Types de données abstraits

Module INFO 0401

2ème année informatique

Reims – Sciences Exactes

Cours 4

A.HEBBACHE

sommaire

1 Type de données abstrait

- Introduction
- Définition
- But

2 Les structures de données

- Liste linaire chainés
- Les files
- Les piles

Les structures de données abstraites

❖ Définition

Un type de données abstrait (TDA) est composé d'un ensemble d'objets, similaires dans la forme et dans le comportement, et d'un ensemble d'opérations sur ces objets.

L'implémentation d'un TDA ne suis pas de schéma préétabli.

Il dépend des objets manipulés et des opérations disponibles pour leur manipulation.

Introduction

❖ But

- Écrire des algorithmes indépendamment de la représentation mémoire.

  ⟹ Donc définition d'une machine abstraite pour chaque structure de données (Modèle) avec un ensemble d'opérations.

- Créer des algorithmes qui combinent les différents types de données définies.
- De pouvoir faire des opérations sur ces données et faire des spécifications sur les opérateurs.

Introduction

❖ But (suite)

- Elle met au même niveau les données et les opérations.
- Elle permet de modéliser les applications informatiques autour des collections imbriquées de TAD.
- Elle permet l'élaboration des algorithmes d'une manière abstraite:
  1) en faisant appel aux données et aux opérations abstraites du TAD ;
  2) suivi d'un choix de représentation du TAD de mémoire.

Introduction

❖Le type de données

- Représentation mémoire (⇒ taille mémoire)
- Vision du programmeur : on parle de type **concret** **(du monde réel : numérique, avec décimale ou non, chaine, …)**
- Fonctions de manipulation (⇒ complexité)
- Vision algorithmique : on parle de type **abstrait**

❖Type concret

Un type concret n'est pas juste une représentation mémoire.

En effet, c'est aussi un ensemble de fonctions de manipulation**.**

Introduction

❖ **Exemple de type concret**

Short, int

C'est un entier sur lequel on effectue des opérations arithmétiques.

char c'est un caractère sur lequel on exécute des opérations d'affichage.

Type booléen ensemble de deux valeurs (faux, vrai) muni des opérations : NON, ET, OU

Introduction

❖ **Type abstrait**

- Dans un algorithme qui manipule des entiers, on s'intéresse, non pas à la représentation des entiers, mais aux opérations définies sur les entiers :

  +, -, *, /

- Les chaînes de caractères, munies des opérations:

  Insertion, Concaténation, …

- Type booléen, ensemble de deux valeurs (faux, vrai) muni des opérations :

  NON, ET, OU

- Les types de données et constructeurs de type déjà rencontrer sont en fait des TAD.

Les structures de données abstraites

❖ Contraintes d'implémentation

L'implémentation d'un type de données abstrait doit respecter deux contraintes :

1. Utiliser un minimum d'espace mémoire;
2. Exécuter un nombre minimal d'instructions pour réaliser une opération.

Les structures de données abstraites

❖ Implémentation

Pour implémenter un TAD on s'intéresse aux :

❖ Spécification fonctionnelle

Description des opérations possibles sur la structure avec leurs propriétés et leurs restrictions.

❖ Description logique

Construction de l'organisation de la structure pour que les primitives définies soient réalisables et efficaces. Construction des algorithmes des primitives avec contrôle de leur conformité aux spécifications.

❖ Représentation physique

Implantation de la structure et des opérations dans un langage de programmation.

Les structures de données abstraites

❖ Implémentation

- **Pour implémenter un type de données abstrait, on utilise :**

  - Les types élémentaires (entiers, caractères, ...)
  - Les pointeurs ;
  - Les tableaux et les enregistrements ;
  - Les types prédéfinis.

Les structures de données abstraites

❖ Implémentation

Pour implémenter un type de données abstrait, on s'intersse aux :

Opérations à exécuter sur le type de données, et les propriétés intrinsèques des opérations.

- On définit un algorithme de mise au point pour l'utilisateur du type de donnée.
- Connaître à priori les propriétés sans savoir comment il est implémenté, vérifier la validité de son utilisation.

Les structures de données abstraites

❖Exemple

type booléen

opérations

vrai : booléen

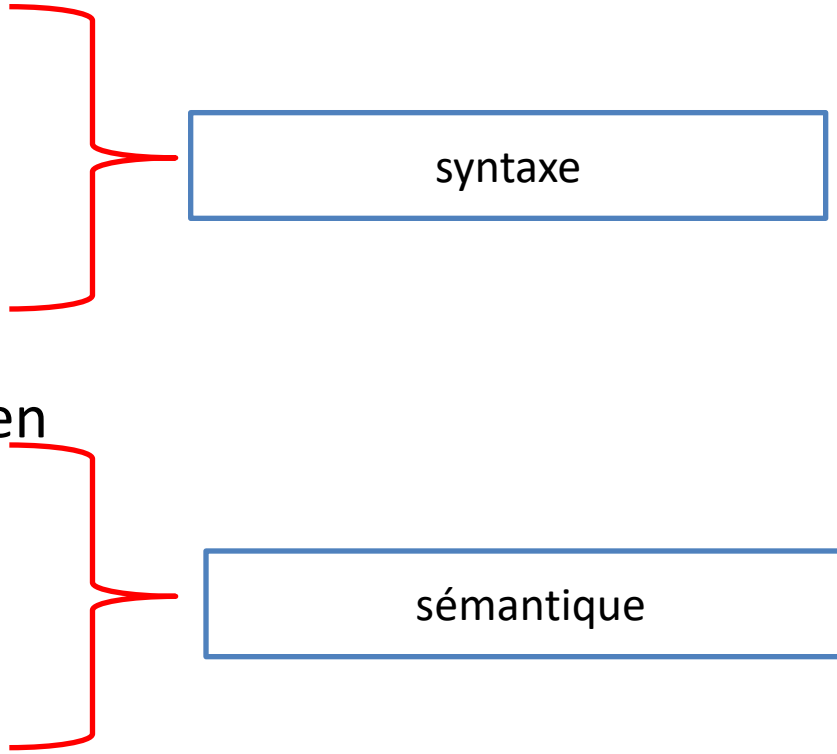
faux : booléen

sémantique

pour tout a, b : booléen

non(vrai) = faux

non(non(a)) = a



Les structures de données abstraites

❖Structure de données :

La structure de données est une implémentation explicite d'un type abstrait.

Les structures les plus simples sont la pile et la file d'attente.

Une structure est un arrangement linéaire d'éléments liés par la relation successeur.

Pourquoi avoir recours à cette notion nouvelle de type abstrait ?

Par ce qu'elle permet de définir des types de données non "primitifs", c'est-à-dire non disponibles (non déjà implémentés) dans les langages de programmation courants.

Les structure de données abstraits

❖Exemple

type Date : notation jj/mm/aa

- **Date() // crée un objet Date non initialisé**
- **Date(int j, int m, int a).**

```
// consultation
  int getJour()
  int getMois()
  int getAnnee()
```

Les structures de données abstraites

❖Types de structures de données abstraites

- ✓ Les structures séquentielles :

  Les **listes,** et leurs cas particuliers que sont les **piles et les files**

- ✓ Les structures arborescentes :

  Les **Arbres (binaires ou généraux)**

- ✓ *Les structures* relationnelles :

  Les **graphes.**

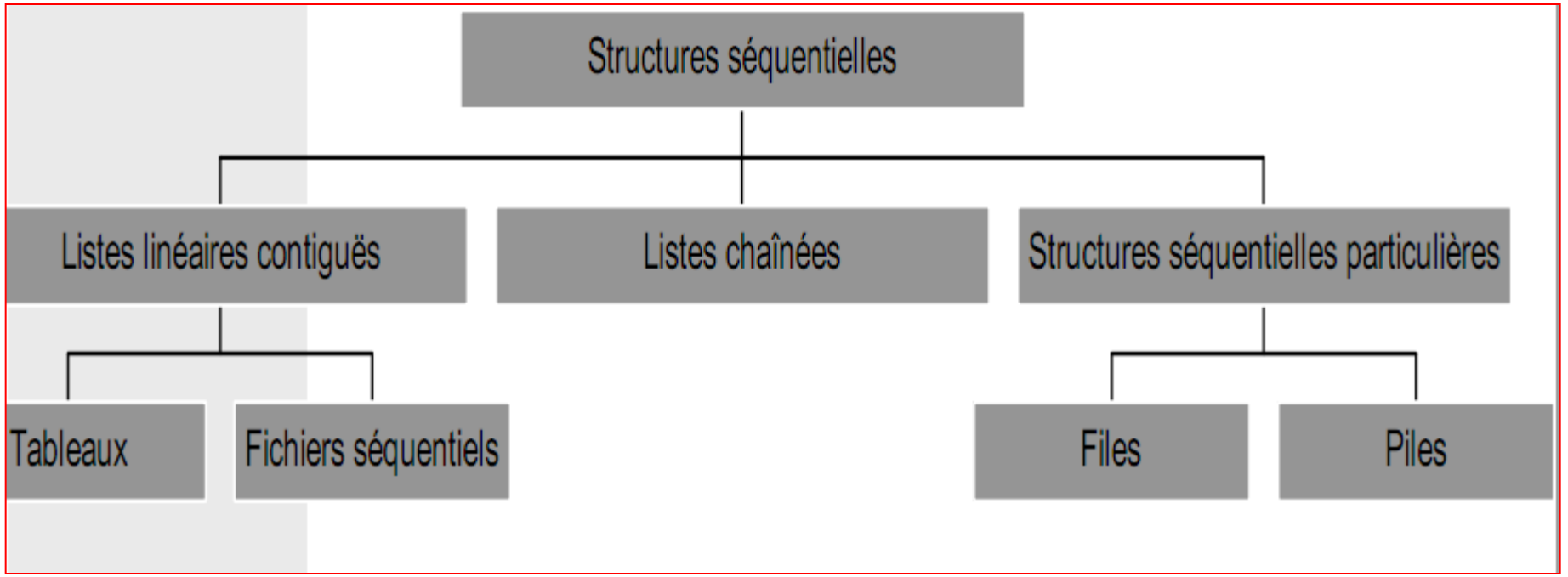
- ✓ *Structures à accès par clé* :

  Les ***tables***

Les structures de données séquentielles

❖Les types de structures séquentielles

Les différentes types de données séquentielles :



Les structures de données séquentielles

Étude des structures de données :

files, listes et piles

Pour chaque structure, on présente :

- ***une définition abstraite***
- ***les différentes représentations en mémoire***
- ***une implémentation en langage C***

Les structures de données séquentielles

❖Exemple d'introduction

- **Trouver tous les nombres premiers de 1 à n et les stocker en mémoire.**

  Si on utilise un tableau, il n'est pas possible de définir la taille de ce vecteur avec précision même si nous connaissons la valeur de n (par exemple 10000).

  Ne connaissant pas la valeur de n, on n'a aucune idée sur le choix de sa taille. On est donc, ici, en face d'un problème où la réservation de l'espace doit être dynamique.

Les structures de données séquentielles

❖Notion d'allocation dynamique et statique

- **Allocation statique**

L'allocation de l'espace se fait tout à fait au début d'un traitement. En termes techniques, on dit que l'espace est connu à la compilation. C'est donc la notion de **tableau**.

