Université Paris 13
Institut Galilée
Deug Mias 1ère année
2002-2003

Programmation Impérative

Quatrième Partie

Enseignante : A. Nazarenko

Table des matières

9 FICHIERS 2

9.1 INTRODUCTION 2
9.2 MANIPULER UN FICHIER DANS UN PROGRAMME C 2
9.2.1 DEFINITION D'UNE VARIABLE POINTEUR SUR UN FICHIER 3
9.2.2 OUVRIR ET FERMER UN FICHIER 3
9.2.3 LECTURE ET ECRITURE DANS UN FICHIER EN C 4

10 LES TYPES ABSTRAITS DE DONNEES LINEAIRES 5

10.1 LE TAD LISTE 5
10.1.1 DEFINITION 5
10.1.2 REPRESENTATION CONTIGUË DES LISTES 7
10.1.3 REPRESENTATION CHAINEE A L'AIDE DES CURSEURS 9
10.1.4 REPRESENTATION CHAINEE A L'AIDE DES POINTEURS 12
10.2 LES PILES 13
10.2.1 DEFINITION ET UTILISATION DES PILES 13
10.2.2 LE TAD PILE 15
10.2.3 REPRESENTATION CONTIGUË DES PILES 16
10.2.4 REPRESENTATION CHAINEE A L'AIDE DE POINTEURS 18
10.3 LES FILES 19
10.3.1 DEFINITION ET UTILISATION DES FILES 19
10.3.2 LE TAD FILE 19
10.3.3 REPRESENTATION CONTIGUË DES FILES 20
10.4 REPRESENTATION CHAINEE EN C 22
10.4.1 CREATION D'UNE LISTE CHAINEE POUR REPRESENTER UNE LISTE EN MACHINE ET ALLOCATION DYNAMIQUE DE MEMOIRE 22
10.4.2 GENERALITES SUR L'ALLOCATION DYNAMIQUE DE MEMOIRE 23

11 LA RECURSIVITE 26

11.1 LE PRINCIPE 26
11.1.1 PRELIMINAIRE : LES ARBRES D'EXECUTION 26
11.1.2 LES FONCTIONS RECURSIVES : DEFINITION 27
11.1.3 FONCTIONNEMENT DE LA RECURSIVITE 28
11.2 LES DIFFERENTS TYPES DE RECURSION 33
11.2.1 EXEMPLES DE FONCTIONS RECURSIVES SIMPLES TERMINALES 33
11.2.2 LES RECURSIONS PLUS COMPLEXES 34

12 QUELQUES ALGORITHMES ELEMENTAIRES DE TRI ET DE RECHERCHE 36

12.1 ALGORITHMES ELEMENTAIRES DE TRI 36
12.1.1 TRI PAR BULLES 36
12.1.2 TRI PAR INSERTION 38
12.1.3 TRI PAR SELECTION-ECHANGE 39
12.2 ALGORITHMES ELEMENTAIRES DE RECHERCHE 41
12.2.1 LA RECHERCHE SEQUENTIELLE 41
12.2.2 LA RECHERCHE DICHOTOMIQUE 42

9 Fichiers

La gestion des entrées et des sorties consiste à gérer la communication entre un programme et son environnement (l'utilisateur et/ou d'autres programmes) en effectuant des opérations de lecture (entrées) et d'écriture (sorties) de données. Cette gestion des entrées/sorties joue un rôle important dans la programmation. Un programme sans donnée d'entrée effectue systématiquement le même calcul à chaque exécution. Rappelons par ailleurs qu'il n'est pas raisonnable de modifier les valeurs des constantes ou les valeurs initiales de certaines variables dans le corps du programme avant son exécution : ce n'est pas le rôle des constantes ou des variables que de représenter les données d'entrée d'un programme. À l'inverse, un programme sans sortie effectue des calculs en pure perte.

Jusqu'à présent, nous n'avons effectué des opérations d'entrée (lecture de données) et de sortie (écriture de données) que sur les périphériques standard que sont le clavier (saisie de données en entrée) et l'écran (affichage de résultats en sortie). Une telle gestion interactive des entrées et sorties est adaptée quand celles-ci sont réduites (peu de données à saisir par l'utilisateur, peu de messages à afficher à l'écran). Elle suppose des messages clairs d'invite et de présentation des résultats. Elle suppose également que les erreurs (erreurs de saisies notamment) soient détectées au plus tôt et puissent être corrigées sans mettre en cause l'exécution du programme.

La gestion interactive des entrées/sorties est en revanche inadaptée quand les données en entrée et/ou sortie sont volumineuses ou quand elles doivent servir à d'autres programmes et ne sont pas directement interprétables par l'utilisateur ou encore quand il faut les conserver. On utilise dans ce cas des fichiers qui stockent les données et les préservent au-delà de l'exécution du programme.

9.1 Introduction

Un fichier est une suite ordonnée d'octets. La taille du fichier n'est pas fixée : elle varie lorsqu'on ajoute ou supprime des données. Le fichier comporte donc une marque caractéristique servant à repérer la fin du fichier : il s'agit de la marque `eof` (*end of file*), dite marque de fin de fichier.

Un fichier sert à stocker des données. Il peut en contenir un nombre important. En général, un fichier est placé sur disque dur ou disquette. Mais la notion de fichier est en fait plus large. N'importe quel périphérique (port de communication, imprimante, clavier…) peut-être considéré comme un fichier.

Les opérations que l'on effectue sur des fichiers sont des opérations de lecture et d'écriture. Il existe les procédures fprintf et fscanf pour gérer les entrées/sorties sur fichiers (elles fonctionnent comme printf et scanf).

9.2 Manipuler un fichier dans un programme C

Dans un programme C, on ne manipule pas les fichiers directement par leur nom, mais par une variable déclarée, de type pointeur sur un fichier. En fait, cette variable pointeur pointe sur une zone mémoire particulière, appelée mémoire tampon (*buffer*, en anglais), qui sert de support de stockage intermédiaire entre la mémoire centrale et l'espace disque sur lequel est stocké le fichier. Quand des données stockées dans un fichier sont lues (fichier->programme), elles sont en un premier temps copiées dans la mémoire tampon pour ensuite être stockées

dans des variables du programme. A l'inverse, quand des données sont « affichées » ou écrites dans un fichier (programme->fichier), elles sont en un premier temps placées dans le tampon pour ensuite être copiées du tampon vers le fichier.

Effectuer des lectures et des écritures sur un fichier suppose de respecter un certain protocole. Il faut, dans l'ordre :

Partie définition	1. définir une variable de type pointeur sur un fichier,
Corps du programme	2. ouvrir un fichier en faisant pointer la variable pointeur sur ce fichier, 3. écrire ou lire dans ce fichier via la variable pointeur, 4. fermer le fichier en « libérant » la variable pointeur.

Nous revenons ci-dessous sur ces différentes étapes et sur les procédures correspondantes.

9.2.1 Définition d'une variable pointeur sur un fichier

L'utilisation d'un fichier se fait par l'intermédiaire de variable pointeur sur un objet de type FILE. Pour définir une telle variable, on suivra la syntaxe suivante :

```
FILE *<id. var. pointeur>;
```

L'identificateur de cette variable pointeur peut être vu comme le nom interne au programme d'un fichier (le type `FILE` est défini dans `stdio.h`).

9.2.2 Ouvrir et fermer un fichier

Après avoir défini une variable pointeur sur un objet de type FILE, il faut, pour ouvrir un fichier, initialiser cette variable en la faisant pointer sur un fichier. On utilise pour cela la fonction prédéfinie fopen. `fopen` prend comme argument un nom de fichier et retourne un pointeur (sur le début de la mémoire-tampon) qui servira par la suite à lire et écrire sur ce fichier. Pour appeler la fonction fopen, on suivra la syntaxe suivante :

```
<id. var. pointeur sur FILE>=fopen("<nom fichier>","<mode>");
```

avec

<nom fichier> : chaîne de caractères qui représente le nom sous lequel le fichier est connu par exemple par le système d'exploitation (c'est le nom externe du fichier).

<mode> : `r` : ouverture en lecture d'un fichier existant,

`w` : ouverture en écriture d'un fichier (si le fichier n'existe pas, il est crée, sinon, si le fichier existe, les données qu'il contient sont effacées)

`a` : ouverture en ajout (si le fichier n'existe pas, il est crée, sinon, si le fichier existe, les données ajoutées le seront en fin de fichier)

`r+` : ouverture en lecture et écriture d'un fichier existant,

`w+` : création d'un fichier et ouverture en lecture-écriture (si le fichier existe déjà, il est détruit et remplacé par le fichier crée)

`a+` : ouverture en lecture et écriture d'un fichier auquel on souhaite ajouter des données.

Si le fichier n'a pu être ouvert, la fonction `fopen` renvoie une valeur nulle.

Pour fermer un fichier : **`fclose`**

Syntaxe : **`fclose(<`**`id. var. pointeur sur FILE>`**`);`**

Un fichier ouvert dans un programme doit nécessairement être fermé en fin de programme.

9.2.3 Lecture et écriture dans un fichier en C

`putc(<``valeur de type char>, <id. var. pointeur sur FILE>`**`)`** : `putc` écrit la valeur de type char donnée comme premier paramètre effectif dans le fichier

`getc(<``id. var. pointeur sur FILE>`**`)`** : `getc` lit le caractère courant dans le fichier

Exemple

Création d'un fichier de données contenant une chaîne de caractères en majuscule alors que l'entrée est une chaîne de caractères en minuscule saisie au clavier

```
#include <stdio.h>
int main() {
FILE *fpt;
char c;
fpt=fopen("essai.dat","w");
printf("Saisir une chaîne de caractères en minuscules :");
do
  putc(toupper(c=getchar()),fpt);
while(c!='\n');
fclose(fpt);
}
```

`fprintf(<``id. var. pointeur sur FILE, format, para1, para2 ,..., paran>`**`)`**

écrit dans le fichier désigné par une variable pointeur sur FILE les valeurs de `para1`, `para2`, ..., `paran` suivant le format précisé dans *`format`*

`fscanf(<``id. var. pointeur sur FILE, format, para1, para2 ,..., paran>`**`)`**

même chose que pour `fprintf` mais pour la lecture du fichier désigné

Les spécifications de format à utiliser sont les mêmes que pour les fonctions `scanf` et `printf`.

10 Les Types Abstraits de Données Linéaires

La conception d'un algorithme se fait par étapes qui correspondent à différents niveaux de raffinement, du plus grossier au plus précis. Dans un premier temps, la conception de l'algorithme se fait indépendamment de la manière dont il sera implanté. En particulier, la manière dont seront représentées en machine les données n'est pas fixée.

A ce premier stade, on considère les données de manière *abstraite*, sans se préoccuper de la manière dont elles seront représentées en machine. On parle alors de *type abstrait de données* (TAD).

Par exemple, quand nous écrivons des algorithmes avec des variables booléennes, nous ne nous préoccupons pas de la manière dont ces variables seront implantées en C où il n'existe pas de type booléen prédéfini. On a seulement besoin de définir une notation pour les valeurs booléennes (*vrai* et *faux*) et de savoir quelles opérations on peut effectuer sur des variables booléennes (opérateurs dits booléens ou « logiques » : $\neg$, $\wedge$, $\vee$, notamment). Ce faisant, on manipule un type abstrait booléen.

Dans un deuxième temps, on choisit une représentation particulière pour ce type abstrait de donnée (on parle alors de *structure de donnée*). Par exemple le type abstrait booléen peut être représenté par un entier. On peut ainsi obtenir plusieurs versions différentes de l'algorithme selon la structure de données retenue.

