

Chapitre 3

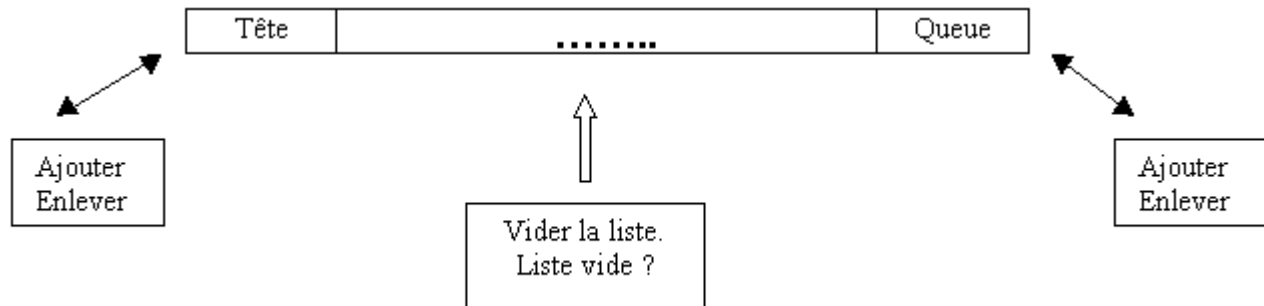
Les structures de base : listes, piles et files

1. Introduction

Le but de ce chapitre est de décrire des représentations des structures de base utilisées en informatique telles les listes en général et deux formes restreintes: les piles et les files. L'autre but recherché est premièrement de donner des exemples de séparations des représentations logiques à travers les TDA des implémentations physiques, et, deuxièmement, d'illustrer l'utilisation de la détermination de la complexité d'un algorithme.

2. Les listes

Les listes sont des **structures informatiques** qui permettent, au même titre que les tableaux par exemple, de garder en mémoire des données en respectant un certain ordre: on peut ajouter, enlever ou consulter un élément en début ou en fin de liste, vider une liste ou savoir si elle contient un ou plusieurs éléments.



La structure de liste particulièrement souple et dont les éléments sont accessibles à tout moment. Toute suite d'informations inscrites les unes après les autres est une liste. Exemple : liste d'étudiants, liste d'inventaire, liste des absents, ...

Une **liste** est définie comme étant une suite ordonnée d'éléments. L'ordre signifie que chaque élément possède une position dans la liste. L'implémentation doit supporter le concept de *position courante*. Cela est fait en définissant la liste en termes de **partition** de *droite* et de *gauche*.

- Chacune de ces partitions peut être vide.
- Les partitions sont séparées par un **séparateur** (courant ou fence).
- L'élément courant est le premier élément de la partition de droite

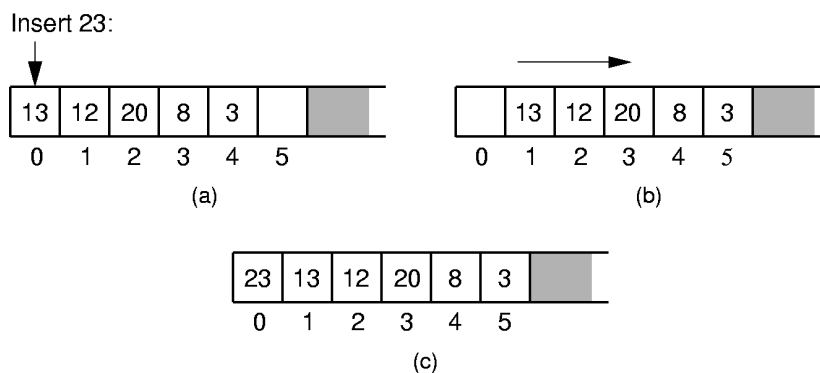
<20, 23 | 12, 15>

2.1 Les opérations sur les listes : Les opérations sur les listes sont très nombreuses, parmi elles, on peut citer les plus fréquentes:

- Créer une liste vide
- Tester si une liste est vide
- Ajouter un élément à la liste
- ajouter en début de liste (tête de liste)
- ajouter à la fin de la liste (fin)
- ajouter un élément à une position donnée
- ajouter un élément après une position donnée
- ajouter un élément avant une position donnée
- Afficher ou imprimer les éléments d'une liste
- Ajouter un élément dans une liste triée (par ordre ascendant ou descendant)
- Supprimer un élément d'une liste
- supprimer en début de liste
- supprimer en fin de liste
- supprimer un élément à une position donnée
- supprimer un élément avant ou après une position donnée.

2.2 Implantation par tableau

La liste peut être implantée à l'aide d'un tableau. Dans ce cas, il faut prévoir une taille maximum pour le tableau. L'opération d'impression des éléments d'une liste et de recherche d'un élément se font en temps linéaire. La recherche du k -ème élément se fait $O(1)$ i.e. en un temps constant. Cependant les insertions et les suppressions d'éléments sont plus coûteuses. Par exemple, l'insertion à la position 0 nécessite le déplacement de tous les éléments de la liste pour libérer la position 0.



Par conséquent la création d'une liste à l'aide de n insertions successives à partir de la position 0 nécessite un temps en $O(n^2)$ (pourquoi?). Ce temps est dit quadratique

```

// fichier LISTE.H
// Déclaration de la classe Liste

#ifndef LISTE_H
#define LISTE_H

#define Taille_de_Liste 100

template <classe T> ;

class Liste {
public :
    Liste ( const int = Taille_de_Liste ) ;    // Constructeur
    ~Liste () ;                                // Destructeur
    void ViderListe () ;                        // Vide la liste
    void Insérer ( const TElement & ) ;        // insère un élément à la position courante
    void InsérerFin ( const TElement & ) ;    // insère un élément à la fin de la liste
    TElement Supprimer () ; // Supprime et retourne l'élément à la position courante
    void FixerTete () ; // met la position courante à la tête de la liste
    void Précédent () ; // Déplace la position courante à la position précédente
    void Suivant () ; // Déplace la position courante à la position suivante
    int Longueur () const ; // retourne la longueur courante de la liste
    void FixerPosition ( const int ) ; // met position courante à position donnée
    void FixerValeur ( const TElement & ) ; // met à jour la valeur à la position courante
    TElement ValeurCourante () const ; // retourne la valeur d'élément à la position courante.
    bool ListeVide () const ; // retourne vrai si la liste est vide
    bool EstDansListe () const ; // retourne vrai si position courante est dans la liste
    bool Trouver ( const TElement & ) ; // recherche une valeur dans la
                                     // liste à partir de la position courante.

private :
    T TailleMax ; // taille maximum de la liste
    T NbElement ; // nombre d'éléments effectifs dans la liste
    T Courant ; // position de l'élément courant
    T * TabElement ; // tableau contenant les éléments de la liste
} ;
#endif

```

Note : remarquez que le mot clé **const** est utilisé à la fin des déclarations de fonction membres en mode lecture seule, à savoir EstDansListe (), ValeurCourante () et Longueur (). Afin de mieux protéger un programme, il y a tout intérêt à déclarer comme constantes les fonctions qui ne sont pas censées modifier l'état des objets auxquels elles sont liées. On bénéficiera ainsi d'une aide supplémentaire du compilateur qui empêchera tout ajout de code modifiant l'état de l'objet dans la fonction.

```

// Fichier LISTE.CPP

#include <cassert>
#include "LISTE.H"

Liste::Liste ( const int taille )          // Constructeur
{
    TailleMax = taille ;
    NbElement = Courant = 0 ;
    TabElement = new TElement [taille] ;
}

Liste::~~Liste ()                          // Destructeur
{
    delete [] TabElement ;
}

void Liste::ViderListe ()
{
    NbElement = Courant = 0 ; // noter que le tableau n'est pas détruit
}

// Insère un élément à la position courante

void Liste::Inserer (const TElement & element )
{
    // le tableau ne doit pas être plein et Courant doit être à une position légale
    assert (( NbElement < TailleMax) && (Courant >= 0) && (Courant <= NbElement)) ;
    for ( int i = NbElement; i > Courant; i-- )
        TabElement[i] = TabElement[i-1] ; // Décale les éléments vers le haut
    TabElement[Courant] = element ;
    NbElement++ ;
}

void Liste::InsererFin ( const TElement & element )
{
    assert ( NbElement < TailleMax ) ; // la liste ne doit pas être pleine
    TabElement[NbElement++] = element ;
}

TElement Liste::Supprimer ()
{
    assert ( !ListeVide() && EstDansListe() ) ; // s'assurer qu'il existe
                                                // un élément à supprimer
    TElement temp = TabElement[Courant] ;
    for ( int i = Courant; i < NbElement-1; i++ ) // décale les éléments
                                                // vers le bas

```

```

        TabElement[i] = TabElement[i+1] ;
        NbElement-- ;
        return temp ;
    }

void Liste::FixerTete () // rend la tête comme position courante
{
    Courant = 0 ;
}

void Liste::Precedent () // met la position courante à la position précédente
{
    Courant-- ;
}
void Liste::Suivant ()
{
    Courant++ ;
}

int Liste::Longueur () const
{
    return NbElement ;
}

void Liste::FixerPosition (const int pos )
{
    Courant = pos ;
}

void Liste::FixerValeur ( const TElement & valeur )
{
    assert ( EstDansListe() ) ; // Courant doit être à une position légale
    TabElement[Courant] = valeur ;
}

TElement Liste::ValeurCourante () const
{
    assert ( EstDansListe() ) ;
    return TabElement[Courant] ;
}

bool Liste::EstDansListe () const
{
    return ( Courant >= 0 ) && ( Courant < NbElement ) ;
}

```

```

bool Liste::ListeVide () const
{
    return NbElement == 0 ;
}

bool Liste::Trouver ( const TElement & valeur )    // recherche la valeur à partir
                                                    // de la position courante
{
    while ( EstDansListe() )
        if ( ValeurCourante() == valeur )
            return true ;
        else
            Suivant () ;
    return false ;
}

```

2.3 Implantation par liste chaînée

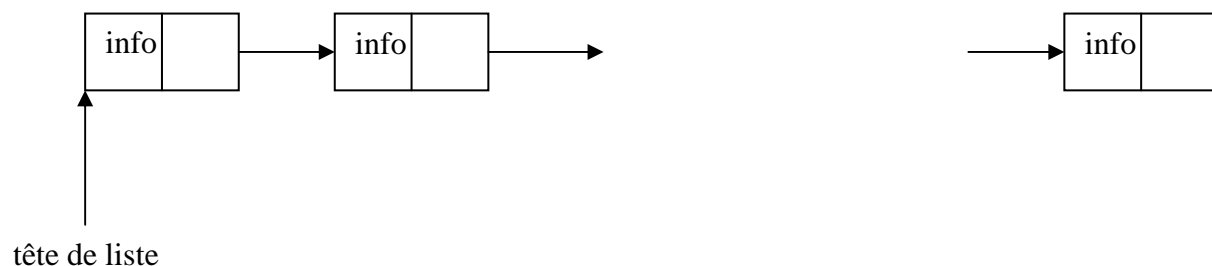
La deuxième approche traditionnelle pour implanter une liste est à travers l'utilisation des pointeurs. Cette approche fait en sorte que la mémoire est allouée d'une manière dynamique dans le sens que les cellules mémoire sont allouées au besoin. Les éléments d'une liste seront tout simplement chaînés entre eux. Il existe plusieurs types de listes chaînées dépendant de la manière dont on se déplace dans la liste :

1. les listes chaînées simples,
2. les listes doublement chaînées,
3. les listes circulaires.

