# Materialien zu den zentralen Abiturprüfungen im Fach Informatik

# Objektorientierter Ansatz

# Java

# 1. Allgemeine Hinweise

Es gelten für das Zentralabitur im Fach Informatik unverändert die Vorgaben (Stand Februar 2005) zu den inhaltlichen Schwerpunkten sowie die Angaben zu den Auswahlmöglichkeiten des Fachlehrers. Die folgenden Hinweise dienen dazu, Fehlinterpretationen der Vorgaben zu vermeiden, Fragen zu klären und dem Fachlehrer weitere fachliche Hinweise, Beispiele und Materialien zur unterrichtlichen Vorbereitung der Schülerinnen und Schüler auf die zentrale Abiturprüfung zu geben.

**Aufgabenvarianten**
Die Aufgaben zu fünf inhaltlichen Schwerpunkte werden in folgenden Varianten angeboten. Die beiden ersten Schwerpunkte werden in den Aufgaben in der Regel kombiniert vorkommen.

- Grundlegende Modellierungswerkzeuge und Modellierungstechniken
    - imperativer Ansatz, Programmiersprache Pascal
    - objektorientierter Ansatz, Programmiersprache Delphi
    - objektorientierter Ansatz, Programmiersprache Java

- Strukturen
    - imperativer Ansatz, Programmiersprache Pascal
    - objektorientierter Ansatz, Programmiersprache Delphi
    - objektorientierter Ansatz, Programmiersprache Java

- *Endliche Automaten und formale Sprachen*
    - Die Aufgaben werden paradigmenunabhängig formuliert. Aufgaben, die Programmcode enthalten, werden in den Varianten Pascal/Delphi und Java angeboten.

- *Stufen zwischen Hardware und Software*
    - Die Aufgaben werden paradigmenunabhängig formuliert und in den Sprachvarianten Pascal und Java angeboten.

- *Kommunikation zwischen Computern, Netze*
    - Da das Thema beim klassischen imperativen Ansatz entsprechend den „gelben Heften“ im Unterricht nicht vorgesehen ist, beziehen sich die Aufgaben dieses inhaltlichen Schwerpunktes ausschließlich auf den objektorientierten Ansatz in den beiden Sprachvarianten Delphi und Java.

**Zusätzliche Materialien für den Unterricht**
Schülerinnen und Schüler eines Informatik-Kurses der gymnasialen Oberstufe müssen zunächst nur einen Grundstock an Sprachelementen und Datentypen der eingeführten Programmiersprache beherrschen, um die im Unterricht entwickelten Algorithmen implementieren zu können. Bei der Behandlung komplexer Strukturen müssen beim imperativen Ansatz abstrakte Datentypen, beim objektorientierten Ansatz entsprechende Klassen entweder im Kurs entwickelt oder von der Lehrkraft vorgegeben werden. Einige Lehrerinnen und Lehrer nutzen die sehr umfangreichen und zum Teil unübersichtlichen spezifischen Klassenbibliotheken von Java oder Delphi, andere entwickeln gemeinsam mit den Kursen im Laufe der Qualifikationsphase nach Bedarf eigene, kleinere Klassenbibliothe-

