

# COMPLEXITÉ ALGORITHMIQUE & STRUCTURES DE DONNÉES

David Auger, Jérémie Cabessa

Laboratoire DAVID, UVSQ

# INTRODUCTION

- ▶ On désire mesurer la complexité d'exécution d'un algorithme.
- ▶ On s'intéresse à la complexité en temps: le temps ou nombre d'opérations  $f(n)$  qu'effectue un algorithme sur un input de taille  $n$ .
- ▶ On laisse de côté la complexité en espace.
- ▶ Pour quantifier la *borne supérieure asymptotique* d'un algorithme, on utilise la notation "grand  $O$ " (big  $O$ ).

# INTRODUCTION

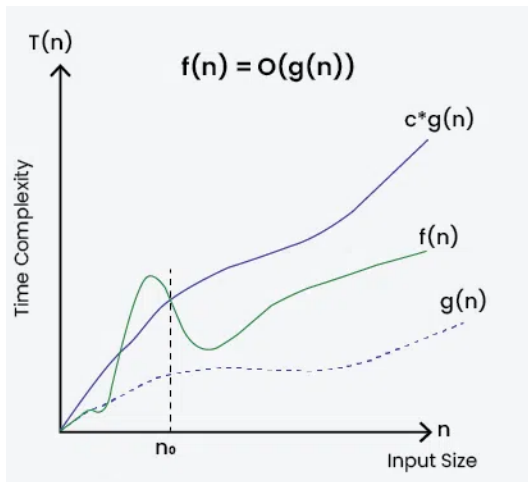
- ▶ On désire mesurer la complexité d'exécution d'un algorithme.
- ▶ On s'intéresse à la complexité en temps: le temps ou nombre d'opérations  $f(n)$  qu'effectue un algorithme sur un input de taille  $n$ .
- ▶ On laisse de côté la complexité en espace.
- ▶ Pour quantifier la *borne supérieure asymptotique* d'un algorithme, on utilise la notation "grand  $O$ " (big  $O$ ).

# INTRODUCTION

- ▶ On désire mesurer la complexité d'exécution d'un algorithme.
- ▶ On s'intéresse à la complexité en temps: le temps ou nombre d'opérations  $f(n)$  qu'effectue un algorithme sur un input de taille  $n$ .
- ▶ On laisse de côté la complexité en espace.
- ▶ Pour quantifier la *borne supérieure asymptotique* d'un algorithme, on utilise la notation "grand  $O$ " (big  $O$ ).

# INTRODUCTION

- ▶ On désire mesurer la complexité d'exécution d'un algorithme.
- ▶ On s'intéresse à la complexité en temps: le temps ou nombre d'opérations  $f(n)$  qu'effectue un algorithme sur un input de taille  $n$ .
- ▶ On laisse de côté la complexité en espace.
- ▶ Pour quantifier la *borne supérieure asymptotique* d'un algorithme, on utilise la notation “grand  $O$ ” (big  $O$ ).

GRAND  $O$ 

GRAND  $O$ DEFINITION (BIG  $O$  NOTATION)

Soient  $f : \mathbb{N} \rightarrow \mathbb{N}$  et  $g : \mathbb{N} \rightarrow \mathbb{N}$  deux fonctions. On dit que

- ▶  $f$  est dominée par  $g$
- ▶  $g$  domine par  $f$
- ▶  $f \in O(g)$

ssi il existe  $C > 0$  et  $N_0 \in \mathbb{N}$  tel que

$$f(n) \leq C \cdot g(n), \text{ pour tout } n \geq N_0$$

GRAND  $O$ DEFINITION (BIG  $O$  NOTATION)

Soient  $f : \mathbb{N} \rightarrow \mathbb{N}$  et  $g : \mathbb{N} \rightarrow \mathbb{N}$  deux fonctions. On dit que

- ▶  $f$  est dominée par  $g$
- ▶  $g$  domine par  $f$
- ▶  $f \in O(g)$

ssi il existe  $C > 0$  et  $N_0 \in \mathbb{N}$  tel que

$$f(n) \leq C \cdot g(n), \text{ pour tout } n \geq N_0$$



GRAND  $O$ DEFINITION (BIG  $O$  NOTATION)

Soient  $f : \mathbb{N} \rightarrow \mathbb{N}$  et  $g : \mathbb{N} \rightarrow \mathbb{N}$  deux fonctions. On dit que

- ▶  $f$  est dominée par  $g$
- ▶  $g$  domine par  $f$
- ▶  $f \in O(g)$

ssi il existe  $C > 0$  et  $N_0 \in \mathbb{N}$  tel que

$$f(n) \leq C \cdot g(n), \text{ pour tout } n \geq N_0$$

GRAND  $O$ DEFINITION (BIG  $O$  NOTATION)

Soient  $f : \mathbb{N} \rightarrow \mathbb{N}$  et  $g : \mathbb{N} \rightarrow \mathbb{N}$  deux fonctions. On dit que

- ▶  $f$  est dominée par  $g$
- ▶  $g$  domine par  $f$
- ▶  $f \in O(g)$

ssi il existe  $C > 0$  et  $N_0 \in \mathbb{N}$  tel que

$$f(n) \leq C \cdot g(n), \text{ pour tout } n \geq N_0$$

GRAND  $O$ DEFINITION (BIG  $O$  NOTATION)

Soient  $f : \mathbb{N} \rightarrow \mathbb{N}$  et  $g : \mathbb{N} \rightarrow \mathbb{N}$  deux fonctions. On dit que

- ▶  $f$  est dominée par  $g$
- ▶  $g$  domine par  $f$
- ▶  $f \in O(g)$

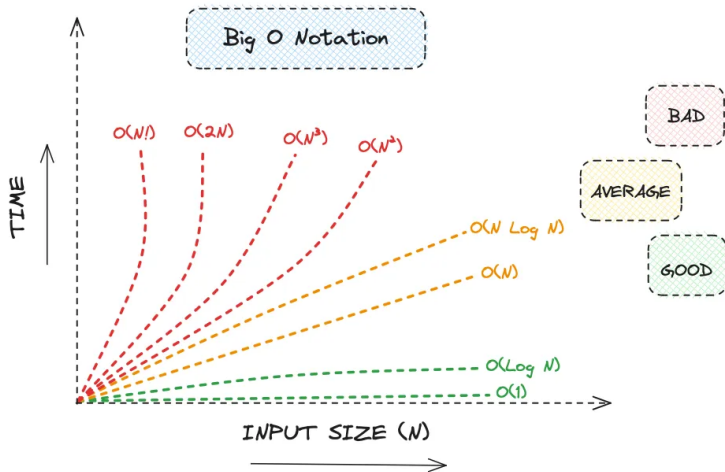
ssi il existe  $C > 0$  et  $N_0 \in \mathbb{N}$  tel que

$$f(n) \leq C \cdot g(n), \text{ pour tout } n \geq N_0$$

GRAND  $O$ 

Complexité	Terminologie
$O(1)$	constant
$O(\log(n))$	logarithmique
$O(\sqrt{n})$	...
$O(n)$	linéaire
$O(n \log(n))$	linéaire-logarithmique
$O(n^k)$	polynomial
$O(c^n)$	exponentiel
$O(n!)$	factoriel
$O(n^n)$	...

