Structures de Données

François Boulier

21 janvier 2014

Introduction

Ce document constitue le support du cours de structures de données, à Polytech'Lille, pour les étudiants en troisième année de la filière GIS (Génie Informatique et Statistique). Les notions présentées dans ce cours sont les suivantes :

- les mécanismes d'allocation dynamique du langage C ;
- la séparation entre le type abstrait et l'implantation d'une structure de données ;
- les mécanismes qui permettent de mettre en œuvre cette séparation (programmation modulaire, notions de constructeur, de destructeur et d'itérateur) ;
- des structures de données destinées à représenter des ensembles de données et à les énumérer suivant différents schémas (piles, files, files avec priorité) ;
- des structures de données destinées à représenter des ensembles de données et à faciliter la recherche d'informations (structures triées, listes chaînées, arbres binaires de recherche, tables de hachage) ;
- des notions élémentaires sur la complexité des algorithmes (relations de récurrence, outils d'estimation de paramètres).

Les structures destinées à faciliter la recherche d'informations sont illustrées avec un algorithme qui reproduit la gestion de la table des symboles de l'éditeur de liens de Linux. Cet algorithme (projet LINKER) donne l'occasion d'étudier les dessous techniques de la programmation modulaire.

Table des matières

Chapitre 1

Principes généraux de programmation modulaire en C

1.1 Notion de structures de données

En pratique, la réalisation d'un type de données (en langage C) s'effectue de la façon suivante : on convient d'une implantation des éléments du type (en général, une structure) ; on spécifie la structure, en précisant le codage des éléments du type ; enfin, on réalise un ensemble de fonctions qui permettent de manipuler ces éléments et de les faire interagir avec des éléments d'autres types.

La partie la plus importante d'un type, c'est l'ensemble des prototypes et des spécifications des fonctions globales [1]. C'est ce qu'on appelle le « type abstrait ». Idéalement, les prototypes et les spécifications de ces fonctions devraient être totalement indépendants de l'implantation des éléments du type (ce n'est pas toujours totalement possible). Pourquoi ? Parce que cela permet de changer l'implantation sans changer le code des fonctions et des programmes qui utilisent le type.

Cette possibilité de modifier facilement l'implantation d'un type est très importante : elle permet de commencer à développer un logiciel avec une implantation rudimentaire d'un type et de reporter les optimisations d'implantation à plus tard.

L'expression « structure de données » peut avoir plusieurs sens, suivant les auteurs. Elle est parfois utilisée pour désigner la façon dont les éléments d'un type sont implantés (l'implantation) mais elle peut aussi désigner le type abstrait ou même la combinaison des deux. Dans ce support de cours, elle désigne le type abstrait.

1. Ça peut sembler surprenant : on pourrait s'attendre à ce que l'implantation, qui est concrète, soit plus importante que les prototypes et les spécifications des fonctions, qui ne décrivent que les relations des éléments du type avec les éléments d'autres types. À croire que les structures de données, comme leurs programmeurs, n'existent que dans le regard des autres :-)

1.2 Programmation modulaire

La programmation modulaire consiste à réaliser un programme par assemblage de modules, ou composants logiciels. C'est en général une bonne idée de réaliser un module pour chaque structure de données importante à réaliser. En C, un module est constitué d'un fichier source et d'un fichier d'entête. Prenons l'exemple d'un type de nombres rationnels. Le code source se trouverait dans un fichier `rationnel.c`. Le fichier d'entête serait `rationnel.h` (le suffixe vient du mot « *header* » en Anglais).

Dans le fichier source, on repère les variables et les fonctions locales au module, au fait que leur définition est précédée du mot-clef `static`. D'une façon générale, il vaut mieux éviter les variables globales, toutes catégories confondues. Les fonctions locales aux modules sont courantes. Voir section 5.1.1 pour plus de détails.

1.2.1 Spécification d'une structure de données

Une déclaration naturelle pour un type de nombres rationnels serait :

```
struct rationnel
{   int numer;
    int denom;
};
```

Cette déclaration ne suffit pas à spécifier ce type, c'est-à-dire à préciser la représentation de ses éléments. Par exemple, on peut préciser que le champ `denom` est strictement positif, que le rationnel 0 est codé par `numer` = 0 et `denom` = 1, et que les fractions ne sont pas obligatoirement réduites.

1.2.2 Schéma d'un fichier d'entête

Traditionnellement, le fichier d'entête d'un module contient les définitions des types, avec leur spécification en commentaire, ainsi que les prototypes des variables et des fonctions globales [2], précédées du mot-clef `extern` [3].

Le fichier d'entête contient aussi les directives d'inclusion de fichiers `#include` nécessaires au compilateur pour comprendre les déclarations. Supposons par exemple qu'on réalise un type spécifique pour les tableaux de nombres rationnels. Le fichier `tableau_rationnel.h` contiendrait les lignes suivantes :

```
#include "rationnel.h"

#define NB_ELEM_MAX 10
struct tableau_rationnel
```

2. Les fichiers d'entête contiennent donc à la fois l'implantation et une partie du type abstrait.

3. Ce mot-clef est optionnel pour les prototypes des fonctions, mais il est obligatoire pour les prototypes des variables globales.

```
{   struct rationnel tab [NB_ELEM_MAX];
    int nb_elem;
};
```

Une difficulté. Le mécanisme décrit ci-dessus pose une difficulté : en raison des directives d'inclusion en cascade, un même fichier d'entête peut être inclus à plusieurs reprises. C'est non seulement une perte de temps mais aussi une source d'erreur puisque les types définis dans ce fichier vont être définis plusieurs fois, ce que le compilateur n'acceptera pas.

Continuons l'exemple et réalisons, en plus du type `struct tableau_rationnel`, un type pour les matrices à coefficients rationnels. Le fichier `matrice_rationnel.h` contiendrait les lignes suivantes :

```
#include "rationnel.h"

#define DIM_MAX 10
struct matrice_rationnel
{   struct rationnel tab [DIM_MAX, DIM_MAX];
    int nb_lig;
    int nb_col;
};
```

Supposons maintenant qu'un programme principal ait besoin d'utiliser ces deux structures de données. Il contiendrait les directives d'inclusion suivantes :

```
#include "tableau_rationnel.h"
#include "matrice_rationnel.h"
```

À la compilation, le fichier `rationnel.h` sera inclus à deux reprises, ce qui provoquera une erreur de compilation. Les inclusions multiples de fichiers d'entête surgissent donc très naturellement, dès qu'un logiciel commence à se développer.

Résolution. Pour éviter les inclusions multiples, on associe une macro à chaque fichier d'entête. Par exemple, au fichier `rationnel.h`, on associe la macro `RATIONNEL_H`. On définit chaque macro à la première inclusion du fichier d'entête correspondant, et on annule l'inclusion d'un fichier d'entête dès que la macro qui lui est associée est définie. Tous les fichiers d'entête peuvent donc être écrits suivant le même schéma, illustré avec l'exemple des rationnels :

```
#if ! defined (RATIONNEL_H)
#define RATIONNEL_H 1

struct rationnel
{   int numer;
```

```
    int denom;
};

#endif
```

1.2.3 Vérification des prototypes

Il est capital que les prototypes de fonctions listés dans les fichiers d'entête correspondent exactement aux fonctions définies dans les fichiers source. Pour cela, on peut commencer par inclure systématiquement le fichier d'entête d'un module dans le fichier source du module, même si le fichier d'entête ne contient que des prototypes de fonctions globales [4]. Le compilateur peut alors vérifier que les prototypes correspondent aux définitions. Pour plus de sécurité, on peut même exiger que toute définition de fonction globale soit bien comparée avec son prototype. Il suffit pour cela de compiler le module avec l'option `-Wmissing-prototypes`. Voici un extrait du manuel du compilateur `gcc` :

```
-Wmissing-prototypes (C and Objective-C only)
    Warn if a global function is defined without a previous prototype
    declaration.  This warning is issued even if the definition itself
    provides a prototype.  The aim is to detect global functions that
    fail to be declared in header files.
```

1.3 Structures de données compréhensibles isolément

Il est préférable de définir des structures de données qui soient compréhensibles isolément, c'est-à-dire indépendemment de la valeur de toute autre variable. L'exemple le plus banal consiste, lorsqu'on implante un tableau pouvant contenir un nombre variable d'éléments, à regrouper dans une même structure le tableau et le nombre d'éléments effectivement présents, comme dans l'exemple du tableau de rationnels.

Des structures de données compréhensibles isolément sont plus faciles à vérifier. Il est aussi plus facile d'écrire des fonctions qui en impriment le contenu à l'écran, ce qui peut être très pratique lorsqu'on cherche une erreur en utilisant le debugger (voir section 5.4.2).

4. Chaque fichier source inclut donc son propre fichier d'entête, qui peut lui-même contenir des directives d'inclusion. Faut-il placer toutes les directives d'inclusion nécessaires à la compilation du fichier source dans son fichier d'entête ? Non : on ne place dans le fichier d'entête que les directives nécessaires au compilateur pour comprendre les déclarations de types et les prototypes des fonctions. D'autres directives peuvent être nécessaires pour compiler le fichier source (`math.h` pour utiliser une fonction de la bibliothèque de maths, ou `stdio.h` pour utiliser `printf`). Ces directives-là doivent figurer dans le fichier source, pas dans le fichier d'entête.

1.4 Constructeurs et destructeurs

Les notions de constructeur et de destructeur sont des notions qui relèvent davantage du langage C++ que du langage C. Ces deux notions sont en fait des réponses à des problèmes qui se posent dans la plupart des langages de programmation, dès qu'on réalise des logiciels un peu volumineux. Voir [1, chapitre 2]. Dans le reste de ce chapitre, on tente d'expliquer ces problèmes et on donne le premier jet d'une réponse en langage C.

Il existe deux catégories de fonctions pour donner une valeur à une variable : les fonctions qui donnent sa première valeur à la variable (ces fonctions sont censées s'appliquer sur des zones mémoire brutes, au contenu aléatoire) et celles qui donnent une nouvelle valeur à la variable, sachant que la variable a reçu précédemment au moins une autre valeur (ces fonctions sont censées s'appliquer sur des zones mémoire cohérentes).

Constructeurs. Les constructeurs d'un type sont des fonctions de la première catégorie, qui agissent sur des zones brutes. Dans ce cours, leur identificateur commence par « `init_` ». Les fonctions de la seconde catégorie ont des identificateurs commençant par « `set_` ».

La distinction entre les deux types de fonctions est importante parce que certaines structures peuvent mobiliser des ressources qu'il faut restituer au système avant qu'elles ne deviennent inaccessibles. C'est le cas des structures contenant des pointeurs vers des zones allouées dynamiquement (via les fonctions système `malloc` ou `realloc`) ou des structures contenant des pointeurs vers des descripteurs de fichiers ouverts (via `fopen` par exemple). Quand une structure comporte de tels champs, les fonctions de la seconde catégorie (qui opèrent sur des zones déjà initialisées) doivent libérer les ressources consommées avant de changer les valeurs des pointeurs (en exécutant `free` sur les zones allouées dynamiquement et `fclose` sur les descripteurs de fichiers ouverts). Par contre les constructeurs, qui opèrent sur des zones brutes ne doivent surtout pas tenter de libérer les zones référencées par les pointeurs !

Destructeurs. À partir du moment où on prévoit un (ou plusieurs) constructeurs pour un type, il faut aussi prévoir une fonction à appliquer sur les éléments du type après leur dernière utilisation. De telles fonctions, appelées destructeurs [5], sont censées libérer les ressources consommées avant que les zones mémoire attribuées aux variables (locales notamment) ne disparaissent [6]. Précisément, le rôle du destructeur [7] consiste à remettre la variable qui lui est passée en paramètre, dans l'état où elle était avant l'application d'un constructeur. Dans ce cours, les destructeurs ont des identificateurs commençant par « `clear_` ».

5. Le terme « nettoyeur » serait mieux approprié mais « destructeur » est classique.

6. Attention à ne pas confondre les destructeurs avec la fonction système `free` : les destructeurs ne libèrent pas la zone occupée par la variable ; ils libèrent les ressources consommées par la variable et qui nécessitent un traitement explicite pour être recyclés par le système.

7. Une structure de données peut avoir plusieurs constructeurs mais ne devrait avoir qu'un seul destructeur.

À quoi ça sert ? Prévoir un ou plusieurs constructeurs et un destructeur pour les structures de données est indispensable si on veut écrire des programmes indépendants de l'implantation de ces structures [8].

1.5 Allocation dynamique de la mémoire

L'utilisation soignée de constructeurs et de destructeurs permet d'éviter beaucoup de difficultés liées à l'allocation dynamique de la mémoire. On commence par décrire, très rapidement, l'organisation d'un programme en cours d'exécution. On décrit ensuite les mécanismes bruts d'allocation dynamique fournis par le langage C. On explique enfin comment les incorporer dans les constructeurs et les destructeurs, afin de minimiser les risques d'erreur.

1.5.1 Programmes exécutables et processus

La description qui suit est très simplifiée.

On distingue les programmes exécutables des processus. Les programmes exécutables sont les programmes en langage machine produits par un compilateur (`gcc`). Les processus sont des programmes exécutables en cours d'exécution.

Dans un programme exécutable, les données sont réparties en deux zones : la zone *text* où sont stockées les instructions en langage machine ; la zone *data* où sont stockées les données du programme C (chaînes de caractères écrites en toutes lettres, tableaux initialisés).

Dans un processus, on trouve deux zones supplémentaires, qui sont créées au lancement de l'exécutable : la *pile d'exécution du processus* (en Anglais, *stack*), qui permet de réaliser le mécanisme des appels de fonction, et où sont stockées, en particulier, les paramètres formels et les variables locales des fonctions ; le *tas* (en Anglais, *heap*), qui est utilisé pour l'allocation dynamique.

Sans entrer dans les détails, à chaque appel de fonction, des zones mémoire sont réservées sur la pile d'exécution, pour les paramètres formels et les variables locales de la fonction. À la fin de l'exécution de la fonction, ces zones sont recyclées par le processus pour les appels des autres fonctions.

Ce mécanisme explique pourquoi on ne peut pas allouer de la mémoire en retournant l'adresse d'une variable locale. Le compilateur C accepte de compiler la fonction ci-dessous, mais prévient qu'elle est plus que douteuse ! En effet, elle retourne bien l'adresse d'une zone de n octets, mais cette zone sera réutilisée par le processus dès le prochain appel de fonction.

```
$ cat mauvais_malloc.c
char* mauvais_malloc (int n)
{   char zone [n];
    return zone;
}
$ gcc -c mauvais_malloc.c
```

8. C'est le cas de ce cours, où les étudiants sont justement censés modifier les implantations.

```
mauvais_malloc.c: In function 'mauvais_malloc':
mauvais_malloc.c:3: warning: function returns address of local variable
```

1.5.2 malloc, realloc, free

Certaines structures de données ont besoin d'allouer dynamiquement de la mémoire. Cette allocation dynamique se fait au moyen des trois fonctions `malloc`, `realloc` et `free`, qu'on passe en revue dans cette section. Voici leur prototype (le type `size_t` est un type d'entier) :

```
#include <stdlib.h>

void* malloc(size_t size);
void  free(void *ptr);
void* realloc(void* ptr, size_t size);
```

malloc. L'appel de fonction `malloc (n)` retourne, en cas de succès, une zone mémoire d'au moins n octets, allouée dans le tas. En cas d'erreur (par exemple en cas de mémoire insuffisante), `malloc` retourne zéro. En général, le paramètre n a la forme d'une constante numérique multipliée par la taille d'un type (voir `sizeof`, un peu plus bas).

free. Si p est l'adresse d'une zone allouée (dans le tas) par `malloc` ou par `realloc`, l'appel de fonction `free (p)` restitue cette zone au processus. Si p vaut zéro (pointeur nul), `free (p)` ne fait rien.

realloc. Si p est l'adresse d'une zone de n octets allouée par `malloc` ou par `realloc`, l'appel de fonction `realloc (p, m)` retourne une zone mémoire, allouée dans le tas, d'au moins m octets. Supposons $m > n$ (le cas le plus fréquent). Alors, le contenu des n premiers octets de la zone retournée est égal au contenu de la zone pointée par p. Si la nouvelle zone est distincte de l'ancienne, alors l'ancienne est automatiquement libérée par un appel à `free`. Si le pointeur p est nul, alors l'appel `realloc (p, m)` est équivalent à `malloc (m)`. Si la nouvelle taille m est nulle, alors l'appel `realloc (p, m)` est équivalent à `free (p)`.

Erreurs classiques

1. oublier d'allouer de la mémoire à un pointeur (se traduit souvent pas une *segmentation fault*) ;
2. appliquer `free` à une adresse non nulle qui n'a pas été obtenue via `malloc` ou `realloc` ;
3. consulter le contenu d'une zone après qu'elle a été libérée avec `free` (voir le destructeur de listes chaînées, en section 1.6.2) ;
4. appliquer `free` deux fois de suite sur la même zone ;

5. allouer une zone trop petite (cette erreur se traduit par un débordement de tableau ; elle survient lorsqu'on oublie de multiplier le nombre d'éléments à allouer par la taille des éléments du type ou, lorsqu'on se trompe sur la nature du type) ;
6. oublier de libérer des zones allouées (cette erreur se traduit par une fuite de mémoire).

L'opérateur `sizeof`

L'opérateur `sizeof`, appliqué à un type du langage C, retourne la taille, en nombre d'octets, des variables du type [9]. Supposons que `n` soit un entier supérieur ou égal à 1. Alors l'instruction suivante :

```
{   struct box* p;
    p = (struct box*)malloc (n * sizeof (struct box));
}
```

affecte à `p` l'adresse d'une zone de taille suffisante pour recevoir un tableau de `n` éléments de type `struct box`. Le paramètre passé à `malloc` devrait toujours être de la forme : un entier multiplié par l'opérateur `sizeof`, appliqué à un type.

L'utilitaire `valgrind`

Sous LINUX, l'utilitaire `valgrind` aide à repérer un grand nombre d'erreurs liées à la gestion de la mémoire, et, en particulier, les fuites de mémoire (*memory leaks* en Anglais). Dans l'exemple suivant, une zone pouvant recevoir 10 doubles est allouée mais jamais libérée.

```
/* Programme a.c */

#include <stdlib.h>

int main ()
{   double* T;
    T = (double*)malloc (10 * sizeof (double));
    return 0;
}
```

Le compilateur ne repère pas l'erreur. L'exécution du programme ne produit aucune erreur visible non plus mais `valgrind` repère la fuite :

```
$ gcc -Wall a.c
$ ./a.out
$ valgrind --leak-check=full ./a.out
  ...
```

9. En pratique, c'est un peu plus compliqué que cela en raison d'un problème de gestion des alignements de zones en mémoire.

```
==3651== LEAK SUMMARY:
==3651==    definitely lost: 80 bytes in 1 blocks
  ...
==3651== For counts of detected and suppressed errors, rerun with: -v
==3651== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 4 from 4)
```

1.5.3 Mise en œuvre avec des constructeurs et des destructeurs

Un constructeur ne devrait jamais contenir d'appel à `free` ou à `realloc`, parce que ce sont des fonctions qui s'appliquent sur des données initialisées (erreur 1).

Une bonne règle consiste à rendre chaque structure de données responsable de la libération des zones mémoires qu'elle a elle-même allouées dynamiquement, *et seulement de celles-là*. En d'autres termes : une instruction `free` placée dans le destructeur d'une structure de données ne doit s'appliquer qu'à une zone allouée dynamiquement par une fonction implantée dans le fichier source associé à cette même structure de données.

1.6 Un exemple : les listes chaînées

À titre d'exemple, on détaille la conception du type `struct liste_double`, qui permet de manipuler des listes de doubles. On définit un fichier d'entête `liste_double.h` et un fichier source `liste_double.c`.

1.6.1 Fichier d'entête

Spécifications du type

On devrait normalement les écrire en commentaire dans le fichier d'entête.

Le type `struct liste_double` permet de manipuler des listes de doubles. Le champ `nbelem` contient le nombre d'éléments de la liste. Le champ `tete` est un pointeur, vers une liste chaînées de maillons. La liste vide est codée par `nbelem` $= 0$ et `tete` $= 0$ (c'est-à-dire le pointeur nul). Le champ `nbelem` est égal au nombre de maillons.

Le type `struct maillon_double` permet de représenter les maillons. Le champ `value` contient la valeur du maillon (un `double`). Le champ `next` est un pointeur vers le maillon suivant. Dans le cas du dernier maillon, on a `next` $= 0$ (c'est-à-dire le pointeur nul). Les deux phrases suivantes sont essentielles.

— Deux listes distinctes n'ont pas de maillon en commun.

— Les maillons sont alloués dynamiquement (dans le tas).

La figure 1.1 illustre le cas de deux variables locales, ou de deux paramètres formels, nommés `src` et `dst`, de type `struct liste_double`.

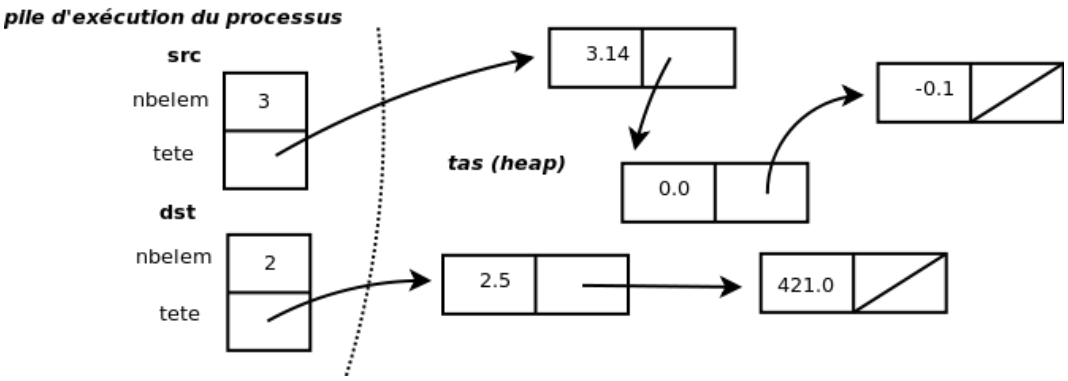


FIGURE 1.1 – deux listes chaînées

Déclarations

On remarque la mécanisme destiné à éviter les inclusions multiples. On a séparé l'implantation des prototypes des fonctions globales.

```
#if ! defined (LISTE_DOUBLE_H)
#define LISTE_DOUBLE_H 1

/*****************************************************************
 * IMPLANTATION
 *****************************************************************/

struct maillon_double
{   double value;
    struct maillon_double* next;
};

struct liste_double
{   struct maillon_double* tete;
    int nbelem;
};

/*****************************************************************
 * PROTOTYPES DES FONCTIONS (TYPE ABSTRAIT)
 *****************************************************************/

extern void init_liste_double (struct liste_double*);
extern void clear_liste_double (struct liste_double*);
extern void set_liste_double (struct liste_double*, struct liste_double*);
extern void ajouter_en_tete_liste_double (struct liste_double*, double);
extern void extraire_tete_liste_double (double*, struct liste_double*);
```

```
extern void imprimer_liste_double (struct liste_double*);
#endif
```

1.6.2 Fichier source

Le fichier source commence par les directives d'inclusion. L'inclusion du fichier d'entête permet de vérifier que les prototypes des fonctions globales sont corrects.

