

# Structures de Données

## Cours 1 - Structures de Données Élémentaires

E. Soutil ([eric.soutil@cnam.fr](mailto:eric.soutil@cnam.fr))

Ensiie

2019-2020

# Objectifs du cours

- Travailler sur des données de plus en plus nombreuses et complexes
- Ce qui oblige à les structurer de façon que les opérations (algorithmes) avec lesquelles on les manipule soient les plus efficaces possibles
- Les informaticiens, au cours du temps, ont défini des **Structures de données générales** (tableau, liste, arbre, graphe, ...) et des **Algorithmes généraux** (tri, accès aux données, ...)
- Avec un souci d'abstraction : dissocier la structure de données de la façon dont elle est implémentée ⇒ notion de **Type abstrait**

- **Structures de données élémentaires** (tableau, liste, pile, file) et opérations élémentaires (création, parcours, tri)
- Mesure d'efficacité des algorithmes : notions de **complexité**
- Des structures plus complexes dédiées : **graphe, arbre, arbre binaire, tas**
- Organisation des données volumineuses : tables de **hachage**
- **Langage d'illustration : Java**
  - ▶ Facilite la structuration et l'abstraction par les objets
  - ▶ Dispose d'un puissant mécanisme de ramasse-miette donc on n'a pas besoin de faire la désallocation d'espaces mémoire inutilisés

# Plan du cours 1 – Structures de données élémentaires

1 Introduction

2 Les tableaux

3 Les listes

4 Les piles

5 Les files

# Plan du cours

1 Introduction

2 Les tableaux

3 Les listes

4 Les piles

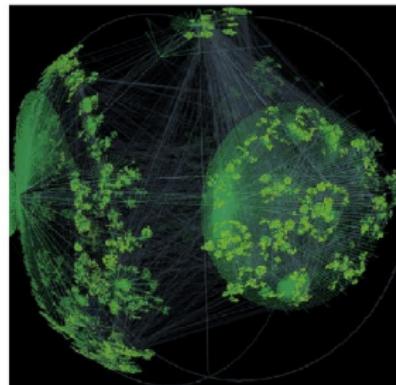
5 Les files

# Structures de données

- Méthode pour stocker et organiser les données pour en faciliter l'accès et la modification
- Une structure de données regroupe :
  - ▶ un certain nombre de données à gérer, et
  - ▶ un ensemble d'opérations pouvant être appliquées à ces données
- Dans la plupart des cas, il existe
  - ▶ plusieurs manières de représenter les données et
  - ▶ différents algorithmes de manipulation.
- On distingue généralement l'**interface** des structures de leur **implémentation**.

# Structures de données et algorithmes

- La résolution de problèmes algorithmiques requiert presque toujours la combinaison de structures de données et d'algorithmes sophistiqués pour la gestion et la recherche dans ces structures.
  - D'autant plus vrai qu'on a à traiter des volumes de données importants.
  - Quelques exemples de problèmes réels :
    - ▶ Routage dans les réseaux informatiques
    - ▶ Moteurs de recherche
    - ▶ Alignement de séquences ADN en bio-informatique



# Types de données abstraits

- Un **type de données abstrait (TDA)** représente l'interface d'une structure de données.
- Un TDA spécifie précisément :
  - ▶ la nature et les propriétés des données
  - ▶ les modalités d'utilisation des opérations pouvant être effectuées
- En général, un TDA admet différentes implémentations (plusieurs représentations possibles des données, plusieurs algorithmes pour les opérations).

# Plan du cours

1 Introduction

2 Les tableaux

3 Les listes

4 Les piles

5 Les files

# Les tableaux

- Un tableau est une structure de données qui réunit des valeurs (données) d'un même type.
- On peut le voir comme une suite de cases contiguës repérées (indiquées) par un entier.
- La taille (nombre de cases) du tableau doit être connue au moment de sa création et ne peut pas être modifiée. La mémoire nécessaire à toutes les cases est allouée une fois pour toutes.
- On a un accès direct en consultation et en modification à chaque case du tableau (repérée par son indice)

# Tableau en Java

- Il existe un type tableau prédéfini en java
- Contrairement aux types primitifs (boolean, int, double, ...) une variable de type tableau :
  - ▶ est un objet
  - ▶ sa valeur est une référence

## Exemple

- ▶ `int[] t; // déclare le tableau d'entiers t et l'initialise à null`
- ▶ `int[] tBis = {1, 2, 3, 4}; // déclare le tableau d'entiers tBis et l'initialise à une référence à un espace mémoire contenant 4 entiers`
- On peut faire toutes sortes d'opérations, dès lors que le tableau n'est pas vide (`==null`) :
  - ▶ `tBis[0] = tBis[0] + 10;`
  - ▶ `for (int i=0; i<tBis.length; i++)`  
`System.out.println(tBis[i]);`

# Plan du cours

1 Introduction

2 Les tableaux

3 Les listes

4 Les piles

5 Les files

# Les listes

## Définition générale

### Définition

*On peut définir une liste comme une suite finie de données de même type (une certaine généralisation du tableau)*

- **définition récursive.** Une liste est :

- ▶ soit vide
- ▶ soit la juxtaposition d'un premier élément avec une autre liste

# Le type abstrait Liste

- Le TDA **Liste** spécifie les 4 opérations que l'on peut appliquer sur une liste :
  - ▶ Tester si elle est vide
  - ▶ La construire par l'ajout d'un élément à une liste existante (éventuellement vide)
  - ▶ Accéder au premier élément (tête de liste)
  - ▶ Accéder à la queue de la liste (c-à-d la liste privée de sa tête)
- On peut dès lors écrire des algorithmes basés sur ces 4 opérations comme le calcul de la taille d'une liste L

# Le type abstrait Liste

- Calcul de la taille d'une liste

```
1 Algo : taille
2 Entrée : une liste L
3 Sortie : un entier n qui représente le nombre d'éléments de la liste
4
5 n=0;
6 liste p=L;
7 Tant que p n'est pas vide faire
8     n=n+1;
9     p= queue(p);
10 Fait;
11 Retourner n
```

# Le type abstrait Liste

- Ou encore tester si un élément  $x$  appartient à une liste  $L$

```
1 Algo : appartient
2 Entrée : une liste L et une donnée x
3 Sortie : un booléen r égal à vrai ssi x appartient à L
4 r = faux;
5 Liste p = L;
6 Tant que p n'est pas vide faire
7   si (tete(p) == x) alors r = vrai; fsi;
8   p = queue(p);
9 Fait;
10 Retourner r
```

- Etc...

