour occurrence  $\mu$  alors, son fils gauche à pour occurrence  $\mu_0$  et son fils droit  $\mu_1$ . Pour l'arbre de la *figure 1*, nous aurions alors :

$$B = \{\epsilon, 0, 1, 00, 10, 11, 000, 001, 111, 0010, 0011\}$$

*ε*      *0*      *1*  
*00*      *001*      *10*  
*0010*      *0011*      *111*

D'autre part, les nœuds peuvent être numérotés de façon hiérarchique. C'est à dire qu'au lieu, comme sur la *figure 1*, de les numérotter par niveau, si l'on a un nœud  $n_i$  alors son fils gauche sera le nœud  $n_{2i}$  et son fils droit le nœud  $n_{2i+1}$ . Ce qui pour l'exemple de la *figure 1* donnerait :

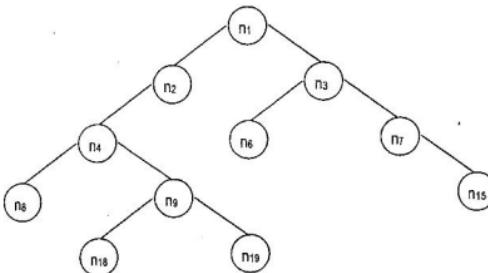


Figure 3. Numérotation hiérarchique

Nous verrons plus loin que cette numérotation présente, entre autres, un intérêt sur certains arbres lors d'une représentation statique.

## Représentation des arbres binaires

Comme pour les structures séquentielles, nous avons la possibilité de représenter les arbres en mémoire sous forme dynamique ou sous forme statique. Cette dernière pouvant être utilisée pour simuler la première.

### Représentation dynamique

Cette représentation est un quasi calque de la structure d'un arbre binaire. Chaque Nœud contient un élément (l'étiquette) et deux liens : un vers le fils gauche et l'autre sur le fils droit. Ce qui donne :

#### Types

```
Element  = ...           /* Définition du type des éléments */  
T_Arbre  = ↑ T_Noeud   /* type pointeur sur T_Noeud */
```

```
T_Noeud = Enregistrement
```

```
    Element  Elt
```

```
    T_Arbre Fg,fd
```

```
Fin Enregistrement T_Noeud
```

#### Variables

```
T_Arbre B
```

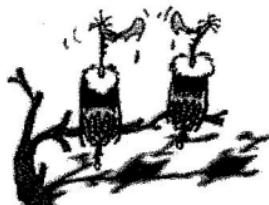
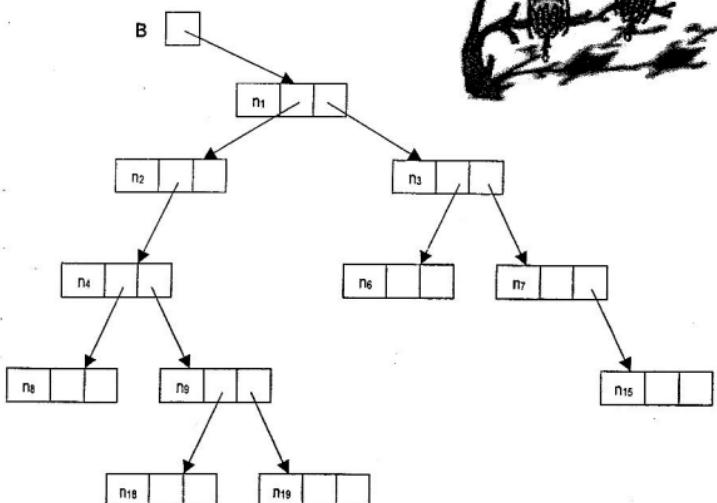


Figure 4. Représentation dynamique d'un arbre binaire.

Dans ce type de représentation (adaptation en *figure 4* de l'arbre donné en *figure 3*), les correspondances entre les opérations abstraites et l'implémentation dynamique du type sont les suivantes :

- $B = \text{Arbre-vide} \Leftrightarrow B = \text{nul}$
- $B \leftarrow \langle \_, \_, \_ \rangle \Leftrightarrow \text{Allouer}(B)$
- $\text{racine}(B) \Leftrightarrow B \uparrow$
- $g(B) \Leftrightarrow B \uparrow.\text{fg}$
- $d(B) \Leftrightarrow B \uparrow.\text{fd}$
- $\text{Contenu}(\text{racine}(B)) \Leftrightarrow B \uparrow.\text{Elt}$

### Représentation statique

Dans cette représentation, l'arbre sera contenu dans un vecteur (tableau à une dimension) d'enregistrements contenant chacun 3 champs : l'élément et deux entiers (un pour le fils gauche et l'autre pour le fils droit). Ces derniers (les deux entiers) référencant dans le vecteur l'indice du fils concerné (Suis-je assez clair ?). La base du Vecteur étant 1, nous pouvons utiliser 0 comme référence de pointeur nul (l'arbre vide, quoi ?!). De plus, il nous faut la position de la racine donnée par un entier.

*Remarque : Lors d'une implémentation en C, il faudra alors utiliser -1, la base étant 0.*

Ceci étant voilà la déclaration algorithmique d'une telle structure :

#### Constante

$\text{Nbmax} = 8$

#### Type

```
Element    = ...      /* Définition du type des éléments */
T_Nœud    = Enregistrement
    Element   Elt
    Entier    fg, fd
Fin Enregistrement T_Nœud
T_VectNbmaxNœud = Nbmax T_Nœud
T_Arbre    = Enregistrement
    T_VectNbmaxNœud Nœuds
    Entier            Racine
Fin Enregistrement T_Arbre
```

#### Variable

$T_Arbre B$

Dans ce type de représentation, les correspondances entre les opérations abstraites et l'implémentation statique du type sont les suivantes :

- $B = \text{Arbre-vide} \Leftrightarrow B.\text{racine}=0$
- $\text{racine}(B) \Leftrightarrow B.\text{racine}$
- $g(B) \Leftrightarrow B.\text{Nœuds}(B.\text{racine}).\text{fg}$
- $d(B) \Leftrightarrow B.\text{Nœuds}(B.\text{racine}).\text{fd}$
- $\text{Contenu}(\text{racine}(B)) \Leftrightarrow B.\text{Nœuds}(B.\text{racine}).\text{Elt}$

Adaptée à l'arbre de la figure 1, nous aurions le tableau présenté en Figure 5. En fait, dans ce type d'arbre, la racine peut être située à n'importe quel indice. Ce qui fait que l'on peut représenter dans un même tableau plusieurs arbres en même temps, une forêt par exemple (Et là, je suis très sérieux.). Il suffit simplement de maîtriser pour chacun la position de sa racine.

Elt	fg	Fd
1	n <sub>1</sub>	2
2	n <sub>2</sub>	4
3	n <sub>3</sub>	5
4	n <sub>4</sub>	7
5	n <sub>5</sub>	0
6	n <sub>6</sub>	0
7	n <sub>7</sub>	0
8	n <sub>8</sub>	10
9	n <sub>9</sub>	0
10	n <sub>10</sub>	0
11	n <sub>11</sub>	0
12		

Figure 5. Représentation statique d'un arbre binaire.

Elt	fg	fd	Elt
1	n <sub>1</sub>	2	1
2	n <sub>2</sub>	4	2
3	n <sub>3</sub>	6	3
4	n <sub>4</sub>	8	4
5			5
6	n <sub>6</sub>	0	6
7	n <sub>7</sub>	0	7
8	n <sub>8</sub>	0	8
9	n <sub>9</sub>	18	9
10			10
11			11
12			12
13			13
14			14
15	n <sub>15</sub>	0	15
16			16
17			17
18	n <sub>18</sub>	0	18
19	n <sub>19</sub>	0	19
20			20

a

b

Figure 6. Représentation statique d'un arbre binaire numéroté hiérarchiquement.

D'autre part, si nous avions utilisé, pour le même arbre, une numérotation hiérarchique comme en figure 3, nous aurions obtenu le tableau suivant (Figure 6a):

Comme on peut le constater sur la figure 6a, cette numérotation présente deux défauts majeurs :

- Elle nous oblige à fixer la racine d'un arbre non vide en indice 1
- Elle génère des trous dans le tableau et par conséquent un taux d'occupation assez faible.

Cela dit, elle est idéale pour les *arbres complets* ou *parfaits* dans la mesure où, pour ceux-ci tous les niveaux, sauf éventuellement le dernier, sont remplis, donc pas de trous (Golfeurs du Monde, Désolé !).

De plus, un nœud  $n_i$  ayant pour fils gauche un nœud  $n_{2i}$  et pour fils droit un nœud  $n_{2i+1}$ , nous n'avons plus besoin de référencer le fils gauche et le fils droit. Ce qui permet de représenter l'arbre à l'aide d'un simple vecteur d'éléments comme montré sur la figure 6b.

*Remarque 1 : Il faut trouver un moyen à l'aide de l'élément pour référencer un arbre vide (la racine à 0, par exemple).*

*Remarque 2 : Cette représentation permet, sans avoir à le mémoriser, de connaître le père d'un nœud. Il suffit pour un nœud  $n_i$  avec  $i > 1$  de rechercher le nœud  $n_{i/2}$ .*

*Remarque 3 : Nous sommes toujours tenus d'avoir une racine en 1 pour un arbre non vide.*

Pour ce type de représentation (figure 6b), la déclaration algorithmique deviendrait :

```
Constante
    Nbmax = 8
Type
    Element = .... /* Définition du type des éléments */
    T_VectNbmaxNoeud = Nbmax Element
    T_Arbre = Enregistrement
        T_VectNbmaxNoeud Noeuds
        Entier Racine
    Fin Enregistrement T_Arbre
Variable
    T_Arbre B
```

Et les correspondances entre les opérations abstraites et l'implémentation statique du type seraient les suivantes :

- $B = \text{Arbre-vide} \Leftrightarrow B.\text{racine}=0$
- $\text{racine}(B) \Leftrightarrow B.\text{racine}$
- $g(B) \Leftrightarrow B.\text{Nœuds}(2^*B.\text{racine})$
- $d(B) \Leftrightarrow B.\text{Nœuds}(2^*B.\text{racine}+1)$
- $\text{Contenu}(\text{racine}(B)) \Leftrightarrow B.\text{Nœuds}(B.\text{racine})$



## Parcours d'un arbre binaire

De nombreux algorithmes sur les arbres examinent systématiquement tous les nœuds d'un arbre pour y effectuer un traitement particulier. Cette opération s'appelle le *parcours* d'un arbre. Il existe plusieurs formes de parcours d'arbre : le *parcours en largeur* (Aaaah...) et le *parcours en profondeur* (Oooh...). Le premier consiste à passer en revue tous les nœuds niveau par niveau. Le deuxième consiste à étudier les nœuds en descendant à chaque fois le plus loin possible dans l'arbre. Nous n'allons présenter que le parcours en profondeur. Celui-ci sera une forme dérivée des sorties labyrinthiques. C'est à dire que l'on pose sa main gauche sur un mur, et que l'on avance en laissant celle-ci posée. Dans la plupart des cas (Franquin et ses idées noires vous en fourniront un autre), nous arriverons fatallement à la sortie. Cette forme dérivée de parcours sur les arbres binaires s'appelle le *parcours en profondeur main gauche*.

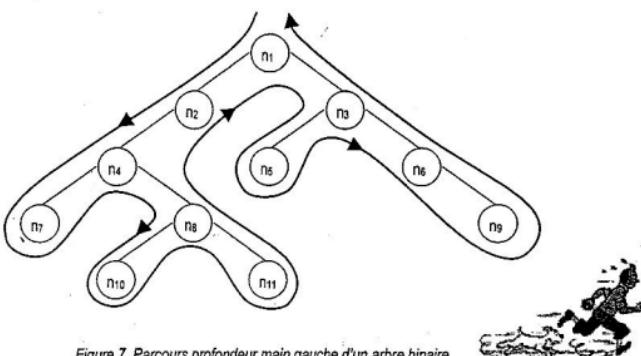


Figure 7. Parcours profondeur main gauche d'un arbre binaire

Sur le parcours de la Figure 7, nous n'avons pas représenté les sous-arbres vides, mais ceux-ci doivent être pris en compte si l'on veut admettre que chaque nœud est rencontré trois fois lors du parcours (Figure 8) :

- à la descente vers le sous-arbre gauche,
- au passage du sous-arbre gauche vers le sous-arbre droit
- à la remontée depuis le sous-arbre droit.

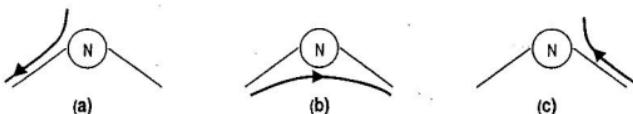


Figure 8. Rencontres d'un Nœud pendant un parcours

## Algorithme de parcours profondeur main gauche

Lors du parcours, chaque nœud étant rencontré trois fois, nous pouvons affecter à chacun d'eux un traitement particulier. De plus, si l'on intègre les arbres vides, nous pouvons aussi y faire correspondre un traitement. Ce qui donne l'algorithme abstrait suivant, de parcours récursif d'un arbre :

```
Algorithme procedure parcours
Paramètres locaux
    Arbre B
Debut
    Si B=arbre-vide alors
        TERMINAISON          /* traitement de terminaison */
    Sinon
        TRT1                  /* traitement 1 (préfixe) */
        parcours(g(B))
        TRT2                  /* traitement 2 (infixe) */
        parcours(d(B))
        TRT3                  /* traitement 3 (postfixe) */
    Fin si
Fin algorithme procedure parcours
```

Pour bien mettre en évidence les différences entre les trois ordres induits par ce parcours, nous allons remplacer à chaque fois chacun d'eux par un ordre d'affichage du nœud (*Ecrire(contenu(racine(B)))*). Les autres traitements étant vides. Voilà ce qui serait obtenu à chaque fois si l'on appliquait cet algorithme de parcours à l'arbre binaire de la figure 7 :  
*(Copié à la main)*

- (a) ordre préfixe (Figure 8 (a)).      n<sub>1</sub>, n<sub>2</sub>, n<sub>4</sub>, n<sub>7</sub>, n<sub>8</sub>, n<sub>10</sub>, n<sub>11</sub>, n<sub>3</sub>, n<sub>5</sub>, n<sub>6</sub>, n<sub>9</sub>
- (b) ordre infixe (Figure 8 (b)).      n<sub>7</sub>, n<sub>4</sub>, n<sub>10</sub>, n<sub>8</sub>, n<sub>11</sub>, n<sub>2</sub>, n<sub>1</sub>, n<sub>5</sub>, n<sub>3</sub>, n<sub>6</sub>, n<sub>9</sub>
- (c) ordre postfixe (Figure 8 (c)).      n<sub>7</sub>, n<sub>10</sub>, n<sub>11</sub>, n<sub>8</sub>, n<sub>4</sub>, n<sub>2</sub>, n<sub>5</sub>, n<sub>9</sub>, n<sub>6</sub>, n<sub>3</sub>, n<sub>1</sub>

On constate que selon le type de parcours l'ordre des nœuds diffère (Fer !).

Remarque 1 : L'ordre hiérarchique n'apparaît pas dans la mesure où celui-ci correspond à un parcours en largeur (par niveau) de l'arbre.

Remarque 2 : Tous les traitements peuvent être utilisés en même temps pour, éventuellement, réaliser des choses diverses dans un ordre différent, ou alors parce que ces traitements sont complémentaires. Nous allons voir dans le paragraphe suivant la nécessité d'utiliser plusieurs de ces traitements.

## Parcours d'un arbre étiqueté représentant une expression arithmétique

Nous pouvons utiliser un arbre binaire pour représenter une expression arithmétique. Dans ce cas, les nœuds internes sont les opérateurs et les feuilles sont les opérandes. Ce qui, pour l'expression arithmétique  $(a + ((b * 4) / 3))$ , pourrait donner :

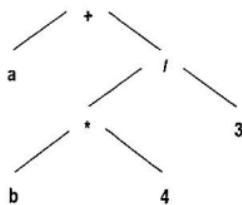


Figure 9. Arbre binaire étiqueté représentant une expression arithmétique

*Remarque : Cette expression n'utilise que des opérateurs binaires (ayant deux opérandes), il est donc localement complet.*

Pour l'arbre de la figure 9, le parcours selon les différents ordres donnerait :

- |                      |               |
|----------------------|---------------|
| (a) ordre préfixe.   | + a * b 4 3   |
| (b) ordre infixe.    | a + b * 4 / 3 |
| (c) ordre postfixe . | a b 4 * 3 / + |

On constate que l'ordre préfixe et l'ordre postfixe sont non ambigus, le premier correspondant à une notation en polonaise et le deuxième à une notation en polonaise inversée. Ce n'est malheureusement pas le cas de l'ordre infixe pour qui cette série pourrait correspondre à un autre arbre donné en figure 10.

En effet les notations en polonaise ou en polonaise inversée font précéder ou suivre les opérateurs de leurs opérandes. Or pour la notation en ordre infixe, nous ne savons pas si  $a+b*4$  correspond à  $(a+b)*4$  ou à  $a+(b*4)$ . En fait cette ambiguïté peut être levée à l'aide de parenthèses qui forceraient alors la priorité des opérateurs.

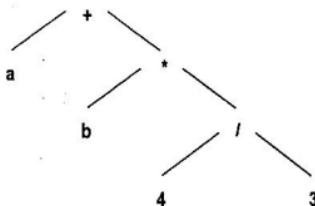


Figure 10. Arbre binaire étiqueté représentant une expression arithmétique donnant le même parcours infixe que celui de la figure 9.