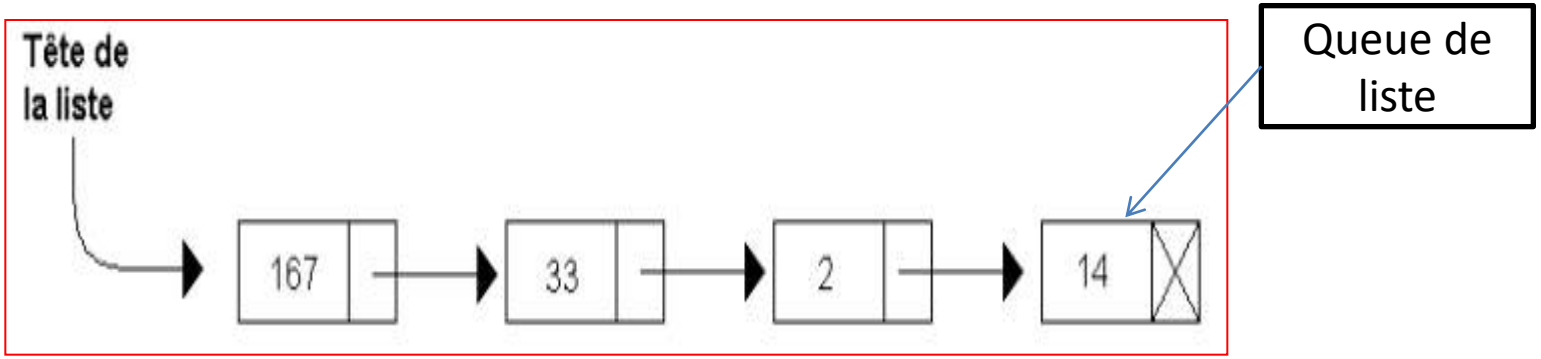
- **Allocation dynamique**

L'allocation de l'espace se fait au fur et à mesure de l'exécution du programme. Pour pouvoir faire ce type d'allocation, l'utilisateur doit disposer des deux opérations: **Allouer et Libérer**.

Les listes

❖Définition

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.



Les listes

❖Terminologies

- Les éléments d'une liste sont donc ***ordonnés*** en fonction de leurs places.

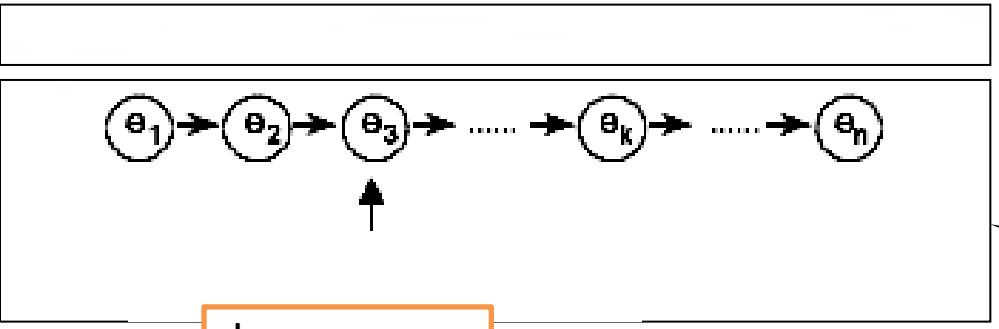
- Il existe *une fonction* notée ***suiv*** qui, appliquée à toute place sauf la dernière, fournit la place suivante.

- *Le nombre total d'éléments*, et par conséquent de places, est appelé ***longueur L de la liste***.

- ***Une liste vide*** est d'une longueur égale 0.

Les listes

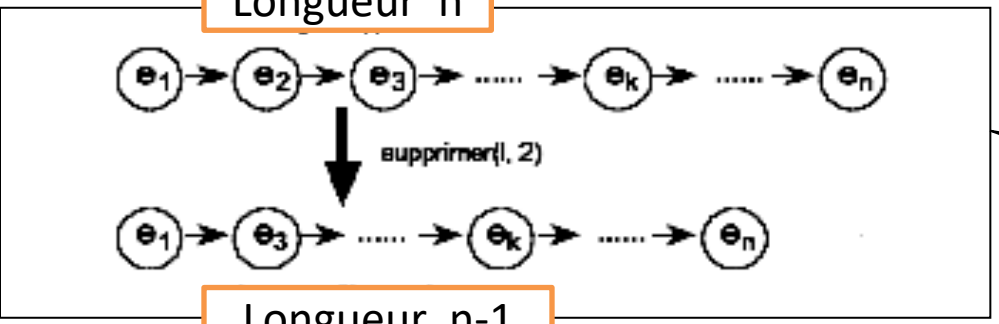
❖Exemple



Liste vide

Longueur n

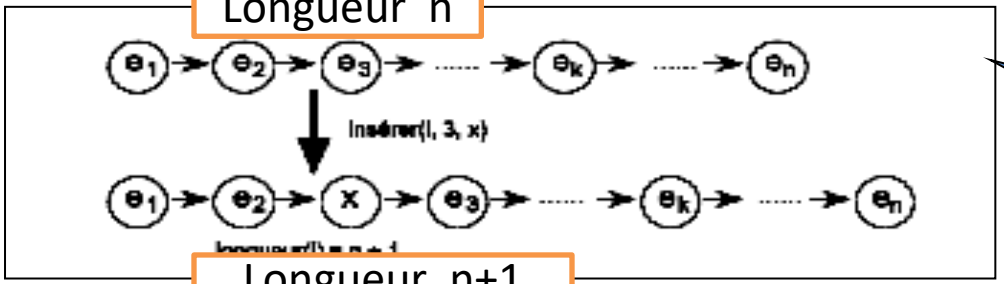
Accès à l'élément de rang 3
dans une liste à n éléments



Longueur n-1

Suppression de l'élément au rang 2
→ longueur(liste) =n-1

Longueur n



Longueur n+1

Ajout de l'élément x au rang 2
→ longueur(liste) =n+1

Les listes

❖Quelques opérations sur les listes

- Tester si la liste est vide (ou la mettre à vide)
- Accéder au $n^{ème}$ élément de la liste (pour le modifier, supprimer, etc ...)
- Insérer un nouvel élément derrière le $n^{ème}$ élément
- Fusionner 2 listes
- Rechercher un élément d'une valeur particulière
- Trier une liste

Les listes

Type Liste

```
Utilise Elément, Booléen, Place
Opérations

    liste_vide : Liste → Booléen
    longueur   : Liste → Entier
    insérer    : Liste x Entier x Elément → Liste
    Supprime   : Liste x Entier → Liste
    Nème       : Liste x Entier → Elément
    accès      : Liste x Entier → Place
    contenu    : Liste x Place → Elément
    suiv       : Liste x Place → Place
```

Les listes

Type Liste

Pré conditions

.insérer(l,k,e) *est-défini-ssi* 1 ≤ k ≤ longueur(l)+1

.supprimer(l,k) *est-défini-ssi* 1 ≤ k ≤ longueur(l)

.Nème(l,k) *est-défini-ssi* 1 ≤ k ≤ longueur(l)

.accès(l,k) *est-défini-ssi* 1 ≤ k ≤ longueur(l)

.suiv(l,p) *est-défini-ssi* p ≠ accès(l,longueur(l))

Les listes

❖Implantation des listes

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

➢ 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 leurs places respectives.

Les listes

❖Implantation des listes

➢ Utilisation de tableaux

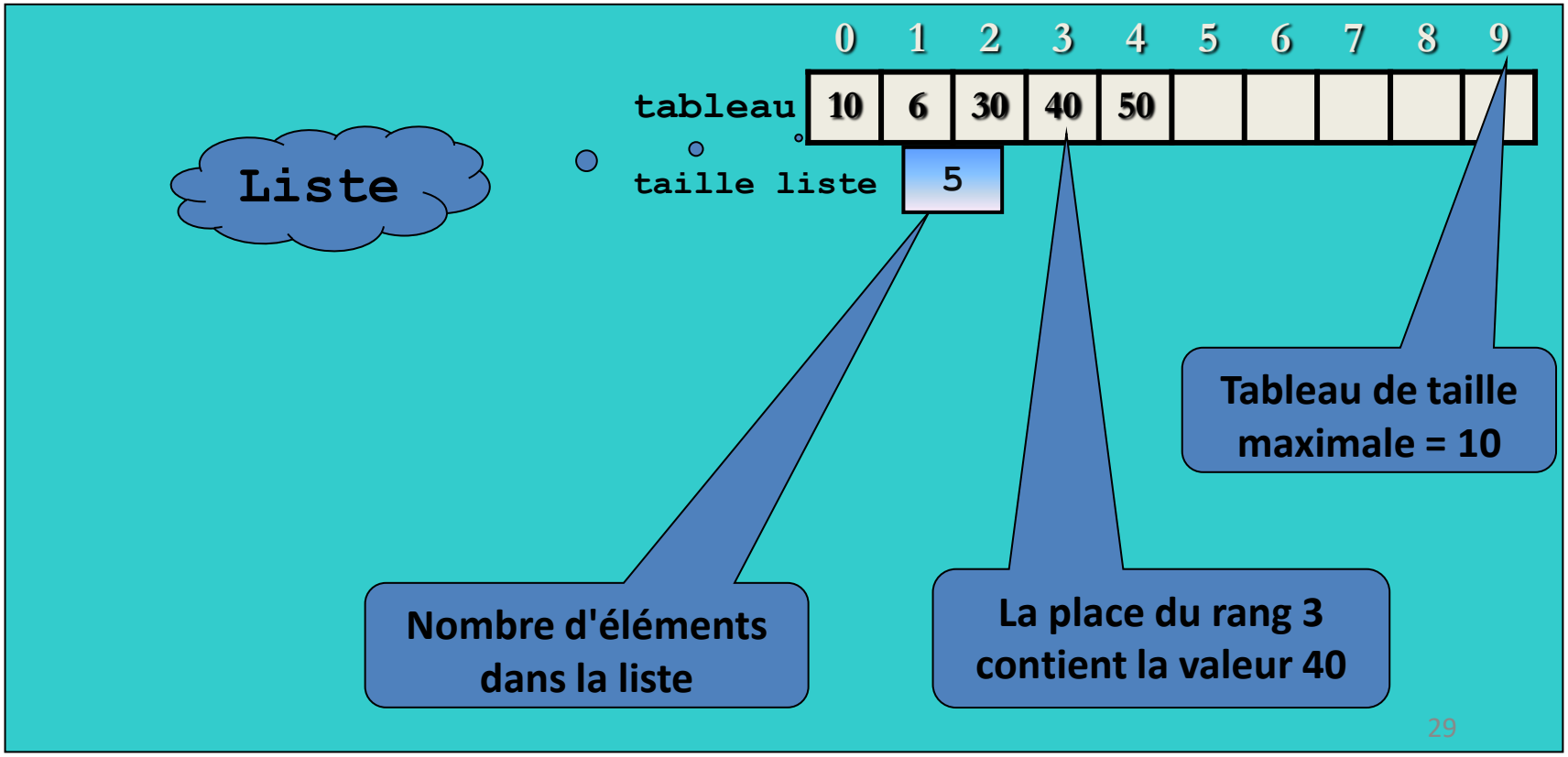
Cependant, l'utilisation de tableaux possède quelques inconvénients :

- 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.

Les listes

❖Implantation des listes

➢ Utilisation de tableaux



Les listes

❖Implantation des listes

➢ Utilisation de tableaux

- 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.

