Algorithmique Avancée et Programmation en C

v20260921

Nicolas Delestre

Table des matières

Avant-propos **9**

1 Introduction au langage C **11**
1.1 Le langage C, un langage compilé . . . 11
1.1.1 Contexte et historique . . . 11
1.1.2 Du code source au programme exécutable . . . 13
1.1.3 Bibliothèques statiques et dynamiques . . . 14
1.1.4 Principales bibliothèques standards . . . 15
1.1.5 Synthèse de la chaîne de compilation . . . 16
1.2 Les bases du langage . . . 17
1.2.1 Types simples . . . 17
1.2.2 Variables et portée . . . 18
1.2.3 Types énumérés . . . 19
1.2.4 Alias de type . . . 19
1.2.5 Constantes . . . 20
1.2.6 Opérateurs . . . 21
1.2.7 Instructions . . . 22
1.2.8 Instructions conditionnelles . . . 23
1.2.9 Instructions itératives . . . 24
1.3 Pointeurs et fonctions . . . 25
1.3.1 Mémoire, adresses et représentation . . . 25
1.3.2 Déclaration et initialisation d'un pointeur . . . 26
1.3.3 Les opérateurs `&` et `*` . . . 27
1.3.4 Fonctions en langage C . . . 28
1.3.5 Passage de paramètres aux fonctions . . . 29
1.4 Entrée et sortie standard . . . 30
1.4.1 Sortie standard : `printf` . . . 30
1.4.2 Entrée standard : `scanf` . . . 31
1.4.3 Fichiers . . . 32
1.5 Types composites . . . 33
1.5.1 Tableaux . . . 34
1.5.2 Chaînes de caractères . . . 35
1.5.3 Structures . . . 36
1.5.4 Unions . . . 38
1.6 Particularités du langage C . . . 38
1.6.1 Le programme principal . . . 38
1.6.2 Pointeurs de fonction . . . 39
1.6.3 Les bibliothèques . . . 40
1.6.4 Inclusions multiples et ordre définition / inclusion . . . 41
1.6.5 Macros paramétrées (macro-fonctions) . . . 42
1.6.6 Mots clés `static` et `extern` . . . 43
1.6.7 Allocation statique et allocation dynamique . . . 44

1.6.8 La bibliothèque standard . . . 45
1.7 Compilation d'un grand projet . . . 47
1.7.1 Une première solution : un script `bash` . . . 47
1.7.2 Pourquoi passer à `make` . . . 47
1.7.3 Un premier `Makefile` équivalent au script . . . 47
1.7.4 Factorisation avec des variables de `make` . . . 48
1.7.5 Règles d'inférence et variables automatiques . . . 49

2 Les types abstraits de données 51
2.1 De l'analyse au développement . . . 51
2.1.1 Types et analyse . . . 51
2.1.2 Qu'est-ce qu'un type de données ? . . . 51
2.1.3 Les étapes de création d'un programme . . . 52
2.2 Définition et spécification d'un type abstrait de données . . . 52
2.2.1 Les constituants d'un TAD . . . 52
2.2.2 Les opérations d'un TAD . . . 53
2.2.3 Le comportement des opérations . . . 53
2.2.4 Les préconditions . . . 53
2.2.5 Un formalisme de spécification . . . 54
2.2.6 Le formalisme axiomatique . . . 54
2.2.7 Le formalisme avec sémantiques en langue naturelle . . . 54
2.3 Les TAD élémentaires . . . 55
2.3.1 Le TAD booléen . . . 55
2.3.2 Le TAD naturel . . . 56
2.3.3 Le TAD réel . . . 57
2.3.4 Le TAD caractère . . . 58
2.3.5 Le TAD chaîne . . . 58
2.4 Un exemple : le TAD Point2D . . . 58
2.4.1 L'analyse : définir les opérations du point . . . 59
2.4.2 La conception préliminaire : fixer les signatures . . . 59
2.4.3 La conception détaillée : choisir une représentation . . . 59
2.4.4 Utiliser le point sans connaître sa représentation . . . 60
2.5 Les règles de conception . . . 61
2.5.1 Lors de l'analyse : une spécification complète et consistante . . . 61
2.5.2 Lors de la conception préliminaire : fonctions et procédures . . . 61
2.5.3 Lors de la conception détaillée : identifier les opérations utilisées . . . 62
2.6 L'implantation d'un TAD en C . . . 62
2.6.1 Uniquement des fonctions . . . 62
2.6.2 Séparer l'interface et l'implantation . . . 63
2.6.3 Adapter l'interface aux possibilités du C . . . 63
2.6.4 Prévoir la libération, la copie et l'égalité . . . 63
2.6.5 Traduire l'exemple Point2D . . . 64
2.7 Documentation et tests unitaires . . . 65
2.7.1 Documenter l'interface avec Doxygen . . . 65
2.7.2 Vérifier le comportement par des tests unitaires . . . 67
2.7.3 Organiser les tests avec CUnit . . . 68
2.7.4 Tester les axiomes de Point2D . . . 68
2.8 L'organisation du projet . . . 70
2.8.1 Répartir les fichiers selon leur rôle . . . 70
2.8.2 Donner les informations nécessaires à l'utilisation du projet . . . 70
2.8.3 Automatiser la construction avec *make* . . . 71
2.9 Synthèse . . . 71

3 Les types abstraits de données collections **73**
3.1 Introduction aux collections . . . 73
3.1.1 Des TAD paramétrés par le type des éléments . . . 73
3.1.2 Un ensemble d'outils pour la conception des algorithmes . . . 73
3.2 Les collections linéaires et le TAD Ensemble . . . 74
3.2.1 La pile : le dernier entré est le premier sorti . . . 74
3.2.2 La file : le premier entré est le premier sorti . . . 75
3.2.3 La liste : accéder aux éléments par leur position . . . 76
3.2.4 La liste ordonnée : maintenir un ordre entre les éléments . . . 76
3.2.5 L'ensemble : l'appartenance sans ordre ni répétition . . . 77
3.2.6 Parcourir une collection avec « pour chaque » . . . 78
3.3 Le TAD associatif Dictionnaire . . . 78
3.3.1 Les clés et les valeurs . . . 78
3.3.2 Spécification des opérations . . . 79
3.3.3 Obtenir les clés et les valeurs . . . 79

4 Les structures de données dynamiques **81**
4.1 Pourquoi des structures de données dynamiques ? . . . 81
4.2 Mémoire, pointeurs et allocation . . . 82
4.2.1 Organisation de la mémoire . . . 82
4.2.2 Allocation statique, automatique et dynamique . . . 82
4.2.3 Référencer et déréférencer . . . 83
4.2.4 Réserver et libérer une zone . . . 83
4.3 Les listes simplement chaînées . . . 83
4.3.1 Représentation et accès aux champs . . . 84
4.3.2 Ajouter en tête . . . 85
4.3.3 Supprimer la tête et libérer la liste . . . 85
4.4 Les algorithmes sur les listes chaînées . . . 85
4.4.1 Parcours itératif et récursif . . . 85
4.4.2 Supprimer la première occurrence : version récursive . . . 86
4.4.3 Supprimer la première occurrence : version itérative . . . 86
4.4.4 Autres traitements à concevoir . . . 87
4.5 Les listes doublement chaînées . . . 87
4.5.1 Représentation et cohérence des liens . . . 87
4.5.2 Les opérations élémentaires . . . 88
4.5.3 Exercice de conception détaillée . . . 89
4.6 Une représentation mixte : la table de hachage . . . 89
4.6.1 Choisir une case par une fonction de hachage . . . 89
4.6.2 Exemple : conserver le prénom et l'âge de personnes . . . 89
4.6.3 Répartition et coût des opérations . . . 89
4.7 Les structures dynamiques hiérarchiques : l'arbre binaire . . . 90
4.7.1 Représentation par des nœuds . . . 90
4.7.2 Construction et consultation . . . 91
4.7.3 Modification et suppression . . . 91
4.7.4 Exercice de conception détaillée . . . 91
4.8 Synthèse : manipuler les liens avec rigueur . . . 92

5 La conception des TAD collections **93**
5.1 De la spécification à la représentation . . . 93
5.2 La conception préliminaire des collections . . . 93
5.2.1 Pile . . . 93
5.2.2 Pile . . . 94
5.2.3 Liste . . . 94
5.2.4 Listes ordonnées . . . 94

5.2.5 Ensemble . . . 95
5.2.6 Dictionnaire . . . 95
5.2.7 Tas et arbres binaires de recherche . . . 95
5.2.8 Utilisation et contraintes de réalisation . . . 96
5.3 Les représentations à l'aide de tableaux . . . 96
5.3.1 Une pile dans un tableau . . . 96
5.3.2 Une liste dans un tableau . . . 97
5.3.3 Portée et limites de la représentation . . . 98
5.4 Les représentations à l'aide de listes chaînées . . . 99
5.4.1 Pile : placer le sommet en tête . . . 99
5.4.2 File : conserver les deux extrémités . . . 99
5.4.3 Listes et listes ordonnées : conserver la longueur . . . 100
5.4.4 Ensemble : garantir l'absence de doublons . . . 101
5.5 La conception d'un arbre binaire de recherche . . . 101
5.5.1 Recherche et insertion . . . 102
5.5.2 Supprimer un élément . . . 102
5.5.3 La forme de l'arbre dépend des insertions . . . 104
5.6 Maintenir l'équilibre : les arbres AVL . . . 104
5.6.1 Rotations simples et doubles . . . 104
5.6.2 Insertion et rééquilibrage . . . 105
5.6.3 Exercice : suppression dans un AVL . . . 107
5.7 Représenter un arbre n-aire par un arbre binaire . . . 107
5.7.1 Exercice de conception . . . 108
5.8 Les représentations à l'aide de tables de hachage . . . 108
5.8.1 Représenter un dictionnaire . . . 108
5.8.2 Représenter un ensemble . . . 109
5.8.3 Conditions des performances attendues . . . 109
5.9 Synthèse : choisir une représentation adaptée . . . 109
5.9.1 Comparer les collections linéaires . . . 109
5.9.2 Comparer les accès par valeur ou par clé . . . 110
5.9.3 Interpréter les coûts des structures hiérarchiques . . . 110
5.9.4 Relier efficacité et respect de la spécification . . . 110

6 Implantation des structures de données dynamiques en C 111
6.1 De la conception à l'implantation . . . 111
6.2 Un exemple directeur : le TAD Polynome . . . 111
6.2.1 Spécifier les services . . . 111
6.2.2 Conception préliminaire . . . 112
6.3 Conception détaillée : représenter le polynôme par des monômes . . . 112
6.4 Première implantation : une liste chaînée de monômes . . . 113
6.4.1 Définir la représentation et l'interface . . . 113
6.4.2 Ajouter et supprimer un nœud . . . 113
6.4.3 Détruire, copier et comparer une liste . . . 114
6.5 Deuxième implantation : une liste chaînée générique . . . 115
6.5.1 Séparer le chaînage du type des éléments . . . 115
6.5.2 Paramétrer les opérations sur les éléments . . . 115
6.5.3 Ajouter un élément en conservant une copie . . . 116
6.5.4 Gérer la durée de vie des objets . . . 116
6.6 Utiliser la liste générique pour implanter les polynômes . . . 117
6.6.1 Adapter les opérations sur les monômes . . . 117
6.6.2 Créer un polynôme . . . 117
6.6.3 Consulter un coefficient . . . 117
6.6.4 Préserver les invariants lors des autres opérations . . . 118
6.7 Synthèse : implantation spécialisée ou générique . . . 118

7 Les graphes **121**
7.1 Modéliser des relations par un graphe . . . 121
7.1.1 Vocabulaire . . . 121
7.2 Les TAD Graphe . . . 122
7.2.1 Graphe non orienté . . . 122
7.2.2 Graphe orienté . . . 123
7.3 Représenter un graphe . . . 124
7.3.1 Matrice d'adjacence . . . 124
7.3.2 Matrice d'incidence . . . 124
7.3.3 Listes d'adjacence . . . 125
7.4 Parcourir un graphe . . . 125
7.5 Ordonner les sommets d'un graphe orienté . . . 126
7.6 Rechercher un plus court chemin . . . 127
7.6.1 Algorithme de Dijkstra . . . 127
7.6.2 Guider la recherche avec A* . . . 129
7.7 Synthèse . . . 131

8 La programmation dynamique **133**
8.1 Des calculs récursifs inutilement répétés . . . 133
8.2 Principes de la programmation dynamique . . . 134
8.3 Tabuler les coefficients binomiaux . . . 134
8.3.1 Réduire le tableau à une dimension . . . 135
8.4 Un autre exemple classique : le problème du sac à dos . . . 136
8.4.1 Explorer tous les choix . . . 137
8.4.2 Définir les sous-problèmes . . . 138
8.4.3 Calculer la valeur optimale . . . 138
8.4.4 Retrouver les objets choisis . . . 139
8.5 Synthèse et autres applications . . . 140

9 Les arbres B **143**
9.1 Des arbres adaptés au stockage externe . . . 143
9.2 Définition et organisation . . . 143
9.2.1 Capacité et hauteur . . . 143
9.2.2 TAD ArbreB . . . 144
9.3 Rechercher une clé . . . 145
9.4 Insérer une clé . . . 145
9.4.1 Insertion sans changement de hauteur . . . 146
9.4.2 Augmentation de la hauteur . . . 146
9.4.3 Organisation de l'algorithme . . . 146
9.5 Supprimer une clé . . . 147
9.5.1 Supprimer dans une feuille . . . 147
9.5.2 Redistribuer les clés . . . 148
9.5.3 Supprimer une clé interne . . . 148
9.5.4 Propagation et diminution de la hauteur . . . 149
9.5.5 Algorithme de suppression . . . 149
9.6 Synthèse . . . 151

Avant-propos

Ce polycopié a été élaboré à partir de mes supports de cours.

La rédaction de ce document a été réalisée avec l'aide de *ChatGPT*, un modèle d'intelligence artificielle développé par OpenAI. ChatGPT a été utilisé comme outil de co-rédaction à partir de mes supports de cours : structuration du polycopié, reformulation et développement des explications, transformation du contenu des présentations en texte continu, production et adaptation d'exemples, et mise en cohérence des différents chapitres.

Le contenu pédagogique, les choix de notions abordées, leur organisation générale et les supports ayant servi de sources restent issus de mes enseignements. Les textes produits avec ChatGPT ont été relus, discutés et modifiés au cours de la rédaction.

Cette mention a pour objectif de rendre explicite l'utilisation d'un système d'intelligence artificielle générative dans la production du document et de distinguer cet usage des sources pédagogiques et scientifiques sur lesquelles le contenu s'appuie.

Chapitre 1

Introduction au langage C

1.1 Le langage C, un langage compilé

1.1.1 Contexte et historique

Le langage C est historiquement situé à l'interface entre le matériel et le logiciel. Il a été conçu pour permettre l'écriture de programmes efficaces et portables, tout en conservant une proximité forte avec le fonctionnement interne des machines. Pour comprendre les choix de conception du langage, ses forces et ses contraintes, il est nécessaire de replacer son utilisation dans le contexte de l'architecture des ordinateurs et de l'évolution des microprocesseurs.

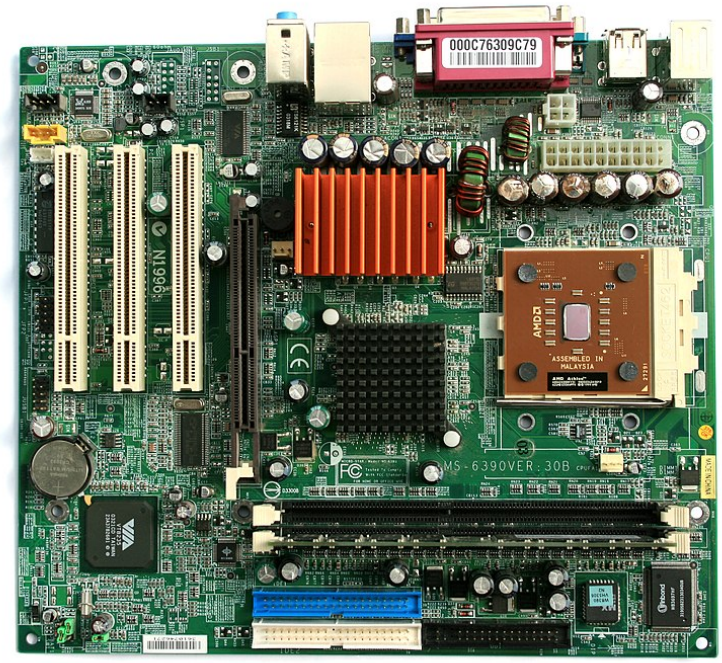


FIGURE 1.1 – Principaux composants matériels d'un ordinateur

Comme l'indique la figure 1.1, un ordinateur est constitué de plusieurs composants matériels interconnectés. Le processeur occupe une place centrale : il exécute les instructions des programmes et effectue les opérations sur les données. La mémoire vive contient à la fois les données manipulées

et le code des programmes en cours d'exécution. Les ports d'extension et les interfaces permettent la communication avec les périphériques externes, mais du point de vue du programmeur, ces éléments restent accessibles indirectement, principalement par l'intermédiaire du processeur et du système d'exploitation.

Le processeur est l'élément chargé d'exécuter les instructions élémentaires d'un programme. Un microprocesseur désigne un processeur dont l'ensemble des composants a été suffisamment miniaturisé pour être regroupé dans un unique boîtier. Les microprocesseurs se caractérisent notamment par leur jeu d'instructions, par la complexité de leur architecture interne, par le nombre de bits qu'ils peuvent traiter simultanément et par leur fréquence de fonctionnement. Ces caractéristiques influencent directement les performances, mais également la manière dont le code machine est généré et exécuté.

L'évolution des microprocesseurs au cours des dernières décennies met en évidence une augmentation considérable de la puissance de calcul et de la complexité des architectures. Le tableau 1.1 présente quelques microprocesseurs emblématiques et illustre cette évolution à travers des critères tels que l'année d'apparition, le nombre de transistors, la fréquence ou encore la largeur de mot.

Année	Fabricant	Nom	Nb instructions	Nb transistors	Fréquence	Nb bits
1979	Motorola	68000	82	68 000	2 à 12 MHz	16/32
1982	Intel	80286	99	134 000	6 à 16 MHz	16/16
1983	Acorn	ARM1	≈ 50	25 000	$\leq$ 8 MHz	32/26
1985	Intel	80386	155	275 000	16 à 40 MHz	32/32
1992	IBM	PowerPC 601		2 800 000	50 à 120 MHz	32/32
1993	Intel	Pentium	168	3 100 000	60 à 233 MHz	32/64
1995	AMD	K5	168	4 300 000	75 à 100 MHz	32/64
⋮	⋮	⋮	⋮	⋮	⋮	⋮
2021	IBM	Power10		3 600 000 000	4 GHz	64
2022	Intel	i9 12900H	981	4 150 000 000	5 GHz	64
2022	Apple	M2 (ARMV8-5)	"232"	20 000 000 000	$\leq$ 3,4 GHz	64

TABLE 1.1 – Historique de quelques microprocesseurs

Ce tableau met en évidence plusieurs tendances majeures, notamment l'augmentation spectaculaire du nombre de transistors et la généralisation des architectures 64 bits. Il illustre également la diversité des familles de microprocesseurs et des choix technologiques réalisés par les industriels. Le cas d'Apple est particulièrement représentatif de cette évolution. Au cours de son histoire, Apple a successivement utilisé des microprocesseurs Motorola, puis des processeurs PowerPC conçus avec IBM, avant de migrer vers les processeurs Intel, et plus récemment vers des microprocesseurs de type ARM développés en interne. Ces transitions, motivées par des contraintes de performance, de consommation énergétique et d'intégration matérielle, montrent qu'un même logiciel peut être amené à être recompilé sur des architectures très différentes, renforçant l'importance de la portabilité offerte par des langages comme le C.

Indépendamment de l'architecture matérielle, un principe fondamental demeure : tout est octet dans une machine. Les données manipulées par les programmes, qu'il s'agisse d'entiers, de réels, de caractères ou de structures plus complexes, sont représentées en mémoire sous forme de suites d'octets. Les programmes eux-mêmes sont également stockés sous forme d'octets, correspondant à des instructions codées selon le jeu d'instructions du microprocesseur. La distinction entre code et données est donc une distinction logique, mais pas une distinction physique, puisqu'ils partagent le même type de représentation en mémoire.

Le code machine, directement exécuté par le microprocesseur, est peu lisible pour un humain. Le langage assembleur fournit une représentation symbolique de ce code, dans laquelle chaque instruction machine est associée à un mnémonique plus compréhensible. L'opération qui consiste à traduire un programme écrit en assembleur vers son équivalent en code machine est appelée assemblage. Inversement, le désassemblage consiste à reconstruire une représentation assembleur à partir d'un code machine existant.

Afin de faciliter le développement de logiciels complexes et portables, des langages de plus haut niveau ont été conçus pour exprimer les algorithmes de manière plus abstraite que l'assembleur.

Le langage C appartient à cette catégorie de langages compilés. Le code source est analysé et traduit intégralement avant l'exécution, ce qui permet d'effectuer des vérifications syntaxiques et sémantiques et de produire un code machine efficace. La compilation permet également de séparer le programme de la machine cible, chaque compilation produisant un code adapté à une architecture donnée, tout en conservant une expression unique de l'algorithme dans le code source. La figure 1.2 illustre ce processus de compilation. Un même code C produit deux codes assembleurs à destination de deux types de microprocesseurs, en l'occurence un processeur Intel et un processeur ARM, qui sont ensuite assemblés pour produire les codes machines correspondants.

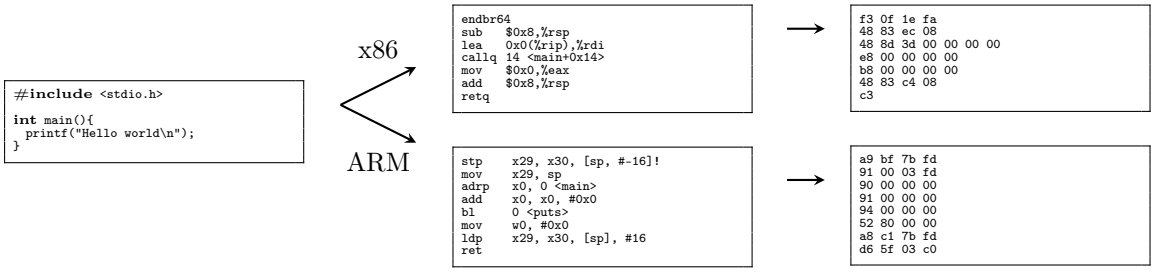


FIGURE 1.2 – Deux compilations d'un même programme pour des microprocesseurs différents

1.1.2 Du code source au programme exécutable

L'un des objectifs fondamentaux du langage C est de permettre l'écriture de programmes portables tout en produisant un code exécutable efficace. Cette efficacité repose sur un processus de traduction préalable, appelé compilation, qui transforme un programme écrit en langage C en un programme exécutable par le microprocesseur. Comprendre cette chaîne de compilation est essentiel pour maîtriser le langage, interpréter les messages du compilateur et structurer correctement des projets de taille non triviale.

Un programme écrit en C est contenu dans un ou plusieurs fichiers sources, dont l'extension est généralement `.c`. Comme nous l'avons vu, ces fichiers ne sont pas directement exécutables par la machine. Ils doivent d'abord être traduits en une représentation intermédiaire, puis combinés avec d'autres modules et bibliothèques afin de produire un exécutable. Cette transformation s'effectue en plusieurs étapes distinctes, qui peuvent être invoquées séparément ou regroupées par un compilateur moderne.

La première étape est le prétraitement. Elle concerne toutes les instructions qui commencent par le caractère `#`. Ces directives ne font pas partie du langage C à proprement parler, mais constituent un langage de macros intégré. Le préprocesseur se charge notamment d'inclure le contenu des fichiers d'en-tête, de remplacer les macros par leur définition et de supprimer certaines parties du code en fonction de conditions. À l'issue de cette étape, le compilateur obtient un fichier source C équivalent, dans lequel toutes les macros ont été résolues.

La seconde étape est la compilation proprement dite. Le compilateur analyse le code source afin de vérifier qu'il respecte la syntaxe et les règles sémantiques du langage. Il s'assure notamment que les types sont compatibles dans les expressions et les appels de fonctions. Lorsque le code est correct, il est traduit en un code de plus bas niveau, souvent sous forme de langage assembleur spécifique à l'architecture cible, puis converti en code machine. Le résultat de cette étape est un fichier objet, généralement identifié par l'extension `.o` [1].

Un fichier objet contient du code machine relogeable ainsi que des informations décrivant les symboles définis et utilisés par le module. Il ne constitue pas un programme exécutable. En particulier, il peut contenir des références à des fonctions ou des variables définies dans d'autres

1. Par défaut le code objet produit est compatible avec le microprocesseur de la machine sur laquelle à lieu la compilation. Il est tout fois possible de produire du code objet à destination d'un autre type de microprocesseur, on réalise alors une cross-compilation.

fichiers objets ou dans des bibliothèques. Ces références doivent être résolues lors d'une étape ultérieure.

L'édition de liens est l'étape qui consiste à regrouper un ou plusieurs fichiers objets et à résoudre toutes les références externes. Lors de cette étape, le code des fonctions provenant de bibliothèques peut être intégré au programme ou référencé dynamiquement. Le résultat est un fichier exécutable, directement utilisable par le système d'exploitation. Cette séparation entre compilation et édition de liens permet une compilation modulaire et favorise la réutilisation de code.

Pour illustrer ce processus, on considère un programme C minimal affichant un message à l'écran.

```
#include <stdio.h>

int main(void) {
  printf("Hello world\n");
  return 0;
}
```

Pour obtenir le programme exécutable, il faut suivre les étapes suivantes (avec le compilateur *open source* `gcc`) :

1. compiler le programme pour obtenir un fichier objet : `gcc -c helloworld.c`. Le compilateur `gcc` invoque alors successivement le préprocesseur, le compilateur, l'assembleur et produit un fichier objet `helloworld.o` ;
2. lier le(s) fichier(s) objet avec les bibliothèques nécessaires (ici une, la `libc`) pour obtenir un exécutable : `gcc -o helloworld helloworld.o -lc`. Par défaut, le fichier exécutable produit se nomme `a.out`, mais l'option `-o` permet de spécifier un nom explicite.

Il est fortement recommandé d'activer les avertissements du compilateur, car ils signalent des constructions potentiellement dangereuses ou ambiguës.

```
gcc -Wall -pedantic helloworld.c
gcc -o helloworld helloworld.o -lc
```

La modularité du langage C repose sur l'utilisation de bibliothèques. Le cœur du langage ne fournit qu'un ensemble minimal de fonctionnalités. Les fonctionnalités supplémentaires sont regroupées dans des bibliothèques, accessibles via des fichiers d'en-tête. Lors de la compilation, ces fichiers d'en-tête doivent être accessibles afin de vérifier les déclarations. Lors de l'édition de liens, le code correspondant aux fonctions utilisées doit être disponible sous forme de bibliothèques statiques ou dynamiques. Cette organisation favorise la structuration des programmes et constitue l'un des fondements du développement logiciel en C.

1.1.3 Bibliothèques statiques et dynamiques

Le langage C repose sur un principe de modularité qui sépare le code source en différents modules et permet la réutilisation de fonctionnalités communes sous forme de bibliothèques. Une bibliothèque regroupe des fichiers objets, qui peuvent être utilisés par plusieurs programmes. Cette organisation évite la duplication de code et facilite la maintenance et l'évolution des logiciels.

Sous les systèmes de type UNIX, les bibliothèques suivent des conventions de nommage précises. Le nom du fichier commence généralement par le préfixe `lib`, et l'extension du fichier indique le type de bibliothèque. On distingue principalement les bibliothèques statiques, dont l'extension est `.a`, et les bibliothèques dynamiques, dont l'extension est `.so`. Ces deux types de bibliothèques se distinguent par le moment auquel le code qu'elles contiennent est intégré ou associé au programme.

Pour les bibliothèques statiques, lors de l'édition de liens, le code des fonctions effectivement utilisées par le programme est copié depuis les bibliothèques et intégré directement dans l'exécutable final. Le lien entre l'appel d'une fonction et son implémentation est donc entièrement résolu à l'édition des liens. Chaque exécutable possède ainsi sa propre copie du code provenant

de la bibliothèque. Cette approche garantit que le programme utilise exactement la version de la bibliothèque avec laquelle il a été lié, indépendamment de l'environnement d'exécution.

L'utilisation de bibliothèques statiques présente cependant des inconvénients. Les exécutables produits sont généralement plus volumineux, puisque le code de la bibliothèque est dupliqué dans chaque programme. De plus, si plusieurs programmes utilisent la même bibliothèque statique, le code correspondant peut être chargé plusieurs fois en mémoire, ce qui entraîne une utilisation moins efficace des ressources.

Une bibliothèque dynamique repose sur un principe différent. Le code de la bibliothèque n'est pas intégré dans l'exécutable lors de l'édition de liens. À la place, l'exécutable contient des références vers une bibliothèque externe qui sera chargée en mémoire au moment du lancement du programme, si ce n'est pas déjà le cas. Le lien entre l'appel d'une fonction et son implémentation est donc réalisé à l'exécution. Plusieurs programmes peuvent ainsi partager une même bibliothèque dynamique, qui n'est chargée qu'une seule fois en mémoire.

Les bibliothèques dynamiques permettent de réduire la taille des exécutables et de mutualiser le code en mémoire. Elles facilitent également la mise à jour de bibliothèques partagées, sans nécessiter la recompilation de tous les programmes qui les utilisent. En contrepartie, elles introduisent des contraintes supplémentaires, notamment la gestion des versions et la nécessité de disposer de la bibliothèque adéquate sur le système d'exécution. L'absence ou l'incompatibilité d'une bibliothèque dynamique peut empêcher l'exécution d'un programme.

L'édition de liens avec le compilateur `gcc` s'effectue en précisant explicitement les bibliothèques à utiliser. L'option `-l` permet d'indiquer le nom logique de la bibliothèque, sans le préfixe `lib` ni l'extension. L'option `-L` permet de spécifier le répertoire dans lequel le compilateur doit rechercher les bibliothèques. Par exemple, lors de la création d'un exécutable utilisant la bibliothèque standard du C, l'éditeur de liens associe automatiquement la bibliothèque `libc`, qui fournit notamment les fonctions d'entrée et de sortie.

Dans la pratique, la bibliothèque standard du langage C est généralement liée par défaut sur les systèmes de type UNIX. Lorsqu'un programme n'utilise aucune autre bibliothèque externe, il est donc possible de compiler et de lier directement un fichier source en une seule commande. Dans l'exemple précédent, on peut obtenir l'exécutable de la façon suivante :

```
gcc -o helloworld helloworld.c
```

Il reste néanmoins essentiel de comprendre le mécanisme de l'édition de liens et la distinction entre bibliothèques statiques et dynamiques, car ces notions interviennent systématiquement dès que l'on développe des projets modulaires ou que l'on crée ses propres bibliothèques.

1.1.4 Principales bibliothèques standards

Le langage C repose sur un cœur volontairement minimal, qui ne fournit que les mécanismes fondamentaux du langage. Les fonctionnalités nécessaires à l'écriture de programmes complets sont regroupées dans un ensemble de bibliothèques standards, définies par la norme du langage. Ces bibliothèques sont disponibles sur toute implémentation conforme et constituent un socle commun garantissant la portabilité des programmes.

Une bibliothèque est constituée d'un ou plusieurs modules. Un module correspond à une unité de compilation et est formé d'un fichier source `.c`, de son fichier d'en-tête associé `.h`, et du fichier objet `.o` produit après compilation. Le fichier `.h` décrit l'interface du module, c'est-à-dire les types, constantes et fonctions accessibles aux autres parties du programme, tandis que le fichier `.c` contient l'implémentation de ces fonctionnalités.

Une bibliothèque peut regrouper plusieurs modules. Il est donc possible, et même fréquent, qu'une même bibliothèque soit associée à plusieurs fichiers d'en-tête. Chacun de ces fichiers décrit une partie spécifique de l'interface de la bibliothèque. Cette organisation favorise la modularité et permet de structurer de grandes bibliothèques en sous-ensembles cohérents.

Les principales bibliothèques standards du langage C sont présentées dans le tableau 1.2, avec leurs fichiers d'en-tête associés et les fonctionnalités qu'elles proposent.

Bibliothèque	Fichiers d'en-tête	Fonctionnalités principales
libc.so	assert.h	macro `assert` pour l'expression de préconditions en mode debug
	errno.h	codes d'erreur utilisés par les fonctions standards
	limits.h	constantes définissant les bornes des types entiers
	stdarg.h	création de fonctions à nombre variable de paramètres
	stdlib.h	conversions, nombres aléatoires, allocation dynamique de mémoire
	stdio.h	gestion des entrées et sorties standard
libm.so	complex.h	manipulation des nombres complexes
	math.h	fonctions mathématiques usuelles (trigonométrie, exponentielle, etc.)

TABLE 1.2 – Deux bibliothèques standard

L'inclusion d'un fichier d'en-tête à l'aide de la directive `#include` permet au compilateur de connaître les déclarations nécessaires à la vérification des appels de fonctions et à l'utilisation des types. Le code correspondant à ces déclarations n'est pas inclus à ce stade. Il est fourni ultérieurement, lors de l'édition de liens, par les fichiers objets ou les bibliothèques associées.

Par abus de langage, on désigne souvent une bibliothèque par son ou ses fichiers d'en-tête. Il est ainsi courant de parler de la « bibliothèque `stdio.h` », alors que `stdio.h` ne constitue que l'interface d'un ou plusieurs modules dont le code est contenu dans une bibliothèque, ici la `libc`. Cette distinction entre interface et implémentation est fondamentale pour comprendre la compilation séparée, la modularité du langage C et le rôle des bibliothèques dans la chaîne de compilation.

1.1.5 Synthèse de la chaîne de compilation

Les différentes étapes de la compilation et de l'édition de liens peuvent être résumées par une vue synthétique de la transformation d'un programme écrit en C vers un exécutable (cf. figure 1.3). Cette vue d'ensemble permet de clarifier le rôle de chaque artefact intermédiaire et de mieux comprendre les erreurs rencontrées lors du développement.

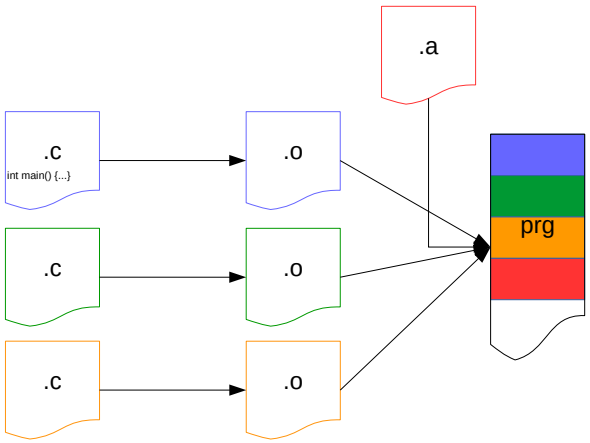


FIGURE 1.3 – Synthèse de la chaîne de compilation et d'édition de liens

Un ou plusieurs fichiers sources `.c` constituent le point de départ du processus. Chaque fichier est compilé séparément afin de produire un fichier objet `.o`, qui contient du code machine relogeable ainsi que des informations sur les symboles définis et utilisés. Ces fichiers objets peuvent ensuite être regroupés, éventuellement avec des bibliothèques statiques, lors de l'édition de liens afin de produire un programme exécutable.

Les bibliothèques statiques peuvent être intégrées directement à l'exécutable, tandis que les bibliothèques dynamiques restent externes et sont chargées au moment de l'exécution. Le programme final est ainsi le résultat de l'assemblage de plusieurs modules compilés indépendamment, ce qui permet une organisation modulaire du code et une compilation incrémentale efficace.

Cette synthèse met en évidence la séparation claire entre les différentes étapes du processus, ainsi que la distinction entre le code source, le code objet et le programme exécutable. Elle constitue un point d'ancrage essentiel pour la compréhension du langage C en tant que langage compilé et modulaire, et servira de référence pour la suite du cours.

1.2 Les bases du langage

1.2.1 Types simples

Le langage C propose un ensemble réduit de types simples, qui constituent la base de la représentation des données en mémoire. Ces types permettent de représenter des entiers, des nombres réels et des caractères, et sont directement liés aux capacités du microprocesseur. Contrairement à certains langages de plus haut niveau, le C ne fournit pas une grande variété de types prédéfinis, mais privilégie des types élémentaires combinables et proches de la machine.

Les principaux types simples du langage C sont les types entiers, les types flottants et le type caractère. Le type `int` est utilisé pour représenter des entiers signés, tandis que les types `float` et `double` permettent de représenter des nombres réels avec une précision simple ou double. Le type `char` permet de représenter un caractère, encodé sur un octet selon l'encodage ASCII ou un encodage compatible. Ces types constituent le socle à partir duquel sont construits les types plus complexes.

Le langage C permet d'affiner la représentation des types entiers à l'aide de qualificateurs. Les qualificateurs `short`, `long` et `long long` modifient la taille mémoire associée au type, et donc l'intervalle de valeurs représentables. Les qualificateurs `signed` et `unsigned` permettent de préciser si les valeurs négatives sont autorisées. La taille exacte des types dépend de l'architecture et du système d'exploitation (cf. tableau 1.3), mais la norme impose des relations d'ordre entre ces tailles. Cette dépendance au matériel impose une certaine prudence lors de l'écriture de programmes portables.

Architecture	**short int**	**int**	**long int**	**long long int**
8 bits	2	2	4	8
16 bits	2	2	4	8
32 bits	2	4	4	8
64 bits (LP64 : Linux/macOS)	2	4	8	8
64 bits (LLP64 : Windows)	2	4	4	8

TABLE 1.3 – Nombre d'octets utilisés pour représenter des int

Historiquement, le langage C ne dispose pas de type booléen natif. Les valeurs de vérité sont représentées par des entiers, la valeur zéro correspondant au faux et toute valeur non nulle correspondant au vrai. Depuis la norme C99, le fichier d'en-tête `stdbool.h` définit un type `bool` et les constantes `true` et `false`, mais cette abstraction reste fondée sur une représentation entière. Il est donc essentiel de garder à l'esprit cette convention, notamment lors de l'écriture d'expressions conditionnelles et de tests.

Le choix explicite des types simples en C a des conséquences importantes sur la précision des calculs, l'occupation mémoire et la portabilité des programmes. Comprendre ces types et leurs

variantes est une étape indispensable avant d'aborder les variables, les expressions et les structures de contrôle du langage.

1.2.2 Variables et portée

Une variable en langage C représente une zone mémoire identifiée par un identifiant, ou son nom, et associée à un type. Avant de pouvoir être utilisée, une variable doit être déclarée, ce qui permet au compilateur de réserver l'espace mémoire nécessaire et de vérifier la cohérence des opérations effectuées sur cette variable. La déclaration d'une variable consiste à préciser son type, suivi de son identifiant, et éventuellement d'une initialisation.

```
int a;
float x = 1.0;
char c;
```

L'initialisation d'une variable permet de lui donner une valeur dès sa déclaration. Lorsqu'une variable n'est pas explicitement initialisée, son contenu est indéterminé dans le cas d'une variable locale, ce qui peut conduire à des comportements imprévisibles si elle est utilisée sans précaution. Il est donc fortement recommandé d'initialiser systématiquement les variables locales avant leur première utilisation.

Le langage C distingue les variables globales et les variables locales. Une variable globale est définie en dehors de toute fonction. Elle est accessible dans le fichier source à partir de sa définition et reste présente en mémoire pendant toute la durée d'exécution du programme. Sa durée de vie correspond donc à celle du programme entier. Lorsqu'aucune valeur initiale n'est fournie, une variable globale est automatiquement initialisée à zéro.

```
int compteurGlobal = 0;

int incrementer(void) {
  compteurGlobal++;
  return compteurGlobal;
}
```

Bien que les variables globales soient simples à utiliser, leur emploi est généralement déconseillé. Elles introduisent des dépendances implicites entre différentes parties du programme, rendent le code plus difficile à comprendre et à maintenir, et compliquent le raisonnement sur le comportement des fonctions. De plus, l'utilisation excessive de variables globales nuit à la modularité et à la réutilisabilité du code. Dans la mesure du possible, il est préférable de limiter leur usage et de privilégier le passage explicite de données par paramètres de fonctions ou l'utilisation de structures adaptées.

Une variable locale est définie à l'intérieur d'une fonction ou d'un bloc. Elle n'est accessible que dans le bloc dans lequel elle est déclarée et sa durée de vie est limitée à l'exécution de ce bloc. À chaque appel de la fonction, une nouvelle instance de la variable locale est créée, généralement sur la pile d'exécution, puis détruite à la fin de l'appel.

```
int carre(int x) {
  int resultat;
  resultat = x * x;
  return resultat;
}
```

La portée d'une variable désigne la partie du programme dans laquelle son identifiant est accessible. En C, la portée est déterminée par les blocs délimités par des accolades. Une variable déclarée dans un bloc n'est visible que dans ce bloc et dans les blocs imbriqués qu'il contient. Ce mécanisme permet de restreindre volontairement la visibilité des variables et de réduire les risques de conflits d'identifiants.

Lorsque plusieurs variables portent le même identifiant dans des portées différentes, la variable déclarée dans la portée la plus interne masque les autres. Ce phénomène, appelé masquage, peut être source d'erreurs s'il est utilisé sans précaution. Il est donc recommandé d'adopter des conventions de nommage claires et d'éviter de redéclarer des identifiants identiques dans des portées imbriquées.

La distinction entre variables globales et variables locales, ainsi que la compréhension de leur portée et de leur durée de vie, est essentielle pour écrire des programmes corrects et lisibles. Ces notions interviennent directement dans la compréhension du passage de paramètres aux fonctions, de l'utilisation des pointeurs et de la gestion de la mémoire, qui seront abordées dans les sections suivantes.

1.2.3 Types énumérés

Le langage C permet de définir des types énumérés à l'aide du mot-clé `enum`. Un type énuméré définit un ensemble fini de constantes symboliques, associées implicitement ou explicitement à des valeurs entières. Il permet de représenter des ensembles de valeurs discrètes de manière plus lisible et plus sûre qu'une utilisation directe d'entiers.

La définition d'un type énuméré consiste à lister les différents identifiants correspondant aux valeurs possibles. Par défaut, les constantes sont associées à des valeurs entières croissantes, en commençant par zéro, sauf si une valeur explicite est fournie.

```
enum Semaine {
  LUNDI,
  MARDI,
  MERCREDI,
  JEUDI,
  VENDREDI,
  SAMEDI,
  DIMANCHE
};
```

Dans cet exemple, le type `enum Semaine` permet de représenter les jours de la semaine de manière explicite. Une variable de ce type ne peut prendre que l'une des valeurs définies dans l'énumération, ce qui améliore la lisibilité du code et réduit les risques d'erreurs liées à l'utilisation de valeurs arbitraires.

```
enum Semaine unJour;
unJour = LUNDI;
```

En C, un type énuméré est représenté en interne par un type entier. Il n'existe pas de mécanisme empêchant strictement l'affectation d'une valeur entière quelconque à une variable de type énuméré, mais l'utilisation systématique des constantes symboliques associées à l'énumération permet de conserver une sémantique claire et cohérente.

1.2.4 Alias de type

Le langage C permet de définir des alias de type à l'aide du mot-clé `typedef`. Un alias de type introduit un nouvel identifiant qui désigne un type existant. Il ne crée pas un nouveau type au sens strict, mais fournit un nom alternatif permettant d'améliorer la lisibilité du code et d'exprimer plus clairement l'intention du programmeur.

La définition d'un alias s'effectue à partir d'une déclaration classique (variable ou fonction), précédée du mot-clé `typedef`. L'identifiant de la variable ou de la fonction définie devient alors l'identifiant de l'alias utilisable partout où un type est attendu.

```
typedef int Entier;
Entier a;
Entier b;
```

Dans cet exemple, `Entier` est un alias du type `int`. L'utilisation de cet alias ne modifie en rien la représentation mémoire ou le comportement du programme, mais elle permet de rapprocher le code de la terminologie algorithmique ou métier utilisée dans la spécification du problème.

Les types énumérés sont souvent utilisés conjointement avec `typedef`, afin d'éviter la manipulation explicite du mot-clé `enum` lors des déclarations. Ainsi l'exemple précédent pourrait être réécrit de la manière suivante :

```
typedef enum {
  LUNDI,
  MARDI,
  MERCREDI,
  JEUDI,
  VENDREDI,
  SAMEDI,
  DIMANCHE
} Semaine;
Semaine unJour;
unJour = LUNDI;
```

Les alias de type sont particulièrement utiles pour masquer des détails d'implémentation. Ils permettent par exemple de changer ultérieurement le type sous-jacent sans modifier l'ensemble du code qui l'utilise, à condition que les opérations effectuées restent compatibles. Ils sont également largement utilisés pour simplifier des déclarations complexes, notamment lorsque l'on manipule des pointeurs, des structures ou des types fonctionnels.

L'utilisation de `typedef` contribue ainsi à rendre les programmes plus lisibles, plus maintenables et plus proches des abstractions manipulées lors de la conception algorithmique.

1.2.5 Constantes

Une constante est une valeur littérale écrite directement dans le code source. Les constantes permettent d'exprimer des valeurs fixes utilisées dans un programme, comme un seuil, une valeur initiale, un caractère particulier ou un texte affiché à l'utilisateur. Le langage C propose plusieurs catégories de constantes, correspondant aux types de base, ainsi que différents mécanismes permettant de définir des constantes symboliques.

Comme nous l'avons vu, les constantes booléennes sont représentées historiquement par des entiers, la valeur zéro correspondant au faux et toute valeur non nulle correspondant au vrai. Depuis la norme C99, le fichier d'en-tête `stdbool.h` fournit le type `bool` ainsi que les constantes `true` et `false`. Même si cette abstraction améliore la lisibilité du code, il est important de retenir que la représentation sous-jacente reste entière.

Les constantes entières peuvent être écrites selon plusieurs notations. La notation décimale est la plus courante et correspond à la base dix. Le langage C permet également d'écrire des constantes en base huit et en base seize. Une constante écrite en base huit commence par un zéro, tandis qu'une constante hexadécimale commence par `0x` ou `0X`. Ces notations sont particulièrement utiles lorsqu'on manipule des masques de bits, des adresses ou des valeurs issues de spécifications matérielles.

```
int a = 379;
int b = 0573;
int c = 0x17B;
```

Les constantes flottantes permettent de représenter des nombres réels. En C, la virgule décimale est représentée par le caractère `.` (point). Le langage permet également d'utiliser une notation scientifique, dans laquelle la lettre `e` représente une puissance de dix. Ces notations sont utilisées pour exprimer des valeurs très grandes ou très petites de manière compacte.

```
double x = 2.0;
```

```
double y = 0.3;
double z = 2.3e4;
```

Les constantes caractères sont entourées d'apostrophes. Elles représentent un caractère unique, encodé sur un octet (code ASCII étendu). Certains caractères ne peuvent pas être écrits directement, soit parce qu'ils ne sont pas imprimables, soit parce qu'ils entrent en conflit avec la syntaxe du langage. Ils doivent alors être écrits sous forme de séquences d'échappement, comme `\n` pour le retour à la ligne ou `\t` pour une tabulation. Il est également possible d'exprimer un caractère par son code numérique, notamment en notation octale.

```
char c1 = 'a';
char c2 = '\n';
char c3 = '\t';
char c4 = '\\';
```

Les constantes chaînes de caractères sont entourées de guillemets. Elles représentent une suite de caractères stockée en mémoire et terminée automatiquement par le caractère `\0`, appelé caractère nul. Ce caractère marque la fin de la chaîne et joue un rôle central dans la manière dont les chaînes sont manipulées en C. Une constante chaîne de caractères est donc, en pratique, un tableau de caractères terminé par un marqueur de fin.

```
char *s = "une chaine";
```

Il est important de noter que la représentation des caractères dépend de l'encodage du fichier source. En pratique, l'encodage ASCII ou des encodages compatibles sont souvent supposés, mais un encodage mal maîtrisé peut conduire à des comportements inattendus lors de l'affichage ou de la manipulation de caractères accentués.

Au-delà des constantes littérales, implicites, il est souvent nécessaire de définir des constantes explicites, afin d'éviter la duplication de valeurs dans le code et d'améliorer la lisibilité. Une approche historique consiste à utiliser le préprocesseur avec la directive `#define`, qui effectue un remplacement textuel avant compilation. Une approche plus robuste consiste à utiliser le mot-clé `const`, introduit dans la norme C90, qui permet de définir une constante typée.

```
#define MAX 100
```

```
const int TAILLE_MAX = 100;
```

L'utilisation de `const` présente plusieurs avantages. Elle permet au compilateur de vérifier que la constante est utilisée conformément à son type et elle exprime explicitement l'intention du programmeur. De plus, dans le cas des pointeurs passés en paramètres de fonctions, `const` permet d'indiquer qu'une fonction ne doit pas modifier la zone mémoire pointée, ce qui renforce la robustesse des interfaces.

Les constantes constituent ainsi un outil essentiel pour écrire un code lisible, maintenable et correct. Elles seront utilisées tout au long du cours, aussi bien pour les valeurs numériques que pour la gestion de chaînes de caractères et la définition de paramètres de fonctions.

1.2.6 Opérateurs

Le langage C propose un ensemble d'opérateurs permettant de construire des expressions, de combiner des valeurs et de produire des résultats. Les opérateurs du C couvrent notamment les opérations arithmétiques, les comparaisons, la logique booléenne et l'affectation. La maîtrise de ces opérateurs est indispensable pour écrire des instructions correctes et comprendre les effets produits lors de l'exécution d'un programme.

