

Angabe zur Klausur

Objektorientierte Programmierung

25. Juni 2024

In dieser Klausur geht es um eine der bekanntesten Geschichten der griechischen Mythologie, nämlich um Theseus und den Minotaurus. Für die Lösung der Aufgabe ist es nicht notwendig, die Geschichte zu kennen, aber als Hintergrundinformation sei erwähnt, dass der Minotaurus ein Ungeheuer in Menschengestalt mit einem Stierkopf war, das vom kretischen König Minos in einem Labyrinth gefangen gehalten wurde. Alle neun Jahre mussten die Athener sieben junge Frauen und sieben junge Männer nach Kreta schicken, wo sie im Labyrinth des Minotaurus geopfert wurden. Der athenische Held Theseus setzte dem ein Ende, indem er den Minotaurus im Labyrinth aufspürte und im Kampf tötete. Damit Theseus den Weg aus dem Labyrinth finden konnte, hatte ihm Ariadne ein Knäuel mit einem Wollfaden gegeben. Diesen Faden befestigte er am Eingang des Labyrinths und wickelte ihn auf seinem Weg durch das Labyrinth immer weiter ab, so dass er zu jeder Zeit wusste, wie er wieder herausfinden konnte. Dieses Prinzip, den eingeschlagenen Weg zu markieren, um ihn später wieder finden zu können, wird seither *Ariadnefaden*, engl. *Ariadne's thread*, genannt.

Sie sollen Theseus nun helfen, zuerst den Minotaurus aufzuspüren und dann den Weg entlang des Ariadnefadens zurück zum Eingang zu finden. Dass Theseus eigentlich erst den Minotaurus besiegen muss, bevor er den Rückweg aus dem Labyrinth antreten kann, wird hier nicht thematisiert.

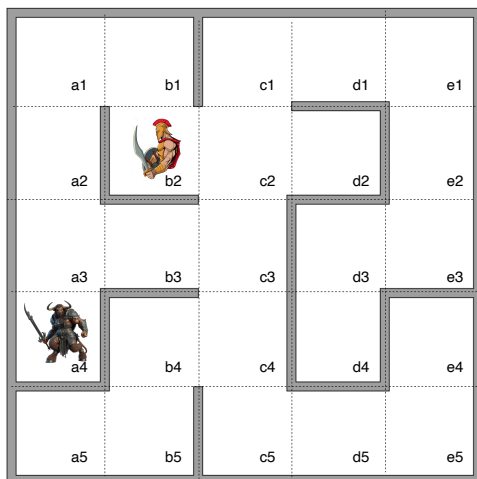


Abbildung 1: Minotaurus und Theseus im Labyrinth.

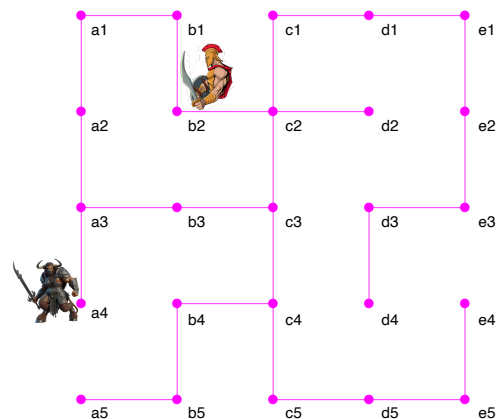


Abbildung 2: Das Labyrinth in Abb. 1 als ungerichteter Graph.

