جامعة ابن طفيل
UNIVERSITÉ IBN TOFAÏL

Faculté des Sciences
Département d'Informatique

Filière : SMI - Semestre (4)

Programmation II

Pr. EL AZAMI

Année Universitaire : 2020/2021

Tables des matières

Partie I : Rappels et compléments du langage C

1 Les types composés

1.1 Introduction

A partir des types prédéfinis du langage **C** (caractères, entiers, flottants), on peut créer de nouveaux types, appelés types composés (tableau, structure, union, énumération, ...). Ils permettent de représenter des ensembles de données organisées.

1.2 Les tableaux

Un tableau est un ensemble fini d'éléments de même type, stockés en mémoire à des adresses contiguës.

La déclaration en **C** d'un tableau à une dimension se fait de la façon suivante :

```
Type nomTableau[nombreElements];
```

Exemple :

```
float tabR[5];              /* tableau de 5 flottants (réels). */
int tabE[8];                /* tableau de 8 entiers.   */
```

Pour plus de clarté, il est recommandé de donner un nom à la constante nombre-éléments par une directive au préprocesseur, par exemple :

```
#define nombreElements 10
```

On accède à un élément du tableau en lui appliquant l'opérateur **[]**.

Les éléments d'un tableau sont toujours numérotés de **0** à **nombreElements -1**.

Exemple :

```
#define N 10
main(){
  int tab[N];
  int i;
  ...
  for (i = 0; i < N; i++)
    printf("tab[%d] = %d\n", i, tab[i]);
}
```

Un **tableau** correspond en fait à un **pointeur constant** (*voir les pointeurs*) vers le premier élément du tableau et aucune opération globale n'est autorisée sur un tableau. Par exemple, on ne peut pas écrire :

```
tab1 = tab2;
```

Il faut effectuer l'affectation pour chacun des éléments du tableau :

```
#define N 10
main(){
  int tab1[N], tab2[N];
  int i;
  ...
  for (i = 0; i < N; i++)
    tab1[i] = tab2[i];
}
```

On peut initialiser un tableau lors de sa déclaration par une liste d'éléments :

Exemple :

```
#define N 4
int tab[N] = {11, 22, 33, 44};
main(){
  int i;
  for (i = 0; i < N; i++)
    printf("tab[%d] = %d\n", i, tab[i]);
}
```

Le programme affichera :

```
tab[0] = 11
tab[1] = 22
tab[2] = 33
tab[3] = 44
```

Cas d'un tableau de caractères : Un tableau de caractères est en fait une chaîne de caractères. Son initialisation peut se faire de plusieurs façons :

```
char p1[10] = 'B','o','n','j','o','u','r';
char p2[10] = "Bonjour";            /* init. par une chaîne litterale */
char p3[] = "Bonjour";              /* p3 aura alors 8 éléments       */
```

ATTENTION ! Le compilateur rajoute toujours un caractère '**\0**' à la fin d'une chaîne de caractères. Il faut donc que le tableau ait au moins un élément de plus.

1.3 Les tableau à n-dimensions

De manière similaire, on peut déclarer un tableau à plusieurs dimensions.

Par exemple, pour un tableau à deux dimensions :

```
Type nomTableau [nombreLignes][nombreColonnes];
```

En fait, un tableau à deux dimensions est un tableau unidimensionnel dont chaque élément est lui-même un tableau.

On accède à un élément du tableau par l'expression "**nomTableau[i][j]**".

Pour initialiser un tableau à plusieurs dimensions à la compilation, on utilise une liste dont chaque élément est une liste de constantes :

```
#define M 2
#define N 3
int tab[M][N] = {{1, 2, 3}, {4, 5, 6}};

main(){
  int i, j;
  for (i = 0 ; i < M; i++){
      for (j = 0; j < N; j++)
        printf("%d\t", tab[i][j]);
    printf("\n");
  }
}
```

Le programme affichera :

```
1       2       3
4       5       6
```

1.4 Les types énumérés

Les énumérations permettent de définir un type par la liste des valeurs qu'il peut prendre. Un objet de type énumération est défini par le mot-clef **enum** et un identificateur de modèle, suivis de la liste des valeurs que peut prendre cet objet :

```
enum modele {constante1, constante2, ..., constanten};
```

En réalité, les objets de type **enum** sont représentés comme des **int**.

Les valeurs possibles **constante1**, **constante2**, **...**, **constanten** sont codées par des entiers de **0** à **n-1**.

Par exemple, le type **enum booleen** défini dans le programme suivant associe l'entier **0** à la valeur **faux** et l'entier **1** à la valeur **vrai**.

```
enum booleen {faux, vrai};
main(){
  enum booleen b;
  b = vrai;
  printf("b = %d", b);
}
```

Le programme affichera :

```
b = 1
```

On peut modifier le codage par défaut des valeurs de la liste lors de la déclaration du type énuméré, par exemple :

```
enum booleen {faux = 12, vrai = 23};
```

Exemple :

```
/* type énuméré couleurs */
enum couleur {
    noir,                           /* 0 = absence de couleur */
    rouge = 1, vert = 2, bleu = 4,  /* couleurs fondamentales */
    jaune = rouge + vert,           /* valeur 3*/
    cyan = vert + bleu,             /* valeur 6 */
    magenta = rouge + bleu,         /* valeur 5 */
    blanc = rouge + vert + bleu     /* valeur 7 */
};
main (){
    enum couleur col;
    col = noir;
    printf ("%d ", col);
    switch(col){
        case 0: printf("noir\n"); break;
        case 1: printf("rouge\n"); break;
        case 2: printf("vert\n"); break;
        case 3: printf("jaune\n"); break;
        case 4: printf("bleu\n"); break;
        case 5: printf("magenta\n"); break;
        case 6: printf("cyan\n"); break;
        case 7: printf("blanc\n"); break;
        default: printf("Ce n'est pas une couleur\n");
    }
    col += rouge;
    printf ("%d ", col);
    switch(col){
        case 0: printf( "noir\n"); break;
        case 1: printf("rouge\n"); break;
        case 2: printf("vert\n"); break;
        case 3: printf("jaune\n"); break;
        case 4: printf("bleu\n"); break;
        case 5: printf("magenta\n"); break;
        case 6: printf("cyan\n"); break;
        case 7: printf("blanc\n"); break;
        default : printf("Ce n'est pas une couleur\n");
    }
    col += cyan;
    printf ("%d ", col);
    switch(col){
        case 0 : printf( "noir\n"); break;
        case 1: printf("rouge\n"); break;
        case 2: printf("vert\n"); break;
        case 3: printf("jaune\n"); break;
        case 4: printf("bleu\n"); break;
        case 5: printf("magenta\n"); break;
        case 6: printf("cyan\n"); break;
        case 7: printf("blanc\n"); break;
        default : printf("Ce n'est pas une couleur\n");
    }
}
```

Le programme affichera :

```
0 noir
1 rouge
7 blanc
```

1.5 Les structures

Une structure est un agrégat de plusieurs objets de types différents regroupés dans une même variable. Chacune des données composant une structure est appelée un champ et peuvent être de types quelconques (type simple, tableaux, autres structures, …). Chacun des champs possède un identificateur qui permet d'accéder directement à l'information qu'il contient.

Exemple :

Pour un étudiant, on peut regrouper dans une seule variable :

- son nom (chaîne de caractères),
- son prénom (chaîne de caractères),
- son âge (entier),
- son sexe (masculin ou féminin),
- son numéro CNE (tableau de 10 chiffres), ...

La déclaration d'un modèle de structure dont l'identificateur est **`modele`** est la suivante :

```
struct modele{
  type1 membre1;
  type2 membre2;
    ...
  typen membren;
};
```

Pour déclarer une variable de type structure correspondante au modèle précédent, on utilise la syntaxe :

```
struct modele objet;
```

ou bien, si le modèle n'a pas été déclaré au préalable :

```
struct modele{
  type1 membre1;
  type2 membre2;
    ...
  typen membren;
}objet;
```

Exemples :

- Une structure de nom **`Etudiant`** correspondant à un étudiant :

```
enum sexes {Feminin, Masculin};

struct Etudiant {
    char nom[20], prenom[20];
    int age;
    enum sexes sexe;
    int numero[5];
} element = {"Aissaoui", "Ali", 22, masculin, {02,49,11,39,42}};
```

- Une structure **`Point`** correspondant à un point de coordonnées **`x`**, **`y`** dans un plan :

```
struct Point {
      int x;
      int y;
};
```

- Une structure de nom **`Adresse`** possible pour coder une adresse postale :

```
struct Adresse {
      int num;
      char rue[40];
      long code;
      char ville[20], pays[20];
};
```

- Une structure de nom **`Individu`** dont l'un des champs a la structure **`Adresse`** précédente (dont la définition est supposée connue) et contenant la définition d'une structure interne anonyme pour le champ de nom **`Identite`** :

```
struct Individu{
       struct {
         char nom[20];
         char prenom[20];
       } Identite;
       int age;
       struct Adresse domicile;
};
```

Il est évidemment possible de créer des alias sur une structure en utilisant **`typedef`** comme dans :

```
typedef struct {
          int a, b;                 // a et b dans la même déclaration
} couple;
```

On accède aux différents membres (champs) d'une structure grâce à l'opérateur membre de structure, noté "**.**".

```
nomObjet.nomChamp
```

Remarque :

- L'affectation globale au moyen de l'opérateur **=** entre deux objets de même structure est maintenant autorisée par la plupart des compilateurs.
- La comparaison (**==**) n'est par contre pas admise généralement. Il est nécessaire de tester chacun des champs l'un après l'autre.

Les règles d'initialisation d'une structure lors de sa déclaration sont les mêmes que pour les tableaux. On écrit par exemple :

```
struct Complexe z = {2. , 2.};
```

En **C**, on peut appliquer l'opérateur d'affectation aux structures (à la différence des tableaux).

Dans le contexte précédent, on peut écrire :

```
struct Complexe z1, z2;
...
z2 = z1;
```

Exemple 1 :

```
#include <math.h>
struct Complexe{
    double reelle;
    double imaginaire;
};
main(){
    struct Complexe z = {3.5, 2.9};
    double norme;
    norme = sqrt(z.reelle * z.reelle + z.imaginaire * z.imaginaire);

    printf("La norme de (%.2f + i %.2f) = %.2f\n",
                                        z.reelle, z.imaginaire, norme);
}
```

Le programme affichera :

```
La norme de (3.50 + i 2.90) = 4.55
```

Exemple 2 :

```
enum propulsion {pedales, moteur, reacteur};
struct vehicule {
    char nom[20];
    int longueur, poids;
    enum propulsion mode;
} velo = {"Euroteam", 2, 5, pedales};
```

```
main () {
    struct vehicule voiture ={"Toyota", 5, 1500, moteur };
    struct vehicule avion;
    printf ("Nom: %s\n", velo.nom);
    printf ("Longueur: %d, poids: %d\n", velo.longueur, velo.poids);
    printf("Mode de propulsion: %d\n ", velo.mode);
    strcpy(avion.nom, "Jumbo");
    avion.longueur = 60;
    avion.poids = 450000;
    avion.mode = reacteur;
    printf ("Nom: %s\n", avion.nom);
    printf ("Longueur: %d, poids: %d\n", avion.longueur, avion.poids);
    printf("Mode de propulsion:%d\n ", avion.mode);
}
```

Le programme affichera :

```
Nom: Euroteam
Longueur: 2, poids: 5
Mode de propulsion: 0
Nom: Jumbo
Longueur: 60, poids: 450000
Mode de propulsion:2
```

1.6 Les champs de bits

Il est possible en **C** de spécifier la longueur des champs d'une structure au bit près si ce champ est de type entier (**`int`** ou **`unsigned int`**). Cela se fait en précisant le nombre de bits du champ avant le "**`;`**" qui suit sa déclaration.

Par exemple, la structure suivante possède deux membres, **`actif`** qui est codé sur un seul bit, et **`valeur`** qui est codé sur 31 bits:

```
struct Registre{
  unsigned int actif : 1;
  unsigned int valeur : 31;
};
```

Tout objet de type **`struct Registre`** est donc codé sur 32 bits.

1.7 Les unions

Une **`union`** désigne un ensemble de variables de types différents susceptibles d'occuper **alternativement** une même zone mémoire. Si les membres d'une **`union`** sont de longueurs différentes, la place réservée en mémoire pour la représenter correspond à la taille du membre le plus grand.

Les déclarations et les opérations sur les objets de type **`union`** sont les mêmes que celles sur les objets de type **`struct`**.

Dans l'exemple suivant, la variable **`hier`** de type **`union Jour`** peut être soit un entier, soit un caractère.

```c
union Jour{
    char lettre;
    int numero;
};

main(){
    union Jour hier, demain;
    hier.lettre = 'J';
    printf("Hier = %c\n", hier.lettre);
    hier.numero = 4;
    demain.numero = (hier.numero + 2) % 7;
    printf("Demain = %d\n", demain.numero);
}
```

Le programme affichera :

```
Hier = J
Demain = 6
```

1.8 Définition de types composés avec typedef

Pour alléger l'écriture des programmes, on peut affecter un nouvel identificateur à un type composé à l'aide de **`typedef`** :

Exemple 1 :

```c
struct Eleve{
    char nom[20];
    char prenom[20];
    float note;
};
typedef struct Eleve Eleve;
 main(){
  Eleve a = {"Alaoui", "Hamza", 17};
  Eleve b;
  strcpy(b.nom, "Hatimi");
  strcpy(b.prenom, "Bouchra");
  b.note = 13;
  printf("Eleve 1 : %s %s %.2f\n", a.nom, a.prenom, a.note);
  printf("Eleve 2 : %s %s %.2f\n", b.nom, b.prenom, b.note);
}
```

Le programme affichera :

```
Eleve 1 : Alaoui Hamza 17.00
Eleve 2 : Hatimi Bouchra 13.00
```

Exemple 2 :

```c
struct Eleve{
    char nom[20];
    char prenom[20];
    float note;
};
typedef struct Eleve Eleve;
```

```c
typedef Eleve TabEleve[100];
main(){
    TabEleve T;
    int i;
    printf("------ Saisie ------\n");
    for(i=0;i<3;i++){
        printf("Eleve %d : ", i+1);
        scanf("%s %s %f", T[i].nom, T[i].prenom, &T[i].note);
    }
    printf("------ Affichage ------\n");
    for(i=0;i<3;i++)
        printf("Eleve %d : %s %s %.2f\n",
                                    i+1, T[i].nom, T[i].prenom, T[i].note);
}
```

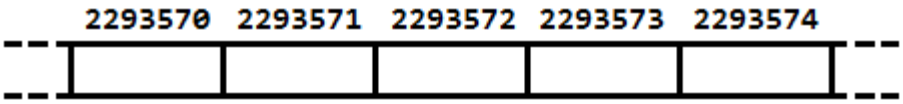
Le programme affichera :

```
------ Saisie ------
Eleve 1 : Hachimi Souad 16
Eleve 2 : Alami Brahim 13
Eleve 3 : Hamdi Merieme 14
------ Affichage ------
Eleve 1 : Hachimi Souad 16.00
Eleve 2 : Alami Brahim 13.00
Eleve 3 : Hamdi Merieme 14.00
```

2 Les pointeurs

2.1 Introduction

Les variables utilisées dans un programme sont stockée quelque part en mémoire centrale. Cette mémoire est constituée d'octets adjacents identifiés par un numéro unique appelé **adresse**.



Pour retrouver une variable, il suffit donc de connaître l'adresse de l'octet où elle est stockée (s'il s'agit d'une variable qui recouvre plusieurs octets adjacents, l'adresse du premier de ces octets).

Rappel : En **C**, l'opérateur d'adresse **&** appliqué à une variable retourne l'adresse-mémoire de cette variable : **`&variable.`**

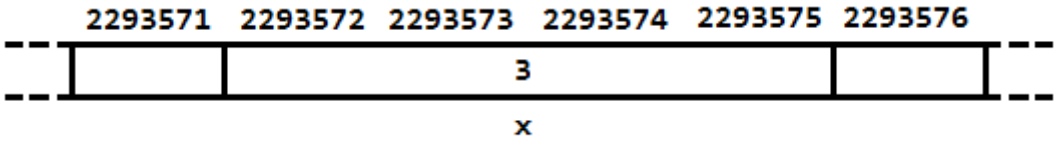
Quand on écrit :

```
int x = 3;
printf("x = %d et se trouve dans l'adresse : %d\n", x, &x);
```

Cela affichera :

```
x = 3 et se trouve dans l'adresse : 2293572
```

Ce fragment de code réserve un emplacement de **4 octets** pour la variable **x** dans la mémoire (les entiers sont codés dans ce cas sur 4 octets), à partir de la case numéro **2293572** dans le cas du schéma et l'initialise avec la valeur **3**.



Rappel : En **C**, l'opérateur fonctionnel **`sizeof(Type)`** retourne le nombre d'octets occupé par le type **`Type.`**

2.2 Adresse et valeur d'un objet

On appelle ***Lvalue*** (*left value*) tout objet pouvant être placé à gauche de l'opérateur d'affectation (**=**).

Une ***Lvalue*** est caractérisée par :

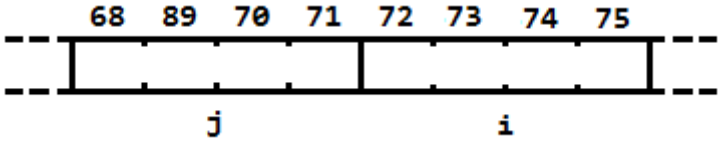
- **son adresse :** l'adresse-mémoire à partir de laquelle l'objet est stocké ;
- **sa valeur :** l'objet stocké à cette adresse.

Dans l'exemple :

```
int i, j;
i = 3;
j = i;
```

Si le compilateur a placé la variable **i** à l'adresse **2293572** en mémoire, et la variable **j** à l'adresse **2293568**, on a :

objet	adresse	valeur
i	2293572	3
j	2293568	3



2.3 Notion de pointeur

Un pointeur est un objet (***Lvalue***) dont **la valeur** est égale à **l'adresse** d'un autre objet.

Un pointeur est déclaré par l'instruction :

```
Type * nomPointeur;
```

où **Type** est le type de l'objet pointé.

Cette déclaration déclare un identificateur, **nomPointeur**, associé à un objet dont la valeur est l'adresse d'un autre objet de type **Type**.

L'identificateur **nomPointeur** est donc en quelque sorte un **identificateur d'adresse**. Comme pour n'importe quelle ***Lvalue***, sa valeur est modifiable.

Exemples :

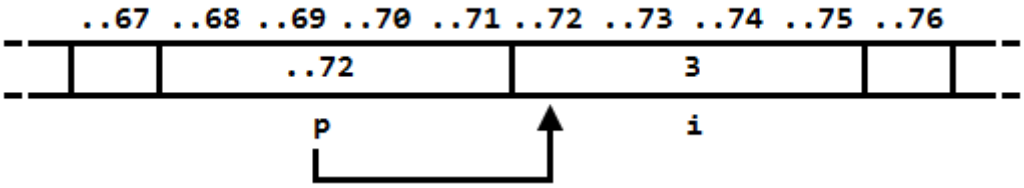
```
int *p1;       /* pointeur sur une variable de type entier */
float *p2;     /* pointeur sur une variable de type float  */
Eleve *p3;     /* pointeur sur une variable de type Eleve  */
double **p5;   /* pointeur sur un pointeur sur un double!  */
```

Dans l'exemple suivant, on définit un pointeur **p** qui pointe vers un entier **i** :

```
int i = 3;
int *p;
p = &i;
```

On se trouve dans la configuration :

objet	Adresse	valeur
i	2293572	3
p	2293568	2293572



L'opérateur unaire d'indirection ***** permet d'accéder directement à la valeur de l'objet pointé.

Si **p** est un pointeur vers un entier **i**, ***p** désigne la valeur de **i**.

Par exemple, le programme :

```
main(){
  int i = 3;
  int *p;
  p = &i;
  printf("*p = %d \n", *p);
}
```

affichera :

```
*p = 3
```

Dans le programme, les objets **i** et ***p** sont identiques : ils ont mêmes adresse et valeur.

Nous sommes dans la configuration :

objet	adresse	valeur
i	**2293572**	**3**
p	**2293568**	**2293572**
***p**	**2293572**	**3**

Cela signifie en particulier que toute modification de ***p** modifie **i**.

Exemple :

```
main(){
    int i = 3;
    int *p;
    p = &i;
    printf("Avant : i = %d et *p = %d \n", i, *p);
    *p = 7;
    printf("Apres : i = %d et *p = %d \n", i, *p);
}
```

Le programme affichera :

```
Avant : i = 3 et *p = 3
Apres : i = 7 et *p = 7
```

On peut donc dans un programme manipuler à la fois les objets **p** et ***p**.

Ces deux manipulations sont très différentes. Comparons par exemple les deux programmes suivants :

(Programme 1)

```
main(){
    int i = 3, j = 6;
    int *p1, *p2;
    p1 = &i;
    p2 = &j;
    *p1 = *p2;
}
```

(Programme 2)

```
main(){
    int i = 3, j = 6;
    int *p1, *p2;
    p1 = &i;
    p2 = &j;
    p1 = p2;
}
```

Avant la dernière affectation des deux programmes, on est dans une configuration du type :

Objet	adresse	valeur
i	2293572	3
j	2293568	6
p1	2293564	2293572
p2	2293560	2293568

Après l'affectation **`*p1 = *p2`** du premier programme, on a :

Objet	adresse	valeur
i	2293572	6
j	2293568	6
p1	2293564	2293572
p2	2293560	2293568

Par contre, l'affectation **`p1 = p2`** du second programme, conduit à la situation :

Objet	adresse	valeur
i	2293572	3
j	2293568	6
p1	2293564	2293568
p2	2293560	2293568

2.4 Arithmétiques des pointeurs

La valeur d'un pointeur étant un entier :

⇨ on peut lui appliquer un certain nombre d'opérateurs arithmétiques classiques.

Les seules opérations arithmétiques valides sur les pointeurs sont :

- **l'addition d'un entier à un pointeur :** Le résultat est un pointeur de même type que le pointeur de départ ;
- **la soustraction d'un entier à un pointeur :** Le résultat est un pointeur de même type que le pointeur de départ ;
- **la différence de deux pointeurs pointant tous deux vers des objets de même type :** Le résultat est un entier.

Notons que la somme de deux pointeurs n'est pas autorisée.

Si **i** est un entier et **p** est un pointeur sur un objet de type **Type** :

- **p + i** désigne un pointeur sur un objet de type **Type** dont la valeur est égale à la valeur de **p** incrémentée de **i * sizeof(type)**.

Il en va de même pour la soustraction d'un entier à un pointeur, et pour les opérateurs d'incrémentation et de décrémentation **++** et **--**.

Exemple :

```
main(){
  int i = 3;
  int *p1, *p2;
  p1 = &i;
  p2 = p1 + 1;
  printf("p1 = %ld \t p2 = %ld\n", p1, p2);
}
```

Le programme affichera :

```
p1 = 2293572      p2 = 2293576
```

Par contre, le même programme avec des pointeurs sur des objets de type double :

```
main(){
  double i = 3;
  double *p1, *p2;
  p1 = &i;
  p2 = p1 + 1;
  printf("p1 = %ld \t p2 = %ld\n", p1, p2);
}
```

Le programme affichera :

```
p1 = 2293568      p2 = 2293576
```

Les opérateurs de comparaison sont également applicables aux pointeurs.

⇨ à condition de comparer des pointeurs qui pointent vers des objets de même type.

L'utilisation des opérations arithmétiques sur les pointeurs est particulièrement utile pour parcourir des tableaux.