Les opérateurs relationnels permettent de comparer des valeurs. Ils produisent un résultat interprété comme une valeur de vérité, selon la convention du C. L'égalité et la différence s'écrivent respectivement `==` et `!=`, ce qui distingue clairement la comparaison de l'affectation. Les comparaisons d'ordre s'expriment avec `<`, `<=`, `>` et `>=`. Ces opérateurs sont utilisés dans les expressions conditionnelles et les tests de boucles.

```
int a = 3;
int b = 5;

if (a!= b) {
  /* ... */
}
```

Les opérateurs logiques permettent de combiner des expressions booléennes. Le ET logique s'écrit `&&`, le OU logique s'écrit `||` et la négation s'écrit `!`. Ces opérateurs sont dits courts-circuités, ce qui signifie que le second opérande n'est évalué que si cela est nécessaire pour déterminer le résultat. Ce mécanisme est très utile, notamment pour éviter des erreurs, par exemple lorsqu'un test dépend de la validité d'un pointeur.

```
if (p!= NULL && *p > 0) {
  /* ... */
}
```

Les opérateurs arithmétiques usuels sont disponibles. L'addition, la soustraction, la multiplication et la division s'écrivent `+`, `-`, `*` et `/`. L'opérateur `%` permet de calculer le reste de la division euclidienne entre deux entiers. Il est important de noter que la division entre deux entiers produit un entier, ce qui correspond à une division entière. Pour obtenir une division réelle, au moins l'un des opérandes doit être de type flottant.

```
int q = 7 / 2;
int r = 7 % 2;
double x = 7.0 / 2.0;
```

Un point fondamental du langage C est que l'affectation est elle-même une opération. L'opérateur `=` ne se contente pas d'assigner une valeur à une variable, il produit également une valeur, qui est la valeur affectée. Cette propriété permet d'écrire des expressions compactes, mais elle peut également conduire à des erreurs, notamment lorsque l'on confond `=` et `==` dans une condition.

```
int x, y;
x = y = 3;
```

Le langage C propose également les opérateurs unaires `++` et `--`, qui permettent respectivement d'incrémenter ou de décrémenter une variable. Lorsqu'ils sont utilisés isolément, `i++` et `++i` ont le même effet, car ils augmentent tous deux la valeur de `i` d'une unité. En revanche, lorsqu'ils sont utilisés dans une expression plus complexe, la position de l'opérateur par rapport à l'opérande modifie l'ordre dans lequel la valeur est produite.

```
int i = 3;
int j;

j = i++;
/* j vaut 3 et i vaut 4 */

i = 3;
j = ++i;
/* i vaut 4 et j vaut 4 */
```

Ces opérateurs introduisent des effets de bord, c'est-à-dire des modifications d'état en plus de la production d'une valeur. Dans la pratique, il est recommandé d'utiliser ces opérateurs avec prudence, en évitant de les combiner dans des expressions trop complexes. Un code plus explicite est souvent plus facile à lire et à maintenir, tout en réduisant les risques d'erreurs.

1.2.7 Instructions

Un programme en langage C est constitué d'une suite d'instructions. Une instruction représente une action élémentaire exécutée par la machine, comme une affectation, un appel de fonction

ou une instruction de contrôle. Dans la syntaxe du langage, une instruction simple se termine systématiquement par un point-virgule, qui marque la fin de l'instruction. Ce point-virgule fait partie intégrante de la grammaire du langage et son oubli est une source fréquente d'erreurs de compilation.

```
a = a + 1;
```

Le langage C permet également de regrouper plusieurs instructions en une seule instruction composée. Une instruction composée est un bloc d'instructions délimité par des accolades. Elle est considérée syntaxiquement comme une instruction unique, ce qui permet de l'utiliser partout où une seule instruction est attendue, par exemple après un `if` ou dans le corps d'une boucle. L'utilisation de blocs est essentielle pour structurer un programme et pour contrôler précisément la portée des variables locales.

```
{
  a = a + 1;
  b = b + 2;
}
```

Un bloc peut contenir des déclarations de variables locales. Ces variables sont alors accessibles uniquement dans le bloc où elles sont déclarées, et leur durée de vie est limitée à l'exécution de ce bloc. Cette propriété permet de limiter volontairement la visibilité de certaines variables, d'éviter des conflits d'identifiants et de clarifier l'organisation du code.

La distinction entre instructions simples et instructions composées est particulièrement importante dans les structures de contrôle. En C, lorsqu'une structure comme `if` ou `while` est suivie d'une instruction, celle-ci peut être soit une instruction simple, soit un bloc. Lorsque l'on souhaite exécuter plusieurs instructions dans ce contexte, il est nécessaire d'utiliser un bloc. L'omission des accolades peut conduire à des erreurs logiques difficiles à détecter, car le compilateur ne signale pas nécessairement ce type de problème.

```
if (a < 10)
  a = a + 1;
  b = b + 2;
```

Dans cet exemple, seule la première affectation est contrôlée par la condition, tandis que la seconde est exécutée systématiquement. Pour exprimer l'intention correcte, il est nécessaire d'utiliser un bloc.

```
if (a < 10) {
  a = a + 1;
  b = b + 2;
}
```

1.2.8 Instructions conditionnelles

Les instructions conditionnelles permettent de modifier le flot d'exécution d'un programme en fonction d'une condition. En langage C, une condition est une expression évaluée selon la convention du langage, où la valeur zéro correspond au faux et toute valeur non nulle correspond au vrai. Les structures conditionnelles sont essentielles pour exprimer des alternatives, tester des cas particuliers et contrôler la logique d'un algorithme.

L'instruction conditionnelle la plus courante est `if`. Elle permet d'exécuter une instruction ou un bloc si une condition est vraie, et éventuellement d'exécuter une autre instruction ou un autre bloc dans le cas contraire, via le mot-clé `else`. La syntaxe générale est simple, mais l'utilisation de blocs est fortement recommandée dès que le corps contient plusieurs instructions.

```
if (a < 10)
  a = a + 1;
else
```

```
  a = a + 2;
```

Une particularité importante du langage C est que le `else` est associé au `if` le plus proche qui ne possède pas déjà de `else`. Ce comportement, conforme à la grammaire du langage, peut introduire des ambiguïtés lorsque des `if` sont imbriqués sans accolades. Même si le compilateur accepte ce code, il peut être difficile à lire et mener à des erreurs logiques.

```
if (a > b)
  if (c < d)
    u = v;
  else
    i = j;
```

Dans ce cas, le `else` est associé au second `if`. Si on veut associer le `else` au premier `if`, il est nécessaire d'utiliser des blocs :

```
if (a > b) {
  if (c < d)
    u = v;
} else {
  i = j;
}
```

Lorsque l'on souhaite choisir entre plusieurs cas en fonction de la valeur d'une expression entière ou d'un caractère, le langage C propose l'instruction `switch`. Elle permet de sélectionner un bloc d'instructions correspondant à une valeur donnée, via les étiquettes `case`. Le mot-clé `default` permet de définir un comportement par défaut lorsque aucun cas ne correspond. L'instruction `break` est généralement nécessaire pour éviter l'enchaînement involontaire des cas, car en l'absence de `break`, l'exécution se poursuit dans le cas suivant.

```
switch (choix) {
  case 't':
    printf("vous voulez un triangle\n");
    break;
  case 'c':
    printf("vous voulez un carre\n");
    break;
  case 'r':
    printf("vous voulez un rectangle\n");
    break;
  default:
    printf("erreur. recommencez!\n");
}
```

1.2.9 Instructions itératives

Les instructions itératives permettent de répéter l'exécution d'une instruction ou d'un bloc tant qu'une condition est satisfaite. Elles sont indispensables pour traiter des collections de données, réaliser des calculs successifs ou parcourir des structures. Le langage C propose trois formes principales de boucles : `for`, `while` et `do ... while`. Ces trois constructions expriment des intentions proches, mais chacune est adaptée à des situations spécifiques.

La boucle `for` est souvent utilisée lorsque le nombre d'itérations est connu ou lorsque l'itération suit une progression simple. Sa syntaxe regroupe, dans une même structure, une initialisation, une condition d'arrêt et une opération effectuée à chaque itération. Contrairement à l'écriture algorithmique classique, cette construction est très flexible et peut être utilisée de manière détournée, ce qui impose une certaine discipline pour conserver un code lisible.

```
int i;
for (i = 0; i < 10; i++)
  printf("%d\n", i);
```

La boucle `while` est adaptée aux situations où l'on souhaite répéter un traitement tant qu'une condition reste vraie. La condition est testée avant chaque itération, ce qui signifie que le corps de la boucle peut ne jamais être exécuté si la condition est fausse dès le départ. Cette boucle est souvent utilisée pour exprimer des algorithmes dont le nombre d'itérations dépend d'un calcul ou d'une convergence.

```
float g, d, m;
g = 1;
d = x;
while (d - g > epsilon) {
  m = (d + g) / 2.0;
  if (m * m > x)
    d = m;
  else
    g = m;
}
```

La boucle `do ... while` permet également de répéter un traitement tant qu'une condition est vraie, mais avec une différence essentielle : la condition est testée après l'exécution du corps de boucle. Le corps est donc exécuté au moins une fois. Cette construction est particulièrement utile lorsque l'on doit effectuer une première lecture ou une première action avant de pouvoir tester une condition de répétition.

```
char temp[MAX];
int motDePasseValide = 0;
int nbEssais = 0;

do {
  printf("Mot de passe: ");
  scanf("%s", temp);
  motDePasseValide = hash(temp) == hashMotPassseRef;
  nbEssais++;
} while (!motDePasseValide && nbEssais < nbEssaisMax);
```

Il est important de noter que, dans certains algorithmes exprimés en pseudo-code, la boucle est décrite sous la forme répéter ... jusqu'à ce que. En langage C, la condition du `do ... while` correspond alors à la négation de la condition d'arrêt exprimée en algorithmique. Cette inversion est une source fréquente d'erreurs, en particulier lors de la traduction d'un algorithme vers du code C.

1.3 Pointeurs et fonctions

1.3.1 Mémoire, adresses et représentation

Les pointeurs constituent l'un des mécanismes centraux du langage C. Ils permettent de manipuler directement des adresses mémoire, c'est-à-dire de désigner l'emplacement où sont stockées les données. Cette capacité est à la fois une force et une source fréquente d'erreurs. Elle explique en grande partie pourquoi le C est considéré comme un langage proche de la machine, et pourquoi la compréhension de la mémoire est indispensable pour programmer correctement.

Lorsqu'un programme s'exécute, il manipule des données qui sont stockées en mémoire vive. Chaque octet de cette mémoire possède une adresse, qui est un entier permettant d'identifier de

manière unique l'emplacement correspondant. Ainsi, lorsqu'une variable est déclarée, le compilateur réserve une zone mémoire suffisamment grande pour contenir une valeur du type associé. Cette zone mémoire est repérée par une adresse, que l'on peut interpréter comme la position du premier octet de la variable.

Le langage C expose explicitement cette notion d'adresse. Une variable ne se réduit pas à sa valeur : elle possède également une localisation en mémoire. Cette localisation devient essentielle dès que l'on souhaite modifier une donnée depuis une autre fonction, partager une zone mémoire entre plusieurs parties du programme, ou manipuler des structures de données dynamiques comme des listes chaînées, des arbres ou des tableaux alloués dynamiquement.

Un pointeur est une variable dont la valeur est une adresse. Autrement dit, un pointeur contient l'adresse d'une donnée. Le type d'un pointeur n'indique pas seulement qu'il s'agit d'une adresse, il indique également le type de la donnée pointée. Cette information est essentielle car elle permet au compilateur de connaître la taille des objets manipulés et de vérifier, partiellement, la cohérence des opérations effectuées.

La représentation mémoire des données dépend de leur type. Un `char` occupe un octet, tandis qu'un `int` occupe plusieurs octets (cf. tableau 1.3). Un pointeur occupe également un certain nombre d'octets, qui dépend de l'architecture. Sur une architecture 64 bits, un pointeur occupe typiquement 8 octets, tandis que sur une architecture 32 bits, il occupe typiquement 4 octets. La taille d'un pointeur ne dépend pas du type pointé, mais uniquement de l'architecture.

Cette uniformité de représentation des adresses est un élément important pour comprendre le langage C. Quel que soit l'objet pointé, un pointeur reste une valeur interprétée comme une adresse. En revanche, le type associé au pointeur influence la manière dont le compilateur interprète cette adresse lorsqu'il faut accéder à la donnée pointée. Cette distinction entre l'adresse elle-même et le type de la donnée pointée est une notion clé, qui sera utilisée dans toute la suite, notamment pour le passage de paramètres aux fonctions et pour la manipulation de tableaux.

Déréférencer un pointeur signifie accéder à la donnée située à l'adresse contenue dans le pointeur. Un déréférencement peut être en lecture ou en écriture, selon que l'on souhaite lire la valeur pointée ou la modifier.

Enfin, il est important de rappeler qu'un pointeur peut contenir une adresse invalide. Le langage C ne vérifie pas automatiquement que l'adresse contenue dans un pointeur correspond à une zone mémoire autorisée. Déréférencer un pointeur invalide conduit à un comportement indéfini, pouvant se manifester par un crash, une corruption de données ou un résultat incohérent. La programmation en C nécessite donc une discipline particulière, notamment dans l'initialisation des pointeurs et la gestion de la durée de vie des objets pointés.

1.3.2 Déclaration et initialisation d'un pointeur

Un pointeur est une variable dont la valeur est une adresse mémoire. Comme toute variable en C, un pointeur doit être déclaré avant d'être utilisé. La déclaration d'un pointeur consiste à préciser le type de la donnée pointée, suivi du symbole `*`, puis de l'identifiant du pointeur. Le symbole `*` fait partie du type et indique que la variable ne contient pas directement une valeur de ce type, mais l'adresse d'une valeur de ce type.

```
int *p;
double *q;
char *s;
```

Dans ces exemples, `p` est un pointeur vers un entier, `q` est un pointeur vers un flottant en double précision et `s` est un pointeur vers un caractère. Le type du pointeur est essentiel car il permet au compilateur de connaître la taille des objets pointés et d'interpréter correctement les accès mémoire.

Comme pour toute variable locale, un pointeur déclaré sans initialisation contient une valeur indéterminée. Dans le cas d'un pointeur, cela signifie qu'il contient une adresse arbitraire, qui ne correspond pas nécessairement à une zone mémoire valide. L'utilisation d'un tel pointeur, en

particulier son déréférencement, conduit à un comportement indéfini. Il est donc indispensable d'initialiser systématiquement un pointeur avant sa première utilisation.

L'initialisation la plus simple consiste à faire pointer un pointeur vers une variable existante, dont la durée de vie est connue. Dans ce cas, le pointeur reçoit l'adresse de la variable, obtenue grâce à l'opérateur `&`. Le pointeur et la variable désignent alors la même zone mémoire.

```
int a = 10;
int *p = &a;
```

Une autre forme d'initialisation courante consiste à initialiser un pointeur à `NULL`. La constante `NULL` représente une adresse invalide particulière, utilisée par convention pour indiquer qu'un pointeur ne pointe sur aucune donnée. Initialiser un pointeur à `NULL` est une bonne pratique, car cela permet de tester explicitement si le pointeur est utilisable avant de l'employer.

```
int *p = NULL;
```

Il est important de noter que `NULL` n'est pas une adresse mémoire réelle utilisable. Déréférencer un pointeur nul conduit à un comportement indéfini. L'intérêt de `NULL` est précisément de fournir une valeur reconnaissable, que l'on peut tester dans des conditions et utiliser comme marqueur d'absence de donnée.

```
if (p!= NULL) {
  /* utilisation possible de p */
}
```

La déclaration d'un pointeur doit également être comprise correctement lorsque plusieurs variables sont déclarées sur une même ligne. En C, le symbole `*` est associé à l'identifiant et non au type global de la ligne. Ainsi, dans une déclaration multiple, il est possible qu'une variable soit un pointeur et une autre non, ce qui peut être source de confusion. Pour éviter ce type d'erreur, il est recommandé d'écrire une déclaration par ligne lorsque l'on manipule des pointeurs.

```
int* p1, p2;
```

Dans cet exemple, `p1` est un pointeur vers `int`, mais `p2` est un entier. Cette syntaxe est légale, mais elle est trompeuse. Une écriture plus claire consiste à séparer les déclarations.

```
int *p1;
int p2;
```

La déclaration et l'initialisation correcte des pointeurs constituent une étape fondamentale. Elles conditionnent la validité des accès mémoire et la robustesse des programmes. Dans la suite, on introduira les opérateurs permettant de manipuler explicitement les adresses et d'accéder à la valeur pointée.

1.3.3 Les opérateurs `&` et `*`

Les pointeurs sont manipulés en langage C à l'aide de deux opérateurs fondamentaux. L'opérateur `&` permet d'obtenir l'adresse d'une variable, tandis que l'opérateur `*` permet de déréférencer un pointeur, d'accéder à la valeur stockée à l'adresse contenue dans un pointeur. Ces deux opérateurs sont complémentaires et constituent le mécanisme de base permettant de passer de la notion de variable à la notion d'adresse, puis de revenir à la valeur.

L'opérateur `&` est appelé opérateur d'adressage. Appliqué à une variable, il produit l'adresse de la zone mémoire dans laquelle cette variable est stockée. Le résultat de `&` est donc une valeur de type pointeur, dont le type dépend du type de la variable. Ainsi, si `a` est un entier, alors `&a` est de type `int*`.

```
int a = 42;
int *p = &a;
```

L'opérateur `*` est appelé opérateur de déréférencement. Appliqué à un pointeur, il permet d'accéder à la valeur stockée à l'adresse contenue dans ce pointeur. Le résultat de `*p` est une valeur du type pointé par `p`. Dans l'exemple précédent, `p` est un pointeur vers `int`, donc `*p` est un entier.

```
int a = 42;
int *p = &a;
int b = *p; /* b vaut 42 */
```

Le déréférencement ne sert pas uniquement à lire une valeur. Il peut également apparaître à gauche d'une affectation, ce qui permet de modifier la valeur stockée en mémoire à l'adresse pointée. Dans ce cas, la modification ne concerne pas le pointeur lui-même, mais bien la donnée pointée.

```
int a = 42;
int *p = &a;
*p = 100; /* a vaut maintenant 100 */
```

L'opérateur `*` possède deux rôles distincts en langage C, selon le contexte. Dans une déclaration, `*` indique qu'une variable est un pointeur. Dans une expression, `*` correspond au déréférencement. Il s'agit du même symbole, mais son sens dépend du contexte syntaxique. Cette particularité est une source fréquente de confusion pour les débutants, et il est important de distinguer clairement ces deux usages.

L'utilisation de ces opérateurs nécessite cependant une condition essentielle : le pointeur doit contenir une adresse valide. Déréférencer un pointeur non initialisé, un pointeur nul ou un pointeur dont la zone mémoire pointée n'existe plus conduit à un comportement indéfini. Ce point est crucial en C, car le langage ne fournit pas de mécanisme automatique empêchant ces erreurs. Dans la suite, on verra comment les pointeurs interviennent dans la manipulation de tableaux, dans la représentation des chaînes de caractères et dans le passage de paramètres aux fonctions.

1.3.4 Fonctions en langage C

Les fonctions permettent de structurer un programme en décomposant un problème en sous-problèmes. Elles rendent le code plus lisible, facilitent sa maintenance et favorisent la réutilisation de traitements. En langage C, une fonction est définie par un type de retour, un identifiant, une liste de paramètres et un corps, qui contient les instructions exécutées lors de l'appel.

La syntaxe générale d'une définition de fonction peut être illustrée par l'exemple suivant.

```
int somme(int a, int b) {
  return a + b;
}
```

Dans cet exemple, `somme` est l'identifiant de la fonction, `a` et `b` sont ses paramètres, et `int` est son type de retour. Lorsqu'elle est appelée, la fonction calcule une valeur entière et la renvoie au code appelant. Le mot-clé `return` permet de terminer l'exécution de la fonction et de transmettre une valeur au point d'appel. Dans une fonction dont le type de retour n'est pas `void`, il est indispensable que le programme exécute un `return` fournissant une valeur compatible avec le type annoncé.

Le langage C permet également de définir des fonctions qui ne renvoient aucune valeur. Dans ce cas, le type de retour est `void`. Une fonction `void` peut être utilisée pour effectuer une action, comme afficher un message, modifier une structure de données ou mettre à jour des variables accessibles par pointeurs.

```
void afficherBonjour(void) {
  printf("Bonjour\n");
}
```

Dans cet exemple, la fonction ne renvoie aucune valeur. Le paramètre `void` dans la liste des paramètres indique explicitement que la fonction ne prend aucun argument. Cette écriture est recommandée, car elle rend l'interface de la fonction non ambiguë. Dans un code C moderne, il est préférable d'écrire `f(void)` plutôt que `f()` lorsque la fonction ne prend aucun paramètre.

Une fonction peut comporter plusieurs instructions `return`. L'exécution de la fonction s'arrête dès que l'un de ces `return` est atteint. Cette possibilité est utile pour gérer des cas particuliers, mais elle doit être utilisée avec modération afin de conserver une structure de code lisible.

Enfin, il est important de distinguer la déclaration et la définition d'une fonction. Une déclaration, appelée prototype, décrit l'interface de la fonction, c'est-à-dire son type de retour, son identifiant et les types de ses paramètres. La définition fournit l'implémentation complète. Les prototypes sont généralement placés dans des fichiers d'en-tête `.h`, tandis que les définitions sont placées dans des fichiers sources `.c`. Cette organisation permet la compilation séparée et la modularité des programmes, ce qui constitue un principe central du développement en langage C.

1.3.5 Passage de paramètres aux fonctions

Lors de l'appel d'une fonction, les valeurs fournies par le code appelant sont transmises aux paramètres formels définis dans l'interface de la fonction. Ces paramètres jouent le rôle de variables locales initialisées au début de l'exécution de la fonction. Ils permettent de fournir à la fonction les données nécessaires à son calcul, et de spécialiser son comportement en fonction des arguments passés.

En langage C, les paramètres sont toujours transmis par valeur. Cela signifie que la fonction reçoit une copie des valeurs fournies lors de l'appel. Cette règle s'applique aussi bien aux types simples, comme les entiers ou les réels, qu'aux pointeurs. Ainsi, lorsqu'un argument est transmis, la fonction ne reçoit pas directement la variable de l'appelant, mais une copie de son contenu.

```
int max2(int a, int b) {
  if (a > b)
    return a;
  return b;
}
```

Dans cet exemple, `a` et `b` sont des paramètres formels de type `int`. À l'intérieur de la fonction, ils peuvent être utilisés comme des variables locales. Toute modification de `a` ou `b` n'a aucun effet sur les variables passées lors de l'appel, car seule une copie a été transmise.

Le passage par valeur a une conséquence importante. Une fonction ne peut pas modifier directement une variable locale appartenant à une autre fonction. Si l'on souhaite qu'une fonction puisse modifier une donnée définie dans la fonction appelante, il est nécessaire de transmettre l'adresse de cette donnée. Cette adresse est stockée dans un pointeur, et la fonction peut alors accéder à la zone mémoire correspondante en déréférençant ce pointeur.

```
void incrementer(int *p) {
  *p = *p + 1;
}
```

Dans cet exemple, la fonction `incrementer` reçoit un pointeur vers `int`. Lors de l'appel, l'argument transmis est l'adresse d'une variable entière, obtenue avec l'opérateur `&`. Le pointeur est transmis par valeur, mais il désigne la zone mémoire originale. Le déréférencement permet donc de modifier directement la valeur de la variable de l'appelant.

```
int main(void) {
  int a = 10;
  incrementer(&a);
  /* a vaut maintenant 11 */
  return 0;
}
```

Ce mécanisme constitue le fondement du passage par adresse en C. Il est omniprésent dans les interfaces des bibliothèques standards, notamment pour les fonctions d'entrée et sortie, ainsi que pour de nombreuses fonctions qui produisent plusieurs résultats ou qui modifient des structures de données.

1.4 Entrée et sortie standard

Les entrées et sorties standard constituent le mécanisme le plus simple pour interagir avec l'utilisateur en langage C. Elles permettent d'afficher par défaut des informations à l'écran et de lire des données saisies au clavier. Ces opérations sont réalisées à l'aide de fonctions de la bibliothèque standard, principalement définies dans le fichier d'en-tête `stdio.h`. Dans ce cours, on se limite dans un premier temps aux entrées et sorties en mode texte, qui suffisent pour écrire de nombreux programmes et illustrer les concepts essentiels.

La notion d'entrée et sortie standard repose sur trois flux prédéfinis. Le flux de sortie standard correspond aux affichages normaux du programme, le flux d'entrée standard correspond aux données lues au clavier ou redirigées depuis un fichier, et le flux d'erreur standard est utilisé pour afficher des messages d'erreur. Ces flux peuvent être redirigés par le système d'exploitation, ce qui permet de chaîner des programmes ou de sauvegarder des sorties dans des fichiers sans modifier le code source.

Dans la suite, on présente les deux fonctions les plus courantes pour interagir avec ces flux. La fonction `printf` permet d'écrire sur la sortie standard, tandis que la fonction `scanf` permet de lire depuis l'entrée standard. Ces fonctions reposent sur un mécanisme de format, qui décrit comment interpréter les données à afficher ou à lire. La compréhension de ce mécanisme est essentielle, car une erreur de format peut provoquer des résultats incorrects, voire des comportements indéfinis.

1.4.1 Sortie standard : `printf`

La fonction `printf` permet d'écrire sur la sortie standard, généralement l'écran. Elle est définie dans le fichier d'en-tête `stdio.h` et constitue l'outil le plus courant pour produire un affichage en mode texte. La fonction repose sur une chaîne de format, qui décrit à la fois le texte à afficher et la manière dont les valeurs des arguments doivent être converties en une représentation lisible.

```
#include <stdio.h>

int main(void) {
  int a = 12;
  printf("a vaut %d\n", a);
  return 0;
}
```

La chaîne de format peut contenir du texte ordinaire, ainsi que des spécificateurs introduits par le caractère `%`. Chaque spécificateur indique le type attendu pour l'argument correspondant et la manière dont il doit être affiché. Les arguments supplémentaires sont fournis après la chaîne de format et doivent apparaître dans le même ordre que les spécificateurs. Si le nombre d'arguments ou leurs types ne correspondent pas à la chaîne de format, le programme peut produire un résultat incohérent ou provoquer un comportement indéfini.

Les spécificateurs les plus couramment utilisés sont présentés dans le tableau 1.4. Ils suffisent pour la majorité des programmes de ce cours et seront réutilisés dans les exemples et les exercices.

Le spécificateur `%p` permet d'afficher une adresse mémoire. Il est particulièrement utile lors du débogage, notamment lorsque l'on manipule des pointeurs. Le type attendu est `void*`, ce qui conduit souvent à effectuer un transtypage explicite afin de respecter le format.

```
int a = 42;
printf("adresse de a = %p\n", (void*)&a);
```

Spécificateur	Type attendu	Remarque
`%d`	`int`	entier signé en base dix
`%u`	`unsigned int`	entier non signé en base dix
`%ld`	`long int`	entier signé de type `long`
`%lu`	`unsigned long int`	entier non signé de type `unsigned long`
`%lld`	`long long int`	entier signé de type `long long`
`%llu`	`unsigned long long int`	entier non signé de type `unsigned long long`
`%x`	`unsigned int`	entier en hexadécimal (lettres minuscules)
`%X`	`unsigned int`	entier en hexadécimal (lettres majuscules)
`%f`	`double`	réel, `float` promu en `double`
`%e`	`double`	réel en notation scientifique
`%c`	`int`	caractère, transmis sous forme d'entier
`%s`	`char*`	chaîne de caractères terminée par `\0`
`%p`	`void*`	adresse mémoire, utile pour les pointeurs
`%%`	aucun	affiche le caractère `%`

TABLE 1.4 – Les spécificateurs les plus utilisés

Il est important de noter que, dans le cas de `printf`, certains types sont automatiquement promus lors du passage des arguments. En particulier, un `float` est automatiquement converti en `double`. C'est pourquoi le spécificateur `%f` attend un `double`, même si l'on souhaite afficher une variable de type `float`. Cette règle fait partie des conversions automatiques appliquées lors de l'appel de fonctions à nombre variable d'arguments.

La fonction `printf` renvoie un entier correspondant au nombre de caractères effectivement affichés. Cette valeur peut être utilisée pour détecter certaines erreurs d'écriture, mais elle est rarement exploitée dans les programmes simples. Dans ce cours, `printf` sera principalement utilisée pour produire des affichages destinés à l'utilisateur ou pour faciliter l'observation et la compréhension du comportement d'un programme.

1.4.2 Entrée standard : `scanf`

La fonction `scanf` permet de lire des données depuis l'entrée standard, typiquement saisies au clavier. Elle est définie dans `stdio.h` et repose également sur une chaîne de format. Contrairement à `printf`, qui reçoit des valeurs à afficher, `scanf` doit écrire les valeurs lues dans des variables. Pour cette raison, les arguments passés à `scanf` sont généralement des adresses de variables.

```
#include <stdio.h>

int main(void) {
  int a;
  scanf("%d", &a);
  printf("a = %d\n", a);
  return 0;
}
```

L'utilisation de `scanf` met en évidence un point essentiel du langage C. Pour modifier une variable définie dans la fonction appelante, il est nécessaire de transmettre son adresse. Ainsi, pour lire un entier, il faut fournir l'adresse d'une variable de type `int`, obtenue avec l'opérateur `&`. Oublier cet opérateur conduit à une erreur grave, car `scanf` interprète alors la valeur fournie comme une adresse, ce qui provoque un comportement indéfini.

Comme pour `printf`, la chaîne de format doit correspondre exactement aux types des variables associées. Par exemple, `%d` est utilisé pour un entier, `%f` pour un `float` et `%lf` pour un `double`. Une incohérence entre le format et le type de la variable peut produire des résultats incorrects ou corrompre la mémoire.

La fonction `scanf` renvoie le nombre de valeurs effectivement lues et affectées. Cette valeur permet de vérifier que la lecture a réussi. Dans un programme robuste, il est recommandé de tester cette valeur afin de détecter les erreurs de saisie. Dans la suite du cours, on insistera sur l'importance de ces tests, car une entrée utilisateur incorrecte est une source fréquente de bugs.

Enfin, l'utilisation de `scanf` pour lire des chaînes de caractères doit être réalisée avec prudence, car une lecture non bornée peut provoquer un dépassement de tampon. Ce point sera détaillé dans la partie consacrée aux chaînes de caractères, une fois les tableaux introduits.

1.4.3 Fichiers

Les entrées et sorties standard permettent d'interagir simplement avec l'utilisateur, mais un programme doit souvent lire ou écrire des données de manière persistante. Le langage C fournit pour cela un mécanisme de manipulation de fichiers, fondé sur la bibliothèque standard `stdio.h`. Ce mécanisme repose sur la notion de flux, qui représente une source ou une destination de données. Un fichier est ainsi vu comme un flux associé à un support externe, généralement le disque.

En langage C, un flux est manipulé à travers un pointeur de type `FILE*`. Ce pointeur désigne une structure interne gérée par la bibliothèque standard, qui contient notamment des informations sur l'état du flux, la position courante dans le fichier, ainsi que des buffers utilisés pour optimiser les accès. Le programmeur ne manipule pas directement cette structure, mais utilise des fonctions standards pour ouvrir, lire, écrire et fermer les fichiers.

Le langage C définit trois flux standards, disponibles dans `stdio.h`. Le flux `stdin` correspond à l'entrée standard, généralement le clavier, le flux `stdout` correspond à la sortie standard, généralement l'écran, et le flux `stderr` correspond à la sortie d'erreur standard. Le flux `stderr` est destiné aux messages d'erreur ou de diagnostic. Cette séparation est particulièrement utile, car le système d'exploitation peut rediriger `stdout` et `stderr` séparément. Un programme peut ainsi produire un résultat normal redirigeable dans un fichier, tout en conservant les erreurs affichées à l'écran.

Les fonctions `printf` et `scanf` sont des cas particuliers de fonctions plus générales. Conceptuellement, `printf` correspond à un appel à `fprintf` sur le flux `stdout`, et `scanf` correspond à un appel à `fscanf` sur le flux `stdin`. Cette remarque est essentielle, car elle montre que les entrées et sorties standard et les entrées et sorties sur fichiers reposent sur le même mécanisme de flux.

```
printf("x = %d\n", x);
/* equivalent conceptuellement */
fprintf(stdout, "x = %d\n", x);

scanf("%d", &x);
/* equivalent conceptuellement */
fscanf(stdin, "%d", &x);
```

L'ouverture d'un fichier se fait avec la fonction `fopen`. Cette fonction prend deux arguments, le nom du fichier et un mode d'ouverture. Le mode `"r"` ouvre un fichier en lecture, le mode `"w"` ouvre un fichier en écriture en le créant ou en l'écrasant, et le mode `"a"` ouvre un fichier en ajout, c'est-à-dire en écrivant à la fin. Si l'ouverture échoue, `fopen` renvoie `NULL`. Ce test est indispensable pour éviter un comportement indéfini et pour produire un message d'erreur exploitable.

```
#include <stdio.h>

int main(void) {
  FILE *f;

  f = fopen("donnees.txt", "r");
  if (f == NULL) {
    fprintf(stderr, "erreur: impossible d'ouvrir donnees.txt\n");
    return 1;
```

```
  }

  fclose(f);
  return 0;
}
```

Une fois le fichier ouvert, il est possible d'écrire des données dans ce fichier avec `fprintf`. Cette fonction fonctionne comme `printf`, mais elle écrit dans un flux donné au lieu d'écrire sur `stdout`. Elle utilise la même notion de chaîne de format et les mêmes spécificateurs. De manière analogue, la lecture de données dans un fichier texte peut être réalisée avec `fscanf`, qui fonctionne comme `scanf`, mais lit dans un flux explicite. Comme `scanf`, `fscanf` attend des adresses de variables, car la fonction doit écrire les valeurs lues dans ces variables.

```
#include <stdio.h>

int main(void) {
  FILE *f;
  int x;

  f = fopen("donnees.txt", "r");
  if (f == NULL) {
    fprintf(stderr, "erreur: fichier introuvable\n");
    return 1;
  }

  if (fscanf(f, "%d", &x) == 1)
    printf("x = %d\n", x);

  fclose(f);
  return 0;
}
```

Les fonctions de manipulation de fichiers sont nombreuses, mais un petit ensemble suffit dans la majorité des cas. Le tableau 1.5 présente les fonctions les plus utilisées dans ce cours, ainsi que leur rôle principal.

Lorsqu'un programme a terminé d'utiliser un fichier, il est indispensable de le fermer avec `fclose`. Cette opération libère les ressources associées au flux et garantit que les données en attente dans les buffers sont effectivement écrites sur le disque. Oublier de fermer un fichier peut conduire à une perte de données ou à une consommation inutile de ressources.

Le modèle des flux en C unifie ainsi l'entrée standard, la sortie standard, la sortie d'erreur et les fichiers. Les fonctions `printf` et `scanf` utilisent implicitement `stdout` et `stdin`, tandis que `fprintf` et `fscanf` permettent de choisir explicitement le flux utilisé. Cette cohérence facilite l'apprentissage et permet de passer progressivement de programmes interactifs à des programmes capables de traiter des fichiers de données.

1.5 Types composites

Les types simples du langage C permettent de représenter des valeurs élémentaires, comme des entiers, des réels ou des caractères. Cependant, la majorité des programmes manipulent des ensembles de données structurées, composées de plusieurs valeurs. Le langage C fournit pour cela des types composites, qui permettent de regrouper plusieurs éléments en mémoire et de construire des structures de données plus riches.

Dans cette section, on introduit quatre familles essentielles de types composites. Les tableaux permettent de représenter une séquence contiguë d'éléments de même type. Les chaînes de caractères reposent sur des tableaux de caractères terminés par un caractère nul et constituent une

Fonction	Prototype simplifié	Rôle principal
`fopen`	`FILE* fopen(const char*, const char*)`	ouvre un fichier et renvoie un flux
`fclose`	`int fclose(FILE*)`	ferme un flux et libère les ressources associées
`fprintf`	`int fprintf(FILE*, const char*, ...)`	écrit des données formatées dans un flux
`fscanf`	`int fscanf(FILE*, const char*, ...)`	lit des données formatées depuis un flux
`fgetc`	`int fgetc(FILE*)`	lit un caractère, renvoie `EOF` en fin de fichier
`fputc`	`int fputc(int, FILE*)`	écrit un caractère dans un flux
`fgets`	`char* fgets(char*, int, FILE*)`	lit une ligne ou un bloc de caractères
`fputs`	`int fputs(const char*, FILE*)`	écrit une chaîne dans un flux
`feof`	`int feof(FILE*)`	teste si la fin de fichier a été atteinte
`ferror`	`int ferror(FILE*)`	teste si une erreur de lecture ou écriture est survenue

TABLE 1.5 – Principales fonctions d'utilisation des flux

convention fondamentale du langage. Les structures permettent de regrouper plusieurs champs de types potentiellement différents au sein d'un même objet. Enfin, les unions permettent de partager une même zone mémoire entre plusieurs représentations possibles, ce qui est utile dans certains contextes bas niveau.

Ces types composites sont directement liés à la représentation mémoire des données. Ils permettent d'exprimer des structures de données proches de celles utilisées en algorithmique, tout en conservant les caractéristiques du langage C, notamment sa proximité avec la machine et sa gestion explicite de la mémoire.

1.5.1 Tableaux

En langage C, un tableau est une structure de données permettant de regrouper plusieurs valeurs de même type sous un même identificateur. Les éléments sont stockés de manière contiguë en mémoire, ce qui permet un accès direct à n'importe quel élément à partir de son indice.

On déclare une variable de type tableau en ajoutant, après l'identificateur, la taille de chaque dimension entre crochets. Par exemple, un tableau d'entiers de taille 10 se déclare ainsi :

```
int tab1[10];
```

Le C autorise également les tableaux multidimensionnels. Un tableau de réels à deux dimensions peut par exemple s'écrire :

```
float tab2[2][5];
```

Le premier indice d'une dimension est toujours 0. Ainsi, pour un tableau de taille n, les indices valides vont de 0 à $n-1$. Dans l'exemple `tab2[2][5]`, la première dimension accepte les indices 0 et 1, et la seconde les indices 0 à 4.

Le langage C ne donne pas de sémantique particulière aux dimensions. C'est au programmeur de décider si une dimension correspond à des lignes, des colonnes, des vecteurs, ou toute autre organisation logique des données.

Une fois déclaré, un élément d'un tableau s'utilise comme une variable classique. Il est donc possible de lire ou d'écrire une case à l'aide de l'opérateur d'indice `[ ]`. Par exemple, la lecture d'un tableau d'entiers peut être réalisée par une boucle :

```
int t[10];
```

```
int i;
for (i = 0; i < 10; i++)
    scanf("%d", &t[i]);
```

On remarque ici que l'on transmet à `scanf` l'adresse de la case `t[i]`. En effet, la fonction `scanf` écrit directement dans la mémoire de la variable cible, et attend donc un pointeur.

En C, le nom d'un tableau est considéré comme un pointeur constant vers son premier élément. Autrement dit, si `t` est un tableau, l'expression `t` représente l'adresse de `t[0]`. Cette propriété est essentielle car elle explique de nombreux comportements du langage.

Le qualificatif constant signifie que l'adresse associée au tableau est fixe. Contrairement à une variable pointeur classique, il est impossible de modifier l'adresse portée par le tableau. Par exemple, une instruction du type `t = t + 1` est interdite : un tableau n'est pas une variable pointeur, mais une zone mémoire dont l'adresse de départ ne peut pas être modifiée.

En revanche, il est tout à fait possible de déclarer un pointeur et de lui affecter l'adresse du premier élément d'un tableau :

```
int t[10];
int *p;
p = t;
```

Dans ce cas, `p` est une variable pointeur, et il est possible de la modifier, par exemple avec de l'arithmétique sur les pointeurs.

Lorsqu'un tableau est utilisé comme argument dans un appel de fonction, il n'est pas recopié. À la place, seule l'adresse de sa première case est transmise. Le passage de paramètre d'un tableau se fait donc par adresse.

Une conséquence directe est que si une fonction modifie les éléments du tableau reçu en paramètre, ces modifications sont visibles dans la fonction appelante. Cette propriété est très utile, car elle évite la copie potentiellement coûteuse de grandes structures de données.

Dans la signature d'une fonction, la dimension du tableau est facultative pour un tableau à une dimension. Les deux écritures suivantes sont donc équivalentes :

```
void initialiser(int t[], int n);
void initialiser(int *t, int n);
```

Dans les deux cas, `t` est traité comme un pointeur vers des entiers. La taille réelle du tableau n'est pas connue automatiquement par la fonction. Il est donc nécessaire de transmettre explicitement cette taille, généralement sous la forme d'un paramètre supplémentaire.

Le cas des tableaux multidimensionnels est plus délicat. Lorsqu'un tableau à deux dimensions est passé en paramètre, la taille de toutes les dimensions sauf la première doit être précisée, afin que le compilateur puisse calculer correctement l'adresse d'un élément.

Par exemple, pour un tableau déclaré par `float tab2[2][5]`, une fonction peut être déclarée de la manière suivante :

```
void traiter(float t[][5], int nbLignes);
```

Ici, la valeur `5` doit obligatoirement apparaître dans le type du paramètre, car elle conditionne la disposition mémoire d'une ligne.

Le langage C ne réalise pas de vérification automatique des dépassements de tableaux. Un accès avec un indice en dehors des bornes constitue une erreur de programmation, et peut conduire à des comportements imprévisibles, parfois difficiles à diagnostiquer.

Il est donc essentiel de manipuler les indices avec rigueur, et de toujours conserver la taille d'un tableau lorsqu'elle n'est pas fixée explicitement et de manière évidente dans le code.

1.5.2 Chaînes de caractères

Le langage C ne propose pas de type chaîne de caractères au sens où on l'entend dans des langages plus récents. Il n'existe donc pas de type `string` en C. À la place, une chaîne est représentée par un tableau de caractères.

Une chaîne de caractères est un tableau de `char` dont le dernier élément significatif est un caractère spécial, appelé caractère nul, noté `'\0'`. Ce caractère marque la fin de la chaîne.

Par exemple, la déclaration suivante réserve un tableau de 30 caractères :

```
char nom[30];
```

Ce tableau peut contenir au plus 29 caractères visibles, car une case doit être réservée pour le caractère `'\0'`.

Ainsi, si l'utilisateur saisit n caractères, le tableau utilisera en réalité $n + 1$ cases, la dernière contenant `'\0'`. Cette convention est utilisée par toutes les fonctions classiques de manipulation de chaînes du C.

Une lecture simple au clavier peut être réalisée avec `scanf`. Par exemple :

```
char nom[30];
printf("Votre nom: ");
scanf("%s", nom);
```

On remarque que l'on ne place pas l'opérateur `&` devant `nom`. Cela s'explique par le fait que le nom du tableau `nom` est déjà l'adresse de sa première case, c'est-à-dire un pointeur constant vers `nom[0]`.

Il faut cependant être prudent : avec le spécificateur `%s`, la fonction `scanf` lit une suite de caractères jusqu'au premier séparateur (espace, tabulation ou retour à la ligne). De plus, `scanf` ne vérifie pas automatiquement que la chaîne saisie tient dans le tableau. Une saisie trop longue peut donc provoquer un dépassement de mémoire, ce qui constitue une erreur grave.

Dans un programme robuste, il est préférable d'utiliser des fonctions plus sûres, comme `fgets`, qui permet de limiter explicitement le nombre de caractères lus.

Le fait que les chaînes soient des tableaux implique plusieurs conséquences importantes.

Tout d'abord, une chaîne n'est pas un type valeur. Il n'est pas possible d'affecter directement une chaîne à une autre avec l'opérateur `=`. Par exemple, si `a` et `b` sont deux tableaux de caractères, une instruction du type `a = b` est interdite. Pour copier une chaîne, on doit utiliser une fonction dédiée, typiquement `strcpy` ou `strncpy`.

Ensuite, la taille maximale d'une chaîne stockée dans un tableau est fixée lors de la déclaration. Cette contrainte peut être levée en utilisant de l'allocation dynamique, ce qui sera abordé plus loin, notamment dans le cadre des pointeurs et de la gestion mémoire.

Enfin, il est important de distinguer deux notions souvent confondues : un tableau de caractères, et un pointeur vers des caractères. Un tableau `char nom[30]` réserve directement 30 cases mémoire. En revanche, une variable `char *p` ne réserve qu'un emplacement pour stocker une adresse. Pour que `p` puisse représenter une chaîne, il faut qu'il pointe vers une zone mémoire valide contenant une suite de caractères terminée par `'\0'`.

En C, une chaîne de caractères est donc une structure construite sur des tableaux et des pointeurs. Cette approche offre une grande efficacité, mais impose au programmeur de gérer explicitement la taille des zones mémoire et de prévenir les dépassements. Cette rigueur est indispensable pour produire un code correct et sûr.

1.5.3 Structures

Les structures permettent de regrouper, sous un même type, plusieurs champs pouvant être de types différents. Elles sont essentielles en C car elles permettent de construire des données plus riches que les types simples ou les tableaux, en donnant une représentation structurée à une entité du programme.

Une structure se définit avec le mot clé `struct`. La syntaxe générale est la suivante :

```
struct NomStructure {
    type1 champ1;
    type2 champ2;
    ...
};
```

Par exemple, une structure représentant une personne peut être définie ainsi :

```
struct Personne {
    char nom[30];
    char prenom[30];
};
```

Une variable de ce type peut ensuite être déclarée avec le nom complet du type :

```
struct Personne p;
```

Un point important du langage C est que la définition d'une structure crée un nouveau type dont le nom commence par `struct`. Autrement dit, le type n'est pas `Personne`, mais `struct Personne`.

Cette particularité est parfois surprenante au début, mais elle reflète l'approche historique du langage C, dans laquelle les mots clés `struct`, `union` et `enum` introduisent des espaces de noms distincts.

Pour éviter de devoir écrire `struct Personne` partout dans le code, on associe très souvent la déclaration de structure à un `typedef`. Cela permet de créer un alias de type plus pratique.

Par exemple :

```
typedef struct {
    char nom[30];
    char prenom[30];
} Personne;
```

Grâce à cet alias, on peut ensuite déclarer des variables simplement avec :

```
Personne p;
```

Cette écriture est très courante, notamment dans les bibliothèques et dans les programmes organisés en modules, car elle améliore nettement la lisibilité du code.

Les champs d'une structure sont accessibles via l'opérateur `.` (point). Par exemple :

```
Personne p;
scanf("%s", p.nom);
scanf("%s", p.prenom);
```

Lorsque l'on manipule un pointeur sur une structure, l'accès aux champs peut se faire avec l'opérateur `->`. Par exemple :

```
Personne *pp;
pp = &p;
printf("%s\n", pp->nom);
```

Cet opérateur est simplement une notation équivalente à une déréférence suivie d'un accès champ. Autrement dit, `pp->nom` est équivalent à `(*pp).nom`.

Une structure occupe en mémoire un bloc dont la taille est au moins la somme des tailles de ses champs. En pratique, la taille réelle peut être légèrement supérieure, car le compilateur peut insérer des octets de remplissage afin de respecter les contraintes d'alignement mémoire.

L'ordre des champs dans une structure a donc un impact sur l'organisation mémoire, et parfois sur la taille totale. Dans un cadre applicatif classique, ce détail n'est pas problématique, mais il devient important lorsqu'on travaille sur des données binaires, des formats de fichiers, ou des communications réseau.

Les structures constituent l'outil principal du langage C pour définir des types composites riches. Elles permettent de modéliser des objets du monde réel ou des entités logiques d'un programme, tout en restant compatibles avec la philosophie du C, fondée sur un contrôle explicite de la mémoire et des représentations.

1.5.4 Unions

Une union est un type composite proche d'une structure, mais dont la logique mémoire est très différente. Comme pour les structures, une union permet de regrouper plusieurs champs potentiellement de types différents. La différence essentielle est qu'une union ne réserve pas un espace mémoire pour chacun de ses champs. Au contraire, tous les champs partagent la même zone mémoire.

On définit une union avec le mot clé `union`. La syntaxe est similaire à celle des structures :

```
union NomUnion {
    type1 champ1;
    type2 champ2;
    ...
};
```

Par exemple :

```
union Nombre {
    int unEntier;
    float unReel;
};
```

La taille mémoire d'une union correspond à la taille de son plus grand champ. En effet, comme tous les champs partagent la même zone mémoire, l'union doit être suffisamment grande pour contenir le champ le plus volumineux.

Cette propriété rend les unions intéressantes dans certains contextes où l'on souhaite stocker une valeur qui peut prendre plusieurs formes possibles, sans allouer plus de mémoire que nécessaire. Une union peut par exemple représenter une valeur numérique qui est, selon le cas, un entier ou un réel.

Cependant, une union impose une contrainte importante : à un instant donné, un seul champ doit être considéré comme valide. Autrement dit, si l'on écrit dans le champ `unEntier`, puis que l'on lit le champ `unReel`, l'interprétation des bits en mémoire change complètement. Le langage n'effectue aucune conversion automatique. Il ne s'agit pas d'un transtypage, mais d'une relecture de la même représentation binaire avec un type différent.

Dans un code correct, il est donc indispensable de conserver, séparément, l'information indiquant quel champ est actuellement actif. Cette information est généralement stockée sous la forme d'un entier, ou plus proprement d'un type énuméré. L'association d'une union avec un champ indiquant son type effectif est un motif classique, souvent appelé union taguée.

Enfin, comme pour les structures, la taille réelle d'une union peut dépendre de l'alignement mémoire imposé par le compilateur. Dans la majorité des cas, ce détail n'a pas d'impact sur l'écriture du programme, mais il est important de savoir qu'une union ne garantit pas une taille strictement égale à celle de son plus grand champ.

Les unions sont donc un outil puissant mais délicat. Elles sont principalement utilisées dans des bibliothèques bas niveau, dans la manipulation de données binaires, ou lorsqu'une contrainte mémoire est critique. Dans un cadre applicatif classique, leur utilisation doit rester maîtrisée, car elles peuvent rendre le code plus difficile à comprendre et à maintenir.

1.6 Particularités du langage C

1.6.1 Le programme principal

En langage C, l'exécution d'un programme commence toujours dans une fonction particulière : la fonction `main`. C'est le point d'entrée du programme, c'est-à-dire la première fonction appelée lorsque l'exécutable est lancé par le système d'exploitation.

Contrairement à ce que l'on voit parfois dans des exemples simplifiés, la signature réelle de la fonction `main` n'est pas seulement `int main(void)`. La forme la plus générale est :

`int main(int argc, char **argv)`

Le type de retour `int` indique que `main` renvoie un code de retour au système d'exploitation. Par convention, une valeur nulle signifie que le programme s'est terminé correctement. Une valeur non nulle indique qu'une erreur a été détectée durant l'exécution.

Le paramètre `argc` (*argument count*) est un entier qui représente le nombre d'arguments passés au programme lors de son lancement. Le paramètre `argv` (*argument vector*) est un tableau de chaînes de caractères, qui contient les arguments eux-mêmes.