# Implémentations d'une Liste

Deux façons de faire :

- **Listes chaînées**, où les différents éléments de la liste sont créés au fur et à mesure des besoins (dynamique)
- **Listes contigües**, où la liste est grosso-modo représentée par un tableau. Ce n'est pas une bonne solution en pratique mais on va le faire ici dans l'objectif d'illustrer la dissociation entre un type abstrait et son implémentation

# Une implémentation java de liste chaînée

- Une liste chaînée est une suite finie de cellules formées
  - ▶ D'une valeur (par exemple un int pour une liste d'entiers)
  - ▶ De la référence (on dit aussi pointeur ou adresse) vers la cellule suivante

```
1 class Liste {  
2     private int valeur;  
3     private Liste suivant;  
4  
5     public Liste(int premier, Liste reste) {  
6         valeur = premier;  
7         suivant = reste;  
8     }  
9  
10    public int tete() { return (this.valeur); }  
11  
12    public Liste queue() { return (this.suivant); }  
13 }
```

# Une implémentation java de liste chaînée

## Utilisation dans un programme

```
1 import java.io.*;
2 public class testListe {
3     static BufferedReader in =
4         new BufferedReader(new InputStreamReader(System.in));
5     public static void main (String[] args) {
6         Liste L = null; int n;
7         try {
8             System.out.print("Combien d'éléments dans la liste ?");
9             n = Integer.parseInt(in.readLine());
10            for (int i=1; i<= n; i++) L = new Liste(i, L);
11            // Affichage
12            Liste p = L;
13            while (p != null){
14                System.out.println(p.tete() + " \n");
15                p = p.queue();
16            }
17        }
18        catch (IOException e) { System.out.print("Problème");
19    }}}
```

# Une implémentation java de liste contiguë (1/4)

- Une liste contiguë est en fait un tableau. Dans cette implémentation, elle est la juxtaposition :
  - ▶ d'un tableau dont chaque case contient un élément de la liste
  - ▶ d'un indice qui indique la tête de liste

```
1 class TabListe{  
2  
3     private int[] tab;  
4     private int indiceFin;  
5  
6     public TabListe() {  
7         tab = new int[100];  
8         indiceFin = -1;  
9     }  
}
```

## Une implémentation java de liste contiguë (2/4)

```
10 // deuxieme constructeur
11 public TabListe(int premier, tabListe reste) {
12     if (reste == null) {
13         tab = new int[100];
14         tab[0] = premier;
15         indiceFin = 0;
16     } else {
17         tab = new int[100];
18         for(int i = 0; i < tab.length; i++)
19             tab[i] = reste.tab[i];
20         indiceFin=reste.indiceFin + 1;
21         tab[indiceFin]=premier;
22     }
23 }
```

## Une implémentation java de liste contiguë (3/4)

```
24     public int tete() {
25         return (this.tab[this.indiceFin]);
26     }
27
28     public TabListe queue(){
29         if (this.indiceFin==0) // un seul element
30             return (null);
31         TabListe L = new TabListe();
32         for (int i=0; i < L.tab.length; i++)
33             L.tab[i] = this.tab[i+1];
34         L.indiceFin = this.indiceFin-1;
35         return (L);
36     }
37 }
```

# Une implémentation java de liste contiguë (4/4)

## Utilisation dans un programme

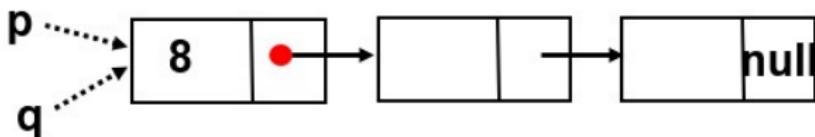
```
1 public class TestTabListe {  
2     static BufferedReader in =  
3         new BufferedReader(new InputStreamReader(System.in));  
4     public static void main(String[] args) {  
5         TabListe L = null; int n;  
6         try {  
7             System.out.print("Combien d'elements avez-vous dans la liste?");  
8             n = Integer.parseInt(in.readLine());  
9             for (int i=1; i<= n; i++) L = new TabListe (i, L);  
10            // Affichage  
11            TabListe p = L;  
12            while(p != null){  
13                System.out.println( p.tete() + " \n");  
14                p = p.queue();  
15            }  
16        }  
17        catch (IOException e) { System.out.print("Problème");  
18    }}}
```

# Conclusion sur nos deux implémentations de Liste

- Clairement transparent au programmeur comme à l'utilisateur du programme
- **Mais**, dans l'implémentation par un tableau, nous avons "ignoré" des problèmes importants :
  - ▶ Initialisation à 100 de la taille du tableau, pourquoi pas une autre taille ?
  - ▶ Arrivée à la fin du tableau (liste pleine) : notre choix de taille maximale peut être sous-dimensionné ou sur-dimensionné suivant l'usage
  - ▶ Il est bien sûr possible de tenir compte de ces problèmes dans une implémentation plus réaliste et plus fine
- L'implémentation d'une liste par **liste chaînée** est
  - ▶ la plus dynamique
  - ▶ la plus efficace
  - ▶ la plus facile

