

LU2IN006

Cours 5 - Structure de données non linéaires

“Structures de données”

Pierre-Henri Wuillemin

2022-2023

# Structures de données linaires (SDL)

LU2IN006

## 1. SD(N)L

Arbre  
Graphe

## 2. Relations en UML

## 3. Implémentation

Aggrégation  
Composition  
Composition interne

## 4. SDNL en C

Arbre  
Graphe orienté

### Structure de données linéaire

Une structure d'aggrégation de données est dite linéaire si, structurellement, il existe dans l'aggrégat un premier et un dernier élément ainsi qu'une fonction 'suivant' permettant d'itérer univoquement sur l'ensemble des éléments.

- un tableau est linéaire,
- une liste est linéaire,
  - une pile est linéaire,
  - une file est linéaire,
- une table de hashage est linéaire, etc.

La complexité des algorithmes dans les SDL est donnée en fonction du nombre d'éléments dans la structure.

## LU2IN006

### 1. SD(N)L

Arbre

Graphe

### 2. Relations en UML

### 3.

### Implémentation

Aggrégation

Composition

Composition interne

### 4. SDNL en C

Arbre

Graphe orienté

# Structure de données non linéaire (SDNL)

LU2IN006

Un tas est linéaire peut être codé par une structure linéaire.

## SDNL prototype 1 : arbre

Une structure de données en arbre est une organisation récursive d'un ensemble d'éléments selon une partition :  $\{x\} \times \prod_i K_i$  où chaque  $K_i$  a une structure d'arbre.

- Le  $\{x\}$  de la partition principale est appelé la racine.
- Les arbres à un seul élément sont appelé les feuilles.
- Un tas est un arbre qui isole le plus grand élément et une partition en 2 sous-ensembles des éléments restants, chacun organisé en tas (c'est un arbre mais particulier).
- Un document est décomposé en paragraphes, eux-même décomposés en phrases, elles-même décomposées en mot, etc...
- Dans un arbre, il y a un premier élément.
- Dans un arbre, il n'y a pas de dernier (sauf?).
- Dans un arbre, la complexité d'un algorithme peut dépendre :
  - du nombre de nœuds (les éléments),
  - de la hauteur de l'arbre,
  - du nombre de feuilles de l'arbre, etc

1. SD(N)L

Arbre

Graphe

2. Relations en UML

3. Implémentation

Aggrégation

Composition

Composition interne

4. SDNL en C

Arbre

Graphe orienté

# SDNL (2)

LU2IN006

## 1. SD(N)L

Arbre

Graphe

## 2. Relations en UML

## 3.

Implémentation

Aggrégation

Composition

Composition interne

## 4. SDNL en C

Arbre

Graphe orienté

### SDNL prototype 2 : graphe

Un structure de donnée en graphe est une représentation d'un ensemble d'éléments et d'une relation par paire dans cet ensemble.

- Un ensemble de villes, la relation entre 2 villes  $A$  et  $B$  est : il y a une route directe entre  $A$  et  $B$ .
- Un ensemble d'étudiants, la relation entre 2 étudiants  $A$  et  $B$  est :  $A$  et  $B$  sont dans le même groupe de TD en LI213.
- Un ensemble d'entier, la relation entre 2 entiers  $A$  et  $B$  est :  $A < B$ , etc.
- Dans un graphe, il n'y a ni premier ni dernier élément.
- Dans un graphe, la complexité d'un algorithme peut dépendre :
  - du nombre de nœuds (les éléments),
  - du nombre d'arcs (le nombre de paires en relation),
  - du diamètre du graphe (le plus long chemin du graphe),
  - de la *treewidth* du graphe ( $\approx$  le plus grand nombre de voisins pour un nœud du graphe), etc.