Die Lösung soll für ein beliebiges Labyrinth funktionieren, das aus verschiedenen Räumen besteht, die durch Wände voneinander getrennt sind, zwischen denen man sich aber beliebig bewegen kann, wenn keine Wände im Weg sind. Ein Raum ist der Eingang des Labyrinths, wo es betreten wird, und in einem Raum wartet der Minotaurus. Abb. 1 zeigt ein Beispiel. Die Räume haben hier eine quadratische Grundfläche und sind durch die grau dargestellten Wände voneinander getrennt. Jeder Raum ist mit einer Kombination aus Buchstabe und Zahl bezeichnet; der Minotaurus lebt im Raum a4, und Theseus ist auf seiner Suche bis b2 vorgedrungen. Im Folgenden wird das Labyrinth als ungerichteter Graph dargestellt: Jeder Raum wird durch einen Knoten repräsentiert, und die Kanten stellen die Möglichkeiten dar, von einem Raum in einen benachbarten Raum zu gelangen. Die Kanten sind ungerichtet, d.h. man kann in beide Richtungen gehen. Abb. 2 zeigt den Graphen für das Labyrinth in Abb. 1; die Knoten sind wie die zugehörigen Räume bezeichnet. Im Folgenden werden die Knoten des Graphen der Einfachheit halber als Räume bezeichnet.

Um den Minotaurus im Labyrinth zu finden, muss Theseus den Graphen systematisch durch-

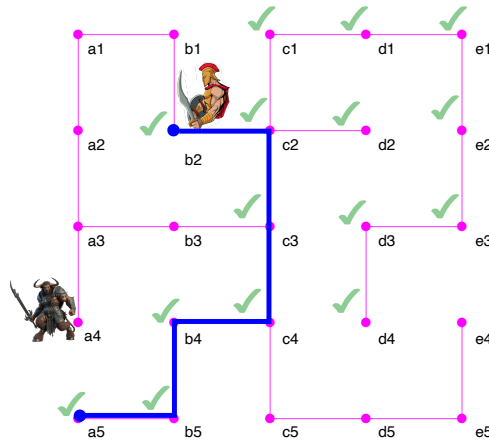


Abbildung 3: Theseus und der Ariadnefaden.

suchen, bis er den Knoten (d.h. Raum) erreicht, wo der Minotaurus auf ihn wartet. Abb. 3 zeigt einen „Schnappschuss“ dieser Suche im Labyrinth: Theseus hat das Labyrinth im Raum a5 betreten, wo er das eine Ende des blau dargestellten Ariadnefadens befestigt hat. Der Ariadnefaden kann als eine Folge von Knoten (Räumen) interpretiert werden, die er durchläuft, in diesem Fall a5, b5, b4, c4, c3, c2 und b2. Theseus befindet sich also immer im Raum am Ende der Folge. Außerdem merkt sich Theseus alle Räume, die er bereits betreten hat; in Abb. 3 sind diese Knoten mit einem grünen Häkchen markiert.

Bei der Suche nach dem Minotaurus wendet Theseus die Methode der *Tiefensuche* (engl. *depth first*) mit *Backtracking* an: Ausgehend vom Eingang versucht er sukzessive nur Räume zu betreten, in denen er noch nicht war. Immer wenn er dabei in eine Sackgasse gerät, von der aus er keinen noch nicht besuchten Raum betreten kann, folgt er dem Ariadnefaden zurück (*Backtracking*), bis er wieder zu einem Raum gelangt, von dem aus er wieder einen noch nicht besuchten Raum betreten kann. Diese Suche endet, wenn er irgendwann den Raum mit dem Minotaurus erreicht. Nach dem Kampf, der hier nicht thematisiert wird, folgt er dem Ariadnefaden zurück zum Eingang des Labyrinths.

In Abb. 3 sieht man, dass Theseus bereits d2 und die leeren Räume beginnend mit c1 besucht hat, bevor er dann b2 erreicht hat.

Für die Bearbeitung der nachstehenden Aufgaben gelten grundsätzlich die folgenden Hinweise:

- Schreiben Sie Code grundsätzlich in der Programmiersprache Java.
- Eventuell nötige Package-Vereinbarungen und Import-Anweisungen dürfen Sie in den folgenden Aufgaben weglassen; sie werden nicht bewertet. Gehen Sie aber davon aus, dass alle der folgenden Klassen sich wie angegeben im Package `maze` befinden und man sie und ihre Methoden auch außerhalb von `maze` benutzen können soll, es sei denn, es ist etwas anderes explizit gefordert.

