Chapitre 4

Structures séquentielles

4.1 Les listes linéaires chaînées (LLC)

4.1.1 Notion d'allocation dynamique

L'utilisation des tableaux statiques implique que l'allocation de l'espace se fait tout à fait au début d'un traitement, c'est à dire que l'espace est connu à la compilation. Pour certains problèmes, on ne connaît pas à l'avance l'espace utilisé par le programme. On est donc contraint à utiliser une autre forme d'allocation. L'allocation de l'espace se fait alors au fur et à mesure de l'exécution du programme. Afin de pratiquer ce type d'allocation, deux opérations sont nécessaires : allocation et libération de l'espace.

Exemple

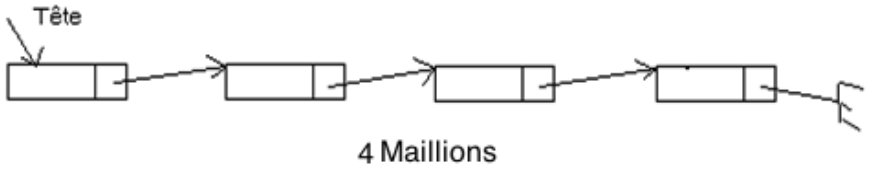
Supposons que l'on veuille résoudre le problème suivant : "Trouver tous les nombres premiers de 1 à n et les stocker en mémoire". Le problème réside dans le choix de la structure d'accueil. Si on utilise un tableau, il n'est pas possible de définir la taille de ce tableau avec précision même si nous connaissons la valeur de n (par exemple 10000). On est donc, ici, en face d'un problème où la réservation de l'espace doit être dynamique.

Un autre inconvénient de l'utilisation des tableaux et que leur espace mémoire est supposé contigu, c'est à dire que chaque élément d'indice i suit immédiatement en mémoire l'élément précédent d'indice $i-1$. Cela empêche les programmes d'être exécuté si l'espace demandé n'existe pas contigu en mémoire. Même si un espace suffisant existe éparpillé en mémoire, il ne peut pas être attribué au programme puisque il ne pourra pas lui accéder par la structure des tableaux.

Les listes linéaires chainées apporte une solution à ces deux problèmes.

4.1.2 Définition d'une liste linéaire chaînée

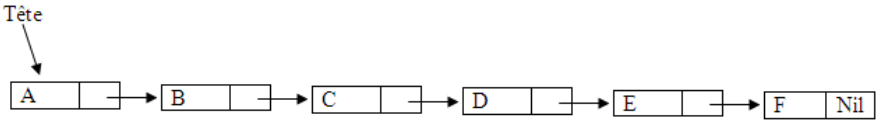
Une liste linéaire chaînée (LLC) ou *"Linear linked lists"* est un ensemble de maillons, alloués dynamiquement, chaînés entre eux. Schématiquement, on peut la représenter comme suit :



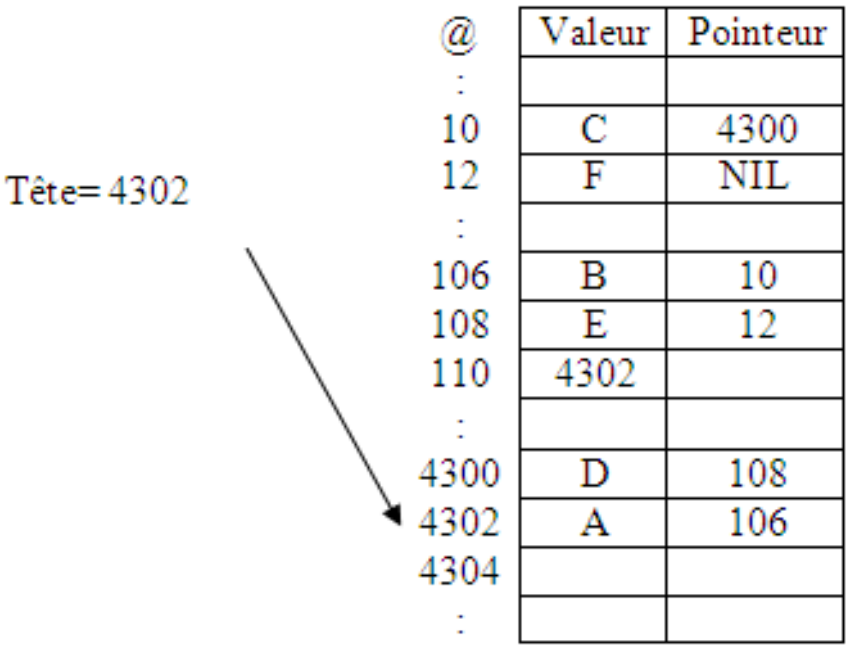
Un élément ou maillon d'une LLC est toujours une structure (Objet composé) avec deux champs : un champ *Valeur* contenant l'information et un champ *Adresse* donnant l'adresse du prochain élément. A chaque maillon est associée une adresse. On introduit ainsi une nouvelle classe d'objet : le type *Pointeur* qui est une variable contenant l'adresse d'un emplacement mémoire.
Une liste linéaire chainée est caractérisée par l'adresse de son premier élément souvent appelé *tête*. *Nil* constitue l'adresse qui ne pointe aucun maillon, utilisé pour indiquer la fin de la liste dans le dernier maillon.