Un type abstrait de données se définit par

- sa signature (ou sa syntaxe) : le nom des opérations et le type de leurs arguments et de leur résultat ;
- ses propriétés.

Dans ce chapitre, nous étudions des types abstraits de données très classiques, les *types abstraits linéaires* qui sont utilisés de façon très intensive en programmation. Leur but est de gérer un ensemble fini et totalement ordonné d'éléments de même type. On ne s'intéresse pas au type de ces éléments qui peut être simple (entier ou réel) ou composé (tableau, enregistrement), mais aux opérations que l'on effectue sur cet ensemble (indépendamment du type des éléments). Pour chaque TAD linéaire, nous envisageons différentes structures de données classiquement utilisées pour les représenter en machine.

10.1 Le TAD liste

10.1.1 Définition

Une liste L est une suite finie, éventuellement vide, d'éléments de même type repérés selon leur position dans la suite.

Les éléments d'une liste sont donc complètement ordonnés en fonction de leur position. La longueur d'une liste désigne le nombre d'éléments contenus dans la liste. Si la liste ne contient aucun élément, on dira que sa longueur est de 0 et qu'elle est *vide*.

La liste est un TAD abondamment utilisé pour écrire des algorithmes car il est fréquent de pouvoir organiser des données sous forme de liste.

Exemple

Pour stocker les 500 premiers nombres premiers, on peut les représenter par une liste. On veut pouvoir savoir par exemple quel est le $323^{ème}$ nombre premier, quel est le plus petit nombre premier supérieur à 400 ou si 29 est un nombre premier.

Exemple

Pour représenter un texte en cours d'édition, on veut représenter l'ensemble des lignes qui le composent par une liste. On a en effet besoin d'avoir un accès direct à la ligne i, d'insérer une nouvelle ligne après la ligne j ou au contraire de supprimer les lignes k à k'.

Pour spécifier complètement le TAD liste, il faut définir un ensemble d'opérations admissibles sur ce TAD.

On note :

- E : ensemble d'éléments de même type
- L : ensemble de listes pouvant être définies sur E
- l : liste appartenant à L, $l = <e_1, e_2, ..., e_n>$ avec $\forall i \in [1,..,n], e_i \in E$
- e : élément appartenant à E
- p : position dans une liste (par convention, on considère que les positions sont numérotées à partir de 1).

Classiquement, on distingue trois types d'opérations :

Les constructeurs de listes

- Créer une liste vide

CreerListe : $\emptyset \rightarrow L$

crerListe() : l où l est une liste vide

- Ajouter un élément e à la position p dans la liste l. Cette opération a pour effet de changer la longueur de la liste et également de décaler de 1 les positions des éléments situés après l'élément inséré.

Insérer : $L \times N^* \times E \rightarrow L$

insérer(l,p,e) : $<e_1, e_2, .., e_{p-1}, e, e_p, .., e_n>$

Les opérations de parcours *de listes (elles laissent la liste inchangée)*

- Déterminer la longueur d'une liste l.

Longueur : $L \rightarrow N^*$

longueur(l) : n

- Prédicat qui teste si une liste l est vide.

EstVide : $L \rightarrow \{vrai, faux\}$

estVide(l) : vrai si l ne contient aucun élément et faux sinon.

- Trouver l'élément e d'une liste l par la position p qu'il occupe dans la liste.

  Accéder : $L \times N^* \rightarrow E$

  accéder(l, p) : e si $p \leq$ longueur(l), indéfinie sinon

- Trouver la position p d'un élément e dans une liste l

  Localiser : $L \times E \rightarrow N^*$

  localiser(l, e_p) : p (0 si e_p n'appartient pas à l)

Une opération de modification de listes

- Suppression de l'élément e_p situé à la position p de la liste l. Elle a également pour conséquence de changer la longueur de la liste et de changer les positions (reculées de 1) des éléments positionnés de (p+1) à n.

  Supprimer : $L \times N^* \rightarrow L$

  supprimer(l,p) : $<e_1, e_2, .., e_{p-1}, e_{p+1}, .., e_n>$ si $p \leq$ longueur(l), indéfinie sinon

10.1.2 Représentation contiguë des listes

On peut représenter en machine une liste l en utilisant le type tableau. La $i^{ème}$ variable du tableau contient l'élément situé en $i^{ème}$ position de la liste. Pour cela, il faut, d'une part, que le type du tableau soit cohérent avec le type des éléments de la liste, et d'autre part, que la taille du tableau soit au moins égale à la longueur de la liste.

Exemple

Pour représenter en machine le TAD liste d'entiers, on introduit le type suivant :

Type
ListeEntier : tableau de 100 entiers

En position 0 d'un tableau de type *ListeEntier*, on stocke le nombre d'éléments de la liste associée.

Exemple

Pour représenter la liste des lignes d'un texte, on introduit le type suivant (on suppose le type *Ligne* déjà défini) :

Type
Texte : enregistrement à 2 membres *tabLignes : tableau de 200 Lignes*
nbreLignes : entier

Le champ *nbreLignes* du type enregistrement *Texte* contient le nombre de lignes effectivement stockées répertoriées dans la liste *tabLignes*.

De façon générale, on considère, pour représenter en machine le TAD liste, le type enregistrement suivant :

Type
Liste : enregistrement à 2 membres *tabÉléments : tableau de TAILLEMAX Éléments*
longueur : entier

où *Élément* désigne le type des éléments appartenant à la liste (*Élément* est un type prédéfini ou préalablement déclaré dans le programme).

On peut alors déclarer une variable de type *Liste* comme suit :
Variable de type Liste : l

En utilisant cette structure de données, il est alors possible d'écrire des procédures et fonctions associées aux opérations admissibles sur le TAD liste.

Exemple

```
fonction localiser(Liste l, Élément e):entier;
{Cette fonction retourne la position de l'élément e dans la liste l. Si e n'appartient pas à l, localiser renvoie la valeur 0.}
1 variable auxiliaire entière : i
1 variable auxiliaire booléenne : trouve
début fonction
trouve<-faux
i<-1
tant que non trouve et i≤ l.longueur faire
    si l.tabElement[i]=e alors trouve<-vrai fin si
    i<-i+1
fin tant que
si trouve alors retourne la valeur i-1
sinon retourne 0
fin si
fin fonction

procédure insérer(Liste l, Élément e, entier p)
{On insère un élément en pième position. Si le tableau est déjà plein, l'insertion provoque un débordement et un message d'erreur. }
1 variable auxiliaire entière : i
début
{test de débordement}
si l.longueur>=TAILLEMAX alors écrire(« Tableau plein »)
sinon
    {insertion}
    pour i variant de l.longueur à p faire l.tabElement[i+1]<-l.tabElement[i]
    l.tabElement[p]<-e
    l.longueur<-l.longueur+1
fin sinon
fin procédure
```

La conception des fonctions et procédures associées aux opérations *longueur*, *estVide*, *accéder* et *supprimer* est laissée en exercice.

Le choix de cette structure de données a les avantages suivants :

- bonne utilisation de la mémoire car les éléments sont stockés dans des cases mémoires physiquement contiguës,

- facilité d'accès au $i^{ème}$ élément puisqu'il est stocké dans la ième cellule du tableau,
- facilité pour parcourir de façon séquentielle la liste,
- facilité pour insérer ou supprimer le dernier élément de la liste.

Par contre, représenter une liste avec cette structure de données présente les inconvénients suivants :

- Il faut *a priori* connaître la longueur maximale de la liste car lorsqu'on manipule une variable de type tableau dans un programme, on doit spécifier sa taille lors de sa déclaration.
- La suppression et l'insertion d'un élément à l'intérieur une liste sont coûteuses. En effet, si dans une liste de longueur n, on veut supprimer l'élément se trouvant en position p (p<n), il faut avancer d'une position le contenu des variables allant de *l.tabÉlément[p+1]* à *l.tabÉlément[n]*, soit n-p copies.

10.1.3 Représentation chaînée à l'aide des curseurs

Exemple

Pour représenter une liste contentant 3 éléments <x,y,z>, on gère un tableau unidimensionnel appelé *tabÉlément* d'enregistrements à deux champs, le premier appelé *val* contient les éléments de la liste et, le second appelé *suivant* indique, pour chaque élément, la position dans le tableau de l'élément suivant dans la liste.

	1	2	3	4	5	6	7
Val		Y		x	z		
Suivant		5		2	0		

Pour retenir la position du premier élément de la liste dans le tableau *tabÉlément*, on utilise une variable entière à laquelle on affecte l'indice dans le tableau du premier élément de la liste (4 dans l'exemple ci-dessus). Le champ *suivant* est en fait appelé curseur : il permet de chaîner les éléments de la liste.

Avec cette représentation, le parcours de la liste et donc l'accès à l'élément situé à une position p donnée sont moins immédiats : il faut suivre la chaîne des indices en comptant les éléments. Dans ce cas, en effet, l'ordre des indices du tableau ne correspond plus à la position des éléments dans la liste.

L'insertion d'un élément à l'intérieur de la liste est économique, en revanche Si on veut ajouter t à la position 3 de la liste ci-dessus, il faut :

- placer t dans un bloc vide (à l'indice 1, par exemple),
- modifier le champ suivant de l'élément précédent (dans notre cas, y qui est en $2^{ème}$ position),
- donner l'ancienne valeur du suivant de y à t.

On obtient le schéma suivant :

	1	2	3	4	5	6	7
Val	**t**	y		x	z		
Suivant	**5**	~~5~~ **1**		2	0		

Pour implanter en machine une représentation chaînée des listes à l'aide de curseurs, on introduit les types suivants :

```
Type
Bloc : enregistrement à 2 membres     val : Élément
                                      suivant : entier
ListeChainee : enregistrement à 2 membres   début : entier
                                            liste : tableau de TAILLEMAX cellules de
                                            type Bloc
```

On suppose que les éléments sont indicés à partir de 1 dans le tableau. Le dernier élément de la liste a 0 comme valeur dans le champ *suivant*. Avec cette structure de données, on peut alors réécrire la fonction *localiser* et la procédure *insérer*.

```
fonction localiser(ListeChainee l, Élément e): entier;
{Cette fonction retourne la position de l'élément e dans la liste l. Si e n'appartient
pas à l, localiser renvoie la valeur 0.}
2 variables auxiliaires entières : i et p
1 variable auxiliaire booléenne : trouve
début fonction
i<-l.début
p<-1
trouve<-faux
tant que non trouve et i≠0 faire
{Le dernier élément de la liste est repéré par le fait que le curseur qui lui est associé
vaut 0.}
    si l.liste[i].val=e alors
        trouve<-vrai
    sinon
        p<-p+1
        i<-l.liste[i].suivant
    fin sinon
fin tant que
si trouve alors localiser retourne la valeur p
sinon localiser retourne 0
fin fonction

procédure insérer(ListeChainée l, Élément e, entier p)
{La procédure insérer permet d'ajouter l'élément e en pième position. Si le tableau
est déjà plein, on a un débordement.}
3 variables auxiliaires entières : i, pos et ind
début procédure
{test de débordement}
{On utilise la fonction longueur qui renvoie la longueur de la liste qui lui est passée
en argument.}
si longueur(l)>=TAILLEMAX alors écrire('Tableau plein')
sinon
    {ind reçoit l'indice d'une variable du tableau l.liste non encore " utilisée "}
    ind<-PlaceVide(l)
```

```
        l.liste[ind].val<-e
        si p=1 alors
                {Si on insère e en première position il faut changer la valeur de l.début}
                l.liste[ind].suivant<-l.début
                l.début<-ind
        sinon
                {Sinon on doit se positionner en position p-1}
                i<-l.début
                pos<-1
                tant que pos<p-1 faire
                        i<-l.liste[i].suivant
                        pos<-pos+1
                fin tant que
                {Insertion de e}
                l.liste[ind].suivant<-l.liste[i].suivant
                l.liste[i].suivant<-ind
        fin sinon
    fin sinon
    fin procédure

    fonction PlaceVide(ListeChainee l): entier
    {Cette fonction renvoie l'indice d'une variable du tableau l.liste non encore " utilisée ". Une telle variable est repérée par le fait que son champ suivant à pour valeur -1. On suppose donc que ce champ suivant est initialisé à -1 pour toutes les blocs du tableau (cf. procédure créerListe).}
    1 variable auxiliaire entière : i
    début fonction
    i<-1
    tant que (i<=TAILLEMAX) et (l.liste[i].suivant≠-1) faire i<-i+1 fin tant que
    retourne la valeur i
    fin fonction
```

Le choix de cette structure de données a l'avantage de faciliter la plupart des opérations sur les listes : le parcours séquentiel, l'insertion et la suppression à une place quelconque (sans avoir à décaler les éléments succédant dans la liste l'élément inséré ou supprimé).