C'est celle que nous privilégions dans la suite de ce cours

# Quelques rappels sur les références et méthodes supplémentaires pour la liste chaînée



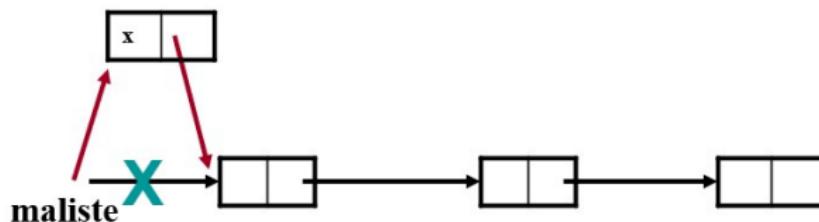
```
1 Liste p, q;      // références sur une « cellule »  
2 p.valeur = 6;  
3 q = p;          // alors q.valeur==6 et q.suivant==p.suivant  
4 q.valeur = 8;    // alors p.valeur==8 aussi  
5  
6  
7 p = null;       // « vide » la liste p
```

# Méthodes supplémentaires pour la liste chaînée

```
1 public Liste insererEnTete(int x){  
2     Liste L = new Liste(x, this);  
3     return L;  
4 }
```

Utilisation dans un programme :

```
maliste = maliste.insererEnTete(x);
```

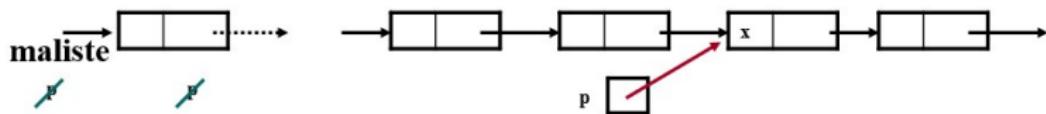


# Méthodes supplémentaires pour la liste chaînée

```
1 public boolean appartient (int x) {  
2     Liste p = this;  
3     while (p != null){  
4         if (p.valeur == x) return true;  
5         p = p.suivant;  
6     }  
7     return false; // l'élément n'a pas été retrouvé  
8 }
```

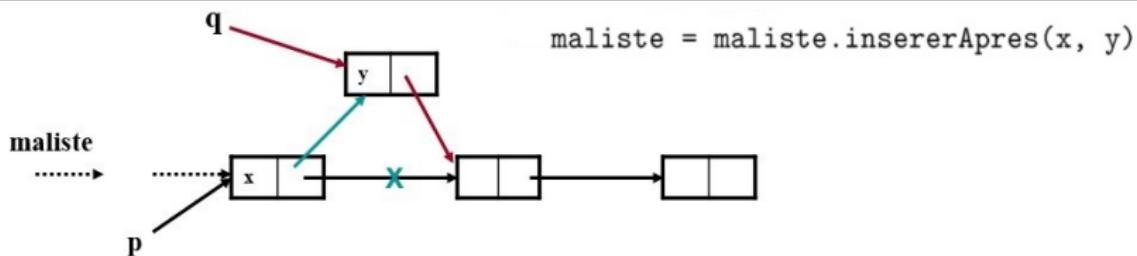
Utilisation dans un programme :

```
boolean present = maliste.appartient(x)
```



# Méthodes supplémentaires pour la liste chaînée

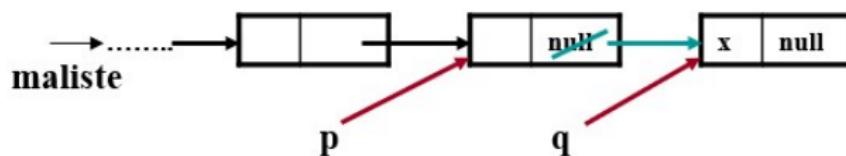
```
1 public Liste insererApres (int x, int y){  
2     // ajoute y apres x si x est present  
3     Liste p = this;  
4     while(p != null){  
5         if(p.valeur != x) {  
6             p = p.suivant;  
7         } else { // x est trouve on insere y juste apres  
8             Liste q = new Liste(y, p.suivant);  
9             p.suivant = q;  
10            return this;      }  
11        }  
12    return this;  
13 }
```



# Méthodes supplémentaires pour la liste chaînée

```
1 public Liste insererEnQueue(int x){  
2     //insere x en queue de liste  
3     Liste p=this;  
4     Liste q = new Liste (x, null);  
5     if (p == null){  
6         return q;  
7     } else { // on va à la fin de la liste  
8         while (p.suivant != null){  
9             p = p.suivant;  
10        }  
11        p.suivant=q;  
12    }  
13    return this;  
14 }
```

maliste = maliste.insererEnQueue(x)



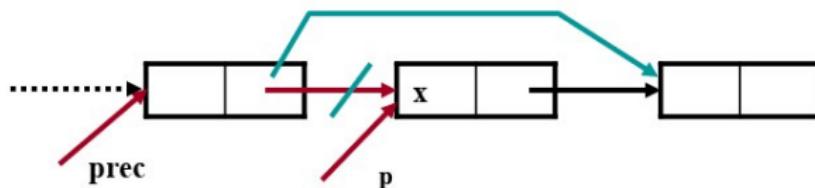
# Méthodes supplémentaires pour la liste chaînée

```
1 public Liste supprimer(int x) {  
2     Liste p = this; //reference cellule courante  
3     Liste prec = null; //predecesseur de p  
4     while(p != null){  
5         if (p.valeur != x) {  
6             prec = p;  
7             p = p.suivant;  
8         } else { // x trouve  
9             if (prec==null) {//premier elem de la liste  
10                 return (p.suivant);  
11             } else {  
12                 prec.suivant=p.suivant;  
13                 return this;  
14             }  
15         }  
16     }  
17     return this;  
18 }
```

# Méthodes supplémentaires pour la liste chaînée

Utilisation dans un programme :

```
maliste = maliste.supprimer(x)
```



# Conclusion la structure de Liste

- C'est aussi une structure de données très souple car on peut ajouter, supprimer, ..., à n'importe quel endroit de la liste
- Nous allons voir des structures plus restrictives : pile et file

# Plan du cours

1 Introduction

2 Les tableaux

3 Les listes

4 Les piles

5 Les files

# Les piles

## Définition

- Une **pile** (en anglais : stack) est une structure de données où les insertions et les suppressions se font toujours du même côté  
**LIFO (Last-in First out)**
- L'insertion s'appelle **empiler**, la suppression **dépiler** et on doit pouvoir tester si la pile est vide.
- Il doit de plus être possible de pouvoir lire le sommet de la pile

