INF3105 – Structures de données et algorithmes
Notes de cours *

Éric Beaudry
Département d'informatique
Université du Québec à Montréal (UQAM)
beaudry.eric@uqam.ca

Automne 2018

*Version 1.3. Document généré le 15 décembre 2018.

Table des matières

Première partie

Introduction

1 Introduction et rappel de notions de base

L'objectif du cours INF3105 est d'approfondir le sujet des structures de données et des algorithmes fondamentaux en informatique. Les structures de données permettent d'organiser l'information dans la mémoire d'une machine. Des algorithmes exploitent les structures de données afin de résoudre efficacement des problèmes précis.

Cette section fait quelques rappels de notions de base utiles pour le cours INF3105.

1.1 Lectures approfondies

Les présentes notes de cours résument l'essentiel du cours. Elles ne remplacent pas un « bon livre ». Pour aller plus en profondeur au sujet des structures de données, la référence recommandée est [GTM11]. Pour approfondir le langage C++, consultez le livre de Bjarne Stroustrup [Str10], l'inventeur du langage C++. Une traduction en français est également disponible. Le site Web `http://cppreference.com` est également une excellente référence technique sur C++.

D'autres livres sont disponibles à la Coop et à la Bibliothèque des sciences de l'UQAM dont : [LJR12], [KW06], [KW05] et [Sav09]. N'hésitez pas à rechercher « data structures » et « C++ » dans `http://virtuose.uqam.ca/`. Parmi les références en français, couvrant les structures de données et le langage C++, vous pouvez considérer le livre [Gab05] de Philippe Gabrini, ancien professeur ayant donné le cours INF3105 à plusieurs reprises. Toutefois, soyez conscients que les implémentations présentées dans ces références peuvent différer de celles présentées en classe.

1.2 Crédits

Certains éléments du cours, dont l'implémentation des listes et arbres AVL, sont en partie inspirés du cours IFT339 – Structures de données, du professeur Jean Goulet du Département d'informatique de l'Université de Sherbrooke, que j'ai suivi en 1999. Les références citées à la fin du présent document ont également servi à la préparation de ces notes de cours.

1.3 Types abstraits de données

Les **types abstraits de données** sont des types de données que l'on peut utiliser de façon abstraite et transparente, et ce, sans se préoccuper des détails de leur implémentation (leur conception interne). Dans les langages de programmation orientée objet (POO), les types abstraits de données permettent de déclarer des variables afin de modéliser des **objets** précis.

Dans ce cours, il faut adopter deux visions face aux types abstraits de données : une vision d'utilisateur et

une vision de concepteur. Dans la vision d'**utilisateur** [1] , on doit développer le réflexe de traiter les objets de façon abstraite. Chaque type abstrait de données a une **interface publique** offrant un certain nombre de fonctionnalités (opérations). Lorsqu'on utilise un type abstrait de données, on doit surtout éviter de se préoccuper de sa représentation interne et de comment ses fonctionnalités sont implémentées. Il faut tout simplement connaître leur existence et savoir comment les utiliser.

Dans la vision de concepteur, on s'intéresse davantage à la représentation interne des types abstraits de données et à l'implémentation de leurs fonctionnalités. La représentation d'un type abstrait de données spécifie comment un objet de ce type doit être modélisé et stocké en mémoire. On doit **encapsuler** les détails d'implémentation à l'intérieur de l'objet de façon à les cacher au monde externe. Un utilisateur ne devrait pas pouvoir accéder à la représentation interne d'un objet.

Les langages de programmation offrent des types abstraits de données de base, comme des entiers, des caractères, des nombres à virgule flottante, etc. On peut construire de nouveaux types abstraits de données à l'aide d'autres types de plus bas niveau. Les sous-sections suivantes font quelques rappels et ont pour but de vous immerger dans une philosophie d'abstraction de données.

1.3.1 Le bit

En informatique numérique et en mathématiques discrètes, le **bit** est la plus petite unité d'information. Le domaine d'un bit, soit l'ensemble de ses valeurs possibles, est $\{0, 1\}$. Pris seul, un bit ne veut rien dire. Ce n'est qu'à partir du moment où on lui donne une signification précise que la valeur d'un bit devient une information utile. Par exemple, on peut représenter une valeur de vérité, communément appelée une variable booléenne, à l'aide d'un bit. Par convention, on représente la valeur **vrai** par un bit à 1 et **faux** par un bit à 0.

On peut voir le bit comme un type abstrait de données. Lorsqu'on utilise un bit, il ne faut pas se soucier de sa représentation interne. Il faut faire abstraction du support physique servant à stocker un bit, qu'il soit mécanique, électronique, optique, etc. Il faut simplement savoir qu'on peut manipuler des bits à l'aide de diverses instructions et opérateurs. Par exemple, on peut lire, copier ou écrire des bits en mémoire. On peut tester si un bit est à 0 ou 1 pour contrôler l'exécution. Enfin, les opérateurs logiques, la négation ($\neg$), le **AND** ($\wedge$), le **OR** ($\vee$), le **XOR** ($\bigoplus$), etc, font aussi partie de l'interface publique du bit.

Les bits peuvent être structurés afin de créer de nouveaux types de données plus complexes. Toutes les structures de données étudiées dans ce cours seront directement ou indirectement représentées à l'aide de **séquences de bits**. Ainsi, il est possible de **sérialiser** un objet en une séquence de bits, et de **désérialiser** une séquence de bits en un objet.

Bien que le bit soit la plus petite unité d'information, les ordinateurs allouent rarement la mémoire un bit à la fois. Pour des raisons techniques, la mémoire est plutôt allouée par blocs de bits, ou plus spécifiquement, par blocs d'octets. Un **octet** (*byte*) est un bloc de huit (8) bits. Toutefois, un octet ne devrait pas être considéré comme un type abstrait de données. Un octet devrait plutôt être considéré comme un bloc de bits ou une unité de quantité de données ou d'information. À noter que certains langages de programmation disposent quand même d'un type nommé `byte` afin de créer des entiers de 8 bits.

1. À ne pas confondre avec l'utilisateur final du logiciel. Ici, l'utilisateur est un programmeur qui utilise un type abstrait dans son programme.

1.3.2 Entiers

Les entiers sont généralement représentés à l'aide d'une séquence de 8, 16, 32 ou 64 bits. Une représentation inefficace serait de compter le nombre de bits à 1 ! Ainsi, la valeur signifiée des 16 bits illustrés à la Figure 1 serait le nombre 4. Évidemment, cette représentation est très inefficace puisqu'elle ne peut que représenter les nombres de 0 à 16. Elle est présentée ici uniquement pour vous sensibiliser au fait qu'il existe souvent plusieurs façons de représenter un type abstrait de données. La « meilleure façon » dépend généralement de l'application dans laquelle le type abstrait de données sera utilisé.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	1	0	0	0	0	1	0	1	0

FIGURE 1 – Séquence de 16 bits

Une représentation d'entier plus appropriée consiste à écrire un nombre en base 2 à l'aide de bits, de façon similaire à la façon dont on écrit des nombres en base 10. Ainsi, la valeur d'un entier naturel $n \in \mathbb{N}$ d'une séquence de n bits $\langle b_0, b_1, ..., b_{n-1} \rangle$ est calculée par l'équation 1. La position de chaque bit est alors importante. Sous cette représentation, la valeur signifiée par la Figure 1 serait de 2314.

$$n = \sum_{i=0}^{n-1} 2^i b_i \tag{1}$$

Plus tôt, il a été dit qu'il fallait faire abstraction de la représentation d'un type abstrait de données. Ici, nous pourrions être tentés de croire qu'il faut plutôt se soucier de la représentation des entiers puisqu'elle ne permet pas de représenter n'importe quel entier. En effet, avec 16 bits, on peut seulement représenter les entiers naturels de 0 à $2^{16} - 1$, soit de zéro à 65535. Une nuance importante s'impose. Ici, ce n'est pas la représentation elle-même qui devrait nous préoccuper. C'est plutôt ses **limites** et ses **capacités**. Évidemment, les limites dépendent de la représentation. Toutefois, on peut tout simplement se contenter de les documenter à l'intention de l'utilisateur (le programmeur qui utilise nos entiers).

La représentation précédente ne permet pas les entiers négatifs. Pour stocker un entier signé $z \in \mathbb{Z}$, il est possible de réserver le premier bit pour encoder le signe. La signification des autres bits reste inchangée. Il ne s'agit pas de la meilleure représentation possible pour des raisons techniques. La représentation à **complément à deux**, que vous devriez avoir vue en INF2170, offre plusieurs avantages.

Quand on utilise un objet de type entier, il faut surtout éviter d'accéder directement à sa représentation. Par exemple, pour vérifier si un entier signé `x` est négatif, il faut éviter de tester le bit du signe à l'aide d'un `ET` logique comme dans l'instruction `if(x & 0x80000000)`. Bien que cela puisse sembler plus rapide, cela constitue une très mauvaise pratique en programmation orientée objet. En effet, si on migre à une autre représentation d'entier, tester le bit de poids fort pourrait ne plus fonctionner. Il faut plutôt voir l'entier comme un type abstrait de données offrant des fonctionnalités, comme les opérations arithmétiques et les comparaisons. Ainsi, pour vérifier si un entier signé est négatif, il faut plutôt tout simplement le comparer avec zéro : `if(x<0)`. Est-ce moins efficace ? Ça dépend du compilateur et du processeur ! Généralement, après les optimisations, cela sera équivalent.

1.3.3 Caractères

Les caractères sont représentés par des nombres. Chaque symbole est associé à un nombre. La Figure 2 présente une table des caractères **ASCII** (*American Standard Code for Information Interchange*). Une norme ISO plus récente est **Unicode** [2]. L'étude de la représentation des caractères Unicode ne fait pas partie du cours. L'important est de retenir que les caractères et les chaînes de caractères sont des types abstraits de données.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

FIGURE 2 – Table ASCII (http://en.wikipedia.org/wiki/File:ASCII_Code_Chart.svg)

1.3.4 Nombres réels

Un nombre réel est un type abstrait de données. La représentation approximative à virgule flottante est couramment utilisée. Celle-ci est composée d'un bit de signe et de deux séquences de bits pour un exposant et une mantisse.

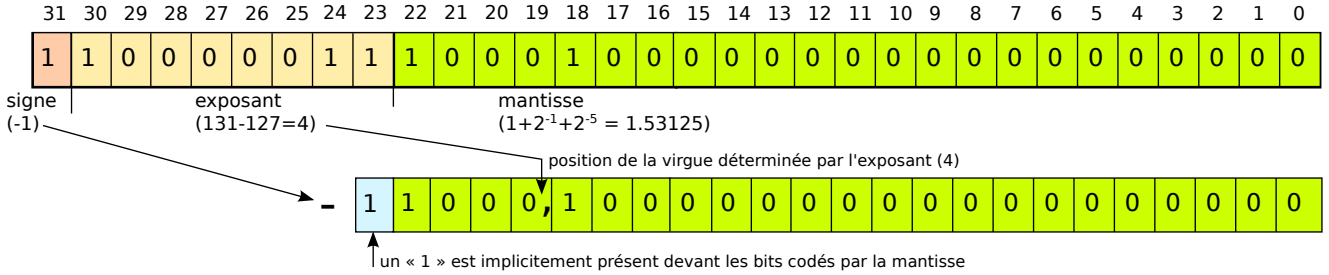


FIGURE 3 – Nombre à virgule flottante à 32 bits

Le standard IEEE 754 définit des normes pour l'encodage de nombres à virgule flottante sur 32, 64 ou 80 bits. La Figure 3 représente le nombre -24.5 sous la représentation IEEE 754 à 32 bits. Le premier bit encode le signe. Un bit à 0 indique un nombre positif alors qu'un bit à 1 indique un négatif. Les 8 bits suivants encodent l'exposant. Pour faciliter l'implémentation matériel, cet exposant n'est pas en représentation « complément à 2 », mais plutôt décalé de $2^{e-1}-1$ où e est le nombre de bits pour l'exposant. Ainsi, dans l'exemple, l'exposant est de 4. Enfin, les 23 derniers bits encodent la mantisse.

2. http://www.unicode.org/

Un « 1 » est implicitement ajouté au début la mantisse. L'équation 2 donne la valeur du nombre réel encodée.

$$valeur = signe \times 2^{exposant} \times mantisse \tag{2}$$

Certaines valeurs sont réservées à des cas spéciaux, comme zéro, les valeurs infinies et « NaN » [3].

1.4 Principes de base en génie logiciel

Bien que le cours INF3105 ne soit pas un cours de génie logiciel, mais bien un cours de structures de données et d'algorithmes, un bref rappel de certains principe de base en génie logiciel peut être tout à fait pertinent. Cela l'est d'autant plus considérant que INF3105 est un cours obligatoire dans le programme de Baccalauréat en informatique et génie logiciel.

Le génie logiciel est une branche de l'informatique qui s'intéresse à la qualité des logiciels. Les principales qualités qui nous intéresseront dans ce cours sont les suivantes.

- **Fonctionnement correct**. Un logiciel doit produire le résultat pour lequel il a été conçu. L'usager doit pouvoir se fier aux résultats sans avoir à les vérifier. C'est au(x) développeur(s) que revient la responsabilité de démontrer le bon fonctionnement du programme. Des preuves formelles ou tests empiriques sont des outils pour démontrer le bon fonctionnement d'un logiciel.
- **Robustesse**. Un bon logiciel doit être robuste aux situations qui sortent du cadre normal d'utilisation. Par exemple, si on fournit des entrées invalides à un logiciel, ce dernier doit au moins être capable d'interrompre le processus et de produire une erreur plutôt que de produire un résultat invalide ou de « planter » subitement. Un logiciel robuste fournit des messages d'erreur détaillés aidant l'usager à appliquer les correctifs.
- **Maintenabilité**. Le code source d'un logiciel doit être lisible et compréhensible. Un nouveau développeur devrait pouvoir comprendre le plus rapidement possible la structure du code d'un logiciel sur lequel il est appelé à travailler. De plus, une nouvelle fonctionnalité devrait être « facile » à réaliser, c'est-à-dire qu'il faudrait éviter de devoir repenser tout le code pour satisfaire de nouveaux besoins.
- **Efficacité**. Un bon logiciel doit utiliser efficacement les ressources spatiales et temporelles. Il existe souvent un compromis entre le temps d'exécution et la taille de mémoire requise. Ainsi, les besoins de l'application et les ressources disponibles doivent guider les choix d'implémentation.

Des bonnes pratiques de programmation orientée objet (POO) et l'usage de structures de données appropriées sont des moyens à considérer pour la production de logiciels de qualité. Ainsi, lors de l'élaboration de types abstraits de données, nous nous préoccuperons des caractéristiques de qualité énumérées ci-dessus.

Certaines des qualités d'un logiciel sont en opposition. Par exemple, il peut y avoir un compromis entre la maintenabilité et l'efficacité. Il revient au concepteur d'évaluer les choix possibles et d'aviser son client de leurs implications. Par la suite, le client pourra indiquer ses préférences. Ses préférences devront être considérées par le programmeur.

3. *Not A Number*, pas un nombre. Par exemple, le résultat de zéro divisé par zéro !

2 Langage C++

Cette section introduit brièvement le langage C++ que nous allons utiliser dans le cours. Le langage C++ est une extension au langage C permettant la **programmation orientée objet**. Il a été créé par Bjarne Stroustrup au *AT&T Bell Labs* aux États-Unis. Depuis, le langage C++ a été standardisé par l'Organisation Internationale de Normalisation (*International Organization for Standardization*).

Puisqu'il est indépendant de la plateforme, le langage C++ est dit « multi-plateforme ». Toutefois, contrairement à Java ayant une API (*application programming interface*) plus large, le langage C++ a une librairie standard plutôt petite. Ainsi, la portabilité effective des programmes C++ dépend beaucoup de la disponibilité des librairies utilisées.

2.1 Organisation d'un projet C++

2.1.1 Fichiers sources

Les programmes C++ sont principalement écrits dans deux types de **fichiers sources**.

- Les fichiers d'**entête** (*header*), ayant pour extension `.h` ou `.hpp`, contiennent généralement des **déclarations**.
- Les fichiers sources ayant pour extension `.cc`, `.cpp` ou `.c++`, contiennent généralement les **définitions** (l'implémentation). Ces fichiers peuvent aussi contenir des déclarations.

Le but de cette séparation est de faciliter l'organisation d'un projet et d'accélérer la compilation des programmes. En effet, suite à une modification durant le développement d'un programme, il est possible de recompiler uniquement les fichiers nécessaires.

Pour pouvoir utiliser les déclarations d'un fichier d'entête dans un fichier source, il faut utiliser la directive `#include`. Le code ci-dessous présente un exemple pour chacune des deux formes d'inclusion de fichier à l'aide de la directive `#include`. Le première spécifie entre crochets `<>` le nom du fichier à trouver dans un répertoire d'une bibliothèque. La deuxième spécifie entre guillemets `""` le nom du fichier dans le répertoire courant.

```
1 #include <iostream> // Inclusion d'une entete d'une bibliotheque
2 #include "point.h" // Inclusion d'un fichier dans le repertoire courant
```

Contrairement à d'autres langages comme Java, les noms de fichier n'ont pas besoin d'être alignés sur leur contenu. Un fichier `allo.h` peut déclarer une classe `bonjour`. Toutefois, il est évidemment recommandé de garder une bonne cohérence dans la façon d'organiser un projet.

2.1.2 Schéma traditionnel d'un compilateur C++

La Figure 4 présente le schéma traditionnel de la compilation d'un programme C++. La **construction** (compilation) d'un exécutable à partir des sources s'effectue en deux passes : la **compilation** et l'**édition des liens**. Les fichiers sources sont habituellement compilés un à un. Le **compilateur** génère un **fichier objet** pour chaque fichier source. Le compilateur C/C++ appelle à l'interne un **préprocesseur** pour traiter les **directives du préprocesseur** avant d'effectuer la génération du code objet. Un **éditeur de liens**

(*linker*) assemble les fichiers objets et les fichiers bibliothèques [4] afin de produire un fichier exécutable (ou une bibliothèque).

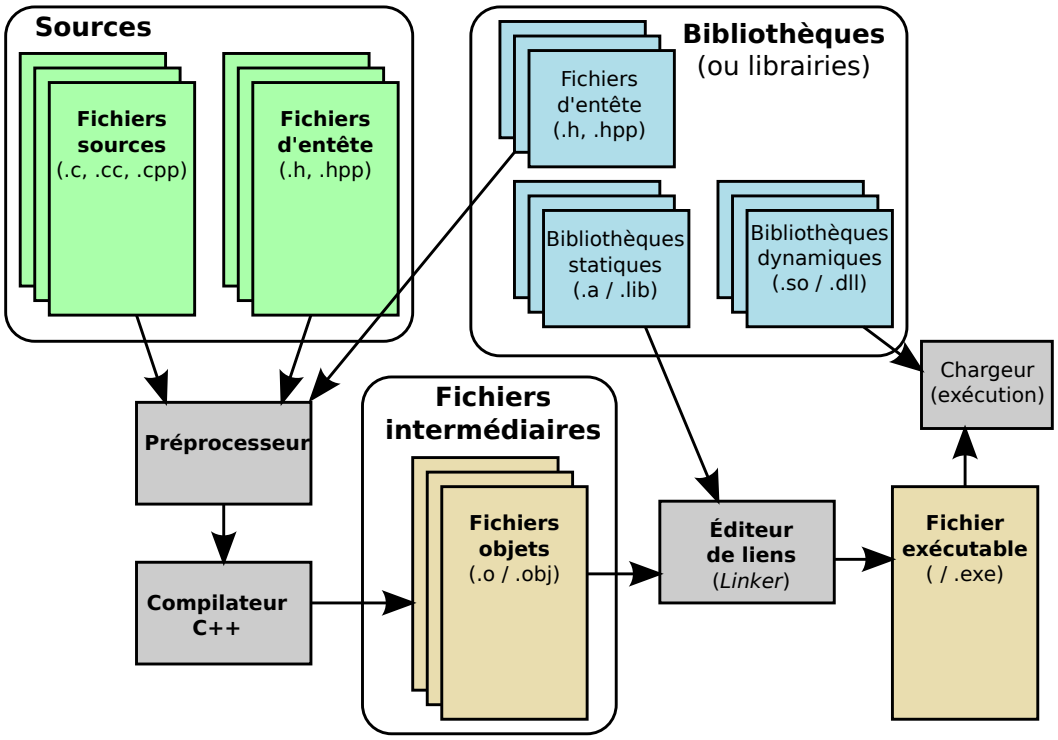


FIGURE 4 – Schéma traditionnel d'un compilateur C++

2.2 Compilateur GNU GCC

Le compilateur **fortement recommandé** dans ce cours est **GCC** [5]. Le compilateur GCC est déjà installé sur les serveurs Sun Solaris (`rayon1` et `rayon2`) et Linux (`chicoree`) du Département d'informatique de l'UQAM. De plus, les distributions Linux les plus populaires disposent de paquetages binaires. Par exemple, sous Ubuntu ou Debian, il suffit de lancer la commande « `sudo apt-get install g++` » pour installer le compilateur. Les compilateurs C et C++ de GCC peuvent être respectivement appelés à l'aide des commandes `gcc` et `g++`.

Syntaxe

```
$ g++ [options] fichier1 [fichier2 [...]]
```

Les fichiers sont des fichiers sources ou objets. Le tableau suivant présente les principales options.

Option	Description
`-o fichiersortie`	Spécifie le nom de fichier pour le code objet ou exécutable.
`-c`	Compilation seulement. N'appelle pas l'éditeur de liens.
`-g`	Génère les informations de débogage.
`-O`	Optimise le code généré. Équivalent à `-O1`.
`-On`	Remplacer `n` par `0`, `1`, `2` ou `3`. Spécifie le niveau d'optimisation.

4. Le terme librairie est parfois aussi employé, mais est considéré comme un anglicisme.
5. `http://gcc.gnu.org/`

Référez-vous au manuel officiel de GCC pour tous les détails. Tapez la commande `man g++` pour en apprendre davantage.

Compilation en un seul appel

Pour compiler le fichier **bienvenue.cpp** suivant :

```
1 #include <iostream>
2 int main(){
3     std::cout << "Bienvenue au cours INF3105 en C++ !" << std::endl;
4     return 0;
5 }
```

il suffit d'exécuter la commande :

```
$ g++ bienvenue.cpp
```

Par défaut, cela générera l'exécutable `a.out`. Pour l'exécuter, il suffit de lancer la commande suivante.

```
$ ./a.out
```

Compilation de deux fichiers en un seul appel

```
$ g++ fichier1.cpp fichier2.cpp
```

Compilation en plusieurs appels

```
$ g++ -c fichier1.cpp
$ g++ -c fichier2.cpp
$ g++ -o executable fichier1.o fichier2.o
```

2.3 Éléments de base

2.3.1 Commentaires

Il existe deux styles de commentaires en C++ :

- **Style C/C++** : `/* texte en commentaire (sur une ou plusieurs lignes) */`
- **Style C++** : `// commentaire sur une ligne (jusqu'à la fin de ligne)`

À noter que les commentaires en style C++ ne permettent pas l'imbrication de commentaires. Le commentaire « `/* /* Sous-commentaire */ Fin commentaire */` » termine après le premier `*/`. Ainsi, la portion « `Fin commentaire */` » n'est pas en commentaire et devrait générer une erreur. Par contre, le commentaire `/* int a = 2; // test */` est correct.

2.3.2 Identificateurs

En C++, les **identificateurs** sont des symboles servant à nommer les variables (objets), les classes, les fonctions, etc. Un identificateur est une chaîne de lettres, chiffres et de traits de soulignement (_). Un identificateur ne peut pas commencer par un chiffre et doit contenir au moins un caractère. L'expression régulière pour décrire des identificateurs est : `[a-zA-Z_][a-zA-Z_0-9]*`.

2.3.3 Types de base

La table 1 présente les principaux types de base en C++ avec leur capacité de représentation.

TABLE 1 – Types de base en C++

Type (mot clé)	Description	Taille (octets)	Capacité
`bool`	Booléen	1	$\{false, true\}$
`char`	Entier / caractère ASCII	1	$\{-128, \ldots, 127\}$
`unsigned char`	Entier / caractère ASCII	1	$\{0, \ldots, 255\}$
`unsigned short` `unsigned short int`	Entier naturel	2	$\{0, \ldots, 2^{16}-1\}$
`short` `short int`	Entier	2	$\{-2^{15}, \ldots, 2^{15}-1\}$
`unsigned int`	Entier naturel	4	$\{0, \ldots, 2^{32}-1\}$
`int`	Entier	4	$\{-2^{31}, \ldots, 2^{31}-1\}$
`unsigned long`	Entier naturel	8	$\{0, \ldots, 2^{64}-1\}$
`long`	Entier	8	$\{-2^{63}, \ldots, 2^{63}-1\}$
`float`	Nombre réel	4	$\pm 3.4 \times 10^{\pm 38}$ (7 chiffres)
`double`	Nombre réel	8	$\pm 1.7 \times 10^{\pm 308}$ (15 chiffres)
`long double`	Nombre réel	8	$\pm 1.7 \times 10^{\pm 308}$ (15 chiffres)

2.3.4 Déclaration de variables (objets)

Une variable est une **instance** d'un type de données. En C++, les variables sont considérées comme des **objets**. Chaque variable est nommée à l'aide d'un identificateur. L'identificateur doit être unique dans sa portée. La déclaration de variables (objets) se fait avec la syntaxe [6] suivante :

```
<DECLARATION> ::= <TYPE> (<IDENTIFICATEUR>((...) | = <EXPRESSION>)?)+
```

Voici un exemple de code pour déclarer des variables.

```
1 // Declaration d'un entier (sans initialisation)
2 int a;
3 /* Declaration de nombres reels (sans initialisation) */
4 float f1, f2, f3;
5 /* Declaration de nombres avec initialisation explicite*/
6 int n1(12), n2=20;
```

2.3.5 Initialisation des variables

Le langage C++ adopte une philosophie orientée objet (OO) pour l'initialisation des variables. Les variables (objets) sont toujours initialisées à l'aide d'un **constructeur** [7]. Quand on demande l'initialisation d'une variable, le compilateur se charge de générer le code pour appeler le bon constructeur. Quand aucune initialisation n'est explicitement spécifiée, c'est le **constructeur sans argument** qui est appelé. Toutefois, les constructeurs sans argument implicite des types de base en C++ ne font rien ! Ainsi, cela est équivalent à dire que **les variables de types de base ne sont pas initialisées par défaut**, comme pour le langage C. Lorsqu'une case mémoire n'est pas initialisée, elle garde le contenu précédent. Une utilisation inadéquate de variables non initialisées peut provoquer des comportements **pseudo non déterministes**

6. Les règles de la grammaire de C++ sont données en notation EBNF (*Extended Backus-Naur Form*).
7. Référez-vous à la Section 2.14 pour en savoir plus sur ce qu'est un constructeur.

(imprévisibles).

Dans la réalité, les constructeurs sans argument des types de base n'existent pas, car ils ne font absolument rien. Ne cherchez pas le constructeur `int::int()`, vous le trouverez pas ! Ainsi, si on regarde le code assembleur généré pour le bout de code « `int a` », on ne trouvera pas d'appel au constructeur de `int::int()`. Toutefois, dans une philosophie orientée objet, on peut supposer l'existence d'un tel constructeur sans argument où ce dernier est sans effet. Cela nous permet d'avoir une vision unifiée au niveau de l'initialisation des variables.

2.4 Énoncés et expressions

Comme dans la plupart des langages de programmation, le corps d'une fonction en C++ est constitué d'énoncés (*statements*). Sommairement, un énoncé peut être :

- une déclaration de variable(s) ;
- une expression d'affectation ;
- une expression ;
- une instruction de contrôle ;
- un bloc d'énoncés entre accolades `{}`.

À l'exception d'un bloc `{}`, un énoncé se termine toujours par un point-virgule (`;`).

2.4.1 Affectation

L'affectation se fait à l'aide de l'opérateur égal =.

```
1 // Declaration
2 int a;
3 // Affectation
4 a = 2 + 10;
```

2.4.2 Expression

En C++, une expression peut être :

- un identificateur (variable) ou un nombre ;
- une expression arithmétique ou logique ;
- un appel de fonction (voir Section 2.10) ;
- une autre expression entre parenthèses ( ) ;
- un opérateur d'affectation (=, +=, etc.) ;
- etc.

Exemple d'expressions :

```
1 4+5*6-8;
2 (4+5)*(6-8);
3 a * 2 + 10;
```

En réalité, une assignation est une expression qui retourne la valeur affectée.

```
a = b = c = d;
// est l'equivalent de :
c = d; b = c; a = b;
```

Il existe aussi des opérateurs combinant l'affectation et un autre opérateur.

```
a++; // a = a + 1;
a+=10; // a = a + 10;
a*=2; // a = a * 2;
a/=2; // a = a / 2;
```

Les opérateurs `++` et `--` sont en deux versions (pré/post).

```
b = a++; // b=a; a=a+1;  // post-increment
b = ++a; // a=a+1; b=a;  // pre-increment
b = a--; // b=a; a=a-1;  // post-decrement
b = --a; // a=a-1; b=a;  // pre-decrement
```

Le point d'interrogation permet la spécification d'une expression conditionnelle ayant pour syntaxe : « `<condition> ? <exp_si_vrai> : <exp_si_faux>` ». Voici un exemple.

```
a=1; b=2;
x = a<b ? 2 : 3; // if(a<b) x=2 else x=3;
```

La table 2 résume les principaux opérateurs :

TABLE 2 – Opérateurs en C++

Opérateurs arithmétiques	
`+, -, *, /, %`	Addition, Soustraction, Multiplication, Division et Modulo
Opérateurs arithmétiques avec affectation	
`++, --, +=, -=, *=, %=`	Pré-/Post-incrément, Pré-Post-décrément, etc.
Opérateurs logiques	
`&&,||,!`	Et, Ou, Négation
Opérateurs bit à bit	
`&,|`	ET logique, Ou logique

2.4.3 Instructions de contrôle

Les instructions de contrôle permettent de contrôler le fil d'exécution. Chaque instruction est spécifiée à l'aide d'un mot clé.

Le `if` a la syntaxe suivante :

```
<ENONCE_IF> ::= "if" <EXPRESSION> <ENONCE>
              |  "if" <EXPRESSION> <ENONCE> "else" <ENONCE>
```

Le `while` a la syntaxe suivante :

```
<ENONCE_WHILE> ::= "while" "(" <EXPRESSION> ")" <ENONCE>
```

Le `do while` a la syntaxe suivante :

```
<ENONCE_WHILE> ::= "do" <ENONCE> "while" "(" <EXPRESSION> ")" ";"
```

Le `for` a la syntaxe suivante :

```
<ENONCE_FOR> ::= "for" "(" <ENONCE> ";" <EXPRESSION> ";" <EXPRESSION> ")"
                    <ENONCE>
```

Le code suivant montre un exemple d'usage de `for` :

```
1  int a;
2  for(int i=0;i<10;i++)
3      a += 4;
```

Le code suivant montre un exemple d'usage de `switch` :

```
1  int a = 4;
2  switch(a){
3      case 0:
4          // ...
5          break;
6      case 1:
7          // ...
8          break;
9      case 4:
10         // ...
11         break;
12     default:
13         // ...
14 }
```

2.5 Entrées/sorties en C++

En C++, les entrées et sorties sont encapsulées dans des **flux** (*streams*). La bibliothèque standard de C++ (*C++ Standard Library*) fournit des flux pour la console et les fichiers. Les flux d'écriture et de lecture héritent respectivement des classes `std::ostream` et `std::istream`. Les opérations de lecture et d'écriture se font respectivement à l'aide des opérateurs << et >>.

2.5.1 Entrée standard et sortie standard

Pour avoir accès aux flux standards de la bibliothèque standard de C++, il faut inclure le fichier d'entête `iostream`. Ce dernier définit les trois flux suivant :

- `std::cin` : flux d'entrée depuis l'entrée standard (*stdin*) ;
- `std::cout` : flux de sortie vers la sortie standard (*stdout*) ;
- `std::cerr` : flux de sortie vers la sortie d'erreurs (*stderr*).

Le code suivant est un exemple qui lit deux nombres et affiche leur somme à la console.

```
#include <iostream>
int main(){
    std::cout << "Entrez deux nombres :" << std::endl;
    int a, b;
    std::cin >> a >> b;
    std::cout << "La somme est : " << (a+b) << std::endl;
}
```

2.5.2 Entrée et sortie dans les fichiers

Pour avoir accès aux flux de la bibliothèque standard de C++, il faut inclure le fichier d'entête `fstream`.

```
#include <fstream>
int main(){
    std::ifstream in("entree.txt");
    int a, b, c;
    // lecture de trois entiers
    in >> a >> b >> c;
    std::ofstream out("sortie.txt");
    // ecriture de trois entiers
    out << a << '\t' << b << '\t' << c << std::endl;
}
```

2.5.3 Manipulateurs

Référez-vous à la documentation : `http://en.cppreference.com/w/cpp/io/manip`.

2.6 Tableaux

Un tableau est une suite d'éléments contigus et du même type en mémoire. Pour accéder à un élément précis, on utilise un **indice**. Le premier élément a l'indice zéro (0). Pour déclarer un tableau, on donne le type de ses éléments, un identificateur et sa taille entre crochets `[]`. Les éléments sont initialisés individuellement de la même manière que les variables (voir Section 2.3.5). Le code suivant donne un exemple de tableau.

```
int tableau[100];
```

Il est possible d'initialiser les éléments explicitement tel qu'illustré ci-dessous. À noter que la taille du tableau peut être supérieure au nombre d'éléments initialisés. Si aucune taille n'est spécifiée, le tableau prendra la taille du nombre d'éléments spécifiés en initialisation.

```
int tableau1[5] = {0, 5, 10, 15, 20};
int tableau2[10] = {0, 5, 10, 15, 20};
int tableau3[] = {0, 5, 10, 15, 20};
```

2.6.1 Limites

La taille des tableaux est fixe et ne peut pas être modifiée. De plus, les indices des tableaux ne sont jamais vérifiés. En fait, la variable du tableau est un pointeur (une adresse mémoire, voir Section 2.8), vers le premier élément du tableau. Dans l'exemple suivant, `tab1[10]` peut référer à la même case mémoire que `tab2[0]`.

```
1 int main() {
2     int tab1[10], tab2[10]
3     tab2[0] = 0;
4     tab1[10] = 10;
5     std::cout << tab2[0] << std::endl;
6 }
```

2.6.2 Chaînes de caractères

En C/C++, une chaîne de caractères est un tableau de `char` terminé par un caractère nul (`\0`).

```
1 char[] chaine = "allo";
```

2.6.3 Tableaux multi-dimension

Un tableau à deux dimensions est tout simplement un tableau de tableaux. L'exemple suivant montre comment déclarer un tableau à 2 dimensions.

```
1 int tableau1[2][5];
2 int tableau2[2][5] = {{1,2,3,4,5}, {10,20,30,40,50}};
```

2.7 Structures

Une **structure** est une composition d'éléments (objets). Le mécanisme de structure a été introduit dans le langage C et est précurseur du mécanisme de classe en C++. Avant d'utiliser une structure, il faut déclarer sa composition à l'aide du mot clé `struct`. Chaque élément (objet) a un type et un nom ; Pour accéder aux éléments d'une structure, on utilise le point (`.`). Voici un exemple de déclaration et d'utilisation de structure.

```
1 struct Date{
2     int annee;
3     int mois;
4     int jour;
5 };
6 Date debut;
7 debut.annee = 2012;
8 debut.mois = 1;
9 debut.jour = 1;
```

À noter qu'il n'est pas requis de nommer la structure lorsqu'on déclare un objet directement comme dans l'exemple suivant.

```
1 struct{
2     int annee;
3     int mois;
4     int jour;
5 }debut;
6 debut.annee = 2012;
7 debut.mois = 1;
8 debut.jour = 1;
```

En C++, les structures offrent plus de fonctionnalités que les structures en C. Dans les faits, une `struct` offre les mêmes possibilités qu'une **classe** (Section 2.14).

2.8 Pointeurs

Un **pointeur** est un objet (variable) qui prend pour valeur une adresse mémoire. Un pointeur permet de **pointer** un objet précis. Les pointeurs sont déclarés au moyen du symbole étoile `*` placé à côté d'une variable. En C++, pour signifier qu'un pointeur pointe vers « rien », on lui affecte la valeur `NULL` (zéro). D'où l'expression **pointeur nul**. Lorsqu'un pointeur n'est pas initialisé, il a le même comportement que les types de base, c'est-à-dire qu'il garde la valeur en mémoire courante. Voici un exemple de déclaration de pointeur.

```
1 int *pointeur = NULL;
```

Les adresses mémoire peuvent être obtenues de deux façons :

— en utilisant l'opérateur `&` pour récupérer l'adresse d'une variable ou objet ;
— ou par allocation de mémoire dynamique (voir Section 2.13.2).

Dans l'exemple suivant, on définit le pointeur `ptr_n` [8] qui pointe vers la variable `n`

```
1 int n = 3;
2 int* ptr_n = &n;
3 int* tableau = new int[100];
```

Dans l'exemple ci-dessus, le symbole `*` a été volontairement collé sur le type. C'est une pratique fréquente en C/C++. Toutefois, il faut être prudent quand on veut définir plusieurs pointeurs, où il faut mettre l'étoile devant chaque variable. Par exemple, dans le code suivant, l'objet `o1` n'est pas un pointeur mais bien un objet de type `int`. La portée de l'étoile `*` est uniquement sur `p1`.

```
1 //Declare le pointeur p1 et l'objet o1
2 int* p1, o1;
3 //Declare les pointeurs p2, p3, p4 et l'objet o2
4 int *p2, *p3, o2, *p4;
```

8. Certaines normes de programmation C/C++ recommandent l'utilisation du préfixe `ptr_` pour nommer les pointeurs. À l'exception de cet exemple, cette norme ne sera pas utilisée dans les présentes notes de cours.

Pour utiliser la valeur pointée ou l'objet pointé par un pointeur, il faut déférencer le pointeur à l'aide de l'**opérateur d'indirection** `*`. Le code suivant affichera la valeur pointée par `pointeur`, c'est-à-dire la valeur de `n`, à laquelle on a assigné 5.

```
1 int n=0;
2 int *pointeur = &n;
3 *pointeur = 5; // effet : n=5
4 std::cout << "n=" << *pointeur << std::endl;
```

Pour accéder aux membres d'une structure ou d'une classe à partir d'un pointeur, on utilise une flèche formée d'un tiret et du symbole plus grand (`->`).

```
1 struct Date{
2     int annee, mois, jour;
3 };
4 Date date1;
5 Date* date2 = &date1;
6 date2->annee = 2012;
7 date2->mois = 1;
8 date2->jour = 1;
```

2.8.1 Arithmétique des pointeurs

Les langages C/C++ offrent beaucoup de souplesse à la façon dont on accède à la mémoire. Par exemple, C/C++ permettent d'effectuer de **l'arithmétique sur les pointeurs**, c'est-à-dire de manipuler (calculer) des adresses mémoires. Certains langages comme Java ne permettent pas ce genre de manipulations puisque ces opérations sont potentiellement dangereuses.

Dans le code suivant, on utilise les opérateurs `+` et `++` pour manipuler des adresses. Après un appel de `i++`, le pointeur `i` pointe sur le prochain élément.

```
1 int tableau[1000];
2 int somme = 0;
3 int* fin = tableau+1000; // pointe sur l'element suivant le dernier element
4 for(int* i=tableau;i<fin;i++)
5     somme += *i;
```

Le code ci-dessus et le suivant sont équivalents. Dans le code ci-dessus, le pointeur `i` est utilisé pour se déplacer d'adresse en adresse. Dans le code ci-dessous (plus simple à lire), on utilise plutôt un entier pour se déplacer d'indice en indice dans le tableau. L'expression `tableau[i]` est implicitement traduite en `*(tableau + i)`, soit l'adresse de `tableau` plus `i * sizeof(int)`. Ce code nécessite plus d'opérations puisqu'il faut recalculer l'adresse à chaque fois en partant de l'adresse initiale de `tableau`.

```
1 int tableau[1000];
2 int somme = 0;
3 for(int i=0;i<1000;i++)
4     somme += tableau[i]; // somme += *(tableau+i)
```

2.9 Références

Une **référence** peut être vue comme un **alias** pour une autre variable. Une **référence** peut aussi être vue comme une sorte de pointeur encapsulé qui peut être utilisé de la même façon qu'un objet. Les références sont apparues dans le C++ et ne sont pas disponibles en C. Contrairement aux pointeurs, une référence doit toujours être initialisée. De plus, on ne peut jamais changer l'adresse d'une référence. Ainsi, une référence réfère toujours au même objet. Une référence se déclare à l'aide du symbole perluète `&`. L'exemple suivant affichera `3`.

```
1 int n = 2;
2 int& ref_n = n; // ref_n est un alias pour n
3 n = 3;
4 std::cout << "ref_n=" << ref_n << std::endl;
```

2.10 Fonctions

Le langage C++ permet la définition de sous-routines appelées **fonctions**. Les fonctions sont très utiles pour morceler un programme en plusieurs parties. Chaque fonction est nommée par un identificateur. Le nom et les types de données des paramètres d'une fonction constituent sa **signature**. Une fonction a également un **type de retour**. Pour spécifier qu'une fonction ne retourne aucune valeur, on utilise le type `void`. À noter que le type de retour ne fait pas partie de la signature. Chaque signature doit être unique. Deux fonctions ne peuvent pas avoir la même signature. Toutefois, il est possible d'avoir plusieurs fonctions ayant le même nom, mais avec une liste de paramètres différents.

Le code suivant présente un exemple de définition et d'utilisation d'une fonction.

```
1 int somme(int a, int b) {
2     return a + b;
3 }
```

Pour appeler une fonction, il suffit de donner son nom et ses paramètres entre parenthèses (). Voici un exemple.

```
1 int s = somme(3, 5);
```

Les fonctions doivent être déclarées avant leur utilisation. Il est possible de **déclarer** le **prototype** d'une fonction avant son utilisation, et de **définir** son corps plus tard. Voici un exemple.

```
1  // Prototype de fonction (declaration)
2  int somme(int, int);
3  int main(){
4      int x = somme(3, 5); // appel de la fonction
5      return x;
6  }
7  // Definition de la fonction
8  int somme(int a, int b) {
9      return a + b;
10 }
```

2.10.1 Passage de paramètres

Le langage C++ permet trois types de passage de paramètres :

- par valeur ;
- par pointeur (cas particulier de par valeur où la valeur passée est une adresse pointant vers un objet) ;
- par référence sur un objet.

