Collections

Nicolas Delestre - Michel Mainguenaud

INSA ROUEN NORMANDIE

Plan

INSA ROUEN NORMANDIE

Introduction

- Quelque soit le domaine, on a souvent besoin de stocker des éléments
- Les TAD collections proposent différentes façons de stocker des éléments
- Ces TAD sont souvent implantés dans les langages de dernières générations
- À la fin de ce cours, ces TAD seront des types de base de l'algorithmique (comme les booléens, les naturels, etc.)

Rappel

- Le champ *paramètre* d'un TAD permet de désigner les types des éléments à stocker sans les connaitre au moment de la spécification de ce TAD
- L'identifiant utilisé dans ce champ *paramètre* doit être générique, sans signification particulière (nous utiliserons ici l'identifiant *Element*)
- S'il il y a des contraintes sur le type qui sera utilisé, celles ci sont exprimés formellement

INSA ROUEN NORMANDIE

TAD Pile 1 / 4

Définition

Collection avec une gestion des éléments du type LIFO (*Last In First Out*)
Deux actions possibles :

- empiler un élément
- dépiler un élément

INSA ROUEN NORMANDIE

TAD Pile 2 / 4

Nom:	Pile	
Paramètre:	Element	
Utilise:	**Booleen**	
Opérations:	pile:	$\rightarrow$ Pile
	estVide:	Pile $\rightarrow$ **Booleen**
	empiler:	Pile $\times$ Element $\rightarrow$ Pile
	dépiler:	Pile $\nrightarrow$ Pile
	obtenirElement:	Pile $\nrightarrow$ Element
Axiomes:	- estVide(pile())	
	- non estVide(empiler(p,e))	
	- depiler(empiler(p,e))=p	
	- obtenirElement(empiler(p,e))=e	
Préconditions:	depiler(p):	$non(estVide(p))$
	obtenirElement(p):	$non(estVide(p))$

TAD Pile 3 / 4

Exemple d'utilisation des piles

- Évaluation des expréssions arithmétiques (notation préfixe, ou polonaise inversée)
- Gestion des appels de fonctions

Vérification du fonctionnement du TAD Pile

Les axiomes permettent de vérifier le bon fonctionnement du TAD Pile

Exercice

Que vaut obtenirElement(dépiler(empiler(empiler(pile(),e1),e2))) ?

TAD Pile 4 / 4

Que vaut obtenirElement(dépiler(empiler(empiler(pile(),e1),e2))) ?

Démonstration :

- D'après Le 3ème axiome :
  dépiler(empiler(empiler(pile(),e1),e2)) = empiler(pile(),e1)
- Donc :
  obtenirElement(dépiler(empiler(empiler(pile(),e1),e2))) =
  obtenirElement(empiler(pile(),e1))
- D'après Le 4ème axiome :
  obtenirElement(empiler(pile(),e1)) = e1
- Donc : obtenirElement(dépiler(empiler(empiler(pile(),e1),e2))) = e1

TAD File 1 / 3

Définition

Collection avec une gestion des éléments du type FIFO (*First In First Out*)

- enfiler un élément
- défiler un élément

TAD File 2 / 3

Nom: File
Paramètre: Element
Utilise: **Booleen**
Opérations:
file: → File
estVide: File → **Booleen**
enfiler: File × Element → File
défiler: File ↛ File
obtenirElement: File ↛ Element

Axiomes:
- estVide(file())
- non estVide(enfiler(f,e))
- defiler(enfiler(file(),e))=file()
- non estVide(f) et defiler(enfiler(f,e))=enfiler(defiler(f),e) [a]
- obtenirElement(enfiler(file(),e))=e
- non estVide(f) et obtenirElement(enfiler(f,e))=obtenirElement(f)

Préconditions:
defiler(f): $non(estVide(f))$
obtenirElement(f): $non(estVide(f))$

a. Cf. le cours de B. Duval `http://www.info.univ-angers.fr/pub/bd/`

TAD File 3 / 3

Exemple d'utilisation des files
- Programmation système
  - Gestion d'une imprimante partagée
  - Allocation du processeur aux programmes en exécution
- Parcours en largeur d'arbres, de treillis, de graphes

Exercice
Que vaut obtenirElement(défiler(enfiler(enfiler(file(),e1),e2))) ?

