Université de Liège

INFO0947

Compléments de Programmation

# Syllabus d'Exercices

Benoit Donnet, Simon Liénardy, Géraldine Brieven

9 avril 2024

# Table des matières

# Introduction

Le cours INFO0947 (Compléments de Programmation) fait suite au cours INFO0946 (Introduction à la Programmation). Les concepts vus, dès lors, dans le cours INFO0946 sont considérés comme connus et maîtrisés.

L'objectif du cours INFO0947 est de permettre d'étendre les connaissances en programmation sur deux plans :

1. Algorithmique. L'idée est d'améliorer les compétences de l'étudiant dans la compréhension et la mise en place d'algorithmes de complexité moyenne. A cette fin, l'approche constructive est mise en avant. L'idée est de se baser sur la spécification et l'Invariant de Boucle Formel d'un problème et sur des assertions intermédiaires pour construire le code. La notion de récursivité est aussi introduite, ainsi que la complexité des fonctions récursives.
2. Structures de Données. Les structures de données vues jusqu'à présent sont assez limitées. Durant le cours, nous allons augmenter nos connaissances en structures de données. En particulier, la notion de type abstrait sera mise en avant. Les structures de données abordées sont des structures linéaires (listes, piles, files).

Ce document se veut être un recueil d'exercices permettant à l'étudiant de pratiquer les concepts vus durant le cours théorique. La pratique est un aspect important (si pas le plus important) du cours INFO0947. C'est dans cette optique que ce syllabus a été rédigé. Les compétences désirées à l'issue du cours ne peuvent s'obtenir sans une pratique intense. La compréhension et l'écoute lors du cours théorique sont des conditions nécessaires mais nullement suffisantes. Il faut s'exercer encore et encore de façon à acquérir l'expérience et l'aisance nécessaires à la réussite de l'examen.

En plus des séances d'exercices, les étudiants devront réaliser, durant le quadrimestre, plusieurs travaux.

# Chapitre 1

# Raisonnement Mathématique

## 1.1 Rappels sur la Logique et Quantificateurs

### 1.1.1 Formalisation d'Énoncé

Si $p(x)$ signifie "$x$ est un slug", $q(x)$ signifie "$x$ aime le chocolat", $r(x)$ signifie "$x$ n'est pas un chat" et $k(x, y)$ signifie "$x$ est plus intéressant que $y$", comment exprimer les propositions suivantes sous forme d'expressions logiques ?

▷ **Exercice 1** Xavier est un slug.

▷ **Exercice 2** Aucun slug n'aime le chocolat.

▷ **Exercice 3** Si tous les chats aiment le chocolat, alors aucun chat n'est un slug.

▷ **Exercice 4** Xavier est plus intéressant que tous les autres slugs.

### 1.1.2 Formalisation d'Énoncé portant sur des Tableaux

Soient $n$ et $m \in \mathbb{N}$ et deux tableaux d'entiers $a[0 \dots n-1]$ et $b[0 \dots m-1]$. Dans les exercices qui suivent, il vous est demandé d'exprimer les propositions sous forme d'expressions logiques.

▷ **Exercice 1** Tous les éléments de $a[j \dots k]$ sont nuls.

▷ **Exercice 2** Tous les éléments pairs de $a[j \dots k]$ sont inférieurs à 10.

▷ **Exercice 3** Tous les nombres pairs de $a[j \dots k]$ se trouvent à des emplacements pairs.

▷ **Exercice 4** $x$ est le minimum de $a$.

▷ **Exercice 5** Si la valeur $x$ apparaît dans $a$, elle ne peut pas apparaître dans $b$.

▷ **Exercice 6** Si la valeur $x$ apparaît deux fois dans $b$, alors elle ne peut apparaître dans $a$.

▷ **Exercice 7** Si une valeur paire apparaît dans $a$, alors elle apparaît à la même place dans $b$.

## 1.2 Introduction de notations afin de formaliser un énoncé

Dans cette section, il vous est demandé, pour chaque exercice, de formaliser l'énoncé à l'aide d'une notation mathématique.

▷ **Exercice 1** Soit la notation suivante :

$$apparait(T, N, x) \equiv \exists i, 0 \leq i < N, T[i] = x$$

où $T$ est un tableau contenant $N$ entiers.

À l'aide de cette notation, réexprimer la proposition suivante (présentée dans la section précédente) : *"Si la valeur $x$ apparaît dans le tableau $a$ de taille $n$, elle ne peut pas apparaître dans le tableau $b$ de taille $m$."*

▷ **Exercice 2** Soit un tableau `T`, à `N` valeurs entières. Introduisez une notation pour indiquer que tous les éléments d'un sous-tableau de `T` respectent une propriété $p(x)$.

▷ **Exercice 3** Soit un tableau `T`, à `N` valeurs entières. Introduisez une notation pour indiquer le nombre d'éléments de `T` qui respectent une propriété $p(x)$.

▷ **Exercice 4** Soit un tableau `T`, à `N` valeurs entières (avec `N` pair). Introduisez une notation pour indiquer que `T` contient autant de valeurs paires que de valeurs impaires.

▷ **Exercice 5** Soit un sous-tableau $tab[min \dots max]$ de taille `size`.

- Introduisez une notation pour indiquer que $tab[min \dots max]$ contient au moins deux fois la valeur $val$.
- Utilisez ensuite ce prédicat ainsi que le prédicat $apparait(T, N, x)$ (défini dans l'exercice 1 de cette section) pour réexprimer la proposition suivante (présentée dans la section précédente) : *"Si la valeur x apparaît deux fois dans le tableau b de taille m, alors elle ne peut pas apparaître dans le tableau a de taille n."*

## 1.3 Preuve par Récurrence

### 1.3.1 Preuves de Base

▷ **Exercice 1** Prouver, par récurrence sur $n$, l'égalité suivante :

$$\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + \ldots + n^2 = \frac{n \times (n+1) \times (2n+1)}{6}, \forall n \in \mathbb{N}$$

▷ **Exercice 2** Prouver, par récurrence sur $n$, l'égalité suivante :

$$\sum_{i=0}^{n} x^i = \frac{x^{n+1} - 1}{x - 1}, \forall x \neq 1, \forall n \in \mathbb{N}$$

▷ **Exercice 3** Si les formules du type $p_i$ sont à valeurs dans $V = \{vrai, faux\}$, la formule $p_1 \Rightarrow p_2$ prend la valeur $vrai$ pour trois éléments de $V^2$ correspondant au couple $(p_1, p_2)$.

De même, la formule $(p_1 \Rightarrow p_2) \Rightarrow p_3$ prend la valeur $vrai$ pour 5 éléments de $V^3$ correspondant au triplet $(p_1, p_2, p_3)$.

Adoptons la notation $Q_n$ pour désigner la formule $(\ldots (p1 \Rightarrow p2) \Rightarrow \ldots) \Rightarrow p_n)$.

Il est demandé de prouver que, si $T_n$ désigne le nombre d'éléments de $V^n$ correspondant au $n$-uplet $(p_1, \ldots, p_n)$ qui donnent à la formule $Q_n$ la valeur $vrai$, on obtient l'égalité suivante :

$$T_{n+1} + T_n = 2^{n+1}, \forall n \geq 2$$

et que, par conséquent, on a si $n$ est pair :

$$T_N = \frac{2^{n+1} + 1}{3}$$

et si $n$ est impair :

$$T_n = \frac{2^{n+1} - 1}{3}$$

## 1.4 Relation d'Ordre

Les espaces suivants sont-ils bien fondés ou non ? On demande, dans les cas négatifs, de fournir un exemple de suite "infinie décroissante".

