

Structures de données

Application en Java
(Master IPS)

Vincent Berry

Université Montpellier II - C.N.R.S.
<http://www.lirmm.fr/~vberry>

18 janvier 2008

Vincent Berry

Structures de données

Solution informatique à un problème

Analyse du problème à résoudre

- **Organisation** des informations : base de données (schéma), fichiers (structures), structures de données en mémoire (dictionnaires, etc).
- **Traitement** des informations : requêtes SGBD, parsing de fichiers, algorithmes utilisant les structures de données.

Dans les modules précédents (SI-BD, Prog)

- la **difficulté** était principalement la modélisation du problème et l'organisation des données.
- Les structures de données manipulées étaient *simples* (des classes ad-hoc, des tableaux)
- Les traitements réalisés par les programmes impliquaient un *petit* nombre d'opérations (créations de quelques objets, quelques affectations, un parcours de tableau)

Position de ce module dans votre formation

Dans ce module

- la partie modélisation des données sera *simple*
- les structures de données manipulées par programmes seront plus **complexes** (pas toujours linéaires).
- nous examinerons des traitements non-triviaux :
 - codes parfois plus longs
 - exécutions de certaines parties de code un grand nombre de fois (boucles imbriquées)
 - certaines instructions demanderont un temps plus important pour être réalisées (opérations sur des structures complexes à maintenir)

Définition

Une structure de données est à la fois

- *une **organisation** particulière des informations stockées*
- *une liste d'**opérations** permettant d'accéder aux données ayant cette structure.*

Exemples : ensembles, listes, arbres, graphes, dictionnaires.

DESSIN

Choix d'une structure de données

Pour les problèmes que nous voudront modéliser, nous aurons souvent le choix entre plusieurs structures de données.

Certaines structures de données auront plusieurs implémentations possibles.

Suivant la quantité d'informations à manipuler pour le problème traité, le **choix** de la structure de données – ou le choix d'une implémentation – pourra être crucial

- en terme de **temps calcul**
- en terme de **simplicité des algorithmes** à écrire

traitements complexes pourront être réalisés par plusieurs algorithmes possibles.

La complexité (nb d'instructions exécutées) de ces algorithmes ne sera pas toujours la même.

Exemples de traitements pour lesquels plusieurs algorithmes existent :

- recherche d'un élément dans une liste
- tri d'un ensemble d'éléments
- recherche d'un sous-ensemble d'éléments ayant la meilleure valeur pour une fonction à optimiser (ex : problème d'EDT, d'affectation de ressources, de configuration d'une molécule, etc)
- recherche du plus court chemin d'un point à un autre dans un réseau

Choix pédagogiques

Dans ce module

- Nous examinerons les structures de données les plus fréquentes
- Nous *implémenterons* certaines structures
- Nous *utiliserons* des implémentations de structures déjà existantes
- Chaque notion vue en cours sera appliquée en TP.
- Les heures de TP sont majoritaires de façon à privilégier l'appropriation des concepts et outils vus en cours

- *Introduction à l'Algorithmique*
T. Cormen, C. Leiserson, R. Rivest
DUNOD ed.
- *Algorithmique et programmation en Java*
V. Granet
DUNOD ed.

Comparaison d'algorithmes

un problème à résoudre ↔ plusieurs algorithmes/programmes

Pour choisir entre ces algorithmes, on a besoin d'un outil aussi juste que possible, donc indépendant

- du langage de programmation
- de la machine qui exécute les programmes
- d'une instance particulière du problème à résoudre

Un tel outil doit permettre de comparer la vitesse de différents algorithmes pour résoudre un problème.

Au bout du compte, on choisira souvent l'algorithme déclaré comme le plus rapide pour résoudre le problème

(implémenté **une fois** – exécuté de **nombreuses fois**)

La mesure retenue pour mesurer la rapidité relative d'algorithmes est la *complexité* :

Définition

La *complexité* d'un algorithme est le **nombre d'opérations élémentaires** exécutées par l'algorithme en fonction de la **taille de la donnée** et **dans le plus mauvais des cas**.

Opération élémentaire

Définition

Opération élémentaire : opération dont le temps d'exécution est borné par une constante (indépendant de la donnée).