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

Voici l'unique constructeur défini. Comme il opère sur des zones brutes, il ne se préoccupe pas des anciennes valeurs des champs de son paramètre. Il initialise la liste dont l'adresse lui est passée en paramètre.

```
void init_liste_double (struct liste_double* L)
{
    L->tete = (struct maillon_double*)0;
    L->nbelem = 0;
}
```

La fonction suivante modifie la liste dont l'adresse lui est passée en paramètre : elle alloue dynamiquement un nouveau maillon, lui donne la valeur `d`, et le place en tête de la liste chaînée de maillons. Le message d'erreur éventuel est imprimé sur l'erreur standard plutôt que la sortie standard. La fonction `exit` arrête l'exécution du processus.

Supposons qu'au lieu d'une liste de doubles, on implante une liste de listes de doubles. Dans ce cas, il faudrait faire attention à l'affectation de la valeur, pour éviter que deux listes différentes aient des maillons en commun. Il faudrait alors impérativement utiliser la fonction `set_liste_double`, pour éviter une fuite mémoire.

```
void ajouter_en_tete_liste_double (struct liste_double* L, double d)
{   struct maillon_double* nouveau;

    nouveau = (struct maillon_double*)malloc (sizeof (struct maillon_double));
    if (nouveau == (struct maillon_double*)0)
    {   fprintf (stderr, "erreur: mémoire saturée\n");
        exit (1);
    }
    nouveau->value = d;       /* affectation de la valeur */
    nouveau->next = L->tete;
    L->tete = nouveau;
    L->nbelem += 1;
}
```

Voici le destructeur. Il libère les maillons en utilisant `free`. Remarquer que dans la boucle, on a pris soin d'accéder au champ `next` du maillon `courant` *avant* de libérer `courant`.

Ce destructeur est correct parce que deux listes distinctes n'ont pas de maillon commun, et parce que les maillons sont tous alloués dynamiquement.

Supposons qu'au lieu d'une liste de doubles, on implante une liste de listes de doubles. Dans ce cas, il faudrait appliquer le destructeur de listes de doubles sur le champ `value` de `courant`, avant de libérer ce maillon avec `free`.

```
void clear_liste_double (struct liste_double* L)
{   struct maillon_double* courant;
    struct maillon_double* suivant;
    int i;

    courant = L->tete;
    for (i = 0; i < L->nbelem; i++)
    {   suivant = courant->next;
        free (courant);
        courant = suivant;
    }
}
```

Une question s'est posée lors d'une séance de travaux pratiques : ne faudrait-il pas appeler la fonction `init_liste_double` à la fin du destructeur pour remettre les champs à zéro ? Réponse en deux parties :

1. remettre les champs à zéro est une bonne idée, qui peut aider à révéler rapidement les bugs du programme utilisateur du module, mais ce n'est pas obligatoire (il ne faut pas confondre le destructeur de listes avec une fonction qui affecterait la liste vide à `L`) ;
2. si on choisit de remettre les champs à zéro, il vaut mieux le faire directement dans la fonction qu'en appelant le constructeur, puisque, en principe, un utilisateur qui souhaiterait se servir d'une variable après appel au destructeur, est censé appliquer explicitement un constructeur à cette variable (voir fonction `set_liste_double`).

La fonction suivante est une fonction locale du module `liste_double`. Elle est utilisée par `set_liste_double`. Elle renverse l'ordre des éléments de la liste dont l'adresse lui est passée en paramètre (si avant l'appel, `L` $= [a_1, a_2, \ldots, a_n]$ alors, après l'appel, on a `L` $= [a_n, \ldots, a_2, a_1]$).

```
static void retourner_liste_double (struct liste_double* L)
{   struct maillon_double *precedent, *courant, *suivant;
    int i;

    if (L->nbelem >= 2)
    {   courant = L->tete;
```

```
        suivant = courant->next;
        courant->next = (struct maillon_double*)0;
        for (i = 1; i < L->nbelem; i++)
        {   precedent = courant;
            courant = suivant;
            suivant = suivant->next;
            courant->next = precedent;
        }
        L->tete = courant;
    }
}
```

La fonction suivante est la plus difficile à écrire. Voir [1, section 6.5]. Elle affecte une copie de `src` à `dst`. On remarque que la fonction ne peut pas se permettre d'affecter tout simplement `src` à `dst`, parce que les deux listes auraient alors des maillons en commun.

La fonction n'est pas un constructeur : la liste `dst` est supposée initialisée. Elle contient donc, en général, des maillons. Pour ne pas perdre de mémoire allouée dynamiquement, on choisit une solution simple : on commence par appliquer le destructeur sur `dst` puis, on réinitialise la liste.

On prévoit la possibilité que `src` et `dst` soient la même liste (dans ce cas, on ne fait rien).

Pour la recopie, on utilise la fonction d'ajout en tête, ce qui a pour effet d'obtenir une copie ... mais à l'envers. On appelle ensuite `retourner_liste_double` pour remettre les éléments à l'endroit.

```
void set_liste_double (struct liste_double* dst, struct liste_double* src)
{   struct maillon_double* M;
    int i;

    if (dst != src)
    {   clear_liste_double (dst);
        init_liste_double (dst);
        M = src->tete;
        for (i = 0; i < src->nbelem; i++)
        {   ajouter_en_tete_liste_double (dst, M->value);
            M = M->next;
        }
        retourner_liste_double (dst);
    }
}
```

La fonction suivante affecte à `d` le double présent dans le premier maillon de la liste `L`, puis supprime ce maillon de la liste. La liste est supposée non vide.

Supposons qu'au lieu d'une liste de doubles, on implante une liste de listes de doubles. Dans ce cas, il faudrait impérativement utiliser la fonction `set_liste_double` pour réaliser l'affectation.

```
void extraire_tete_liste_double (double* d, struct liste_double* L)
{   struct maillon_double* tete;

    if (L->nbelem == 0)
    {   fprintf (stderr, "erreur : liste vide\n");
        exit (1);
    }
    tete = L->tete;
    *d = tete->value;      /* affectation */
    L->tete = tete->next;
    L->nbelem -= 1;
    free (tete);
}
```

La fonction suivante imprime la liste passée en paramètre, sur la sortie standard.

```
void imprimer_liste_double (struct liste_double* L)
{   struct maillon_double* M;
    int i;

    printf ("[");
    M = L->tete;
    for (i = 0; i < L->nbelem; i++)
    {   if (i == 0)
            printf ("%f", M->value);
        else
            printf (", %f", M->value);
        M = M->next;
    }
    printf ("]\n");
}
```

En voici une variante qui n'utilise pas le champ `nbelem`.

```
void imprimer_liste_double (struct liste_double* L)
{   struct maillon_double* M;
    int i;

    printf ("[");
    M = L->tete;
```

```
    while (M != (struct maillon_double*)0)
    {   if (M == L->tete)
            printf ("%f", M->value);
        else
            printf (", %f", M->value);
        M = M->next;
    }
    printf ("]\n");
}
```

1.6.3 Utilisation du module

Le programme principal suivant, qui n'a ni queue ni tête[10], constitue un exemple d'utilisation des fonctions ci-dessus. Le fichier d'entête du module est chargé. Deux variables de type `struct liste_double` sont déclarées. Le programme commence par appliquer le constructeur sur chacune des deux variables. Il effectue ensuite diverses manipulations sur les deux variables. Avant de s'arrêter, le destructeur est appliqué sur chaque variable.

```
#include "liste_double.h"

int main ()
{   struct liste_double A, B;

    init_liste_double (&A);
    init_liste_double (&B);

    ajouter_en_tete_liste_double (&A, 3.14);
    ajouter_en_tete_liste_double (&A, 2.718);
    set_liste_double (&B, &A);
    imprimer_liste_double (&B);

    clear_liste_double (&A);
    clear_liste_double (&B);
    return 0;
}
```

1.7 Itérateurs

Pour bien séparer l'implantation du type abstrait de la structure de données `struct liste_double`, il reste à fournir à l'utilisateur de la structure de données, des moyens

10. Un comble pour un programme de manipulation de listes.

d'accéder aux doubles stockés dans les listes, sans manipuler directement la structure C. Ce problème se résout en définissant un *itérateur*.

D'une façon générale, un itérateur est une structure de données I destinée à parcourir une autre structure E représentant un ensemble de données. Écrire un itérateur présente deux avantages :

1. masquer l'implantation de la structure de données parcourue E afin de faciliter son évolution future ou des variantes d'implantation ;
2. écrire une fois pour toutes certains algorithmes de parcours qui peuvent être particulièrement délicats.

Un exemple bien connu est l'itérateur de fichiers. L'ensemble de fonctions `fopen`, `fgetc` et `fclose` permettent d'énumérer tous les caractères d'un fichier texte indépendemment de l'implantation des fichiers. Les programmes C qui utilisent ces fonctions sont portables sur toutes les plateformes, alors que les implantations des fichiers peuvent être très différentes ; la représentation des fichiers sous UNIX est fort compliquée. L'algorithme masqué par `fgetc` est loin d'être élémentaire.

On détaille ci-dessous un itérateur pour le type `struct liste_double`. Voici la partie à ajouter au fichier d'entête `liste_double.h`. La structure de données associée à l'itérateur se nomme `struct iterateur_liste_double`. Elle comporte un pointeur `liste` vers la liste à parcourir, un pointeur `maillon` vers le dernier maillon dont la valeur a été consultée et un champ `indice` contenant l'indice de ce maillon dans la liste. Le champ `maillon` vaut zéro (le pointeur nul) si l'itérateur est positionné « à l'extérieur » de la liste. L'indice du premier maillon vaut zéro. L'itérateur est lui-même une structure de données. Il comporte un constructeur et un destructeur.

```
struct iterateur_liste_double
{   struct liste_double* liste;
    struct maillon_double* maillon;
    int indice;
};

extern void init_iterateur_liste_double
                (struct iterateur_liste_double*, struct liste_double*, int);
extern void clear_iterateur_liste_double (struct iterateur_liste_double*);
extern bool next_iterateur_liste_double
                (struct iterateur_liste_double*, double*);
extern bool prev_iterateur_liste_double
                (struct iterateur_liste_double*, double*);
```

Voici le constructeur de l'itérateur. Il initialise l'itérateur dont l'adresse est passée dans `iter` et le positionne, soit au début de la liste `L`, avant le premier maillon (dans le cas où `pos` vaut zéro), soit à la fin de la liste, après le dernier maillon (dans le cas où `pos` est non nul).

```
void init_iterateur_liste_double
        (struct iterateur_liste_double* iter, struct liste_double* L, int pos)
{
    iter->liste = L;
    iter->maillon = (struct maillon_double*)0;
    if (pos == 0)
        iter->indice = -1;
    else
        iter->indice = L->nbelem;
}
```

On a prévu un constructeur. Voici donc le destructeur ... qui ne fait rien ! Cela peut sembler surprenant : pourquoi le destructeur n'applique-t-il pas le destructeur de listes sur le champ `liste` ?

Une première réponse est une réponse bon sens : le destructeur d'itérateur sera appliqué à l'itérateur, à la fin d'un quelconque parcours d'une liste. Mais ce n'est pas parce qu'un parcours est terminé que la liste elle-même doit disparaître !

On peut proposer une deuxième réponse, plus syntaxique : le destructeur de listes ne devrait pas être appelé par le destructeur d'itérateurs, parce que le constructeur de listes n'est pas appelé par le constructeur d'itérateurs. Voir la section 1.5.3.

```
void clear_iterateur_liste_double (struct iterateur_liste_double* iter)
{
}
```

Les deux fonctions suivantes permettent de parcourir la liste dans les deux sens. L'un des deux parcours est nettement plus efficace que l'autre. C'est dû à l'implantation que nous avons choisie.

La fonction suivante fait avancer l'itérateur d'un maillon. Si le maillon courant fait bien partie de la liste, la fonction affecte sa valeur au double dont l'adresse est dans `d` et retourne `true`. Dans l'autre cas, elle retourne `false`.

```
bool next_iterateur_liste_double
                        (struct iterateur_liste_double* iter, double* d)
{
    iter->indice += 1;
    if (iter->indice == 0)
        iter->maillon = iter->liste->tete;
    else if (iter->indice < iter->liste->nbelem)
        iter->maillon = iter->maillon->next;

    if (iter->indice < 0 || iter->indice >= iter->liste->nbelem)
        return false;
```

```
    *d = iter->maillon->value;   /* affectation */
    return true;
}
```

La fonction suivante a la même spécification que la précédente, si ce n'est qu'elle fait reculer l'itérateur d'un maillon.

```
bool prev_iterateur_liste_double
                        (struct iterateur_liste_double* iter, double* d)
{   int i;

    iter->indice -= 1;
    if (iter->indice < 0 || iter->indice >= iter->liste->nbelem)
        return false;
    iter->maillon = iter->liste->tete;
    for (i = 0; i < iter->indice; i++)
        iter->maillon = iter->maillon->next;

    *d = iter->maillon->value;  /* affectation */
    return true;
}
```

Question 1. Réécrire la fonction `imprimer_liste_double` de telle sorte qu'elle utilise un itérateur de listes.

Question 2. Modifier l'implantation de la structure de données, en ajoutant à la structure C, un pointeur vers le dernier maillon.

1.8 Gestion des erreurs

Dans les implantations des différents modules, en cas d'erreur, le processus s'arrête brutalement (voir le fichier `error.c`). Cette stratégie simple est admissible si on développe un programme. Elle est inadmissible si on développe une bibliothèque. Dans ce cas, l'information « une erreur s'est produite » doit systématiquement être remontée aux fonctions appelantes, sans arrêter le processus.

Lorsqu'on traite les erreurs ainsi, il faut aussi prévoir le cas où une erreur se produit *lors de l'exécution d'un constructeur* : la construction n'étant pas terminée, les champs de la variable en cours d'initialisation ne sont peut-être pas tous dans un état cohérent : on ne peut donc pas appliquer le destructeur sur la variable et libérer les ressources qui avaient été allouées avec succès, avant que l'erreur ne se produise.

Une solution consiste à programmer les constructeurs pour qu'ils initialisent immédiatement tous les champs des variables avec des valeurs par défaut (typiquement, affecter zéro à

tous les pointeurs) et, seulement après, effectuent les véritables initialisations. Parallèlement, les destructeurs doivent être programmés pour reconnaître ces valeurs par défaut et ne pas libérer des ressources qui n'ont pas été initialisées.

La mise en œuvre de la solution décrite ci-dessus peut demander du soin, dans le cas de structures de données complexes dont les champs doivent eux-mêmes être initialisés par des constructeurs.

Bibliographie

[1] Jacquelin Charbonnel. *Langage C++. Les spécifications du standard ANSI/ISO expliquées*. InterEditions, Paris, 1997. Deuxième édition.

Chapitre 2

Énumération de données

Les structures de données présentées dans ce chapitre permettent de représenter des ensembles de données et de les énumérer suivant différents schémas.

2.1 Les piles et les files

Les piles et les files sont des structures de données permettant de représenter, toutes deux, des ensembles de données. Voir [1, chapitre 10, page 195] ou [2, section 3.3]. L'ajout d'un élément dans l'ensemble et le retrait d'un élément hors de l'ensemble se font par des opérations spécialisées, qui assurent

— dans le cas d'une file, que les éléments sortent dans le même ordre que celui dans lequel ils sont entrés (principe du premier entré, premier sorti) ;

— dans le cas d'une pile, que les éléments sortent dans l'ordre inverse de celui dans lequel ils sont entrés (principe du premier entré, dernier sorti).

La tradition veut que les opérations d'ajout et de retrait portent les noms « enfiler » et « défiler », dans le cas d'une file, et « empiler » et « dépiler », dans le cas d'une pile. Il est possible de réaliser une file d'éléments de type `element` avec le type abstrait minimaliste suivant. La première fonction permet de vider la file F. La deuxième permet de tester si F est vide. La troisième enfile e dans F. La quatrième défile l'élément le plus ancien de F et le retourne.

```
void vider (struct file* F);
bool est_vide (struct file* F);
void enfiler (struct file* F, element e);
element defiler (struct file* F);
```

De même, il serait possible de réaliser une pile d'`element` avec le type abstrait minimaliste suivant. La première fonction permet de vider la pile P. La deuxième permet de tester si P est vide. La troisième empile e dans P (au sommet de P). La quatrième dépile l'élément le plus récent de P (au sommet de P) et le retourne.

```
void vider (struct pile* P);
bool est_vide (struct pile* P);
void empiler (struct pile* P, element e);
element depiler (struct pile* P);
```

Pour des raisons de confort, on implante généralement un peu plus de fonctions. Pour permettre l'utilisation simultanée de piles et de files d'éléments de types différents, on peut faire apparaître le type des éléments dans les identificateurs de fonctions.

Les prototypes suivants permettraient d'implanter un module de files de doubles. On remarque la présence d'un constructeur et d'un destructeur ainsi que de quelques fonctions supplémentaires comme `longueur_file_double`, qui retourne le nombre d'éléments présents dans la file (cette fonction pourrait se reprogrammer avec les quatre fonctions mentionnées plus haut) et `est_pleine_file_double`, qui teste s'il est encore possible d'enfiler un double (cette fonction est nécessaire pour toute implantation réaliste). Enfin, on a tourné un peu différemment la fonction qui défile un double.

```
void init_file_double (struct file_double*);
void vider_file_double (struct file_double*);
void clear_file_double (struct file_double*);
int longueur_file_double (struct file_double*);
bool est_vide_file_double (struct file_double*);
bool est_pleine_file_double (struct file_double*);
void enfiler_double (struct file_double*, double);
void defiler_double (double*, struct file_double*);
void imprimer_file_double (struct file_double*);
```

Question 3. Donner une implantation en C de la fonction `longueur_file_double`, en utilisant les autres fonctions de manipulation de files de doubles.

2.1.1 Implantation avec un tableau de taille fixe

Dans cette section et les deux suivantes, les implantations varient considérablement mais les types abstraits ne changent pas.

On obtient une implantation très simple d'une pile avec un tableau et un indice, comme le montre l'implantation des piles de doubles ci-dessous. Les doubles empilés sont enregistrés dans le champ `tab` entre les indices 0 (bas de la pile) et `sp` (sommet de la pile). Il y a deux conventions naturelles possibles pour `sp` : soit ce champ contient l'indice du dernier double empilé, soit il contient l'indice du premier emplacement libre dans `tab`. On a choisi la première convention. Le nombre d'éléments présents dans la pile est donc égal à $\texttt{sp} + 1$. Quand la pile est vide, `sp` vaut -1.

```
#define SIZE_PILE_DOUBLE 1000
struct pile_double
```

```
{   int sp;                          /* l'indice du sommet de pile */
    double tab [SIZE_PILE_DOUBLE];   /* la zone de stockage */
};
```

On obtient une implantation assez simple d'une file avec un tableau `tab` de dimension N et deux indices r et w. L'indice r (champ `read_end` ci-dessous) désigne l'extrémité en lecture de la file (c'est l'indice du prochain élément à être défilé). L'indice w (champ `write_end` ci-dessous) désigne l'extrémité en écriture de la file (c'est l'indice du dernier élément à avoir été enfilé). L'astuce consiste à faire avancer les indices « circulairement » sur le tableau en les incrémentant modulo N : si un élément e doit être enfilé, on calcule la nouvelle valeur de w par la formule $w = (w + 1) \mod N$ et on range e dans `tab` à l'indice w ; si un élément doit être défilé et rangé dans une variable e, on affecte à e l'élément de `tab` situé à l'indice r et on calcule la nouvelle valeur de r par $r = (r + 1) \mod N$. On peut initialiser la file avec $r = 1$ et $w = 0$. Attention au fait que la file ainsi décrite a une capacité maximale de $N - 1$ éléments (et pas N car, sans information supplémentaire, rien ne permet de distinguer la file vide de la file à N éléments). Dans la structure de données ci-dessous, on a résolu ce dernier problème en mémorisant le nombre d'éléments enfilés dans un champ de la structure.

```
#define SIZE_FILE_DOUBLE 10000
struct file_double
{   int n;                           /* le nombre de doubles dans la file */
    int read_end;                    /* indice du prochain élément défilé */
    int write_end;                   /* indice du dernier élément enfilé */
    double tab [SIZE_FILE_DOUBLE];   /* la zone de stockage */
};
```

Principales fonctions

Le constructeur est très simple. Supposons qu'au lieu d'une pile de doubles, on réalise une pile de liste de doubles. Il faudrait alors appliquer le constructeur de listes de doubles sur chaque emplacement du tableau `tab`.

```
void init_pile_double (struct pile_double* P)
{
    P->sp = -1;
}
```

Le destructeur est vide. Supposons qu'au lieu d'une pile de doubles, on réalise une pile de liste de doubles. Il faudrait alors appliquer le destructeur de listes de doubles sur chaque emplacement du tableau `tab`.

```
void clear_pile_double (struct pile_double* P)
{
}
```

La fonction suivante est sans difficulté.

```
bool est_pleine_pile_double (struct pile_double* P)
{
    return P->sp == SIZE_PILE_DOUBLE-1;
}
```

La fonction suivante empile un double. Les macros __FILE__ et __LINE__ sont remplacés par le nom du fichier et le numéro de ligne courants, pour mieux localiser les erreurs. Supposons qu'au lieu d'une pile de doubles, on réalise une pile de liste de doubles. Il faudrait alors utiliser la fonction `set_liste_double` pour réaliser l'affectation.

```
void empiler_double (struct pile_double* P, double d)
{
    if (est_pleine_pile_double (P))
    {   fprintf (stderr, "%s:%d\n", __FILE__, __LINE__);
        exit (1);
    }
    P->sp += 1;
    P->tab [P->sp] = d;          /* affectation */
}
```

2.1.2 Implantation avec un tableau redimensionnable

On raisonne sur la pile, qui est plus simple, mais les idées sont exactement les mêmes pour la file. L'idée consiste à faire de `tab` un pointeur plutôt qu'un tableau, à lui allouer de le mémoire avec `malloc` et à mémoriser dans un champ `alloc` le nombre d'emplacements alloués. Lorsque le nombre d'emplacements effectivement utilisés, `sp`, atteint la valeur de `alloc`, la zone allouée à `tab` est agrandie en utilisant `realloc`. On obtient :

```
struct pile_double
{   int alloc;    /* le nombre d'emplacements alloués à tab */
    int sp;       /* le sommet de pile = nombre d'emplacements utilisés */
    double* tab;  /* un pointeur vers la zone de stockage */
};
```

Principales fonctions

Le constructeur initialise la pile avec le tableau vide.

```
void init_pile_double (struct pile_double* P)
{
    P->alloc = 0;
    P->sp = -1;
    P->tab = (double*)0;
}
```

Le destructeur. Supposons qu'au lieu d'une pile de doubles, on réalise une pile de liste de doubles. Il faudrait alors appliquer le destructeur de listes de doubles sur tous les emplacements du tableau, avant de libérer la zone avec `free`.

```
void clear_pile_double (struct pile_double* P)
{
    free (P->tab);
}
```

La spécification de la fonction suivante a légèrement évolué [1]. C'est elle qui est chargée du redimensionnement du tableau. Elle utilise la fonction `realloc`. Supposons qu'au lieu d'une pile de doubles, on réalise une pile de liste de doubles. Il faudrait alors appliquer le constructeur de listes de doubles sur tous les emplacements de `newtab` situés entre les indices `P->alloc` et `newalloc` $-$ 1.

```
bool est_pleine_pile_double (struct pile_double* P)
{   int newalloc;
    double* newtab;
    bool b;

    if (P->sp + 1 < P->alloc)
        b = false;
    else
    {   newalloc = 2 * P->alloc + 1;
        newtab = (double*)realloc (P->tab, newalloc * sizeof (double));
        if (newtab == (double*)0)
            b = true;
        else
        {
/* On ne modifie P que si on est sûr realloc a fonctionné */
            P->alloc = newalloc;
            P->tab = newtab;
            b = false;
        }
    }
    return b;
}
```

La fonction suivante n'a pas vraiment changé. La fonction `assert` teste si la condition qui lui est passée en paramètre est vraie. Si elle ne l'est pas, elle arrête le processus en imprimant le nom du fichier et le numéro de ligne où l'erreur s'est produite. Pour l'utiliser, il est nécessaire d'inclure le fichier `assert.h`. On peut désactiver cette fonction en passant l'option `-DNDEBUG` au compilateur.