▷ **Exercice 1** $(\mathbb{N}, <)$ où $<$ représente l'ordre strict usuel sur $\mathbb{N}$.

▷ **Exercice 2** $(\mathbb{R}_0^+, \leq)$

▷ **Exercice 3** $(\mathbb{L}, \mathcal{R})$ où $\mathbb{L}$ désigne l'ensemble des listes finies de réels et $\mathcal{R}$ la relation binaire définie par

$$\forall l, l' \in \mathbb{L}, l \ \mathcal{R} \ l' \Leftrightarrow (longueur(l) < longueur(l'))$$

## 1.5 Induction

### 1.5.1 Notations

On considère des suites finies de naturels. On note ces suites par des lettres majuscules éventuellement indicées. On note, par simple juxtaposition les opérations de concaténation des suites et d'ajout d'un élément en tête d'une suite.

Par exemple, si $n = 3$, $S_1 = (0, 7, 4)$ et $S_2 = (3, 9)$, on a

$$\begin{cases} nS_1 = (3, 0, 7, 4) \\ S_1S_2 = (0, 7, 4, 3, 9) \\ nS_2S_1 = (3, 3, 9, 0, 7, 4) \end{cases}$$

On dit que les suites finies $S_1, \ldots, S_n$ (prises dans cet ordre) constituent une décomposition de la suite finie $S$ si et seulement si $S = S_1 \ldots S_n$.

### 1.5.2 Définitions et exemples

On appelle *suite équilibrée* toute suite construite par applications répétées de la règle suivante. Si $n$ est un naturel et si $S_1, \ldots, S_n$ sont des suites équilibrées déjà construites, la suite $nS_1 \ldots S_n$ est équilibrée.

En particulier, la suite $(0)$ est équilibrées car on l'obtient en appliquant la règle $n = 0$. Exemples de suites équilibrées :

$(2, 0, 0)$,

$(1, 1, 0)$,

$(5, 0, 1, 0, 2, 0, 0, 3, 1, 0, 0, 0, 0)$

▷ **Exercice 1** On demande de prouver les trois propriétés suivantes :

- La longueur d'une suite équilibrée est égale à la somme de ses éléments augmentée de 1. Autrement dit, si $S = (s_1, \ldots, s_n)$ est équilibrée, on a $n = s_1 + s_2 + \ldots + s_n + 1$.
- Soient $n \in \mathbb{N}, m \in \mathbb{N}_0$ et $S$ et $S'$ deux suite finies quelconques telles que $S = nS'$. Alors, $S$ est décomposable en $m$ suites équilibrées si et seulement si $S'$ est décomposable en $m+n-1$ suites équilibrées.
- Une suite finie de nombres naturels admet au plus une décomposition en suites équilibrées.

# Chapitre 2

# Construction de Programme (en reposant sur des assertions)

## 2.1 Assertions

### 2.1.1 Évaluation

▷ **Exercice 1** Évaluez les assertions $A_1$ à $A_6$ par rapport aux situations concrètes $S_j^1 \wedge S_k^2 \wedge S^3$, avec $1 \leq j \leq 5$ et $1 \leq k \leq 2$, $a[0 \ldots 5]$ étant un tableau d'entiers.

$S_1^1$ : $i$ : ? $S_1^2$ : $x$ : ? $S^3$ : $a$ : 5 | -3 | 2 | 5 | -3 | 2 (n=6)
$S_2^1$ : $i$ : -1 $S_2^2$ : $x$ : 4
$S_3^1$ : $i$ : 3
$S_4^1$ : $i$ : 5
$S_5^1$ : $i$ : 6

$A_1 : x$ initialisée
$A_2 : a[i]$ initialisé
$A_3 : (0 \leq i \leq n-1) \wedge (a[i] == x)$
$A_4 : (-1 \leq i \leq n) \wedge (a[i] == x)$
$A_5 : (-1 \leq i \leq n-1) \wedge (\forall j : i < j \leq n-1 : a[j] \neq x)$
$A_6 : (-1 \leq i \leq n-1) \wedge (\forall j : i \leq j \leq n-1 : a[j] \neq x)$

À chaque fois, on demande de fournir trois réponses : **T** (pour True), **F** (pour False) ou **?** (pour Undefined).

Pour ce faire, on complètera un tableau du même style que le tableau 2.1 (les couples en ordonnées représentent les différentes valeurs pour les paramètres $j$ et $k$).

## 2.2 Expression Schématique d'Assertions

▷ **Exercice 1** Soit $a[0 \ldots n-1]$ un tableau d'entiers. Exprimez les assertions suivantes à l'aide d'un schéma :

1. $a = a_0 \ \wedge \ 0 \leq i \leq n-1 \ \wedge \ \forall j, 0 \leq j \leq i, a[j] \neq x$

| | $A_1$ ... $A_6$ |
|---|---|
| (1, 1) | |
| (1, 2) | |
| (2, 1) | |
| ⋮ | |
| (5, 1) | |

TABLE 2.1 – Évaluation d'Assertion

2. $a = a_0 \ \wedge \ (0 \leq p \leq q \leq n) \ \wedge \ (\forall i, 0 \leq i \leq p-1, a[i] \leq x) \ \wedge \ (\forall i, q < i \leq n-1, a[i] > x)$
3. $a = a_0 \ \wedge \ (0 \leq i \leq j < n) \ \wedge \ (\forall k, (0 \leq k < i) \vee (j < k < n-1), a[k] \leq a[k+1])$

▷ **Exercice 2** Exprimez le schéma suivant sous forme d'assertion.

a :

| 0 | $i$ $n-1$ |
|---|---|
| trié (croissant) | vide |

## 2.3 Spécifications Formelles

### 2.3.1 Tableaux

On se situe dans un programme contenant les déclarations suivantes :

```
1 const int N = ...;
2 const int M = ...;
3
4 int a[N];
5 int b[M];
```

Spécifiez complètement des procédures (ou fonctions) C correspondant aux problèmes suivants. Il vous appartient justement de les éliminer. Si nécessaire, n'hésitez pas à introduire des notations mathématiques. Adoptez l'approche la plus formelle possible.

▷ **Exercice 1** Rechercher l'emplacement d'une valeur donnée dans un tableau d'entiers `a[0...N-1]`.

▷ **Exercice 2** Rechercher l'emplacement d'une valeur donnée dans une portion du tableau `a` d'entiers.

▷ **Exercice 3** Déplacer une portion du tableau `a` d'un nombre de pas donné vers la droite.

▷ **Exercice 4** Construire un tableau d'entiers strictement trié correspondant à l'union de deux tableaux d'entiers strictement triés.

### 2.3.2 Procédures/Fonctions

Spécifiez (de manière formelle) complètement des fonctions (ou procédures) C correspondant aux problèmes suivants. Si des définitions s'avèrent nécessaires ou simplement pertinentes, n'hésitez pas à introduire de « nouvelles notations mathématiques ».

▷ **Exercice 1** Trouver la plus longue portion triée d'un tableau d'entiers.

▷ **Exercice 2** 1

## 2.4 Assertions Intermédiaires

Voici une série d'extraits de code. En supposant que les assertions intermédiaires indiquées sont vraies, complétez les assertions manquantes (indiquées par des $\{\ldots\}$). Si ce n'est pas précisé, les variables sont entières.

