# Les files

**Objectifs**

— Connaître la structure de file et savoir s'en servir

## Introduction

La deuxième structure dynamique abstraite que nous allons voir est la **file**. Il s'agit d'une structure basée sur le principe **FIFO (First In First Out)** : le premier élément entré est le premier à sortir.

On l'utilise pour le traitement des files d'attentes, les communications entre processus, les parcours de graphes (BFS), etc.

## Structure d'une file

### Principe

**Définition**

Une file `F` est une structure de données qui ne propose que les 4 opérations suivantes :

— `Lire(F)` : renvoie la valeur de l'élément en **tête** de la file

— `EstVide(F)` : renvoie `VRAI` si la file est vide, `FAUX` sinon

— `Enfiler(F, x)` : ajoute l'élément `x` à la **fin** de la file

— `Défiler(F)` : supprime l'élément en **tête** de la file et renvoie sa valeur

En théorie, chacune de ces opérations s'exécute en temps constant : $O(1)$.

**Exemple d'évolution d'une file**

Considérons la séquence d'instructions suivante sur une file initialement vide :

1. `Enfiler(F, 1)` $\rightarrow$ File : `[1]` (1 en tête)

2. `Enfiler(F, 3)` → File : `[1, 3]` (1 en tête, 3 en fin)
3. `Défiler(F)` → Renvoie 1, File : `[3]`
4. `Défiler(F)` → Renvoie 3, File : `[]` (vide)

**Représentation horizontale (convention usuelle) :**

```
Enfiler(F,1)    Enfiler(F,3)     Défiler(F)      Défiler(F)

     []       →     [1]         →   [1|3]    →    [3]       →     []
                  (tête)          (tête|fin)     (tête)
```

Le principe FIFO signifie que le premier élément enfilé (1) est le premier à être défilé.

### Implémentation naïve

On peut implémenter une file contenant au plus $n$ éléments à l'aide d'un tableau de taille $n$ et deux indices `tete` et `queue` :

- Le tableau `F[1..n]` contient les éléments enfilés
- Deux variables entières `tete` et `queue` indiquent respectivement la position de l'élément en tête et la position disponible pour enfiler un élément à la fin
- La file correspond aux éléments `F[tete : queue - 1]`

## EstVide

```
ALGORITHME EstVide(F):
DONNEES
  F : tableau de taille n
DEBUT
  SI tete = queue ALORS
    RENVOYER VRAI
  SINON
    RENVOYER FAUX
  FSI
FIN
```

## Lire

```
ALGORITHME Lire(F):
DONNEES
  F : tableau de taille n
```

```
DEBUT
  RENVOYER F[tete]
FIN
```

### Enfiler

```
ALGORITHME Enfiler(F, x):
DONNEES
  F : tableau de taille n
DEBUT
  SI queue = n ALORS
    AFFICHER "débordement"
    RENVOYER ERREUR
  SINON
    F[queue] ← x
    queue ← queue + 1
  FSI
FIN
```

### Défiler

```
ALGORITHME Defiler(F):
DONNEES
  F : tableau de taille n
DEBUT
  SI EstVide(F) = VRAI ALORS
    RENVOYER ERREUR
  SINON
    tete ← tete + 1
    RENVOYER F[tete-1]
  FSI
FIN
```

> ⚠ Attention : Problème de débordement
>
> Cette implémentation **naïve** a un problème majeur : même si la file est logiquement vide après plusieurs enfilements et défilements, l'indice `queue` continue d'avancer. On peut donc avoir un débordement même si la file ne contient que quelques éléments !
> **Solution :** Utiliser un **tableau circulaire**.

#### Implémentation avec tableau circulaire

Il faut que lorsque la `queue` arrive à $n$, le prochain élément soit à l'indice 1 (on "boucle" le tableau).

> **Exemple de tableau circulaire**
>
> Tableau de taille 7 avec les éléments 7, 3, 9 aux positions 2, 3, 4 :
>
> ```
> Indices:  1    2    3    4    5    6    7    (1)   ...
>          [ ] [7] [3] [9] [ ] [ ] [ ]  (retour à 1)
>               ↑              ↑
>             tête           queue
> ```
>
> Après la case 7, on revient à la case 1 grâce à l'opération modulo.

**Algorithmes corrigés avec modulo :**

### Enfiler (circulaire)

```
ALGORITHME Enfiler(F, x):
DONNEES
  F : tableau de taille n
DEBUT
  SI (queue + 1) mod n = tete ALORS
    AFFICHER "débordement"
    RENVOYER ERREUR
  SINON
    F[queue] ← x
    queue ← (queue + 1) mod n
  FSI
FIN
```

### Défiler (circulaire)

```
ALGORITHME Defiler(F):
DONNEES
  F : tableau de taille n
VARIABLES
  x : entier
DEBUT
  SI EstVide(F) = VRAI ALORS
```

```
    RENVOYER ERREUR
  SINON
    x ← F[tete]
    tete ← (tete + 1) mod n
    RENVOYER x
  FSI
FIN
```

**!** Remarques sur l'implémentation circulaire

- Les variables `tete` et `queue` sont **globales** et partagées entre toutes les opérations
- L'opération `mod n` permet de "boucler" les indices
- La condition de débordement devient : `(queue + 1) mod n = tete`
- Chaque opération reste en temps $O(1)$
- On perd une case (ne peut stocker que $n-1$ éléments) pour distinguer file vide de file pleine

