Structures de Données

Structures Statiques

Contenu

- Structures Statiques
  - Definition
  - Cardinalité
  - Tableaux
  - Ensembles
  - Listes

Introduction

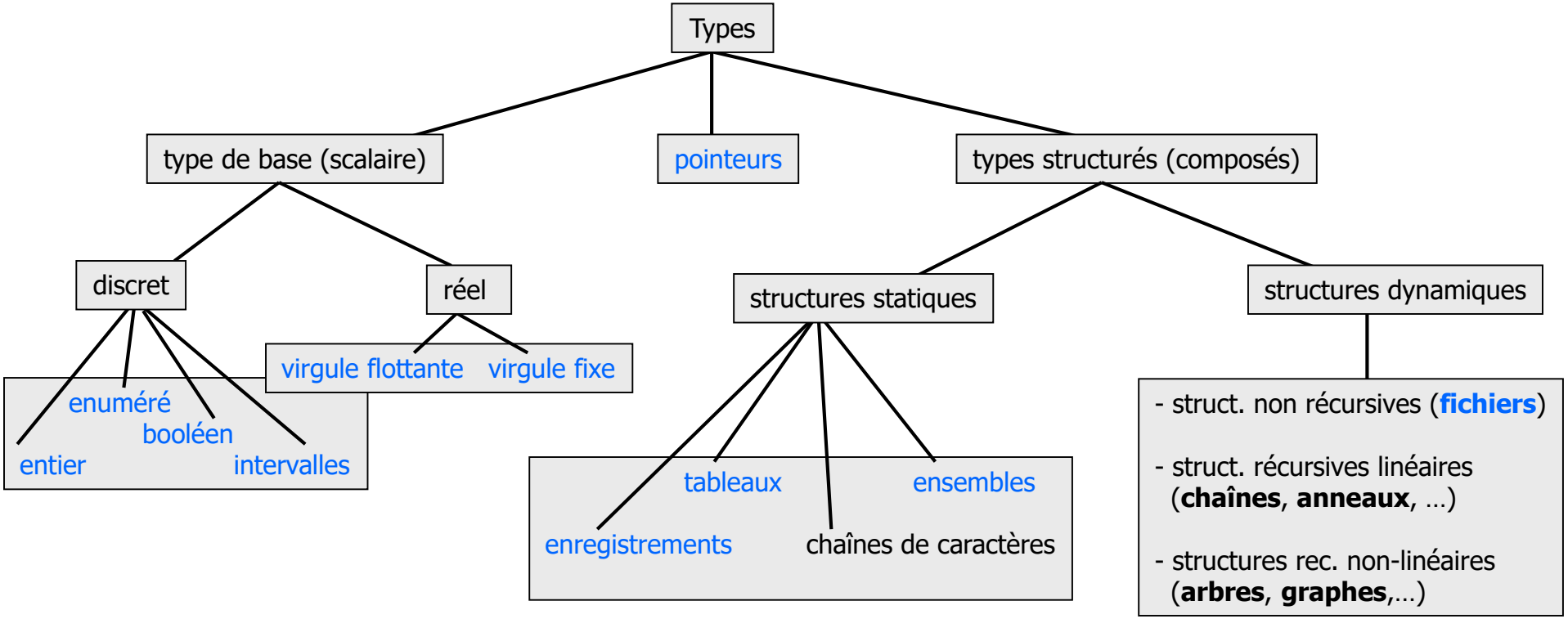
- Une structure est dite "statique" si le volume mémoire utilisé par une variable ayant cette structure est connu dès le début de l'exécution du programme

- Une structure statique est caractérisée par une cardinalité finie (cf plus loin)

- Dans ce contexte, une structure "statique" est opposée à une structure "dynamique" (prochain chapitre)

- Dans beaucoup de cas, une structure statique correspond à une version "initiale" de la structure qui peut éventuellement devenir dynamique

Taxonomie des types



Cardinalité

- La cardinalité d'un type T est le nombre de valeurs distinctes que peut prendre une variable de type T.
- Pour calculer la cardinalité d'un nouveau type, il faut savoir se ramener à ces types de base
- Tout type statique est composé de types dont la cardinalité est fixée par le système
- Une structure statique est caractérisée par une cardinalité finie

