Ecole Supérieure en Génie Electrique et Energétique d'Oran

Algorithmique 2 et Structures de Données Avancées

SUPPORT DE COURS

Classes Préparatoires

(Ancien Programme)

par

Fatima Zohra LEBBAH

À ma mère

TABLE DES MATIÈRES

TABLE DES FIGURES

LISTE DES TABLEAUX

INTRODUCTION GÉNÉRALE

Le présent support de cours est conforme à l'ancien programme enseigné à la $2^{ème}$ année des classes préparatoires au sein de notre école. Ce document à pour but l'étude de l'algorithmique et structures de données avancées en utilisant les techniques du langage C++.

L'algorithmique est une notion fondamentale pour proposer des méthodes de résolution des problèmes complexes issus de différents domaines. Éventuellement, la mise en place d'un algorithme induit la manipulation des données de types appropriés.

Dans ce présent support pédagogique, nous introduirons un ensemble de figures, avec le souci de toujours bien présenter et transmettre les différents concepts algorithmiques et de structures de données. Nous couvrirons quatre aspects :

- les différents types de boucles et d'instructions de contrôle, les notions de base de la programmation structurée et la portée des variables,
- les principes de base et mathématique de récursivité en algorithmique. Un accent particulier est mis sur le comportement d'un algorithme récursif en mémoire, à savoir l'arbre et la pile d'exécution, à travers une variété d'exemples d'application. L'apport de la récursivité et ces inconvénients par rapport à sa version itérative,
- les notions fondamentales de la complexité algorithmique. Nous mettons en avant son intérêt dans la conception des algorithmes. Nous introduisons les techniques de calculs de la complexité d'une manière simple et conforme au niveau de l'étudiant,
- la définition des structures de données linéaires et arborescentes et leurs applications. Pour chaque type de structure, nous mettons en avant sa position et le comportement des primitives correspondants en mémoire. Outre leur importance intrinsèque, nous exposons l'apport d'un usage judicieux de ces structures plus ou moins complexes.

Ce support de cours s'organise en quatre chapitres :

— Le chapitre 1 les techniques d'évaluation de coûts d'algorithmes des bases algorithmiques et de quelques notions du langage C++.
— Le chapitre 2 porte sur les techniques de programmation C++.
— Le chapitre 3 introduit les fondamentales de la récursivité algorithmique.
— Le chapitre 4 concerne la complexité algorithmique et les techniques d'évaluation de coûts d'algorithmes.
— Le chapitre 5 consacré aux structures de données complexes, à savoir les structures linéaires, les structures arborescentes et les fichiers.

1

RAPPEL DE COURS

1.1 INTRODUCTION

Concevoir un algorithme c'est comme monter une machine à calcul, sous formes d'une suite d'actions élémentaires conçue pour réaliser un traitement bien défini sur des données (des entrées) et fournir en sortie des résultats bien précis.

En d'autres termes, un algorithme est une méthode qui permet de résoudre un problème issu d'un domaine donné. Sa mise en œuvre impose le passage par un langage de programmation.

Ce chapitre est un rappel, que nous jugeons indispensable, des notions de base de l'algorithmique enseigné en première année. Afin de faciliter la compréhension des techniques de programmation en C++, nous mettons en avant la relation qui existe entre le langage algorithmique et le langage C++, ainsi que les techniques de passage d'une solution algorithmique vers un programme C++.

1.2 TYPES

L'application d'un algorithme sur un problème donné, fait appel à la manipulation d'un ensemble de variables.

Une variable doit être définie, en lui attribuant un *nom* et un *type*. En algorithmique et dans tout langage de programmation, nous distinguons deux catégories de type, à savoir : les types standards 1.2.1 et les types personnels 1.2.2.

Définition 1.2.1 (type standard) *Un type standard est un type de base prédéfini dans un langage bien défini.*

Définition 1.2.2 (type personnel) *Un type personnel est un type qui n'est pas prédéfini dans le langage de programmation. Il est défini par le programmeur.*

Un type standard en algorithmique est caractérisé par :

1. le domaine de définition : ensemble des valeurs que peut prendre une variable de ce type,
2. les présentations interne et externe :
   a) *interne* : présentation d'une variable de ce type en machine,
   b) *externe* : présentation d'une variable de ce type vis-à-vis le programmeur et l'utilisateur.

3. les opérateurs applicables sur ce type : ensemble des opérateurs que nous pouvons appliquer sur une variable de ce type,
4. les fonctions standards ou prédéfinies : ensemble de fonctions propre au langage, que nous pouvons appliquer sur une variable de ce type.

Nous distinguons deux classes de types standards en algorithmique : ordinaux 1.2.3 et non-ordinaux 1.2.4.

Définition 1.2.3 (type ordinal) *Un type ordinal est un type dont toute variable est codée en machine par un entier qui précise son rang dans la suite des valeurs qui définissent ce type.*

Définition 1.2.4 (type non-ordinal) *Un type non-ordinal est un type dont la variable est codée en machine par un entier qui ne précise en aucun cas, son rang par rapport aux valeurs qui définissent ce type.*

Les tableaux 1 et 2 [Bensaoud-Senhadji, 2005] représentent, respectivement les différentes caractéristiques des deux classes types *ordinaux* et *non-ordinaux*

TABLE 1 – Types ordinaux

type	**domaine de valeurs**	**présentation externe**	**représentation interne**	**opérateurs**	**fonctions**
booléen	faux, vrai	faux, vrai	0,1	et,ou,non, <,>,=, ≠, ≤,≥	ord,pred, succ
entier	min-entier..max-entier	Décimal (15 ou -25)	complément à 2 (16/32 bits)	+,-,*, <,>,=, ≠, ≤,≥, div,mod	ord,pred, succ
caractère	jeu fini et ordonné de caractères	('x', '?', '9', ' " ')	code ASCII (1 octet)	<,>, ≠, ≤,≥	ord,pred, succ,chr

TABLE 2 – Types non-ordinaux

type	**domaine de valeurs**	**présentation externe**	**représentation interne**	**opérateurs**	**fonctions**
réel	sous ensemble des réels	Décimal (3.5 ou -2.3)	virgule flottante (32 bits)	+,-,*,/, <,>,=, ≠, ≤,≥	sin,cos,abs, sqrt,trunc, round, ...
chaîne	suite de caractères du code ASCII	'1 chaîne' 'aujourd'hui'	suite de code ASCII	<,>, =,≠, ≤,≥	length, concat

1.3 ALGORITHME

Un algorithme est une suite d'instructions qui doit respecter la syntaxe du langage algorithmique. Il permet la réalisation d'une suite de traitements sur une

(des) donné(es), et fournir un(des) résultat(s). Ce qui peut être illustré comme suit :

Informations/données ⟶ Traitement ⟶ Informations/résultats

1.3.1 *Structure générale de l'algorithme*

La structure générale d'un algorithme regroupe trois parties dans l'ordre, comme suit :

1. Entête de l'algorithme : comporte le mot clé **Algorithme-principal** suivi du nom de l'algorithme *Nom-algo*, qui doit être de préférence conforme au rôle de l'algorithme.
2. Environnement de l'algorithme : à ce niveau, nous devons :
   — définir les types personnels, la syntaxe regroupe le mot clef **type** suivi du nom du type *nom-type*, puis sa définition.
   — déclarer les variables et les constantes.
3. Corps de l'algorithme : constitue une suite d'instructions, encadrée par les deux mots clés "*début*" et "*fin*" qui marquent, respectivement, le début et la fin de l'algorithme. Ce qui donne le listing suivant :

```
début
    ...
    instruction i-1;
    instruction i;
    instruction i+1;
    ...
fin
```

1.3.2 *Alternatives*

Les alternatives ou les structures conditionnelles servent à effectuer des vérifications avant d'exécuter un bloc d'instructions.

Nous distinguons deux types d'alternatives : complète et réduite.

A. **Alternatives complètes :** la syntaxe de l'alternative se donne comme suit :

```
si <Condition> alors
    <InstructionsSiConditionVrai>
sinon
    <InstructionsSiConditionFausse>
finsi
```

Elle comporte quatre mots clés **si**, **alors**, **sinon** et **finsi**.

Si la condition $<$ *Condition* $>$ est vérifiée alors le bloc d'instructions $<$ *InstructionsSiConditionVrai* $>$ sera exécuté, dans le cas contraire$<$ *InstructionsSiConditio* sera exécuté.

Le mot clé **finsi** marque la fin de l'alternative.

B. **Alternatives réduites :** la syntaxe de l'alternative se donne comme suit :

```
si <Condition> alors
    <InstructionsSiConditionVrai>
finsi
```

Elle comporte trois mots clés **si**, **alors** et **finsi**.

Si la condition $<$ *Condition* $>$ est vérifiée alors le bloc d'instructions $<$ *InstructionsSiConditionVrai* $>$ sera exécuté, sinon c'est l'instruction qui suit le mot clé **finsi** qui sera exécutée.

1.3.3 *Boucles*

Une boucle permet de boucler sur un bloc d'instructions et répéter son exécution pour un nombre fini de fois. L'itération s'arrête après avoir atteint la *condition d'arrêt*, qui est exprimée soit par une expression booléenne, soit par le nombre d'itération fixé au préalable.

Nous distinguons trois types de boucles : *tantque*, *pour* et *répéter*.

A. **La boucle *tant que* :** le schéma de la boucle *tantque* est le suivant :

```
tantque <Condition> faire
  ...
  <InstructionsTantQueConditionVrai>
  ...
finfaire
```

Le principe de la boucle est d'exécuter les instructions <*InstructionsTantQueConditionVrai*> tant que la condition <*Condition*> est vérifiée. Donc, <*Condition*> est une condition de continuité pour la boucle.

B. **La boucle *répéter* :** le schéma de la boucle *répéter* se donne comme suit :

```
répéter
  ...
  <InstructionsTantQueConditionFausse>
  ...
jusqu'à <Condition>
```

Elle permet d'exécuter les instructions <*InstructionsTantQueConditionFausse*> jusqu'à ce que la condition <*Condition*> soit vérifiée. Par conséquent, <*Condition*> est une condition d'arrêt.

C. **La boucle *pour* :** nous donnons le schéma de la boucle *pour* comme suit :

```
pour comp de ValInit à ValFinal pas faire
  ...
  <Instructions>
  ...
finfaire
```

L'exécution de la boucle *pour* suit le principe suivant :

1. le compteur *comp* est initialisé à la valeur initiale *ValInit*,
2. *comp* est testé s'il ne dépasse pas la valeur finale *ValFinal* : si $comp > ValFinal$ alors on sort de la boucle sinon on passe à l'étape 3,
3. le bloc d'instructions <*Instructions*> est exécuté,
4. le compteur *comp* est incrémenté $comp \leftarrow comp + pas$,
5. aller à l'étape (2).

NB : On peut omettre le pas dans le cas où $pas = 1$.

1.4 TYPES STRUCTURÉS

Un type structuré est un type composé et basé sur les types prédéfinis ou d'autres types personnels. Nous avons deux types structurés : les tableaux et les enregistrements(ou les articles).

1. les tableaux : ils regroupent obligatoirement, plusieurs valeurs de même type,
2. les enregistrements(ou les articles) : il peuvent regrouper des valeurs de types différents.

Le tableau ci-dessous 3 [Bensaoud-Senhadji, 2005] les différentes syntaxes qui nous permettent de :
— définir le type,
— déclarer une variable de ce type,
— accéder à une variable (un champs ou un composant) de ce type.

TABLE 3 – Types structurés : tableaux et enregistrements/articles

Type	**Définition**	**Nom des composants**	**Accès aux composants**
vecteur ou tableau à 1 dimension array	tableau [*typeindice*] de type de base T : tableau $[-2 \cdots 10]$ de entier	variable indicée	T[*expression*] T[$2 * 3$] composant de rang 9
tableau à n dimensions	tableau $[t_{i,1}, t_{i,2}, ..., t_{i,n}]$ de type base	variable indicée	T$[exp_1, exp_2, ..., exp_n]$
enregistrement article structure record	article $\{i_1 : t_1; i_2 : t_2; ...; i_n : t_n\}$ Date : article $\{j$: 1..31 ; m : 1..12; a : 2000..2010$\}$	champs	Par selecteur de champs Date.j

1.4.1 *Implémentation des vecteurs*

Un vecteur est un tableau à une dimension. Il peut être illustré par la figure ci-dessous 1 :

A. Déclaration d'un vecteur : la déclaration d'un vecteur de taille $t = ind_n - ind_1 + 1$ se donne comme suit :

nom-tab : **tableau** $[ind_1 \cdots ind_n]$ **de** *type-valeur* ;

sachant que :

- *nom-tab* est le nom de la variable de type tableau,
- **tableau** est le mot clé qui désigne le type tableau en algorithmique,
- ind_1 et ind_n sont les indices début et fin des composants du tableau (voir la figure 1),

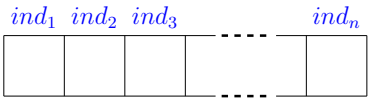


FIGURE 1 – Structure d'une variable de type tableau à une dimension

- *type-valeur* est le type des valeurs qui vont être chargées dans le tableau.

B. Accès à un composant du vecteur : l'accès à la $i^{ème}$ case du vecteur se donne comme suit :

$$nom\text{-}tab[i],$$

Exemple 1.4.1 (manipulation des boucles et des vecteurs) *L'algorithme ci-dessous permet de :*

— *saisir le nombre de jours et la température de chaque jour,*
— *calculer et afficher la moyenne des températures.*

Par conséquent, nous avons à utiliser les variables :

- ***nbj**, **S** et **i** de type entier qui représentent respectivement, le nombre de jours, la somme des températures et le compteur de la boucle "pour",*
- ***T** de type tableau d'entiers, qui doit contenir les températures à saisir dans l'algorithme,*
- ***moyenne** de type réel, qui doit contenir la moyenne calculée des **nbj** températures.*
- *La boucle "pour" :*

```
Algorithme-principal température
 var nbj,T : tableau[1..31] de entier;
       S,i : entier;
       moyenne : réel;
   début
       écrire("le nombre de jours du mois : ");
       lire(nbj);
       S← 0;
       pour i de 1 à nbj faire
            écrire("tempéture du jour", i, ":");
            lire(T[i]);
            S←S+T[i];
       finfaire;
       moyenne←S/nbj;
       écrire("Moyenne des températures :", moyenne);
   fin
```

- *La boucle "répéter" :*

```
Algorithme-principal température
 var nbj,T : tableau[1..31] de entier;
       S,i : entier;
       moyenne : réel;
   début
       écrire("le nombre de jours du mois : ");
       lire(nbj);
       S← 0;
       i← 1;
       répéter
            écrire("tempéture du jour", i, ":");
            lire(T[i]);
            S←S+T[i];
            i←i+1;
       jusqu'à i>nbj;
       moyenne←S/nbj;
       écrire("Moyenne des températures :", moyenne);
   fin
```

- *La boucle "tantque" :*

```
Algorithme-principal température
 var nbj,T : tableau[1..31] de entier;
       S,i : entier;
       moyenne : réel;
   début
       écrire("le nombre de jours du mois : ");
       lire(nbj);
       S← 0;
       i← 1;
       tantque i⩽nbj faire
            écrire("tempéture du jour", i, ":");
            lire(T[i]);
            S←S+T[i];
       finfaire;
       moyenne←S/nbj;
       écrire("Moyenne des températures :", moyenne);
   fin
```

1.4.2 *Implémentation des matrices*

Une matrice est un tableau à deux dimensions. L'illustration de sa structure se donne via la figure ci-dessous 2 :

A. Déclaration de la matrice : la déclaration d'une matrice de taille $n \times m$ telle que $n = ind_n - ind_1 + 1$ et $m = ind'_m - ind'_1 + 1$ se donne comme suit :

$$\boxed{\textit{nom-tab} : \textbf{tableau}\ [ind_1 \cdots ind_n, ind'_1 \cdots ind'_m]\ \textbf{de}\ \textit{type-valeur}}$$

sachant que :

- *nom-tab* est le nom de la variable de type tableau,
- **tableau** est le mot clé qui désigne le type tableau en algorithmique,
- ind_1 et ind_n représentent le premier et le dernier des indices ligne de la matrice (voir la figure 1),
- ind'_1 et ind'_m représentent le premier et le dernier des indices colonne de la matrice (voir la figure 1),
- *type-valeur* est le type des valeurs qui vont être chargées dans le tableau.

La figure 2 illustre la structure d'un tableau à 2 dimensions.

B. Accès à un composant de la matrice : l'accès au composant de la $i^{ème}$ ligne et $j^{ème}$ colonne de la matrice se donne comme suit :

$$\textit{nom-tab}[i, j],$$

Exemple 1.4.2 (manipulation des matrices) *L'algorithme ci-dessous* ***Algorithme-principal moyenne-matrice*** *permet de :*

— *saisir les éléments (les composants) de la matrice,*

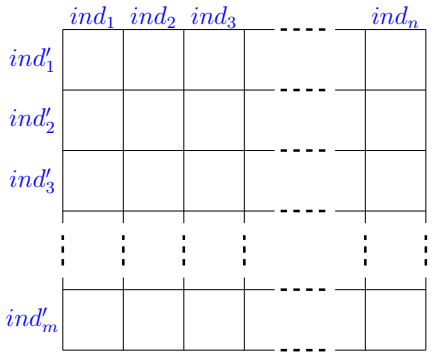


FIGURE 2 – Structure de la matrice

— *calculer et afficher les moyennes correspondant à chaque ligne et de tous les éléments de la matrice.*

Par conséquent, nous avons à utiliser les variables :

- ***M*** *de type tableau d'entiers et de taille* 10×20.
- ***S1***, ***i*** *et* ***j*** *de type entier qui représentent respectivement, la somme de la ligne en cours, le compteur de la première boucle "pour" et le compteur de la deuxième boucle "pour",*
- ***moyenne*** *et* ***S2*** *de type réel qui doivent contenir respectivement, la somme et la moyenne des moyennes des* 10 *lignes.*

Nous donnons l'algorithme en utilisant la boucle ***pour****, puisque le nombre des itérations est connu au préalable. Les deux boucles, première et deuxième, se terminent respectivement en* 10 *et* 20 *itérations.*

```
Algorithme-principal moyenne-matrice;
  var M:tableau [ 1..10, 1..20] de entier;
      S1,i,j : entier;
      moyenne, S2 : réel;
  début
    S2← 0;
    pour i de 1 à 10 faire
       S1← 0;
       pour j de 1 à 20 faire
          écrire("M[", i, ",", j, " ]=");
          lire(M[ i,j]); S← S+M[i,j];
       finfaire
       moyenne← S1/10;
       écrire("La moyenne de la ligne ", i, ":", moyenne);
       S2←S2+moyenne;
    finfaire
    moyenne← S2/20;
    écrire("Moyenne des moyennes des lignes :", moyenne);
  fin
```

L'algorithme peut être écrit en fonction des deux autres types de boucles.

1.4.3 *Implémentation des enregistrements/articles*

Un article (ou un enregistrement) est un type (ou une structure) qui permet de regrouper des valeurs de types différents.

A. Déclaration d'un article : la déclaration d'un article qui regroupe les informations $info_1$, $info_2$, ... et $info_n$, de types respectifs $type_1$, $type_2$, ... et $type_n$, se donne comme suit :

```
nom_article : article
{
        info1 : type1;
        info2 : type2;
                ...
        infon : typen;
}
```

Sachant que :

- *nom-article* est le nom de la variable de type article,
- **article** est le mot clé qui désigne le type article en algorithmique.

B. Accès à un champ de l'article : l'accès au champ $info_i$ de l'article se donne comme suit :

nom-article.infoi,

Exemple 1.4.3 (enregistrements) *L'algorithme ci-dessous "perso", qui permet :*
- *la déclaration de l'article client qui regroupe les données : nom, age et taille concernant un client,*
- *la saisie des données du client et la vérification s'il est vieux ou jeune.*

```
Algorithme-principal perso;
   var client : article
        {
          nom : chaîne de caractères;
          age : entier naturel;
          taille : réel;
        }
  début
      écrire("Donner le nom, l'age et la taille du client :");
      lire(client.nom); lire(client.age);lire(client.taille);
      si (client.age > 50) alors écrire(client.nom, "est vieux")
      sinon écrire(client.nom, "est jeune");
  fin
```

1.5 PROCÉDURES ET FONCTIONS

Souvent, la conception d'un algorithme de résolution d'un problème complexe s'avère difficile. Dans ce type de situation, nous somme amenés à décomposer

le problème en sous-problèmes plus ou moins plus abordables. A ce niveau nous parlons de programmation structurée1.5.1, là où nous faisons appel à deux notions fondamentales, à savoir les procédures et les fonctions.

Définition 1.5.1 (programmation structurée) *La programmation structurée permet de maîtriser et de résoudre séparément, des sous-problèmes issus d'un problème complexe, via des outils de programmation : procédures et fonctions.*

Les procédures et les fonctions sont conçues pour la réalisation d'un traitement bien défini sur des données afin de fournir des résultats attendus. Pour définir une fonction ou une procédure, nous devons préciser les différents paramètres manipulés, à savoir les paramètres d'entrée,les paramètres de sortie et les paramètres d'entrée/sortie. A chacun de ces paramètres, nous devons fixer le nom, le type et le mode de transmission. Nous avons trois modes de transmission :

1. **paramètre d'entrée** : il représente une donnée (ou une entrée) de la procédure ou de la fonction. Il est symbolisé par $\downarrow$.
2. **paramètre de sortie** : il représente un résultat (ou une sortie) de la procédure. Il est symbolisé par $\uparrow$.
3. **paramètre d'entrée/sortie** : il représente parallèlement une donnée et un résultat de la procédure. Il est symbolisé par $\uparrow\downarrow$.

Comme nous pouvons le remarquer, les paramètres d'entrée sont définis pour les procédures et les fonction, alors que les paramètre de sortie et ceux d'entrée/-sortie ne sont définies que pour les procédures.

Les fonctions ne peuvent fournir (ou retourner) qu'un seul résultat. Ce dernier s'exprime via une syntaxe bien définie (voir la sous-section 1.5.2). En d'autres termes les fonctions n'admettent que des paramètres d'entrée. Par conséquent, la précision du mode de transmission dans ce cas devient facultatif.

1.5.1 *Procédures*

Une procédure est une action non primitive qui regroupe des instructions. Elle peut avoir plusieurs paramètres d'entrée, plusieurs paramètres de sortie et plusieurs paramètres de d'entrée/sortie.

De point de vue syntaxique, nous avons deux points à mettre en avant :

A. **Définition d'une procédure :** Une procédure a une structure similaire à celle d'un algorithme principal, à savoir l'en-tête, l'environnement de travail et le corps de la procédure. Nous donnons ci-dessous la syntaxe en algorithmique de ces trois parties :

   1. *Entête de la procédure* : regroupe le nom de la procédure, suivi de ses paramètres formels, séparés par des virgules et encadrés par une parenthèse ouvrante et une parenthèse fermente. Il faut préciser, dans l'ordre, pour chaque paramètre : le mode de transmission ($\downarrow$, $\uparrow$ ou $\downarrow\uparrow$), le nom du paramètre et le type du paramètre. Ce qui se présente comme suit :

$$\textbf{procédure } \textit{NomProcédure } (m_1\ p_1 : t_1, m_2\ p_2 : t_2, ...m_n\ p_n : t_n)$$

Avec :

$NomProcédure$: nom de la procédure,

p_i : $i^{ème}$ paramètre formel,

m_i : mode de transmission du i^{me} paramètre,

t_i : type du $i^{ème}$ paramètre.

2. *Environnement local* : dans cette partie nous décrivons tous les objets à manipuler dans la procédure. Nous précisons, que tout objet défini ou déclaré à ce niveau est propre à la procédure et il ne sera pas reconnu par les parties indépendantes de la procédure.
3. *Corps de la procédure* : regroupe les différentes instructions primitives (ou non-primitives) conçues pour la réalisation du traitement fixé au préalable pour la procédure. Ces instructions doivent être encadrées par les mots clé **début** et **fin**.

Ainsi, la procédure en algorithmique se donne comme suit :

```
procédure NomProcédure(m1p1 : t1, m2p2 : t2, ...mnpn : tn)
        ...
début
        ...
        instructioni
        ...
fin
```

B. **Appel d'une procédure :** les paramètres cités dans la définition sont des paramètres formels. Dans l'appel de la procédure, on fait appel à des paramètres effectifs en suivant la syntaxe ci-dessous :

$$NomProcédure\ (pe_1, pe_2, ...pe_n).$$

Sachant que :

- l'appel de la procédure doit porter le nom *NomProcédure* qui lui a été attribué au cours de la déclaration,
- l'ordre des paramètres effectifs dans l'appel doit être conforme à celui de la définition de la procédure en type et en mode de transmission. En d'autres termes, pe_i doit avoir le même type et le même mode de transmissions avec p_i. Ainsi, le nombre des paramètres effectifs doit être égal au nombre des paramètres formels.

Exemple 1.5.2 (permutation de deux entiers) *Soient deux entiers x et y. Donnez la procédure* ***permut*** *et son application (dans un algorithme principal) qui fait la permutation entre x et y.*

En considérant ***permut*** *comme une boîte noire, la situation s'exprime via la figure 3 comme suit :*

Supposant, que les deux variables x et y sont assignées des valeurs respectives 3 *et* -1*. L'illustration en mémoire, avant et après la permutation, se décrit via la figure 4.*

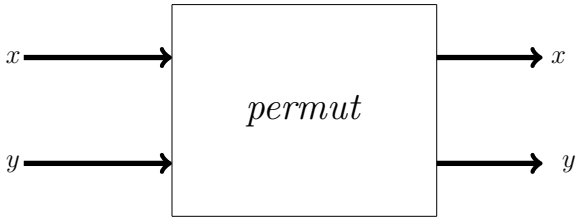
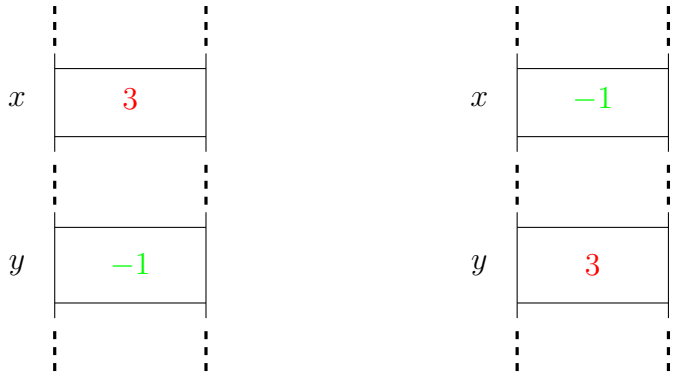


FIGURE 3 – Paramètres d'entrée/sortie de la procédure **permut**



(a) Avant la permutation

(b) Après la permutation

FIGURE 4 – Les variables x et y en mémoire

La figure 4a représente la position des x et y en mémoire.

La figure 4b montre le changement réalisé sur les valeurs portées auparavant par les deux variables.

Nous constatons que l'emplacement des variables ne change pas, alors que leurs valeurs sont permutées.

La procédure de permutation des deux entiers se donne comme suit :

***procédure** permut(↓↑a,b :entier)*

```
var z : entier;
début
  z← x;
  x← y;
  y← z;
fin
```

*Nous constatons que a et b sont des paramètres d'entrée/sortie, puisque la procédure **permut** les reçoit en données au début et les retourne en résultats à la fin du traitement. Autrement dit, a et b jouent le rôle de paramètres d'entrée et de sortie en parallèle.*

Pour faire appel à la procédure dans un algorithme principal nommé AlgoTest :

Algorithme Principal *AlgoTest*

```
var x,y : entier;
début
      x=← 3;
      y=← -1;
      permut(x,y);
      écrire("x=", x, "et y=",y);
fin
```

Les variables x et y représentent les paramètres effectifs de l'appel de la procédure ***permut*** *lors de son appel. Effectivement, la procédure s'applique sur les variables nommées x et y.*

1.5.2 *Fonctions*

Une fonction est une action non primitive qui regroupe des instructions. Elle ne peut avoir que des paramètres d'entrée et elle ne peut retourner qu'un seul résultat.

De point de vue syntaxique, nous avons les deux points ci-dessous :

A. **Définition d'une fonction :** La structure d'une fonction est composée d'un en-tête, un environnement de travail et le corps de la fonction. Nous donnons ci-dessous la syntaxe en algorithmique de ces trois parties :

1. *Entête de la fonction* : regroupe le nom de la fonction, suivi de ses paramètres formels séparés par des virgules et encadrés par une parenthèse ouvrante et une parenthèse fermente, puis le type du retour. Il faut préciser, dans l'ordre, pour chaque paramètre : le mode de transmission (↓), le nom du paramètre et le type du paramètre. Ce qui se présente comme suit :

   fonction *NomFonction* $(\downarrow p_1 : t_1, \downarrow p_2 : t_2, ... \downarrow p_n : t_n)$: type_retour

   ou

   fonction *NomFonction* $(p_1 : t_1, p_2 : t_2, ...p_n : t_n)$: type_retour

   Avec :

   NomProcédure : nom de la procédure,

   pf_i : $i^{ème}$ paramètre formel,

   t_i : type du $i^{ème}$ paramètre.

2. *Environnement local* : dans cette partie nous décrivons tous les objets (variables et types personnels) à manipuler dans la fonction. Nous précisons, que tout objet défini ou déclaré à ce niveau est propre à la fonction et il ne sera pas reconnu par les parties indépendantes de la fonction.

3. *Corps de la fonction* : regroupe les différentes actions primitives (ou non-primitives) conçues pour la réalisation du traitement fixé au préalable. Ces instructions doivent être encadrées par les mots clé **début** et **fin**. La dernière instruction doit être une instruction de retour. Cette dernière est

une instruction qui exprime l'action qui fournit le résultat calculé par la fonction, elle suit la syntaxe suivante :

NomFonction=exp ;

exp est une expression de calcul qui retourne un résultat de type *type_retour* [1].

Ainsi, la fonction en algorithmique se donne comme suit :

```
fonction NomFonction(p1 : t1, p2 : t2, ...pn : tn)
        ...
début
        ...
        instructioni
        ...
        NomFonction=exp;
fin
```

B. **Appel d'une fonction :** les paramètres cités dans la définition sont des paramètres formels. Dans l'appel de la fonction, on fait appel à des paramètres effectifs en suivant la syntaxe ci-dessous :

$$\text{res}=NomFonction(pe_1, pe_2, ...pe_n).$$

Sachant que :

- l'appel de la procédure doit porter le nom *NomProcédure* qui lui a été attribué au cours de la déclaration,
- l'ordre des paramètres effectifs doit être conforme à celui de la définition de la procédure en type. En d'autres termes, pe_i et p_i doivent avoir le même type. Ainsi, le nombre des paramètres effectifs doit être égal au nombre des paramètres formels.

Exemple 1.5.3 (somme des n premiers nombres entiers positifs) *Donnez la fonction* ***somme*** *qui calcule la somme des n premiers entiers positifs.*

Pour cela la fonction reçoit en entrée la valeur de n et retourne en résultat la somme

$$\sum_{i=1..n} i$$

. Ce qui s'exprime par la figure 5 c-dessous :

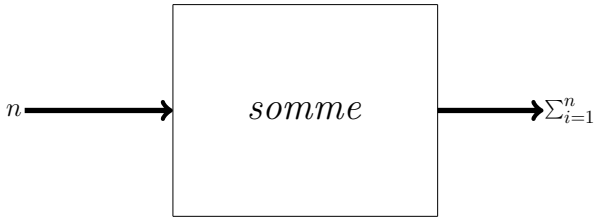


FIGURE 5 – Paramètre d'entrée et résultat de la fonction **somme**

1. Cette expression peut être une variable qui est de type *type_retour*

La fonction ***somme*** *et son appel se donnent respectivement, comme suit :*

```
fonction somme(m:0..maxentier):0..maxentier;
var S,i : entier;
début
   S←0;
   pour i de 1 à n faire
      S←S+i;
   finfaire
   Somme←S;
fin
```

Algorithme Principal *AlgoTest*

```
var n : entier;
début
  n←5;
      res←somme(n);
fin
```

et

La figure 6 illustre l'impact de l'appel de la fonction ***somme*** *en mémoire, pour* $n = 5$ *:*

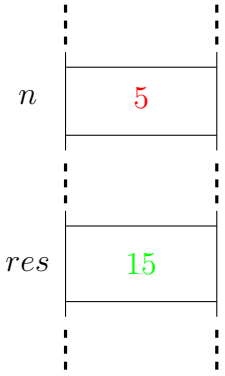


FIGURE 6 – Application de la fonction **somme** : le paramètre d'entrée et le résultat en mémoire

La variable ***n*** *représente le paramètre effectif de la fonction* ***somme*** *lors de son appel. Effectivement, la somme s'applique sur la variable nommée* ***n****. Le résultat est chargé dans la variable* ***res*** *dont le type est identique à celui de la fonction.*

1.6 PASSAGE VERS LE C/C++

1.6.1 *Boucles*

Dans la section 1.3.3 nous avons décrit les trois types de boucles, à savoir *tant que*, *répéter* et *pour*. Nous donnons dans la suite l'équivalent de chacune de ces boucles en C/C++.