INSA ROUEN NORMANDIE

TAD Liste 1 / 2

Définition
Collection avec une gestion des éléments avec accès par position
- insérer un élément à une position
- supprimer un élément à une position

INSA ROUEN NORMANDIE

TAD Liste 2 / 2

Nom: Liste
Paramètre: Element
Utilise: **Booleen**, **NaturelNonNul**, **Naturel**
Opérations:
liste: → Liste
estVide: Liste → **Booleen**
insérer: Liste × **NaturelNonNul** × Element ↛ Liste
supprimer: Liste × **NaturelNonNul** ↛ Liste
obtenirElement: Liste × **NaturelNonNul** ↛ Element
longueur: Liste → **Naturel**

Axiomes:
- estVide(liste())
- non estVide(insérer(l,i,e))
- suprimer(insérer(l,i,e),i)=l
- obtenirElement(insérer(insérer(l,i,e1),i,e2),$i+1$)=e1
- longueur(liste())=0
- longueur(insérer(l,i,e))=1+longueur(l)

Préconditions:
inserer(l,i,e): $i \leq longueur(l)+1$
supprimer(l,i): $i \leq longueur(l)$
obtenirElement(l,i): $i \leq longueur(l)$

TAD ListeOrdonnee 1 / 2

Définition

Collection (d'un type de données possédant un ordre total) avec une gestion des éléments avec accès par position

- insérer un élément
- supprimer un élément à une position

TAD ListeOrdonnee 2 / 2

Nom: ListeOrdonnee
Paramètre: Element ($\forall e_1 \in Element, e_2 \in Element, e_1 \neq e2 \Rightarrow e_1 < e_2$ *ou* $e_1 > e_2$)
Utilise: **Booleen**, **NaturelNonNul**, **Naturel**
Opérations:

listeOrdonnée:	$\rightarrow$ ListeOrdonnee
estVide:	ListeOrdonnee $\rightarrow$ **Booleen**
insérer:	ListeOrdonnee $\times$ Element $\rightarrow$ ListeOrdonnee
supprimer:	ListeOrdonnee $\times$ **NaturelNonNul** $\nrightarrow$ ListeOrdonnee
obtenirElement:	ListeOrdonnee $\times$ **NaturelNonNul** $\nrightarrow$ Element
longueur:	ListeOrdonnee $\rightarrow$ **Naturel**

Axiomes:
- estVide(listeOrdonnee())
- non estVide(insérer(l,e))
- supprimer(insérer(l,e),i)=l et obtenirElement(insérer(l,e),i)=e
- obtenirElement(insérer(insérer(l,e'),e),i)=e et obtenirElement(insérer(insérer(l,e'),e),j)=e' et (($e' < e$ et $j < i$) ou ($e' > e$ et $j > i$))
- longueur(liste())=0
- longueur(insérer(l,e))=1+longueur(l)

Préconditions:
supprimer(l,i): $i \leq longueur(l)$
obtenirElement(l,i): $i \leq longueur(l)$

TAD Ensemble 1 / 2

Définition

Collection d'éléments n'apparaissant qu'une seule fois

- ajouter un élément (pas de notion d'ordre, et si déjà présent alors pas d'ajout)
- retirer un élément
- rechercher un élément
- *parcourir les éléments présents* (nouvelle instruction)

