

# Algorithmique

**Année scolaire  
2020-2021**



**8 septembre 2020**

**E. Guérin - V.-M. Scuturici**

## TABLE DES MATIÈRES

<b>1 Introduction à l'algorithmique</b>	<b>5</b>
1.1 Motivation . . . . .	5
1.2 Algorithme : variables, expressions, affectations, enchaînement d'instructions . . . . .	5
1.3 Saisie et affichage de données . . . . .	11
1.4 Enchaînement d'instructions . . . . .	11
1.5 Enchaînement séquentiel . . . . .	11
1.6 Enchaînement alternatif . . . . .	12
1.7 Instructions répétitives . . . . .	18
1.8 Les tableaux . . . . .	25
1.9 Analyse d'un algorithme . . . . .	29
1.10 Procédures et fonctions . . . . .	35
1.11 Récursivité - Divide et impera . . . . .	42
1.12 Problèmes . . . . .	45
<b>2 Algorithmes de tri</b>	<b>46</b>
2.1 Tri par sélection . . . . .	46
2.2 Tri par insertion . . . . .	48
2.3 Tri à bulles (Bubble sort) . . . . .	49
2.4 Tri rapide (Quicksort) . . . . .	50
2.5 Rangs . . . . .	52
<b>3 Structures de données</b>	<b>53</b>
3.1 Structures séquentielles : des limitations . . . . .	53
3.2 Allocation dynamique . . . . .	53
3.3 Structures . . . . .	56
<b>4 Briques de base</b>	<b>60</b>
4.1 Le tableau statique . . . . .	60
4.2 Le tableau dynamique . . . . .	60
4.3 Liste simplement chaînée . . . . .	62
4.4 Liste doublement chaînée . . . . .	69
4.5 Arbre . . . . .	70

---

<b>5 Types de données abstraits</b>	<b>79</b>
5.1 Piles . . . . .	80
5.2 Files . . . . .	81
5.3 File double (deque) . . . . .	84
5.4 Listes . . . . .	85
5.5 Vecteur . . . . .	86
5.6 File de priorité . . . . .	87
5.7 Dictionnaires . . . . .	87
5.8 Arbre binaire de recherche . . . . .	88
5.9 Table de hachage . . . . .	93

## CHAPITRE 1

### INTRODUCTION À L'ALGORITHMIQUE

#### 1.1 Motivation

Le Département Informatique a introduit un cours d'algorithme/structures de données à partir de l'année universitaire 2014/2015. Nous souhaitons apporter les outils nécessaires pour programmer un ordinateur d'une manière efficace. Nous insistons sur la qualité d'un algorithme, un point très important dans la formation d'un ingénieur informaticien.

Ce document joue le rôle de support écrit pour ce cours, et il a vocation de s'enrichir dans le temps. Pour approfondir ce domaine, d'autres livres sont conseillés, principalement le support du cours enseigné à MIT ([Cormen] :*Introduction to algorithms*, Cormen, Leiserson, Rivest, Stein, MIT press, 2009).

Dans ce document nous apportons un volume important d'exemples détaillés, mais aussi des problèmes que nous proposons aux lecteurs. Les problèmes proposés ont un niveau de difficulté associé, entre 1 (très simple) et 10 (très difficile). Des riches collections de problèmes d'algorithme se retrouvent en accès libre sur le Web, comme par exemple UVJudge<sup>1</sup>, collection utilisée pour la préparation de concours internationaux d'algorithme.

#### 1.2 Algorithme : variables, expressions, affectations, enchaînement d'instructions

Nous définissons un algorithme comme une succession finie d'instructions permettant de transformer des entrées en sorties. Pratiquement, un algorithme spécifie précisément le lien entre les entrées et les sorties. Un algorithme représente la solution d'un problème donné, spécifié en langage naturel.

Les instructions utilisées par un algorithme doivent être comprises par un ordinateur. Pour éviter des contraintes techniques liées au génie logiciel nous utilisons un langage proche de la langue naturelle pour décrire un algorithme : le **pseudo-code**. Ce langage est suffisamment simple pour nous permettre de nous concentrer sur la résolution d'un problème, sans se soucier des détails d'implémentation d'un langage de programmation particulier (comme le C, C++, Python, Java etc.).

Par exemple, un algorithme qui permet de calculer la moyenne de deux nombres entiers  $a$  et  $b$  est décrit en Algorithme 1.

Nous remarquons dans cet exemple que la description de l'algorithme comporte deux parties : une description des entrées et des sorties (partie statique ou descriptive, la ligne 1), et une succession d'instructions permettant de faire le lien entre les entrées et les sorties (partie dynamique, description du comportement, les lignes 2-4). Les objets que

1. <http://uva.onlinejudge.org/>

**Algorithme 1 : La moyenne de deux nombres**

```

1 Procédure moyenne(a, b, resultat)
    Entrée   : entier a
               entier b
    Sortie    : réel resultat, correspondant à (a + b) / 2
2 début
3     resultat ← (a + b);
4     resultat ← resultat / 2;

```

nous manipulons sont de type *entier* (*a*, *b*) ou *réel* (*resultat*). Nous appelons ces objets des *variables*, et une variable est associée à une seule valeur à un moment donné.

### ► Variables : nom, type et valeur

Un algorithme manipule 3 catégories de données, plus généralement appelées variables : les paramètres en entrée, les paramètres en sortie et les variables locales ; ce qui peut être schématisé de la façon suivante :

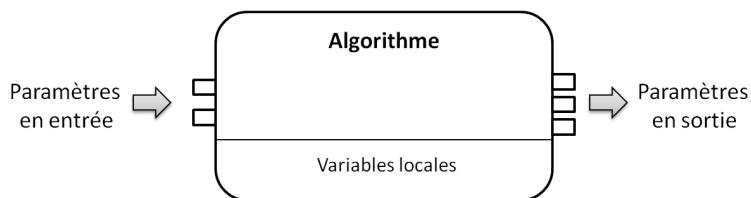


FIGURE 1.1 – Algorithme : relation entre entrées et sorties

Chaque variable (paramètre en entrée, paramètre en sortie ou variable locale) peut être vue comme une boîte contenant une information. Elle est caractérisée par un nom, permettant de référencer la boîte, un type, permettant de caractériser la nature des informations que l'on pourra déposer dans la boîte, et une valeur, désignant l'information effectivement contenue dans la boîte.

Le **nom** d'une variable, aussi appelé identificateur, est la représentation symbolique de l'adresse où est stockée la valeur de la variable dans la mémoire. En effet, les données sont conservées dans la mémoire de l'ordinateur. Cette mémoire est constituée d'une suite de bits (informations binaires) regroupés par groupe de 8 en octets. Chaque octet a une adresse permettant de le retrouver. Au lieu de désigner une variable par son adresse en mémoire, on donne un nom symbolique à cette adresse. Syntaxiquement, le nom d'une variable est une suite de caractères alpha-numériques, commençant par une lettre (majuscule ou minuscule). Le nom de la variable doit être choisi judicieusement afin de donner une indication sur la nature et le rôle de la donnée. Ainsi, si la variable correspond au total d'une facture, il est préférable de l'appeler *total-facture* plutôt que *toto* ou *X23*.

Toutes les données sont stockées en mémoire sous la forme d'une suite de 0 et de 1. En fonction du **type** (de la nature) de la donnée, cette suite de 0 et de 1 sera interprétée (décodée) différemment. Par exemple,

- un entier relatif peut être codé sur 2 octets consécutifs (soit 16 bits) de telle sorte que le bit le plus à gauche indique le signe de l'entier (positif si le bit est à 0, négatif sinon), tandis que les autres bits correspondent à l'entier codé en base 2 ;
- un nombre flottant peut être codé sur 4 octets consécutifs de telle sorte que l'octet le plus à gauche représente l'exposant (en base 2) et les 3 octets suivants la mantisse (en base 2) ;
- un caractère est codé sur un octet par son code ASCII en base 2 ;
- etc, ...

Pour connaître la valeur d'une variable, il est alors indispensable de connaître au préalable son type pour savoir sur combien d'octets elle est codée et comment interpréter correctement la suite correspondante de 0 et de 1.

Ainsi, il faut déclarer le type de chaque variable (paramètres et variables locales) avant de l'utiliser. A la suite de cette déclaration, on ne pourra mettre dans la variable que des informations appartenant au type déclaré.

La déclaration d'une donnée s'effectuera en faisant suivre le type de la variable de son nom. On utilisera dans la suite les types de données suivants :

- Les types de données *élémentaires* :
  - le type entier, désignant l'ensemble des entiers relatifs ;  
En pratique, le nombre d'entiers que l'on peut représenter est limité par le nombre d'octets utilisés pour les représenter, et les entiers trop grands (en valeur absolue) doivent être codés différemment.
  - le type réel, désignant l'ensemble des réels ;  
Là encore, il n'est pas possible de coder l'ensemble de tous les réels. En pratique, les réels sont approximés par des nombres flottants (ce qui peut provoquer des erreurs numériques) ;
  - le type car, désignant l'ensemble des caractères du clavier (caractères alpha-numériques et caractères spéciaux) ;
  - le type logique (aussi appelé booléen), désignant les deux valeurs vrai et faux ;
  - le type pointeur, désignant l'ensemble des adresses mémoires (ce type sera étudié au deuxième semestre) ;
  - le type texte, aussi appelé chaîne, désignant l'ensemble des chaînes de caractères<sup>2</sup>
- Les types de données *composés* :
  - le type tableau, désignant une suite indexée comportant un nombre variable mais borné de données de même type ;
  - le type structure, désignant une suite comportant un nombre fixé de données de types différents.

La **valeur** d'une variable est contenue dans l'emplacement mémoire se trouvant à l'adresse représentée par le nom de la variable. Ce contenu (une suite de 0 et de 1) doit être interprété correctement en fonction du type de la variable. Durant l'exécution d'un algorithme, une même variable peut avoir des valeurs différentes en temps.

Attention : à la suite de sa déclaration, la valeur d'une variable n'est pas définie. Dans nos exemples, nous utilisons le caractère '?' pour représenter cette valeur.

**Données constantes :** Certaines variables ont une valeur constante, qui ne change pas pendant toute l'exécution de l'algorithme. Ces variables sont appelées *constantes*. Elles sont déclarées au début de l'algorithme en faisant suivre le nom de la variable par sa valeur.

## ► Description des variables dans un algorithme

Par la suite, la partie statique d'un algorithme, spécifiant les paramètres et déclarant les variables, sera décrite selon le formalisme suivant :

Par exemple, l'algorithme calculant les racines d'une équation du second degré peut être spécifié de la façon décrite en algorithme 3.

Pour abréger, on pourra regrouper plusieurs déclarations de paramètres sur une même ligne lorsqu'ils ont le même mode de passage (entrée ou sortie) et le même type (algorithme 4).

## ► L'affectation

Pour résoudre un problème, un algorithme décrit la suite d'instructions à effectuer pour calculer les valeurs des paramètres en sortie à partir des valeurs des paramètres en entrée. Il existe 2 instructions différentes : l'affectation, qui

2. Dans de nombreux langages, le type texte est représenté par un tableau de caractères.

**Algorithme 2 : Structure d'un algorithme****1 Procédure** nom-de-la-procédure(<noms-des-paramètres>)

**Entrée** : pour chaque paramètre en entrée, préciser son type et son nom

**Sortie** : pour chaque paramètre en sortie, préciser son type et son nom

**Précondition** : Conditions sur les paramètres en entrée

**Postcondition** : Relation entre les paramètres en entrée et ceux en sortie

**Déclaration** : pour chaque variable locale, préciser son type et son nom

**const** : pour chaque constante, préciser son nom et sa valeur

**début**

    Suite d'opérations élémentaires permettant de calculer les paramètres en sortie  
    en fonction des paramètres en entrée  
    (partie dynamique de l'algorithme)

**Algorithme 3 : Racines d'une équation de deuxième degré****1 Procédure** racines( $a, b, c, r_1, r_2$ )

**Entrée** : réel  $a$

    réel  $b$

    réel  $c$

**Sortie** : réel  $r_1$

    réel  $r_2$

**Précondition** :  $b^2 - 4ac \geq 0$  et  $a \neq 0$

**Postcondition** :  $ar_1^2 + br_1 + c = ar_2^2 + br_2 + c = 0$  ou autrement dit,  
 $r_1$  et  $r_2$  sont les deux solutions de  
l'équation  $ax^2 + bx + c = 0$

**Déclaration** : réel  $\delta$

**début**

    Suite d'opérations élémentaires  
    permettant de calculer  $r_1$  et  $r_2$  à partir de  $a, b$  et  $c$ .

**Algorithme 4 : Racines d'une équation de deuxième degré - forme abrégée****1 Procédure** racines( $a, b, c, r_1, r_2$ )

**Entrée** : réel  $a, b, c$

**Sortie** : réel  $r_1, r_2$

permet de changer la valeur d'une variable, et l'appel de procédure, qui permet d'appeler (d'utiliser) un algorithme dans un autre. On étudie ici l'affectation ; l'appel de procédure sera étudié plus tard.

L'affectation permet de changer la valeur d'une variable, autrement dit de modifier le contenu de la mémoire à l'adresse symbolisée par le nom de la variable. Syntaxiquement, une affectation sera représentée par :

nom-var  $\leftarrow$  expr

et a pour signification : la variable de nom nom-var prend pour valeur la valeur de l'expression expr, autrement dit, la valeur de l'expression expr est stockée dans la mémoire à l'adresse symbolisée par le nom nom-var. La valeur de l'expression expr doit appartenir au type déclaré pour la variable nom-var.

## ► Définition d'une expression

L'expression affectée à une variable peut être une valeur explicite :

par exemple, l'instruction  $a \leftarrow 25$  affecte la valeur 25 à la variable de nom  $a$ .

L'expression affectée à une variable peut également être la valeur contenue dans une autre variable :

par exemple, l'instruction  $a \leftarrow b$  affecte à la variable de nom  $a$  la valeur contenue dans la variable de nom  $b$ .

Enfin, l'expression affectée à une variable peut être le résultat d'une opération entre d'autres expressions. Il existe 3 types d'opérations : les opérations arithmétiques, les opérations de comparaison et les opérations logiques.

### Opérations arithmétiques

**Les opérations arithmétiques binaires** sont : l'addition, notée "+", la soustraction, notée "-", la multiplication, notée "\*", la division réelle, notée "/", la division entière notée "div" et le modulo<sup>3</sup>, noté "mod". Ces opérations prennent en argument deux expressions numériques et rendent la valeur numérique correspondant à l'application de l'opération sur les valeurs des expressions en argument.

Ces opérations sont définies sur les réels et sur les entiers, sauf "div" et "mod" qui ne sont définies que sur les entiers. Enfin, les opérateurs de multiplication, de division et de modulo sont plus prioritaires que (et donc évalués avant) les opérateurs d'addition et de soustraction. Une expression arithmétique peut être parenthésée pour spécifier l'ordre de son évaluation.

**Les opérations arithmétiques unaires** sont le plus unaire, noté "+", et le moins unaire, noté "-". Ces opérations prennent en argument une expression numérique et rendent une valeur numérique.

### Opérations de comparaison

Une opération de comparaison prend en argument deux expressions de même type élémentaire (2 entiers, 2 réels, 2 caractères ou 2 chaînes de caractères), et rend une valeur logique (vrai ou faux). On utilisera les opérations de comparaison suivantes :  $<$ ,  $\leq$ ,  $=$ ,  $\neq$ ,  $\geq$  et  $>$ .

Les entiers et les réels sont comparés selon l'ordre numérique usuel. Les caractères sont comparés selon l'ordre défini par le code ASCII : chaque caractère est codé par un entier compris entre 0 et 255, appelé code ASCII, et pour comparer deux caractères, on compare leur code ASCII. Dans ce code, les chiffres sont inférieurs aux caractères majuscules qui sont eux-mêmes inférieurs au caractères minuscules.

Les chaînes de caractères sont comparées selon l'ordre lexicographique : un texte  $t_1$  est inférieur à un texte  $t_2$  s'il existe un caractère  $c$  de  $t_1$  tel que

- pour chaque caractère se trouvant avant  $c$  dans  $t_1$ , le caractère correspondant de  $t_2$  soit identique,
- le caractère correspondant à  $c$  dans  $t_2$  soit supérieur à  $c$ .

Par exemple, "aababcd" < "aax" et "10" < "9".

### Opérations logiques

**Les opérations logiques binaires** sont la disjonction, notée ou, et la conjonction, notée et. Ces opérateurs prennent en argument 2 expressions de valeur logique et rendent une valeur logique. La signification des opérateurs logiques binaires est donnée par la table 1.1.

3. On rappelle que le modulo est le reste de la division entière. Autrement dit,  $x \bmod y = x - y * (x \div y)$  Par exemple,  $23 \bmod 7 = 2$  car  $23 \div 7 = 3$  et  $23 - 3 * 7 = 2$ .

A	B	A et B	A ou B
vrai	vrai	vrai	vrai
vrai	faux	faux	vrai
faux	vrai	faux	vrai
faux	faux	faux	faux

TABLE 1.1 – Les opérateurs logiques binaires et et ou

A	non A
vrai	faux
faux	vrai

TABLE 1.2 – Les opérateurs logiques unaires et et ou

**L'opération logique unaire** est la négation, notée `non`. Cet opérateur prend en argument une expression de valeur logique et rend une valeur logique, selon la table 1.2.

### Exemples

Soient les variables suivantes :

entier  $e_1, e_2$ ,  
 réel  $r_1$ ,  
 logique  $b_1, b_2$

Après l'exécution de l'instruction

$$e_1 \leftarrow (10 \bmod 3) + (5 \bmod 2)$$

la variable  $e_1$  a pour valeur  $1 + 2 = 3$ ;

après l'exécution de l'instruction

$$e_2 \leftarrow e_1 * 2$$

la variable  $e_2$  a pour valeur  $3 * 2 = 6$ ;

après l'exécution de l'instruction

$$r_1 \leftarrow 4.5 * 2.0$$

la variable  $r_1$  a pour valeur 9.0;

après l'exécution de l'instruction

$$r_1 \leftarrow r_1 / 3.0$$

la variable  $r_1$  a pour valeur  $9.0 / 3.0 = 3.0$ ;

après l'exécution de l'instruction

$$b_1 \leftarrow (r_1 > 4.2) \text{ ou } (e_1 = e_2)$$

la variable  $b_1$  a pour valeur *faux*;

après l'exécution de l'instruction

$$b_2 \leftarrow \text{non}(b_1) \text{ et } r_1 \leq 4.2$$

la variable  $b_2$  a pour valeur *vrai*.

## 1.3 Saisie et affichage de données

Un algorithme calcule les valeurs des paramètres en sortie, en fonction des valeurs des paramètres en entrée. Les valeurs des paramètres en entrée sont “données” à l’algorithme, tandis que les valeurs des paramètres en sortie sont fournies comme résultat de l’algorithme à celui qui l’exécutera. Ainsi, on ne s’occupe pas *a priori* de la saisie des valeurs des paramètres en entrée ni de l’affichage des valeurs calculées pour les paramètres en sortie. De fait, il est possible que les valeurs calculées par un algorithme ne soient pas affichées, mais par exemple qu’elles soient fournies comme valeurs d’entrée d’un autre algorithme.

Dans certains cas, on peut cependant souhaiter saisir au clavier une valeur pour l’affecter à une variable. Pour cela, on pourra utiliser l’instruction “saisir( $x$ )” qui lit au clavier une valeur de même type que la variable  $x$  et affecte cette valeur à  $x$ . De même, pour afficher à l’écran la valeur d’une variable  $x$ , on pourra utiliser l’instruction “afficher( $x$ )”.

## 1.4 Enchaînement d’instructions

Les instructions d’un algorithme sont enchainées selon un ordre déterminé. Il existe trois façons différentes d’enchaîner des instructions : en séquence, de façon alternative ou répétitive.

## 1.5 Enchaînement séquentiel

L’enchaînement séquentiel d’une suite d’instructions permet d’exécuter les instructions les unes à la suite des autres. Syntaxiquement, les instructions seront notées les unes en dessous des autres (une instruction par ligne). Dans le cas d’instructions courtes, on pourra noter plusieurs instructions les unes à côté des autres, séparées par un point virgule. Dans ce cas, les instructions sont tout naturellement exécutées de la gauche vers la droite.

Quelques exemples : l’algorithme 5 et l’algorithme 6.

---

### Algorithme 5 : Calcul du diamètre, du périmètre et de la surface d’un cercle à partir de son rayon

---

```

1 Procédure cercle( $R, D, P, S$ )
    Entrée      : réel  $R$ 
    Sortie       : réel  $D, P, S$ 
    Précondition :  $R \geq 0$ 
    Postcondition :  $D, P$  et  $S$  contiennent respectivement le diamètre,
                    le périmètre et la surface d’un cercle de rayon  $R$ 
    Déclaration   : const : pi = 3.14
2   début
3      $D \leftarrow 2 * R$ 
4      $P \leftarrow D * pi$ 
5      $S \leftarrow R * R * pi$ 

```

---

La partie statique de l’algorithme (instruction 1) ne correspond pas à une instruction exécutable. Dans nos exemples la première ligne de la partie dynamique de l’algorithme (*début*, ligne 2) n’est pas une instruction exécutable, elle est utilisée pour délimiter un bloc d’instructions. Plusieurs instructions font partie d’un même bloc si ces instructions ont le même niveau d’indentation. Un bloc est considéré à son tour comme une instruction, ce qui nous permet d’obtenir des agrégations assez complexes, comme par exemple en Algorithme 7.

L’algorithme 5 va se terminer après l’exécution de 3 instructions (les instructions 3, 4 et 5) et l’algorithme 6 va se terminer après l’exécution de 2 instructions.

**Algorithme 6 : Calcul des coefficients d'une droite à partir de deux points**

```

1 Procédure droite( $x_1, y_1, x_2, y_2$ )
Entrée      : réel  $x_1, y_1, x_2, y_2$ 
Sortie       : réel  $a, b$ 
Précondition : Les 2 points de coordonnées  $(x_1, y_1)$  et  $(x_2, y_2)$  sont distincts, et ne sont pas alignés
                  verticalement. Autrement dit,  $x_1 \neq x_2$ 
Postcondition :  $a$  et  $b$  sont les coefficients de la droite passant par les 2 points de coordonnées  $(x_1, y_1)$  et
                   $(x_2, y_2)$ . Autrement dit,  $y_1 = ax_1 + b$  et  $y_2 = ax_2 + b$ 

2 début
3    $a \leftarrow (y_1 - y_2)/(x_1 - x_2)$ 
4    $b \leftarrow y_1 - a * x_1$ 

```

**Algorithme 7 : Instructions organisées en blocs**

**Déclaration** : entier  $a$   
                  entier  $b$

```

1 début
2    $a \leftarrow 1;$ 
3    $b \leftarrow 1;$ 
4   début
5      $a \leftarrow a + b;$ 
6      $b \leftarrow a + b;$ 
7   début
8      $a \leftarrow b * 10;$ 
9      $b \leftarrow a * 10;$ 
10    début
11       $a \leftarrow a * 10;$ 
12       $b \leftarrow b * 10;$ 

```

## 1.6 Enchaînement alternatif

L'instruction conditionnelle *si* (ou l'enchaînement alternatif) est une autre instruction en pseudo-code. L'enchaînement alternatif permet d'exécuter alternativement une première suite d'instructions, si une certaine condition est vérifiée, ou bien une autre série d'instructions, si la condition n'est pas vérifiée. La syntaxe générale de l'instruction *si* est donnée dans la figure Algorithme 8.

**Algorithme 8 : L'instruction *si***

```

1 début
2   si condition alors
3        bloc du alors
4   sinon
5        bloc du sinon

```

Cette instruction permet d'exécuter le bloc d'instructions *bloc du alors* si *condition* est vrai, et le bloc *bloc du sinon* en cas contraire. Cette instruction propose une alternative à un enchainement séquentiel, en permettant d'exécuter

seulement une de deux instructions, en fonction de la valeur d'un prédicat.

Par exemple, un algorithme qui permet de calculer le minimum entre deux nombres entiers  $a$  et  $b$  est décrit en Algorithme 9.

---

**Algorithme 9 : Le minimum de deux nombres**


---

**1 Procédure**  $\min(a, b, resultat)$

<b>Entrée</b>	: entier $a$
	entier $b$
<b>Sortie</b>	: entier $resultat$

**début**

<b>si</b> $a < b$ <b>alors</b>	<b>sinon</b>
$resultat \leftarrow a$	$resultat \leftarrow b$

2

3      **si**  $a < b$  **alors**

4          |  $resultat \leftarrow a$

5      **sinon**

6          |  $resultat \leftarrow b$

Si nous exécutons l'algorithme 9 sur le cas concret  $\langle a=10, b=12 \rangle$ , une description de son exécution est décrite dans le tableau 1.3.

Ligne	<i>a</i>	<i>b</i>	<i>resultat</i>
2	10	12	?
3	10	12	?
4	10	12	10

TABLE 1.3 – Valeurs de variables à l'exécution de l'algorithme 9 pour  $\langle a=10, b=12 \rangle$

Dans ce tableau nous affichons en ordre les instructions effectuées par une exécution de l'algorithme. Chaque instruction est identifiée avec le numéro de la ligne correspondante. Nous affichons aussi les valeurs de différents variables. Ces valeurs correspondent à l'instant d'après l'exécution de l'instruction, et la valeur " ?" correspond au fait que la variable n'a pas de valeur précisément définie. Même si l'instruction *début* n'est pas exécutable, nous la représentons dans ce tableau pour montrer les valeurs initiales des différents variables.

Une exécution du même algorithme pour les entrées  $\langle a=10, b=-1 \rangle$  va produire la trace décrite dans le tableau 1.4.

Ligne	<i>a</i>	<i>b</i>	<i>resultat</i>
2	10	-1	?
3	10	-1	?
5	10	-1	-1

TABLE 1.4 – Valeurs de variables à l'exécution de l'algorithme 9 pour  $\langle a=10, b=-1 \rangle$

Une version simplifiée de l'instruction *si* est décrite dans l'algorithme 10.

Cet énoncé est interprété de la façon suivante :

"si l'expression logique condition est évaluée à vrai, alors la suite d'instructions bloc du alors est exécutée, sinon (si l'expression logique condition est évaluée à faux), on ne fait rien."

Enfin, dans le cas où l'on a plus de deux alternatives dont toutes les conditions sont exclusives, on pourra utiliser l'énoncé décrit en algorithme 11.

Cet énoncé est équivalent à l'énoncé décrit en algorithme 12, imbriquant 4 énoncés alternatifs.

**Algorithme 10 : L'instruction *si* - version simplifiée**

```

1 début
2   | si condition alors
3     |   bloc du alors

```

**Algorithme 11 : L'instruction *si* - version avec plusieurs alternatives**

```

1 si condition1 alors
2   |   suite-1
3 sinon si condition2 alors
4   |   suite-2
5 sinon si condition3 alors
6   |   suite-3
7 sinon si condition4 alors
8   |   suite-4
9 sinon
10  |   suite-5

```

**Algorithme 12 : L'instruction *si* - version équivalente avec l'algorithme 11**

```

1 si condition1 alors
2   |   suite-1
3 sinon
4   |   si condition2 alors
5     |     suite-2
6   |   sinon
7     |     si condition3 alors
8       |       suite-3
9     |     sinon
10    |       si condition4 alors
11      |         suite-4
12    |       sinon
13      |         suite-5

```

## Exemples

Quelques exemples d'utilisation des instructions d'enchaînement alternatif : la recherche du nombre de solutions d'une équation du second degré quelconque (13) et le minimum de trois nombres (algorithmes 14 et 15).

---

### **Algorithm 13 : Recherche du nombre de solutions d'une équation du second degré quelconque**

---

1 **Procédure** *nb-solutions*(*a, b, c, nbSol*)

**Entrée**           : réel *a, b, c*

**Sortie**           : entier *nbSol*

**Postcondition** : *nbSol* = nombre de solutions sur les réels de l'équation  $ax^2 + bx + c = 0$

**Déclaration**   : réel *delta*

2     **début**

3         **si** *a* = 0 **alors**

4             **si** *b* = 0 **alors**

5                 **si** *c* = 0 **alors**

6                     *nbSol*  $\leftarrow \infty$

7                 **sinon**

8                     *nbSol*  $\leftarrow 0$

9                 **sinon**

10                     *nbSol*  $\leftarrow 1$

11         **sinon**

12             *delta*  $\leftarrow b * b - 4 * a * c$

13             **si** *delta* < 0 **alors**

14                     *nbSol*  $\leftarrow 0$

15             **sinon si** *delta* = 0 **alors**

16                     *nbSol*  $\leftarrow 1$

17             **sinon**

18                     *nbSol*  $\leftarrow 2$

Dans tous les cas, l'algorithme 14 fera 2 comparaisons et 1 affectation. Un autre algorithme pour rechercher le plus petit de trois nombres est l'algorithme 15.

Dans tous les cas, l'algorithme 15 fera 2 comparaisons. En plus, dans le meilleur des cas, il fera encore 1 affectation, et dans le pire des cas il fera encore 2 affectations.

**Algorithme 14 :** Recherche du plus petit nombre parmi trois nombres

```

1 Procédure plus-petit(a, b, c, pp)
2   Entrée      : entier a, b, c
3   Sortie       : entier pp
4   Postcondition : pp = plus petit nombre de l'ensemble {a, b, c}
5   début
6     si a < b alors
7       // b n'est pas le plus petit  $\Rightarrow$  recherche du plus petit entre a et c
8       si a < c alors
9         pp  $\leftarrow$  a
10      sinon
11        pp  $\leftarrow$  c
12
13      sinon
14        // a n'est pas le plus petit  $\Rightarrow$  recherche du plus petit entre b et c
15        si b < c alors
16          pp  $\leftarrow$  b
17        sinon
18          pp  $\leftarrow$  c

```

**Algorithme 15 :** Recherche du plus petit nombre parmi trois nombres

```

1 Procédure plus-petit(a, b, c, pp)
2   ...
3   début
4     si a < b alors
5       pp  $\leftarrow$  a
6     sinon
7       pp  $\leftarrow$  b
8     // pp = plus petit nombre entre a et b
9     si c < pp alors
10       pp  $\leftarrow$  c

```

## ► Problèmes

1. (Difficulté=1) Donner un algorithme qui permet de calculer le maximum entre deux nombres entiers.
2. (Difficulté=1) Donner un algorithme qui permet de calculer la moyenne de deux nombres réels.
3. (Difficulté=2) Donner un algorithme qui permet de calculer l'aire de l'intersection de deux intervalles  $[a_1, b_1]$  et  $[a_2, b_2]$ .  $a_1, b_1, a_2, b_2$  sont des nombres réels, et le résultat est un réel.
4. (Difficulté=2) Pour un triangle défini par les coordonnées  $\{x_1, y_1\}, \{x_2, y_2\}, \{x_3, y_3\}$ , proposez un algorithme qui permet de savoir s'il est équilatéral.
5. (Difficulté=2) Proposez un algorithme qui vérifie si un point  $x_0, y_0$  se trouve à l'intérieur d'un cercle défini par son origine  $x_c, y_c$  et un rayon  $R$ .
6. (Difficulté=2) Proposez un algorithme qui donne les solutions réelles de l'équation  $a * x^2 + b * x + c = 0$ . Les sorties attendues : un boolean qui précise si l'équation admette des solutions réelles (*valide*) et les deux solutions  $(x_1, x_2)$ .
7. (Difficulté=1) Soit l'algorithme décrit dans Algorithme 16. Complétez le tableau 1.5 avec les valeurs de différents variables pendant l'exécution de l'algorithme.

