

# Algorithmique et Analyse d'Algorithmes

L3 Info

Cours 5 : Structures de données linéaires

Benjamin Wack



2021 – 2022

## La dernière fois

- ▶ Logique de Hoare
- ▶ Dichotomie

## Aujourd'hui

- ▶ Type Abstrait de Données
- ▶ Pile, File, File à Priorité
- ▶ Algorithmes de bon parenthésage

# Plan

Notion de Type Abstrait de Données

Structures linéaires

- Pile

- File

- File à Priorité

Un TAD, plusieurs implantations

Bon parenthésage

- Version simple

- Parenthèses multiples

- Imbrication, contextes

# Motivation

L'écriture d'un algorithme efficace requiert souvent une façon adéquate :

- ▶ d'accéder aux données
- ▶ de les modifier
- ▶ de les réorganiser

Le langage fournit des types **atomiques** (int, float, char, bool...) pour des données *simples*.

Pour des données *complexes*, c'est le rôle de la **structure de données**

- ▶ définie par l'utilisateur
- ▶ ou fournie dans une bibliothèque.

# Type Abstrait de Données : pourquoi, pour qui ?

Ce qui est important c'est **l'interface** de la structure de données.

Le TAD est un contrat entre :

- ▶ chef de projet (décide, spécifie)
- ▶ développeur (implante)
- ▶ utilisateur (utilise)

Il dit **quoi** faire, comment on va s'en **servir**, mais pas **comment** on doit le **réaliser**.

## Remarque

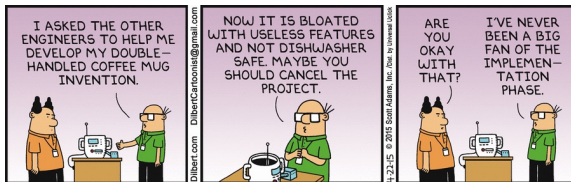
Même les types atomiques ou les tableaux sont déjà une abstraction d'une représentation concrète.

On ne s'amuse pas à manipuler la représentation binaire d'un entier pour changer son signe !

# Avantages de l'approche TAD

En phase de conception :

- ▶ permet de se concentrer sur les fonctionnalités attendues
- ▶ indépendant du langage



En phase de développement :

- ▶ on ne code pas en fonction d'un scénario d'utilisation privilégié
- ▶ démarche de test unitaire

En phase d'utilisation :

- ▶ travail à haut niveau
- ▶ on ne casse pas les propriétés de la structure en la manipulant directement

# Méthodologie

1. Lister les **opérations** autorisées avec leurs profils (parmi lesquelles on peut distinguer des *constructeurs*)
2. Préciser les **préconditions** de chaque opération
3. Spécifier le comportement des différentes opérations au moyen d'**axiomes**

Les axiomes :

- ▶ ne doivent pas être contradictoires
- ▶ doivent permettre de prévoir le résultat de toute opération pour peu que les préconditions soient respectées

Guide d'écriture : toutes les combinaisons valides de la forme *opération(constructeur)* doivent être traitées.

## À propos des opérations

On prend dans ce cours le point de vue fonctionnel :

- ▶ Une opération sur  $t$  renvoie un **nouvel objet** (souvent de type  $t$ )
- ▶ **Pas** d'effets de bord
- ▶ **Pas** de donnée-résultat

En pratique dans la plupart des langages de programmation :

- ▶ Paramètres modifiables
  - ▶ Passage par référence (C)
  - ▶ Caractère modifiable explicite (Ada)
  - ▶  $\Rightarrow$  évite les problèmes d'empreinte mémoire, la gestion d'un ramasse-miettes...

ET/OU

- ▶ Objets et méthodes (Java)
  - ▶ Sous-entend que la méthode peut modifier l'objet
  - ▶ Évite de passer la structure en paramètre
  - ▶ Permet d'utiliser la valeur de retour pour autre chose



## Notion de structure linéaire

On maintient une collection dans laquelle on peut insérer et extraire des éléments.

$$l = (a_1, a_2 \dots a_n) \quad \text{éventuellement vide}$$

- ▶ Chaque élément est placé à une « position » dans la structure
- ▶ Tous les autres éléments sont « avant » ou bien « après »
- ▶ Pas d'autre hiérarchie entre les éléments

Il reste à choisir :

- ▶ où les éléments peuvent être insérés/extraits
- ▶ comment leurs positions respectives sont déterminées

# Aperçu

## Idée intuitive

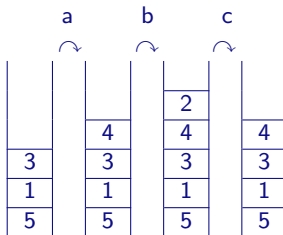
Semblable à une pile d'assiettes.