L'argument `argv[0]` est toujours le nom du programme tel qu'il a été invoqué. Les arguments suivants, s'ils existent, sont accessibles via `argv[1]`, `argv[2]`, etc.

Le code suivant illustre ce mécanisme en affichant le nombre d'arguments et leur contenu :

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
    int i;

    printf("argc: %d\n", argc);

    for (i = 0; i < argc; i++)
        printf("argv[%d]: %s\n", i, argv[i]);

    return EXIT_SUCCESS;
}
```

Dans cet exemple, la constante `EXIT_SUCCESS` provient de l'en-tête `stdlib.h`. Elle représente une terminaison correcte du programme. Il existe également la constante `EXIT_FAILURE` pour indiquer une terminaison en erreur.

L'accès aux arguments de la ligne de commande est particulièrement utile lorsque l'on souhaite paramétrer un programme sans interaction avec l'utilisateur. C'est un mécanisme très courant sous les systèmes de type UNIX, notamment pour les programmes de traitement de fichiers, les outils de compilation, ou les scripts automatisés.

Enfin, il est important de noter que `argv` est un tableau de pointeurs. Chaque élément `argv[i]` est un pointeur vers une chaîne de caractères terminée par `'\0'`. Cette représentation est cohérente avec la manière dont le langage C manipule les chaînes : il ne s'agit pas d'un type dédié, mais d'une convention fondée sur les tableaux de caractères.

1.6.2 Pointeurs de fonction

En langage C, une fonction n'est pas uniquement un bloc d'instructions que l'on appelle. Une fonction possède également une adresse en mémoire, comme une variable. Cette propriété permet de manipuler une fonction comme une valeur, c'est-à-dire de stocker son adresse dans une variable, puis de transmettre cette adresse à une autre fonction.

On parle alors de pointeur de fonction. Un pointeur de fonction est une variable dont la valeur est l'adresse d'une fonction, et dont le type décrit précisément la signature de cette fonction, c'est-à-dire le type de retour ainsi que les types de ses paramètres.

Cette fonctionnalité est particulièrement importante car elle permet d'écrire des fonctions génériques, capables de recevoir un comportement en paramètre. Par exemple, une fonction de tri peut recevoir en paramètre une fonction de comparaison, ce qui permet de changer la manière de trier sans modifier l'algorithme de tri lui-même.

Pour illustrer l'idée, considérons deux fonctions qui comparent deux entiers. L'une indique si le premier entier doit être placé avant le second pour un tri croissant, l'autre pour un tri décroissant :

```
int plusPetit(int a, int b) {
    return a > b;
```

```
}

int plusGrand(int a, int b) {
    return a < b;
}
```

Ces fonctions ont la même signature : elles prennent deux entiers et retournent un entier. Elles peuvent donc être stockées dans un même type de pointeur de fonction.

Supposons maintenant que l'on dispose d'une fonction `trierEtAfficherTableauDEntiers` qui prend en paramètre un pointeur de fonction correspondant à un comparateur. On peut alors choisir le comportement de tri au moment de l'appel :

```
int main() {
    trierEtAfficherTableauDEntiers(plusPetit);
    trierEtAfficherTableauDEntiers(&plusGrand);
    return 0;
}
```

On remarque que les deux écritures `plusPetit` et `&plusGrand` sont acceptées. En effet, le nom d'une fonction est interprété comme l'adresse de cette fonction. L'opérateur `&` est donc facultatif dans ce contexte.

L'intérêt des pointeurs de fonction est double. D'une part, ils permettent d'écrire du code plus modulaire, dans lequel un algorithme est séparé de la politique de traitement. D'autre part, ils permettent de factoriser du code, en évitant de dupliquer plusieurs versions d'un même algorithme pour des comportements légèrement différents.

Ce mécanisme est très utilisé dans les bibliothèques standards du C. Un exemple classique est la fonction `qsort`, qui réalise un tri générique en recevant en paramètre une fonction de comparaison. Les pointeurs de fonction sont également utilisés pour implémenter des tables de fonctions, des callbacks, ou encore des mécanismes proches du polymorphisme dans certains codes bas niveau.

1.6.3 Les bibliothèques

Le langage C est un langage modulaire. Le noyau du langage reste volontairement réduit, et l'on obtient la plupart des fonctionnalités supplémentaires en s'appuyant sur des bibliothèques. Cette organisation pousse naturellement à structurer un programme en plusieurs modules, plutôt que d'écrire un unique fichier source. Dans un projet bien organisé, le programme principal est souvent court et se contente d'orchestrer l'appel à des fonctions fournies par ces modules.

En pratique, un module est généralement réparti en deux fichiers. Le fichier d'en-tête, de type `.h`, décrit ce que le module rend disponible aux autres parties du programme. Il contient notamment les prototypes des fonctions publiques et, lorsqu'il y a lieu, les définitions des types nécessaires à l'utilisation de ces fonctions. Le fichier source, de type `.c`, contient l'implémentation, c'est-à-dire le code des fonctions déclarées dans le `.h`, ainsi que d'éventuels éléments internes au module. Cette séparation est importante car elle permet de compiler indépendamment chaque fichier source tout en conservant un typage cohérent et vérifiable par le compilateur.

La compilation d'un fichier `.c` produit un fichier objet `.o`. Un fichier objet contient du code machine, mais il n'est pas directement exécutable, notamment parce qu'il peut référencer des symboles définis ailleurs. La commande suivante illustre une compilation typique d'un module, avec activation des avertissements, vérification de conformité au standard et création du fichier objet dans le répertoire `src` :

```
gcc -osrc/module.o -Wall -pedantic -c -Iinclude src/module.c
```

Lorsque plusieurs fichiers objets sont disponibles, il devient possible de les regrouper dans une bibliothèque afin de faciliter leur réutilisation. Il existe deux grandes familles de bibliothèques en C, qui se distinguent par leur moment de liaison avec le programme. Une bibliothèque statique, généralement un fichier `.a` (`.lib` sous Windows), est intégrée à l'exécutable lors de l'édition de

liens. Le code nécessaire est alors copié dans le binaire final, ce qui rend l'exécutable plus autonome, mais potentiellement plus volumineux. Une bibliothèque statique se construit classiquement à l'aide de l'outil `ar`, par exemple :

```
ar -r libexemple.a module1.o module2.o
```

À l'inverse, une bibliothèque dynamique, généralement un fichier `.so` (ou `.dll` sous Windows), n'est pas recopiée dans l'exécutable. Le binaire final contient des références vers cette bibliothèque, et le chargement effectif est réalisé à l'exécution. Cette stratégie permet à plusieurs programmes de partager une même bibliothèque, et facilite la mise à jour d'un composant sans recompiler tous les exécutables qui l'utilisent, à condition que l'interface reste compatible. Une bibliothèque dynamique peut être produite avec `gcc` via une commande de la forme :

```
gcc -o libexemple.so -shared module1.o module2.o
```

Pour qu'une bibliothèque soit réellement utilisable, il ne suffit pas de fournir le fichier compilé `.a` ou `.so`. Il faut également fournir les fichiers d'en-tête associés, car ce sont eux qui décrivent l'interface et permettent au compilateur de vérifier les types et les signatures lors de la compilation du programme utilisateur. Par abus de langage, on appelle souvent bibliothèque l'ensemble constitué des fichiers `.h` et du fichier compilé correspondant. Il faut enfin garder à l'esprit qu'une bibliothèque peut regrouper plusieurs fichiers objets, et qu'inversement plusieurs fichiers d'en-tête peuvent être associés à une même bibliothèque, selon la manière dont l'interface est organisée.

1.6.4 Inclusions multiples et ordre définition / inclusion

Dans un projet C modulaire, les fichiers d'en-tête sont souvent inclus dans plusieurs fichiers, parfois directement dans un `.c`, parfois indirectement via d'autres `.h`. Cette organisation est normale, mais elle peut conduire à des inclusions multiples, car `#include` correspond à une inclusion textuelle effectuée par le préprocesseur : le contenu du fichier est copié à l'endroit de l'inclusion.

Le cas devient particulièrement instructif lorsque deux fichiers d'en-tête définissent deux types `TypeA` et `TypeB`, et que chacun déclare une fonction de transtypage vers l'autre type. Par exemple, on souhaite déclarer dans `TypeA.h` une fonction `TypeBEnTypeA` qui prend un `TypeB`, et dans `TypeB.h` une fonction `TypeAEnTypeB` qui prend un `TypeA`. Dans ce contexte, on est tenté d'écrire une première version où chaque fichier inclut l'autre avant de faire ses propres définitions.

Une version incorrecte typique est la suivante :

```
/* TypeA.h: version incorrecte */
#include "TypeB.h"
typedef int TypeA;
TypeA TypeBEnTypeA(TypeB);

/* TypeB.h: version incorrecte */
#include "TypeA.h"
typedef int TypeB;
TypeB TypeAEnTypeB(TypeA);
```

Cette organisation déclenche un enchaînement d'inclusions. Lorsque `TypeA.h` est inclus, il inclut `TypeB.h`, qui inclut à son tour `TypeA.h`, et ainsi de suite. Même si le compilateur finit par signaler une redéfinition, le problème de fond est plus général : au moment où `TypeA.h` déclare `TypeBEnTypeA(TypeB)`, le type `TypeB` doit déjà être connu. Or, dans la version incorrecte, `TypeB` n'est défini qu'après l'inclusion de `TypeA.h`. On se retrouve donc avec des prototypes qui utilisent un type non défini au moment où le compilateur les lit.

La résolution repose sur deux idées complémentaires. D'abord, on protège chaque fichier d'en-tête contre les inclusions multiples à l'aide d'include guards. Ensuite, et c'est le point essentiel dans cet exemple, on place la définition du type du fichier courant avant l'inclusion de l'autre fichier. Ainsi, dès qu'un des fichiers est rencontré, son type devient immédiatement connu, ce qui permet à l'autre fichier de déclarer ses prototypes de transtypage.

On obtient alors une version correcte :

```
/* TypeA.h: version correcte */
#ifndef __TYPE_A__
#define __TYPE_A__

typedef int TypeA;
#include "TypeB.h"
TypeA TypeBEnTypeA(TypeB);

#endif

/* TypeB.h: version correcte */
#ifndef __TYPE_B__
#define __TYPE_B__

typedef int TypeB;
#include "TypeA.h"
TypeB TypeAEnTypeB(TypeA);

#endif
```

Dans cette version, la définition `typedef int TypeA;` est placée avant `#include "TypeB.h"`. Ainsi, lorsque `TypeB.h` inclut `TypeA.h`, le type `TypeA` est défini immédiatement, ce qui permet ensuite à `TypeB.h` de déclarer correctement `TypeAEnTypeB(TypeA)`. Le même raisonnement s'applique symétriquement à `TypeB.h`.

Cet exemple montre donc deux règles de base pour écrire des fichiers d'en-tête robustes. Un fichier `.h` doit toujours être protégé contre les inclusions multiples. De plus, lorsqu'un en-tête doit exposer un type et des fonctions qui manipulent ce type, la définition du type doit apparaître suffisamment tôt pour que les autres en-têtes qui en dépendent puissent l'utiliser au moment de la lecture des prototypes. L'ordre entre définition et inclusion n'est donc pas un détail de style : c'est une contrainte de compilation.

1.6.5 Macros paramétrées (macro-fonctions)

Le préprocesseur du langage C permet de définir des macros, c'est-à-dire des règles de substitution textuelle appliquées avant la compilation. Une macro paramétrée, parfois appelée macro-fonction, ressemble à une fonction car elle possède des paramètres, mais son mécanisme est très différent : il ne s'agit pas d'un appel de fonction, il s'agit d'un remplacement de texte.

Une macro paramétrée est définie avec la directive `#define`. Sa forme générale consiste à donner un nom, une liste de paramètres, puis un corps qui sera substitué. La substitution intervient partout où le nom de la macro apparaît ensuite dans le code, après analyse par le préprocesseur.

Les macros paramétrées sont parfois utilisées pour éviter le coût d'un appel de fonction dans des portions critiques, ou pour exprimer des opérations génériques sans recourir au typage. En contrepartie, elles sont plus délicates à maîtriser, car la substitution textuelle ignore les règles habituelles d'évaluation des expressions et de contrôle de portée.

Dans l'exemple suivant, on définit une macro paramétrée sur plusieurs lignes. Elle affiche une addition, calcule un résultat, puis affiche ce résultat :

```
#include <stdio.h>
#include <stdlib.h>

#define ADD(a, b, c) \
printf("%d + %d ", a, b); \
c = a + b; \
```

```
printf("= %d\n", c);

int main(int argc, char *argv[]) {
    int i = 0;
    int r = 0;

    while (argv[i++])
        ADD(r, 1, r);

    return EXIT_SUCCESS;
}
```

Cet exemple illustre un premier piège : une macro paramétrée peut contenir plusieurs instructions, mais, du point de vue du langage C, son utilisation reste une simple substitution dans le code. Ici, l'appel `ADD(r, 1, r)` remplace textuellement l'appel par trois instructions consécutives. Si l'on utilise cette macro dans un contexte où une seule instruction est attendue, par exemple juste après un `if` sans accolades, le comportement peut devenir incorrect.

Un autre piège fréquent est l'utilisation multiple d'un même paramètre dans le corps de la macro. Si un paramètre est remplacé plusieurs fois, et si l'utilisateur fournit une expression avec effets de bord, alors ces effets peuvent être déclenchés plusieurs fois. Par exemple, passer `i++` à une macro qui utilise son paramètre deux fois conduit à deux incrémentations, ce qui n'est généralement pas souhaité.

Les macros paramétrées sont donc puissantes, mais elles doivent être utilisées avec rigueur. Lorsqu'une fonctionnalité peut être exprimée correctement avec une fonction, cette solution est souvent préférable car elle bénéficie du typage, du débogage et d'un comportement bien défini. Les macros paramétrées restent néanmoins utiles dans certains cas, en particulier pour des constructions génériques, des constantes symboliques, ou des mécanismes conditionnels contrôlés par le préprocesseur.

1.6.6 Mots clés `static` et `extern`

Le langage C propose plusieurs mots clés qui ne modifient pas directement le déroulement des instructions, mais qui influencent la durée de vie, la visibilité et l'édition de liens des variables et des fonctions. Parmi eux, `static` et `extern` jouent un rôle central dans l'écriture de programmes modulaires.

Le mot clé `static` possède deux usages principaux, qu'il est indispensable de distinguer. Lorsqu'il est utilisé devant une variable locale d'une fonction, `static` modifie la durée de vie de cette variable. Une variable locale classique est créée à chaque appel de la fonction, puis détruite à la fin de l'exécution. À l'inverse, une variable locale déclarée `static` n'est initialisée qu'une seule fois et conserve sa valeur entre deux appels. La variable reste locale au sens de sa portée, car elle n'est accessible que dans la fonction, mais sa durée de vie devient celle du programme entier.

Par exemple :

```
void foo() {
    static int compteurDAppels = 0;
    compteurDAppels++;
}
```

Dans cet exemple, `compteurDAppels` permet de mémoriser combien de fois la fonction `foo` a été appelée. Si la variable n'était pas déclarée `static`, elle serait recréée à chaque appel et sa valeur serait réinitialisée.

Le second usage de `static` concerne les variables globales et les fonctions. Lorsqu'il est placé devant la définition d'une variable globale ou d'une fonction, `static` limite la visibilité du symbole au fichier source courant. La variable ou la fonction devient alors privée au module, c'est-à-dire qu'elle ne peut pas être utilisée depuis un autre fichier `.c`. Ce mécanisme est essentiel pour éviter

les conflits de noms entre modules et pour empêcher qu'un utilisateur d'une bibliothèque accède à des éléments internes qui ne font pas partie de l'interface publique.

Le mot clé `extern` joue un rôle complémentaire. Il est utilisé pour indiquer qu'un symbole est défini ailleurs, généralement dans un autre fichier source. Il s'applique principalement aux variables globales. Lorsqu'une variable globale est définie dans un fichier, on peut la rendre accessible dans un autre fichier en écrivant une déclaration `extern` correspondante. Le rôle de `extern` est donc de dire au compilateur que le symbole existe, mais que sa définition réelle sera trouvée lors de l'édition de liens.

Pour les fonctions, l'utilisation de `extern` est généralement implicite. En effet, une fonction déclarée dans un fichier d'en-tête est par défaut considérée comme définie dans un autre fichier. Le mot clé `extern` peut être écrit explicitement, mais il est facultatif.

Les mots clés `static` et `extern` sont donc fondamentaux pour comprendre la différence entre portée, durée de vie et visibilité entre fichiers. Ils permettent de contrôler précisément ce qui doit rester interne à un module, ce qui doit être partagé entre plusieurs fichiers, et ce qui doit être accessible depuis l'extérieur.

1.6.7 Allocation statique et allocation dynamique

Lorsqu'un programme en C manipule des données, il doit nécessairement réserver de la mémoire pour les stocker. Cette réservation peut être réalisée de plusieurs manières, qui correspondent à des durées de vie et des usages différents. On distingue principalement l'allocation statique et l'allocation dynamique.

L'allocation statique correspond à une réservation mémoire dont la taille est connue à la compilation, et dont la durée de vie est déterminée par la structure du programme. Par exemple, une variable locale déclarée dans une fonction est allouée automatiquement lors de l'appel de la fonction, puis libérée automatiquement à la fin de l'exécution de cette fonction. On parle souvent d'allocation automatique, car le programmeur n'a rien à gérer explicitement. Une variable globale, quant à elle, est allouée une seule fois pour toute la durée du programme.

Ces allocations sont simples à utiliser, sûres du point de vue de la gestion mémoire, et très efficaces. En revanche, elles ont une limitation importante : la taille des structures de données doit être fixée lors de l'écriture du code, ou au moins déterminable sans intervention de l'utilisateur au moment de l'exécution.

L'allocation dynamique permet de lever cette contrainte. Elle consiste à demander explicitement au système une zone mémoire dont la taille est calculée pendant l'exécution. Cette zone est obtenue dans le tas (*heap*). L'allocation dynamique est indispensable dès que l'on souhaite créer des structures de taille variable, par exemple un tableau dont la taille dépend d'une saisie utilisateur, une liste chaînée, un arbre, ou encore une structure de données construite progressivement.

En C, l'allocation dynamique est réalisée principalement à l'aide de la fonction `malloc`, définie dans l'en-tête `stdlib.h`. La fonction `malloc` prend en paramètre une taille en octets, et retourne un pointeur vers une zone mémoire suffisamment grande, ou `NULL` si l'allocation échoue.

Par exemple, pour allouer un tableau de n entiers :

```
#include <stdlib.h>

int *t;
int n = 100;

t = (int*) malloc(n * sizeof(int));
```

L'expression `sizeof(int)` permet de rendre le code portable, car la taille d'un `int` dépend de l'architecture et du compilateur. On obtient ainsi un tableau dynamique contenant n cases de type `int`.

Une fois la zone mémoire allouée, elle se manipule comme un tableau classique. Par exemple, `t[i]` est valide pour tout indice compris entre 0 et $n - 1$. Il faut cependant garder à l'esprit

que, contrairement à un tableau statique, la taille n'est pas connue du compilateur. Il est donc nécessaire de conserver la valeur de `n` pour éviter tout dépassement.

Une allocation dynamique doit être libérée explicitement. En C, la libération se fait avec la fonction `free`. Oublier de libérer une zone mémoire conduit à une fuite mémoire, c'est-à-dire une perte progressive de mémoire disponible. À l'inverse, libérer deux fois la même zone mémoire, ou libérer une adresse qui n'a pas été obtenue par `malloc`, est une erreur grave qui peut conduire à un crash ou à un comportement imprévisible.

La fonction `calloc` est une variante de `malloc` qui alloue une zone mémoire et initialise tous les octets à zéro. Elle est utile lorsque l'on souhaite obtenir une zone mémoire propre, notamment pour des tableaux. La fonction `realloc` permet quant à elle de redimensionner une zone déjà allouée, ce qui est pratique lorsqu'on veut agrandir progressivement une structure de données.

L'allocation dynamique donne donc une grande flexibilité, mais elle impose une discipline stricte. Le programmeur doit vérifier le résultat de `malloc`, conserver les pointeurs retournés, libérer toutes les zones allouées, et s'assurer de ne jamais utiliser une zone après l'avoir libérée. La maîtrise de ces règles est indispensable pour écrire des programmes fiables en C.

Enfin, il est important de souligner que l'allocation dynamique est intimement liée à l'utilisation des pointeurs. La plupart des structures de données avancées en algorithmique, comme les listes chaînées, les piles, les files, ou les arbres, reposent sur des allocations dynamiques et sur la manipulation rigoureuse de pointeurs.

1.6.8 La bibliothèque standard

Le langage C fournit un ensemble de bibliothèques standardisées, accessibles via des fichiers d'en-tête, qui complètent le noyau du langage. Ces bibliothèques définissent des types, des constantes et des fonctions qui permettent de réaliser les opérations usuelles, comme l'affichage, la gestion de chaînes de caractères, l'allocation mémoire, ou encore l'accès aux fichiers.

Dans cette section, on s'intéresse plus particulièrement à deux fichiers d'en-tête importants, qui permettent d'écrire des programmes plus robustes : `errno.h` et `assert.h`. Le premier est lié à la gestion des erreurs dans les appels de fonctions, et le second permet de vérifier des propriétés du programme pendant son exécution.

`errno.h`

De nombreuses fonctions de la bibliothèque standard, et plus généralement de l'interface POSIX, signalent une erreur en renvoyant une valeur particulière. Par exemple, certaines fonctions retournent `NULL`, d'autres retournent `-1`, et d'autres encore retournent une valeur spéciale indiquant un échec.

Lorsqu'une erreur survient, ces fonctions peuvent également mettre à jour une variable globale nommée `errno`. Cette variable contient un code numérique représentant la cause de l'erreur. Le fichier d'en-tête `errno.h` définit cette variable ainsi que les constantes associées aux codes d'erreurs les plus courants.

L'utilisation de `errno` permet donc d'obtenir une information plus précise que le simple test de la valeur de retour. Elle est particulièrement utile lorsqu'une fonction peut échouer pour plusieurs raisons différentes.

Par exemple, la fonction `fopen` retourne `NULL` si l'ouverture d'un fichier échoue. Dans ce cas, `errno` est mis à jour et indique la cause, par exemple un fichier inexistant, un problème de permission, ou un chemin invalide. Une manière classique de traiter cette situation consiste à tester le retour, puis à utiliser une fonction d'affichage d'erreur.

La fonction `perror`, disponible via `stdio.h`, affiche un message correspondant à la valeur courante de `errno`. On peut également utiliser `strerror` (dans `string.h`) pour obtenir une chaîne de caractères décrivant l'erreur.

L'exemple suivant illustre une ouverture de fichier robuste :

```
#include <stdio.h>
#include <stdlib.h>
```

```c
#include <errno.h>

int main() {
    FILE *f;

    f = fopen("donnees.txt", "r");
    if (f == NULL) {
        perror("Erreur fopen");
        return EXIT_FAILURE;
    }

    fclose(f);
    return EXIT_SUCCESS;
}
```

Dans cet exemple, la valeur de retour de `fopen` est testée immédiatement. Si l'ouverture échoue, le programme affiche un message explicite puis se termine proprement.

Il est important de noter que `errno` n'a de sens que juste après un appel de fonction ayant échoué. En effet, `errno` n'est pas automatiquement remis à zéro après un succès. De plus, certaines fonctions peuvent modifier `errno` même en cas de réussite. Une bonne pratique consiste donc à ne consulter `errno` qu'après avoir détecté une valeur de retour indiquant un échec.

assert.h

Le fichier d'en-tête `assert.h` fournit une macro nommée `assert`. Cette macro permet de vérifier, à l'exécution, qu'une condition est vraie. Si la condition est fausse, le programme s'arrête immédiatement et affiche un message d'erreur contenant la condition testée, ainsi que le fichier et la ligne où l'erreur a été détectée.

La macro `assert` est particulièrement utile pour exprimer des hypothèses internes au programme, c'est-à-dire des propriétés qui doivent être vraies si le code est correct. Il ne s'agit pas d'un mécanisme de gestion d'erreur destiné à l'utilisateur, mais d'un outil de vérification pour le programmeur.

Par exemple, si une fonction suppose qu'un pointeur reçu en paramètre n'est jamais `NULL`, on peut écrire :

```c
#include <assert.h>

void traiter(int *p) {
    assert(p!= NULL);
    /* utilisation de p */
}
```

Si la fonction `traiter` est appelée avec un pointeur nul, le programme s'arrête immédiatement. Cette détection précoce permet d'éviter des erreurs plus graves, comme une déréférence de pointeur nul, qui conduirait à un crash moins explicite.

L'utilisation de `assert` est également utile pour vérifier des invariants d'algorithmes ou des préconditions de fonctions. Par exemple, une fonction de recherche dans un tableau peut vérifier que la taille est positive, ou qu'un indice est bien dans les bornes attendues.

Enfin, il est important de souligner que `assert` est une macro contrôlée par le préprocesseur. Si le symbole `NDEBUG` est défini lors de la compilation, alors toutes les assertions sont désactivées et ne produisent plus aucun code. Cela permet de conserver les assertions durant le développement et les tests, puis de les retirer automatiquement dans une version finale destinée à la production.

Ainsi, `assert` constitue un outil simple mais puissant pour renforcer la fiabilité d'un programme C, en détectant tôt les incohérences et en documentant explicitement les hypothèses faites par le code.

1.7 Compilation d'un grand projet

Lorsque l'on passe d'un exercice court à un projet organisé en modules, la compilation ne consiste plus à appeler une seule commande `gcc` sur un fichier. On souhaite compiler séparément chaque module, regrouper certains objets dans une bibliothèque, puis lier l'exécutable final. L'objectif est double : obtenir une compilation reproductible et, surtout, éviter de recompiler inutilement tout le projet lorsqu'un seul fichier change.

Prenons comme exemple un programme de tri par minimum successif. Le projet est organisé de manière classique, avec un répertoire `src` contenant les fichiers `.c`, un répertoire `include` pour les fichiers `.h`, un répertoire `lib` pour les bibliothèques produites, et un répertoire `bin` pour l'exécutable final. L'exécutable `bin/tri` dépend de plusieurs modules, dont un module utilitaire `echanger` fourni sous la forme d'une bibliothèque statique `lib/libechanger.a`.

1.7.1 Une première solution : un script `bash`

Une première manière de compiler un tel projet consiste à écrire un script `bash` qui enchaîne toutes les commandes nécessaires.

```
#!/bin/bash
gcc -o src/echanger.o -Wall -pedantic -c -Iinclude src/echanger.c
ar -r lib/libechanger.a src/echanger.o
gcc -o src/triParMinSuccessif.o -Wall -pedantic -c -Iinclude src/triParMinSuccessif.c
gcc -o src/main.o -Wall -pedantic -c -Iinclude src/main.c
gcc -o bin/tri src/main.o src/triParMinSuccessif.o -Llib -lechanger
```

Il compile d'abord `echanger.c` en fichier objet, construit la bibliothèque `libechanger.a`, compile ensuite les autres modules en objets, puis réalise l'édition de liens de l'exécutable :

Ce script met en évidence les étapes réelles de construction. Les compilations avec l'option `-c` produisent des fichiers objets `.o`. La commande `ar` regroupe certains objets dans une bibliothèque statique `.a`. Enfin, l'édition de liens produit l'exécutable en liant les objets du projet avec la bibliothèque `echanger` via `-Llib` et `-lechanger`.

Cette approche fonctionne, mais elle présente une limitation importante : le script exécute toutes les commandes à chaque fois, même si un seul fichier a été modifié.

1.7.2 Pourquoi passer à `make`

L'outil `make` est conçu pour automatiser ce processus en tenant compte des dépendances. L'idée est de décrire ce que l'on veut produire, de préciser de quoi cela dépend, et de laisser `make` décider quoi recompiler. Si un fichier source n'a pas changé depuis la dernière compilation, il n'est pas recompilé. Si, au contraire, un fichier source ou un en-tête impactant un module a changé, seuls les éléments nécessaires sont reconstruits.

Un fichier `Makefile` est composé de deux types d'éléments : des variables, qui factorisent les commandes et options, et des règles, qui expriment les dépendances entre fichiers. Une règle indique une cible, ses dépendances, et les actions à exécuter pour construire la cible.

1.7.3 Un premier `Makefile` équivalent au script

On peut commencer par écrire un `Makefile` qui reproduit exactement les étapes du script. La différence est que l'on exprime des cibles et des dépendances, ce qui permet à `make` de ne construire que ce qui est nécessaire.

Le fichier suivant correspond à une première version simple :

```
all: bin/tri

bin/tri: lib/libechanger.a src/triParMinSuccessif.o src/main.o
	gcc -o bin/tri src/main.o src/triParMinSuccessif.o -Llib -lechanger
```

```
lib/libechanger.a: src/echanger.o
	ar -r lib/libechanger.a src/echanger.o

src/echanger.o: src/echanger.c
	gcc -o src/echanger.o -Wall -pedantic -c -Iinclude src/echanger.c

src/triParMinSuccessif.o: src/triParMinSuccessif.c
	gcc -o src/triParMinSuccessif.o -Wall -pedantic -c -Iinclude src/triParMinSuccessif.c

src/main.o: src/main.c
	gcc -o src/main.o -Wall -pedantic -c -Iinclude src/main.c
```

Cette version décrit déjà correctement les dépendances. Si l'on modifie uniquement `src/main.c`, `make` recompilera `src/main.o` puis relancera l'édition de liens de `bin/tri`, mais ne recompilera pas `echanger.c` et ne reconstruira pas `libechanger.a`.

1.7.4 Factorisation avec des variables de `make`

La version précédente fonctionne, mais elle duplique de nombreuses options. `make` permet de factoriser ces éléments en variables, ce qui rend le fichier plus lisible et plus simple à maintenir.

On utilise typiquement une variable pour le compilateur, des variables pour les options, et des variables décrivant les fichiers sources et objets. Une convention courante consiste à utiliser `CC` pour le compilateur C, `CFLAGS` pour les options de compilation, `CPPFLAGS` pour les options de préprocesseur comme `-I`, et `LDFLAGS` ou `LDLIBS` pour l'édition de liens.

Une version plus propre du même fichier est la suivante :

```
CC = gcc
AR = ar
ARFLAGS = -r
CFLAGS = -Wall -pedantic
CPPFLAGS = -Iinclude
BIN = bin/tri
LIB = lib/libechanger.a

OBJ_MAIN = src/main.o src/triParMinSuccessif.o
OBJ_LIB = src/echanger.o

all: $(BIN)

$(BIN): $(LIB) $(OBJ_MAIN)
	$(CC) -o $@ $(OBJ_MAIN) -Llib -lechanger

$(LIB): $(OBJ_LIB)
	$(AR) $(ARFLAGS) $@ $(OBJ_LIB)

src/main.o: src/main.c
	$(CC) $(CFLAGS) $(CPPFLAGS) -c -o $@ $<

src/triParMinSuccessif.o: src/triParMinSuccessif.c
	$(CC) $(CFLAGS) $(CPPFLAGS) -c -o $@ $<

src/echanger.o: src/echanger.c
	$(CC) $(CFLAGS) $(CPPFLAGS) -c -o $@ $<
```

Cette version introduit aussi des variables automatiques de `make`. Dans une règle, `$@` représente la cible, et `$<` représente la première dépendance. Cela évite de recopier les noms de fichiers dans les commandes, ce qui limite les erreurs lors des modifications.

1.7.5 Règles d'inférence et variables automatiques

La majorité des règles de compilation des fichiers objets ont exactement la même forme : un fichier `.o` se construit à partir d'un fichier `.c` en appelant `gcc` avec les bons drapeaux. Écrire une règle distincte pour chaque fichier devient vite inutilement répétitif.

`make` propose alors deux mécanismes complémentaires. Le premier consiste à écrire une règle de motif, qui décrit une famille entière de règles. Le second consiste à utiliser les variables automatiques pour récupérer, dans la commande, le nom de la cible et celui de la dépendance.

Une règle de motif classique pour la compilation est :

```
%.o: %.c
	$(CC) $(CFLAGS) $(CPPFLAGS) -c -o $@ $<
```

Avec cette seule règle, `make` sait comment construire n'importe quel fichier `.o` à partir du `.c` correspondant, tant que les chemins correspondent. Dans ce cadre, on peut simplifier le `Makefile` en listant uniquement les objets nécessaires et en laissant `make` déduire la manière de les produire.

On obtient alors un fichier plus concis :

```
CC = gcc
AR = ar
ARFLAGS = -r
CFLAGS = -Wall -pedantic
CPPFLAGS = -Iinclude
BIN = bin/tri
LIB = lib/libechanger.a

OBJ_MAIN = src/main.o src/triParMinSuccessif.o
OBJ_LIB = src/echanger.o

all: $(BIN)

$(BIN): $(LIB) $(OBJ_MAIN)
	$(CC) -o $@ $(OBJ_MAIN) -Llib -lechanger

$(LIB): $(OBJ_LIB)
	$(AR) $(ARFLAGS) $@ $^

%.o: %.c
	$(CC) $(CFLAGS) $(CPPFLAGS) -c -o $@ $<

clean:
	rm -f $(OBJ_MAIN) $(OBJ_LIB) $(BIN) $(LIB)
```

Dans la règle de construction de la bibliothèque, on a utilisé `$^`, qui représente l'ensemble des dépendances. Cette variable est pratique lorsqu'une cible dépend de plusieurs fichiers objets. La cible `clean` permet quant à elle de supprimer les fichiers générés pour repartir d'un état propre avant une recompilation.

Avec cette organisation, on obtient un mécanisme de compilation plus fiable et plus efficace que le script initial. La compilation devient incrémentale, la structure du projet est explicitée par des dépendances, et les options sont centralisées dans des variables, ce qui rend le projet plus simple à faire évoluer.

Chapitre 2

Les types abstraits de données

2.1 De l'analyse au développement

Lors de l'analyse d'un problème, les données manipulées ne se limitent pas nécessairement aux types élémentaires proposés par un langage de programmation. Un programme peut par exemple manipuler des graphiques, des écrans, des points du plan ou des polylignes. Ces entités correspondent à des concepts du problème et doivent pouvoir être décrits avant que l'on choisisse la manière de les représenter en mémoire.

L'objectif de l'analyse n'est donc pas de déterminer immédiatement comment les entités seront construites ou stockées. Il s'agit d'abord de préciser ce qu'elles représentent, comment elles sont utilisées et comment elles interviennent dans la résolution du problème. Cette distinction permet de raisonner sur le problème sans dépendre trop tôt d'un langage de programmation ou d'une représentation particulière.

2.1.1 Types et analyse

Une analyse décrit la résolution d'un problème à un niveau abstrait. Elle peut notamment décomposer cette résolution en fonctions hiérarchiquement organisées, en précisant les données nécessaires à chaque étape et les résultats produits. Pour que cette description soit complète, les types utilisés par ces fonctions doivent eux aussi être définis de manière abstraite.

Par exemple, une fonction qui calcule la longueur d'une polyligne doit pouvoir manipuler le concept de polyligne et celui de point, même si l'implantation utilisera finalement des structures, des tableaux ou des pointeurs. À ce stade, la question importante est de savoir quelles opérations sont nécessaires sur ces entités, et non de connaître leur organisation mémoire.

Cette séparation entre l'utilisation d'une donnée et sa représentation est au cœur de la démarche par types abstraits de données. Elle permet de faire évoluer l'implantation sans remettre en cause l'analyse, à condition de conserver le comportement attendu des opérations.

2.1.2 Qu'est-ce qu'un type de données ?

Mathématiquement, un type de données est un ensemble d'éléments auquel est associé un ensemble d'opérations. Ces opérations permettent soit de désigner des éléments de l'ensemble, soit de manipuler des éléments déjà désignés pour en produire de nouveaux.

Les opérations de désignation permettent de construire ou d'identifier des éléments du type. Les opérations de manipulation permettent ensuite de calculer des éléments, qui peuvent appartenir au même type ou à un autre type. Cette distinction ne dépend pas de la manière dont les éléments sont représentés en machine : elle décrit les possibilités offertes par le type au niveau de l'analyse.

Le type booléen fournit un exemple simple. Son ensemble d'éléments contient deux éléments, généralement notés `vrai` et `faux`. Les opérations `vrai` et `faux` désignent ces deux éléments. Les

opérations `et`, `ou` et `non` manipulent des éléments booléens pour produire un nouvel élément booléen.

Ainsi, la description d'un type ne consiste pas seulement à donner un nom à un ensemble d'éléments. Elle doit également préciser les opérations disponibles et le rôle de chacune d'elles. Cette description constitue la base sur laquelle pourront être définis les algorithmes qui utilisent le type.

2.1.3 Les étapes de création d'un programme

La création d'un programme peut être organisée en trois étapes complémentaires : l'analyse, la conception et le développement. Cette organisation permet de séparer les décisions relatives au problème de celles qui concernent sa mise en œuvre.

Analyse. Cette étape consiste à identifier et à définir les types abstraits de données nécessaires au programme. Elle comprend également une analyse descendante de la résolution du problème, c'est-à-dire une décomposition progressive du problème en fonctions ou en tâches plus simples.

Conception. La conception précise la manière dont les éléments identifiés lors de l'analyse seront organisés pour obtenir une solution réalisable. Elle comporte deux niveaux. La conception préliminaire consiste à choisir un paradigme de programmation et donc à indiquer comment les opérations du TAD sont utilisables. Dans notre cas, celui de la programmation structurée, il faut donner les signatures des fonctions et des procédures correspondant aux opérations des TAD, comme nous l'avons fait jusqu'à présent pour l'analyse descendante. La conception détaillée consiste ensuite à écrire les algorithmes qui conduisent à la solution et à choisir une représentation concrète pour les entités de l'analyse. Ce choix donne corps aux fonctionnalités décrites au niveau abstrait.

Développement. Enfin, le développement consiste à choisir un langage de programmation et à traduire dans ce langage les éléments définis pendant l'analyse et la conception. Cette traduction ne doit pas modifier le sens des opérations ni le comportement attendu des algorithmes.

2.2 Définition et spécification d'un type abstrait de données

Après avoir défini ce qu'est un TAD, il faut disposer d'un formalisme pour le représenter. Ce formalisme doit rendre explicites les caractéristiques du TAD et décrire sans ambiguïté les opérations qui lui sont associées.

2.2.1 Les constituants d'un TAD

La description d'un TAD doit fournir suffisamment d'informations pour que son utilisation soit non ambiguë. Elle comporte généralement les constituants suivants :

Le nom. Le nom identifie le TAD dans l'analyse et permet de le distinguer des autres types utilisés par le programme.

Les paramètres. Un TAD peut être paramétré par un ou plusieurs types. Cette possibilité est notamment utile pour définir un TAD qui manipule des éléments dont le type n'est pas fixé à l'avance. Une collection peut ainsi être définie pour un type générique d'éléments.

Les dépendances. Les dépendances indiquent les types déjà définis dont le TAD utilise les éléments ou les opérations. Elles rendent explicites les notions nécessaires à la définition du nouveau type.

Les opérations. Les opérations constituent l'interface du TAD. Elles permettent de désigner des éléments, de produire de nouveaux éléments ou d'obtenir des informations sur les éléments manipulés.

La définition du comportement. Les opérations doivent être accompagnées d'une description de leur comportement. Cette description peut prendre la forme d'axiomes ou d'une sémantique exprimée en langue naturelle.

Les préconditions. Certaines opérations ne sont définies que lorsque leurs paramètres respectent une condition particulière. Cette condition est appelée précondition et doit être indiquée dans la spécification du TAD.

Tous ces constituants ne sont pas obligatoires dans les mêmes circonstances. Un TAD simple peut ne nécessiter ni paramètre ni dépendance, alors qu'un TAD générique ou construit à partir d'autres types devra les préciser.

2.2.2 Les opérations d'un TAD

Les opérations décrivent les services rendus par le TAD. Elles peuvent être regroupées selon leur rôle. Les opérations de construction ou de désignation permettent d'obtenir des éléments du type. Les opérations d'observation permettent d'extraire une information d'un élément sans en modifier la signification. Les opérations de manipulation produisent également un élément ou un résultat à partir d'un ou de plusieurs éléments.

Lors de la spécification d'un TAD, on se situe au niveau de l'analyse, celui des mathématiques. Les opérations décrites sont donc des fonctions qui associent des éléments à d'autres éléments ou à des résultats. Elles ne modifient pas l'état des entités : la notion de modification de l'état apparaîtra seulement au moment de concevoir puis d'implanter le TAD dans un langage de programmation.

Une opération est décrite par son nom, les types de ses paramètres d'entrée et les types de ses résultats. Sa signature indique donc comment elle peut être appelée, mais elle ne suffit pas à définir ce qu'elle fait. Deux opérations peuvent avoir des signatures différentes tout en nécessitant une description précise de leurs effets et de leurs résultats.

Dans une spécification de TAD, on considère généralement que l'égalité et l'inégalité sont des opérations disponibles pour tout type. Elles sont notées `=` et `!=`. Leur présence ne dispense pas de préciser ce que signifie l'égalité entre deux éléments du TAD. Selon le type considéré, il peut s'agir d'une égalité définie par les propriétés des éléments, d'une égalité structurelle ou d'une autre relation explicitement choisie dans la spécification.

2.2.3 Le comportement des opérations

Une interface qui ne contient que les noms et les signatures des opérations reste incomplète. Elle indique comment demander un service, mais pas le résultat que ce service doit produire. La définition du comportement est donc une partie essentielle du TAD : elle sert de référence pour les algorithmes qui l'utilisent, pour son implantation et pour les tests qui vérifieront cette implantation.

Le comportement peut être défini par des axiomes. Un axiome exprime alors une propriété logique du TAD ou le résultat d'une composition d'opérations. Cette forme convient particulièrement aux types dont les opérations peuvent être décrites par des équations.

Lorsque les axiomes seraient trop nombreux ou difficiles à lire, le comportement peut être décrit par une sémantique en langue naturelle. Cette description doit rester précise : elle doit indiquer le résultat attendu, l'effet produit et les conditions particulières éventuelles. Une simple reformulation du nom de l'opération ne constitue pas une sémantique suffisante.

2.2.4 Les préconditions

Une opération est totale lorsqu'elle est définie pour tous les éléments et tous les paramètres autorisés par sa signature. Elle est partielle lorsqu'elle n'est définie que dans certaines situations. Dans ce second cas, la condition qui doit être satisfaite avant l'appel est une précondition.

Par exemple, l'opération qui renvoie le premier élément d'une séquence ne peut être définie que si la séquence n'est pas vide. La précondition précise alors que l'opération doit être appelée uniquement sur une séquence non vide. Elle fait partie de la définition du TAD et ne doit pas être laissée à la seule connaissance de l'implantation.

La distinction entre une opération totale et une opération partielle permet de décrire explicitement les cas dans lesquels le TAD garantit un résultat. Elle évite également de confondre un

résultat incorrect avec une utilisation de l'opération en dehors du domaine dans lequel elle est définie.

2.2.5 Un formalisme de spécification

Pour représenter une spécification de manière homogène, on utilise un formalisme qui rassemble les différents constituants du TAD. Une spécification commence par le nom du TAD. Elle indique ensuite ses paramètres éventuels et les types dont il dépend, puis énumère ses opérations. Elle se termine par la description du comportement des opérations et, si nécessaire, par la liste de leurs préconditions.

Le formalisme retenu dans ce polycopié rassemble ces informations sous une forme structurée. Il ne constitue pas un langage de programmation : il sert à présenter une définition mathématique. La représentation obtenue doit donc rester indépendante de toute structure de données ou organisation mémoire particulière. Ce formalisme s'inspire de celui présenté par Bertrand Meyer dans son livre *Conception et programmation orientées objet*.

2.2.6 Le formalisme axiomatique

Dans sa forme la plus formelle, la spécification décrit le comportement des opérations à l'aide d'axiomes. Un axiome exprime une propriété logique du TAD, généralement sous la forme d'une égalité entre des expressions construites à partir de ses opérations.

La structure générale d'une spécification axiomatique peut être représentée de la manière suivante :

Nom: *Nom du TAD*
Paramètre: *Paramètres éventuels du TAD*
Utilise: *Types utilisés par le TAD*
Opérations: *operation*: $Type_1 \times Type_2 \rightarrow Type_3$

operationPartielle: $Type_4 \nrightarrow Type_5$

Axiomes: - *Une propriété logique du TAD*

Préconditions: *operationPartielle*: *Condition à respecter*

La partie consacrée aux opérations donne les signatures des opérations. Le symbole de flèche indique les types des paramètres d'entrée et du résultat. Une opération accompagnée d'une précondition est signalée séparément comme une opération partielle. La partie consacrée aux axiomes définit ensuite le comportement attendu, indépendamment de la manière dont les opérations seront implantées.

Les axiomes permettent de raisonner sur des expressions abstraites. Pour vérifier une implantation, il faudra montrer que les opérations programmées respectent les propriétés exprimées par ces axiomes. Cette vérification porte sur le comportement visible du TAD et non sur la ressemblance entre l'implantation et une représentation particulière utilisée dans les axiomes.

2.2.7 Le formalisme avec sémantiques en langue naturelle

L'écriture d'axiomes n'est pas toujours la forme la plus lisible pour décrire une opération. Lorsque son comportement est complexe ou que les équations seraient trop nombreuses, les axiomes peuvent être remplacés par des sémantiques exprimées en langue naturelle.

La structure générale reste la même : le nom, les paramètres, les dépendances et les opérations sont décrits de la même manière. Seule la partie qui définit le comportement change. Chaque opération est alors accompagnée d'une description de son résultat et de ses effets au niveau de l'analyse.

Cette description doit être suffisamment précise pour permettre la conception de l'algorithme et la construction des tests. Elle doit notamment préciser les éléments utilisés, le résultat produit et les cas dans lesquels l'opération est définie. Les préconditions restent indiquées séparément, car elles définissent le domaine d'utilisation de l'opération.

On peut par exemple décrire un TAD à l'aide du modèle suivant :

Nom: *Nom du TAD*
Paramètre: *Type générique des éléments éventuellement manipulés*
Utilise: *Types utilisés par le TAD*
Opérations: *operation*: $Type_1 \times Type_2 \rightarrow Type_3$
operationPartielle: $Type_4 \nrightarrow Type_5$
Sémantiques: *operation*: *Description précise du résultat produit par l'opération*
operationPartielle: *Description du résultat lorsque la précondition est respectée*
Préconditions: *operationPartielle*: *Condition à respecter avant l'appel*

Dans cet exemple, la partie consacrée aux opérations indique les paramètres d'entrée et le résultat de chaque opération. La partie consacrée aux sémantiques précise ensuite ce que produit chaque opération. La précondition indique enfin dans quel cas l'opération partielle peut être utilisée. Le modèle reste abstrait : aucune représentation des éléments ni aucun algorithme d'implantation n'y apparaît.

Les deux formalismes sont donc complémentaires. Les axiomes offrent une description concise et directement exploitable pour un raisonnement mathématique. Les sémantiques en langue naturelle permettent de conserver une spécification rigoureuse lorsque la formulation axiomatique serait trop lourde.

2.3 Les TAD élémentaires

Les types élémentaires utilisés dans un algorithme peuvent eux aussi être décrits comme des TAD. Cette description permet de préciser les éléments du type, les opérations qui permettent de les construire et les propriétés que ces opérations doivent respecter. Les exemples suivants illustrent plusieurs niveaux de formalisation, depuis une définition axiomatique jusqu'à une description des notations utilisées pour représenter les éléments.

2.3.1 Le TAD booléen

Le TAD booléen possède deux éléments, `vrai` et `faux`. Les opérations `et`, `ou` et `non` permettent de construire d'autres éléments booléens à partir de ceux qui existent déjà. Sa spécification peut être donnée à l'aide des axiomes suivants :

Nom: **Booleen**
Opérations: vrai: $\rightarrow$ **Booleen**
faux: $\rightarrow$ **Booleen**
non: **Booleen** $\rightarrow$ **Booleen**
et: **Booleen** $\times$ **Booleen** $\rightarrow$ **Booleen**
ou: **Booleen** $\times$ **Booleen** $\rightarrow$ **Booleen**
Axiomes:
- $non(vrai()) = faux()$
- $non(faux()) = vrai()$
- $non(non(b)) = b$
- $et(b, vrai()) = b$

- $et(b, faux()) = faux()$
- $et(b_1, b_2) = et(b_2, b_1)$
- $ou(b_1, b_2) = non(et(non(b_2), non(b_1)))$

Les opérations `vrai` et `faux` sont des opérations de désignation : elles permettent d'obtenir les deux éléments du TAD. Les autres opérations sont des opérations de manipulation. Les axiomes décrivent leur comportement sans faire référence à une représentation particulière, comme un entier ou un emplacement mémoire.

2.3.2 Le TAD naturel

Le TAD naturel permet de représenter les entiers naturels. Une première spécification, très mathématique, utilise l'élément 0 et l'opération `succ`, qui associe à chaque naturel son successeur. Les autres opérations peuvent alors être définies par leurs propriétés.

Nom: **Naturel**

Utilise: **Booleen**

Opérations:

0:	$\rightarrow$	**Naturel**
succ:	**Naturel** $\rightarrow$	**Naturel**
+:	**Naturel** $\times$ **Naturel** $\rightarrow$	**Naturel**
-:	**Naturel** $\times$ **Naturel** $\nrightarrow$	**Naturel**
*:	**Naturel** $\times$ **Naturel** $\rightarrow$	**Naturel**
div:	**Naturel** $\times$ **Naturel** $\nrightarrow$	**Naturel**
<:	**Naturel** $\times$ **Naturel** $\rightarrow$	**Booleen**

Axiomes:

- $succ(a) \neq 0$
- $succ(a) = succ(b) \Rightarrow a = b$
- $a + b = b + a$
- $a + 0 = a$
- $a + succ(b) = succ(a + b)$
- $a * b = b * a$
- $a * 0 = 0$
- $a * succ(b) = a + (a * b)$

Préconditions:

- a-b: $b < a$ ou $a = b$
- a div b: $b \neq 0$

Dans cette première version, les naturels sont engendrés par 0 et par `succ`. Les axiomes de l'addition et de la multiplication donnent leur comportement par rapport à cette construction. La soustraction est partielle dans les naturels, car le résultat n'est défini que lorsque le second opérande ne dépasse pas le premier. La division entière est également partielle, car son diviseur doit être différent de zéro.

Une deuxième version utilise directement les écritures décimales des naturels comme opérations de désignation. Au lieu de construire chaque naturel par des applications successives de `succ` à 0, on dispose d'une opération sans paramètre pour chaque naturel : 0, 1, 2, etc. Les points de suspension dans la spécification suivante indiquent que cette énumération se poursuit.

Nom: **Naturel**

Utilise: **Booleen**,**Reel**,**Entier**

Opérations:

0:	$\rightarrow$	**Naturel**
1:	$\rightarrow$	**Naturel**
2:	$\rightarrow$	**Naturel**
	...	
+:	**Naturel** $\times$ **Naturel** $\rightarrow$	**Naturel**
-:	**Naturel** $\times$ **Naturel** $\rightarrow$	**Entier**
*:	**Naturel** $\times$ **Naturel** $\rightarrow$	**Naturel**

	div:	$\textbf{Naturel} \times \textbf{Naturel} \nrightarrow \textbf{Naturel}$
	/:	$\textbf{Naturel} \times \textbf{Naturel} \nrightarrow \textbf{Reel}$
	<:	$\textbf{Naturel} \times \textbf{Naturel} \rightarrow \textbf{Booleen}$
Préconditions:	a/b:	$b \neq 0$
	a div b:	$b \neq 0$

Cette version fait également évoluer les résultats de certaines opérations. La soustraction produit désormais un entier : elle est donc définie même lorsque le second opérande dépasse le premier. Par exemple, $2 - 3$ produit l'entier -1. À la division entière `div` s'ajoute la division `/`, dont le résultat est un réel. Ainsi, $5\,div\,2$ vaut 2, tandis que $5/2$ vaut 2,5. Les deux divisions conservent la précondition d'un diviseur non nul. Ces choix expliquent les dépendances supplémentaires envers les TAD entier et réel. Les opérations arithmétiques sont ici comprises dans leur sens usuel, sans reprendre leur définition axiomatique.

La troisième version conserve ces opérations et leurs résultats, mais remplace l'énumération des opérations de désignation par une expression rationnelle, également appelée expression régulière. L'expression `0|[1-9][0-9]*` décrit les noms de toutes ces opérations en une seule ligne :

Nom:	**Naturel**	
Utilise:	**Booleen**,**Reel**,**Entier**	
Opérations:	`0|[1-9][0-9]*`:	$\rightarrow \textbf{Naturel}$
	+:	$\textbf{Naturel} \times \textbf{Naturel} \rightarrow \textbf{Naturel}$
	-:	$\textbf{Naturel} \times \textbf{Naturel} \rightarrow \textbf{Entier}$
	*:	$\textbf{Naturel} \times \textbf{Naturel} \rightarrow \textbf{Naturel}$
	div:	$\textbf{Naturel} \times \textbf{Naturel} \nrightarrow \textbf{Naturel}$
	/:	$\textbf{Naturel} \times \textbf{Naturel} \nrightarrow \textbf{Reel}$
	<:	$\textbf{Naturel} \times \textbf{Naturel} \rightarrow \textbf{Booleen}$
Préconditions:	a/b:	$b \neq 0$
	a div b:	$b \neq 0$

La barre verticale exprime une alternative : le nom est soit `0`, soit un chiffre de `1` à `9`, suivi de zéro ou plusieurs chiffres de `0` à `9`. L'étoile indique cette répétition éventuellement vide. Les écritures `0`, `7` et `123` sont donc autorisées, contrairement à `007`. Cette notation désigne une famille d'opérations sans paramètre, et non une opération qui recevrait une chaîne de caractères. Elle rend la présentation plus compacte sans changer le comportement de la deuxième version.

2.3.3 Le TAD réel

Le TAD réel peut être défini à partir des écritures décimales comportant une partie entière et une partie fractionnaire. Il dépend notamment du TAD booléen pour les comparaisons.

Nom:	**Reel**	
Utilise:	**Booleen**	
Opérations:	`(0|[1-9][0-9]*),[0-9]+`:	$\rightarrow \textbf{Reel}$
	+:	$\textbf{Reel} \times \textbf{Reel} \rightarrow \textbf{Reel}$
	-:	$\textbf{Reel} \times \textbf{Reel} \rightarrow \textbf{Reel}$
	*:	$\textbf{Reel} \times \textbf{Reel} \rightarrow \textbf{Reel}$
	/:	$\textbf{Reel} \times \textbf{Reel} \nrightarrow \textbf{Reel}$
	<:	$\textbf{Reel} \times \textbf{Reel} \rightarrow \textbf{Booleen}$
Préconditions:	a/b:	$b \neq 0$

La division est une opération partielle du TAD réel : elle n'est définie que si son second opérande est non nul. Cette précondition exprime une propriété du type mathématique et ne dépend pas du message d'erreur ou du mécanisme qui serait choisi dans une implantation.

2.3.4 Le TAD caractère

Le TAD caractère associe les caractères aux naturels. L'opération `car` construit un caractère à partir de son code, et l'opération `ord` fournit le code d'un caractère. Les opérations `pred` et `succ` permettent d'obtenir les caractères précédents et suivants.

Nom: **Caractere**
Utilise: **Naturel**
Opérations:
- '.': $\rightarrow$ **Caractere**
- car: **Naturel** $\rightarrow$ **Caractere**
- pred: **Caractere** $\nrightarrow$ **Caractere**
- succ: **Caractere** $\rightarrow$ **Caractere**
- ord: **Caractere** $\rightarrow$ **Naturel**

Axiomes:
- $car(ord(c)) = c$
- $pred(c) = car(ord(c) - 1)$
- $succ(c) = car(ord(c) + 1)$

Préconditions: pred(c): $ord(c) > 1$

Le point utilisé dans l'opération de construction représente un caractère quelconque. L'opération `pred` est partielle, car le caractère qui précède le premier caractère représentable n'existe pas dans le TAD.

2.3.5 Le TAD chaîne

Le TAD chaîne dépend des TAD caractère et naturel. Il possède une chaîne vide, une opération de conversion d'un caractère en chaîne, une concaténation, la longueur d'une chaîne et l'accès à un caractère d'indice donné.

Nom: **Chaine de caracteres**
Utilise: **Caractere**,**Naturel**
Opérations:
- "": $\rightarrow$ **Chaine de caracteres**
- caractereEnChaine: **Caractere** $\rightarrow$ **Chaine de caracteres**
- +: **Chaine de caracteres** $\times$ **Chaine de caracteres** $\rightarrow$ **Chaine de caracteres**
- longueur: **Chaine de caracteres** $\rightarrow$ **Naturel**
- iemeCaractere: **Chaine de caracteres** $\times$ **Naturel** $\nrightarrow$ **Caractere**

Axiomes:
- $longueur("") = 0$
- $longueur(ch1 + caractereEnChaine(c)) = longueur(ch1) + 1$
- $iemeCaractere(ch + caractereEnChaine(c), longueur(ch) + 1) = c$

Préconditions: iemeCaractere(ch,i): $0 < i \leq longueur(ch)$

Les axiomes indiquent notamment que la longueur de la chaîne vide est nulle et que l'ajout d'un caractère augmente la longueur d'une unité. L'opération `iemeCaractere` n'est définie que pour un indice compris entre 1 et la longueur de la chaîne. La spécification ne décrit pas la représentation concrète de la chaîne : elle ne dit pas si celle-ci sera implantée par un tableau, une zone mémoire allouée dynamiquement ou une structure chaînée.

2.4 Un exemple : le TAD Point2D

Le point du plan permet d'illustrer le passage de l'analyse à la conception sur un exemple simple. On souhaite construire un point à partir de deux coordonnées et pouvoir consulter son abscisse et son ordonnée. Ces services peuvent être définis sans décider comment les coordonnées seront conservées en mémoire. Nous allons d'abord spécifier le TAD, puis examiner deux représentations possibles et leur utilisation.

2.4.1 L'analyse : définir les opérations du point

Le TAD Point2D dépend du TAD réel. L'opération `point2D` construit un point à partir d'une abscisse et d'une ordonnée. Les deux opérations d'observation permettent de retrouver ces coordonnées. Leur comportement est précisé par deux axiomes, valables pour tous réels a et b.

Nom: Point2D
Utilise: **Reel**
Opérations: point2D: **Reel** × **Reel** → Point2D
abscisse: Point2D → **Reel**
ordonnée: Point2D → **Reel**
Axiomes: - $abscisse(point2D(a, b)) = a$
- ordonnée$(point2D(a, b)) = b$

Ces axiomes relient la construction du point à son observation : quel que soit le choix de représentation, consulter l'abscisse du point construit à partir de a et b doit rendre a, et consulter son ordonnée doit rendre b. Aucune précondition n'est nécessaire pour ces trois opérations.

2.4.2 La conception préliminaire : fixer les signatures

Dans le cadre de la programmation structurée, les trois opérations sont traduites en fonctions. On nomme leurs paramètres et on précise les types des résultats :
— **fonction** point2D (lAbscisse,lOrdonnee : **Reel**) **:** Point2D
— **fonction** abscisse (lePoint : Point2D) **: Reel**
— **fonction** ordonnee (lePoint : Point2D) **: Reel**

Ces signatures suffisent pour écrire des algorithmes qui utilisent le point. Elles ne donnent pas encore le corps des fonctions. Selon le paradigme de programmation retenu, la conception préliminaire peut faire apparaître des opérations supplémentaires ; dans cet exemple, les trois fonctions ci-dessus correspondent directement aux opérations du TAD.

2.4.3 La conception détaillée : choisir une représentation

Une première solution consiste à conserver les coordonnées cartésiennes dans un enregistrement. Le champ `x` contient l'abscisse et le champ `y` contient l'ordonnée. La construction du point affecte les deux paramètres à ces champs ; les fonctions d'observation lisent ensuite le champ correspondant.

