

# Chapitre 3

## Structures de données linéaires :

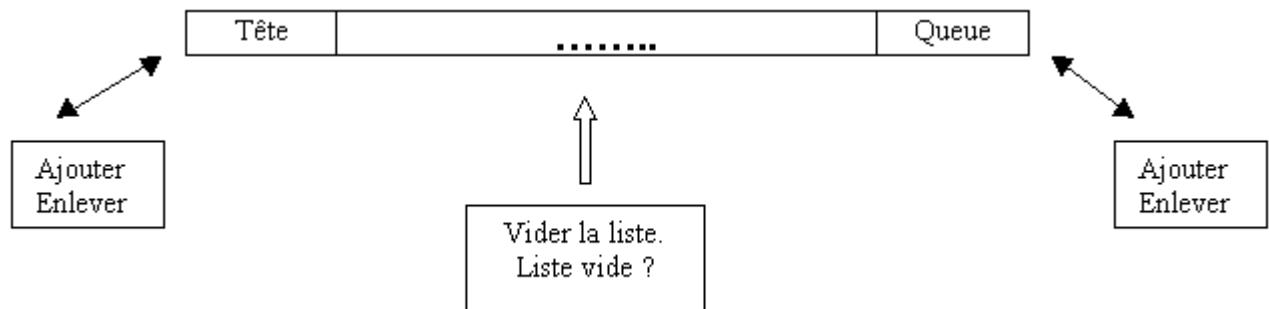
### listes, piles et files

#### 1. Introduction

Le but de ce chapitre est de décrire des représentations des structures de données de base telles les listes en général et deux formes restreintes: les piles et les files. L'autre but recherché est de voir l'importance de ces structures à travers quelques exemples d'applications

#### 2. Les listes

Les listes sont des **structures de données 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.



#### 2.1. Quelques opérations sur les listes

1. tester si la liste est vide (ou la mettre à vide)
2. accéder au *k*ème élément de la liste (pour le modifier, supprimer, etc ...)
3. insérer un nouvel élément derrière le *k*ème
4. fusionner 2 listes
5. rechercher un élément d'une valeur particulière
- 6 . trier
- 7) ... etc.

**But:** en fonction des opérations choisies, choisir une représentation interne pour lesquelles les opérations sont efficaces.

## **2.2 Implantation des listes.**

Il existe plusieurs méthodes pour implémenter des listes. Les plus courantes sont l'utilisation de tableaux et de pointeurs.

**A. Utilisation de tableaux :** implémenter une liste à l'aide d'un tableau n'est pas très compliqué. Les éléments de la liste sont simplement rangés dans le tableau à leur place respective. Cependant, l'utilisation de tableaux possède quelques inconvénients :

- La dimension d'un tableau doit être définie lors des déclarations et ne peut donc pas être modifiée « dynamiquement » lors de l'exécution d'un programme. La solution consiste donc à **définir un tableau dont la taille sera suffisante** pour accueillir la plus grande liste pouvant être utilisée, et d'associer au tableau une variable indiquant le nombre d'éléments contenus dans le tableau.
- Le **tableau** étant **surdimensionné**, il **encombre** en général la mémoire de l'ordinateur.
- Si la taille maximum venait à être augmentée, il faudrait modifier le programme et recompiler.
- Lorsque l'on **retire un élément** du tableau, en particulier en début de liste, il est nécessaire de **décaler tous les éléments** situés après l'élément retiré.

**B. Utilisation de pointeurs :** les pointeurs définissent une adresse dans la mémoire de l'ordinateur, adresse qui correspond à l'emplacement d'une autre variable. Il est possible à tout moment d'allouer cet espace dynamiquement lors de l'exécution du programme

### **2.2.1 Représentation contiguë d'une liste par tableau**

Avantage: l'accès au *k*ème élément est immédiat:  $t[k]$

**Inconvénient:**

- l'insertion d'un élément est très coûteux car il faut décaler d'un cran tous les éléments suivants (jusqu'à  $n+1$  affectations)
- la suppression d'un élément, pour les mêmes raisons, peut entraîner jusqu'à  $n-1$  affectations.