Voici un exemple d'utilisation des différents types de passage de paramètres. Essayez-le !

```
1  void test(int a, int* b, int* c, int& d, int*& e){
2      a=11; // effet local
3      b++; // change l'adresse locale de b
4      *c=13; // change la valeur pointee par c
5      d=14; //  change la valeur referee par d
6      e=c; // change la valeur du pointeur (adresse) pour celle de c.
7  }
8  int main(){
9      int v1=1, v2=2, v3=3, v4=4, *p5=&v1;
10     test(v1, &v2, &v3, v4, p5);
11     cout<<v1<<'\t'<<v2<<'\t'<<v3<<'\t'<<v4<<'\t'<<*p5<<'\t'<<endl;
12     // affiche : 1  2  13  14 13
13     return 0;
14 }
```

2.10.2 Le point d'entrée : la fonction *main*

Tout comme en C, le point d'entrée d'un programme C++ est une fonction nommée **`main`**. La fonction **`main`** peut avoir l'une des trois signatures suivantes.

```
1  int main(){/* aucun parametre */}
2  int main(int argc, const char** argv){
3    // argc : contient le nombre d'arguments passes
4    // argv : contient les arguments
5  }
6  int main(int argc, const char** argv, int envc, const char** envs){
7    // envc : le nombre de variables d'environnement
8    // envs : contient les variables d'environnement
9  }
```

La valeur retournée par la fonction `main` est affectée au code de retour du processus résultant de l'exécution du programme. Le processus parent peut obtenir cette valeur. Par convention, un code de retour à zéro (0) signifie que le programme s'est terminé avec succès.

2.11 Espaces de nom (*Namespaces*)

Pour éviter des collisions de noms d'identificateur (pour nommer les variables globales, les fonctions et les classes), un mécanisme d'espace de nommage (*namespace*) peut être utilisé. Ce mécanisme est similaire au mécanisme de `package` en Java, mais avec plus de liberté.

Les variables globales, des classes et des fonctions fournies dans la bibliothèque standard C++ sont définies dans le *namespace* `std`. Pour utiliser des objets dans un *namespace*, il faut soit utiliser l'opérateur de porté `::` ou utiliser la mot clé `using` au préalable. Voici deux exemples.

```
1 #include <iostream>
2 using namespace std;
3 int main(){
4     cout << "Exemple 1" << endl;
5 }
```

```
1 #include <iostream>
2 int main(){
3     std::cout << "Exemple 2" << std::endl;
4 }
```

Il est possible de spécifier le *namespace* courant à l'aide du mot clé `namespace`. L'exemple suivant montre comment définir et utiliser un *namespace*.

```
1 namespace inf3105{
2     void allo(){
3     }
4 }
5 int main(){
6     inf3105::allo();
7 }
```

2.12 Les énumérations

Le type énumération permet de définir un ensemble discret dont les valeurs peuvent être utilisées comme des constantes.

```
1 // Definit un ensemble de constantes
2 enum Jour {dimanche, lundi, mardi, mercredi, jeudi, vendredi, samedi};
3 // Instancie une variable jour de type Jour
4 Jour jour = lundi;
```

Les types énumération sont représentés par des entiers. Par défaut, les valeurs sont attribuées aux constantes à partir de zéro. Dans le code précédent, `dimanche` se voit attribuer la valeur 0, `lundi` 1, etc. L'exemple suivant montre comment définir des valeurs personnalisées pour chaque élément.

```
1 enum Lettres {A=1, B=0, C=2, D=4, E, F};
2 // E et F auront respectivement les valeurs 5 et 6
```

2.13 Gestion de la mémoire

La mémoire de travail d'une machine est constituée d'un vecteur linéaire de cases mémoires. Chaque case mémoire est accessible à l'aide d'une adresse mémoire qui désigne sa position dans le vecteur. La mémoire de travail permet un **accès direct** (*direct access*), c'est-à-dire qu'il est possible de lire depuis ou d'écrire dans n'importe quelle case mémoire, et ce, sans avoir à lire les cases précédentes ou suivantes. Certains auteurs utilisent le terme **accès aléatoire** (*random access*) pour désigner un accès direct.

Dans les systèmes d'exploitation modernes, chaque processus dispose de son propre espace d'adressage. Il s'agit de l'espace mémoire utilisateur (*user space memory*) dans une mémoire virtuelle. L'espace mémoire d'un processus est généralement divisé en plusieurs segments. Il y a entre autre une **pile d'exécution** et un **tas** (*heap*). En fait, il y a une pile d'exécution par fil d'exécution (*thread*). Comme montrée par la Figure 5, le tas peut être au début et la pile à la fin. À noter que la pile d'exécution est généralement de taille fixe alors que le tas est de taille variable.

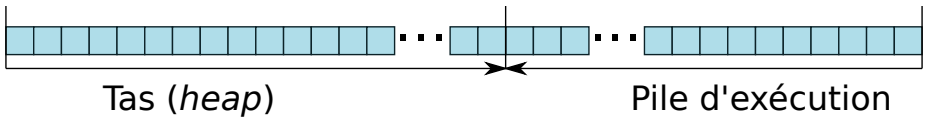


FIGURE 5 – La mémoire sous forme d'un vecteur

2.13.1 Allocation automatique (sur la pile d'exécution)

La pile d'exécution (*call stack*) est un espace mémoire essentiel pour capturer l'état des appels de fonction. À chaque appel de fonction, on y empile les paramètres, les variables locales, les adresses de retour, la valeur de retour, la valeur des registres, etc. Lorsqu'une fonction retourne, on dépile tous les éléments afin de retrouver le contexte d'exécution originale de l'endroit d'où la fonction avait été appelée.

En C/C++, lorsqu'une variable locale est déclarée, elle est automatiquement allouée sur la pile d'exécution. Le compilateur réserve l'espace mémoire à l'aide d'une instruction d'incrémentation (ou de décrémentation) du **compteur de pile** (*stack pointer*). Lorsqu'on utilise une variable, on réfère à la mémoire sur la pile à l'aide d'un déplacement (*offset*) relatif au pointeur de pile (*stack pointer*).

```
1  short int f1(){
2      short int d = 4;   short int e;
3      return d;
4  }
5  short int f2(short int a){
6      short int c = a + f1();
7      a += 2;
8      return c;
9  }
10 int main(){
11     short int x=3;   short int y=5;
12     y = f2(x);  f1();
13     return 0;
14 }
```

La Figure 6 montre un exemple simplifié de quelques états de la pile pendant l'exécution du programme C++ ci-dessus. Seul l'essentiel, soit les variables locales, y est présenté. Chaque case mémoire contient un octet dont la valeur est notée en hexadécimal. Les points d'interrogation indiquent les endroits où le contenu de la mémoire est indéterminé [9]. La valeur dans ces cases dépend des valeurs précédentes.

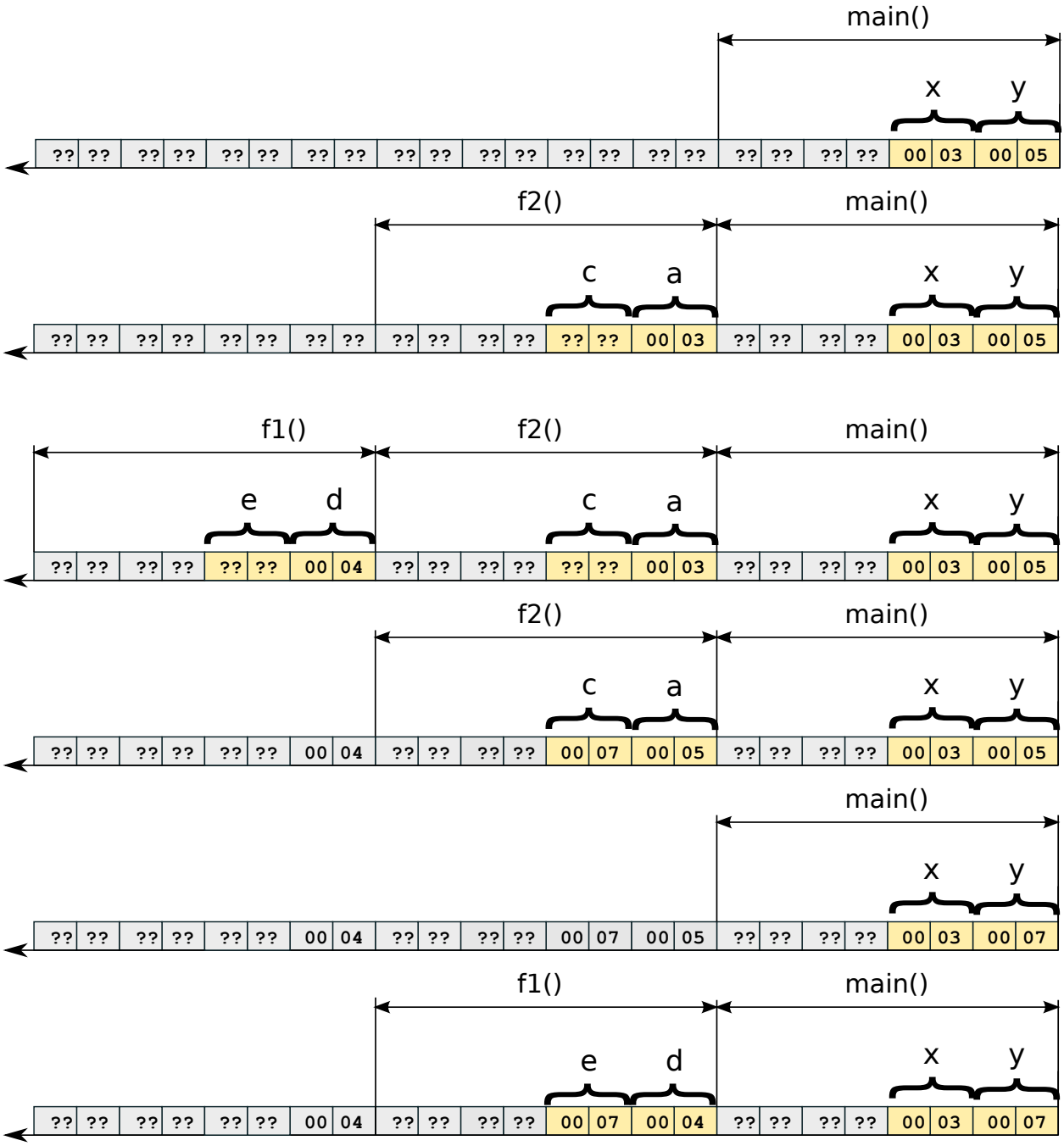


FIGURE 6 – Séquence d'états de la pile d'exécution du programme ci-dessus

9. Notez que les systèmes d'exploitation modernes peuvent réinitialiser l'espace mémoire usager à une suite de zéros.

2.13.2 Allocation dynamique sur le tas (*heap*)

L'allocation automatique n'est pas toujours appropriée. Par exemple, lorsqu'on ne connaît pas à l'avance la taille idéale d'un tableau, il est préférable de l'allouer dynamiquement au moment approprié. De plus, comme la taille de la pile d'exécution est généralement fixe, il n'est pas recommandé d'allouer de gros objets sur la pile. Cela risque de provoquer un débordement de pile (*stack overflow*).

En C, la mémoire est allouée à l'aide de la fonction `malloc()` et libérée par `free()`. En C++, on utilise les opérateurs **`new`** et **`delete`**. L'opérateur `new` s'occupe de calculer l'espace mémoire requis, d'allouer la mémoire requise et d'appeler les constructeur requis. L'opérateur `delete` s'occupe d'appeler le destructeur requis et de libérer la mémoire utilisée.

Dans le présent cours, nous ferons l'hypothèse que l'allocation (`new` en C++) et que la libération (`delete` en C++) de la mémoire se font en temps constant, c'est-à-dire $O(1)$.

2.13.3 Exemples d'allocations automatiques et dynamiques

Le code suivant montre des exemples d'allocations automatiques et dynamiques. La ligne 6 montre comment créer un tableau dynamique. Ce tableau est libéré à la ligne 13. Pour les tableaux, les `[]` devraient presque toujours être spécifiés. Cela indique qu'il faut appeler le destructeur sur chaque élément du tableau. La ligne 8 alloue un objet de type `A`. Ce dernier est détruit et libéré à la ligne 14. Enfin, la ligne 10 crée un tableau dynamique de 3 objets de type `A`.

```
1  struct A{
2      short int v1, v2;
3  };
4  int main(){
5      char c = 0;
6      short int *tab = new short int[6]{0x00,0x01,0xc712,0x03,0x14,0x3b05};
7      A a1; a1.v1=0x00b5; a1.v2=0x0073;
8      A* a2 = new A();
9      a2->v1=0; a2->v2=2;
10     A* a3 = new A[3];
11     a3[1].v1=0x7fff; a3[1].v2=0x0020;
12     // ...
13     delete[] tab;
14     delete a2;
15     delete[] a3;
16     return 0;
17 }
```

La Figure 7 montre un exemple hyper simplifié de l'état de la mémoire lorsque l'exécution du programme ci-dessus est rendue à la ligne 12. Les adresses (taille des pointeurs) ont 8 bits (1 octet). Les tailles du tas et de la pile sont respectivement de 48 octets et 16 octets. La case mémoire à la case 0 est généralement non utilisée afin de ne pas être confondue avec un pointeur nul (`NULL`). Dans cet exemple, les entiers à 16 bits (`short int`) sont représentés en ***big endian***. Dans la suite des notes, nous utiliserons une représentation abstraite et simplifiée de la mémoire pouvant ressembler à l'exemple de la Figure 8. Des flèches sont employées pour représenter les pointeurs.

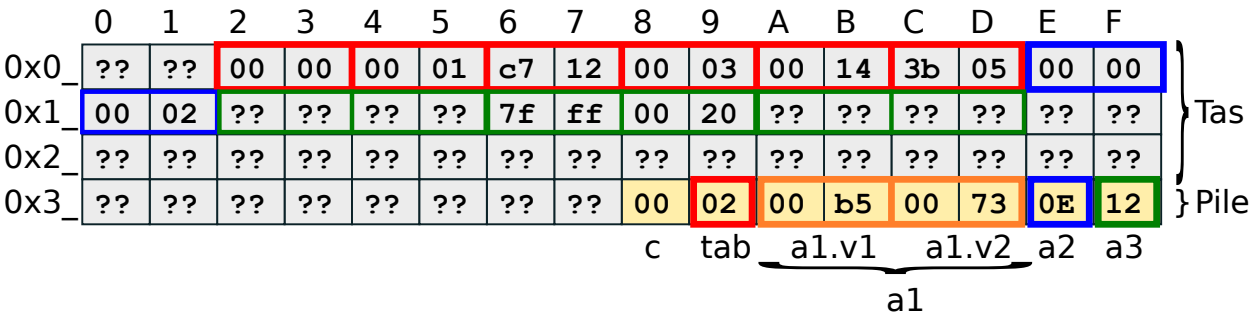


FIGURE 7 – Exemple d'allocations de mémoire avec un tas de 48 octets et une pile de 16 octets

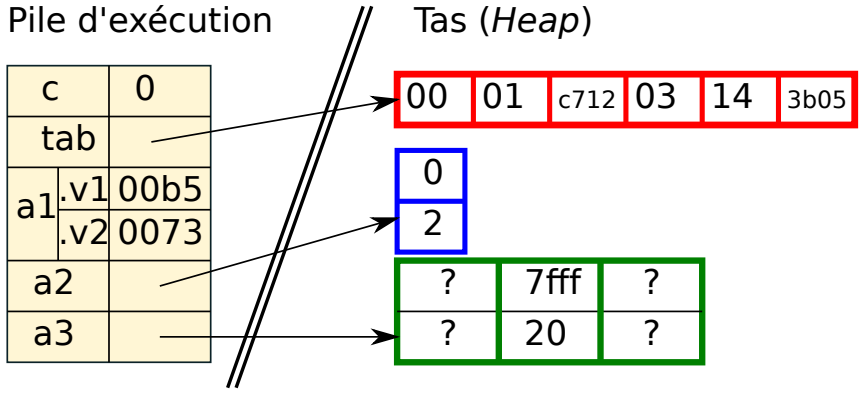


FIGURE 8 – Exemple de représentation abstraite d'un espace mémoire

2.14 Programmation orientée objet en C++

Le C++ est un langage de programmation orientée objet. La programmation orientée objet permet de définir de nouveaux objets à l'aide d'autres objets. C'est à l'aide du mot clé `class` qu'on peut déclarer de nouvelles **classes** d'objets, c'est-à-dire de nouveaux types d'objets, ou plus spécifiquement de nouveaux types abstraits de données. Une classe définit un ensemble de membres. Un membre peut être une variable définissant un objet ou une opération (constructeur, destructeur, fonction et opérateur). À l'exception des **variables statiques** définies à l'aide du mot clé `static`, les variables sont aussi appelées des **variables d'instance** puisqu'il y a exactement une instance pour chaque objet créé (instancié).

Les objets (variables) et les opérations d'une classe ont un niveau d'accès. Le niveau d'accès est spécifié à l'aide d'un mot clé de **modification d'accès** suivit d'un deux-points (`:`). Les niveaux d'accès possibles sont `public`, `protected` (protégé) et `private` (privé). Un membre public est accessible depuis l'extérieur. Un membre privé est uniquement accessible depuis l'intérieur. Un membre protégé est accessible de l'intérieur et depuis les classes héritées. Par défaut, c'est-à-dire si aucun modificateur d'accès n'est spécifié, les membres de la classes sont `private`. Lorsqu'un modificateur d'accès est spécifié, il s'applique à tous les membres suivants et jusqu'au prochain modificateur.

Le code suivant montre un exemple de classe `Point` définie par deux variables de type `double` définissant les coordonnées X et Y.

```
1 class Point {
2   public:
3     double distance(const Point& p) const;
4   private:
5     double x, y;
6 };
```

L'**encapsulation** consiste à cacher les détails à l'utilisateur. Par exemple, dans le code ci-dessus, les variables `x` et `y` sont privées. Donc, elles ne peuvent être accédées qu'à partir de la classe `Point`.

2.14.1 Constructeurs et destructeurs

Un **constructeur** ressemble à une fonction. Un constructeur porte le nom de la classe et peut avoir zéro, un ou plusieurs arguments. Comme son nom l'indique, le rôle d'un constructeur est de construire (instancier) un objet. Un constructeur effectue dans l'ordre :

- appelle le constructeur de la ou des classes héritées ;
- appelle le constructeur de chaque variable d'instance ;
- exécute le code dans le corps du constructeur.

L'exemple ci-dessous montre un exemple de la déclaration et de la définition de deux constructeurs dans la classe `Point`.

```
1 class Point {
2   public:
3     Point(); // constructeur sans argument
4     Point(double x, double y);
5     //...
6 };
```

```
1 Point::Point(){
2     x = y = 0.0;
3 }
4 Point::Point(double x_, double y_)
5  : x(x_), y(y_) // le deux-points (":") est pour l'initialisation
6 {
7 }
```

Dans l'exemple ci-dessus, les deux-points (« : ») marquent le début de l'initialisation des membres de la classe. L'initialisation offre la possibilité au programmeur de choisir quel constructeur doit être utilisé pour construire chaque membre de la classe. L'initialisation est facultative. Pour chaque membre qui n'est pas explicitement initialisé, le constructeur sans argument sera automatiquement utilisé.

Pour utiliser les constructeurs, il suffit de déclarer un objet et de spécifier les paramètres du constructeur désiré.

```
1 Point p1; // premier constructeur
2 Point p2(20.0, 10.0); // deuxieme constructeur
```

En C++, lorsqu'aucun constructeur n'est spécifié, le compilateur génère automatiquement deux **constructeurs par défaut**, soit le **constructeur sans argument** et le **constructeur par copie**. Si nous n'avions pas déclaré de constructeur pour la classe `Point`, le compilateur aurait généré les deux constructeurs suivants. Le constructeur `Point()` appelle le constructeur `double::double()` pour les objets `x` et `y`. Évidemment, comme `double` est un type de base, son constructeur ne fait rien. Le constructeur par copie par défaut initialise une copie pour tous les objets membre de la classe, soit `x` et `y`.

```
1 Point::Point()
2 {
3 }
4 Point::Point(const Point& p)
5  : x(p.x), y(p.y)
6 {
7 }
```

C'est généralement dans le constructeur que la mémoire dynamique est allouée. Voici un exemple.

```
1  class Tableau10int{
2    public:
3      Tableau10int();
4      ~Tableau10int();
5    private:
6      int* elements;
7  };
8  Tableau10int::Tableau10int() {
9      elements = new int[10];
10 }
```

La mémoire allouée dynamiquement doit être libérée par le **destructeur**. Chaque classe dispose d'un destructeur nommé par le nom de la classe précédé d'un tilde (~). Le destructeur fonctionne dans le sens inverse du constructeur. Il :

- exécute le code dans le corps du destructeur.
- appelle le destructeur de chaque membre de la classe ;
- appelle le destructeur de la ou des classes héritées ;

Lorsque le destructeur n'est pas défini explicitement, il est généré implicitement par le compilateur. Ce dernier a un corps vide. Si de la mémoire a été allouée dynamiquement, le destructeur par défaut ne libérera pas la mémoire correctement. Il faudra en définir un. Voici le destructeur pour la classe `Tableau10int`.

```
1 Tableau10int::~Tableau10int() {
2     delete [] elements ;
3 }
```

2.14.2 Fonctions membres

Dans une classe, on peut définir des fonctions membres afin d'implémenter des opérations sur les objets instanciés. Voici un exemple.

```
1 double Point::distance(const Point& p) const{
2     double dx = x - p.x;
3     double dy = y - p.y;
4     return sqrt(dx*dx + dy*dy);
5 }
```

Les fonctions membres sont des fonctions comme en langage C, à l'exception qu'elles prennent un paramètre implicite `this`. Le mot clé `this` est un pointeur sur l'objet courant. Par exemple, dans le code ci-dessous, l'expression `a.distance(b)` appelle la fonction `Point::distance` avec deux paramètres : le paramètre implicite `this=&a` et le paramètre explicite `p=b`. Dans le corps d'une fonction d'une classe, lorsqu'on utilise une variable d'instance `x`, cela est équivalent à `this->x`.

```
1 Point a(0,0), b(3,4);
2 double d = a.distance(b);
```

Une fonction a un **effet de bord** qu'elle modifie une ou plusieurs variables autre que ses variables locales. Dans le code précédent, la fonction `Point::distance` est sans effet de bord. Toutefois, une fonction `Point::setX()` aurait un effet de bord, car elle modifie l'objet.

2.14.3 Surcharge d'opérateurs

En C++, il est possible de surcharger des opérateurs. Cela permet à l'utilisateur d'utiliser les objets de façon plus intuitives. Par exemple, on peut additionner deux vecteurs (objets de type `Vecteur`) de façon naturelle à l'aide de l'opérateur `+`.

```
1 Vecteur v1(2,3), v2(10,10);
2 Vecteur v3 = v1 + v2;
```

Pour réaliser cette opération, il faut surcharger l'opérateur + comme suit.

```
1  class Vecteur{
2    public:
3      Vecteur(double vx_=0, double vy_=0):vx(vx_),vy(vy_){}
4      Vecteur& operator += (const Vecteur& v);
5      Vecteur operator + (const Vecteur&) const;
6    private:
7      double vx, vy;
8  };
9  Vecteur& Vecteur::operator += (const Vecteur& autre) {
10     vx+=autre.vx;  vy+=autre.vy;
11     return *this;
12 }
13 Vecteur Vecteur::operator + (const Vecteur& autre) const{
14     return Vecteur(vx+autre.vx, vy+autre.vy);
15 }
```

Il est possible de surcharger un bon nombre d'opérateurs dont : `+`, `-`, `++`, `--`, `+=`, `-=`, `*`, `/`, `*=`, `/=`, `^`, `!`, `=`, `==`, `<`, `>`, `<=`, `>=`, `!=`, `()` et `[]`. Voir références pour la liste complète.

2.14.4 Héritage

Le langage C++ supporte le mécanisme d'héritage. Il est possible de créer de nouvelles classes qui héritent des fonctionnalités d'autres classes.

```
1 class Forme{ ... };
2 class Cercle : public Forme { ... };
3 class Polygone : public Forme { ... };
4 class Carre : public Polygone{ ... };
5 class Rectangle : public Polygone{ ... };
```

2.14.5 Polymorphisme et fonctions virtuelles

Le **polymorphisme** consiste à rendre abstrait certaines opérations sur des types abstraits sans connaître à l'avance quels seront les types exacts qui seront effectivement utilisés.

```
1  class Forme{
2    public:
3      virtual double aire() const = 0;
4      virtual double perimetre() const = 0;
5  };
6  class Cercle : public Forme {
7    public:
8      virtual double aire() const {return PI * rayon*rayon;}
9      virtual double perimetre() const {return 2 * PI * rayon;}
10   private:
11     Point  centre;
12     double rayon;
13 };
14 class Rectangle : public Forme {
15   public:
16     Rectangle(double lon, double lar) : longueur(lon), largeur(lar) {}
17     virtual double aire() const {return longueur * largeur;}
18     virtual double perimetre()const { return 2*(longueur+largeur);}
19   private:
20     double longueur, largeur;
21 };
```

2.15 Mot clé const

Le mot clé **const** est votre ami ! Lorsque const apparaît devant une variable, cela indique que son contenu (la valeur) doit rester constant. C'est une forme de « lecture seule » qui est validée par le compilateur. Toute tentative de modification provoquera une erreur au moment de la compilation. L'usage du mot clé const est fortement encouragé dans diverses situations. Le cas le plus fréquent est le passage en paramètre par **référence constante**. L'avantage du passage par référence par rapport au passage par valeur est d'éviter de copier de gros objets sur la pile d'exécution. Un passage par référence permet

un accès indirect à l'objet d'origine via sa référence (son adresse mémoire) placée sur la pile. Cependant, cela n'est pas sans danger, car la fonction pourrait modifier l'objet alors que la fonction appelante s'attend à ce que son objet local reste intact. Le code suivant montre ce type d'erreur.

```
1 double Point::distance(Point& p2){
2     p2.x -= x; y-=p2.y;
3     return sqrt(p2.x*p2.x + y*y);
4 }
5 int main(){
6     Point p1 = ... , p2 = ...;
7     double d = p1.distance(p2);
8     // p1 et p2 ont ete modifies par Point::distance(...).
9 }
```

L'usage du mot clé `const` permet d'éviter ce type d'erreur. On peut ajouter un premier mot clé `const` devant le paramètre `p2`, et un deuxième `const` à la fin de la signature de la fonction afin de rendre le paramètre implicite (`(*this)`) constant. Ainsi, le code modifié ne compilerait plus. Il faudra le modifier pour obtenir le code suivant.

```
1 double Point::distance(const Point& p2) const{
2     double dx = p2.x-x; double dy-=p2.y-y;
3     return sqrt(dx*dx+dy*dy);
4 }
```

À noter que le mot clé `const` n'est qu'un outil pour vous aider à éviter des modifications par oubli. Le mot clé `const` n'est pas d'une protection absolue. En effet, il est possible de contourner aisément un `const` en l'enlevant. Dans certaines situations, cela peut être nécessaire. Le code ci-dessous montre un exemple.

```
1 void test(const int& a){
2     int& a2 = (int&) a;
3     a2 = 0;
4 }
```

2.16 Fonctions et classes amies (*friends*)

Dans certaines situations, l'usage des modificateurs d'accès `private` et `protected` peut causer des difficultés. Deux exemples sont présentés pour motiver les fonctions amies et les classes amies. Considérez le premier exemple suivant. Ce code génère une erreur puisque `x` et `y` sont privés.

```
1 class Point{
2   private:
3     double x, y;
4 };
5 std::ostream& operator << (std::ostream& os, const Point& p){
6     os << "(" << p.x << "," << p.y << ")";
7     return os;
8 }
```

Il est possible de contourner ce problème en laissant `x` et `y` privés et en ajoutant des *getters* `getX()` et `getY()`. Or, cette solution n'est pas idéale puisqu'elle représente un bris d'abstraction. En effet, on rend public l'accès (en lecture seule) à la représentation interne de l'objet. Remarquez que cela pourrait être encore pire si des *setters* `setX()` et `setY()` étaient définis !

Une meilleure solution consiste à rendre les opérateurs de sortie `<<` et d'entrée `>>` amis avec la classe `Point`. À noter que les opérateurs `<<` et `>>` ne sont pas des fonctions membres de `Point`, mais uniquement des fonctions amies.

```
1  class Point{
2    private:
3      double x, y;
4      friend std::istream& operator >> (std::istream& is, Point& p);
5      friend std::ostream& operator << (std::ostream& os, const Point& p);
6  };
7  std::istream& operator >> (std::istream& is, Point& p){
8      char parouvr, vir, parferm;
9      is >> parouvr >> p.x >> vir >> p.y >> parferm;
10     assert(parouvr=='(' && vir==',' && parferm==')');
11     return is;
12 }
13 std::ostream& operator << (std::ostream& os, const Point& p){
14     os << "(" << p.x << "," << p.y << ")";
15     return os;
16 }
```

L'amitié est également possible entre les classes. Considérons l'exemple suivant où le constructeur de la classe `Vecteur` prend deux objets de type `Point`. Une fois de plus, le code suivant ne compilera pas puisque les membres `x` et `y` sont privés à la classe `Point`.

```
1  class Vecteur{
2      double vx, vy;
3    public:
4      Vecteur(const Point& p1, const Point& p2){
5          vx = p2.x - p1.x;
6          vy = p2.y - p1.y;
7      }
8  };
```

Une solution possible consiste à rendre la classe `Point` amie avec la classe `Vecteur`. Notez que la relation d'amitié est unidirectionnelle. Dans cet exemple, la classe `Point` n'a pas accès aux membres protégés de la classe `Vecteur`.

```
1  class Point{
2      double x, y;
3      friend class Vecteur;
4  };
```

2.17 Généricité (*templates*)

Les langages C et C++ sont des langages fortement typés. Pour répondre à des besoins particuliers, il faut parfois réécrire plusieurs fois le même code, mais avec de très petites variantes. Par exemple, les fonctions `min` et `max` doivent être écrites pour tous les types de données qu'on veut utiliser. Voici trois fonctions `min`, chacune pour un type différent.

```
1 int min(int a, int b){return a<b ? a : b;}
2 int min(float a, float b){return a<b ? a : b;}
3 int min(double a, double b){return a<b ? a : b;}
```

La nécessité de réécrire du code similaire existe aussi pour les classes. Par exemple, on peut avoir besoin de plusieurs classes `Point`, l'une où les coordonnées sont représentées par des entiers (`int`), une autre des `float` ou des `double`.

```
1 class PointI{ int x, y; };
2 class PointF{ float x, y; };
3 class PointD{ double x, y; };
```

Afin d'éviter l'écriture de plusieurs fonctions ou classes très semblables, le langage C++ offre un mécanisme de généricité basé sur des patrons (`templates`). Au lieu d'utiliser des types précis dans les déclarations et les définitions, on peut utiliser des variables de type. À titre d'exemples, voici une fonction générique `min()` et la classe générique `Point`.

```
1 template <class T>
2 T min(const T& a, const T& b){return a<b ? a : b;}
```

```
1 template <class T>
2 class Point{ T x, y; };
3 Point<double> pd;
4 Point<int> pi;
```

En C++, la généricité est traitée par le compilateur. Lorsqu'une fonction ou une classe générique est utilisée, une nouvelle copie typée est dynamiquement générée. Par exemple, à la première occurrence de `Point<int> pi`, le compilateur génère à l'interne (en mémoire) un nouveau type `Point<int>`. Pour obtenir ce nouveau type, le compilateur fait un traitement équivalant à :

- copier et coller le code du patron (le *template*) ;
- rechercher et remplacer toutes les occurrences de `T` par `int`.

Une fois les patrons instanciés à l'interne dans le compilateur, le compilateur compile un code similaire à ceci.

```
1 class Point<double>{ double x, y; };
2 class Point<int>{ int x, y; };
3 Point<double> pd;
4 Point<int> pi;
```

Les patrons sont copiés et adaptés autant de fois que nécessaire. Généralement, on place le code générique dans les fichiers d'entête (`.h`) ou dans des fichiers inclus depuis les entêtes (`.hcc`).

3 Analyse et complexité algorithmique

Cette section fait un rappel des notions de base au sujet de la **complexité algorithmique**. Ce sujet est étudié plus en profondeur dans le cours INF4100.

L'évaluation d'un algorithme est essentiellement réalisée à l'aide de deux mesures :

- la **complexité temporelle** mesure le **temps de calcul** (temps réel ou temps CPU) requis ;
- la **complexité spatiale** mesure la **quantité de mémoire** requise.

Le temps de calcul et la quantité de mémoire requise varient en fonction de plusieurs facteurs. Le **principal facteur est la taille du problème** (ou taille de l'entrée). Par exemple, pour les algorithmes de tri, la taille du problème est tout simplement le nombre d'éléments à trier. On utilise généralement la variable n pour noter la taille du problème. Toutefois, la taille du problème peut être quantifiée à l'aide de plusieurs variables.

Il existe également des **facteurs secondaires**, comme le type de processeur, le système d'exploitation, le langage de programmation, le compilateur et sa configuration, la qualité de l'implémentation de l'algorithme, etc. Bien que ces facteurs secondaires puissent être significatifs en pratique, ils sont généralement indépendants de l'algorithme et de la taille du problème. Pour ces raisons, on leur accorde une moins grande importance.

Enfin, on peut généralement exprimer le temps d'exécution ou la quantité de mémoire requise d'un algorithme à l'aide d'une **fonction** $f(n)$ où n est la taille du problème.

3.1 Analyse empirique

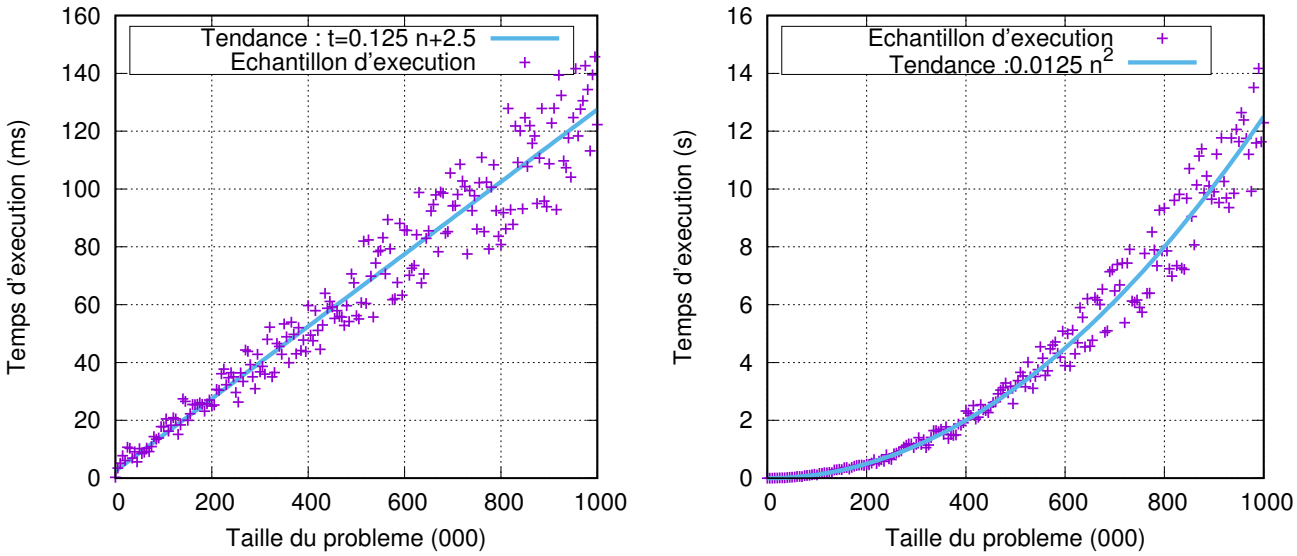
Une façon simple d'évaluer un algorithme consiste à l'implémenter et à l'expérimenter sur plusieurs instances (entrées ou tests) en faisant varier leur taille. Pour chaque instance, on mesure le temps d'exécution et la quantité de mémoire. Après un certain nombre de problèmes, on trace un graphique afin de trouver une relation entre la taille du problème et le temps d'exécution ou la quantité de mémoire utilisée.

Sous Unix, Linux et Mac OS, on peut mesurer le temps d'exécution d'un processus à l'aide de la commande `time`. Par exemple, on peut taper la commande « `time ./lab1 < test5.txt` » pour obtenir le temps d'exécution. En C/C++, il est possible d'avoir accès aux ressources consommées à l'aide de la fonction `getrusage()`. Ainsi, il est possible de mesurer une section précise d'un programme en faisant la différence entre deux mesures par la fonction `getrusage()`.

La Figure 9 montre deux exemples d'analyse sur deux programmes. Sur chaque graphique, on place la taille du problème sur l'axe des X et le temps d'exécution mesuré sur l'axe des Y. Chaque point représente l'exécution de l'algorithme sur une instance donnée. À gauche, on peut remarquer que le temps d'exécution varie de façon linéaire, alors qu'à droite, le temps varie de façon quadratique.

3.2 Analyse asymptotique

Dans une analyse théorique d'un algorithme, on s'intéresse exclusivement aux facteurs principaux, soit généralement la taille du problème. On fait abstraction des facteurs secondaires (machine, langage de programmation), car ils sont propres à chaque environnement d'exécution.



(a) Programme en temps linéaire

(b) Programme en temps quadratique

FIGURE 9 – Analyse empirique du temps d'exécution de deux programmes

Puisqu'on fait abstraction des facteurs secondaires, on ne peut pas mesurer le temps de façon précise (ex. : en secondes). On donne plutôt un **ordre de grandeur**. La **notation grand O** est utilisée pour noter l'ordre de grandeur. Un algorithme est dit $O(n)$ si son temps d'exécution, exprimé à l'aide d'une fonction $f(n)$, est proportionnel à n.

Plus formellement, un ordre de grandeur $O(g(n))$ est un ensemble de fonctions $f(n)$ telles qu'il existe deux constantes non négatives c et k vérifiant $|f(n)| \leq c \cdot |g(n)|$ pour tout $n \geq k$. On peut laisser tomber les valeurs absolues étant donné que le temps d'exécution et que la quantité de mémoire sont des valeurs positives. Ainsi, on obtient : $f(n) \leq c \cdot g(n)$. On peut interpréter cette inégalité par l'affirmation suivante : la fonction $f(n)$ ne croît pas asymptotiquement plus rapidement que $g(n)$.

La constante c précédente peut être vue comme un poids accordé aux facteurs secondaires. Par exemple, soit deux solutions informatiques s_1 et s_2 implémentant un même algorithme, mais écrits dans deux langages de programmation différents et qui s'exécutent sur deux machines différentes. Les solutions s_1 et s_2 auront respectivement un temps d'exécution d'environ $c_1 \cdot f(n)$ et $c_2 \cdot f(n)$. Ainsi, leurs temps d'exécution sont équivalents à une constante multiplicative près.

3.3 Classes de complexités

Le Tableau 3 montre en ordre croissant les classes de complexité les plus fréquentes.

3.4 Méthodes d'analyse

- Compter les opérations.
- Analyser les récurrences.

TABLE 3 – Classes de complexité

Ordre	Complexité	Exemples
$O(1)$	Temps constant	Un accès aléatoire, un calcul arithmétique, etc.
$O(\log n)$	Logarithmique	Recherche dichotomique (binaire) dans un tableau trié.
$O(n)$	Linéaire	Itérer sur les éléments d'un tableau ou d'une liste.
$O(n \log n)$	« n log n »	Tri de fusion et de monceau. Tri rapide (excepté le pire cas).
$O(n^2)$	Quadratique	Parcours d'un tableau 2 dimensions. Tri de sélection.
$O(n^3)$	Cubique	Multiplication matricielle naïve.
$O(b^n)$ où $b \geq 2$	Exponentiel	Problèmes de planification.
$O(n!)$	Factoriel	Problèmes d'ordonnancement. Problème du voyageur de commerce.

3.5 Quoi analyser

- Pire cas : le pire cas où l'algorithme se comporte le moins bien.
- Cas moyen : la moyenne de tous les cas possibles.
- Temps amorti : le temps moyen qu'une opération met lorsqu'elle est répétée exactement une fois sur toutes les instances des éléments. Un exemple est donné à la Section 4.3.

3.6 Études de cas – Algorithmes de tri

3.6.1 Tri de sélection

L'Algorithme 1 montre le tri de sélection. Ce dernier a une complexité temporelle de $O(n^2)$.

Algorithme 1 Tri de sélection

```
1. TRISELECTION(a[0 : n−1])
2.   pour i = 0,...,n−1
3.     k ← i
4.     pour j = k+1,...,n−1
5.       si a[j] < a[k]
6.         k ← j
7.     ÉCHANGER(a[i],a[k])
```

3.6.2 Tri de fusion

L'Algorithme 2 montre le tri de fusion. Ce dernier a une complexité temporelle de $O(n \log n)$.

3.6.3 Tri de monceau

Voir Section 9. Pour faire le tri de monceau, il suffit de construire un monceau à partir d'un vecteur. Le tri de monceau a une complexité temporelle de $O(n \log n)$.