Exemples

- Affectation de variables simples
- Accès à un élément de tableau
- Opérations booléennes, comparaisons
- Opérations arithmétiques (lorsque opérandes et résultats sont bornés)

Contre-Exemples

- Initialisation, comparaison, affectation de tableau

Algorithme: Moyenne (d a, b : entiers) : réel

Données: deux nombre entiers a et b

Résultat: renvoie la moyenne de ces deux nombres

Variable : som entier ;

$som \leftarrow a + b$;

retourner ($som / 2$)

Cet algorithme a une compx de 4 :

- 1 affectation
- 2 opérations arithmétiques
- 1 opération "renvoyer"

Règles pour compter le nombre d'opérations élémentaires

- La complexité d'une **séquence** d'instructions est la somme des complexités des instructions composant la séquence.
- La complexité d'une **instruction conditionnelle** est la complexité de sa condition + $\max\{$ complexité du bloc "alors" , complexité du bloc "sinon" $\}$

Algorithme: **Min** (**d** a, b : entiers) : entier

Données: deux nombre entiers a et b

Résultat: renvoie le plus petit de ces deux nombres

si ($a < b$) **alors**

└ retourner a

sinon

└ retourner b

;

Cet algorithme a une complexité de 2 opérations

Règles pour compter le nombre d'opérations élémentaires

La complexité d'une **boucle** est la \sum de

- complexité de l'initialisation de la variable de boucle
- (nb tours de boucles + 1) \times complexité de la condition de boucle
- nb tours de boucles \times complexité du corps de la boucle

La complexité d'un **appel de fonction** est la \sum de

- complexité de l'évaluation des valeurs passées en paramètre
- complexité des instructions de la fonction pour ces valeurs

Algorithme: Somme-1-a-10 () : entier

Résultat: affiche la somme des 10 premiers entiers

Variable i, s : entiers

$s \leftarrow 0 ; i \leftarrow 1$

tant que $i \leq 10$ **faire**

└ $s \leftarrow s + i ; i \leftarrow i + 1$

afficher_écran (s)

La complexité de cet algorithme est :

2 pour les initialisations de variables **+11** = 11×1 pour la condition de boucle **+40** = 10×4 pour le corps de la boucle **+1** pour la fonction d'affichage = **54**

EXERCICE : *Quelle complexité si la boucle tant que est remplacée par une boucle pour ?*

Taille de la donnée

Définition

Taille de la donnée : taille du codage (nombre de mots mémoire) de l'ensemble des paramètres-donnée.

Simplification

- pour un booléen, caractère, nombre borné : 1
- pour un tableau ou un ensemble :
nombre d'éléments \times taille d'un élément
- pour un arbre :
nombre de noeuds \times taille des infos stockées en un noeud
- pour un graphe :
nb de sommets et nb d'arêtes \times tailles des infos qui leur sont associées

Algorithme: SommeCases (d T[1..n]) : entier

Données: T , un tableau de n cases de type entier

Résultat: affiche la somme des cases de ce tableau

Variable i, s : entiers

$s \leftarrow 0 ; i \leftarrow 1$

tant que $i \leq \text{taille}(T)$ **faire**

$s \leftarrow s + T[i] ; i \leftarrow i + 1$

afficher_écran (s)

La complexité de cet algorithme est de $7n + 5$ (en supposant que la fonction *taille()* a une complexité de 1 opération).

EXERCICE : combien gagne-t-on si on évite un appel à la fonction *taille* à chaque évaluation de la condition ?

Comparaisons d'algorithmes

Motivation pour compter le nombre d'opérations :

comparaison d'algorithmes

Un pbm P à résoudre \longleftrightarrow choix d'algorithmes A_1, A_2, \dots

- Pour une même taille de donnée, le nombre d'opérations élémentaires exécutées par 1 algorithme peut varier selon la valeur des données ;
- Pour un même problème, et une même donnée, le nombre d'opérations de deux algorithmes peut différer, sans que l'un des algorithmes soit meilleur que l'autre en général.