4.1.3 Représentation réelle en mémoire

Soit la liste linéaire chainée représentée par la figure suivante :



La représentation réelle en mémoire de cette liste ressemble à la représentation suivante :



4.1.4 Modèle sur les listes linéaires chaînées

Dans le langage algorithmique, on définira le type d'un maillon comme suit :

```
Type  TMaillon = Structure
    Valeur : Typeqq ; // désigne un type quelconque
    Suivant : Pointeur(TMaillon) ;
Fin ;
```

Afin de développer des algorithmes sur les LLCs, on construit une machine abstraite avec les opérations suivantes : Allouer, Libérer, Aff_Adr, Aff_Val, Suivant et Valeur définies comme suit :

- **Allouer(P) :** allocation d'un espace de taille spécifiée par le type de P. L'adresse de cet espace est rendue dans la variable de type Pointeur P.
- **Libérer(P) :** libération de l'espace pointé par P.
- **Valeur(P) :** consultation du champ Valeur du maillon pointé par P.
- **Suivant(P) :** consultation du champ Suivant du maillon pointé par P.
- **Aff_Adr(P, Q) :** dans le champ Suivant du maillon pointé par P, on range l'adresse Q.

– **Aff_Val(P,Val) :** dans le champ Valeur du maillon pointé par P, on range la valeur Val.

Cet ensemble est appelé modèle.

4.1.5 Algorithmes sur les listes linéaires chaînées

De même que sur les tableaux, on peut classer les algorithmes sur les LLCs comme suit :

– Algorithmes de parcours : accès par valeur, accès par position,...
– Algorithmes de mise à jour : insertion, suppression,...
– Algorithmes sur plusieurs LLCs : fusion, interclassement, éclatement,...
– Algorithmes de tri sur les LLCs : trie par bulle, tri par fusion,...

Création d'une liste et listage de ses éléments

```
Algorithme CréerListe;
Type  TMaillon = Structure
    Valeur : Typeqq ;
    Suivant : Pointeur(TMaillon) ;
Fin ;
Var  P, Q, Tete : Pointeur(TMaillon ;
    i, Nombre : entier ;
    Val : Typeqq ;
Début
| Tete ← Nil ;
| P ← Nil ;
| Lire(Nombre) ;
| Pour i de 1 à Nombre faire
|     Lire(Val) ;
|     Allouer(Q) ;
|     Aff_val(Q, val) ;
|     Aff_adr(Q, NIL) ;
|     Si (Tete ≠ Nil) Alors
|         | Aff_adr(P, Q)
|     Sinon
|         | Tete ← Q
|     Fin Si;
|     P ← Q ;
| Fin Pour;
| P ← Tete ;
| Tant que ( P ≠ Nil) faire
|     | Ecrire(Valeur(P)) ;
|     | P ← Suivant(P) ;
| Fin TQ;
Fin.
```

cet algorithme crée une liste linéaire chaînée à partir d'une suite de valeurs données, puis imprime la liste ainsi créée.