### **2.2.2. Représentation chaînée d'une liste à l'aide d'un tableau**

**Principe:** représenter chaque élément de la liste à un endroit quelconque de la mémoire

**Exemple:** la liste  $L = \langle a_1, a_2, a_3, a_4 \rangle$

	valeur	pointeur
1	a4	0 (indice fictif)
2	a2	5
3		
4	a1	2
5	a3	1
6		
7		

**Inconvénient :**

**Avantage:** si par exemple on insère a'3 après a3 dans la liste, on ne réalise que 3 affectations. Idem pour l'insertion et la suppression, coût:  $O(1)$  si on connaît l'endroit pour insérer. Une liste se définit de façon récursive

**Exemple:**

```
struct T_cellule{
    int val;
    T_cellule * suiv;
}
```

**Définition:** Une liste d'éléments de type T est :

soit vide // ici le pointeur = 0

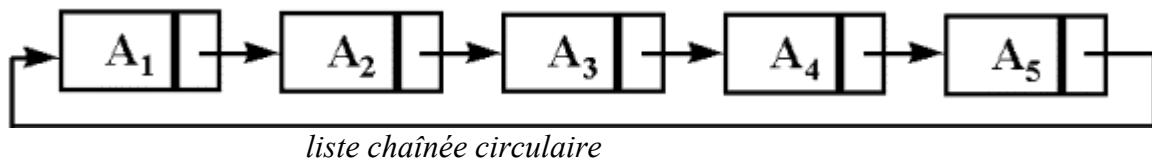
soit la donnée de :

\* un élément de type T (le premier élément de la liste)

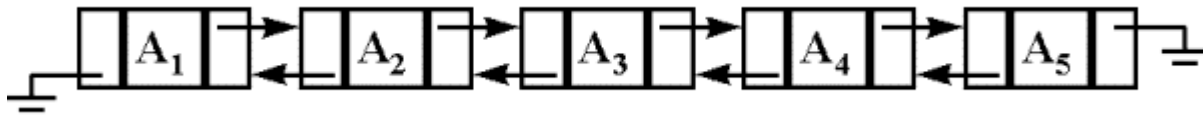
\* une liste (d'élément de type T): la liste privée de son premier élément.

**Variantes utiles d'une liste**

--> les listes circulaires: on a une liste  $L = \langle a_0, a_1, a_2, \dots, a_{n-1} \rangle$  dont le suivant du dernier  $a_{n-1}$  est le premier ( $a_0$ )



→ les listes doublement chaînées: utile quand on veut accéder facilement au prédécesseur d'un élément de la liste.



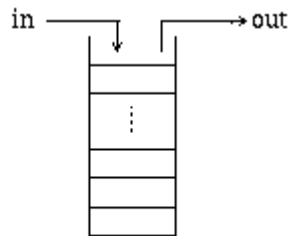
*liste doublement chaînée*

### 3. Les piles

Une pile est une liste sur laquelle on autorise seulement 4 opérations:

1. consulter le dernier élément de la pile (le sommet de la pile)
2. tester si la pile es vide
3. empiler un élément, le mettre au sommet de la pile ==> PUSH
4. dépiler un élément (par le sommet) ==> POP

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



*Fig. 7.5. Structure de pile.*

#### 3.1. Représentation d'une pile par tableau

**Avantage:** Facile car on ne modifie une pile que par un bout. Les opérations sont faciles.

**Inconvénient:** la hauteur est bornée (allocation statique de la mémoire)

#### Représentation chaînée

**Avantage:** facile avec la tête de liste chaînée sur le haut de la pile (en particulier  $p = 0$  si la pile est vide)

**Inconvénient:** espace occupé par les pointeurs

### 3.2. Applications des piles

**1. Appels de fonctions:** Quand une fonction est appelée, les paramètres (incluant l'adresse de retour) doivent être passés à la fonction appelante. Si ces paramètres sont sauvegardés dans une mémoire fixe, alors la fonction ne peut pas être appelée récursivement du moment que la première adresse va être écrasée par le deuxième retour d'adresse avant que la première ne soit utilisée. Les piles sont souvent nécessaires pour rendre itératif un algorithme récursif.

