

Angabe zur Klausur

Objektorientierte Programmierung

16. September 2024

Wir befinden uns im Jahre 50 vor Christus. Ganz Gallien ist von den Römern besetzt. Ganz Gallien? Nein! Ein von unbeugsamen Galliern bevölkertes Dorf hört nicht auf, dem Eindringling Widerstand zu leisten.

So beginnt jeder Asterix-Band, und um dieses Dorf (siehe Abb. 1) geht es in dieser Klausur. Für die Lösung der Aufgabe ist es nicht notwendig, die Geschichten zu kennen, aber als Hintergrundinformation sei erwähnt, dass der Erfolg der unbeugsamen Gallier auf einem Zaubertrank beruht, der vom Dorfdruiden *Miraculix* gebraut wird und für kurze Zeit übernatürliche Kräfte verleiht. *Obelix*, engster Freund von *Asterix*, war als Kind in den Zaubertrank gefallen und ist seitdem immer übernatürlich stark. Trotzdem will er bei jeder sich bietenden Gelegenheit von dem Zaubertrank kosten. *Miraculix* verweigert ihm den Zaubertrank aber jedesmal und schickt ihn fort.



figs/Dorf.jpg

Abbildung 1: Das gallische Dorf.

Konkret geht es hier um *Miraculix*, der irgendwo im Dorf seinen Zaubertrank braut. *Obelix* will natürlich wieder vom Zaubertrank kosten, weiß aber nicht, wo genau sich *Miraculix* mit dem Zaubertrank befindet. *Obelix* muss daher das ganze Dorf nach *Miraculix* absuchen. Sobald er ihn gefunden hat, verweigert *Miraculix* ihm erneut den Zaubertrank und schickt ihn fort, woraufhin *Obelix* schmallend das Dorf verlässt.

Bei der systematischen Suche nach *Miraculix* bedient sich *Obelix* einer Technik, die er sich von den alten Griechen abgeschaut hat: Um den Weg zurück aus dem Dorf zu finden, nutzt er ein Knäuel mit einem Wollfaden. Diesen Faden befestigte er am Dorfeingang und wickelte ihn auf seinem Weg durch das Dorf immer weiter ab, so dass er zu jeder Zeit weiß, wie er am Schluss wieder herausfindet. Dieses Prinzip, den eingeschlagenen Weg zu markieren, um ihn später wieder finden zu können, heißt bekanntlich *Ariadnefaden*, engl. *Ariadne's thread*.

Sie sollen *Obelix* nun helfen, zuerst *Miraculix* im Dorf aufzuspüren und dann den Weg entlang des *Ariadnefadens* zurück zum Dorfeingang zu finden. Die kurze Interaktion, während der *Obelix* vergeblich versucht, einen Schluck Zaubertrank von *Miraculix* zu bekommen, wird hier nicht thematisiert.

Die Lösung soll für ein beliebiges Dorf funktionieren, das aus verschiedenen Häusern und sonstigen Plätzen besteht, die durch kurze Wege miteinander verbunden sind. Der Dorfeingang ist auch ein „Platz“, wo *Obelix* das Dorf auf der Suche nach *Miraculix* betritt, und in einem

