

# Structure de Données

## Pile, File

Marie Pelleau

`marie.pelleau@univ-cotedazur.fr`

Semestre 3

# Plan

- 1 Pile
- 2 File
- 3 Deque
- 4 Queue de priorité

# Pile

- Une pile (en anglais *stack*) est une structure de données fondée sur le principe “dernier arrivé, premier sorti” (ou LIFO pour *Last In, First Out*)
- Les derniers éléments ajoutés à la pile seront les premiers à être récupérés

## Exemple

- Pile d'assiettes : on ajoute des assiettes sur la pile, et on les récupère dans l'ordre inverse, en commençant par la dernière ajoutée
- Pile de crêpes

# Pile

## Opérations

- **Sommet**(P) : renvoie le dernier élément ajouté et non encore retiré : le sommet (*top*)
- **Empiler**(P, elt) : comme insérer, place l'élément au sommet de la pile P (*push*)
- **Désempler**(P) : comme supprimer, retire de la pile le sommet (*pop*)
- **estVide**(P) : renvoie vrai si la pile est vide et faux sinon (*empty*)

# Pile

- On considérera qu'à l'instar d'un tableau, une pile passée en paramètre est systématiquement passée en entrée/sortie
- Une pile passée en paramètre sera globalement modifiée si elle est localement modifiée
- **Déempiler**(P) : modifie effectivement la pile P

# Pile

Une des structures de données les plus fondamentales en informatique : très simple et puissante

## Exemple

La plupart des microprocesseurs gèrent nativement une pile. X86 :

- Le registre ESP sert à indiquer l'adresse du sommet d'une pile dans la RAM
- Les opcodes "PUSH" et "POP" permettent respectivement d'empiler et de dépiler des données
- Les opcodes "CALL" et "RET" utilisent la pile pour appeler une fonction et la quitter par la suite en retournant à l'instruction suivant immédiatement l'appel
- En cas d'interruption, les registres EFLAGS, CS et EIP sont automatiquement empilés

# Pile

Une des structures de données les plus fondamentales en informatique : très simple et puissante

## Exemple

Langages de programmation compilés, pour chaque fonction la pile contient

- Les paramètres d'appel des procédures ou fonctions
- Les variables locales
- Le point de retour

# Pile

## Utilisation

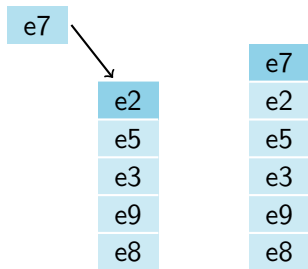
- La fonction “Annuler la frappe” (en anglais *Undo*) mémorise les modifications apportées au texte dans une pile
- Parseur d'expressions XML, des pages web
- Un algorithme de recherche en profondeur dans un graphe utilise une pile pour mémoriser les nœuds visités
- Les algorithmes récursifs utilisent implicitement une pile d'appels



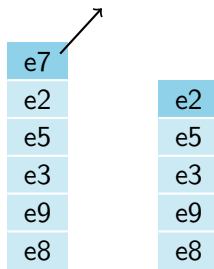
# Pile

## Représentation

Empiler e7

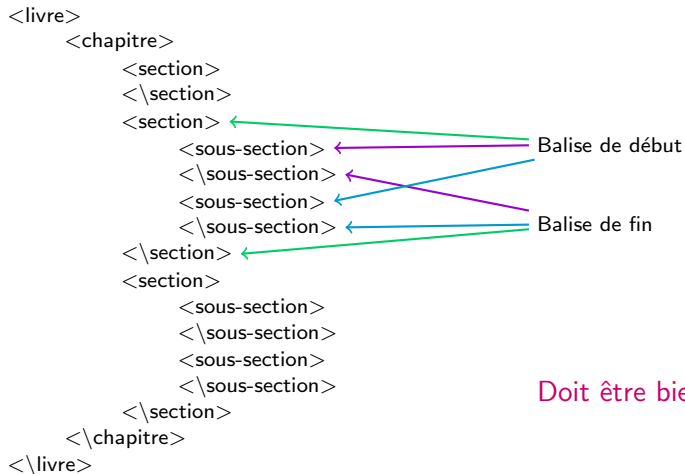


Désempiler



# Pile

## Vérification expression XML



Doit être bien équilibré !