1. Ce qu'on ne devrait pas faire si on appliquait dogmatiquement nos théories.

Supposons qu'au lieu d'une pile de doubles, on réalise une pile de liste de doubles. Il faudrait alors utiliser la fonction `set_liste_double` pour réaliser l'affectation.

```
void empiler_double (struct pile_double* P, double d)
{
    assert (! est_pleine_pile_double (P));
    P->sp += 1;
    P->tab [P->sp] = d;     /* affectation */
}
```

2.1.3 Implantation avec listes chaînées

On les laisse en exercice.

2.2 Files avec priorité

Voir [1, section 6.5, page 131]. Une file avec priorité (en Anglais, *priority queue*), est une file où les éléments enfilés ont différents niveaux de priorité, et où les éléments prioritaires sont défilés avant les autres. Une file avec priorité permet de trier un ensemble d'éléments (il suffit de tous les enfiler, puis de tous les défiler) mais c'est une structure de données aux applications plus générales.

Dans ce cours, on se concentre sur une implantation des files avec priorité, utilisant des tableaux. On suppose de plus que les éléments enfilés sont d'un type `struct element` (une structure) contenant un champ `indice`. Dans ce champ, on enregistrera la position de l'élément dans la file, c'est-à-dire l'indice de l'élément dans le tableau.

Supposons qu'on veuille réaliser une file avec priorité de doubles. Avec les conventions énoncées dans le paragraphe précédent, on est amené à encapsuler les doubles dans une structure :

```
struct element
{   double value;
    int indice;
};
```

Comment déterminer si un élément est plus prioritaire qu'un autre ? Pour atteindre un certain degré de généralité, on paramètre une file avec priorité de `struct element` avec une fonction du type suivant :

```
typedef bool fonction_de_priorite (struct element*, struct element*);
```

qui retourne `true` si son premier paramètre est plus prioritaire que le second, `false` sinon.

Il est possible de réaliser une file avec priorité d de `struct element` avec le type abstrait minimaliste suivant. Lors des opérations de file, le champ `indice` des éléments présents dans

la file sera modifié. Pour cette raison, il est préférable de supposer que le tableau contient les *adresses* des éléments enfilés et pas une copie de ces éléments.

La fonction `initialiser` vide la file F et reçoit l'adresse de la fonction de priorité à utiliser. Les fonctions `longueur`, `enfiler` et `defiler` se passent de commentaires. La fonction `changement_priorite` permet de mettre à jour la position d'un élément dont on connaît l'indice dans le tableau, après que son degré de priorité a changé.

```
void initialiser
        (struct file_priorite*, fonction_de_priorite* est_prioritaire);
int longueur (struct file_priorite*);
void enfiler (struct file_priorite*, struct element*);
struct element* defiler (struct file_priorite*);
void changement_priorite (int indice, struct file_priorite*);
```

La fonction `changement_priorite` ne semble pas très naturelle mais elle répond à un vrai besoin, qui apparaît dans plusieurs applications importantes des files avec priorité. C'est le cas de l'algorithme de Dijkstra, en théorie des graphes [2, section 5.2.3], où un sommet a est plus prioritaire qu'un sommet b si le *potentiel* de a est inférieur à celui de b ; les potentiels des sommets sont régulièrement modifiés au cours de l'algorithme. C'est la gestion de ces changements de priorité qui motive la gestion du champ `indice` dans les éléments enfilés.

2.2.1 Comment munir un tableau d'une structure de tas

L'idée consiste à stocker les éléments de la file avec priorité dans un tableau T muni d'une structure de tas. Les éléments sont stockés dans des emplacements T_i avec $0 \leq i < n$. On munit le tableau d'une structure de tas en posant que chaque emplacement T_i du tableau a un fils gauche $T_{2\,i+1}$ et un fils droit $T_{2\,i+2}$. Les emplacements T_i tels que $2\,i + 2 \geq n$ n'ont pas de fils droit. Ceux tels que $2\,i + 1 \geq n$ n'ont pas de fils gauche. Par conséquent, tout emplacement T_i, à l'exeption de T_0, a pour père l'emplacement $T_{\lfloor (i-1)/2 \rfloor}$. On maintient la propriété clef suivante :

$$T_i \textit{ est plus prioritaire que chacun de ses fils, pour tout } 0 \leq i < n.$$

On ne maintient aucun lien de priorité entre les deux fils d'un emplacement. Plus précisément, on s'interdit de maintenir le moindre lien de priorité entre les deux fils. L'efficacité de la structure est une conséquence de cette interdiction. Un exemple est donné en figure 2.1.

Les deux sous-algorithmes importants sont ceux qui permettent de mettre à jour la position d'un élément dans la file, suite à une augmentation ou une diminution de son degré de priorité. Ils sont donnés figures 2.2, page 32 (pour l'augmentation) et 2.3, page 33 (pour la diminution).

Pour enfiler un nouvel élément e dans le tas, il suffit d'incrémenter n, de ranger e en T_{n-1} et d'appeler la fonction de la figure 2.2 (augmentation de priorité) avec $n - 1$ pour paramètre. L'élément le plus prioritaire est nécessairement en T_0. Lorsque cet élément est défilé, un emplacement libre se crèe en T_0. Il suffit alors de recopier le dernier élément du

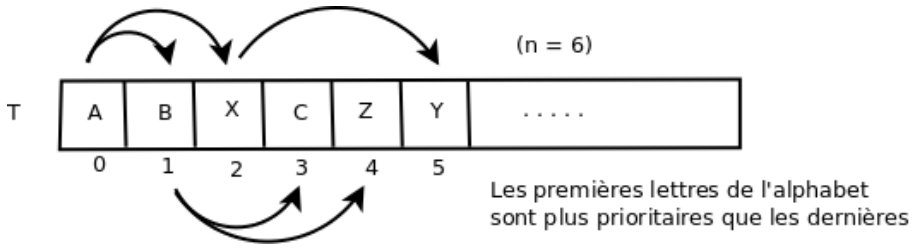


FIGURE 2.1 – Un exemple de tableau vu comme un tas

function augmentation_priorité (i)
La priorité de l'élément T_i a été augmentée. Il faut le faire progresser dans la file
begin
$elt = T[i]$
$fils = i$
$pere = \lfloor (fils - 1)/2 \rfloor$
Invariant : l'emplacement du fils est libre
while $fils > 0$ et elt est plus prioritaire que $T[pere]$ do
L'emplacement libre remonte au niveau du père
$T[fils] = T[pere]$
$fils = pere$
$pere = \lfloor (fils - 1)/2 \rfloor$
end do
$T[fils] = elt$
end

FIGURE 2.2 – La fonction augmentation_priorité

tas, T_{n-1}, dans l'emplacement libre T_0, puis d'appeler la fonction de la figure 2.3 (diminution de priorité) avec 0 pour paramètre. Enfin, lors d'un changement de priorité à l'indice i, il suffit d'appeler la fonction de la figure 2.2 (on fait le pari qu'il s'agit d'une augmentation de priorité) avec i pour paramètre. Si l'élément n'a pas été déplacé par la fonction, on appelle la fonction de la figure 2.3 (au cas où il s'agirait d'une diminution).

2.2.2 Implantation

On précise une implantation d'une file avec priorité d'éléments de type `struct element*`. On rappelle que ce sont les adresses des éléments qui sont mis dans la file, pas des copies des éléments. En effet, pour simplifier la gestion du champ `indice` des éléments, on veut éviter que les éléments soient dupliqués. Le champ `n` contient le nombre d'éléments présents dans la file. La zone de stockage est implantée sous la forme d'un tableau de taille fixe. L'adresse de la fonction de priorité à utiliser est stockée dans la structure C.

```
function diminution_priorite (i)
  La priorité de l'élément T_i a été diminuée. Il faut le faire reculer dans la file
  On note n le nombre d'éléments présents dans la file
begin
  elt = T[i]
  pere = i
  gauche = 2 pere + 1
  droit = gauche + 1
Invariant : l'emplacement du père est libre
  do
    fini = vrai
    if gauche < n then
      if droit < n then
        max = l'indice du plus prioritaire des deux fils
      else
        max = gauche
      end if
      if T[max] est plus prioritaire que elt then
Le plus prioritaire des fils remonte au niveau du père et l'emplacement libre descend
        T[pere] = T[max]
        pere = max
        gauche = 2 pere + 1
        droit = gauche + 1
        fini = faux
      end if
    end if
  while non fini
  T[pere] = elt
end
```

FIGURE 2.3 – La fonction diminution_priorité

```
#define SIZE_FILE_PRIORITE_ELEMENT 100
struct file_priorite_element
{   int n;
    struct element* tab [SIZE_FILE_PRIORITE_ELEMENT];
    fonction_de_priorite* est_prioritaire;
};
```

Principales fonctions

Les deux sous-algorithmes sont implantés sous la forme de fonctions locales. Ce sont des variantes proches des versions idéalisées des figures 2.2 et 2.3. On commence par le constructeur :

```
void init_file_priorite_element
        (struct file_priorite_element* F, fonction_de_priorite* fonction)
{
    F->n = 0;
    F->est_prioritaire = fonction;
}
```

Dans la fonction suivante, l'élément X est extérieur à la file. Un emplacement vide a été créé en `tab [pos]`. Cet emplacement vide, qui va recevoir X en fin de boucle, est éventuellement remonté dans la file. La fonction retourne `true` si l'emplacement a été remonté au moins une fois. La fonction `prioritaire` est utilisée pour comparer les degrés de priorité des éléments avec X. Chaque fois qu'un élément est déplacé dans le tableau, son champ `indice` est mis-à-jour.

```
static bool augmentation_priorite_element
        (int pos, struct element* X, struct file_priorite_element* F)
{   int fils, pere;
    bool augmentation_effective;

    augmentation_effective = false;
    fils = pos;
    pere = (fils - 1) / 2;
/* Invariant : l'emplacement vide est au niveau du fils */
    while (fils > 0 && (*F->est_prioritaire) (X, F->tab [pere]))
    {
/* On remonte l'emplacement vide au niveau du père */
        F->tab [fils] = F->tab [pere];
        F->tab [fils]->indice = fils;
        fils = pere;
        pere = (fils - 1) / 2;
        augmentation_effective = true;
    }
    F->tab [fils] = X;
    F->tab [fils]->indice = fils;
    return augmentation_effective;
}
```

La fonction suivante est une variante de la précédente. L'élément X est extérieur à la file. Un vide a été créé en `tab [pos]`. Cet emplacement vide, qui va recevoir X en fin de

boucle, est éventuellement descendu dans la file. La fonction retourne `true` si l'élément a effectivement été descendu.

```
static bool diminution_priorite_element
        (int pos, struct element* X, struct file_priorite_element* F)
{   int pere, gauche, droit, max;
    bool fini, diminution_effective;

    diminution_effective = false;
    pere = pos;
    gauche = 2*pos + 1;
    droit = gauche + 1;
/* Invariant : l'emplacement vide est au niveau du père */
    do
    {   fini = true;
        if (gauche < F->n)
        {   if (droit < F->n)
            {   if ((*F->est_prioritaire) (F->tab [gauche], F->tab [droit]))
                    max = gauche;
                else
                    max = droit;
            } else
                max = gauche;
            if ((*F->est_prioritaire) (F->tab [max], X))
            {
/*
 * On descend l'emplacement vide au niveau du fils le plus prioritaire.
 * On remonte le fils le plus prioritaire au niveau du père
 */
                F->tab [pere] = F->tab [max];
                F->tab [pere]->indice = pere;
                pere = max;
                gauche = 2*pere + 1;
                droit = gauche + 1;
                diminution_effective = true;
                fini = false;
            }
        }
    } while (!fini);
    F->tab [pere] =X;
    F->tab [pere]->indice = pere;
    return diminution_effective;
}
```

L'insertion d'un nouvel élément dans la file.

```
void enfiler_priorite_element
                (struct file_priorite_element* F, struct element* X)
{
    assert (F->n < SIZE_FILE_PRIORITE_ELEMENT);
    F->n += 1;
    augmentation_priorite_element (F->n - 1, X, F);
}
```

L'extraction d'un élément de la file. La fonction retourne l'adresse de l'élément défilé.

```
struct element* defiler_priorite_element (struct file_priorite_element* F)
{   struct element* X;

    assert (F->n > 0);
    X = F->tab [0];
    F->n -= 1;
    diminution_priorite_element (0, F->tab [F->n], F);
    return X;
}
```

La priorité de l'élément `tab [pos]` a été modifiée, mais on ne sait pas s'il s'agit d'une augmentation ou d'une diminution de priorité. On fait le pari d'une augmentation de priorité et on se sert du booléen retourné pour vérifier si on a raison ou tort, et appeler, le cas échéant, la fonction qui gère les diminutions de priorité.

```
void changement_priorite_element (int pos, struct file_priorite_element* F)
{   struct element* X;

    assert (pos >= 0 && pos < F->n);
    X = F->tab [pos];
    if (! augmentation_priorite_element (pos, X, F))
        diminution_priorite_element (pos, X, F);
}
```

Bibliographie

[1] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction à l'algorithmique*. Dunod, Paris, 2ème edition, 2002.

[2] Clarisse Dhaenens. Graphes et Combinatoire. Support du cours de GIS 3, Polytech'Lille, 2010.

Chapitre 3

Introduction à la notion de complexité

Étudier la complexité d'un algorithme, c'est étudier son efficacité lorsque la taille de sa donnée tend vers l'infini. Dans le cadre de ce cours, on se concentre sur l'efficacité en temps de calcul. Dans certains problèmes, il est utile d'étudier l'efficacité en taille mémoire.

Pour pouvoir mener l'étude de la complexité d'un algorithme, on est toujours obligé d'abstraire le problème, c'est-à-dire d'étudier un problème mathématique un peu arbitraire, dont on espère qu'il reflète les points essentiels du comportement du vrai algorithme.

Une approximation courante consiste à mesurer la taille de la donnée d'un algorithme par une unique variable n. Dans le cas d'un algorithme de tri, n pourrait désigner le nombre d'éléments du tableau à trier ; dans le cas de l'algorithme d'Euclide, n pourrait être le nombre de bits du plus grand des deux entiers dont on cherche le pgcd. Cette approximation n'est pas toujours souhaitable : en théorie des graphes, il est courant de mesurer la taille de la donnée avec deux variables : le nombre de sommets et le nombre d'arcs.

Une autre approximation courante consiste à ne mesurer l'efficacité que selon un seul critère. Dans le cas d'un algorithme de tri, on peut ne s'intéresser qu'au nombre de comparaisons d'éléments ; dans le cas de l'algorithme d'Euclide, on peut ne s'intéresser qu'au nombre d'opérations arithmétiques. Ici aussi, l'approximation effectuée peut masquer des phénomènes importants : deux multiplications d'entiers peuvent prendre des temps de calcul très différents suivant que les entiers comportent un grand nombre de chiffres ou pas.

En résumé, dans ce cours, la complexité d'un algorithme est une fonction $f : \mathbb{N} \to \mathbb{N}$, qui à un entier n, représentant la taille de sa donnée, associe un nombre $f(n)$ d'opérations considéré comme représentatif du comportement de l'algorithme.

Pour une taille de la donnée, n, fixée, l'algorithme à étudier peut se comporter très différemment suivant les cas. Le cas le plus couramment étudié est le « pire des cas » (définition ci-dessous) mais on peut s'intéresser aussi au « meilleur des cas » ou même à la complexité « en moyenne ».

Le pire des cas est, par définition, le cas où la fonction $f(n)$ croît le plus vite vers l'infini, quand n tend vers l'infini. Le meilleur des cas est, par définition, celui où la fonction $f(n)$ croît le moins vite vers l'infini, quand n tend vers l'infini.

Prenons l'exemple d'un algorithme, paramétré par un élément et un tableau, qui cherche

si l'élément appartient au tableau, par une recherche exhaustive. L'entier n est égal au nombre d'éléments du tableau. La fonction $f(n)$ est égale au nombre de comparaisons d'éléments. Le meilleur des cas est celui où l'élément recherché est trouvé tout de suite (on a $f(n) = 1$). Le pire des cas est celui où l'élément n'appartient pas au tableau (on a $f(n) = n$).

Le cas moyen suppose qu'on fasse quelques hypothèses. Par exemple, on peut supposer que l'élément appartient au tableau et qu'il a la probabilité $1/n$ de se trouver dans chacune des n cases. l'algorithme va trouver l'élément en 1 comparaison avec une probabilité $1/n$, en 2 comparaisons avec une probabilité $1/n$, en 3 comparaisons avec une probabilité $1/n$, etc. Le nombre de comparaisons devient alors une variable aléatoire. Le nombre $f(n)$ est l'espérance de cette variable :

$$f(n) = \frac{1}{n} \frac{n\,(n+1)}{2} = \frac{n+1}{2}.$$

Dans ce cours, on ne s'intéressera qu'à la complexité dans le pire des cas, sauf lorsqu'on s'intéressera aux tables de hachage, où la seule complexité vraiment intéressante est une complexité en moyenne.

Question 4. On s'intéresse au nombre de comparaisons d'éléments effectués par l'algorithme du tri par insertion, appliqué à un tableau de n éléments. À la question « quel est le meilleur des cas ? » un étudiant répond : « c'est le cas où $n = 0$: le tableau est vide et il n'y a aucune comparaison à effectuer ». Qu'en pensez-vous ?

3.1 Grand O, grand Ω et grand Θ

Les définitions suivantes ont été introduites par Knuth [2] en 1976. Voir [3, page 4] ou [1, chapitre 3]. Soient $f(n)$ et $g(n)$ deux fonctions de $\mathbb{R}$ dans $\mathbb{R}$. On dit que

$\mathbf{f(n) \in \Theta(g(n))}$ s'il existe deux constantes c_{inf}, c_{sup} telles que, quand n tend vers plus l'infini,

$$0 < c_{\text{inf}} \leq \left| \frac{f(n)}{g(n)} \right| \leq c_{\text{sup}}$$

$\mathbf{f(n) \in O(g(n))}$ s'il existe une constante c_{sup} telle que, quand n tend vers plus l'infini,

$$\left| \frac{f(n)}{g(n)} \right| \leq c_{\text{sup}}$$

$\mathbf{f(n) \in \Omega(g(n))}$ s'il existe une constante c_{inf} telle que, quand n tend vers plus l'infini,

$$0 < c_{\text{inf}} \leq \left| \frac{f(n)}{g(n)} \right|$$

En pratique, $f(n)$ représente le nombre d'opérations effectuées par un certain algorithme sur une donnée de taille n et $g(n)$ est une formule (par exemple, $g(n) = n^2$ ou $g(n) = \log(n)$).

Beaucoup d'auteurs écrivent $f(n) = \Theta(g(n))$ au lieu de $f(n) \in \Theta(g(n))$. Cet abus de langage ne se justifie pas dans ce cours et on ne l'utilisera pas.

Dire que $f(n) \in \Theta(g(n))$, revient à dire que, quand n est grand, $f(n)$ est à peu près égal à $g(n)$, à une constante multiplicative près. On dit que $g(n)$ est un « équivalent asymptotique » de $f(n)$. L'exemple de la recherche dichotomique, traitée ci-dessous, montre que cette appellation peut cacher des comportements un peu surprenants. Si $f(n) \in O(g(n))$, on dit que $g(n)$ est une « borne asymptotique supérieure » de $f(n)$. Si $f(n) \in \Omega(g(n))$, on dit que $g(n)$ est une « borne asymptotique inférieure » de $f(n)$.

Du point de vue de l'analyse de la complexité, ces notations sont importantes parce qu'elles permettent de faire abstraction des facteurs constants dus aux détails d'implantation ou des termes négligeables dans les formules.

Question 5. Deux algorithmes ont des complexités respectives en $\Theta(n)$ et en $\Theta(n^2)$. Peut-on affirmer que le premier est toujours plus rapide que le second ?

Question 6. Que dire d'un algorithme dont la complexité appartient à $\Theta(1)$?

3.1.1 L'exemple des logarithmes

On rappelle que si $n = 2^p$ alors p est le *logarithme en base* 2 de n, noté $\log_2(n)$. Plus généralement, si b est un réel positif et $n = b^p$ alors p est le *logarithme en base* b de n, noté $\log_b(n)$. Tous les logarithmes de n sont égaux à une constante multiplicative près. En effet, $\log_b(n) = \ln(n)/\ln(b)$ où « ln » désigne le classique *logarithme neperien*. Par conséquent, lorsqu'on écrit qu'une opération a une complexité en $\Theta(\log n)$, on ne précise pas la base.