### Algorithme 16 : Problème : Succession d'affectations et différents opérations

```

1 début
2   Déclaration : entier s
3     entier p
4     entier i
5
6   s ← 0;
7   p ← 1;
8   i ← 1;
9   début
10    s ← s + i;
11    p ← p * i;
12    i ← i + 1;
13    début
14      s ← s + i;
15      p ← p * i;

```

Ligne	s	p	i
1	?	?	?
2	0	?	?
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			

TABLE 1.5 – Valeurs de variables à l'exécution de l'algorithme 16

## 1.7 Instructions répétitives

Une catégorie spéciale d'instructions, très utile en pratique, est constituée des instructions répétitives (dénommées aussi des *boucles*). L'enchaînement répétitif permet d'exécuter plusieurs fois une même suite d'instructions ; le nombre de fois où la suite d'instructions est exécutée étant déterminé par une condition logique. Les trois catégories de boucles que nous utilisons sont décrites dans Algorithme 17 : *pour*, *tant que* et *répéter jusqu'à*.

---

### Algorithme 17 : Les boucles

---

```

1 début
2   pour (init; cond; passage) faire
3     bloc de la boucle
4   tant que (condition) faire
5     bloc de la boucle
6   répéter
7     bloc de la boucle
8   jusqu'à (condition d'arrêt);

```

---

L'instruction *tant que* a la signification suivante : "tant que l'expression logique *condition* est évaluée à *vrai*, exécuter les instructions *bloc de la boucle* et recommencer ; ce processus itératif s'arrête quand *condition* est évaluée à *faux*".

Par exemple, si nous souhaitons calculer  $x^n$ , l'algorithme 18 utilise l'instruction *tant que*.

Une exécution de cet algorithme pour les entrées  $<n = 3, x = 2>$  va produire la trace décrite dans la table 1.6.

On peut imaginer d'autres algorithmes pour calculer  $x$  à la puissance  $n$ . On aurait par exemple pu compter de 1 à  $n$  (algorithme 19) ou encore de  $n$  à 1 (algorithme 20).

**Algorithme 18 :** Algorithme pour calculer  $x^n$

1 **Procédure** puissance( $n, x, p$ )

**Entrée**           : entier  $n$   
                      réel  $x$

**Sortie**          : réel  $p$

**Précondition** :  $n \geq 0$

**Postcondition** :  $p = x^n$

**Déclaration**   : entier  $cpt$

2     **début**

3         $p \leftarrow 1$

4         $cpt \leftarrow 0$

5        **tant que**  $cpt < n$  **faire**

          // invariant :  $p = x^{cpt}$

$p \leftarrow p * x$

$cpt \leftarrow cpt + 1$

    // Nombre de passages dans la boucle =  $n$  ;  $cpt = n$  et  $p = x^{cpt} = x^n$

Ligne	$n$	$x$	$p$	$cpt$
2	4	2	?	?
3	4	2	1	?
4	4	2	1	0
6	4	2	2	0
7	4	2	2	1
6	4	2	4	1
7	4	2	4	2
6	4	2	8	2
7	4	2	8	3

TABLE 1.6 – Trace de l'exécution de l'algorithme 18

**Algorithme 19 :** Algorithme alternatif pour calculer  $x^n$

1 **Procédure** puissance( $n, x, p$ )

2     ...

**Déclaration**   : entier  $cpt$

**début**

$p \leftarrow 1$

$cpt \leftarrow 1$

**tant que**  $cpt \leq n$  **faire**

          // invariant :  $p = x^{cpt-1}$

$p \leftarrow p * x$

$cpt \leftarrow cpt + 1$

    // Nombre de passages dans la boucle =  $n$  ;  $cpt = n + 1$  et  $p = x^{cpt-1} = x^n$

**Algorithme 20 :** Algorithme alternatif pour calculer  $x^n$ 1 Procédure puissance( $n, x, p$ )

2 ...

```

Déclaration : entier cpt
début
  p ← 1
  cpt ← n
  tant que cpt > 0 faire
    // invariant : p =  $x^{n-cpt}$ 
    p ← p * x
    cpt ← cpt - 1
  // Nombre de passages dans la boucle = n ; cpt = 0 et p =  $x^{n-cpt} = x^n$ 

```

**Quelques conseils au sujet des enchaînements répétitifs**

Quand, pour résoudre un problème, on se rend compte que l'on va avoir besoin d'un énoncé répétitif, on cherche à identifier les "blocs" suivants :

- Instructions d'initialisation : ce sont les instructions qui permettent d'initialiser les variables sur lesquelles on va travailler dans l'énoncé répétitif. Dans l'exemple précédent du calcul de  $x$  à la puissance  $n$ , ce sont les deux instructions qui initialisent  $p$  et  $cpt$ .
- Condition d'arrêt : c'est la condition qui doit être satisfaite pour arrêter de boucler. Dans l'exemple précédent, on doit arrêter de boucler lorsqu'on a exécuté  $n$  fois les instructions à répéter. Notons que la condition mise derrière le mot clé **tant que** est la négation de la condition d'arrêt : on doit continuer tant qu'on n'a *pas encore* exécuté  $n$  fois les instructions à répéter. On prendra l'habitude de mettre en commentaire après la condition d'arrêt (en vérifiant qu'il s'agit bien de la négation de la condition mise derrière **tant que**).
- Instructions à répéter : ce sont les instructions qui sont exécutées à chaque passage dans la boucle. On distingera généralement deux sous blocs dans ce bloc d'instructions : les instructions "de traitement" et les instructions "de passage".
  - Les instructions "de traitement" sont celles à l'origine du fait que l'on écrit un enchaînement répétitif. Dans l'exemple précédent, il s'agit de l'instruction  $p ← p * x$ .
  - Les instructions "de passage" sont celles qui permettent de modifier les variables sur lesquelles porte la condition d'arrêt (les variables qui permettent de contrôler le nombre de passages dans la boucle). Dans l'exemple précédent, il s'agit de l'instruction  $cpt ← cpt + 1$ .

En général, on commence par identifier les instructions "de traitement". Ensuite, il s'agit d'écrire les instructions d'initialisation, la condition d'arrêt et les instructions de passage. Ces trois blocs dépendent les uns des autres, comme l'illustre l'exemple précédent (où l'on a proposé 3 algorithmes différents pour résoudre un même problème). Ils doivent donc être conçus en même temps. A ce moment, on se posera les questions suivantes :

- Est-on certain que la condition d'arrêt est atteinte ?
- Combien de fois les instructions à répéter sont-elles exécutées ?
- Quelles sont les valeurs des variables lorsqu'on sort de la boucle ?

**Exemple : calcul du produit des  $n$  premiers entiers positifs**

Il s'agit maintenant de calculer factorielle  $n$ , c'est-à-dire,  $1 * 2 * 3 * \dots * n$ . On peut spécifier ce problème de la façon suivante :

Pour cela, on se rend compte que l'on a besoin de faire  $n$  multiplications, mais contrairement à l'exemple précédent (puissance), le facteur multiplicatif change à chaque fois : il faut d'abord multiplier par 1, puis par 2, puis par 3, ... jusqu'à

1 **Procédure** *factorielle*(*n, f*)  
**Entrée** : entier *n*  
**Sortie** : entier *f*  
**Précondition** :  $n \geq 0$   
**Postcondition** :  $f = n!$  où  $n!$  est la fonction récursivement définie par :  
 $0! = 1$   
 $n! = n * (n - 1)!, \forall n \geq 1$   
ou, autrement dit,  $f = 1 * 2 * 3 * \dots * n$

*n*. On va donc d'abord écrire une boucle où une variable (par exemple *i*) va prendre successivement les valeurs 1, 2, 3, ... jusque *n*, soit :

```
1 i ← 1
2 tant que i ≤ n faire
3   i ← i + 1
// Nombre de passages dans la boucle = n ; i = n + 1
```

On peut ensuite ajouter les instructions permettant de multiplier une variable *f* par les valeurs successivement prises par *i*. On obtient l'algorithme suivant :

#### **Algorithme 21** : Calcul de factorielle *n*

1 **Procédure** *factorielle*(*n, f*)  
**Déclaration** : entier *i*  
**début**
 3 *f* ← 1
 4 *i* ← 1
 5 **tant que** *i* ≤ *n* **faire**
 6 // invariant :  $f = (i - 1)!$ 
 7 *f* ← *f* \* *i*
 8 *i* ← *i* + 1
 // Nombre de passages dans la boucle = *n* ; *i* = *n* + 1 et  $p = (i - 1)! = n!$

**Exemple d'exécution** pour *n* = 4 :

**Terminaison** : La terminaison de l'algorithme est assurée par le fait que la valeur de *i* augmente de 1 à chaque passage dans la boucle “tant que” et que l'on s'arrête quand elle devient supérieure ou égale à celle de *n*.

**Complexité** : Pour calculer *f* à partir de *n*, il faut d'abord effectuer 2 affectations, puis il faut répéter *n* fois la boucle “tant que”. A chaque passage dans la boucle, on effectue 1 test ( $i \leq n$ ), 1 addition, 1 multiplication, et 2 affectations. Au total, on effectuera donc  $2 * n + 2$  affectations, *n* additions, *n* multiplications et  $n + 1$  tests (le +1 est du au test effectué pour la sortie de la boucle, quand  $i = n + 1$ ).

nb de passages dans la boucle	0	1	2	3	4
valeur de $i$	1	2	3	4	5
valeur de $f$	1	1	2	6	24

TABLE 1.7 – Exemple d'exécution de l'algorithme 7, pour  $n = 4$ 

**Correction :** La correction de l'algorithme peut être démontrée à l'aide de la propriété invariante. En effet, à chaque passage dans la boucle, la propriété invariante  $f = (i - 1)!$  est vérifiée (on vérifie facilement qu'elle est vraie au premier passage, et que si elle est vraie à un passage, alors elle est encore vraie au passage suivant). On arrête de boucler lorsque la condition  $i \leq n$  n'est plus vérifiée, autrement dit lorsque  $i = n + 1$ . A ce moment, du fait de la propriété invariante, on sait que  $fact = (i - 1)! = n!$ .

### Les autres instructions répétitives : “pour” et “répéter ... jusqu'à”

Quand il y a une seule instruction d'initialisation et une seule instruction de passage *concernant une variable sur laquelle porte la condition d'arrêt*, alors on peut utiliser l'énoncé “pour” suivant :

---

```

1 pour (init; cond; passage) faire
2   traitement

```

---

Cet énoncé est équivalent à l'énoncé suivant :

---

```

1 init
2 tant que cond faire
3   traitement
4   passage

```

---

Par exemple, les 3 algorithmes permettant de calculer  $x$  à la puissance  $n$  peuvent être écrits :

---

#### Algorithme 22 : Algorithme pour calculer $x^n$

---

```

1 Procédure puissance( $n, x, p$ )
2   ...
3   début
4      $p \leftarrow 1$ 
5     pour ( $cpt \leftarrow 0 ; cpt < n ; cpt \leftarrow cpt + 1$ ) faire
6       // invariant :  $p = x^{cpt}$ 
        $p \leftarrow p * x$ 
    // Nombre de passages dans la boucle =  $n$  ;  $cpt = n$  et  $p = x^{cpt} = x^n$ 

```

---

L'intérêt d'un énoncé *pour* est de rendre plus lisible l'algorithme en regroupant sur une même ligne les instructions et la condition déterminant le nombre de passages dans une boucle : on peut, sans lire les instructions de traitement à répéter, savoir combien de fois sera exécutée la boucle... sous réserve que les instructions de traitement ne modifient pas des variables sur lesquelles porte la condition d'arrêt (dans ce cas, il est probablement plus lisible de ne pas utiliser une boucle *pour*).

**Algorithme 23 : Algorithme pour calculer  $x^n$**

```

1 Procédure puissance( $n, x, p$ )
2   ...
3   début
4      $p \leftarrow 1$ 
5     pour ( $cpt \leftarrow 1 ; cpt \leq n ; cpt \leftarrow cpt + 1$ ) faire
6       // invariant :  $p = x^{cpt-1}$ 
7        $p \leftarrow p * x$ 
8   // Nombre de passages dans la boucle = n ; cpt = n + 1 et p =  $x^{cpt-1} = x^n$ 

```

**Algorithme 24 : Algorithme pour calculer  $x^n$**

```

1 Procédure puissance( $n, x, p$ )
2   ...
3   début
4      $p \leftarrow 1$ 
5     pour ( $cpt \leftarrow n ; cpt > 0 ; cpt \leftarrow cpt - 1$ ) faire
6       // invariant :  $p = x^{n-cpt}$ 
7        $p \leftarrow p * x$ 
8   // Nombre de passages dans la boucle = n ; cpt = 0 et p =  $x^{n-cpt} = x^n$ 

```

L'instruction "répéter ... jusqu'à" est une version légèrement différente de l'instruction "tant que". La structure de cette instruction :

- 1 **répéter**
- 2 | bloc de la boucle
- 3 **jusqu'à** (*condition d'arrêt*)

Cet énoncé est équivalent à l'énoncé suivant :

- 1 bloc de la boucle
- 2 **tant que** *non*(*condition d'arrêt*) **faire**
- 3 | bloc de la boucle

Par exemple, un algorithme permettant de calculer  $x$  à la puissance  $n$  peut être écrit :

---

**Algorithme 25 :** Algorithme pour calculer  $x^n$

---

```
1 Procédure puissance( $n, x, p$ )
2   ...
3   début
4      $p \leftarrow 1$ 
5      $cpt \leftarrow 0$ 
6     répéter
7       si ( $cpt > 0$ ) alors
8          $p \leftarrow p * x$ 
9          $cpt \leftarrow cpt + 1$ 
10      jusqu'à ( $cpt \geq n$ )
```

---

## 1.8 Les tableaux

Un tableau est une suite de  $n$  données de même type, rangées consécutivement. On dira que  $n$  est la *taille* du tableau et que les données sont ses *éléments*. Chaque élément est repéré dans le tableau par son *indice*. Un indice correspond à un numéro d'ordre dans le tableau, les éléments étant rangés par ordre d'indices consécutifs croissants.

**Déclaration d'un tableau :** une variable de type tableau est déclarée selon la syntaxe suivante

1 type-elem nom-tableau[d..f]

où *type-elem* est le type des éléments du tableau, *nom-tableau* est le nom du tableau, et *d* et *f* représentent respectivement les indices du premier et du dernier élément du tableau. Si  $d > f$  alors le tableau est vide (il n'a aucun élément), sinon il a  $f-d+1$  éléments de type *type-elem*.

Par exemple, à la suite des déclarations suivantes :

**Déclaration** : entier tab1[1..10]  
 réel tab2[12..44]

*tab1* est un tableau de 10 entiers indicés de 1 à 10 et *tab2* est un tableau de 33 réels indicés de 12 à 44.

Remarque : quand on déclare un tableau en C, on ne donne que le nombre d'éléments du tableau, l'indice du premier élément du tableau étant toujours 0. Ainsi, l'instruction C suivante :

```
int tab[10];
```

déclare un tableau *tab* de 10 entiers, dont le premier élément est à l'indice 0 et le dernier à l'indice 9, ce qui correspond à la déclaration algorithmique suivante :

**Déclaration** : entier tab[0..9]

Un cas particulier de tableau est la chaîne de caractères, que nous pouvons représenter comme dans l'exemple suivant (chaîne avec 1024 caractères) :

**Déclaration** : car chaîne[0..1023]

**Accès à un élément du tableau :** on accède à un élément dans un tableau à partir de son indice, la valeur de cet indice devant être comprise entre les indices de début et de fin du tableau. Par exemple, *tab1[8]* désigne l'élément d'indice 8 dans le tableau *tab1*, autrement dit le 8ème élément du tableau ; tandis que *tab2[14]* désigne l'élément d'indice 14 dans le tableau *tab2*, autrement dit le 3ème élément du tableau.

**Pré-condition** implicite aux algorithmes ayant des tableaux passés en paramètres : quand un tableau est déclaré en paramètre formel d'une procédure, les indices du premier et du dernier élément du tableau ne seront pas toujours précisés dans le type mais passés en paramètre. Ainsi, dans les exercices suivants, on déclarera souvent un paramètre de type tableau de la façon suivante :

Dans ce cas, on supposera en pré-condition que le tableau physique passé en paramètre effectif à la place de *tab* sera défini pour les indices compris entre *d* et *f*. Notons que le tableau passé en paramètre effectif pourra avoir été défini pour un intervalle d'indices  $[d_e..f_e]$  supérieur (tel que  $d - e \leq d$  et  $f \leq f_e$ ). Dans ce cas, la procédure appelée ne travaillera que sur une partie du tableau passé en paramètre (la partie entre *d* et *f*).

**1 Procédure** nom-algo(*tab, d, f*)  
**Entrée** : TypeEl<sup>t</sup> *tab[?..?]*  
                   entier *d, f*

**Parcours des éléments d'un tableau :** dans de nombreux algorithmes, on doit parcourir le tableau pour faire un traitement sur chaque élément du tableau. L'algorithme générique pour le parcours d'un tableau *tab* indicé de *d* à *f* est le suivant :

---

**Déclaration** : entier *i*  
**1 début**  
**2** *i*  $\leftarrow d  
**3** **tant que** *i*  $\leq f$  **faire**  
     // invariant : les éléments d'indice *j < i* ont déjà été traités  
     // Traitement de l'élément d'indice *i*  
**4** *i*  $\leftarrow i + 1  
   // Nombre de passages = *f - d + 1* ; *i = f + 1*$$

---

Cet algorithme peut être exprimé de façon équivalente de la façon suivante :

---

**Déclaration** : entier *i*  
**1 début**  
**2** **pour** (*i*  $\leftarrow d$ ; *i*  $\leq f$ ; *i*  $\leftarrow i + 1$ ) **faire**  
     // invariant : les éléments d'indice *j < i* ont déjà été traités  
     // Traitement de l'élément d'indice *i*  
   // Nombre de passages = *f - d + 1* ; *i = f + 1*

---

Par exemple, une version plus générale de l'algorithme 9 va prendre en entrée un tableau de nombres et va produire en sortie le minimum de ces nombres : algorithme 26.

---

#### Algorithme 26 : Le minimum dans un vecteur de nombres entiers

---

**1 Procédure** min(*vec, result*)  
**Entrée** : entier[1..taille] *vec*  
**Sortie** : entier *result*, correspondant à *min(vec[1], vec[2], ..., vec[taille])*  
**Précondition** : *taille*  $\geq 1$   
**Postcondition** : *result*  $\in vec$   
**début**  
**2** *result*  $\leftarrow vec[1];  
**3** **pour** *i*  $\leftarrow 2$ ; *i*  $\leq taille$ ; *i*  $\leftarrow i + 1$  **faire**  
**4**    **si** *vec[i] < result* **alors**  
**5**      *result*  $\leftarrow vec[i];$$

---

Une exécution de cet algorithme pour les entrées *taille=3, vec=<11, 4, 24>* va produire la trace Table 1.8.  
L'algorithme 26 peut être décrit en utilisant la boucle *tant que* (algorithme 27) ou *répéter jusqu'à* (algorithme 28).

Ligne	<i>taille</i>	<i>vec[1]</i>	<i>vec[2]</i>	<i>vec[3]</i>	<i>i</i>	<i>result</i>
1	3	11	4	24	?	?
2	3	11	4	24	?	11
3	3	11	4	24	2	11
4	3	11	4	24	2	11
5	3	11	4	24	2	4
3	3	11	4	24	3	4
4	3	11	4	24	3	4

TABLE 1.8 – Trace d'exécution de l'algorithme 26 pour  $\langle \text{taille}=3, \text{vec}=\langle 11, 4, 24 \rangle \rangle$

---

**Algorithme 27 : Le minimum dans un vecteur de nombres entiers**

---

**Entrée** : vecteur[entier] *vec*  
                   entier *taille*, le nombre d'éléments du vecteur *vec*

**Sortie** : entier *result*, correspondant à  $\min(\text{vec}[1], \text{vec}[2], \dots, \text{vec}[\text{taille}])$

**1 début**

2     *result*  $\leftarrow \text{vec}[1];$   
 3     *i*  $\leftarrow 2;$   
 4     **tant que** *i*  $\leq \text{taille}$  **faire**  
 5         **si** *vec[i]*  $< \text{result}$  **alors**  
 6             *result*  $\leftarrow \text{vec}[i];$   
 7         *i*  $\leftarrow i + 1;$

---

**Algorithme 28 : Le minimum dans un vecteur de nombres entiers**

---

**Entrée** : vecteur[entier] *vec*  
                   entier *taille*, le nombre d'éléments du vecteur *vec*

**Sortie** : entier *result*, correspondant à  $\min(\text{vec}[1], \text{vec}[2], \dots, \text{vec}[\text{taille}])$

**1 début**

2     *result*  $\leftarrow \text{vec}[1];$   
 3     *i*  $\leftarrow 1;$   
 4     **répéter**  
 5         *i*  $\leftarrow i + 1;$   
 6         **si** *vec[i]*  $< \text{result}$  **alors**  
 7             *result*  $\leftarrow \text{vec}[i];$   
 8     **jusqu'à** *i*  $= \text{taille};$

---

► **Problèmes**

- (Difficulté=2)Donner un algorithme qui permet de calculer le maximum d'un vecteur de nombres entiers.
- (Difficulté=2)Donner un algorithme qui permet de calculer la moyenne d'un vecteur de nombres flottants.
- (Difficulté=5)Proposer un algorithme qui permet de rechercher la première occurrence d'une chaîne de caractères *mot* dans une autre chaîne de caractères *page*.
- (Difficulté=5)Pour un séparateur connu (de type caractère) trouvez le nombre de mots dans un texte (chaîne de caractères).

5. (Difficulté=2) Soit l'algorithme décrit dans Algorithme 29. Quelle valeur va renvoyer cet algorithme pour  $n = 4$  ?
6. (Difficulté=4) Donner un algorithme qui permet de trouver le nombre qui apparaît le plus souvent dans un tableau de nombres. Si plusieurs solutions sont possibles (plusieurs nombres avec cette propriété) donner aussi le nombre de ces solutions.

**Algorithme 29 : Problème : boucles****Entrée** : entier  $n$ **Sortie** : entier  $resultat$ **1 début**    **Déclaration** : entier  $i$         boolean  $b$      $b \leftarrow true;$     **pour**  $i \leftarrow 0, n$  **faire**        **si**  $b = true$  **alors**             $resultat \leftarrow resultat + 1/(2 * i + 1);$         **sinon**             $resultat \leftarrow resultat - 1/(2 * i + 1);$          $b \leftarrow \neg b;$      $resultat \leftarrow resultat * 4;$

## 1.9 Analyse d'un algorithme

Pour un algorithme il est important de savoir s'il est correct et s'il consomme efficacement les ressources à sa disposition (mémoire, processeur).

Prenons comme exemple le problème de calcul de la somme de  $n$  premiers nombres entiers positifs non-nulles ( $\sum_{i=1}^n i$ ). Nous proposons trois algorithmes différents pour résoudre ce problème : les algorithmes 30, 31 et 32.

---

### Algorithme 30 : La somme de $n$ premiers entiers strictement positifs : Solution 1

---

```

1 Procédure somme( $n, sum$ )
    Entrée      : entier  $n$ 
    Sortie       : entier  $sum$ 
    début
2        sum ←  $n/2$ ;
3        sum ← sum * ( $n + 1$ );
4

```

---



---

### Algorithme 31 : La somme de $n$ premiers entiers strictement positifs : Solution 2

---

```

1 Procédure somme( $n, sum$ )
    Entrée      : entier  $n$ 
    Sortie       : entier  $sum$ 
    début
2        sum ←  $n * (n + 1)$ ;
3        sum ← sum/2;
4

```

---



---

### Algorithme 32 : La somme de $n$ premiers entiers strictement positifs : Solution 3

---

```

1 Procédure somme( $n, sum$ )
    Entrée      : entier  $n$ 
    Sortie       : entier  $sum$ 
    début
2        sum ← 1;
3        pour ( $i \leftarrow 2; i \leq n; i \leftarrow i + 1$ ) faire
4            sum ← sum +  $i$ ;
5

```

---

Nous observons que la première proposition (Solution 1) est incorrecte : pour des valeurs impaires de  $n$  le résultat ne correspond pas au résultat attendu. L'erreur se situe sur la ligne 2, où  $sum$  est un entier et sa valeur ne prendra pas en compte la fraction décimale de la division par 2. Un algorithme est **correct** si pour chaque entrée possible il donne la sortie attendue.

Les deux autres solutions sont correctes, mais avec un coût calculatoire (**complexité**) différent. Pour simplifier, nous considérons que chaque instruction élémentaire s'effectue dans une unité de temps. Le nombre d'unités de temps nécessaires pour exécuter l'algorithme 31 est de 2, tandis que l'algorithme 32 nécessite  $2*n$  unités de temps (voir les tables 1.9 et 1.10 détaillant le calcul). Ceci nous conduit à privilégier la solution 2, qui sera plus efficace pour des valeurs grandes pour  $n$ . Nous disons que l'algorithme 31 a une complexité temporelle de l'ordre de  $\Theta(1)$  et l'algorithme 32 est de l'ordre de  $\Theta(n)$ .

Ligne	Coût en temps	Nombre de fois
1	0	1
2	1	1
3	1	1
<b>Total</b>		2

TABLE 1.9 – Temps d'exécution de l'algorithme 31

Ligne	Coût en temps	Nombre de fois
1	0	1
2	1	1
3	1	$n$
4	1	$n - 1$
<b>Total</b>		$2n$

TABLE 1.10 – Temps d'exécution de l'algorithme 32

Prouver qu'un algorithme est correct est un problème difficile et dépasse le cadre de ce cours introductif. Nous nous intéressons dans cette section à l'analyse du temps d'exécution d'un algorithme par rapport à ses entrées.

### ► Notation $\Theta$

La notation  $\Theta$  (pronunciation : *thêta*) est utilisée pour donner un ordre de grandeur du temps d'exécution d'un algorithme. Plus exactement, pour une fonction  $g(n)$  nous définissons  $\Theta(g(n))$  comme étant l'ensemble suivant :

$$\Theta(g(n)) = \{f(n) \mid \exists c_1, c_2 > 0, n_0 \in \mathbb{N} \text{ telles que } 0 \leq c_1 * g(n) \leq f(n) \leq c_2 * g(n), \forall n \geq n_0\}$$

D'une manière graphique (figure 1.2), l'ensemble  $\Theta(g(n))$  va contenir toutes les fonctions  $f(n)$  qui peuvent être "prises en sandwich" avec deux fonctions  $c_1 * g(n)$  et  $c_2 * g(n)$  à partir d'un  $n_0$  bien précisé,  $c_1$  et  $c_2$  étant des constantes. Plus exactement, la fonction  $f(n)$  est égale à la fonction  $g(n)$  à un facteur constant près, à partir d'un  $n_0$  fixé. Par convention, si  $f(n) \in \Theta(g(n))$ , nous écrivons  $f(n) = \Theta(g(n))$ .

**Exemple** ([Cormen]) : prouver que  $\frac{1}{2} * n^2 - 3 * n = \Theta(n^2)$ . Pour ce faire, on doit déterminer les constantes strictement positives  $c_1$ ,  $c_2$  et  $n_0$  telles que :

$$c_1 * n^2 \leq \frac{1}{2} * n^2 - 3 * n \leq c_2 * n^2, \forall n \geq n_0$$

La division de cette inégalité par  $n^2$  donne :

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

Cette inégalité est satisfaite si nous prenons par exemple  $c_1 = \frac{1}{14}$ ,  $c_2 = \frac{1}{2}$  et  $n_0 = 7$ . En conclusion  $\frac{1}{2} * n^2 - 3 * n = \Theta(n^2)$ . D'autres choix sont possibles pour les constantes  $c_1$ ,  $c_2$  et  $n_0$  - nous avons mis en évidence une solution.

**Exemple** ([Cormen]) : prouver que  $6 * n^3 \neq \Theta(n^2)$ . Supposons que  $c_2$  et  $n_0$  existent tels que  $6 * n^3 \leq c_2 * n^2, \forall n \geq n_0$ . Dans ce cas  $n \leq \frac{c_2}{6}$ , ce qui est impossible pour un  $n$  arbitrairement grand, puisque  $c_2$  est une constante.

Intuitivement, les termes d'ordre inférieur dans la définition de  $\Theta$  peuvent être ignorés. Par exemple :  $\Theta(n^5 - 2 * n) = \Theta(n^5)$ .

Quelques propriétés utiles de l'ensemble  $\Theta$  :

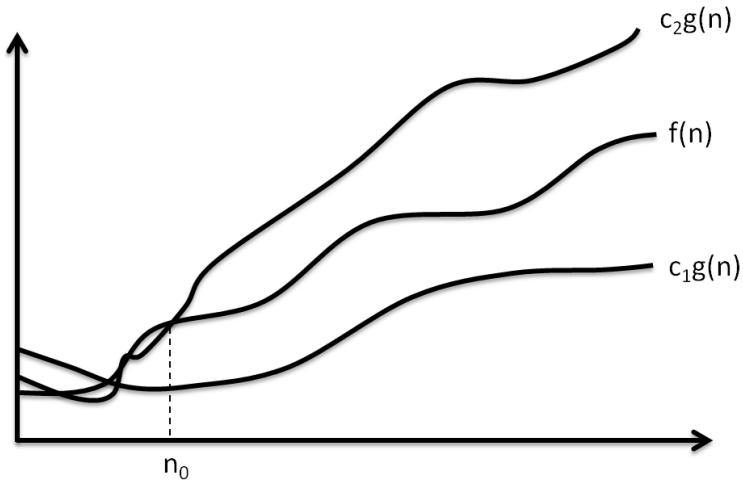


FIGURE 1.2 – Explication graphique pour la notation  $f(n) = \Theta(g(n))$

- *transitivité* :  $f(n) = \Theta(g(n)) \wedge g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$
- *réflexivité* :  $f(n) = \Theta(f(n))$
- *symétrie* :  $f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$

Quelques exemples de complexité pour différentes fonctions décrivant le nombre d'instructions exécutées par un algorithme :

- $2 + 15 = \Theta(1)$
- $2n + 1 = \Theta(n)$
- $n^2 - 2n + 1 = \Theta(n^2)$
- $\sum_{i=0}^k c_i * n^i = \Theta(n^k)$
- $2^n + n^2 = \Theta(2^n)$

Pour l'algorithme 32, le nombre d'opérations exécutées est  $2n$ , donc la complexité temporelle de cet algorithme est de  $\Theta(n)$ .