Aufgabe 1 (Klasse Room): 8 Punkte

Abb. 5 auf Seite 7 zeigt ein Klassendiagramm und darin die Klasse `Room`, womit die Knoten des ungerichteten Graphen, also die Räume des Labyrinths repräsentiert werden. Das unveränderliche Attribut `name` enthält die Bezeichnung des Raums, z.B. `a4`, `minotaur` gibt an, ob der Minotaurus in diesem Raum lebt, und die Assoziation `reachable` referenziert alle über eine Kante verbundenen, anderen Räume. Initial soll der Minotaurus sich nicht in dem Raum befinden. Die Methoden `getName()`, `hasMinotaur()` und `getReachable()` sind die entsprechenden Getter-Methoden, und `setMinotaur(...)` eine Setter-Methode. Die `addReachable`-Methode soll eine Kante zum angegebenen Knoten ziehen.

Schreiben Sie die vollständige Klasse `Room`, die Getter-Methoden `getName()`, `hasMinotaur()` und `getReachable()` müssen Sie aber **nicht** hinschreiben.

Hinweise:

- Achten Sie auf den Rückgabebetyp der `getReachable`-Methode und eine geeignete Realisierung des `reachable`-Attributs.
- Die Kanten sind ungerichtet. Daher muss der Aufruf `a1.addReachable(b1)` zur Folge haben, dass anschließend sowohl `b1` in `a1.getReachable()` enthalten ist als auch `a1` in `b1.getReachable()`.

Aufgabe 2 (Beispiellabyrinth): 7 Punkte

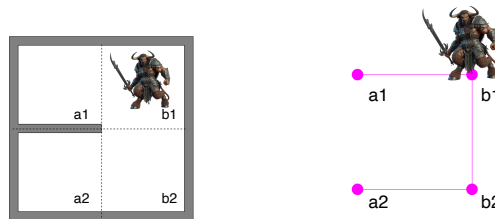


Abbildung 4: Ein kleines Labyrinth und seine Repräsentation als ungerichteter Graph.

Die Methode `createSmallMaze` der Klasse `SampleMaze` im Klassendiagramm (Abb. 5) erzeugt ein Labyrinth, das in Abb. 4 dargestellt ist. Von der Methode wird der Raum `a2` als Eingang des Labyrinths zurückgegeben. Schreiben Sie die vollständige Klasse `SampleMaze`.

Aufgabe 3 (Testfall): 15 Punkte

Im Klassendiagramm (Abb. 5) ist die Iteratorklasse `AriadneIterator` angegeben. Dem Konstruktor wird als Parameter ein Raum übergeben, der als Eingang zum Labyrinth dient. Der Iterator iteriert dann nacheinander über die Räume, die Theseus durchläuft, beginnend mit dem zuvor angegebenen Eingang des Labyrinths, bis er den Minotaurus gefunden hat. Danach iteriert er über die Räume, die Theseus auf seinem Rückweg entlang des Ariadnefadens durchquert, d.h. am Anfang und am Ende der Iteration steht der Eingang des Labyrinths.

In dieser Aufgabe wird wieder das (zugegebenermaßen) sehr kleine Labyrinth in Abb. 4 verwendet. Wenn der Eingang `a2` in der Variablen `entry` gespeichert ist und der Iterator dann mit `new AriadneIterator(entry)` erzeugt wird, iteriert er über die Räume `a2`, `b2`, `b1`, `a2` und `a2` in dieser Reihenfolge.

Erstellen Sie die Testklasse `AriadneIteratorTest`, die diesen Testfall mit JUnit 4 implementiert.

Hinweis: Nutzen Sie die `createSmallMaze`-Methode (siehe Aufgabe 2), um das kleine Labyrinth zu erzeugen.