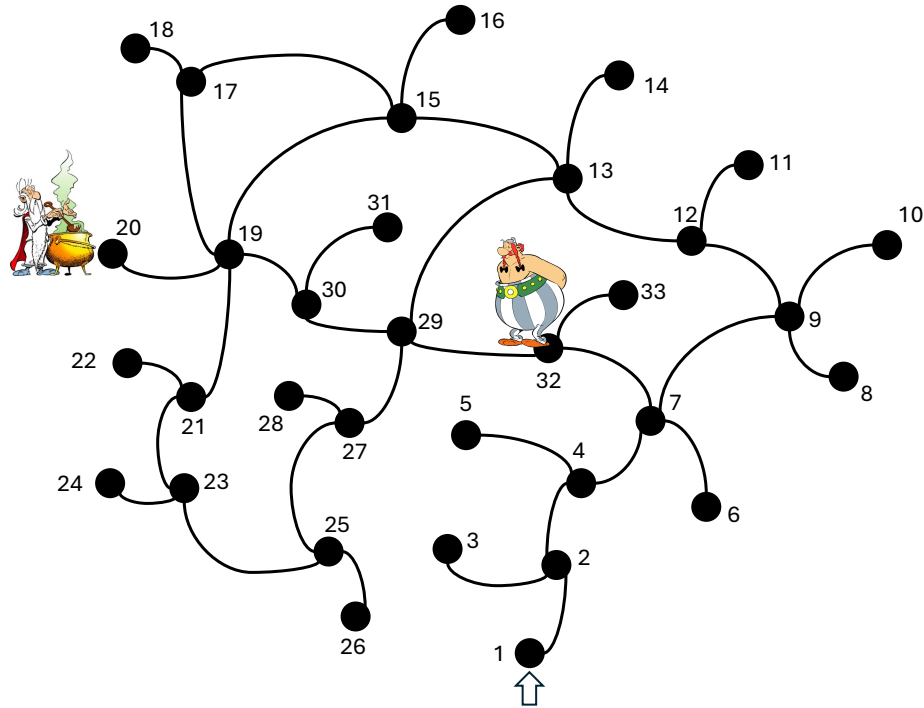


Abbildung 2: Eine Karte des gallischen Dorfs mit Obelix und Miraculix, dargestellt als ungerichteter Graph.

Haus oder an einem Platz braut Miraculix seinen Zaubertrank. Abb. 2 zeigt ein Beispiel, dargestellt als ungerichteter Graph. Jedes Haus und jeder Platz wird durch einen Knoten mit einer eindeutigen Nummer repräsentiert, und die Kanten stellen die Möglichkeiten dar, von einem Haus bzw. Platz zu einen „benachbarten“ Haus bzw. Platz zu gelangen. Die Kanten sind ungerichtet, d.h. man kann in beide Richtungen gehen. Im Folgenden bezeichnen wir Häuser und Plätze grundsätzlich nur als Knoten. Den Dorfeingang repräsentiert hier der Knoten 1, Miraculix braut seinen Zaubertrank in Knoten 20, und Obelix ist auf seinem Weg durch das Dorf bis zu Knoten 32 vorgedrungen.

Um Miraculix im Dorf zu finden, muss Obelix den Graphen systematisch durchsuchen, bis er den Knoten erreicht, wo Miraculix den Zaubertrank braut. Abb. 3 zeigt einen „Schnappschuss“ dieser Suche im Dorf: Obelix hat das Dorf im Knoten 1 betreten, wo er das eine Ende des rot und gestrichelt dargestellten Ariadnefadens befestigt hat. Der Ariadnefaden kann als eine Folge von Knoten interpretiert werden, die er durchläuft, in diesem Fall 1, 2, 4, 7, 9, 12, 13, 29 und 32. Obelix befindet sich also immer im Knoten am Ende der Folge. Außerdem merkt sich Obelix alle Knoten, die er bereits betreten hat; in Abb. 3 sind diese Knoten weiß mit schwarzem Rand dargestellt, die noch nicht besuchten Knoten dagegen vollkommen schwarz.

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

In Abb. 3 sieht man, dass Obelix bereits die „leeren“ Knoten 3, 5, 11 und 33 besucht hat. Da er mit Knoten 33 in eine Sackgasse geraten war, kehrte er entlang des Ariadnefadens zum Knoten 32 zurück. Den zuvor bis Knoten 33 abgewickelten Ariadnefaden hat er wieder aufgewickelt. Als nächstes wird er dem Ariadnefaden zurück zu Knoten 29 folgen (und dabei den Ariadnefaden auch wieder aufwickeln), weil er zwar von Knoten 32 den Knoten 7 erreichen könnte, diesen