Algorithme 2 Tri de fusion

1. TRIFUSION($a[0:n-1]$)
2. si $n \leq 1$ retourner
3. $m \leftarrow \lfloor n/2 \rfloor$

4. TRIFUSION($a[0:m]$)
5. TRIFUSION($a[m+1:n-1]$)

6. créer $b[0:n-1]$
7. $i \leftarrow 0$
9. $j \leftarrow m+1$
9. $k \leftarrow 0$
10. Tant que $i \leq m$ et $j < n$
11. $b[k++] \leftarrow a[j] < a[i] \,?\, a[j++] : a[i++]$
12. Tant que $i \leq m$
13. $b[k++] \leftarrow a[i++]$
14. Tant que $j < n$
15. $b[k++] \leftarrow a[j++]$
16. $a \leftarrow b$

3.6.4 Tri rapide (*Quicksort*)

Le code suivant montre une implémentation du tri rapide en C/C++. Dans le cas moyen, le tri rapide a une complexité temporelle de $O(n \log n)$. Dans le pire cas, où le pivot est mal choisi, le tri rapide dégénère à une complexité temporelle de $O(n^2)$.

```
1  template <class T>
2  void tri_rapide(T* tab, int n)
3  {
4      if(n<=1) return;
5
6      // Choisir un pivot : il existe plusieurs techniques
7      int p = n/2;  // on pourrait le choisir au hasard!
8      swap(tab[0],tab[p]);
9
10     //Diviser le vecteur en deux
11     int k = 0;
12     for(int i = 1;i<n;i++)
13         if(tab[i] < tab[0])
14             swap(tab[++k],tab[i]);
15     swap(tab[0],tab[k]);
16     //On obtient tab[0:k-1] < tab[k]<= tab[k+1:n-1]
17
18     //Appels recursifs
19     tri_rapide(tab, k);
20     tri_rapide(tab+k+1, n-k-1);
21 }
```

Deuxième partie

Structures linéaires

Cette partie introduit les structures linéaires de base, incluant des tableaux à taille dynamique, des piles, des files et des listes. Bien que certaines notions ont déjà été vues en INF2120, cette partie vous aidera à vous familiariser davantage avec le langage C++. Il y aura certaines nouvelles notions comme le concept d'itérateur de liste.

4 Tableau générique à taille automatique

Les tableaux natifs en C++ (les *arrays*) ont plusieurs limites dont :

- non vérification des indices (problème de robustesse) ;
- taille fixe déterminée à l'allocation ;
- taille du tableau allouée «détachée» du pointeur (il faut se rappeler de la taille allouée) ;
- non manipulable comme un objet.

L'objectif de la présente section est de concevoir un type abstrait de données `Tableau`. Il s'agit d'un excellent exercice pour mettre en pratique le concept de classe modèle (`template class`) en C++ (voir Section 2.17). Ce `Tableau` s'apparente aux classes `Vector` et `ArrayList` en Java que vous devriez avoir vues dans le cours INF2120.

4.1 Représentation et déclaration

Notre classe `Tableau` est en quelque sorte une encapsulation d'un tableau natif de C++. Un objet `Tableau` contient un pointeur vers un tableau natif et deux entiers pour mémoriser la capacité de ce dernier et le nombre d'éléments courants. Le code suivant montre le fichier `tableau.h`. Prenez note à la ligne `template <class T>`. Celle-ci signifie que la classe `Tableau` est générique : ce conteneur stockera des objets de type `T`. Le type `T` est spécifié lorsqu'on définit un objet de type `Tableau`.

```
1  #if !defined(__TABLEAU_H__) // Ces 2 lignes permettent d'eviter
2  #define __TABLEAU_H__      // d'inclure plusieurs fois ce fichier .h
3  template <class T>
4  class Tableau {
5    public:
6      Tableau(int capacite_initiale=4);
7      ~Tableau();
8      int taille() const {return nbElements;}
9    private:
10     T*             elements;
11     int            capacite;
12     int            nbElements;
13 };
14 // mettre la suite ici OU dans tableau.hcc qu'on inclut ici.
15 #endif
```

La Figure 10 illustre la représentation d'un `Tableau`.

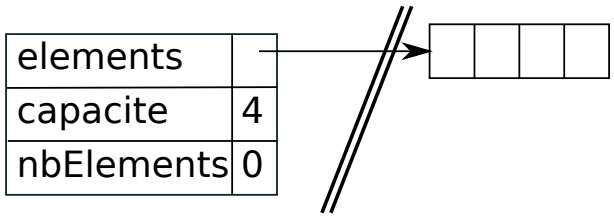


FIGURE 10 – Exemple de représentation de la classe `Tableau`

4.2 Définition du constructeur et du destructeur

Le code suivant définit le constructeur et le destructeur

```
1  template <class T>
2  Tableau<T>::Tableau(int initCapacite) {
3    capacite = initCapacite;
4    nbElements = 0;
5    elements = new T[capacite];
6  }
7  template <class T>
8  Tableau<T>::~Tableau() {
9     delete[] elements;
10    elements = NULL; // optionnel
11 }
```

Ce code peut être placé à deux endroits :

- à la fin de `tableau.h` (après la déclaration, mais avant la directive `#endif`);
- ou dans un deuxième fichier nommé `tableau.hcc` qui est inclut à la fin de `tableau.h`.

Cette règle s'applique aussi aux prochaines fonctions.

4.3 Fonction `ajouter`

L'ajout se fait par une fonction public `ajouter`. Une fonction auxiliaire privée `redimensionner` se chargera d'agrandir le tableau lorsque cela sera nécessaire.

```
1  template <class T> class Tableau {
2    public:
3      //...
4      void          ajouter(const T& item);
5      T&            operator[] (int index);
6      const T&      operator[] (int index) const;
7    private:
8      void          redimensionner(int nouvCapacite);
```

Voici la définition des fonctions `ajouter` et `redimensionner`.

```
1  template <class T>
2  void Tableau<T>::ajouter(const T& item){
3      assert(nbElements <= capacite);
4      if(nbElements == capacite)
5          redimensionner(capacite*2);
6      elements[nbElements++] = item;
7  }
8  template <class T>
9  void Tableau<T>::redimensionner(int nouvCapacite){
10     capacite = nouvCapacite;
11     T* temp = new T[capacite];
12     for(int i=0;i<nbElements;i++)
13         temp[i] = elements[i];
14     delete [] elements;
15     elements = temp;
16 }
```

Quand le tableau a une capacité trop petite, il faut l'agrandir. La politique d'agrandissement doit être soigneusement choisie. Agrandir le tableau d'un élément à chaque fois permet d'éviter le gaspillage de la mémoire. Toutefois, cela est coûteux en temps. En effet, il faudra recopier tous ses éléments à chaque ajout. Ainsi, pour ajouter n élément, il en coûtera $1+2+3+\ldots+n = n(n-1)/2$, soit $O(n^2)$. Donc, chaque ajout coûte $O(n)$ en temps amorti.

Doubler la capacité du tableau est une meilleure politique quant au temps d'exécution. En effet, si on ajoute n éléments dans un tableau, il faudra recopier les éléments à chaque fois que le nombre d'éléments franchis une puissance de deux. Dans ces cas, l'opération d'ajoute coûtera $O(n)$. Toutefois, il faut considérer le temps amorti. Pour faire $n = 2^k$, il y aura k agrandissements occasionnant un coût total de $1+2+4+8+16+...+n = 2n$ recopies. Or, $2n \in O(n)$. Ainsi, l'ajout coûte $O(1)$ en temps amorti.

4.4 Opérateurs []

On peut redéfinir des opérateurs `[]` pour utiliser notre classe tableau de la même façon qu'un tableau natif en C/C++. Il faut généralement définir deux opérateurs `[]`, une version constante et l'une non constante. Ces opérateurs sont comme des fonctions et donnent l'opportunité d'ajouter certains traitements. Par exemple, afin d'intégrer de la robustesse, on peut tester si l'indice est valide.

```
1  template <class T>
2  T& Tableau<T>::operator[] (int index){
3      assert(index<nbElements);
4      return elements[index];
5  }
6
7  template <class T>
8  const T& Tableau<T>::operator[] (int index) const{
9      assert(index<nbElements);
10     return elements[index];
11 }
```

On pourrait aussi faire un autre choix d'implémentation qui consiste à augmenter la taille du tableau au besoin.

```
1 template <class T>
2 T& Tableau<T>::operator[] (int index){
3     if(index>=capacite)
4         redimensionner(max(index, 2*capacite));
5     return elements[index];
6 }
```

4.5 Test d'équivalence

En C++, on peut utiliser l'opérateur == pour tester si deux objets sont équivalents. On veut plutôt vérifier si tous les éléments sont équivalents. À noter que le compilateur ne synthétise pas d'opérateur == par défaut.

```
1 template <class T>
2 bool Tableau<T>::operator == (const Tableau& autre) {
3     if(this==&autre) return true; // meme Tableau
4     if(nbElements!=autre.nbElements) return false;
5     for(int i=0;i<nbElements;i++)
6         if(elements[i] != autre.elements[i])
7            return false;
8     return true;
9 }
```

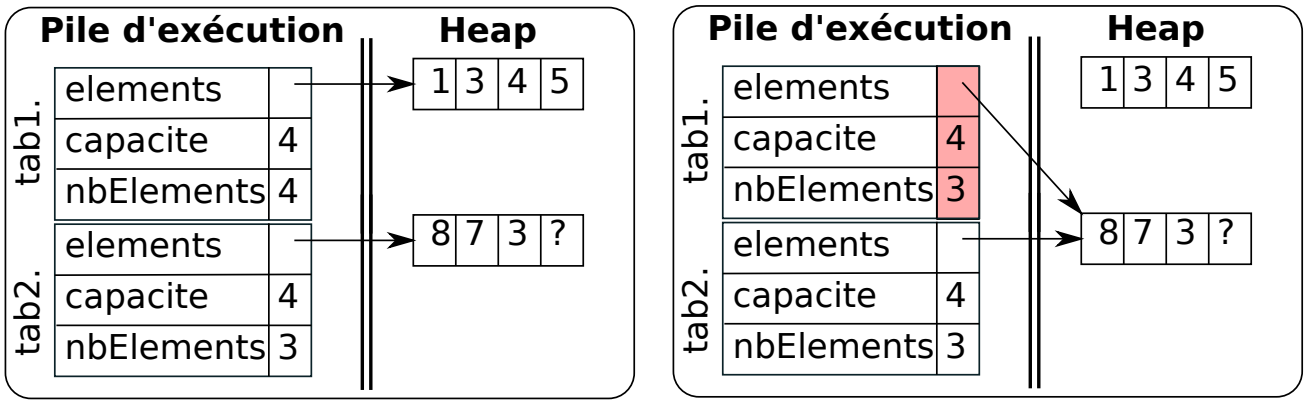
4.6 Opérateur d'affectation (=) et Constructeur par copie

En C++, on peut affecter un objet à un autre en utilisant l'opérateur =. Si l'opérateur = n'est pas défini par le programmeur, le compilateur peut en générer un automatiquement. Toutefois, s'il y a de l'allocation dynamique de la mémoire, cela peut nous jouer des tours. Examinons le cas du code situation.

```
1 void fonction(){
2     Tableau<int> tab1();
3     tab1.ajouter(1); tab1.ajouter(3); tab1.ajouter(4); tab1.ajouter(5);
4     Tableau<int> tab2();
5     tab2.ajouter(8); tab2.ajouter(7); tab2.ajouter(3);
6     tab1 = tab2;
7 }
```

À la ligne 6, l'opérateur d'affectation = est appelé. Comme ce dernier n'a pas encore été défini par le programmeur, le compilateur synthétise **l'opérateur = par défaut**. L'opérateur = par défaut ne fait qu'appeler l'opérateur = sur tous les variables locales de l'objet. La Figure 11 montre le résultat de l'opérateur. Comme on peut le voir, les variables `elements`, `capacite` et `nbElements` sont copiées. Cela n'est pas tout à fait le comportement souhaité car la mémoire de `tab1` est incorrectement libérée. Elle est même perdue, car on n'a plus aucun pointeur référant au tableau contenant les valeur 1,3,4 et 5.

De plus, les pointeurs `elements` des objets `tab1` et `tab2` pointent maintenant vers le même emplacement mémoire. Cela cause quelques problèmes.



(a) Avant la ligne 6 (b) Après la ligne 6

FIGURE 11 – Illustration d'un opérateur = par défaut

Pour éviter ces problèmes, il faut redéfinir l'opérateur = afin de spécifier le comportement souhaité. Pour ce faire, il faut ajouter la déclaration de l'opérateur dans la déclaration de la classe.

```
1  template <class T> class Tableau
2  {
3      //...
4      Tableau(const Tableau&);
5      //...
6      Tableau<T>&   operator = (const Tableau<T>& autre);
7      //...
```

Par la suite, on peut définir l'opérateur. Au lieu de copier le pointeur, il faut plutôt réallouer de la mémoire et recopier les éléments un à un. Ainsi, on obtient deux objets indépendants.

```
1  template <class T>
2  Tableau<T>& Tableau<T>::operator = (const Tableau<T>& autre){
3      if(this==&autre) return *this;//cas special lorsqu'on affecte un
       objet a lui-meme
4      nbElements = autre.nbElements;
5      if(capacite<autre.nbElements){
6          delete[] elements;
7          capacite = autre.nbElements; //ou autre.capacite
8          elements = new T[capacite];
9      }
10     for(int i=0;i<nbElements;i++)
11         elements[i] = autre.elements[i];
12     return *this;
13 }
```

En plus de l'opérateur d'affectation, il faut aussi définir le **constructeur par copie**.

```
template <class T>
Tableau<T>::Tableau(const Tableau& autre) {
    capacite = autre.nbElements; // ou autre.capacite
    nbElements = autre.nbElements;
    elements = new T[capacite];
    for(int i=0;i<nbElements;i++)
        elements[i] = autre.elements[i];
}
```

4.7 Recherche

Si les éléments ne sont pas triés, il faut parcourir tous les éléments du tableau. Donc $O(n)$.

```
template <class T>
bool Tableau<T>::contient(const T& element) {
    for(int i=0;i<nbElements;i++)
        if(elements[i] == element)
            return true;
    return false;
}
```

Toutefois, si le tableau est trié, on peut faire une recherche dichotomique. Cette recherche est efficace puisqu'on réduit de moitié l'espace de recherche à chaque itération. Donc $O(\log n)$.

```
template <class T>
bool Tableau<T>::contient(const T& element) {
    int a=0; b=nbElements;
    while(a<b){
        int c = (a+b)/2;
        if(element<elements[c]) b=c;
        else if(elements[c]<element) a=c+1;
        else return true;
    }
    return false;
}
```

5 Les Piles

Les piles sont des conteneurs basés sur le modèle *LIFO : last-in-first-out* (dernier arrivé, premier servi). Attention, il ne faut pas confondre la **pile d'exécution** (voir Section 2.13) avec la **structure de données pile**. La pile (*stack*) est une structure de données à accès implicite. Par exemple, contrairement à un objet tableau, on accède toujours au sommet de la pile et jamais (ou rarement) aux autres éléments. Cela s'apparente à une pile d'assiettes dans une cafétéria.

5.1 Interface abstraite publique

Le tableau suivant présente les opérations typiques sur une pile.

empiler(*e*)	push(*e*)	Ajoute l'élément *e* au sommet de la pile.
depiler()	pop()	Enlève l'élément au sommet de la pile.
sommet()	top()	Retourne l'élément au sommet de la pile.
taille()	size()	Retourne le nombre d'éléments dans la pile.
vide()	empty()	Retourne vrai si la pile est vide, sinon faux.

Le code suivant présente l'interface publique d'une implémentation en C++.

```
1  template <class T> class Pile {
2    public:
3      Pile();
4      ~Pile();
5      int  taille() const; // fonction optionnelle
6      bool vide() const;
7
8      const T& sommet() const;
9      void empiler(const T& e);
10     // Au choix, l'une des fonctions suivantes :
11     T depiler(); // retourne l'objet qui était au sommet de la pile
12     void depiler(); // l'objet dépile n'est pas retourné
13     void depiler(T& e); // dépile l'élément dans l'objet e en référence
14 };
```

La fonction `depiler()` a plusieurs signatures et retours possibles. La plus naturelle est la première, qui consiste à retourner l'objet qui était au sommet de la pile. Or, elle nécessite de retourner l'objet sur la pile d'exécution, ce qui peut être moins efficace en pratique. Les deux autres options peuvent être plus efficaces.

5.2 Implémentation à l'aide d'un tableau

Il existe plusieurs façons d'implémenter une pile. La représentation la plus simple consiste en un tableau. La voici.

```
1 template <class T> class Pile {
2   public:
3     //...
4   private:
5     Tableau<T> elements;
6 };
```

Pour empiler un élément, il suffit de l'ajouter à la fin du tableau. Pour dépiler un élément, il suffit de réduire la taille du tableau en enlevant le dernier élément.

```
1 template <class T>
2 void Pile::empiler(const T& e) {
3     elements.ajouter(e);
4 };
5 T Pile::depiler() {
6     return elements.enlever_dernier();
7 };
```

La représentation à l'aide d'un tableau n'est pas toujours appropriée, car elle hérite des compromis de la politique d'agrandissement de `Tableau`.

5.3 Implémentation à l'aide de cellules chaînées

La Figure 12 présente un exemple de pile implémentée avec une chaîne de cellules. Un objet pile consiste en un pointeur `sommet` pointant vers la cellule qui contient l'élément au sommet. Chaque cellule contient un pointeur vers la cellule suivante. La dernière cellule, le fond de la pile, contient un pointeur nul. Une pile vide est représentée par un pointeur `sommet` à nul.

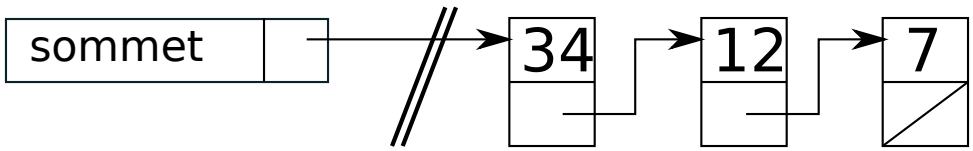


FIGURE 12 – Représentation d'une pile à l'aide de cellules chaînées

Le code C++ pour la représentation d'une pile.

```
1  template <class T> class Pile{
2    public: // ...
3    private:
4      class Cellule{
5        public:
6          Cellule(const T& c, Cellule* s=NULL) : contenu(c), suivante(s){}
7          T contenu;
8          Cellule* suivante;
9      };
10   Cellule* sommet;
11 };
```

Le constructeur crée une pile vide, c'est-à-dire qu'il initialise le pointeur `sommet` à `NULL`. Le destructeur est plutôt simple. Il suffit de vider la pile à l'aide de la fonction `vider()` définie plus loin.

```
1 template <class T>
2 Pile::Pile() : sommet(NULL) {}
3 template <class T>
4 Pile::~Pile() { vider(); }
```

L'empilement d'un élément se fait en insérant une nouvelle cellule au sommet. Le pointeur `suivante` de la nouvelle cellule est l'ancien pointeur `sommet`.

```
1 template <class T>
2 void Pile<T>::empiler(const T& element) {
3     sommet = new Cellule(element, sommet);
4 }
```

Le dépilement d'un élément se fait en détruisant la cellule au sommet. Si cette cellule n'existe pas, c'est qu'il y a une erreur dans le programme. En effet, la fonction `depiler()` ne devrait jamais être appelée sur une pile vide.

```
1 template <class T>
2 void Pile<T>::depiler() {
3     assert(sommet!=NULL);
4     Cellule* anciensommet = sommet;
5     sommet = sommet->suivante;
6     delete anciensommet;
7 }
```

La fonction `depiler()` peut avoir plusieurs signatures et types de retour.

```
1 template <class T>
2 T Pile<T>::depiler() {
3     assert(sommet!=NULL);
4     T element = sommet->contenu;
5     Cellule* anciensommet = sommet;
6     sommet = sommet->suivante;
7     delete anciensommet;
8     return element;
9 }
```

```
1 template <class T>
2 void Pile<T>::depiler(T& sortie) {
3     assert(sommet!=NULL);
4     sortie = sommet->contenu;
5     Cellule* anciensommet = sommet;
6     sommet = sommet->suivante;
7     delete anciensommet;
8 }
```

La fonction `vide()` est plutôt simple : il suffit de tester si le sommet est un pointeur nul.

```
1 template <class T>
2 bool Pile<T>::vide()
3 {
4     return sommet==NULL;
5 }
```

Et voici la fonction `vider()`.

```
1 template <class T>
2 void Pile<T>::vider()
3 {
4     while(!vide())
5         depiler();
6 }
```

5.3.1 Opérateur d'affectation = et Constructeur par copie

De façon similaire à la classe générique `Tableau`, il faut redéfinir l'opérateur d'affectation et le constructeur par copie. Une façon simple d'implémenter l'opérateur = est présentée ci-dessous. Elle consiste à vider la pile au préalable et à créer de nouvelles cellules en copiant le contenu de l'autre pile.

```
1  template <class T>
2  Pile<T>& Pile<T>::operator = (const Pile<T>& autre){
3      if(this==&autre) return *this; // cas special lorsqu'on affecte un
       objet a lui-meme
4      vider(); // ou: while(sommet!=NULL){Cellule*t=sommet->suivante; delete
       sommet; sommet=t;}
5      if(autre.sommet !=NULL){
6          sommet = new Cellule(autre.sommet->contenu);
7          Cellule *cp = autre.sommet;
8          Cellule *c = sommet;
9          while(cp->suivante != NULL){
10             c->suivante = new Cellule(cp->suivante->contenu);
11             c = c->suivante;
12             cp = cp->suivante;
13         }
14     }
15     return *this;
16 }
```

Cette implémentation peut être améliorée. Au lieu de vider au préalable la pile, on peut tenter de réutiliser les cellules existantes en changeant leur contenu. Si l'objet implicite (`*this`) a trop de cellules, on libère les cellules en trop. S'il en manque, on crée les cellules manquantes. L'écriture de ce code est laissée en exercice (question d'examen 2012H).

6 Les Files

Les files sont des conteneurs utilisant le modèle *FIFO : first-in-first-out* (premier arrivé, premier servi). Une file s'apparente à file d'attente pour être servi. Tout comme une pile (*stack*), une file est une structure de données à accès implicite.

6.1 Interface abstraite publique

Le tableau suivant présente les opérations typiques sur une file.

enfiler(*e*)	enqueue(*e*)	Ajoute un élément *e* à la queue de la file.
defiler()	dequeue()	Enlève l'élément à la tête de la file.
tete()	front()	Retourne l'élément à la tête de la file.
taille()	size()	Retourne le nombre d'éléments dans la file.
vide()	empty()	Retourne vrai si la file est vide, sinon faux.

Le code suivant présente l'interface publique d'une implémentation en C++.

```
1  template <class T> class File {
2    public:
3      File();
4      ~File();
5      int  taille() const; // optionnel
6      bool vide() const;
7      void vider();
8      const T& tete() const; // retourne sans enlever l'element en tete
9      void enfiler(const T& e);
10     // Au choix, l'une des fonctions suivantes :
11     T defiler(); // retourne et enleve l'element a la tete
12     void defiler(); // enleve l'element a la tete
13     void defiler(T& e); // copie l'element a la tete dans sortie et l'enleve
14 };
```

6.2 Implémentation avec un tableau circulaire

La Figure 13 montre la représentation d'une file circulaire.

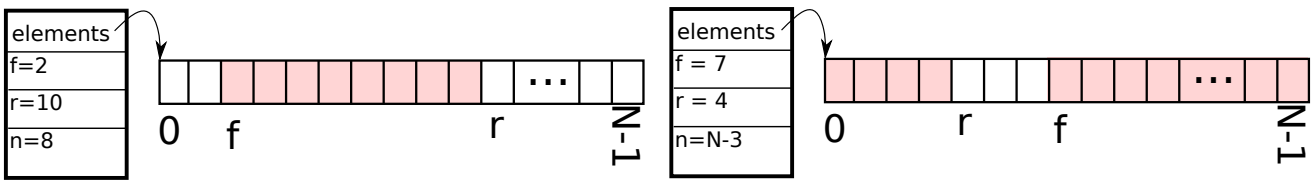


FIGURE 13 – Représentation d'une file circulaire

```
template <class T>
class FileCirculaire{
  public:
    FileCirculaire(int capacite=100);
    ~FileCirculaire();
    //...
  private:
    int capacite; // capacite de elements
    int f; // index sur la tete (front)
    int r; // index sur la cellule suivant la queue
    int n; // nombre d'elements dans la file
    T* elements;
};
```

Constructeur et destructeur.

```
template <class T>
FileCirculaire<T>::FileCirculaire(int _capacity) {
    elements = new T[_capacity];
    capacite = _capacity;
    n = f = r = 0;
}
template <class T>
FileCirculaire<T>::~FileCirculaire() {
  delete[] elements;
  //elements = NULL; // optionnel
}
```

Fonctions simples.

```
template <class T>
int FileCirculaire<T>::taille() const {
    return n;
}
template <class T>
bool FileCirculaire<T>::vide() const {
    return n == 0;
}
template <class T>
void FileCirculaire<T>::vider() const {
    n = f = r = 0;
}
template <class T>
const T& FileCirculaire<T>::tete() const {
    assert(!vide());
    return elements[f];
}
```

Fonctions pour enfiler et défiler.

```
1  template <class T>
2  void FileCirculaire<T>::enfiler(const T& element) {
3      assert(taille()<capacite);
4      // On peut aussi reallouer le tableau dynamiquement
5      elements[r] = element;
6      r = (r+1) % capacite;
7      n++;
8  }
9
10 template <class T>
11 void FileCirculaire<T>::defiler() {
12     assert(!vide());
13     f = (f+1) % capacite;
14     n--;
15 }
```

Les opérateurs d'affectation (=) de test d'équivalence (==) et le constructeur par copie sont laissés en exercice.

6.3 Implémentation avec un chaînage de cellules

La Figure 14 montre la représentation d'une file.

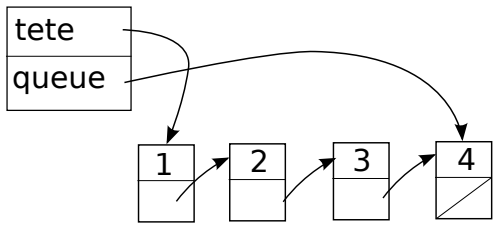


FIGURE 14 – Implémentation naïve d'une file

La Figure 15 montre une représentation améliorée où seul le pointeur `queue` est gardé. Pour accéder à la tête, on utilise le pointeur `suivante` de la cellule queue.

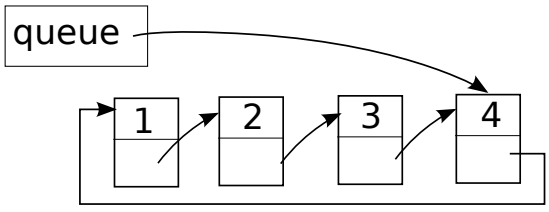


FIGURE 15 – Implémentation d'une file

Voici le constructeur, le destructeur et quelques fonctions.

```
1  template <class T>
2  File<T>::File()
3   : queue(NULL) { }
4  template <class T>
5  File<T>::~File() {
6    vider();
7  }
8
9  template <class T>
10 bool File<T>::vide() const {
11     return queue==NULL;
12 }
13 template <class T>
14 void File<T>::vider(){
15     while(!vide())
16        defiler();
17 }
18 template <class T>
19 const T& File<T>::tete() const {
20     assert(queue!=NULL);
21     return queue->suivante->contenu;
22 }
23
24 template <class T>
25 void File<T>::enfiler(const T& element) {
26     if(queue==NULL){
27         queue = new Cellule(element);
28         queue->suivante = queue;
29     }else
30         queue = queue->suivante = new Cellule(element, queue->suivante);
31 }
32 T File<T>::defiler() // ou return void
33 {
34     Cellule* c = queue->suivante;
35     T e = c->element;
36     if(queue==c)
37         queue = NULL;
38     else
39         queue-> suivante = c->suivante;
40     delete c;
41     return e;
42 }
```

Les opérateurs d'affectation (=) et d'égalité (==), ainsi que le constructeur par copie sont laissés en exercice.

7 Les Listes

Les listes sont des structures linéaires permettant l'insertion et la suppression en temps $O(1)$, et ce, peu importe à quel endroit dans la liste.

7.1 La Liste naïve simplement chaînée

La représentation naïve d'une liste est similaire à celle d'une file ou d'une pile basée sur une liste de cellules vues aux sections 5.3 et 6.3. Le code suivant donne la représentation d'une telle liste.

```
1  template <class T> class Liste{
2    public:
3      Liste();
4      ~Liste();
5      void vider();
6      void insererDebut(const T& e);
7      struct Cellule{ // temporairement public, sera private...
8          Cellule(const T& c, Cellule* s) : suivante(s){contenu=c;}
9          T contenu;
10         Cellule* suivante;
11     };
12     void inserer(const T& e, Cellule* c);
13   private:
14     Cellule*  premiere;
15 };
```

La Figure 16 montre un exemple de représentation de liste. Une liste est représentée par un pointeur vers une première cellule. Chaque cellule de la liste contient un élément de la liste et un pointeur vers la prochaine cellule. La liste est terminée par un pointeur `NULL`.

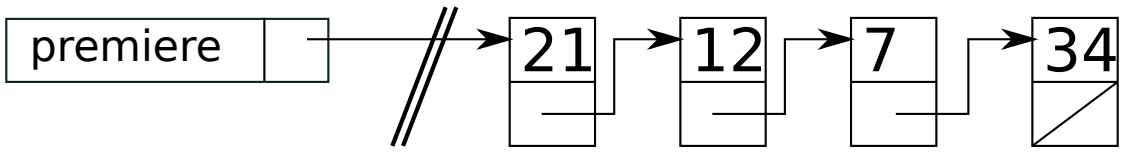


FIGURE 16 – Représentation naïve d'une liste

Pour construire une liste vide, il suffit d'initialiser le pointeur `premiere` à `NULL`. Pour détruire une liste, on peut appeler la fonction `vider()`.

```
1  template <class T> Liste<T>::Liste()
2   : premiere(NULL) {
3  }
4  template <class T> Liste<T>::~Liste() {
5      vider();
6  }
```

L'insertion d'un nouvel élément au début de la liste est simple. Il suffit de créer une nouvelle cellule dont le pointeur `suivante` est le pointeur vers l'ancienne première cellule.

```
1 template <class T> void Liste<T>::insererDebut(const T& element){
2     premiere = new Cellule(element, premiere);
3 }
```

Dans une liste, on peut insérer des éléments à n'importe quelle position, pas uniquement au début ou à la fin. Pour faire une telle insertion, on doit introduire un concept de position dans la liste. On pourrait utiliser, comme dans un tableau, un entier `index`. Or, cela n'est pas efficace dans une liste, car il faudra naviguer dans les cellules pour retrouver l'endroit où faire l'insertion.

```
1 template <class T>  void Liste<T>::inserer(const T& item, int position){
2     Cellule** c = &premiere;
3     for(int i=0;i<position;i++){
4         assert(**c != NULL);
5         c = &((**c)->suivante);
6     }
7     *c = new Cellule(item, *c);
8 }
```

Une meilleure approche consiste à utiliser un pointeur de cellule (voir Figure 17). Pour insérer un élément, on donne le pointeur.

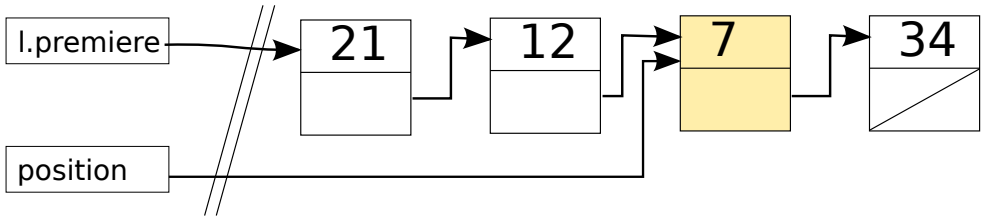


FIGURE 17 – Position dans une liste

Le code suivant montre l'insertion d'un élément **après** une position donnée. Cela est plutôt simple et se fait en $O(1)$. Voir exemple de la Figure 18.

```
1 void Liste<T>::insererApres(const T& element, Cellule* position){
2     position->suivante = new Cellule(element, position->suivante);
3 }
```

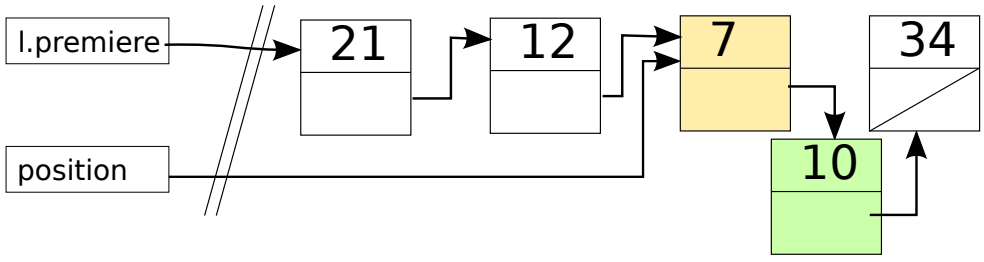


FIGURE 18 – Insertion après une cellule

Cependant, on veut généralement insérer un élément **avant** une position désignée et non après. Comme montré dans le code suivant, cela n'est pas efficace avec une liste naïve puisqu'il faut retrouver le pointeur de la cellule précédente. L'insertion serait donc en $O(n)$. Voir exemple de la Figure 19.

```
1 void Liste<T>::inserer(const T& element, Cellule* position){
2     Cellule* nouvellecellule = new Cellule(element, position);
3     Cellule* c = premiere;
4     while(c->suivante!=position)
5         c = c->suivante;
6     c->suivante = nouvellecellule;
7 }
```

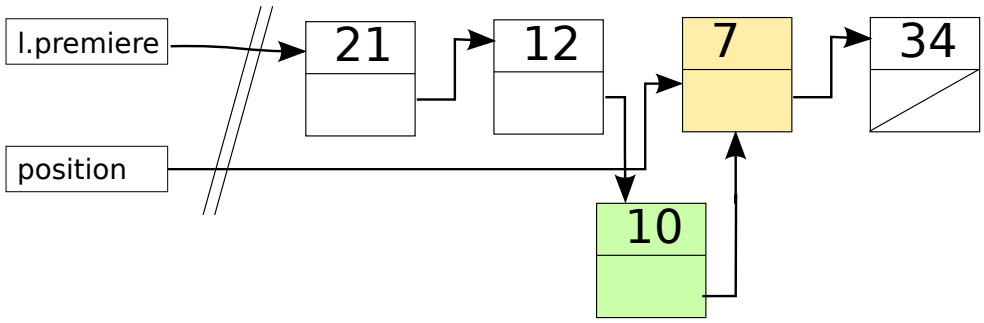


FIGURE 19 – Insertion avant une cellule

Le code suivant propose une solution au problème de performance. La Figure 20 montre un exemple. On crée une nouvelle cellule après la position désignée comme dans `insererApres`. Pour obtenir le comportement souhaité, on copie l'élément à la position désignée dans la nouvelle cellule et on met l'élément à insérer dans l'ancienne cellule. Bien que cette insertion soit en temps $O(1)$, elle a pour désavantage de déplacer des éléments (potentiellement gros). De plus, après l'insertion, la position pointe vers l'élément inséré et non l'ancien élément. L'analyse des enlèvements est laissée en exercice.

```
1 void Liste<T>::inserer(Cellule* position, const T& element){
2     Cellule* nouvellecellule = new Cellule(position->contenu,
     position->suivante);
3     position->suivante = nouvellecellule;
4     position->contenu = element;
5 }
```

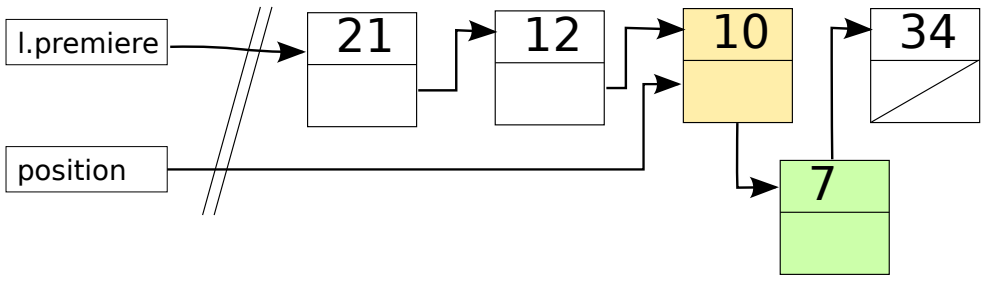


FIGURE 20 – Insertion avant une cellule avec échange de contenu

7.2 Itérateurs de liste

Dans l'ébauche de la liste naïve, l'utilisateur doit fournir un pointeur de `Cellule` pour désigner une position. Par exemple, il peut itérer de la façon suivante.

```
1 Liste<int> liste;
2 liste.inserer(NULL, 2);
3 // ...
4 for(Liste::Cellule* iter=liste.debut();iter!=NULL;iter=iter->suivante)
5     somme += iter->contenu;
```

Rendre public les pointeurs de `Cellule` n'est pas souhaitable. En procédant ainsi, on donne à l'utilisateur un certain accès à la structure de la liste. Il est possible de cacher ces pointeurs en les encapsulant dans des **itérateurs**. Un **itérateur est un objet abstrait qui représente une position dans une structure de données séquentielle**. Les itérateurs sont généralement conçus afin d'être utilisés de façon similaire à un entier servant à itérer dans un tableau.

Dans le cas d'une liste, on peut définir une classe `Liste<T>::Iterateur` comme suit.

```
1  template <class T> class Liste{
2    private:
3      struct Cellule{ // ... };
4      // ...
5    public:
6      // ...
7      class Iterateur{
8        public:
9          Iterateur(const Liste& l) : liste(l), position(l.premiere){}
10         operator bool() const; // retourne vrai ssi position != fin de
       la liste
11         Iterateur& operator++();
12       private:
13         Cellule* position;
14         const Liste& liste;
15       friend class Liste;  // pour des fins de robustesse
16     };
17     T&      operator[](const Iterateur& );
18     constT& operator[](const Iterateur& ) const;
19 };
```

Le code suivant montre un exemple de comment itérer sur une liste à l'aide d'itérateurs.

```
1 Liste<int> liste = ...;
2 for(Liste<int>::Iterateur iter=liste.debut();iter;++iter){
3     somme += liste[iter];
```

Pour des fins de robustesse, on associe à l'itérateur une référence sur la liste sur laquelle il itère. Cela a pour but d'empêcher d'utiliser un itérateur sur une mauvaise liste.

7.3 La Liste simplement chaînée avec pointeurs décalés

Comme il a été dit, la structure de la liste naïve posait quelques difficultés pour retrouver la cellule ayant un pointeur vers la cellule désignée par une position. Il est possible de résoudre ces difficultés en changeant le concept de position dans une liste. On introduit un concept de pointeurs décalés. Au lieu de désigner une position dans une liste à l'aide d'un pointeur, on utilise plutôt un pointeur vers l'élément précédent. Par exemple, à la Figure 21, la position de l'élément 7 est désignée par un pointeur sur la cellule précédente contenant 12. Ainsi, il devient facile d'insérer devant 7 ou d'enlever le 7, car on a plus besoin de recherche la cellule précédente afin de mettre à jour le pointeur `suivante`.

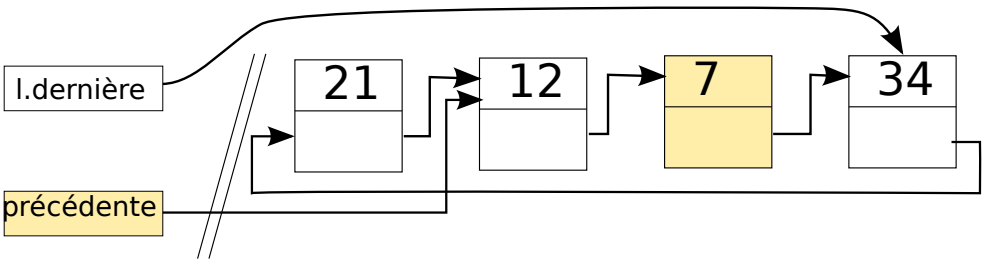


FIGURE 21 – Liste chaînée avec un concept de pointeurs décalés (sur le précédent)

Ainsi, une liste simplement chaînée est représentée par un pointeur vers la dernière cellule, qui elle contient un pointeur vers la première cellule. De façon similaire, un itérateur est représenté par un pointeur vers la cellule précédente.

```
1  template <class T> class Liste {
2    private:
3      struct Cellule{
4        Cellule(const T& c, Cellule* s=NULL) : suivante(s) { contenu=c; }
5        T contenu;
6        Cellule* suivante;
7      };
8      Cellule* derniere;
9
10   public:
11     Liste();
12     ~Liste();
13
14     class Iterateur{
15         Cellule* precedente;
16         const Liste& liste; // pour des fins de robustesse.
17         friend class Liste;
18       public: // ...
19         Iterateur(const Iterateur& i) : liste(i.liste),
       precedente(l.precedente) {}
20         Iterateur(const Liste& l) : liste(l), precedente(l.derniere){}
21         operator bool() const;
22         bool operator!() const;
23         bool operator==(const Iterateur&) const;
24         bool operator!=(const Iterateur&) const;
25         Iterateur operator++(int);
26         Iterateur& operator++();
27         const T& operator*() const;
28         Iterateur operator+(int) const;
29         Iterateur& operator+=(int);
30         Iterateur& operator = (const Iterateur&);
31     };
32
33     bool estVide() const;
34     void vider();
35     const Liste& operator = (const Liste&);
36     T& operator[] (const Iterateur&);
37     const T& operator[] (const Iterateur&) const;
38     Iterateur inserer(const T&, const Iterateur&);
39     Iterateur enlever(const Iterateur&);
40     Iterateur debut() const;
41     Iterateur fin() const;
42     Iterateur trouver(const T&) const;
43 };
```

Voici le code pour l'insertion et l'enlèvement.

```
1  template <class T>
2  typename Liste<T>::Iterateur Liste<T>::inserer(const T& e, const
       Iterateur& i)
3  {
4    assert(this == &i.liste);
5    Iterateur position(i);
6    Cellule* c = i.precedente;
7    if(derniere==NULL)
8    {
9      derniere = new Cellule(e);
10     c = derniere->suivante = derniere;
11   }else
12   if(c==NULL)
13   {
14     c=derniere;
15     derniere->suivante = new Cellule(e, derniere->suivante);
16     derniere = derniere->suivante;
17   }else
18     c->suivante = new Cellule(e, c->suivante);
19   position.precedente = c;
20   return position;
21 }
22
23 template <class T>
24 typename Liste<T>::Iterateur Liste<T>::enlever(const Iterateur& i)
25 {
26   assert(&i.liste == this);
27   Iterateur position(i);
28   Cellule* c = i.precedente;
29   assert(c!=NULL && derniere!=NULL);
30   Cellule* temp = c->suivante;
31   c->suivante = temp->suivante;
32   delete temp;
33   Cellule* retour = temp==derniere ? NULL : c;
34   if(derniere==temp) derniere = c;
35   if(temp == c)
36     derniere = c = NULL;
37   position.precedente = retour;
38   return position;
39 }
```

```
1  template <class T>
2  void Liste<T>::vider(){
3    Cellule* c = derniere;
4    while(c!=NULL) { // ou while(c)
5      Cellule* t = c->suivante;
6      delete c;
7      c = t;
8      if(c==derniere) c = NULL;
9    }
10   derniere = NULL;
11 }
12 template <class T>
13 T& Liste<T>::operator [](const Iterateur& i) {
14   assert(&i.liste == this);
15   assert(i.precedente!=NULL);
16   return i.precedente->suivante->contenu;
17 }
18 template <class T>
19 const T& Liste<T>::operator [](const Iterateur& i) const {
20   assert(&i.liste == this);
21   assert(i.precedente!=NULL);
22   return i.precedente->suivante->contenu;
23 }
24 template <class T>
25 typename Liste<T>::Iterateur Liste<T>::debut() const {
26   return Iterateur(*this); // voir constructeur Iterateur
27 }
28 template <class T>
29 typename Liste<T>::Iterateur Liste<T>::fin() const {
30   Iterateur iter(*this);
31   iter.precedente = NULL;
32   return iter;
33 }
34 template <class T>
35 typename Liste<T>::Iterateur Liste<T>::trouver(const T& e) const {
36   Iterateur iter(*this);
37   while(iter && !(*iter == e))
38     iter++;
39   return iter;
40 }
41 template <class T>
42 typename Liste<T>::Iterateur& Liste<T>::Iterateur::operator ++ () {
43    precedente = precedente->suivante;
44    if(precedente==liste.derniere) precedente = NULL;
45    return *this;
46 }
```

7.4 La Liste doublement chaînée

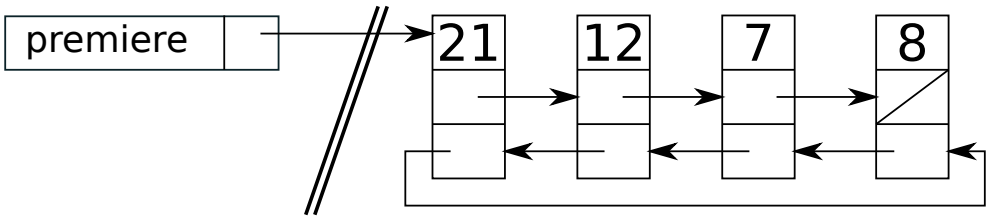


FIGURE 22 – Liste doublement chaînée

```
1  template <class T>
2  class ListeD{
3    public:
4      ListeD();
5      ~ListeD();
6    private:
7      struct Cellule{
8          T contenu;
9          Cellule* precedente;
10         Cellule* suivante;
11     };
12     Cellule* premiere;
13 };
```

Troisième partie

Structures de données avancées

8 Les Arbres

Les arbres sont utilisés dans une multitude d'applications. Grâce à leur structure sous forme d'arborescence, les arbres permettent d'organiser l'information afin qu'elle soit plus facile d'accès. Un système de fichiers est un bel exemple d'application d'une structure arbrescente (voir Figure 23).

