TDA : tableau et vecteur

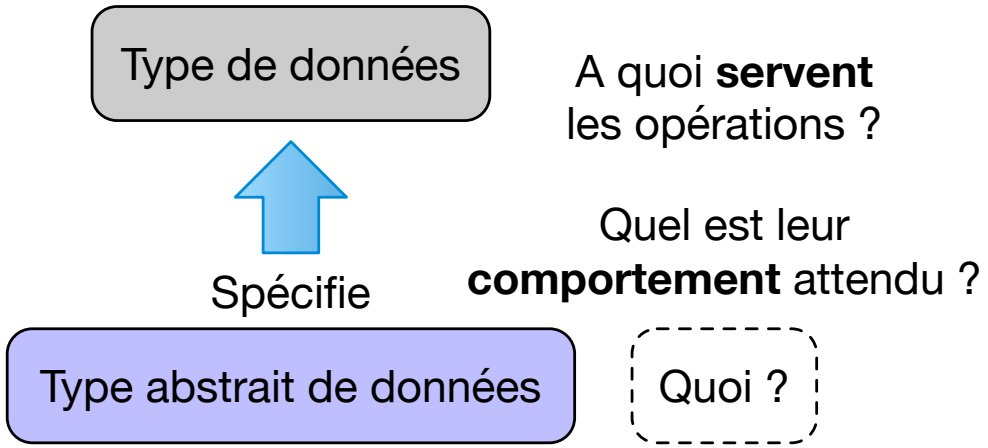
Sébastien Jean

IUT de Valence
Département Informatique

v1.0, 17 novembre 2025

IUT Valence
Université Grenoble Alpes

Types de Données Abstrait (rappels)



- **Spécification mathématique** d'un type de données
  - **Nom**
  - **Dépendances** (autres TDA sont nécessaires)
  - **Opérations** (types des opérandes et du résultat, description)
  - **Pré-conditions** (conditions de validité des paramètres)
  - **Axiomes** (*vérités* sur le comportement des opérations)

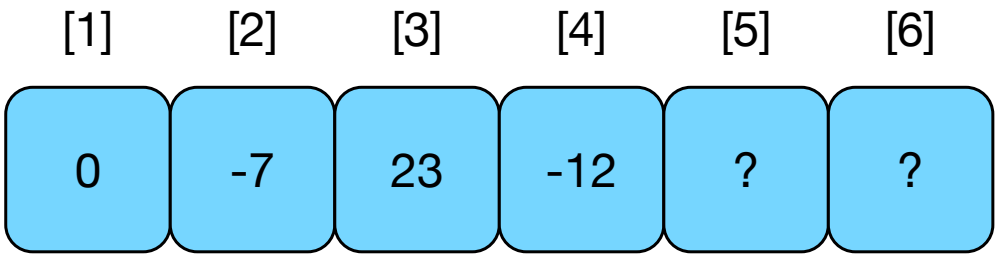
IUT Valence Université Grenoble Alpes

TDA : opérations (rappels, suite)

- 3 catégories :
  - **Constructeurs** : créent une donnée parmi les possibles
  - **Transformateurs**
    - **Producteurs** : produisent une nouvelle donnée à partir d'une ancienne
    - **Mutateurs** : modifient une donnée
  - **Observateurs** : donnent des informations sur la donnée
- **Syntaxe** de la description d'une opération
  - nom : type operande1 x ... x type operandeN $\rightarrow$ type résultat

IUT Valence
Université Grenoble Alpes

TDA : *Tableau*



- Type de données linéaire et **statique**, collection de valeurs stockées dans des cases contigües identifiées par un indice
  - Capacité (nombre de cases) **fixe**, indiquée à la création (statique)
  - Indices allant de 1 à la capacité
- Possibilité de **lire** et **écrire** l'élément à un indice donné, d'**obtenir la capacité**

IUT Valence Université Grenoble Alpes

Interlude : types paramétrés, généricité

- Le tableau contient des valeurs d'un même type, noté `T`
- Mais le type `T` peut varier d'un tableau à l'autre
  - `Entier`, `Booléen`, `Réel`, `Chaîne`, ...
- Le TDA `Tableau` est un **type de données abstrait paramétré** (ou générique), car il dépend d'un paramètre de type
  - Les algorithmes conçus en utilisant un TDA générique sont eux aussi génériques, il ne dépendent pas du (ou des) paramètres de type