3.2 Déterminer la complexité d'un algorithme

En pratique, on peut tenter deux approches complémentaire :

1. chercher une équation de récurrence pour la fonction $f(n)$ et essayer de la résoudre, à la main ou avec un logiciel de calcul formel ;
2. fabriquer un fichier de mesures pour $f(n)$ en exécutant l'algorithme à analyser, tracer la courbe et utiliser un logiciel d'estimation de paramètres.

3.2.1 Le Taylor shift

Le petit programme suivant effectue le changement de variable $x = \alpha\, y + \beta$ sur un polynôme : il calcule les coefficients b_i du polynôme $p(\alpha\, y + \beta)$ à partir des coefficients a_i du polynôme $p(x) = a_0\, x^n + a_1\, x^{n-1} + \cdots + a_n$.

```
void TaylorShift (double* b, double* a, int n, double alpha, double beta)
{   int i, j;
    for (i = 0; i <= n; i++)
    {   b [i] = a [i];
        for (j = i-1; j >= 0; j--)
        {   b [j+1] = beta * b [j] + b [j+1];
            b [j] = alpha * b [j];
        }
    }
}
```

On cherche à déterminer le nombre $f(n)$ d'opérations arithmétiques effectuées sur les doubles. C'est assez facile à faire à la main sur cet exemple mais on donne une méthode qui s'applique à des exemples plus compliqués.

On commence par s'intéresser à la boucle intérieure. Dans cette boucle, l'indice i est fixé. On note $s(i)$ le nombre d'opérations arithmétiques effectuées sur des doubles, dans cette boucle. On a $s(i) = 3\,i$. On s'intéresse ensuite à la boucle extérieure et on exprime $f(n)$ en fonction de $s(i)$. On a :

$$f(n) = s(0) + s(1) + \cdots + s(n) = 0 + 3 + \cdots + 3\,n\,.$$

Supposons qu'on ne reconnaisse pas la somme ci-dessus. On peut alors définir $f(n)$ comme la solution d'une relation de récurrence avec condition initiale :

$$f(n) = f(n-1) + 3\,n\,, \quad f(0) = 0\,.$$

On peut maintenant se servir du solveur de relations de récurrence `rsolve` de MAPLE :

```
> syst := { f(n) = f(n-1) + 3*n, f(0) = 0 };
                    syst := {f(0) = 0, f(n) = f(n - 1) + 3 n}
> rsolve (syst, f(n));
                           3 (n + 1) (n/2 + 1) - 3 - 3 n
> expand (%);
                                        2
                                  3/2 n  + 3/2 n
```

On peut aussi s'intéresser au nombre $f(n)$ d'affectations effectuées. On commence par s'intéresser au nombre d'affectations effectuées dans la boucle intérieure. Ce nombre, noté $s(i)$ dépend de i, qui est fixé. On a : $s(i) = 2\,i$. Cela fait, on peut s'intéresser au nombre $f(n)$ d'affectations effectuées par la boucle extérieure. Ce nombre est solution d'une relation de récurrence avec condition initiale :

$$f(n) = f(n-1) + s(n) + 1 = f(n-1) + 2\,n + 1\,, \quad f(0) = 1\,.$$

Toujours avec MAPLE :

```
> syst := { f(n) = f(n-1) + 2*n + 1, f(0) = 1 };
                 syst := {f(0) = 1, f(n) = f(n - 1) + 2 n + 1}
> rsolve (syst, f(n));
                          -1 - n + 2 (n + 1) (n/2 + 1)
> expand (%);
                                                2
                                    1 + 2 n + n
```

Dans les deux cas, on a $f(n) \in \Theta(n^2)$, ce qui nous permet d'affirmer que le temps de calcul de l'algorithme croît avec le carré du degré du polynôme.

3.2.2 La recherche dichotomique

Résumé. La fonction $f(n)$ naturelle de l'algorithme de la recherche dichotomique est une fonction en escalier, dont l'analyse n'est pas élémentaire. Dans cette section, on détaille une démarche qui peut servir sur d'autres exemples. On commence par définir $f(n)$ par une relation de récurrence. On s'aperçoit qu'elle est trop compliquée pour être résolue directement. On cherche alors une relation de récurrence proche mais plus simple. La solution de cette relation plus simple nous donne une forme présumée pour $f(n)$ dépendant de constantes. On ajuste les valeurs des constantes avec l'algorithme d'estimation de paramètres de `gnuplot` et un fichier de mesures obtenu en simulant $f(n)$ numériquement. Le résultat est convaincant graphiquement et il est plus facile de terminer l'analyse.

Le problème initial

On considère la fonction suivante, qui retourne `true` si `elt` appartient au tableau `T`, entre les indices `deb` et `fin` -1. Le tableau est supposé trié par ordre croissant. La fonction applique l'algorithme de la recherche dichotomique. Elle est tournée récursivement.

```
bool appartient (double* T, int deb, int fin, double elt)
{   int m;
    if (deb >= fin)
        return false;
    else
    {   m = (deb + fin) / 2;
        if (elt < T [m])
            return appartient (T, deb, m, elt);
        else if (elt > T [m])
            return appartient (T, m+1, fin, elt);
        else
            return true;
    }
}
```

La même fonction, mais tournée itérativement :

```
bool appartient (double* T, int deb, int fin, double elt)
{   int a, b, m;
    bool trouve;
    a = deb;
    b = fin;
    trouve = false;
    while (a < b && !trouve)
    {   m = (a + b) / 2;
        if (elt < T [m])
            b = m;
        else if (elt > T [m])
            a = m+1;
        else
            trouve = true;
    }
    return trouve;
}
```

Simplification du problème

On s'intéresse au nombre $f(n)$ de comparaisons de doubles effectuées par l'algorithme, dans le cas où `elt` n'appartient pas au tableau. Ici, $n =$ `fin` $-$ `deb` représente le nombre d'éléments du tableau. À chaque appel récursif, on passe de n éléments à $\lfloor n/2 \rfloor$ (où $\lfloor n/2 \rfloor$ désigne la partie entière de $n/2$) ; l'algorithme s'arrête lorsque $n = 0$. On a donc envie d'écrire

$$f(n) = f\left(\lfloor n/2 \rfloor\right) + 2\,, \quad f(0) = 0\,.$$

Malheureusement, MAPLE ne parvient pas à résoudre la récurrence :

```
> syst := { f(n) = f(floor(n/2)) + 2, f(0) = 0 };
                 syst := {f(0) = 0, f(n) = f(floor(n/2)) + 2}
> rsolve (syst, f(n));
              rsolve({f(0) = 0, f(n) = f(floor(n/2)) + 2}, f(n))
```

On est alors tenté de simplifier la relation de récurrence en supprimant cavalièrement le calcul de partie entière. Ce n'est pas suffisant, MAPLE ne parvient toujours pas à résoudre la récurrence, en raison d'un problème lié à la condition initiale :

```
> syst := { f(n) = f(n/2) + 2, f(0) = 0 };
                      syst := {f(0) = 0, f(n) = f(n/2) + 2}
> rsolve (syst, f(n));
Error, (in rsolve/dc) initial conditions are inconsistent with the recurrence
```

On soupçonne MAPLE de procéder au changement de variable $p = \log_2(n)$ et de rencontrer un problème avec la fonction logarithme, en $n = 0$. On modifie donc légèrement la condition initiale. Cette nouvelle condition initiale apparaîtrait d'ailleurs naturellement si on changeait très légèrement la condition d'arrêt de la fonction `appartient`. Cette fois-ci, le logiciel trouve la solution : $f(n) = 2\log_2(n) + 1$.

```
> syst := { f(n) = f(n/2) + 2, f(1) = 1 };
                      syst := {f(1) = 1, f(n) = f(n/2) + 2}
> rsolve (syst, f(n));
                                 ln(2) + 2 ln(n)
                                 ---------------
                                       ln(2)
```

Retour au problème initial

On a approximé la relation de récurrence initiale, qu'on ne parvenait pas à résoudre, par une relation plus simple, qu'on est parvenu à résoudre. Mais l'approximation était-elle légitime ? Pour tenter de le savoir, on commence par construire un fichier de mesures donnant le graphe de la récurrence initiale. On peut utiliser pour cela le programme suivant, qui fabrique un fichier de mesures, nommé `"stats"` [1] :

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

int f (int n)
{
    if (n == 0)
        return 0;
    else
        return f (n/2) + 2;
}

int main ()
{   FILE* fichier;
    int n;
    fichier = fopen ("stats", "w");
    if (fichier == (FILE*)0)
    {   fprintf (stderr, "erreur fopen\n");
        exit (1);
    }
    for (n = 1; n < 1000; n++)
```

1. Sur cet exemple-ci, le fichier de mesures est donné par une fonction C qui évalue la fonction $f(n)$ à partir de son équation de récurrence. Sur d'autres exemples, on peut obtenir le fichier de mesures en modifiant le programme à analyser et en lui faisant compter les opérations de comparaison.

```
        fprintf (fichier, "%d\t%d\n", n, f(n));
    fclose (fichier);
    return 0;
}
```

L'idée consiste alors à chercher une fonction $f(n)$ de la forme suggerée par l'analyse du cas simplifié, c'est-à-dire une fonction de la forme $a \log_2(n) + b$, où a et b sont deux paramètres qui restent à estimer. On utiliser pour cela la fonction `fit`[2] de `gnuplot`. On trouve une approximation $f(n) \simeq 1.91 \log_2(n) + 1.6$ assez proche du $2 \log_2(n)$ obtenu en résolvant la récurrence de la version simplifiée.

```
gnuplot> log2(x) = log(x)/log(2)
gnuplot> fit a*log2(x)+b "stats" via a,b

Final set of parameters            Asymptotic Standard Error
=======================            ==========================

a               = 1.91324          +/- 0.01228      (0.6418%)
b               = 1.6559           +/- 0.1062       (6.411%)

gnuplot> plot "stats" with lines, 1.91*log2(x) + 1.6
```

Les graphiques, donnés figure 3.1, suggèrent bien un « comportement logarithmique ». Mais peut-on affirmer que $f(n) \in \Theta(\log(n))$? Pour s'en assurer, on extrait du fichier `"stats"`, les mesures qui correspondent aux angles des marches. Pour les angles inférieurs, on trouve les lignes suivantes

```
# n   finf(n)
   1      2
   3      4
   7      6
  15      8
  31     10
  63     12
 127     14
 255     16
 511     18
```

2. La fonction `fit` implante un algorithme de moindres carrés non linéaires : la méthode de Levenberg-Marquardt, qui fait partie de la famille des méthodes de Newton. Cela signifie que l'expression vis-à-vis de laquelle l'estimation est faite pourrait dépendre non linéairement des paramètres à estimer (les paramètres pourraient figurer en exposant, par exemple). Dans notre cas, l'expression dépend linéairement des paramètres à estimer et on aurait pu utiliser des moindres carrés linéaires.

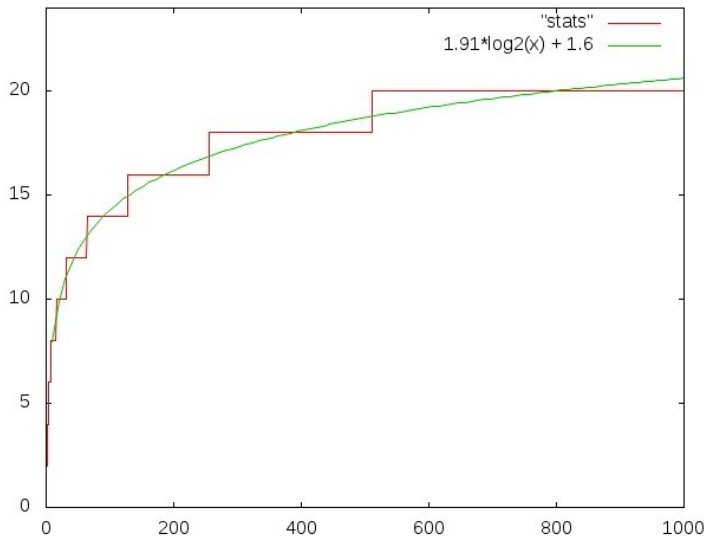


FIGURE 3.1 – La fonction $f(n)$ expérimentale et celle obtenue par estimation de paramètres.

La récurrence se résout facilement [3] et on trouve $f_{\text{inf}}(n) = 2\log_2(n+1)$. Pour les angles supérieurs, on trouve les lignes suivantes

```
# n   fsup(n)
   1      2
   2      4
   4      6
   8      8
  16     10
  32     12
  64     14
 128     16
 256     18
 512     20
```

La récurrence se résout encore plus facilement et on trouve $f_{\text{sup}}(n) = 2\log_2(n) + 2$. Voir les courbes figure 3.2. Effectuons deux calculs de limites avec MAPLE :

```
> finf := 2*log[2](n+1);
                                    2 ln(n + 1)
                          finf := -----------
                                      ln(2)
> fsup := 2*log[2](n) + 2;
```

3. Ajouter une colonne imaginaire p commençant à 0, incrémentée de 1 en 1. Exprimer n et $f_{\text{inf}}(n)$ en fonction de p. On trouve $n(p) = 2^{p+1} - 1$ et $f_{\text{inf}}(p) = 2\,p + 2$. Il ne reste plus qu'à tirer p en fonction de n dans la première formule et à reporter le résultat dans la seconde.

```
                                    2 ln(n)
                          fsup := ------- + 2
                                    ln(2)
> limit (finf/log[2](n), n = infinity);
                                    2
> limit (fsup/log[2](n), n = infinity);
                                    2
```

Les limites calculées nous suggèrent des valeurs pour les constantes. Prenons $0 < c_{\text{inf}} < 2$. Par exemple $c_{\text{inf}} = 1$. On a

$$\frac{f(n)}{\log_2(n)} \geq \frac{f_{\text{inf}}(n)}{\log_2(n)} > c_{\text{inf}}$$

quand n tend vers plus l'infini. Par conséquent, $f(n) \in \Omega(\log(n))$. Similairement, prenons $c_{\text{sup}} > 2$. Par exemple $c_{\text{sup}} = 3$. On a

$$\frac{f(n)}{\log_2(n)} \leq \frac{f_{\text{sup}}(n)}{\log_2(n)} < c_{\text{sup}}$$

quand n tend vers plus l'infini. Par conséquent, $f(n) \in O(\log(n))$ et donc $f(n) \in \Theta(\log(n))$.

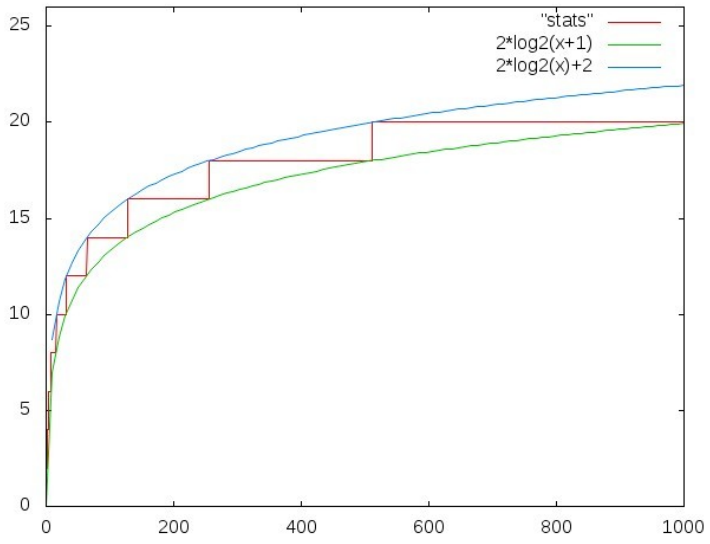


FIGURE 3.2 – La fonction $f(n)$ est encadrée par les fonctions $f_{\text{inf}}(n)$ et $f_{\text{sup}}(n)$.

Conclusion

On a trouvé que l'algorithme de la recherche dichotomique a une complexité en temps, dans le pire des cas, en $\Theta(\log(n))$. Pourtant, $f(n)$ est en fait une fonction constante par

morceaux et on voit que, dans l'ensemble des fonctions appartenant à $\Theta(\log(n))$, on peut trouver des fonctions aux comportements très différents.

On a obtenu un équivalent asymptotique de la fonction $f(n)$ du pire des cas. On a donc une borne asymptotique de la fonction $f(n)$ dans tous les cas : l'algorithme de la recherche dichotomique a toujours une complexité en temps en $O(\log(n))$.

3.2.3 La suite de Fibonacci

La suite de Fibonacci est définie par :

$$F(0) = F(1) = 1\,, \quad F(n+2) = F(n+1) + F(n)\,. \tag{3.1}$$

Cette suite est un exemple de relation de récurrence linéaire à coefficients constants (elle est linéaire parce que $F(n+2)$ dépend linéairement de $F(n+1)$ et de $F(n)$). La forme générale des solutions des relations de récurrence linéaires à coefficients constants est bien connue (voir la section suivante) : il faut s'attendre, dans le cas général, à une combinaison linéaire d'exponentielles (la variable n figurant en exposant). Le nombre d'exponentielles de la combinaison linéaire est, en général, égal à l'ordre de la suite. Ici, c'est 2. On devrait donc avoir

$$F(n) = a\,\lambda^n + b\,\mu^n$$

où les paramètres a, b, λ et μ restent à déterminer. Vérification avec MAPLE :

```
syst := { F(n) = F(n-1) + F(n-2), F(0) = 1, F(1) = 1};
          syst := {F(0) = 1, F(1) = 1, F(n) = F(n - 1) + F(n - 2)}

> rsolve (syst, F(n));
          /       1/2\ /   1/2      \n   / 1/2      \ / 1/2      \n
          |      5   | |  5         |    |5         | |5         |
          |1/2 - ----| |- ---- + 1/2|  + |---- + 1/2| |---- + 1/2|
          \       10 / \   2        /    \ 10       / \ 2        /

> evalf (%);
                                      n                              n
          0.2763932023 (-0.6180339880)  + 0.7236067977 1.618033988
```

Quand n tend vers l'infini, $F(n)$ croît comme φ^n où φ désigne le « nombre d'or » $(1+\sqrt{5})/2$. La suite de Fibonacci peut sembler terriblement académique. Elle apparaît en fait dans de nombreux problèmes de calculs de complexité. On en aura un exemple lors de l'étude des arbres AVL. L'analyse de cet exemple est en fait proche de celle la fonction suivante, qui calcule $F(n)$.

```
int Fibonacci (int n)
{
    if (n <= 1)
```

```
        return 1;
    else
        return Fibonacci (n-1) + Fibonacci (n-2);
}
```

On s'intéresse au nombre $f(n)$ d'additions effectuées par cette fonction. Ce nombre est solution d'une équation de récurrence avec conditions initiales, qui est proche de la suite de Fibonacci :

$$f(0) = f(1) = 0\,, \quad f(n+2) = f(n+1) + f(n) + 1\,. \tag{3.2}$$

Les conditions initiales ont changé et une constante supplémentaire est apparue dans la relation de récurrence. En fait, ces différences sont sans importance si on s'intéresse au comportement asymptotique des fonctions (voir la section suivante). Pour une justification calculatoire, on peut utiliser MAPLE :

```
> syst := { f(n) = f(n-1) + f(n-2) + 1, f(0) = 0, f(1) = 0};
         syst := {f(0) = 0, f(1) = 0, f(n) = f(n - 1) + f(n - 2) + 1}

> rsolve (syst, f(n));
                        1/2 /      2    \n      1/2 /     2    \n
                    2 5    |- ---------|    2 5    |- --------|
                           |    1/2    |           |   1/2    |
                           \  -5    + 1/           \  5    + 1/
                -1 - ---------------------- + --------------------
                                1/2                    1/2
                           5 (-5    + 1)          5 (5    + 1)
> evalf (%);
                                        n                                 n
        -1. + 0.7236067980 1.618033989  + 0.2763932022 (-0.6180339888)
```

On trouve une suite différente, bien sûr, mais le comportement asymptotique n'a pas changé. Pour conclure cette section, on peut remarquer que, comme on connaît la forme de la fonction,

$$f(n) \simeq a\,\lambda^n + b\,\mu^n$$

il est possible d'estimer les paramètres a, b, λ et μ avec un algorithme d'estimation de paramètres. Pour cela, on simule numériquement la fonction $f(n)$ avec la fonction C suivante (en produisant un fichier `"stats"` comme précédemment) :

```
int f (int n)
{
    if (n <= 1)
        return 0;
    else
        return f(n-1) + f(n-2) + 1;
}
```

La fonction `fit` de `gnuplot` donne une bonne approximation de l'exponentielle dominante (l'estimation de l'autre exponentielle, qui tend vers zéro, est perturbée par la constante -1 de la vraie solution) :