Exemple :

```
#define N 5
int tab[5] = {1, 2, 6, 0, 7};
main(){
    int *p;
    printf("Ordre croissant:\t");
    for (p = &tab[0]; p <= &tab[N-1]; p++)
        printf("%d \t",*p);
    printf("\nOrdre decroissant:\t");
    for (p = &tab[N-1]; p >= &tab[0]; p--)
        printf("%d \t",*p);
}
```

Le programme affichera :

```
Ordre croissant:      1     2     6     0     7
Ordre decroissant:    7     0     6     2     1
```

Remarque : Si **p** et **q** sont deux pointeurs sur des objets de type **Type**, l'expression **p - q** désigne un entier dont la valeur est égale à **(p - q)/sizeof(type)**.

2.5 Allocation dynamique

Avant de manipuler un pointeur, il faut l'initialiser. Sinon, par défaut, la valeur du pointeur est égale à une constante symbolique notée **NULL** définie dans **stdio.h**. En général, cette constante vaut **0**.

Le test **p == NULL** permet de savoir si le pointeur **p** pointe vers un objet.

On peut initialiser un pointeur **p** par une affectation sur **p**.

```
p = &a;
```

Il est également possible d'affecter directement une valeur à ***p**, mais pour cela, il faut d'abord réserver à ***p** un espace-mémoire de taille adéquate.

L'allocation de la mémoire en **C** se fait par la fonction **malloc** de la librairie standard **stdlib.h**. dont le prototype est :

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

Le seul paramètre à passer à **malloc** est le nombre d'octets à allouer. La valeur retournée est l'adresse du premier octet de la zone mémoire alloué. Si l'allocation n'a pu se réaliser (par manque de mémoire libre), la valeur de retour est la constante **NULL**.

Pour initialiser un pointeur vers un entier, on écrit :

```
#include <stdlib.h>
main(){
  int *p;
  p = (int*)malloc(sizeof(int));
}
```

On aurait pu écrire également

```
p = (int*)malloc(4);
```

puisqu'un objet de type **int** est stocké sur **4** octets. Mais on préférera la première écriture qui a l'avantage d'être portable.

Le programme suivant définit un pointeur **p** sur un objet ***p** de type **int**, et affecte à ***p** la valeur de la variable **i** :

```c
#include <stdio.h>
#include <stdlib.h>
main(){
    int i = 3;
    int *p;
    printf("valeur de p avant initialisation = %d\n", p);
    p = (int*)malloc(sizeof(int));
    printf("valeur de p apres initialisation = %d\n", p);
    *p = i;
    printf("valeur de *p = %d\n",*p);
}
```

Le programme affichera :

```
valeur de p avant initialisation = 2293576
valeur de p apres initialisation = 5508960
valeur de *p = 3
```

Lorsque l'on n'a plus besoin de l'espace-mémoire alloué dynamiquement, il faut libérer cette place en mémoire. Ceci se fait à l'aide de la fonction **`free`** dont le prototype est :

```c
void free(void *ptr);
```

2.6 Pointeurs et tableaux

L'usage des pointeurs en **C** est, en grande partie, orienté vers la manipulation des tableaux. Tout tableau en **C** est en fait un pointeur constant. Dans la déclaration :

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

`tab` est un pointeur constant (non modifiable) dont la valeur est l'adresse du premier élément du tableau. Autrement dit, **`tab`** a pour valeur **`&tab[0]`**.

On peut donc utiliser un pointeur initialisé à **`tab`** pour parcourir les éléments du tableau.

```c
#define N 5
int tab[5] = {1, 2, 6, 0, 7};
main(){
    int i;
    int *p;
    p = tab;
    for (i = 0; i < N; i++){
        printf("%d\t",*p);
        p++;
    }
}
```

Le programme affichera :

```
1       2       6       0       7
```

On accède à l'élément d'indice **i** du tableau **tab** grâce à l'opérateur d'indexation **[]**, par l'expression **tab[i]**. Cet opérateur d'indexation peut en fait s'appliquer à tout objet **p** de type pointeur. Il est lié à l'opérateur d'indirection ***** par la formule :

```
p[i] == *(p + i)
```

Pointeurs et tableaux se manipulent donc exactement de même manière. Par exemple, le programme précédent peut aussi s'écrire :

```
#define N 5
int tab[5] = {1, 2, 6, 0, 7};
main(){
    int i;
    int *p;
    p = tab;
    for (i = 0; i < N; i++)
        printf("%d\t", p[i]);
}
```

Le programme affichera :

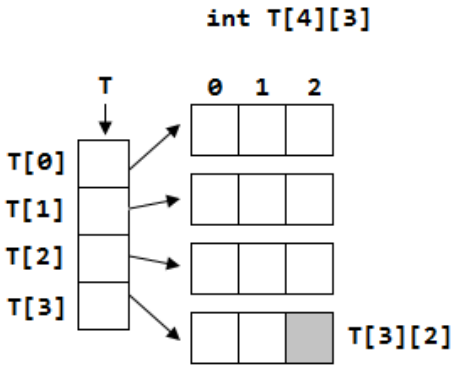
```
1       2       6       0       7
```

Un tableau à deux dimensions est, par définition, un tableau de tableaux. Il s'agit donc en fait d'un pointeur vers un pointeur. Considérons le tableau à deux dimensions défini par :

```
int tab[M][N];
```

tab est un pointeur, qui pointe vers un objet lui-même de type pointeur d'entier et a une valeur constante égale à l'adresse du premier élément du tableau, **&tab[0][0]**.

De même **tab[i]** (pour **i** entre **0** et **M-1**) est un pointeur constant vers un objet de type entier, qui est le premier élément de la ligne d'indice **i**. **tab[i]** a donc une valeur constante qui est égale à **&tab[i][0]**.



On déclare un pointeur qui pointe sur un objet de type **Type *** (deux dimensions) de la même manière qu'un pointeur, c'est-à-dire :

```
Type ** nomPointeur;
```

De même un pointeur qui pointe sur un objet de type **Type ** (équivalent à un tableau à 3 dimensions) se déclare par :

```
Type *** nomPointeur;
```

Par exemple, pour créer avec un pointeur de pointeur une matrice à **k** lignes et **n** colonnes à coefficients entiers, on écrit :

```
main(){
  int k, n;
  int **tab;
  tab = (int**)malloc(k * sizeof(int*));

  for (i = 0; i < k; i++)
    tab[i] = (int*)malloc(n * sizeof(int));
      ...

  for (i = 0; i < k; i++)
    free(tab[i]);
  free(tab);
}
```

2.7 Pointeurs et structures

Contrairement aux tableaux, les objets de type structure en **C** sont des ***Lvalues***. Ils possèdent une adresse, correspondant à l'adresse du premier élément du premier membre de la structure. On peut donc manipuler des pointeurs sur des structures.

Exemple :

```
#include <stdlib.h>
#include <stdio.h>
struct Eleve {
    char nom[20];
    float note;
};
typedef struct Eleve Eleve;
typedef Eleve * Classe;
main(){
    int n, i;
    Classe TE;
    printf("Nombre d'eleves de la classe : ");
    scanf("%d",&n);
    TE = (Classe)malloc(n * sizeof(Eleve));
    for (i =0 ; i < n; i++){
        printf("\nSaisie de l'eleve numero : %d\n",i);
        printf("\tNom : ");    scanf("%s", TE[i].nom);
        printf("\tNote : ");    scanf("%f",&TE[i].note);
    }
    printf("\nEntrez un numero : ");   scanf("%d",&i);
    printf("\nEleve numero %d est : ", i);
    printf("\nNom ===> %s",TE[i].nom);
    printf("\nNote ===> %.2f\n",TE[i].note);
    free(TE);
}
```

Si **p** est un pointeur sur une structure, on peut accéder à un membre de la structure pointé par l'expression :

```
p->membre
```

ou

```
(*p).membre
```

Exemple :

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

struct Eleve {
    char nom[20];
    float note;
};

typedef struct Eleve Eleve;

main(){
    Eleve * pE;
    pE = (Eleve*)malloc(sizeof(Eleve));
    strcpy(pE->nom, "Alami");
    pE->note = 13;
    printf("L'eleve %s a %.2f/20\n", (*pE).nom, (*pE).note);
    free(pE);
    getch();
}
```

Le programme affichera :

```
L'eleve Alami a 13.00/20
```

3 Les fonctions et la récursivité

3.1 Introduction

La structuration de programmes en sous-programmes se fait en **C** à l'aide de fonctions. Les fonctions en **C** correspondent aux fonctions et procédures en langage algorithmique. Créer une fonction est utile quand on a à faire le même type de traitement plusieurs fois dans le programme, mais avec des valeurs différentes.

3.2 Définition d'une fonction

On définit une fonction comme suit :

```
Type nomFonction(Type1 param1 , Type2 param 2, ... , Typen paramn){
   déclaration variables locales ;
   instructions ;
   return (expression) ;
}
```

Remarque : Quand le programme rencontre l'instruction **`return`**, l'appel de la fonction est terminé. Toute instruction située après lui sera ignorée.

Exemples :

```
int produit (int a, int b) {
  return (a * b);
}
float affine(float x ) {
    int a, b;
    a = 3;
    b = 5;
    return (a * x + b) ;
}
float distance(int x, int y){
    return (sqrt(x * x + y * y)) ;
}
float valAbsolue(float x ) {
    return (x < 0) ? (-x) : (x) ;
}
double pi() {
    return (3.14159) ;
}
void messageErreur() {
    printf("Vous n'avez fait aucune erreur\n") ;
}
```

3.3 Appel d'une fonction

Une fonction **f()** peut être appelée depuis le programme principal **main()** ou bien depuis une autre fonction **g()**.

Exemple 1 :

```c
#include <stdio.h>
main(){
    int x, y, r;
    int plus( int x, int y ) ;   /* déclaration de la fonction */
    x = 5;
    y = 235;
    r = plus(x, y);              /* appel d'une fonction avec arguments */
    printf("%d + %d = %d", x, y, r);
}
int plus(int x, int y){
    void mess();                 /* déclaration de la fonction */
    mess() ;                     /* appel d'une fonction sans arguments */
    return (x+y) ;
}
void mess() {
    printf("Vous n'avez fait aucune erreur\n");
    return ;
}
```

Le programme affichera :

```
Vous n'avez fait aucune erreur
5 + 235 = 240
```

Exemple 2 :

```c
double conversion(double euros){
    double dhs;
    dhs = 11 * euros;
    return dhs;
}
main(){
    printf("10 euros = %.2f dhs\n", conversion(10));
    printf("50 euros = %.2f dhs\n", conversion(50));
    printf("100 euros = %.2f dhs\n", conversion(100));
    printf("200 euros = %.2f dhs\n", conversion(200));
}
```

Le programme affichera :

```
10 euros = 110.00 dhs
50 euros = 550.00 dhs
100 euros = 1100.00 dhs
200 euros = 2200.00 dhs
```

Exemple 3 :

```c
#include <stdio.h>
main(){
    /* Prototypes des fonctions appelées */
    int ENTREE(void);
    int MAX(int N1, int N2);
    /* Déclaration des variables */
    int A, B;
    /* Traitement avec appel des fonctions */
    A = ENTREE();
    B = ENTREE();
    printf("Le maximum est %d\n", MAX(A,B));
}
/* Définition de la fonction ENTREE */
int ENTREE(void){
    int NOMBRE;
    printf("Entrez un nombre entier : ");
    scanf("%d", &NOMBRE);
    return NOMBRE;
}
/* Définition de la fonction MAX */
int MAX(int N1, int N2){
    return (N1>N2) ? N1 : N2;
}
```

3.4 Durée de vie des variables

Les variables manipulées dans un programme **C** n'ont pas toutes la même durée de vie. On distingue deux catégories de variables.

- **Les variables permanentes (statiques) :** occupe le même emplacement en mémoire (*segment de données*) durant toute l'exécution du programme. Elles sont initialisées à zéro par le compilateur par défaut et se caractérisent par le mot-clef **`static`**.
- **Les variables temporaires (automatiques) :** se voient allouer un emplacement en mémoire (*segment de pile*) de façon dynamique lors de l'exécution du programme et ne sont pas initialisées par défaut. Leur emplacement en mémoire est libéré par exemple à la fin de l'exécution d'une fonction secondaire.

3.4.1 Variables globales

On appelle variable globale une variable déclarée en dehors de toute fonction et est connue du compilateur dans toute la portion de code qui suit sa déclaration. Les variables globales sont systématiquement permanentes. Dans le programme suivant, **`n`** est une variable globale :

```c
int n;
void fonction(){
  n++;      printf("appel numero %d\n",n);
}
main() {
  int i;
  for (i = 0; i < 5; i++)
    fonction();
}
```

Le programme affichera :

```
appel numero 1
appel numero 2
appel numero 3
appel numero 4
appel numero 5
```

3.4.2 Variables locales

On appelle variable locale une variable déclarée à l'intérieur d'une fonction (ou d'un bloc d'instructions) du programme. Par défaut, les variables locales sont temporaires. Quand une fonction est appelée, elle place ses variables locales dans la pile. A la sortie de la fonction, les variables locales sont dépilées et donc perdues. Les variables locales n'ont en particulier aucun lien avec des variables globales de même nom.

Exemple :

```c
int n = 10;
void fonction(){
  int n = 0;
  n++;
  printf("appel numero %d\n",n);
  return;
}
main(){
  int i;
  for (i = 0; i < 5; i++)
    fonction();
}
```

Le programme affiche :

```
appel numero 1
appel numero 1
appel numero 1
appel numero 1
appel numero 1
```

Les variables locales à une fonction ont une durée de vie limitée à une seule exécution de cette fonction. Leurs valeurs ne sont pas conservées d'un appel au suivant.

Il est toutefois possible de créer une variable locale de classe statique en faisant précéder sa déclaration du mot-clef **`static`** :

```c
static Type nomVariable;
```

Une telle variable reste locale à la fonction dans laquelle elle est déclarée, mais sa valeur est conservée d'un appel au suivant. Elle est également initialisée à zéro à la compilation. Par exemple, dans le programme suivant, **`n`** est une variable locale à la fonction secondaire fonction, mais de classe **statique**.

Exemple :

```
int n = 10;
void fonction(){
  static int n;
  n++;
  printf("appel numero %d\n",n);
  return;
}
main(){
  int i;
  for (i = 0; i < 5; i++)
    fonction();
}
```

Le programme affichera :

```
appel numero 1
appel numero 2
appel numero 3
appel numero 4
appel numero 5
```

On voit que la variable locale **n** est de classe statique (elle est initialisée à zéro, et sa valeur est conservée d'un appel à l'autre de la fonction). Par contre, il s'agit bien d'une variable locale, qui n'a aucun lien avec la variable globale du même nom.

3.5 Transmission des paramètres d'une fonction

Les paramètres d'une fonction sont traités de la même manière que les variables locales de classe automatique. La fonction travaille alors uniquement sur cette copie qui disparaît lors du retour au programme appelant. Cela implique en particulier que, si la fonction modifie la valeur d'un de ses paramètres, seule la copie sera modifiée (La variable du programme appelant, elle, ne sera pas modifiée). On dit que les paramètres d'une fonction sont transmis par valeurs.

Exemple :

```
void echange (int x, int y){
    int t;
    printf("Debut fonction :\n x = %d \t y = %d\n", x, y);
    t = x;      x = y;     y = t;
    printf("Fin fonction :\n x = %d \t y = %d\n", x, y);
}
main(){
    int a = 2, b = 5;
    printf("Debut programme principal :\n a = %d \t b = %d\n", a, b);
    echange(a, b);
    printf("Fin programme principal :\n a = %d \t b = %d\n", a, b);
}
```

Le programme affichera :

```
Debut programme principal :
 a = 2   b = 5
Debut fonction :
 x = 2   y = 5
Fin fonction :
 x = 5   y = 2
Fin programme principal :
 a = 2   b = 5
```

Pour qu'une fonction modifie la valeur d'un de ses arguments, il faut qu'elle ait pour paramètre l'adresse de cet objet et non sa valeur (sa copie). Par exemple, pour échanger les valeurs de deux variables, il faut écrire :

```c
void echange (int * adrA, int * adrB){
    int t = *adrA;
    *adrA = *adrB;
    *adrB = t;
}
main(){
    int a = 2, b = 5;
    printf("Debut programme principal :\n a = %d \t b = %d\n", a, b);
    echange(&a, &b);
    printf("Fin programme principal :\n a = %d \t b = %d\n", a, b);
}
```

Le programme affichera :

```
Debut programme principal :
 a = 2   b = 5
Fin programme principal :
 a = 5   b = 2
```

Rappelons qu'un tableau est un pointeur (sur le premier élément du tableau). Lorsqu'un tableau est transmis comme paramètre à une fonction secondaire, ses éléments sont donc modifiés par la fonction.

Exemple :

```c
void init (int tab[], int n){
    int i;
    for (i = 0; i < n; i++)
        tab[i] = i;
}
main(){
    int i;
    int T[5];
    init(T, 5);
    for (i = 0; i < 5; i++){
        printf("%d\t",T[i]);
    }
}
```

Exemple de fonctions utilisant une structure :

```
struct Eleve{
    int code;
    char nom[20];
    float note;
};
typedef struct Eleve Eleve;

void initEleve(Eleve * adrE, int c, char nm[20], float nt){
    adrE->code = c;      strcpy(adrE->nom, nm);      adrE->note = nt;
}
void modifierNote(Eleve * adrE, float nt){
    adrE->note = nt;
}
void afficher(Eleve E){
    printf("Eleve : %d %s %.2f\n", E.code, E.nom, E.note);
}
main(){
    Eleve a;
    initEleve(&a, 111, "Khalid", 13);
    afficher(a);
    modifierNote(&a, 16);
    afficher(a);
}
```

Le programme affichera :

```
Eleve : 111 Khalid 13.00
Eleve : 111 Khalid 16.00
```

3.6 Pointeurs de fonction

Comme une fonction n'est pas un objet ou une variable, il n'est pas possible de passer une fonction directement en argument à une fonction. Par contre, les fonctions ont une adresse. Il est donc possible de déclarer un pointeur vers cette adresse et de passer ce pointeur à une fonction.

Voici un exemple de la déclaration d'un pointeur qui pointe successivement vers les fonctions **`fonction1()`** et **`fonction2()`** :

```
void fonction1() {
  printf("affiche fonction 1\n");
}
void fonction2() {
   printf("affiche fonction 2\n");
}
main() {
   void (*fonction)();
   fonction = fonction1;
   fonction();
   fonction = fonction2;
   fonction();
}
```

Le programme affichera :

```
affiche fonction 1
affiche fonction 2
```

Exemple de fonctions utilisant une structure :

```c
#include <math.h>

// typedef pour simplifier la notation
typedef double(*Fonction)(double );

// Liste des fonctions "calculables"
double carre(double x) { return x*x;}
double inverse(double x) { return 1/x;}
double racine(double x) { return sqrt(x);}
double exponentielle(double x) { return exp(x);}

double minimum(double a, double b, Fonction f){
    double x;
    double min = 100000;
    for(x=a; x<b ; x+= 0.01)
        min = min< f(x)? min : f(x);
    return min;
}

main(){
    printf("De quelle fonction voulez-vous chercher le minimum ?\n");
    printf("1  --  x^2\n");
    printf("2  --  1/x\n");
    printf("3  --  racine de x\n");
    printf("4  --  exponentielle de x\n");
    printf("5  --  sinus de x\n");

    int reponse;
    scanf("%d", &reponse);
    //On declare un pointeur sur fonction
    Fonction monPointeur;

    //Et on déplace le pointeur sur la fonction choisie
    switch(reponse){
        case 1:  monPointeur = carre; break;
        case 2:  monPointeur = inverse;  break;
        case 3:  monPointeur = racine;  break;
        case 4:  monPointeur = exponentielle;  break;
        // On peut même utiliser les fonctions de math.h !
        case 5:  monPointeur = sin; break;
    }
    //Finalement on affiche le résultat de l'appel de la fonction via le pointeur
    printf("Le minimum de la fonction entre 3 et 4 est : %.2f\n",
                                                    minimum(3, 4, monPointeur));
}
```

Le programme affichera :

```
De quelle fonction voulez-vous chercher le minimum ?
1  --  x^2
2  --  1/x
3  --  racine de x
4  --  exponentielle de x
5  --  sinus de x
4
Le minimum de la fonction entre 3 et 4 est : 20.09
```

3.7 Les fonctions récursives

Une fonction est dite récursive lorsqu'elle est définie en fonction d'elle-même. La programmation récursive est une technique de programmation qui remplace les instructions de boucle (**`while`**, **`for`**, etc.) par des appels de fonction (**N.B. :** ne pas confondre avec la notion de récursivité en mathématiques).

3.7.1 Premier exemple

Le mécanisme le plus simple pour faire boucler une fonction : elle se rappelle elle-même.

Exemple :

```
void boucle() {
  boucle();
}
```

On peut en profiter pour faire quelque chose :

```
void boucle() {
  printf ("Je tourne \n");
  boucle();
}
```

C'est ce qu'on appelle une récursivité simple. Il faut encore ajouter un mécanisme de test d'arrêt.

Exemple :

Ecrire 100 fois "**`Je tourne`**" : on a besoin d'un compteur. On choisit ici de le passer d'un appel de fonction à l'autre comme un paramètre et il faut appeler **`boucle(0)`**.

```
void boucle(int i) {
  if (i < 100) {
    printf ("Je tourne \n") ;
    boucle(i+1) ;
  }
  //sinon on ne relance pas => fin du programme
}
```

3.7.2 Apprendre à programmer récursivement avec des variables

Calculer la somme des **`n`** premiers nombres avec une boucle **`while`** :

```
int somme(int n) {
  int s = 0;
  int i = 1;
  while (i <= n) {
    s += i;
    i++;
  }
  return s ;
}
```

Par une procédure récursive, **méthode très naïve** :

```
int n = 100 ;
int s = 0 ;
void sommeRec(int i) {
  if (i <= n) {
    s += i ;
    sommeRec(i+1) ;
  }
  //si i = n, fin du programme
}
```

On lance **`sommeRec(0);`**

C'est extrêmement maladroit parce que :

1. on ne contrôle pas la valeur de **`s`** au début
2. on ne contrôle pas la valeur terminale **`N`**.

Pour éviter cela, il faut accéder à **`s`** et **`n`** sans qu'elles soient en variables globales. Une solution pour construire une procédure récursive sérieuse est d'utiliser pour passer **`s`** et **`n`** les paramètres et la valeur de retour de la fonction.

