**UNIVERSITE PAUL SABATIER**

TOULOUSE III

# Algorithmique, Structures de données et langage C

L3 IUP AISEM/ICM

Janvier 2005

J.M. ENJALBERT

# Chapitre 1

# Rappels et compléments de C

## 1.1 Structures

Une structure rassemble des variables, qui peuvent être de types différents, sous un seul nom ce qui permet de les manipuler facilement. Elle permet de simplifier l'écriture d'un programme en regroupant des données liées entre elles.

Un exemple type d'utilisation d'une structure est la gestion d'un répertoire. Chaque fiche d'un répertoire contient (par exemple) le nom d'une personne, son prénom, son adresse, ses numéros de téléphone, etc... Le regroupement de toutes ces informations dans une structure permet de manipuler facilement ces fiches.

Autre exemple: On peut représenter un nombre complexe à l'aide d'une structure.

On définit une structure à l'aide du mot-clé: `struct` suivi d'un identificateur (nom de la structure) et de la liste des variables qu'elle doit contenir (type et identificateur de chaque variable). Cette liste est délimitée par des accolades. Chaque variable contenue dans la structure est appelée un *champ* ou un *membre*.

Définir une structure consiste en fait à définir un nouveau type. On ne manipulera pas directement la structure de la même manière que l'on ne manipule pas un *type*. On pourra par contre définir des variables ayant pour type cette structure.

Exemple:

```
struct complexe
  {
    double reel;
    double imag;
  };

struct complexe x,y;
```

Dans cet exemple, on définit une structure contenant deux réels puis on déclare deux variables ayant pour type `struct complexe`.

Les opérations permises sur une structure sont l'affectation (en considérant la structure comme un tout), la récupération de son adresse (opérateur &) et l'accès à ses membres.

On accède a la valeur d'un membre d'une structure en faisant suivre l'identificateur de la variable de type structure par un point suivi du nom du membre auquel on souhaite accéder. Par exemple `x.reel` permet d'accéder au membre 'reel' de la variable x.

On peut initialiser une structure au moment de sa déclaration, par exemple:

```
struct complexe x={10,5};
```

On peut définir des fonctions qui renvoie un objet de type structure. Par exemple, la fonction suivante renvoie le complexe conjugué de x:

```
struct complexe conjugue(struct complexe x);
{
  struct complexe y;
```

```
  y.reel=x.reel;
  y.imag=-x.imag;

  return y;
}
```

L'utilisation de cette fonction pourra ressembler à:

```
struct complexe x,z;

z=conjugue(x);
```

L'affectation revient à affecter chaque membre de la structure, par exemple:

```
struct complexe x,z;

x=z;
```

est équivalent à:

```
struct complexe x,z;

x.reel=z.reel;
x.imag=z.imag;
```

## 1.2 Types synonymes

Le langage C permet de créer de nouveaux noms de types de données grace à la fonction `typedef`. Par exemple: `typedef int longueur` fait du nom `longueur` un synonyme de `int`. La déclaration: `longueur l` est alors équivalente à `int l`.

Autre exemple, la déclaration `typedef struct complexe comp,*p_comp` permet de créer deux nouveaux mot-clés: `comp` équivalent à `struct complexe` et `p_comp` équivalent à `struct complexe*` (pointeur de `struct complexe`).

Attention, un `typedef` ne crée pas de nouveaux types mais simplement de nouveaux noms pour des types existants.

## 1.3 Pointeurs typés

### 1.3.1 Présentation

A une variable correspond un emplacement mémoire caractérisé par une adresse et une longueur (par exemple 4 octets consécutifs pour un `long int`). C'est, bien sur, le compilateur qui assure la gestion de la mémoire et affecte à chaque variable un emplacement déterminé. On peut accéder a la valeur de cette adresse grace à l'opérateur unaire & dit opérateur d'adressage.

Un pointeur est une variable d'un type spécial qui pourra contenir l'adresse d'une autre variable. On dit qu'il *pointe* vers cette variable. Celui-ci devra aussi connaître le type de la variable vers laquelle il pointe puisque la taille d'une variable (en octets) dépend de son type. La déclaration d'un pointeur devra donc indiquer le type d'objet vers lequel il pointe (on dit d'un pointeur qu'il est typé). La syntaxe est la suivante:

```
type *identificateur;
```

Par exemple, pour déclarer un pointeur vers un entier on écrira:

```
int* p_entier;
```

ou encore:

```
int *p_entier;
```

On peut réaliser des opérations sur des pointeurs de même type. On peut en particulier affecter à un pointeur un autre pointeur du même type ou l'adresse d'une variable de même type que celle du pointeur.

On accède a la valeur stockée à l'adresse contenue dans un pointeur grace à l'opérateur unaire dit de référencement ou d'indirection: *

Dans l'exemple suivant:

```
int a;
int* p_a;

p_a=&a;
```

`*p_a` et `a` font référence au même emplacement mémoire (et ont donc la même valeur).

Un pointeur peut par ailleurs pointer vers un autre pointeur.

On peut aussi incrémenter un pointeur. Celà revient à augmenter sa valeur de la taille de l'objet pointé et donc à pointer sur l'objet suivant du même type (s'il en existe un!).

La déclaration d'un pointeur alloue un espace mémoire pour le pointeur mais pas pour l'objet pointé. Le pointeur pointe sur n'importe quoi au moment de sa déclaration. Il est conseillé d'initialiser tout pointeur avant son utilisation effective avec la valeur NULL (constante prédéfinie qui vaut 0) ce qui, par convention, indique que le pointeur ne pointe sur rien.

### 1.3.2 Pointeurs et tableaux

La déclaration d'un tableau réserve de la place en mémoire pour les éléments du tableau et fournit une constante de type pointeur qui contient l'adresse du premier élément du tableau. Cette constante est identifiée par l'identificateur donné au tableau (sans crochets). C'est cette constante de type pointeur qui va permettre de manipuler facilement un tableau en particulier pour le passer en paramètre d'une fonction puisqu'on ne passera à la fonction que l'adresse du tableau et non tous ses éléments.

L'exemple suivant:

```
int tab[10];
```

déclare un tableau de 10 éléments. `tab` contient l'adresse du premier élément et donc l'expression:

```
tab == &tab[0]
```

est VRAIE. On a donc de même:

```
*tab == tab[0]
tab[1] == *(tab+1)
tab[k] == *(tab+k)
```

Les deux écritures sont autorisées.

La différence entre un tableau et un pointeur est qu'un tableau est une *constante* non modifiable alors qu'un pointeur est une *variable*.

### 1.3.3 Passage de paramètres à une fonction

On a vu que les paramètres passés à une fonction le sont par *valeur* et ne peuvent pas être modifiés par l'exécution de la fonction. Ceci est très contraignant si l'on souhaite qu'une fonction renvoie plusieurs résultats. Par exemple, si l'on souhaite écrire une fonction permuttant deux entiers, le code suivant qui parait correct ne fonctionnera pas:

```
void permutation(int a, int b)
{
   int c;

   c=a;
   a=b;
   b=c;
}
```

L'appel de la fonction:

```
permutation(x,y);
```

n'aura ainsi aucun effet sur x et sur y.

La solution consiste à passer, pour les paramêtres que l'on souhaite voir modifier par la fonction, non plus les valeurs de ces paramètres mais les valeurs des adresses de ces paramètres.

Les paramètres de la fonction devront donc être des pointeurs. On accèdera aux valeurs propement dites à l'intérieur de la fonction grâce à l'opérateur d'indirection `*`.