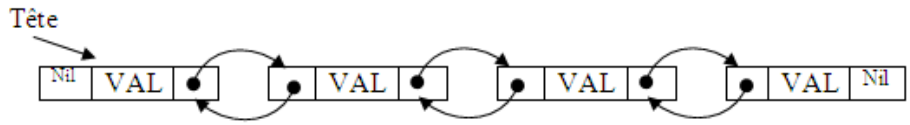
Recherche d'un élément dans une liste

Il s'agit bien entendu de la recherche séquentielle d'une valeur donnée.

```
Algorithme Recherche;
Type  TMaillon = Structure
    Valeur : Typeqq ;
    Suivant : Pointeur(TMaillon) ;
Fin ;
Var  P, Tete : Pointeur(TMaillon ;
    Trouv : booleen ;
    Val : Typeqq ;
Début
| Lire(Val) ;
| P ← Nil ;
| Trouv ← Faux ;
| Tant que ( P ≠ Nil et non Trouv) faire
|     Si (Valeur(P)=Val) Alors
|         Trouv ← Vrai
|     Sinon
|         P ← Suivant(P)
|     Fin Si;
| Fin TQ;
| Si (Trouv=Vrai) Alors
|     Ecrire(Val,'Existe dans la liste')
| Sinon
|     Ecrire(Val,'n"existe pas dans la liste'
| Fin Si;
Fin.
```

4.1.6 Listes linéaires chaînées bidirectionnelles

C'est une LLC où chaque maillon contient à la fois l'adresse de l'élément précédent et l'adresse de l'élément suivant ce qui permet de parcourir la liste dans les deux sens.



Déclaration

```
Type  TMaillon = Structure
    Valeur : Typeqq ; // désigne un type quelconque
    Précédent, Suivant : Pointeur(TMaillon) ;
Fin ;
Var  Tete : TMaillon ;
```

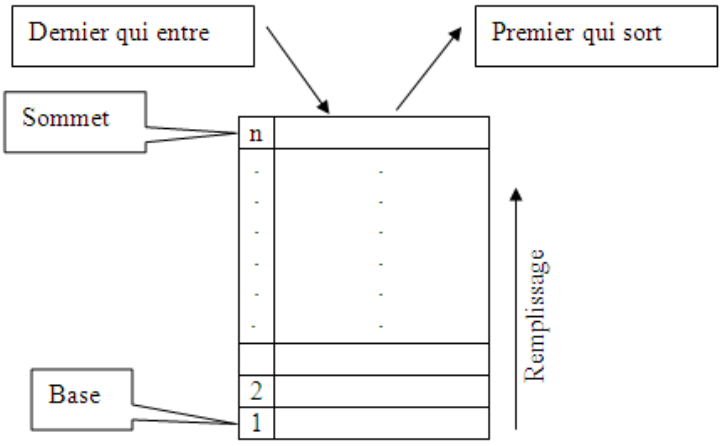
Modèle sur les listes linéaires chaînées bidirectionnelles

- **Allouer :** Création d'un maillon.
- **Libérer :** Libérer un maillon.
- **Valeur(P) :** retourne la valeur du champ val du maillon pointé par P .
- **Suivant(P) :** retourne le pointeur se trouvant dans le champs suivant du maillon pointé par P
- **Précédent(P) :** retourne le pointeur se trouvant dans le champs suivant du maillon pointé par P
- **Aff_Val(P,x) :** Ranger la valeur de x dans le champs val du maillon pointé par P
- **Aff_Adr_Précedent(P,Q) :** Ranger Q dans le champs précédent du maillon pointé par P
- **Aff_Adr_Suivant(P,Q) :** Ranger Q dans le champs suivant du maillon pointé par P

4.2 Les piles (stacks)

4.2.1 Définition

Une pile est une liste ordonnée d'éléments où les insertions et les suppressions d'éléments se font à une seule et même extrémité de la liste appelée *le sommet de la pile.*
Le principe d'ajout et de retrait dans la pile s'appelle LIFO (*Last In First Out*) : "le dernier qui entre est le premier qui sort"



4.2.2 Utilisation des piles

4.2.2.1 Dans un navigateur web

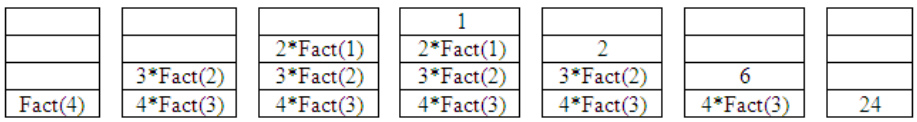
Une pile sert à mémoriser les pages Web visitées. L'adresse de chaque nouvelle page visitée est empilée et l'utilisateur dépile l'adresse de la page précédente en cliquant le bouton "Afficher la page précédente".