Les listes

❖Implantation des listes

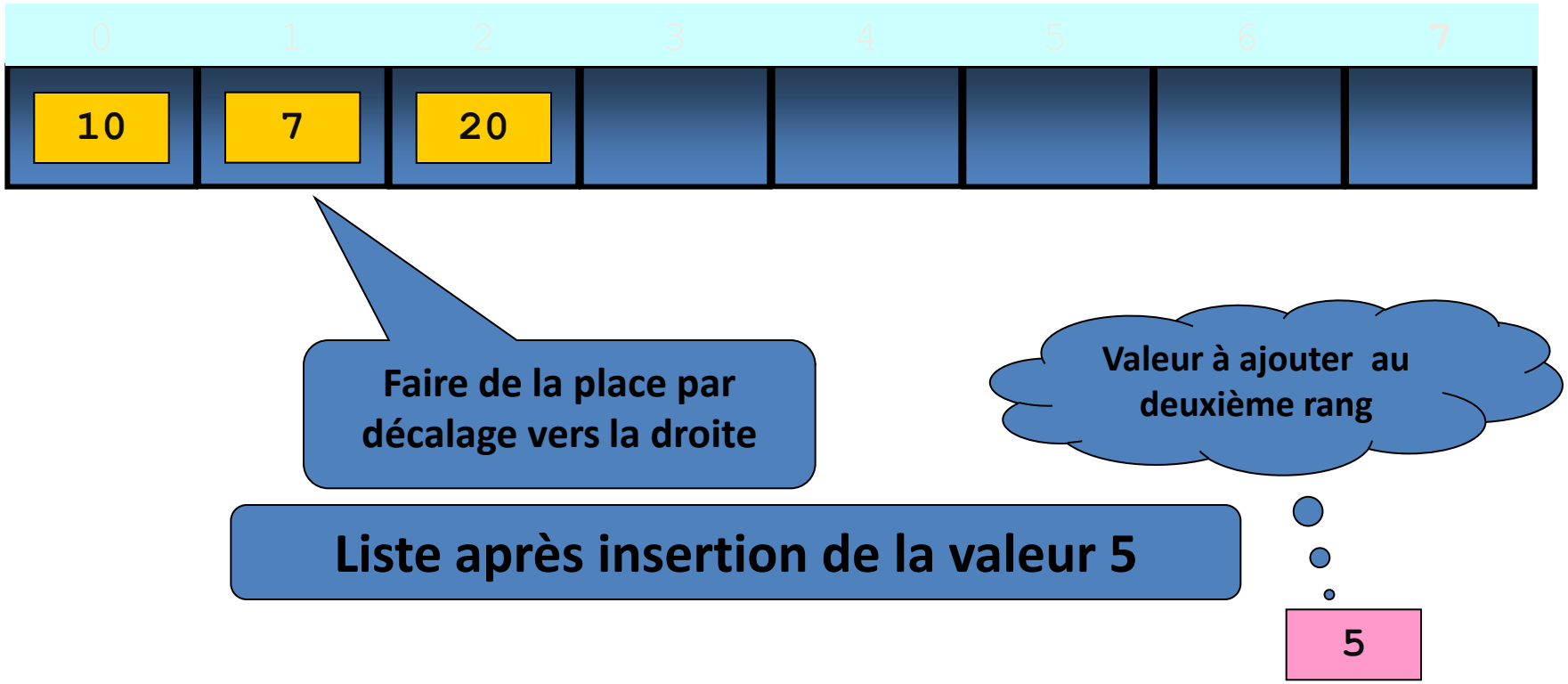
➢ Utilisation de tableaux

- **Représentation contiguë d'une liste par tableau**

  Avantage: L'accès au $n^{ème}$ élément est immédiat: t[n].

Les listes

❖ **Représentation d'une liste à l'aide d'un tableau:**



Les listes

❖Implantation des listes

➢ 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.

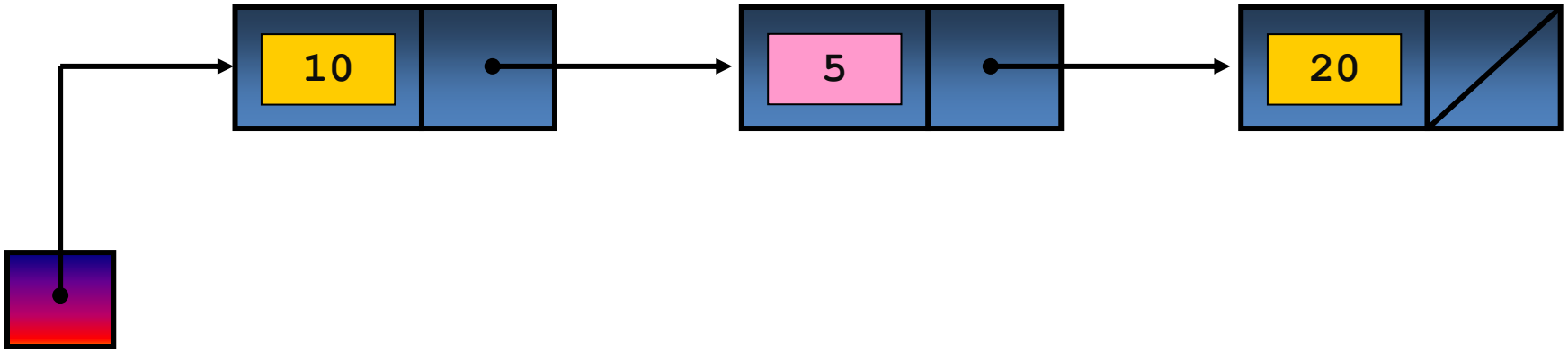
➢ Inconvénient

- 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.

Les listes

❖Implantation des listes

- Utilisation de pointeurs



Les listes

❖Représentation 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 = {a1, a2, a3, a4}

Si par exemple on insère a'3 après a3 dans la liste, on réalise 3 affectations.

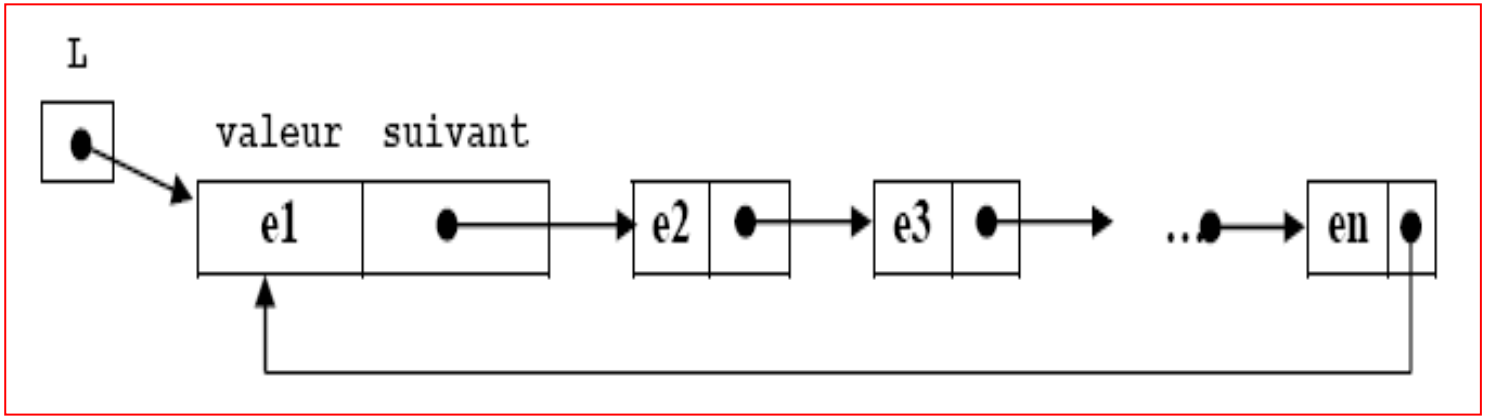
Idem pour Insertion et la Suppression, Coût: O(1) si on connaît l'endroit pour insérer.

Une liste se définit de façon récursive .

Les listes

❖Variantes utiles d'une liste

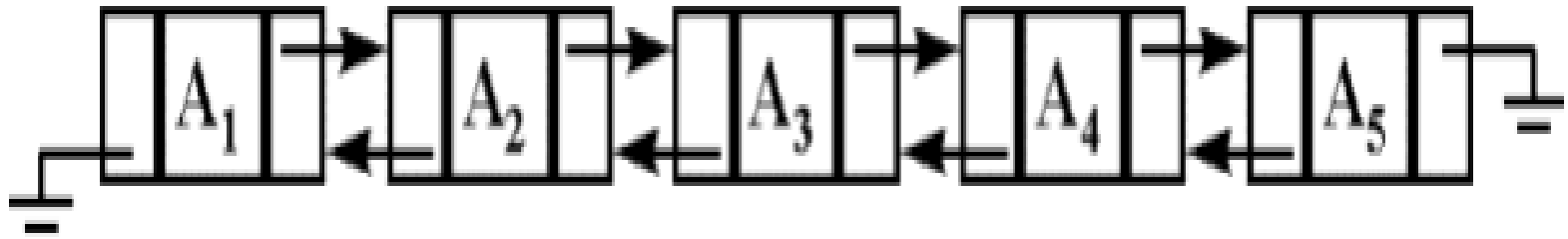
Les listes circulaires: On a une liste L = { e_1, e_2, ..., e_n} dont le suivant du dernier e_n est le premier e_1



Les listes

❖ Les listes doublement chaînées

Utile quand on veut accéder facilement au prédécesseur d'un élément de la liste.



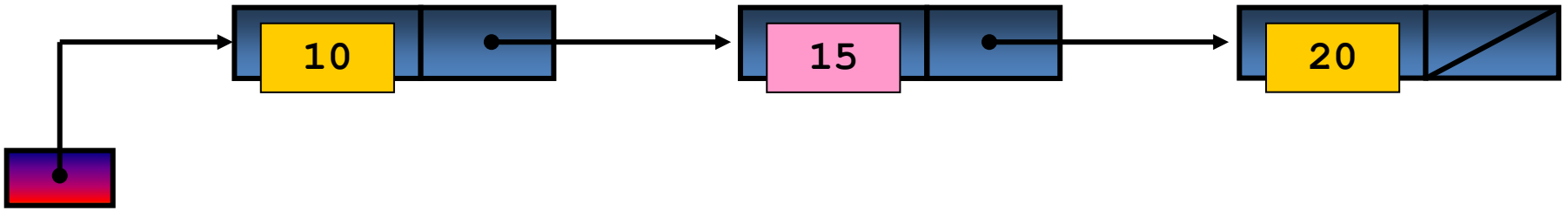
liste doublement chaînée

Les listes

❖Les listes triées

- Dans cette liste, il existe un ordre total sur les clés.