Par contre, représenter une liste avec cette structure de données présente les inconvénients suivants :

- L'accès au $k^{ième}$ élément de la liste n'est plus direct, mais il faut parcourir k-1 éléments à partir du 1^{er} élément de la liste pour atteindre l'élément situé en position k (voir le parcours dans la procédure *insérer*).
- Utilisation d'une place mémoire plus importante pour stocker les champs *suivant*.

Il n'existe pas de représentation en soi. Selon le problème à résoudre on aura intérêt à choisir plutôt la représentation contiguë ou plutôt une représentation chaînée à l'aide de curseurs.

Exemple

Pour la liste des 500 premiers nombres premiers, la représentation contiguë est bien adaptée. On connaît la taille de la liste et on construit la liste dans l'ordre croissant, donc

en ajoutant des entiers à la fin de la liste. On n'a pas besoin de supprimer ou d'insérer des entiers à l'intérieur de la liste.

Exemple

Pour représenter un texte en cours d'édition, il faut tenir compte au contraire du coût des insertions et suppression à l'intérieur de la liste. Dans ce cas, une représentation chaînée sera donc mieux adaptée. L'inconvénient, c'est qu'il faut, là encore, fixer a priori le nombre maximal de lignes du tableau.

En général, pour la représentation chaînée, on utilise les **pointeurs**.

10.1.4 Représentation chaînée à l'aide des pointeurs

Le pointeur est l'outil privilégié pour le chaînage puisqu'un pointeur a pour fonction de pointer sur un autre objet, ce que faisaient les curseurs ci-dessus. Entre représentation à l'aide curseurs et représentation à l'aide de pointeurs, le choix n'est pas toujours évident. La manipulation des pointeurs dans ce contexte fait généralement appel aux techniques d'allocation dynamique de mémoire que nous verrons plus loin.

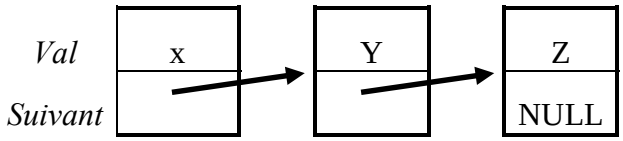
Exemple

Pour représenter la liste contentant les 3 éléments <x,y,z> mentionnée ci-dessus, on gère un ensemble d'enregistrements à deux champs, le premier appelé *val* contient les éléments de la liste et, le second appelé *suivant* est un pointeur qui pointe sur l'enregistrement suivant dans la liste. En pratique pour gérer la liste, on se contente de manipuler un pointeur sur le premier enregistrement.

Pour implanter en machine une représentation chaînée des listes à l'aide de pointeurs, on introduit les types suivants :

```
Type
Bloc: enregistrement à 2 membres :   val : Élément
                                     suivant : pointeur sur Bloc
ListeChaînéeeP : pointeur sur Bloc
```

Soulignons le fait que la structure *Bloc* comporte un champ qui est lui-même de type *Bloc*. On parle dans ce cas de structures *auto-référentielles*. On peut représenter ce chaînage par le schéma suivant :



Avec cette structure de données, on peut alors réécrire la fonction *localiser* et la procédure *insérer* :

```
fonction localiser(ListeChaînéeP l, Élément e): entier
{Cette fonction retourne la position de l'élément e dans la liste l. Si e n'appartient
pas à l, localiser renvoie la valeur 0.}
1 variable auxiliaire entière : p
1 variable auxiliaire de type pointeur sur Bloc: pt
1 variable auxiliaire booléenne : trouve
début fonction
```

```
pt<- l
p<-1
trouve<-faux
tant que non trouve et pt≠ NULL faire
{Le dernier élément de la liste est repéré par le fait que le champ suivant est un pointeur NULL}
    si (*pt).val=e alors
        trouve<-vrai
    sinon
        p<-p+1
        pt<-(*pt).suivant
    fin sinon
fin tant que
si trouve alors localiser retourne la valeur p
sinon localiser retourne 0
fin si
fin fonction
```

Le choix de cette structure de données a les mêmes avantages et inconvénients que la représentation chaînée à l'aide de curseurs pour les opérations sur les listes : le parcours séquentiel, l'insertion et la suppression à une place quelconque (sans avoir à décaler les éléments succédant dans la liste l'élément inséré ou supprimé). Elle présente l'avantage supplémentaire de s'affranchir des contraintes de taille liées à la manipulation des tableaux. La seule contrainte de taille pour les listes est la place mémoire.

10.2 Les piles

10.2.1 Définition et utilisation des piles

Une pile est une liste pour laquelle on ne peut accéder, insérer et supprimer des éléments qu'à une extrémité appelée le *sommet*. En conséquence, ce qui différencie le TAD pile du TAD liste concerne la définition des opérations d'accès, d'insertion et de suppression. On modifie une pile par les deux opérations suivantes :

- empiler : opération qui consiste à ajouter un et un seul élément au sommet de la pile.
- dépiler : opération qui consiste à supprimer un élément positionné au sommet de la pile.

Exemple

> On peut imaginer une pile comme une pile d'assiettes dans un placard. Pour ranger une assiette, on la place sur le somment de la pile existante. Pour prendre un assiette dans le placard, on prend celle qui est situé sur le dessus de la pile. Pour ranger/prendre n assiettes, on les range/prend une par une sur le sommet de la pile.

Il est important de noter que dans une pile, on ne peut accéder qu'à l'objet situé au sommet de la pile. On retire les éléments dans l'ordre inverse de celui dans lequel on les a mis (on parle de l'ordre LIFO *Last in / First out*).

La notion de pile intervient fréquemment en programmation.

Exemple

Elle permet d'implémenter la fonction *Défaire* (*Undo*) d'un traitement de texte (*Annuler frappe* par exemple, dans Word). En effet, la fonction *Défaire* permet d'annuler la dernière action (frappe) effectuée. Si on veut annuler l'avant-dernière action (a1), on n'a pas d'autre choix que d'annuler la dernière action (a2) puis d'annuler la précédente (a1) et de recommencer l'action a2.
Cette fonction *Défaire* gère ainsi une pile des actions exécutées qui sont mémorisées avec leur contexte.

Exemple

Les piles sont également utilisées pour la gestion des environnements d'exécution des procédures et fonctions. Cette utilisation est transparente pour le programmeur. Considérons par exemple l'exécution du programme suivant :

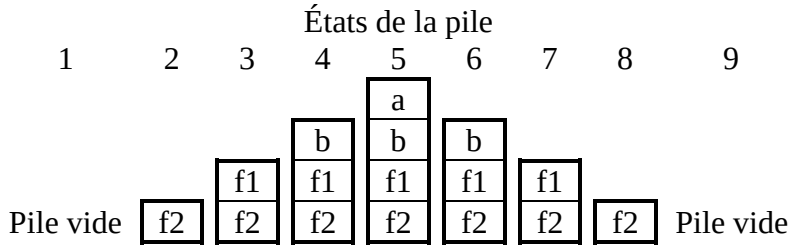
```
procédure a(...);
  début
  ...
  fin procédure a
procédure b(...);
  début
  ...
  a(...);
  ...
  fin procédure b;
fonction f1(...):...;
  début
  ...
  b(...);
  ...
  fin fonction f1
fonction f2(...):...;
  début
  ...
  x← f1(...);
  ...
  fin fonction f2
début fonction principale
resu← f2(...);
fin fonction principale
```

La première fonction appelée est f2 : lors de cet appel, un environnement est crée contenant les valeurs des variables globales. Cet environnement est empilé dans une pile dite d'exécution. Cet environnement de f2 ne sera dépilé que lorsque la fonction f2 aura été exécutée dans sa totalité. Dans les instructions de f2, la fonction f1 est appelée. Un nouvel environnement est créé et empilé...

L'évolution de la pile d'exécution donc la suivante :

Exécution du programme	Actions sur la pile
début	
...	Etat 1 de la pile

appel de f2	L'environnement de f2 est empilé
début de f2	
...	Etat 2 de la pile
appel de f1	L'environnement de f1 est empilé
début de f1	
...	Etat 3 de la pile
appel de b	L'environnement de b est empilé
début de b	
...	Etat 4 de la pile
appel de a	L'environnement de a est empilé
début de a	
...	Etat 5 de la pile
fin de a	L'environnement de a est dépilé
...	Etat 6 de la pile
fin de b	L'environnement de b est dépilé
...	Etat 7 de la pile
fin de f1	L'environnement de f1 est dépilé
...	Etat 8 de la pile
fin de f2	L'environnement de f2 est dépilé
...	Etat 9 de la pile
fin du programme	



10.2.2 Le TAD pile

Pour spécifier complètement le TAD pile, il faut définir un ensemble d'opérations admissibles sur ce TAD.

On note :

- E : ensemble d'éléments de même type
- P : ensemble de piles pouvant être définies sur E
- p : pile appartenant à P, $p = <e_1, e_2, ..., e_n>$ avec $\forall i \in [1,..,n], e_i \in E$
- e : élément appartenant à E

Classiquement, on considère les opérations suivantes :

Les constructeurs de pile

- Créer une pile

  CréerPile : $\varnothing \rightarrow P$

  créerPile() :p où p est une pile vide

- Ajouter un élément e au somment de la pile p. Elle a pour effet de changer la taille de la pile.

  Empiler : $P \times E \rightarrow P$

  empiler(p,e) : $<e_1, e_2, .., e_n, e>$

Les opérations de parcours de pile

- Trouver le sommet de la pile p.

  Sommet : $P \rightarrow E$

  sommet(p) : e_nOpération non définie si la pile p est vide

- Prédicat qui teste si une pile p est vide.

  PileVide : $P \rightarrow$ {vrai, faux}

  pileVide(p) : vrai si l ne contient élément et faux sinon.

Une opération de modification de pile

- Supprimer de l'élément situé au sommet de la pile p. Cette opération a également pour conséquence de changer la taille de la pile.

  Dépiler : $L \rightarrow L$

  dépiler(l) : $<e_1, e_2, .., e_{n-1}>$ Opération non définie si la pile p est vide

10.2.3 Représentation contiguë des piles

On peut représenter en machine une pile p en utilisant un tableau. Le somment est la dernière cellule remplie du tableau. Donc il faut donc toujours savoir quelle est la taille effective du tableau. La pile est vide lorsque la taille effective du tableau est 0.

