ENSMA

Département I & A (Informatique & Automatique)

Cours

METHODES AVANCEES DE PROGRAMMATION

Chapitre

STRUCTURES DE DONNEES DYNAMIQUES & LANGAGE C

Adresse sur le réseau interne de l'ENSMA :

\\S-applis-ens\DATAPROFS\Informatique\guittet\A3\cours\Sdd.doc

L. GUITTET

email : guittet@ensma.fr

09/2010

Table des matières

1 Introduction

1.1 Pourquoi modéliser des données ?

Programme = algorithme + données

- Pour réaliser un « bon » programme (i.e. rapide et faible en mémoire), le choix de la structure de données est aussi important que celui de l'algorithme.
- La structure de données doit modéliser au mieux les informations à traiter pour en faciliter le traitement.
- L'utilisation de **structures de données abstraites** (S.D.A.) associées à des fonctions de manipulation facilite la conception d'un algorithme.

1.2 Pourquoi des structures de données dynamiques ?

- La **spécification** d'une S.D.A. permet de décrire des objets au niveau fonctionnel / logique tout en cachant leur implémentation.
- L'implémentation (ou réalisation physique) peut-être **statique** (taille fixée) ou **dynamique** (taille variable) pour optimiser certaines fonctions (par ex : insérer une donnée)
- Pour pouvoir faire varier la taille des données, le langage doit permettre la création / destruction de zone mémoire : les **pointeurs**.

1.3 Quel langage ?

- Le **langage C** est très utilisé, possède de nombreux environnements de développement (MS Visual C++) et est à la base de C++ et JAVA.

⚠ Il oblige le concepteur à maîtriser les **allocations dynamiques** de mémoire.

1.4 Quel programme (du cours☺) ?

- Nous allons d'abord nous familiariser avec le langage C
- Nous verrons dans le chapitre *Structure de Donnée Abstraite* comment spécifier puis réaliser une SDA grâce à un exemple simple : la **liste**
- Nous étudierons alors au chapitre *Structures Classiques* les structures suivantes :
  - Linéaires (piles, files, listes ordonnées, chaînées, ensemble, dictionnaire)
  - Hiérarchiques (arbres binaires, arbres de recherche, arbre n-aires)
  - Graphes (notions générales)

Une bibliographie sur le WEB est en dernière page du polycopié ainsi que la table des figures.

2.Le langage C

2.1 Introduction

C est un langage impératif, normalisé et à typage fort, mais dangereux ! Sa syntaxe très souple permet l'écriture de programmes corrects du point de vue du langage mais dont l'exécution ne fourni pas le résultat attendu.

⚠ Par exemple, if (a=0) printf("oui") ; n'affichera jamais oui !!!

Nous aborderons le langage au travers d'exemples de programmes commentés. Chacun d'eux montre des notions nouvelles listées en fin de paragraphe.

2.2 Bonjour

Le programme suivant affiche « bonjour » à l'écran et passe à la ligne.

Programme	Explications
`#include <stdio.h>`	Insertion du fichier bibliothèque stdio. Il contient les E/S standard (ici printf)
`/*fin des include*/`	Le texte entre /* et */ est un commentaire
`int main(void)`	Entête de la fonction principale. Elle n'a aucun paramètre d'entrée (void) et rend un entier (int). Son nom doit être « main »
`{`	Corps du programme entre { et }
`printf("bonjour\n");`	Ecriture de « bonjour » suivi d'un passage à la ligne « \n » à l'écran. Le ; est séparateur d'instructions
`return 0;`	Retour de la valeur 0 au système (tout est OK)
`}`	

Notions vues : include, bibliothèque stdio, commentaires, main, fonction, paramètre void, printf,\n, valeur de retour, int, {}, ;

Ce programme, édité et sauvé sous forme de fichier texte « bonjour.c » nécessite certaines manipulations pour voir le résultat d'exécution. Deux moyens s'offrent à nous :

2.2.1 Avec le compilateur « gcc » de cygwin

Cygwin est un logiciel installé sur Windows qui simule le système d'exploitation « Linux ».
Le texte doit être compilé dans la fenêtre cygwin par la commande :

```
> gcc bonjour.c
```

Ceci génère un fichier « a.exe » qui est l'équivalent exécutable de bonjour.c

```
>a.exe
bonjour
```

La commande suivante permet de nommer différemment l'exécutable (au lieu de a.exe)

```
> gcc bonjour.c –o bonjour
```

La production de l'exécutable se réalise en 2 étapes :

1. Compilation : source.c -------> objet.o
2. Edition de liens : objet.o + bibliothèques --------> exécutable
   Dans notre cas, nous avons : bonjour.c ------> bonjour.o
   bonjour.o + stdiolib.a -------> bonjour (ou a.exe)

L'aide de gcc est fournie par l'option suivante :

```
>gcc --help
```

2.2.2 Avec l'environnement « Visual C++ 2008 Express Edition »

La **Figure 1** montre la fenêtre principale de l'environnement. Le programme est édité dans la zone de droite (onglet « bonjour.c »), et le compte-rendu de fabrication de l'exécutable est en bas (fenêtre « sortie »).
Le fichier appartient à un projet appelé « bonjour » (en gras dans la fenêtre de gauche), qu'il faut créer avant par la commande « Fichier/Nouveau/Projet … ».
Le bouton encerclé de rouge permet de « faire tourner » le programme jusqu'au point d'arrêt (avant `return 0;`), ou de faire l'exécution en « pas à pas ».

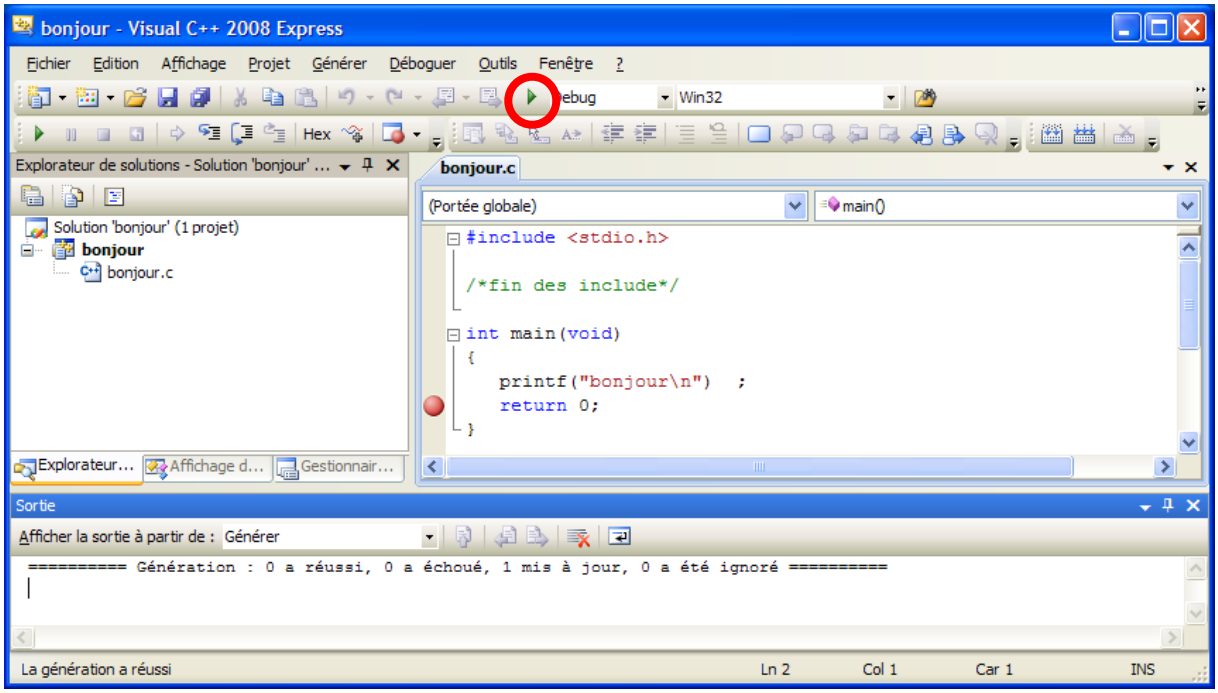


Figure 1 : écran de l'environnement Visual C++ 2008 Express

2.3 Calcul de factorielle

Le programme suivant (re)demande à l'utilisateur un nombre entre 0 et 7 jusqu'à ce qu'il soit effectivement entre 0 et 7, puis affiche la factorielle du nombre saisi.

```
#include <stdio.h>
int main (){
char n;     /*le type « char » est un 'petit' entier de 0 à 255*/
int i , fact = 1; /*initialisation possible lors de la déclaration*/

/*Lecture au clavier d'un décimal mis dans n (à son adresse)*/
printf("entrez 0<=n<=7 : "); scanf("%d",&n);
/*While effectue le bloc {} tant que la condition entre () est vraie*/
while((n<=0)||(n>7)) {        /*|| est le « ou » logique*/
if (n<=0) printf("0>n! "); else printf("n>7! ");
printf("entrez 0<=n<=7 : "); scanf("%d",&n);
}
/*Pour i de 2 « i=2 » à n « i<=n » par pas de 1 « i++ »*/
for(i=2;i<=n;i++) fact*=i; /*fact*=i <=> fact=fact*i;*/
/*Le 1er %d affiche n en décimal, le 2ème fact. Ex : écrit 5!=120 si n=5.*/
printf("%d! = %d\n",n,fact);
return 0;
}
```

*Notions vues : char, déclaration de variable, =, initialisation, scanf, &variable, while, ||, condition, if, for, i++, *=*
Notions approfondies : printf avec format, %d

2.4 Trier un tableau

Le programme suivant trie et affiche un tableau de valeurs entières.

```
/*Ce programme comporte plusieurs fonctions*/
#include <stdio.h>

enum {MAX = 5}; /*MAX est une constante entière globale*/
int tab[MAX] = {3,10,-5,8,2 } ; /*le tableau tab[0..4] est global*/

/* prototype des fonctions : type rendu nom ( paramètres ) */
/* trie le tableau global tab de taille MAX (pas de paramètres) */
void trier(void);
/*affiche le tableau t de taille éléments */
void afficher(int taille,int t[]);

void main(void) { /* main a pour rôle d'appeler trier et afficher */
trier();
afficher(MAX,tab);
}

void trier(void) {
int i, j, cle;

for (i = 1; i < MAX; i++) {
cle = tab[i];
for (j = i-1; (j >= 0) && (cle<tab[j]); j--) tab[j+1] = tab[j];
tab[j+1] = cle;
}
}

void afficher(int taille,int t[]) {
int i;

printf("\nvoici le tableau trié\n");
for(i = 0; i < taille; i++) printf("%5d",t[i]);
printf("\n");
}
```

Notions vues : structure de programme, type énuméré, type tableau, initialisation de tableau, variable globale, prototype
Notions approfondies : paramètres, for
Fonctionnement : Passage de paramètres

Exercices :

- *Ajouter une fonction permettant de lire les MAX entiers*
- *Modifier le programme pour autoriser d'en lire moins que MAX*

2.5 Échanger deux variables

Le programme échange deux valeurs réelles saisies au clavier.

```
#include <stdio.h>

void echanger(float * ad_f1, float * ad_f2) {
/* ad_f1 est un pointeur (ad comme adresse) sur un réel (appelé f1)
*ad_f1 est la valeur réelle pointée :float *ad_f1<=>float f1 */
float tampon = *ad_f1;
*ad_f1 = *ad_f2;
*ad_f2 = tampon;
}

void main() {
float x, y;

printf("Donnez deux réels : \n");
scanf("%f%f", &x, &y);
printf("Avant ,  x=%f ;  y=%f\n", x, y);
echanger(&x, &y); /* A l'appel : ad_f1 reçoit &x, ad_f2 reçoit &y */
printf("Après ,  x=%f ;  y=%f\n", x, y);
}
```

*Notions vues : type float, *variable (**pointeur**), %f*

Notions approfondies : adresse d'une variable passée en paramètre.

