Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Ecole Supérieure en Sciènes Appliquées de Tlemcen
Département De la Formation Préparatoire

Polycopié de Cours, Travaux dirigés et Travaux pratiques

Avec solutions des exercices

Les structures de données complexes

Rédigé par :

Dr. Fouad MALIKI

Préface

La majorité des algorithmes efficaces optimisent le stockage des données afin d'assurer une meilleure gestion de la mémoire. De ce fait, l'utilisation des structures de données en algorithmique s'avère importante pour une meilleure visibilité et exécution des programmes.

En général, les structures de données sont réparties en cinq catégories, à savoir :

- Les structures de données séquentielles (Tableaux)
- Les structres de données linéaires (Listes chainées)
- Les arbres
- Les graphes
- Les fichiers

Ce polycopié est le fruit d'effort de plusieurs années d'enseignement du module algorithmiques 2 et structures de données avancés à l'Ecole Supérieure en Sciences Appliquées de Tlemcen. Il contient sept chapitres de cours portant principalement sur les structures de donnée ainsi que les concepts utilisés pour les implémenter et les manipuler. Plusieurs exercices sont aussi présentés sous forme de travaux dirigés et de travaux pratiques avec solution en langage C afin de mieux comprendre l'utilisation et la manipulation des structures de données expliquées préalablement.

Fouad MALIKI
Maitre de conférences « B »
ESSA Tlemcen

Table des matières

Chapitre I.

Fonctions, procédures et récursivité

I.1 Introduction

Dans tout langage de programmation évolué, il est possible de définir un bloc d'instructions qui pourra être appelé dans n'importe quelle partie du programme principal en faisant référence uniquement par son identifiant. Cette technique de programmation simplifie grandement les algorithmes (programmes) et fait appel à des sous-programmes effectuant chacun des tâches précises.
Il existe deux types de sous-programme:

- Les fonctions
- Les procédures

I.2 Intérêt des fonctions et procédures

Certains problèmes conduisent à des programmes longs, difficiles à écrire et à comprendre. On les découpe en des parties appelées ***sous-programme*** ou ***module***. Les fonctions et les procédures sont des modules (ensemble d'instructions) indépendants désigné par un nom et qui possèdent plusieurs intérêts :

- Permettent de factoriser les programmes c'est-à-dire de mettre en commun les partiesqui se répètent.
- Permettent une structuration et une meilleure lisibilité des programmes.
- Facilitent la maintenance du code (il suffit de modifier une seule fois).
- Ces procédures et fonctions peuvent éventuellement être réutilisées dans d'autreprogramme.

I.3 Les fonctions

Définition

Une **fonction** est un ensemble d'instructions exécutant une ou plusieurs tâches. La fonction est identifiée à un type et restitue une valeur en fin d'exécution. Donc, le rôle d'une fonction en programmation est similaire à celui d'une fonction en mathématique qui retourne un résultat à partir des valeurs des paramètres.
Exemple : y = sin(x) : retourne la valeur du sinus de x.

Déclaration d'une fonction

La déclaration d'une fonction se fait, en précisant:

1. **Son type**
2. **Son nom**
3. **Une liste d'arguments dont on précise le type**

La déclaration algorithmique d'une fonction est sous la forme :
DébutFonction <Nom de fonction> **(**argument1 : **type,** argument 2 : **type,** …**): type de la fonction**

Une fonction possède trois aspects:

1. Le **prototype** : c'est la déclaration nécessaire avant tout.
2. L'**appel** : c'est l'utilisation d'une fonction à l'intérieur d'une autre fonction (par exemplele programme principale).
3. La **définition** : c'est l'écriture proprement dite de la fonction, en-tête et corps.

La ligne de déclaration d'une fonction s'écrit sous la forme:

type <Nom de fonction> **(type** argument1**, type** argument 2**,** …**) ;**

- Les arguments (paramètres) servent à échanger des données entre le programme principale (ou la fonction appelante) et la fonction appelée.
- Les paramètres placés dans la déclaration d'une fonction sont appelés ***paramètres formels***. Ces paramètres peuvent prendre toutes les valeurs possibles mais ils sont abstraits (n'existent pas réellement).
- Les paramètres placés dans l'appel d'une fonction sont appelés ***paramètres effectifs***, ils contiennent les valeurs pour effectuer le traitement.
- Le nombre de paramètres effectifs doit être égal au nombre de paramètres formels.
- L'ordre et le type des paramètres doivent correspondre.

Exemple: Programme utilisant une fonction qui renvoie le maximum de deux entiers

```
Algorithme max_entier
Variables utilisées:
a,b,x : nombres entiers
DébutFonction fun_max(a:entier , b:entier) : entier
 max : nombre entier

Si (a1 >= b1) alors
   max = a1
 Sinon max
 = b1 FinSi
 Retourner(max)
FinFonction
Début { Programme Principal}
 1) lire(a,b)
 2) x = fun_max(a,b)
 3) ecrire(x)
Fin {Programme Principal}
```

```
#include <stdio.h>
int a, b, x;

int fun_max(int a1, int b1);

int fun_max(int a1, int b1){
 int max;

 if (a1 >= b1)
  max = a1;
  else
   max = b1;
 return max;
}

int main(){
 printf("Donnez deux entiers \n");
 scanf( "%d %d", &a, &b);
 x = fun_max(a, b);
 printf("%d\n", x);
 return 0;
}
```

Dans cet exemple, les arguments "a1" et "b1" sont utilisés pour calculer le maximum. Les variables "a", "b" et "x" sont déclarés dans le programme principal. On les appelle **variables globales**. **Une variable globale** déclarée dans le programme principal est reconnue à la fois par le programme principal et par tous les sous-programmes qui sont déclarées par la suite. La variable "max" est dite variable locale.

I.4 Les procédures

Définition

Dans certains cas, on ne peut avoir besoin de répéter une tâche dans plusieurs endroits du programme, mais dans cette tâche on ne calcule pas de résultats ou qu'on calcule plusieurs résultats à la fois. Dans ce cas on utilise une procédure.
Une **procédure** peut être définit comme étant un ensemble d'instructions, identifié par un nom, qui exécutent une ou plusieurs tâches. Elle est un bloc de programme qui travaille sur des données fournies ou contenues dans le bloc de programme. Aucune valeur n'est associée au nom de la procédure.
Remarque : Une procédure est une fonction qui ne renvoie aucune valeur.

Déclaration d'une procédure

La déclaration d'une procédure se fait de la même manière qu'une fonction avec la différencede ne pas préciser son type. La déclaration algorithmique d'une procédure est sous la forme : **DébutFonction** <Nom de fonction> **(**argument1 : **type,** argument 2: **type,** …**)**
La ligne de déclaration d'une procedure s'écrit sous la forme:
void <Nom de fonction> **(type** argument1**, type** argument 2**,** …**);**

Exemple: programme qui affiche les 10 premiers entiers en utilisant un tableau

```
Algorithme max_entier
Variables utilisées:
 a (50): tableau d'entiers
 n : nombre entier

DébutFonction init (n :entier) Pour
 i Dans [0..n-1] ParPasDe 1a(i) = i
 FinPour
FinFonction

DébutFonction imprimer (n :entier)
 Pour i Dans [0..n-1] ParPasDe 1
 ecrire(a(i))
 FinPour
FinFonction


Début { Programme Principal}
 1) n = 10
 2) init(n)
 3) imprimer(n)
Fin {Programme Principal}
```

```
#include <stdio.h>
#define taille 50
int a[taille];
int n ;

void init(int i);
void imprimer(int i);

void init(int n){
 int i;

 for (i = 0 ; i < n; i++)
  a[i] = i;
}

void imprimer (int n){
 int i;

 for (i = 0; i < n; i++)
  printf("%d\n", a[i]) ;
}

int main (){

 n = 10;
 init(n);
 imprimer(n);
 return 0;
}
```

Remarques

La fonction est un cas particulier de la procédure. La différence entre fonction et procédure setrouve à deux niveaux :

- Au niveau du résultat, la fonction délivre un résultat et un seul.
- Au niveau de l'appel, l'appel apparaît toujours dans une expression ou affectation.

Une procédure peut ne pas posséder aucuns paramètres. Dans ce cas, elle réalise toujours lamême action lorsqu'on l'invoque.

I.5 Retour sur la fonction main

Nous avions indiqué que tout programme C/C++ devait comporter une procédure main, et nous pouvons désormais préciser quels en sont les arguments. Rappelons que le prototype de cette procédure est : **int** main (**int** argc, **char ***argv[]);
Les paramètres (que tout le monde a coutume de dénommer argc et argv bien que, comme pour toute procédure, le nom puisse être choisi tout à fait arbitrairement) sont donc : un entier, qui est le nombre d'arguments de la ligne de commande au moment de l'exécution duprogramme (argc pour argument count) ; un tableau de chaînes de caractères, qui sont les arguments (argv pour argument values). Ainsi, si le programme prog est lancé par la commande :

prog arg1 arg2

La valeur de la variable argc sera de 3, et le tableau argv contiendra :

argv[0] = "prog"
argv[1] = "arg1"
argv[2] = "arg2"

NB1 : noter que le nom du programme lui-même fait partie des arguments, et que la valeur argc vaut donc au minimum 1 !
NB2 : tous les arguments sont du type chaîne de caractères ; si l'on souhaite utiliser un argument de type entier ou flottant, il faudra utiliser une fonction de conversion à l'intérieur du code source du programme ;
NB3 : il faut noter que l'argument argv n'est pas défini comme un tableau à deux dimensions de caractères ; en effet, rien n'impose que tous les arguments de la ligne de commande aientla même longueur !

I.6 Porté des fonctions et des variables

En C, le principe de modularité permet qu'un programme soit constitué d'un ou plusieurs modules. Un module pouvant contenir des fonctions et des déclarations de variables.

Au niveau de la définition des variables, on distingue deux types de variables:

- les variables communes appelées **variables globales** qui sont connues de toutes les fonctions définies au sein du même programme source.
- Les **variables locales** qui sont définies au sein d'une fonction et qui ne sont connues qu'à l'intérieur de cette fonction.

Les variables globales

Dans le cas d'une variable globale, sa définition doit apparaitre en-dehors des fonctions qui l'utilisent et doit précéder leur définition.

On dit que sa portée est limitée à la partie du programme qui suit sa définition. La définition d'une variable entraine l'allocation d'un espace mémoire permanent pour cette variable et éventuellement l'initialisation de sa valeur. Quand la valeur de cette variable est modifiée dans le corps d'une fonction, sa nouvelle valeur sera connue de toutes les fonctions qui l'utilisent.

D'une manière générale, les variables globales existent pendant toute l'exécution du programme dans lequel elles apparaissent. Leurs emplacements en mémoire sont parfaitement définis lorsde l'édition de liens. On traduit cela en disant qu'elles font partie **de la classe d'allocation statique**. De plus, ces variables se voient **initialisées à zéro**, avant le début de l'exécution du programme, sauf, bien sûr, si vous leur attribuez explicitement une valeur initiale au moment de leur déclaration.

En C, il est possible de faire la **compilation séparée** de plusieurs modules réparties sur différents fichiers, et faisant partie d'un même projet. La portée d'une variable globale étant limitée au fichier source dans lequel elle a été définie, le langage C prévoit une déclaration quispécifie que la variable a déjà été définie dans un autre fichier source en faisant inclure une déclaration de cette variable spécifiant que la variable est **externe**. La syntaxe de cette déclaration est comme suit:

```
extern  type nom
```

Exemple **:**

```
int x;                  extern int x;
int main(){             fonct1(){x =
5;
return 0;
}                       }
```

La déclaration d'une variable externe **n'effectue pas d'allocation d'espace mémoire**, ni d'initialisation de sa valeur. Elle ne fait que préciser que la variable globale x est définie par ailleurs et elle en précise le type.

En C, il est possible de limiter la portée d'une variable globale et la rendre inaccessible à l'extérieur du fichier source où elle a été définie, en la déclarant de classe **static** de la façon suivante :

```
static type nom
```

Cette notion généralise la notion de variable locale à tout un fichier source. Il sera doncimpossible d'y faire référence depuis une autre source par **extern**. Mieux, si une autre variable globale apparaît dans un autre fichier source, elle sera acceptée à l'édition de liens puisqu'elle ne pourra pas interférer avec celle du premier source.

Exemple **:**

```
static int x;           int x;
int main(){             fonct2{x =
5;
return 0;               }                       }
```

La variable x déclarée dans le fichier source2 n'aura aucun lien avec la variable x déclarée dans le fichier source1. Car la déclaration **static** de la variable x demande qu'aucune trace de x n'existe en dehors du fichier source1.

Les variables locales

Les variables définies au sein d'une fonction sont dites locales à cette fonction. Elles ne sont connues

qu'à l'intérieur de cette fonction (qui pouvait être main). On dit que leur portée est limitée à cette fonction.

Par défaut, une variable locale a une durée de vie limitée à celle d'une exécution de la fonction dans laquelle elle a été définie. Un espace mémoire lui est alloué à chaque entrée dans la fonction et libéré à chaque sortie. On dit que sa **classe d'allocation est automatique.**

Contrairement à cette classe d'allocation ; il existe une classe d'allocation statique ; qui permet d'attribuer à une variable locale un emplacement permanent. Pour cela ; il suffit de spécifier le mot clé **static** dans la déclaration de la variable :

Exemple :

```
#include<stdio.h>int
main(){
 void fonct();int
 n;
 for( n=1; n<=5; n++)
  fonct();
 return 0;
}


void fonct(){ static
 int i = 0;i++;
 printf("i = %d\n", i) ;
}
```

- Lorsqu'on exécute le programme voici ce qu'on obtient à l'écran :

```
i = 1
i = 2
i = 3
i = 4
i = 5
```

La variable locale i a été déclarée de classe « statique ». On constate bien que sa valeur progresse de un à chaque appel. De plus, on note qu'au premier appel sa valeur est nulle. En effet, comme pour les variables globales (lesquelles sont aussi de classe statique) : **les variables locales de classe statique sont, par défaut, initialisées à zéro.**

Prenez garde à ne pas confondre une variable locale de classe statique avec une variable globale. En effet, la portée d'une telle variable reste toujours limitée à la fonction dans laquelle elle est définie. Ainsi, dans notre exemple, nous pourrions définir une variable globale nommée i qui n'aurait alors aucun rapport avec la variable i de *fonct()*.

Voici un tableau récapitulant la portée et la classe d'allocation des différentes variables suivant la nature de leur déclaration (la colonne « Type » donne le nom qu'on attribue usuellement à de telles variables).

Type de variable	Déclaration	Portée	Classe d'allocation
Globale	en dehors de toute fonction	• la partie du fichier source suivant sa déclaration, • n'importe quel fichier source, avec `extern`.	**Statique**
Globale cachée	en dehors de toute fonction, avec l'attribut `static`	uniquement la partie du fichier source suivant sa déclaration	
Locale « rémanente »	au début d'une fonction, avec l'attribut `static`	la fonction	
Locale à une fonction	au début d'une fonction	la fonction	**Automatique**
Locale à un bloc	au début d'un bloc	le bloc	

I.7 Programmation modulaire

Dès que l'on écrit un programme de taille importante ou destiné à être utilisé et maintenu par d'autres personnes, il est indispensable de se fixer un certain nombre de règles d'écriture. En particulier, il est nécessaire de fractionner le programme en plusieurs fichiers sources, que l'on compile séparément. Ces règles d'écriture ont pour objectifs de rendre un programme lisible, portable, réutilisable, facile à maintenir et à modifier.

Comme tous les langages, le langage C permet de découper un programme en plusieurs parties nommées souvent "modules". Cette Programmation dite "modulaire" se justifie pour de multiples raisons :

- Un programme écrit d'un seul tenant devient difficile à comprendre dès qu'il dépasse une ou deux pages de texte. Une écriture modulaire permet de le scinder en plusieurs parties et de regrouper dans le "programme principal" les instructions en décrivant les enchaînements. Chacune de ces parties peut d'ailleurs, si nécessaire, être décomposée à son tour en modules plus élémentaires ; ce processus de décomposition pouvant être répété autant de fois que nécessaire, comme le préconisent les méthodes de"programmation structurée".

- La programmation modulaire permet d'éviter des séquences d'instructions répétitives, et cela d'autant plus que la notion d'"argument" permet de "paramétrer" certains modules.

- La programmation modulaire permet le partage d'outils communs qu'il suffit d'avoir écrits et mis au point une seule fois. Cet aspect sera d'autant plus marqué que C autorise effectivement la compilation séparée de tels modules.

Il existe deux types de fichier en langage C :

- **Les .h** : appelés fichiers headers.
- **Les .c** : les fichiers sources.

Exemple :

main.c

```
#include <stdio.h>
#include "produit.h"
int main()
{
 int a, b, c;
 scanf("%d",&a);
 scanf("%d",&b);
 c = produit(a,b);
 printf("le produit vaut %d\n",c);
 return 0 ;
}
```

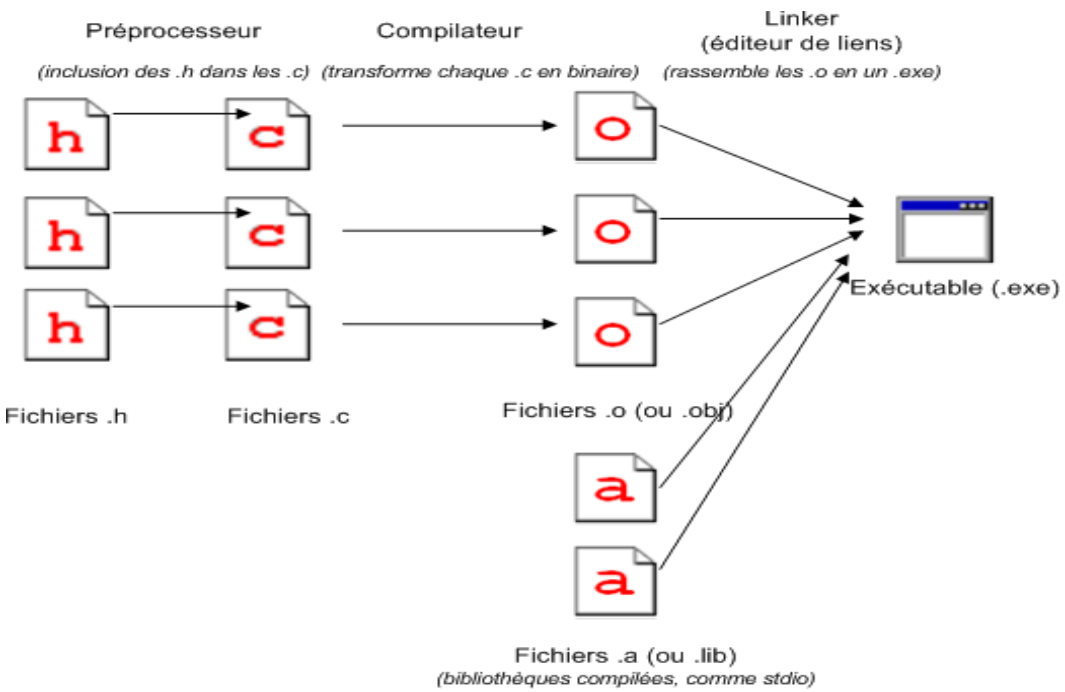
produit.h

```
int produit(int a, int b);
int produit(int a, int b)
 {
  return(a * b);
 }
```

I.8 Compilation séparée

Si le langage C est effectivement un langage que l'on peut qualifié d'opérationnel, c'est en partie grâce à ses possibilités dites de **compilation séparée**. En C, en effet, il est possible de compiler séparément plusieurs programmes fichiers source et de rassembler les modules objet correspondants au moment de l'édition de liens. D'ailleurs, dans certains environnements de programmation, la notion de projet permet de gérer la multiplicité des fichiers (source et modules objet) pouvant intervenir dans la création d'un programme exécutable.
Cette notion de projet fait intervenir précisément les fichiers à considérer ; généralement, il est possible de demander de créer le programme exécutable, en ne recompilant que les sources ayant subi une modification depuis leur dernière compilation. Maintenant que vous savez qu'un projet est composé de plusieurs fichiers sources, nous pouvons rentrer plus en détail dans le fonctionnement de la compilation. Voici un schéma détaillé de la compilation.

- **Préprocesseur** : le préprocesseur est un programme **qui démarre avant la compilation**. Son rôle est d'exécuter les instructions spéciales qu'on lui a données dans des directives de préprocesseur, ces fameuses lignes qui commencent par un #.



- **Compilation** : cette étape très importante consiste à transformer vos fichiers sources en code

binaire compréhensible par l'ordinateur (code assembleur). Le compilateur compile chaque fichier .c séparément. Il compile tous les fichiers source de votre projet, d'où l'importance d'avoir bien ajouté tous vos fichiers au projet.

- **Edition de liens** : le linker (ou "éditeur de liens" en français) est un programme dont le rôle est d'assembler les fichiers binaires .o. et les bibliothèques compilées dont vous avez besoin (.a ou .lib selon le compilateur) dans un seul fichier exécutable final.

I.9 Introduction à la rècursivité

Au cours des chapitres précédents, nous avons eu l'occasion d'utiliser des fonctions, que ce soit pour éviter de recopier plusieurs fois le même code, ou pour découper le code en parties plus simples. Dans ce chapitre, nous allons voir une autre utilisation des fonctions, qui en fait un outil très puissant pour écrire des algorithmes. Cette utilisation est ce que l'on appelle la *récursivité*.

La notion de récursivité est avant tout un problème algorithmique plus qu'au niveau du langage lui même. Que ce soit en C, C++, ou autres langages, l'implémentation d'une fonction récursive se fera toujours plus ou moins de la même manière.

D'un point de vue théorique la récursivité s'exprimera comme étant des programmes ou des fonctions d'un programme qui ont la faculté de s'appeler eux-mêmes (on entend également le terme d'auto-appel ce qui est logique). La récursivité est une manière simple et élégante de résoudre certains problèmes algorithmiques, notamment en mathématique mais cela ne s'improvise pas, il convient donc de savoir comment ce principe fonctionne. Nous verrons comment cela fonctionne en pratique, et à quoi cela peut servir.

La récursivité permet de résoudre certains problèmes d'une manière très rapide, alors que si on devrait les résoudre de manière itérative, il nous faudrait beaucoup plus de temps et de structures de données intermédiaires.

Nous ne verrons pas de nouvelles notions du langage C dans ce chapitre, et n'utiliserons que des éléments du langage que vous connaissez déjà. La récursivité est cependant un concept de base de la programmation, et il est important pour tout programmeur d'être à l'aise avec ce concept.

I.10 Fonctions récursives

Une fonction récursive est une fonction qui s'appelle elle même. Voyons un premier exemple d'une fonction récursive, et d'un programme qui l'utilise:

Exemple 1

```
#include <stdio.h>

void debutFin(int nb){
 printf("début %d\n", nb);
 if (nb > 1)
  debutFin(nb - 1);
  printf("fin %d\n", nb);
}


int main(){
 debutFin(3);
```

```
    return 0;
    }
```

En exécutant ce programme, on obtient l'affichage suivant :

```
début 3
début 2
début 1
fin 1
fin 2
fin 3
```

Dans cet exemple, on passe en paramètre à la fonction le nombre d'appels récursifs imbriqués que l'on souhaite exécuter. Ce nombre est diminué de 1 à chaque appel : pour afficher 3 fois "début" et "fin", on affiche "début", puis on appelle la fonction pour afficher 2 fois "début" et "fin", puis on afficher "fin".

Que se passerait-il si l'on n'avait pas ce paramètre, comme dans l'exemple suivant :

```
void debutFin(){
 printf("début\n");
 debutFin();
 printf("fin\n");
}
```

Lors de son appel, la fonction debutFin() afficherait "début", puis s'appellerait elle-même, donc afficherait début, puis s'appellerait elle-même, donc... cela n'aurait pas de fin, et à aucun moment, le mot fin ne serait affiché donc; **Une fonction récursive doit toujours comporter une condition de fin des appels, pour ne pas avoir une exécution infinie appeler cas de base.**

Voyons maintenant ce qui se passe si au lieu d'avoir un seul appel, la fonction récursive se rappelle elle-même deux fois de suite :

```
void debutFin(int nb){
 printf("début %d\n", nb);
 if (nb > 1){
  debutFin(nb - 1);
  debutFin(nb - 1);
 }
 printf("fin %d\n", nb);
}
int main(){
 debutFin(3);
 return 0;
}
```

L'affichage du programme est alors le suivant :

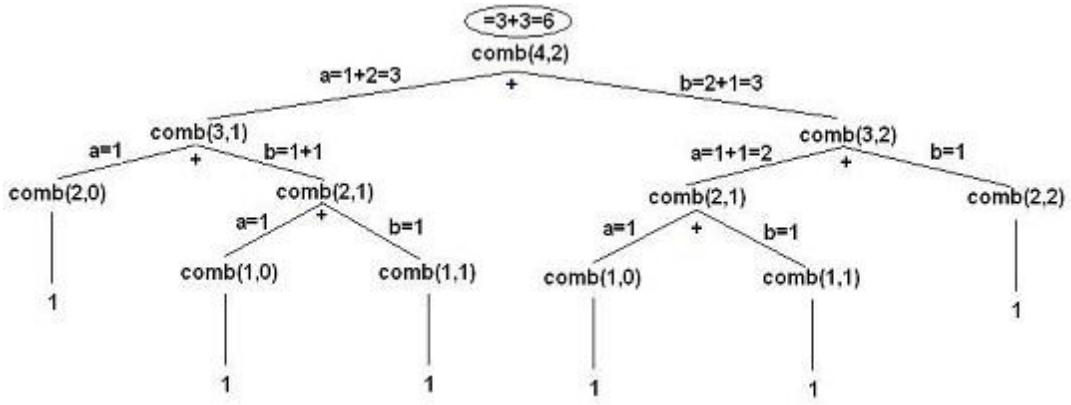
Notons que chaque appel récursif dispose de ses propres variables locales. Ces variables sont déclarées à chaque appel (voir l'exemple ci-dessous), ceci est également vrai pour les paramètres.

Exemple 2

```
int comb(int n, int p){
 int a,b;
  if((p==0)||(p==n))
   return 1;
  else{
```

```
    a=comb(n-1,p-1);
    b=comb(n-1,p);
    return  a+b;
   }
}
int main(){
 comb(4,2);
}
```

Les variables a et b sont déclarés à chaque appel. Ce qui donne comme arbre d'exécution :



Cette fonction permet de calculer la valeur de $\mathbf{C}^{\mathbf{P}}_{\mathbf{.n}}$

Exemples d'algorithmes récursifs et itératifs

Les exemples d'utilisation des fonctions récursives que nous avons vus jusqu'à présent avaient tous une nature récursive, car ils mettaient en œuvre des éléments imbriqués les uns dans les autres. Comme nous allons le voir, il aurait tout à fait été possible de programmer ces exemples sans utiliser de fonctions récursives. De la même manière, il n'est pas nécessaire qu'un problème ait en lui-même une nature récursive, pour qu'il soit possible de le résoudre très simplement avec une fonction récursive.

Prenons par exemple le calcul de la factorielle d'un nombre, une fonction mathématique qui pour une valeur entière positive, retourne le produit de tous les entiers entre 1 et cette valeur. Pour une valeur nulle, la fonction retourne 1. Par exemple, la factorielle de 5, que l'on note "5!", vaut1 * 2 *3 * 4 * 5 = 120. On peut écrire la fonction factorielle sous la forme d'une simple boucle, de la manière suivante :

```
#include <stdio.h>
unsigned int n, i, result;

unsigned int factoriel(unsigned int n);

unsigned int factoriel(unsigned int
n){unsigned int f = 1;
for (i = 1; i <= n; i++)
 f = f * i;
return f;
}

int main(){
 printf("Entrer un entier positif
 :");scanf("%d", &n);
 result = factoriel(n);
 printf("Le factoriel de %d est %d\n", n,
```

```
 result);return 0;
}
```

Il est cependant possible de donner une définition récursive de la fonction factorielle :

La factorielle d'un nombre N vaut 1 si N est égal à 0, et N multiplié par la factorielle de N - 1 sinon. Cette définition est parfaitement équivalente à la précédente, et peut se traduire en code par une fonction récursive :

```
#include <stdio.h>
unsigned int n, result;

unsigned int factoriel(unsigned int

n);unsigned int factoriel(unsigned int

n){
  if(n <= 1)
   return 1;
  else
   return n * factoriel(n-1);
}

int main(){
 printf("Entrer un entier positif:
 ");scanf("%d", &n);
 result = factoriel(n);
 printf("Le factoriel de %d est %d\n", n,
 result);return 0;
}
```

On peut remarquer que le code de cette deuxième version est plus simple que la version avec une boucle, et qu'il peut se lire quasiment comme une définition.

La première version, qui utilise une boucle, est ce que l'on appelle une ***implémentation itérative*** de la fonction factorielle: on effectue un certain nombre d'itérations d'une boucle. La deuxième version s'appelle tout simplement ***l'implémentation récursive***.