Il suffit alors de modifier l'algorithme de parcours en profondeur main gauche donné précédemment de la manière suivante :

```
Algorithme procedure parcoursinfixe
Paramètres locaux
    Arbre B
Debut
    Si (g(B)=arbre-vide) et (d(B)=arbre-vide) alors /* feuille */
        Ecrire(contenu(racine(B)))           /* traitement de terminaison */
    Sinon
        Ecrire ('(')
        parcours(g(B))
        Ecrire(contenu(racine(B)))           /* traitement 1 (préfixe) */
        parcours(d(B))
        Ecrire (')')                         /* traitement 2 (infixe) */
    Fin si
Fin algorithme procedure parcoursinfixe
```

Ce qui pour les arbres donnés en figure 9 et en figure 10 donnerait :

- Figure 9.       $(a + ((b^4)/3))$
- Figure 10.     $(a + (b^*(4/3)))$

Remarque : Dans l'algorithme, nous aurions pu modifier le test qui vérifie si le nœud sur lequel nous sommes est une feuille. Pour cela, il aurait fallu faire une extension au type abstrait ArbreBinaire en créant une nouvelle opération comme suit :

```
OPERATIONS
feuille      : ArbreBinaire → Booléen

AXIOMES
feuille(B)=Vrai → B≠arbre-vide & g(B)=arbre-vide & d(B)=arbre-vide
```

Le début de l'algorithme serait alors :

```
Algorithme procedure parcoursinfixe
Paramètres locaux
    Arbre B
Debut
    Si feuille(B) alors           /* feuille */
```

## Les arbres généraux

Un arbre général ou arbre est une structure arborescente où le nombre de fils n'est pas limité à deux. La définition récursive d'un arbre général est  $A = \langle o, A_1, \dots, A_N \rangle$  où  $A$  est la donnée d'une racine  $o$  et d'une liste finie, éventuellement vide, d'arbres disjoints. On appelle cette liste une *forêt* (Alors, hein ? On ne me croyait pas !). On obtient donc un arbre en ajoutant une racine à une forêt (Ca tombe sous le sens).

KESBOUL

$A = \langle o, A_1, \dots, A_N \rangle$   
racine                          forêt

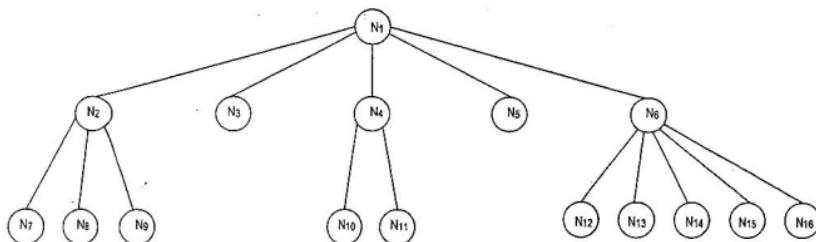


Figure 11. Représentation graphique d'un arbre général.

Le type abstrait d'un arbre général est le suivant :

**SORTÉ**  
Arbre, Forêt

noirs

**UTILISE**  
Noeud, Entier



**OPÉRATIONS**

cons	:	Noeud x Forêt → Arbre
racine	:	Arbre → Noeud
liste-arbre	:	Arbre → Forêt
forêt-vide	:	→ Forêt
ième	:	Forêt x Entier → Arbre
nb-arbres	:	Forêt → Entier
insérer	:	Forêt x Entier x Arbre → Forêt

**PRÉCONDITIONS**

insérer(F, i, A) est-défini-ssi  $1 \leq i \leq 1 + \text{nb-arbres}(F)$

**AXIOMES**

- racine(cons(o, F))=o
- liste-arbre(cons(o, F))=F
- nb-arbres(forêt-vide)=0
- $1 \leq i \leq 1 + \text{nb-arbres}(F) \Rightarrow \text{nb-arbres}(\text{insérer}(F, i, A)) = \text{nb-arbres}(F) + 1$
- $1 \leq i < k \Rightarrow \text{ième}(\text{insérer}(F, i, A), k) = \text{ième}(F, k)$
- $k = i \Rightarrow \text{ième}(\text{insérer}(F, i, A), k) = A$
- $i + 1 \leq k \leq 1 + \text{nb-arbres}(F) \Rightarrow \text{ième}(\text{insérer}(F, i, A), k) = \text{ième}(F, k - 1)$

**AVEC**

o	:	Noeud
F	:	Forêt
A	:	Arbre
i, k	:	Entier

*Remarque : La notion de « gauche-droite » propre aux arbres binaires disparaît. Cela dit, le vocabulaire employé pour les arbres généraux reste le même hormis pour tout ce qui fait appel à cette notion (de*

généralité). Par exemple, on ne parle plus de fils gauche-fils droit, mais de 1<sup>er</sup> fils, 2<sup>ème</sup> fils, etc. et de dernier fils.

Remarque 2 : Les mesures sur les arbres sont conservées (hauteur, longueur de cheminement, taille, etc.).

## Représentation des arbres généraux

Il existe plusieurs formes de représentations des arbres généraux. Les plus intuitives sont la représentation sous forme de listes chaînées et n-uplet. Une autre forme très usitée est la représentation sous forme d'arbres binaires.

### Représentation sous forme statique-dynamique

Dans ce cas de figure, l'ensemble des noeuds est représenté dans un tableau à une dimension (vecteur) surdimensionné pour pouvoir y ajouter des noeuds le cas échéant. Chaque élément est un enregistrement qui contient l'étiquette du noeud plus un pointeur sur sa liste de fils. La liste de fils est composée d'enregistrements contenant un pointeur sur le type d'élément constituant le tableau et un pointeur suivant. Dans ce cas, sa description algorithmique serait :

```
Constante
NbMaxNoeud = 200           /* Dimension du tableau */

Type
Element    = ....           /* Etiquette de chaque noeud */
T_Pfils   = ↑ T_Fils        /* Pointeur sur fils (liste) */
T_Pnoeud = ↑ T_Noeud       /* Pointeur sur Noeud (Tableau) */

T_Noeud   = Enregistrement
          Element Elt
          T_Pfils Premierfils
Fin enregistrement T_Noeud

T_Arbre   = NbMaxNoeud T_Noeud /* Vecteur */

T_Fils    = Enregistrement
          T_Pnoeud Noeufils
          T_Pfils FilsSuivant
Fin Enregistrement T_Fils

Variable
T_Arbre   A
```

Et sa représentation graphique en utilisant l'arbre de la *figure 11* serait :

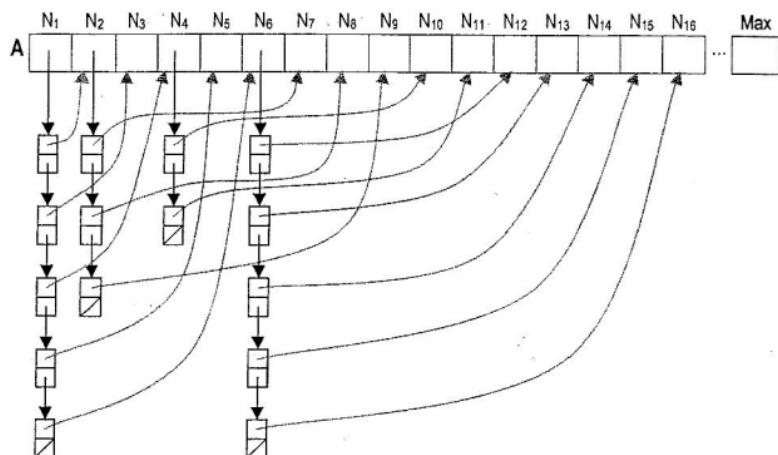


Figure 12. Représentation statique-Dynamique d'un arbre général

Remarque : Une possibilité est de transformer cette représentation en totale dynamique en remplaçant le tableau statique par une liste chaînée.

### Représentation sous forme de N-uplet

Si l'on connaît le nombre maximum de fils que peut avoir un nœud, il y a possibilité de représenter l'arbre sous forme de n-uplet. Cette représentation est une extension de la représentation dynamique des arbres binaires. Là, le nombre de fils n'est pas deux, mais celui que l'on s'est fixé. Pour l'arbre de la figure 11, en supposant qu'il y ait cinq fils maximum, la description algorithmique serait la suivante :

#### Constante

```
NbMaxFils = 5      /* Nombre maximum de fils */
```

#### Type

```
Element = ...          /* Etiquette de chaque noeud */
```

```
T_PArbre = ↑ T_Noeud    /* Pointeur sur Noeud (n-uplet) */
```

```
T_Tabfils = NbMaxFils T_PArbre  /* Tableau de pointeur sur fils */
```

```
T_Noeud = Enregistrement
```

```
Element Elt
```

```
T_Tabfils Fils
```

```
Fin enregistrement T_Noeud
```

#### Variable

```
T_PArbre A
```

Et sa représentation graphique en utilisant l'arbre de la figure 11 serait :

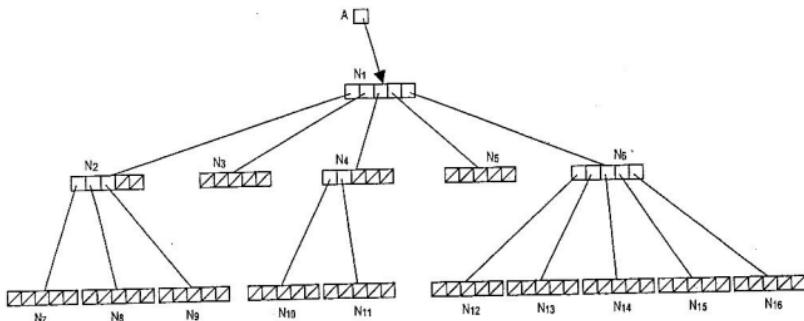


Figure 13. Représentation en n-uplet d'un arbre général

Remarque : La figure 13 ne correspond pas tout à fait à la déclaration précédente dans la mesure où elle ne représente que les pointeurs de chaque nœud et ne précise pas l'étiquette de ceux-ci.

#### Représentation sous forme d'arbre binaire

Le moyen de représenter les arbres généraux sous forme d'arbres binaires est d'utiliser le lien gauche comme **premier fils** et le lien droit comme **frère droit**. Cette représentation présente plusieurs avantages :

- Il y a un nœud par élément ni plus ni moins (*bijection fils ainé-frère droit*).
- Nous pouvons utiliser les algorithmes sur les arbres binaires (parcours, ajout de nœud, etc.) qui sont très simples à mettre en place.

Remarque : Nous utiliserons dans ce cas l'implémentation dynamique de l'arbre binaire.

La description algorithmique serait alors la suivante :

Type  
Element = ... /\* Définition du type des éléments \*/  
T\_Arbre = ↑ T\_Noeud /\* type pointeur sur T\_Noeud \*/  
T\_Noeud = Enregistrement  
    Element Elt  
    T\_Arbre Premierfils,frèredroit  
Fin Enregistrement T\_Noeud  
Variable  
T\_Arbre A

Ce qui pour l'arbre de la *figure 11*, la représentation graphique donnerait :

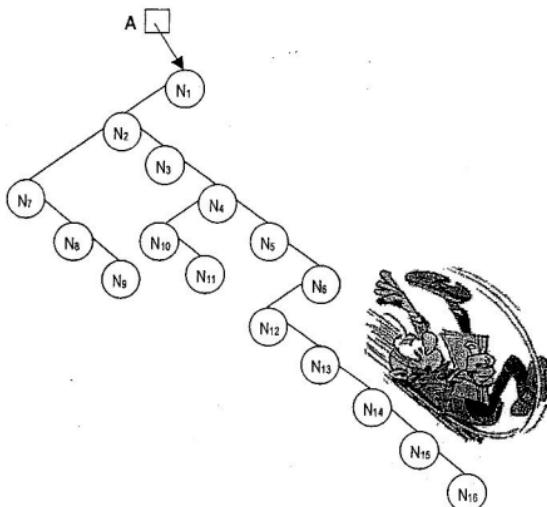


Figure 14. Représentation sous forme d'arbre binaire d'un arbre général

Travailler à l'aide de cet arbre est très simple. Par exemple pour connaître les fils d'un nœud, il suffit de parcourir le bord droit de son sous-arbre gauche.

*Remarque : Nous pouvons noter que l'arbre obtenu n'est pas du tout équilibré.*

*Remarque 2 : Le parcours préfixe et le parcours symétrique de cet arbre binaire donne respectivement le même ordre que le parcours préfixe et le parcours postfixe de l'arbre général que celui-ci représente.*

### Parcours d'un arbre général

Pour parcourir un arbre général, on peut réaliser une extension du parcours en profondeur main gauche des arbres binaires. La différence réside dans le fait qu'il n'y a pas de traitement infixe, mais un traitement intermédiaire à chaque passage d'un fils à un autre. En d'autres termes, un nœud est rencontré son nombre de fils plus une fois : la descente sur le premier fils, les nombres de fils moins un passage intermédiaire d'un fils à l'autre et enfin, la remontée depuis le dernier fils (C'est pas clair Junior ?). L'algorithme employé pour ce parcours est celui du parcours d'arbre binaire « légèrement modifié ». En effet, il faut placer les traitements à l'intérieur d'une boucle dont le nombre d'itérations repose sur le nombre de fils que possède l'arbre parcouru.

Pour l'exemple, si l'on définit un arbre général  $A = \langle o, A_1, A_2, A_3, A_4, A_5 \rangle$ , nous obtiendrons graphiquement le parcours représenté en figure 15, soit :

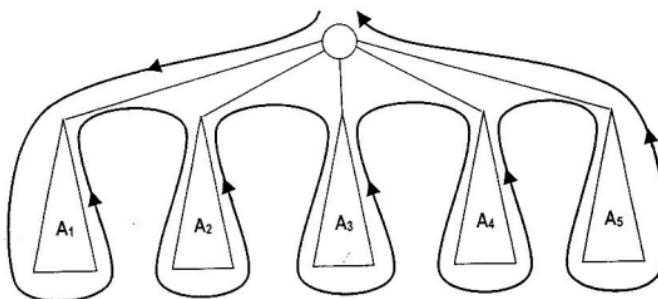


Figure 15. Parcours profond main gauche d'un arbre général

Sur ce parcours, on peut constater que le noeud racine est visité six fois ; une fois en descendant vers  $A_1$  (traitement préfixe), quatre fois en passant de  $A_1$  à  $A_2$ ,  $A_2$  à  $A_3$ ,  $A_3$  à  $A_4$  et  $A_4$  à  $A_5$  (quatre traitements intermédiaires) et enfin en remontant de  $A_5$  (traitement suffixe). Et là Junior, tu crois qu'ils ont compris ?

L'algorithme abstrait du parcours en profondeur d'un arbre général est alors :

```
Algorithme procedure parcours
Paramètres locaux :
    Arbre A
Variables
    Entier i, nbfil
Debut
    Nbfil <- nb-arbres(liste-arbre(A))
    Si feuille(A) alors
        TERMINAISON
        /* traitement de terminaison */
    Sinon
        TRTpréfixe
        Pour i <- 1 jusqu'à nbfil-1 Faire
            parcours(ième(liste-arbre(A),i))
            TRTintermédiaire
            /* traitement 2 (infixe) */
        Fin pour
        parcours(ième(liste-arbre(A),nbfil))
        TRTsuffixe
        /* traitement 3 (postfixe) */
    Fin si
Fin algorithme procedure parcours
```

*Remarque : Nous utilisons une extension présentée lors du parcours d'arbre binaire à savoir : feuille. Cette opération devrait bien sûr être définie.*

*Remarque 2 : Le nombre de traitements possibles sur un nœud ne se limite pas à trois, mais peut être réellement égal au nombre de fils plus un. En effet, le traitement intermédiaire peut varier selon la valeur de l'indice de boucle et par conséquent renvoyer à un traitement particulier à chaque fois.*

Par exemple, en utilisant l'arbre général de la figure 11, nous pourrions lister les nœuds de la manière suivante : (N1)(N2(N7)(N8)(N9))(N3)(N4(N10)(N11))(N5)(N6(N12)(N13)(N14)(N15)(N16))), en utilisant l'algorithme ci-dessous :

Algorithme procedure parclisp

Paramètres locaux :

Arbre A

Variables

Entier i, nbfilis

Début

Nbfilis  $\leftarrow$  nb-arbres(liste-arbre(A))

Si feuille(A) alors

Ecrire('(', contenu(racine(A)), ')') /\* traitement de terminaison \*/

Sinon

Ecrire('(', contenu(racine(A)))

Pour i  $\leftarrow$  1 jusqu'à nbfilis-1 Faire  
parcours(ième(liste-arbre(A), i))

Fin pour

parcours(ième(liste-arbre(A), nbfilis))

Ecrire(')')

/\* traitement 3 (postfixe) \*/

Fin si

Fin algorithme procedure parclisp



# Chapitre 4

## Les graphes

De nombreux problèmes courants, tels que la gestion de réseau routier, ferroviaire, aérien ou de communication peuvent être modélisés avec des graphes. Un graphe est un ensemble d'objets appelés **sommets** et de relations entre ces sommets.

Il existe deux sortes de graphes, les **graphes non orientés**, pour lesquels les relations, symétriques, sont appelées **arêtes** (figure 1) et les **graphes orientés** pour lesquels les relations, non symétriques, sont appelées **arcs** (figure 2).



Figure 1. Représentation graphique d'un graphe non orienté,  
Liaison aérienne entre différentes villes de France.