Exemple

Recherche d'un zéro dans un tableau T de nombres :

- A_1 parcourt T de la gauche vers la droite jusqu'à trouver un zéro
- A_2 parcourt T de la droite vers la gauche jusqu'à trouver un zéro

Ne pas comparer A_1 , A_2 forcément sur la même instance du problème :

pour un tableau, A_1 peut être plus rapide que A_2 , sans que ce soit le cas en général.

Type d'analyses

En pratique on effectue plusieurs types d'analyses :

- **dans le plus mauvais des cas** : on compte, le nombre maximum d'opérations élémentaires exécutées par un algorithme A pour les données de taille n .
- **en moyenne** : moyenne des nombres d'opérations élémentaires exécutées par A pour toutes les données de taille n

Le problème P1

Algorithme : Recherche d'un élément dans un tableau.

Données :

- T , un tableau de n cases de type E
- un élément e de type E .

Résultat : renvoie *Vrai* si e est dans une case de T , *Faux* sinon.

	1				n	
	12	8	5	3	10	2

Algorithme: Recherche(d T[1..n], d

) : Bool

Données: T[1 ... n] tableau de E et e
un élément de type E

Résultat: Renvoie Vrai si et
seulement si $e \in T$

variable : i entier

pour $i \leftarrow 1$ à n **faire**

si $T[i] = e$ **alors**

retourner Vrai

retourner Faux

- Taille du problème : $n + 1$

- **Meilleur cas** : $T[1] = e$

affectation 1

addition 0

accès tableau 1

comparaisons 1

renvoyer 1

en tout 4.

- **Pire cas** : $e \notin T$

n itérations

affectations $n + 1$

accès tableau n

additions n

comparaisons $2n + 1$

renvoyer 1

La mesure de complexité des algorithmes permet aussi de comparer la difficulté de problèmes à résoudre.

Définition

La complexité d'un problème est la complexité de l'algorithme le plus rapide résolvant ce problème.

Exemple

Le problème de recherche d'un élément dans un tableau trié (**P2**) est-il plus facile que le problème de recherche dans un tableau non trié (**P1**) ?

Un premier algorithme pour P2

Algorithme: rechSeq(**d** T[1..n], **d** e) :

bool

Données: T[1 ... n] tableau trié ↗ et
e

Résultat: Renvoie Vrai ssi $e \in T$

Variables : i entier ; $i \leftarrow 1$;

tant que $i \leq n$ et $T[i] < e$ **faire**

$i \leftarrow i + 1$;

si $i > n$ ou $T[i] > e$ **alors**

retourner Faux

sinon

retourner Vrai

- Taille du problème : $n + 1$

- **Meilleur cas** : $T[1] \geq e$

affectation 1

accès tableau 2

comparaisons 4

opérations bool. 2

renvoyer 1

en tout **10.**

- **Pire cas** : $T[n] < e$

n itérations

affectations $n + 1$

accès tableau n

additions n

comparaisons $2n + 2$

opérations bool. $n + 2$

Algorithme: $\text{rechDicho}(d \ T[1..n], d \ e) :$
Bool

Données: $T[1 \dots n]$ tableau trié ↗, e

Résultat: Renvoie $e \in T$

Variables : Deb, Fin, Mil entiers

$Deb \leftarrow 1 ; Fin \leftarrow n ;$

tant que $Deb \leq Fin$ **faire**

$Mil \leftarrow (Deb + Fin) \text{ div } 2 ;$

si $T[Mil] = e$ **alors retourner** *Vrai*

sinon si $T[Mil] \leq e$ **alors**

$Deb \leftarrow Mil + 1$

sinon

$Fin \leftarrow Mil - 1$

retourner *Faux*

Preuve

- Arrêt :
 $Fin - Deb \geq -1$ et décroît strictement à chaque itération.
- Invariant :
 $\forall i \in [1 \dots Deb[$
 $\forall j \in]Fin \dots n]$
 $T[i] < e < T[j]$

$Deb \leftarrow 1 ; Fin \leftarrow n ;$

tant que $Deb \leq Fin$ **faire**

$Mil \leftarrow (Deb + Fin) \text{ div } 2 ;$

si $T[Mil] = e$ **alors**

retourner *Vrai*

sinon si $T[Mil] \leq e$ **alors**

$Deb \leftarrow Mil + 1$

sinon

$Fin \leftarrow Mil - 1$

retourner *Faux*

Analyse de la complexité

- taille du problème : $n + 1$
- pire des cas : $e \notin T$;
- nb d'opérations hors boucle : 3
- complexité d'une itération : 8
- nb d'itérations : $\approx \log n$.