Cardinalité: exemple

La cardinalité du type entier est

card(**entier**) = 2***MAX_ENTIER** + 1

car une variable de type entier peut prendre les valeurs:

[-MAX_ENTIER,…,-1,0,1,2,…, MAX_ENTIER]

Attention: ne pas confondre la cardinalité et la taille

Type "Tableau"

- Un tableau est une structure permettant de stocker une valeur d'un type de base pour chaque valeur d'index de l'intervalle donné
- Pour définir le tableau, on a besoin de sa taille ou de la liste des valeurs d'index ou d'un intervalle de valeurs
- Les opérations associées sont
  - Le dépôt d'une valeur à une valeur d'index
  - La consultation de la valeur stockée à une valeur d'index
- Pour que ces opérations se passent bien, il faut que la valeur d'index soit valide
  - On rajoute une fonction de diagnostic

TAD: Tableau

Tableau
→ Conserver pour chaque valeur d'indice une valeur de type de base associée
→ Une seule valeur du type de base peut être associée à chaque valeur d'indice.

Déclaration:
→ La déclaration spécifie un intervalle et un type de base

Opérations:

1. **Création, initialisation, construction**

2. **Modification**
   a. Associer une valeur du type de base à une certaine valeur d'indice.

3. **Destruction**

4. **Accès**
   a. Fournir la valeur de type de base associée à une certaine valeur d'indice.

   b. En fonction de la déclaration, diagnostiquer la validité de l'indice fourni.

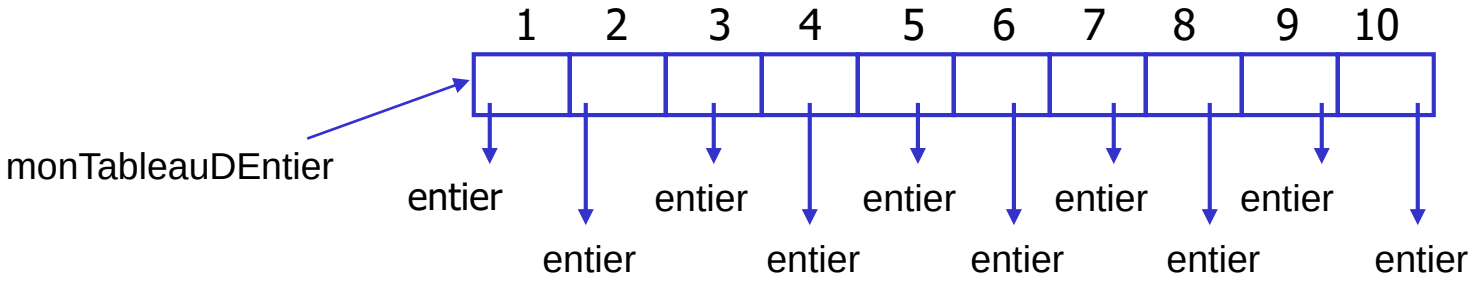
- On veut traiter les erreurs éventuelles
  - Exple: mauvais indice

⇒ On déclare des cas d'**exceptions**

⇒ Cas dans lesquels la structure déclenchera une erreur à gérer

Exceptions:
- Associer une valeur à un indice hors de l'intervalle déclaré (2.a)
- Accéder à une valeur correspondant à un indice hors de l'intervalle déclaré (4.a)