▷ **Exercice 1**

```
// {a = ??}
a = 1;
// {...}
```

▷ **Exercice 2**

```
// {a < 3}
a += 1;
// {...}
```

▷ **Exercice 3**

```
// {x > 4}
x /= 2;
// {...}
```

▷ **Exercice 4**

```
// {2 ≤ b < 7}
b *= 3;
// {...}
```

▷ **Exercice 5**

```
// {Predicat(i)}
++i;
// {...}
```

▷ **Exercice 6**

```
// {∀k, 0 ≤ k < d, vect[k] = 0}
vect[d] = 0;
// {...}
```

▷ **Exercice 7**

```
// {N > 0, tab[0, N − 1] init}
for(int j = 0; j < N; ++j){
    tab[j] = N;
}
// {...}
```

▷ **Exercice 8**

```
// {m = max_{0≤k<i} t[k]}
if (t[i] > m) {
    // {...}
    m = t[i]
    // {...}
} else {
    // {...}
}
// {...}
++i;
// {...}
```

▷ **Exercice 9**

```
// {N > 0, t[0, N − 1] init}
int i = 0; j = N-1
while(j >= i) {
    // {(∀k, 0 ≤ k < i, t[k] = k mod 2) ∧ (∀k, j < k < N, t[k] = k mod 2)}
    t[j] = j % 2; t[i] = i % 2;
    // {...}
    j--; i++;
    // {...}
}
// {...}
```

▷ **Exercice 10**

```
int sum = 0, i = 0;

while(i <= N) {
    // {sum = Σ_{j=0}^{i-1} j}
    sum += i;
    // {...}
    ++i;
    // {...}
}
// {...}
```

## 2.5 Invariant Formel

Dans les exercices qui suivent, soyez le plus formel possible.

▷ **Exercice 1** Soit le bout de code suivant :

PréCondition : $a$ initialisé $\wedge$ $0 < n$

```
int i = -1;
int ou = 0;
while(!ou && i < n - 1){
  ou = a[i + 1];
  i++;
}
```

PostCondition : $a$ inchangé $\wedge$ $ou = \bigvee_{j=0}^{n-1} a[j] \wedge i$ contient la position du premier élément non-nul dans a, s'il y en a un. Sinon, $i$ vaut $n$

Remarque : la notation $\bigvee_{i=0}^{n-1} a[i]$ est équivalente à $a[0] \vee a[1] \vee \ldots a[n-1]$. En particulier, on a $\bigvee_{i=0}^{-1} a[i] =$ False.

Donnez un Invariant Formel pour le bout de code ci-dessus. Si nécessaire, n'hésitez pas à passer par un Invariant Graphique pour ensuite le traduire sous forme d'Invariant Formel.

▷ **Exercice 2** Soit `b[0 ...m-1]`, un tableau d'entiers.

Pré : $b$ et $x$ initialisés

```
int i = 0, nbre = 0;
while(i < m){
  if(b[i] == x)
    nbre++;
```

```
5   i++;
6 }
```

Post : $b, x$ inchangés; $nbre = \#\{k | 0 \leq k \leq m-1; b[k] = x\}$

Donnez un Invariant Formel pour le bout de code ci-dessus. Si nécessaire, n'hésitez pas à passer par un Invariant Graphique pour ensuite le traduire sous forme d'Invariant Formel.

## 2.6 Construction de Programme

### 2.6.1 Problèmes Élémentaires

Spécifiez et construisez, en suivant la méthode constructive vue au cours, des procédures (ou des fonctions) C permettant de résoudre les problèmes ci-après. Pour chacun de ces problèmes, on demande de trouver une résolution ne parcourant qu'une seule fois chaque tableau.

Les variables `a`, `b`, et `c` sont des tableaux de taille $N$ contenant des entiers.

Remarque : si de nouvelles notations mathématiques s'avèrent utiles, n'hésitez pas à les introduire.

▷ **Exercice 1** Trouver le maximum d'un préfixe donné du tableau `a`.

▷ **Exercice 2** Trouver l'emplacement de la plus grande valeur strictement inférieure à une valeur donnée dans un préfixe de `a` trié par ordre croissant.

▷ **Exercice 3** Soient un préfixe de `a` et un préfixe de `b`, tous deux strictement triés (par ordre croissant). Construire un tableau correspondant à « l'intersection » de ces deux préfixes (i.e., contenant les valeurs communes aux deux préfixes et uniquement elles).

▷ **Exercice 4** Tester si un préfixe donné de `b` constitue une sous-suite d'un préfixe donné de `a` (pour rappel si $s = (s_1, s_2, \ldots, s_n)$, on appelle sous-suite de $s$ toute suite de forme $(s_{i_1}, \ldots, s_{i_m})$ avec $0 \leq m \leq n$ et $1 \leq i_1 < \ldots < i_m \leq n$ et où $s_{i_1}, \ldots, s_{i_m}$ apparaissent dans le même ordre dans la suite donnée $s$).

▷ **Exercice 5** Déterminer un sous-tableau de somme maximale dans un préfixe donné de `a`.

Exemple, pour le préfixe de `a` suivant :

| 0 | | | | | | | | | 9 | N-1 |
|---|---|---|---|---|---|---|---|---|---|---|
| -3 | 1 | 4 | -2 | -1 | 5 | -7 | 0 | 2 | -1 | |

le sous-tableau `a[1 ...5]` convient avec une somme de 7.

### 2.6.2 Tri par Insertion

Il s'agit de construire un programme implémentant le *tri par insertion* pour un tableau d'entiers. Ce tri fonctionne comme suit : à chaque étape le tabealu est divisé en deux parties ; la seconde partie est inchangée depuis le début tandis que la première est triée. Il s'agit alors « tout simplement » d'insérer le premier élément de la seconde partie « à sa place » dans la première partie.

Par exemple, considérons le tableau (de taille 5) suivant :

| 0 | | | | 4 |
|---|---|---|---|---|
| 3 | 2 | 1 | 1 | 0 |

On obtiendra successivement les résultats suivants :

- 1ère itération

| 0 | | | | 4 |
|---|---|---|---|---|
| 3 | 2 | 1 | 1 | 0 |

- 2ème itération

| 0 | | | | 4 |
|---|---|---|---|---|
| 2 | 3 | 1 | 1 | 0 |

- 3ème itération

| 0 | | | | 4 |
|---|---|---|---|---|
| 1 | 2 | 3 | 1 | 0 |

- 4ème itération

| | | | | |
|---|---|---|---|---|
| 0 | | | | 4 |
| 1 | 1 | 2 | 3 | 0 |

- 5ème itération

| | | | | |
|---|---|---|---|---|
| 0 | | | | 4 |
| 0 | 1 | 1 | 2 | 3 |

▷ **Exercice 1** Spécifier précisément le problème.

▷ **Exercice 2** Décomposer le problème en sous-problèmes.

▷ **Exercice 3** Spécifier précisément chaque sous-problèmes.

▷ **Exercice 4** Construire le morceau de programme correspondant à chaque sous-problème en utilisant l'approche constructive vue au cours.

▷ **Exercice 5** Rédiger le morceau de programme final.

# Chapitre 3

# Introduction à la Complexité

Ce chapitre aborde un sujet relatif à l'informatique théorique. L'idée est d'introduire les notions permettant d'évaluer, de manière générale, les performances d'un programme et de comparer différentes solutions entre elles afin de s'orienter vers la plus efficace. La notion derrière tout cela est la *complexité*. Dans le cadre du cours, nous nous intéressons à la *complexité temporelle* dans le *pire des cas*. Ceci nous est indiqué par une notation spéciale, le "$O$" (ou notation de *Landau*). La Sec. 3.1 nous permet d'apprendre à manipuler cette notation tandis que la Sec. 3.2 nous permet d'apprendre à analyser la complexité d'un programme.