# Une implémentation java de pile

- On va s'appuyer sur la structure de liste d'entiers que nous avons déjà implémentée

```
1 class Pile {  
2     private Liste L;  
3  
4     public Pile() {  
5         L = null;  
6     }  
7     public boolean estVide() {  
8         return (this.L == null);  
9     }  
10    public void empiler(int a) {  
11        this.L = new Liste(a, this.L);  
12    }  
13    public void depiler() {  
14        this.L = this.L.queue();  
15    }  
16    public int sommet() { return (this.L.tete()); }  
17 }
```

# Pile – Utilisation en informatique

- Pile d'appels de fonctions dans un programme
- Pile d'annulation d'actions dans un éditeur de textes
- Pile de vérification de la correction des parenthèses d'une chaîne de caractères
- ...

# Utilisation pour la vérification des parenthèses

```
1 import java.io.*;
2 public class testPile {
3     static BufferedReader in =
4         new BufferedReader(new InputStreamReader(System.in));
5     public static void main (String[] args) {
6         Pile P = new Pile ();
7         try {
8             for (int i=0; i< args.length; i++) {
9                 if (args[i].equals("(")) P.empiler(0);
10                if (args[i].equals(")")) P.depiler();
11            }
12            if (P.estVide())
13                System.out.print("Expression bien parenthésée");
14            else System.out.print("Expression mal parenthésée");
15        }
16        catch (Exception e) {
17            System.out.print("Problème, expression mal parenthésée");
18        }}}
```

# Utilisation pour la vérification des parenthèses

## Exemples d'utilisations

- `java testPile a ( b ) ( ( c ) )`

Expression bien parenthésée

- `java testPile a b cc ( ( (`

Expression mal parenthésée

- `java testPile a ( ) f )`

Problème, expression mal parenthésée

# Plan du cours

1 Introduction

2 Les tableaux

3 Les listes

4 Les piles

5 Les files

# Les files – Définition

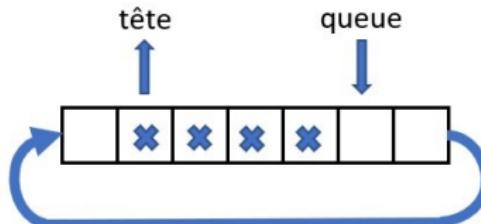
- Une **file** (en anglais : queue) est une structure de données où les insertions se font d'un côté et les suppressions se font de l'autre côté.

**FIFO (First-in First out)**

- insérer = **enfiler**
- supprimer = **défiler**



- Exemples : files d'attente à un guichet, tampon de messages
- En général : **File circulaire**



# Les files

## Méthodes

`enfiler(Elt x)`

`defiler()`

`estVide()`

`estPleine()`

*file vide :*

*file pleine :*

## Conditions

file non pleine

dépôt en queue

file non vide

retrait en tête

retourne booléen

retourne booléen

*tête == queue*

*tête == (queue+1) mod taille*

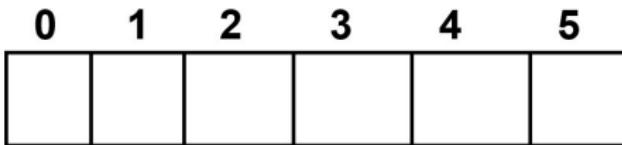
# Les files

| Méthodes                    | Conditions                                            |
|-----------------------------|-------------------------------------------------------|
| <code>enfiler(Elt x)</code> | file non pleine<br>dépôt en queue                     |
| <code>defiler()</code>      | file non vide<br>retrait en tête                      |
| <code>estVide()</code>      | retourne booléen                                      |
| <code>estPleine()</code>    | retourne booléen                                      |
| <i>file vide :</i>          | $\text{tête} == \text{queue}$                         |
| <i>file pleine :</i>        | $\text{tête} == (\text{queue}+1) \bmod \text{taille}$ |