Aufgabe 4 (AriadneIterator): 7 + 23 = 30 Punkte

Der Ariadne-Iterator (Abb. 5) folgt der eingangs beschriebenen Suchstrategie gemäß Tiefensuche mit Backtracking. Er speichert die von Theseus bereits besuchten Räume in `visited`, und `ariadnesThread` speichert die Folge der Räume, durch die der Ariadnefaden läuft (siehe erneut Abb. 3). Betritt Theseus einen neuen Raum, dann fügt er den neuen Raum ans Ende von `ariadnesThread` an. Wenn er entlang des Ariadnefadens zum vorhergehenden Raum zurückkehren muss, entfernt er den Eintrag in `ariadnesThread`. Damit entspricht die aktuelle Position von Theseus immer dem letzten Eintrag in `ariadnesThread`.

Hinweis: Sie können `ariadnesThread` mit einer beliebigen Liste realisieren, es empfiehlt sich aber die Verwendung eines *Kellers* (engl. *Stack*) in Form der Klasse `Stack` der Java-API (siehe folgende Seiten) und hier insbesondere die Methoden `push`, `pop` und `peek`.

Die Idee hinter dem Ariadne-Iterator ist wie folgt: Wird eine Instanz erzeugt, dann merkt sich die Instanz den Eingang, der als Konstruktorparameter übergeben wurde. Da Theseus das Labyrinth aber noch nicht betreten hat, ist die Folge der Räume in `ariadnesThread` noch leer, ebenso `visited`. Die Folge der Räume, über die der Iterator iteriert, wird sukzessive von der Methode `next()` zurückgegeben. Bei jedem Aufruf ruft `next` zuerst `step()` auf und gibt dann die aktuelle Position von Theseus zurück. Diese befindet sich, wie oben beschrieben, immer am Ende des Ariadnefadens, ist also der letzte Eintrag in `ariadnesThread`. Das Verhalten der `step`-Methode ist mit dem Zustandsdiagramm in Abb. 6 auf Seite 7 angegeben.

Im Zustandsdiagramm werden verschiedene Methode des Ariadne-Iterators aufgerufen. Ihre Bedeutung ist:

`init()` lässt Theseus das Labyrinth am Eingang betreten, d.h. man muss den Eingangsraum am Ende von `ariadnesThread` ablegen und den Raum in `visited` eintragen.

`foundMinotaur()` liefert genau dann **`true`** zurück, wenn Theseus den Raum erreicht hat, in dem der Minotaurus haust, sonst **`false`**.

`seekingStep()` und `nextForward()` sind für die Klausurlösung nicht relevant, weil Sie den Zustand *Seeking Minotaur* nicht realisieren müssen.

`retreatingStep()` lässt Theseus entlang des Ariadnefadens einen Schritt zurück zum Eingang machen.

Das Zustandsdiagramm sollen Sie mit Hilfe des *State Patterns* umsetzen. Die Methode `hasNext()` des Ariadne-Iterators soll jeden Aufruf an den jeweiligen Zustand delegieren, d.h. der jeweilige Zustand entscheidet, ob die Iteration zu Ende ist oder nicht.

- a) Ergänzen Sie in Abb. 5 alle Angaben (z.B. Klassen, Attribute, Methoden, Generalisierungsbeziehungen etc.), um das Zustandsdiagramm mit Hilfe des State-Musters umzusetzen. Falls Sie innere Klassen ergänzen wollen, geben Sie an, innerhalb welcher anderen Klasse sie definiert sein sollen. Das müssen Sie nicht graphisch darstellen; ein textueller Kommentar reicht vollkommen aus.
- b) Implementieren Sie die Klasse `AriadneIterator` mit allen ihren Attributen und Methoden **mit Ausnahme von** `seekingStep()` und `nextForward()`. Realisieren Sie außerdem die beiden Zustände *Starting* und *Mission Accomplished* des Zustandsdiagramms in Abb. 6 mit allen von `State` vereinbarten Methoden.

Beachte: Den Zustand *Seeking Minotaur* müssen Sie **nicht** realisieren, und auch den Code für das Interface `State` müssen Sie **nicht** hinschreiben.