ken oder abstrakte Datentypen. Die aus dem Unterricht bekannten Klassen oder ADT konnten beim dezentralen Abitur als bekannt vorausgesetzt und in der Abiturprüfung verwendet werden. Bei einer zentralen Prüfung kann auf die sehr unterschiedlichen Unterrichtsmaterialien nicht zurückgegriffen werden. Daher müssen die Aufgaben der zentralen Abiturprüfung als Anlage die Dokumentationen der benötigten Klassen und abstrakten Datentypen enthalten. An den Beispielaufgaben für Grund- und Leistungskurs wird deutlich, dass ein Schüler, bevor er ein vorgegebenes Programmstück analysieren oder einen Algorithmus in einer Programmiersprache implementieren kann, die für ihn neuen Klassendokumentationen gründlich analysieren muss. Dadurch entstehen erhöhte inhaltliche und zeitliche Anforderungen, die in diesem Umfang in der dezentralen Abiturprüfung nicht vorhanden waren. Um diese zusätzliche Schwierigkeit zu reduzieren, enthält diese Schrift, die auch vom Bildungsserver learn-line heruntergeladen werden kann, eine Zusammenstellung der Dokumentationen von Klassen bzw. abstrakten Datentypen, die als Materialvorgaben Bestandteil der Aufgaben der zentralen Abiturprüfungen 2007 und 2008 sein können. Es empfiehlt sich, diese Materialien in der Jahrgangsstufe 13 insbesondere in Wiederholungsphasen unterrichtlich zu nutzen. Es sei noch einmal ausdrücklich darauf hingewiesen, dass diese Materialien kein Bestandteil der inhaltlichen Vorgaben für die zentrale Abiturprüfung im Fach Informatik sind. Alle Aufgaben können auch ohne vorherige Kenntnis dieser spezifischen Dokumentationen bearbeitet werden.
Neben den Dokumentationen der Datenstrukturen enthält diese Schrift eine Zusammenstellung der bei der zentralen Abiturprüfung vorausgesetzten Basis-Sprachelemente und Standard-Datentypen der Programmiersprache Java sowie einige Anwendungsbeispiele (keine Aufgaben) für Klassen, die in keiner der vorliegenden Beispielaufgaben für die zentrale Abiturprüfung vorkommen.

## 2. Basis-Sprachelemente und -Datentypen

Kenntnisse über Java-spezifische Klassen auch zur Gestaltung einer grafischen Benutzeroberfläche werden bei den Abituraufgaben nicht vorausgesetzt.

**Sprachelemente**

- Klassendefinitionen
- Ist-, Hat- und Kennt-Beziehungen zwischen Klassen
- Attribute und Methoden (mit Parametern und Rückgabewerten)
- Wertzuweisung
- Verzweigungen
- Schleifen (do - while, while -, for -)

**Datentypen**

| **Datentyp** | **Operationen** | **Methoden** |
|---|---|---|
| int | +, -, *, /, %, <, >, <=, >=,==,!= | Integer.toString() |
| double | +, -, *, /, <, >, <=, >=,==,!= | Math.round(), Double.toString() |
| boolean | &&, \|\|, !, ==, !=, | |
| char | <, >, <=, >=,==,!= | |
| Klasse String | | length(),indexOf(), substring(), charAt(), equals(), compareTo(), Integer.parseInt(), Double.parseDouble() |
| array: bis 2-Dimensional | | |

## 3. Lineare Strukturen

### Die Klasse Queue

Objekte der Klasse *Queue* (Schlange) verwalten beliebige Objekte nach dem First-In-First-Out-Prinzip, d.h., dass das zuerst abgelegte Element als erstes wieder entnommen wird.

Die Klasse *Queue* stellt Methoden in folgender Syntax zur Verfügung:

```
public Queue()

public  boolean isEmpty()

public  void enqueue (Object pObject)

public  void dequeue ()

public  Object front()
```

## Dokumentation der Methoden der Klasse Queue

| | |
|---|---|
| **Konstruktor** | **`Queue()`** |
| *Nachher* | Eine leere Schlange ist erzeugt. |
| **Anfrage** | **`isEmpty()`** |
| *Nachher* | Die Anfrage liefert den Wert ***true***, wenn die Schlange keine Elemente enthält, sonst liefert sie den Wert ***false***. |
| **Auftrag** | **`enqueu (Object pObject)`** |
| *Vorher* | Die Schlange ist erzeugt. |
| *Nachher* | *pObject* ist als letztes Element in der Schlange abgelegt. |
| **Auftrag** | **`dequeu()`** |
| *Vorher* | Die Schlange ist nicht leer. |
| *Nachher* | Das vorderste Element ist aus der Schlange entfernt. |
| **Anfrage** | **`front(): Object`** |
| *Vorher* | Die Schlange ist nicht leer. |
| *Nachher* | Die Anfrage liefert das vorderste Element der Schlange. Die Schlange ist unverändert. |

**Die Klasse Stack**

Objekte der Klasse *Stack* (Keller, Stapel*)* verwalten beliebige Objekte nach dem Last-In-First-Out-Prinzip, d.h., dass das zuletzt abgelegte Element als erstes wieder entnommen wird.