I.11 Avantages et inconvénients

Une grande partie des problèmes peut se résoudre avec une implémentation récursive, comme avec une implémentation itérative. L'une ou l'autre peut paraître plus ou moins naturelle suivant le problème, ou suivant les habitudes du programmeur. Avec un peu d'habitude, utiliser l'implémentation récursive permet souvent d'avoir un programme plus simple, plus facile à comprendre, donc à débugger.

L'implémentation récursive a cependant deux principaux inconvénients, qui peuvent être gênants dans certains cas :

- Un appel de fonction prend plus de temps qu'une simple itération de boucle.
- Un appel de fonction utilise une grande quantité de mémoire.

Le premier inconvénient fait que des programmes implémentés avec une fonction récursive seront souvent légèrement plus lents que leurs équivalents itératifs. Si le moindre gain de vitesse pour cette partie de votre programme est important, il peut donc être préférable d'utiliser une implémentation itérative. Dans le cas contraire, la perte de performances peut être largement compensée par le gain en clarté du code, donc en réduction de risques de laisser des bugs.

Le deuxième inconvénient peut être très gênant si le nombre d'appels imbriqués est très important. Chaque appel de fonction imbriqué utilise une certaine quantité de mémoire, plus ou moins importante selon le nombre de paramètres et de variables de votre fonction. Cette mémoire est libérée dès que l'exécution de la fonction se termine, mais dans le cas d'une fonction récursive, cette quantité de mémoire est multipliée par le nombre d'appels imbriqués à un moment donné. Sice nombre d'appels imbriqués peut atteindre des centaines de milliers, voire des millions, on peut facilement atteindre des méga-octets de mémoire, pour un calcul qui ne prendrait aucune mémoire avec une fonction itérative.

Dans certains cas, le compilateur est capable d'éviter de lui-même ces deux inconvénients, en transformant automatiquement votre fonction récursive en un programme itératif. Ceci reste cependant assez rare, et il ne faut donc pas trop compter dessus avec les compilateurs actuels.

I.12 Passage du récursif à l'itératif

Un programme itératif se base sur des boucles pour traiter un certain nombre d'éléments. Un programme itératif simple peut donc ressembler à l'exemple suivant, qui affiche un certain nombre de fois un caractère :

```
void afficheLigne(int nb, char
 c){int i;
 for(i = 0; i < nb; i++)
  printf("%c", c);
 printf("\n");
}
```

Pour écrire une version récursive de ce programme, on commence par se demander dans quel cas la boucle n'est pas du tout utilisée. En l'occurrence, il s'agit du cas où le paramètre **nb** vaut 0, donc qu'on ne fait qu'afficher le retour à la ligne. On peut alors commencer à écrire une fonction qui gèrece cas :

```
void afficheLigne(int nb, char
 c){if(nb == 0)
  printf("\n");
}
```

Reste à gérer le cas où il y a des choses à afficher. Le principe de la fonction récursive est qu'elle s'occupe d'une seule étape, et laisse les étapes suivantes pour les appels imbriqués. Dans le cas où ily a des caractères à afficher, la fonction doit donc afficher un caractère, puis se rappeler, avec comme paramètre le nombre de caractères restant à afficher. Il s'agit de la valeur qu'on lui a transmise, diminuée de 1 :

```
void afficheLigne(int nb, char
 c){if(nb == 0)
  printf("\n");
 else{
  printf("%c", c);
  afficheLigne(nb-1, c);
 }
}
```

Cette fonction réalise exactement la même chose que la version itérative. On peut ainsi dire en français : pour afficher une ligne de N caractères, il faut afficher un caractère, puis afficher une lignede N-1 caractères.

Chapitre II.

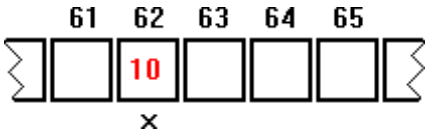
Pointeurs et Allocation dynamique de la mémoire

II.1 Pointeurs

Une variable est destinée à contenir une valeur du type avec lequel elle est déclarée physiquement cette valeur se situe en mémoire. Prenons comme exemple un entier nommé x :

Exemple 1

```
int x;  Réserve un emplacement pour un entier en
mémoire.x = 10;  Ecrit la valeur 10 dans
l'emplacement réservé.
```



Pour obtenir l'adresse d'une variable on fait précéder son nom avec l'opérateur '&' (adresse de) :

```
printf("%d", &x);
```

Ce qui dans le cas du schéma ci-dessus, afficherait 62.

Définition

Un pointeur est aussi une variable, il est destiné à contenir une adresse mémoire, c'est à dire une valeur identifiant un emplacement en mémoire. Pour différencier un pointeur d'une variable ordinaire, on fait précéder son nom du signe '*' lors de sa déclaration.

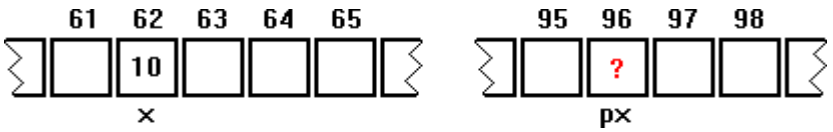
Déclaration :

La syntaxe la plus simple de la déclaration d'un pointeur est : **type *variable**. Exemple: int *p (p est un pointeur de type entier).

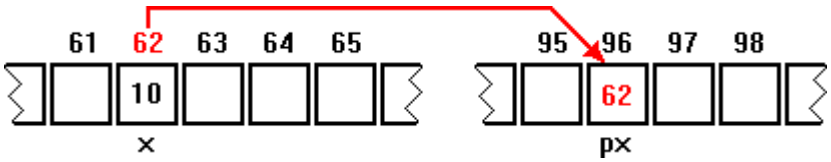
Un pointeur contient en principe une adresse valide, donc il peut prendre l'adresse d'une variable existante, reprenons l'exemple 1:

```
int *px; Réserve un emplacement pour stocker une adresse
mémoire.px = &x;  Ecrit l'adresse de x dans cet emplacement.
```

Lorsqu'on écrit `int *px` :



Lorsqu'on écrit `px = &x` :



A l'emplacement réservé pour le pointeur px, nous avons maintenant l'adresse de x. Ce pointeur ayant comme valeur cette adresse, nous pouvons donc utiliser ce pointeur pour aller chercher (lire ou écrire) la valeur de x. Pour cela on fait précéder le nom du pointeur de l'opérateur de déréférencement '*'. Donc l'instruction suivante :

```
printf("%d",*px);
```

Affiche la valeur de x par pointeur déréférencé (10 dans le cas du schéma). On peut donc de la même façon modifier la valeur de x :

```
*px = 20;  Maintenant x est égal à 20.
```

On se rend vite compte qu'un pointeur doit être initialisé avec une adresse valide, c'est à dire qui a été réservée en mémoire (allouée) par le programme pour être utilisé. Imaginez-vous l'instruction précédente, si nous n'avions pas initialisé le pointeur avec l'adresse de x, l'écriture se ferait en un lieu indéterminé de la mémoire.

Exemple 1 :

```
int *
ad;int
n;
n = 20;
ad = &n;
*ad = 30;
```

La première instruction réserve une variable nommée ad comme étant un pointeur sur des entiers. Nous verrons que * est un opérateur qui désigne le contenu de l'adresse qui le suit. Ainsi, à titre mnémonique, on peut dire que cette déclaration signifie que *ad, c'est-à-dire l'objet d'adresse ad, est de type int ; ce qui signifie bien que ad est l'adresse d'un entier.

L'instruction `ad = &n;`

affecte à la variable ad la valeur de l'expression `&n`. L'opérateur & (que nous avons déjà utilisé avec scanf) est un opérateur unaire qui fournit comme résultat l'adresse de son opérande. Ainsi, cette instruction place dans la variable `ad` l'adresse de la variable n.

L'instruction suivante : `*ad = 30;`

affecte à *ad la valeur 30. Or *ad représente l'entier ayant pour adresse ad (notez bien que nous disons l'entier et pas simplement la valeur car, ne l'oubliez pas, ad est un pointeur sur des entiers). Ceci dit que n soit égale à 30.

Exemple 2:

```
#include <stdio.h>

int
main(){int
a;
int *x,*y;

a = 90;
```

```
    x = &a;
    printf("%d\n",*x);
    *x = 100;
    printf("a vaut : %d\n", a);
    y = x;
    *y = 80;
    printf("a vaut : %d\n", a);
    return 0;
}
```

Explications:

- Dans cet exemple, on définit un entier a et deux pointeurs sur des entiers x et y.
- a est initialisée à la valeur 90.
- Après l'instruction x=&a, x pointe vers a. x contient l'adresse en mémoire où est stockée la variable a.
- L'instruction *x=100; modifie le contenu de la variable a et met la valeur 100 dans cette variable.
- L'instruction y=x copie le pointeur x dans le pointeur y. Après cette instruction, les deux pointeurs x et y pointent vers la même variable a.
- Lorsqu'on écrit *y=80, on modifie alors le contenu de la variable a qui vaut alors 80.
- On voit donc sur cet exemple qu'un pointeur permet de modifier indirectement le contenu d'une variable.
- Lorsqu'on exécute le programme voici ce qu'on obtient à l'écran :

```
90
a vaut : 100
a vaut : 80
```

II.2 Pointeurs et tableaux

Prenons le cas des tableaux : Le nom d'un tableau sans décoration retourne l'adresse du premier élément du tableau. Ceci fait que l'on peut l'utiliser de la même manière qu'un pointeur. Soit le tableau Tab suivant :

```
int Tab[10]={5,8,4,3,9,6,5,4,3,8};
```

L'instruction suivante affiche bien la valeur du premier élément du tableau par pointeur déréférencé.

```
printf("%d",*Tab);
```

Ceci nous montre que le nom d'un tableau peut très bien s'utiliser comme un pointeur sur son premier élément. On peut alors tout à fait déclarer un pointeur et l'initialiser avec le nom du tableau. A condition bien sûr qu'il soit du même type, pointeur sur des entiers dans notre cas :

```
int *pTab;
pTab =
Tab;
```

On peut donc se servir de ce pointeur comme s'il était un tableau et des opérateurs crochets [] pour accéder à ses éléments :

```
printf("%d",pTab[0]);  ou printf("%d",*pTab); Affiche 5.
```

Si on incrémente le pointeur pTab il ne contiendra pas l'adresse du premier élément du tableau + 1, mais l'adresse de l'élément suivant. La valeur de l'adresse qu'il contient sera donc incrémentée de la taille du type qu'il référence. Ceci est l'une des raisons pour lesquelles il faut donner un type à un pointeur. Donc si nous écrivons :

```
printf("%d",*(Tab + 1));     Affiche 8.
```

Nous avons l'affichage du deuxième élément du tableau

II.3 Pointeurs comme paramètres de fonctions

En C, les arguments sont transmis par valeur, les conséquences et les limitations de ce mode de transmission n'apparaissent pas. Or voyez cet exemple :

```
#include <stdio.h>

void echange(int a, int
b);int n=10, p=20;

int main(){
 printf("avant appel : %d %d\n", n, p);
 echange(n, p);
 printf("après appel: %d %d", n, p);
 return 0;
}

void echange(int a, int b){
 int c;
 printf("début echange : %d %d\n", a, b);
 c = a;
 a = b;
 b = c;
 printf("fin echange : %d %d\n", a, b);
}
```

La fonction `echange` reçoit deux valeurs correspondant à ses deux arguments muets a et b. Elle effectue un échange de ces deux valeurs. Mais, lorsque l'on est revenu dans le programme principal, aucune trace de cet échange ne subsiste sur les arguments effectifs n et p.

a. Solution en C (Passage de paramètres par pointeur)

Nous avons vu que le mode de transmission par valeur semblait interdire à une fonction de modifier la valeur de ses arguments effectifs, les pointeurs fourniraient une solution à ce problème. Nous sommes maintenant en mesure d'écrire une fonction effectuant la permutation des valeurs de deux variables. Voici un programme qui réalise cette opération avec des valeurs entières :

```
#include <stdio.h>

void echange(int * ad1, int * ad2);
int a=10, b=20;

int main(){
 printf("avant appel %d %d\n", a, b);
 echange(&a, &b);
 printf("après appel %d %d", a, b);
 return 0;
}
```

```
void echange(int * ad1, int * ad2){
int x;
 x = *ad1;
 *ad1 = *ad2;
 *ad2 = x;
}
```

Les arguments effectifs de l'appel de `echange` sont, cette fois, les adresses des variables n et p (et non plus leurs valeurs). Notez bien que la transmission se fait toujours par valeur, à savoir que l'on transmet à la fonction `echange` les valeurs des expressions &n et &p.

b. Solution en C ++ (Passage de paramètres par référence)

Dans le langage C++, on parle de passage d'arguments par référence :

```
#include
<iostream> using
namespace std;

void echange(int & ad1,int & ad2);
int a=10, b=20;

int main(){
 cout<<"avant appel : "<<a<<" "<<b<<"\n";
 echange(a, b);
 cout<<"après appel : "<<a<<" "<<b;
 return 0;
}

void echange(int & ad1, int & ad2){
int x;
 x = ad1;
 ad1 = ad2;
 ad2 = x;
}
```

Un autre exemple est illustré par le programme suivant qui permet de calculé le minimum et le maximum de deux entiers:

```
 #include <stdio.h>

 void minmax(int i, int j, int * min, int * max);
 void minmax(int i, int j, int * min, int * max){
  if(i<j){
   *min = i;
   *max = j;
  }
  else{
   *min = j;
   *max = i;
  }
 }
 int main(){
  int a, b, w, x;
  printf("Tapez la valeur de a : "); scanf("%d", &a);
  printf("Tapez la valeur de b : "); scanf("%d", &b);
  minmax(a, b, &w, &x);
  printf("Le plus petit vaut : %d\n", w);
```

```
 printf("Le plus grand vaut : %d\n", x);
 return 0;
}
```

Explication:

- Dans cet exemple, on a une fonction minmax qui a comme paramètres 2 entiers i et j et 2 pointeurs vers des entiers min et max. Cette fonction trouve le plus petit de i et de j et le met dans l'entier pointé par min. Elle trouve le plus grand des 2 entiers et le copie dans l'entier pointé par max.
- Dans la fonction main(), on déclare 4 entiers a, b, w, et x. On demande à l'utilisateur de saisir au clavier les entiers a et b. Lors de l'appel de fonction minmax(a,b,&w,&x), on copie la valeur de a dans i, la valeur de b dans j. On copie la valeur de &w (un pointeur vers w) dans min et on copie &x (un pointeur vers x) dans max: min pointe donc vers w et max vers x. Lors de l'appel, on va donc récupérer dans w le plus petit des entiers a et b et dans x le plus grand de ces 2 entiers.
- Lorsqu'on exécute le programme voici ce qu'on obtient à l'écran :

```
Tapez la valeur de a : 25
Tapez la valeur de b : 12
Le plus petit vaut : 12
Le plus grand vaut : 25
```

II.4 Pointeurs Génériques

Nous avons naturellement déjà rencontré des cas d'affectation de la valeur d'un pointeur à un pointeur de même type. A priori, c'est le seul cas autorisé par le langage C/C++. Une exception a toutefois lieu en ce qui concerne la "valeur entière 0", ainsi que pour le type "générique" `void *` dont nous parlerons un peu plus loin. Cette tolérance est motivée par le besoin de pouvoir représenter un pointeur "`nul`" (c'est-à-dire ne pointant sur rien). Bien entendu, cela n'a d'intérêt que parce qu'il est possible de comparer n'importe quel pointeur (de n'importe quel type) avec ce "pointeur nul". D'une manière générale, plutôt que la valeur 0, il est conseillé d'employer la constante NULL prédéfinie dans la library « stdio.h » (bien entendu, elle sera remplacée par la constante entière 0 lors du traitement par le préprocesseur, mais les programmes source en seront néanmoins plus lisibles).

Exemple **:**

```
int * n;
double * x;

n = 0;  ou n = NULL;x = 0;  ou x = NULL;
if (n == 0) ou  if (n == NULL)
```

En effet, un pointeur correspond à la fois à une adresse en mémoire et à un type. Précisément, ce "typage des pointeurs" peut s'avérer gênant dans certaines circonstances telles que celles où une fonction doit manipuler les adresses d'objets de type non connu (ou plutôt susceptible de varier d'un appel à un autre). En fait, il existe le type pointeur **void *** qui désigne un pointeur sur un objet de type quelconque (on parle souvent de "pointeur générique"). Il s'agit exceptionnellement d'un pointeur sans type.
Une variable de type **void *** ne peut pas intervenir dans des opérations arithmétiques; notamment, si p et q sont de type **void ***, on ne peut pas parler de p+i (i étant entier) ou de p-q; on ne peut pas davantage utiliser l'expression p++; ceci est justifié par le fait qu'on ne connaît pas la taille des objets pointés. Pour des raisons similaires, il n'est pas possible d'appliquer l'opérateur de déréférencement * à un pointeur de type **void ***.

Comme tous les autres pointeurs, les "pointeur générique" sont destinés à stocker des adresses. Ils disposent cependant d'un privilège unique qui justifie le qualificatif "générique" qui consiste à prendre pour valeur l'adresse de n'importe quel objet, quel qu'en soit son type.

Exemple **:**

```
void * p;
int n =4;
char l;
p = &n;
p = &l;
```

II.5 Allocation dynamique de la mémoire

En langage C, un programme comporte trois types de données :

Les **données statiques** qui occupent un emplacement parfaitement défini lors de la compilation.
Les **données automatiques** qui n'ont pas une taille définie a priori. En effet, elles ne sont créées et détruites qu'au fur et à mesure de l'exécution du programme. Elles sont souvent gérées sous forme de ce que l'on nomme une pile.
Les **données dynamiques** qui n'ont pas non plus de taille définie a priori. Leur création ou leur libération dépend, cette fois de demandes explicites faites lors de l'exécution du programme.

En définitive, les données d'un programme se répartissent en trois catégories : statiques, automatiques et dynamiques. Les données statiques sont définies dès la compilation ; la gestion des données automatiques reste transparente au programmeur et seules les données dynamiques sont véritablement créées sur son initiative.

Définition

L'allocation dynamique de mémoire permet la réservation d'un espace mémoire pour son programme au moment de son exécution. Ceci est à mettre en opposition avec l'allocation statique de mémoire. En effet, dans ce type d'allocation, la mémoire est réservée dès le début de l'exécution d'un bloc. Commençons par étudier les deux fonctions les plus classiques de gestion dynamique de la mémoire, à savoir **malloc()** et **free()**.

La fonction malloc()

La fonction **malloc()** a la signature suivante :

```
void* malloc (size_t taille);
```

Cette signature montre tout d'abord que le résultat fourni par **malloc()** est un "pointeur générique". Il pourra donc être converti implicitement en un pointeur de n'importe quel type. D'autre part, nous constatons que l'unique argument de malloc est d'un type a priori inattendu (nous aurions pu penser à int ou long). En fait, **size_t** est un nom de type prédéfini.

Exemple 1 **:**

```
#include <stdlib.h>

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

Exemple 2 **:**

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

```
char * adr;
adr = (char *)malloc(50);
for (i=0 ; i<50 ; i++)
  *(adr+i) = 'x';
```

La fonction prend un paramètre : le nombre d'octets à réserver. Ainsi, il suffira d'écrire sizeof(int) dans ce paramètre pour réserver suffisamment d'espace pour stocker un int. Notons que l'inclusion du **stdlib.h** est obligatoire.

La fonction free()

L'un des intérêts essentiels de la gestion dynamique de la mémoire est de pouvoir récupérer des emplacements dont on n'a plus besoin. Le rôle de la fonction **free** est de libérer un emplacement préalablement alloué. La fonction **free()** a la signature suivante :

```
void free(void* pointeur);
```

La fonction free a juste besoin de l'adresse mémoire à libérer. On va donc lui envoyer notre pointeur.

Exemple

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

int main(){
int* p;
 p = (int *)malloc(sizeof(int));
 if (p == NULL)
  exit(0);
 free(p);
 return 0;
}
```

Grâce à malloc et free, on peut gérer des tableaux dont la taille est variable : un tableau peut s'allonger ou se réduire en fonction des besoins du programmeur. Prenons l'exemple suivant :

```
#include<stdio.h>
#include<stdlib.h>
int *t, i;

int main(){

 t = (int *)malloc(5 * sizeof(int));
 if (t==NULL)
 printf("pas assez de mémoire\n");
 else{
  for(i=0; i<5; i++)
  t[i] = i * i;
  for(i=0; i<5; i++)
   printf("%d ",t[i]);
   printf("\n");
  free(t);
 }
 t = (int *)malloc( 10 * sizeof(int));
 if (t==NULL)
```

```
 printf("pas assez de mémoire\n");else{
 for(i=0; i<10; i++)
 t[i] = i * i;
 for(i=0; i<10; i++)
  printf("%d  ",t[i]);
 free(t);
}
return 0;
}
```

Explication:

- Dans cet exemple, t est un pointeur vers un entier.
- Après l'appel à la fonction malloc dans l'instruction *t = malloc(5*sizeof(int))*, t contient l'adresse d'une zone mémoire dont la taille est 5 fois la taille d'un entier. La variable t devient ainsi un tableau de 5 entiers qu'on peut utiliser comme n'importe quel tableau d'entiers.
- Si t n'est pas NULL, ce qui signifie qu'il y avait assez de mémoire disponible, alors on peut accéder à n'importe quelle élément du tableau en écrivant t[i] (i étant bien sûr compris entre 0 et 4).
- Dans ce programme, on remplit les 5 cases du tableau t en mettant i*i dans la case i et on affiche ce tableau.
- Ensuite, on libère l'espace occupé par le tableau en appelant la fonction free(t). t devient alors un pointeur non initialisé et on a plus le droit d'accéder aux différentes cases du tableau qui a été détruit.
- On appelle ensuite la fonction *t = malloc(10*sizeof(int))*, t devient alors cette fois-ci un tableau à 10 cases (sauf si t est NULL) et on peut accéder à la case i en écrivant t[i] (i compris entre 0 et 9).
- Dans ce programme, on remplit les 10 cases du tableau t en mettant i*i dans la case i et on affiche ce tableau.
- On détruit ensuite le tableau t en appelant free (t).
- Lorsqu'on exécute le programme voici ce qu'on obtient à l'écran :

```
0 1 4 9 16
0 1 4 9 16 25 36 49 64 81
```

Bien qu'elles soient moins fondamentales que les précédentes, les deux fonctions `calloc` et `realloc` peuvent s'avérer pratiques dans certaines circonstances.

La fonction calloc()

Nous pouvons utiliser la fonction **calloc()** pour allouer de la mémoire dynamiquement , cette fonction posséde la signature suivante :

```
void * calloc (size_t nb_bloc, size_t taille) ;
```

La fonction alloue `nb_bloc` blocs de taille `taille`, elle est donc presque équivalente à l'appelsuivant :

```
malloc(nb_bloc * taille);
```

La seule différence réside dans le contenu des cases qui sont allouée. En effet, avec **malloc()**,

le contenu est totalement aléatoire tandis qu'avec **calloc()** les cases contiennent des valeurs nulles (tous les octets du bloc alloué sont mis à 0). Ceci est très utile pour initialiser rapidement un tableau de nombre par exemple. Si ce n'est ce point de détail, il n'y a aucune autre différence.

Une zone allouée par **calloc()** ne peut être libérée qu'en une seule fois par **free()**. Il n'est pas question d'essayer de n'en libérer qu'un "morceau" (comme nous l'avons déjà dit, les octets alloués par calloc forment un tout pour le système).

Exemple **:**

```
#include <stdlib.h>

int * p;
p = calloc (3, sizeof(int));
```

La fonction realloc()

La dernière fonction d'allocation dynamique de la mémoire est **realloc()**, cette dernière possède la signature suivante :

```
void * realloc (void * pointeur, size_t taille);
```

Cette fonction permet de modifier la taille d'une zone préalablement allouée (par malloc, calloc ou realloc). Le **pointeur** doit être l'adresse de début de la zone dont on veut modifier la taille. Quant à **taille** de type **size_t**, elle représente la nouvelle taille souhaitée. Si une erreur se produit, la fonction retourne le pointeur NULL. Si en revanche, tout s'est bien passé, la fonction renvoie un pointeur vers l'adresse du nouveau bloc réalloué. La fonction réalloue le bloc mémoire tout en gardant le contenu de ce qui se trouvait dans le bloc précédent. La fonction ne fait donc qu'un changement de taille.

Lorsque la nouvelle taille demandée est supérieure à l'ancienne, le contenu de l'ancienne zone est conservé (quitte à le recopier si la nouvelle adresse est différente de l'ancienne). Dans le cas où la nouvelle taille est inférieure à l'ancienne, le début de l'ancienne zone (c'est-à-dire taille octets) verra son contenu inchangé.

Ceci est donc utile pour un tableau dynamique : en effet, on peut ajouter ou enlever une case à la fin du tableau sans le modifier. Voici un exemple illustratif:

Exemple **:**

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

 int * t,
 i;int
 main(){
  t = (int *)calloc (3, sizeof(int));
  t[0] = 1;
  t[1] = 2;
  t[2] = 3;
  t = (int *)realloc (t, 4 * sizeof(int));
  t[3] = 4;
  for ( i = 0 ; i <= 3 ; i++ )
   printf("%d ", t[i]);
  return 0;
 }
```

Lorsqu'on exécute le programme voici ce qu'on obtient à l'écran :

```
1 2 3 4
```

Exemple 2 **:** La fonction **realloc** peut être utilisée pour allouer de la mémoire.

```
#include<stdio.h>
#include<stdlib.h>
char * t;int i;
 int main(){
 t = NULL;
  t = (char *)realloc (t, 2 * sizeof(char) );
  t[0] = 'a';
  t[1] = 'b';
  for ( i = 0 ; i < 2 ; i++ )
   printf("%c ", t[i]);
  return 0;
 }
```

Les fonctions new et delete

En C++, il existe une autre fonction qui permet l'allocation dynamique de la mémoire, la fonction **new** possède la syntaxe suivante :

```
new type[taille];
```

new renvoie un tableau dont la taille est taille éléments, chaque élément étant de type **type**. S'il n'y a pas assez de mémoire, new renvoie NULL. Sinon new renvoie un élément dont le type est type *.

Exemple **:**

```
int *t ;
t = new int[3];
```

Pour libérer un espace mémoire alloué par **new**, nous utilisons la fonction **delete**, cette derniére possède la syntaxe suivante :

```
delete []t;
```

Si t est un tableau qui a été créé en utilisant la méthode précédente, l'utilisation de **delete** détruit ce tableau. Le tableau t n'est donc plus utilisable dès qu'on a utilisé l'opérateur **delete**.

Exemple **:**

```
#include<iostream>
using namespace
std;

int *t, i;
int
main(){
 t = new int[5];if (t == NULL)
  cout <<"pas assez de mémoire" <<endl;
  else{
  for ( i=0 ; i<5 ; i++ )
```

```
 t[i] = i * i;
 for ( i=0 ; i<5 ; i++ )
  cout <<t[i] << " ";
 cout <<endl;
 delete [] t;
 t = new int[10];
 if (t == NULL)
  cout <<"pas assez de mémoire"<<endl;
  else{
  for ( i=0 ; i<10 ; i++ )
  t[i] = i * i;
  for ( i=0 ; i<10 ; i++ )
  cout <<t[i] <<" ";
  cout <<endl;
  delete [] t;
 }
}
return 0;
}
```

Explication:

- Dans cet exemple, t est un pointeur vers un entier.
- Grâce à l'opérateur new, on transforme t en un tableau de 5 entiers grâce à l'instruction. t=new int[5]; On remplit ce tableau et on affiche le contenu des 5 cases de ce tableau. On détruit ce tableau en utilisant l'instruction delete [ ]t;
- On crée ensuite un autre tableau comportant cette fois 10 cases, on le remplit, on l'affiche et on le détruit de la même manière que précédemment.
- Lorsqu'on exécute le programme voici ce qu'on obtient à l'écran :

```
0 1 4 9 16
0 1 4 9 16 25 36 49 64 81
```

Chapitre III.

Structures et Listes chainées

III.1 Introduction

Avant d'entamer les structures de données complexes représentés par les types abstraits de données, il est primordiale de connaitre le type ***structure***.

Nous avons déjà vu comment le tableau permettait de désigner sous un seul nom un ensemble de valeurs de même type, chacune d'entre elles étant repérée par un indice.

La structure, quant à elle, va nous permettre de désigner sous un seul nom un ensemble de valeurs pouvant être de types différents. L'accès à chaque élément de la structure (nommé ***champ***) se fera, cette fois, non plus par une indication de position, mais par son nom au sein de la structure.

III.2 Le type structure

Déclaration d'une structure

Soit la déclaration de structure suivante :

```
struct prod{
  int numero;
  int qte;
  float prix;
};
```

Celle-ci définit un ***modèle de structure*** mais ne réserve pas de variables correspondantes à cette structure. Ce modèle s'appelle ici ***prod*** et il précise le nom et le type de chacun des champs constituant la structure (numero, qte et prix).

Une fois un tel modèle défini, nous pouvons déclarer des variables du type correspondant (souvent, nous parlerons de structure pour désigner une variable dont le type est un modèle de structure). Par exemple :

```
struct prod art1;
```

Réserve un emplacement nommé art1 « de type ***prod*** » destiné à contenir deux entiers et un flottant. De manière semblable :

```
struct prod art1, art2;
```

Réserve deux emplacements art1 et art2 du type ***prod***.

Bien que ce soit peu recommandé, sachez qu'il est possible de regrouper la définition du modèle de structure et la déclaration du type des variables dans une seule instruction comme dans cet exemple :

```
struct prod {
  int numero;
  int qte;
  float prix;
 }art1, art2;
```

Utilisation d'une structure

Chaque champ d'une structure peut être manipulé comme n'importe quelle variable du type correspondant. La désignation d'un champ se note en faisant suivre le nom de la variable structure de l'opérateur par un « point » (.) suivi du nom de champ tel qu'il a été défini dans le modèle.

Voici un exemple utilisant le modèle ***prod*** et les variables art1 et art2 déclarées de ce type.