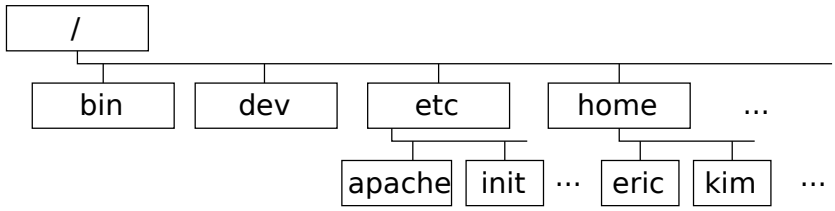


FIGURE 23 – Arborescence d'un système de fichiers

La Figure 24 présente une classification simplifiée des principaux types d'arbres abordés dans cette section. Dans un premier temps, nous introduirons les arbres généraux, soit ceux n'ayant aucune particularité. Par la suite, nous étudierons les arbres binaires, les arbres binaires de recherche et les arbres équilibrés (AVL, rouge-noir, B-Tree et 2-3).

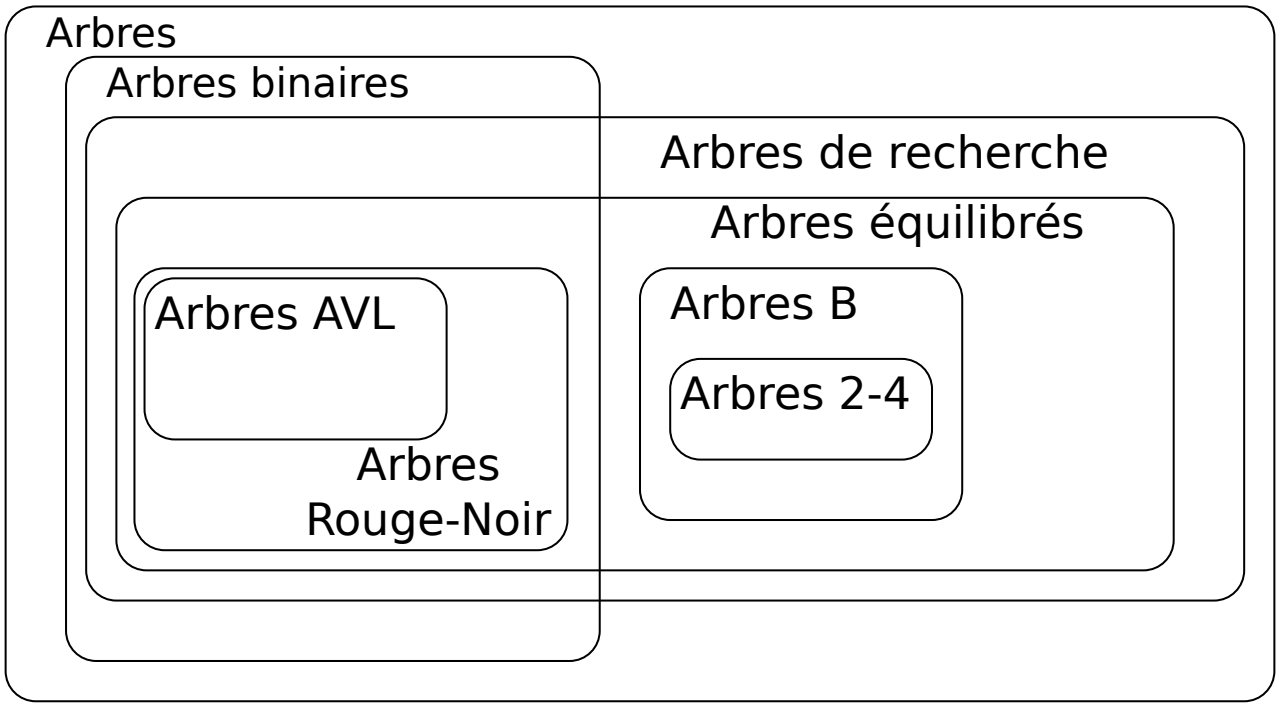


FIGURE 24 – Classification des arbres étudiés

8.1 Définitions

Un **arbre** est un ensemble de **nœuds** liés par des relations de type **parent-enfant**. Chaque nœud a exactement un parent, à l'exception de la racine qui n'a aucun parent. Un **nœud** dans un arbre sert à stocker une valeur ou un objet (des données).

La **racine** est le premier nœud d'un arbre. Tous les autres nœuds sont accessibles depuis la racine. La racine n'existe pas lorsque l'arbre est vide. Une **feuille** est un nœud sans enfant. Les nœuds entre la racine et les feuilles sont des **nœuds internes**.

Il existe plusieurs relations possibles entre les nœuds. Un nœud **parent** peut avoir des nœuds **enfants** (*children*). Un nœud **frère** (*siblings*) d'un nœud est un nœud ayant le même nœud parent. Un nœud **ancêtre** d'un nœud est accessible à travers des relations parents. Si x est le parent de n, alors $x \in ancetres(n)$. Si $x \in ancetres(n1)$ et $n1$ est le parent de n, alors $x \in ancetres(n)$. Un nœud **descendant** est accessible à travers des relations enfants. Si x est un enfant de n, alors $x \in descendants(n)$. Si $x \in descendants(n1)$ et $n1$ est un enfant de n, alors $x \in descendants(n)$.

Il existe plusieurs définitions pour la **hauteur** d'un arbre. Certains auteurs définissent la hauteur d'un arbre comme étant la longueur du plus long chemin (de la racine vers une feuille) dans l'arbre. Sous cette définition, un arbre n'ayant qu'un seul nœud a une hauteur de zéro (0). Une autre définition est retenue pour les présentes notes de cours. La **hauteur** d'un arbre est le nombre de nœuds sur le chemin le plus long reliant la racine à une feuille. Sous cette définition, l'arbre vide a une hauteur de zéro (0). La **profondeur** d'un enfant x est de $p+1$ où $p = profondeur(parent(x))$.

Un **sous-arbre** d'un arbre a est l'arbre engendré à partir d'un nœud enfant de a.

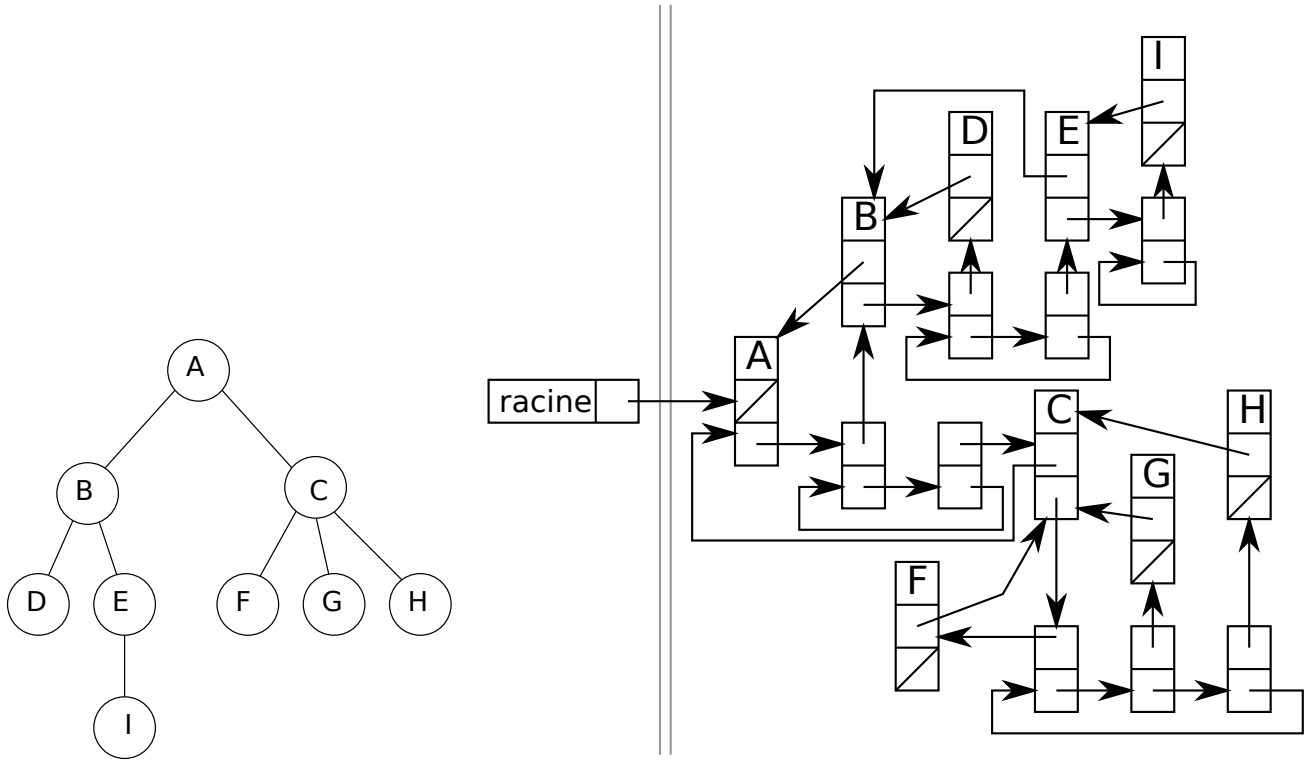
Un **arbre de recherche** est un arbre organisé de façon à rendre les recherches d'éléments (clés) efficaces. Par exemple, les éléments peuvent être ordonnés de façon à ne pas visiter tout l'arbre en entier pour trouver un nœud recherché.

8.2 Représentation d'un arbre

Le code suivant donne une représentation possible pour représenter un arbre.

```
1  template <class T>
2  class Arbre{
3  private:
4    class Noeud{
5      //...
6      private:
7        T contenu;
8        Noeud* parent;
9        Liste<Noeud*> enfants;
10   };
11   Noeud* racine;
12 public:
13   //...
14 };
```

La Figure 25 montre un arbre et sa représentation en mémoire en utilisant la classe `Arbre` précédente.

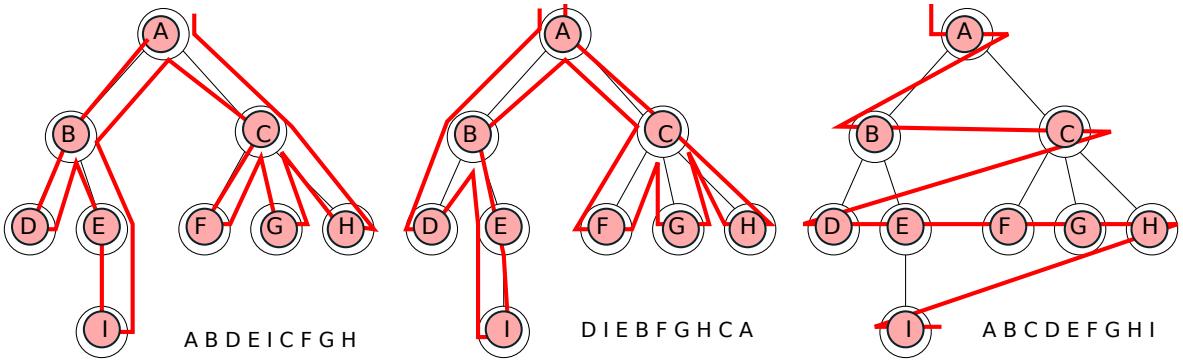


(a) Un arbre (b) Représentation correspondante utilisant la classe `Arbre`

FIGURE 25 – Exemples de représentations

8.3 Parcours dans un arbre

La Figure 26 montre les trois types de parcours dans un arbre. Le parcours en **préordre** traite le nœud courant avant de parcourir ses nœuds enfants. Le parcours en **postordre** traite le nœud courant après avoir parcouru ses nœuds enfants. Le parcours en **largeur** traite les nœuds par profondeur.



(a) Préordre (b) Postordre (c) Largeur

FIGURE 26 – Exemples de parcours d'arbre

8.4 Approche récursive pour l'implémentation des opérations

Dans l'implémentation des opérations des arbres, nous utiliserons généralement une approche **récursive**, puisqu'elle est généralement plus intuitive et plus compacte à écrire, donc plus facile à comprendre. Les opérations implémentées par une approche récursive nécessiteront généralement au moins deux fonctions. La première est une fonction publique qui sert de point d'entrée. Celle-ci fait un appel à la deuxième fonction récursive avec la racine de l'arbre. La deuxième fonction est protégée ou privée. Celle-ci fait le traitement récursif pour réaliser l'opération. À noter que toute fonction récursive peut toujours être traduite en fonction non récursive.

Le code suivant présente les déclarations requises pour réaliser les parcours d'arbre de façon récursive.

```
1  template <class T>
2  class Arbre{
3    private:
4      //...
5      void afficherPreOrdre(Noeud* n) const;
6      void afficherPostOrdre(Noeud* n) const;
7    public:
8      void afficherPreOrdre() const;
9      void afficherPostOrdre() const;
10 }
```

Enfin, voici la définition des fonctions.

```
1  template <class T>
2  void Arbre<T>::afficherPreOrdre() const{
3      afficherPreOrdre(racine);
4  }
5  template <class T>
6  void Arbre<T>::afficherPreOrdre(Noeud* n) const{
7      if(n==NULL) return;
8      std::cout << n->contenu << " ";
9      Liste<Noeud*>::Iterateur iter = n->enfants.debut();
10     while(iter) afficherPreOrdre(*iter++);
11 }
12 template <class T>
13 void Arbre<T>::afficherPostOrdre() const{
14     afficherPostOrdre(racine);
15 }
16 template <class T>
17 void Arbre<T>::afficherPostOrdre(Noeud* n) const{
18     if(n==NULL) return;
19     Liste<Noeud*>::Iterateur iter = n->enfants.debut();
20     while(iter) afficherPostOrdre(*iter++);
21     std::cout << n->contenu << " ";
22 }
```

8.5 Approche non récursive pour l'implémentation des opérations

Certaines opérations s'implémentent mieux à l'aide de fonctions non récursives. Par exemple, le code suivant présente un parcours en largeur.

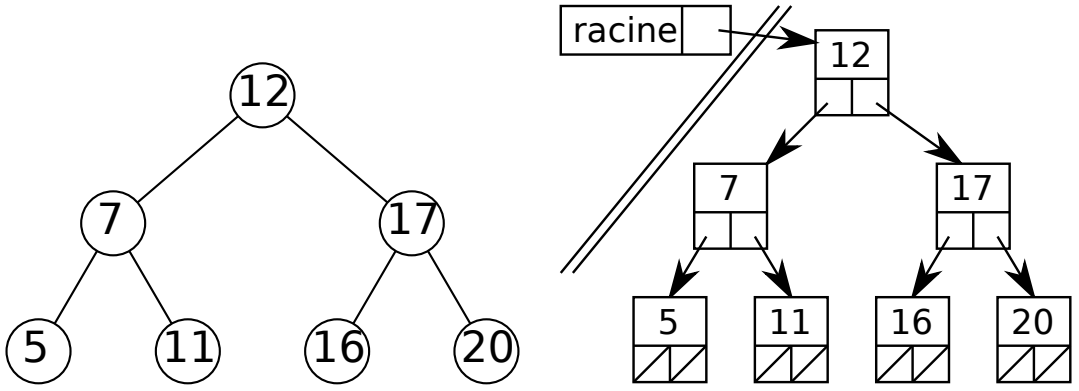
```
1  template <class T>
2  void Arbre<T>::afficherLargeur(){
3      File<Noeud*> noeuds_a_visiter;
4      if(racine!=NULL) noeuds_a_visiter.enfiler(racine);
5      while(!noeuds_a_visiter.estVide()){
6          Noeud* courant = noeuds_a_visiter.defiler();
7          std::cout << courant->contenu << " ";
8          Liste<Noeud*>::Iterateur iter = courant->enfants.debut();
9          while(iter)
10             noeuds_a_visiter.enfiler(*iter++);
11   }
12 }
```

8.6 Arbres binaires

Un **arbre binaire** est un arbre dans lequel tous les nœuds ont au plus deux enfants. Les deux enfants d'un nœud sont généralement appelés enfant de **gauche** et enfant de **droite**. La représentation d'arbre général donnée à la Section 8.2 n'est pas la plus appropriée pour les arbres binaires. Nous utiliserons plutôt la représentation ci-dessous. À noter que le pointeur `parent` n'est plus conservé. Il ne sera plus nécessaire, car les opérations s'implémentent en partant de la racine vers les nœuds concernés.

```
1  template <class T>
2  class ArbreBinaire{
3    public:
4      ArbreBinaire();
5      ~ArbreBinaire();
6      void vider();
7    private:
8      class Noeud{
9          T contenu;
10         Noeud* gauche;
11         Noeud* droite;
12     };
13     Noeud* racine;
14     vider(Noeud*);
15 };
```

La Figure 27 montre un exemple de représentation d'arbre binaire.



(a) Un arbre binaire (b) Représentation avec la classe `ArbreBinaire`

FIGURE 27 – Exemple de la représentation d'un arbre binaire en mémoire

```
1  template <class T> ArbreBinaire<T>::ArbreBinaire() : racine(NULL) {}
2  template <class T> ArbreBinaire<T>::~ArbreBinaire(){
3      vider();
4  }
5  template <class T> void ArbreBinaire<T>::vider(){
6      vider(racine);
7      racine=NULL;
8  }
9  template <class T> void ArbreBinaire<T>::vider(Noeud* n){
10     if(n==NULL) return;
11     vider(n->gauche); vider(n->droite);
12     delete n; // Facultatif : n=NULL;
13 }
```

8.6.1 Parcours en inordre

En plus des parcours en préordre, postordre et largeur (Section 8.3), un arbre binaire permet un **parcours en inordre**. Le code suivant montre le parcours en inordre. Cela consiste à visiter récursivement le sous-arbre de gauche en premier, traiter le nœud courant et enfin visiter récursivement le sous-arbre de droite.

```
1  template <class T>
2  void ArbreBinaire<T>::afficher_inordre(){
3      afficher_inordre(racine);
4  }
5  template <class T>
6  void ArbreBinaire<T>::afficher_inordre(Noeud* n){
7      if(n==NULL) return;
8      afficher_inordre(n->gauche);
9      std::cout << n->contenu << " ";
10     afficher_inordre(n->droite);
11 }
```

8.7 Arbres binaires de recherche

Un **arbre binaire de recherche** est un arbre binaire où il y a une **relation d'ordre** entre un parent et ses enfants : *gauche* $<$ *parent* $<$ *droite*. Cette organisation permet d'effectuer des recherches efficaces. La Figure 28 montre un exemple d'arbre binaire de recherche. Dans un arbre binaire de recherche, un parcours en inordre visite les éléments dans leur ordre naturel.

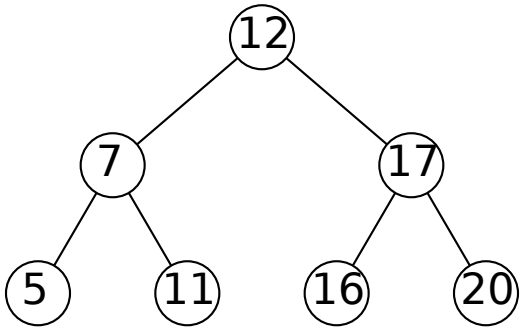


FIGURE 28 – Arbre binaire de recherche

Le code suivant montre la déclaration de la classe `ArbreBinRech`. Comme la représentation ne change pas, `ArbreBinRech` hérite de `ArbreBinaire` (Section 8.6). Une fois de plus, nous utiliserons une approche récursive pour l'implémentation des opérations de recherche, d'insertion et d'enlèvement.

```
1  template <class T>
2  class ArbreBinRech : public ArbreBinaire{
3    public:
4      bool contient(const T& element) const;
5      const T* rechercher(const T& element) const;
6      void inserer(const T& element);
7    private:
8      const T* rechercher(Noeud* n, const T& element) const;
9      void inserer(Noeud*& n, const T& element);
10 };
```

8.7.1 Recherche

Une recherche dans un arbre permet de trouver et localiser un élément afin de vérifier sa présence ou d'effectuer un autre traitement. Le code ci-dessous montre une implémentation récursive de la fonction `contient` qui permet de vérifier l'existence d'un élément dans l'arbre à l'aide d'une recherche binaire. La recherche commence par la racine. Si le pointeur courant est `NULL`, alors l'élément recherché n'est pas présent dans l'arbre. On retourne donc `false`. Si l'élément recherché est celui dans le nœud courant, alors on retourne `true`. Sinon, on compare l'élément recherché avec le nœud courant (ligne 8). S'il est plus petit, un appel récursif est lancé dans le sous-arbre de gauche. S'il est plus grand, ce sera dans le sous-arbre de droite.

```
1  template <class T>
2  bool ArbreBinRech<T>::contient(const T& element) const {
3      return recherche(racine, element)!=NULL;
4  }
5  template <class T>
6  const T* ArbreBinRech<T>::recherche(Noeud*noeud, const T& element)const{
7      if(noeud==NULL) return NULL;
8      if(element==noeud->contenu) return &(noeud->contenu);
9      if(element<noeud->contenu)
10         return recherche(noeud->gauche, element);
11     else
12         return recherche(noeud->droite, element);
13 }
```

La Figure 29 montre un exemple de recherche de l'élément 11. Les nœuds visités sont en vert.

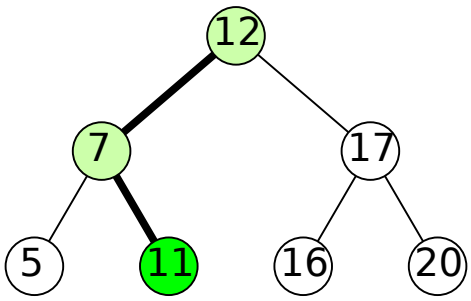


FIGURE 29 – Exemple de recherche de l'élément 11 dans un binaire

8.7.2 Insertion

L'insertion dans un arbre binaire de recherche fonctionne de façon similaire à la recherche. On commence par chercher l'endroit où serait l'élément à insérer. Si l'élément n'est pas déjà présent dans l'arbre, alors on l'insère à cet endroit. La Figure 30 montre l'insertion successif des éléments 8 et 9 dans un arbre.

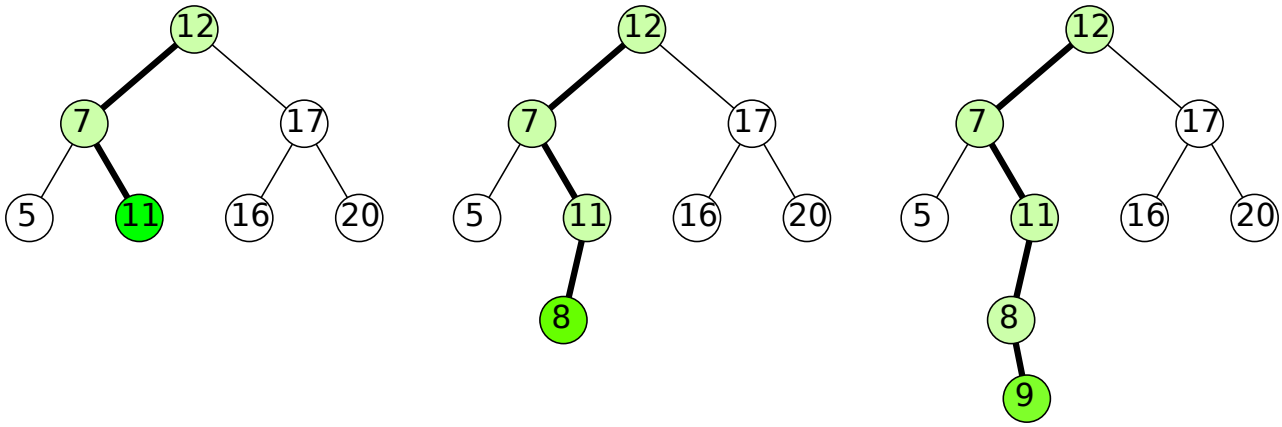


FIGURE 30 – Exemple d'insertion dans un arbre binaire de recherche

Le code suivant présente une implémentation de l'insertion. Remarquez la signature de la fonction privée `inserer`. Le pointeur `courant` est de type `Noeud*&`. Il s'agit d'une référence sur un pointeur de `Noeud`. Cette astuce élégante permet de sauver du code et d'unifier plusieurs cas. Sans cette astuce, il aurait fallu mettre plusieurs `if` pour traiter le cas limite (insertion dans un arbre vide) et les insertions à gauche et à droite. En utilisant une référence, on délègue l'appel du `new` à l'appel récursif suivant.

```
1  template <class T>
2  void ArbreBinRech<T>::inserer(const T& element) {
3      inserer(racine, element);
4  }
5  template <class T>
6  void ArbreBinRech<T>::inserer(Noeud*& noeud, const T& element) {
7      if(noeud==NULL)
8          noeud = new Noeud(element);
9      if(element==noeud->contenu) return;
10     if(element<noeud->contenu)
11         inserer(noeud->gauche, element);
12     else
13         inserer(noeud->droite, element);
14 }
```

La deuxième fonction `insérer` requiert un constructeur `ArbreBinRech::Noeud`.

```
1  template <class T>
2  ArbreBinRech<T>::Noeud::Noeud(const T& element)
3   : gauche(NULL), droite(NULL) {
4      contenu = element;
5  }
```

8.7.3 Enlèvement

Les fonctions d'enlèvement seront présentées en classe.

8.8 Arbres binaires de recherche équilibrés

Les arbres binaires de recherche peuvent s'avérer inefficaces s'ils ne sont pas adéquatement **équilibrés**. Par exemple, en regardant la Figure 30, vous avez peut-être anticipé un problème suite à l'insertion de deux éléments. En effet, l'insertion crée une branche plus longue que toutes les autres.

Le pire cas se produit lorsqu'on insère des éléments totalement ordonnés (croissant ou décroissant). La Figure 31 montre deux exemples de cas dégénérés. Dans le pire cas, la recherche s'effectue en temps $O(n)$ où n est le nombre de nœuds.

Pour garantir une recherche plus efficace, s'effectuant en temps $O(\log n)$, il faut être en mesure d'éliminer environ la moitié de l'espace de recherche à chaque étape. Afin de couper par deux l'espace de recherche, les sous-arbres de gauche et de droite doivent être **équilibrés** en terme de nombre de nœuds qu'ils contiennent.

(a) Arbre résultant de l'insertion de 1, 2, 3 et 4 (b) Arbre résultant de l'insertion de 4, 3, 2, 1

FIGURE 31 – Exemples d'arbres dégénérés

Il y a plusieurs façons de définir la notion **d'arbre équilibré**. Nous en verrons quelques unes. Par exemple, l'**équilibre parfait** est atteint lorsque pour tous les nœuds, leurs sous-arbres (gauche et droite) ont exactement le même nombre de nœuds (ou la même hauteur). Cette définition est plutôt contraignante car elle ne peut s'appliquer qu'aux arbres ayant $2^h - 1$ nœuds, où $h \in \mathbb{N}$. Cependant, cette définition peut être assouplie : un arbre est **quasi parfaitement équilibré** si tous ses niveaux sont remplis, mais à l'exception du dernier niveau.

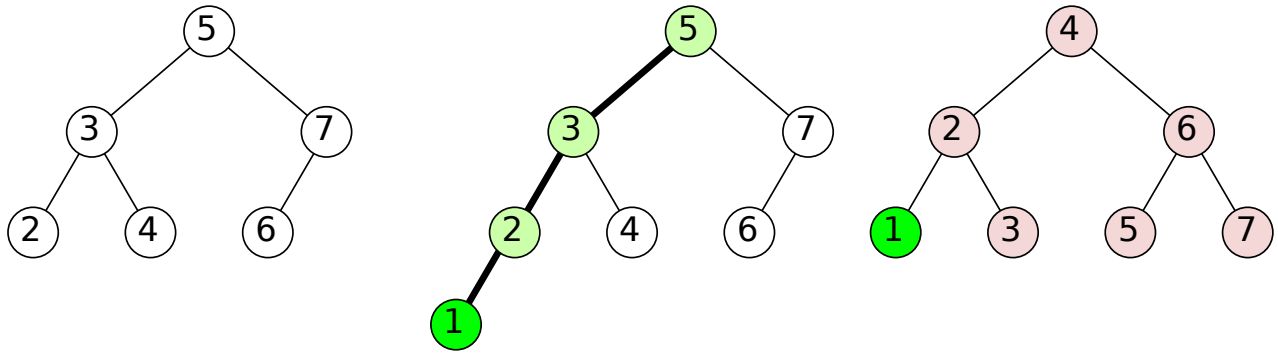
Il y a deux approches possibles pour travailler avec des arbres équilibrés. La première approche consiste à **retarder l'équilibrage** jusqu'au moment où il est nécessaire. Par exemple, on peut faire plusieurs insertions et enlèvements qui brisent l'équilibre. Ensuite, à un certain moment, on rééquilibre l'arbre au complet. La deuxième approche consiste à **maintenir l'arbre équilibré après chaque modification**. Ainsi, il faut modifier les opérations d'insertion et d'enlèvement pour maintenir l'arbre équilibré en tout temps. Dans le reste de cette section, nous adopterons la deuxième approche.

Pour détecter si une opération a déséquilibré un arbre, il suffit de comparer le nombre de nœuds ou la hauteur des sous-arbres par chaque nœud mis à jour. Cela peut se faire à l'aide de la fonction hauteur suivante.

```
1 template <class T>
2 int ArbreBinRech<T>::hauteur(Noeud* noeud) const {
3     if(noeud==NULL)
4         return 0;
5     return max(hauteur(noeud->gauche), hauteur(noeud->droite))+1;
6 }
```

Cependant, quel est le coût de la fonction `hauteur` ? Dans le pire cas, lorsqu'appelée à la racine, cette fonction récursive visitera tout l'arbre en entier. Pour contourner ce problème, il faut modifier la représentation des nœuds afin de conserver la hauteur dans chaque nœud.

L'équilibre quasi parfait est difficile à maintenir. La Figure 32 montre un exemple d'un cas dégénéré où l'insertion d'un nouvel élément cause la réorganisation complète de l'arbre. Ainsi, l'insertion coûterait $O(n)$, soit aussi pire que l'insertion dans un tableau linéaire ! Pour contourner ce problème, il est possible d'assouplir la définition d'arbre équilibré. C'est ce que nous verrons dans les prochaines sections.



(a) Arbre initial (b) Arbre pendant l'insertion de 1 (c) Arbre rééquilibré

FIGURE 32 – Réorganisation complète de l'arbre pour maintenir l'équilibre suite à une insertion

8.9 Arbres AVL

Les arbres **AVL** [AVL62] portent le nom de leurs inventeurs, les russes Georgii **A**delson-**V**elsky et Evguenii **L**andis. Un arbre AVL est un arbre binaire de recherche et équilibré. La notion d'équilibre dans un arbre AVL est assouplie par rapport à la définition d'arbre quasi parfaitement équilibré introduite dans la Section 8.8. **Pour chacun des nœuds d'un arbre AVL, la différence de hauteur entre le sous-arbre de gauche et le sous-arbre de droite est d'au plus de un**, c.-à-d. $|h_g - h_d| \leq 1$. Pour maintenir l'équilibre, des rotations sont faites autour des nœuds lorsque la contrainte $|h_g - h_d| \leq 1$ est violée. Cette règle d'équilibre permet de garantir que les opérations de recherche, d'insertion, et d'enlèvement demeurent toujours en temps $O(\log n)$. La Figure 33 présente les arbres minimaux, c'est-à-dire les arbres ayant le nombre minimal de nœuds pour une hauteur 0, 1, 2, etc.

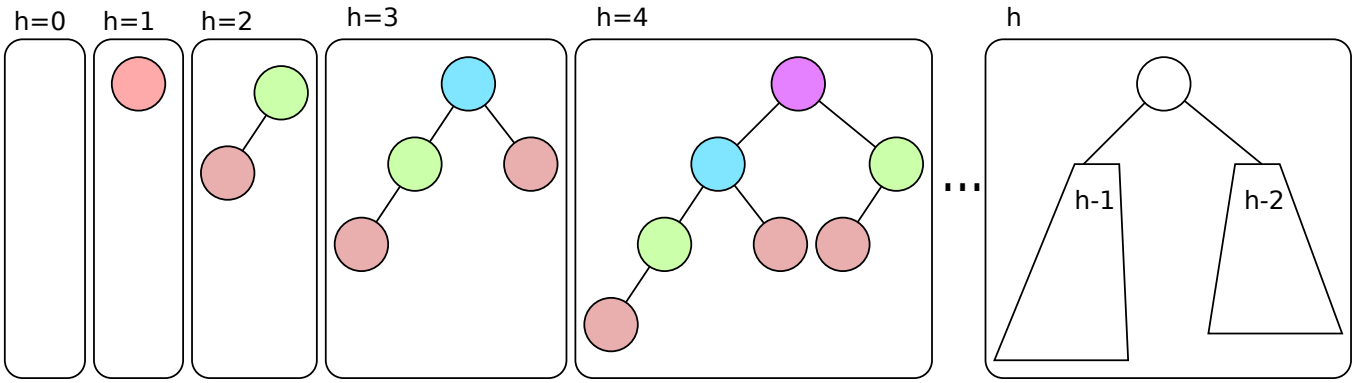


FIGURE 33 – Arbres minimaux d'hauteur 0, 1, 2, 3, ..., h

On peut définir la fonction $n(h)$ comme étant le nombre minimal de nœuds d'un arbre AVL de hauteur h. Cette fonction s'exprime récursivement à l'aide de l'Équation (3).

$$n(h) = \begin{cases} 0 & \text{si } h = 0 \\ 1 & \text{si } h = 1 \\ n(h-1) + n(h-2) + 1 & \text{si } h \geq 2 \end{cases} \quad (3)$$

On peut remarquer que la fonction $n(h)$ est strictement supérieure, pour $h > 1$ à la suite de Fibonacci définie par l'Équation (4).

$$fib(h) = \begin{cases} 0 & \text{si } h = 0 \\ 1 & \text{si } h = 1 \\ n(h-1)+n(h-2) & \text{si } h \geq 2 \end{cases} \quad (4)$$

On peut démontrer que la hauteur maximale d'un arbre AVL de n nœuds est strictement inférieure à :

$$1.44\log_2(n+2) - 0.328 \quad (5)$$

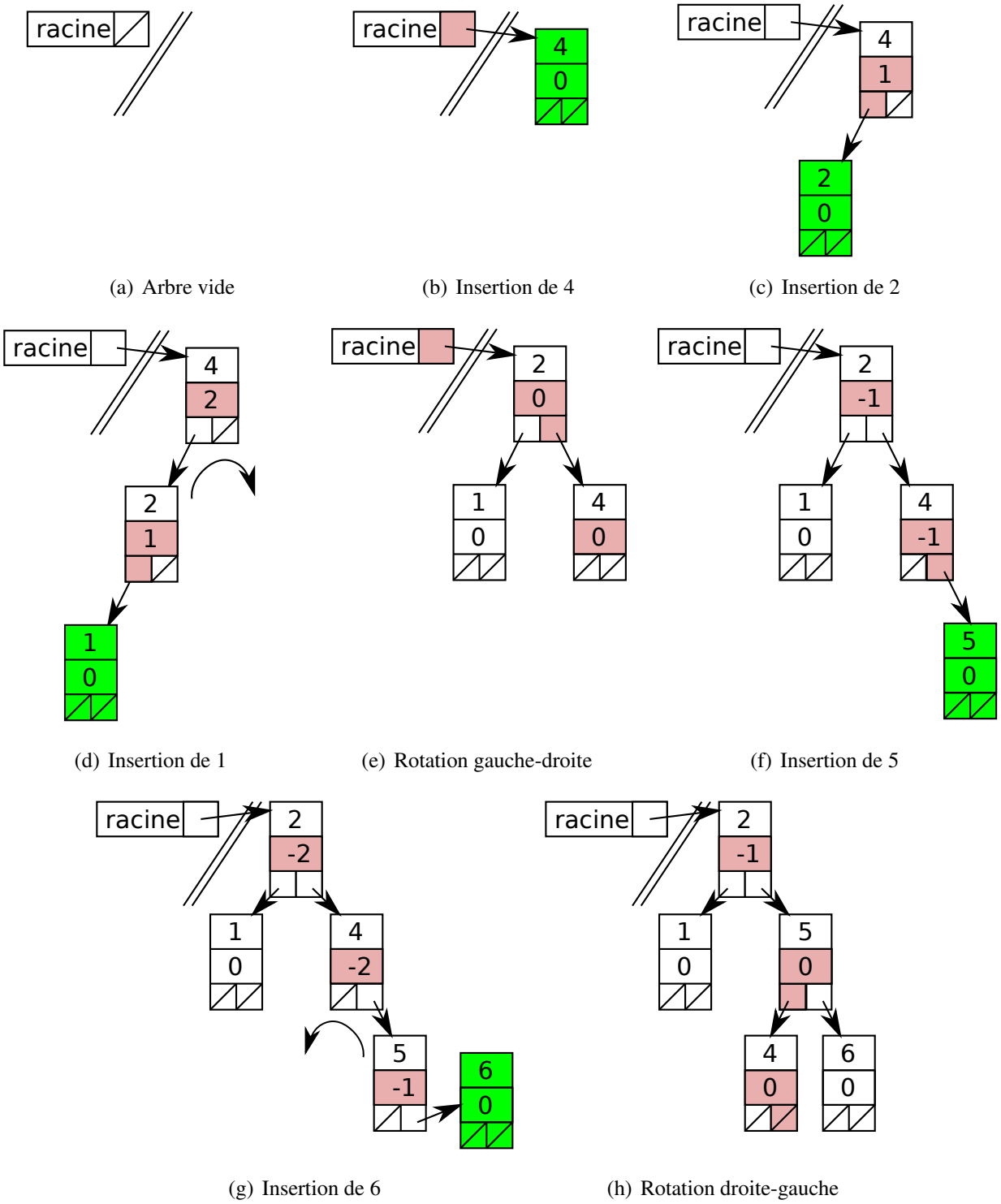
8.9.1 Représentation

Pour détecter quand et comment faire les rotations, on compare la hauteur des sous-arbres de gauche et de droite. Si la différence de hauteur est plus grande que un (1), alors une ou deux rotation(s) sont nécessaires pour rééquilibrer l'arbre. Au lieu de systématiquement recalculer la hauteur des sous-arbres, nous pourrions conserver un champ `hauteur` dans chacun des nœuds. En fait, c'est surtout la différence de hauteur qui nous intéresse. Donc, pour simplifier le code, on conserve **indice d'équilibre** dans chaque nœud en ajoutant un champ `equilibre` dans la classe `Noeud` qui est égale à $h_g - h_d$ (hauteur sous-arbre gauche moins droite).

```
1  template <class T>
2  class ArbreAVL {
3    private:
4      class Noeud{
5        public:
6          Noeud(const T& element);
7          T contenu;
8          Noeud *gauche, *droite;
9          int equilibre;
10     };
11     Noeud* racine;
12     // ...
13   public:
14     // ...
15 };
```

8.9.2 Insertion

La Figure 34 présente cinq insertions à partir d'un arbre AVL vide. Lors de l'insertion d'un nouveau nœud, l'indice d'équilibre est mis à jour pour chacun des nœuds visités dans le chemin menant à la nouvelle feuille. Une insertion peut briser l'équilibre d'un sous-arbre. Par exemple, en (d), suite à l'insertion de 1, le sous-arbre à partir du nœud 4 est déséquilibré. En effet, le sous-arbre de gauche a une hauteur de 2 alors que le sous-arbre de droite a une hauteur de zéro. C'est ainsi que l'indice d'équilibre égale 2. Lorsque l'équilibre est brisé, il est possible de rééquilibrer l'arbre au moyen d'une rotation. Dans le cas (d), une rotation vers la droite est nécessaire. On remonte le nœud 2 et on connecte le nœud 4 à sa droite pour obtenir (e). L'insertion de 6 (g) cause un autre déséquilibre qui est résolu au moyen d'une rotation vers la droite.