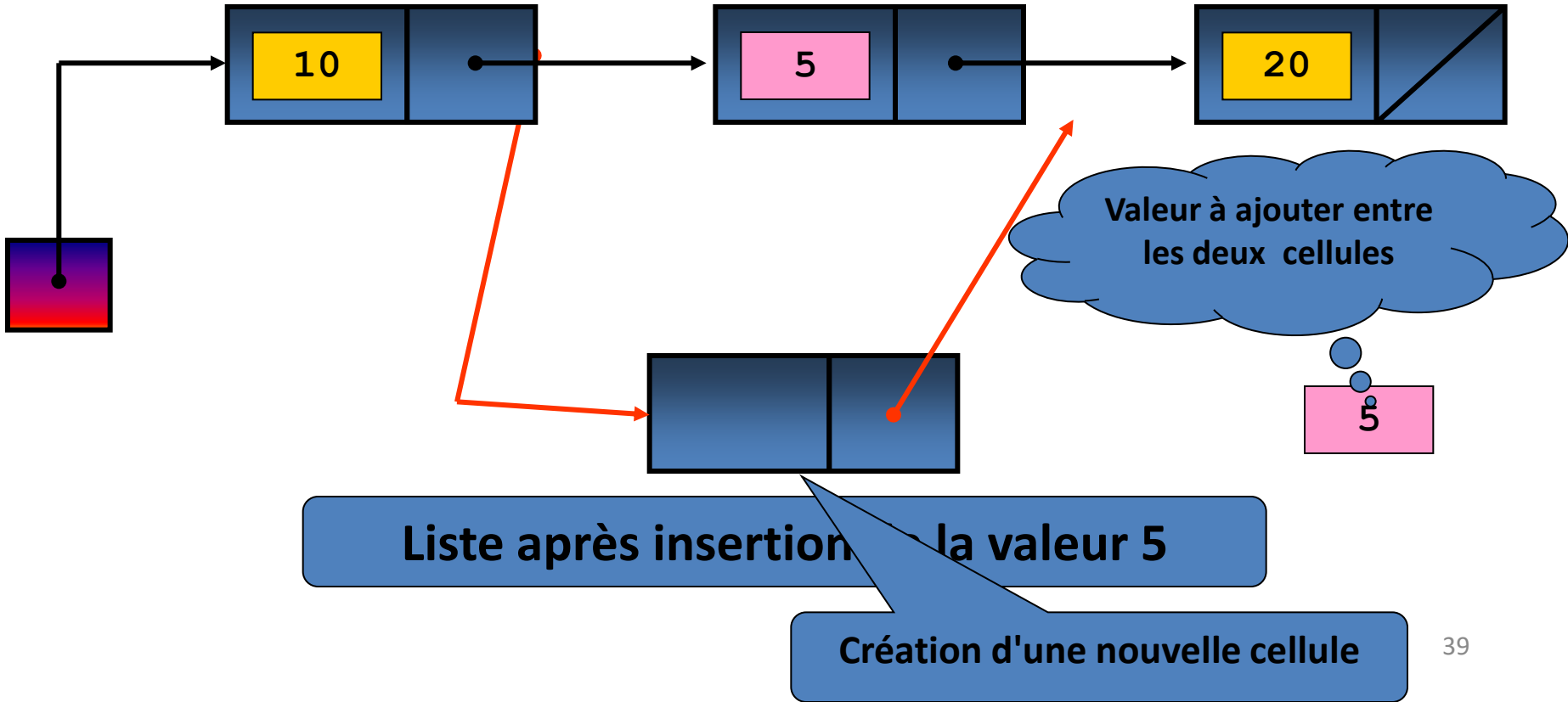
- L'ordre des enregistrements dans la liste respecte l'ordre sur les clés.



Les listes

❖ Les opérations sur les listes

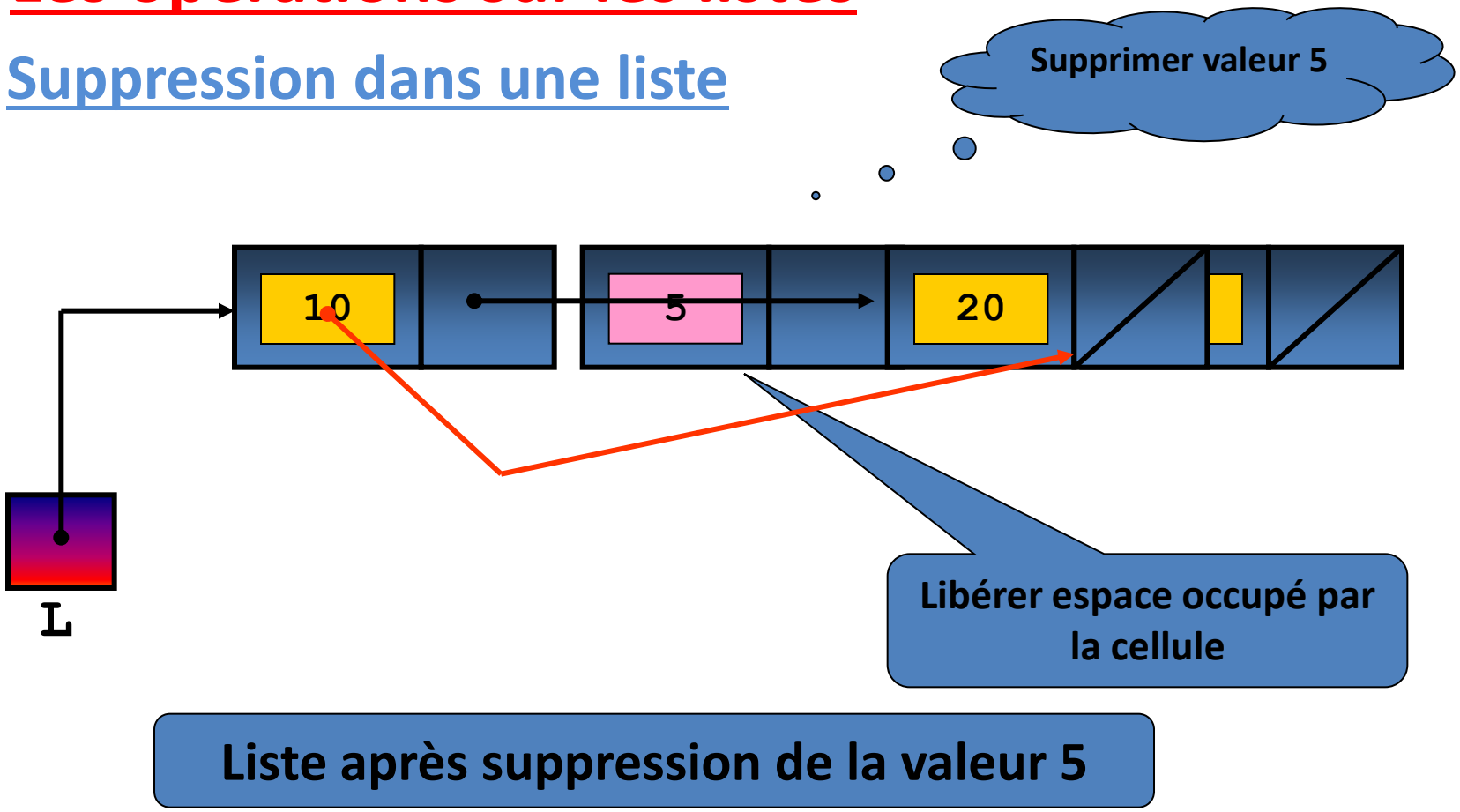
➢ Insertion dans une liste



Les listes

❖ Les opérations sur les listes

➢ Suppression dans une liste



Liste après suppression de la valeur 5

Les files

❖Définition

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).

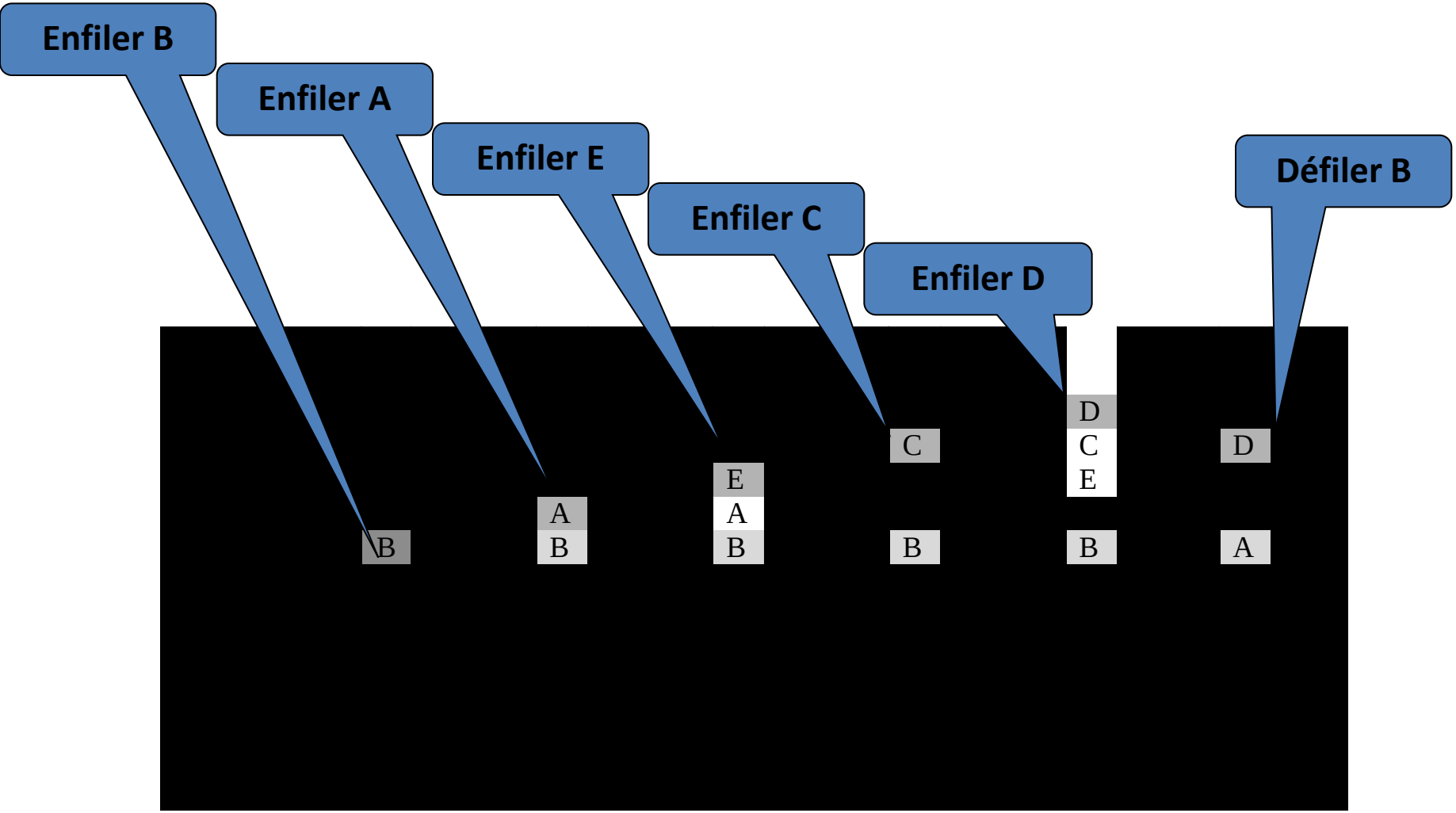
Les files

❖Définition

- Une file d'attente peut être définie comme une collection d'éléments dans laquelle tout nouveau élément est inséré à la fin et tout élément ne peut être supprimé que du début.

- C'est le principe "**FIFO**", abréviation de "First In, First Out" qui veut dire "premier entré premier servi".