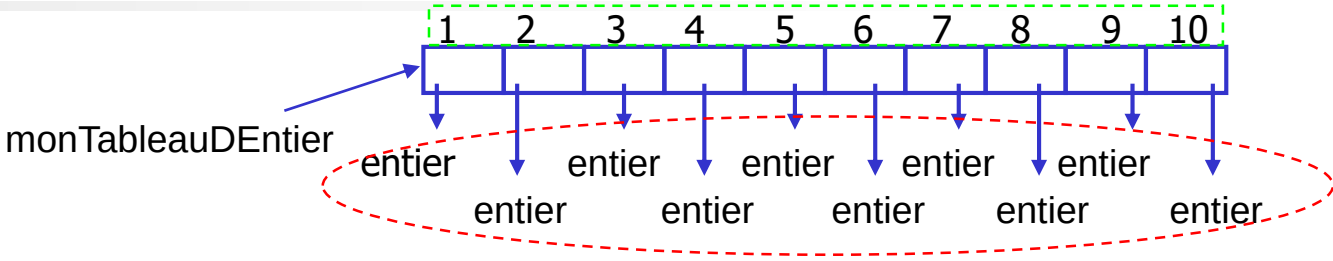
Tableau : Représentation



- Le tableau est homogène
  - ⇒ on a un seul type de base

- On a besoin d'un intervalle d'indices
  - En général de 1 (ou 0) à *N*
    - ⇒ On déclare *N*
  - On peut vouloir spécifier un intervalle

Tableau : Utilisation



- Pseudo-code

```
entier[10] : monTableauDEntier // l'indice va de 1 à 10
entier: i
monTableauDEntier[1] <- 3
monTableauDEntier[i] // gestion des erreurs selon i
```

- Pascal

```
monTableauDEntier : array[1..10] of integer;
i : integer
monTableauDEntier[1] := 3;
monTableauDEntier[i]; // gestion des erreurs selon i
```

TypeDeBase

Intervalle

- C

```
int monTableauDEntier[10]; // l'indice va de 0 à 9
int i;
monTableauDEntier[0] = 3;
monTableauDEntier[i]; // risque de dépassement mémoire
```

Cardinalité d'un tableau

Rappel: la cardinalité d'un type T est le nombre de valeurs distinctes que peut prendre une variable de type T

La cardinalité d'un type T basé sur un type de base T_0 et un intervalle d'indice I est

$$card(T) = card(T_o)^{Card(I)}$$

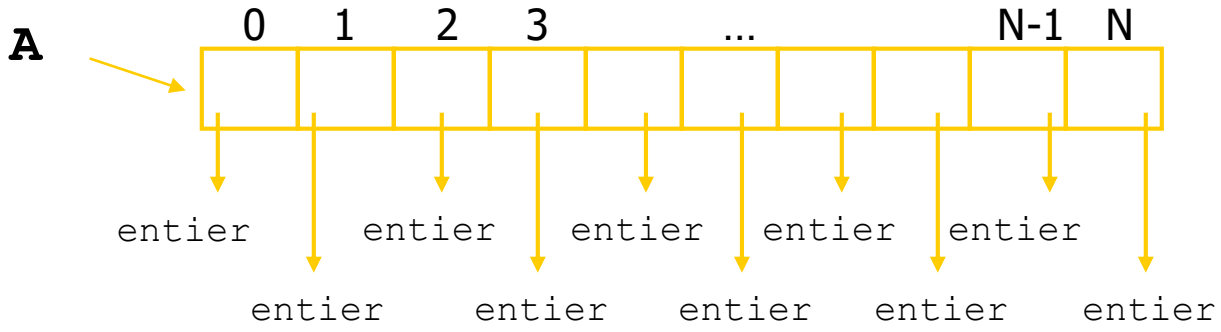
Ne pas confondre Cardinalité du type et Taille de la variable:

$$taille(A)=card(I)$$

On peut (abusivement) parler de la cardinalité d'une variable (au lieu de la cardinalité du type).

Exemple

Tableau de N entiers



I = 1..N
T_o = entier

entier[N]: A

$T_0 \in \{-$ **MAX_ENTIER**, ... ,0, ..., + **MAX_ENTIER**$\}$

card(T_o)=(2*MAX_ENTIER)+1

card(I) = N

card(entier[N])=[(2* MAX_ENTIER)+1]^N

A noter: **taille(entier[N])=N**

Les tableaux multidimensionnels

Un tableau peut être multidimensionnel.