4.2.2.2 Annulation des opérations

La fonction "Annuler la frappe" d'un traitement de texte mémorise les modifications apportées au texte dans une pile.

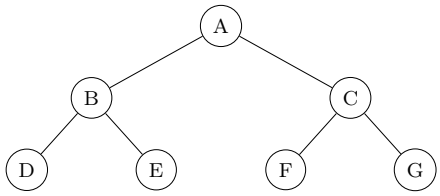
4.2.2.3 Gestion des appels dans l'exécution des programmes

Fact(4)=4*Fact(3)=4*3*Fact(2)=4*3*2*Fact(1)=4*3*2*1=4*3*2=4*6=24



4.2.2.4 Parcours en profondeur des arbres

Soit l'arbre suivant :



L'algorithme de parcours en profondeur est le suivant :

```
Mettre la Racine dans la pile ;
Tant que (La pile n'est pas vide) faire
    Retirer un noeud de la pile ;
    Afficher sa valeur ;
    Si (Le noeud a des fils) Alors
        Ajouter ces fils à la pile
    Fin Si;
Fin TQ;
```

Le résultat de parcours en profondeur affiche : A, B, D, E, C, F, G.

4.2.2.5 Evaluation des expressions postfixées

Pour l'évaluation des expressions arithmétiques ou logiques, les langages de programmation utilisent généralement les représentation préfixée et postfixée. Dans la représentation postfixée, on représente l'expression par une nouvelle, où les opérations viennent toujours après les opérandes.

Exemple

L'expression $((a + (b * c))/(c - d)$ est exprimée, en postfixé, comme suit : $bc * a + cd - /$
Pour l'évaluer, on utilise une pile. On parcourt l'expression de gauche à droite, en exécutant l'algorithme suivant :

$i \leftarrow 1$;

Tant que ($i < Longeur(Expression)$) **faire**

  Si ($Expression[i]$ est un Opérateur) **Alors**

    Retirer deux éléments de la pile ;

    Calculer le résultat selon l'opérateur ;

    Mettre le résultat dans la pile ;

  Sinon

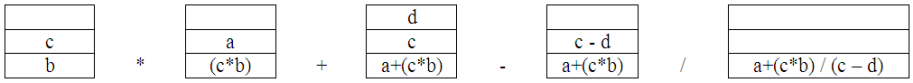
    Mettre l'opérande dans la pile ;

  Fin Si;

  $i \leftarrow i + 1$;

Fin TQ;

Le schéma suivant montre l'évolution du contenu de la pile, en exécutant cet algorithme sur l'expression précédente.



4.2.3 Opérations sur les piles

Les opérations habituelles sur les piles sont :

- Initialisation de la pile (généralement à vide)
- Vérification du contenu de la pile (pile pleine ou vide)
- Dépilement (POP) : retirer un élément du sommet de la pile si elle n'est pas vide
- Empilement (PUSH) : ajout d'un élément au sommet de la pile si elle n'est pas saturée.

Exercice : Donner l'état de la pile après l'exécution des opérations suivantes sur une pile vide :

Empiler(a), Empiler(b), Dépiler, Empiler(c), Empiler(d), Dépiler, Empiler(e), Dépiler, Dépiler.

4.2.4 Implémentation des piles

Les piles peuvent être représentés en deux manières : par des tableaux ou par des LLCs :

4.2.4.1 Implémentation par des tableaux

L'implémentation statique des piles utilise les tableaux. Dans ce cas, la capacité de la pile est limitée par la taille du tableau. L'ajout à la pile se fait dans le sens croissant des indices, tandis que le retrait se fait dans le ses inverse.

4.2.4.2 Implémentation par des LLCs

L'implémentation dynamique utilise les listes linéaires chinées. Dans ce cas, la pile peut être vide, mais ne peut être jamais pleine, sauf bien sur en cas d'insuffisance de l'espace mémoire. L'empilement et le dépilement dans les piles dynamiques se font à la tête de la liste.