```
#include <stdio.h>
struct prod{
  int numero;
  int qte;
  float prix;
}art1, art2;

int main(){
 art1.numero = 15;
 art2.qte = 25;
 printf("%f\n", art1.prix);
 printf("%d\n", art2.qte);
 scanf("%f", &art2.prix);
 art1.numero = 0;
 art1.numero++;
 printf("%d\n", art1.numero);
 return 0;
}
```

Le programme permet de :

- Affecter la valeur `15` au champ `numero` de `art1.`
- Affecter la valeur `25` au champ `qte` de `art1.`
- Afficher la valeur du champ `prix` de `art1.`
- Afficher la valeur du champ `qte` de `art1.`
- Lire une valeur qui sera affectée au champ `prix` de `art2.`
- Affecter la valeur `0` au champ `numero` de `art1.`
- incrémenter de `1` la valeur du champ `numero` de `art1.`
- Afficher la valeur du champ `numero` de `art1.`

Il est possible d'affecter à une variable de type structure le contenu d'une autre variable définie à partir du **même modèle**. Par exemple, si les structures art1 et art2 ont été déclarées suivant le modèle ***prod*** défini précédemment, nous pourrons écrire : `art1 = art2;`

Une variable de type structure peut aussi être initialisée au moment de sa déclaration, voici un exemple d'initialisation de notre structure art1, au moment de sa déclaration :

```
struct prod art1 = {100, 285, 200.0} ;
```

Vous voyez que la description des différents champs se présente sous la forme d'une liste de valeurs séparées par des virgules, chaque valeur étant une constante ayant le type du champ correspondant.

En faisant usage de ***typedef***, les déclarations des structures art1 et art2 peuvent être réalisées comme suit :

```
struct prod{
 int numero;
 int qte;
 float prix;
```

```
};

typedef struct prod enreg;
enreg art1, art2;
```

Ou encore, plus simplement :

```
  typedef struct{
   int numero;
   int qte;
   float prix;
  }enreg;

enreg art1, art2;
```

Structure comportant de tableaux

Soit la déclaration suivante :

```
struct personne{
  char nom[30];
  char prenom [20];
  float heures [31];
} employe, courant;
```

Celle-ci réserve les emplacements pour deux variables nommées `employe` et `courant`. Ces dernières comportent trois champs :

- `nom` qui est un tableau de 30 caractères ;
- `prenom` qui est un tableau de 20 caractères ;
- `heures` qui est un tableau de 31 flottants.

A titre indicatif, voici un exemple d'initialisation d'une structure de type personne lors de sa déclaration :

```
struct personne employe = {"Mohamed", "Mohamed",
{8,7,8,6,8,0,0,8}};
```

III.3 Types abstraits de données

Introduction

Une bonne habitude de programmation est de concevoir des programmes modulaires, c'est-à-dire constitués de parties indépendantes les unes des autres, chacune pouvant être réutilisées dans d'autres programmes.

Lorsqu'on utilise un module, l'important est de connaître les opérations que l'on peut effectuer sur les données. La façon dont les opérations sont programmées n'a pas d'intérêt pour l'utilisateur.
Un **type abstrait de données (TAD)** (Abstract Data Type ou ADT) est une description d'un module comprenant :

1. Un type de données ;
2. Une description des opérations possibles sur ces données.

Un TAD est un ensemble de données organisé, et d'opérations sur ces données. Il est défini d'une manière indépendante de la représentation des données en mémoire qui doit fournir une description

détaillée des opérations que l'on peut effectuer sur les données, mais aucune information sur la façon dont les données sont stockées et comment les opérations sont implémentées.

L'abstraction consiste à penser à un objet en termes d'actions que l'on peut effectuer sur lui, et non pas en termes de représentation et d'implantation de cet objet.

Un type abstrait est composé de cinq champs :

- *TA* (Type Abstrait)
- *Utilise*
- *Opérations*
- *Pré-conditions*
- *Axiomes*

Le champ « TA » (Type Abstrait) contient le nom du type que l'on est en train de décrire et précise éventuellement si celui-ci n'est pas une extension d'un autre type abstrait. Par exemple, on écrira « TA : Pile » pour créer un type nommé Pile.

Le champ « Utilise » contient les types abstraits que l'on va utiliser dans celui que l'on est en train de écrire. Par exemple, le type abstrait ensemble que l'on définit va utiliser dans sa spécification le type abstrait Booléen, et on écrira « Utilise : Booléen ».

Le champ « Opérations » contient le prototypage de toutes les opérations. Par prototypage, on entend une description des opérations par leur nom.

Le champ « Pré-conditions » contient les conditions à respecter sur les arguments d'une fonction pour que celle-ci puisse avoir un comportement normal.

Le champ « Axiomes » contient une série d'axiomes pour décrire le comportement de chaque opération d'un type abstrait. Chaque axiome est une proposition logique vraie.

Exemple :

TA ensemble
Utilise élément, booléen
Opérations
est vide : ensemble ⟶ booléen
ensemble vide : ⟶ ensemble
appartient : élément * ensemble ⟶ booléen
ajouter : élément * ensemble ⟶ ensemble
enlever : élément * ensemble ⟶ ensemble
choisir : ensemble ⟶ élément

On dit que le type abstrait ***ensemble*** est définit en utilisant le concept de signature.

Définition : La signature d'un type de donnée décrit la syntaxe du type (nom des opérations et type de leurs arguments) mais elle ne définie pas les propriétés des opérations du type.

III.4 Listes (concepts et implémentation)

Définition

Une liste est un ensemble d'objets de même type constituant les éléments de la liste. Les éléments sont chaînés entre eux et on peut facilement ajouter ou extraire un ou plusieurs éléments. Une liste simple est une structure de données telle que chaque élément contient :

- des informations caractéristiques de l'application (les caractéristiques d'une personne par exemple).
- un pointeur vers un autre élément ou une marque de fin s'il n'y a pas d'élément successeur.

La liste est la forme la plus usuelle des structures de données. Une liste est une suite d'éléments finie, éventuellement vide. L'ordre des éléments dans une liste est sans importance. Par contre, la place de chaque élément est importante.

Si $[e_1, e_2, \ldots, e_n]$ représentent les éléments d'une liste, cette liste est notée : $L = [e_1, e_2, \ldots, e_n]$, l'élément e_2 succède à l'élément e_1 et précède e_3. La notion de place implique souvent des algorithmes séquentiels sur les éléments d'une liste.

Opérations

En plus du traitement séquentiel des éléments en suivant l'ordre des places, les opérations de base que l'on effectue sur les listes sont les suivantes :

- Accéder au $k^{ème}$ élément.
- Supprimer le $k^{ème}$ élément.
- Insérer un nouvel élément à la $k^{ème}$ place.

La signature du type liste est la suivante :

TA Liste
Utilise élément, entier
Opérations
liste_vide : ⟶ Liste
acces : Liste * entier ⟶ Place
contenu : Place ⟶ élément
longueur : Liste ⟶ entier
insérer : Liste * entier * élément ⟶ Liste
supprimer : Liste * entier ⟶ Liste
$i^{ème}$: Liste * entier ⟶ élément
succ : Place ⟶ Place

Les opérations ci-dessus ne sont pas définies partout, on a les pré-conditions suivantes où L est de sorte Liste, k est de sorte entier et e est de sorte élément:

acces(L, k) **est définie ssi** $1 \leq k \leq$ longueur(L)
supprimer(L, k) **est définie ssi** $1 \leq k \leq$ longueur(L)
insérer(L, k, e) **est définie ssi** $1 \leq k \leq$ longueur(L) +1
$i^{ème}$(L, k) **est définie ssi** $1 \leq k \leq$ longueur(L)

En supposant les pré-conditions vérifiées, on peut écrire les propriétés suivantes :

- longueur (liste_vide) = 0
- longueur (insérer (L, k, e)) = longueur (L) + 1
- longueur (supprimer (L, k, e)) = longueur (L) – 1

- $i^{ème}$ (insérer (L, k, e), k) = e
- Si $1 \leq i < k$ alors $i^{ème}$ (insérer (L, k, e), i) = $i^{ème}$ (L, i)

Si on insère un élément e dans une liste L au rang k, k plus grand que i, la place de l'élément de rang i n'est pas modifiée.

- Si $k < i \leq$ longueur (L) + 1 alors $i^{ème}$ (insérer (L, k, e), i) = $i^{ème}$ (L, i - 1)

Si on insère un élément e dans une liste L au rang k, k plus petit que i, la place de l'élément de rang i est celle qui se trouvait au rang i-1.

- Si $1 \leq i < k$ alors $i^{ème}$ (supprimer (L, k), i) = $i^{ème}$ (L, i)

Si l'on supprime un élément de la liste L au rang k, k plus grand que i, la place de l'élément i n'est pas modifiée.

- Si $k \leq i <$ longueur (L) alors $i^{ème}$ (supprimer (L, k), i) = $i^{ème}$ (L, i + 1)

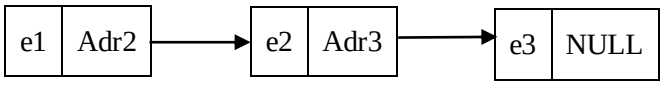
Si l'on supprime dans la liste L, l'élément de rang k, k plus petit que i, la place de l'élément i est celle qui se trouvait au rang i+1

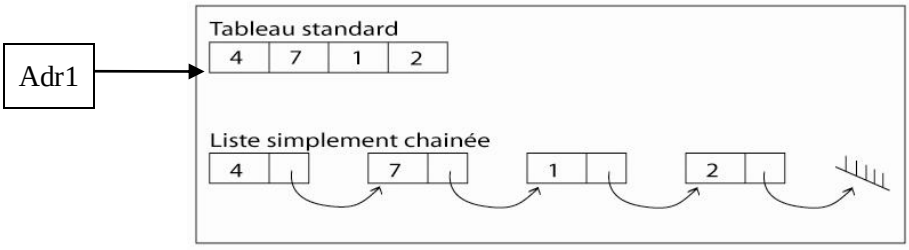
III.5 Les listes chaînées

Lorsque vous créez un tableau, les éléments de celui-ci sont placés de façon contiguë en mémoire. Pour pouvoir le créer, il vous faut connaître sa taille. En effet, pour insérer un élément dans un tableau, il faut d'abord déplacer tous les éléments qui sont en amont de l'endroit où l'on souhaite effectuer l'insertion, ce qui peut prendre un temps non négligeable proportionnel à la taille du tableau et à l'emplacement du futur élément. Il en est de même pour la suppression d'un élément; il faut décaler tous les éléments se trouvant après l'élément à supprimer bien sûr ceci ne s'applique pas en cas d'ajout ou de suppression en fin de tableau. Une liste chaînée est différente dans le sens où les éléments de votre liste sont répartis dans la mémoire et reliés entre eux par des pointeurs. Vous pouvez ajouter et enlever des éléments d'une liste chaînée à n'importe quel endroit, à n'importe quel instant.

On utilise des pointeurs pour une représentation chaînée afin de lier les éléments successifs de la liste. Ainsi, la liste est identifiée par l'adresse de son premier élément.

Exemple





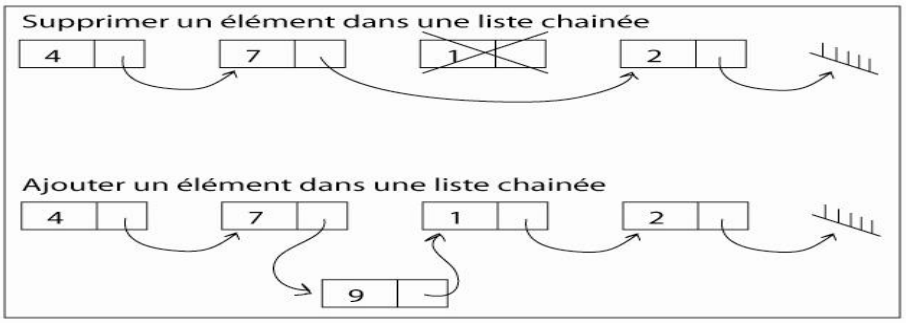
- Dans une liste chaînée, la taille est inconnue au départ, la liste peut avoir autant d'éléments que votre mémoire le permet. Il est en revanche impossible d'accéder directement à l'élément i de la liste chainée, pour ce faire, il vous faudra traverser les i-1 éléments précédents de la liste.
- Pour déclarer une liste chaînée, il suffit de créer le pointeur qui va pointer sur le premier élément de votre liste chaînée, aucune taille n'est donc à spécifier.
- Il est possible d'ajouter, de supprimer, d'intervertir des éléments d'une liste chaînée sans avoir à déplacer les éléments de la liste, mais en manipulant simplement leurs pointeurs.

Chaque élément d'une liste chaînée est composé de deux parties :

- Une ou plusieurs données que vous voulez stocker.
- L'adresse de l'élément suivant, s'il existe, s'il n'y a plus d'élément suivant, alors l'adresse sera `NULL`, et désignera le bout de la liste.

On accède à la liste par un pointeur L sur la première cellule, puis en parcourant la liste d'une cellule à l'autre en suivant les pointeurs suivant. Le dernier pointeur suivant vaut `NULL`, ce qui indique la fin de la liste.

Voilà deux schémas qui expliquent l'ajout et la suppression d'un élément d'une liste chaînée. Remarquez le symbole en bout de chaîne qui signifie que l'adresse de l'élément suivant ne pointe sur rien, c'est-à-dire sur `NULL`.



Donc une liste simplement chainée peut être représentée par une structure suivant cette définition:

```
struct element{
  type_elt info;
  element * suivant;
};
```

Type_elt est le type des éléments de la liste.

La liste vide est représentée par le pointeur `NULL`. Si la liste n'est pas vide, la liste est représentée par le pointeur qui pointe sur le premier élément.

Soit `e1` une variable de type `element`, l'entier `info` représente la valeur de cet élément et `suivant` représente l'adresse de l'élément suivant de `e1`.
Après avoir déclarer la liste, Il est important de toujours initialiser la liste chaînée à `NULL`. Ce qui dit que la liste ne contient aucun élément, ceci peut ce faire en utilisant le programme suivant:

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

```
element* initialiser(){
 return NULL;
}

int main(){
 element* ptrNoeud;
 ptrNoeud = initialiser();
 return 0;
}
```

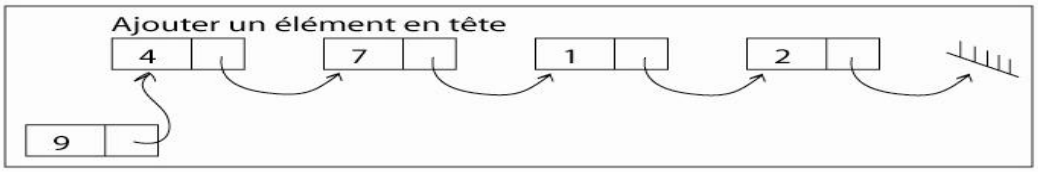
Maintenant que nous savons comment déclarer une liste chaînée, il serait intéressant de passer aux différentes opérations qu'on peut effectuer sur les listes.

Ajouter un élément

Lorsque nous voulons ajouter un élément dans une liste chaînée, il faut savoir où l'insérer. Les deux ajouts génériques des listes chaînées sont les ajouts en tête, et les ajouts en fin de liste. Nous allons étudier ces deux moyens d'ajouter un élément à une liste.

Ajouter en tête

Lors d'un ajout en tête, nous allons créer un élément, lui assigner la valeur que l'on veut ajouter, puis pour terminer, raccorder cet élément à la liste passée en paramètre. Lors d'un ajout en tête, on devra donc assigner à `suivant` l'adresse du premier élément de la liste passé en paramètre. Visualisons tout ceci sur un schéma :




```
element* ajouterEnTete(element* debut, int valeur){
 element* nouvelElement;
 nouvelElement = (element*)malloc(sizeof(element));
 nouvelElement->info = valeur;
 nouvelElement->suivant = debut;

 return nouvelElement;
}
```

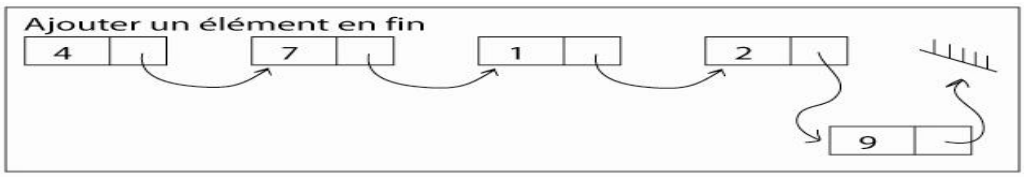
Explication

- On crée un nouvel élément ;
- On assigne la valeur au nouvel élément ;
- On assigne l'adresse de l'élément suivant au nouvel élément ;
- On retourne la nouvelle liste, c'est-à-dire le pointeur sur le premier élément.

Remarque: Ne pas confondre l'utilisation du point (.) et l'utilisation de la flèche (->) pour accéder aux champs d'une structure. On utilise le point pour une variable de type structure, et une flèche pour une variable de type pointeur sur structure.

Ajouter en fin de liste

Cette fois-ci, c'est un peu plus compliqué. Il nous faut tout d'abord créer un nouvel élément, lui assigner sa valeur, et mettre l'adresse de l'élément suivant à `NULL`. En effet, comme cet élément va terminer la liste nous devons signaler qu'il n'y a plus d'élément suivant. Ensuite, il faut faire pointer le dernier élément de la liste originale sur le nouvel élément que nous venons de créer. Pour ce faire, il faut créer un pointeur temporaire sur le type `element` qui va se déplacer d'élément en élément, et regarder si cet élément est le dernier de la liste. Un élément sera forcément le dernier de la liste si `NULL` est assigné à son champ suivant.




```
element* ajouterEnFin(element * debut, int valeur){
 element* nouvelElement,* temp;

 nouvelElement = (element*)malloc(sizeof(element));
 nouvelElement->info = valeur;
 nouvelElement->suivant = NULL;
 if(debut == NULL)
  return nouvelElement;
 else{
  temp = debut;
  while(temp->suivant != NULL)
   temp = temp->suivant;
  temp->suivant = nouvelElement;
  return debut;
 }
}
```

Explication

- On crée un nouvel élément ;
- On assigne la valeur au nouvel élément ;
- On ajoute en fin, donc aucun élément ne va suivre ;
- Si la liste est vide il suffit de renvoyer l'élément créé, Sinon, on parcourt la liste à l'aide d'un pointeur temporaire et on indique que le dernier élément de la liste est relié au nouvel élément.

Supprimer un élément

Les deux suppressions génériques des listes chaînées sont les suppressions en tête, et les suppressions en fin de liste. Nous allons étudier ces deux moyens de supprimer un élément d'une liste.

Supprimer un élément en tête

Il s'agit là de supprimer le premier élément de la liste. Pour ce faire, il nous faudra utiliser la fonction `free`. Si la liste n'est pas vide, on stocke l'adresse du premier élément de la liste après suppression (i.e. l'adresse du 2ème élément de la liste originale), on supprime le premier élément, et

on renvoie la nouvelle liste. Attention quand même à ne pas libérer le premier élément avant d'avoir stocké l'adresse du second, sans quoi il sera impossible de le récupérer.

```
element * supprimerElementEnTete(element * debut){
 element* aRenvoyer;
 if(debut != NULL){
  aRenvoyer = debut->suivant;
  free(debut);
  return aRenvoyer;
 }
 else
  return NULL;
}
```

Explication

- Si la liste est non vide, on se prépare à renvoyer l'adresse de l'élément qui se trouve en 2ème position et on libère le premier élément ;
- Sinon, on retourne `NULL`;

Supprimer un élément en fin de liste

Cette fois-ci, il va falloir parcourir la liste jusqu'à son dernier élément, indiquer que l'avant-dernier élément va devenir le dernier de la liste et libérer le dernier élément pour enfin retourner le pointeur sur le premier élément de la liste d'origine.

```
element * supprimerElementEnFin(element * debut){
 element* tmp,* ptmp;

 if(debut == NULL)
  return NULL;
 else{
  if(debut->suivant == NULL){
   free(debut);
   return NULL;
  }
  else{
   tmp = debut;
   ptmp = debut;
   while(tmp->suivant != NULL){
    ptmp = tmp;
    tmp = tmp->suivant;
   }
   ptmp->suivant = NULL;
   free(tmp);
   return debut;
  }
 }
}
```

- Si la liste est vide, on retourne `NULL` ;
- Si la liste contient un seul élément, on le libère et on retourne `NULL` (la liste est maintenant vide) ;
- Si la liste contient au moins deux éléments. Tant qu'on n'est pas au dernier élément `ptmp` stock l'adresse de `tmp` ;

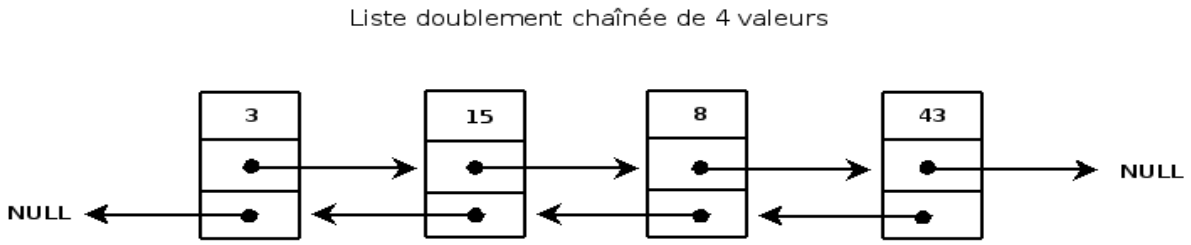
- On déplace `tmp` (`ptmp` garde l'ancienne valeur de `tmp`) ;

A la sortie de la boucle, `tmp` pointe sur le dernier élément, et `ptmp` sur l'avant-dernier. On indique que l'avant-dernier devient la fin de la liste et on supprime le dernier élément.

III.6 Les listes doublement chaînées

Une liste doublement chaînée est une liste dont chaque élément peut accéder à l'aide de pointeurs aux éléments positionnés immédiatement avant et après lui dans la liste. Le chaînage se fait donc dans les deux sens, ce qui permet de parcourir la liste en avant comme en arrière, ce qui n'était pas possible avec la liste simple. De plus s'il est aisé d'ajouter des éléments à chaque extrémité d'une liste simple, cela l'est beaucoup moins quand il s'agit de retirer l'élément en fin de liste (dans le sens du chaînage). La liste doublement chaînée nous en facilitera la tâche.

Comme pour la liste simple les éléments de la liste sont chaînés entre eux à l'aide de pointeurs sur des éléments du même type qu'eux. Dans le cas de la liste chaînée double, chaque élément aura un pointeur sur l'élément précédent et un pointeur sur l'élément suivant.



Nous allons nous aider d'un exemple simple. Comme dans la liste simple nous mémoriserons seulement un entier dans chaque élément. Donc une liste simplement chainée peut être représentée par une structure suivant cette définition:

```
struct element{
  int info;
  element * precedent;
  element * suivant;
};
```

III.7 Les listes circulaires

La liste circulaire est une sorte de liste simplement ou doublement chaînée, qui comporte une caractéristique supplémentaire pour le déplacement dans la liste, "**elle n'a pas de fin**". Pour rendre la liste sans fin, le pointeur suivant du dernier élément pointera sur le 1er élément de la liste au lieu de la valeur NULL, que nous avons vu dans le cas des listes simplement et doublement chaîné.

Dans les listes circulaires, nous n'arriverons jamais à une position depuis laquelle nous ne pourrons plus nous déplacer. En arrivant au dernier élément, le déplacement recommencera au premier élément. En bref, il s'agit d'une rotation. Donc, une liste circulaire est représentée par la même structure qu'une liste simple, sauf que le pointeur suivant du dernier élément pointera sur le 1er élément.

III.8 Les listes simplement chainées en représentation contiguë

Dans une représentation contiguë des listes, les éléments de la liste sont mémorisés dans un tableau dont la i^{ème} case est la i^{ème} place de la liste. On a besoin également d'une variable pour mémoriser le

nombre significatif d'éléments du tableau (à cause des opérations d'insertion et de suppression). On regroupe ces deux informations dans un enregistrement :

```
 struct liste {
  int taille_max;
  int taille;
  int *tab;
 };
```

Soit L une variable de type Liste, si L représente la liste vide nous avons *L.taille = 0*.
La fonction d'accès à un élément de la liste est la fonction d'accès aux éléments du tableau, l'accès à un élément s'écrit donc: *L.tab[i]*.

```
liste liste_vide(int n){
 liste l;
 l.tab = (int *)malloc(n * sizeof(int));
 l.taille_max = n;
 l.taille = 0;
 return l;
}
```

Les seules opérations complexes à programmer sont l'insertion et la suppression car il est nécessaire de décaler les éléments du tableau :

```
void ajouter(liste * l, int e, int i){
int j, n;

 n = l->taille;
 if ((n < l->taille_max) && (i > 0) && (i < n + 1)){
  for(j = n - 1; j >= i; j--)
   l->tab[j + 1] = l->tab[j];
  l->tab[i] = e;
  l->taille = l->taille + 1;
 }
 else
  printf("indice incorrect\n");
}

void supprimer(liste * l, int i){
int j, n;

 n = l->taille;
 if ((i >= 0) && (i < n)){
  for(j = i; j < n - 1; j++)
   l->tab[j] = l->tab[j + 1];
  l->taille = l->taille - 1;
 }
 else
  printf("indice incorrect\n");
}
```

Chapitre IV.

Piles et Files

IV.1 Les piles (concepts et implémentation)

Définition

Les piles constituent des **structures de données**. Elles vont, comme leur nom l'indique, nous permettre de stocker diverses données, comme pourrait le faire un tableau.

Une pile est gérée suivant la politique **LIFO** (**L**ast **I**n **F**irst **O**ut) (dernier arrivé premier servi), ce qui signifie en clair que les derniers éléments à être ajoutés à la pile seront les premiers à être récupérés. Il est possible de comparer cela à une pile d'assiettes. Lorsqu'on ajoute une assiette en haut de la pile, on retire toujours en premier celle qui se trouve en haut de la pile, c'est-à-dire celle qui a été ajoutée en dernier, sinon tout le reste s'écroule.

Dans les piles, les insertions et les suppressions se font à une seule extrémité appelée ***sommet de pile***.

Opérations

Comme pour les listes, plusieurs opérations peuvent être effectuées sur les piles dont les plus importantes sont:

- Créer une pile vide;
- Tester si une pile est vide;
- Accéder à l'information contenue dans le sommet de la pile;
- Ajouter un élément au sommet de la pile (empiler);
- Supprimer l'élément qui se trouve au sommet de la pile (dépiler).

La signature du type pile est la suivante :

TA Pile
Utilise élément, booléen
Opérations
pile_vide : ⟶ Pile
est vide : Pile ⟶ booléen
empiler : Pile * élément ⟶ Pile
dépiler : Pile ⟶ Pile
sommet : Pile ⟶ élément

Les opérations ci-dessus ne sont pas définies partout, on a les pré-conditions suivantes où P est de sorte Pile et e est de sorte élément:

dépiler(P) **est définie ssi** est vide(P) = faux
sommet(P) **est définie ssi** est vide(P) = faux

En supposant les pré-conditions vérifiées, ces opérations vérifient les axiomes suivants :

dépiler(empiler(P, e)) = P
sommet(empiler(P, e)) = e
est vide(pile_vide) = vrai
est vide (empiler(P, e))= faux

IV.2 Représentation des piles

1. Représentation contiguë

Dans cette représentation, les éléments de la pile sont rangés dans un tableau. De plus, il faut conserver l'indice du sommet de la pile et la taille maximale du tableau utilisé. Donc, le modèle de structure suivant est utilisé pour représenté une pile :

```
struct pile{
 int taille_max;
 int sommet;
 int *tab;
};
```

Soit P de type pile. Si p est vide, nous avons `p.sommet = -1`, les opérations du type abstrait Pile sont données comme suit :

La fonction permettant de créer une pile vide est la suivante :

```
pile pile_vide(int n){
 pile p;

 p.tab = (int *)malloc(sizeof(int) * n);
 p.taille_max = n;
 p.sommet = -1;
 return p;
}
```

Le sommet de la pile est le dernier élément entré, qui est le dernier élément du tableau. La fonction permettant d'accéder au sommet de la pile est donc la suivante :

```
int sommet(pile p){

if (p.sommet == -1)
 printf("La pile est vide\n");
else
 return p.tab[p.sommet];
}
```

Pour ajouter un élément au sommet de la pile, il suffit d'incrémenter la valeur de `p.sommet` et d'affecter la valeur à ajouter dans le nouveau sommet.