Dans cet exemple, les sommets sont des villes et les relations entre ces sommets symbolisent l'existence d'une liaison aérienne. On prend pour principe que s'il y a un vol aller, il y a un vol retour. Le graphe est dit non orienté et si deux sommets sont en relation, on dit alors qu'il existe une arête entre ces deux sommets. Par exemple, il existe une arête (Lyon, Strasbourg). Les questions auxquelles on peut répondre dans ce cas sont : Peut-on aller de Marseille à Brest ? Franchement, quelle idée, être à Marseille et vouloir aller à Brest. Enfin admettons... Alors dans ce cas, quel est le chemin qui nécessite le moins d'escales ?

Pour l'exemple suivant (figure 2), nous avons la représentation d'un organigramme (un diagramme des flux, ça fait toujours bien dans les cocktails) qui détermine la valeur de la factorielle de n. Dans ce cas, les sommets sont des blocs d'instructions, et les relations ne symbolisent pas seulement le passage d'un bloc à l'autre. En effet, elles en définissent aussi le sens. Par exemple, on peut passer du sommet S<sub>2</sub> au sommet S<sub>3</sub>, mais pas l'inverse. On dit dans ce cas que le graphe est orienté et l'existence d'une relation entre deux sommets est appelée arc. Par exemple, il existe un arc (S<sub>5</sub>, S<sub>4</sub>).

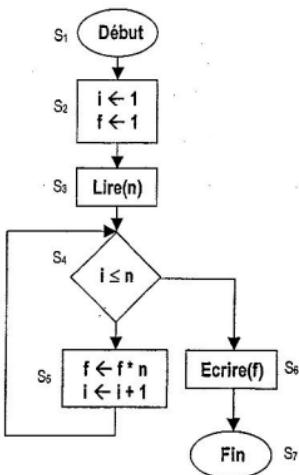


Figure 2. Représentation graphique d'un graphe orienté,  
Organigramme de calcul de la factorielle de  $n$ .

Bien entendu, il est possible de modéliser beaucoup d'autres choses à l'aide des graphes : Réseau informatique, recette de cuisine, etc.

## Définitions

- Un graphe orienté  $G$  est un couple  $\langle S, A \rangle$ , où  $S$  est un ensemble fini de sommets et où  $A$  est un ensemble fini de paires ordonnées de sommets appelées arcs.
- Un graphe non orienté  $G$  est un couple  $\langle S, A \rangle$ , où  $S$  est un ensemble fini de sommets et où  $A$  est un ensemble fini de paires de sommets appelées arêtes.
- Un graphe non orienté (orienté) **valué**  $G$  est un triplet  $\langle S, A, C \rangle$  où  $S$  est un ensemble fini de sommets, où  $A$  est un ensemble fini de paires (ordonnées) de sommets appelées arêtes (arcs) et où  $C$  est une fonction de  $A$  sur  $R$  appelée **fonction de coût**.

*Remarque : Dans ce dernier cas, les relations sont porteuses d'une valeur. Sur l'exemple de la figure 1, ce coût pourrait être le prix du billet, la durée du vol, etc.*

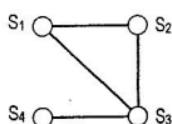
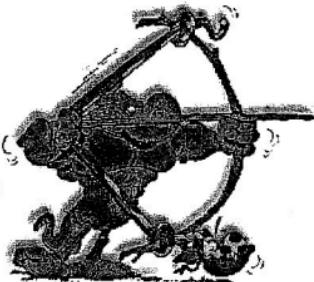
*Remarque 2 : Un graphe orienté peut parfois être traité de manière non orientée quand la nature du problème l'impose. Le métro Parisien, par exemple, est de nature orienté (du à quelques stations) et pourtant, pour une recherche d'itinéraire, nous avons tout intérêt à le considérer comme non orienté.*

*Remarque 3 : Certains graphes admettent plusieurs relations entre deux mêmes sommets. On parle dans ce cas de graphes à arcs ou arêtes multiples, voire de multigraphes.*

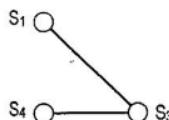
## Terminologie

Les graphes n'échappant pas à la règle, voici les différents termes qui les caractérisent :

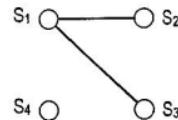
- On note l'arc  $(x,y)$ ,  $x \rightarrow y$ . On dit que :
  - $x$  est l'extrémité initiale de l'arc
  - $y$  est l'extrémité terminale de l'arc
  - $y$  est le successeur de  $x$
  - $x$  est le prédécesseur de  $y$
- On note l'arête  $\{x,y\}$ ,  $x — y$ . On dit que :
  - $x$  et  $y$  sont les deux extrémités de l'arête
  - $x$  est le successeur de  $y$
  - $y$  est le successeur de  $x$
  - $x$  est le prédécesseur de  $y$
  - $y$  est le prédécesseur de  $x$
- Soit  $G = \langle S, A \rangle$  un graphe. Le **sous-graphe** de  $G$  engendré par  $S' \subseteq S$  est le graphe  $G' = \langle S', A' \rangle$  (figure 3(b)) dont les sommets sont les éléments de  $S'$  et dont les arcs (arêtes) sont ceux de  $G$  ayant leurs deux extrémités dans  $S'$ .
- Soit  $G = \langle S, A \rangle$  un graphe. Le **graphe partiel** de  $G$  engendré par  $A' \subseteq A$  est le graphe  $G' = \langle S, A' \rangle$  (figure 3(c)) dont les sommets sont les éléments de  $S$  et dont les arcs (arêtes) sont ceux de  $A'$ .



(a)  $G = \langle S, A \rangle$  Graphe non orienté  
 $S = \{S_1, S_2, S_3, S_4\}$   
 $A = \{\{S_1, S_2\}, \{S_1, S_3\}, \{S_1, S_4\}, \{S_2, S_3\}, \{S_2, S_4\}, \{S_3, S_4\}\}$



(b)  $G' = \langle S', A' \rangle$  Sous-graphe  
 $S' = \{S_1, S_3, S_4\}$   
 $A' = \{\{S_1, S_3\}, \{S_3, S_4\}\}$



(c)  $G' = \langle S, A' \rangle$  Graphe partiel  
 $S = \{S_1, S_2, S_3, S_4\}$   
 $A' = \{\{S_1, S_2\}, \{S_1, S_3\}\}$

Figure 3. Exemple de sous-graphe et de graphe partiel

- Deux arcs (arêtes) d'un graphe orienté (non orienté) sont dits **adjacents** s'ils ont au moins une extrémité commune.
- Deux sommets d'un graphe non orienté sont dits **adjacents** s'il existe une arête les joignant.
- Dans un graphe orienté, le sommet  $x$  est dit **adjacent à  $y$**  s'il existe un arc  $x \rightarrow y$ .
- Un graphe orienté (non orienté) est dit **complet** si, pour tout couple de sommets  $(x,y)$ , il existe un arc  $x \rightarrow y$  (une arête  $x — y$ ).
- Dans un graphe orienté, si un sommet  $x$  est l'extrémité initiale d'un arc  $u=x \rightarrow y$ , on dit que l'arc  $u$  est **incident à  $x$  vers l'extérieur**.
- On appelle le nombre d'arcs ayant  $x$  pour extrémité initiale le **demi-degré extérieur** de  $x$  et on le note  $d^+(x)$ .
- Dans un graphe orienté, si un sommet  $x$  est l'extrémité terminale d'un arc  $u=y \rightarrow x$ , on dit que l'arc  $u$  est **incident à  $x$  vers l'intérieur**.
- On appelle le nombre d'arcs ayant  $x$  pour extrémité terminale le **demi-degré intérieur** de  $x$  et on le note  $d^-(x)$ .
- Dans un graphe non orienté, on appelle **degré** de  $x$  le nombre d'arêtes pour lesquelles  $x$  est une extrémité.
- Dans un graphe orienté, le degré est égal à la somme des demi-degrés (figure 4).

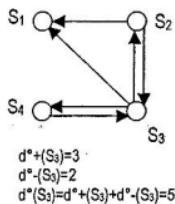


Figure 4. Degré et demi-degrés d'un graphe orienté

- Dans un graphe orienté (non orienté)  $G$ , on appelle **chemin (chaîne) de longueur  $\lambda$**  une suite de  $(\lambda+1)$  sommets  $(S_0, S_1, \dots, S_\lambda)$  tels que pour tout  $i$  tel que  $0 \leq i \leq \lambda-1$ ,  $S_i \rightarrow S_{i+1}$  est un arc (arête) de  $G$ .
- On considère que tout chemin d'un sommet vers lui-même est de longueur 0.
- Un chemin (une chaîne) est dit **élémentaire** s'il ne contient pas plusieurs fois le même sommet.
- Dans un graphe orienté (non orienté), un chemin (une chaîne)  $(S_0, S_1, \dots, S_\lambda)$  dont les  $\lambda$  arcs (arêtes) sont tous distincts deux à deux et tel que les deux sommets aux extrémités coïncident est appelé **circuit (cycle)**.
- Un graphe orienté est dit **fortement connexe** si pour toute paire ordonnée de sommets distincts  $(x,y)$ , il existe un chemin de  $x$  vers  $y$  et un chemin de  $y$  vers  $x$ .
- Un graphe non orienté est dit **connexe** si pour toute paire de sommets distincts  $(x,y)$ , il existe un chaîne reliant  $x$  et  $y$ .

**SORTÉ**  
Graphe

**UTILISE**  
Sommet, Entier, Booléen

**OPÉRATIONS**

graphe_vide	:	→ Graphe
ajouter-le-sommet _ à _	:	Sommet x Graphe → Graphe
ajouter-l'arc <_,> à _	:	Sommet x Sommet x Graphe → Graphe
_ est -un-sommet-de _	:	Sommet x Graphe → Booléen
<_,> est -un-arc-de _	:	Sommet x Sommet x Graphe → Booléen
d <sup>e</sup> + de _ dans _	:	Sommet x Graphe → Entier
_ ème-succ-de _ dans _	:	Entier x Sommet x Graphe → Sommet
d <sup>e</sup> - de _ dans _	:	Sommet x Graphe → Entier
_ ème-pred-de _ dans _	:	Entier x Sommet x Graphe → Sommet
retirer-le-sommet _ de _	:	Sommet x Graphe → Graphe
retirer-l'arc <_,> de _	:	Sommet x Sommet → Graphe
// Fonctions complémentaires		
premsucc	:	Sommet x Graphe → Sommet
succsuivant	:	Sommet x Sommet x Graphe → Sommet
coût	:	Sommet x Sommet x Graphe → Réel
ajouter-l'arc <_,> de coût _ à _	:	Sommet x Sommet x Réel x
Graphe → Graphe		
nb-sommets	:	Graphe → Entier
nb-arcs	:	Graphe → Entier

## Types abstraits des graphes non orientés

De même que pour les graphes orientés, nous n'allons pas fournir la définition axiomatique complète, mais seulement la déclaration des fonctions qui caractérisent ce graphe, auxquelles nous ajouterons quelques fonctions de manipulation complémentaires, pour pouvoir les utiliser dans les algorithmes abstraits qui suivront ; nous obtenons alors :

**SORTÉ**  
Graphe

**UTILISE**  
Sommet, Entier, Booléen

**OPÉRATIONS**

graphe_vide	:	→ Graphe
ajouter-le-sommet _ à _	:	Sommet x Graphe → Graphe
ajouter-l'arête <_,> à _	:	Sommet x Sommet x Graphe → Graphe

```

_ est -un-sommet-de _ : Sommet x Graphe → Booléen
<_,_> est -une-arête-de _ : Sommet x Sommet x Graphe → Booléen

d° de _ dans _ : Sommet x Graphe → Entier
_ ème-succ-de _ dans _ : Entier x Sommet x Graphe → Sommet

retirer-le-sommet _ de _ : Sommet x Graphe → Graphe
retirer-l'arête <_,_> de _ : Sommet x Sommet → Graphe

// Fonctions complémentaires
premsucc : Sommet x Graphe → Sommet
succsuivant : Sommet x Sommet x Graphe → Sommet

coût : Sommet x Sommet x Graphe → Réel

ajouter-l'arête <_,_> de coût _ à _ : Sommet x Sommet x Réel ×
Graphe → Graphe

nb-sommets : Graphe → Entier
nb-arêtes : Graphe → Entier

```

## Représentation des graphes

Il existe plusieurs façons de représenter les graphes. Le choix de la méthode de représentation dépend de plusieurs critères dont le côté évolutif ou statique du graphe. Nous allons donc envisager trois formes de représentation :

- Les matrices (S complètement statique)
- Les listes statique-dynamique (S complètement statique)
- Les listes dynamiques (S évolutif).

### Représentation sous forme de matrice

Si l'ensemble S n'évolue jamais, il est possible d'utiliser des matrices carrées  $n \times n$  de booléens appelées **Matrices d'adjacences** ( $n$  représentant le nombre de sommets du graphe). Nous avons alors la déclaration suivante :

```

Constante
Nbsommet = 4      /* Valeur donnée en exemple */

Type
T-Graphe = Nbsommet x Nbsommet Booléen

Variable
T-Graphe G

```

L'élément  $G[i,j]$  est vrai s'il existe une relation entre le sommet  $i$  et le sommet  $j$ .  
 Sur la Figure 5, nous pouvons voir un exemple de graphe orienté, et sa représentation sous forme de matrice de booléen.

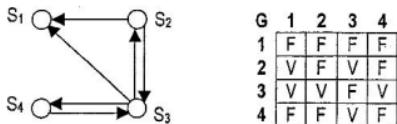


Figure 5. Représentation d'un graphe orienté sous forme de matrice de booléens.

Bien entendu, si le graphe est valué il n'est plus possible de se contenter d'une matrice de booléen. Dans ce cas, la matrice devient : soit une matrice d'élément de types simples (entier, réel, etc.) si l'on n'utilise pas l'intégralité du domaine de définition de ces types et qu'une valeur peut être utilisée pour signaler l'absence d'arc, soit une matrice d'éléments définis par l'utilisateur contenant les valeurs plus un booléen. Ce qui donnerait pour un graphe valué par des entiers positifs la déclaration suivante :

#### Constante

Nbsommet = 4 /\* Valeur donnée en exemple \*/

#### Type

T-Graphe = Nbsommet x Nbsommet Entier

#### Variable

T-Graphe G

Il existe une relation entre deux sommets  $i$  et  $j$  si l'élément  $G[i,j]$  est d'une valeur différente de -1. Sur la Figure 6, nous pouvons voir un exemple de graphe orienté valué, et sa représentation sous forme de matrice d'entiers.

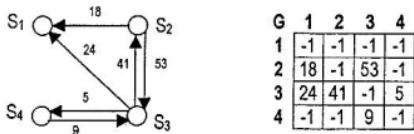


Figure 6. Représentation d'un graphe orienté valué sous forme de matrice d'entiers.

*Remarque : L'ensemble S dans certains cas particuliers pourrait légèrement évoluer. Pour cela, il est nécessaire de surdimensionner la matrice au départ.*

La représentation matricielle est très pratique pour tester l'existence d'un arc. Il est facile d'ajouter ou de supprimer un arc. Il est tout aussi facile de parcourir tous les successeurs ou prédécesseurs d'un sommet et donc de déterminer son degré ou ses demi-degrés.

Malheureusement, un parcours complet exige un temps d'ordre  $n^2$  et le stockage du graphe un espace mémoire du même ordre. Pour régler ces différents problèmes, nous utiliserons une autre forme de représentation appelée *Liste d'adjacence*.

## Représentation sous forme de Liste

Cette représentation peut revêtir plusieurs formes (Ca lui fait un Look ?!). En fait, cela dépend de S. Si celui-ci est statique, nous aurons alors un tableau représentant les sommets, auquel sera associé pour chaque sommet une liste dynamique représentant les successeurs. Dans le cas où S est évolutif, le tableau sera remplacé par une liste dynamique.

### Cas où S est statique

Dans ce cas, la déclaration de la structure utilisée serait la suivante :

#### Constante

Nbsommet = 4

#### Type

T\_Psommet =  $\uparrow T_{\text{Sommet}}$  /\* type pointeur sur  $T_{\text{Sommet}}$  \*/

$T_{\text{Sommet}} = \text{Enregistrement}$

Entier NumSommet /\* Place du coût si le graphe est valué \*/

$T_{\text{Psommet}} = \text{Lien}$

Fin Enregistrement  $T_{\text{Sommet}}$

$T_{\text{Graphe}} = \text{Nbsommet } T_{\text{Psommet}}$

#### Variable

$T_{\text{Graphe}} G$

Nous avons un graphe représenté par un vecteur de pointeur sur Sommet. Chaque sommet contient le numéro de sommet sous forme d'entier et un lien sur le sommet adjacent suivant (voir Figure 7).

Attention : la liste associée à un sommet contient uniquement ses successeurs. Cette liste ne représente en aucun cas un chemin à partir du sommet.

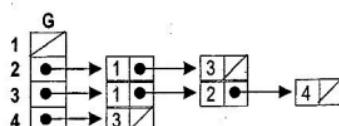
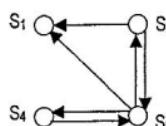


Figure 7. Représentation d'un graphe orienté sous forme de liste d'adjacence.

Remarque : Pour ajouter le coût de la relation, il suffit d'ajouter un champ à l'enregistrement  $T_{\text{Sommet}}$ .