- Déclaration:

```
type
   TMatrice: T°[MAX_LIGNE,MAX_COLONNE]
```

- Sélecteur:

```
TMatrice: m
entier: i,j
m[i,j]
```

- Cardinalité:

$$card(\texttt{TMatrice}) = card(\texttt{T}_\circ)^{\texttt{MAX\_LIGNE*MAX\_COLONNE}}$$

A noter: Le nombre de dimensions d'un tableau n'est, dans la plupart des langages, pas limité

Les tableaux de tableaux

Un tableau multidimensionnel peut être vu comme un tableau de tableaux

Déclaration:

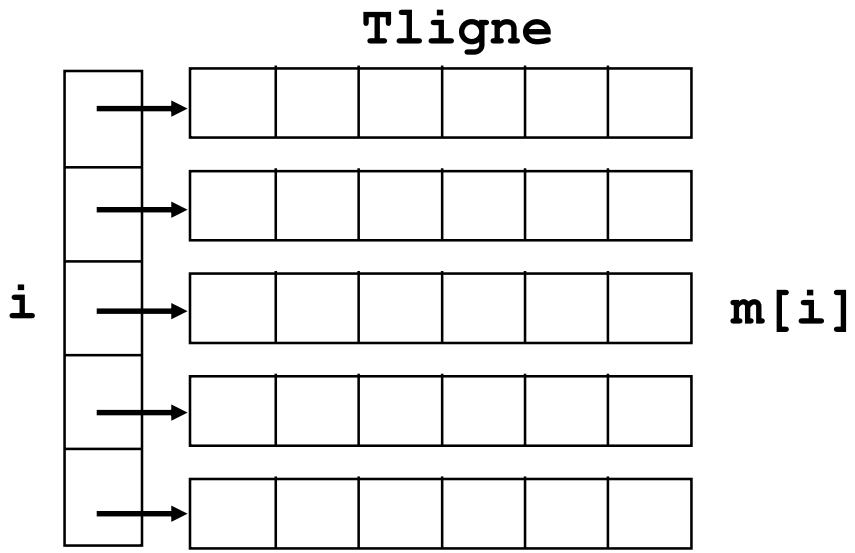
```
type
    Tligne: T°[MAX_LIGNE]
    TMatrice: Tligne[MAX_COLONNE]
```

- Sélecteur:

```
TMatrice: m
entier: i,j
m[i][j]
```

`m[i]` est de type `TLigne`

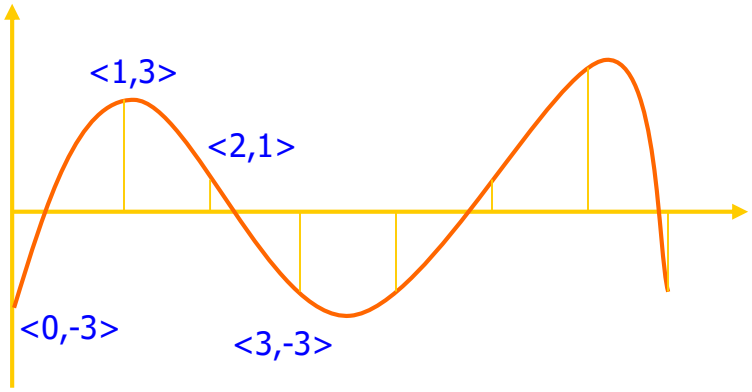
⇒ La cardinalité ne change pas



Les tableaux d'enregistrements

- Déclaration:

```
type
   TPoint struct:
            entier: x,y
   TSuite: TPoint[MAX_POINT]
Tsuite: uneSuiteDePoints
```



- Sélecteur:
  - L'abscisse du premier point de la suite

    `uneSuiteDePoints[1].x`
  - L'ordonnée du premier point de la suite

    `uneSuiteDePoints[1].y`

- Cardinalité:

  $card(\texttt{TSuite}) = card(\texttt{entier})^{2*\texttt{MAX\_POINT}}$

Les enregistrements de tableaux

- Déclaration:

```
const L_MAX
type
  TString: car[L_MAX]
  TAdulte struct:
   TString: nom
   TString[3]: prenom
TAdulte[10]: listeDePersonnes
```

- Sélecteur: (deuxième prénom de la première personne de la liste) :

```
listeDePersonnes[1].prenom [2]
```

- Cardinalité:

$$Card(\texttt{TAdulte}) = card(car)^{4*L\_MAX}$$

Type "Ensemble"