2.6 Concaténer deux chaines

Le programme concatène deux chaînes de caractères saisies au clavier.

```
#include <stdio.h>
#include <stdlib.h> /* pour malloc et free */
#include <string.h>

enum {MAXCAR = 10};

void Lire(char *prompt,char s[MAXCAR]);
/* affiche prompt (dont la taille est quelconque)
puis lit au clavier la chaîne s (au max MAXCAR caractères) */
char * AlloueEtConcat(char *, char *);
/* Alloue une chaîne pour stocker la concaténation des 2 chaînes
puis concatène effectivement et rend l'adresse du résultat */

void main() {
char prenom[MAXCAR], nom[MAXCAR]; /* chaînes de MAXCAR allouées */
char *prenom_nom;     /* chaîne de taille quelconque (vide ici) */

Lire("prénom",prenom);
Lire("nom",nom);
prenom_nom = AllouEtConcat(prenom, nom);        /* prenom_nom pointe
       maintenant sur la chaîne allouée et remplie dans AllouerEtConcat */
printf("prénom nom=%s",prenom_nom);
free(prenom_nom); /* récupère l'espace mémoire alloué précédemment */
}

void Lire(char * prompt,char s[MAXCAR]) {
printf("Donnez le %s (au plus %d caractères) : ",prompt,MAXCAR - 1);
scanf("%s",s);
/* utiliser fgets(s,MAXCAR,stdin); si on veut protéger le dépassement */
}

char * AllouEtConcat(char * s1, char * s2) {
int longueur=strlen(s1) + strlen(s2) + 2;
char * s3=(char *) malloc(longueur*sizeof(char));
/* (char*) devant malloc transforme le void* rendu par malloc en char* */

if (s3==NULL) {
puts("mémoire insuffisante");
exit(1); } /* arrêt immédiat du programme en erreur */
else {
strcpy(s3, s1); /* copie s1 en début de s3 */
strcat(s3, " "); /* puis ' ' */
strcat(s3, s2); /* enfin s2 */
}
return s3; /* rend l'adresse de la chaîne résultat */
}
```

Notions vues :string.h, strlen, strcpy, strcat, stdlib.h, malloc, free, exit, sizeof, conversion de type
Notions approfondies :scanf, chaîne de caractère
Fonctionnement : malloc.

2.7 Les nombres complexes

Le programme définit un nouveau type permettant de mémoriser des nombres complexes. La partie de description du type et les entêtes des fonctions sont dans le fichier "complexe.h"

```
typedef struct qqconque {
float re;
float im;
} complexe ;

complexe creer (float rho,float theta) ;
/* création par coordonnées polaires
creer=(rho*cos(theta), rho*sin(theta))*/
complexe somme (complexe x,complexe y) ;
/* somme=x+y (en complexe)*/
float module(complexe c);
/* module**2=c.re**2+c.im**2 */
```

Le corps des fonctions sur les complexes est décrit dans "complexe.c".

```
#include <math.h>
#include "complexe.h"

complexe creer (float rho,float theta) {
complexe c={rho*cos(theta),rho*sin(theta)};
return (c);
};
complexe somme (complexe x,complexe y) {
complexe c={x.re+y.re,x.im+y.im};
return (c);
};
float module(complexe c){
return (sqrt(c.re*c.re+c.im*c.im));
};
```

Le programme principal est le suivant (dans le fichier "testComplexe.c").

```
#include <stdio.h>
#include <math.h>
#include "complexe.h"

int main(void){
complexe c=creer(1.,0.);
printf("c=(%f,%f)\n",c.re,c.im);
c=creer(1.,4.*atan(1.));
printf("c=(%f,%f)\n",c.re,c.im);
c=somme(c,creer(1.,0.));
printf("c=(%f,%f)\n",c.re,c.im);
c=creer(1.,atan(1.));
printf("|(%f,%f)|=%f\n",c.re,c.im,module(c));
};
```

Le programme exécutable est fabriqué comme suit avec gcc :

>gcc –c complexe.c => fabrication de complexe.o
>gcc –c testComplexe.c => fabrication de testComplexe.o
>gcc testComplexe complexe.o –o testComplexe => fabrication de testComplexe.exe

Notions vues :struct, typedef, compilation séparée, math.h,
Notions approfondies : prototype

3.Structure de Donnée Abstraite

3.1 C'est quoi une SDA ?

C'est un ensemble de données reliées logiquement, et dont l'organisation permet la manipulation individuelle ou collective de ces données.

Cette notion de **SDA** est indépendante de tout langage de programmation. Un exemple élémentaire est le type entier (`int`). Les actions possibles sont l'affectation, la lecture au clavier, les opérations +,*,-, /, …

Un exemple plus complexe, défini comme type construit dans les langages de programmation est le tableau (`type []`). Il contient divers éléments d'un même type de base.

Les actions sont : définir/demander la taille, lire/écrire un élément par son numéro, …

Le concepteur du logiciel va **regrouper** tous les éléments informatiques (types, constantes, procédures et fonctions) qui concourent à la définition d'une SDA dans un **module** informatique. Cet élément deviendra à son tour une brique de base pour de nouvelles conceptions au même titre que les types existant dans les langages classiques.

3.2 Spécification et implémentation d'une SDA

Dans le même esprit de l'indépendance par rapport à un langage de programmation, une SDA se spéficifie par la description logique du **contenu** et des **actions** possibles sur les données sans connaître sa programmation effective.

Une SDA définit une **abstraction** des données et de leurs manipulations et **cache l'implémentation**.

Reprenons l'exemple des entiers. Aucun besoin de connaître la manière de **coder** chaque entier en mémoire pour comprendre les actions et opérations associées. La **spécification** logique est simplement celle de l'entier mathématique.

On décrit en général l'ensemble des actions possibles en 4 classes :

- Les constructeurs (fabriquer une donnée à partir de valeurs de base)
- Les sélecteurs (interroger la SDA sur ces valeurs de base)
- Les modificateurs
- Les itérateurs (parcourir toutes les données de la SDA)

Exemples d'actions pour le tableau :

- Modificateur : `t[i]=a ;`
- Sélecteur : `if (t[i]==a) …`
- Itérateur : index de l'élément `[i]`

Le tableau est un exemple significatif de la description d'un ensemble (toujours au sens mathématique) avec éventuellement répétition de la même valeur (alors appelé « sac »). Bien évidemment, un constructeur comme « ajouter » n'aura pas la même signification pour un ensemble ($E \cup \{x\}=E$ si $x \in E$) et pour un sac ($S \cup \{x\} \neq S \ \forall \ S$).

Les structures de données que nous allons étudier sont vues essentiellement comme des représentations informatiques d'ensembles de valeurs. Les différences résident dans leur structure : **linéaire** (liste, file, pile, …) ou **hiérarchique** (arbre binaire, arbre de recherche, …). Celles-ci influent sur la performance - appelée « **complexité** » - des actions, que ce soit en occupation mémoire (en espace), ou un temps d'exécution (en temps).

Une manière élégante (et souvent optimale) de spécifier et d'implémenter des SDA fait appel à la **récursivité**. La définition récursive d'un objet fait référence à l'objet lui-même (**Figure 2**) !

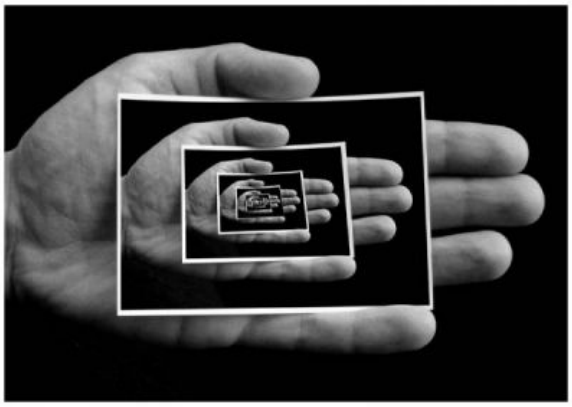


Figure 2 : exemple « imagé » de la récursivité !

Un exemple bien connu des musiciens est l'effet Larsen ! Le son émis dans le micro est amplifié puis diffusé par les enceintes. Si celles-ci sont trop proches du micro (ou dirigées vers elles), elles alimentent à leur tour le micro qui amplifie le son … etc.

L'exemple courant en mathématique, est la fonction « factorielle » :

- Par définition 0!=1 et pour n>0 n! = n × (n-1)!

Exercice :

- *Ecrire le programme récursif de la factorielle*

3.3 La liste linéaire

C'est une structure qui permet de ranger des valeurs les unes après les autres. Nous verrons sur cet exemple simple, que plusieurs spécifications et implémentations permettent de répondre à des problèmes différents tout en conservant le même modèle de données, le n-uplet.

3.3.1 Spécifications

L'objet mathématique (e_1, e_2, … e_n), suite finie de n éléments d'un ensemble E, modélise bien les données de la structure. Par contre, ses utilisations peuvent être diverses. Voyons deux exemples plus spécialisés de spécifications :

3.3.1.1 Le Vecteur

Dans le cas d'un problème de géométrie en 3D, la taille est fixe, et l'accès aux composantes peut se faire aléatoirement (dans n'importe quel ordre).

$v \in \Re^3 \Leftrightarrow v=(x_1, x_2, x_3)$ est le modèle de données (appelons-le Vecteur)

Créer(x1,x2,x3 : réels) → Vecteur	**constructeur**

X1(v : Vecteur) → réel ; X2(v : Vecteur) → réel ; X3(v : Vecteur) → réel ou bien

X (i : Entier ; v : Vecteur) → réel **sélecteurs**

X1(v : ↑Vecteur ; r : réel) ; X2(v : ↑Vecteur ; r : réel) ; X3(v : ↑Vecteur ; r : réel)
ou bien
X (i : Entier ; v : ↑Vecteur ; r : réel) **modificateurs**

Les actions sont décrites par leur **signature :** le nom, les paramètres (avec le mode de sortie éventuel représenté par ↑), puis le type de retour pour une fonction.