## 3.1 Notation "$O$"

### 3.1.1 Grandeurs

▷ **Exercice 1**

1. Comparez $10^{100}$ et $100^{10}$.
2. Comparez $10^{(10)^{10}}$ et $(10^{10})^{10}$.

▷ **Exercice 2** Rangez les fonctions suivantes par ordre croissant de grandeur :

$$n,\ n \times \log(n),\ \sqrt{n},\ \log(n),\ \sqrt{n} \times \log^2(n)$$
$$n^2,\ (3/2)^n,\ n^{10},\ \log^2(n),\ e^{n^2},\ 1/3^n$$

▷ **Exercice 3** Complétez les tableaux suivants.

Dans le Tableau 3.1.1, il s'agit de calculer le nombre d'opérations nécessaire pour exécuter le programme en fonction de sa complexité et de la taille d'instance du problème traité. Dans le Tableau 3.2, il s'agit de calculer le temps nécessaire à cela en supposant qu'on dispose d'un ordinateur capable de réaliser $10^9$ opérations/seconde. Enfin, dans le Tableau 3.3, on calcule la plus grande instance du problème traitable dans le temps imparti.

Rappels : $1\mu\text{s} = 10^{-6}\text{s}$ ; $1\text{ ns} = 10^{-9}\text{s}$ ; $10^{x^y}$ et $\sqrt{n} = n^{1/2}$.

### 3.1.2 Comportements Asymptotiques

▷ **Exercice 1** Déterminez si les affirmations suivantes sont vraies ou fausses. Justifiez (sur base de la définition de la notation de Landau) votre réponse.

1. $2 \times n + 3 \in O(n)$
2. $\log^{145} n \in O(\log(n))$
3. $2^n + \log(n) \in O(n^2)$
4. $n^2 + 4 \times n^2 + \log(n) \in O(2^n)$

| **Complexité** | **n=10** | **n=100** | **n=1000** |
|---|---|---|---|
| $n$ | | | |
| $n^2$ | | | |
| $n^3$ | | | |
| $2^n$ | | | |
| $\sqrt{n}$ | | | |
| $\sqrt[3]{n}$ | | | |
| $\log(n)$ | | | |

TABLE 3.1 – Nombre d'opérations

| **Complexité** | **n=10** | **n=100** | **n=1000** |
|---|---|---|---|
| $n$ | | | |
| $n^2$ | | | |
| $n^3$ | | | |
| $2^n$ | | | |
| $\sqrt{n}$ | | | |
| $\sqrt{3}n$ | | | |
| $\log(n)$ | | | |

TABLE 3.2 – Temps nécessaire à $10^9$ opérations/seconde

| **Complexité** | **1 sec.** | **1 heure** | **1 an** |
|---|---|---|---|
| $n$ | | | |
| $n^2$ | | | |
| $n^3$ | | | |
| $2^n$ | | | |
| $\sqrt{n}$ | | | |
| $\sqrt{3}n$ | | | |
| $\log(n)$ | | | |

TABLE 3.3 – Plus grande instance faisable à $10^9$ opérations/seconde

5. $2 \times n^7 + 4 \times n^6 + 5 \times n^5 + 2 \times n^3 + 2 \times n \in O(n^7)$

▷ **Exercice 2** Quelle est la classe de complexité de la fonction suivante :

$$T(n) = 2 \times n^3 + 4 \times n^2 + 2^3$$

Démontrez.

▷ **Exercice 3** Montrez que les assertions suivantes sont exactes :

- $n \times (n-1)$ est en $O(n^2)$
- $\max(n^3, 10 \times n^2)$ est en $O(n^3)$

▷ **Exercice 4** Soient $f(x)$ et $g(x)$, des fonctions positives asymptotiquement. On suppose que $S(n) \in O(f(n))$ et $T(n) \in O(g(n))$. Montrez que

$$\text{si } f(n) \in O(g(n)), \text{ alors } S(n) + T(n) \in O(g(n)).$$

▷ **Exercice 5** Soient $f(x)$ et $g(x)$, des fonctions positives asymptotiquement. On suppose que $S(n) \in O(f(n))$ et $T(n) \in O(g(n))$. Montrez que

$$\text{si } f(n) \in O(g(n)), \text{ alors } S(n) \times T(n) \in O(f(n) \times g(n))$$

▷ **Exercice 6** Peut-on écrire :

1. $2^{n+1} \in O(2^n)$ ?
2. $2^{2^n} \in O(2^n)$ ?

▷ **Exercice 7** Soient $f(x)$ et $g(x)$, des fonctions asymptotiquement strictement positives. Peut-on affirmer que $f(n) \in O(g(n))$ implique que $(g(n) \in O(f(n)))$. Justifiez !

## 3.2 Analyse de Complexité

Pour chacun des exercices suivants, il vous est demandé d'effectuer une analyse complète de la complexité du code. Le raisonnement que vous devez donner doit être **formel** (i.e., mathématique). Nous vous suggérons de procéder de la sorte :

1. construire de la fonction $T(\cdot)$ quantifiant les instructions élémentaires du programme. En particulier :
   (a) découper le programme en différentes zones ;
   (b) établir la fonction $T(\cdot)$ de chaque zone en appliquant les différentes règles (Chap. 4, Slides 20 $\rightarrow$ 27) ;
   (c) combiner les différentes fonctions $T(\cdot)$ en une seule fonction (Règle 6 – Chap. 4, Slide 27).
2. borner la fonction $T(\cdot)$ à l'aide de la fonction de Landau.

▷ **Exercice 1**

```c
#include <stdio.h>

int main(){
  int i, n;
  scanf("%d", &n);

  for(i=0; i<n; i++)
    printf("Bonjour!");
}//fin programme
```

Donnez la complexité du programme. Justifiez votre réponse.

▷ **Exercice 2**

```c
#include <stdio.h>

int main(){
  int a, b, r, i;

  scanf("%d%d", &a, &b);

  r = 1;
  for(i=0; i<b; i++)
    r = r*a;

  printf("%d\n", r);
}//fin programme
```

Que calcule ce programme ? Donnez la complexité du programme. Justifiez votre réponse.

▷ **Exercice 3**

```c
#include <stdio.h>

int main(){
  int i, j, n;

  scanf("%d", &n);

  for(i=1; i<=n; i++)
    for(j=1; j<=n; j++)
      printf("Salut!\n");
}//fin programme
```

Donnez la complexité du programme. Justifiez votre réponse.

▷ **Exercice 4**

```c
#include <stdio.h>

int main(){
  int i, n;

  scanf("%d", &n);

  i = 1;
  while(i<=n){
    printf("Salut!\n");
    i *= 2;
  }//fin while
}//fin programme
```

Donnez la complexité du programme. Justifiez votre réponse.

▷ **Exercice 5**

```c
#include <stdio.h>

int main(){
  int i, j, n;

  scanf("%d", &n);

  i = 1;
  while(i<=n){
   for(j=1; j<=i; j++)
      printf("Salut\n");

   i *= 2;
  }//fin while
}//fin programme
```

Donnez la complexité du programme. Justifiez votre réponse.

▷ **Exercice 6**

```c
#include <stdio.h>

int main(){
  int i, j, n;

  scanf("%d", &n);

  for(j=1; j<=n; j++){
    i = 1;
    while(i<=j){
      printf("Salut\n");
      i *= 2;
    }//fin while
  }//fin for
}//fin programme
```

Donnez la complexité du programme. Justifiez votre réponse.

▷ **Exercice 7**

```c
#include <stdio.h>

int main(){
  int a, b, p, x, i;

```

```
6   scanf("%d%d", &a, &b);
7
8   p = 1;
9   x = a;
10  i = b;
11
12  while(i!=0){
13    if(i%2){
14      p = p*x;
15      i--;
16    }
17
18    x = x*x;
19    i = i/2;
20  }//fin while - i
21
22  printf("%d\n", p);
23 }//fin programme
```

Donnez la complexité du programme. Justifiez votre réponse.