# Les files

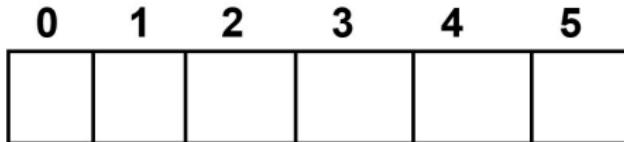
Exemple

**file= new File(6)**



**file.tête=**  
**file.queue=0**

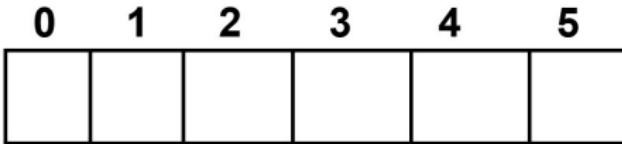
**file.estVide()==vrai**



# Les files

Exemple

**file = new File(6)**

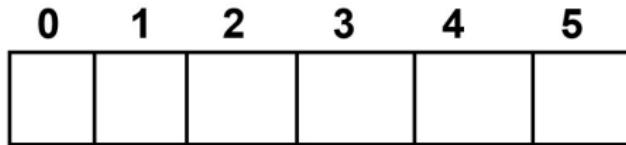


**file.tête=**

**file.queue=0**

**file.estVide() == vrai**

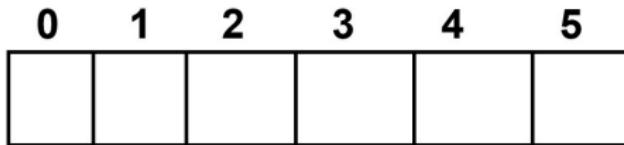
**file.enfiler(3)**



# Les files

Exemple

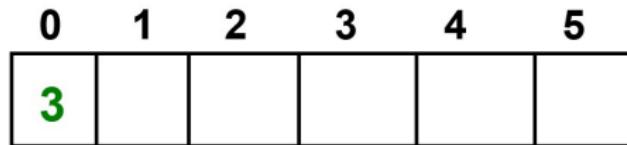
**file= new File(6)**



**file.tête=**  
**file.queue=0**

**file.estVide()==vrai**

**file.enfiler(3)**

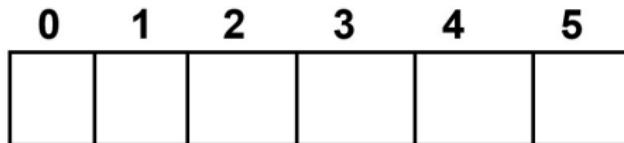


**file.tête= 0   file.queue= 1**

## Les files

## Exemple

**file= new File(6)**

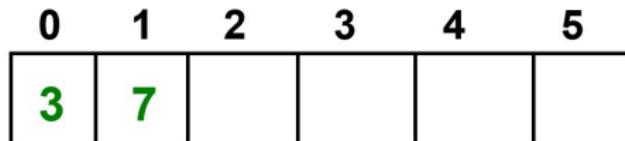


**file.tête=**  
**file.queue=0**

**file.estVide()==vrai**

### file.enfiler(3)

## file.enfiler(7)

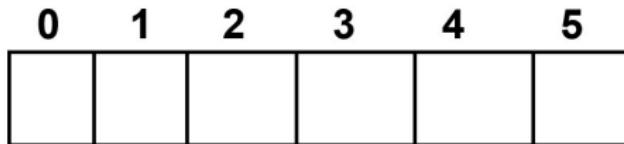


file.tête= 0 file.queue= 1  
2

# Les files

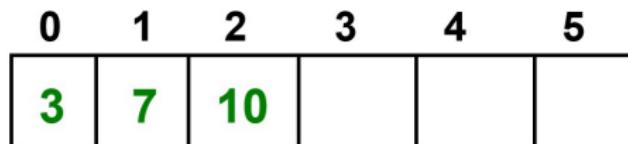
Exemple

**file = new File(6)**



**file.tête=**  
**file.queue=0**

**file.estVide()==vrai**



**file.enfiler(3)**

**file.enfiler(7)**

**file.enfiler(10)**

**file.tête= 0    file.queue= 1**

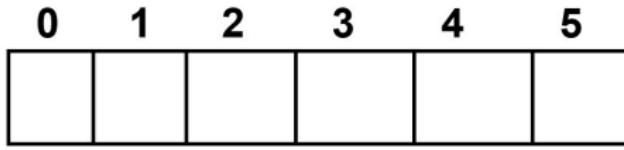
2

3

# Les files

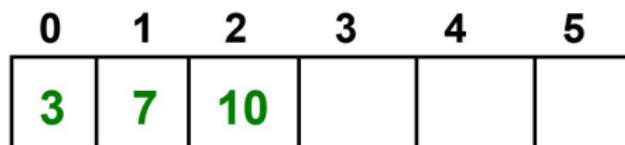
Exemple

`file = new File(6)`



`file.tête=`  
`file.queue=0`

`file.estVide() == vrai`



`file.enfiler(3)`

`file.enfiler(7)`

`file.enfiler(10)`

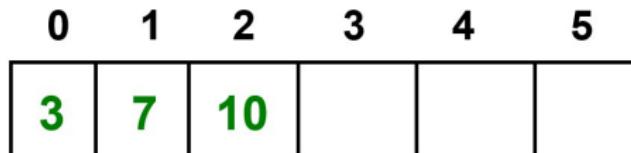
`file.tête= 0    file.queue= 1`

2  
3

`file.estVide( ) == faux`

# Les files

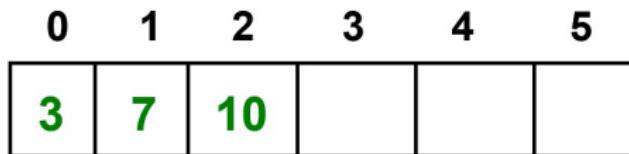
## Exemple



**file.tête=0    file.queue= 3**

# Les files

## Exemple

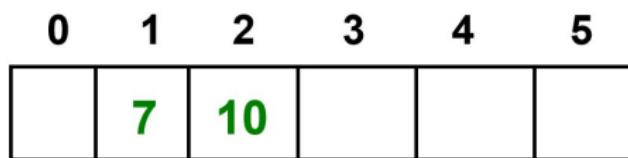


**x=file.defiler()**

**file.tête=0    file.queue= 3**

# Les files

Exemple



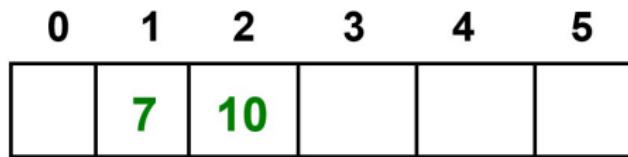