Exemple

Pour représenter en machine une pile d'entiers, on introduit le type suivant :

Type
PileEntier : tableau de 100 entiers

En position 0 d'un tableau de type *PileEntier*, on stocke la taille de la pile.

Exemple

Pour représenter la pile des actions effectuées dans un traitement de texte, on introduit le type suivant (on suppose le type *Action* déjà défini) :

Type
PileActions : enregistrement à 2 membres tabActions : tableau de 200 Actions

nbreActions : entier

Le champ *nbreActions* du type enregistrement *PileActions* contient le nombre d'actions effectuées.

De façon générale, on considère, pour représenter en machine le TAD pile, le type enregistrement suivant :

Type
Pile : enregistrement à 2 membres *tabÉlément : tableau de TAILLEMAX Éléments*
taille : entier

où *Élément* désigne le type des éléments appartenant à la pile (*Élément* est un type prédéfini ou préalablement déclaré dans le programme).

On peut alors déclarer une variable de type *Pile* comme suit :

Variable de type Pile : p

En utilisant cette structure de données, il est alors possible d'écrire des procédures et fonctions associées aux opérations admissibles sur le TAD pile.

Exemple

```
fonction pileVide(Pile p):booléen
{Cette fonction retourne vrai si la pile p est vide et faux sinon}
début fonction
si p.taille=0 alors pileVide retourne la valeur vrai
sinon pileVide retourne la valeur faux
fin fonction

fonction sommet(Pile p) : Élément
{Cette fonction retourne l'élément situé au sommet de la pile si la pile est non vide et envoie un message d'erreur sinon. Il faut donc veiller lorsqu'on appelle cette fonction à ne pas l'appeler sur une pile vide. Cette fonction ne modifie pas la pile.}
début
si pileVide(p) alors écrire(« ERREUR »)
sinon sommet retourne p.tabElement[p.taille]
fin fonction

procédure dépiler(Pile p)
{Cette procédure ôte de la pile le dernier élément entré. La taille de la pile est diminuée de 1. Si la pile est vide, l'appel à cette procédure provoque une erreur. }
début procédure
si pileVide(p) alors écrire(« ERREUR »)
sinon p.taille<-p.taille - 1
fin procédure

procédure empiler(Pile e, Élément e)
{Cette procédure ajoute l'élément e au somment de la pile. La taille de la pile est augmentée de 1. Si la pile est déjà pleine, empiler provoque un débordement. }
début procédure
{test de débordement}
si pilePleine(p) alors écrire(« Débordement de pile »)
```

sinon
p.taille=p.taille+1
p.tabElement[p.taille]<-e
fin sinon
fin procédure

fonction pilePleine(Pile p):booléen
{Cette fonction retourne vrai si la pile p est pleine et faux sinon}
début fonction
si p.taille=TAILLEMAX alors pilePleine retourne la valeur vrai
sinon pilePleine retourne la valeur faux
fin fonction

On remarquera la simplicité de cet ensemble de procédures et de fonctions (contrairement aux listes, en effet, on n'a jamais de décalages à gérer). Avec ce mode de représentation des piles, les opérations sont peu coûteuses. L'inconvénient tient à l'utilisation des tableaux. Il faut fixer une taille maximale de pile suffisamment grande pour éviter les débordement de pile et réserver en mémoire l'espace correspondant même si on ne manipule que des « petites » piles.

10.2.4 Représentation chaînée à l'aide de pointeurs

Pour s'affranchir des contraintes liées à l'utilisation des tableaux, on peut, comme pour les listes, utiliser une représentation chaînée à l'aide de pointeurs.

On introduit alors les types suivants, correspondant à une structure auto-référentielle :

Type
Bloc : enregistrement à 2 membres val : Élément
suivant : pointeur sur Bloc
PileChainee : pointeur sur Bloc

Avec cette structure, le sommet est toujours le premier élément de la chaîne et donc l'élément le plus accessible. La représentation chaînée des listes était coûteuse pour les parcours mais dans le cas des piles, on ne fait pas de parcours, on n'accède qu'au sommet.

Avec cette structure de données, on peut alors réécrire les procédures empiler et dépiler.

procédure depiler(PileChaînée p)
{Cette procédure ôte de la pile le dernier élément entré. Si la pile est vide, l'appel à cette procédure provoque une erreur. }
début procédure
si pileVide(p) alors écrire(« ERREUR »)
sinon p <contenu(p).suivant [1]
fin procédure

procédure empiler(PileChaînée p, Élément e)
{Cette procédure ajoute l'élément e au somment de la pile.}
1 variable auxiliaire de type Bloc: b
début procédure

[1] La fonction contenu(p) retourne la valeur contenu dans la case mémoire sur laquelle pointe Isabelle.

b.val<- e
b.suivant<-p
p<-adresse(b)
fin procédure

10.3 Les files

10.3.1 Définition et utilisation des files

Une file est semblable à une file d'attente à un guichet. Le premier arrivé est le premier servi (Fist in/First out FIFO). Une file a deux extrémités : on entre dans la file à une extrémité et on en sort de la file à l'autre (au guichet). La file diffère donc de la pile qui n'a qu'une seule extrémité.

Le type file est également très utilisé en informatique. C'est une file qui gère la suite des caractères entrés au clavier, le flux des impressions envoyés sur une imprimante.

10.3.2 Le TAD file

On note :

- E : ensemble d'éléments de même type
- F : ensemble de file pouvant être définies sur E
- f : file appartenant à F, p = $<e_1, e_2, ..., e_n>$ avec $\forall i \in [1,..,n], e_i \in E$
- e : élément appartenant à E

La spécification du TAD file comporte les opérations suivantes

- Deux constructeurs
  - Créer une file f.

    CreerFile : $\varnothing \rightarrow F$

    créerFile() : f, file vide
  - Ajouter un élément e à une file f.

    Ajouter : F x E -> F

    ajouter(f,e) : $<e_1, e_2, ..., e_{n,}, e>$
- Un prédicat qui teste si une file f est vide.

  FileVide : $F \rightarrow \{vrai, faux\}$

  fileVide(f) : vrai si f ne contient pas élément et faux sinon.
- Sélectionner le premier élément de la file f.

  Premier : $F \rightarrow E$

  premier(f) : e1, opération non définie si f est vide

- Une procédure de modification qui ôte le premier élément de la file f. Cette opération a pour conséquence de changer la taille effective de la file.

  supprimer : $F \rightarrow F$

  supprimer(l) : $<e_2, .., e_{n-1}>$, opération non définie si f est vide

10.3.3 Représentation contiguë des files

La représentation des files est moins simple que celle des piles. On peut représenter en machine une file f par un tableau et gérer deux indices qui indiquent respectivement le début et la fin de la file.

On peut alors représenter en machine le TAD file par le type enregistrement suivant :

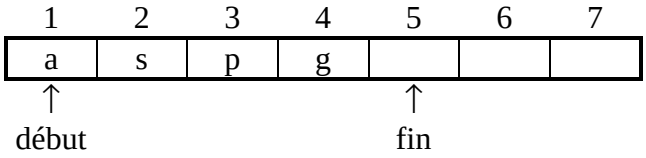
Type
File : enregistrement à 3 membres *tabÉlément : tableau de TAILLEMAX Éléments*
début, fin : entiers

où *Élément* désigne le type des éléments appartenant à la file (*Élément* est un type prédéfini ou préalablement déclaré dans le programme) et les champs *début* et *fin* indiquent ou il faut supprimer (*début*) et ajouter (*fin*) des éléments.

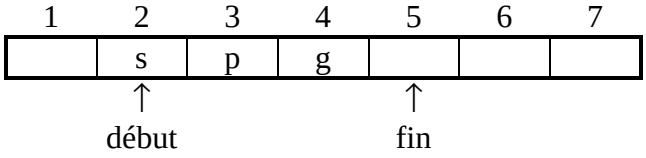
On peut alors déclarer une variable de type *File* comme suit :

Variable de File : f

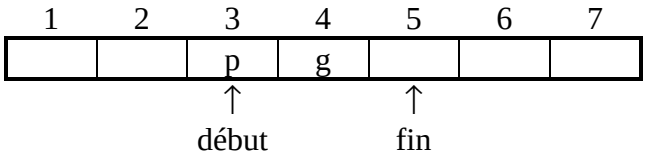
On peut ainsi représenter la file de caractères a, s, p, g par le tableau suivant. Les indices de début et de fin indiquent où supprimer et ajouter des éléments. On a *début* ≤ *fin.*



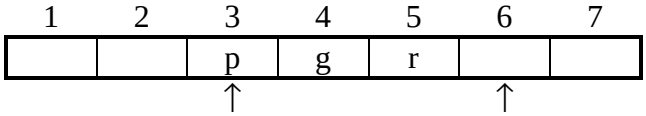
Mais si on ôte a et s et si on ajoute les éléments r, k, m et b, on rencontre un problème : on ne peut pas ajouter ce dernier élément alors même que le tableau n'est pas plein. On passe par les étapes suivantes en effet :



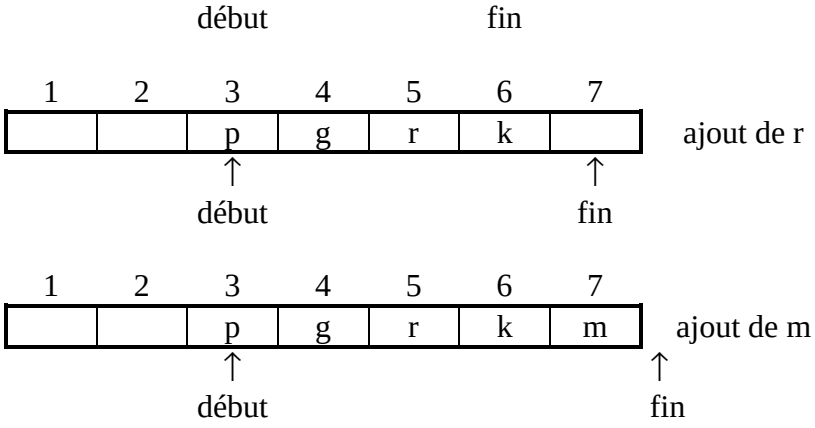
suppression de a



suppression de s

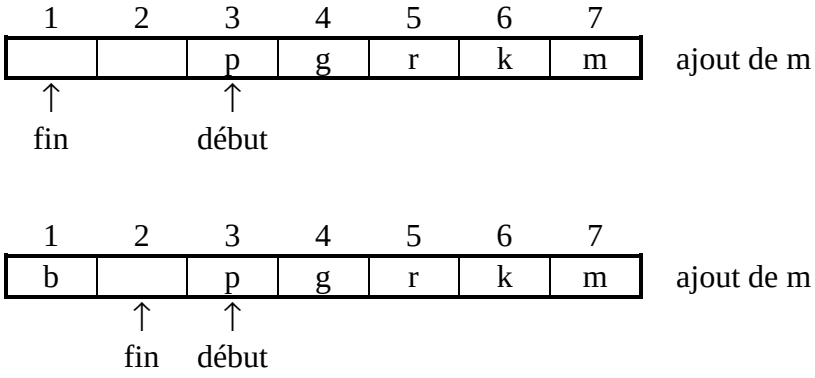


ajout de r



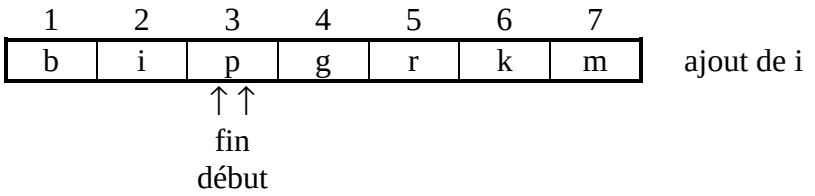
Pour éviter ce problème, il faudrait décaler tous les éléments d'une file vers la gauche chaque fois qu'on supprime un élément. Cette solution qui multiplie les décalages est trop coûteuse.