## A



GRAND  $O$ 

## EXAMPLE

Soit  $f : \mathbb{N} \rightarrow \mathbb{N}$  définie par

$$f(n) = 3n^2 + 6n - 100$$

On montre que  $f(n) \in O(n^2)$ . Fixons  $C = 4$  et  $N_0 = 0$ . On a alors:

$$\begin{aligned} 3n^2 + 6n - 100 &\leq Cn^2 && \text{pour tout } n \geq N_0 \\ \Leftrightarrow 3n^2 + 6n - 100 &\leq 4n^2 && \text{pour tout } n \geq 0 \\ \Leftrightarrow 3n^2 + 6n - 100 &\leq 3n^2 + n^2 && \text{pour tout } n \geq 0 \\ \Leftrightarrow n^2 - 6n + 100 &\leq 0 && \text{pour tout } n \geq 0 \end{aligned}$$

qui est toujours valable. On a donc bien que  $f(n) \in O(n^2)$ .

GRAND  $O$ 

## EXAMPLE

Soit  $f : \mathbb{N} \rightarrow \mathbb{N}$  définie par

$$f(n) = 3n^2 + 6n - 100$$

On montre que  $f(n) \in O(n^2)$ . Fixons  $C = 4$  et  $N_0 = 0$ . On a alors:

$$\begin{aligned} 3n^2 + 6n - 100 &\leq Cn^2 && \text{pour tout } n \geq N_0 \\ \Leftrightarrow 3n^2 + 6n - 100 &\leq 4n^2 && \text{pour tout } n \geq 0 \\ \Leftrightarrow 3n^2 + 6n - 100 &\leq 3n^2 + n^2 && \text{pour tout } n \geq 0 \\ \Leftrightarrow n^2 - 6n + 100 &\leq 0 && \text{pour tout } n \geq 0 \end{aligned}$$

qui est toujours valable. On a donc bien que  $f(n) \in O(n^2)$ .

GRAND  $O$ 

## EXAMPLE

Soit  $f : \mathbb{N} \rightarrow \mathbb{N}$  définie par

$$f(n) = 3n^2 + 6n - 100$$

On montre que  $f(n) \in O(n^2)$ . Fixons  $C = 4$  et  $N_0 = 0$ . On a alors:

$$\begin{aligned} 3n^2 + 6n - 100 &\leq Cn^2 && \text{pour tout } n \geq N_0 \\ \Leftrightarrow 3n^2 + 6n - 100 &\leq 4n^2 && \text{pour tout } n \geq 0 \\ \Leftrightarrow 3n^2 + 6n - 100 &\leq 3n^2 + n^2 && \text{pour tout } n \geq 0 \\ \Leftrightarrow n^2 - 6n + 100 &\leq 0 && \text{pour tout } n \geq 0 \end{aligned}$$

qui est toujours valable. On a donc bien que  $f(n) \in O(n^2)$ .



GRAND  $O$ 

## EXAMPLE

Soit  $f : \mathbb{N} \rightarrow \mathbb{N}$  définie par

$$f(n) = 3n^2 + 6n - 100$$

On montre que  $f(n) \in O(n^2)$ . Fixons  $C = 4$  et  $N_0 = 0$ . On a alors:

$$\begin{aligned} 3n^2 + 6n - 100 &\leq Cn^2 && \text{pour tout } n \geq N_0 \\ \Leftrightarrow 3n^2 + 6n - 100 &\leq 4n^2 && \text{pour tout } n \geq 0 \\ \Leftrightarrow 3n^2 + 6n - 100 &\leq 3n^2 + n^2 && \text{pour tout } n \geq 0 \\ \Leftrightarrow n^2 - 6n + 100 &\leq 0 && \text{pour tout } n \geq 0 \end{aligned}$$

qui est toujours valable. On a donc bien que  $f(n) \in O(n^2)$ .

GRAND  $O$ 

## EXAMPLE

Soit  $f : \mathbb{N} \rightarrow \mathbb{N}$  définie par

$$f(n) = 3n^2 + 6n - 100$$

On montre que  $f(n) \in O(n^2)$ . Fixons  $C = 4$  et  $N_0 = 0$ . On a alors:

$$\begin{aligned} 3n^2 + 6n - 100 &\leq Cn^2 && \text{pour tout } n \geq N_0 \\ \Leftrightarrow 3n^2 + 6n - 100 &\leq 4n^2 && \text{pour tout } n \geq 0 \\ \Leftrightarrow 3n^2 + 6n - 100 &\leq 3n^2 + n^2 && \text{pour tout } n \geq 0 \\ \Leftrightarrow n^2 - 6n + 100 &\leq 0 && \text{pour tout } n \geq 0 \end{aligned}$$

qui est toujours valable. On a donc bien que  $f(n) \in O(n^2)$ .

GRAND  $O$ 

## EXAMPLE

Soit  $f : \mathbb{N} \rightarrow \mathbb{N}$  définie par

$$f(n) = 3n^2 + 6n - 100$$

On montre que  $f(n) \in O(n^2)$ . Fixons  $C = 4$  et  $N_0 = 0$ . On a alors:

$$\begin{array}{ll} 3n^2 + 6n - 100 \leq Cn^2 & \text{pour tout } n \geq N_0 \\ \Leftrightarrow 3n^2 + 6n - 100 \leq 4n^2 & \text{pour tout } n \geq 0 \\ \Leftrightarrow 3n^2 + 6n - 100 \leq 3n^2 + n^2 & \text{pour tout } n \geq 0 \\ \Leftrightarrow n^2 - 6n + 100 \leq 0 & \text{pour tout } n \geq 0 \end{array}$$

qui est toujours valable. On a donc bien que  $f(n) \in O(n^2)$ .

GRAND  $O$ 

## EXAMPLE

Soit  $f : \mathbb{N} \rightarrow \mathbb{N}$  définie par

$$f(n) = 3n^2 + 6n - 100$$

On montre que  $f(n) \in O(n^2)$ . Fixons  $C = 4$  et  $N_0 = 0$ . On a alors:

$$\begin{array}{ll} 3n^2 + 6n - 100 \leq Cn^2 & \text{pour tout } n \geq N_0 \\ \Leftrightarrow 3n^2 + 6n - 100 \leq 4n^2 & \text{pour tout } n \geq 0 \\ \Leftrightarrow 3n^2 + 6n - 100 \leq 3n^2 + n^2 & \text{pour tout } n \geq 0 \\ \Leftrightarrow n^2 - 6n + 100 \leq 0 & \text{pour tout } n \geq 0 \end{array}$$

qui est toujours valable. On a donc bien que  $f(n) \in O(n^2)$ .