# Pile

## Vérification expression XML

On rencontre une balise

```
si (balise de début) {  
    on l'empile  
}  
si (balise de fin) {  
    si (sommet correspond à la balise de début) {  
        on désempile  
    } sinon {  
        erreur  
    }  
}
```

# Pile

## Vérification expression XML

```
<livre>  
  <chapitre>  
    <section>  
    <\section>  
    <section>  
      <sous-section>  
      <\sous-section>  
      <sous-section>  
      <\sous-section>  
    <\section>  
    <section>  
      <sous-section>  
      <\sous-section>  
      <sous-section>  
      <\sous-section>  
    <\section>  
  <\chapitre>  
<\livre>
```

|          |
|----------|
| chapitre |
| livre    |

# Pile

## Vérification expression XML

```

<livre>
  <chapitre>
    <section>
      <\section>
    <section>
      <sous-section>
      <sous-section>
      <\sous-section>
    <\section>  PROBLÈME ICI
    <section>
      <sous-section>
      <\sous-section>
      <sous-section>
      <\sous-section>
    <\section>
  <\chapitre>
<\livre>
  
```

|          |
|----------|
| chapitre |
| livre    |

# Pile

## Vérification expression XML

```

booléen textOk(textXML) {
  P ← CréerPile()
  pour (chaque balise b) {
    si (b est une balise de début) {
      Empiler(P, b)
    } sinon {
      b' ← Sommet(P)
      si (b' n'est pas la balise de début de b) {
        erreur("b_et_b'_incompatibles")
        retourner faux
      } sinon {
        Désempiler(P)
      }
    }
  }
  si (estVide(P)) {
    retourner vrai
  } sinon {
    retourner faux
  }
}

```

# Pile

## Implémentation

- À l'aide de tableaux (stack overflow)
- À l'aide de listes chaînées

# Pile

## Implémentation par un tableau

### Une structure composée

- un tableau (T)
- taille courante (s)

### Opérations

- **Créer**(P, n) : créer P.T de taille n;  $P.s \leftarrow 0$
- **Sommet**(P) : retourner P.T[P.s]
- **Empiler**(P, elt) :  $P.s \leftarrow P.s + 1$ ;  $P.T[P.s] \leftarrow elt$
- **Désempler**(P) :  $P.s \leftarrow P.s - 1$
- **estVide**(P) : retourner  $P.s = 0$

### Attention

- **Désempler**(P) : P.s ne doit pas devenir négatif
- **Empiler**(P, elt) : stack overflow = dépassement de la taille de T



# Plan

1 Pile

2 File

3 Deque

4 Queue de priorité

# File

- Une **file** (en anglais *queue*) est une structure de données basée sur le principe “premier arrivé, premier sorti”, en anglais FIFO (*First In, First Out*),
- Les premiers éléments ajoutés à la file seront les premiers à être récupérés
- Le fonctionnement ressemble à une file d'attente : les premières personnes à arriver sont les premières personnes à sortir de la file

## Exemple

- Une file d'attente : les premières personnes à arriver sont les premières personnes à sortir de la file

# File

## Opérations

- **Début**(F) : renvoie le premier élément ajouté et non encore retiré : le début ou le premier (*front*)
- **Enfiler** (F, elt) : comme insérer, place l'élément à la fin de la file F (*enqueue*)
- **Dé filer** (F) : comme supprimer, retire de la file le premier (*dequeue*)
- **estVide**(F) : renvoie vrai si la file est vide et faux sinon (*empty*)

# File

- On considérera qu'à l'instar d'un tableau ou d'une pile, une file passée en paramètre est systématiquement passée en entrée/sortie
- Une file passée en paramètre sera globalement modifiée si elle est localement modifiée
- **Dé filer** (F) : modifie effectivement la file F

# File

Application principale : les **buffers** (mémoire tampon = espace de mémorisation temporaire)

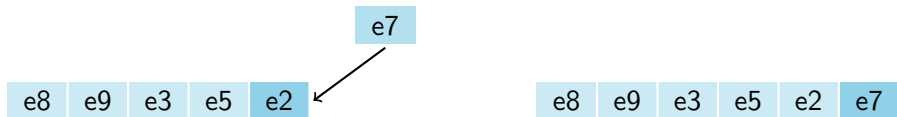
## Utilisation

- Les serveurs d'impression, qui doivent traiter les requêtes dans l'ordre dans lequel elles arrivent, et les insèrent dans une file d'attente
- Certains moteurs multitâches, dans un système d'exploitation, qui doivent accorder du temps-machine à chaque tâche, sans en privilégier aucune
- Un algorithme de parcours en largeur d'un graphe utilise une file pour mémoriser les nœuds visités