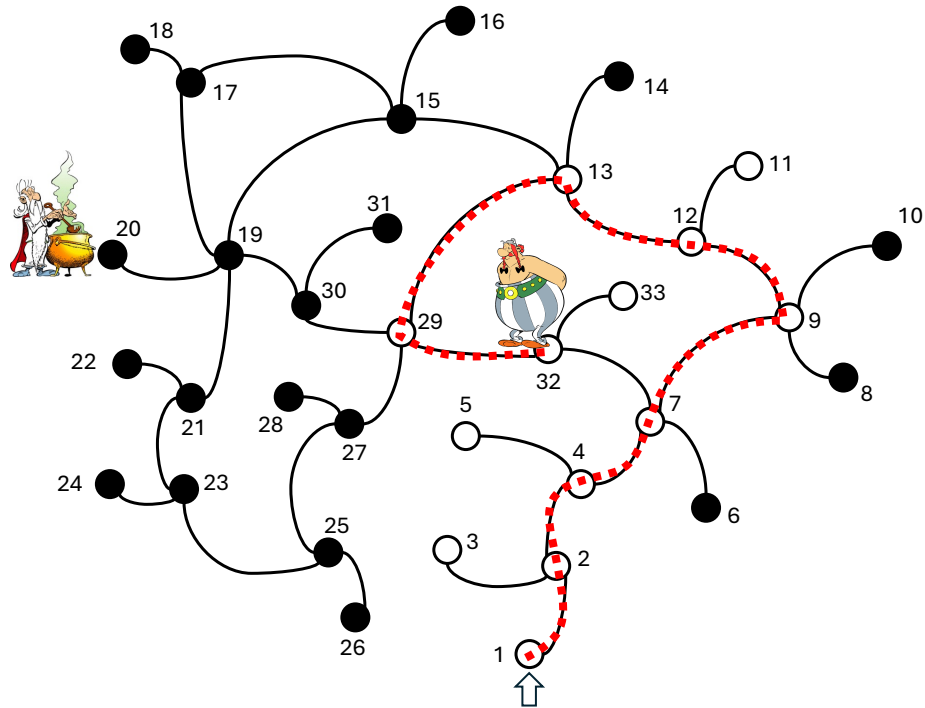


Abbildung 3: Obelix und der Ariadnefaden.

aber bereits vorher besucht hatte.

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 `village` befinden und man sie und ihre Methoden auch außerhalb von `village` benutzen können soll, es sei denn, es ist etwas anderes explizit gefordert.

Aufgabe 1 (Klasse Node): 8 Punkte

Abb. 5 auf Seite 8 zeigt ein Klassendiagramm und darin die Klasse `Node`, womit die Knoten des ungerichteten Graphen, also die Häuser und sonstigen Plätze des Dorfs repräsentiert werden. Das unveränderliche Attribut `number` enthält die Nummer des Knotens, z.B. 7, `miraculix` gibt an, ob Miraculix in diesem Knoten seinen Zaubersaft braut, und die Assoziation `reachable` referenziert alle über genau eine Kante erreichbaren Knoten. Initial soll Miraculix sich nicht in dem Knoten befinden. Die Methoden `getNumber()`, `hasMiraculix()` und `getReachable()` sind die entsprechenden Getter-Methoden, und `setMiraculix(...)` eine Setter-Methode. Die `addReachable`-Methode soll eine Kante zum angegebenen Knoten ziehen.

Schreiben Sie die vollständige Klasse `Node`, die Getter-Methoden `getNumber()`, `hasMiraculix()` 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 `a.addReachable(b)` (`a` und `b` sollen Knoten beinhalten) zur Folge haben, dass anschließend sowohl `b` in `a.getReachable()` enthalten ist als auch `a` in `b.getReachable()`.

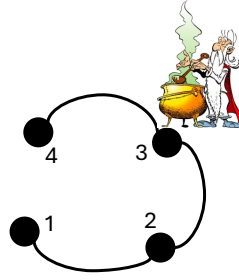


Abbildung 4: Karte eines (sehr) kleinen Dorfs als ungerichteter Graph.

Aufgabe 2 (Beispieldorf): 7 Punkte

Die Methode `createSmallVillage` der Klasse `SampleVillage` im Klassendiagramm (Abb. 5) erzeugt den ungerichteten Graphen des Dorfs, das in Abb. 4 dargestellt ist. Von der Methode wird der Knoten 1 als Dorfeingang zurückgegeben. Schreiben Sie die vollständige Klasse `SampleVillage`.

Aufgabe 3 (Testfall): 15 Punkte

Im Klassendiagramm (Abb. 5) ist die Iteratorklasse `AriadneIterator` angegeben. Dem Konstruktor wird als Parameter ein Knoten übergeben, der als Dorfeingang dient. Der Iterator iteriert dann nacheinander über die Knoten, die Obelix durchläuft, beginnend mit dem zuvor angegebenen Dorfeingang, bis er `Miraculix` gefunden hat. Danach iteriert er über die Knoten, die Obelix auf seinem Rückweg entlang des Ariadnefadens durchquert, d.h. am Anfang und am Ende der Iteration steht der Knoten des Dorfeingangs.