# STRUCTURES DE DONNÉES

- ▶ Une **structures de données (data structure)** permet de stocker en machine des *ensembles dynamiques de données*.
- ▶ Ensemble *dynamique* de données: qui peut évoluer au cours du temps. Insérer, modifier, supprimer des éléments...
- ▶ Exemples (Python): listes, tuples, dictionnaires, sets.

# STRUCTURES DE DONNÉES

- ▶ Une **structures de données (data structure)** permet de stocker en machine des *ensembles dynamiques de données*.
- ▶ Ensemble *dynamique* de données: qui peut évoluer au cours du temps. Insérer, modifier, supprimer des éléments...
- ▶ Exemples (Python): listes, tuples, dictionnaires, sets.

# STRUCTURES DE DONNÉES

- ▶ Une **structures de données (data structure)** permet de stocker en machine des *ensembles dynamiques de données*.
- ▶ Ensemble *dynamique* de données: qui peut évoluer au cours du temps. Insérer, modifier, supprimer des éléments...
- ▶ **Exemples (Python):** listes, tuples, dictionnaires, sets.

# TABLEAUX

- Un **tableau (array)** est une *structure de données*.

indices	0	1	2	3	4	5	6	7
clés	12	-7	33	11	-42	1	0	8

- Représente un *ensemble dynamique, ordonné et de taille fixe* d'éléments de même types.
- Usage: rapidité de traitement en ce qui concerne l'accès aux éléments.



# TABLEAUX

- Un **tableau (array)** est une *structure de données*.

indices	0	1	2	3	4	5	6	7
clés	12	-7	33	11	-42	1	0	8

- Représente un *ensemble dynamique, ordonné et de taille fixe* d'éléments de même types.
- Usage: rapidité de traitement en ce qui concerne l'accès aux éléments.

# TABLEAUX

- Un **tableau (array)** est une *structure de données*.

indices	0	1	2	3	4	5	6	7
clés	12	-7	33	11	-42	1	0	8

- Représente un *ensemble dynamique, ordonné et de taille fixe* d'éléments de même types.
- Usage: rapidité de traitement en ce qui concerne l'accès aux éléments.

# TABLEAUX

- ▶ Taille fixe.
- ▶ Accès aux éléments de manière *directe*, par adressage des cellules via des *indices* (simple).  
`t[3]`, `t[3:7]`, ...
- ▶ Insertion et suppression d'éléments nécessitent la création d'un nouveau tableau (coûteux, procédure de “realloc”).  
`t.append(14)`, `t.pop()`, ...

# TABLEAUX

- ▶ Taille fixe.
- ▶ Accès aux éléments de manière *directe*, par adressage des cellules via des *indices* (simple).

`t[3]`, `t[3:7]`, ...

- ▶ Insertion et suppression d'éléments nécessitent la création d'un nouveau tableau (coûteux, procédure de “realloc”).

`t.append(14)`, `t.pop()`, ...

# TABLEAUX

- ▶ Taille fixe.
- ▶ Accès aux éléments de manière *directe*, par adressage des cellules via des *indices* (simple).  
 $t[3]$ ,  $t[3:7]$ , ...
- ▶ Insertion et suppression d'éléments nécessitent la création d'un nouveau tableau (coûteux, procédure de “realloc”).  
 $t.append(14)$ ,  $t.pop()$ , ...

# TABLEAUX

- ▶ **Principe:** les éléments sont stockés en mémoire de façon *contigüe et accessibles via un indice*.
- ▶ Les *éléments* de du tableaux sont des *objets* qui possèdent une *clé* et un *indice* (et une adresse mémoire).
- ▶ Les clés sont accessibles via leurs indices.



- ▶ Illustration et pseudo codes pour les 3 procédures *rechercher(k)*, *insérer(x)*, *supprimer(x)*

# TABLEAUX

- ▶ **Principe:** les éléments sont stockés en mémoire de façon *contigüe et accessibles via un indice*.
- ▶ Les *éléments* de du tableaux sont des *objets* qui possèdent une *clé* et un *indice* (et une adresse mémoire).
- ▶ Les clés sont accessibles via leurs indices.



- ▶ Illustration et pseudo codes pour les 3 procédures *rechercher(k)*, *insérer(x)*, *supprimer(x)*

# TABLEAUX

- ▶ **Principe:** les éléments sont stockés en mémoire de façon *contigüe et accessibles via un indice*.
- ▶ Les *éléments* de du tableaux sont des *objets* qui possèdent une *clé* et un *indice* (et une adresse mémoire).
- ▶ Les clés sont accessibles via leurs indices.

élément x

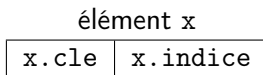
x.cle	x.indice
-------	----------

- ▶ Illustration et pseudo codes pour les 3 procédures *rechercher(k)*, *insérer(x)*, *supprimer(x)*



# TABLEAUX

- ▶ **Principe:** les éléments sont stockés en mémoire de façon *contigüe et accessibles via un indice*.
- ▶ Les *éléments* de du tableaux sont des *objets* qui possèdent une *clé* et un *indice* (et une adresse mémoire).
- ▶ Les clés sont accessibles via leurs indices.



- ▶ Illustration et pseudo codes pour les 3 procédures *rechercher(k)*, *insérer(x)*, *supprimer(x)*

## REMARQUE

- L'accès aux éléments est simple!

# LISTES CHAÎNÉES

- ▶ Une **liste chaînée (linked list)** est une structure de données.
- ▶ Représente un *ensemble dynamique, ordonné et de taille arbitraire* d'éléments de même types.
- ▶ Usage: rapidité de traitement, lorsque l'ordre des éléments est important et que les insertions et les suppressions d'éléments sont plus fréquentes que les accès.

# LISTES CHAÎNÉES

- ▶ Une **liste chaînée (linked list)** est une structure de données.
- ▶ Représente un *ensemble dynamique, ordonné et de taille arbitraire* d'éléments de même types.
- ▶ Usage: rapidité de traitement, lorsque l'ordre des éléments est important et que les insertions et les suppressions d'éléments sont plus fréquentes que les accès.

# LISTES CHAÎNÉES

- ▶ Une **liste chaînée (linked list)** est une structure de données.
- ▶ Représente un *ensemble dynamique, ordonné et de taille arbitraire* d'éléments de même types.
- ▶ Usage: rapidité de traitement, lorsque l'ordre des éléments est important et que les insertions et les suppressions d'éléments sont plus fréquentes que les accès.

# LISTES CHAÎNÉES

- ▶ Taille arbitraire, limitée par la mémoire.
- ▶ Accès aux éléments de manière *séquentielle* via une suite de *pointeurs* (coûteux).
- ▶ Insertion et suppression en début de liste d'éléments en temps constant (simple).

# LISTES CHAÎNÉES

- ▶ Taille arbitraire, limitée par la mémoire.
- ▶ Accès aux éléments de manière *séquentielle* via une suite de *pointeurs* (coûteux).
- ▶ Insertion et suppression en début de liste d'éléments en temps constant (simple).