Cette définition ne suffit pas à savoir précisément ce que fait l'action, même si le nom le laisse présager. La définition du modèle logique de données permet de décrire plus précisément le rôle de l'action dans une **post-condition** :

X1(v : ↑Vecteur ; r : réel) ;
-- le coefficient x1 de v vaut r en fin d'action X1 {écriture informelle}
-- $x'_1=r$ {écriture formelle}

Implicitement, le terme x_1 désigne la composante de v qui est décrit par (x_1, x_2, x_3). L'apostrophe indique la valeur en fin d'action.

Cet ensemble de la signature de l'action et de la description de ce qu'elle réalise sur les paramètres et appelée **spécification logique** de l'action. Le regroupement du modèle de données et des spécifications des actions constitue la **spécification de la SDA**.

3.3.1.2 Le Tirage

Si on veut mémoriser des tirages d'un dé pour les analyser globalement. La taille, initialement nulle augmentera au fur et à mesure. L'ajout d'une valeur se fera par exemple toujours au bout, et l'accès aux valeurs se fera en « prenant » les éléments un par un.

$t \in \Re^n \Leftrightarrow t=(e_1, e_2, \ldots, e_n)$ est le modèle de données (appelons-le Tirage)
Tirage_Vide() → Tirage
-- Tirage_Vide =()
Ajouter(t : ↑Tirage ; e : entier)
-- $t'=(e_1, e_2, \ldots, e_{n+1})$ et $e_{n+1}=e$ {e est ajouté en fin de t}
Taille(t : Tirage) → entier
-- Taille=n
Enlever(t : ↑Tirage ; e : ↑entier)
-- Taille(t)>0 {cette assertion est appelée une **pré-condition**}
-- $t'= (e_1, e_2, \ldots, e_{n-1})$ et $e_n=e$ {e_n est enlevé de t et rendu dans e}

On peut noter que « Enlever » joue à la fois le rôle de sélecteur, de modificateur, et d'**itérateur**, car il permet de traiter les éléments les uns après les autres.

Ces deux exemples diffèrent essentiellement par un critère : la taille qui est fixe ou variable. Il en résulte deux manières différentes d'implémenter ces structures.

3.3.2 Implémentations

L'implémentation d'une SDA répond au « comment est programmée » la structure. Cette étape est déterminante pour l'efficacité des **programmes**. La mesure de cette efficacité se décline à la fois en temps d'exécution (**complexité temporelle**) et en espace mémoire utilisée (**complexité spatiale**).

Pour le Vecteur, un tableau de 3 réels suffit, la création correspond à la déclaration de la variable, et les accès se font par indexation (voir exemple du 3.2).

3.3.2.1 Le Vecteur

Un vecteur se déclare en utilisant le type tableau du langage C comme suit :

```
double v[3];
```

Les actions sont immédiates avec l'accès direct. La seule difficulté réside dans la transposition de l'indice !

`v[0]`, `v[1]` et `v[2]` représentent les coordonnées 1,2 et 3

Qu'en est-il de la complexité ?

La taille utilisée pour mémoriser un vecteur de R^3 est exactement de 3 réels. Il suffit de connaître la taille mémoire d'un réel (4 ou 8 octets) pour en déduire celle d'un vecteur. Pour un vecteur de R^n, il faudrait une taille de 4n ou 8n octets. On utilise la notation **O(1)** lorsque la taille est **constante** et **O(n)** si elle est **linéaire** par rapport à n.

Le temps d'accès à une composante d'un vecteur est aussi constant, et indépendant du nombre de composantes. `v[i]` permet d'accéder en une instruction à la $i^{ème}$ composante du vecteur. Cette complexité temporelle est aussi notée **O(1)** : elle ne dépend pas de la taille des données. Par contre, l'initialisation d'un vecteur de R^n à 0 nécessite n initialisations élémentaires (`v[i]=0` pour i de 0 à n-1), soit une complexité temporelle de **O(n)**.

Les calculs précédents fournissent un ordre de grandeur, valeur suffisante pour déterminer la faisabilité d'un algorithme ou d'une description de données. Rappelons une échelle de comparaison de valeurs qui sera utile pour la suite. Lorsque n est grand, on a :

$$\log(n) \ll \sqrt{n} \ll n \ll n\log(n) \ll n^2 \ll n^3 \ll 2^n \ll e^n \ll n! \ll n^n \ll 2^{2^n}$$

Dès que la taille de la structure varie, il faut créer des données en fonction de la demande (réalisation **dynamique**) et non pas une seule fois en début de programme (réalisation **statique**).

Pour le Tirage, on peut choisir le tableau ou la liste. La première solution nécessite de connaître le nombre maximal de tirages. La deuxième permet d'ajouter autant de valeurs que l'on veut, avec obtention de la mémoire en cours de programme, pour peu que l'on ne dépasse pas la capacité totale de notre machine !

3.3.2.2 Le Tirage

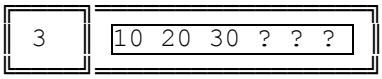
Sous forme de **tableau** avec taille maximum, une solution est :

```
enum {MAX = 3};
typedef struct
{
      int   n;                /* entre 0 et MAX */
      int   e[MAX];           /* les valeurs rangées dans 0,1,…n-1 */

} ttirage;
```

Exemple de remplissage :

```
ttirage t; for (t.n=0 ;t.n<MAX ;t.n++) t.e[t.n]=10*(t.n+1) ;
```

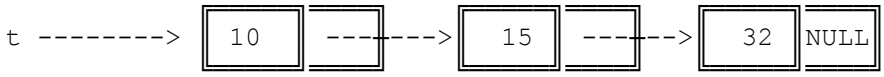


Exercice :

- *écrire les spécifications et l'implémentation d'une fonction d'ajout d'une valeur dans la SDA (attention si c'est plein) et d'affichage de la SDA.*

Sous forme de **liste**, l'implémentation est plus délicate. Elle nécessite l'utilisation d'un pointeur sur une cellule (`cellule*`) qui contient l'entier à mémoriser et un pointeur sur la cellule suivante ! C'est une définition **récursive** car elle fait référence à elle-même.

```
typedef struct  cell   {
      int               valeur;           /* valeur mémorisée */
      struct cell       *suivant;   /* pointeur sur la cellule suivante : on
utilise récursivement struct cell !*/
} cellule;
typedef cellule *tirage;
```

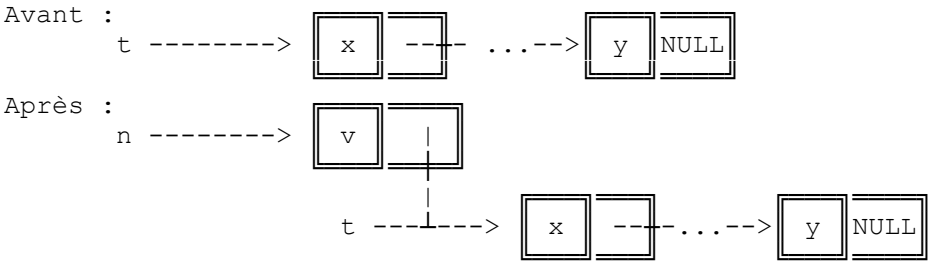


Le type ci-dessus décrit une liste associative. On ne peut accéder au i^{ème} élément (à partir de la tête de liste) qu'en « passant par » les précédents (**Figure** 3) !

Figure 3 **: impossible de voir la 3ème image directement sans passer par la 2ème**

Lorsqu'on déclare `tirage` t, il faut initialiser t à `NULL` (constante provenant de <stdlib.h>) pour que la liste soit vide.

```
      tirage ajouter (int v, tirage t){
          tirage n=(tirage) malloc(sizeof(cellule)); /* nouvelle cellule */
            if (n) { /* n a bien été créé */
                  n->valeur=v;
                  n->suivant=t;
            };
            return n; /* retourne NULL si problème d'allocation */
      };
```



Pour que n pointe sur une cellule, lors de l'ajout d'une nouvelle valeur, il faut allouer la place par : `malloc()`. Si la place est bien créée, n !=NULL. On accède alors aux deux parties de la structure par : `n->valeur` et `n->suivant` :`n->xxx est une abbréviation de (*n).xxx`
Pour parcourir une liste déjà construite (ici juste l'afficher), la solution récursive suivante est facile à comprendre.

```
void afficher (tirage t){
      if (t!=NULL) {
            printf("%d ",t->valeur);
            afficher(t->suivant); /* appel récursif avec la suite */
      }
};
```

Si t est vide (i.e. `t==NULL`), il n'y a rien à faire, sinon il faut afficher la valeur, puis recommencer avec la liste suivante. Le suivant est aussi une liste, donc soit vide soit pointant sur une autre cellule, etc.
Notons une notion importante pour la **programmation récursive**. Les appels doivent s'arrêter, sinon le programme est faux. Ici, comme on suppose que la liste n'est pas circulaire, le nombre de valeurs du `t` transmis diminue de un à chaque nouvel appel. Le `t` transmis sera bien `NULL` au bout de n appels si la liste initiale contenait n cellules. Ce n≥0, appelé **paramètre de taille de la récursivité**, doit décroître strictement à chaque appel.
Le calcul de la complexité est simple ici, car il y a un appel par valeur de la liste, et chaque appel est en O(1) – un test puis un affichage et un appel récursif -, donc l'affichage résultant d'une liste de n cellules est en O(n).

Exercices :

- *écrire l'affichage sans appel récursif.*
- *écrire un ajout récursif des entiers de 1 à n.*
- *comment modifier afficher pour que les valeurs soient dans l'ordre d'ajout ?*

La récursivité n'est pas utile qu'en écriture de code. On peut l'utiliser aussi pour caractériser le comportement des actions. Voyons encore un exemple avec la liste. Il est inspiré du langage précurseur : le LISP (list processing 1958 !).

3.3.3 Spécifications algébriques

Les actions sont décrites d'une part, par leur **signature fonctionnelle**. Une liste d'entiers (par exemple) possède :

	Deux constructeurs :				
Nil	:	Ø	→	liste	-- la liste vide
Cons	:	entier x liste	→	liste	-- concaténer un entier suivi d'une liste
	Un testeur :				
Vide	:	liste	→	logique	
	Deux sélecteurs :				
Car	:	liste	→	entier	-- le premier entier
Cdr	:	liste	→	liste	-- la liste suivante

Note : les termes « car » et « cdr », acronymes de Content Address Register et Content Decrement Register ont pour origine l'implantation des listes en registre de l'IBM 704 !

D'autre part, le comportement de ces fonctions est régit par une suite d'axiomes :
Car(Cons(i,l))=i Cdr(Cons(i,l)=l
Vide(Nil) est vrai Vide(Cons(i,l)) est faux

Ces **équations fonctionnelles** jouent le rôle des post-conditions.
Des pré-conditions doivent aussi être exprimées :
Car(l) est_défini_si non(Vide(l))
Cdr(l) est_défini_si non(Vide(l))

On peut alors enrichir en fonctions de calcul, par exemple la longueur :
Long : liste → int -- la longueur de la liste
Axiomes :
Long(Nil)=0 Long(Cons(i,l))=Long(l)+1

L'intérêt est alors la simplicité de la **programmation récursive** à partir de la spécification !
Long(l)= si Vide(l) alors 0 sinon Long(Cdr(l))+1

Exercice :

- *Spécifier puis programmer l'ajout d'un entier en fin de liste (au contraire de Cons).*

Le passage rapide (ou simple) des spécifications à l'implémentation est un challenge important de l'informatique actuelle. Il permet non seulement d'accélérer le processus de développement de code, mais aussi de le sécuriser car les raisonnements à base de spécifications sont relativement faciles à faire. Vous étudierez ces concepts lors des cours ultérieurs (essentiellement en AFGL - Aspect Formels du Génie Logiciel). Reste le problème de l'implémentation efficace !