Complexité de l'algorithme de recherche dichotomique pour P2 : $\approx 4 + 8 \log n$.

Rôle des constantes

Dans la complexité d'un algorithme, les constantes additives et multiplicatives ont :

- **peu de sens** : par exemple l'exécution d'une opération booléenne est plus rapide que l'accès à un élément de tableau.
- **peu d'importance** : Si pour un même problème 2 algorithmes A et B ont les complexités $t^A(n) = n^2 + 1$ et $t^B(n) = 4.n + 3$, on préférera l'algorithme B même si $t^A(n) < t^B(n)$ pour $n < 5$.

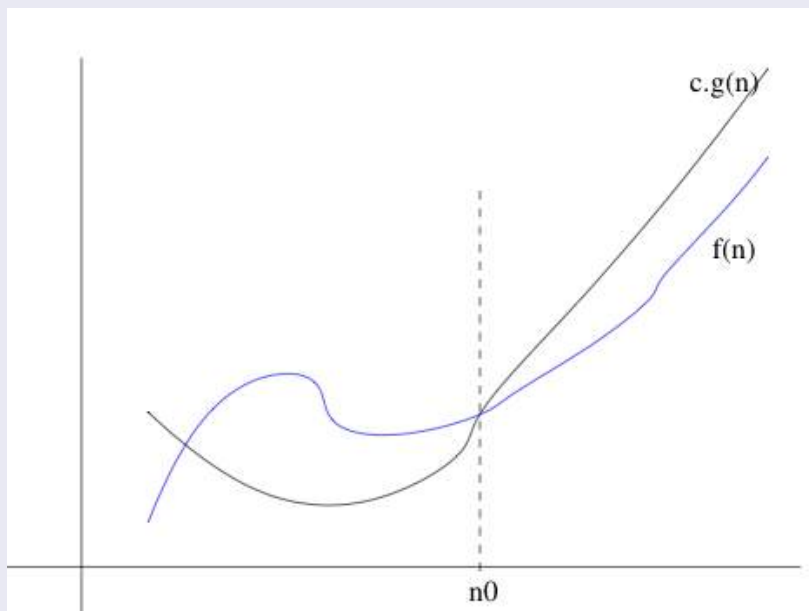
Ce qui nous intéresse

Un ordre de grandeur sur la complexité.

Ordre de grandeur

Définition

Pour 2 fonctions f, g : on note $f(n) \in O(g(n))$ si $\exists c > 0, \exists n_0$ tels que $\forall n > n_0, f(n) \leq c.g(n)$



Exemples

- $6n + 6 = O(n)$ (par exemple $c = 12$ et $n_0 = 1$).
- $(n + 1)^2 = O(n^2)$ (par exemple $c = 3$, $n_0 = 1$)
- $4n^3 + 10n^2 + 8 = O(n^3)$.
- Pour RechercheSeq : $t(n) = O(n)$

Simplification des calculs

Propriétés

$$O(f).O(g) = O(f.g)$$

$$O(f) + O(g) = O(f + g) =$$

$$O(\max(f, g))$$

Dans le pire des cas

ligne 1	$O(1)$
---------	--------

n itérations

Une itération	$O(1)$
---------------	--------

Conditionnelle	$O(1)$
----------------	--------

$$O(1+n.1+1)=O(n)$$

```
1  $i \leftarrow 1$  ;  
  tant que  $i \leq n$  et  $T[i] < e$  faire  
    |  $i \leftarrow i + 1$  ;  
  si  $i > n$  ou  $T[i] > e$  alors  
    | retourner Faux  
  sinon  
    | retourner Vrai
```

Algorithme de recherche dichotomique

```
Deb ← 1 ; Fin ← n ;
tant que Deb ≤ Fin faire
  | Mil ← (Deb + Fin) div 2 ;
  | si T[Mil] = e alors retourner Vrai
  | sinon si T[Mil] ≤ e alors
  | | Deb ← Mil + 1
  | sinon
  | | Fin ← Mil - 1
retourner Faux
```