Une représentation linéaire intuitive ne convient pas. Il faut considérer la file comme une structure circulaire. la taille effective du tableau. Poursuivons sur l'exemple ci-dessus :



Dans ce cas, on supprime la contrainte *début* ≤ *fin*. L'ordre de la file ne correspond à l'ordre des indices du tableau. Dans la file le suivant de l'élément e peut avoir un indice inférieur à ce dernier si e est le dernier élément du tableau.

Une difficulté est cependant soulevée dans le cas suivant où les indices de début et de fin pointent sur la même cellule. On ne sait pas si cela signifie que la file est pleine ou vide.



Ceci nous amène à corriger le type TypeFile défini ci-dessus en ajoutant un champ booléen supplémentaire qui indique si la file est vide ou non.

Type
File : enregistrement à 4 membres *tabElements : tableau de TAILLEMAX Élément*
début,fin : entier
vide :booléen

En utilisant cette structure de données, il est alors possible d'écrire des procédures et fonctions associées aux opérations admissibles sur le TAD file. Ce point est laissé en exercice.

10.4 Représentation chaînée en C

10.4.1 Création d'une liste chaînée pour représenter une liste en machine et allocation dynamique de mémoire

Nous souhaitons représenter en machine la liste d'entiers <50,68,10,3> par une liste chaînée avec des pointeurs. On gère pour cela un ensemble d'enregistrements à deux champs, le premier appelé *val* contient les éléments de la liste et, le second appelé *suivant* est un pointeur qui pointe sur l'enregistrement suivant dans la liste.

Pour implanter en machine une représentation chaînée d'une liste d'entiers à l'aide de pointeurs, on introduit les types C suivants :

```
typedef struct bloc {
  int val;
  struct bloc *suivant;
            }
typedef struct bloc *ListeChaineeP;
```

Il est alors possible de définir une variable de type `ListeChaineeP` de la façon suivante :

```
ListeChaineeP l;
```

`l` est donc une variable pointeur sur une valeur de type struct bloc. Elle n'est encore pas initialisée. Pour lui attribuer une valeur initiale, on peut procéder comme suit :

```
/* Définir une variable de type struct bloc */
struct bloc b1;

/* Affecter des valeurs initiales à cette variable enregistrement */
b1.val = 50;
b1.suivant =  NULL;

/* Puis faire pointer l sur cette variable de type struct bloc */
l = &b1;
```

En conséquence, on a : {(*l).val=50} (ou de façon équivalente {l->val=50}) et {l->suivant=NULL}

Pour maintenant ajouter l'entier 68 en position 2 dans la liste, on peut procéder de la même manière en manipulant non plus une variable de type struct bloc mais deux comme suit :

```
/* Définir une variable de type struct bloc */
struct bloc b1,b2;

/* Affecter des valeurs initiales aux variables b1 et b2 */
b1.val = 50;
b1.suivant =  NULL;
b2.val = 68;
b2.suivant =  NULL;

/* Puis construire la liste chaînée */
l = &b1;
l-> suivant = &b2;
```

Cette façon de procéder implique de définir et de manipuler autant de variables de type struct bloc que d'éléments dans la liste que l'on souhaite représenter. C'est donc rapidement inefficace lorsque la longueur de la liste est élevée.

Il est possible de procéder différemment en faisant appel à une fonction prédéfinie, appelée malloc, qui permet de réserver une zone mémoire d'une taille donnée et, qui retourne l'adresse de cette zone. On réalise alors une allocation dynamique de mémoire. Sur l'exemple, on obtient :

```
/* Réserver une zone mémoire pour le premier enregistrement de la liste chaînée */
l=(ListeChaineeP)malloc(sizeof(struct bloc));
l->val = 50;

/*Réserver une zone mémoire pour le deuxième enregistrement de la liste chaînée */
l->suivant = (ListeChaineeP)malloc(sizeof(struct bloc));
l->suivant->val = 68;

...
```

sizeof(x), où x désigne un identificateur de type ou une expression, est un opérateur unaire du C qui renvoie le nombre d'octets nécessaire pour représenter en machine une valeur de type x.

Même si on n'a plus besoin de variables struct bloc intermédiaires pour créer la liste, cette méthode n'est pas efficace lorsque la longueur de la liste est très élevée. Classiquement, on procèdera comme suit :

```
int main() {
ListeChaineeP l;
int n,e,i;
printf("Donner la longueur de la liste");
scanf("%d",&n);
printf("Entrer les éléments de la liste (du dernier au
premier) :");
for(i=0;i<n;i=i+1) {
  scanf("%d",&e);
  l=inserer(l,e)
}
ListeChaineeP inserer(ListeChaineeP liste, int elt){
ListeChaineeP pt;
pt=(ListeChaineeP)malloc(sizeof(struct bloc));
pt->val=e;
pt->suivant=liste;
liste=pt;
return(liste);
}
```

10.4.2 Généralités sur l'allocation dynamique de mémoire

Pour initialiser une variable pointeur p, nous connaissons 3 façon de procéder : en la faisant pointer vers une variable déjà définie grâce à l'opérateur &, en lui affectant la valeur d'une autre variable pointeur de même type, en la faisant pointer sur rien. Grâce à l'allocation dynamique de mémoire, nous disposons d'un procédé supplémentaire. Un appel à malloc permet de lui affecter l'adresse d'une zone mémoire libre comme suit :

```
p=(<id. type>*)malloc(sizeof(<id. type>);
```

La fonction malloc, déclarée dans le fichier d'en-tête stdlib.h, réserve une zone mémoire et retourne son adresse, c'est-à-dire une valeur de type pointeur. Cette fonction attend comme argument une valeur entière strictement positive k. malloc(k) renvoie donc une valeur de type pointeur sur une zone mémoire de k octets. Par défaut, la zone mémoire de k octets n'étant pas typée, le pointeur retourné par un appel à malloc est un pointeur de type void. Un pointeur sur void est un pointeur générique. L'opérateur d'indirection ne s'applique par à un pointeur sur void. Aussi, lorsqu'on veut réserver une zone mémoire de taille k pour y stocker une valeur d'un type donné, il est nécessaire de faire un cast pour changer le type du pointeur retourné par malloc.

Exemple :

```
/*On définit une variable pointeur sur int */
int *pt;
/* On l'initialise grâce à un appel à la fonction malloc */
pt=(int*)malloc(2);
/* On utilise la zone mémoire sur laquelle pointe pt pour y
stocker la valeur 10 */
*pt=10;
```

Comme sur certaine machine, une valeur de type int est codée sur 4 octets plutôt que 2, il est préférable d'initialiser pt de la façon suivante :

```
pt=(int*)malloc(sizeof(int));
```

La portabilité du code est alors renforcée !

La fonction malloc renvoie la valeur `NULL` si elle n'a pas trouvé de zone mémoire disponible. Ainsi, pour éviter des erreurs, il est nécessaire après chaque appel à malloc de tester la valeur retournée comme suit :

```
pt=(int*)malloc(sizeof(int));
if(pt==NULL) {
  printf("Problème lors de l'allocation mémoire !");
  ...
}
```

Il est également possible de faire un appel à malloc pour réserver une zone mémoire pour un tableau comme suit :

```
int *pt;
pt=(int*)malloc(10*sizeof(int));
/* malloc renvoie l'adresse d'une zone mémoire de taille
égale à 10*sizeof(int) */
```

Pour récupérer la place mémoire allouée dynamiquement à une variable pointeur que l'on n'utilise plus, il ne suffit pas de changer la valeur de la variable. Il faut faire un appel à la fonction prédéfinie free.

Exemple :

```
int *pt;
pt=(int*)malloc(sizeof(int));
if(pt==NULL) {
  printf("Problème lors de l'allocation mémoire !");
}
else {
  *pt=10 ;
}
```

La variable pt pointe sur une zone mémoire contenant 10.

Si plus tard dans le programme, on exécute :

```
pt=NULL ;
```

L'adresse de la zone mémoire sur laquelle pointait pt est perdue et cette zone devient inacessible pendant toute la durée d'exécution du programme. Elle sera libérée uniquement à la terminaison du programme.

Pour libérer cette zone mémoire pendant l'exécution du programme, il ne faut pas affecter la valeur NULL à pt mais il faut faire appel à la fonction free comme suit :

```
free(pt);
```

`free(<id.pointeur>)` : libère la zone mémoire précédemment allouée à la variable pointeur `id. pointeur` grâce à un appel à `malloc`.

11 La récursivité

La récursivité est une notion essentielle en programmation. Au point que la récursivité est plus naturelle que l'itération dans certains langages comme Lisp. Mais la récursivité n'est pas seulement une méthode de programmation, c'est en soi une manière de considérer certains problèmes.

Dans ses très grandes lignes le principe de la récursivité consiste à résoudre un problème en se ramenant au même problème, de taille plus réduite, de manière à ce que d'étapes en étapes, on arrive à un problème trivial.

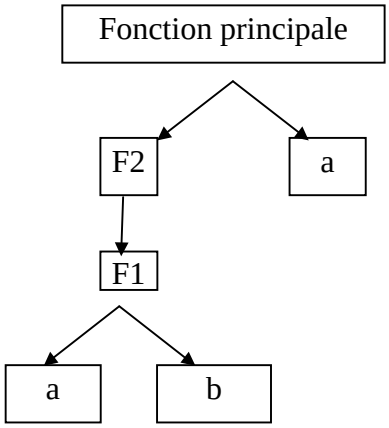
11.1 Le principe

Nous avons vu dans les cours qui précèdent qu'un programme peut comporter des sous-programmes telles que des procédures et des fonctions. La récursivité désigne un cas particulier où un sous-programme s'appelle lui-même.

11.1.1 Préliminaire : les arbres d'exécution

L'ensemble des relations de dépendances des sous-programmes dessine un arbre d'exécution. Dans le cas du programme suivant, on obtient l'arbre ci-dessous.

```
procédure a(...);
  début
  ...
  fin procédure a
procédure b(...);
  début
  ...
  fin procédure b;
fonction f1(...):...;
  début
  ...
  a(...);
  b(...);
  ...
fin fonction f1
fonction f2(...):...;
  début
  ...
  x← f1(...);
  ...
fin fonction f2
début fonction principale
resu← f2(...);
a(...) ;
fin fonction principale
```



La trace des appels de sous-programmes montre le parcours que suit l'exécution du programme dans cet arbre de dépendances. C'est un parcours en profondeur d'abord mais les fonctions et procédures sont ouvertes au cours du parcours descendant et fermées au cours de la remontée, c'est-à-dire dans l'ordre inverse.

Trace des appels de fonctions et procédures :

Début fonction principale
  Début f2
    Début f1
      Début a (1ère version)
      Fin a
      Début b
      Fin b
    Fin f1
  Fin f2
  Début a (2ème version)
  Fin a
Fin fonction principale

Rappelons par ailleurs que les appels de sous-programmes sont gérés par une pile des contextes d'exécution. A chaque appel de fonction ou procédure, un ensemble d'information est mémorisé : valeurs courantes des variables et paramètres, adresse de la valeur de retour.