```
int sommeRec(int i, int s, int n) {
  if (i <= n)
    return sommeRec(i+1, s+i, n) ;
  else
    return (s) ;                    // sommeRec(n,s,n), on a fini
}
```

On lance **`sommeRec(0, 0, 100);`**

On aurait pu éviter de passer à la fois **`i`** et **`n`**, en comptant à l'envers de **`n`** à **`0`** :

```
int sommeRec(int s, int n) {
  if (n > 0)
    return sommeRec(s + n, n - 1 );
  else
    return (s) ;                   // Pour sommeRec(0, s), le calcul est immédiat
}
```

On lance **`x = sommeRec(0, 100);`**

Et on pouvait même éviter de passer s, en gérant plus efficacement la valeur de retour :

```
int sommeRec(int n) {
  if (n > 0)
    return sommeRec(n - 1 ) + n;
  else
    return (0) ;            // pour sommeRec(0), le calcul est immédiat
}
```

On lance **`x = sommeRec(100);`**

Enfin, on peut s'apercevoir qu'il est plus astucieux de programmer une fonction plus générale :

`sommeRec(debut, fin)` : c'est "faire la somme des nombres de **`debut`** jusqu'à **`fin`** ".

La programmation récursive, c'est : on appelle la même fonction avec un nombre de moins dans la liste, puis on ajoute ce nombre au résultat.

D'où deux versions :

```
int sommeRec(int deb, int fin) {
  if (fin >= deb)
    return sommeRec(deb, fin - 1 ) + fin;
  else
    return (0) ;           // pour sommeRec(x,x), le calcul est immédiat
}
```

ou

```
int sommeRec(int deb, int fin) {
  if (fin >= deb)
    return deb + sommeRec(deb + 1, fin );
  else
    return (0) ;          // pour sommeRec(x,x), le calcul est immédiat
}
```

Dans les deux cas, on lance **`sommeRec(0, 100)`**, **`sommeRec(10, 50)`**, etc.

3.7.3 Différents types de récursivité

a. Récursivité simple

Une fonction simplement récursive, c'est une fonction qui s'appelle elle-même une seule fois, comme c'était le cas pour **`sommeRec()`** ci dessus.

Prenons à la fonction puissance $x \to x^n$ x. Cette fonction peut être définie récursivement :

$$x^n = \begin{cases} 1 & si \;\; n = 0; \\ x \times x^{n-1} & si \;\; n \geq 1. \end{cases}$$

La fonction correspondante s'écrit :

```
int puissance(int x, int n){
    if(n == 0)
        return 1;
    else
        return x * puissance(x, n-1);
}
```

b. Récursivité multiple

Une fonction peut exécuter plusieurs appels récursifs.

Par exemple le calcule des combinaisons C_n^p en se servant de la relation de Pascal :

$$C_n^p = \begin{cases} 1 & si \quad p = 0 \quad ou \quad p = n; \\ C_{n-1}^p + C_{n-1}^{p-1} & sinon. \end{cases}$$

La fonction correspondante s'écrit :

```
int combinaison(int n, int p){
    if(p == 0 || p == n)
        return 1;
    else
        return combinaison(n-1, p) + combinaison(n-1, p-1);
}
```

c. Récursivité mutuelle

Des fonctions sont dites mutuellement récursives si elles dépendent les unes des autres. Par exemple, deux fonctions **`A(x)`** and **`B(x)`** définies comme suit :

$$A(x) = \begin{cases} 1 & si \quad x \leq 1; \\ B(x+2) & si \quad x > 1. \end{cases} \qquad B(x) = \{A(x-3) + 4$$

Les fonctions correspondantes s'écrivent :

```
int A(int x){
    if(x <= 1)
        return 1;
    else
        return B(x+2);
}
int B(int x){
    return A(x-3) + 4;
}
```

d. Récursivité imbriquée

Une fonction contient une récursivité imbriquée s'il contient comme paramètre un appel à lui-même.

C'est le cas de la fonction d'Ackermann définie comme suit :

$$A(m,n) = \begin{cases} n+1 & si \;\; m = 0; \\ A(m-1,1) & si \; m > 0 \quad et \;\; n = 0; \\ A\big(m-1, A(m,n-1)\big) & sinon. \end{cases}$$

La fonction correspondante s'écrit :

```
int ackermann(int m, int n) {
    if (m == 0)
        return n + 1;
    else
        if (n == 0)
            return ackermann (m - 1, 1);
        else
            return ackermann (m - 1, ackermann (m, n - 1));
}
```

3.7.4 Principe et dangers de la récursivité

Une fonction récursive est dite bien définie si elle possède les deux propriétés suivantes:

- Il doit exister certains critères, appelés critères d'arrêt ou conditions d'arrêt, pour lesquels la fonction ne s'appelle pas elle-même.
- Chaque fois que la procédure s'appelle elle-même (directement ou indirectement), elle doit converger vers ses conditions d'arrêt.

4 Les fichiers

4.1 Introduction

Le **C** offre la possibilité de lire et d'écrire des données dans un fichier et pour des raisons d'efficacité, les accès à un fichier se font par l'intermédiaire d'une mémoire-tampon (*buffer*).

Pour manipuler un fichier, un programme a besoin d'un certain nombre d'informations :

- l'adresse de l'endroit de la mémoire tampon où se trouve le fichier,
- la position de la tête de lecture
- le mode d'accès.

Ces informations sont rassemblées dans une structure **`FILE *`** défini dans **`stdio.h`**. Un objet de type **`FILE *`** est appelé *flot de données* déclaré comme suit :

```
FILE * pF;
```

4.2 Ouverture et fermeture d'un fichier

4.2.1 La fonction fopen

L'ouverture d'un fichier est réalisée par la fonction **`fopen`**. Son prototype est :

```
FILE * fopen(const char * path, const char * mode);
```

La valeur retournée par **`fopen`** est un flot de données. Si l'exécution de cette fonction ne se déroule pas normalement, la valeur retournée est le pointeur **`NULL`**. Il est donc recommandé de toujours tester si la valeur renvoyée par la fonction **`fopen`** est égale à **`NULL`** afin de détecter les erreurs (lecture d'un fichier inexistant...).

La chaîne de caractères constante **`path`** est le chemin (absolu ou relatif) du fichier à ouvrir. La seconde variable, nommée **`mode`**, indique le mode d'ouverture du fichier. Voici les différents modes d'ouvertures disponibles :

- "**`r`**" : **lecture** à partir du début du fichier. Le fichier indiqué par le premier argument doit obligatoirement exister, sinon la fonction échoue.
- "**`w`**" : **écriture** à partir du début du fichier. Si le fichier n'existe pas, il sera créé. S'il existe, son contenu est effacé.
- "**`a`**" : **écriture** à partir de la fin du fichier. Si le fichier n'existe pas, il sera créé.
- "**`r+`**" : **lecture** et **écriture** à partir du début du fichier. Le fichier indiqué par le premier argument doit obligatoirement exister, sinon la fonction échoue.
- "**`w+`**" : **lecture** et **écriture** à partir du début du fichier. Si le fichier n'existe pas, il sera créé. S'il existe, son contenu est effacé.

- "**a+**" : **lecture** et **écriture** à partir de la fin du fichier. Si le fichier n'existe pas, il sera créé.

Pour une ouverture en modification, le flux peut être utilisé en lecture et en écriture. La norme ANSI propose, pour n'importe quel mode mentionnés plus haut, d'ajouter en fin de chaîne le caractère '**b**' ou '**t**' pour traiter le flux en **mode binaire** ou en **mode texte**. Le mode binaire permet, sur certaines machines, de ne pas étendre le caractère '**\n**' en un retour de chariot '**\r**' suivi du caractère d'interligne. Le mode texte effectue cette transformation par défaut sur les PC.

Trois flots standards peuvent être utilisés en **C** sans qu'il soit nécessaire de les ouvrir ou de les fermer :

- **stdin** (*standard input*) : unité d'entrée (par défaut, le clavier) ;
- **stdout** (*standard output*) : unité de sortie (par défaut, l'écran) ;
- **stderr** (*standard error*) : unité d'affichage des messages d'erreur (par défaut, l'écran).

Il est fortement conseillé d'afficher systématiquement les messages d'erreur sur **stderr** afin que ces messages apparaissent à l'écran même lorsque la sortie standard est redirigée.

4.2.2 La fonction fclose

Elle permet de fermer le flot qui a été associé à un fichier par la fonction **fopen**. Son prototype est :

```
int fclose(FILE * flot)
```

L'entier retourné par **fclose** vaut zéro si l'opération s'est déroulée normalement (et une valeur non nulle en cas d'erreur).

Exemple :

```
#include <stdio.h>
main(){
    FILE * pF;
    char nomFichier[] = "donnee.txt";
    pF = fopen(nomFichier, "r");
    if(pF == NULL){
        printf("Impossible d'ouvrir le fichier %s\n", nomFichier);
        exit(1);
    }

    /* Opérations sur le fichier ... */

    fclose(pF) ;
    getch();
}
```

L'appel à la fonction **exit()** a pour effet, non seulement de terminer le programme, mais aussi de vider tous les tampons et de fermer tous les fichiers ouverts. Il est généralement nécessaire d'inclure le fichier de déclarations **stdlib.h** pour pouvoir utiliser cette fonction.

4.3 Les entrées-sorties formatées

4.3.1 La fonction d'écriture `fprintf`

La fonction **`fprintf`**, analogue à **`printf`**, permet d'écrire des données dans un fichier. Son prototype est :

```
int fprintf (FILE * stream, const char * format, ...)
```

Elle écrit tous les arguments passés en paramètres dans le flot **`stream`** : elle fonctionne exactement comme **`printf`**, à la différence près que le premier argument est un **`FILE*`**. Voici un exemple qui écrit dans un flux le pseudo et le score d'un joueur :

```
#include <stdio.h>

main(){
    FILE * pF;
    const char * pseudo = "Lecteur";
    const int score = 50;

    pF = fopen("scores.txt", "w");
    if(pF == NULL){
        printf("Impossible d'ouvrir le fichier\n");
        exit(1);
    }
    fprintf(pF, "%s %d\n", pseudo, score);

    fclose(pF);
    pF = NULL;
    return 0;
}
```

4.3.2 La fonction de saisie `fscanf`

La fonction **`fscanf`**, analogue à **`scanf`**, permet de lire des données dans un fichier. Son prototype est semblable à celle de **`scanf`** :

```
int fscanf(FILE * stream, const char * format, ...)
```

Voici un exemple concret d'utilisation, en supposant que j'ai sous la main un fichier nommé **`scores.txt`** contenant :

```
ali 100
brahim 96
bouchra 95
hamid 42
hind 37
```

Le programme suivant permet de récupérer les noms des joueurs ainsi que leurs points :

```c
#include <stdio.h>

main(){
    FILE * pF;
    char nom[5][50];
    int score[5];
    int i;
    pF = fopen("scores.txt", "r");
    if(pF == NULL){
        printf("Impossible d'ouvrir le fichier de scores\n");
        exit(1);
    }
    for(i = 0; i < 5; i++){
        fscanf(pF, "%s %d", nom[i], &score[i]);
        printf("Le joueur %s a %d points\n", nom[i], score[i]);
    }
    fclose(pF);
    pF = NULL;
}
```

Le programme affichera :

```
Le joueur ali a 100 points
Le joueur brahim a 96 points
Le joueur bouchra a 95 points
Le joueur hamid a 42 points
Le joueur hind a 37 points
```

4.4 Impression et lecture de caractères

Les fonctions **fgetc** et **fputc** permettent respectivement de lire et d'écrire un caractère dans un fichier. La fonction **fgetc**, de type **int**, retourne le caractère lu dans le fichier. Elle retourne la constante **EOF** lorsqu'elle détecte la fin du fichier. Son prototype est :

```c
int fgetc(FILE * flot)
```

La fonction **fputc** écrit un et seulement un seul caractère dans fichier. Elle retourne l'entier correspondant au caractère lu (ou la constante **EOF** en cas d'erreur). Son prototype est :

```c
int fputc(int caractere, FILE * flot)
```

Il existe également deux versions optimisées des fonctions **fgetc** et **fputc** qui sont implémentées par des macros. Il s'agit respectivement de **getc** et **putc**. Leur prototype est similaire à celle de **fgetc** et **fputc** :

```c
int getc(FILE * flot);
int putc(int caractere, FILE * flot)
```

Ainsi, le programme suivant lit le contenu du fichier texte **entree.txt**, et le recopie caractère par caractère dans le fichier **sortie.txt** :

```c
#include <stdio.h>
#include <stdlib.h>
#define ENTREE "entree.txt"
#define SORTIE "sortie.txt"
main(){
    FILE *f_in, *f_out;
    int c;
    if((f_in = fopen(ENTREE,"r")) == NULL){
        fprintf(stderr, "\nImpossible de lire le fichier\n");
        return(1);
    }
    if ((f_out = fopen(SORTIE,"w")) == NULL){
        fprintf(stderr, "\nImpossible d'ecrire dans le fichier\n");
        return(1);
    }
    while ((c = fgetc(f_in)) != EOF)
        fputc(c, f_out);
    fclose(f_in);
    fclose(f_out);
}
```

4.5 Relecture d'un caractère

Il est possible de replacer un caractère dans un flot au moyen de la fonction **ungetc** :

```c
int ungetc(int caractere, FILE *flot);
```

Cette fonction place le caractère **caractere** (converti en **unsigned char**) dans le flot **flot**. En particulier, si **caractere** est égal au dernier caractère lu dans le flot, elle annule le déplacement provoqué par la lecture précédente. Toutefois, **ungetc** peut être utilisée avec n'importe quel caractère (sauf **EOF**). Par exemple, l'exécution du programme suivant :

```c
#include <stdio.h>
#include <stdlib.h>
#define ENTREE "entree.txt"

main(){
    FILE *f_in;
    int c;
    if((f_in = fopen(ENTREE,"r")) == NULL){
        fprintf(stderr, "\nImpossible de lire le fichier\n");
        return(1);
    }
    while((c = fgetc(f_in)) != EOF){
        if (c == '0')
            ungetc('.',f_in);
        putchar(c);
    }
    fclose(f_in);
}
```

sur le fichier **entree.txt** dont le contenu est **097023** affiche à l'écran **0.970.23**

4.6 Les entrées-sorties binaires

Les fonctions d'entrées-sorties binaires permettent de transférer des données dans un fichier sans transcodage. Elles sont donc plus efficaces que les fonctions d'entrée-sortie standard, mais les fichiers produits ne sont pas portables puisque le codage des données dépend des machines.

Elles sont notamment utiles pour manipuler des données de grande taille ou ayant un type composé. Leurs prototypes sont :

```
size_t fread(void *pointeur, size_t taille, size_t nombre, FILE *flot);
size_t fwrite(void *pointeur, size_t taille, size_t nombre, FILE *flot);
```

où **pointeur** est l'adresse du début des données à transférer, **taille** la taille des objets à transférer, **nombre** leur nombre. Rappelons que le type **size_t**, défini dans **stddef.h**, correspond au type du résultat de l'évaluation de **sizeof**. Il s'agit du plus grand type entier non signé.

La fonction **fread** lit les données sur le flot **flot** et la fonction **fwrite** les écrit. Elles retournent toutes deux le nombre de données transférées.

Par exemple, le programme suivant écrit un tableau d'entiers (contenant les 50 premiers entiers) avec **fwrite** dans le fichier **sortie.txt**, puis lit ce fichier avec **fread** et imprime les éléments du tableau.

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

#define NB 50
#define F_SORTIE "sortie"

main(){
    FILE *f_in, *f_out;
    int *tab1, *tab2;
    int i;
    tab1 = (int*)malloc(NB * sizeof(int));
    tab2 = (int*)malloc(NB * sizeof(int));
    for (i = 0 ; i < NB; i++)
        tab1[i] = i;

    /* ecriture du tableau dans F_SORTIE */
    if ((f_out = fopen(F_SORTIE, "wb")) == NULL){
        fprintf(stderr, "\nImpossible d'ecrire dans le fichier\n");
        return(1);
    }
    fwrite(tab1, NB * sizeof(int), 1, f_out);
    fclose(f_out);

    /* lecture dans F_SORTIE */
    if ((f_in = fopen(F_SORTIE, "rb")) == NULL){
        fprintf(stderr, "\nImpossible de lire dans le fichier\n");
        return(1);
    }
    fread(tab2, NB * sizeof(int), 1, f_in);
    fclose(f_in);
```

```
    for (i = 0 ; i < NB; i++)
        printf("%d ",tab2[i]);
    printf("\n");
    free(tab1);
    free(tab2);
    getch();
}
```

Les éléments du tableau sont bien affichés à l'écran. Par contre, on constate que le contenu du fichier **sortie** n'est pas encodé.

4.7 Positionnement dans un fichier

Les différentes fonctions d'entrées-sorties permettent d'accéder à un fichier en mode séquentiel : les données du fichier sont lues ou écrites les unes à la suite des autres. Il est également possible d'accéder à un fichier en mode direct, c'est-à-dire que l'on peut se positionner à n'importe quel endroit du fichier. La fonction **fseek** permet de se positionner à un endroit précis ; elle a pour prototype :

```
int fseek(FILE *flot, long deplacement, int origine);
```

La variable **deplacement** détermine la nouvelle position dans le fichier. Il s'agit d'un déplacement relatif par rapport à l'origine ; il est compté en nombre d'octets. La variable origine peut prendre trois valeurs :

- **SEEK_SET** (égale à **0**) : début du fichier ;
- **SEEK_CUR** (égale à **1**) : position courante ;
- **SEEK_END** (égale à **2**) : fin du fichier.

La fonction :

```
int rewind(FILE *flot);
```

permet de se positionner au début du fichier. Elle est équivalente à :

```
fseek(flot, 0, SEEK_SET);
```

La fonction :

```
long ftell(FILE *flot);
```

retourne la position courante dans le fichier (en nombre d'octets depuis l'origine).

Exemple :

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

#define NB 50
#define F_SORTIE "sortie"
```

```c
main(){
    FILE *f_in, *f_out;
    int *tab;
    int i;

    tab = (int*)malloc(NB * sizeof(int));
    for (i = 0 ; i < NB; i++)
        tab[i] = i;

    /* ecriture du tableau dans F_SORTIE */
    if ((f_out = fopen(F_SORTIE, "wb")) == NULL){
        fprintf(stderr, "\nImpossible d'ecrire dans le fichier\n");
        exit(1);
    }
    fwrite(tab, NB * sizeof(int), 1, f_out);
    fclose(f_out);

    /* lecture dans F_SORTIE */
    if ((f_in = fopen(F_SORTIE, "rb")) == NULL){
        fprintf(stderr, "\nImpossible de lire dans le fichier\n");
        exit(1);
    }

    /* on se positionne a la fin du fichier */
    fseek(f_in, 0, SEEK_END);
    printf("\n Position %ld", ftell(f_in));

    /* deplacement de 10 int en arriere */
    fseek(f_in, -10 * sizeof(int), SEEK_END);
    printf("\n Position %ld", ftell(f_in));
    fread(&i, sizeof(i), 1, f_in);
    printf("\t i = %d", i);

    /* retour au debut du fichier */
    rewind(f_in);
    printf("\n Position %ld", ftell(f_in));
    fread(&i, sizeof(i), 1, f_in);
    printf("\t i = %d", i);

    /* deplacement de 5 int en avant */
    fseek(f_in, 5 * sizeof(int), SEEK_CUR);
    printf("\n Position %ld", ftell(f_in));
    fread(&i, sizeof(i), 1, f_in);
    printf("\t i = %d\n", i);
    fclose(f_in);
    getch();
}
```

L'exécution de ce programme affiche à l'écran :

```
 Position 200
 Position 160    i = 40
 Position 0      i = 0
 Position 24     i = 6
```

On constate en particulier que l'emploi de la fonction **fread** provoque un déplacement correspondant à la taille de l'objet lu à partir de la position courante.

Partie II : Implémentation des algorithmes de trie et de recherche en C

1 Algorithmes de tries et de recherches

1.1 Définition d'un algorithme de Tri

Les tableaux permettent de stocker plusieurs éléments de même type au sein d'une seule entité. Lorsque le type de ces éléments possède un ordre total, on peut donc les ranger en ordre croissant ou décroissant. Trier un tableau c'est donc ranger les éléments d'un tableau en ordre croissant ou décroissant. Dans ce cours on ne fera que des tris en ordre croissant.

Il existe plusieurs méthodes de tri qui se différencient par leur complexité d'exécution et leur complexité de compréhension pour le programmeur.

1.2 La fonction d'échange

Tous les algorithmes de tri utilisent une fonction qui permet d'échanger (permuter) la valeur de deux variables. Dans le cas où les variables sont entières, la fonction reçoit en entrée un tableau et les deux indices des valeurs du tableau qui vont être échangées.

```
void echanger(int tab [], int i, int j){
   int temp;
   temp = tab [i];
   tab [i] = tab [j];
   tab [j] = temp;
}
```

1.3 Implémentation des algorithmes de tri en C

Analysons quelques algorithmes de tri et leur implémentation en C.

1.3.1 Le tri à bulle

Cet algorithme consiste à comparer les différentes valeurs adjacentes d'un tableau. Les éléments 0 et 1 sont comparés. Si le premier est supérieur au second, on effectue une permutation. Ensuite on fait de même avec l'élément 1 et 2, 2 et 3, etc. Dans cette étape, il y a n-1 tests et on est sûr que le plus grand élément se trouve à la fin du tableau. Il reste à traiter les n-1 premiers éléments. L'algorithme se termine lorsqu'il n'y a plus de permutations possibles.

Le pire des cas est lorsque le tableau est classé par ordre décroissant.

```
void triBulle(int tab[], int longueur){
   int i;
   int permutation;
   do{
      permutation = 0;
      for(i=0; i<longueur-1; i++){
         if(tab[i]>tab[i+1]){
            echanger(tab, i, i+1);
            permutation = 1;
         }
```

```
        }
        longueur--;
    }
    while(permutation);
}
```

Exemple :

Les différentes étapes de l'algorithme :

7	3	0	1	9	6
3	7	0	1	9	6
3	0	7	1	9	6
3	0	1	7	9	6
3	0	1	7	9	6
3	0	1	7	6	9
0	3	1	7	6	9
0	1	3	7	6	9
0	1	3	7	6	9
0	1	3	6	7	9

1.3.2 Le tri par sélection

Le principe est de rechercher la valeur maximale dans un tableau de taille **n** et de l'échanger avec le dernier élément du tableau. Il faut ensuite faire la même chose avec les **n-1** premiers termes du tableau puis les **n-2** termes, etc... jusqu'à ce qu'il ne reste plus qu'une seule valeur.

```
void triSelection(int tab[], int longueur){
    int i, maximum;
    while(longueur>0){
        // recherche de la plus grande valeur du tableau
        maximum = 0;
        for(i=1; i<longueur; i++)
            if(tab[i]>tab[maximum])
                maximum = i;
        // echange du maximum avec le dernier élément
        echanger(tab, maximum, longueur-1);
        // on traite le reste du tableau
        longueur--;
    }
}
```

Exemple :

Les différentes étapes de l'algorithme :

7	3	0	1	9	6
7	3	0	1	6	9
6	3	0	1	7	9
1	3	0	6	7	9
1	0	3	6	7	9

0	1	3	6	7	9

1.3.3 Le tri par insertion

Le but de cet algorithme est de prendre les éléments un par un en commençant en début de tableau et de les classer par rapport aux éléments se trouvant à leur gauche. L'élément **0** est par définition classé. L'élément **1** est comparé avec l'élément **0**, s'il est inférieur alors on échange les deux éléments. Ensuite on passe à l'élément **2** : il est d'abord comparé à l'élément **1**. S'il lui est inférieur alors on procède à l'échange, on compare l'élément **1** à l'élément **0** (rappel : l'élément **1** et **2** ont été échangés), on fait l'échange si l'élément **1** est inférieur à l'élément **0** et on passe à l'élément **3**. Sinon on passe directement à l'élément **3** etc. Le problème de cet algorithme est que l'on peut être amené à décaler le dernier élément du tableau en première position, ce qui implique beaucoup d'opérations.

```
void triInsertion(int tab[], int longueur){
    int i;
    int memoire; // memoire:valeur en cours de traitement
    int compteur; // indique la partie du tableau à traiter
    int marqueur; // faut-il continuer les comparaisons?

    for(i=1; i<longueur; i++){
        memoire = tab[i];
        compteur = i-1;
        do{
            marqueur = 0;
            // comparaisons et décalages vers la droite si nécessaire
            if(tab[compteur] > memoire){
                echanger(tab, compteur+1, compteur);
                compteur--;
                marqueur = 1;
            }
            if(compteur < 0) // on évite de dépasser les limites
                marqueur = 0;
        }while(marqueur);
    }
    tab[compteur+1] = memoire;
    // on a classé le tableau jusqu'à l'indice i
}
```

Exemple :

Les grandes étapes de l'algorithme :

7	3	0	1	9	6
3	7	0	1	9	6
0	3	7	1	9	6
0	1	3	7	9	6
0	1	3	7	9	6
0	1	3	7	9	9
0	1	3	7	7	9
0	1	3	6	7	9

Les 3 dernières lignes de l'exemple montrent le détail de l'algorithme. La valeur du dernier élément : 6 étant stockée en mémoire.

1.3.4 Le tri rapide

Le principe de cet algorithme est de diviser un ensemble d'éléments en deux parties. Pour faire cette séparation, une valeur pivot est choisie. Les valeurs sont séparées par le pivot suivant qu'elles lui soient supérieures ou inférieures. Ensuite, on fait la même chose pour les deux sous ensembles. Cet algorithme est donc récursif. Le choix du pivot est un problème majeur, le mieux serait de pouvoir séparer les ensembles en deux parties égales. Toutefois une recherche s'avérerait trop coûteuse. C'est pour cela qu'on choisit le premier élément ou le dernier élément.