Plus d'accès

cessou

$T \in G[\tau]$

si que  $T \neq \text{Null}$  faire

si  $T^.NumSommet = y$  alors

fin boucle

fin  $T \leftarrow T^.succSucc$

PAGE 48

Accès

$T \rightarrow A[1]$

Si  $n$  est le nombre de sommets et  $p$  le nombre d'arcs, l'espace mémoire utilisé pour un graphe est d'ordre  $n+p$ . Pour un graphe non orienté, il serait de  $n+2p$  (De là à penser, qu'une une arête vaut deux arcs, il n'y a qu'un pas que je ne franchirai pas). Ce qui est le cas aussi d'un graphe orienté pour lequel il serait nécessaire d'effectuer un traitement sur les prédécesseurs. En effet, cette représentation ne prend en compte que les successeurs, il suffit alors de posséder une autre liste chaînée pour les prédécesseurs. D'autre part, si le graphe présente un ensemble de sommets évolutif, la représentation de ce dernier doit aussi se faire sous forme de liste chaînée. Ce qui donnerait l'exemple de la figure 8. La déclaration quant à elle serait la suivante :

```
Type Élement = ...
T_Psomadj = ↑ T_Somadj           /* type pointeur sur Sommet adjacent */
T_Psommet = ↑ T_Sommet          /* type pointeur sur Sommet */

T_Somadj = Enregistrement
    T_Psommet   NumSommet      /* Place du coût si le graphe est valué */
    T_Psomadj   Lien           /* Prédécesseur ou Successeur */
Fin Enregistrement T_Somadj

T_sommet = Enregistrement
    Entier       NumSommet
    T_Psommet   Suiv, Prec     /* Sommet suivant et précédent */
    T_Psomadj   Pred, Succ     /* Prédécesseur ou Successeur */
Fin Enregistrement T_sommet
T_Graphe = T_Psommet

Variable
T_Graphe G
```

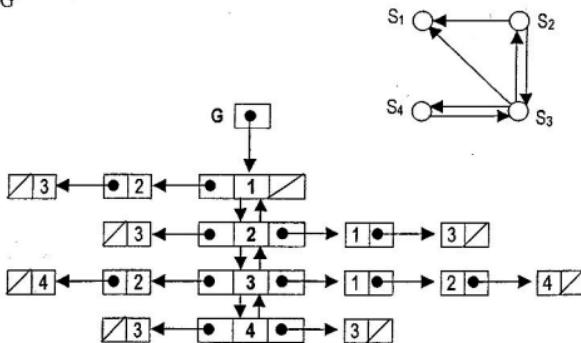


Figure 8. Représentation d'un graphe orienté (S évolutif) sous forme de listes d'adjacence (prédécesseurs et successeurs).

*Remarque : Par souci de lisibilité, dans les listes de prédecesseurs et de successeurs, les numéros de sommets remplacent les flèches symbolisant les pointeurs associés.*

Avec une telle représentation, il devient facile de trouver un sommet adjacent, de calculer le demi-degré intérieur et extérieur d'un sommet, ainsi que de créer ou détruire des relations et des sommets. En revanche, ces facilités ne modifient pas le temps de recherche qui pour l'existence d'un arc, par exemple, peut être d'ordre n.

## Parcours de Graphes

L'utilisation des graphes nécessite souvent l'étude exhaustive des sommets et arcs ou arêtes de celui-ci. Nous allons donc décrire les parcours de graphes. Ceux-ci se présentent sous deux formes qui sont, comme pour une arborescence, le parcours en profondeur et le parcours en largeur.

### Parcours en profondeur

Ce parcours consiste à suivre, à partir d'un sommet donné, un chemin le plus loin possible. Puis à revenir en arrière pour explorer les chemins ignorés. Intrinsèquement, ce parcours est récursif. Considérons un graphe orienté dont les sommets sont non marqués. Nous choisissons un sommet S, nous le marquons. Nous prenons son premier successeur (s'il existe, évidemment...) puis nous recommençons à partir de celui-ci le même traitement que pour le sommet S. En fin de chemin (plus de successeur, damned !), nous revenons au sommet précédent et nous étudions son deuxième successeur, et ainsi de suite...

Malheureusement si le graphe n'est pas fortement connexe, il est probable que certains sommets n'aient pas été visités (marqués). Pour cela, nous plaçons une procédure itérative externe qui vérifie que les sommets sont marqués. Si elle en trouve un qui ne l'est pas, elle rappelle la procédure de parcours à partir de ce dernier.

L'algorithme de parcours profondeur (version récursive) donne :

Algorithme principal

Constante

Nbsommet = 4

Type

T\_VectNbool = Nbsommet Booléen

Variables

T\_VectNbool Marque

T\_Graphe G

Entier i

Debut

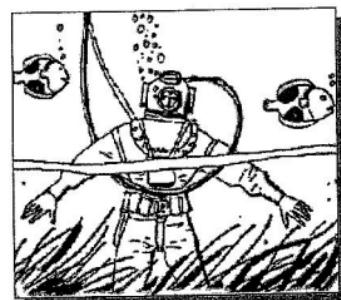
Pour i ← 1 jusqu'à Nbsommet Faire

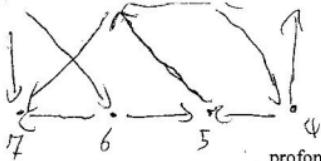
    Marque[i] ← faux

Fin pour

Pour i ← 1 jusqu'à Nbsommet Faire

    Si Non(Marque[i]) alors





1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15  
suf : 3, 7, 6, 8, 1, 3, 4, 2

```

    profondeur(i,G,Marque)
Fin si
Fin pour
Fin algorithme principal

```

#### Algorithme procedure profondeur

Paramètres locaux

Entier s  
T\_Graphe G

Paramètres globaux

T\_VectNbool Marque

Variables

Entier i,t

/\* s est un sommet \*/

Debut

Marque[s] ← Vrai

/\* 1<sup>re</sup> rencontre de s \*/

Pour i ← 1 jusqu'à d°+ de s dans G Faire

t ← i ème-succ- de s dans G

/\* rencontre à aller de l'arc(s,t) \*/

Si Non(Marque[t]) alors

profondeur(t,G,Marque)

/\* appel récursif sur suivant \*/

Fin si

/\* rencontre au retour de l'arc(s,t) \*/

Fin pour

/\* dernière rencontre de s \*/

Fin algorithme procedure profondeur

En supposant que l'ordre d'utilisation des sommets est croissant, et que le sommet S<sub>1</sub> est le premier choisi, Si l'on utilise le graphe orienté de la Figure 9 et que l'on insère un ordre d'écriture de s lors de la première rencontre, nous obtenons l'affichage suivant :

S<sub>1</sub>, S<sub>2</sub>, S<sub>6</sub>, S<sub>5</sub>, S<sub>8</sub>, S<sub>7</sub>, S<sub>3</sub>, S<sub>4</sub>

De la même manière, si l'ordre d'écriture était inséré lors de la dernière rencontre, nous obtenons :

S<sub>2</sub>, S<sub>5</sub>, S<sub>7</sub>, S<sub>8</sub>, S<sub>6</sub>, S<sub>1</sub>, S<sub>4</sub>, S<sub>3</sub>

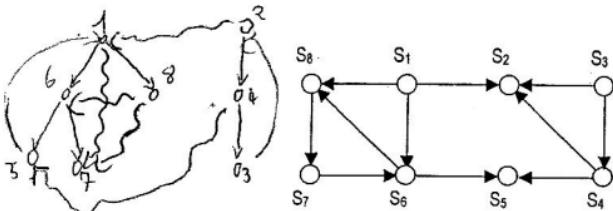


Figure 9.

- arc en arrière
- arc courrant
- arc en avant
- arc croisé

## Forêt couvrante associée au parcours en profondeur d'un graphe

Dans le parcours en profondeur d'un graphe orienté, on distingue plusieurs sortes d'arcs : les arcs couvrants, les arcs en arrière, les arcs en avant et les arcs croisés. Ils correspondent aux définitions suivantes :

- Les arcs  $x \rightarrow y$  tels que  $\text{profondeur}(x) < \text{profondeur}(y)$  sont appelés **arcs couvrants**. Ils constituent une **forêt couvrante de recherche en profondeur**. Par exemple  $S_1 \rightarrow S_6$  dans l'exemple donné en *Figure 9*.
- Les arcs dont l'extrémité terminale est un descendant de l'extrémité initiale dans la forêt sont appelés **arcs en arrière**. Par exemple  $S_7 \rightarrow S_6$  dans l'exemple donné en *Figure 9*.
- Les arcs dont l'extrémité terminale est un descendant dans la forêt de l'extrémité initiale sont appelés **arcs en avant**. Par exemple  $S_1 \rightarrow S_8$  dans l'exemple donné en *Figure 9*.
- Les arcs pour lesquels il n'existe pas de chemin entre leurs extrémités, dans la forêt, sont appelés **arcs croisés**. Par exemple  $S_4 \rightarrow S_5$  dans l'exemple donné en *Figure 9*.

Nous pouvons voir une représentation de ces différents arcs en *Figure 10*.

*Remarque : Si par convention, on dessine les arcs de la forêt au fur et à mesure de leur rencontre et les différentes arborescences de la gauche vers la droite, les arcs croisés sont nécessairement dirigés de la droite vers la gauche.*

*Remarque 2 : Certaines propriétés des forêts couvrantes associées aux parcours de graphe étant définies géographiquement, il est impératif de respecter la convention de représentation graphique évoquée dans la précédente remarque.*

Si lors du parcours en profondeur du graphe, on affecte à chaque sommet une valeur de compteur (compt) qui correspond à son ordre de rencontre dans le parcours, on obtient les propriétés suivantes :

- Un sommet  $y$  est l'un des  $k$  descendants d'un sommet  $x$  dans la forêt si et seulement si  $\text{compt}(x) < \text{compt}(y) \leq \text{compt}(x)+k$
- Si l'arc  $x \rightarrow y$  est un arc couvrant ou un arc en avant alors  $\text{compt}(x) < \text{compt}(y)$
- Si l'arc  $x \rightarrow y$  est un arc en arrière ou un arc croisé alors  $\text{compt}(x) > \text{compt}(y)$

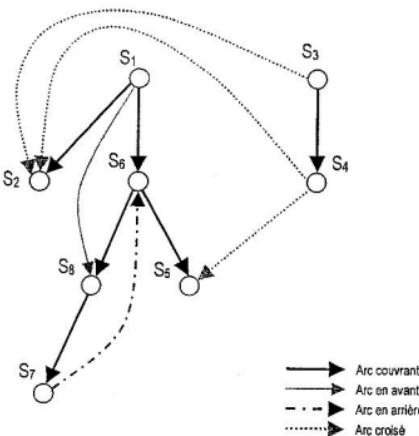


Figure 10. Forêt couvrante associée au parcours en profondeur du graphe de la Figure 9.

### Parcours en profondeur d'un graphe non orienté

L'algorithme utilisé est le même que pour un graphe orienté. La complexité reste la même dans la mesure où l'on considérera pour un graphe ayant  $p$  arêtes qu'il possède  $2p$  arcs (vous vous rappelez ?). Par contre, le sens du parcours oriente les arêtes et l'on parle encore d'arcs pour la forêt couvrante associée. Mais l'on ne distingue plus que deux sorte d'arcs :

- Les arcs couvrants : Ceux tels que  $\text{profondeur}(x)$  appelle directement  $\text{profondeur}(y)$ .
- Les arcs en arrières : Ceux tels que  $x$  est un descendant de  $y$  dans la forêt.

*Remarque : Si l'arête  $x-y$  est devenue un arc couvrant, on ne doit pas la considérer quand on construit les arcs en arrière.*

*Remarque 2 : Il n'y a pas d'arcs en avant puisque les arêtes sont non orientées.*

*Remarque 3 : Il n'y a pas d'arcs croisés puisque  $x$  et  $y$ , deux sommets, ne peuvent pas être adjacents sans être descendants l'un de l'autre (relation symétrique).*

Pour exemple, transformons le graphe orienté de la Figure 9 en graphe non orienté (Figure 11), modifions le légèrement et observons la forêt couvrante obtenue en Figure 11. Comme le graphe est connexe, il n'y a qu'une seule arborescence. Comme précédemment, les sommets sont traités en ordre numérique croissant et nous commençons par le sommet  $S_1$ .

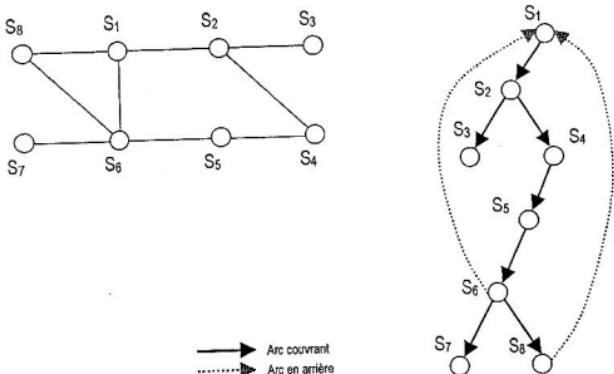


Figure 11. Forêt couvrante associée au parcours en profondeur d'un graphe non orienté.

### Parcours en largeur

Ce parcours consiste à suivre, à partir d'un sommet donné, tous les successeurs de celui-ci, puis à passer à tous les successeurs du 1<sup>er</sup> successeur, puis à tous les successeurs du 2<sup>ème</sup>, etc. Le parcours se fait, en fait, par distance, c'est à dire que l'on parcourt d'abord tous les sommets se trouvant à une distance de 1 du sommet de départ, puis tous ceux qui se trouvent à une distance de 2 et ainsi de suite... Intrinsèquement, ce parcours est itératif.

De la même façon que pour le parcours en profondeur, nous allons marquer les sommets parcourus, et si le graphe n'est pas fortement connexe, il est probable que certains sommets n'aient pas été visités (marqués). Pour cela, nous plaçons une procédure itérative externe qui vérifie que les sommets sont marqués. Si elle en trouve un qui ne l'est pas, elle rappelle la procédure de parcours à partir de ce dernier.

L'algorithme de parcours largeur (version itérative) utilise une File et donne :

```

Algorithme principal
Constante
    Nbsommet = 4
Type
    T_VectNbool = Nbsommet Booléen
Variables
    T_VectNbool    Marque
    T_Graphe        G
    Entier          i
...
Début
    Pour i ← 1 jusqu'à Nbsommet Faire

```



```

        Marque[i] ← faux
    Fin pour
    Pour i ← 1 jusqu'à Nbsommet Faire
        Si Non(Marque[i]) alors
            largeur(i,G,Marque)
        Fin si
    Fin pour
Fin algorithme principal

Algorithme procedure largeur
Paramètres locaux :
    Entier      s           /* s est un sommet */
    T_Graphe   G
Paramètres globaux :
    T_VectNbool   Marque
Variables
    Entier      i,t,u       /* t et u sont des sommets */
    File       F
Début
    Marque[s] ← Vrai
    F ← File_vide
    F ← Ajouter(F,s)
    Tant que non(Est_vide(F)) Faire
        t ← Premier(F)
        F ← Retirer(F)
        Pour i ← 1 jusqu'à d°+ de t dans G Faire
            u ← i ème-succ-de t dans G
            Si Non(Marque[u]) alors
                Marque[u] ← Vrai
                F ← Ajouter(F,u)
            Fin si
        Fin pour
    Fin tant que
Fin algorithme procedure largeur

```

Pour cet algorithme, le fait qu'il soit itératif rend extrêmement simple l'inclusion du programme principal dans la procédure largeur. Malgré cela, il est préférable de laisser les choses en l'état. En effet, vous pouvez désirer effectuer le parcours uniquement à partir d'un sommet sans vouloir parcourir l'intégralité du graphe, ce que l'inclusion ne permettrait plus.

En supposant que l'ordre d'utilisation des sommets soit croissant, et que le sommet  $S_1$  soit le premier choisi, Si l'on utilise le graphe orienté de la Figure 9 et que l'on insère un ordre d'écriture de  $s$  lors de la première rencontre, nous aurions l'affichage suivant :

$S_1, S_2, S_6, S_8, S_5, S_7, S_3, S_4$

Remarque : La complexité est la même que pour le parcours en profondeur.

# Chapitre 6

## Algorithmes de recherche – Arbres binaires de recherche

Dans le chapitre précédent, nous avons vu que la recherche la plus performante et la plus facile à mettre en place est la recherche dichotomique. Or, celle-ci ne peut être implantée que sur des structures contiguës (tableaux). De plus ce genre de structures n'est pas très adapté à des collections de données qui évoluent. C'est pourquoi, il nous faut fournir une structure chaînée permettant d'avoir les mêmes temps de recherche. Dans ce cas, les structures les mieux adaptées sont les structures arborescentes. Nous allons donc envisager, pour commencer, les arbres binaires dits de recherche.

### Définition

Un **arbre binaire de recherche** est un arbre binaire étiqueté tel que pour tout noeud  $v$  de l'arbre :

- Les éléments de tous les noeuds du sous-arbre gauche de  $v$  sont inférieurs ou égaux à l'élément contenu dans  $v$ .
- Les éléments de tous les noeuds du sous-arbre droit de  $v$  sont strictement supérieurs à l'élément contenu dans  $v$ .

*Remarque : Le parcours infixé d'un ABR (Arbre binaire de recherche) fournit la liste des éléments triés en ordre croissant.*

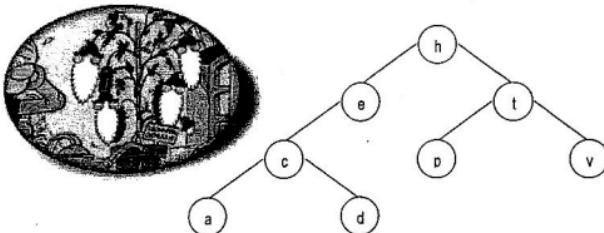


Figure 1. Exemple d'Arbre binaire de recherche.

