# NFP136 – VARI 2

# Algorithmique et Structures de données

Prolongement de la partie Algorithmique et Programmation du cours NFP135- Vari1

# Objectifs de cette partie 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 soucis d'abstraction : dissocier la structure de données de la façon dont elle est implémentée ⇒ notion de *Type abstrait*

# Contenu :

- 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

# Contenu :

⇒ 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

# Cours 1
# STRUCTURES DE DONNEES ELEMENTAIRES

# Les tableaux

# Tableau

- 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 (indicées) par un entier.

# Tableau

- 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- rappels

- Il existe un type tableau prédéfini en java

- Contrairement aux types primitifs (boolean, byte, int, double, …) une variable de type tableau :
  - Est un objet
  - Sa valeur est une référence

# Tableau- en java- rappels

- 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(T[i]);
    ```
    ```

# Les listes

# Liste- définition générale

- 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

- Liste à laquelle on peut appliquer les 4 opérations suivantes :
  - 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)

# Le type abstrait Liste

- On peut dès lors écrire des algorithmes basés sur ces 4 opérations comme par exemple le calcul de la taille d'une liste L

```
Algo : taille

Entrée : une liste L

Sortie : un entier n qui représente le nombre d'éléments
de la liste

n=0;

liste p=L;

Tant que p n'est pas vide faire

  n=n+1;

  p= queue(p);

Fait;
```

# Le type abstrait Liste

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

```
Algo : appartient
Entrée : une liste L et une donnée x
Sortie : un booléen r égal à vrai ssi x appartient à L
r= faux;
liste p=L;
Tant que p n'est pas vide faire
  si (tete(p)= x) alors r= vrai; fsi;
  p= queue(p);
Fait;
```

- Etc …

# Implémentations d'une Liste

Deux façons de faire :

- **Listes chaînées**, où les différents éléments de la liste sont crées au fur et à mesure des besoins (dynamique)

- **Listes contigües**, où la liste est grosso-modo représentée pas 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

Révision de la notion de liste vue en NFP135-Vari1

# Liste chaînée en java

- 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

# Liste chaînée en java- début

```
class Liste{
   private int valeur;
   private Liste suivant;

   public Liste(int premier, Liste reste)
   {
      valeur = premier;
      suivant = reste;
    }
```

# Liste chaînée en java- fin

```
    public int tete (){
        return (this.valeur);
     }

    public Liste queue (){
        return (this.suivant);
     }
}//fin class
```

# utilisation dans un programme

```
import java.io.*;
public class testListe {
     static BufferedReader in =
        new BufferedReader(new InputStreamReader(System.in));
     public static void main (String[] args) {
        Liste L= null; int n;
        try {
           System.out.print("Combien d'elements avez-vous dans la liste?");
          n=Integer.parseInt(in.readLine());
          for (int i=1; i<= n; i++) L= new Liste (i, L);
           //affichage
          Liste p=L;
          while(p != null){
               System.out.println( p.tete() + " \n");
               p = p.queue();
          }
        }
        catch (IOException e) { System.out.print("Probleme");
}}}
```

# Une implémentation java de liste contiguë

# Liste contiguë en java

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

# Liste contiguë en java- début

```
class tabListe{
    private int[] tab;
    private int indiceFin;

    public tabListe(int premier, tabListe reste){
      if (reste==null) {
        tab=new int[100];
        tab[0]=premier;
        indiceFin=0;
      } else {
        tab=reste.tab;
        indiceFin=reste.indiceFin + 1;
        tab[indiceFin]=premier;
       }
    }
```

# Liste contiguë en java- fin

```
    public int tete (){
      return (this.tab[this.indiceFin]);
    }

    public tabListe queue (){
      if (this.indiceFin==0) //un seul element
          return (null);
      tabListe L=this;
      L.tab=this.tab;
      L.indiceFin=this.indiceFin-1;
      return (L);
    }
}//fin class
```