# Chapitre 4

# Récursivité

## 4.1 Compréhension

▷ **Exercice 1** Soit la fonction suivante

```
int f(int n){
  if(n == 0)
    return 1;
  else
    return f(n + 1);
}//fin f()
```

La fonction `f()` décrite ci-dessus est exécutée avec une valeur `n` $\geq 0$. Exécutez la fonction sur un exemple. Justifiez si cette fonction se termine, en général, ou non. Si oui, déterminez ce qu'elle fait ou calcule.

▷ **Exercice 2** Soit la fonction suivante

```
int f(int n){
  if(n == 0)
    return 0;
  else{
    int result = f(n - 1);
    result += n;

    return result;
  }
}//fin f()
```

La fonction `f()` décrite ci-dessus est exécutée avec une valeur `n` $< 0$. Exécutez la fonction sur un exemple. Justifiez si cette fonction se termine, en général, ou non. Si oui, déterminez ce qu'elle fait ou calcule.

▷ **Exercice 3** Soit la fonction suivante :

```
int f(int n){
  if(n <= 1)
    return 1;
  else
    return 1 + f(n - 2);
}//fin f()
```

La fonction `f()` décrite ci-dessus est exécutée avec une valeur `n` $\geq 0$. Exécutez la fonction sur un exemple. Justifiez si cette fonction se termine, en général, ou non. Si oui, déterminez ce qu'elle fait ou calcule.

## 4.2 Construction de Fonctions Récursives

### 4.2.1 Exercices Simples

Pour tous les exercices qui suivent, vous devez appliquer, au moment de l'écriture du code, l'approche constructive telle que vue au cours.

▷ **Exercice 1** Exprimez de manière récursive la somme des nombres de 1 à n, avec $n > 0$. Spécifiez et construisez ensuite la fonction C correspondante.

▷ **Exercice 2** Soit $a$, un entier et $n$ un entier positif ou nul. On se propose de calculer $a^n$ en utilisant les 3 propriétés suivantes :

$$1.\ a^n = \begin{cases} a \times a^{n-1} & \text{si n>0,} \\ 1 & \text{si n=0.} \end{cases}$$

$$2.\ a^n = \begin{cases} (a^2)^{\lfloor n/2 \rfloor} & \text{si n>0 est pair,} \\ a \times (a^2)^{\lfloor n/2 \rfloor} & \text{si n est impair,} \\ 1 & \text{si n=0.} \end{cases}$$

$$3.\ a^n = \begin{cases} (a^{\lfloor n/2 \rfloor})^2 & \text{si n>0 est pair,} \\ a \times a^{\lfloor n/2 \rfloor})^2 & \text{si n est impair,} \\ 1 & \text{si n=0.} \end{cases}$$

Spécifiez et construisez, pour chacune de ces trois propriétés, une fonction récursive qui retourne $a^n$.

▷ **Exercice 3** Spécifiez et construisez une fonction récursive qui calcule l'addition de deux entiers $x$ et $y$ en utilisant uniquement les opérations $+1$ et $-1$. On rappelle que

$$x + y = \begin{cases} x & \text{si y=0,} \\ (x+1) + (y-1) & \text{si y>0.} \end{cases}$$

Écrivez ensuite un programme qui calcule et affiche la somme de 4 et 3. Enfin, représentez la pile d'appels à l'exécution du programme.

### 4.2.2 Suite de Lucas

On considère la suite de Lucas définie par $L_0 = 2, L_1 = 1$ et, pour $n > 0$, $L_{n+2} = L_{n+1} + L_n$.
Spécifiez et écrivez une fonction récursive qui calcule $L_n$ en fonction de $n$. On admet que

$$L_n = \left(\frac{1+\sqrt{5}}{2}\right)^n + \left(\frac{1-\sqrt{5}}{2}\right)^n$$

### 4.2.3 Algorithmes sur des Tableaux

Pour tous les exercices qui suivent, vous devez appliquer, au moment de l'écriture du code, l'approche constructive telle que vue au cours.

▷ **Exercice 1** Spécifiez et construisez une fonction récursive qui retourne la valeur minimum d'un tableau d'entiers `t`.

▷ **Exercice 2** Spécifiez et construisez une fonction récursive renvoyant la position d'un entier donné dans un tableau `t` d'entiers donné (-1 si le nombre ne s'y trouve pas).

▷ **Exercice 3** Spécifiez et écrivez une fonction récursive qui détermine si une matrice carrée (d'entiers) donnée est symétrique, en vous basant sur le fait qu'une matrice carrée est symétrique si sa première ligne est égale à sa première colonne et si le reste de la matrice est symétrique.

## 4.3 Calcul de Complexité

▷ **Exercice 1** Soit la fonction récursive suivante :

```
int fonction(int n){
  if (n == 0)
    return 2;
  else
    return 2* fonction(n - 1);
}//fin fonction()
```

Déterminez le plus formellement possible la complexité de cette fonction.

▷ **Exercice 2** Soit la fonction récursive suivante :

```
void fonction(float x, float y, float z){
  y = 2 * z;

  if (x > x / (y - z)){
    x -= 2;
    y /= 4;
    z /= 5;

    fonction(x, y, z);
  }
}//fin fonction()
```

Déterminez le plus formellement possible la complexité de cette fonction (Envisagez d'abord le cas où x, y et z sont positifs. Envisagez par la suite les autres cas).

▷ **Exercice 3** Soit la fonction récursive suivante :

```
int fonction(int n){
  if(n == 0)
    return 2;
  else if (n == 1)
    return 1;
  else
    return fonction(n - 1) + fonction(n - 2);
}//fin fonction()
```

Déterminez le plus formellement possible la complexité de cette fonction.

▷ **Exercice 4** Donnez la complexité pour toutes les fonctions récursives que vous avez définies à la Sec. 4.2.

# Chapitre 5

# Types Abstraits

## 5.1 Définition de Types Abstraits

### 5.1.1 TAD Date

Une date est un triplet de nombres ($\in \mathbb{N}$) où le premier indique le jour, le deuxième le mois, et le troisième l'année.

▷ **Exercice 1** Spécifiez complètement le TAD `Date`. On doit pouvoir, sur base d'une date, obtenir le jour, le moi, l'année, et, enfin, passer à la date du lendemain.

▷ **Exercice 2** Proposez une structure de données concrètes permettant d'implémenter une `Date`.

### 5.1.2 Les Entiers Naturels

On désire représenter les entiers naturels. La suite d'entiers est infinie et dénombrable.

▷ **Exercice 1** Spécifiez complètement le TAD `Natural` qui représente les entiers naturels. On doit pouvoir créer un entier naturel, déterminer son successeur, son prédécesseur, additionner deux entiers et, enfin, déterminer si un entier naturel est égal à 0.

Attention, il est important que votre spécification soit complète et consistante. Il est donc demandé d'appliquer la technique vue au cours (i.e., composer tous les observateurs avec toutes les opérations internes via un tableau à double entrée).

### 5.1.3 Ensemble d'entiers positifs

On souhaite représenter un *ensemble d'entiers positif* (i.e., nombre naturel). Ce type devra disposer des opérations suivantes :

- créer un ensemble vide ;
- déterminer si un ensemble est vide ;
- déterminer le cardinal (i.e., le nombre d'éléments) de l'ensemble ;
- déterminer si un ensemble est un sous-ensemble d'un autre ;
- déterminer si un élément appartient ou non à un ensemble ;
- déterminer si deux ensembles sont égaux ;
- déterminer l'union de deux ensembles d'entiers ;
- déterminer l'intersection de deux ensembles d'entiers ;
- ajouter un entier à l'ensemble ;
- retirer un entier de l'ensemble ;

**À titre d'exercice supplémentaire** Vous pouvez ajouter d'autres opérations sur les ensembles.

▷ **Exercice 1** Spécifiez complètement le TAD `Ensemble` qui représente les ensembles d'entiers positifs.