En résumé, nous avons vu la notion de Structure de Données Abstraite au travers d'un exemple simple : la liste.
Une SDA se spécifie en fonction de l'utilisation que l'on en a, et peut s'implémenter de diverses manières. Chaque implémentation doit être « mesurée » autant en temps d'exécution qu'en occupation mémoire par un calcul de complexité. Dans certains cas, la récursivité s'adapte à la définition de la SDA et en facilite l'écriture.

4.Structures Classiques

Nous avons vu que les listes modélisent un ensemble d'éléments successifs (sans accès direct) et sans notion d'ordre. Le mode d'accès à chaque élément et le mode d'ajout gouvernent les différentes SDA suivantes.

4.1 Structures dérivées des listes

4.1.1 La pile

Cette SDA dynamique n'a qu'un point d'accès : son sommet. On peut :

- ajouter une valeur au sommet (empiler ou *push*)
- lire ou retirer (dépiler ou *pop*) la dernière valeur ajoutée
- tester si la pile est vide

Un exemple concret et donné par les poupées russes (**Figure 4**) :

Figure 4 : Les poupées russes s'emboitent les unes dans les autres

Le terme anglais est *stack* ou LIFO, acronyme de Last In First Out.
Fonctionnement de la pile : ENSTA C02 tr. 38..50

4.1.1.1 Spécification, implémentation et test en C

Ecrivons directement en C dans trois fichiers distincts une solution :

Le fichier « pile.h » contient la définition du type pile (sa spécification et son implémentation) et les spécifications des actions portant sur la pile :

```
/* le type pile est :
  1- modélisé comme une suite (p1,p2, ...pn)
  2- implémenté comme une liste */
typedef struct  cell   {
      int               sommet;          /* de la pile */
      struct cell       *suivant;
} cellule;
typedef cellule *pile;

pile pileVide();
      /* pile=() */
void empiler (int sommet, pile *p);
      /* p'=(p1,p2,...pn,sommet) */
void depiler (pile *p);
      /* non(vide(p))
         p'=(p1,p2,...pn-1) */
int sommet (pile p);
      /* non(vide(p))
         sommet=pn */
int vide (pile p);
      /* vide=(p=()) */
```

Le fichier « pile.c » contient l'implémentation des actions :

```
#include "pile.h"
#include <stdlib.h>

pile pileVide(){ return NULL;} ;
void empiler (int sommet, pile *p){
      pile n=(pile) malloc(sizeof(cellule));
      if (n) {
            n->sommet=sommet;
            n->suivant=*p;
      };
      *p=n;
};
void depiler (pile *p){ *p=(*p)->suivant; };
int sommet (pile p){ return p->sommet;};
int vide (pile p){ return p==NULL;};
```

Ces deux fichiers constituent le module décrivant en C la SDA « pile »

Le programme de test montre à la fois l'utilisation du module « pile » et la cohérence des actions du module (non exhaustive) !

```
#include "pile.h" /* utilisation du module pile */
#include <stdio.h> /* utilisation du module stdio */

int main(void)    /* les commentaires décrivent les vérifications faites */
{ pile p=pileVide();
printf("la pile est %s vide ",(vide(p)?"":"non"));   /* vide(pileVide()) */
      empiler(1,&p);
printf("la pile est %s vide ",(vide(p)?"":"non"));
                                          /* !vide(p) après empiler(1,p) */
printf("%d ",sommet(p));                 /* 1=sommet(p) après empiler(1,p) */
      empiler(2,&p);
printf("%d ",sommet(p));                 /* 2=sommet(p) après empiler(2,p) */
      depiler(&p);
printf("%d ",sommet(p));   /* 1=sommet(p) après empiler(2,p);depiler(&p) */
      depiler(&p);
printf("la pile est %s vide ",(vide(p)?"":"non"));
                        /* pile vide après empiler,empiler,depiler,depiler */
      depiler(&p);                    /* doit planter car la pile est vide */
return 0;
}
```

L'inclusion de « pile.c » n'est pas nécessaire car lors de la compilation, seuls les éléments de « pile.h » sont utiles au système (nom du type et signature des actions). D'autre part, l'environnement se charge de la relation avec « pile.c » pour la construction de l'exécutable.
Le « run » du programme produit l'affichage suivant :

la pile est vide la pile est non vide 1 2 1 la pile est vide

puis une erreur (volontaire) d'exécution survient car on essaye d'accéder, dans « depiler », à *p ce qui est impossible car p est NULL ! D'où l'importance des pré-conditions.

Les piles sont très utilisées en algorithmique. Un exemple interne au mécanisme de compilation/exécution est l'empilement du contexte lors de l'appel d'un sous-programme puis sa restitution par dépilage au retour dans l'appelant. Un autre exemple classique est la vérification des parenthèses d'une expression (empile sur "(" et dépile sur ")").

Voyons une utilisation élémentaire pour la programmation – non triviale – d'un jeu connu depuis longue date … Édouard Lucas en 1892 !

4.1.1.2 Les tours de Hanoï : « des tours » récursif ☺

Des disques de plus en plus petits sont empilés sur la première de trois tours. Il faut transférer les disques sur une autre tour en conservant la propriété suivante : aucun disque ne doit être posé sur un plus petit (**Figure 5**) !

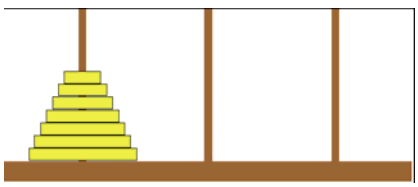


Figure 5 : Tours de Hanoï avec sept disques

Modélisons les tours par des piles : P1, P2, P3 et chaque disque par son numéro (de 1 à n). L'état initial du jeu est construit en empilant successivement les disques n, n-1, … ,1 sur P1.

```
#include "pile.h"
enum {NT = 3,ND = 3};
pile tours[NT]; /* tours[0]=P1, tours[1]=P2, tours[2]=P3 */

int main(void)
{ int i;
      for (i=0;i<NT;i++) tours[i]=pileVide();
      for (i=0;i<ND;i++) empiler(ND-i,&tours[0]);
      ...
}
```

Le problème s'écrit comme suit : passer n disques de P1 à P2 en passant par P3. Voici les spécifications de cette action.

```
void hanoi(int n, int de, int vers, int par) ;
/* passer n disques de tours[de] vers tours[vers] en passant par tours[par]
   pré: n<=ND et tours[de] possède les disques n,n-1, ...,1 (autres vides)
   post: vide(tours'[de]) et tours'[vers]=( n,n-1,...1) */
```

Le « main » appelle l'action « hanoi » ainsi :

```
hanoi(ND,0,1,2);
      /* passer les ND disques de la tour[0] vers tour[1] via tour[2] */
```

Essayez d'imaginer une solution non récursive à ce problème, même en voyant une solution dans un cas particulier (**Figure 6**) ? C'est presque impossible …

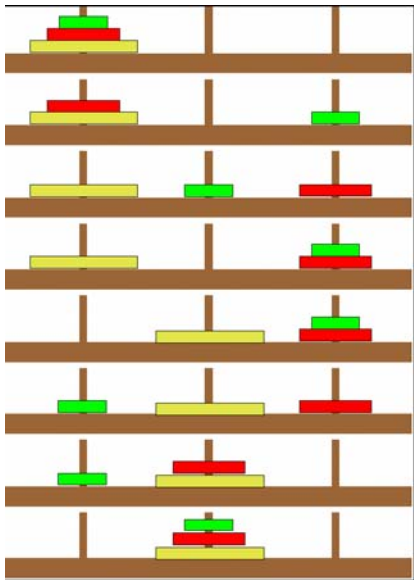


Figure 6 : Solution avec trois disques

Grâce à la récursivité, le problème se réduit ainsi :
Pour passer n disques de P1 à P2 par P3, il suffit de ❶ passer les n-1 du dessus de P1 à P3 par P2, ❷ déplacer le disque n qui reste de P1 à P2, ❸ remettre les n-1 de P3 à P2 par P1 !
Le paramètre de taille est n, il décroit bien strictement entre chaque appel, et le cas trivial est n=0 (il n'y a rien à faire).
Voici l'implémentation de l'action `hanoi` avec l'affichage intermédiaire de déplacement :

```
      if (n>0) {
            hanoi(n-1,de,par,vers); ❶
            ❷ depiler(&tours[de]);       /* le disque n */
            ❷ empiler(n,&tours[vers]);   /* passe de "de" vers "vers" */
```

```
        printf("disque %d de P%d vers P%d",n,de+1,vers+1);
        hanoi(n-1,par,vers,de); ❸
};
```

Et le résultat d'exécution montrant les sept déplacements :

```
disque 1 de P1 vers P2
disque 2 de P1 vers P3
disque 1 de P2 vers P3
disque 3 de P1 vers P2
disque 1 de P3 vers P1
disque 2 de P3 vers P2
disque 1 de P1 vers P2
```

Notons que la modélisation des tours par des piles et théoriquement inutile dans ce programme, car on sait que c'est le disque transmis en paramètre qui est déplacé d'une tour vers l'autre ! Mais pour assurer une visualisation comme celle de la **Figure 6**, cette modélisation est nécessaire. D'où l'exercice suivant !

Exercice :

- *écrire le corps de l'action spécifiée ci-dessous qui affiche le contenu d'une tour*

```
void affiche(int n) ; /* affiche les disques de la tour tour[n] */
```

puis écrire le main pour produire l'affichage suivant :

```
Etat initial ------->  P1=(3 2 1 )  P2=()  P3=()
disque 1 de P1 vers P2 P1=(3 2 )  P2=(1 )  P3=()
disque 2 de P1 vers P3 P1=(3 )  P2=(1 )  P3=(2 )
disque 1 de P2 vers P3 P1=(3 )  P2=()  P3=(2 1 )
disque 3 de P1 vers P2 P1=()  P2=(3 )  P3=(2 1 )
disque 1 de P3 vers P1 P1=(1 )  P2=(3 )  P3=(2 )
disque 2 de P3 vers P2 P1=(1 )  P2=(3 2 )  P3=()
disque 1 de P1 vers P2 P1=()  P2=(3 2 1 )  P3=()
```

Fonctionnement de la pile : ENSTA C02 tr. 9..13
Fonctionnement : hanoï

4.1.2 La file

Cette SDA dynamique a deux points d'accès : une tête et une queue. On peut :

- ajouter une valeur à la queue (enfiler ou *enqueue*)
- lire ou retirer (défiler ou *dequeue*) la première valeur ajoutée
- tester si la file est vide

Le terme anglais est *queue* ou FIFO, acronyme de First In First Out.