Sur la Figure 1, nous pouvons voir un des arbres binaires de recherche associé à l'ensemble de données  $E = \{a, c, d, e, h, p, t, v\}$  muni de l'ordre alphabétique. Un des arbres binaires, parce qu'en effet, il existe plusieurs ABR possibles pour une même collection de données. En fait, on le verra plus tard (Si j'y pense), cela dépend de l'ordre d'arrivée de ces données.

### Recherche d'un élément

La recherche d'un élément s'inspire de la dichotomie, sauf qu'il n'est nul besoin de découper l'ensemble en deux, puisque l'on part vers un des deux sous-arbres.

### Principe algorithmique

On compare l'élément x recherché au contenu de la racine r de l'arbre ; dans ce cas trois possibilités:

- $x = \text{contenu}(r)$  alors on a trouvé x et la recherche s'arrête
- $x < \text{contenu}(r)$  alors on poursuit la recherche dans le sous-arbre gauche.
- $x > \text{contenu}(r)$  alors on poursuit la recherche dans le sous-arbre droit.

Si le sous-arbre sur lequel doit se poursuivre la recherche est vide, il y a échec, l'élément x n'existe pas dans cette collection de données et la recherche est négative.

### Spécification formelle

Un arbre binaire étiqueté B dont la racine contient l'élément r est noté  $B = \langle r, G, D \rangle$  avec G et D sous-arbres gauche et droit. Dans ce cas, la spécification formelle (extension aux arbres binaires) s'écrit :

**OPERATION**  
 rechercher : Element x ArbreBinaire  $\rightarrow$  Booléen

**AXIOMES**

```

  rechercher(x, Arbre-vide)=Faux
  x=r => rechercher(x,<r,G,D>)=Vrai
  x<r => rechercher(x,<r,G,D>)=rechercher(x,G)
  x>r => rechercher(x,<r,G,D>)=rechercher(x,D)
  
```

**AVEC**

```

  G, D : ArbreBinaire
  x, r : Element
  
```

### Algorithme (version récursive)

Comme on peut le voir, cet algorithme ressemble étrangement (voire curieusement) à celui de la recherche dichotomique.

Algorithme Fonction rechercher : Booléen  
 Paramètres locaux

```

    Element x
    ArbreBinaire B
  
```

Debut

```

    Si B=Arbre-vide alors
      Retourne(Faux)                                /* recherche négative */
    Sinon
      Si Contenu(racine(B))=x alors
        Retourne(Vrai)                            /* recherche positive */
      Sinon
        Si x<contenu(racine(B)) alors
          
```

```

        Retourne(rechercher(x,g(B)) /* poursuite sur sous-arbre gauche */
Sinon
        Retourne(rechercher(x,d(B)) /* poursuite sur sous-arbre droit */
Fin si
Fin si
Fin si
Fin algorithme fonction rechercher

```

(version itérative)

La traduction reste aussi simple et la performance est accrue (sans selle, quoi.).

```

Algorithme Fonction rechercher : Booléen
Paramètres locaux
    Element   x
    ArbreBinaire   B
Debut
    Tant que B<>Arbre-vide Faire
        Si Contenu(racine(B))=x alors
            Retourne(Vrai)           /* recherche positive */
        Sinon
            Si x<contenu(racine(B)) alors
                B←g(B)             /* poursuite sur sous-arbre gauche */
            Sinon
                B←d(B)             /* poursuite sur sous-arbre droit */
            Fin si
        Fin si
    Fin tant que
    Retourne(Faux)           /* recherche négative */
Fin algorithme fonction rechercher

```



## Ajout d'un élément

Il existe deux façons d'ajouter un élément dans un ABR ; soit aux feuilles, soit à la racine.

### Ajout aux feuilles

#### Principe algorithmique

Pour ajouter un élément  $x$  aux feuilles d'un arbre binaire, on le compare au contenu de la racine  $r$  ; dans ce cas deux possibilités (donc une alternative) :

- $x \leq \text{contenu}(r)$  alors on poursuit dans le sous-arbre gauche.
- $X > \text{contenu}(r)$  alors on poursuit dans le sous-arbre droit.

Si le sous-arbre sur lequel doit se poursuivre la recherche est vide, c'est là qu'il faut l'ajouter. On crée le nœud et on lui affecte l'élément. Ce nouveau nœud devient une feuille de l'arbre.

### Spécification formelle

Nous avons juste besoin de déclarer la fonction Ajouter-feuille qui s'écrit :

**OPERATION**  
Ajouter-feuille : Element x ArbreBinaire → ArbreBinaire

**AXIOMES**

Ajouter-feuille(x,Arbre-vide)= $\langle x, \text{Arbre-vide}, \text{Arbre-vide} \rangle$   
 $x \leq r \Rightarrow \text{Ajouter-feuille}(x, \langle r, G, D \rangle) = \langle r, \text{Ajouter-feuille}(x, G), D \rangle$   
 $x > r \Rightarrow \text{Ajouter-feuille}(x, \langle r, G, D \rangle) = \langle r, G, \text{Ajouter-feuille}(x, D) \rangle$

**AVEC**

G, D : ArbreBinaire  
x, r : Element  
 $r \rightarrow \text{contenu}(\text{racine}(r))$

### Algorithme

(version récursive)

Cette version est, comme d'habitude, une pure traduction de la définition axiomatique, et donne :

Algorithme Fonction Ajouter\_feuille : ArbreBinaire

Paramètres locaux

Element            x  
ArbreBinaire      B

Debut

Si B=Arbre-vide alors

    Retourne( $\langle x, \text{Arbre-vide}, \text{Arbre-vide} \rangle$ ) /\* Crédit de la feuille contenant x \*/

Sinon

    Si  $x \leq \text{contenu}(\text{racine}(B))$  alors /\* poursuite sur sous-arbre gauche \*/

        Retourne( $\langle r, \text{Ajouter\_feuille}(x, g(B)), D \rangle$ )

    Sinon /\* poursuite sur sous-arbre droit \*/

        Retourne( $\langle r, G, \text{Ajouter\_feuille}(x, d(B)) \rangle$ )

    Fin si

Fin si

Fin algorithme fonction Ajouter\_feuille

(version itérative)

Pour cette version, il est nécessaire de tester à l'avance si le nœud suivant sera une feuille. En effet, il n'y a pas d'affectation de la feuille à l'arbre au retour et nous ne voulons pas mettre en place une pile de gestion du chemin parcouru (oui c'est vrai, si on peut s'en passer). Par conséquent, nous allons devoir expressément affecter le nœud à son père, donc le connaître, d'où le test sur l'existence des fils un coup avant.

**Algorithme Fonction Ajouter\_feuille : ArbreBinaire**  
**Paramètres locaux**  
 Element x  
 ArbreBinaire B  
**Variables**  
 ArbreBinaire Aux *Aux 2*  
 Booléen Poursuite  
**Debut** *Aux 2*  $\leftarrow$  B  
 Aux  $\leftarrow$  x, Arbre-vide, Arbre-vide) /\* Création du noeud contenant x \*/  
 Si B=Arbre-vide alors  
     B  $\leftarrow$  Aux *Aux 2*  $\leftarrow$  Aux  
 Sinon  
     Poursuite  $\leftarrow$  Vrai  
     Tant que Poursuite Faire  
         Si x <= contenu(racine(B)) alors  
             Si g(B)=Arbre-vide Alors /\* test d'existence du fils gauche \*/  
                 g(B)  $\leftarrow$  Aux  
                 Poursuite  $\leftarrow$  Faux  
             Sinon  
                 B  $\leftarrow$  g(B)  
             Fin si  
         Sinon  
             Si g(B)=Arbre-vide Alors /\* test d'existence du fils droit \*/  
                 d(B)  $\leftarrow$  Aux  
                 Poursuite  $\leftarrow$  Faux  
             Sinon  
                 B  $\leftarrow$  d(B)  
             Fin si  
         Fin tant que  
     Fin si  
     Retourne(*Aux 2*)  
**Fin algorithme fonction Ajouter\_feuille**

### Construction d'un ABR par ajout successif aux feuilles

Nous pouvons donc construire un ABR par ajouts successifs d'éléments aux feuilles. Comme nous pouvons le voir sur la *Figure 2*, voilà comment cela se déroulerait pour l'ajout des éléments *h, e, c, t, p, d, v, a*. Comme nous pouvons le constater, la structure de l'arbre dépend de l'ordre d'arrivée des éléments (Incroyable, j'y ai pensé). Si, par exemple, le *t* était arrivé le premier, l'arbre aurait cet élément pour contenu de sa racine.

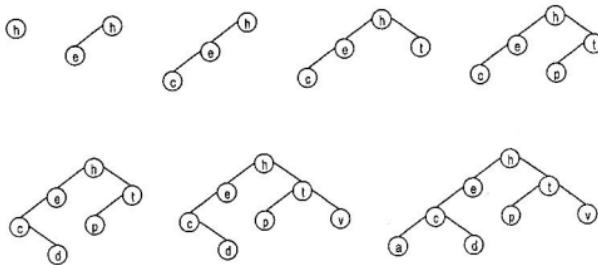


Figure 2. Construction de l'ABR par ajouts successifs aux feuilles de la série {h,e,c,t,p,d,v,a}.

## Ajout à la racine

### Principe algorithmique

En fait, cette façon d'ajouter nous permet de créer des nœuds à la racine et à n'importe quel niveau de l'arbre dans la mesure où tout nœud est la racine d'un arbre. Pour ajouter un élément  $x$  à la racine d'un arbre binaire  $B$  il faut d'abord découper celui-ci en deux arbres binaires  $G$  et  $D$  formés respectivement des éléments inférieurs ou égaux à  $x$  et des éléments supérieurs à  $x$  (Figure 3). Ensuite, il ne reste plus qu'à créer un nouvel arbre dont la racine est  $x$  et dont les sous-arbres gauche et droit sont  $G$  et  $D$ .

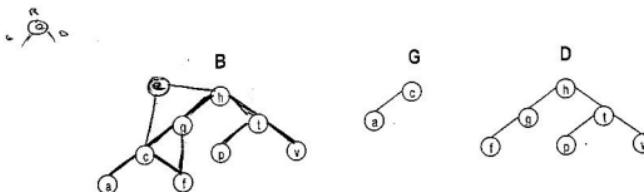


Figure 3. Coupe de l'arbre  $B$  en sous-arbres  $G$  et  $D$  selon l'élément (e).

L'étape la plus importante de cet algorithme est évidemment celle de la coupe. Il n'est pas nécessaire de parcourir tous les nœuds de l'arbre. En fait, le parcours de coupe est celui de la recherche de position d'un nœud pour un ajout aux feuilles. Lorsque l'on trouve un élément plus petit (grand), il emmène avec lui tout son sous-arbre gauche (droit).

### Spécification formelle

Nous avons juste besoin de déclarer la fonctions Ajouter-racine qui s'écrit :

OPERATION  
 Ajouter-racine : Element x ArbreBinaire → ArbreBinaire

**AXIOMES**

```

Ajouter-racine(x, Arbre-vide) = <x, Arbre-vide, Arbre-vide>
Racine(ajouter-racine(x, <r, G, D>)) = x
x ≤ r => g(Ajouter-racine(x, <r, G, D>)) = g(Ajouter-racine(x, G))
x ≤ r => d(Ajouter-racine(x, <r, G, D>)) = <r, d(Ajouter-racine(x, G)), D>
x > r => d(Ajouter-racine(x, <r, G, D>)) = d(Ajouter-racine(x, D))
x > r => g(Ajouter-racine(x, <r, G, D>)) = <r, G, g(Ajouter-racine(x, D))>

```

**AVEC**

G, D : ArbreBinaire  
 x, r : Element

Algorithme  
*(Procédure de coupe)*

Cette procédure récursive coupe l'ABR B en deux ABRs G et D selon un l'élément x. G contient tous les éléments inférieurs ou égaux à x et D tous ceux qui lui sont supérieurs. Cela donne :

Algorithme Procedure Couper  
 Paramètres locaux  
 Element x  
 Paramètres globaux  
 ArbreBinaire B,G,D  
 Debut  
 Si B=Arbre-vide alors  
     G←Arbre-vide  
     D←Arbre-vide  
 Sinon  
     Si x<=contenu(racine(B)) alors  
         D←B /\* On relie à D, B et son sous-arbre droit \*/  
         Couper(x, g(B), G, g(D)) /\* poursuite sur sous-arbre gauche \*/  
     Sinon  
         G←B /\* On relie à D, B et son sous-arbre gauche \*/  
         Couper(x, d(B), d(G), D) /\* poursuite sur sous-arbre gauche \*/  
     Fin si  
 Fin si  
 Fin algorithme fonction Couper



(Procédure d'ajout à la racine)

Cette procédure crée le nouveau nœud contenant l'élément x, puis fait appel à la procédure couper pour connaître les arbres à lui affecter comme fils. Nous pouvons noter que les liens sur ces fils étant passés comme argument par adresse, leur affectation est automatiquement faite par la procédure couper. Ce qui donne :

Algorithme Procedure Ajouter\_racine  
 Paramètres locaux :  
 Element x

```

Paramètres globaux
    ArbreBinaire    B
Variables
    ArbreBinaire    R
Debut
    R ← <x,Arbre-vide,Arbre-vide>          /* Création du nœud contenant x */
    Couper(x, B, g(R), d(R))                /* Appel de Couper */
    B ← R
Fin algorithme Procedure Ajouter-racine

```

## Suppression d'un élément

Pour supprimer un élément d'un ABR, il faut d'abord déterminer sa place, ensuite le supprimer et enfin regarder si cela nécessite une réorganisation. On dégage alors trois cas :

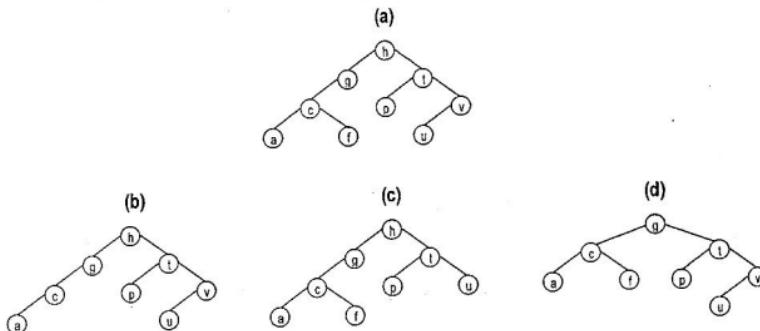


Figure 4. Exemples de suppression dans un Arbre binaire de recherche.

- C'est une feuille (pas de fils). Dans ce cas, il est simplement supprimé. Par exemple, Si la figure 4.a est l'arbre d'origine, la suppression de la feuille {f} donnera l'arbre figure 4.b.
- C'est un point simple (1 seul fils). Dans ce cas, il est supprimé et son fils prend sa place. Par exemple, Si la figure 4.a est l'arbre d'origine, la suppression du nœud v donnera l'arbre figure 4.c. On constate que son fils gauche {u} l'a remplacé.
- C'est un point double (2 fils). Dans ce cas, deux possibilités :
  - Le remplacer par l'élément qui lui est immédiatement inférieur, à savoir le plus grand des plus petits (bout du bord droit de son sous-arbre gauche). . Par exemple, Si la figure 4.a est l'arbre d'origine, la suppression du nœud racine {h} donnera l'arbre figure 4.d. On constate que le nœud {g} a été supprimé et a remplacé le nœud racine. C'était effectivement le plus grand des plus petits.

- Le remplacer par l'élément qui lui est immédiatement supérieur, à savoir le plus petit des plus grands (bout du bord gauche de son sous-arbre droit).