Die Klasse *Stack* stellt Methoden in folgender Syntax zur Verfügung:

```
public Stack()

public boolean isEmpty()

public void push (Object pObject)

public void pop ()

public Object top ()
```

## Dokumentation der Methoden der Klasse Stack

| | |
|---|---|
| **Konstruktor** | **Stack()** |
| *Nachher* | Ein leerer Stapel ist erzeugt. |
| **Anfrage** | **isEmpty(): boolean** |
| *Nachher* | Die Anfrage liefert den Wert ***true***, wenn der Stapel keine Elemente enthält, sonst liefert sie den Wert ***false***. |
| **Auftrag** | **push (Object pObject)** |
| *Vorher* | Der Stapel ist erzeugt. |
| *Nachher* | *pObject* liegt oben auf dem Stack. |
| **Auftrag** | **pop()** |
| *Vorher* | Der Stapel ist nicht leer. |
| *Nachher* | Das zuletzt eingefügte Element ist aus dem Stapel entfernt. |
| **Anfrage** | **top():Object** |
| *Vorher* | Der Stapel ist nicht leer. |
| *Nachher* | Die Anfrage liefert das oberste Stapelelement. Der Stapel ist unverändert. |

**Die Klasse List**

Objekte der Klasse *List* verwalten beliebige Objekte nach einem Listenprinzip. Ein interner Positionszeiger wird durch die Listenstruktur bewegt, seine Position markiert ein aktuelles Objekt. Die Lage des Positionszeigers kann abgefragt, verändert und die Objektinhalte an den Positionen können gelesen oder verändert werden.

Die Klasse *List* stellt Methoden in folgender Syntax zur Verfügung:

```
public List()

public boolean isEmpty()

public boolean isBefore()

public boolean isBehind()

public void next()

public void previous()

public void toFirst()

public void toLast()

public Object getItem()

public void update (Object pObject)

public void insertBefore (Object pObject)

public void insertBehind (Object pObject)

public void delete()
```

## Dokumentation der Methoden der Klasse List

| | |
|---|---|
| **Konstruktor** | **`List()`** |
| *Nachher* | Eine leere Liste ist angelegt. Der interne Positionszeiger steht vor der leeren Liste |
| **Anfrage** | **`isEmpty(): boolean`** |
| *Nachher* | Die Anfrage liefert den Wert ***true***, wenn die Liste keine Elemente enthält, sonst liefert sie den Wert ***false***. |
| **Anfrage** | **`isBefore(): boolean`** |
| *Nachher* | Die Anfrage liefert den Wert ***true***, wenn der Positionszeiger vor dem ersten Listenelement oder vor der leeren Liste steht, sonst liefert sie den Wert ***false***. |
| **Anfrage** | **`isBehind(): boolean`** |
| *Nachher* | Die Anfrage liefert den Wert ***true***, wenn der Positionszeiger hinter dem letzten Listenelement oder hinter der leeren Liste steht, sonst liefert sie den Wert ***false***. |
| **Auftrag** | **`next()`** |
| *Nachher* | Der Positionszeiger ist um eine Position in Richtung Listenende weitergerückt, d.h. wenn er vor der Liste stand, wird das Element am Listenanfang zum aktuellen Element, ansonsten das jeweils nachfolgende Listenelement. Stand der Positionszeiger auf dem letzten Listenelement, befindet er sich jetzt hinter der Liste. Befand er sich hinter der Liste, hat er sich nicht verändert. |
| **Auftrag** | **`previous()`** |
| *Nachher* | Der Positionszeiger ist um eine Position in Richtung Listenanfang weitergerückt, d.h. wenn er hinter der Liste stand, wird das Element am Listenende zum aktuellen Element, ansonsten das jeweils vorhergehende Listenelement. Stand der Positionszeiger auf dem ersten Listenelement, befindet er sich jetzt vor der Liste. Befand er sich vor der Liste, hat er sich nicht verändert. |
| **Auftrag** | **`toFirst()`** |
| *Nachher* | Der Positionszeiger steht auf dem ersten Listenelement. Falls die Liste leer ist befindet er sich jetzt hinter der Liste. |
| **Auftrag** | **`toLast()`** |
| *Nachher* | Der Positionszeiger steht auf dem letzten Listenelement. Falls die Liste leer ist befindet er sich jetzt vor der Liste. |
| **Anfrage** | **`getItem(): Object`** |
| *Nachher* | Die Anfrage liefert den Wert des aktuellen Listenelements bzw. *null*, wenn die Liste keine Elemente enthält, bzw. der Positionszeiger vor oder hinter der Liste steht. |
| **Auftrag** | **`update (Object pObject)`** |
| *Vorher* | Die Liste ist nicht leer. Der Positionszeiger steht nicht vor oder hinter der Liste. |
| *Nachher* | Der Wert des Listenelements an der aktuellen Position ist durch *pObject* ersetzt. |