TAD Ensemble 2 / 2

Nom: Ensemble
Paramètre: Element
Utilise: **Booleen**,**Naturel**
Opérations:

ensemble:	$\rightarrow$ Ensemble
ajouter:	Ensemble $\times$ Element $\rightarrow$ Ensemble
retirer:	Ensemble $\times$ Element $\rightarrow$ Ensemble
estPresent:	Ensemble $\times$ Element $\rightarrow$ **Booleen**
cardinalite:	Ensemble $\rightarrow$ **Naturel**
union:	Ensemble $\times$ Ensemble $\rightarrow$ Ensemble
intersection:	Ensemble $\times$ Ensemble $\rightarrow$ Ensemble
soustraction:	Ensemble $\times$ Ensemble $\rightarrow$ Ensemble

Axiomes:
- ajouter(ajouter(s,e),e)=ajouter(s,e)
- retirer(ajouter(s,e),e)=s
- estPresent(ajouter(s,e),e)
- non estPresent(retirer(s,e),e)
- cardinalite(ensemble())=0
- cardinalite(ajouter(s,e))=1+cardinalite(s) et non estPresent(s,e)
- cardinalite(ajouter(s,e))=cardinalite(s) et estPresent(s,e)
- ...

Une nouvelle instruction

Pour chaque

Afin de permettre le parcours d'un ensemble ou de faciliter le parcours d'une liste (ordonnée ou pas), nous pouvons maintenant utiliser l'instruction pour chaque :

pour chaque element **de** liste
    actions utilisant element
finpour

Exemple

fonction compter (l : Liste<**Entier**>, unEntier : **Entier**) : **Naturel**
    Déclaration somme : **Naturel**
                e : **Entier**
debut
    somme ← 0
    pour chaque e **de** l
        si unEntier = e **alors**
            somme ← somme + 1
        finsi
    finpour
    retourner somme
fin

TAD Dictionnaire 1 / 2

Définition

Collection où les éléments sont constitués de deux parties : une clé et une valeur. Chaque valeur est identifiée par la clé.

- ajouter un couple (clé,valeur)
- retirer un élément (clé)
- tester la présence (clé)
- obtenir les clés
- obtenir les valeurs
- obtenir une valeur (clé)

INSA ROUEN NORMANDIE

TAD Dictionnaire 2 / 2

Nom: Dictionnaire
Paramètre: Cle,Valeur
Utilise: **Booleen**,Liste
Opérations:

dictionnaire:	$\rightarrow$ Dictionnaire
ajouter:	Dictionnaire $\times$ Cle $\times$ Valeur $\rightarrow$ Dictionnaire
retirer:	Dictionnaire $\times$ Cle $\rightarrow$ Dictionnaire
estPresent:	Dictionnaire $\times$ Cle $\rightarrow$ **Booleen**
obtenirValeur:	Dictionnaire $\times$ Cle $\nrightarrow$ Valeur
obtenirCles:	Dictionnaire $\rightarrow$ Liste<Cle>
obtenirValeurs:	Dictionnaire $\rightarrow$ Liste<Valeur>

Axiomes:
- ajouter(ajouter(d,c,v_2),c,v_1)=ajouter(d,c,v_1)
- retirer(ajouter(d,c,v),c)=d
- rechercher(ajouter(d,c,v),c)=v
- estPresent(ajouter(d,c,v),c)
- estPresent(d,obtenirElement(obtenirCles(d),i)) et $0 < i \leq$ longueur(obtenirCles(d))
- non estPresent(retirer(d,c),c)
- ...

Préconditions: obtenirValeur(d,c): *estPresent*(d, c)

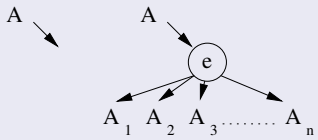
INSA ROUEN NORMANDIE

TAD hiérarchiques ou arbre

Un arbre sur un ensemble E est une imbrication de suites finies d'éléments de E, où chaque élément de cette suite permet d'accéder à une nouvelle suite finie

- un arbre vide est notée ()
- un arbre non vide est noté $(e, A_1, A_2, \ldots, A_n)$ où $e \in E$ et $A_i, i \in [1..n]$ sont des arbres