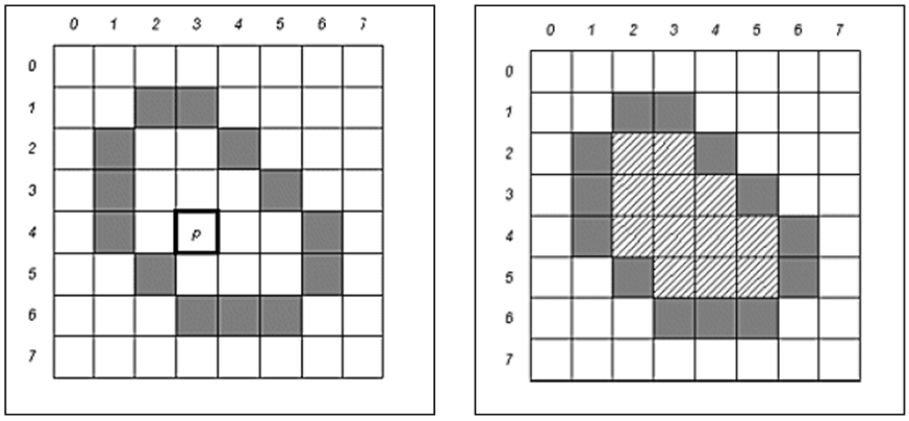
Les deux algorithmes "PileParTableaux" et "PileParLLCs" suivant présentent deux exemples d'implémentation statique et dynamique des piles.

```
Algorithme PileParTableaux;
Var  Pile : Tableau[1..n] de entier ;      Sommet : Entier ;
Procédure InitPile();
Début
    Sommet ← 0 ;
Fin;
Fonction PileVide() : Booleen;
Début
    PileVide ← (Sommet = 0) ;
Fin;
Fonction PilePleine() : Booleen;
Début
    PilePleine ← (Sommet = n) ;
Fin;
Procédure Empiler( x : entier);
Début
    Si (PilePleine) Alors
        Ecrire('Impossible d'empiler, la pile est pleine ! !')
    Sinon
        Sommet ← Sommet + 1 ;
        Pile[Sommet] ← x ;
    Fin Si;
Fin;
Procédure Dépiler( x : entier);
Début
    Si (PileVide) Alors
        Ecrire('Impossible de dépiler, la pile est vide ! !')
    Sinon
        x ← Pile[Sommet] ;
        Sommet ← Sommet − 1 ;
    Fin Si;
Fin;

Début
    ... Utilisation de la pile ...
Fin.
```

```
Algorithme PileParLLCs;
Type  TMaillon = Structure
    Valeur : entier ;
    Suivant : Pointeur(TMaillon) ;
Fin ;
Var  P, Sommet : Pointeur(TMaillon) ;
Procédure InitPile();
Début
    | Sommet ← Nil ;
Fin;
Fonction PileVide() : Booleen;
Début
    | PileVide ← (Sommet = Nil) ;
Fin;
Procédure Empiler( x : entier);
Début
    | Allouer(P) ;          Aff_Val(P,x) ;
    | Aff_Adr(P,Sommet) ;   Sommet ← P ;
Fin;
Procédure Dépiler( x : entier);
Début
    | Si (PileVide) Alors
    |     | Ecrire('Impossible de dépiler, la pile est vide !!')
    | Sinon
    |     | x ← Valeur(Sommet) ;          P ← Sommet ;
    |     | Sommet ← Suivant(Sommet) ;    Libérer(P) ;
    | Fin Si;
Fin;

Début
| ... Utilisation de la pile ...
Fin.
```

4.2.5 Exemple d'application : Remplissage d'une zone d'une image

Une image en informatique peut être représentée par une matrice de points $'Image'$ ayant M colonnes et N lignes. Un élément $Image[x, y]$ de la matrice représente la couleur du point p de coordonnées (x, y). On propose d'écrire ici une fonction qui, à partir d'un point p, *étale* une couleur c autour de ce point. La progression de la couleur étalée s'arrête quand elle rencontre une couleur autre que celle du point p. La figure suivante illustre cet exemple, en considérant $p = (3, 4)$.