| | |
|---|---|
| **Auftrag** | **`insertBefore (Object pObject)`** |
| *Vorher* | Der Positionszeiger steht nicht vor der Liste. |
| *Nachher* | Ein neues Listenelement mit dem entsprechenden Objekt ist angelegt und vor der aktuellen Position in die Liste eingefügt worden. Der Positionszeiger steht hinter dem eingefügten Element. |

| | |
|---|---|
| **Auftrag** | **`insertBehind (Object pObject)`** |
| *Vorher* | Der Positionszeiger steht nicht hinter der Liste. |
| *Nachher* | Ein neues Listenelement mit dem entsprechenden Objekt ist angelegt und hinter der aktuellen Position in die Liste eingefügt worden. Der Positionszeiger steht vor dem eingefügten Element. |

| | |
|---|---|
| **Auftrag** | **`delete()`** |
| *Vorher* | Der Positionszeiger steht nicht vor oder hinter der Liste. |
| *Nachher* | Das aktuelle Listenelement ist gelöscht. Der Positionszeiger steht auf dem Element hinter dem gelöschten Element, bzw. hinter der Liste, wenn das gelöschte Element das letzte Listenelement war. |

# 4. Baumstrukturen

### Die Klasse BinTree

In einem Objekt der Klasse *BinTree* werden beliebige Objekte nach dem Binärbaumprinzip verwaltet. Ein Binärbaum ist entweder leer oder besteht aus einem Knoten, dem ein Element und zwei binäre Teilbäume, die so genannten linken und rechten Teilbäume, zugeordnet sind.

Die Klasse *BinTree* stellt Methoden in folgender Syntax zur Verfügung:

```
public BinTree()

public BinTree (Object pObject)

public BinTree (Object pObject, BinTree pLeftTree, BinTree
                                                    pRightTree)

public boolean isEmpty()

public void clear()

public void setRootItem (Object pObject)

public Object getRootItem()

public void addTreeLeft (BinTree pTree)

public void addTreeRight (BinTree pTree)

public BinTree getLeftTree()

public BinTree getRightTree()
```

