

Réversivité et TDA

Thierry Lecroq

Université de Rouen
FRANCE

Plan du cours

- 1 La récursivité
- 2 Les listes chaînées
- 3 Les types de données abstraits

Plan

- 1 La récursivité
- 2 Les listes chaînées
- 3 Les types de données abstraits

La récursivité

Définition informelle

Un objet est dit **récursif** s'il est utilisé dans sa définition ou sa composition.

Exemple

Une poupée russe est une poupée qui contient une poupée russe.

La récursivité

On peut utiliser la récursivité pour définir des objets mathématiques comme les entiers naturels.

Exemple

0 est un entier naturel.

Le successeur d'un entier naturel est un entier naturel.

La récursivité

Dans toute définition récursive doit apparaître une **condition d'arrêt**.

Exemple

Une poupée russe est une poupée qui contient une poupée russe **ou une poupée pleine**.

Fonction récursive

Une fonction est dite récursive si sa définition contient un appel à elle-même.

La condition d'arrêt permet alors d'arrêter les appels récursifs empêchant ainsi le programme de s'exécuter indéfiniment.

La plupart des langages de programmation autorise l'utilisation de sous-programmes récursifs.

Calcul de factorielle

$n!$

$$n! = 1 \times 2 \times \cdots \times n - 1 \times n = \prod_{i=1}^n i$$

$$0! = 1$$

Fonction itérative

fonction factIter(n : entier) : entier

auxiliaires i, f : entiers

début

$f \leftarrow 1$

$i \leftarrow 0$ $\{f = i!\}$

tant que $i < n$ faire $\{f = i! \text{ et } i < n\}$

$i \leftarrow i + 1$ $\{f = (i - 1)! \text{ et } i \leq n\}$

$f \leftarrow f \times i$ $\{f = i! \text{ et } i \leq n\}$

fin tant que $\{f = i! \text{ et } i = n\}$

retourner f

fin

Calcul de factorielle

Exemple

i	0	1	2	3	4	5
f	1	1	2	6	24	120

Complexité

La boucle tant que s'exécute n fois, la complexité est donc de l'ordre de n .

Calcul de factorielle

$$0! = 1$$

$$n! = n \times (n - 1)! \quad \forall n > 0$$

Fonction récursive

fonction factRec(n : entier) : entier

début

 si $n = 0$ alors

 retourner 1

 sinon

 retourner $n \times \text{factRec}(n - 1)$

 fin si

fin

Mémoire allouée à un processus

Zone de code (instructions)
Zone des données statiques
Pile d'exécution
Tas (données dynamiques)

Calcul de factorielle

Exemple

Zone de code (instructions)
Zone des données statiques
factRec($n = 5$)
Tas (données dynamiques)

Calcul de factorielle

Exemple

Zone de code (instructions)	
Zone des données statiques	
	factRec($n = 5$) $5 \times \text{factRec}(4)$
Tas (données dynamiques)	

Calcul de factorielle

Exemple

Zone de code (instructions)	
Zone des données statiques	
	factRec($n = 5$)
	5 × factRec(4)
	factRec($n = 4$)
	4 × factRec(3)
Tas (données dynamiques)	

Calcul de factorielle

Exemple

Zone de code (instructions)	
Zone des données statiques	
	factRec($n = 5$)
	$5 \times \text{factRec}(4)$
	factRec($n = 4$)
	$4 \times \text{factRec}(3)$
	factRec($n = 3$)
	$3 \times \text{factRec}(2)$
Tas (données dynamiques)	

Calcul de factorielle

Exemple

Zone de code (instructions)	
Zone des données statiques	
	factRec($n = 5$)
	$5 \times \text{factRec}(4)$
	factRec($n = 4$)
	$4 \times \text{factRec}(3)$
	factRec($n = 3$)
	$3 \times \text{factRec}(2)$
	factRec($n = 2$)
	$2 \times \text{factRec}(1)$
	factRec($n = 1$)
	$1 \times \text{factRec}(0)$
	factRec($n = 0$)
Tas (données dynamiques)	

Calcul de factorielle

Exemple

Zone de code (instructions)	
Zone des données statiques	
	factRec($n = 5$)
	$5 \times \text{factRec}(4)$
	factRec($n = 4$)
	$4 \times \text{factRec}(3)$
	factRec($n = 3$)
	$3 \times \text{factRec}(2)$
	factRec($n = 2$)
	$2 \times \text{factRec}(1)$
	factRec($n = 1$)
	$1 \times \text{factRec}(0)$
	factRec($n = 0$)
	$\leftarrow 1$
Tas (données dynamiques)	