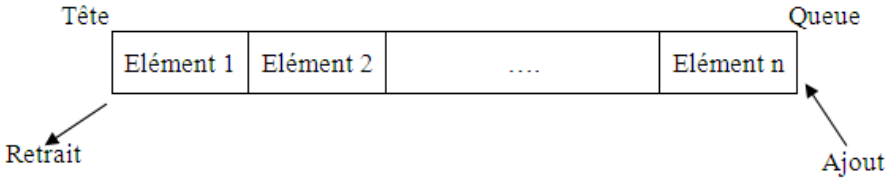
Pour effectuer le remplissage, on doit aller dans toutes les directions à partir du point p. Ceci ressemble au parcours d'un arbre avec les noeuds de quatre fils. La procédure suivante permet de résoudre le problème en utilisant une pile.

```
Procédure Remplir( Image : Tableaux[0..M−1, 0..N−1] de couleur ; x, y : entier ;
c : couleur);
Var c1 : couleur ;
Début
    c1 ← Image[x, y] ;
    InitPile ;
    Empiler((x, y)) ;
    Tant que (¬ PileVide) faire
        Depiler((x,y)) ;
        Si (Image[x, y] = c1) Alors
            Image[x, y] ← c ;
            Si (x > 0) Alors
                Empiler((x − 1, y))
            Fin Si;
            Si (x < M − 1) Alors
                Empiler((x + 1, y))
            Fin Si;
            Si (y > 0) Alors
                Empiler((x, y − 1))
            Fin Si;
            Si (y < N − 1) Alors
                Empiler((x, y + 1))
            Fin Si;
        Fin Si;
    Fin TQ;
Fin;
```

4.3 Les Files d'attente (Queues)

4.3.1 Définition

La file d'attente est une structure qui permet de stocker des objets dans un ordre donné et de les retirer dans le même ordre, c'est à dire selon le protocole FIFO *'first in first out'*. On ajoute toujours un élément en queue de liste et on retire celui qui est en tête.



4.3.2 Utilisation des files d'attente

Les files d'attente sont utilisées, en programmation, pour gérer des objets qui sont en attente d'un traitement ultérieur, tel que la gestion des documents à imprimer, des programmes à exécuter, des messages reçus,...etc. Elles sont utilisées également dans le parcours des arbres.

Exercice

Reprendre le parcours de l'arbre de la section 4.2.2.4 (parcours en profondeur) en utilisant une file d'attente au lieu de la pile.

4.3.3 Opérations sur les files d'attente

Les opérations habituelles sur les files sont :

- Initialisation de la file
- Vérification du contenu de la file (vide ou pleine)
- Enfilement : ajout d'un élément à la queue de la file
- Défilement : retrait d'un élément de la tête de la file

4.3.4 Implémentation des files d'attente

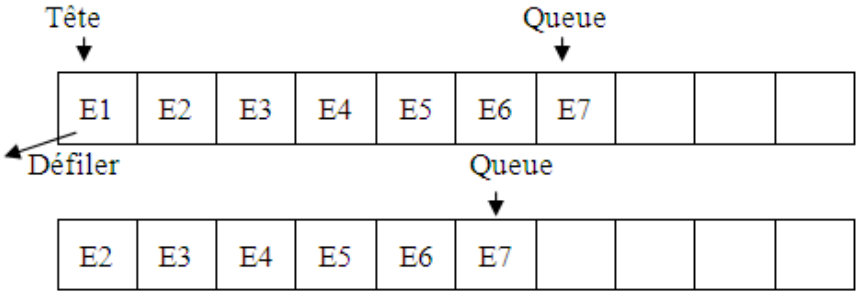
De même que pour les piles, les files d'attente peuvent être représentées en deux manières :

- par représentation statique en utilisant les tableaux,
- par représentation dynamique en utilisant les listes linéaires chaînées.

4.3.4.1 Implémentation statique

L'implémentation statique peut être réalisée par décalage en utilisant un tableau avec une tête fixe, toujours à 1, et une queue variable. Elle peut être aussi réalisée par flot utilisant un tableau circulaire où la tête et la queue sont toutes les deux variables.