2.3.1 Liste chaînée simple: C'est la liste de base dont chaque élément appelé nœud contient deux parties :

1. une partie contenant l'information proprement dite
2. et une partie appelée pointeur qui lie le nœud au nœud suivant.

Une liste chaînée simple est représentée graphiquement comme suit :



La liste étant une structure dynamique, son nombre de nœuds varie en cours de programme. Il faut donc supposer un certain nombre d'opérations primitives telles que:

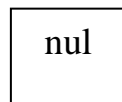
- créer (p) : qui permet d'obtenir une mémoire nœud pointée par p.
- libérer (p) : qui permet de libérer un nœud pointé par p

- nœud (p) : permet d'accéder au nœud pointé par p
- info (p) : permet d'accéder à la partie information d'un nœud pointé par p.
- suivant (p) est le pointeur (adresse) du prochain nœud

Pseudo-code de quelques opérations

a. Créer une liste vide

initialiser (Liste) /* Liste est un pointeur*/
 Liste \leftarrow nul



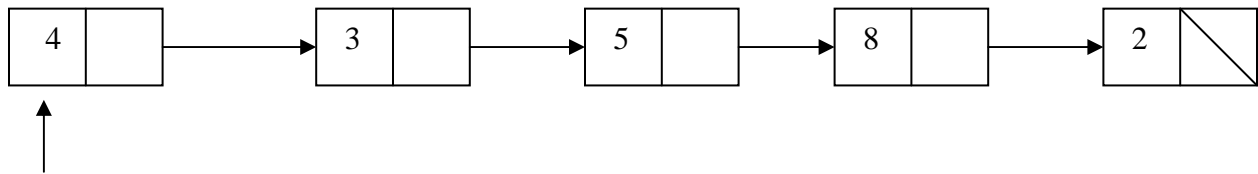
Liste

b. Créer un nœud

Creernœud () : le résultat est un pointeur vers le nœud créé

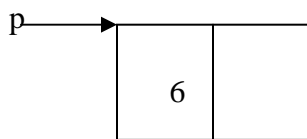
c. Ajout d'un élément en tête de liste

Exemple : On suppose que l'on veut rajouter l'élément 6 au début de la liste suivante :

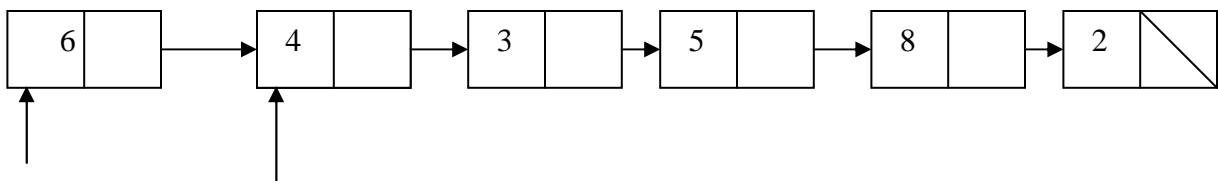


teteListe

a) on commence par créer le nouveau nœud et lui donner son contenu



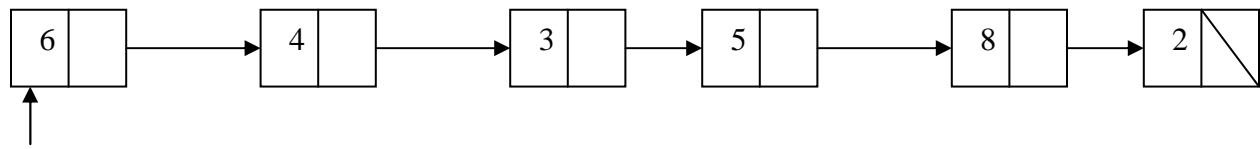
b) le suivant de p est l'ancienne tête de liste



p

teteListe

c) Le pointeur p devient tête de liste



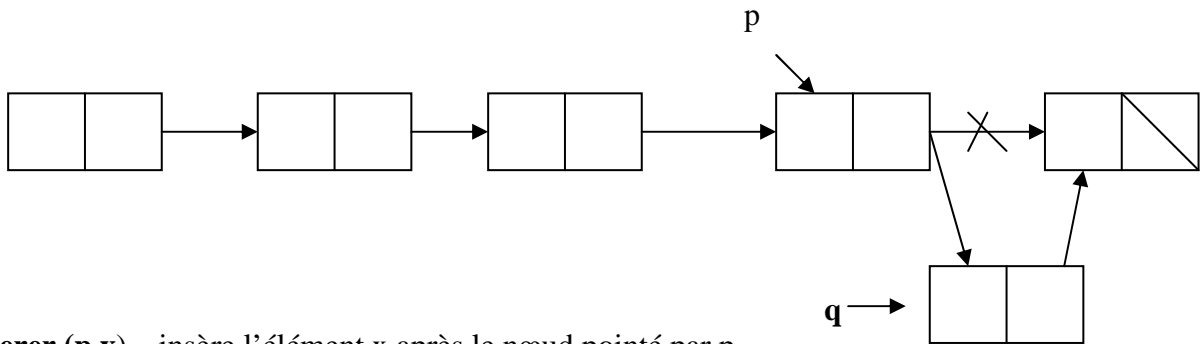
teteListe

Les opérations a, b et c se traduisent par le pseudo-code suivant :

```

p ← creernoeud ()
information (p) ← x      où x est l'information à stocker dans le nœud ajouté.
suivant (p) ← teteListe
teteListe ← p
  
```

Ajout d'un élément après un nœud pointé par p



```

insérer (p,x)  insère l'élément x après le nœud pointé par p.
q ← creernoeud ()
info (q) ← x
suivant (q) ← suivant (p)
suivant (p) = q;
  
```

Noter que le nombre d'opérations nécessaires pour réaliser cet algorithme est indépendant du nombre d'éléments de la liste.

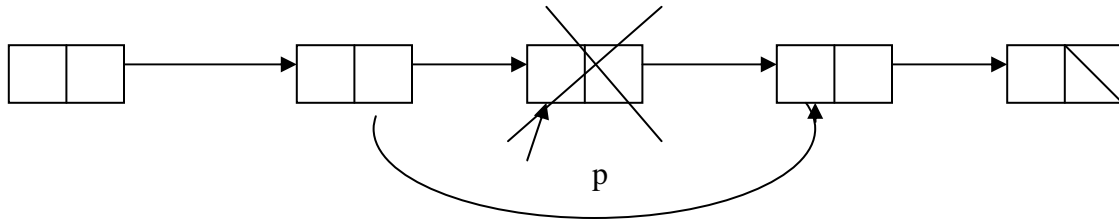
Ajout d'un élément en fin de liste

```

SI teteListe = nul ALORS
    ajout en tête de liste
SINON // chercher l'adresse du nœud de fin de liste
    q ← teteListe
    TANT QUE ( suivant (q) ≠ nul )
        q ← suivant (q)
    ajouter l'élément après l'élément pointé par q
  
```


Suppression d'un nœud

Pour supprimer un nœud, il n'est pas suffisant de donner un pointeur vers ce nœud. Il faut retrouver le prédécesseur de p pour établir le lien entre le prédécesseur de p et le successeur de p.



Suppression d'un élément après l'élément pointé par p

supprimer (p,x) : supprime l'élément x après p et renvoie la valeur de l'information du nœud supprimé dans x. Les opérations à effectuer sont comme suit :

```
q ← suivant (p)
x ← information (q)
suivant (p) ← suivant (q)
libérer (q)
```

2.3.2. Implantation de la liste chaînée simple en C++

Le nœud étant un objet distinct, il est intéressant de définir une classe pour les nœuds.

```
// fichier NOEUD.H
// Déclaration de la classe Noeud

#ifndef NOEUD_H
#define NOEUD_H

template < classe TElement >

class noeud
{
public :
    TElement element ;
    noeud * Suivant ;
    noeud ( const TElement & info, noeud * suiv = NULL ) // constructeur1
    {
        element = info ;
        Suivant = suiv ;
    }
}
```