Calcul de factorielle

Exemple

Zone de code (instructions)	
Zone des données statiques	
	factRec($n = 5$)
	$5 \times \text{factRec}(4)$
	factRec($n = 4$)
	$4 \times \text{factRec}(3)$
	factRec($n = 3$)
	$3 \times \text{factRec}(2)$
	factRec($n = 2$)
	$2 \times \text{factRec}(1)$
	factRec($n = 1$)
	1×1
Tas (données dynamiques)	

Calcul de factorielle

Exemple

Zone de code (instructions)	
Zone des données statiques	
	factRec($n = 5$)
	$5 \times \text{factRec}(4)$
	factRec($n = 4$)
	$4 \times \text{factRec}(3)$
	factRec($n = 3$)
	$3 \times \text{factRec}(2)$
	factRec($n = 2$)
	$2 \times \text{factRec}(1)$
	factRec($n = 1$)
	$\leftarrow 1$
Tas (données dynamiques)	

Calcul de factorielle

Preuve

- $n = 0$: $\text{factRec}(0) = 1 = 0!$
- HR : on suppose que $\text{factRec}(k)$ retourne $k!$ $\forall k < n$
- n : $\text{factRec}(n) = n \times \text{factRec}(n - 1) \stackrel{\text{HR}}{=} n \times (n - 1)! = n!$

Calcul de factorielle

Complexité

La complexité se calcule en nombre d'appels.

Pour $n = 5$ il y a 1 appel initial et 5 appels récursifs

Proposition

Le nombre d'appels récursifs de $\text{factRec}(n)$ est n .

Preuve

Par récurrence sur n

- $n = 0$: aucun appel récursif
- HR : $\forall k < n$ le nombre d'appels récursifs est k
- n : nbre d'appels récursifs = 1 appel à $\text{factRec}(n - 1)$ + nbre d'appels récursifs de $\text{factRec}(n - 1) \stackrel{\text{HR}}{=} 1 + n - 1 = n$

Au total le nombre d'appels est égal à 1 appel initial + n appels récursifs
= $n + 1$ appels

Les nombres de Fibonacci

$$F_0 = 0$$

$$F_1 = 1$$

$$F_n = F_{n-1} + F_{n-2} \quad \forall n > 1$$

Les nombres de Fibonacci

Fonction itérative

fonction fibolter(n : entier) : entier

auxiliaires e, f, i : entiers

début

si $n = 0$ alors

retourner 0

sinon

$e \leftarrow 0$

$f \leftarrow 1$

$i \leftarrow 1$ $\{f = F_i\}$

tant que $i < n$ faire $\{e = F_{i-1}, f = F_i \text{ et } i < n\}$

$f \leftarrow f + e$ $\{e = F_{i-1}, f = F_{i+1} \text{ et } i < n\}$

$e \leftarrow f - e$ $\{e = F_i, f = F_{i+1} \text{ et } i < n\}$

$i \leftarrow i + 1$ $\{e = F_{i-1}, f = F_i \text{ et } i \leq n\}$

fin tant que $\{e = F_{i-1}, f = F_i \text{ et } i = n\}$

retourner f

fin si

fin

Les nombres de Fibonacci

Fonction récursive

```
fonction fiboRec( $n$  : entier) : entier
```

```
début
```

```
  si  $n < 2$  alors
```

```
    retourner  $n$ 
```

```
  sinon
```

```
    retourner fiboRec( $n - 1$ ) + fiboRec( $n - 2$ )
```

```
  fin si
```

```
fin
```

La récursivité

Intérêts

Une fois le problème présenté de manière récursive, la construction de l'algorithme est immédiate.

Les preuves de programme faites par récurrence sont également très simples.

Inconvénients

La récursivité nécessite de l'espace pour mémoriser les opérations en cours.

Plan

- 1 La récursivité
- 2 Les listes chaînées
- 3 Les types de données abstraits

Les listes chaînées

Soit D un ensemble de valeurs (ou domaine).

Une liste chaînée à valeurs dans D est une structure à accès séquentiel qui permet de représenter toute séquence sur D .

Elle est formée :

- d'un pointeur, dit « tête de liste » permettant d'accéder au premier maillon de la liste ;
- d'une séquence de maillons dont chacun pointe vers le suivant, excepté le dernier.

Chaque maillon a deux composantes :

- une composante « valeur » sur D ;
- une composante « pointeur ».

Les listes chaînées

La séquence sur D représentée par une liste chaînée est la séquence des composantes « valeurs » considérées à partir de la tête de liste.