Dans le processus de conception d'un algorithme pour un problème donné, l'intérêt est de choisir l'algorithme avec une complexité temporelle minimale. Pour certains problèmes nous pouvons prouver que le  $\Theta$  d'un algorithme est optimal, comme par exemple la complexité de  $\Theta(n * \log(n))$  pour le tri d'un vecteur de nombres. Pour d'autres problèmes l'optimalité reste un problème ouvert. Parmi les différentes complexités suivantes, l'analyste va chercher à trouver un algorithme avec une complexité la plus à gauche possible :

$$\Theta(1) < \Theta(\log(n)) < \Theta(n) < \Theta(n * \log(n)) < \Theta(n^2) < \dots < \Theta(n^k) < \dots < \Theta(2^n) < \dots$$

## ► Notation $\mathcal{O}$

La notation  $\Theta$  est utilisée dans le cas où nous pouvons trouver des constantes pour borner inférieurement et supérieurement la fonction qui donne le temps d'exécution d'un algorithme. Mais dans beaucoup des cas pratiques nous pouvons trouver seulement une limite supérieure. Dans ce cas nous utilisons la notation  $\mathcal{O}$ .

Plus exactement, pour une fonction  $g(n)$  nous définissons  $\mathcal{O}(g(n))$  comme étant l'ensemble suivant :

$$\mathcal{O}(g(n)) = \{f(n) | \exists c > 0, n_0 \in \mathbb{N} \text{ telles que } 0 \leq f(n) \leq c * g(n), \forall n \geq n_0\}$$

D'une manière graphique (figure 1.3), l'ensemble  $\mathcal{O}(g(n))$  va contenir toutes les fonctions  $f(n)$  qui sont inférieures à  $g(n)$ , à un facteur constant près. La notation  $\mathcal{O}$  est moins stricte que  $\Theta$  : si un algorithme a une complexité  $\Theta(f(n))$ ,

alors il est aussi de complexité  $\mathcal{O}(f(n))$ , mais la réciproque n'est pas valable. Dans la pratique, nous utilisons plus la notation  $\mathcal{O}$ , qui est plus facilement démontrable.

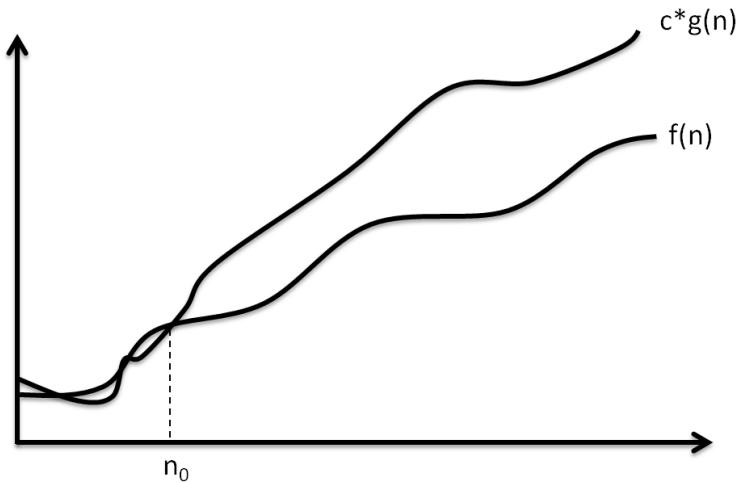


FIGURE 1.3 – Explication graphique pour la notation  $f(n) = \mathcal{O}(g(n))$

### ► Notation $\Omega$

Si la notation  $\mathcal{O}$  représente une  *borne asymptotique supérieure* d'un fonction  $g$ , nous pouvons définir aussi une  *borne asymptotique inférieure*  $\Omega$ .

Plus exactement, pour une fonction  $g(n)$  nous définissons  $\Omega(g(n))$  comme étant l'ensemble suivant :

$$\Omega(g(n)) = \{f(n) | \exists c > 0, n_0 \in \mathbb{N} \text{ telles que } 0 \leq c * g(n) \leq f(n), \forall n \geq n_0\}$$

D'une manière graphique (figure 1.4), l'ensemble  $\Omega(g(n))$  va contenir toutes les fonctions  $f(n)$  qui sont supérieures à  $g(n)$ , à un facteur constant près. La notation  $\Omega$  est moins stricte que  $\Theta$  : si un algorithme à une complexité  $\Theta(f(n))$ , alors il est aussi de complexité  $\Omega(f(n))$ , mais la réciproque n'est pas valable.

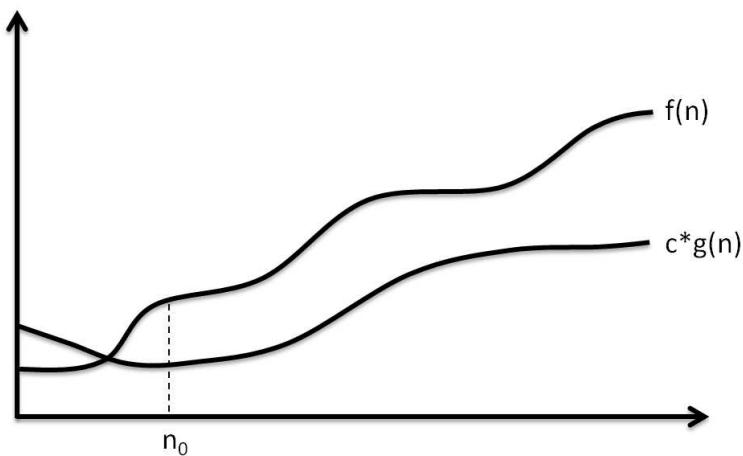


FIGURE 1.4 – Explication graphique pour la notation  $f(n) = \Omega(g(n))$

Une propriété importante :

$$f(n) = \Theta(g(n)) \Leftrightarrow (f(n) = \mathcal{O}(g(n))) \wedge (f(n) = \Omega(g(n)))$$

## ► Exemples

---

### Algorithme 33 : Calcul des $n$ premières valeurs de la suite de Fibonacci

---

```

1 Procédure fibonacci( $n$ ,  $fibo$ )
    Entrée      : entier  $n$ 
    Sortie       : entier  $fibo[0..n]$ 
    Précondition :  $n \geq 1$ 
    Postcondition :  $fibo[0] = 1$ 
                       $fibo[1] = 1$ 
                       $fibo[n] = fibo[n - 1] + fibo[n - 2], \forall n \geq 2$ 
    Déclaration   : entier  $i$ 
début
    3    $fibo[0] \leftarrow 1$ 
    4    $fibo[1] \leftarrow 1$ 
    5    $i \leftarrow 1$ 
    6   tant que  $i < n$  faire
        // invariant :  $\forall j \in [2..i], fibo[j] = fibo[j - 1] + fibo[j - 2]$ 
        7    $i \leftarrow i + 1$ 
        8    $fibo[i] \leftarrow fibo[i - 1] + fibo[i - 2]$ 
    // Nombre de passages =  $n - 1$  ;  $i = n$  et  $\forall j \in [2..i], fibo[j] = fibo[j - 1] + fibo[j - 2]$ 

```

---

**Complexité :** A l'initialisation, on exécute 3 instructions. Puis, on passe  $n - 1$  fois dans la boucle, en exécutant  $3 * (n - 1) + 1$  instructions. En total  $3 * n + 1$  lignes de code seront exécutées. Par conséquent la complexité de l'algorithme est linéaire, en  $\Theta(n)$  et aussi  $\mathcal{O}(n)$ .

**Complexité** On passe  $n$  fois dans une première boucle ( $\mathcal{O}(n)$ ). On passe ensuite  $n - 2$  fois dans une deuxième boucle. A chaque passage dans cette deuxième boucle, si  $i$  est un nombre premier, on passe  $n/i - 1$  fois dans une troisième boucle, sinon on ajoute une opération. Pratiquement pour chaque nombre premier  $i$  nous ajoutons un nombre d'instructions de l'ordre de  $n/i$ . La troisième boucle ajoute donc de l'ordre de  $n + n/2 + n/3 + n/5 + \dots + n/n = n * (1 + 1/2 + 1/3 + 1/5 + \dots + 1/n)$  opérations. En sachant que :

$$\lim_{n \rightarrow +\infty} \left( \sum_{p \leq n} \left( \frac{1}{p} \right) - \log \log(n) \right) = M$$

où  $M = 0.26147\dots$  est la constante de Meissel-Mertens, nous obtenons que la complexité de cet algorithme est de l'ordre ( $\mathcal{O}(n * \log \log(n))$ )

## ► Problèmes

1. (Difficulté=2) Démontrer que  $n^3 - 2n + 1 = \Theta(n^3)$ .

**Algorithme 34 : Crible d'Eratosthène**

```

1 Procédure crible(n, crible)
    Entrée      : entier n
    Sortie       : logique crible[1..n]
    Précondition : n ≥ 2
    Postcondition : pour tout i ∈ [1..n], crible[i] = vrai ssi i est un nombre premier
    Déclaration   : entier i, j
début
    pour (i ← 1; i ≤ n; i ← i + 1 faire
           crible[i] ← vrai
        // Nombre de passages = n ; ∀i ∈ [1..n], crible[i] = vrai
    pour (i ← 2; i < n; i ← i + 1) faire
        // invariant : ∀k ∈ [1..i], crible[k] = vrai ssi k est un nombre premier et
        // ∀k ∈ [i + 1..n], crible[k] = vrai ssi k n'est pas multiple d'un nombre inférieur à i
        si crible[i] alors
               pour (j ← i + i; j ≤ n; j ← j + i) faire
                // invariant : ∀k < j, k est un multiple de i ⇒ crible[k] = faux
                crible[j] ← faux
                // Nombre de passages = n/i − 1 ; j > n et
                // ∀k < j, k est un multiple de i ⇒ crible[k] = faux
           // Nombre de passages = n − 2
        // i = n et ∀k ∈ [1..i], crible[k] = vrai ssi k est un nombre premier

```

---

2. (Difficulté=2) Démontrer que  $n^3 - 2n + 1 = \mathcal{O}(n^3)$ .

3. (Difficulté=5) Démontrer que  $\Theta(2^n) \neq \Theta(n^3)$ .

## 1.10 Procédures et fonctions

Un algorithme est généralement élaboré par une *démarche descendante*, qui consiste à décomposer le problème en sous-problèmes, chaque sous-problème devant être de nouveau spécifié puis résolu. Cette décomposition permet d'aborder le problème progressivement en créant des niveaux de description de plus en plus détaillés. Elle permet également de réutiliser la résolution de certains sous-problèmes pour résoudre de nouveaux problèmes.

Ainsi, un algorithme peut être “appelé” dans le corps d'un autre algorithme afin de résoudre un sous-problème.

### ► Paramètres effectifs

Lors de l'appel d'un algorithme, il faut préciser les valeurs des paramètres en entrée et, en retour, récupérer les valeurs des paramètres en sortie. Les paramètres utilisés pour cela sont appelés *paramètres effectifs*.

**Exemple :** Considérons le problème “somme-cos”, qui consiste à calculer la somme  $\cos(1) + \cos(2) + \cos(3) + \dots + \cos(n)$ .

---

#### Algorithme 35 : Calcul de la somme $\cos(1) + \cos(2) + \cos(3) + \dots + \cos(n)$

---

1 **Procédure** *somme-cos*(*n, sc*)

**Entrée** : entier *n*

**Sortie** : entier *sc*

**Précondition** :  $n \geq 1$

**Postcondition** :  $sc = \sum_{i=1}^n \cos(i)$

        ou, autrement dit,  $sc = \cos(1) + \cos(2) + \cos(3) + \dots + \cos(n)$

---

Pour résoudre ce problème, on a successivement besoin de calculer  $\cos(1)$ ,  $\cos(2)$ , ..., jusqu'à  $\cos(n)$ . Une possibilité pour calculer le  $\cos(x)$  est décrite dans l'algorithme 36.

---

#### Algorithme 36 : Calcul de $\cos(x)$ avec une précision *eps*

---

1 **Procédure** *cos*(*x, eps, cosx*)

**Entrée** : réel *x, eps*

**Sortie** : réel *cosx*

**Précondition** :  $eps > 0$

**Postcondition** :  $cosx = \cos(x)$ , avec une précision de *eps*

        (utilise le développement en série :  $\cos(x) = \sum_{i=0}^{\infty} (-1)^i \frac{x^{2i}}{(2i)!}$ )

---

On va donc réutiliser la résolution de  $\cos$  pour résoudre *somme-cos*. On dira que la procédure *somme-cos appelle la procédure cos*. Lors de cet appel, *somme-cos* doit préciser à *cos* les paramètres effectifs, c'est à dire :

1. la valeur de *x* pour laquelle on souhaite calculer  $\cos(x)$ ,
2. la valeur de *eps*, donnant la précision avec laquelle on veut calculer  $\cos(x)$ ,
3. la variable de *somme-cos* dans laquelle on souhaite récupérer le résultat du calcul de  $\cos(x)$  (résultat contenu dans le paramètre *cosx* de *cos*).

Ces paramètres sont précisés dans l'ordre de leur déclaration. Ainsi, l'appel de “cos” se fera par l'instruction

$\cos(expr1, expr2, var)$

où *expr1* et *expr2* sont deux expressions dont l'évaluation donne deux réels et *var* est une variable de type réel. L'exécution de cette instruction a pour conséquence d'affecter à *var* la valeur de *cosx* calculée par *cos* pour *x* = *expr1* et *eps* = *expr2*. Le problème *somme-cos* peut alors être résolu par l'algorithme 37.

**Algorithme 37 : Calcul de somme – cos(n)**

```

1 Procédure somme-cos(n, sc)
2   ...
3   Déclaration   : réel i, cosi
4       const eps = 0.0001
5   début
6     sc ← 0
7     pour (i ← 1; i ≤ n; i ← i + 1) faire
8       // invariant: sc =  $\sum_{k=1}^{i-1} \cos(k)$ 
9       cos(i, eps, cosi)
10      sc ← sc + cosi
11    // Nombre de passages dans la boucle = n ; i = n + 1 et sc =  $\sum_{k=1}^{i-1} \cos(k)$ 

```

**D'une façon plus générale,** quand un algorithme est appelé dans un autre algorithme, on précise la liste des paramètres effectifs, en respectant l'ordre donné lors de la définition de l'algorithme appelé. Le type d'un paramètre effectif doit être le même que celui du paramètre formel correspondant. L'exécution d'un tel appel de procédure s'effectue alors en 3 étapes :

1. Affectation des valeurs des paramètres effectifs aux paramètres en entrée correspondants
2. Exécution de la procédure appelée dans un nouvel environnement
3. Affectation des valeurs des paramètres en sortie de la procédure appelée aux paramètres effectifs correspondants

**Remarques.**

- Le paramètre effectif correspondant à un paramètre en entrée peut être une valeur, une variable contenant une valeur ou une expression retournant une valeur. Dans tous les cas, la valeur doit être de même type que le paramètre formel correspondant.
- Le paramètre effectif correspondant à un paramètre en sortie doit toujours être une variable. De plus, s'il y a plusieurs paramètres en sortie, alors lors de l'appel de la procédure, il faudra utiliser des variables différentes comme paramètres effectifs.

### ► Appels successifs d'algorithmes

Soit l'algorithme pour le calcul de la somme des  $n$  premières factorielles suivant : pour calculer la somme des  $n$  premières factorielles, on pourrait écrire une boucle faisant varier une variable  $i$  de 1 à  $n$ , et calculer à chaque passage la valeur de  $i!$  en faisant un appel à la procédure *factorielle*, puis additionner cette valeur à une variable *sf*. On obtiendrait alors l'algorithme 38.

La boucle **tant que** de cet algorithme contient un appel à *factorielle*. On sait que pour calculer la factorielle d'un nombre  $i$ , cet algorithme effectue de l'ordre de  $\mathcal{O}(i)$  opérations. Or *somme-fact* appelle  $n - 1$  fois *factorielle* pour les valeurs de  $i$  variant de 2 à  $n$ . Par conséquent, pour calculer *sf* à partir de  $n$ , l'algorithme *somme-fact* fera de l'ordre de  $2 + 3 + 4 + \dots + n = n(n+1)/2 - 1$  opérations. Ainsi, la complexité de *somme-fact* est  $\mathcal{O}(n^2)$ .

On souhaite connaître la valeur de  $x$  après l'exécution de “somme-fact(3, x)”. Pour cela, on va simuler l'exécution de cette instruction en représentant les environnements d'exécution des procédures (ces environnements sont en fait empilés dans une pile d'environnements) : Table 1.11.

Il existe un algorithme bien plus efficace pour calculer la somme des  $n$  premières factorielles : il suffit de mémoriser la valeur de  $i!$  dans une variable  $f$  ; à chaque passage dans la boucle, au lieu de recalculer depuis le début la valeur de  $i!$ , il suffit de mettre à jour la valeur de  $f$ . On obtient l'algorithme 39.

**Algorithme 38 : Somme des  $n$  premières factorielles**

```

1 Procédure somme-fact( $n, sf$ )
    Entrée      : entier  $n$ 
    Sortie       : entier  $sf$ 
    Précondition :  $n \geq 1$ 
    Postcondition :  $sf = \sum_{i=1}^n i!$ 
                    ou, autrement dit,  $sf = 1! + 2! + 3! + \dots + n!$ 
    Déclaration   : entier  $i, fi$ 
    début
         $sf \leftarrow 1$ 
         $i \leftarrow 1$ 
        tant que  $i < n$  faire
            // invariant:  $sf = \sum_{k=1}^i k!$ 
             $i \leftarrow i + 1$ 
            factorielle( $i, fi$ )
             $sf \leftarrow sf + fi$ 
        // Nombre de passages dans la boucle =  $n - 1$  ;  $i = n$  et  $sf = \sum_{k=1}^i k!$ 
    
```

**Algorithme 39 : Somme des  $n$  premières factorielles - version plus performante**

```

1 Procédure somme-fact( $n, sf$ )
    Entrée      : entier  $n$ 
    Sortie       : entier  $sf$ 
    Précondition :  $n \geq 1$ 
    Postcondition :  $sf = \sum_{i=1}^n i!$ 
                    ou, autrement dit,  $sf = 1! + 2! + 3! + \dots + n!$ 
    Déclaration   : entier  $i, f$ 
    début
         $f \leftarrow 1$ 
         $sf \leftarrow 1$ 
         $i \leftarrow 1$ 
        tant que  $i < n$  faire
            //  $f = i!$  et  $sf = \sum_{j=1}^i j!$ 
             $i \leftarrow i + 1$ 
             $f \leftarrow f * i$ 
             $sf \leftarrow sf + f$ 
        // Nombre de passages dans la boucle =  $n$  ;  $i = n$  et  $sf = \sum_{j=1}^i j!$ 
    
```

**Complexité (algorithme 39) :** A l'initialisation, on fait 3 affectations. On passe  $n - 1$  fois dans la boucle, et à chaque passage, on fait 1 comparaison, 3 affectations, 2 additions et 1 multiplication. Par conséquent la complexité de l'algorithme est en  $\mathcal{O}(n)$ . Cet algorithme est donc bien plus rapide que celui consistant à appeler factorielle à chaque itération!!!

Variables : $x$	Environnement de somme-fact Variables : $n, sf, i, si$	Environnement de factorielle Variables : $n, f$	Environnement de factorielle Variables : $n, f$
appel de somme-fact(3,sf)			
	<p>passage des entrées :</p> $n \leftarrow 3$ <p>exécution de somme-fact(3,sf) :</p> $sf \leftarrow 1$ $i \leftarrow 1$ $i \leftarrow 2$ <p>appel de factorielle(2,f)</p>		
		<p>passage des entrées :</p> $n \leftarrow 2$ <p>exécution de factorielle(2,f) :</p> $\dots$ $f \leftarrow 1 * 2$ <p>fin de factorielle(2,f)</p>	
	<p>passage des sorties :</p> $fi \leftarrow 2$ $sf \leftarrow 1 + 2$ $i \leftarrow 3$ <p>appel de factorielle(3,f)</p>		
			<p>passage des entrées :</p> $n \leftarrow 3$ <p>exécution de factorielle(3,f) :</p> $\dots$ $f \leftarrow 1 * 2 * 3$ <p>fin de factorielle(3,f)</p>
	<p>passage des sorties :</p> $fi \leftarrow 6$ $sf \leftarrow 3 + 6$ $i \leftarrow 4$ <p>fin de somme-fact(3,sf)</p>		
passage des sorties : $sf \leftarrow 9$			

TABLE 1.11 – Simulation de l'exécution de  $somme - fact(3, sf)$

## ► Paramètres en entrée et en sortie

Lors de l'appel d'un algorithme, une même variable peut être utilisée pour passer une valeur en entrée et en récupérer une autre en sortie. Considérons par exemple la suite d'instructions :

```
n ← 4
factorielle(n, n)
```

La valeur 4 de la variable  $n$  est passée comme paramètre en entrée de *factorielle*; tandis qu'à la fin de l'exécution de *factorielle*, la valeur du paramètre en sortie est affectée à  $n$ . Ainsi, à la suite de l'exécution de ces 2 instructions, la variable  $n$  aura pour valeur 24.

Il s'agit là d'une utilisation particulière de *factorielle* qu'il n'est probablement pas intéressant de généraliser. En revanche, certains problèmes consistent systématiquement à modifier la valeur d'un paramètre, autrement dit, le paramètre doit être passé à la fois en entrée, pour connaître sa valeur de départ, et en sortie, pour modifier cette valeur. Dans ce cas, plutôt que de dupliquer ce paramètre (en le faisant apparaître à la fois en entrée et en sortie), on dira qu'il s'agit d'un paramètre en entrée/sortie. Un paramètre en entrée/sortie sera déclaré par :

**1 Procédure** nom-algo(*var*)  
  └ Entrée/Sortie : type-de-var *var*

Cet algorithme pourra être appelé par :

nom-algo(*x*)

de telle sorte que le paramètre effectif *x* soit une variable de même type que la variable *var*. Lors de l'appel de "nom-algo(*x*)", la valeur du paramètre effectif *x* est affectée à *var*, tandis qu'au retour de l'appel, la valeur de *var* est affectée à *x*.

Pour spécifier le rôle d'un tel algorithme, c'est à dire la relation entre la valeur initiale du paramètre en entrée/sortie, et sa valeur finale, on a besoin de distinguer ces deux valeurs. Ainsi, on notera  $var^{in}$  la valeur du paramètre *var* en entrée et  $var^{out}$  sa valeur en sortie.

**Exemple :** Echanger les valeurs de 2 variables

**Algorithme 40 :** Echanger les valeurs de 2 variables

**1 Procédure** échanger(*a, b*)  
  └ Entrée/Sortie : entier *a, b*  
  └ Postcondition :  $a^{out} = b^{in}$   
     $b^{out} = a^{in}$   
  └ Déclaration : entier *aux*  
  └ début  
    └ aux  $\leftarrow a$   
    └ a  $\leftarrow b$   
    └ b  $\leftarrow aux$

## ► Procédures et fonctions

On distingue deux types d'algorithmes :

- les algorithmes dont le but est de calculer une (et une seule) valeur à partir d'un certain nombre d'autres valeurs données en entrée ; ces algorithmes seront implémentés par des *fonctions*,

- les autres algorithmes, ayant un nombre quelconque de paramètres en entrée et en sortie ; ces algorithmes seront implémentés par des *procédures*.

Une fonction calcule une (et une seule) valeur à partir d'un certain nombre d'autres valeurs. Autrement dit, une fonction prend en entrée 0, 1 ou plusieurs paramètres et retourne en sortie une valeur. Par exemple, l'algorithme "factorielle" calcule  $n!$  pour une valeur de  $n$  donnée ; l'algorithme "puissance" calcule  $x^n$  à partir de valeurs données pour  $x$  et  $n$ . La déclaration d'une fonction s'effectuera de la façon suivante :

**1 Fonction type-fct nom-fct(< paramètres >)**

**Entrée** : liste des paramètres (types et noms)

**Précondition** : Conditions sur les paramètres en entrée

**Postcondition** : relation entre la valeur rentrée par la fonction et ses paramètres en entrée

La valeur rentrée en sortie par la fonction sera spécifiée dans le corps de la fonction par l'instruction

**1 retourner expr**

où  $expr$  est une expression (une valeur, une variable contenant une valeur, ou une opération entre expressions) de type *type-fct*.

Attention : les instructions se trouvant après une instruction de retour de fonction ne sont pas exécutées. Autrement dit, l'exécution de la fonction se termine avec la première instruction de retour trouvée.

Une fonction peut être appelée dans un autre algorithme. Un appel de fonction renvoie une valeur et est donc assimilé à une valeur du type de la fonction.

**Exemple 1 :** reprise de factorielle.

**Algorithme 41 : L'algorithme "factorielle" vu comme une fonction****1 Fonction entier factorielle(n)**

**Entrée** : entier  $n$

**Précondition** :  $n \geq 0$

**Postcondition** : retourne  $n!$

**Déclaration** : entier  $i, f$

**début**

$f \leftarrow 1$

**pour** ( $i \leftarrow 1; i \leq n; i \leftarrow i + 1$ ) **faire**

        // invariant:  $f = (i - 1)!$

$f \leftarrow f * i$

    // Nombre de passages =  $n$  ;  $i = n + 1$  et  $f = (i - 1)! = n!$

**retourner**  $f$

6

La fonction "factorielle" peut alors être appelée dans une expression. Par exemple, après l'exécution des instructions

$x \leftarrow \text{factorielle}(4)$

$y \leftarrow \text{factorielle}(x - 3 * \text{factorielle}(3))$

la variable  $x$  a pour valeur  $4! = 24$ , et la variable  $y$  a pour valeur  $(24 - 3 * 3!)! = 6! = 240$ .

## ► Modes de passage des paramètres

Lorsque l'on conçoit un algorithme, on a juste besoin de spécifier, pour chaque paramètre, s'il est donné en entrée, ou s'il est calculé en sortie, ou s'il est donné en entrée puis modifié par l'algorithme pour être ensuite retourné en sortie. Ainsi, au niveau de l'algorithme, on spécifie pour chaque paramètre son mode (entrée, sortie ou entrée/sortie). Au moment de programmer l'algorithme, il s'agit de trouver un mécanisme permettant d'implémenter le comportement correspondant au mode choisi :

- pour un paramètre en entrée : lors de l'appel de la procédure, il faut passer la valeur du paramètre effectif au paramètre formel ; pendant l'exécution de la procédure appelée, si le paramètre formel est modifié, il ne faut pas répercuter cette modification sur le paramètre effectif.
- pour un paramètre en sortie : à la fin de l'exécution de la procédure appelée, il faut transmettre la valeur du paramètre en sortie au paramètre effectif correspondant.
- pour un paramètre en entrée/sortie : lors de l'appel de la procédure, il faut passer la valeur du paramètre effectif au paramètre formel ; à la fin de l'exécution de la procédure appelée, il faut transmettre la nouvelle valeur du paramètre formel au paramètre effectif correspondant.

Pour implémenter ces comportements, on dispose (dans la plupart des langages de programmation) de deux mécanismes de passage de paramètres : le passage par valeur et le passage par référence.

Considérons par exemple une procédure  $q$  qui appelle une procédure  $p$ , et supposons que la procédure  $p$  a un seul paramètre  $x$ .

- Si  $x$  est passé par valeur :
  - lors de l'appel de  $p$ , on crée un nouvel environnement au dessus de l'environnement de  $q$ . Ce nouvel environnement contient une nouvelle variable, de nom  $x$ , et on recopie la valeur du paramètre effectif correspondant à  $x$  dans cette nouvelle variable.
  - Lors de l'exécution de  $p$ , on utilise cette nouvelle variable dont la valeur peut éventuellement être modifiée.
  - A la fin de l'exécution de  $p$ , son environnement est détruit (et la valeur du paramètre aussi). En revanche, l'environnement de la procédure  $q$  n'est pas modifié.
- Si  $x$  est passé par référence :
  - lors de l'appel de  $p$ , on ne crée pas un nouvel emplacement mémoire pour stocker  $x$ . A la place, on crée un lien (une référence !) entre le paramètre formel de nom  $x$  et le paramètre effectif correspondant, de sorte que  $p$  peut accéder au paramètre effectif correspondant à  $x$ .
  - Lors de l'exécution de  $p$ , les modifications faites sur  $x$  sont directement effectuées dans l'environnement de  $q$  sur le paramètre effectif correspondant à  $x$ .
  - A la fin de l'exécution de  $p$ , on détruit le lien entre  $x$  et le paramètre effectif de  $q$  correspondant, mais les modifications faites par  $p$  sur ce paramètre effectif restent.

Ainsi, lors de l'implémentation d'un algorithme dans un langage de programmation,

- un paramètre en sortie ou en entrée/sortie sera nécessairement passé par référence ;
- un paramètre en entrée sera généralement passé par valeur.

Cependant, lorsque le paramètre en entrée est codé sur un grand nombre d'octets (ce sera le cas des tableaux par exemple), on ne le passera pas par valeur mais par référence afin d'éviter d'avoir à recopier sa valeur, ce qui peut être long et couteux en place mémoire.

## ► Problèmes

1. (Difficulté=3) Décrivez en pseudo-code l'algorithme 36.

## 1.11 Récursivité - Divide et impera

La récursivité est une stratégie de résolution de problèmes algorithmiques où un problème est divisé en sous-problèmes de même type. Cette technique porte aussi le nom de *divide et impera* (diviser et conquérir).

Un exemple classique est le calcul du factoriel d'un nombre naturel  $n$ . La formule utilisée pour définir le factoriel :

$$n! = 1 * 2 * \dots * (n - 1) * n \quad (1.1)$$

or, d'une manière différente :

$$\text{fact}(n) = \text{fact}(n - 1) * n \quad (1.2)$$

Pratiquement nous avons ré-formulé un problème ( $\text{fact}(n)$ ) en utilisant un sous-problème ( $\text{fact}(n - 1)$ ). Nous effectuons cette substitution jusqu'à ce que le sous-problèmes sont banales ( $\text{fact}(0)$ ). Ceci nous conduit à l'algorithme suivant :

---

### Algorithme 42 : Calcul de factorielle en utilisant la récursivité

---

1 **Fonction**  $\text{Fact}(n)$

```

Entrée      : entier  $n$ 
Sortie      : entier factoriel
Précondition :  $n \geq 0$ 
Postcondition : factoriel =  $n!$ 

début
  si  $n = 0$  alors
    |   factoriel  $\leftarrow 1$ 
  sinon
    |   factoriel  $\leftarrow n * \text{Fact}(n - 1)$ 
  retourne factoriel
```

---

Un exemple de trace d'exécution de cet algorithme pour  $n = 4$  est donné dans le tableau 1.12.