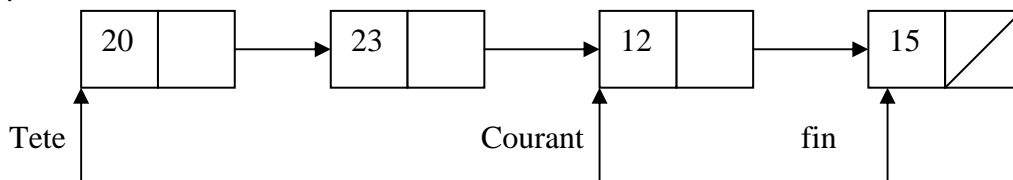
```

noeud ( noeud * suiv = NULL ) // constructeur 2
{
    Suivant = suiv ;
}
~noeud ()
{
}
};

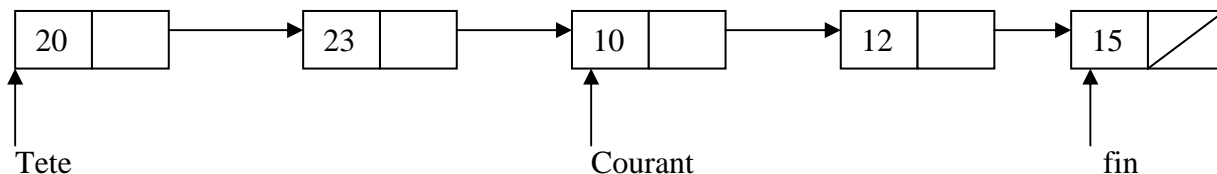
#endif

```

La classe `nœud` contient deux constructeurs, un qui prend une valeur initiale pour la partie information et l'autre non. Elle contient également un destructeur qui ne fait rien de spécial. Dans cette implémentation, nous utiliserons trois pointeurs : `Tete`, `fin` et `Courant` qui pointent respectivement vers le premier élément, dernier élément et l'élément courant de la liste. Par exemple :



Si on veut insérer une nouvelle valeur à la position courante, le nœud qui contient la valeur 10, on obtient la liste suivante :

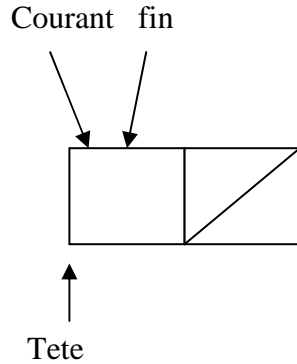


Le nœud 23 doit pointer vers le nouveau nœud. Une solution est de parcourir la liste depuis le début jusqu'à rencontrer le nœud qui précède le nœud pointé par `Courant`.

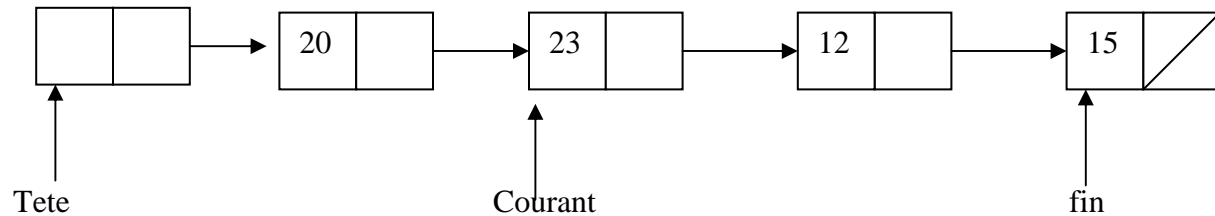
Une autre alternative est de faire pointer `Courant` vers l'élément qui précède l'élément courant. Dans notre exemple si 10 est l'élément courant, on ferait pointer `Courant` sur 23. Un problème surgit avec cette convention. Si la liste contient un seul élément, il n'y a pas de précédent pour `Courant`.

Les cas spéciaux peuvent être éliminés si on utilise un nœud *entête*, comme le premier élément de la liste. Ce nœud est un nœud **bidon**, sa partie information est complètement ignorée, et il pointe vers le premier élément de la liste. Ce nœud nous évitera de considérer les cas où la liste n'a qu'un seul élément, les cas où elle est vide et le cas où le pointeur courant est sur le début de la liste.

Dans ce cas, initialement on a :



Si on reprend notre exemple on aurait:



Dans ce cas, l'insertion du 10 se fera normalement entre le 20 et le 23.

Cette opération qui se fait en trois étapes :

1. créer un nouveau nœud,
2. mettre son suivant vers l'élément courant actuel,
3. faire pointer le champ suivant du nœud précédent le nœud courant vers le nouveau nœud.

Il est facile de voir l'opération d'insertion se fait en un temps constant $\theta(1)$.

L'opération de suppression d'un élément se fait aussi en $\theta(1)$ puisqu'elle consiste essentiellement en des mises à jour de pointeurs. La classe Liste avec cette implantation est :

```
// fichier LISTE.H
// Déclaration de la classe Liste
```

```
#ifndef LISTE_H
#define LISTE_H
```

```
#include "NOEUD.H"
```

```

class Liste{
public :
    Liste () ;                // Constructeur
    ~Liste () ;              // Destructeur
    void ViderListe () ;      // Vide la liste
    void Insérer ( const TElement & ) ; // insère un élément à la position courante
    void InsérerFin ( const TElement & ) ; // insère un élément à la fin de la liste
    TElement Supprimer () ; // Supprime et retourne l'élément à la position courante
    void FixerTete () ; // met la position courante à la tête de la liste
    void Precedent () ; // Déplace la position courante à la position précédente
    void Suivant () ; // Déplace la position courante à la position suivante
    int Longueur () const ; // retourne la longueur courante de la liste
    void FixerPosition ( const int ) ; // met position courante à position donnée
    void FixerValeur ( const TElement & ) ; // met à jour la valeur à la position courante
    TElement ValeurCourante () const ; //retourne la valeur d'élément à la position courante.
    bool ListeVide () ; // retourne vrai si la liste est vide
    bool EstDansListe () const ; //retourne vrai si position courante est dans la liste
    bool Trouver ( const TElement & ) ; // recherche une valeurs dans la
                                     // liste à partir de la position courante.

private :
    noeud * Tete ; // position du premier élément
    noeud * fin ; // position du dernier élément
    noeud * Courant ; // position de l'élément courant
} ;
#endif

```

// Fichier LISTE.CPP

```

#include <cassert>
#include "LISTE.H"

```

```

Liste::Liste () // Constructeur
{
    fin = Tete = Courant = new noeud ; // crée le noeud entête
}

Liste::~~Liste () // Destructeur
{
    while ( Tete != NULL )
    {
        Courant = Tete ;
        Tete = Tete->Suivant ;
        delete Courant ;
    }
}

```

```

void Liste::ViderListe ()
{
    // libère l'espace alloué aux noeuds, garde l'entête.
    while ( Tete->Suivant != NULL )
    {
        Courant = Tete->Suivant ;
        Tete->Suivant = Courant->Suivant ;
        delete Courant ;
    }
    Courant = fin = Tete ; // réinitialise
}

// Insère un élément à la position courante

void Liste::Inserer ( const TElement & element )
{
    assert ( Courant != NULL ) ; // s'assure que Courant pointe vers un noeud
    Courant->Suivant = new noeud ( element, Courant->Suivant ) ;
    if ( fin == Courant )
        fin = Courant->Suivant ; //l'élément est ajouté à la fin de la liste.
}

TElement Liste::Supprimer ()          // supprime et retourne l'élément courant
{
    assert ( EstDansListe() ) ; // Courant doit être une position valide
    TElement temp = Courant->Suivant->element ; //Sauvegarde de l'élément courant
    noeud * ptemp = Courant->Suivant ; // Sauvegarde du pointeur du noeud Courant
    Courant->Suivant = ptemp->Suivant ; // suppression de l'élément
    if ( fin == ptemp )
        fin = Courant ; // C'est le dernier élément supprimé, mise à jour de Fin
    delete ptemp ;
    return temp ;
}

void Liste::InsererFin ( const TElement & element ) // insère en fin de liste
{
    fin = fin ->Suivant = new noeud (element, NULL) ;
}

void Liste::FixerTete () // rend la tête comme position courante
{
    Courant = Tete ;
}

void Liste::Precedent () // met la position courante à la position précédente
{
    noeud * temp = Tete ;

```

```

    if (( Courant == NULL ) || ( Courant == Tete ) ) // pas d'élément précédent
    {
        Courant = NULL ;
        return ;
    }
    while ( (temp != NULL) && (temp->Suivant != Courant) )
        temp = temp->Suivant ;
    Courant = temp ;
}

void Liste::Suivant ()
{
    if ( Courant != NULL )
        Courant = Courant->Suivant ;
}

int Liste::Longueur () const
{
    int cpt = 0 ;
    for ( noeud * temp = Tete->Suivant; temp != NULL; temp = temp->Suivant )
        cpt++ ; // compte le nombre d'éléments
    return cpt ;
}

void Liste::FixerPosition ( const int pos )
{
    Courant = Tete ;
    for ( int i = 0 ; ( Courant != NULL ) && ( i < pos ); i++ )
        Courant = Courant->Suivant ;
}

void Liste::FixerValeur ( const TElement & valeur )
{
    assert ( EstDansListe() ) ;
    Courant->Suivant->element = valeur ;
    return ;
}

TElement Liste::ValeurCourante () const
{
    assert ( EstDansListe() ) ;
    return Courant->Suivant->element ;
}

bool Liste::EstDansListe () const
{
    return ( Courant != NULL ) && ( Courant->Suivant != NULL ) ;
}

```

```

bool Liste::ListeVide ()
{
    return Tete->Suivant == NULL ;
}

bool Liste::Trouver ( const TElement & valeur )    // recherche la valeur à partir
                                                    // de la position courante
{
    while ( EstDansListe() )
        if ( Courant->Suivant->element == valeur )
            return true ;
        else
            Courant = Courant->Suivant ;
    return false ;
}

```

3.4 Comparaison des implantations de la liste

L'implantation de la liste par tableau impose le choix d'une taille maximum de la liste. Beaucoup d'espace risque d'être inutilisé. Dans le cas de l'implantation par liste chaînée, l'espace est alloué uniquement aux éléments qui appartiennent effectivement à la liste.

Dans le cas de l'implantation par tableau aucune mémoire supplémentaire n'est nécessaire pour stocker un élément de la liste. Dans le cas de liste chaînée, un espace pour le pointeur est ajouté pour chaque élément de la liste.

L'accès à un élément par sa position est plus rapide dans le cas de l'implantation par tableau, il se fait en un temps constant ($O(1)$). Dans le cas de la liste chaînée, nous devons parcourir la liste du début jusqu'à la position désirée ($O(n)$). Les insertions et les suppressions d'éléments sont plus coûteuses dans le cas dans l'implantation par tableau car elles nécessitent des déplacements d'éléments.

