51IF2IK3 2007/2008 - Examen

durée: 2h30 - aucun document autorisé

La figure 1 donne le graphe de dépendance entre les chapitres du livre *"Mon livre"* : une flèche $x \to y$ signifie que la lecture du chapitre x est nécessaire à la compréhension du chapitre y. Ainsi, le chapitre 6 dépend directement des chapitres 2 et 3 ; pour le comprendre il faut avoir lu les chapitres 0, 1, 2 et 3.

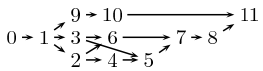

FIG. 1 – Graphe de dépendance de *"Mon livre"*

L'exemple *"Mon livre"* doit vous aider à réfléchir aux implémentations demandées, mais celles-ci doivent évidemment fonctionner pour tout autre livre.

On fait l'hypothèse que le livre peut être lu dans l'ordre croissant des chapitres (c'est-à-dire du premier au dernier chapitre). Cela implique en particulier que si le chapitre c_1 dépend du chapitre c_0, alors $c_1 > c_0$ (ie le chapitre c_1 se trouve après le chapitre c_0 dans le livre), en particulier le chapitre 0 est donc sans prérequis. Une autre conséquence de cette hypothèse est que le graphe ne possède pas de cycle : il n'est pas possible de trouver une suite de chapitres $(c_i)_{0 \leq i \leq n}$ telle que c_0 dépend de c_1 qui dépend de c_2 qui dépend ...qui dépend de c_n qui dépend de c_0.

Le lecteur cherche toutes les suites croissantes de chapitres à lire pour pouvoir lire un chapitre donné. Par exemple avec le livre de la figure 1, pour lire le chapitre 6, les suites croissantes cherchées sont les trois suites : $0 \to 1 \to 2 \to 3 \to 4 \to 5 \to 6$ **ou** $0 \to 1 \to 2 \to 3 \to 4 \to 6$ **ou** $0 \to 1 \to 2 \to 3 \to 6$.

Les algorithmes et programmes demandés se réfèrent toujours à ce problème. Les solutions données devront donc respecter l'ordre croissant.

De plus, pour cette épreuve, on s'interdit de reculer dans le choix des chapitres[1]**.** Notez également qu'on cherche les suites **croissantes** de chapitres, il est donc inutile de tester les chapitres dont le numéro est supérieur à celui qu'on cherche à atteindre.

Exercice 1 Backtracking

On suppose que l'ordre normal de choix de chapitre est l'ordre croissant.

1. Donner une suite de chapitres convenable pour pouvoir lire le chapitre n.

[1] Cette décision vous est imposée, mais choisir les chapitres en ordre décroissant n'est pas absurde d'un point de vue algorithmique.

2. Dérouler à la main l'algorithme de *backtracking* pour un lecteur qui voudrait connaître toutes les façons (croissantes) de lire le chapitre 6 sans sauter un chapitre dont ce dernier dépend, pour le livre de la figure 1. Par "dérouler à la main", on entend donner pas à pas le contenu de la pile et les actions faites pour modifier ce contenu.
3. Ecrire (en français) l'algorithme de *backtracking* général pour afficher toutes les suites croissantes possibles de chapitres pour lire un chapitre donné. En particulier, expliquer en détail comment on sait qu'on a trouvé une solution.
4. Adapter la méthode précédente avec un prétraitement de manière à sélectionner plus efficacement les chapitres pouvant être lus à chaque étape.

Exercice 2 *Programmation*

Les implémentations demandées sont à faire dans le *package* `if2ik3.examen`.

On pourra utiliser les classes `java.util.Stack<Integer>` et `if2ik3.examen.Livre` dont les documentations sont fournies en annexe.

Le graphe de dépendance d'une instance quelconque de la classe `Livre` est supposé obéïr à l'hypothèse de base faite précédemment et être clos transitivement : si c_2 dépend de c_1 et c_1 de c_0,alors c_2 dépend de c_0. **On ne vous demande ni programmer cette clôture transitive, ni de programmer sa vérification**, c'est le constructeur de la classe `Livre` qui s'en charge (comme précisé dans la documentation de cette classe fournie en annexe) et il ne fait pas partie des questions.