A. **Boucle « *while do* » :** Le schéma de la boucle *while*, l'équivalent de *tant que* est le suivant :

```
while (<Condition>) do {
  ...
  <InstructionsTantQueConditionVrai>
  ...
}
```

Le principe de la boucle est identique à celui de la boucle *tant que*. La condition de continuité est encadrée par les deux parenthèses et les instructions <InstructionsTantQueConditionVrai> sont encadrées par une accolade ouvrante et une accolade fermente.

Dans le cas où <InstructionsTantQueConditionVrai> ne contient qu'une seule instruction, les deux accolades deviennent facultatives.

Le bloc d'instructions <InstructionsTantQueConditionVrai> s'exécute zéro ou plusieurs fois.

B. **Boucle « *do while* » :** Le schéma de la boucle se donne comme suit :

```
do{
  ...
  <InstructionsTantQueConditionVrai>
  ...
}while(<Condition>);
```

Elle permet d'exécuter les instructions <*InstructionsTantQueConditionVrai*> jusqu'à ce que la condition <*Condition*> ne soit plus vérifiée. Par conséquent, la négation de <*Condition*> est une condition d'arrêt.

Le bloc d'instructions <InstructionsTantQueConditionVrai> s'exécute au moins une fois.

C. **Boucle « *for* » :** Le schéma de la boucle *for* se donne comme suit :

```
for(int comp=ValInit, comp<=ValFinal, i=i+pas) do
{
  ...
  <Instructions>
  ...
}
```

L'exécution de la boucle *for* suit le même principe que celui de la boucle *pour*. Le bloc d'instructions <Instructions> concernant la boucle *pour* est encadré par les deux accolades.

Le bloc d'instructions <InstructionsTantQueConditionVrai> s'exécute zéro ou plusieurs fois.

1.6.2 *Instructions de contrôle*

Pareillement, les alternatives réduite et complète, se comportent de la même manière que celles citées dans la section 1.3.2. La condition <Condition> doit être

encadrée par des parenthèses et chacun des blocs d'instruction <InstructionsSiConditionVrai> et <InstructionsSiConditionFaux> par des accolades.

A **Alternatives réduites :** La syntaxe se donne comme suit :

```
if (<Condition>) {<InstructionsSiConditionVrai>}
```

B **Alternatives complètes** : La condition doit être encadrée par les deux parenthèses et les bloc d'instruction <InstructionsSiConditionVrai> par les accolades ouvrante et fermente. La syntaxe se donne comme suit :

```
if (<Condition>) {<InstructionsSiConditionVrai>}
else {<InstructionsSiConditionFausse>}
```

ou

```
<Condition> ? <InstructionsSiConditionVrai> :
<InstructionsSiConditionFausse>
```

Exemple 1.6.1 (calcul du minimum de deux valeurs) *L'instruction ci-dessous calcule le minimum de i et de j.*

```
Min=(i<j)?i:j;
```

C **Branchement conditionnel :** Contrairement aux alternatives (complète et réduite) qui ne traitent que deux cas au maximum, le branchement conditionnel permet de manipuler plusieurs cas. La syntaxe en C++ se donne comme suit :

```
switch (valeur)
{
case cas1:
        [instruction;
        [break;]
        ]
case cas2:
        [instruction;
        [break;]
        ]
        ...
case casN:
        [instruction;
        [break;]
        ]
default:
        [instruction;]
}
```

1.6.3 *Types prédéfinis*

Les types prédéfinis les plus courants en C/C++ sont récapitulés dans le tableau 4 ci-dessous :

TABLE 4 – Types prédéfinis

Type	**signification**
void	le type vide : utilisé pour spécifier le fait qu'il n'y a pas de type
bool	les booléens : qui peuvent prendre les valeurs true et false
char	le type caractère
wchar_t	les caractères longs
int	le type entier
float	le type réel
double	les réels en double précision
long int ou *long*	entiers longs
short int ou *short*	les entiers courts
long double	les réels en quadruple précision

1.6.4 *Opérateurs*

Le tableau 5 regroupe de Nombreux opérateurs utilisés en C++.

TABLE 5 – Opérateurs utilisés en C++

Opérateur	Nom ou signification
+	Opérateur d'addition
-	Opérateur de soustraction
«	Opérateur de décalage à gauche
»	Opérateur de décalage à droite
<	Opérateur d'infériorité
>	Opérateur de supériorité
<=	Opérateur d'infériorité ou d'égalité
>=	Opérateur de supériorité ou d'égalité
==	Opérateur d'égalité
!=	Opérateur d'inégalité
&	Opérateur et binaire
^	Opérateur ou exclusif binaire
|	Opérateur ou inclusif binaire
&&	Opérateur et logique
||	Opérateur ou logique
? :	Opérateur ternaire
=	Opérateur d'affectation
*=	Opérateur de multiplication et d'affectation
/=	Opérateur de division et d'affectation
% =	Opérateur de modulo et d'affectation
+=	Opérateur d'addition et d'affectation
-=	Opérateur de soustraction et d'affectation
«=	Opérateur de décalage à gauche et d'affectation
»=	Opérateur de décalage à droite et d'affectation
& =	Opérateur de et binaire et d'affectation
| =	Opérateur de ou inclusif binaire et d'affectation
^=	Opérateur de ou exclusif binaire et d'affectation
,	Opérateur virgule

2

TECHNIQUES DE PROGRAMMATION C++

2.1 INTRODUCTION

Ce chapitre regroupe les notions de base des techniques de programmation C++ à savoir, les pointeurs, l'allocation de la mémoire, les similitudes et les différences entre les tableaux statiques et les tableaux dynamiques et la portée des fonctions et des variables.

Pour une meilleure compréhension nous avons intégré différentes figures illustratives qui décrivent le principe et le déroulement de différentes instructions en mémoire.

La première section comporte les principes des pointeurs et l'allocation dynamique de la mémoire. Dans la deuxième nous présentons l'allocation dynamique et les tableaux. La dernière section est consacrée à la portée des fonctions et des variables.

2.2 POINTEURS

Les pointeurs et l'allocation dynamique représentent deux aspects ayant une relation directe avec l'adresse de l'objet informatique.

2.2.1 *Définitions*

Un pointeur est une variable dont la valeur ne peut pas être de type prédéfini ou personnel. La valeur est plutôt de type adresse.

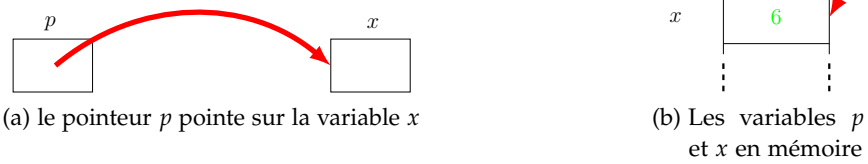
En d'autres termes le pointeur est utilisé pour charger l'adresse d'un emplacement mémoire (Ex. une variable).

Définition 2.2.1 (adresse d'une variable) *L'adresse d'une variable est son emplacement en mémoire qui contient sa valeur. Une variable est physiquement identifiée de façon unique par son* ***adresse****.*

Définition 2.2.2 (pointeur) *Un pointeur est une variable qui contient l'adresse d'un autre objet informatique.*

Exemple 2.2.3 (pointeur et variable) *Soient la variable pointeur p et la variable de type entier x. On suppose que la variable $x = 6$ et le pointeur p sont d'adresses respectives* 1008 *et* 109*. La variable p pointe sur x (ou p contient l'adresse de la variable x).*

La figure 7 illustre la relation entre p et x et sa représentation en mémoire.



(a) le pointeur p pointe sur la variable x

(b) Les variables p et x en mémoire

FIGURE 7 – Le pointeur p et la variable entier x

La figure 7a illustre la relation pointeur-variable entre les variables p et x. La figure 7b montre cette relation en mémoire.

A. **Déclaration d'un pointeur :** On déclare un pointeur en donnant le type de l'objet qu'il doit pointer, suivi de son identificateur précédé d'une étoile :

```
type* identificateur;
```

Exemple 2.2.4 (déclaration d'une variable pointeur) `int *x;`
Cette instruction déclare une variable pointeur x qui pointe sur une valeur de type int.

B. **Initialisation d'un pointeur :** L'initialisation d'un pointeur se fait selon la syntaxe suivante :

```
type* identifiateur(adresse);
```

Cette instruction regroupe deux actions :

1. définition de la variable pointeur *identificateur*, qui doit pointer sur une variable de type *type*. Ce qui s'exprime par l'instruction *type* identificateur*,
2. initialisation de la variable identificateur avec la valeur **adresse** [1]. Ce qui s'exprime par l'instruction *identificateur=adresse*.

Exemple 2.2.5 (initialisation d'une variable pointeur) *Les instructions écrites en C++ ci-dessous expriment par ligne :*
— *la déclaration du pointeur entier **ptr** et son initialisation à 0 (NULL). Donc, **ptr** ne pointe sur aucun entier en mémoire,*
— *la déclaration de la variable **c** de type caractère,*
— *la déclaration du pointeur caractère **pc** et son initialisation à l'adresse de **c** en mémoire. Donc, **pc** pointe initialement sur **c**.*

```
int *ptr(0); // ptr : pointeur de type int initialisé à 0 (NULL)
char c;
char *pc(&c); // pc : pointeur sur la variable c
```

1. *adresse* constitue l'adresse d'un objet informatique de type *type*

2.2.2 *Opérateurs associés aux pointeurs*

Les pointeurs manipulent deux opérateurs particuliers : **&** et *.

A. **L'opérateur unaire & :** L'opérateur **&** retourne l'adresse mémoire d'une variable, telle qu'illustré dans la figure 8.

Définition 2.2.6 (opérateur &) *Soit la variable **x** de type **T**, la combinaison de l'opérateur & suivi de x (&x) sera de type T∗. En d'autre termes, &x fournit l'adresse de la variable x en mémoire. .*

FIGURE 8 – Opérateur &(le pointeur *px* pointe sur la variable *x*)

Exemple 2.2.7 (manipulation des pointeurs) *Les instructions ci-dessous :*

```
int x(3); // x est un entier initialisée à 3
int* px; // px est un pointeur qui pointe sur un entier
px = &x; // px reçoit l'adresse mémoire de la variable x
```

représentent respectivement :

- *la déclaration de la variable x de type entier et son initialisation à* 3,
- *la déclaration de la variable pointeur px qui ne peut pointer que sur un entier,*
- *le chargement de l'adresse de x dans px.*

B. **L'opérateur unaire d'indirection « * » :** Tel qu'illustré dans la figure 9, l'opérateur * retourne la valeur pointée par une variable pointeur.

Définition 2.2.8 (opérateur *) *Soit la variable pointeur **px** de type **T***. La combinaison de l'opérateur * suivi de la variable pointeur **px** : ***px**, fournit la valeur de type **T** ayant l'adresse mémoire **px**.*

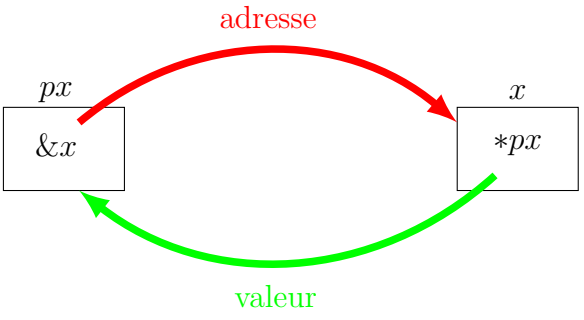


FIGURE 9 – Opérateurs & et ∗

Exemple 2.2.9 (manipulation des pointeurs) *Soit le bloc d'instructions ci-dessous :*

```
int x(3); //x : variable est de type entier initialisée à 3
int* px; //px : variable pointeur int* (qui pointe sur un entier)
px = &x; //px reçoit l'adresse mémoire de la variable x
cout << *px << endl; // affiche la valeur pointée par px
```

La dernière instruction affiche le contenu de la case mémoire ayant l'adresse px. En d'autres termes, elle affiche le contenu de la variable x

Nous constatons que l'expression $*\&i$ est strictement équivalente à i.

Exemple 2.2.10 (manipulation des pointeurs) *Soit le programme* C++ *ci-dessous :*

```
int main()
{
  int i=99;
  int *ptr;
  ptr=&i;
  return 0;
}
```

Dans la figure 10, nous illustrons l'impact de chaque instruction sur la variable manipulée :

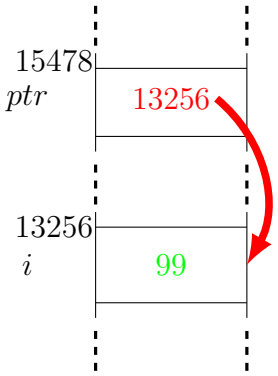


FIGURE 10 – Exemple de manipulation des pointeurs

La figure ci-dessus 10 montre l'état de la mémoire après exécution de chaque instruction du programme :

- *les adresses mémoire respectives des variables* ***i*** *et* ***ptr*** *sont* ***13256*** *et* ***15478****,*
- *la première instruction définit la variable* ***i*** *et l'initialise à 99,*
- *la troisième instruction affecte l'adresse mémoire de* ***i*** *(la valeur 13256) à la variable* ***ptr****.*

Trois points importants à noter sur les pointeurs et l'allocation de la mémoire :

1. la définition (ou la déclaration) d'un pointeur n'alloue en mémoire que l'espace mémoire nécessaire pour stocker ce pointeur et non pas ce qu'il pointe !
2. il est préférable d'initialiser (généralement à *NULL*) le pointeur avant de l'utiliser,
3. un pointeur sur un type "void" pointe une zone sans type (exemple : void *pt).

2.3 ALLOCATION DE LA MÉMOIRE

En C++ comme en C, on distingue deux types d'allocation de la mémoire :

1. allocation statique : la réservation de la mémoire est réalisée à la compilation.
2. allocation dynamique : la réservation de la mémoire se fait pendant l'exécution du programme.

2.3.1 *Allocation dynamique*

L'allocation dynamique permet de réserver de la mémoire indépendamment de toute variable. En d'autres termes, une variable pointeur pointe directement sur une zone mémoire plutôt que sur une variable existante.

La syntaxe de l'allocation dynamique porte sur deux mots clés **new** et **delete** qui permettent respectivement d'allouer et de libérer dynamiquement de la mémoire. Ci-dessous, les principes essentiels de l'allocation dynamique :

1. Afin de réserver une zone mémoire de type **type** et mettre son adresse dans la variable **pointeur**, nous utilisons la syntaxe suivante :

```
pointeur = new type;
```

   Il est également possible d'initialiser l'élément pointé directement lors de son allocation, comme suit :

```
pointeur = new type(valeur);
```

2. Afin de libérer l'espace mémoire dont l'adresse est conservée dans la variable **pointeur**, nous utilisons la syntaxe suivante :

```
delete pointeur;
```

   Après la libération de l'espace mémoire pointé par **pointeur**, cette zone mémoire peut être exploitée par d'autres processus.

Exemple 2.3.1 (allocation et libération dynamique de la mémoire) *Les trois blocs d'instructions ci-dessous sont équivalents :*

Bloc 1 :

```
int*px(0);
px = new int;
*px = 20;
cout << *px << endl;
delete px;
px = 0;
```

Le bloc 1*, manipule une seule variable pointeur nommée* px*. Les six instructions permettent les actions respectives notées ci-dessous :*

- *déclarer la variable* px *et l'initialiser à* 0*,*
- *allouer un espace mémoire pour un entier avec « **new** int »et charger l'adresse de l'emplacement mémoire de l'espace réservé dans* px*,*
- *charger la valeur* 20 *dans l'espace mémoire réservé,*
- *afficher la valeur contenu dans la case mémoire réservée,*
- *libérer l'espace mémoire alloué par la deuxième instruction,*
- *affecter la valeur* 0 *à* px [2]*. Ce qui désigne que* px *ne pointe sur aucune case mémoire.*

Bloc 2 :

```
int* px(0);
px = new int(20);
cout << *px << endl;
delete px;
px = 0;
```

Les instructions 2 *et* 3 *du bloc* 1*, sont exprimées via une seule instruction « **px = new int(20)** »dans le bloc* 2*.*

Bloc 3 :

```
int* px = new int(20);
cout << *px << endl;
delete px;
px = 0;
```

Les instructions 1 *et* 2 *du bloc* 2*, sont remplacées par une seule instruction « **int* px = new int(20)** »dans le bloc* 3*. La variable* px *est initialisée directement par l'adresse de l'espace mémoire alloué.*

2.3.2 *Pointeurs et tableaux*

Dans la section 1.4.1, nous avons donné le principe des tableaux statiques. En C++, la déclaration d'un vecteur suit la syntaxe suivante :

2. Après la libération de l'espace mémoire, le pointeur *px* contient toujours l'adresse manipulée auparavant. L'accès à cet espace mémoire s'avère impossible.

```
T NomTab[taille];
```

Avec :

NomTab : le nom du tableau,

T : le type des valeurs que peut contenir *NomTab*,

taille : la taille du tableau qui est le nombre de valeurs que peut contenir *NomTab*.

Dans un tableau statique, la taille doit être constante et elle est fixée au cours de la compilation.

Pour accéder à un composant du tableau, nous avons deux possibilités, à savoir : l'accès via l'indice et l'accès via l'adresse.

A. **Accès via l'indice :** Pour accéder à un élément d'indice *i* du tableau *NomTab*, nous utilisons la syntaxe *NomTab[i]*. Ainsi, le terme *NomTab[i]* sera exploité par la suite comme une variable simple de type *T*.

Contrairement au langage algorithmique, en C++ les indices sont toujours de type **entier positif**, ils varient de 0 à $taille - 1$ (*taille* est la taille de **NomTab**).

Exemple 2.3.2 (manipulation d'un tableau via les indices) *Soit le programme écrit en C++ ci-dessous :*

```
main()
{
  int vect[5];
  for(int i(0);i<5;i++){
    cout<< "vect[" << i << "]=";
    cin >> vect[i];
  }
  for(int i(0);i<5;i++)
    cout<< "vect[" << i << "]=" << vect[i] << endl;
  vect[3]=-1;
}
```

Les instructions du programme expriment les actions respectives ci-dessous :

1. *la première instruction permet de créer le tableau d'entiers* ***vect****, dont la taille est de* 5 *(voir la figure 11),*

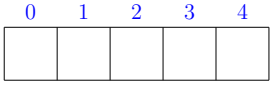


FIGURE 11 – Le tableau *vect*

2. *la première boucle saisit* 5 *composants de* ***vect****,*
3. *la deuxième boucle affiche les* 5 *composants de* ***vect****,*
4. *la dernière instruction affecte la valeur « -1 »au composant d'indice i.*

B. **Accès via l'adresse :**
En C++, le nom d'un tableau *NomTab* dans le cas statique, joue le rôle d'un pointeur qui contient initialement l'adresse du premier élément ($NomTab\,[0]$). Effectivement, le nom du tableau peut être utilisé comme un pointeur sans qu'il soit un pointeur pour autant.
En utilisant l'opérateur « * », l'accès aux différents éléments du tableau *NomTab* suit le raisonnement suivant :
$*NomTab$: fournit le premier élément du tableau,
$*(NomTab+1)$: fournit le deuxième élément du tableau,
$*(NomTab+2)$: fournit le troisième élément du tableau,
$*(NomTab+i)$: fournit le $(i+1)^{ème}$ élément du tableau.

Exemple 2.3.3 (manipulation d'un tableau avec les pointeurs) *Soit le programme écrit en* C++, *ci-dessous :*

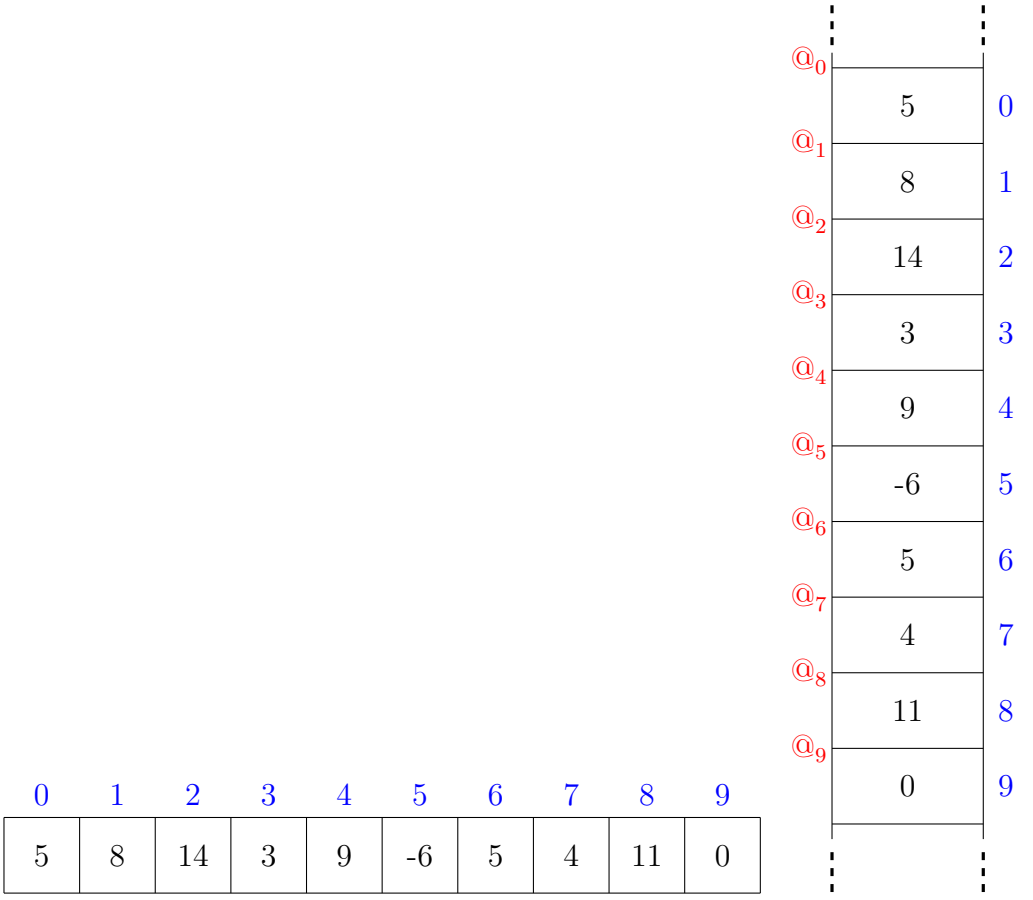
```
main()
{
  int tab[10]={5,8,14,3,9,-6,5,4,11,0};
  cout << *tab; // affiche la valeur 5
  cout << *(tab+1); // affiche la valeur 8
  cout << *(tab+4); // affiche la valeur 9
}
```

Le tableau ***tab*** *est un tableau d'entiers de taille* 10. *Il est chargé initialement de* 10 *valeurs, comme le montre la figure 12.*
La figure 12a illustre le tableau ***tab****, les valeurs en noir représentent les valeurs chargées dans le tableau via la première instruction du programme.*
La figure 12b montre la position du tableau en mémoire. Les valeurs en rouge (les $@_i$*) donne les adresses en mémoire des éléments du tableau. La valeur* $@_i$ *est l'adresse de la case d'indice i. Les valeurs en bleu représentent les indices du tableau qui varient entre* 0 *et* 9. *Nous remarquons que les composants du tableau sont consécutifs au niveau de la mémoire, du premier au dixième élément.*
Les instructions du programme expriment les actions respectives ci-dessous :

1. *créer le tableau d'entiers* ***tab****, dont la taille est de* 10, *et charger les* 10 *valeurs* $\{5,8,14,3,9,-6,5,4,11,0\}$ *(voir la figure 12a),*
2. *affiche la première valeur du tableau «* 5 *»,*
3. *affiche la deuxième valeur «* 8 *»,*
4. *affiche la cinquième valeur «* 9 *».*

La relation entre les adresses des éléments du tableau ***tab****, s'expriment comme suit :*

$$@_1 = @_0 + 1$$
$$@_2 = @_1 + 1$$
$$@_3 = @_2 + 1$$
$$@_4 = @_3 + 1$$
$$\dots$$
$$@_9 = @_8 + 1$$



(a) La structure du tableau *tab*

(b) Le tableau *tab* en mémoire

FIGURE 12 – Le tableau *tab* et son illustration en mémoire

2.3.3 *Arithmétique des pointeurs*

L'accès aux éléments d'un tableau peut se faire via un pointeur ayant un type compatible avec le contenu du tableau, tout en l'initialisant avec le nom du tableau. Dans ce cas la variable pointeur va pointer sur le premier élément. Ainsi, nous pouvons s'en servir d'une manière similaire que le tableau lui même. Cette variable pointeur est appelée *pointeur référé*. En d'autres termes, si nous déclarons le tableau *NomTab* et le pointeur référé *pr* comme suit :

```
T NomTab[taille];
T *pr=NomTab; // le pointeur référé pr
                                  // pointe sur le premier
                                  // élément de NomTab
```

l'instruction « pr=NomTab »est équivalente à l'instruction « pr=& NomTab[0] ».

Donc, pour accéder au $i^{ème}$ élément de *NomTab*, en utilisant le la variable *pr*, nous avons deux possibilités :

1. *via l'indice* : $pr[i]$
2. *via l'adresse* : $*(pr + i - 1)$

Exemple 2.3.4 (manipulation de l'opérateur référé sur les tableau) *Considérons le programme cité dans l'exemple précédent 2.3.3.*

Les deux blocs d'instructions ci-dessous utilisent le pointeur référé ***ptab*** *sur le tableau* ***tab****.*

Bloc 1 :

```
int* ptab;
ptab = tab;
cout << ptab[0] << endl;
cout << ptab[9] << endl;
```

Bloc 2 :

```
int* ptab;
ptab = tab;
cout << *ptab << endl;
cout << *(ptab+9) << endl;
```

La deuxième instruction peut être remplacée par : « ***ptab=& tab[o]*** *»*
Ainsi, ***ptab*** *pointe sur le premier élément du tableau comme le montre la figure 13.*

2.3.4 *Tableaux et allocation dynamique*

Quand on fait une allocation dynamique de la mémoire, on obtient en retour un pointeur sur la zone mémoire allouée (ou la première case de la zone mémoire allouée).

A. **Allocation dynamique d'un tableau :** L'allocation de l'espace mémoire pour un tableau de taille « taille »dont les éléments de type « T »et dont l'adresse du premier élément est chargée dans « PTab », se donne comme suit :

```
T * PTab= new T[taille];
```

Ce qui donne en mémoire la variable *PTab*[3] qui pointe sur le premier élément du tableau dont les valeurs sont de type *T* (voir la figure 14).

Dans la figure 14, les valeurs en bleu représentent les indices des éléments du tableau et les valeurs en rouge leurs adresses.

B. **Accès aux éléments d'un tableau dynamique :** Comme le montre la figure 14, la disposition des composants d'un tableau dynamique est similaire à celle d'un tableau statique. Ainsi, l'accès aux différents éléments suit la même syntaxe que celle des tableaux statiques.

3. Il ne faut pas modifier le pointeur *PTab*, car il est nécessaire pour tout traitement à réaliser sur le tableau

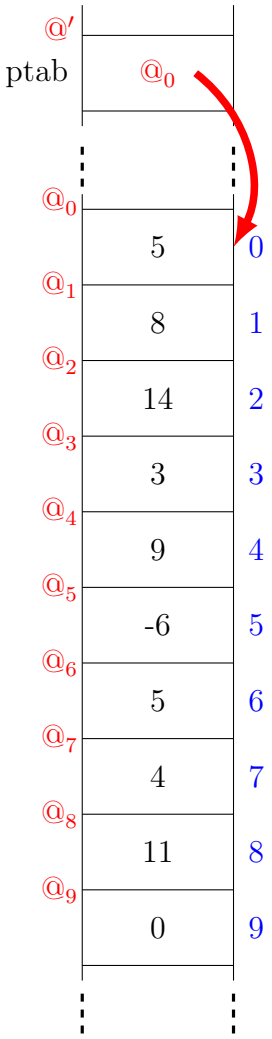


FIGURE 13 – Utilisation du pointeur référé *ptab* sur le tableau *tab*

c. **Libération de la mémoire :** La libération de l'espace mémoire d'un tableau dynamique se donne comme suit :

```
delete[]PTab;
```

Avec « PTab », le nom du tableau dynamique (ou le pointeur qui comporte la première adresse du tableau).

Ce qui permet aux processus étrangers à notre programme d'exploiter cet espace mémoire.

Exemple 2.3.5 (allocation dynamique d'un tableau d'entiers) *Soit le bout de code* C++ *ci-dessous :*

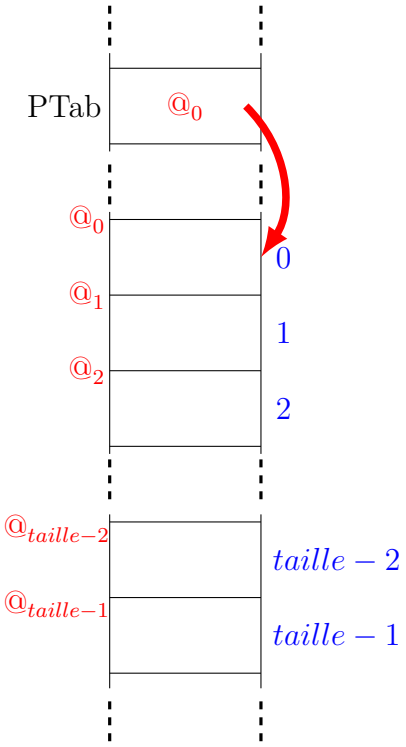


FIGURE 14 – Allocation dynamique d'un tableau en mémoire

```
int * p= new int[10];
if (p) {
        p[0]=5;
        cout << p[0] << endl;
        *(p+1)=-2;
        cout << *(p+1) << endl;
        delete[]p;
}
```

Les instruction du programme expriment respectivement les actions ci-dessous :

1. *créer un tableau d'entiers de taille* 10,
2. *vérifier si l'allocation a été réalisée avec succès,*
3. *mettre la valeur* 5 *dans la première case,*
4. *afficher la première valeur,*
5. *mettre la valeur* -2 *dans la deuxième case,*
6. *afficher la deuxième valeur,*
7. *libérer l'espace mémoire alloué.*

2.4 PORTÉE DES FONCTIONS ET DES VARIABLES

2.4.1 *Passage des arguments*

Dans la section 1.5, nous avons introduit la notion de procédure et de fonction en algorithmique. En C++, nous ne pouvons mettre en œuvre que les fonctions. Sachant que les fonctions n'admettent que les paramètres d'entrée, nous avons les types de passage des arguments (ou paramètres), et ils sont trois :

1. passage par défaut,
2. passage par valeur,
3. passage par référence.

A. **Passage par défaut :** Une fonction peut donner des valeurs par défaut à ses arguments lors de son prototypage[4]. Ceux pour lesquels on ne doit pas nécessairement fournir des valeurs lors de l'appel de la fonction.

Exemple 2.4.1 (fonction avec passage par défaut des arguments) *Soit le prototype suivant :*

```
void f(int i, char c='a', double x=0.0);
```

Les paramètres c et x sont de passage par défaut, leurs valeurs sont fixées au cours du prototypage de la fonction f. Ces valeurs sont manipulées par le programme appelant, avec possibilité de les changer au cours de l'appel.
Soient les différents appels de f ci-dessous :

```
f(1); // correct et vaut f(1, 'a', 0.0)
f(1,'b'); // correct et vaut  f(1, 'b', 0.0)
f(1,3.0); // incorrect
f(1, , 3.0); // incorrect
f(1,'b',3.0); // correct et vaut  f(1, 'b', 3.0)
```

Sachant que l'appel d'une fonction doit être conforme à son entête (ou son prototype), nous commentons ces appels comme suit :

$f(1)$: *appel correct, il est similaire à « f(1, 'a', 0.0) »,*

$f(1,'b')$: *appel correct, il est similaire à « f(1, 'b', 0.0) »,*

$f(1,3.0)$: *appel syntaxiquement incorrect. Le deuxième paramètre est un* ***float*** *dans l'appel, alors que son type est défini* ***char*** *dans le prototype.*

$f(1,,'3.0')$: *appel syntaxiquement incorrect. Si l'objectif est de manipuler le troisième paramètre avec la valeur* 3.0 *au lieu de* 0.0 *et de laisser le deuxième paramètre inchangée, avec sa valeur par défaut b, alors l'appel doit se faire conformément au prototype,*

4. Le prototype d'une fonction est la structure de l'en-tête, à savoir le type de retour, le nom de la fonction et la suite des paramètres(ou arguments) et leurs types

$f(1,`b',3.0)$: *appel syntaxiquement correct. Il peut donner une correction à l'instruction qui vient tout juste avant.*

B. **Passage par valeur :** La variable locale associée à un argument passé par valeur correspond à une copie de l'argument. Par conséquent, les modifications effectuées à l'intérieur de la fonction ne se répercutent pas sur l'environnement extérieur à la fonction.

Exemple 2.4.2 (passage par valeur) *Nous donnons, ci-dessous, le code* C++ *qui contient :*

— *la fonction* ***permut*** *ayant le prototype* ***void permut(int, int)****. Elle est conçue pour faire la permutation de deux entiers,*

— *le programme principal* ***main()****, qui fait appel à* ***permut****.*

```
void permut(int a, int b)
{
  int c(a); a=b; b=c;
}
main()
{
  int i(10), j(55);
  permut(i,j);
  cout << "Après permutation : i=" << i << "et j=" << j << endl;
}
```

Après exécution de notre programme et l'appel de permut, i vaut toujours 10 et j toujours 55.