```
Type Point2D = Structure
    x : Reel
    y : Reel
finstructure
fonction point2D (x,y : Reel) : Point2D
    Déclaration  resultat : Point2D
debut
    resultat.x ← x
    resultat.y ← y
    retourner resultat
fin
fonction abscisse (p : Point2D) : Reel
debut
    retourner p.x
fin
fonction ordonnee (p : Point2D) : Reel
debut
    retourner p.y
```

fin

La fonction `ordonnee`, analogue à `abscisse`, complète ici les algorithmes présentés dans le support. Les deux axiomes se vérifient directement : le constructeur place les coordonnées dans les champs que les fonctions d'observation renvoient.

Une autre solution consiste à conserver des coordonnées polaires. Un point est alors représenté par sa distance à l'origine, appelée module, et par un angle, appelé argument. Les coordonnées cartésiennes sont calculées à partir de ces deux informations. Le support propose l'esquisse suivante :

Type Point2D = **Structure**
  module : **Reel**
  argument : **Reel**
finstructure
fonction point2D (x,y : **Reel**) **:** Point2D

  Déclaration resultat : Point2D

debut
  resultat.module ← sqrt($x^2 + y^2$)
  ...
  retourner resultat
fin
fonction abscisse (p : Point2D) **: Reel**
debut
  retourner p.module ∗ cosinus(p.argument)
fin

La fonction `sqrt` calcule la racine carrée. Les points de suspension correspondent notamment au calcul et à l'affectation de l'argument. Cette esquisse n'est donc pas un constructeur complet. Pour la compléter, il faudrait préciser le choix de l'angle selon la position du point et adopter une convention à l'origine, où le module est nul et où aucun angle n'est privilégié. L'ordonnée s'obtiendrait de manière analogue à l'abscisse, en utilisant le sinus de l'argument.

Ces deux représentations sont des solutions alternatives pour le même TAD. Dans la première, l'abscisse est stockée directement ; dans la seconde, elle est calculée. Les signatures et les résultats attendus restent les mêmes. C'est le respect des axiomes qui permet de considérer ces représentations comme des réalisations du même type abstrait.

2.4.4 Utiliser le point sans connaître sa représentation

La procédure suivante saisit deux réels, puis appelle le constructeur pour obtenir un point. Elle ne crée pas elle-même un enregistrement et ne remplit aucun de ses champs.

procédure saisirPoint2D (**S** pt : Point2D)

  Déclaration x,y : **Reel**

debut
  ecrire("Abscisse :")
  lire(x)
  ecrire("Ordonnée :")
  lire(y)
  pt ← point2D(x,y)
fin

On peut aussi représenter le point par une chaîne de la forme `(abscisse,ordonnée)` en utilisant uniquement les fonctions d'observation. La fonction `reelEnChaine` convertit un réel en chaîne, et l'opération `+` concatène les chaînes obtenues.

fonction point2DEnChaine (p : Point2D) **: Chaine de caracteres**
debut

```
    retourner "(" + reelEnChaine(abscisse(p)) + "," +
      reelEnChaine(ordonnee(p)) + ")"
fin
```

Ces deux sous-programmes restent utilisables avec chacune des représentations précédentes, dès lors que les opérations du TAD respectent leur spécification. Un accès direct à `p.x`, en revanche, dépendrait de la représentation cartésienne et devrait être revu avec la représentation polaire. L'exemple met ainsi en évidence la séparation entre l'utilisation d'un TAD et sa réalisation, sur laquelle reposent les règles de conception qui suivent.

2.5 Les règles de conception

L'exemple du point relie la spécification abstraite aux sous-programmes utilisés dans les algorithmes. Cette démarche demande de vérifier la qualité de la spécification, puis de conserver son sens lors des choix de conception. Les règles suivantes précisent les points auxquels il faut être attentif à chaque étape.

2.5.1 Lors de l'analyse : une spécification complète et consistante

La complétude consiste à vérifier que les opérations proposées suffisent pour décrire les propriétés du type abstrait et permettre les utilisations prévues. Il ne s'agit pas d'ajouter toutes les opérations imaginables, mais de s'assurer que les services nécessaires sont disponibles. Dans l'exemple Point2D, la construction d'un point et la consultation de ses deux coordonnées permettent notamment de réaliser la saisie et la conversion en chaîne présentées précédemment. Si l'une des coordonnées ne pouvait pas être consultée, cette dernière utilisation ne serait plus possible avec la seule interface du TAD.

La consistance concerne la cohérence des propriétés annoncées. Certains TAD possèdent des invariants, c'est-à-dire des propriétés qui doivent être respectées par tous leurs éléments. Les opérations, y compris lorsqu'elles sont composées, doivent rester compatibles avec ces propriétés. Une spécification ne doit donc pas imposer des résultats contradictoires ni permettre, dans les conditions d'utilisation prévues, de produire un élément qui viole un invariant du type.

Les axiomes de Point2D donnent un exemple de propriétés à préserver : observer un point après sa construction doit restituer les coordonnées fournies au constructeur. Examiner les compositions d'opérations permet ainsi de vérifier que les services proposés s'accordent entre eux, au-delà de la seule lecture de leurs signatures.

Pour rendre cette vérification possible, il faut choisir des identifiants significatifs et accompagner les opérations d'axiomes ou de sémantiques précises. Ces descriptions serviront également à construire les tests unitaires. Par exemple, les axiomes de Point2D conduisent à construire un point à partir de coordonnées connues, puis à comparer les résultats des opérations d'observation aux valeurs attendues.

Enfin, les préconditions doivent être explicites. Elles délimitent les cas dans lesquels le comportement annoncé est garanti. La division des naturels requiert ainsi un diviseur non nul. Une signature correcte ne suffit pas à garantir qu'un appel respecte cette condition : celle-ci doit être prise en compte dans l'algorithme qui utilise l'opération.

2.5.2 Lors de la conception préliminaire : fonctions et procédures

Dans le paradigme de la programmation structurée, les opérations des TAD deviennent des fonctions ou des procédures qui manipulent des variables du type considéré. On introduit alors une distinction absente de la spécification mathématique : une variable possède un état, et certains sous-programmes peuvent modifier cet état.

Pour rappel dans le paradigme de la programmation structurée, une fonction calcule et renvoie une ou plusieurs valeurs à partir de ses paramètres, sans modifier l'état du programme. Une procédure quant à elle modifie au contraire l'état du programme, donc une ou plusieurs variables.

Les variables concernées apparaissent dans sa signature comme paramètres de sortie ou d'entrée-sortie [1].

Pour un TAD, on utilise donc une fonction pour observer une valeur ou en construire une nouvelle, et une procédure pour modifier une variable existante. Les fonctions `abscisse` et `ordonnee` illustrent le premier cas. La procédure `saisirPoint2D` illustre le second : elle fournit le point saisi dans son paramètre de sortie. Une fonction de construction telle que `point2D` produit, quant à elle, une valeur nouvelle sans modifier une variable reçue en paramètre.

Le choix des sous-programmes doit aussi tenir compte de l'efficacité de la future implantation. Un type immuable impose de produire une nouvelle valeur lorsqu'on souhaite représenter une donnée différente. Un type muable permet de modifier une variable existante. Le support préconise de privilégier cette seconde possibilité lorsqu'elle permet d'éviter des reconstructions ou des copies coûteuses. Le choix doit néanmoins être apprécié en fonction de la représentation et des opérations nécessaires.

Pour Point2D, la conception préliminaire peut ainsi ajouter des procédures permettant de modifier séparément l'abscisse et l'ordonnée. Modifier l'abscisse doit conserver l'ordonnée, et réciproquement. Le point concerné serait alors un paramètre d'entrée-sortie : son état initial est utilisé et son état final est modifié. Ces procédures complètent les services de construction et d'observation pour permettre une utilisation muable du type. Elles ne doivent pas imposer à l'appelant de connaître ses champs internes.

Les préconditions établies lors de l'analyse doivent être conservées dans les spécifications des fonctions et des procédures correspondantes. Le passage à des sous-programmes ne supprime pas les restrictions du domaine de définition des opérations.

2.5.3 Lors de la conception détaillée : identifier les opérations utilisées

Un algorithme peut utiliser plusieurs TAD qui proposent des opérations de même nom. Si un appel est ambigu, on préfixe le nom de la fonction ou de la procédure par celui du TAD, en séparant les deux noms par un point. Pour le constructeur de Point2D, cette notation donne :

```
pt : Point2D
...
pt ← Point2D.point2D(x,y)
```

Le préfixe indique ici à quel TAD appartient l'opération appelée. Il ne désigne pas un champ d'enregistrement : `Point2D` est le nom du type, tandis que `pt` est une variable de ce type.

Cette qualification précise l'interface utilisée sans révéler sa réalisation. L'appelant continue à employer les opérations du TAD ; seuls les algorithmes qui les réalisent dépendent de la représentation choisie. La traduction en C devra conserver cette séparation entre les services proposés et leur implantation.

2.6 L'implantation d'un TAD en C

Le développement traduit les choix de conception dans un langage de programmation. En C, cette traduction s'appuie sur les types composés, les fonctions et la séparation entre fichiers d'entête et fichiers sources. Elle doit préserver le comportement spécifié tout en tenant compte des possibilités du langage. Certaines décisions supplémentaires concernent notamment les noms des opérations, la gestion des erreurs et la mémoire.

2.6.1 Uniquement des fonctions

En C, les fonctions et les procédures du formalisme algorithmique sont toutes réalisées par des fonctions du langage. Une procédure qui ne renvoie aucun résultat utilise le type de retour

1. Dans une architecture n-tiers, les procédures de la partie interface homme-machine (IHM) ou interface machine-machine (IMM) peuvent utiliser l'entrée et la sortie standards sans que celles-ci apparaissent explicitement dans leur signature. Ces échanges avec l'extérieur restent des effets de la procédure.

`void`. Pour modifier une variable de l'appelant, elle reçoit généralement son adresse : modifier une simple copie passée par valeur ne modifierait pas la variable initiale. Les paramètres de sortie et d'entrée-sortie conduisent ainsi à employer des pointeurs.

2.6.2 Séparer l'interface et l'implantation

Le fichier d'en-tête, d'extension *.h*, déclare l'interface de programmation, ou API, du TAD. Il rassemble les constantes, les types et les prototypes des fonctions nécessaires à son utilisation. Un fichier source qui utilise le TAD inclut cet en-tête pour que le compilateur connaisse ces déclarations et puisse vérifier les appels.

Le fichier *.c* associé définit les fonctions annoncées dans l'en-tête. Il peut également contenir des constantes, des types et des fonctions auxiliaires destinés à la seule implantation.

Dans la forme d'implantation retenue pour Point2D, la définition de la structure figure dans le *.h*. Le compilateur doit en effet connaître sa taille et sa composition pour permettre au code utilisateur de déclarer des variables de ce type et de les manipuler par valeur. La représentation, pourtant privée au niveau de l'analyse et de la conception, devient donc visible dans l'en-tête. Il revient à l'utilisateur de ne pas y accéder directement, mais de se limiter aux fonctions fournies. Accéder directement à un champ crée une dépendance envers la représentation et rend plus difficile son remplacement.

2.6.3 Adapter l'interface aux possibilités du C

Le C ne fournit pas d'espaces de nommage permettant de regrouper les noms publics sous celui d'un TAD. On préfixe donc les identifiants publics pour éviter les collisions entre modules. Une bonne pratique est de mettre en préfixe le nom du TAD ou une abréviation de ce nom., suivi d'un *underscore* suivi des identifivcnats des types, fonctions ou constantes. Pour Point2D, le préfixe serait ainsi `PT_`.

La généricité demande également une adaptation. Un TAD de collection peut être défini abstraitement à partir d'un type d'élément paramètre, mais cette description ne se traduit pas directement par un type générique en C. Il y a alors deux possibilités : développer une version de la collection pour chaque type d'élément nécessaire, ou utiliser des pointeurs `void*` et des pointeurs de fonction. Dans ce second cas, le pointeur générique permet de désigner les données et les fonctions fournies précisent les opérations à effectuer sur elles. Ce choix impose de veiller à la cohérence des types manipulés.

La gestion des erreurs peut enfin conduire à adapter les signatures. Une fonction peut signaler une erreur par sa valeur de retour ou, selon la convention retenue, au moyen d'`errno`, déclaré dans *errno.h*. Ce mécanisme n'est pas automatique : l'interface doit indiquer comment détecter une erreur et interpréter les informations fournies. Si la valeur de retour est réservée à un code d'erreur, le résultat du calcul peut, par exemple, être transmis par un paramètre de sortie. Ces conventions doivent être documentées dans l'en-tête. Elles ne remplacent pas les préconditions, qui continuent à définir les conditions d'utilisation des opérations.

2.6.4 Prévoir la libération, la copie et l'égalité

Le C ne possède pas de ramasse-miettes assurant automatiquement la récupération de la mémoire allouée dynamiquement. Lorsqu'une implantation effectue de telles allocations, l'interface doit prévoir les opérations nécessaires à leur libération et préciser qui en est responsable.

L'affectation demande aussi une attention particulière. Le langage ne permet pas de redéfinir l'opérateur `=` pour lui donner un comportement propre au TAD. Pour une structure, l'affectation copie les valeurs de ses membres. Si la structure contient des pointeurs, les adresses sont copiées, sans dupliquer les objets désignés : on obtient une copie en surface. Les deux structures peuvent alors partager les mêmes données. Une copie en profondeur nécessite une opération spécifique qui duplique aussi les données désignées, conformément aux besoins du TAD.

Enfin, l'égalité doit traduire le sens donné par la spécification. L'opérateur == appliqué à des pointeurs compare leurs adresses, et non le contenu des objets désignés. De plus, il ne permet pas de comparer directement deux structures. Si l'utilisation du TAD nécessite de tester l'égalité entre deux valeurs, il faut donc proposer une fonction adaptée. Pour un point, comparer les coordonnées répond à une question différente de celle qui consiste à déterminer si deux pointeurs désignent le même objet.

Ces adaptations montrent pourquoi le développement ne se réduit pas à une traduction syntaxique des algorithmes. Les choix propres au langage doivent être explicités dans l'interface et sa documentation pour que l'utilisateur du TAD puisse respecter ses conditions d'emploi. La documentation et les tests unitaires permettent ensuite de rendre ce contrat accessible et d'en vérifier le respect.

2.6.5 Traduire l'exemple Point2D

Le fichier *point2D.h* utilise une structure à deux champs, conformément à la représentation cartésienne. Le type abstrait réel est ici traduit par `float` : il s'agit d'une représentation approchée des réels, avec les limites de précision et d'étendue des nombres flottants. Le préfixe `PT_` identifie les noms publics associés au point.

```
#ifndef __POINT_2D__
#define __POINT_2D__

typedef struct {
  float x;
  float y;
} PT_Point2D;

PT_Point2D PT_point2D(float x, float y);
float PT_abscisse(PT_Point2D pt);
float PT_ordonnee(PT_Point2D pt);

#endif
```

La garde d'inclusion évite de traiter plusieurs fois les mêmes déclarations dans une unité de compilation. Le nom `__POINT_2D__` est conservé ici pour reproduire le fichier du support, mais les identifiants commençant par deux traits de soulignement sont réservés à l'implantation du langage C. Pour un nouveau fichier, on choisirait par exemple `PT_POINT2D_H`.

Le fichier *point2D.c* définit les trois fonctions. Il inclut son propre en-tête, ce qui permet au compilateur de vérifier la cohérence entre les prototypes publics et les définitions.

```
#include "point2D.h"

PT_Point2D PT_point2D(float x, float y) {
  PT_Point2D pt;
  pt.x = x;
  pt.y = y;
  return pt;
}

float PT_abscisse(PT_Point2D pt) {
  return pt.x;
}

float PT_ordonnee(PT_Point2D pt) {
  return pt.y;
}
```

Le constructeur initialise une variable locale puis renvoie sa valeur. Il ne renvoie pas l'adresse de cette variable : la fin de sa durée de vie ne rend donc pas le résultat invalide. Les fonctions

d'observation reçoivent également le point par valeur et renvoient le champ correspondant. Elles traduisent directement les algorithmes de la représentation cartésienne.

2.7 Documentation et tests unitaires

Une implantation doit pouvoir être utilisée sans obliger son utilisateur à lire les signatures de toutes les fonctions. Sa documentation expose les services disponibles et leurs conditions d'emploi. Les tests unitaires complètent ce travail en confrontant les résultats de l'implantation aux propriétés attendues. Documentation et tests s'appuient ainsi sur la même référence : la spécification du TAD.

2.7.1 Documenter l'interface avec Doxygen

Lorsque plusieurs TAD interviennent dans un programme, la documentation permet de retrouver le rôle des types, des paramètres et des résultats. Elle doit également préciser les préconditions et les conventions de gestion des erreurs. Une signature indique les types attendus, mais ne dit pas à elle seule ce que la fonction garantit.

Pour rapprocher cette documentation du code qu'elle décrit, on peut l'intégrer aux fichiers sources sous forme de commentaires structurés. Un outil extrait ces commentaires et produit des documents consultables dans différents formats. Doxygen permet d'appliquer cette démarche au C, ainsi qu'à d'autres langages. Il peut notamment produire des pages HTML, des pages de manuel et des fichiers LaTeX permettant d'obtenir un PDF.

La figure 2.1 représente cette chaîne de génération. Doxygen lit les sources et un fichier de configuration, généralement nommé *Doxyfile*. L'outil *doxywizard* aide à préparer cette configuration. Les pages HTML peuvent être consultées directement, tandis que la production d'un PDF passe par la compilation des fichiers LaTeX générés. Les autres branches illustrent des possibilités d'export ou de traitement supplémentaires.

Les commentaires de documentation peuvent être délimités par `/**` et `*/`. Des commandes introduites par un *backslash* précisent la nature des informations. Les principales commandes utiles à une interface C se regroupent ainsi :

Éléments documentés. `\file`, `\fn`, `\struct`, `\union`, `\enum`, `\typedef`, `\var` et `\def` identifient respectivement un fichier, une fonction, une structure, une union, une énumération, un nom de type, une variable et une macro.

Contrat d'utilisation. `\brief` donne une description courte, `\param` décrit un paramètre et `\return` le résultat. Les commandes `\pre` et `\post` expriment les préconditions et les propriétés garanties après l'appel. `\attention` attire l'attention sur une condition d'emploi particulière.

Identification. `\author`, `\date` et `\version` fournissent des informations sur l'auteur et la version documentée.

Voici une version documentée de l'en-tête de Point2D. Le premier commentaire identifie le fichier ; les suivants décrivent les trois fonctions publiques.

```
/**
 * \file point2D.h
 * \brief Implantation du TAD Point2D
 * \author N. Delestre
 * \version 1.0
 * \date 15/12/2013
 *
 */


#ifndef __POINT_2D__
#define __POINT_2D__
```

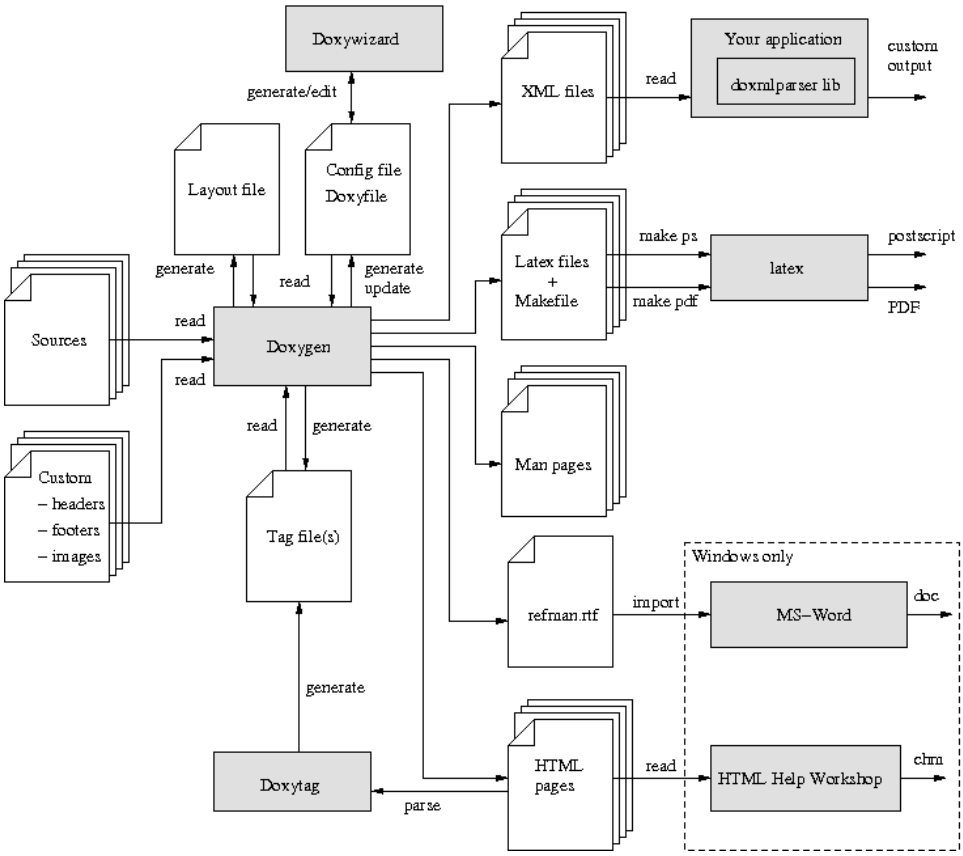


FIGURE 2.1 – Organisation de la génération de documentation avec Doxygen.

```
typedef struct {
  float x;
  float y;
} PT_Point2D;

/**
 * \fn PT_Point2D PT_point2D(float x, float y)
 * \brief Création d'un Point2D à partir de son abscisse et de son ordonnée
 * \param x l'abscisse du point
 * \param y l'ordonnée du point
 * \return PT_Point2D
 */
PT_Point2D PT_point2D(float x, float y);

/**
 * \fn float PT_abscisse(PT_Point2D pt)
 * \brief L'abscisse d'un point Point2D
 * \param pt le point
 * \return float
 */
float PT_abscisse(PT_Point2D pt);

/**
 * \fn float PT_ordonnee(PT_Point2D pt)
 * \brief L'ordonnée d'un point Point2D
 * \param pt le point
 * \return float
 */
float PT_ordonnee(PT_Point2D pt);

#endif
```

Ces commentaires ne modifient pas le comportement du code. Ils fournissent une première description de l'interface, que l'on peut enrichir pour rendre le contrat plus précis. La génération automatique organise les informations écrites ; elle ne garantit ni leur exactitude ni leur complétude. La documentation doit donc évoluer avec l'interface.

2.7.2 Vérifier le comportement par des tests unitaires

Un test unitaire exerce une unité limitée du programme, par exemple une opération du TAD, et compare son comportement à un résultat attendu. Il permet de rechercher des écarts entre l'implantation et la spécification, ainsi que des défauts sur des cas particuliers. Deux approches se complètent.

Les tests en boîte noire sont construits à partir de l'interface et de sa spécification, sans utiliser la représentation interne. Les axiomes donnent alors des propriétés à vérifier sur des exemples. Pour Point2D, on peut construire un point avec deux coordonnées connues, puis vérifier que les fonctions d'observation les restituent.

Les tests en boîte blanche tiennent compte du code de l'implantation. Le choix des entrées vise notamment à exécuter ses différentes branches et à examiner les cas limites. Cette démarche devient difficile lorsque les chemins possibles sont nombreux. Exécuter chaque branche ne suffit d'ailleurs pas à démontrer l'absence d'erreur.

Dans les deux cas, un ensemble de tests réussis établit que les exemples exécutés donnent les résultats attendus ; il ne prouve pas que tous les appels possibles sont corrects. Les cas choisis doivent respecter les préconditions de l'opération, sauf si l'on vérifie un mécanisme de détection d'erreur explicitement prévu par l'interface.

2.7.3 Organiser les tests avec CUnit

Une bibliothèque de tests fournit les mécanismes pour enregistrer les tests, les exécuter et produire un bilan. CUnit remplit ce rôle en C. Il propose plusieurs modes d'exécution : *Automated* produit des résultats en XML sans interaction, *Basic* fournit un compte rendu textuel sans interaction, et les modes *Console* et *Curses* proposent des interfaces interactives. Nous utilisons ici le mode *Basic*.

Les tests sont organisés dans un registre qui contient une ou plusieurs suites. Une suite, désignée par une valeur de type `CU_pSuite`, regroupe des tests et peut être accompagnée d'une fonction d'initialisation et d'une fonction de nettoyage. Ces deux fonctions ont la signature `int f(void)` et renvoient zéro en cas de succès. Elles encadrent l'exécution de la suite, et non chaque test individuellement. Un test est une fonction de signature `void f(void)`.

À l'intérieur d'un test, les assertions CUnit enregistrent le succès ou l'échec des vérifications. On trouve notamment `CU_ASSERT_TRUE` et `CU_ASSERT_FALSE` pour les conditions booléennes. Les macros `CU_ASSERT_EQUAL` et `CU_ASSERT_NOT_EQUAL` comparent des valeurs. Pour les pointeurs, on utilise notamment `CU_ASSERT_PTR_EQUAL`. L'écriture d'un test suit quatre étapes :

1. préparer les paramètres de l'appel ;
2. déterminer le résultat attendu à partir de la spécification ;
3. appeler la fonction et récupérer le résultat obtenu ;
4. comparer le résultat obtenu au résultat attendu par une assertion.

Le résultat attendu ne doit pas être calculé en recopiant l'algorithme de la fonction testée : une même erreur pourrait alors se retrouver dans les deux calculs et passer inaperçue.

2.7.4 Tester les axiomes de Point2D

Les deux tests suivants vérifient les axiomes du point pour les coordonnées 1 et 2. Ils utilisent uniquement le constructeur et les fonctions d'observation, ce qui en fait des tests en boîte noire.

```
#include<stdio.h>
#include<CUnit/Basic.h>
#include<string.h>
#include"point2D.h"

void test_abscisse(void){
  PT_Point2D pt = PT_point2D(1,2);
  float resultatAttendu = 1;
  float resultatObtenu = PT_abscisse(pt);
  CU_ASSERT_EQUAL(resultatAttendu, resultatObtenu);
}

void test_ordonnee(void){
  PT_Point2D pt = PT_point2D(1,2);
  float resultatAttendu = 2;
  float resultatObtenu = PT_ordonnee(pt);
  CU_ASSERT_EQUAL(resultatAttendu, resultatObtenu);
}
```

Chaque test contient une assertion. Le premier contrôle l'abscisse du point construit et le second son ordonnée. Les valeurs 1 et 2 sont exactement représentables par le type `float`, et l'implantation cartésienne se contente de les copier : la comparaison exacte convient à ces tests. Pour des résultats issus de calculs flottants, il faudrait tenir compte des arrondis et choisir un critère de comparaison adapté.

Les fonctions suivantes préparent et terminent la suite. Aucune ressource particulière n'étant nécessaire ici, elles se limitent à signaler le succès.

```
int init_suite_success(void) {
  return 0;
}

int clean_suite_success(void) {
  return 0;
}
```

Le programme principal initialise ensuite le registre, crée une suite et lui ajoute les deux tests. Chaque échec d'enregistrement interrompt cette préparation ; le registre est nettoyé lorsqu'il a été initialisé.

```
int main(int argc, char** argv){
  CU_pSuite pSuite = NULL;

  /* initialisation du registre de tests */
  if (CU_initialize_registry() != CUE_SUCCESS)
    return CU_get_error();

  /* ajout d'une suite de test */
  pSuite = CU_add_suite("Tests boite noire", init_suite_success, clean_suite_success);
  if (pSuite == NULL) {
    CU_cleanup_registry();
    return CU_get_error();
  }

  /* Ajout des tests à la suite de tests boite noire */
  if ((CU_add_test(pSuite, "Test general de PT_abscisse", test_abscisse) == NULL)
      || (CU_add_test(pSuite, "Test general de PT_ordonnee", test_ordonnee) == NULL)
      ) {
      CU_cleanup_registry();
      return CU_get_error();
  }
```

Enfin, les tests sont exécutés en mode détaillé et les éventuels échecs sont affichés avant la libération du registre.

```
  /* Lancement des tests */
  CU_basic_set_mode(CU_BRM_VERBOSE);
  CU_basic_run_tests();
  printf("\n");
  CU_basic_show_failures(CU_get_failure_list());
  printf("\n\n");

  /* Nettoyage du registre */
  CU_cleanup_registry();
  return CU_get_error();
}
```

Avec l'implantation cartésienne présentée précédemment, le bilan attendu est une suite exécutée, deux tests réussis et deux assertions réussies. Ce bilan vérifie les deux axiomes sur le point choisi. D'autres coordonnées, notamment nulles ou négatives, permettraient d'étendre les cas examinés.

Il faut distinguer une erreur de fonctionnement de CUnit d'une assertion échouée. Le retour de `CU_get_error()` renseigne sur la première ; il ne suffit pas, à lui seul, pour faire échouer une commande dès qu'un test échoue. Pour intégrer ce programme à une vérification automatisée, il faudrait également consulter le nombre d'assertions échouées avant de nettoyer le registre et en tenir compte dans le code de retour du programme.

La documentation décrit ainsi ce que l'utilisateur peut attendre du TAD, et les tests vérifient des conséquences concrètes de ce contrat. Leur place dans les fichiers et les commandes de construction

du projet doit permettre de les mettre à jour et de les utiliser régulièrement.

2.8 L'organisation du projet

Un projet ne se limite pas aux fichiers qui implantent ses TAD. Il comprend également les programmes qui les utilisent, les tests et la documentation. Une organisation commune permet de retrouver ces éléments et d'identifier leur rôle. Elle facilite aussi l'automatisation des tâches de construction avec *make*, dont le fonctionnement a été présenté dans la section 1.7.

2.8.1 Répartir les fichiers selon leur rôle

L'arborescence suivante distingue les sources, les interfaces et les différents résultats de la construction du projet :

```
projet/
|-- bin/
|-- tests/
|-- lib/
|-- include/
|-- src/
|-- doc/
|-- Makefile
|-- readme
‘-- changelog
```

Le répertoire *include* rassemble les en-têtes qui déclarent les interfaces des modules. Le répertoire *src* contient les fichiers C qui définissent leurs fonctions, ainsi que les programmes principaux. Dans cette organisation, les sources des tests sont également placées dans *src* : pour Point2D, on y trouve ainsi *point2D.c* et *testPoint2D.c*, tandis que *point2D.h* se trouve dans *include*.

Les répertoires *bin* et *tests* accueillent respectivement les exécutables de l'application et ceux des tests unitaires. Le répertoire *lib* contient les bibliothèques construites à partir des modules. Enfin, *doc* reçoit la documentation générée, par exemple les pages HTML produites par Doxygen. Ces répertoires n'ont pas tous nécessairement un contenu dès le début du projet : un TAD testé isolément peut disposer d'un exécutable de test sans être encore utilisé dans une application.

Cette répartition distingue notamment les fichiers que l'on maintient des fichiers que les outils peuvent reconstruire. Pour corriger une description de fonction, on modifie son commentaire de documentation dans l'en-tête, puis on régénère la documentation. Une modification directe d'une page HTML générée serait perdue lors de la génération suivante.

2.8.2 Donner les informations nécessaires à l'utilisation du projet

Le fichier *readme* constitue le point d'entrée pour une personne qui découvre le projet. Il en présente l'objectif et indique comment le construire et l'utiliser. Il précise également les outils et bibliothèques nécessaires, les commandes pour lancer les tests et l'emplacement de la documentation. Dans le cas de Point2D, il doit notamment permettre de savoir que les tests utilisent CUnit et que la documentation est produite par Doxygen.

Le fichier *changelog* consigne les évolutions significatives du projet, telles qu'une modification de l'interface ou la correction d'une erreur. Il aide à comprendre ce qui a changé d'une version à l'autre. Ces deux fichiers complètent la documentation de l'API : ils décrivent l'utilisation et l'évolution du projet dans son ensemble, plutôt que le contrat de chaque fonction.

2.8.3 Automatiser la construction avec *make*

Le *Makefile* rassemble les règles nécessaires à la construction. Les cibles *all*, *tests* et *doc* donnent accès aux principales tâches. La cible *all* regroupe les éléments à construire par défaut, notamment le programme lorsqu'il existe. La cible *tests* construit les exécutables de tests, et la cible *doc* génère la documentation. Pour un projet limité au TAD Point2D, la construction par défaut peut ainsi regrouper les tests et la documentation.

Construire les tests et les exécuter sont deux actions distinctes. Une cible qui construit un exécutable ne vérifie pas à elle seule les assertions. Après la construction, on lance le programme de test pour obtenir le bilan des tests. Les commandes suivantes illustrent cette succession :