IUT Valence Université Grenoble Alpes

TDA : *Tableau* (suite)

- **Nom** : `Tableau` **(de T)**
- **Dépendances** : `Entier`, `T` (type des éléments)
- **Opérations** :
  - **Constructeurs** :
    - `tableau_vide` : `Entier` $\rightarrow$ `Tableau`
  - **Transformateurs** :
    - `ecrire` : `Tableau` x `Entier` x `T` $\rightarrow$ `Tableau`
  - **Observateurs** :
    - `lire` : `Tableau` x `Entier` $\rightarrow$ `T`
    - `capacite` : `Tableau` $\rightarrow$ `Entier`
- *A suivre* . . .

IUT Valence
Université Grenoble Alpes

TDA : *Tableau* (suite)

- **Pré-conditions**
  - `tableau_vide(n)` $\rightarrow$ `n` ≥ 1
  - `ecrire(t, n, v)` $\rightarrow 1 \leq$ `n` $\leq$ (capacité du tableau)
  - `lire(t, n)` $\rightarrow 1 \leq$ `n` $\leq$ (capacité du tableau)
- **Axiomes**
  - `capacite(tableau_vide(n)) = n`
  - `lire(ecrire(t, n, v), n) = v`

- Opérations indépendantes de l'implémentation ?

IUT Valence
Université Grenoble Alpes

Interlude : raccourcis de syntaxe

- Dans notre *pseudo code*, on utilise le raccourci de syntaxe :

```
VARIABLE t : tableau d'entiers [5]
// T est le type entier
```

- Au lieu d'écrire :

```
VARIABLE t : tableau d'entiers
t ← tableau_vide(5)
```

- Dans notre *pseudo code*, on utilise le raccourci de syntaxe :

```
t[3] ← t[2]
```

- Au lieu d'écrire :

```
t ← ecrire(t, 3, lire(t, 2))
```

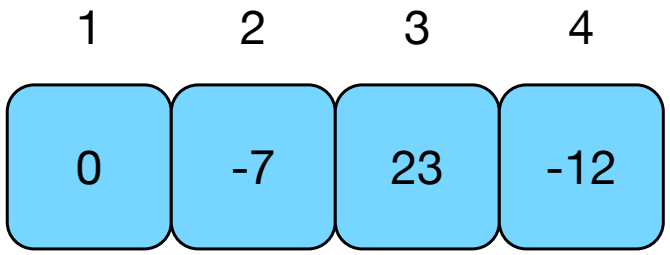
IUT Valence Université Grenoble Alpes

SDD Tableau

- On ne parlera pas d'implémentation du TDA Tableau
  - Il n'y très souvent pas d'alternative pour leur représentation avec un couple (machine, langage)
    - En C, un tableau est une zone contigüe de mémoire, le compilateur résout nativement les accès, les demandes de taille, ...

IUT Valence
Université Grenoble Alpes

TDA : *Vecteur*



- Type de données linéaire et **dynamique**, collection de valeurs (de type `T`) stockées dans des cases contigües identifiées par un indice
  - Indices allant de 1 au **nombre d'éléments**
- Possibilité de **lire** et **écrire** l'élément à un indice donné
- Possibilité d'**insérer** et **retirer** un élément à un indice donné
- Possibilité d'**obtenir la taille** (nombre d'éléments)

IUT Valence Université Grenoble Alpes

TDA : *Vecteur* (suite)

- **Nom** : `Vecteur` **(de T)**
- **Dépendances** : `Entier`, `T` (type des éléments)
- **Opérations** :
  - **Constructeurs** :
    - `vecteur_vide` : $\rightarrow$ `Vecteur`
  - **Transformateurs** :
    - `ecrire` : `Vecteur` x `Entier` x `T` $\rightarrow$ `Vecteur`
    - `inserer` : `Vecteur` x `Entier` x `T` $\rightarrow$ `Vecteur`
    - `retirer` : `Vecteur` x `Entier` $\rightarrow$ `Vecteur`
  - **Observateurs** :
    - `lire` : `Vecteur` x `Entier` $\rightarrow$ `T`
    - `taille` : `Vecteur` $\rightarrow$ `Entier`
- *A suivre* ...