- Un "Ensemble" est une structure permettant d'indiquer la présence ou l'absence de chacune des valeurs possibles du type de base dans cet ensemble.
- Pour définir un ensemble, on a besoin de spécifier un type de base dont les valeurs sont en nombre fini
- Les opérations associées sont
  - Création d'un Ensemble vide (tous les indicateurs sont négatifs)
  - Ajout et suppression de valeurs
  - Opérations ensemblistes (intersection, union,…)
- On exclu les opérations portant sur des ensembles définis avec des types de base différents

Opérations sur les ensembles

Les opérations sur les ensembles sont:

- L'affectation (:=)
- La réunion (+)
- L'intersection (*)
- La différence (-)
- La comparaison (=, <>)
- L'inclusion (<=)
- Le test d'appartenance d'une valeur de type de base (in)

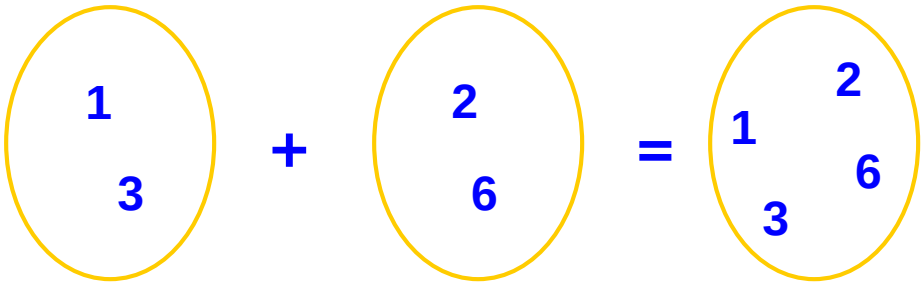
A noter: L'expression a $\notin$ S sera notée: not (a in S)

Opérateur de réunion (+)

Rappel: la réunion de deux ensembles est l'ensemble formé des éléments appartenant au moins à l'un des deux ensembles.

- Exemple:

A	B	A + B
[1, 3]	[2, 6]	[1, 2, 3, 6]
[1, 3]	[3, 4]	[1, 3, 4]
[1, 5]	[1, 5, 6]	[1, 5, 6]

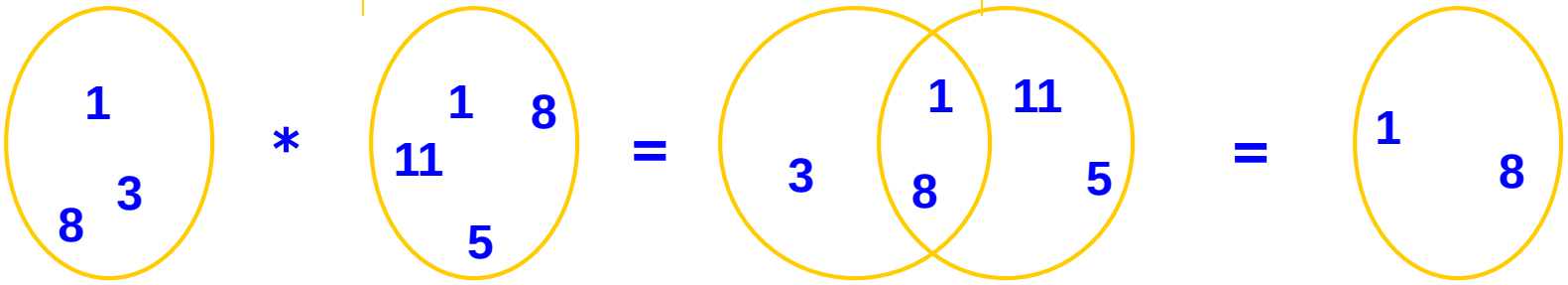


Opérateur d'intersection (*)

Rappel: l'intersection de deux ensembles est l'ensemble formé des éléments appartenant simultanément aux deux ensembles.

- Exemple:

A	B	A * B
[1, 3, 8]	[1, 5, 8, 11]	[1, 8]
[1, 5]	[1, 3, 5, 8]	[1, 5]
[1, 5]	[6, 9]	[ ]



Opérateur de différence (-)

Si A et B sont deux ensembles, A-B est l'ensemble formé des éléments appartenant à A et n'appartenant pas à B.

