

Structures de données dynamiques

Nicolas Delestre



SDD - v1.3

1 / 28

Introduction

Introduction 1 / 2

Attention

- Les concepts présentés sont des mécanismes de conception du paradigme de la programmation structurée
- Les concepts sont mis en oeuvre au niveau de la conception détaillée (en préservant toutefois le principe d'encapsulation)



SDD - v1.3

3 / 28

Plan...

- 1 Introduction
- 2 Allocation statique et allocation dynamique
- 3 Structures de données dynamiques linéaires
 - Liste chaînée
 - Liste doublement chaînée
- 4 Structure de donnée dynamique hiérarchique
 - Arbre binaire
- 5 Conclusion



SDD - v1.3

2 / 28

Allocation statique et allocation dynamique

Introduction 2 / 2

Pointeur

- On nomme un « pointeur » p une variable permettant de référencer une zone mémoire permettant de stocker une information de type T
- Le type de p est nommé « pointeur sur T ». Il est noté $\wedge T$
- Lorsqu'une variable ne pointe sur aucune zone mémoire, il faut l'initialiser avec la valeur NIL

Opérateurs sur les pointeurs

- \wedge opérateur unaire (opérande à gauche de l'opérateur) permettant de « déréférencer » un pointeur (accéder à la valeur de la zone mémoire pointée)
- $@$ opérateur unaire (opérande à droite de l'opérateur) permettant d'obtenir un pointeur sur une variable



SDD - v1.3

4 / 28

Mémoire et allocation 1 / 2

Allocation de mémoire

- Les entités utilisées (variables, constantes, (sous-)programmes) par le programme source sont représentées en mémoire (RAM de l'ordinateur)
- Il y a une relation directe entre l'identifiant que l'on utilise et un espace mémoire qui stocke l'information correspondante

Plusieurs emplacements dans la mémoire vive (segment)

statique ou *text* emplacement pour les programmes, sous-programmes

bss emplacement pour les variables globales

data emplacement pour les constantes

tas ou *heap* emplacement où sont stockés des espaces mémoires alloués dynamiquement. La taille du tas varie en fonction de l'exécution du programme

Allocation statique

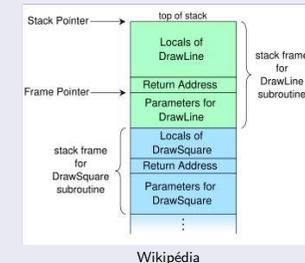
Définition

- Allocation de mémoire effectuée lors de l'exécution mais dont la taille est prévue lors de la compilation
 - À chaque type de données correspond une taille mémoire et une façon de représenter l'information
 - À chaque variable ou paramètre formel déclaré correspond un espace mémoire dont la taille est fonction du type
- Le compilateur ajoute donc automatiquement du code pour réserver de l'espace mémoire avant utilisation (au niveau de la déclaration) et pour libérer si besoin est (dans la pile)

Mémoire et allocation 2 / 2

Plusieurs emplacements dans la mémoire vive (segment)

pile ou *stack* emplacement où sont stockées les variables locales et les paramètres formels des sous-programmes. La taille de la pile varie en fonction de l'exécution du programme



Wikipédia

Allocation dynamique

Définition

- Allocation de mémoire effectuée lors de l'exécution mais dont la taille n'est pas prévue lors de la compilation
 - Allocation qui se fait uniquement dans le tas
 - Cette allocation est à la charge du programmeur, il lui faut donc :
 - une procédure permettant de réserver une zone mémoire (allouer)
 - une procédure permettant de libérer une zone mémoire (libérer)
 - une variable (et donc un type) permettant de référencer cette zone mémoire allouée

Le problème 1 / 3

Contexte

- Lorsque l'on veut stocker en mémoire n éléments de même type, on utilise jusqu'à présent les tableaux
- Les tableaux sont généralement des allocations statiques (la taille du tableau est définie à la compilation), on ne peut pas l'adapter au contexte. Le fait de réserver MAX éléments :
 - consomme beaucoup de mémoire si peu d'éléments réellement utilisés ($n \ll MAX$)
 - pose problème si on a besoin de plus de MAX éléments à stocker ($n > MAX$)
- Il faudrait pouvoir stocker en mémoire autant de données dont on a besoin et pas plus
- Mais ce nombre de données ne peut être déterminé à la compilation, il ne peut être déterminé qu'à l'exécution

C'est le rôle des structures de données dynamiques (SDD)



Le problème 2 / 3

Comment les concevoir ?