Listes avec tableau:

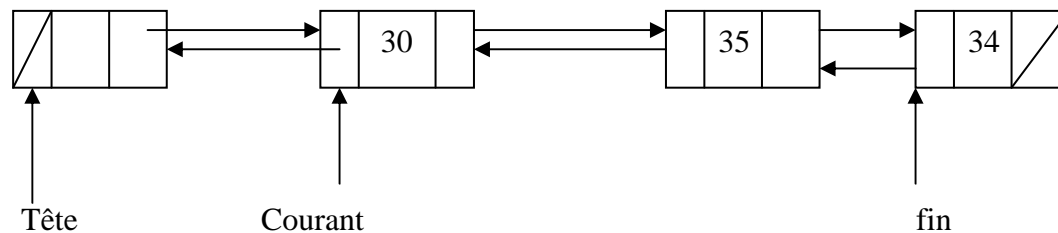
1. Insertion et suppression sont $\Theta(n)$.
2. Précédent et accès direct sont $\Theta(1)$.
3. Tout l'espace est alloué à l'avance.
4. Pas d'espace autre que les valeurs.

Listes chaînées:

1. Insertion et suppression sont $\Theta(1)$.
2. Précédent et accès direct sont $\Theta(n)$.
3. L'espace augmente avec la liste.
4. Chaque élément requiert de l'espace pour les pointeurs

Les listes doublement chaînées : Dans les listes simplement chaînées, à partir d'un nœud donné, on ne peut accéder qu'au successeur de ce nœud. Dans une liste doublement chaînée, à partir d'un nœud donné, on peut accéder au nœud successeur et au nœud prédécesseur.

Exemple:



Cela a pour conséquence qu'un nœud de cette liste va avoir trois champs :

1. un ou des champs information,
2. un pointeur vers le successeur,
3. un pointeur vers le prédécesseur.

Dans le cas de la liste doublement chaînée, on pourrait avoir le pointeur Courant, directement sur l'élément courant car il est possible d'accéder au prédécesseur. Cependant, nous allons garder la même convention que dans le cas de la liste simplement chaînée (Courant pointe une position avant l'élément courant) pour:

- éviter un cas spécial quand on fait une insertion dans une liste vide
- permettre l'insertion d'un nœud à n'importe quelle position de la liste, y compris en fin de liste.

Pour implanter la liste doublement chaînée, il faut redéfinir le nœud comme suit :

// fichier NOEUD.H

// Déclaration de la classe Noeud

#ifndef NOEUD_H

#define NOEUD_H

template <classe Telement>

class noeud

{

public :

 Telement element ;

 noeud * Suivant ;

 noeud * Precedent ;

 noeud (const TElement & info, noeud * suiv = NULL, noeud * prec = NULL)

 { // constructeur1

 element = info ;

 Suivant = suiv ;

 Precedent = prec ;

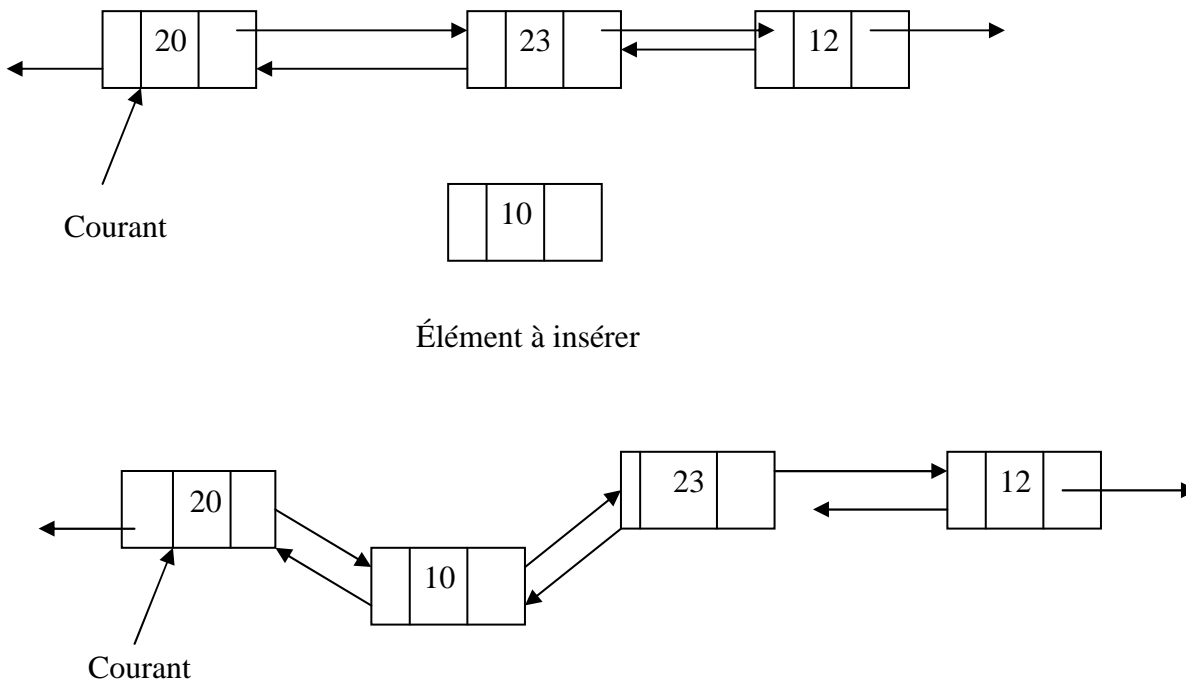

```

    }
    noeud ( noeud * suiv = NULL, noeud * prec = NULL ) // constructeur 2
    {
        Suivant = suiv ;
        Precedent = prec ;
    }
    ~noeud ()
    {
    }
};
#endif

```

La liste doublement chaînée peut être implantée de la même façon que la liste simplement chaînée. La partie privée reste inchangée. Cependant, les fonctions d'insertion à la position courante, l'insertion en fin, la suppression sont à définir ainsi qu'une fonction qui nous permet d'avoir le prédécesseur. Les autres fonctions sur les listes chaînées simples peuvent être utilisées sans changement dans le cadre des listes doublement chaînées.

- **l'insertion à la position courante :**



Cette insertion est réalisée par les instructions suivantes :

```

Courant->suivant = new noeud ( 10, Courant->suivant, Courant ) ;
if ( Courant->suivant->suivant != NULL )
    Courant->suivant->suivant->precedent = Courant->suivant ;

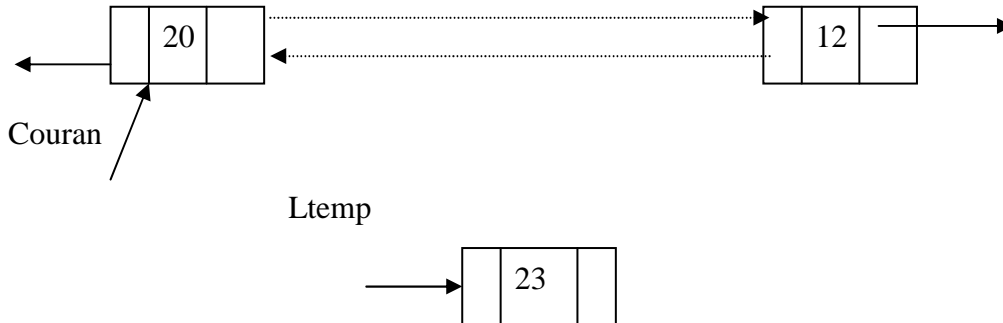
```

Le test permet de s'assurer que l'insertion n'est pas à la fin de la liste, auquel cas il faut mettre à jour le champ precedent du nœud après le nouveau nœud.

- **Ajout à la fin de la liste :**

```
fin->suivant = new nœud ( elem, NULL, fin ) ;
fin = fin->suivant ;
```

- **Suppression de l'élément à la position courante :**



```
if ( ltemp->suivant != NULL )
    ltemp->suivant->precedent = Courant ;
else
    fin = Courant ; // C'est le dernier élément qui est supprimé.
Courant->suivant = ltemp->suivant;
```

// ajout d'un élément à la position courante

```
void Liste::Inserer ( const TElement & element )
{
    assert ( Courant != NULL ) ;
    Courant->suivant = new nœud ( elem, Courant->suivant, Courant ) ;
    if ( Courant->suivant->suivant != NULL )
        Courant->suivant->suivant->precedent = Courant->suivant ;
    if ( fin == Courant )
        fin = Courant->suivant ;
}
```

// ajout d'un élément à la fin de la liste

```
void Liste::InsererFin ( const TElement & elem )
{
    fin->suivant = new nœud ( elem, NULL, Fin ) ;
    fin = fin->suivant ;
}
```

// Supprime et retourne l'élément à la position courante

TElement Liste::Supprimer ()

```
{
    assert ( EstDansListe ( ) );
    int temp = Courant->suivant->element ;
    nœud * ltemp = Courant->suivant ;
    if ( ltemp->suivant != NULL )
        ltemp->suivant->precedent = Courant ;
    else
        fin = Courant // C'est le dernier élément qui est supprimé.
    Courant->suivant = ltemp->suivant ;
    delete ltemp ;
    return temp;
}
```

// Déplace le pointeur Courant à la position précédente

void Liste::Precedent()

```
{
    if ( Courant != NULL )
        Courant = Courant->precedent ;
}
```

Les listes circulaires

Une liste où le pointeur NULL du dernier élément est remplacé par l'adresse du premier élément est appelée liste circulaire.

Dans une liste circulaire tous les nœuds sont accessibles à partir de n'importe quel autre nœud. Une liste circulaire n'a pas de premier et de dernier nœud. Par convention, on peut prendre le pointeur externe de la liste vers le dernier élément et le suivant serait le premier élément de la liste. Par ailleurs, un pointeur nul signifie une liste vide.

Une liste circulaire peut être simplement chaînée ou doublement chaînée. Notez que la concaténation de deux listes circulaires peut se faire sans avoir à parcourir les deux listes.

Les Piles

3-1 Définition et exemple

La pile est une structure très utilisée en informatique, surtout dans les langages de programmation (appel de procédures, calcul des expressions arithmétiques,...).