# Digression UMLesque : Aggrégation/Composition

LU2IN006

## 1. SD(N)L

Arbre  
Graphe

## 2. Relations en UML

3. Implémentation  
Aggrégation  
Composition  
Composition interne

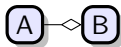
## 4. SDNL en C

Arbre  
Graphe orienté

**Problème** : Comment représenter deux entités en relation ?

- Le **cheval** porte le **cavalier**,
- L'**ordinateur** contient un **microprocesseur**,
- Le **capitaine** donne des ordres aux **soldats**,
- La **voiture** a 4 **roues**, etc.
- Le **bus** contient des **passagers**.

Ces relations logiques sont parfois symétrique mais, lorsque l'on doit énoncer la relation en français, on doit faire un **choix asymétrique** dans la représentation **B [verbe transitif] A**.



On pourra noter ainsi la relation qui devient non-symétrique

# Digression UMLesque : Aggrégation/Composition

LU2IN006

## 1. SD(N)L

Arbre  
Graphe

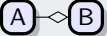

## 2. Relations en UML

3. Implémentation  
Aggrégation  
Composition  
Composition interne

## 4. SDNL en C

Arbre  
Graphe orienté

### Aggrégation/Composition

Une aggrégation  tout comme une composition  est une relation non symétrique entre  $A$  et  $B$  indiquant un couplage fort entre des valeurs de type  $A$  et  $B$ , voire une relation d'appartenance (élément/ensemble).

- Une composition indique qu'un élément de type  $B$  nécessite un élément de type  $A$  alors qu'une aggrégation est plus souple.
- La composition indique également l'exclusivité de la ressource de type  $A$  : elle ne peut être partagée.

Le choix entre aggrégation et composition dépend

- de l'interprétation que l'on donne à la relation
- mais aussi d'un choix d'implémentation : en effet, cela implique que  $B$  contienne un lien vers  $A$ .

# Digression UMLesque : Aggrégation/Composition

LU2IN006

## 1. SD(N)L

Arbre  
Graphe

## 2. Relations en UML

### 3.

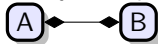
## Implémentation

Aggrégation  
Composition  
Composition interne

## 4. SDNL en C

Arbre  
Graphe orienté

Si l'on désire créer une relation symétrique entre  $A$  et  $B$ .



On s'éloigne de UML : cette représentation (et les suivantes) n'existe que pour ce cours. UML utiliserait des *arités* sur les relations.

On peut utiliser deux références croisées, qu'elles soient des aggrégations ou des compositions :





# Implémentation d'une aggrégation en C



: Lien faible entre les 2 entités.

Un processeur :

```
1 typedef struct {  
2     int vitesse;  
3 } processeur;
```

Un ordinateur :

```
1 typedef struct {  
2     float prix;  
3     processeur* proc;  
4 } ordi;
```

Création (un processeur étant donné) :

```
1 ordi* cree_ordinateur(float prix, processeur* p) {  
2     ordi* o=(ordi *)malloc(sizeof(ordi));  
3     o->prix=prix;  
4     o->proc=p;  
5     return o;  
6 }
```

Utilisation :

```
1 if (o!=NULL) {  
2     if (o->proc!=NULL) {  
3         o->proc->vitesse=1;  
4     }  
5 }
```

Suppression :

```
1 free(o);  
2 /* somewhere else */  
3 free(p);
```

Différents cycle de vie.

# Implémentation d'une composition en C

processeur

ordinateur

: Lien fort  $\Rightarrow$  délégation de création.

Un processeur :

```
1 typedef struct {  
2     int vitesse;  
3 } processeur;
```

Un ordinateur :

```
1 typedef struct {  
2     float prix;  
3     processeur* proc;  
4 } ordi;
```

Création (délégation de création) :

```
1 ordi* cree_ordinateur(float prix, int vitesse) {  
2     ordi* o=(ordi *)malloc(sizeof(ordi));  
3     processeur *p=(processeur *)malloc(sizeof(processeur));  
4     o->prix=prix;  
5     o->proc=p;  
6     p->vitesse=vitesse;  
7     return o;  
8 }
```

Utilisation :

```
1 if (o!=NULL) {  
2     o->proc->vitesse=1;  
3 }
```

Suppression :

```
1 free(o->p);  
2 free(o);
```



Seule la **logique des fonctions** est différente !