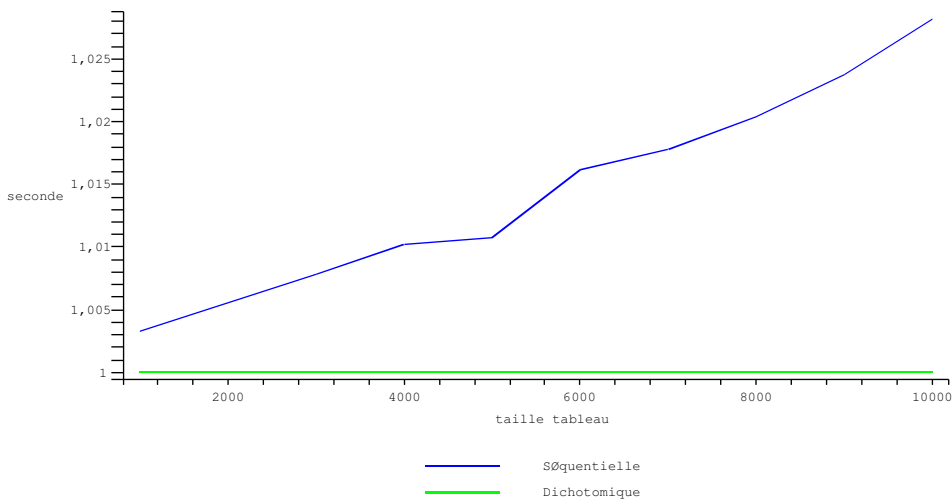
Calcul du nombre maximum d'itérations

- À chaque itération $Fin - Deb$ est divisé par 2. Soit Nbe le nombre d'éléments entre les indices Deb et Fin .
 - $Nbe_0 = n$
 - $Nbe_1 \leq \frac{n}{2}$
 - $Nbe_i \leq \frac{Nbe_{i-1}}{2} \leq \frac{n}{2^i}$
- Soit m le nombre d'itérations :
- $Nbe_{m-1} \geq 1$
 - $\frac{n}{2^{m-1}} \geq 1$
 - $m \leq \log_2(n) + 1$.

Complexité dans le pire des cas $t(n) \in O(\log_2 n)$.

chaque étape, on voit apparaître un **terme logarithmique** dans sa complexité.

$O(n)$ ou $O(\log n)$ est-ce que ça change quelque chose à son temps d'exécution ?



On retrouvera relativement souvent un tel terme dans la complexité des opérations accédant à une structure de données.

Des chiffres

Exemple

Temps d'exécution d'algorithmes de complexités différentes sur un ordinateur exécutant 1 inst^o / nano-seconde (10^{-9} s) :

taille instance	Complexité		
	n^2	n^3	2^n
1000	1 ms	1 s	4×10^{13} années
10^5	10 s	11,5 jours	-
10^6	16,6 mn	31,7 années	-

(l'âge de l'univers est estimé à 14 milliards d'années – 10^{10} a.)

Evolution de la taille de données qu'on peut traiter (même tps calc) si la vitesse d'exécution de l'ordinateur s'améliore :

	n^2	n^3	2^n
$\times 100$	$10n$	$1,61n$	$n + 6,64$

- Tris
 - Tri à bulles
 - Tri par insertion
- Types de Données Abstrait (TDA)
 - définitions
 - les Listes
 - les Piles
 - les Files
- Programmation en langage Java
 - implémentation d'un TDA
 - les exceptions

- Trier un ensemble de valeurs d'objets est une opération fondamentale en informatique.
 - une recherche dans un SGBD base de données ne peut être efficace que si les enregistrements sont triés (suivant une clé, un index).
 - Exemple : le *36 15 Qui c'est ?* = 10^8 informations de type (numéros de téléphones, noms, adresses) à retenir. Si triés, une recherche de numéro se fait en seulement ≈ 18 consultations de numéros par une recherche dichotomique de complexité $O(\log n)$:
⇒ une poignée de secondes si la base de données est stockée sur disque.
- Certains estiment qu'environ 20% du temps de calcul planétaire est dédié au tri.