- Il faut donc que la mémoire soit réservée à l'exécution \Rightarrow besoin d'allocations dynamiques
- Mais il faut pouvoir référencer ces allocations dynamiques \Rightarrow besoins de pointeurs (les pointeurs sont des variables donc allocation statique)
- Ainsi le nombre de pointeurs est fonction du nombre d'éléments que l'on veut stocker, ce qui est contradictoire avec notre objectif
- Il faut donc que les futurs espaces mémoires alloués ne soient pas référencés par des variables mais par les espaces mémoires déjà alloués
- Ainsi **les espaces mémoires déjà alloués** stockent l'information à réellement stocker *et* également une référence vers **les autres espaces mémoires alloués ou à allouer** (définition récursive)

Le problème 3 / 3

Comment les concevoir? (suite)

- Une SDD est donc au départ une variable pointeur (allocation statique) qui fait référence à une zone mémoire allouée dynamiquement qui stocke à la fois un élément à réellement stocker et une ou plusieurs références vers le même type de zone mémoire

Remarque

Les SDD sont donc récursives



Liste chaînée 1 / 9

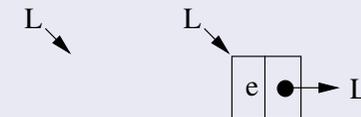
Définition

Une liste chaînée est soit :

- une liste vide
- un élément suivi d'une liste chaînée

Graphiquement

On représente une liste chaînée L de la façon suivante :



On voit donc apparaître le concept de *pointeur* et de *noeud*



Liste chaînée 2 / 9

Conception

On peut donc concevoir le type ListeChaînee de la façon suivante :

Type ListeChaînee = $\hat{}$ Noeud

Type Noeud = **Structure**

 element : Element

 listeSuiVante : ListeChaînee

finstructure

Signatures des fonctions/procédures

- **fonction** listeVide () : ListeChaînee
- **fonction** estVide (uneListe : ListeChaînee) : **Booleen**
- **procédure** ajouter (E/S uneListe : ListeChaînee, E element : Element)
- **fonction** obtenirElement (uneListe : ListeChaînee) : Element
 |précondition(s) non(estVide(uneListe))
- **fonction** obtenirListeSuiVante (uneListe : ListeChaînee) : ListeChaînee
 |précondition(s) non(estVide(uneListe))

Liste chaînée 4 / 9

Conception détaillée

fonction listeVide () : ListeChaînee

debut

 retourner NIL

fin

fonction estVide (l : ListeChaînee) : **Booleen**

debut

 retourner l=NIL

fin

procédure ajouter (E/S l : ListeChaînee, E e : Element)

Déclaration temp : ListeChaînee

debut

 temp \leftarrow l

allouer(l)

 l^.element \leftarrow e

 fixerListeSuiVante(l,temp)

fin

Liste chaînée 3 / 9

Signatures des fonctions/procédures (suite)

- **procédure** fixerListeSuiVante (E/S uneListe : ListeChaînee, E nelleSuite : ListeChaînee)
 |précondition(s) non(estVide(uneListe))
- **procédure** supprimerTete (E/S uneListe : ListeChaînee)
 |précondition(s) non(estVide(uneListe))
- **procédure** supprimer (E/S uneListe : ListeChaînee)

Liste chaînée 5 / 9

Conception détaillée (suite)

fonction obtenirElement (l : ListeChaînee) : Element

 |précondition(s) non estVide(l)

debut

 retourner l^.element

fin

procédure fixerListeSuiVante (E/S l : ListeChaînee, E l' : ListeChaînee)

 |précondition(s) non estVide(l)

debut

 l^.listeSuiVante \leftarrow l'

fin

fonction obtenirListeSuiVante (l : ListeChaînee) : ListeChaînee

 |précondition(s) non estVide(l)

debut

 retourner l^.listeSuiVante

fin

Liste chaînée 6 / 9

Conception détaillée (fin)

```

procédure supprimerTete (E/S l : ListeChaînee)
  |précondition(s) non estVide(l)
  Déclaration temp : ListeChaînee
debut
  temp ← l
  l ← obtenirListeSuivante(l)
  liberer(temp)
fin
procédure supprimer (E/S l : ListeChaînee)
debut
  tant que non estVide(l) faire
    supprimerTete(l)
  fintantque
fin

```



Utilisation des listes chaînées

Des algorithmes naturellement récursifs

- De part la définition récursive des listes chaînées, les algorithmes les utilisant sont naturellement récursifs
- Certains algorithmes sont uniquement récursifs

Exemples d'algorithmes

- obtenir le nombre d'éléments, le ième élément
- savoir si un élément est présent
- concaténer deux listes chaînées
- insérer un élément à la ième position
- insérer un élément dans une liste chaînée ordonnée
- fusionner deux listes chaînées ordonnées
- inverser une liste chaînée
- ...

Liste chaînée 7 / 9

Exemple d'utilisation : parcours de liste (version itérative)

```

procédure afficher (E l : ListeChaînee)
debut
  tant que non estVide(l) faire
    afficherElement(obtenirElement(l))
    l ← obtenirListeSuivante(l)
  fintantque
fin

```

Exemple d'utilisation : parcours de liste (version récursive)

```

procédure afficher (E l : ListeChaînee)
debut
  si non estVide(l) alors
    afficherElement(obtenirElement(l))
    afficher(obtenirListeSuivante(l))
  finsi
fin

```



Liste chaînée 8 / 9

Attention

Le fait d'utiliser le principe d'encapsulation peut obliger à utiliser des variables intermédiaires dans des algorithmes récursifs