Si l'on reprend l'exemple précédent, cela donne:

```
void permutation(int* p_a, int* p_b)
{
   int c;

   c=*p_a;
   *p_a=b;
   *p_b=c;
}
```

Remarque: on aurait pu garder les identificateurs initiaux a et b!

Et l'appel devra se faire en passant en paramètres les adresses des variables à modifier grâce à l'opérateur d'adressage &:

```
permutation(&x,&y);
```

### 1.3.4 Allocation dynamique

La déclaration de variables dans la fonction `main` ou globalement réserve de l'espace en mémoire pour ces variables pour toute la durée de vie du programme. Elle impose par ailleurs de connaître avant le début de l'exécution l'espace nécessaire aux stockage de ces variables et en particulier la dimension des tableaux. Or dans de nombreuses applications le nombre d'éléments d'un tableau peut varier d'une exécution du programme à l'autre.

La bibliothèque `stdlib` fournit des fonctions qui permettent de réserver et de libérer de manière dynamique (à l'exécution) la mémoire.

La fonction qui permet de réserver de la mémoire est `malloc` [1] Sa syntaxe est:

```
void* malloc(unsigned int taille)
```

La fonction malloc réserve une zone de *taille* octets en mémoire et renvoie l'adresse du début de la zone sous forme d'un pointeur non-typé (ou NULL si l'opération n'est pas possible).

En pratique, on a besoin du type d'un pointeur pour pouvoir l'utiliser. On souhaite d'autre part ne pas avoir à préciser la taille de la mémoire en octets surtout s'il s'agit de structures. L'usage consiste donc pour réserver de la mémoire pour une variable d'un type donné à:

- Déclarer un pointeur du type voulu.
- Utiliser la fonction `sizeof(type)` qui renvoie la taille en octets du type passé en paramètre.
- forcer `malloc` à renvoyer un pointeur du type désiré.

Par exemple, pour réserver de la mémoire pour un entier, on écrira:

```
int* entier;

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

Ceci est surtout utilisé pour des tableaux, par exemple pour un tableau de N "complexe" (cf. le paragraphe sur les structures) on écrirai:

```
struct complexe * tabcomp;

tabcomp=(struct complexe*) malloc(N*sizeof(struct complexe));
```

La fonction `free()` permet de libérer la mémoire précédemment réservée. Sa syntaxe est:

```
void free(void* p)
```

1. il existe aussi la fonction `calloc` assez similaire et donc superflu...

Ainsi, dans le second exemple on écrirai:

```
free(tabcomp);
```

## 1.4 Pointeurs génériques

En C, les pointeurs comme les autres variables sont typés. Par exemple un pointeur d'entiers: `int *p` est différent d'un pointeur de réels `float *p` même si chacun d'entre eux contient une adresse en mémoire. Ceci permet au compilateur de connaître le type de la valeur pointée et de la récupérer (c'est l'opération de *déréférencement*).

C ne permet les affectations entre pointeurs que si ceux-ci sont de même type. Pour écrire des fonctions indépendantes d'un type particulier (par exemple une fonction de permutation) le mécanisme de typage peut être contourné en utilisant des pointeurs génériques.

Pour créer un pointeur générique en C, il faut le déclarer de type `void*`.

### 1.4.1 Copie de zones mémoires

L'utilisation de pointeurs génériques ne permet pas d'utiliser l'opérateur de déréférencement * et donc d'accéder au contenu d'une variable. Ceci pose un problème si l'on veut copier des données d'une variable désignée par un pointeur générique vers une autre.

La librairie `string.h` fournit une fonction qui permet de résoudre ce problème: `memcpy()`:

Syntaxe: `void *memcpy(void *pa, void *pb, unsigned int N)`

Copie N octets de l'adresse `pb` vers l'adresse `pa` et retourne `pa`. L'exemple qui suit illustre l'utilisation de cette fonction.

### 1.4.2 Exemple

Si l'on n'utilise pas des pointeurs génériques, il faut écrire autant de versions de la fonction permutation que de types de variables à permutter:

Pour des entiers:

```
void permut_int(int *p_a, int *p_b)
{
  int c;

  c=*p_a;
  *p_a=*p_b;
  *p_b=c;
}
```

et pour des réels:

```
void permut_float(float *p_a, float *p_b)
{
  float c;

  c=*p_a;
  *p_a=*p_b;
  *p_b=c;
}
```

Que l'on utilisera de la manière suivante:

```
int i=2,j=3;
float x=3.4,y=6.5;

permut_int(&i,&j);
permut_float(&x, &y);
```

Pour pouvoir utiliser la même fonction quel que soit le type des données à manipuler il est nécessaire de passer en paramètre la taille des données à traiter qui dépend de leurs types.

La fonction de permutation générique peut s'écrire:

```
void permut(void* p_a, void* p_b, int taille)
{
  void* p_c;

  p_c=malloc(taille);
  memcpy(p_c,p_a,taille); /* *p_c=*p_a n'est pas autorisé */
  memcpy(p_a,p_b,taille);
  memcpy(p_b,p_c,taille);
  free(p_c);
}
```

Que l'on pourra utiliser ainsi:

```
int i=2,j=3;
float x=3.4,y=6.5;

permut(&i,&j, sizeof(i));
permut(&x, &y, sizeof(x));
```

On rapelle que la fonction `sizeof()` renvoie la taille correspondant au type de la variable passée en paramètre.

Remarquez que cette fonction reste valable pour des structures complexes. Par exemple:

```
  struct complexe
  {
    double reel;
    double imag;
  };

  struct complexe cx={1,2}, cy={3,4};

  permut(&cx, &cy, sizeof(cx));
```

## 1.5 Pointeurs de fonctions

Les pointeurs de fonction sont des pointeurs qui, au lieu de pointer vers des données, pointent vers du code exécutable. La déclaration d'un pointeur de fonction ressemble à celle d'une fonction sauf que l'identificateur de la fonction est remplacé par l'identificateur du pointeur précédé d'un astérisque (*) le tout mis entre parenthèses.

*Exemple:*

```
int (* p_fonction) (int x, int y);
```

déclare un pointeur vers une fonction de type entier nécessitant deux paramètres de type entier.

L'intérêt est, par exemple, de pouvoir passer en paramètre d'une fonction, une autre fonction.

*Utilisation:*

```
int resultat;

int calcul(int x, int y)
{
...
```

```
}

p_fonction=calcul;
resultat=p_fonction(3,5); /* Equivalent à resultat=calcul(3,5) */
```

### 1.5.1 Exemple complet

Recherche du minimum d'une fonction monovariable y=f(x) entre 2 bornes (Algorithme plutôt simpliste!):

```
#include <stdio.h>
/*-------------------------------------------------------*/
float carre(float x)
{
  return x*x;
}
/*-------------------------------------------------------*/
float parabole(float x)
{
  return x*x-4*x+2;;
}
/*-------------------------------------------------------*/
float min_fct(float a, float b, float (* pF) (float x))
{
  int i;
  float pas;
  float vmin;
  float valeur;

  pas=(b-a)/100;
  vmin=pF(a);
  for (i=1; i<101; i++)
   {
    valeur=pF(a+i*pas);
    if (vmin > valeur)
      vmin=valeur;
   }
  return vmin;
}
/*-------------------------------------------------------*/
int main()
{
  printf("%f \n",min_fct(-3.0,3.0,carre));
  printf("%f \n",min_fct(-3.0,3.0,parabole));

  return 0;
}
```

## 1.6 Compilation séparée et classes d'allocation de variables

### 1.6.1 Variables locales et globales

*Variables locales ou internes*

Les variables dites locales sont celles qui sont déclarées dans un bloc (séparateurs: { et } ). Elles ne sont visibles (donc utilisables) que dans ce bloc. Leur durée de vie va de l'exécution du début de

bloc jusqu'à la fin du bloc (variables volatiles).

*Variables globales ou externes*
Ce sont les variables déclarées hors de tout bloc. Elles sont visibles à partir de leur définition.

*Exemple*