# utilisation dans un programme

```
import java.io.*;
public class testtabListe {
    static BufferedReader in =
              new BufferedReader(new InputStreamReader(System.in))
    public static void main (String[] args) {
       tabListe L= null; int n;
       try {
          System.out.print("Combien d'elements avez-vous dans la liste?");
          n=Integer.parseInt(in.readLine());
          for (int i=1; i<= n; i++) L= new tabListe (i, L);
           //affichage
          tabListe p=L;
          while(p != null){
               System.out.println( p.tete() + " \n");
               p = p.queue();
          }
       }
       catch (IOException e) { System.out.print("Probleme");
}}}
```

# Conclusion sur nos deux implémentations de Liste

- Clairement transparent au programmeur comme à l'utilisateur du programme

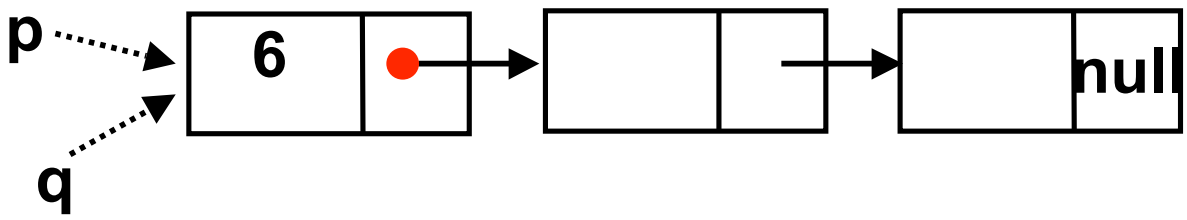
- **Mais**, dans l'implémentation par un tableau, nous avons « ignoré » des problèmes importants :
  - Initialisation à 100 de la taille du tableau, pourquoi pas une autre taille ?
  - Arrivée à la fin du tableau (liste pleine) Notre choix de taille maximale peut être sous-dimensionné ou sur-dimensionné suivant l'usage
  - Il est bien sûr possible de tenir compte de ces problèmes dans une implémentation plus réaliste et plus fine

# Conclusion sur nos deux implémentations de Liste

- 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**

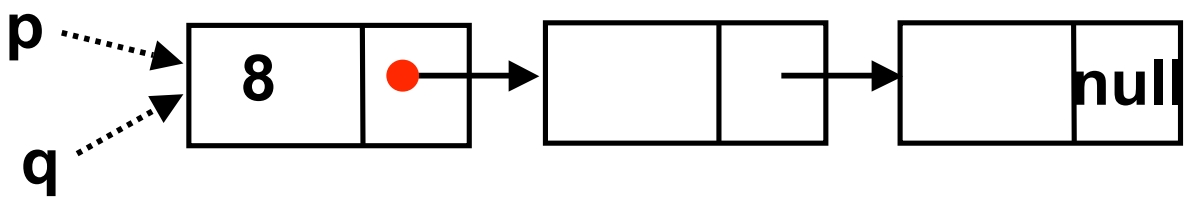
# Quelques rappels sur les références et méthodes supplémentaires pour la liste chaînée