# LISTES CHAÎNÉES

- ▶ Taille arbitraire, limitée par la mémoire.
- ▶ Accès aux éléments de manière *séquentielle* via une suite de *pointeurs* (coûteux).
- ▶ Insertion et suppression en début de liste d'éléments en temps constant (simple).



# LISTES CHAÎNÉES

- ▶ **Principe:** les *éléments* de la liste sont des *objets* qui possèdent une *clé* et un *pointeur* vers l'élément suivant dans la liste.

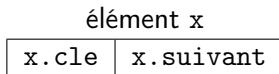
élément x

x.cle	x.suivant
-------	-----------

- ▶ Le dernier élément pointe vers NULL.
- ▶ La liste chaînée peut comporter un attribut *tete* qui désigne le premier élément de la liste.
- ▶ Illustration

# LISTES CHAÎNÉES

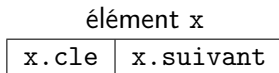
- ▶ **Principe:** les *éléments* de la liste sont des *objets* qui possèdent une *clé* et un *pointeur* vers l'élément suivant dans la liste.



- ▶ Le dernier élément pointe vers NULL.
- ▶ La liste chaînée peut comporter un attribut *tete* qui désigne le premier élément de la liste.
- ▶ Illustration

# LISTES CHAÎNÉES

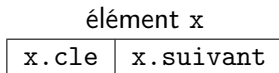
- ▶ **Principe:** les *éléments* de la liste sont des *objets* qui possèdent une *clé* et un *pointeur* vers l'élément suivant dans la liste.



- ▶ Le dernier élément pointe vers NULL.
- ▶ La liste chaînée peut comporter un attribut *tete* qui désigne le premier élément de la liste.
- ▶ Illustration

# LISTES CHAÎNÉES

- ▶ **Principe:** les *éléments* de la liste sont des *objets* qui possèdent une *clé* et un *pointeur* vers l'élément suivant dans la liste.



- ▶ Le dernier élément pointe vers NULL.
- ▶ La liste chaînée peut comporter un attribut *tete* qui désigne le premier élément de la liste.
- ▶ **Illustration**

# IMPLÉMENTATION: ÉLÉMENT DE LC (CELLULE)

```
1 class Cellule:
2     def __init__(self, valeur, suivant=None):
3         """creation d'une cellule
4
5         Args:
6             valeur (quelconque): valeur stockee
7             suiv (Cellule): cellule suivante dans la LC
8         """
9
10        self.valeur = valeur
11        self.suiv = suivant
```

# IMPLÉMENTATION: LC

```
1 class LC:
2
3     def __init__(self):
4         """creation d'une liste chainee"""
5
6         self.tete = None # premier element de la liste
7
8     def est_vide(self):
9         """renvoie un booléen indiquant si
10        la liste est vide"""
11
12        return self.tete is None
```

# IMPLÉMENTATION: LC

```
1  def ajouter_debut(self, x):
2      """Ajoute x en tete de la liste"""
3
4      x.suiv = self.tete
5      self.tete = x
6
7  def ajouter_fin(self, x):
8      """Ajoute le x en queue de la liste"""
9
10     m = self.tete
11     while m.suiv is not None:
12         m = m.suiv
13     m.suiv = x
```

# IMPLÉMENTATION: LC

```
1  def supprimer_debut(self):  
2      """Supprime le 1er element de la liste  
3      et le renvoie"""  
4  
5      if self.est_vide():  
6          raise Exception("La liste est vide !")  
7      t = self.tete  
8      self.tete = t.suiv  
9      return t
```



# IMPLÉMENTATION: LC

```
1  def supprimer_fin(self):
2      """Supprime le dernier element de la liste
3      et le renvoie"""
4
5      if self.est_vide():          # pile vide
6          raise Exception("La pile est vide !")
7      elif self.tete.suiv is None: # 1 seule element
8          e = self.tete
9          self.tete = None
10         return e
11     else:                        # au moins 2 elements
12         e = self.tete
13         e_courant = self.tete
14         while e.suiv is not None:
15             e_courant = e
16             e = e.suiv
17         e_courant.suiv = None
18         return e
```

# IMPLÉMENTATION: LC

```
1  def rechercher(self, valeur):
2      """Recherche valeur dans la liste
3      et renvoie l'element associe"""
4
5      if self.est_vide():
6          raise Exception("La pile est vide !")
7      else:
8          e = self.tete
9          while e is not None:
10             if e.valeur == valeur:
11                 return e
12             else:
13                 e = e.suiv
```

# REMARQUES

- ▶ Les insertions et suppression en tête de liste sont simples!
- ▶ Des piles et files *de tailles arbitraires* peuvent être implémentées par des listes chaînées (TP).
- ▶ Dans le cas de la file, plus besoin du concept d'assignation circulaire des éléments (puisque taille arbitraire).

# REMARQUES

- ▶ Les insertions et suppression en tête de liste sont simples!
- ▶ Des piles et files *de tailles arbitraires* peuvent être implémentées par des listes chaînées (TP).
- ▶ Dans le cas de la file, plus besoin du concept d'assignation circulaire des éléments (puisque taille arbitraire).

# REMARQUES

- ▶ Les insertions et suppression en tête de liste sont simples!
- ▶ Des piles et files *de tailles arbitraires* peuvent être implémentées par des listes chaînées (TP).
- ▶ Dans le cas de la file, plus besoin du concept d'assignation circulaire des éléments (puisque taille arbitraire).

# PILES ET FILES

- ▶ Une **pile (stack)** et une **file (queue)** sont des structures de données.
- ▶ Représentent des *ensembles dynamiques, ordonnés et de tailles arbitraires* d'éléments de même types.
- ▶ Une pile ou une file  $S$  comporte trois opérations (une requête et deux opérations pour être précis):
  - `est_vide(S)`: teste si  $S$  est vide (requête);
  - `ajouter(S,x)`: ajoute l'élément  $x$  dans  $S$  (opération);
  - `supprimer(S)`: supprime l'élément de  $S$  (opération).
- ▶ Ces opérations pourront porter d'autres noms...
- ▶ **Remarque:** Pour ajouter, on spécifie l'élément  $x$  à ajouter. Pour supprimer, pas d'élément  $x$  spécifié. Déterminé par la nature intrinsèque de l'ensemble: pile ou file (cf. suite).

# PILES ET FILES

- ▶ Une **pile (stack)** et une **file (queue)** sont des structures de données.
- ▶ Représentent des *ensembles dynamiques, ordonnés et de tailles arbitraires* d'éléments de même types.

- ▶ Une pile ou une file  $S$  comporte trois opérations (une requête et deux opérations pour être précis):

`est_vide(S)`: teste si  $S$  est vide (requête);

`ajouter(S,x)`: ajoute l'élément  $x$  dans  $S$  (opération);

`supprimer(S)`: supprime l'élément de  $S$  (opération).