**Exemple :** Une version du tri rapide récursif en tri rapide itératif est comme suit. La pile est ainsi mise à contribution pour sauvegarder les extrémités de la partie du tableau pour laquelle on veut trouver le pivot.

```
void tri_rapide_bis(int tableau[],int debut,int fin){
    int pivot,i,j;
    empiler(debut);
    empiler(fin);
    while (!empty_stack){
        pivot=partition(tableau,debut,fin);
        i =depiler;
        j= depiler;
        pivot=partition(tableau,i,j); /* partition à gauche */
        if (pivot > i){
            empiler(i);
            empiler(pivot-1);
        };
        else if (j > pivot){
            empiler(pivot+1); /* partition à droite */
            empiler(j)}}}
```

**Exercice 1:** rendre itératif le tri par fusion.

**Exercice 2:** discuter les avantages et inconvénients l'utilisation explicite d'une pile pour simuler la récursivité.

**2.2. Analyse syntaxique :** qui est une phase de la compilation qui nécessite une pile. Par exemple, le problème de la reconnaissance des mots bien parenthésés Nous devons écrire un programme qui :

- accepte les mots comme (), ()() ou (((()))());
- rejette les mots comme )(, ()() ou (((())))).

Nous stockons les parenthèses ouvrantes non encore refermées dans une **pile** de caractères,

Le programme lit caractère par caractère le mot entré :

- si c'est une ouvrante, elle est empilée ;
- si c'est une fermante, l'ouvrante correspondante est dépilée.

Le mot est accepté si

- la pile n'est jamais vide à la lecture d'une fermante ; et si
- la pile est vide lorsque le mot a été lu.

**2.3. 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 (-, !,...). 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'écrit en notation postfixée (notation polaise):  $3\ 5\ +\ 2\ *$

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

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:  $A/B ** C + D * E - A * C$

Forme postfixée:  $ABC ** /DE * + AC * -$

#### Evaluation en Postfix

Considérons l'expression en postfixée 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 postfixée;
    if(item est une valeur)
        empiler;
    else if(item opérateur binaire) {
        dépiler dans x;
        dépiler dans y;
        effectuer y opérateur x;
```