Un exemple concret et donné par les files d'attente (**Figure 7**) :

Figure 7 : Une file d'attente, le premier entré est le premier servi !

On peut reprendre tout de la structure de la pile, sauf les actions sommet et défiler qui doivent traiter « l'autre bout » de la liste. Soit ① il faut parcourir toute la liste pour chercher le dernier, soit ② on conserve un pointeur sur le dernier élément. L'étude des complexités nous permet de décider : ① le temps d'exécution est en O(n) au lieu de O(1), ② on occupe un pointeur de plus, ce qui est négligeable par rapport aux données stockées.

Voici la solution du ② , considérée comme meilleur choix « file.h » :

```
/* le type file est :
  1- modélisé comme une suite (f1=tete,f2, ...fn=queue)
  2- implémenté comme une liste */
typedef struct  cell   {
      int               valeur;          /* de la file */
      struct cell       *suivant;
} cellule;
typedef struct {
      cellule *tete; /* enfile en queue et defile en tête */
      cellule *queue;
} file;

file fileVide();                                        /* file=() */
void enfiler (int valeur, file *f);  /* f'=(f1,f2,...fn,valeur) */
void defiler (file *f);   /* non(vide(f)) et f'=(f2,...fn-1,fn) */
int tete (file f);                  /* non(vide(f)) et tete=f1 */
int vide (file f);                              /* vide=(f=()) */
```

« file.c » :

```
#include "file.h"
#include <stdlib.h>

file fileVide(){ file f = {NULL,NULL}; return f; } ;
void enfiler (int valeur, file *f){
      cellule *n=(cellule*) malloc(sizeof(cellule));
      cellule c= {valeur,NULL} ;
      if (n) *n= c;
      if (f->queue!=NULL) f->queue->suivant=n; else f->tete=n;
      f->queue=n;
};
void defiler (file *f){ (*f).tete=(*f).tete->suivant;
                        if (f->tete==NULL) f->queue=NULL; };
int tete (file f){ return f.tete->valeur;};
int vide (file f){ return f.tete==NULL;};
```

« testFile.c » :

```
#include "file.h" /* utilisation du module file */
#include <stdio.h> /* utilisation du module stdio */

int main(void)
{ file f=fileVide(); int i ;
printf("la file est %s vide ",(vide(f)?"":"non"));  /* vide(fileVide()) */
      for(i=1;i<=3;i++) enfiler(i,&f);
printf("la file est %s vide ",(vide(f)?"":"non"));
                                  /* !vide(f) après enfiler((1,2,3),f) */
      for(i=1;i<=3;i++) {                  /* tete(f)=1 puis 2 puis 3 */
            printf("%d ",tete(f));
            defiler(&f);
      };
printf("la file est %s vide ",(vide(f)?"":"non"));
```

```
                                    /* file vide après 3 enfiler et 3 defiler */
      i=tete(f); /* doit planter car on ne doit pas lire d'une file vide */
return 0;
}
```

Exercice :

- *Vous avez du remarquer que ni « depiler » ni « defiler » ne rendent la mémoire utilisée. Modifiez leur implantation à l'aide de la fonction « free » pour récupérer la place.*

Fonctionnement de la file : ENSTA C02 tr. 38..50

4.1.3 Autres structures ensemblistes linéaires

La liste simplement chaînée (**Figure 8**), la pile et la file constituent la base des SDA ensemblistes.

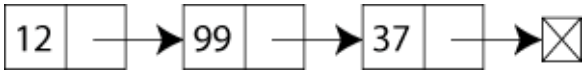


Figure 8 : liste d'entiers simplement chaînée (12,99,37)

Certaines variations permettent d'adapter ces structures à des situations particulières. Citons comme exemples :
La **liste avec itérateur**. Vous avez du remarquer qu'on doit modifier la SDA pour pouvoir scruter tous ces éléments ! Ceci peut se résoudre en rendant accessible un **curseur courant** Cc (**Figure 9**) qui peut se déplacer dans le sens du chaînage. Il est ainsi possible de visiter successivement tous les éléments de la SDA, mais aussi d'insérer après Cc ou de supprimer la cellule pointée par Cc.

Figure 9 : liste avec curseur courant

La **liste doublement chaînée** (**Figure 10**) permet de se « balader » dans les deux sens, mais aussi d'ajouter (**Figure 11**) ou de supprimer des cellules avant ou après Cc!

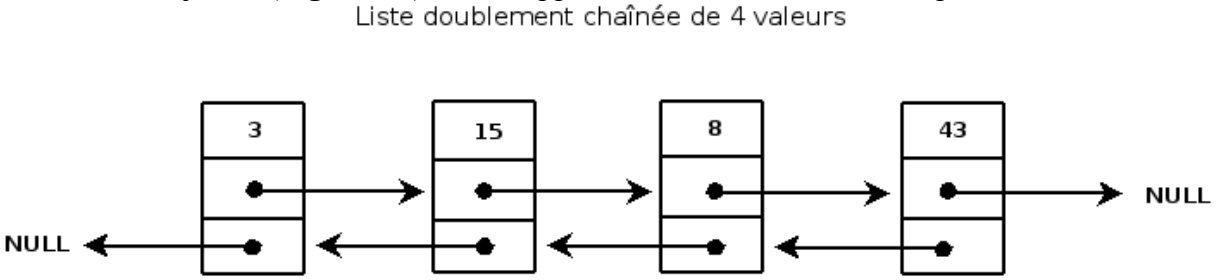


Figure 10 : liste doublement chaînée (3,15,8,43)

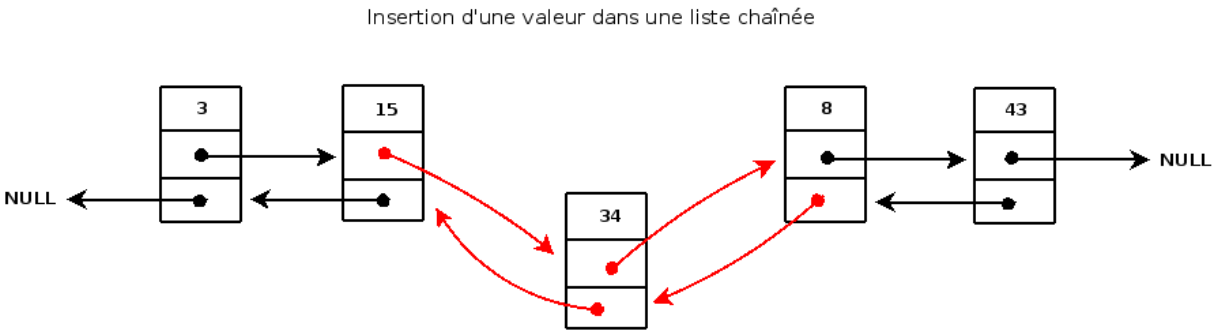


Figure 11 : insertion de 34 dans (3,15,8,43) entre 15 et 8

La **liste ordonnée** ne fonctionne qu'avec un type de données possédant un ordre total ! L'ajout d'un élément ne se fera ni en tête ni en queue, mais de manière à ce que les éléments successifs restent dans l'ordre. La complexité de cet ajout (qui nécessite la comparaison de l'élément à ajouter avec ceux de la liste) est en O(n) au lieu de O(1).

Fonctionnement : insertion dans une liste chaînée ordonnée.

L'**ensemble** peut se modéliser comme une liste sans répétition de valeur. La spécification mathématique est connue que ce soit sur un ensemble lui-même (ensemble vide, appartenance, cardinal, ajout, …) ou des opérations concernant plusieurs ensembles (union, intersection, …).

Le **dictionnaire** est une extension de l'ensemble, chaque élément possédant 2 parties : la **clef** et la valeur. L'unicité est testée sur la clef. Les actions sont généralement : ajouter un couple (clef, valeur), retirer un couple (clef), tester la présence (clef) obtenir la valeur (clef).

Ces trois dernières SDA font référence à un ordre de valeur. On note que l'ajout dans l'ordre ou la recherche sont de complexité O(n) s'il y a n éléments !
Un simple calcul montre que pour insérer correctement n valeurs, le temps d'exécution sera de l'ordre de 1+2+…+n opérations, soit $O(n^2)$. Ce coût devient prohibitif dans certains cas. Les structures arborescentes que nous allons étudier dans le prochain chapitre permettent la réduction sensible de cette complexité d'ajout et de recherche de valeur.

Exercice :

- *Si l'ensemble est implémenté dans un tableau ordonné t(1..n), la recherche **dichotomique** permet de trouver un élément e en $O(\log_2(n))$. On compare e au milieu du tableau t(n/2). S'il est plus petit on recommence entre 1 et n/2, sinon entre n/2 et n, ainsi de suite jusqu'à soit trouver e, soit être sûr de son absence. Ecrivez cet algorithme (en récursif ou non).*

4.2 Arbres

4.2.1 Principe, forme générale

Un arbre est un modèle abstrait d'une structure hiérarchique. Il est constitué de nœuds reliés par des arrêtes (une relation parent-enfant). Il n'y a pas de boucles (**Figure 12**)

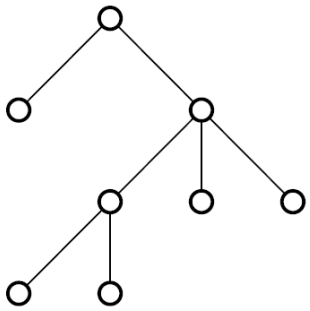


Figure 12 : représentation d'un arbre

Terminologie

On utilise classiquement la terminologie suivante pour décrire les arbres (**Figure 13**) :
La **racine** de l'arbre est l'unique nœud qui n'a pas de père.
Un **nœud interne** possède au moins un fils.
Une **feuille** de l'arbre est un nœud qui n'a pas de fils.
Un **sous-arbre** d'un nœud A est un arbre dont la racine est un fils de A.

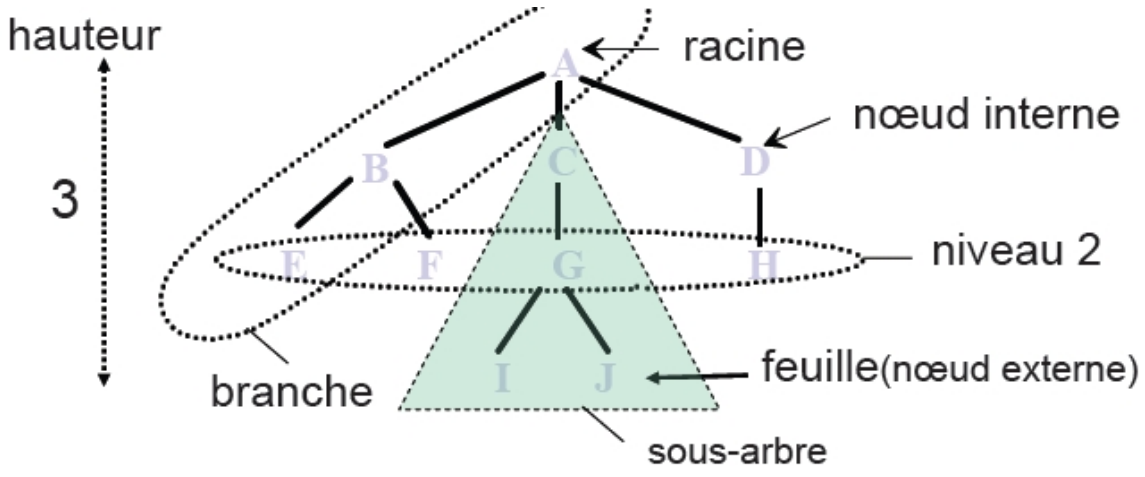


Figure 13: quelques termes

Les termes **père**, **fils** sont repris du langage « généalogique ». De même, (C,D) sont **frères** de B, (A,C) **ancêtres** de G, (G,I) **descendants** de C. Les termes suivants sont plus spécifiques :

Un chemin qui relie une feuille à la racine est une **branche**.
La **profondeur** d'un nœud est son nombre d'ancêtres. Le **niveau** situe les nœuds de même profondeur. La **hauteur** d'un arbre est sa profondeur maximale.