La figure 15, illustre le résultat de chacune des instructions en mémoire.

La figure 15a montre la réservation de la mémoire pour les variables i et j, et le chargement de leurs valeurs initiales respectives 10 *et* 55.

A l'appel de la fonction (voir la figure 15b), une copie de chacune des variables i et j est créée. Une allocation de la mémoire est réalisée pour la variable c déclarée dans l'environnement locale de la fonction. Cette variable est initialisée avec la valeur de la copie de i. Nous rapportons les variables qui figurent dans la mémoire à l'appel de la fonction. Elles sont 5 *:* ***i****=10,* ***j****=55,* ***copie de i****=10(notée aussi a),* ***copie de j****=55(notée aussi b) et* ***c****=10.*

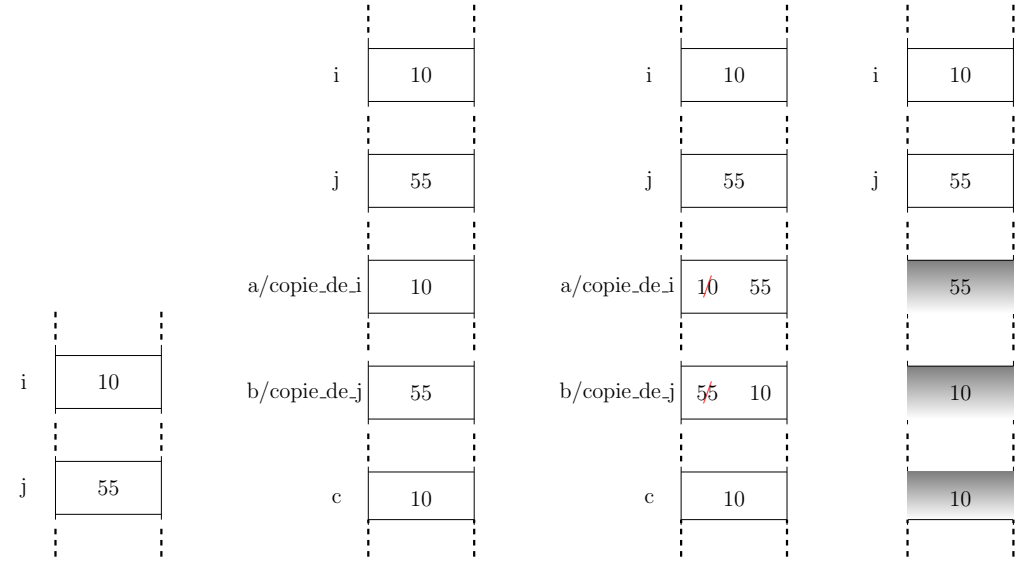
Nous constatons à travers la figure 15c, que la permutation se fait entre les copies respectives de i et j. Ce qui donne en mémoire : ***i****=10,* ***j****=55,* ***copie de i****=55 et* ***copie de j****=10.*

Après exécution de la fonction, on sort avec les variables i et j non permutées. Les parties en gris dans la figure 15d représentent les cases mémoires libérées.

Par conséquent, la permutation a été réalisée sur les copies des arguments qui restent inchangés.

C. **Passage par référence :** Pour parler du passage par référence, on doit passer par définir et décrire le type référence [5] et son utilisation en C++.

5. Le type référence est propre au langage C++ et il n'est pas manipulé en C



(a) Avant l'appel de *permut* (b) À l'appel de *permut* (c) Exécution de *permut* (d) Après l'appel de *permut*

FIGURE 15 – La fonction « permut » avec passage par valeur

Définition 2.4.3 (type référence) *Une référence est un synonyme d'identificateur. La déclaration d'une référence se fait selon la syntaxe suivante :*

```
T & NomRef(identificateur);
```

Après une telle déclaration, NomRef peut être utilisé partout où identificateur peut l'être.

Comme illustré dans la figure 16[Sam-Haroud, 2012], la référence pareillement au pointeur, permet de désigner un objet indirectement (à travers son adresse).

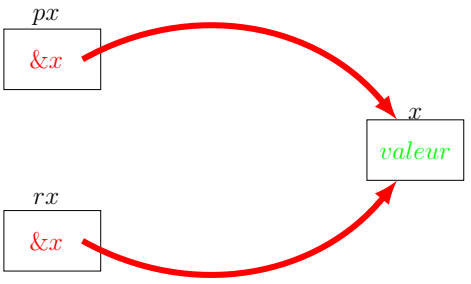


FIGURE 16 – Manipulation d'un pointeur-référence

Exemple 2.4.4 (pointeur-référence) *Soient les trois instructions :*

```
int i(2);
int * pi;
pi=&i;
int & ri(i);
```

La deuxième et la quatrième instructions permettent respectivement :

- *la déclaration du pointeur d'entier, nommé pi. Dans l'instruction qui suit, pi est affectée de l'adresse de i.*
- *la déclaration de la variable référence de i.*

Deux points essentiels à constater via cet exemple :

1. *pi pointe après exécution de l'instruction « pi=&i » sur l'entier i. Il peut être utilisé pour pointer sur un autre entier.*
2. *ri ne peut référencer que la variable i.*

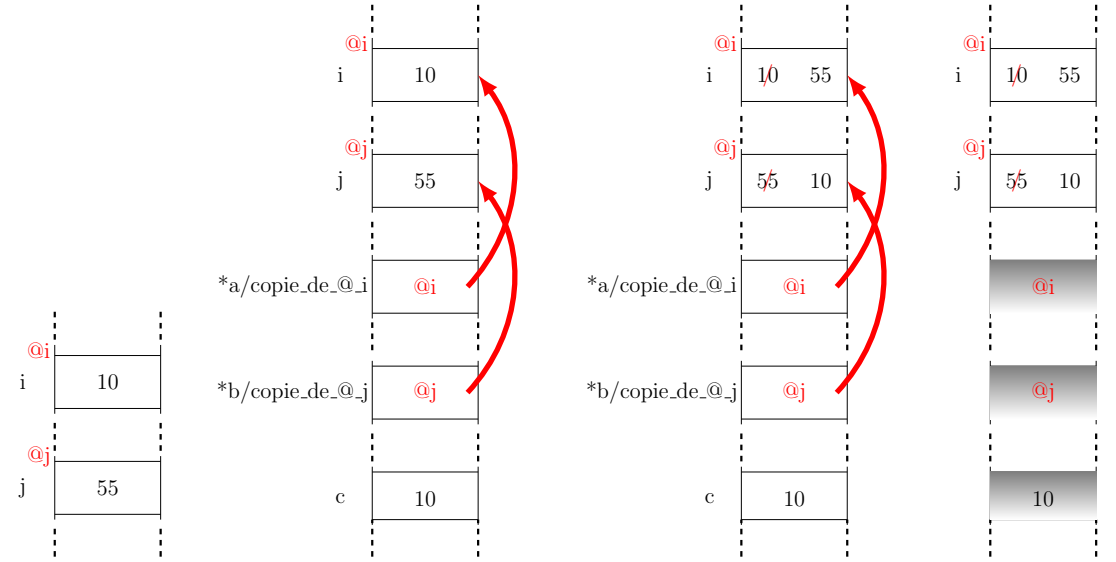
La différence entre une référence et un pointeur est le fait qu'une référence :
— doit absolument être initialisée (sauf si c'est un argument de la fonction),
— ne peut être liée qu'à un seul objet,
— ne peut pas être initialisée à zéro,
— ne peut pas référencer une autre référence.
Revenons maintenant au passage par référence des paramètres d'une fonction. On opte généralement pour ce type de passage dans le cas où la fonction doit retourner plusieurs résultats. On distingue deux solutions :

1. **Solution du C :**
   Dans le langage C, on parle de passage d'arguments sous forme de pointeurs. Donc, au cours de l'appel de la fonction, nous manipulons les adresses des variables au lieu des copies des variables. Puisque le pointeur et sa copie pointent sur la même case mémoire, l'inconvénient du passage par valeur est résolu. Nous illustrons la situation correspondant, via l'exemple ci-dessous :

   Exemple 2.4.5 (passage par adresse) *L'application des pointeurs sur les paramètres de la fonction* ***permut****, donnée précédemment, donne le code suivant :*

```
void permut(int * a, int * b)
{
  int c(*a); *a=*b; *b=c;
}
main()
{
  int i(10), j(55);
  permut(&i,&j);
  cout << "Après permutation : i=" << i << "et j=" << j << endl;
}
```

   La figure 17, illustre le résultat de chacune des instructions en mémoire.



(a) Avant l'appel de *permut* (b) À l'appel de *permut* (c) Exécution de *permut* (d) Après l'appel de *permut*

FIGURE 17 – La fonction« permut » avec passage par adresse

La figure 17a montre la réservation de la mémoire pour les variables i et j, et le chargement de leurs valeurs initiales respectives 10 *et* 55.

A l'appel de la fonction (voir la figure 17b), une copie d'adresse de chacune des variables i et j est créée. Une allocation de la mémoire est réalisée pour la variable c déclarée dans l'environnement locale de la fonction. Cette variable est initialisée avec la valeur pointée par la variable copie_@i qui contient l'adresse de i. Les variables qui figurent dans la mémoire à l'appel de la fonction sont 5 *:* ***i****=10,* ***j****=55,* ***copie de l'adresse de i****=10(notée aussi copie_@i),* ***copie de l'adresse de j****=55 (notée aussi copie_@j) et* ***c****=10.*

Nous constatons à travers la figure 17c, que la permutation se fait entre i et j en manipulant leurs adresses. Ce qui donne en mémoire : ***i****=10,* ***j****=55,* ***copie de l'adresse de i****=adresse de i et* ***copie de l'adresse j****=adresse de j.*

Après exécution de la fonction, on sort avec les variables effectivement permutées. Les parties en gris dans la figure 15d représentent les cases mémoires libérées.

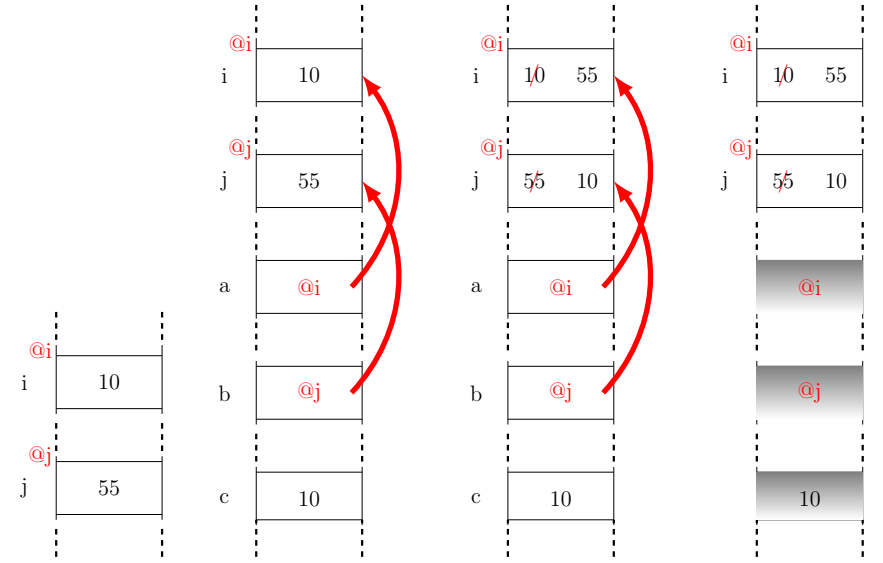
Nous constatons qu'après l'appel de ***permut****,* $i = 55$ *et* $j = 10$*. La permutation des deux valeurs a bien été réalisée.*

2. **Solution du** C++ **:**
   Dans le langage C++, on parle de passage d'arguments par référence.

   Exemple 2.4.6 (passage par référence) *L'application du passage par référence sur les paramètres de la fonction* ***permut****, donnée précédemment avec des paramètres de passage par valeur, donne le code source suivant :*

```
void permut(int & a, int & b)
{
  int c(a); a=b; b=c;
}
main()
{
  int i(10), j(55);
  permut(i,j);
  cout << "Après permutation i=" << i << "et j=" << j << endl;
}
```

La figure 18, illustre le résultat de chacune des instructions en mémoire.



(a) Avant l'appel de *permut* (b) À l'appel de *permut* (c) Exécution de *permut* (d) Après l'appel de *permut*

FIGURE 18 – La fonction « permut » avec passage par référence

La figure 18a montre la réservation de la mémoire pour les variables i et j, et le chargement de leurs valeurs initiales respectives 10 *et* 55.

A l'appel de la fonction (voir la figure 18b), les références a et b respectives de i et j sont créées et chargées. Une allocation de la mémoire est réalisée pour la variable c déclarée dans l'environnement locale de la fonction. Cette variable est initialisée avec la valeur de la variable i référencée par a. Nous montrons les variables qui figurent en mémoire à l'appel de la fonction : ***i****=10,* ***j****=55,* ***a****=@i,* ***b****=_@j et* ***c****=10.*

Nous constatons à travers la figure 18c que la permutation se fait entre i et j en manipulant leurs références respectives a et b. Ce qui donne en mémoire : ***i****=10,* ***j****=55,* ***a****=@$_i$ et* ***b****=@j.*

Après exécution de la fonction, on sort avec les variables i et j permutées. Les parties en gris dans la figure 18d représentent les cases mémoires libérées.

2.4.2 *Portée des variables*

Les blocs d'instructions en C++ ont une grande autonomie, que ceux soient des corps de fonction ou non. Un bloc d'instructions peut contenir son propre environnement de travail.

Comme on peut le constater dans le bout code C++ ci-dessous, la variable y est déclarée à l'intérieur du bloc d'instruction du if, par conséquent elle ne sera plus reconnue en dehors du bloc en question. Dans ce cas, on dit que la variable y est locale au bloc.

```
if (x != 0.0) {
   double y(0.0);
     ...
   y = 5.0 / x;
     ...
}
// ici on ne peut plus utiliser y
```

2.4.3 *Variable globale/locale*

Le principe de la globalité et de la localité en programmation d'une manière générale est comme suit :

— Les variables déclarées à l'intérieur d'un bloc sont appelées variables locales (au bloc). Elles ne sont accessibles qu'à l'intérieur du bloc.

— Les variables déclarées en dehors de tout bloc (même du bloc *main*) seront appelées variables globales. Elles sont accessibles dans l'ensemble du programme.

Nous conseillons aux programmeurs de ne jamais utiliser à tort et à travers la notion de la globalité des variables dans leurs programmes.

Définition 2.4.7 (portée d'une variable) *La portée d'une variable est délimitée par l'ensemble des lignes de code où cette variable est accessible, définie, existe et a un sens.*

2.4.4 *Règles de résolution de portée*

En cas d'ambiguïté dans le nom ; plusieurs variables (de portées différentes) portant le même nom, la règle utilisée en C++ est la suivante :

— L'objet interne (i.e. « le plus proche ») est systématiquement choisi. On parle de **portée locale**.

— Les arguments d'une fonction sont de *portée locale par rapport au corps de la fonction*.

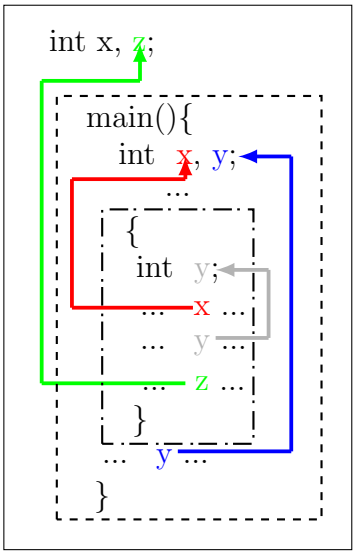
Il est généralement préférable d'éviter l'utilisation du même nom de l'objet informatique plusieurs fois (sauf peut être pour des variables locales aux boucles : i, j, etc.)

Exemple 2.4.8 (portée des variables) *Dans la figure 19, nous introduisons la portée des variables locales et globales en* C++.

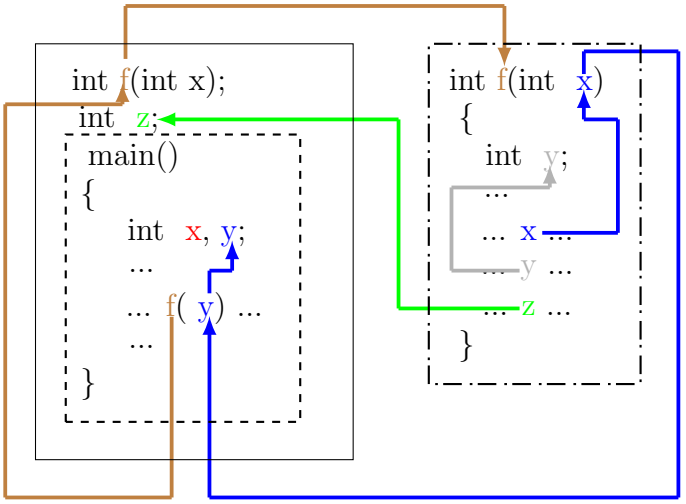
Les figures 19a et 19b[Sam-Haroud, 2012] représentent respectivement un programme avec blocs d'instructions et un programme avec appel d'une fonction.

- *Les instructions encadrées avec la ligne « — · — » est le corps de la fonction f qui a comme paramètre l'entier x et variable locale l'entier y. Ce bloc d'instructions peut être liée à une alternative, une boucle, etc.*
- *La partie encadrée avec la ligne « —— » est le programme principal, qui manipule deux variables locales x et y de type entier. Il fait appel à la fonction f.*

le programme final est encadré avec une ligne continue. Il manipule deux objets informatiques globaux : la fonction f et l'entier z.



(a) Portée des variables-programme avec le *main()*



(b) Portée des variables-programme avec appel d'une fonction

FIGURE 19 – Portée des variables

Nous décrivons via la figure 19a la portée de chacune des variables :

· *la variable y (de couleur gris) de portée locale au bloc n'est manipulée qu'à l'intérieur.*

- *les variables x et y (de couleurs respectives rouge et bleu) de portée locale au « main » ne sont manipulées que par les instructions du main.*
- *les variables x et z sont de portée globale. Elles peuvent être manipulées par toutes les instructions du programme.*

Nous illustrons via la figure 19b la portée des objets informatiques (variables et fonctions). Nous avons la fonction f à droite et le programme appelant à gauche. La portée des variables et de la fonction se décrit comme suit :

- *les variables x et y (de couleurs respectives rouge et bleu) sont des variables de portée locale au main. Par conséquent elles ne sont manipulées qu'à travers les instructions du main, telles que l'appel de la fonction f qui prend y comme paramètre effectif.*
- *la variable z et la fonction f (de couleurs respectives vert et marron) sont des objets de portée globale. Les deux objets peuvent être utilisés par toutes les parties qui composent le programme (le main et la fonction f).*

3

RÉCURSIVITÉ

3.1 INTRODUCTION

Dans les chapitres précédents nous avons parlé des boucles et leurs différents types. L'itérativité en algorithmique est un concept qui permet la mise en œuvre d'un bloc d'instructions qui doit s'exécuter d'une manière itérée.

Dans certains cas, la définition du problème à résoudre se décrit par récurrence, tels que les fonctions récursives en mathématiques. Par conséquent, la récursivité en algorithmique permet de résoudre certains problèmes d'une manière très simple et directe, alors que la résolution itérative, nécessite beaucoup plus de travail et de structures de données intermédiaires.

Dans ce chapitre nous introduisons le principe de la récursivité, le passage réciproque récursive-itérative, et une comparaison entre les deux versions. Des figures illustratives ont été intégrées à chaque étape.

Ce chapitre comporte quatre sections qui décrivent respectivement, la construction d'un algorithme récursif, l'environnement d'une fonction récursive, le fonctionnement de la récursivité, le passage réciproque récursivité-itérativité et quelques exemples d'application.

3.2 CONSTRUCTION D'UN ALGORITHME RÉCURSIF

L'algorithme récursif peut être défini comme suit :

Définition 3.2.1 (algorithme récursif) *Un algorithme récursif est un algorithme qui s'appelle lui même.*

Concevoir un algorithme récursif est un peu comme définir une suite par récurrence en mathématiques, il faudra :

1. Un ou plusieurs cas de base, dans lequel (lesquels) l'algorithme ne fait pas appel à lui même, sinon l'algorithme ne peut pas s'arrêter.
2. Un ou plusieurs cas inductif(s), dans lequel (lesquels) l'algorithme fait appel à lui même.

Chaque appel récursif doit, en principe, se rapprocher d'un cas de base, de façon à permettre la terminaison du programme.

Exemple 3.2.2 (calcul de la factorielle) *On rappelle que la factorielle d'un nombre n se définit comme suit :* $n! = 1 * 2 * 3... * n$.

La solution itérative qui calcule la factorielle de n s'exprime via la fonction « fact » comme suit :

La version itérative :

```
int fact(int n)
{
   if (n==0) return 1;
   else {
   int p=1;
   for (int i=1; i<n; i++)
     p=p*i ;
   return p;
 }
}
```

La version récursive :

L'écriture de n! sous forme d'une suite par récurrence, notée « fact » se donne comme suit :

$$\begin{cases} fact_0 = 1 \\ fact_n = n * fact_{(n-1)} \end{cases}$$

Ce qui se traduit, d'une manière directe, par la fonction récursive nommée ***int fact(int)*** *en C++ ci-dessous :*

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

L'arbre d'exécution de la fonction « fat » avec le paramètre effectif d'entrée « 3 », se donne comme suit :

Comme nous pouvons le constater via la figure 20, le parcours d'un arbre d'exécution suit un ordre postfixe pour calculer la valeur de « fact(3) ».

3.3 ENVIRONNEMENT D'UNE FONCTION RÉCURSIVE

3.3.1 *Environnement local*

Chaque appel récursif dispose de ses propres variables locales. Elles sont déclarées à chaque appel (voir l'exemple ci-dessous3.3.1), ceci est également vrai pour les paramètres locaux de la fonction.

Les variables de portée locale sont des variables propres à chaque appel récursif de la fonction.

Exemple 3.3.1 (fonction *cmb* **avec variables locales)** *Soit la fonction récursive cmb ci-dessous :*

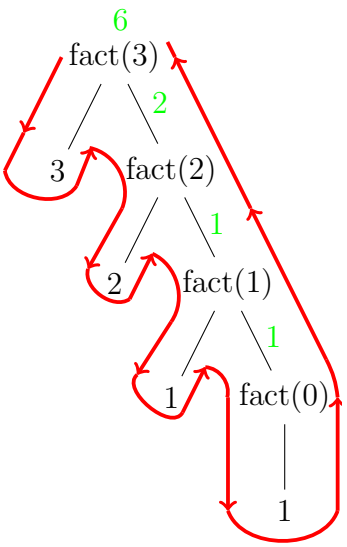


FIGURE 20 – Arbre d'exécution de $fact(3)$

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

Les variables a et b sont de portée locale, elles sont donc propres à chaque appel. En d'autres termes, un nouvel espace mémoire est réservé aux deux variables à chaque appel récursif de la fonction. Ce qui donne l'arbre d'exécution illustré dans la figure 21.

3.3.2 *Environnement global*

Dans le cas où la variable est partagée par tous les appels récursifs d'une fonction, on parle de variables globales. Ces variables doivent être déclarées à l'extérieur de la fonction.

Exemple 3.3.2 (fonction *cmb* avec variables globales) *Soit la fonction cmb citée dans l'exemple 3.3.1 avec les variables a et b déclarées globales, ce qui donne le code suivant :*

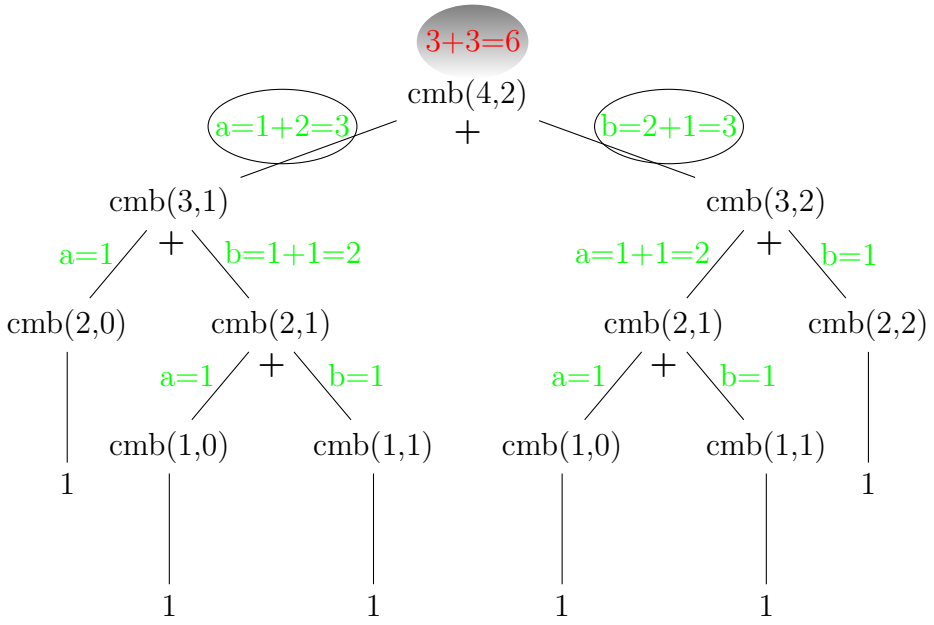


FIGURE 21 – Arbre d'exécution de $cmb(4,2)$ avec a et b variables locales

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

L'arbre d'exécution de $cmb(4,2)$ *est illustré dans la figure 22.*
Les valeurs successives affectées aux variables a et b se donnent comme suit :

$a = 1 \to 1 \to 3 \to 1 \to 2$

$b = 1 \to 2 \to 1 \to 1 \to 3$

ce qui donne en fin de compte la valeur de « cmb(4,2) » qui est égale à $a + b = 5$.

Nous constatons que la valeur de $cmb(4,2) = 5$ *avec a et b globales, alors que* $cmb(4,2) = 6$ *avec a et b locales (voir l'exemple 3.3.1).*

3.4 FONCTIONNEMENT DE LA RÉCURSIVITÉ

L'un des inconvénients majeurs de la récursivité est sa consommation en espace mémoire. Ceci est du à l'utilisation d'une pile d'exécution 3.4.1, une structure dont la mémoire réservée augmente à chaque nouvel appel récursif de la fonction.

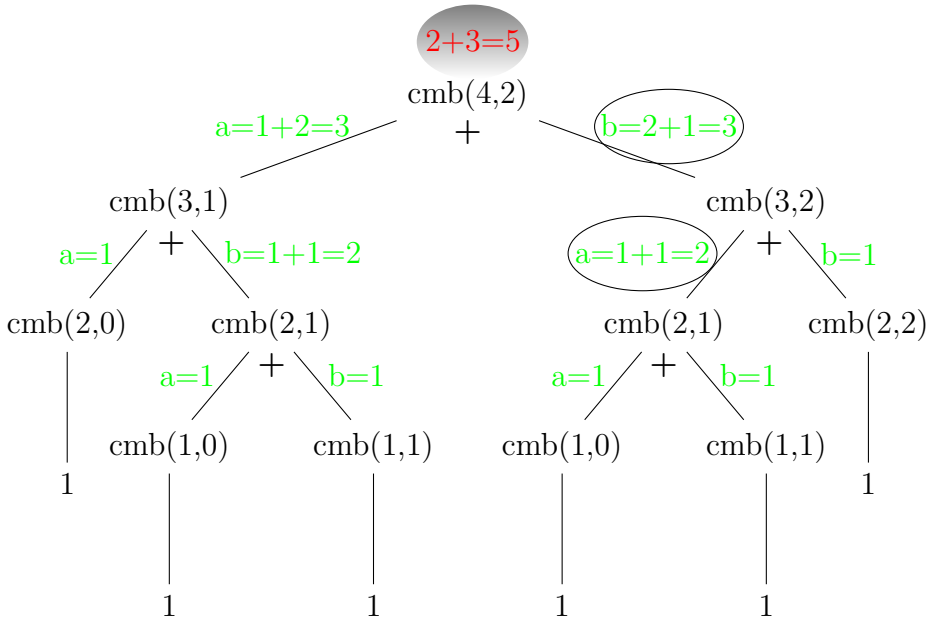


FIGURE 22 – Arbre d'exécution de $cmb(4,2)$ avec les variables a et b globales

Définition 3.4.1 (pile d'exécution) *La pile d'exécution du programme en cours est un emplacement mémoire destiner à mémoriser les paramètres, les variables locales ainsi que les adresses de retour des fonctions en cours d'exécution.*

Lors de l'exécution d'un programme, cette pile permet de :

1. mémoriser le contexte appelant lors de chaque appel de la fonction (adresses des variables, adresse de la prochaine instruction à exécuter en sortant de la fonction, etc.). A ce niveau, le nouveau contexte est empilé dans la pile d'exécution.
2. au retour, ou en atteignant le cas de base, le contexte est dépilé.

En d'autres termes, tant que le cas de base n'est pas atteint la taille de la pile d'exécution augmente.

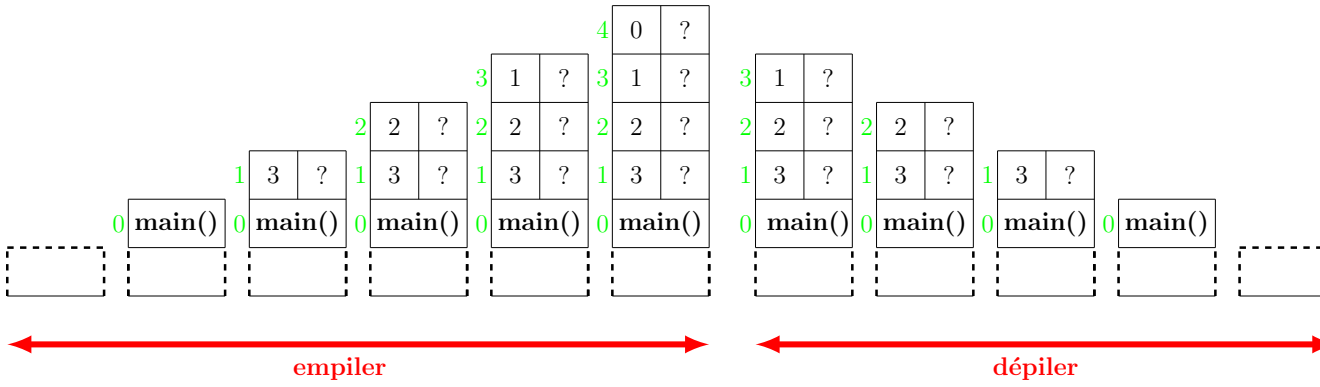
Exemple 3.4.2 (pile d'exécution de *fact(3)*) *Soit la fonction récursive « fact » définie précédemment. On donne ci-dessous la table d'exécution 6 de la fonction $fact(3)$, suivie de la pile d'exécution illustrée dans la figure 23.*

La table 6 [GREFFIER, 2007] est la table d'exécution s'apprêtant à l'exécution de « fact(3) ». Comme nous pouvons le constater à travers l'arbre d'exécution (voir la figure 20), nous avons quatre appels de la fonction « fact » : fact(3), fact(2), fact(1) puis fact(0). A chaque appel, le contexte correspondant est empilé, jusqu'à ce qu'on atteint le cas de base ($n = 0$), pour dépiler les contextes dans l'ordre inverse de l'empilement : fact(0), fact(1), fact(2) puis fact(3). L'opération de multiplication est réalisée à chaque fois que le contexte est dépilé.

TABLE 6 – Pile d'exécution de $fact(3)$

Num d'appel	Num Retour	Contexte	Action
1 (empiler)		n=3	n*fact(2)
2 (empiler)		n=2	n*fact(1)
3 (empiler)		n=1	n*fact(0)
4 (empiler)		n=0	return(1)
	(dépiler)→ 4	n=0	return(1)
	(dépiler)→ 3	n=1	return(1*1)
	(dépiler)→ 2	n=1	return(2*1)
	(dépiler)→ 1	n=1	return(3*2)
	(dépiler)	Pile Vide	Retour au point appelant

Les actions « empiler » 23a et « dépiler » 23b réalisées respectivement via l'appel récursif et le cas de base sont illustrées dans la figure 23.



(a) La pile d'exécution au cours d'empilement

(b) La pile d'exécution au cours du dépilement

FIGURE 23 – Pile d'exécution au cours d'exécution de $fact(3)$

3.5 LE PASSAGE ALGORITHME RÉCURSIF-ALGORITHME ITÉRATIF

L'inconvénient majeur des algorithmes récursifs est le problème d'encombrement mémoire. Par conséquent, on essaie de diminuer la taille de la pile d'exécution en utilisant des algorithmes itératifs.

Pour écrire un algorithme équivalent, on doit gérer dans l'algorithme la pile des sauvegardes (ne garder que celles utiles). En d'autres termes, « *Transformer un algorithme récursif en une version itérative équivalente dans le but de diminuer la taille de la pile d'exécution.* »

Malheureusement, il n'existe pas de règle générale pour passer d'une version récursive vers une version itérative et vice versa. Dans la littérature, nous ne trouvons que des cas particuliers de fonctions récursives, dont le passage suit une forme bien définie.