```
gnuplot> fit a*lambda**x+b*mu**x "stats" via a,b,lambda,mu

Final set of parameters            Asymptotic Standard Error
=======================            ==========================

a               = 0.72351          +/- 0.0001611    (0.02226%)
b               = -1.04019         +/- 0.03473      (3.339%)
lambda          = 1.61804          +/- 1.837e-05    (0.001135%)
mu              = 0.994882         +/- 0.005082     (0.5108%)
```

3.3 Récurrences linéaires à coefficients constants

Cette section donne quelques précisions mathématiques sur les relations de récurrence linéaires. Le théorème suivant est extrait de [3, Theorem 2.2, page 48].

Théorème 1 *Toute solution de l'équation de récurrence linéaire à coefficients constants*

$$f(n) + a_1\, f(n-1) + a_2\, f(n-2) + \cdots + a_t\, f(n-t) = 0$$

est une combinaison linéaire de termes de la forme $n^j\, \beta^n$ où β est une racine du polynôme caractéristique

$$C(z) = z^t + a_1\, z^{t+1} + a_2\, z^{t+2} + \cdots + a_t$$

et où l'exposant j est strictement inférieur à la multiplicité de la racine β de $C(z)$. Les coefficients dépendent des conditions initiales $f(0), f(1), \ldots, f(t-1)$.

Reprenons l'exemple de la suite de Fibonacci (3.1). Son polynôme caractéristique, $C(z) = z^2 - z - 1$, a deux racines simples

$$z = \frac{1 \pm \sqrt{5}}{2}\cdot$$

Comme les racines sont simples, c'est-à-dire de multiplicité 1, les exposants j mentionnés dans le théorème sont nuls et toute solution est de la forme

$$F(n) = a \left(\frac{1+\sqrt{5}}{2}\right)^n + b \left(\frac{1-\sqrt{5}}{2}\right)^n \cdot \tag{3.3}$$

Les constantes a et b dépendent des conditions initiales. Rappelons que $F(0) = F(1) = 1$. Les constantes sont donc solutions du système d'équations linéaires :

$$F(0) = a + b = 1\,, \quad F(1) = a\,\frac{1+\sqrt{5}}{2} + b\,\frac{1-\sqrt{5}}{2} = 1\,.$$

On trouve

$$a = \frac{5+\sqrt{5}}{10}\,, \quad b = \frac{5-\sqrt{5}}{10}\cdot$$

Relations non homogènes. La relation de récurrence considérée dans le théorème 1 est une relation « homogène », c'est-à-dire que son membre droit est égal à zéro. Pour résoudre une relation de récurrence linéaire à coefficients constants, non homogène, c'est-à-dire avec un membre droit non nul,

$$f(n) + a_1\, f(n-1) + a_2\, f(n-2) + \cdots + a_t\, f(n-t) = \textit{membre droit},$$

il suffit de résoudre l'équation homogène associée (en oubliant le membre droit) et d'ajouter une solution particulière à l'expression obtenue. Reprenons l'exemple de la suite (3.2). La relation homogène associée est celle de la suite de Fibonacci. Sa solution générale est de la forme (3.3). Une solution particulière de la relation non homogène est $f(n) = -1$. La solution générale de la relation non homogène est donc

$$f(n) = a \left(\frac{1+\sqrt{5}}{2}\right)^n + b \left(\frac{1-\sqrt{5}}{2}\right)^n - 1\,. \tag{3.4}$$

Comme précédemment, les constantes a et b s'obtiennent en résolvant le système d'équations linéaires obtenu pour $n = 0$ et $n = 1$.

Un autre exemple est apparu lors de l'étude de l'algorithme de la recherche dichotomique (calcul de f_{inf}). Après ajout d'une colonne imaginaire p, on cherche à exprimer n en fonction de p :

```
# p   n(p)
  0    1
  1    3
  2    7
  3   15
  4   31
```

On trouve facilement une relation de récurrence non homogène :

$$n(p) \;=\; 2\,n(p-1) + 1\,. \tag{3.5}$$

Le polynôme caractéristique de la relation homogène associée, $C(z) = z - 2$, a une racine évidente, simple, $\beta = 2$. Toute solution de la relation homogène a donc la forme $n(p) = a\,2^p$. On cherche maintenant une solution particulière de la relation (3.5). On commence par chercher une solution constante $n(p) = n(p-1) = b$. En substituant cette valeur dans la relation (3.5), on trouve $b = -1$. Toute solution de la relation (3.5) est donc de la forme

$$n(p) = a\,2^p - 1\,.$$

En utilisant la condition initiale $n(0) = 1$, on trouve la valeur de la dernière constante, $a = 2$. En inversant la formule, on trouve $p(n) = \log_2(n+1) - 1$.

Récurrence du type « diviser pour régner ». Soit à résoudre une récurrence de la forme suivante, qui apparaît naturellement dans des méthodes telles que la méthode dichotomique, ou, plus généralement, dans toutes les méthodes du type « diviser pour régner ».

$$f(n) = f\left(\frac{n}{2}\right) + 1\,, \quad f(1) = 1\,.$$

L'idée consiste à faire l'hypothèse que n est une puissance de deux (mettons $n = 2^p$) et à poser $g(p) = f(2^p) = f(n)$. En effet, on a alors $g(p-1) = f(n/2)$ et, dans les nouvelles variables, on a affaire à une récurrence classique :

$$g(p) = g(p-1) + 1\,, \quad g(0) = 1\,.$$

Sur l'exemple, la solution est évidente $g(p) = p + 1$. Comme $n = 2^p$, on a $p = \log_2(n)$ et, dans les anciennes variables, la solution s'écrit :

$$f(n) = \log_2(n) + 1\,.$$

Bibliographie

[1] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction à l'algorithmique*. Dunod, Paris, 2ème edition, 2002.

[2] Donald Erwin Knuth. Big Omicron and big Omega and big Theta. In *SIGACT News*, pages 18–24, April-June 1976.

[3] Robert Sedgewick and Philippe Flajolet. *An Introduction to the Analysis of Algorithms*. Addison Wesley, 1996.

Chapitre 4

Recherche de données

Dans ce chapitre, on étudie des structures de données qui permettent de rechercher une donnée dans un ensemble (pour retrouver une information qui lui est associée ou tout simplement déterminer si la donnée fait partie de l'ensemble). Une structure de données qui fournit une telle fonctionnalité (avec l'ajout d'un élément à l'ensemble et, parfois, la suppression d'un élément de l'ensemble) constitue ce qu'on appelle un dictionnaire [1, Partie 3, page 191].

Les structures de données et les algorithmes de recherche sont illustrées avec le projet LINKER, dont les détails sont présentés au chapitre 5. Une version abstraite est décrite dans les figures 4.1 et 4.2. Le dictionnaire maintenu par l'algorithme du projet LINKER s'appelle une « table des symboles ».

```
function linker_abstrait
  reçoit une liste de fichiers objets et de bibliothèques sur sa ligne de commandes
begin
  vider la table des symboles T
  for chaque fichier listé sur la ligne de commandes do
    if ce fichier est un fichier objet O then
      enregistrer tous les symboles de O dans T (figure 4.2)
    else (c'est donc une bibliothèque B)
      do
        n'enregistrer les symboles d'un fichier objet O de B dans T, que si O
          fournit une définition à un symbole indéfini de T
      while au moins un fichier objet de B a été incorporé dans T
    end if
  end do
end
```

FIGURE 4.1 – Version abstraite de l'algorithme du projet LINKER.

```
function enregistrer (s, T)
  enregistre un symbole s dans la table des symboles T = {s_1, ..., s_m}
begin
  if ∃ i tel que s_i et s ont même identificateur then
    if s est un symbole d'un type « défini » then
      if s_i est un symbole de type « indéfini » then
        changer le type de s_i pour celui de s
      end if
      incrémenter de 1 le nombre de fois où s_i est défini
    end if
  else
    fixer à 0 ou à 1 le nombre de fois où s est défini,
      suivant que s est d'un type « défini » ou pas
    T = T ∪ {s}
  end if
end
```

FIGURE 4.2 – Version abstraite de la fonction qui enregistre un symbole dans la table des symboles. Remarquer que tout enregistrement commence par une recherche.

4.1 La complexité étudiée

Dans ce chapitre, on étudie le comportement aymptotique de la fonction $f(n)$ (voir chapitre 3), qui donne le nombre de comparaisons de chaînes de caractères en fonction du nombre n d'accès à la table. Dans le cadre du projet LINKER, cette fonction $f(n)$ est donnée, pour chaque exemple, par les deux premières colonnes du fichier `"linker.stats"`. Il est intéressant d'étudier le comportement de la fonction $f(n)$ non seulement sur un exemple réel, mais aussi dans le pire des cas.

Dans le pseudo-code de la figure 4.1, le pire des cas se produit quand le symbole s_i est systématiquement enregistré dans T. Dans le cadre du projet, on l'obtient en considérant un unique fichier objet, ne comportant que des symboles définis ou indéfinis.

Dans le pseudo-code de la figure 4.1, le meilleur des cas se produit quand un seul [1] symbole s_i est enregistré dans T. Dans le cadre du projet, on l'obtient en considérant n fichiers objets : le premier définit un symbole s, les autres ne font que des références à s.

4.2 Implantation avec un tableau non ordonné

Une première méthode consiste à implanter la table des symboles par un tableau non ordonné : les symboles sont stockés, suivant leur ordre d'arrivée, dans la table.

1. On choisit de ne pas considérer le cas où aucun symbole n'est enregistré dans T.

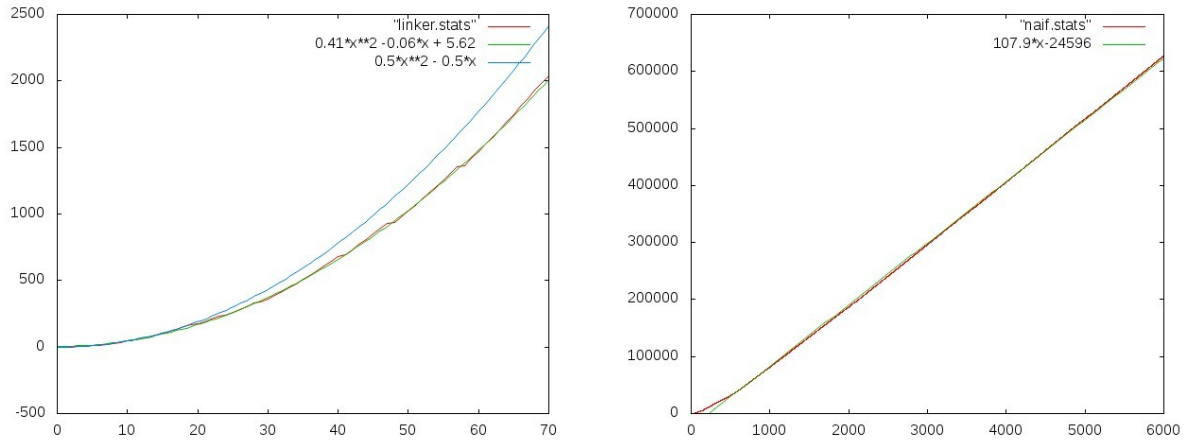


FIGURE 4.3 – Implantation avec un tableau désordonné. À gauche, la première phase de l'algorithme avec la courbe expérimentale, celle obtenue par estimation de paramètres et celle du pire des cas. À droite, la seconde phase avec la courbe expérimentale (fichier `"naif.stats"`) et celle obtenue par estimation de paramètres.

4.2.1 Implantation

L'implantation repose sur un tableau redimensionnable. Le champ `mesures` contient des mesures permettant d'analyser le comportement de la table et de produire le fichier `"linker.stats"`.

```
struct tableau
{   int alloc;            /* le nombre d'emplacements alloués */
    int size;             /* le nombre d'emplacements utilisés */
    struct symbole* tab;  /* la zone de stockage */
};

struct symtable
{   struct tableau T;
    struct stats mesures; /* mesures */
};
```

La recherche d'un symbole dans la table est effectuée par un parcours séquentiel des éléments du tableau. Les nouveaux éléments sont enregistrés en fin de tableau.

4.2.2 Analyse

Dans le meilleur des cas, on a $f(n) \simeq n$ puisque, à chacune des n itérations (sauf la première), on effectue exactement 1 comparaison de chaîne.

Le pire des cas

Le pire des cas est un peu plus compliqué : à la première itération, il n'y a aucune comparaison effectuée, à la deuxième, il y en a une, à la troisième, il y en a deux, jusqu'à la enième, où $n-1$ comparaisons sont effectuées. Au total,

$$f(n) = 0 + 1 + 2 + \cdots + (n-1) = \frac{n\,(n-1)}{2} = \frac{1}{2}\,n^2 - \frac{1}{2}\,n\,.$$

Un cas réel

À quoi faut-il s'attendre sur un cas réel ? Pour toutes les implantations, on a choisi de tester le programme `linker` sur lui-même, en exécutant la commande (le répertoire comporte dix fichiers objets) :

```
$ ./linker *.o /usr/lib/libc.a
```

On assiste à un comportement en deux phases. Durant la première phase, les symboles considérés sont ceux présents dans les fichiers objets. Ils vont être systématiquement incorporés dans la table des symboles mais ils ne vont pas systématiquement faire augmenter la dimension de cette table. On devrait donc avoir une fonction $f(n) \simeq a\,n^2 + b\,n + c$ pour certaines valeurs de a, b et c. Durant la seconde phase, les symboles considérés sont ceux de la bibliothèque standard. Il y a de nombreuses comparaisons (la bibliothèque est parcourue quatre fois de suite) mais la dimension de T va très peu augmenter. On devrait alors avoir une fonction $f(n) \simeq a\,n + b$, pour certaines valeurs de a et de b. La constante a devrait être approximativement égale à la dimension de la table T finale (112 symboles).

Au vu de l'évolution de la troisième colonne du fichier `"linker.stats"` (qui donne la dimension de la table T), la première phase s'arrête vers $n = 70$. On peut donc estimer les coefficients a, b et c par la commande suivante. On trouve une fonction qui croît un peu moins vite que le pire des cas.

```
$ gnuplot
gnuplot> fit [0:70] a*x**2+b*x+c "linker.stats" via a,b,c

Final set of parameters            Asymptotic Standard Error
=======================            ==========================

a               = 0.413054         +/- 0.00349      (0.8449%)
b               = -0.0600846       +/- 0.2557       (425.5%)
c               = 5.62386          +/- 3.933        (69.94%)
```

Pour la deuxième phase, on peut estimer les paramètres a et b par la commande suivante. Le coefficient directeur $a \simeq 108$ n'est pas très éloigné de la valeur prédite 112. Graphiquement, on a bien le sentiment d'observer une croissance linéaire. Voir figure 4.3.

```
gnuplot> fit a*x+b "linker.stats" via a,b

Final set of parameters            Asymptotic Standard Error
=======================            ==========================

a               = 107.974          +/- 0.03241      (0.03002%)
b               = -24596.1         +/- 115.8        (0.4706%)
```

La visualisation peut s'obtenir par la commande :

```
gnuplot> plot "linker.stats" with lines, 107.9*x - 24596
```

4.3 Implantation avec un tableau ordonné

4.3.1 Implantation

La déclaration de type est la même que précédemment mais les algorithmes mis en œuvre sont différents. La recherche d'un élément s'effectue par une méthode dichotomique. Voir la section 3.2.2. Les nouveaux éléments sont ajoutés à un emplacement précis du tableau, pour préserver le fait que le tableau est trié. Cet emplacement est déterminé lui-aussi par une méthode dichotomique, en adaptant un peu l'algorithme de la section 3.2.2. L'ajout d'un élément implique donc une translation d'une partie du tableau.

4.3.2 Analyse

Le pire des cas

Le pire des cas est intéressant parce qu'il soulève une question : la fonction $f(n)$ ne compte que des comparaisons de symboles. Mais ne faudrait-il pas compter aussi le coût de la translation des éléments du tableau lors de l'ajout des nouveaux éléments ?

Supposons, pour commencer, qu'on ne compte que les comparaisons de symboles. À l'itération numéro i, on effectue une recherche dichotomique dans un tableau de taille i. En nous inspirant des résultats établis dans la section 3.2.2, on est tenté de considérer que chaque recherche coûte $\log_2(i)$ comparaisons et donc que la fonction $f(n)$ devrait être, approximativement, de la forme :

$$f(n) \simeq \log_2(1) + \log_2(2) + \cdots + \log_2(n)\,.$$

Chaque logarithme de la somme peut être majoré par $\log_2(n)$. La somme peut donc être majorée par $n \log_2(n)$. On s'attend donc à ce que $f(n) \in O(n \log(n))$. La vraie courbe du pire des cas est en fait une courbe « continue », obtenue en raccordant des segments de droites. On peut mener, pour cette courbe, une analyse aussi fine que celle menée pour la recherche dichotomique seule, dans la section 3.2.2. On peut ainsi montrer (les calculs

sont un peu trop longs pour être détaillés ici), non seulement que notre borne supérieure asymptotique est correcte, mais aussi que

$$f(n) \in \Theta(n \log(n)) .$$

Supposons maintenant qu'on intègre le coût des translations dans la fonction $f(n)$. À l'itération numéro i, on effectue une recherche dichotomique dans un tableau de taille i plus une translation de i éléments. Dans ce cas, la fonction $f(n)$ devrait être, approximativement, de la forme :

$$f(n) \simeq 1 + 2 + \cdots + n + \log_2(1) + \log_2(2) + \cdots + \log_2(n) .$$

La somme des logarithmes peut être négligée et on retrouve la complexité, dans le pire des cas, des tableaux non ordonnés :

$$f(n) \in \Theta(n^2) .$$

Conclusion. En général, on évite d'implanter un dictionnaire par un tableau trié, par crainte du surcoût lié aux translations d'éléments. Voir toutefois [1, Problème 17.2, page 416].

Cet exemple met en lumière les remarques faites en début de chapitre 3 : pour étudier la complexité d'un algorithme, on est obligé d'abstraire le problème, c'est-à-dire d'étudier un problème mathématique un peu arbitraire, dont on espère qu'il reflète les points essentiels du comportement du vrai algorithme. Dans ce cas-ci, on ne devrait pas négliger le coût des translations.

Un cas réel

Sur le code du projet LINKER, on observe à nouveau un comportement en deux phases. On a compté chaque translation d'un symbole avec compteur comme une comparaison. Lors de la première phase, la courbe a une forme difficile à reconnaître. Lors de la seconde, on observe un comportement linéaire $f(n) \simeq a\,n + b$. La constante a étant très proche du logarithme en base 2 du nombre de symboles présents dans la table. On est très loin du pire des cas, en raison du très petit nombre d'ajouts d'éléments par rapport au nombre de recherches de symboles. Les courbes expérimentales sont données figure 4.4.

```
gnuplot> fit a*x+b "linker.stats" via a,b

Final set of parameters            Asymptotic Standard Error
=======================            ==========================

a               = 6.84046          +/- 0.001675     (0.02449%)
b               = 1173.85          +/- 5.983        (0.5097%)

gnuplot> print log(112)/log(2)
6.8073549220576
```

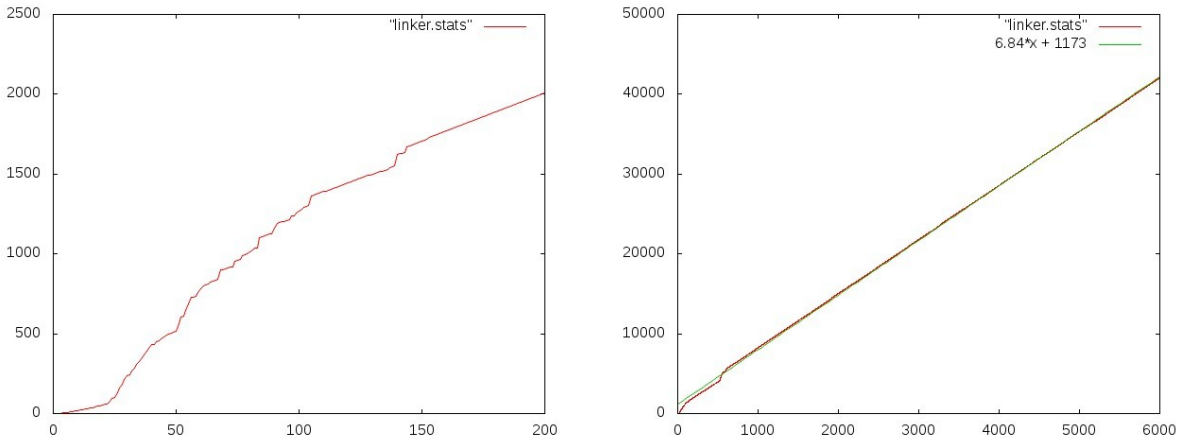


FIGURE 4.4 – Implantation avec un tableau ordonné. À gauche, la courbe expérimentale de la première phase de l'algorithme. À droite, la seconde phase avec la courbe expérimentale et celle obtenue par estimation de paramètres.

4.4 Implantation par arbres binaires de recherche

Pour éviter le comportement d'un tableau ordonné dans le pire des cas, on peut utiliser une méthode fondée sur les « arbres binaires de recherche » qui présente les mêmes avantages que la méthode dichotomique mais sans le surcoût dû aux translations.

4.4.1 Arbres binaires

Un arbre binaire [2] A est la donnée d'un ensemble de « nœuds » et « d'arcs ». Un arc est un couple de nœuds.

Il existe un arbre particulier, « l'arbre vide », qui ne contient ni nœud, ni arc. Les arbres non vides ont au moins un nœud.

Si (a, b) est un arc, alors on dit que a est le « père » de b et que b est un « fils » de a. Chaque nœud a exactement un père, à l'exception d'un nœud particulier, appelée « racine » de l'arbre A. Un nœud peut avoir zéro, un ou deux fils (c'est le maximum quand l'arbre est binaire). Les nœuds qui n'ont aucun fils sont appelés des « feuilles » de l'arbre A.

Un « chemin » dans un arbre A est une suite d'arcs appartenant à A, qui se suivent :

$$(a_1, a_2),\ (a_2, a_3),\ \ldots,\ (a_{n-1}, a_n)\,.$$

La « longueur » d'un chemin est le nombre d'arcs qui le composent ($n-1$ sur l'exemple). La « hauteur » d'un nœud a est le maximum des longueurs des chemins qui commencent en a. La hauteur d'un arbre est la hauteur de sa racine.

2. On utilise le terme « arbre » au lieu de « arborescence » mais « arborescence » serait plus juste. Cet abus de langage est classique. Voir [1, chapitre 12, page 247, note en bas de page].

Dans un arbre binaire, il est souvent utile de distinguer les deux fils d'un nœud a. On les appelle alors, le « fils gauche » et le « fils droit » du nœud a. Un nœud qui n'a qu'un seul fils peut donc avoir, soit un fils gauche, soit un fils droit (ce n'est pas équivalent).

Le « sous-arbre gauche » d'un nœud a est l'arbre obtenu en ne gardant que les nœuds de l'arbre A qui sont accessibles à partir du fils gauche de a ainsi que les arcs où ils apparaissent. Le fils gauche de a est donc la racine du sous-arbre gauche de a. On définit le « sous-arbre droit » de a de façon similaire.