*Remarque : Ces deux dernières solutions sont équivalentes si tous les éléments de l'ABR sont distincts.*

### Spécification formelle

Pour pouvoir effectuer la suppression, nous avons besoin de trois opérations, la suppression, la récupération du plus grand des plus petits (max) et une opération qui retourne l'arbre amputé (sans anesthésie) de cet élément (suppmax), ce qui s'écrit :

#### OPERATION

Supprimer	:	Element x ArbreBinaire → ArbreBinaire
Max	:	ArbreBinaire → Element
Suppmax	:	ArbreBinaire → ArbreBinaire

#### PRECONDITIONS

Max(G) est définissi G≠Arbre-vide  
Suppmax(G) est définissi G≠Arbre-vide

#### AXIOMES

Supprimer(x,Arbre-vide)=Arbre-vide  
 $x=r \Rightarrow \text{Supprimer}(x,<r,\text{Arbre-vide}>)=G$   
 $x=r \& D \neq \text{Arbre-vide} \Rightarrow \text{Supprimer}(x,<r,\text{Arbre-vide},D>)=D$   
 $x=r \& G \neq \text{Arbre-vide} \& D \neq \text{Arbre-vide}$   
 $\Rightarrow \text{Supprimer}(x,<r,G,D>) = <\text{Max}(G), \text{Suppmax}(G), D>$   
 $x < r \Rightarrow \text{Supprimer}(x,<r,G,D>) = <r, \text{Supprimer}(x,G), D>$   
 $x > r \Rightarrow \text{Supprimer}(x,<r,G,D>) = <r, G, \text{Suprimer}(x,D)>$

Max(<r,G,Arbre-vide>)=r  
 Suppmax(<r,G,Arbre-vide>)=G  
 $D \neq \text{Arbre-vide} \Rightarrow \text{Max}(<r,G,D>) = \text{Max}(D)$   
 $D \neq \text{Arbre-vide} \Rightarrow \text{Suppmax}(<r,G,D>) = <r, G, \text{Suppmax}(D)>$

#### AVEC

G, D	:	ArbreBinaire
x, r	:	Element

### Algorithme

*Remarque : Ces deux fonctions présentent un ordre de désallocation (libération), ce n'est théoriquement pas nécessaire sur un algorithme abstrait. Mais le préciser permettra de ne pas l'oublier lors de l'implémentation dans un langage.*

### (Fonction Suppmax)

Cette fonction récursive regroupe les deux opérations Max et Suppmax. En effet, elle renvoie le plus grand élément d'un arbre et cet arbre amputé (Sans... Non pas là !) de cet élément. Ce qui donne :

Algorithme Fonction Suppmax : Element

Paramètres locaux

ArbreBinaire B

Variables

ArbreBinaire R

Element Max

Debut

Si d(B)=Arbre-vide alors

~~REC~~

Max ← contenu(racine(B)) /\* Récupération de la valeur Max \*/

B ← g(B) /\* Affectation du fils gauche \*/

Libérer(R) /\* Désallocation du Nœud \*/

Retourne(Max)

Sinon

Retourne(Suppmax(d(B))) /\* Poursuite sur bord droit \*/

Fin si

Fin algorithme fonction Suppmax

(Procédure de suppression)

Cette procédure recherche le nœud contenant l'élément x, puis détermine combien il possède de fils. Selon les différents cas, la suppression se fait directement ou fait appel à Suppmax(2 fils). Ce qui donne :

Algorithme Procedure Supprimer

Paramètres locaux :

Element x

Paramètres globaux

ArbreBinaire B

Variables

ArbreBinaire R

Debut

Si B > Arbre-vide alors /\* Recherche de x \*/

Si x < Contenu(racine(B)) alors  $A_{\forall x} \leftarrow B$

Supprimer(x,g(B))

Sinon  $R \leftarrow A_{\forall x}$

Si x > Contenu(racine(B)) alors  $A_{\forall x} \leftarrow B$

Supprimer(x,d(B))  $B \leftarrow A_{\forall x}$

Sinon

Si g(B)=Arbre-vide alors

R ← B

B ← d(B)

Libérer(R)

Sinon

Si d(B)=Arbre-vide alors

R ← B

B ← g(B)

/\* Trouvé \*/

/\* 1 seul fils (le droit) \*/

/\* Affectation du fils droit \*/

/\* Désallocation du Nœud \*/

/\* 1 seul fils (le gauche) \*/

/\* Affectation du fils gauche \*/

```

        Libérer(R)           /* Désallocation du Nœud */
        Sinon
            Contenu(racine(B)) ← Suppmax(g(B))
        Fin si
    Fin si
    Fin si
Fin si
Fin algorithme Procedure Supprimer

```

## Conclusion

Dans le cas où tous les éléments sont distincts, on montre que la complexité est logarithmique en moyenne et linéaire au pire. Les opérations d'ajout, de modification et de suppression reposent sur le nombre de comparaisons de nœuds. C'est l'opération fondamentale pour l'analyse de la complexité de ces algorithmes. Ce qui nous donne le tableau suivant pour un nœud  $x$  donné et pour une recherche positive et négative :

	Recherche positive	Recherche Négative
Rechercher	$2^{\text{Profondeur}(x)+1}$	$2^{\text{Profondeur}(x)+1}$
Ajouter-Feuille		$\text{Profondeur}(x)+1$
Ajouter-racine		$\text{Profondeur}(x)+1$
Supprimer	$2^{\text{Profondeur}(x)+1}$	$2^{\text{Profondeur}(x)+1}$

Il n'y a pas de précision sur recherche positive pour l'ajout dans la mesure où l'on fonctionne sans redondance d'éléments, donc pas de recherche d'égalité. L'ajout est divisé par deux par rapport à la recherche ou à la suppression parce qu'il n'y a qu'un seul test de comparaison au lieu de deux.

Quoi qu'il en soit, les complexités pour ces quatre opérations sont d'ordre logarithmique en moyenne et linéaire au pire.



# Chapitre 7

## Algorithmes de recherche – Arbres équilibrés

Dans le chapitre précédent, nous avons vu que la complexité des opérations de recherche, ajout et suppression pouvait être au pire linéaire. Cela est du au fait qu'un arbre binaire n'est pas nécessairement équilibré, et que dans le pire des cas, il peut être filiforme. Imaginez un ajout aux feuilles avec comme entrées successives la liste  $\{a, b, c, d, e, f\}$ .

Pour conserver, cette complexité logarithmique, nous devons donc équilibrer l'arbre. Dans ce cas, il est clair que la réorganisation de celui-ci doit être la plus rapide possible. C'est pourquoi, nous autoriserons un léger déséquilibre de l'arbre. Nous allons dans ce chapitre étudier deux classes d'arbres équilibrés, les AVL (arbres binaires) et les Arbres 2.3.4 (arbres généraux). Pour chacune de ces classes, les complexités des algorithmes de recherche, d'ajout et de suppression seront logarithmiques.

### Rotations

Dans un premier temps, nous allons implémenter des algorithmes de rééquilibrage appelés rotations. Ils effectuent des transformations locales de l'arbre et sont au nombre de quatre : **rotation simples** (gauche et droite) ; **rotation double** (gauche-droite et droite-gauche). Des exemples de ces rotations sont montrés en figure 1,2,3 et 4.



Figure 1. Rotation droite de l'arbre A.

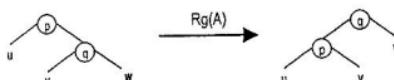


Figure 2. Rotation gauche de l'arbre A.

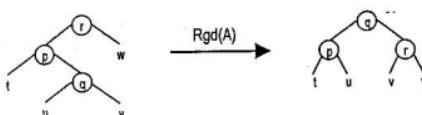


Figure 3. Rotation gauche-droite de l'arbre A.

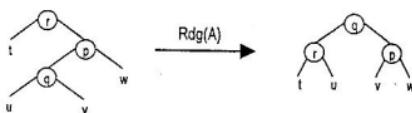


Figure 4. Rotation droite-gauche de l'arbre A.

### Spécification formelle

Ces quatre opérations (rg, rd, rgd et rdg) sont définies de la manière suivante :

#### OPERATION

```

rg : ArbreBinaire → ArbreBinaire
rd : ArbreBinaire → ArbreBinaire
rgd : ArbreBinaire → ArbreBinaire
rdg : ArbreBinaire → ArbreBinaire

```

#### PRECONDITIONS

```

rg(B) est défini ssi B≠Arbre-vide & d(B)≠Arbre-vide
rd(B) est défini ssi B≠Arbre-vide & g(B)≠Arbre-vide
rgd(B) est défini ssi
    B≠Arbre-vide & g(B)≠Arbre-vide & d(g(B))≠Arbre-vide
rdg(B) est défini ssi
    B≠Arbre-vide & d(B)≠Arbre-vide & g(d(B))≠Arbre-vide

```

#### AXIOMES

```

rg(<p,U,<q,V,W>>) = <q,<p,U,V> , W>
rd(<q,<p,U,V> , W>) = <p,U,<q,V,W>>
rgd(<r,<p,T,<q,U,V> , W>>) = <q,<p,T,U>,<r,V,W>>
rdg(<r,T,<p,<q,U,V> , W>>) = <q,<r,T,U>,<p,V,W>>

```

#### AVEC

```

B,T,U,V,W : ArbreBinaire
P,q,r : Nœud

```



### Algorithme (Procédure rg)

Algorithme Procédure Rg

Paramètres globaux

ArbreBinaire B

Variables

ArbreBinaire R

Début

```

R ← d(B)           /* Affectations des différents liens */
d(B) ← g(R)
g(R) ← B
B ← R

```

Fin algorithme Procédure Rg

(Procédure rd)

```

Algorithme Procedure Rd
Paramètres globaux
    ArbreBinaire    B
Variables
    ArbreBinaire    R
Debut
    R ← g(B)          /* Affectations des différents liens */
    g(B) ← d(R)
    d(R) ← B
    B ← R
Fin algorithme Procedure Rd

```

(Procédure rgd)

```

Algorithme Procedure Rgd
Paramètres globaux
    ArbreBinaire    B
Debut
    Rg(g(B))        /* Appel des rotations simples */
    Rd(B)
Fin algorithme Procedure Rgd

```

(Procédure rdg)

```

Algorithme Procedure Rdg
Paramètres globaux
    ArbreBinaire    B
Debut
    Rd(d(B))        /* Appel des rotations simples */
    Rg(B)
Fin algorithme Procedure Rdg

```

Remarque : Comme on peut le constater, les rotations doubles sont la combinaison de deux rotations simples. La rotation gauche-droite (droite-gauche) est composée d'une rotation gauche (droite) sur le sous-arbre gauche (sous-arbre droit) de B suivie d'une rotation droite (gauche) sur B.

Ces deux fonctions sont optimisables en remplaçant les deux appels de fonction par les affectations adéquates. L'intérêt étant d'éviter les deux appels de fonctions et de réduire le nombre d'affectations de 8 à 6.

Remarque 2 : Les rotations conservent la propriété d'arbre binaire de recherche (heureusement, hein ?). On constate que la lecture symétrique de l'arbre B, rd(B), rg(B), rgd(B) et de rdg(B) est la même.

appel de procédure Rdg

Paramètres globaux	Rdg
ArbreBinaire B	d(B) ← d(B)
Variables	d(B) ← d(B)
ArbreBinaire T	g(T) ← g(T)
Debut	g(T) ← g(B)
	d(T) ← d(g(B))
	d(g(B)) ← g(T)
	g(T) ← g(B)
	g(B) ← d(T)
	d(T) ← B
	B ← T
	fin

## Arbres A.V.L.

Les arbres AVL datent des années 60 et sont la première classe d'arbres *H-équilibrés*. Leur nom vient de leurs concepteurs (Adelson-Velskii, Landis).

### Définitions

On dit qu'un arbre binaire est *H-équilibré* si en tout nœud de l'arbre, les hauteurs des sous-arbres gauche et droit diffèrent au plus de 1.

#### Spécification formelle

Nous définissons donc une opération de déséquilibre, définie récursivement par :

##### OPERATION

déséquilibre : ArbreBinaire → Entier

##### AXIOMES

déséquilibre(Arbre-vide)=0

déséquilibre(<o, G, D>) = hauteur(G) - hauteur(D)

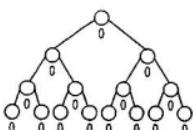
##### AVEC

B, G, D : ArbreBinaire

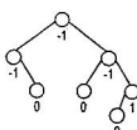
o : Nœud

Donc, un Arbre B est équilibré si pour tout sous-arbre C de B on a :  $Déséquilibre(C) \in \{-1, 0, 1\}$

Nous pouvons voir sur la Figure 5.a un exemple d'arbres H-équilibré, et sur la Figure 5.b l'exemple d'un arbre qui ne l'est pas.



(a) Arbres H-équilibrés



(b) Arbre Non H-équilibré

Déséquilibre

Figure 5. Exemples d'arbres H-équilibré et non H-équilibré.

Nous ne détaillerons pas l'algorithme de recherche dans les AVL. Dans la mesure où ceux-ci sont des arbres binaires de recherche, il suffit de reprendre ceux de la recherche dans un ABR. Le nombre de comparaisons est toujours d'ordre logarithmique.

En revanche, l'ajout ou la suppression dans un AVL peut provoquer un déséquilibre qui générera alors une réorganisation locale ou complète de l'arbre. Nous allons donc étudier (enfin on va essayer) les algorithmes d'ajout et de suppression dans les AVL.

## Ajout dans un AVL

Le principe de construction d'un AVL est le suivant ; on fait l'ajout du nouvel élément aux feuilles. Puis on rééquilibre l'arbre si cet ajout l'a déséquilibré. Dans ce cas, le déséquilibre maximum que l'on peut rencontrer est  $\pm 2$ . Lorsqu'il y a déséquilibre après l'ajout d'un nœud  $x$ , il suffit de rééquilibrer l'arbre à partir d'un nœud  $y$  (dans le chemin de la racine à la feuille  $x$ ). Ce nœud  $y$  est le dernier (le plus proche de  $x$ ) pour lequel le déséquilibre est  $\pm 2$ .

### Principe général de rééquilibrage

Soit  $B = \langle r, G, D \rangle$  un AVL, donc n'ayant pas de sous-arbres en déséquilibre  $\pm 2$ . Supposons que l'on ajoute un élément  $x$  sur une feuille de  $G$  et que la hauteur de ce dernier augmente de 1 et que  $G$  reste un AVL (avant l'ajout dans  $G$ , son déséquilibre était 0). Dans ce cas, nous avons les trois possibilités suivantes :

- Si le déséquilibre de  $B$  valait 0 avant l'ajout, il vaut 1 après.  $B$  reste un AVL et sa hauteur a augmenté de 1.
- Si le déséquilibre de  $B$  valait  $-1$  avant l'ajout, il vaut 0 après.  $B$  reste un AVL et sa hauteur n'est pas modifiée.
- Si le déséquilibre de  $B$  valait 1 avant l'ajout, il vaut 2 après.  $B$  n'est plus H-équilibré, il faut le réorganiser.

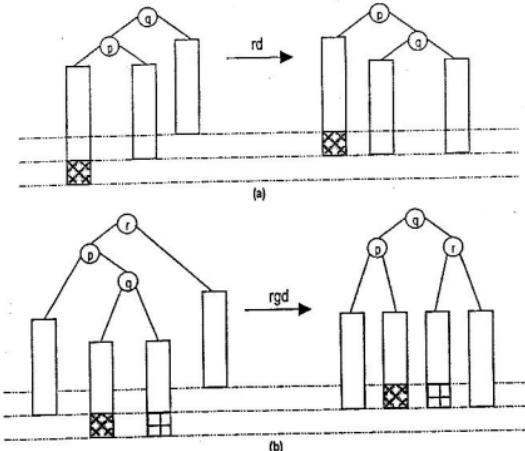


Figure 6.Ajonction dans un AVL.

Comme on peut le voir sur la *Figure 6*, la troisième hypothèse présente deux possibilités :

- Le déséquilibre de G passe de 0 à 1 (*Figure 6.a*) et le rééquilibrage de B se fait à l'aide d'une rotation simple à droite.
- Le déséquilibre de G passe de 0 à -1, il y a inversion du sens de déséquilibre entre B et G. Dans ce cas, le rééquilibrage s'effectue à l'aide d'une rotation double gauche-droite (*Figure 6.b*).

Dans les deux cas, on récupère un arbre B rééquilibré qui est bien un AVL et dont la hauteur est redevenue celle qu'il avait avant l'ajout de l'élément x (magnifique !).

*Remarque : Bien évidemment il existe les trois cas symétriques dans le cas où l'ajout se fasse sur D.*

#### Spécification formelle

Pour l'ajout dans un AVL, nous avons besoin d'une opération ajouter-AVL et de deux opérations gérant le déséquilibre, à savoir « rééquilibrer » et « déséquilibre ». La première rééquilibrer l'arbre, la deuxième permet de connaître le déséquilibre d'un arbre. Ces opérations sont définies de la manière suivante :

##### OPERATION

```
Ajouter-avl   : Element x ArbreBinaire → ArbreBinaire
rééquilibrer  : ArbreBinaire → ArbreBinaire
déséquilibre  : ArbreBinaire → Entier
```

##### PRECONDITIONS

rééquilibrer(B) est défini ssi déséquilibre(B) ∈ {-2, -1, 0, 1, 2}

##### AXIOMES

```
Ajouter-avl(x, Arbre-vide)=x
x≤r => Ajouter-avl(x, <r, G, D>) = rééquilibrer(<r, ajouter-avl(x, G), D>)
x>r => Ajouter-avl(x, <r, G, D>) = rééquilibrer(<r, G, ajouter-avl(x, D)>)
Déséquilibre(B)=-1 => Rééquilibrer(B)=B
Déséquilibre(B)=0 => Rééquilibrer(B)=B
Déséquilibre(B)=1 => Rééquilibrer(B)=B
Déséquilibre(B)=2 & Déséquilibre(G)=1 => Rééquilibrer(B)=rd(B)
Déséquilibre(B)=2 & Déséquilibre(G)=-1 => Rééquilibrer(B)=rgd(B)
Déséquilibre(B)=-2 & Déséquilibre(D)=1 => Rééquilibrer(B)=rdg(B)
Déséquilibre(B)=-2 & Déséquilibre(D)=-1 => Rééquilibrer(B)=rg(B)
```

##### AVEC

```
B, G, D    : ArbreBinaire
r          : Nœud
x          : Element
```

*Remarque : On peut noter qu'il y a au plus un rééquilibrage à chaque ajout.*



### Algorithme

En terme d'implémentation, nous devons mémoriser dans chaque nœud la valeur de déséquilibre de l'arbre dont il est racine. Il est donc nécessaire de rajouter un champ Deseq prenant les valeurs {-2,-1,0,1,2} dans l'enregistrement de type T\_Nœud défini dans le chapitre 3. Ce qui donne :

```
Type
    Element = .... /* Le type des éléments est défini là */
    T_Avl = ↑ T_Nœud /* type pointeur sur T_Nœud */

    T_Nœud = Enregistrement
        Entier Deseq
        Element Elt
        T_Avl Fg,fd
    Fin Enregistrement T_Nœud

Variable
    T_Avl B
```

Remarque : Nous allons rester dans un formalisme abstrait, et pour cela nous utiliserons le type Avl qui est un dérivé du type ArbreBinaire auquel nous avons ajouté les opérations propres aux Avl, en détaillant toutefois certaines d'entre elles comme rééquilibrer.