On désire implémenter deux autres classes :
- la classe `Lecture` dont la documentation et une partie du code sont données en annexe et qui implémenterait l'algorithme de la question 3 de l'exercice 1,
- la classe `LectureRapide` dont la documentation est donnée en annexe et qui implémenterait l'algorithme de la question 4 de l'exercice 1.

Il est bien entendu que tout le code que vous proposerez doit être en accord avec la documentation et le code fournis, qui doivent donc vous guider dans vos choix. Veillez en particulier à bien respecter les noms des méthodes.

Il est conseillé de lire toutes les questions de cet exercice et d'y réfléchir globalement plutôt que de se lancer directement dans le code.

1. Proposer une implémentation de la méthode `lireChapitre` de la classe `Lecture` qui implémente l'algorithme de la question 3 de l'exercice 1 quand elle est invoquée à travers une instance de la classe `Lecture` et l'algorithme de la question 4 de l'exercice 1 quand elle est invoquée à travers une instance de la classe `LectureRapide`. Préciser quelles sont les autres méthodes et/ou constructeurs à redéfinir pour que cela fonctionne.
2. Donner une implémentation des méthodes marquées à faire dans la classe `Lecture`.
3. Donner une implémentation des méthodes et/ou constructeurs à redéfinir dans la classe `LectureRapide` pour la question 1.

Vous trouverez en annexe :
- la documentation de la classe `Livre`,
- la documentation de la classe `Lecture`,
- la documentation de la classe `LectureRapide`,
- une partie du code de la classe `Lecture`.
- la documentation de la classe `java.util.Stack<E>`,

if2ik3.examen

Class Livre

```
java.lang.Object
  └ if2ik3.examen.Livre
```

Direct Known Subclasses:
Lecture

```
public class Livre
extends java.lang.Object
```

Author:
klimann

Constructor Summary

Livre()	La classe Livre permet de représenter le graphe de dépendance d'un livre.

Method Summary

boolean	estDirectementDependant(int chapitre, int autreChapitre)
int	getNbChapitres()
boolean	peutEtreLu(boolean[] dejaLus, int chapitre)

Methods inherited from class java.lang.Object
equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Livre

```
public Livre()
```

La classe Livre permet de représenter le graphe de dépendance d'un livre.
Une instance de cette classe est supposée obéïr aux hypothèses suivantes :

- on peut lire le livre dans l'ordre croissant des chapitres,
- le graphe de dépendance ne comporte pas de cycle (conséquence de l'hypothèse précédente).

Method Detail

getNbChapitres

```
public int getNbChapitres()
```

Returns:
le nombre de chapitres de l'objet livre à travers laquelle elle est invoquée

estDirectementDependant

```
public boolean estDirectementDependant(int chapitre,
                                       int autreChapitre)
```

Parameters:
`chapitre` - un numéro de chapitre
`autreChapitre` - un autre numéro de chapitre
Returns:
true si `chapitre` dépend directement d'`autreChapitre` (sans chapitre intermédiaire), false sinon

peutEtreLu

```
public boolean peutEtreLu(boolean[] dejaLus,
                          int chapitre)
```

Parameters:
`dejaLus` - un tableau des chapitres déjà lus
`chapitre` - un numéro de chapitre
Returns:
true si on peut lire le chapitre, false sinon

if2ik3.examen

Class Lecture

java.lang.Object
  └ if2ik3.examen.Livre
    └ **if2ik3.examen.Lecture**

Direct Known Subclasses:
LectureRapide

public class **Lecture**
extends Livre

Author:
klimann

Constructor Summary

Lecture()	initialise la pile des chapitres déjà lus

Method Summary

void	**ajouteChapitre**(int chapitre) ajoute le paramètre dans la pile et le marque comme déjà lu
boolean	**chapitreLu**(int chapitre)
int	**chapitreSuivant**(int chapitre)
void	**lireChapitre**(int chapitreALire) affiche toutes les suites croissantes possibles de chapitres pour arriver à lire le chapitre dont le numéro est passé en paramètre
int	**supprimeChapitre**() enlève le paramètre de la pile et le marque comme non encore lu