On le représente souvent graphiquement :



INSA ROUEN NORMANDIE

Vocabulaire 1 / 2

Arité d'un noeud

Nombre de sous-arbres non vide
Une feuille est un noeud d'arité 0

Chemin

Une séquence de noeuds $(n_0, n_1, \ldots, n_p)$ ou n_{i-1} est le père de n_i $(0 < i \leq p)$
La longueur d'un chemin $(n_0, \ldots, n_p)$ vaut p

- p = nombre d'arcs
- $p+1$ = nombre de noeuds

Niveau d'un noeud

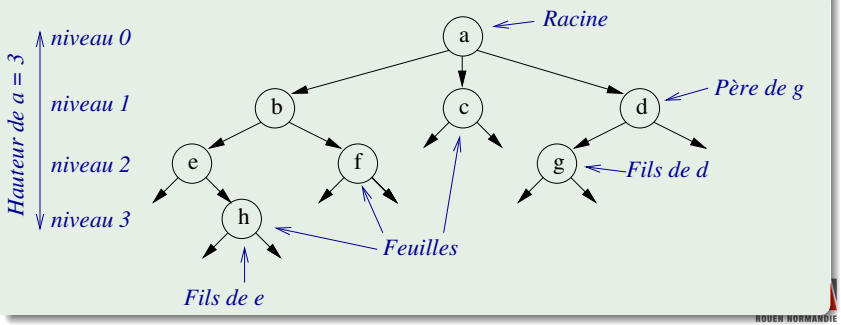
Le niveau (la hauteur) d'un noeud est la longueur de l'unique chemin de la racine à ce noeud
Le niveau de la racine vaut 0

Vocabulaire 2 / 2

Hauteur d'un arbre

Le maximum des hauteurs de tous les noeuds
La hauteur d'un arbre vide vaut -1 par définition

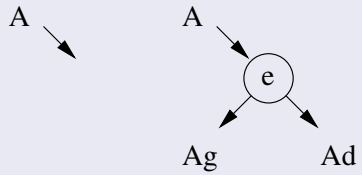
Un exemple



TAD Arbre Binaire 1 / 3

Définition

- Un arbre binaire est un arbre
- Si cet arbre n'est pas vide alors il a exactement deux fils (nommé fils gauche et fils droit)



TAD Arbre Binaire 2 / 3

Nom: ArbreBinaire
Paramètre: Element
Utilise: **Booleen**
Opérations:

arbreBinaire:		$\rightarrow$ ArbreBinaire
ajouterRacine:		Element $\times$ ArbreBinaire $\times$ ArbreBinaire $\rightarrow$ ArbreBinaire
estVide:		ArbreBinaire $\rightarrow$ **Booleen**
obtenirElement:		ArbreBinaire $\nrightarrow$ Element
obtenirFilsGauche:		ArbreBinaire $\nrightarrow$ ArbreBinaire
obtenirFilsDroit:		ArbreBinaire $\nrightarrow$ ArbreBinaire

Axiomes:
- estVide(arbreBinaire())
- non estVide(ajouterRacine(e,a_g,a_d))
- obtenirElement(ajouterRacine(e,a_g,a_d))=e
- obtenirFilsGauche(ajouterRacine(e,a_g,a_d))=a_g
- obtenirFilsDroit(ajouterRacine(e,a_g,a_d))=a_d

Préconditions:

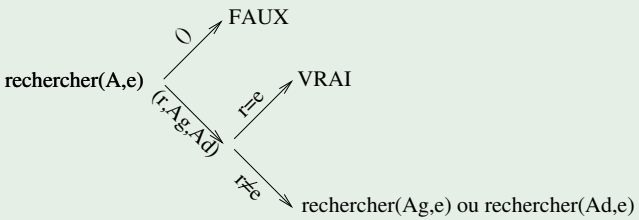
obtenirElement(a):	$non(estVide(a))$
obtenirFilsGauche(a):	$non(estVide(a))$
obtenirFilsDroit(a):	$non(estVide(a))$