```
int partition(int tableau[], const int debut, const int fin){
   int compteur = debut;
   int pivot = tableau[debut];
   int i;

   for(i=debut+1; i<=fin; i++){
      if(tableau[i]<pivot) { // si élément inférieur au pivot
         // incrémente compteur cad la place finale du pivot
         compteur++;
         echanger(tableau, compteur, i); // élément positionné
      }
      echanger(tableau, compteur, debut); // le pivot est placé
      return compteur; // et sa position est retournée
}

void triRapideAux(int tableau[], const int debut, const int fin){
   if(debut<fin) { // cas d'arrêt pour la récursivité
      // division du tableau
      int pivot = partition(tableau, debut, fin);
      triRapideAux(tableau, debut, pivot-1); // trie partie1
      triRapideAux(tableau, pivot+1, fin); // trie partie2
   }
}

void triRapide(int tableau[], const int longueur)
{
   triRapideAux(tableau, 0, longueur-1);
}
```

Exemple :

Les grandes étapes de l'algorithme :

53	31	10	87	13	59	62	26	47	38
38	31	10	13	26	47	53	87	59	62
26	31	10	13	38	47	53	87	59	62
13	10	26	31	38	47	53	87	59	62
10	13	26	31	38	47	53	87	59	62
10	13	26	31	38	47	53	87	59	62
10	13	26	31	38	47	53	62	59	87
10	13	26	31	38	47	53	59	62	87

1.3.6 Le tri fusion

Le principe de cet algorithme est de diviser le tableau en sous tableaux de les traiter et ensuite de les fusionner. Cet algorithme est récursif. On divise le tableau en deux sous tableaux qui sont eux mêmes sont divisés en deux sous tableaux, etc. La condition d'arrêt est lorsque le tableau ne comporte plus qu'un seul élément. L'algorithme contient plusieurs parties : la division du tableau en deux, le tri des deux tableaux et la fusion des deux tableaux.

```
void fusion(int tableau[], const int debut1,
                           const int fin1, const int fin2){
   int *tableau2;
   int debut2 = fin1+1;
   int compteur1 = debut1;
   int compteur2 = debut2;
   int i;

   tableau2 = (int*)malloc((fin1-debut1+1)*sizeof(int));

   // copie des éléments du début de tableau
   for(i=debut1; i<=fin1; i++)
      tableau2[i-debut1] = tableau[i];

   // fusion des deux tableaux
   for(i=debut1; i<=fin2; i++){
      if(compteur1==debut2) // éléments du 1er tableau tous utilisés
         break; // éléments tous classés
      else
         // élts du 2nd tableau tous utilisés
         if(compteur2==(fin2+1)) {
            // copie en fin de tableau des élts du 1er sous tableau
            tableau[i] = tableau2[compteur1-debut1];
            compteur1++;
         }
         else
            if(tableau2[compteur1-debut1]<tableau[compteur2]) {
               // ajout d'1 élément du 1er sous tableau
               tableau[i] = tableau2[compteur1-debut1];
               compteur1++; // on avance ds le 1er sous tableau
            }
            else { // copie de l'élément à la suite du tableau
               tableau[i] = tableau[compteur2];
               compteur2++; // on avance ds le 2nd sous tableau
            }
   }
   free(tableau2);
}

void triFusionAux(int tableau[], const int debut, const int fin){
   if(debut!=fin) {// condition d'arrêt
      int milieu = (debut+fin)/2;
      triFusionAux(tableau, debut, milieu); // trie partie1
      triFusionAux(tableau, milieu+1, fin); // trie partie2
      fusion(tableau, debut, milieu, fin);  // fusion des 2 parties
   }
}

void triFusion(int tableau[], const int longueur){
   if(longueur>0)
```

```
      triFusionAux(tableau, 0, longueur-1);
}
```

Exemple :

Les grandes étapes de l'algorithme :

53	31	10	87	13	59	62	26	47	38
53	31	10	87	13	59	62	26	47	38
53	31	10	87	13	59	62	26	47	38
53	31	10	87	13	59	62	26	47	38
31	53	10	87	13	59	62	26	47	38
10	31	53	87	13	59	62	26	47	38
10	31	53	13	87	59	62	26	47	38
10	13	31	53	87	59	62	26	47	38
etc...									
10	13	31	53	87	26	38	47	59	62
etc...									
10	13	26	31	38	47	53	59	62	87

1.3.7 Le tri Shell

C'est une variante du tri par insertion. Dans ce tri, les éléments sont décalés de plusieurs éléments. La distance qui les sépare est appelée "pas". A chaque étape le tableau est affiné (mieux organisé) et le pas réduit. Lorsque le pas est de 1, cela revient à un tri par insertion.

Le pas est généralement calculé à partir de la formule suivante :

$$u(n+1) = (3*u(n)+1) \quad \text{avec } u(0) = 1$$

```
void triShell(int tableau[], const int longueur){
   int pas, i, j, memoire;
   pas = 0;
   // Calcul du pas
   while(pas<longueur){
      pas = 3*pas+1;
   }
   while(n!=0) { // tant que le pas est > 0
      pas = pas/3;
      for(i=n; i<longueur; i++){
         memoire = tableau[i]; // valeur à décaler éventuellement
         j = i;
         while((j>(pas-1)) && (tableau[j-pas]>memoire)) {
            // échange des valeurs
            tableau[j] = tableau[j-pas];
            j = j-pas;
         }
         tableau[j] = memoire;
      }
   }
}
```

Exemple :

Calcul du pas pour un tableau de taille n = 100 :

u(0) = 0

u(1) = 3 * u(0) + 1 = 3 * 0 + 1 = 1

u(2) = 3 * u(1) + 1 = 3 * 1 + 1 = 4

u(3) = 3 * u(2) + 1 = 3 * 4 + 1 = 13

u(4) = 3 * u(3) + 1 = 3 * 13 + 1 = 40

u(5) = 3 * u(4) + 1 = 3 * 40 + 1 = 121

Les valeurs successives que la pas prendra seront donc :

pas1 = 121 / 3 = 40

pas2 = pas1 / 3 = 40 / 3 = 13

pas3 = pas2 / 3 = 13 / 3 = 4

pas4 = pas3 / 3 = 4 / 3 = 1

Les grandes étapes de l'algorithme :

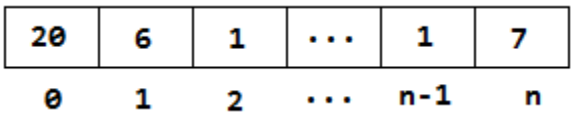
53	31	10	87	13	59	62	26	47	38
13	31	10	87	53	59	62	26	47	38
13	31	10	87	53	59	62	26	47	38
13	31	10	87	53	59	62	26	47	38
13	31	10	26	53	59	62	87	47	38
13	31	10	26	47	59	62	87	53	38
13	31	10	26	47	38	62	87	53	59
maintenant le pas = 1 donc on passe à un tri par insertion									
13	31	10	26	47	38	62	87	53	59
13	31	10	26	47	38	62	87	53	59
10	13	31	26	47	38	62	87	53	59
10	13	26	31	47	38	62	87	53	59
10	13	26	31	47	38	62	87	53	59
10	13	26	31	38	47	62	87	53	59
10	13	26	31	38	47	62	87	53	59
10	13	26	31	38	47	62	87	53	59
10	13	26	31	38	47	53	62	87	59
10	13	26	31	38	47	53	59	62	87

Partie III : Implémentation des Types de Données Abstraits en C

1 Les listes chainées (*Linked List*)

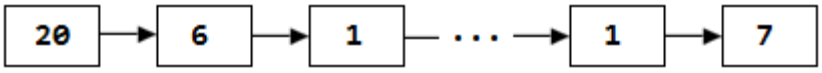
1.1 Généralités

Lorsque vous créez un algorithme utilisant des conteneurs, il existe différentes manières de les implémenter, la façon la plus courante étant les tableaux. Dans un tableau, les éléments sont placés de façon contiguë en mémoire.



Pour pouvoir le créer, il vous faut connaître sa taille. Si vous voulez supprimer un élément au milieu du tableau, il vous faut recopier les éléments temporairement, réallouer de la mémoire pour le tableau, puis le remplir à partir de l'élément supprimé. En bref, beaucoup de manipulations coûteuses en ressources.

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 adresses (pointeurs). Vous pouvez ajouter et enlever des éléments d'une liste chaînée à n'importe quel endroit, à n'importe quel instant, sans devoir recréer la liste entière.



Une liste chaînée donc est un ensemble n_i d'éléments notée L = e_1, $e_{2, ..., }$ e_n ou e_1 est le premier élément, e_2 le deuxième, etc... Lorsque n=0 on dit que la liste est vide.

Les listes servent à gérer un ensemble de données, un peu comme les tableaux. Elles sont cependant plus efficaces pour réaliser des opérations comme l'insertion et la suppression d'éléments. Elles utilisent par ailleurs l'allocation dynamique de mémoire et peuvent avoir une taille qui varie pendant l'exécution.

Remarque : Un tableau peut aussi être défini dynamiquement mais pour modifier sa taille, il faut en créer un nouveau, transférer les données puis supprimer l'ancien.). L'allocation (ou la libération) se fait élément par élément.

Les opérations sur une liste peuvent être:

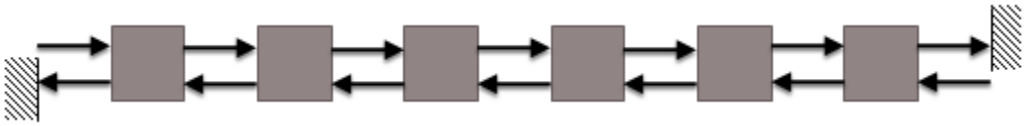
- Créer une liste
- Supprimer une liste
- Rechercher un élément particulier
- Insérer un élément (en début, en n ou au milieu)
- Supprimer un élément particulier
- Permuter deux éléments
- Concaténer deux listes
- ...

1.2 Types de liste chainée

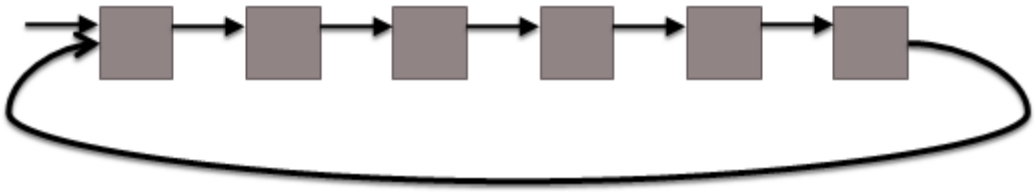
Les listes peuvent par ailleurs être :

- simplement chaînées,

- doublement chaînées



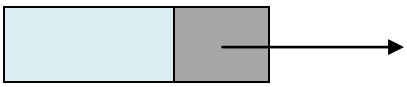
- circulaires (chaînage simple ou double).



1.3.1 Listes simplement chaînées

Une liste simplement chaînée est composée d'éléments distincts liés par un simple pointeur. Chaque élément (**Noeud**) d'une liste simplement chaînée est formé de deux parties:

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



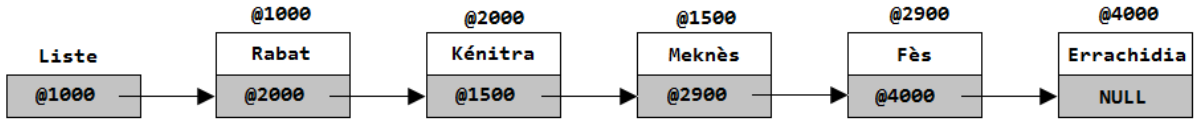
Données Pointeur

(Information) (Suivant)

On implémente un élément d'une liste simple contenant une donnée de type **caractère** (`char`) sous forme d'une structure **C**:

```c
typedef char Type;

typedef struct Noeud * Liste;

typedef struct Noeud{
    Type info;
    Liste suivant;
}Noeud;
```

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



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

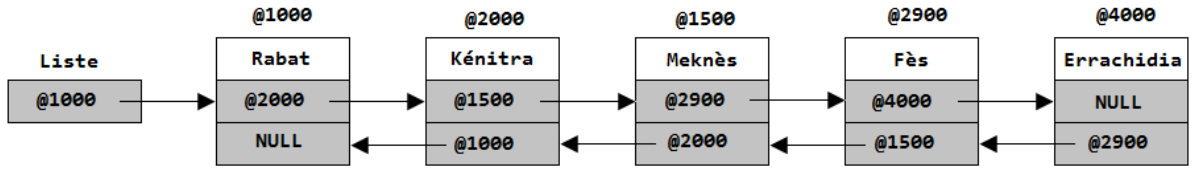
En pratique, les éléments étant créent par allocation dynamique, ne sont pas contigus en mémoire contrairement à un tableau. La suppression d'un élément sans précaution ne permet plus d'accéder aux éléments suivants. D'autre part, une liste simplement chaînée ne peut être parcourue que dans un sens (de la tête vers la queue).

1.3.2 Listes doublement chaînées

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

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

Elles peuvent donc être parcourues dans les deux sens.

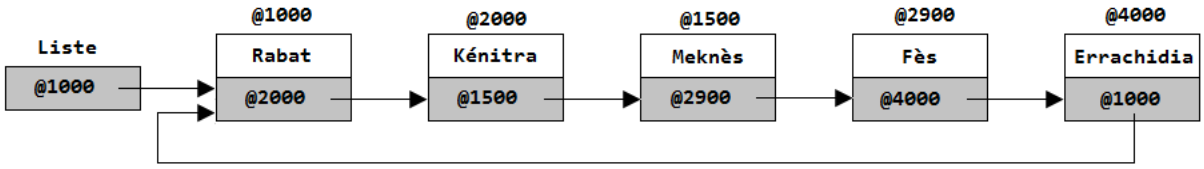


En **C**, on implémente un élément d'une liste doublement chainée contenant une donnée entière sous forme d'une structure :

```c
typedef char Type;

typedef struct Noeud * Liste;

typedef struct Noeud {
    Type info;
    Liste suivant;
    Liste precedent;
}Noeud;

typedef struct NoeudD * ListeD;

typedef struct NoeudD{
    int taille;
    Liste tete;
    Liste queue;
}NoeudD;
```

1.3.3 Listes circulaires

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



1.4 Opérations sur une liste simplement chainée

On décrira dans ce paragraphe quelques opérations sur les listes simplement chaînées. La liste chaînée de caractères utilisée sera vide au départ. L'exemple suivant défini le type abstrait **`Noeud`** permettant de créer le type **`Liste`** représentant une liste chaînée de **caractères** (**`char`**).

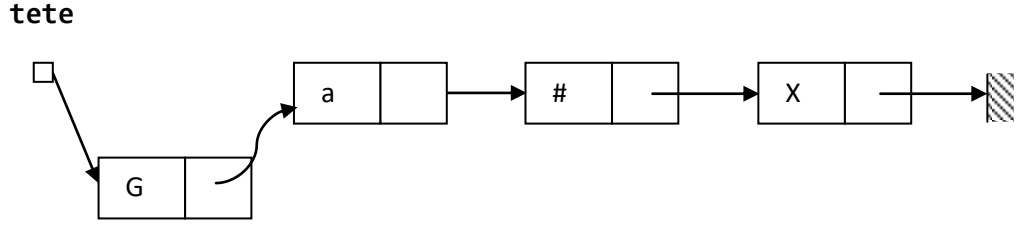
```
typedef char Type;

typedef struct Noeud * Liste;

typedef struct Noeud{
    Type info;
    Liste suivant;
}Noeud;
```

1.4.1 Insertion d'un élément

Pour insérer un élément dans une liste chaînée, il faut savoir où l'insérer. Les trois insertions possibles dans une liste chaînée sont les insertions en tête de liste, les insertions en fin de liste, et les insertions à n'importe où (à une position fixé).

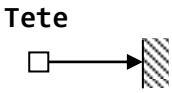
1.4.1.1 Insertion de nœuds en tête de la liste chaînée

L'insertion en tête, se fait en créant un élément, lui assigner la valeur que l'on veut insérer, puis pour terminer, raccorder cet élément à la liste. Lors d'une insertion en tête, on devra donc assigner au **`suivant`** l'adresse du premier élément de la liste. Visualisons tout ceci sur un schéma :



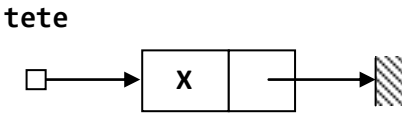
Illustrons d'un chaînage en tête : au départ, la liste chaînée est vide, et on suppose que l'on saisit, dans l'ordre, les caractères 'X', '#' et 'a'.

- Initialisation de la liste à **NULL** :

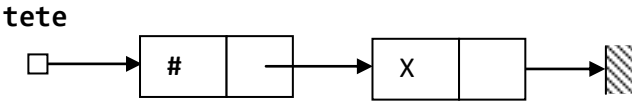



```
// ...
main(){
    Liste tete;
    tete = NULL;
}
```

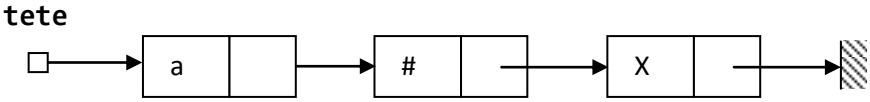
- Ajout de 'X' :



- Ajout de '#' :



- Ajout de 'a' :



Le programme suivant implémente les différentes étapes illustrées :

```
// ...
main(){
    // ...
    // Création du noeud contenant 'X'
    Noeud * nouveau = (Noeud*)malloc(sizeof(Noeud));
    nouveau->info = 'X';
    nouveau->suivant = tete;
    tete = nouveau;
    // Création du noeud contenant '#'
    nouveau = (Noeud*)malloc(sizeof(Noeud));
    nouveau->info = '#';
    nouveau->suivant = tete;
    tete = nouveau;
    // Création du noeud contenant 'a'
    nouveau = (Noeud*)malloc(sizeof(Noeud));
    nouveau->info = 'a';
    nouveau->suivant = tete;
    tete = nouveau;
}
```

Pour afficher le contenu de la liste, on doit parcourir la liste avec un pointeur, pour ne pas perdre la tête de la liste, jusqu'au bout et afficher toutes les valeurs qu'elle contient :

```
// ...
main(){
    // ...
    Liste tmp = tete;
    while(tmp!=NULL){
        printf("%c ", tmp->info);
        tmp = tmp->suivant;
    }
}
```

A ce niveau, on a tout intérêt d'améliorer notre programme en créant des fonctions réalisant les opérations sur la liste simplement chainée crée. Une fonction d'initialisation de la liste « **initialiser** », une fonction d'insertion en tête « **insererEnTete** » et une fonction d'affichage « **afficher** ».

Le programme sera ainsi :

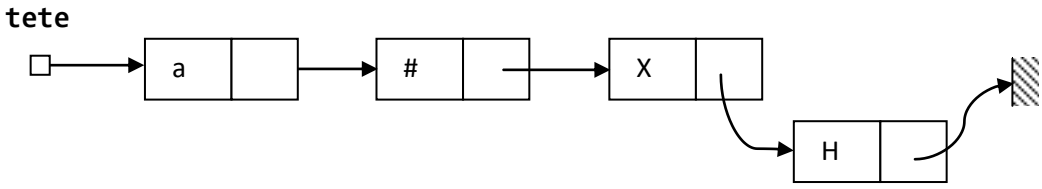
```
#include <stdio.h>
#include <stdlib.h>

typedef char Type;

typedef struct Noeud * Liste;

typedef struct Noeud{
    Type info;
    Liste suivant;
}Noeud;
void initialiser(Liste * adrListe){
    (*adrListe) = NULL ;
}
void insererEnTete(Liste * adrListe, Type valElement){
    Noeud * nouveau ;
    nouveau = (Noeud*)malloc(sizeof(Noeud));
    if(nouveau != NULL ){
        nouveau->suivant = (*adrListe);
        nouveau->info = valElement;
        (*adrListe) = nouveau;
    }
}
void afficher(Liste liste){
    while(liste!=NULL){
        printf("%c ", liste->info);
        liste = liste->suivant;
    }
    printf("\n");
}
main(){
    Liste tete;
    initialiser(&tete);
    insererEnTete(&tete, 'X');
    insererEnTete(&tete, '#');
    insererEnTete(&tete, 'a');
    afficher(tete);
}
```

Le programme affichera :

```
a # X
```

1.4.1.2 Insertion de nœuds en fin de la liste chaînée

L'insertion en fin de liste est un peu plus compliquée. Il 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 liste originale sur le nouvel élément que nous venons de créer. Pour ce faire, il faut créer un pointeur temporaire sur l'élément 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. Visualisons tout ceci sur un schéma :



Le code **C** de la fonction d'insertion en fin est le suivant :

```c
void insererEnQueue(Liste * adrListe, Type valElement){
    // On crée un nouvel élément
    Noeud * nouveau = (Noeud*)malloc(sizeof(Noeud));
    // On assigne la valeur au nouvel élément
    nouveau->info = valElement;
    // On ajoute en fin, donc aucun élément ne va suivre
    nouveau->suivant = NULL;
    if((*adrListe) == NULL){
        // Si la liste est videé il suffit d'assigner
        // au liste de la liste l'élément créé
        (*adrListe) = nouveau;
    }
    else{
        // 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
        Liste tmp = (*adrListe);
        while(tmp->suivant != NULL){
            tmp = tmp->suivant;
        }
        tmp->suivant = nouveau;
    }
}
```

1.4.2 Suppression d'un élément

1.4.2.1 Supprimer un élément en tête

Pour supprimer le premier élément de la liste, il faut 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 la récupérer.

```
void supprimerEnTete(Liste * adrListe){
    if((*adrListe) != NULL){
        // Si la liste est non vide, on se prépare
        // à renvoyer l'adresse de l'élément en 2ème position
        Liste tmp = (*adrListe)->suivant;
        // On libère le premier élément
        free(*adrListe);
        // On retourne le nouveau début de la liste
        (*adrListe) = tmp;
    }
}
```

1.4.2.2 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.

```
void supprimerEnFin(Liste * adrListe){
    // Si la liste est non vide, on on peut supprimer
    if((*adrListe) != NULL){
        // Si la liste contient un seul élément
        if((*adrListe)->suivant == NULL){
            // On le libère et on retourne NULL
            // (la liste est maintenant vide)
            free(*adrListe);
            (*adrListe) = NULL;
        }
        // Si la liste contient au moins deux éléments
        Liste tmp = (*adrListe);
        Liste ptmp = (*adrListe);
        // Tant qu'on n'est pas au dernier élément
        while(tmp->suivant != NULL){
            // ptmp stock l'adresse de tmp
            ptmp = tmp;
            // On déplace tmp (mais ptmp garde l'ancienne
            // valeur de tmp
            tmp = tmp->suivant;
        }
        // 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
        ptmp->suivant = NULL;
        free(tmp);
    }
}
```

1.4.3 Rechercher un élément dans une liste

Le but cette fois est de renvoyer l'adresse du premier élément trouvé ayant une certaine valeur. Si aucun élément n'est trouvé, on renverra **`NULL`**. L'intérêt est de pouvoir, une fois le

premier élément trouvé, chercher la prochaine occurrence en recherchant à partir de **`elementTrouve->suivant`**. On parcourt donc la liste jusqu'au bout, et dès qu'on trouve un élément qui correspond à ce que l'on recherche, on renvoie son adresse.