- ▶ Ces opérations pourront porter d'autres noms...
- ▶ **Remarque:** Pour ajouter, on spécifie l'élément  $x$  à ajouter. Pour supprimer, pas d'élément  $x$  spécifié. Déterminé par la nature intrinsèque de l'ensemble: pile ou file (cf. suite).

# PILES ET FILES

- ▶ Une **pile (stack)** et une **file (queue)** sont des structures de données.
- ▶ Représentent des *ensembles dynamiques, ordonnés et de tailles arbitraires* d'éléments de même types.
- ▶ Une pile ou une file  $S$  comporte trois opérations (une requête et deux opérations pour être précis):

`est_vide(S)`: teste si  $S$  est vide (requête);

`ajouter(S,x)`: ajoute l'élément  $x$  dans  $S$  (opération);

`supprimer(S)`: supprime l'élément de  $S$  (opération).

- ▶ Ces opérations pourront porter d'autres noms...
- ▶ Remarque: Pour ajouter, on spécifie l'élément  $x$  à ajouter. Pour supprimer, pas d'élément  $x$  spécifié. Déterminé par la nature intrinsèque de l'ensemble: pile ou file (cf. suite).



# PILES ET FILES

- ▶ Une **pile (stack)** et une **file (queue)** sont des structures de données.
- ▶ Représentent des *ensembles dynamiques, ordonnés et de tailles arbitraires* d'éléments de même types.
- ▶ Une pile ou une file  $S$  comporte trois opérations (une requête et deux opérations pour être précis):

`est_vide(S)`: teste si  $S$  est vide (requête);

`ajouter(S,x)`: ajoute l'élément  $x$  dans  $S$  (opération);

`supprimer(S)`: supprime l'élément de  $S$  (opération).

- ▶ Ces opérations pourront porter d'autres noms...
- ▶ Remarque: Pour ajouter, on spécifie l'élément  $x$  à ajouter. Pour supprimer, pas d'élément  $x$  spécifié. Déterminé par la nature intrinsèque de l'ensemble: pile ou file (cf. suite).

# PILES ET FILES

- ▶ Une **pile (stack)** et une **file (queue)** sont des structures de données.
- ▶ Représentent des *ensembles dynamiques, ordonnés et de tailles arbitraires* d'éléments de même types.
- ▶ Une pile ou une file  $S$  comporte trois opérations (une requête et deux opérations pour être précis):

`est_vide(S)`: teste si  $S$  est vide (requête);

`ajouter(S,x)`: ajoute l'élément  $x$  dans  $S$  (opération);

`supprimer(S)`: supprime l'élément de  $S$  (opération).

- ▶ Ces opérations pourront porter d'autres noms...
- ▶ Remarque: Pour ajouter, on spécifie l'élément  $x$  à ajouter. Pour supprimer, pas d'élément  $x$  spécifié. Déterminé par la nature intrinsèque de l'ensemble: pile ou file (cf. suite).

# PILES ET FILES

- ▶ Une **pile (stack)** et une **file (queue)** sont des structures de données.
- ▶ Représentent des *ensembles dynamiques, ordonnés et de tailles arbitraires* d'éléments de même types.
- ▶ Une pile ou une file  $S$  comporte trois opérations (une requête et deux opérations pour être précis):

`est_vide(S)`: teste si  $S$  est vide (requête);

`ajouter(S,x)`: ajoute l'élément  $x$  dans  $S$  (opération);

`supprimer(S)`: supprime l'élément de  $S$  (opération).

- ▶ Ces opérations pourront porter d'autres noms...
- ▶ **Remarque:** Pour ajouter, on spécifie l'élément  $x$  à ajouter. Pour supprimer, pas d'élément  $x$  spécifié. Déterminé par la nature intrinsèque de l'ensemble: pile ou file (cf. suite).

# PILES ET FILES

- ▶ Une **pile (stack)** et une **file (queue)** sont des structures de données.
- ▶ Représentent des *ensembles dynamiques, ordonnés et de tailles arbitraires* d'éléments de même types.
- ▶ Une pile ou une file  $S$  comporte trois opérations (une requête et deux opérations pour être précis):
  - `est_vide(S)`: teste si  $S$  est vide (requête);
  - `ajouter(S,x)`: ajoute l'élément  $x$  dans  $S$  (opération);
  - `supprimer(S)`: supprime l'élément de  $S$  (opération).
- ▶ Ces opérations pourront porter d'autres noms...
- ▶ Remarque: Pour ajouter, on spécifie l'élément  $x$  à ajouter. Pour supprimer, pas d'élément  $x$  spécifié. Déterminé par la nature intrinsèque de l'ensemble: pile ou file (cf. suite).

# PILES ET FILES

- ▶ Une **pile (stack)** et une **file (queue)** sont des structures de données.
- ▶ Représentent des *ensembles dynamiques, ordonnés et de tailles arbitraires* d'éléments de même types.
- ▶ Une pile ou une file  $S$  comporte trois opérations (une requête et deux opérations pour être précis):
  - `est_vide(S)`: teste si  $S$  est vide (requête);
  - `ajouter(S,x)`: ajoute l'élément  $x$  dans  $S$  (opération);
  - `supprimer(S)`: supprime l'élément de  $S$  (opération).
- ▶ Ces opérations pourront porter d'autres noms...
- ▶ **Remarque:** Pour ajouter, on spécifie l'élément  $x$  à ajouter. Pour supprimer, pas d'élément  $x$  spécifié. Déterminé par la nature intrinsèque de l'ensemble: pile ou file (cf. suite).

# PILES

- ▶ Une **pile (stack)** implémente le principe:  
dernier entré, premier sorti, Last In First Out, (LIFO)
- ▶ Comme des assiettes sales que l'on empile: la dernière assiette empilée sera la première à être retirée pour être lavée.
- ▶ Illustration.
- ▶ Applications:
  - ★ undo / redo dans des programmes;
  - ★ page précédente / page suivante dans une navigation internet;
  - ★ récursivité (pile d'appels de fonctions).

# PILES

- ▶ Une **pile (stack)** implémente le principe:  
**dernier entré, premier sorti, Last In First Out, (LIFO)**
- ▶ Comme des assiettes sales que l'on empile: la dernière assiette empilée sera la première à être retirée pour être lavée.
- ▶ Illustration.
- ▶ Applications:
  - ★ undo / redo dans des programmes;
  - ★ page précédente / page suivante dans une navigation internet;
  - ★ récursivité (pile d'appels de fonctions).

# PILES

- ▶ Une **pile (stack)** implémente le principe:  
**dernier entré, premier sorti, Last In First Out, (LIFO)**
- ▶ Comme des assiettes sales que l'on empile: la dernière assiette empilée sera la première à être retirée pour être lavée.
- ▶ Illustration.
- ▶ Applications:
  - ★ undo / redo dans des programmes;
  - ★ page précédente / page suivante dans une navigation internet;
  - ★ récursivité (pile d'appels de fonctions).