Attention, il est important que votre spécification soit complète et consistante. Il est donc demandé d'appliquer la technique vue au cours (i.e., composer tous les observateurs avec toutes les opérations internes via un tableau à double entrée).

### 5.1.4 Polynômes

Dans cet exercice, nous vous demandons de réaliser un type abstrait pour un **polynôme**. Pour simplifier les choses, nous ne considérons que les polynômes à une variable et à coefficients réels. En mathématiques, un tel polynôme $P$ s'écrit :

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x^1 + a_0$$

où $n$ est un entier naturel et les $a_i$ $(0 \leq i \leq n)$ sont les coefficients réels. Le polynôme $P$ est constitué des **monômes** $a_i x^i$ $(0 \leq i \leq n)$, chaque monôme $a_i x^i$ étant défini par le réel $a_i$, son coefficient, et par l'entier $i$, son degré.

Sur base de cette définition, il nous semble évident qu'il est nécessaire de définir deux types abstraits, l'un décrivant un monôme, l'autre un polynôme.

Le type monôme devra disposer des opérations suivantes :

- créer un monôme sur base de son coefficient et de son degré ;
- retourner le coefficient d'un monôme
- retourner le degré d'un monôme
- additionner deux monômes
- soustraire deux monômes
- multiplier deux monômes
- diviser deux monômes

Le type polynôme devra disposer des opérations suivantes :

- créer un polynôme "nul"
- ajouter un monôme à un polynôme
- fournir le monôme de plus haut degré
- supprimer le monôme de plus haut degré
- tester si un polynôme est "nul"
- déterminer le degré d'un polynôme
- additionner deux polynômes
- multiplier deux polynômes
- calculer la valeur du polynôme en un point donné
- dériver un polynôme

▷ **Exercice 1** Spécifiez complètement les TAD `Monome` et `Polynome`.

Attention, il est important que votre spécification soit complète et consistante. Il est donc demandé d'appliquer la technique vue au cours (i.e., composer tous les observateurs avec toutes les opérations internes via un tableau à double entrée).

## 5.2 Exercice Complet

### 5.2.1 Le Type Abstrait de Données `Complex`

On désire définir un type abstrait permettant de manipuler un *nombre complexe*. Ce type abstrait sera appelé `Complex`.

Le type défini devra comporter des opérations pour additionner, soustraire, multiplier, et diviser deux nombres complexes, pour calculer le module et le conjugué d'un nombre complexe, pour indiquer la partie réelle et la partie imaginaire d'un nombre complexe ainsi qu'un constructeur donnant, à partir de deux réels $a$ et $b$, le nombre complexe $a + bi$.

▷ **Exercice 1** Spécifiez complètement le type abstrait `Complex`.

Attention, il est important que votre spécification soit complète et consistante. Il est donc demandé d'appliquer la technique vue au cours (i.e., composer tous les observateurs avec toutes les opérations internes via un tableau à double entrée).

▷ **Exercice 2** Proposez une structure de données permettant de représenter le type abstrait `Complexe`.

▷ **Exercice 3** Écrivez le *header* C pour le type abstrait `Complex`. N'oubliez pas de spécifier, formellement, chaque fonction/procédure.

▷ **Exercice 4** Écrivez le module C implémentant le header défini à l'exercice 3.

▷ **Exercice 5** Écrivez un programme utilisant le type abstrait `Complex`.

### 5.2.2 Les TADs représentant une carte (`Card`) et une séquence de cartes (`CardsSeq`)

On désire définir 2 types abstraits permettant de manipuler une *séquence de cartes*. Le premier type abstrait à définir est `Card`. Ce type est caractérisé par un nombre (de 1 à 13). Il doit être possible de connaitre ce nombre à partir d'une carte donnée et savoir si 2 cartes données sont consécutives (le nombre associé à la seconde devant suivre celui de la première).

Le second type abstrait sera appelé `CardsSeq`. Ce type devra comporter des opérations pour ajouter ou retirer une carte à la séquence et, en particulier, ajouter une carte en fin de séquence (le numéro de cette carte ajoutée devant suivre celui de la dernière carte). On voudra également pouvoir connaitre le nombre de cartes composant la séquence et la somme des nombres portés par les différentes cartes. Enfin, on voudra savoir si l'ensemble des cartes sont ordonnées de manière croissante et on voudra accéder à une carte particulière, sur base de sa position.

▷ **Exercice 1** Spécifiez complètement les types abstraits `Card` et `CardsSeq`.

Attention, il est important que votre spécification soit complète et consistante. Il est donc demandé d'appliquer la technique vue au cours (i.e., composer tous les observateurs avec toutes les opérations internes via un tableau à double entrée).

▷ **Exercice 2** Proposez une structure de données permettant de représenter les types abstraits `Card` et `CardsSeq`.

▷ **Exercice 3** Écrivez le *header* C pour le type abstrait `CardsSeq`. N'oubliez pas de spécifier, formellement, chaque fonction/procédure.

### 5.2.3 Le TAD représentant un arbre binaire (`BT`)

On désire définir un type abstrait permettant de manipuler un *arbre binaire*. Ce type abstrait sera appelé `BT` [1]. On considère que le type de données `Node` [2] est déjà défini et mis à notre

1. Acronyme de Binary Tree
2. Représente un noeud dans un arbre. Un Node pourrait simplement contenir comme information un nombre, mais il pourrait tout aussi bien représenter un objet plus complexe avec différents attributs. Ce n'est pas important ici.

disposition.

Le type défini devra comporter des opérations pour ajouter ou retirer un noeud, inverse l'ensemble de l'arbre (l'enfant de gauche (resp. droite) devient l'enfant de droite (resp. gauche)) et rendre un arbre complet. On voudra également pouvoir connaitre la profondeur de l'arbre, obtenir sa racine, savoir s'il est complet (i.e., tous les noeuds possèdent zéro ou deux enfants) et connaitre le nombre de noeuds et de feuilles (les feuilles étant les extrémités de l'arbres, sans enfant).

▷ **Exercice 1** Spécifiez complètement le type abstrait `BT`.

Attention, il est important que votre spécification soit complète et consistante. Il est donc demandé d'appliquer la technique vue au cours (i.e., composer tous les observateurs avec toutes les opérations internes via un tableau à double entrée).

▷ **Exercice 2** Proposez une structure de données permettant de représenter le type abstrait `BT`.

▷ **Exercice 3** Écrivez le *header* C pour le type abstrait `BT`. N'oubliez pas de spécifier, formellement, chaque fonction/procédure.

▷ **Exercice 4** Écrivez le module C implémentant le header défini à l'exercice 3.

▷ **Exercice 5** Écrivez un programme utilisant le type abstrait `BT`.

# Chapitre 6

# Listes

## 6.1 Listes Simplement Chainées

Pour chacun des exercices de cette section, il est bien entendu demander d'appliquer la démarche constructive vue au cours (spécifications formelles et construction du code basée sur l'invariant). Si nécessaire, n'hésitez pas à introduire des notations.

### 6.1.1 Exercices Simples

▷ **Exercice 1** Écrivez un programme qui lit une suite de nombres au clavier et crée la liste chainée correspondante. L'ordre des éléments dans la liste doit traduire l'ordre de lecture des nombres.