Algorithme	Ligne	Algorithme	Ligne
Fact(4)	1	Fact(1)	1
Fact(4)	2	Fact(1)	2
Fact(4)	4	Fact(1)	4
Fact(4)	5	Fact(1)	5
Fact(3)	1	Fact(0)	1
Fact(3)	2	Fact(0)	2
Fact(3)	4	Fact(0)	3
Fact(3)	5	Fact(0)	6
Fact(2)	1	Fact(1)	6
Fact(2)	2	Fact(2)	6
Fact(2)	4	Fact(3)	6
Fact(2)	5	Fact(4)	6

TABLE 1.12 – Traces d'exécution de l'algorithme 42.

Chaque trace est constitué du nom du sous-algorithme et la ligne exécutée. Nous utilisons deux colonnes pour économiser l'espace.

Un autre exemple d'algorithme récursif est le calcul du  $n$ -ème terme de la suite de Fibonacci. La suite de Fibonacci se définit récursivement par les 3 règles suivantes :

1. première règle de base

$$fibo(0) = 1$$

2. deuxième règle de base

$$fibo(1) = 1$$

3. règle récursive

$$fibo(n) = fibo(n - 1) + fibo(n - 2) \text{ pour } n \geq 2$$

---

#### **Algorithme 43 : Calcul récursif de la $n$ -ième valeur de la suite de Fibonacci**

---

**1 Fonction entier  $fibo(n)$**

Entrée : entier  $n$

Précondition :  $n \geq 0$

Postcondition : retourne  $fibo(n)$

où  $fibo$  est la suite définie récursivement par :

$$fibo(0) = 1$$

$$fibo(1) = 1$$

$$fibo(n) = fibo(n - 1) + fibo(n - 2), \forall n \geq 2$$

**2 début**

3    **si**  $n \leq 1$  **alors**

4     **retourner** 1

5    **sinon**

6     **retourner**  $fibo\_rec(n-1)+fibo\_rec(n-2)$

Remarques :

- Cet algorithme termine. En effet, à chaque appel récursif la valeur de  $n$  est décrémentée de 1 ou de 2 de telle sorte que  $n$  converge vers le cas de base  $n \leq 1$ .
- Cet énoncé récursif est particulièrement plus concis que l'énoncé répétitif. Néanmoins, chaque appel à fibonacci engendrant 2 nouveaux appels récursifs, sa complexité est exponentielle en  $\mathcal{O}(2^n)$ . Sachant que la complexité de la version répétitive de cet algorithme est linéaire, *cet énoncé récursif est à proscrire*.

► **Problèmes**

1. (Difficulté=4) Proposer un algorithme non-récursif pour calculer la série de Fibonacci :  $F_0 = 0; F_1 = 1; F_n = F_{n-1} + F_{n-2}$ .
2. (Difficulté=7) **Les tours du Hanoi** : Proposer un algorithme pour déplacer une pile de  $n$  disques d'une tour A à une tour C à l'aide d'une tour auxiliaire B. Tous les disques ont un diamètre différent. Les règles à respecter :
  - On ne peut déplacer qu'un seul disque à la fois d'une tour à une autre ;
  - On ne peut empiler un disque sur un disque de diamètre inférieur.
3. (Difficulté=5) Proposer un algorithme récursif pour calculer la fonction d'Ackerman :

$$A(m, n) = \begin{cases} n + 1 & \text{si } m = 0 \\ A(m - 1, 1) & \text{si } m > 0 \text{ et } n = 0 \\ A(m - 1, A(m, n - 1)) & \text{si } m > 0 \text{ et } n > 0 \end{cases}$$

- 
4. (Difficulté=7) Proposer un algorithme pour placer huit dames d'un jeu d'échecs sur un échiquier de  $8 \times 8$  cases sans que les dames ne puissent se menacer mutuellement, conformément aux règles du jeu d'échecs (la couleur des pièces étant ignorée). Par conséquent, deux dames ne devraient jamais partager la même rangée, colonne, ou diagonale.
  5. (Difficulté=5) Afficher l'ensemble des nombres de  $n$  chiffres ne comportant que des 1 et des 2. Par exemple, l'ensemble des nombres de 3 chiffres ne comportant que des 1 et des 2 est  $\{222; 221; 212; 211; 122; 121; 112; 111\}$

## 1.12 Problèmes

- (Difficulté = 6)(sujet d'admission à l'université "Babes-Bolyai" de Cluj, 2013). Soit un ensemble de nombres naturelles positives  $X = \{x_1, x_2, \dots, x_n\}$ . Proposez un algorithme qui pour l'entrée X produit la sortie Y définie de la façon suivante :  $y \in Y \Leftrightarrow (y \in X) \wedge (y = \text{palindrome})$ . Un palindrome est un nombre qui à une représentation symétrique dans la base 10. Par exemple, 12721 est un palindrome (il peut se lire dans les deux sens avec la même valeur). Le Y en sortie doit être trié en ordre décroissante.

Exemple :  $X = (2, 2442, 2, 13, 131, 1, 313, 44, 677) \rightarrow Y = (2442, 313, 131, 44, 2, 1)$

- (Difficulté = 6). Soit  $IS$  en ensemble d'intervalles de forme  $IS = \{[x_1, y_1], [x_2, y_2], \dots, [x_n, y_n]\}$ , avec  $x_i \leq y_i$ . Dans cet ensemble les intervalles ne forment pas une partition, ça veut dire qu'un point peut se retrouver dans plusieurs intervalles. Nous définissons l'aire de l'ensemble  $IS$  comme étant  $\text{area}(IS) = \sum_{i=1}^n (b_i - a_i)$ . Deux ensembles  $IS_1$  et  $IS_2$  sont équivalents s'ils contiennent les mêmes points :  $IS_1 \equiv IS_2 \Leftrightarrow (\forall x \in IS_1 \Rightarrow x \in IS_2) \wedge (\forall x \in IS_2 \Rightarrow x \in IS_1)$ . Décrivez un algorithme qui produit un ensemble  $IS'$  équivalent avec  $IS$  avec une aire minimale.

Exemple :  $IS = \{[-8, 2], [1, 5], [7, 9]\} \Rightarrow IS' = \{[-8, 5], [7, 9]\}$

- (Difficulté=4) Soit un tableau de nombres distincts. Trouvez la position du deuxième plus grand nombre dans le tableau.
- (Difficulté=5) Soit deux tableaux avec le même nombre d'éléments. Proposez un algorithme qui vérifie que les deux tableaux ont les mêmes éléments. Exemple : {7; 23; 4; 15; 7} et {4; 7; 23; 7; 15}.

**CHAPITRE 2****ALGORITHMES DE TRI**

L'une des problèmes fréquents en informatique est le tri d'un ensemble d'éléments. Une version simplifiée de ce problème de tri : pour un tableau de nombres en entrée  $(a_1, a_2, \dots, a_n)$ , trouver une permutation du tableau en sortie  $(a'_1, a'_2, \dots, a'_n)$  avec la propriété que ses éléments soit réarrangés en ordre croissante ( $a'_1 \leq a'_2 \leq \dots \leq a'_n$ ). Par exemple, pour le tableau d'entrée  $(26, -14, 17, 34, 0, 18)$  nous attendons en sortie  $(-14, 0, 17, 18, 26, 34)$ .

La spécification d'une procédure de tri pour le tableau de réels  $A$  est la suivante :

**Algorithme 44 : Spécification d'une procédure de tri****1 Procédure  $tri(A, n)$** 

**Entrée/Sortie** : réel  $A[1..n]$

**Entrée** : entier  $n$

**Postcondition** : 1)- Le tableau en sortie est une permutation du tableau en entrée.

2)- Les éléments du tableau en sortie sont triés par ordre croissant, i.e.,

$$\forall i \in [1..n-1], A^{out}[i] \leq A^{out}[i+1]$$

Un nombre important d'algorithmes de tri est disponible avec des caractéristiques assez hétérogènes (occupation de la mémoire, temps d'exécution, performances sur des données particulières). Cependant il n'y a pas un algorithme "meilleur", la méthode de tri est à choisir pour chaque problème particulière en fonction de ses caractéristiques. Dans ce chapitre nous décrivons les algorithmes de tri les plus utilisés, et aussi un problème dérivé : la sélection d'éléments dans un ensemble.

Les caractéristiques qui nous intéressent pour un algorithme de tri :

- la mémoire occupée : plus précisément, nous souhaitons savoir si un algorithme utilise une quantité de mémoire constante à l'extérieur du tableau (*tri sur place*) ;
- la complexité temporelle de l'algorithme.

Une présentation ludique présentant les algorithmes de tri les plus importants est disponible sur le site Web :"Animated Sorting Algorithms"<sup>1</sup>.

## 2.1 Tri par sélection

Principe de fonctionnement :

1. on recherche le plus petit élément du tableau, et on l'échange avec le premier élément du tableau,

1. <http://www.sorting-algorithms.com/>

2. on recherche le plus petit élément du sous-tableau commençant au deuxième indice, et on l'échange avec le deuxième élément du tableau,
3. on recherche le plus petit élément du sous-tableau commençant au troisième indice, et on l'échange avec le troisième élément du tableau,
4. ...

D'une façon plus générale, on répète les 3 opérations suivantes :

1. chercher l'indice  $ipp$  du plus petit élément du sous-tableau commençant à l'indice  $i$ ,
2. échanger l'élément d'indice  $ipp$  avec l'élément d'indice  $i$  du tableau,
3. incrémenter  $i$

Au départ,  $i$  est initialisé à 1 ; on arrête le processus lorsque  $i$  est égal à  $n$ .

Pour chercher l'indice du plus petit élément d'un sous-tableau, on peut utiliser la fonction donnant l'indice de l'élément le plus petit "indice\_plus\_petit" (Algorithme 45) ou une version qui ne fait pas appel à ce type de fonction (Algorithme 46).

---

#### Algorithme 45 : Tri par sélection

---

```

1 Procédure tri_selection( $A, n$ )
  Déclaration : entier  $i, ipp$ 
  début
    pour ( $i \leftarrow 1; i < n; i \leftarrow i + 1$ ) faire
       $ipp \leftarrow \text{indice\_plus\_petit}(A, i, n)$ 
      // invariant :  $A[ipp] = \min(A[1], \dots, A[i])$ 
      échanger( $A[ipp], A[i]$ )
      // invariant :  $A[1], \dots, A[i]$  sont à leur place finale

```

---



---

#### Algorithme 46 : Tri par sélection - 2

---

```

1 Procédure tri_selection( $A, n$ )
  Déclaration : entier  $i, ipp, j$ 
  début
    pour ( $i \leftarrow 1; i < n; i \leftarrow i + 1$ ) faire
       $ipp \leftarrow i$ 
      pour ( $j \leftarrow i + 1; j \leq n; j \leftarrow j + 1$ ) faire
        si ( $A[j] < A[ipp]$ ) alors
           $ipp \leftarrow j$ 
        échanger( $A[ipp], A[i]$ )

```

---

**Exemple :** la figure 2.1 présente les différents étapes d'exécution de l'algorithme 45 sur le vecteur d'entrée  $(26, -14, 17, 34, 0, 18)$ . Chaque étape de l'exécution (1 à 7 dans la figure 2.1) correspond à une exécution des lignes 4 et 5 de l'algorithme 45.

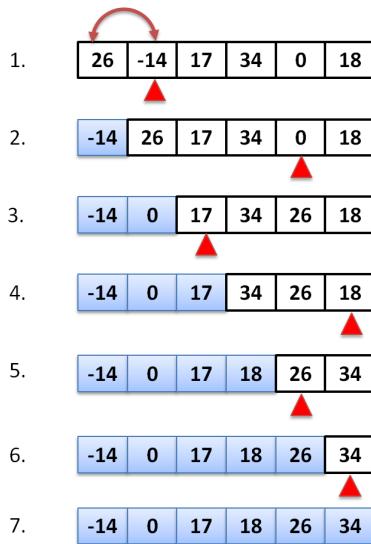


FIGURE 2.1 – Tri par sélection : exemple

**Complexité :** Pour trier le tableau  $A$ , l'algorithme 45 passe  $n - 1$  fois dans la boucle 3. La fonction  $echanger(a, b)$  permet d'inter-changer les valeurs de  $a$  et  $b$ , avec une complexité constante. La fonction  $indice\_plus\_petit(A, start, end)$  a une complexité linéaire en fonction de la taille du tableau passé en paramètre ( $end - start$ ). Or, au premier appel à  $indice\_plus\_petit$ , le tableau a  $n$  éléments, au deuxième appel, il en a  $n - 1$ , au troisième il en a  $n - 2$ , ..., au  $n - 1^{eme}$  appel il en a 2. Ainsi, le nombre total d'opérations à effectuer pour trier un tableau comportant  $n$  éléments sera de l'ordre de  $n + n - 1 + n - 2 + \dots + 3 + 2 = n(n + 1)/2 - 1$ . Par conséquent, la complexité de ce tri est en  $\Theta(n^2)$ .

**Mémoire :** Cet algorithme réalise un tri sur place ( $\mathcal{O}(1)$ ).

## 2.2 Tri par insertion

Le tri par insertion procède de la même façon qu'un joueur de cartes pour trier ses cartes, en insérant successivement chaque élément du tableau dans un sous-tableau déjà trié : à la  $i^{eme}$  étape, les  $i - 1$  premiers éléments du tableau sont déjà triés et on insère le  $i^{eme}$  élément du tableau dans ce sous-tableau trié.

### Algorithme 47 : Tri par insertion

```

1 Procédure tri_insertion( $A, n$ )
    Déclaration : entier  $i$ 
2   début
3     pour ( $i \leftarrow 2 ; i \leq n ; i \leftarrow i + 1$ ) faire
4       pour ( $k \leftarrow i ; (k > 1) \text{ and } (A[k] < A[k - 1]) ; k \leftarrow k - 1$ ) faire
5         echanger( $A[k], A[k - 1]$ )
          // invariant :  $A[1..i]$  est trié

```

**Exemple :** l'exécution de cet algorithme sur le vecteur d'entrée  $(26, -14, 17, 34, 0, 18)$  est présentée dans la figure 2.2. Chaque exécution de la boucle 3 de l'algorithme 47 correspond à l'une des états décrits dans la figure 2.2 (1 à

7).

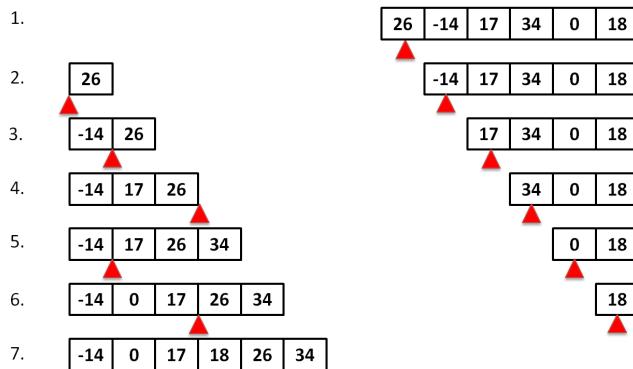


FIGURE 2.2 – Tri par insertion : exemple

**Complexité :** Pour trier ce tableau, on passera  $n - 1$  fois dans la boucle 3. A chaque passage dans cette boucle (itération  $i$ ), on exécute au maximum  $i$  fois la procédure *echanger*, qui à un coût constant. Or, au premier passage dans la boucle 3, nous appelons 1 fois *echanger*, au deuxième passage, il en a 2 appels, au troisième il en a 3, ..., au  $n - 1^{eme}$  passage il en a  $n - 1$ . Ainsi, le nombre total d'opérations à effectuer pour trier un tableau comportant  $n$  éléments sera de l'ordre de  $1 + 2 + 3 + \dots + n - 1 = n(n - 1)/2$ . Par conséquent, la complexité de ce tri est en  $\mathcal{O}(n^2)$ .

**Mémoire :** Cet algorithme réalise un tri sur place ( $\mathcal{O}(1)$ ).

## 2.3 Tri à bulles (Bubble sort)

Le principe de cet algorithme de tri : faire remonter progressivement les plus grands éléments d'un tableau, comme les bulles d'air remontent à la surface d'un liquide.

---

### Algorithme 48 : Tri à bulles

---

```

1 Procédure tri_bulles( $A, n$ )
    Déclaration : entier  $i, j$ 
                  logique  $fin$ 
2   début
3     pour ( $i \leftarrow 1 ; i \leq n ; i \leftarrow i + 1$ ) faire
4        $fin \leftarrow vrai$ 
5       pour ( $j \leftarrow n ; j \geq i + 1 ; j \leftarrow j - 1$ ) faire
6         si ( $A[j] < A[j - 1]$ ) alors
7            $fin \leftarrow faux$ 
8           echanger( $A[j], A[j - 1]$ )
9         // invariant :  $A[1] \dots A[i]$  sont à leur place finale
10        si ( $fin$ ) alors
11          break

```

---

**Exemple :** l'exécution de cet algorithme sur le vecteur d'entrée  $(26, -14, 17, 34, 0, 18)$  est présentée dans la figure 2.3. Chaque permutation de deux éléments (ligne 8 de l'algorithme 47) correspond à l'une des états décrits dans la figure 2.3 (1 à 8).

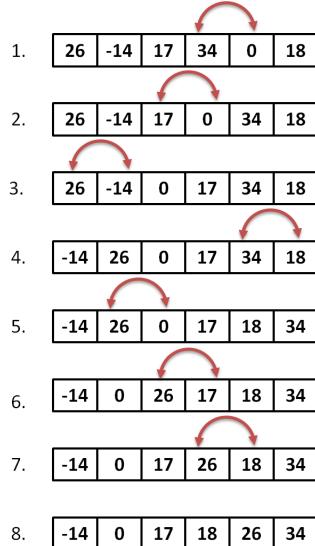


FIGURE 2.3 – Tri à bulles : exemple

**Complexité :** Pour trier le tableau en entrée, on passera  $n$  fois dans la boucle 3. A chaque passage dans cette boucle (itération  $i$ ), on passe  $n - i$  fois dans la boucle 5. La procédure *echanger* a un coût constant (disons 1, pour simplifier). Chaque passage par la boucle 5 a un coût maximal de 5 (exécutions des lignes 6 – 10, chaque ligne avec un cout de 1) et un coût minimal de 2 (les lignes 6 et 9). Ainsi, le nombre total d'opérations à effectuer pour trier un tableau comportant  $n$  éléments sera au maximum  $\sum_{i=1}^n (1 + 5 * (n - i))$  et au minimum  $\sum_{i=1}^n (1 + 2 * (n - i))$ . Par conséquent, la complexité de ce tri est en  $\Theta(n^2)$ .

**Mémoire :** Cet algorithme réalise un tri sur place ( $\mathcal{O}(1)$ ).

## 2.4 Tri rapide (Quicksort)

Le tri rapide est un exemple de tri par dichotomie. L'idée est de récursivement :

1. partitionner le tableau à trier en deux sous-tableaux tels que tous les éléments du premier sous-tableau soient inférieurs à tous les éléments du second tableau,
2. recommencer ce processus sur chacun des sous-tableaux

jusqu'à ce que les sous-tableaux à trier ne contiennent plus qu'un seul élément. Cet algorithme principal est décrit en Algorithme 49.

La procédure *partition* est la clé du tri rapide.

**Partition d'un tableau :** il s'agit de permuter les éléments d'un tableau de telle sorte que tous les éléments du tableau dont la valeur est supérieure à une valeur pivot donnée soient placés à la fin du tableau.

Analyse de la complexité de partition : pour partitionner un tableau comportant  $n$  éléments, il faudra comparer successivement le pivot avec chacun des éléments du tableau. On fera donc de l'ordre de  $\mathcal{O}(n)$  comparaisons. Par ailleurs,

**Algorithme 49 : Quicksort : l'algorithme principal**

```

1 Procédure tri_rapide( $A, d, f$ )
    Déclaration : entier  $pos$ 
    début
        si  $d < f$  alors
             $pos = \text{partition}(A, d, f)$ 
            //  $\forall i \in [d..pos - 1], A[i] \leq A[pos]$  et  $\forall i \in [pos + 1..f], A[i] > A[pos]$ 
            tri_rapide( $A, d, pos - 1$ )
            tri_rapide( $A, pos + 1, f$ )

```

```

1 Fonction entier partition( $A, d, f$ )
    Déclaration : entier  $pivotIndex, i$ 
    début
        // choix du pivot
        echanger( $A[d], A[\text{rand}(d, f)]$ )
        // garder le pivot dans le premier élément
         $pivotIndex \leftarrow d$ 
        // partition
        pour ( $i \leftarrow d + 1; i \leq f; i \leftarrow i + 1$ ) faire
            si  $A[i] < A[d]$  alors
                 $pivotIndex \leftarrow pivotIndex + 1$ 
                echanger( $A[pivotIndex], A[i]$ )
        // positionner le pivot à sa place
        echanger( $A[pivotIndex], A[d]$ )
    retourner  $pivotIndex$ 

```

dans le pire des cas (c'est à dire si la moitié des éléments sont supérieurs au pivot, et que tous ces éléments se trouvent au début du tableau), il faudra faire  $n/2$  échanges. Par conséquent, la complexité totale de partition est en  $\mathcal{O}(n)$ .

**Etude de la complexité en moyenne et dans le pire des cas du tri rapide :** Un appel à la procédure tri\_rapide avec un tableau de  $n$  éléments engendre :

- un appel à la procédure partition (de complexité en  $\mathcal{O}(n)$ ),
- un appel récursif à la procédure tri\_rapide avec un sous-tableau de  $k$  éléments (avec  $k = pos - d$ ),
- un deuxième appel récursif à la procédure tri\_rapide avec un sous-tableau de  $n - k - 1$  éléments.

*Dans le pire des cas*, à chaque appel récursif,  $k = 0$  (c'est à dire que le tableau est déjà trié, ce qui fait que le pivot est toujours le plus petit élément du sous-tableau considéré). Dans ce cas extrême, l'appel à tri\_rapide avec un tableau de  $n$  éléments engendre un appel à tri\_rapide avec un tableau de  $n - 1$  éléments, qui lui-même engendre un appel à tri\_rapide avec un tableau de  $n - 2$  éléments, ... etc ... Chaque appel nécessitant un appel à partition dont la complexité est linéaire en fonction de la taille du tableau, la complexité totale est de l'ordre de  $n + n - 1 + n - 2 + \dots + 1 = n(n + 1)/2 = \mathcal{O}(n^2)$ .

En revanche, si l'on considère *le cas moyen*, lors des appels récursifs à la procédure tri\_rapide, la taille des 2 sous-tableaux est relativement équilibrée. Dans ce cas, l'appel à tri\_rapide avec un tableau de  $n$  éléments engendre deux appels à tri\_rapide avec des tableaux de  $n/2$  éléments, qui eux-mêmes engendrent des appels à tri\_rapide avec des

tableaux de  $n/4$  éléments, ... etc ... Chaque appel nécessitant un appel à partition dont la complexité est linéaire en fonction de la taille du tableau, la complexité totale est de l'ordre de  $n + 2*n/2 + 2^2*n/2^2 + 2^3*n/2^3 + \dots + 2^k*n/2^k$  avec  $k = \log n$ . Par conséquent, la complexité en moyenne du tri rapide est en  $\mathcal{O}(n * \log n)$ .

Pour garantir que le tri rapide soit effectivement rapide, il est donc important de bien choisir le pivot à chaque partition.

Pour cela, on peut tirer aléatoirement trois valeurs d'indices  $i_1, i_2$  et  $i_3$  comprises entre  $d$  et  $f$  et choisir comme pivot non plus l'élément d'indice  $d$ , mais l'élément médian de  $\{A[i_1], A[i_2], A[i_3]\}$ .

## 2.5 Rangs

Un problème similaire au tri est la sélection du  $k_{me}$  plus petit élément dans un tableau de taille  $n$ . Cet élément est aussi appelé le  $k_{me}$  rang statistique. Des cas particuliers sont le minimum ( $k = 1$ ) et le maximum ( $k = n$ ).

Un algorithme très similaire au *quicksort* est disponible pour trouver le  $k_{me}$  rang statistique : Algorithme 50.

---

**Algorithme 50 :** Algorithme pour la sélection rapide, similaire au *quicksort*.

---

1 **Fonction** entier *selection\_rapide*( $A, d, f, k$ )

    Déclaration : entier *pos*

    début

        si  $d = f$  alors

            | retourner *d*

*pos* = partition( $A, d, f$ )

        si  $k = pos$  alors

            | retourner *k*

        sinon si  $k < pos$  alors

            | *selection\_rapide*( $A, d, pos - 1$ )

        sinon

            | *selection\_rapide*( $A, pos + 1, f$ )

---

Pour plus d'information sur les algorithmes de tri et de sélection : *Introduction to algorithms*, Cormen, Leiserson, Rivest, Stein, MIT press, 2009.

## CHAPITRE 3

## STRUCTURES DE DONNÉES

### 3.1 Structures séquentielles : des limitations

Les structures de données dites séquentielles (de type tableau) permettent de répondre à diverses problématiques algorithmiques. Elles ont le gros avantage de procurer un temps d'accès à chaque élément en un temps constant. Malheureusement, elles possèdent aussi des limites :

1. le nombre d'éléments que l'on peut stocker est en général fixe ;
2. chaque élément constituant possède le même type, on ne peut pas mélanger des types différents.

La première limitation est assez gênante car elle oblige à savoir avant l'exécution du programme combien d'éléments un tableau devra posséder. Cette première limitation est levée par *l'allocation dynamique* qui permet au cours de l'exécution d'un programme de demander au système une zone mémoire. Contrairement aux variables classiques d'un programme, cette zone mémoire ne sera pas libérée automatiquement à la fin de l'exécution, c'est à l'utilisateur de faire une libération explicite. L'allocation dynamique est expliquée en détail dans la section suivante.

La deuxième limitation est levée par la définition de *structures de données*. Ces types de données ont la particularité de pouvoir contenir d'autres sous-types dans un même conteneur. Les structures de données sont expliquées en détail dans la section 3.3 de ce chapitre. Elles servent de base à la construction des structures de données élémentaires que sont les piles, files, listes chaînées, arbres et graphes qui seront développés dans les sections suivantes.

### 3.2 Allocation dynamique

#### ► Variables statiques

Une variable classique allouée de manière statique possède un nom, un type, est stockée à une adresse et a une valeur. C'est le compilateur qui automatise la création de cette variable *au sein même de l'espace de l'exécutable*. Une adresse lui est donc réservée dans un des segments mémoire de l'exécutable.

Ce type d'allocation a l'avantage d'être rapide, car c'est le compilateur qui va aller chercher de manière relative l'emplacement de la variable au sein de l'exécutable. Tout est donc décidé au moment de la compilation : on ne perd pas de temps au moment de l'exécution à allouer des variables. Ce type d'allocation est très limité car il ne concerne que les constantes d'un programme où les variables dites statiques c'est-à-dire dont on est sûr qu'elles ne doivent être allouées qu'une seule fois. Les compilateurs modernes limitent en général l'utilisation de ces allocations aux constantes (exemple : une chaîne de caractères non affectée à une variable) ou quasi-constantes (variables dont la valeur peut être déterminée avant l'exécution).

## ► Variables dynamiques sur la pile

Par définition, on ne peut pas savoir combien de fois une variable déclarée dans une fonction devra être allouée. De plus, à la fin de l'exécution de la fonction, la variable doit être libérée car elle ne sert plus à rien. Dans ce cas-là le compilateur va recourir à l'utilisation de variables dynamiques et va automatiser la procédure d'allocation/libération. Dès lors qu'une variable est déclarée dans un contexte de fonction, le compilateur va générer automatiquement une *allocation dynamique sur la pile* de cette variable. L'allocation est faite au début de l'exécution de la fonction et la libération est faite à la fin. Le segment de mémoire utilisé pour ces allocations est appelé le *segment de pile*.

## ► Variables dynamiques sur le tas

C'est l'allocation qui donne le plus de contrôle à l'utilisateur sur la durée de vie de la variable. En effet, c'est lui qui décide de manière explicite du moment où la variable va être allouée, et du moment où la variable va être libérée. Ce mécanisme ne peut donc pas être effectué par une déclaration de variable standard.

La figure 3.1 résume les trois types d'allocation possibles. L'allocation dynamique sur le tas est le moyen le plus fin pour contrôler la façon dont va être allouée la mémoire mais elle oblige à allouer et libérer explicitement.

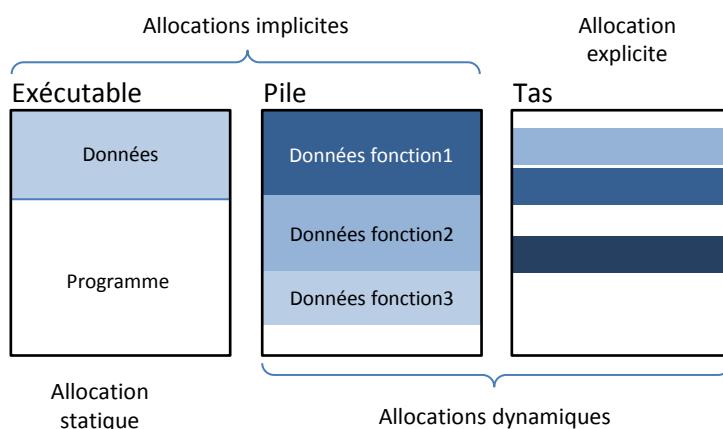


FIGURE 3.1 – L'allocation statique se fait directement dans l'espace mémoire du programme (gauche). L'allocation dynamique sur le segment de pile se fait au fur et à mesure des appels de fonctions du programme et est libérée automatiquement à la fin de ces appels (centre). Dans l'exemple, fonction1 a appelé fonction2 qui a elle-même appelé fonction3. L'allocation dynamique sur le tas se fait dans un ordre déterminé par l'utilisateur et peut laisser des "trous" dans le segment de mémoire (droite). Alors que les deux premiers types d'allocation sont implicites (faits de manière automatique), le troisième est explicite et nécessite une action de l'utilisateur.

## ► Fonctionnement

Étant donnée la nature même de l'allocation dynamique<sup>1</sup>, la variable qui va stocker la valeur ne peut plus être déclarée de la même manière. Au lieu de cela, nous allons déclarer une variable qui va stocker l'adresse de l'élément alloué, on appelle cette variable *un pointeur*. Cela signifie que pour accéder au contenu de la variable dynamique il faudra aller chercher l'emplacement mémoire associé, cette opération s'appelle déréférencement ou indirection. En notation algorithmique comme en langage C ou C++, l'opérateur d'indirection sera noté \* et sera préfixé.