TAD Arbre Binaire 3 / 3

Algorithmes sur les arbres

De part leur définition récursive, les algorithmes appliqués aux arbres sont naturellement récursifs

Méthode : recherche d'un élément

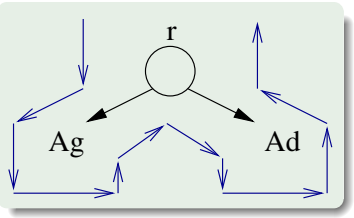


Parcours en profondeur 1 / 2

Il y a trois types de parcours

parcoursRGD

1. Traiter la racine r
2. parcoursRGD A_g
3. parcoursRGD A_d



parcoursGRD

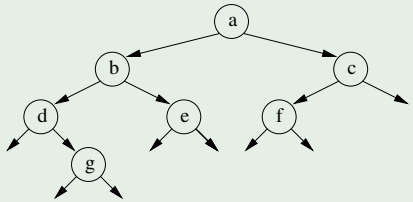
1. parcoursGRD A_g
2. Traiter la racine r
3. parcoursGRD A_d

parcoursGDR

1. parcoursGDR A_g
2. parcoursGDR A_d
3. Traiter la racine r

Parcours en profondeur 2 / 2

Un exemple



parcoursRGD a,b,d,g,e,c,f

parcoursGRD d,g,b,e,a,f,c

parcoursGDR g,d,e,b,f,c,a

Théorème

Il faut au moins deux parcours différents pour reconstruire un arbre

TAD Tas 1 / 2

Définition

Un tas est un arbre binaire tel que :

- la valeur se trouvant à la racine est plus petite (ou plus grande) que celles contenues dans les sous arbres
- l'arbre est équilibré
- la hauteur du sous arbre gauche est supérieure ou égale à la hauteur du sous arbre droit
- le sous arbre gauche et sous arbre droit sont des tas

TAD Tas 2 / 2

Nom: Tas
Paramètre: Element ($\forall e_1 \in Element, e_2 \in Element, e_1 \neq e2 \Rightarrow e_1 < e_2$ *ou* $e_1 > e_2$)
Utilise: **Booleen**
Opérations:

tas:		$\rightarrow$ Tas
estVide:		Tas $\rightarrow$ **Booleen**
estPresent:		Tas $\times$ Element $\rightarrow$ **Booleen**
inserer:		Tas $\times$ Element $\nrightarrow$ Tas
supprimer:		Tas $\times$ Element $\rightarrow$ Tas
obtenirElement:		Tas $\nrightarrow$ Element
obtenirFilsGauche:		Tas $\nrightarrow$ Tas
obtenirFilsDroit:		Tas $\nrightarrow$ Tas

Axiomes:
- estVide(tas())
- non estVide(inserer(t,e))
- nbElements(inserer(t,e)) = 1 + nbElements(t,e)
- obtenirElement(inserer(t,e)) $\leq e$
- obtenirElement(inserer(t,e))=e ou estPresent(obtenirFilsGauche(inserer(t,e))) ou estPresent(obtenirFilsDroit(inserer(t,e)))
- non estPresent(t,e) $\Rightarrow$ supprimer(t,e) = t
- estPresent(t,e) $\Rightarrow$ nbElements(supprimer(t,e)) = nbElements(t) - 1
- hauteur(obtenirFilsGauche(t)) - hauteur(obtenirFilsDroit(t)) $\in \{0, 1\}$
- ...
- estUnTas(obtenirFilsGauche(t))
- estUnTas(obtenirFilsDroit(t))

Préconditions:

obtenirElement(a):		*non*(*estVide*(a))
...:		...