**Liste p, q;** //références sur une « cellule»

**p.valeur=6;**

**q=p;** //alors q.valeur=6 et q.suivant=p.suivant




**Liste p, q;** //références sur une « cellule »

**p.valeur=6;**

**q=p;** //alors q.valeur=6 et q.suivant=p.suivant

**q.valeur=8;** //alors p.valeur=8 aussi

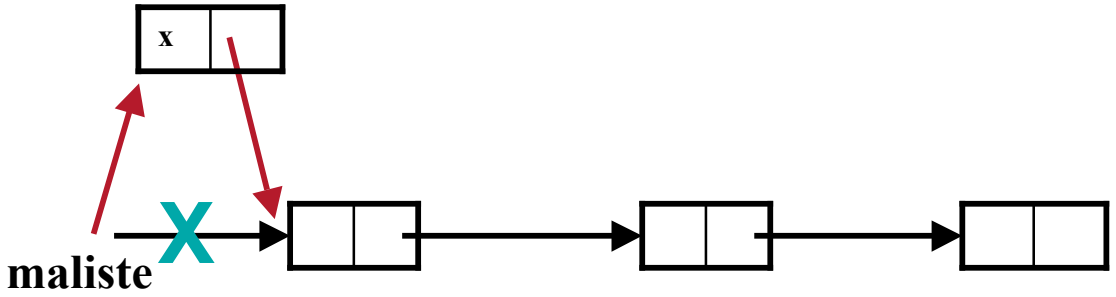
**p = null;** // « vide » la liste p

# LES METHODES

```
public Liste insererentete(int x){
  Liste L =new Liste(x, this);
  return L;
}
```

Utilisation dans un programme :

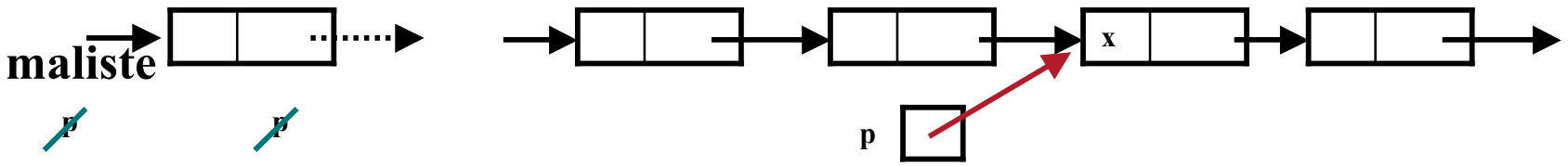
```
maliste = maliste.insererentete(x)
```



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

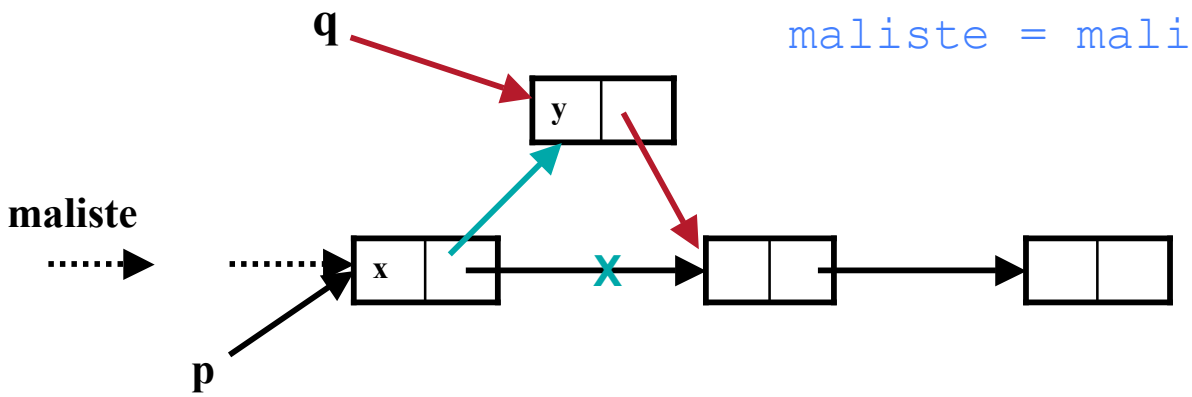
Utilisation dans un programme :

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



# LES METHODES

```
public Liste insererapres (int x, int y){
  //ajoute y apres x si x est present
  Liste p=this;
  while(p != null){
      if(p.valeur != x) {
            p=p.suivant;
      } else {// x est trouve on insere y juste apres
            Liste q = new Liste(y, p.suivant);
            p.suivant=q;
            return this;          }
  }
  return this;
}
```