1. Par décalage

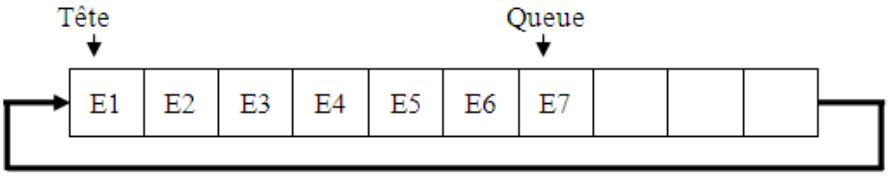


→ La file est vide si $Queue = 0$

→ La file est pleine si $Queue = n$

↓ Problème de décalage à chaque défilement

2. Par flot : La file est représentée par un tableau circulaire



→ La file est vide si $Tête = Queue$

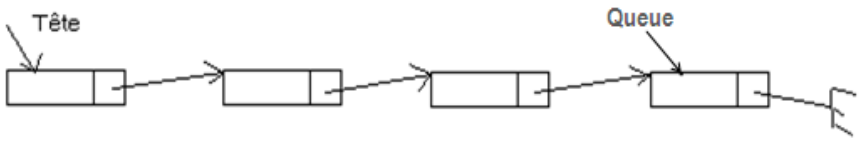
→ La file est pleine si $(Queue + 1)\ mod\ n = Tête$

↓ On sacrifie une case pour distinguer le cas d'une file vide de celui d'une file pleine.

Exercice Pensez à une solution qui évite le sacrifice d'une case.

4.3.4.2 Implémentation dynamique

La représentation dynamique utilise une liste linéaire chaînée. L'enfilement se fait à la tête de la liste et de défilement se fait de la queue. La file d'attente, dans ce cas, peut devenir vide, mais ne sera jamais pleine.



Les deux algorithmes FileParFlot et FileParLLCs, à la fin de cette section, présentent des exemples d'implémentation, respectivement, statique et dynamique.

4.3.5 File d'attente particulière (File d'attente avec priorité)

Une file d'attente avec priorité est une collection d'éléments dans laquelle l'insertion ne se fait pas toujours à la queue. Tout nouvel élément est inséré, dans la file, selon sa priorité. Le retrait se fait toujours du début.
Dans une file avec priorité, un élément prioritaire prendra la tête de la file même s'il arrive le dernier. Un élément est toujours accompagné d'une information indiquant sa priorité dans la file.
L'implémentation de ces files d'attente peut être par tableau ou listes, mais l'implémentation la plus efficace et la plus utilisée utilise des arbres particuliers qui s'appellent 'les tas'.

```
Algorithme FileParFlot;
Var  File : Tableau[1..n] de entier ;      Tête, Queue : Entier ;
Procédure InitFile();
Début
    Tête ← 1 ; Queue ← 1 ;
Fin;
Fonction FileVide() : Booleen;
Début
    FileVide ← (Tête = Queue) ;
Fin;
Fonction FilePleine() : Booleen;
Début
    FilePleine ← (((Queue + 1) mod n) = Tête) ;
Fin;
Procédure Enfiler( x : entier);
Début
    Si (FilePleine) Alors
        Ecrire('Impossible d'enfiler, la file est pleine ! !')
    Sinon
        File[Queue] ← x ;
        Queue ← (Queue + 1) mod n ;
    Fin Si;
Fin;
Procédure Défiler( x : entier);
Début
    Si (FileVide) Alors
        Ecrire('Impossible de défiler, la file est vide ! !')
    Sinon
        x ← File[Tete] ;
        Tête ← (Tête + 1) mod n ;
    Fin Si;
Fin;

Début
    ... Utilisation de la File ...
Fin.
```

```
Algorithme FileParLLCs;
Type  TMaillon = Structure
    Valeur : entier ;
    Suivant : Pointeur(TMaillon) ;
Fin ;
Var  P, Tête, Queue : Pointeur(TMaillon) ;
Procédure InitFile();
Début
    | Tête ← Nil ;        Queue ← Nil ;
Fin;
Fonction FileVide() : Booleen;
Début
    | FileVide ← (Tête = Nil) ;
Fin;
Procédure Enfiler( x : entier);
Début
    | Allouer(P) ;          Aff_Val(P,x) ;
    | Aff_adr(P,Nil) ;
    | Si (Queue = Nil) Alors
    |     | Tête ← P ;
    | Sinon
    |     | Aff_Adr(Queue,P) ;
    | Fin Si;
    | Queue ← P ;
Fin;
Procédure Défiler( x : entier);
Début
    | Si (FileVide) Alors
    |     | Ecrire('Impossible de défiler, la file est vide !!')
    | Sinon
    |     | x ← Valeur(Tete) ;        P ← Tête ;
    |     | Tête ← Suivant(Tête) ;        Libérer(P) ;
    | Fin Si;
Fin;

Début
| ... Utilisation de la File ...
Fin.
```