Risque de corruption de l'hypothèse d'existence du processeur.



# Implémentation d'une composition en C (2)

LU2IN006

processeur

ordinateur

: Lien TRES fort  $\Rightarrow$  modification de la structure.

Un ordinateur et son processeur (**Composition interne**)

```
1  ordi* cree_ordinateur(float prix,int vitesse) {  
2      ordi* o=(ordi *)malloc(sizeof(ordi));  
3  
4      o->prix=prix;  
5      o->proc.vitesse=vitesse;  
6      return o;  
7  }
```

Utilisation :

```
1  if (o!=NULL) {  
2      o->proc.vitesse=1;  
3  }
```

Suppression :

```
1  free(o);
```

1. SD(N)L

Arbre

Graphe

2. Relations en  
UML

3.  
Implémentation

Aggrégation

Composition

Composition interne

4. SDNL en C

Arbre

Graphe orienté

# Choix d'implémentations : comparaison rapide

LU2IN006

## 1. SD(N)L

Arbre

Graphe

## 2. Relations en UML

## 3. Implémentation

Aggrégation

Composition

Composition interne

## 4. SDNL en C

Arbre

Graphe orienté

- **Souplesse d'utilisation** : l'aggrégation simple permet à la relation et aux entités d'exister ou non. La composition interne est la plus stricte.
- **Générateur de bug** : plus la structure est souple, plus elle est génératrice de bug.
- **Aide au développement** : les relations externes permettent aux entités d'être développées séparément alors que la composition interne fusionne les 2 objets.

	Aggrégation (externe)	Composition externe	Composition interne
Souplesse	++	+	-
Anti-bug	-	+	++
Développement	++	++	-

# Avantage des relations externes : référence doublée

LU2IN006

## 1. SD(N)L

Arbre  
Graphe

## 2. Relations en UML

## 3. Implémentation

Aggrégation  
Composition  
Composition interne

## 4. SDNL en C

Arbre  
Graphe orienté

Il est important pour l'ordinateur de pouvoir avoir accès à son processeur. Réciproquement, il serait intéressant au microprocesseur de savoir quel est son ordinateur.



On s'éloigne de UML : cette représentation (et les suivantes) n'existe que pour ce cours. UML utiliserait des *arités* sur les relations.

Un processeur :

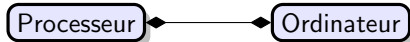
```
1  /* defini ailleurs */
2  struct s_ordi;
3
4  typedef struct s_proc {
5      int vitesse;
6
7      struct s_ordi* ordi;
8  } processeur;
```

Un ordinateur :

```
1  /* defini ailleurs */
2  struct s_proc;
3
4  typedef struct s_ordi {
5      float prix;
6
7      struct s_proc* proc;
8  } ordinateur;
```

# Référence doublée (2)

LU2IN006



Création (délégation de création) :

```
1 ordinateur* cree_ordinateur(float prix,int vitesse) {
2     ordinateur* o=(ordinateur *)malloc(sizeof(ordinateur));
3     processeur* p=(processeur *)malloc(sizeof(processeur));
4
5     o->prix=prix;
6     o->proc=p;
7
8     p->vitesse=vitesse;
9     p->ordi=o;
10
11     return o;
12 }
```

1. SD(N)L

Arbre

Graphe

2. Relations en  
UML

3. Implémentation

Aggrégation

Composition

Composition interne

4. SDNL en C

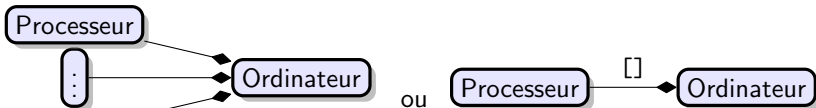
Arbre

Graphe orienté

# Avantage des relations externes : référence multiple

LU2IN006

Un ordinateur peut avoir plusieurs microprocesseurs.