Nous introduisons dans cette section les cas suivants :

— un seul appel récursif terminal,
— un seul appel récursif non terminal,

Lorsque la fonction récursive comporte plus d'un appel récursif (deux ou plus), l'écriture de la version itérative devint très difficile à réaliser.

3.5.1 *Un seul appel récursif terminal*

Définition 3.5.1 (fonction récursive terminale) *Un algorithme est dit récursif terminal si aucun traitement n'est effectué à la remontée d'un appel récursif (sauf en cas de retour de la valeur).*

Généralement, une fonction récursive terminale suit la structure ci-dessous :

```
void fRec1(T x){
      if (c(x)) A0(x);
      else {
              A1(x);
              fRec1(F1(x));
             }
      }
```

La version itérative de $fRec1$ se donne directement via la fonction $fIter1$ comme suit :

```
void fIter1(T x)
{
      while(!c(x)){
       A1(x);
      }
      A0(x);
}
```

Exemple 3.5.2 (transformation de la fonction « enumerer ») *Soit la fonction récursive enumerer :*

```
void enumerer(int x)
{
      if (n<=0) cout << "fin";
      else {
              cout << n;;
              enumerer(n-1);
             }
}
```

La version itérative équivalente se donne comme suit :

```
void enumerer(int x){
      while(n>0) {
          cout << n;
          n=n-1;
        }
        cout << "fin";
      }
```

3.5.2 *Un seul appel récursif non-terminal*

Définition 3.5.3 (algorithme récursif non terminal) *Un algorithme est dit récursif non terminal si un traitement est effectué à la remontée d'un appel récursif.*

Généralement, une fonction récursive non-terminale suit la structure suivante :

```
T1 fRec2(T2 x){
      if (c(x)) return F(x);
      else return (u(G(x), fRec2(H(x))));
}
```

La version itérative et équivalente à $fRec2$ se donne par la fonction $fIter2$ comme suit :

```
T1 fIter2(T2 x){
     ... p;
     if (c(x)) return F(x);
      else {
                p=G(x); x=H(x);
                while(!c(x)) {
                    p=u(p,G(x));
                    x=H(x)
              }
           return u(p,F(x));
        }
}
```

Exemple 3.5.4 (transformtion de la fonction « fact ») *Soit la fonction récursive fact :*

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

La version itérative équivalente :

```
int fonct(int n){
      if (n==0) return 1;
      else {
                int p=n; n=n-1;
                while(n!=0) {
                    p=n*p;
                    n=n-1;
                }
            return p*n;
        }
    }
```

3.6 EXEMPLES D'APPLICATION

3.6.1 *Calcul du Plus Grand Commun Diviseur (PGCD)*

Soit la définition suivante :

```
PGCD(p, q) = si p = 0 alors q
             sinon si q = 0 alors p
                   sinon si q = p alors PGCD(p-q, q)
                         sinon PGCD(p, q-p)
```

Avant de travailler sur la solution algorithmique du calcul du *PGCD*, nous commençons par donner l'écriture mathématique correspondante :

$$PGCD : \mathbb{N}^2 \longrightarrow \mathbb{N}$$

$$(p,q) \longmapsto PGCD(p,q) = \begin{cases} q & si \quad p = 0 \\ p & si \quad q = 0 \\ PGCD(p-q,q) & si \quad q \leq p \\ PGCD(p,q-p) & si \quad q > p \end{cases}$$

L'ensemble de départ $\mathbb{N}^2$ exprime le nombre des paramètres d'entrée qui est de 2 et leur type qui est le *int* (ou *unsigned int*). L'ensemble d'arrivée $\mathbb{N}$ exprime le type de retour de la fonction *PGCD*. Ce qui donne le prototype : *int PGCD(int, int)* (ou l'entête *int PGCD(int p, int q)*).

Concernant le corps de la fonction, nous réalisons une traduction brut de la partie

$$PGCD(p,q) = \begin{cases} q & si \quad p = 0 \\ p & si \quad q = 0 \\ PGCD(p-q,q) & si \quad q \leq p \\ PGCD(p,q-p) & si \quad q > p \end{cases}$$

dans le langage algorithmique (en C++). Ce qui donne la fonction récursive *PGCD* suivante :

```
int PGCD(int p, int q)
{
        if  (p == 0) return q;
        else if (q == 0) return p;
                else if (q <= p)  return PGCD(p-q, q);
                        else return PGCD(p, q-p);
}
```

3.6.2 *La suite de Fibonacci*

La suite de *Fibonacci* se donne comme suit :

$$\begin{cases} Fibo_1 = Fibo_2 = 1 \\ Fibo_{n+2} = Fibo_{n+1} + Fibo_n \end{cases}$$

En passant par l'écriture de la suite sous forme d'une fonction mathématique nommée *Fibo* :

$$Fibo : \mathbb{N}^* \longrightarrow \mathbb{N}^*$$
$$n \longmapsto Fibo(n) = \begin{cases} 1 & \text{si n=1,2} \\ Fibo(n-1) + Fibo(n-2) & sinon \end{cases}$$

Les ensembles de départ et d'arrivée donne respectivement, des informations sur les paramètres d'entrées et le type de retour. Le prototype de la fonction correspond à *int Fibo(int)* (ou l'en-tête *int Fibo(int n)*).

La fonction récursive se donne en C++ comme suit :

```
int Fibo(int n) {
 if  (n==1 || n==2) return 1;
 else return Fibo(n-2)+Fibo(n-1);
}
```

4

COMPLEXITÉ ALGORITHMIQUE

4.1 INTRODUCTION

La complexité algorithmique est une notion qui permet de mesurer l'efficacité d'un algorithme [Crochemore et Rytter, 1994, Cormen et al., 2001] indépendamment des contraintes matérielles.

Après les trois premiers chapitres, l'étudiant doit connaître les principes de la programmation, comme il doit constater que la résolution d'un problème peut donner lieu à plusieurs solutions algorithmiques, appelées algorithmes candidats. À ce niveau, l'étudiant se pose la question sur :

1. l'algorithme à prendre en considération, ce qui doit correspondre à l'algorithme le plus efficace.
2. et l'algorithme à rejeter ou le moins efficace.

Certainement, ce choix peut mener à distinguer plusieurs algorithmes efficaces et plusieurs algorithmes inefficaces.

Donc, la complexité algorithmique est une mesure qui permet de décider de la qualité de l'algorithme à savoir : bon ou mauvais.

Ce chapitre comporte quatre sections qui correspondent respectivement à l'analyse d'un algorithme, calcul de la complexité d'un algorithme, les types de complexité et bons et mauvais algorithmes.

4.2 ANALYSE D'UN ALGORITHME

Analyser un algorithme [Cormen et al., 2001] revient à prévoir les ressources que l'algorithme nécessite. Occasionnellement, des ressources telles que la mémoire, la largeur de la bande passante d'une communication, ou le matériel informatique sont les principales préoccupations, mais le plus souvent c'est le temps de calcul [1]. que nous voulons mesurer.

Généralement, en analysant plusieurs algorithmes candidats pour un problème, nous pouvons identifier le plus efficace. Une telle analyse peut indiquer plus d'un candidat viable, mais nous pouvons souvent rejeter plusieurs algorithmes dans le processus.

Considérons un problème donné et un algorithme A pour le résoudre [Prins, 1994]. Sur une donnée x de taille n, l'algorithme requiert un certain temps, mesuré en

1. Le temps de calcul (ou le temps CPU) est le temps consommé par l'exécution de l'algorithme

nombre d'opérations fondamentales [2]. $C_A(x)$ est le coût en temps qui varie évidemment avec la taille de la donnée, mais qui peut aussi varier sur les différentes données de même taille n.

Exemple 4.2.1 (algorithme de tri) *[Danièle Beauquier, 2003] Considérons l'algorithme de tri AT qui, partant d'un tableau $[a_1, ..., a_n]$ de nombres réels distincts à trier en ordre croissant. Donc, l'algorithme AT :*

1. *cherche la première descente, c'est-à-dire le plus petit entier i tel que $a_i > a_{i+1}$,*
2. *échange ces deux éléments,*
3. *recommence sur le tableau obtenu.*

Si l'on compte le nombre d'inversions ainsi réalisées, il varie de 0 pour un tableau trié à $n(n-1)/2$ pour un tableau dans l'ordre décroissant.
Donc, le coût de AT dépend de la taille de la donnée ainsi que l'état de la donnée.

Notre but est d'évaluer le coût d'un algorithme, selon certains critères et en fonction de la taille n des données.

4.2.1 *Déterminer les opérations fondamentales*

Pour certains problèmes, on peut mettre en évidence une ou plusieurs opérations qui sont fondamentales au sens où le temps d'exécution d'un algorithme résolvant ce problème est toujours proportionnel au nombre de ces opérations. Il est alors possible de comparer des algorithmes traitant ce même problème selon cette mesure simplifiée.

Les opérations fondamentales sont par exemple [Danièle Beauquier, 2003] :
— les opérations arithmétiques usuelles,
— les transferts de données,
— les comparaisons entre données,
— etc.

Le choix des opérations fondamentales [Danièle Beauquier, 2003] dépend du niveau d'abstraction où l'on se place. Par exemple, dans le cas où les opérations arithmétiques et les objets sur lesquels elles portent peuvent être plus ou moins compliquées, il peut s'agir simplement d'additionner des entiers naturels, ou de multiplier des polynômes, ou encore de calculer les valeurs propres d'une matrice. A la limite [Prins, 1994], si on veut faire une micro-analyse très précise du temps d'exécution du programme, il suffit de décider que toutes les opérations du programme sont fondamentales.

Remarquons que si on choisit plusieurs opérations fondamentales, on doit les décompter séparément. En d'autres termes, on doit donner un poids à chaque opération fondamentale, en tenant compte du temps d'exécution de celle ci.

On a fait l'hypothèse que le temps d'exécution est proportionnel à la mesure choisie. On ne peut pas comparer deux algorithmes utilisant des mesures différentes [Prins, 1994]. En d'autres termes, la comparaison de deux algorithmes doit

2. Appelées aussi caractéristiques ou élémentaires

se faire sur le même ensemble d'opérations, caractérisées au préalable fondamentales.

4.2.2 *Compter le nombre d'opérations (Calculer le coût)*

Après avoir déterminé les opérations fondamentales, il s'agit de compter leur nombre.

Définition 4.2.2 (nombre d'opérations) *Le nombre d'opérations fondamentales d'un algorithme A est une fonction $T_A(n)$ donnant le nombre d'opérations exécutées par A et caractérisées « fondamentales »au préalable.*

Il n'existe pas de système complet de règles permettant de compter le nombre d'opérations en fonction de la syntaxe des algorithmes, mais l'on peut faire quelques remarques [Cormen et al., 2001, Prins, 1994] :

1. Lorsque les opérations sont dans une séquence d'instructions, leurs nombres s'ajoutent.
2. Pour les branchements conditionnels [3], il est en général difficile de déterminer quelle branche de la condition est exécutée, et donc quelles sont les opérations à compter. Cependant, on peut majorer ce nombre d'opérations [4].
3. Pour les boucles, le nombre d'opérations dans une boucle de n itérations est

$$\sum_{i=1}^{n} T(i)$$

   où i est la variable de contrôle de la boucle, et $T(i)$ le nombre d'opérations fondamentales exécutées à la $i^{ème}$ itération. Si le nombre d'itérations est difficile à calculer, on peut se contenter d'une borne de majoration.
4. Pour les appels de procédures et de fonctions, on peut s'arranger à calculer la complexité de ces appels, et les prendre en compte suivant l'imbrication de l'appel dans l'algorithme.
5. Pour les appels de procédures et de fonctions récursives, compter le nombre d'opérations fondamentales donne en général lieu à la résolution de relations de récurrence. En effet le nombre $T(n)$ d'opérations dans l'appel de la procédure avec un argument de taille n s'écrit selon la récursion, en fonction de divers $T(k)$, pour $k < n$ [5].

Il est évident que le calcul du coût d'un algorithme dépend de la donnée sur laquelle il opère [Danièle Beauquier, 2003]. Les différentes étapes de ce calcul se donnent comme suit [Prins, 1994] :

3. Les alternatives complètes et les alternatives incomplètes
4. En pratique ça va dépendre du type de complexité à calculer : dans le pire des cas, dans le moyen des cas ou dans le meilleur des cas (voir la section 4.4)
5. Dans le cas d'un algorithme récursif, la complexité est généralement loin d'être polynomial

1. Il faut d'abord définir une mesure de taille sur les données qui reflètent la quantité d'information contenue. Par exemple, pour additionner ou multiplier des entiers, on peut prendre comme mesure significative le nombre de chiffres des nombres.
2. Pour certains algorithmes, le temps d'exécution ne dépend que de la taille des données ; mais la plupart du temps la complexité varie aussi, pour une taille déterminée des données, en fonction de la donnée elle-même.

4.3 CALCUL DE LA COMPLEXITÉ D'UN ALGORITHME

Définition 4.3.1 (complexité d'un algorithme) *[Prins, 1994] La complexité d'un algorithme A est une fonction $C_A(N)$ donnant le nombre d'instructions caractéristiques (opérations fondamentales) exécutées par A dans le pire des cas pour une donnée de taille N.*

4.3.1 *Taille d'une donnée*

Considérons un problème donné, et un algorithme pour le résoudre. Sur une donnée x de taille n, l'algorithme requiert un certain temps, mesuré en nombre d'opérations fondamentales, soit $T_n(x)$. Le coût en temps varie évidemment avec la taille de la donnée, mais peut aussi varier sur les différentes données de même taille n[Danièle Beauquier, 2003].

La taille d'une donnée est la **quantité mémoire** nécessaire pour la stocker. Pour être rigoureux, il faudrait la mesurer en nombre de bits. Dans notre cours et dans le cas général, il suffit de compter le nombre de mots-mémoire nécessaires, sans précision. Selon le contexte, ces mots peuvent être des entiers courts ou longs, des réels, etc.

Exemple 4.3.2 (taille d'une donnée) *[Prins, 1994] Pour le tri de N nombres, on va avoir besoin de $N+1$ mots-mémoire : N nombres + 1 pour le nombre N. Ce qui donne une taille de $N+1$.*

4.3.2 *Ordre d'une fonction*

Dans le but de bien présenter les notions de base sur **l'ordre d'une fonction**, nous préférons s'initier à travers un exemple illustratif pris de [Prins, 1994].

Considérons deux algorithmes A_1 et A_2 pour un même problème de taille N, avec les complexités respectives $C_{A_1}(N) = 0.5N^2$ et $C_{A_2}(N) = 5N$.

La figure 24 montre le tracé des courbes C_{A_1} et C_{A_2} en fonction de N.

Nous constatons que A_2 fait plus d'opérations que A_1 pour $N = 5$ mais moins dès que $N \geq 10$. En fait, quels que soient les coefficients positifs de N et N^2 dans C_{A_1} et C_{A_2}, A_2 ira toujours plus vite à partir d'une certaine taille du problème.

Définition 4.3.3 (la relation d'ordre) *[Prins, 1994] Soient f et g deux fonctions de $\mathbb{R}$ dans $\mathbb{R}$. On dit que f est d'ordre inférieur ou égal à g, ou d'ordre au plus g, si on peut trouver un réel x_0 et un réel positif c tels que : $\forall x \geq x_0, f(x) \leq c.g(x)$.*

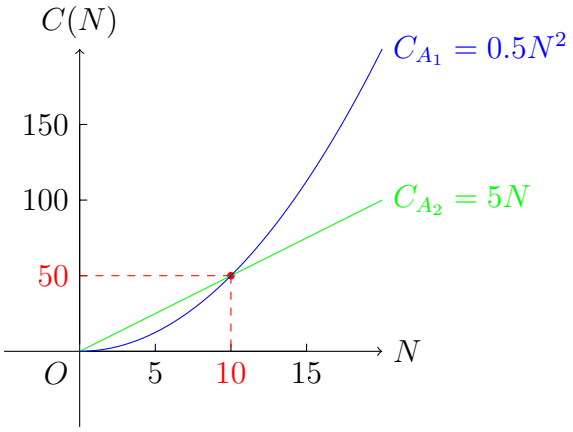


FIGURE 24 – Courbes respectives de C_{A_1} et C_{A_2}

En d'autres termes, g devient plus grande que f à partir d'un certain nombre x_0, à un facteur c près. On écrit f est $O(g)$, f est en $O(g)$ ou $f = O(g)$ et on prononce grand O de g. Cette notation, dite de landau, est consacré par l'usage. En fait, $O(g)$ est un ensemble de fonctions, celles d'ordre au plus g : il sera préférable d'écrire $f \in O(g)$.

Définition 4.3.4 (la notation O) *[Danièle Beauquier, 2003] On considère une fonction $g : \mathbb{R} \to \mathbb{R}$. Étant donné un point $x_0 \in \mathbb{R} \cup \{-\infty, +\infty\}$, on désigne par $O(g)$ l'ensemble des fonctions f pour lesquelles il existe un voisinage V de x_0 et une constante $k > 0$ tels que :*

$$| f(x) | \leqslant k | g(x) | \quad (x \in V)$$

Exemple 4.3.5 (relation d'ordre) *[Prins, 1994] Un cas simple d'application de la définition est quand $f \leq g$ à partir d'une certaine valeur. Ainsi en considérant les algorithmes A_1 et A_2 cités précédemment et dont les complexités respectives $C_{A_1}(N) = 0.5N^2$ et $C_{A_2}(N) = 5N$. Soient les fonctions :*
- *$f(N) = 5N$,*
- *et $g(N) = 0.5N^2$.*

On aura $5N \geq 0.5N^2$ pour $N \geq 10$ (voir le figure 24). Dans ce cas on peut dire que g devient plus grande que f à partir de $N = 10$, à un facteur 1. En d'autres termes : $5N \in O(0.5N^2)$ ou $f \in O(g)$

Si f et g admettent des limites, on a les définitions d'ordre suivantes [Prins, 1994]

1. si $\lim\limits_{n\to\infty} \frac{f}{g} = c > 0$, f et g sont du même ordre : $f = O(g)$ et $g = O(f)$, on note $f = \Theta(g)$,
2. si $\lim\limits_{n\to\infty} \frac{f}{g} = 0$, $f = O(g)$ mais $g \neq O(f)$,
3. si $\lim\limits_{n\to\infty} \frac{f}{g} = +\infty$, f est d'ordre supérieur à g, on note $f = \Omega(g)$, ou $g = O(f)$,

Les complexités sont le plus souvent exprimées à l'ordre près grâce aux avantages suivants [Prins, 1994] :

- $f = O(g)$ signifie en fait que g est un majorant de la complexité réelle. On peut se contenter, par exemple, de borner grossièrement le comportement au pire d'un algorithme.
- $10N^2 = O(0.5N^2)$ et vice versa : deux fonctions différant d'un facteur constant sont du même ordre. Ceci permet de dépasser les problèmes d'implémentation qui mettent en échec le calcul de la complexité.
- $N^2 + 3N + 4 = O(N^2)$: les termes d'ordre inférieur peuvent être négligés. Ainsi, pour N assez grand, l'initialisation d'un tableau T de N éléments en $O(N)$ est négligeable devant deux boucles imbriquées de N itérations sur T qui coûtent $O(N^2)$.
- Il n'est pas besoin d'être précis sur la notion d'opérations caractéristiques d'un algorithme. Le nombre exact d'instructions compté dans un corps d'une boucle, par exemple, sera sans effet sur la complexité, s'il est constant.

4.4 TYPES DE COMPLEXITÉ

La complexité d'un algorithme dépend du nombre d'opérations fondamentales exécutées. Ce nombre est lié à l'état des données (en plus de leur taille), particulièrement lorsque l'algorithme contient des alternatives. Dans la littérature, nous distinguons trois types de complexité [Froidevaux et al., 1993, Cormen et al., 2001, Shaffer, 2012, Danièle Beauquier, 2003] :

1. complexité dans le meilleur des cas,
2. complexité dans le pire des cas,
3. et complexité dans le moyen des cas.

Dans la suite, nous utilisons le notations suivantes :

x : la donnée,

$T_A(n)$: le nombre d'opérations exécutées par l'algorithme A,

$D(n)$: le nombre de données de taille n [6].

4.4.1 *Complexité dans le meilleur des cas*

On calcule la complexité dans le cas le plus favorable. En d'autres termes, le cas où l'algorithme exécute le minimum d'opérations fondamentales. Elle est notée $Min_A(n)$, tel que A est le nom de l'algorithme et n la taille de la donnée. Le calcul de la complexité dans le pire des cas se donne par la relation 1.

$$Min_A(n) = min_{\substack{|x| = n \\ i = 1..D(n)}} T_i(x) \tag{1}$$

4.4.2 *Complexité dans le pire des cas*

Le coût $Max_A(n)$ d'un algorithme A dans le cas le plus défavorable ou dans le pire des cas est par définition le maximum des coûts, sur toutes les données

6. Le nombre des différents cas ou des différents états de la donnée x.

de taille n. Le calcul de la complexité dans le meilleur des cas suit la formule ci-dessous 2.

$$Max_A(n) = max_{\substack{|x| = n \\ i = 1..D(n)}} T_i(x) \tag{2}$$

4.4.3 *Complexité en moyenne*

Dans des situations où l'on pense que le cas le plus défavorable ne se présente que rarement, on est plutôt intéressé par le coût moyen de l'algorithme. Une formulation correcte de ce coût moyen suppose que l'on connaisse une distribution de probabilités sur les données de taille n. Si $p_i(x)$ est la probabilité de la $i^{ème}$ donnée x[7], le coût moyen $Moy_A(n)$ d'un algorithme A sur les données de taille n suit par définition la formule 3.

$$Moy_A(n) = \sum_{\substack{|x| = n \\ i = 1..D(n)}} p_i(x) T_i(x) \tag{3}$$

Le plus souvent, on suppose que la distribution est uniforme, c'est-à-dire que $p_i(x) = \frac{1}{D(n)}, \forall i \in \{1, ..D(n)\}$. Alors, l'expression du coût moyen prend la forme donnée via la relation 4.

$$Moy_A(n) = \frac{1}{D(n)} \sum_{\substack{|x| = n \\ i = 1..D(n)}} T_i(x) \tag{4}$$

En pratique, l'estimation du coût moyen d'un algorithme est en général plus délicate que l'estimation du coût dans le cas le plus défavorable, et nécessite une très bonne connaissance des combinatoires sur les objets traités. Puisque une formulation correcte de ce coût moyen suppose que l'on connaisse une distribution de probabilités sur les données de taille n.

Raisonnement, il existe entre les trois complexités la relation suivante :

$$Min_A(n) \leq Moy_A(n) \leq Max_A(n)$$

Dans le cas où le comportement de l'algorithme dépend uniquement de la taille des données $D(n) = 1$ (comme dans l'exemple de la multiplication des matrices), la relation d'égalité $Min_A(n) = Moy_A(n) = Max_A(n)$ est vérifiée, relation qui n'est souvent pas vraie.

4.5 BONS ET MAUVAIS ALGORITHMES

4.5.1 *Classification des algorithmes[Prins, 1994]*

Une classification grossière mais reconnue distingue les algorithmes polynomiaux et des algorithmes exponentiels.

7. la probabilité de croiser le $i^{ème}$ état de la donnée x

Des exemples de complexités polynomiales sont : $log(n)$, $n^{0.5}$, $n.log(n)$, n^2,etc. Une complexité exponentielle peut être une vraie exponentielle au sens mathématique du mot : e^n,2^n mais aussi les fonctions $n^{log(n)}$ (dites sous exponentielle), $n!$ et n^n.

Un algorithme est caractérisé « efficace » ou « bon algorithme » s'il est de complexité polynomiale. Un critère d'efficacité confirmé par la pratique :

— Une exponentielle dépasse tout polynôme pour n assez grand. Par exemple, 1.1^n croît d'abord lentement, mais finit par dépasser n^{100}.

— Il est vrai qu'un algorithme avec la complexité exponentielle 1.1^n, est plus intéressant qu'un autre de complexité polynomiale n^{100} pour n pas trop grand. Mais il est rare de trouver des complexités polynomiales d'ordre supérieur à 4 en pratique.

— L'ensemble des polynômes a d'intéressantes propriétés de fermeture, l'addition, la multiplication, la composition de polynômes donnent des polynômes. On peut ainsi constituer de grands algorithmes polynomiaux à partir de plus petits.

Exemple 4.5.1 *Le tableau ci-dessous 7 donne une estimation du temps d'exécution de chacun des algorithmes pour différentes tailles n des données du problème. On fait l'hypothèse que l'ordinateur exécute* 10^{16} *opérations par seconde, c'est-à-dire dix millions de milliards d'opérations par seconde.*

TABLE 7 – Le temps d'exécution nécessaire pour une donnée $n = 20, 50, 100, 200, 500$ et 1000 et avec une complexité en fonction de n [8]

Taille → Complexité ↓	20	50	100	200	500	1000
$10^3.n$	$2.10^{-6}\mu s$	$5.10^{-6}\mu s$	$1.10^{-5}\mu s$	$2.10^{-5}\mu s$	$5.10^{-5}\mu s$	$1.10^{-4}\mu s$
$10^3.n.log_2 n$	$5,99.10^{-6}\mu s$	$1,96.10^{-5}\mu s$	$4,61.10^{-5}\mu s$	$1,06.10^{-4}\mu s$	$3,11.10^{-4}\mu s$	$6,91.10^{-4}\mu s$
$100.n^2$	$4.10^{-6}\mu s$	$2,5.10^{-5}\mu s$	$1.10^{-4}\mu s$	$4.10^{-4}\mu s$	$2,5.10^{-3}\mu s$	$0,001\mu s$
$10.n^3$	$8.10^{-6}\mu s$	$1,25.10^{-4}\mu s$	$1.10^{-3}\mu s$	$0.008\mu s$	$0.125\mu s$	$1\mu s$
$n^{log_2 n}$	$7,90.10^{-7}\mu s$	$4,43.10^{-4}\mu n$	$0,162\mu s$	$155,449\mu s$	$5,93s$	$14,69h$
$2^{\frac{n}{3}}$	$6,4.10^{-9}\mu s$	$6,55.10^{-6}\mu s$	$0,859\mu s$	$2,05h$	$--$	$--$
2^n	$1,05.10^{-4}\mu s$	$0,1126s$	$22.10^6 ans$	$--$	$--$	$--$
3^n	$0,3487\mu s$	$31 jours$	$--$	$--$	$--$	$--$
$n!$	$243,29s$	$--$	$--$	$--$	$--$	$--$

La puissance de la machine ne résout rien pour les complexités exponentielles. Ainsi, pour traiter un problème de taille 100 *dans le même temps qu'un problème de taille* 50*, il suffit d'un ordinateur* 16 *fois plus puissant pour* $10.n^3$*,* 366 *fois plus puissant pour* $n^{log_2 n}$*,* 131145 *fois plus pour* $2^{n/3}$*. Et tout cela pour seulement doubler la taille des problèmes traités.*

Les algorithmes utilisables pour des données de grande taille sont ceux qui s'exécutent :

8. Le tableau est pris de [Prins, 1994] avec mise-à-jour des temps.

- en temps constant (c'est le cas de la complexité en moyenne de certaines méthodes de hachage),
- en temps logarithmique (par exemple la recherche dichotomique, ou les arbres binaires de recherche),
- en temps linéaire (par exemple la recherche séquentielle d'un élément dans un tableau).

4.5.2 *Exemple illustratif*

Soit l'algorithme RSALGO qui fait la recherche d'un élément v dans un tableau T [Cormen et al., 2001]. L'objectif de cet exemple est de déterminer les trois types de complexité : au pire, au meilleur et en moyenne des cas.

Nous ne considérons comme opérations fondamentales que les comparaisons réalisées entre l'élément v et les composants du tableau.

Algorithm 1 RSALGO(L, j)

Require: T un tableau à n éléments
Ensure: Recherche l'indice i de élément v dans le tableau T. Si v ne figure pas dans le tableau, i est égal à $n+1$.

```
1: i ← 1
2: trouve ← false
3: while (i ≤ n)∧(non trouve) do
4:     if T[i] = v then
5:         trouve ← true
6:     else
7:         i ← i + 1
8:     end if
9: end while
```

1. Dans le meilleur des cas, l'élément v se trouve dans la première case du tableau, d'où $Min(n) = 1$.
2. Dans le pire des cas, v ne figure pas dans la liste, d'où $Max(n) = n$ (on parcourt tout le tableau).
3. Pour calculer $Moy(n)$, on suppose que tous les éléments sont distincts et on doit se donner des probabilités sur T et v. Nous avons les informations ci-dessous sur le système probabiliste de notre problème :

   CAS 1 : v n'est pas dans T et la probabilité est q,

   CAS 2 : v est dans T et tous les indices des composants sont équiprobables.

   On note $T_{n,i}$ pour $1 \leq i \leq n$, le nombre d'opérations fondamentales exécutées dans le cas où v apparaît à la $i^{ième}$ case de T et $T_{n,n+1}$ le nombre d'opérations fondamentales exécutées dans le cas où v ne figure pas dans T. D'après les données citées ci-dessus, nous avons :

$$\forall i \in \{1, ..n\}, p_{n,i} = q/n$$

et

$$p_{n,n+1} = 1 - q$$

D'après l'analyse de l'algorithme, on a :

$$\forall i \in \{1, ..n\}, T_{n,i} = i - 1$$

et

$$T_{n,n+1} = n$$

Donc, on aura :

$$Moy(n) = \sum_{i=0..n} p_{n,i} T_{n,i} = (1-q)n + \sum_{i=1..n} (i-1)q/n$$

$$Moy(n) = (1-q)n + (n-1)q/2.$$

Si on sait que v figure dans le tableau, on a $q = 1$, ce qui donne :

$$Moy(n) = (n-1)/2$$

Si v a une chance sur deux d'être dans le tableau, on a $q = 1/2$, ce qui donne :

$$Moy(n) = n/2 + (n-1)/4 = \frac{3n-1}{4}$$

5

STRUCTURES DE DONNÉES COMPLEXES

5.1 INTRODUCTION

Dans ce chapitre nous introduisons les structures de données complexes. Nous présentons les types usuels, leurs définitions en C++ et les différentes primitives correspondants.

Chacune de ces structures est accompagnée de figures illustratives permettant une meilleure compréhension des différents concepts et fonctions. Comme nous essayons d'étudier et de mettre en avant la présentation de la structure en mémoire et sa mise en œuvre et son emplacement dans un programme C++.

Ce chapitre est organisé de la manière suivante :

- La première section introduit les notions des structures linéaires (ou séquentielles). Elle regroupe les listes contiguës, les listes chaînées, les piles et les files.
- La deuxième section traite les structures arborescences, à savoir les arbres et les graphes.
- La troisième et dernière section est consacrée aux fichiers.

5.2 STRUCTURES LINÉAIRES

Les structures linéaires permettent de représenter en mémoire des données de même type. L'une des caractéristiques fondamentales de ces structures est que chaque élément (ou donnée) possède un successeur sauf le dernier.

Dans ce cours nous introduirons quatre structures linéaires, à savoir les listes contiguës, les listes chaînées (simples et double), les piles et les files.

5.2.1 *Listes contiguës*

Définition de la liste en C++

La structure d'une liste contiguë comporte deux champs : le tableau *Tab* et le nombre d'éléments lg rangés dans le tableau, avec s la taille de *Tab*. Par conséquent, la valeur de lg ne doit pas dépasser s ; en d'autres termes $lg \leqslant s - 1$.

La déclaration de la structure de la liste contiguë se donne comme suit :

```
struct contig
{
      TypeElt Tab[s];
      int lg;
}
```

Sachant que *TypeElt* est le type des éléments à charger dans le tableau *Tab*. La figure 25 illustre la la structure de la liste en mémoire.

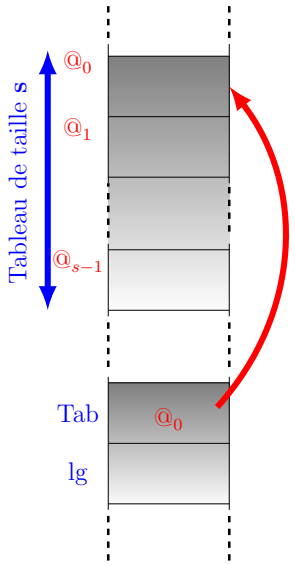


FIGURE 25 – Illustration de la liste contiguë en mémoire

Les cases grisâtres représentent l'espace mémoire occupé par une variable de type *contig*.

Nous donnons les primitives d'insertion et de suppression dans les deux sous-sections 5.2.1 et 5.2.2. L'exécution de ces primitives doit se faire selon des normes (conditions à vérifier), afin de conserver la caractéristique de contiguïté au niveau de la liste.

Ajout d'un élément

Pour ajouter un élément dans la $k^{ème}$ cellule d'une liste contiguë, nous devons :

1. vérifier si l'insertion de l'élément est possible. En d'autres termes, vérifier si $lg \leq s - 1$ et $k \leqslant lg$ sinon l'insertion est impossible.
2. décaler tous les éléments se trouvant entre les cases k et $lg - 1$ vers la droite,
3. saisir la valeur à insérer dans la case k de la liste,
4. mettre à jour le champ lg.