# PILES

- ▶ Une **pile (stack)** implémente le principe:  
**dernier entré, premier sorti, Last In First Out, (LIFO)**
- ▶ Comme des assiettes sales que l'on empile: la dernière assiette empilée sera la première à être retirée pour être lavée.
- ▶ **Illustration.**
- ▶ Applications:
  - ★ undo / redo dans des programmes;
  - ★ page précédente / page suivante dans une navigation internet;
  - ★ récursivité (pile d'appels de fonctions).

# PILES

- ▶ Une **pile (stack)** implémente le principe:  
**dernier entré, premier sorti, Last In First Out, (LIFO)**
- ▶ Comme des assiettes sales que l'on empile: la dernière assiette empilée sera la première à être retirée pour être lavée.
- ▶ **Illustration.**
- ▶ **Applications:**
  - ★ undo / redo dans des programmes;
  - ★ page précédente / page suivante dans une navigation internet;
  - ★ récursivité (pile d'appels de fonctions).

# PILES

- ▶ Une **pile (stack)** implémente le principe:  
**dernier entré, premier sorti, Last In First Out, (LIFO)**
- ▶ Comme des assiettes sales que l'on empile: la dernière assiette empilée sera la première à être retirée pour être lavée.
- ▶ **Illustration.**
- ▶ **Applications:**
  - ★ undo / redo dans des programmes;
  - ★ page précédente / page suivante dans une navigation internet;
  - ★ récursivité (pile d'appels de fonctions).

# PILES

- ▶ Une **pile (stack)** implémente le principe:  
**dernier entré, premier sorti, Last In First Out, (LIFO)**
- ▶ Comme des assiettes sales que l'on empile: la dernière assiette empilée sera la première à être retirée pour être lavée.
- ▶ **Illustration.**
- ▶ **Applications:**
  - ★ undo / redo dans des programmes;
  - ★ page précédente / page suivante dans une navigation internet;
  - ★ récursivité (pile d'appels de fonctions).

# PILES

- ▶ Une **pile (stack)** implémente le principe:  
**dernier entré, premier sorti, Last In First Out, (LIFO)**
- ▶ Comme des assiettes sales que l'on empile: la dernière assiette empilée sera la première à être retirée pour être lavée.
- ▶ **Illustration.**
- ▶ **Applications:**
  - ★ undo / redo dans des programmes;
  - ★ page précédente / page suivante dans une navigation internet;
  - ★ récursivité (pile d'appels de fonctions).

# PILES

- ▶ Une **pile** d'au plus  $n$  éléments peut être implémentée par un tableau  $P[0, \dots, n-1]$  ( $0, \dots, n-1$  sont les indices des clés).
- ▶ La pile possède un attribut `P.sommet` qui est l'indice du dernier élément inséré (indice et non élément).
- ▶ Si `P.sommet == -1`, alors la pile est vide.

C'est la requête `est_vide`.

- ▶ Pour insérer un élément  $x$  dans  $P$ , on assigne  $P[P.sommet] = x$  et on incrémente `P.sommet`.

C'est l'opération **empiler** ou **push**.

- ▶ Pour supprimer l'élément en haut de la pile  $P$ , on décrémente `P.sommet` et on retourne  $P[P.sommet+1]$ .

C'est l'opération **dépiler** ou **pop**.

# PILES

- ▶ Une **pile** d'au plus  $n$  éléments peut être implémentée par un tableau  $P[0, \dots, n-1]$  ( $0, \dots, n-1$  sont les indices des clés).
- ▶ La pile possède un attribut `P.sommet` qui est l'indice du dernier élément inséré (indice et non élément).
- ▶ Si `P.sommet == -1`, alors la pile est vide.

C'est la requête `est_vide`.

- ▶ Pour insérer un élément  $x$  dans  $P$ , on assigne `P[P.sommet] = x` et on incrémente `P.sommet`.

C'est l'opération **empiler** ou **push**.

- ▶ Pour supprimer l'élément en haut de la pile  $P$ , on décrémente `P.sommet` et on retourne `P[P.sommet+1]`.

C'est l'opération **dépiler** ou **pop**.

# PILES

- ▶ Une **pile** d'au plus  $n$  éléments peut être implémentée par un tableau  $P[0, \dots, n-1]$  ( $0, \dots, n-1$  sont les indices des clés).
- ▶ La pile possède un attribut `P.sommet` qui est l'indice du dernier élément inséré (indice et non élément).
- ▶ Si `P.sommet == -1`, alors la pile est vide.

C'est la requête `est_vide`.

- ▶ Pour insérer un élément  $x$  dans  $P$ , on assigne `P[P.sommet] = x` et on incrémente `P.sommet`.

C'est l'opération **empiler** ou **push**.

- ▶ Pour supprimer l'élément en haut de la pile  $P$ , on décrémente `P.sommet` et on retourne `P[P.sommet+1]`.

C'est l'opération **dépiler** ou **pop**.



# PILES

- ▶ Une **pile** d'au plus  $n$  éléments peut être implémentée par un tableau  $P[0, \dots, n-1]$  ( $0, \dots, n-1$  sont les indices des clés).
- ▶ La pile possède un attribut  $P.sommet$  qui est l'indice du dernier élément inséré (indice et non élément).
- ▶ Si  $P.sommet == -1$ , alors la pile est vide.

C'est la requête **est\_vide**.

- ▶ Pour insérer un élément  $x$  dans  $P$ , on assigne  $P[P.sommet] = x$  et on incrémente  $P.sommet$ .

C'est l'opération **empiler** ou **push**.

- ▶ Pour supprimer l'élément en haut de la pile  $P$ , on décrémente  $P.sommet$  et on retourne  $P[P.sommet+1]$ .

C'est l'opération **dépiler** ou **pop**.

# PILES

- ▶ Une **pile** d'au plus  $n$  éléments peut être implémentée par un tableau  $P[0, \dots, n-1]$  ( $0, \dots, n-1$  sont les indices des clés).
- ▶ La pile possède un attribut `P.sommet` qui est l'indice du dernier élément inséré (indice et non élément).
- ▶ Si `P.sommet == -1`, alors la pile est vide.

C'est la requête **est\_vide**.

- ▶ Pour insérer un élément  $x$  dans  $P$ , on assigne `P[P.sommet] = x` et on incrémente `P.sommet`.

C'est l'opération **empiler** ou **push**.

- ▶ Pour supprimer l'élément en haut de la pile  $P$ , on décrémente `P.sommet` et on retourne `P[P.sommet+1]`.

C'est l'opération **dépiler** ou **pop**.

# PILES

- ▶ Une **pile** d'au plus  $n$  éléments peut être implémentée par un tableau  $P[0, \dots, n-1]$  ( $0, \dots, n-1$  sont les indices des clés).
- ▶ La pile possède un attribut  $P.sommet$  qui est l'indice du dernier élément inséré (indice et non élément).
- ▶ Si  $P.sommet == -1$ , alors la pile est vide.

C'est la requête **est\_vide**.