## Dokumentation der Methoden der Klasse BinTree

| | |
|---|---|
| **Konstruktor** | **`BinTree()`** |
| *nachher* | Ein leerer Baum existiert |
| **Konstruktor** | **`BinTree (Object pObject)`** |
| *nachher* | Der Binärbaum existiert und hat einen Wurzelknoten mit dem Inhalt *pObject* und zwei leeren Teilbäumen. |
| **Konstruktor** | **`BinTree (Object pObject, BinTree pLeftTree, BinTree pRightTree)`** |
| *nachher* | Der Binärbaum existiert hat einen Wurzelknoten mit dem Inhalt *pObject,* dem linken Teilbaum pLeftTree und dem rechten Teilbaum pRightTree. |
| **Anfrage** | **`isEmpty(): boolean`** |
| *nachher* | Diese Anfrage liefert den Wahrheitswert ***true***, wenn der Binärbaum leer ist, sonst liefert sie den Wert ***false***. |
| **Auftrag** | **`clear()`** |
| *nachher* | Der Binärbaum ist leer. |
| **Auftrag** | **`setRootItem (Object pObject)`** |
| *nachher* | Die Wurzel hat – unabhängig davon, ob der Binärbaum leer ist oder schon eine Wurzel hat – *pObject* als Inhalt. Eventuell vorhandene Teilbäume werden nicht geändert. |
| **Anfrage** | **`getRootItem(): Object`** |
| *vorher* | Der Binärbaum ist nicht leer. |
| *nachher* | Diese Anfrage liefert den Inhalt des Wurzelknotens des Binärbaums. |
| **Auftrag** | **`addTreeLeft (BinTree pTree)`** |
| *vorher* | Der Binärbaum ist nicht leer. |
| *nachher* | Die Wurzel hat den übergebenen Baum als linken Teilbaum. |
| **Auftrag** | **`addTreeRight (BinTree pTree)`** |
| *vorher* | Der Binärbaum ist nicht leer. |
| *nachher* | Die Wurzel hat den übergebenen Baum als rechten Teilbaum. |
| **Anfrage** | **`getLeftTree(): BinTree`** |
| *vorher* | Der Binärbaum ist nicht leer |
| *nachher* | Diese Anfrage liefert den linken Teilbaum der Wurzel des Binärbaums, bzw. *null*, wenn der Teilbaum nicht vorhanden ist. Der Binärbaum ist unverändert. |
| **Anfrage** | **`getRightTree(): BinTree`** |
| *vorher* | Der Binärbaum ist nicht leer |
| *nachher* | Diese Anfrage liefert den rechten Teilbaum der Wurzel des Binärbaums, bzw. *null*, wenn der Teilbaum nicht vorhanden ist. Der Binärbaum ist unverändert. |

**Die Klasse Item**

Die Klasse *Item* ist Oberklasse aller Klassen, deren Objekte in einen geordneten Baum eingefügt werden sollen. Die Ordnungsrelation wird in den Unterklassen von *Item* durch Überschreiben der drei abstrakten Methoden *isEqual*, *isGreater* und *isLower* festgelegt.

Die Klasse *item* stellt Methoden in folgender Syntax zur Verfügung:

```
public abstract boolean isEqual(Item pItem)

public abstract boolean isLower(Item pItem)

public abstract boolean isGreater(Item pItem)
```

**Die Klasse OrderedTree**

Die Klasse *OrderedTree* verwaltet Objekte in einem geordneten Binärbaum, für den gilt, dass alle Knoteninhalte im linken Unterbaum kleiner sind als der Wurzelinhalt und alle Knoteninhalte im rechten Teilbaum größer sind als der Wurzelinhalt. Diese Bedingung gilt auch in allen Unterbäumen. Die Knoteninhalte sind Objekte einer Unterklasse von *Item*, in der durch Überschreiben der drei Vergleichsmethoden *isLower, isEqual, isGreater* (s. *Item*) eine eindeutige Ordnungsrelation festgelegt werden muss.

Die Klasse *OrderedTree* stellt Methoden in folgender Syntax zur Verfügung:

```
public OrderedTree()

public void insertItem(Item pItem)

public Item searchItem(Item pItem)

public boolean isEmpty()

public void deleteItem(Item pItem)

public List getSortedList()
```

## Dokumentation der Methoden Klasse OrderedTree

**Konstruktor**
*nachher*

**`OrderedTree()`**
Der geordnete Baum existiert und ist leer.

**Anfrage**
*nachher*

**`isEmpty(): boolean`**
Diese Anfrage liefert den Wahrheitswert ***true***, wenn der geordnete Baum leer ist, sonst liefert sie den Wert ***false***.

**Auftrag**
*nachher*

**`insertItem(Item pItem)`**
*pItem* ist entsprechend der Ordnungsrelation in den Baum eingeordnet.

**Anfrage**
*nachher*

**`searchItem (Item pItem): Item`**
Falls ein bezüglich der verwendeten Ordnungsrelation mit *pItem* übereinstimmendes Objekt im geordneten Baum enthalten ist, liefert die Anfrage dieses, ansonsten wird *null* zurückgegeben.

**Auftrag**
*nachher*

**`deleteItem(Item pItem)`**
Falls ein bezüglich der verwendeten Ordnungsrelation mit *pItem* übereinstimmendes Objekt im Baum enthalten war, wurde dieses entfernt.