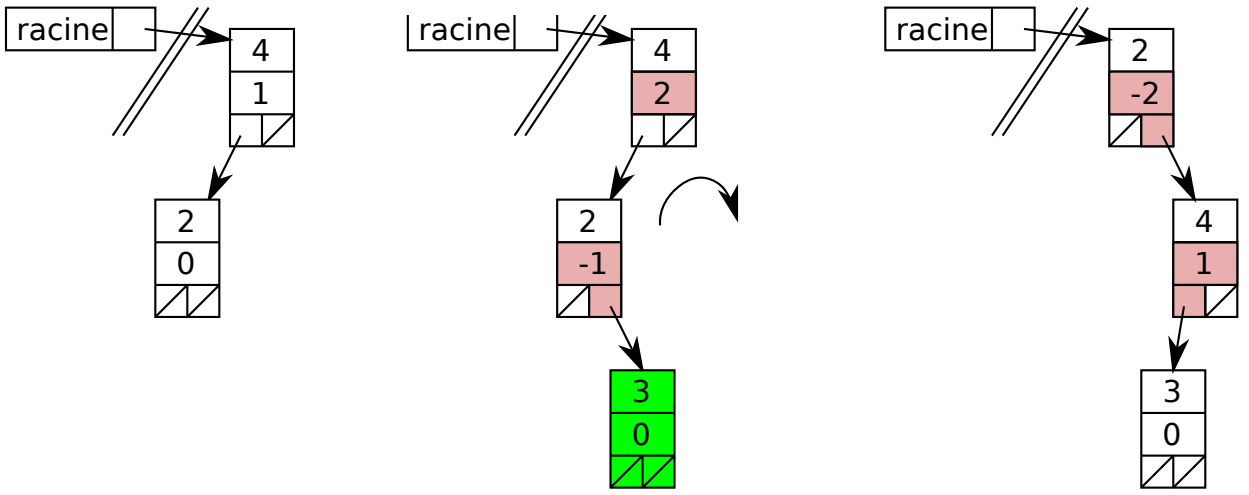
(a) Arbre vide (b) Insertion de 4 (c) Insertion de 2

(d) Insertion de 1 (e) Rotation gauche-droite (f) Insertion de 5

(g) Insertion de 6 (h) Rotation droite-gauche

FIGURE 34 – Insertions dans un arbre AVL. Le vert signifie l'allocation de nœud. Le rouge signifie la mise à jour d'une valeur ou d'un pointeur.

Dans deux cas particuliers, une simple rotation ne permet pas de rééquilibrer l'arbre correctement. La Figure 35 montre un tel exemple où une simple rotation ne fait que changer le sens du déséquilibre.



(a) Arbre initial (b) Insertion de 3, une rotation requise (c) Résultat de la rotation

FIGURE 35 – Cas où une simple rotation ne suffit pas

Pour résoudre ces cas particulier, une **double rotation** est requise. La Figure 36 montre le schéma général où deux rotations sont requises (gauche suivie de droite). Il existe aussi un le cas inverse (droite suivie de gauche). Pour détecter ce cas, il suffit de regarder l'indice d'équilibre du nœud du côté ayant la hauteur la plus élevée. Si la différence est de 3, alors une double rotation est requise.

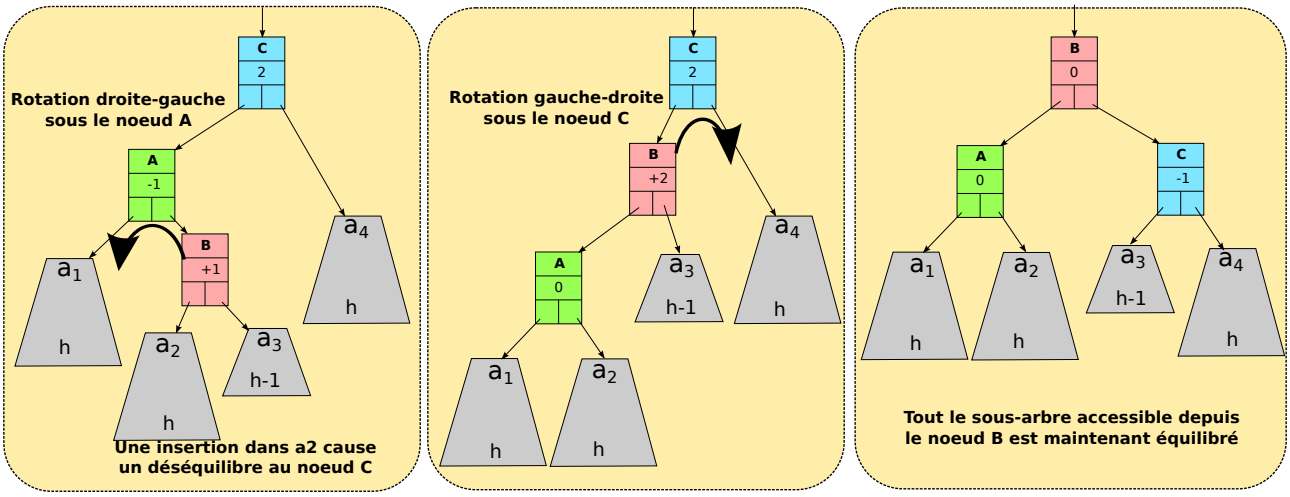


FIGURE 36 – Double rotation

Le code suivant présente une implémentation de l'insertion dans un arbre AVL. La fonction privée `insérer` retourne maintenant un booléen afin d'indiquer si l'insertion a provoqué une augmentation de la hauteur du sous-arbre `n`. C'est en considérant les valeurs retournées que les indices d'équilibre sont mis à jour. On peut aussi remarquer une différence avec la fonction `ArbreRechBin::inserer` (Section 8.7.2) au niveau des comparaisons. On utilise uniquement l'opérateur < pour effectuer les opérations. Cela évite d'avoir à surcharger les opérateur > et == pour la classe `T`.

```
1  template <class T>  // La fonction public
2  void ArbreAVL<T>::inserer(const T& element)
3  {
4      inserer(racine, element);
5  }
6
7  template <class T>  // La fonction private
8  bool ArbreAVL<T>::inserer(Noeud*& n, const T& element)
9  {
10     if(n==NULL){
11         //creation d'une nouvelle feuille
12         n = new Noeud(e);
13         return true; // la hauteur a augmentee dans cette feuille
14     }
15     if(element < n->contenu){
16         // cas de gauche
17         if(inserer(n->gauche, element)){
18             n->equilibre++; // le sous-arbre n->gauche a grandi en hauteur
19             if(n->equilibre==0) return false; // sous-arbre n meme hauteur
20             if(n->equilibre==1) return true; // sous-arbre n a grandi
21             assert(n->equilibre==2); // reequilibrage requis
22             if(n->gauche->equilibre==-1) // cas double rotation
23                 rotationDroiteGauche(n->gauche);
24             rotationGaucheDroite(n); // rotation pour equilibrer n
25         }
26         return false;
27     }else if(n->contenu < element){ // equivalant  if(element>n->contenu)
28         /* le cas de droite est laisse en exercice */
29
30     }else{ // equivalant if(n->contenu == element)
31         n->contenu = element; // ecrasement de la valeur
32         return false; // le sous-arbre n conserve la meme hauteur
33   }
34 }
```

8.9.3 Rotations

La Figure 37 montre le cas général d'une rotation gauche-droite. Deux nœuds sont impliqués dans le rotation : a et b. Le nœud a a les deux sous-arbres A_1 et A_2 tandis que a est le sous-arbre de gauche de b et A_3 est le sous-arbre de droite de b. Les sous-arbres A_1, A_2 et A_3 ont respectivement les hauteurs h_1, h_2 et h_3. À noter que ces hauteurs peuvent être égales ou différentes.

La rotation elle-même est la partie la plus facile. Elle s'effectue par la mise-à-jour de trois (3) pointeurs. Le code suivant montre une ébauche de la rotation gauche-droite.

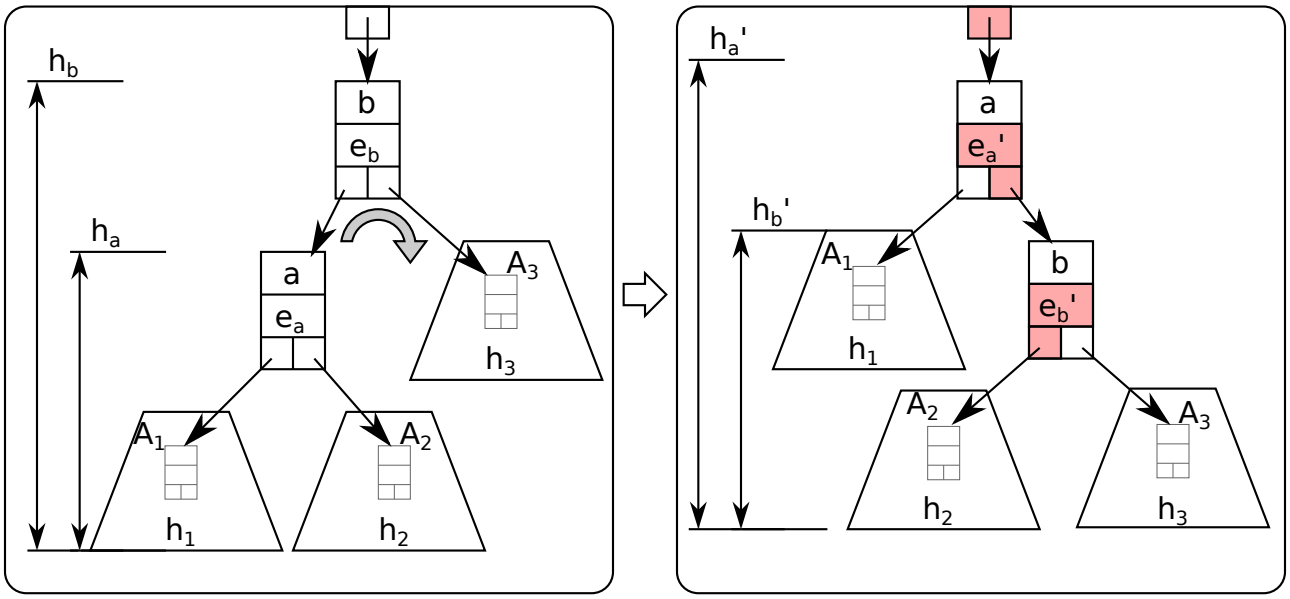


FIGURE 37 – Rotation gauche-droite

```
1  template <class T>
2  void ArbreAVL<T>::rotationGaucheDroite(Noeud*& racinesousarbre){
3    Noeud* a = racinesousarbre->gauche;
4    Noeud* b = racinesousarbre;
5    Noeud* A2 = a->droite;
6    a->droite = b;
7    b->gauche = A2;
8    racinesousarbre = a;
9  }
```

L'opération la plus complexe dans les rotations est la mise-à-jour des indices d'équilibre. Les équations (6) à (14) montrent comment obtenir le nouvel indice e'_b à partir des valeurs de e_a et e_b.

$$
\begin{aligned}
e'_b &= h_2 - h_3 && \text{Par définition.} && (6)\\
h_3 &= h_a - e_b && \text{Par définition.} && (7)\\
h_a &= \max(h_1, h_2) + 1 && \text{Par définition.} && (8)\\
h_1 &= h_2 + e_a && \text{Par définition.} && (9)\\
h_a &= \max((h_2 + e_a), h_2) + 1 && (8),(9) && (10)\\
&= h_2 + \max(e_a, 0) + 1 && (10) && (11)\\
e'_b &= h_2 - (h_a - e_b) && (6)\ (7) && (12)\\
&= h_2 - ((h_2 + \max(e_a, 0) + 1) - e_b) && (12) && (13)\\
&= -max(e_a, 0) - 1 + e_b && (13) && (14)
\end{aligned}
$$

Les équations (15) à (23) montrent comment obtenir le nouvel indice e'_a à partir des valeurs de e_a, e_b et e'_b.

$$
\begin{aligned}
e'_a &= h_1 - h'_b && \text{Par définition.} && (15)\\
h'_b &= \max(h_2, h_3) + 1 && \text{Par définition.} && (16)\\
h_3 &= h_2 - e'_b && \text{Par définition.} && (17)\\
h'_b &= \max(h_2, (h_2 - e'_b)) + 1 && (16), (17) && (18)\\
&= h_2 + \max(0, -e'_b) + 1 && (18) && (19)\\
h_1 &= h_2 + e_a && \text{Par définition.} && (20)\\
e'_a &= (h_2 + e_a) - (h_2 + \max(0, -e'_b) + 1) && (15), (20), (19) && (21)\\
&= e_a - \max(0, -e'_b) - 1 && (21) && (22)\\
&= e_a + \min(0, e'_b) - 1 && (22) && (23)
\end{aligned}
$$

C'est ainsi qu'on obtient le code final suivant.

```
1  template <class T>
2  void ArbreAVL<T>::rotationGaucheDroite(Noeud*& racinesousarbre)
3  {
4      Noeud *a = racinesousarbre->gauche;
5      Noeud *b = racinesousarbre;
6      int  ea = a->equilibre;
7      int  eb = b->equilibre;
8      int  ebp = - (ea > 0 ? ea : 0) - 1 + eb;
9      int  eap = ea + (ebp < 0 ? ebp : 0) - 1;
10
11     a->equilibre = eap;
12     b->equilibre = ebp;
13     b->gauche = a->droite;
14     a->droite = b;
15     racinesousarbre = a;
16 }
```

8.9.4 Enlèvement

Les fonctions d'enlèvement seront présentées en classe.

8.10 Itérateurs d'arbres binaires de recherche

Une opération fréquente sur les ensembles de données, représentées par des arbres binaires de recherche, est d'itérer sur tous les éléments (ou une partie) dans l'ordre. Cela peut se faire à l'aide d'un parcours en **inordre** (voir Section 8.6.1). Revoici le code.

```
1 template <class T> void ArbreBinRech<T>::parcours_inordre(Noeud* n){
2     if(n==NULL) return;
3     parcours_inordre(n->gauche);
4     traiter(n);
5     parcours_inordre(n->droite);
6 }
```

Si on veut faire un traitement particulier sur chaque élément, on pourrait passer en paramètre un pointeur de fonction (une fonction *call back*). Toutefois, cela n'est pas toujours commode. Par exemple, si on veut tester l'équivalence de deux arbres, il faut être capable de les itérer simultanément. Comme cela a été fait pour les listes, nous allons implémenter un type d'**itérateur d'arbre binaire de recherche**.

Un itérateur d'arbre binaire doit en quelque sorte simuler la pile de récursivité d'un parcours en inordre. Fondamentalement, la pile de récursivité du parcours en inordre mémorise les nœuds de niveaux supérieurs. De plus, la pile de récursivité mémorise l'état de progression pour chaque nœud. Une façon imagée de noter l'état de progression dans le code consiste à noter le numéro de ligne pour chaque appel récursif de la fonction `parcours_inordre`. Supposons que l'élément courant est 11 tel qu'indiqué par la flèche rouge dans Figure 38. À ce moment précis, la pile de récursion contient les nœuds 13 (ligne 3), 7 (ligne 5) et 11 (ligne 4). Comme 11 est l'élément courant, cela signifie que le sous-arbre de gauche (9) a été visité.

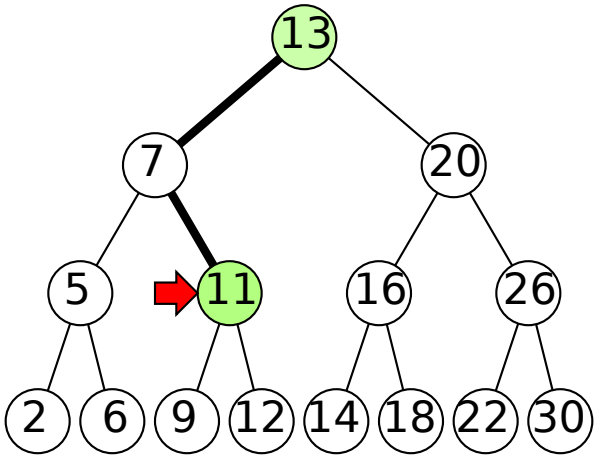


FIGURE 38 – État d'itération d'un parcours en inordre

L'état de progression d'un parcours dans un arbre binaire peut être simplifié. Par exemple, dans la Figure 38, l'élément 7 n'a pas besoin d'être gardé sur la pile, car il n'y a plus de nouveaux enfants à explorer : 5 a déjà été visité, et 11 a déjà été placé sur la pile. Ainsi, on peut simplement garder les nœuds pour lesquels le sous-arbre de droite n'a pas encore été visités (13 et 11).

Le code suivant donne la déclaration de la classe `ArbreAVL::Iterateur`. Un objet `Iterateur` est représenté par un pointeur sur le nœud courant et une pile de pointeurs de nœud formant le chemin menant au nœud courant. Pour des fins de robustesse, un objet `Iterateur` conserve une référence sur l'arbre auquel il est associé. Dans le code, regardez attentivement l'ordre des déclarations. La classe `Iterateur` doit être public, car l'utilisateur doit pouvoir instancier et manipuler des itérateurs. D'ailleurs, plusieurs fonctions retournent un objet de type `Iterateur`. Comme un itérateur contient un pointeur de `Noeud`, la classe `Noeud` doit être déclarée ou annoncée avant `Iterateur`. Toutefois, la classe `Noeud` doit être privée.

```
1  template <class T>
2  class ArbreAVL {
3    public:
4      ArbreAVL();
5      ~ArbreAVL();
6      // ...
7      class Iterateur;
8      Iterateur debut() const;
9      Iterateur fin() const;
10     Iterateur rechercher(const T&) const;
11     Iterateur rechercherEgalOuSuivant(const T&) const;
12     Iterateur rechercherEgalOuPrecedent(const T&) const;
13   private:
14     class Noeud { // ... };
15     Noeud* racine;
16     // ...
17   public:
18     class Iterateur{
19       public:
20         Iterateur(const ArbreAVL& a);
21         Iterateur(const Iterateur& a);
22         operator bool() const;
23         bool operator!() const;
24         bool operator==(const Iterateur&) const;
25         bool operator!=(const Iterateur&) const;
26         const T& operator*() const;
27         Iterateur& operator++(); // pre-increment
28         Iterateur operator++(int); // post-increment
29         Iterateur& operator = (const Iterateur&);
30       private:
31         const ArbreAVL& arbre_associe;
32         Noeud* courant;
33         Pile<Noeud*> chemin;
34         friend class ArbreAVL;
35     };
36
37 };
38
39 template <class T>
40 ArbreAVL<T>::Iterateur::Iterateur(const ArbreAVL& a)
41  : arbre_associe(a), courant(NULL)
42 {
43 }
```

Comme pour les listes, un arbre a plusieurs fonctions pour construire un itérateur. Un itérateur pointant

sur la fin [10] est tout simplement un pointeur `NULL` et une pile vide. À noter que le mot clé `typename` a dû être ajouté pour indiquer au compilateur que ce qui suit `ArbreAVL<T>::...` n'est pas le commencement de la signature du constructeur, mais bien un type. Sans ce mot clé, certains compilateurs pourraient trouver l'expression ambiguë.

```
1  template <class T>
2  typename ArbreAVL<T>::Iterateur ArbreAVL<T>::fin() const {
3      return Iterateur(*this);
4  }
```

La construction d'un itérateur représentant le début d'un arbre est légèrement plus complexe. Pour trouver l'élément le plus petit, il suffit de descendre dans l'arbre en utilisant toujours le sous-arbre de gauche. Voici le code.

```
1  template <class T>
2  typename ArbreAVL<T>::Iterateur ArbreAVL<T>::debut() const {
3      Iterateur iter(*this);
4      iter.courant = racine;
5      if(iter.courant!=NULL)
6          while(iter.courant->gauche!=NULL){
7              iter.chemin.empiler(iter.courant);
8              iter.courant = iter.courant->gauche;
9          }
10     return iter;
11 }
```

L'opérateur ++ (pré-incrément) permet de passer à l'élément suivant. Si l'enfant de droite existe, on descend une fois à droite et on descend encore à gauche jusqu'à une feuille.

```
1  template <class T>
2  typename ArbreAVL<T>::Iterateur& ArbreAVL<T>::Iterateur::operator++() {
3      assert(courant);
4      Noeud* suivant = courant->droite;
5      while(suivant){
6          chemin.empiler(suivant);
7          suivant = suivant->gauche;
8      }
9      if(!chemin.vide())
10         courant = chemin.depiler();
11     else
12         courant = NULL;
13     return *this;
14 }
```

Pour simplifier le code dans les boucles, on peut créer des fonctions auxiliaires. Par exemple, les opérateurs `bool()` et `!()` permettent de tester si l'itérateur courant est rendu la fin.

10. La fin n'est pas le dernier élément, mais la position suivante au dernier élément.

```
template <class T>
ArbreAVL<T>::Iterateur::operator bool() const {
    return courant!=NULL;
}

template <class T>
bool ArbreAVL<T>::Iterateur::operator!() const{
    return courant==NULL;
}
```

Exemple d'utilisation.

```
ArbreAVL<int> arbre;
// ...
for(ArbreAVL<int>::Iterateur iter=arbre.debut();iter;++iter){
   // ...
}
```

8.10.1 Recherche d'un élément

Parfois, on veut itérer à partir d'un élément précis. Ainsi, on définit une fonction `rechercher()` qui retourne un itérateur positionné sur l'élément recherché. Cette fonction s'apparente à la fonction `contient()` vu à la Section 8.7.1, excepté qu'elle retourne un objet itérateur plutôt qu'un booléen.

```
template <class T>
typename ArbreAVL<T>::Iterateur ArbreAVL<T>::rechercher(const T& e)const
{
    Iterateur iter(*this);
    Noeud* n = racine;
    while(n){
        if(e < n->contenu){
            iter.chemin.empiler(n);
            n = n->gauche;
        }
        else if(n->contenu < e)
            n = n->droite;
        else{
            iter.courant = n;
            return iter;
        }
    }
    iter.chemin.vider();
    return iter;
}
```

Pour construire l'itérateur, il suffit d'empiler les nœuds à chaque fois qu'on descend du côté gauche dans l'arbre. On n'empile pas les nœuds pour lesquels on descend à droite. Lorsque l'élément recherché est

trouvé (ligne 14), il suffit de retourner l'itérateur construit. Si l'élément n'est pas trouvé (`n==NULL`), on vide la pile.

8.10.2 Bornes inférieures ou supérieures

Dans certaines situations, on souhaite retourner l'élément précédent ou suivant lorsque l'élément recherché n'existe pas dans l'arbre. Le code suivant montre la fonction `rechercherEgalOuPrecedent`. L'astuce consiste à descendre dans l'arbre pour rechercher l'élément en question. Si on trouve l'élément recherché, il suffit de relancer la recherche avec la fonction `rechercher`. Sinon, l'élément précédent est tout simplement le dernier nœud pour lequel on a descendu par son enfant droite.

```
1  template <class T>
2  typename ArbreAVL<T>::Iterateur
       ArbreAVL<T>::rechercherEgalOuPrecedent(const T& e) const
3  {
4      Noeud* n = racine, *dernier=NULL;
5      while(n){
6          if(e < n->contenu){
7              n = n->gauche;
8          }
9          else if(n->contenu < e){
10             dernier = n;
11             n = n->droite;
12         }else{
13             return rechercher(e);
14         }
15     }
16     if(dernier!=NULL)
17         return rechercher(dernier->contenu);
18     return Iterateur(*this);
19 }
```

L'écriture du code `rechercherEgalOuSuivant()` est laissée en exercice.

8.11 Arbres associatifs (*Mapping Trees*)

Un arbre associatif permet d'implémenter un **dictionnaire**. Un **dictionnaire** est une structure de données permettant d'associer une valeur à une clé. Pour implémenter un dictionnaire, on peut créer une classe `Entree` ayant pour représentation une clé et une valeur. On définit un opérateur < qui compare uniquement la clé. Enfin, un arbre associatif est tout simplement un arbre binaire de recherche (nous avons choisi l'arbre AVL) dont les nœuds sont des objets de type `Entree`.

```
#if !defined(__ARBRE_MAP_H__)
#define __ARBRE_MAP_H__
#include "arbreavl.h"
template <class K, class V>
class ArbreMap {
    class Entree{
      public:
        Entree(const K& c):cle(c),valeur(){}
        K cle;
        V valeur;
        bool operator < (const Entree& e) const { return cle < e.cle; }
    };
    ArbreAVL<Entree> entrees;
  public:
    bool contient(const K&) const;
    void enlever(const K&);
    void vider();
    const V& operator[] (const K&) const;
    V& operator[] (const K&);
};
```

L'opérateur `[]` vient en deux versions.

```
template <class K, class V>
const V& ArbreMap<K,V>::operator[] (const K& c)  const {
  typename ArbreAVL<Entree>::Iterateur iter=entrees.rechercher(c);
  return entrees[iter].valeur;
}
template <class K, class V>
V& ArbreMap<K,V>::operator[] (const K& c)  {
  typename ArbreAVL<Entree>::Iterateur iter=entrees.rechercher(Entree(c));
  if(!iter){
    entrees.inserer(Entree(c));
    iter = entrees.rechercher(c);
  }
  return entrees[iter].valeur;
}
```

Cette implémentation nécessite l'instanciation d'un objet `Entree` pour faire une recherche. Il est possible d'éviter cela avec une autre implémentation où on modifie la classe `ArbreAVL<T>` pour en faire une classe `ArbreMapAVL<K,V>`. La représentation de la classe `Noeud` est adaptée comme suit.

```
1  template <class K, class V>
2  class ArbreMapAVL{
3      // ...
4    private:
5      class Noeud{
6          Noeud(const K& cle_, const V& valeur_);
7          K cle;
8          V valeur;
9          Noeud *gauche, *droite;
10         int equilibre;
11     };
12 };
```

8.12 Arbres rouge-noir

Un arbre rouge-noir est un arbre de recherche équilibré dans lequel les nœuds sont coloriés (de façon abstraite) à l'aide de deux couleurs (rouge et noir). La coloration des nœuds suit les règles suivantes :

- la racine est noire ;
- le nœud parent d'un nœud rouge doit être noir, c'est-à-dire qu'on ne peut pas avoir deux nœuds rouges de suite sur un même chemin ;
- toutes les feuilles sont à une « profondeur noire » égale, la profondeur noire étant définie par le nombre de nœuds noirs sur le chemin de la racine à une feuille.

Dans un arbre rouge-noir, on utilise un objet spécial, nommé **sentinelle**, pour les feuilles. Ces nœuds correspondent aux pointeurs nuls qui marquaient des terminaisons dans un arbre AVL. Dans l'arbre rouge-noir, ces terminaisons sont comptées dans la « profondeur noire », mais pas dans la hauteur de l'arbre. La Figure 39 montre un exemple d'arbre rouge-noir. Comme on peut le voir, toutes les sentinelles `nul` sont à une profondeur noire de 3.

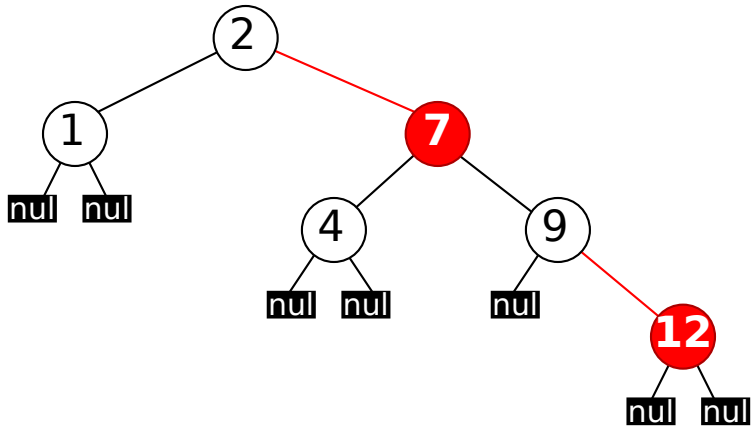


FIGURE 39 – Exemple d'arbre rouge-noir

L'équilibre dans un arbre rouge-noir est obtenu par la notion de profondeur noire combinée au fait qu'on ne peut pas avoir deux nœuds rouges de suite sur un même chemin. Dans le pire cas, une feuille se

retrouve à une profondeur d'au plus du double d'une autre feuille. La feuille la plus profonde emprunte un chemin qui alterne des nœuds rouges et noirs alors que la moins profonde est accessible par un chemin ne comportant aucun nœud rouge. Un arbre rouge-noir contenant n éléments aura une hauteur maximale de $2\log_2 n$.

Tout comme pour les arbres AVL, les opérations d'insertion, de recherche et d'enlèvement dans des arbres rouge-noir peuvent se faire en temps $O(\log n)$. Toutefois, les opérations ont tendance à être plus rapides dans un arbre rouge-noir, car elles nécessitent moins de réorganisation que pour un arbre AVL. Par contre, un arbre rouge-noir peut devenir plus profond qu'un arbre AVL. Ainsi, un arbre AVL est généralement plus rapide pour les opérations de recherche.

8.12.1 Insertion

L'insertion d'un élément commence par l'ajout d'un nouveau nœud rouge au bon endroit. Une fois le nœud inséré, on vérifie si la contrainte interdisant deux nœuds rouges a été violée. Lors d'une violation, c'est-à-dire que le parent du nouveau nœud rouge est aussi rouge, il faut la résoudre en réorganisant l'arbre localement. Les réorganisations se font au cas par cas et du bas vers le haut.

Le premier type de conflit survient lorsque le parent d'un nœud rouge est aussi rouge et qu'il a un frère noir. La Figure 40 montre les 4 cas généraux possibles. Ce type de conflit se résout en réorganisant les nœuds de façon à obtenir un nœud supérieur noir (B) et deux enfants rouges (A et C). Ainsi, la profondeur noire demeure inchangée pour toutes les sentinelles des sous-arbres w, x, y et z.

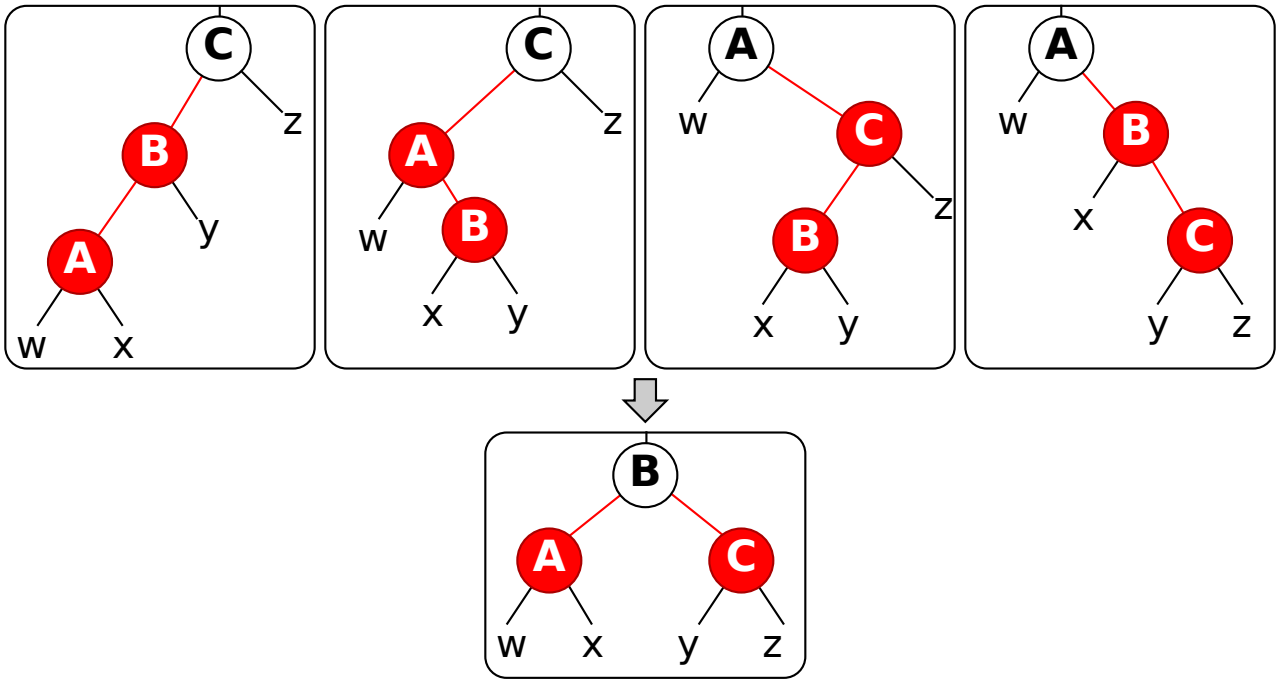


FIGURE 40 – Type 1 : le parent du nouveau nœud rouge est rouge et a un frère noir

Le deuxième type de conflit survient lorsque le parent rouge a un frère rouge. La Figure 41 montre les 4 cas généraux possibles. Pour résoudre ce type de conflit, l'astuce consiste à faire remonter la couleur rouge en inversant localement la couleur sur 2 niveaux. Par exemple, dans le premier cas à gauche, le nœud C devient rouge et les nœuds B et D deviennent noirs. Ainsi, la profondeur noires inchangée pour

toutes les sentinelles des sous-arbres v, w, x, y et z. Il faut noter que cette résolution peut provoquer d'autres conflits rouge-rouge vers le haut. Ainsi, il faut traiter récursivement la détection et la résolution de conflit en remontant dans l'arbre. Si la racine devient rouge, il suffit de la recolorer en noir.

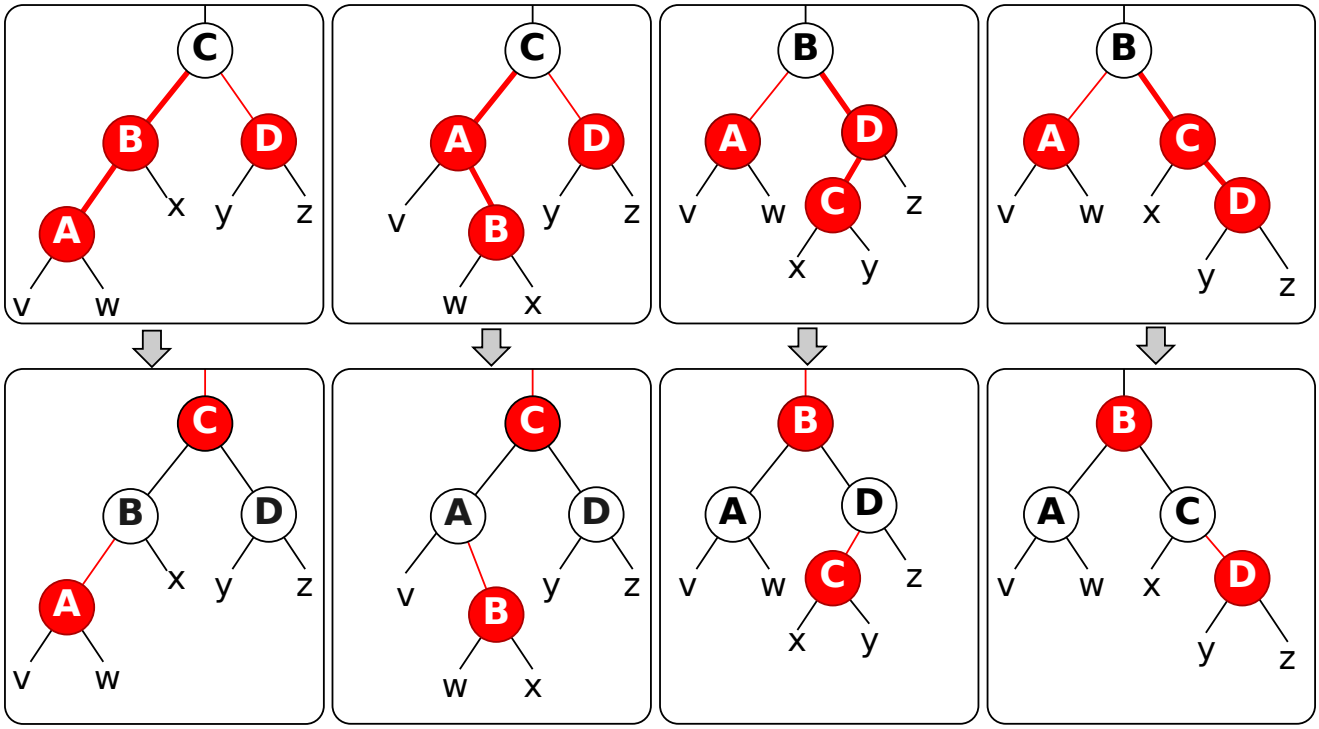


FIGURE 41 – Type 2 : le parent rouge a un frère rouge

Des exemples seront présentés en classe. Une démonstration interactive est accessible à : `http://www.ece.uc.edu/~franco/C321/html/RedBlack/redblack.html`.

8.12.2 Enlèvement

L'enlèvement sera vu en classe.

8.12.3 Représentation

Un peu comme l'indice d'équilibre dans l'arbre AVL, il faut conserver la couleur des nœuds. Cela peut se faire en ajoutant un booléen `rouge` ou un `enum couleur{rouge, noir}` dans une classe `ArbreRN::Noeud`. Pour économiser la mémoire, il est possible de colorer les pointeurs `gauche` et `droite` à l'aide du bit de poids faible. À cause de l'alignement de la mémoire, quelques bits de poids faible sont toujours à zéro.

8.13 Arbres B (*B-Tree*)

8.13.1 Motivation

Les arbres binaires offrent des opérations d'insertion, de recherche et d'enlèvement avec une complexité temporelle de $O(\log n)$. Toutefois, il faut être conscient qu'en pratique, le temps effectif de ces opérations

intègre une constante : $c \log n$. Les arbres B visent à réduire la constante c lorsque :

- l'accès à la mémoire (généralement secondaire) a un temps de latence significatif ;
- et le débit de lecture de la mémoire contigue est élevé.

Un exemple classique, qui réunit ces deux conditions, est l'accès à un disque dur où chaque lecture peut nécessiter un déplacement de la tête de lecture. Ce temps est appelé **temps de latence** ou **temps d'accès**, soit la durée écoulée entre le moment où la requête est donné et le moment de réponse. Par contre, une fois que la tête de lecture déplacée au bon endroit, elle peut potentiellement lire toutes les données sur une piste dans la durée d'une seule rotation du disque. Ainsi, le **débit de lecture**, soit la quantité de données lues par unité de temps, est très élevé.

Ne pas exploiter ces caractéristiques peut avoir un impact négatif sur les performances d'un système. Par exemple, imaginez le service d'authentification de Facebook qui revendique plus de un milliard d'utilisateurs ($n = 10^9$) actifs [11]. Ce service peut être implémenté à l'aide d'un arbre de recherche équilibré qui associe une adresse de courriel (la clé) à un pointeur vers la fiche de l'usager (mot de passe, etc.). Comme cet arbre est immense et doit être persistent, il doit être stocké sur un disque. Le temps total pour effectuer une recherche dans cette arbre est donné par l'Équation 24.

$$t = \log n \times latence\_moyenne + \frac{\log n \times sizeof(noeud)}{debit} \tag{24}$$

Supposons que l'arbre associatif soit un arbre AVL stocké sur un disque dur offrant une latence moyenne de 10 ms et un débit moyen de lecture de 100 Mo/s. Supposons que la taille de la clé (adresse courriel) ait une longueur de 104 octets et que les pointeurs gauche, droite et contenu aient une longueur de 8 octets, soit un total de 128 octets par nœud. Le temps de recherche est donc :

$$\begin{aligned} t &= \lceil 1.44 \log_2 n - 0.328 \rceil \times 10\text{ms} + \frac{\lceil 1.44 \log_2 n - 0.328 \rceil \times 128 octets}{100 \times 10^6 \text{octets/secondes}} \\ t &= 0.3 + 2.34 \times 10^{-9} \\ t &\approx 0.3\text{ms} \end{aligned} \tag{25}$$

Les arbres B offrent une solution au problème de temps de latence. La stratégie consiste à couper l'espace de recherche par un facteur significativement plus grand que deux. Ainsi, au lieu d'avoir des nœuds à 2 enfants, on peut avoir des dizaines ou même des centaines d'enfants. Le nombre d'enfants optimal à utiliser peut être déterminer par la taille d'un bloc contiguë de mémoire pouvant être lu avec un seul accès disque (exemple : un secteur).

8.13.2 Définition

Un **arbre B d'ordre** m, aussi appelé arbre B$(m/2, m)$, [12] est un arbre équilibré ayant les caractéristiques suivantes :

- tous les nœuds ont au plus m enfants ;
- les nœuds intérieurs stockent une liste de clés triées $\langle k_0, k_1, \cdots \rangle$ et des pointeurs vers les enfants ;
- le nombre de clés dans un nœud est égale au nombre d'enfants moins un ;
- tous les nœuds intérieurs (autres que les feuilles et la racine) ont au moins $\lfloor \frac{m}{2} \rfloor$ enfants ;
- toutes les feuilles sont à distance égale de la racine.

11. Ce nombre est probablement exagéré !
12. Il y a plusieurs façons de définir l'ordre des arbres B.

- tous les éléments accessibles depuis le i-ème enfants sont supérieurs à la clé k_{i-1} et inférieurs à la clé k_i.

On peut implémenter deux types de nœuds (Figure 42).

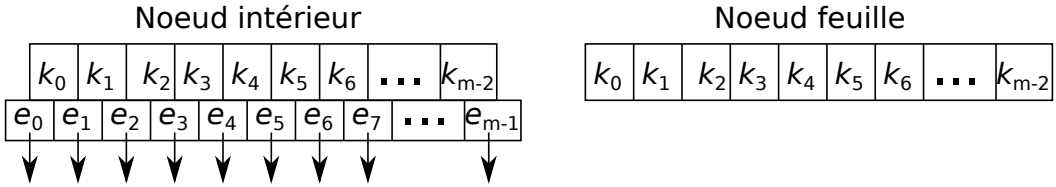


FIGURE 42 – Types de nœuds dans un arbre B

Dans les sections suivantes, les exemples seront illustrés à l'aide d'arbres B d'ordre 5, arbre B(2,5).