Un processeur :

```
1 struct s_ordi;  
2  
3 typedef struct s_proc {  
4     int vitesse;  
5     struct s_ordi* ordi;  
6 } processeur;
```

Un ordinateur :

```
1 struct s_proc;  
2  
3 typedef struct s_ordi {  
4     float prix;  
5     struct s_proc** procs;  
6 } ordinateur;
```

1. SD(N)L

Arbre

Graphe

2. Relations en  
UML

3. Implémentation

Aggrégation

Composition

Composition interne

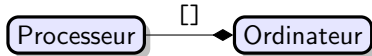
4. SDNL en C

Arbre

Graphe orienté

# Référence multiple (2)

LU2IN006



Création (délégation de créations) :

```
1 ordinateur* cree_ordinateur(float prix,int nbr,int
2     vitesse) {
3     ordinateur* o=(ordinateur *)malloc(sizeof(ordinateur));
4     o->prix=prix;
5     /* creation du tableau des processeurs */
6     o->procs=(processeur**)malloc(nbr*sizeof(processeur));
7
8     /* creation des nbr processeurs */
9     for(int i=0;i<nbr;i++) {
10        o->procs[i]=(processeur *)malloc(sizeof(processeur));
11        o->procs[i]->vitesse=vitesse;
12        o->procs[i]->ordi=o;
13    }
14
15    return o;
16 }
```

1. SD(N)L

Arbre

Graphe

2. Relations en  
UML

3. Implémentation

Aggrégation

Composition

Composition interne

4. SDNL en C

Arbre

Graphe orienté



# Retour sur les SDNL

LU2IN006

## 1. SD(N)L

Arbre  
Graphe

## 2. Relations en UML

## 3. Implémentation

Aggrégation  
Composition  
Composition interne

## 4. SDNL en C

Arbre  
Graphe orienté

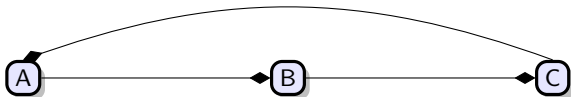
### Aggrégation récursive

On appelle **aggrégation (resp. composition) récursive** une relation d'aggrégation (resp. composition) entre une entité et elle-même.

- Les cellules d'une liste chaînée sont des valeurs d'une entité possédant une relation d'aggrégation récursive (appelée suivant).



- une aggrégation récursive n'est pas forcément directe :



### Structure de données non linéaire

Les structures de données non linéaires sont des entités possédant plusieurs relations d'aggrégation récursive (non redondantes).

# SDNL 1 : arbre binaire en C

LU2IN006

1. SD(N)L

Arbre

Graphe

2. Relations en

UML

3.

Implémentation

Aggrégation

Composition

Composition interne

4. SDNL en C

Arbre

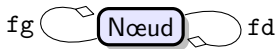
Graphe orienté

Un arbre binaire est défini par la présence en chaque nœud d'au plus deux fils. La structure de données représentant un tel arbre doit donc vérifier cette propriété.

## SDL arbre binaire

Un arbre binaire peut être représenté par une structure de données possédant deux relations d'aggrégation récursive.

On nomme **arbitrairement** ces 2 relations : "fils gauche" et "fils droit".



```
1 typedef struct s_noeudB {
2     int contenu; /* contenu de chaque noeud */
3
4     struct s_noeudB* fg;
5     struct s_noeudB* fd;
6 } noeud_binaire;
```



Les relations sont bien des aggrégations et non des compositions : il faut pouvoir ne pas mettre de fils gauche ou droit.