```
Noeud * rechercherElement(Liste liste, Type valElement){
    Noeud * tmp = liste;
    // Tant que l'on n'est pas au bout de la liste
    while(tmp != NULL){
        if(tmp->info == valElement){
            // Si l'élément a la valeur recherchée,
            // on renvoie son adresse
            return tmp;
        }
        tmp = tmp->suivant;
    }
    return NULL;
}
```

1.4.4 Compter le nombre d'occurrences d'une valeur

Pour ce faire, nous allons utiliser la fonction précédente permettant de rechercher un élément. On cherche une première occurrence : si on la trouve, alors on continue la recherche à partir de l'élément suivant, et ce tant qu'il reste des occurrences de la valeur recherchée. Il est aussi possible d'écrire cette fonction sans utiliser la précédente bien entendu, en parcourant l'ensemble de la liste avec un compteur que l'on incrémente à chaque fois que l'on passe sur un élément ayant la valeur recherchée. Cette fonction n'est pas beaucoup plus compliquée, mais il est intéressant d'un point de vue algorithmique de réutiliser des fonctions pour simplifier nos codes.

```
int nombreOccurences(Liste liste, Type valElement){
    int i = 0;
    // Si la liste est vide, on renvoie 0
    if(liste == NULL)
        return 0;
    // Sinon, tant qu'il y a encore un élément ayant la val = valeur
    while((liste = rechercherElement(liste, valElement)) != NULL){
        // On incrémente
        liste = liste->suivant;
        i++;
    }
    // Et on retourne le nombre d'occurrences
    return i;
}
```

1.4.5 Compter le nombre d'éléments d'une liste chaîné

C'est un algorithme vraiment simple. On parcoure la liste de bout en bout et incrémentez un compteur pour chaque nouvel élément trouvé. Jusqu'à maintenant, nous n'avons utilisé que des algorithmes itératifs qui consistent à boucler tant que l'on n'est pas au bout. Cette fois-ci, on va créer un algorithme récursif.

```
int nombreElements(Liste liste){
    // Si la liste est vide, il y a 0 élément
    if(liste == NULL)
        return 0;
    // Sinon, il y a un élément (celui que l'on est en train de traiter)
```

```
    // plus le nombre d'éléments contenus dans le reste de la liste
    return nombreElements(liste->suivant)+1;
}
```

1.4.1.7 Recherche du k-ème élément

Pour le coup, c'est une fonction relativement simple. Il suffit de se déplacer **k** fois à l'aide du pointeur **tmp** le long de la liste chaînée et de renvoyer l'élément à l'indice **k**. Si la liste contient moins de **i** élément(s), alors nous renverrons **NULL**.

```
Noeud * kiemeNoeud(Liste liste, int k){
    int i;
    // On se déplace de k cases, tant que c'est possible
    for(i=0; i<k && liste != NULL; i++){
        liste = liste->suivant;
    }
    // Si l'élément est NULL, c'est que la liste contient
    // moins de i éléments
    if(liste == NULL){
        return NULL;
    }
    else{
        // Sinon on renvoie l'adresse de l'élément i
        return liste;
    }
}
```

1.4.7 Effacer tous les éléments ayant une certaine valeur

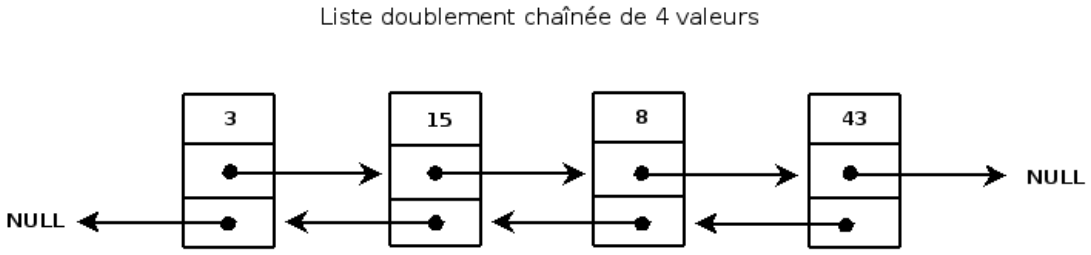
Dans cette dernière fonction, nous allons encore une fois utiliser un algorithme récursif.

```
void supprimerElement(Liste * adrListe, Type valElement){
    // On supprime si la liste est non vide
    if((*adrListe) != NULL){
        // Si l'élément en cours de traitement doit être supprimé
        if((*adrListe)->info == valElement){
            // On le supprime en prenant soin de mémoriser
            // l'adresse de l'élément suivant
            Noeud * tmp = (*adrListe)->suivant;
            free(*adrListe);
            (*adrListe) = tmp;
            // L'élément ayant été supprimé, la liste commencera
            // à l'élément suivant pointant sur une liste qui
            // ne contient plus aucun élément ayant la valeur recherchée
            supprimerElement(&tmp, valElement);
        }
        else{
            // Si l'élement en cours de traitement ne doit pas être
            // supprimé, alors la liste finale commencera par cet élément
            // et suivra une liste ne contenant plus d'élément ayant
            // la valeur recherchée
            supprimerElement(&((*adrListe)->suivant), valElement);
        }
    }
}
```

1.5 Opération sur les listes doublement chainées

Rappelons que lorsque chaque élément d'une liste chainée pointe vers l'élément suivant, nous parlons de liste simplement chainée et lorsque chaque élément d'une liste pointe à la fois vers l'élément suivant et précédent, nous parlons alors de liste doublement chainée. Retenez donc qu'une liste chainée nous permet de stoker un nombre inconnu d'éléments.

Voici une représentation schématique des listes doublement chainées:



Vous pouvez donc voir sur ce schéma que chaque élément d'une liste doublement chainée contient :

- Une donnée (ici un simple entier)
- Un pointeur vers l'élément suivant (**`NULL`** si l'élément suivant n'existe pas)
- Un pointeur vers l'élément précédent (**`NULL`** si l'élément précédent n'existe pas)

Passons maintenant à la représentation de ces listes en langage **C**.

```c
typedef char Type;

typedef struct Noeud * Liste;

typedef struct Noeud {
    Type info;
    Liste suivant;
    Liste precedent;
}Noeud;
```

Cette première structure va nous permettre de représenter un nœud (élément) de notre liste chaînée. Nous pouvons alors voir que chaque élément de notre liste contiendra un élément de type **`Type`**.

Pour représenter notre liste chaînée à proprement parler, nous utiliserons une deuxième liste que voici :

```c
typedef struct NoeudD * ListeD;

typedef struct NoeudD{
    int taille;
    Liste tete;
    Liste queue;
}NoeudD;
```

Le champ Taille représente la taille de notre liste chaînée. **`tete`** va pointer vers le premier élément de notre liste alors que **`queue`** va pointer vers le dernier élément. Cela va tout simplement servir à faciliter les différentes opérations que nous effectuerons sur nos listes.

Pour utiliser une liste dans nos programmes nous utiliserons alors :

```
ListD *list = NULL; /* Déclaration d'une liste vide */
```

Nous allons maintenant créer des fonctions nous permettant de réaliser plusieurs opérations sur ces listes.

1.5.1 Allouer une nouvelle liste

Avant de pouvoir commencer à utiliser notre liste chaînée, nous allons créer une fonction nous permettant d'allouer de l'espace mémoire pour notre liste chaînée. La fonction retournera la liste chaînée nouvellement créée.

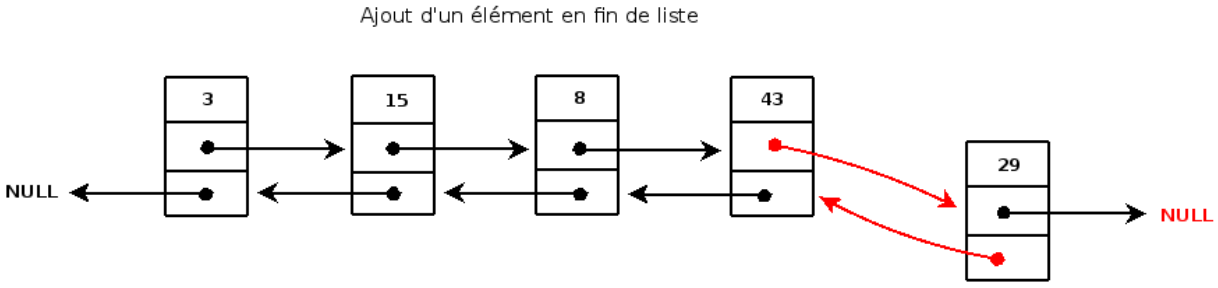
```
void initialiser(ListeD * adrListe){
    (*adrListe) = (ListeD)malloc(sizeof(NoeudD));
    if ((*adrListe) != NULL){
        (*adrListe)->taille = 0;
        (*adrListe)->tete = NULL;
        (*adrListe)->queue = NULL;
    }
}
```

1.5.2 Ajouter un élément

Après avoir alloué une nouvelle liste chaînée, voyons maintenant comment ajouter un élément dans celle-ci.

1.5.2.1 Ajout en fin de liste

Grâce à la forme de notre structure, l'ajout en fin de liste va être simplifié. En effet, rappelez-vous, nous gardons toujours un pointeur vers la fin de notre liste, nous n'avons donc nul besoin de parcourir la liste en entier afin d'arriver au dernier élément, nous l'avons déjà. Voici comment va se passer l'ajout en fin de liste:



Voici l'implémentation :

```
void insererFin(ListeD * adrListe, Type valeur){
    // On vérifie si notre liste a été allouée
    if((*adrListe) != NULL){
        // Création d'un nouveau noeud
        Noeud * nouveau = (Noeud*)malloc(sizeof(Noeud));
        // On vérifie si le malloc n'a pas échoué
        if(nouveau != NULL){
            // On 'enregistre' notre donnée
```

```
            nouveau->info = valeur;
            // On fait pointer suivant vers NULL
            nouveau->suivant = NULL;
            // Cas où notre liste est vide
            //(pointeur vers fin de liste à  NULL)
            if ((*adrListe)->queue == NULL){
                // On fait pointer precedent vers NULL
                nouveau->precedent = NULL;
                // On fait pointer la tête de liste vers le nouvel élément
                (*adrListe)->tete = nouveau;
                // On fait pointer la fin de liste vers le nouvel élément
                (*adrListe)->queue = nouveau;
            }
            // Cas où des éléments sont déjà présents dans notre liste
            else {
                // On relie le dernier élément de la liste vers notre
                // nouvel élément (début du chaînage)
                (*adrListe)->queue->suivant = nouveau;
                // On fait pointer precedent vers le dernier élément de la liste
                nouveau->precedent = (*adrListe)->queue;
                // On fait pointer la fin de liste vers notre nouvel élément
                // (fin du chaînage: 3 étapes)
                (*adrListe)->queue = nouveau;
            }
            // Incrémentation de la taille de la liste
            (*adrListe)->taille++;
        }
    }
}
```

1.5.2.2 Ajout en début de liste

Pour ajouter un élément en début de liste, nous allons utiliser exactement le même procédé que pour l'ajout en fin de liste. Grâce à nos pointeurs en début et en fin de liste, nous pouvons nous permettre de reprendre nos implémentations.

Voici la fonction finale :

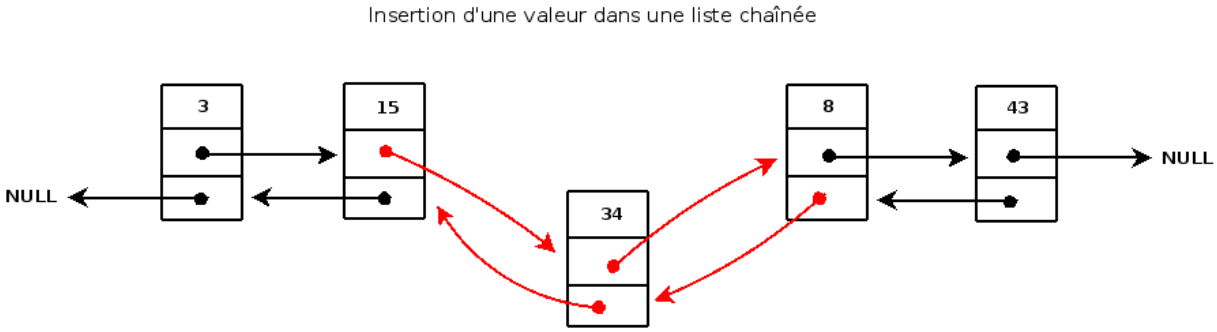
```
void insererDebut(ListeD * adrListe, Type valeur){
    if((*adrListe) != NULL){
        Noeud * nouveau = (Noeud*)malloc(sizeof(Noeud));
        if(nouveau != NULL){
            nouveau->info = valeur;
            nouveau->precedent = NULL;
            if ((*adrListe)->queue == NULL){
                nouveau->suivant = NULL;
                (*adrListe)->tete = nouveau;
                (*adrListe)->queue = nouveau;
            }
            else {
                (*adrListe)->tete->precedent = nouveau;
                nouveau->suivant = (*adrListe)->tete;
                (*adrListe)->tete = nouveau;
            }
            (*adrListe)->taille++;
        }
    }
}
```

1.5.2.3 Insérer un élément dans une position k

Nous disposons désormais de fonctions permettant d'ajouter un élément en début ainsi qu'en fin de liste. Mais si l'on désire ajouter un élément n'importe où dans notre, nous aurons besoin de parcourir notre liste. Nous aurons aussi besoin d'un compteur (que l'on nommera) **k** afin de nous arrêter à la position où nous souhaitons insérer notre nouvel élément. Il nous faut alors réfléchir des différents cas de figure qui peuvent intervenir lorsque nous aurons trouvé notre position:

- Soit nous sommes en fin de liste
- Soit nous sommes en début de liste
- Soit nous sommes en milieu de liste

Cependant, les deux premiers cas sont très faciles à traîter. Enfin, nous disposons de fonctions permettant d'ajouter un élément en début et en fin de liste, il nous suffit donc de les réaliser. Le plus gros de notre travail sera alors de gérer le cas où nous nous trouvons en milieu de liste. Voici un petit schéma permettant de mieux cerner la situation:



Le chaînage va être légèrement plus compliqué. En effet, nous devrons tout d'abord relier nos éléments suivant et précédent à notre nouvel élément puis, inversement, nous devrons relier notre nouvel élément aux éléments suivant et précédent. Le chaînage va alors se dérouler en 4 étapes. A noter qu'il sera nécessaire d'avoir préalablement créé un nouvel élément sans quoi le chaînage ne pourra pas avoir lieu. Pour parcourir notre liste, nous récupérerons le pointeur vers notre début de liste dans un pointeur temporaire. C'est ce pointeur temporaire qui nous servira à parcourir notre liste. Schématiquement, notre liste sera parcourue de gauche à droite. Notre compteur sera bien évidemment incrémenté lors du parcours de chaque maillon de la liste.

Voici ce que cela donne:

```
void insererPositionK(ListeD * adrListe, Type valeur, int position){
    if ((*adrListe) != NULL){
        Noeud * temp = (*adrListe)->tete;
        int k = 1;
        while (temp != NULL && k <= position){
            if (position == k){
                if (temp->suivant == NULL){
                    insererFin(adrListe, valeur);
                }
                else{
                    if (temp->precedent == NULL){
                        insererDebut(adrListe, valeur);
                    }
                    else{
                        Noeud * nouveau = (Noeud*)malloc(sizeof(Noeud));
```

```
                        if (nouveau != NULL){
                            nouveau->info = valeur;
                            temp->precedent->suivant = nouveau;
                            nouveau->precedent = temp->precedent;
                            nouveau->suivant = temp;
                            temp->precedent = nouveau;

                            (*adrListe)->taille++;
                        }
                    }
                }
            }
            else{
                temp = temp->suivant;
            }
            k++;
        }
    }
}
```

1.5.2.4 Libérer une liste

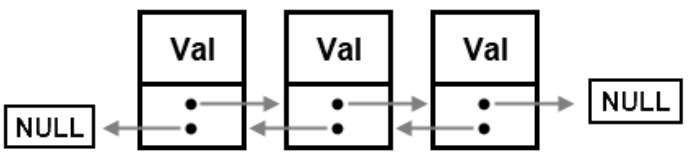
Après avoir utilisé notre liste, nous nous devons de **libérer** tous nos éléments alloués par nos fonctions sous peine d'obtenir ce que l'on nomme des **fuites de mémoire** (*leak memory*).

La fonction suivante permet de libérer notre liste doublement chaînée.

```
void libererListeD(ListeD * adrListe){
    if((*adrListe) != NULL){
        Liste tmp = (*adrListe)->tete;
        while (tmp != NULL){
            Noeud * del = tmp;
            tmp = tmp->suivant;
            free(del);
        }
        free(*adrListe), (*adrListe) = NULL;
        initialiser(adrListe);
    }
}
```

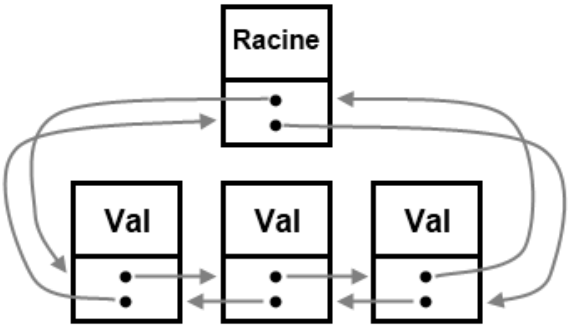
1.6 Opération sur les listes circulaires doublement chainées

Le principe d'une liste doublement chainée est de garder pour chaque élément de la liste un pointeur sur l'élément précédent et sur l'élément suivant. Cela permet notamment de simplifier l'insertion ou la suppression d'un élément donné, ainsi que le parcours en sens inverse. Cependant, cela introduit aussi une certaine dose de complexité dans le codage, car la liste finit dans les deux sens par un pointeur sur NULL, ce qui nécessite d'ajouter dans le code la gestion de ces cas particuliers.



Imaginez maintenant que l'on ferme la boucle, en faisant pointer le pointeur 'precedent' du premier élément sur le dernier élément, et vice-versa.

Nous allons créer un élément spécial, qui sera la racine de notre liste. Cet élément sera à la fois avant le premier élément et après le dernier. C'est lui qui va nous permettre de manipuler tranquillement la liste sans risquer quoi que ce soit.

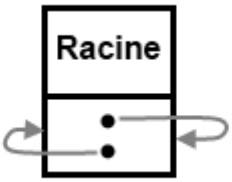


1.6.1 Mise en œuvre : création, parcours et suppression

La définition de la liste se fait de la manière habituelle, avec une structure :

```
typedef char Type;

typedef struct Noeud * ListeC;

typedef struct Noeud{
    Type info;
    ListeC suivant;
    ListeC precedent;
}Noeud;
```

En revanche, une liste vide n'est plus représentée simplement par **NULL** comme le montre la figure suivante :



En effet, pour pouvoir utiliser notre liste, nous devons au préalable la créer, c'est-à-dire créer sa racine. Pour cela, nous ferons :

```
void creerListeCirculaire(ListeC * adrRacine){
    (*adrRacine) = (ListeC)malloc(sizeof(Noeud));
    // si la racine a été correctement allouée
    if ((*adrRacine) != NULL ){
        // pour l'instant, la liste est vide, donc 'precedent'
        // et 'suivant' pointent vers la racine elle-même
        (*adrRacine)->precedent = (*adrRacine);
        (*adrRacine)->suivant = (*adrRacine);
```

```
    }
}
```

1.6.1.1 Parcourir la liste

Pour parcourir la liste, on se sert de sa racine. C'est pourquoi, on doit toujours garder un pointeur sur la racine de la liste. Ce pointeur sur la racine sert en quelque sorte d' "objet" liste. Il nous servira à manipuler la liste. On commence depuis le premier élément après la racine, et on s'arrête lorsque l'on arrive à la racine.

```
/* parcours à l'endroit */

void afficherEndroit(ListeC racine){
    ListeC it, next;
    for(it = racine->suivant; it != racine; it = next ){
        printf("%c ", it->info);
        next = it->suivant;
    }
    printf("\n");
}
```

```
/* parcours à l'envers */

void afficherEnvers(ListeC racine){
    ListeC it, back;
    for(it = racine->precedent; it != racine; it = back ){
        printf("%c ", it->info);
        back = it->precedent;
    }
    printf("\n");
}
```

1.6.1.2 Vider la liste

Pour vider la liste, c'est le même principe : on parcourt la liste et on libère ses éléments.

```
void viderListe(ListeC * adrRacine){
    ListeC it, next;
    for(it = (*adrRacine)->suivant; it != (*adrRacine); it = next ){
        // on enregistre le pointeur sur l'élément suivant
        // avant de supprimer l'élément courant
        next = it->suivant;
        // on supprime l'élément courant
        free(it);
    }
}
```

1.6.1.3 Supprimer la liste

On vide d'abord la liste, puis on supprime la racine. Cette fois ci, il est bon de passer un pointeur sur la racine, pour pouvoir passer la racine à **NULL**;

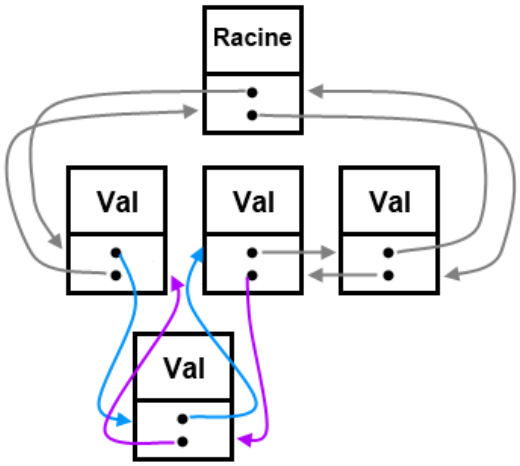
```
void supprimerListe (ListeC * adrRacine){
```

```
    // on vide d'abord la liste
    viderListe(adrRacine);
    free(*adrRacine), (*adrRacine) = NULL;
}
```

1.6.2 Mise en œuvre : opérations sur la liste

1.6.2.1 Ajout d'éléments

Un petit aperçu en image d'une insertion dans une liste circulaire doublement chainée :



On commence par les opérations d'ajout en avant et d'ajout après un élément, car elles vont nous servir pour implémenter l'ajout en tête et en queue.

```
void ajouterAvant(ListeC element, Type val){
    ListeC nouvelElement = (ListeC)malloc(sizeof(Noeud));
    if (nouvelElement != NULL ){
        nouvelElement->info = val;
        // on définit les pointeurs du nouvel élément
        nouvelElement->precedent = element->precedent;
        nouvelElement->suivant = element;
        // on modifie les éléments de la liste
        element->precedent->suivant = nouvelElement;
        element->precedent = nouvelElement;
    }
}
```

```
void ajouterApres(ListeC element, Type val){
    ListeC nouvelElement = (ListeC)malloc(sizeof(Noeud));
    if(nouvelElement != NULL ){
        nouvelElement->info = val;
        // on définit les pointeurs du nouvel élément
        nouvelElement->precedent = element;
        nouvelElement->suivant = element->suivant;
        // on modifie les éléments de la liste
        element->suivant->precedent = nouvelElement;
        element->suivant = nouvelElement;
```

```
    }
}
```

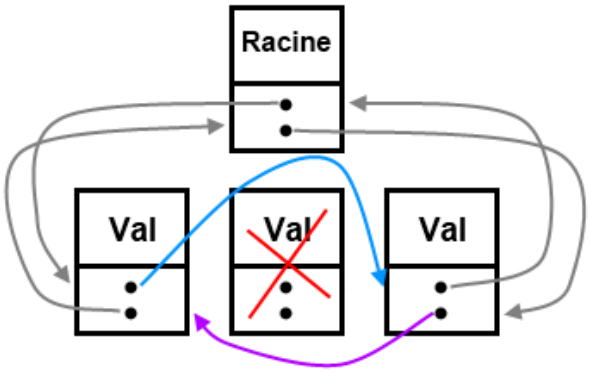
1.6.2.2 Ajout en tête / queue

A présent, pour ajouter un élément en début ou en fin de liste, rien de plus simple : nous allons nous servir de la racine. L'élément qui précède la racine est le dernier élément de la liste, et l'élément suivant la racine est le premier.