```
make tests
./tests/testPoint2D
make doc
```

Les règles doivent exprimer les dépendances réelles des fichiers produits. L'exécutable de test dépend notamment du code objet des tests, de celui du TAD et de la bibliothèque CUnit utilisée lors de l'édition de liens. Une modification de l'en-tête doit provoquer la recompilation des fichiers qui en dépendent. De même, une modification des commentaires documentaires doit être prise en compte lors de la génération de la documentation. Les noms des cibles ne suffisent donc pas : ce sont leurs dépendances et leurs recettes qui rendent la construction correcte et reproductible.

Cette organisation donne une place aux activités qui accompagnent le développement. Après une modification du TAD, on peut reconstruire les éléments concernés, exécuter les tests et actualiser la documentation en utilisant les mêmes commandes. La lisibilité des sources, la précision de l'interface et l'utilisation régulière de ces outils participent ensemble à la qualité du projet.

2.9 Synthèse

Un type abstrait de données décrit les entités manipulées par un programme à travers leurs opérations et les propriétés de celles-ci. Sa spécification précise les services disponibles, leur comportement et leurs préconditions, indépendamment de leur représentation en mémoire. Elle permet ainsi de raisonner sur les données du problème avant de choisir leur implantation.

L'analyse identifie ces entités et formalise leurs TAD, tout en décomposant la résolution du problème. La conception préliminaire traduit ensuite les opérations en signatures adaptées au paradigme retenu. En programmation structurée, les fonctions calculent des valeurs et les procédures modifient l'état du programme. Les algorithmes utilisateurs peuvent alors être décrits à partir de ces services, sans connaître leur réalisation. La conception détaillée choisit les représentations et définit les algorithmes des opérations, éventuellement à l'aide d'autres TAD.

L'exemple Point2D illustre cette séparation. Les coordonnées cartésiennes et les coordonnées polaires offrent deux représentations possibles d'un même point. Les calculs internes diffèrent, mais les opérations doivent respecter la même spécification. Un algorithme qui utilise uniquement le constructeur et les fonctions d'observation reste donc indépendant de ce choix de représentation.

Le développement traduit la conception dans un langage concret. En C, il faut notamment organiser l'interface et l'implantation dans les fichiers *.h* et *.c*, distinguer les noms publics et préciser, lorsque cela est nécessaire, la gestion des erreurs, de la mémoire, de la copie et de l'égalité. Cette étape demande une connaissance du langage : la traduction des algorithmes ne suffit pas à résoudre automatiquement ces questions.

La documentation rend le contrat du TAD accessible à ses utilisateurs. Les tests unitaires vérifient des conséquences de ce contrat sur des cas choisis, sans constituer une preuve générale de correction. Doxygen et CUnit facilitent respectivement ces deux activités ; l'organisation du projet et les règles de construction permettent de les pratiquer régulièrement. La lisibilité du code et la précision des explications restent essentielles pour que d'autres puissent utiliser et faire évoluer le travail réalisé.

La démarche repose donc sur une distinction à conserver tout au long du projet : ce que les opérations garantissent appartient à la spécification ; la manière d'obtenir ces résultats relève de leur réalisation. Cette distinction permet d'utiliser les données au niveau d'abstraction adapté au problème, tout en laissant évoluer les choix techniques qui les mettent en œuvre.

Chapitre 3

Les types abstraits de données collections

3.1 Introduction aux collections

Dans de nombreux problèmes, un algorithme doit manipuler un groupe d'éléments plutôt qu'une valeur isolée. Il faut pouvoir conserver ces éléments, en ajouter, en retirer ou retrouver ceux dont on a besoin. Les types abstraits de données collections décrivent ces possibilités en précisant les opérations disponibles et les règles qui organisent l'accès aux éléments.

Toutes les collections ne répondent pas au même besoin. Certaines imposent un ordre de retrait, d'autres permettent un accès par position ou par clé, et d'autres encore organisent les éléments de manière hiérarchique. Le choix d'une collection dépend donc des opérations nécessaires à la résolution du problème. Comme pour tout TAD, ce choix peut être effectué sans décider immédiatement comment les données seront représentées en mémoire.

3.1.1 Des TAD paramétrés par le type des éléments

Une même organisation peut servir à manipuler des éléments de types différents. La spécification d'une collection ne doit donc pas dépendre d'un type particulier lorsque ses opérations n'en ont pas besoin. On utilise pour cela le champ *paramètres* du TAD, introduit au chapitre précédent. Le nom générique *Element* y désigne le type des éléments de la collection, sans fixer ce type lors de la spécification.

Ce paramètre est un type, et non une valeur fournie à une opération. Par exemple, une liste d'entiers et une liste de points suivent les mêmes règles d'accès par position, mais les éléments obtenus sont respectivement des entiers et des points. La notation `Liste<Entier>` précise le type choisi pour les éléments d'une liste. Une fois ce choix effectué, les signatures des opérations s'interprètent en remplaçant *Element* par ce type.

Le paramétrage ne signifie pas que tout type convient nécessairement à toute collection. Lorsqu'une opération exige une propriété particulière des éléments, cette contrainte doit figurer explicitement dans la spécification. Une collection qui maintient ses éléments dans un ordre doit notamment disposer d'une relation d'ordre adaptée sur leur type. Les contraintes font ainsi partie du contrat du TAD, au même titre que les propriétés de ses opérations.

3.1.2 Un ensemble d'outils pour la conception des algorithmes

Ce chapitre présente d'abord les piles, les files, les listes et les listes ordonnées, puis les ensembles. L'instruction *pour chaque* permettra d'exprimer un parcours des éléments. Le dictionnaire introduira ensuite l'accès à une valeur à partir d'une clé. Enfin, les arbres permettront d'étudier une organisation hiérarchique, ses parcours et plusieurs TAD possédant des propriétés particulières.

Pour chaque collection, l'objectif est de comprendre les services proposés, leurs propriétés et leurs conditions d'utilisation. Les axiomes et les sémantiques permettent de raisonner sur les opérations sans recourir à une implantation. Les exemples de parcours montrent ensuite comment utiliser ces services dans des algorithmes.

Une fois définies, ces collections pourront être considérées comme des types disponibles lors de la conception détaillée, au même titre que les booléens ou les naturels. Leur disponibilité dans un algorithme ne suppose pas qu'elles soient directement fournies par le langage de programmation retenu. Leurs représentations et leur implantation en C seront étudiées ultérieurement ; nous nous concentrons ici sur leur spécification et leur utilisation abstraite.

3.2 Les collections linéaires et le TAD Ensemble

Les piles, les files et les listes organisent les éléments selon un ordre linéaire, mais ne proposent pas les mêmes possibilités d'accès. L'ensemble se distingue de ces collections par l'absence d'ordre et de répétitions. Dans les spécifications suivantes, *Element* désigne le type des éléments manipulés. Les opérations qui produisent une collection décrivent une valeur résultat : elles ne modifient pas une variable au niveau de la spécification mathématique.

3.2.1 La pile : le dernier entré est le premier sorti

Une pile suit une discipline LIFO, pour *Last In First Out*. L'élément ajouté le plus récemment est le premier accessible, au sommet de la pile. Empiler ajoute un élément à ce sommet ; dépiler produit la pile privée de son sommet. L'opération `obtenirElement` permet de consulter cet élément sans le retirer. Dépiler et consulter sont donc deux opérations distinctes, qui nécessitent une pile non vide.

Nom:	Pile	
Paramètre:	Element	
Utilise:	**Booleen**	
Opérations:	pile:	$\rightarrow$ Pile
	estVide:	Pile $\rightarrow$ **Booleen**
	empiler:	Pile $\times$ Element $\rightarrow$ Pile
	depiler:	Pile $\nrightarrow$ Pile
	obtenirElement:	Pile $\nrightarrow$ Element
Axiomes:	- $estVide(pile())$	
	- $non(estVide(empiler(p, e)))$	
	- $depiler(empiler(p, e)) = p$	
	- $obtenirElement(empiler(p, e)) = e$	
Préconditions:	depiler(p):	$non(estVide(p))$
	obtenirElement(p):	$non(estVide(p))$

Le constructeur `pile` fournit la pile vide. Les deux derniers axiomes expriment la discipline LIFO : retirer l'élément que l'on vient exactement d'empiler restitue la pile précédente, et consulter le sommet après un empilement donne l'élément ajouté.

Ces propriétés permettent de calculer le résultat d'une expression sans connaître la représentation de la pile. Considérons :

$$obtenirElement(depiler(empiler(empiler(pile(), e_1), e_2))).$$

Le troisième axiome donne d'abord

$$depiler(empiler(empiler(pile(), e_1), e_2)) = empiler(pile(), e_1).$$

Le quatrième donne ensuite

$$obtenirElement(empiler(pile(), e_1)) = e_1.$$

Les appels respectent leurs préconditions : la pile dépilée contient deux éléments, et celle dont on consulte le sommet en contient encore un.

Les piles interviennent notamment dans l'évaluation d'expressions arithmétiques en notation postfixée, appelée aussi notation polonaise inversée, et dans la gestion des appels de fonctions. Dans ce dernier cas, le retour du dernier appel effectué précède celui de l'appel qui l'a provoqué.

3.2.2 La file : le premier entré est le premier sorti

Une file suit une discipline FIFO, pour *First In First Out*. L'ajout se fait à une extrémité et le retrait à l'autre : parmi les éléments encore présents, le plus ancien est le premier accessible. Enfiler produit une file contenant un élément supplémentaire à la fin ; défiler retire l'élément de tête. Comme pour la pile, la consultation est séparée du retrait.

Nom:	File	
Paramètre:	Element	
Utilise:	**Booleen**	
Opérations:	file:	$\rightarrow$ File
	estVide:	File $\rightarrow$ **Booleen**
	enfiler:	File $\times$ Element $\rightarrow$ File
	defiler:	File $\nrightarrow$ File
	obtenirElement:	File $\nrightarrow$ Element
Axiomes:	- $estVide(file())$	
	- $non(estVide(enfiler(f, e)))$	
	- $defiler(enfiler(file(), e)) = file()$	
	- $obtenirElement(enfiler(file(), e)) = e$	
Préconditions:	defiler(f):	$non(estVide(f))$
	obtenirElement(f):	$non(estVide(f))$

Deux axiomes conditionnels complètent cette description. Pour toute file f non vide, on a :

$$defiler(enfiler(f, e)) = enfiler(defiler(f), e),$$

$$obtenirElement(enfiler(f, e)) = obtenirElement(f).$$

Ils expriment que l'ajout en fin de file ne change pas l'élément de tête. La condition sur f est essentielle : sur une file vide, l'élément ajouté est précisément celui qui devient accessible.

Reprenons l'expression utilisée pour la pile en remplaçant les opérations par celles de la file :

$$obtenirElement(defiler(enfiler(enfiler(file(), e_1), e_2))).$$

Après les deux ajouts, e_1 est en tête. L'axiome conditionnel du retrait permet d'écrire :

$$defiler(enfiler(enfiler(file(), e_1), e_2)) = enfiler(defiler(enfiler(file(), e_1)), e_2).$$

La file obtenue est donc $enfiler(file(), e_2)$, et la consultation renvoie e_2. Une même succession d'ajouts et d'un retrait conduit ainsi à un résultat différent selon que l'on utilise une pile ou une file.

Les files servent notamment à gérer les demandes adressées à une imprimante partagée ou des programmes en attente du processeur, lorsque la politique retenue s'appuie sur cet ordre de traitement. Elles interviennent également dans les parcours en largeur d'arbres, de treillis ou de graphes.

3.2.3 La liste : accéder aux éléments par leur position

Une liste permet de consulter, d'insérer ou de supprimer un élément à une position donnée. Les positions commencent ici à 1. Une liste de longueur n possède donc les positions 1 à n, et une insertion peut avoir lieu aux positions 1 à $n+1$. La dernière possibilité correspond à un ajout en fin de liste. Plusieurs positions peuvent contenir des éléments égaux.

Nom: Liste
Paramètre: Element
Utilise: **Booleen**, **NaturelNonNul**, **Naturel**
Opérations:

liste:	$\rightarrow$ Liste
estVide:	Liste $\rightarrow$ **Booleen**
inserer:	Liste $\times$ **NaturelNonNul** $\times$ Element $\nrightarrow$ Liste
supprimer:	Liste $\times$ **NaturelNonNul** $\nrightarrow$ Liste
obtenirElement:	Liste $\times$ **NaturelNonNul** $\nrightarrow$ Element
longueur:	Liste $\rightarrow$ **Naturel**

Axiomes:
- $estVide(liste())$
- $non(estVide(inserer(l, i, e)))$
- $supprimer(inserer(l, i, e), i) = l$
- $obtenirElement(inserer(inserer(l, i, e_1), i, e_2), i+1) = e_1$
- $longueur(liste()) = 0$
- $longueur(inserer(l, i, e)) = 1 + longueur(l)$

Préconditions:

inserer(l,i,e):	$i \leq longueur(l) + 1$
supprimer(l,i):	$i \leq longueur(l)$
obtenirElement(l,i):	$i \leq longueur(l)$

Les axiomes comportant une insertion s'appliquent lorsque ses préconditions sont respectées. La borne inférieure $i \geq 1$ est déjà imposée par le type **NaturelNonNul**. Sur une liste vide, seule l'insertion à la position 1 est possible ; aucune consultation ni suppression n'est définie.

Insérer à la position i place le nouvel élément à cette position et décale les positions des éléments suivants d'une unité. Le quatrième axiome le montre : après avoir inséré e_1, puis e_2 à la même position, e_1 se trouve à la position $i+1$. Supprimer un élément réduit la longueur d'une unité et décale les positions suivantes dans l'autre sens. Il s'agit ici d'un changement de positions dans la liste abstraite, sans présumer des déplacements en mémoire nécessaires dans une implantation.

3.2.4 La liste ordonnée : maintenir un ordre entre les éléments

Une liste ordonnée conserve un accès par position, mais la position d'un nouvel élément est déterminée par sa valeur. L'utilisateur fournit donc l'élément à insérer, sans choisir son indice. Le type *Element* doit posséder un ordre total : en particulier, deux éléments distincts sont comparables. Les éléments sont rangés dans l'ordre croissant, avec la possibilité de valeurs égales.

Nom: ListeOrdonnee
Paramètre: Element muni d'un ordre total
Utilise: **Booleen**, **NaturelNonNul**, **Naturel**
Opérations:

listeOrdonnee:	$\rightarrow$ ListeOrdonnee
estVide:	ListeOrdonnee $\rightarrow$ **Booleen**
inserer:	ListeOrdonnee $\times$ Element $\rightarrow$ ListeOrdonnee
supprimer:	ListeOrdonnee $\times$ **NaturelNonNul** $\nrightarrow$ ListeOrdonnee
obtenirElement:	ListeOrdonnee $\times$ **NaturelNonNul** $\nrightarrow$ Element
longueur:	ListeOrdonnee $\rightarrow$ **Naturel**

Axiomes:
- $estVide(listeOrdonnee())$
- $non(estVide(inserer(l, e)))$

- $longueur(listeOrdonnee()) = 0$
- $longueur(inserer(l, e)) = 1 + longueur(l)$

Préconditions: supprimer(l,i): $i \leq longueur(l)$
obtenirElement(l,i): $i \leq longueur(l)$

Pour préciser les propriétés des positions, posons $l' = inserer(l, e)$. Il existe une position i dans l' telle que

$$obtenirElement(l', i) = e \quad \text{et} \quad supprimer(l', i) = l.$$

Cet indice désigne une position de l'élément inséré, et non une position arbitraire. L'insertion augmente toujours la longueur, même si un élément égal est déjà présent : une liste ordonnée n'élimine pas les doublons.

L'ordre est une propriété de toute la liste. Pour deux positions valides $i < j$, on a

$$obtenirElement(l, i) \leq obtenirElement(l, j).$$

En particulier, après les insertions de deux éléments distincts e' et e, si leurs positions sont respectivement j et i, alors $e' < e$ implique $j < i$, et $e' > e$ implique $j > i$. La suppression conserve cet ordre, puisqu'elle retire un élément sans changer l'ordre relatif de ceux qui restent.

3.2.5 L'ensemble : l'appartenance sans ordre ni répétition

Un ensemble regroupe des éléments sans leur associer de position. Chaque élément n'y apparaît qu'une fois : ajouter un élément déjà présent ne change pas l'ensemble. On peut tester son appartenance et le retirer, mais on ne demande pas l'élément d'indice donné. La cardinalité indique le nombre d'éléments distincts présents.

Nom: Ensemble
Paramètre: Element
Utilise: **Booleen**, **Naturel**
Opérations:
ensemble: $\rightarrow$ Ensemble
ajouter: Ensemble $\times$ Element $\rightarrow$ Ensemble
retirer: Ensemble $\times$ Element $\rightarrow$ Ensemble
estPresent: Ensemble $\times$ Element $\rightarrow$ **Booleen**
cardinalite: Ensemble $\rightarrow$ **Naturel**
union: Ensemble $\times$ Ensemble $\rightarrow$ Ensemble
intersection: Ensemble $\times$ Ensemble $\rightarrow$ Ensemble
soustraction: Ensemble $\times$ Ensemble $\rightarrow$ Ensemble
Axiomes:
- $ajouter(ajouter(s, e), e) = ajouter(s, e)$
- $estPresent(ajouter(s, e), e)$
- $non(estPresent(retirer(s, e), e))$
- $cardinalite(ensemble()) = 0$

Le retrait d'un élément absent laisse l'ensemble inchangé. Si e est absent de s, les propriétés suivantes complètent les axiomes :

$$retirer(ajouter(s, e), e) = s,$$

$$cardinalite(ajouter(s, e)) = 1 + cardinalite(s).$$

Si e est déjà présent, on a au contraire

$$cardinalite(ajouter(s, e)) = cardinalite(s).$$

Dans ce dernier cas, ajouter puis retirer e ne restitue pas s, car le retrait supprime l'élément qui était déjà présent. Cette distinction explique pourquoi la première égalité exige une condition d'absence.

L'union de deux ensembles contient les éléments présents dans au moins l'un des deux. Leur intersection contient ceux qui appartiennent aux deux. La soustraction $soustraction(s_1, s_2)$ contient les éléments de s_1 qui n'appartiennent pas à s_2. Contrairement à l'union et à l'intersection, cette dernière opération dépend de l'ordre de ses arguments.

3.2.6 Parcourir une collection avec « pour chaque »

L'instruction *pour chaque* permet d'exécuter un traitement pour chaque élément d'un ensemble et simplifie également le parcours d'une liste, ordonnée ou non. Elle exprime le parcours sans imposer une représentation de la collection ni la gestion explicite d'un indice.

pour chaque element **de** collection
  actions utilisant element
finpour

Pour une liste, le parcours suit les positions successives et visite chaque occurrence, y compris lorsque plusieurs éléments sont égaux. Pour un ensemble, chaque élément est visité une fois, mais aucun ordre de visite n'est garanti par le TAD. Un algorithme sur un ensemble ne doit donc pas faire dépendre son résultat d'un ordre particulier de parcours.

La fonction suivante compte les occurrences d'un entier dans une liste. Elle conserve le compteur `somme`, initialisé à zéro, et l'incrémente chaque fois que l'élément visité est égal à l'entier recherché.

fonction compter (l : Liste<**Entier**>, unEntier : **Entier**) **: Naturel**

  Déclaration somme : **Naturel**
  e : **Entier**

debut
  somme ← 0
  pour chaque e **de** l
    si unEntier = e **alors**
      somme ← somme + 1
    finsi
  finpour
  retourner somme
fin

Sur une liste vide, le corps du parcours n'est jamais exécuté et le résultat est zéro. Sur une liste non vide, toutes les occurrences sont examinées. La fonction ne modifie pas la liste reçue ; les affectations concernent uniquement ses variables locales. Le même principe de parcours s'applique à un ensemble, mais le nombre d'occurrences d'un élément y est nécessairement zéro ou un.

3.3 Le TAD associatif Dictionnaire

Un dictionnaire associe des clés à des valeurs. Contrairement à une liste, dans laquelle un élément est retrouvé par sa position, il permet d'obtenir une valeur à partir de la clé qui lui est associée. La clé identifie une association : dans un même dictionnaire, une clé ne peut désigner qu'une seule valeur. En revanche, plusieurs clés distinctes peuvent être associées à des valeurs égales.

3.3.1 Les clés et les valeurs

Le TAD Dictionnaire possède deux paramètres de type, *Cle* et *Valeur*. Ils précisent respectivement la nature des clés et celle des valeurs, sans imposer qu'elles soient de même type. Une clé n'est pas un indice de position : l'accès repose sur son identité, et non sur son rang dans une succession d'éléments. Aucune relation d'ordre sur les clés n'est requise par cette spécification.

Ajouter un couple clé-valeur a deux effets possibles. Si la clé est absente, une nouvelle association est créée. Si elle est déjà présente, la valeur qui lui était associée est remplacée. Le dictionnaire

ne conserve donc pas plusieurs versions d'une association sous la même clé. Le retrait désigne également une association par sa clé ; il ne supprime pas les autres associations qui pourraient contenir une valeur égale.

3.3.2 Spécification des opérations

Le constructeur fournit un dictionnaire vide. Les opérations permettent d'ajouter ou de retirer une association, de tester la présence d'une clé, d'obtenir la valeur correspondante et de consulter les collections de clés ou de valeurs. L'accès à une valeur exige que la clé soit présente.

Nom: Dictionnaire
Paramètre: Cle, Valeur
Utilise: **Booleen**, Ensemble, Liste
Opérations:
dictionnaire: $\rightarrow$ Dictionnaire
ajouter: Dictionnaire $\times$ Cle $\times$ Valeur $\rightarrow$ Dictionnaire
retirer: Dictionnaire $\times$ Cle $\rightarrow$ Dictionnaire
estPresent: Dictionnaire $\times$ Cle $\rightarrow$ **Booleen**
obtenirValeur: Dictionnaire $\times$ Cle $\nrightarrow$ Valeur
obtenirCles: Dictionnaire $\rightarrow$ Ensemble<Cle>
obtenirValeurs: Dictionnaire $\rightarrow$ Liste<Valeur>
Axiomes:
- $non(estPresent(dictionnaire(), c))$
- $ajouter(ajouter(d, c, v_2), c, v_1) = ajouter(d, c, v_1)$
- $obtenirValeur(ajouter(d, c, v), c) = v$
- $estPresent(ajouter(d, c, v), c)$
- $non(estPresent(retirer(d, c), c))$

Préconditions: obtenirValeur(d,c): $estPresent(d, c)$

Le deuxième axiome exprime le remplacement : après deux ajouts avec la même clé c, seule la dernière valeur, v_1, lui est associée. Le troisième axiome précise la valeur obtenue après un ajout, et le quatrième garantit que cette consultation respecte sa précondition. Les opérations d'ajout et de retrait laissent les associations des autres clés inchangées.

Le retrait est défini même lorsque la clé est absente ; dans ce cas, il laisse le dictionnaire inchangé. Pour une clé c absente de d, on a également :

$$retirer(ajouter(d, c, v), c) = d.$$

La condition d'absence est nécessaire. Si c était déjà associée à une valeur, l'ajout remplace celle-ci, puis le retrait supprime l'association. Il ne rétablit pas la valeur précédente.

L'opération `estPresent` porte sur les clés, et non sur les valeurs. Elle permet de vérifier qu'un appel à `obtenirValeur` est défini. L'absence d'une clé n'est donc pas représentée par une valeur particulière du type *Valeur* : elle est décrite par le résultat du test de présence.

3.3.3 Obtenir les clés et les valeurs

L'opération `obtenirCles` renvoie un ensemble, car chaque clé n'apparaît qu'une fois et aucun ordre n'est imposé. Cet ensemble contient exactement les clés présentes dans le dictionnaire. Pour toute clé c :

$$Ensemble.estPresent(obtenirCles(d), c) \iff Dictionnaire.estPresent(d, c).$$

Les préfixes distinguent ici le test d'appartenance à l'ensemble du test de présence dans le dictionnaire. On ne peut pas accéder à cet ensemble par un indice de liste. Pour examiner les associations, on peut parcourir ses clés avec l'instruction *pour chaque*, puis demander la valeur associée à chacune. Tant que le dictionnaire reste inchangé pendant ce parcours, la présence de la clé visitée garantit la précondition de la consultation.

L'opération `obtenirValeurs` renvoie une liste contenant une valeur par association. Une valeur apparaît donc plusieurs fois si plusieurs clés lui sont associées. Utiliser un ensemble de valeurs ferait perdre cette multiplicité. L'ordre des valeurs dans la liste n'est pas fixé ici : il ne faut ni en déduire un ordre sur les clés, ni supposer qu'une position de cette liste permet de retrouver une clé correspondante.

Le dictionnaire complète ainsi les collections précédentes en proposant un accès par clé. Sa spécification décrit les associations et leur évolution, sans indiquer comment les clés sont recherchées ni comment les couples sont représentés en mémoire.

Chapitre 4

Les structures de données dynamiques

4.1 Pourquoi des structures de données dynamiques ?

Le nombre d'éléments qu'un programme doit conserver n'est pas toujours connu à l'avance. Il peut dépendre des données reçues et évoluer pendant l'exécution, au fil des ajouts et des suppressions. La manière d'organiser ces éléments en mémoire doit tenir compte de cette variation.

Un tableau de capacité fixée à l'avance permet de réserver de la place pour un nombre maximal d'éléments, noté `MAX`. Cette solution impose cependant un compromis. Si le nombre n d'éléments effectivement stockés est très inférieur à cette capacité, une partie de l'espace réservé reste inutilisée. Si n doit dépasser `MAX`, le tableau ne peut plus accueillir toutes les données. Augmenter la capacité prévue repousse cette limite, mais ne permet pas de l'adapter aux besoins réels de chaque exécution.

Les structures de données dynamiques permettent de faire évoluer le stockage en réservant de la mémoire au cours de l'exécution et en la libérant lorsqu'elle n'est plus nécessaire. Pour les structures chaînées étudiées ici, les éléments sont placés dans des zones mémoire distinctes, appelées nœuds. Chaque nœud contient l'information à conserver et une ou plusieurs références vers d'autres nœuds. Ces références, représentées par des pointeurs, permettent de retrouver les éléments sans exiger qu'ils occupent des emplacements contigus.

Il n'est donc pas nécessaire de déclarer à l'avance une variable pour désigner chaque futur élément. Un pointeur donne accès à la structure, puis les liens contenus dans les nœuds permettent d'atteindre les autres parties. Cette organisation conduit naturellement à des définitions récursives : un nœud d'une liste donne accès à la suite de cette liste, tandis qu'un nœud d'un arbre donne accès à ses sous-arbres.

L'adaptation du stockage ne signifie pas que son coût se réduit à la seule taille des éléments. Les liens occupent eux aussi de la mémoire, et les opérations d'allocation et de libération doivent être prises en compte. La quantité de mémoire disponible reste limitée. Le programme doit surtout conserver des liens cohérents : perdre le dernier accès à une zone allouée peut empêcher de l'utiliser ou de la libérer, tandis qu'un accès à une zone déjà libérée est invalide.

Ce chapitre présente ces mécanismes au niveau de la conception détaillée, dans le cadre de la programmation structurée. Nous préciserons d'abord le rôle de la mémoire, des pointeurs et de l'allocation. Nous étudierons ensuite les listes simplement chaînées et leurs algorithmes, puis les listes doublement chaînées. Les tables de hachage illustreront une organisation combinant un tableau et des listes chaînées. Enfin, les arbres binaires permettront d'aborder les structures dynamiques hiérarchiques.

Les représentations et les opérations seront décrites en pseudocode. L'objectif est de comprendre comment construire, parcourir, modifier et libérer ces structures, en suivant précisément les effets des opérations sur les nœuds et leurs liens.

4.2 Mémoire, pointeurs et allocation

Pour manipuler une structure dynamique, il faut distinguer une zone mémoire, la valeur qu'elle contient et le pointeur qui permet d'y accéder. Réserver une zone, y écrire une information et conserver son adresse sont trois opérations différentes.

4.2.1 Organisation de la mémoire

Lors de l'exécution, le programme et ses données occupent de la mémoire. Une organisation courante distingue la zone du code, appelée *text*, les données de durée de vie statique, la pile d'exécution et le tas. La zone *data* contient généralement les données statiques initialisées, tandis que *bss* accueille celles initialisées à zéro. Les constantes peuvent être placées dans une zone en lecture seule. Cette organisation décrit un modèle usuel, dont les détails dépendent de l'environnement d'exécution.

La pile d'exécution conserve notamment les informations nécessaires aux appels des sous-programmes : paramètres, variables locales et adresse de retour. Chaque appel dispose d'un cadre d'exécution ; à son retour, ce cadre peut être retiré. La figure 4.1 représente deux cadres correspondant aux sous-programmes `DrawSquare` et `DrawLine`. Les paramètres et les variables locales du dernier appel se trouvent près du sommet de la pile.

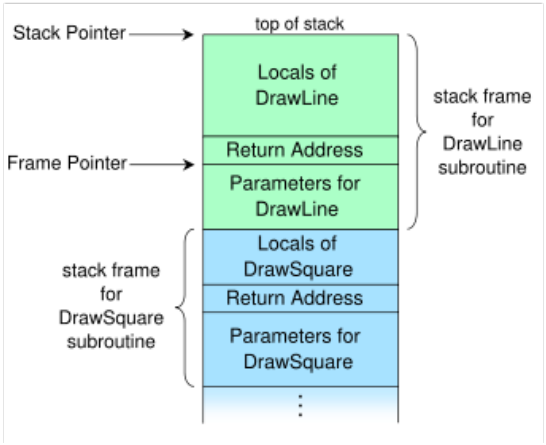


FIGURE 4.1 – Cadres d'appels dans la pile d'exécution. Source : Wikipédia.

Le tas, ou *heap*, accueille les zones allouées dynamiquement. Leur durée de vie n'est pas limitée à l'appel du sous-programme qui les a créées. Une zone peut rester accessible après son retour, à condition qu'un pointeur valide permette encore de l'atteindre.

4.2.2 Allocation statique, automatique et dynamique

Pour une variable de taille fixée à l'avance, le compilateur dispose des informations nécessaires pour organiser sa réservation mémoire. Il faut cependant distinguer la taille prévue et la durée de vie. Une variable de durée de vie statique existe pendant toute l'exécution. Une variable locale automatique existe pendant l'exécution du bloc ou de l'appel concerné, même lorsque sa taille est connue à la compilation. Dans ce dernier cas, la réservation et la récupération sont prises en charge automatiquement.

L'allocation dynamique est demandée explicitement pendant l'exécution. Elle permet notamment de créer un nombre de nœuds qui n'était pas connu à la compilation, même si la taille de chaque nœud est fixée par son type. Le programme doit conserver l'accès aux zones obtenues et les libérer lorsqu'elles ne sont plus nécessaires.

4.2.3 Référencer et déréférencer

Un pointeur sur un type T, noté $\hat{}T$, contient une adresse permettant d'accéder à une zone destinée à une valeur de type T. La valeur **null** indique l'absence de zone désignée. Un pointeur doit recevoir une valeur définie avant d'être utilisé ; déclarer une variable pointeur ne crée pas la zone qu'elle pourra désigner.

L'opérateur @x donne l'adresse de la variable x. L'expression p$\hat{}$ déréférence p : elle désigne la valeur contenue dans la zone pointée. L'exemple suivant montre la différence entre modifier un pointeur et modifier la zone désignée :

x : **Entier** ; p : $\hat{}$**Entier**
x $\leftarrow$ 3
p $\leftarrow$ @x
p$\hat{}$ $\leftarrow$ 5

Après ces instructions, x vaut 5 : la dernière affectation écrit dans x par l'intermédiaire de son adresse. Déréférencer exige une adresse valide. Un pointeur égal à **null**, non initialisé ou désignant une zone dont la durée de vie est terminée ne satisfait pas cette condition.

4.2.4 Réserver et libérer une zone

La procédure **allouer**(p) réserve une zone du type pointé et place son adresse dans p. Son contenu doit ensuite être initialisé. La procédure **desallouer**(p) libère la zone précédemment allouée. Dans les algorithmes de ce chapitre, on suppose que les allocations réussissent afin de se concentrer sur les liens ; une réalisation complète doit aussi prévoir le traitement d'un échec de réservation.

p : $\hat{}$**Entier**
allouer(p)
p$\hat{}$ $\leftarrow$ 7
desallouer(p)
p $\leftarrow$ **null**

La libération ne doit pas être confondue avec une affectation à **null**. Affecter cette valeur seule ne libère aucune zone. Inversement, libérer une zone ne remet pas automatiquement à **null** tous les pointeurs qui la désignaient. Ces autres pointeurs deviennent invalides. On ne libère que des zones allouées dynamiquement, une seule fois chacune.

Une variable pointeur peut elle-même être locale, globale ou contenue dans une zone allouée. Cette dernière possibilité est essentielle : un nœud peut contenir les pointeurs qui donnent accès aux nœuds suivants. Un nombre variable de zones devient ainsi accessible à partir d'un seul point d'entrée.

4.3 Les listes simplement chaînées

Une liste simplement chaînée est soit vide, soit constituée d'un élément suivi d'une liste simplement chaînée. Un nœud contient donc deux champs : l'élément et un pointeur vers la suite. La liste vide est représentée par **null**. Pour les listes étudiées ici, les liens ne forment pas de cycle : en suivant les suivants, on finit par atteindre cette valeur.

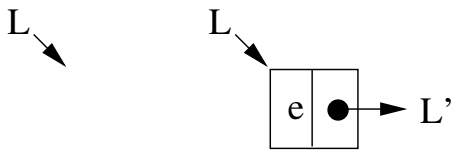


FIGURE 4.2 – Liste vide à gauche ; nœud contenant e et donnant accès à la suite L' à droite.

4.3.1 Représentation et accès aux champs

Le type `Element` désigne le type des informations stockées. La définition suivante est récursive par l'intermédiaire d'un pointeur : un nœud ne contient pas toute la suite, mais seulement son adresse.

Type ListeChainee = ^Noeud
Type Noeud = **Structure**
element : Element
listeSuivante : ListeChainee
finstructure

Nous pourrions très bien utiliser les pointeurs pour écrire des algorithmes sur les listes chainées. Cependant le code produit devient assez rapidement illisible. Une bonne pratique est donc de lister un ensemble de fonctions et procédures qui les encapsulent, et de se limiter à leur utilisation, rendant la lecture des algorithmes plus facile. Quelque soit l'option choisie, il est important de ne pas mélanger ces deux points vues. Dans la suite de ce document nous utiliseront uniquement ces fonctions d'encapsualtion. Les deux premières sont :

fonction listeVide **() :** ListeChainee
debut
retourner null
fin
fonction estVide (l : ListeChainee) **: Booleen**
debut
retourner l = **null**
fin

Le pointeur l désigne la tête de la liste. Les opérations suivantes centralisent l'accès aux champs du premier nœud. Leur précondition évite de déréférencer une liste vide. Obtenir la suite ne copie aucun nœud : la valeur renvoyée donne accès aux nœuds déjà présents.

fonction obtenirElement (l : ListeChainee) **:** Element
⌊**précondition(s)** non estVide(l)
debut
retourner l^.element
fin
fonction obtenirListeSuivante (l : ListeChainee) **:** ListeChainee
⌊**précondition(s)** non estVide(l)
debut
retourner l^.listeSuivante
fin
procédure fixerElement (**E**/**S** l : ListeChainee, **E** e : Element)
⌊**précondition(s)** non estVide(l)
debut
l^.element ← e
fin
procédure fixerListeSuivante (**E**/**S** l : ListeChainee,
E suite : ListeChainee)
⌊**précondition(s)** non estVide(l)
debut
l^.listeSuivante ← suite
fin

Ces procédures modifient le contenu du nœud, sans déplacer celui-ci. Remplacer son lien suivant exige de conserver ailleurs l'accès à l'ancienne suite si elle doit encore être utilisée ou libérée. Il faut aussi éviter de relier un nœud à lui-même ou à un de ses prédécesseurs, ce qui créerait un cycle.

4.3.2 Ajouter en tête

Ajouter un élément crée un nœud dont le suivant est l'ancienne tête. Le pointeur de la liste doit ensuite désigner ce nouveau nœud. L'algorithme conserve d'abord l'ancienne adresse dans `temp`, car l'allocation remplace la valeur de l.

procédure ajouter **(E/S** l : ListeChainee, **E** e : Element**)**

Déclaration temp : ListeChainee

debut
temp ← l
allouer(l)
l^.element ← e
fixerListeSuivante(l,temp)
fin

La même suite d'instructions convient à une liste vide : le nouveau nœud pointe alors vers **null**. Aucun parcours n'est nécessaire ; les nœuds qui formaient la liste restent à leur emplacement.

4.3.3 Supprimer la tête et libérer la liste

Pour retirer la tête, on mémorise son adresse, puis on fait avancer l vers la suite avant de libérer l'ancien nœud. Lire le lien suivant après la libération serait un accès invalide.

procédure supprimerTete **(E/S** l : ListeChainee**)**

⌊**précondition(s)** non estVide(l)

Déclaration temp : ListeChainee

debut
temp ← l
l ← obtenirListeSuivante(l)
desallouer(temp)
fin
procédure supprimer **(E/S** l : ListeChainee**)**
debut
tant que non estVide(l) **faire**
supprimerTete(l)
fintantque
fin

La suppression complète répète ce traitement jusqu'à la liste vide. Chaque nœud est libéré une fois, et l vaut finalement **null**. Ces algorithmes supposent que les nœuds ne sont pas partagés avec une autre structure qui devrait continuer à les utiliser. Ils libèrent les nœuds ; si les éléments contiennent eux-mêmes des ressources allouées, leur libération doit être prévue séparément.

4.4 Les algorithmes sur les listes chaînées

Une liste chaînée peut être traitée en avançant de nœud en nœud ou en traitant sa tête puis sa suite par un appel récursif. Les deux approches s'appuient sur les opérations précédentes, ce qui évite de répéter les accès aux champs dans chaque algorithme.

4.4.1 Parcours itératif et récursif

Le parcours itératif utilise un pointeur local qui avance jusqu'à la liste vide. Le paramètre d'entrée l est ici une copie du pointeur de l'appelant : modifier cette copie ne modifie ni la tête conservée par l'appelant ni les liens des nœuds. La procédure `afficherElement` assure l'affichage d'une information de type `Element`.

procédure afficher **(E** l : ListeChainee**)**

```
debut
   tant que non estVide(l) faire
      afficherElement(obtenirElement(l))
      l ← obtenirListeSuivante(l)
   fintantque
fin
```

La version récursive réalise le même affichage. La liste vide constitue le cas d'arrêt ; sur une liste non vide, on affiche l'élément de tête puis on poursuit sur la suite.

```
procédure afficher (E l : ListeChainee)
debut
   si non estVide(l) alors
      afficherElement(obtenirElement(l))
      afficher(obtenirListeSuivante(l))
   finsi
fin
```

Chaque version visite les nœuds dans le même ordre. La version itérative conserve un pointeur de parcours ; la version récursive peut conserver un cadre d'appel par nœud dans la pile d'exécution. La définition récursive de la liste facilite cette seconde écriture, mais ne l'impose pas.

4.4.2 Supprimer la première occurrence : version récursive

La procédure suivante retire le premier élément égal à e. Une liste vide ne demande aucun traitement. Si la tête contient e, on la supprime. Sinon, on recherche dans la suite et on rattache la suite obtenue à la tête conservée.

```
procédure supprimerElement (E/S l : ListeChainee, E e : Element)
   Déclaration   temp : ListeChainee
debut
   si non estVide(l) alors
      si obtenirElement(l) = e alors
         supprimerTete(l)
      sinon
         temp ← obtenirListeSuivante(l)
         supprimerElement(temp,e)
         fixerListeSuivante(l,temp)
      finsi
   finsi
fin
```

La variable intermédiaire `temp` est nécessaire pour transmettre une variable modifiable à l'appel récursif. Si la tête de cette suite est supprimée, son adresse change. Il faut donc réécrire le lien du nœud précédent avec `fixerListeSuivante` ; modifier seulement la copie locale du pointeur laisserait ce lien désigner un nœud libéré. L'algorithme s'arrête au premier élément trouvé et conserve les éventuelles occurrences suivantes. Si e est absent, la liste reste inchangée.

4.4.3 Supprimer la première occurrence : version itérative

La suppression en tête est traitée séparément. Pour les autres positions, deux pointeurs sont nécessaires : `droite` désigne le nœud examiné et `gauche` son prédécesseur. Tant que l'élément n'est pas trouvé, ils avancent ensemble.

```
procédure supprimerElement (E/S l : ListeChainee, E e : Element)
   Déclaration   gauche,droite : ListeChainee
                 trouve : Booleen
debut
```

```
    si non estVide(l) alors
        si obtenirElement(l) = e alors
            supprimerTete(l)
        sinon
            gauche ← l
            droite ← obtenirListeSuivante(l)
            trouve ← FAUX
            tant que non estVide(droite) et non trouve faire
                si obtenirElement(droite) = e alors
                    trouve ← VRAI
                sinon
                    gauche ← droite
                    droite ← obtenirListeSuivante(droite)
                finsi
            fintantque
            si trouve alors
                fixerListeSuivante(gauche,obtenirListeSuivante(droite))
                supprimerTete(droite)
            finsi
        finsi
    finsi
fin
```

Lorsque l'élément est trouvé, on relie son prédécesseur à son suivant, puis on libère le nœud retiré. L'ordre de ces opérations préserve l'accès à la suite. Si le parcours atteint la fin sans succès, aucun lien n'est modifié. Sur une liste d'un seul élément, soit la tête est supprimée, soit `droite` est vide dès le début de la recherche.

4.4.4 Autres traitements à concevoir

Les mêmes mécanismes permettent de compter les éléments, d'obtenir celui d'une position donnée ou de tester sa présence. On peut également concevoir la concaténation de deux listes, l'insertion à une position, l'insertion dans une liste ordonnée, la fusion de deux listes ordonnées et l'inversion d'une liste. Ces traitements constituent des exercices de conception : pour chacun, il faut préciser si les nœuds sont conservés, déplacés par modification des liens ou copiés dans de nouvelles allocations.

Avant d'écrire les instructions, il est utile d'identifier les pointeurs à conserver et les liens à modifier. Les cas de la liste vide, de la tête et du dernier nœud doivent être examinés explicitement. Une représentation graphique avant et après la modification aide à vérifier qu'aucune partie de la liste ne devient inaccessible.

4.5 Les listes doublement chaînées

Dans une liste simplement chaînée, un nœud ne donne directement accès qu'à son suivant. Pour retrouver son prédécesseur, il faut avoir conservé celui-ci pendant un parcours. Une liste doublement chaînée ajoute un lien vers le nœud précédent, ce qui permet de se déplacer dans les deux sens. Ce lien supplémentaire occupe de la mémoire et doit être maintenu lors des insertions et des suppressions.

4.5.1 Représentation et cohérence des liens

Nous considérons une liste non circulaire. Le premier nœud n'a pas de prédécesseur et le dernier n'a pas de suivant : les liens correspondants valent **null**. Un pointeur sur un nœud

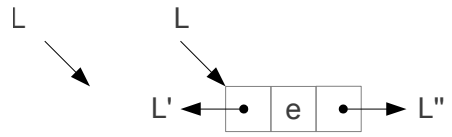


FIGURE 4.3 – Liste vide et nœud doublement chaîné : L' désigne le précédent et L'' le suivant.

intermédiaire donne accès aux deux directions de la liste. L'abréviation `LDC` désigne le type `ListeDoublementChainee`.

Type LDC = ^NoeudDouble
Type NoeudDouble = **Structure**
element : Element
precedent : LDC
suivant : LDC
finstructure

Les deux directions doivent être cohérentes. Si le suivant d'un nœud p est q, alors le précédent de q doit être p. Modifier un seul lien sans mettre à jour son correspondant peut donner deux parcours incompatibles. Les procédures élémentaires de modification d'un champ sont donc des briques dont l'appelant doit coordonner l'utilisation.

4.5.2 Les opérations élémentaires

Les fonctions d'accès exigent un nœud non vide ; le précédent ou le suivant renvoyé peut toutefois être vide. Les signatures suivantes utilisent `Element` pour conserver le même type d'information dans toutes les opérations.

fonction listeVide **() :** LDC
fonction estVide (l : LDC) **: Booleen**
fonction obtenirElement (l : LDC) **:** Element
⌊**précondition(s)** non estVide(l)
fonction obtenirListeSuivante (l : LDC) **:** LDC
⌊**précondition(s)** non estVide(l)
fonction obtenirListePrecedente (l : LDC) **:** LDC
⌊**précondition(s)** non estVide(l)

Les procédures suivantes permettent de modifier l'information ou les liens. Le mode entrée-sortie souligne que la structure désignée est modifiée, même si l'adresse du nœud reste identique.

procédure inserer **(E/S** l : LDC, **E** e : Element**)**
procédure fixerElement **(E/S** l : LDC, **E** e : Element**)**
⌊**précondition(s)** non estVide(l)
procédure fixerListeSuivante **(E/S** l : LDC, **E** suite : LDC**)**
⌊**précondition(s)** non estVide(l)
procédure fixerListePrecedente **(E/S** l : LDC, **E** avant : LDC**)**
⌊**précondition(s)** non estVide(l)
procédure supprimerNoeud **(E/S** l : LDC, **S** avant,apres : LDC**)**
⌊**précondition(s)** non estVide(l)
procédure supprimer **(E/S** l : LDC**)**

La signature d'`inserer` ne suffit pas à décider si l'ajout se fait avant ou après le nœud désigné. De même, la suppression d'un nœud doit préciser la valeur finale de l, le rôle des paramètres `avant` et `apres`, et les raccordements réalisés. Ces choix doivent être fixés avant d'écrire les corps des procédures.

4.5.3 Exercice de conception détaillée

Écrire les algorithmes des opérations précédentes en précisant d'abord les conventions d'insertion et de suppression. Pour une insertion entre deux nœuds, représenter les quatre liens qui relient le nouveau nœud à ses voisins. Pour un retrait, conserver les adresses des voisins avant la libération, puis assurer leur raccordement dans les deux directions.

Traiter séparément la liste vide, le premier nœud, le dernier nœud et le cas d'un nœud unique. Préciser également si la suppression complète reçoit nécessairement la tête de la liste ou si elle accepte un nœud quelconque. Dans ce dernier cas, ne libérer que les suivants laisserait une partie de la liste allouée. La cohérence du parcours dans les deux sens permet de vérifier les algorithmes proposés.

4.6 Une représentation mixte : la table de hachage

Une table de hachage peut combiner un tableau et des listes chaînées. Chaque case du tableau contient un pointeur vers une liste, initialement vide. Le tableau permet d'accéder directement à une case ; la liste qui lui est associée peut contenir un nombre variable d'éléments.

4.6.1 Choisir une case par une fonction de hachage

Une fonction de hachage calcule un entier à partir d'une valeur ou d'une partie de celle-ci. Une réduction par modulo ramène cet entier dans l'intervalle des indices du tableau. Pour un tableau de m cases numérotées de 1 à m, l'indice s'écrit ainsi

$$i = 1 + (h(v) \bmod m).$$

Si les indices commencent à zéro, le décalage de 1 n'est pas nécessaire. La même information et la même règle de calcul doivent être employées pour insérer un élément et pour le retrouver.

Des éléments différents peuvent aboutir à la même case : on parle de collision. Ils sont alors conservés dans la même liste chaînée. Pour rechercher un élément, on calcule son indice, puis on parcourt cette liste en comparant les informations recherchées. L'égalité des indices ne suffit pas à établir l'égalité des éléments.

4.6.2 Exemple : conserver le prénom et l'âge de personnes

Comme l'indique l'exemple de la figure 4.4, on utilise onze cases et, pour calculer l'indice, on additionne les rangs des lettres du prénom dans l'alphabet. Pour CECILE, on obtient :

$$1 + ((3 + 5 + 3 + 9 + 12 + 5) \bmod 11) = 1 + (37 \bmod 11) = 5.$$

ALAIN conduit également à la case 5 ; on nomme cela une collision.

La présence d'une collision n'est pas une erreur. La recherche de CECILE examine la liste de la case 5 et distingue son nœud de celui d'ALAIN. Une case inutilisée contient une liste vide. L'ajout d'un nœud à une liste ne demande pas d'agrandir le tableau, mais une accumulation dans quelques cases allonge les recherches.

4.6.3 Répartition et coût des opérations

L'accès à une case du tableau est en temps constant, mais cela ne rend pas toute recherche automatiquement constante. Il faut compter le calcul du hachage et le parcours de la liste sélectionnée. Si tous les éléments aboutissent à la même case, on retrouve le coût d'une recherche dans une liste de tous les éléments.

Les performances dépendent donc du nombre d'éléments au regard du nombre de cases, de la répartition entre ces cases et du coût de calcul de la fonction. Une répartition aussi uniforme que possible évite les listes excessivement longues. La somme des lettres est facile à calculer, mais elle

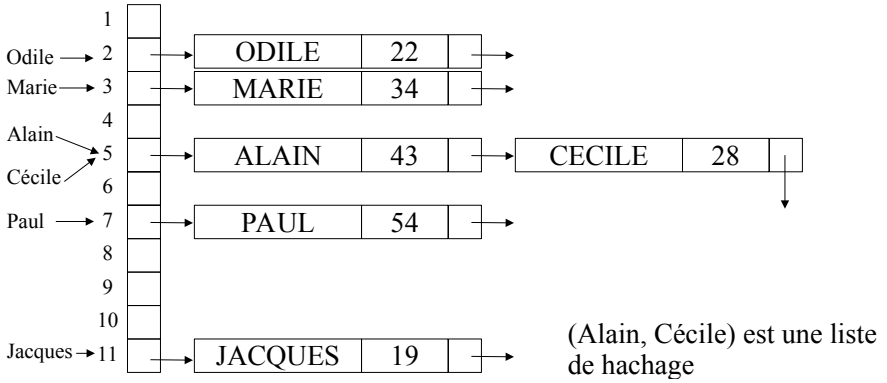


FIGURE 4.4 – Tableau de listes chaînées contenant des prénoms et des âges.

donne notamment le même résultat pour deux prénoms constitués des mêmes lettres dans un ordre différent. Elle illustre le mécanisme sans assurer une bonne répartition pour toutes les données.

Plusieurs méthodes de calcul peuvent être utilisées :

Extraction. On sélectionne des bits de la représentation de la valeur. Leur choix doit tenir compte des variations effectives des données.

Compression. On découpe la représentation en parties de même taille, puis on les combine, par exemple par addition ou par ou exclusif.

Division. On prend le reste d'une division entière par un nombre m. Choisir un nombre premier est une possibilité courante pour limiter certaines régularités, sans garantir à lui seul la qualité de la répartition.

La représentation combine ainsi une partie de taille fixée, le tableau, et des parties de taille variable, les listes. Sa libération doit parcourir les listes de toutes les cases pour récupérer les nœuds alloués.

4.7 Les structures dynamiques hiérarchiques : l'arbre binaire

Une structure hiérarchique relie un nœud à plusieurs descendants. Lorsque le nombre de liens est fixé, on peut réserver un champ pour chacun ou utiliser un tableau de liens. Si le nombre de descendants immédiats est variable, une liste chaînée peut les regrouper. Nous étudions ici le cas de deux liens, vers les sous-arbres gauche et droit.

4.7.1 Représentation par des nœuds

Un arbre binaire est représenté par un pointeur vers sa racine. Chaque nœud contient un élément et deux pointeurs du même type. Un sous-arbre absent est représenté par **null** ; une feuille possède donc deux liens nuls. L'arbre vide est lui-même représenté par **null**.

Type ArbreBinaire = ^Noeud
Type Noeud = **Structure**
  lElement : Element
  filsGauche : ArbreBinaire
  filsDroit : ArbreBinaire
finstructure

Pour former un arbre, ces liens ne doivent créer aucun cycle, et un même nœud ne doit pas appartenir à deux sous-arbres distincts. Ces conditions sont importantes pour le parcours et pour une libération qui doit visiter chaque nœud une seule fois.

4.7.2 Construction et consultation

Les signatures suivantes distinguent la création de l'arbre vide, le test du vide, la création d'une racine et la consultation d'un nœud existant.

fonction arbreBinaire **() :** ArbreBinaire
fonction estVide (a : ArbreBinaire) **: Booleen**
fonction ajouterRacine (fg,fd : ArbreBinaire, e : Element) **:** ArbreBinaire
fonction obtenirElement (a : ArbreBinaire) **:** Element

⌊**précondition(s)** non estVide(a)

fonction obtenirFilsGauche (a : ArbreBinaire) **:** ArbreBinaire

⌊**précondition(s)** non estVide(a)

fonction obtenirFilsDroit (a : ArbreBinaire) **:** ArbreBinaire

⌊**précondition(s)** non estVide(a)

La construction d'une racine réserve un nœud, y place l'élément et les adresses des deux sous-arbres, puis renvoie son adresse. Elle ne copie pas les sous-arbres : ceux-ci doivent être disjoints pour obtenir une structure sans partage. Les fonctions de consultation exigent que la racine existe, mais les liens gauche et droit renvoyés peuvent être nuls.

4.7.3 Modification et suppression

Changer l'élément ou un fils modifie le nœud désigné. Les paramètres entrée-sortie ci-dessous rendent cet effet explicite. Remplacer un fils ne libère pas automatiquement l'ancien sous-arbre : il faut en conserver l'accès ou l'avoir libéré avant de perdre son adresse.

procédure fixerElement (**E**/**S** a : ArbreBinaire, **E** e : Element)

⌊**précondition(s)** non estVide(a)

procédure fixerFilsGauche (**E**/**S** a : ArbreBinaire, **E** ag : ArbreBinaire)

⌊**précondition(s)** non estVide(a)

procédure fixerFilsDroit (**E**/**S** a : ArbreBinaire, **E** ad : ArbreBinaire)

⌊**précondition(s)** non estVide(a)

procédure supprimerRacine (**E**/**S** a : ArbreBinaire, **S** fg,fd : ArbreBinaire)

⌊**précondition(s)** non estVide(a)

procédure supprimer (**E**/**S** a : ArbreBinaire)

Supprimer la racine seule exige de préserver ses deux fils. Les paramètres de sortie `fg` et `fd` permettent de rendre ces sous-arbres accessibles après la libération du nœud. Le contrat doit aussi préciser la valeur finale du pointeur a, qui ne doit plus désigner la zone libérée.

Supprimer l'arbre entier consiste au contraire à libérer tous ses nœuds. Une démarche récursive naturelle traite les deux sous-arbres avant de libérer la racine. Si l'on choisit de libérer d'abord la racine, il faut avoir sauvegardé les deux adresses avant cette libération. La récursivité suit la forme de la structure ; une réalisation itérative reste possible à condition de mémoriser explicitement les parties encore à traiter.

4.7.4 Exercice de conception détaillée

Écrire les algorithmes des fonctions et procédures précédentes. Expliciter les conditions de raccordement des sous-arbres et l'état final des paramètres modifiés. Pour la suppression complète, vérifier le cas vide, la feuille, le nœud à un seul fils et le nœud à deux fils. Justifier que chaque nœud est libéré une seule fois et qu'aucun accès n'utilise un nœud déjà libéré. Comme pour les listes, la destruction des informations éventuellement allouées à l'intérieur d'un élément demande un traitement complémentaire.

4.8 Synthèse : manipuler les liens avec rigueur

Les structures dynamiques organisent des zones allouées pendant l'exécution et reliées par des pointeurs. Une liste simplement chaînée conserve un lien vers la suite ; une liste doublement chaînée ajoute le lien précédent ; un arbre binaire possède deux liens vers ses sous-arbres. Une table de hachage peut associer un tableau de taille fixée à des listes dont la taille évolue. Ces représentations adaptent le nombre de nœuds aux données présentes, au prix de la mémoire nécessaire aux liens et de leur gestion explicite.

L'affectation d'un pointeur copie une adresse, sans dupliquer la zone pointée. Deux pointeurs peuvent ainsi désigner le même nœud. Modifier la valeur de l'un des pointeurs ne modifie pas celle de l'autre ; modifier le nœud par l'un des accès est en revanche visible depuis l'autre. Libérer ce nœud invalide tous les accès qui le désignaient. L'identité des adresses et l'égalité des informations sont donc deux questions distinctes : des nœuds différents peuvent contenir des éléments égaux.

Une modification correcte préserve l'accès aux parties encore utiles. Avant de remplacer un lien ou de libérer un nœud, il faut conserver les adresses nécessaires à la poursuite du traitement. Les suppressions récursives montrent aussi qu'une adresse modifiée localement doit parfois être réécrite dans le nœud précédent. La libération complète doit visiter chaque zone dont la structure est responsable, sans en oublier et sans libérer deux fois la même zone.

La définition récursive des listes et des arbres guide les algorithmes, mais laisse le choix entre plusieurs écritures. Les parcours itératifs conservent explicitement les pointeurs nécessaires ; les parcours récursifs utilisent également les cadres d'appels. Dans les deux cas, examiner les structures vides, les extrémités et les liens absents permet de vérifier les conditions des accès et l'arrêt des traitements.

La conception d'une structure dynamique exige ainsi autant d'attention aux adresses et à leur durée de validité qu'aux informations stockées. Des opérations élémentaires bien définies, des conventions de modification explicites et des schémas des liens permettent de raisonner sur ces effets avant toute traduction dans un langage de programmation.

Chapitre 5

La conception des TAD collections

5.1 De la spécification à la représentation

Les collections ont été décrites par leurs opérations, leurs propriétés et leurs préconditions. Les structures de données dynamiques ont ensuite fourni des moyens de relier des éléments en mémoire. Il reste à établir comment ces moyens permettent de réaliser les services attendus d'une pile, d'une file, d'un ensemble ou d'un dictionnaire.

La spécification ne détermine pas une représentation unique. Une pile peut être représentée par un tableau accompagné d'un compteur ou par une liste chaînée dont la tête constitue le sommet. Inversement, une même structure chaînée peut servir à réaliser plusieurs collections, à condition de lui associer les algorithmes et les invariants appropriés. La structure indique comment les éléments sont organisés ; les opérations déterminent comment cette organisation est utilisée.

La conception préliminaire fixe d'abord les signatures des fonctions et des procédures. Elle précise notamment quelles opérations produisent une valeur nouvelle et lesquelles modifient une variable existante. La conception détaillée choisit ensuite la représentation et décrit les algorithmes de ces sous-programmes. Le type ainsi représenté est concret, mais son utilisateur doit continuer à raisonner à partir des services annoncés, sans dépendre de ses champs internes.

Nous étudierons trois familles de représentations : les tableaux, les structures dynamiques chaînées et les représentations mixtes fondées sur le hachage. Les exemples montreront comment conserver les propriétés des collections tout en tenant compte de la mémoire et du coût des opérations. Les algorithmes resteront en pseudocode ; leur traduction en C constitue une étape ultérieure.

5.2 La conception préliminaire des collections

Une opération mathématique associe des valeurs à une valeur résultat. Sur une machine, créer systématiquement une nouvelle collection peut demander des copies et des allocations. Une conception muable permet de modifier une variable existante. Ce choix ne change pas les propriétés à respecter, mais modifie la manière d'exprimer et d'enchaîner les opérations.

Les fonctions ci-dessous construisent une collection ou calculent un résultat sans modifier leurs paramètres. Les procédures réalisent les modifications, indiquées par le mode entrée-sortie. Par exemple, l'opération mathématique $empiler(p, e)$ devient un appel de procédure qui modifie p. La collection initiale n'est alors pas conservée automatiquement.

5.2.1 Pile

Pour la pile nous avons les signatures suivantes :

— **fonction** pile () **:** Pile
— **fonction** estVide (p : Pile) **: Booleen**

— **procédure** empiler (**E/S** p : Pile, **E** e : Element)
— **procédure** depiler (**E/S** p : Pile)

⌊**précondition(s)** non estVide(p)

— **fonction** obtenirElement (p : Pile) **:** Element

⌊**précondition(s)** non estVide(p)

5.2.2 Pile

Pour la file nous avons les signatures suivantes :
— **fonction** file () **:** File
— **fonction** estVide (f : File) **: Booleen**
— **procédure** enfiler (**E/S** f : File, **E** e : Element)
— **procédure** defiler (**E/S** f : File)

⌊**précondition(s)** non estVide(f)

— **fonction** obtenirElement (f : File) **:** Element

⌊**précondition(s)** non estVide(f)

Ces signatures sont proches, mais les résultats attendus restent différents : la pile rend accessible le dernier élément ajouté, la file le plus ancien encore présent. C'est la spécification qui distingue ces comportements.

5.2.3 Liste

Pour une liste, la position d'insertion est fournie par l'appelant. Les positions vont de 1 à la longueur, avec une position supplémentaire en fin de liste pour l'insertion.
— **fonction** liste () **:** Liste
— **fonction** estVide (l : Liste) **: Booleen**
— **fonction** longueur (l : Liste) **: Naturel**
— **procédure** inserer (**E/S** l : Liste, **E** i : **Naturel**, e : Element)

⌊**précondition(s)** $1 \leq i \leq longueur(l) + 1$

— **procédure** supprimer (**E/S** l : Liste, **E** i : **Naturel**)

⌊**précondition(s)** $1 \leq i \leq longueur(l)$

— **fonction** obtenirElement (l : Liste, i : **Naturel**) **:** Element

⌊**précondition(s)** $1 \leq i \leq longueur(l)$

5.2.4 Listes ordonnées

Pour une liste ordonnée, l'insertion doit choisir elle-même une position qui conserve l'ordre. Le type des éléments possède donc un ordre total. Les doublons restent autorisés.
— **fonction** listeOrdonnee () **:** ListeOrdonnee
— **fonction** estVide (l : ListeOrdonnee) **: Booleen**
— **fonction** longueur (l : ListeOrdonnee) **: Naturel**
— **procédure** inserer (**E/S** l : ListeOrdonnee, **E** e : Element)
— **procédure** supprimer (**E/S** l : ListeOrdonnee, **E** i : **Naturel**)

⌊**précondition(s)** $1 \leq i \leq longueur(l)$

— **fonction** obtenirElement (l : ListeOrdonnee, i : **Naturel**) **:** Element

⌊**précondition(s)** $1 \leq i \leq longueur(l)$

5.2.5 Ensemble

L'ajout et le retrait d'un élément modifient un ensemble. En revanche, l'union, l'intersection et la soustraction construisent un ensemble résultat sans modifier les deux ensembles reçus.

— **fonction** ensemble **() :** Ensemble
— **procédure** ajouter **(E/S** s : Ensemble, **E** e : Element**)**
— **procédure** retirer **(E/S** s : Ensemble, **E** e : Element**)**
— **fonction** estPresent **(**s : Ensemble, e : Element**) : Booleen**
— **fonction** cardinalite **(**s : Ensemble**) : Naturel**
— **fonction** union **(**s1,s2 : Ensemble**) :** Ensemble
— **fonction** intersection **(**s1,s2 : Ensemble**) :** Ensemble
— **fonction** soustraction **(**s1,s2 : Ensemble**) :** Ensemble

5.2.6 Dictionnaire

L'ajout à un dictionnaire crée une association ou remplace la valeur d'une clé existante. Les noms `Cle` et `obtenirCles` reprennent ceux de sa spécification.

— **fonction** dictionnaire **() :** Dictionnaire
— **procédure** ajouter **(E/S** d : Dictionnaire,
E c : Cle, v : Valeur**)**
— **procédure** retirer **(E/S** d : Dictionnaire, **E** c : Cle**)**
— **fonction** estPresent **(**d : Dictionnaire, c : Cle**) : Booleen**
— **fonction** obtenirValeur **(**d : Dictionnaire, c : Cle**) :** Valeur
⌊**précondition(s)** estPresent(d,c)
— **fonction** obtenirCles **(**d : Dictionnaire**) :** Ensemble<Cle>
— **fonction** obtenirValeurs **(**d : Dictionnaire**) :** Liste<Valeur>

5.2.7 Tas et arbres binaires de recherche

Les collections hiérarchiques offrent également des opérations de création, de recherche, d'insertion et de suppression.

Tas

Pour un tas, l'élément observé à la racine est un extremum, selon la convention choisie.

— **fonction** tas **() :** Tas
— **fonction** estVide **(**a : Tas**) : Booleen**
— **procédure** inserer **(E/S** a : Tas, **E** e : Element**)**
— **procédure** supprimer **(E/S** a : Tas, **E** e : Element**)**
— **fonction** estPresent **(**a : Tas, e : Element**) : Booleen**
— **fonction** obtenirElement **(**a : Tas**) :** Element
⌊**précondition(s)** non estVide(a)
— **fonction** obtenirFilsGauche **(**a : Tas**) :** Tas
⌊**précondition(s)** non estVide(a)
— **fonction** obtenirFilsDroit **(**a : Tas**) :** Tas
⌊**précondition(s)** non estVide(a)

Arbre binaire de recherche

Pour l'ABR étudié ici, les éléments sont distincts. L'insertion suppose l'absence de l'élément, conformément à ce choix. La suppression d'un élément absent laisse l'arbre inchangé.

— **fonction** aBR **() :** ABR
— **fonction** estVide **(**a : ABR**) : Booleen**

— **procédure** inserer **(E/S** a : ABR, **E** e : Element**)**
  |**précondition(s)** non estPresent(a,e)
— **procédure** supprimer **(E/S** a : ABR, **E** e : Element**)**
— **fonction** estPresent (a : ABR, e : Element**) : Booleen**
— **fonction** obtenirElement (a : ABR**) :** Element
  |**précondition(s)** non estVide(a)
— **fonction** obtenirFilsGauche (a : ABR**) :** ABR
  |**précondition(s)** non estVide(a)
— **fonction** obtenirFilsDroit (a : ABR**) :** ABR
  |**précondition(s)** non estVide(a)

5.2.8 Utilisation et contraintes de réalisation

Une variable de collection précise le type effectif de ses éléments, par exemple `Liste<Entier>`. Si plusieurs collections proposent une opération de même nom, on qualifie l'appel.

L'appel `ArbreBinaire.estVide(a)` désigne ainsi le test du vide d'un arbre binaire, sans imposer à l'appelant de connaître sa représentation.

La mémoire étant finie, les opérations qui ajoutent des éléments peuvent rencontrer une limite de capacité ou un échec d'allocation. Les signatures précédentes décrivent les services attendus ; la conception concrète doit préciser le traitement de ces limites. Nous indiquerons explicitement la capacité disponible dans les exemples à tableaux et supposerons la réussite des allocations dans les exemples chaînés, comme au chapitre précédent.

5.3 Les représentations à l'aide de tableaux

Un tableau réserve des emplacements pour les éléments. Pour représenter une collection dont la taille varie, on lui associe le nombre d'éléments significatifs. Les cases occupées sont ici contiguës et commencent à l'indice 1. L'invariant de représentation est $0 \leq nbElements \leq MAX$, et seules les cases d'indices 1 à $nbElements$ appartiennent à la collection.

5.3.1 Une pile dans un tableau

Le sommet est placé dans la dernière case significative. Empiler prolonge la partie utilisée et dépiler la raccourcit, sans déplacer les éléments qui restent dans la pile.

Type Pile = **Structure**
  lesElements : **Tableau[**1..MAX**] de** Element
  nbElements : **Naturel**
finstructure

Avec cette conception, initialiser le compteur suffit pour représenter le vide. Les autres cases ne doivent pas être consultées tant qu'elles n'ont pas reçu un élément. L'empilement nécessite une case libre ; cette condition provient de la représentation bornée et s'ajoute aux contraintes de la collection abstraite. Dépiler ne remet pas nécessairement l'ancienne case à zéro : elle cesse simplement d'être significative. Sous l'hypothèse d'opérations élémentaires de coût constant, ces trois opérations sont en $O(1)$. Elles réalisent bien le comportement LIFO puisque la dernière valeur ajoutée est celle qui est consultée ou retirée.

Ce qui nous donnne les algorithmes suivants :

fonction pile **() :** Pile
  Déclaration resultat : Pile
debut
  resultat.nbElements ← 0
  retourner resultat

fin
fonction estVide (p : Pile) **: Booleen**
debut
  retourner p.nbElements = 0
fin
procédure empiler (**E/S** p : Pile, **E** e : Element)
  ⌊**précondition(s)** $p.nbElements < MAX$
debut
  p.nbElements ← p.nbElements + 1
  p.lesElements[p.nbElements] ← e
fin
procédure depiler (**E/S** p : Pile)
  ⌊**précondition(s)** non estVide(p)
debut
  p.nbElements ← p.nbElements - 1
fin
fonction obtenirElement (p : Pile) **:** Element
  ⌊**précondition(s)** non estVide(p)
debut
  retourner p.lesElements[p.nbElements]
fin

5.3.2 Une liste dans un tableau

La liste utilise la même forme d'enregistrement, mais permet un accès et une modification à une position quelconque. L'ordre des cases significatives est celui des éléments de la liste.

Type Liste = **Structure**
  lesElements : **Tableau[**1..MAX**] de** Element
  nbElements : **Naturel**
finstructure

L'insertion réserve une place à l'indice demandé en décalant les éléments suivants d'une case vers la droite. La suppression comble le trou laissé par l'élément retiré en décalant la suite vers la gauche.

fonction liste () **:** Liste
  Déclaration resultat : Liste
debut
  resultat.nbElements ← 0
  retourner resultat
fin
fonction estVide (l : Liste) **: Booleen**
debut
  retourner l.nbElements = 0
fin
fonction longueur (l : Liste) **: Naturel**
debut
  retourner l.nbElements
fin
fonction obtenirElement (l : Liste, indice : **Naturel) :** Element
  ⌊**précondition(s)** $1 \leq indice \leq longueur(l)$
debut
  retourner l.lesElements[indice]

fin
procédure inserer **(E/S** l : Liste, **E** indice : **Naturel**, e : Element**)**
⌊**précondition(s)** $1 \leq indice \leq longueur(l) + 1$ et $l.nbElements < MAX$
debut
decalerVersLaDroite(l,indice)
l.nbElements ← l.nbElements + 1
l.lesElements[indice] ← e
fin
procédure supprimer **(E/S** l : Liste, **E** indice : **Naturel)**
⌊**précondition(s)** $1 \leq indice \leq longueur(l)$
debut
decalerVersLaGauche(l,indice+1)
l.nbElements ← l.nbElements - 1
fin

Pour décaler vers la droite, il faut commencer par la fin. Une progression dans l'autre sens écraserait une valeur avant de l'avoir déplacée. Pour le décalage vers la gauche, on procède au contraire dans l'ordre croissant. Ces procédures auxiliaires ne changent pas le compteur, dont la mise à jour revient aux opérations précédentes. Lorsque l'intervalle à parcourir est vide, la boucle n'exécute aucune itération. L'insertion à la fin et la suppression du dernier élément n'effectuent ainsi aucun décalage. À l'autre extrême, une modification au début peut déplacer tous les autres éléments : son coût est en $O(n)$ pour une liste de longueur n. L'accès par position et la longueur sont en $O(1)$.

procédure decalerVersLaDroite **(E/S** l : Liste, **E** indice : **Naturel)**
⌊**précondition(s)** $1 \leq indice \leq l.nbElements + 1$ et $l.nbElements < MAX$
Déclaration i : **Naturel**
debut
pour i ←l.nbElements **à** indice **pas de** -1 **faire**
l.lesElements[i+1] ← l.lesElements[i]
finpour
fin
procédure decalerVersLaGauche **(E/S** l : Liste, **E** indice : **Naturel)**
⌊**précondition(s)** $2 \leq indice \leq l.nbElements + 1$
Déclaration i : **Naturel**
debut
pour i ←indice **à** l.nbElements **pas de** 1 **faire**
l.lesElements[i-1] ← l.lesElements[i]
finpour
fin

5.3.3 Portée et limites de la représentation

Le tableau permet des accès directs et des algorithmes simples, mais réserve MAX emplacements même lorsque la collection est presque vide. Il faut décider comment signaler que cette capacité est atteinte. Ajouter une précondition de capacité, comme ci-dessus, rend explicite la limite ; une interface utilisable sans cette connaissance devrait prévoir un mécanisme de signalement de l'échec.

Un tableau peut aussi coder une organisation hiérarchique. Un tas binaire rempli en largeur se représente en plaçant la racine à l'indice 1, puis les fils du nœud i aux indices $2i$ et $2i+1$, lorsqu'ils existent. La représentation d'une hiérarchie n'exige donc pas toujours des pointeurs. Elle demande une convention de correspondance entre indices et relations entre nœuds.

5.4 Les représentations à l'aide de listes chaînées

Les opérations de `ListeChainee` permettent de construire des collections sans réserver à l'avance un tableau de capacité maximale. Le nombre de nœuds suit celui des éléments, sous réserve de mémoire disponible. La conception doit préciser ce que représente la tête de la chaîne, quelles informations supplémentaires sont conservées et quels invariants les opérations doivent préserver.

5.4.1 Pile : placer le sommet en tête

On peut définir directement le type `Pile` à partir de `ListeChainee`. Le premier nœud contient le sommet ; les suivants contiennent les éléments dans l'ordre inverse de leurs empilements. Les opérations de pile se ramènent alors aux opérations de tête.

Type Pile = ListeChainee
fonction pile **() :** Pile
debut
  retourner ListeChainee.listeVide()
fin
fonction estVide (p : Pile**) : Booleen**
debut
  retourner ListeChainee.estVide(p)
fin
procédure empiler **(E/S** p : Pile, **E** e : Element**)**
debut
  ListeChainee.ajouter(p,e)
fin
procédure depiler **(E/S** p : Pile**)**
  ⌊**précondition(s)** non estVide(p)
debut
  ListeChainee.supprimerTete(p)
fin
fonction obtenirElement (p : Pile**) :** Element
  ⌊**précondition(s)** non estVide(p)
debut
  retourner ListeChainee.obtenirElement(p)
fin

Les préfixes séparent les opérations de la collection de celles de sa représentation. Aucune opération ne parcourt la chaîne : consultation, empilement et dépilement sont en $O(1)$, en considérant ici l'allocation et la libération d'un nœud comme des opérations de coût constant.

5.4.2 File : conserver les deux extrémités

Pour une file, l'ajout et le retrait ont lieu à des extrémités différentes. La tête de la chaîne désigne l'élément à défiler. Un deuxième pointeur conserve l'adresse du dernier nœud pour éviter de parcourir toute la file lors d'un ajout.

Type File = **Structure**
  debut : ListeChainee
  fin : ListeChainee
finstructure

Lorsque la file est vide, les deux champs valent **null**. Sinon, `fin` désigne le dernier nœud accessible depuis `debut`, et son suivant est vide. Les deux champs sont des accès à la même chaîne, et non deux listes à libérer séparément.

L'enfilement crée un nœud isolé. Si la file était vide, ce nœud devient à la fois le début et la fin. Sinon, il devient le suivant de l'ancienne fin. Les algorithmes suivants explicitent ces deux cas.