8.13.3 Recherche

La recherche d'un élément e dans un arbre B se fait comme suit :

- le nœud de départ est la racine ;
- on recherche la plus grande clé k_i dans le nœud courant tel quel $k_i <= e$;
- si $k_i = e$, alors l'élément a été trouvé ;
- sinon, le i-ème nœud enfant devient le nœud courant et on répète l'étape 2 de la procédure ;
- si on ne trouve pas la clé dans une feuille, alors l'élément e n'existe pas dans l'arbre.

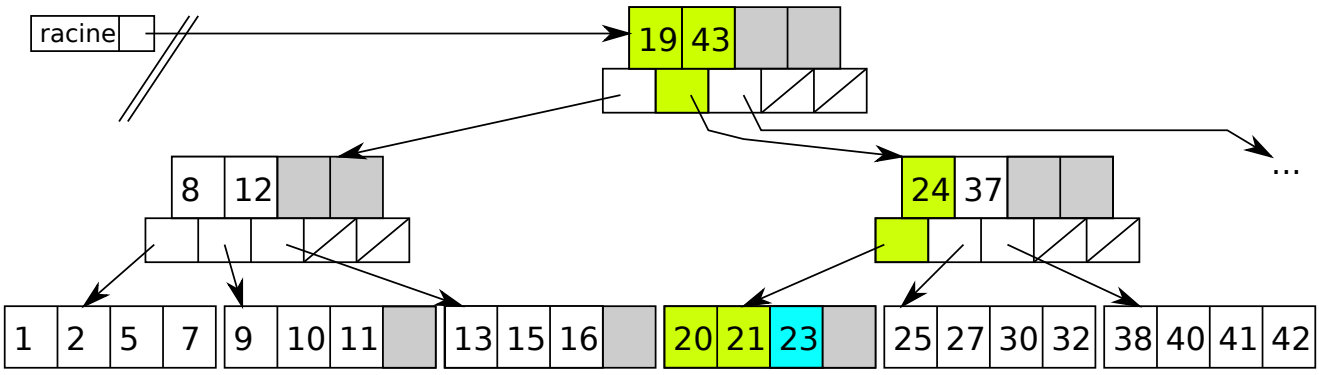


FIGURE 43 – Exemple de recherche de l'élément 23 dans un arbre B

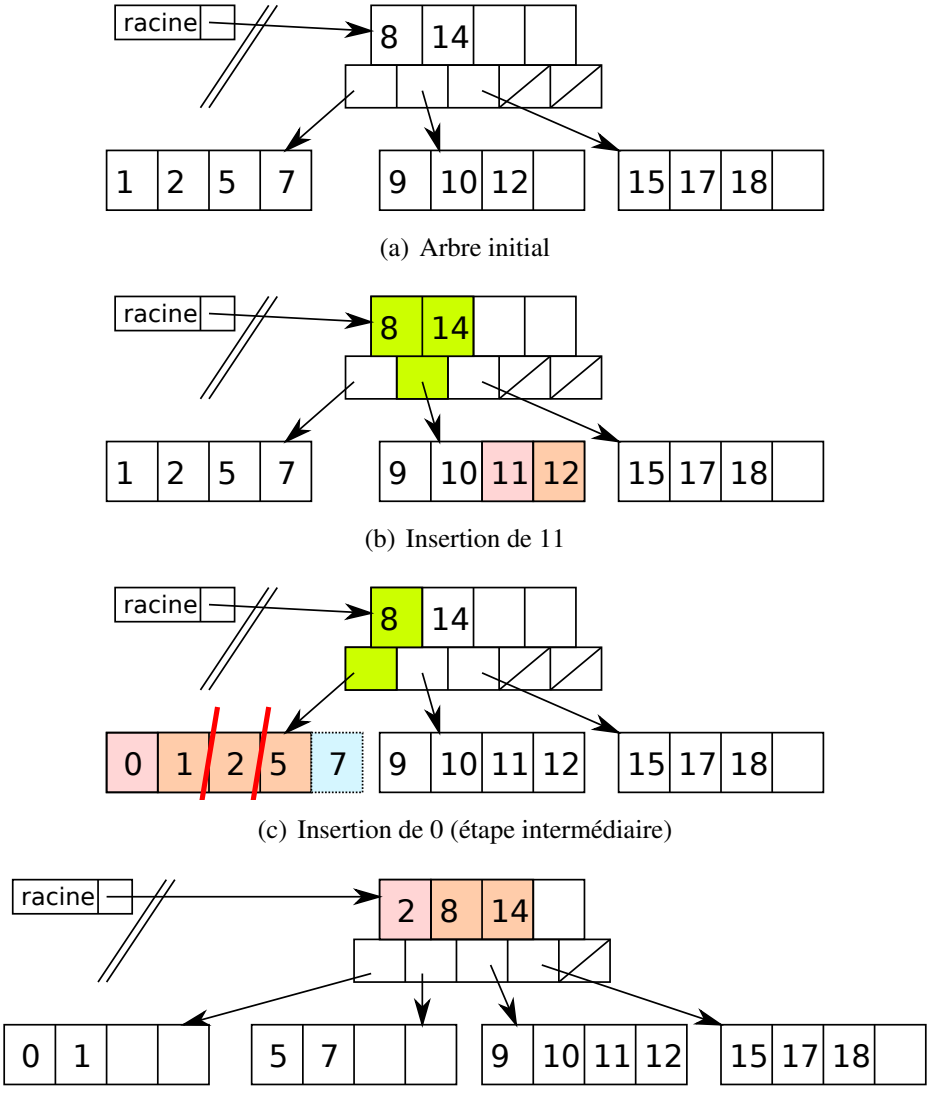
8.13.4 Insertion

Contrairement aux arbres AVL et rouge-noir, les arbres B croient de la racine plutôt que des feuilles. L'insertion de l'élément e fonctionne comme suit :

- on recherche la feuille où devrait se trouver l'élément e ;
- une fois la feuille trouvée, on insère l'élément dans la feuille trouvée ; l'unique exception est l'insertion du premier élément où la racine doit être créée ;

- si la capacité du nœud n'est pas dépassée (un nœud contient au plus $m-1$ clés), l'opération est terminée ;
- si la capacité est dépassée, on scinde le nœud en deux, et on remonte la clé médiane vers le nœud parent ;
- si le nœud parent dépasse sa capacité, on le scinde à nouveau, et ce, jusqu'à temps de remonter à la racine ;
- enfin, dans le cas où la capacité de la racine est dépassée, une nouvelle racine est créée.

La Figure 44 présente un exemple d'insertions dans un arbre B.



(a) Arbre initial

(b) Insertion de 11

(c) Insertion de 0 (étape intermédiaire)

(d) Insertion de 0 (étape finale après scission)

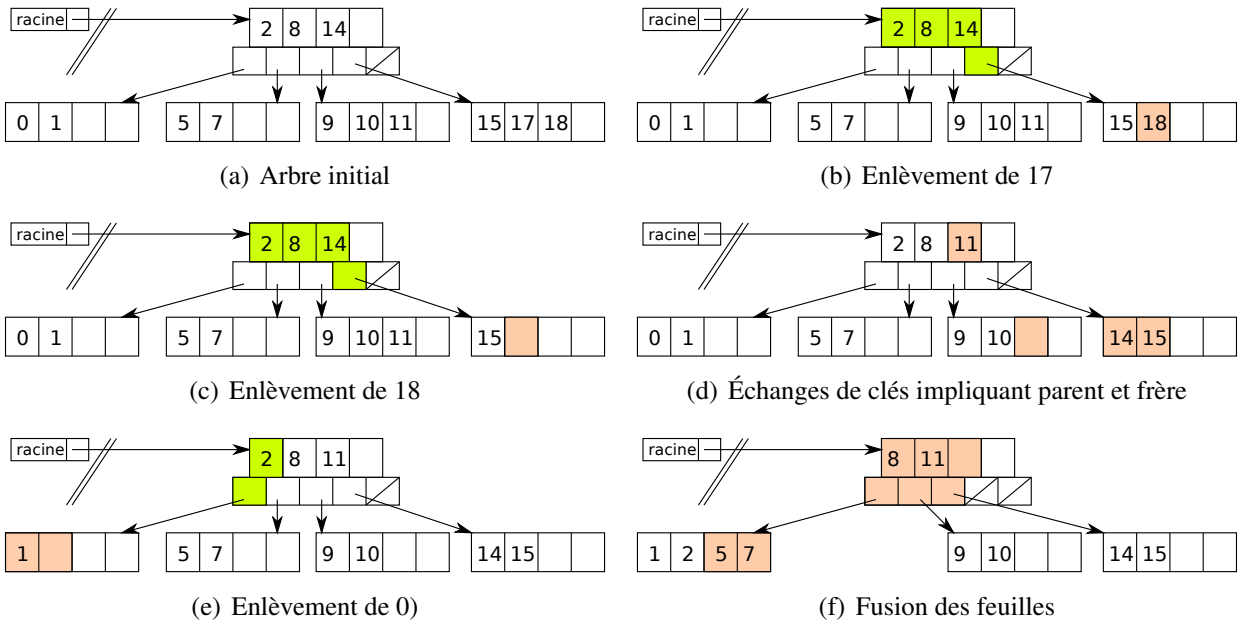
FIGURE 44 – Exemples d'insertions dans un arbre B. Le vert indique la lecture de nœuds. Le orange indique les déplacements. Le rouge indique un nouvel élément. Le bleu indique un débordement de la capacité.

8.13.5 Enlèvement

L'enlèvement d'un élément e fonctionne comme suit :

- on recherche la feuille où se trouve l'élément e et on y enlève e ;
- si le nombre de clés devient inférieur à $\lfloor m/2 \rfloor$:
  - si les deux nœuds frères ont un nombre de clés supérieur à $\lfloor m/2 \rfloor$, on peut emprunter une clé d'un frère ;
  - sinon, on fusionne le nœud avec l'un de ses frères ;
- lors d'une fusion, on descends une clé du nœud parent et on répète l'étape 2 sur ce dernier.

La Figure 45 présente un exemple d'insertions dans un arbre B.



(a) Arbre initial

(b) Enlèvement de 17

(c) Enlèvement de 18

(d) Échanges de clés impliquant parent et frère

(e) Enlèvement de 0)

(f) Fusion des feuilles

FIGURE 45 – Exemples d'enlèvements dans un arbre B. Le vert indique la lecture de nœuds. Le orange indique les déplacements. Le rouge indique un nouvel élément.

8.13.6 Variantes

Arbre B+ : les données attachées aux clés sont exclusivement dans les feuilles. Les clés dans les nœuds internes sont répétées.

Arbre B* : meilleur équilibre en forçant les nœuds à être plein au moins au $2/3$.

8.14 Arbres d'intervalles

Cette partie sera vue en classe uniquement si le temps le permet.

9 Le monceau (*Heap*)

Un **monceau** (***heap***) [13], est une structure de données linéaire et partiellement ordonnée qui permet d'accéder efficacement au plus petit (ou au plus grand) élément d'une collection d'éléments. La structure linéaire d'un monceau, qui ne nécessite aucun pointeur pour lier les éléments entre eux, permet une représentation compacte en mémoire. La principale application du monceau est l'implémentation de **files prioritaires**. Un monceau (*heap*) est donc une structure de données à accès implicite. Les principales opérations sur un monceau sont :

- l'insertion d'un élément ;
- l'accès à l'élément le plus petit (ou plus grand) ;
- la suppression de l'élément le plus petit (ou plus grand).

9.1 Relation avec un arbre binaire complet

Il est possible d'illustrer graphiquement un monceau (*heap*) à l'aide d'un arbre binaire complet équivalent. À titre de rappel, un arbre complet est un arbre où seul le dernier niveau peut être incomplet, et où les éléments du dernier niveau sont à gauche. Un arbre binaire équivalent à un monceau est structuré de la façon suivante :

- l'élément le plus petit (ou plus grand) se trouve dans le nœud racine ;
- les enfants d'un nœud sont plus grands (ou plus petits) que ce dernier ;
- il n'y a pas de contraintes d'ordonnancement entre les enfants de gauche et de droite.

La Figure 46 illustre un monceau sous forme d'arbre binaire.

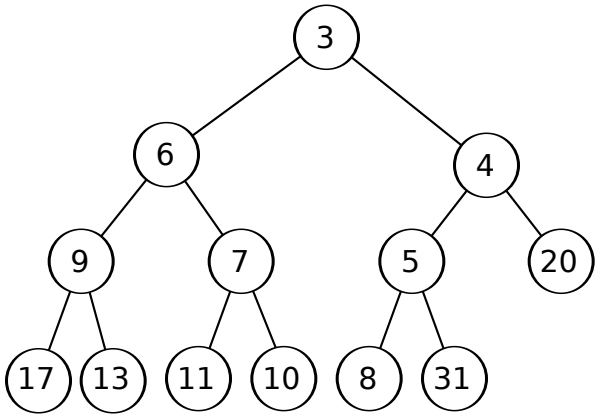


FIGURE 46 – Représentation graphique d'un monceau sous forme d'arbre binaire

9.2 Implémentation et représentation

Un monceau est représenté à l'aide d'un tableau linéaire où :

- la racine se trouve à l'indice zéro (0) ;
- les enfants d'un nœud à l'indice i se trouvent aux indices $2i+1$ et $2i+2$;

13. À ne pas confondre avec le *heap* (tas) pour la mémoire dynamique.

- le parent d'un nœud à l'indice $i > 0$ se trouve à l'indice $\lfloor (i-1)/2 \rfloor$.

La Figure 47 montre la représentation linéaire d'un monceau équivalent à l'arbre binaire de la Figure 46.

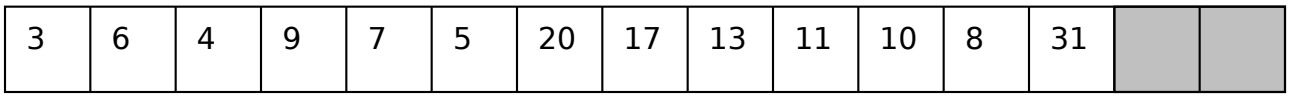


FIGURE 47 – Représentation d'un monceau linéaire sous forme de tableau

Le listing C++ suivant est une ébauche de la déclaration d'une classe générique `Monceau` pour stocker des objets de type `T`. La partie privée déclare la représentation. Un objet de type `Monceau` est représenté à l'aide de la classe générique `Tableau`. Rappel : la classe générique `Tableau<T>` encapsule un tableau natif (*array* C++) d'éléments de type `T`.

```
1  template <class T>
2  class Monceau{
3    private:
4      Tableau<T> valeurs;
5    public:
6      void inserer(const T&);
7      const T& minimum() const;
8      void enleverMinimum();
9      bool estVide() const;
10 };
```

La partie publique déclare l'interface d'utilisation d'un objet de type `Monceau`. Les fonctions publiques permettent d'insérer un nouvel élément, de récupérer l'élément minimum (ou maximum) et d'enlever ce dernier. Il est également possible de vérifier si un monceau est vide.

Pour des raisons d'efficacité et de convivialité, la signature de la fonction publique `enleverMinimum()` et le type de retour peuvent être légèrement modifiés. Par exemple, il peut être convivial de récupérer l'élément minimum en même temps que son enlèvement. Ainsi, la fonction `T enleverMinimum()` pourrait retourner une copie de l'élément minimal enlevé. Toutefois, ce choix est critiquable : pour retourner l'élément minimal, il doit être préalablement copié sur la pile d'exécution. Ensuite, une deuxième copie doit être faite pour l'assignation dans la fonction appelante. Afin d'éviter une copie, il est possible de passer en paramètre une référence à un objet dans lequel on veut copier directement l'élément minimal avant son enlèvement. Ainsi, on obtient la fonction `void enleverMinimum(T&)`.

La récupération de l'élément le plus petit (ou plus grand) est la fonction la plus simple, car il se trouve toujours dans la première position (la racine). Puisqu'un objet `Tableau` est utilisé, il n'est pas nécessaire de tester si le tableau `valeurs` contient au moins un élément. La classe `Tableau` se charge de cette vérification à l'aide des mécanismes d'assertion ou d'exception en C++. Ainsi, on obtient tout simplement le code suivant.

```
1  template <class T>
2  const T& Monceau<T>::minimum(){
3      return valeurs[0];
4  }
```

Plusieurs fonctions de la classe `Monceau` ont besoin d'accéder au parent et aux enfants d'un élément à un indice donné. Pour simplifier l'implémentation de ces fonctions, on peut créer des fonctions auxiliaires privées. Les fonctions `parent()`, `enfant1()` et `enfant2()` prennent en paramètre un indice et retournent respectivement l'indice du parent et des deux enfants.

```
1 template <class T>
2 class Monceau{
3   private:
4     //...
5     inline int parent(int indice) const { return (indice - 1)/2;}
6     inline int enfant1(int indice) const { return 2*indice+1;}
7     inline int enfant2(int indice) const { return 2*indice+2;}
8     //...
9 };
```

9.2.1 Insertion

L'insertion d'un élément dans un monceau fonctionne comme suit. On insère le nouvel élément à la fin du tableau. Cela est équivalent à ajouter une feuille à l'arbre binaire équivalent. Ensuite, on fait remonter l'élément jusqu'au bon endroit. Un élément est au bon endroit lorsqu'il n'est pas plus petit (ou plus grand) que son parent. Pour remonter un élément, il suffit de l'échanger avec son parent. La Figure 48 illustre un exemple où l'élément 2 est inséré dans un monceau existant. L'élément 2 est initialement placé à la fin, soit sous le nœud 20. Ensuite, pour le remonter au bon endroit, on l'échange avec le 20, le 4 et le 3.

Le code suivant définit la fonction `inserer()`. On commence par mémoriser l'indice du nouvel élément inséré qui est égal à la taille du tableau avant l'ajout. Ensuite, on insère l'élément à la fin. Enfin, on appelle la fonction `remonter` qui s'occupe de déplacer l'élément inséré au bon endroit. La Figure 49 montre l'exemple précédent, mais avec la représentation tableauïde du `Monceau`.

```
1 template <class T>
2 void Monceau<T>::inserer(const T& element){
3     int indice = valeurs.taille();
4     valeurs.inserer_fin(element);
5     remonter(indice);
6 }
```

```
1 template <class T>
2 void Monceau<T>::remonter(int indice){
3     if(indice==0) return;
4     int p = parent(indice);
5     if(valeurs[indice]<valeurs[p]){
6         echanger<T>(valeurs[indice], valeurs[p]); // enlevez <T> au besoin
7         remonter(p);
8     }
9 }
```

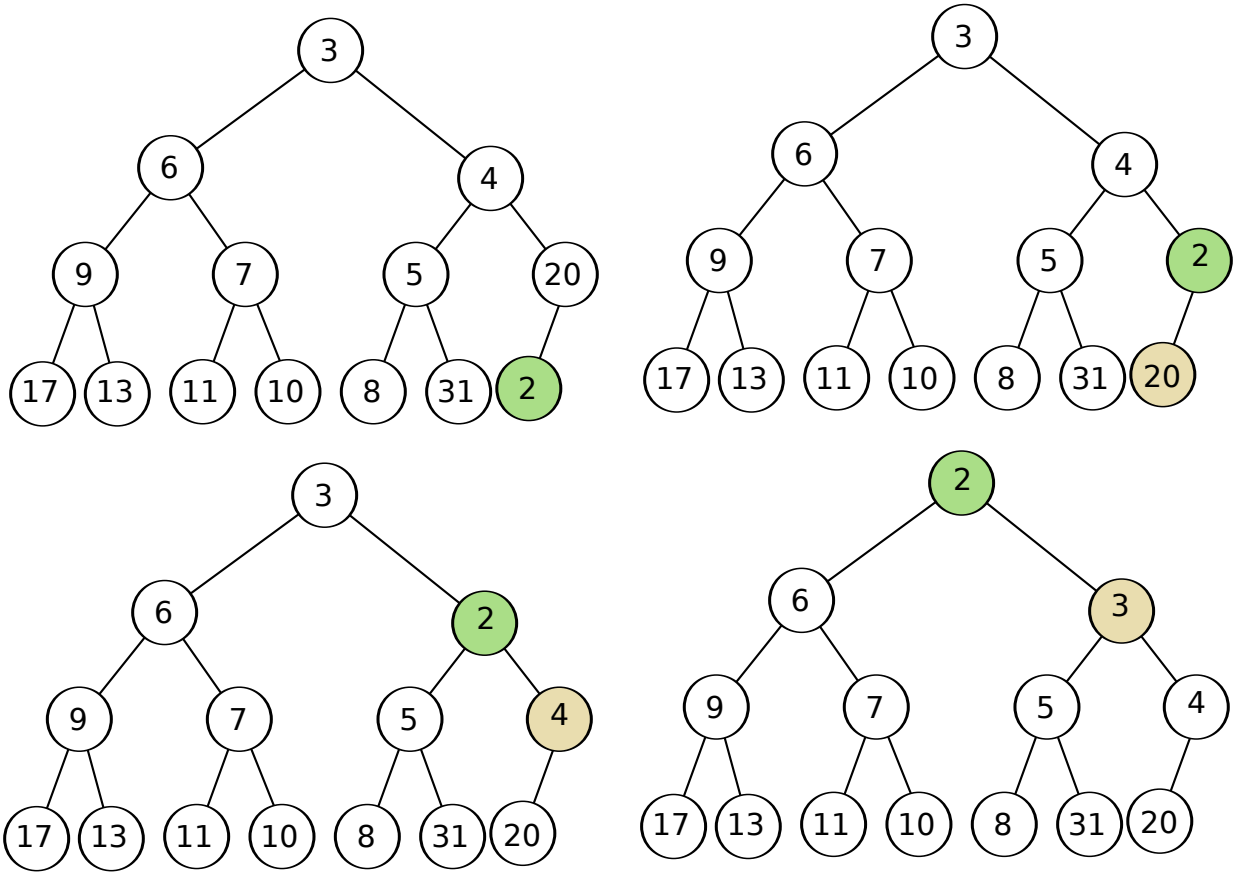


FIGURE 48 – Insertion dans un monceau (*heap*) illustrée à l'aide d'un arbre

3	6	4	9	7	5	20	17	13	11	10	8	31		
3	6	4	9	7	5	20	17	13	11	10	8	31	2	
3	6	4	9	7	5	2	17	13	11	10	8	31	20	
3	6	2	9	7	5	4	17	13	11	10	8	31	20	
2	6	3	9	7	5	4	17	13	11	10	8	31	20	

FIGURE 49 – Insertion dans un monceau (*heap*) illustrée à l'aide d'un tableau

La fonction `remonter()` doit être privée, car elle implémente une opération interne à la classe `Monceau`. Le code ci-dessous présente une version récursive de la fonction `remonter()`. La fonction prend en paramètre l'indice de la position de l'élément qui doit être placé au bon endroit. Tout d'abord, on teste si on est rendu à la racine ; un élément rendu à la racine est implicitement au bon endroit. Par la suite, on place dans la variable *p* l'indice du nœud parent. Si l'élément à remonter est inférieur (ou supérieur)

à l'élément parent, alors on procède à un échange. Par la suite, on doit appeler à nouveau la fonction `remonter` à partir de l'indice *p* afin de continuer de le remonter. À noter qu'une relation d'ordre doit être spécifiée au moyen d'un opérateur < (ou >) pour le type `T`. L'écriture d'une version non récursive de la fonction `remonter()` est laissée en exercice.

La fonction `echanger()` peut être implémentée à l'aide d'une fonction globale suivante.

```
1 template <class T>
2 void echanger(T& a, T& b){
3     T temp = a;
4     a = b;
5     b = temp;
6 }
```

9.2.2 Enlèvement

L'enlèvement de l'élément minimal (ou maximal) dans un monceau fonctionne à l'inverse de l'insertion. On commence par mémoriser le premier élément (la racine). Ensuite, on remplace le premier élément par le dernier élément. Enfin, on descend cet élément jusqu'au bon endroit. Des échanges sont faits pour descendre l'élément. Pour décider dans quel sous-arbre descendre l'élément, il suffit de prendre l'enfant ayant l'élément minimal. La Figure 50 illustre un exemple d'enlèvement de l'élément minimal dans un monceau illustré à l'aide d'un arbre. La Figure 51 présente le même exemple dans un monceau illustré à l'aide d'un tableau.

Le code suivant implémente la fonction `enlever()`. Comme mentionné plus tôt, selon la signature de la fonction et le type retour désirés, il est possible de retourner l'élément enlevé.

```
1 template <class T>
2 void Monceau<T>::enlever(/*T& temp*/){
3     //T temp = valeurs[0]; // à adapter selon la signature/type de retour
4     valeurs[0] = valeurs[valeurs.taille()-1];
5     valeurs.enlever_dernier();
6     descendre(0);
7     //return temp; // à adapter selon la signature/type de retour
8 }
```

Le code suivant présente une version récursive de la fonction privée `descendre()`. Tout d'abord, on affecte l'indice du premier enfant à *suivant*. Si $suivant \geq taille$, alors il faut arrêter, car l'élément à la position `indice` est déjà une feuille. Ensuite, on vérifie si le deuxième enfant ($suivant + 1$) est encore une position valide. Si c'est le cas, on vérifie si l'élément à cette position est inférieur (ou supérieur). On garde l'indice vers l'élément inférieur (ou supérieur). Enfin, on descend l'élément à la position indice si ce dernier doit être descendu. Une fois une descente effectuée, il faut effectuer un autre appel de fonction afin de vérifier s'il faut poursuivre la descente.

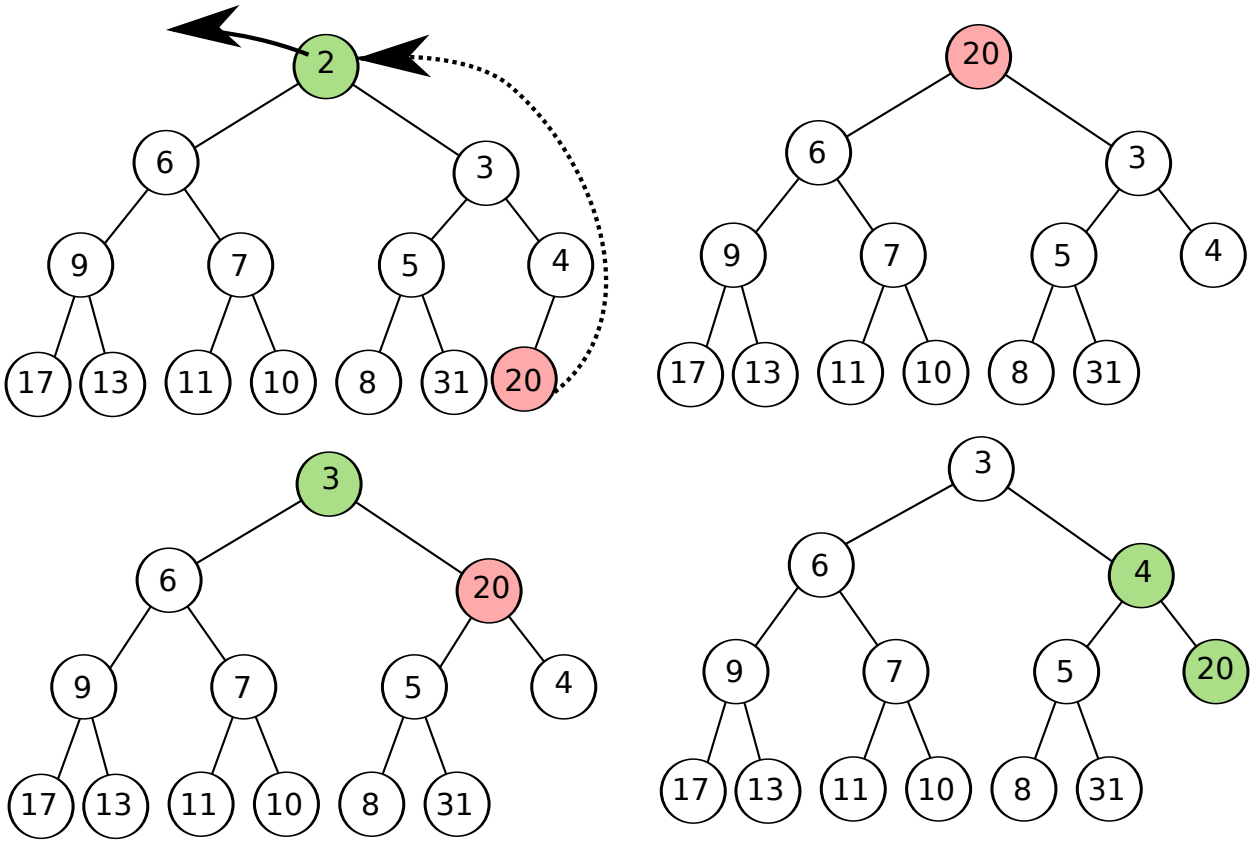


FIGURE 50 – Enlèvement de l'élément minimal dans un monceau illustré à l'aide d'un arbre

2	6	3	9	7	5	4	17	13	11	10	8	31	20	
20	6	3	9	7	5	4	17	13	11	10	8	31		
3	6	20	9	7	5	4	17	13	11	10	8	31		
3	6	4	9	7	5	20	17	13	11	10	8	31		

FIGURE 51 – Enlèvement de l'élément minimal dans un monceau illustré à l'aide d'un tableau

```
1  template <class T> void Monceau<T>::descendre(int indice){
2      int suivant = enfant1(indice);
3      if(suivant>=valeurs.taille()) return;
4      if(suivant+1<valeurs.taille() && valeurs[suivant+1]<valeurs[suivant])
5           suivant++; // suivant=enfant2(indice);
6      if(valeurs[suivant]<valeurs[indice]){
7          echanger<T>(valeurs[suivant],valeurs[indice]);//enlevez <T> si ...
8          descendre(suivant);
9      }
10 }
```

9.3 Analyse et discussion

Quelle est la complexité de l'insertion dans un monceau (*heap*) ? Dans le pire cas, un élément inséré devra être remonté jusqu'à la racine. Ainsi, il faudra faire $O(\log n)$ échanges. Donc, l'**insertion** dans un monceau se fait en temps $O(\log n)$. Avec un raisonnement similaire, on peut conclure que l'enlèvement dans un monceau se fait aussi en temps $O(\log n)$.

Une question très pertinente qu'on pourrait se poser est la suivante : quel serait l'avantage d'utiliser un monceau plutôt qu'un arbre binaire pour implémenter une file d'attente ? Après tout, l'insertion et l'enlèvement se font aussi en temps $O(\log n)$ dans un arbre binaire AVL ou rouge-noir ! Voici quelques pistes de réponse.

Bien que les opérations d'un monceau et d'un arbre binaire aient la même complexité asymptotique, soit $O(\log n)$, le monceau est généralement plus rapide, à une constante près, puisque : (1) il ne fait pas un **ordonnancement total** des éléments et (2) la hauteur d'un arbre binaire complet est généralement moindre que les arbres AVL et rouge-noir. Puisque la recherche n'est pas une opération possible d'un monceau, ses éléments ne sont que **partiellement ordonnés**. Les éléments n'ont pas besoin d'être **totalement ordonnés**. L'ordonnancement partiel d'un monceau permet de sauver quelques opérations (comme les rotations dans un arbre AVL) comparativement à un arbre binaire de recherche.

Un deuxième avantage se trouve dans la compacité de la structure linéaire d'un monceau. Puisque les éléments sont stockés dans un tableau, un monceau ne requiert aucun pointeur pour lier les éléments entre eux (comme les pointeurs gauche et droite dans les noeuds d'arbre binaire de recherche). Cela économise de la mémoire. Toutefois, cet avantage devient négligeable lorsque les éléments à stocker sont de grande taille (beaucoup plus grande qu'un pointeur). De plus, si on ne connaît pas le nombre d'éléments à l'avance, on peut perdre de l'espace dû au redimensionnement automatique du tableau.

Lorsque la taille maximale de la file prioritaire est connue à l'avance, l'usage d'un monceau permet d'éliminer toute opération d'allocation (`new`) ou de libération (`delete`) de mémoire. Bien que nous avions fait l'hypothèse que ces opérations soient en $O(1)$, les opérations de gestion de mémoire sont généralement considérées comme coûteuses. Pour des applications critiques, l'usage d'un monceau offre des garanties de réponse en temps réel lors des insertions et des enlèvements.

En conclusion, le contexte de l'application doit guider le choix entre un monceau et un arbre binaire.

10 Adressage dispersé et Table de hachage (*Hashtable*)

Les tables de hachage (*hashtable*) permettent d'effectuer les opérations d'insertion, d'enlèvement et de recherche en temps quasi constant, soit $O(1)$. Il s'agit d'une amélioration significative par rapport aux arbres binaires de recherche (Section 8.6) qui effectuent ces opérations en temps $O(\log n)$. Pour cette raison, les tables de hachages sont très utilisées pour implémenter des structures de données de type dictionnaire (*map*) contenant un très grand nombre d'entrées. Par exemple, les serveurs de noms de domaine (DNS) peuvent utiliser des tables de hachage afin de traduire efficacement des noms de domaine en adresses IP.

10.1 Motivation

Comment cherchez-vous un mot dans un dictionnaire ? Ou le nom d'une personne dans un bottin téléphonique ? Après avoir étudié les arbres binaires de recherche et la recherche dichotomique dans un tableau trié, vous pourriez être tentés par une recherche dichotomique. Cette technique consiste à couper l'intervalle de recherche par deux à chaque étape. Par exemple, si vous cherchez le mot « structure » dans un dictionnaire de 1024 pages, vous allez commencer par ouvrir le dictionnaire au milieu, soit vers la page 512. En vous apercevant que le mot « structure » vient après la page 512, vous allez encore couper par deux afin de limiter la recherche entre les pages 512 et 768. Cela se répétera jusqu'à ce que le mot soit trouvé. Dans le pire cas, $\log_2(1024) = 10$ pages auront été visitées.

Est-il possible de faire mieux ? Bien sûr que oui ! Comme le mot « structure » commence par la lettre « **S** », soit la 19e lettre de l'alphabet, vous pourriez présumer que ce mot devrait se trouver autour du $19/26^{\text{e}}$ du dictionnaire. Donc, pourquoi ne pas tenter d'aller approximativement vers la page $19 \times 1024/26 = 748$? Une fois cette page ouverte, on avance ou recule jusqu'au bon mot. Évidemment, comme les mots ne sont pas distribués uniformément (davantage de mots commencent par S que par Z), l'expérience aide à estimer la bonne page.

Cette dernière méthode de recherche s'apparente à un **accès direct** dans une table. Fondamentalement, cela revient à trouver à quelle **adresse** (indice) une **clé** est stockée dans une table. Dans les prochaines sections, nous verrons comment réaliser ce type d'accès direct à l'aide d'une table de hachage.

10.2 Accès direct et adressage

Pour trouver en temps constant un élément précis dans un ensemble, il faut être capable de calculer en temps constant une adresse mémoire à partir de sa clé. Cela revient à calculer le **nombre ordinal** (position ou indice) d'un élément dans son ensemble. Pour certains types de données, comme les entiers, cela peut se faire naturellement.

Par exemple, imaginez que vous deviez développer un logiciel pour un club vidéo dans votre quartier. Ce logiciel doit permettre la recherche de fiches de client à partir d'une clé, soit le numéro de téléphone. Pour récupérer la fiche client en temps constant, on peut créer un tableau de la dimension du nombre de numéros de téléphones possibles. Dans le cas de numéros à 7 chiffres, la taille serait de 10^7, soit 10 millions. Le code suivant illustre un exemple avec un tableau de pointeurs.

```
1  class FicheClient{
2  //...
3  };
4  class Base{
5  public:
6      Base(){
7          clients = new FicheClient*[10000000];
8      }
9      //...
10     FicheClient* getClient(int numero) {return  clients[numero];}
11     void ajouterClient(int numero, FicheClient* c){clients[numero]=c;}
12     //...
13 private:
14     FicheClient** clients;
15 };
```

Il n'y a pas que pour les entiers que l'accès direct est possible. Théoriquement, il est possible de le faire pour n'importe quel type de données. Il suffit d'avoir une **relation d'ordre** totale et une fonction pour calculer le **nombre ordinal** (la position) de chaque élément. Par exemple, imaginons des noms d'usager formés de huit lettres minuscules. L'adresse a d'un nom d'usager $n = (l_0, l_1, ..., l_7)$ peut être calculée à l'aide de l'équation 26. Les lettres sont représentées par des nombres de 0 à 25, c'est-à-dire $l_i \in \{0, 1, ..., 25\}$.

$$a = \sum_{i=0}^{7} 26^i \cdot l_i \tag{26}$$

En pratique, les ensembles à mémoriser ont généralement une taille nettement inférieure à la taille du domaine des valeurs possibles. Par exemple, bien qu'il y ait 10 millions de numéros à 7 chiffres, notre club vidéo local n'a peut-être que quelques centaines de clients. Cela est encore plus vrai pour une base de noms d'usager qui contient typiquement entre quelques usagers et quelques millions d'usagers. Pourtant, le domaine des noms d'usager à huit lettres contient $26^8 = 208827064576$ possibilités ! Avec des pointeurs de 64 bits, il faudrait une mémoire de 1.67×10^{12} octets (ou 1555 Go). Ainsi, l'accès direct avec un tel adressage est généralement non souhaitable, ou même irréalisable, car elle nécessite une trop grande quantité de mémoire.

10.3 Adressage réduit

Pour réduire l'utilisation de mémoire, il est possible de réduire la taille de l'espace d'adressage. Ainsi, au lieu d'utiliser un immense tableau pour stocker les valeurs d'un ensemble, on utilise un tableau de plus petite taille. La taille peut avoir le même ordre de grandeur que l'ensemble de valeurs à stocker. Pour savoir à quelle adresse (position) dans le tableau une clé est associée, une **fonction de réduction** est utilisée. Cette dernière utilise une **fraction de la clé** pour calculer une adresse. Généralement, cela se fait à l'aide d'un **modulo** ou d'une **division** arithmétique.

Pour illustrer la réduction d'adressage, reprenons l'exemple de notre club vidéo. Supposons qu'il a 1000 clients. Comment pourrions-nous réduire la taille du tableau requis pour stocker les pointeurs vers les fiches ? Une technique simple est de tronquer la clé en n'utilisant que les 4 derniers chiffres du numéro

de téléphone. Ainsi, au lieu d'avoir 10 millions d'entrées, il n'y en aurait que 10 milles. Une nette amélioration ! Cela revient à calculer le modulo 10000 du numéro de téléphone.

Cette technique fonctionnait plutôt bien à une époque où les numéros de téléphones d'un même quartier avaient de longs préfixes communs, où seuls les 3 ou 4 derniers chiffres changeaient. Cependant, rien ne garantit que chaque client possède un suffixe à 4 chiffres unique. Par exemple, si deux clients ont respectivement les numéros `222-4545` et `333-4545`, la fonction de réduction ne retournera pas une adresse unique ! Cela provoque une **collision**. Une **collision** survient lorsque deux clés différentes génèrent la même adresse. Nous verrons à la Section 10.6 comment gérer ces collisions.

Pour l'exemple des noms d'usager à huit lettres, nous pourrions être tentés de reprendre la même astuce, soit d'utiliser les trois premières ou dernières lettres pour calculer une adresse réduite. Ainsi, la taille du nouvel espace d'adressage sera réduit à $26^3 = 17576$ entrées. Toutefois, utiliser les 3 premières ou dernières lettres n'est généralement pas une bonne idée, car les noms d'usager peuvent être mal distribués dans l'espace des possibilités. Comme les noms d'usager sont souvent basés sur les noms et prénoms des personnes physiques, il risque d'y avoir beaucoup de collisions puisque beaucoup de noms auront des préfixes ou suffixes communs.

10.4 Adressage dispersé

L'utilisation d'une fraction de la clé a tendance à générer beaucoup de collisions lorsque la répartition des éléments n'est pas uniforme. Pour réduire le nombre de collisions potentielles, on utilise plutôt un **adressage dispersé**. Une **fonction de hachage** calcule une **valeur de hachage** à partir de la clé. Cette valeur sera utilisée comme **adresse dispersée**.

Idéalement, une fonction de hachage doit être le plus **chaotique** possible. Une fonction **chaotique** est une fonction dont une très petite variation de l'entrée entraîne une très grande variation de sa sortie. Cette propriété permet de réduire le taux de collisions entre les clés similaires. De plus, il est préférable que la fonction de hachage utilise toute la clé plutôt qu'une fraction de celle-ci.

La conception d'une bonne fonction de hachage dépend de la distribution des éléments. Dans l'exemple club vidéo, si tous les chiffres des numéros de téléphones sont relativement bien distribués, on peut se contenter d'utiliser directement le numéro de téléphone. Sinon, on peut « brasser » les chiffres pour tenter d'éviter les collisions. Pour ce faire, on peut multiplier les chiffres à l'aide de nombres premiers afin d'éviter que des motifs dans les clés soient reproduits dans les adresses. L'équation 27 présente un exemple de fonction de hachage pour convertir un numéro de téléphone n en une adresse dispersée.

$$h = 11 \cdot \lfloor n/10000 \rfloor + n mod 10000 \tag{27}$$

Les chaînes de caractères sont converties de façon similaire. Par exemple, en Java, la valeur de hachage h d'une `String` (chaîne de caractères) $\langle s_0, s_1, s_2, ..., s_{n-1} \rangle$ est calculée de l'aide de l'équation 28.

$$h = \sum_{i=0}^{7} s_i \cdot 31^{n-i-1} \tag{28}$$

10.5 Ébauche d'implémentation

Le code suivant présente une ébauche de déclaration d'une classe générique `EnsembleHache` pour représenter un ensemble d'éléments de type `T`. Pour l'instant, les collisions ne seront pas gérées. La représentation d'un `EnsembleHache` est un tableau linéaire de casiers (***buckets***). Un **casier** (***bucket***) contient un élément de type `T` et un booléen `occupe` indiquant son état d'occupation. L'interface publique de la classe `EnsembleHache` contient trois opérations : l'ajout d'un élément, l'enlèvement d'un élément et la vérification de l'existence d'un élément dans l'ensemble.

```
1  template <class T>
2  class EnsembleHache{
3    private:
4      class Casier{
5        public:
6          Casier() : occupe(false){}
7          T element;
8          bool occupe;
9      };
10     Tableau<Casier> casiers;
11   public:
12     EnsembleHache(int taille=100):casiers(taille){}
13
14     void ajouter(const T&);
15     bool contient(const T&) const;
16     void enlever(const T&);
17 };
```

Le constructeur `EnsembleHache::EnsembleHache(int taille)` crée un tableau de casiers. La `taille` est le nombre de casiers à créer. Généralement, plus la taille est grande, moins il y aura de collisions. Il y a donc un compromis entre l'espace mémoire utilisé et le taux de collisions. Chaque casier est initialisé à l'aide du constructeur sans argument de `EnsembleHache::EnsembleHache::Casier`. Ce dernier initialise le membre `occupe` à `false` afin de marquer que le *casier* est initialement inoccupé.