```

    empiler le résultat obtenu;
  } else if (item opérateur unaire) {
    dépiler dans x;
    effectuer opérateur(x);
    empiler le résultat obtenu;
  }
}

```

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

Opérateur binaires: +, -, \*, /, etc.,

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

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

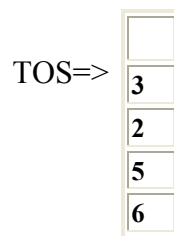
Le premier item est une valeur (6); elle est empilée.

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

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

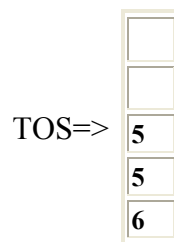
Le prochain item est une valeur (3); elle est empilée.

La pile devient

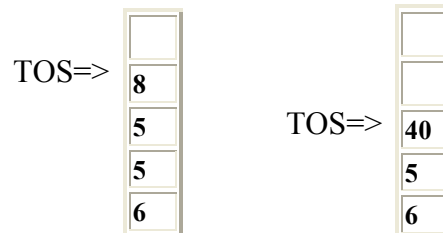


Les items restants à cette étape sont: **+ 8 \* + 3 + \***

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

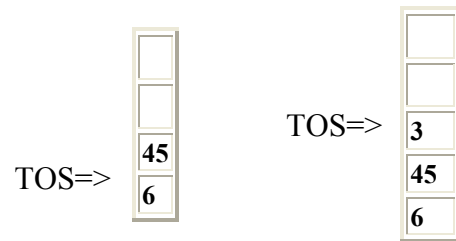


Ensuite 8 est empilé et le prochain opérateur \*:



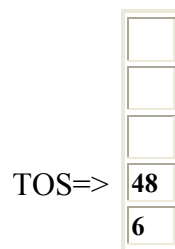
(8, 5 sont dépilés, 40 est empilé)

Ensuite l'opérateur + suivi de 3:

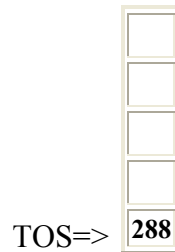


(40, 5 sont dépilés ; 45 pushed, 3 est empilé

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



Ensuite c'est l'opérateur \*: 48 et 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 trouvé 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.

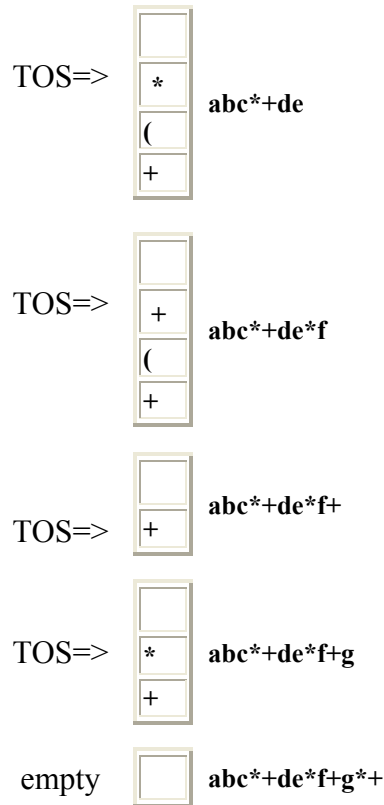
**Exemple:** considérons la forme infixe de l'expression **a+b\*c+(d\*e+f)\*g**

**Stack                      Output**

TOS=>  **ab**

TOS=>  **abc**

TOS=>  **abc\*+**



### Un autre exemple

Comme nous voulons que  $A + B * C$  génère  $ABC * +$ , notre algorithme effectuera une succession d'empilement - dépilement suivants (la pile est à la droite):

Item	Pile	Postfixe
-----	-----	-----
aucun	vide	aucun
A	vide	A
+	+	A
B	+	AB

À ce stade, l'algorithme détermine que  $*$  et  $+$  doit être placé à la fin de la pile. Comme l'opération  $*$  a une plus grande précedence, on empile  $*$