Politique : **Dernier arrivé, premier servi** (LIFO).

## Applications

- ▶ Appels de fonctions
- ▶ Parcours d'arbre, de graphe avec retour arrière
- ▶ Parenthésage (cf fin du cours)

- a. insérer 4
- b. insérer 2
- c. extraire



# Spécification

**Opérations**

PileVide	:	$void \rightarrow Pile$
Empiler	:	$Element \times Pile \rightarrow Pile$
EstVide	:	$Pile \rightarrow bool$
Sommet	:	$Pile \rightarrow Element$
Dépiler	:	$Pile \rightarrow Pile$

**Préconditions**

Sommet( $p$ )	:	$p$ est non vide
Dépiler( $p$ )	:	$p$ est non vide

**Axiomes**

EstVide(PileVide())	=	vrai
EstVide(Empiler( $e$ , $p$ ))	=	faux
Sommet(Empiler( $e$ , $p$ ))	=	$e$
Dépiler(Empiler( $e$ , $p$ ))	=	$p$

# Aperçu

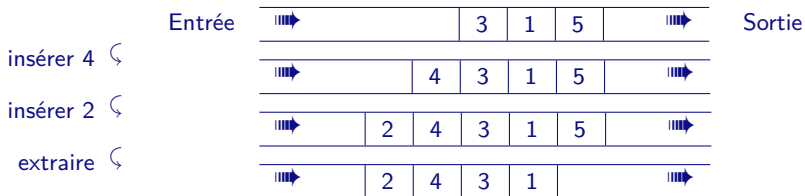
## Idée intuitive

Correspond à la « file d'attente » des supermarchés.

Politique : **Premier arrivé, premier servi (FIFO)**.

## Applications

- ▶ Accès de plusieurs clients à une ressource (service web, imprimante, processeur)
- ▶ Parcours « équilibré » d'arbre, de graphe



# Spécification

**Opérations**

FileVide :  $void \rightarrow File$   
 Enfiler :  $Element \times File \rightarrow File$   
 EstVide :  $File \rightarrow bool$   
 Tête :  $File \rightarrow Element$   
 Défiler :  $File \rightarrow File$

**Préconditions**

Tête( $f$ ) :  $f$  est non vide  
 Défiler( $f$ ) :  $f$  est non vide

## Axiomes

EstVide(FileVide()) = vrai  
 EstVide(Enfiler( $e$ ,  $f$ )) = faux  
 Tête(Enfiler( $e$ , FileVide())) =  $e$   
 Tête(Enfiler( $e$ ,  $f$ )) = Tête( $f$ ) si  $f$  non vide  
 Défiler(Enfiler( $e$ , FileVide())) = FileVide()  
 Défiler(Enfiler( $e$ ,  $f$ )) = Enfiler( $e$ , Défiler( $f$ )) si  $f$  non vide

# Aperçu

## Idée intuitive

Chaque élément est muni d'une priorité.

Politique : **Le plus prioritaire est le premier servi.**

## Applications

- ▶ Files d'attente avec passe-droit
- ▶ Processus dans un système d'exploitation
- ▶ Plusieurs algorithmes (cf fin du semestre)

# Spécification

<b>Opérations</b>	FàPVide	:	$void \rightarrow F\grave{a}P$
	Insérer	:	$Element \times int \times F\grave{a}P \rightarrow F\grave{a}P$
	EstVide	:	$F\grave{a}P \rightarrow bool$
	Prioritaire	:	$F\grave{a}P \rightarrow Element$
	Extraire	:	$F\grave{a}P \rightarrow F\grave{a}P$
	<b>Préconditions</b>	Prioritaire( $f$ )	:
	Extraire( $f$ )	:	$non\ EstVide(f)$