La fonction `ajouter` est présentée ci-dessous et fonctionne comme suit. Une adresse dispersée est calculée à l'aide de la fonction de hachage et est placée dans la variable *h*. Ensuite, un modulo arithmétique est appliqué afin de réduire cette adresse à l'espace disponible, soit la taille du tableau. Une fois cette nouvelle adresse calculée, on s'assure que le casier soit bien inoccupé. S'il est déjà occupé, il y a collision. Comme les collisions ne sont pas encore gérées, on se contente de générer une erreur pour l'instant. Enfin, l'élément est inséré à son endroit et le casier est marqué occupé.

```
1  template <class T>
2  void EnsembleHache::ajouter(const T& element){
3      int h = element.valeurHachee();
4      int i = h % casiers.taille();
5      assert !casiers[i].occupe;
6      casiers[i].element = element;
7      casiers[i].occupe = true;
8  }
```

La fonction `contient` vérifie la présence d'un élément dans l'ensemble. On commence par calculer l'adresse dispersée `h` et l'adresse réduite `i`. L'élément existe si et seulement si le casier à la position `i` est occupé et contient l'élément à vérifier. À noter qu'il est très important de tester s'il s'agit du même élément puisque deux éléments différents peuvent avoir la même adresse réduite !

```
1 template <class T>
2 bool EnsembleHache::contient(const T& element) const {
3     int h = element.valeurHachee();
4     int i = h % casiers.taille();
5     return casiers[i].occupe && casiers[i].element==element;
6 }
```

La fonction `enlever` ci-dessous calcule une adresse dispersée `h` et une adresse réduite `i`. Optionnellement, selon le comportement souhaité, on peut vérifier si l'élément à supprimer est bien présent dans l'ensemble. Cela aurait pour effet d'empêcher le programmeur d'enlever un élément inexistant. Il s'agit d'un choix d'implémentation non recommandé. Enfin, on marque le casier `i` comme étant inoccupé.

```
1 template <class T>
2 void EnsembleHache::enlever(const T& element){
3     int h = element.valeurHachee();
4     int i = h % casiers.taille();
5     //assert casiers[i].occupe && casiers[i].element==element;
6     casiers[i].occupe = false;
7 }
```

La Figure 52 montre un exemple d'ajouts de A, H, N et V dans un `EnsembleHache`. Ici, la fonction de hachage retourne tout simplement la position de la lettre dans l'alphabet ordonné.

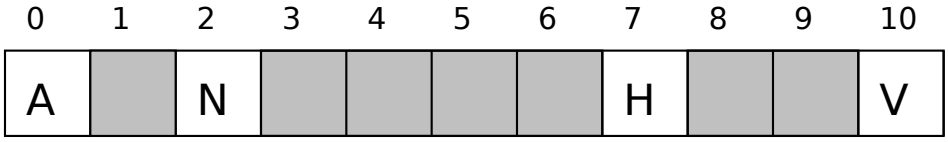


FIGURE 52 – Exemple d'une table de hachage

10.6 Gestion des collisions

Bien que nous puissions espérer qu'une bonne fonction de hachage génère peu de collisions, il est pratiquement impossible de les éviter puisque l'espace d'adressage réduit est nettement inférieur à la taille du domaine des clés possibles.

De plus, même si le nombre de casiers dépasse largement le nombre d'éléments à mémoriser, les collisions demeurent difficiles à éviter. Ce constat peut être illustré à l'aide du paradoxe des dates de naissances dans un groupe de n personnes. Quelle est la probabilité qu'au moins deux personnes d'un groupe de 25 aient la même date d'anniversaire ? On pourrait penser que c'est très peu puisque $25 << 365$. La réalité est qu'il y a 56.8 % de chances qu'au moins 2 personnes aient la même date d'anniversaire. Et ce nombre dépasse 99 % à partir de 60 personnes ! Cet exemple devrait vous convaincre que les collisions

sont pratiquement inévitables. Heureusement, il existe plusieurs stratégies pour gérer les collisions. Les plus fréquentes sont présentées ici.

10.6.1 Adressage ouvert

L'adressage ouvert consiste à résoudre une collision en utilisant une autre entrée (adresse alternative). L'adresse alternative peut être calculée de façon linéaire, quadratique ou à l'aide d'une seconde fonction de hachage.

Deux méthodes de résolution seront illustrées à l'aide d'un exemple qui consiste à ajouter les lettres suivante dans un `EnsembleHash` : A, N, V, L, W, H et K. La fonction de hachage utilisée retourne la position de la lettre dans l'alphabet. Les adresses dispersées sont : A(0), N(13), V(21), L(11), W(22), H(7) et K(10).

La Figure 53 illustre les étapes d'insertion à l'aide d'une stratégie ouverte avec résolution **linéaire**. Puisque le tableau contient 11 casiers, on utilise la fonction de réduction $modulo 11$. On obtient ainsi les adresses réduites suivantes : A(0), N(2), V(10), L(0), W(0), H(7) et K(10). Le résultat à la fin de chaque étape est présenté sur une ligne. Les casiers grisés sont ceux inoccupés. Ceux en rouges indiquent les tentatives d'insertion suite à une collision. La case verte indique le casier où l'élément a été inséré. Les lettres A, N et V sont respectivement insérées dans les casiers $0 mod 11 = 0$, $13 mod 11 = 2$ et $21 mod 11 = 10$. La première collision survient à l'insertion de L, car le casier 0 contient déjà A. Avec la résolution linéaire, on place L dans le prochain casier disponible (1). De façon similaire, W ne peut être inséré dans le casier 0. Il faut alors le placer dans le premier disponible, soit le casier 3. La lettre H va directement dans le casier 7. Enfin, la lettre K est placée dans le casier 4 après avoir fait des tentatives dans les casiers 10, 0, 1, 2 et 3.

0	1	2	3	4	5	6	7	8	9	10
A										
A		N								
A		N								V
A	L	N								V
A	L	N	W							V
A	L	N	W				H			V
A	L	N	W	K			H			V

FIGURE 53 – Exemple avec une gestion de collision linéaire

La Figure 54 illustre le résultat d'une résolution **quadratique**. Avec cette résolution, une équation quadratique est utilisée pour déterminer les tentatives. Ici, on utilise la formule i^2 où ii est représente le numéro de tentative, soit : $0, 1, 4, 9, 16, \cdots$. La première différence avec la résolution linéaire survient à l'insertion de W. Après l'échec dans les casiers 0 et 1, la lettre W est placée dans le casier 4. La résolution quadratique a pour avantage d'être davantage aux fonctions de hachages imparfaites qui généreraient des valeurs concentrées dans certaines régions de l'espace d'adressage.

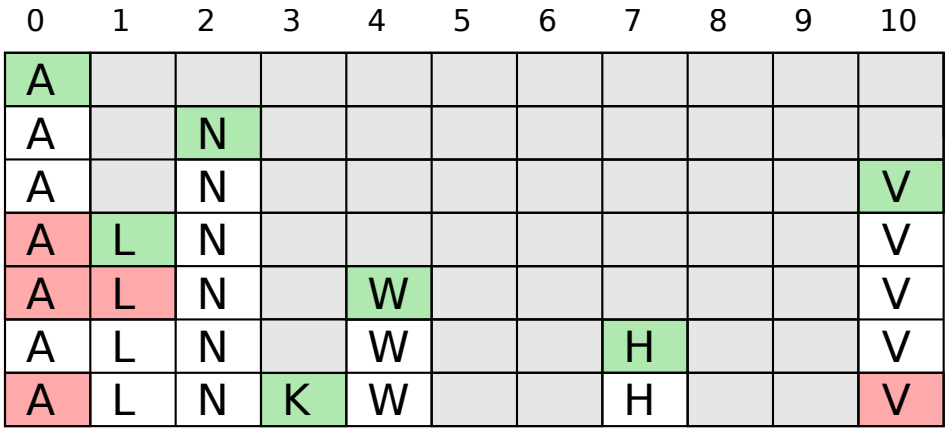


FIGURE 54 – Exemple avec une gestion de collision quadratique

Une autre technique de résolution consiste à utiliser une deuxième fonction de hachage. Toutefois, celle-ci à elle seule n'est pas suffisante. Elle doit donc être combinée à une autre technique de résolution, comme les résolutions linéaires et quadratiques.

10.6.2 Chaînage à l'aide d'une deuxième structure

Une autre stratégie de gestion de collisions consiste à utiliser une deuxième structure pour stocker plusieurs objets dans une même entrée (adresse réduite). La Figure 55 illustre un exemple avec une structure linéaire telle une liste chaînée ou un tableau.

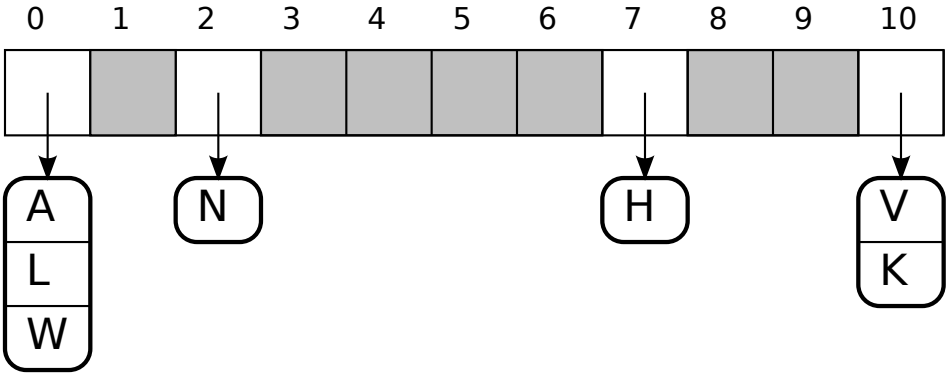


FIGURE 55 – Exemple d'une table de hachage avec chaînage

Le code suivant révise la représentation de la classe `EnsembleHache::Casier`.

```
1  template <class T>
2  class EnsembleHache{
3    private:
4      class Casier{
5        public:
6          Liste<T> entrees;
7      };
8      Tableau<Casier> casiers;
9    public:
10 //...
```

La Figure 56 illustre un exemple avec un arbre AVL.

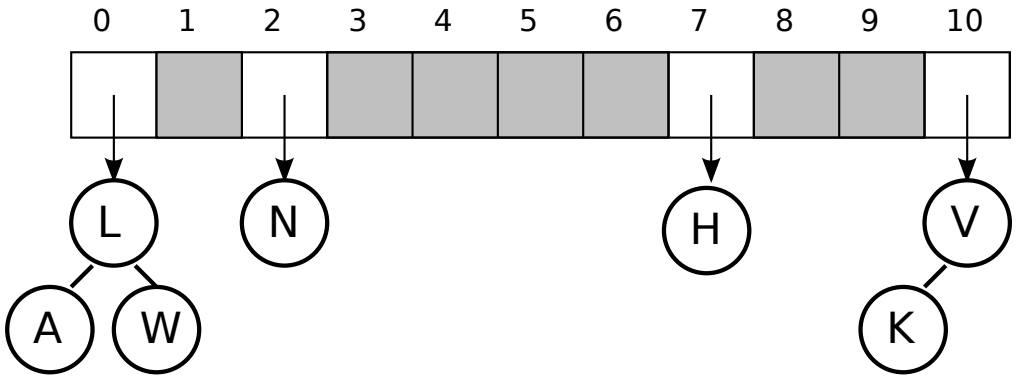


FIGURE 56 – Exemple d'une table de hachage avec arbres binaires

Le code suivant révise la représentation de la classe `EnsembleHache::Casier`. Le code des fonctions est laissé en exercice.

```
1  template <class T>
2  class EnsembleHache{
3    private:
4      class Casier{
5        public:
6          ArbreAVL<T> entrees;
7      };
8      Tableau<Casier> casiers;
9    public:
10 //...
```

10.7 Augmentation de la taille variable du tableau

La performance d'une table de hachage décroit en fonction du nombre de collisions. Le nombre de collisions dépend du **taux d'occupation**, soit le rapport nombre d'éléments sur la taille du tableau (nombre de casiers disponibles). Ce taux est aussi appelé **facteur de chargement** (*load factor*). Comme on ne

connaît pas toujours le nombre d'éléments à stocker à l'avance, une technique fréquente consiste à commencer avec un tableau de petite taille et à l'augmenter à chaque fois qu'une limite sur le facteur de chargement est atteinte. Pour éviter de redimensionner le tableau trop souvent, on peut réutiliser la même stratégie que pour les tableaux génériques, qui consiste à doubler la taille du tableau à chaque fois.

10.8 Table de hachage associative (dictionnaire)

Il est possible de créer une table de hachage associative afin d'associer des valeurs à des clés. Le code suivant donne l'ébauche de la déclaration d'une classe `DictionnaireHache`.

```
1  template <class K, V>
2  class DictionnaireHache{
3    private:
4      class Casier{
5        public:
6          ListeMap<K, V> entrees;
7      };
8      Tableau<Casier> casiers;
9    public:
10 //...
```

10.9 Analyse et discussion

En pratique, l'adressage dispersé offre généralement de bonnes performances. Les opérations se font pratiquement en $O(1)$. Toutefois, une performance d'accès en temps $O(1)$ n'est pas toujours garantie. Dans les faits, la performance d'une table de hachage dépend de la fonction de hachage utilisée. Comme les fonctions de hachage sont généralement conçues de façon empirique, elle n'offrent pas de garanties théoriques pour disperser uniformément les éléments. Si une fonction s'avère peu chaotique, elle pourrait entraîner un haut taux de collisions. Dans le pire cas, la fonction de hachage placera tous les éléments dans un même casier. Ainsi, les gains de performance liés à l'adressage dispersés seront complètement annulés.

La performance dans le pire cas dépend de la gestion des collisions. Avec un adressage ouvert et avec un chaînage avec une liste chaînée, les opérations de recherche et d'enlèvement se feront en temps $O(n)$. Cela s'avère pire que $O(\log n)$ offert par les arbres binaires de recherche. Au mieux, avec une stratégie de chaînage avec un arbre binaire, la complexité temporelle sera de $O(\log n)$, et ce, même dans le pire cas.

Quatrième partie

Bibliothèques normalisées

11 Bibliothèques normalisées

Les structures de données linéaires (Partie II) et avancées (Partie III) sont généralement disponibles dans des bibliothèques normalisées. Dans cette section, les bibliothèques normalisées suivantes seront introduites :

- la *Standard Template Library*, ou simplement la STL en C++ ;
- la Java Collection dans l'API standard de Java.

Seule la première la STL sera approfondie.

Le Tableau 4 suivant présente les équivalences des principaux types de structure d'une bibliothèque à l'autre. Les noms dépréciés sont rayés (en date de juin 2013).

TABLE 4 – Équivalences des structures entre Lib3105++, STL et Java

Structure (concept)	Lib3105++	STL C++	Java Collection
Pile	`Pile`	`stack`	`Stack`
File	`File`	`deque`	`LinkedList`, `ArrayQueue`
Tableau dynamique	`Tableau`	`vector`	~~`Vector`~~ `ArrayList`
Liste chaînée	`Liste`	`forward_list`	–
Liste double chaînée	–	`list`	`LinkedList`
Ensemble ordonné	`ArbreAVL`	`set`	`TreeSet`
Dictionnaire	`ArbreAVLMap`	`map`	`TreeMap`
Ensemble avec doublons	–	`multiset`	–
Dictionnaire avec doublons	–	`multimap`	–
File prioritaire	`Monceau`	`priority_queue`	`PriorityQueue`
Ensemble non ordonné	–	~~`hash_set`~~ `unordered_hash`	`HashSet`
Table de hachage	`TableHache`	~~`hash_map`~~ `unordered_map`	~~`Hashtable`~~ `HashMap`

11.1 *Standard Template Library (STL)* en C++

La présente section n'est qu'une brève introduction à la STL. Pour une référence complète, référez-vous à `http://www.sgi.com/tech/stl/` ou `http://en.cppreference.com/w/cpp/container`.

La STL a été initialement créé par Alexander Stepanov qui a travaillé aux *AT&T Bell Labs* et *Hewlett-Packard Reseach Labs*. La STL a été disponible sous plusieurs implémentations, dont la STL originale chez Hewlett-Packard, une version chez Silicon Graphics (SGI), etc. Heureusement, la STL influencé la *Standard C++ Library* [14] qui inclut maintenant la majorité des fonctionnalités de la STL. De plus, beaucoup d'efforts ont été fait pour améliorer la comptabilité entre compilateurs.

14. Norme ISO C++.

11.1.1 Concepts

La STL introduit des concepts. Dans la terminologie STL, un **concept** est une abstraction ou une interface. Cependant, contrairement à d'autres bibliothèques, les concepts n'existent pas dans le code C++ de la STL. Par exemple, il n'y a pas de super classe `container` au-dessus de `vector`, `list`. Il appartient à l'utilisateur de la STL (le programmeur) d'utiliser les structures correctement.

Un **conteneur** est un concept d'objet ayant pour fonctionnalité de stocker d'autres objets. Les principaux conteneurs sont : `stack`, `deque`, `vector`, `list`, `forward_list`, `set`, `multiset` et `priority_queue`. Selon le type de conteneur, les objets peuvent être ordonnés ou non ordonnés. De façon similaire à ce que nous avons fait jusqu'à présent dans ce cours, lorsqu'on utilise un conteneur, il faut donner le type de ses éléments. Ce type doit être **assignable**. Le concept **assignable** signifie que l'opérateur d'assignation (=) doit existé pour le type utilisé.

Les conteneurs ont été conçus dans un souci d'uniformité. Ils offrent tous les fonctions de base suivantes :

a.begin()	Retourne un itérateur sur le début.
a.end()	Retourne un itérateur sur la fin (position après le dernier élément)
a.size()	Retourne la taille, soit le nombre d'éléments dans le conteneur.
a.empty()	Retourne vrai si le conteneur est vide.
a.max_size()	Retourne le nombre maximal d'éléments supporté par le conteneur.
a.swap(b)	Échange de façon efficace le contenu des conteneurs a et b.

Toutes ces opérations s'exécutent en $O(1)$ amorti.

Un **conteneur associatif** est un concept de dictionnaire où on peut associer une **valeur** à une **clé**. Un conteneur associatif peut être **unique** ou **multiple**. Les principaux conteneurs associatifs sont `map` et `unordered_map`.

Le concept d'**itérateur** est utilisé pour pointer ou référencer **de façon abstraite** un élément dans un conteneur. Les itérateurs de la STL sont conçus de façon à être utilisés de la même façon que des indices sur des tableaux linéaire natifs en C/C++. La STL introduit les concepts d'itérateur suivants.

- Un **Trivial Iterator** est un itérateur n'offrant pas de possibilité de déplacement. Il offre le déréférencement à l'aide de l'opérateur `*`.
- Un **Forward Iterator** est un itérateur unidirectionnel pour lequel les opérateurs préincrément et postincrément `++` existent ;
- un **Bidirectional Iterator** est un itérateur bidirectionnel pour lequel les opérateurs pré/post `++` et `--` existent ;
- un **Random Access Iterator** est offrant la possibilité de se déplacer n'importe où. Ils supportent les opérateurs `++`, `--`, `+=`, `-=` et `[]`.

11.1.2 Exemple d'utilisation de `std::vector`

```
1 #include <vector>
2 int main(){
3   std::vector<int> v;
4   a.push_back(1);
5   a.push_back(2);
6   a.push_back(3);
7 }
```

11.1.3 Exemple d'utilisation de `std::set`

```
1 #include <set>
2 int main(){
3   std::set<int> a;
4   a.insert(1);
5   a.insert(2);
6   a.insert(3);
7 }
```

11.1.4 Algorithmes de la STL

La STL offre plusieurs algorithmes regroupés en plusieures catégories :

- algorithmes non mutables : `for_each`, `find`, `find_if`, `count`, etc. ;
- algorithmes mutables : `copy`, `fill`, `swap`, `replace`, etc. ;
- algorithmes de tri ;
- algorithmes numériques.

11.2 Java Collection de l'API standard de Java

Référence : `http://docs.oracle.com/javase/7/docs/api/`.

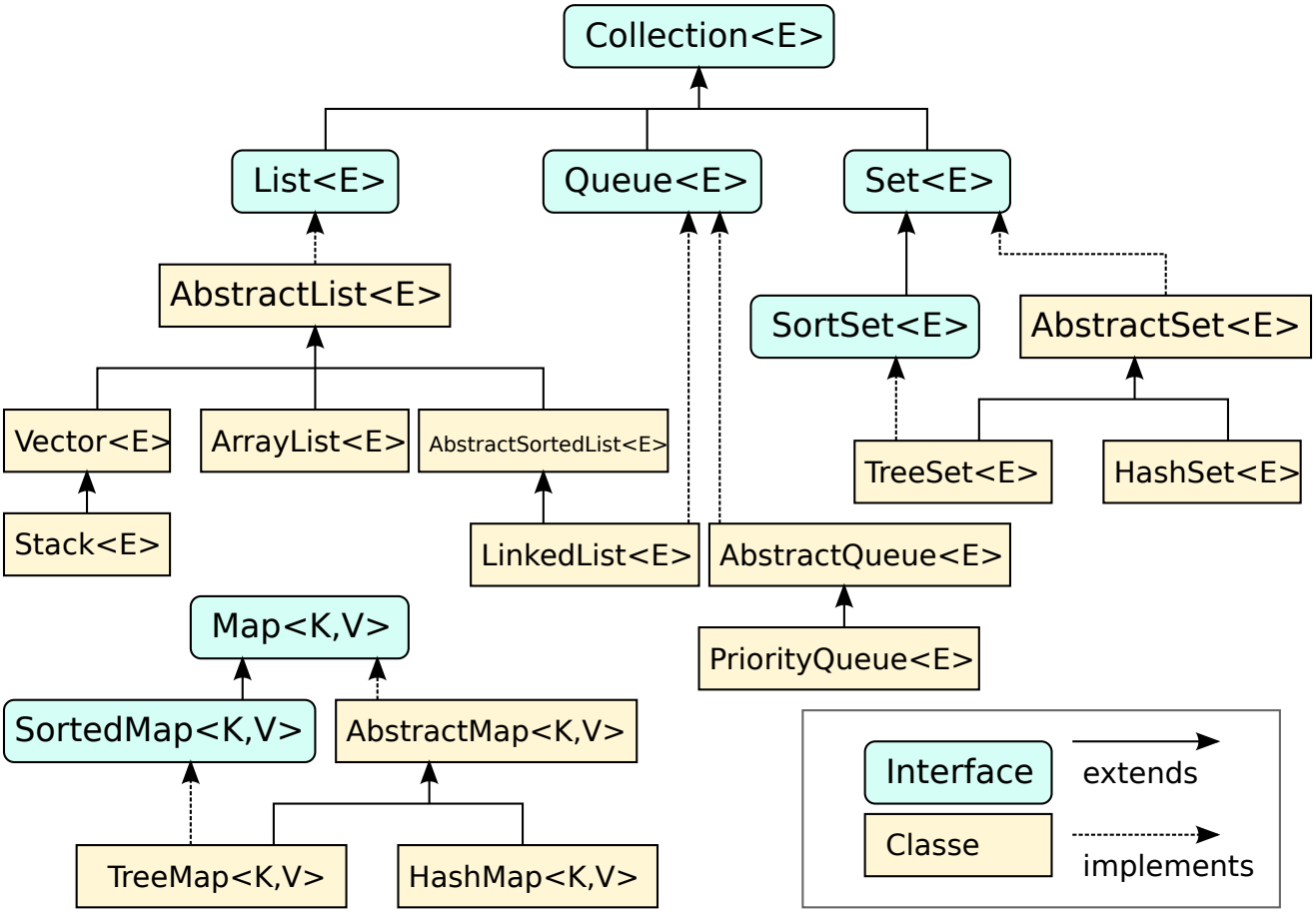


FIGURE 57 – Diagramme de classes de la Java Collection

11.3 *QT Core* en C++

QT a été conçu par la compagnie TrollTech et racheté par Nokia. QT est avant-tout une bibliothèque portable (Linux/Unix (X11), Windows et Mac OS) pour créer des interfaces graphiques. La bibliothèque QT offre ses propres structures de données pouvant être utilisées en alternative à STL.

Référence : `http://doc-snapshot.qt-project.org/5.0/index.html`.

11.4 .NET Framework

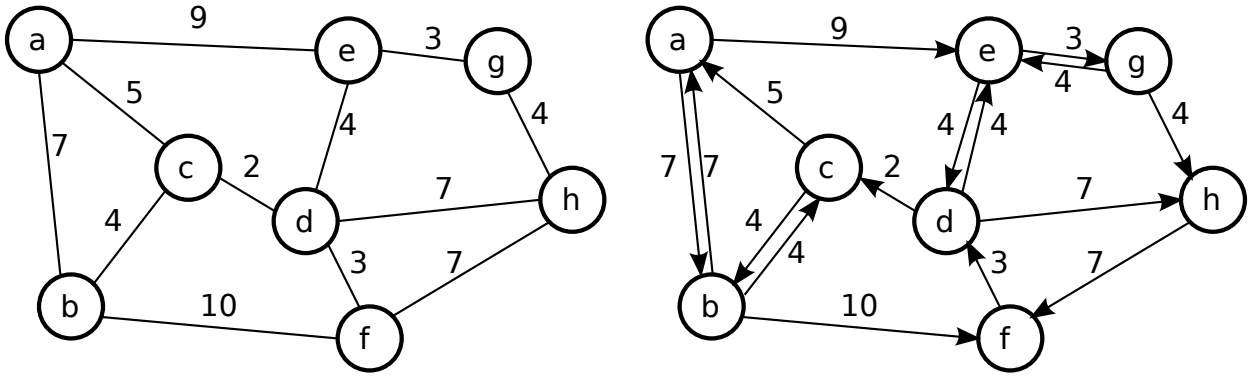
Référence : `http://msdn.microsoft.com/en-us/library/system.collections.generic.aspx`.

Cinquième partie

Graphes et algorithmes

12 Graphes

Les graphes sont des structures abstraites utilisées dans une multitude d'applications. La Figure 58(a) illustre un exemple de graphe non orienté modélisant une carte. Les sommets $\{a,b,...,h\}$ et les arêtes représentent respectivement des lieux et des routes. Les arêtes représentent les routes et sont étiquetées par leur distance. La Figure 58(b) illustre un exemple de graphe orienté. Les cartes avec des sens uniques sont des exemples classiques de graphes orientés.



(a) Non orienté (b) Orienté

FIGURE 58 – Exemples de graphes

12.1 Définitions

Un **graphe** est une paire $G = (V,E)$ où V est un ensemble de **sommets** (*vertices*) et E est une collection d'**arêtes** (*edges*). Les sommets et arêtes peuvent également être appelés respectivement des **nœuds** (*nodes*) et des **arcs**. Un **sommet** (*vertex*) $v \in V$ représente un objet. Une **arête** $e = (x,y) \in E$ exprime une **relation** entre deux objets x,y tels que $x \in V, y \in V$. Comme E est une collection plutôt qu'un ensemble, plusieurs relations (arêtes) peuvent être définies entre une même paire de sommets. Toutefois, cela constitue davantage l'exception que la règle. En effet, il existe généralement au plus une relation (arête) d'un sommet à un autre. Ainsi, pour ces applications, les arêtes E peuvent être représentées par un ensemble plutôt que par une collection.

Un graphe peut être **orienté** (*directed*) ou **non orienté** (*undirected*). Un graphe est orienté dès qu'il contient une arête orientée. Sinon, le graphe est non orienté. Une arête $e = (x,y)$ est **orientée** lorsque le sens (direction) de la relation entre les sommets x et y est important. Dans la représentation graphique, on utilise une flèche pour indiquer le sens de la relation. Dans un graphe non orienté, les relations sont **symétriques** : l'existence d'une relation (arête) (x,y) implique l'existence d'une relation (y,x).

Un **sous-graphe** $G' = (V',E')$ du graphe $G = (V,E)$ est un graphe qui contient un sous-ensemble des sommets et arêtes de G. Tous les sommets aux extrémités des arêtes dans E' doivent être dans V'.

Un **chemin** dans un graphe $G = (V,E)$ est une séquence d'arêtes $c = \langle e_1, \ldots, e_n \rangle$ tel que $e_i \in E$. Une notation alternative ajoute les sommets empruntés : $c = \langle v_0, e_0, v_1, \ldots, e_{n-1}, v_n \rangle$ tel que $v_i \in V$ et $e_i \in E$. La **longueur** d'un chemin c est égale au nombre d'arêtes qu'il contient.

Un **cycle** est un chemin $c = \langle v_0, e_0, v_1, \ldots, e_{n-2}, v_{n-1} \rangle$ tel que $v_0 = v_{n-1}$. Une **boucle** est une arête $e = (x,y)$ tel que $x = y$. Une boucle peut aussi être définie comme étant un cycle ayant une seule arête. Un graphe est dit **acyclique** si et seulement si il ne contient aucun cycle.

Un graphe non orienté est **connexe** (ou **connecté**) s'il existe au moins un chemin entre toutes les paires de sommets. Un graphe $G = (V,E)$ non connecté est composé de plusieurs composantes connexes. Une **composante connexe** est un sous-graphe connecté et maximal. Les sommets des composantes connexes d'un graphe constituent une partition des sommets. Un graphe orienté est **fortement connexe** (ou **fortement connecté**) si et seulement si pour toute paire de sommets (a,b) il existe un chemin de a à b, et de b à a. Un graphe orienté qui n'est pas fortement connexe est composé de **composantes fortement connexes**.

Les arêtes peuvent être **étiquetées**. Une **étiquette** indique une propriété d'une arête. Par exemple, dans les graphes de la Figure 58, les étiquettes représentent des distances. Certains auteurs qualifient de **poids** (*weights*) les valeurs des arêtes lorsqu'elles représentent une notion de **distance** ou de **coût**.

Dans un graphe non orienté, le **degré** d'un sommet $v \in V$, noté $deg(v)$, est le nombre d'arêtes qui y sont reliées. Dans un graphe orienté, on fait la distinction entre le **degré sortant** et le **degré entrant**, qui sont respectivement notés $deg_{out}(v)$ et $deg_{in}(v)$.

Un **arbre** est un cas particulier de graphe non orienté. À l'exception de l'arbre vide, un arbre est un graphe ayant une seule composante connexe tel que le nombre de sommets est égal au nombre d'arêtes plus un ($|V| = |E| + 1$), et où tous les sommets sont accessibles à partir d'un sommet qualifié de **racine**. Un arbre est un graphe qui est forcément acyclique. Un graphe qui est composé d'un ensemble d'arbres est appelé une **forêt**.

12.2 Opérations typiques sur un graphe

Pour créer et exploiter un graphe, on a généralement besoin des opérations suivantes :

- insérer, modifier et supprimer un sommet ;
- insérer, modifier et supprimer une arête ;
- vérifier s'il existe une arête (relation) reliant deux sommets ;
- lister (itérer) les sommets ;
- lister (itérer) les arêtes ;
- lister (itérer) les arêtes (sortantes ou entrantes) connectées à un sommet particulier ;

La complexité temporelle de ces opérations varie selon la représentation de graphe utilisée et s'exprime généralement en fonction des paramètres n, m et deg_{max} désignant respectivement le nombre de sommets, le nombre d'arêtes et le degré maximal d'un sommet. Généralement, $n < m < n^2$ et $deg_{max} << n$.

12.3 Représentations

Il existe plusieurs façons de représenter un graphe en mémoire. Le choix de la représentation dépend des opérations requises qui elles dépendent à leur tour de l'application. La complexité spatiale des représen-

tations peut aussi influencer ce choix.

Dans les représentations suivantes, les types de données `S` et `A` sont utilisés pour étiqueter les sommets et les arêtes. Certaines représentations sont construites à l'aide des classes abstraites et génériques `Ensemble` et `Collection`. Ces types indiquent une certaine liberté dans le choix d'implémentation. Par exemple, le type `Ensemble` peut être un arbre de recherche (AVL ou rouge-noir) ou un ensemble haché (*hash set*). Une collection peut être implémentée au moyen d'un tableau, d'une liste chaînée, d'un arbre de recherche [15], etc.

12.3.1 Ensemble de sommets et collection d'arêtes

```
1  template <class S, class A>
2  class Graphe {
3      struct Arete{
4          S depart, arrivee;
5          A etiquette;
6      };
7      Ensemble<S> sommets;
8      Collection<Arete> aretes;
9    //...
10 };
```

Le code ci-dessus implémente directement de la définition formelle d'un graphe orienté. La Figure 59 illustre cette représentation pour le graphe de la Figure 58(b).

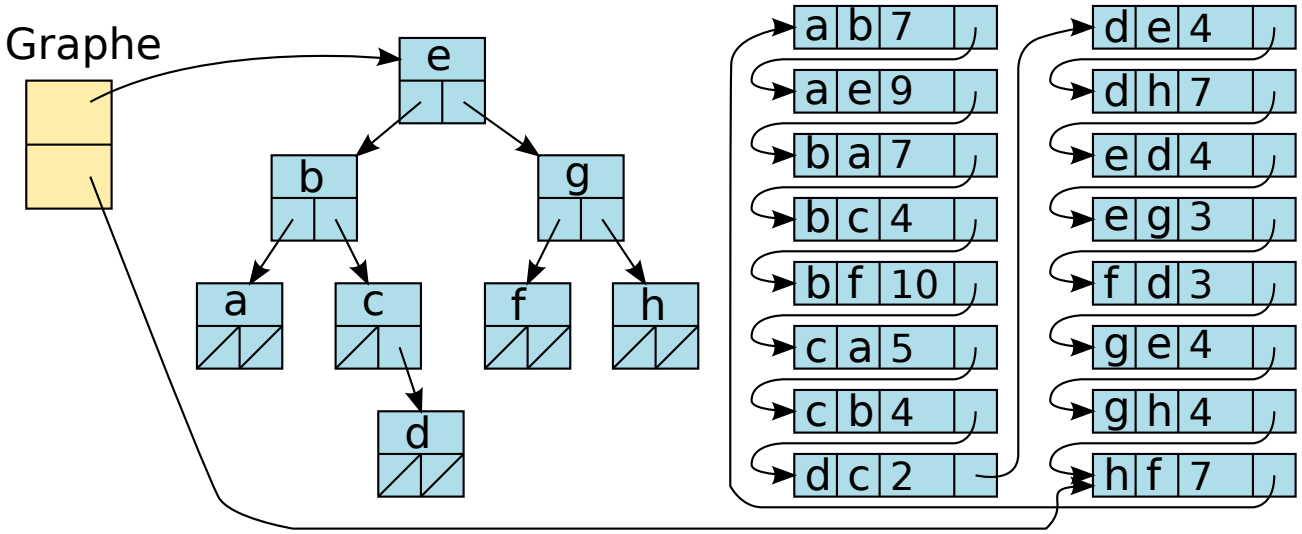


FIGURE 59 – Exemple de représentation formelle d'un graphe

Bien qu'elle soit simple, cette représentation a plusieurs lacunes. Par exemple, afin de pouvoir lister les arêtes sortantes et entrantes d'un sommet, il faut itérer sur toutes les arêtes. La complexité de ces opérations est donc $O(m)$.

15. Il est possible de modifier les arbres binaires de recherche afin de permettre plusieurs instances d'un même objet. La classe `std::multiset` dans la STL le permet.

12.3.2 Matrice d'adjacence

Un graphe peut être représenté à l'aide d'une **matrice d'adjacence**. Le code ci-dessous illustre une telle représentation. Cette ébauche a deux tableaux à deux dimensions. Le premier tableau marque l'absence ou la présence d'une relation à l'aide de booléens. Le second mémorise l'étiquette de chaque relation.

```
1  template <class S, class A>
2  class Graphe {
3      Tableau<S> sommets;
4      Tableau2D<bool> relations;
5      Tableau2D<A> etiquettes;
6  };
```

Le code suivant présente une version plus compacte que la précédente. Pour représenter l'absence de relation, on peut réserver une valeur spéciale d'étiquette. Celle-ci est mémorisée dans la variable statique `AUCUNE_RELATION`. Par exemple, lorsque les étiquettes sont des nombres réels, on peut utiliser la valeur `NaN`. La Figure 60 montre un exemple de représentation matricielle. Les cases vides marquent l'absence de relation.

```
1  template <class S, class A>
2  class Graphe {
3      Tableau<S> sommets;
4      Tableau2D<A> etiquettes;
5      static A AUCUNE_RELATION;
6  };
7  //...
8  double Graphe<std::string, double>::AUCUNE_RELATION = NaN; // 0/0
9  //...
10 Graphe<std::string, double> graphe.
```

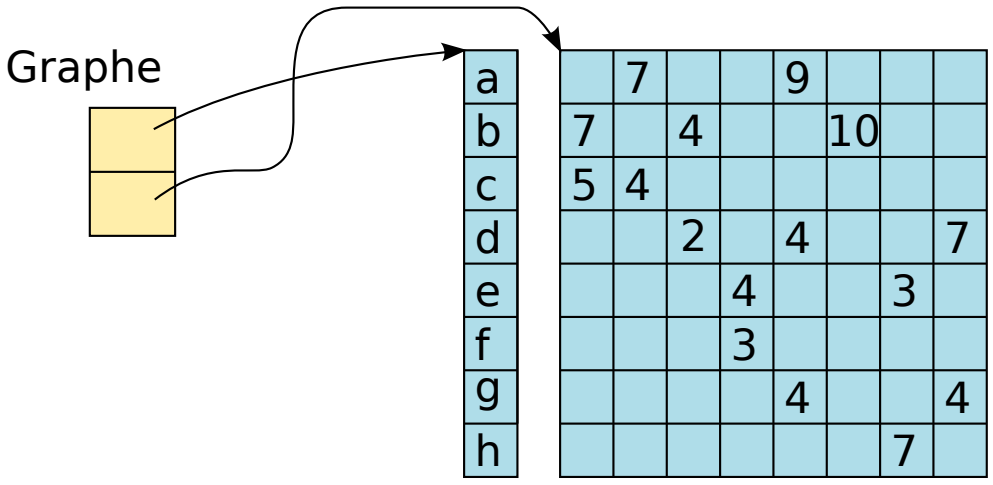


FIGURE 60 – Exemple de représentation matricielle

Une représentation matricielle permet de déterminer si deux sommets ont une relation en temps $O(1)$. Toutefois, il faut un temps $O(n)$ pour lister les arêtes (entrantes ou sortantes) connectées à un sommet

puisqu'il faut itérer toute une ligne de la matrice. La représentation matricielle a une complexité spatiale de $O(n^2)$. Cela constitue un important désavantage lorsque m est nettement inférieur à n^2.

Dans le cas d'un graphe non orienté, la matrice d'adjacence est toujours symétrique. Pour économiser la mémoire, il est possible de ne garder qu'une seule moitié (un triangle) de la matrice. Cette représentation est laissée en exercice.

12.3.3 Listes d'adjacence

Si $m << n^2$, on peut utiliser une représentation basée sur une «matrice creuse». Pour lister efficacement les arêtes connectées à un sommet, il est possible d'attacher une ou deux listes d'adjacence à chaque sommet. Le code suivant présente une telle ébauche. Les sommets sont représentés à l'aide d'une classe interne `Graphe::Sommet`, qui contient la valeur du sommet, et deux listes chaînées, l'une pour les arêtes sortantes et l'autre pour les entrantes. Les arêtes sont représentées par les valeurs des sommets aux extrémités, et de la valeur étiquetée. Cette représentation permet, lorsqu'un sommet a est sélectionné, de lister les arêtes sortantes en temps $O(deg_{out}(a))$, et les arêtes entrantes en temps $O(deg_{in}(a))$. À noter qu'il est possible d'économiser de la mémoire en ne conservant qu'une seule extrémité dans la classe `Arete`. La Figure 61 montre un exemple de représentation avec une liste d'adjacente.

```
1  template <class S, class A>
2  class Graphe {
3      struct Arete{
4          S depart, arrivee;
5          A valeur;
6      };
7      struct Sommet {
8          S valeur;
9          Liste<Arete> aretesSortantes;
10         Liste<Arete> aretesEntrantes; // optionnel
11     };
12     Tableau<Sommet> sommets;
13   //...
14 };
```

Le code précédent a deux lacunes. La première est au niveau de la robustesse. Si on considère la définition formelle d'un graphe, les sommets sont un ensemble et non une collection. Ainsi, il ne doit pas y avoir deux sommets avec la même valeur. Donc, lors de l'insertion d'un nouveau sommet, il faudrait parcourir le `Tableau<Sommet> sommets` pour vérifier qu'il ne contient pas déjà le sommet en question. L'autre lacune consiste à sélectionner efficacement un sommet. En effet, les extrémités des arêtes sont représentées à l'aide de valeurs de type `S`. Pour retrouver l'élément `Sommet` correspondant dans le tableau sommets, il faut itérer sur les éléments du tableau.