```
void ajouterEnTete(ListeC * adrRacine, Type val){
    ajouterApres(*adrRacine, val);
}

void ajouterEnQueue(ListeC * adrRacine, Type val){
    ajouterAvant(*adrRacine, val);
}
```

1.6.2.3 Suppression d'éléments

La suppression d'éléments est encore plus simple, puisque chaque élément connait le précédent et le suivant, et que la racine existe toujours :



a. Supprimer un élément

```
void supprimerElement (ListeC element){
    element->precedent->suivant = element->suivant;
    element->suivant->precedent = element->precedent;
    // on libère la mémoire allouée
    free(element);
}
```

b. Supprimer la tête / queue

Comme d'habitude, on utilise la racine. Attention seulement à vérifier que l'élément existe bien.

```
void supprimerPremierElement(ListeC racine){
    if(racine->suivant != racine)
        supprimerElement(racine->suivant);
}
```

```
void supprimerDernierElement(ListeC racine){
    if(racine->precedent != racine)
        supprimerElement(racine->precedent);
}
```

1.6.2.4 Accès aux éléments

a. Accéder au premier / dernier élément

Ici, on accède aux éléments sans les enlever de la liste :

```
void premierElement(ListeC * adrRacine){
    // on vérifie que l'élément existe bien
    if ((*adrRacine)->suivant != (*adrRacine))
        (*adrRacine) = (*adrRacine)->suivant;
    // sinon on assigne NULL
    else
        (*adrRacine) = NULL;
}

void dernierElement(ListeC * adrRacine){
    // on vérifie que l'élément existe bien
    if ((*adrRacine)->precedent != (*adrRacine))
        (*adrRacine) = (*adrRacine)->precedent;
    // sinon on assigne NULL
    else
        (*adrRacine) = NULL;
}
```

1.6.3 Programme de teste de la liste circulaire

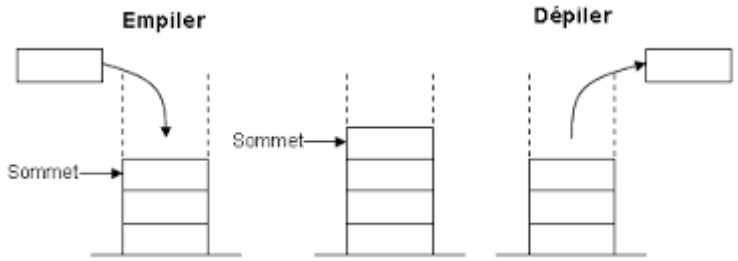
Le programme suivant teste la liste circulaire :

```
main(){
    ListeC lc;
    creeListeCirculaire(&lc);
    ajouterEnTete(&lc, 'A');
    ajouterEnTete(&lc, 'B');
    ajouterEnTete(&lc, 'C');

    ajouterEnQueue(&lc, 'V');

    afficherEndroit(lc);
    afficherEnvers(lc);

    supprimerPremierElement(lc);

    afficherEndroit(lc);
    afficherEnvers(lc);

}
```

2 Les piles (*stack*)

2.1 Introduction

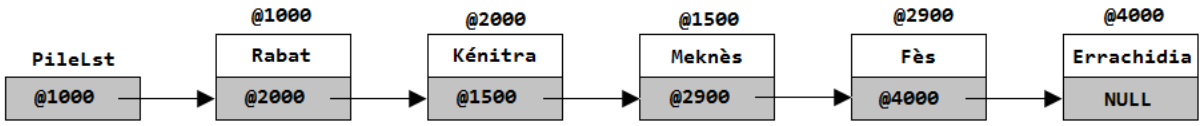
Une pile est une structure qui stocke de manière ordonnée des éléments, mais rend accessible uniquement un seul d'entre eux, appelé le **sommet** de la pile. Quand on ajoute un élément, celui-ci devient le sommet de la pile, c'est-à-dire le seul élément accessible. Quant on retire un élément de la pile, on retire toujours le sommet, et le dernier élément ajouté avant lui devient alors le sommet de la pile. Pour résumer, le dernier élément ajouté dans la pile est le premier élément à en être retiré. Cette structure est également appelée une liste LIFO (Last In, First Out).



Généralement, il y a deux façons de représenter une pile. La première s'appuie sur la structure de liste chaînée vue précédemment et la seconde utilise un tableau (*pointeur*).

2.2 Modélisation par liste chaînée

La première façon de modéliser une pile consiste à utiliser une liste chaînée en n'utilisant que les opérations **`ajouterTete`** et **`retirerTete`**. Dans ce cas, on s'aperçoit que le dernier élément entré est toujours le premier élément sorti. La figure suivante représente une pile par cette modélisation. La pile contient les chaînes de caractères suivantes qui ont été ajoutées dans cet ordre: "**`Fès`**", "**`Errachidia`**", "**`Meknès`**" et "**`Rabat`**".

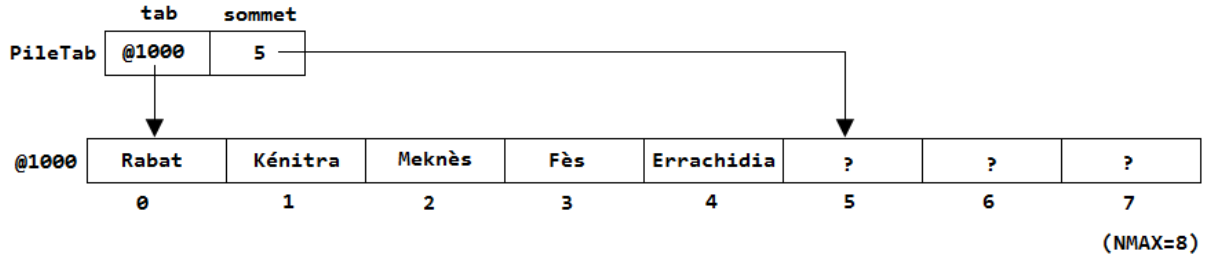


Pour cette modélisation, la structure d'une pile est celle d'une liste chaînée.

```
typedef int Type;

typedef struct Noeud * Liste;

typedef struct Noeud{
    Type info;
    Liste suivant;
}Noeud;

typedef Liste PileLst;
```

2.3 Modélisation par tableau

La deuxième manière de modéliser une pile consiste à utiliser un tableau. L'ajout d'un élément se fera à la suite du dernier élément du tableau. Le retrait d'un élément de la pile se fera en enlevant le dernier élément du tableau. La figure suivante représente une pile par cette modélisation. Elle reprend la même pile que pour l'exemple de modélisation par liste chaînée.



La structure de données correspondant à une pile représentée par un tableau est comme suit :

```
#define NMAX 8

typedef int Type;

typedef struct PileTab{
    Type tab[NMAX];
    int sommet;
}PileTab;
```

NMAX représentant la taille du tableau alloué et **sommet** est le nombre d'éléments dans la pile.

2.4 Opérations sur la structure

A partir de maintenant, nous allons employer le type **Pile** qui représente une pile au sens général, c'est-àdire sans se soucier de sa modélisation. **Pile** représente aussi bien une pile par liste chaînée (**PileLst**) qu'une liste par pointeurs (**PileTab**). Voici les opérations que nous allons détailler pour ces deux modélisations.

- Initialiser une pile vide.
- Tester si une pile est vide.
- Retourne l'élément au sommet d'une pile.
- Empile un élément au sommet d'une pile.
- Dépiler et délivrer l'élément au sommet d'une pile. La nouvelle pile est aussi délivrée.

Les prototypes de ces opérations (paramètres et type de retour) sont les mêmes quelque soit la modélisation choisie.

2.4.1 Opérations pour la modélisation par liste chaînée

2.4.1.1 Initialiser la pile

Cette fonction initialise les valeurs de la structure représentant la pile, afin que celle-ci soit vide. Dans le cas d'une représentation par liste chaînée, il suffit d'initialiser la liste chaînée qui représente la pile.

```
void initialiserPile(PileLst * adrPile){
    initialiserListe(adrPile);
}
```

2.4.1.2 Pile vide ?

Cette fonction indique si la pile **pile** est vide. Dans le cas d'une représentation par liste chaînée, la pile est vide si la liste qui la représente est vide.

```
int pileVide(PileLst pile){
    return listeVide(pile);
}
```

2.4.1.2 Sommet d'une pile

Cette fonction retourne l'élément au sommet de la pile **pile**. Dans le cas d'une représentation par liste chaînée, cela revient à retourner la valeur de l'élément en tête de la liste. A n'utiliser que si la pile **pile** n'est pas vide.

```
Type sommetPile(PileLst pile){
    if(!pileVide(pile))
        return teteListe(pile);
}
```

2.4.1.3 Empiler un élément sur une pile

Cette fonction empile l'élément **valElement** au sommet de la pile **pile**. Pour la représentation par liste chaînée, cela revient à ajouter l'élément **valElement** en tête de la liste.

```
void empiler(PileLst * adrPile, Type valElement){
    insererEnTete(adrPile, valElement);
}
```

2.4.1.4 Dépiler un élément d'une pile

Cette fonction dépile l'élément au sommet de la pile **p** et stocke sa valeur dans **e**. Pour la représentation par liste chaînée, cela revient à récupérer (si possible) la valeur de l'élément en tête de liste avant de le supprimer de cette dernière.

```
void depiler(PileLst * adrPile){
    if(!pileVide(*adrPile))
        supprimerEnTete(adrPile);
}
```

2.4.2 Opérations pour la modélisation par tableau

2.4.2.1 Initialiser la pile

Cette fonction initialise les valeurs de la structure représentant une pile, afin que celle-ci soit vide. Dans le cas d'une représentation par tableau, il suffit de rendre n nul.

```
void initialiserPile(PileTab * adrPile){
    adrPile->sommet = 0;
}
```

2.4.2.2 Pile vide ?

Cette fonction indique si la pile **p** est vide. Dans le cas d'une représentation par tableau, la pile est vide si le **sommet** est nul.

```
int pileVide(PileTab pile){
    return (pile.sommet == 0) ? 1 : 0;
}
```

2.4.2.3 Pile pleine ?

Cette fonction indique si la pile **p** est pleine. Dans le cas d'une représentation par tableau, la pile est pleine si le champ **sommet** est égal à **NMAX**.

```
int pilePleine(PileTab pile){
    return (pile.sommet == NMAX) ? 1 : 0;
}
```

2.4.2.4 Sommet d'une pile

Cette fonction retourne l'élément au sommet de la pile **p**. Dans le cas d'une représentation par tableau, cela revient à retourner la valeur du nième élément du tableau (i.e. l'élément d'indice **sommet - 1**). A n'utiliser que si la pile **p** n'est pas vide.

```
Type sommetPile(PileTab pile){
    if(!pileVide(pile))
        return (pile.tab[pile.sommet - 1]);
}
```

2.4.2.4 Empiler un élément sur une pile

Cette fonction empile l'élément **e** au sommet de la pile pointée par **p**. Pour la représentation par tableau, cela revient à ajouter l'élément **e** à la fin du tableau. S'il reste de la place dans l'espace réservé au tableau, l'empilement peut avoir lieu.

```
void empiler(PileTab * adrPile, Type valElement){
    if(!pilePleine(*adrPile)){
         adrPile ->tab[p->sommet] = valElement;
         adrPile ->sommet ++;
    }
```

```
}
```

2.4.2.5 Dépiler un élément d'une pile

Cette fonction dépile l'élément au sommet de la pile **p**. Pour la représentation par tableau, cela revient à diminuer d'une unité le champ **sommet**. Si la pile n'est pas déjà vide, on peut dépiler.

```
void depiler(PileTab * adrPile){
    if(!pileVide(*adrPile)){
         adrPile->sommet --;
    }
}
```

2.5 Conclusion

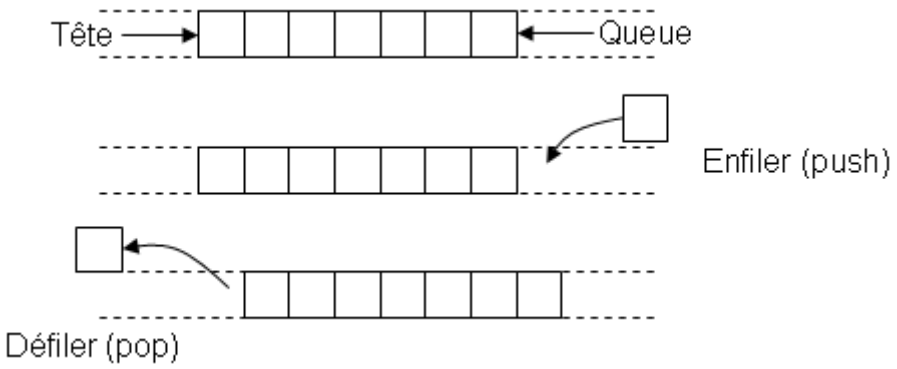
Comme la pile ne permet l'accès qu'à un seul de ses éléments, son usage est limité. Cependant, elle peut être très utile pour supprimer la récursivité d'une fonction. La différence entre la modélisation par liste chaînée et la modélisation par tableau est très faible. L'inconvénient du tableau est que sa taille est fixée à l'avance, contrairement à la liste chaînée qui n'est limitée que par la taille de la mémoire centrale de l'ordinateur. En contrepartie, la liste chaînée effectue une allocation dynamique de mémoire à chaque ajout d'élément et une libération de mémoire à chaque retrait du sommet de la pile. En résumé, la modélisation par liste chaînée sera un peu plus lente, mais plus souple quant à sa taille.

3 Les files (*queue*)

3.1 Introduction

Une file d'attente est une structure qui stocke de manière ordonnée des éléments, mais rend accessible uniquement un seul d'entre eux, appelé la tête de la file. Quant on ajoute un élément, celui-ci devient le dernier élément qui sera accessible. Quant on retire un élément de la file, on retire toujours la tête, celle-ci étant le premier élément qui a été placé dans la file.

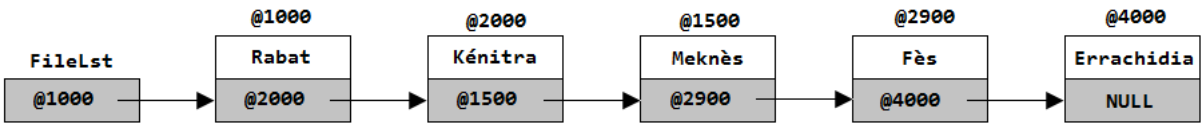
Pour résumer, le premier élément ajouté dans la pile est le premier élément à en être retiré. Cette structure est également appelée une liste FIFO (First In, First Out).



Généralement, il y a deux façons pour représenter une file d'attente. La première s'appuie sur la structure de liste chaînée vue précédemment. La seconde manière utilise un tableau d'une façon assez particulière que l'on appelle modélisation par "tableau circulaire".

3.2 Modélisation par liste chaînée

La première façon de modéliser une file d'attente consiste à utiliser une liste chaînée en n'utilisant que les opérations **`ajouterQueue`** et **`retirerTete`**. Dans ce cas, on s'aperçoit que le premier élément entré est toujours le premier élément sorti. La figure suivante représente une file d'attente par cette modélisation. La file contient les chaînes de caractères suivantes qui ont été ajoutées dans cet ordre: "**`Fès`**", "**`Errachidia`**", "**`Meknès`**" et "**`Rabat`**".

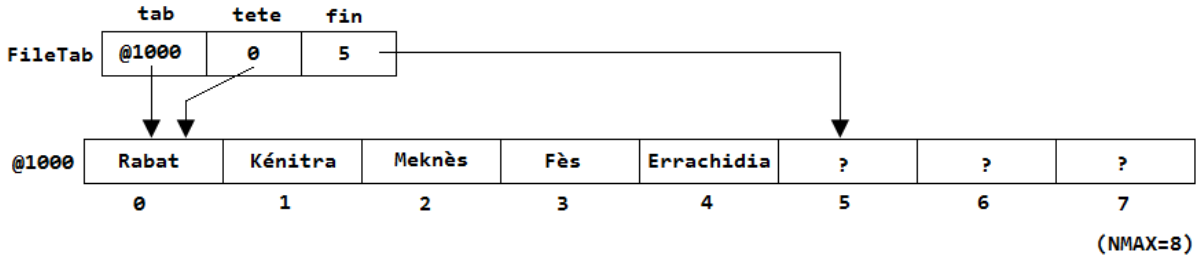


Pour cette modélisation, la structure d'une file d'attente est celle d'une liste chaînée (avec tête et queue).

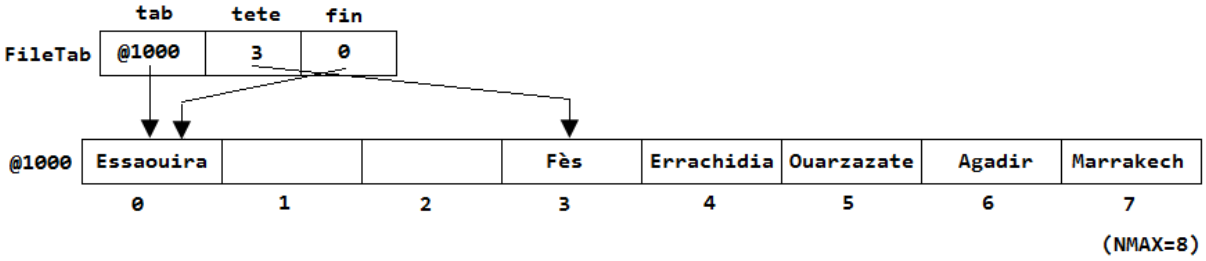
```c
typedef char Type;

typedef struct Noeud * Liste;

typedef struct Noeud{
    Type info;
    Liste suivant;
}Noeud;

typedef Liste FileLst;
```

3.3 Modélisation par tableau circulaire

La deuxième manière de modéliser une file d'attente consiste à utiliser un tableau. L'ajout d'un élément se fera à la suite du dernier élément du tableau. Le retrait d'un élément de la file se fera en enlevant le premier élément du tableau. Il faudra donc deux indices pour ce tableau, le premier qui indique le premier élément de la file et le deuxième qui indique la fin de la file. La figure suivante représente une file d'attente par cette modélisation. Elle reprend la même pile que pour l'exemple de modélisation par liste chaînée.



On peut noter que progressivement, au cours des opérations d'ajout et de retrait, le tableau se déplace sur la droite dans son espace mémoire. A un moment, il va en atteindre le bout de l'espace. Dans ce cas, le tableau continuera au début de l'espace mémoire comme si la première et la dernière case étaient adjacentes, d'où le terme "tableau circulaire". Ce mécanisme fonctionnera tant que le tableau n'est effectivement pas plein. Pour illustrer, reprenons l'exemple précédent auquel on applique **trois opérations de retrait** de l'élément de tête. On ajoute également en queue les éléments suivants et dans cet ordre: "**Ouarzazate**", "**Agadir**", "**Marrakech**", "**Essaouira**".



La file d'attente contient alors, et dans cet ordre: "**Fès**", "**Errachidia**", "**Ouarzazate**", "**Agadir**", "**Marrakech**" et "**Essaouira**". On s'aperçoit que, ayant atteint la fin du tableau, la file redémarre à l'indice 0.

Voici la structure de données correspondant à une file d'attente représentée par un tableau circulaire.

```c
#define NMAX 10

typedef int Type;

typedef struct FileTab{
    Type tab[NMAX];
    int tete;
    int fin;
}FileTab;
```

NMAX est une constante représentant la taille du tableau alloué. tete est l'indice de tableau qui pointe sur la tête de la file d'attente. fin est l'indice de tableau qui pointe sur la case suivant le dernier élément du tableau, c'est-à-dire la prochaine case libre du tableau.

3.4 Opérations sur la structure

A partir de maintenant, nous allons employer le type **`File`** qui représente une file d'attente au sens général, c'est-à-dire sans se soucier de sa modélisation. **`File`** représente aussi bien une file d'attente par liste chaînée (**`FileLst`**) qu'une file d'attente par tableau circulaire (**`FileTab`**). Voici les opérations que nous allons détailler pour ces deux modélisations.

- Initialise une file vide.
- Indique si la file **`file`** est vide.
- Retourne l'élément en tête de la file **`file`**.
- Entre l'élément e dans la file **`file`**.
- Sort l'élément en tête de la file **`file`**.

Les prototypes de ces opérations (paramètres et type de retour) sont les mêmes quelque soit la modélisation choisie.

3.4.1 Opérations pour la modélisation par liste chaînée

3.4.1.1 Initialiser une file

La fonction suivante permet d'initialiser les valeurs de la structure représentant la file **`file`** pour que celle-ci soit vide. Dans le cas d'une représentation par liste chaînée, il suffit d'initialiser la liste chaînée qui représente la file d'attente.

```c
void initialiserFile(FileLst *adrFile){
    initialiserListe(adrFile);
}
```

3.4.1.2 File vide ?

Cette fonction indique si la file **`file`** est vide. Dans le cas d'une représentation par liste chaînée, la file est vide si la liste qui la représente est vide.

```c
int fileVide(FileLst file){
    return listeVide(file);
}
```

3.4.1.3 Tête d'une file

Cette fonction retourne l'élément en tête de la file **`file`**. Dans le cas d'une représentation par liste chaînée, cela revient à retourner la valeur de l'élément en tête de la liste. A n'utiliser que si la file **`file`** n'est pas vide.

```
Type teteFile(FileLst file){
    if(!fileVide(file))
        return teteListe(file);
}
```

3.4.1.4 Entrer un élément dans une file

Cette fonction place un élément **`valElement`** en queue de la file pointée par **`adrFile`**. Pour la représentation par liste chaînée, cela revient à ajouter l'élément **`valElement`** en queue de la liste.

```
void entrerElement(FileLst * adrFile, Type valElement){
    insererEnQueue(adrFile, valElement);
}
```

3.4.1.5 Sortir un élément d'une file

Cette fonction retire l'élément en tête de la file pointée par **`adrFile`**. Pour la représentation par liste chaînée, cela revient à supprimer l'élément en tête de liste.

```
void sortirElement(FileLst * adrFile){
    if(!fileVide(*adrFile))
        supprimerEnTete(adrFile);
}
```

3.4.2 Opérations pour la modélisation par tableau circulaire

3.4.2.1 Initialiser une file

Cette fonction initialise les valeurs de la structure représentant la file pointée par **`adrFile`** pour que celle-ci soit vide. Dans le cas d'une représentation par tableau circulaire, on choisira de considérer la file vide lorsque **`adrFile->tete = adrFile->fin`**. Arbitrairement, on choisit ici de mettre ces deux indices à **`0`** pour initialiser une file d'attente vide.

```
void initialiserFile(FileTab * adrFile){
    adrFile->tete = 0;
    adrFile->fin = 0;
}
```

3.4.2.2 File vide ?

Cette fonction indique si la file **`file`** est vide. Dans le cas d'une représentation par tableau circulaire, la file est vide lorsque **`file.tete = file.fin`**.

```
int fileVide(FileTab file){
    return (file.tete == file.fin) ? 1 : 0;
}
```

3.4.2.3 Tête d'une file

Cette fonction retourne l'élément en tête de la file **file**. Dans le cas d'une représentation par tableau circulaire, il suffit de retourner l'élément pointé par l'indice de tête dans le tableau. A n'utiliser que si la file **file** n'est pas vide.