# SDNL 2 : arbre $n$ -aire V1 en C

LU2IN006

## 1. SD(N)L

Arbre

Graphe

## 2. Relations en

UML

## 3.

Implémentation

Aggrégation

Composition

Composition interne

## 4. SDNL en C

Arbre

Graphe orienté

Un arbre  $n$ -aire est défini par la présence en chaque nœud d'un nombre indéfini de fils. La structure de données représentant un tel arbre doit donc vérifier cette propriété.

### SDNL arbre $n$ -aire V1

Un arbre  $n$ -aire peut être représenté par une structure de données possédant une référence multiple représentant des relations d'aggrégation récursive.



```
1 typedef struct s_noeudN {  
2     int contenu; /* contenu de chaque noeud */  
3  
4     struct s_noeudN** fils;  
5 } noeud_naire;
```



Qu'obtient-on comme SDNL en remplaçant fils par voisins ?

# SDNL 3 : arbre $n$ -aire V2 en C

LU2IN006

## 1. SD(N)L

Arbre

Graphe

## 2. Relations en UML

UML

## 3.

Implémentation

Aggrégation

Composition

Composition interne

## 4. SDNL en C

Arbre

Graphe orienté

Une relation n'est pas représentée dans la V1 : la relation d'aggrégation permettant de retrouver le nœud père (s'il existe).

### SDNL arbre $n$ -aire V2

Un arbre  $n$ -aire peut être représenté par une structure de données qui possède également une relation d'aggrégation récursive nommée "père".



```
1 typedef struct s_noeudN {  
2     int contenu; /* contenu de chaque noeud */  
3  
4     struct s_noeudN* pere;  
5     struct s_noeudN** fils;  
6 } noeud_naire;
```



La SDNL ne dit rien sur la présence de cycle, etc. Ce sera à la logique des programmes de vérifier ou de s'en assurer.

# SDNL 4 : graphe orienté V1

LU2IN006

## 1. SD(N)L

Arbre

Graphe

## 2. Relations en UML

## 3.

Implémentation

Aggrégation

Composition

Composition interne

## 4. SDNL en C

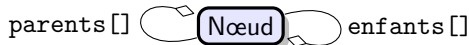
Arbre

Graphe orienté

Un graphe orienté est défini par la présence en chaque nœud d'un nombre indéfini de parents et d'un nombre indéfini d'enfants.

### SDNL graphe orienté V1

Un graphe orienté peut être représenté par une structure de données qui possède deux références multiples représentant des relations d'aggrégation récursive "parents" et "enfants".



```
1 typedef struct s_noeudG {  
2     int contenu; /* contenu de chaque noeud */  
3  
4     struct s_noeudG** parents;  
5     struct s_noeudG** enfants;  
6 } noeud_graphe_oriente;
```



Les SDNLs peuvent également utiliser des listes linéaires chaînées au lieu de tableau pour ses références multiples.

# SDNL 5 : graphe orienté V2

LU2IN006

## 1. SD(N)L

Arbre

Graphe

## 2. Relations en UML

## 3.

Implémentation

Aggrégation

Composition

Composition interne

## 4. SDNL en C

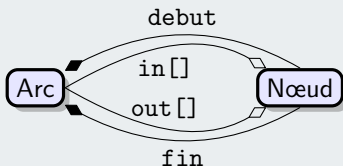
Arbre

Graphe orienté

Un graphe orienté est défini par la présence en chaque nœud d'un nombre indéfini d'arcs entrants et d'arcs sortants.

Un arc est défini par la présence d'un nœud début et d'un nœud fin.

### SDNL graphe orienté V2



```
1 struct s_arc;  
2  
3 typedef struct s_noeudG {  
4     int contenu;  
5  
6     struct s_arc** in;  
7     struct s_arc** out;  
8 } noeud_graphe_oriente;
```

```
1 struct s_noeudG;  
2  
3 typedef struct s_arc {  
4     struct s_noeudG* debut;  
5     struct s_noeudG* fin;  
6 } arc;
```