Module java.base
Package java.util
Interface **Iterator<E>**

Type Parameters:
E - the type of elements returned by this iterator

All Known Subinterfaces:
EventIterator, ListIterator<E>, PrimitiveIterator<T, T_CONS>, PrimitiveIterator.OfDouble, PrimitiveIterator.OfInt, PrimitiveIterator.OfLong, XMLStreamReader

All Known Implementing Classes:
BeanContextSupport.BCIterator, EventReaderDelegate, Scanner

public interface **Iterator<E>**

An iterator over a collection. Iterator takes the place of Enumeration in the Java Collections Framework. Iterators differ from enumerations in two ways:

- Iterators allow the caller to remove elements from the underlying collection during the iteration with well-defined semantics.
- Method names have been improved.

This interface is a member of the Java Collections Framework.

API Note:
An Enumeration can be converted into an Iterator by using the Enumeration.asIterator() method.

Since:
1.2

See Also:
Collection, ListIterator, Iterable

Method Summary

All Methods	Instance Methods	Abstract Methods	Default Methods
Modifier and Type	Method	Description	
default void	forEachRemaining(Consumer<? super E> action)	Performs the given action for each remaining element until all elements have been processed or the action throws an exception.	
boolean	hasNext()	Returns true if the iteration has more elements.	
E	next()	Returns the next element in the iteration.	
default void	remove()	Removes from the underlying collection the last element returned by this iterator (optional operation).	

Method Details

hasNext
boolean hasNext()
Returns true if the iteration has more elements. (In other words, returns true if next() would return an element rather than throwing an exception.)
Returns: true if the iteration has more elements
next
E next()
Returns the next element in the iteration.
Returns: the next element in the iteration
Throws: NoSuchElementException - If the iteration has no more elements

remove

default void remove()

Removes from the underlying collection the last element returned by this iterator (optional operation). This method can be called only once per call to next().

The behavior of an iterator is unspecified if the underlying collection is modified while the iteration is in progress in any way other than by calling this method, unless an overriding class has specified a concurrent modification policy.

The behavior of an iterator is unspecified if this method is called after a call to the forEachRemaining method.

Implementation Requirements:

The default implementation throws an instance of UnsupportedOperationException and performs no other action.

Throws:

UnsupportedOperationException - if the remove operation is not supported by this iterator
IllegalStateException - if the next method has not yet been called, or the remove method has already been called after the last call to the next method

forEachRemaining

default void forEachRemaining(Consumer<? super E> action)

Performs the given action for each remaining element until all elements have been processed or the action throws an exception. Actions are performed in the order of iteration, if that order is specified. Exceptions thrown by the action are relayed to the caller.

The behavior of an iterator is unspecified if the action modifies the collection in any way (even by calling the remove method or other mutator methods of Iterator subtypes), unless an overriding class has specified a concurrent modification policy.

Subsequent behavior of an iterator is unspecified if the action throws an exception.

Implementation Requirements:

The default implementation behaves as if:

```
while (hasNext())  
    action.accept(next());
```

Parameters:

action - The action to be performed for each element

Throws:

NullPointerException - if the specified action is null

Since:
1.8

Report a bug or suggest an enhancement
For further API reference and developer documentation see the Java SE Documentation, which contains more detailed, developer-targeted descriptions with conceptual overviews, definitions of terms, workarounds, and working code examples. Other versions.
Java is a trademark or registered trademark of Oracle and/or its affiliates in the US and other countries.
Copyright © 1993, 2024, Oracle and/or its affiliates. 500 Oracle Parkway, Redwood Shores, CA 94065 USA.
All rights reserved. Use is subject to license terms and the documentation redistribution policy. Modify Cookie-Einstellungen, Modify Ad Choices.

Module java.base
Package java.util
Class Stack<E>
java.lang.Object
java.util.AbstractCollection<E>
java.util.AbstractList<E>
java.util.Vector<E>
java.util.Stack<E>

Type Parameters:
E - Type of component elements

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

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.