**Anfrage**

**`getSortedList (): List`**
Die Knoteninhalte des geordneten Binärbaums werden als sortierte Liste zurückgegeben. Ist der geordnete Baum leer, wird *null* zurückgegeben.

# 5. Graphen

### Die Klasse GraphNode

Objekte der Klasse *GraphNode* sind Knoten eines Graphen. Ein Graphknoten kann durch seinen Namen eindeutig identifiziert werden. Darüber hinaus enthält er eine Liste der Kanten zu seinen Nachbarknoten sowie eine boolesche Variable, die den Wert ***true*** hat, wenn der Knoten markiert ist.

Die Klasse *GraphNode* stellt Methoden in folgender Syntax zur Verfügung:

```
public GraphNode (String pName)
public void insertEdge (Edge pEdge)
public void mark()
public void unmark()
public boolean isMarked()
public List edges()
public String name()
```

## Dokumentation der Methode der Klasse GraphNode

| | |
|---|---|
| **Konstruktor** | **`GraphNode (String pName)`** |
| *nachher* | Ein Knoten mit dem Namen *pName* wurde erzeugt. Es gibt keine Kanten, die diesen Knoten mit anderen Knoten verbinden. Der Knoten ist nicht markiert. |
| **Auftrag** | **`insertEdge (Edge pEdge)`** |
| *nachher* | Der Knoten hat die Kante *pEdge* zu einem Nachbarknoten. |
| **Auftrag** | **`mark()`** |
| *nachher* | Der Knoten ist markiert. |
| **Auftrag** | **`unmark()`** |
| *nachher* | Der Knoten ist nicht markiert. |
| **Anfrage** | **`isMarked(): boolean`** |
| *nachher* | Die Anfrage liefert ***true***, wenn der Knoten markiert ist, sonst liefert sie ***false***. |
| **Anfrage** | **`edges(): List`** |
| *nachher* | Die Anfrage liefert die Liste der Kanten des Knotens. Die Inhalte der Listenknoten sind von der Klasse *Edge*. Hat der Knoten keine Kanten, wird *null* zurückgeliefert. |
| **Anfrage** | **`name(): String`** |
| *nachher* | Die Anfrage liefert den Namen des Knotens. |

**Die Klasse Edge**

Objekte der Klasse *Edge* sind Kanten eines Graphen. Eine Kante hat ein Gewicht und enthält einen Verweis auf den zu ihr gehörenden Nachbarknoten des Knotens, zu dessen Kantenliste sie gehört.

Die Klasse *Edge* stellt Methoden in folgender Syntax zur Verfügung:

```
public Edge (GraphNode pNeighbour, double pWeight)
public GraphNode neighbour()
public double weight()
```

## Dokumentation der Methoden der Klasse Edge

**Konstruktor**
**`Edge (GraphNode pNeighbour, double pWeight)`**
*nachher*
Eine Kante mit dem Nachbarknoten *pNeighbour* und dem Gewicht *pWeight* wurde erzeugt.

**Anfrage**
**`neighbour(): GraphNode`**
*nachher*
Die Anfrage liefert den Nachbarknoten des Knotens, zu dem die Kante gehört.

**Anfrage**
**`weight(): double`**
*nachher*
Die Anfrage liefert das Gewicht der Kante.

### Die Klasse Graph

Objekte der Klasse *Graph* sind ungerichtete, gewichtete Graphen. Sie enthalten Objekte der Klasse *GraphNode*, die durch Objekte der Klasse *Edge* verbunden sind.

Die Klasse *Graph* stellt Methoden in folgender Syntax zur Verfügung:

```
public Graph()
public void insertNode (String pName)
public void connectNodes (String pName1, String pName2, double
                                                          pWeight)
public GraphNode searchNode (pName: string)
public boolean allMarked()
public boolean hasEdge (GraphNode pNode1, GraphNode pNode2)
```

## Dokumentation der Methoden der Klasse Graph

**Konstruktor** **`Graph()`**
Ein neuer Graph wurde erzeugt. Er enthält keine Knoten.

**Auftrag** **`insertNode (String pName)`**
*vorher* Der Graph hat keinen Knoten mit dem Namen *pName*.
*nachher* Im Graph existiert ein Knoten mit dem Namen *pName.*

**Auftrag** **`connectNodes (String pName1, String pName2, double pWeight)`**
*vorher* Der Graph enthält Knoten mit den Namen *pName1* und *pName2* und es gibt keine Kante zwischen diesen beiden Knoten.
*nachher* Die Knoten mit Namen *pName1* und *pName2* sind durch eine Kante verbunden, die das Gewicht *pWeight* hat. *pName1* ist also Nachbarknoten von *pName2* und umgekehrt.