Il y a donc quatre étapes dans l'allocation dynamique (voir la figure 3.2) :

1. Déclaration d'une variable recevant l'adresse de la variable dynamique

1. dans la suite du document, on parlera d'allocation dynamique sans préciser *sur le tas*

2. Allocation. L'appel à une fonction du système est nécessaire pour cette opération, en langage algorithmique, cet appel sera fait avec la fonction Allouer() dont l'argument est le type que l'on souhaite allouer.
3. Utilisation. Pour utiliser la variable, il faudra déréférencer le pointeur.
4. Libération. L'appel à une fonction dédiée du système sera aussi nécessaire. En langage algorithmique, ce sera la fonction Libérer() dont l'argument est le pointeur à libérer.

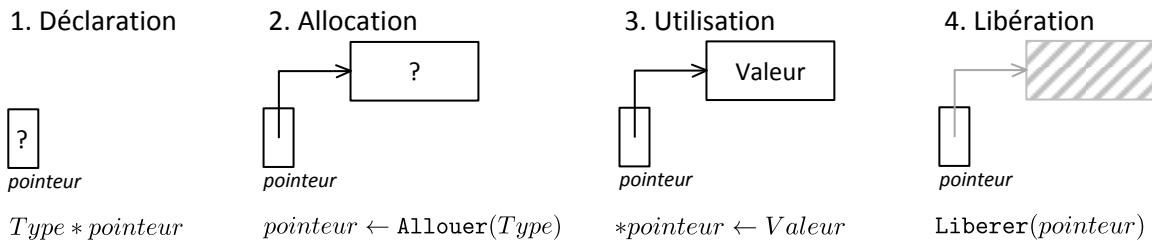


FIGURE 3.2 – L'allocation dynamique se fait en quatre étapes. À l'issue de la libération, la variable contenant le pointeur existe encore et peut être source de problèmes si on l'utilise.

**Attention :** Alors que l'allocation implicite garantit que la durée de vie de la variable est compatible avec son utilisation, l'allocation dynamique laisse un contrôle total à l'utilisateur. Cela signifie par exemple qu'un pointeur dont la valeur est une zone mémoire déjà libérée peut être déréférencé. La conséquence est sans appel lors de l'exécution d'un tel programme sur un système muni d'une vérification de la mémoire : le programme prend fin (il plante !).



## ► En pratique

En langage C, l'allocation dynamique se fait grâce aux fonctions malloc() et free() :

1. malloc() permet d'allouer une zone mémoire et prend en argument le nombre d'octets à allouer. Il est souvent combiné avec l'opérateur sizeof() qui permet d'extraire la taille qu'occupe un type ou une variable en mémoire. Cette fonction renvoie un pointeur vers la zone allouée ou 0 si l'allocation n'a pas pu se faire (comportement différent suivant les systèmes).<sup>2</sup>
2. free() permet de libérer une zone mémoire (qui doit obligatoirement être allouée avec malloc()) et prend en argument le pointeur de la zone mémoire à libérer.

La déclaration et l'utilisation se font de manière semblable à ce que l'on fait en langage algorithmique. Voici un exemple :

```
#include <stdlib.h>
Type * pointeur;
pointeur = (Type*) malloc(sizeof(Type));
if (pointeur)
{
    *pointeur = valeur;
    free(pointeur);
}
```

2. la fonction malloc() renvoie un pointeur anonyme de type void\*, pour éviter un avertissement du compilateur, on pourra forcer ce retour à un autre type de pointeur.

## ► Retenir sur les pointeurs

La notation que nous utiliserons pour les pointeurs en langage algorithmique est identique à celle des langages C et C++ et permet d'avoir une cohérence globale entre les déclarations et l'opérateur d'indirection. Prenons la déclaration suivante :

Type \* *pointeur*

Cette déclaration nous dit que *pointeur* est de type (Type \*). Maintenant, si nous coupions avant l'étoile, nous obtenons que \**pointeur* est de type Type ce qui est bien compatible avec l'opérateur d'indirection \*. Ce principe est applicable même si plusieurs \* sont utilisées.

### Mémo

1. L'allocation dynamique sur le tas est le seul moyen de contrôler le cycle de vie d'une variable.
2. Ce contrôle a un coût de programmation puisqu'il nécessite plus d'étapes et de précautions.
3. Le segment de mémoire dans lesquels sont stockées les variables allouées dynamiquement est séparé des autres segments.
4. L'opération de déréférencement ou indirection est faite grâce à l'opérateur \* et permet d'accéder au contenu d'une variable à partir de son pointeur.

**Exercice.** Représenter les états successifs de la mémoire pour l'algorithme suivant :

```

entier * a
entier * b
a ← Allouer(entier)
b ← a
*a ← 1
b ← Allouer(entier)
*b ← *a
Libérer(a)
Libérer(b)
    
```

## 3.3 Structures

Afin de pouvoir grouper au sein d'une même variable plusieurs sous-variables de types différents, on a recours aux structures parfois appelées aussi enregistrements.

### Définition

Une structure est un type  $T$  défini comme la juxtaposition (y compris en mémoire) de plusieurs variables de types différents ( $v_i, T_i$ ). Ces variables qui composent la structures sont appelées membres. Les  $v_i$  sont les noms ou identificateurs de la variable, c'est eux qui vont permettre d'y accéder.

## ► En langage algorithmique

Voici la syntaxe de déclaration et utilisation d'une structure en langage algorithmique :

```
// Déclaration de la structure
Structure T
  T1 v1
  T2 v2
  ...
  Tn vn
// Déclaration d'une variable du type T
T v
// Utilisation
v.v1 ← valeur
```

On accède donc au contenu des membres grâce à l'opérateur ".".

## ► Exemple

On peut imaginer que l'on stocke une date sous la forme d'une chaîne de caractères. Le problème est que pour faire des traitements, il faut analyser cette chaîne de caractères pour en extraire les informations. Afin de disposer directement de ces informations dans un seul type, on va définir une structure de données dans laquelle se trouve l'année, le mois et le jour sous la forme de trois entiers<sup>3</sup> :

```
Structure Date
  entier année
  entier mois
  entier jour
```

Quelques exemples d'utilisation de cette structure :

```
Date date
date.annee ← 2014
date.mois ← 9
date.jour ← 24
date ← { 2014, 09, 24 }
si (date.annee mod 4 = 0 et date.annee mod 100 ≠ 0)
ou date.annee mod 400 = 0 alors
  Afficher("Année bissextile")
```

## ► Avec allocation dynamique

Il est très courant d'utiliser les structures avec l'allocation dynamique. Cela n'a rien de compliqué, il suffit d'indiquer que l'on veut allouer le type correspondant à la structure :

3. Quelle est la différence entre cette solution et un tableau de 3 entiers ?

```
T * pointeur
pointeur ← Allouer(T)
(*pointeur).v1 ← valeur
pointeur → v1 ← valeur
```

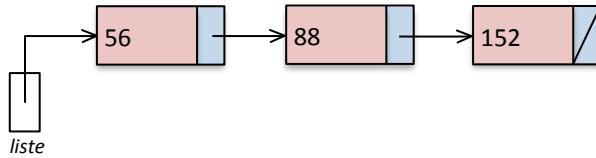
L'opérateur  $\rightarrow$  permet de faire deux choses en même temps : l'indirection nécessaire pour aller chercher la variable pointée par l'adresse, et l'opérateur de membre `".".`. C'est un raccourci que l'on utilise tout le temps et qui existe aussi dans les langages C et C++.

### ► Auto-référence

Un membre d'une structure peut contenir n'importe quel type de données, y compris un pointeur. Ce pointeur peut être l'adresse d'une structure, voire de la même structure... C'est un mécanisme que nous allons utiliser fréquemment dans la suite de ce chapitre car il permet de lier des éléments entre eux et former ainsi une structuration topologique de la mémoire. L'exemple le plus classique est la liste chaînée :

```
Structure Liste
entier valeur
Liste * suivant
```

Chaque élément de la structure possède une information sous forme d'entier, mais aussi une autre information indiquant l'élément suivant dans la liste. Pour indiquer qu'il n'y a pas de suivant, on utilise le pointeur nul. La représentation mémoire d'une liste chaînée ressemble à cela :



Les membres *valeur* ont été coloriés en rouge alors que les membres *suivant* en bleu. *liste* est une variable qui stocke le pointeur vers le début de la liste, c'est-à-dire vers la première cellule de cette liste. Les trois cellules ont forcément été allouée dynamiquement. Dans cet exemple précis, nous avons les propriétés suivantes :

```

liste ≠ ∅
liste → valeur = 56
liste → suivant ≠ ∅
liste → suivant → valeur = 88
liste → suivant → suivant ≠ ∅
liste → suivant → suivant → valeur = 152
liste → suivant → suivant → suivant = ∅
liste(→ suivant)3 = ∅
  
```

On notera au passage qu'il faut bien vérifier qu'un pointeur n'est pas nul avant d'effectuer une indirection (ou l'opérateur  $\rightarrow$ ) dessus sous peine de s'exposer à de gros problèmes de protection de la mémoire.

Cette structuration de la mémoire permet de représenter la suite d'entiers (56, 88, 152). Par rapport à un tableau :

1. Elle est plus coûteuse en terme de stockage car nécessite l'utilisation d'un pointeur pour chaque élément.

2. Elle est plus flexible pour l'ajout ou la suppression d'éléments en début ou milieu de liste.
3. Elle nécessite l'utilisation de l'allocation dynamique.
4. Elle est moins compacte en mémoire car deux éléments consécutifs peuvent se trouver à des endroits complètement différents de la mémoire (peut engendrer des *cache miss* : perte d'optimisation de la mémoire).

Il ne faut donc pas croire que l'utilisation d'une liste chaînée est la solution idéale à tous les problèmes, mais bien faire attention à tous ces facteurs. Par exemple, si on ne souhaite faire que des insertions en fin de suite, un tableau peut être plus performant.

## ► En langage C

En langage C, la déclaration d'une structure se fait grâce au mot-clé `struct`.

La syntaxe est la suivante :

```
struct NomStructure
{
    type1 membre1;
    type2 membre2;
    /* etc. */
};
```

Cette syntaxe permet d'utiliser ensuite la structure comme un type en répétant le mot-clé `struct` :

```
struct NomStructure variable;
```

Si on veut vraiment définir un nouveau type sans avoir à mettre le mot-clé, il faut utiliser le mot-clé `typedef` en plus :

```
typedef struct NomStructure TypeStructure;
TypeStructure variable;
```

Il est recommandé d'utiliser deux identifiants différents pour la structure et le type. Pour une structure qui s'autoréférence, on peut procéder de la manière suivante :

```
struct StructListe
{
    int valeur;
    struct StructListe * suivant;
};
```

### Mémo

1. Les structures permettent de regrouper dans un seul type plusieurs variables qui peuvent avoir des types différents.
2. Chacune de ces variables est appelée membre et on y accède grâce à l'opérateur `."` suivi du nom de membre.
3. Les structures sont compatibles avec l'utilisation de l'allocation dynamique.
4. Une structure peut faire référence à un autre élément d'une structure grâce à un de ses membres qui sera de type pointeur. Ces pointeurs ont un coût de stockage qu'il faut intégrer dans le choix d'une représentation pour un problème donné.

## CHAPITRE 4

### BRIQUES DE BASE

Les briques de base (les façons d'organiser les données en mémoire) que l'on va utiliser dans la suite du cours : tableau statique, tableau dynamique, liste chaînée et doublement chaînée, arbre et tas binaires.

#### 4.1 Le tableau statique

Le tableau statique est un tableau dont le nombre d'éléments est déterminé avant l'exécution du programme, c'est-à-dire au moment de la compilation de l'exécutable. Le coût de modification ou lecture d'un élément de tableau est constant car l'accès aux éléments est direct par l'indice de l'élément. Par contre, dès lors que l'on veut insérer un élément, deux cas de figures se présentent : insertion en fin de tableau (coût constant) et insertion en début ou milieu de tableau (coût linéaire en fonction du nombre d'éléments à déplacer). L'autre limitation du tableau statique est qu'on ne pourra pas modifier le nombre d'éléments maximum. Inversement, lorsque les limitations mentionnées ici ne sont pas un problème pour l'application visée, il est très recommandé d'utiliser le tableau statique.

#### 4.2 Le tableau dynamique

Un tableau dynamique est un tableau alloué dynamiquement. Plusieurs méthodes de gestion existent :

1. Gestion quasi-statique : le tableau est alloué dynamiquement en début d'exécution du programme en fonction des paramètres d'exécution et ne sera jamais réalloué. Les caractéristiques de ce type de tableau sont quasiment identiques à celles du tableau statique.
2. Gestion dynamique de base : le tableau est initialement alloué avec une taille faible. Lorsque la taille est amenée au cours de l'exécution à être plus élevée, une réallocation est effectuée correspondant exactement à la taille nécessaire. Chaque réallocation nécessite la recopie de tous les éléments présents dans le tableau.
3. Gestion dynamique avec politique d'allocation : lorsque la taille du tableau est amenée à évoluer pendant l'exécution du programme, une réallocation est effectuée avec une politique qui fait en sorte d'optimiser le nombre de changements de taille. Une politique standard consiste à doubler la taille à chaque réallocation.

Les politiques de réallocation sont un facteur important de l'efficacité du programme. En effet, alors que l'ajout d'un élément à la fin d'un tableau a un coût constant lorsqu'il n'y a pas d'allocation à faire, ce coût devient linéaire s'il y a une réallocation à faire (car copie).

La figure 4.1 résume les méthodes de gestion de tableaux dynamiques.

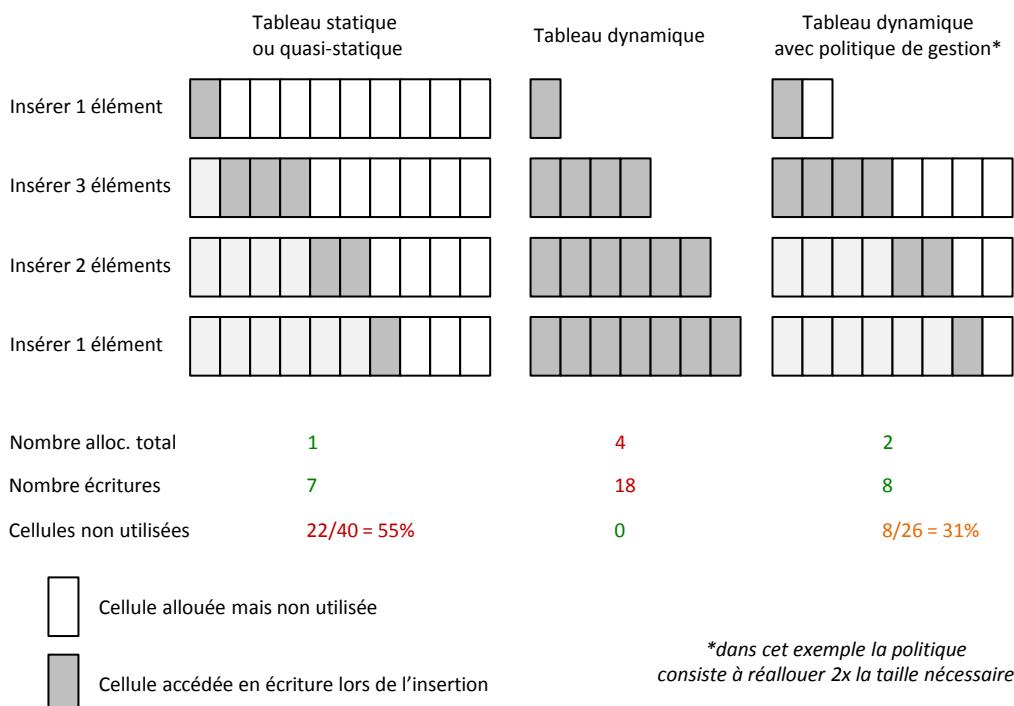


FIGURE 4.1 – Les différentes façons de gérer un tableau dynamique mènent à un compromis entre coût d'insertion en nombre d'écritures et en nombre d'allocation et quantité de mémoire inutilisée donc gaspillée. Si les tableaux statiques sont très performants, ils ont l'inconvénient de gaspiller de la place. Les tableaux dynamiques "sur mesure" sont quant à eux peu performant, mais ne gaspillent pas de mémoire. Il existe des compromis consistant à appliquer une politique de gestion qui fera que chaque ajout ne donnera pas lieu à une réallocation.

## 4.3 Liste simplement chaînée

La liste simplement chaînée consiste à créer une structure qui combine une valeur à stocker avec un pointeur vers la cellule suivante. Lorsque la fin de la suite d'élément est atteinte, le pointeur vers le suivant est positionné à nul. La liste chaînée comme toute structure non compacte perd de la mémoire à cause du stockage des pointeurs. Cette perte est directement proportionnelle au nombre d'éléments. La taille de stockage d'un pointeur dépend du système sur lequel va être implémenté la solution. Les systèmes 32 bits stockent leurs pointeurs sur 4 octets tandis que les systèmes 64 bits les stockent sur 8 octets. Si on rapporte ce chiffre à l'information qui doit être stockée pour chaque élément cela peut devenir une surcharge importante. Exemple : le stockage d'une liste chaînée contenant des entiers codés sur 4 octets gaspillera la moitié de la mémoire à la gestion de la liste sur un système 32 bits et les deux tiers sur un système 64 bits... À quoi bon utiliser les listes chaînées alors ? Tout simplement parce qu'une insertion ou une suppression en tête ou en milieu de liste chaînée est bien moins coûteuse que dans un tableau.

Concrètement, pour accéder à un élément d'une liste chaînée, il faut parcourir tous les éléments qui se trouvent avant. En revanche, une fois cet élément trouvé, sa modification, suppression, ou l'ajout d'un autre élément ont un coût constant. Dans ce cadre-là, l'insertion ou la suppression en début de liste sont donc très peu coûteux (temps constant). L'ajout ou la suppression en fin de liste est par contre coûteuse car oblige à parcourir toute la liste. C'est pourquoi en plus du pointeur sur le début de la liste, on peut stocker un pointeur sur la fin de liste. La figure 4.2 montre les différents cas de figure d'une insertion dans une liste chaînée. L'insertion à la fin de la liste est très coûteuse sauf si on avait dès le départ le pointeur sur la fin de la liste.

Suivant les contraintes que l'on va mettre sur les éléments, la liste des opérations que l'on va pouvoir effectuer ne sera pas la même. Par exemple, si les éléments ne sont pas triés, l'insertion n'a de sens qu'en tête ou en queue de liste.

### ► Insertion en tête

Etant donné qu'une liste chaînée possède un point d'entrée sur son premier élément, l'insertion en tête est triviale. Il suffit pour cela de changer le premier élément et de le faire pointer sur l'ancienne tête. L'algorithme 51 donne l'algorithme de cette opération.

---

#### **Algorithme 51 :** Algorithme d'insertion en tête dans une liste simplement chainée

---

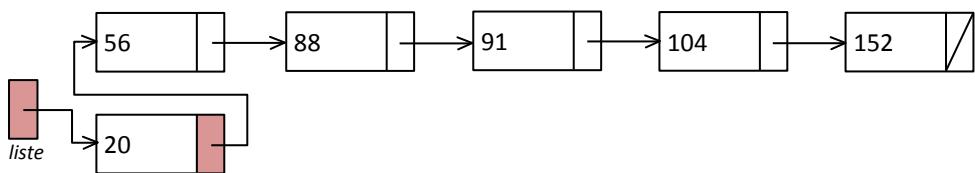
```

1 Structure Liste
2   entier valeur
3   Liste * suivant
4 Fonction InsereListeTete(nouvelleVal, liste)
   Entrée      : Liste * liste
                  entier nouvelleVal
   Sortie       : renvoie le nouveau pointeur de début de liste
   Précondition : La liste est bien construite, elle peut être vide, dans ce cas-là le pointeur est nul
   Postcondition : Le pointeur renvoyé correspond à la nouvelle liste chaînée avec l'élément inséré en tête.
   Déclaration  : Liste * nouveau
5   nouveau ← Allouer(Liste)
6   nouveau → valeur ← nouvelleVal
7   nouveau → suivant ← liste
8   retourne nouveau
```

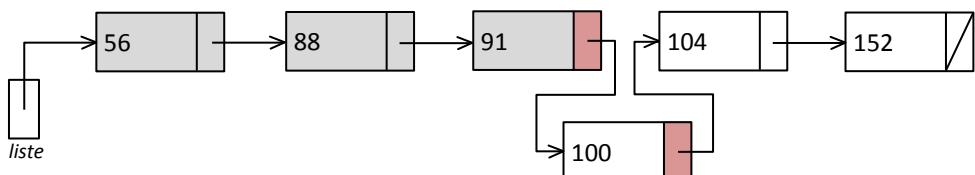
---

**Insertion en tête avec maintien d'un pointeur sur la fin de liste.** Laissé en exercice (attention, c'est plus compliqué que le cas précédent).

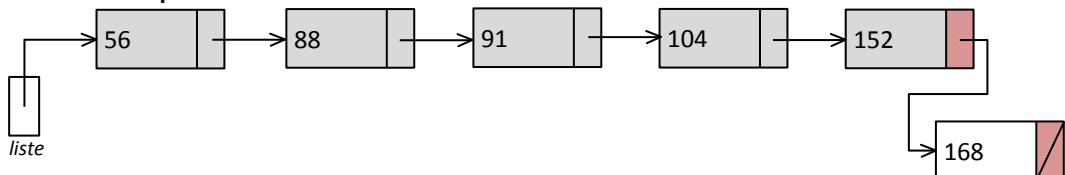
**Insertion en tête**



**Insertion en milieu**



**Insertion en queue**



**Insertion en queue avec information de fin de liste**

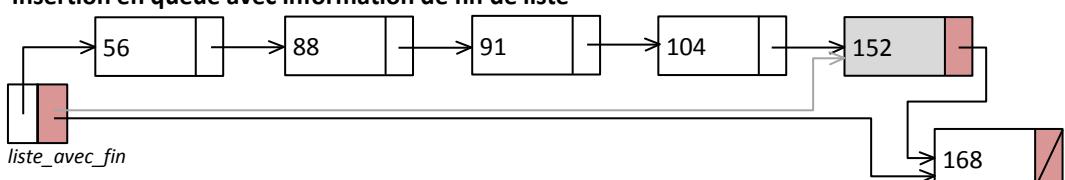


FIGURE 4.2 – Une liste simplement chaînée est coûteuse pour l'insertion lorsque l'élément se trouve à la fin. Cet inconvénient peut être évité grâce à l'ajout d'un pointeur de fin de liste.

## ► Insertion en queue

L'insertion en queue est coûteuse sauf si l'on possède l'information du pointeur sur le dernier élément. Si cette information est disponible, la seule difficulté est dans la gestion de tous les cas de figure (liste vide, liste à un seul élément, liste à plus d'éléments<sup>1</sup>). En voici le détail :

1. Liste vide (*premier* et *dernier* sont tous les deux nuls) : dans ce cas, la nouvelle liste est constituée d'un seul élément et *premier* et *dernier* pointent tous les deux sur celui-ci.
2. Liste comprenant un seul élément (*premier* et *dernier* pointent sur le même élément) : dans ce cas, le nouvel élément est ajouté à la fin et *dernier* pointera sur celui-ci. *premier* n'est pas modifié.
3. Liste de 2 ou plus éléments (*premier* *dernier* pointent sur des éléments différents) : dans ce cas, le nouvel élément est ajouté à la fin et *dernier* pointera sur celui-ci. *premier* n'est pas modifié.

On constate rapidement que certaines opérations sont communes entre les cas de figures. En effet, l'élément inséré sera toujours le nouveau pointeur *dernier*, et ce quel que soit le cas. De plus, les cas 2 et 3 n'ont aucune différence entre eux, ce n'est pas la peine de les différencier dans l'algorithme. Seul le cas 1 est légèrement différent car il modifie *premier*. L'algorithme 52 montre une implémentation naïve de l'algorithme ne tenant pas compte de cette analyse. Cet algorithme fait apparaître de façon flagrante qu'une portion de code est *factorisable*, c'est-à-dire qu'elle est exécutée quel que soit le cas de figure. Cela permet de simplifier grandement l'algorithme pour arriver simplement à modifier *premier* que lorsque la liste est vide et à lui affecter le pointeur vers la nouvelle cellule dans ce cas de figure. La modification de *dernier* est tout le temps la même :

```
si premier = Ø alors
    premier ← nouveau
    dernier ← nouveau
```

## ► Insertion positionnée

Lorsque la liste est triée, on spécifie que l'on veut insérer un élément mais on ne donne pas sa position. Cette dernière va être déduite de la façon de placer la nouvelle valeur par rapport à celles déjà présentes dans la liste. L'algorithme 53 donne le détail de la fonction d'insertion d'un élément dans une liste simplement chaînée sans pointeur sur la fin de liste.

## ► Preuve de correction

Afin de prouver que l'algorithme d'insertion est bien correct, nous allons utiliser la méthode des triplets de HOARE. Un triplet de HOARE est défini par  $\{P\}$  inst  $\{Q\}$ .  $P$  est une pré-condition,  $Q$  est une post-condition, tandis que inst est une séquence d'instructions. Le triplet est vérifié si on peut démontrer que  $Q$  est vrai lorsque  $P$  est vrai et que les instructions sont exécutées.

Voici quelques exemples de triplets faciles à vérifier :

$$\begin{aligned} &\{x > 0\} \ x \leftarrow x + 1 \ \{x > 1\} \\ &\{a = 43\} \ b \leftarrow a \ \{b = 43\} \end{aligned}$$

Un algorithme complet est prouvé si ses pré-conditions engendrent les post-conditions quand on l'exécute. On voit bien ici l'importance de la formalisation des pré et post-conditions. D'une manière générale, on décompose le triplet du programme en une suite de triplets qui se suivent grâce à la règle suivante. Soit  $I = (I_i)_{i=1 \dots n}$  une suite d'instructions. Si tous les triplets  $\{P_i\} \ I_i \ \{P_{i+1}\}$  (avec  $i = 1 \dots n$ ) sont vérifiés (prouvés), alors le triplet  $\{P_1\} \ I \ \{P_{n+1}\}$  est lui aussi vérifié.

1. nous allons volontairement faire l'exercice de séparer les deux derniers cas alors qu'ils peuvent ne pas l'être

**Algorithme 52 :** Algorithme d'insertion en queue dans une liste simplement chaînée avec pointeur de fin (version naïve)

1 **Structure** *Liste*

2    entier *valeur*

3    *Liste* \* *suivant*

4 **Fonction** *InsereListeQueueNaif(nouvelleVal, premier, dernier)*

**Entrée**            : *Liste* \* *premier*

*Liste* \* *dernier*

                      entier *nouvelleVal*

**Sortie**            : *premier* et *dernier* sont en sortie.

**Précondition** : La liste est bien construite, elle peut être vide, dans ce cas-là les pointeurs *premier* et *dernier* sont nuls

**Postcondition** : *premier* et *dernier* sont modifiés de telle sorte que l'élément avec la valeur *nouvelleVal* soit à la fin de la liste.

**Déclaration**    : *Liste* \* *nouveau*

5    *nouveau*  $\leftarrow$  Allouer(*Liste*)

6    *nouveau*  $\rightarrow$  *valeur*  $\leftarrow$  *nouvelleVal*

7    *nouveau*  $\rightarrow$  *suivant*  $\leftarrow$   $\emptyset$

8    **si** *premier* =  $\emptyset$  et *dernier* =  $\emptyset$  **alors**

9        *premier*  $\leftarrow$  *nouveau*

10      *dernier*  $\leftarrow$  *nouveau*

11     **sinon si** *premier* = *dernier* **alors**

12        *dernier*  $\leftarrow$  *nouveau*

13     **sinon**

14        *dernier*  $\leftarrow$  *nouveau*

Il existe d'autres règles de calcul pour faire des preuves formelles que vous verrez dans le cours de Qualité Logicielle. Nous n'aborderons ici qu'une approche intuitive du concept de preuve.

Une première difficulté pour l'analyse de correction de cet algorithme est dans la formalisation des pré et post-conditions. En effet, la phrase *La liste est triée* est claire en français, mais comment la traduire sous la forme de conditions ? Comment exprimer une condition qui doit être valable pour tous les éléments de la liste et faisant intervenir le suivant de la liste ? Nous allons définir la longueur de la liste comme étant  $n$  et représentant le nombre d'éléments non nuls de la liste. Si la liste est vide, nous aurons  $n = 0$ . La vérification du caractère trié de la liste n'a de sens que quand  $n$  est au moins égal à 2. Avec cette notation, nous avons la définition suivante de la liste triée :

$$\text{ordered}(l) \Leftrightarrow \exists n \geq 0, l(\rightarrow \text{suivant})^n = \emptyset \text{ et } \forall i \in [0 \dots n-2], l(\rightarrow \text{suivant})^i \leq l(\rightarrow \text{suivant})^{i+1}$$

qui peut se décliner en trois cas de figures qui s'excluent mutuellement :

$$l = \emptyset$$

$$\text{ou } l \rightarrow \text{suivant} = \emptyset$$

$$\text{ou } (\exists n \geq 2 \text{ tel que } l(\rightarrow \text{suivant})^n = \emptyset \text{ et } \forall i \in [0 \dots n-2], l(\rightarrow \text{suivant})^i \leq l(\rightarrow \text{suivant})^{i+1})$$

Cette définition n'est toujours pas très satisfaisante et sera difficile à utiliser. Pourquoi ne pas tenter une définition récursive ? En effet, une liste est triée si elle est vide ou si la valeur du premier élément est inférieure à celle du second pourvu que la liste composée du second élément soit elle-même triée. Appelons  $v(p)$  la valeur de l'élément  $p$

**Algorithme 53 :** Algorithme d'insertion dans une liste simplement chainée

```

1 Structure Liste
2   entier valeur
3   Liste * suivant

4 Fonction InsereListe(nouvelleVal, liste)
  Entrée      : Liste * liste
                 entier nouvelleVal
  Sortie       : renvoie le nouveau pointeur de début de liste
  Précondition : La liste est déjà triée par ordre croissant des valeurs, elle peut être vide, dans ce cas-là le
                 pointeur est nul
  Postcondition : Le pointeur renvoyé correspond à la nouvelle liste chaînée, triée avec l'élément inséré. Si la
                  valeur était déjà présente, un élément sera inséré quand même créant ainsi un doublon.
  Déclaration   : Liste * nouveau
                  Liste * precedent
                  Liste * courant
  nouveau ← Allouer(Liste)
  nouveau → valeur ← nouvelleVal
  si liste = Ø ou liste → valeur > nouvelleVal alors          /* Insertion en tête */
    nouveau → suivant ← liste
    retourne nouveau
  courant ← liste
  répéter
    precedent ← courant
    courant ← courant → suivant
  jusqu'à courant = Ø ou courant → valeur > nouvelleValeur;
  precedent → suivant ← nouveau
  nouveau → suivant ← courant
  retourne liste

```

et définissons la de cette manière :

$$v(p) = \begin{cases} \infty & \text{si } p = \emptyset \\ p \rightarrow \text{valeur} & \text{sinon} \end{cases}$$

Dans ce cas là, une liste  $l$  est triée si et seulement si :

$$\text{ordered}(l) \Leftrightarrow l = \emptyset \vee (v(l) \leq v(l \rightarrow suivant) \wedge \text{ordered}(l \rightarrow suivant))$$

Cette définition est plus compacte et surtout plus locale. Afin de simplifier les notations, nous introduisons aussi l'opérateur  $\triangleright$  avec la définition suivante :

$$a \triangleright b \Leftrightarrow a \rightarrow \text{suivant} = b$$

Commençons donc par vérifier la première partie de l'algorithme (lignes 7 à 9). Ce test permet de faire une insertion en tête lorsque cela est nécessaire, mais prend aussi le cas de la liste vide. Étant donné que le test du **si** comporte un ou, nous allons traiter les deux cas séparément. Nous devons donc vérifier que si  $liste = \emptyset$ , alors la liste  $nouveau$  qui est renvoyée est bien triée. C'est bien vrai car  $v(nouveau)$  est forcément inférieure à  $v(liste) = v(\emptyset) = \infty$  et que  $nouveau \rightarrow suivant$  est une liste triée puisque vide. Dans le deuxième cas de figure, nous avons aussi  $v(nouveau) = nouvelleValeur \leq v(nouveau \rightarrow suivant) = v(liste)$  et  $\text{ordered}(nouveau \rightarrow suivant) = \text{ordered}(liste)$  qui est vrai.