La pile est une liste ordonnée sans fin d'éléments dans laquelle on ne peut introduire ou enlever un élément qu'à une extrémité appelée tête de pile ou sommet de pile. Exemple une pile de livres, une pile d'assiettes, une pile à un jeu de cartes si l'on se refuse le droit de manipuler plus d'un de leurs éléments à la fois. Noter que dans une pile, le dernier élément inséré sera le premier à être supprimé, retiré : on parle de liste LIFO 'Last In First Out'.

3-2 Opérations sur les piles :

Les opérations de base sur les piles sont très simples.

- **Empiler (p,e)** (en anglais **push**) : empile l'élément e dans la pile p
- **Dépiler (p)** (en anglais **pop**) : dépile un élément de la pile et retourne la valeur de cet élément ($e = \text{Depiler}(p)$)

Toutes les opérations sont effectuées sur la même extrémité; on parle de structure en FIFO.

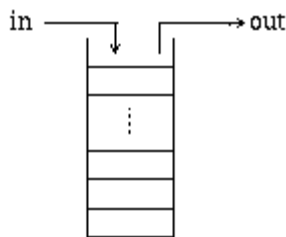


Fig. 7.5. Structure de pile.

Il n'y a aucune limite au nombre d'éléments qu'on peut empiler. Par contre, on ne peut dépiler d'une pile vide. A cet effet, on a une fonction **PileVide(p)** qui retourne Vrai si la pile est vide sinon Faux.

InitialiserPile(p) ou ViderPile(p) : vide la pile p

On peut vouloir aussi connaître le sommet de pile sans le supprimer (dépiler).

e = SommetDePile(p) : Ce n'est pas réellement une nouvelle opération car on peut l'exprimer comme suit :

$e = \text{Dépiler}(p)$

Empiler (e,p)

SommetDePile() et Dépiler() ne fonctionnent pas sur une pile vide. Aussi, on ne peut empiler dans une pile pleine.

Implantation par tableau

Il faut être bien conscient qu'un tableau est différent d'une pile. Un tableau a une taille fixe alors que la pile est une structure dynamique: sa taille change à chaque fois que de nouveaux éléments sont empilés ou dépilés. Cependant, on peut utiliser un tableau pour accueillir une pile en respectant les conditions suivantes :

- Dimensionner le tableau au nombre d'éléments que peut contenir la pile.
- Restreindre les éléments de la pile à un type identique.
- Utiliser une variable spéciale, *sommet*, contenant à tout instant l'indice de la case du tableau où se trouve l'élément du sommet de pile.

Une pile en C++ est une classe définie comme suit.

```
// Fichier Pile.h
```

```
// declaration de la classe pile
```

```
#ifndef PILE_H
```

```
#define PILE_H
```

La pile générique (template) :

```
#include <cassert>
```

```
const Taille_de_Pile = 10;
```

```
template <class T>
```

```
class Pile
```

```
{
```

```
private :
```

```
    int Taille;           // Taille Maximum de la pile
```

```
    int Sommet;          // indice de la première position libre dans le tableau
```

```
    T * TabElement;      // Tableau contenant les éléments de la pile
```

```
public :
```

```
    Pile (const int dim = Taille_de_Pile ) ; // Constructeur
```

```
    ~Pile () ;           // destructeur : libère l'espace alloué à la pile
```

```
    void ViderPile ();    // enlève tous les éléments de la pile
```

```
    void Empiler ( const T & ); // ajoute un élément au dessus de la pile
```

```
    T Depiler ();         // retire un élément du sommet de la pile
```

```
    T Sommet_De_Pile () const; // Renvoie la valeur du sommet de la pile
```

```
    bool PileVide () const; // Renvoie vrai (true) si la pile est vide
```

```
};
```

```

template <class T>
Pile<T>::Pile ( const int dim ) // Constructeur
{
    Taille = dim;
    Sommet = 0;
    TabElement = new T[dim];
}

template <class T>
Pile<T>::~~Pile ()          // destructeur : libère l'espace alloué à la pile
{
    delete TabElement;
}

template <class T>
void Pile<T>::ViderPile ()
{
    Sommet = 0;
}

template <class T>
void Pile<T>::Empiler ( const T & elem )
{
    assert ( Sommet < Taille );
    TabElement[Sommet++] = elem;
}

template <class T>
T Pile<T>::Depiler ()
{
    assert( !PileVide() );
    return TabElement[--Sommet];
}

template <class T>
T Pile<T>::Sommet_De_Pile () const
{
    assert ( !PileVide() );
    return TabElement[Sommet-1];
}

template <class T>
bool Pile<T>::PileVide () const
{
    return Sommet == 0;
}

```

```
}
```

La déclaration `Pile <int> p1(6)` ; crée une instance de pile d'entiers de 6 éléments.

La déclaration `Pile <char> p1(6)` ; crée une instance de pile de caractères de 6 éléments.

Noter l'utilisation de `assert (!PileVide)` ; Cette instruction permet de vérifier que la condition indiquée est vraie. Ici, si la condition est vraie c'est-à-dire `PileVide()` est faux, le programme se poursuit sinon, il s'arrête.

Une autre possibilité est de prévoir dans la fonction `Depiler()` et `Sommet_De_Pile()`, une variable booléenne qui indique si la fonction a réussi ou échoué. Cette variable sera testée à chaque retour de l'appel de la fonction.

Pour utiliser la fonction `assert()` dans un programme, il faut inclure le fichier `<cassert>`.

Il en est de même pour la fonction `Empiler()`, il serait préférable de prévoir une variable qui indiquera au programme appelant si la pile est pleine ou pas, ceci permettra au programme appelant d'effectuer les actions nécessaires dans un pareil cas.

Il faut cependant noter que la contrainte de pile pleine n'est pas une contrainte liée à la structure de donnée *pile*. Théoriquement, il n'y a pas de limite sur le nombre d'éléments d'une pile. En pratique, son implantation dans un tableau limite le nombre d'éléments à *Taille*.

Exemple d'utilisation de la classe Pile:

On considère un programme qui lit 5 entiers, les empile dans une pile et les affiche.

```
#include <iostream>
    using namespace std ;
#include "Pile.h"

int main ()
{
    int n ;
    Pile pile1(5) ;    // 5 est la taille de la pile

    cout << "Entrez 5 entiers : " ;
    for ( int i = 1; i <= 5; i++ )
    {
        cin >> n ;
        pile1.Empiler(n) ;    // remplissage de la pile
    }

    while ( !pile1.PileVide() )    // affichage des éléments de la pile
        cout << pile1.Depiler() << " " << endl ;

    return 0 ;
}
```

```
}
```

Implantation par liste chaînée

L'implantation d'une pile par une liste (simplement) chaînée est une version simplifiée de celle d'une liste par liste chaînée.

```
template <class Elem>
class Pile {
private:
    noeud * top;          // sommet de la pile
    int size;             // nombre d'élément dans la pile
public:
    Pile ()               // constructeur
    {
        top = NULL; size = 0;
    }
    ~Pile() { clear(); }  // Destructeur
    void clear() {
        while (top != NULL) { // Delete link nodes
            noeud * temp = top;
            top = top->suivant;
            delete temp;
        }
        size = 0;
    }

    bool push(const Elem& item) {
        top = new noeud(item,top);
        size++;
        return true;
    }
    bool pop(Elem& it) {
        if (size == 0) return false;
        it = top->element;
        noeud * ltemp = top->suivant;
        delete top;
        top = ltemp;
        size--;
        return true;
    }
    bool sommet_pile(Elem& it) const {
        if (size == 0) return false;
        it = top->element;
        return true;
    }
}
```



```
int length() const { return size; } // retourne la taille de la pile
};
```

Comparaison des deux implantations

Toutes les implantations des opérations d'une pile utilisant un tableau ou une liste chaînée prennent un temps constant *i.e.* en $O(1)$. Par conséquent, d'un point de vue d'efficacité, aucune des deux implantations n'est mieux que l'autre. D'un point de vue de complexité spatiale, le tableau doit déclarer une taille fixe initialement. Une partie de cet espace est perdue quand la pile n'est pas bien remplie. La liste peut augmenter ou rapetisser au besoin mais demande un extra espace pour mémoriser les adresses des pointeurs.

Quelques applications des piles

1. Reconnaissance syntaxique : Soit une chaîne de caractères définie par la règle suivante :

Une chaîne quelconque S suivie du caractère $*$, suivi de la chaîne S inversée.

Exemple $abc*cba$

La chaîne peut contenir n'importe quel caractère alphanumérique. La chaîne se termine par le marqueur fin de ligne.

Conception : Pour déterminer si la chaîne est légale, il faut :

- 1) Lire jusqu'à $*$ les caractères un à un en les empilant dans une pile.
- 2) Après $*$, jusqu'à la fin de la chaîne, lire un caractère, dépiler le caractère au sommet de la pile et comparer les deux, s'ils ne sont pas égaux, la chaîne est invalide.
- 3) Si tous les caractères sont égaux et que la pile s'est vidée, la chaîne est valide.

Une autre application est celle de la vérification du nombre de parenthèses ouvrante et fermantes dans une expression arithmétique.

2. Calcul arithmétique : Une application courante des piles se fait dans le calcul arithmétique: l'ordre dans la pile permet d'éviter l'usage des parenthèses. La notation postfixée (polonaise) consiste à placer les opérandes devant l'opérateur. La notation infixée (parenthésée) consiste à entourer les opérateurs par leurs opérandes. Les parenthèses sont nécessaires uniquement en notation infixée. Certaines règles permettent d'en réduire le nombre (priorité de la multiplication par rapport à l'addition, en cas d'opérations unaires représentées par un caractère spécial $(-, !, \dots)$). Les notations préfixée et postfixée sont d'un emploi plus facile puisqu'on sait immédiatement combien

d'opérandes il faut rechercher. Détaillons ici la saisie et l'évaluation d'une expression postfixée:

La notation usuelle, comme $(3 + 5) * 2$, est dite infixée. Son défaut est de nécessiter l'utilisation de parenthèses pour éviter toute ambiguïté (ici, avec $3 + (5 * 2)$). Pour éviter le parenthésage, il est possible de transformer une expression infixée en une expression postfixée en faisant "glisser" les opérateurs arithmétiques à la suite des expressions auxquelles ils s'appliquent.