**Anfrage** **`searchNode (String pName): GraphNode`**
*nachher* Die Anfrage liefert den Knoten mit dem Namen *pName*, wenn er im Graphen gefunden wurde, andernfalls wird *null* zurückgegeben.

**Anfrage** **`allMarked(): boolean`**
*nachher* Die Anfrage liefert ***true***, wenn alle Knoten des Graphen markiert sind, sonst liefert sie den Wert ***false***.

**Anfrage** **`hasEdge (GraphNode pNode1, GraphNode pNode2): boolean`**
*nachher* Die Anfrage liefert ***true***, wenn die Konten pNode1 und pNode2 im Graphen enthalten sind und zwischen ihnen eine Kante existiert, sonst liefert sie ***false***.

# 6. Client-Server

## Die Klasse Connection

Objekte der Klasse *Connection* ermöglichen eine Netzwerkverbindung mit dem TCP-Protokoll. Es können nach Verbindungsaufbau zu einem Server Zeichenketten (Strings) gesendet und empfangen werden. Zur Vereinfachung geschieht dies zeilenweise, d.h. beim Senden einer Zeichenkette wird ein Zeilentrenner ergänzt und beim Empfangen wird er entfernt.
Eine Fehlerbehandlung ist in dieser Klasse aus Gründen der Vereinfachung nicht vorgesehen.

Die Klasse *Connection* stellt Methoden in folgender Syntax zur Verfügung:

```
public Connection (String pServerIP, int pServerPort)

public void send (String pMessage)

public String receive()

public void close()
```

## Dokumentation der Methoden der Klasse Connection

**Konstruktor**
*nachher*

**`Connection (String pServerIP, int pServerPort)`**
Es ist eine Verbindung zum durch IP-Adresse und Portnummer angegebenen Server eingerichtet, so dass Daten gesendet und empfangen werden können.
Anstelle einer IP-Adresse kann *pServerIP* auch ein Rechnername sein.

**Auftrag**
*nachher*

**`send (String pMessage)`**
Die angegebene Nachricht wurde, um einen Zeilentrenner erweitert, an den Server versandt.

**Anfrage**
*nachher*

**`receive():String`**
Es wurde auf eine eingehende Nachricht vom Server gewartet und diese Nachricht zurückgegeben, wobei der vom Server angehängte Zeilentrenner entfernt wurde. Während des Wartens war der ausführende Prozess blockiert.

**Auftrag**

**`close()`**
Die Verbindung wurde getrennt und kann nicht mehr verwendet werden.

**Die Klasse Client**

Über die Klasse *Client* werden Netzwerkverbindungen mit dem TCP-Protokoll ermöglicht. Es können nach Verbindungsaufbau zu einem Server Zeichenketten (Strings) gesendet und empfangen werden, wobei der Empfang nebenläufig geschieht. Zur Vereinfachung geschieht dies zeilenweise, d.h. beim Senden einer Zeichenkette wird ein Zeilentrenner ergänzt und beim Empfangen wird er entfernt.
Die empfangene Nachricht wird durch eine Ereignisbehandlungsmethode verarbeitet, die in Unterklassen überschrieben werden muss. Eine Fehlerbehandlung ist in dieser Klasse aus Gründen der Vereinfachung nicht vorgesehen.