### (Procédure AjouterAvl)

```
Algorithme Procedure AjouterAvl
Paramètres globaux
    Avl B
Paramètres locaux
    Element x
Variables
    Avl Y, A, AA, P, PP
Debut
    Y <->0,x,Arbre-vide,Arbre-vide> /* Création du nœud, Deseq=0 */
    Si B=Arbre-vide Alors
        B <- Y
    Sinon
        A <- B
        AA <- Arbre-vide /* AA est le père de A */
        P <- B
        PP <- Arbre-vide /* PP est le père de P */
        Tant que P>>Arbre-vide Faire /* Recherche de la feuille d'ajout */
            Si Déséquilibre(P)>>0 Alors /* Si un nœud a un déséquilibre */
                A <- P /* différent de 0, on le mémorise */
                AA <- PP /* dans A, et son père dans AA */
            Fin si
            PP <- P
            Si x≤Contenu(racine(P)) Alors
```



```
P ← g(P)
Sinon
    P ← d(P)
Fin si
Fin tant que
Si x ≤ Contenu(racine(PP)) Alors      /* Ajout du nœud Y */
    g(PP) ← Y
Sinon
    d(PP) ← Y
Fin si
P ← A                                /* Modification du déséquilibre */
Tant que P <> Y Faire                /* sur le chemin de A vers Y */
    Si x ≤ Contenu(racine(P)) Alors
        Déséquilibre(P) ← Déséquilibre(P)+1
        P ← g(P)
    Sinon
        Déséquilibre(P) ← Déséquilibre(P)-1
        P ← d(P)
    Fin si
Fin tant que
Selon les valeurs de Déséquilibre(A) Faire /* Rééquilibrage */
    0, 1, -1 : Retourne /* Quitte l'ajout */
    2 : Si Déséquilibre(g(A))=1 Alors /* Déséquilibre Gauche */
        Rd(A) /* Rotation droite */
        Déséquilibre(A) ← 0
        Déséquilibre(d(A)) ← 0
    Sinon
        Rgd(A) /* Rotation gauche-droite */
    Selon les valeurs de Déséquilibre(A) Faire
        -1 : Déséquilibre(g(A)) ← 1
            Déséquilibre(d(A)) ← 0
        1 : Déséquilibre(g(A)) ← 0
            Déséquilibre(d(A)) ← -1
        0 : Déséquilibre(g(A)) ← 0 /* A=Y */
            Déséquilibre(d(A)) ← 0
    Fin selon
    Déséquilibre(A) ← 0
Fin si
-2 : Si Déséquilibre(d(A))=-1 Alors /* Déséquilibre Droit */
    Rg(A) /* Rotation gauche */
    Déséquilibre(A) ← 0
    Déséquilibre(d(A)) ← 0
Sinon
    Rgd(A) /* Rotation droite-gauche */
Selon les valeurs de Déséquilibre(A) Faire
-1 : Déséquilibre(g(A)) ← 0
```

```

        Déséquilibre(d(A))←-1
1 : Déséquilibre(g(A))←1
        Déséquilibre(d(A))←0
0 : Déséquilibre(g(A))←0 /* A=Y */
        Déséquilibre(d(A))←0
    Fin selon
    Déséquilibre(A)←0
Fin si

Fin selon
Si AA=Arbre-vide Alors /* Mise à jour des liens */
    B←A
Sinon
    Si Contenu(racine(A))≤Contenu(racine(AA)) Alors
        g(AA)←A
    Sinon
        d(AA)←A
    Fin si
Fin si
Fin si
Fin algorithme Procedure AjouterAvl

```

Remarque : L'utilité du débranchement (et là, je m'adresse à certains correcteurs de ce poly...), pour un déséquilibre de {0,1,-1}, est nécessaire pour éviter la mise à jour des pointeurs en sortie. Cela permet d'éviter un test supplémentaire à la fin (on ne va quand même pas en rajouter, c'est suffisamment pénible comme ça !).

## Supprimer dans un AVL

Le principe de suppression d'un élément dans un AVL est le même que dans un arbre binaire. C'est à dire recherche de l'élément à supprimer, suppression et remplacement, le cas échéant, par l'élément qui lui est immédiatement inférieur. Le problème est que cet arbre n'est plus forcément *H-équilibré*. Il faut donc le rééquilibrer. Après la première phase de suppression, la hauteur de l'arbre est diminuée de 1. S'il y a déséquilibre, la rotation appliquée peut diminuer à son tour la hauteur de l'arbre et générer, éventuellement un nouveau déséquilibre. En fait les rotations peuvent s'enchaîner en cascade depuis l'élément supprimé jusqu'à la racine de l'arbre. Nous devons donc connaître ce chemin. Cela implique l'utilisation d'une pile sur une version itérative de l'algorithme.

En fait le principe est simple, si l'on supprime un nœud x, on examine la hauteur de l'arbre de racine x. Si cette hauteur a diminué de 1, il faut éventuellement faire une rotation au niveau du père de x. Et ainsi de suite...

### Spécification formelle

Pour supprimer dans un AVL, nous avons besoin des opérations « rééquilibrage » (définie sur l'ajout AVL), de « Max » et « Suppmax » (définies dans la suppression ABR) et de « Supprimer-AVL » que nous allons définir, soit :

**OPERATION**

Supprimer-avl : Element x ArbreBinaire  $\rightarrow$  ArbreBinaire  
rééquilibrer : ArbreBinaire  $\rightarrow$  ArbreBinaire  
Max : ArbreBinaire  $\rightarrow$  Entier  
Suppmax : ArbreBinaire  $\rightarrow$  ArbreBinaire

**PRECONDITIONS**

Max(B) est défini ssi B $\neq$ Arbre-Vide  
Suppmax(B) est défini ssi B $\neq$ Arbre-Vide

**AXIOMES**

Supprimer-avl(x, Arbre-vide)=Arbre-vide  
 $x=r \Rightarrow$  Supprimer-avl(x, <r, G, Arbre-vide>) = G  
 $x=r \& D \neq$ Arbre-Vide  $\Rightarrow$  Supprimer-avl(x, <r, Arbre-vide, D>) = D  
 $x=r \& G \neq$ Arbre-Vide & D $\neq$ Arbre-Vide  
 $\Rightarrow$  Supprimer-avl(x, <r, G, D>) = rééquilibrer(<Max(G), Suppmax(G), D>)

$x < r \Rightarrow$  Supprimer-avl(x, <r, G, D>) = rééquilibrer(<r, Supprimer-avl(x, G), D>)  
 $x > r \Rightarrow$  Supprimer-avl(x, <r, G, D>) = rééquilibrer(<r, G, Supprimer-avl(x, D)>)  
 $D \neq$ Arbre-Vide  $\Rightarrow$  Max(<r, G, D>) = Max(D)  
 $D \neq$ Arbre-Vide  $\Rightarrow$  Suppmax(<r, G, D>) = rééquilibrer(<r, G, Suppmax(D)>)

**AVEC**

B, G, D : ArbreBinaire  
r : Nœud  
x : Element

Après suppression d'un nœud  $r$ , il faut examiner la hauteur du sous-arbre  $V_r$ . Si la hauteur a diminué de 1, alors on a une rotation au niveau du père de  $r$ . Remarque : On peut noter qu'il peut y avoir plus d'un rééquilibrage à chaque suppression.

**Algorithme**

L'algorithme de suppression faisant l'objet d'un exercice, il n'est pas fourni dans le présent polycopié. Je sais c'est un peu facile. Mais bon, personne ne le fournit, je ne vois pas pourquoi je le ferais. De plus, je m'attirerais alors l'inimitié de certains collègues (dont je tairais les noms) qui se retrouveraient, du coup, avec un exercice de moins en TD.

**Arbres 2.3.4**

Nous avons vu que pour éviter la dégénérescence de l'arbre, il existait les arbres binaires H-équilibrés, une autre possibilité est de multiplier les axes de recherche à partir d'un nœud. Dans ce cas, il nous faut utiliser des arbres de recherche. L'arbre 2.3.4 en est un. Ses nœuds peuvent contenir 1, 2 ou 3 éléments et toutes ses feuilles sont situées au même niveau. La hauteur de l'arbre est alors toujours une fonction logarithmique du nombre d'éléments qu'il contient. Il existe des fonctions de rééquilibrage de l'arbre, utilisées éventuellement après chaque ajout ou suppression d'un élément de l'arbre. Toutes les opérations sur les arbres 2.3.4 sont au pire logarithmique.

## Définitions

Un **arbre de recherche** est un arbre général étiqueté dont chaque nœud contient un **k-uplet** d'éléments distincts et ordonnés ( $k=1, 2, 3, \dots$ ). Un nœud contenant les éléments  $x_1 < x_2 < \dots < x_k$  possède  $k+1$  sous-arbres, tels que :

- Tous les éléments du premier sous-arbre sont inférieurs ou égaux à  $x_1$
- Tous les éléments du  $j^{\text{ème}}$  sous-arbre ( $j=2, \dots, k$ ) sont strictement supérieurs à  $x_{j-1}$  et inférieurs ou égaux à  $x_j$
- Tous les éléments du  $(k+1)^{\text{ème}}$  sous-arbre sont strictement supérieurs à  $x_k$

Dans un arbre de recherche, on appelle **k-nœud** un nœud contenant  $(k-1)$  éléments.

Un **arbre 2.3.4** est un arbre de recherche dont les nœuds sont de trois types : 2-nœuds, 3-nœuds ou 4-nœuds et dont toutes les feuilles sont situées au même niveau.

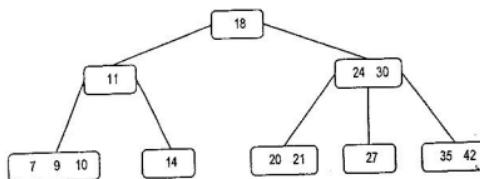


Figure 7.Exemple d'arbre 2.3.4

Comme nous pouvons le constater sur la figure 7, nous avons un exemple d'arbre 2.3.4 contenant 13 éléments distincts et constitué de 8 nœuds. Les éléments à l'intérieur de chaque nœud sont rangés en ordre croissant et toutes les feuilles de l'arbre se situent au même niveau.

## Recherche dans un Arbre 2.3.4

La recherche d'un élément  $x$  dans un arbre 2.3.4 A fonctionne globalement de la même manière que pour la recherche dans un **ABR**, le principe est le suivant :

On compare  $x$  avec l'élément ou les éléments  $x_1, \dots, x_i$  ( $i=1, 2$  ou  $3$ ) contenus dans le nœud racine de A , dans ce cas plusieurs possibilités :

- s'il existe  $j \in [1, i]$  tel que  $x=x_j$ , alors on a trouvé x et la recherche s'arrête (recherche positive)
- si  $x \leq x_i$ , la recherche de x se poursuit dans le  $1^{\text{er}}$  sous-arbre de A
- si  $x_j < x \leq x_{j+1}$  ( $j=1, \dots, i-1$ ), la recherche de x se poursuit dans le  $(j+1)^{\text{ème}}$  sous-arbre de A

- si  $x > x_i$ , la recherche de  $x$  se poursuit dans le dernier sous-arbre de  $A$
- si la recherche se termine sur une feuille ne contenant pas  $x$ , alors  $x$  n'existe pas dans  $A$  (recherche négative)

*Remarque : La recherche d'un élément dans un arbre 2.3.4 suit un chemin de la racine vers une feuille. La hauteur de l'arbre étant fonction logarithmique du nombre d'élément, la complexité de l'algorithme de recherche en nombre de comparaisons entre éléments, est toujours logarithmique.*

## Ajout dans un arbre 2.3.4

Nous allons étudier l'ajout aux feuilles. Le seul problème posé par ce dernier se présente si la feuille qui doit recevoir le nouvel élément est un 4-nœud. Dans ce cas, il va falloir ajouter un nouveau nœud et rééquilibrer l'arbre. C'est une fonction d'éclatement qui permet cela. L'éclatement des 4-nœuds peut se faire : soit en descendant le long du chemin à la recherche de la feuille d'ajout, soit en remontant (algorithme implicitement récursif).

## Ajout avec éclatements à la remontée

Nous allons présenter un exemple d'ajouts successifs aux feuilles qui montre l'évolution de l'arbre. Nous utiliserons successivement les éléments suivants : 30, 11, 35, 18, 27, 42, 14, 10, 24, 7, 21, 9, 20.

L'ajout de 30 crée un 2-nœud qui se transforme en 3-nœud lors de l'ajout de 11 puis en 4-nœud pour 35, soit l'évolution présentée figure 8.

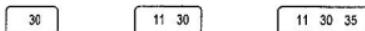


Figure 8. Ajout de 30, 11 et 35.

A ce stade, l'ajout d'un élément est impossible, l'unique feuille est un 4-nœud. Le nœud unique pourrait être assimilé à l'arbre binaire Figure 9. Pour pouvoir ajouter un élément, nous allons éclater le nœud en créant deux 2-nœuds contenant respectivement le plus petit et le plus grand élément du nœud éclaté. Si le nœud éclaté est racine de l'arbre, nous créons un nœud supplémentaire et la hauteur de l'arbre augmente de 1. Si le nœud éclaté n'est pas racine, l'élément central remonte vers le nœud père et s'y insère.

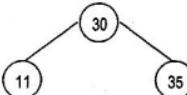


Figure 9. Arbre binaire équivalent de l'Arbre 2.3.4 de la Figure 8.

On reprend donc cet arbre qui est bien un arbre 2.3.4 et l'on ajoute 18, 27, 42, et l'on obtient l'arbre de la Figure 10.

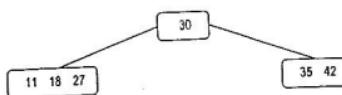


Figure 10.

L'ajout de 14 devrait se faire dans la première feuille, mais celle-ci est pleine. Nous devons donc l'éclater. Celle-ci n'étant pas une racine, l'élément central (18) va remonter vers le nœud père et s'insérer naturellement (enfin, c'est une façon d'écrire) à sa place. Nous obtiendrons alors une nouvelle feuille dans laquelle l'élément 14 va pouvoir s'insérer. Ensuite, nous ajoutons 10 et 24, ce qui donne l'arbre en Figure 11.

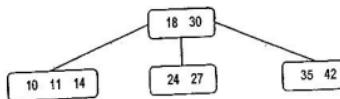


Figure 11.

L'ajout du 7 éclate la première feuille en générant la remontée du 11 vers la racine qui devient alors un 4-nœud. Ensuite, on ajoute le 21 et le 9. On obtient alors l'arbre 2.3.4 de la Figure 12.

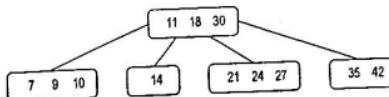


Figure 12.

Enfin, on ajoute le 20. Celui-ci devrait se trouver dans la troisième feuille qui nécessite un éclatement. Or, la remontée du 24 ne peut se faire dans la mesure où son nœud père (la racine) est aussi un 4-nœud. Dans ce cas, lui aussi sera éclaté et la hauteur de l'arbre va augmenter de 1. Ce qui donne l'arbre de la Figure 13.

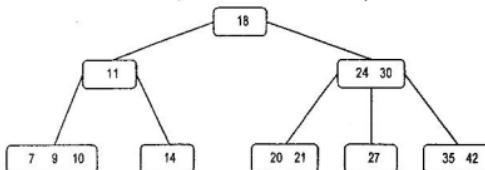


Figure 13.

### Ajout avec éclatements à la descente

Nous n'allons pas présenter l'ajout avec éclatement à la descente, mais faire simplement quelques remarques. Son but est d'éviter les cascades d'éclatement comme générées par l'ajout du 20 (Figure 13). En effet, lorsque l'on parcourt l'arbre à la recherche de la feuille dans laquelle doit s'effectuer l'ajout, nous éclatons systématiquement chaque 4-nœud rencontré. Dès lors, il ne peut pas y en avoir deux consécutifs (eh oui ! Je vous laisse réfléchir.). L'avantage de cette méthode est que l'on peut écrire une version itérative de l'algorithme sans utiliser de pile dans la mesure où nous n'avons pas besoin de mémoriser le chemin de la racine à la feuille que l'on crée. L'inconvénient est d'augmenter inutilement la hauteur de l'arbre (par éclatement de la racine) comme il aurait pu se produire à la Figure 12 si nous avions ajouté la valeur 13 ; Ce qui ne nécessitait pas l'éclatement de la racine dans le cas d'un ajout à la remontée.

*Remarque : Les deux méthodes opérant sur un chemin allant de la racine à la feuille dans laquelle doit se faire l'ajout, leur complexité est dans tous les cas fonction logarithmique du nombre d'éléments de l'arbre.*

*Remarque 2 : Dans le cas de redondance de valeurs, deux éléments de même valeur pourraient à la suite d'éclatements se trouver dans des nœuds différents. Nous ne traitons pas ces cas particuliers. En effet, les éléments d'un nœud fils pourraient ne pas être strictement supérieurs à celui du nœud père. Dans ce cas une modification de la définition des arbres 2.3.4 ainsi que des algorithmes de recherche et d'ajout serait nécessaire.*

### Représentation des arbres 2.3.4

Le passage à l'implémentation peut se faire de différentes manières, la première consiste à créer la structure propre à chaque nœud comme montré sur la Figure 14.

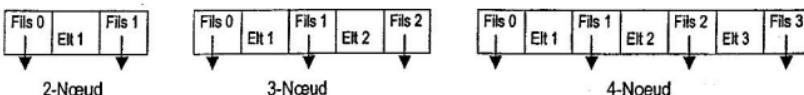


Figure 14.