« On retire de A tout ce qui appartient à B »

- Exemple:

A	B	A - B
['a', 'c', 'f', 'k']	['c', 'f']	['a', 'k']
['a', 'b', 'c']	['d', 'e']	['a', 'b', 'c']
['f', 'g']	['a', 'f' , 'g', 'm']	[ ]

Opérateurs de comparaison

Ils s'appliquent sur des ensembles formés sur les mêmes type de base, ce sont:

= (égalité)	A = B si A et B sont formés des mêmes éléments
<= (inclusion)	A <= B si A est inclus dans B (on utilise aussi « in »)
>=	A >= B si B est inclus dans A
<>	A <> B si A et B n'ont aucun élément en commun

- Exemples:

```
voyelles <- [ 'a', 'e', 'i', 'o', 'u', 'y' ];
     test <- [ 'a', 'i', 'o' ];

si (voyelles = test) alors …          // faux

si (test <= voyelles ) alors …        //vrai

si (voyelles >= test ) alors …        // vrai

si (test in voyelles) alors …         // vrai
```

TAD: Ensemble

Ensemble

→ Indiquer la présence ou l'absence de chacune des valeurs possibles du type de base dans cet ensemble.

Déclaration:

→ La déclaration spécifie un type de base

Opérations:

1. **Création, initialisation, construction**
   a. L'ensemble est vide a priori
2. **Modification**
   a. Un ensemble: affectation, suppression
   b. 2 ensembles: réunion, intersection, différence
3. **Destruction**
4. **Accès**
   a. 1 ensemble: Test d'appartenance
   b. 2 ensembles: Test d'inclusion, comparaison

Exceptions:

• Toute opération sur deux ensembles ne portant pas sur le même type

- Comparaison (4.b)
- Réunion (2.b)
- Intersection (2.b)
- Différence (2.b)

Exemple d'implémentation

On stocke les valeurs effectivement présentes dans l'ensemble en les marquant avec un booléen

```
type TSet:
    TTypeDeBase[CARDINALITE_DU_TYPEDEBASE]:valeurPossible
    booleen[CARDINALITE_DU_TYPEDEBASE]: presenceValeur
```

- L'initialisation se fera par le remplissage du tableau **`valeurPossible`** et la mise à **`FAUX`** de **`presence`**
- L'ajout d'une **`valeur`** se fera par la recherche de son indice dans **`valeurPossible`** et la mise à **`VRAI`** du meme indice dans **`presenceValeur`**
- Les opérations d'intersection, union et différence se feront par simples opération booléennes sur **`presence`**

⇒ Comment spécifier le type de base?

⇒ *En pratique: utilisation de Templates*

Interface

TAD: Ensemble ➔ Interface **TSet**

```
fonction TSet init()<TTypeDeBase>
procedure add(TSet set, TTypeDeBase valeur)
procedure remove(TSet set, TTypeDeBase valeur)
fonction booleen contains(TSet set, TTypeDeBase valeur)
fonction TSet intersect(TSet setA, TSet setB)
fonction TSet join(TSet setA, TSet setB)
fonction TSet difference(TSet setA, TSet setB)
```

Extra (*convenience functions*)

```
fonction booleen isEmpty(TSet set)
fonction entier size(TSet set)
```

Exemple: *Set* Java

Method Summary	
boolean	**add**(E o) Adds the specified element to this set if it is not already present (optional operation).
boolean	**addAll**(Collection<? extends E> c) Adds all of the elements in the specified collection to this set if they're not already present (optional operation).
void	**clear**() Removes all of the elements from this set (optional operation).
boolean	**contains**(Object o) Returns `true` if this set contains the specified element.
boolean	**containsAll**(Collection<?> c) Returns `true` if this set contains all of the elements of the specified collection.
boolean	**equals**(Object o) Compares the specified object with this set for equality.
int	**hashCode**() Returns the hash code value for this set.
boolean	**isEmpty**() Returns `true` if this set contains no elements.
Iterator<E>	**iterator**() Returns an iterator over the elements in this set.
boolean	**remove**(Object o) Removes the specified element from this set if it is present (optional operation).
boolean	**removeAll**(Collection<?> c) Removes from this set all of its elements that are contained in the specified collection (optional operation).
boolean	**retainAll**(Collection<?> c) Retains only the elements in this set that are contained in the specified collection (optional operation).
int	**size**() Returns the number of elements in this set (its cardinality).
Object[]	**toArray**() Returns an array containing all of the elements in this set.
<T> T[]	**toArray**(T[] a) Returns an array containing all of the elements in this set; the runtime type of the returned array is that of the specified array.