- Trier des entiers ou des ensembles de valeurs complexes : **mêmes algorithmes** ! seule la **fonction de comparaison** de deux éléments change.
- De nombreuses méthodes de tri existent, nous en verrons 3 ou 4.
- Pour choisir parmi ces méthodes, il faut trouver un **compromis** entre simplicité de mise en oeuvre (implémentation) et efficacité (complexité).

Propriétés des méthodes de tri

- La **complexité** est un facteur déterminant dans le choix d'une méthode de tri.
- Un tri s'effectue **en place** s'il ne nécessite pas que les éléments triés soient mis dans une structure de données annexe de façon temporaire.
- Un tri est dit **stable** s'il ne change pas l'ordre dans lequel les informations ayant une même clé apparaissent initialement.

Soit T un tableau de n éléments (cases de 0 à $n - 1$)

Principes :

trie le tableau par échanges d'éléments.

Ce tri comporte $n - 1$ étapes, chacune étendant la zone triée du tableau.

Chaque étape consiste à faire remonter la plus petite valeur de la zone non-triée en tête de cette zone.

Tris à bulles

Un exemple :

Principes :

trie le tableau par échanges d'éléments.

Ce tri comporte $n - 1$ étapes, chacune étendant la zone triée du tableau.

Chaque étape consiste à prendre la 1^{ère} valeur de la zone non-triée et à l'**insérer** dans la zone triée par décalage d'éléments.

Tri par insertions

Algorithme: `Tri-Insertion` (T tableau de n entiers, i entier)

Données: T un tableau de nombres à trier par ordre croissant,
 i un entier correspondant à un indice du tableau

Résultat: trier la partie $T[0]..T[i]$ du tableau

(Appel initial de l'algo : `Tri-Insertion(T, n - 1)`)

// test d'arrêt de la récursivité

si $i > 0$ **alors**

 // a) Tri récursivement $T[0] \dots T[i - 1]$

`Tri-Insertion(T, i - 1)`

$x \leftarrow T[i]; j \leftarrow i - 1$

 // b) Insère x dans $T[0]..T[j]$ en conservant l'ordre croissant

tant que ($j \geq 0$) *et-alors* ($T[j] > x$) **faire**

$T[j + 1] \leftarrow T[j]; j \leftarrow j - 1$

$T[j + 1] \leftarrow x$

Complexité : $O(n^2)$, mais avec une plus faible constante que le tri à bulles

Types de données abstraits

Un **type de données abstrait** (TDA) est défini par sa “**signature**” ou “**interface**”, composée de :

- une déclaration des ensembles définis et utilisés
- une description fonctionnelle des opérations
- une description axiomatique de la sémantique des opérations

- non-ambiguïté des définitions, évitant des interprétations divergentes
- preuve de la validité des opérateurs définis
- preuve de propriétés sur le fonctionnement de la structure de données (attendus d'un projet avant même sa réalisation)
- l'indépendance vis à vis de l'implémentation, ce qui permet le développement successif et même simultané des différentes parties d'un projet informatique par des équipes de programmeurs différents.

L'inconvénient de la spécification formelle d'une structure de données est principalement la difficulté de sa rédaction (*abstraction de haut niveau*).

Le TDA Ensemble

Déclaration

Ensemble **utilise** Elément, booléen

$\emptyset \in \text{Ensemble}$

Description fonctionnelle

est-vide ?	:	Ensemble	→	booléen
\in	:	Ensemble × Elément	→	booléen
ajouter	:	Ensemble × Elément	→	Ensemble
enlever	:	Ensemble × Elément	→	Ensemble
union	:	Ensemble × Ensemble	→	Ensemble