Exemple:

$(3 + 5) * 2$ s'écrira en notation postfixée (notation polaise): $3\ 5\ +\ 2\ *$

alors que $3 + (5 * 2)$ s'écrira: $3\ 5\ 2\ *\ +$

Notation infixée: $A * B / C$. En notation postfixée est: $AB * C /$.

On voit que la multiplication vient immédiatement après ses deux opérandes A et B . Imaginons maintenant que $A * B$ est calculé et stocké dans T . Alors la division $/$ vient juste après les deux arguments T et C .

Forme infixée: $A / B^{**} C + D * E - A * C$ (** étant l'opérateur d'exponentiation)

Forme postfixée: $ABC^{**} / DE * + AC * -$

Evaluation en Postfix

Considérons l'expression en postfixe suivante:

6 5 2 3 + 8 * + 3 + *

Algorithm

```
Initialiser la pile à vide;
while (ce n'est pas la fin de l'expression postfixée) {
    prendre l'item prochain de postfixe;
    if(item est une valeur)
        empiler;
    else if(item est un opérateur binaire) {
        dépiler dans x;
        dépiler dans y;
        effectuer y opérateur x;
        empiler le résultat obtenu;
    } else if (item est un opérateur unaire) {
        dépiler dans x;
        effectuer opérateur(x);
        empiler le résultat obtenu;
    }
}
```

la seule valeur qui reste dans la pile est le résultat recherché.

Opérateurs binaires: +, -, *, /, etc.,

Opérateurs unaires: moins unaire, racine carrée, sin, cos, exp, ... etc.

Pour **6 5 2 3 + 8 * + 3 + ***

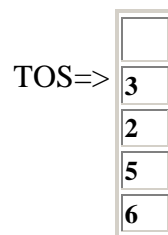
Le premier item est une valeur (6); elle est empilée.

Le deuxième item est une valeur (5); elle est empilée.

Le prochain item est une valeur (2); elle est empilée.

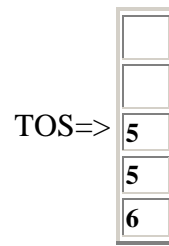
Le prochain item est une valeur (3); elle est empilée.

La pile devient

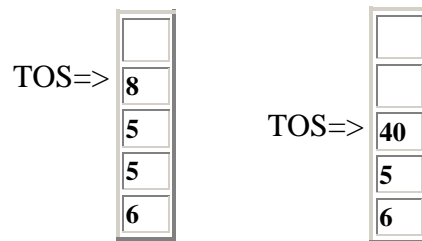


Les items restants à cette étape sont: **+ 8 * + 3 + ***

Le prochain item lu est '+' (opérateur binaire): 3 et 2 sont dépilés et leur somme '5' est ensuite empilée:

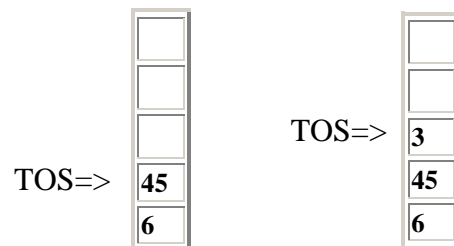


Ensuite **8** est empilé et le prochain opérateur *:



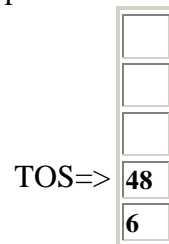
(8, 5 sont dépilés, 40 est empilé)

Ensuite l'opérateur + suivi de **3**:



(40, 5 sont dépilés ; 45 pushed, 3 est empilé)

Ensuite l'opérateur +: 3 et 45 sont dépilés et **45+3=48** est empilé



Ensuite c'est l'opérateur *: 48 and 6 sont dépilés et **6*48=288** est empilé



Il n'y plus d'items à lire dans l'expression postfixée et aussi il n'y a qu'une seule valeur dans la pile représentant la réponse finale: **288**.

La réponse est trouvée en balayant une seule fois la forme postfixée. La complexité de l'algorithme est par conséquent en $O(n)$; n étant la taille de la forme postfixée. La pile a été utilisée comme une zone tampon sauvegardant des valeurs attendant leur opérateur.

Infixe à Postfixe

Bien entendu la notation postfixe ne sera pas d'une grande utilité s'il n'existait pas un algorithme simple pour convertir une expression infixe en une expression postfixe. Encore une fois, cet algorithme utilise une pile.

L'algorithme









```
initialise la pile et l'output postfixe à vide;
while(ce n'est pas la fin de l'expression infixe) {
  prendre le prochain item infixe
  if (item est une valeur) concaténer item à postfixe
  else if (item == '(') empiler item
  else if (item == ')') {
    dépiler sur x
    while(x != '(')
      concaténer x à postfixe & dépiler sur x
  }
  else {
    while(precedence(stack top) >= precedence(item))
      dépiler sur x et concaténer x à postfixe;
    empiler item;
  }
}
while (pile non vide)
  dépiler sur x et concaténer x à postfixe;
```

Précédence des opérateurs (pour cet algorithme):

```
4 : '(' – déplée seulement si une ')' est trouvée
3 : tous les opérateurs unaires
2 : / *
1 : + -
```

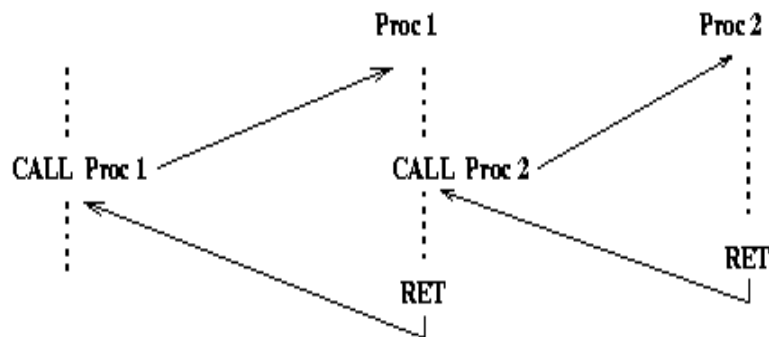
L'algorithme passe les opérandes à la forme postfixe, mais sauvegarde les opérateurs dans la pile jusqu'à ce que tous les opérandes soient tous traduits.

Exemple: considérons la forme infixe de l'expression $a+b*c+(d*e+f)*g$

Stack	Output
TOS=> 	ab
TOS=> 	abc
TOS=> 	abc*+
TOS=> 	abc*+de
TOS=> 	abc*+de*f
TOS=> 	abc*+de*f+
TOS=> 	abc*+de*f+g
empty 	abc*+de*f+g*+

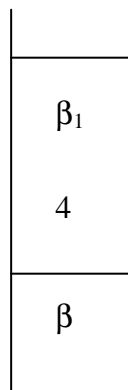
3. Appels de fonctions et implantation de la récursivité

Lors de l'appel d'une fonction, toutes les informations importantes, qui ont besoin d'être sauvegardées, telles que les valeurs des registres (correspondant aux noms des variables), l'adresse de retour, sont stockées dans une pile (activation). Le contrôle est ensuite transféré à la nouvelle fonction, qui est libre de remplacer les valeurs des registres avec ses propres valeurs. Si la nouvelle fonction fait un appel, le même processus est répété (sauvegarde des infos dans la pile). Quand la fonction termine, elle remet à jour les registres à l'aide des informations du sommet de la pile, le contrôle est ensuite transféré à la fonction appelante à l'adresse de retour.



C'est suivant ce principe, bien simple, qu'il est possible au processeur d'effectuer l'exécution des fonctions récursives. Voyons cela sur un exemple d'une fonction récursive.

Reprenons l'exemple de la fonction factoriel (n) avec $n = 4$. On utilise β pour indiquer l'adresse de retour à la fonction appelante. Au départ, la fonction doit sauvegarder β et la valeur 4 est passée à factoriel. Ensuite, un appel récursif à factoriel est effectué, cette fois-ci avec la valeur 3. Si β_1 est l'adresse où cet appel est fait, alors β_1 et la valeur 4 sont sauvegardées dans la pile. La fonction factoriel est maintenant exécutée avec le paramètre 3.



La fonction factoriel est maintenant exécutée avec le paramètre 3. De la même manière, un autre appel est récursif est effectué avec un paramètre $n = 2$ nécessitant la sauvegarde de l'adresse β_2 et

la valeur courante de $n = 3$. Ainsi de suite, jusqu'à arriver à factoriel (1). A ce stade, la pile contient les informations suivantes :

β_3
2
β_2
3
β_1
4
β

Une fois factoriel(1) atteint, les appels récursifs s'arrêtent. À partir de là, la récursion commence à retourner des résultats. Ainsi, chaque return de factoriel implique un dépilement de la pile de la valeur de n et celle de l'adresse de retour de la fonction appelante.

Comme chaque appel de fonction implique une activation est créée, les appels de fonctions sont assez coûteux en temps et en espace mémoire. Même si la récursion est utilisée pour permettre une implantation claire et facile, on souhaite souvent éliminer les overheads générés par les appels récursifs. Dans certains cas, la récursion peut être facilement remplacée par l'itération (comme c'est le cas de la fonction factoriel).

Dans notre cas, on a tout simplement à empiler les valeurs de n jusqu'à atteindre le test d'arrêt. Ensuite, on commence à partir de là à dépiler de la pile les valeurs sauvegardées et à les multiplier au fur et à mesure avec le résultat courant.

Cela nous donne la fonction suivante :

```

long factoriel(int n, Pile<int> & S)
{
    //pour avoir n! dans une variable de type long nécessite n <= 12

    assert ( (n >= 0 && (n <= 12)) );
    while (n > 1) S.push(n--);
    long result = 1;
    while (S.pop(val)) result = result * val;
    return result;
}

```

En pratique, une forme itérative de la fonction factoriel est plus simple et rapide que la version ci-dessous utilisant une pile. Malheureusement, il n'est pas toujours possible ainsi. Dans certaines situations, la récursivité est nécessaire pour pouvoir résoudre le problème en question comme dans le cas des tours de Hanoi, de la travers d'arbres et de certains algorithmes de tri tels que le tri rapide ou le tri par fusion.