*	+ *	AB
C	+ *	ABC

Et on dépile

$ABC*+$

Comme les données en entrée sont terminées, on dépile tout le reste de la pile pour donner l'expression postfixe finale suivante:

ABC \* +

#### 4. Les files (d'attente)



##### 4.1. Définition et exemples

Une file est une structure de données dynamique dans laquelle on insère des nouveaux éléments à la fin (queue) et où on enlève des éléments au début (tête de file).

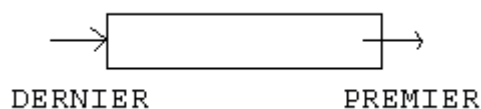
L'application la plus classique est la file d'attente, et elle sert beaucoup en simulation. Elle est aussi très utilisée aussi bien dans la vie courante que dans les systèmes informatiques. Par exemple, elle modélise la file d'attente des clients devant un guichet, les travaux en attente d'exécution dans un système de traitement par lots, ou encore les messages en attente dans un commutateur de réseau téléphonique. On retrouve également les files d'attente dans les programmes de traitement de transactions telle que les réservations de sièges d'avion ou de billets de théâtre.

Noter que dans une file, le premier élément 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.

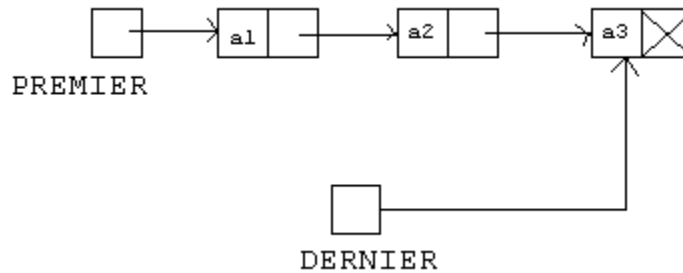


## 4.2. Quatre opérations de base

1. consulter le premier élément de la file
2. tester si la file est vide
3. enfiler un nouvel élément: le mettre en dernier
4. défiler un élément, le premier (le supprimer)

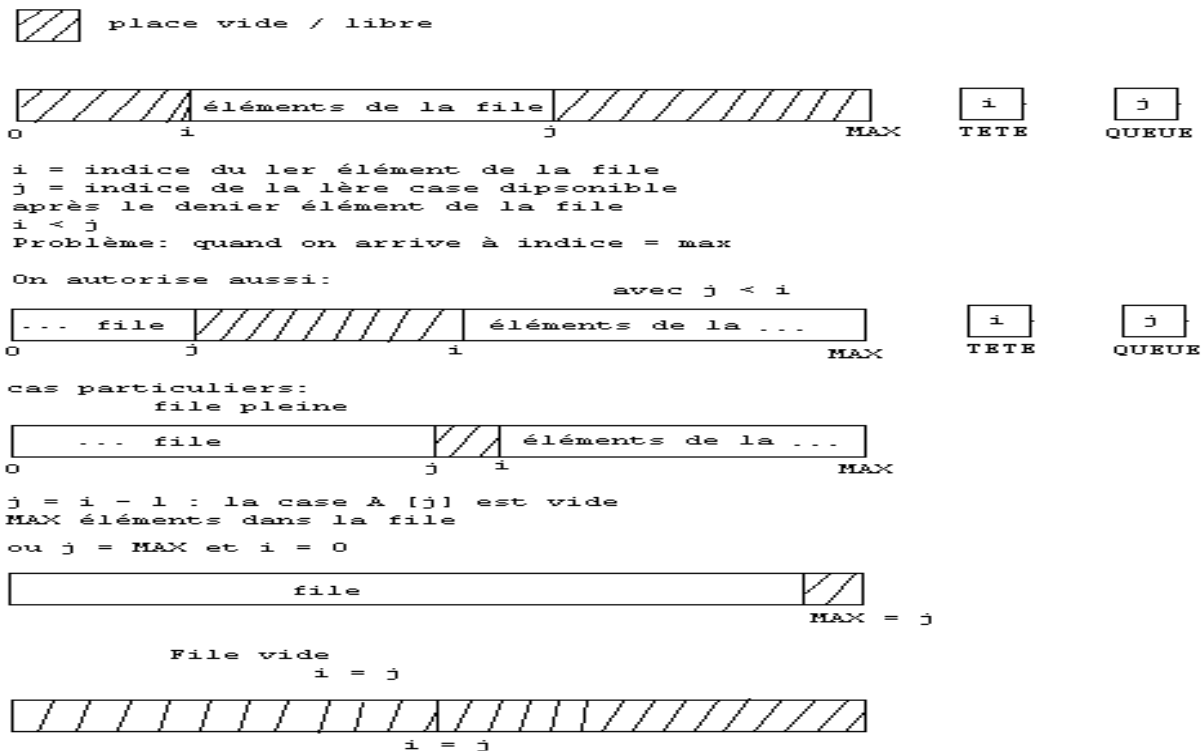
### 4.3. Représentation d'une file par liste chaînée

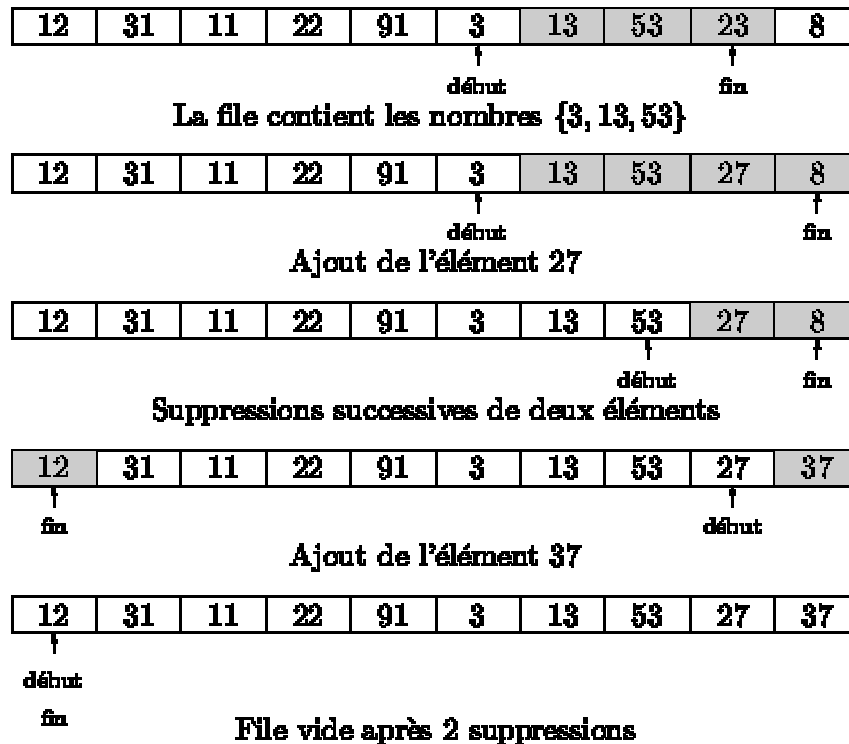
Il est facile d'implanter les quatre opérations ci-dessus, mais on perd la place due aux pointeurs.



#### 4.4. Représentation d'une file par tableau

vision circulaire du tableau





**4.5 File de priorité:** Les éléments de la file ont chacun une priorité. Il faut accéder à l'élément prioritaire et de le retirer (voir dans le chapitre sur les arbres).

#### 4.6 Quelques applications

##### 1. Impression de programmes

maintenir une file de programmes en attente d'impression