```
function recherche_dans_un_ABR (racine, valeur)
begin
  a = racine
  trouvé = false
  while a ≠ NIL et non trouvé do
    if la valeur de a est égale à valeur then
      trouvé = true
    elif la valeur de a est inférieure à valeur then
      a = le fils droit de a
    else
      a = le fils gauche de a
    end if
  end do
  return trouvé
end
```

FIGURE 4.5 – Recherche d'un élément dans un arbre binaire de recherche

Un point d'algorithmique

Un arbre qui n'est pas vide est complètement défini par sa racine. Plutôt que d'écrire des fonctions paramétrées par des arbres, on écrit donc plutôt des fonctions paramétrées par des racines, c'est-à-dire des nœuds. Ce choix peut sembler bizarre mais il rend le pseudo-code plus proche des implantations en langage C. Un problème se pose alors avec l'arbre vide, qui n'a normalement pas de racine. On le contourne en assimilant l'arbre vide à une racine conventionnelle, notée NIL [1, section 10.4, page 208].

4.4.2 Arbres binaires de recherche

Un « arbre binaire de recherche » (ABR) A est un arbre binaire particulier [1, chapitre 12, page 247]. À chaque nœud, on associe une valeur (dans notre cas : un symbole avec compteur) et on exige que la propriété suivante soit vérifiée pour tout nœud a de A :

```
function ajout_dans_un_ABR (racine, valeur)
  Cette fonction modifie l'arbre désigné par racine.
  On suppose que valeur n'appartient pas à l'arbre.
begin
  Créer une nouveau nœud (une feuille), de valeur valeur
  if racine = NIL then
    résultat = la nouvelle feuille
  else
    pred = indéfini
    succ = racine
On est certain d'entrer au moins une fois dans la boucle
    while succ ≠ NIL do
      pred = succ
      if la valeur de succ est inférieure à valeur then
        succ = le fils droit de succ
      else
        succ = le fils gauche de succ
      end if
    end do
    if la valeur de pred est inférieure à valeur then
      Modifier le fils droit de pred pour qu'il pointe vers la nouvelle feuille
    else
      Modifier le fils gauche de pred pour qu'il pointe vers la nouvelle feuille
    end if
    résultat = racine
  end if
  return résultat
end
```

FIGURE 4.6 – Ajout d'un élément dans un arbre binaire de recherche

> Tout symbole qui apparaît dans le sous-arbre gauche de a est inférieur [3] lexicographiquement [4] au symbole de a. Tout symbole qui apparaît dans le sous-arbre droit de a est supérieur au symbole de a.

La recherche d'un élément dans un arbre binaire de recherche peut se faire par l'algorithme de la figure 4.5, qui est un analogue de la méthode dichotomique. L'ajout d'un élément peut se faire par l'algorithme de la figure 4.6.

3. Dans notre cas, tous les symboles sont distincts deux-à-deux : on ne se soucie pas de la différence entre « strictement inférieur » et « inférieur ou égal ».

4. C'est-à-dire suivant l'ordre du dictionnaire.

L'efficacité de ces deux algorithmes dépend en fait considérablement de la forme de l'arbre.

Si l'arbre est équilibré *en nombre de nœuds*, c'est-à-dire si, pour tout nœud a, le nombre de nœuds du sous-arbre gauche de a est égal au nombre de nœuds du sous-arbre droit de a (plus ou moins 1) alors les deux algorithmes se comportent vraiment comme une méthode dichotomique et le nombre de comparaisons de valeurs effectuées lors d'un ajout ou d'une recherche infructueuse dans un arbre de p nœuds est approximativement égal au logarithme en base 2 de p.

Par contre, un arbre peut fort bien avoir la forme d'une liste chaînée. Dans ce cas, le nombre de comparaisons de valeurs effectuées est égal à p. Une telle configuration apparaît si un arbre est construit avec la fonction de la figure 4.6, en ajoutant une séquence de valeurs déjà triées (par ordre croissant ou décroissant).

4.4.3 Implantation

On implante un arbre binaire de recherche au moyen de la structure suivante. Un nœud est un pointeur sur un `struct ABR`. L'arbre NIL est codé par le pointeur nul.

```
struct ABR
{   struct ABR* gauche;   /* sous-arbre gauche (symboles plus petits) */
    struct ABR* droit;    /* sous-arbre droit (symboles plus grands)*/
    struct symbole value; /* valeur du noeud */
};
#define NIL (struct ABR*)0
```

La structure `struct symtable`, qui sert à implanter les tables de symboles, est adaptée comme suit :

```
struct symtable
{   struct ABR* arbre;    /* la racine de l'ABR */
    struct stats stats;   /* mesures */
};
```

4.4.4 Analyse

Le pire des cas

Le pire des cas n'a pas changé par rapport à l'implantation par un tableau désordonné ! En effet, dans le pire des cas, les symboles arrivent déjà triés dans l'ordre lexicographique, l'arbre a la forme d'une liste et on retrouve :

$$f(n) = \frac{n\,(n-1)}{2}.$$

Un cas réel

Dans un cas réel (le code du projet LINKER), heureusement, le comportement asymptotique s'améliore nettement. On observe à nouveau un comportement en deux phases. La première correspond au traitement des fichiers objets. La courbe expérimentale ressemble à une succession de petites paraboles (une sorte de pire des cas par morceaux). Ce phénomène surprenant semble dû au fait que les symboles extraits d'un même fichier objet sont donnés par ordre croissant. La seconde phase correspond au traitement de la bibliothèque standard. On s'attend à trouver une fonction $f(n) \simeq a\,n + b$. Si de plus, l'arbre est équilibré en nombre de nœuds, on s'attend à ce que a soit approximativement égal au logarithme en base 2 du nombre de symboles, qui vaut $\log_2(112) \simeq 6.8$. On trouve plutôt, par estimation de paramètres, une valeur de l'ordre de 18, ce qui montre que l'arbre n'est pas vraiment équilibré. La hauteur de l'arbre est de 26. Voir figure 4.7.

```
gnuplot> fit a*x+b "linker.stats" via a,b

Final set of parameters            Asymptotic Standard Error
=======================            ==========================

a               = 17.9296          +/- 0.001681     (0.009376%)
b               = -956.96          +/- 6.008        (0.6279%)
```

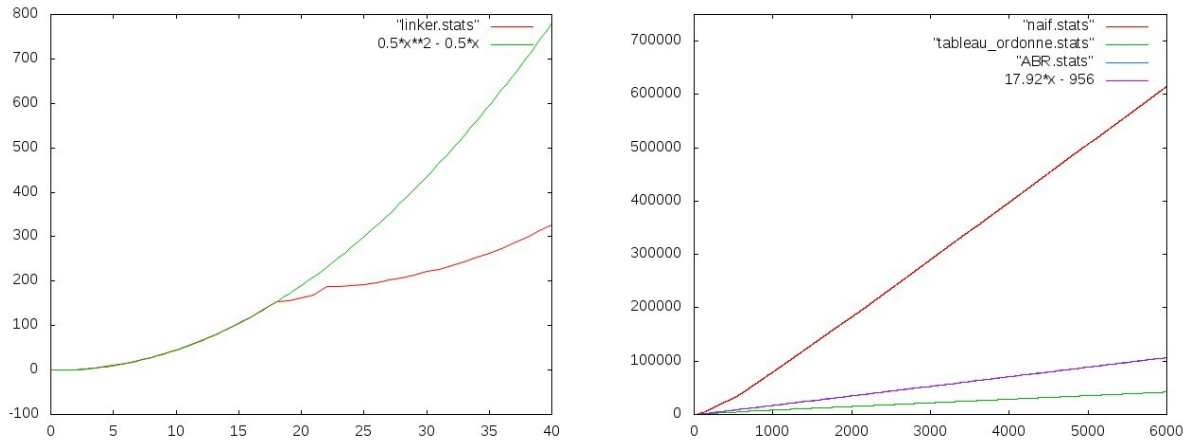


FIGURE 4.7 – Implantation avec arbres binaires de recherche. À gauche, la première phase de l'algorithme avec la courbe expérimentale et celle du pire des cas. À droite, la seconde phase. Tout en haut, la courbe expérimentale correspondant au tableau désordonné. Tout en bas, celle correspondant au tableau ordonné. Au milieu, celle correspondant à un ABR et celle obtenue par estimation de paramètres (ces deux-là sont quasiment indiscernables).

On peut comparer ce résultat avec la hauteur moyenne d'un arbre binaire de recherche après n insertions d'éléments aléatoires, qui se comporte, asymptotiquement, comme $c \ln n$

où la constante c vaut approximativement 4.311 [2, Theorem 5.10, page 261]. Dans notre cas ($n = 112$), la hauteur devrait être 20 ou 21. On peut penser que la hauteur trouvée, 26, est moins bonne parce que les symboles des fichiers objets sont donnés par ordre croissant.

L'itérateur de symboles récupère la sortie de la commande `nm`, qui, par défaut, trie les symboles par ordre alphabétique. En s'arrangeant pour que l'itérateur exécute la commande `nm -p`, on évite ce tri et on améliore nettement les performances de la structure, sans pour autant atteindre les performances des AVL, décrits dans la section suivante.

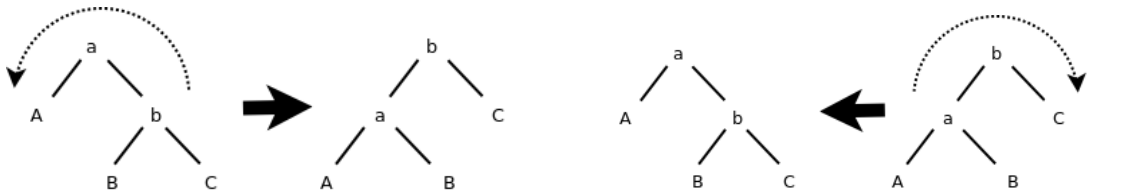
4.5 Implantation par arbres AVL

Le sigle « AVL » vient des initiales des inventeurs de la méthode : Adel'son-Vel'skii et Landis. Voir [1, Notes, page 293] et [2, chapter 5, pages 290-295].

Un arbre AVL est un arbre binaire de recherche particulier [1, section 13.3]. En effet, c'est un arbre équilibré *en hauteur* (ce qui n'est pas pareil que équilibré en nombre de nœuds) : pour chaque nœud a, la hauteur du sous-arbre gauche et celle du sous-arbre droit de a ne diffèrent au plus que de 1.

La recherche d'un élément dans un arbre AVL n'est pas aussi efficace que la recherche d'un élément dans un ABR équilibré en nombre de nœuds mais on dispose d'un algorithme efficace pour ajouter un élément dans un AVL, alors qu'on n'en connaît aucun pour les arbres équilibrés en nombre de nœuds.

Ajout d'un élément dans un arbre AVL. L'idée consiste, lorsqu'on ajoute un élément dans un arbre AVL, à l'ajouter comme dans un arbre ABR, puis à rééquilibrer l'ABR pour obtenir à nouveau un AVL. Les opérations de rééquilibrage s'appuient sur les opérations ci-dessous (les lettres minuscules représentent des nœuds, les majuscules des arbres, éventuellement vides). Ces opérations sont appelées « rotation gauche » et « rotation droite » dans [1, section 13.2].

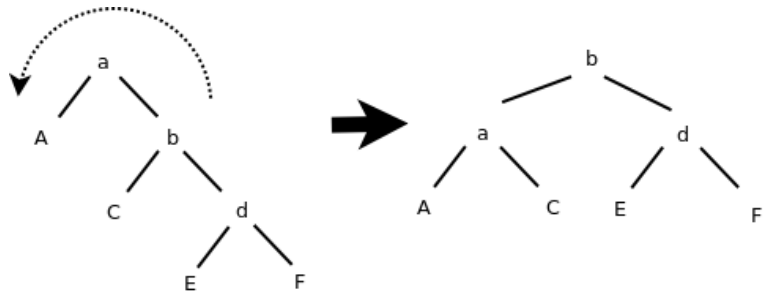


Rotation dans le sens trigonométrique, appliquée à a.

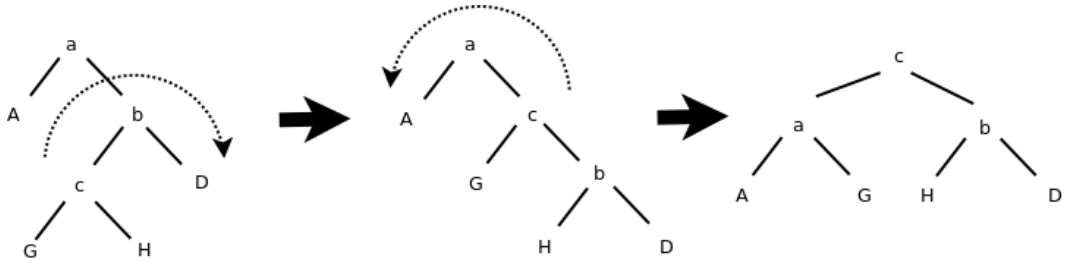
Rotation dans le sens trigonométrique inverse, appliquée à b.

Principe du rééquilibrage. Considérons un AVL de racine a avec un sous-arbre gauche A et un sous-arbre droit B. Supposons qu'après ajout d'un nouvel élément, l'arbre ne soit plus un AVL, parce que B est devenue trop haut et donc parce que la hauteur de A est devenue égale à la hauteur de B moins 2. L'arbre B comporte alors nécessairement une racine b avec un sous-arbre gauche C et un sous-arbre droit D.

Premier cas. Le nouvel élément a été inséré dans l'arbre D, qui comporte donc nécessairement une racine d avec un sous-arbre gauche E et un sous-arbre droit F. Il suffit d'une rotation dans le sens trigonométrique, appliquée à a, pour rééquilibrer l'arbre :



Second cas. Le nouvel élément a été inséré dans l'arbre C, qui comporte donc nécessairement une racine c avec un sous-arbre gauche G et un sous-arbre droit H. Il suffit d'enchaîner une rotation dans le sens trigonométrique inverse, appliquée à b, puis une rotation dans le sens trigonométrique, appliquée à a, pour rééquilibrer l'arbre :



Il y a bien sûr deux autres cas symétriques, à appliquer si le nouvel élément est inséré dans A plutôt que dans B.

Mise en œuvre. Pour commencer, on rajoute deux champs à la structure `struct ABR` pour mémoriser les hauteurs des sous-arbres gauche et droit de chaque nœud. On modifie la fonction de la figure 4.6 en lui ajoutant une pile de `struct ABR*` dans laquelle on mémorise le chemin suivi dans la boucle. À la fin de l'ajout, on parcourt le chemin en sens inverse, grâce à la pile et on vérifie que chaque nœud est équilibré en hauteur. S'il ne l'est pas, on lui applique les rotations décrites plus haut.

4.5.1 Analyse

Le pire des cas

Supposons qu'il y ait n symboles dans la table des symboles. Le nombre de comparaisons effectuées lors de la recherche infructueuse d'un symbole est égale à 1 plus la hauteur de l'arbre binaire de recherche associé à la table des symboles. Que peut valoir cette hauteur, pour un arbre AVL, dans le pire des cas ?

Notons $F(h)$ le nombre de nœuds minimal pour obtenir un arbre AVL de hauteur $h-1$, c'est-à-dire un arbre qui provoque h comparaisons de symboles, dans le pire des cas. On a $F(h) = F(h-1) + F(h-2) + 1$ avec $F(1) = 1$ et $F(2) = 2$, c'est-à-dire une relation de récurrence proche de celle de la suite de Fibonacci. On trouve (voir la section 3.3) :

$$F(h) = a \left(\frac{1+\sqrt{5}}{2}\right)^h + b \left(\frac{1-\sqrt{5}}{2}\right)^h - 1\,,$$

où a et b désignent deux constantes. Notons $\varphi = (1+\sqrt{5})/2$ le « nombre d'or ». En travaillant les formules (toujours avec un logiciel de calcul formel), on trouve que $F^{-1}(n) \leq \log_\varphi n$. On en conclut que le nombre de comparaisons effectuées lors de la recherche infructueuse d'un symbole dans une table de n symboles est, dans le pire des cas, inférieur ou égal à $\log_\varphi n$. Par conséquent,

$$f(n) \leq \sum_{i=1}^{n} \log_\varphi i \leq n \log_\varphi n = \left(\frac{1}{\log_2 \varphi}\right) n \log_2 n \leq 1.45\ n \log_2 n\,.$$

Expérimentalement, on constate, de plus, que $n \log_2 n \leq f(n)$, ce qui tend à montrer que $f(n) \in \Theta(n \log(n))$ et que la constante 1.45 est assez précise. On trouve un résultat à peine moins bon que celui qu'on trouverait si on utilisait des arbres équilibrés en nombre de nœuds au lieu d'arbres AVL. C'est un très bon résultat.

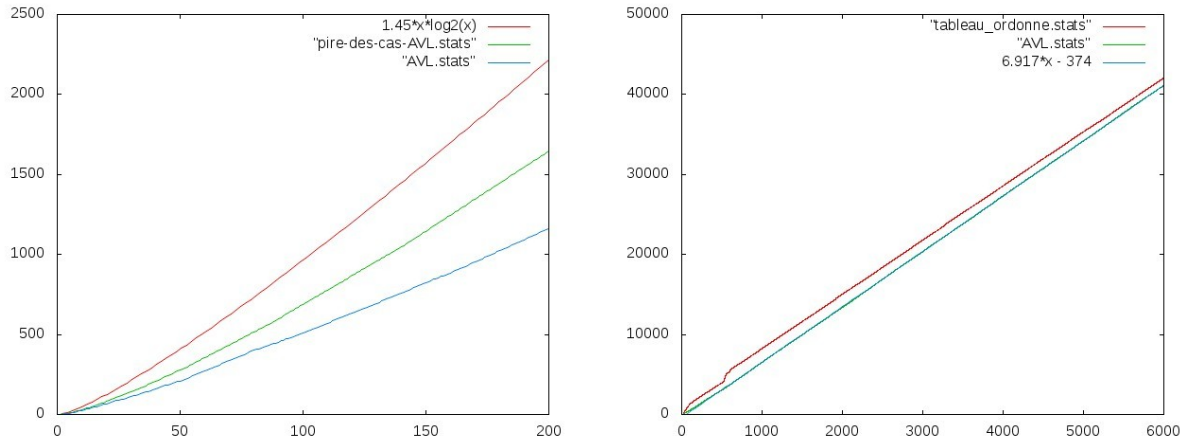


FIGURE 4.8 – Implantation avec AVL. À gauche, la première phase de l'algorithme avec la courbe expérimentale, celle du pire des cas et la courbe théorique qui majore le pire des cas. À droite, la seconde phase avec la courbe expérimentale correspondant à un tableau ordonné, celle correspondant à un AVL et la courbe obtenue par estimation de paramètres (ces deux-là étant quasiment indiscernables). L'implantation avec AVL est meilleure qu'avec un tableau ordonné, parce que son comportement est meilleur lors de la première phase.

Un cas réel

Dans un cas réel (toujours le code du projet LINKER), il devient difficile de distinguer les deux phases. Par estimation de paramètres, on trouve une fonction $f(n) \simeq a\,n + b$ avec $a \simeq 6.91$, ce qui est proche de $\log_2(112) \simeq 6.80$. La hauteur du dictionnaire est de 8.

```
gnuplot> fit a*x+b "linker.stats" via a,b

Final set of parameters            Asymptotic Standard Error
=======================            ==========================

a               = 6.91759          +/- 0.0003905    (0.005644%)
b               = -374.431         +/- 1.396        (0.3728%)
```

4.5.2 Parcours d'arbres binaires

function imprimer_ABR (*racine*)
Cette fonction imprime les valeurs de l'ABR désigné par racine. *Le parcours « gauche-racine-droite » a pour effet d'imprimer les valeurs par ordre alphabétique.*
begin
  if *racine* $\neq$ NIL then
    imprimer_ABR (le fils gauche de *racine*)
    imprimer la valeur de *racine*
    imprimer_ABR (le fils droit de *racine*)
  end if
end

FIGURE 4.9 – Impression des valeurs d'un arbre binaire de recherche. Implantation récursive d'un parcours « gauche-racine-droite ».

La fonction `synthese_symtable` du projet LINKER est amenée à parcourir la table des symboles pour lister les symboles indéfinis et ceux définis plusieurs fois. Une façon simple de procéder consiste à implanter une fonction récursive, sur le modèle de la fonction de la figure 4.9. Une autre solution, itérative, est donnée dans la figure 4.10. Cette deuxième solution, qui utilise une pile d'ABR, peut être adaptée pour définir un itérateur d'ABR (une structure de données qui peut être fort utile). Il suffit d'incorporer la pile dans la structure de données qui implante l'itérateur. La boucle d'initialisation doit être placée dans la fonction qui positionne l'itérateur en début d'ABR. La boucle qui énumère les valeurs doit être un peu adaptée et placée dans la fonction qui fait avancer l'itérateur.