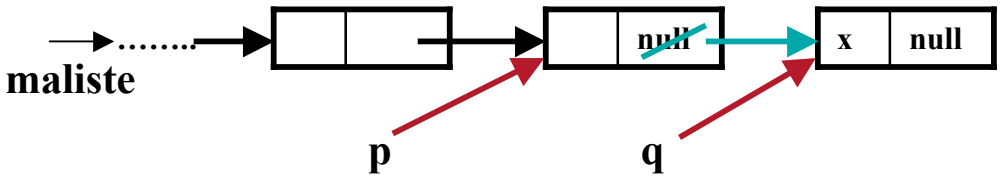



```
maliste = maliste.insererapres(x,y)
```

# LES METHODES

```
public Liste insererenqueue (int x){
   //insere  x en queue de liste
   Liste p=this;
   Liste q = new Liste (x, null);
   if (p == null){
       p=q;
   } else {// on va a la fin de la liste
       while(p.suivant != null){
                   p=p.suivant;
       }
       p.suivant=q;
   }
   return this;
}
```

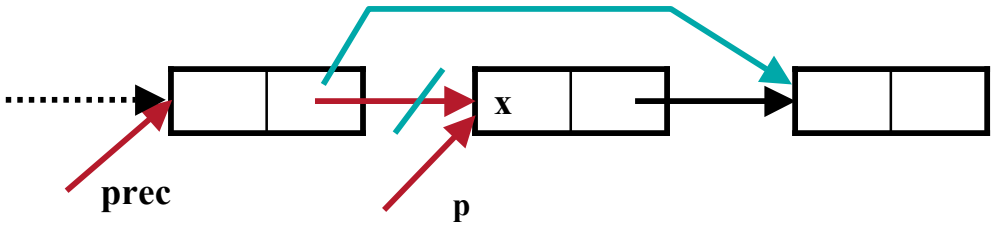
maliste = maliste.insererenqueue(x)



# LES METHODES

```
public Liste supprimer (int x){
   Liste p=this; //reference cellule courante
   Liste prec = null; //predecesseur de p
   while(p != null){
        if (p.valeur != x) {
                prec=p;
                p=p.suivant;
        } else { //x trouve
                if (prec==null) {//premier elem de la liste
                        return (p.suivant);
                } else {
                        prec.suivant=p.suivant;
                        return this;
                }
        }
   }
   return this;
}
```

# LES METHODES




```
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

# Les Piles

# Pile - Définition

- Une pile est une structure de données où les insertions et les suppressions se font toujours du même coté
  - LIFO (Last-in First out)

- L'insertion s'appelle empiler, la suppression dépiler et on doit pouvoir tester si la pile est vide. Il est de plus pratique de pouvoir lire le sommet de la pile

# Une implémentation java de pile

# Pile en java

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

# Pile en java

```
class Pile {
   private Liste L;

   public Pile() {
        L = null;
   }
   public boolean estVide(){
        return (this.L==null);
   }
   public void empiler( int a) {
        this.L = new Liste(a, this.L);
   }
   public void depiler() {
        this.L = this.L.queue();
   }
   public int sommet(){
        return (this.L.tete());
   }
}
```

# 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

```
import java.io.*;
public class testPile {
    static BufferedReader in =
              new BufferedReader(new InputStreamReader(System.in));
    public static void main (String[] args) {
        Pile P=new Pile ();
        try {
          for (int i=0; i< args.length; i++) {
                 if (args[i].equals("(")) P.empiler(0);
                if (args[i].equals(")")) P.depiler();
           }
           if (P.estVide())
              System.out.print("Expression bien parenthesee");
           else  System.out.print("Expression mal parenthesee");
        }
        catch (Exception e) {
          System.out.print("Probleme, expression mal parenthesee");
}}}
```

# Utilisation pour la vérification des parenthèses

## Exemples d'utilisation