Applications

Quelques exemples donnent un aperçu de l'étendue de leur utilisation :

1. Le livre, organisé en chapitres, sections, paragraphes
2. La représentation d'une expression arithmétique pour l'évaluer, la dériver,… (**Figure 14**)

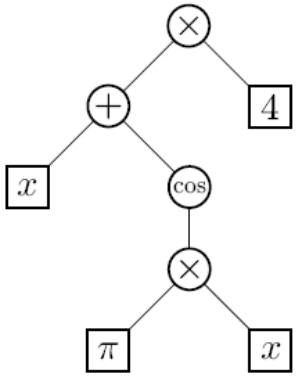


Figure 14: modélisation arborescente de l'expression (x + cos (π × x)) × 4

3. L'organisation des dossiers, sous-dossiers et fichiers
4. Les arbres généalogiques
5. Un arbre de jeu (**Figure 15**)

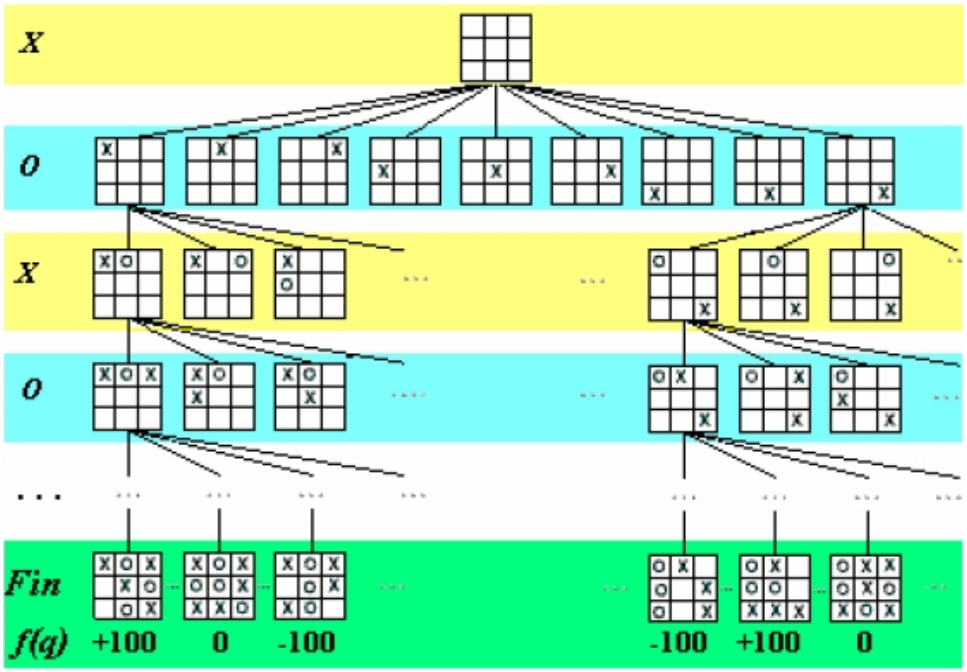


Figure 15: arbre des coups du jeu de « Tic tac toe »

Types d'arbres

Un arbre dont les nœuds ont au plus n fils est un arbre **n-aire**. Ce type d'arbre comprend les arbres **de recherche** ou ABR, et les **B-arbres**.

Lorsque n vaut 2, l'arbre est dit **binaire**. Dans ce cas on utilise les termes de fils gauche et fils droits d'un nœud, ainsi que les notions de sous-arbre gauche(SAG) et de sous-arbre droit (SAD). L' **AVL** est un arbre binaire **équilibré** : | hauteur(SAG)-hauteur(SAD) | ≤ 1

4.2.2 Arbres binaires

La forme d'arbre la plus simple à manipuler est l'arbre binaire. De plus les formes plus générales d'arbres (avec n>2) peuvent être représentées par des arbres binaires.

4.2.2.1 Spécification algébrique

Nous pouvons nous inspirer de la liste d'entiers. Un arbre est soit vide, soit composé d'une valeur (ici entière), d'un sous-arbre gauche et d'un droit :

Deux constructeurs :

arbre : Ø → arbre -- arbre vide
arbre : entier x arbre x arbre → arbre -- (valeur,SAG,SAD)

Notons que deux fonctions ont le même nom, mais se différencient par leurs signatures.

Un testeur :

vide : arbre → logique

Trois sélecteurs qui doivent vérifier la pré-condition : non(Vide(a))

valeur : arbre → entier -- l'entier figurant à la racine
SAG : arbre → arbre -- le sous-arbre gauche
SAD : arbre → arbre -- le sous-arbre droit

Axiomes :

vide(arbre())=vrai vide(arbre(i,a1,a2))=faux
valeur(arbre(i,a1,a2))=i SAG(arbre(i,a1,a2))=a1 SAD(arbre(i,a1,a2))=a2

Comme nous l'avons remarqué au § 3.3.3, il est « aisé » de définir les fonctions :
taille(arbre())=0 et taille (arbre(i,a1,a2))=1+taille (a1)+taille (a2) (nombre de nœuds)
hauteur(arbre())=0 et hauteur (arbre(i,a1,a2))=1+max(hauteur (a1), hauteur (a2))
profondeur(arbre())=0 et profondeur(a_k)$_{k=1,2}$ =profondeur(arbre(i,a1,a2))+1

Les **parcours** se définissent aussi récursivement mais sous 2 formes (**Figure 16**) :

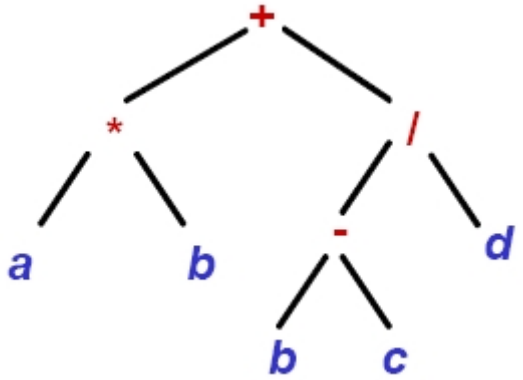


Figure 16**: exemple d'arbre d'expression**

- En **profondeur**, il y a trois possibilités simples :
  préfixe(arbre())=() préfixe(arbre(i,a1,a2))=(i,préfixe(a1),préfixe(a2))
  + * a b / - b c d utile pour l'évaluation
  infixe(arbre())=() infixe(arbre(i,a1,a2))=(infixe(a1), i,infixe(a2))
  (a*b)+((b-c)/d) soit l'écriture « normale » (ici parenthésée)
  suffixe(arbre())=() suffixe(arbre(i,a1,a2))=(suffixe(a1),suffixe(a2),i)
  a b * b c – d / + aussi appelée « polonaise inverse »
- En **largeur**, c'est un parcours par niveau : + puis *,/ puis a,b,–,d et enfin b,c

Notons qu'ici, la structure n'est pas adaptée à l'écriture récursive de ce parcours.

4.2.2.2 Spécification et implémentation en C

Les noms ne peuvent être conservés car C n'accepte pas le même nom de type et de fonction. Les deux constructeurs sont donc nommés arbreVide et arbreCons

```
/* le type arbre est :
  1- modélisé algébriquement : soit vide, soit (arbre gauche,entier,arbre
droit)
  2- implémenté avec deux pointeurs (gauche et droit) par noeud*/
typedef struct cell {
      int valeur;
      struct cell *gauche;
      struct cell *droit;
} noeud ;
typedef noeud *arbre;

arbre arbreVide();      /* arbre=() */
arbre arbreCons(int valeur, arbre fg,arbre fd);/* arbre=(fg,valeur,fd) */
int vide(arbre a);/* vide=(arbre=()) */
int valeur(arbre a);
      /* non(vide(a))
         valeur(arbreCons(i,g,d))=i */
arbre sag(arbre a);
      /* non(vide(a))
         sag(arbreCons(i,g,d))=g */
arbre sad(arbre a);
      /* non(vide(a))
         sag(arbreCons(i,g,d))=d */
```

L'implémentation n'offre aucune surprise :

```
#include "arbre.h"
#include <stdlib.h>
arbre arbreVide(){return NULL;};
arbre arbreCons(int valeur, arbre fg,arbre fd){
      arbre a=(arbre) malloc(sizeof(noeud));
      noeud n={valeur,fg,fd};
      if (a) {*a=n;return a;};
};
int vide(arbre a){return (a==NULL);};
int valeur(arbre a){return (a->valeur);};
arbre sag(arbre a){return (a->gauche);};
arbre sad(arbre a){return (a->droit);};
```

Voici un exemple de construction de l'arbre de la **Figure 17** :

```
arbre v=arbreVide();
arbre a=arbreCons(1,arbreCons(2,arbreCons(3,v,v),v),arbreCons(4,v,v));
```

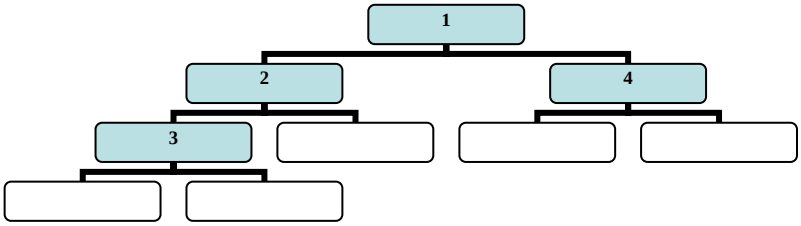


Figure 17: arbre binaire

4.2.2.3 Fonctions simples sur l'arbre

La fonction taille calcule le nombre de nœuds de l'arbre de manière récursive. Si l'arbre est vide, la fonction retourne 0. Si l'arbre contient au moins un nœud, la fonction retourne la somme des tailles du sous-arbre gauche et du sous-arbre droit plus 1.

```
int taille(arbre a){
      if (vide(a)) return 0;
      else return (1+taille(sag(a))+taille(sad(a)));
};
```