▷ **Exercice 2** On considère une liste représentée par son pointeur de début `L`. Écrivez une fonction qui prend une valeur `X` et `L` pour argument et retourne l'adresse de la cellule contenant `X`.

▷ **Exercice 3** On considère une liste, donnée par son pointeur de début `L`. Écrivez une fonction qui recopie cette liste et retourne le pointeur de début de cette nouvelle liste.

▷ **Exercice 4** On considère une liste, donnée par son pointeur de début `L`. Écrivez une fonction qui recopie cette liste en inversant l'ordre de ses éléments. La fonction retourne le pointeur de début de cette nouvelle liste.

### 6.1.2 Récursivité

▷ **Exercice 1** Spécifiez et construisez (de manière récursive) une fonction qui affiche à l'écran le contenu d'une liste simplement chainée d'entiers.

▷ **Exercice 2** Spécifiez et construisez (de manière récursive) une fonction qui teste la présence d'une valeur donnée ($\in \mathbb{Z}$) dans une liste simplement chainée d'entiers.

▷ **Exercice 3** Spécifiez et construisez (de manière récursive) une fonction qui détermine le nombre d'occurrences d'une valeur donnée ($\in \mathbb{Z}$) dans une liste simplement chainée d'entiers.

### 6.1.3 Gestion de Personnes

On considère une personne comme ayant un nom (maximum 20 caractères) et un âge. On demande de créer une structure de données permettant de manipuler une liste simplement chainée, triée par ordre croissant d'âge, de personnes.

▷ **Exercice 1** Écrire une fonction qui insère une personne dans une liste chainée en fonction de son âge.

▷ **Exercice 2** Écrire une fonction qui détermine si une personne particulière est présente ou non dans la liste de personnes.

▷ **Exercice 3** Écrire une fonction qui enlève une personne particulière de la liste. La fonction retourne 1 en cas de succès, -1 si la personne n'est pas présente dans la liste.

▷ **Exercice 4** Écrire une fonction qui va retourner une liste de personnes correspondant à la fusion de deux listes données en argument. La liste résultante conserve, bien entendu, la propriété de tri en fonction de l'âge.

▷ **Exercice 5** Écrire une fonction qui imprime, dans un fichier, le contenu de la liste de personnes. Le fichier doit avoir la forme suivante :

```
NOM         AGE
Laurence    19
Catherine   20
Sébastien   21
Yvette      35
```

## 6.2 Listes Doublement Chainées

Pour chacun des exercices de cette section, il est bien entendu demander d'appliquer la démarche constructive vue au cours (spécifications formelles et construction du code basée sur l'invariant). Si nécessaire, n'hésitez pas à introduire des notations.

### 6.2.1 Exercices Simples

▷ **Exercice 1** Donnez les définitions de type nécessaire pour représenter une liste doublement chainée de caractères.

▷ **Exercice 2** Écrivez une fonction qui prend en argument une liste doublement chainée et qui détermine si cette liste correspond à un début de liste.

▷ **Exercice 3** Écrivez une fonction qui prend en argument une liste doublement chainée et retourne un pointeur vers le début de la liste.

▷ **Exercice 4** Écrivez une procédure qui affiche à l'écran tous les éléments d'une liste doublement chainée.

▷ **Exercice 5** Écrivez une fonction qui détermine, pour une variable pointant sur une liste doublement chainée passée en argument, le nombre de cellules avant cette variable dans la liste. De même, écrivez une autre fonction qui calcule le nombre de cellules dans la liste après cette variable.

▷ **Exercice 6** Écrivez une fonction qui ajoute un élément en tête d'une liste doublement chainée. De même, écrivez une fonction qui ajoute un élément en fin de liste.

### 6.2.2 Ligne de Métro

▷ **Exercice 1** Une station de métro a un nom. Une ligne de métro est constituée :

- d'un numéro (unique) permettant de l'identifier
- d'une liste (bi-directionnelle) de stations

Proposez des structures de données permettant de représenter une station de métro et une ligne de métro. Écrivez aussi toutes les fonctions nécessaires à la création de ces structures.

▷ **Exercice 2** Proposez une structure de données permettant de gérer l'ensemble des lignes de métro d'une ville. Pensez à écrire toutes les fonctions nécessaires à la création de la structure.

▷ **Exercice 3** Écrivez une fonction qui prend en argument une ligne de métro et une station et ajoute cette station en fin de ligne.

▷ **Exercice 4** Écrivez une fonction qui prend en argument une chaine de caractères correspondant au chemin du fichier contenant une ligne de métro, charge les informations en mémoire et renvoie un pointeur vers la ligne de métro nouvellement créée.

On supposera que le format du fichier est le suivant :

- la première ligne est le numéro de la ligne de métro ;
- la deuxième ligne est le nombre de stations de la ligne ;
- jusqu'à la fin du fichier, un nom de station est écrit par ligne.

▷ **Exercice 5** Écrivez une fonction qui prend en entrée une ligne de métro et affiche son numéro et ses terminus.

▷ **Exercice 6** Écrivez une fonction qui permet de savoir si une station particulière appartient ou non à une ligne de métro donnée. On pensera à utiliser la fonction `strcmp` définie dans `string.h`.

▷ **Exercice 7** Écrivez une fonction qui prend en entrée une station de métro et affiche la liste des lignes de métro auxquelles elle appartient.

▷ **Exercice 8** Écrivez une fonction qui prend en argument deux stations de métro et calcule un trajet permettant d'aller de l'une à l'autre.

# Chapitre 7

# Piles

## 7.1 Manipulation d'une Pile

▷ **Exercice 1** Soit le bout de code suivant manipulant une pile d'entiers :

```
1  Stack *s = empty_stack();
2
3  s = push(s, 1);
4  s = push(s, 7);
5  s = push(s, 5);
6
7  int i = top(s);
8  printf("%d\n", i);
9  s = pop(s);
10
11 i = top(s);
12 printf("%d\n", i);
13 s = pop(s);
14
15 i = top(s);
16 printf("%d\n", i);
17 s = pop(s);
18
19 printf("%d\n", is_empty(s));
```

Donnez les états successifs de la pile `s` dans le bout de code supra.

## 7.2 Expressions

### 7.2.1 Écriture d'Expressions

▷ **Exercice 1** Écrivez sous forme postfixée l'expression $((((A + B) + D) > H) \wedge (E < F))$.

▷ **Exercice 2** Écrivez sous forme préfixée l'expression $((((A + B) + D) > H) \wedge (E < F))$.

### 7.2.2 Manipulation d'Expressions

Dans les exercices qui suivent, nous supposons que nous disposons d'une pile telle que spécifiée comme suit :

```
Type:
  Stack
Utilise:
  Boolean, Element
Opérations:
  empty_stack: → Stack
  is_empty: Stack → Boolean
  push: Stack × Element → Stack
  pop: Stack → Stack
  top: Stack → Element
Préconditions:
  ∀ s ∈ Stack, ∀ e ∈ Element
  pop(s) is defined iff is_empty(s) = False
  top(s) is defined iff is_empty(s) = False
Axiomes:
  ∀ s ∈ Stack, ∀ e ∈ Element
  is_empty(empty_stack) = True
  is_empty(push(s, e)) = False
  pop(push(s, e)) = s
  top(push(s, e)) = e
```

Pour les exercices qui suivent, vous disposez aussi des fonctions/procédures suivantes :

- `operateur(`$\alpha$`)` qui détermine si $\alpha$ est un opérateur binaire ;
- `unaire(`$\alpha$`)` qui détermine si $\alpha$ est un opérateur unaire ;
- `variable(`$\alpha$`)` qui détermine si $\alpha$ est une variable ;
- `valeur(`$\alpha$`)` qui retourne la valeur associée à la variable $\alpha$.