Reprenons l'exemple de Hanoi :

```

typedef int Pole;
#define move(X, Y)
    cout << " Déplacer " << (X) << " vers " << (Y) << endl
void TOH(int n, Pole start, Pole goal, Pole tmp)
{
    if (n == 0) return;           // Cas de base
    TOH(n-1, start, tmp, goal); // Appel récursif: n-1 disques
    move(start, goal);           // Déplacer un disque
    TOH(n-1, tmp, goal, start); // Appel récursif n-1 disques
}

```

La fonction TOH fait deux appels récursifs: un déplacement de n-1 disques de start à tmp et un déplacement de n-1 disques de tmp à goal. On peut éliminer la récursion en utilisant une pile pour sauvegarder une représentation que TOH doit effectuer : 2 appels et une opération de déplacement. Pour y arriver, nous devons avant tout avoir une des différentes opérations, implantées comme une classe dont les membres vont être sauvegardés dans une pile. Cette classe peut être comme ci-dessous :

```

enum TOHop { DOMOVE, DOTOH };
class TOHobj {
public:
    TOHop op;
    int num;
    Pole start, goal, tmp;
    TOHobj(int n, Pole s, Pole g, Pole t) {
        op = DOTOH; num = n;
        start = s; goal = g; tmp = t;
    }
    TOHobj(Pole s, Pole g)
    { op = DOMOVE; start = s; goal = g; }
};

```

La classe Tohobjet stocke 5 valeurs: un indicateur Pour savoir s'il s'agit d'un mouvement ou d'une nouvelle opération TOH, le nombre de disques, et les trois poles. Noter que l'opération de mouvement a seulement besoin de stocker l'information sur deux poles. Par conséquent, il ya deux constructeurs : un pour stocker l'état pour imiter un appel récursif et un autre pour stocker l'état d'une opération de mouvement. La version non-récursive de TOH est alors comme suit :

```

void TOH(int n, Pole start, Pole goal, Pole tmp, Stack<TOHobj*>& S){
    S.push(new TOHobj(n, start, goal, tmp)); // Initialisation
    TOHobj* t;
    while (S.pop(t)) { // Déterminer la prochaine tâche
        if (t->op == DOMOVE) // Déplacement
            move(t->start, t->goal);
        else if (t->num > 0) { // 3 étapes de la récursion en ordre inverse
            int num = t->num;
            Pole tmp = t->tmp, goal = t->goal, start = t->start;
            S.push(new TOHobj(num-1, tmp, goal, start));
            S.push(new TOHobj(start, goal));
            S.push(new TOHobj(num-1, start, tmp, goal));
        }
        delete t; // Must delete the TOHobj we made
    }
}

```

On définit en premier un type d'énumération appelé TOHop, avec deux valeurs : MOVE et TOH, pour indiquer les appels à la fonction MOVE et les appels récursifs à TOH, respectivement.

Question : pourquoi utiliser le tableau pour représenter la pile.

La nouvelle version de TOH commence par placer dans la pile une description du problème initial avec n disques. Le reste de la fonction est simplement une boucle while qui dépile de la pile et exécute les opérations appropriées. Dans le cas d'une opération TOH (pour n>0), nous empilons dans la pile la représentation des trois opérations exécutées par la version récursive.

Toutefois, elles doivent être placées dans un ordre inverse pour qu'elles soient dépliées dans le bon ordre.

Les files

1 Définition et exemples



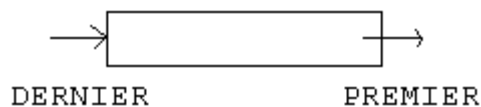
Une file est une structure de données dynamique dans laquelle on insère des nouveaux éléments à la fin (queue) et on enlève des éléments au début (tête de file). L'application la plus classique est la file d'attente. La file sert beaucoup en simulation. Elle est aussi très utilisée aussi bien dans la vie courante que dans les systèmes informatiques. Par exemple, elle modélise la file d'attente des clients devant un guichet, les travaux en attente d'exécution dans un système de traitement par lots, ou encore les messages en attente dans un commutateur de réseau téléphonique. On retrouve également les files d'attente dans les programmes de traitement de transactions telle que les réservations de sièges d'avion ou de billets de théâtre.

Noter que dans une file, le premier élément inséré est aussi le premier retiré. On parle de mode d'accès FIFO (First In First Out).

Comportement d'une pile: Last In First Out (LIFO)

Comportement d'une file: First In First Out (FIFO)

La file est modifiée à ses deux bouts.



2 Les opérations de base

Les opérations primitives sur une file sont **DÉFILER** pour le retrait d'un élément et **ENFILER** pour l'ajout d'un élément. L'opération **ENFILER** (f, e) ajoute l'élément e à la fin de la file f .

Ainsi, la suite des opérations **ENFILER** (f, A), **ENFILER** (f, B) et **ENFILER** (f, C) donnent la file de la figure 1a. Noter qu'en principe, il n'y a aucune limite aux nombres d'éléments que l'on peut ajouter dans une file.

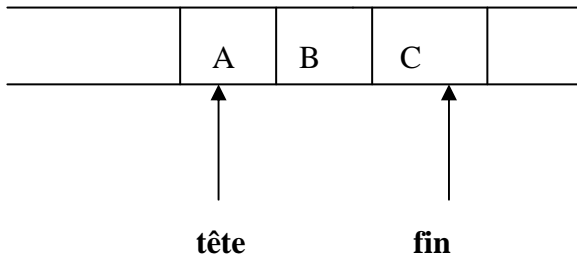


Figure 1a

L'opération $e = \text{DÉFILER}(f)$, retire (supprime) un élément de la fin de la file et renvoie sa valeur dans e .

Ainsi l'opération $e = \text{DÉFILER}(f)$ provoque un changement d'état de la file (voir figure 1b) et met l'élément A dans e .

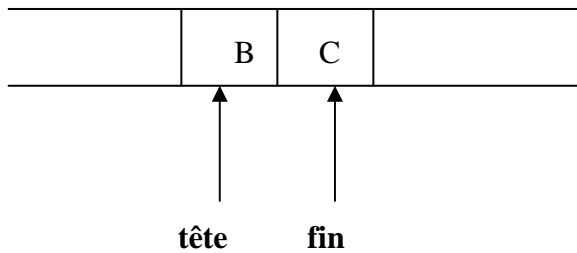


Figure 1b

L'opération **DÉFILER** ne peut s'effectuer sur une file vide. Il faudra donc utiliser une opération **FILEVIDE** qui permet de tester si une file est vide ou pas.

Le résultat des opérations **ENFILER** (f, D) et **ENFILER** (f, E) est donné par la figure 1c

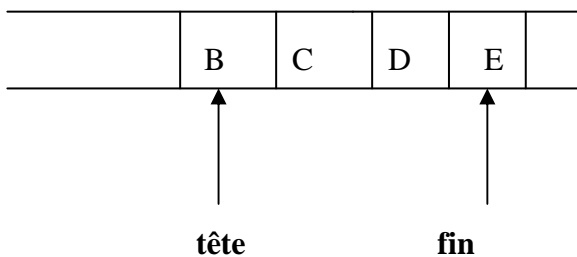


Figure 1c

3 Implantation d'une file par un tableau

Pour représenter une file par un tableau, on utilise deux variables **tête** et **fin** qui contiennent respectivement les indices du premier et du dernier élément de la file.

Les valeurs initiales de **fin** est -1 et **tête** est 0 ; la file sera vide quand $\text{fin} < \text{tête}$.

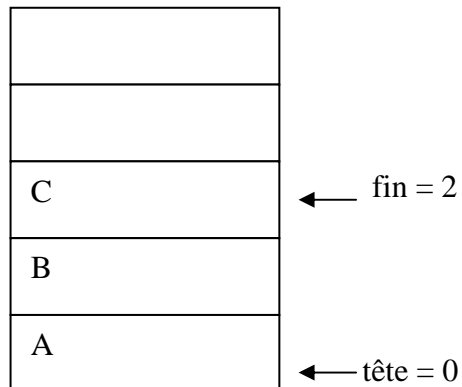
À tout instant le nombre d'éléments de la file est : $\text{fin} - \text{tête} + 1$.

Prenons un exemple avec cette représentation.

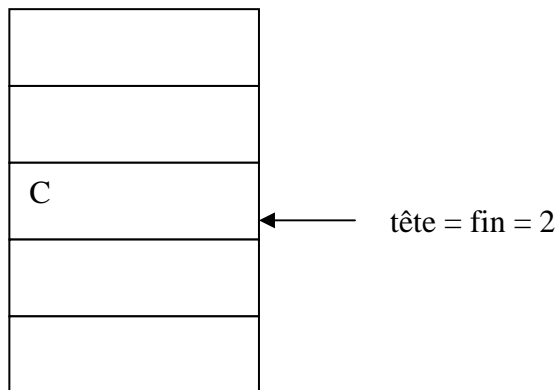
Supposons une file de taille maximum égale à 5. Initialement, la file est vide.

$\text{tête} = 0;$
 $\text{fin} = -1;$

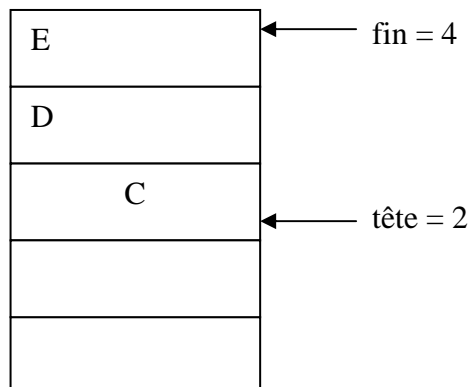
Après l'ajout des éléments A, B et C, la file devient :



Après le retrait de deux éléments on a :



Après l'insertion des éléments D et E, on a :



Si maintenant nous voulons ajouter l'élément F dans la file, fin passerait à 5; or, l'indice maximum dans notre tableau est 4. On ne peut plus ajouter des éléments alors qu'il y a de la place dans le tableau.