```
void empiler(pile * p, int e){

if (p->sommet == p->taille_max - 1)
 printf("La pile est pleine\n");
else{
  p->sommet = p->sommet + 1;
  p->tab[p->sommet] = e;
 }
}
```

La fonction **depiler()** supprime le sommet de la pile en cas de pile non vide.

```
int depiler(pile * p){
```

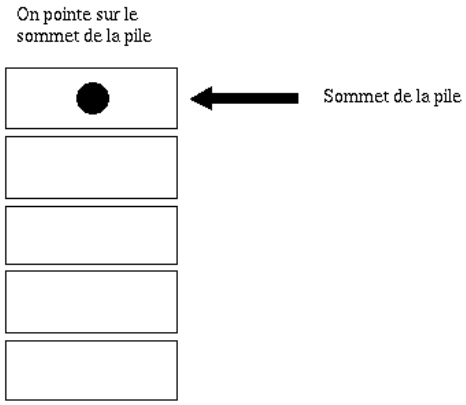
```
int elem;

if (p->sommet == -1)
 printf("La pile est vide\n");
else{
  elem = p->tab[p->sommet];
  p->sommet = p->sommet - 1;
  return elem;
 }
}
```

2. Représentation chaînée

Tout d'abord, nous allons commencer par définir le modèle de structure qui représente une pile, l'implémentation des piles est basée sur les **listes simplement chaînées**. Chaque élément de la pile pointera vers l'élément précédent. La liste pointera toujours vers le sommet de la pile. Voici donc la structure qui constituera notre pile :

```
struct pile{
 int donnee;
 pile * precedent;
};
```

Chaque case d'une pile représente un élément. Comme vous le voyez (voir figure suivante), les cases sont en quelque sorte emboîtées les unes sur les autres. Le pointeur est alors représenté par le jeton noir, figurant tout en haut de le pile, car, rappelez-vous, notre pointeur devra toujours pointer vers le sommet de la pile, Enfin, on peut dire que les piles sont un cas particulier des listes chaînées. En effet, lors de l'***ajout*** d'un nouvel élément, celui-ci se fera en ***fin de liste***. Identiquement, l'élément qui sera ***effacé*** en premier sera l'élément se trouvant en ***fin de liste***.



Ajout d'un nouvel élément

Lors de l'ajout d'un élément dans une pile, nous allons créer un élément, lui assigner la valeur que l'on veut ajouter, puis nous devrons assigner au champ `precedent` du sommet de la pile l'adresse du nouvel élément. Ceci dit que le nouvel élément est devenu le sommet de la pile.

```
pile * empiler(pile * p, int e){
 pile * p_nouveau;
```

```
 p_nouveau = (pile *) malloc(sizeof (pile));
 if (p_nouveau != NULL){
  p_nouveau->donnee = e;
  p_nouveau->precedent = p;
  return p_nouveau;
 }
}
```

Explication

- On crée un nouvel élément de type Pile ;
- On vérifie que l'élément a bien été créé ;
- On assigne à la donnée de cet élément la donnée que l'on veut ajouter ;
- On fait pointer cet élément sur le sommet de la pile ;
- On retourne le nouveau sommet de la pile ;

Suppression d'un élément

Dans une pile, nous supprimons toujours l'élément qui se trouve en sommet de pile, pour ce faire, il nous faudra utiliser la fonction free. Si la liste n'est pas vide, on stocke l'adresse du sommet de pile après suppression (i.e. l'adresse de l'élément précédent du sommet de la pile originale).

```
int depiler(pile ** p){
 pile * temp;
 int elem;

 if(*p != NULL){
  elem = (*p)->donnee;
  temp = (*p)->precedent;
  free(*p);
  *p = temp;
  return elem;
 }
 else
  printf("La pile est vide");
}
```

Explication

- Vérifier si la pile n'est pas vide ;
- Si elle ne l'est pas, stockez dans un élément temporaire l'avant-dernier élément de la pile ;
- Supprimer le dernier élément ;
- Faire pointer la pile vers notre élément temporaire ;
- On retourne le nouveau sommet de la pile ;

IV.3 Les files (concepts et implémentation)

Définition

Une file est une structure de données dans laquelle on insère de nouveaux éléments à la fin (queue) et où on enlève des éléments au début (tête de file). L'application la plus classique est la file d'attente, et elle sert beaucoup en simulation. Elle est aussi très utilisée aussi bien dans la vie courante que dans les systèmes informatiques. Par exemple, elle modélise la file d'attente des clients devant un guichet, les travaux en attente d'exécution dans un système de traitement par lots, ou encore les messages en

attente dans un commutateur de réseau téléphonique. On retrouve également les files d'attente dans les programmes de traitement de transactions telle que les réservations de sièges d'avion ou de billets de théâtre.

Noter que dans une file, le premier élément arrivé est le premier servis, Une file est donc gérée suivant la politique **FIFO** (**F**irst **I**n **F**irst **O**ut), ce qui signifie en clair que les premiers éléments à être ajoutés à la file seront les premiers à être récupérés. Par conséquent, l'ajout d'un élément se fera en fin de file. Le retrait d'un élément se fera en début de file.

Opérations

Plusieurs opérations peuvent être effectuées sur les files dont les plus importantes sont :

- Créer une file vide ;
- Tester si une file est vide ;
- Accéder à l'information contenue dans la tête de file ;
- ajouter un élément en queue de file (enfiler) ;
- Supprimer l'élément qui se trouve en tète file (défiler).

La signature du type file est la suivante :

TA File
Utilise élément, booléen
Opérations
file_vide : ⟶ File
est vide : File ⟶ booléen
enfiler : File * élément ⟶ File
défiler : File ⟶ File
premier : File ⟶ élément

Les opérations premier et défiler ne sont définie que si la file n'est pas vide.

premier(F) **est définie ssi** est vide(F) = faux
défiler(F) **est définie ssi** est vide(F) = faux

En supposant les pré-conditions vérifiées, ces opérations vérifient les axiomes suivants, pour tous F de sorte File et e de sorte élément:

premier(enfiler(F, e)) = e
premier(enfiler(F, e)) = premier(F)
défiler(enfiler(F, e)) = enfiler(défiler(F), e)

IV.4 Représentation des files

1. Représentation chaînée

Pour la gestion par listes chaînées, on introduit un pointeur sur la tête de file et un pointeur sur la queue de file. Ceci permet de faire les insertions en queue de file sans avoir à parcourir la liste pour trouver l'adresse de la dernière cellule. Voici donc la structure qui constituera notre file:

```
struct cellule{
 int donnee;
 cellule * suivant;
};
```

```
struct file{
 cellule * tete;
 cellule * queue;
};
```

Nous Remarquons alors que, cette fois-ci, nous utilisons un pointeur vers l'élément suivant et non plus vers l'élément précédent comme pour les piles. Cela s'explique par le fait que nous pointons à la base de la file, c'est-à-dire sur le premier élément de la file. La création d'une file vide se fait par la fonction suivante :

```
file initialiser(){
 file filevide;

 filevide.tete = NULL;
 return filevide;
}
```

Ajout d'un nouvel élément

Lors de l'ajout d'un élément dans une file, nous allons créer un élément, lui assigner la valeur que l'on veut ajouter, puis nous devrons assigner au champ `suivant` du dernier élément de la file l'adresse du nouvel élément. Ceci dit que le nouvel élément est devenu le dernier élément de la file.

```
file enfiler(file f, int e){
 cellule * p_nouveau;

 p_nouveau = (cellule *)malloc(sizeof (cellule));
 if (p_nouveau != NULL){
  p_nouveau->suivant = NULL;
  p_nouveau->donnee = e;
  if (f.tete == NULL){
   f.tete = p_nouveau;
   f.queue = p_nouveau;
  }
  else{
   f.queue->suivant = p_nouveau;
   f.queue = p_nouveau;
  }
 }
 return f;
}
```

Explication

- On crée un nouvel élément de type File;
- On vérifie que le nouvel élément a bien été créé;
- On fait pointer l'élément suivant à cet élément vers NULL;
- On assigne à la donnée de cet élément la donnée que l'on veut ajouter;
- Si la file est vide, alors on fait pointer la tete et la queue vers l'élément que l'on vient de créer;
- Sinon, On fait pointer la queue vers le nouvel élément créé, et on modifie la queue de la file qui deviendra le nouvel élément créé.

Suppression d'un élément

Tout comme les piles, nous aurons aussi besoin d'une fonction permettant d'enlever un élément de la file. L'élément qui sera enlevé est le premier élément de la file. Le corps de la fonction nous permettant de réaliser cette opération est le suivant :

```
int defiler(file *f){
 cellule * tmp;
 int elem ;

 if (f->tete != NULL){
  elem = f->tete->donnee;
  tmp = f->tete->suivant;
  free(f->tete);
  f->tete = tmp;
  return elem;
 }
 else
  printf("La file est vide\n");
}
```

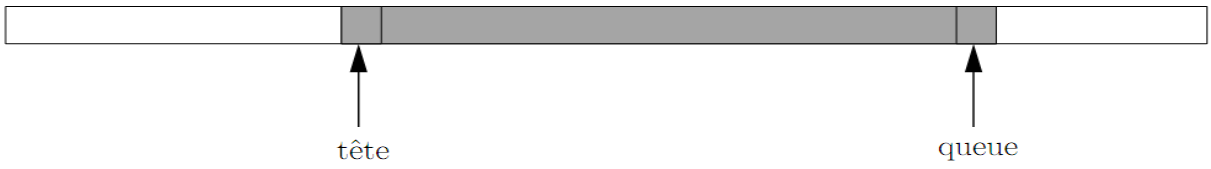
Explication

- On teste si la file n'est pas vide;
- On crée un élément temporaire pointant vers le deuxième élément de la file;
- On libère le premier élément de la file;
- On retourne le deuxième élément de la file originale;

2. Représentation contiguë

Dans cette représentation, les éléments de la file sont rangés dans un tableau. Donc, le modèle de structure suivant est utilisé pour représenté une file :

```
struct file{
  int taille_max;
  int tete;
  int queue;
  int *tab;
 };
```



Soit f de type file. Si f est vide, nous avons f.queue = -1, les opérations du type abstrait File sont données comme suit :

La fonction permettant de créer une file vide est la suivante :

```
file file_vide(int n){
 file f;
 f.tab = (int *)malloc(sizeof(int) * n);
 f.taille_max = n;
```

```
 f.queue = -1;
 f.tete = 0;
 return f ;
}
```

Le premier élément de la file est le dernier élément entré, qui est le premier élément du tableau. La fonction permettant d'accéder au premier élément de la file est donc la suivante :

```
int premier(file * f){

if(f->tete == f->queue+1)
 printf("La file est vide\n");
else
 return f->tab[f->tete];
}
```

Pour ajouter un élément enfin de la file, il suffit d'incrémenter la valeur de `f.queue` et d'affecter la valeur à ajouter dans la nouvelle position obtenue.

```
void enfiler(file * f, int e){

 if(f->queue == f->taille_max-1)
  printf("La file est pleine\n");
 else{
  f->queue = f->queue + 1;
  f->tab[f->queue] = e;
 }
}
```

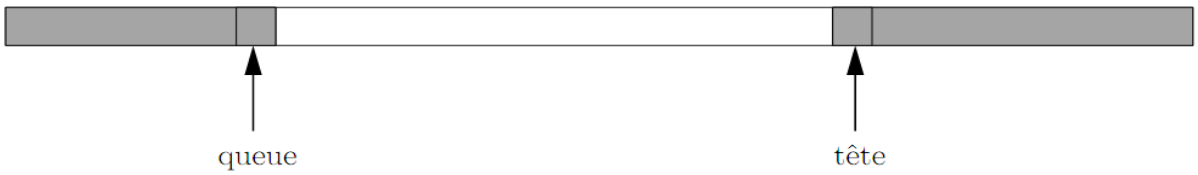
La fonction **defiler()** supprime le premier élément de la file en cas de file non vide.

```
int defiler(file *f){
int elem ;

 if(f->tete == f->queue+1)
  printf("La file est vide\n");
 else{
  elem = f->tab[f->tete];
  f->tete = f->tete + 1;
  return elem;
 }
}
```

Le problème dans cette gestion des files par tableaux est qu'à mesure qu'on insère et qu'on supprime des éléments, les indices de tête et de queue ne font qu'augmenter. L'utilisation d'une telle représentation est donc limitée, nous utilisons donc une gestion circulaire par tableau.

3. Représentation contiguë (gestion circulaire)

Dans une gestion circulaire des files, lorsque le bout du tableau est atteint, on réutilise la mémoire du début du tableau. Pour cela, on utilise des modulo sur les indices du tableau.



Dans cette représentation, les éléments de la file sont rangés dans un tableau. Donc, le modèle de structure suivant est utilisé pour représenté une file où `fin` représente l'indice du dernier élément de la file et tete représente l'indice du premier élément avant la file :

```
struct file{
  int taille_max;
  int tete;
  int fin;
  int *tab;
 };
```

La fonction permettant de créer une file vide est la suivante :

```
file file_vide(int n){
 file f;

 f.tab = (int *)malloc(sizeof(int) * (n + 1));
 f.taille_max = n + 1;
 f.fin = 0;
 f.tete = 0;
 return f ;
}
```

Le premier élément de la file est le dernier élément entré, qui est le premier élément du tableau. La fonction permettant d'accéder au premier élément de la file est donc la suivante :

```
int premier(file * f){

if(f->tete == f->fin)
 printf("La file est vide\n");
else
 return f->tab[f->tete];
}
```

Pour ajouter un élément enfin de la file, il suffit d'incrémenter la valeur de `f.queue` et d'affecter la valeur à ajouter dans la nouvelle position obtenue.

```
void enfiler(file * f, int e){
int n;

 n = f->taille_max;
 if(f->tete == (f->fin + 1) % n)
  printf("La file est pleine\n");
 else{
  f->tab[f->fin] = e;
  f->fin = (f->fin + 1) % n;
 }
}
```

La fonction **defiler()** supprime le premier élément de la file en cas de file non vide.

```
int defiler(file *f){
int elem, n;

 n = f->taille_max;
 if(f->tete == f->fin)
  printf("La file est vide\n");
 else{
  elem = f->tab[f->tete] ;
  f->tete = (f->tete + 1) % n;
  return elem ;
 }
}
```

Chapitre V.

Structures arborescentes

V.1 Structures arborescentes (Arbres)

Définition

Un arbre est un ensemble de nœuds organisés de façon hiérarchique à partir d'un nœud distingué appelé la ***racine***. Un arbre est une structure non-linéaire dans laquelle chaque nœud peut avoir des nœuds successeurs appelés fils avec un seul chemin existant de la racine á tout autre nœud. Donc un arbre est une structure de données composée d'un ensemble de nœuds. Chaque nœud contient l'information spécifique de l'application et des liens vers d'autres nœuds (d'autres sous-arbres). La structure d'arbre est l'une des plus importantes de l'informatique par exemple : l'organisation des fichiers au niveau d'un disque dur. Une propriété intrinsèque de la structure d'arbre est la récursivité car la plupart des algorithmes qui manipulent des arbres s'écrivent d'une manière récursive.

Exemples

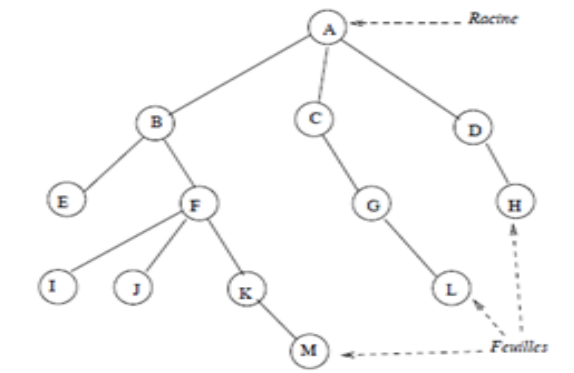


Fig. 1. Un exemple d'arbre

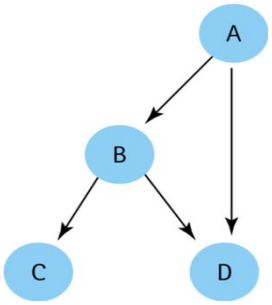


Fig. 2. Ce n'est pas un arbre

V.2 Arbres binaires

Un arbre binaire est un arbre tel que les nœuds ont au plus deux fils (gauche et droit). Par conséquent, un arbre binaire est définie par :

- Une racine ***r.***
- Un sous arbre gauche ***G***.
- Un sous arbre droit ***D***.

Où ***G*** et ***D*** sont eux-mêmes des arbres binaires disjoints, On note un arbre par A = <r, G, D>. Un arbre binaire peut être vide, il est noté ∅.

Exemples

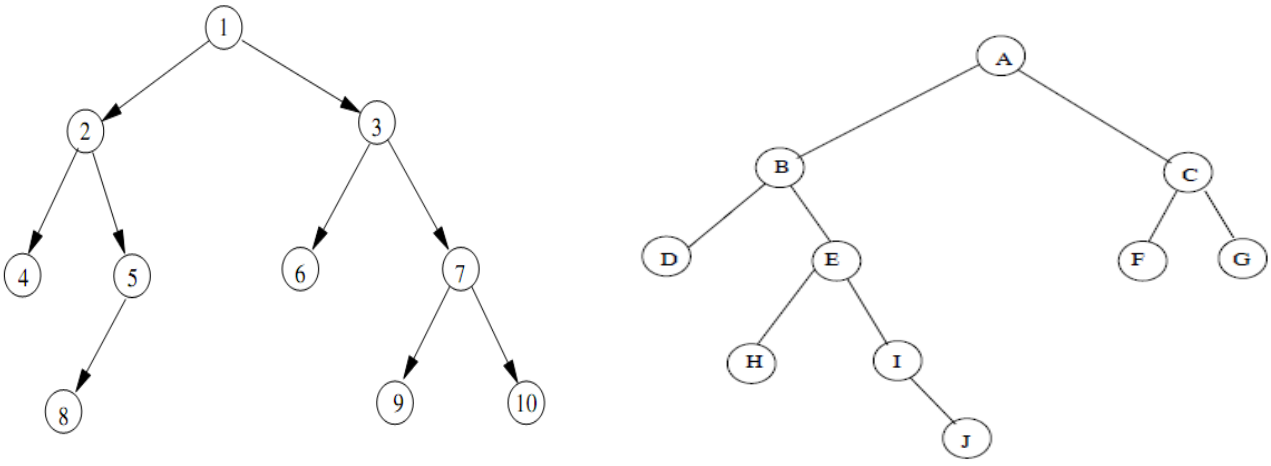


Fig. 3. Exemples d'arbres binaires

La signature du type abstrait arbre binaire est la suivante :

TA Arbre
Utilise Nœud, élément
Opérations
arbre_vide : ⟶ Arbre
est vide : Arbre ⟶ booléen
< , , > : Nœud * Arbre * Arbre ⟶ Arbre
Racine : Arbre ⟶ Nœud
G : Arbre ⟶ Arbre
D : Arbre ⟶ Arbre
Contenue : Nœud ⟶ élément

Soit r de sorte nœud, B de sorte arbre, on a les pré-conditions suivantes :

Racine(B) **est définie ssi** arbre_vide(B) = faux
G(B) **est définie ssi** arbre_vide(B) = faux
D(B) **est définie ssi** arbre_vide(B) = faux

En supposant les pré-conditions vérifiées, ces opérations vérifient les axiomes suivants, pour tous B1, B2 de sorte Arbre et r de sorte nœud :

Racine(<r, B1, B2>) = r
G(<r, B1, B2>) = B1
D(<r, B1, B2>) = B2

Terminologie

- Un arbre dont les nœuds contiennent des éléments est dit arbre ***étiqueté***.
- Soit B =< r, B1, B2 > un arbre binaire. Le symbole r est la racine de B, B1 (resp. B2) est le sous-arbre gauche (resp. droit) de B. On dit que C est un sous-arbre de B si C = B1 ou bien C = B2.
- L'enfant gauche (resp. droit) d'un nœud est la racine de son sous-arbre gauche (resp. droit) et il y a un lien gauche (resp. droit) entre le nœud et son enfant gauche (resp. droit).
- Si un nœud n_i a pour enfant gauche (resp. droit) n_j , on dit que n_i est le père de n_j; deux nœuds ayant même parents sont frères (siblings).
- Un nœud qui à deux fils est dit nœud interne ou point double. Un nœud qui a seulement un fils gauche (resp. droit) est dit point simple à gauche (resp. droit).
- Un nœud qui n'a pas de fils est dit nœud externe ou feuille.
- On appelle branche de l'arbre B, tout chemin (une suite de nœuds consécutifs) de la racine à une feuille de B, un arbre a donc autant de branches que de feuilles.

Opérations sur les arbres

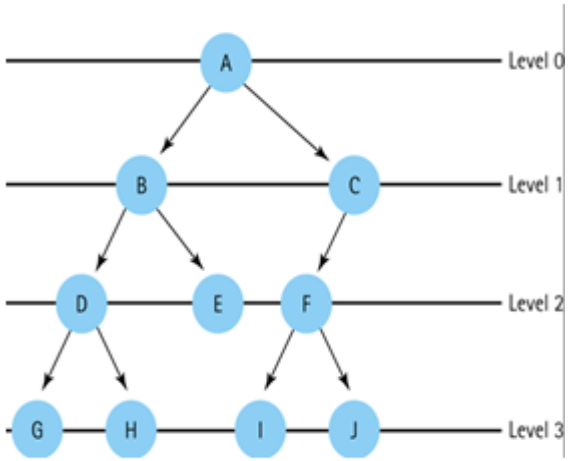
On introduit quelques opérations sur les arbres qui seront utile pour les évaluer :

- La taille d'un arbre est le nombre de ces nœuds, on définit récursivement l'opération taille par :

```
1. taille(∅) = 0.
2. taille(<r, B1, B2>) = 1 + taille(B1) + taille(B2).
```

- La hauteur d'un nœud (profondeur ou niveau) est définie récursivement de la façon suivante, étant donnée x un nœud de B.
  1. `h(r) = 0` (r racine de l'arbre).
  2. `h(x) = 1 + h(y)` ssi y est le père de x.
- La hauteur d'un arbre B est `h(B) = max {h(x), x nœud de B}`
- La longueur de cheminement d'un arbre B est `LC(B)` $= \sum h(x)$, la somme étant prise sur tous les nœuds de l'arbre B.
- La longueur de cheminement externe d'un arbre B est `LCE(B)` $= \sum h(f)$, la somme étant prise sur tous les feuilles de l'arbre B.
- La longueur de cheminement interne d'un arbre B est `LCI(B)` $= \sum h(x)$, la somme étant prise sur tout nœud x non externe (ayant au moins un fils) de l'arbre B.
- Logiquement on a la relation suivante : `LC(B) = LCE(B) + LCI(B).`
- Le nombre maximal de nœuds du N^{éme} niveau d'un arbre binaire est 2^N.
- Le nombre maximal de nœuds d'un arbre binaire à N niveaux est $2^{N+1} - 1$.

Soit l'arbre binaire B suivant :



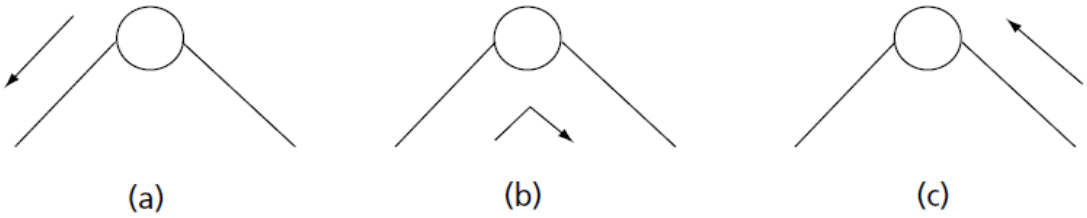
Les Nœuds internes sont : A, B, D, F.
Les points simples à gauche : C.
Les feuilles sont : G, H, I, J, E.
Les hauteurs des nœuds sont :
`h(A) = 0.`
`h(B) = h(C) = 1.`
`h(D) = h(E) = h(F) = 2.`
`h(G) = h(H) = h(I) = h(J) = 3.`
Les caractéristiques de l'arbre sont :
`h(B) = 3.`
`LC(B) = 20.`
`LCE(B) = 14.`
`LCI(B) = 6.`

Parcours d'un arbre

L'opération de parcours sur les arbres binaires consiste à examiner systématiquement, dans un certain ordre, les nœuds de l'arbre pour y effectuer un traitement donné. Un algorithme de parcours d'arbre est un procédé permettant d'accéder à chaque nœud de l'arbre. Un certain traitement est effectué pour chaque nœud (test, écriture, comptage, etc.), mais le parcours est indépendant de cette action et commun à des algorithmes qui peuvent effectuer des traitements très divers. Le ***parcours en***

profondeur consiste à partir de la racine et à choisir toujours l'enfant le plus à gauche d'abord. On constate que chaque nœud est rencontré une fois en descendant, puis une deuxième fois en remontant depuis l'enfant gauche, et une troisième fois en remontant depuis l'enfant droit. Donc dans un parcours d'arbre, un nœud est visité trois fois :

- lors de la première rencontre du nœud, avant de parcourir le sous-arbre gauche.
- après parcours du sous-arbre gauche, avant de parcourir le sous-arbre droit.
- après examens des sous-arbres gauche et droit.



a. Parcours préfixé

Le premier type de parcours est appelé parcours préfixé. Il faut traiter le nœud lors de la **première** visite, puis explorer le sous-arbre gauche (en appliquant la même méthode) avant d'explorer le sous-arbre droit. La procédure se schématise comme suit :

- ✓ traitement de la racine;
- ✓ traitement du sous-arbre gauche;
- ✓ traitement du sous-arbre droit.

b. Parcours infixé

Dans un parcours infixé, le nœud est traité lors de la **deuxième** visite, après avoir traité le sous-arbre gauche, mais avant de traiter le sous-arbre droit. La procédure se schématise comme suit :

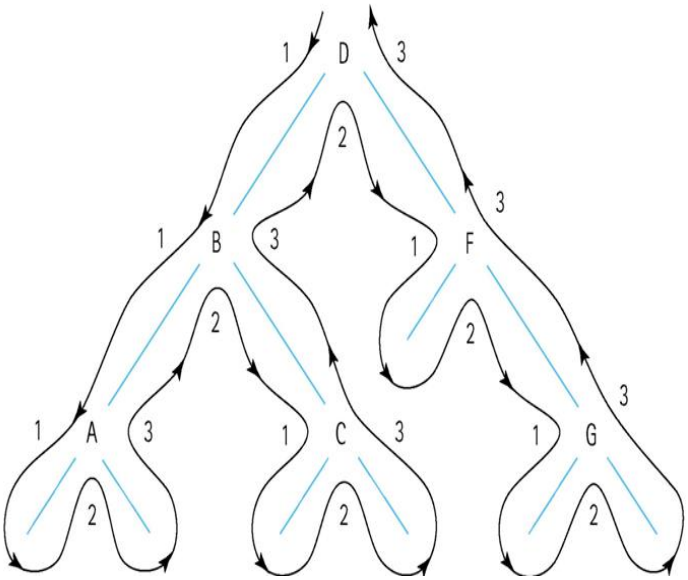
- ✓ traitement du sous-arbre gauche;
- ✓ traitement de la racine;
- ✓ traitement du sous-arbre droit.

c. Parcours postfixé

En parcours postfixé, le nœud est traité lors de la **troisième** visite, après avoir traité le SAG et le SAD. La procédure à suivre est donnée ci-dessous :

- ✓ traitement du sous-arbre gauche;
- ✓ traitement du sous-arbre droit;
- ✓ traitement de la racine.

Exemple 1

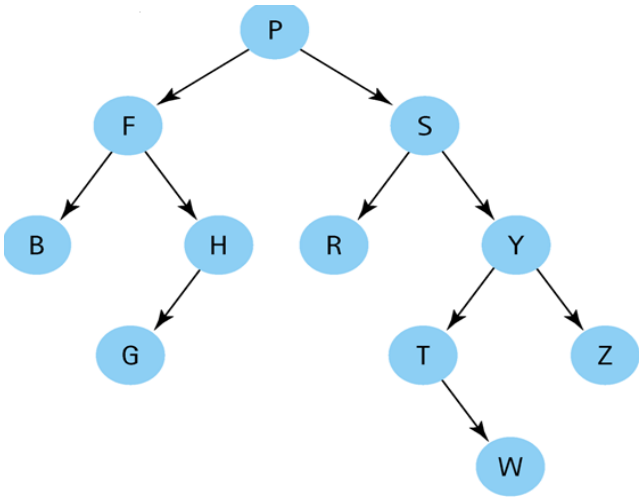


Préfixé : D, B, A, C, F, G.

Infixé : A, B, C, D, F, G.

Postfixé : A, C, B, G, F, D.

Exemple 2



Préfixé : P, F, B, H, G, S, R, Y, T, W, Z.

Infixé : B, F, G, H, P, R, S, T, W, Y, Z.

Postfixé : B, G, H, F, R, W, T, Z, Y, S, P.

II.5.3 Représentation des arbres binaires

Représentation chaînée des arbres binaires

En ***représentation chainée*** on associe à chaque nœud deux pointeurs l'un vers le sous arbre gauche et l'autre vers le sous arbre droit, par conséquent un arbre est représenté par l'adresse de sa racine lorsque l'arbre est étiqueté on représente dans un champ supplémentaire l'information contenu dans le nœud. Donc un arbre binaire peut être représenté par une structure contenant un champ donnée et deux pointeurs vers les nœuds fils:

```
Struct noeud{
 int valeur;
 noeud * gauche;
 noeud * droite;
```

```
}
```

Un arbre binaire vide est représenté par le pointeur NULL, si l'arbre n'est pas vide il est représenté par le pointeur qui pointe sur la racine de l'arbre.

Représentation contiguë des arbres binaires

En ***représentation contiguë*** on associe un indice à chaque nœud de l'arbre dans un tableau à deux champ G et D. Le champ G (resp. D) contient l'indice associé à la racine du sous arbre gauche (resp. droit).

- Si le nœud n'a pas de fils gauche (resp. droit), on mentionne dans le champ G (resp. D) la valeur -1.
- Si de plus l'arbre est étiqueté, on ajoute un champ supplémentaire qui va contenir l'information associée à ce nœud.
- On doit mémoriser particulièrement l'indice de la racine de l'arbre.

Chaque nœud est représenté par la structure suivante :

```
struct noeud{
 int valeur;
 int G;
 int D;
}
```

Donc un arbre binaire peut être représenté par la structure suivante :

```
struct arbre{
 int racine;
 noeud * tab;
 int taille_max;
}
```

Si `A` est de type `arbre` et `r = A.racine` alors `A.tab[r]` correspond à racine(A), `A.tab[r].val` est le contenu de cette racine et `A.tab[r].G` et `A.tab[r].D` sont les indices des racines de G(A) et D(A).

V.3 Propriété de l'arbre binaire

a. Taille d'un arbre binaire

La taille d'un arbre est le nombre de nœuds de cet arbre. La taille à partir du nœud pointé par racine est de 0 si l'arbre est vide, et vaut 1 (le nœud pointé par racine) plus le nombre de nœuds du sous-arbre gauche et plus le nombre de nœuds du sous arbre droit sinon.