11.1.2 Les fonctions récursives : définition

Une fonction récursive est une fonction qui s'appelle elle-même. Cette idée de s'appeler soi-même pour un sous-programme peut paraître surprenante mais la récursivité est une manière très naturelle de résoudre certains problèmes.

a. Relation de récurrence

Considérons par exemple la définition de l'opérateur mathématique de puissance. Il se définit très naturellement par une relation de récurrence :

$A^n = A * A^{n-1}$ si $n > 0$

$A^n = 1$ si $n = 0$

Cette récurrence est ici très classique : on a

- un cas général (n>0) qui permet de transformer pas à pas un problème complexe en un problème simple ;
- un cas particulier (n=0), qui est le point d'appui de la récurrence, le cas élémentaire dont la résolution ne fait pas appel à la relation de récurrence.

b. Calcul récursif

Cette relation de récurrence peut également s'interpréter comme une méthode de calcul. On cherche à calculer A^i. Si i=0 alors le résultat est 1, sinon il faut multiplier *base* (A) par le résultat du calcul de la puissance i-1ème de *base* (A^{i-1}). Un algorithme récursif en découle :

```
Fonction puissance (entier base, entier exposant) : entier
début
si exposant = 0 alors retourne 1
sinon retourne base*puissance(base, exposant-1)
fin
```

On remarque en effet que le calcul de *puissance(base,exposant)* fait appel à la fonction *puissance(base, exposant-1)*. Cette fonction distingue deux cas, comme la relation de récurrence précédente :

- un cas général (*retourne base*puissance(base, exposant-1)*) qui comporte un appel récursif ;
- un cas particulier qui joue le rôle de point d'appui ou de condition d'arrêt (*retourne 1*) ; dans ce cas, il n'y pas d'appel récursif à la fonction puissance.

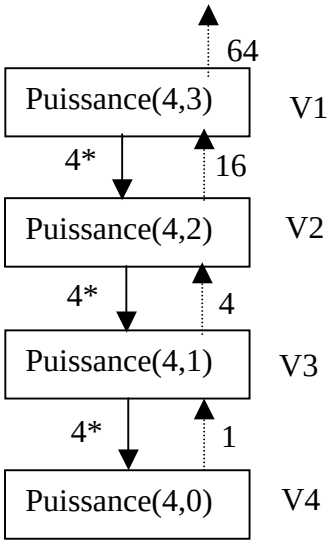
11.1.3 Fonctionnement de la récursivité

a. Simulation de l'exécution pas à pas

En réalité, le fait que la même fonction soit appelée plusieurs fois dans le même programme n'a rien d'original. Ce sont des versions différentes de cette fonction qui sont exécutées à chaque fois. Considérons pour l'exemple le calcul de la fonction puissance(4,3) (soit 4^3) et numérotons v1, v2, v3... les différentes versions de la fonction puissance.

Montrons ce qui se passe lors de l'exécution de la fonction puissance(4,3). Cette exécution est schématisée dans la figure ci-dessous. Chaque appel à la fonction puissance correspond à une exécution de la séquence des instructions qui composent cette fonction. Chaque exécution correspond à une version différente de la fonction et figure dans un encadré. Le premier appel provoque donc l'exécution de la version v1 avec comme paramètres (base1=4, exposant1=3). Comme exposant1>0, l'exécution de ces instructions fait de nouveau appel à la fonction puissance, ce qui provoque l'exécution de la version v2 de cette suite d'instructions avec cette fois comme paramètres (base2=4, exposant2=2). Comme exposant2>0, là encore, l'exécution de ces instructions fait appel à la fonction puissance, ce qui provoque l'exécution de la version v3 de cette suite d'instructions avec cette fois comme paramètres (base3=4, exposant3=1). Cette fois encore, comme exposant3>0, l'exécution de ces instructions fait appel à la fonction puissance, ce qui provoque l'exécution de la version v4 de cette suite d'instructions avec cette fois comme paramètres (base4=4, exposant4=0). Cette fois, exposant4 étant nul, le calcul de puissance ne fait pas appel à un sous-programme : le résultat est immédiat : 1. C'est la valeur de retour de la version v4 de puissance (case grisée). Le renvoi de cette valeur au programme appelant, c'est-à-dire v3, permet de calculer le résultat

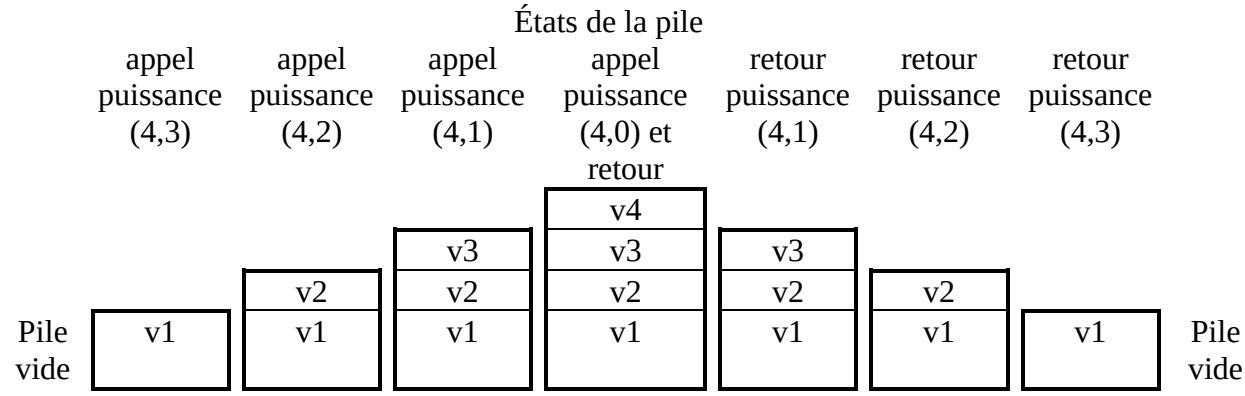
de la fonction puissance(4,1). Le résultat de v3 est 4*1=4, valeur qui est retournée au programme appelant, v2, et qui permet le calcul de puissance(4,2). Le résultat de v2 est 4*4=16. Cette valeur est à son tour retournée au programme appelant (v1) et permet de calculer le résultat de puissance(4,3). Le résultat de v1 est 4*16=64. L'arbre d'exécution de puissance(4,3) est:



L'exécution de puissance(4,3) fait ainsi appel à plusieurs couples de variables : base1, exposant1 ; base2, exposant2 ;...

On remarque sur cette simulation que la première fonction appelée est la première fonction puissance à se terminer est la plus enchâssée, c'est v4. Les fonctions enchâssées les uns dans les autres se terminent dans l'ordre inverse de celui où elles sont appelées. Rappelons en effet, que les contextes d'exécution des procédures et fonction sont gérés dans une structure de pile. Chaque version de la fonction puissance a son propre contexte d'exécution et ces différents contextes sont empilés dans la pile des contextes d'exécution.

Voici comment évolue cette pile lors de l'exécution de puissance(4,3) :



b. Simulation d'un appel récursif

Voici la simulation de l'exécution de puissance(4,3).

Début fonction puissance (version 1) {base1=4, exposant1=3}
{(exposant1=0)=faux} début sinon
Appel de la fonction puissance (version 2)
{paramètres effectifs : base1=4, exposant1-1=2}

{paramètres formels : base2=4, exposant2=2}
{(exposant2=0)=faux} début sinon
Appel de la fonction puissance (version 3)
{paramètres effectifs : base2=4, exposant2-1=1}
{paramètres formels : base3=4, exposant3=1}
{(exposant3=0)=faux} début sinon
Appel de la fonction puissance (version 4)
{paramètres effectifs : base3=4, exposant3-1=0}
{paramètres formels : base4=4, exposant4=0}
{(exposant3=0)=vrai} retourne 1
Fin puissance (version 4)
{paramètres formels : base4=4, exposant4=0}
{paramètres effectifs : base3=4, exposant3=1}
*retourne 4*1=4*
fin sinon
Fin puissance (version 3)
{paramètres formels : base3=4, exposant3=1}
{paramètres effectifs : base2=4, exposant2=2}
*retourne 4*4=16*
fin sinon
Fin puissance (version 2)
{paramètres formels : base2=4, exposant2=2}
{paramètres effectifs : base1=4, exposant1=3}
*retourne 4*16=64*
fin sinon
Fin puissance (version 1)

c. Élégance et généricité de la programmation récursive

Pour le calcul de puissance(4,3), en réalité, on aurait pu introduire plusieurs fonctions différentes :

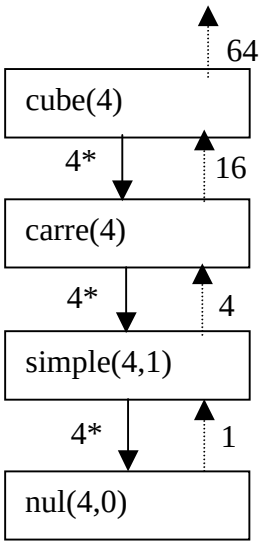
```
Fonction cube (entier base) : entier
début
retourne base*carré(base)
fin

Fonction carré (entier base) : entier
début
retourne base*simple(base)
fin

Fonction simple (entier base) : entier
début
retourne base*nul(base)
fin

Fonction nul (entier base) : entier
début
retourne 1
fin
```

Le calcul de puissance(4,3) correspond cette fois au calcul de cube(4). L'exécution de cette fonction provoque l'appel à carré(4) qui fait elle-même appel à simple(4) qui fait encore appel à nul(4). On a alors l'arbre d'exécution suivant :



On n'est plus ici dans un cas de récursivité puisque chaque fonction fait appel à une fonction différente mais cela suppose de définir autant de fonctions différentes que nécessaire. Il est beaucoup plus économique et rationnel d'introduire un deuxième paramètre (exposant) et de n'utiliser qu'une seule fonction générique.

On le voit, il n'y pas de différence fondamentale pour une fonction entre s'appeler elle-même ou' appeler une autre fonction. La seule difficulté concerne le risque que la récursivité ne se termine pas.

d. Importance du point d'appui de la récursion

Pour mettre en évidence l'importance de ce point d'appui, considérons la formule de récurrence suivante pour le calcul de la puissance :

$A^n=A^{n+1}/A$ si $n > 0$

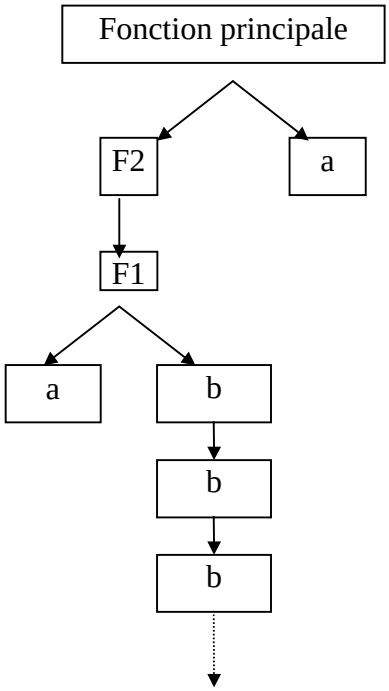
$A^n=1$ si $n=0$

La structure de cette récurrence est identique à la précédente, avec un cas général et un cas particulier. Mais la cette récurrence ne donne pas un programme récursif valide. En effet la définition suivante comporte une récursion infinie. Cette fonction comporte comme la précédente deux cas, l'un récursif et l'autre non, mais l'arrêt de la récursion sur le cas non récursif n'est pas garanti.

```
Fonction puissance-infinie (entier base, entier exposant) : entier
début
si exposant = 0 alors retourne 1
sinon retourne puissance(base, exposant-1)/base
fin
```

Le calcul de puissance-infinie(4,3) fait appel à puissance-infinie(4,4) qui fait appel à puissance-infinie(4,5), puisssance-infinie(4,6)... puissance-infinie(4,n),... à l'infini. On ne tombe jamais sur le cas d'arrêt puissance-infinie(4,0).