- ▶ Pour insérer un élément  $x$  dans  $P$ , on assigne  $P[P.sommet] = x$  et on incrémente  $P.sommet$ .

C'est l'opération **empiler** ou **push**.

- ▶ Pour supprimer l'élément en haut de la pile  $P$ , on décrémente  $P.sommet$  et on retourne  $P[P.sommet+1]$ .

C'est l'opération **dépiler** ou **pop**.

# PILES

- ▶ Une **pile** d'au plus  $n$  éléments peut être implémentée par un tableau  $P[0, \dots, n-1]$  ( $0, \dots, n-1$  sont les indices des clés).
- ▶ La pile possède un attribut  $P.sommet$  qui est l'indice du dernier élément inséré (indice et non élément).
- ▶ Si  $P.sommet == -1$ , alors la pile est vide.

C'est la requête **est\_vide**.

- ▶ Pour insérer un élément  $x$  dans  $P$ , on assigne  $P[P.sommet] = x$  et on incrémente  $P.sommet$ .

C'est l'opération **empiler** ou **push**.

- ▶ Pour supprimer l'élément en haut de la pile  $P$ , on décrémente  $P.sommet$  et on retourne  $P[P.sommet+1]$ .

C'est l'opération **dépiler** ou **pop**.

# PILES

- ▶ Une **pile** d'au plus  $n$  éléments peut être implémentée par un tableau  $P[0, \dots, n-1]$  ( $0, \dots, n-1$  sont les indices des clés).
- ▶ La pile possède un attribut  $P.sommet$  qui est l'indice du dernier élément inséré (indice et non élément).
- ▶ Si  $P.sommet == -1$ , alors la pile est vide.

C'est la requête **est\_vide**.

- ▶ Pour insérer un élément  $x$  dans  $P$ , on assigne  $P[P.sommet] = x$  et on incrémente  $P.sommet$ .

C'est l'opération **empiler** ou **push**.

- ▶ Pour supprimer l'élément en haut de la pile  $P$ , on décrémente  $P.sommet$  et on retourne  $P[P.sommet+1]$ .

C'est l'opération **dépiler** ou **pop**.

# PILES

- Illustrations et pseudo-codes.

# FILES

- ▶ Une **file** (**queue**) implémente le principe:  
premier entré, premier sorti, First In First Out, (FIFO)
- ▶ Comme une file d'attente dans un magasin: la première personne arrivée sera la première servie.
- ▶ Illustration.
- ▶ Applications:
  - ★ traiter des connexions, messages, processus, etc. dans leur ordre d'arrivée.

# FILES

- ▶ Une **file (queue)** implémente le principe:  
**premier entré, premier sorti, First In First Out, (FIFO)**
- ▶ Comme une file d'attente dans un magasin: la première personne arrivée sera la première servie.
- ▶ Illustration.
- ▶ Applications:
  - ★ traiter des connexions, messages, processus, etc. dans leur ordre d'arrivée.



# FILES

- ▶ Une **file** (**queue**) implémente le principe:  
**premier entré, premier sorti, First In First Out, (FIFO)**
- ▶ Comme une file d'attente dans un magasin: la première personne arrivée sera la première servie.
- ▶ Illustration.
- ▶ Applications:
  - ★ traiter des connexions, messages, processus, etc. dans leur ordre d'arrivée.

# FILES

- ▶ Une **file** (**queue**) implémente le principe:  
**premier entré, premier sorti, First In First Out, (FIFO)**
- ▶ Comme une file d'attente dans un magasin: la première personne arrivée sera la première servie.
- ▶ **Illustration.**
- ▶ Applications:
  - ★ traiter des connexions, messages, processus, etc. dans leur ordre d'arrivée.

# FILES

- ▶ Une **file** (**queue**) implémente le principe:  
**premier entré, premier sorti, First In First Out, (FIFO)**
- ▶ Comme une file d'attente dans un magasin: la première personne arrivée sera la première servie.
- ▶ **Illustration.**
- ▶ **Applications:**
  - ★ traiter des connexions, messages, processus, etc. dans leur ordre d'arrivée.

# FILES

- ▶ Une **file** (**queue**) implémente le principe:  
**premier entré, premier sorti, First In First Out, (FIFO)**
- ▶ Comme une file d'attente dans un magasin: la première personne arrivée sera la première servie.
- ▶ **Illustration.**
- ▶ **Applications:**
  - ★ traiter des connexions, messages, processus, etc. dans leur ordre d'arrivée.

# FILES

- ▶ Une **file** d'au plus  $n-1$  éléments (et non  $n$  éléments) peut être implémentée par un tableau  $F[0, \dots, n-1]$  ( $0, \dots, n-1$  sont les indices des clés).
- ▶ La file possède deux attributs:
  - $F.tete$ : indice du prochain indice du prochain élément à sortir (élément courant de la file)
  - $F.queue$ : indice du prochain élément à entrer
- ▶ Les éléments de la file se trouvent aux emplacements  $F.tete, F.tete + 1, \dots, F.queue - 1$  après quoi on boucle (l'emplacement 0 suit l'emplacement  $n-1$ ).
- ▶ Illustration

# FILES

- ▶ Une **file** d'au plus  $n-1$  éléments (et non  $n$  éléments) peut être implémentée par un tableau  $F[0, \dots, n-1]$  ( $0, \dots, n-1$  sont les indices des clés).

- ▶ La file possède deux attributs:

$F.tete$ : indice du prochain indice du prochain élément à sortir  
(élément courant de la file)

$F.queue$ : indice du prochain élément à entrer

- ▶ Les éléments de la file se trouvent aux emplacements  
 $F.tete, F.tete + 1, \dots, F.queue - 1$   
après quoi on boucle (l'emplacement 0 suit l'emplacement  $n-1$ ).

- ▶ Illustration

# FILES

- ▶ Une **file** d'au plus  $n-1$  éléments (et non  $n$  éléments) peut être implémentée par un tableau  $F[0, \dots, n-1]$  ( $0, \dots, n-1$  sont les indices des clés).

- ▶ La file possède deux attributs:

$F.tete$ : indice du prochain indice du prochain élément à sortir (élément courant de la file)

$F.queue$ : indice du prochain élément à entrer

- ▶ Les éléments de la file se trouvent aux emplacements  $F.tete, F.tete + 1, \dots, F.queue - 1$  après quoi on boucle (l'emplacement 0 suit l'emplacement  $n-1$ ).