Les files

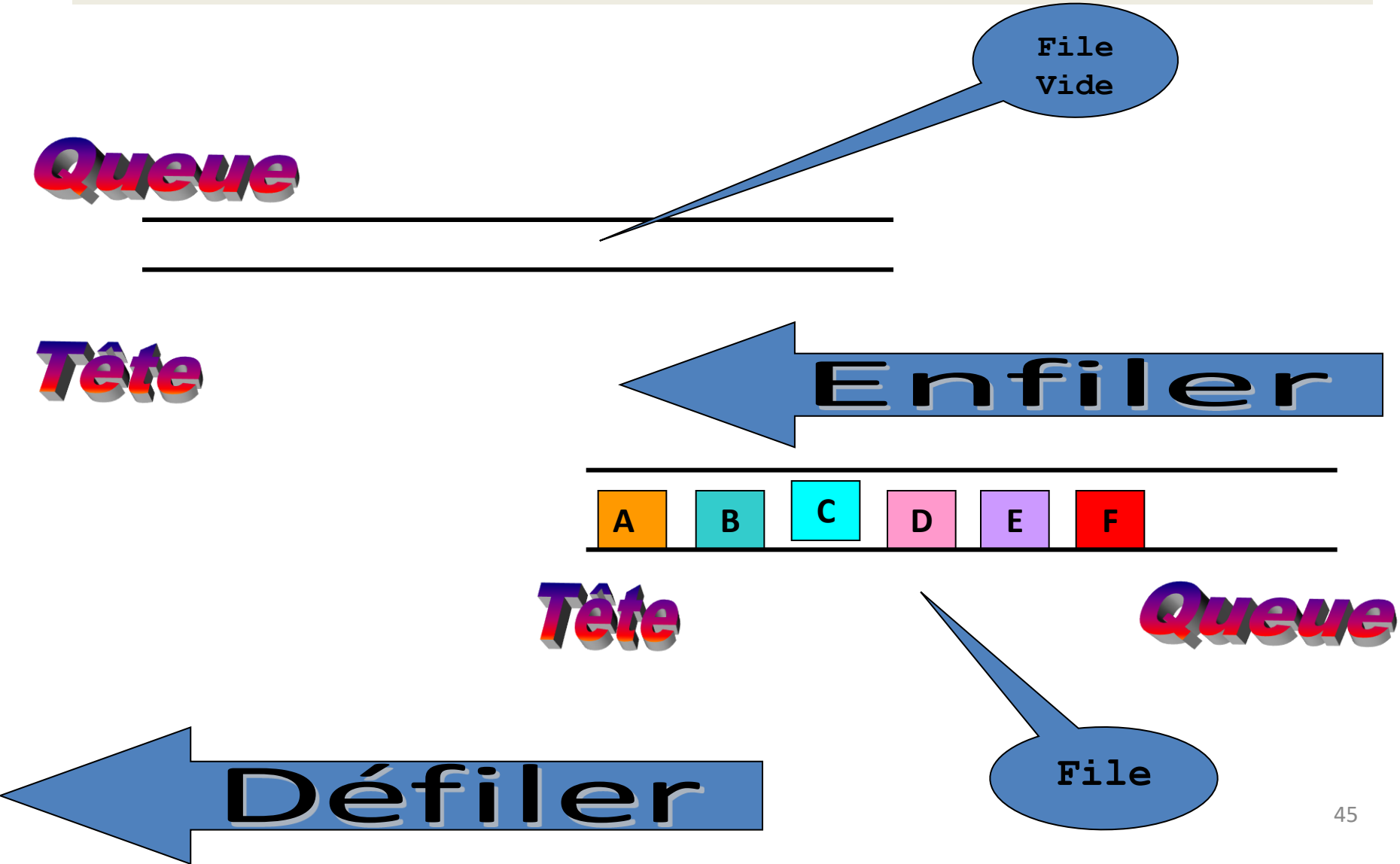

Enfiler B
Enfiler A
Enfiler E
Enfiler C
Enfiler D
Défiler B
B
A
B
E
A
B
C
B
D
C
E
B
D
A

Les files

❖Domaine d'application

- 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.
- Modélise la file d'attente des clients devant un guichet.
- Les travaux en attente d'exécution dans un système de traitement par lots (batch).
- Les messages en attente dans un commutateur de réseau téléphonique.

Les files : Exemple



Les files

❖Les opération sur les files

```
CreerFile( var F:File ) => File
initialise la file F à vide

Enfiler( x:Tqlq; var F:File ) File x Elt => File
insère x en queue de la file F

Defiler( var x:Tqlq; var F:File ) File => Elt x File
retire dans x, l'élément en tête de la file F

FileVide( F:File ) : Booleen File    => Bool
FilePleine( F:File ) : Booleen File  => Bool
testent l'état de F (vide ou pleine)

Conditions: Défiler est défini si not FileVide
Enfiler est défini si not FilePleine
```

Les files

❖Implantation des files

- **Représentation contiguë (*par tableau*) :**
  - Les éléments de la file sont rangés dans un tableau
  - Deux entiers représentent respectivement les positions de la tête et de la queue de la file

- **Représentation chaînée (*par pointeurs*) :**
  - Les éléments de la file sont chaînés entre eux
  - Un pointeur sur le premier élément désigne la file et représente la tête de cette file
  - Un pointeur sur le dernier élément représente la queue de file
  - Une file vide est représentée par le pointeur **`NULL`**

Les files

❖Représentation par tableau d'une File

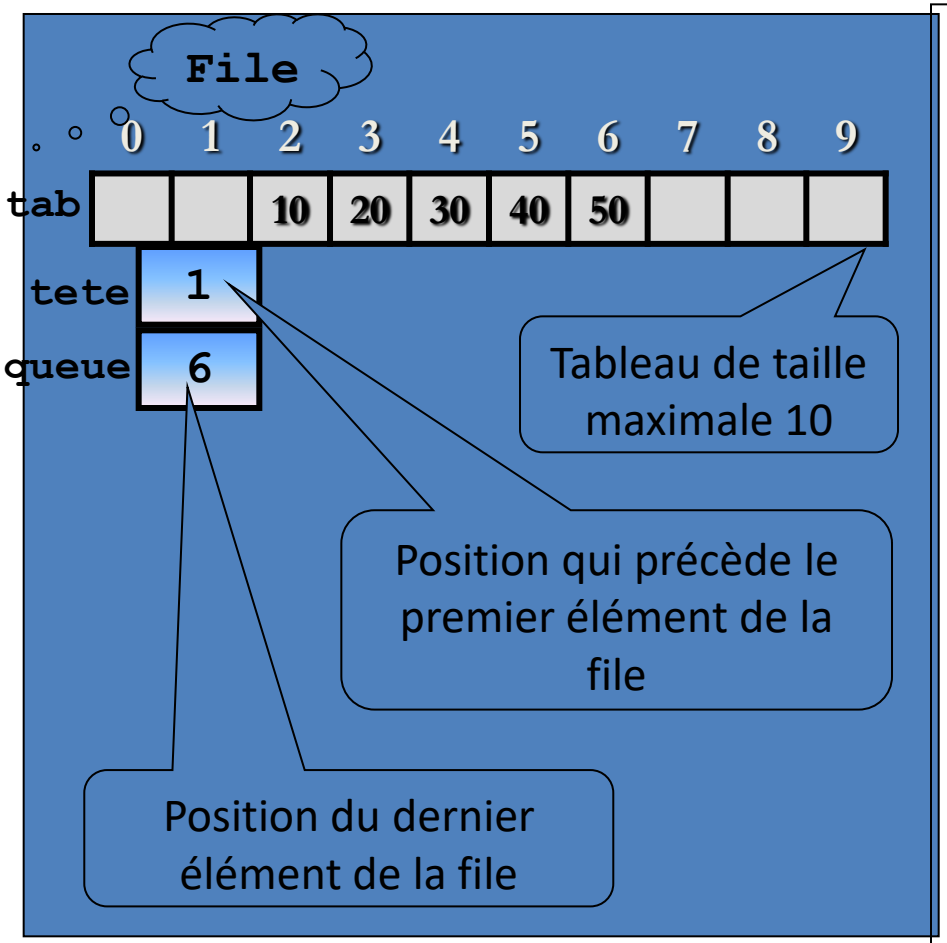
- ***tête de file :*** position précédant premier élément
- ***queue de file :*** position du dernier élément

- ***Initialisation :*** `tête ← queue ← -1`

- ***Inconvénient :*** on ne peut plus ajouter d'éléments dans la file, alors qu'elle n'est pas pleine

Les files

❖La représentation par tableau

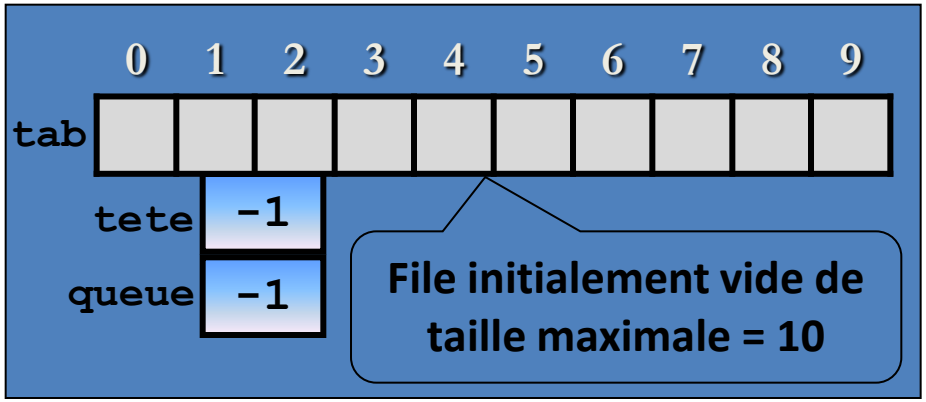


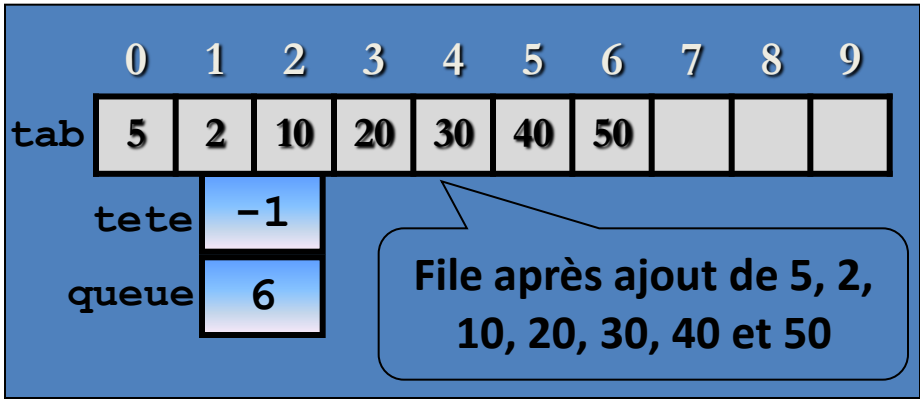

```
/* File contiguë en C */

// taille maximale file
#define MAX_FILE 10

// type des éléments
typedef int Element;

// type File
typedef struct {
 Element tab[MAX_FILE];
 int tete;
 int queue;
} File;
```

Les files

❖ Représentation Contiguë d'une File (par tableau simple avec décalage)