Dans le cas d'un récursion infinie, on a un arbre d'exécution comportant une branche infinie. Dans l'exemple du programme abstrait présenté ci-dessus, supposons que b soit une fonction récursive qui s'appelle elle-même sans point d'arrêt, l'arbre d'exécution aura l'allure suivante.



De la même manière qu'il faut toujours vérifier qu'une boucle *tant que* se termine, il faut donc veiller à ce que toute les exécutions possibles d'un programme récursif butent sur un point d'appui.

On peut simuler l'exécution d'un programme récursif qui se termine par une procédure itérative. On obtient la fonction *puissance-itérative* suivante (NB. il y a d'autres manières plus simples d'effectuer ce calcul itératif).

Fonction puissance-itérative (entier base, entier exposant) : entier
2 variables auxiliaires entières : i et résultat
i<-exposant
tant que i> 0 faire {recherche du point d'appui}
i<-i-1
fin tant quse
résultat<-1 {Le point d'appui est atteint}
tant que i≤ exposant faire
*résultat<-base**
i<-i+1
fin tant que
retourne résultat

On retrouve ici les deux boucles, la première va du plus général au plus particulier, c'est la recherche du point d'appui ; la deuxième va dans le sens inverse.

11.2 Les différents types de récursion

On distingue différent types de récursion selon l'endroit où se trouve l'appel récursif dans le codes d'une fonction ou d'une procédure et selon le nombre d'appel récursif qu'elle comporte.

Les cas les plus simples sont les fonctions ou procédures récursives terminales, c'est à dire qu'il n'y a qu'un seul appel récursif et que c'est la dernière instruction du sous-programme.

11.2.1 Exemples de fonctions récursives simples terminales

Citons quelques exemples simples très classiques de fonction récursive.

```
Fonction factorielle-récursive( entier n) : entier
{Calcule la factorielle de n}
1 variable auxiliaire entière : res
si n=0 alors retourne la valeur 1
sinon   res<-n*factorielle-recrusive(n-1)
        retourne la valeur res
        fin sinon
fin fonction
```

```
Fonction somme-vecteur(tableau d'Eléments : t, entier nbElements) : entier
{Calcule la somme des éléments d'un vecteur. Les éléments sont indicés à partir de 0
dans le tableau qui représente le vecteur.}
si n=0 alors retourne t[0]
sinon retourne t[n]+somme(t,n-1)
fin fonction
```

Dans les cas très simples de récursion simple terminale, on montre qu'il y a équivalence (en nombre d'opérations) entre la version récursive et la version itérative. De plus, on dispose de méthode pour transformer une fonction récursive en une fonction itérative. Cette méthode repose sur l'explicitation de l'empilement des contextes.

```
Fonction factorielle-itérative( entier n) : entier
{Calcule la factorielle de n, de manière itérative}
2 variables auxiliaires entières : res et i
i<-n
tant i <>0 faire            {recherche du point d'appui ; empilement des contextes
d'exécution}
i<-i-1
fin tant que
res<-=1                     {point d'appui}
tant que i<>n faire         {calcul ; dépilement}
res<-res*i
i<-i+1
fin tant que
retourne res
fin fonction
```

Evidemment dans certains cas, cette méthode générale est peu élégante. Dans le cas de factorielle, la première boucle est inutile :

```
Fonction factorielle-itérative2( entier n) : entier
{Calcule la factorielle de n, de manière itérative.}
2 variables auxiliaires entières : res et i
res<-1                          {point d'appui}
pour i variant de 1 à n faire
res<-res*i
fin pour
retourne res
fin fonction
```

11.2.2 Les récursions plus complexes

Certaines récursions sont plus complexes. Transformer une procédure récursive en procédure itérative est alors plus complexe et nous n'aborderons pas ce problème dans le cas général ici.

a. Récursion double : l'exemple de fibonacci

Dans les exemples présentés jusqu'ici, une fonction ne comporte qu'un seul appel récursif mais elle peut en comporter plusieurs. On parle alors de récursion multiple.

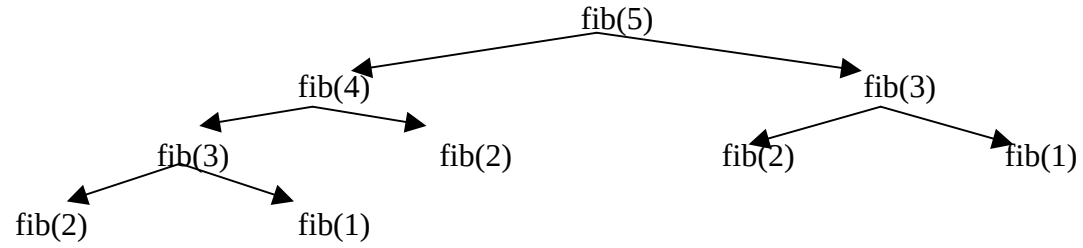
La suite de fibonacci est un exemple très classique de récursion double. Cette suite se définie de la manière suivante : c'est une suite doublement récurrente.

$u_1=u_2=1$
$u_i=u_{i-1}+u_{i-2}$ $\quad 1<i\leq n$

On peut calculer un valeur u_n de cette suite par la fonction suivante :

```
fonction fib (entier n) :entier
si n=1 ou n=2 alors retourne 1
sinon retourne fib(n-1)+fib(n-2)
fin fonction
```

Le graphe d'exécution pour fib(5) est le suivant :



Sur cet exemple simple, on constate que cette fonction doublement récursive n'est pas efficace. Pour calculer fib(5), on fait 2 fois appel à fib(3) par exemple. On exécute deux fois le même calcul. Si on avait calculer fib(6), on aurait fait appel 2 fois à fib(4) et 3 fois à fib(3).

Il faut ainsi veiller à ce que la récursivité ne conduise pas à des calculs dupliqués et inutile. Dans le cas de la fonction fib, il existe une méthode simple améliore significativement l'efficacité de cette fonction. Il s'agit de conserver dans un tableau les valeurs de fib déjà calculées pour éviter de recommencer les mêmes calculs à plusieurs reprises.

1 variable globale tabFib qui est un tableau de TAILLEMAX entiers initialisé à 0

fonction fibAméliorée (entier n) :entier

début fonction
si n=1 ou n=2 alors retourne 1
sinon si tabFib[n]=0 alors tabFib[n]<- fib(n-1)+fib(n-2)
fin si
retourne tabFib[n]
fin sinon
fin fonction

b. Récursion croisée

Une autre forme plus complexe de récursivité est le cas où la récursion met en œuvre 2 fonctions différentes. Dans l'exemple suivant, le calcul de a(n, …) dans le cas général fait appel à b(n,…) qui lui-même fait appel à a(n-1,…)

fonction a (n,…)
1 variable x auxiliaire
début fonction
…
si n=0 alors retourne 0
sinon x<-b(n,…)
retourne x
fin sinon
fin fonction a

fonction b (n,…)
1 variable x auxiliaire
début fonction
…
si n=1 alors retourne 0
sinon x<-a(n-1,…)
retourne x
fin sinon
fin fonction b

On parle dans ce cas de récursion croisée.

12 Quelques algorithmes élémentaires de tri et de recherche

12.1 Algorithmes élémentaires de tri

L'action de trier ou d'ordonner une liste d'objets selon une relation d'ordre linéaire donnée, comme la relation « inférieure ou égale » pour les nombres, est tellement importante et fréquente que le sujet a été abondamment étudié. Ce problème est un classique de l'informatique.

On suppose que les n objets à trier sont des enregistrements composés de plusieurs champs. Un de ces champs, appelé la clé, est d'un type sur lequel il est possible de définir une relation d'ordre linéaire « inférieure ou égale ». On souhaite que le tri se fasse in situ, c'est-à-dire que le résultat soit au même endroit que la liste d'enregistrements initiale. On introduit donc les types suivants :

```
type
Objet = enregistrement à deux membres
    valeur de type Valeur
    cle de type Cle
    fin enregistrement
ListeObjet = enregistrement à deux membres
    objets de type tableau unidimensionnel de 100 cellules de type Objet (indicées de 1 à
    100)
    nb de type entier
    fin enregistrement
```

Notre objectif est de réorganiser une liste d'enregistrements de telle sorte que les valeurs de leur champ clé forment une suite monotone croissante. Les enregistrements n'ont pas nécessairement des valeurs toutes distinctes et aucune supposition n'est faite sur l'ordre relatif de champs ayant la même clé.

12.1.1 Tri par bulles

L'idée de ce tri est d'imaginer que les enregistrements sont rangés dans un tableau vertical et de permettre aux enregistrements dont les clés sont les plus petites de remonter comme des bulles vers le haut du tableau.

Ainsi, à chaque phase de l'algorithme, on fait remonter un enregistrement vers le haut. Pour cela, on parcourt le tableau du bas vers le haut et, si on rencontre deux éléments en ordre inverse (celui de plus grande clé placé « sur » celui de plus petite clé), on les inverse. A la première phase, on regarde tous les enregistrements du tableau, et celui de plus petite clé est remonté en haut. Puis, on ne considère plus que les n-1 enregistrements non encore triés, et parmi ces derniers, on fait remonter le plus petit en position 2 ...

Cet algorithme peut être programmé sous la forme de la procédure suivante :

```
procedure TriBulle(ListeObjet l)
début
variables locales entières : i et j
pour i variant de 1 à l.nb faire
    pour j variant de l.nb à i+1 faire
        si l.objets[j].cle < l.objets[j-1].cle alors echanger(l.objets[j], l.objets[j-1])
        finsi
```

```
    fin pour
fin pour
fin procédure
```

Exemple

Voici la liste des notes de 5 étudiants. On souhaite réaliser un tri par ordre croissant des notes.

	Nom	Note
1	Balto	14
2	Sitbon	8
3	Granat	16
4	Garel	9
5	Joly	18
6	Enaux	2

Les types associés sont :

```
Etudiant = enregistrement à deux membres
    nom de type chaine de caractères
    note de type entier
    fin enregistrement
ListeEtudiant = enregistrement à deux membres
    etudiants de type tableau unidimensionnel de 100 cellules de type Etudiant
    nb de type entier
    fin enregistrement
```

Avec la procédure *TriBulle*, la liste des étudiants sera triée en trois phases (par phase nous entendons un parcours du tableau, c'est-à-dire un passage dans la boucle *pour i variant de 1 à l.nb*) :

Enaux	2
Balto	14
Sitbon	8
Granat	16
Garel	9
Joly	18

{i=1}

Enaux	2
Sitbon	8
Balto	14
Garel	9
Granat	16
Joly	18

{i=2}

Enaux	2
Sitbon	8
Garel	9
Balto	14
Granat	16
Joly	18

{i=3}

En phases 4 et 5, la procédure ne change rien car Balto, Granat et Joly sont correctement ordonnés. Ces phases sont donc inutiles.

De façon générale, si au cours d'une phase, aucun échange n'a été effectué, on peut arrêter la procédure de tri car les enregistrements restant à trier le sont déjà.