Type "Liste"

- Une liste permet de stocker un nombre indéterminé d'éléments d'un type de base dans un certain ordre. Un élément peut être lui-même une liste. L'ordre de placement des éléments dans la liste peut dépendre de la chronologie d'insertion, du contenu des éléments insérés ou d'un quelconque autre critère fixé par l'utilisateur

- La définition ci-dessus est récursive car le terme à définir (la liste) est utilisée à l'intérieur de la définition:
  - Cela permet de définir et de décrire toutes les structures de listes possibles.

- La notion de liste est similaire à celle utilisée dans LISP (*cf* cours Algorithmique)
  - Pour la carté, on se restreint à une liste composée d'éléments identiques
  - La construction se fait en fusionnant une liste à une nouvelle tête de liste

Liste

Exemples de notation:

- A=() liste vide (longueur 0).
- B=(a,(b,c)) liste de longueur 2, le 1er élément est un atome «a», le 2ème élément est une liste «(b,c)».
- C=(B,B,()) liste de longueur 3 dont les 2 premiers éléments sont identiques et correspondent chacun à la liste B, le 3ème élément est la liste vide.
- D=(a,D) liste récursive de longueur 2. D correspond à la liste infinie (a, (a, (a, (…)))). Cette possibilité de déclarer des listes récursives est souvent utilisée pour représenter la grammaire d'un langage de programmation.

Opérations sur les listes

- Les différentes opérations sur les listes sont:
  - Insertion: en début, en fin (cons).
  - Extraction: du premier élément (car), de tout sauf le premier élément (cdr).
  - Construction: combinaison d'un élément et d'une liste.
  - Comparaison: = ou <>
  - Prédicat de liste vide: revient à comparer à la liste vide.

Opérations sur les listes

- La combinaison de «car» et «cdr» permet d'isoler n'importe quel élément d'une liste. Pour abréger, on ne note souvent que les lettres du milieu (a ou d), le tout entouré de c et r.

  car(cdr(cdr(L))) peut s'abréger caddr(L)

- L'opération de construction consiste à créer une nouvelle liste à partir d'un élément et d'une liste de façon à avoir la relation suivante:

  L = cons(car(L), cdr(L))

- La comparaison de deux listes tient compte des niveaux d'imbrication des sous-listes ainsi que du nombre d'éléments. Ainsi, (a) est différent de (a,()) et de ((a)), de même (()) est différent de ().

TAD: Liste

Liste

→ Stocker une suite finie d'éléments d'un type de base

→Une liste est formée d'une **tête** et d'un **reste** étant lui-même une liste

Déclaration:

→ La déclaration spécifie un type de base

Opérations:

1. **Création, initialisation, construction**
   a. Une liste est vide a priori
2. **Modification**
   a. Insérer un élément en tête de liste
   b. Insérer un élément en fin de liste
   c. Fusion d'un élément et d'une liste
3. **Destruction**
   a. Supprimer la tête de liste
   b. Supprimer le reste de la liste
4. **Accès**
   a. Indiquer si la liste est vide
   b. Indiquer la longueur de la liste
   c. Indiquer l'égalité de deux listes
   d. Accès à la tête de liste
   e. Accès au reste de la liste

Exceptions:

- Accès au reste d'une liste vide
  - Accès (4.e)
  - Suppression (3.b)
- Toute opération concernant une liste et un élément de type incompatible (2.)

Structures statiques: Résumé

- La cardinalité d'un type est le nombre de valeurs possibles d'une variable de ce type
- Une structure statique possède une cardinalité finie
- On a caractérisé plusieurs structures statiques comme TAD
  - Les tableaux et les tableaux multidimensionnels
  - Les ensembles
  - Les listes
- Tous ces types permettent de manipuler des groupes de données
  - Tableau: ordonné, redondant
  - Ensemble: non-ordonné, non-redondant
  - Liste: structuré
- On va compléter ces structures avec les structures dynamiques