```
int taille (noeud * racine) {

 if (racine == NULL)
  return 0;
 else
  return 1 + taille (racine->gauche) + taille (racine->droite);
}
```

b. Feuilles d'un arbre binaire

La fonction ***estFeuille()*** est une fonction booléenne qui indique si un nœud donnée est une feuille (n'a pas de successeur).

```
// Le noeud x est-il une feuille ?

int estFeuille (noeud * x){

 if (x->gauche==NULL) && (x->droite==NULL)
  return 1 ;
 else
  return 0;
}
```

c. Nombre de feuilles dans un arbre binaire

La fonction ***nbFeuilles()*** compte le nombre de feuilles de l'arbre binaire à partir du nœud racine. Si l'arbre est vide, le nombre de feuilles est 0 ; sinon si racine repère une feuille, le nombre de feuilles est de 1, sinon, le nombre de feuilles en partant du nœud racine est le nombre de feuilles du SAG, plus le nombre de feuilles du SAD.

```
int nbFeuilles (noeud * racine){

 if (racine == NULL)
  return 0;
 else
  if (estFeuille (racine))
   return 1;
  else
   return nbFeuilles (racine->gauche)+ nbFeuilles (racine->droite);
}
```

d. Parcours préfixe

Le nœud racine est traité (affiché) avant les deux appels récursifs.

```
void prefixe (noeud * racine){
  if (racine != NULL){
   printf ("%d ", racine->valeur);
   prefixe (racine->gauche);
   prefixe (racine->droite);
 }
}
```

e. Parcours infixe

Le nœud racine est traité (affiché) entre les deux appels récursifs.

```
void infixe (noeud* racine){

 if (racine != NULL){
  infixe (racine->gauche);
  printf ("%d", racine->valeur);
  infixe (racine->droite);
 }
}
```

f. Parcours postfixe

Le nœud racine est traité (affiché) après les deux appels récursifs.

```
void postfixe (noeud * racine){

 if (racine != NULL){
  postfixe (racine->gauche);
  postfixe (racine->droite);
  printf ("%d", racine->valeur);
 }
}
```

V.4 Arbres binaires de recherche (ABR)

Un arbre binaire de recherche est un arbre binaire dans lequel chaque nœud possède une étiquette (clé), telle que chaque nœud du sous arbre *gauche* ait une clé inférieure ou égale à celle du nœud considéré, et que chaque nœud du sous arbre *droit* possède une clé supérieure ou égale à celle-ci. Selon la mise en œuvre de l'arbre, on pourra interdire ou non des clés de valeur égale.

Exemple

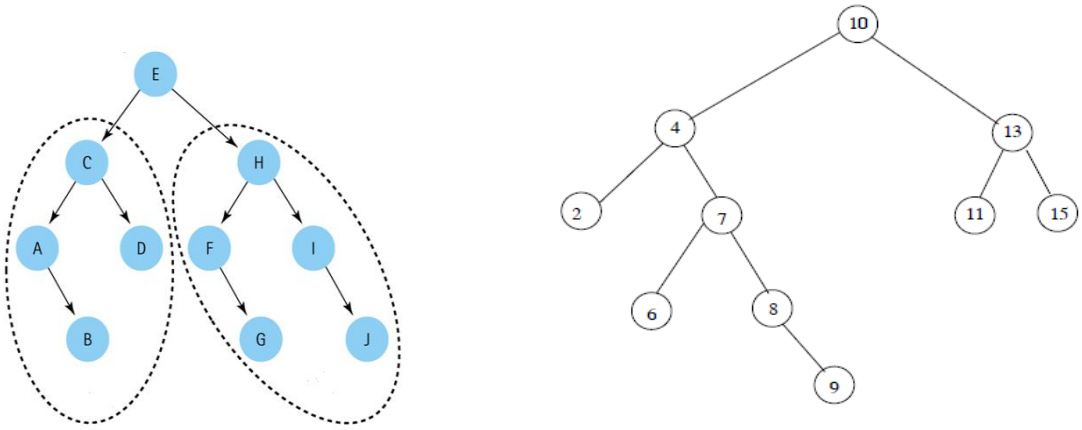


Fig. 4. Exemples d'arbres binaires de recherche

Par conséquent, un arbre binaire de recherche est un arbre binaire ayant les propriétés suivantes :

1. Tous les nœuds du sous arbre gauche de la racine ont des valeurs strictement inférieures à la valeur de la racine.
2. Tous les nœuds du sous arbre droit de la racine ont des valeurs supérieures ou égales à la valeur de la racine.
3. Les sous-arbres gauches et droits sont eux mêmes des arbres binaires de recherche.

- Le seul parcours de l'arbre qui donne lieu à une liste ordonnée est le **parcours infixé** (SAG, racine, SAD).
- La reconstruction d'un arbre binaire de recherche à partir de sa liste obtenu après **parcours préfixé** redonne l'arbre initial.

a. Recherche d'un nœud dans un arbre binaire de recherche

La recherche d'un nœud dans un arbre vide retourne la valeur NULL (pas trouvé). Si le nœud pointé par racine contient la valeur que l'on cherche alors le résultat est le pointeur racine ; sinon, on cherche le nœud dans le SAG ou le SAD selon que la valeur recherché soit inférieur au supérieur de la racine.

```
noeud* trouverNoeud(noeud* racine, int x) {
if (racine == NULL)
 return NULL;
else
 if (racine->valeur == x)
  return racine ;
 else
  if (x < racine->valeur)
   return trouverNoeud(racine->gauche, x);
  else
   return trouverNoeud(racine->droite, x);
 }
```

b. Ajouter un nœud dans un arbre binaire de recherche

Pour conserver les propriétés d'un arbre binaire de recherche, l'ajout d'un nouvel élément ne peut pas se faire n'importe comment. La fonction récursive d'ajout d'un élément peut s'exprimer ainsi:

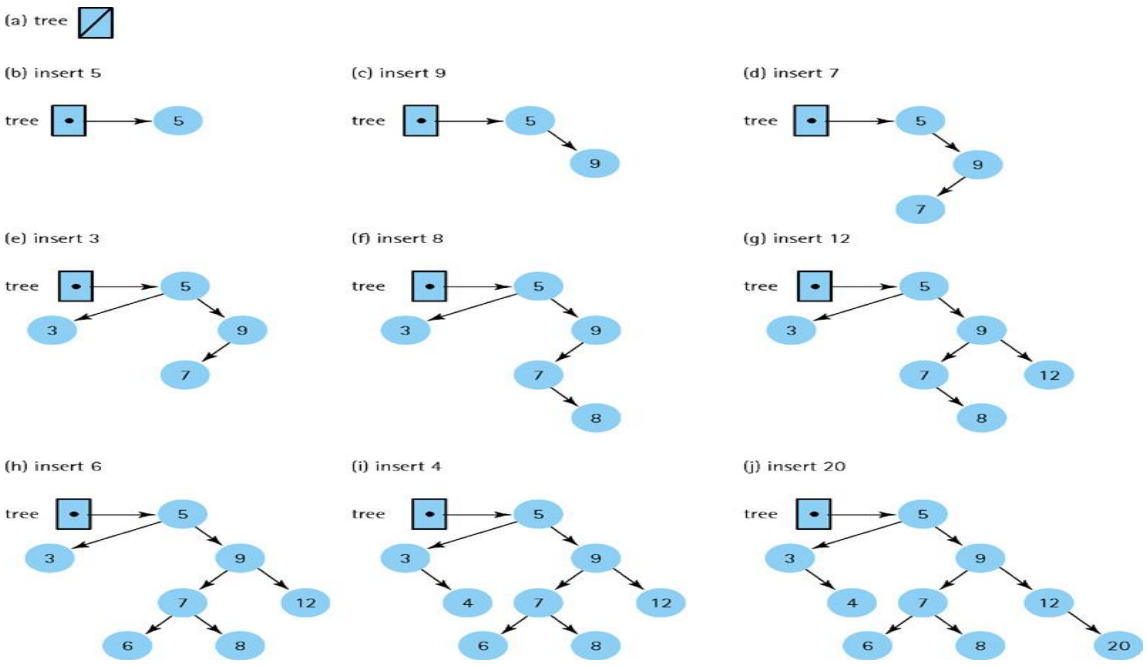
Soit x la valeur de l'élément à insérer et v la valeur du nœud racine n_i d'un sous-arbre :

- si n_i n'existe pas, le créer avec la valeur x.
- sinon, si x est plus petit que v, remplacer n_i par son fils gauche. Recommencer à partir de la nouvelle racine.
- Sinon, remplacer n_i par son fils droit. Recommencer à partir de la nouvelle racine.

```
noeud * ajouter (noeud * racine, int x){

 if (racine == NULL){
  racine = (noeud *)malloc(sizeof(noeud));
  racine->valeur = x;
  racine->gauche = NULL;
  racine->droite = NULL;
 }
 else{
  if (x < racine->valeur)
   racine->gauche = ajouter(racine->gauche, x);
  else
   racine->droite = ajouter(racine->droite, x);
 }
 return racine;
}
```

Exemple

Créer l'ABR correspondant à la liste de nœud suivante : 5 9 7 3 8 12 6 4 20.
Donnez la suite des nœuds obtenus par le parcours infixé de l'ABR obtenu.



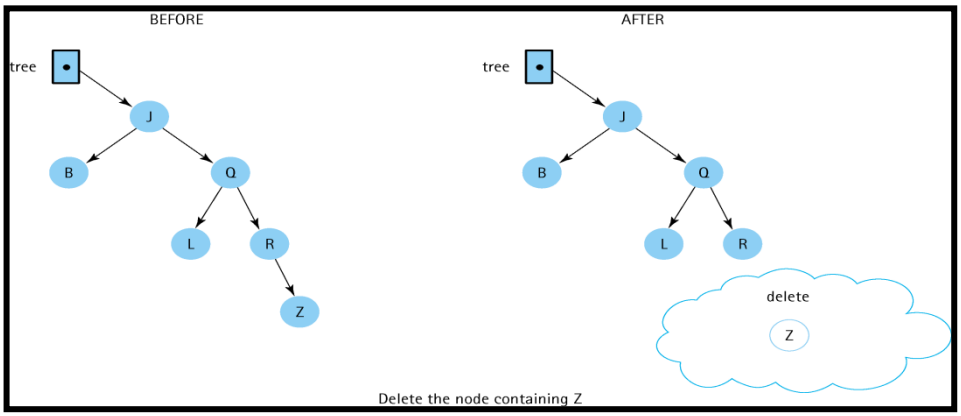
c. Supprimer un nœud d'un arbre binaire de recherche

La suppression est la plus difficile des opérations sur un ABR car la valeur à supprimer peut appartenir à un nœud interne de l'arbre. Nous devons nous assurer, quand on supprime un nœud, que la propriété de l'arbre binaire de recherche est maintenue, il y a 3 cas à considérer :

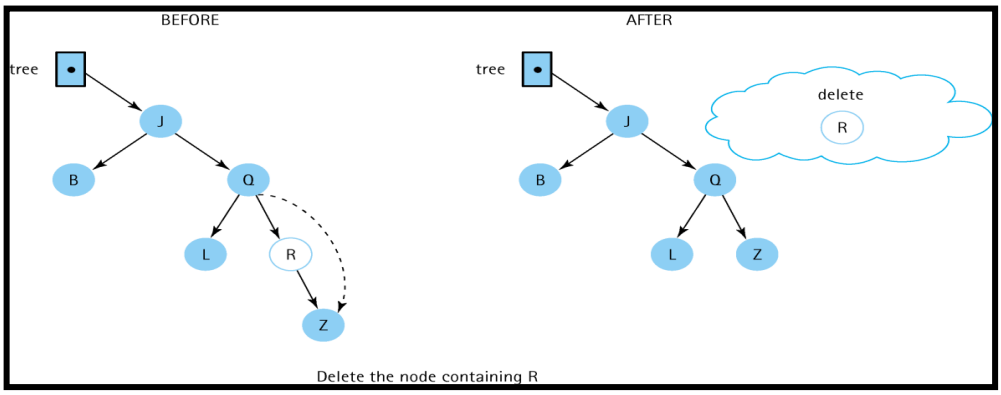
1. Le nœud à supprimer est une feuille. Dans ce cas, il suffit de retrouver la feuille et de la supprimer.
2. Le nœud à supprimer n'a qu'un sous-arbre (gauche ou droit). Dans ce cas, il suffit d'attacher ce sous-arbre au parent du nœud supprimé.
3. Le nœud à supprimer a deux sous-arbres (gauche et droit). Dans ce cas, remplacer le nœud à supprimer par le nœud du sous-arbre gauche contenant la valeur maximale, ou par le nœud du sous-arbre droit contenant la valeur minimale.

Exemples

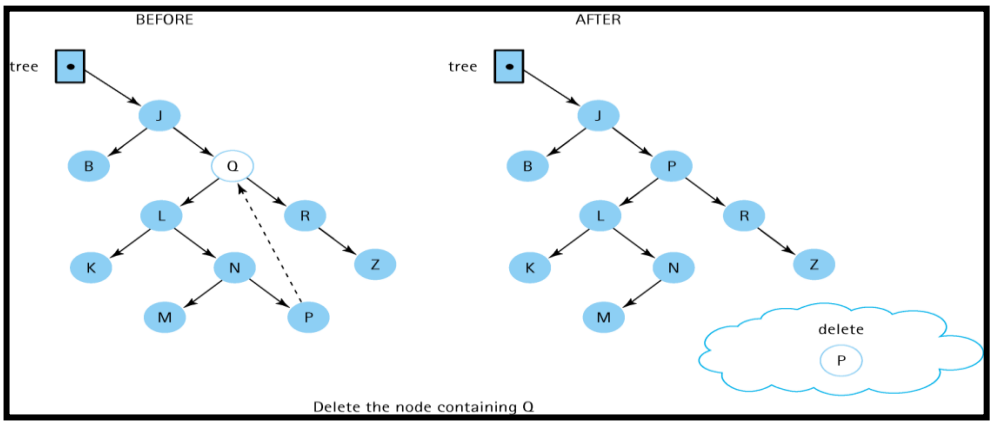
Suppression d'une feuille



Suppression d'un nœud ayant un fils

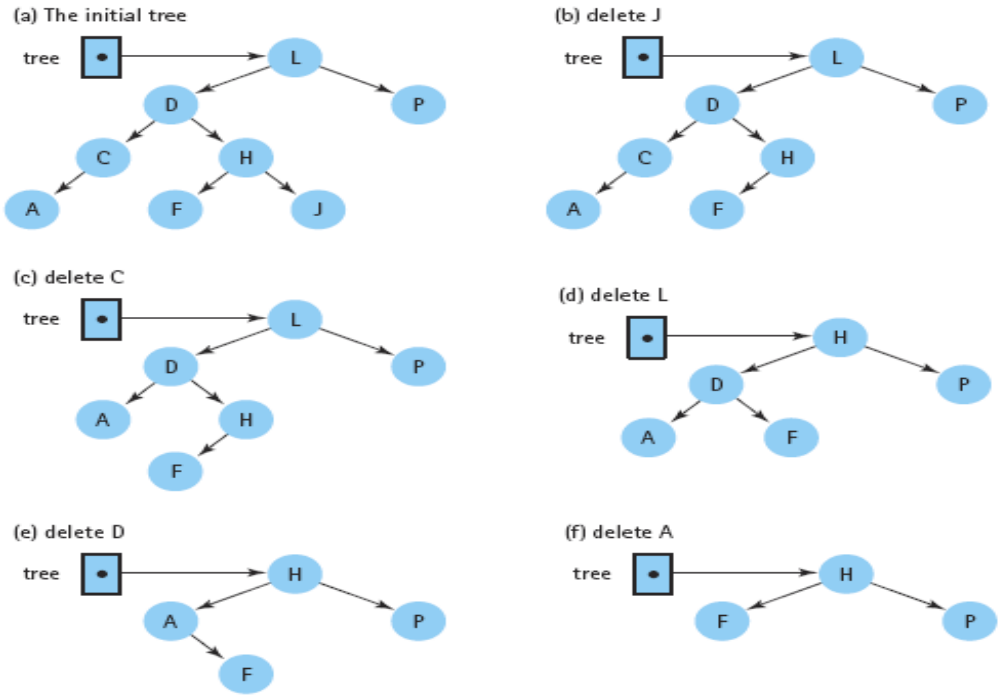


Suppression d'un nœud ayant deux fils



Exemple

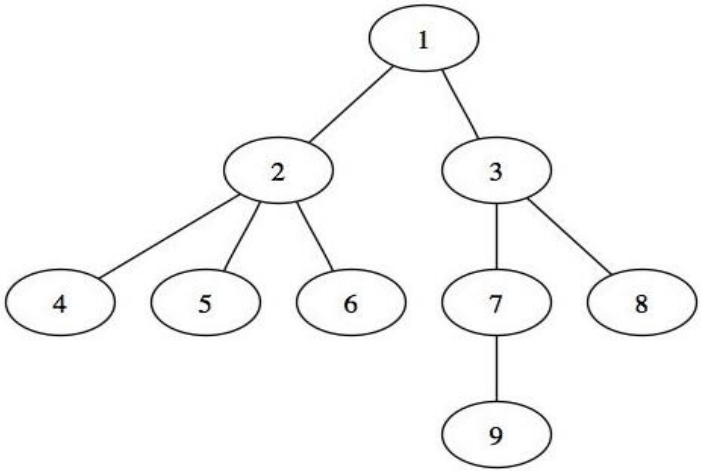
Supprimer les nœuds J, C, L, D, A de l'ABR initial.



V.5 Arbres planaires généraux (arbres n-aires)

Un arbre planaire général est une structure arborescente plus large où les nœuds peuvent avoir un nombre quelconque de fils. Par conséquent, un arbre planaire général est définit par une racine ***r*** et d'une liste finie éventuellement vide d'arbres appelée forêt. On note un arbre par planaire général par $A = <r, A_1, \ldots, A_n>$.

Exemples



La signature du type abstrait arbre planaire général est la suivante :

TA Arbregen
Utilise Forêt, Nœud, Entier
Opérations
créer : Forêt * Nœud ⟶ Arbregen
racine : Arbregen ⟶ Nœud
sous-arbres : Arbregen ⟶ Forêt
iéme : Forêt*entier ⟶ Arbregen
nb_arbres : Forêt ⟶ Entier
insérer : Forêt*Arbregen*Entier ⟶ Forêt

Soit n de sorte nœud, B de sorte arbregen, F de sorte Forêt et i de sorte entier, on a les pré-conditions suivantes :

insérer(F, B, i) **est définie ssi** $1 \leq i \leq$ nb_arbres(F) + 1
iéme(F, i) **est définie ssi** $1 \leq i \leq$ nb_arbres(F)

En supposant les pré-conditions vérifiées, ces opérations vérifient les axiomes suivants :

longueur(insérer(F, B, i)) = longueur(F) +1
sous-arbre(créer(F, n)) = F

V.6 Représentation des arbres planaires généraux

Représentation chaînée des arbres planaires

Les éléments sont alloués au cours de la construction de l'arbre et reliés entre eux. Le nombre de pointeurs présents dans chaque nœud dépend du degré de l'arbre. Si le degré de chaque nœud est constant, cette mémorisation est parfaite. Sinon le nombre maximum N d'enfants pour une personne

est difficile à définir. De plus, il conduit à une perte importante de place puisqu'il faut prévoir N pointeurs pour chaque nœud, y compris les feuilles. Pour un arbre où le nombre de fils maximum est de trois, nous pouvons utiliser le modèle de structure suivant :

```
struct noeud{
 int valeur;
 noeud * P1;
 noeud * P2;
 noeud * P3;
}
```

Représentation contiguë des arbres planaires

L'arbre peut être mémorisé dans un espace contiguë. On associe un indice à chaque nœud de l'arbre dans un tableau de plusieurs champs, ces champs contiennent les indices associés aux racines des sous arbres du nœud en question.

- Si le nœud n'a pas de fils, on mentionne dans le champ correspondant la valeur -1.
- Si de plus l'arbre est étiqueté, on ajoute un champ supplémentaire qui va contenir l'information associée à ce nœud.
- On doit mémoriser particulièrement l'indice de la racine de l'arbre.

Chaque nœud est représenté par la structure suivante :

```
struct noeud{
 int valeur;
 int P1;
 int P2;
 int P3;
}
```

Donc un arbre planaire peut être représenté par la structure suivante :

```
struct arbre{
 int racine;
 noeud * tab;
 int taille_max;
}
```

Mémorisation par listes de fils

L'allocation est dans ce cas en partie contiguë et en partie dynamique. Avec cette structure de données, les insertions et les suppressions de nœuds ne sont pas faciles à gérer pour la partie contiguë. De plus, il est difficile de donner un maximum pour la déclaration de cette partie si on peut ajouter des éléments.

Chaque nœud est représenté par la structure suivante :

```
struct noeud{
 int valeur;
 succ * adr;
}

struct succ{
 int indice;
```

```
       succ * suivant;
      }
```

Donc un arbre planaire peut être représenté par la structure suivante :

```
      struct arbre{
       int racine;
       noeud * tab;
       int taille_max;
      }
```

V.7 Parcours en largeur d'un arbre planaire général