```
#include <stdio.h>

double x;  /* Variables globales */
int N;

double f1()
{
  int N; /* variable locale qui masque la variable globale de même nom */
  int i; /* autre variable locale */
...
  x=...; /* Utilisation de la variable globale x (déconseillé) */
}

int main()
{
  /* dans le main, x et N sont accessibles mais pas i */
}
```

### 1.6.2 Définition et déclaration

La définition d'une variable effectue une réservation de mémoire.
Ex: `int N`
La déclaration fait référence à une définition. Elle n'effectue pas de réservation en mémoire, la variable doit avoir été définie par ailleurs.
Ex: `extern int N`

### 1.6.3 Variables communes

Un programme C, dès qu'il devient un peu important, est constitué de plusieurs fichiers sources. Le partage des variables entre les différents fichiers nécessite certaines précautions.
Une variable globale commune à plusieurs fichiers doit être *définie* dans un seul fichier et *déclarée* dans tous les fichiers qui doivent y avoir accès.

Ex: fichier1.c: `int N;` fichier2.c: `extern int N;`

### 1.6.4 Classe d'allocations de variables

Une variable est définie par sa classe d'allocation qui peut être: `extern, auto, static, register`. Par défaut (sans précision) les variables sont de classe `auto`. Pour définir une variable dans une autre classe, il faut le préciser en tête de définition.
Le tableau suivant résume les caractéristiques de ces 4 classes.

| Classe | Mémoire | Type |
|---|---|---|
| auto | pile (volatile) | locale |
| static | permanente | locale |
| register | registres | locale |
| extern | permanente | globale |

Une variable locale peut être allouée en mémoire permanente si elle est définie de classe `static`. Elle reste visible uniquement dans le bloc où elle est définie.

Exemple:

```
int f()
{
  static int N=0; /* allouée et initialisée au premier appel de la fonction*/

  ...
  N++; /* compte le nombre d'appels de la fonction */
}
```

La déclaration d'une fonction possède aussi une classe:

- Elle peut être de classe `static`. Celà signifie qu'elle n'est visible (appelable) que dans le fichier où elle est définie.
- Elle peut être de classe `extern`. Celà signifie qu'elle est *définie* dans un autre fichier (seule sa déclaration apparait).

### 1.6.5 Fichiers d'entêtes

Les fichiers d'entêtes d'extension `.h` regroupent les déclarations de types, de fonctions et de variables *exportables* c'est à dire suceptibles d'être utilisées dans plusieurs fichiers sources.

En général, on crée un fichier d'entête `.h` pour chaque fichier source `.c` (excepté pour le fichier contenant la fonction `main`). Le fichier source contient les déclarations des fonctions (entête+code) et des variables globales. Le fichier d'entête contient les définitions des fonctions et variables partageables.

Exemple:

fichier: `complexe.h`

```
struct s_comp
{
  float reel;
  float imag;
}
typedef struct s_comp t_comp

extern t_comp(J);
extern t_comp sommme(t_comp, t_comp);
extern t_comp produit(t_comp, t_comp);
...
```

fichier: `complexe.c`

```
#include "complexe.h"

t_comp J={0,1};

t_comp somme(t_comp a, t_comp b)
{
  t_comp c;

  c.reel=a.reel+b.reel;
  c:imag=a.imag+b.imag;
  return c;
}
...
```

Utilisation: fichier `prog.c`

```
#include "complexe.h"

int main()
{
  t_comp x={3,2},z;

  z=somme(x,J);
  ...
  return 0;
}
```

Compilation: `gcc -Wall -o prog prog.c complexe.c`

# Chapitre 2

# Algorithmes

## 2.1 Notion d'algorithme

Un algorithme est une suite de traitements élémentaires effectués sur des données en vue d'obtenir un résultat en un nombre finis d'opérations.

Traitement élémentaire: traitement pouvant être effectué par un ordinateur. Notion relative. Exemple: le calcul de la racine carrée d'un nombre peut être considéré comme élémentaire en C en utilisant la bibliothèque mathématique. Il ne l'est pas pour un microcontroleur programmé en assembleur.

Exemple: l'algorithme d'Euclide calculant le PGCD de deux entiers:

Soit la division euclidienne de $a$ par $b$, $a = bq + r$. L'algorithme d'Euclide est basé sur le principe que les diviseurs communs de $a$ et $b$ sont les mêmes que ceux de $b$ et $r$. En remplaçant $a$ par $b$ et $b$ par $r$ et en divisant à nouveau, on obtient deux entiers ayant les mêmes diviseurs que les entiers $a$ et $b$ d'origine.

Finalement, on obtient deux entiers divisibles entre eux ($r = 0$) dont le plus petit est le PGCD de $a$ et de $b$.

## 2.2 Représentation des algorithmes

Un programme est la réalisation d'un algorithme. Pour s'affranchir d'un langage de programmation particulier, différentes techniques de représentation sont utilisées.

### 2.2.1 Pseudo langage

Permet de représenter formellement un algorithme indépendamment de l'implémentation (langage). Basé sur les instructions disponibles dans la plupart des langages.

Structures élémentaires:

- Entrées/Sorties: LIRE, ECRIRE
- affectation: $X \leftarrow Y$
- instruction conditionnelle: SI condition ALORS instructions SINON instructions FIN SI
- répétition
  - TANT QUE condition FAIRE instructions FIN TANT QUE
  - POUR i=0 A n FAIRE instructions FIN POUR
  - FAIRE instructions TANT QUE condition

Exemple: calculer la factorielle de N (version itérative):
LIRE N
$F \leftarrow 1$
POUR I=1 A N FAIRE
$F \leftarrow F * I$
FIN POUR
ECRIRE F Il n'existe pas de formalisme universel. Chaque auteur à sa syntaxe particulière.

### 2.2.2 Organigramme

Un organigramme permet de représenter graphiquement un algorithme.
Exemple pour le calcul de la factorielle:

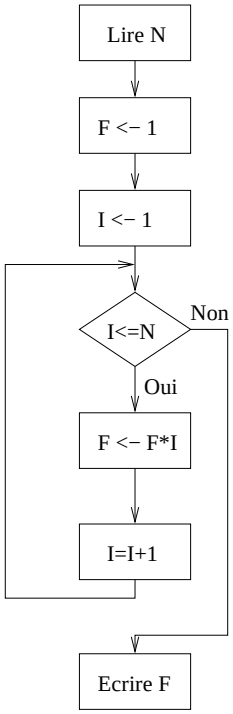


FIG. 2.1 – *Exemple d'organigramme*

En pratique cette représentation devient vite illisible pour des problèmes complexes.

## 2.3 Analyse et complexité d'un algorithme

Il existe souvent plusieurs algorithmes permetant de résoudre un même problème. Exemple: les algorithmes de tri. Le choix du meilleur algorithme implique une analyse de ses performances. En général, le critère le plus important est celui du temps nécessaire à son exécution. Celui ci dépend le plus souvent de la quantité de données à traiter. Par exemple, le temps nécessiare pour trier un ensemble d'objets dépend du nombre d'objets.

L'étude de la complexité d'un algorithme consiste essentiellement à évaluer la dépendance entre le temps d'éxécution et le volume de données [1]. Les résultats s'expriment de manière qualitative: On cherche par exemple à savoir si la complexité croit de manière linéaire ou polynomiale avec le volume $n$ de données.

1. On peut aussi s'intéresser à la quantité de mémoire nécessaire

Par exemple, la recherche d'un élément particulier dans un ensemble de $n$ éléments non triés consiste à examiner chaque élément jusqu'à trouver celui que l'on cherche. Dans le meilleur cas, cela prendra une seule comparaison, dans le pire des cas, il faudra effectuer $n$ comparaisons. En moyenne on aura $n/2$ comparaisons à effectuer.

On s'intéresse en général a la complexité d'un algorithme:

- dans le meilleur cas,
- en moyenne,
- dans le pire des cas.

On exprime la complexité d'un algorithme en ne gardant que le terme variant le plus avec $n$ et en omettant les termes constants. Par exemple, un algorithme nécessitant $100n^3+1000n^2+5000n+10000$ instructions élémentaires sera dit de complexité $O(n^3)$. Un algorithme ne dépendant pas du volume de données sera dit de complexité $O(1)$.

En général un algorithme de complexité $O(n \log n)$ sera plus efficace qu'un algorithme de complexité $O(n^2)$ mais ceci peut n'être vrai que si $n$ est assez grand. En effet la complexité mesure le comportement asymptotique d'un algorithme quand $n$ tend vers l'infini.

Le tableau suivant donne quelques exemples de complexité que l'on retrouve couramment. Soit E un ensemble de $n$ données:

| Algorithme | Complexité |
|---|---|
| Accés au 1er élément de E | $O(1)$ |
| Recherche dichotomique (E trié) | $O(\log n)$ |
| Parcours de E | $O(n)$ |
| Tri rapide | $O(n \log n)$ |
| Parcours de E pour chaque élément d'un ensemble F de même dimension | $O(n^2)$ |
| Génération de tous les sous-ensembles de E | $O(2^n)$ |
| Génération de toutes les permutations de E | $O(n!)$ |

Le tableau suivant donne les ordres de grandeurs des différentes complexités en fonction de la taille de l'ensemble de données:

| Complexité | n=1 | n=4 | n=16 | n=64 | n=256 | n=4096 |
|---|---|---|---|---|---|---|
| $O(1)$ | 1 | 1 | 1 | 1 | 1 | 1 |
| $O(\log n)$ | 0 | 2 | 4 | 6 | 8 | 12 |
| $O(n)$ | 1 | 4 | 16 | 64 | 256 | 4096 |
| $O(n \log n)$ | 0 | 8 | 64 | 384 | 2048 | 49152 |
| $O(n^2)$ | 1 | 16 | 256 | 4096 | 65536 | 16777216 |
| $O(2^n)$ | 2 | 16 | 65536 | 18446744073709551616 | | |
| $O(n!)$ | 1 | 24 | 20922789888000 | | | |

En résumé, la complexité d'un algorithme est la courbe de croissance des ressources qu'il requiert (essentiellement le temps) par rapport au volume des données qu'il traite.

## 2.4 Récursivité

Une fonction récursive est une fonction qui s'appelle elle-même.

Exemple: calcul de la factorielle d'un nombre.

Définition itérative: $n! = F(n) = n(n-1)(n-2)...2 \cdot 1$

Ce qui donne en C:

```
long int fact(int N)
{
  long int F=1;
  int i;

  for (i=1; i<=N; i++)
```

```
    F=F*i;

  return F;
}
```

Définition récursive: $F(n) = nF(n-1), F(0) = 1$. Soit en C:

```
long int factr(int N)
{
  if (N==0)
    return 1;
  else
    return N*factr(N-1);
}
```

Si, en général, la version récursive d'un algorithme est plus élégante à écrire que sa version itérative[2], elle est cependant plus difficile à mettre au point et moins efficace en temps calcul.

Pour qu'un algorithme récursif fonctionne:

- il doit exister un cas terminal que l'on est sur de rencontrer.
- Un mécanisme de pile est par ailleurs nécessaire. Il est naturel en C dans le passage de paramètres à une fonction.

## 2.5 Algorithmes de tri

Il s'agit d'un problème classique (et utile) de l'algorithmique. On considère un ensemble d'éléments possédant une relation d'ordre total (Exemple: entiers, réels, caractères).

On cherche à ordonner cet ensemble dans l'ordre croissant (ou décroissant).

### 2.5.1 Tri par sélection

#### Principe

Soit un ensemble de n éléments indicés de 0 à n-1. On suppose que les m premiers éléments (0 a m-1) sont triés.

On cherche la position k du plus petit élément parmi les éléments m à n-1. On le permutte avec l'élément m. L'ensemble se trouve donc trié de l'indice 0 à l'indice m.

On parcourt ainsi l'ensemble de l'élément m=0 à l'élément n-2.

#### Illustration

En gras, les éléments déjà triés, en italique, les éléments à permutter.

| | | | | | |
|---|---|---|---|---|---|
| *18* | 10 | 3 | 25 | 9 | *2* |
| **2** | *10* | *3* | 25 | 9 | 18 |
| **2** | **3** | *10* | 25 | *9* | 18 |
| **2** | **3** | **9** | *25* | *10* | 18 |
| **2** | **3** | **9** | **10** | *25* | *18* |
| **2** | **3** | **9** | **10** | **18** | **25** |

2. On peut toujours transformer un algorithme récursif en algorithme itératif

**Algorithme**

Soit à trier un tableau de N éléments (entiers), t[0] à t[N-1]

POUR m=0 a N-2 FAIRE
$k \leftarrow p$ (indice du plus petit élément entre t[m] et t[N-1])
SI k différent de m ALORS permutter t[k] et t[m] FIN SI
FIN POUR

**Programmation en C**

Le programme peut se décomposer en trois fonctions:
- une fonction de calcul de l'indice du plus petit élément entre t[m] et t[N-1],
- une fonction de permutation,
- La fonction de tri proprement dite.

```
/* ------------------------ indice du plus grand élément entre m et n-1 */
int indice_min(int t[], int m, int n)
{
  int i;
  int imin;

  imin=m;
  for (i=m+1; i<n; i++)
   if (t[i]<t[imin])
    imin=i;

  return imin;
}
/* ------------------------------------------- permutation de 2 entiers */
void permutte(int *a, int *b)
{
  int c;