```
java testPile a ( b ) ( ( c ) )
Expression bien parenthesee

java testPile a b cc ( ( (
Expression mal parenthesee

java testPile a ( ) f )
Probleme, expression mal parenthesee
```

# Les Files

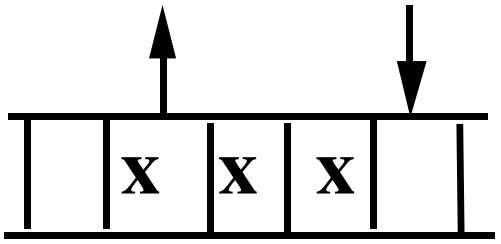
# File - Définition

- Une pile est une structure de données où les insertions se font d'un coté et les suppressions se font de l'autre coté
  - FIFO (Fast-in First out)

# File - Définition

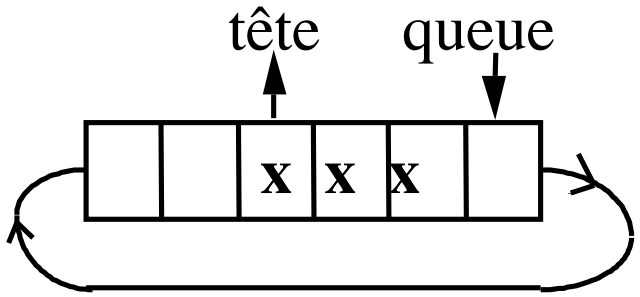
insérer = enfiler

supprimer = défiler




- Exemples: file d'attente à un guichet, tampon de messages

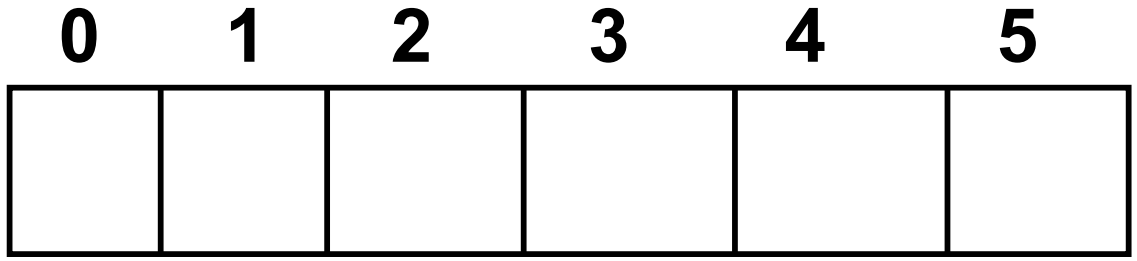
- En général: **File circulaire**



| **méthodes** | **conditions** |
| --- | --- |
| **enfiler(Elt x)** | **file non pleine**<br>**dépôt en queue** |
| **defiler( )** | **file non vide**<br>**retrait en tête** |
| **estvide( )** | **retourne booléen** |
| **estpleine( )** | **retourne booléen** |