Nous allons maintenant passer à la deuxième partie de l'algorithme, composée d'une boucle **tantque**. Pour rappel, la vérification qu'une boucle est correcte par la méthode des triplets de HOARE, nécessite de la décomposer de la manière suivante. Au départ, nous possédons cette forme :

```
{ P }
init
tant que condition faire
  corps
  fin
{ Q }
```

que nous transformons en :

```
{ P }
init
{ I }
tant que condition faire
  { I  $\wedge$  condition }
  corps
  { I }
  { I  $\wedge$  non condition }
  fin
{ Q }
```

Il est appelé invariant de boucle. Pour résumer : l'invariant est toujours vrai, que ce soit avant le corps de la boucle ou après le corps de la boucle. Ce qui différencie le fait qu'on soit sorti de la boucle est la condition du **tantque** qui est respectée dans la boucle et non respectée lorsqu'on en est sorti. Au final, nous avons 3 triplets à vérifier, qui sont indépendants les uns des autres.

Dans notre cas de figure, la difficulté réside dans l'utilisation d'une boucle dont le test de sortie se trouve en fin (boucle **répéter** plutôt que **tantque**). Même si algorithmiquement, c'est assez élégant, cela ne facilite pas l'obtention de l'invariant de boucle. Pour remédier à cela, il suffit de la transformer en boucle **tantque** en recopiant le corps de la boucle avant celle-ci. Dans ce cas-là, on peut exprimer l'invariant comme étant tout simplement  $I = \{precedent \triangleright courant\}$ . La vérification cet invariant est assez triviale pourvu que l'on ait bien fait la transformation indiquée précédemment. Malheureusement, elle ne permet pas de conclure que l'algorithme est correct. En effet, à la sortie de la boucle, on a bien les relations qui font que les éléments *precedent*, *nouveau* et *courant* se suivent, mais rien ne garantit que les valeurs associées à ces éléments sont ordonnées. On a uniquement la garantie que  $v(nouveau) \leq v(courant)$  ainsi que  $v(precedent) \leq v(courant)$  mais pourquoi aurait-on  $v(precedent) \leq v(nouveau)$ ? C'est tout simplement parce que l'algorithme a itéré jusqu'à ce qu'il trouve une valeur *v(courant)* qui soit plus grande que *v(nouveau)*. Cela induit qu'à l'itération précédente, cette relation était fausse, or l'itération précédente est représentée par la variable *precedent*. Il faut donc ajouter dans l'invariant de boucle le fait que  $v(nouveau) \geq v(precedent)$  soit :

$$I = \{precedent \triangleright courant \wedge v(nouveau) \geq v(precedent)\}$$

Ce nouvel invariant n'est pas très dur à vérifier. La figure 4.3 montre des éléments de cette validation. L'encart 1 illustre la propagation de la condition d'itération sur la deuxième partie de l'invariant (car  $p \leftarrow c$ ). Pour ce qui est de la première partie de l'invariant, il s'agit d'une propriété de la liste chaînée qui est bien construite en entrée et qui n'a pas été modifiée.

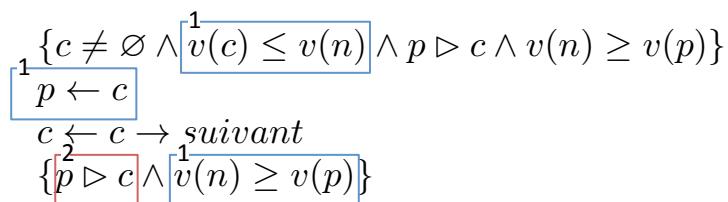


FIGURE 4.3 – Preuve que l'invariant de boucle est correct. Dans ce schéma, nous avons simplifié les variables en ne conservant que leur première lettre.

### ► Preuve de terminaison

La preuve de correction n'est pas suffisante, il faut aussi prouver que l'algorithme se termine. Cela se fait par l'introduction d'une fonction strictement monotone dont la valeur ne pourra pas faire autre chose que tendre vers 0 ou une valeur donnée. Dans notre cas précis, la formalisation de notre pré-condition nous indique qu'il existe un entier positif ou nul indiquant le nombre d'éléments dans la liste chaînée que l'on note  $n$ . Dans la boucle, la variable *courant* représente  $\text{liste}(\rightarrow \text{suivant})^i$  où  $i$  est le nombre de passage dans la boucle. Ce nombre  $i$  ne peut qu'augmenter de 1 à chaque passage, c'est une fonction strictement monotone croissante. Or, sa valeur ne pourra pas être supérieure à  $n$  car la boucle prend fin quand *courant* est nul. La terminaison est donc prouvée.

### ► En pratique

L'outil de preuve de correction que l'on vient de voir ne permet pas de concevoir un algorithme. Il permet simplement de savoir si cet algorithme est correct une fois qu'il est conçu. Pour concevoir un algorithme, il faut donc réfléchir à ce que doit faire l'algorithme, faire des schémas et bien penser à tous les cas de figure. Néanmoins, la notion d'invariant de boucle peut être utilisée dès la conception afin d'aider à prouver que la boucle est bien formée.

### ► Parcours

Le parcours d'une liste chaînée est extrêmement simple, il consiste à faire une boucle et tant que le pointeur n'est pas nul, passer au suivant :

```

courant ← liste
tant que courant ≠ Ø faire
    Faire quelque chose avec courant
    courant ← courant → suivant

```

### ► Suppression

En exercice, faites l'algorithme de suppression d'un élément de la liste. Commencer par exprimer en termes d'entrées, sorties, pré et post-conditions ce que doit faire l'algorithme. Ensuite, concevez le, et n'oubliez pas de libérer la mémoire !

### ► Gestion par tableau

Afin d'éviter de stocker un pointeur coûteux (surtout sur un système 64 bits), on peut gérer ses propres pointeurs sous forme d'indices entiers (qui seront stockés sur 16 bits par exemple). En lieu et place de l'allocation dynamique pour chaque cellule, un tableau contenant toutes les cellules sera construit et les pointeurs seront remplacés par les indices dans ce tableau. La figure 4.4 montre un exemple de stockage en mémoire basé sur ce principe. Le pointeur de début

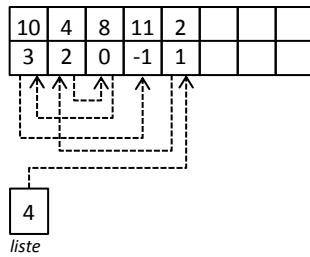


FIGURE 4.4 – Une liste chaînée peut être représentée en mémoire par un tableau. Chaque pointeur est remplacé par un entier qui indique le prochain indice dans la liste. Le pointeur nul est remplacé par une valeur particulière (ici -1).

de liste sera tout simplement l'indice de la première cellule. Ce principe peut être appliqué à n'importe quelle structure de données basée sur des pointeurs. L'inconvénient est qu'il faut implémenter soi-même un mécanisme de gestion du remplissage du tableau (risque de trous lors de suppressions).

**Exercice.** (difficulté 4) Proposer un algorithme qui inverse l'ordre d'une liste chaînée simple avec les contraintes suivantes : pas de récursivité, pas de stockage intermédiaire non constant. La nouvelle liste sera renvoyée sous forme de son pointeur sur la tête, l'ancienne liste ne sera pas détruite.

## 4.4 Liste doublement chaînée

Une liste doublement chaînée comporte en plus du lien vers la cellule suivante un lien vers la cellule précédente (d'où le nom !). Cette fonctionnalité permet de parcourir la liste dans les deux sens (en avant comme en arrière), mais a un surcoût d'un pointeur par cellule. Au niveau de sa déclaration, ce sera comme cela :

```
Structure ListeDouble
    entier valeur
    ListeDouble * suivant
    ListeDouble * precedent
```

Les algorithmes d'insertion, suppression sont assez proches de ceux de la liste simplement chaînée, mais attention, il faut garder la cohérence globale de la liste donc mettre à jour tous les pointeurs. La figure 4.5 montre un exemple d'une insertion. Là où l'insertion ne nécessite que la modification de 2 pointeurs dans une liste simplement chaînée, elle devra en modifier 4 pour une liste doublement chaînée. Par contre, le fonctionnement de l'algorithme est identique en tous points.

### ► Sentinelle

Afin de simplifier les procédures d'insertion et suppression sur une liste doublement chaînée, on ajoute parfois un élément fictif appelé sentinelle, qui rend la liste circulaire et permet de s'affranchir de certains tests. Le pointeur sur l'élément suivant de la sentinelle est le premier élément de la liste, tandis que le pointeur sur le précédent est le dernier de la liste.

**Exercice.** (difficulté 3) Donner l'algorithme d'insertion dans une liste doublement chaînée circulaire avec sentinelle.

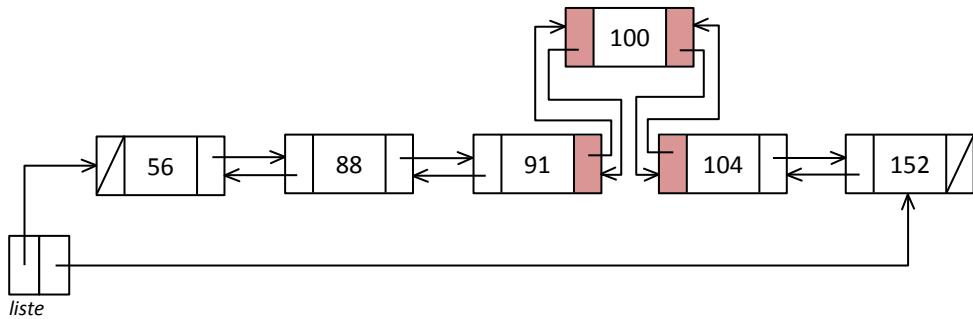


FIGURE 4.5 – Pour une liste doublement chaînée, l'insertion d'un élément nécessite le changement de 4 pointeurs contre 2 habituellement dans une liste chaînée simple.

**Exercice.** (difficulté 3) Montrer comment implémenter l'opération d'union de deux listes en  $\mathcal{O}(1)$  (on ne cherche pas à conserver les deux listes données en argument). Quel type de liste chaînée est le plus adapté pour ce genre d'opérations ?

Mémo

1. Les listes chaînées permettent une structuration des données qui morcellent l'information
2. Les procédures d'ajout en début et queue sont facilitées
3. La gestion d'une liste chaînée induit un surcoût mémoire qui peut être loin de négligeable
4. Ca n'est donc pas la solution universelle à tous les problèmes

## 4.5 Arbre

Un arbre est une structure où chaque cellule pointe sur des cellules filles. Lorsque le nombre de cellules filles est égal à 2, on parle d'arbre binaire. La figure 4.6 montre un exemple d'un tel arbre. Une cellule qui possède des cellules filles est appelé nœud tandis qu'une cellule sans cellule fille est appelé feuille de l'arbre. Voici un exemple de structure associée :

```
Structure Arbre
entier valeur
Arbre * gauche
Arbre * droit
```

Les arbres peuvent servir à stocker des informations de toutes sortes mais ont toujours un point commun : la relation de parenté entre les cellules. Une utilisation classique des arbres est la représentation des expressions, par exemple arithmétiques. Dans ce cas, on parle souvent de nœuds qui représentent les opérateurs et de feuilles qui représentent les valeurs de l'expression. La figure 4.7 montre un exemple graphique de la représentation d'un tel arbre.

### ► Propriétés

Le point d'entrée de l'arbre, c'est-à-dire le nœud qui n'a pas de père est appelé racine de l'arbre. On définit la profondeur d'une cellule par la longueur du chemin qui la sépare de la racine. La profondeur du nœud racine est donc de 0. La

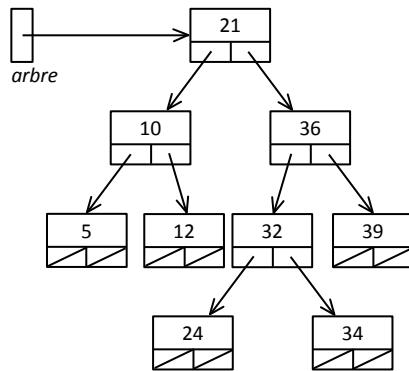


FIGURE 4.6 – Exemple d'un arbre binaire particulier appelé arbre binaire de recherche (les valeurs des nœuds du sous-arbre de gauche sont plus petites que la valeur de la cellule mère et inversement du côté du sous-arbre de droite). La variable *arbre* sert ici de point d'entrée sur l'arbre, on l'appelle la racine de l'arbre.

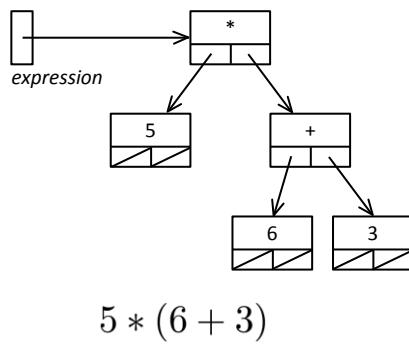


FIGURE 4.7 – Exemple d'un arbre représentant l'expression arithmétique  $5 * (6 + 3)$ . Chaque nœud correspond à un opérateur alors que les feuilles sont des valeurs.

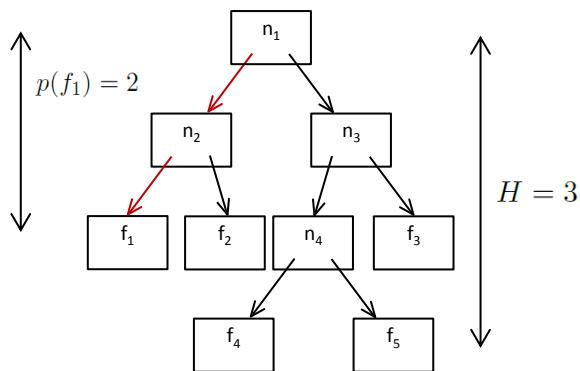


FIGURE 4.8 – Les différentes propriétés d'un arbre. Le nœud  $f_1$  a une profondeur de 2 car il faut traverser 2 arrêtes avant de rejoindre la racine  $n_1$ . La hauteur de cet arbre est de 3, la valeur maximum de profondeur.

hauteur  $H$  d'un arbre est la valeur maximale de la profondeur des feuilles de l'arbre. L'arbre de la figure 4.8 a une hauteur  $H = 3$ . La profondeur du nœud  $f_1$  est de 2. Un arbre équilibré est un arbre dont les sous-arbres de droite et de gauche ont une différence de profondeur d'au plus 1. Un arbre parfait ou complet est un arbre dont toutes les feuilles ont une profondeur égale à  $H$ . L'arbre de la figure 4.8 est équilibré mais pas complet. Un arbre complet est aussi équilibré.

### ► Méthodes de parcours

Imaginons qu'un arbre binaire est construit en mémoire. Il existe plusieurs façons de parcourir les cellules avec une fonction récursive. La différence se situe dans la façon de faire les appels récursifs par rapport au traitement de la cellule elle-même. Voici les trois façons de procéder :

1. Parcours préfixe : on traite la cellule, puis on traite le sous-arbre gauche, puis le droit (pour l'arbre de la figure 4.7 cela donne \* 5 + 6 3)
2. Parcours infixé : on traite le sous-arbre gauche, puis la cellule puis le sous-arbre droit (pour l'arbre de la figure 4.7 cela donne 5 \* 6 + 3)
3. Parcours postfixe : on traite le sous-arbre gauche, le droit, puis la cellule elle-même (pour l'arbre de la figure 4.7 cela donne 5 6 3 + \*)

Ces trois types de parcours ont un point en commun, ils sont facilement implémentable avec une fonction récursive (voir l'algorithme 54 pour le parcours préfixe) et sans l'aide d'une structure de données annexe. On les appelle des parcours en profondeur (en anglais *depth-first order*) car ils cherchent à traiter les cellules filles avant de traiter les cellules *sœurs* c'est-à-dire d'un même niveau. Par opposition, il existe une méthode de parcours appelée parcours en largeur qui permet de traiter toutes les cellules d'un même niveau de profondeur avant de traiter les suivantes (en anglais *breadth-first order*). La figure 4.9 montre la différence entre ces deux types de parcours.

### ► Parcours itératif

Parfois, il n'est pas souhaité de faire un parcours récursif de l'arbre. Parfois même cela n'est pas possible comme pour le parcours en largeur vu précédemment. L'idée derrière la transformation de l'algorithme réside dans l'utilisation d'une structure de données indiquant l'ensemble des cellules restant à traiter. La boucle principale consistera donc à continuer tant que cet ensemble de cellules n'est pas vide. A l'intérieur de la boucle, on ajoutera ou supprimera des cellules à traiter. Suivant que l'on utilisera une structure de données de type file ou pile, l'effet sera de faire un parcours en largeur ou en profondeur respectivement. L'algorithme 55 donne l'implémentation d'un parcours en largeur sur un

**Algorithme 54 :** Algorithme de parcours d'un arbre binaire en profondeur dans l'ordre préfixé.

**1 Structure Arbre**

```

2   entier valeur
3   Arbre * gauche
4   Arbre * droit

```

**5 Fonction ParcoursArbrePrefixe(noeud)**

Entrée : Arbre \* noeud

Précondition : L'arbre est bien construit

Postcondition : Effectue l'opération Operation() sur chaque cellule de l'arbre dans l'ordre d'un parcours en profondeur préfixe

```

6   Operation(noeud)
7   si noeud → gauche ≠ Ø alors
8     ParcoursArbrePrefixe(noeud → gauche)
9   si noeud → droit ≠ Ø alors
10    ParcoursArbrePrefixe(noeud → droit)

```

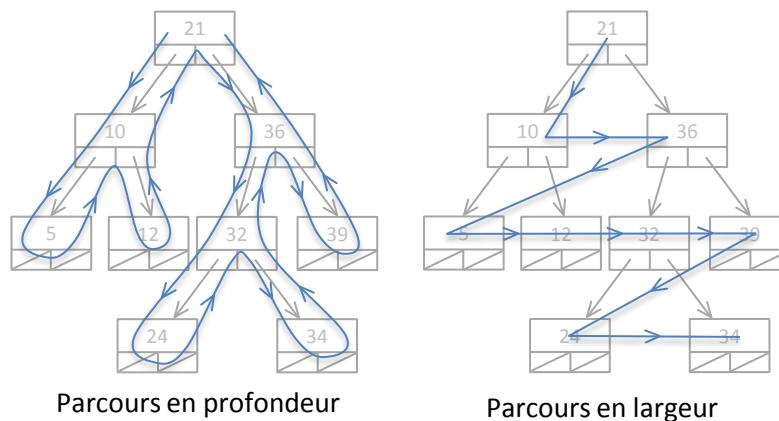


FIGURE 4.9 – Les deux types fondamentaux de parcours d'un arbre.

arbre binaire en utilisant le type de données abstrait file (que nous verrons plus tard dans le cours). Pour le moment retenez simplement qu'une file fonctionne en FIFO (*First In First Out*) donc que le premier qui rentre sera le premier à sortir. Lorsqu'un arbre possède des propriétés particulières, son parcours itératif peut permettre de le coder de manière linéaire. Par exemple, un arbre binaire complet parcouru de manière itérative en largeur aura toujours la propriété de décrire un nombre de nœuds étant une puissance de 2 croissante à chaque niveau. Cela peut servir aussi de base de stockage sans pointeur.

**Exercice.** Sur l'exemple de l'arbre de la figure 4.9, déroulez l'algorithme 55 en simulant l'état de la file.

**Exercice.** Peut-on prévoir la taille maximum de la file dans cet algorithme ?

**Exercice.** Modifier l'algorithme pour qu'il fasse un parcours en profondeur grâce à l'utilisation d'une pile à la place de la file.

**Algorithme 55 :** Algorithme de parcours d'un arbre binaire en largeur (par niveau).

**1 Structure Arbre**

```

2   entier valeur
3   Arbre * gauche
4   Arbre * droit

```

**5 Fonction** ParcoursArbreLargeur(*racine*)

**Entrée** : Arbre \* *racine*

**Précondition** : L'arbre est bien construit

**Postcondition** : Effectue l'opération *Operation()* sur chaque cellule de l'arbre dans l'ordre d'un parcours en largeur (par niveau)

**Déclaration** : Arbre \* *courant*  
                  File *file*

```

6   file.Enfiler(racine)
7   tant que non file.Vide() faire
8       courant  $\leftarrow$  file.Defiler()
9       Operation(courant)
10      si courant  $\rightarrow$  gauche  $\neq \emptyset$  alors
11          file.Enfiler(courant  $\rightarrow$  gauche)
12      si courant  $\rightarrow$  droit  $\neq \emptyset$  alors
13          file.Enfiler(courant  $\rightarrow$  droit)

```

## ► Aplatissement

Un arbre peut sous certaines conditions être mis à plat, c'est-à-dire placé dans un tableau sans coder de manière explicite la structure arborescente. Cela est tout simplement possible grâce à l'algorithme précédent de parcours itératif. Imaginons que l'arbre sur lequel on applique cet algorithme soit complet, cela signifie que chaque niveau sauf le dernier est entièrement rempli. Si en plus de cela on ajoute le fait que les cellules du dernier niveau sont *tassées* vers la gauche, alors on peut prédire de façon automatique la position d'une cellule dans l'arbre à partir de son parcours en largeur. La figure 4.10 illustre ce principe.

Les indices du tableau permettent de parcourir l'arbre de manière aisée :

- L'indice du père d'un nœud *i* est  $\lfloor (i - 1)/2 \rfloor$
- L'indice du fils gauche d'un nœud *i* est  $2i + 1$
- L'indice du fils droit d'un nœud *i* est  $2i + 2$

Si la numérotation commence à 1, les formules sont évidemment différentes.

## ► Adressage d'une cellule

Afin de décrire la position d'une cellule dans l'arbre, on peut indiquer le chemin qu'il a fallu prendre pour parvenir à cette cellule depuis la racine. Pour un arbre binaire, on a le choix entre deux directions, la gauche ou la droite que l'on peut étiqueter avec un alphabet binaire. La figure 4.11 illustre ce principe. Par convention, l'adresse de la racine sera le mot vide  $\epsilon$ .

**Exercice.** (difficulté 3) Donner l'algorithme qui permet de transformer l'adresse d'une cellule (sous forme d'une chaîne de caractères) en indice dans un tableau d'arbre aplati (entier). On considère l'arbre complet.

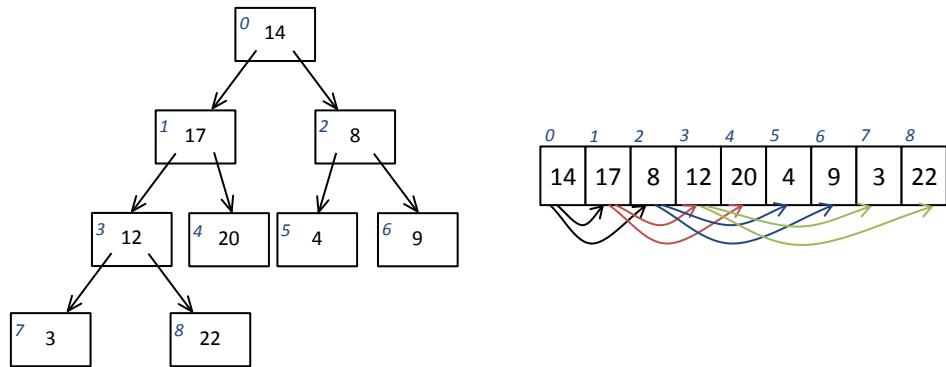


FIGURE 4.10 – Aplatissement d'un arbre quasi-complet.

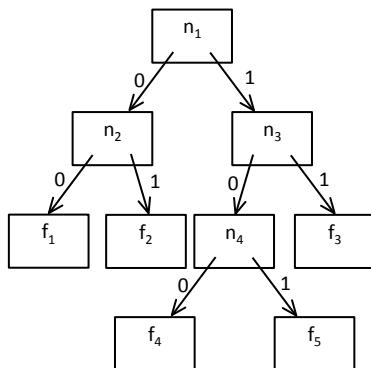


FIGURE 4.11 – Adressage des cellules d'un arbre. La cellule  $f_4$  a pour adresse 100 dans cet exemple.

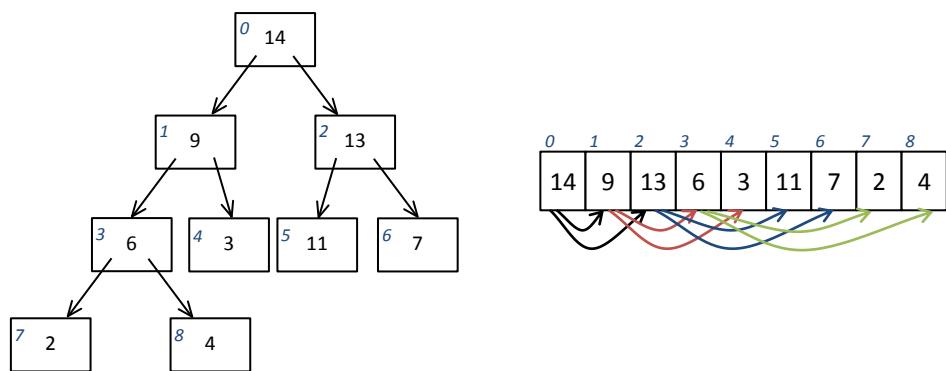


FIGURE 4.12 – Exemple de tas binaire et de son aplatissement sous forme de tableau.

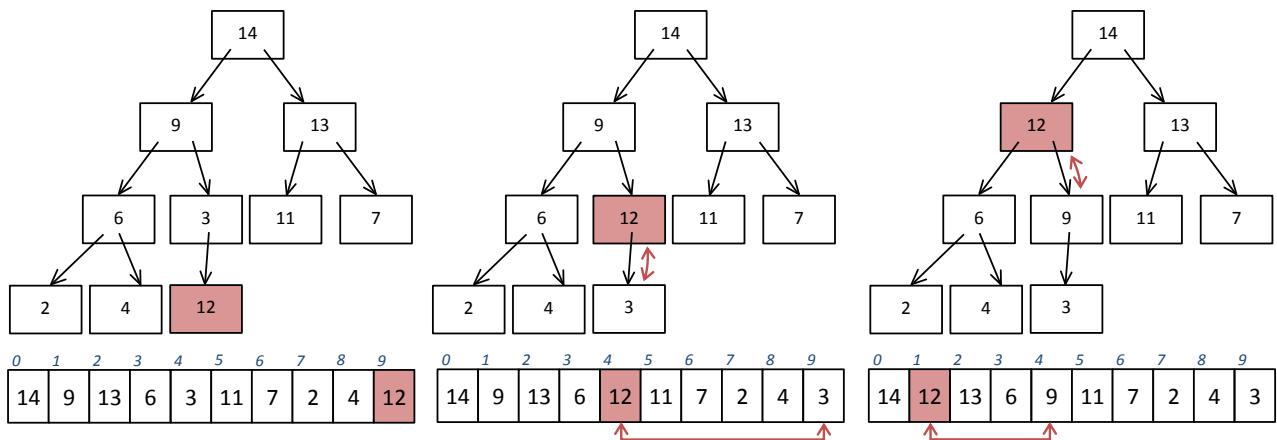


FIGURE 4.13 – Insertion dans un tas binaire. L’élément à insérer est d’abord placé dans la prochaine case libre. On permute la cellule avec son père lorsque la relation n’est pas respectée. Dans cet exemple, il a fallu permuter deux niveaux avant d’arriver à satisfaire la relation. La version aplatie de l’arbre est représentée en dessous.

**Exercice.** (difficulté 5) Donner l’algorithme qui permet de transformer l’indice dans un tableau d’arbre aplati en adresse. On considère l’arbre complet.

### ► Tas binaire

Un tas binaire est un arbre binaire complet qui possède une propriété particulière. En effet, la valeur associée à chaque nœud est supérieure à celle du fils gauche ainsi que celle du fils droit. On ne donne en revanche aucune contrainte sur lequel des deux fils doit avoir la valeur la plus petite. Une propriété qui découle directement est que la racine de l’arbre a la valeur la plus grande.

La figure 4.12 à gauche donne un exemple de tas binaire et sa représentation sous forme d’arbre. Les tas binaires sont très souvent représentés sous leur forme aplatie, donc dans un tableau car cela permet d’économiser le stockage des pointeurs (voir figure 4.12).

**Insertion.** Pour insérer une nouvelle valeur dans un tas binaire, il faut simplement la placer à la fin du tas (dernier niveau, première place libre) et remonter l’arbre en faisant les permutations nécessaires. La figure 4.13 illustre le cas d’insertion dans un tas binaire.