**x=file.defiler()**

**file.tête=0**   **file.queue= 3**

**1**

# Les files

## Exemple



**file.tête=0**   **file.queue=3**

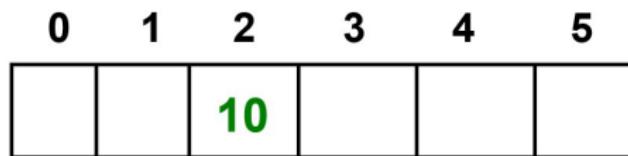
**1**

**x=file.defiler()**

**x=file.defiler()**

# Les files

## Exemple



**x=file.defiler()**  
**x=file.defiler()**

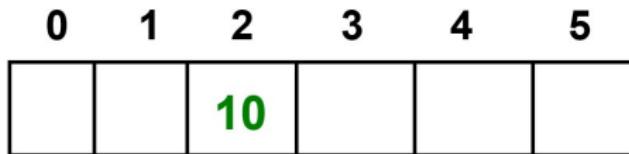
**file.tête=0**   **file.queue= 3**

1

2

# Les files

## Exemple



**file.tête=0**   **file.queue=3**

1

2

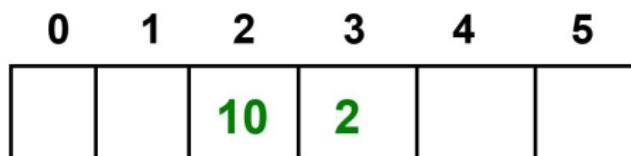
**x=file.defiler()**

**x=file.defiler()**

**file.enfiler(2)**

# Les files

Exemple



**file.tête=0**   **file.queue=3**

~~1~~

2

4

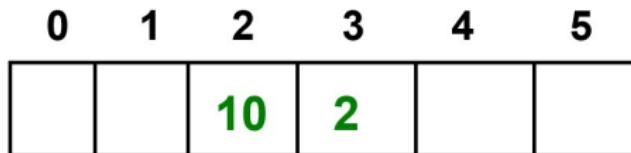
**x=file.defiler()**

**x=file.defiler()**

**file.enfiler(2)**

# Les files

## Exemple



**file.tête=0**   **file.queue=3**

~~1~~

2

4

**x=file.defiler()**

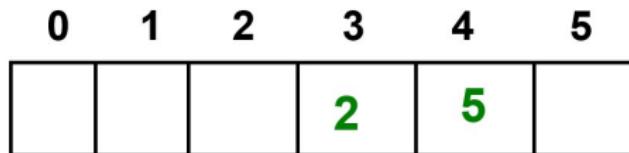
**x=file.defiler()**

**file.enfiler(2)**

**x=file.defiler()**

# Les files

## Exemple



**file.tête=0**   **file.queue=3**

~~1~~

2

4

**x=file.defiler()**

**x=file.defiler()**

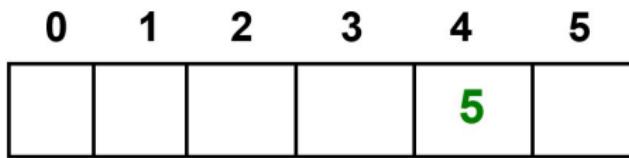
**file.enfiler(2)**

**x=file.defiler()**

**file.enfiler(5)**

# Les files

Exemple



**file.tête=0**   **file.queue=3**

~~x~~

2

4

**x=file.defiler()**

**x=file.defiler()**

**file.enfiler(2)**

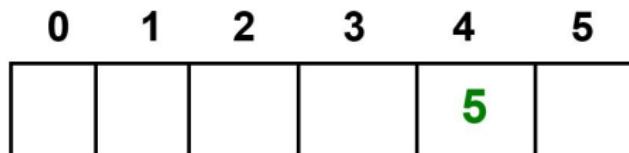
**x=file.defiler()**

**file.enfiler(5)**

**x=file.defiler()**

# Les files

Exemple



**file.tête=0**   **file.queue=3**

~~1~~

2

4

**file.tête=4**   **file.queue= 5**

**x=file.defiler()**

**x=file.defiler()**

**file.enfiler(2)**

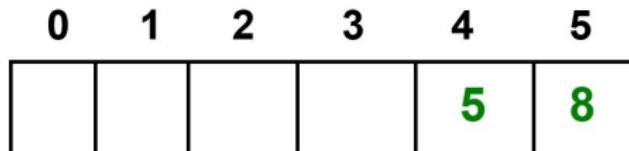
**x=file.defiler()**

**file.enfiler(5)**

**x=file.defiler()**

# Les files

## Exemple



file.tête=0 file.queue=3

x

2

4

file.tête=4 file.queue=5

x=file.defiler()

x=file.defiler()

file.enfiler(2)

x=file.defiler()

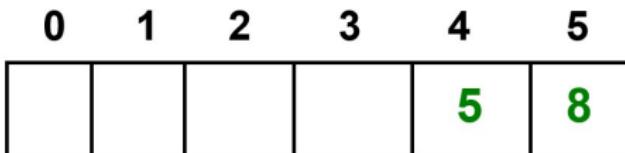
file.enfiler(5)

x=file.defiler()

file.enfiler(8)