- ***Décaler les éléments de la file après chaque suppression***
- ***Inconvénient :*** décalage très coûteux si la file contient beaucoup d'éléments





Les files

❖ Représentation Contiguë d'une File par tableau circulaire

- **Gérer le tableau de manière circulaire *:***

  Le suivant de l'élément à la position `i` est l'élément à la position `(i+1) modulo MAX_FILE`

- ***Convention :***

  File autorisée à contenir `MAX_FILE-1` éléments

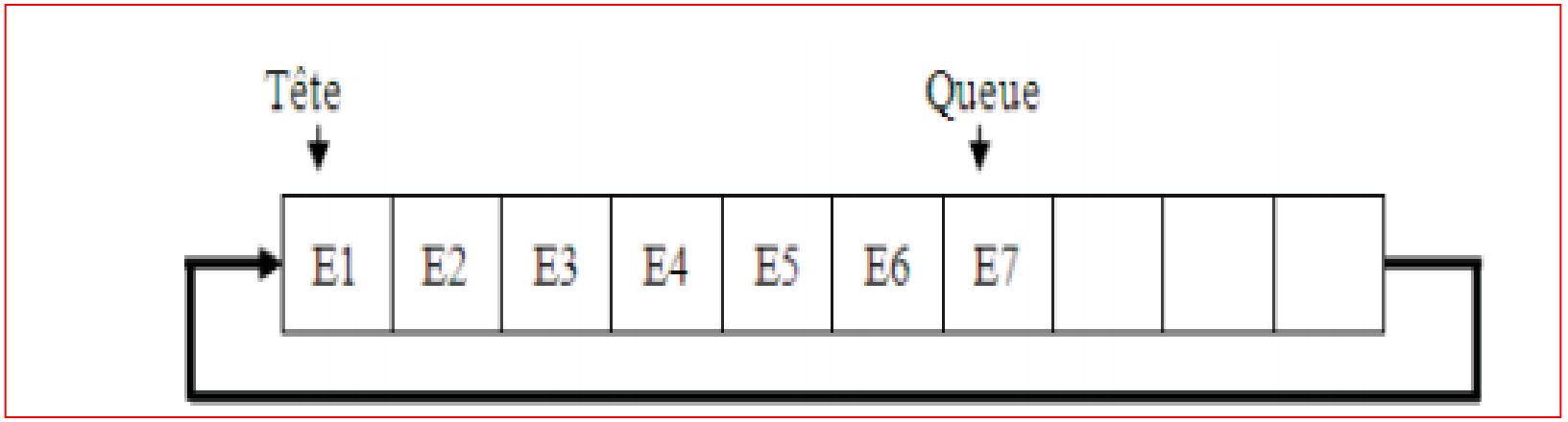
- **Initialisation *:***

  `tête ← queue ← 0`

- **Toutes les places dans le tableau sont exploitées**

Les files

❖Représentation Contiguë d'une File (par tableau circulaire)



La file est vide si Tete = Queue

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

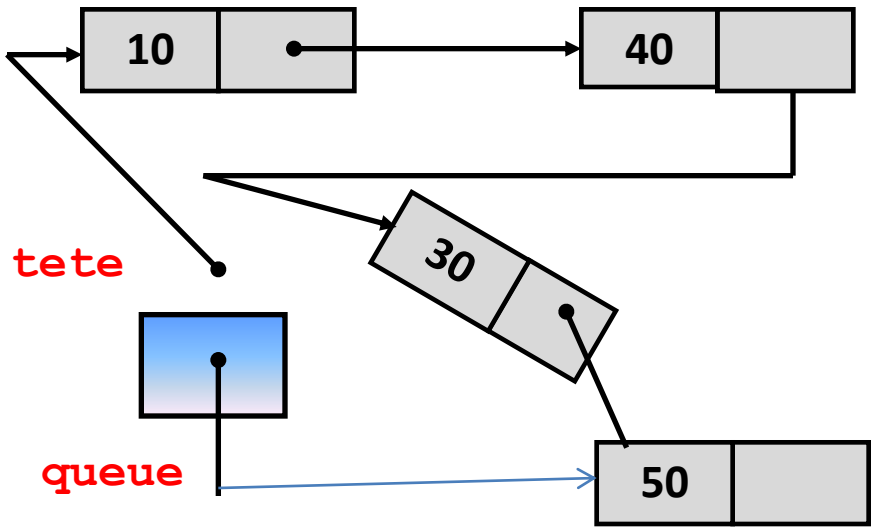
Les files

❖Représentation dynamique

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

Les files

❖ **Représentation dynamique**

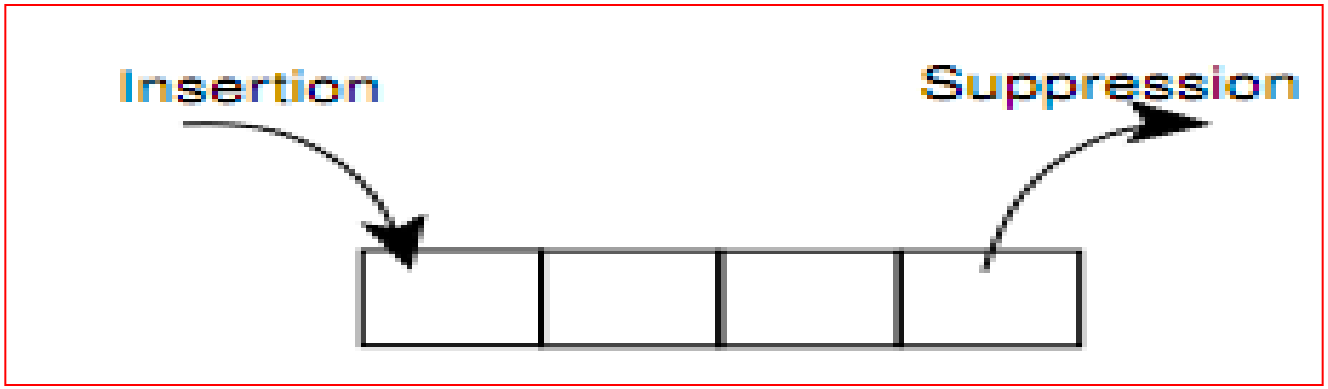



```
/* File chaînée en C */

// type des éléments
typedef int element;

// type Cellule
typedef struct cellule {
  element valeur;
  struct cellule *suivant;
 } Cellule;

// type File
typedef struct {
 struct cellule *tete;
 struct cellule *queue;
} File;
```

Les files

❖Les opérations sur les files

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)



Les files

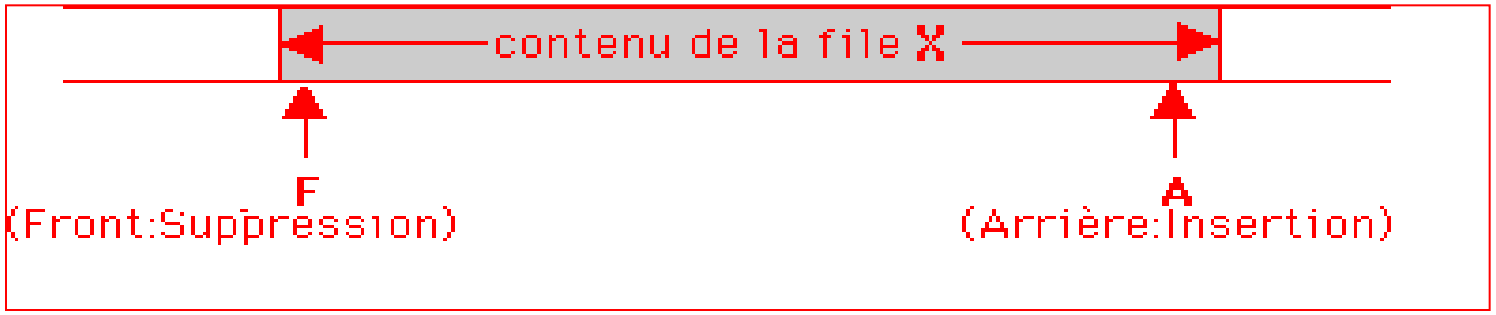
❖Les opérations sur les files

- Créer une file vide : **creeFile** (opération d'initialisation de la file)
- Insérer un élément dans la file : **enfile** (F e)
- Supprimer un élément dans la file : **defile** (F)
- Tester si une file est vide : **estFileVide** (F)
- Récupérer l' élément en queue de file : **queueFile**(F)

Les files

❖Les opérations sur les files

- Les opérations d'insertion se font à une extrémité (arrière) et les opérations de suppression se font à l'autre extrémité (front).



Pour cela, on utilisera un pointeur F qui nous indiquera où se trouve le front et un pointeur A qui nous indiquera où se trouve l'arrière.

Les files

File d'attente particulière

File d'attente avec priorité

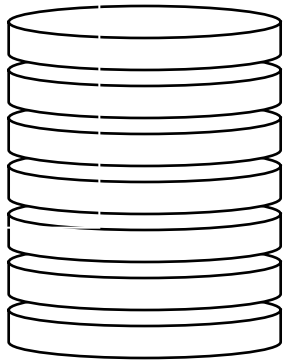
- Une file d'attente avec priorité est une collection d'éléments dans laquelle l'insertion ne se fait pas toujours à la queue. Tout nouvel élément est inséré ,dans la file, selon sa priorité. Le retrait se fait toujours du début.
- Dans une file avec priorité, un élément prioritaire prendra la tête de la file même s'il arrive le dernier. Un élément est toujours accompagné d'une information indiquant sa priorité dans la file.

Les piles

❖Définition

Une pile peut être définie comme une collection d'éléments dans laquelle tout nouveau élément est inséré à la fin et tout élément ne peut être supprimé que de la fin.