▷ **Exercice 1** Spécifiez et construisez un algorithme utilisant une pile et permettant d'évaluer une expression écrite sous forme postfixée. Nous considérons que l'expression postfixée se présente comme un tableau de caractères.

▷ **Exercice 2** Soit les variables $(A, B, C, D)$ et l'environnement $(20, 4, 9, 7)$. Indiquez la trace de votre algorithme, sur votre pile, pour l'expression $AB \times CD + /$.

▷ **Exercice 3** Spécifiez et construisez un algorithme utilisant une pile et permettant d'évaluer une expression complètement parenthésée. Nous considérons que l'expression complètement parenthésée se présente comme un tableau de caractères.

▷ **Exercice 4** Spécifiez et construisez un algorithme utilisant une pile et permettant d'évaluer une expression écrite sous forme préfixée. Nous considérons que l'expression préfixée se présente comme un tableau de caractères.

▷ **Exercice 5** Donnez la trace de votre algorithme pour l'expression préfixée obtenue à l'Exercice 2 de la Sec. 7.2.1 avec l'environnement $(4, 5, 1, 3, 2, 0)$ pour les variables $(A, B, D, E, H, F)$

### 7.2.3 Transformation d'Expressions

Nous avons vu que l'évaluation d'une expression est plus simple à partir d'une représentation postfixée. Cependant, une représentation parenthésée rend l'expression plus lisible.

Nous nous trouvons, ici, devant un aspect très simplifié du processus de compilation d'un langage de programmation qui ne serait formé que d'expressions complètement parenthésées : le *programme source* fourni par l'utilisateur doit être transformé en une chaînée postfixée équivalente (*programme objet*) directement interprétable pour être évaluée.

▷ **Exercice 1** Construisez (via l'approche constructive) un algorithme qui permet de transformer une expression complètement parenthésée en une expression postfixée. Les deux expressions (i.e., ECP et EPOST) sont présentées comme des tableaux de caractères. Le signe '\0' indique la fin de l'expression.

Nous faisons ici l'hypothèse que l'ECP fournie en entrée est syntaxiquement correcte.

▷ **Exercice 2** Modifiez l'algorithme obtenu à l'Exercice 1 afin qu'il détecte les erreurs de syntaxe dans l'ECP fournie en entrée.

# Chapitre 8

# Files

## 8.1 Manipulation d'une File

Pour chacun des exercices qui suit, il est demandé d'évaluer l'expression à l'aide du type abstrait Queue tel que vu au cours.

▷ **Exercice 1** Soit l'expression

```
is_empty(enqueue(dequeue(enqueue(empty_queue, a)), b))
```

Évaluez cette expression à l'aide des axiomes du type abstrait Queue.

▷ **Exercice 2** Soit l'expression

```
head(dequeue(enqueue(enqueue(enqueue((empty_queue, a)), b), c))
```

Évaluez cette expression à l'aide des axiomes du type abstrait Queue.

## 8.2 Utilisation des Files

Soit le bout de code C suivant manipulant une file.

```
1  int i;
2  Queue *q = empty_queue();
3
4  for(i=1; i<=8; i++)
5     q = enqueue(q, i);
6
7  q = dequeue(q);
8  q = dequeue(q);
9
10 for(i=0; i<3; i++)
11    q = enqueue(q, i);
12
13 q = dequeue(q);
14
15 q = enqueue(q, 10);
16 q = enqueue(q, 12);
```

▷ **Exercice 1** Supposons que la file utilisée dans le bout de code indiqué supra soit implémentée comme un tableau contigu à $N$ cases numérotées de 0 à $N-1$.

Supposons que $N = 10$. Quel sera le contenu de la file `q` après l'exécution du bout de code ?

Indiquez les différentes étapes de l'évolution de `q`.

▷ **Exercice 2** Supposons que la file utilisée dans le bout de code indiqué supra soit implémentée comme un tableau circulaire à $N$ cases numérotées de 0 à $N-1$.

Supposons que $N = 10$. Quel sera le contenu de la file `q` après l'exécution du bout de code ?

Indiquez les différentes étapes de l'évolution de `q`.

▷ **Exercice 3** Supposons que la file utilisée dans le bout de code indiqué supra soit implémentée de manière dynamique à l'aide de pointeurs (i.e., liste chaînée avec pointeur de début et de fin).

Quel sera le contenu de la file `q` après l'exécution du bout de code ?

Indiquez les différentes étapes de l'évolution de `q`.

## 8.3 Implémentation des Files

▷ **Exercice 1** Montrez comment implémenter une file à l'aide de deux piles.

▷ **Exercice 2** Analysez le temps d'exécution des primitives de la file que vous venez d'implémenter.

# Chapitre 9

# Élimination de la Récursivité

## 9.1 Chaines de Caractères

L'objectif de ces exercices est de travailler sur les chaines de caractères. Pour ce faire, nous disposons d'un type abstrait `String` dont voici les opérations de base (nous n'indiquons pas ici les axiomes liés à ces opérations, ce n'est pas important pour les exercices qui suivent) :

```
Type:
  String
Utilise:
  Integer, Char, Boolean
Operations:
  empty_string: → String
  is_empty: String → Boolean
  first: String → Char
  remainder: String → String
  adj: String × Char → String
  last: String → Char
  except_last: String → String
Préconditions:
  ∀ s ∈ String
  first(s) is defined iff is_empty(s) = False
  remainder(s) is defined iff is_empty(s) = False
  last(s) is defined iff is_empty(s) = False
```

`remainder()` est l'opération qui enlève le premier caractère de la chaine (en paramètre) et retourne les caractères restants. Par exemple, `remainder("toto")` retourne "`oto`". `adj` est l'opération de concaténation. `last()` est l'opération qui retourne le dernier caractère de la chaine. `except_last()` est l'opération qui retourne la chaine amputée de son dernier caractère.

### 9.1.1 Opération **belong_to**

Soit l'opération `belong_to` qui retourne `True` si le caractère passé en paramètre appartient à la chaine. La définition est la suivante :

```
belong_to: String × Char → Boolean
```

▷ **Exercice 1** Spécifiez et construisez une implémentation récursive de l'opération `belong_to()`.
▷ **Exercice 2** Éliminez la récursivité dans l'algorithme obtenu à l'exercice 1. Attention, il n'est pas question ici de fournir un algorithme itératif mais bien d'éliminer la récursivité comme cela a été vu au cours.

### 9.1.2 Opération `occurrence`

Soit l'opération `occurrence` qui retourne un entier (positif) correspondant au nombre d'occurrences d'un caractère donné dans une chaine. La définition est la suivante :

```
occurrence: String × Char → Integer
```

▷ **Exercice 1** Spécifiez et construisez une implémentation récursive de l'opération `occurence()`.
▷ **Exercice 2** Éliminez la récursivité dans l'algorithme obtenu à l'exercice 1. Attention, il n'est pas question ici de fournir un algorithme itératif mais bien d'éliminer la récursivité comme cela a été vu au cours.

### 9.1.3 Opération `reverse`

Soit l'opération `reverse` qui retourne la chaine passée en argument mais à l'envers. La définition est la suivante :

```
reverse: String → String
```

▷ **Exercice 1** Spécifiez et construisez une implémentation récursive de l'opération `reverse()`.
▷ **Exercice 2** Éliminez la récursivité dans l'algorithme obtenu à l'exercice 1. Attention, il n'est pas question ici de fournir un algorithme itératif mais bien d'éliminer la récursivité comme cela a été vu au cours.