```
function imprimer_ABR (racine)
  Cette fonction imprime les valeurs de l'ABR désigné par racine. Le parcours
  « gauche-racine-droite » a pour effet d'imprimer les valeurs par ordre alphabétique.
local variables
  Une pile P d'ABR
begin
  vider la pile P
  a = racine
Boucle d'initialisation
  while a ≠ NIL do
    empiler a dans P
    a = le fils gauche de a
  end do
Invariant : on a fini de traiter le sous-arbre gauche de l'ABR en sommet de pile
  while P n'est pas vide do
    dépiler un ABR de P dans a
    imprimer la valeur de a
    a = le fils droit de a
    while a ≠ NIL do
      empiler a dans P
      a = le fils gauche de a
    end do
  end do
end
```

FIGURE 4.10 – Impression des valeurs d'un arbre binaire de recherche. Implantation itérative d'un parcours « gauche-racine-droite ».

4.6 Implantation par tables de hachage

On commence par une présentation simple. On ajoutera les complications ensuite. Une table de hachage est un tableau T de N emplacements, appelés « alvéoles ». Un alvéole est soit libre, soit occupé. S'il est occupé, il contient un ou plusieurs éléments (dans notre cas, des symboles). Parmi les trois champs, qui constituent un symbole, l'identificateur joue un rôle particulier. Il constitue ce qu'on appelle la « clef » de l'élément. Les autres champs sont les données satellites de l'élément.

Pour associer un alvéole à un élément, on utilise une « fonction de hachage », c'est-à-dire une fonction $h(s)$, paramétrée par la clef s d'un élément quelconque et qui retourne un indice entre 0 et $N-1$. La fonction h doit être déterministe. À part cela, tous les coups sont permis, l'idéal étant que h paraisse « aléatoire ».

On aimerait bien que des clefs différentes aient des valeurs de hachage différentes, c'est-à-dire $s \neq s' \Rightarrow h(s) \neq h(s')$. Si c'était le cas, en effet, pour tester si un élément de

clef s appartient à la table, il suffirait de tester si l'alvéole d'indice $h(s)$ est occupé. Pour l'enregistrer dans la table, il suffirait de l'affecter à l'alvéole d'indice $h(s)$ et de mettre cet alvéole dans l'état occupé.

En pratique, même si la table contient beaucoup d'alvéoles libres, il arrive que des clefs différentes aient même valeur de hachage [5]. On dit alors qu'il se produit une « collision ».

Une façon simple de résoudre le problème des collisions consiste à stocker des listes d'éléments, plutôt que des éléments, dans les alvéoles. D'autres méthodes, dites « d'adressage ouvert » évitent le recours à des structures de données auxiliaires telles que les listes. On se concentre ci-dessous sur une méthode particulière d'adressage ouvert : le « double hachage ».

Double hachage. Avec cette méthode, la fonction de hachage retourne non pas une valeur de hachage mais deux : $h(s) = (h_1(s), h_2(s))$. Les deux valeurs de hachage sont comprises entre 0 et $N-1$. La valeur $h_2(s)$ doit être non nulle.

Pour tester si un élément e de clef s est présent dans la table, on teste si l'élément se trouve dans l'alvéole d'indice $h_1(s)$. Si cet alvéole est occupé par un autre élément que e, on teste l'alvéole d'indice $h_1(s) + h_2(s) \mod N$. Si celui-ci aussi est occupé par un autre élément que e, on teste l'alvéole d'indice $h_1(s) + 2\,h_2(s) \mod N$ et ainsi de suite, jusqu'à trouver, soit e, soit un alvéole libre. De même, pour enregistrer un élément e de clef s dans la table, il suffit de chercher un emplacement libre en énumerant les alvéoles d'indice

$$h_1(s) + i\,h_2(s) \mod N\,, \quad i = 0,\, 1,\, 2,\, \ldots \tag{4.1}$$

Choix de N. Supposons que la table contienne encore un alvéole libre, d'indice k. Est-on certain que la formule (4.1) le trouve, c'est-à-dire qu'il existe un entier i tel que $h_1(s) + i\,h_2(s) = k \mod N$? Oui, si le plus grand diviseur commun de $h_2(s)$ et de N vaut 1 [1, chapitre 31.4]. Dans ce cas en effet, $h_2(s)$ est inversible modulo N et i vaut $(k - h_1(s))\,h_2(s)^{-1} \mod N$. Cette difficulté se résout très facilement en choisissant un nombre premier pour N.

Suppression d'un élément. Si on utilise une technique d'adressage ouvert, il faut gérer avec finesse les suppressions d'éléments. La solution la plus simple consiste à définir trois états pour les alvéoles, au lieu de deux : l'état libre, l'état occupé et l'état détruit. Attention aux algorithmes !

Saturation d'une table. L'adressage ouvert présente enfin un inconvénient sur la méthode des listes chaînées : la table peut devenir pleine et il n'est pas simple de redimensionner la table. En pratique, on considère qu'une table de hachage est pleine dès que son taux de remplissage α (égal au nombre d'alvéoles occupés [6] divisé par le nombre total d'alvéoles) est

5. Supposons que les valeurs de hachage soient tirées de façon équiprobable dans l'intervalle $[0, N-1]$ et notons $p(n)$ la probabilité que n valeurs de hachages aient des valeurs différentes deux-à-deux. On a $p(1) = 1$ et $p(n) = p(n-1)\,(N-n+1)/N$. Un calcul rapide montre que, pour $N = 400$, on a $p(24) < 1/2$. En d'autres termes, une collision est probable dès que le 24ème élément est enregistré dans la table. Ce phénomène est très lié au « paradoxe des anniversaires ».

6. Ou non vides, suivant les choix d'implantation.

supérieur à 1/3. C'est en effet à partir de ce taux que les performances de la structure de données commencent à se dégrader en moyenne (voir section 4.6.2).

Parcours d'une table. Il est parfois utile d'énumérer tous les éléments présents dans un dictionnaire. Dans le projet LINKER, la fonction `synthese_symtable` doit effectuer cette opération. Sans implantation d'un mécanisme supplémentaire, il est nécessaire de parcourir les N alvéoles or N peut être beaucoup plus grand que le nombre d'éléments présents dans la table. Autre inconvénient : les éléments sont alors énumérés dans le désordre.

4.6.1 Implantation

On implante une table de hachage utilisant la méthode du double hachage au moyen des structures suivantes. Le type `struct valeur_double_hachage` permet de représenter les deux valeurs de hachage retournées par les fonctions de hachage. Le type `enum etat_alveole` permet de coder les trois états possibles pour un alvéole (bien que dans le projet, l'état « détruit » ne soit pas utilisé). Le type `struct alveole` permet de représenter les alvéoles. Le type `fonction_double_hachage` fournit le prototype des fonctions de hachage. Le premier paramètre correspond à la clef ; le second est la table elle-même, dont la dimension est nécessaire pour calculer les valeurs de hachage.

```
struct valeur_double_hachage
{   int h1;
    int h2;
};

enum etat_alveole { alveole_vide, alveole_occupe, alveole_detruit };

struct alveole
{   struct symbole sym;     /* la valeur, si etat = alveole_occupe */
    enum etat_alveole etat; /* l'état de l'alvéole */
};

struct table_double_hachage;

typedef struct valeur_double_hachage
           fonction_double_hachage (char*, struct table_double_hachage*);

struct table_double_hachage
{   fonction_double_hachage* h;  /* pointeur vers la fonction de hachage */
    struct alveole* tab;         /* la zone de stockage */
    long N;                      /* nombre d'alvéoles alloués à tab */
};
```

Le constructeur de `struct table_double_hachage` est un peu particulier. Il reçoit la table T à initialiser en premier paramètre. Le deuxième paramètre est le nombre minimal d'alvéoles à allouer. Le troisième paramètre est la fonction de hachage à utiliser (s'il est nul, une fonction de hachage par défaut est choisie).

```
void init_table_double_hachage
      (struct table_double_hachage* T, int N0, fonction_double_hachage* h)
```

La structure `struct symtable`, qui sert à implanter les tables de symboles, est modifiée comme suit.

```
struct symtable
{   struct table_double_hachage T;  /* la table de hachage */
    struct stats mesures;           /* mesures */
};
```

4.6.2 Analyse

Le pire des cas

Dans le pire des cas, il y a systématiquement collision et l'algorithme de recherche d'un élément parcourt systématiquement tous les alvéoles occupés de la table. La fonction $f(n) \in O(n^2)$.

Un cas en moyenne

Il est beaucoup plus intéressant de procéder à une analyse de complexité en moyenne. On considère pour cela une table de N alvéoles, dont n sont occupés. Le taux de remplissage de la table est le réel $\alpha = n/N < 1$. Soit s un symbole n'appartenant pas à T. À combien de comparaisons de chaînes de caractères faut-il s'attendre pour que la fonction de recherche prouve que s n'appartient pas à T ? Cette fonction énumère des alvéoles d'indices $h_1(s) + i\,h_2(s)$ à partir de $i = 0$ jusqu'à trouver un alvéole vide. Pour pouvoir mener le calcul de complexité, on choisit de voir les indices énumérés comme des entiers de l'intervalle $[0,\ N-1]$ tous susceptibles d'être tirés avec la même probabilité. Le nombre de comparaisons de chaînes de caractères effectuées par la fonction de recherche devient donc une variable aléatoire X et on a :

x_i	0	1	2	3	4
$p(X = x_i)$	$1-\alpha$	$\alpha\,(1-\alpha)$	$\alpha^2\,(1-\alpha)$	$\alpha^3\,(1-\alpha)$	$\alpha^4\,(1-\alpha)$

Comme $\alpha < 1$, on a $p(X = x_i) = 0$ dès que $x_i > n$. L'espérance $E(X)$ de la variable aléatoire X vaut donc

$$E(X) = \sum_{i=0}^{n} i\,\alpha^i\,(1-\alpha)\,.$$

On en conclut que

$$E(X) < \sum_{i=0}^{\infty} i\,\alpha^i\,(1-\alpha) = \frac{\alpha}{1-\alpha}\cdot$$

Si $\alpha = 1/3$, on a $E(X) < 1/2$, ce qui signifie qu'on peut s'attendre, avec une fonction de hachage idéale, à une comparaison de chaînes de caractères tous les deux appels à la fonction de recherche.

Un cas réel

On a fait deux simulations avec le code du projet LINKER : une avec $N = 521$ et une autre avec $N = 16411$. On rappelle que la table reçoit 112 symboles. Même avec $N = 521$ (cas où la table est relativement bien remplie), les performances sont impressionnantes. Avec $N = 16411$, la fonction $f(n)$ est quasiment constante, ce qui signifie qu'il n'y a quasiment aucune collision. La forme un peu répétitive de la courbe est peut-être due au fait que la bibliothèque est parcourue quatre fois de suite. Un graphique est donné figure 4.11.

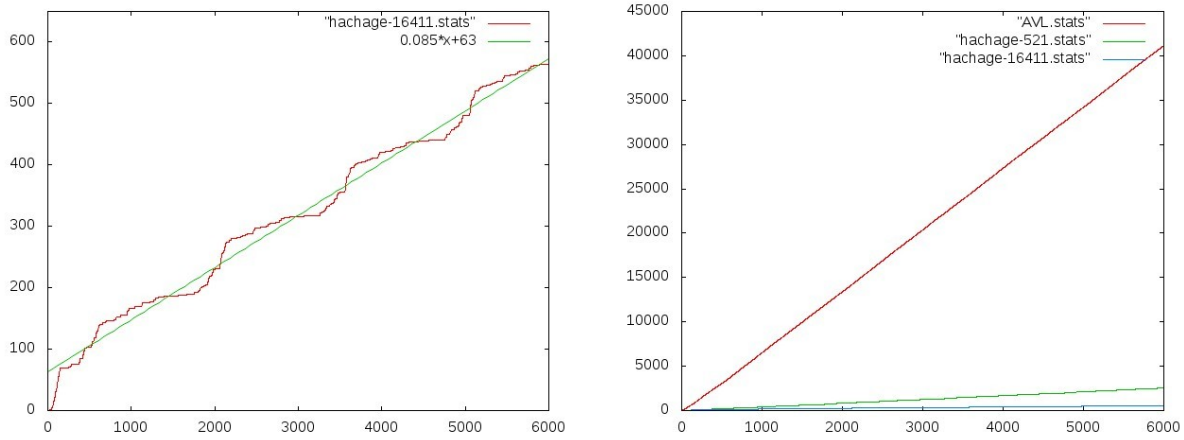


FIGURE 4.11 – Implantation avec table de hachage. À gauche, la courbe expérimentale pour $N = 16411$ et celle obtenue par estimation de paramètres. À droite, les deux courbes expérimentales (en bas) et celle correspondant aux AVL.

```
gnuplot> fit a*x+b "hachage-521.stats" via a,b

Final set of parameters            Asymptotic Standard Error
=======================            ==========================

a                 = 0.425538       +/- 0.0001248    (0.02932%)
b                 = -21.4653       +/- 0.4458       (2.077%)

gnuplot> fit a*x+b "hachage-16411.stats" via a,b
```

```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================

a                = 0.0854385       +/- 0.0001183    (0.1385%)
b                = 63.4569         +/- 0.4227       (0.6661%)
```

Voici la fonction de hachage par défaut du module `table_double_hachage`.

```
static struct valeur_double_hachage
                fonction_double_hachage_par_defaut
                        (char* clef, struct table_double_hachage* table)
{   struct valeur_double_hachage hashval;
    long p, i;

    p = table->N;
    hashval.h1 = 0;
    for (i = 0; clef [i] != '\0'; i++)
        hashval.h1 = (hashval.h1 + (i + 1) * clef [i] * clef [i]) % p;
    hashval.h2 = 1 + hashval.h1 % (p-1);
    return hashval;
}
```

4.7 Conclusion

Si le dictionnaire tient dans la mémoire vive de l'ordinateur, les tables de hachage fournissent une excellente implantation. Les arbres AVL (et d'autres variantes plus sophistiquées [1, chapitre 13, notes, page 293]) sont une bonne alternative si on est gêné par les défauts des tables de hachage : crainte du pire des cas, choix de la fonction de hachage, difficulté d'énumérer tous les éléments du dictionnaire par ordre croissant ou décroissant, difficulté de redimensionner la table lorsqu'elle devient saturée. Quant aux dictionnaires beaucoup plus gros, il vaut mieux les implanter avec des bases de données.

Bibliographie

[1] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction à l'algorithmique*. Dunod, Paris, 2ème edition, 2002.

[2] Robert Sedgewick and Philippe Flajolet. *An Introduction to the Analysis of Algorithms*. Addison Wesley, 1996.

Chapitre 5

L'algorithme de l'éditeur de liens

Dans ce chapitre, on présente une version un peu simplifiée de l'algorithme appliqué par l'éditeur de liens pour gérer les tables de symboles. On détaille son implantation dans le projet LINKER, qui sert de fil conducteur pour les méthodes du chapitre 4.

5.1 L'édition des liens

La génération d'un programme exécutable par un compilateur se fait en deux étapes principales :

1. la compilation proprement dite, au cours de laquelle le compilateur fabrique des fichiers « objets » à partir des fichiers sources,
2. l'édition des liens, au cours de laquelle l'éditeur de liens (le « *linker* » en Anglais) fabrique un exécutable à partir des fichiers objets et des bibliothèques.

Au cours de l'étape de compilation, le compilateur traduit le code source, écrit en C, en du code plus proche du langage machine mais, où les références aux fonctions et aux variables globales qui ne sont pas définies dans le code source, sont laissées « indéfinies ». Le travail de l'éditeur de liens consiste à donner une valeur aux symboles indéfinis en allant chercher les définitions manquantes, soit dans les autres fichiers objets, soit dans les bibliothèques, qui lui sont passés en paramètre.

5.1.1 Symboles locaux et symboles globaux

Avant de continuer, il est utile de faire un point rapide sur la notion de symbole. Essentiellement, un symbole est un identificateur de variable ou de fonction d'un programme exécutable ou d'un fichier objet, auquel correspond un emplacement précis dans ce programme ou ce fichier.

Les variables

Dans un programme C, il y a trois grands types de variables : les variables globales, les variables locales aux modules et les variables locales aux fonctions.

Les variables locales aux fonctions ne sont associées à aucun symbole. Leur emplacement mémoire varie d'un appel de fonction sur l'autre. Il est pris automatiquement sur la pile des appels de fonctions, lorsque la fonction est appelée. Il est repris automatiquement lorsque l'appel de fonction est terminé. Ces variables sont en fait locales à des appels de fonctions plutôt qu'à des fonctions.

Les variables locales à un module sont des variables déclarées dans un module (c'est-à-dire dans un fichier source) mais précédées du mot-clef `static`. Elles sont invisibles en dehors de leur module. Par exemple, la variable `primes` suivante, est locale au module qui la contient :

```
static int primes [] = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41 };
```

Le mot-clef `static` précède parfois la déclaration d'une variable, à l'intérieur du corps d'une fonction. La variable est alors une variable locale au module (il lui correspond un symbole) mais invisible en dehors de la fonction où elle est déclarée. C'est le cas de la variable `buffer` de la fonction `printf_gnuplot` suivante :

```
void printf_gnuplot (struct gnuplot_window* gnuplot, char* format, ...)
{   static char buffer [1024];

    ...
}
```

Les fonctions

Il y a deux types de fonctions : les fonctions locales à un module (leur définition est précédée du mot-clef `static`) et les fonctions globales. À chaque fonction correspond un symbole. La fonction suivante, par exemple, est locale au module qui la contient :

```
static double max (double a, double b)
{
    return a < b ? b : a;
}
```

5.1.2 Édition des liens entre fichiers objets

Dans l'exemple élémentaire suivant, deux symboles sont définis : le symbole `Tableau` (un identificateur de variable globale) et le symbole `main` (un identificateur de fonction globale) ; par contre, un symbole est indéfini : celui de la fonction `func`, appelée dans `main` :

```
/* fichier a.c */
extern char func (int);

char Tableau [3];

int main ()
{   int i;
    for (i = 0; i < 3; i++)
        Tableau [i] = func (i);
    return 0;
}
```

En compilant `a.c` avec l'option `-c`, on produit un fichier objet `a.o`. L'utilitaire `nm` permet ensuite de visualiser les symboles globaux définis dans le fichier (leur nom est précédé d'une majuscule différente de `U`) et des références vers des symboles indéfinis (leur nom est précédé d'un `U`) :

```
$ gcc -c a.c
$ nm a.o
0000000000000003 C Tableau
                 U func
0000000000000000 T main
```

Considérons maintenant le fichier source `b.c` suivant. Il définit une variable locale au module `b.c` et une fonction globale.

```
/* fichier b.c */
static char constante = '0';

char func (int i)
{
    return (char)(constante + i);
}
```

En compilant `b.c` avec l'option `-c`, on produit un fichier objet `b.o`. L'utilitaire `nm` permet ensuite de visualiser les symboles présents dans le fichier. Le symbole `func` est global : son nom est précédé d'une majuscule différente de `U` ; le symbole `constante` est défini mais n'est pas global, puisque sa définition est précédée du mot-clef `static` : son nom est précédé d'une minuscule.

```
$ gcc -c b.c
$ nm b.o
0000000000000000 d constante
0000000000000000 T func
```

L'édition des liens est maintenant possible. L'exécutable `a.out` contient les deux fichiers objets avec tous leurs symboles (plus d'autres, qu'on ne détaille pas). Tous les symboles sont définis :

```
$ gcc a.o b.o
$ nm a.out
0000000000601030 B Tableau
...
0000000000601018 d constante
...
0000000000400504 T func
00000000004004c4 T main
```

5.1.3 Qu'est-ce qui peut faire échouer l'édition des liens ?

Si on tente l'édition des liens, sans définir le symbole manquant, l'éditeur de liens `ld` (appelé par `gcc`), échoue :

```
$ gcc a.o
a.o: In function ‘main’:
a.c:(.text+0x1b): undefined reference to ‘func’
collect2: ld returned 1 exit status
```

Si un symbole global est défini plus d'une fois, l'édition des liens échoue aussi [1]. C'est ce qui arrive si on mentionne deux fois le même fichier objet, comme dans l'exemple ci-dessous :

```
$ gcc a.o b.o b.o
b.o: In function ‘func’:
b.c:(.text+0x0): multiple definition of ‘func’
b.o:b.c:(.text+0x0): first defined here
collect2: ld returned 1 exit status
```

Enfin, l'édition des liens échoue aussi si le symbole spécial `main` n'est pas défini.

5.1.4 Édition des liens avec une bibliothèque

Une bibliothèque est un ensemble de fichiers objets. Toutefois, les bibliothèques sont traitées différemment des fichiers objets lors des éditions des liens.

Pour commencer, considérons le fichier source `c.c` suivant, qui est similaire à `b.c` :

1. Sauf quand le symbole désigne une variable. Ce traitement particulier à `gcc` peut être évité en passant l'option `-fno-common` au compilateur.

```
/* fichier c.c */

void action (void)
{
}
```

Compilons-le et tentons l'édition des liens entre les trois fichiers `a.o`, `b.o` et `c.o`. On observe que, le fichier `c.o` est incorporé à l'exécutable *bien que l'exécutable ne fasse aucun appel direct ou indirect aux symboles présents dans* `c.o` :

```
$ gcc -c c.c
$ nm c.o
0000000000000000 T action
$ gcc a.o b.o c.o
$ nm a.out
0000000000601038 B Tableau
...
000000000040051c T action
...
0000000000601018 d constante
...
0000000000400504 T func
00000000004004c4 T main
```

Fabriquons maintenant une bibliothèque avec les deux fichiers `b.o` et `c.o` (commande `ar` pour « `archive` » en Anglais). En utilisant la commande `nm`, on voit que la bibliothèque `libdemo.a` contient les deux fichiers objets :

```
$ ar cru libdemo.a b.o c.o
$ nm libdemo.a

b.o:
0000000000000000 d constante
0000000000000000 T func

c.o:
0000000000000000 T action
```

Procédons maintenant à l'édition des liens entre le fichier objet `a.o` qui contient le programme principal et la bibliothèque [2]. On observe que l'éditeur des liens n'a incorporé à l'exécutable, que le contenu du fichier `b.o` (le symbole `action` n'apparaît pas). *Le fichier* `c.o` *n'a pas été incorporé parce qu'aucun de ses symboles n'est utilisé, directement ou indirectement, par le programme.*

2. Au lieu de : `gcc a.o libdemo.a`, on aurait pu écrire, plus traditionnellement : `gcc -L. a.o -ldemo` (l'option `-L.` indique à l'éditeur de liens qu'une des bibliothèques qu'il cherche appartient au répertoire courant).

```
$ gcc a.o libdemo.a
$ nm a.out
0000000000601030 B Tableau
...
0000000000601018 d constante
...
0000000000400504 T func
00000000004004c4 T main
```

5.1.5 La bibliothèque standard

Les fonctions de la bibliothèque standard du langage C (comme `printf`, `scanf`, `exit`, `strlen` ...) sont définies dans la bibliothèque `/usr/lib/libc.a`. Cette bibliothèque fait partie des bibliothèques ajoutées par défaut à la ligne de commande de `gcc`. Par contre, la bibliothèque mathématique `/usr/lib/libm.a` doit être explicitement mentionnée si on utilise une de ses fonctions (il suffit alors de passer le paramètre `-lm` à l'éditeur de liens).

5.1.6 Un piège

L'ordre dans lequel les bibliothèques sont listées est important. En effet, l'éditeur des liens incorpore les fichiers donnés sur sa ligne de commande, en parcourant cette ligne de gauche à droite. Lorsqu'il traite une bibliothèque, il n'incorpore un fichier objet de cette bibliothèque que si ce fichier définit un symbole indéfini, *au moment où la bibliothèque est traitée*. Dans notre exemple, si on place la bibliothèque en début de la ligne de commande, aucun symbole n'est encore défini au moment où la bibliothèque est traitée, et l'édition des liens échoue :

```
$ gcc libdemo.a a.o
a.o: In function ‘main’:
a.c:(.text+0x1b): undefined reference to ‘func’
collect2: ld returned 1 exit status
```

5.1.7 Cas de l'édition des liens dynamique

On a décrit jusqu'ici le mécanisme d'édition des liens *statique*, où l'exécutable contient tous les fichiers objets nécessaires. De tels exécutables peuvent ensuite être transportés d'une machine à une autre et fonctionner sans problème mais peuvent devenir vite très volumineux. Le compilateur `gcc` effectue par défaut un autre type d'édition des liens : une édition des liens *dynamique*. Dans ce cas, c'est l'emplacement des bibliothèques qui est enregistré dans l'exécutable ; l'incorporation du contenu des fichiers objets présents dans les bibliothèques est retardée jusqu'au *lancement de l'exécutable*. Dans le cas de l'édition des liens dynamique, les règles concernant l'ordre d'apparition des bibliothèques peuvent être partiellement relâchées. Voici un exemple utilisant l'édition des liens dynamiques. On utilise `gcc -shared` plutôt

que `ar` pour créer la bibliothèque dynamique. L'édition des liens réussit, même en plaçant la bibliothèque en début de la ligne de commande.