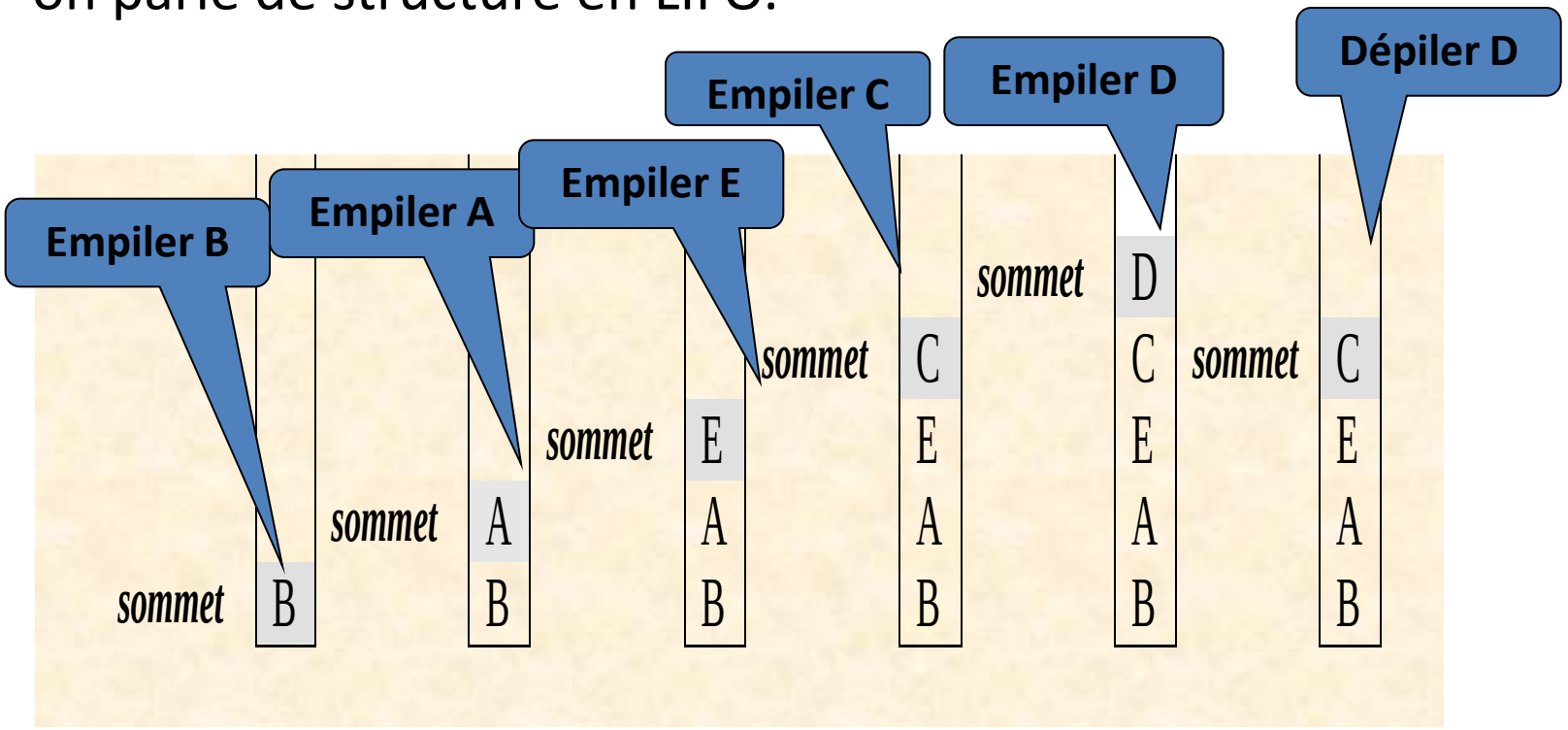
C'est le principe "LIFO", abréviation de "**Last In First Out**" qui veut dire " ***dernier entré premier servi*** **".**



Les piles

❖Définition

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

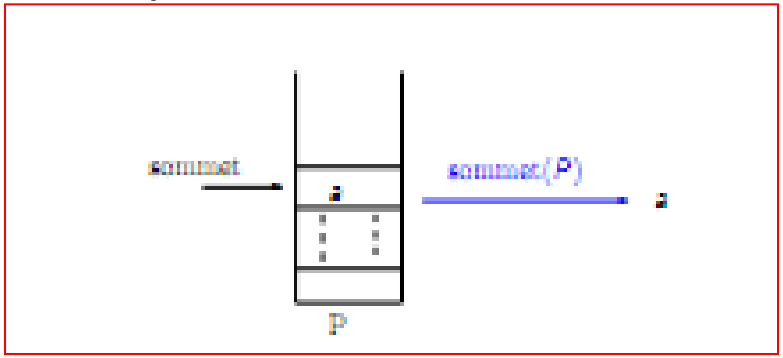


Les piles

❖ Définition

➢ Accès à un élément d'une pile

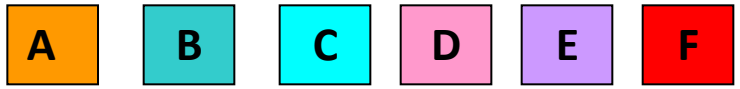
Seul élément accessible d'une pile : son sommet.

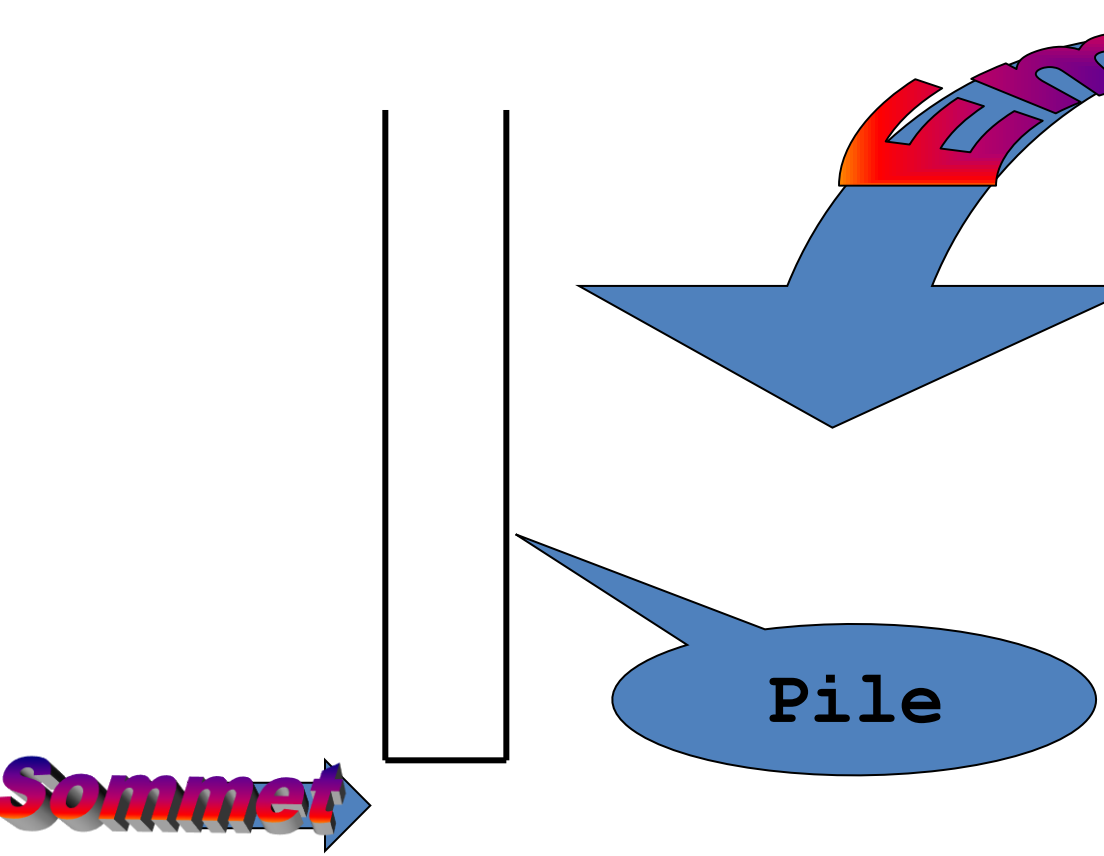


Pour accéder aux éléments situés sous le sommet, nécessité de dépiler la pile.

Les piles

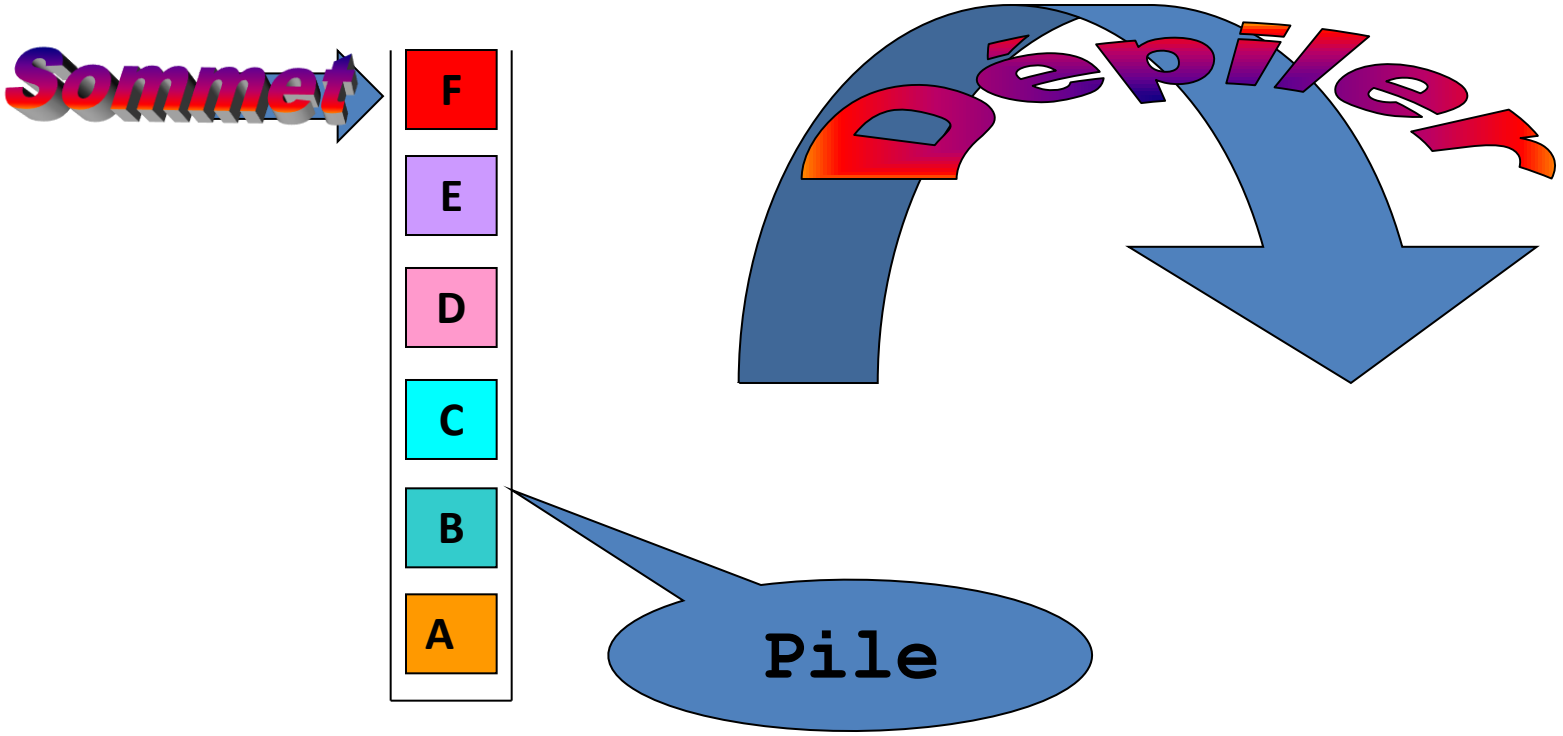
Ajouter dans cet ordre





Les piles

L'opération de dépilement



Les piles

❖Etats d'une pile

Une pile peut être:

- **vide :** elle ne contient aucune donnée ;
- **pleine :** il n'est plus possible de lui ajouter un élément.

En théorie, une pile n'est jamais pleine. Mais toute implémentation des piles utilise de la mémoire, ressource disponible en quantité finie.

Les piles

❖Opérations sur les piles

Créerpile(P) :créer une pile

videEmpiler(P,Val):ajouter Val en sommet de pile.

Dépiler(P,Val):retirer dans Val l'élément en sommet de pile.