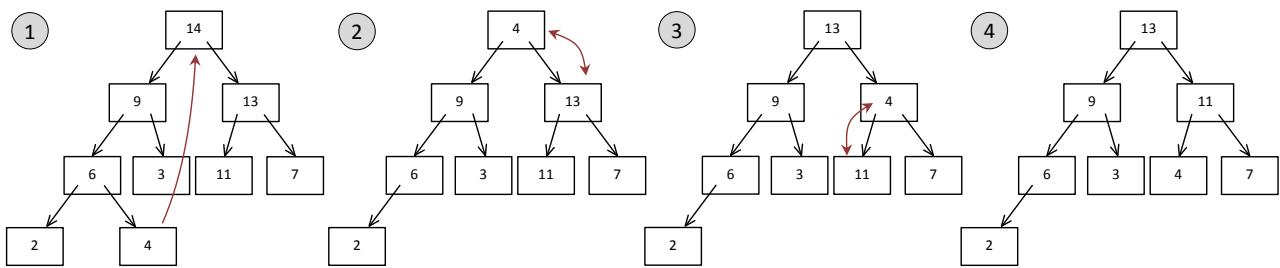


FIGURE 4.14 – Suppression de la racine d'un tas binaire. 1) On commence par permute la valeur racine avec la dernière valeur afin de supprimer la valeur racine. 2) La propriété du nœud 4 n'est plus respectée, on permute avec le plus grand, donc le 13. 3) La propriété n'est toujours pas vérifiée, on permute avec le nœud 11. 4) On arrive à une feuille, la propriété est forcément vérifiée.

**Exercice.** (difficulté 3) Donner l'algorithme qui permet d'insérer une nouvelle valeur dans un tas binaire. Le tas sera représenté sous forme d'arbre binaire avec pointeurs. On supposera que la cellule est déjà insérée et que son pointeur est donné en argument de la procédure. On supposera aussi que chaque cellule stocke l'adresse de son père (nul si racine).

**Exercice.** (difficulté 7) Prouver que les permutations qui sont induites par cet algorithme aboutissent à un tas binaire bien formé *i.e.* qui respecte bien la relation de supériorité de la valeur du père par rapport à ses fils.

**Suppression.** En général, dans un tas binaire on supprime l'élément racine (celui dont la valeur est la plus grande) mais le principe de suppression marche quel que soit la valeur à supprimer (il suffit d'avoir sa position initiale). Étant donné que le tas doit toujours avoir une forme bien respectée (quasi-complet et dernier niveau tassé à gauche), la suppression de l'élément va se faire toujours au même endroit de l'arbre. Il faut donc permute l'élément à supprimer avec le dernier avant de le supprimer. Ensuite, un peu de la même manière que pour l'insertion, nous allons vérifier localement la propriété de tas et si elle n'est pas vérifiée, faire une permutation. Il faut toujours permute avec la valeur la plus grande car c'est ce qui garantit la propriété. Cet algorithme est fait de manière récursive jusqu'à l'obtention d'une feuille de l'arbre. La figure 4.14 illustre le principe de la suppression de la racine d'un tas.

**Avec un tableau.** Il s'avère que la gestion d'un tas binaire dans sa version aplatie est en fait beaucoup moins coûteuse en terme d'espace mais aussi plus simple algorithmiquement. Pour cela il suffit de stocker le nombre d'éléments présents dans le tas binaire et la position de l'insertion se fait alors plus facilement. La relation de parenté peut se traduire directement sur les indices du tableau et les permutations de valeurs dans un tableau sont très aisées. La figure 4.12 montre comment un tas binaire peut être aplati pour le stocker dans un tableau.

L'algorithme 56 donne l'algorithme de l'insertion d'une valeur dans un tas binaire.

**Algorithme 56 :** Algorithme d'insertion d'une valeur dans un tas binaire codé sous forme de tableau.

```

1 Structure Tas
2   entier taille
3   entier taillemax
4   entier * tab

5 Fonction InsertionTasBinaire(tas,valeur)
  Entrée/Sortie : Tas tas
                  entier valeur
  Précondition : Le tas est bien construit.
  Postcondition : Effectue l'insertion de la valeur valeur dans le tas. La fonction renvoie vrai en cas de succès,
                    faux en cas d'échec (taille max dépassée).
  Déclaration : entier indice
                  entier pere
                  entier tmp
6   si tas.taille = tas.taillemax alors
7     retourne faux
8   tas.tab[tas.taille] ← valeur
9   indice ← tas.taille
10  tas.taille ← tas.taille + 1
11  pere ← ArrondiInf((indice – 1)/2)
12  tant que indice > 0 et tas.tab[pere] < tas.tab[indice] faire
13    tmp ← tas.tab[pere]
14    tas.tab[pere] ← tas.tab[indice]
15    tas.tab[indice] ← tmp
16    indice ← pere
17    pere ← ArrondiInf((indice – 1)/2)
18  retourne vrai

```

## CHAPITRE 5

### TYPES DE DONNÉES ABSTRAITS

Un type de données abstrait est différent d'une structure telle que nous l'avons vue dans la section précédente car il contient un cahier des charges des opérations que l'on va pouvoir exécuter dessus.

#### Définition

Un type de données abstrait (TDA) est l'association d'une structure de données et d'un ensemble d'opérations que l'on peut exécuter dessus.

Dans la définition précédente, le terme structure de données est pris au sens large, c'est-à-dire qu'on ne dit pas comment la structure de données doit être implémentée mais seulement le type d'organisation que l'on souhaite. Par exemple, une collection ordonnée peut être représentée en mémoire par une liste chaînée, un tableau, un arbre binaire, une liste chaînée de tableaux, *etc.* À ce titre là, il convient de bien faire la différence entre certains termes qui sont assez proches entre eux. Par exemple, *une liste* est un type de donnée abstrait, alors qu'*une liste chaînée* (façon d'organiser la mémoire) est une structure de données. On peut implémenter une liste grâce à une liste chaînée mais rien n'y oblige. De manière complémentaire, on ne peut pas donner de complexité aux opérations d'un type de données abstrait, car par définition il n'est pas encore implémenté : il est abstrait.

Quelques TDA standards :

1. Pile : une collection d'objets accessible selon une politique de LIFO (*Last In First Out*, le dernier entré est le premier sorti)
2. File : une collection d'objets accessible selon une politique de FIFO (*First In First Out*, le premier entré est le premier sorti)
3. File double : mélange de la pile et de la file
4. Liste : collection ordonnée d'objets accessible selon la position relative
5. Vecteur : collection ordonnée d'objets accessible selon la position absolue (indice)
6. File de priorité : collection d'objets avec une priorité associée dont le seul accessible est celui qui a la priorité la plus forte
7. Dictionnaire : collection de paires (clé, valeur) avec des opérations de recherche, d'insertion et de suppression basées sur la clé
8. Ensemble : collection non ordonnée d'objets sans doublon
9. *etc.*

Dans les sections suivantes nous allons nous pencher sur quelques TDA très connus et très utilisés et étudier les possibilités d'implémentation associées.

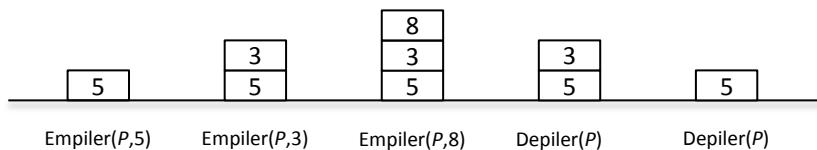


FIGURE 5.1 – Exemple d'utilisation d'une pile.

## 5.1 Piles

Une pile est un type de données abstrait qui permet de représenter un ensemble dynamique d'objets dont l'accès se fait par une politique LIFO (*Last In First Out*, le dernier entré est le premier à sortir).

L'interface qui permet de la gérer est simple et consiste en 3 fonctions (l'instance de pile sera notée  $P$ ) :

1.  $\text{Vide}(P)$  est une fonction qui renvoie vrai si la pile est vide, faux sinon
2.  $\text{Empiler}(P, v)$  permet de placer la valeur  $v$  sur le sommet de la pile
3.  $\text{Depiler}(P)$  renvoie la valeur qui se trouve sur le sommet de la pile et enlève ce dernier

La figure 5.1 montre une représentation graphique de la pile et son évolution en fonction des opérations effectuées dessus.

### ► Implémentations

**Implémentation par un tableau.** Une manière simple d'implémenter une pile est d'utiliser un tableau auquel on adjoint une information sur l'indice du dernier élément (le sommet). Ainsi, pour empiler, il suffit d'incrémenter cette valeur et mettre à jour la case correspondante. Pour dépiler, il faut renvoyer la valeur qui se trouve à l'indice du sommet et décrémenter le compteur. La figure 5.2 illustre le fonctionnement d'une pile gérée par un tableau. Ce type d'implémentation est très performant (toutes les opérations sont effectuées en temps constant) à partir du moment où on connaît à l'avance la taille maximale de la pile.

**Implémentation avec une liste chaînée.** L'inconvénient de la méthode précédente est la limitation de la taille de la pile, qui engendre une réallocation si on la dépasse (temps qui devient linéaire du coup). On peut remarquer que les opérations que l'on fait sur une pile sont toujours sur le sommet de cette pile. On peut alors facilement imaginer une implémentation à base de liste chaînée simple. On n'aura même pas besoin de pointeur de fin de liste si on décide que le sommet de la pile est la tête de la liste chaînée.

**Gestion d'erreur.** Quelle que soit la façon de gérer la pile, un cas de figure peut se présenter : on demande de dépiler l'élément au sommet de la pile alors que la pile est vide. Il faut prévoir un mécanisme qui permet d'informer l'utilisateur qu'il y a eu un problème. Il existe plusieurs façons de gérer ce problème :

1. On transforme la fonction  $\text{Depiler}()$  pour qu'elle renvoie aussi un booléen d'état. Cela peut se faire par une procédure du type  $\text{Depiler}(P, \text{valeur}, \text{diagnostic})$  où  $\text{valeur}$  et  $\text{diagnostic}$  sont des variables en sortie.  $\text{diagnostic}$  est de type booléen et indique par un *vrai* que l'opération s'est bien passée ou par un *faux* qu'on ne pouvait pas dépiler (donc que la pile était vide) et qu'il est inutile de lire  $\text{valeur}$ .
2. Une valeur particulière permet d'indiquer que cela ne s'est pas bien passé. Imaginez que vous rangez uniquement des entiers positifs dans la pile. Si la valeur renvoyée est négative, alors on saura qu'il y a eu un problème. Cette solution ne s'applique pas dans tous les cas de figure.
3. On peut utiliser le mécanisme de gestion d'erreur du langage cible (en C++ par exemple, les exceptions).

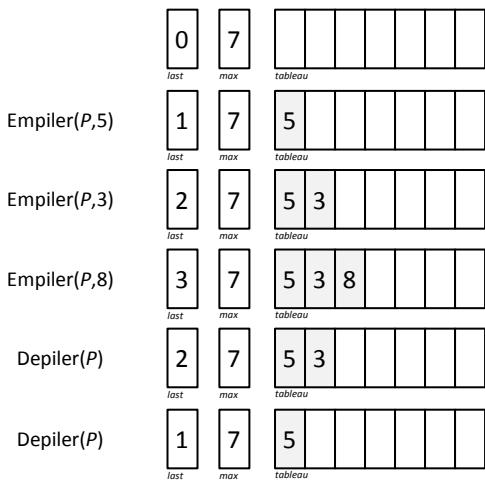


FIGURE 5.2 – Exemple de représentation d'une pile par un tableau. On doit stocker la taille actuelle de la pile dans une variable (ici *last*) ainsi que la taille du tableau (ici *max*). Cette taille de tableau servira à réallouer le tableau si la taille venait à ne pas être suffisante ou gérer une exception le cas échéant.

**Exercice.** (difficulté 2) Donner la structure d'une pile implémentée avec une liste chaînée.

**Exercice.** (difficulté 3) Implémenter les trois algorithmes qui permettent de la traiter. Faut-il en rajouter un autre ?

## ► Applications

Les piles sont très utilisées dans de nombreuses applications. On peut les utiliser par exemple pour vérifier dans une chaîne de caractères que les parenthèses sont bien ouvertes et fermées de manière cohérente. À l'ouverture, on empile le symbole, à la fermeture, on vérifie que le symbole est bien le même et on le dépile. Les piles peuvent être aussi utilisées pour l'évaluation d'expressions. Dans ce cas là, une valeur dans l'expression donnera lieu à l'empilement de cette dernière, tandis qu'un opérateur prendra les deux valeurs de la pile, les combinera avec l'opérateur souhaité et empilera le résultat (l'expression doit être dans un ordre post-fixé). L'algorithme 57 montre comment on peut évaluer une expression stockée dans une liste chaînée sous forme post-fixée l'aide d'une pile.

**Exercice.** (difficulté 3) Transformer l'expression  $((6 + 3) * (4 + 3)) / 3$  en notation post-fixée. Simuler ensuite l'exécution de l'algorithme précédent en représentant les états successifs de la pile.

## 5.2 Files

Une file (appelée aussi file d'attente) est un type de données abstrait qui permet de représenter un ensemble dynamique d'objets dont l'accès se fait par une politique FIFO (*First In First Out*, le premier entré est le premier à sortir). L'interface qui permet de la gérer ressemble à celle de la pile et consiste en 3 fonctions (l'instance de file sera notée  $F$ ) :

1.  $\text{Vide}(F)$  est une fonction qui renvoie vrai si la file est vide, faux sinon
2.  $\text{Enfiler}(F, v)$  permet de placer la valeur  $v$  dans la file
3.  $\text{Defiler}(F)$  renvoie la prochaine valeur dans la file et la supprime

**Algorithme 57 :** Evaluation d'une expression post-fixée.

```

1 Structure Expression
2   booléen estvaleur
3   réel valeur
4   caractère operateur
5   Expression * suivant

6 Fonction EvalExpression(expression,diagnostic)
  Entrée      : Expression * expression
  Précondition : expression contient une liste chaînée représentant une expression arithmétique en notation
                  post-fixée.
  Postcondition : Renvoie la valeur de l'expression. S'il y a une erreur dans l'expression, diagnostic est
                  positionné à faux, sinon à vrai.
  Déclaration : Expression * courant, * tmp
                  Pile pile
                  réel v1, v2
                  booléen d1, d2

7   courant ← expression
8   tant que courant ≠ Ø faire
9     si courant → estvaleur alors
10    |   Empiler(pile, courant → valeur)
11    sinon
12    |   Depiler(pile,v1,d1)
13    |   Depiler(pile,v2,d2)
14    |   si non d1 ou non d2 alors
15    |     diagnostic ← faux
16    |     Detruire(pile)
17    |     retourne 0
18    |   si courant → operateur = '+' alors
19    |     Empiler(pile,v1 + v2)
20    |     /* Compléter ici avec d'autres opérateurs
21    |   courant ← courant → suivant
22    |   Depiler(pile,v1,d1)
23    |   si d1 alors
24    |     diagnostic ← vrai
25    |     retourne v1
26    |   sinon
27    |     diagnostic ← faux
28    |     retourne 0

```

---

**► Implémentation**

Un peu comme pour les pile, il existe deux manières principalement d'implémenter une file : un tableau circulaire, ou une liste chaînée.

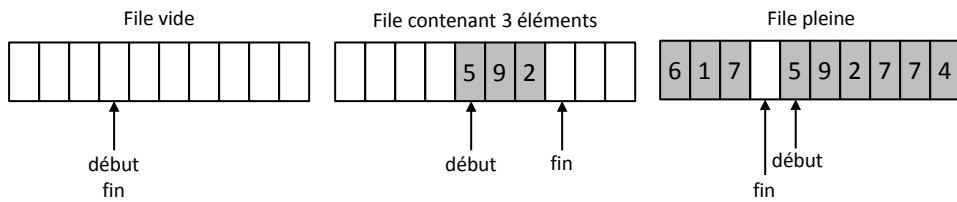


FIGURE 5.3 – Représentation d'une file par un tableau circulaire. Lorsque la file est pleine (droite), une case ne peut pas être utilisée car sinon, on ne pourrait pas différencier le cas de la file vide (à gauche) de celui de la file pleine.

**Implémentation par un tableau circulaire.** La gestion d'une file par un tableau est un peu plus complexe que celle de la pile. Elle nécessite l'introduction de 2 indices, un qui indique le début de l'utilisation du tableau, un autre la fin. Ces deux indices correspondent à la tête et la queue de la file. Les deux opérations `Enfiler()` et `Defiler()` vont avoir pour conséquence d'incrémenter un de ces deux indices (ils ne vont jamais décroître sauf lorsque l'on atteint la fin du tableau). Il faut pouvoir représenter deux cas de figures particuliers : la file vide (les deux indices sont égaux) et la file pleine (un décalage d'un entre les deux indices). On ne pourra pas remplir complètement la file (une case perdue) sous peine de ne pas pouvoir séparer les deux cas (voir figure 5.3). Les algorithmes 58 et 59 donnent les éléments principaux de cette gestion.

**Exercice.** (difficulté 1) Faites les algorithmes de libération de la file et de test pour savoir si elle est vide.

---

**Algorithme 58 :** Gestion d'une file par un tableau circulaire (structure et procédure d'initialisation).

---

```

1 Structure File
2   entier taille
3   entier * tab
4   entier tete
5   entier queue

6 Procédure Init(file,taille)
Entrée      : entier taille
Entrée/Sortie : File file
Postcondition : Initialise la structure de stockage d'une file sous forme de tableau circulaire de taille maximum taille.
7   file.taille ← taille
8   file.tab ← Allouer(entier,taille)
9   file.tete ← 0
10  file.queue ← 0

```

---

**Implémentation par une liste chaînée.** L'implémentation par une liste chaînée est assez simple et est laissée en exercice (difficulté 2).

## ► Applications

Dès lors que l'on veut garder une trace de l'ordre d'événements pour pouvoir les traiter dans le même ordre d'arrivée, les files sont très utiles. Les événements sont alors stockés en attendant d'être traités, mais sans pour autant empêcher l'arrivée d'autres événements.

**Algorithme 59 :** Gestion d'une file par un tableau circulaire.

1 **Procédure** Enfiler(*file,v,diagnostic*)

**Entrée/Sortie** : File *file*

**Entrée** : entier *v*

**Sortie** : booléen *diagnostic*

**Précondition** : *file* contient un tableau circulaire qui code la file. Si la file est vide, alors la queue et la tête ont la même valeur. Si la file est pleine, il y a un décalage d'un entre les deux valeurs.

**Postcondition** : L'élément *v* est mis dans la file d'attente. Si la file est pleine, *diagnostic* est positionné à *faux* et l'élément ne sera pas inséré, sinon il est positionné à *vrai*.

2   **si** ((*file.tete* + *file.taille* – *file.queue*)%*file.taille*) = 1 **alors**

  |   *diagnostic* ← *faux*

4   **sinon**

5     *file.tab[file.queue]* ← *v*

6     **si** *file.queue* = *file.taille* – 1 **alors**

  |     *file.queue* ← 0

8     **sinon**

9        *file.queue* ← *file.queue* + 1

10      *diagnostic* ← *vrai*

11 **Procédure** Defiler(*file,v,diagnostic*)

**Entrée/Sortie** : File *file*

**Sortie** : entier *v*

**Sortie** : booléen *diagnostic*

**Précondition** : *file* contient un tableau circulaire qui code la file. Si la file est vide, alors la queue et la tête ont la même valeur. Si la file est pleine, il y a un décalage d'un entre les deux valeurs.

**Postcondition** : Le prochain élément de la file est retiré du tableau est renseigné dans *v*. Si la file ne peut pas être défilée car vide, *diagnostic* est positionné à *faux*, sinon à *vrai*.

12   **si** *file.queue* = *file.tete* **alors**

  |   *diagnostic* ← *faux*

14   **sinon**

15     *v* ← *file.tab[file.tete]*

16     **si** *file.tete* = *file.tete* – 1 **alors**

  |     *file.tete* ← 0

18     **sinon**

19        *file.tete* ← *file.tete* + 1

20      *diagnostic* ← *vrai*

### 5.3 File double (deque)

#### ► Principe

Une file double ou *double ended-queue* (*deque*) est une généralisation des deux TDA précédents. On peut ajouter et supprimer des éléments au début ou à la fin de la collection dynamique.

Voici les opérations standard que l'on peut effectuer sur ce TDA :

1. InsererDebut() permet d'insérer un élément au début
2. InsererFin() permet d'insérer un élément à la fin

3. SupprimerDebut() permet de supprimer le premier élément
4. SupprimerFin() permet de supprimer le dernier élément

### ► Implémentation

L'implémentation la plus courante de ce TDA est la liste chaînée double avec ou sans sentinelles. L'introduction d'une sentinelle permet de simplifier le code pour un surcoût constant (la sentinelle ne contient pas de données).

**Exercice.** (difficulté 3) Donner les algorithmes d'une implémentation en liste chaînée avec sentinelle.

## 5.4 Listes

Les listes sont un TDA qui généralise encore les TDA précédents en proposant des manières d'accéder et modifier les éléments à l'intérieur de la collection là où les piles et files ne le permettaient pas. Ce TDA est très particulier et nous parlerons uniquement de son implémentation sous forme de liste doublement chaînée ce qui implique que l'on indique la position dans la liste par un pointeur sur l'élément de la liste (il existe des variantes avec la notion d'itérateur).

### ► Opérations

Voici les opérations que l'on peut effectuer sur le TDA liste :

1. Toutes les opérations que l'on peut faire sur une file double (insertion et suppression en début et fin)
2. InsererAvant( $L, p, valeur$ ) : insère *valeur* avant la position  $p$
3. InsererApres( $L, p, valeur$ ) : insère *valeur* après la position  $p$
4. Supprimer( $L, p$ ) : supprime l'élément qui se trouve à la position  $p$
5. Remplacer( $L, p, valeur$ ) : remplace l'élément à la position  $p$  par la valeur *valeur*
6. Premier( $L$ ) : renvoie la première position de la liste
7. Dernier( $L$ ) : renvoie la dernière position de la liste
8. Suivant( $L, p$ ) : renvoie la position suivante de  $p$  dans la liste
9. Precedent( $L, p$ ) : renvoie la position précédente de  $p$  dans la liste

où  $L$  représente la liste,  $p$  représente une position dans cette liste (pointeur le plus souvent), et *valeur* est la valeur à insérer ou remplacer.

Les opérations Premier() et Suivant() vont permettre de parcourir la liste entière là où ce n'était pas possible avec les pile ou les files.

### ► Implémentations

**Liste chaînée.** Un liste est souvent implantée sous la forme d'une liste chaînée. Si on veut pouvoir effectuer la totalité des opérations définies précédemment, cette liste devra être doublement chaînée. On accède aux éléments grâce aux pointeurs sur les cellules de la liste chaînée. Toutes les opérations sur la liste auront un temps d'exécution constant avec cette implémentation.

**Autres possibilités.** Il existe bien d'autres possibilités pour représenter une liste, à base de tableau notamment, mais certaines opérations ne pourront plus s'effectuer en temps constant (insertion et suppression).

## 5.5 Vecteur

Une liste offre la possibilité d'accéder aux éléments à d'autres endroits que le début ou la fin. Néanmoins, pour accéder à ces éléments il faut toujours le faire de manière incrémentale (partir du début et se déplacer jusqu'à la position voulue). Cela implique que pour accéder au énième élément, il faut effectuer  $n$  opérations. Parfois, on aimerait pouvoir accéder directement à la valeur d'un élément à une certaine position, sans avoir à faire explicitement une boucle sur les éléments précédents. C'est le but du TDA vecteur.

Voici la liste des opérations que l'on peut effectuer sur ce dernier :

1. `ElementPosition( $V, i$ )` : renvoie la valeur du  $i^{\text{eme}}$  élément
2. `RemplacerPosition( $V, i, valeur$ )` : remplace le  $i^{\text{eme}}$  élément par la valeur *valeur*
3. `InsererPosition( $V, i, valeur$ )` : insère l'élément de valeur *valeur* à la  $i^{\text{eme}}$  position et déplace les éléments suivants d'un rang en avant
4. `SupprimerPosition( $V, i$ )` : supprime le  $i^{\text{eme}}$  élément du vecteur et déplace les suivants d'un rang en arrière
5. `Taille( $V$ )` : renvoie la taille du vecteur, afin de pouvoir effectuer des boucles basées sur le rang

Parfois, les deux premières opérations sont représentées grâce à l'utilisation des crochets à cause de leur similitude avec les tableaux. Certains langages de programmation comme le C++ permettent de faire des surcharges d'opérateurs rendant même cette notation possible.

### ► Applications

Ce TDA est une formalisation et généralisation d'un tableau dynamique, et peut donc être utilisé dans n'importe quelle application qui nécessite d'avoir un tableau dont la taille peut varier et sur lequel on veut pouvoir ajouter et supprimer des éléments sans se soucier des autres.

### ► Implémentations

Quelle que soit l'implémentation, il y aura toujours un inconvénient dû à la présence dans ce TDA de deux choses contradictoires :

1. L'accès aux éléments par un rang entier (un indice)
2. La possibilité de pouvoir enlever ou ajouter des éléments n'importe où dans la collection

Une implémentation sous forme de liste chaînée sera bonne pour le deuxième point mais mauvaise pour le premier. Une implémentation sous la forme de tableau extensible (voir page 60 pour l'implémentation) sera à l'inverse bonne pour l'accès par rang mais mauvaise pour l'insertion ou la suppression (surtout si ces opérations sont en début de collection). Un compromis souvent fait est de fabriquer une liste chaînée de tableaux.

**Exercice.** (difficulté 2) Donner les complexités en temps des opérations de base sur un vecteur implémenté par une liste chaînée.

**Exercice.** (difficulté 2) Donner les complexités en temps des opérations de base sur un vecteur implémenté par un tableau extensible.

**Exercice.** (difficulté 5) Proposer une variante construite par une liste chaînée de tableaux extensibles. Imaginer une stratégie de gestion des ajout et suppressions.

## 5.6 File de priorité

Ce TDA possède une interface très simple. Les éléments de la collection ont une priorité. On peut ajouter un élément de n'importe quelle priorité, mais on veut toujours extraire celui qui a la plus grande priorité. Voici les opérations que l'on souhaite effectuer :

1.  $\text{Inserer}(FP, p)$  : insère l'élément de priorité  $p$  dans la file de priorité
2.  $\text{Maximum}(FP)$  : renvoie l'élément de priorité maximum
3.  $\text{ExtraireMaximum}(FP)$  : renvoie l'élément de priorité maximum et le supprime de la file de priorité

### ► Applications

Ce TDA est très utilisé lorsqu'il s'agit d'affecter des ressources dont la priorité est variable, comme par exemple la gestion de processus dans un système d'exploitation. On peut par exemple affecter une priorité forte aux processus liés à l'interface graphique afin de ne pas obtenir des temps de réponses utilisateur trop long.

### ► Implémentations

**Tableau.** Une façon simple d'implémenter une file de priorité est de faire un tableau statique (dont la taille est fixée à l'avance). Comme ce tableau est trié (on mettra l'élément le plus prioritaire en dernier), l'extraction est simple et peu coûteuse. Par contre, l'insertion d'un nouvel élément peut être coûteuse car il faut déplacer les éléments plus prioritaires que celui inséré. La complexité en espace est constante car la taille est figée.

**Liste chaînée.** Une autre possibilité est de faire une liste chaînée triée. Là encore la complexité d'extraction sera constante mais l'insertion sera coûteuse (linéaire) à cause de la recherche de la position d'insertion.

**Exercice.** (difficulté 1) Comment obtenir un coût constant pour l'insertion et linéaire pour l'extraction ?

**Tas.** L'implémentation la plus standard d'une file de priorité est celle à partir d'un tas. L'opération d'insertion est celle décrite dans l'algorithme 55 de la page 74.

Pour extraire (donc supprimer) la racine du tas dont la valeur est celle la plus grande, on procède ainsi :

- On place la dernière valeur du tas en premier (le tas n'est alors plus correct)
- Si la propriété de tas n'est pas satisfaite localement, on intervertit la valeur de la cellule avec la plus grande des deux valeurs filles et on procède récursivement avec la cellule fille

**Exercice.** (difficulté 3) Donner l'algorithme d'extraction du maximum d'un tas binaire codé dans un tableau.

## 5.7 Dictionnaires

Un dictionnaire est un ensemble dynamique d'objets possédant une propriété appelée *clé*. Les clés peuvent être comparées entre elles et servent à organiser les données. En général, on ajoute à l'objet une information que l'on appelle *valeur* de telle sorte qu'il y ait une association de type paire entre la clé et la valeur. Ce TDA est parfois appelé tableau associatif mais nous n'utiliserons pas ce terme qui peut être ambigu.

Les opérations que l'on peut effectuer sont les suivantes :

1.  $\text{RechercherDico}(D, k)$  retourne un pointeur  $p$  sur l'élément du dictionnaire tel quel que  $p \rightarrow cle = k$ . Si aucun élément du dictionnaire n'a la clé demandée, un pointeur nul est renvoyé.
2.  $\text{InsererDico}(D, e)$  insère l'élément  $e$  dans le dictionnaire. Si un élément possède déjà la clé de  $e$  on met à jour sa valeur.

3. SupprimerDico( $D, k$ ) supprime l'élément du dictionnaire ayant pour clé  $k$ . Rien n'est fait si cette clé n'est pas présente dans le dictionnaire.

On ne peut optimiser les accès aux éléments que s'il existe un ordre total sur les clés.

## ► Applications

Il existe en général deux objectifs associés aux dictionnaires (qui ne sont pas forcément compatibles) :

1. Minimiser les coûts d'accès et modification (recherche, insertion, suppression)
2. Minimiser l'espace de stockage associé

Les applications sont très nombreuses, on citera la table des symboles d'un compilateur (la clé est le nom du symbole) ou la table de correspondance d'un serveur DNS (la clé est le nom de domaine).

## ► Implémentations

Il existe de nombreuses manières d'implémenter un dictionnaire.

**Liste chaînée.** Pour représenter un dictionnaire dans une liste chaînée il suffit de stocker une clé dans chaque cellule. Selon que l'on décide de trier ou de ne pas trier la liste, la complexité moyenne des opérations peut varier.

**Exercice.** (difficulté 2) Donner les complexités maximum et moyenne des opérations du dictionnaire implémenté dans une liste chaînée triée et non triée.

**Tableau trié.** L'avantage du tableau trié pour l'implémentation du dictionnaire réside dans son opération de recherche qui peut être optimisée grâce à une procédure dichotomique (temps logarithmique). Par contre, dès lors qu'il faut modifier les données, des cellules devront être déplacées, la complexité deviendra alors linéaire. Du point de vue de l'espace de stockage, cette solution ne perd pas de place (sauf si des cellules ne sont pas utilisées) contrairement à la solution de la liste chaînée mais aussi la solution d'arbre binaire de recherche que nous allons voir plus loin. C'est donc un choix avisé lorsque les recherches sont nombreuses mais les modifications ne le sont pas.

**Autres implémentations.** Un dictionnaire peut-être implémenté par un arbre binaire de recherche ou une table de hachage. Ces deux modes de représentation sont détaillés dans les sections suivantes.