A more complete and consistent set of LIFO stack operations is provided by the Deque interface and its implementations, which should be used in preference to this class. For example:

Deque<Integer> stack = new ArrayDeque<Integer>();

Since:
1.0

See Also:
Serialized Form

Field Summary

Fields declared in class java.util.Vector

capacityIncrement, elementCount, elementData

Fields declared in class java.util.AbstractList

modCount

Constructor Summary

Constructors

Constructor
Stack()
Creates an empty Stack.

Method Summary

All Methods	Instance Methods	Concrete Methods
Modifier and Type	Method	Description
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 declared in class java.util.Vector

add, addAll, addAll, addElement, capacity, clear, clone, contains, containsAll, copyInto, elementAt, elementAt, ensureCapacity, equals, firstElement, forEach, get, hashCode, indexOf, insertElementAt, isEmpty, iterator, lastElement, lastIndexOf, lastIndexOf, listIterator, listIterator, remove, remove, removeAllElements, removeElement, removeElementAt, removeIf, removeRange, replaceAll, retainAll, set, setElementAt, setSize, size, splitIterator, subList, toArray, toString, trimToSize

Methods declared in class java.lang.Object

finalize, getClass, notify, notifyAll, wait, wait, wait

Methods declared in interface java.util.Collection

parallelStream, stream, toArray

Methods declared in interface java.util.List

addFirst, addLast, addLast, getFirst, getLast, removeFirst, removeLast, reversed, sort

Constructor Details

Stack

public Stack()
Creates an empty Stack.

Method Details

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 equal's 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.

Name:

Matrikelnummer:

Abbildung 5: Klassendiagramm

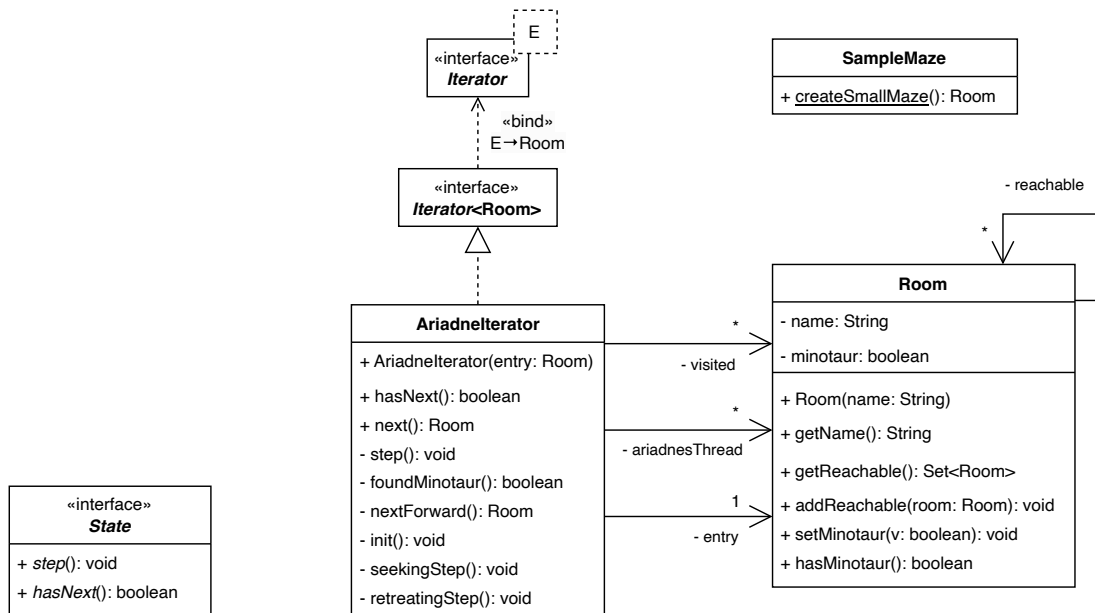


Abbildung 6: Zustandsdiagramm des Ariadne-Iterators.