##### 2. Disque driver

maintenir une file de requêtes disque d'input/output disque

##### 3. Ordonnanceur (dans les systèmes d'exploitation)

maintenir une file de processus en attente d'un temps machine

4. Considérons un fichier d'enregistrements. Chaque enregistrement est constitué d'informations concernant le nom, sexe, date de naissance d'une personne. Les enregistrements sont triés par dates de naissance. Le problème qu'on se propose de traiter est de réarranger les enregistrements de telle manière que les enregistrements de personnes de sexe féminin précèdent ceux de sexe masculin.

**Solution 1 :** utiliser un algorithme de tri  $\rightarrow$  complexité  $O(n \log n)$ .

**Solution 2 :** on peut faire mieux en utilisant une file  $\rightarrow$  complexité  $O(n)$ .

- a. créer une file pour les femmes et une autre file pour les hommes
- b. répéter jusqu'à ce que le fichier soit vide
  - b.1. soit  $p$  la prochaine personne à traiter dans le fichier
  - b.2. si  $p$  est une femme, l'ajouter à la fin de la file des femmes
  - b.3. si  $p$  est un homme, l'ajouter à la fin de la file des hommes
- c. répéter jusqu'à ce que la file des femmes soit vide  
prendre l'élément de la tête de file des femmes et le recopier
- d. répéter jusqu'à ce que la file des hommes soit vide  
prendre l'élément de la tête de file des hommes et le recopier

#### Références et sources :

- 1. Cormen et al. (1990) : Algorithms, MacGraw Hill.
- 2. G. Bebis (2003): Notes de cours, Université du Nevada, USA.
- 3. S. Sahni (1999) : Notes de cours sur data structures and algorithms, University of Florida.
- 4. R. Cori, J.J. Lévy (2000) : Notes de cours sur les algorithmes et programmation, École Polytechnique.
- 5. D.A. Watt, D.F. Brown (2001): Notes de cours sur Queue ADT, Université de Montréal.