## Spécification de la FàP (axiomes)

$$\text{EstVide}(\text{FàPVide}()) = \text{vrai}$$

$$\text{EstVide}(\text{Insérer}(e, i, f)) = \text{faux}$$

$$\begin{aligned} \text{Prioritaire}(\text{Insérer}(e, i, f)) &= e \\ &\text{si } \text{EstVide}(f) \\ &\text{ou } i > \text{ la priorité de tous les éléments de } f \end{aligned}$$

$$\begin{aligned} \text{Prioritaire}(\text{Insérer}(e, i, f)) &= \text{Prioritaire}(f) \\ &\text{si } f \text{ contient un élément de priorité } > i \end{aligned}$$

$$\begin{aligned} \text{Extraire}(\text{Insérer}(e, i, f)) &= f \\ &\text{si } \text{EstVide}(f) \\ &\text{ou } i > \text{ la priorité de tous les éléments de } f \end{aligned}$$

$$\begin{aligned} \text{Extraire}(\text{Insérer}(e, i, f)) &= \text{Insérer}(e, i, \text{Extraire}(f)) \\ &\text{si } f \text{ contient un élément de priorité } > i \end{aligned}$$

Remarque : comportement de Extraire et Prioritaire **non spécifié** si  $f$  contient plusieurs éléments de priorité maximale.

Dans ce cas on extrait **un des** éléments de priorité maximale, mais pas forcément le premier ou le dernier arrivé.



# Méthodologie d'implantation d'un TAD

- ▶ Choisir une représentation interne (dans un type déjà connu)
- ▶ Fournir les opérations de la spécification
- ▶ Si possible **prouver** les axiomes
- ▶ (Programmation *défensive* : lever une exception si précondition non respectée)

## Principe d'encapsulation

Un TAD bien fait empêche de se servir de l'implantation pour court-circuiter les opérations de la spécification (modules en Ada, OCaml...).

## Exemple : file dans un tableau (naïve)

### Représentation

2	4	3	1	5		
---	---	---	---	---	--	--

- ▶ on mémorise également le nombre  $nb$  d'éléments dans la file
- ▶  $T[nb]$  est en tête de file, et  $T[1]$  en queue

### Opérations

- ▶ FileVide  
 $nb := 0$
- ▶ Défiler( $f$ )  
 $nb := nb - 1$
- ▶ Enfiler( $e, f$ )

2	4	3	1	5		
---	---	---	---	---	--	--

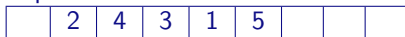


$e$	2	4	3	1	5	
-----	---	---	---	---	---	--

$nb := nb + 1$

# File dans un tableau (efficace)

## Représentation



queue



tete

- ▶ on mémorise **deux** indices tete et queue

## Opérations

- ▶ FileVide  
 $tete := 1$  ;  $queue := 1$
- ▶ Défiler( $f$ )  
 $tete := tete - 1$
- ▶ Enfiler( $f, e$ )  
 $T[queue] := e$   
 $queue := queue - 1$

En cas de dépassement des bornes : tableau « circulaire »



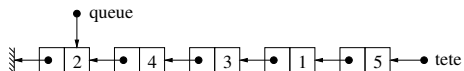
tete



queue

# File dans une liste chaînée

## Représentation



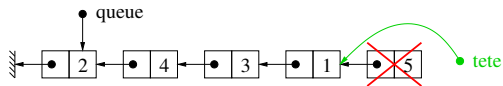
- ▶ on maintient aussi un pointeur sur le dernier élément de la liste.
- ▶ Attention, ce dernier élément est la **queue** de la file.

## Opérations

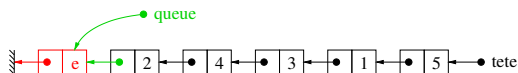
- ▶ FileVide



- ▶ Défiler( $f$ )



- ▶ Enfiler( $f, e$ )



## Que choisir ?

### Tableau

**Avantages** : allocation unique, accès rapide, empreinte mémoire faible

**Inconvénients** : file bornée, ou espace mémoire jamais utilisé

### Liste chaînée

**Avantages** : optimisation de la mémoire, pas de borne sur la taille

**Inconvénients** : gestion de mémoire plus lourde (avec les risques de fuite inhérents)

Cela peut enfin dépendre des opérations supplémentaires voulues (concaténation de files...)