Methods inherited from class if2ik3.examen.Livre

estDirectementDependant, getNbChapitres, peutEtreLu

Methods inherited from class java.lang.Object

equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Lecture

public **Lecture**()

initialise la pile des chapitres déjà lus

Method Detail

chapitreSuivant

public int **chapitreSuivant**(int chapitre)
throws java.lang.Exception

Parameters:
chapitre - un numéro de chapitre
Returns:
le chapitre suivant qui peut être lu
Throws:
java.lang.Exception - s'il n'y a pas de chapitre suivant qui puisse être lu

chapitreLu

public boolean **chapitreLu**(int chapitre)

Parameters:
chapitre - un numéro de chapitre
Returns:
true si le chapitre a déjà été lu, false sinon

ajouteChapitre

public void **ajouteChapitre**(int chapitre)

ajoute le paramètre dans la pile et le marque comme déjà lu

Parameters:
chapitre - un numéro de chapitre

supprimeChapitre

public int **supprimeChapitre**()

enlève le paramètre de la pile et le marque comme non encore lu

Returns:
le sommet de la pile

lireChapitre

public void **lireChapitre**(int chapitreALire)

affiche toutes les suites croissantes possibles de chapitres pour arriver à lire le chapitre dont le numéro est passé en paramètre

Parameters:
chapitreALire - le chapitre qu'on veut lire

if2ik3.examen

Class LectureRapide

java.lang.Object
  └ if2ik3.examen.Livre
    └ if2ik3.examen.Lecture
      └ **if2ik3.examen.LectureRapide**

public class **LectureRapide**
extends Lecture

Author:
klimann

Constructor Summary

LectureRapide()

Method Summary

void	ajouteChapitre(int chapitre) ajoute le paramètre dans la pile et le marque comme déjà lu
int	chapitreSuivant(int chapitre)
int	supprimeChapitre() enlève le paramètre de la pile et le marque comme non encore lu

Methods inherited from class if2ik3.examen.Lecture

chapitreLu, lireChapitre

Methods inherited from class if2ik3.examen.Livre

estDirectementDependant, getNbChapitres, peutEtreLu

Methods inherited from class java.lang.Object

equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

LectureRapide

public **LectureRapide**()

Method Detail

chapitreSuivant

public int **chapitreSuivant**(int chapitre)
throws java.lang.Exception

Overrides:
chapitreSuivant in class Lecture
Parameters:
chapitre - un numéro de chapitre
Returns:
le chapitre suivant qui peut être lu en utilisant un prétraitement
Throws:
java.lang.Exception - s'il n'y a pas de chapitre suivant qui puisse être lu
See Also:
Lecture.chapitreSuivant(int)

supprimeChapitre

public int **supprimeChapitre**()

Description copied from class: Lecture
enlève le paramètre de la pile et le marque comme non encore lu

Overrides:
supprimeChapitre in class Lecture
Returns:
le sommet de la pile

ajouteChapitre

public void **ajouteChapitre**(int chapitre)

Description copied from class: Lecture
ajoute le paramètre dans la pile et le marque comme déjà lu

Overrides:
ajouteChapitre in class Lecture
Parameters:
chapitre - un numéro de chapitre

```java
package if2ik3.examen;
import java.util.*;


public class Lecture extends Livre {
	private boolean[] dejaLus;    // chapitres deja lus
	private Stack<Integer> pile;  // pile de backtracking

	public Lecture() {
		super();
		this.dejaLus = new boolean[this.getNbChapitres()];
		this.pile = new Stack<Integer>();
	}


	public int chapitreSuivant(int chapitre) throws Exception {
	  // TODO
	}


	public boolean chapitreLu(int chapitre){
		return this.dejaLus[chapitre];
	}


	public void ajouteChapitre(int chapitre){
		this.pile.push(chapitre);
		this.dejaLus[chapitre] = true;
	}

	public int supprimeChapitre(){
	  //TODO
	}

	public void lireChapitre(int chapitreALire){
	  //TODO
	}

}
```


java.util

Class Stack<E>

java.lang.Object
  └ java.util.AbstractCollection<E>
    └ java.util.AbstractList<E>
      └ java.util.Vector<E>
        └ **java.util.Stack<E>**