Exemple

$$\bar{\ell}_1 = \langle e_1, e_3, e_4 \rangle$$

$$\bar{\ell}_2 = \langle e_2, e_5 \rangle$$

$$\bar{\ell}_3 = \langle \rangle$$

La séquence vide est représentée par une tête de liste de valeur NIL (NULL en C, \times sur les dessins).

La composante « pointeur » du dernier maillon d'une liste non vide vaut NIL.

Les listes chaînées

Manipulation

type

Liste = enregistrement

 element : D

 suivant : pointeur vers Liste

fin enregistrement

Pliste = pointeur vers Liste

fonction listeVide() : Pliste

début

 retourner NIL

fin

fonction getSuivant(ℓ : Pliste) : Pliste

Précondition : ℓ est non vide

début

 retourner $\uparrow \ell$.suivant

fin

Les listes chaînées

Manipulation

procédure setSuivant($\ell, p : \text{Pliste}$)

Précondition : ℓ est non vide

début

$\uparrow \ell.\text{suivant} \leftarrow p$

fin

fonction getÉlément($\ell : \text{Pliste}$) : D

Précondition : ℓ est non vide

début

 retourner $\uparrow \ell.\text{element}$

fin

procédure setÉlément($\ell : \text{Pliste}, x : D$)

Précondition : ℓ est non vide

début

$\uparrow \ell.\text{element} \leftarrow x$

fin

Les listes chaînées

Manipulation

fonction $\text{estListeVide}(\ell : \text{Pliste}) : \text{booléen}$

début

 retourner $\ell = \text{NIL}$

fin

fonction $\text{ajoutEnTête}(\ell : \text{Pliste}, x : D) : \text{Pliste}$

auxiliaire $p : \text{Pliste}$

début

$p \leftarrow \text{allouer}$

$\text{setÉlément}(p, x)$

$\text{setSuivant}(p, \ell)$

 retourner p

fin

Les listes chaînées

Programmation

```
struct Liste {
    D element ;
    struct Liste * suivant ;
};
typedef struct Liste * PListe ;

PListe listeVide() {
    return NULL ;
}

PListe getSuivant(PListe  $\ell$ ) {
    // Précondition :  $\ell$  est non vide
    return  $\ell \rightarrow$ suivant ;
}
```

Les listes chaînées

Programmation

```
void setSuivant(PListe  $\ell$ , PListe  $p$ ) {  
  // Précondition :  $\ell$  est non vide  
   $\ell \rightarrow \text{suivant} = p$ ;  
}
```

```
D getElement(PListe  $\ell$ ) {  
  // Précondition :  $\ell$  est non vide  
  return  $\ell \rightarrow \text{element}$ ;  
}
```

```
void setElement(PListe  $\ell$ , D  $x$ ) {  
  //Précondition :  $\ell$  est non vide  
   $\ell \rightarrow \text{element} = x$ ;  
}
```

Les listes chaînées

Programmation

```
int estListeVide(PListe  $\ell$ ) {
    return  $\ell == \text{NULL}$ ;
}

PListe ajoutEnTete(PListe  $\ell$ ,  $D x$ ) {
    PListe  $p$ ;

     $p = (\text{PListe})\text{malloc}(\text{sizeof}(\text{struct Liste}));$ 
    if ( $p == \text{NULL}$ ) exit(1);
    setElement( $p, x$ );
    setSuivant( $p, \ell$ );
    return  $p$ ;
}
```

Les listes chaînées

Affichage

Pour afficher les valeurs d'une liste il faut la parcourir du premier maillon jusqu'au dernier.

Longueur

Idem affichage

Affichage inverse

Application de la récursivité

Les listes chaînées

```
fonction parcoursDirect( $\ell$  : PListe)
début
    tant que non estListeVide( $\ell$ ) faire
        écrire(getElement( $\ell$ ))
         $\ell \leftarrow$  getSuivant( $\ell$ )
    fin tant que
fin
```

```
fonction longueur( $\ell$  : PListe) : entier
auxiliaire  $n$  : entier
début
     $n \leftarrow 0$ 
    tant que non estListeVide( $\ell$ ) faire
         $n \leftarrow n + 1$ 
         $\ell \leftarrow$  getSuivant( $\ell$ )
    fin tant que
    retourner  $n$ 
fin
```

Les listes chaînées

```
fonction parcoursInverse( $\ell$  : PListe)
début
  si non estListeVide( $\ell$ ) faire
    parcoursInverse(getSuivant( $\ell$ ))
    écrire(getElement( $\ell$ ))
  fin si
fin
```

Les listes chaînées : implantation dynamique versus tableaux

Avantages de l'implantation dynamique

- bien adaptées pour la manipulation de séquences de longueurs différentes : gain de place mémoire ;
- insertion et suppression d'un maillon sans avoir à réarranger les autres valeurs (insertion par « déviation » et suppression par « court circuit ») : gain de temps.