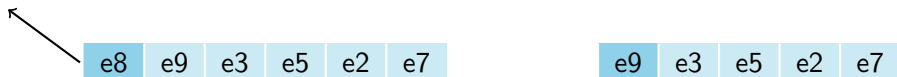
# File

## Représentation

### Enfiler e7



### Défiler



# File

## Implémentation

- À l'aide de tableaux ?
- À l'aide de listes chaînées

# File

## Implémentation par un tableau

### Opérations

- **Enfiler** (F, elt) : on met après le dernier
- **Défiler** (F) : on retire le premier, le tableau se décale vers la droite

### Exemple

|           |   |   |   |   |  |
|-----------|---|---|---|---|--|
|           | 2 | 3 |   |   |  |
| Enfiler 4 | 2 | 3 | 4 |   |  |
| Défiler   |   | 3 | 4 |   |  |
| Enfiler 1 |   | 3 | 4 | 1 |  |
| Défiler   |   |   | 4 | 1 |  |

- On doit gérer un début et une fin de tableau
- Que faire lorsqu'on atteint le borne droite ?  $\Rightarrow$  On devient circulaire



# File

## Implémentation

- Le concept de mémoire tampon circulaire (i.e. tableau dont les extrémités coïncident logiquement)
- Une file est implémentée par une mémoire tampon circulaire
- Physiquement on garde la structure de tableau, mais on considère que l'indice suivant le dernier (i.e.  $n$ ) est 1 (celui du début) et que l'indice précédant le premier (i.e. 1) est la fin (i.e.  $n$ )
- On utilisera un indice de début ( $d$ ) et un indice de fin ( $f$ )

# File

## Une structure composée

- un tableau (T)
- début (d) et fin (f) du tableau

## Implémentation

- Au début  $d = f = 1$
- Quand on ajoute un élément on le met a la place de f et on incrémente f
- Quand on supprime un élément on incrémente d

```
incrémenter(x) {  
    si (x = n) {  
        retourner 1  
    }  
    sinon {  
        retourner x + 1  
    }  
}
```

- On laissera aussi une case vide

# File

## Implémentation par un tableau

### Opérations

- **Début**(F) : retourner F.T[F.d]
- **Enfiler** (F, elt) : F.T[F.f]  $\leftarrow$  elt; F.f  $\leftarrow$  incrémenter(F.f)
- **Dé filer** (F) : F.d  $\leftarrow$  incrémenter(F.d)
- **estVide**(F) : retourner F.d = F.f
- **estPlein** (F) : retourner F.d = incrémenter(F.f)

### Attention

- **Dé filer** (F) : la file ne doit pas être vide
- **Enfiler** (F, elt) : la file ne doit pas être pleine

# Plan

1 Pile

2 File

3 Deque

4 Queue de priorité

# Deque

- Une **double-ended queue** (abrégé *deque* et prononcé “deck”) est une structure de données qui implémente une file pour laquelle les éléments peuvent être ajoutés au début et en fin
- Elle est souvent appelée **head-tail linked list**

# Deque

## Opérations

- **front**(D) : retourne le premier
- **push\_front**(D,elt) : ajoute au début
- **pop\_front**(D) : supprime le premier
- **back**(D) : retourne le dernier
- **push\_back**(D,elt) : ajoute en fin
- **pop\_back**() : supprime le dernier
- **estVide**(D) : retourne vrai si la deque est vide et faux sinon (*empty*)

# Deque

- On considérera qu'à l'instar d'un tableau, d'une pile, ou d'une file, une deque passée en paramètre est systématiquement passée en entrée/sortie
- Une deque passée en paramètre sera globalement modifiée si elle est localement modifiée
- `pop_front(D)` : modifie effectivement la deque D

# Deque

## Implémentation

- À l'aide de tableaux
- À l'aide de listes chaînées



# Plan

- 1 Pile
- 2 File
- 3 Deque
- 4 Queue de priorité

# Queue de priorité

- En informatique, une **queue de priorité** est un type abstrait élémentaire qui manipule des éléments, chacun ayant une clé, sur laquelle on peut effectuer trois opérations :
  - insérer un élément
  - lire puis supprimer l'élément ayant la plus grande clé
  - tester si la queue de priorité est vide ou pas.
- On ajoute parfois à cette liste l'opération
  - augmenter la clé d'un élément

# Queue de priorité

- Une des structures de données les plus étudiées
- A donné naissance à des tas de structures de données très complexes (vraiment **très** complexes)
- Souvent on impose que la queue soit monotone
  - La valeur du maximum ne fait que décroître
  - La valeur du minimum ne fait que croître

# Queue de priorité

## Implémentation

Une des implémentations les plus souples est d'utiliser un tas binaire

- Augmenter ou diminuer la clé est possible
- On peut ajouter des éléments
- On peut demander le maximum (ou le minimum)
- Toutes les opérations sont en  $O(\log(n))$