La fonction hauteur calcule la hauteur de l'arbre de manière récursive. Elle utilise une fonction max qui retourne le maximum de deux entiers. Si l'arbre est vide, la fonction hauteur retourne 0. Si l'arbre contient au moins un nœud, la fonction hauteur retourne le maximum des hauteurs de ses sous-arbres plus 1.

```
int max(int a, int b){if (a > b) return a; else return b;};
int hauteur(arbre a){
      if (vide(a)) return 0;
      else return (1+max(hauteur(sag(a)),hauteur(sad(a))));
};
```

4.2.2.4 Parcours en profondeur

Les trois versions du parcours en profondeur récursif sont présentées ci-après. Pour chaque version, la fonction affiche le contenu de l'arbre sous une forme parenthésée. Le résultat est donné pour l'exemple déjà vu à la **Figure** 17 dont la construction est :

```
arbre v=arbreVide();
arbre a=arbreCons(1,arbreCons(2,arbreCons(3,v,v),v),arbreCons(4,v,v));
```

- **Parcours préfixe**

Le traitement effectué sur chaque nœud est simplement l'affichage de sa valeur. Dans ce premier parcours préfixé, le traitement du nœud courant a lieu avant les deux appels récursifs.

```
void affichePrefixe(arbre a){
      printf("(");
      if (!vide(a)){
            printf("%d ",valeur(a));
            affichePrefixe(sag(a));
            affichePrefixe(sad(a));
      printf(")");
      };
};
```

Voici le résultat de l'appel affichePrefixe(a) : (1(2(3()())())(4()()))

- **Parcours infixe**

```
void afficheInfixe(arbre a){
      printf("(");
      if (!vide(a)){
            afficheInfixe(sag(a));
            printf("%d",valeur(a));
```

```
            afficheInfixe(sad(a));
      };
      printf(")");
};
```

Voici le résultat de l'appel afficheInfixe(a) : ((()3())2())1(()4())) .
C'est la représentation « naturelle » d'un arbre.

- **Parcours postfixe**

```
void affichePostfixe(arbre a){
      printf("(");
      if (!vide(a)){
            affichePostfixe(sag(a));
            affichePostfixe(sad(a));
            printf("%d",valeur(a));
      };
      printf(")");
};    }
```

Voici le résultat de l'appel affichePostfixe(a) : ((()()3)()2)(()()4)1)

4.2.2.5 Parcours en largeur

Le parcours dit en largeur d'abord consiste à explorer un arbre de gauche à droite puis de haut en bas. Une solution largement répandue fait intervenir une file. Voici l'algorithme :

```
void afficheLargeur(arbre a){
      file f=fileVide();
      printf("(");
      if (!vide(a)){
            enfiler(a,&f);
            while (!videF(f)) {
                  a=tete(f);
                  printf("%d ",valeur(a));
                  defiler(&f);
                  if (!vide(sag(a))) enfiler(sag(a),&f);
                  if (!vide(sad(a))) enfiler(sad(a),&f);
            };
      };
      printf(")");
};
```

Nous détectons deux problèmes majeurs du langage C dans la réutilisation de la « file » !

1. les noms des types « cell », « cellule » ou des fonctions « vide » étant les mêmes dans la définition de la file et de l'arbre, C considère que se sont des redéfinitions illégales. Il faut donc faire attention aux noms identiques dans des modules différents. Ada résout parfaitement le problème grâce au « private » au « package » et au « with ». Vous verrez que C++ utilise les « namespace » et Java le package. En C, il faut renommer ces éléments …
2. la file déjà étudiée mémorise des entiers, et nous voulons une file d'arbres (ou au moins de nœuds d'arbre). Il faut donc modifier le type du champ « valeur », celui du paramètre « valeur » de « enfiler » et de retour de la fonction « tete ». Ada, là encore, résout le problème grâce au « generic », C++ possède une notion moins performante, le « template », et Java s'en sort avec la notion d'héritage et d'interface. En C, là encore, nous devons dupliquer la file initiale et changer les éléments demandés.

La nouvelle file est définie comme suit dans « fileA.h » (cf. § 4.1.2) :

```
#include "arbre.h"
typedef struct  cellF   {
      arbre                  valeur;
      struct cellF           *suivant;
} celluleF;
typedef struct {
      celluleF *tete;
      celluleF *queue;
} file;

file fileVide();
void enfiler (arbre valeur, file *f);
void defiler (file *f);
arbre tete (file f);
int videF (file f);
```

Le jeu des `#include` de fichier .h fait que pour une même chaîne de compilation nous pourrions inclure deux fois « arbre.h ». Voici le fichier « arbreFonctions.h » :

```
#include "arbre.h"

int taille(arbre a);
      /* taille(arbreVide())=0
         taille (arbre(i,a1,a2))=1+taille (a1)+taille (a2)*/
int hauteur(arbre a);
      /* hauteur(arbreVide())=0
         hauteur (arbre(i,a1,a2))=1+max(hauteur (a1), hauteur (a2))*/
void affichePrefixe(arbre a);
      /*préfixe(arbre())=()
        préfixe(arbre(i,a1,a2))=(i,préfixe(a1),préfixe(a2))*/
void afficheInfixe(arbre a);
      /*infixe(arbre())=()
        infixe(arbre(i,a1,a2))=(infixe(a1),i,infixe(a2))*/
void afficheSuffixe(arbre a);
      /*suffixe(arbre())=()
        suffixe(arbre(i,a1,a2))=(suffixe(a1),suffixe(a2),i)*/
void afficheLargeur(arbre a);
      /*affiche par niveau de gauche à droite*/
```

Le fichier « arbreFonctions.c » à pour entête :

```
#include "arbreFonctions.h" /* les fonctions taille, afficheInfixe, … */

#include <stdio.h> /* pour affiche */
#include "fileA.h" /* pour parcours en largeur */
```

… puis les corps des fonctions spécifiées …

Or le fichier « fileA.h » inclut lui-même arbre.h ! ce qui provoquerait l'erreur. Trois directives de compilation conditionnelle associées permettent de résoudre le problème :

```
#ifndef ARBRE_H
#define ARBRE_H
```

… insertion du code du fichier arbre.h qui ne sera alors inclus qu'une fois …

```
#endif
```

4.3 Autres formes d'arbres

4.3.1 Arbres généralisés

Principe

Pour représenter des arbres quelconques on utilise une variante de la représentation fils gauche/fils droit appelée fils gauche/frère droit. Chaque nœud contient un pointeur vers son premier fils (le plus à gauche) et un pointeur vers son frère droit. Avec ce schéma, le nombre de fils d'un nœud donné est arbitraire et indépendant du nombre de fils des autres nœuds.

Déclaration du nœud généralisé en C

```
typedef struct s_noeud_gen *p_noeud_gen_t;
typedef struct s_noeud_gen
{
  int valeur;
  p_noeud_gen_t fils_gauche;
  p_noeud_gen_t frere_droit;

} noeud_gen_t;
```

Déclaration de l'arbre en C

```
typedef p_noeud_gen_t  arbre_t;
typedef arbre_t       *p_arbre_t;
```

4.3.2 Structures dérivées

Les forêts sont des structures constituées d'un ou plusieurs arbres. Une manière de représenter une forêt est d'utiliser un nœud généralisé ayant pour fils chacun des arbres de la forêt.

Les treillis sont des structures similaires aux arbres, à ceci près que les nœuds peuvent avoir plusieurs parents. Cette structure est un cas particulier de graphe appelé aussi D.A.G. - acronyme de directed acyclic graph (graphe orienté sans cycle).

4.4 Les graphes

Les graphes, orientés ou non, sont des structures encore plus générales que les arbres : chaque nœud appelé sommet contient une valeur et un certain nombre de pointeurs vers d'autres sommets appelés arcs (ou arêtes).
Autant la structure de données peut être simple du point de vue de l'implémentation, autant les algorithmes de parcours peuvent être complexes (en particulier les problèmes de chemins dans les graphes) ! Comme il n'y a plus de notion de parenté (comme dans les arbres), il faut mémoriser les nœuds visités lors des parcours pour ne pas traiter deux fois le même …
Nous donnons un bref aperçu de deux modélisations possibles.

4.4.1 Intérêt

On retrouve cette structure pour modéliser divers problèmes tels que :

1. les réseaux : routier (**Figure** 18), aérien, de télécommunications, …

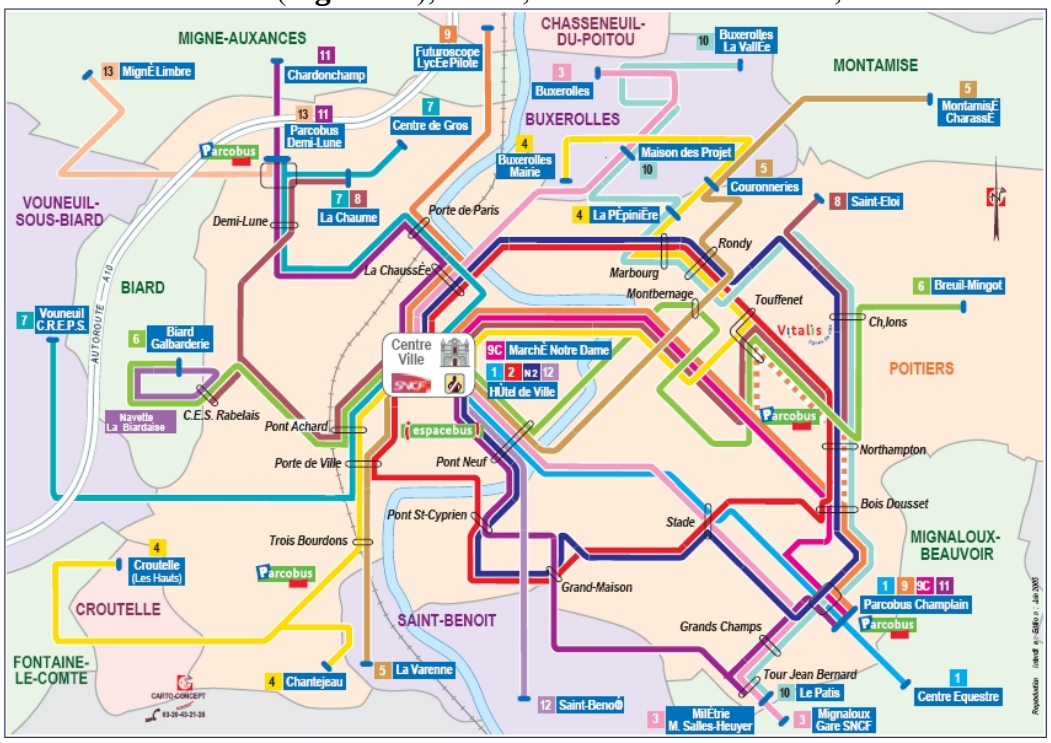


Figure 18: le bus de Poitiers

2. l'organisation temporelle de tâches dans une entreprise ou dans un atelier
3. une carte géographique
4. un automate d'états pour représenter l'évolution d'un système
5. la représentation symbolique de nombreux problèmes tel celui présenté **Figure** 19 : comment en partant d'un endroit de la ville, parcourir tous les ponts une seule fois puis revenir au même endroit ?