```
fonction file () : File
   Déclaration  r : File
debut
   r.debut ← ListeChainee.listeVide()
   r.fin ← ListeChainee.listeVide()
   retourner r
fin
fonction estVide (f : File) : Booleen
debut
   retourner ListeChainee.estVide(f.debut)
fin
procédure enfiler (E/S f : File, E e : Element)
   Déclaration  nouveau : ListeChainee
debut
   nouveau ← ListeChainee.listeVide()
   ListeChainee.ajouter(nouveau,e)
   si estVide(f) alors
      f.debut ← nouveau
   sinon
      ListeChainee.fixerListeSuivante(f.fin,nouveau)
   finsi
   f.fin ← nouveau
fin
```

Défiler retire la tête. Lorsque cette tête était le dernier nœud, le pointeur de fin doit être remis à vide pour ne pas conserver une adresse libérée.

```
procédure defiler (E/S f : File)
   ⌊précondition(s)  non estVide(f)
debut
   ListeChainee.supprimerTete(f.debut)
   si estVide(f) alors
      f.fin ← ListeChainee.listeVide()
   finsi
fin
fonction obtenirElement (f : File) : Element
   ⌊précondition(s)  non estVide(f)
debut
   retourner ListeChainee.obtenirElement(f.debut)
fin
```

Les opérations usuelles de la file sont ainsi en $O(1)$. Sans le champ `fin`, l'ajout en fin exigerait un parcours en $O(n)$. Ce gain dépend du maintien de l'invariant après chaque modification.

5.4.3 Listes et listes ordonnées : conserver la longueur

Une liste avec accès par position peut être représentée par une chaîne et un compteur. L'ordre des nœuds est celui des positions de la liste.

```
Type Liste = Structure
   lesElements : ListeChainee
   nbElements : Naturel
finstructure
```

Le compteur est initialisé à zéro et doit toujours être égal au nombre de nœuds accessibles. Consulter la longueur est alors en $O(1)$. Obtenir l'élément de position i demande d'avancer $i-1$ fois depuis la tête. Pour une insertion ou une suppression hors de la tête, on rejoint le prédécesseur afin de modifier son lien suivant. Les opérations en tête restent particulières puisqu'elles changent le point d'entrée lui-même.

Une insertion réussie incrémente le compteur, une suppression réussie le décrémente. Le simple changement de liens est local, mais la recherche de la position peut parcourir toute la chaîne. Le coût maximal de l'accès, de l'insertion et de la suppression par position est donc en $O(n)$.

La représentation d'une `ListeOrdonnee` utilise les mêmes champs. Son invariant ajoute que les valeurs sont en ordre croissant. L'insertion parcourt les nœuds pour trouver une position compatible avec cet ordre, puis raccorde le nouveau nœud. Le retrait par position et la consultation sont ceux d'une liste ; ils ne changent pas l'ordre relatif des éléments conservés. Chaque insertion augmente la longueur, y compris si la valeur était déjà présente.

5.4.4 Ensemble : garantir l'absence de doublons

Un ensemble peut lui aussi contenir une chaîne et un compteur :

Type Ensemble = **Structure**
lesElements : ListeChainee
nbElements : **Naturel**
finstructure

La chaîne possède un ordre de parcours, mais cet ordre ne fait pas partie de la signification de l'ensemble. L'invariant essentiel est l'absence de deux nœuds contenant des éléments égaux. Avant d'ajouter en tête, on recherche donc l'élément : on ne crée un nœud et on n'incrémente le compteur que s'il était absent. De même, un retrait ne décrémente le compteur que si un élément a effectivement été retiré.

La recherche est un parcours, la cardinalité une lecture du compteur. L'ajout n'est pas globalement en temps constant : le raccordement l'est, mais le test préalable d'appartenance est en $O(n)$ au pire.

L'union crée un ensemble vide puis y ajoute les éléments des deux ensembles reçus. L'opération d'ajout empêche les doublons. La soustraction crée un ensemble contenant les éléments du premier qui sont absents du second. Pour l'intersection, on peut ajouter dans un résultat neuf les éléments de chaque ensemble présents dans l'autre ; l'absence de doublons rend ce double parcours correct. Un seul de ces parcours suffit toutefois à obtenir le résultat, puisque tout élément de l'intersection appartient aux deux ensembles.

Les ensembles résultats doivent disposer de leurs propres nœuds. Une simple affectation du pointeur de tête partagerait les nœuds au lieu de construire une collection indépendante. Le choix de copier ou de partager les ressources éventuellement contenues dans les éléments demande, lui, un contrat supplémentaire. La libération d'une collection chaînée doit toujours rester cohérente avec ces choix de copie.

5.5 La conception d'un arbre binaire de recherche

Un ABR utilise la structure `ArbreBinaire`, mais impose un ordre entre ses éléments. Pour chaque nœud, toutes les valeurs du sous-arbre gauche sont strictement inférieures à la sienne et celles du sous-arbre droit strictement supérieures. Les deux sous-arbres vérifient la même propriété. Nous conservons ici des éléments distincts.

Type ABR = ArbreBinaire

La création du vide, son test et les observations de la racine et des fils utilisent directement les opérations de la structure binaire. Les opérations de modification doivent, elles, conserver l'ordre. Dans les algorithmes suivants, les accès aux fils et les modifications de liens sont ceux d'`ArbreBinaire` ; `aBR()` fournit un arbre vide. Le constructeur est appelé dans l'ordre `ajouterRacine(fg,fd,e)`.

5.5.1 Recherche et insertion

La recherche compare l'élément à la racine. S'ils sont différents, l'ordre permet d'écarter entièrement un sous-arbre. Un arbre vide termine la recherche par un échec.

```
fonction estPresent (a : ABR, e : Element) : Booleen
debut
   si estVide(a) alors
      retourner FAUX
   sinon
      si e = obtenirElement(a) alors
         retourner VRAI
      sinon
         si e < obtenirElement(a) alors
            retourner estPresent(obtenirFilsGauche(a),e)
         sinon
            retourner estPresent(obtenirFilsDroit(a),e)
         finsi
      finsi
   finsi
fin
```

L'insertion suit le même chemin jusqu'à une place vide, où elle crée une feuille. Sa précondition exclut les doublons. L'appel récursif peut changer la racine du sous-arbre traité : sa nouvelle adresse est donc réaffectée au lien du parent.

```
procédure inserer (E/S a : ABR, E e : Element)
   ⌊précondition(s)   non estPresent(a,e)
   Déclaration   temp : ABR
debut
   si estVide(a) alors
      a ← ajouterRacine(aBR(),aBR(),e)
   sinon
      si e < obtenirElement(a) alors
         temp ← obtenirFilsGauche(a)
         inserer(temp,e)
         fixerFilsGauche(a,temp)
      sinon
         temp ← obtenirFilsDroit(a)
         inserer(temp,e)
         fixerFilsDroit(a,temp)
      finsi
   finsi
fin
```

L'ordre strict reste vérifié : la nouvelle valeur se trouve dans tous les sous-arbres imposés par les comparaisons effectuées. La précondition exprime une obligation de l'appelant ; elle ne signifie pas que chaque appel récursif relance automatiquement une recherche complète.

5.5.2 Supprimer un élément

On recherche d'abord le nœud selon les mêmes comparaisons. S'il n'existe pas, rien ne change. Lorsqu'il est trouvé, trois cas se distinguent :

1. une feuille peut être libérée et remplacée par l'arbre vide ;
2. un nœud à un seul fils est remplacé par ce fils ;

3. un nœud à deux fils est remplacé par une valeur compatible avec les deux sous-arbres, par exemple le plus grand élément du sous-arbre gauche.

Ce dernier élément s'obtient en suivant les fils droits jusqu'au dernier nœud non vide. La fonction suivante renvoie le nœud qui le contient.

fonction lePlusGrand **(**a : ABR**) :** ABR

⌊**précondition(s)** non estVide(a)

debut
  si estVide(obtenirFilsDroit(a)) **alors**
    retourner a
  sinon
    retourner lePlusGrand(obtenirFilsDroit(a))
  finsi
fin

Pour garder lisible la suppression, la procédure privée `supprimerRacineABR` réalise les trois cas. Elle utilise `ArbreBinaire.supprimerRacine`, qui libère uniquement la racine et rend ses deux fils dans les paramètres de sortie. Nous fixons ici explicitement l'arbre à vide lorsqu'aucun fils ne reste.

procédure supprimerRacineABR **(E/S** a : ABR**)**

⌊**précondition(s)** non estVide(a)

Déclaration g,d : ABR
nouvelleValeur : Element

debut
  ArbreBinaire.supprimerRacine(a,g,d)
  si estVide(g) et estVide(d) **alors**
    a ← aBR()
  sinon
    si estVide(g) ou estVide(d) **alors**
      si estVide(g) **alors**
        a ← d
      sinon
        a ← g
      finsi
    sinon
      nouvelleValeur ← obtenirElement(lePlusGrand(g))
      supprimer(g,nouvelleValeur)
      a ← ajouterRacine(g,d,nouvelleValeur)
    finsi
  finsi
fin

Dans le cas à deux fils, la valeur choisie est retirée de l'ancien sous-arbre gauche afin de ne pas créer de doublon. Elle est supérieure à toutes les valeurs qui y restent et inférieure à toutes celles du sous-arbre droit. La reconstruction conserve donc l'invariant d'ordre. Cette version libère l'ancienne racine puis en alloue une nouvelle ; comme précédemment, les allocations sont supposées réussir.

procédure supprimer **(E/S** a : ABR, **E** e : Element**)**

Déclaration temp : ABR

debut
  si non estVide(a) **alors**
    si $e < obtenirElement(a)$ **alors**
      temp ← obtenirFilsGauche(a)
      supprimer(temp,e)

```
            fixerFilsGauche(a,temp)
        sinon
            si e > obtenirElement(a) alors
                temp ← obtenirFilsDroit(a)
                supprimer(temp,e)
                fixerFilsDroit(a,temp)
            sinon
                supprimerRacineABR(a)
            finsi
        finsi
    finsi
fin
```

Les deux procédures se complètent : la première traite une racine déjà identifiée, la seconde trouve le nœud et rétablit les liens au retour. Le remplacement par le plus petit élément du sous-arbre droit donnerait une construction symétrique.

5.5.3 La forme de l'arbre dépend des insertions

Les insertions successives de 1, 2, 3 produisent une chaîne vers la droite, tandis que celles de 2, 1, 3 donnent une racine à deux fils. Les éléments sont les mêmes, mais la hauteur et les chemins de recherche diffèrent.

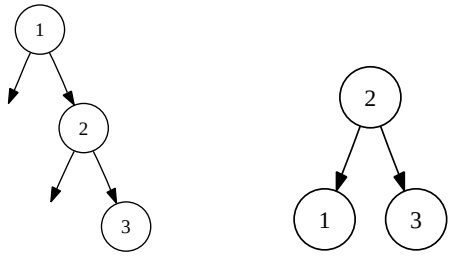


FIGURE 5.1 – ABR obtenus par les insertions de $1, 2, 3$ puis de $2, 1, 3$.

Recherche, insertion et suppression suivent des chemins dont la longueur dépend de la hauteur h : leur coût est en $O(h+1)$. Dans un arbre réduit à une chaîne, cette hauteur peut atteindre $n-1$. L'invariant d'ordre seul ne garantit donc pas un coût logarithmique. Pour obtenir cette garantie, il faut aussi contrôler la forme de l'arbre.

5.6 Maintenir l'équilibre : les arbres AVL

Un arbre AVL est un arbre binaire de recherche dans lequel, à chaque nœud, les hauteurs des sous-arbres gauche et droit diffèrent d'au plus une unité. Cet invariant complète l'ordre entre les éléments. Avec la convention d'une hauteur -1 pour l'arbre vide, une feuille a une hauteur nulle. La condition s'écrit, pour tout nœud a :

$$|hauteur(obtenirFilsGauche(a)) - hauteur(obtenirFilsDroit(a))| \leq 1.$$

Elle limite la hauteur à un ordre logarithmique en nombre de nœuds. Une insertion peut rompre cet équilibre ; les rotations réorganisent alors localement les liens sans changer l'ordre des éléments.

5.6.1 Rotations simples et doubles

Une rotation simple à droite fait remonter le fils gauche A d'une racine B. Le nœud B devient le fils droit de A, et l'ancien sous-arbre droit de A devient le sous-arbre gauche de B.

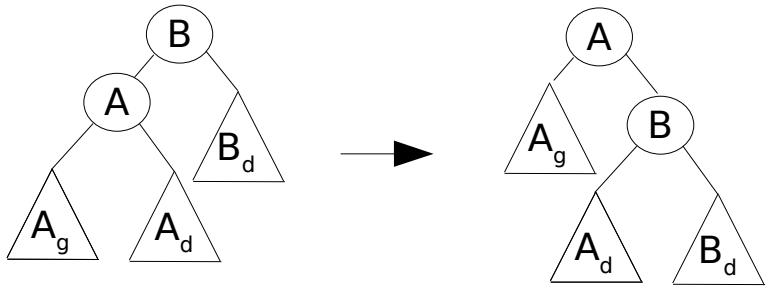


FIGURE 5.2 – Rotation simple à droite autour de B.

Les éléments de A_d sont supérieurs à A et inférieurs à B : leur nouvelle position respecte donc l'ordre de recherche. Les sous-arbres A_g et B_d ne changent pas de contenu. Une rotation ne copie ni les éléments ni les sous-arbres ; elle change leurs liens et la racine locale. La rotation à gauche est symétrique.

Lorsque le déséquilibre est du côté gauche mais que ce fils gauche est plus haut à droite, une rotation simple ne suffit pas. Une double rotation à droite effectue d'abord une rotation à gauche sur le fils gauche, puis une rotation à droite sur la racine.

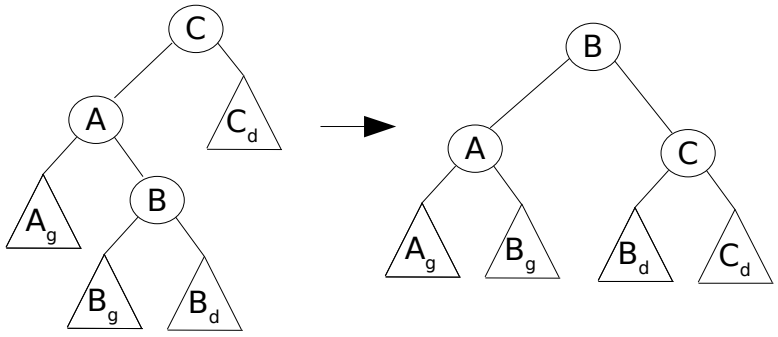


FIGURE 5.3 – Double rotation à droite : le nœud intermédiaire B devient la racine locale.

Dans la figure, l'ordre des racines est $A < B < C$. Après réorganisation, B possède A à gauche et C à droite. Les sous-arbres B_g et B_d sont rattachés respectivement à A et à C, sans modifier l'ordre des valeurs. Les situations symétriques se traitent par une double rotation à gauche.

5.6.2 Insertion et rééquilibrage

L'insertion commence comme dans un ABR : on descend vers une place vide et on crée une feuille. Au retour de l'appel récursif, le lien modifié est rétabli, puis on vérifie l'équilibre de la racine courante. L'élément inséré est supposé absent. Les procédures privées `reequilibrerApresGauche` et `reequilibrerApresDroite` explicitent les deux traitements.

procédure inserer **(E/S** a : ABR, **E** e : Element**)**

⌊**précondition(s)** a est AVL et non estPresent(a,e)

Déclaration temp : ABR

debut

si estVide(a) **alors**

a ← ajouterRacine(aBR(),aBR(),e)

```
    sinon
        si e < obtenirElement(a) alors
            temp ← obtenirFilsGauche(a)
            inserer(temp,e)
            fixerFilsGauche(a,temp)
            reequilibrerApresGauche(a)
        sinon
            temp ← obtenirFilsDroit(a)
            inserer(temp,e)
            fixerFilsDroit(a,temp)
            reequilibrerApresDroite(a)
        finsi
    finsi
fin
```

Après une insertion à gauche, un excès de hauteur de ce côté est traité en examinant les deux fils du sous-arbre gauche. Les variables g et d ci-dessous sont des accès temporaires aux fils, sans copie des nœuds.

```
procédure reequilibrerApresGauche (E/S a : ABR)
    ⌊précondition(s)   non estVide(a)
    Déclaration   g,d : ABR
debut
    g ← obtenirFilsGauche(a)
    d ← obtenirFilsDroit(a)
    si hauteur(g) > hauteur(d) + 1 alors
        si hauteur(obtenirFilsGauche(g)) ≥ hauteur(obtenirFilsDroit(g)) alors
            faireSimpleRotationDroite(a)
        sinon
            faireDoubleRotationDroite(a)
        finsi
    finsi
fin
```

Le test d'excès de hauteur garantit que g n'est pas vide avant que ses fils soient consultés. Après une insertion à droite, les rôles sont inversés :

```
procédure reequilibrerApresDroite (E/S a : ABR)
    ⌊précondition(s)   non estVide(a)
    Déclaration   g,d : ABR
debut
    g ← obtenirFilsGauche(a)
    d ← obtenirFilsDroit(a)
    si hauteur(d) > hauteur(g) + 1 alors
        si hauteur(obtenirFilsGauche(d)) ≤ hauteur(obtenirFilsDroit(d)) alors
            faireSimpleRotationGauche(a)
        sinon
            faireDoubleRotationGauche(a)
        finsi
    finsi
fin
```

Ces procédures sont destinées au contexte de l'insertion dans un AVL : les sous-arbres ont déjà été rééquilibrés et leur hauteur n'a changé que de façon limitée. Elles ne constituent pas un algorithme de rééquilibrage d'un ABR arbitraire. Les procédures de rotation modifient le paramètre a pour qu'il désigne la nouvelle racine locale.

Pour une insertion en temps logarithmique, la hauteur doit être disponible sans reparcourir chaque sous-arbre. On peut la mémoriser dans les nœuds et la mettre à jour après les modifications et les rotations. Si `hauteur` recalcule récursivement toute la hauteur à chaque appel, le seul équilibre de l'arbre ne suffit pas à garantir ce coût pour l'algorithme écrit.

5.6.3 Exercice : suppression dans un AVL

Concevoir la suppression en reprenant les cas de l'ABR, puis en rétablissant l'équilibre sur le chemin de retour. Contrairement à l'insertion, la suppression peut demander des rééquilibrages à plusieurs niveaux. Les rotations doivent préserver l'ordre de recherche et tenir compte des hauteurs des sous-arbres restants.

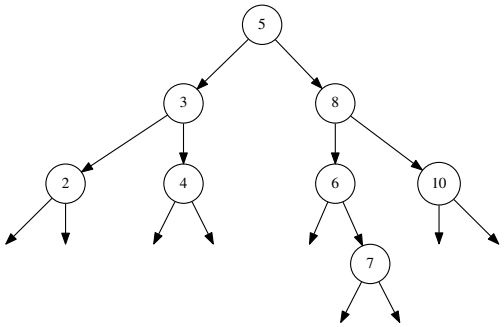


FIGURE 5.4 – Arbre AVL pour l'étude de la suppression.

Sur la figure 5.4, étudier notamment la suppression de 5, qui possède deux fils. Choisir une valeur de remplacement, suivre la suppression de son ancien nœud et contrôler les hauteurs à chaque retour. Descendre un nœud par des rotations ne garantit pas à lui seul l'équilibre final : chaque transformation doit être justifiée par les invariants et accompagnée des mises à jour nécessaires.

5.7 Représenter un arbre n-aire par un arbre binaire

Un arbre n-aire peut avoir un nombre variable de fils pour chaque nœud. Il est pourtant possible de le représenter avec seulement deux liens par nœud, en changeant leur interprétation. Le lien gauche donne accès au premier fils ; le lien droit donne accès au frère suivant, c'est-à-dire au prochain fils du même parent.

Type Arbre = ArbreBinaire

Cette déclaration décrit une représentation, et non une identité de sens entre les opérations des deux types. Le fils droit dans la structure binaire ne représente pas un fils supplémentaire dans l'arbre n-aire : il représente un frère. Les opérations publiques doivent traduire cette convention pour que l'utilisateur continue à manipuler l'arbre n-aire.

Dans la figure 5.5, les trois fils de la racine sont retrouvés en suivant une fois le lien gauche, puis les liens droits successifs. Le premier de ces fils possède lui-même deux fils, représentés par le même mécanisme. Un lien gauche nul indique l'absence de fils ; un lien droit nul termine une liste de frères. La racine de l'arbre entier n'a pas de frère, donc son lien droit est nul.

Cette organisation conserve l'ordre des fils sans réserver un tableau de taille maximale dans chaque nœud. Accéder au premier fils est immédiat ; accéder au i-ème fils demande ensuite de suivre $i - 1$ liens droits. Un parcours de la structure binaire doit être interprété avec soin : son

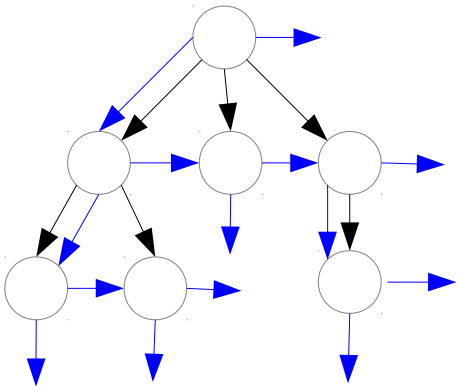


FIGURE 5.5 – Relations de filiation en noir et liens « premier fils, frère suivant » en bleu.

sous-arbre droit peut contenir des frères qui n'appartiennent pas au sous-arbre n-aire du nœud courant.

5.7.1 Exercice de conception

Donner les algorithmes des fonctions et procédures de l'arbre n-aire avec cette représentation. Pour consulter les fils, distinguer le premier lien vers un fils des liens entre frères. Pour insérer ou supprimer un fils, préserver l'accès aux frères suivants et préciser les effets sur le premier fils du parent.

La suppression d'un sous-arbre mérite une attention particulière. Appliquer sans adaptation une suppression binaire au pointeur du fils pourrait également libérer ses frères. Il faut d'abord distinguer la partie à retirer de la chaîne des frères à conserver, puis traiter les descendants. Cette différence illustre pourquoi la correspondance entre opérations ne se réduit pas à renommer les champs d'une structure.

5.8 Les représentations à l'aide de tables de hachage

La table de hachage étudiée au chapitre précédent associe un tableau à des listes chaînées. Elle convient aux collections dans lesquelles on recherche une information à partir de son contenu ou d'une clé, sans imposer un ordre de parcours. Le calcul de l'indice réduit la recherche à une liste, puis une comparaison identifie l'élément voulu parmi les collisions.

5.8.1 Représenter un dictionnaire

Pour un dictionnaire, chaque nœud conserve un couple clé-valeur. La fonction de hachage porte sur la clé : toutes les opérations concernant une même clé doivent rejoindre la même case. Deux clés égales doivent donc produire le même indice, sans que la réciproque soit exigée.

L'ajout calcule l'indice puis recherche la clé dans la liste associée. Si elle existe, sa valeur est remplacée ; sinon, un nouveau couple est inséré. Ajouter systématiquement un nœud créerait plusieurs associations pour une même clé et violerait la spécification. Le retrait recherche également la clé, puis supprime son nœud si elle est présente.

Le test de présence et l'obtention d'une valeur utilisent la même recherche. La seconde opération conserve sa précondition de présence. Pour obtenir les clés ou les valeurs, il faut parcourir toutes les cases et toutes leurs listes. On construit respectivement un ensemble de clés ou une liste de valeurs, avec une valeur par association. La représentation ne garantit aucun ordre particulier pour ces résultats.

5.8.2 Représenter un ensemble

Pour un ensemble, l'élément joue lui-même le rôle de clé ; il n'est pas nécessaire de conserver deux copies de la même information. L'ajout teste la présence dans la liste sélectionnée et n'insère que les éléments absents. Le retrait et l'appartenance suivent le même calcul d'indice. Un compteur peut fournir la cardinalité en temps constant, à condition de n'être modifié que lors des ajouts ou retraits effectifs.

Les opérations ensemblistes peuvent être construites par parcours et tests d'appartenance, comme avec une liste chaînée. La différence porte sur le coût des recherches, réparties entre les cases de la table. Les propriétés d'unicité et d'indépendance des ensembles résultats restent nécessaires.

5.8.3 Conditions des performances attendues

Pour n éléments et k cases, une bonne répartition donne des listes de longueur moyenne voisine de n/k. Sous l'hypothèse d'un calcul de hachage et de comparaisons en temps constant, le coût attendu d'une recherche est en $O(1 + n/k)$. Le terme constant tient compte de l'accès à la case, même si la liste est vide. Maintenir n/k borné permet donc un coût attendu constant, mais pas une garantie sur chaque recherche.

Dans le pire cas, tous les éléments aboutissent à la même case et la recherche redevient linéaire. De plus, si le calcul du hachage dépend de la longueur de la clé, ce coût doit être ajouté. Le parcours complet, quant à lui, visite les k cases et les n nœuds : il est en $O(k + n)$ avec cette organisation.

Le choix du nombre de cases et de la fonction de hachage fait ainsi partie de la conception. Une table très grande réserve de nombreux pointeurs inutilisés ; une table trop petite allonge les listes. Il faut évaluer ce compromis en fonction des données et des opérations réellement utilisées.

5.9 Synthèse : choisir une représentation adaptée

Une même collection peut être réalisée par plusieurs structures. La spécification fixe le résultat des opérations ; la représentation et les algorithmes déterminent les moyens et les coûts nécessaires pour l'obtenir. La comparaison doit donc porter sur des opérations précises, avec des hypothèses explicites sur les représentations.

5.9.1 Comparer les collections linéaires

Le tableau suivant donne des bornes supérieures dans le pire cas pour n éléments. Les accès, comparaisons, copies d'éléments et opérations sur un nœud sont supposés de coût constant. La file à tableau mentionnée ici est une représentation circulaire avec indices de début et de fin : une réalisation qui décale tous les éléments à chaque retrait n'aurait pas les mêmes coûts.

Collection et opération	Tableau	Liste chaînée
Pile : consulter, empiler, dépiler	$O(1)$	$O(1)$
File : consulter, enfiler, défiler	$O(1)$	$O(1)$
Liste : consulter par position	$O(1)$	$O(n)$
Liste : insérer ou supprimer par position	$O(n)$	$O(n)$
Liste ordonnée : consulter par position	$O(1)$	$O(n)$
Liste ordonnée : insérer, supprimer	$O(n)$	$O(n)$

Pour la file chaînée, le coût constant suppose les deux pointeurs `debut` et `fin`. Pour les listes, un raccordement local ne dispense pas de rechercher la position. Certaines situations sont plus favorables que ces bornes : insérer en tête d'une chaîne ou à la fin d'un tableau disposant de place est en $O(1)$.

Les tableaux réservent une capacité maximale et facilitent les accès par position. Les listes chaînées adaptent leur nombre de nœuds au nombre d'éléments, mais consomment de la mémoire pour les liens et demandent des parcours. Conserver un compteur rend la longueur immédiate au prix d'une mise à jour à chaque modification.

5.9.2 Comparer les accès par valeur ou par clé

Pour les représentations non ordonnées par tableau ou liste, rechercher un élément ou une clé est linéaire au pire. L'ajout à un ensemble doit vérifier l'absence de doublon ; l'ajout à un dictionnaire doit déterminer s'il remplace une association existante. Ils incluent donc ce coût de recherche, même lorsque l'insertion matérielle est immédiate.

Représentation	Recherche	Ajout ou retrait
Tableau non ordonné	$O(n)$	$O(n)$
Liste chaînée non ordonnée	$O(n)$	$O(n)$
ABR quelconque	$O(h+1)$	$O(h+1)$
ABR équilibré	$O(\log(n+1))$	$O(\log(n+1))$
Table de hachage, coût attendu	$O(1+n/k)$	$O(1+n/k)$

Ici, h est la hauteur de l'ABR et k le nombre de cases de la table. Les coûts du hachage supposent une bonne répartition et un calcul de hachage constant ; son pire cas reste en $O(n)$. Les coûts des arbres équilibrés supposent une relation d'ordre total et une gestion efficace des informations nécessaires au rééquilibrage.

Un ABR peut donc représenter un ensemble, ou un dictionnaire en ordonnant les couples selon leur clé. En revanche, l'accès au i-ème élément d'une liste ordonnée n'est pas automatiquement logarithmique avec un simple ABR : la recherche par valeur et la recherche par rang sont des services différents. La représentation doit apporter les informations nécessaires au service que l'on souhaite accélérer.

5.9.3 Interpréter les coûts des structures hiérarchiques

Consulter une racine ou créer un nœud à partir de deux sous-arbres déjà construits est en $O(1)$. Supprimer seulement cette racine en conservant les fils est également local ; détruire tous les nœuds d'un arbre de taille n exige en revanche de les visiter et est en $O(n)$.

Un tas binaire complet peut donner accès à son extremum en $O(1)$. L'insertion et le retrait de cet extremum peuvent être en $O(\log(n+1))$ avec les algorithmes appropriés. Ce résultat ne s'applique pas sans précision à la suppression d'une valeur quelconque : si sa position n'est pas connue, il faut d'abord la rechercher. La propriété de tas n'offre pas l'orientation gauche-droite de la recherche dans un ABR.

5.9.4 Relier efficacité et respect de la spécification

Le choix d'une représentation dépend de la fréquence des opérations, du volume des données et de la mémoire disponible. Il doit aussi préciser la capacité, le partage éventuel des nœuds, la copie des résultats et les conditions de libération. Une représentation rapide ne convient pas si elle ne respecte pas l'ordre, l'unicité ou les préconditions attendues.

La conception établit ainsi un lien explicite entre chaque service et les opérations de la structure choisie. Les algorithmes doivent conserver les invariants après toute modification. Cette justification, associée à l'analyse des coûts, prépare une implantation dont le comportement et les limites sont connus avant la traduction dans un langage concret.

Chapitre 6

Implantation des structures de données dynamiques en C

6.1 De la conception à l'implantation

Les représentations étudiées au chapitre précédent donnent aux collections une organisation concrète et des algorithmes qui respectent leur spécification. Leur implantation en C demande de traduire les enregistrements en structures, les liens en pointeurs et les opérations en fonctions. Elle demande aussi de préciser qui alloue les objets, qui peut les modifier et qui les libère. Ces décisions conditionnent la validité du programme autant que les algorithmes.

Nous suivrons la réalisation d'une bibliothèque de polynômes. Une liste chaînée de monômes fournit une première solution, directement adaptée au problème. Nous en dégagerons ensuite une liste générique, utilisable avec d'autres types d'éléments. Les opérations de chaînage resteront les mêmes ; les opérations propres aux éléments seront confiées à des fonctions fournies par l'utilisateur.

Comme pour les TAD déjà implantés, le fichier *.h* déclare les types et les fonctions accessibles, tandis que le fichier *.c* contient leurs définitions. Les préfixes `M_`, `LCDM_`, `LC_` et `P_` distinguent respectivement les monômes, les listes de monômes, les listes génériques et les polynômes.

Le passage par valeur du C mérite une attention particulière. Copier une structure copie ses champs, mais copier un pointeur ne duplique pas l'objet qu'il désigne. De même, une fonction qui doit changer le pointeur de tête d'une liste reçoit l'adresse de ce pointeur. Ces deux observations expliquent une grande partie des signatures et des précautions de ce chapitre.

6.2 Un exemple directeur : le TAD Polynome

6.2.1 Spécifier les services

Un polynôme associe à chaque degré naturel un coefficient réel, avec un nombre fini de coefficients non nuls. La bibliothèque doit permettre de créer le polynôme nul, de consulter son degré, de consulter un coefficient et de modifier un coefficient. Nous pouvons formaliser le TAD Polynome de la manière suivante :

Nom:	Polynome	
Utilise:	**Naturel**,**Reel**	
Opérations:	polynome:	$\rightarrow$ Polynome
	obtenirDegre:	Polynome $\rightarrow$ **Naturel**
	obtenirCoefficient:	Polynome $\times$ **Naturel** $\rightarrow$ **Reel**
	modifierCoefficient:	Polynome $\times$ **Reel** $\times$ **Naturel** $\rightarrow$ Polynome
Axiomes:	- $obtenirCoefficient(polynome(), i) = 0$	

- $obtenirDegre(polynome()) = 0$
- $i > obtenirDegre(p) \Rightarrow obtenirCoefficient(p, i) = 0$
- $p \neq polynome() \Rightarrow obtenirCoefficient(p, obtenirDegre(p)) \neq 0$
- $obtenirCoefficient(modifierCoefficient(p, v, i), i) = v$
  $j \neq i \Rightarrow$
- $obtenirCoefficient(modifierCoefficient(p, v, i), j)$
  $= obtenirCoefficient(p, j)$

Les deux premiers axiomes caractérisent le polynôme nul. Par convention de cette interface, son degré est égal à zéro. Cette convention informatique doit être distinguée de la définition du degré d'un polynôme non nul ; elle évite d'ajouter une valeur spéciale au type naturel. Les deux axiomes suivants précisent qu'aucun coefficient au-delà du degré n'est non nul et que le coefficient dominant d'un polynôme non nul est non nul. Les deux derniers axiomes décrivent la modification : le coefficient choisi prend la valeur demandée et tous les autres sont conservés. Aucune précondition n'est nécessaire pour ces quatre opérations ; en particulier, consulter un coefficient au-delà du degré renvoie zéro.

6.2.2 Conception préliminaire

La conception préliminaire détermine comment représenter ces opérations (dans le paradigme de la programmation structurée, fonctions ou procédures) et précise les passages de paramètre. La création et les consultations deviennent des fonctions. La modification d'un coefficient devient une procédure : elle modifie le polynôme reçu en entrée-sortie, tandis que le coefficient et le degré sont des paramètres d'entrée.

— **fonction** polynome **() :** Polynome
— **fonction** obtenirDegre **(**p : Polynome**) : Naturel**
— **fonction** obtenirCoefficient **(**p : Polynome, degre : **Naturel) : Reel**
— **procédure** modifierCoefficient **(E/S** p : Polynome, **E** coef : **Reel**, degre : **Naturel)**
— **fonction** sontEgaux **(**p1,p2 : Polynome**) : Booleen**
— **fonction** copier **(**p : Polynome**) :** Polynome

Ces signatures expriment les services attendus sans encore fixer les types C ni les mécanismes d'allocation. Le passage en entrée-sortie sera traduit en C par un passage de l'adresse du polynôme.

6.3 Conception détaillée : représenter le polynôme par des monômes

On choisir de représenté le polynôme une liste de monômes et un champ `degres` mémorisant son degré. Ainsi, $x^3 + 3x + 1$ peut être décrit par les couples $(1, 3)$, $(3, 1)$ et $(1, 0)$. Il n'est pas nécessaire de stocker un coefficient nul pour chaque puissance absente. Consulter un degré absent de la liste doit alors renvoyer zéro.

Exercice. Proposez le TAD Monome.

La représentation doit conserver au plus un monôme par degré et un champ `degres` cohérent avec les coefficients. L'ordre des nœuds ne fixe pas le sens mathématique du polynôme. Il intervient en revanche dans la comparaison de deux listes, distinction qui sera importante pour l'égalité des polynômes. La création présentée plus loin représente le polynôme nul par un monôme de coefficient zéro et de degré zéro.

Ce choix de conception conduit également à expliciter les services d'égalité et de copie. La fonction `sontEgaux` compare les valeurs des deux polynômes. La fonction `copier` produit un polynôme indépendant : modifier cette copie ne doit pas modifier l'original.

Enfin, à cause la représentation dynamique, on ajoute une procédure de destruction. Elle libère les ressources détenues par le polynôme et modifie la variable reçue pour ne pas y conserver de liens vers la mémoire libérée.

— **procédure** supprimer (**E/S** p : Polynome)

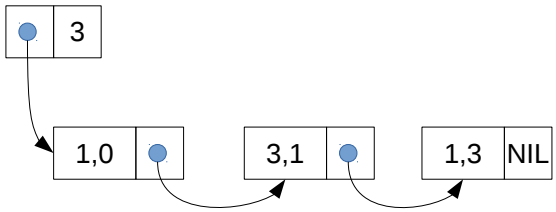
6.4 Première implantation : une liste chaînée de monômes

6.4.1 Définir la représentation et l'interface

Une liste est un pointeur vers son premier nœud. Chaque nœud contient ici un monôme par valeur et un pointeur vers la suite. La valeur `NULL` représente la liste vide. L'en-tête *ListeChainee-DeMonomes.h* contient notamment les définitions suivantes.