```
Type teteFile(FileTab file){
    if(!fileVide(file))
        return (file.tab[file.tete]);
}
```

3.4.2.4 Entrer un élément dans une file

Cette fonction place un élément **valElement** en queue de la file d'adresse **adrFile**. Pour la représentation par tableau circulaire, l'élément est placé dans la case pointée par l'indice fin. Ce dernier est ensuite augmenté d'une unité, en tenant compte du fait qu'il faut revenir à la première case du tableau s'il a atteint la fin de celui-ci. D'où l'utilisation de l'opérateur **%** qui retourne le reste de la division entre ses deux membres. Ceci ne peut être exécuté que si le tableau n'est pas plein au moment de l'insertion.

```
void entrerElement(FileTab * adrFile, Type valElement){
    if((adrFile->fin + 1) % NMAX != adrFile->tete){
        adrFile->tab[adrFile->fin] = valElement;
        adrFile->fin = (adrFile->fin + 1) % NMAX;
    }
    else
        printf("Entrer : File pleine !!\n");
}
```

On détecte que le tableau est plein si l'indice fin est juste une case avant l'indice **tete**. En effet, si on ajoutait une case, **fin** deviendrait égale à **tete**, ce qui est notre configuration d'une file vide. La taille maximale de la file est donc **NMAX - 1**, une case du tableau restant toujours inutilisée.

3.4.2.5 Sortir un élément d'une file

Cette fonction retire l'élément en tête de la file d'adresse **adrFile**. Pour la représentation par tableau circulaire, cela revient à récupérer la valeur de l'élément en tête de file avant de le supprimer en augmentant l'indice **tete** d'une unité. Il ne faut pas oublier de ramener **tete** à la première case du tableau au cas où il a atteint la fin de ce dernier.

```
void sortirElement(FileTab * adrFile){
    if(!(fileVide(*adrFile)))
        adrFile->tete = (adrFile->tete + 1) % NMAX;
}
```

3.5 Conclusion

Ce genre de structure de données est très utilisé par les mécanismes d'attente. C'est le cas notamment d'une imprimante en réseau, où les tâches d'impressions arrivent aléatoirement de n'importe quel ordinateur connecté. Les tâches sont placées dans une file d'attente, ce qui permet de les traiter selon leur ordre d'arrivée. Les remarques concernant les différences entre la modélisation par liste chaînée et la modélisation par tableau circulaire sont les mêmes que pour la structure de file. Très peu de différences sont constatées. En résumé, la modélisation par liste chaînée sera un peu plus lente, mais plus souple quant à sa taille.

4 Les arbres

4.1 Généralités

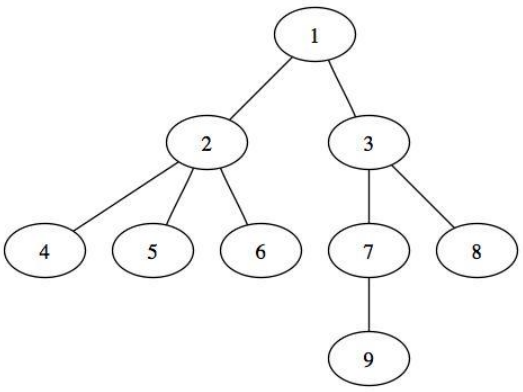
Ce chapitre s'intéresse aux structures de données arborescentes. Ces structures dynamiques récursives constituent un outil important qui est couramment utilisé dans de nombreuses applications : codage de Huffman, interfaces, SGBD, expressions arithmétiques, intelligence artificielle, génomique, etc. Nous distinguerons dans un premier temps les arbres binaires, avant d'étendre et de généraliser cette structure, qu'on peut définir simplement comme une disposition hiérarchique de nœuds, dans laquelle le nœud situé au dessus de chaque fils est son père.

4.2 Définitions

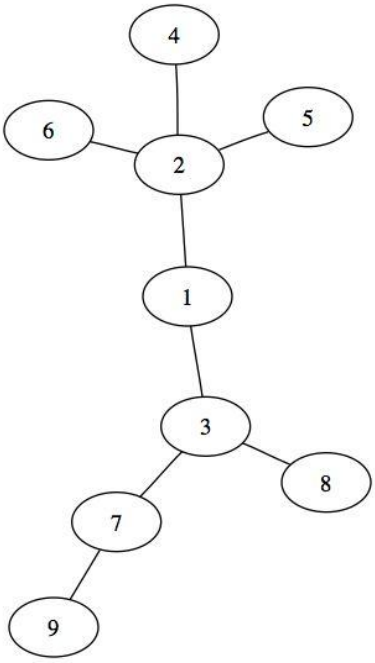
Un arbre est une structure qui peut se définir de manière récursive : un arbre est un arbre qui possède des liens ou des pointeurs vers d'autres arbres. Cette définition plutôt étrange au premier abord résume bien la démarche qui sera utilisé pour réaliser cette structure de données.

4.2.1 Arbres enracinés

On distingue deux grands types d'arbres : les arbres enracinés et les arbres non enracinés. Le premier type d'arbre est celui qui nous intéressera le plus. Un arbre enraciné est un arbre hiérarchique dans lequel on peut établir des niveaux. Il ressemblera plus à un arbre généalogique tel qu'on le conçoit couramment. Voici un exemple d'arbre enraciné :



Le deuxième grand type d'arbre est un arbre non enraciné. Il n'y a pas de relation d'ordre ou de hiérarchie entre les éléments qui composent l'arbre. On peut passer d'un arbre non enraciné à un arbre enraciné. Il suffit de choisir un élément comme sommet de l'arbre et de l'organiser de façon à obtenir un arbre enraciné. Voici un exemple d'arbre non enraciné :



Vous remarquerez qu'il y a équivalence entre les deux arbres précédents. En effet, en réorganisant l'arbre non enraciné, on peut obtenir strictement le même arbre que le premier. En revanche, ce n'est qu'un exemple d'arbre enraciné que l'on peut effectuer avec cet arbre non enraciné, il en existera d'autres.

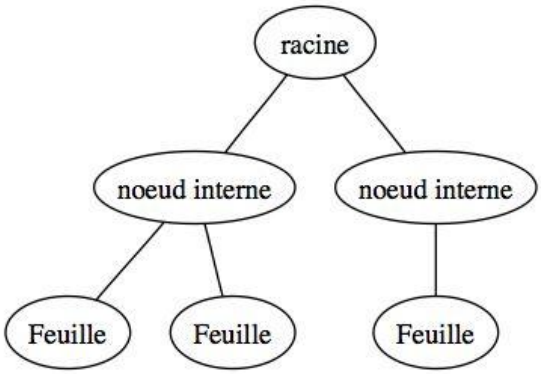
Dans la suite, nous nous intéresserons uniquement aux arbres enracinés.

4.2.2 Terminologie

Précisons maintenant un peu plus les termes désignant les différents composants d'un arbre. Tout d'abord, chaque élément d'un arbre se nomme un nœud. Les nœuds sont reliés les uns aux autres par des relations d'ordre ou de hiérarchie. Ainsi on dira qu'un nœud possède un père, c'est à dire un nœud qui lui est supérieur dans cette hiérarchie. Il possède éventuellement un ou plusieurs fils.

Il existe un nœud qui n'a pas de père, c'est donc la racine de l'arbre. Un nœud qui n'a pas de fils est appelé un feuille. Parfois on appelle une feuille un nœud externe tout autre nœud de l'arbre sera alors appelé un nœud interne.

Voici donc un schéma qui résume les différents composants d'un arbre :



4.2.3 Arité d'un arbre

On a aussi envie de qualifier un arbre sur le nombre de fils qu'il possède. Ceci s'appelle l'arité de l'arbre. Un arbre dont les nœuds ne comporteront qu'au maximum n fils sera d'arité n. On parlera alors d'arbre n-aire. Il existe un cas particulièrement utilisé : c'est l'arbre binaire. Dans un tel arbre, les nœuds ont au maximum 2 fils. On parlera alors de fils gauche et de fils droit pour les nœuds constituant ce type d'arbre.

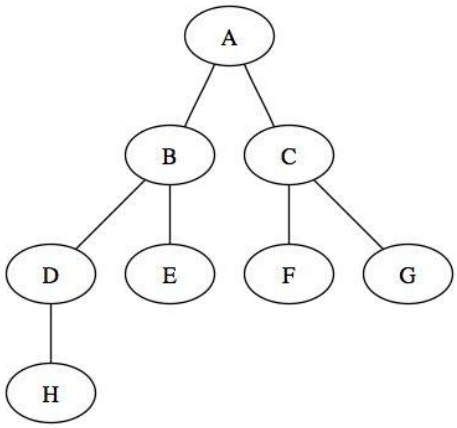
L'arité n'impose pas le nombre minimum de fils, il s'agit d'un maximum, ainsi un arbre d'arité 3 pourra avoir des nœuds qui ont 0, 1, 2 ou 3 fils, mais en tout cas pas plus.

On appelle degré d'un nœud, le nombre de fils que possède ce nœud.

4.2.4 Taille et hauteur d'un arbre

On appelle la taille d'un arbre, le nombre de nœud interne qui le compose. C'est à dire le nombre nœud total moins le nombre de feuille de l'arbre.

On appelé également la profondeur d'un nœud la distance en terme de nœud par rapport à l'origine. Par convention, la racine est de profondeur 0. Dans l'exemple suivant le nœud F est de profondeur 2 et le nœud H est de profondeur 3.

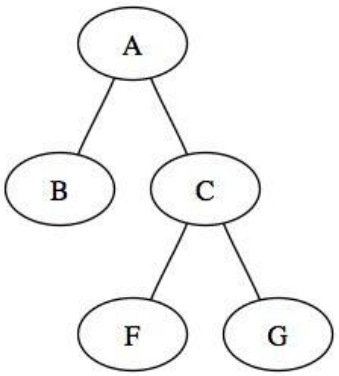


La hauteur de l'arbre est alors la profondeur maximale de ses nœuds. C'est à dire la profondeur à laquelle il faut descendre dans l'arbre pour trouver le nœud le plus loin de la racine. On peut aussi définir la hauteur de manière récursive : la hauteur d'un arbre est le maximum des hauteurs ses fils. C'est à partir de cette définition que nous pourrons exprimer un algorithme de calcul de la hauteur de l'arbre.

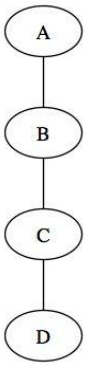
La hauteur d'un arbre est très importante. En effet, c'est un repère de performance. La plupart des algorithmes que nous verrons dans la suite ont une complexité qui dépend de la hauteur de l'arbre. Ainsi plus l'arbre aura une hauteur élevée, plus l'algorithme mettra de temps à s'exécuter.

4.2.5 Arbre localement complet, dégénéré, complet

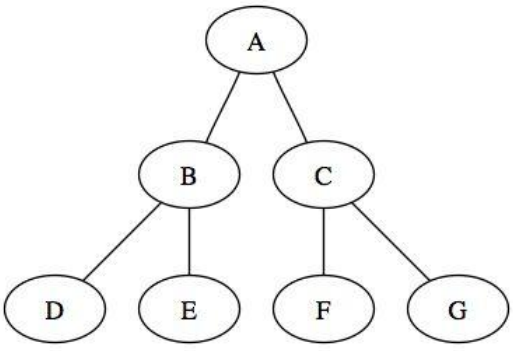
Un arbre binaire localement complet est un arbre binaire dont chacun des nœuds possèdent soit 0 soit 2 fils. Ceci veut donc dire que les nœuds internes auront tous deux fils. Dans ce type d'arbre, on peut établir une relation entre la taille de l'arbre et le nombre de feuille. En effet, un arbre binaire localement complet de taille n aura n+1 feuille. L'arbre suivant est localement complet :



Un arbre dégénéré (appelé aussi filiforme) est un arbre dont les nœuds ne possède qu'un et un seul fils. Cet arbre est donc tout simplement une liste chaînée. Ce type d'arbre est donc à éviter, puisqu'il n'apportera aucun avantage par rapport à une liste chaînée simple. On a alors une relation entre la hauteur et la taille : un arbre dégénéré de taille n a une hauteur égale à n+1. L'arbre suivant est un arbre dégénéré.



On appellera arbre binaire complet tout arbre qui est localement complet et dont toutes les feuilles ont la même profondeur. Dans ce type d'arbre, on peut exprimer le nombre de nœuds n de l'arbre en fonction de la hauteur h : n = 2^(h+1) -1.



4.3 Implémentation des arbres n-aire

Nous allons maintenant discuter de l'implémentation des arbres. Tout d'abord, définissons le nœud. Un nœud est une structure (ou un enregistrement) qui contient au minimum trois champs : un champ contenant l'élément du nœud, c'est l'information qui est importante. Cette information peut être un entier, une chaîne de caractère ou tout autre chose que l'on désire stocker. Les deux autres champs sont le fils gauche et le fils droit du nœud. Ces des fils sont en fait des arbres, on les appelle généralement

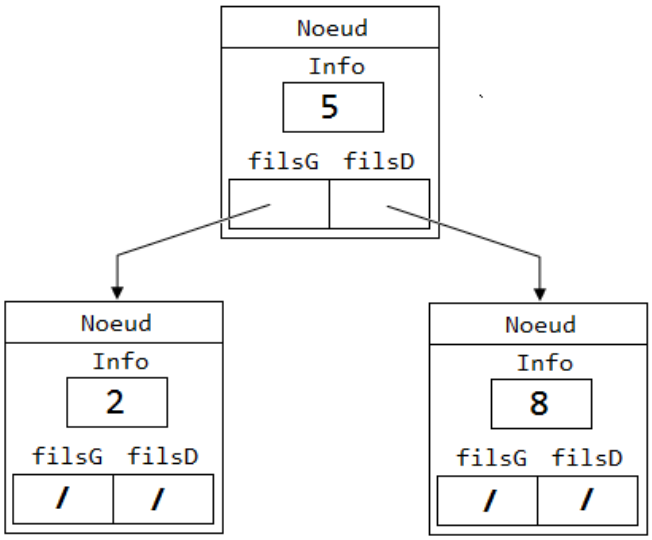
les sous arbres gauches et les sous arbres droit du nœud. De part cette définition, un arbre ne pourra donc être qu'un pointeur sur un nœud.

Voici donc une manière d'implémenter un arbre binaire en langage C :

```
typedef int Type;

typedef struct Noeud * Arbre;

typedef struct Noeud {
    Type info;
    Arbre filsG;
    Arbre filsD;
}Noeud;
```

On remplacera le type **Type** par le type ou la structure de données que l'on veut utiliser comme entité significative des nœuds de l'arbre.

De cette définition, on peut donc aisément constater que l'arbre vide sera représenté par la constante **NULL**.

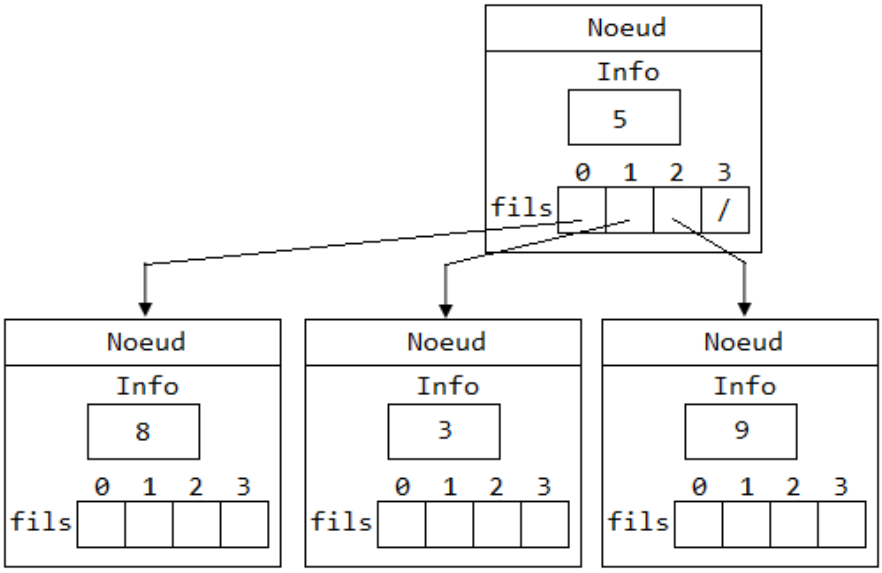


Maintenant, si on veut représenter un autre type d'arbre, nous avons deux solutions : soit nous connaissons à l'avance le nombre maximal de fils des nœuds (ce qui veut dire que l'on connaît l'arité de l'arbre), soit nous ne la connaissons pas.

Dans le premier cas, nous pouvons utiliser un tableau pour stocker les fils. Ainsi pour un arbre d'arité 4 nous aurons l'implémentation suivante :

```
typedef int Type;

typedef struct Noeud * Arbre;

typedef struct Noeud {
      Type info;
      Arbre fils[4];
}Noeud;
```

Ainsi dans ce cas, les fils ne seront pas identifiés par leur position (droite ou gauche) mais par un numéro.



La deuxième solution consiste à utiliser une liste chainée pour la liste des fils.

```
typedef int Type;

typedef struct Noeud * Arbre;
typedef struct Cell * List;

typedef struct Cell{
       Arbre fils;
       List suivant;
}Cell;

typedef struct Noeud{
    Type info;
    List fils;
}Noeud;
```

Ce type d'implémentation est déjà un peu plus compliqué, en effet, nous avons une récursivité mutuelle. Le type **`List`** à besoin du type **`Arbre`** pour s'utiliser mais le type **`Tree`** a besoin du type **`List`** pour fonctionner. Le langage **C** autorise ce genre de construction du fait que le type **`List`** et le type **`Arbre`** sont définis via des pointeurs **`Noeud`**).

Nous nous contenterons que de la première implémentation : l'implémentation des arbres binaires. Mais sachez qu'avec un peu d'adaptation, les algorithmes que nous allons voir sont parfaitement utilisables sur des arbres n-aires.

4.4 Les fonctions de base sur la manipulation des arbres

Afin de faciliter notre manipulation des arbres, nous allons créer quelques fonctions. La première détermine si un arbre est vide.

Voici ce que peut donner cette fonction en **C** :

```
int estVide(Arbre * adrArbre){
       return (*adrArbre == NULL) ? 1 : 0;
```

```
}
```

On peut se demander tout simplement l'utilité d'une telle fonction. Tout simplement parce qu'il est plus simple de comprendre la signification de **`estVide(adrArbre)`** plutôt que **`adrArbre==NULL`**.

Maintenant, prenons deux fonctions qui vont nous permettre de récupérer le fils gauche ainsi que le fils droit d'un arbre. Il faut faire attention à un problème : le cas où l'arbre est vide. En effet, dans ce cas, il n'existe pas de sous arbre gauche ni de sous arbre droit. Pour régler ce problème nous décidons arbitrairement de renvoyer l'arbre vide comme fils d'un arbre vide.

Voici ce que cela donne en **C** :

```
Arbre * filsGauche(Arbre * adrArbre) {
	if (estVide(adrArbre))
		return NULL;
	else
		return &(*adrArbre)->filsG;
}
```

La fonction qui retourne le fils droit sera codée de la même manière mais tout simplement au lieu de renvoyer le fils gauche, nous renvoyons le fils droit. Voila ce que cela donne en C :

```
Arbre * filsDroit(Arbre * adrArbre) {
	if (estVide(adrArbre))
		return NULL;
	else
		return &(*adrArbre)->filsD;
}
```

Passons à une autre fonction qui peut nous être utile : savoir si nous sommes sur une feuille. Ceci donne en C :

```
int estFeuille(Arbre * adrArbre){
    if(estVide(adrArbre))
        return 0;
    else
        if(estVide(filsGauche(adrArbre)) && estVide(filsDroit(adrArbre)))
            return 1;
        else
            return 0;
}
```

Enfin, nous pouvons créer une dernière fonction bien que très peu utile : déterminer si un nœud est un nœud interne. Pour ce faire, deux méthodes : soit on effectue le test classique en regardant si un des fils n'est pas vide, soit on utilise la fonction précédente. On aura donc en **C** la fonction suivante :

```
int estNoeudInterne(Arbre * adrArbre){
```

```
        return !estFeuille(adrArbre);
}
```

4.5 Algorithmes de base sur les arbres binaires

Nous présentons les algorithmes de base sur les arbres en exploitant les propriétés de la récursivité. Il faut savoir qu'il n'y a pas d'autres alternatives. Mais rassurez vous, nous allons procéder en douceur, les fonctions que nous allons voir ne sont pas compliquée. De plus, le schéma est quasiment le même, une fois que vous aurez vu deux ou trois fois ce schéma.

4.5.1 Calcul de la hauteur d'un arbre

Pour calculer la hauteur d'un arbre, nous allons nous baser sur la définition récursive :

- un arbre vide est de hauteur **0**.
- un arbre non vide a pour hauteur **1 +** la hauteur maximale entre ses fils.

De part cette définition, nous pouvons en déduire la fonction **C** suivante :

```
unsigned hauteur(Arbre * adrArbre){
        if(estVide(adrArbre))
                return 0;
        else
                return 1 + max(hauteur(filsGauche(adrArbre)),
                                                hauteur(filsDroit(adrArbre)));
}
```

La fonction **max** n'est pas définie c'est ce que nous faisons maintenant :

```
unsigned max(unsigned a,unsigned b){
        return (a>b)? a : b ;
}
```

4.5.2 Calcul du nombre de nœud

Le calcul du nombre de nœud est très simple. On définit le calcul en utilisant la définition récursive :

- Si l'arbre est vide : renvoyer **0**.
- Sinon renvoyer **1 +** la somme du nombre de nœuds des sous arbres.

On aura donc la fonction **C** suivante :

```
unsigned nbNoeud(Arbre * adrArbre){
        if(estVide(adrArbre))
                return 0;
        else
                return 1 + nbNoeud(filsGauche(adrArbre))
                         + nbNoeud(filsDroit(adrArbre));
}
```

4.5.3 Calcul du nombre de feuilles

Le calcul du nombre de feuille repose sur la définition récursive :

- un arbre vide n'a pas de feuille.

- un arbre non vide a son nombre de feuille défini de la façon suivante :
  - si le nœud est une feuille alors on renvoie **1**
  - si c'est un nœud interne alors le nombre de feuille est la somme du nombre de feuille de chacun de ses fils.

Voici ce que cela peut donner en langage **C** :