## ▶ Illustration

# FILES

- ▶ Une **file** d'au plus  $n-1$  éléments (et non  $n$  éléments) peut être implémentée par un tableau  $F[0, \dots, n-1]$  ( $0, \dots, n-1$  sont les indices des clés).
- ▶ La file possède deux attributs:
  - $F.tete$ : indice du prochain indice du prochain élément à sortir (élément courant de la file)
  - $F.queue$ : indice du prochain élément à entrer
- ▶ Les éléments de la file se trouvent aux emplacements  $F.tete, F.tete + 1, \dots, F.queue - 1$  après quoi on boucle (l'emplacement 0 suit l'emplacement  $n-1$ ).
- ▶ Illustration



# FILES

- ▶ Une **file** d'au plus  $n-1$  éléments (et non  $n$  éléments) peut être implémentée par un tableau  $F[0, \dots, n-1]$  ( $0, \dots, n-1$  sont les indices des clés).
- ▶ La file possède deux attributs:
  - $F.tete$ : indice du prochain indice du prochain élément à sortir (élément courant de la file)
  - $F.queue$ : indice du prochain élément à entrer
- ▶ Les éléments de la file se trouvent aux emplacements  $F.tete, F.tete + 1, \dots, F.queue - 1$   
après quoi on boucle (l'emplacement 0 suit l'emplacement  $n-1$ ).

## ▶ Illustration

# FILES

- ▶ Une **file** d'au plus  $n-1$  éléments (et non  $n$  éléments) peut être implémentée par un tableau  $F[0, \dots, n-1]$  ( $0, \dots, n-1$  sont les indices des clés).
- ▶ La file possède deux attributs:
  - $F.tete$ : indice du prochain indice du prochain élément à sortir (élément courant de la file)
  - $F.queue$ : indice du prochain élément à entrer
- ▶ Les éléments de la file se trouvent aux emplacements  $F.tete, F.tete + 1, \dots, F.queue - 1$   
après quoi on boucle (l'emplacement 0 suit l'emplacement  $n-1$ ).
- ▶ **Illustration**

# FILES

- ▶ Au départ, on a  $F.tete = F.queue = 0$
- ▶ Si  $F.tete = F.queue$ , alors la file est vide.  
C'est la requête **est\_vide**.
- ▶ Si  $F.tete = F.queue + 1$ , alors la file est pleine.
- ▶ Pour insérer un élément  $x$  dans  $F$ , on assigne  $F[F.queue] = x$  et on incrémente  $F.queue$ .  
C'est l'opération **enfiler** ou **push**.
- ▶ Pour supprimer l'élément en premier de la file  $F$ , on décrémente  $F.tete$  et on retourne  $F[F.tete + 1]$ .  
C'est l'opération **défiler** ou **pop**.

# FILES

- ▶ Au départ, on a  $F.tete = F.queue = 0$
- ▶ Si  $F.tete = F.queue$ , alors la file est vide.

C'est la requête **est\_vide**.

- ▶ Si  $F.tete = F.queue + 1$ , alors la file est pleine.
- ▶ Pour insérer un élément  $x$  dans  $F$ , on assigne  $F[F.queue] = x$  et on incrémente  $F.queue$ .

C'est l'opération **enfiler** ou **push**.

- ▶ Pour supprimer l'élément en premier de la file  $F$ , on décrémente  $F.tete$  et on retourne  $F[F.tete + 1]$ .

C'est l'opération **défiler** ou **pop**.

# FILES

- ▶ Au départ, on a  $F.tete = F.queue = 0$
- ▶ Si  $F.tete = F.queue$ , alors la file est vide.  
C'est la requête **est\_vide**.
- ▶ Si  $F.tete = F.queue + 1$ , alors la file est pleine.
- ▶ Pour insérer un élément  $x$  dans  $F$ , on assigne  $F[F.queue] = x$  et on incrémente  $F.queue$ .  
C'est l'opération **enfiler** ou **push**.
- ▶ Pour supprimer l'élément en premier de la file  $F$ , on décrémente  $F.tete$  et on retourne  $F[F.tete + 1]$ .  
C'est l'opération **défiler** ou **pop**.

# FILES

- ▶ Au départ, on a  $F.tete = F.queue = 0$
- ▶ Si  $F.tete = F.queue$ , alors la file est vide.  
C'est la requête **est\_vide**.
- ▶ Si  $F.tete = F.queue + 1$ , alors la file est pleine.
- ▶ Pour insérer un élément  $x$  dans  $F$ , on assigne  $F[F.queue] = x$  et on incrémente  $F.queue$ .  
C'est l'opération **enfiler** ou **push**.
- ▶ Pour supprimer l'élément en premier de la file  $F$ , on décrémente  $F.tete$  et on retourne  $F[F.tete + 1]$ .  
C'est l'opération **défiler** ou **pop**.

## FILES

- ▶ Au départ, on a  $F.tete = F.queue = 0$
- ▶ Si  $F.tete = F.queue$ , alors la file est vide.  
C'est la requête **est\_vide**.
- ▶ Si  $F.tete = F.queue + 1$ , alors la file est pleine.
- ▶ Pour insérer un élément  $x$  dans  $F$ , on assigne  $F[F.queue] = x$  et on incrémente  $F.queue$ .

C'est l'opération **enfiler** ou **push**.

- ▶ Pour supprimer l'élément en premier de la file  $F$ , on décrémente  $F.tete$  et on retourne  $F[F.tete + 1]$ .

C'est l'opération **défiler** ou **pop**.

## FILES

- ▶ Au départ, on a  $F.tete = F.queue = 0$
- ▶ Si  $F.tete = F.queue$ , alors la file est vide.  
C'est la requête **est\_vide**.
- ▶ Si  $F.tete = F.queue + 1$ , alors la file est pleine.
- ▶ Pour insérer un élément  $x$  dans  $F$ , on assigne  $F[F.queue] = x$  et on incrémente  $F.queue$ .  
C'est l'opération **enfiler** ou **push**.
- ▶ Pour supprimer l'élément en premier de la file  $F$ , on décrémente  $F.tete$  et on retourne  $F[F.tete + 1]$ .  
C'est l'opération **défiler** ou **pop**.



# FILES

- ▶ Au départ, on a  $F.tete = F.queue = 0$
- ▶ Si  $F.tete = F.queue$ , alors la file est vide.  
C'est la requête **est\_vide**.
- ▶ Si  $F.tete = F.queue + 1$ , alors la file est pleine.
- ▶ Pour insérer un élément  $x$  dans  $F$ , on assigne  $F[F.queue] = x$  et on incrémente  $F.queue$ .  
C'est l'opération **enfiler** ou **push**.
- ▶ Pour supprimer l'élément en premier de la file  $F$ , on décrémente  $F.tete$  et on retourne  $F[F.tete + 1]$ .  
C'est l'opération **défiler** ou **pop**.

# FILES

- ▶ Au départ, on a  $F.tete = F.queue = 0$
- ▶ Si  $F.tete = F.queue$ , alors la file est vide.  
C'est la requête **est\_vide**.
- ▶ Si  $F.tete = F.queue + 1$ , alors la file est pleine.
- ▶ Pour insérer un élément  $x$  dans  $F$ , on assigne  $F[F.queue] = x$  et on incrémente  $F.queue$ .  
C'est l'opération **enfiler** ou **push**.
- ▶ Pour supprimer l'élément en premier de la file  $F$ , on décrémente  $F.tete$  et on retourne  $F[F.tete + 1]$ .  
C'est l'opération **défiler** ou **pop**.

# FILES

- Illustrations et pseudo-codes.