```
typedef struct LCDM_Noeud* LCDM_ListeChainee;
typedef struct LCDM_Noeud {
  M_Monome lElement;
  LCDM_ListeChainee listeSuivante;
} LCDM_Noeud;
```

Le nom de structure permet de déclarer le pointeur avant que tous les champs soient connus. Le type `LCDM_ListeChainee` est déjà un type pointeur : un paramètre de type `LCDM_ListeChainee*` est donc un pointeur vers un pointeur de nœud.



Les monômes font partie des nœuds eux-mêmes. Libérer un nœud libère donc aussi l'espace occupé par ses champs coefficient et degré. Aucune allocation séparée n'est nécessaire pour le monôme.

L'interface regroupe les opérations déjà étudiées sur les listes chaînées. Les opérations de consultation supposent une liste non vide.

```
LCDM_ListeChainee LCDM_listeChainee();
int LCDM_estVide(LCDM_ListeChainee);
void LCDM_ajouter(LCDM_ListeChainee*, M_Monome);
M_Monome LCDM_obtenirElement(LCDM_ListeChainee);
LCDM_ListeChainee LCDM_obtenirListeSuivante(LCDM_ListeChainee);
void LCDM_fixerElement(LCDM_ListeChainee*, M_Monome);
void LCDM_fixerListeSuivante(LCDM_ListeChainee*,
                        LCDM_ListeChainee);
void LCDM_supprimerTete(LCDM_ListeChainee*);
void LCDM_supprimer(LCDM_ListeChainee*);
LCDM_ListeChainee LCDM_copier(LCDM_ListeChainee);
int LCDM_egale(LCDM_ListeChainee, LCDM_ListeChainee);
```

Créer une liste vide renvoie `NULL` ; tester si elle est vide revient à comparer son pointeur à `NULL`. Obtenir l'élément ou la suite consiste à accéder au champ correspondant après vérification de la précondition. Fixer l'élément effectue une affectation de structure. Fixer la suite change seulement un lien : cette opération ne détruit pas les anciens nœuds et ne doit pas rendre inaccessible une chaîne encore allouée.

6.4.2 Ajouter et supprimer un nœud

L'ajout en tête alloue un nœud, y place le monôme et le relie à l'ancienne tête avant de changer la variable de l'appelant. La convention d'erreur de cette bibliothèque utilise `errno`, déclaré dans *errno.h*, et la constante locale `LCDM_ERREUR_MEMOIRE`, de valeur 1.

```
void LCDM_ajouter(LCDM_ListeChainee* pl, M_Monome m) {
  LCDM_ListeChainee pNoeud =
    (LCDM_ListeChainee)malloc(sizeof(LCDM_Noeud));
  if (pNoeud != NULL) {
    errno = 0;
    pNoeud->lElement = m;
    pNoeud->listeSuivante = *pl;
    *pl = pNoeud;
  } else {
    errno = LCDM_ERREUR_MEMOIRE;
  }
}
```

L'appel `LCDM_ajouter(&l, m)` peut modifier `l`. En cas d'échec de l'allocation, aucun lien n'a été modifié et la liste reste inchangée. Il faut examiner `errno` immédiatement après l'appel : les autres opérations de cette bibliothèque peuvent le remettre à zéro. Ce protocole est une convention du module, non un traitement automatique des erreurs par le langage.

La suppression en tête suit l'ordre inverse : conserver l'adresse du nœud, faire avancer la tête, puis libérer le nœud devenu inutile.

```
void LCDM_supprimerTete(LCDM_ListeChainee* pl) {
  LCDM_ListeChainee temp;
  assert(!LCDM_estVide(*pl));
  errno = 0;
  temp = *pl;
  *pl = LCDM_obtenirListeSuivante(*pl);
  free(temp);
}
```

Accéder à la suite après `free(temp)` serait invalide. Les autres pointeurs éventuels vers ce nœud deviennent également invalides. L'assertion exprime une précondition de programmation ; elle ne traite pas une erreur d'allocation. Elle peut être désactivée à la compilation et ne dispense donc pas l'appelant de respecter le contrat.

6.4.3 Détruire, copier et comparer une liste

La destruction complète peut suivre la définition récursive de la liste : si elle n'est pas vide, supprimer sa tête puis détruire sa suite.

```
void LCDM_supprimer(LCDM_ListeChainee* pl) {
  errno = 0;
  if (!LCDM_estVide(*pl)) {
    LCDM_supprimerTete(pl);
    LCDM_supprimer(pl);
  }
}
```

À la fin, la variable de l'appelant vaut `NULL`. Pour n nœuds, la destruction demande $O(n)$ opérations et cette version récursive utilise jusqu'à $O(n)$ cadres d'appel.

Copier une liste demande de nouveaux nœuds. L'affectation `l2 = l1` ne suffit pas : elle fait désigner la même chaîne aux deux variables. L'algorithme récursif copie d'abord la suite, puis ajoute en tête une copie du monôme courant. Cet ordre conserve celui des éléments. Les monômes ne contenant que des valeurs scalaires, leur affectation suffit à les dupliquer. Si une allocation échoue, la copie partielle doit être libérée et l'erreur conservée avant tout nouvel appel susceptible de modifier `errno` ; une copie incomplète ne doit pas être annoncée comme réussie.

Pour comparer deux listes, deux listes vides sont égales, tandis qu'une seule liste vide suffit à conclure à l'inégalité. Sinon, on compare les monômes de tête avec `M_egal`, puis les suites si les têtes sont égales. Il s'agit d'une égalité de séquences, dans le même ordre, et non d'une égalité d'adresses. La comparaison est en $O(n)$ dans le pire cas pour deux listes de longueur n, comme la copie, en supposant les opérations sur les monômes de coût constant.

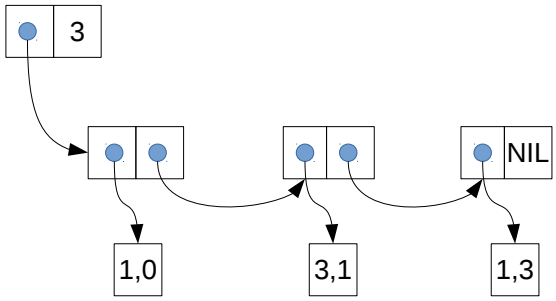
6.5 Deuxième implantation : une liste chaînée générique

6.5.1 Séparer le chaînage du type des éléments

La liste précédente dépend du type `M_Monome`. Une liste de chaînes de caractères exigerait une autre version, alors que ses opérations de chaînage seraient identiques. Pour rendre ces opérations réutilisables, chaque nœud peut contenir un pointeur générique vers l'élément.

```
typedef struct LC_Noeud* LC_ListeChainee;
typedef struct LC_Noeud {
  void* lElement;
  LC_ListeChainee listeSuivante;
} LC_Noeud;
```

Un `void*` peut désigner un objet sans exprimer son type dans l'interface. On ne peut pas le déréférencer directement pour accéder à un monôme : il faut retrouver un pointeur du type approprié. La cohérence entre les objets stockés et les fonctions qui les manipulent devient une responsabilité de l'utilisateur de la bibliothèque.



Cette fois, le nœud et le monôme sont deux objets distincts. Libérer le nœud ne libère pas l'objet désigné par `lElement`. Réciproquement, libérer cet objet ne détruit pas le nœud.

6.5.2 Paramétrer les opérations sur les éléments

La liste ne connaît ni la taille ni la structure des objets stockés. Elle ne peut donc pas décider seule comment les copier, les libérer ou les comparer. L'en-tête *copieLiberationComparaison.h* définit les types de fonctions qui lui apportent ces services.

```
typedef void* (*CLC_FonctionCopier)(void*);
typedef void (*CLC_FonctionLiberer)(void*);
typedef int (*CLC_FonctionComparer)(void*, void*);
```

Une fonction de copie produit un nouvel objet indépendant et renvoie son adresse, ou `NULL` en cas d'échec. Une fonction de libération détruit cet objet. Ici, la fonction de comparaison est un test d'égalité : elle renvoie une valeur non nulle si les éléments sont égaux et zéro sinon. Il ne s'agit pas d'une comparaison d'ordre renvoyant zéro pour l'égalité.

L'interface transmet ces fonctions aux opérations qui en ont besoin.

```
LC_ListeChainee LC_listeChainee();
int LC_estVide(LC_ListeChainee);
void* LC_obtenirElement(LC_ListeChainee);
LC_ListeChainee LC_obtenirListeSuivante(LC_ListeChainee);
void LC_fixerListeSuivante(LC_ListeChainee*, LC_ListeChainee);
void LC_ajouter(LC_ListeChainee*, void*, CLC_FonctionCopier);
void LC_fixerElement(LC_ListeChainee*, void*,
                     CLC_FonctionCopier, CLC_FonctionLiberer);
void LC_supprimerTete(LC_ListeChainee*, CLC_FonctionLiberer);
void LC_supprimer(LC_ListeChainee*, CLC_FonctionLiberer);
LC_ListeChainee LC_copier(LC_ListeChainee,
```

```
                        CLC_FonctionCopier);
int LC_egales(LC_ListeChainee, LC_ListeChainee,
              CLC_FonctionComparer);
```

6.5.3 Ajouter un élément en conservant une copie

La convention retenue est que l'ajout duplique l'élément reçu. La liste devient responsable de cette copie ; l'appelant conserve la responsabilité de son objet initial. L'ajout exige donc deux allocations possibles, celle du nœud et celle effectuée par la fonction de copie. L'extrait suivant traite séparément les deux échecs, sans changer la tête avant que les deux objets soient disponibles.

```
void LC_ajouter(LC_ListeChainee* pl, void* source,
                CLC_FonctionCopier copierElement) {
  LC_ListeChainee pNoeud =
    (LC_ListeChainee)malloc(sizeof(LC_Noeud));
  void* donnee;
  if (pNoeud == NULL) {
    errno = LC_ERREUR_MEMOIRE;
    return;
  }
  donnee = copierElement(source);
  if (donnee == NULL) {
    free(pNoeud);
    errno = LC_ERREUR_MEMOIRE;
    return;
  }
  pNoeud->lElement = donnee;
  pNoeud->listeSuivante = *pl;
  *pl = pNoeud;
  errno = 0;
}
```

La constante `LC_ERREUR_MEMOIRE` joue le même rôle que dans la version spécialisée. La fonction de copie doit elle-même nettoyer ses allocations intermédiaires si elle échoue. Le nœud est ici alloué en premier : cela évite de créer un élément qu'il faudrait ensuite détruire si l'allocation du nœud échouait, alors que l'ajout ne reçoit pas de fonction de libération.

6.5.4 Gérer la durée de vie des objets

Supprimer la tête nécessite de faire avancer le pointeur de liste, de libérer l'élément par la fonction fournie, puis de libérer le nœud. La destruction complète répète cette opération. Copier la liste nécessite, pour chaque nœud, un nouveau nœud et une copie de son élément : c'est une copie en profondeur. Comparer deux listes applique le test d'égalité fourni aux éléments de même position.

Le remplacement d'un élément doit copier la nouvelle valeur avant de libérer l'ancienne. Si la copie échoue, l'ancien élément reste ainsi valide. Cet ordre permet aussi de remplacer un élément par une copie de lui-même sans lire une zone déjà libérée. Après réussite seulement, on libère l'ancien objet et on installe la nouvelle adresse.

La consultation renvoie l'adresse de l'objet appartenant à la liste, et non une copie. L'appelant ne doit pas libérer cet objet séparément. L'adresse cesse d'être utilisable lorsque l'élément est remplacé ou supprimé. Ces règles font partie du contrat, même si les types C ne permettent pas de les imposer.

L'interface de copie complète mérite une réserve : elle reçoit une fonction de copie, mais aucune fonction de libération. Elle ne dispose donc pas, à elle seule, du moyen général de détruire les éléments déjà copiés si une allocation ultérieure échoue. Une version qui garantit le nettoyage sur tous les échecs doit disposer aussi de ce service. Cette limite doit être prise en compte avant d'utiliser cette interface dans une bibliothèque exigeant cette garantie.

6.6 Utiliser la liste générique pour implanter les polynômes

6.6.1 Adapter les opérations sur les monômes

Le module *monomeAllocationDynamique* relie le type Monome aux signatures génériques. Il connaît la représentation des monômes et peut donc allouer la place nécessaire, copier leur valeur et les comparer. La liste reste indépendante de cette connaissance.

```
void* MAD_copierMonome(void* m) {
  M_Monome* resultat = (M_Monome*)malloc(sizeof(M_Monome));
  if (resultat == NULL)
    return NULL;
  memcpy((void*)resultat, m, sizeof(M_Monome));
  return (void*)resultat;
}

void MAD_desallouerMonome(void* m) {
  free(m);
}

int MAD_comparerMonome(void* m1, void* m2) {
  return M_egal(*(M_Monome*)m1, *(M_Monome*)m2);
}
```

Ces fonctions supposent que les pointeurs reçus désignent des monômes valides. La vérification de l'allocation précède la copie mémoire. `memcpy`, déclaré dans *string.h*, convient ici car le monôme ne contient pas de pointeur vers une autre allocation. Pour un objet qui possède de telles ressources, copier ses octets ne suffirait pas à réaliser une copie indépendante. `malloc` et `free` sont déclarés dans *stdlib.h*.

6.6.2 Créer un polynôme

L'en-tête du polynôme associe la liste générique au degré mémorisé.

```
typedef struct {
  LC_ListeChainee monomes;
  unsigned int degres;
} P_Polynome;
```

La création utilise un monôme local de coefficient nul. L'ajout en conserve une copie allouée dynamiquement, qui reste valide après le retour de la fonction.

```
P_Polynome P_polynome() {
  P_Polynome resultat;
  M_Monome m = M_monome(0, 0);
  resultat.monomes = LC_listeChainee();
  LC_ajouter(&(resultat.monomes), &m, MAD_copierMonome);
  resultat.degres = 0;
  return resultat;
}
```

L'appelant doit vérifier immédiatement `errno` : si l'ajout échoue, le monôme attendu n'a pas été créé. Transmettre simplement l'adresse de `m` sans la copier laisserait dans la liste l'adresse d'une variable locale dont la durée de vie est terminée.

6.6.3 Consulter un coefficient

Le degré mémorisé se consulte directement. La recherche d'un coefficient parcourt les monômes jusqu'à trouver le degré demandé ou atteindre la fin. La valeur initiale zéro donne le résultat attendu lorsque ce degré est absent.

```
float P_obtenirCoefficient(P_Polynome p, unsigned int d) {
  float resultat = 0.0;
  LC_ListeChainee l = p.monomes;
  int trouve = false;
  M_Monome* monomeCourant;

  while (!LC_estVide(l) && !trouve) {
    monomeCourant = (M_Monome*)LC_obtenirElement(l);
    if (M_degre(*monomeCourant) == d) {
      resultat = M_coef(*monomeCourant);
      trouve = true;
    } else {
      l = LC_obtenirListeSuivante(l);
    }
  }
  return resultat;
}
```

Les constantes `false` et `true` sont fournies par *stdbool.h*. Le pointeur local `l` avance sans modifier la tête conservée dans `p.monomes`. La conversion en `M_Monome*` est justifiée par le contrat du module : cette liste contient uniquement des monômes. Avec n monômes, la consultation demande $O(n)$ opérations dans le pire cas ; la consultation du degré mémorisé est en $O(1)$.

6.6.4 Préserver les invariants lors des autres opérations

Modifier un coefficient commence par le même parcours. Si le degré existe, on remplace le monôme à l'aide de `LC_fixerElement` et des fonctions de copie et de libération adaptées. Sinon, on ajoute un monôme. Le degré mémorisé n'est mis à jour qu'après réussite de l'opération. Si le coefficient dominant devient nul, il faut rechercher le plus grand degré dont le coefficient reste non nul ; s'il n'en reste aucun, on applique la convention du polynôme nul. Augmenter seulement le degré lors des insertions serait insuffisant pour respecter la spécification.

L'égalité des polynômes porte sur leurs coefficients, indépendamment de l'ordre des monômes. Deux listes contenant les mêmes monômes dans des ordres différents représentent donc le même polynôme, alors que `LC_egales` les déclare différentes. Utiliser directement cette fonction exige une représentation canonique, notamment un ordre commun et une convention commune pour les coefficients nuls. Sans cette contrainte, il faut comparer les coefficients associés aux degrés présents dans les deux représentations.

De même, renvoyer simplement `p` ne réalise pas une copie indépendante du polynôme : la liste serait partagée. La copie doit dupliquer les nœuds et les monômes, puis recopier le degré, avec un traitement explicite des échecs. La destruction libère la liste avec `MAD_desallouerMonome` et réinitialise le degré. Ces opérations doivent respecter les mêmes règles de possession mémoire que les opérations élémentaires.

Le découpage obtenu sépare ainsi quatre responsabilités : Monome manipule une valeur, son module d'allocation adapte cette valeur au protocole générique, la liste organise les objets en mémoire, et Polynome interprète ces objets comme des coefficients associés à des degrés.

6.7 Synthèse : implantation spécialisée ou générique

La liste spécialisée place directement un monôme dans chaque nœud. Son interface exprime le type des éléments et le compilateur peut vérifier les arguments correspondants. Sa réutilisation pour un autre type demande cependant une autre implantation.

La liste générique conserve les mêmes principes de chaînage, mais désigne ses éléments par des `void*`. Les fonctions de copie, de libération et d'égalité donnent un sens aux opérations sur ces objets. Cette séparation rend la liste réutilisable, au prix d'une responsabilité supplémentaire : l'utilisateur doit associer les bons objets aux bonnes fonctions et respecter leur durée de vie.

Propriété	Liste de monômes	Liste générique
Élément dans le nœud	Valeur du monôme	Pointeur vers l'objet
Copie d'un élément	Affectation	Fonction fournie
Libération de l'élément	Avec le nœud	Fonction fournie
Égalité des éléments	Fonction du monôme	Fonction fournie
Réutilisation	Type fixé	Plusieurs types possibles

Dans les deux cas, il faut distinguer copier une adresse et copier une structure complète, vérifier les allocations avant d'utiliser leur résultat, et conserver les invariants lors des modifications. Le coût du chaînage seul ne suffit pas toujours à caractériser une opération générique : il faut ajouter celui des fonctions fournies pour les éléments.

Trois voies sont donc possibles pour disposer de structures dynamiques en C : développer des versions spécialisées, réaliser des structures génériques ou utiliser une bibliothèque existante, telle que GLib. Dans ce dernier cas, l'étude du contrat de copie et de libération reste indispensable. Quelle que soit la solution choisie, la spécification du TAD détermine les services à rendre et l'implantation doit expliciter les moyens de les rendre correctement.

Chapitre 7

Les graphes

7.1 Modéliser des relations par un graphe

Une liste organise ses éléments en séquence et un arbre les organise selon une hiérarchie. Un graphe permet de décrire une organisation plus générale : un réseau routier, les liens entre des pages Web, les dépendances entre les tâches d'un projet ou encore les connexions d'un réseau informatique. Dans chacun de ces exemples, les objets sont reliés sans former nécessairement une suite ou une hiérarchie.

Un graphe $G = (S, A)$ est composé de deux ensembles finis. L'ensemble S contient les *sommets*, également appelés nœuds. L'ensemble A contient les *arcs*. Un arc est un couple (i, j) de sommets ; il exprime une relation allant de i vers j. Lorsqu'un graphe possède n sommets, nous les noterons souvent $1, 2, \ldots, n$.

Dans un *graphe orienté*, les couples (i, j) et (j, i) désignent deux arcs différents. Dans un *graphe non orienté*, la relation est symétrique : (i, j) et (j, i) représentent le même lien. Le mot *arête* est aussi couramment employé pour un tel lien non orienté. Sauf indication contraire, les exemples de ce chapitre emploient le terme générique d'arc.

La figure 7.1 représente le graphe orienté $G = (S, A)$ défini par

$$S = \{1, 2, 3, 4\} \quad \text{et} \quad A = \{(1, 1), (1, 2), (2, 3), (3, 1), (3, 4)\}.$$

L'arc $(1, 1)$ est une *boucle* : son origine et son extrémité sont le même sommet.

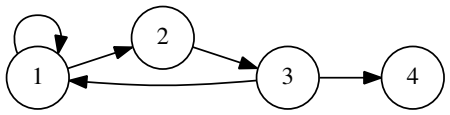


FIGURE 7.1 – Un graphe orienté à quatre sommets.

7.1.1 Vocabulaire

Deux sommets reliés par un arc sont dits *adjacents*. Pour un arc (a, b) d'un graphe orienté, l'arc est incident extérieurement à a et incident intérieurement à b. Le nombre d'arcs incidents à un sommet est son *degré*. Dans un graphe orienté, on distingue plus précisément le degré entrant, qui compte les arcs arrivant au sommet, et le degré sortant, qui compte ceux qui en partent. Une boucle contribue à la fois à ces deux degrés.

Un *chemin* est une séquence de sommets $(s_0, s_1, \ldots, s_p)$ telle que (s_{i-1}, s_i) soit un arc pour tout i compris entre 1 et p. Sa longueur est le nombre d'arcs parcourus, donc p. Un chemin est dit *simple* lorsqu'il n'emprunte pas plusieurs fois le même arc et *élémentaire* lorsqu'il ne passe pas

plusieurs fois par le même sommet. Un chemin dont le premier et le dernier sommet coïncident forme un *circuit*, ou un *cycle*.

Un graphe non orienté est *connexe* lorsqu'un chemin relie toute paire de sommets. Pour un graphe orienté, l'existence d'un chemin orienté entre toute paire ordonnée de sommets caractérise plus précisément la forte connexité. Un graphe est *complet* lorsque deux sommets distincts quelconques sont adjacents. Enfin, une *clique* est un ensemble de sommets deux à deux adjacents ; le sous-graphe qu'ils engendrent est complet.

La structure du graphe ne porte pas toujours toute l'information utile. Dans un *graphe étiqueté*, une information est associée à chaque sommet : le nom d'une ville, par exemple. Dans un *graphe valué*, une valeur est associée à chaque arc : une distance, une durée ou un coût. Orientation, étiquetage et valuation sont trois caractéristiques indépendantes qui déterminent les services attendus du TAD.

7.2 Les TAD Graphe

Il n'existe pas un unique TAD Graphe. Selon le problème, les arcs sont orientés ou non, les sommets portent ou non une étiquette et les arcs portent ou non une valeur. Les trois choix étant indépendants, ils conduisent à huit variantes. Nous décrivons deux interfaces générales, l'une non orientée et l'autre orientée. Une application n'utilise que les opérations correspondant aux informations effectivement présentes dans son graphe.

7.2.1 Graphe non orienté

Le TAD **Graphe** est paramétré par les types **Sommet**, **Etiquette** et **Valeur**. Les opérations de consultation ne modifient pas le graphe. Les opérations d'ajout, de suppression et de fixation produisent le graphe modifié.

Nom:	Graphe	
Paramètre:	Sommet, Etiquette, Valeur	
Utilise:	**Booleen**, Liste	
Opérations:	graphe:	$\rightarrow$ Graphe
	ajouterSommet:	Graphe $\times$ Sommet $\nrightarrow$ Graphe
	ajouterArc:	Graphe $\times$ Sommet $\times$ Sommet $\nrightarrow$ Graphe
	sommetPresent:	Graphe $\times$ Sommet $\rightarrow$ **Booleen**
	arcPresent:	Graphe $\times$ Sommet $\times$ Sommet $\rightarrow$ **Booleen**
	supprimerSommet:	Graphe $\times$ Sommet $\nrightarrow$ Graphe
	supprimerArc:	Graphe $\times$ Sommet $\times$ Sommet $\nrightarrow$ Graphe
	obtenirSommets:	Graphe $\rightarrow$ Liste<Sommet>
	obtenirSommetsAdjacents:	Graphe $\times$ Sommet $\nrightarrow$ Liste<Sommet>
	possedeEtiquette:	Graphe $\times$ Sommet $\nrightarrow$ **Booleen**
	obtenirEtiquette:	Graphe $\times$ Sommet $\nrightarrow$ Etiquette
	fixerEtiquette:	Graphe $\times$ Sommet $\times$ Etiquette $\nrightarrow$ Graphe
	possedeValeur:	Graphe $\times$ Sommet $\times$ Sommet $\nrightarrow$ **Booleen**
	obtenirValeur:	Graphe $\times$ Sommet $\times$ Sommet $\nrightarrow$ Valeur
	fixerValeur:	Graphe $\times$ Sommet $\times$ Sommet $\times$ Valeur $\nrightarrow$ Graphe
Préconditions:	ajouterSommet(g,s):	non sommetPresent(g,s)
	ajouterArc(g,s1,s2):	sommetPresent(g,s1) et sommetPresent(g,s2) et non arcPresent(g,s1,s2)
	supprimerSommet(g,s):	sommetPresent(g,s)
	supprimerArc(g,s1,s2):	arcPresent(g,s1,s2)
	obtenirSommetsAdjacents(g,s):	sommetPresent(g,s)

possedeEtiquette(g,s):	sommetPresent(g,s)
obtenirEtiquette(g,s):	sommetPresent(g,s) et possedeEtiquette(g,s)
fixerEtiquette(g,s,e):	sommetPresent(g,s)
possedeValeur(g,s1,s2):	arcPresent(g,s1,s2)
obtenirValeur(g,s1,s2):	arcPresent(g,s1,s2) et possedeValeur(g,s1,s2)
fixerValeur(g,s1,s2,v):	arcPresent(g,s1,s2)

Supprimer un sommet supprime aussi les arcs qui lui sont incidents. Dans un graphe non orienté, ajouter ou supprimer le lien entre s_1 et s_2 produit le même effet quel que soit l'ordre des deux sommets. De même, la valeur du lien est commune aux couples (s_1, s_2) et (s_2, s_1).

7.2.2 Graphe orienté

Dans un graphe orienté, la notion de sommets adjacents se décompose en deux consultations. Les prédécesseurs de s sont les sommets p tels que (p, s) soit un arc ; ses successeurs sont les sommets q tels que (s, q) soit un arc. Le TAD **GrapheOriente** reprend donc les opérations précédentes en remplaçant `obtenirSommetsAdjacents` par ces deux services.

Nom: GrapheOriente

Paramètre: Sommet, Etiquette, Valeur

Utilise: **Booleen**, Liste

Opérations:

grapheOriente:	$\rightarrow$ GrapheOriente
ajouterSommet:	GrapheOriente $\times$ Sommet $\nrightarrow$ GrapheOriente
ajouterArc:	GrapheOriente $\times$ Sommet $\times$ Sommet $\nrightarrow$ GrapheOriente
sommetPresent:	GrapheOriente $\times$ Sommet $\rightarrow$ **Booleen**
arcPresent:	GrapheOriente $\times$ Sommet $\times$ Sommet $\rightarrow$ **Booleen**
supprimerSommet:	GrapheOriente $\times$ Sommet $\nrightarrow$ GrapheOriente
supprimerArc:	GrapheOriente $\times$ Sommet $\times$ Sommet $\nrightarrow$ GrapheOriente
obtenirSommets:	GrapheOriente $\rightarrow$ Liste<Sommet>
obtenirPredecesseurs:	GrapheOriente $\times$ Sommet $\nrightarrow$ Liste<Sommet>
obtenirSuccesseurs:	GrapheOriente $\times$ Sommet $\nrightarrow$ Liste<Sommet>
possedeEtiquette:	GrapheOriente $\times$ Sommet $\nrightarrow$ **Booleen**
obtenirEtiquette:	GrapheOriente $\times$ Sommet $\nrightarrow$ Etiquette
fixerEtiquette:	GrapheOriente $\times$ Sommet $\times$ Etiquette $\nrightarrow$ GrapheOriente
possedeValeur:	GrapheOriente $\times$ Sommet $\times$ Sommet $\nrightarrow$ **Booleen**
obtenirValeur:	GrapheOriente $\times$ Sommet $\times$ Sommet $\nrightarrow$ Valeur
fixerValeur:	GrapheOriente $\times$ Sommet $\times$ Sommet $\times$ Valeur $\nrightarrow$ GrapheOriente

Préconditions:

ajouterSommet(g,s):	non sommetPresent(g,s)
ajouterArc(g,s1,s2):	sommetPresent(g,s1) et sommetPresent(g,s2) et non arcPresent(g,s1,s2)
supprimerSommet(g,s):	sommetPresent(g,s)
supprimerArc(g,s1,s2):	arcPresent(g,s1,s2)
obtenirPredecesseurs(g,s):	sommetPresent(g,s)
obtenirSuccesseurs(g,s):	sommetPresent(g,s)
possedeEtiquette(g,s):	sommetPresent(g,s)
obtenirEtiquette(g,s):	sommetPresent(g,s) et possedeEtiquette(g,s)
fixerEtiquette(g,s,e):	sommetPresent(g,s)
possedeValeur(g,s1,s2):	arcPresent(g,s1,s2)
obtenirValeur(g,s1,s2):	arcPresent(g,s1,s2) et possedeValeur(g,s1,s2)

fixerValeur(g,s1,s2,v): arcPresent(g,s1,s2)

Cette interface générale peut être réduite au besoin. Un graphe non orienté simple dont les sommets sont des naturels non nuls peut ainsi être vu comme un **Graphe<NaturelNonNul>** ; les opérations relatives aux étiquettes et aux valeurs ne sont pas utilisées. Pour un réseau routier non orienté dont les arcs portent une distance réelle, on peut employer le type

Graphe<NaturelNonNul,,Reel>.

Les deux virgules indiquent que le paramètre d'étiquette n'est pas instancié.

7.3 Représenter un graphe

La spécification précise les services du graphe sans imposer son implantation. Le choix d'une représentation dépend notamment du nombre de sommets, du nombre d'arcs et des opérations les plus fréquentes. Nous présentons trois représentations classiques de la relation d'adjacence.

7.3.1 Matrice d'adjacence

Soit un graphe dont les n sommets sont numérotés de 1 à n. Sa matrice d'adjacence est une matrice booléenne M de taille $n \times n$ telle que

$$M[i,j] = \begin{cases} \text{VRAI} & \text{si } (i,j) \text{ est un arc;} \\ \text{FAUX} & \text{sinon.} \end{cases}$$

Pour le graphe de la figure 7.1, on obtient

$$M = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}.$$

La ligne i décrit les successeurs du sommet i. Dans un graphe non orienté, la matrice est symétrique. Cette représentation occupe $\Theta(n^2)$ cases, qu'il y ait peu ou beaucoup d'arcs. En contrepartie, tester la présence de (i,j) s'effectue en temps constant. Elle convient donc notamment aux graphes denses ou aux applications qui interrogent très souvent l'existence d'un arc.

Pour un graphe valué, chaque case peut contenir la valeur de l'arc plutôt qu'un booléen. Il faut alors choisir une valeur particulière pour signaler l'absence d'arc, sans la confondre avec une valeur valide.

7.3.2 Matrice d'incidence

Si le graphe possède n sommets et p arcs $a_1, \ldots, a_p$, sa matrice d'incidence contient n lignes et p colonnes. Pour un arc $a_k = (i,j)$ avec $i \neq j$, la colonne k contient 1 à la ligne i et -1 à la ligne j lorsque le graphe est orienté. Pour un graphe non orienté, ces deux valeurs sont égales à 1. Une boucle (i,i) est représentée par la valeur 2 à la ligne i ; toutes les autres cases sont nulles.

En numérotant les arcs de l'exemple dans l'ordre $(1,1), (1,2), (2,3), (3,1), (3,4)$, on obtient

$$M = \begin{pmatrix} 2 & 1 & 0 & -1 & 0 \\ 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & -1 & 1 & 1 \\ 0 & 0 & 0 & 0 & -1 \end{pmatrix}.$$

Cette représentation rend explicite l'incidence entre sommets et arcs. Elle occupe toutefois $\Theta(np)$ cases et est moins directe pour obtenir tous les successeurs d'un sommet.

7.3.3 Listes d'adjacence

Une liste d'adjacence associe à chaque sommet la liste de ses successeurs. Pour le graphe de la figure 7.1, les quatre listes sont

$$((1,2),(3),(1,4),()).$$

La première composante indique que 1 a pour successeurs 1 et 2 ; la dernière indique que 4 n'en a aucun. Pour un graphe non orienté, chaque arête reliant deux sommets distincts apparaît dans les deux listes concernées.

Les listes d'adjacence occupent $\Theta(n+p)$ emplacements pour un graphe orienté. Parcourir les successeurs de i demande un temps proportionnel au degré sortant de i. En revanche, rechercher un successeur particulier peut demander de parcourir toute sa liste. Cette représentation est particulièrement adaptée aux graphes creux, dans lesquels p est faible devant n^2.

Représentation	Espace	Test de l'arc (i,j)
Matrice d'adjacence	$\Theta(n^2)$	$\Theta(1)$
Matrice d'incidence	$\Theta(np)$	dépend de l'accès aux colonnes
Listes d'adjacence	$\Theta(n+p)$	$O(deg^+(i))$

Les étiquettes et les valeurs peuvent être conservées indépendamment de la relation d'adjacence. Un dictionnaire dont les clés sont les sommets associe une étiquette à chaque sommet. Un second dictionnaire, indexé par des couples de sommets, associe une valeur à chaque arc. Il est aussi possible d'intégrer directement la valeur d'un arc dans la cellule de matrice ou le maillon de liste qui le représente. Ces choix relèvent de la conception détaillée et ne modifient pas l'interface du TAD.

7.4 Parcourir un graphe

Parcourir un graphe consiste à visiter ses sommets une seule fois afin d'appliquer le même traitement à chacun. À partir d'un sommet donné, seuls les sommets accessibles peuvent être visités. Si le graphe n'est pas connexe, un parcours de tous les sommets doit donc être relancé depuis un sommet non encore visité.

Deux stratégies principales se distinguent par l'ordre des visites. Le *parcours en profondeur* suit une direction aussi loin que possible avant de revenir en arrière. Le *parcours en largeur* visite d'abord les sommets situés à un arc du point de départ, puis ceux situés à deux arcs, et ainsi de suite. Sur le graphe de la figure 7.2, et en examinant les voisins dans l'ordre croissant, les ordres obtenus depuis 1 sont :

profondeur : $1, 2, 5, 6, 8, 3, 7, 4$;
largeur : $1, 2, 3, 4, 5, 6, 7, 8$.

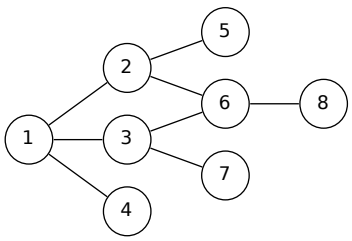


FIGURE 7.2 – Un graphe pour comparer les ordres de parcours.

Le choix de la collection des sommets en attente détermine le parcours. Une pile traite en premier le dernier sommet ajouté et conduit à un parcours en profondeur. Une file traite en premier le sommet en attente depuis le plus longtemps et conduit à un parcours en largeur.

Dans les deux cas, un mécanisme est indispensable pour ne pas suivre indéfiniment un cycle. On peut marquer les sommets dans le graphe ou conserver un ensemble des sommets déjà visités. Pour le parcours en largeur ci-dessous, un sommet n'est marqué qu'au moment de son retrait de la file. Il faut donc aussi vérifier qu'il n'est pas déjà en attente avant de l'enfiler.

```
procédure parcoursEnLargeurIteratif (E g : Graphe<Sommet>, s : Sommet,
traiter : procedure(Graphe<Sommet>,Sommet))
   ⌊précondition(s)  sommetPresent(g,s)
   Déclaration  f : File<Sommet>
                sCourant : Sommet
debut
   f ← file()
   enfiler(f,s)
   tant que non estVide(f) faire
      sCourant ← obtenirElement(f)
      defiler(f)
      marquer(sCourant)
      traiter(g,sCourant)
      pour chaque s' de obtenirSommetsAdjacents(g,sCourant)
         si non estMarque(s') et non estPresent(f,s') alors
            enfiler(f,s')
         finsi
      finpour
   fintantque
fin
```

Marquer un sommet dès son insertion dans la file est une variante équivalente qui dispense du test `estPresent`. Avec des listes d'adjacence, chaque sommet et chaque arc accessible sont examinés un nombre borné de fois. Le parcours a donc un coût en $O(n + p)$, où n est le nombre de sommets et p le nombre d'arcs.

7.5 Ordonner les sommets d'un graphe orienté

Un graphe orienté acyclique peut représenter des contraintes d'antériorité. Un arc (a, b) signifie alors que a doit précéder b : une tâche doit être achevée avant d'en commencer une autre, ou un module doit être compilé avant celui qui l'utilise. La relation définit un ordre partiel, car certains sommets peuvent ne pas être comparables.

Un *tri topologique* construit une liste de tous les sommets telle que chaque sommet apparaisse après tous ses prédécesseurs. Cette liste constitue un ordre total compatible avec l'ordre partiel. Elle n'est généralement pas unique : deux sommets indépendants peuvent être placés dans un ordre ou dans l'autre.

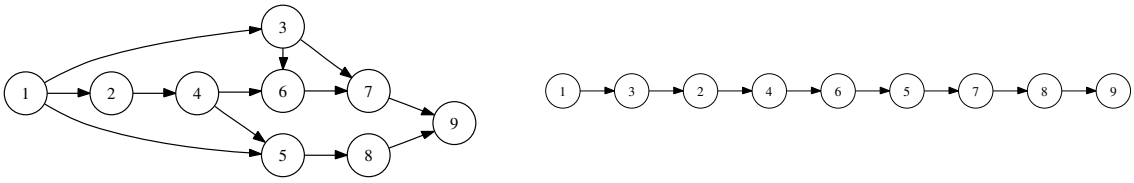


FIGURE 7.3 – Un graphe orienté acyclique et l'un de ses tris topologiques.

Dans la version suivante, le graphe possède une seule racine s : elle n'a aucun prédécesseur et permet d'atteindre tous les autres sommets. Une file contient les sommets qui peuvent être

ajoutés à la liste. Lorsqu'un sommet est traité, un de ses successeurs devient disponible si tous ses prédécesseurs ont déjà été marqués.

```
fonction triTopologique (g : GrapheOriente<Sommet>, s : Sommet) : Liste<Sommet>
  ⌊précondition(s)  sommetPresent(g,s) et estVide(obtenirPredecesseurs(g,s))
  Déclaration  l : Liste<Sommet>
               f : File<Sommet>
               sCourant : Sommet
debut
  l ← liste()
  f ← file()
  enfiler(f,s)
  tant que non estVide(f) faire
    sCourant ← obtenirElement(f)
    defiler(f)
    inserer(l,longueur(l)+1,sCourant)
    marquer(sCourant)
    pour chaque s' de obtenirSuccesseurs(g,sCourant)
      si non estMarque(s') et non estPresent(f,s') et
      tousPredecesseursMarques(g,s') alors
        enfiler(f,s')
      finsi
    finpour
  fintantque
  retourner l
fin
```

Si le graphe contient un cycle, aucun ordre ne peut respecter tous les arcs du cycle. L'algorithme doit donc être appliqué à un graphe acyclique, ou complété par une vérification : si la liste obtenue contient moins de sommets que le graphe, les contraintes ne permettent pas un tri topologique complet.

Exercice. Donner l'algorithme de la fonction `tousPredecesseursMarques`. Expliquer comment adapter le tri à un graphe acyclique possédant plusieurs sommets sans prédécesseur.

7.6 Rechercher un plus court chemin

Dans un graphe valué, le coût d'un chemin est la somme des valeurs de ses arcs. Le problème du plus court chemin consiste à trouver, entre une source et une destination, un chemin de coût minimal. Selon la nature des valeurs et les chemins recherchés, différents algorithmes peuvent être employés, notamment Dijkstra, Bellman–Ford ou Floyd–Warshall. Nous supposons disposer de la fonction suivante :

— **fonction** obtenirCout (g : Graphe, s1,s2 : Sommet) **: ReelPositif**

7.6.1 Algorithme de Dijkstra

L'algorithme de Dijkstra s'applique lorsque les coûts des arcs sont positifs ou nuls. Il construit progressivement un ensemble de sommets atteints dont la distance minimale depuis la source est définitivement connue. La distance de la source à elle-même vaut initialement zéro.

À chaque étape, on considère les arcs qui partent de l'arbre déjà construit et arrivent à un sommet qui n'y appartient pas. Parmi ces arcs, on choisit celui qui minimise la somme de la distance connue jusqu'à son origine et du coût de l'arc. Le sommet d'arrivée et cette distance sont alors ajoutés. Les coûts non négatifs garantissent qu'un chemin découvert ultérieurement ne pourra pas améliorer la distance du sommet ainsi choisi.

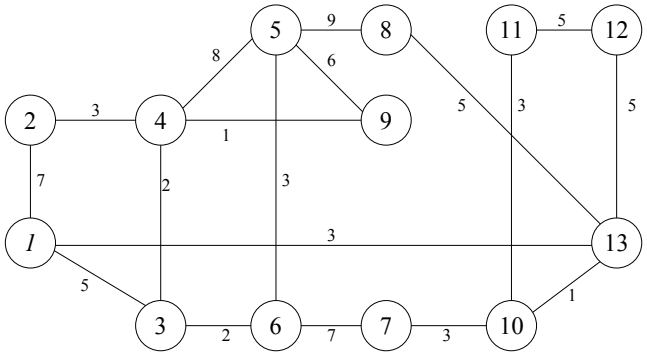


FIGURE 7.4 – Un graphe valué dans lequel rechercher des chemins minimaux.

L'arbre obtenu mémorise un plus court chemin de la source vers chaque sommet accessible. Le dictionnaire `cout` associe à ces sommets leur distance minimale. Si seule une destination est recherchée, le calcul peut s'arrêter dès que celle-ci entre dans l'arbre.

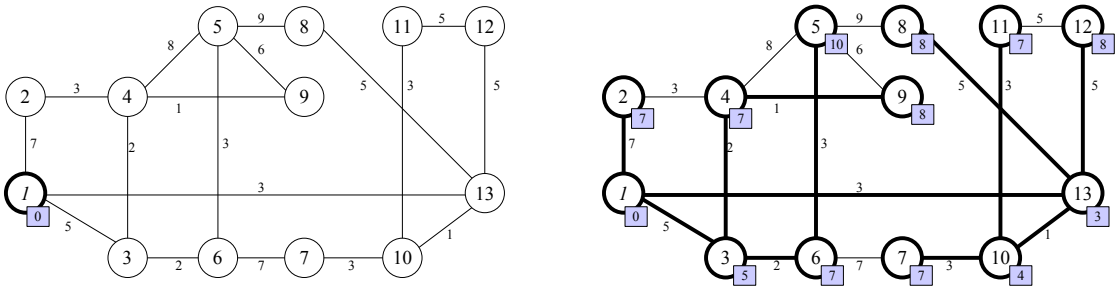


FIGURE 7.5 – Initialisation depuis le sommet 1 et arbre des plus courts chemins obtenu.

L'algorithme utilise quatre services auxiliaires :

— **fonction** arbreInitial (s : Sommet) **:** Arbre<Sommet>

— **fonction** arcsEntreArbreEtGraphe (a : Arbre<Sommet>,
g : Graphe<Sommet,„**ReelPositif**>) **:** Liste<Liste<Sommet>>

— **fonction** arcMinimal (g : Graphe<Sommet,„**ReelPositif**>, arcs : Liste<Liste<Sommet>>,
cout : Dictionnaire<Sommet,**ReelPositif**>) **:** Sommet, Sommet, **ReelPositif**

⌊**précondition(s)** non estVide(arcs)

— **procédure** ajouterCommeFils (**E**/**S** a : Arbre<Sommet>, **E** pere,fils : Sommet)

⌊**précondition(s)** estPresent(a,pere)

La fonction `arcMinimal` renvoie l'origine, la destination et le coût propre de l'arc qui donne la plus petite distance totale depuis la source. Elle suppose que le dictionnaire contient la distance de l'origine de chacun des arcs reçus.

fonction dijkstra (g : Graphe<Sommet,„**ReelPositif**>,
s : Sommet) **:** Arbre<Sommet>,
Dictionnaire<Sommet,**ReelPositif**>

⌊**précondition(s)** sommetPresent(g,s)

Déclaration a : Arbre<Sommet>
cout : Dictionnaire<Sommet,**ReelPositif**>
arcs : Liste<Liste<Sommet>>

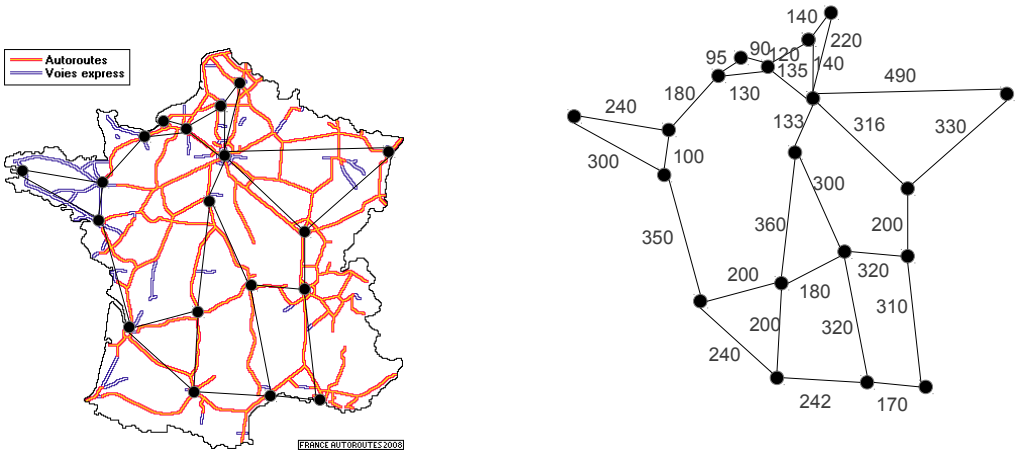
```
                c : ReelPositif
                origine,destination : Sommet
debut
    a ← arbreInitial(s)
    cout ← dictionnaire()
    ajouter(cout,s,0)
    arcs ← arcsEntreArbreEtGraphe(a,g)
    tant que non estVide(arcs) faire
        origine,destination,c ← arcMinimal(g,arcs,cout)
        ajouter(cout,destination,obtenirValeur(cout,origine)+c)
        ajouterCommeFils(a,origine,destination)
        arcs ← arcsEntreArbreEtGraphe(a,g)
    fintantque
    retourner a,cout
fin
```

Une implantation directe qui recherche linéairement le prochain sommet a un coût en $O(n^2)$. Avec des listes d'adjacence et une file de priorité fondée sur un tas binaire, la complexité devient $O((n+p)\log n)$. D'autres structures de priorité conduisent à d'autres bornes, mais aucune ne dispense de parcourir les sommets et les arcs utiles.

Exercice. Donner les algorithmes des fonctions `arbreInitial`, `arcsEntreArbreEtGraphe` et `arcMinimal`, puis celui de la procédure `ajouterCommeFils`.

7.6.2 Guider la recherche avec A*

Considérons la recherche d'un itinéraire de Rouen à Montpellier dans le réseau autoroutier de la figure 7.6. Les sommets du graphe représentent des villes et la valeur de chaque arc est la distance routière entre les deux villes qu'il relie. L'objectif est de trouver un chemin dont la somme des distances est minimale.



Carte du réseau et villes retenues — Graphe valué sous-jacent

FIGURE 7.6 – Du réseau autoroutier à sa représentation par un graphe valué.

L'algorithme de Dijkstra ne connaît que la distance déjà parcourue depuis Rouen. Il développe donc les chemins par coût croissant dans toutes les directions, y compris vers des villes manifestement éloignées de Montpellier. Dans cet exemple, il parcourt pratiquement tout le graphe : seule la

branche menant à Marseille n'est pas explorée avant que Montpellier soit atteint. La partie gauche de la figure 7.7 met en évidence ce grand nombre d'arcs explorés.

Pour orienter la recherche, on tient compte de la position du prochain sommet par rapport à la destination. La distance à vol d'oiseau entre une ville et Montpellier constitue une estimation de la distance qu'il reste à parcourir. Une telle fonction d'estimation est appelée une *heuristique*. Nous la notons h ; pour tout sommet s, $h(s)$ estime le coût d'un chemin allant de s à Montpellier.

Ce nouveau critère conduit à redéfinir la fonction `arcMinimal` :

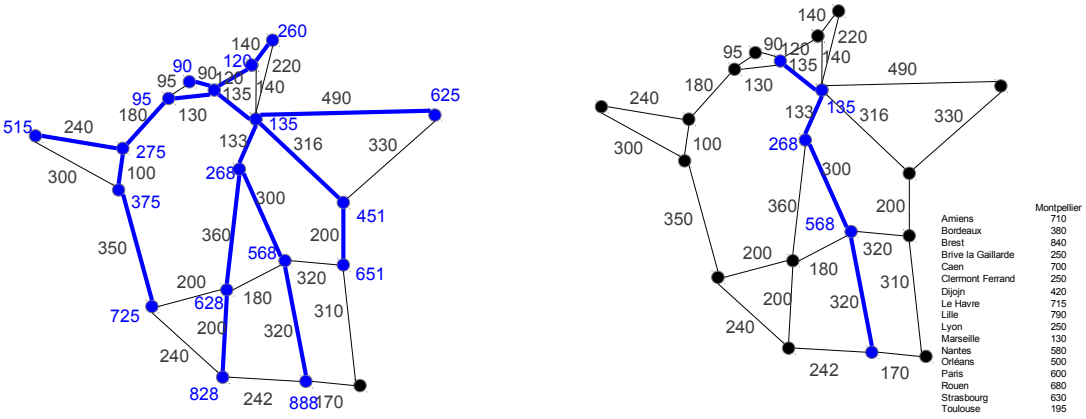
— **fonction** arcMinimal (g : Graphe<Sommet,„**ReelPositif**>,
arcs : Liste<Liste<Sommet>>,
cout : Dictionnaire<Sommet,**ReelPositif**>,
h : fonction(Sommet) : **ReelPositif) :** Sommet, Sommet, **ReelPositif**

⌊**précondition(s)** non estVide(arcs)

Pour chaque arc candidat (u, v), la fonction compare désormais la quantité

$$obtenirValeur(cout, u) + obtenirCout(g, u, v) + h(v).$$

Les deux premiers termes donnent le coût du chemin connu de Rouen à v en passant par u. Le dernier terme estime le coût restant de v à Montpellier. L'algorithme A* choisit donc le sommet qui offre le meilleur compromis entre la distance déjà parcourue et la distance qui semble rester à parcourir. La construction de l'arbre et la mise à jour du dictionnaire des coûts restent identiques à celles de Dijkstra.



Exploration par Dijkstra Exploration par A*

FIGURE 7.7 – Comparaison des arcs explorés pour rechercher un plus court chemin de Rouen à Montpellier. Les arcs explorés sont représentés en bleu.

La partie droite de la figure 7.7 montre l'effet de ce choix. La recherche se dirige rapidement vers le sud et n'examine qu'une petite partie du réseau avant d'atteindre Montpellier. Le même plus court chemin est ainsi trouvé en explorant beaucoup moins d'arcs.

Si $h(s) = 0$ pour tout sommet, le critère redevient celui de Dijkstra. Pour garantir que le chemin trouvé reste optimal, l'heuristique ne doit pas surestimer le coût réel restant ; on dit qu'elle est *admissible*. La distance à vol d'oiseau satisfait cette propriété lorsque les valeurs des arcs sont des distances routières. Une heuristique mal choisie peut, en revanche, faire perdre la garantie d'optimalité.

7.7 Synthèse

Un graphe représente une relation entre des sommets au moyen d'arcs. Selon le problème, ces arcs sont orientés ou non, les sommets sont étiquetés et les arcs sont valués. Le TAD doit exposer uniquement les services cohérents avec ces choix, sans imposer une représentation particulière.

La matrice d'adjacence favorise le test direct de l'existence d'un arc, au prix d'un espace quadratique. Les listes d'adjacence occupent un espace proportionnel au nombre de sommets et d'arcs et conviennent aux graphes creux. La matrice d'incidence met l'accent sur le lien entre chaque arc et ses extrémités. Le choix de la représentation doit donc être guidé par la densité du graphe et par les opérations attendues.

Les parcours en profondeur et en largeur fournissent les mécanismes de base pour explorer un graphe. Le tri topologique ordonne les sommets d'un graphe orienté acyclique en respectant les contraintes d'antériorité. Dijkstra calcule des plus courts chemins lorsque les coûts sont non négatifs ; A* dirige la recherche vers une destination grâce à une heuristique.

Ces quelques algorithmes n'épuisent pas les usages des graphes. Les réseaux routiers, électriques et informatiques, la compilation, le Web sémantique et de nombreux modèles d'apprentissage reposent sur cette structure. D'autres problèmes portent notamment sur les flots, les cliques, l'isomorphisme ou la comparaison de graphes.

Chapitre 8

La programmation dynamique

8.1 Des calculs récursifs inutilement répétés

La récursivité permet souvent d'exprimer directement la définition d'un problème. Cette traduction peut toutefois conduire à recalculer un grand nombre de fois les mêmes valeurs. Le calcul des coefficients binomiaux fournit un premier exemple de ce phénomène.

Pour deux naturels n et p tels que $p \leq n$, le nombre de combinaisons de p éléments parmi n est noté C_n^p. Il satisfait la relation

$$C_n^p = \begin{cases} 1 & \text{si } p = 0 \text{ ou } p = n; \\ C_{n-1}^p + C_{n-1}^{p-1} & \text{sinon.} \end{cases}$$

Une première fonction suit littéralement cette définition.

fonction cnpRecursif (n,p : **Naturel**) **: NaturelNonNul**

⌊**précondition(s)** $p \leq n$

debut
  si $p = 0$ ou $p = n$ **alors**
    retourner 1
  sinon
    retourner cnpRecursif(n-1,p)+cnpRecursif(n-1,p-1)
  finsi
fin

Cette fonction est courte et correcte, mais son arbre d'appels contient de nombreux doublons. Dans le calcul de C_5^3 représenté par la figure 8.1, C_3^2, C_2^1 ou encore C_1^1 sont demandés plusieurs fois. Chaque appel recommence pourtant le calcul sans profiter des résultats déjà obtenus.

Dans le pire cas, le nombre d'appels croît exponentiellement avec n. Ce n'est pas la relation mathématique qui impose ce coût, mais la manière dont elle est évaluée. Il suffit de calculer chaque coefficient nécessaire une seule fois et de conserver sa valeur pour la réutiliser. Cette idée est au cœur de la programmation dynamique.

La traduction C est tout aussi directe que le pseudocode et présente le même coût exponentiel :