```
void largeur(noeud * racine){
 file f;
 noeud * n;
 f.tete = NULL;
 if (racine != NULL){
  f = enfiler(f, racine);
  while(!estvide(f)){
   n = defiler(&f);
   printf ("%d ", n->valeur);
   if(n->p1 != NULL)
    f = enfiler(f, n->p1);
   if(n->p2 != NULL)
    f = enfiler(f, n->p2);
   if(n->p3 != NULL)
    f = enfiler(f, n->p3);
  }
 }
}
```

Chapitre VI.

Graphes

VI.1 Les graphes

Beaucoup de problème de la vie courante tel que la gestion des réseaux de communication ou l'ordonnancement des taches d'un projet peuvent être modélisé par des graphes. Un graphe est une structure de données composée d'un **ensemble de sommets**, et d'un ensemble de **relations entre ces sommets**.

Si la relation n'est pas orientée, la relation est supposée exister dans les deux sens. Le graphe est dit **non orienté** ou **symétrique**. Dans le cas contraire, si les relations sont orientées, le graphe est dit **orienté**. Une relation est appelée un **arc** (une arête pour les graphes non orientés). Les sommets sont aussi appelés nœuds ou points.

VI.2 Graphes non orientés (ou symétriques)

Le graphe de la Figure 6.1 peut se noter comme suit :
S = {S1, S2, S3, S4, S5}; ensemble des sommets
A = {S1S2, S1S3, S2S3, S3S4, S3S5, S4S5}; ensemble des relations symétriques. Par exemple, S1S2 est vrai, de même que S2S1.

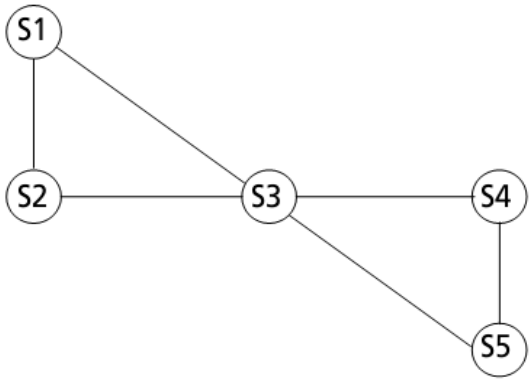


Figure 6.1. Un graphe non orienté : les relations existent dans les deux sens.

Exemple

Dans une carte de liaison aérienne, les villes sont des sommets du graphe et l'existence d'une liaison aérienne entre les deux villes est la relation, cette relation est symétrique. Dans ce cas le graphe est **non orienté**.

Donc un graphe **non orienté** G est un couple <S, A>, où S est un ensemble finie de sommets et où A est un ensemble fini d'**arcs**.

Graphe connexe : un graphe non orienté est dit connexe si on peut aller de tout sommet vers tous les autres sommets. Le graphe de la **Figure 6.1** est connexe; celui de la **Figure 6.2** ne l'est pas ; il est constitué de deux composantes connexes.

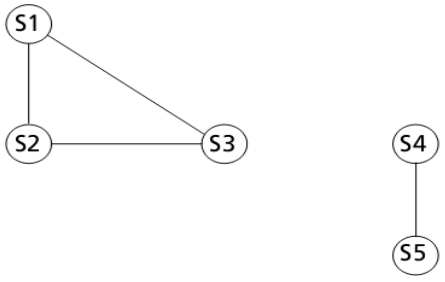


Figure 6.2. Un graphe non connexe et ses deux composantes connexes.

VI.3 Graphes orientés

Si les relations sont orientées, le graphe est dit orienté.

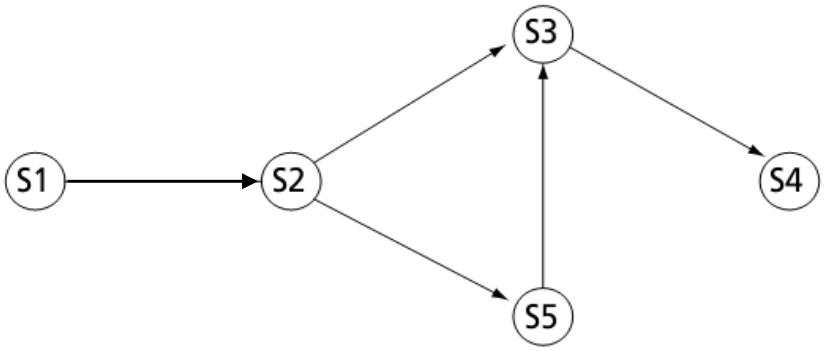


Figure 6.3. Un graphe orienté.

Pour un arc S1S2, S2 est dit successeur de S1 ou encore adjacent à S1 ; S1 est le prédécesseur de S2

Exemple

Dans un projet ou certaines tâches doivent être exécuté avant d'autres, on peut représenter l'ordonnancement des tâches par un graphe où les sommets sont les tâches et les arcs sont les liens entre les tâches. Dans ce cas on dit que le graphe est **orienté**.

Donc un graphe **orienté** G est un couple <S, A>, où S est un ensemble finie de sommets et où A est un ensemble fini d'arêtes.

Le degré d'un graphe :

$d^{\circ}(S)$: nombre d'arcs entrants et sortants de S.
$d^{+}(S)$: nombre d'arcs sortants de S : demi-degré extérieur ou degré d'émission
$d^{-}(S)$: nombre d'arcs entrants de S: demi-degré intérieur ou degré de réception

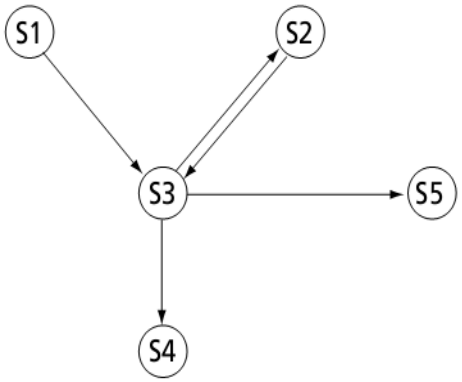


Figure 6.4. Degré d'un graphe orienté.

$d^{\circ}(S3) = 5$
$d^{+}(S3) = 3$
$d^{-}(S3) = 2$

Graphe fortement connexe : un graphe orienté est dit fortement connexe si on peut aller de tout sommet vers tous les autres sommets (en passant éventuellement par un ou plusieurs sommets intermédiaires).

Figure 6.5. Graphe non fortement connexe et composantes fortement connexes.

II.6.4 Graphes orientés ou non orientés

Les définitions suivantes s'appliquent aux graphes orientés comme aux graphes non orientés.

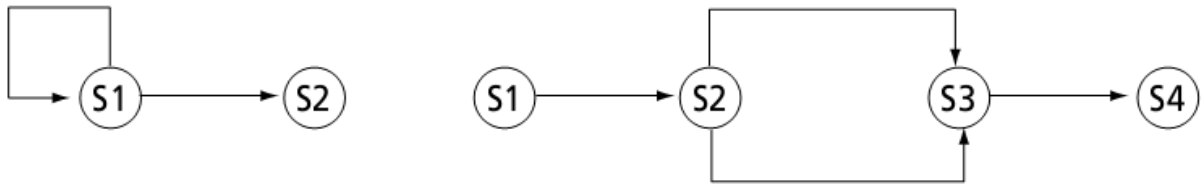


Figure 6.6. Une boucle et un graphe multiple.

Une boucle (auto boucle) est une relation (Si, Si). Un multi graphe ou graphe multiple est un graphe tel qu'il existe plusieurs arcs entre certains sommets. Sur la Figure 6.6, la relation S2S3 existe 2 fois; le graphe est un 2-graphe orienté ou plus généralement un p-graphe. Un graphe simple est un graphe sans boucle et sans arc multiple.

Un graphe est dit valuès (pondéré) si à chaque arc on associe une valeur représentant le coût de la transition de cet arc.

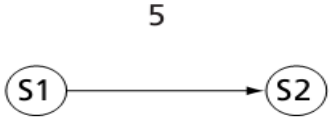


Figure 6.7. Un graphe valué : la transition S1 vers S2 coûte 5.

Un **chemin** dans un graphe est une suite d'arcs consécutifs. La **longueur d'un chemin** est le nombre d'arcs constituant ce chemin. Un **chemin simple** est un chemin où aucun arc n'est utilisé 2 fois.
Un **circuit simple** est un chemin simple tel que le premier et le dernier sommet sont les mêmes.

Remarque : Un graphe non orienté peut toujours être considéré comme un graphe orienté où les relations symétriques sont explicitement mentionnées.

VI.4 Types abstraits graphes

On va avoir deux types abstraits un pour les graphes orientés et un autre pour les graphes non orientés.

Spécification des graphes orientés

Pour le développement de la signature **graphe**, on considère les conventions suivantes :

- Quand on ajoute un sommet, celui-ci est isolé.
- Quand on ajoute un arc, si les sommets adjacents à cet arc n'appartiennent pas au graphe, on les ajoute. Par contre, quand on retire un arc, les sommets ne sont pas retirés.
- Les opérations d'ajout d'un sommet ou d'un arc ne sont pas définies si le sommet ou l'arc est déjà présent dans le graphe concerné.

La signature du type abstrait Graphe est la suivante :

TA Graphe
Utilise Sommet, Booléen, Entier
Opérations
Graphe_vide : ⟶ Graphe
ajouter_sommet: Sommet* Graphe ⟶ Graphe
ajouter_arc : Sommet*Sommet *Graphe ⟶ Graphe
supprimer_sommet : Sommet *Graphe ⟶ Graphe
supprimer_arc : Sommet*Sommet *Graphe ⟶ Graphe
appartient_sommet: Sommet *Graphe ⟶ Booléen
appartient_arc: Sommet*Sommet *Graphe ⟶ Booléen
d^+ : Sommet*Graphe ⟶ Entier
d^- : Sommet*Graphe ⟶ Entier
i ^{éme} succ_sommet : Entier*Sommet *Graphe ⟶ Sommet
i ^{éme} pred_sommet : Entier*Sommet *Graphe ⟶ Sommet

Les opérations ci-dessus ne sont pas définies partout, on a les pré-conditions suivantes où s et s' sont de sorte sommet, G de sorte graphe et i de sorte entier :

ajouter_sommet(s, G) **est définie ssi** appartient_sommet(s, G) = Faux.
ajouter_arc (s, s', G) **est définie ssi** appartient_arc(s, s', G) = Faux.
d^+ (s, G) **est définie ssi** appartient_sommet(s, G) = Vrai.
i ^{éme} succ_sommet(i, s, G) **est définie ssi** ($i \leq d^+$ (s, G)).

En supposant les pré-conditions vérifiées, nous avons les axiomes suivants :

Si $s \neq s'$ **alors** appartient_sommet(s, ajouter(s', G)) = appartient_sommet(s, G).
Si $s \neq s'$ et $s' \neq s''$ **alors** appartient_sommet(s, supprimer_arc(s', s", G)) = ppartient_sommet(s, G).
Si appartient_sommet(s, G) = Vrai **alors** d^+(s, ajouter_arc (s, s', G)) = d^+(s, G) + 1.
Si appartient_sommet(s, G) = Vrai **alors** d^+(s, retirer_arc (s, s', G)) = d^+(s, G) - 1.

Spécification des graphes non orientés

Pour les graphes non orientés, nous avons la signature suivante :

TA Graphe
Utilise Sommet, Booléen, Entier
Opérations
Graphe_vide : ⟶ Graphe
ajouter_sommet: Sommet* Graphe ⟶ Graphe

ajouter_arrête : Sommet*Sommet *Graphe ⟶ Graphe
supprimer_sommet : Sommet *Graphe ⟶ Graphe
supprimer_arrête : Sommet*Sommet *Graphe ⟶ Graphe
appartient_sommet: Sommet *Graphe ⟶ Booléen
appartient_arrête: Sommet*Sommet *Graphe ⟶ Booléen
d° : Sommet*Graphe ⟶ Entier
i éme succ_sommet : Entier*Sommet *Graphe ⟶ Sommet

Les axiomes sont similaires à ceux d'un graphe orienté, il suffit de remplacer arc par arête et d^+ par d°. Si une arête <s, s'> est ajoutée dans un graphe G, appartient_arête(s, s', G) = vrai et appartient_arête(s', s, G) = vrai, de même d°(s) et d°(s') sont augmenté de 1.

VI.5 Représentation des graphes

Suivant le rapport entre le nombre de sommets et le nombre d'arcs, on choisit soit une mémorisation sous forme de matrice, soit une mémorisation sous forme de listes d'adjacence.

Mémorisation sous forme de matrices d'adjacence

On souhaite définir une structure de données pour représenter un graphe. Considérons S = {s_0, s_1, …, s_{n-1}} l'ensemble des sommets. Nous allons représenter le graphe par une matrice A de taille (n × n). L'élément A[i][j] vaudra 1 s'il y a un arc (i, j) du sommet s_i vers le sommet s_j, et A[i][j]vaudra 0 sinon.

Dans la **figure 6.7** Le tableau **nomS** contient les noms des sommets et leurs caractéristiques. On peut facilement ajouter des sommets (si **ns** : nombre de sommets < **nMax** : nombre maximum de sommets) et des arcs entre deux sommets à partir de leur nom.

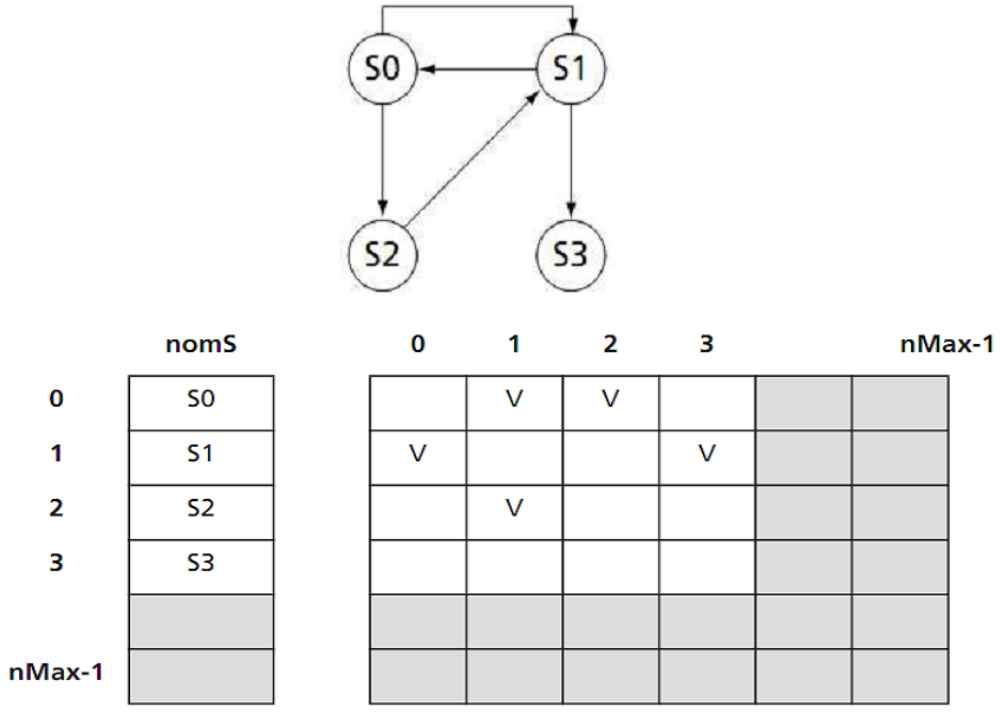


Figure 6.7. Mémorisation sous forme d'une matrice d'adjacence.

Pour cette représentation, un graphe peut être représenté par le modèle de structure suivant :

```
struct sommet{
 /* mettre ici les données relatives aux sommets */
};

struct graphe{
 int nMax;
 sommet * tab;
 int ** matrice;
};
```

Dans le cas où le graphe est orienté, la matrice est symétrique. Dans le cas où le graphe est valué, on utilise une matrice où l'élément i et j a pour valeur le poids de l'arc entre les sommets d'indice i et j (resp. de l'arête entre les sommets d'indice i et j) si l'arc n'existe pas (resp. l'arête) on associe à cet arc (resp. arête) une valeur qui ne peut pas être un poids (une valeur n négative par exemple)

Mémorisation table de liste d'adjacence

Dans cette représentation, La partie concernant les caractéristiques des sommets est mémorisée dans un tableau contenant, pour chaque entrée, une liste des sommets successeurs que l'on peut atteindre directement en partant du sommet correspondant à cette entrée, cette liste est appelé liste d'adjacence. Par exemple, du sommet 0 (S0), on peut aller au sommet numéro 1 (S1) et au sommet numéro 2 (S2).

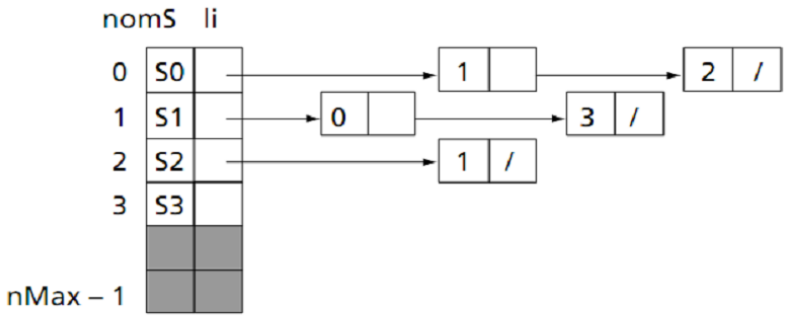


Figure 6.8. Mémorisation sous forme d'une table de listes d'adjacence

Chaque sommet est représenté par la structure suivante :

```
struct sommet{
 /* mettre ici les données relatives aux sommets */
 succ * adr;
};

struct succ{
 int indice;
 succ * suivant;
};
```

Donc un graphe peut être représenté par la structure suivante :

```
struct graphe{
 sommet * tab;
 int nMax;
};
```

Liste des sommets et listes d'adjacence : allocation dynamique

On peut tout allouer dynamiquement : la liste des sommets, et pour chaque sommet, la liste des sommets successeurs. On n'a plus besoin de définir nMax; l'allocation est entièrement dynamique. Le premier champ des listes de successeurs contient soit le numéro ou le nom du sommet successeur, soit un pointeur sur le sommet. Si le graphe est valué, il faut ajouter le poids de l'arc dans chacun des éléments des listes de successeurs.

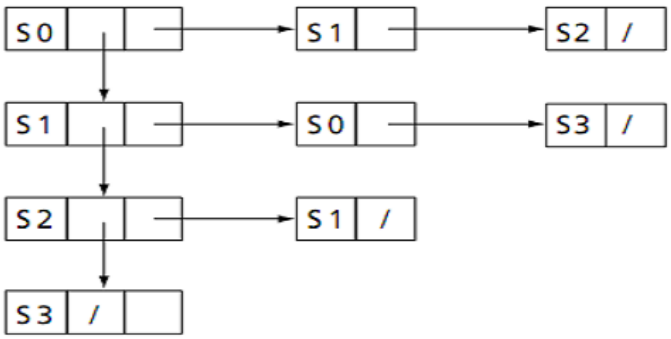


Figure 6.9. La mémorisation sous forme de listes d'adjacence.

Chaque sommet est représenté par la structure suivante :

```
struct sommet{
 /* mettre ici les données relatives aux sommets */
};

struct succ{
 sommet info;
 succ * suivant;
};
```

Donc un graphe peut être représenté par la structure suivante :

```
struct graphe{
 sommet info;
 graphe * suivant;
 succ * adr;
};
```

VI.6 Parcours d'un graphe

Il s'agit d'examiner les sommets d'un graphe pour un traitement donné une et une seule fois. La présence de circuits doit être prise en considération de façon à ne pas visiter plusieurs fois le même sommet. Il faut donc marquer les sommets déjà visités. Comme pour les arbres, on distingue deux types de parcours : le parcours en profondeur et le parcours en largeur.

Principe du parcours en profondeur d'un graphe

On part d'un sommet donné. On énumère le premier fils de ce sommet, puis on repart de ce dernier sommet pour atteindre le premier petit-fils, etc. Il s'agit pour chaque sommet visité, de choisir un des sommets successeurs du sommet en cours, jusqu'à arriver sur une impasse ou un sommet déjà visité. Dans ce cas, on revient en arrière pour repartir avec un des successeurs non visité du sommet courant. En partant de S0 sur la **Figure 6.10**, on peut aller en S1 ou S6. On choisit S1. De S1, on peut aller en S2, S3 ou S5. On choisit S2. De S2, on peut aller en S3, seule possibilité. De S3, on pourrait aller en S1 mais S1 a déjà été marqué. On revient en arrière sur S2 où il n'y a pas d'autre alternative. On revient en arrière sur S1 ; reste à essayer S3 et S5. S3 a déjà été visité. On prend donc le chemin S5. De S5, on ne peut explorer de nouveaux sommets. On revient en S1, puis S0. Pour S0, il reste une alternative vers S6. Tous les sommets n'ont pas été visités en partant de S0. Il faut repartir d'un des sommets non encore visités, et essayer d'explorer en partant de ce sommet. On repart de S4 qui mène à S7.

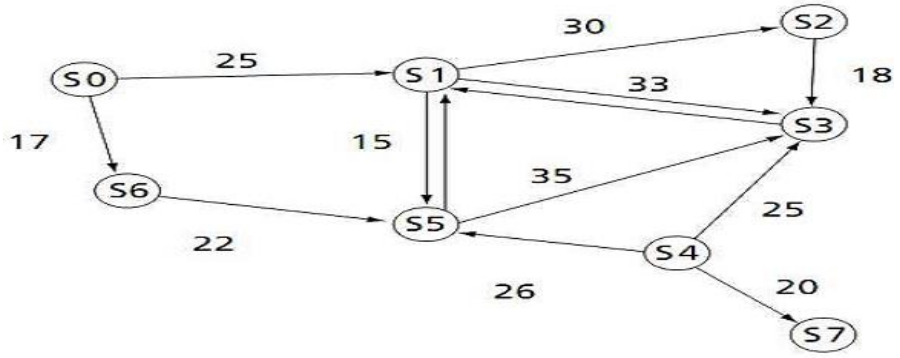


Figure 6.10: Un graphe value de distances entre lieux.

L'ordre de parcours en profondeur du graphe est donc le suivant : **S0, S1, S2, S3, S5, S6, S4, S7.**

Principe du parcours en profondeur d'un graphe

On part d'un sommet donné. On énumère tous les fils (les suivants) de ce sommet, puis tous les petits-fils non encore énumérés, etc. C'est une énumération par génération : les successeurs directs, puis les successeurs au 2e degré, etc. En partant de S0 sur la **Figure 6.10**, on visite S1 et S6. De S1, on visite S2, S3 et S5. De S6, on ne peut pas explorer de nouveaux sommets. De S2, S3 et S5, on ne peut pas explorer de nouveaux sommets. Il faut également repartir d'un sommet non encore visité et non accessible de S0. On repart avec S4 qui nous conduit à S7. Le graphe entier a été parcouru.

L'ordre de parcours en largeur du graphe est donc le suivant : **S0, S1, S6, S2, S3, S5, S4, S7.**

VI.7 Parcours en profondeur d'un graphe

Lors du parcours d'un graphe, on marque les sommets avec une étiquette 0 ou 1 pour indiquer les sommets qui ont déjà été visités. Cela évite de passer plusieurs fois par le même sommet et assure que le programme ne boucle pas. Le parcours en profondeur commence par marquer tous les sommets à 0 (sommets non encore visités).

Le principe du parcours en profondeur récursif est de créer une fonction récursive de visite des sommets. La visite d'un sommet V consiste à marquer V à 1, puis à visiter tous les successeurs de V qui n'ont pas été précédemment visités tour à tour (on reconnaît les sommets qui n'ont pas été visités car ils sont marqués à 0). À la fin de l'algorithme, les sommets qui sont accessibles à partir du sommet de départ par un chemin sont marqués à 1 ; tous les autres sommets sont marqués à 0.

Voici un schéma d'algorithme de parcours en profondeur récursif :

```
Algorithme profondeur;
Var D : sommet;
    G : graphe

DébutFonction visiter(graphe G, sommet V)
marquer le sommet V à 1;
pour chaque successeur S de V faire
 si S est marqué à 0 faire
  Visiter(G, S);
FinPour
FinFonction

/* Programme principal */
Début
 marquer tous les sommets à 0;
 visiter(G, D); /* D est un sommet de départ*/
Fin
```

VI.8 Parcours en largeur d'un graphe

Le parcours en largeur permet, en utilisant une file, de parcourir les sommets d'un graphe à partir d'un sommet de départ D , du plus proche au plus éloigné de D .On visite les voisins de D , puis les voisins des voisins de D ,etc. Le principe est le suivant. Lorsqu'on visite un sommet, on insère ses voisins non encore insérés (marqués à 0) dans une file, et on les marque à 1. On défile les sommets lorsqu'on les visite. On itère le processus tant que la file est non vide. À mesure qu'on insère les sommets dans la file, on les marque pour ne pas les insérer deux fois. Le schéma de l'algorithme est le suivant :

```
Algorithme Largeur

 initialiser une file F à vide;
 marquer tous les sommets à zéro;
 marquer le sommet de départ D à 1;
 insérer D dans F;
 tant que F est non vide faire
  défiler un sommet V de F;
  pour chaque voisin S de V
   si S est marqué à 0 faire
    marquer S à 1;
    insérer S dans F;
   finsi
  finpour
 fin tantque
fin
```

Parcours en largeur

```
void largeur(graphe G, char x){
file F;
int i, j, k;
char y;
succ * tmp;

F.tete = NULL;
j = indice_sommet(G, x);
G.tab[j].marque = 1;
F = enfiler(F, x);
printf("%c  ", G.tab[j].valeur);
while(!est_vide(F)){
 y = defiler(&F);
 j = indice_sommet(G, y);
 tmp = G.tab[j].adr;
 while(tmp != NULL){
  k = tmp->indice;
  if(G.tab[k].marque == 0){
   G.tab[k].marque = 1;
   F = enfiler(F, G.tab[k].valeur);
   printf("%c  ", G.tab[k].valeur);
  }
  tmp = tmp->suivant;
  }
 }
}
```

Parcours en profondeur

```
void profondeur(graphe G, char x){
succ * tmp;
int i, j;

 i = indice_sommet(G, x);
 G.tab[i].marque = 1;
 printf("%c", G.tab[i].valeur);
 tmp = G.tab[i].adr;
 while(tmp != NULL){
  j = tmp->indice;
  if(G.tab[j].marque == 0)
   profondeur(G, G.tab[j].valeur);
  tmp = tmp->suivant;
 }
}
```

Chapitre VII.

Fichiers

VII.1 Les fichiers

Nous avons déjà eu l'occasion d'étudier les « entrées-sorties conversationnelles », c'est-à-dire les fonctions permettant d'échanger des informations entre le programme et l'utilisateur. Nous vous proposons ici d'étudier les fonctions permettant au programme d'échanger des informations avec des fichiers. A priori, ce terme de fichier désigne plutôt un ensemble d'informations situé sur une « mémoire de masse » telle que le disque ou la disquette.

VII.2 Les fichiers textes

Un fichier est une série de données stockées sur un disque ou dans un périphérique de stockage. Un fichier texte est un fichier qui contient du texte ASCII. On appelle lecture dans un fichier le transfert de données du fichier vers la mémoire centrale. La lecture dans un fichier texte est analogue à la lecture au clavier : le texte vient du fichier au lieu de venir du clavier.

On appelle écriture dans un fichier le transfert de données de la mémoire centrale vers le fichier. L'écriture dans un fichier texte est analogue à l'écriture à l'écran : le texte va dans le fichier au lieu de s'afficher. Le but de cette partie est de voir comment un programme C peut lire et écrire des données dans un fichier texte.

Ouverture et fermeture d'un fichier texte

Pour pouvoir utiliser les fichiers texte, on doit inclure la bibliothèque d'entrées-sorties :

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

Pour lire ou écrire dans un fichier texte, nous avons besoin d'un pointeur de fichier. Le pointeur de fichier nous permet de désigner le fichier dans lequel nous souhaitons lire ou écrire. Un pointeur de fichier est de type `FILE *`.
On déclare un tel pointeur comme toute autre variable : `FILE *fp;`

Avant de pouvoir écrire ou lire dans le fichier, il faut lier le pointeur de fichier à un fichier sur le disque. On appelle cette opération l'ouverture du fichier. L'ouverture se fait avec la fonction `fopen`, qui prend en paramètre le nom du fichier sur le disque et le mode d'ouverture du fichier. Cette fonction possède le prototype suivant :

```
FILE * fopen(char * filename, char * mode);
```

Prenons le cas de l'ouverture d'un fichier en lecture, c'est-à-dire qu'on ouvre le fichier uniquement pour pouvoir lire des données dedans.

```
FILE *fp;
fp = fopen("monfichier.txt","r");

/*(exemple de chemin relatif: répertoire local) */
```

Le premier paramètre est le nom du fichier, ou plus exactement le chemin vers le fichier dans l'arborescence des répertoires. Le deuxième paramètre de la fonction `fopen` est le mode et "r" signifie "fichier ouvert en lecture seule". Les différents modes possibles pour un fichier texte sont :

r : lecture seulement ; le fichier doit exister.

w : écriture seulement. Si le fichier n'existe pas, il est créé. S'il existe, son (ancien) contenu est perdu.
a : écriture en fin de fichier (append). Si le fichier existe déjà, il sera étendu. S'il n'existe pas, il sera créé.

r+: mise à jour (lecture et écriture). Le fichier doit exister. Notez qu'alors il n'est pas possible de réaliser une lecture à la suite d'une écriture ou une écriture à la suite d'une lecture, sans positionner le pointeur de fichier. Il est toutefois possible d'enchaîner plusieurs lectures ou écritures consécutives (de façon séquentielle).

w+ : création pour mise à jour. Si le fichier existe, son (ancien) contenu sera détruit. S'il n'existe pas, il sera créé. Notez que l'on obtiendrait un mode comparable à w+ en ouvrant un fichier vide (mais existant) en mode r+.

a+ : extension et mise à jour. Si le fichier n'existe pas, il sera créé. S'il existe, le pointeur sera positionné en fin de fichier.

Les trois derniers modes (lecture-écriture) ne sont pas souvent utilisés pour les fichiers texte, mais nous verrons dans une autre partie comment les utiliser dans les fichiers binaires.

La fonction `fopen` retourne le pointeur `NULL` en cas d'erreur d'ouverture de fichier (fichier inexistant, erreur dans le nom de fichier, ou permissions, en lecture, écriture, selon le cas).

Après avoir utilisé un fichier, il faut le refermer en utilisant la fonction `fclose`. Cette fonction prend en paramètre le pointeur de fichier et ferme le fichier.

```
int fclose(FILE *F);
```

Lire et écrire des données formatées

Lire des données formatées

Pour lire dans un fichier, le fichier doit préalablement avoir été ouvert en mode "r", "r+", "w+", ou "a+".

Pour lire des données numériques (des nombres) ou autres dans un fichier texte, on peut utiliser la fonction **`fscanf`**, qui est analogue à `scanf` sauf qu'elle lit dans un fichier au lieu de lire au clavier.

La fonction **`fscanf`** prend pour premier paramètre le pointeur de fichier, puis les autres paramètres sont les mêmes que ceux de `scanf` (la chaîne de format avec des %d,%f,... , puis les adresses des variables avec des &). La fonction **`fscanf`** retourne le nombre de variables effectivement lues, qui peut être inférieur au nombre de variables dont la lecture est demandée en cas d'erreur ou de fin de fichier. Ceci permet de détecter la fin du fichier ou les erreurs de lecture.

Exemple de lecture dans un fichier texte

Chargement d'un fichier d'entiers en mémoire centrale.
On suppose que dans un fichier texte sont écrits des nombres entiers séparés par des espaces :
10 2 35 752 -5 4 -52 etc.

Nous allons charger ces entiers en mémoire centrale (c'est-à-dire les mettre dans un tableau), puis nous allons afficher le tableau

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

int ChargeFichier(int t[], int n){
 FILE *fp;
```

```
 int i = 0;

 fp = fopen("monfichier.txt", "r");
 if (fp ==NULL){
  printf("erreur de fichier");
  exit(0);
 }
 while (i < n && fscanf(fp, "%d", &t[i])==1)
  i = i+1;
 fclose(fp);
 return i;
}

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

 for (i = 0 ; i < n ; i++)
  printf("t[%d] = %d \n", i, t[i]);
}

int main(){
 int tab[6];
 int n;
 n = ChargeFichier(tab, 6);
 afficher(tab, n);
 printf("n = %d", n);
 return 0;
}
```

La fonction exit de la bibliothèque stdlib.h permet de terminer le programme à l'endroit où elle est appelée (avec éventuellement un code d'erreur passé en paramètre qui est transmis au système).

Notons la condition dans le while qui teste la valeur retournée par **`fscanf`**. Dans cet exemple, on demande à lire une seule valeur dans le **`fscanf`**. La fonction **`fscanf`** doit retourner 1 en cas de succès. À la fin du fichier, la lecture de la valeur échoue dans le **`fscanf`**, et la fonction **`fscanf`** retourne alors une valeur différente de 1. On sort alors du while sans incrémenter i. À mesure des appels de **`fscanf`**, les éléments du tableau sont lus à partir du fichier. À la fin, la variable i contient le nombre d'éléments qui ont été lus dans le fichier. La condition du while vérifie aussi que l'on ne dépasse pas la taille physique du tableau. Il s'agit d'une sécurité qui évite une erreur de segmentation si le fichier est plus long que le programmeur ne l'a prévu.