TAD arbres binaires de recherche 1 / 3

Définition

Un arbre binaire de recherche est un arbre binaire tel que :
- le sous-arbre gauche et le sous-arbre droit sont des arbres binaires de recherche
- tous les éléments du sous-arbre gauche sont (strictement) plus petits que l'élément de la racine
- tous les éléments du sous-arbre droit sont strictement plus grands que l'élément de la racine

- Dans la spécification suivante nous prenons le cas où un élément ne peut pas être présent plus d'une fois dans un aBR (tous les éléments du sous-arbre gauche sont strictement plus petits que l'élément de la racine)

TAD arbres binaires de recherche 2 / 3

Nom: ABR (ArbreBinaireDeRecherche)
Paramètre: Element ($\forall e_1 \in Element, e_2 \in Element, e_1 \neq e2 \Rightarrow e_1 < e_2$ *ou* $e_1 > e_2$)
Utilise: **Booleen**
Opérations:

aBR:		$\rightarrow$ ABR
estVide:		ABR $\rightarrow$ **Booleen**
inserer:		ABR $\times$ Element $\nrightarrow$ ABR
supprimer:		ABR $\times$ Element $\rightarrow$ ABR
estPresent:		ABR $\times$ Element $\rightarrow$ **Booleen**
obtenirElement:		ABR $\nrightarrow$ Element
obtenirFilsGauche:		ABR $\nrightarrow$ ABR
obtenirFilsDroit:		ABR $\nrightarrow$ ABR

Axiomes:
- estVide(aBR())
- non estVide(inserer(a,e))
- obtenirElement(inserer(aBR(), e))=e
- estPresent(inserer(a, e), e)
- non estPresent(supprimer(a, e))
- estPresent(obtenirFilsGauche(a,e)) $\Rightarrow e <$ obtenirElement(a)
- estPresent(obtenirFilsDroit(a,e)) $\Rightarrow e >$ obtenirElement(a)

Préconditions:

obtenirElement(a):		*non*(*estVide*(a))
...:		...

TAD arbres binaires de recherche 3 / 3

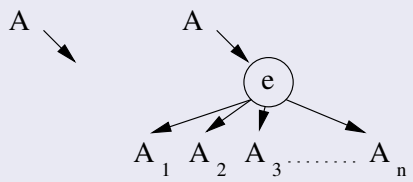
Remarque

- Suivant les algorithmes qui seront utilisés pour implanter les opérations d'insertion et de suppression (qui auront un impact sur la complexité de l'opération estPrésent), les arbres binaires de recherche peuvent changer de nom (AVL, arbre coloré, etc.)

TAD Arbre *n-aire* (nommé aussi Arbre tout court) 1 / 2

Définition

Un arbre *n-aire* est un arbre avec *n* fils



TAD Arbre *n-aire* (nommé aussi Arbre tout court) 2 / 2

Nom:	Arbre	
Paramètre:	Element	
Utilise:	**Booleen**,Liste	
Opérations:	arbre:	→ Arbre
	ajouterRacine:	Element × Liste<Arbre> → Arbre
	estVide:	Arbre → **Booleen**
	obtenirElement:	Arbre ↛ Element
	obtenirFils:	Arbre ↛ Liste<Arbre>
Axiomes:	- estVide(arbre())	
	- non estVide(ajouterRacine(*e*,*l*))	
	- obtenirElement(ajouterRacine(*e*,*l*))=*e*	
	- obtenirFils(ajouterRacine(*e*,*l*))=*l*	
Préconditions:	obtenirElement(a):	*non*(*estVide*(*a*))
	...:	...

Conclusion

Analyse - Conception - Développement

- Dans ce cours nous avons listé un ensemble de TAD collections
- À partir de maintenant, nous considérerons que l'on possède ces types lors de la conception détaillée
- Cependant certains langages ne proposent pas des implantations de ces TAD : il faut les développer
  - Nous allons voir dans les prochains cours comment nous pouvons les concevoir et les implanter en C

Références...

- Cours “Structure de données linéaires” de Christophe Hancart de l'Université de Rouen
- Conception et Programmation Objet de Bertrand Meyer Eyrolles ISBN : 2-212-09111-7