```
unsigned int combinaison(unsigned int n, unsigned int p) {
  if (p == 0 || p == n)
    return 1;
  return combinaison(n - 1, p)
       + combinaison(n - 1, p - 1);
}
```

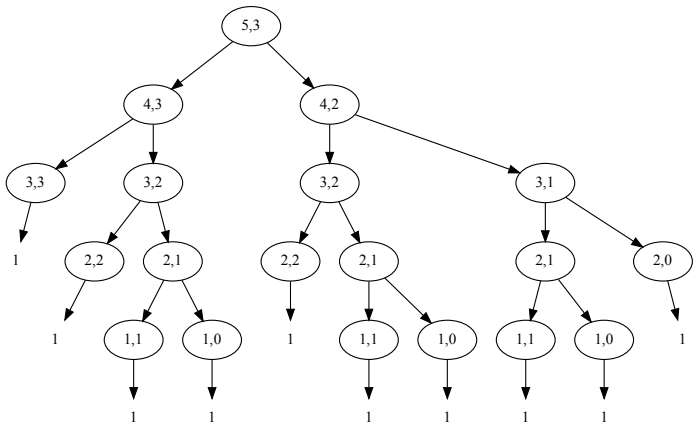


FIGURE 8.1 – Arbre des appels récursifs pour le calcul de C_5^3.

8.2 Principes de la programmation dynamique

La programmation dynamique est un paradigme de conception d'algorithmes [1]. Elle s'applique lorsqu'un problème peut être décomposé en sous-problèmes et que les mêmes sous-problèmes apparaissent plusieurs fois. Contrairement à une application classique de la méthode diviser pour régner, les sous-problèmes ne sont donc pas indépendants.

La méthode repose sur trois propriétés :

— une solution peut être exprimée à partir de solutions de problèmes plus petits ;
— plusieurs décompositions demandent les mêmes sous-problèmes ;
— les résultats intermédiaires peuvent être conservés dans une table et réutilisés.

Pour concevoir un algorithme dynamique, on suit généralement les étapes suivantes :

1. définir précisément l'état d'un sous-problème ;
2. établir les cas de base et une relation de récurrence ;
3. choisir un ordre de calcul dans lequel chaque dépendance est déjà disponible ;
4. stocker chaque résultat dans une table ;
5. si la solution elle-même est recherchée, et pas seulement sa valeur, conserver ou retrouver les choix qui ont conduit à l'optimum.

Deux organisations mettent en œuvre ce principe. La *tabulation* part des cas de base et remplit itérativement la table jusqu'au problème recherché. La *mémoïsation* conserve une formulation récursive, mais associe à chaque état son résultat lors du premier calcul ; les appels suivants consultent cette valeur. Dans ce chapitre, nous emploierons principalement la tabulation, qui rend explicites l'ordre de calcul et l'espace occupé.

Le mot *programmation* est ici employé au sens de planification et non au sens d'écriture d'un programme. La difficulté principale n'est d'ailleurs pas de remplir un tableau, mais de choisir les bons sous-problèmes et d'établir la relation qui les relie.

8.3 Tabuler les coefficients binomiaux

Pour calculer C_n^p, il suffit de connaître les coefficients C_i^j tels que $0 \leq i \leq n$ et $0 \leq j \leq \min(i, p)$. La relation de récurrence impose de calculer une ligne avant la suivante. Les cas de base, tous égaux

1. À ne pas confondre avec un paradigme de programmation. Un paradigme de programmation se décide à la conception préliminaire, alors qu'un paradigme de conception d'algorithmes se décide à la conception détaillée.

à un, forment les bords du triangle de Pascal.

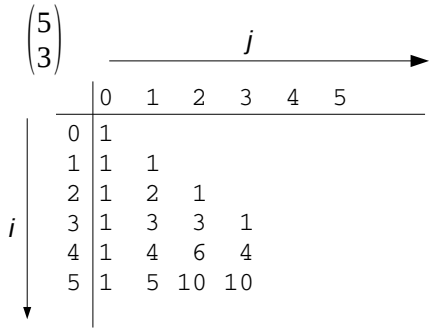


FIGURE 8.2 – Calcul de C_5^3 par remplissage du triangle de Pascal.

Le tableau b mémorise chaque coefficient une seule fois. Les cases telles que $j > i$ ne correspondent à aucun coefficient utile et ne sont pas calculées.

fonction cnpDynamique (n,p : **Naturel**) **: NaturelNonNul**

⌊**précondition(s)** $p \leq n$

Déclaration i,j : **Naturel**
b : tableau[0..n,0..p] de **NaturelNonNul**

debut
  pour i ←0 **à** n **faire**
    pour j ←0 **à** min(i,p) **faire**
      si $j = 0$ ou $j = i$ **alors**
        b[i,j] ← 1
      sinon
        b[i,j] ← b[i-1,j-1]+b[i-1,j]
      finsi
    finpour
  finpour
  retourner b[n,p]
fin

Au plus $(n + 1)(p + 1)$ cases sont remplies. La complexité temporelle est donc en $O(np)$ et l'espace utilisé est en $O(np)$, contre un nombre exponentiel d'appels pour la version récursive naïve.

8.3.1 Réduire le tableau à une dimension

Pour calculer une ligne, seules les valeurs de la ligne précédente sont nécessaires. Il est même possible de mettre à jour un seul tableau de $p + 1$ cases. Les indices doivent alors être parcourus de droite à gauche. Ainsi, au moment de calculer $b[j]$, les anciennes valeurs $b[j]$ et $b[j - 1]$ n'ont pas encore été remplacées.

fonction cnpDynamiqueCompact (n,p : **Naturel**) **: NaturelNonNul**

⌊**précondition(s)** $p \leq n$

Déclaration i,j : **Naturel**
b : tableau[0..p] de **Naturel**

debut
  pour j ←0 **à** p **faire**
    b[j] ← 0

```
    finpour
    b[0] ← 1
    pour i ←1 à n faire
        pour j ←min(i,p) à 1 pas de -1 faire
            b[j] ← b[j]+b[j-1]
        finpour
    finpour
    retourner b[p]
fin
```

La complexité temporelle reste en $O(np)$, mais l'espace supplémentaire devient $O(p)$. Parcourir j dans l'ordre croissant serait incorrect : une case utiliserait alors une valeur appartenant déjà à la nouvelle ligne.

Une traduction en C peut utiliser un tableau de taille variable, disponible à partir de C99 :

```
unsigned int combinaison(unsigned int n, unsigned int p) {
  unsigned int b[p + 1];
  unsigned int i, j, borne;

  for (j = 0; j <= p; j++)
    b[j] = 0;
  b[0] = 1;
  for (i = 1; i <= n; i++) {
    borne = i < p ? i : p;
    for (j = borne; j > 0; j--)
      b[j] = b[j] + b[j - 1];
  }
  return b[p];
}
```

La figure 8.3 compare les temps mesurés pour la version récursive et la version dynamique. À l'échelle du graphique, le temps de la seconde reste proche de l'axe horizontal lorsque celui de la première augmente rapidement.

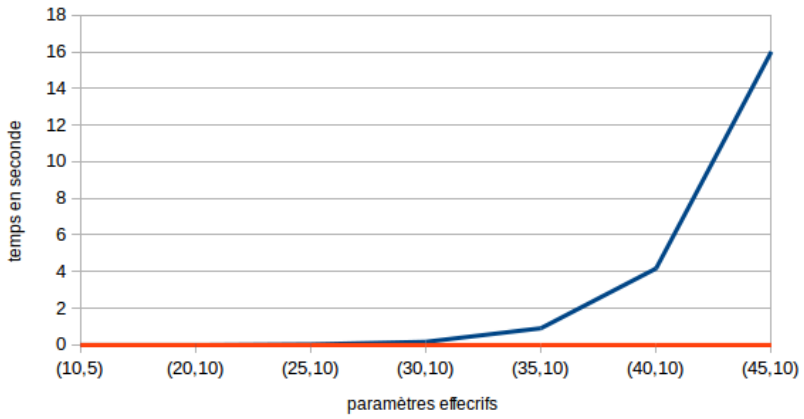


FIGURE 8.3 – Temps d'exécution du calcul récursif (bleu) et du calcul dynamique (rouge).

8.4 Un autre exemple classique : le problème du sac à dos

Considérons n objets. L'objet i possède un poids w_i et une valeur v_i. Un sac peut contenir un poids total au plus égal à W. Le problème consiste à choisir les objets qui maximisent la valeur

transportée sans dépasser cette capacité.

Une variable binaire x_i indique si l'objet i est retenu. La formulation mathématique est donc

$$\text{maximiser } \sum_{i=1}^{n} x_i v_i \quad \text{sous la contrainte} \quad \sum_{i=1}^{n} x_i w_i \leq W, \qquad x_i \in \{0,1\}.$$

Chaque objet peut être pris au plus une fois.

La fonction à concevoir renvoie l'ensemble des indices des objets retenus :

— **fonction** resoudreSacADos (tailleDuSac,nbObjets : **NaturelNonNul** ;
valeurs,poids : tableau[1..MAX_OBJETS] de **Naturel) :** Ensemble<**NaturelNonNul**>

|**précondition(s)** $nbObjets \leq MAX_OBJETS$

Nous utiliserons l'instance suivante tout au long de la section :

Objet	1	2	3	4	5
Valeur v_i	1	6	18	22	28
Poids w_i	1	2	5	6	7

La capacité du sac est $W = 11$. Les objets 3 et 4 occupent exactement cette capacité et fournissent une valeur totale de 40, qui est optimale.

8.4.1 Explorer tous les choix

Pour chaque objet, une recherche exhaustive examine deux possibilités : ne pas le retenir ou le retenir. La fonction récursive reçoit donc l'ensemble des objets déjà choisis et celui des objets qui restent à examiner. Les ensembles sont ici manipulés comme des valeurs : affecter ou transmettre un ensemble produit une valeur indépendante que l'appel peut modifier.

```
fonction resoudreSacADosR (tailleDuSac : NaturelNonNul ;
objetsChoisis,objetsRestants : Ensemble<NaturelNonNul> ;
valeurs,poids : tableau[1..MAX_OBJETS] de Naturel) : Ensemble<NaturelNonNul>
   Déclaration  sansObjet,avecObjet,choixAvecObjet : Ensemble<NaturelNonNul>
                objet : NaturelNonNul
debut
   si somme(objetsChoisis,poids)>tailleDuSac alors
      retourner ensemble()
   sinon
      si estVide(objetsRestants) alors
         retourner objetsChoisis
      sinon
         obtenirSupprimerUnElement(objetsRestants,objet)
         choixAvecObjet ← objetsChoisis
         ajouter(choixAvecObjet,objet)
         sansObjet ← resoudreSacADosR(tailleDuSac, objetsChoisis, objetsRestants, valeurs, poids)
         avecObjet ← resoudreSacADosR(tailleDuSac, choixAvecObjet, objetsRestants, valeurs,poids)
         si somme(sansObjet,valeurs)>somme(avecObjet,valeurs) alors
            retourner sansObjet
         sinon
            retourner avecObjet
         finsi
      finsi
   finsi
fin
```

L'appel initial utilise un ensemble de choix vide et l'ensemble $\{1, \ldots, n\}$ comme objets restants. Une branche qui dépasse la capacité renvoie l'ensemble vide ; une branche qui a examiné tous les objets renvoie les objets choisis. Dans le dernier cas, les deux solutions récursives sont comparées par leur valeur totale.

L'arbre de recherche possède jusqu'à 2^n feuilles, une par sous-ensemble d'objets. La complexité est donc exponentielle. Cette solution reste utile pour comprendre le problème et pour valider de petites instances, mais elle devient rapidement impraticable.

8.4.2 Définir les sous-problèmes

Pour construire une solution dynamique, notons $V_{i,j}$ la valeur maximale obtenue en choisissant parmi les i premiers objets avec une capacité j. Le problème initial consiste alors à calculer $V_{n,W}$.

Les cas de base sont immédiats. Sans objet, la valeur est nulle quelle que soit la capacité ; avec une capacité nulle, aucun objet de poids strictement positif ne peut être ajouté :

$$V_{0,j} = 0 \qquad \text{et} \qquad V_{i,0} = 0.$$

Pour $i > 0$ et $j > 0$, deux situations se présentent. Si $w_i > j$, l'objet i ne tient pas dans le sac et la meilleure valeur reste $V_{i-1,j}$. Sinon, on compare la solution qui ne le retient pas avec celle qui le retient. Dans ce second cas, l'objet apporte v_i et laisse une capacité $j - w_i$ aux objets précédents. On obtient la relation

$$V_{i,j} = \begin{cases} V_{i-1,j} & \text{si } w_i > j; \\ \max(V_{i-1,j}, v_i + V_{i-1,j-w_i}) & \text{sinon.} \end{cases}$$

Chaque ligne ne dépend que de la ligne précédente. Le tableau peut donc être rempli de haut en bas et, dans chaque ligne, dans n'importe quel ordre. La figure 8.4 montre le tableau obtenu pour l'exemple. Sa case inférieure droite contient $V_{5,11} = 40$.

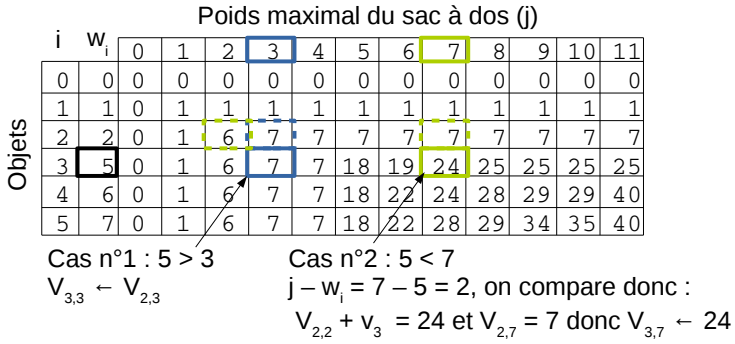


FIGURE 8.4 – Table des valeurs optimales pour l'instance du sac à dos.

8.4.3 Calculer la valeur optimale

L'algorithme initialise la première ligne et la première colonne, puis applique la relation précédente à chaque autre case.

fonction toutesLesValeursPossibles (tailleDuSac,nbObjets : **NaturelNonNul** ;
valeurs,poids : tableau[1..MAX_OBJETS] de **Naturel**) **:** tableau[0..nbObjets,0..tailleDuSac] de **Naturel**

⌊**précondition(s)** $nbObjets \leq MAX_OBJETS$ et
$tailleDuSac \leq TAILLE_MAX_SAC$

Déclaration v : tableau[0..nbObjets,0..tailleDuSac] de **Naturel**
i,j : **Naturel**

debut
pour i ←0 **à** nbObjets **faire**
v[i,0] ← 0

```
    finpour
    pour j ←0 à tailleDuSac faire
        v[0,j] ← 0
    finpour
    pour i ←1 à nbObjets faire
        pour j ←1 à tailleDuSac faire
            si poids[i] ≤ j alors
                v[i,j] ← max(v[i-1,j],valeurs[i]+v[i-1,j-poids[i]])
            sinon
                v[i,j] ← v[i-1,j]
            finsi
        finpour
    finpour
    retourner v
fin
```

La table comporte $(n+1)(W+1)$ cases. Son remplissage demande un temps en $O(nW)$ et un espace en $O(nW)$. Cette complexité est dite *pseudo-polynomiale* : elle est polynomiale en la valeur numérique W, mais pas en le nombre de bits nécessaires pour représenter W.

8.4.4 Retrouver les objets choisis

La valeur $V_{n,W}$ ne donne pas directement les objets qui la produisent. Il faut remonter dans la table depuis la case (n, W). Si $V_{i,j} = V_{i-1,j}$, une solution optimale existe sans l'objet i et la capacité reste j. Si les deux valeurs diffèrent, l'objet i a été retenu ; on l'ajoute au résultat et on poursuit depuis la capacité $j - w_i$.

Dans la figure 8.5, la remontée part de $V_{5,11}$. L'objet 5 n'est pas retenu, puis les objets 4 et 3 le sont. Leur poids total vaut $6 + 5 = 11$ et leur valeur totale $22 + 18 = 40$.

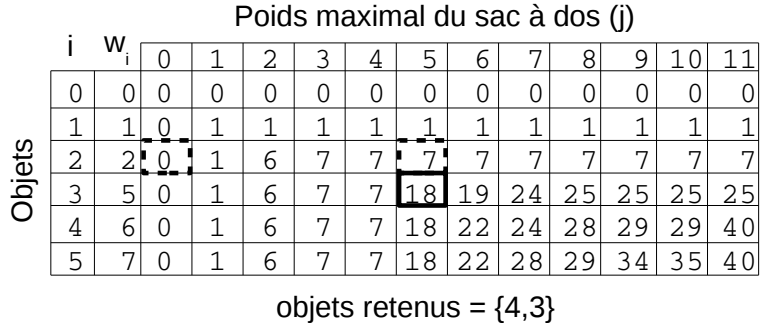


FIGURE 8.5 – Reconstruction de la solution optimale à partir de la table.

```
fonction rechercherObjetsARetenir (tailleDuSac,nbObjets : NaturelNonNul ;
v : tableau[0..nbObjets,0..tailleDuSac] de Naturel ;
poids : tableau[1..MAX_OBJETS] de Naturel) : Ensemble<NaturelNonNul>
    Déclaration  i,j : Naturel
                 resultat : Ensemble<NaturelNonNul>
debut
    resultat ← ensemble()
    i ← nbObjets
    j ← tailleDuSac
    tant que i > 0 et j > 0 faire
        si v[i,j]≠v[i-1,j] alors
```

```
            ajouter(resultat,i)
            j ← j-poids[i]
        finsi
        i ← i-1
    fintantque
    retourner resultat
fin
```

La reconstruction examine au plus une fois chaque objet et coûte donc $O(n)$. La fonction principale appelle d'abord `toutesLesValeursPossibles`. Elle transmet ensuite la table obtenue à `rechercherObjetsARetenir`.

Les mesures de la figure 8.6 illustrent l'écart entre l'exploration exhaustive et la programmation dynamique. La courbe rouge de la version dynamique reste presque confondue avec l'axe des abscisses à l'échelle imposée par la croissance de la version exhaustive.

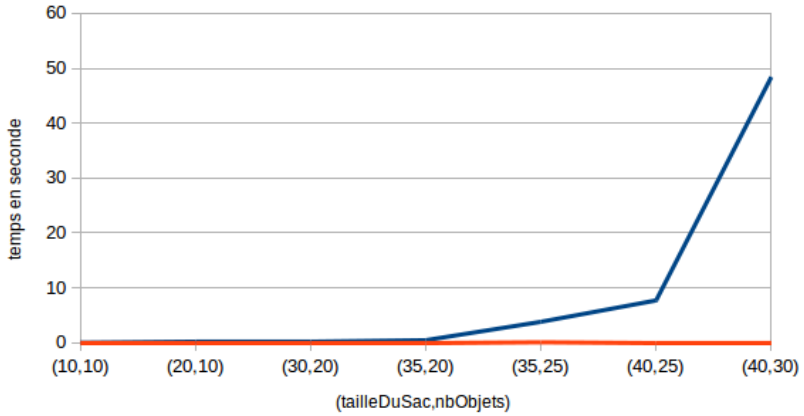


FIGURE 8.6 – Temps d'exécution de la force brute (bleu) et de la programmation dynamique (rouge).

8.5 Synthèse et autres applications

La programmation dynamique remplace la répétition de sous-problèmes par leur mémorisation. Elle est particulièrement utile lorsqu'une solution exacte est recherchée, qu'une relation de récurrence est disponible et qu'un algorithme récursif naïf résout plusieurs fois les mêmes instances.

Le gain en temps a pour contrepartie l'espace occupé par la table. Une fois la relation établie, il faut donc examiner quelles valeurs anciennes sont encore nécessaires. Le calcul des coefficients binomiaux montre qu'une table à deux dimensions peut parfois être réduite à une seule ligne. Pour le sac à dos, une réduction analogue suffit pour connaître la valeur optimale, mais la table complète facilite la reconstruction des objets choisis. L'espace n'est donc pas seulement déterminé par le calcul de la valeur : il dépend aussi du résultat que l'interface doit fournir.

De nombreux problèmes reposent sur la même démarche :

— la distance de Levenshtein entre deux chaînes, fondée sur les coûts d'insertion, de suppression et de substitution ;
— le rendu de monnaie ;
— le voyageur de commerce, pour lequel une solution dynamique classique ramène une exploration en $O(n!)$ à $O(n^2 2^n)$;
— le choix du parenthésage d'une chaîne de multiplications de matrices.

Le dernier exemple rappelle que des expressions mathématiquement équivalentes peuvent avoir des coûts très différents. Si les dimensions de A_1, A_2 et A_3 sont respectivement $(10, 100)$, $(100, 5)$ et $(5, 50)$, le calcul de $(A_1A_2)A_3$ demande $10 \times 100 \times 5 + 10 \times 5 \times 50 = 7500$ multiplications scalaires. Le calcul de $A_1(A_2A_3)$ en demande $100 \times 5 \times 50 + 10 \times 100 \times 50 = 75000$. Pour une longue chaîne, la programmation dynamique permet de choisir le meilleur parenthésage sans les énumérer tous.

Une table dynamique peut aussi être interprétée comme un graphe orienté acyclique : chaque état est un sommet et chaque dépendance relie un sous-problème à un autre. Cette lecture rapproche la programmation dynamique des algorithmes de chemin optimal du chapitre précédent.

La méthode ne rend pas nécessairement le problème facile. Le sac à dos reste pseudo-polynomial et la solution dynamique du voyageur de commerce reste exponentielle. Lorsqu'une solution exacte est trop coûteuse, il peut être préférable de rechercher une solution approchée à l'aide, par exemple, d'un algorithme glouton, d'un recuit simulé ou d'une recherche tabou.

Chapitre 9

Les arbres B

9.1 Des arbres adaptés au stockage externe

Un arbre binaire de recherche équilibré garantit une hauteur logarithmique, mais chacun de ses nœuds ne contient qu'une clé et au plus deux liens. Si les données sont stockées sur disque, une recherche peut ainsi provoquer un accès à un bloc pour chaque niveau visité. Or un bloc contient beaucoup plus d'espace qu'une clé et deux adresses.

Un *arbre B* regroupe plusieurs clés dans chaque nœud et possède donc plusieurs sous-arbres. Un nœud peut être dimensionné pour occuper un bloc ou un cluster de l'unité de stockage. La forte ramification réduit alors la hauteur de l'arbre et, avec elle, le nombre d'entrées-sorties. Cette structure est notamment employée pour les index de bases de données et dans des systèmes de fichiers.

Comme un arbre AVL, un arbre B reste équilibré après les insertions et les suppressions. L'équilibre ne repose toutefois pas sur des rotations binaires, mais sur la répartition des clés entre des nœuds voisins, leur éclatement et leur fusion.

9.2 Définition et organisation

Dans la convention de ce cours, un arbre B d'ordre m respecte les invariants suivants :

1. chaque nœud non racine contient entre m et $2m$ clés ;
2. la racine contient entre une et $2m$ clés, sauf lorsque l'arbre est vide ;
3. les clés d'un nœud sont triées ;
4. un nœud qui contient k clés est soit une feuille, soit un nœud interne possédant exactement $k+1$ fils ;
5. toutes les feuilles se trouvent à la même profondeur.

Soient $k_1 < k_2 < \cdots < k_k$ les clés d'un nœud interne et $P_0, P_1, \ldots, P_k$ ses fils. Toutes les clés de P_0 sont inférieures à k_1, celles de P_i sont comprises entre k_i et k_{i+1}, et celles de P_k sont supérieures à k_k. La figure 9.1 résume cette organisation.

La figure 9.2 présente un arbre B d'ordre 2. Chaque nœud non racine contient donc entre deux et quatre clés. Les feuilles sont toutes au même niveau et chaque clé de la racine ou d'un nœud interne sépare les intervalles de valeurs de ses fils.

9.2.1 Capacité et hauteur

On appelle hauteur h le nombre d'arcs d'un chemin entre la racine et une feuille. Un arbre B d'ordre m et de hauteur h contient au minimum

$$2(m+1)^h - 1$$

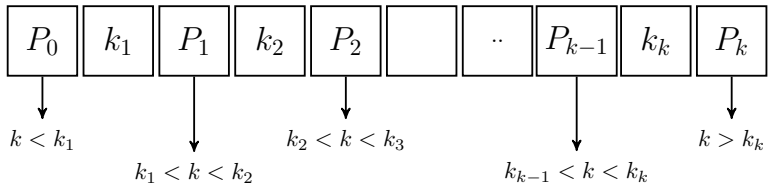


FIGURE 9.1 – Clés et références contenues dans un nœud d'arbre B.

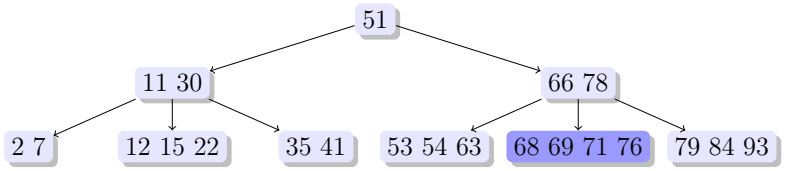


FIGURE 9.2 – Exemple d'arbre B d'ordre 2.

clés et au maximum

$$(2m+1)^{h+1} - 1$$

clés. La hauteur augmente donc logarithmiquement avec le nombre de clés. Par exemple, pour $m = 100$ et $h = 2$, l'arbre peut contenir jusqu'à $201^3 - 1 = 8\,120\,600$ clés. Une recherche n'examine pourtant que trois nœuds : la racine, un nœud intermédiaire et une feuille.

Cette propriété explique l'intérêt d'un ordre élevé pour le stockage externe. Le parcours de plusieurs clés déjà chargées en mémoire coûte généralement moins cher qu'un accès supplémentaire au disque.

9.2.2 TAD ArbreB

Le type des clés possède un ordre total. L'interface publique permet de créer un arbre de l'ordre souhaité, de rechercher une clé et de modifier l'ensemble des clés. Les doublons ne sont pas autorisés dans cette spécification.

Nom: ArbreB

Paramètre: Cle possédant un ordre total

Utilise: **NaturelNonNul**,**Booleen**

Opérations:

arbreB:	**NaturelNonNul**	$\rightarrow$ ArbreB
ordre:	ArbreB	$\rightarrow$ **NaturelNonNul**
estPresent:	ArbreB $\times$ Cle	$\rightarrow$ **Booleen**
inserer:	ArbreB $\times$ Cle	$\nrightarrow$ ArbreB
supprimer:	ArbreB $\times$ Cle	$\rightarrow$ ArbreB

Préconditions: inserer(a,c): non estPresent(a,c)

Axiomes:
- $ordre(arbreB(n)) = n$
- $estPresent(inserer(a, c), c)$
- $non\ estPresent(supprimer(a, c), c)$

Une implantation en mémoire associe à chaque nœud son nombre de clés, un tableau de clés triées et un tableau de références vers les fils. Les tableaux doivent prévoir temporairement $2m+1$ clés et $2m+2$ fils afin de représenter un débordement avant l'éclatement. Sur disque, les références sont des identifiants de blocs plutôt que des pointeurs mémoire.

9.3 Rechercher une clé

La recherche généralise celle d'un arbre binaire de recherche. Dans chaque nœud, on localise la première clé supérieure ou égale à la clé recherchée. Une égalité termine la recherche avec succès. Sinon, la position obtenue détermine l'unique fils susceptible de contenir la clé. Atteindre une feuille sans trouver d'égalité termine la recherche avec un échec.

Dans l'algorithme suivant, les clés sont numérotées de 1 à `nbCles(a)` et les fils de 0 à `nbCles(a)`.

```
fonction estPresent (a : ArbreB, c : Cle) : Booleen
   Déclaration  i : NaturelNonNul
debut
   si estVide(a) alors
      retourner FAUX
   sinon
      i ← 1
      tant que i ≤ nbCles(a) et cle(a, i) < c faire
         i ← i+1
      fintantque
      si i ≤ nbCles(a) et cle(a, i) = c alors
         retourner VRAI
      sinon
         si estUneFeuille(a) alors
            retourner FAUX
         sinon
            retourner estPresent(sousArbre(a,i-1),c)
         finsi
      finsi
   finsi
fin
```

La recherche de la position dans le nœud est ici linéaire. Elle peut être remplacée par une recherche dichotomique puisque les clés sont triées. Pour un arbre en mémoire, le coût tient compte à la fois de la hauteur et de cette recherche locale. Pour un arbre stocké sur disque, le nombre de blocs chargés, donc le nombre de niveaux visités, est souvent le facteur dominant.

La recherche descend dans un seul fils à chaque niveau. Elle effectue ainsi $h+1$ visites de nœuds au pire, soit un nombre logarithmique dans le nombre de clés lorsque l'ordre est fixé.

9.4 Insérer une clé

L'insertion commence par la recherche de la feuille dans laquelle la nouvelle clé doit apparaître. La clé est insérée à sa position dans le tableau trié. Si la feuille contient alors au plus $2m$ clés, tous les invariants sont encore respectés et l'opération est terminée.

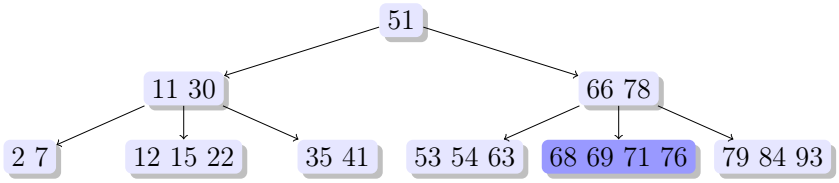
Un nœud qui atteint $2m + 1$ clés déborde. Il est alors *éclaté* :

1. les m plus petites clés restent dans un nœud gauche ;
2. les m plus grandes clés forment un nœud droit ;
3. la clé médiane, en position $m + 1$, remonte dans le père ;
4. dans le cas d'un nœud interne, ses $2m + 2$ fils sont répartis par moitié entre les deux nouveaux nœuds.

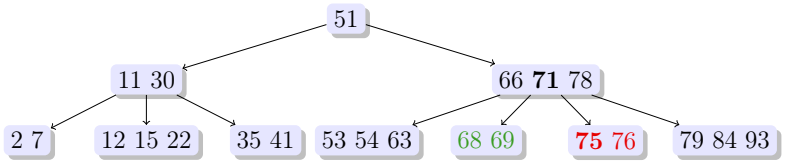
L'ajout de la médiane peut faire déborder le père. Le même traitement se propage alors vers la racine. L'éclatement de la racine constitue le seul cas où l'insertion augmente la hauteur : la médiane devient l'unique clé d'une nouvelle racine dont les deux fils sont les parties issues de l'éclatement.

9.4.1 Insertion sans changement de hauteur

Dans l'arbre d'ordre 2 de la figure 9.3, la clé 75 doit rejoindre la feuille [68 69 71 76]. Cette feuille possède déjà quatre clés. Après l'insertion, les cinq clés sont [68 69 71 75 76]. La médiane 71 remonte dans le père ; les feuilles [68 69] et [75 76] restent de part et d'autre.



Avant l'insertion



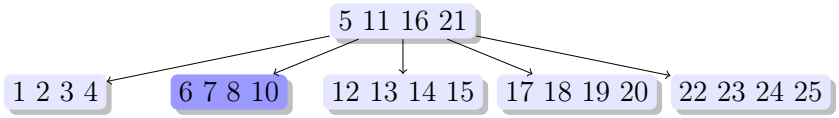
Après l'éclatement et la remontée de 71

FIGURE 9.3 – Insertion de la clé 75 dans un nœud plein.

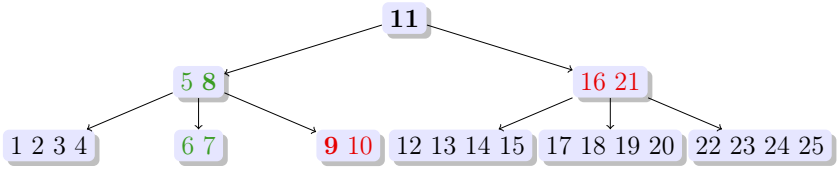
Le père passe de deux à trois clés et ne déborde pas. L'arbre conserve donc la même hauteur.

9.4.2 Augmentation de la hauteur

La figure 9.4 illustre une propagation jusqu'à la racine. L'insertion de 9 fait déborder la feuille contenant [6 7 8 10]. La clé 8 remonte, mais la racine contient alors cinq clés et déborde à son tour. Son éclatement fait remonter 11 dans une nouvelle racine. Tous les anciens niveaux sont repoussés d'un niveau vers le bas.



Arbre initial



Nouvelle racine après insertion de 9

FIGURE 9.4 – Propagation d'un éclatement jusqu'à la racine.

9.4.3 Organisation de l'algorithme

Une implantation récursive fait remonter au père un résultat de type `Promotion`. Ce résultat vaut soit `AUCUNE`, soit un triplet $(cle, gauche, droit)$ produit par l'éclatement d'un nœud. L'opération `integrerPromotion(a,i,p)` remplace le fils i de a par les deux nœuds de la promotion

p et insère entre eux sa clé. L'opération `eclater(a)` répartit les $2m + 1$ clés de a et renvoie la promotion ainsi obtenue.

L'algorithme récursif suivant descend jusqu'à la feuille, puis traite les éventuels débordements en remontant.

```
fonction insererRec (a : ArbreB, c : Cle) : Promotion
    Déclaration  i : NaturelNonNul, p : Promotion
debut
    i ← positionDInsertion(a,c)
    si estUneFeuille(a) alors
        insererCleDansFeuille(a,c,i)
    sinon
        p ← insererRec(sousArbre(a,i-1),c)
        si p ≠ AUCUNE alors
            integrerPromotion(a,i-1,p)
        finsi
    finsi
    si nbCles(a) > 2 × ordre(a) alors
        retourner eclater(a)
    sinon
        retourner AUCUNE
    finsi
fin
```

L'opération publique traite séparément l'arbre vide et la promotion éventuelle de l'ancienne racine. Dans ce dernier cas, elle crée le niveau supplémentaire.

```
procédure inserer (E/S a : ArbreB, E c : Cle)
    ⌊précondition(s)  non estPresent(a,c)
    Déclaration  p : Promotion
debut
    si estVide(a) alors
        a ← feuille(c,ordre(a))
    sinon
        p ← insererRec(a,c)
        si p ≠ AUCUNE alors
            a ← noeud(p.cle,p.gauche,p.droit,ordre(a))
        finsi
    finsi
fin
```

Le résultat explicite de type `Promotion` évite de confondre un véritable éclatement avec un sous-arbre qui posséderait naturellement une seule clé, notamment dans un arbre d'ordre 1.

Une insertion descend sur un unique chemin, puis peut effectuer au plus un éclatement par niveau en remontant. Son nombre de visites de nœuds est donc en $O(h)$, auquel s'ajoutent les décalages de clés à l'intérieur des nœuds.

9.5 Supprimer une clé

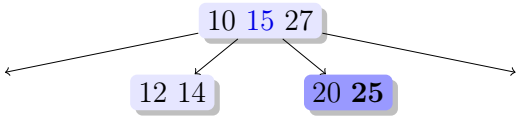
La suppression doit préserver simultanément l'ordre des clés, le nombre de fils et la borne minimale de m clés par nœud non racine. Elle est donc plus délicate que l'insertion. Le traitement peut être compris à partir de quatre situations : la suppression simple dans une feuille, la fusion, la redistribution et la suppression d'une clé interne.

9.5.1 Supprimer dans une feuille

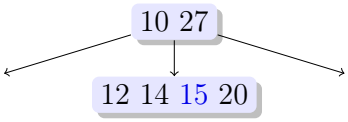
Si la feuille contient plus de m clés avant l'opération, il suffit de retirer la clé et de décaler les suivantes. Elle contient encore au moins m clés et aucun rééquilibrage n'est nécessaire.

En revanche, retirer une clé d'une feuille qui en contenait exactement m laisse seulement $m-1$ clés. Le nœud est alors sous-rempli. On le combine avec un frère adjacent et avec la clé du père qui séparait les deux nœuds.

Dans la figure 9.5, supprimer 25 laisse la feuille [20]. Sa combinaison avec [12 14] et avec la clé séparatrice 15 produit [12 14 15 20]. Ce nœud respecte la capacité maximale de quatre clés. La clé 15 et l'une des références sont retirées du père.



Avant la suppression de 25



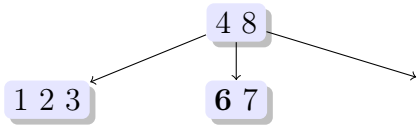
Après la fusion

FIGURE 9.5 – Correction d'un sous-remplissage par fusion.

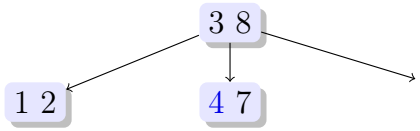
9.5.2 Redistribuer les clés

La combinaison avec un frère qui contient plus de m clés peut produire plus de $2m$ clés. Elle est alors suivie d'un nouvel éclatement. Cette opération redistribue les clés entre deux nœuds et remplace la clé séparatrice du père par une nouvelle médiane.

Dans la figure 9.6, supprimer 6 laisse [7]. En combinant cette clé avec le frère [1 2 3] et la clé 4 du père, on obtient cinq clés. Elles sont à nouveau partagées : 3 remonte dans le père, tandis que [1 2] et [4 7] deviennent les deux fils.



Avant la suppression de 6



Après la redistribution

FIGURE 9.6 – Fusion temporaire suivie d'un éclatement.

9.5.3 Supprimer une clé interne

Une clé interne sépare deux intervalles de valeurs et ne peut pas simplement être effacée. On la remplace par une clé adjacente dans l'ordre total :

— soit la plus grande clé du sous-arbre situé immédiatement à gauche, appelée prédécesseur ;

— soit la plus petite clé du sous-arbre situé immédiatement à droite, appelée successeur.

Cette clé de remplacement appartient à une feuille. Après sa copie dans le nœud interne, elle est supprimée récursivement de cette feuille. Les cas de fusion ou de redistribution précédents s'appliquent alors si nécessaire. Dans la figure 9.7, la clé interne 5 est remplacée par son prédécesseur 4, qui est ensuite retiré de la feuille gauche.

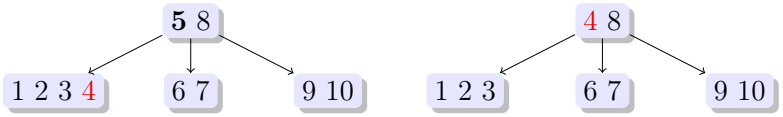
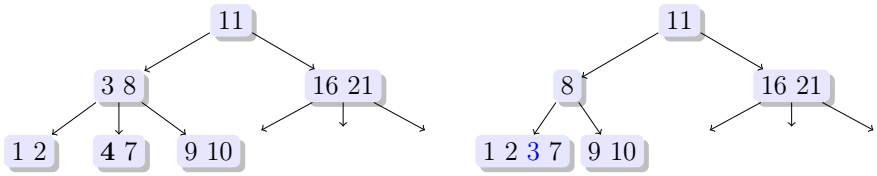


FIGURE 9.7 – Suppression d'une clé interne par substitution avec son prédécesseur.

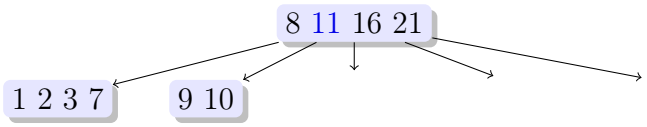
9.5.4 Propagation et diminution de la hauteur

Une fusion retire une clé au père. Si ce père devient à son tour sous-rempli, la correction se propage vers le haut. Lorsque la racine perd sa dernière clé et ne possède plus qu'un fils, ce fils devient la nouvelle racine. C'est le seul cas où une suppression diminue la hauteur de l'arbre.

La figure 9.8 montre une telle propagation dans un arbre d'ordre 2. La suppression de 4 provoque d'abord une fusion au niveau des feuilles, puis un sous-remplissage du nœud interne. La descente de la clé 11 regroupe finalement les deux sous-arbres sous une seule racine et réduit la hauteur d'une unité.



Avant la suppression de 4



Après diminution de la hauteur

FIGURE 9.8 – Propagation d'une fusion jusqu'à la racine.

9.5.5 Algorithme de suppression

Le pseudocode ci-dessous formalise le traitement précédent. L'opération `position(a,c)` renvoie la position de la première clé supérieure ou égale à c. Après une suppression récursive dans le fils i, l'opération `corrigerSousRemplissage(a,i)` rétablit la borne minimale si ce fils ne contient

plus que $m-1$ clés.

```
procédure supprimerRec (E/S a : ArbreB, E c : Cle)
    Déclaration  i : NaturelNonNul, pred : Cle
debut
    i ← position(a,c)
    si i ≤ nbCles(a) et cle(a,i) = c alors
        si estUneFeuille(a) alors
            supprimerCle(a,i)
        sinon
            pred ← plusGrandeCle(sousArbre(a,i-1))
            changerCle(a,i,pred)
            supprimerRec(sousArbre(a,i-1),pred)
            corrigerSousRemplissage(a,i-1)
        finsi
    sinon
        si non estUneFeuille(a) alors
            supprimerRec(sousArbre(a,i-1),c)
            corrigerSousRemplissage(a,i-1)
        finsi
    finsi
fin
```

Pour corriger le sous-remplissage, on emprunte d'abord une clé à un frère qui en possède plus de m. La clé séparatrice du père descend dans le fils et une clé du frère la remplace dans le père. Si aucun frère ne peut céder de clé, le fils est fusionné avec l'un de ses frères et avec leur clé séparatrice. Les opérations de déplacement mettent à jour simultanément les clés et, pour des nœuds internes, les références vers les sous-arbres.

```
procédure corrigerSousRemplissage (E/S a : ArbreB, E i : Naturel)
debut
    si nbCles(sousArbre(a,i)) < ordre(a) alors
        si i > 0 et nbCles(sousArbre(a,i-1)) > ordre(a) alors
            deplacerDepuisFrereGauche(a,i)
        sinon
            si i < nbCles(a) et nbCles(sousArbre(a,i+1)) > ordre(a) alors
                deplacerDepuisFrereDroit(a,i)
            sinon
                si i > 0 alors
                    fusionnerAvecFrereGauche(a,i)
                sinon
                    fusionnerAvecFrereDroit(a,i)
                finsi
            finsi
        finsi
    finsi
fin
```

L'opération publique élimine enfin une racine devenue vide. Si elle possède un fils, celui-ci devient la nouvelle racine ; sinon l'arbre devient vide.

```
procédure supprimer (E/S a : ArbreB, E c : Cle)
debut
    si non estVide(a) alors
        supprimerRec(a,c)
        si nbCles(a) = 0 alors
            si estUneFeuille(a) alors
                a ← arbreVide(ordre(a))
            sinon
                a ← sousArbre(a,0)
            finsi
        finsi
```

finsi
fin

Une suppression descend d'abord le long d'un chemin de recherche et peut ensuite corriger au plus un nœud par niveau. Comme l'insertion, elle effectue donc $O(h)$ visites de nœuds, auxquelles s'ajoutent les déplacements de clés et de références au sein des blocs modifiés.

9.6 Synthèse

L'arbre B conserve les propriétés de recherche d'un arbre ordonné tout en regroupant de nombreuses clés dans un même nœud. Ses invariants imposent un taux de remplissage minimal aux nœuds non racines et placent toutes les feuilles à la même profondeur. Ils garantissent ainsi une faible hauteur.

La recherche choisit un seul fils par niveau. L'insertion se fait dans une feuille et répare les débordements par éclatement et promotion d'une médiane. La suppression remplace d'abord les clés internes par un prédécesseur ou un successeur, puis corrige les sous-remplissages par fusion ou redistribution. Seul un éclatement de la racine augmente la hauteur ; seule la disparition de la dernière clé de la racine peut la diminuer.

Propriété	Arbre AVL	Arbre B
Clés par nœud	Une	Plusieurs
Nombre maximal de fils	Deux	$2m + 1$
Équilibrage	Rotations	Éclatements, fusions, redistribution
Hauteur	Logarithmique	Logarithmique et généralement faible
Usage privilégié	Structures en mémoire	Index et stockage externe

Le coût réel d'une implantation sur disque ne se résume pas au nombre de comparaisons. Il faut dimensionner un nœud en fonction de la taille d'un bloc, organiser la lecture et l'écriture des blocs modifiés et limiter les accès au support. C'est précisément sur ce point que la forte ramification de l'arbre B apporte son principal avantage par rapport aux arbres binaires équilibrés.