  c=*a;
  *a=*b;
  *b=c;
}
/* ------------------------------------------------- tri par selection */
void tri_selection(int t[], int n)
{
  int m;
  int p;
  for (m=0; m<N-1; m++)
   {
    p=indice_min(t,m,N);
    if (p!=m)
      permutte(&t[p],&t[m]);
   }
}
```

**Analyse**

Cet algorithme nécessite $n^2/2$ comparaisons et $n$ permutations. Pour un nombre d'éléments donné, il effectue le même nombre d'opérations que les éléments soient pratiquements déjà triés ou totalement en désordre. Sa complexité est en $O(n^2)$.

### 2.5.2 Tri à bulle

**Principe**

Le principe consiste à parcourir les éléments de l'ensemble de i=0 à n-1 en permuttant les éléments consécutifs non ordonnés.

L'élément le plus grand se trouve alors en bonne position. On recommence la procédure pour l'ensemble de i=0 à n-2 sauf si aucune permutation n'a été nécessaire à l'étape précédente. Les éléments les plus grands se déplacent ainsi comme des bulles vers la droite du tableau.

### 2.5.3 Illustration

En italique, les deux éléments à comparer, en gras les éléments en bonne place.

| | | | | | |
|---|---|---|---|---|---|
| *18* | *10* | 3 | 25 | 9 | 2 |
| 10 | *18* | *3* | 25 | 9 | 2 |
| 10 | 3 | *18* | *25* | 9 | 2 |
| 10 | 3 | 18 | *25* | *9* | 2 |
| 10 | 3 | 18 | 9 | *25* | *2* |
| *10* | *3* | 18 | 9 | 2 | **25** |
| 3 | *10* | *18* | 9 | 2 | **25** |
| 3 | 10 | *18* | *9* | 2 | **25** |
| 3 | 10 | 9 | *18* | *2* | **25** |
| *3* | *10* | 9 | 2 | **18** | **25** |
| 3 | *10* | *9* | 2 | **18** | **25** |
| 3 | 9 | *10* | *2* | **18** | **25** |
| *3* | *9* | 2 | **10** | **18** | **25** |
| 3 | *9* | *2* | **10** | **18** | **25** |
| *3* | *2* | **9** | **10** | **18** | **25** |
| **2** | **3** | **9** | **10** | **18** | **25** |

**Algorithme**

$k \leftarrow N - 1$
FAIRE
  POUR i=0 a J-1 FAIRE
    SI t[i] > t[i+1] ALORS
      permutter t[i] et t[i+1]
      permutation=VRAI
    FIN SI
  FIN POUR
TANT QUE permutation=VRAI

**Analyse**

Dans le pire des cas, le nombre de comparaisons et le nombre de permutations à effectuer sont de $n^2/2$. Dans le meilleur des cas (ensemble déjà trié), le nombre de comparaisons est de $n - 1$ et l'algorithme est donc de complexité linéaire.

### 2.5.4 Tri par insertion

**Principe**

On prend 2 éléments et on les met dans l'ordre. On prend un troisième élément qu'on insère dans les 2 éléments déjà triés, etc..

Un élément m va être inséré dans l'ensemble déjà trié des éléments 0 à m-1. Ce qui donnera m+1 éléments triés (0 à m).

L'insertion consiste a chercher l'élément de valeur immédiatement supérieure ou égale à celle de l'élément à insérer. Soit k l'indice de cet élément, on décale les éléments k à m-1 vers k+1 à m et l'on place l'élément à insérer en position k.

**Illustration**

En gras, l'élément à insérer dans la partie triée du tableau.

| | | | | | |
|---|---|---|---|---|---|
| 18 | **10** | 3 | 25 | 9 | 2 |
| 10 | 18 | **3** | 25 | 9 | 2 |
| 3 | 10 | 18 | **25** | 9 | 2 |
| 3 | 10 | 18 | 25 | **9** | 2 |
| 3 | 9 | 10 | 18 | 25 | **2** |
| 2 | 3 | 9 | 10 | 18 | 25 |

**Analyse**

Dans le pire des cas, le nombre de comparaisons et de $n^2/2$. Dans le meilleur des cas il est de $N$. L'algorithme est de complexité $O(N^2)$ mais il est plus efficace que les deux précédents si le tableau est en partie trié.

## 2.6 Algorithmes de recherche dans un ensemble

Un problème courant est la recherche d'un élément particulier dans un ensemble. La solution la plus simple consiste à parcourir l'ensemble des éléments jusqu'à trouver celui que l'on cherche.

Dans le cas ou l'ensemble de recherche est trié, une solution plus efficace consiste à faire une recherche dichotomique.

On peut aussi utiliser des tables de *hachage*. Le principe consiste à associer à chaque élément de l'ensemble une clé calculé, cette clé permettant un accès direct à un élément. Le calcul de la clé pour l'élément recherché permet d'y accéder directement.

On crée en général un index contenant l'information sur lequel se fera la recherche.

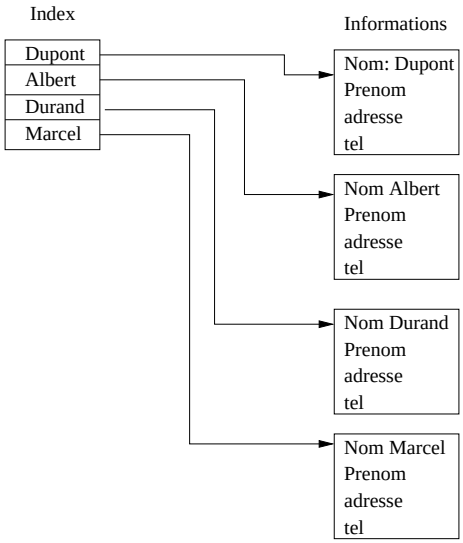


FIG. 2.2 – *Création d'un index*

### 2.6.1 Recherche séquentielle

La recherche séquentielle consiste à parcourir chaque élément de l'ensemble avec celui recherché. En général cette recherche se fait sur une information particulière de l'élément ou dans une table d'index. Cette méthode fonctionne que l'ensemble soit trié ou non.

Dans le meilleur cas, cela prendra une seule comparaison, dans le pire des cas, il faudra effectuer $n$ comparaisons. En moyenne on aura $n/2$ comparaisons à effectuer.

La complexité de l'algorithme est donc en $O(n)$.

### 2.6.2 Recherche dichotomique

Dans le cas ou l'ensemble est trié par rapport à l'index sur lequel doit se faire la recherche (par exemple le nom pour un annuaire), un algorithme plus efficace que la simple recherche séquentielle peut être utilisé.

On compare l'élément recherché avec celui du milieu de l'ensemble. Si c'est le même, la recherche est terminé sinon c'est qu'il est plus grand ou plus petit et on recommence en ne gardant que la moitié de l'ensemble. On divise ainsi l'ensemble de recherche par deux à chaque itération. La complexité de l'algorithme est donc en $O(\log n)$.

### 2.6.3 Tables de hachage

Le hachage consiste à calculer une clé $h(x)$ (nombre entier) pour chaque élément $x$. $h(x)$ contient l'endroit ou l'on trouve $x$ dans l'ensemble. Si l'application est injective (une clé unique par élément), il suffit de calculer la clé de l'élément recherché et, s'il existe, on y accède directement.

Pour des chaînes de caractères $S = s_0 s_1 s_2 ... s_{l-1}$, on utilise par exemple la fonction:

$$h(S) = (s_0 B^{l-1} + s_1 B^{l-2} + ... + s_{l-2} B + s_{l-1}) \bmod N$$

Ou $N$ est la taille de la table de hachage (on choisi un nombre premier) et $B$ une puissance de 2. $s_i$ est le code ASCII du caractère d'indice $i$ de la chaîne.

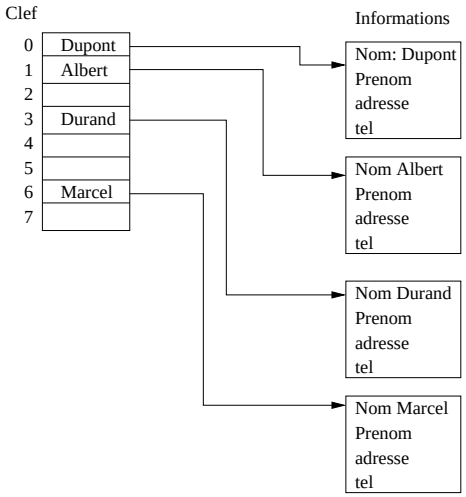


FIG. 2.3 – *Table de hachage*

En pratique, l'unicité de la clé pour une entrée est rarement réalisable. On peut donc avoir plusieurs éléments ayant la même clé. On parle de *collision*. Une méthode simple de gérer les collisions et de les lister dans une table parallèle.

# Chapitre 3

# Structures de données

## 3.1 Types de données abstraits (TDA)

Un TDA est un ensemble de données organisé de sorte que les spécifications des objets et des opérations sur ces objets (interface) soient séparées de la représentation interne des objets et de de la mise en oeuvre des opérations.

Exemple de TDA: le type entier muni des opérations $+, -, *, \%, /, >, <, <=, >=, ==$ est un TDA.

Une mise en oeuvre d'un TDA est la structure de données particulière et la définition des opérations primitives dans un langage particulier.

Les avantages des TDA sont:

- prise en compte de types complexes.
- séparation des services et du codage. L'utilisateur d'un TDA n'a pas besoin de connaître les détails du codage.
- écriture de programmes modulaires.

## 3.2 Piles

Une pile est un ensemble de données auquelles on accède dans l'ordre inverse ou on les a insérées (Last Input, First Ouput).

On ne peut accéder qu'au sommet de la pile. Les opérations (primitives) sur une pile sont les suivantes:

- empiler(élément): rajoute un élément sur la pile.
- dépiler: renvoie la valeur de l'élément au sommet de la pile et le supprime.
- vide: renvoie VRAI si et seulement si la pile est vide

En informatique, une pile sert, par exemple, pour gérer les appels et retours de fonctions. Si au cours de l'exécution d'une fonction A, la machine doit exécuter une fonction B, elle place au sommet de la pile d'exécution l'adresse à laquelle la fonction A a été interrompue puis exécute les instructions de la fonction B avant de reprendre l'exécution de A à l'adresse contenue au sommet de la pile d'exécution. Ceci permet d'imbriquer des fonctions sans limitation (théorique) de profondeur.

Les piles sont aussi utilisées pour passer des paramètres à une fonction ou pour stocker des variables locales.

### 3.2.1 Exemple d'utilisation: calcul en notation polonaise inversée (notation *postfixé*)

Une formule mathématique peut être représentée par une liste de *lexèmes* un lexème pouvant représenter:

- une valeur

- un opérateur binaire
- un opérateur unaire
- une parenthèse (ouvrante ou fermante)

Exemple:

$$\frac{4+2\sqrt{16}}{6}$$

peut être représentée par la liste:

```
{(, 4, +, 2, *, sqrt, (, 16, ), ), /, 6}
```

on parle de forme *infixé*, les opérateurs binaires étant placés entre leurs opérandes.

Il existe aussi une forme *préfixée* ou chaque opérateur précède ses opérandes:

```
{/, +, 4, *, 2, sqrt, 16, 6}
```

et une forme *postfixée* ou chaque opérateur vient après son dernier argument:

```
{4, 2, 16, sqrt, *, +, 6, /}
```

L'intérêt de cette dernière forme est qu'elle ne nécessite pas l'utilisation de parenthèses.

L'évaluation de la formule postfixée peut être réalisée de manière simple en utilisant une pile et en suivant les rêgles suivantes:

On lit les lexèmes dans l'ordre de la liste:

- si l'élément de la liste est une valeur, la placer sur la pile.
- si l'élément de la liste est un opérateur unaire, appliquer l'opérateur sur le sommet de la pile.
- si l'élément de la liste est un opérateur binaire, appliquer l'opérateur sur les deux éléments de tête de la pile.

Ce qui peut se traduire par l'algorithme suivant utilisant les opérations primitives sur une pile:

```
pour chaque élément de la liste:
 si élément=valeur
    empiler(valeur)
 si élément= opérateur unaire
    operation=élément
    résultat=opération(depiler)
    empiler(résultat)
 si élément= opérateur binaire
    opération=élément
    opérande1=dépiler
    opérande2=dépiler
    résultat=opération(opérande1, opérande2)
    empiler(résultat)
```

A la fin, le sommet de la pile contient le résultat de la formule.

| élément | pile (sommet à droite) |
|---|---|
| 4 | 4 |
| 2 | 4 2 |
| 16 | 4 2 16 |
| sqrt | 4 2 4 |
| * | 4 8 |
| + | 12 |
| 6 | 12 6 |
| / | 2 |

### 3.2.2 Implémentation des piles

La manière la plus simple d'implémenter une pile consiste à utiliser un tableau. La pile peut être caractérisée par l'indice dans le tableau de son sommet et par une *sentinelle* permettant de gérer les débordements de la pile.

## 3.3 Files

Dans une file, les éléments sont ajoutés en queue et supprimés en tête (First Input First Ouput). Ils sont donc traités dans l'ordre d'arrivée. On retrouve les opérations suivantes comme pour les piles:

- ajouter: rajoute un élément en queue de file.
- supprimer: renvoie la valeur de l'élément en tête de la file et le supprime.
- vide: renvoie VRAI si et seulement si la file est vide

Les files sont utilisées, par exemple, dans les applications pilotées par évènements. Les évènements (clavier, souris, ...) sont stockés dans une file par ordre d'arrivée avant d'être traités.

On les utilise aussi pour simuler des files d'attente.

## 3.4 Listes

### 3.4.1 Généralités

Une liste est un ensemble fini d'éléments notée $L = e_1, e_2, ..., e_n$ où $e_1$ est le premier élément, $e_2$ le deuxième, etc... Lorsque n=0 on dit que la liste est vide.

Les listes servent à gérer un ensemble de données, un peu comme les tableaux. Elles sont cependant plus efficaces pour réaliser des opérations comme l'insertion et la suppression d'éléments. Elles utilisent par ailleurs l'allocation dynamique de mémoire et peuvent avoir une taille qui varie pendant l'éxécution. L'allocation (ou la libération) se fait élément par élément [1].

Les opérations sur une liste peuvent être:

- Créer une liste
- Supprimer une liste
- Rechercher un élément particulier
- Insérer un élément (en début, en fin ou au milieu)
- Supprimer un élément particulier
- Permuter deux éléments
- Concaténer deux listes
- ...

Les listes peuvent par ailleurs être:

- simplement chaînées
- doublement chaînées
- circulaires (chaînage simple ou double)

### 3.4.2 Listes simplement chaînées

Une liste simplement chaînée est composée d'éléments distincts liés par un simple pointeur. Chaque élément d'une liste simplement chaînée est formée de deux parties:

- un champ contenant la donnée (ou un pointeur vers celle-ci)
- un pointeur vers l'élément suivant de la liste.

Le premier élément d'une liste est sa *tête*, le dernier sa *queue*. Le pointeur du dernier élément est initialisé à une valeur sentinelle, par exemple la valeur `NULL` en C.

Pour accéder à un élément d'une liste simplement chaînée, on part de la tête et on passe d'un élément à l'autre à l'aide du pointeur *suivant* associé à chaque élément.

En pratique, les éléments étant crées par allocation dynamique, ne sont pas contigus en mémoire contrairement à un tableau. La suppression d'un élément sans précaution ne permet plus d'accéder

1. Un tableau peut aussi être défini dynamiquement mais pour modifier sa taille, il faut en créer un nouveau, transférer les données puis supprimer l'ancien.

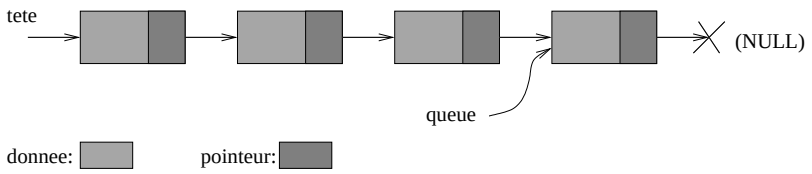


FIG. 3.1 – *Liste simplement chaînée*

aux éléments suivants. D'autre part, une liste simplement chainée ne peut être parcourue que dans un sens (de la tête vers la queue).

Exemple d'implémentation sous forme d'une structure en C:

```
struct s_element
{
  int donnee;
  struct s_element* suivant;
};
typedef struct s_element t_element;
```

### 3.4.3 Listes doublement chaînées

Les listes doublement chaînées sont constituées d'éléments comportant trois champs:

- un champ contenant la donnée (ou un pointeur vers celle-ci)
- un pointeur vers l'élément suivant de la liste.
- un pointeur vers l'élément précédent de la liste.

Elles peuvent donc être parcourues dans les deux sens.

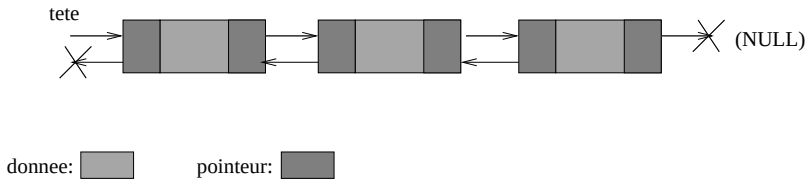


FIG. 3.2 – *Liste doublement chaînée*

### 3.4.4 Listes circulaires

Une liste circulaire peut être simplement ou doublement chaînée. Sa particularité est de ne pas comporter de queue. Le dernier élément de la liste pointe vers le premier. Un élément possède donc toujours un *suivant*.

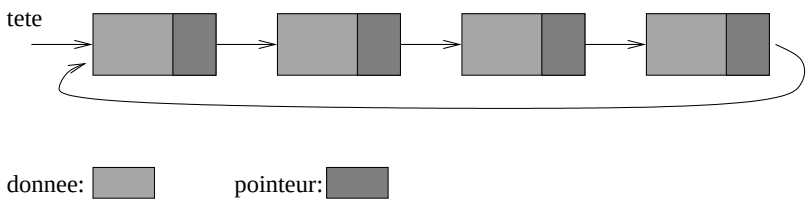


FIG. 3.3 – *Liste circulaire simplement chaînée*

### 3.4.5 Opérations sur une liste

On ne décrira dans ce paragraphe que quelques opérations sur les listes simplement chaînées.

#### Insertion d'un élément

L'insertion d'un élément dans une liste peut se faire:
- en tête de liste
- en queue de liste
- n'importe ou (à une position fixé par un pointeur dit *courant*)

L'exemple choisi est celui de l'insertion *n'importe où* (après l'élément référencé par le pointeur *courant*):

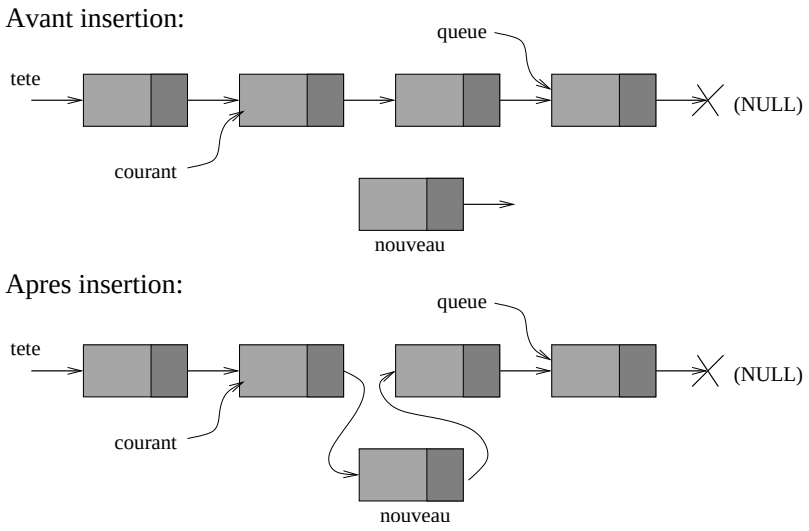


FIG. 3.4 – *Insertion*

Les opérations à effectuer sont (dans l'ordre!):
- allouer de la mémoire pour le nouvel élément
- copier les données
- faire pointer le nouvel élément vers l'élément suivant de celui pointé par *courant* (vers NULL s'il n'y a pas de suivant)
- faire pointer l'élément pointé par courant vers le nouvel élément

Le cas ou la liste est vide (tete égal à NULL) doit être traité à part.

Exemple d'implémentation de la fonction d'insertion:

```
void insertion(t_element** tete, t_element* courant, int data)
{
  t_element* nouveau;

  nouveau=(t_element*) malloc(sizeof(t_element));
  nouveau->donnee=data;

  if (courant!=NULL)
   {
    nouveau->suivant=courant->suivant;
    courant->suivant=nouveau;
```

```
   }
  else   /* insertion en tete ou liste vide */
   {
    nouveau->suivant=*tete;
    *tete=nouveau;
   }
}
```

**Suppression d'un élément**

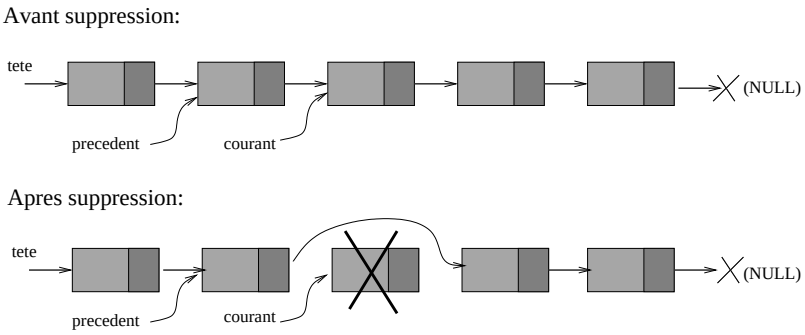


FIG. 3.5 – *Suppression*

Dans le cas d'un liste simplement chaînée, la fonction de suppression demande un peu de réflexion pour être implémenté.

Par exemple, une fonction permettant de supprimer l'élément pointé par *courant* pourrait avoir cette interface:

```
void suppression(t_element** tete, t_element* courant)
```

Le passage du pointeur de tête par adresse est nécessaire pour pouvoir le modifier si la suppression rend la liste vide ou si l'élément à supprimer était le premier de la liste.

On doit d'abord distinguer s'il s'agit du 1er élément de la liste ou pas:

```
if (courant!=*tete) /* pas le premier element */
```

Dans ce cas il faut chercher le prédécesseur de *courant*. La liste étant simplement chaînée, on n'a pas d'autre solution que de faire un parcours depuis la tête jusqu'à trouver le précédent de *courant*.

```
precedent=*tete;
while (precedent->suivant!=courant)
  precedent=precedent->suivant;
```

On peut alors récupérer le lien contenu dans le champ *suivant* de l'élément à supprimer:

```
precedent->suivant=courant->suivant;
```

Dans le cas ou c'est le premier élément que l'on supprime, on modifie simplement le pointeur de tête:

```
else /* suppression du 1er element */
  *tete=courant->suivant;
```

Enfin, dans tous les cas on libère la mémoire:

```
free(courant);
```

Noter que dans cette implémentation, le pointeur courant devient un pointeur *pendant* au retour de la fonction. Ce problème doit être traité au niveau de l'appel de la fonction ou en modifiant un peu l'implémentation de la fonction (préférable!).

### 3.4.6 Exemple d'utilisation

Une fois que les primitives de manipulation de la liste ont été implémentées, l'utilisation est relativement simple.

L'exemple suivant utilise les deux primitives vues précédemment (insertion et suppression).