Ce qui donne en C++ la fonction *void ajout(contigu &, int, TypeElt)* ci-dessous :

```
void ajout(contigu & g, int k, TypeElt val)
{
  int l=g.lg;
  if (l<s-1 && k<=l)
  {
    for(int i=l-1;i>=k; i--)
      g.tab[i+1]=g.tab[i];

    g.tab[k]=val;
    g.lg = l+1;
  }
  else cout << "Insertion Impossible !" << endl;
}
```

Exemple 5.2.1 (insertion dans une liste contiguë) *Soit la liste contiguë d'entiers définie en C++ comme suit :*

```
struct contig
       {
           int Tab[8];
           int lg;
       }
```

On suppose que la liste contient 4 éléments $\{12, -1, 11, -3\}$*, ce qui donne « lg=4 ».*
Soit $k = 2$ *l'emplacement où on désire insérer la valeur « 100 » dans le tableau Tab.*
La figure 26 illustre l'état de la liste en mémoire à chaque étape réalisée pour insérer la valeur « 100 » dans la case d'indice « k=2 ».
La figure ***(a)*** *illustre la liste et les variables manipulées en mémoire avant l'insertion.*
Nous avons $lg \leq s - 1$ $(4 \leqslant 7)$ *et* $k \leq lg$ $(2 \leqslant 4)$*, donc l'insertion est possible.*
La figure ***(b)*** *illustre l'étape de décalage réalisée sur les éléments d'indices* $ind = k..lg - 1$ $(ind = 2..3)$*.*
Après avoir décalé les éléments $\{11, -3\}$*, nous insérons la nouvelle valeur « 100 » à la* k^{eme} *case, avec incrémentation de* lg *(puisque le nombre d'éléments qui composent la liste a augmenté de 1). Ce qui donne la figure* ***(c)*** *qui illustre l'état de la liste en mémoire après l'insertion.*
Dans la figure 26, les cases en gris représentent les cases non occupées dans la liste.

Suppression d'un élément

La suppression d'un élément d'indice k de la liste contiguë, doit passer par les étapes suivantes :

1. vérifier si $k \leqslant lg - 1$ pour que la suppression de l'élément soit possible.
2. décaler tous les éléments se trouvant entre les cases $k + 1$ et $l - 1$ vers la gauche,
3. mettre à jour le champ lg.

Nous donnons ci-dessous la fonction *void supprime(contigu &,int)* qui permet la suppression.

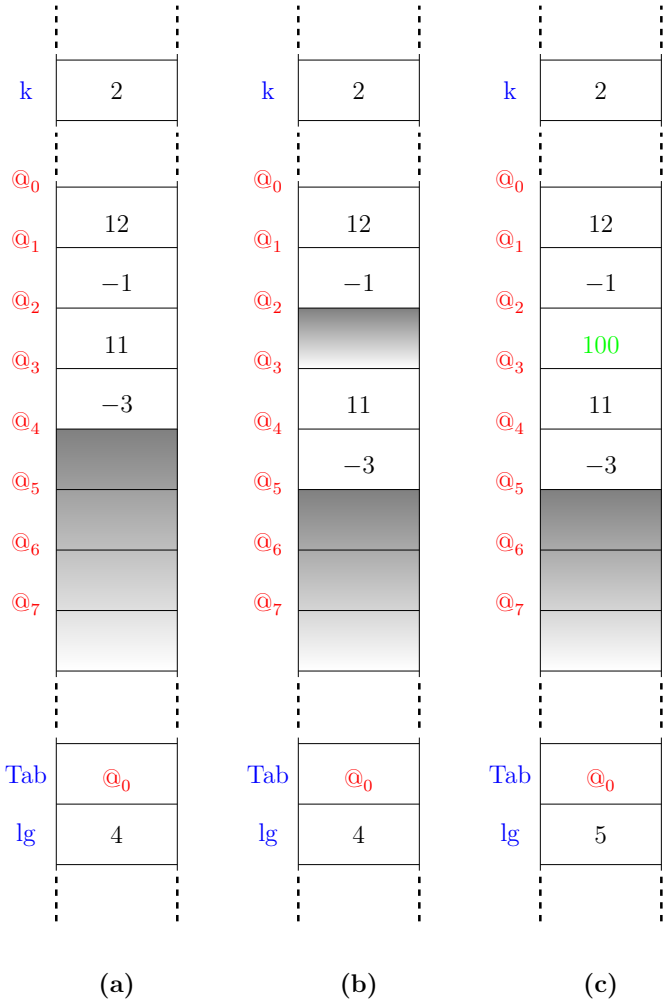


FIGURE 26 – Liste contiguë de type *contig*

```
void supprime(contigu & g,int k)
{
   int l=g.lg;
  if (k<=l-1)
  {
    for(int i=k;i<l-1; i++)
      g.tab[i]=g.tab[i+1];
    g.lg = l-1;
  }
  else cout << "Suppression Impossible !" << endl;
}
```

Exemple 5.2.2 (suppression dans une liste contiguë) *Soit la liste contiguë définie en C++ dans l'exemple 5.2.1.*

Nous supposons que la liste est à son état final après l'insertion et elle contient 5 éléments $\{12, -1, 100, 11, -3\}$ *avec « lg=5 ».*

Soit $k = 1$ *l'emplacement de la valeur qu'on souhaite supprimer du le tableau Tab.*

La figure 27 illustre l'état de la liste en mémoire à chaque étape réalisée pour supprimer la valeur d'indice « k=1 ».

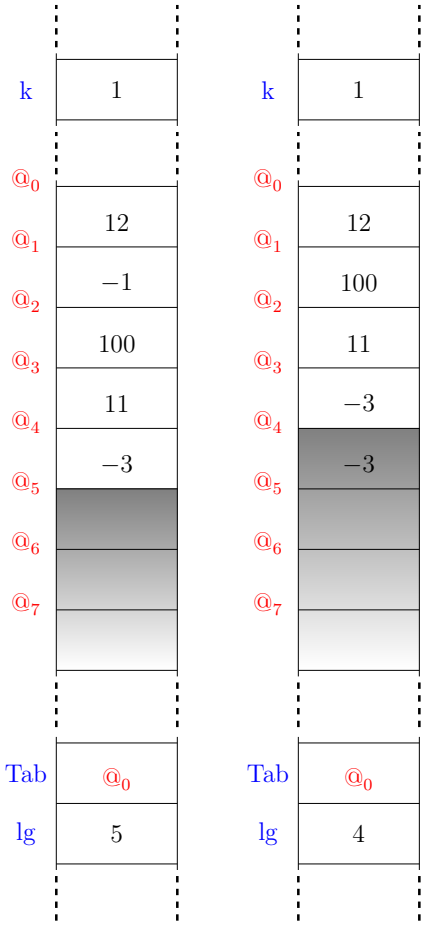


FIGURE 27 – Liste contiguë de type *contig*

*La figure **(a)** illustre la liste et les variables manipulées en mémoire avant la suppression. Nous avons $k \leqslant lg - 1$ $(2 \leqslant 4)$, donc la suppression est possible.*

*La figure **(b)** illustre le décalage réalisée sur les éléments de la liste d'indice $ind = k + 1..lg - 1$ $(ind = 2..4)$. La valeur de lg se décrémente de 1 (puisque le nombre d'éléments qui composent la liste a diminué de 1). Donc, la figure **(c)** illustre l'état de la liste en mémoire après suppression.*

Dans la figure 27, les cases en gris représentent les cases non occupées dans la liste.

5.2.2 *Listes chaînées simples*

Une liste chaînée représente un ensemble d'éléments organisés séquentiellement, tout comme un tableau. Sauf que dans un tableau, l'organisation séquentielle est donnée implicitement par la position dans le tableau. Dans une liste chaînée, on utilise un arrangement explicite dans lequel chaque élément appartient à un *nœud* qui contient un *lien* au nœud suivant. La figure 28[Sedgewick, 1991]

montre une liste chaînée dans laquelle les éléments sont représentés par des lettres, les nœuds par des cercles et les liens par des segments reliant les nœuds.

Il est habituel de trouver un nœud sentinelle à premier bout de la liste, ce nœud appelé "début" ou *entête*, pointera sur le premier élément de la liste.

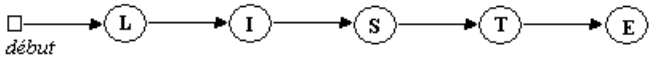


FIGURE 28 – Liste chaînée

Définition de la liste en C++

Une liste chaînée est une structure composée d'une suite de maillons, dont chacun porte une donnée et une valeur adresse qui pointe sur le maillon suivant.

Afin d'avoir accès aux éléments (ou maillons) de la liste chaînée, nous avons besoin d'un élément sentinelle, appelé *entête* qui doit pointer sur le premier.

La déclaration en C++ du maillon et de l'entête de la liste se donne comme suit :

```
struct liste
{
        TypeElt elt;
        liste * suivant;
}
liste * tete = NULL;
```

L'entête de la liste est initialisé à *NULL*, puisque la liste est initialement vide.

La figure ci-dessous 29 représente la structure d'un maillon i, qui est composé de deux champs *elt* de valeur val_i et l'adresse en mémoire $@_i$. La valeur val_i est la valeur de la donnée de type *TypeElt* que nous devons charger dans la liste.

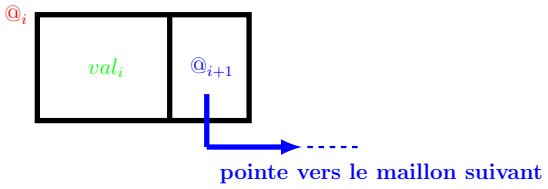


FIGURE 29 – Structure d'un maillon

Le lien entre les différents éléments est réalisé par un chaînage simple, tout en vérifiant la relation de *successeur* entre ces éléments.

La figure 30 représente la structure d'une liste simplement chaînée, qui est composée de $n + 1$ éléments.

Nous constatons que chaque élément i possède une adresse $@_i$ et contient une donnée et pointe sur l'élément suivant (ou l'élément successeur).

Contrairement à la structure tableau, où les éléments sont superposés d'une manière contiguë en mémoire, la disposition des éléments d'une liste chaînée

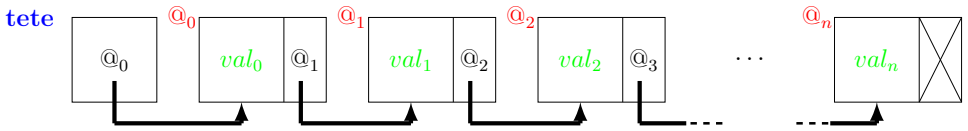


FIGURE 30 – Structure d'une liste simplement chaînée

dépend des emplacements libres en mémoire. Ce qui donne l'impression que ces éléments sont éparpillés en mémoire, tel illustré dans la figure 31.

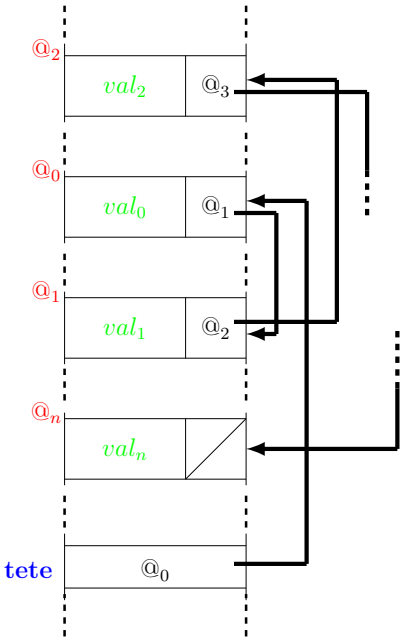


FIGURE 31 – Illustration d'une liste simplement chaînée en mémoire

Création d'un maillon

Sachant qu'une liste chaînée est une structure dynamique, on fait correspondre à chaque apparition d'un nouvel élément à insérer, une allocation dynamique d'un espace mémoire suffisant.

Nous avons ci-dessous la fonction *liste * creer_maillon(TypeElt)* qui permet de réserver l'espace mémoire nécessaire pour le maillon et charger la valeur de la donnée de type *TypeElt*. Elle retourne l'adresse du maillon (de type *liste* *).

```
liste * creer_maillon(TypeElt val)
{
  liste * maillon = new liste;
  if (maillon) {
    maillon->elt = val;
    maillon->suivant = NULL
  }
  return maillon;
}
```

Dans le cas où l'espace mémoire est insuffisant, la fonction retourne la valeur *NULL*, ce qui exprime l'impossibilité de créer le maillon.

Ajout d'un élément

La taille de la liste n'est pas connue au préalable, elle démarre de 0 et s'incrémente à chaque insertion d'un nouveau maillon.

L'insertion d'un maillon dans la liste se donne en C++ comme suit :

1. Il faut tout d'abord créer le nouvel élément et saisir ou stocker les valeurs.
2. Insérer cet élément dans la liste. La cohérence de la liste doit finalement être rétablie en indiquant que le nouvel élément est désormais le successeur de celui après lequel il va prendre place.

L'insertion dans une liste chaînée simple peut se faire au début, au milieu et à la fin.

A. Insertion en tête de liste : La fonction d'insertion à l'entête de la liste se donne via la fonction *void ajout_tete(liste*&, TypeElt)*. L'élément à insérer est créé en faisant appel à la fonction *creer_maillon*.

```
void ajout_tete(liste*& tete, TypeElt val)
{
  liste *maillon=creer_maillon(val);
  if (maillon != NULL)
  {
    maillon->suivant = tete;
    tete = maillon;
  }
}
```

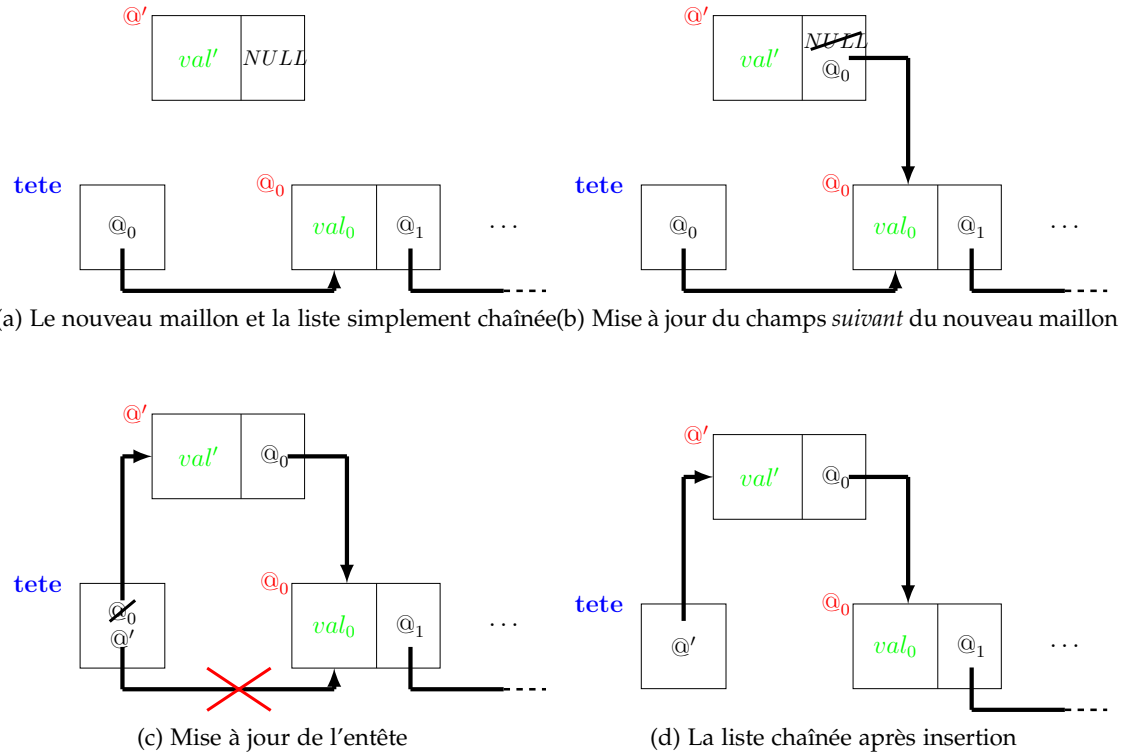
Dans le cas où le maillon est déjà présent en mémoire, on se contente de mettre son adresse en entrée de la fonction. Ce qui donne en C++ la version suivante :

```
void ajout_tete(liste *& tete, liste * maillon)
{
    maillon->suivant=tete;
    tete = maillon;
}
```

La figure ci-dessous 32 illustre les différentes étapes d'insertion d'un maillon à l'entête d'une liste chaînée.

Initialement la variable *tete* pointe sur le premier élément ayant l'adresse mémoire $@_0$, le maillon à insérer est présent en mémoire dont l'adresse est $@'$ (voir la sous-figure 32a.

L'instruction *maillon->suivant = tete* est illustrée dans la sous-figure 32b. Le champ suivant du nouveau maillon dont la valeur initiale est *NULL*, est mis à jour par l'adresse du premier élément de la liste. Par conséquent, le premier élément de la liste devient le successeur du nouveau.



(a) Le nouveau maillon et la liste simplement chaînée (b) Mise à jour du champs *suivant* du nouveau maillon

(c) Mise à jour de l'entête (d) La liste chaînée après insertion

FIGURE 32 – Insertion d'un maillon en tête de liste

La sous-figure suivante 32c montre l'impact de l'exécution de *tete* = *maillon* sur la liste. L'entête est mis à jour, il pointe sur le nouveau maillon d'adresse @′.

Enfin, nous avons la sous-figure 32d qui illustre l'état de la liste après l'exécution de la fonction *ajout_tete*. Nous avons une nouvelle liste dont le nouveau maillon devient le premier et pointe sur un deuxième élément d'adresse $@_0$.

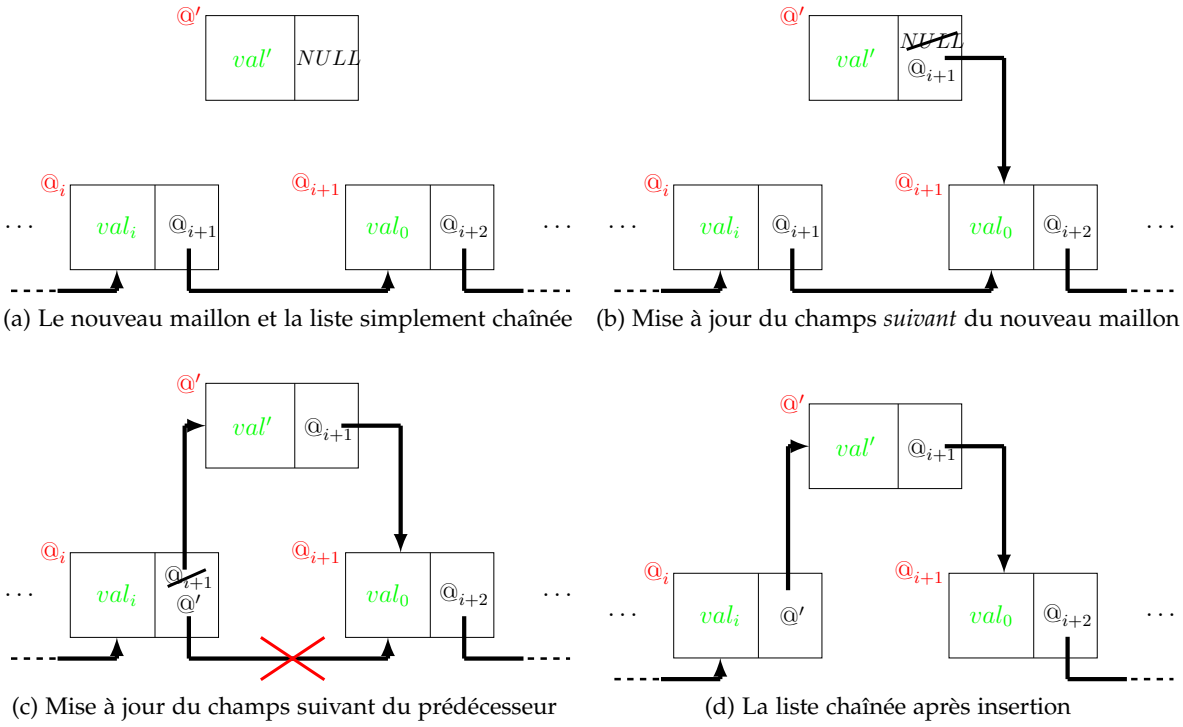
B. Insertion au milieu de la liste : L'insertion au milieu de la liste impose la connaissance de l'adresse de l'élément qui doit précéder le nouveau maillon.

La fonction d'insertion au milieu de la liste se donne comme suit :

```
void ajout_milieu(liste * pred, TypeElt val)
{
  liste *maillon=creer_maillon(val);
  if (maillon != NULL)
  {
    maillon->suivant = pred->suivant;
    pred->suivant = maillon;
  }
}
```

Sachant que *pred* est l'adresse de l'élément qui va précéder l'élément à insérer.

Nous montrons via la figure ci-dessous 33 les différentes étapes d'insertion d'un maillon au milieu d'une liste chaînée, entre les deux maillons dont les adresses respectives $@_i$ et $@_{i+1}$.



(a) Le nouveau maillon et la liste simplement chaînée

(b) Mise à jour du champs *suivant* du nouveau maillon

(c) Mise à jour du champs suivant du prédécesseur

(d) La liste chaînée après insertion

FIGURE 33 – Insertion d'un maillon au milieu de la liste

Initialement le maillon i est le prédécesseur du maillon $i+1$, qui joue à son tour le rôle de successeur, le maillon à insérer est présent en mémoire dont l'adresse est @′ (voir la sous-figure 33a.

L'application de l'instruction *maillon->suivant = pred->suivant* sur la liste est illustrée via la sous-figure 33b. Le champ suivant du nouveau maillon dont la valeur initiale est $NULL$, est remplacé par l'adresse du successeur du $i^{ème}$ élément : $@_{i+1}$. Par conséquent, le successeur de l'élément i devient le successeur du nouveau maillon.

La sous-figure suivante 33c montre l'impact de l'exécution de *pred->suivant = maillon* sur la liste. La variable *pred* contient l'adresse @i du $i^{ème}$ élément, dont le champ suivant est mis à jour et pointe sur le nouveau maillon d'adresse @′.

Le résultat final de l'exécution de la fonction *ajout_milieu* est illustré dans la sous-figure 33d. Nous avons une nouvelle liste dont le nouveau maillon est bien inséré entre les éléments i et $i+$.

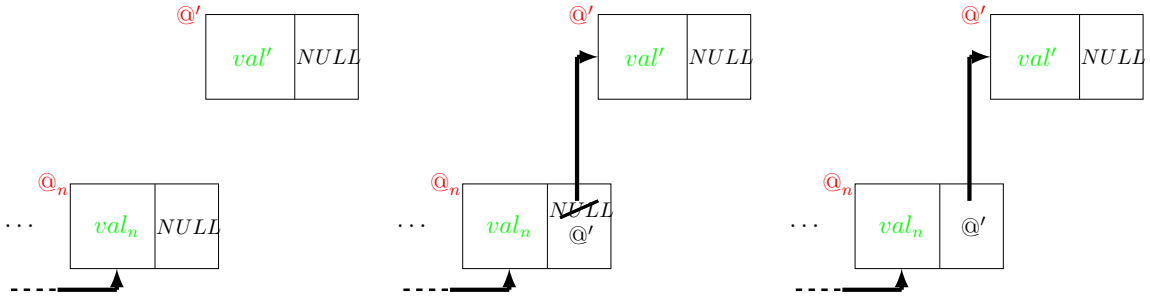
C. Insertion en queue de la liste : Afin d'insérer un maillon en queue de la liste, nous devions avoir en entrée l'adresse du maillon en queue (ou en fin) de la liste.

La fonction d'insertion en queue de la liste se donne comme suit :

```
void ajout_queue(liste * q, TypeElt val)
{
  liste *maillon=creer_maillon(val);
  if (maillon != NULL)
  {
    maillon->suivant = NULL;
    q->suivant = maillon;
  }
}
```

Avec, q l'adresse de l'élément en queue de liste avant l'insertion.

Nous montrons via la figure ci-dessous 34 les différentes étapes d'insertion d'un maillon en queue de la liste chaînée.



(a) Le nouveau maillon et la liste simplement chaînée

(b) Mise à jour du champs *suivant* du dernier maillon

(c) La liste après insertion

FIGURE 34 – Insertion d'un maillon en queue de liste

Initialement, tel illustré dans la sous-figure 33a, le maillon en queue de la liste possède l'adresse $@_n$ et il ne pointe sur aucun autre élément, le champ suivant est $NULL$ et le maillon à insérer est déjà créé à l'adresse est $@'$.

L'application de l'instruction *q->suivant = maillon* sur la liste est donnée via la sous-figure 33b. Le champ suivant du $n^{ème}$ maillon est mis à jour en lui affectant l'adresse $@'$.

La sous-figure suivante 33c montre l'impact de la fonction *ajout_queue* sur la liste. Nous avons une nouvelle liste dont le nouveau maillon est bien inséré en queue de la liste.

Suppression d'un élément d'une liste chaînée

La création d'un élément d'une liste chaînée fait appel à une allocation dynamique de la mémoire. Lorsque certains (ou presque tous les éléments) de la liste ne sont plus utiles, il est donc nécessaire de restituer cette mémoire au système.

La logique des opérations est assez simple :

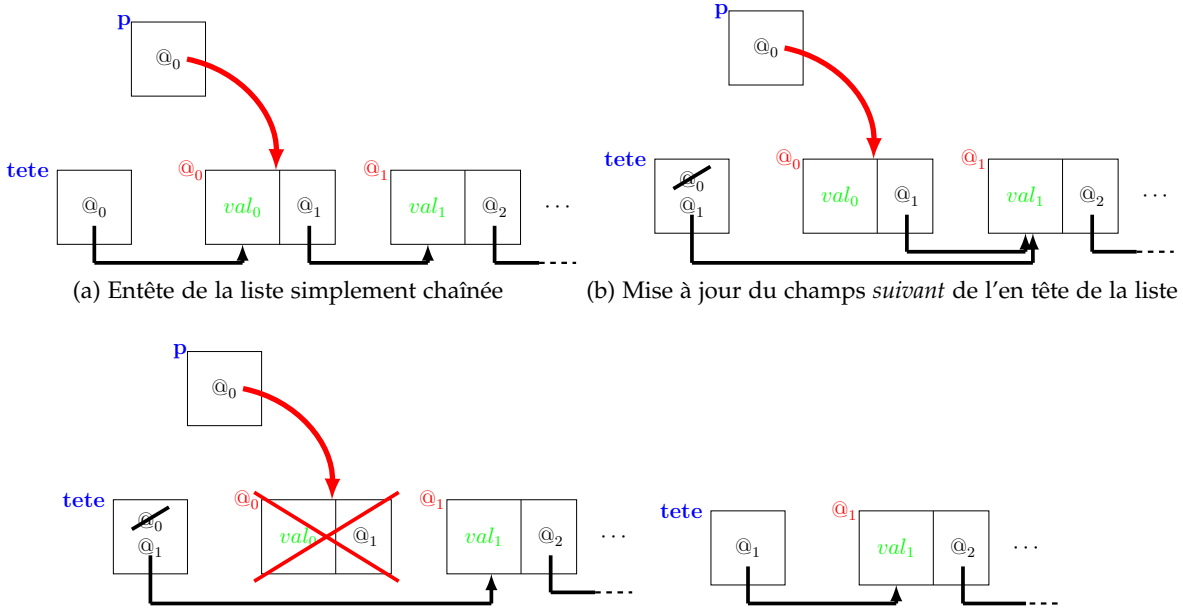
1. L'élément précédant celui qu'on doit supprimer doit adopter comme successeur le successeur de ce dernier.
2. L'espace mémoire réservé à l'élément supprimé de la liste doit être libéré.

Pareillement à l'insertion, la suppression dans une liste chaînée est autorisée au début, au milieu et en queue de la liste.

A. Suppression à l'entête de la liste : Nous donnons ci-dessous la fonction *void supprimer_tete(liste *& tete)* qui permet la suppression du premier élément de la liste.

```
void supprimer_tete(liste * & tete)
{
  if (tete != NULL)
  {
    liste * p = tete;
    tete = p->suivant;
    delete p;
  }
}
```

La figure ci-dessous 35 illustre les différentes étapes d'exécution de la fonction *supprimer_tete*.



(a) Entête de la liste simplement chaînée

(b) Mise à jour du champs *suivant* de l'en tête de la liste

(c) Libération de l'espace mémoire occupé par le maillon supprimé de la liste

(d) La liste chaînée après suppression

FIGURE 35 – Suppression d'un maillon à l'entête de la liste

Tel illustré dans la sous-figure 35a, initialement la variable *tete* pointe sur le premier élément ayant l'adresse mémoire $@_0$, qui pointe à son tour sur l'élément d'adresse $@_1$.

L'effet de l'instruction *tete = p->suivant* sur la liste se donne via la sous-figure 35b. L'entête est mis à jour par l'adresse du deuxième élément de la liste $@_1$. Par conséquent, le premier élément de la liste devient l'élément qui prenait auparavant la deuxième position.

La sous-figure suivante 35c montre l'impact de l'instruction *delete p* sur le maillon supprimé de la liste et l'espace mémoire qui lui a été réservé. La case mémoire d'adresse $@_0$ et qui contenait la valeur val_0 et l'adresse du successeur $@_1$ est libérée afin qu'elle puisse être exploitée par d'autres processus lancés par le système.

Enfin, nous avons la sous-figure 35d qui montre l'état de la liste après exécution de *supprimer_tete*. Nous avons une nouvelle liste dont l'entête pointe sur l'élément d'adresse $@_1$.

B. Suppression au milieu de la liste : La fonction ci-dessous dont l'entête *void supprimer_milieu(liste * pred)* permet la suppression d'un maillon au milieu de la liste, avec *pred* paramètre qui contient l'adresse de son prédécesseur.

```
void supprimer_milieu(liste * pred)
{
  if (pred->suivant != NULL)
  {
    liste * p = pred->suivant;
    pred->suivant = p->suivant;
    delete p;
  }
}
```

La figure ci-dessous 36 illustre les différentes étapes de suppression du maillon d'adresse $@_i$ et dont le prédécesseur et le successeur sont d'adresses respectives $@_{i-1}$ et $@_{i+1}$.

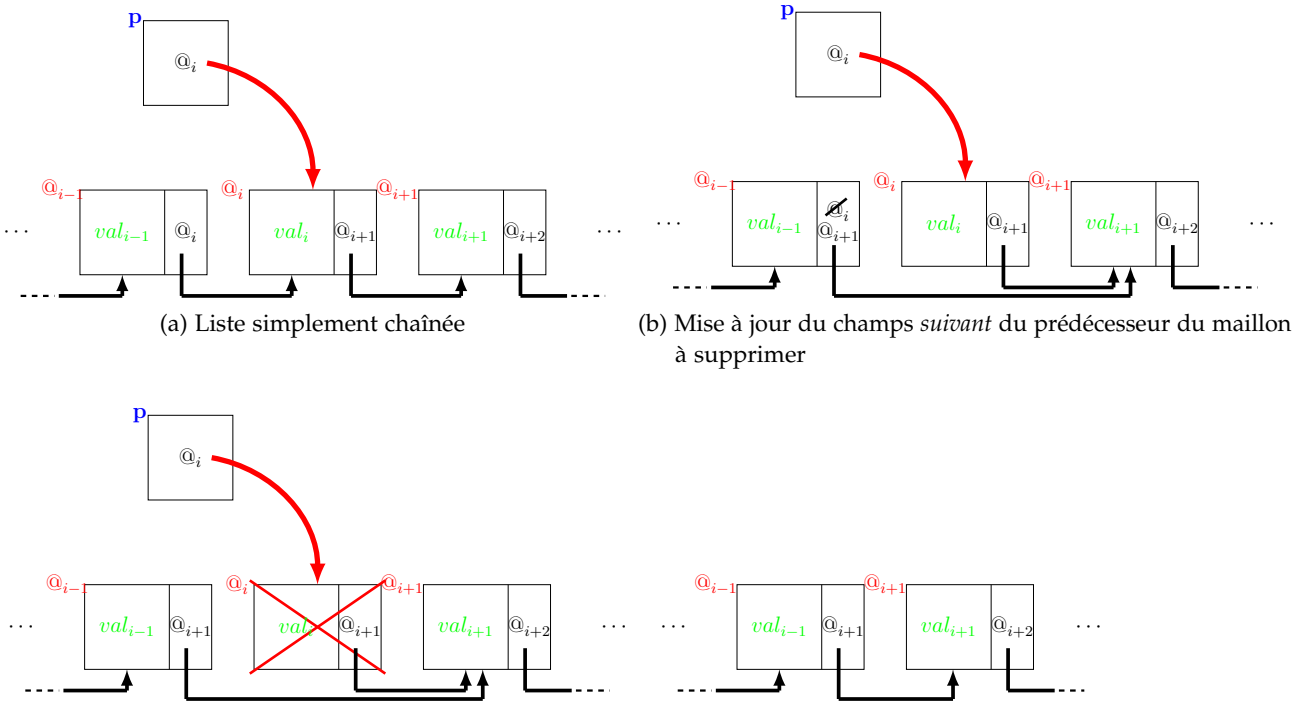
La sous-figure 36a illustre l'état initial de la liste. Le champ *suivant* du prédécesseur pointe sur l'élément ayant l'adresse mémoire $@_i$, qui pointe à son tour sur l'élément d'adresse $@_{i+1}$. La case mémoire qui correspond à la variable *p* contient l'adresse de l'élément à supprimer $@_i$, résultat de l'exécution de l'instruction *liste * p = pred->suivant*.