```
unsigned nbFeuille(Arbre * adrArbre){
    if(estVide(adrArbre))
        return 0;
    else
        if (estFeuille(adrArbre) )
            return 1;
        else
            return nbFeuille(filsGauche(adrArbre)) +
                                              nbFeuille(filsDroit(adrArbre));
}
```

4.5.4 Nombre de nœud internes

Maintenant, pour finir avec les algorithmes de base, nous allons calculer le nombre de nœud interne, Cela repose sur le même principe que le calcul du nombre de feuille. La définition récursive est la suivante :

- un arbre vide n'a pas de nœud interne.
- si le nœud en cours n'a pas de fils alors renvoyer **0**
- si le nœud a au moins un fils, renvoyer **1 +** la somme des nœuds interne des sous arbres.

On en déduit donc aisément la fonction C suivante :

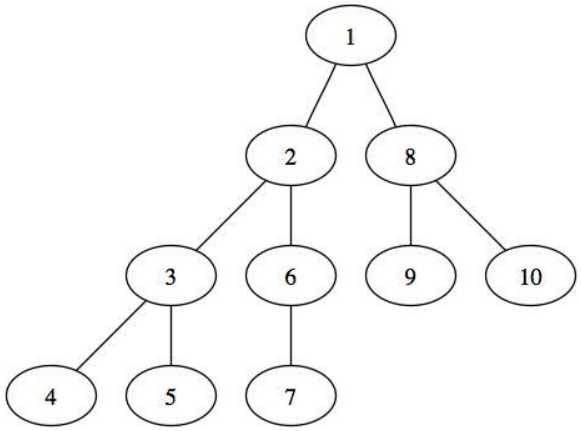
```
unsigned nbNoeudInterne(Arbre * adrArbre){
    if(estVide(adrArbre))
        return 0;
    else
        if(estFeuille(adrArbre))
            return 0;
        else
            return 1 + nbNoeudInterne(filsGauche(adrArbre))
                     + nbNoeudInterne(filsDroit(adrArbre));
}
```

4.6 Parcours d'un arbre

Nous allons découvrir des algorithmes de parcours d'un arbre. Cela permet de visiter tous les nœuds de l'arbre et éventuellement appliquer une fonction sur ces nœuds. Nous distinguerons deux types de parcours : le parcours en profondeur et le parcours en largeur. Le parcours en profondeur permet d'explorer l'arbre en explorant jusqu'au bout une branche pour passer à la suivante. Le parcours en largeur permet d'explorer l'arbre niveau par niveau. C'est à dire que l'on va parcourir tous les nœuds du niveau 1 puis ceux du niveau deux et ainsi de suite jusqu'à l'exploration de tous les nœuds.

4.6.1 Parcours en profondeur

Le parcours en profondeur de l'arbre suivant donne : 1,2,3,4,5,6,7,8,9,10 :



Parcours préfixe de l'arbre : 1-2-3-4-5-6-7-8-9-10

Ceci n'est en fait qu'un seul parcours en profondeur de l'arbre. Il s'agit d'un parcours d'arbre en profondeur à gauche d'abord et préfixe. Précisons tout de suite ces termes.

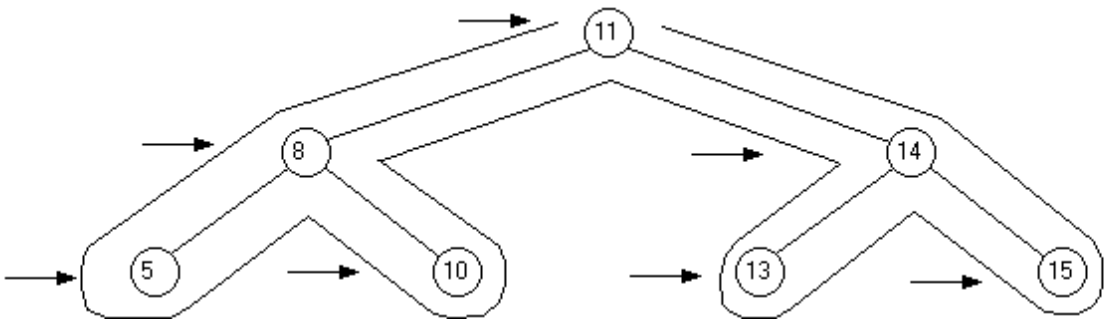
Un parcours en profondeur à gauche est simple à comprendre, cela signifie que l'on parcourt les branches de gauche avant les branches de droite. On aura donc deux types de parcours : un parcours à gauche et un parcours à droite. Dans la plupart des cas, nous utiliserons un parcours à gauche.

La deuxième caractéristique de notre arbre est le parcours dit préfixe. Cela signifie que l'on affiche la racine de l'arbre, on parcourt tout le sous arbre de gauche, une fois qu'il n'y a plus de sous arbre gauche on parcourt les éléments du sous arbre droit. Ce type de parcours peut être résumé en trois lettres : R G D (pour Racine Gauche Droit). On a aussi deux autres types de parcours : le parcours infixe et le parcours suffixe (appelé aussi postfixe). Le parcours infixe affiche la racine après avoir traité le sous arbre gauche, après traitement de la racine, on traite le sous arbre droit (c'est donc un parcours G R D). Le parcours postfixe effectue donc le dernier type de schéma : sous arbre gauche, sous arbre droit puis la racine, c'est donc un parcours G D R. Bien sur, nous avons fait l'hypothèse d'un parcours à gauche d'abord mais on aurait pû aussi faire un parcours à droite d'abord.

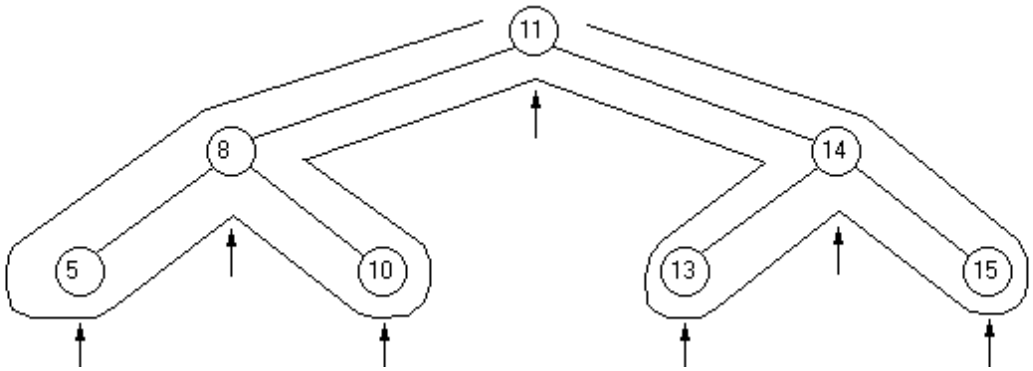
Les types de parcours infixe, suffixe et postfixe sont les plus importants, en effet chacun à son application particulière. Nous verrons cela dans la suite

Maintenant que nous avons la définition des types de parcours, exprimons ceci en C. La fonction traiter_racine, est une fonction que vous définissez vous même, il s'agit par exemple d'une fonction d'affichage de l'élément qui est à la racine.

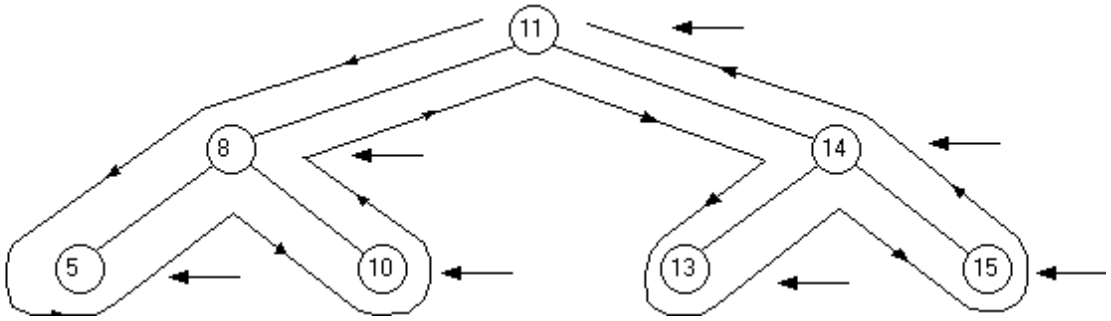
Commençons par le parcours préfixe, celui ci traite la racine d'abord.



Maintenant le parcours infixe :



Et enfin le parcours suffixe :



Voilà pour les trois fonctions de parcours. Vous remarquerez que seul le placement de la fonction (ou procédure) **`traiterRacine`** diffère d'une fonction à l'autre.

La fonction **`traiterRacine`** peut être coder en C comme suit :

```c
void traiterRacine(Arbre * adrArbre){
    if(!estVide(adrArbre))
        printf("%d ", (*adrArbre)->info);
    else
        printf("NULL ");
}
```

Nous pouvons donc traiter ceci facilement, afin de ne pas à avoir à écrire trois fois de suite le même code.

Voici donc le code que l'on peut avoir en C :

```c
void DFS(Arbre * adrArbre, char type){
    if(!estVide(adrArbre)){
        if(type ==1){
            // traiter racine
            traiterRacine(adrArbre);
        }
        DFS(filsGauche(adrArbre), type);
        if(type == 2){
            // traiter racine
            traiterRacine(adrArbre);
        }
        DFS(filsDroit(adrArbre), type);
        if(type == 3){
            // traiter racine
```

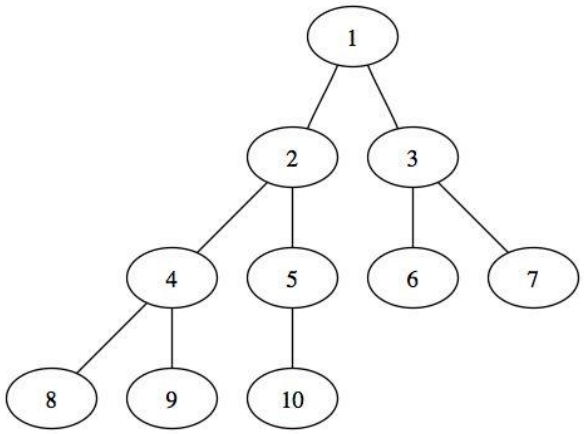
```
            traiterRacine(adrArbre);
        }
    }
}
```

Ainsi à partir de ce code, on peut facilement créer trois fonctions qui seront respectivement le parcours préfixe, le parcours infixe et le parcours suffixe.

```
void DFSPrefix(Arbre * adrArbre){
    DFS(adrArbre,1);
}

void DFSInfix(Arbre * adrArbre){
      DFS(adrArbre,2);
}

void DFSPostfix(Arbre * adrArbre){
      DFS(adrArbre,3);
}
```

4.6.2 Parcours en largeur (ou par niveau)

Nous allons aborder un type de parcours un peu plus compliqué, c'est le parcours en largeur. Il s'agit d'un parcours dans lequel, on traite les nœuds un par un sur un même niveau. On passe ensuite sur le niveau suivant, et ainsi de suite. Le parcours en largeur de l'arbre suivant est : 1 2 3 4 5 6 7 8 9 10.



Une méthode pour réaliser un parcours en largeur consiste à utiliser une structure de données de type file d'attente. Le principe est le suivant, lorsque nous somme sur un nœud nous traitons ce nœud (par exemple nous l'affichons) puis nous mettons les fils gauche et droit non vides de ce nœud dans la file d'attente, puis nous traitons le prochain nœud de la file d'attente.

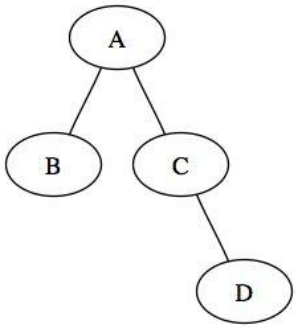
Au début, la file d'attente ne contient rien, nous y plaçons donc la racine de l'arbre que nous voulons traiter. L'algorithme s'arrête lorsque la file d'attente est vide. En effet, lorsque la file d'attente est vide, cela veut dire qu'aucun des nœuds parcourus précédemment n'avait de sous arbre gauche ni de sous arbre droit. Par conséquent, on a donc bien parcouru tous les nœuds de l'arbre.

Voici donc le code **C** en de ladite fonction :

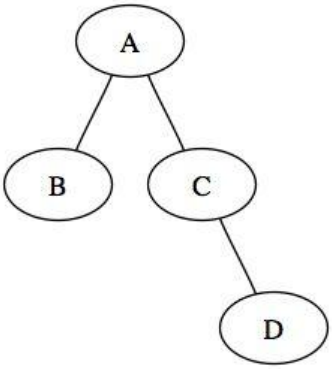
```
void WideSearch(tree T){
      tree Temp;
```

```
        queue F;
        if( ! IsEmpty(T) ){
                Add(F,T);
                while( ! Empty(F) ){
                        Temp = Extract(F);

                        /* Traiter la racine */
                        if( ! IsEmpty(Left(Temp)) )
                                Add(F,Temp);
                        if( ! IsEmpty(Right(Temp)) )
                                Add(F,Temp);
                }
        }
}
```

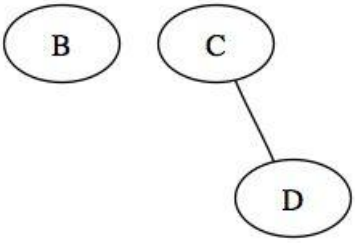
Appliquons ce code à l'arbre suivant :



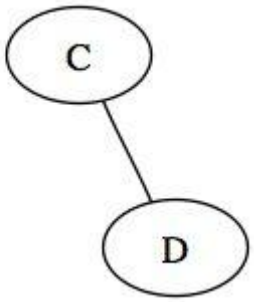
Au tout début de l'algorithme, la file ne contient rien, on y ajoute donc l'arbre, la file d'attente devient donc :



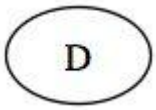
On traite la racine puis on y ajoute les fils droits et gauche, la file vaut donc :



On traite ensuite le prochain élément de la file d'attente. Ce nœud n'a pas de sous arbre, on ajoute donc rien à la file. Celle ci vaut donc :



On traite le prochain nœud dans la file d'attente. Celui ci a un fils droit, nous l'ajoutons donc à la file d'attente. Cette dernière ne contient donc maintenant plus qu'un nœud :



On traite ce nœud. Celui ci n'ayant pas de fils, nous n'ajoutons donc rien à la file. La file est désormais vide, l'algorithme se termine.

4.7 Opérations élémentaires sur un arbre

Maintenant que nous savons parcourir un arbre, que nous savons obtenir des informations sur un arbre, il serait peut être temps de créer un arbre, de l'alimenter et enfin de supprimer des éléments.

4.7.1 Création d'un arbre

On peut distinguer deux types de création d'un arbre : création d'un arbre vide, et création d'un arbre à partir d'un élément et de deux sous arbres. La première méthode est très simple, étant donné que nous avons crée un arbre comme étant un pointeur, un arbre vide est donc un pointeur **`NULL`**. La fonction de création d'un arbre vide est donc une fonction qui nous renvoie la constante **`NULL`**.

La deuxième fonction est un peu plus compliquée mais rien de très impressionnant. Il faut tout d'abord créer un nœud, ensuite, on place dans les fils gauche et droit les sous arbres que l'on a passés en paramètre ainsi que la valeur associée au nœud. Enfin, il suffit de renvoyer ce nœud. En fait, ce n'est pas tout à fait exact, puisque ce n'est pas un nœud mais un pointeur sur un nœud qu'il faut renvoyer. Mais nous utilisons le terme nœud pour spécifier qu'il faut allouer un nœud en mémoire.

Ceci donnera donc en **C** :

```
Arbre creerNoeud(Type val, Arbre ag, Arbre ad){
	Arbre res;
	res = (Arbre)malloc(sizeof(Noeud));
	if(res == NULL){
		printf("Impossible d'allouer le noeud");
		return NULL;
	}
	res->info = val;
	res->filsG = ag;
	res->filsD = ad;
	return res;
}
```

Notre fonction renvoie **NULL** s'il a été impossible d'allouer le nœud. Ceci est un choix arbitraire, vous pouvez très bien effectuer d'autres opérations à la place.

4.7.2 Ajout d'un élément

L'ajout d'un élément est un peu plus délicat. En effet, il faut distinguer plusieurs cas : soit on insère dès que l'on peut. Soit on insère de façon à obtenir un arbre qui se rapproche le plus possible d'un arbre complet, soit on insère de façon à garder une certaine logique dans l'arbre.

Le premier type d'ajout est le plus simple, dès que l'on trouve un nœud qui a un fils vide, on y met le sous arbre que l'on veut insérer. Cependant ce type de technique, si elle sert à construire un arbre depuis le début à un inconvénient : on va créer des arbres du type peigne. C'est à dire que l'on va insérer notre élément tel que l'arbre final ressemblera à ceci :

Ceci est très embêtant dans la mesure ou cela va créer des arbres de très grande hauteur, et donc très peu performant. Néanmoins, il peut arriver des cas où on a parfois besoin de ce type d'insertion. Dans ce cas, l'insertion se déroule en deux temps : on cherche d'abord un fils vide, puis on insère. Ceci peut s'écrire récursivement :

- si l'arbre dans lequel on veut insérer notre élément est vide, alors il s'agit d'une création d'arbre.
- sinon, si le nœud en cours a un fils vide, alors on insère dans le fils vide.
- sinon, on insère dans le fils gauche.

Vous remarquerez que l'on insère du coté gauche, ceci aura pour effet de produire un peigne gauche, et si on insérait du coté droit, nous aurions un peigne droit.

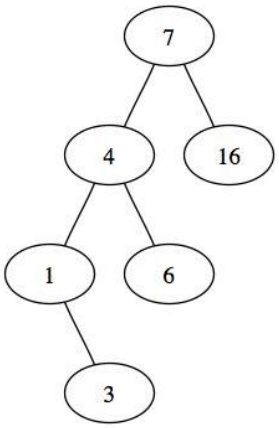
Nous pouvons donc écrire le code C correspondant :

```
void ajouterNoeud(Arbre * adrArbre, Type val){
    if (*adrArbre == NULL ){
        *adrArbre = creerNoeud(val, NULL, NULL);
    }
    else{
        if(estVide(filsGauche(adrArbre))){
            (*adrArbre)->filsG = creerNoeud(val, NULL, NULL);
        }
        else{
            if(estVide(filsDroit(adrArbre))){
                (*adrArbre)->filsD = creerNoeud(val, NULL, NULL);
            }
            else{
                ajouterNoeud(filsGauche(adrArbre), val);
            }
        }
    }
}
```

On remarque donc qu'ici nous créons des arbres qui ne sont pas performant (nous verrons ceci dans la recherche d'élément). Afin d'améliorer ceci, on peut éventuellement effectuer notre appel récursif soit à gauche soit à droite et ceci de façon aléatoire. Ceci permet un équilibrage des insertions des nœuds mais ce n'est pas parfait.

La solution pour obtenir des arbres les plus équilibrés possible consiste en l'utilisation d'arbres binaires dit rouge-noir. Ceci est assez compliqué et nous ne les verrons pas. Cependant, nous allons voir un type d'arbre qui facilite les recherches : les arbres binaires de recherche.

Dans ce type d'arbre, il y a une cohérence entre les nœuds, c'est à dire que la hiérarchie des nœuds respecte une règle. Celle ci est simple. Pour un arbre binaire de recherche contenant des entiers, nous considérerons que les valeurs des nœuds des sous arbres gauche sont inférieures à la racine de ce nœud et les valeurs des sous arbres droit sont supérieures à cette racine. Voici un exemple d'un tel arbre :



Puisque nous sommes dans la partie ajout de nœud, voyons comment ajouter un nœud dans un tel arbre : Le principe d'insertion est simple : si on a un arbre vide, alors il s'agit de la création d'un arbre. Sinon, on compare la valeur de l'élément à insérer avec la valeur de la racine. Si l'élément est plus petit alors on insère à gauche sinon, on insère dans le fils droit de la racine.

Voici le code en C que l'on peut écrire pour insérer un élément dans un arbre binaire de recherche.

```
void insererArbreRecherche(Arbre * adrArbre, Type val){
    if(estVide(adrArbre)){
        *adrArbre = creerNoeud(val, NULL, NULL);
    }
    else{
        if (val < (*adrArbre)->info){
            insererArbreRecherche(filsGauche(adrArbre), val);
        }
        else{
            insererArbreRecherche(filsDroit(adrArbre), val);
        }
    }
}
```

Vous remarquerez que l'on insère les éléments qui sont égaux à la racine du coté droit de l'arbre. Autre remarque, vous constaterez que nous effectuons des comparaisons sur les entités du type Type, ceci n'est valable que pour des valeurs numériques (entier, flottant et caractère). S'il s'agit d'autres types, vous devrez utiliser votre propre fonction de comparaison (comme strcmp pour les chaînes de caractère etc.).

4.7.3 Recherche dans un arbre

Après avoir alimenté notre arbre, il serait peut être temps d'effectuer des recherches sur notre arbre. Il y a principalement deux méthodes de recherche. Elles sont directement liées au type de l'arbre :

- si l'arbre est quelconque et
- si l'arbre est un arbre de recherche.

Nos recherches se contenteront seulement de déterminer si la valeur existe dans l'arbre. Avec une petite adaptation, on peut récupérer l'arbre dont la racine contient est identique à l'élément cherché.

Commençons par chercher l'élément dans un arbre quelconque. Cette méthode est la plus intuitive : on cherche l'élément dans tous les nœuds de l'arbre. Si celui ci est trouvé, on renvoie vrai, si ce n'est pas le cas, on renvoie faux.

Voici le code C associé à cette recherche :

```
int existeArbreSimple(Arbre * adrArbre, Type val){
    if(estVide(adrArbre)){
        return 0;
    }
    else{
        if((*adrArbre)->info == val){
            return 1;
        }
        else{
            return (existeArbreSimple(filsGauche(adrArbre), val)  ||
                        existeArbreSimple(filsDroit(adrArbre), val));
        }
    }
}
```

Nous renvoyons un ou logique entre le sous arbre gauche et le sous arbre droit, pour pouvoir renvoyer vrai si l'élément existe dans l'un des sous arbres et faux sinon.

Ce genre de recherche est correct mais n'est pas très performant. En effet, il faut parcourir quasiment tous les nœuds de l'arbre pour déterminer si l'élément existe.

C'est pour cela que sont apparus les arbres binaires de recherche. En effet, on les nomme ainsi parce qu'ils optimisent les recherches dans un arbre. Pour savoir si un élément existe, il faut parcourir seulement une branche de l'arbre. Ce qui fait que le temps de recherche est directement proportionnel à la hauteur de l'arbre.

L'algorithme se base directement sur les propriétés de l'arbre, si l'élément que l'on cherche est plus petit que la valeur du nœud alors on cherche dans le sous arbre de gauche, sinon, on cherche dans le sous arbre de droite.

Cela se traduit en **C** de la façon suivante :

```
int existArbreRecherche(Arbre *adrArbre, Type val){
    if(estVide(adrArbre)){
        return 0;
    }
    else{
        if((*adrArbre)->info == val){
            return 1;
        }
        else{
            if ((*adrArbre)->info > val){
                return existArbreRecherche(filsGauche(adrArbre), val);
            }
            else{
                return existArbreRecherche(filsGauche(adrArbre), val);
            }
```

```
        }
    }
}
```

4.7.4 Suppression d'un arbre

Par suppression de nœud, nous entendrons suppression d'une feuille. En effet, un nœud qui possède des fils s'il est supprimé, entraîne une réorganisation de l'arbre. Que faire alors des sous arbres du nœud que nous voulons supprimer ? La réponse à cette question dépend énormément du type d'application de l'arbre. On peut les supprimer, on peut réorganiser l'arbre (si c'est un arbre de recherche) en y insérant les sous arbres qui sont devenus orphelins. Bref, pour simplifier les choses, nous allons nous contenter de supprimer complètement l'arbre.

L'algorithme de suppression de l'arbre est simple : on supprime les feuilles une par unes. On répète l'opération autant de fois qu'il y a de feuilles. Cette opération est donc très dépendante du nombre de nœud.

En fait cet algorithme est un simple parcours d'arbre. En effet, lorsque nous devrons traiter la racine, nous appellerons une fonction de suppression de la racine. Comme nous avons dis plutôt que nous ne supprimerons que des feuilles, avant de supprimer la racine, il faut supprimer les sous arbres gauche et droit. On en déduit donc que l'algorithme de suppression est un parcours d'arbre **postfixe**.

Voici le code C associé :

```
void supprimerNoeud(Arbre * adrArbre){
      Arbre * ag = filsGauche(adrArbre);
      Arbre * ad = filsDroit(adrArbre);
      if(!estVide(adrArbre)){
            supprimerNoeud(ag);
            supprimerNoeud(ad);
            free(*adrArbre);
            *adrArbre = NULL;
      }
}
```

On peut se demander pourquoi nous passons par un pointeur sur un arbre pour effectuer notre fonction. Ceci est tout simplement du au fait que le **C** ne fait que du passage de paramètre par copie et par conséquent si on veut que l'arbre soit réellement vide (i.e. : qu'il soit égal à **`NULL`**) après l'exécution de la fonction, il faut procéder ainsi.

Les appels à **`filsGauche`** et **`filsDroit`** au début ne posent aucun problème dans le cas où **`a`** vaut **`NULL`**. En effet, dans ce cas, les deux fonctions renverront le pointeur **`NULL`**.