| *file vide:* | *tête == queue* |
| *file pleine:* | *tête == (queue+1) mod taille* |

# EXEMPLE

file= new File(6)



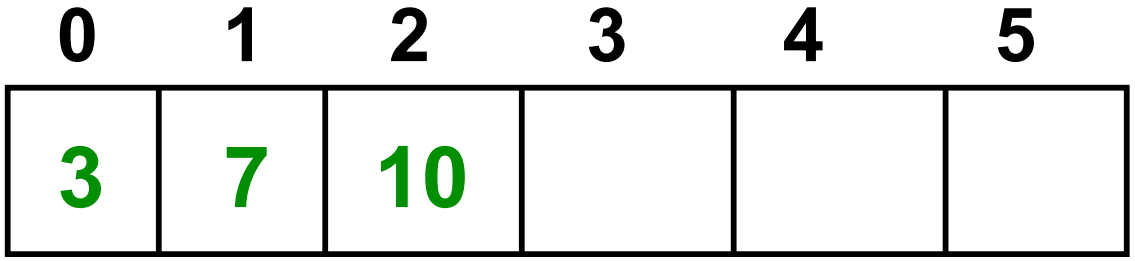

file.tête=
file.queue=0

file.estvide=vrai

file.enfiler(3)

file.enfiler(7)

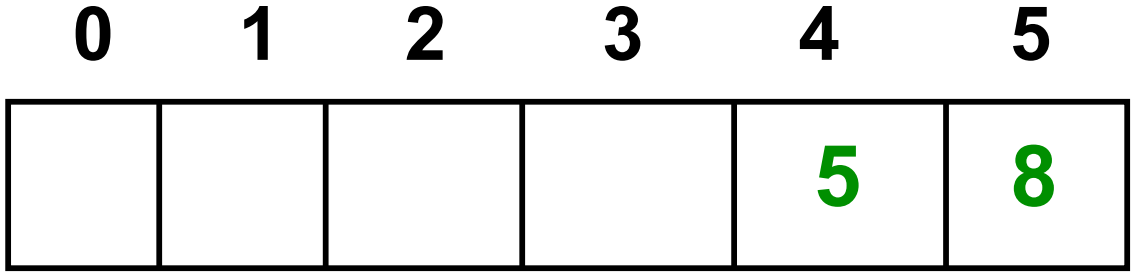
file.enfiler(10)




file.tête= 0 file.queue= ~~1~~ ~~2~~ 3

file.estvide( )=faux

# *EXEMPLE*




file.tête=0 file.queue=3

1

2

4

file.tête=4 file.queue=5

(5+1) mod 6 = 0

x=file.defiler()

x=file.defiler()

file.enfiler(2)

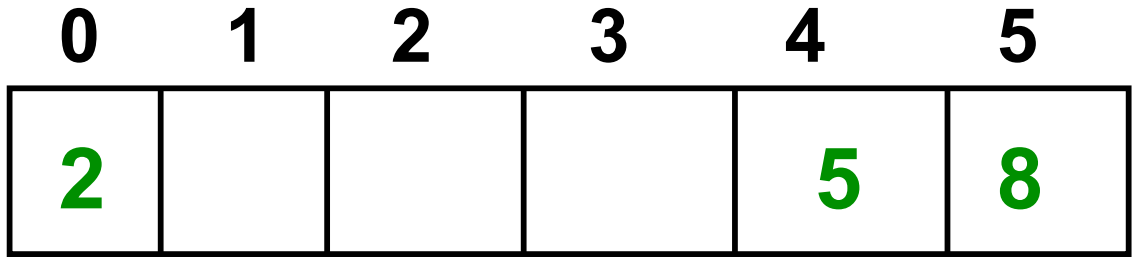
x=file.defiler()

file.enfiler(5)

x=file.defiler()

file.enfiler(8)

# EXEMPLE

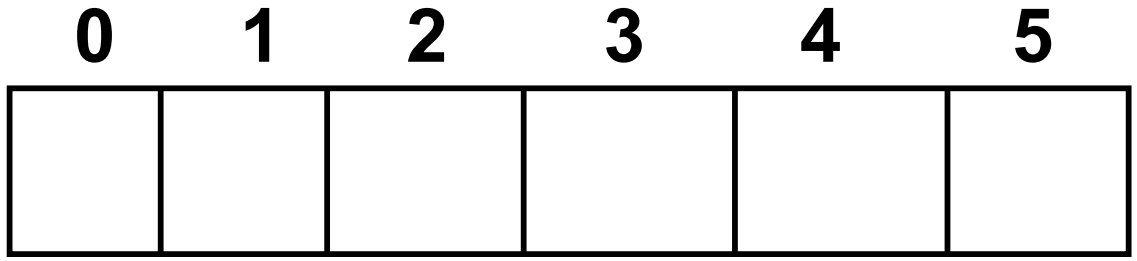



file.tête=4
file.queue=0

file.enfiler(2)

file.tête=4
file.queue=1

# EXEMPLE




x=file.defiler()

x=file.defiler()

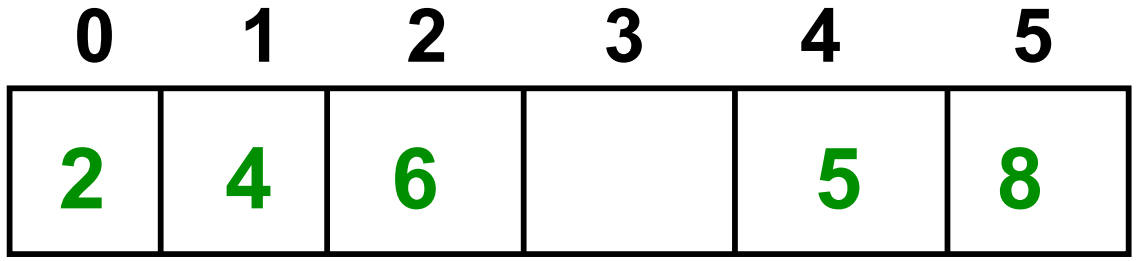
file.tête=~~4~~ ~~5~~ (5+1) mod6=~~0~~ 1

x=file.defiler()

file.queue= 1

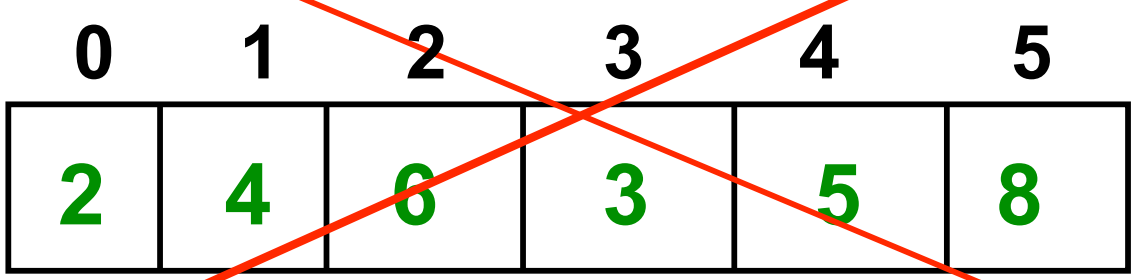