Notons enfin qu'à chaque lecture par **`fscanf`**, le pointeur de fichier passe à la suite dans le fichier. Le pointeur avance automatiquement dans le fichier lorsqu'on effectue une lecture, sans qu'il n'y ait besoin d'incrémenter une variable.

Ecrire des données formatées

Pour écrire dans un fichier, le fichier doit préalablement avoir été ouvert en mode "w", "a", "r+", "w+" ou "a+".

Pour écrire des données numériques (des nombres) ou autre dans un fichier texte, on peut utiliser la fonction **`fprintf`**, qui est analogue à la fonction printf. La fonction **`fprintf`** prend comme premier paramètre un pointeur de fichier. Les autres paramètres sont les mêmes que pour printf ; le second paramètre est la chaîne de format avec le texte à écrire et les %d,%f, puis suivent les variables à écrire séparées par des virgules.

Exemple d'écriture dans un fichier texte

Voici un programme qui lit un fichier texte contenant des nombres entiers, et écrit un fichier texte contenant les entiers triples (chaque entier est multiplié par 3).

```
#include <stdio.h>

int TestFichier(){
 FILE *fpr, *fpw;
 int n;

 fpr = fopen("fichierLec.txt", "r");
 fpw = fopen("fichierEcr.txt", "w");
 if (fpr==NULL || fpw==NULL)
  return 1;
 while (fscanf(fpr, "%d", &n)==1)
  fprintf(fpw, "%d ", 3*n);
 fclose(fpr);
 fclose(fpw);
 return 0;
}

int main(){
 int codeErr;

 codeErr = TestFichier();
 if(codeErr != 0)
  printf("Erreur d'ouverture de fichier");
 return 0;
}
```

VII.3 Les fichiers binaires

Un fichier texte contient du texte ASCII. Lorsqu'un fichier texte contient des nombres, ces nombres sont codés sous forme de texte à l'aide des caractères '1', '2', etc. Dans ce format, chaque chiffre prend 1 octet en mémoire. On peut visualiser le contenu d'un fichier texte avec un éditeur de texte.
Les fonctions de lecture et écriture dans un fichier texte (**`fscanf`**, **`fprintf`**...) sont analogues aux fonctions de lecture et d'écriture de texte dans une console scanf et printf.

Un fichier binaire contient du code binaire. On ne peut pas visualiser son contenu avec un éditeur de texte. Lorsqu'une variable est écrite dans un fichier binaire, on écrit directement la valeur exacte de la variable, telle qu'elle est codée en binaire en mémoire. Cette manière de stocker les données est plus précise et plus compacte pour coder des nombres. Les fonctions de lecture et d'écriture dans un fichier binaire sont **`fread`** et **`fwrite`** qui lisent et écrivent des blocs de données sous forme binaire.

Ouverture et fermeture d'un fichier binaire

De même que pour les fichiers texte, pour pouvoir utiliser les fichiers binaires, on doit inclure la bibliothèque d'entrées-sorties :

`#include<stdio.h>`

Et déclarer pour chaque fichier un pointeur de fichier : **`FILE`** ***`fp`**;

On ouvre le fichier avec les modes "r", "w", "a", "r+", "w+" ou "a+". Prenons le cas de l'ouverture d'un fichier en lecture seule :

```
FILE *fp;
fp = fopen("monfichier.dat","r");
```

Les différents modes possibles pour un fichier binaire sont identiques à ceux utilisé pour les fichiers textes.

Après avoir utilisé un fichier, il faut le refermer en utilisant **`fclose`**. La fonction **`fclose`** prend en paramètre le pointeur de fichier et ferme le fichier.

Lecture dans un fichier binaire

Pour lire dans un fichier binaire, on lit en général dans le fichier les éléments d'un tableau. Chaque élément du tableau est appelé un bloc. Chaque bloc possède une taille en octets. Par exemple, un char correspond à 1 octet, un float correspond à 4 octets, etc.

La fonction **`sizeof`** donne la taille de chaque type. Par exemple, sizeof(char) vaut 1, et sizeof(float) vaut 4. On utilisera de préférence la fonction **`sizeof`** plutôt qu'une constante comme 1 ou 4 car cela augmente la lisibilité du programme et le programme ne dépend pas du compilateur ou du système. Par exemple la taille d'un int peut être soit 2 soit 4, mais sizeof(int) est toujours correct.

La fonction de lecture **`fread`** prend en paramètre le tableau, la taille de chaque bloc, le nombre de blocs à lire (nombre d'éléments du tableau), et le pointeur de fichier. La taille physique du tableau doit être au moins égale au nombre de blocs lus, pour éviter une erreur mémoire. La fonction **`fread`** transfère les données du fichier binaire vers le tableau.

On peut lire une variable x avec la fonction **`fread`**. Il suffit pour cela de mettre l'adresse **`&x`** de la variable à la place du tableau et de mettre le nombre de blocs égal à **`1`** ; les données sont alors transférées dans la variable.

La fonction **`fread`** retourne le nombre d'éléments effectivement lus. Si ce nombre est inférieur au nombre effectivement demandé, soit une erreur de lecture s'est produite, soit la fin du fichier a été atteinte.

Exemple de lecture dans un fichier binaire

Exemple 1

Supposons qu'un fichier contienne le codage d'un tableau d'entiers que l'on va charger en mémoire centrale. Le fichier contiendra un int et des float. Le premier élément du fichier (de type int) donne le nombre d'éléments de type float qui se trouvent à la suite. Le programme suivant réalise le chargement d'un tableau en mémoire et son affichage dans la console.

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

float* Chargement(char* nomFichier, int * NbElem){
int n, ret;
float *t;

FILE *fp;
```

```
 fp=fopen(nomFichier, "r");
 if (fp == NULL){
  printf("Erreur de fichier");
  exit(1);
 }
 fread(&n, sizeof(int), 1, fp);
 *NbElem = n;
 t = (float*)malloc(n * sizeof(float));
 ret = fread(t, sizeof(float), n, fp);
 if (ret!=n)
  printf("Erreur de lecture ou fin de fichier!");
 fclose(fp);
 return t;
}

void Affichage(float* t, int nb){
 int i;

for (i=0 ; i < nb ; i++)
 printf("%6.2f ", t [i]);
}

int main(){
int nb;
float *t;

 t = Chargement("monfichier.dat", &nb);
 Affichage(t, nb);
 return 0;
}
```

Exemple 2

```
#include <stdio.h>

int main(){
 char nomfich[20];
 int n;
 FILE * fp;

 printf ("nom du fichier à lister : ") ;
 scanf ("%s", nomfich) ;
 fp = fopen(nomfich, "r");
 while (fread (&n, sizeof(int), 1, fp) == 1 && ! feof(fp))
  printf ("\n%d", n);
 fclose (fp);
 return 0;
}
```

`feof(fp)` est une fonction qui retourne la valeur vrai (c'est-à-dire 1) lorsque la fin du fichier a été rencontrée.

Ecriture dans un fichier binaire

Pour écrire dans un fichier binaire, on utilise la fonction **`fwrite`** qui transfère des données de la mémoire centrale vers un fichier binaire. Comme la fonction **`fread`**, la fonction **`fwrite`** prend en paramètre le tableau, la taille de chaque bloc, le nombre de blocs à écrire et le pointeur de fichier. La taille physique du tableau doit être au moins égale au nombre de blocs écrits, pour éviter une erreur mémoire. La fonction **`fwrite`** transfère les données du tableau vers le fichier binaire.

La fonction **`fwrite`** retourne le nombre d'éléments effectivement écrits. Si ce nombre est inférieur au nombre effectivement demandé, il s'est produit une erreur d'écriture (fichier non ouvert, disque plein...).

Exemples d'écriture dans un fichier binaire

Le programme suivant lit au clavier un tableau de nombres réels et les sauvegarde dans un fichier binaire. Le format du fichier est le même que pour l'exemple précédent : on trouve d'abord le nombre d'éléments, puis les éléments à la suite dans le fichier.

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

void LectureTableau(float *t, int n){
int i, nb;

 t = (float*)calloc(n, sizeof(float));
 for (i = 0; i < n; i++)
  scanf("%f", &t [i]);
}

void Sauvegarde(float *t, int nb, char* nomFichier){
FILE *fp;

 fp = fopen(nomFichier, "w");
 if (fp == NULL){
  printf("Erreur de lecture ou fin de fichier!");
  exit(1);
 }
 fwrite(&nb, sizeof(int), 1, fp);
 if (fwrite(t, sizeof(float), nb, fp) != nb)
  printf("Erreur de fichier !");
 fclose(fp);
}

int main(void){
int nb;
float* t;

 printf("Entrez le nombre d'éléments : ");
 scanf("%d", &nb);
 LectureTableau(t, nb);
 Sauvegarde(t, nb, "monfichier.dat");
 return 0;
}
```

Se positionner dans un fichier binaire

À chaque instant, un pointeur de fichier ouvert se trouve à une position courante, c'est-à-dire que le pointeur de fichier est prêt pour lire ou écrire à un certain emplacement dans le fichier. Chaque appel à **`fread`** ou **`fwrite`** fait avancer la position courante du nombre d'octets lus ou écrits. La fonction **`fseek`** permet de se positionner dans un fichier, en modifiant la position courante pour pouvoir lire ou écrire à l'endroit souhaité. Lorsqu'on écrit sur un emplacement, la donnée qui existait éventuellement à cet emplacement est effacée et remplacée par la donnée écrite. Le prototype de la fonction **`fseek`** permettant de se positionner est :

```
int fseek(FILE *fp, long offset, int origine);
```

La fonction modifie la position du pointeur fichier fp d'un nombre d'octets égal à offset à partir de l'origine. L'origine peut être:

- SEEK_SET : on se positionne par rapport au début du fichier ;
- SEEK_END : on se positionne par rapport à la fin du fichier ;
- SEEK_CUR : on se positionne par rapport à la position courante actuelle (position avant l'appel de **`fseek`**).

Exemple d'utilisation de la fonction fseek

L'exemple suivant traite de fichiers d'entiers. La fonction prend un entier i en paramètre permet à l'utilisateur de modifier le $(i + 1)^{ème}$ entier du fichier.

```
void ModifieNombre(FILE *fp, int i){
int n, nouveau;

  fseek(fp, i * sizeof(int), SEEK_SET);
  fread(&n, sizeof(int), 1, fp);
  printf("L'ancien entier vaut %d\n", n);
  printf ("Veuillez entrer la nouvelle valeur");
  scanf("%d", &nouveau);
  fseek(fp, -sizeof(int), SEEK_CUR);
  fwrite(&nouveau, sizeof(int), 1, fp);}
```

Série de TD N° 1

Fonctions et Procédures

Exercice 1

a/ Créez une fonction nommée maxTableau qui renvoie la valeur maximale d'un tableau passé en paramètres. Pour vous aider, voici le prototype de la fonction à créer :

```
int maxTableau(int tab[], int tailleTab);
```

b/ Créez une deuxième fonction nommée copierTableau qui prend en paramètre deux tableaux. Le contenu du premier tableau devra être copié dans le second tableau. Prototype :

```
void copierTableau(int tabOriginal[], int tabCopie[], int tailleTab);
```

Exercice 2

Ecrire un programme qui lit la taille N d'un tableau T de type **int** (taille maximale: 50 éléments), remplit le tableau par des valeurs entrées au clavier et affiche le tableau. Ranger ensuite les éléments du tableau T dans l'ordre inverse sans l'utilisation d'un tableau d'aide. Afficher le tableau résultant.

Idée: Echanger les éléments du tableau à l'aide de deux indices qui parcourent le tableau en commençant respectivement au début et à la fin du tableau et qui se rencontrent en son milieu.

Exercice 3

a/ Ecrire une fonction qui permet de calculer le PGCD (Plus Grand Commun Diviseur) de deux entiers naturels.

b/ Ecrire une deuxième fonction permettant de calculer le PPMC (Plus grand Multiplicateur Commun) de deux entiers naturels en faisant appel à la fonction PGCD.

Exercice 4

Ecrire une fonction qui prend en paramètre une phrase (100 caractères maximum) et un caractère quelconque, et retourne un booléen égale à vrai si ce caractère apparaît dans cette phrase.

Exercice 5

Ecrire une fonction *swap* ayant comme paramètres 2 entiers a et b et qui permet d'échanger le contenu dea et celui de b.

Série de TD N° 2
Pointeurs

Exercice 1

Soit le programme C suivant, complétez le tableau pour chaque instruction du programme :

```
#include<stdio.h>
int a = 1;
int b = 2;
int c = 3;
int *p1, *p2;

int main(){
 p1=&a;
 p2=&c;
 *p1=(*p2)++;
 p1=p2;
 p2=&b;
 (*p1)-=*p2;
 ++(*p2);
 (*p1)*=*p2;
 a=(++(*p2))*(*p1);
 p1=&a;
 *p2=((*p1)/=(*p2));
 return 0;
}
```

a	b	c	P1	P2
1	2	3	&a	&c

Exercice 2

Soit la portion de code suivante :

```
int A[] = {13, 22, 31, 40, 49, 58, 67, 76, 78};
int *P;
P = A;
```

Quelles valeurs ou adresses fournissent chacune des expressions suivantes:

```
a)  *P+3
b)  *(P+3)
c)  &P+1
d)  &A[4]-3
e)  A+3
f)  &A[7]-P
g)  P+(*P-10)
h)  *(P+*(P+8)-A[7])
```

Exercice 3

Ecrire une fonction qui a comme paramètre une chaîne de caractères et qui renvoie le nombre d'occurrences de la lettre 'A' dans cette chaîne de caractères. Utiliser un pointeur P pour parcourir la chaîne.

Exercice 4

Ecrire une fonction qui a pour paramètre un tableau A d'entiers et sa taille. Cette fonction doit ranger les éléments du tableau A dans l'ordre inverse. La fonction utilisera deux pointeurs P1 et P2 pour le parcours du tableau.

Exercice 5

Ecrire une fonction qui a pour paramètre deux tableaux d'entiers A et B et leurs tailles respectives N et M et qui ajoute les éléments de B à la fin de A. Utiliser deux pointeurs PA et PB pour le transfert des éléments.

Exercice 6

Ecrire une fonction à laquelle on envoie un nombre de minutes. Celle-ci renverrait le nombre d'heures et minutes correspondantes.

Exemples :

si on envoie 45, la fonction doit renvoyer 0 heure et 45 minutes ;
si on envoie 60, la fonction doit renvoyer 1 heure et 0 minutes ;
si on envoie 90, la fonction doit renvoyer 1 heure et 30 minutes.

Série de TD N° 3
Allocation dynamique de la mémoire

Exercice 1

Etant donné deux chaines de caractères **CH1** et **CH2** initialisé lors de la déclaration, écrire un programme qui permet de concaténer ces deux chaines de caractères et de tester si **CH2** et une sous chaine de **CH1**.

Exercice 2

Ecrire une fonction qui permet de créer un tableau d'entier de taille **n** et d'initialiser ces composantes par une valeur **val** en utilisant une allocation dynamique de la mémoire, cette fonction prend en arguments la taille et la valeur val et retourne le tableau en question.

Exercice 3

Ecrire une fonction qui permet de copier une chaîne de caractère dans une autre en inversant la casse (Passer de minuscule à majuscule), cette fonction doit avoir comme arguments deux chaînes de caractères créés par allocation dynamique de la mémoire.

Exercice 4

Ecrire un programme qui à partir d'un tableau d'entiers, construits deux tableaux alloués dynamiquement, l'un contenant les valeurs négatives, l'autre les valeurs positives ou nulles.

Exercice 5

Définir un type Date pour des variables formées d'un numéro de jour, d'un nom de mois et d'un numéro d'année.

Ecrire des fonctions de lecture et d'écriture d'une variable de type Date. Dans un premier temps, on ne se préoccupera pas de la validité de la date entrée.

Série de TD N° 4
Listes chainées

Exercice 1

Ecrire une fonction qui permet de retournée l'adresse de l'élément qui se trouve dans la position i (le $i^{éme}$ élément) dans une liste chainées d'entiers.

Exercice 2

Ecrire une fonction qui permet de calculer le nombre d'éléments d'une liste chainées d'entiers.

Exercice 3

Ecrire une fonction qui permet de fusionner deux listes chaînées de réels, cette fonction doit avoir comme arguments les deux listes et devra retournée la liste obtenue après fusion.

Exercice 4

Ecrire une fonction qui prend en paramètre un tableau d'entiers et son nombre d'éléments, et qui permet de créer une liste chaînée contenant les mêmes éléments du tableau.

Exercice 5

Ecrire une fonction qui prend en paramètre une liste chaînée d'entiers et vérifie si elle est triée par ordre croissant.

Exercice 6

Ecrire une fonction qui permet d'inverser l'ordre des éléments d'une liste chainée de nombres réels. Cette fonction doit avoir comme argument une liste chainée et devra retournée la nouvelle liste obtenue.

Exercice 7

Ecrire une fonction qui permet d'insérer un élément dans une liste chaînée d'entier triée par ordre croissant. Cette fonction devra retournée la liste triée obtenue après ajout de l'élément.

Série de TD N° 5
Piles et Files

Exercice 1

On se donne une pile P1 contenant des entiers positifs, écrire une fonction qui permet de déplacer les éléments de P1 dans une pile P2 de façon à avoir dans P2 tous les nombres pairs au dessous des nombres impairs.

Exercice 2

Un problème fréquent d'un compilateur et des traitements de textes est de déterminer si les parenthèses d'une chaîne de caractères sont balancées et proprement incluses l'une dans l'autre. Par exemple, la chaîne ((( )) ( )) ( ) est bien balancée et proprement écrite. Mais les chaînes)( )(ou ( )) ne le sont pas. Ecrire une fonction qui permet de tester si une chaîne de caractères est proprement écrite et bien balancée, pour ce faire utiliser une pile.

Exercice 3

Un palindrome est une chaîne de caractères qui se lit de la même manière de gauche à droite et de droite à gauche. En utilisant un nombre fixe de piles et de files, écrire une fonction qui détermine si une chaîne de caractère est un palindrome. On suppose que la chaîne de caractère est représentée par une liste chainée.

Exercice 4

Etant donné deux files F1 et F2 et une pile P. La file F1 contient une suite d'entiers, F2 et P sont initialement vides. Ecrire une fonction nommée **transferer** qui met dans F2 les entiers impairs de F1 et laisse dans F1 les entiers pairs. Notons que les éléments de F1 doivent être dans le même ordre qu'au début et les éléments de F2 dans un ordre inversé. Par exemple, si la file F1 contient les éléments [2,3,6,11,7,15,18] **(2 est le premier élément de F1)**, la file F1 sera [2, 6, 18] et la file F2 sera [15,7,11,3]. (Utiliser les fonctions de manipulation des piles et des files sans donner leurs codes correspondants).

Exercice 5

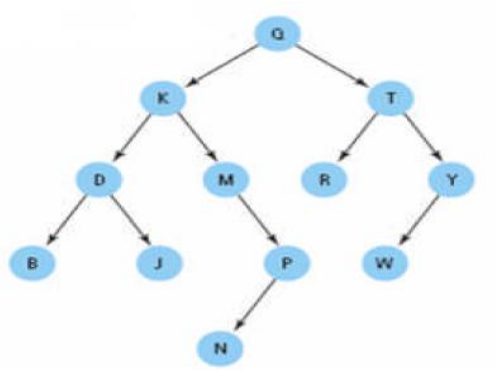
Etant donnée une chaîne de caractères représentant un nombre réel stockée dans une pile P, on considère qu'un nombre réel est représenté par une partie entière et une partie décimale séparées par un point. Ecrire une fonction nommée **calculer** qui possède comme arguments la pile et qui permet de retournée la valeur correspondante au nombre réel stocké dans la pile en utilisant une seule pile intermédiaire.

Exercice 6

On se donne deux piles P et Q. La pile P contient une suite d'entiers, Q est initialement vide. On appel x le sommet initial de la pile P. Ecrire une fonction qui met dans Q les éléments de P inférieur à x, l'élément x puis les éléments supérieurs à x en utilisant un minimum de piles intermédiaires. Notons que les entiers doivent être dans le même ordre qu'au début. Par exemple, si la pile P contient les éléments [11,2,3,10,4,1,8,5] (5 est le sommet de P), la pile Q sera [2,3,4,1,5,11,10,8].

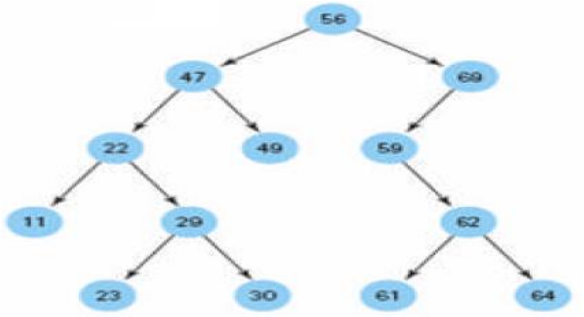
Série de TD N° 6
Arbres binaires

Exercice 1



- Quels sont les nœuds ancêtres du nœud P?
- Quels sont les nœuds descendants du nœud K?
- Quel est le nombre maximum possible de nœud au niveau du nœud W?
- Quel est le nombre maximum possible de nœud au niveau du nœud N?
- Quel est l'ordre des nœuds visités selon la règle infixé, préfixé et postfixé?

Exercice 2



- Quels sont les nœuds du niveau 3?
- Quels niveaux contiennent le nombre maximal de nœuds qu'ils peuvent contenir?
- Quel l'ordre des nœuds visités selon la règle infixé, la règle préfixé et la règle postfixé?

Exercice 3

Donnez l'arbre binaire de recherche dont les éléments sont insérés dans l'ordre suivant:

50 72 96 94 107 26 12 11 9 2 10 25 51 16 17 95

- Donner la liste infixée de l'arbre obtenu. Quelle propriété a-t-elle!? Est-ce toujours le cas!?
- Donner la liste préfixée de l'arbre.
- Donner l'arbre obtenu par suppression de 26, 12 et 107 dans l'arbre.

Exercice 4

1. Ecrire une fonction **`hauteur()`** qui permet de calculer la hauteur d'un arbre binaire.
2. Ecrire une fonction **`max_elem()`** qui permet de rechercher l'élément de plus grande valeur d'un arbre binaire de recherche.

TP N°1 : Allocation dynamique de la mémoire

Objectif

Le but de ce TP est d'écrire des programmes utilisant des chaînes de caractères et des tableaux en se basant sur les pointeurs et l'allocation dynamique de la mémoire tout en utilisant un format d'affichage imposé.

I. Un mot est dit palindrome s'il reste le même qu'on le lise de gauche à droite ou de droite à gauche.

La première ligne de l'entrée contient un mot de taille quelconque.

Ecrivez une fonction qui retourne 1 si le texte est un palindrome et 0 sinon, cette fonction possède un seul argument qui est le mot en question, ce dernier doit être représenté par un tableau créé en utilisant une allocation dynamique de la mémoire.

Exemple :

Entrez un mot à tester : **Rester**

Rester n'est pas palindrome.

II. Ecrire une fonction ***concat()*** qui possède comme argument deux tableaux A et B ainsi que la taille des deux tableaux et qui retourne un tableau C contenant les éléments de A suivi de ceux de B. Votre programme principal doit permettre à l'utilisateur de saisir les éléments de A et de B avant de faire appel à la fonction ***concat()*** puis d'afficher le contenu du tableau retourné par la fonction, sachant que les trois tableaux sont créés en utilisant une allocation dynamique de la mémoire.

Exemple :

Entrer les éléments du tableau A :
A[0] = **10**, A[1] = **12**, A[2] = **5**, A[3] = **15**

Entrer les éléments du tableau B :
B[0] = **1**, B[1] = **2**, B[2] = **25**

Le tableau résultant est :
C[0] = 10, C[1] = 12, C[2] = 5, C[3] = 15, C[4] = 1, C[5] = 2, C[6] = 25

Remarque : Les nombres et les textes en gras sont saisis directement par le clavier.

TP N°2 : Manipulation des listes chaînées

Objectif

Le but de ce TP est d'écrire plusieurs fonctions permettant de manipuler les listes chaînées.

Etant donnée un ensemble d'étudiants qu'on veut représenter par une liste chaînée, chaque étudiant est représenté par son nom et sa moyenne, écrire une fonction correspondante à chacune des opérations suivantes :

- Ajouter un élément enfin de la liste;
- Afficher les éléments de la liste;

Après avoir terminé ces opérations, écrire une fonction qui permet de trier les éléments de la liste par ordre décroissant des moyennes des étudiants.

Votre programme principal doit permettre à l'utilisateur de créer une liste d'étudiants en utilisant les fonctions définies ci-dessus et d'afficher les éléments de la liste. Par la suite, Vous faites appel à la fonction de tri et vous affichez les éléments de la liste triée.

Exemple

```
Etudiant 1
 Nom : Mohamed
 Moyenne : 12.30

Etudiant 2
 Nom : Sara
 Moyenne : 11.80

Etudiant 3
 Nom : Réda
 Moyenne : 15.40

**********LISTE TRIEE**********
Etudiant 1
 Nom : Réda
 Moyenne : 15.40

Etudiant 2
 Nom : Mohamed
 Moyenne : 12.30

Etudiant 3
 Nom : Sara
 Moyenne : 11.80
```

TP N° 3 : Manipulation des piles

Objectif

L'objectif de ce TP est d'écrire une fonction permettant d'effectuer un tri par insertion d'un ensemble d'entiers en utilisant les piles.

Etant donnée une pile A dont les éléments sont des entiers, l'objectif est de construire une pile B contenant les mêmes éléments que A mais dans un ordre trié avec le minimum au sommet de la pile B. Pour ce faire, écrire en langage C les fonctions de manipulation des piles correspondantes aux opérations suivantes :

- Empiler un élément ;
- Dépiler un élément ;
- Retourner le sommet d'une pile ;
- Tester si une pile est vide ;
- Afficher les éléments d'une pile.

Pour la création de la pile B, écrire une fonction ***trier*** qui utilise une pile C vide au début en se basant sur l'algorithme suivant :

```
Algorithme Tri;
Var A, B, C : pile;
    x : entier;
Début
 //Les piles B et C sont vide au début
 Tant que (A n'est pas vide) faire
  Si (B est vide ou sommet(A) ≤ sommet(B)) alors
   x = depiler(A);
   empiler(B, x);
   Tant que (C n'est pas vide) faire
    x = depiler(C);
    empiler(B, x);
   Fin Tant que
  Else
   x = depiler(B);
   empiler(C, x);
  Finsi
 Fin Tant que
Fin
```

Votre programme principal doit permettre à l'utilisateur de créer une pile A qui contient les éléments d'un tableau t = {4, 2, 3, 5, 8, 2, 6, 9, 3, 10}, afficher les éléments de la pile A et de faire appel à la fonction ***trier*** afin de créer la pile B et d'afficher ces éléments trié.

TP N° 4 : Manipulation des files

Objectif

L'objectif de ce TP est d'écrire une fonction permettant d'effectuer la fusion de deux files d'entiers triées par ordre croissant.

Etant donnée deux files A et B dont les éléments sont des entiers triées par ordre croissant, l'objectif est de construire une file C contenant les éléments de A et de B triée par ordre croissant où le minimum est le premier élément de la file C. Pour se faire, écrire en langage C les fonctions de manipulation des files correspondantes aux opérations suivantes :

- Enfiler un élément ;
- Défiler un élément ;
- Retourner le premier élément d'une file ;
- Tester si une file est vide ;
- Afficher les éléments d'une file.

Pour la création de la file C, écrire une fonction ***fusionner*** qui utilise une file C vide au début en se basant sur l'algorithme suivant :

```
Algorithme Fusion;
Var A, B, C : file;
    x : entier;
Début
 //La file C est initialement vide
Tant que (A n'est pas vide et B n'est pas vide) faire
  Si (premier(A) ≤ premier(B)) alors
   x = defiler(A);
  Else
   x  =  defiler(B);
  C = enfiler(C, x);
Fin Tant que
Tant que (A n'est pas vide) faire
 X = defiler(A);
 C = enfiler(C, x);
Fin Tant que
Tant que (B n'est pas vide) faire
 X = defiler(B);
 C = enfiler(C, x);
Fin Tant que
Fin
```

Votre programme principal doit permettre à l'utilisateur de créer deux file A et B qui contiennent les éléments des tableaux t = {2, 6, 13, 15, 18} et s = {6, 8, 11, 16, 30} respectivement, afficher les éléments des files A et B et de faire appel à la fonction ***fusionner*** afin de créer la file C et d'afficher ces éléments triés.