L'instruction *pred->suivant = p->suivant* est illustrée dans la sous-figure 36b. Le champ suivant du maillon $i-1$ est mis à jour par l'adresse de l'élément $i+1$. Par conséquent, l'élément $i+1$ de la liste devient l'élément successeur de l'élément $i-1$.

La sous-figure suivante 36c montre l'impact de l'exécution de *delete p* sur le maillon supprimé. La case mémoire d'adresse $@_i$ est libérée afin qu'elle puisse être exploitée par d'autres processus lancés par le système.

Enfin, nous avons la sous-figure 36d qui illustre l'état final de la liste après application de la fonction *void supprimer_milieu(liste *&, TypeElt)*.

C. Suppression en queue de la liste : Ci-dessous, nous décrivons la primitive *void supprimer_queue(liste * pred)* de suppression du dernier élément de la liste. Le paramètre *pred* correspond à l'adresse de l'avant dernier maillon de notre structure.



(a) Liste simplement chaînée

(b) Mise à jour du champs *suivant* du prédécesseur du maillon à supprimer

(c) Libération de l'espace mémoire occupé par le maillon supprimé de la liste

(d) La liste chaînée après suppression

FIGURE 36 – Suppression d'un maillon au milieu de liste

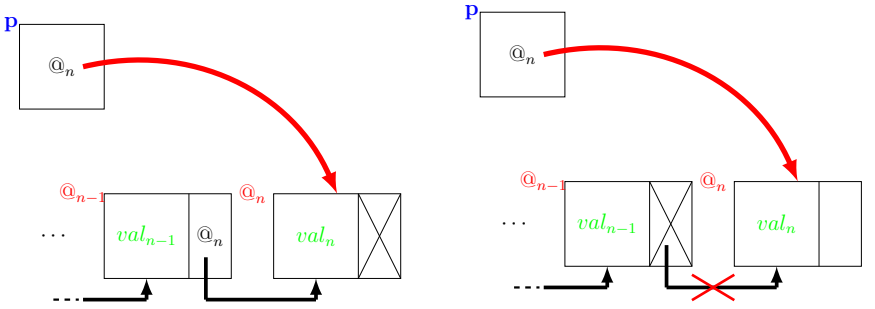
```
void supprimer_queue(liste * pred)
{
  if (pred->suivant != NULL)
  {
    liste * p = pred->suivant;
    pred->suivant = NULL;
    delete p;
    }
}
```

La figure ci-dessous 37 illustre les différentes étapes de suppression d'un maillon au milieu d'une liste chaînée.

On suppose qu'on désire supprimer le maillon d'adresse $@_i$, situé entre ceux d'adresses respectives $@_{i-1}$ et $@_{i+1}$.

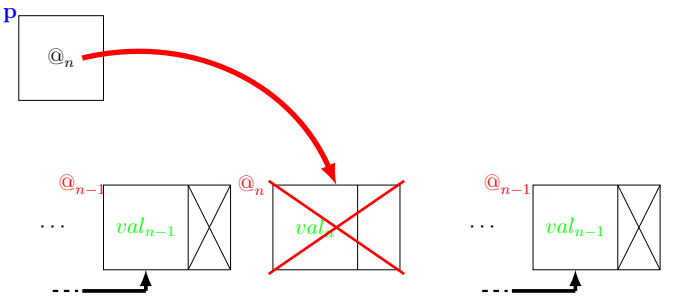
La sous-figure 37a illustre l'état initial de la liste. Le champ *suivant* de l'avant dernier élément *pred* pointe sur le dernier élément d'adresse $@_n$. La variable p contient l'adresse de l'élément à supprimer $@_n$, ce qui est le résultat de l'exécution de l'instruction *liste * p = pred->suivant*.

L'instruction *pred->suivant = NULL* est illustrée dans la sous-figure 37b. Le champ suivant du maillon $n-1$ est mis à *NULL*, puisqu'il ne doit pointer sur aucun autre élément.



(a) Liste simplement chaînée

(b) Mise à jour du champs *suivant* du prédécesseur du maillon à supprimer



(c) Libération de l'espace mémoire occupé par le maillon supprimé de la liste

(d) La liste chaînée après suppression

FIGURE 37 – Suppression d'un maillon au milieu de liste

La sous-figure suivante 37c montre l'impact de l'exécution de *delete p* sur le maillon supprimé de la liste. La case mémoire d'adresse $@_n$ est libérée afin qu'elle puisse être exploitée par d'autres processus lancés par le système.

Enfin, nous avons la sous-figure 37d qui illustre l'état de la liste après l'exécution de la fonction *void supprimer_queue(liste * &, TypeElt)*.

5.2.3 *Exemples d'applications sur les listes chaînées*

Soit une liste simplement chaînée d'entiers et dont la déclarons en C++ est comme suit :

```
struct liste
{
        int elt;
        liste * suivant;
}
liste * tete = NULL;
```

Calcul de la longueur de la liste chaînée

Calculer le nombre d'entiers chargés dans la liste revient à compter le nombre de maillons qui composent la liste. Ce qui donne lieu à la fonction *int calcul_longueur(liste * tete)* :

```
int calcul_longueur(liste * tete)
{
  int n=0; liste * p = tete;
  while(p!=NULL)
  {
    n++;
    p = p->suivant;
   }
 return n;
}
```

La boucle *while* permet de boucler sur les maillons à travers le champ *suivant*, jusqu'à atteindre la valeur *NULL* ; condition d'arrêt qui marque la fin de la liste.

Saisie des n éléments de la liste

La fonction suivante *void saisie_element(liste *&)* permet l'insertion des éléments à l'entête de la liste.

```
void saisie_element(liste *& tete)
{
  liste * p;
  TypeElt val;
  for(int i=0; i<n; i++)
  {
        lire(val); //saisir les valeurs du maillon
        ajout_tete(tete, val);
  }
}
```

5.2.4 *Listes doublement chaînées*

Une liste doublement chaînée est une liste qu'on peut parcourir dans les deux sens ; du premier élément au dernier et inversement. Ce qui impose l'ajout d'un autre champ à la structure du maillon qu'on appellera *precedent* et qui doit contenir l'adresse de son prédécesseur. La structure d'un maillon d'une liste doublement chaînée est illustrée dans la figure 38.

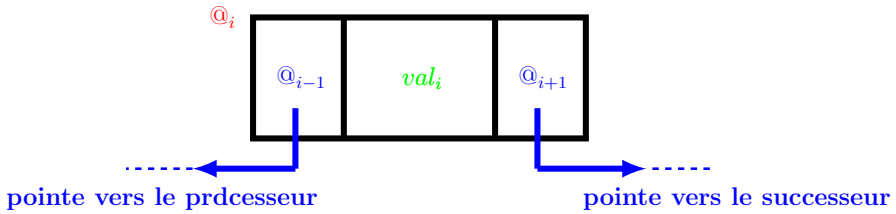


FIGURE 38 – Structure d'un maillon d'une liste doublement chaînée

Définition de la liste en C++

En suivant la structure du maillon illustrée dans la figure 38, la déclaration d'une liste doublement chaînée se donne en C++ comme suit :

```
struct listd
{
        TypeElt elt;
        listd * suivant;
        listd * precedent;
}
listd * tete = NULL;
```

Dans le cas où la liste contient n éléments, nous obtenons une liste qui a la structure que nous exposons via la figure ci-dessous 39.

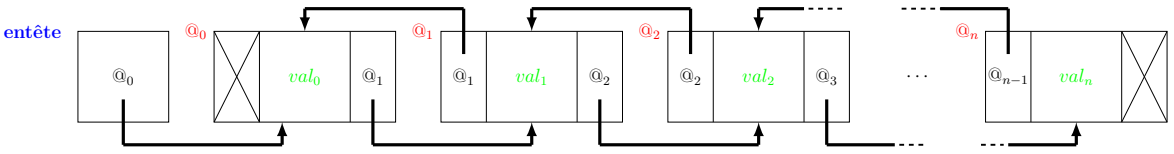


FIGURE 39 – Structure d'une liste doublement chaînée

Création d'un maillon

La fonction *liste * creer_maillon(TypeElt val)* alloue dynamiquement la mémoire nécessaire pour le nouveau maillon à travers l'instruction *listd * maillon = new listd*. Un test est réalisé afin de vérifier si l'allocation a été effectuée, afin de charger la valeur *val* qui doit être de type *TypeElt*. Les champs *precedent* et *suivant* sont initialisés à *NULL*.

La dernière instruction *return maillon* retourne l'adresse du maillon créé. Cette adresse va nous servir ensuite pour réaliser différents traitements sur ce maillon.

```
liste * creer_maillon(TypeElt val)
{
  listd * maillon = new listd;
  if (maillon!=NULL)
  {
    maillon->elt = val;
    maillon->suivant = NULL;
    maillon->precedent = NULL;
  }
  return maillon;
}
```

Ajout d'un élément

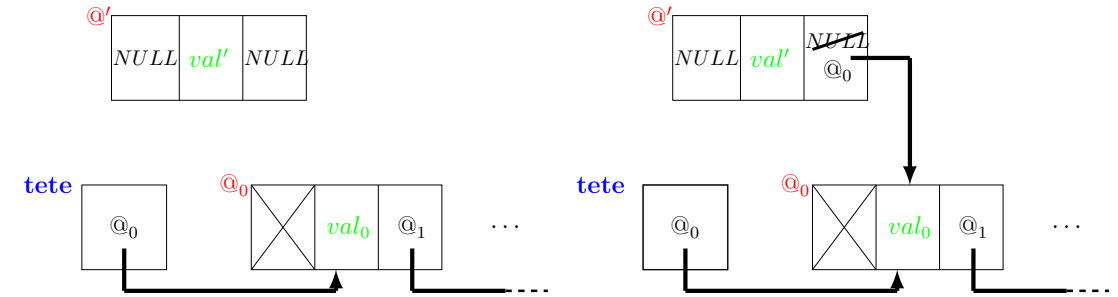
L'insertion d'un maillon dans une liste doublement chaînée nécessite plus d'étapes qu'une liste simple, puisque nous avons un nouveau champ à prendre en considération *precedent*. Ces étapes se donnent comme suit :

1. Il faut tout d'abord créer le nouvel élément et saisir ou stocker les valeurs.
2. Insérer cet élément dans la liste. La cohérence de la liste doit finalement être rétablie en indiquant que le nouvel élément est désormais le successeur de celui après lequel il va prendre place et le prédécesseur de celui avant lequel il doit être inséré.

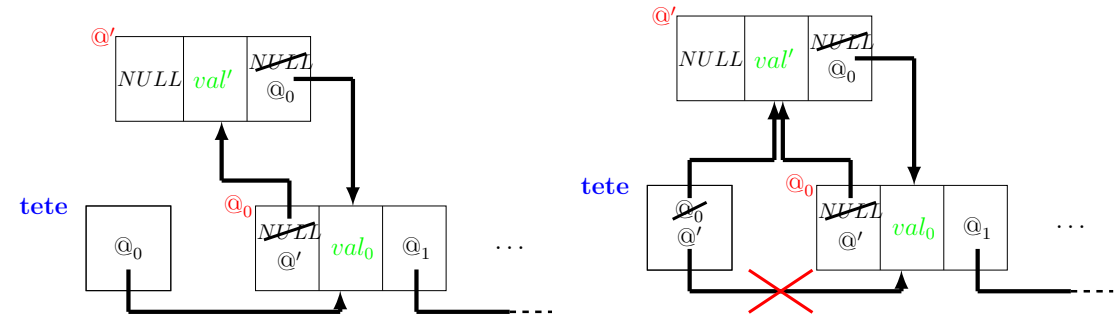
A. Insertion en tête de liste : L'insertion nécessite la mise à jour du champ suivant du nouveau maillon, du champ précédent du premier élément de la liste et de *tete*. La fonction ci-dessous *void ajout_tete(listd * & tete, TypeElt val)* permet la création du nouveau maillon et son insertion à l'entête de la liste :

```
void ajout_tete(listd *& tete, TypeElt val)
{
  liste *maillon=ceer_maillon(val);
  if (maillon != NULL)
  {
    maillon->suivant = tete;
    maillon->precedent = NULL;
    if (tete) tete->precedent =  maillon;
    tete = maillon;
  }
}
```

Nous résumons les instructions d'insertion via la figure 40.

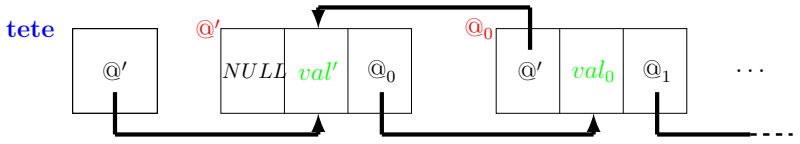


(a) Le nouveau maillon et la liste doublement chaînée (b) Mise à jour du champs *suivant* du nouveau maillon



(c) Mise à jour du champs *precedent* du nouveau maillon

(d) Mise à jour de l'entête



(e) La liste chaînée après insertion

FIGURE 40 – Insertion d'un maillon à l'entête de la liste

La sous-figure 40a illustre l'état de la liste doublement chaînée avec le nouveau maillon créé. L'adresse du premier maillon est $@_0$ et celle du nouveau maillon est $@'$.

La mise à jour du champ *suivant* du nouveau maillon se donne via la sous-figure 40b, résultat de l'instruction *maillon->suivant = tete*.

Après avoir mis le nouveau maillon en première position de la liste, il doit jouer le rôle de prédécesseur de l'élément d'adresse $@_0$. Cette étape est illustrée dans la figure 42c qui montre l'impact de l'instruction *tete->precedent=maillon*. Le champ *precedent* de l'élément d'adresse $@_0$ contient la valeur $@'$.

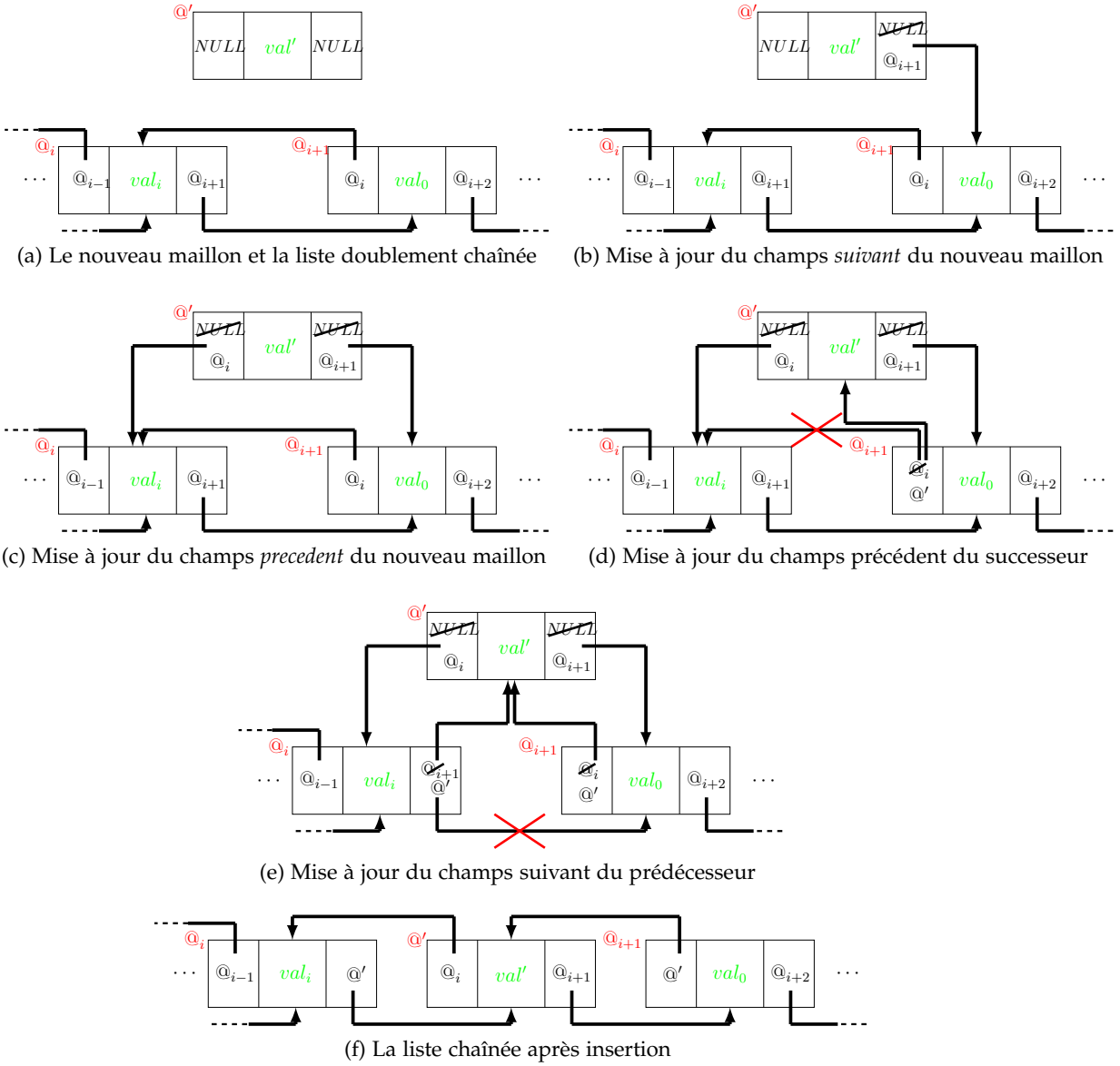
La dernière instruction exécutée dans la fonction est *tete=maillon* ; telle illustré dans la figure 40d. La variable *tete* est mise à jour et elle contient l'adresse $@'$.

L'état final de la liste après insertion se donne via la figure 40e.

B. Insertion au milieu de la liste : Idem que l'insertion au milieu d'une liste simple, en entrée de la fonction nous avons l'adresse du prédécesseur. Les différentes étapes d'insertion au milieu d'une liste doublement chaînée se donne via la fonction d'en tête *void ajout_milieu(listd * pred, TypeElt val)*.

```
void ajout_milieu(listd * pred, TypeElt val)
{
  listd *maillon=creer_maillon(val);
  if (maillon != NULL)
  {
    maillon->suivant = pred->suivant;
    maillon->precedent = pred;
    pred->suivant->precedent = maillon;
    pred->suivant = maillon;
  }
}
```

La figure 41 décrit en détails l'application de la fonction *ajout_milieu* qui insert un nouveau maillon d'adresse @′ au milieu de la liste.



(a) Le nouveau maillon et la liste doublement chaînée

(b) Mise à jour du champs *suivant* du nouveau maillon

(c) Mise à jour du champs *precedent* du nouveau maillon

(d) Mise à jour du champs précédent du successeur

(e) Mise à jour du champs suivant du prédécesseur

(f) La liste chaînée après insertion

FIGURE 41 – Insertion d'un maillon au milieu de la liste

La sous-figure 41a illustre l'état de la liste doublement chaînée et le nouveau maillon créé d'adresse @′. Le maillon doit être inséré entre les éléments d'adresses respectifs $@_i$ et $@_{i+1}$. Le paramètre *pred* est donc de valeur $@_i$.

La première étape consiste à mettre à jour les champs *precedent* et *suivant* du nouveau maillon. Cette mise à jour est réalisée à travers les deux instructions *maillon->suivant = pred->suivant* et *maillon->precedent = pred*. L'impact de ces deux instructions est illustrée dans les deux sous-figures respectives 41b et 41c. Nous obtenons les champs pointeurs du nouveau maillon *precedent* $= @_i$ et *suivant* $= @_{i+}$.

La sous-figure 41d montre le résultat de l'instruction *pred->suivant->precedent=maillon* ; le nouveau maillon doit être le prédécesseur de l'élément d'adresse $@_{i+1}$. Par conséquent, le champ *precedent* de l'élément d'adresse $@_{i+1}$ contient la valeur @′.

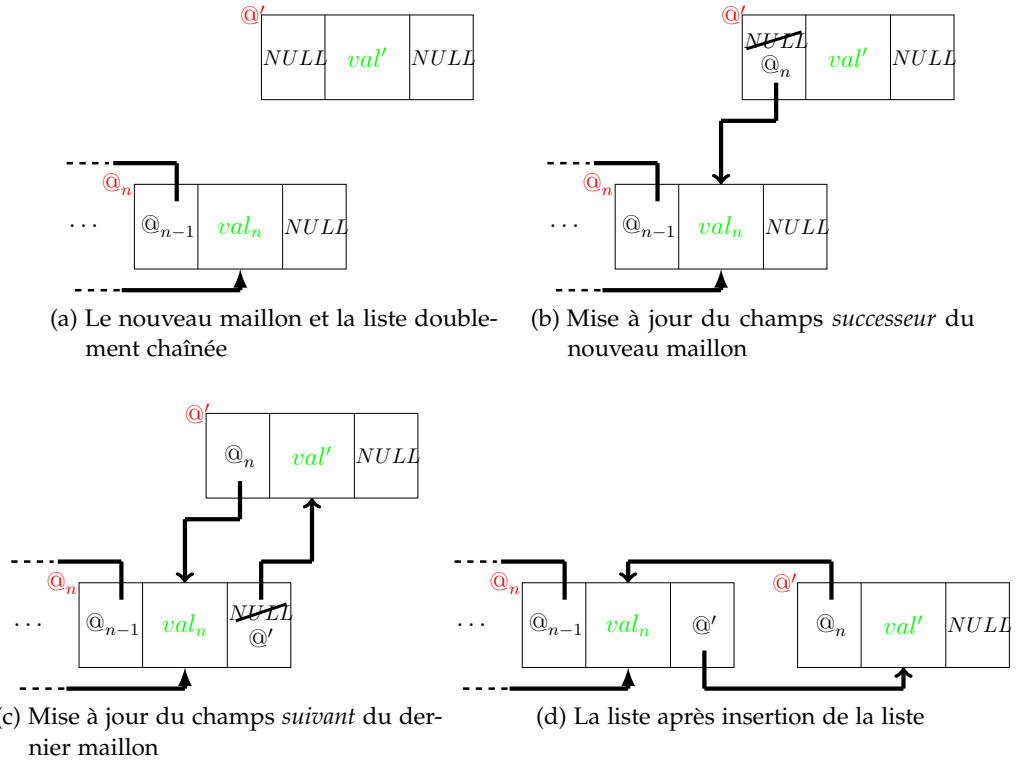
La dernière instruction exécutée dans la fonction est *pred->suivant = maillon ;* nous affectons au champs *suivant* du $i^{ème}$ élément la valeur @′ (l'adresse du nouveau maillon) (voir la figure 41e).

L'état final de la liste se donne via la sous-figure 41f.

C. Insertion en queue de la liste : La fonction *void ajout_queue(listd * pred, TypeElt val)* donnée ci-dessous permet l'insertion du nouveau maillon en queue de la liste, avec *p* pointeur sur l'avant dernier élément.

```
void ajout_queue(listd * pred, TypeElt val)
{
  listd *maillon=creer_maillon(val);
  if (maillon != NULL)
  {
    maillon->precedent = pred;
    maillon->suivant = NULL;
    pred->suivant = maillon;
  }
}
```

La figure 42 représente les différentes étapes d'insertion d'un maillon en queue d'une liste doublement chaînée et qui contient $n + 1$ éléments.



(a) Le nouveau maillon et la liste doublement chaînée

(b) Mise à jour du champs *successeur* du nouveau maillon

(c) Mise à jour du champs *suivant* du dernier maillon

(d) La liste après insertion de la liste

FIGURE 42 – Insertion d'un maillon en tête de liste

La figure 42a représente la fin d'une liste doublement chaînée dont le dernier élément est d'adresse $@_n$ et le nouveau maillon d'adresse @′.

La première étape consiste à mettre à jour le champs *precedent* du nouveau maillon avec l'adresse $@_n$, et ce à travers l'instruction *maillon->precedent = pred* où $pred = @_n$ (voir la figure 42b). Le champ suivant est *NULL* puisque le nouveau maillon sera le dernier et ne doit pointer sur aucun successeur.

La figure 42c illustre l'impact de l'exécution de l'instruction *pred->suivant=maillon* sur la liste ; le $n^{'eme}$ élément a comme successeur le nouveau maillon.

Le résultat final de la fonction *ajout_queue* est illustré dans la figure 42d.

Suppression d'un élément d'une liste chaînée

La logique des opérations qui permettent la suppression d'un élément d'une liste doublement chaînée est assez simple :

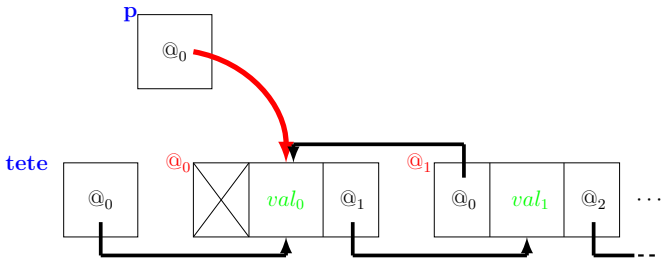
1. L'élément précédant celui qu'on doit supprimer doit adopter comme successeur le successeur de ce dernier.
2. L'élément succédant celui qu'on doit supprimer doit adopter comme prédécesseur le prédécesseur de ce dernier.
3. L'espace mémoire déjà réservé à l'élément supprimé doit être libéré.

Effectivement, après la mise à jour des pointeurs *precedent* et *suivant*, nous devons libérer l'espace mémoire occupé par le maillon supprimé de la liste. Ce qui nécessite le chargement de l'adresse du maillon dans une variable temporaire noté *p*.

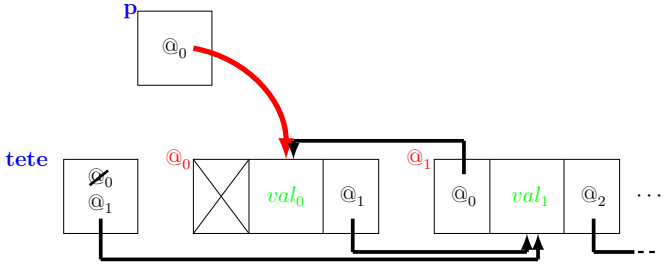
A. Suppression à l'entête de la liste : La suppression à l'entête de la liste désigne la suppression du premier élément de la liste. Elle est décrite via la fonction *void supprimer_tete(listd * & tete)*. L'adresse de ce dernier est chargé dans la variable pointeur *p*. Par la suite, l'entête et le champ *precedent* du deuxième élément sont mis à jour.

```
void supprimer_tete(listd * & tete)
{
  if (tete != NULL)
  {
    liste * p = tete;
    tete = p->suivant;
    p->precedent = NULL;
    delete p;
  }
}
```

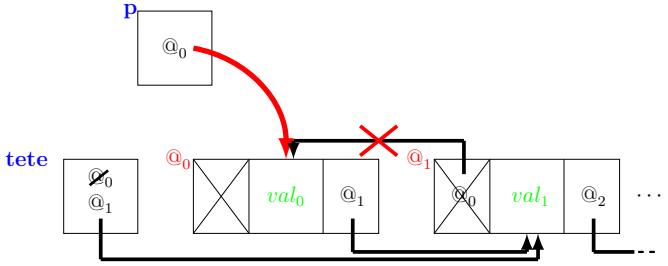
La figure 43 décrit en détail les différentes étapes de suppression du maillon en première position d'une liste doublement chaînée.



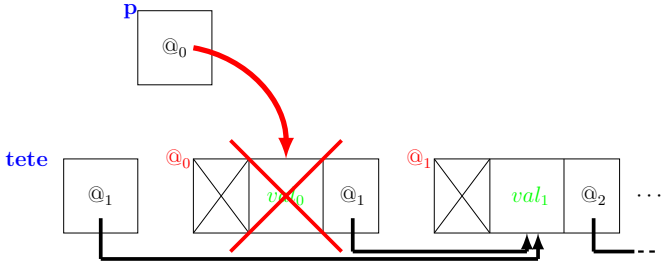
(a) L'entête de la liste doublement chaînée



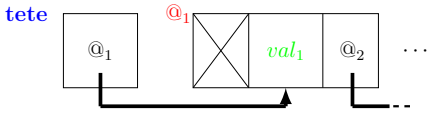
(b) Mise à jour de la variable *entete*



(c) Mise à jour du champ *precedent* du maillon succédant celui à supprimer



(d) Libération de l'espace mémoire occupé par le maillon supprimé



(e) La liste chaînée après suppression

FIGURE 43 – Suppression d'un maillon à l'entête de la liste

La première sous-figure 43a illustre l'entête d'une liste doublement chaînée. Nous avons la variable entête $tete = @_0$ et la variable temporaire $p = @_0$ qui pointe sur l'élément à supprimer.

La sous-figure 43b illustre le changement de la valeur de *tete* qui pointe sur le deuxième élément d'adresse $@_1$. Elle met en avant l'impact de l'instruction *tete = p->suivant*.

L'instruction qui suit est *p->precedent=NULL*, son résultat est illustré dans la figure 43c. L'élément d'adresse $@_1$ n'a plus de prédécesseur ; le champ *precedent* est mis à *NULL*.

Après la mise à jour des chaînages, il faut libérer l'espace mémoire occupé par le maillon supprimé et dont l'adresse est dans p. Cette action est réalisée par l'instruction *delete p* et illustrée dans la figure 43d.

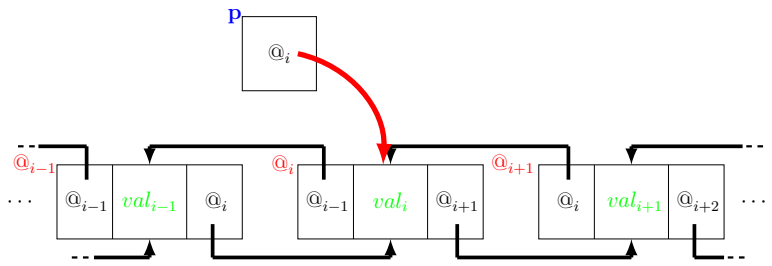
L'état final de la liste après suppression est donné dans la figure 43e.

B. Suppression au milieu de la liste : La fonction ci-dessous *void supprimer_milieu(listd * p)* réalise la suppression d'un maillon au milieu d'une liste doublement chaînée et dont l'adresse est représentée par le paramètre d'entrée p.

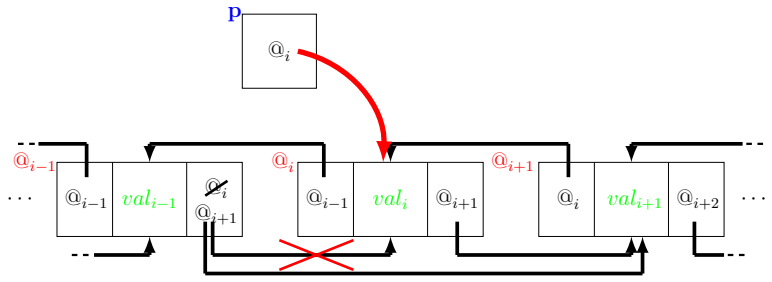
```
void supprimer_milieu(listd * p)
{
    p->precedent->suivant = p->suivant;
    p->suivant->precedent = p->precedent;
    delete p;
}
```

L'exécution de cette fonction est illustrée dans la figure 44.

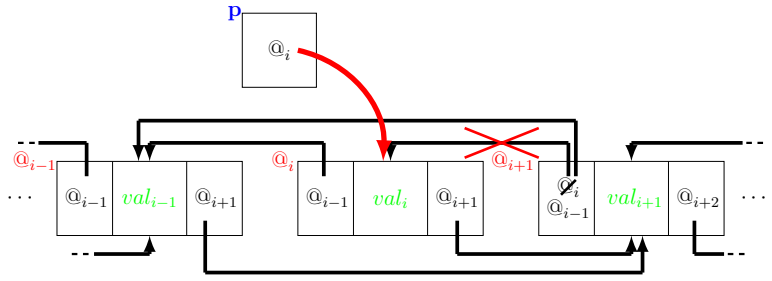
L'élément à supprimer est d'adresse $@_i$ entre un élément prédécesseur et un élément successeur d'adresses respectives $@_{i-1}$ et $@_{i+1}$. La variable p contient la valeur $@_i$; l'adresse du maillon à supprimer.



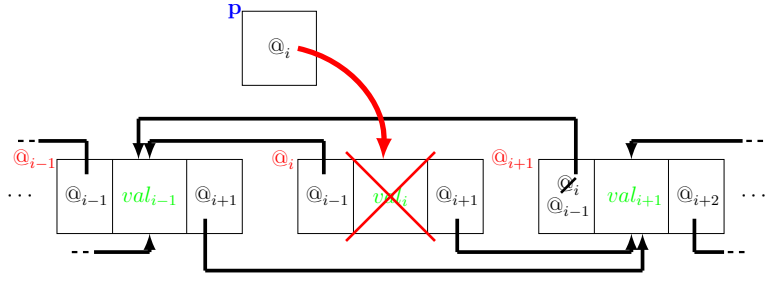
(a) La liste doublement chaînée



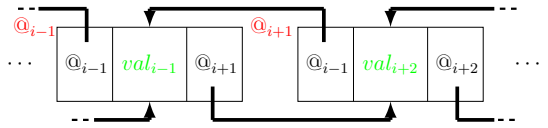
(b) Mise à jour du champ *suivant* du maillon précédant celui à supprimer



(c) Mise à jour du champ *precedent* du maillon succédant celui à supprimer



(d) Libération de la mémoire du maillon supprimé



(e) La liste chaînée après insertion

FIGURE 44 – Suppression d'un maillon au milieu d'une liste doublement chaînée

La sous-figure 44a montre la liste doublement chaînée et la variable p qui pointe sur le maillon à supprimer d'adresse.