### Exemple : Affichage de tous les nombres binaires entre 1 et $n$

**i** Spécification du problème

| | |
|---|---|
| *Problème* | Affichage des écritures binaires des nombres entre 1 et $n$ |
| *Entrée* | Un entier $n$ |
| *Sortie* | Les écritures en binaire des nombres de 1 à $n$ |

#### Version naïve

**Principe :** On énumère les entiers de 1 à $n$ et à chaque fois, on les convertit en binaire.

### ConvertirBase2

```
ALGORITHME ConvertirBase2(x):
DONNEES
  x : entier en base 10
VARIABLES
  C : chaîne de caractères vide
DEBUT
```

```
  TQ x > 0 FAIRE
    SI (x mod 2) = 1 ALORS
      C ← Concatener("1", C)
    SINON
      C ← Concatener("0", C)
    FSI
    x ← x / 2
  FTQ
  RENVOYER C
FIN
```

**Complexité :** $\Theta(\log_2(x))$

## EnumereBase2

```
ALGORITHME EnumereBase2(n):
DONNEES
  n : entier
VARIABLES
  i : entier
DEBUT
  i ← 1
  TQ i   n FAIRE
    Afficher(ConvertirBase2(i))
    i ← i + 1
  FTQ
FIN
```

**Complexité :** $\Theta(n \log_2(n))$

### Version avec file

**Principe :**

L'idée principale repose sur le fait qu'on peut obtenir toutes les écritures des nombres entiers entre $2^k$ et $2^{k+1} - 1$ en prenant les écritures des entiers entre $2^{k-1}$ et $2^k - 1$ et en ajoutant "0" puis "1" à la fin de l'écriture.

De plus, comme on enfile les écritures dans l'ordre croissant des valeurs et qu'on n'omet aucune écriture, notre file contient dans l'ordre les nombres de 1 à $n$.

**ℹ Arbre de génération**

```
          1
        /   \
      10     11
     / \    / \
   100 101 110 111
```

À partir de “1”, on génère “10” et “11”. À partir de “10”, on génère “100” et “101”. À partir de “11”, on génère “110” et “111”.
**Parcours en largeur** : 1, 10, 11, 100, 101, 110, 111…

### Algorithme avec file

```
ALGORITHME EnumereFileBase2(n):
DONNEES
  n : entier
VARIABLES
  i : entier
  C : chaîne de caractères
  F : File vide
DEBUT
  i ← 1
  Enfiler(F, "1")
  TQ i   n FAIRE
    C ← Defiler(F)
    Afficher(C)
    Enfiler(F, Concatener(C, "0"))
    Enfiler(F, Concatener(C, "1"))
    i ← i + 1
  FTQ
FIN
```

### Complexité

**Complexité :** $\Theta(n)$

Bien plus efficace que la version naïve en $\Theta(n \log n)$ !

**ℹ Exemple d'exécution pour n=7**

| Étape | Défile | Affiche | Enfile | État de la file |
|---|---|---|---|---|
| Init | - | - | "1" | `[1]` |
| 1 | "1" | 1 | "10", "11" | `[10, 11]` |
| 2 | "10" | 10 | "100", "101" | `[11, 100, 101]` |
| 3 | "11" | 11 | "110", "111" | `[100, 101, 110, 111]` |
| 4 | "100" | 100 | ... | ... |
| 5 | "101" | 101 | ... | ... |
| 6 | "110" | 110 | ... | ... |
| 7 | "111" | 111 | ... | ... |

Résultat : `1, 10, 11, 100, 101, 110, 111` (en binaire : 1, 2, 3, 4, 5, 6, 7)

### Comparaison File vs Pile : Parcours de graphes

**! Remarque fondamentale**

Utiliser une **file** revient à faire un **parcours en largeur (BFS)** d'un graphe, alors qu'une **pile** permet de faire le **parcours en profondeur (DFS)**.

#### Parcours en largeur (BFS) avec file

On visite tous les nœuds niveau par niveau :

```
              1 (1)
            /       \
        10 (2)       11 (3)
       /    \       /     \
  100(4)  101(5)  110(6)  111(7)
```

**Ordre de visite :** 1, 10, 11, 100, 101, 110, 111

#### Parcours en profondeur (DFS) avec pile

On explore complètement une branche avant de passer à la suivante :

```
              1 (1)
            /       \
      10 (5)       11 (2)
     /    \       /    \
100(7) 101(6) 110(4) 111(3)
```

**Ordre de visite :** 1, 11, 111, 110, 10, 101, 100

Applications

— **BFS (File)** : Plus court chemin, exploration niveau par niveau, génération ordonnée
— **DFS (Pile)** : Détection de cycles, exploration exhaustive, problèmes de backtracking

### Comparaison Pile vs File

| Caractéristique | Pile (Stack) | File (Queue) |
|---|---|---|
| Principe | LIFO (Last In First Out) | FIFO (First In First Out) |
| Ajout | Empiler au sommet | Enfiler à la fin |
| Retrait | Dépiler du sommet | Défiler de la tête |
| Parcours graphe | Profondeur (DFS) | Largeur (BFS) |
| Applications | Récursion, undo/redo, expressions | Files d'attente, BFS, ordonnancement |
| Complexité ops | $O(1)$ | $O(1)$ (avec tableau circulaire) |