Solution TD N° 1

Exercice 1

```
#include <stdio.h>
int T[50]; /* tableau donné */
 int N;     /* dimension     */
 int I,J;  /* indices courants */
 int AIDE;  /* pour l'échange  */
main()
{
 /* Saisie des données */
 printf("Dimension du tableau (max.50) : ");
 scanf("%d", &N );
 for (I=0; I<N; I++)
    {printf("Elément %d : ", I);
     scanf("%d", &T[I]);}

 /* Affichage du tableau */
 printf("Tableau donné : \n");
 for (I=0; I<N; I++)
     printf("%d ", T[I]);
 printf("\n");

 /* Inverser le tableau */
 for (I=0, J=N-1 ; I<J ; I++,J--)       /* Echange de T[I] et T[J] */
       {
       AIDE = T[I];
        T[I] = T[J];
        T[J] = AIDE;
       }

  /* Edition des résultats */
 printf("Tableau résultat :\n");
 for (I=0; I<N; I++)
     printf("%d ", T[I]);
 printf("\n");
 return 0;
}
```

Exercice 2

a/

```
int maxTableau(int tab[], int tailleTab) {
   int i,max;

   max = tab[0];
   for(i=1;i<tailleTab;i++)
   if(tab[i] > max) max = tab[i];

   return max;
}
```

b/

```
void copierTableau(int tabOriginal[], int tabCopie[], int tailleTab) {
    int i;
    for(i=0;i< tailleTab;i++)
    tabCopie[i] = tabOriginal[i];
}

void afficherTableau(int tab[], int tailleTab) {
   int i;

   printf("Les elements du tableau : \n");
   for(i=0;i<tailleTab;i++)
   printf("%d \n",tab[i]);
}

int main() {
   int a[5] = {8,10,0,5,11};
   int b[5];

   printf("Le max est :  %d \n",maxTableau(a,5));
   copierTableau(a,b,5);
   afficherTableau(b,5);

   return 0;
}
```

Exercice 3

a/

```
int pgcd(int a, int b) {
   int min, max, r;
   if(a > b) {
       max = a;
       min = b;
       }
       else {
           max = b;
```

```
            min = a;}
    do {
       r = max % min;
       max = min;
       min = r;
       }
    while (r != 0);

    return max;
}
```

b/

```
int ppmc(int a, int b) {
    int pgdc = pgcd(a,b);

    if(pgdc!= 0)
    return (a*b) / pgdc;
    else return 0;
}

int main() {
    int x,y;

    printf("Entrer deux entiers naturels x et y");
    scanf("%d",&x);
    scanf("%d",&y);
    printf("PGCD = %d \n",pgcd(x,y));
    printf("PPMC = %d \n",ppmc(x,y));

    return 0;
}
```

Exercice 4

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

int trouver(char x[100], char c){
int i, b ;
 b = 0;
 for (i=0; i< strlen(x); i++)
  if (c == x[i]){
   b = 1;
   break;
  }
 return b;
}

int main()  {
char a[100], car;
int b ;
```

```
printf("Ecrivez une phrase :\n");
scanf("%s", &a);
printf("Donnez un caractère: ");
while(getchar()!='\n');
car=getchar();

b = trouver(a, car);

if (b == 1)
 printf("le caractére apparait dans la phrase");
else
 printf("le caractére n'apparait pas dans la phrase");
return 0;
}
```

Exercice 5

```
#include <stdio.h>

int a, b;
void swap(int *x, int *y);

void swap(int *x, int *y)
{
  int temp;
  temp = *x;
  *x = *y;
  *y = temp;
}

int main()
{
  printf("Tapez la valeur de a : ");
  scanf("%d", &a);
  printf("Tapez la valeur de b : ");
  scanf("%d", &b);
  swap(&a, &b);
  printf("la valeur de a est %d: \n", a);
  printf("la valeur de b est %d: \n", b);
  return 0;
}
```

Solution TD N° 2

Exercice 1

a	b	c	P1	P2
1	2	3	&a	&c
3	2	4	&a	&c
3	2	4	&c	&c
3	2	4	&c	&b
3	2	2	&c	&b
3	3	2	&c	&b
3	3	6	&c	&b
24	4	6	&c	&b
24	4	6	&a	&b
6	6	6	&a	&b

Exercice 2

P = A c'est-à-dire P = &A[0] c'est-à-dire *p = A[0] et p + i = &A[i] et * (p+i) = A[i]

a) **16**
b) **40**
c) **L'adresse mémoire qui vient après l'adresse mémoire du pointeur P.**
d) **&A[4] – 3 = P + 4 – 3 = p + 1 = &A[1]**
e) **&A[3]**
f) **7**
g) **&A[5]**
h) **31**

Exercice 3

```
#include<stdio.h>
char tab[50];
int nb;

int nombre_occ (char t[]);

int nombre_occ (char t[]){
 char * p;
 int n = 0;
```

```
 p = t;
 while (*p != '\0'){
   if (*p == 'A')
   n++;
     p++;
  }
 return n;
}

int main(){
 printf("Entrer la chaine de caractére : ");
 scanf("%s", &tab);
 nb = nombre_occ(tab);
 printf("Le nombre d'occurence de A est : %d", nb);
 return 0;
}
```

Exercice 4

```
#include <stdio.h>

void inversion(int A[], int N);

void inversion(int A[], int N){
 int x, *P1, *P2;

  for (P1=A,P2=A+(N-1); P1<P2; P1++,P2--){
   x = *P1;
   *P1 = *P2;
   *P2 = x;
  }
}

void affichage(int T[], int N) {
 int *P;

  for(P=T; P<T+N; P++)
   printf("T[%d] = %d \n", P-T, *P);
}

int main() {
 int A[6] = {5,4,3,2,1,0};
 int i;
 inversion(A,6);
 printf("Les éléments du tableau A inversé : \n");
 affichage(A,6);
```

Exercice 5

```
#include <stdio.h>

void ajouterEnFin(int A[], int B[], int N, int M);

void ajouterEnFin(int A[], int B[], int N, int M){
  int *PA, *PB;

  for(PA=A+N,PB=B; PB<B+M; PA++, PB++)
    *PA = *PB;
}

void affichage(int T[], int N) {
 int *P;

  for(P=T; P<T+N; P++)
    printf("T[%d] = %d \n", P-T, *P);
}

int main(){
  int A[6] = {0,1,2,3,4,5};
  int B[4] = {6,7,8,9};

  ajouterEnFin(A,B,6,4);
  printf("Les éléments du tableau A résultant : \n");
  affichage(A,10);
  return 0;
}
```

Exercice 6

```
#include <stdio.h>
int horloge(int *pHeures, int *pMinutes);

int horloge(int *pHeures, int *pMinutes) {
 *pHeures = *pMinutes / 60;
 *pMinutes = *pMinutes % 60;
}

int main() {
  int h,m;

  printf("Entrer le nombre de minutes : \n");
  scanf("%d",&m);
  horloge(&h,&m);
  printf("Ceci correspond à %d heures et %d minutes", h, m);
  scanf("%d", &h);
  return 0;
}
```

Solution TD N° 3

Exercice 1

```
#include<stdio.h>
#include<string.h>
char CH1[50] = "classe preparatoire";
char * CH2 = "prepa";
char * p;

 int main(){
  p = strstr(CH1, CH2);
  if (p == NULL)
   printf("%s ne fait pas partie de %s\n", CH2, CH1);
  else
   printf("%s fait partie de %s\n", CH2, CH1);
  strcat(CH1, CH2);
  printf("%s\n", CH1);
  return 0;
}
```

Exercice 2

```
#include<stdio.h>
#include<stdlib.h>
int *t, *p;
int i, n , val;

int * allouer(int n, int val){
int *t;

  t = (int *)malloc(n * sizeof(int));
  for(p = t; p < t + n; p++)
    *p = val;
 return t;
}

int main(){
 printf("Entrez la taille du tableau : ");
 scanf("%d", &n);
 printf("Entrez la valeur de val : ");
 scanf("%d", &val);
 t = allouer(n, val);
```

```
 for(i = 0; i < n; i++)
 printf("t[%d] = %d\n",i, t[i]);
 return 0;
 }
```

Exercice 3

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
char * ch1, * ch2;

void min_maj(char * ch1, char * ch2){
char * p, * q;
 q = ch2;
 for(p = ch1; p < ch1 + strlen(ch1); p++){
  *q = *p - 32;
  q = q + 1;
 }
}
int main(){
 ch1 = (char *)malloc(50 * sizeof(char));
 printf("Entrez une chaine de cractéres : ");
 scanf("%s", ch1);
 ch2 = (char *)malloc((strlen(ch1) + 1) * sizeof(char));
 min_maj(ch1, ch2);
 printf("%s", ch2);
 free(ch1);
 free(ch2);
 return 0;
}
```

Exercice 4

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

void affichage(int T[], int N);

void affichage(int T[], int N) {
 int *P;

 for(P=T; P<T+N; P++)
  printf("T[%d] = %d \n", P-T, *P);
}

int main(){
 int t[6] = {1,-2,3,-4,5,-6};
 int *t1 = NULL, *t2 = NULL, *p = NULL;
 int n = 0, m = 0;
```

```
for(p = t; p < t + 6; p++){
 if(*p >= 0) {
   n = n + 1;
   t1 = (int *)realloc(t1,n * sizeof(int));
   t1[n - 1] = *p;
 }
 else{
  m = m + 1;
  t2 = (int *)realloc(t2,m * sizeof(int));
  t2[m - 1] = *p;
 }
}
printf("Les éléments de t1 : \n");
affichage(t1,n);
printf("Les éléments de t2 : \n");
affichage(t2,m);
free(t1);
free(t2);
free(t);
return 0;
}
```

Exercice 5

```
#include<stdio.h>
typedef struct DAT {
int jour;
char mois[10];
 int annee;
}  DATE ;


void AfficheDate(DATE date) {
printf("La date est: %d %s %d\n", date.jour, date.mois, date.annee);
}

DATE LireDate() {
DATE temp;
scanf("%d %s %d", &temp.jour, temp.mois, &temp.annee);
return temp;
}
Ou bien
void LireDate(DATE * temp) {
scanf("%d %s %d", &temp->jour, temp->mois, &temp->annee);
}
```

```
int main() {
 DATE date;
 date = LireDate();
```
Ou bien
```
LireDate(&date);
AfficheDate(date);
return 0;
}
```

Solution TD N° 4

Exercice 1

```
struct element{
  int val;
  element * suivant;
};

int nombreElements(element * debut){
  int compt = 0;
  element* tmp ;

   tmp = debut;
   while (tmp != NULL){
    compt = compt + 1;
    tmp = tmp->suivant;
   }
    return compt;
  }
}
```

Exercice 2

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

struct element{
  int val;
  element * suivant;
};

element* element_i(element* debut, int i){
  int j;
  element * tmp;

  tmp = debut;
  for(j=0; j<i && tmp != NULL; j++)
   tmp = tmp->suivant;
  return tmp;
}

int main(){
  element* l,
  element *res;

  l = NULL;
  res = element_i(l, 6);
```

```
  if (res == NULL)
   printf("valeur incorrecte de i");
  else
   printf("La valeur qui se trouve à l'indice i est %d", res->val);
}
```

Exercice 3

```
struct element{
 float val;
 element * suivant;
};

element * fusionner (element * debut1, element * debut2){
 element * tmp;

 if (debut1 != NULL){
  if (debut2 != NULL){
  tmp = debut1;
   while(tmp->suivant != NULL){
    tmp = tmp->suivant;
   }
   tmp->suivant = debut2;
   return debut1;
  }
  else
   return debut1;
 }
 else
  return debut2;
}
```

Exercice 4

```
struct element{
 int val;
 element * suivant;
};

element * TabToList(int t[], int n){
 int i;
 element * nouv, * debut, *tmp;

 debut = NULL;
 for(i = 0; i < n; i++){
  nouv = (element*)malloc(sizeof(element));
  nouv->val = t[i];
  nouv->suivant = NULL;
  if(debut == NULL)
  debut = nouv;
  else{
   tmp = debut;
   while(tmp->suivant != NULL)
```

```
    tmp = tmp->suivant;
   tmp->suivant = nouv;
  }
 }
 return debut;
}

int main(){
  element* l;
  int A[] = {5,10,2,8,9};

  l = TabToList (A, 5);
}
```

Exercice 5

```
struct element{
  int val ;
  element * suivant ;
};

int verifie_tri(element * debut){
 element * ptmp, * tmp;

 if (debut == NULL)
  return 0;
 else{
  ptmp = debut;
  tmp = debut->suivant;
  while(tmp != NULL && ptmp->val < tmp->val){
   ptmp = tmp;
   tmp = tmp->suivant;
  }
  if (tmp == NULL)
   return 1;
  else
   return 0;
 }
}
```

Exercice 6

```
struct element{
 float val;
 element * suivant;
};

element * inverser(element * debut){
 element * nl, * tmp, * nouvelElement;

 tmp = debut;
 nl = NULL;
 while(tmp != NULL){
```

```
  nouvelElement = (element*)malloc(sizeof(element));
  nouvelElement->val = tmp->val;
  nouvelElement->suivant = nl;
  nl = nouvelElement;
  tmp = tmp->suivant;
 }
 return nl;
}
```

Exercice 7

```
struct element{
  int val ;
  element * suivant ;
};

element * ajouter(element* debut, int x){
  element * nouveau, * ptmp, * tmp;

  nouveau = (element *) malloc(sizeof(element));
  nouveau->val = x;
  if (debut == NULL){
   nouveau->suivant = NULL;
   return nouveau;
  }
  else{
   if (x < debut->val){
    nouveau->suivant = debut;
    return nouveau;
   }
   else{
    ptmp = debut;
    tmp = debut->suivant;
    while((tmp != NULL)&&(x > tmp->val)){
      ptmp = tmp;
      tmp = tmp->suivant;
    }
    ptmp->suivant = nouveau;
    nouveau->suivant = tmp;
    return debut;
   }
  }
}
```

Solution TD N° 5

La structure utilisée :

```
struct pile{
  int val;
  pile * precedant;
};
```

Exercice 1

```
pile * deplacer(pile * p){
pile *q,  *t;
int x;

 q = NULL;
 t = NULL;

 while(!est_vide(p)){
  x = depiler(&p);
  if (x % 2 == 0)
   q = empiler(q, x);
  else
   t = empiler(t, x);
 }

 while(!est_vide(t)){
  x = depiler(&t);
  q = empiler(q, x);
 }

 return q;
}
```

Exercice 2

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

int i, n;
char t[50];

int balancer(char *t){
 int i, n;
 char x;
 pile * p = NULL;
```

```
 n = strlen(t);
 for(i = 0; i < n; i++)
  if(t[i] == '(')
   p = empiler(p, t[i]);
  else
   if(t[i] == ')')
    if  (!est_vide(p))
    x = depiler(&p);
    else
     return 0;

 if(!est_vide(p))
  return 0;
 else
  return 1;
}

int main(){
 printf("Entrez la chaine de caractére :");
 scanf("%s", t);
 n = strlen(t);
 i = balancer(t);
 if (i == 0)
  printf("La chaine de caractéres n'est pas bien balancer");
 else
  printf("La chaine de caractéres est bien balancer");
 return 0;
}
```

Exercice 3

```
int palindrome(element * debut){
 element * tmp;
 pile * p;
 file f;
 char x, y;

 p = NULL;
 f.tete = NULL;

 tmp = debut;
 while(tmp != NULL){
  p = empiler(p, tmp->info);
  f = enfiler(f, tmp->info);
  tmp = tmp->suivant;
 }

 while(!est_vide(p)){
  x = depiler(&p);
  y = defiler(&f) ;
  if (x != y)
  return 0;
 }
 return 1;
```

Exercice 4

```
file transferer(file * F1){
int x;
file F2;
pile * P;

F2.tete = NULL;
P = NULL;

 while(!est_vide(*F1)){
  x = defiler(F1);
  if(x % 2 == 0)
   F2 = enfiler(F2, x);
  else
   P = empiler(P, x);
 }
 while(!est_vide(F2)){
  x = defiler(&F2);
  *F1 = enfiler(*F1, x);
 }
 while(!est_vide(P)){
  x = depiler(&P);
  F2 = enfiler(F2, x);
 }
 return F2;
}
```

Exercice 5

```
float calculer(pile * p){
 int n, i;
 pile * q;
 char x;
 float val;

 q = NULL;
 val = 0;
 i = 0;

 while(sommet(p) != '.'){
  x = depiler(&p);
  q = empiler(q, x);
 }

 x = depiler(&p);
 while(!est_vide(p)){
 x = depiler(&p);
  n = x - '0';
  val = val + n * pow(10, i);
  i = i + 1;
```

```
 i = 1;
 while(!est_vide(q)){
  x = depiler(&q);
  n = x - '0';
  val = val + n / pow(10, i);
  i = i + 1;
 }
 return val;
}
```

Exercice 6

```
pile * transferer(pile * p){
 pile *q, *t, *v;
 int x, y;
 t = NULL;
 v = NULL;
 q = NULL;
 x = sommet(p);

while(!est_vide(p)){
  y = depiler(&p);
  if (y <= x)
   t = empiler(t, y);
  else
   v = empiler(v, y);
 }
 while(!est_vide(t)){
  x = depiler(&t);
  q = empiler(q, x);
 }
 while(!est_vide(v)){
  x = depiler(&v);
  q = empiler(q, x);
 }
 return q;
}
```

Solution TD N° 6

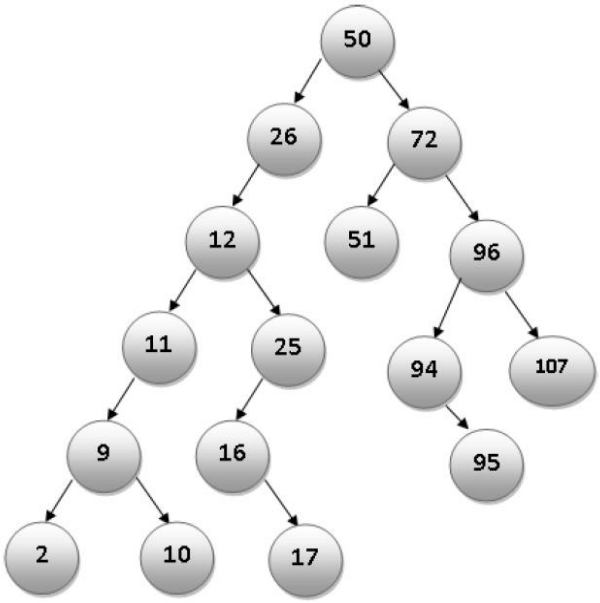
Exercice 1

1. **Les nœuds ancêtres du nœud P sont :** Q, K et M.
2. **Les nœuds descendants du nœud K sont :** D, M, B, J, P et N.
3. **Le nombre maximum possible de nœud au niveau du nœud W est :** $2^3 = 8$.
4. **Le nombre maximum possible de nœud au niveau du nœud N est :** $2^4 = 16$.
5. **Préfixé:** Q, K, D, B, J, M, P, N, T, R, Y, W.
   Infixé: B, D, J, K, M, N, P, Q, R, T, W, Y.
   Postfixé: B, J, D, N, P, M, K, R, W, Y, T, Q.

Exercice 2

1. **Les nœuds du niveau 3 sont :** 11, 29 et 62
2. **Les niveaux qui contiennent le nombre maximal de nœuds qu'ils peuvent contenir sont :** les niveaux 1 et 2.
3. **Préfixé:** 56, 47, 22, 11, 29, 23, 30, 49, 69, 59, 62, 61, 64.
   Infixé: 11, 22, 23, 29, 30, 47, 49, 56, 59, 61, 62, 64, 69.
   Postfixé: 11, 23, 30, 29, 22, 49, 47, 61, 64, 62, 59, 69, 56.

Exercice 3



Exercice 4

On suppose que la racine est au niveau 0.

```
int hauteur(noeud * racine){
if (racine != NULL)
return 1 + max(hauteur(racine->gauche),hauteur(racine->droite));
else
return -1;
}
```

```
int max_abr(noeud * racine){
if (racine->droite == NULL)
return racine->valeur;
else
return max_abr(racine->droite);
}
```

Solution TP 1

Exercice I

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


int n, p, i;
char * t;

int palindrome(char * t){int p
 = 1;
 n = strlen(t);
 for(i = 0; i <= n/2 -1; i++)if
  (t[i] != t[n-i-1]){
  p = 0;
  break;
  }
 return p;
}

int main(){
 t = (char *)malloc(20 * sizeof(char)); printf("Entrez
 une chaine de caractéres : ");scanf("%s", t);
 p = palindrome(t);if
 (p == 1)
  printf("%s est palindrome\n", t);else
  printf("%s n'est pas palindrome\n", t);return
0;
}
```

Exercice II

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

int * A, * B, * C, n, m, i;

int * concat(int * A, int * B, int n, int m);

int * concat(int * A, int * B, int n, int m){
int * C;

C = (int *)malloc((n + m )* sizeof(int));
 for (i = 0; i < n; i++)
  C[i] = A[i];
 for (i = 0; i < m; i++)
  C[n + i] = B[i];
 return C;
}

int main(){
 printf("Entrez la taille du tableau A: ");
 scanf("%d", &n);
 A = (int *)malloc(n * sizeof(int));
 printf("Entrez les éléments du tableau A : \n");
 for (i = 0; i < n; i++){
  printf("A[%d] = ", i);
  scanf("%d", &A[i]);
 }
 printf("Entrez la taille du tableau B: ");
 scanf("%d", &m);
 B = (int *)malloc(m * sizeof(int));
 printf("Entrez les éléments du tableau B : \n");
 for (i = 0; i < m; i++){
  printf("B[%d] = ", i);
  scanf("%d", &B[i]);
 }
 C = concat(A, B, n, m);
 printf("Le tableau résultant est : \n");
 for(i = 0; i < n + m; i++)
 printf("C[%d] = %d\n", i, C[i]);
 return 0;
}
```

Solution TP 2

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

typedef struct element{
  char nom[100];
  float note;
  struct element * suivant;
}element;

element* ajouterEnFin(element * debut, char nom[100], float note){
 element* nouvelElement,* temp;

  nouvelElement = (element*)malloc(sizeof(element));
  strcpy(nouvelElement->nom, nom);
  nouvelElement->note = note;
  nouvelElement->suivant = NULL;
  if(debut == NULL)
   return nouvelElement;
  else{
   temp = debut;
   while(temp->suivant != NULL)
    temp = temp->suivant;
   temp->suivant = nouvelElement;
   return debut;
  }
}

void afficher(element * debut){
 int i = 1;
 element * tmp;

  tmp = debut;
  while(tmp != NULL){
   printf("Etudiant(%d) \n", i);
   printf(" Nom: %s\n", tmp->nom);
   printf(" Moyenne: %6.2f\n", tmp->note);
   i = i + 1;
   tmp = tmp->suivant;
  }
}
```

```
element * trier(element * debut){
 element * ptmp, * tmp;
 char nom[100];
 float x;

  ptmp = debut;
  while (ptmp != NULL){
   tmp = ptmp->suivant;
   while (tmp != NULL){
    if(ptmp->note < tmp->note){
     strcpy(nom, ptmp->nom);
     strcpy(ptmp->nom, tmp->nom);
     strcpy(tmp->nom, nom);
     x = ptmp->note;
     ptmp->note = tmp->note;
     tmp->note = x;
    }
    tmp = tmp->suivant;
   }
   ptmp = ptmp->suivant;
  }
  return debut;
}

int main(){
 element * l;
 l = NULL ;
  l = ajouterEnFin(l, "Mohamed", 16.8);
  l = ajouterEnFin(l, "Sara", 11.5);
  l = ajouterEnFin(l, "Farid", 15.8);
  l = ajouterEnFin(l, "Leila", 12.3);
  l = ajouterEnFin(l, "Réda", 10.8);
  l = ajouterEnFin(l, "Rafik", 10.2);
  afficher(l);
  l = trier(l);
  printf("LISTE TRIEE\n");
  afficher(l);
  return 0;
}
```

Solution TP 3

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

typedef struct pile{
  int donnee;
  struct pile * precedent;
}pile;

pile * empiler(pile * p, int donnee){
 pile * p_nouveau;

 p_nouveau = (pile *) malloc(sizeof(pile));
 if (p_nouveau != NULL)
 {
   p_nouveau->donnee = donnee;
   p_nouveau->precedent = p;
   return p_nouveau;
 }
}

int depiler(pile ** p){
 pile * temp;
 int elem;

 if(p != NULL){
  elem = (*p)->donnee;
  temp = (*p)->precedent;
  free(*p);
  *p = temp;
  return elem;
 }
 else
  printf("La pile est vide");
}

int sommet(pile * p){

if (p == NULL)
 printf("La pile est vide\n");
else
 return p->donnee;
}

int estvide(pile * p){
 if (p != NULL)
  return 0;
 else
 return 1;
}
```

```
pile * afficher(pile* p){
 pile * f = NULL;
 int x;

 while(estvide(p) == 0){
  x = depiler(&p);
  printf(" %d\n", x);
  f = empiler(f, x);
 }
 return f;
}

pile * trier(pile * A){
 pile * B, * C;
 int x;

 B = NULL;
 C = NULL;
 while (estvide(A) == 0){
  if(estvide(B) == 1 || sommet(A) <= sommet(B)){
   x = depiler(&A);
   B = empiler(B, x);
   while(estvide(C) == 0){
   x = depiler(&C);
    B = empiler(B, x);
   }
  }
  else{
   x = depiler(&B);
   C = empiler(C, x);
  }
 }
 return B;
}

int main(){
 int i;
 pile * A, * B;
 A = NULL;
 int t[] = {4, 2, 3, 5, 8, 2, 6, 9, 3, 10};

 for(i = 0; i < 10; i++)
  A = empiler(A, t[i]);
 printf("La pile A contient les éléments: \n");
 A = afficher(A);
 B = trier(A);
 printf("La pile B triée\n");
 afficher(B);
 return 0;
}
```

Solution TP 4

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

struct cellule{
 int donnee;
 cellule * suivant;
};

struct file{
 cellule * tete;
 cellule * queue;
};

file initialiser(){
 file filevide;

 filevide.tete = NULL;
 return filevide;
}

int est_vide(file f){
 if(f.tete == NULL)
 return 1;
 else
  return 0;
}

int premier(file f){
 if(f.tete != NULL)
 return f.tete->donnee;
}

file enfiler(file f, int e){
 cellule * p_nouveau;

 p_nouveau = (cellule *) malloc(sizeof (cellule));
 if (p_nouveau != NULL){
  p_nouveau->suivant = NULL;
  p_nouveau->donnee = e;
  if (f.tete == NULL){
   f.tete = p_nouveau;
   f.queue = p_nouveau;
  }
  else{
   f.queue->suivant = p_nouveau;
   f.queue = p_nouveau;
  }
 }
 return f;
}
```

```
int defiler(file *f){
 cellule * tmp;
 int elem ;

 if (f->tete != NULL){
  elem = f->tete->donnee;
  tmp = f->tete->suivant;
  free(f->tete);
  f->tete = tmp;
  return elem;
 }
}

file afficher(file f){
 file p;
 int x;

 p.tete = NULL;
 while(est_vide(f) == 0){
 x = defiler(&f);
 printf("%d\n", x);
  p = enfiler(p, x);
 }
 return p;
}

file fusionner(file f1, file f2){
 file f;
 int x;

 f.tete = NULL;
 while(!est_vide(f1) && !est_vide(f2)){
  if(premier(f1) <= premier(f2))
   x = defiler(&f1);
  else
   x = defiler(&f2);
  f = enfiler(f, x);
 }
 while(!est_vide(f1)){
  x = defiler(&f1);
  f = enfiler(f, x);
 }
 while(!est_vide(f2)){
  x = defiler(&f2);
  f = enfiler(f, x);
 }
 return f;
}

int main(){
 int i;
 file A, B, C;
 A.tete = NULL;
 B.tete = NULL;
```

```
int t[] = {2, 6, 13, 15, 18};
int s[] = {6, 8, 11, 16, 30};

for(i = 0; i < 5; i++)
A = enfiler(A, t[i]);
for(i = 0; i < 5; i++)
B = enfiler(B, s[i]);
C = fusionner(A, B);

A = afficher(A);
printf("\n*******************************\n");
B = afficher(B);
printf("\n*******************************\n");
C = afficher(C);
return 0;
}
```

Références bibliographiques

Michel DIVAY, Algorithmiques et structure de données. Cours et exercices corrigés en langage C, Editions Dunod, 1999.

MALGOUYRES, Rémy, ZROUR, Rita, et FESCHET, Fabien. Initiation à l'algorithmique et aux structures de données en C. Dunod, 2008.

DELANNOY, Claude. Programmer en langage C: cours et exercices corrigés. Editions Eyrolles, 2016.

DOGHMANE, Hakim. Programmation orientée objet en C++ Cours et Travaux pratiques. 2020.

Frèdèric DEROUILLON, Langage C Maîtriser la programmation procédurale (avec exercices pratiques) (2e édition).

Sergine GUEYE, Algorithmique, Structures des Données et Programmation Pascal et C++.