La sous-figure 44b représente l'exécution de l'instruction *p->precedent->suivant = p->suivant* et qui permet d'affecter la valeur $@_{i+1}$ au champ *suivant* du maillon d'adresse $@_{i-1}$.

L'instruction suivante *p->suivant->precedent = p->precedent* assigne le champ *precedent* de l'élément d'adresse $@_{i-1}$ avec la valeur $@_{i+1}$. Elle est illustrée dans la figure 44c.

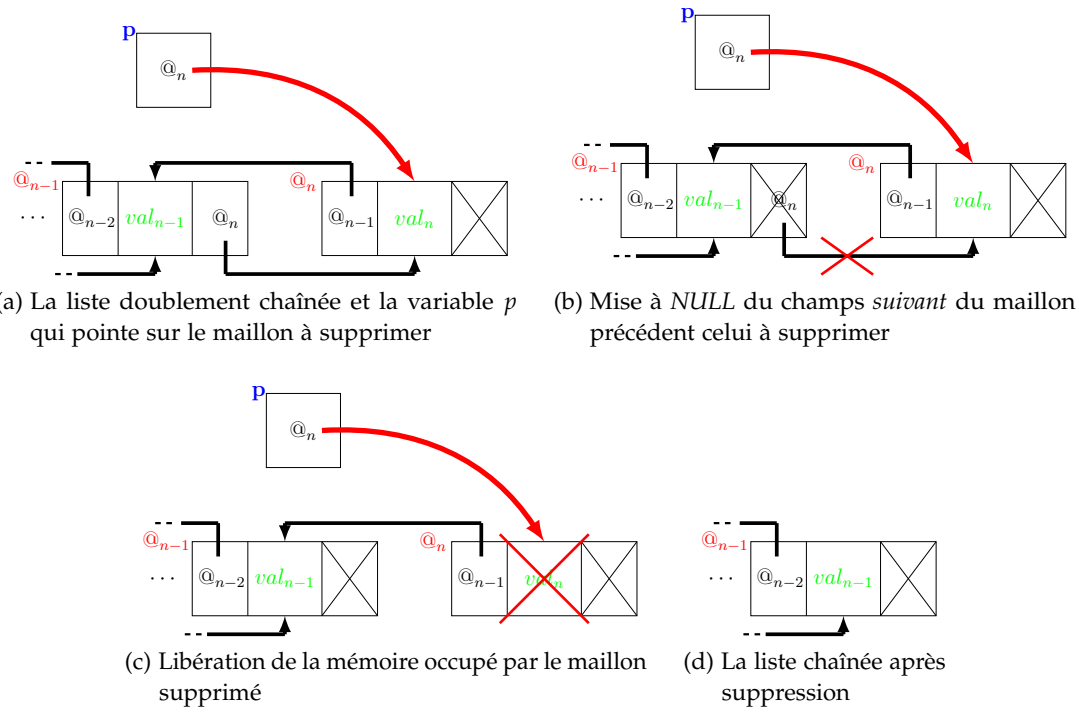
La dernière étape consiste à libérer l'espace mémoire occupé par le maillon supprimé de la liste, et ce en exécutant l'instruction *delete p*. Elle est illustrée dans la figure 44d.

La dernière sous-figure 44e représente l'état final de la liste après exécution de la fonction.

C. Suppression en queue de la liste : La fonction ci-dessous *void supprimer_en_queue(listd * p)* réalise la suppression du dernier maillon de la liste doublement chaînée et dont l'adresse est l'entrée *p* de la fonction.

```
void supprimer_en_queue(listd * p)
{
    p->precedent->suivant = NULL;
    delete p;
}
```

L'exécution de cette fonction est illustrée dans la figure 45. L'élément à supprimer est d'adresse $@_n$. Elle est contenue dans la variable pointeur *p*.



(a) La liste doublement chaînée et la variable *p* qui pointe sur le maillon à supprimer

(b) Mise à *NULL* du champs *suivant* du maillon précédent celui à supprimer

(c) Libération de la mémoire occupé par le maillon supprimé

(d) La liste chaînée après suppression

FIGURE 45 – Suppression du maillon en queue d'une liste doublement chaînée

La sous-figure 45a illustre la fin de la liste doublement chaînée et la variable *p* qui pointe sur le maillon à supprimer.

La sous-figure 45b représente l'exécution de l'instruction *p->precedent->suivant = NULL*. Elle permet d'affecter la valeur *NULL* au champ *suivant* du maillon d'adresse $@_{n-1}$.

La dernière étape consiste à libérer l'espace mémoire occupé par le maillon supprimé de la liste, en exécutant l'instruction *delete p*. Elle est illustrée dans la figure 45c.

La dernière sous-figure 45d représente l'état final de la liste.

5.2.5 *Piles*

Une pile est une structure de données où les insertions et les suppressions se font toutes du même coté. Une telle structure est aussi appelée LIFO (Last In First Out).

Représentation d'une pile en C++

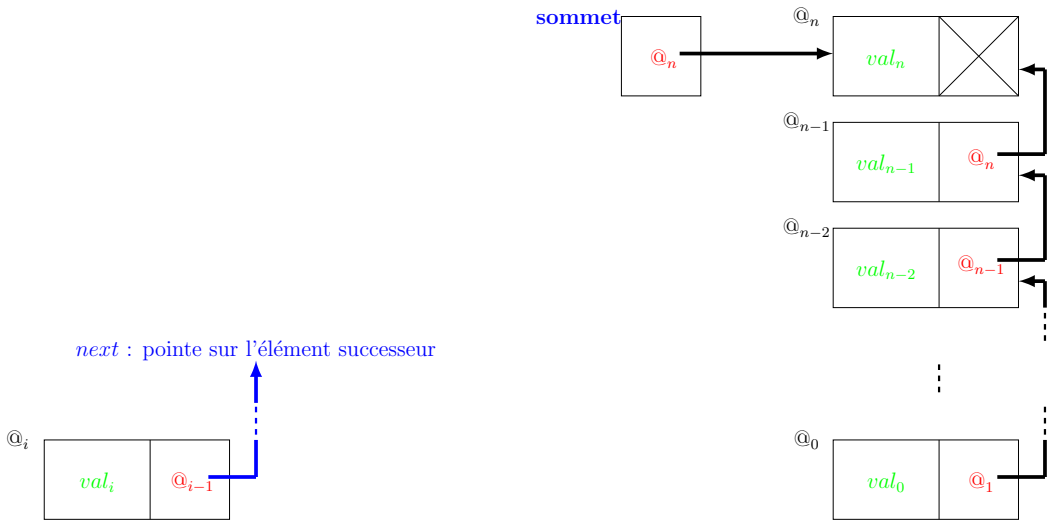
Nous exposons dans cette section la structure d'une pile basée sur les listes simplement chaînées. Effectivement, une pile est déclarée sous forme d'une liste simplement chaînée avec l'élément sentinelle *sommet*.

Les éléments de la pile sont chaînés entre eux, et le sommet d'une pile non vide est le premier de la liste ; dans le cas où la pile est vide *sommet* est *NULL*.

La définition de la pile en C++ se donne comme suit :

```
struct pile
{
        TypeElt elt;
        pile * next;
};
pile * sommet=NULL;
```

La figure ci-dessous 46 illustre la structure d'une pile.



(a) Elément d'une pile (b) La structure d'une pile

FIGURE 46 – Structure d'une pile en utilisant une liste simplement chaînée

La figure 46a décrit la structure d'un élément d'une pile qui a la structure d'une liste simplement chaînée, donnée à son tour dans la figure 46b. L'élément sentinelle est *sommet* qui pointe sur le sommet de la pile. Les éléments ont des adresses numérotées de 0 à n, sachant que l'adresse $@_0$ correspond à la première insertion, l'adresse $@_1$ à la deuxième insertion, etc. La variable *sommet* pointe sur l'élément d'adresse $@_n$; adresse du dernier élément inséré. Sachant que, l'insertion n'est autorisée qu'au sommet de la pile.

Nous précisons qu'aucun accès n'est autorisé aux autres éléments de la pile hors le sommet.

Les piles admettent deux primitives, à savoir *empiler* et *dpiler*, qui permettent respectivement l'empilement et le dépilement d'un élément. Bien sûr, les deux primitives ne sont appliquées qu'au sommet de la pile [1].

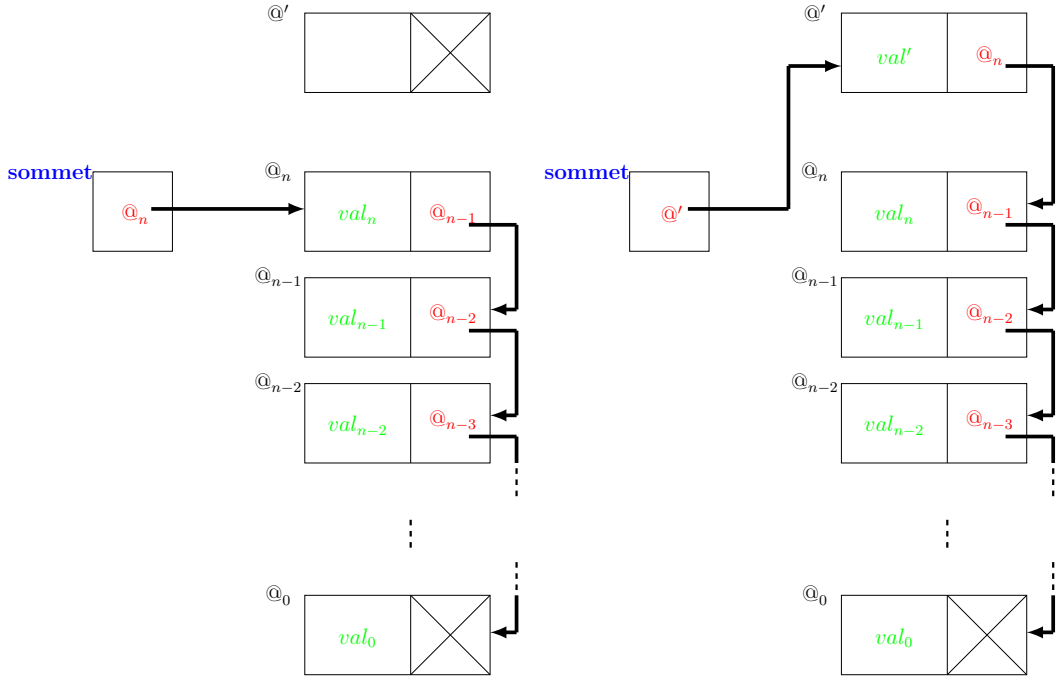
Empiler un élément

Empiler un nouvel élément dans la liste revient à insérer un nouveau maillon à l'entête de la liste. La fonction d'empilement *void empiler(pile * & sommet, TypeElt)* est similaire à celle de l'insertion à l'entête d'une liste simplement chaînée décrite dans section 5.2.2.

1. Le sommet de la pile correspond à l'entête de la liste simplement chaînée

```
void empiler(pile *& sommet, TypeElt val)
{
  pile *E = new pile;
  if (E)
  {
    E->elt = val;
    E->next = sommet;
    sommet = E;
  }
}
```

La figure ci-dessous 47 illustre l'exécution de la fonction *empiler*



(a) La pile avant l'epilement de l'élément d'adresse @'

(b) La pile après empilement

FIGURE 47 – Insertion d'un élément dans une pile

Nous avons dans la figure 47a le maillon à empiler d'adresse @' et la pile dont le sommet est d'adresse $@_n$.

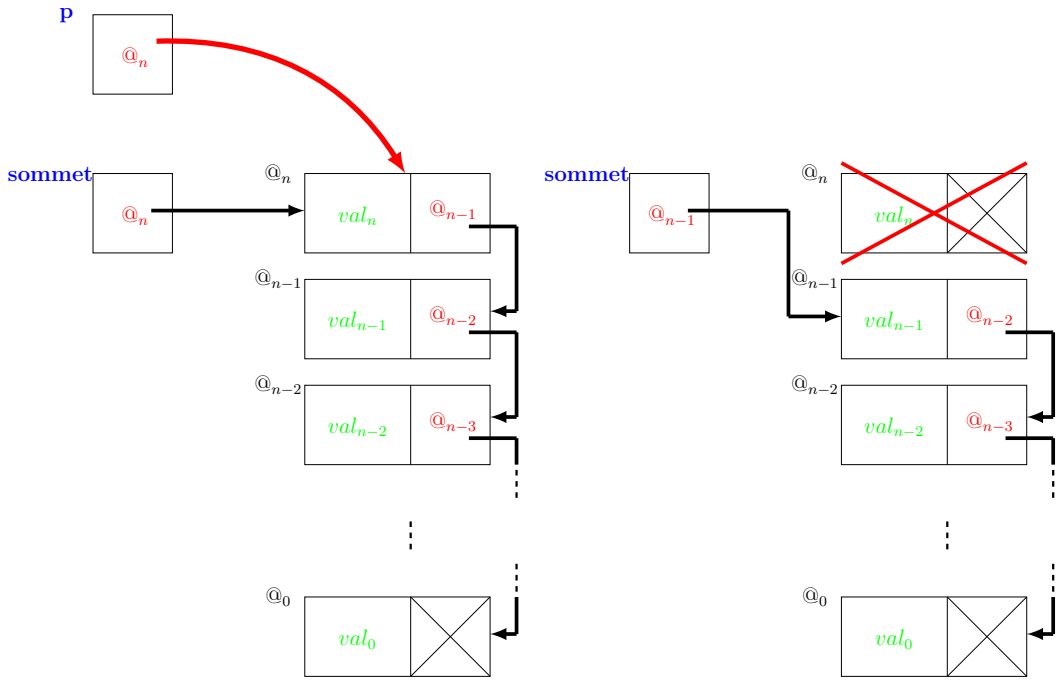
Après exécution de la fonction, situation illustrée dans la sous-figure 47b, l'élément sentinelle *sommet* est mis à jour. L'élément au sommet de la pile est le nouveau qui pointe sur l'élément d'adresse $@_n$

Dépiler un élément

Dépiler un élément de la pile revient à supprimer l'élément au sommet de la pile. La fonction correspondant *void depiler(pile *& sommet)* est similaire à la fonction de suppression d'un maillon au sommet d'une liste simplement chaînée (voir la section 5.2.2).

```
void depiler(pile *& sommet)
{
  if (sommet == NULL) return;
  else
  {
    pile *p = sommet;
    sommet = sommet->lien;
    delete p;
  }
}
```

La figure 48 illustre l'exécution de la fonction *void depiler(pile *& sommet)*.



(a) La pile avant l'empilement (b) La pile après le dépilement

FIGURE 48 – Suppression d'un élément dans une pile

La sous-figure 51a montre la mise à jour de l'élément sentinelle *sommet* à $@_{n-1}$. Nous avons par la suite, la libération de la mémoire occupée par l'élément supprimé.

L'état final de la pile se donne dans la figure 48bavec $sommet = @_{n-1}$.

Valeur du sommet de la pile

Nous avons ci-dessous la fonction *TypeElt depiler(pile * sommet)* qui retourne la valeur chargée au sommet de la pile.

Si on considère la figure 46b, la fonction retourne la valeur val_n. Dans le cas où la pile est vide elle retourne la valeur *val_pile_vide*, une valeur fixé par le programmeur et qui exprime l'état « vide »de la pile.

```
TypeElt depiler(pile * sommet)
{
  if (sommet == NULL) return val_pile_vide;
  else return sommet->elt;
}
```

5.2.6 *Files*

Dans le cas d'une file, on fait les adjonctions (insertions) à une extrémité, les accès et les suppressions de l'autre extrémité. Les files sont aussi appelées FIFO (First In First Out). En d'autres termes, l'accès à un élément de la file suit le raisonnement d'une file d'attente « premier venu premier servi »

Représentation d'une file en C++

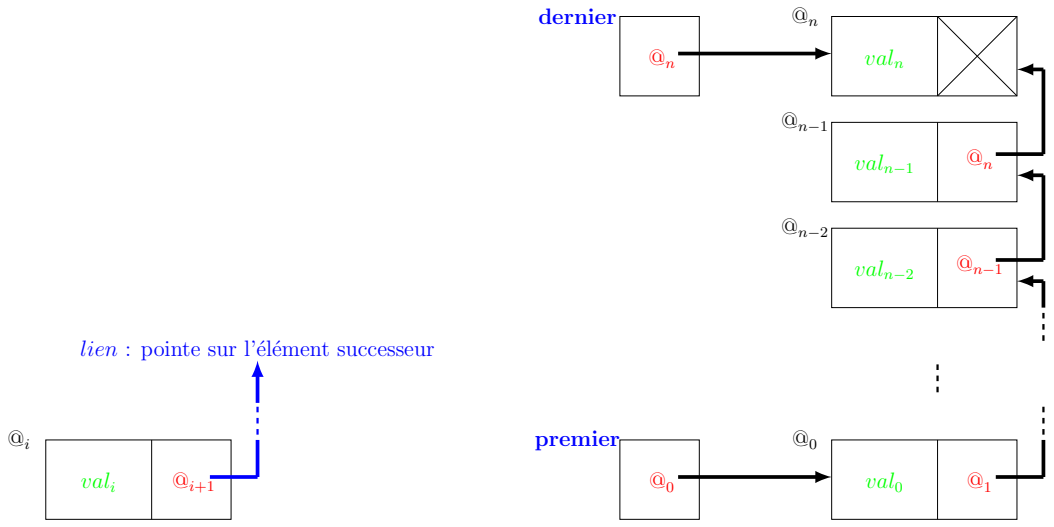
Dans notre cours, la structure d'une file repose sur les listes chaînées simples. Une file possède deux éléments sentinelle *premier* et *dernier* :

premier : pointe sur le premier élément de la file.

dernier : pointe sur le dernier élément de la file.

```
struct file
{
        TypeElt elt;
        file * lien;
};
file * premier=NULL;
file * dernier=NULL;
```

La figure 49 montre la structure d'un composant de la file et la structure d'une file avec un chaînage simple.



(a) Elément d'une file (b) La structure d'une file

FIGURE 49 – Structure d'une file en utilisant une liste simplement chaînée

La sous-figure 49a est une illustration d'un élément de la file ; il contient un champ *valeur* et un autre adresse du successeur *lien*.

La structure de la file est donnée dans la sous-figure 49b. Les éléments sont enfilés de $@_0$ à $@_n$, raison pour laquelle *premier* pointe sur l'élément d'adresse $@_0$ et *dernier* sur l'élément d'adresse $@_n$.

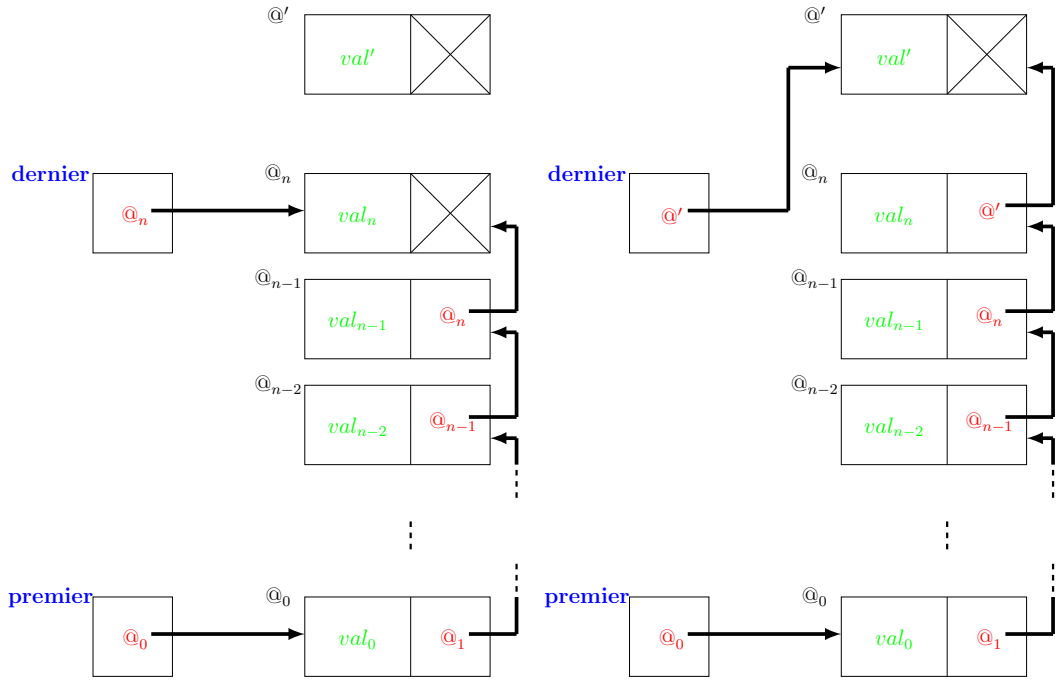
Enfiler un élément dans la file

L'insertion d'un élément dans une file (ou l'enfilement) se fait en queue de la file. Par conséquent la variable *dernier* doit être mise à jour avec l'adresse du nouvel élément. La primitive *enfiler* se donne comme suit :

```
void enfiler(file *& dernier, file *& premier, TypeElt val)
{
  file * E = new file;
  if (E)
  {
    E->elt = val;
    E->lien = NULL;
    if (dernier==NULL)
    {
                dernier = E;
                premier=dernier;
        }
    else dernier->lien = E;
  }
}
```

Nous constatons que la fonction admet le paramètre *premier* par référence en plus du paramètre *dernier*, et ce afin de traiter le cas où la file est vide *premier* = *dernier* = *NULL*.

La figure 50 illustre l'exécution de la fonction sur une file non-vide *premier* ≠ *NULL* et *dernier* ≠ *NULL*.



(a) La file avant l'insertion de l'élément d'adresse @′

(b) La file après insertion

FIGURE 50 – Insertion d'un élément dans une file

Nous avons dans la sous-figure 50a l'élément à enfiler d'adresse @′ et la file qui contient $n+1$ élément dont les adresses varient de $@_0$ à $@_n$.

La sous-figure 50b illustre l'application de *enfiler*. La variable *dernier* est mise à jour, ainsi que le champ *lien* du maillon d'adresse $@_n$. Par conséquent, *dernier* pointe sur @′, successeur de l'élément d'adresse $@_n$.

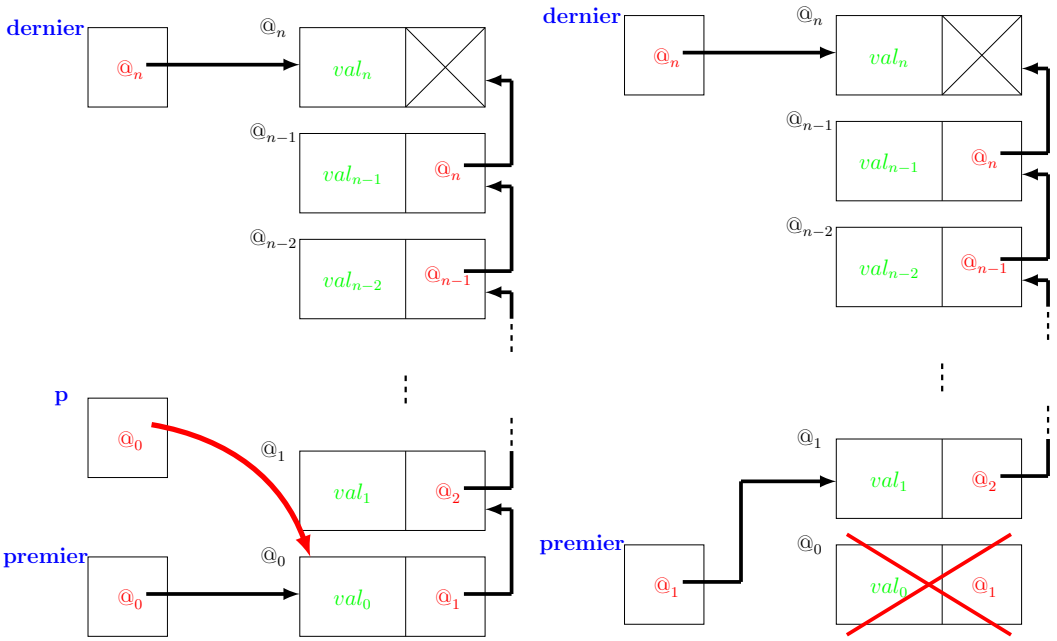
Défiler un élément de la file

La suppression d'un élément d'une file (ou le défilement) se fait en tête de la liste comme suit :

```
void defiler(file *& premier,file *& dernier)
{
        file *p = premier;
    if (premier)
    {
        premier = premier->lien;
        delete p;
                if (!dernier) dernier=NULL;
        }
}
```

La fonction admet le paramètre *dernier* par référence en plus du paramètre *premier*, et ce afin de traiter le cas où la file ne contient qu'un seul élément $premier = dernier \neq NULL$.

La figure 51 illustre l'action *défiler* appliquée sur une file qui contient plus d'un élément.



(a) La file avant suppression (b) La file après suppression

FIGURE 51 – Suppression d'un élément dans une file

Nous considérons dans la figure 51a une file qui contient $n+1$ éléments.

Puisque nous ne pouvons supprimer que le premier élément de la file, celui d'adresse $@_0$ est l'élément qui sera éliminé de la file (l'élément d'adresse $@_0$ est le premier enfilé dans la file). La sous-figure 50b montre le résultat de l'application de la primitive sur la file, avec $premier = @_1$ et l'espace mémoire occupé par l'élément supprimé est libéré.

5.3 STRUCTURES ARBORESCENTES

5.3.1 *Arbres*

Un arbre est un ensemble de nœuds organisés de façon hiérarchique, à partir d'un nœud distingué appelé racine.

Une propriété intrinsèque de la structure d'arbre est la récursivité et les définitions caractéristiques des arbres s'écrivent très naturellement de manière récursive.

Arbres binaires

Soit la définition 5.3.1 ci-dessous :

Définition 5.3.1 *[Froidevaux et al., 1993] Un arbre binaire est soit vide (noté ∅), soit de la forme $B = \langle \circ, B_1, B_2 \rangle$, où B_1 et B_2 sont des arbres disjoints et $\circ$ est un nœud appelé racine.*

Cette définition est récursive ; elle peut s'écrire sous la forme de l'équation : $B = \varnothing + \langle \circ, B, B \rangle$, tel que B représente l'ensemble des arbres binaires, $\varnothing$ représente l'arbre vide et $\circ$ désigne un nœud.

1. Représentation d'un arbre binaire en C++

Un arbre est une structure arborescente avec un élément sentinelle appelé *racine*. Dans cette section nous introduisons les arbres binaires où chaque nœud ne peut avoir plus de deux fils : fils gauche et fils droit.

Par exemple, la figure 52 prise de[Froidevaux et al., 1993] représente un arbre binaire qui modélise l'expression arithmétique $(x - (2 * y)) + (x + (y/z) * 3)$.

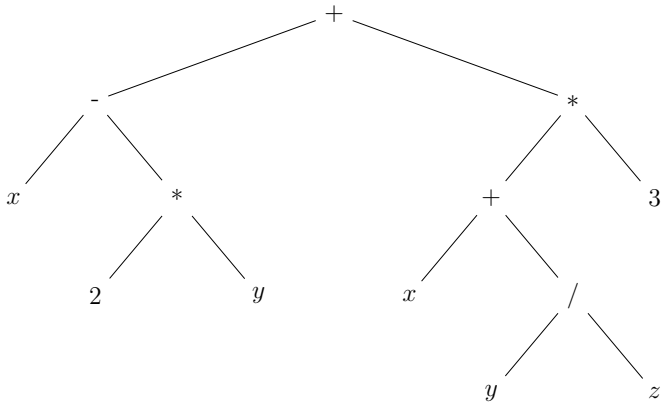


FIGURE 52 – Expression arithmétique $(x - (2 * y)) + (x + (y/z) * 3)$

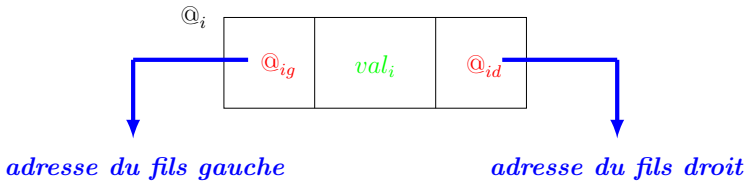
Nous remarquons que la racine correspond à l'opérateur **+**, dont le fils gauche et le fils droit donnent lieu respectivement à :

— un sous-arbre gauche qui modélise l'expression $x - (2 * y)$,
— et un sous-arbre droit qui modélise l'expression $(x + (y/z) * 3$.

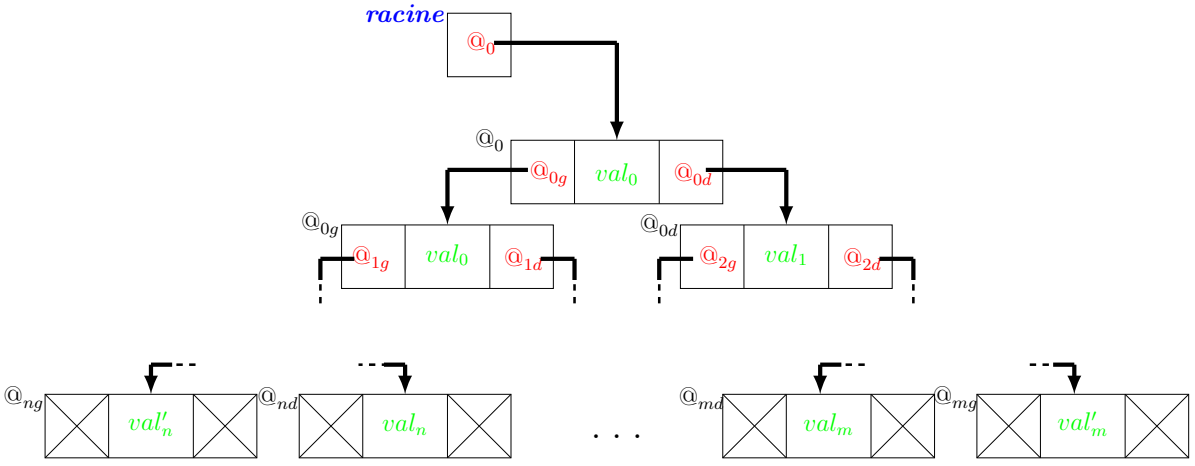
La déclaration en C++ d'un arbre binaire se donne comme suit :

```
struct noeud
{
        TypeElt elt;
        noeud * gauche;
        noeud * droit;
}
noeud * racine=NULL;
```

La figure 53 donne la structure d'un arbre binaire et de ses nœuds.



(a) Nœud d'un arbre binaire



(b) Structure d'un arbre binaire avec chaînage

FIGURE 53 – Structure d'un arbre binaire

La sous-figure 53a illustre la composition d'un nœud ; il comporte un champ pointeur qui pointe sur le fils gauche, la valeur à charger et un autre champ pointeur sur le fils droit.

Nous montrons via la figure 53b la représentation en mémoire d'un arbre binaire. L'élément *racine* est d'adresse $@_0$; il a un fils gauche d'adresse $@_0g$ et un fils droit $@_0d$. Ces derniers, peuvent à leur tour avoir des fils gauche et droit. Les nœuds ayant les adresses $@_{ng}$, $@_{nd}$, $@_{mg}$ et $@_{md}$ n'ont pas d'enfants (ni fils gauche, ni fils droit) sont appelés *feuilles*.

Hors le nœud *racine* et les nœuds *feuille*, le nœud est dit *interne*.

Pour une meilleur compréhension de la structure nous introduisons ci-dessous un exemple 5.3.3 qui introduit un arbre binaire d'entiers et sa représentation en mémoire.

Exemple 5.3.2 (arbre binaire d'entiers) *Soit l'arbre binaire d'entier illustré dans la figure 54 ci-dessous*

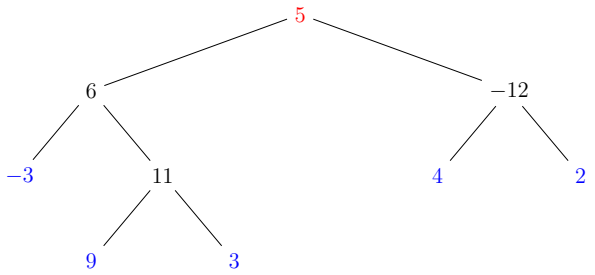


FIGURE 54 – Arbre binaire d'entiers

La représentation en mémoire de l'arbre est représenté par la figure 55 ci-dessous :

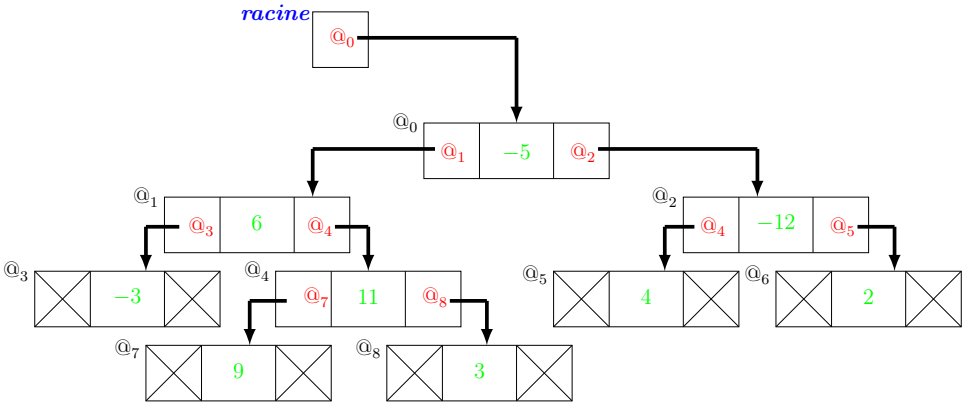


FIGURE 55 – Représentation en mémoire de l'arbre de la figure 54

2. Vérifier si l'arbre est vide

Un arbre binaire vide est marqué par la variable *racine* qui doit être de valeur $NULL$.