Pilevide(P):tester si la pile est **vide.**

Pilepleine(P):tester si la pile est pleine

Les piles

❖Implémentation

❑Au moyen de tableaux

- Les éléments de la pile sont rangés dans un tableau
- Un entier représente la position du sommet de la pile

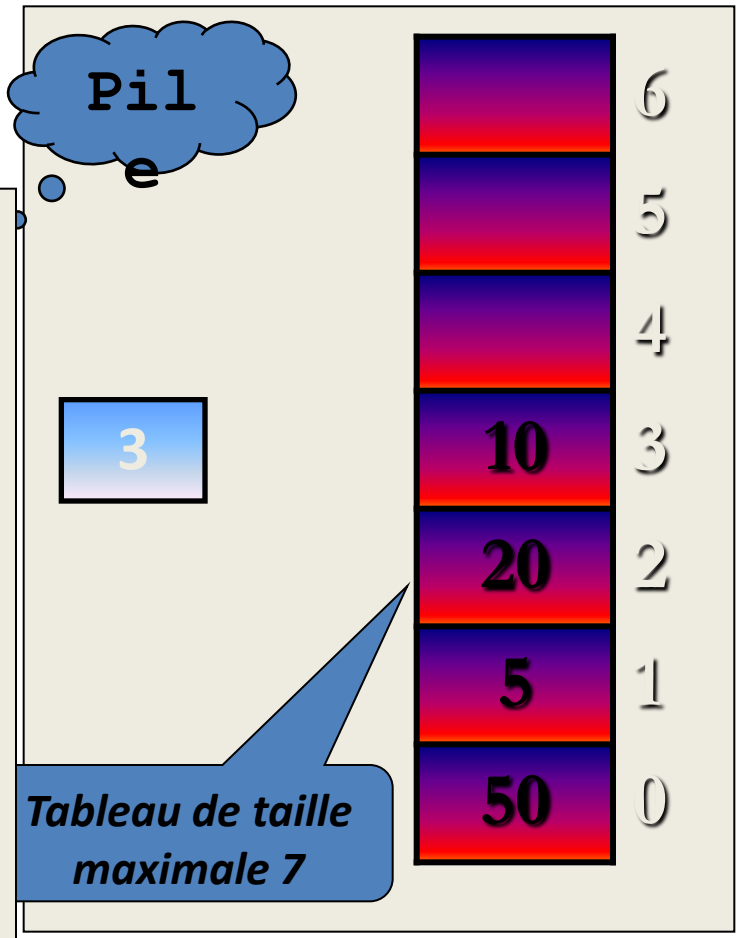
- **Avantage**: Facile car on ne modifie une pile que par un bout. Les opérations sont faciles.
- **Inconvénient**: la hauteur est bornée (allocation statique de la mémoire)

Les piles

❖Implémentation

❑Au moyen de tableaux

```
    /* Pile contiguë en C */

// taille maximale pile
#define MAX_PILE 7

// type des éléments
typedef int Element;

// type Pile
typedef struct {
 Element elements[MAX_PILE];
 int sommet;
} Pile;
```

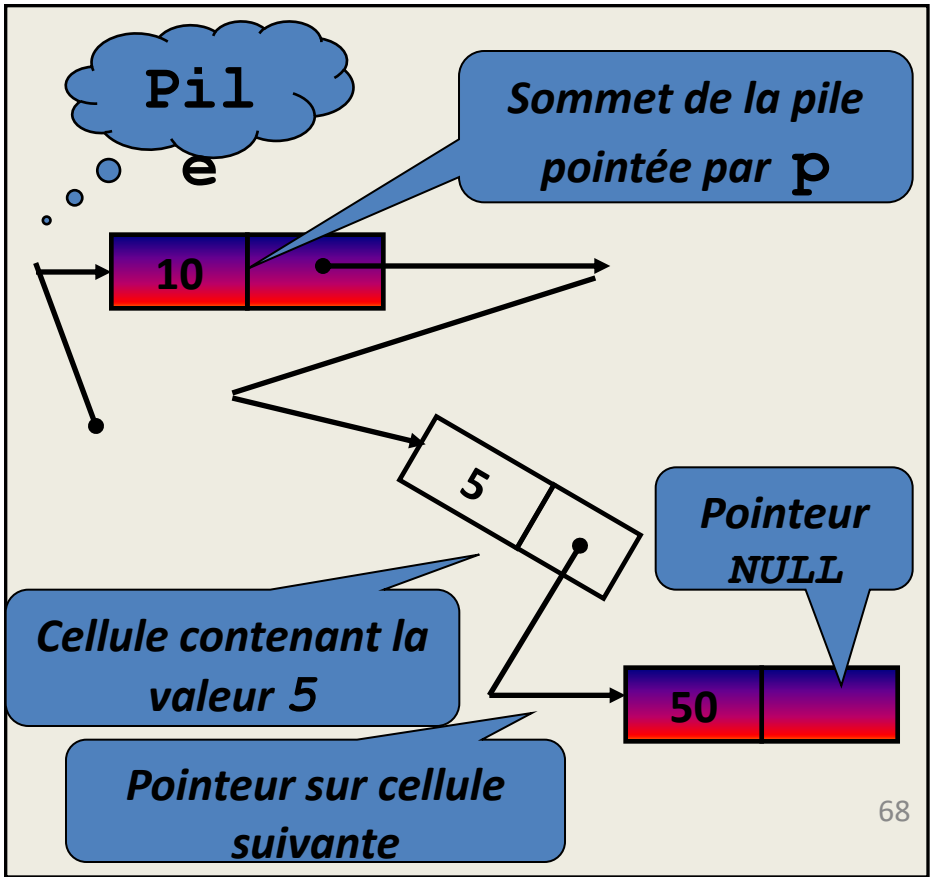


Les piles

❑Représentation chaînée

- **Avantage**: facile avec la tête de liste chaînée sur le haut de la pile.
- **Inconvénient:**
- espace occupé par les pointeurs

```
    /* Pile chaînée en C */

// type des éléments
typedef int element;

// type Cellule
typedef struct cellule {
  element valeur;
  struct cellule *suivant;
 } Cellule;

// type Pile
typedef Cellule *Pile;
```



Les piles

❖Application des piles

❑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.

Les piles

❖ Application des piles

❑ Calcul arithmétique

- Une application courante des piles se fait dans le calcul arithmétique:

  La notation post fixée (polonaise) consiste à placer les opérandes devant l'opérateur. La notation infixée (parenthèsée) 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 piles

❖ Application des piles

❑ Calcul arithmétique

- 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 .

Les piles

❖Application des piles

❑Calcul arithmétique

- 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.

Les piles

❖ Application des piles

❑ Calcul arithmétique

- 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 infixe: A * B/C.

En notation postfixe 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.

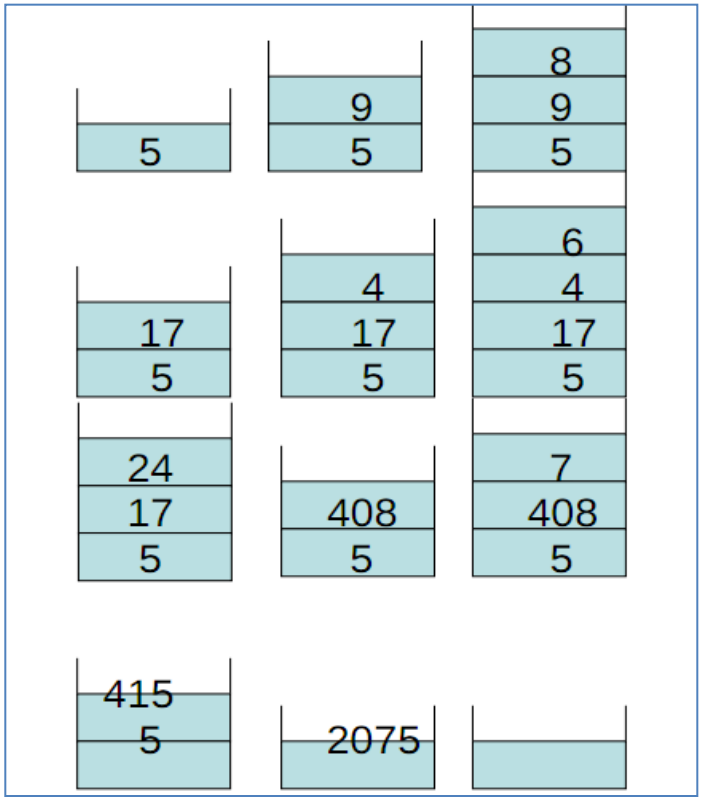
Les piles

➢ Exemple

- Evaluation d'une expression infixée (plus complexe)

  5 * (((9+8)*(4*6))+7)

1 - empiler(5)
2 - empiler(9)
3 - empiler(8)
4 - empiler(dépiler() + dépiler())
5 - empiler(4)
6 - empiler(6)
7 - empiler(dépiler() * dépiler())
8 - empiler(dépiler() * dépiler())
9 - empiler(7)
10 - empiler(dépiler() + dépiler())
11 - empiler(dépiler() * dépiler())
12 - écrire(dépiler())



Les piles

❑ Transformation infixée → postfixée

- Règles de transformation

- Les opérandes gardent le même ordre dans les deux notations
- Les opérateurs sont empilés, en prenant la précaution de dépiler d'abord tous les opérateurs plus prioritaires les dépilements se font dans la chaîne postfixée
- '(' est empilée sans faire de test
- ')' Depiler tous les opérateurs dans la chaîne postfixée, jusqu'à trouver une '(', qui sera écartée
- A la fin, tous les opérateurs encore dans la pile seront dépilés dans la chaîne postfixée.

Les piles

Transformation infixée → postfixée

Exemple :

a + b * c → a

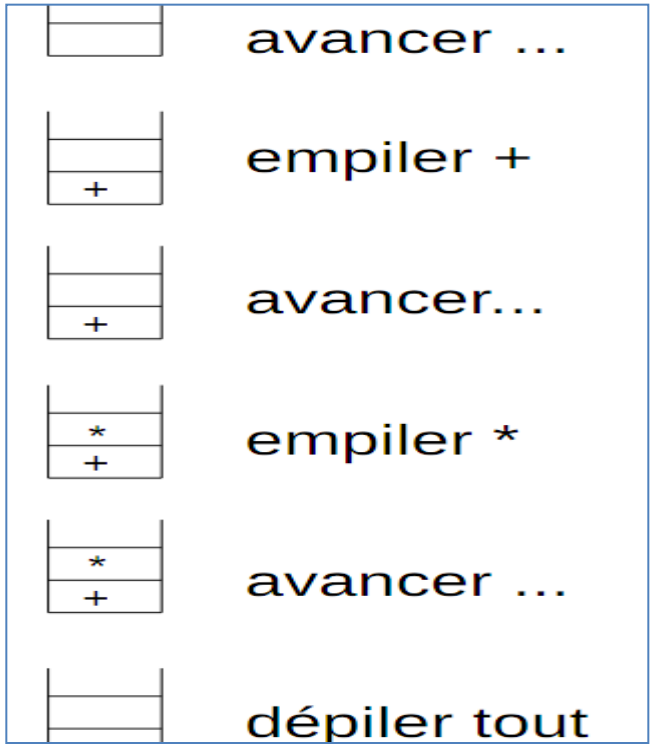
a + b * c → a

a + b * c → a b

a + b * c → a b

a + b * c → a b c

a + b * c → a b c * +



Les piles

❖ Application des piles

- ❑ Mémorisation des appels de procédures imbriquées au cours de l'exécution d'un programme, et en particulier les appels des procédures récursives ;

- ❑ Parcours « en profondeur » de structures d'arbres.