# Les files

## Exemple



**file.tête=0    file.queue=3**

~~1~~

2

4

**file.tête=4    file.queue=5**

$$(5+1) \bmod 6 = 0$$

**x=file.defiler()**

**x=file.defiler()**

**file.enfiler(2)**

**x=file.defiler()**

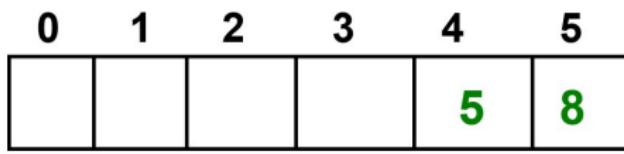
**file.enfiler(5)**

**x=file.defiler()**

**file.enfiler(8)**

# Les files

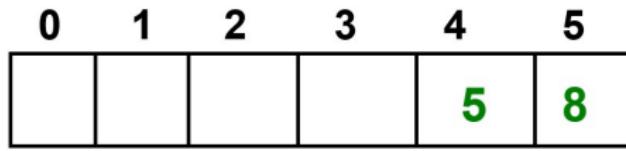
## Exemple



**file.tête=4  
file.queue=0**

# Les files

## Exemple

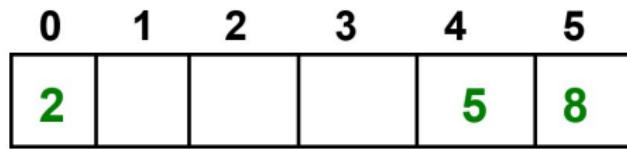


**file.tête=4  
file.queue=0**

**file.enfiler(2)**

# Les files

Exemple



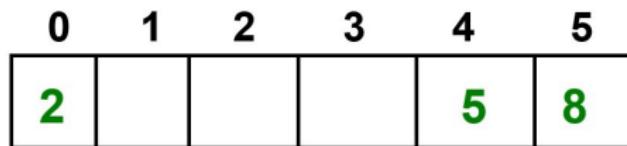
**file.tête=4  
file.queue=0**

**file.enfiler(2)**

**file.tête=4  
file.queue=1**

# Les files

## Exemple

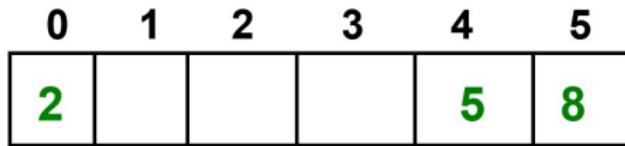


**file.tête= 4**

**file.queue= 1**

# Les files

Exemple



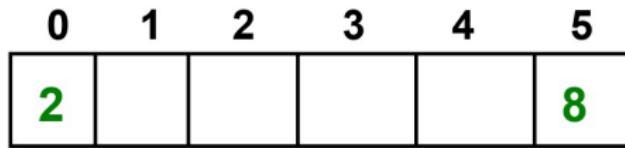
`x=file.defiler()`

`file.tête= 4`

`file.queue= 1`

# Les files

Exemple

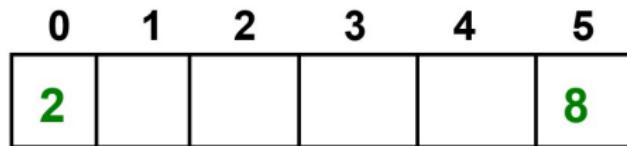


`x=file.defiler()`

`file.tête=` 5  
`file.queue=` 1

# Les files

## Exemple



**x=file.defiler()**

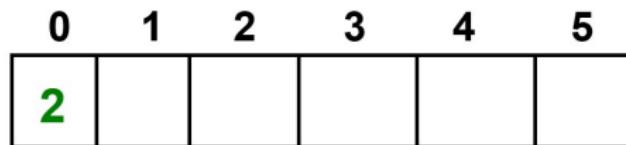
**x=file.defiler()**

**file.tête= 5**

**file.queue= 1**

# Les files

## Exemple



**x=file.defiler()**

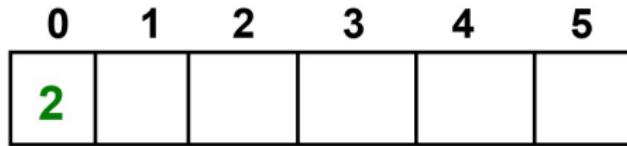
**x=file.defiler()**

~~file.tête=4, 5~~    $(5+1) \bmod 6 = 0$

**file.queue= 1**

# Les files

## Exemple



~~file.tête=4 8~~    $(5+1) \bmod 6 = 0$   
file.queue=1

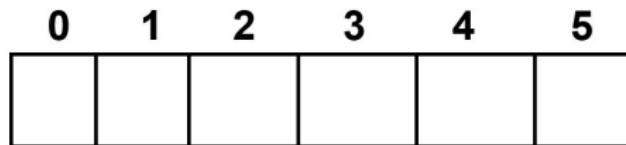
**x=file.defiler()**

**x=file.defiler()**

**x=file.defiler()**

# Les files

## Exemple



file.tête= ~~4~~ 5    (5+1) mod 6 = 0' 1  
file.queue= 1

x=file.defiler()

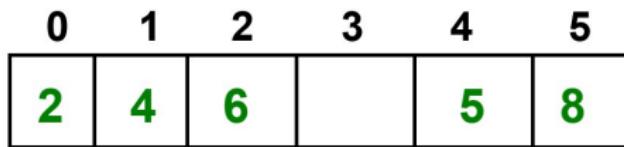
x=file.defiler()

x=file.defiler()

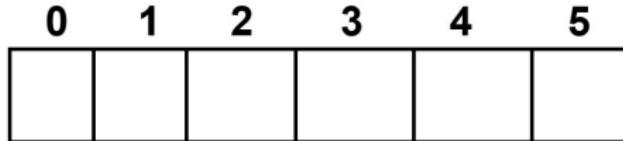
**File vide car tête= queue (= 1)**

# Les files

Exemple



**file.tête=4  
file.queue=3**



# Les files

Exemple

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 2 | 4 | 6 |   | 5 | 8 |

**file.tête=4  
file.queue=3**

**file.enfiler(3)**

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
|   |   |   |   |   |   |

# Les files

## Exemple

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 2 | 4 | 6 |   | 5 | 8 |

**file.tête=4  
file.queue=3**

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 2 | 4 | 6 | 3 | 5 | 8 |

**file.enfiler(3)**

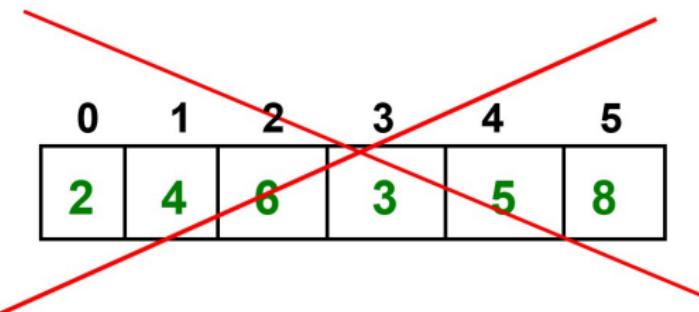
**file.tête=4  
file.queue=4**

# Les files

## Exemple

|   |   |   |   |   |   |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |
| 2 | 4 | 6 |   | 5 | 8 |

**file.tête=4  
file.queue=3**



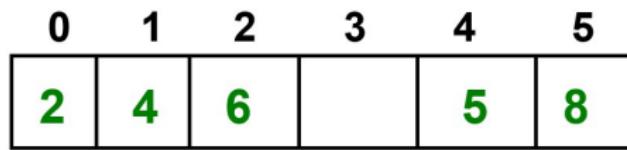
|   |   |   |   |   |   |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |
| 2 | 4 | 6 | 3 | 5 | 8 |