Ce qui permet de minimiser l'utilisation mémoire, mais qui rend l'implémentation plus complexe, dans la mesure où il est nécessaire de traiter chaque opération pour chaque type de nœud. De plus l'augmentation du nombre de tests ralentira notablement l'exécution.

Une autre possibilité est la représentation maximale. C'est à dire que l'on prend le cas maximum (4-Nœud) et l'on utilise cette représentation pour tous les types de nœud comme montré Figure 15. Dans ce cas, si le nœud est un 2-nœud, nous n'utiliserons que les trois premiers champs de la structure.

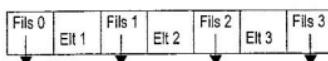


Figure 15. Représentation maximale d'un arbre 2.3.4

Le problème, dans ce cas, est le gaspillage de mémoire pour tout ce qui n'est pas 4-nœud. D'autre part, la suppression d'un élément oblige éventuellement le décalage des éléments et des pointeurs sur fils. Une possibilité de simplification est de construire une structure contenant deux tableaux : un de trois éléments et un de quatre pointeurs. Ce qui donnerait :

```
Constante
    MaxElt      = 3           /* Nombre maximum d'éléments */
    MaxFils     = MaxElt + 1

Type
    Element     = ....        /* Le type des éléments est défini là */
    T_A234      = ↑ T_Noeud   /* type pointeur sur T_Noeud */
    T_VectElt   = MaxElt Element /* Tableau d'éléments */
    T_VectFils  = MaxFils T_A234 /* Tableau de pointeurs sur fils */

    T_Noeud     = Enregistrement
        T_VectElt Elts
        T_VectFils Fils
    Fin Enregistrement T_Noeud

Variable
    T_A234     A
```

Il existe une troisième forme de représentation des arbres 2.3.4 qui présente l'avantage d'être uniforme et minimale. Ce sont les arbres bicolores.

## Arbres bicolores

Les arbres bicolores, inventés par Guibas et Sedgewick, sont des arbres binaires de recherche. Leur particularité tient au fait que leurs nœuds sont soit rouges, soit noirs. Ces arbres sont une représentation des arbres 2.3.4 et ne sont utilisés que dans le but de faciliter l'implémentation de ces derniers.

Dans un arbre 2.3.4, deux éléments appartenant au même nœud sont dits *jumeaux*. Dans un arbre binaire, les nœuds ne peuvent contenir qu'un seul élément, c'est là qu'intervient la couleur du nœud. Elle permet de

distinguer la hiérarchie des éléments entre eux. Si un nœud fils (dans l'arbre bicolore) contient un élément jumeau de celui contenu dans le nœud père, le nœud fils sera rouge. Si ces éléments appartiennent à deux nœuds différents dans l'arbre 2.3.4, le nœud fils (dans l'arbre bicolore) sera noir.

Nous allons présenter les étapes de transformation d'un arbre 2.3.4 en arbre bicolore. Le polycopié ne présentant pas la couleur, nous allons représenter les fils rouges à l'aide d'un double cercle. Pour un 4-nœud, la transformation est simple comme montré en *Figure 16*.

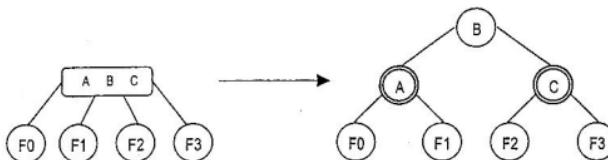


Figure 16. Transformation d'un 4-nœud.

Pour un 3-nœud, il existe deux possibilité. En effet, à la suite d'ajout et de rotation de rééquilibrage, le 3-nœud peut être penché à gauche ou penché à droite comme le montre la *Figure 17*.

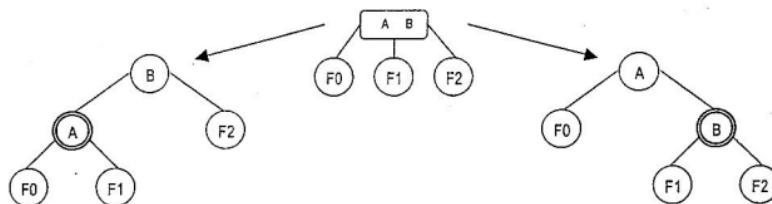


Figure 17. Transformation d'un 3-nœud (penché à gauche ou à droite).

Remarque : Comme il ne peut jamais y avoir deux liens rouges consécutifs (liens vers fils rouge) et que les chemins ont un nombre de liens noirs égal à la hauteur de l'arbre 2.3.4 initial, la hauteur de l'arbre bicolore est au plus égale à deux fois la hauteur de l'arbre 2.3.4 initial.

Remarque 2 : Un arbre bicolore associé à un arbre 2.3.4 de  $n$  éléments a une hauteur d'ordre logarithmique.

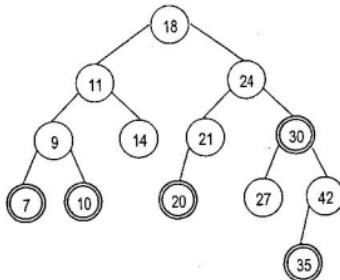


Figure 18. Arbre bicolore associé à l'arbre 2.3.4 de la Figure 7.

La figure 18 représente une possibilité d'arbre bicolore associé à celui de la figure 7. En effet, une arrivée de données différente (au vu d'éventuels rééquilibrages) donnerait un autre arbre.

### Recherche dans un arbre bicolore

En fait, indépendamment de la couleur des noeuds, c'est un arbre binaire et la recherche s'effectue à l'aide des algorithmes de recherche sur arbre binaire. Le nombre de comparaisons est donc logarithmique dans le pire des cas.

### Ajout dans un arbre bicolore

L'ajout d'un élément peut être réalisé en simulant l'ajout avec éclatement à la descente. L'éclatement d'un 4-noeud revient à inverser les couleurs des noeuds comme le montre la figure 19.

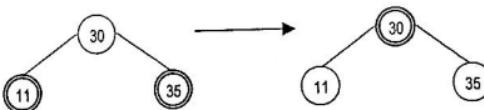


Figure 19. Simulation d'éclatement d'un 4-noeud en représentation bicolore.

Cette transformation peut malheureusement faire apparaître des noeuds rouges consécutifs dans l'arbre. Cela a pour conséquence de déséquilibrer l'arbre. Il faut donc le rééquilibrer, et pour cela nous allons utiliser les rotations vu précédemment.

Plusieurs cas peuvent se présenter :

- Le 4-nœud à éclater est attaché à un 2-nœud. Une simple inversion des couleurs suffira (Figure 20).

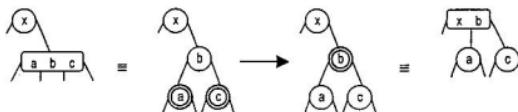


Figure 20.

- Le 4-nœud à éclater est attaché à un 3-nœud. Nous considérerons que le 3-nœud est *penché à droite*, il y alors 3 possibilités :

- Il est le premier fils du 3-nœud. Il suffit là encore d'inverser les couleurs (Figure 21).

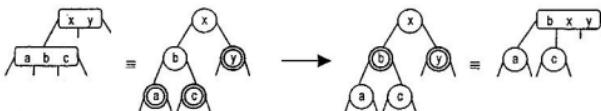


Figure 21.

- Il est le second fils du 3-nœud. L'inversion des couleurs entraîne un déséquilibre, il suffit alors d'effectuer une rotation droite-gauche pour rééquilibrer l'arbre (Figure 22).

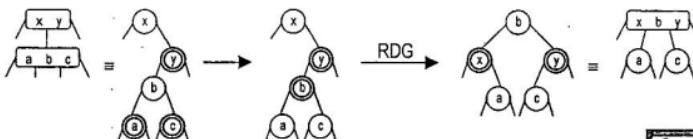


Figure 22.



- Il est le dernier fils du 3-nœud. L'inversion des couleurs entraîne un autre déséquilibre, il suffit alors d'effectuer une rotation gauche pour rééquilibrer l'arbre (Figure 23).

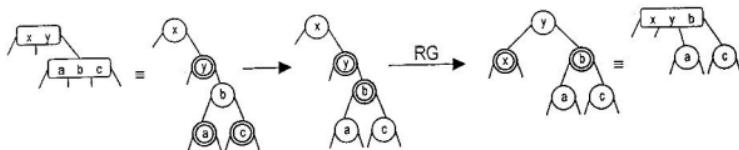


Figure 23.

Remarque : Il y a, bien évidemment, les trois cas symétriques lorsque le 3-nœud est penché à gauche.

### Algorithme

Le principe est simple, nous descendons à partir de la racine à la recherche de la feuille où doit se faire l'ajout du nouvel élément. Sur ce chemin, à chaque fois que l'on rencontre un nœud noir possédant deux fils rouges, nous inversons leur couleur (éclatement du nœud). Dans ce cas, il faut regarder si le père de ce nœud n'est pas rouge auquel cas il est nécessaire d'effectuer la rotation appropriée pour rééquilibrer l'arbre.

Une fois l'ajout de l'élément fait, il faut regarder si son père est rouge. Si c'est le cas, une rotation est là aussi nécessaire. En effet, les nouvelles feuilles sont systématiquement rouges dans la mesure où l'on ajoute un **élément jumeau**.

En terme d'implémentation, nous devons mémoriser dans chaque nœud la couleur de celui-ci. Il est donc nécessaire de rajouter un champ Coul prenant les valeurs {Rouge, Noir} dans l'enregistrement de type **T\_Noeud** défini dans le chapitre 3. Ce qui donne :

```

Type
Element = ...
Couleur = (Rouge, Noir)
T_Bic = ↑ T_Noeud /* type pointeur sur T_Noeud */

T_Noeud = Enregistrement
    Couleur Coul
    Element Elt
    T_Bic Fg,fd
Fin Enregistrement T_Noeud

Variable
    T_Bic B

```

Remarque : Nous allons rester dans un formalisme abstrait, et pour cela nous utiliserons le type **Bic** qui est un dérivé du type **ArbreBinaire** auquel nous avons ajouter les opérations propres aux **Arbres Bicolores**, comme **couleur** nous permettant d'extraire la couleur d'un nœud.

Pour simplifier l'algorithme, la procédure d'ajout utilise un arbre bicolore avec trois éléments supplémentaires ; une tête T, une sentinelle Z et un élément Q pointé par la sentinelle, comme le montre la Figure 24.

La tête T contient un élément dont la valeur est strictement inférieure à toutes celles possibles, sa couleur est le noir, son fils gauche pointe sur la sentinelle Z et son fils droit sur le nœud racine de l'arbre s'il existe. L'intérêt de son utilisation est de supprimer le cas particulier d'un ajout sur arbre vide.

La sentinelle Z est classique. Tous les liens de l'arbre devant pointer vers un arbre vide pointe sur elle. Au début, on lui affecte la valeur à ajouter, ce qui permet de ne pas avoir à tester deux possibilités de sortie de la recherche (arbre vide et l'élément existe déjà). Sa couleur est le noir.

L'élément Q, quant à lui, permet de n'avoir qu'un seul traitement, que l'inversion de couleurs (éclatement) se fasse au milieu de l'arbre où sur la feuille que l'on vient d'ajouter. Sa couleur est le rouge (je ne me permettrai pas de mauvais jeu de mot) et ses deux fils pointent sur un arbre vide.

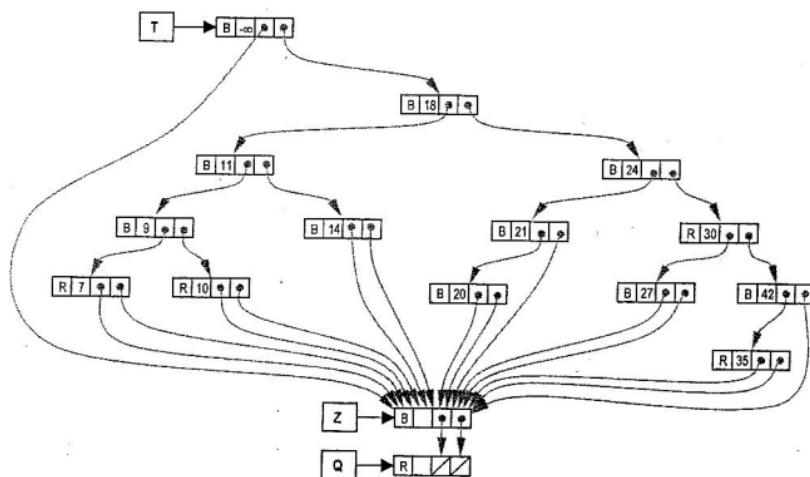


Figure 24. Représentation de l'arbre bicolore de la Figure 18.

(Procédure AjouterBic)

Algorithme Procédure AjouterBic

Paramètres globaux

Bic T, A

Booleen B

Paramètres locaux

Element x

Bic Z, Q

Variables

Bic P, GP, AGP /\* Mémorisation des trois ascendants de A \*/

Debut

A  $\leftarrow$  T

Contenu(racine(Z))  $\leftarrow$  x

B  $\leftarrow$  Faux

P  $\leftarrow$  T

GP  $\leftarrow$  T

Faire

AGP  $\leftarrow$  GP

GP  $\leftarrow$  P

P  $\leftarrow$  A

Si x < contenu(racine(A)) Alors

A  $\leftarrow$  g(A)

Sinon

A  $\leftarrow$  d(A)

Fin si

Si couleur(g(A))=Rouge et couleur(d(A))=Rouge Alors /\* 2 fils rouges \*/

Si A = Z Alors /\* Ajout \*/

B  $\leftarrow$  Vrai

A  $\leftarrow$  <Rouge,x,Z,Z> /\* Création du nœud, Coul=Rouge \*/

Si x < contenu(racine(P)) Alors

g(P)  $\leftarrow$  A

Sinon

d(P)  $\leftarrow$  A

Fin si

Sinon

Couleur(A)  $\leftarrow$  Rouge

Couleur(g(A))  $\leftarrow$  Noir

Couleur(d(A))  $\leftarrow$  Noir

Fin si

Si couleur(P)=Rouge Alors /\* Rééquilibrage \*/

Si Contenu(racine(P)) > contenu(racine(GP)) Alors

Si Contenu(racine(A)) > contenu(racine(P)) Alors

RG(GP)

Sinon

RDG(GP)

Fin si



```

Sinon
    Si Contenu(racine(A))>contenu(racine(P)) Alors
        RGD(GP)
    Sinon
        RD(GP)
    Fin si
Fin si
Si Contenu(racine(AGP))>contenu(racine(GP)) Alors
    g(AGP)←GP
Sinon
    d(AGP)←GP
Fin si
Couleur(GP)←Noir          /* Rétablissement des couleurs */
Couleur(g(GP))←Rouge
Couleur(d(GP))←Rouge
P←GP                         /* Rétablissement des hiérarchies */
GP←AGP
Si x=contenu(racine(P)) Alors
    A←P
Sinon
    Si x<contenu(racine(P)) Alors
        A←g(P)
    Sinon
        A←d(P)
    Fin si
Fin si
Fin si
Fin si
Tant que x<>contenu(racine(A))
    Couleur(d(T))←Noir           /* Racine toujours noire */
Fin Algorithme procedure AjouterBic.

```

## Conclusion

En ce qui concerne ces différents types d'arbres de recherche, la complexité des algorithmes de recherche, d'ajout et de suppression est toujours majorée, au pire, par une fonction logarithmique du nombre d'éléments. En ce qui concerne la complexité en moyenne, nous ne possédons, dans de nombreux cas, qu'une valeur expérimentale. Cela dit, il faut tenir compte du coût de réorganisation (éclatement, rotation) pour l'ajout et la suppression. Un des cas extrêmes étant la suppression d'un élément dans un AVL qui peut entraîner jusqu'à  $1.5 \log_2 n$  rotations.

# Bibliographie

Aho, Hopcroft & Ullmann, Data Structures and Algorithms, Addison-Wesley, 1983

Froidevaux, Gaudel & Soria, Types de données et algorithmes, McGraw-Hill, 1990

Dufourd, Bechmann & Bertrand, Spécifications algébriques, algorithmique et programmation, InterEditions, 1995

Courtin & Kowarski, Initiation à l'algorithmique et aux structures de données – récursivité et structures de données avancées, Dunod, 1995

Divay, Algorithmes et structures de données – Cours et exercices corrigés en langage C, Dunod, 1999

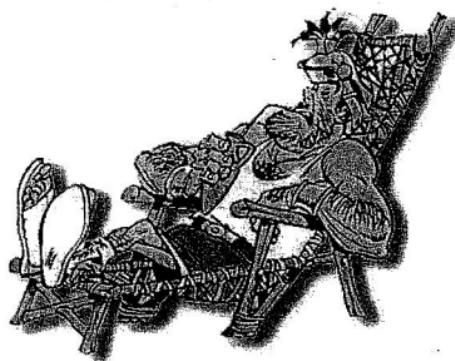
# Remerciements

A Junior et Francis pour m'avoir corrigé (pas physiquement, je vous rassure...),

à la machine à café (pour être rarement en panne),

à vous d'être arrivé jusqu'à cette page (uniquement si votre lecture fut séquentielle),

et aux quelques dessinateurs de bandes dessinées, parmi toutes celles que j'aime, qui m'ont permis d'égayer ce poly.



© Toutes les images qui ont permis d'illustrer ce poly sont la propriété de leurs auteurs et éditeurs. Si ces derniers ne souhaitaient pas que ces images y figurent, je les retirerais sur simple demande.