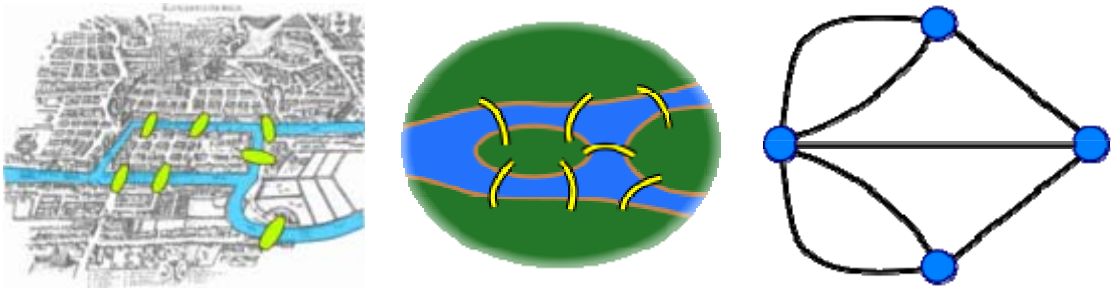


Figure 19: les sept ponts de Königsberg

4.4.2 Modélisation d'un graphe

Mathématiquement, un graphe est un ensemble de sommets et un ensemble d'arcs. Chaque arc est un couple (graphe orienté) ou une paire (graphe non orienté) de sommets.
Le problème des ponts, nécessite d'identifier les zones de la ville et les ponts (**Figure** 20)

Sommets={A,B,C,D}, Arcs={a,b,c,d,e,f,g}, a=(A,B), b=(A,B), c=(A,C), d=(A,C), e=(A,D), f=(C,D), g=(B,D). Le graphe non orienté est le couple (Sommets, Arcs).

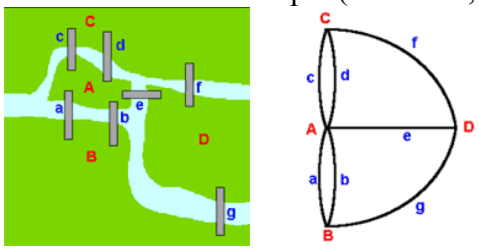


Figure 20: modélisation du problème des 7 ponts de Königsberg

Une implémentation similaire à celle de l'arbre généralisé, même si elle est possible, n'est pas adaptée car le père n'est pas unique ! De plus il manque une information nécessaire ici : la valeur associée à chaque arc (on n'a qu'une information à chaque sommet de l'arbre). D'où la version suivante :

Déclaration du graphe en C

```
typedef struct so sommet; // définitions préliminaires
typedef struct ar arc;
typedef struct nso *l_sommet; // liste de sommets
typedef struct nar *l_arc; // liste d'arcs

typedef struct so // sommet //
{
  char valeur; // l'information associée au sommet
  l_arc arcs; // la liste des arcs
} ;
typedef struct nso // noeud de la liste des sommets
{
  sommet s;
  l_sommet suivant;
} ;

typedef struct ar // arc //
{
  char valeur; // l'information associée à l'arc
  l_sommet s1; // premier sommet de l'arc
  l_sommet s2; // deuxième sommet de l'arc (ordre inutile)
} ;

  typedef struct nar // noeud de la liste des arcs
{
  arc a;
  l_arc suivant;
} ;

typedef l_sommet graphe; // le graphe contient sa liste de sommets //
// elle est suffisante car les sommets contiennent leurs arcs //
```

Primitives élémentaires sur le graphe en C

```
  graphe grapheVide(); // crée un graphe vide //
  void ajouterSommet ( char valeur, graphe *g);
  // ajoute le sommet 'valeur' dans g//
```

```
// pré : le sommet ne doit pas exister //
void ajouterArc ( char valeur, char s1, char s2, graphe *g);
// ajoute l'arc 'valeur' entre les sommets 's1' et 's2' dans g//
// pré : l'arc 'valeur' ne doit pas exister //
void afficher ( graphe g);
```

Le programme permettant de supporter le problème des 7 ponts est :

```
graphe g=grapheVide();
ajouterSommet('A',&g);ajouterSommet('B',&g);ajouterSommet('C',&g);
ajouterSommet('D',&g);
ajouterArc('a','A','B',&g);ajouterArc('b','A','B',&g);
ajouterArc('c','A','C',&g);ajouterArc('d','A','C',&g);
ajouterArc('e','A','D',&g);
ajouterArc('f','C','D',&g);
ajouterArc('g','B','D',&g);
afficher(g);
```

Les ajouts se faisant en tête de liste, voici le résultat du run :

sommet D
  arc g =(B,D)
  arc f =(C,D)
  arc e =(A,D)
sommet C
  arc f =(C,D)
  arc d =(A,C)
  arc c =(A,C)
sommet B
  arc g =(B,D)
  arc b =(A,B)
  arc a =(A,B)
sommet A
  arc e =(A,D)
  arc d =(A,C)
  arc c =(A,C)
  arc b =(A,B)
  arc a =(A,B)

Les actions `grapheVide`, `ajouterSommet` et `afficher` sont simple à écrire (je vous les laisse en *exercice*☺). Par contre, l'ajout d'un arc dans la structure est intéressant à analyser. Il faut d'abord récupérer les pointeurs sur les sommets dont les valeurs sont s1 et s2 (`cherche`), puis ajouter l'arc reliant s1 à s2 dans la liste des arcs de s1 et s2 :

```
void ajouterArc ( char valeur, char s1, char s2, graphe *g){
      l_sommet ps1,ps2;
      arc a;
      ps1=cherche(s1,*g);
      ps2=cherche(s2,*g);
      a.valeur=valeur;a.s1=ps1;a.s2=ps2;
      ajouterArcListe (a,&(ps1->s.arcs));
      ajouterArcListe (a,&(ps2->s.arcs));
};
```

Cette action nécessite deux fonctions intermédiaires :

```
l_sommet cherche (char valeur, l_sommet ls){
      // rend le pointeur sur le sommet 'valeur' //
      if (ls->s.valeur==valeur) return ls;
      return cherche(valeur,ls->suivant);
```

```
};
```

Et

```
void ajouterArcListe ( arc a, l_arc *la){
      // ajoute 'a' dans la liste 'la' //
      l_arc l=(l_arc) malloc(sizeof(struct nar));
      if (l) {
            l->a=a;
            l->suivant=*la;
      };
      *la = l;
};
```

La solution du problème lui-même (le parcours existe t'il ?) possède une réponse (purement mathématique) qui sort du cadre du cours. Un volume énorme de documentation à ce sujet est accessible sur le NET. Voici un des liens provenant d'un site de maths, http://mathenjeans.free.fr/amej/edition/0403koni/04koenig.html dans lequel une réponse est donnée en construisant l'arbre de tous les chemins … encore un bon *exercice* de synthèse.
Le graphe de la ville est complexe, car :

1) il y a plusieurs arcs reliant les mêmes sommets (ici, a et b relient A et B).
2) le graphe n'est pas orienté, ce qui nous oblige à mémoriser deux fois le même arc ('a' appartient aux arcs de 'A' et de 'B').

Il existe des structures plus adaptées aux cas les plus simples.
Lorsqu'il n'y a pas plusieurs arcs reliant les deux mêmes sommets, alors une simple matrice (appelée matrice d'adjacence) suffit. La figure ci-dessous montre un graphe orienté et sa matrice :

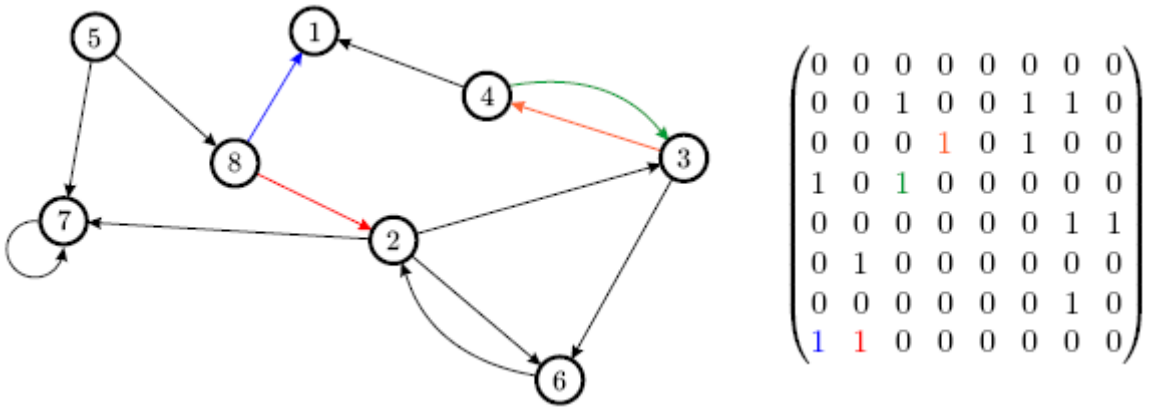


Figure 21: graphe orienté représenté par sa matrice d'adjacence

Cette matrice M permet de résoudre facilement le problème suivant :
Existe-t-il un chemin entre deux sommets donnés ?
$M^k(i,j)$ est le nombre de chemins de longueur k de i vers j (attention, les arcs sont orientés).

Table des figures

Webographie française:

http://runtime.bordeaux.inria.fr/oaumage/oa/Teaching/SSEP_08/
C, Olivier Aumage
http://www.infres.enst.fr/~charon/CFacile/
C_Facile, Irène Charon
http://www-clips.imag.fr/commun/bernard.cassagne/Introduction_ANSI_C/
C, Bernard Cassagne
http://www.di.ens.fr/~fouque/enseignement/ensta/
C, Pierre-Alain Fouque
Cours 1 : programme, macros et fonctions, typage, if et tests, while
Cours 2 : tableaux et for, argc et argv et atoi
Cours 3 : fonctions (prototype, utilisation, paramètres), header,
Cours 4 : * et &, malloc, sizeof, tableaux dynamiques, free, passage par adresse
Cours 5 : struct, typedef, pointeur et ->, récursivité
Cours 6 : structures dynamiques, listes (création, affichage, copie récursifs)
http://www.ibisc.univ-evry.fr/~klaudel/SUPPORTS/coursHK.pdf
Cours unix et C, Hanna Klaudel
http://www-roc.inria.fr/secret/Matthieu.Finiasz/teaching.html
Cours 1 & 2 : complexité, tri & récursivité
Cours 3 & 4 : arbres et tas, AVL, ABR
Cours 5 & 6 : graphes et automates
http://wwwens.uqac.ca/azinflou/INF840.php
Cours d'algorithmique, Arnaud Zinflou
http://www.info.univ-angers.fr/pub/bd/StrDonnees/LPF.pdf
Listes, piles et files, Béatrice Duval
http://cowww.epfl.ch/infosv/slides/cours09.pdf
Structures de Données Abstraites, EPFL Jamila Sam Haroud
https://moodle.insa-rouen.fr/ catégorie de cours ASI cours Base de la programmation et algorithmique
Types Abstraits de Données, Nicolas Delestre