**File vide car tête= queue (= 1)**

# *EXEMPLE*



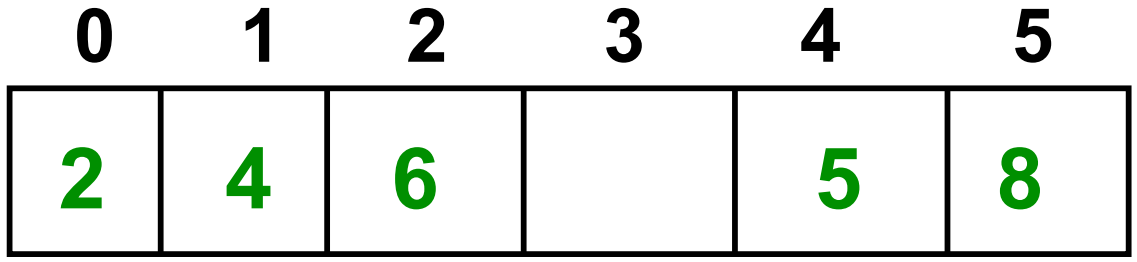

**file.tête=4**
**file.queue=3**

**file.enfiler(3)**




**file.tête=4**
**file.queue=4**

# EXEMPLE



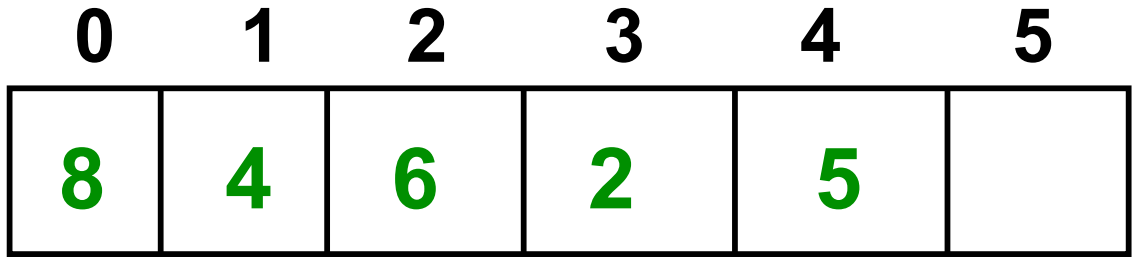

file.tête=4
file.queue=3

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

**tête = queue+1**

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

# EXEMPLE




file.tête=0
file.queue=5

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

tête = (queue+1) mod 6

**Remarque:**
si queue<5 alors (queue+1) mod 6 = queue + 1
si queue=5 alors (queue+1) mod 6 = 0

# Une implémentation java de File circulaire

# File circulaire en java

- 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" (queue) de la file si la file n'est pas pleine
- On peut retirer l'élément "au début" (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)

# File circulaire en java