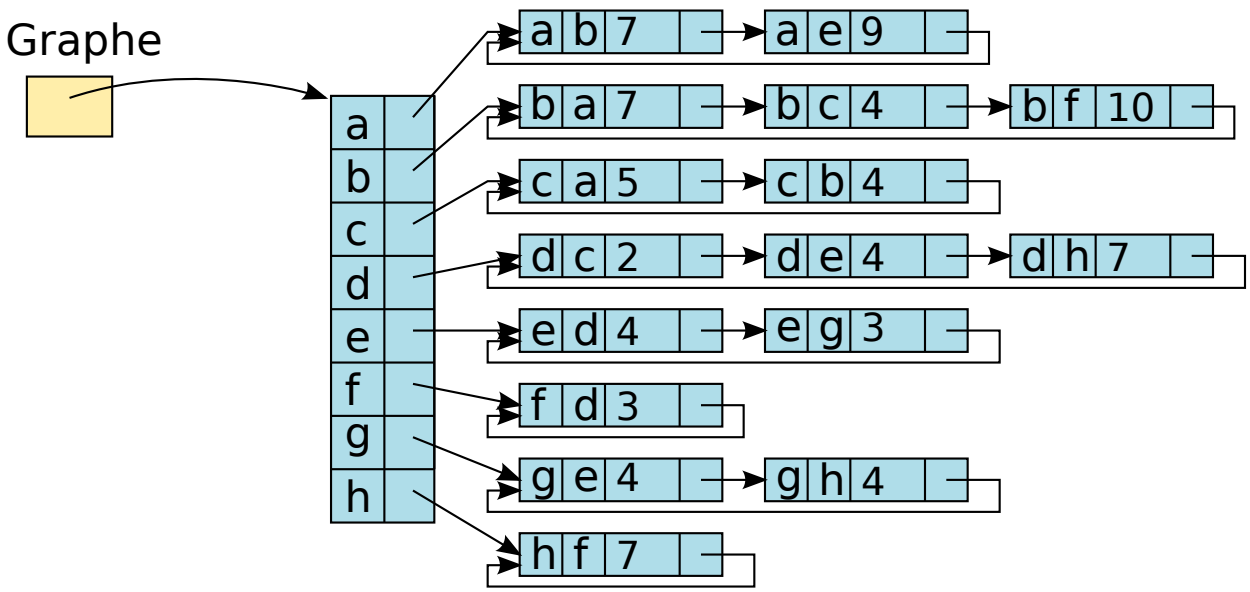


FIGURE 61 – Exemple de représentation avec des listes d'adjacence

12.3.4 Ensembles d'adjacence avec des indices

Pour résoudre les lacunes de la représentation précédente, on peut utiliser des pointeurs ou des indices pour lier les sommets. Le code à la page suivante présente une représentation avec des indices. Les arêtes sortantes sont stockées dans un *map* : chaque arc sortant du sommet courant est une entrée dans le *map* où la clé est le sommet d'arrivée et la valeur est l'étiquette associée. S'il n'est pas requis de remonter les arêtes vers l'arrière, le *map* `aretesEntrantes` peut être enlevé.

La Figure 62 illustre un exemple d'une telle représentation. À noter que pour des fins d'espace, seules les arêtes liées aux sommets *a* et *b* sont illustrées.

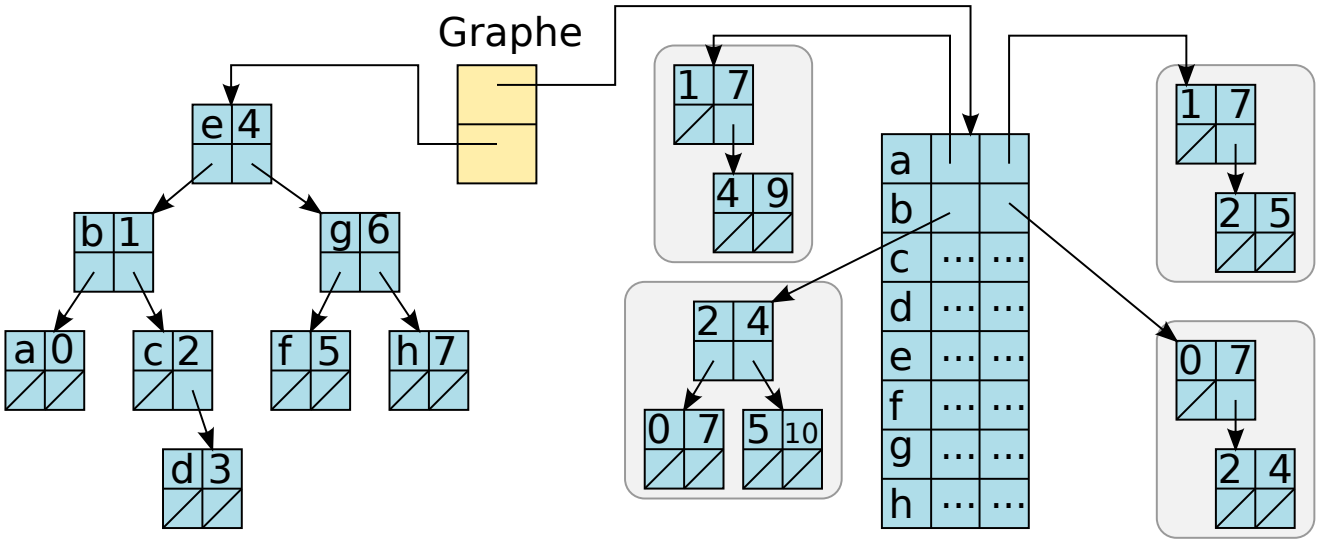


FIGURE 62 – Exemple de représentation avec des ensembles (arbres de recherche) d'adjacence

```
1  template <class S, class A /*, class ES*/ > // ES=Etiquette sommet
2  class Graphe {
3      struct Sommet{
4          Sommet(const S& s_) : s(s_){}
5          S s; // optionnel : utile pour traduire indice -> objet S
6          ArbreMap<int, A> aretesSortantes;
7          ArbreMap<int, A> aretesEntrantes; // optionnel
8          //ES etiquette; // utile pour associer des objets a un sommet
9      };
10     ArbreMap<S, int> indices; // pour traduire S en indice
11     Tableau<Sommet> sommets;  // stock les sommets
12   public:
13     void ajouterSommet(const S& s){
14         assert(!indices.contient(s));
15         int indice = indices.taille();
16         indices.inserer(s, indice);
17         sommets.inserer_fin(Sommet(s));
18     }
19     void ajouterArete(const S& a, const S& b, const A& etiquette){
20         int ia = indices[a];
21         int ib = indices[b];
22         sommets[ia].aretesSortantes[ib] = etiquette;
23         sommets[ib].aretesEntrantes[ia] = etiquette;
24     }
25 };
```

13 Algorithmes pour les graphes

13.1 Recherche et parcours dans un graphe

Comme pour les nœuds d'un arbre, les sommets et les arêtes d'un graphe peuvent être visités. Deux types de parcours sont présentés, soit la recherche en profondeur et la recherche en largeur. Contrairement aux arbres, les graphes peuvent contenir des boucles et des cycles. En raison de ces cycles, il est requis de marquer les sommets visités. Dans l'implémentation, cela peut se faire au moyen d'un tableau temporaire de marqueurs ou en ajoutant une variable booléenne `visité` dans la classe `Graphe::Sommet`.

13.1.1 Recherche en profondeur

Le premier type de parcours est une **recherche en profondeur** (*depth-first search*) tel que décrit par l'Algorithme 3. Ce type de parcours est qualifié de recherche en **profondeur** puisqu'elle « creuse » en profondeur dans le graphe plutôt que de commencer par itérer sur les sommets adjacents. Cet algorithme récursif reçoit en entrée un graphe $G = (V,E)$ et un sommet de départ $v \in V$. La propriété *visité* des sommets doit être initialisée à faux. L'algorithme visite tous les sommets accessibles depuis v.

Algorithme 3 Recherche en profondeur

```
1. RECHERCHEPROFONDEUR(G = (V,E), v ∈ V)
2.     v.visité ← vrai
3.     pour toute arête e ∈ v.aretesSortantes()
4.         w ← e.arrivee
5.         si ¬w.visité
6.             RechercheProfondeur(G, w)
```

La Figure 63 illustre un exemple de recherche en profondeur à partir de a. On suppose que les arêtes sont ordonnées dans l'ordre lexicographique. Les nombres dans les rectangles indiquent l'ordre de visite des sommets. Ainsi, un appel à RECHERCHEPROFONDEUR(G, a) visite les sommets dans l'ordre suivant : $\langle a,b,c,f,e,g,h\rangle$. À noter que le sommet d n'est pas visité, car il n'est pas accessible depuis a.

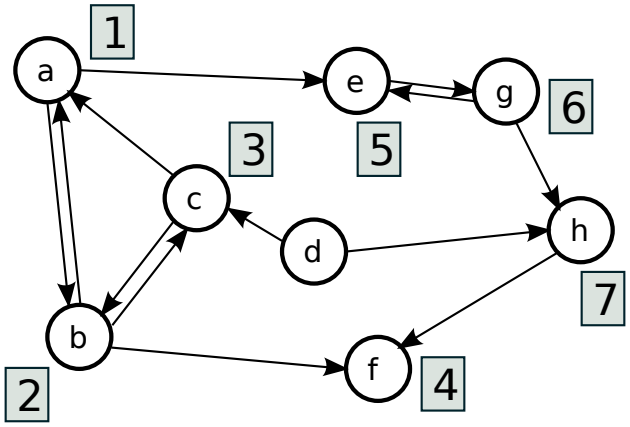


FIGURE 63 – Exemple de recherche en profondeur à partir du sommet a

L'Algorithme 3 s'implémente naturellement de façon récursive. Toutefois, une implémentation récursive n'est pas toujours appropriée. Selon le langage, le compilateur et l'environnement d'exécution utilisés,

la taille maximale de la pile d'exécution peut être fixe. Or, si la recherche en profondeur emprunte un long chemin, il y aura un nombre important d'appels récursifs sur la pile d'exécution. Ainsi, un débordement de la pile d'exécution (*stack overflow*) n'est pas inévitable. Afin d'éviter le débordement de la pile d'exécution, il est possible d'implémenter la recherche en profondeur de façon non récursive. Ainsi, au lieu d'utiliser la pile d'exécution, on utilise une pile allouée sur le tas (*heap*). En pratique, les versions non récursives sont généralement légèrement plus rapides. Une variante possible de la recherche en profondeur est *iterative depth-first search* (IDFS).

13.1.2 Recherche en largeur

Le parcours de **recherche en largeur** (*breadth-first search*) est décrit par l'Algorithme 4. Ce parcours est qualifié de recherche en largeur puisqu'il visite en premier les sommets adjacents avant d'aller fouiller plus profondément.

Algorithme 4 Recherche en largeur

```
1.  RECHERCHELARGEUR(G = (V,E), v ∈ V)
2.      file ← CRÉERFILE
3.      s.visité ← vrai
4.      file.ENFILER(v)
5.      tant que ¬file.vide()
6.          s ← file.defiler()
7.          pour toute arête a = (s,s') ∈ E
8.              si ¬s'.visité
9.                  s'.visité ← vrai
10.                 file.ENFILER(s')
```

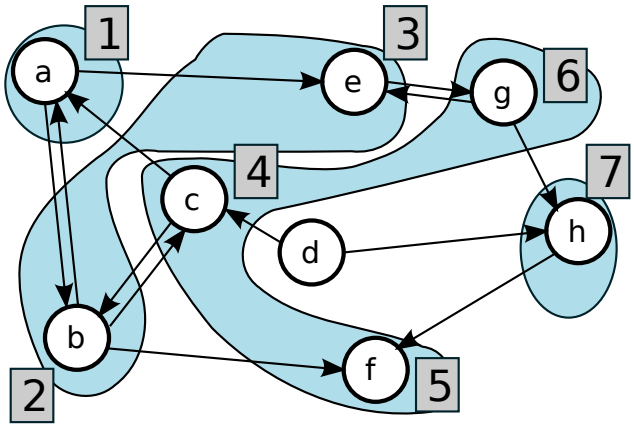


FIGURE 64 – Exemple de recherche en largeur à partir du sommet *a*

13.2 Extraction des composantes connexes

13.2.1 Extraction des composantes connexes dans des graphes non orientés

Le code suivant présente la fonction `extraireComposanteConnexe()` qui extrait les sommets des composantes connexes d'un graphe non orienté. Il s'agit d'une adaptation de `rechercheLargueur()` qui

retourne une liste des sommets accessibles depuis un sommet de départ.

```
1  Liste<ArbreAVL<S> > Graphe::extraireComposanteConnexe() const{
2      Tableau<bool> marqueurs(sommets.taille());
3      Liste<ArbreAVL<S> > sommets_composantes_connexes;
4      marqueurs.remplir(false);
5      for(int i=0;i<sommets.taille();i++){
6          if(marqueurs[i]) continue;
7          ArbreAVL<S> sommets_visites;
8          sommets_visites = rechercheLargueur(i, marqueurs);
9          sommets_composantes_connexes.inserer(sommets_visites);
10     }
11     return sommets_composantes_connexes;
12 }
```

La fonction `extraireComposanteConnexe` pourrait être modifiée pour retourner une liste d'objets `Graphe`.

13.2.2 Extraction des composantes fortement connexes (algorithme de Tarjan)

L'Algorithme 5 décrit l'algorithme de Tarjan [Tar72] qui extrait les sommets des composantes fortement connexes d'un graphe orienté. La Figure 65 montre une trace d'exécution de l'algorithme de Tarjan.

Algorithme 5 Calcul des composantes fortement connexes

```
1.  TARJAN(G = (V,E))
2.    pour tout v ∈ V
3.      v.num ← −1
4.    compteur ← 0
5.    chemin ← Pile vide
6.    R ← { }
7.    pour tout v ∈ V
8.      si v.num = −1
9.        ParcoursProfondeur(v)
10.   retourner R
11. PARCOURSPROFONDEUR(v)
12.   v.num ← compteur++
13.   v.min ← v.num
14.   chemin.empiler(v)
15.   pour toute arête sortante (v,w)
16.     si w.num = −1
17.       ParcoursProfondeur(w)
18.       v.min ← min(v.min, w.min)
19.     sinon si w ∈ chemin
20.       v.min ← min(v.min, w.num)
21.   si v.num = v.min
22.     C ← {}
23.     répéter
24.       w ← chemin.dépiler()
25.       C.ajouter(w)
26.       tant que v ≠ w
27.     R ← R ∪ C
```

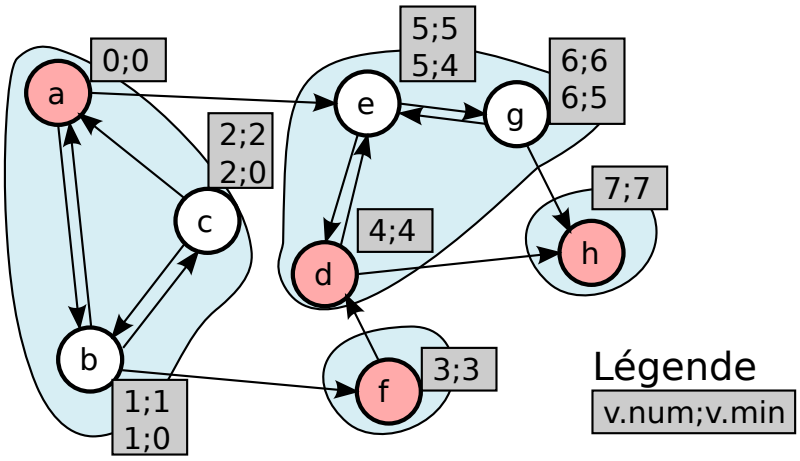


FIGURE 65 – Exemple de calcul des composantes connexes

13.3 Recherche de chemins

13.3.1 Algorithme de Dijkstra

L'Algorithme 6 présente l'algorithme de Dijkstra qui permet de trouver un **chemin optimal** (la distance la plus courte) reliant une paire de sommets dans un graphe connexe. En fait, cet algorithme ne se limite pas au calcul d'un chemin reliant une paire de sommets. Il permet de calculer un **arbre de chemins** à partir d'un sommet de départ vers tous les autres sommets. À noter que l'algorithme de Dijkstra assume que les distances (poids des arêtes) sont positives (≥ 0).

Algorithme 6 Dijkstra

```
1.  DIJKSTRA(G = (V,E), s ∈ V)
2.    pour tout v ∈ V
3.      distances[v] ← +∞
4.      parents[v] ← indéfini
5.    distances[s] ← 0
6.    Q ← créer FilePrioritaire(V)
7.    tant que ¬Q.vide()
8.      v ← Enlever v ∈ Q avec la plus petite valeur distances[v]
9.      si distances[v] = +∞ break
10.     pour toute arête sortante e = (v,w) depuis le sommet v
11.       d ← distances[v] + e.distance
12.       si d < distances[w]
13.         parents[w] ← v
14.         distances[w] ← d
15.         Q.RéduireClé(w) // ou Q.insérer(w)
16.   retourner (distances, parents)
```

Le fonctionnement de l'algorithme repose sur le calcul incrémental des distances minimales vers chacun des sommets. À la fin du calcul, chaque élément du tableau $distances[i]$ contient la distance minimale entre le sommet de départ s et un sommet de destination i. Les distances sont initialisées à $+\infty$ (ligne 2), à l'exception du sommet de départ à 0 (ligne 5). À la fin du calcul, chaque case du tableau $parents[i]$ contient un pointeur (indice) vers le sommet parent dans l'arbre de chemins les plus courts. Par exemple,

le chemin le plus court pour se rendre à un sommet x contient l'arête $(parents[x], x)$. Initialement, les parents sont indéfinis. Le cœur de l'algorithme repose sur la boucle à la ligne 7. Cette boucle itère sur les sommets insérés dans la file prioritaire Q afin de mettre à jour le tableau de distances. Pour chaque sommet v, on itère sur les arêtes sortantes $e = (v, w)$ pour visiter tous les sommets w accessibles à partir de v. La ligne 11 calcule la distance d pour se rendre au sommet w en passant par v. Si cette distance est meilleure que celle déjà présente dans la case $distances[i]$, alors un meilleur chemin a été trouvé. Ainsi, on met à jour $parents[i]$ et $distances[i]$. Puisqu'un nouveau meilleur chemin a été trouvé pour se rendre à w, il faudra revérifier si d'autres distances vers des sommets à partir de w peuvent être améliorées. Ainsi, à la ligne 15, on ajoute w à Q.

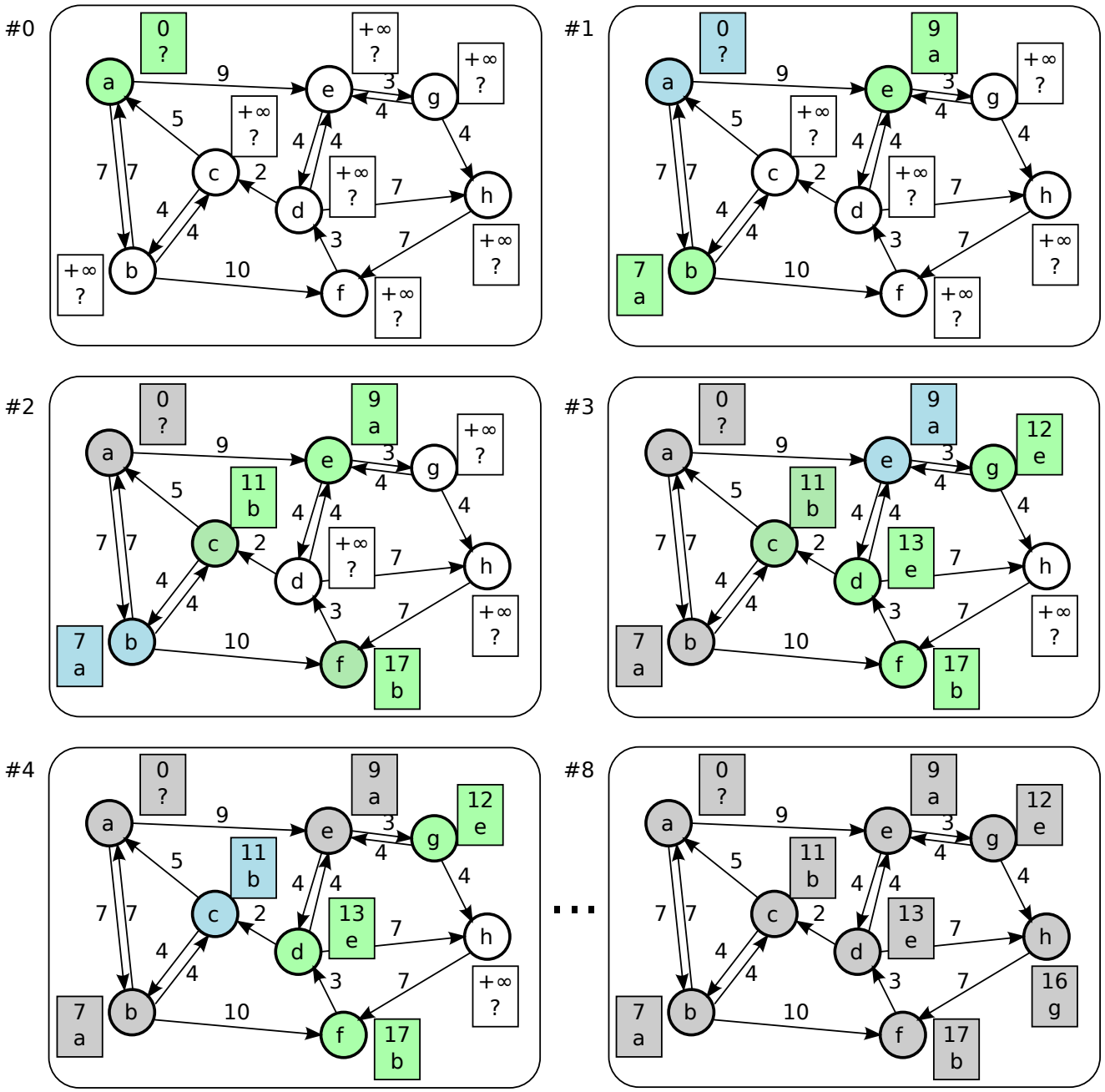


FIGURE 66 – Exemple de l'algorithme de Dijkstra

La Figure 66 montre un exemple d'application de Dijkstra où le sommet de départ est a. À chaque étape, le sommet courant (v à la ligne 8 de l'Algorithme 6) est indiqué en bleu. La couleur verte indique les sommets mis à jour dans la file Q. Les sommets sont visités dans l'ordre suivant : $\langle a,b,e,c,g,d,h,f\rangle$. Une fois l'arbre de chemins calculé, on peut extraire un chemin vers un sommet particulier en remontant les pointeurs *parents*. Par exemple, pour aller au sommet h, on sait qu'il faut passer par g. Pour arriver à g, il faut passer par e. En remontant jusqu'au sommet initial a, on obtient le chemin $\langle a,e,g,h\rangle$.

13.3.2 Analyse de Dijkstra

La complexité temporelle de l'algorithme de Dijkstra s'exprime en fonction du nombre de sommets ($n = |V|$) et d'arêtes ($m = |E|$). Dans le pire cas, l'algorithme doit visiter une fois tous les sommets et toutes les arêtes. La file prioritaire Q joue un rôle central. Tous les sommets dans V doivent être extrait de Q. Les sommets peuvent être réordonnés (remontés en priorité) dans Q à chaque fois qu'une arête permet la découverte d'un meilleur coût (ligne 12). Ainsi, on a n retraits du sommet minimum et m réordonnancements.

La complexité temporelle dépends donc de l'implémentation de la file prioritaire Q. On peut supposer l'utilisation d'une file prioritaire basée sur un monceau (Section 9). L'insertion et l'enlèvement de l'élément minimal sont en temps logarithmique, soit $O(\log n)$. Ainsi, la complexité temporelle de Dijkstra est de $O(n\log(n)+m\log(n))$. Toutefois, on peut utiliser une structure de données plus efficace pour la file prioritaire, comme le **monceau de Fibonacci** ou la **file brodée** offrant une insertion en temps constant et l'enlèvement en temps logarithmique. Ainsi, la complexité temporelle descends à $O(n\log(n)+m)$.

La complexité spatiale de Dijkstra est de $O(n)$. En effet, la mémoire temporaire [16] requise pour exécuter Dijkstra se limite aux deux tableaux *distances* et *parents* ayant chacun la taille du nombre de sommets.

13.3.3 Adaptations à l'algorithme de Dijkstra

Selon les applications, il est possible de faire quelques améliorations et adaptations. Par exemple, si on s'intéresse uniquement à une source et une seule destination, il est possible d'ajouter une condition à l'algorithme pour arrêter le calcul dès que la destination est visitée. Cela peut éviter d'avoir à visiter tout le graphe en entier.

L'Algorithme 6 présenté est *point to multi-point*. Il est possible d'adapter l'algorithme à *multi-point to point*. Pour cela, il suffit de parcourir les arêtes dans le sens inverse.

13.3.4 Algorithme de Floyd-Warshall

L'Algorithme 7 présente l'algorithme de Floyd-Warshall. Ce dernier calcule les meilleurs chemins pour toutes les paires de sommets d'un graphe. Le fonctionnement de cet algorithme consiste à calculer de façon itérative la distance minimale entre chaque paire de sommets. Les coûts des arêtes initialisent les distances connues (lignes 4-6). Dans les 3 boucles pour imbriquées, on vérifie si le chemin passant par k pour se rendre de i à j est plus court que le meilleur chemin connu jusqu'à présent. Si c'est le cas, on met à jour la nouvelle distance minimale trouvée. Pour chaque paire, on note aussi la direction à prendre. Par exemple, pour se rendre de a à b, on passe par $x_1 = directions[a][b], x_2 = directions[x_1][b], \cdots$.

16. excluant la mémoire servant à stocker le graphe

Algorithme 7 Floyd-Warshall

```
1.  FLOYD_WARSHALL(G = (V,E))
2.    distances ← créer tableau |V| × |V| initialisé à +∞
3.    directions ← créer tableau |V| × |V| initialisé à « ?»
4.    pour tout v ∈ V
5.      distances[v][v] ← 0
6.      directions[v][v] ← v
7.    pour tout e ∈ E
8.      distances[e.out][e.in] ← e.cout
9.      directions[e.out][e.in] ← e.in
10.   pour k = 0 à |V| − 1
11.     pour i = 0 à |V| − 1
12.       pour j = 0 à |V| − 1
13.         si distances[i][k] + distances[k][j] < distances[i][j]
14.           distances[i][j] ← distances[i][k] + distances[k][j]
15.           directions[i][j] ← directions[i][k]
16.   retourner directions
```

13.4 Recouvrement minimum

Un **arbre de recouvrement** d'un graphe non orienté et connexe est un sous-graphe connexe qui contient tous les sommets mais sans cycle. Dans un arbre, il existe exactement un chemin unique pour relier toute paire de sommets. À l'exception du graphe vide, le nombre d'arrêtes est égal au nombre de sommets moins un. Il existe habituellement plusieurs arbres de recouvrement.

Généralement, on s'intéresse à un **arbre de recouvrement à coût minimum** (ARM). Plusieurs applications font appel au calcul d'un ARM. L'exemple classique est la construction d'un réseau de télécommunication reliant plusieurs sites. La solution la plus économique est obtenue en calculant un ARM. La Figure 67 présente un exemple d'arbre de recouvrement minimal pour un graphe. Les arêtes en gras indique l'arbre. Les arêtes en pointillés sont celles enlevées du graphe d'origine.

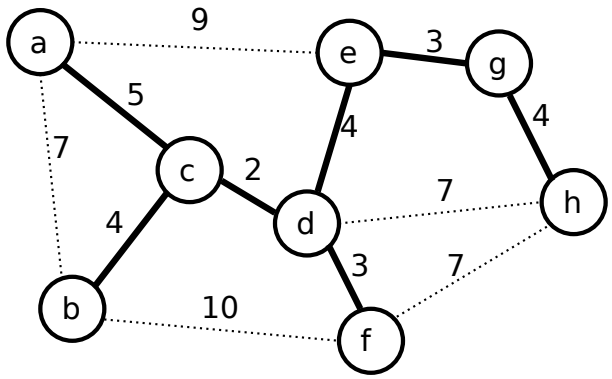


FIGURE 67 – Exemple d'arbre de recouvrement minimal d'un graphe

13.4.1 Algorithme de Prim-Jarnik

L'Algorithme 8 présente l'algorithme général de Prim-Jarnik (aussi appelé algorithme de Prim). Cet algorithme construit itérativement une composante connexe en partant d'un sommet choisi arbitrairement.

À chaque étape, on ajoute l'arête ayant le coût minimal qui permet de relier un sommet de la composante connexe à un sommet non relié.

Algorithme 8 Ébauche de Prim-Jarnik

```
1.  PRIM_JARNIK(G = (V,E))
2.     V' ← {v} où v est un sommet choisi arbitrairement dans V
3.     E' ← {}
3.        tant que V' ≠ V
4.           e = (s1,s2) ← e = (s1,s2) ∈ E tel que s1 ∈ V', s2 ∉ V' et cout(e) est minimal
5.           ajouter s2 à V'
6.           ajouter e à E'
7.     retourner G' = (V',E')
```

La Figure 68 montre un exemple d'exécution de l'algorithme de Prim-Jarnik. Le graphe original est montré en (a). En (b), le sommet a est choisi arbitrairement (ligne 3 de Algorithme 8). À la ligne 4, l'algorithme n'a qu'un choix pour $s_1 \in V$, soit $s_1 = a$, et trois choix pour $s_2 \notin V$, soit les sommets candidats sont b, c, e. Parmis ces trois choix, on doit choisir l'arête au coût minimal, soit (a,c). C'est ainsi qu'on ajoute c à V' et (a,c) à E' en (c). Ensuite, on doit choisir un sommet $s_2 \in \{b,d,e\}$. Il y a deux arêtes au coût minimun de 4. En (d), on choisit arbitrairement l'arête (c,b). On ajoute ainsi b à V' et (c,b) à E'. Enfin, on continue jusqu'à (i) où tous les sommets sont visités. L'arbre de recouvrement à coût minimal est montré à l'aide d'arêtes à trait épais.

Algorithme 9 Prim-Jarnik

```
1.  PRIM_JARNIK(G = (V,E))
2.     D ← créer un tableau de réels de dimension |V|
3.     pour tout i ∈ {0,...,|V|-1}
4.        D[i] ← +∞
5.     v ← choisir arbitrairement un sommet v ∈ V
6.     D[v] ← 0
7.     Q ← créer une FilePrioritaire<Sommet v, Arête e> où la priorité de chaque élément est D[v].
8.     pour tout v ∈ V
9.        ajouter (v,nul) dans Q avec priorité D[v]
10.    E' ← {}
11.    tant que Q n'est pas vide
12.       (v,e) ← Q.enleverMinimum()
13.       ajouter l'arête e à E'
14.       pour tout arête sortante a = (v,w) à partir du sommet v, tel que le sommet w ∈ Q
15.          si cout(a) < D[w] alors
16.             D[w] ← cout(a)
17.             mettre à jour l'élément (w,a) pour le sommet w dans Q
18.             mettre à jour la clé du sommet w dans Q à D[w]
19.    retourner G' = (V,E')
```

L'Algorithme 9 présente une version plus détaillée de l'algorithme de Prim-Jarnik. Le tableau D contient la distance entre chaque sommet et le sommet le plus proche dans la composante connexe visitée (ligne 2). Ces distances sont initialisées à $+\infty$ (lignes 3-4). Elles seront mises à jour à chaque fois qu'un sommet sera ajouté. À la ligne 5, on pré-choisi un sommet à partir duquel l'algorithme est initialisé. On assigne la valeur zéro pour ce sommet à la ligne 6. Cela forcera l'algorithme à visiter ce sommet en premier. À la ligne 7, une file prioritaire est créée. Chaque sommet élément de la file est une paire $< v,e >$ où v est

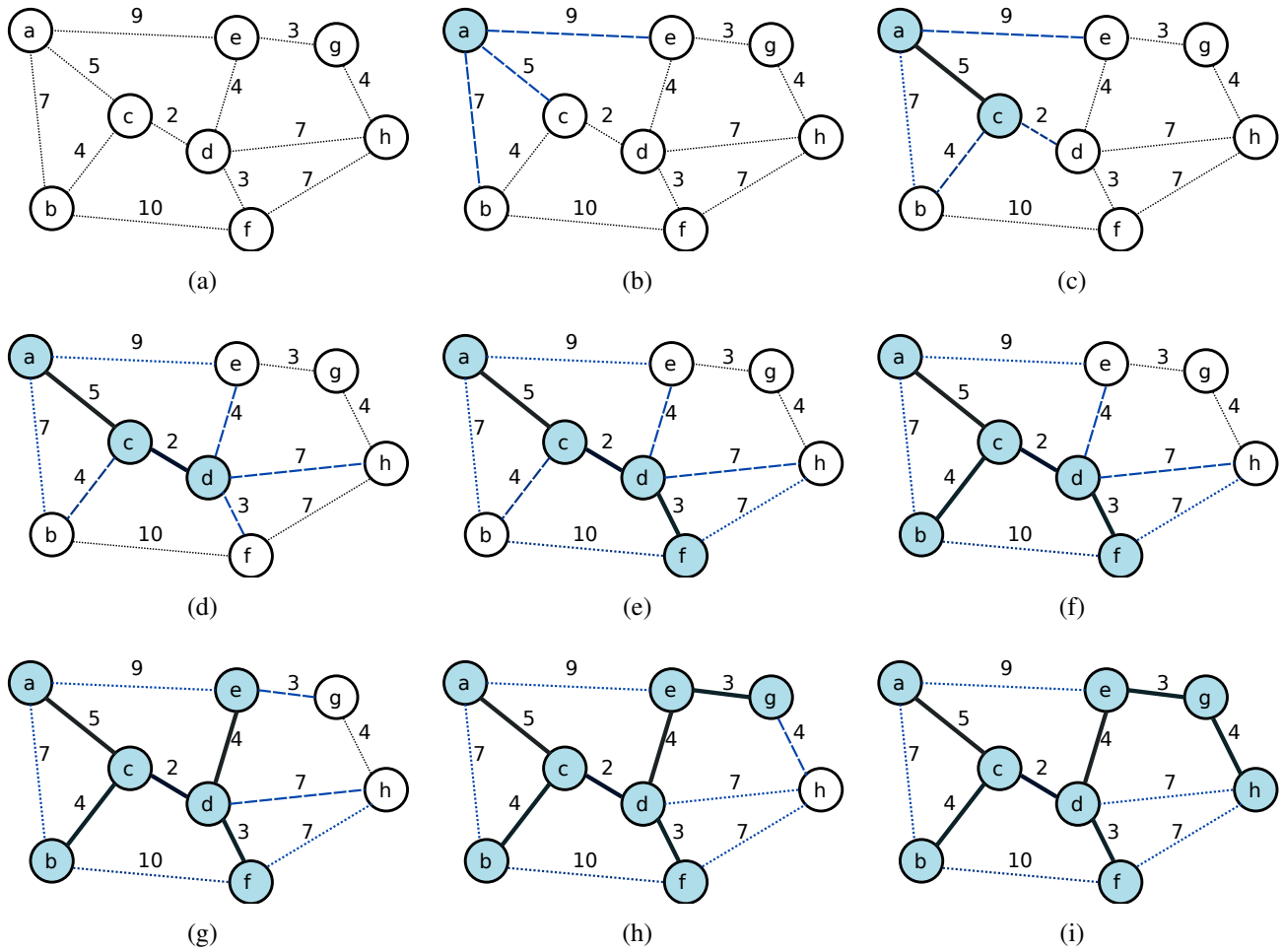


FIGURE 68 – Exemple d'exécution de l'algorithme de Prim-Jarnik

un sommet à visiter et e est l'arête offrant une distance minimal pour relier v à la composante connexe en cours de construction. Les lignes 8-9 ajoutent tous les sommets dans la file prioritaire Q. À la ligne 10, on initialise E' qui contiendra le résultat, soit les arêtes formant un arbre de recouvrement minimum. La boucle à la ligne 11 itère sur tous les sommets. À la ligne 12, on choisi un sommet v ayant un coût minimal à être relié à la composante connexe en cours de construction. On ajoute l'arête e, qui permet d'atteindre v, à E' (ligne 13). Maintenant que le sommet v est ajouté à la composante connexe, on doit regarder ce qui est atteignable à partir de v. C'est ainsi qu'on itère à la ligne 14 sur toutes les arêtes sortantes de v. Si une arête permet d'atteindre un sommet $w \notin Q$ avec un coût moindre (ligne 16), on met à jour cet information dans Q (lignes 17-18).

La Figure 68 permet de suivre le déroulement le l'Algorithme 9. Les arcs avec de longs traits en bleu sont les arêtes dans Q. Dans cet exemple, le sommet a a été choisi arbitrairement. À la première itération de la boucle tant que (ligne 11), on visite les arêtes sortantes de a. C'est à ce moment qu'on découvre que les 3 meilleures façons de connecter b, c, et e passent par a. Ainsi, on met à jour les valeurs $D[b]$, $D[c]$, $D[e]$ et la file prioritaire Q. La complexité temporelle de Prim-Jarnik est de $O(|E|\log|V|)$ [GTM11].

13.4.2 Algorithme de Kruskal

L'algorithme de Kruskal consiste à fusionner des composantes connexes jusqu'à temps qu'on retrouve la ou les composantes connexes d'origine. La complexité temporelle de l'algorithme de Kruskal est de $O(|E|\log|V|)$ [GTM11]. La Figure 69 montre un exemple d'exécution de l'algorithme de Kruskal.

Algorithme 10 Kruskal

1. KRUSKAL($G=(V,E)$)
2. pour tout sommet $v \in V$
3. $C(v) \leftarrow$ créer un ensemble $\{v\}$
4. $Q \leftarrow$ créer FilePrioritaire<Arête>(E) où la clé est le poids
5. tant que $|E'| < |V|-1$
6. $e=(v_1,v_2) \leftarrow Q.enleverMinimum()$
7. si $C(v_1) \neq C(v_2)$
8. ajouter e à E'
9. fusionner $C(v_1)$ et $C(v_2)$ afin que $C(v_1)=C(v_2)$
10. retourner $G'=(V,E')$

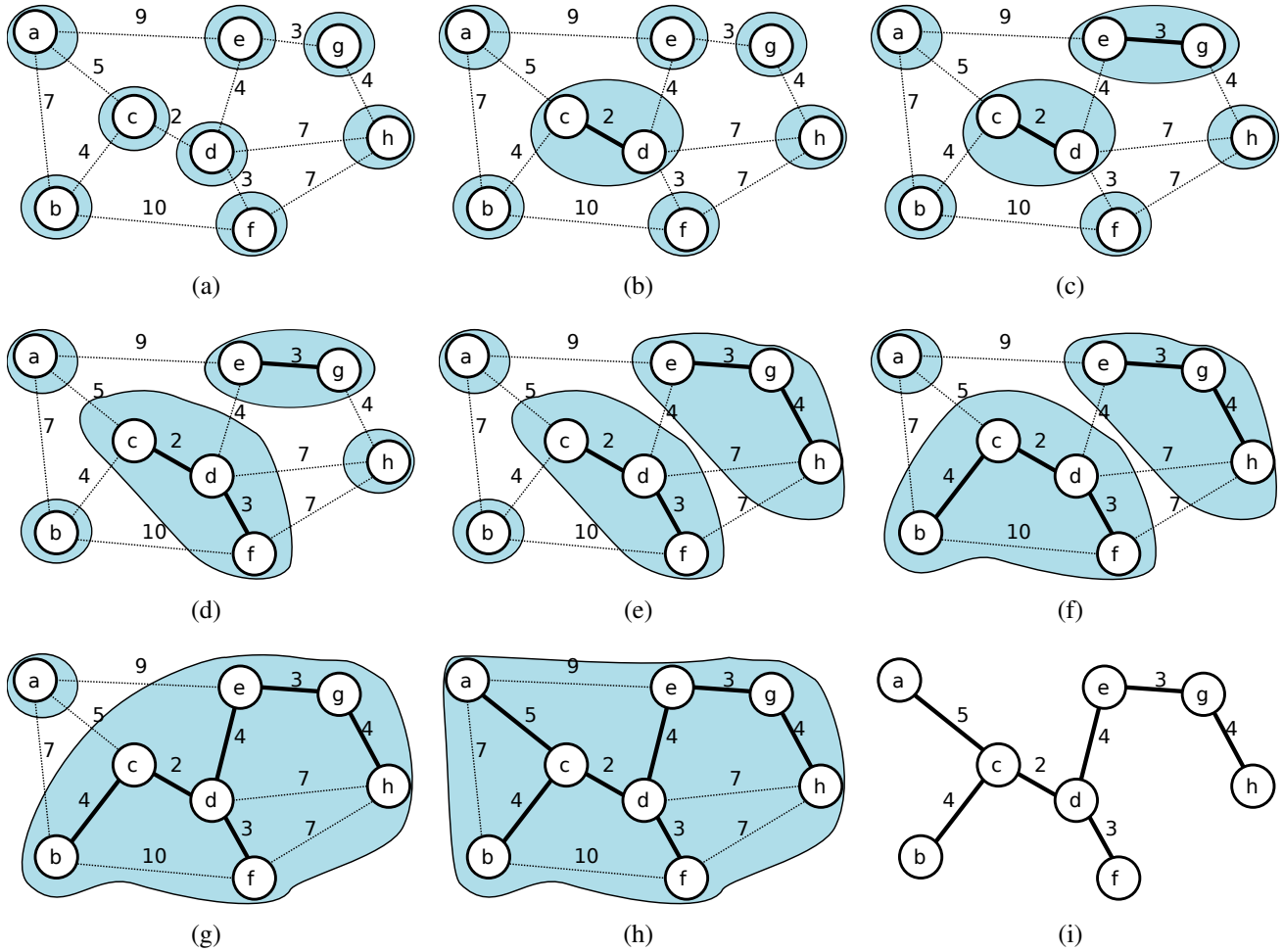


FIGURE 69 – Exemple d'exécution de l'algorithme de Kruskal

Références

[AVL62] G. Adelson-Velskii and E. M. Landis. An algorithm for the organization of information. Proceedings of the USSR Academy of Sciences, 146(3) :263–266, 1962.

[Gab05] Philippe J. Gabrini. Structures de données avancées avec la STL ; POO en C++. Loze-Dion, 2005.

[GTM11] Michael T. Goodrich, Roberto Tamassia, and David Mount. Data Structures & Algorithms in C++. Wiley, second edition, 2011.

[KW05] Elliot B. Koffman and Paul A. T. Wolfgang. Objects, abstraction, data structures and design using C++. Wiley, 2005.

[KW06] Elliot B. Koffman and Paul A. T. Wolfgang. Objects, Abstraction, Data Structures and Design : Using C++. Addison-Wesley, 2006.

[LJR12] Jesse Liberty, Bradley Jones, and Siddhartha Rao. Le langage C++ : Initiez-vous à la programmation C++ (Édition mise à jour avec la norme C++11). Pearson, 2012.

[Sav09] Walter Savitch. Problem Solving with C++. Addison-Wesley, 2009.

[Str10] Bjarne Stroustrup. The C++ Programming Language. Addison-Wesley, special edition, 2010.

[Tar72] Robert E. Tarjan. Depth-first search and linear graph algorithms. SIAM Journal on Computing, 1(2) :146–160, 1972.