Inconvénient de l'implantation dynamique

accès séquentiel (pas direct contrairement aux tableaux), pour accéder au i^{e} élément il faut accéder aux 1^{er} , 2^{e} , ... et $(i - 1)^{\text{e}}$ avant : perte de temps

Les listes chaînées

On peut également représenter des listes chaînées dans des tableaux :

0		1
1		6
2	e_3	7
3	e_5	-1
4	e_1	2
5	e_2	3
6		8
7	e_4	-1
8		9
9		10
10		-1

$$l_1 = 4, l_2 = 5, l_3 = -1$$

Liste des emplacements disponibles : *libre* = 0

Les listes chaînées

```
fonction allouer( $T$  : tableau de maillons)
auxiliaire  $i$  : entier
début
  si  $libre = -1$  alors
    erreur(" plus de place" )
  sinon
     $i \leftarrow libre$ 
     $libre \leftarrow T[libre].suivant$ 
  retourner  $i$ 
fin si
fin
```

Durée de vie d'une variable dynamique

de l'instant où elle a été créée jusqu'à la fin de l'exécution du programme sauf :

- désallocation implicite par un mécanisme de ramasse-miettes (garbage collector) : collecte et agrège entre elles les zones mémoire qui ne sont plus référencées (effectif en Java mais pas en C)
- désallocation explicite par $\text{désallouer}(p)$ ($\text{free}(p)$ en C) qui rend disponible l'espace occupé par $\uparrow p$ (ne change pas la valeur de la variable p)

Attention aux références fantômes !

Plan

- 1 La récursivité
- 2 Les listes chaînées
- 3 Les types de données abstraits

Types de Données Abstraits (TDA)

Un TDA est la description d'un ensemble organisé d'entités et des opérations de manipulation sur cet ensemble.

Ces opérations comprennent les moyens d'accéder et de modifier les éléments.

Un TDA comprend :

- une partie interface (description de l'ensemble et des opérations) ;
 - une partie implémentation (description de la structure de données et des algorithmes de manipulation).
-
- interface : description des opérations possibles ;
 - implémentation : explicite la représentation des éléments.

Types de Données Abstraits

Le concept de TDA ne dépend pas d'un langage de programmation.

Le programmeur convient d'accéder au TDA à travers l'ensemble limité de fonctions définies par l'interface et s'interdit notamment tout accès direct aux structures de données sous-jacentes qui serait rendu possible par une connaissance précise de l'implémentation particulière.

La réalisation efficace de TDA par des structures de données qui implémentent les opérations de manière optimale en temps et en espace constitue l'une des préoccupations de l'algorithmique.

Les piles

LIFO (Last In First Out)

Une pile est une liste linéaire où les insertions et les suppressions se font toutes du même côté (analogie : pile d'assiettes).

- insertion : empiler
- suppression : dépiler

Si la pile n'est pas vide, le seul élément accessible est le sommet de pile.

Propriété fondamentale

La suppression annule l'insertion (si on empile un élément puis on dépile alors on se retrouve dans l'état de départ).

Les piles

Interface

`pileVide()` : `Pile` retourne une pile vide

`getSommet(p : Pile)` : `D` retourne l'élément au sommet de la pile p , la pile p doit être non vide

`empiler(x : D, p : Pile)` insère l'élément x au sommet de la pile p

`dépiler(p : Pile)` supprime l'élément au sommet de la pile p , la pile p doit être non vide

`estPileVide(p : Pile)` : `booléen` teste si la pile p est vide

Les files d'attente

FIFO (First In First Out)

Une file est une liste linéaire où les insertions se font toutes d'un même côté et où les suppressions se font toutes de l'autre côté.

- insertion : enfiler
- suppression : défiler

Si la pile n'est pas vide, le seul élément accessible est la tête de file.

Les files

Interface

`fileVide()` : `File` retourne une file vide

`getTête(f : File)` : `D` retourne l'élément en tête de la file f , la file f doit être non vide

`enfiler(x : D, f : File)` insère l'élément x en queue de la file f

`défiler(f : File)` supprime l'élément en tête de la file f , la file f doit être non vide

`estFileVide(f : File)` : `booléen` teste si la file f est vide