Description Axiomatique

- 1 est-vide ?(\emptyset) = vrai
- 2 $\forall x \in \text{Elément}, \forall e \in \text{Ensemble}, \text{est-vide ?}(\text{ajouter}(e, x)) = \text{faux}$
- 3 $\forall x \in \text{Elément}, x \in \emptyset = \text{faux}$
- 4 $\forall x \in \text{Elément}, \forall e \in \text{Ensemble},$
 $x = y \Rightarrow y \in \text{ajouter}(e, x) = \text{vrai}$
 $x \neq y \Rightarrow y \in \text{ajouter}(e, x) = y \in e$
- 5 $\forall x \in \text{Elément}, \forall e \in \text{Ensemble},$
 $x = y \Rightarrow y \in \text{enlever}(e, x) = \text{faux}$
 $x \neq y \Rightarrow y \in \text{enlever}(e, x) = y \in e$
- 6 $\forall x \in \text{Elément}, \forall e \in \text{Ensemble},$
 $x \notin e \Rightarrow \exists e' \in \text{Ensemble}, e' = \text{enlever}(e, x)$
- 7 $\forall x \in \text{Elément}, \forall e, e' \in \text{Ensemble}, x \in \text{union}(e, e')$
 $\Rightarrow x \in e \text{ ou } x \in e'$

Traduction des TDA en Java

- Pour pouvoir utiliser un TDA comme un outil dans des programmes, il faut l'implémenter
- Java dispose essentiellement de deux mécanismes pour implémenter un TDA :
 - les **interfaces** pour réaliser la description fonctionnelle des opérations du TDA
 - les **exceptions** afin d'assurer que l'implémentation vérifie la description axiomatique du TDA.
- Une interface est une façon de déclarer ce qu'une classe doit pouvoir rendre comme service quand elle implémente ce TDA.
Exemple des scouts : badges pour dire qu'ils savent construire une cabane, faire un feu, etc.

- L'**interface** contient uniquement des déclarations de méthodes avec leurs signatures.
- Une classe qui **implémente** le TDA doit réaliser chacune des méthodes déclarées dans l'interface.
- Séparer l'interface de l'implémentation permet de proposer **plusieurs implémentations pour réaliser un TDA**, chacune ayant en général un intérêt quand une certaine opération est utilisée plus fréquemment que les autres (faible complexité pour cette opération).

Attention

Ne pas confondre la notion d'interface/implémentation (déclaration de services qu'on peut rendre) avec la notion d'héritage (spécialisation).

Traduction Java du TDA Ensemble

```
public interface Ensemble {
    public boolean estVide (); // une opération
    public boolean appartient (Object x); // une autre
    public void ajouter (Object x);
    public void enlever (Object x) throws ElementAbsentExc
    public Ensemble union (Ensemble x);
}
```

Utilisation du mécanisme des exceptions

notez qu'une des méthodes lève une exception afin que le TDA respecte l'axiome 6. Celle-ci est implémentée au minimum de la façon suivante :

```
public class ElementAbsentException extends Exception
public ElementAbsentException () {
```

```

public class EnsembleVecteur implements Ensemble {
    Vector T; // structure de données choisie
    public EnsembleTableau() { // constructeur
        T = new Vector();
    }
    public boolean estVide() { // implémentation dépendante
        return T.isEmpty(); // de la s.d. choisie
    }
    public boolean appartient(Object x) {
        if (...) return true; else return false;
    }
    public void ajouter(Object x) {
        if (! t.contains(x)) T.add(x);
    }
    public void enlever(Object x) { T.remove(x); }
    .....
}

```

implémentation et polymorphisme

```

public interface PeutCouper { public void coupe(); }

public class CouteauSuisse implements PeutCouper { ..

public class Bricoleur {
    ....
    public void main() {
        Bricoleur McGyver = new Bricoleur();
        CouteauSuisse UnCouteau = new CouteauSuisse();
        PeutCouper Coupeur = UnCouteau; // upcast
        Coupeur.coupe(); // pas de pbm
        UnCouteau.coupe(); // pas de pbm
        ((CouteauSuisse) Coupeur).coupe(); // downcast
    }
}

```

PeutCouper.java

```
public interface PeutCouper {  
    public void coupe();  
}
```

couteau.java

```
public class couteau implements PeutCouper {...}
```

scie.java

```
public class scie implements PeutCouper {...}
```

bricoleur.java

```
PeutCouper[] tableau = new PeutCouper[10];  
tableau[0] = new couteau();  
tableau[1] = new scie();
```