All Implemented Interfaces:
Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

---

public class **Stack<E>**
extends Vector<E>

The Stack class represents a last-in-first-out (LIFO) stack of objects. It extends class Vector with five operations that allow a vector to be treated as a stack. The usual push and pop operations are provided, as well as a method to peek at the top item on the stack, a method to test for whether the stack is empty, and a method to search the stack for an item and discover how far it is from the top.

When a stack is first created, it contains no items.

Since:
JDK1.0
See Also:
Serialized Form

Field Summary

Fields inherited from class java.util.Vector
capacityIncrement, elementCount, elementData

Fields inherited from class java.util.AbstractList
modCount

Constructor Summary

Stack() Creates an empty Stack.

Method Summary

boolean	**empty**() Tests if this stack is empty.
E	**peek**() Looks at the object at the top of this stack without removing it from the stack.

E	**pop**() Removes the object at the top of this stack and returns that object as the value of this function.
E	**push**(E item) Pushes an item onto the top of this stack.
int	**search**(Object o) Returns the 1-based position where an object is on this stack.

Methods inherited from class java.util.Vector
add, add, addAll, addAll, addElement, capacity, clear, clone, contains, containsAll, copyInto, elementAt, elements, ensureCapacity, equals, firstElement, get, hashCode, indexOf, indexOf, insertElementAt, isEmpty, lastElement, lastIndexOf, lastIndexOf, remove, remove, removeAll, removeAllElements, removeElement, removeElementAt, removeRange, retainAll, set, setElementAt, setSize, size, subList, toArray, toArray, toString, trimToSize

Methods inherited from class java.util.AbstractList
iterator, listIterator, listIterator

Methods inherited from class java.lang.Object
finalize, getClass, notify, notifyAll, wait, wait, wait

Methods inherited from interface java.util.List
iterator, listIterator, listIterator

Constructor Detail

Stack

```
public Stack()
```

Creates an empty Stack.

Method Detail

push

```
public E push(E item)
```

Pushes an item onto the top of this stack. This has exactly the same effect as:

```
addElement(item)
```

Parameters:
item - the item to be pushed onto this stack.
Returns:
the item argument.
See Also:
Vector.addElement(E)

---

pop

```
public E pop()
```

Removes the object at the top of this stack and returns that object as the value of this function.

Returns:
The object at the top of this stack (the last item of the Vector object).
Throws:
EmptyStackException - if this stack is empty.

---

peek

```
public E peek()
```

Looks at the object at the top of this stack without removing it from the stack.

Returns:
the object at the top of this stack (the last item of the Vector object).
Throws:
EmptyStackException - if this stack is empty.

---

empty

```
public boolean empty()
```

Tests if this stack is empty.

Returns:
true if and only if this stack contains no items; false otherwise.

---

search

```
public int search(Object o)
```

Returns the 1-based position where an object is on this stack. If the object o occurs as an item in this stack, this method returns the distance from the top of the stack of the occurrence nearest the top of the stack; the topmost item on the stack is considered to be at distance 1. The equals method is used to compare o to the items in this stack.

Parameters:
o - the desired object.
Returns:
the 1-based position from the top of the stack where the object is located; the return value -1 indicates that the object is not on the stack.

Submit a bug or feature
For further API reference and developer documentation, see Java 2 SDK SE Developer Documentation. That documentation contains more detailed, developer-targeted descriptions, with conceptual overviews, definitions of terms, workarounds, and working code examples.

Copyright 2004 Sun Microsystems, Inc. All rights reserved. Use is subject to license terms. Also see the documentation redistribution policy.