# Le problème

Étant donné un « texte » (flot de caractères) comportant des parenthèses ouvrantes et fermantes, vérifier si chaque parenthèse possède une « correspondante ».

## Exemples

( ( ) ( ) )  
( ( ) ) ( ) sont bien parenthésés.

( ( ) ( )  
( ( ) ) ) ne le sont pas.  
) (

Enfin le texte peut comporter des caractères « neutres » :  
aaa(bbb(cc)dd)e.

# Algorithme

PARENTHEPAGE\_SIMPLE

$p := \text{PileVide}()$

**while** *il reste des caractères*

┌  $c :=$  caractère suivant

┌ **if**  $c = '('$

└ Empiler( $p, c$ )

┌ **if**  $c = ')'$

└ **if**  $\text{EstVide}(p)$

└ ┌ **return** *Erreur de parenthésage*

└ **else**

└ ┌ Dépiler( $p$ )

┌ **if**  $\text{EstVide}(p)$

└ **return** *Parenthésage correct*

**else**

└ **return** *Erreur de parenthésage*

# Une première variante

## Remarque

La pile ne sert ici qu'à compter les parenthèses ouvertes... il suffirait d'un entier !

Cas plus général : différentes paires de parenthèses

- ▶ Langages de programmation     `() {} begin end`
- ▶ Langages de balises     `<html></html> <div></div> <b></b>`
- ▶ ...

## Exemple

`( { ) }` est correct si on ne tient compte que d'un type de parenthèses, mais pas pour les deux à la fois.



## Adaptation de l'algorithme

PARENTHESAGE\_MULTIPLE

$p := \text{PileVide}()$

**while** *il reste des caractères*

$c := \text{caractère suivant}$

**if**  *$c$  est une parenthèse ouvrante*

        └ Empiler( $p$ ,  $c$ )

**if**  *$c$  est une parenthèse fermante*

**if** *EstVide( $p$ ) ou Sommet( $p$ ) ne correspond pas à  $c$*

            └ **return** *Erreur de parenthésage*

**else**

            └ Dépiler( $p$ )

**if** *EstVide( $p$ )*

    └ **return** *Parenthésage correct*

**else**

    └ **return** *Erreur de parenthésage*

Adaptation possible pour les parenthèses non orientées (guillemets...)

# Notion de contexte

Le **niveau d'imbrication** en un point du texte est le nombre de parenthèses ouvertes (et non encore fermées).

On appelle **contexte** une partie du texte comprise entre deux parenthèses correspondantes et de même niveau d'imbrication.

## Exemple

Dans `aaa[bb(ccc{ddd}ccc[eee]ccc)bb]a` :

- ▶ Le niveau d'imbrication maximal est 3 (pour les lettres d et e).
- ▶ Chaque caractère correspond à un contexte différent.

## Quelques problèmes

On peut alors poser plusieurs problèmes :

Calculer le niveau d'imbrication maximal

Il suffit de mémoriser et d'actualiser la hauteur courante de la pile, ainsi que sa hauteur maximale.

Apparier les parenthèses correspondantes en donnant leurs positions dans le texte

Il suffit d'empiler avec chaque parenthèse sa position dans le texte.

Calculer les longueurs des contextes

On maintient à jour la longueur du contexte courant.

Lorsqu'on ouvre une nouvelle parenthèse, on empile également la longueur du contexte « supérieur » et on repart à 0.

Lorsqu'on ferme une parenthèse, on dépile la longueur déjà lue du contexte et on reprend à partir de cette valeur.

# En résumé

## Aujourd'hui

- ▶ Un **Type Abstrait de Données** décrit les **opérations autorisées**, pas le détail de la représentation
- ▶ En **variant les opérations** permises on obtient différentes **structures linéaires** : Pile, File, File à Priorité...
- ▶ Algorithme de bon parenthésage : **efficace** grâce à l'utilisation d'une **structure de données appropriée**

## La prochaine fois

- ▶ Type abstrait Arbre
- ▶ Structures arborescentes
- ▶ Partition Binaire de l'Espace