## 5.8 Arbre binaire de recherche

Un arbre binaire de recherche est un arbre binaire qui stocke des valeurs sur chaque nœud avec une propriété particulière qui est que toutes les valeurs dans le sous-arbre de gauche sont inférieures à la valeur du nœud considéré et que toutes les valeurs du sous-arbre de droite sont supérieures à celle du nœud considéré. D'une manière formelle, soit  $a$  un nœud,  $l(a)$  son sous-arbre de gauche et  $r(a)$  son sous-arbre de droite. Par abus de langage, nous écrirons désignerons avec la même notation un nœud et sa valeur associée. Nous avons donc :

$$\begin{cases} \forall i \in l(a) \quad i < a \\ \forall i \in r(a) \quad i > a \end{cases}$$

Cette propriété est récursive, donc elle se propage pour tous les nœuds des sous-arbres.

L'avantage, une fois que cet arbre est construit, est qu'un parcours infixé (cf. page 72) de l'arbre est ordonné. Le deuxième avantage est que pour trouver une valeur dans la collection, on peut savoir à chaque nœud de l'arbre si la

---

**Algorithme 60 :** Recherche d'une valeur dans un arbre binaire de recherche (algorithme récursif).

---

**1 Structure ABR**

```

2   entier valeur
3   ABR * gauche
4   ABR * droit

```

**5 Fonction** RechercherABR(*abr,v*)

Entrée : ABR \* *abr*

Entrée : entier *v*

**Précondition** : *abr* contient un arbre binaire de recherche bien formé (éventuellement vide donc pointeur nul).

**Postcondition** : Retourne un pointeur vers le nœud trouvé ou nul si la valeur n'a pas été trouvée.

```

6   si abr = Ø ou abr → valeur = v alors
7       retourne abr
8   sinon
9       si v > abr → valeur alors
10      retourne RechercherABR(abr → droit,v)
11   sinon
12      retourne RechercherABR(abr → gauche,v)

```

---

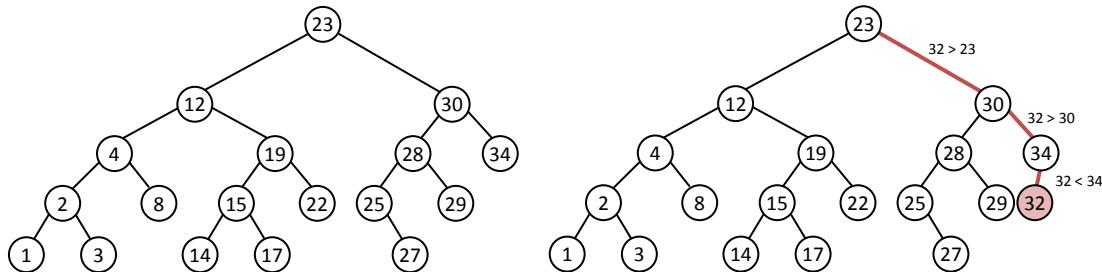


FIGURE 5.4 – Insertion du nœud 32 dans un arbre binaire de recherche. L'algorithme est assez proche de celui de recherche.

valeur se trouve du côté gauche ou droit (ou si on a trouvé cette valeur). Il est donc inutile de parcourir tous les nœuds de l'arbre mais seulement de trouver un chemin vers le nœud cible.

D'une manière complémentaire, l'insertion dans une arbre binaire de recherche est simple. Il suffit de faire une recherche et d'insérer l'élément à l'endroit où il aurait dû être si on l'avait trouvé. La figure 5.4 et l'algorithme 61 donnent le détail de cet algorithme.

**Exercice.** (difficulté 1) Simulez l'insertion des valeurs 20, 24 et 5 dans l'arbre binaire de recherche de la figure 5.4.

**Exercice.** (difficulté 3) Transformez l'algorithme 61 pour qu'il ne soit plus récursif.

**La suppression d'un élément** dans un arbre binaire de recherche est un peu plus complexe que l'insertion. En effet, il se peut que l'on soit amené à supprimer un nœud qui possède un ou plusieurs fils. Dans ce cas-là, il faut modifier l'arbre localement afin de raccorder les branches. La figure 5.5 montre les différents cas de figures qui peuvent se présenter pour la suppression d'un élément. Si on reprend le cas le plus complexe 4,  $b'$  représente la valeur inférieure la plus proche de  $a$ . Pour la trouver, il faut systématiquement considérer le fils droit et s'arrêter lorsqu'il n'y en a plus.

**Algorithme 61 :** Insertion d'une valeur dans un arbre binaire de recherche (algorithme récursif).

**1 Procédure** InsererABR(*abr,v*)

Entrée/Sortie : ABR \* *abr*

Entrée : entier *v*

Précondition : *abr* contient un arbre binaire de recherche bien formé (éventuellement vide donc pointeur nul).

Postcondition : L'élément *v* est inséré au bon endroit dans l'arbre binaire de recherche, sauf si cette valeur existe déjà dans l'arbre.

Déclaration : ABR \* *nouveau*

2   **si** *abr* =  $\emptyset$  **alors**

3     *nouveau*  $\leftarrow$  Allouer(*ABR*)

4     *nouveau*  $\rightarrow$  *valeur*  $\leftarrow$  *v*

5     *nouveau*  $\rightarrow$  *gauche*  $\leftarrow$   $\emptyset$

6     *nouveau*  $\rightarrow$  *droit*  $\leftarrow$   $\emptyset$

7     *abr*  $\leftarrow$  *nouveau*

8   **sinon**

9     **si** *v* > *abr*  $\rightarrow$  *valeur* **alors**

10       InsererABR(*abr*  $\rightarrow$  *droit*,*v*)

11     **sinon si** *v* < *abr*  $\rightarrow$  *valeur* **alors**

12       InsererABR(*abr*  $\rightarrow$  *gauche*,*v*)

Cette cellule peut avoir un fils à gauche en revanche, ce qui provoquera une autre modification locale de l'arbre. On peut faire exactement la même opération mais en symétrique, c'est-à-dire remplacer *a* par la valeur la plus proche mais supérieure. Dans ce cas, il faut considérer le fils le plus à gauche du sous-arbre de droite.<sup>1</sup>

### ► Arbre binaire équilibré

Un arbre binaire de recherche, lors de sa création, peut être amené à dégénérer vers une structure qui n'est pas équilibrée (si les valeurs sont ajoutées dans un ordre monotone par exemple). La figure 5.6 illustre le cas du remplissage d'un arbre binaire de recherche par des valeurs aléatoires (donc normalement un cas favorable).

Un arbre est dit équilibré (plus précisément H-équilibré) si l'écart maximum entre la profondeur des sous-arbres de gauche et droite est au plus de 1 quel que soit le nœud considéré (voir figure 5.7).

On notera que l'arbre complet est un cas particulier d'arbre H-équilibré qui implique que le nombre de noeuds soit  $2^{(h+1)} - 1$  où *h* est la hauteur de l'arbre.

**Exercice.** (difficulté 1) Est-ce que l'arbre représenté dans la figure 5.4 est H-équilibré ? (avant et après l'insertion)

L'opération d'insertion dans un arbre binaire de recherche peut faire disparaître la propriété d'équilibre comme le montre la figure 5.8. L'écart qui était de 1 devient de 2 à l'issue de cette insertion. Si l'insertion avait été dans le sous arbre de gauche, alors l'équilibre aurait été maintenu.

Il est possible grâce à quelques opérations assez simples et surtout peu coûteuses de maintenir un arbre équilibré.

### ► AVL

Les arbres AVL (de leurs auteurs Adelson-Velskii et Landis) sont une implémentation d'arbre binaire qui s'auto équilibre. Cet équilibrage se fait par des rotations locales. La figure 5.9 illustre le principe de rotation avec deux configurations

1. ne voyez rien de politique là-dedans

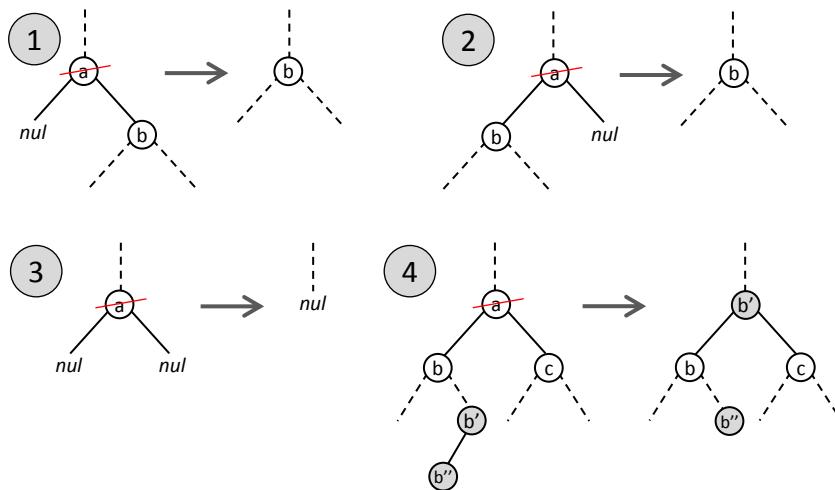


FIGURE 5.5 – Schéma de fonctionnement de la suppression d'un élément dans un arbre binaire de recherche. Lorsque l'élément à supprimer est une feuille il suffit de le supprimer (cas 3). Lorsque l'élément à supprimer n'a qu'un seul fils (cas 1 et 2), il faut faire glisser ce fils pour remplacer le nœud supprimé. Quand les deux fils sont définis, on remplace la valeur à supprimer par celle la plus proche numériquement, comme celle la plus à droite du sous-arbre de gauche (dans le schéma  $b'$ ). Si la cellule avait un fils (dans le schéma  $b''$ ), on fait aussi remonter ce dernier (cas 4).

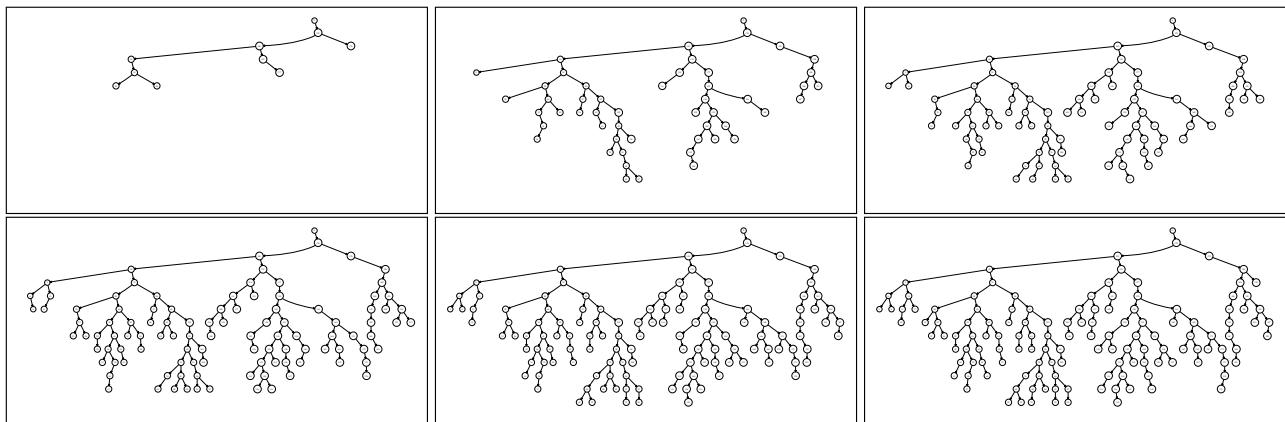


FIGURE 5.6 – Remplissage d'un arbre binaire de recherche. L'arbre final a une hauteur de 12 alors que s'il avait été complet, il aurait fallu seulement une hauteur de 7.

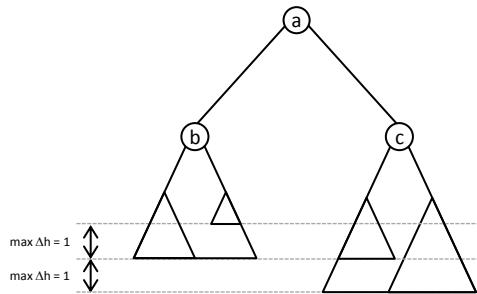


FIGURE 5.7 – Un arbre binaire H-équilibré a un écart maximum de 1 entre la profondeur maximale de ses sous-arbres, quel que soit le nœud considéré. Ici sont représentés 3 nœuds  $a$ ,  $b$  et  $c$ . On notera au passage qu'un arbre H-équilibré n'a pas un écart de 1 entre la profondeur de toutes ses feuilles.

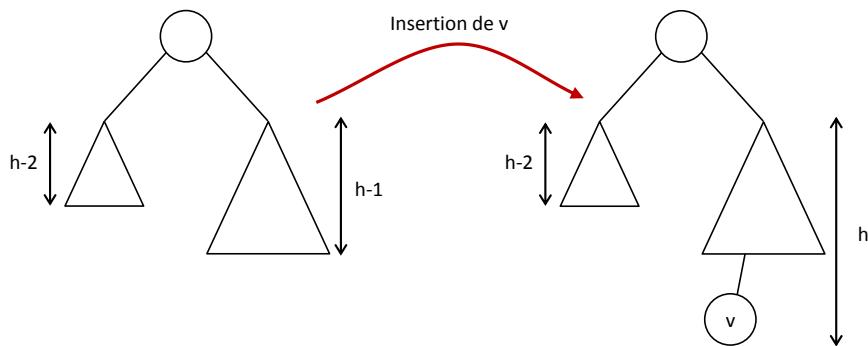


FIGURE 5.8 – Un arbre binaire de recherche perd sa propriété d'équilibre après une insertion.

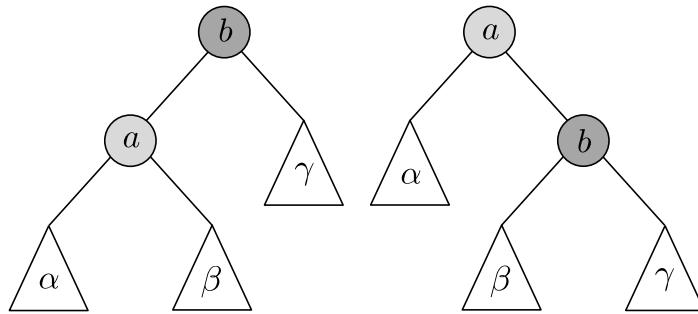


FIGURE 5.9 – Principe de rotation autour des nœuds  $a$  et  $b$ . Les configurations de droite et gauche sont équivalentes car les propriétés de l’arbre binaire de recherche sont conservées.

qui sont équivalentes au sens des propriétés d’un arbre binaire de recherche mais dont la hauteur locale est différente. Soient  $\alpha$ ,  $\beta$  et  $\gamma$  trois sous-arbres comme illustré dans la figure 5.9. Par définition, nous avons :

$$\begin{cases} \forall k \in \alpha & k < a \\ \forall k \in \beta & a < k < b \\ \forall k \in \gamma & b < k \end{cases}$$

Ces relations sont respectées dans la configuration de droite comme celle de gauche. De plus, le fait de remplacer le nœud  $a$  par le nœud  $b$  dans un arbre plus grand que celui-là ne pose pas de problème. Pour le remarquer, il suffit d’imaginer que ce sous-arbre est pris entre deux bornes et que  $a$  et  $b$  sont forcément dans un même intervalle. Les intervertir ne change donc rien par rapport à cet intervalle.

L’intérêt d’une rotation est qu’elle change localement la hauteur de l’arbre. Soient  $h(\alpha)$ ,  $h(\beta)$  et  $h(\gamma)$  les hauteurs de chacun des sous-arbres. La hauteur de l’arbre dans sa configuration à gauche est  $\max(h(\alpha)+2, h(\beta)+2, h(\gamma)+1)$  alors que celle de droite a pour hauteur  $\max(h(\alpha)+1, h(\beta)+2, h(\gamma)+2)$ . Si  $\alpha$  et  $\gamma$  ont des hauteurs différentes, alors les hauteurs de chaque configuration peuvent être différentes et cela permet de rééquilibrer localement le sous-arbre.

### ► Arbre rouge-noir

Il existe d’autres manières de garder un arbre équilibré dont une technique reposant sur l’étiquetage avec deux couleurs (rouge et noir) des nœuds de l’arbre. Cette technique est plus performante pour l’insertion et la suppression mais ne maintient pas tout à fait la propriété d’équilibre, elle est donc plus lente que l’AVL pour la recherche d’éléments (hauteur plus grande).

## 5.9 Table de hachage

Les implémentations de dictionnaire que nous avons vues jusqu’à présent permettent d’atteindre des complexités pour la recherche, l’insertion et la suppression au mieux logarithmiques. Nous allons maintenant changer complètement de philosophie afin d’améliorer ces complexités. La table de hachage est un moyen de faire correspondre une case d’un tableau à une valeur de clé grâce à une fonction (dite fonction de hachage). Le calcul de la fonction se fait en temps constant et ainsi dans le meilleur des cas, on accède à la case en temps constant aussi.

### ► Tableau à accès direct

Nous allons commencer par voir le fonctionnement d’un tableau à accès direct qui est la base d’une table de hachage. Soit  $\Omega = \{0, \dots, n-1\}$  l’ensemble des valeurs possibles pour la clé. On veillera à ce que  $n$  ne soit pas trop grand

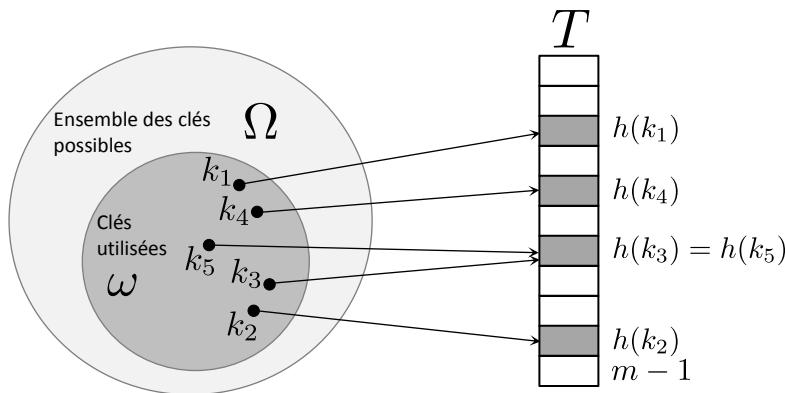


FIGURE 5.10 – Principe de la collision : deux clés (ici  $k_3$  et  $k_5$ ) donnent la même valeur de hachage par la fonction  $h$  et sont censées être stockées dans la même case du tableau.

car il est directement lié à la taille du tableau. On suppose aussi que deux éléments du dictionnaire ne peuvent pas avoir la même valeur de clé.

Le tableau à accès direct est un tableau de taille  $n$  contenant des pointeurs. Si l'élément du dictionnaire a une clé de valeur  $k$ , alors on stocke le pointeur vers cet élément dans  $T[k]$ , sinon on laisse le pointeur nul à cette clé.

Ce dictionnaire a un temps d'accès et de recherche qui est constant puisque la simple donnée de la clé permet d'accéder directement à la valeur. Par contre, l'espace de stockage est directement lié à la taille de l'ensemble des valeurs possibles, donc  $n$ . Si cet ensemble est beaucoup plus grand que l'ensemble des valeurs réellement utilisées, alors beaucoup d'espace est gaspillé. Comment éviter cela ?

La table de hachage répond à ce problème en réduisant la taille du tableau à une valeur plus proche du nombre de valeurs réellement utilisées.

### ► Table de hachage : principe

L'idée est d'utiliser un tableau dont la taille  $m$  est bien inférieure à celle de  $\Omega$  :

$$m \ll n$$

et d'utiliser une fonction  $h$  (dite de hachage) pour passer de l'ensemble des valeurs de clés à l'indice dans le tableau :

$$h : \Omega \rightarrow \{0, \dots, m-1\}$$

Ainsi au lieu d'utiliser la case  $T[k]$  du tableau, nous allons utiliser  $T[h(k)]$ . Au passage, on verra que les valeurs de  $\Omega$  n'ont plus besoin d'être des entiers comme dans le tableau à accès direct précédent. Il suffit de construire une fonction dont le résultat est un entier inférieur à  $m$ . La gestion d'une telle table de hachage est détaillée dans les algorithmes 62.

Ces algorithmes naïfs ne peuvent pas marcher pour une raison très simple : la fonction  $h$  fait passer d'un ensemble de cardinal  $n$  à un ensemble de cardinal  $m$  bien inférieur. Cette fonction ne peut donc pas être injective (toute valeur dans l'espace d'arrivée peut avoir plusieurs antécédents dans l'espace de départ). Si un tel cas se produit, alors deux clés de  $\Omega$  vont être stockées dans la même case du tableau de hachage. Cela s'appelle une collision (figure 5.10). Il existe deux manières principalement de gérer la collision : l'adressage ouvert ou l'adressage fermé.

### ► Chaînage

Une solution simple consiste à effectivement stocker plusieurs éléments dans la même case du tableau par une liste chaînée par exemple. Cet adressage est dit fermé car on va stocker les informations relatives à une clé à l'endroit

**Algorithme 62 : Opérations de base dans une table de hachage (version naïve).**

```

1 Structure Element
2   Ω cle
3   V valeur
4 Procédure InsererHash( $T, x$ )
5   Entrée/Sortie : Element *  $T$ 
6   Entrée : Element *  $x$ 
7   Précondition :  $T$  contient un table de hachage basée sur la fonction  $h$ .
8   Postcondition : L'élément  $x$  est inséré à la position  $h(x \rightarrow \text{cle})$ .
9    $T[h(x \rightarrow \text{cle})] \leftarrow x$ 
10
11 Procédure SupprimerHash( $T, k$ )
12   Entrée/Sortie : Element *  $T$ 
13   Entrée :  $\Omega k$ 
14   Précondition :  $T$  contient un table de hachage basée sur la fonction  $h$ .
15   Postcondition : L'élément dont la clé est  $k$  est supprimé à la position  $h(k)$ .
16    $T[h(k)] \leftarrow \emptyset$ 
17
18 Fonction RechercherHash( $T, k$ )
19   Entrée/Sortie : Element *  $T$ 
20   Entrée :  $\Omega k$ 
21   Précondition :  $T$  contient un table de hachage basée sur la fonction  $h$ .
22   Postcondition : Retourne le pointeur présent à la position  $h(k)$  (peut être nul).
23   retourne  $T[h(k)]$ 

```

que la fonction de hachage a indiqué. Soit  $\omega \subset \Omega$  l'ensemble des clés réellement utilisées par le dictionnaire et  $u = \text{card}(\omega)$ . La figure 5.11 illustre le principe de l'adressage ouvert.

Les complexités des opérations d'ajout, suppression et recherche dans la table de hachage construite vont directement dépendre du rapport entre  $u$ , le nombre de clés réellement utilisées et  $m$  le nombre de cases disponibles dans le tableau. En effet, la recherche dans la liste chaînée est linéaire en fonction du nombre d'éléments qui se trouvent dedans. Si on arrive à garantir que ce nombre d'éléments est de l'ordre de 1, alors les opérations restent en temps constant. Pour arriver à cela il faut garantir que  $u = \mathcal{O}(m)$  autrement dit, que le tableau est choisi en fonction du nombre d'éléments que l'on souhaite mettre dedans.

On peut très bien utiliser autre chose qu'une liste chaînée derrière une table de hachage, par exemple rien n'empêche de mettre un arbre binaire de recherche.

**Exercice.** Discuter de l'intérêt de mettre un arbre binaire de recherche derrière une table de hachage.

### ► Adressage ouvert

L'adressage ouvert est plus complexe à mettre en œuvre que la solution précédente. Il consiste à placer l'information dans une autre case du tableau lorsqu'il y a collision. Il existe plusieurs possibilités pour trouver une case libre. La plus simple consiste à prendre la première libre après la valeur désignée par la fonction de hachage (voir figure 5.12). L'adressage ouvert est peu adapté à la suppression qui va placer des cases vides et empêcher le fonctionnement de la fonction de recherche. Une solution à ce problème consiste à utiliser une valeur spéciale de pointeur (ou une structure combinant l'information de statut avec l'information à stocker) pour désigner une case supprimée. Ainsi, pour l'insertion, cette case sera considérée comme vide, alors que pour la recherche, on la considérera comme une case pleine et on continuera de chercher. S'il y a beaucoup de suppressions, la complexité de recherche deviendra de plus

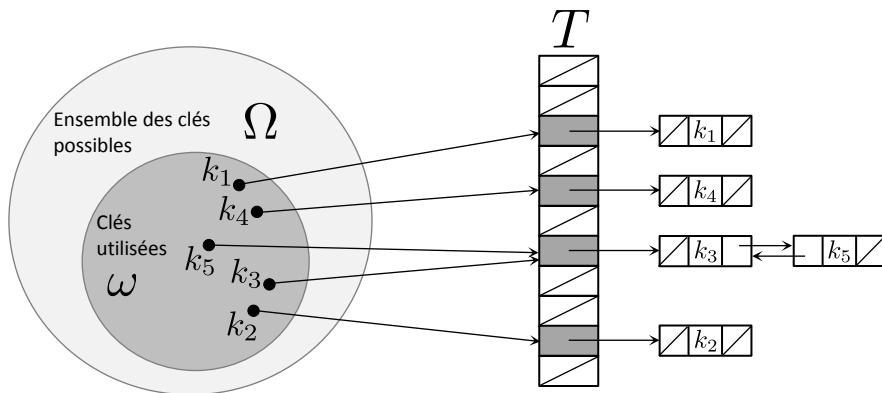


FIGURE 5.11 – Adressage fermé : chaque case du tableau est un pointeur vers une liste chaînée (ici doublement) dont le nombre d’éléments dépend du nombre de collisions qu’il y a eu.

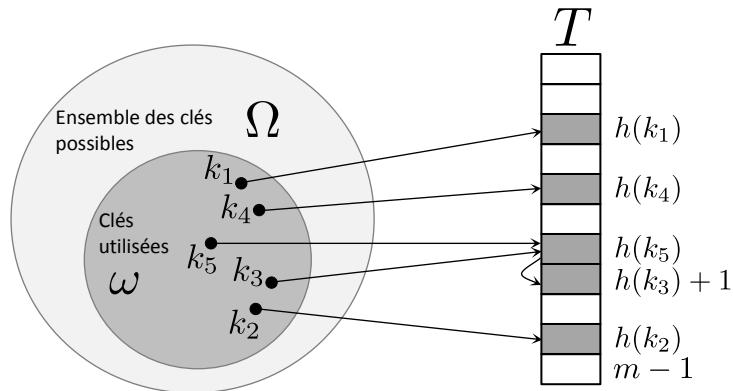


FIGURE 5.12 – Adressage ouvert : chaque case du tableau est utilisée une seule fois. Lorsqu'il y a collision, on cherche une autre case libre (dans cette exemple,  $k_3$  a été inséré après et on lui attribué la première case libre qui est la suivante).

en plus grande, mais les cases supprimées resteront utilisables dans le futur pour des insertions.

## ► Fonctions de hachage

Les tables de hachage reposent sur des fonctions de hachages qui doivent garantir de bonnes propriétés. En outre, elles devront garantir que la distribution des valeurs est bien uniforme. En plus de cela, il faut que la fonction de hachage soit rapide à évaluer.

**Codage.** Avant de hacher, il convient de disposer d’une valeur entière naturelle. Pour une chaîne de caractères codée en ASCII, on peut appliquer une fonction simple qui garantira que deux chaînes différentes ont une valeur différente :

$$\text{code(chaine)} = \sum_{i=0}^{\text{taille(chaine)}} \text{chaine}[i] 128^i$$

Ici 128 est utilisé car les codes ASCII s’arrêtent à 128.

Pour chaque type de données il faut définir préalablement le codage sous forme d’entier de la valeur.

**Méthode basée sur le reste.** Une méthode classique et rapide pour que la valeur de hachage se trouve bien dans l'intervalle  $[0, \dots, m - 1]$  est de faire une division par  $m$  et d'en prendre le reste (opération modulo) :

$$h(k) = k \bmod m$$

L'inconvénient de cette méthode est qu'elle ne prend en compte que les bits les moins significatifs de  $k$ . Si la valeur de  $m$  coïncide avec le codage utilisé pour calculer  $k$ , les données en entrées ne seront pas toutes prises en compte. Par exemple, avec le codage proposé pour les chaînes de caractères, si on prend  $m = 128$ , alors  $h(\text{code(chaine)})$  ne dépendra que de  $\text{chaine}[0]$ . Ainsi, toutes les chaînes qui commencent par la même lettre auront la même place dans la table de hachage. Il existe d'autres problèmes liés à l'utilisation de la fonction de reste pour le calcul d'un hachage lorsque les données sont périodiques.

La bonne valeur à utiliser pour  $m$  (donc pour la taille de la table) est un nombre entier qui ne soit pas trop proche d'une puissance de 2.

**Méthode basée sur les parties entières.** Une méthode plus lente consiste à appliquer la fonction suivante :

$$h(k) = \lfloor m \operatorname{frac}(\lambda k) \rfloor$$

où  $\lambda \in [0, 1]$  et  $\operatorname{frac}(x) = x - \lfloor x \rfloor$ . Bien entendu, le choix de  $\lambda$  est très important, il convient de ne pas utiliser une fraction (sauf cas particulier) mais plutôt un nombre irrationnel.

**Exercice.** Pour  $\lambda = \frac{1}{2}$ , expliciter les valeurs que peuvent prendre  $h(k)$  et pourquoi ce n'est pas un bon choix.

En pratique, certaines valeurs donnent de bons résultats comme :

$$\lambda = \frac{\sqrt{5} - 1}{2}$$

## ► Comparaison

Le chaînage (adressage fermé) a l'avantage de pouvoir gérer un nombre quelconque d'éléments et de collisions. Les performances sont plus stables car la suppression n'engendre pas de détérioration. En revanche la gestion de la liste chaînée sous-jacente a un coût mémoire.

Concernant l'adressage ouvert, il est rapide et n'a pas de surcoût mémoire lié à l'utilisation de pointeurs. Par contre, la choix de la fonction de hachage est plus délicat (il faut éviter d'avoir des valeurs proches entre elles). Un autre inconvénient majeur est que l'on ne peut pas stocker plus d'éléments que la taille initiale du tableau sans procéder à un rehashage. De plus, la suppression pose quelques problèmes de performances si elle est faite trop souvent.

En pratique, il faut donc tenir compte des contraintes. Important : lorsque vous utilisez une table de hachage d'une bibliothèque standard (comme la STL par exemple) vérifiez bien quel est son fonctionnement afin que cela soit adapté à votre application. Les tables de hachage sont souvent paramétrées, encore faut-il se documenter sur ces paramètres. Concernant la comparaison avec les autres méthodes pour un dictionnaire comme l'arbre binaire de recherche, n'oubliez pas que la table de hachage n'a pas de notion d'ordre. Conclusion : si vous souhaitez absolument avoir des clés triées, alors la table de hachage n'est pas compatible. Inversement, lorsque le tri n'est pas obligatoire, la table de hachage est un choix à privilégier car il offre de meilleures performances que l'arbre binaire de recherche.