IUT Valence Université Grenoble Alpes

TDA : *Vecteur* (suite)

- **Pré-conditions**
  - `ecrire(v, n, e)` $\rightarrow 1 \leq$ `n` $\leq$ taille(v)
  - `inserer(v, n, e)` $\rightarrow 1 \leq$ `n` $\leq$ taille(v) $+ 1$
  - `retirer(v, n, e)` $\rightarrow 1 \leq$ `n` $\leq$ taille(v)
  - `lire(v, n)` $\rightarrow 1 \leq$ `n` $\leq$ taille(v)

IUT Valence
Université Grenoble Alpes

TDA : *Vecteur* (suite)

- **Axiomes**
  - `taille(vecteur_vide()) = 0`
  - `taille(ecrire(v, n, e)) = taille(v)`
  - `taille(inserer(v, n, e)) = taille(v) + 1`
  - `taille(retirer(v, n)) = taille(v) - 1`
  - `lire(ecrire(vecteur_vide(), 0, e), 0) = e`
  - `lire(ecrire(v, 0, e), n) = lire(v, n) si n` $\geq$ `1`
  - `retirer(inserer(v, n, e), n) = v`
  - `lire(inserer(v, n, e), n) = e`
  - `lire(inserer(v, n, e), p) = lire(v, p) si p` $<$ `n`
  - `lire(inserer(v, n, e), p) = lire(v, p-1) si p` $\geq$ `n`
  - `lire(retirer(v, n), p) = lire(v, p) si p` $<$ `n`
  - `lire(retirer(v, n), p) = lire(v, p+1) si p` $\geq$ `n`

IUT Valence
Université Grenoble Alpes

TDA Vecteur : Pseudo code

- Dans notre *pseudo code*, on suppose que
  - Le constructeur est remplacé par la déclaration de la variable
  - Les transformateurs sont des mutateurs (vecteur modifié en place)

```
VARIABLE v : Vecteur d'entiers
VARIABLE i : entier

inserer(v, 1, 3)

ecrire(v, 1, 2)

i ← lire(v, 1)

retirer(v, 0)

AFFICHER(taille(v))
```

IUT Valence Université Grenoble Alpes

Exercice

Enoncé du problème

Etant donnés un vecteur d'entiers `t`, on souhaite obtenir une copie triée par ordre croissant des éléments.

Spécification du problème

Signature de la fonction

IUT Valence
Université Grenoble Alpes

Exercice

Enoncé du problème

Etant donnés un vecteur d'entiers `t`, on souhaite obtenir une copie triée par ordre croissant des éléments.

Spécification du problème

- **Donnée d'entrée** : *v*, vecteur d'entiers (vecteur original)
- **Donnée de sortie** : *c*, vecteur d'entiers (copie triée de de `v`)
- **Pré-condition** : (aucune)
- **Post-condition** : `c` est une copie de `v` triée par ordre croissant des éléments.

Signature de la fonction

- **tri_vecteur** (v : vecteur d'entiers) : vecteur d'entiers

IUT Valence
Université Grenoble Alpes

Exercice

Corps de la fonction

Exercice

```
FONCTION tri_vecteur(v : vecteur d'entiers) : vecteur
    d'entier

 VARIABLE indice_v   : entier
 VARIABLE indice_c   : entier
 VARIABLE element    : entier
 VARIABLE c          : vecteur d'entiers

 POUR indice_v de 1 A taille(v) PAR PAS DE 1
  element ← lire(v, indice)
  indice_c ← 1
  TANT QUE     indice_c ≤ taille(c)
               ET lire(c, indice_c) < element
   indice_c ← indice_c + 1
  FIN TANT QUE
  c ← inserer(c, indice_c)
 FIN POUR
 RETOURNER c
FIN FONCTION
```

IUT Valence
Université Grenoble Alpes

Fin !