**file.enfiler(3)**

**file.tête=4  
file.queue=4**

# Les files

Exemple



file.tête=4  
file.queue=3

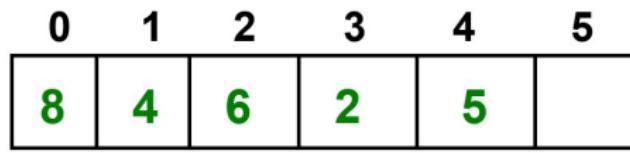
La file est pleine      file.enfiler(3) est interdit

$$\text{tête} = \text{queue} + 1$$

*une file est "pleine" si  
il reste une seule place non occupée dans le tableau*

# Les files

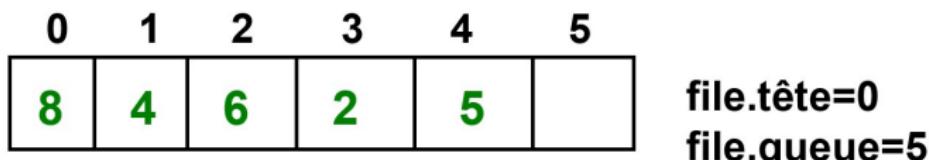
## Exemple



**file.tête=0  
file.queue=5**

# Les files

## Exemple

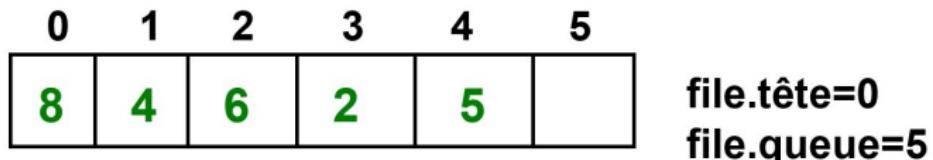


La file est pleine      file.enfiler(3) est interdit

$$\text{tête} = (\text{queue}+1) \bmod 6$$

# Les files

Exemple



La file est pleine      `file.enfiler(3)` est interdit

$$\text{tête} = (\text{queue}+1) \bmod 6$$

Remarque:

si  $\text{queue} < 5$  alors  $(\text{queue}+1) \bmod 6 = \text{queue} + 1$

si  $\text{queue} = 5$  alors  $(\text{queue}+1) \bmod 6 = 0$

# Une implémentation java des Files circulaires

- Une file est la juxtaposition d'un tableau avec trois entiers qui indiquent la taille, le début et la fin de la file
- On peut ajouter un élément “à la fin” (en queue) de la file si la file n'est pas pleine. On peut retirer l'élément “au début” (en tête) de la file si la file n'est pas vide
- Le seul élément de la file auquel on peut avoir accès est le plus ancien élément qui y a été placé (donc en tête)

# Une implémentation java des Files circulaires

```
1 public class File{  
2     private int[] tab;  
3     private int queue; // pointe sur case vide qui recevra  
4                         // le prochain element  
5     private int tete; // pointe sur element le plus ancien ds la file  
6     private int taille;  
7     // ne contiendra pas plus de taille-1 elements a la fois  
8  
9     public File( int t ){  
10        this.taille = t;  
11        tab = new int[taille];  
12        queue = 0;  
13        tete = 0;  
14    }  
}
```

# Une implémentation java des Files circulaires

```
15     public boolean estVide() {  
16         return (tete == queue) ;  
17     }  
18  
19     public boolean estPleine () {  
20         return ( ((queue+1) % taille) == tete ) ;  
21     }
```

# Une implémentation java des Files circulaires

```
22     public void enfile(int x) throws FileException {  
23         if (this.estPleine()) throw new FileException();  
24         tab[queue] = x;  
25         queue = (queue+1) % taille;  
26     }  
27  
28     public int defiler() throws FileException {  
29         if (this.estVide()) throw new FileException();  
30         int x = tab[tete];  
31         tete = (tete + 1) % taille ;  
32         return x;  
33     }
```

# Une implémentation java des Files circulaires

```
34 public void affichage() {  
35     System.out.println("debut affichage");  
36     int i = tete;  
37     while (i != queue){  
38         System.out.println (tab[i]);  
39         i = (i+1) %taille;  
40     }  
41     System.out.println("fin affichage");  
42 }  
43 }
```

## File circulaire en java – Que fait ce programme ? (1/2)

```
1 import java.io.*;
2 public class TestFile {
3     static BufferedReader in =
4         new BufferedReader(new InputStreamReader(System.in));
5     public static void main (String[] args) {
6         File F=new File("6");
7         try {
8             if (!F.estPleine()) F.enfiler(10);
9             else System.out.println ("---file pleine");
10            F.affichage();
11            if (!F.estPleine()) F.enfiler(20);
12            else System.out.println ("---file pleine");
13            if (!F.estPleine()) F.enfiler(30);
14            else System.out.println ("---file pleine");
15            if (!F.estPleine()) F.enfiler(40);
16            else System.out.println ("---file pleine");
17            if (!F.estPleine()) F.enfiler(50);
18            else System.out.println ("---file pleine");
19            F.affichage();
```

## File circulaire en java – Que fait ce programme ? (2/2)

```
20     System.out.println(x + " defile");
21     F.affichage();
22     x = F.defiler();
23     System.out.println(x + " defile");
24     F.affichage();
25     if (!F.estPleine()) F.enfiler(60);
26     else System.out.println("---file pleine");
27     F.affichage();
28     if (!F.estPleine()) F.enfiler(70);
29     else System.out.println ("---file pleine");
30     F.affichage();
31     if (!F.estPleine()) F.enfiler(80);
32     else System.out.println ("---file pleine");
33     F.affichage();
34 }
35 catch (Exception e) {
36     System.out.print("Problème");
37 }
38 }
39 }
```



# Conclusion

- Nous avons revu des structures de données “linéaires” : tableau, liste
- D'autres structures linéaires : piles et files
- Nous avons illustré la notion de type abstrait
  - ▶ À travers l'implémentation contiguë ou chaînée d'une liste
  - ▶ En “cachant” la façon dont est vraiment représentée notre structure de données