```
public class File{
   private int[] tab;
   private int queue; //pointe sur case vide qui recevra
                      //le prochain element
   private int tete; //pointe sur element le plus ancien ds la
   file
   private int taille;
   // ne contiendra pas plus de taille-1 elements a la fois

public File( int t ){
   this.taille = t;
   tab = new int[taille];
   queue = 0;
   tete = 0;
}
```

# File circulaire en java

```
public boolean estVide () {
  return ( tete==queue ) ;
}

public boolean estPleine () {
  return ( ((queue+1) % taille) == tete ) ;
}
```

# File circulaire en java

```
public void enfiler( int  x )throws FileException{
   if ( this.estPleine()) throw new FileException ();
   tab[queue] = x;
   queue=(queue+1) % taille;
}

public int defiler()throws FileException{
   if ( this.estVide()) throw new FileException();
   int x = tab[tete];
   tete = (tete + 1) % taille ;
   return x;
}
```

# File circulaire en java

```
public void affichage(){
  System.out.println("debut affichage");
  int i=tete;
  while (i != queue){
     System.out.println (tab[i]);
     i=(i+1) %taille;
  }
  System.out.println("fin affichage");
}
```

# File circulaire en java- Que fait ce programme ?

```
import java.io.*;
public class testFile {
    static BufferedReader in =
              new BufferedReader(new InputStreamReader(System.in));
    public static void main (String[] args) {
         File F=new File (6);
         try {
                    if (!F.estPleine()) F.enfiler(10);
                        else System.out.println ("---file pleine");
                   F.affichage();
                    if (!F.estPleine()) F.enfiler(20);
                        else System.out.println ("---file pleine");
                    if (!F.estPleine()) F.enfiler(30);
                         else System.out.println ("---file pleine");
                   if (!F.estPleine()) F.enfiler(40);
                        else System.out.println ("---file pleine");
                    if (!F.estPleine()) F.enfiler(50);
                        else System.out.println ("---file pleine");
                   F.affichage();
```

# File circulaire en java- Que fait ce programme ?

```
                int x=F.defiler();
                System.out.println (x+" defile");
                 F.affichage();
                x=F.defiler();
                System.out.println (x+" defile");
                 F.affichage();
                  if (!F.estPleine()) F.enfiler(60);
                      else System.out.println ("---file pleine");
                 F.affichage();
                  if (!F.estPleine()) F.enfiler(70);
                      else System.out.println ("---file pleine");
                 F.affichage();
                  if (!F.estPleine()) F.enfiler(80);
                      else System.out.println ("---file pleine");
                 F.affichage();

            }
                catch (Exception e) {
                 System.out.print("Probleme");
        }
    }
}
```

# 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
  - A travers l'implémentation contiguë ou chaînée d'une liste
  - En « cachant » la façon dont est vraiment représentée notre structure de données