La version améliorée de tri par bulles est donc :

```
procedure TriBulleAmeliore(ListeObjet l)
début
variables locales entières : i et j
variable booléenne : stop
I<-1
faire
    stop <- true
```

```
    pour j variant de l.nb à i+1 faire
        si l.objets[j].cle < l.objets[j-1].cle alors
            echanger(l.objets[j],l.objets[j-1]) ;
            stop <- faux
            fin si
        fin pour
    i <- i+1
tant que stop=faux et i<l.nb
fin procedure
```

Ce tri est efficace dans les situations où les listes sont déjà « presque » triées, comme par exemple, dans le cas où on cherche à insérer un enregistrement à la bonne place dans une liste déjà triée. Dans ce cas, deux phases suffiront à retrier la liste : une première pour faire remonter le nouvel enregistrement à la bonne place, une seconde pour indiquer que la liste est correctement triée.

La notion d'efficacité repose ici sur celle du coût d'un procédure. De manière classique, on mesure le coût d'une procédure par le nombre d'opérations élémentaires qu'elle effectue. Dans la première version du tri donnée ci-dessus, on peut ainsi considérer que ce coût se mesure en nombre de comparaisons entre clés et en nombre de déplacements d'objets. Ceci sous-entend que les opérations d'initialisation et d'incrémentation de compteurs sont négligées.

Si l'on retient cette définition du coût, on peut calculer le coût du tri de notre liste d'étudiant par la procédure initiale de tri :

	Nb de comparaisons de clés	Nb de déplacements d'objets
1ère phase (i=1)	5	6 (soit 3 échanges)
2ème phase	4	4
3ème phase	3	2
4ème phase	2	0
5ème phase	1	0
Total	**15**	**12**

La deuxième version du tri permet de supprimer la dernière phase et supprime donc 1 comparaison. Sur cet exemple simpliste, l'économie est mineure mais elle peut-être importante sur d'autres jeux de données.

12.1.2 Tri par insertion

Dans cet algorithme, l'objectif est qu'à l'issue de la i^{ème} phase, tous les enregistrements situés entre la première et la i^{ème} position soient triés. Pour cela, à la i^{ème} phase, on insère le i^{ème} enregistrement de la liste à la bonne position parmi les (i-1) premiers, sachant que ces enregistrements ont été correctement triés dans les phases précédentes.

Cet algorithme peut être programmé sous la forme de la procédure suivante :

```
procedure TriInsertion(ListeObjet l)

début
variables locales entières : i et j
pour i variant de 2 à l.nb faire
```

```
        j<-i
        tant que l.objets[j].cle < l.objets[j-1].cle et j<>1
        faire
            echanger(l.objets[j],l.objets[j-1]) ;
            j<-j-1
        fin tant que
    fin pour
    fin procédure
```

Exemple

Avec la procédure *TriInsertion*, la liste des étudiants sera triée en 5 phases :

Balto	14
Sitbon	8
Granat	16
Garel	9
Joly	18
Enaux	2

Liste initiale

Sitbon	8
Balto	14
Granat	16
Garel	9
Joly	18
Enaux	2

{i=2}

Sitbon	8
Balto	14
Granat	16
Garel	9
Joly	18
Enaux	2

{i=3}

Sitbon	8
Garel	9
Balto	14
Granat	16
Joly	18
Enaux	2

{i=4}

Sitbon	8
Garel	9
Balto	14
Granat	16
Joly	18
Enaux	2

{i=5}

Enaux	2
Sitbon	8
Garel	9
Balto	14
Granat	16
Joly	18

{i=6}

Si on évalue selon les mêmes règles que précédemment le coût de ce tri, on obtient le résultat suivant :

	Nb de comparaisons de clés	Nb de déplacements d'objets
1ère phase (i=1)	1	2
2ème phase	1	0
3ème phase	3	3
4ème phase	1	0
5ème phase	5	6
Total	**11**	**11**

Ce deuxième algorithme de tri s'avère donc plus efficace pour cette liste particulière.

12.1.3 Tri par sélection-échange

L'idée de ce tri est la suivante : à la i^{ème} phase, on sélectionne l'enregistrement ayant la plus petite clé parmi les enregistrements positionnés de i à n dans la liste, et on l'échange avec l'enregistrement positionné en i. En conséquence, à la i^{ème} phase les i premiers enregistrements sont triés, mais contrairement au tri par insertion, le nouvel enregistrement trié ne se trouve pas en i mais à une position en aval.

La procédure associée est donc :

```
procedure TriSelectionEchange(ListeObjet l)

début
variable de type Cle : CleMin
variables de type entier : PositionMin,i,j
pour i variant de 1 à l.nb-1 faire
    PositionMin <-.i
    CleMin <- l.objets[i].cle
    pour j <- i+1 à l.nb faire
        si l.objets[j].cle < CleMin alors
            CleMin<-l.objets[j].cle;
            PositionMin<-j
            Fin si
        Fin pour
    echanger(l.objets[i],l.objets[PositionMin]) ;
fin pour
fin procédure
```

Exemple

Avec la procédure *TriSelectionEchange*, la liste des étudiants sera triée en phases :

Balto	14
Sitbon	8
Granat	16
Garel	9
Joly	18
Enaux	2

Liste intiale

Enaux	2
Sitbon	8
Granat	16
Garel	9
Joly	18
Balto	14

{i=1}

Enaux	2
Sitbon	8
Granat	16
Garel	9
Joly	18
Balto	14

{i=2}

Enaux	2
Sitbon	8
Garel	9
Granat	16
Joly	18
Balto	14

{i=3}

Enaux	2
Sitbon	8
Garel	9
Balto	14
Joly	18
Granat	16

{i=4}

Enaux	2
Sitbon	8
Garel	9
Balto	14
Granat	16
Joly	18

{i=5}

On peut remarquer que, contrairement aux deux algorithmes précédents, ce nouvel algorithme de tri assure qu'un objet sélectionné trouve d'emblée sa place définitive. En revanche, cette méthode de tri ne préserve pas l'ordre relatif des objets, ce qui est préjudiciable si la liste est en partie triée au départ.

Le tableau suivant récapitule les coût du tri de notre liste d'étudiant par cette méthode :

	Nb de comparaisons de clés	Nb de déplacements d'objets
1ère phase (i=1)	5	2
2ème phase	4	0
3ème phase	3	2
4ème phase	2	2
5ème phase	1	2
Total	**15**	**8**

On note donc des différences de fonctionnement et de coût entre ces trois algorithmes sur cet exemple de données particulier. Pour en tirer des conclusions fiables, il faudrait faire cette estimation de coût sur une liste quelconque[2].

Ces notions de coût sont importante pour choisir quel algorithme de tri utiliser sur quel jeu de données. Si les données sont partiellement triées au départ, on privilégiera le tri à bulles ; si le déplacement d'objet est considéré comme coûteux par rapport à celui des comparaisons, on préfèrera le tri par sélection échange…

12.2 Algorithmes élémentaires de recherche

L'action de rechercher un objet parmi une liste d'objets est également très fréquente en informatique. Cette problématique a également donné lieu à de nombreuses études.

On suppose que les objets sont des enregistrements de type *Objet* et qu'ils sont représentés en machine à l'aide d'une variable de type *ListeObjet*.

12.2.1 La recherche séquentielle

La première méthode pour rechercher un objet dans une liste consiste à faire une recherche séquentielle (ou linéaire). Afin de trouver l'objet recherché, on examine successivement tous les éléments de la liste. Pour programmer cet algorithme, on peut écrire une fonction qui renvoie un entier, la position de l'objet recherché dans la liste :

```
fonction RechercheSequentielle(ListeObjet l, Objet o):entier
debut
1 variable locale entiere : i
1 variable locale booleenne : trouve
i <-1
trouve <- faux
tant que (trouve=faux) et (i<=l.nb) faire
    trouve <-  (o.valeur = l.objets[i].valeur)
    i <- i+1
    fin tant que
si (trouve=vrai) alors RechercheSequentielle retourne i-1
sinon RechercheSequentielle retourne 0
fin si
fin fonction
```

[2] Ceci sort du cadre de ce cours de 1ère année.

Pour éviter de faire deux tests dans la boucle *tant que*, on peut procéder différemment en utilisant une « sentinelle » en début de liste comme suit. L'idée est d'insérer une butée en début de tableau pour éviter de tester à chaque pas qu'on ne sort pas du tableau. En pratique on insère en début du tableau (à la position 0 si le 1er élément de la liste figure à l'indice 1) l'élément qu'on recherche. On est donc certain de trouver l'élément recherché mais si on ne le trouve qu'à l'indice 0 cela signifie qu'il ne figurait pas dans le tableau initial.

```
fonction RechercheSequentielle2(ListeObjet l, Objet o):entier
debut
1 variable locale entiere : i
i <-l.nb
l.objets[0] <-o
tant que o.valeur <- l.objets[i].valeur faire i <-i-1
   fin tant que
RechercheSequentielle2 retourne i
fin fonction
```

Exemple

Simulation d'un appel à la fonction *RechercheSequentielle2* lorsque les paramètres effectifs sont, pour `l`, la liste des étudiants de l'exemple précédent et, pour `o`, l'étudiant Garel ayant une note de 9 :

{i=6}
{l.etudiants[0].nom=Garel ; l.etudiants[0].note=16}
{o.nom=Garel ; l.etudiants[6].nom=Enaux ; (o.nom ≠ l.etudiants[6].nom)=vrai}
{i=5}
{o.nom=Garel ; l.etudiants[5].nom=Joly ; (o.nom ≠ l.etudiants[5].nom)=vrai}
{i=4}
{o.nom=Garel ; l.etudiants[4].nom=Garel ; (o.nom ≠ l.etudiants[4].nom)=faux}
Retourne le valeur 4.

12.2.2 La recherche dichotomique

Si la liste est triée par ordre croissant des valeurs de clé, on peut utiliser une autre méthode pour rechercher un objet. Au lieu de faire une recherche séquentielle, on visite en premier l'objet se trouvant au milieu de la liste. S'il s'agit de l'objet recherché, l'algorithme s'arrête, sinon on compare la clé de l'objet recherché avec la clé de l'objet positionné au milieu de la liste. Le résultat de cette comparaison détermine de quel côté du tableau peut se trouver l'objet recherché. Dans ce sous-tableau, on effectue la même analyse. On s'arrête lorsque l'objet recherché est trouvé, ou lorsque le sous-tableau considéré est vide, ce qui signifie que l'objet recherché n'appartient pas à la liste.

```
fonction RechercheDichotomique(ListeObjet l,
                    Objet o):entier
debut
3 variables entieres : i,g,d
g <- 1
d <-l.nb
faire
```

i <- (g+d) div 2
si o.cle<l.objets[i].cle alors d <- i-1
sinon g <- i+1
tant que (o.valeur □ l.objets[i].valeur) et (g<=d)
si o.valeur = l.objets[i].valeur alors RechercheDichotomique retourne i
sinon RechercheDichotomique retourne 0
fin si
fin fonction

Exemple

Simulation d'un appel à la fonction *RechercheDichotomique* lorsque les paramètres effectifs sont, pour `l`, la liste triées des étudiants de l'exemple précédent et, pour `o`, l'étudiant Garel ayant une note de 9.

{g=1}{d=6}
{i=3}
{o.note=9 ;l.etudiants[3].note=9 ; (o.note<l.etudiants[3].note)=faux}
{g=2}
{o.nom=Garel ;l.etudiants[3].nom=Garel ; (o.nom=l.etudiants[3].nom)=vrai}
{o.nom=Garel ;l.etudiants[3].nom=Garel ; (o.nom=l.etudiants[3].nom)=vrai}
Retourne la valeur 3.