La primitive *est_vide* retourne *true* si l'arbre est vide et $false$ sinon.

```
bool est_vide(noeud * racine)
{
  return (racine==NULL);
}
```

Récupérer l'un des deux fils

Afin de récupérer l'adresse du fils gauche ou celle du fils droit d'un nœud donné nous avons les deux primitives *gauche* et *droit*.

Le fils gauche : La fonction *noeud * gauche(noeud * R)* retourne l'adresse du fils gauche du nœud d'adresse *R*.

```
noeud * gauche(noeud * R)
{
  if (est_vide(R)) return NULL;
  else return R->gauche;
}
```

Évidemment, il faut vérifier si $R \neq NULL$ [2].

Le fils droit : La fonction *noeud * droit(noeud * R)* retourne l'adresse du fils droit du nœud d'adresse R.

```
noeud * droit(noeud * R)
{
  if (est_vide(R)) return NULL;
  else return R->droit;
}
```

3. Vérifier si un nœud est une feuille

La fonction *bool est_feuille(noeud * R)* vérifie si le nœud d'adresse R est une feuille. Autrement dit, elle retourne la valeur *true* si le nœud n'a pas de fils et $false$ sinon.

```
bool est_feuille(noeud * R)
{
  if (est_vide(R)) return false;
  else
  {
    if (est_vide(R->gauche) && est_vide(R->droit)) return true;
    else return false;
  }
}
```

4. Vérifier si un nœud est interne (pas une feuille)

La fonction *bool est_interne(noeud * R)* vérifie si le nœud d'adresse R est un nœud interne. Autrement dit, elle retourne la valeur *vrai* si le nœud a des enfants et $faux$ sinon.

```
bool est_interne(noeud * R)
{
  return (! est_feuille(R));
}
```

2. Dans le cas $R = NULL$, l'accès à un des fils gauche ou droit peut être l'origine d'un bug

5. Calcul de la hauteur d'un arbre

Le calcul de la hauteur d'un arbre revient à calculer le nombre de niveaux contenus dans l'arbre.

Pour mieux expliquer la notion de « hauteur d'un arbre », nous introduisons l'exemple 5.3.3 ci-dessous :

Exemple 5.3.3 (hauteur d'un arbre) *Soit la figure 56 ci-dessous qui donne la structuration en niveaux de l'arbre binaire d'entier introduit dans l'exemple 5.3.3.*

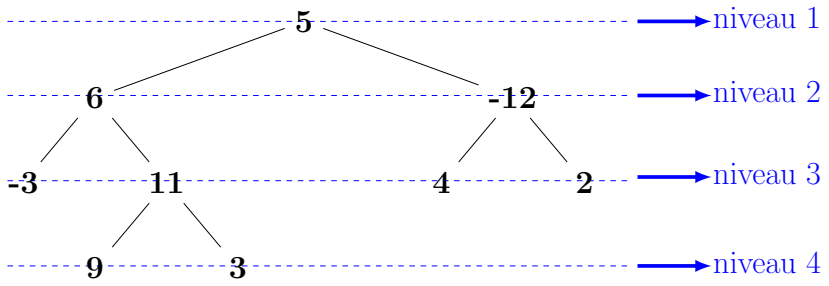


FIGURE 56 – Niveaux de l'arbre binaire de la figure 54

Les niveaux sont numérotés de 1 *à* 4 *en commençant par celui de la racine. Donc, la hauteur de l'arbre est* 4.

La fonction récursive *unsigned int hauteur(noeud * racine)* retourne la hauteur d'un arbre binaire en démarrant de la racine[3].

```
unsigned int hauteur(noeud * racine)
{
  if (est_vide(racine)) return 0;
  else
  return(1+max(hauteur(gauche(racine)), hauteur(droit(racine))));
}
```

6. Calcul du nombre de nœuds d'un arbre

La fonction *unsigned int nbre_noeud(noeud * racine)* calcule le nombre de nœuds contenus dans un arbre binaire.

```
unsigned int nbre_noeud(noeud * racine)
{
  if (est_vide(racine)) return 0;
  else
  return(1+nbre_noeud(gauche(racine)),nbre_noeud(droit(racine))));
}
```

3. La fonction max est une fonction prédéfinie dans la bibliothèque *math.h*, elle calcule le maximum de deux paramètres

7. Calcul du nombre de feuilles d'un arbre

Calculer le nombre de feuilles revient à calculer le nombre de nœuds sans enfants. Nous avons ci-dessous la fonction *unsigned int nbre_feuille(noeud * racine)* qui retourne le nombre de feuilles contenues dans d'un arbre binaire dont l'adresse de la racine est mis en paramètre d'entrée *racine*.

```
unsigned int nbre_feuille(noeud * racine)
{
  if (est_vide(racine)) return 0;
  else
  {
    if (est_feuille(racine)) return 1;
    else
    return
    nbre_feuille(gauche(racine))+nbre_feuille(droit(racine));
  }
}
```

8. Différents parcours d'un arbre

Parcourir un arbre, c'est accéder aux nœuds de l'arbre en suivant un des types de parcours ci-dessous :

1. parcours en profondeur préfixe,
2. parcours en profondeur infixe,
3. parcours en profondeur suffixe,
4. parcours en largeur.

Parcours en profondeur préfixe : Dans ce cas, les nœuds sont parcourus dans l'ordre : $racine \rightarrow fils\_gauche \rightarrow fils\_droit$. Ce qui donne en C++ la fonction ci-dessous *parcours_profondeur_prefixe*.

```
void parcours_profondeur_prefixe(noeud * racine)
{
  if (! est_vide(racine))
  {
    traiter(racine);//traiter le noeud racine
    parcours_profondeur_prefixe(gauche(racine));
    parcours_profondeur_prefixe(droit(racine));
  }
}
```

En prenant comme exemple l'arbre binaire illustré dans la figure 54 et en utilisant le parcours en profondeur préfixe, l'accès au nœuds suit l'ordre suivant : $5, 6, -3, 11, 9, 3, -12, 4, 2$.

Parcours en profondeur infixe : Dans ce cas, les nœuds sont parcourus dans l'ordre : $fils\_gauche \rightarrow racine \rightarrow fils\_droit$.

Nous introduisons ci-dessous la fonction *parcours_profondeur_infixe* qui exprime le parcours *infixe* en C++ :

```
void parcours_profondeur_infixe(noeud*racine)
{
  if (!est_vide(racine))
  {
    parcours_profondeur_infixe(gauche(racine));
    traiter(racine);//traiter le noeud racine
    parcours_profondeur_infixe(droit(racine));
  }
}
```

L'application de ce type de parcours sur l'arbre de la figure 54, les nœuds seront parcourus comme suit : $-3, 6, 9, 11, 3, 5, 4, -12, 2$.

Parcours en profondeur suffixe : Dans ce cas, la racine est parcourue en dernier, l'ordre des nœuds se donne comme suit : $fils\_gauche \rightarrow fils\_droit \rightarrow racine$.

```
void parcours_profondeur_sufffixe(noeud*racine)
{
  if (! est_vide(racine))
  {
    parcours_profondeur_sufffixe(gauche(racine));
    parcours_profondeur_sufffixe(droit(racine));
    traiter(racine);//traiter le noeud racine
  }
}
```

En suivant ce parcours, l'ordre d'accès aux nœuds de l'arbre de la figure 54 se donne comme suit : $-3, 9, 3, 11, 6, 4, 2, -12, 5$.

Parcours en largeur : Il est différent du parcours en profondeur, dans ce cas le parcours de l'arbre se fait par niveau, tel exprimé par la fonction ci-dessous *parcours_largeur*.

```
void parcours_largeur(noeud * racine)
{
  noeud * temp;
  file F;//F est une file pour stocker les noeuds en attente
  if (!est_vide(racine))
  {
    enfiler(F,racine);
    while(! est_vide(F))
    {
      temp = defiler(F);
      // traiter le noeud temp
     if (!est_vide(gauche(Temp))) enfiler(F,Temp);
     if (!est_vide(droit(Temp))) enfiler(F,Temp);
    }
  }
}
```

Dans la fonction nous utilisons une structure intermédiaire *file*. Le principe est d'enfiler les valeurs des nœuds par niveau. Par conséquent le fait de défiler les éléments de la file implique la récupération des valeurs de gauche à droite par niveau.

L'application de la fonction sur l'arbre de la figure 56 donne en résultat l'ordre des valeurs comme suit : $5, 6, -12, -3, 11, 4, 2, 9, 3$.

9. Créer un nœud d'un arbre binaire

Puisque nous utilisons une structure dynamique, la création d'un nœud nécessite l'allocation de l'espace mémoire suffisant pour charger ce nœud.

La fonction *noeud * creer_noeud(TypeElt val)* permet la création d'un nœud d'un arbre binaire, dont la valeur est donnée en entrée *val* et l'adresse est le résultat retourné.

```
noeud * creer_noeud(TypeElt val)
{
  noeud * r = new noeud;
  if (r == NULL) cout << "mémoire insuffisante!" << endl;
  else
  {
    r->elt = val;
    r->gauche = NULL;
    r->droit = NULL;
  }
    return r;
}
```

10. Créer la racine d'un arbre binaire

La fonction *noeud * creer_racine(TypeElt val, noeud *g, noeud *d)* permet la création de la racine d'un arbre binaire, avec l'existence au préalable des deux fils d'adresses respectives g et d.

```
noeud * creer_racine(TypeElt val, noeud * g, noeud * d)
{
  noeud * r = new noeud;
  if (r == NULL) cout << "mémoire insuffisante!" << endl;
  else
  {
    r->elt =val;
    r->gauche = g;
    r->droit = d;
  }
  return r;
}
```

11. Exemple d'insertion d'un nœud

Soient les valeurs à insérer dans l'ordre, dans un arbre binaire vide *racine* = *NULL* : 5, 6, -12, -3, 11, 4 et 2.

Nous introduisons ci-dessous la déclaration de l'arbre.

```
struct noeud
{
        int elt;
        noeud * gauche;
        noeud * droit;
}
noeud * racine=NULL;
```

La fonction ci-dessous *ajout_elt* permet l'insertion de ces valeurs de telle sorte à ce qu'elle nous donne en résultat l'arbre binaire illustré dans la figure 57.

```
void ajout_elt(noeud * r, int val)
{
  if (r != NULL)
  {
    if (est_vide(gauche(r))) r->gauche = creer_noeud(val);
    else
      if (est_vide(droit(r))) r->droit = creer_noeud(val);
      else ajout_elt(gauche(r), val);
   }
   else r= creer_racine(val, NULL,NULL);
}
```

L'insertion de 7 éléments d'un tableau d'entiers se fait via l'appel de *ajout_elt* se donne comme suit :

```
...
int tab[7]={5,6,-12,-3,11,4,2};
for(int i=0;i<=6;i++)
        ajouter_elt(racine,tab[i]);
...
```

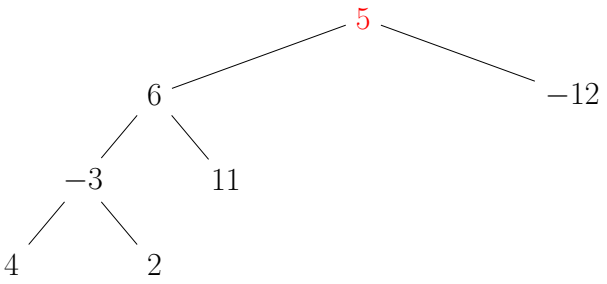


FIGURE 57 – Arbre binaire correspondant à l'application de la fonction *ajout_elt* après insertion des valeurs $5, 6, -12, -3, 11, 4, 2$

Arbre planaires généraux

Soit la définition 5.3.4 suivante :

Définition 5.3.4 (arbre planaire général) *[Froidevaux et al., 1993] Un arbre* $A = \langle \circ, A_1, \cdots A_p \rangle$ *est la donnée d'un racine et d'une liste finie, éventuellement vide (si* $p = 0$*), d'abord disjoints.*

Dans une structure arborescente appelée arbre planaire général ou tout brièvement arbre général ou arbre, le nombre de fils de chaque nœud n'est plus limité à 2. On peut représenter les arbres en trois façons[Froidevaux et al., 1993].

1. Représentation dynamique

On peut donner à chaque nœud la liste de ses fils. Cette représentation des arbres peut être utile, mais elle se prête mal à une gestion dynamique.

```
struct list_noeud
{
        int noeud; //valeur du noeud
        list_noeud * suivant; //pointeur vers le noeud frère
}
liste_noeud * vect_noeud[nbre_noued]; //nbre_noeud est le
                                      //nombre de noeuds
```

2. Représentation analogue à un arbre binaire

On peut aussi décrire une représentation analogue à celle des arbres binaires. Chaque nœud contient un pointeur vers chacun des sous-arbres et éventuellement un champ pour stocker l'élément contenu dans le nœud.

```
struct arbre_noeud
{
        int val_noeud; //valeur du noeud
        list_noeud * frere1; //pointeur vers premier frère
        list_noeud * frere2; //pointeur vers le 2ème frère
           ...
        list_noeud * freren; //pointeur vers le n-ème frère
}
arbre_noeud * racine;
```

Pour utiliser cette structure il faut connaître le nombre maximum de fils que peut avoir un nœud de l'arbre.

3. Représentation par un arbre binaire

On peut transformer l'arbre général en un arbre binaire et utiliser la structure en C++ décrite précédemment dans la section 5.3.1. La transformation se fait au niveau de chaque nœud de la manière suivante :

Considérons un nœud x qui a n fils notés f_1, f_2, ...f_n, dans un arbre général. Pour constituer un arbre binaire :

- f_1 devient le fils gauche de x,
- f_2 devient le fils droit de f_1,
- f_3 devient le fils droit de f_2,
- ...
- f_n devient le fils droit de f_{n-1}.

Exemple 5.3.5 (passage d'un arbre général à un arbre binaire) *Considérons l'arbre général d'entiers donné dans la figure ci-dessous 58.*

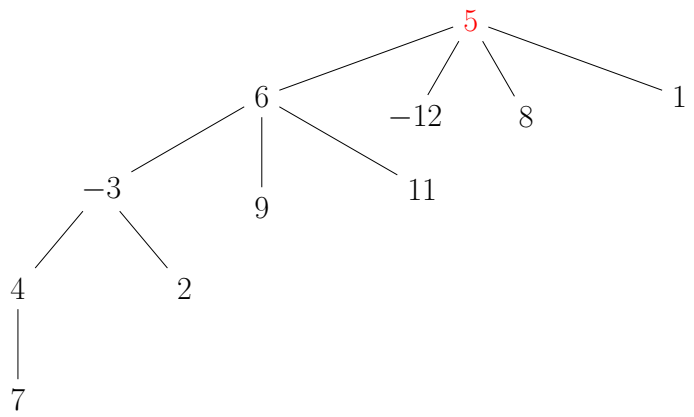


FIGURE 58 – Arbre général d'entiers

En suivant le principe de passage vers un arbre binaire, cité précédemment, nous obtenons l'arbre binaire illustré dans la figure ci-dessous 59.

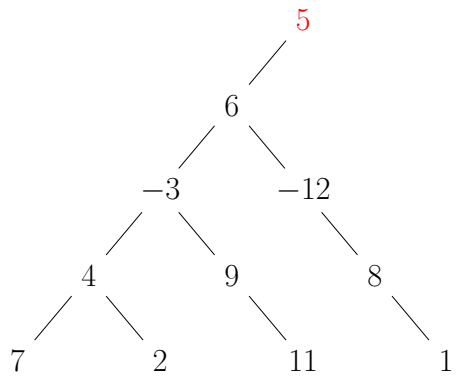


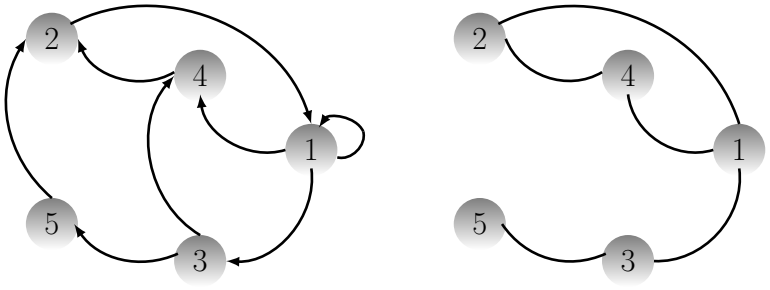
FIGURE 59 – Arbre binaire correspondant à l'arbre général 58

5.3.2 *Graphes*

Un graphe est un concept utilisé pour la modélisation de nombreux problèmes issus de la physique, l'automatique, la mécanique, l'électronique, etc.

Nous avons deux catégories de graphes : (1) graphes orientés et graphes non orientés.

Dans la figure 60, nous introduisons deux graphes orienté et non-orienté exprimés respectivement par les deux sous-figures 60a et 60b.



(a) Graphe orienté (b) Graphe non-orienté

FIGURE 60 – Exemple de graphes orienté et non-orienté

Définition 5.3.6 (graphe orienté) *[Froidevaux et al., 1993] Un graphe orienté G est un couple $\langle S, A \rangle$, où S est un ensemble fini de sommets et où A est un ensemble fini de paires ordonnées de sommets, appelées arcs.*

Définition 5.3.7 (graphe non orienté) *[Froidevaux et al., 1993] Un graphe non orienté G est un couple $\langle S, A \rangle$, où S est un ensemble fini de sommets et où A est un ensemble fini de paires de sommets, appelées arêtes.*

Terminologie[Froidevaux et al., 1993]

— Soit $G = \langle S, A \rangle$ un graphe, le sous-graphe de G engendré par $S' \subseteq S$ est le graphe G' dont les sommets sont les éléments de S' et dont les arcs (resp. arêtes) sont les arcs (resp. les arêtes) ayant leurs deux extrémités dans S'.
— Soit $G = \langle S, A \rangle$ un graphe, le graphe partiel de G engendré par $A' \subseteq A$ est le graphe G' dont les sommets sont les éléments de S et dont les arcs (resp. arêtes) sont les éléments de A'.
— deux arcs (resp. arêtes) d'un graphe orienté (resp. non orienté) sont dits adjacents s'ils ont au moins une extrémité en commun.
— Un graphe orienté (resp. non orienté) est dit complet , si pour tout couple de sommets (x, y), il existe un arc $x \to y$ (resp. une arête x____y)

1. Représentation des graphes en utilisant la structure d'un tableau

En utilisant la matrice d'incidence :[Gondran et Minoux, 1995] Dans ce cas on doit connaître au préalable le nombre de sommets constituant le graphe :

1. Pour un graphe non pondéré (les arcs ne portent pas de valeurs),on utilise la déclaration suivante :

```
typedef bool graphe[n][n];//n est le nombre de sommets
```

2. Pour un graphe pondéré (chaque arc porte un poids), on utilise la déclaration suivante :

```
typedef int graphe[n][n];//n est le nombre de sommets
```

ou

En utilisant l'allocation dynamique :

1. Pour un graphe non pondéré, la création du graphe se fait comme suit :

```
int n;
cin >>n;
bool ** graphe =NULL;
graphe = new bool * [n];
for(int i=0;i<n;i++)
{
  graphe[i] = new bool[n];
}
```

2. Pour un graphe pondéré, la création du graphe se fait comme suit :

```
int n;
cin >>n;
int ** graphe =NULL;
graphe = new int * [n];
for(int i=0;i<n;i++)
{
  graphe[i] = new int[n];
}
```

L'évaluation des éléments du graphe : Les éléments de la matrice "graphe" sont évalués comme suit :

1. Si le graphe est non valué :
   — graphe[i][j] = 1 si l'arc $i \to j \in A$
   — graphe[i][j] = 0 sinon
2. Si le graphe est valué :
   — graphe[i][j] = val($i \to j$) si l'arc $i \to j \in A$, tel que val($i \to j$) est la valeur de l'arc,
   — graphe[i][j] = -1 sinon

Exemple 5.3.8 (représentation des graphes-matrice d'adjacence) *Soient les graphes illustrés dans la figure 60.*

Les matrices d'adjacence M et M′ correspondent respectivement au graphe orienté de la sous-figure 60a et au graphe non-orienté de la sous-figure 60a.

$$M = \begin{bmatrix} 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix} \quad M' = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Soit le graphe de la figure 61 ci-dessous :

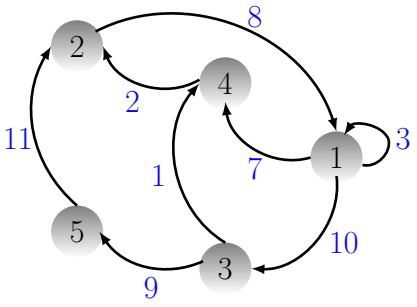


FIGURE 61 – Graphe orienté valué

La matrice d'adjacence M″ correspondant se donne comme suit :

$$M'' = \begin{bmatrix} 3 & 0 & 10 & 7 & 0 \\ 8 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 9 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 11 & 0 & 0 & 0 \end{bmatrix}$$

2. Représentation des graphes en utilisant les listes d'adjacence

Une autre représentation classique des graphes consiste à représenter l'ensemble des sommets et à associer à chaque sommet de la liste de ses successeurs rangés dans un certain ordre. Ces listes sont appelées listes d'adjacence [Froidevaux et al., 1993].

Nous introduisons ci-dessous la déclaration de la liste d'adjacence en C++.

```
struct sommet
{
  int val;
  sommet * suivant;
}
Sommet * graphe[n];//n : le nombre de sommets
```

Exemple 5.3.9 (représentation des graphes-liste d'adjacence) *La liste d'adjacence qui correspond au graphe orienté de la sous-figure 60a se donne via la figure 61 ci-dessous :*

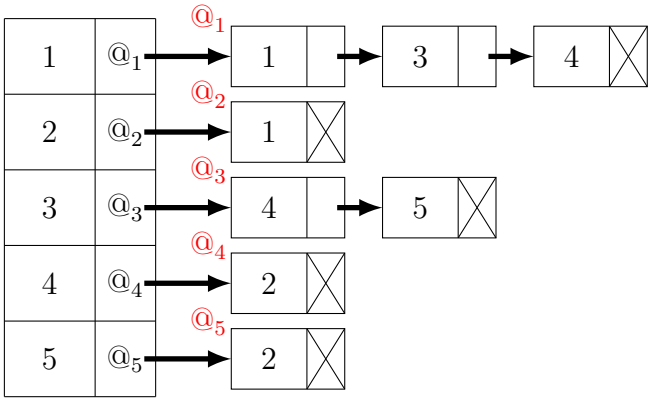


FIGURE 62 – Liste d'adjacence correspondant au graphe de la sous-figure 61

5.4 FICHIERS

5.4.1 *Généralités sur les flots*

Les entrées et les sorties sont désignées par les flots d'entrées et ceux de soties qu'on peut considérer comme des canaux qui transmettent de l'information. Tel illustré dans la figure 63, ces deux types de flots s'expriment comme suit :

— **Les flots de sortie :** transmettent l'information des composants *mémoire* + *processeur* vers une sortie.

— **Les flots d'entrée :** fournissent l'information pour les composants *mémoire* + *processeur*.

Flux d'entrée

Le flux d'entrée standard est *cin* (voir la figure 63). Ce flux est connecté à une entrée (clavier, souris, etc.). On utilise l'opérateur ">>" qui permet la lecture sur le flux d'entrée.

Flux de sortie

Le flux de sortie standard est *cout* (voir la figure 63). Ce flux est connectée à une sortie (écran, imprimante, etc.). On utilise l'opérateur "<<", qui permet l'écriture sur un flux de sortie.

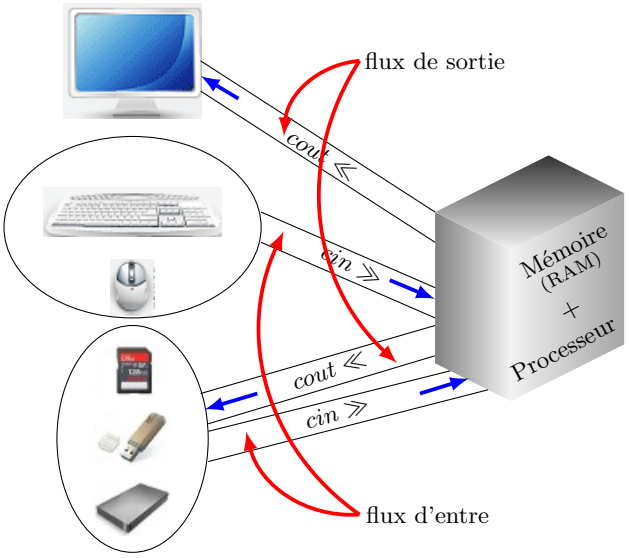


FIGURE 63 – Flux d'E/S

5.4.2 *Ouverture et fermeture d'un fichier*

Exploiter les fichiers en C++ implique l'inclusion du fichier bibliothèque $\langle fstream \rangle$.

Syntaxiquement parlant, on ne manipule jamais le nom du fichier physique (ou réel), on passe toujours par un fichier logique, que ce soit pour l'ouverture ou l'écriture ou l'accès aux données du fichier.

Ouverture d'un fichier en lecture

La syntaxe en C++ se donne comme suit :

```
ifstream nom_fichier_logique("chemin_fichier_physique");
```

On fait appel au mot clé *ifstream* qui est composé de trois parties *i*, *f* et *stream*, avec :

i : première lettre du mot en anglais *in* (*entrée* en français. Ce qui indique qu'il s'agit d'une ouverture en mode lecture,

f : première lettre de *file* (*fichier* en français) ,

stream : flot en français.

ifstream est suivi de :

nom_fichier_logique : nom du fichier logique,

chemin_fichier_physique : nom du fichier physique avec le chemin.

Exemple 5.4.1 (ouverture en lecture) *Soit le fichier* ***tp.txt*** *qui se situe dans le répertoire* ***info****, qui est à son tour dans le répertoire source* ***c :***\.

L'ouverture du fichier en lecture se donne comme suit :

```
ifstream test("c:\info\tp.txt")
```

Avec **test** le nom du fichier logique.

Ouverture d'un fichier en écriture

La syntaxe d'ouverture en mode écriture se donne comme suit :

```
ofstream nom_fichier_logique("chemin_fichier_physique");
```

On fait appel au mot clé *ofstream* qui commence avec la lettre *o* à la place de *i*. Elle désigne première lettre du mot en anglais *out* (*sortie* en français. Ce qui indique qu'il s'agit d'une ouverture en mode écriture.

Exemple 5.4.2 (ouverture en écriture) *Soit le fichier* ***tp.txt*** *de l'exemple précédent 5.4.1.*

L'ouverture du fichier en mode écriture se donne comme suit :

```
ofstream test("c:\info\tp.txt")
```

Fermeture d'un fichier

La fermeture d'un fichier après toute ouverture est obligatoire. La syntaxe en C++ se donne comme suit :

```
nom_fichier_logique.close();
```

Exemple 5.4.3 (fermeture d'un fichier) *L'instruction qui permet la fermeture du fichier* ***tp.txt*** *de l'exemple 5.4.1 est la suivante :*

```
test.close();
```

5.4.3 *Manipulations sur les fichiers*

Test de fin

Lors d'un accès au contenu d'un fichier, le test de fin du fichier s'avère nécessaire. Ce test s'exprime en C++ comme suit :

```
nom_fichier_logique.eof();
```

Nous utilisons le mot clé *eof* composé trois caractères :

e : *end* en anglais (fin en français),

o : *of* en anglais (de en français),

f : *file* en anglais (fin en français),

La fonction retourne un résultat booléen, *true* si la fin du fichier est atteinte et *false* sinon.

Écriture dans un fichier

L'écriture dans un fichier peut se faire de différentes manières :

Si on utilise l'opérateur <<, on a la syntaxe en C++ :

```
nom_fichier_logique<<expression1[<<expression2<<...];
```

Exemple 5.4.4 (écriture dans un fichier) *Soit le petit texte suivant :*

L'algorithmique avancé
Différentes structures

L'écriture du paragraphe dans le fichier tp.txt (voir l'exemple 5.4.2), se donne comme suit :

```
test << "L'algorithmique"<< " " << "avancée"<<
        "\n" << "Différentes" << " " <<"structures";
```

Si on utilise la fonction *get*, on a la syntaxe en C++ :

```
nom_fichier_logique.get(caractère);
```

Avec *get*, l'écriture se fait par caractère.

Si on utilise la fonction *getline*, on a la syntaxe en C++ :

```
getline(nom_fichier_logique, chaineDeCaractères);
```

La fonction *getline* permet une écriture par ligne.

Lecture à partir d'un fichier

Pour la lecture d'un fichier, nous utilisons l'opérateur <<. La syntaxe de lecture est comme suit :

```
nom_fichier_logique << avriable1[<<variable2<<...];
```

Sachant que $variable_i$ est de type chaîne de caractère distingués par des séparateurs de lecture, tels que l'espace et le saut de ligne.

Exemple 5.4.5 (lecture d'un fichier) *Soient le fichier tp.txt (voir l'exemple 5.4.4) et les instructions* C++ *ci-dessous :*

```
string st1,st2,st3,st4;
test >> st1 >> st2 >> st3 >> st4;
```

L'exécution de ces instructions modifie le contenu des variables : ***st1****="L'algorithmique",* ***st2****="avancée",* ***st3****="Différentes" et* ***st4****="structures".*

5.4.4 *Exemple illustratif*

— Programme d'écriture sur le fichier *personne.txt* :

```
#include <iostream>
using namespace std;
#include <fstream>
main ( )
{
  string nom, tel;
  ofstream fichrep ("personne.txt");
  for(int i=0; i<10; i++)
  {
    cout << "\n personne "<< i+1 << ":\n"
    cout << "nom? ";
    cin >> nom;
    fichrep << nom << " ";
    cout << "\ntelephone?";
    cin >> tel;
    fichrep << tel << "\n";
  }
  fichrep.close();
}
```

— Programme de lecture sur le fichier *personne.txt* :

```
#include <conio.h>
#include <stdio.h>
#include <iostream>
using namespace std;
#include <string.h>
main()
{
  string nom, tel;
  ifstream fichrep ("personne.txt");
  fichrep >> nom >> tel;
  while (!fichrep.eof())
  {
    cout << nom << " \t"<< tel << "\n";
    fichrep >> nom >> tel;
  }
  fichrep.close();
}
```

BIBLIOGRAPHIE

[Bensaoud-Senhadji, 2005] Bensaoud-Senhadji, T. (2005). Concepts et outils de base de la programmation. Science et Technologie. ANEP.

[Cormen et al., 2001] Cormen, T. H., Leiserson, C. E., Rivest, R. L. et Stein, C. (2001). Introduction to Algorithms. MIT Press, Cambridge, MA, USA, 2nd édition.

[Crochemore et Rytter, 1994] Crochemore, M. et Rytter, W. (1994). Text Algorithms. Oxford University Press, Inc., New York, NY, USA.

[Danièle Beauquier, 2003] Danièle Beauquier, Jean Berstel, P. C. (2003). Éléments d'algorithmique. Masson.

[Froidevaux et al., 1993] Froidevaux, C., Gaudel, M. et Soria, M. (1993). Types de données et algorithmes. Collection Informatique. Ediscience international.

[Gondran et Minoux, 1995] Gondran, M. et Minoux, M. (1995). Graphes et algorithmes. Collection De la Direction des études et recherches d'Électricité de France. Eyrolles.

[GREFFIER, 2007] GREFFIER, F. (2007). Algorithmique sur les données. `circe.univ-fcomte.fr/Francoise-Greffier/pwpt/8_RECURSIVITE.pps`. Licence Informatique.

[Prins, 1994] Prins, C. (1994). Algorithmes de graphes : avec programmes en Pascal. Eyrolles.

[Sam-Haroud, 2012] Sam-Haroud, D. (2012). Informatique ii : Cours de programmation (c++) pointeurs, références & fonctions. `http://icwww.epfl.ch/~sam/infosv/slides/cours02.pdf`. Laboratoire d'Intelligence Artificielle Faculté I&C.

[Sedgewick, 1991] Sedgewick, R. (1991). Algorithmes en langage C. I.I.A. Informatique intelligence artificielle. Dunod.

[Shaffer, 2012] Shaffer, C. (2012). Data Structures and Algorithm Analysis in Java, Third Edition. Dover Books on Computer Science. Dover Publications.