```
int main()
{
  t_element* tete=NULL;
  t_element* courant=NULL;

/* insertion d'un premier element: liste: 1 */
  insertion(&tete,courant,1);

/* deuxieme apres le premier: liste: 1 2  */
  courant=tete;
  insertion(&tete,courant,2);

/* courant pointe sur 2. liste: 1 2 3 */
  courant=courant->suivant;
  insertion(&tete,courant,3);

/* courant pointe sur 2. liste: 1 2 4 3 */
  insertion(&tete,courant,4);

/* insertion en tete. liste: 5 1 2 4 3 */
  insertion(&tete,NULL,5);

/* suppression de l'element 1. liste: 5 2 4 3 */
  courant=tete->suivant;
  suppression(&tete,courant);

/* suppression de la tete. liste: 2 4 3 */
  courant=tete;
  suppression(&tete,courant);

  return 0;
}
```

# Chapitre 4

# Arbres binaires

## 4.1 Introduction

Un arbre est une structure composée de noeuds et de feuilles (noeuds terminaux) reliés par des branches. On le représente généralement en mettant la racine en haut et les feuilles en bas (contrairement à un arbre réel).

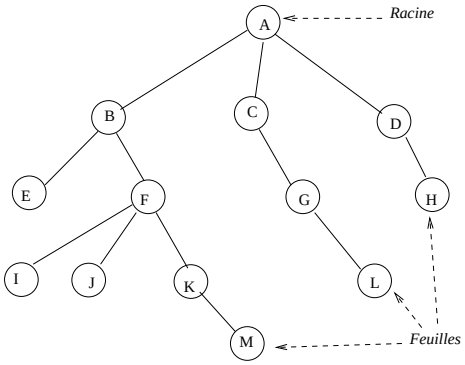


FIG. 4.1 – *Exemple d'arbre*

- Le noeud A est la racine de l'arbre.
- Les noeuds E, I, J, M, L et H sont des feuilles.
- Les noeuds B, C, D, F, G et K sont des noeuds intermédiaires.
- Si une branche relie un noeud $n_i$ à un noeud $n_j$ situé plus bas, on dit que $n_i$ est un ancêtre de $n_j$.
- Dans un arbre, un noeud n'a qu'un seul père (ancêtre direct).
- Un noeud peut contenir une ou plusieurs valeurs.
- La hauteur (ou profondeur) d'un noeud est la longueur du chemin qui le lie à la racine.

## 4.2 Arbres binaires

Un arbre binaire est un arbre tel que les noeuds ont au plus deux fils (gauche et droit).

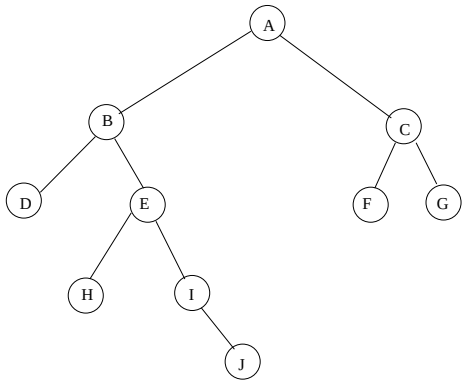


FIG. 4.2 – *Exemple d'arbre binaire*

## 4.3 Arbres binaires de recherche

Un arbre binaire de recherche est un arbre binaire qui possède la propriété fondamentale suivante:

- tous les noeuds du sous-arbre de gauche d'un noeud de l'arbre ont une valeur inférieure ou égale à la sienne.
- tous les noeuds du sous-arbre de droite d'un noeud de l'arbre ont une valeur supérieure ou égale à la sienne.

Exemple:

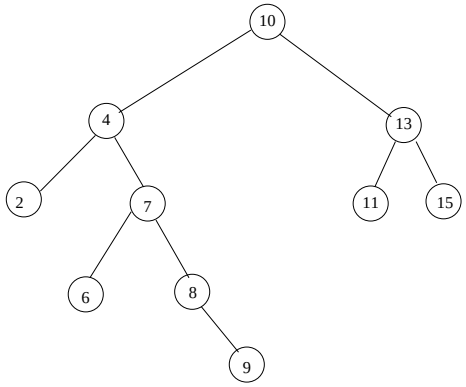


FIG. 4.3 – *Arbre binaire de recherche*

### 4.3.1 Recherche dans l'arbre

Un arbre binaire de recherche est fait pour faciliter la recherche d'informations.
La recherche d'un noeud particulier de l'arbre peut être définie simplement de manière récursive:

Soit un sous-arbre de racine $n_i$,

- si la valeur recherchée est celle de la racine $n_i$, alors la recherche est terminée. On à trouvé le noeud recherché.
- sinon, si $n_i$ est une feuille (pas de fils) alors la recherche est infructueuse et l'algorithme se termine.
- si la valeur recherchée est plus grande que celle de la racine alors on explore le sous-arbre de droite c'est à dire que l'on remplace $n_i$ par son noeud fils de droite et que l'on relance la procédure de recherche à partir de cette nouvelle racine.

– de la même manière, si la valeur recherchée est plus petite que la valeur de $n_i$, on remplace $n_i$ par son noeud fils de gauche avant de relancer la procédure..

Si l'arbre est équilibré chaque itération divise par 2 le nombre de noeuds candidats. La complexité est donc en $O(\log_2 n)$ si $n$ est le nombre de noeuds de l'arbre.

### 4.3.2 Ajout d'un élément

Pour conserver les propriétés d'un arbre binaire de recherche nécessite de, l'ajout d'un nouvel élément ne peut pas se faire n'importe comment.

L'algorithme récursif d'ajout d'un élément peut s'exprimer ainsi:

- soit $x$ la valeur de l'élément à insérer.
- soit $v$ la valeur du noeud racine $n_i$ d'un sous-arbre.
  - si $n_i$ n'existe pas, le créer avec la valeur $x$. fin.
  - sinon
    - si $x$ est plus grand que $v$,
      - remplacer $n_i$ par son fils droit.
      - recommencer l'algorithme à partir de la nouvelle racine.
    - sinon
      - remplacer $n_i$ par son fils gauche.
      - recommencer l'algorithme à partir de la nouvelle racine.

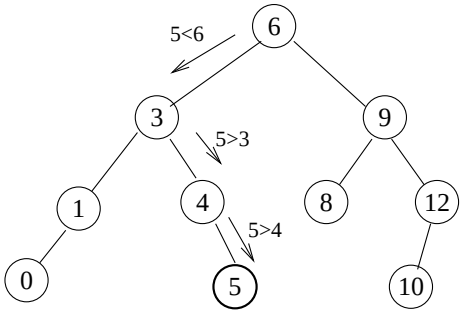


FIG. 4.4 – *Ajout de 5 dans l'arbre*

### 4.3.3 Implémentation

En langage C, un noeud d'un arbre binaire peut être représenté par une structure contenant un champ donnée et deux pointeurs vers les noeuds fils:

```
struct s_arbre
{
  int valeur;
  struct s_arbre * gauche;
  struct s_arbre * droit;
};
typedef struct s_arbre t_arbre;
```

La fonction d'insertion qui permet d'ajouter un élément dans l'arbre et donc de le créer de manière à ce qu'il respecte les propriétés d'un arbre binaire de recherche peut s'écrire ainsi:

```
void insertion(t_arbre ** noeud, int v)
```

```
{
  if (*noeud==NULL) /* si le noeud n'existe pas, on le crée */
    {
      *noeud=(t_arbre*) malloc(sizeof(t_arbre));
      (*noeud)->valeur=v;
      (*noeud)->gauche=NULL;
      (*noeud)->droit=NULL;
    }
  else
    {
      if (v>(*noeud)->valeur)
        insertion(&(*noeud)->droit,v);  /* aller a droite */
      else
        insertion(&(*noeud)->gauche,v); /* aller a gauche */
    }
}
```

On peut noter par ailleurs que l'arbre qui sera construit dépendra de l'ordre dans lequel seront insérées les différentes valeurs. En particulier, si les valeurs sont insérées dans un ordre croissnt on obtiendra un arbre complètement déséquilibré, chaque noeud n'ayant qu'un fils droit (gauche si les valeurs sont dans un ordre décroissant).