```
$ gcc -shared -o libdemo.so b.o c.o
$ gcc a.o ./libdemo.so
$ ./a.out
$ gcc ./libdemo.so a.o
$ ./a.out
```

5.2 Le projet LINKER

L'algorithme du projet LINKER gère une table des symboles T. Les symboles enregistrés dans cette table peuvent être soit indéfinis, soit définis. On ne tient pas compte des symboles locaux.

Lorsqu'un symbole s est enregistré dans T, plusieurs cas peuvent se produire : si s n'est pas déjà présent dans T, on l'enregistre avec son état (défini ou indéfini) ; si s déjà présent dans T dans l'état indéfini alors, soit le nouveau symbole est indéfini (dans ce cas, rien ne change), soit le nouveau symbole est défini et s passe alors dans l'état défini ; si s déjà présent dans T dans l'état défini alors, soit le nouveau symbole est indéfini (dans ce cas, rien ne change), soit le nouveau symbole est défini et on enregistre le fait qu'il est défini plusieurs fois.

L'algorithme considère successivement tous les fichiers passés sur sa ligne de commande. Si le fichier courant est un fichier objet, tous ses symboles sont enregistrés dans T. Si le fichier courant est une bibliothèque, on cherche si elle contient un symbole s, appartenant à un fichier objet *obj*, qui fournirait une définition pour un des symboles indéfinis présents dans T. Si c'est le cas, tous les symboles du fichier objet *obj* sont enregistrés dans T. À chaque fois que les symboles d'un des fichiers objets de la bibliothèque sont incorporés à la table, on recommence le parcours de la bibliothèque.

À la fin de l'algorithme, l'édition des liens est considérée comme réussie si T contient le symbole `main`, ne comporte aucun symbole indéfini et aucun symbole défini plusieurs fois.

5.2.1 Implantation

Symboles

Un symbole est représenté par la structure suivante. Le champ `ident` contient l'identificateur du symbole. Le champ `type` contient le type, tel qu'il est fourni par la commande `nm`. Le champ `nbdef` contient le nombre de fois où le symbole est défini. Ce champ n'acquiert une signification que lorsque le symbole est enregistré dans la table des symboles.

```
struct symbole
{   char type;
    char* ident;
```

```
    int nbdef;
};
```

La table des symboles

Elle est implantée sous la forme d'une structure `struct symtable` dont l'implantation peut varier (voir chapitre 4). Dans sa version la plus simple, la structure est un tableau redimensionnable `T` (voir ci-dessous). Le champ `mesures` contient une structure, inutile au fonctionnement de l'algorithme, qui sert à compter les comparaisons de chaînes de caractères, pour mesurer l'efficacité de la structure de données (voir chapitre 4).

```
struct symtable
{   struct tableau T;     /* tableau redimensionnable de symboles */
    struct stats mesures; /* mesures */
};
```

Le module `symtable` exporte les cinq fonctions globales ci-dessous. La première est un constructeur. La deuxième est le destructeur. La troisième enregistre le symbole `sym` dans la table `table`. La quatrième teste si `table` contient un symbole d'identificateur `clef`. Si c'est le cas, elle retourne l'adresse du symbole, sinon, elle retourne le pointeur nul. La cinquième parcourt la table, teste si l'édition des liens est réussie ou non et imprime une synthèse sur la sortie standard. Elle retourne zéro en cas de succès.

```
void init_symtable (struct symtable*)
void clear_symtable (struct symtable*)
void enregistrer_dans_symtable (struct symtable* table, struct symbole* sym)
struct symbole* rechercher_dans_symtable (char* clef, struct symtable* table)
int synthese_symtable (struct symtable*)
```

Voici le code de la fonction qui enregistre un symbole (supposé global) dans la table. Les instructions qui mettent à jour le champ `mesures` ont été supprimées.

```
void enregistrer_dans_symtable (struct symtable* table, struct symbole* sym)
{   struct symbole* symp;
/*
 * Si table->T contient un symbole d'identificateur sym->ident,
 * l'adresse de ce symbole est affectée à symp, sinon, symp reçoit zéro.
 */
    rechercher_symbole_dans_tableau (&symp, sym->ident, &table->T);
    if (symp)
    {   if (est_defini_symbole (sym))
        {   if (est_indefini_symbole (symp))
                changer_type_symbole (symp, sym->type);
            ajouter_definition_symbole (symp);
        }
```

```
    } else
    {
/* Le nb de définitions vaut 1 ou 0 suivant que le symbole est défini ou pas */
        if (est_defini_symbole (sym))
            changer_nbdef_symbole (sym, 1);
        else
            changer_nbdef_symbole (sym, 0);
        ajouter_symbole_dans_tableau (&table->T, sym);
    }
}
```

Voici le code de la fonction qui recherche un symbole à partir de sa clef, dans la table. Les instructions qui mettent à jour le champ `mesures` ont été supprimées.

```
struct symbole* rechercher_dans_symtable (char* clef, struct symtable* table)
{   struct symbole* symp;
    rechercher_symbole_dans_tableau (&symp, clef, &table->T);
    return symp;
}
```

L'itérateur de symboles

Un itérateur de symboles est implanté dans dans le module `iterateur_symbole`. Il permet d'extraire les symboles des fichiers objets et des bibliothèques. Ce module exporte les cinq fonctions suivantes. La première est un constructeur. Elle positionne l'itérateur `iter` au début du fichier `fname` (qui doit être soit un fichier objet, soit une bibliothèque) et retourne l'adresse de son premier symbole (le pointeur nul si aucun symbole). La deuxième est un autre constructeur. Elle s'applique dans le cas où `iter1` est un itérateur, en train de parcourir les symboles d'une bibliothèque. Elle positionne `iter2` au début du fichier objet courant de `iter1` et retourne l'adresse de son premier symbole (le pointeur nul si aucun symbole). Après appel à cette fonction, les deux itérateurs `iter1` et `iter2` sont indépendants l'un de l'autre : il est possible d'appliquer le destructeur sur l'un sans affecter le fonctionnement de l'autre. La troisième fonction est le destructeur. La quatrième fonction fait avancer l'itérateur et retourne l'adresse du symbole suivant (le pointeur nul si aucun symbole). La cinquième fonction s'applique dans le cas où l'itérateur est en train de parcourir une bibliothèque. Elle fait avancer l'itérateur jusqu'au fichier objet suivant de la bibliothèque et retourne l'adresse de son premier symbole (le pointeur nul si aucun symbole). Les pointeurs retournés par les fonctions pointent sur des champs internes des itérateurs [3]. Les ressources consommées par ces symboles seront automatiquement libérées par l'appel à `clear_iterateur_symbole`.

```
struct symbole* first_symbole (struct iterateur_symbole* iter, char* fname)
struct symbole* first_symbole_objet_courant
```

3. Un choix de conception discutable : il simplifie l'écriture des programmes mais complique les spécifications et la documentation.

```
                                (struct iterateur_symbole* iter2,
                                 struct iterateur_symbole* iter1)
void clear_iterateur_symbole (struct iterateur_symbole*)
struct symbole* next_symbole (struct iterateur_symbole*)
struct symbole* next_symbole_next_objet (struct iterateur_symbole*)
```

Le code source du programme principal

```
int main (int argc, char** argv)
{   struct symtable table;
    struct iterateur_symbole iter, iter2;
    struct symbole *sym, *sym2;
    int i, status;
    bool reloop;

    init_symtable (&table);
    i = 1;
    while (i < argc)
    {   commenter_stats (&table.mesures, argv [i]);
        reloop = false;
        sym = first_symbole (&iter, argv [i]);
        if (est_fichier_objet (argv [i]))
        {
/* Charge tous les symboles de l'objet */
            while (sym != (struct symbole*)0)
            {   if (! est_local_symbole (sym))
                    enregistrer_dans_symtable (&table, sym);
                sym = next_symbole (&iter);
            }
        } else
        {
/*
 * On parcourt toute la bibliothèque à la recherche d'une définition
 * pour un symbole indéfini.
 */
            while (sym != (struct symbole*)0)
            {   if (est_global_et_defini_symbole (sym))
                {   sym2 = rechercher_dans_symtable (sym->ident, &table);
                    if (sym2 != (struct symbole*)0 &&
                                                est_indefini_symbole (sym2))
                    {
/* On en a trouvé un : on charge l'objet auquel ce symbole appartient */
```

```
                        sym2 = first_symbole_objet_courant (&iter2, &iter);
                        while (sym2 != (struct symbole*)0)
                        {   if (! est_local_symbole (sym2))
                                enregistrer_dans_symtable (&table, sym2);
                            sym2 = next_symbole (&iter2);
                        }
                        clear_iterateur_symbole (&iter2);
/* On devra parcourir à nouveau la bibliothèque */
                        reloop = true;
/*
 * On saute tous les symboles qui suivent sym dans le fichier objet de sym
 * puisqu'on vient juste de les charger dans la table.
 */
                            sym = next_symbole_next_objet (&iter);
                    } else
                            sym = next_symbole (&iter);
                } else
                    sym = next_symbole (&iter);
            }
        }
        clear_iterateur_symbole (&iter);
        if (!reloop)
            i += 1;
    }
/* Indique si l'édition des liens est réussie ou pas */
    status = synthese_symtable (&table);
    clear_symtable (&table);

    return status;
}
```

5.2.2 Exemples

Sur l'exemple donné en début de chapitre, avec l'implantation naïve de la table des symboles, on a les résultats suivants. Le fichier `linker.stats` contient trois colonnes de nombres : la première colonne compte le nombre d'appels à `get_symbole_symtable` et `put_symbole_symtable`, la deuxième compte le nombre de comparaisons de chaînes de caractères (appels à `strcmp`) effectuées par des deux fonctions, la troisième compte le nombre de symboles présents dans la table des symboles de l'exécutable en cours de construction.

```
$ ./linker a.o libdemo.a
edition des liens reussie
$ ./linker libdemo.a a.o
```

```
symboles indefinis
        func
$ ./linker a.o libdemo.a a.o
symboles dupliques
        Tableau
        main
$ cat linker.stats
# nb appels get | nb comp. chaines | nb symboles
...
       12                24               3
```

Avec le code du projet LINKER,

```
$ ./linker *.o /usr/lib/libc.a
edition des liens reussie
$ cat linker.stats
# nb appels get | nb comp. chaines | nb symboles
...
      6185              648141          112
```

5.3 Le makefile

On en donne ici une présentation très simplifiée, qui suffit pour les séances de travaux pratiques de ce cours. Un *Makefile* est un fichier qui contient deux types d'informations :

1. des commandes de compilation séparée et d'édition des liens, qui peuvent être longues, en raison des options,
2. des dépendances entre fichiers : si on modifie un fichier `a.h`, quels sont les fichiers à recompiler ? Probablement `a.c` mais peut-être, aussi d'autres fichiers.

Avec l'utilitaire `make` de GNU, la conception d'un *makefile* est vraiment très simple : les commandes de compilation n'ont pas besoin d'être écrites (elles sont implicites) ; les dépendances peuvent être fabriquées par `gcc` lui-même !

5.3.1 Un exemple de makefile

Prenons l'exemple d'un TP, comportant cinq fichiers :

```
$ ls
chaine.c  chaine.h  liste_char.c  liste_char.h  main.c
```

Le programme principal `main.c` inclut `chaine.h`. Le fichier d'entête `chaine.h` inclut `liste_char.h`. L'ensemble des dépendances s'obtient par la commande suivante. La première ligne signifie : toute modification à `chaine.c`, `chaine.h` ou `liste_char.h`, doit entraîner la reconstruction du fichier objet `chaine.o`.

```
$ gcc -MM *.c
chaine.o: chaine.c chaine.h liste_char.h
liste_char.o: liste_char.c liste_char.h
main.o: main.c chaine.h liste_char.h
```

On obtient un *makefile* en reportant ces trois lignes à la fin du fichier `Makefile` suivant :

```
CC=gcc
CFLAGS=-g -Wall -Wmissing-prototypes
LDFLAGS=-g
objects := $(patsubst %.c,%.o,$(wildcard *.c))
all: main
clean:
        -rm $(objects)
        -rm main
main: $(objects)
chaine.o: chaine.c chaine.h liste_char.h
liste_char.o: liste_char.c liste_char.h
main.o: main.c chaine.h liste_char.h
```

5.3.2 Utilisation

Maintenant, si on lance la commande `make`, toutes les commandes nécessaires à la production de l'exécutable `main` sont exécutées :

```
$ make
gcc -g -Wall -Wmissing-prototypes   -c -o main.o main.c
gcc -g -Wall -Wmissing-prototypes   -c -o chaine.o chaine.c
gcc -g -Wall -Wmissing-prototypes   -c -o liste_char.o liste_char.c
gcc -g  main.o chaine.o liste_char.o   -o main
```

Si on lance à nouveau la commande, rien ne se passe :

```
$ make
make: Nothing to be done for ‘all’.
```

Par contre, si on modifie `chaine.h`, certaines compilations (pas toutes) sont exécutés à nouveau :

```
$ touch chaine.h
$ make
gcc -g -Wall -Wmissing-prototypes   -c -o main.o main.c
gcc -g -Wall -Wmissing-prototypes   -c -o chaine.o chaine.c
gcc -g  main.o chaine.o liste_char.o   -o main
```

Enfin, si on veut effacer tous les fichiers objets ainsi que l'exécutable, il suffit de lancer la commande :

```
$ make clean
rm chaine.o liste_char.o main.o
rm main
```

5.3.3 Explications

Les trois premières lignes de `Makefile` donnent des valeurs à des variables reconnues par l'utilitaire `make` et utilisées pour compiler des programmes C. La variable `CC` contient le nom du compilateur. La variable `CFLAGS` contient les options à passer lors des étapes de compilation séparées. La variable `LDFLAGS` contient les options à passer lors de l'édition des liens.

La ligne suivante est un peu compliquée. Elle affecte à la variable `objects`, les noms des fichiers objets du TP. Ces noms sont obtenus en substituant le suffixe « .o » au suffixe « .c », pour tous les fichiers « .c » présents dans le répertoire courant.

La ligne suivante donne la cible par défaut : si on lance `make` sans paramètre, il faut construire `main`.

La ligne suivante donne la cible `clean` : si on lance `make clean`, il faut effacer les fichiers présents dans la variable `objects` ainsi que `main`. Le signe moins devant `rm` indique à `make`

de ne pas s'arrêter si ces commandes échouent (par exemple, dans le cas où les fichiers à détruire n'existent pas). Attention : les espaces devant les actions sont impérativement des tabulations.

La ligne suivante indique une première dépendance : la cible `main` dépend de tous les fichiers objets. Si un fichier objet est modifié ou reconstruit, alors, il faut reconstruire `main`. Cette ligne n'est suivie d'aucune action : il s'agit d'une action implicite.

Les dernières lignes ont été produites par `gcc -MM`. Elles ne sont suvies d'aucune action. Les actions associées sont implicites.

5.4 Le debugger

Le *debugger* `gdb` permet d'exécuter des programmes C pas à pas. Il facilite considérablement la mise au point des programmes. Pour pouvoir l'appliquer à un exécutable, il faut que l'option `-g` soit passée à `gcc`, à l'étape de compilation séparée ainsi qu'à l'étape d'édition des liens.

5.4.1 Principales commandes

Les commandes les plus utiles sont les suivantes. Il y en a d'autres. Même celles-là ont des options supplémentaires très utiles aussi.

```
(gdb) run
(gdb) run < nom-de-fichier
```

Lance l'exécution du programme jusqu'au prochain point d'arrêt.

```
(gdb) start
(gdb) start < nom-de-fichier
```

Commence l'exécution du programme et s'arrête devant la première ligne.

```
(gdb) next
```

Exécute la ligne courante et passe à la suivante

```
(gdb) step
```

Variante de `next` : si la ligne courante est un appel de fonction, le debugger s'arrête à la première ligne de cette fonction.

```
(gdb) break numéro-de-ligne
(gdb) break nom-de-fonction
```

Pose un point d'arrêt.

```
(gdb) break numéro-de-ligne if cond
(gdb) break nom-de-fonction if cond
```

Pose un point d'arrêt, qui n'arrête l'exécution du programme que, si la condition (écrite en C) est réalisée.

```
(gdb) cont
```

Continue l'exécution jusqu'au prochain point d'arrêt.

```
(gdb) list
(gdb) list numéro-de-ligne
(gdb) list nom-de-fonction
```

Liste une dizaine de lignes du programme.

```
(gdb) print nom-de-variable
```

Affiche le contenu de la variable.

```
(gdb) where
```

Imprime la pile des appels de fonctions.

```
(gdb) up
(gdb) down
```

Permet de monter et de descendre dans la pile des appels de fonctions, pour consulter, par exemple, le contenu des variables locales.

```
(gdb) call appel-de-fonction
(gdb) print appel-de-fonction
```

Permet d'appeler une fonction du programme, avec des paramètres, éventuellement. Très utile dans le cas d'une structure de données compliquée, munie d'une fonction d'affichage.

5.4.2 Exemple

Le programme C en section 5.4.2 implante un algorithme de tirage de boules de loto. La fonction principale utilise deux ensembles de boules, un pour les boules qui restent à tirer et un autre pour les boules déjà tirées. La structure de données utilisée, `struct ensemble`, est constituée de deux champs : le tableau `elt` contient les boules (des entiers, en fait) ; le champ `size` donne le nombre de boules réellement présentes dans le tableau. La structure `struct ensemble` est compréhensible isolément. Cela a permis d'écrire une fonction `print_ensemble` à un seul argument. Par contre, le tableau `elt` tout seul ne l'est pas. La fonction `tirage_loto` effectue des calculs un peu redondants. Si on considère par exemple la boucle qui initialise l'ensemble `boules`, on voit qu'on aurait pu écrire les choses beaucoup plus concisément, en évitant l'appel à `init_ensemble`, et en ne fixant la valeur du champ `size` qu'à la fin de la boucle `for`. Le programme aurait été non seulement plus court mais aussi plus rapide. Pourquoi a-t-on préféré la « version longue » ? Parce qu'elle s'efforce de maintenir la structure dans un état cohérent, ce qui permettrait, si on le souhaitait, d'effectuer des appels à `print_ensemble` au début de chaque itération. Des raisonnements similaires ont guidé l'écriture de la deuxième boucle. Cette possibilité est particulièrement intéressante avec le debugger, puisqu'il est possible d'appeler une fonction comme `print_ensemble` depuis n'importe quel point d'arrêt.

Démonstration

```
# L'option -g est nécessaire pour utiliser le debugger
$ gcc -g tirage_loto.c

$ gdb a.out
[...]
Reading symbols from /home/boulier/ENSEIGN/debugger/a.out...done.

# On pose un point d'arrêt dans la fonction erreur
(gdb) break erreur
Breakpoint 1 at 0x400826: file tirage_loto.c, line 43.

# Première exécution : ok
(gdb) run
Starting program: /home/boulier/ENSEIGN/debugger/a.out
{5, 12, 7, 2, 19, 10, 20}

Program exited normally.

# Deuxième exécution : l'erreur apparaît
(gdb) run
Starting program: /home/boulier/ENSEIGN/debugger/a.out
```

```
Breakpoint 1, erreur () at tirage_loto.c:43
42          fprintf (stderr, "une erreur s'est produite\n");

# On remonte à la ligne d'où vient l'appel à erreur
(gdb) up
#1  0x0000000000400918 in tirage_loto () at tirage_loto.c:68
63                  erreur ();
(gdb) list
60          for (i = 0; i < n; i++)
61          {   k = (int)(drand48 () * (boules.size + 1));
62              if (appartient (boules.elt [k], &tirage))
63                  erreur ();
64              tirage.elt [tirage.size] = boules.elt [k];
65              tirage.size += 1;
66      /* ici, la structure "tirage" est cohérente */
(gdb) print k
$1 = 6
(gdb) print boules.elt [k]
$2 = 20
(gdb) call print_ensemble (&tirage)
{18, 20, 7, 13}
```

En plaçant un point d'arrêt dans `erreur`, on a trouvé de quelle ligne venait l'erreur. En consultant interactivement les structures `boules` et `tirage`, on voit qu'on tente d'insérer dans `tirage` la boule numéro 20, qui s'y trouve déjà. On remarque l'intérêt de disposer d'une fonction d'affichage pour la structure `struct ensemble` et de garder, en permanence, cette structure dans un état cohérent.

```
(gdb) list 56
50          init_ensemble (&boules);
51          init_ensemble (&tirage);
52      /* À chaque itération, la structure "boules" est cohérente */
53          for (i = 0; i < SIZE_MAX; i++)
54          {   boules.elt [boules.size] = i+1;
55              boules.size += 1;
56          }
57 /* n = le nombre de boules à tirer */
# Pour une raison qq on souhaite s'arrêter dans la boucle 1 quand i vaut 7
(gdb) break 54 if i == 7
Breakpoint 1 at 0x400883: file tirage_loto.c, line 54.
(gdb) run
Starting program: /home/boulier/ENSEIGN/debugger/a.out
```

```
Breakpoint 1, tirage_loto () at tirage_loto.c:54
54          {   boules.elt [boules.size] = i+1;
(gdb) call print_ensemble (&boules)
{1, 2, 3, 4, 5, 6, 7}
```

Le programme

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

#define SIZE_MAX 20
struct ensemble
{   int size;
    int elt [SIZE_MAX];
};

void init_ensemble (struct ensemble* E)
{
    E->size = 0;
}

void print_ensemble (struct ensemble* E)
{   int i;

    putchar ('{');
    if (E->size > 0)
    {   printf ("%d", E->elt [0]);
        for (i = 1; i < E->size; i++)
            printf (", %d", E->elt [i]);
    }
    putchar ('}');
    putchar ('\n');
}

bool appartient (int b, struct ensemble* E)
{   bool found;
    int i;

    found = false;
    for (i = 0; i < E->size && !found; i++)
```

```
        found = E->elt [i] == b;
    return found;
}

void erreur (void)
{
    fprintf (stderr, "une erreur s'est produite\n");
    exit (1);
}

void tirage_loto (void)
{   struct ensemble boules, tirage;
    int i, n, k;

    init_ensemble (&boules);
    init_ensemble (&tirage);
/* À chaque itération, la structure "boules" est cohérente */
    for (i = 0; i < SIZE_MAX; i++)
    {   boules.elt [boules.size] = i+1;
        boules.size += 1;
    }
/* n = le nombre de boules à tirer */
    srand48 ((long)time (NULL));
    n = (int)(drand48 () * (SIZE_MAX + 1));
    for (i = 0; i < n; i++)
    {   k = (int)(drand48 () * (boules.size + 1)); /* <- bug ! */
        if (appartient (boules.elt [k], &tirage))
            erreur ();
        tirage.elt [tirage.size] = boules.elt [k];
        tirage.size += 1;
/* ici, la structure "tirage" est cohérente */
        boules.elt [k] = boules.elt [boules.size - 1];
        boules.size -= 1;
/* ici, la structure "boules" est cohérente */
        if (boules.size < 0 || tirage.size > SIZE_MAX)
            erreur ();
    }
    print_ensemble (&tirage);
}

int main ()
{
```

```
    tirage_loto ();
    return 0;
}
```

Table des figures

Index