Exemple suppression d'un élément

```

procédure supprimerElement (E/S l : ListeChaînee, e : Element)
  Déclaration temp : ListeChaînee
debut
  si non est Vide(l) alors
    si obtenirElement(l)=e alors
      supprimerTete(l)
    sinon
      temp ← obtenirListeSuivante(l)
      supprimerElement(temp,e)
      fixerListeSuivante(l,temp)
  finsi
fin

```



Liste chaînée 9 / 9

Le vision itérative du précédent algorithme va obliger à utiliser deux « pointeurs »

Exemple suppression d'un élément

```

procédure supprimerElement (E/S l : ListeChaînee, E e : Element)
  Déclaration  gauche, droite : ListeChaînee; trouve : Booleen
  debut
    si non est Vide(l) alors
      si obtenirElement(l)=e alors
        supprimerTete(l)
      sinon
        gauche ← l
        droite ← obtenirListeSuiVante(l)
        trouve ← FAUX
        tant que non estVide(droite) et non trouve faire
          si obtenirElement(droite)=e alors
            trouve ← VRAI
          sinon
            gauche ← droite
            droite ← obtenirListeSuiVante(droite)
        finsi
        fintantque
        si trouve alors
          fixerListeSuiVante(gauche, obtenirListeSuiVante(droite))
          supprimerTete(droite)
        finsi
      finsi
    fin
  fin

```

SDD - v1.3

21 / 28

Liste doublement chaînée 1 / 3

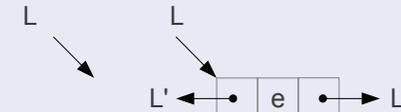
Définition

Une liste doublement chaînée est soit :

- une liste vide
- un élément suivi d'une liste doublement chaînée et précédé d'une liste doublement chaînée

Graphiquement

On représente une liste doublement chaînée L de la façon suivante :



SDD - v1.3

22 / 28

Liste doublement chaînée 2 / 3

Signature des fonctions/procédures (LDC = ListeDoublementChaînee)

- fonction listeVide () : LDC
- fonction estVide (l : LDC) : Booleen
- procédure inserer (E/S l : LDC, E element : Entier)
- fonction obtenirElement (l : LDC) : Entier
|précondition(s) non(estVide(l))
- fonction obtenirListeSuiVante (l : LDC) : LDC
|précondition(s) non(estVide(l))
- fonction obtenirListePrecedente (l : LDC) : LDC
|précondition(s) non(estVide(l))
- procédure fixerListeSuiVante (E l : LDC, l' : LDC)
|précondition(s) non(estVide(l))
- procédure fixerListePrecedente (E/S l : LDC, l' : LDC)
|précondition(s) non(estVide(l))
- procédure supprimerNoeud (E/S l : LDC, S avant, apres : LDC)
|précondition(s) non estVide(l)
- procédure supprimer (E/S l : LDC)

SDD - v1.3

23 / 28

Liste doublement chaînée 3 / 3

Exercice

Faire la conception détaillée du type ListeDoublementChaînee

SDD - v1.3

24 / 28

Les SDD hiérarchiques s'inspirent des TAD hiérarchiques, donc des arbres :

- le nombre de fils est de taille fixe (très souvent 2) : utilisation de champs ou d'un tableau
- le nombre de fils varie : utilisation d'une liste chaînée



Conception

```

Type ArbreBinaire = ^ Noeud
Type Noeud = Structure
  |Element : Element
  |filsGauche : ArbreBinaire
  |filsDroit : ArbreBinaire
finstructure

```

Signatures des fonctions/procédures

```

fonction arbreBinaire () : ArbreBinaire
fonction estVide (a : ArbreBinaire) : Booleen
fonction ajouterRacine (fg,fd : ArbreBinaire,e : Element) : ArbreBinaire
fonction obtenirElement (a : ArbreBinaire) : Element
  |précondition(s) non estVide(a)
fonction obtenirFilsGauche (a : ArbreBinaire) : ArbreBinaire
  |précondition(s) non estVide(a)
fonction obtenirFilsDroit (a : ArbreBinaire) : ArbreBinaire
  |précondition(s) non estVide(a)

```

Signatures des fonctions/procédures (suite)

```

procédure fixerFilsGauche (E a : ArbreBinaire, ag : ArbreBinaire)
  |précondition(s) non estVide(a)
procédure fixerFilsDroit (E a : ArbreBinaire, ad : ArbreBinaire)
  |précondition(s) non estVide(a)
procédure supprimerRacine (E/S a : ArbreBinaire, S fg,fd : ArbreBinaire)
  |précondition(s) non estVide(a)
procédure supprimer (E/S a : ArbreBinaire)

```

Remarque

Les algorithmes utilisant les arbres binaires sont obligatoirement récursifs

Exercice

Donnez les algorithmes des ces fonctions et procédures



- Les structures dynamiques sont très utilisées
- Certains langages les proposent de base, d'autres pas, comme le C. Nous verrons comment les implanter dans un prochain cours
- Il faut être rigoureux pour ne pas perdre des éléments

Attention

- L'utilisation de l'affectation peut être piègeuse
- Bien faire la différence entre identique et égal