In dieser Aufgabe wird wieder das (zugegebenermaßen) sehr kleine Dorf in Abb. 4 verwendet. Wenn der Eingangsknoten 1 in der Variablen `entry` gespeichert ist und der Iterator dann mit `new AriadneIterator(entry)` erzeugt wird, iteriert er über die Knoten 1, 2, 3, 2 und 1 in dieser Reihenfolge.

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

Hinweis: Nutzen Sie die `createSmallVillage`-Methode (siehe Aufgabe 2), um das kleine Dorf 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 Obelix bereits besuchten Knoten in `visited`, und `ariadnesThread` speichert die Folge der Knoten, durch die der Ariadnefaden läuft (siehe erneut Abb. 3). Betritt Obelix einen neuen Knoten, dann fügt er den neuen Knoten ans Ende von `ariadnesThread` an. Wenn er entlang des Ariadnefadens zum vorhergehenden Knoten zurückkehren muss, entfernt er den Eintrag in `ariadnesThread`. Damit entspricht die aktuelle Position von Obelix 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 Eingangsknoten, der als Konstruktorparameter übergeben wurde. Da Obelix das Dorf aber noch nicht betreten hat, ist die Folge der Knoten in `ariadnesThread` vorerst leer, ebenso `visited`. Die Folge der Knoten, 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 Obelix 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 8 angegeben.

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

init() lässt Obelix das Dorf am Eingang betreten, d.h. man muss den Eingangsknoten am Ende von `ariadnesThread` ablegen und in `visited` eintragen.

foundMiraculix() liefert genau dann **true** zurück, wenn Obelix den Knoten erreicht hat, wo sich Miraculix befindet, sonst **false**.

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

retreatingStep() lässt Obelix 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 Miraculix* 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.BCSIterator, 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	Description
Constructor Stack()	Creates an empty Stack.

Method Summary

All Methods	Instance Methods	Concrete Methods	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

Methods declared in interface java.util.Collection

parallelStream, stream, toArray

Methods declared in interface java.util.List

addFirst, 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

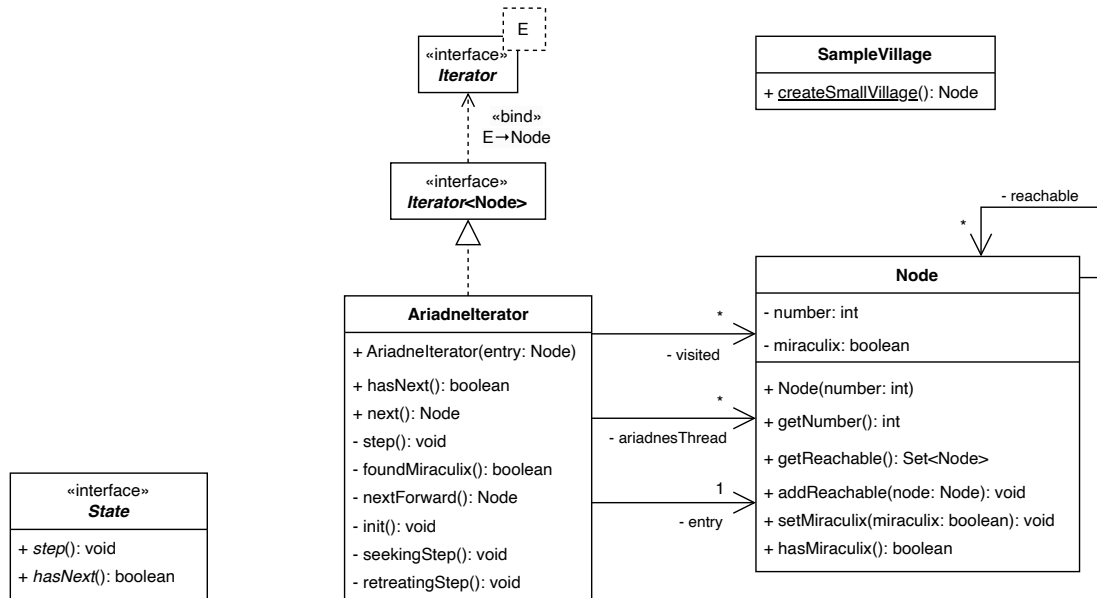


Abbildung 6: Zustandsdiagramm des Ariadne-Iterators.