Une solution possible consiste à décaler vers le bas tous les éléments de la file à chaque fois qu'on supprime un élément.

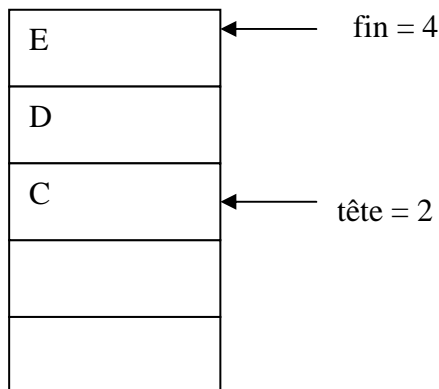
Dans ce cas, il n'est plus nécessaire d'avoir la variable tête de file, puisque l'élément en tête de file sera toujours à la position 0 du tableau. La file vide sera représentée par $\text{fin} = -1$. Ce procédé de décalage d'éléments est simple à réaliser mais coûteux en temps (pourquoi?). De plus, la suppression d'un élément d'une file nécessite logiquement la manipulation d'un seul élément de la file, celui en tête de file.

Implantation par tableau circulaire

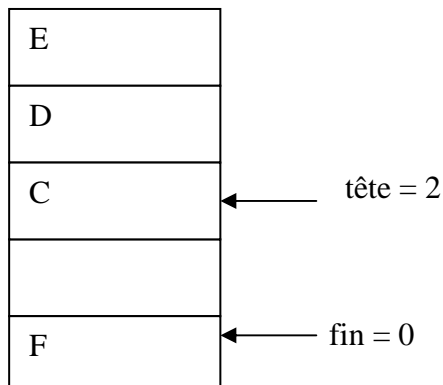
Un solution plus élégante au problème précédent consiste à utiliser *un tableau circulaire*. Quand la valeur de la fin de la file atteint le haut du tableau, on effectue les ajouts des éléments à partir du bas, les indices tête et fin progressent maintenant modulo la taille du tableau. C'est le cas par exemple d'une interface de communication entre deux processus non synchronisés comme un programme et un périphérique d'entrée.

Examinons l'exemple suivant.

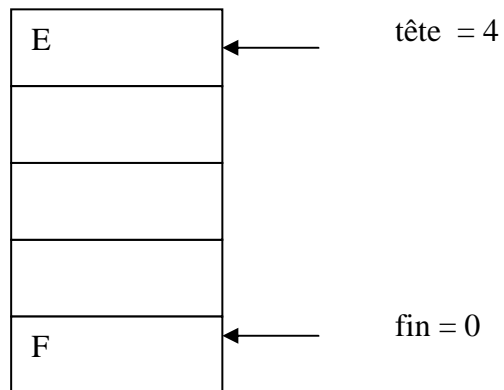
Nous partons avec une file qui contient trois éléments C, D et E.



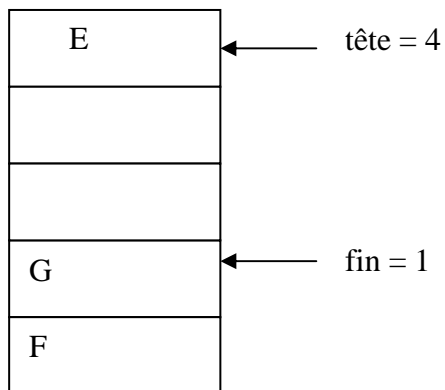
Après l'ajout de l'élément F la file devient :



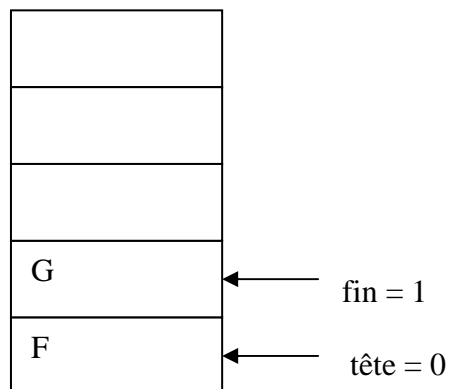
Après la suppression des éléments C et D, la file devient



Après l'ajout de l'élément G la file devient :



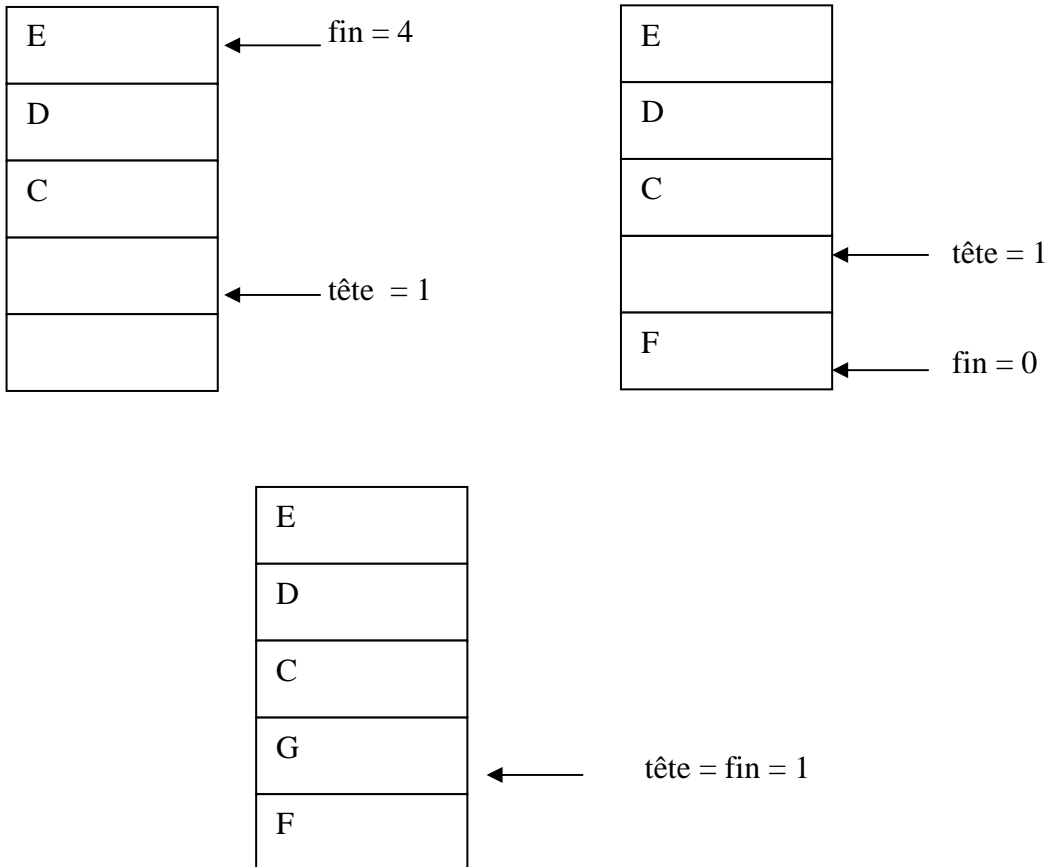
À la suppression de E, la file devient :



À travers cet exemple, on voit surgir un autre problème lié au test de vacuité (file vide) : la condition $\text{fin} < \text{tête}$: cette condition a été vérifiée sans que la file soit vide.

Une solution possible est de choisir tête comme la position du tableau qui précède le premier élément et non la position du premier élément lui même. Comme fin est la valeur de la position du dernier élément dans la file, la file sera vide lorsque la **tête est égale à la fin**.

Examinons d'abord un exemple où l'on insère C, D, E, F et G .



Dans ce cas, la file est pleine et c'est la même condition ($\text{tête} = \text{fin}$) qui est satisfaite quand la file est vide. Il est clair que cette situation n'est pas satisfaisante.

Une solution consiste à laisser au moins l'espace d'un élément entre la fin et la tête : si la taille de la file est de 100 par exemple, il faudra réserver un tableau à 101 éléments.

Question : voyez-vous une autre manière de résoudre ce problème?

Un file en C++ est une classe définie comme suit. Pour simplifier nous allons considérer une file d'entiers.

```

// fichier File.h
// déclaration de la classe File

#ifndef FILE_H
#define FILE_H

const int Taille_de_File = 100 ;

class File
{
public :
    File ( int dim = Taille_de_File ) ;    // Constructeur
    ~File () ;                            // destructeur
    void ViderFile () ;                   // vide la file
    void Enfiler ( const int & ) ;        // ajoute un élément en fin de file
    int Defiler () ;                      // retire l'élément en tête de file
    int PremierElement () const ;         // retourne la valeur en tête de file
    bool FileVide () const ;

private :
    int Taille ;                          // Taille maximum de la file
    int Tete ;
    int Fin ;
    int *TabElement ;    // tableau contenant les éléments de la file
};
#endif

```

```

// Fichier File.cpp
// Définitions des fonctions membres de la classe File

#include <cassert>
#include "File.h"

File::File ( int dim )                    // Constructeur
{
    Taille = dim + 1 ;                    // ajoute une position supplémentaire
    Tete = Fin = Taille - 1 ;
    TabElement = new int [Taille];
}

File::~~File ()
{

```

```

        delete []TabElement;
    }                                     // destructeur

void File::ViderFile ()
{
    Tete = Fin;
}

void File::Enfiler ( const int & element )
{
    assert (((Fin+1) % Taille) != Tete ); // la file n'est pas pleine ?
    Fin = (Fin+1) % Taille; //incrmente fin dans le tableau circulaire
    TabElement[Fin] = element;
}

int File::Defiler ()
{
    assert ( !FileVide() );           // vérifie que la file n'est pas vide
    Tete = ( Tete + 1 ) % Taille;
    return TabElement[Tete]; // retourne la valeur en tête de file.
}

int File::PremierElement () const      // retourne la valeur en tête de file
{
    assert ( !FileVide() );
    return TabElement[(Tete+1) % Taille];
}

bool File::FileVide () const
{
    return Tete == Fin;
}

```

Références

1. Boudreault *et al.* Notes de cours 8INF211, UQAC
2. C.A. Shaffer (2001): A practical introduction to data structures and algorithms (chapitre2)