Die Klasse *Client* stellt Methoden in folgender Syntax zur Verfügung:

```
public Client (String pServerIP, int pServerPort)

public void send (String pMessage)

public void processMessage (String pMesssage)

public void close()
```

## Dokumentation der Methoden der Klasse Client

**Konstruktor** **`Client (String pServerIP, int pServerPort)`**

*nachher* Es ist eine Verbindung zum durch IP-Adresse und Portnummer angegebenen Server eingerichtet, so dass Zeichenketten gesendet und empfangen werden können.
Anstelle einer IP-Adresse kann *pServerIP* auch ein Rechnername sein.

**Auftrag** **`send (String pMessage)`**

*nachher* Die angegebene Nachricht wurde, um einen Zeilentrenner erweitert, an den Server versandt.

**Auftrag** **`processMessage (String pMesssage)`**

*vorher* Der Server hat die angegebene Nachricht gesendet. Der Zeilentrenner wurde entfernt.

*nachher* Der Client hat auf die Nachricht reagiert. Diese Methode enthält keine Anweisungen und muss in Unterklassen überschrieben werden, damit die Nachricht verarbeitet werden kann.

**Auftrag** **`close()`**

Die Verbindung wurde getrennt und kann nicht mehr verwendet werden.

**Die Klasse Server**

Über die Klasse Server ist es möglich, eigene Serverdienste anzubieten, so dass Clients Verbindungen gemäß dem TCP-Protokoll hierzu aufbauen können. Nachrichten werden grundsätzlich zeilenweise verarbeitet. Verbindungsaufbau, Nachrichtenempfang und Verbindungsende geschehen nebenläufig. Durch Überschreiben der entsprechenden Methoden kann der Server auf diese Ereignisse reagieren. Eine Fehlerbehandlung ist in dieser Klasse aus Gründen der Vereinfachung nicht vorgesehen.

Die Klasse *Server* stellt Methoden in folgender Syntax zur Verfügung:

```
public Server (int pPortNr)

public void closeConnection (String pClientIP, int pClientPort)

public void processClosedConnection (String pClientIP,
                                     int pClientPort)

public void processMessage (String pClientIP, int pClientPort,
                            String pMessage)

public void processNewConnection (String pClientIP,
                                  int pClientPort)

public void send (String pClientIP, int pClientPort,
                  String pMessage)

public void sendToAll (String pMessage)

public void close()
```

## Dokumentation der Methoden der Klasse Server

**Konstruktor** **`Server (int pPortNr)`**
*nachher* Der Server bietet seinen Dienst über die angegebene Portnummer an. Clients können sich nun mit dem Server verbinden.

**Auftrag** **`closeConnection (String pClientIP, int pClientPort)`**
*vorher* Eine Verbindung mit dem angegebenen Client besteht.
*nachher* Die Verbindung besteht nicht mehr. Der Server hat sich die Nachricht *processClosedConnection* gesendet.

**Auftrag** **`processClosedConnection (String pClientIP, int pClientPort)`**
*vorher* Die Verbindung zu dem angegebenen Client wurde beendet.
*nachher* Der Server hat auf das Beenden der Verbindung reagiert. Diese Methode enthält keine Anweisungen und kann in Unterklassen überschrieben werden.

**Auftrag** **`processMessage (String pClientIP, int pClientPort, String pMessage)`**
*vorher* Der Client mit der angegebenen IP und der angegebenen Portnummer hat dem Server eine Nachricht gesendet.
*nachher* Der Server hat auf die Nachricht reagiert. Diese Methode enthält keine Anweisungen und kann in Unterklassen überschrieben werden.

**Auftrag** **`processNewConnection (String pClientIP, int pClientPort)`**
*vorher* Der Client mit der angegebenen IP-Adresse und der angegebenen Portnummer hat eine Verbindung zum Server aufgebaut.
*nachher* Der Server hat auf den Verbindungsaufbau reagiert. Diese Methode enthält keine Anweisungen und kann in Unterklassen überschrieben werden.

**Auftrag** **`send(String pClientIP, int pClientPort, String pMessage)`**
*vorher* Eine Verbindung zu dem angegebenen Client besteht.
*nachher* Die angegebene Nachricht wurde dem Client, um einen Zeilentrenner erweitert, gesendet.

**Auftrag** **`sendToAll (String pMessage)`**
*nachher* Die angegebene Nachricht wurde an alle verbundenen Clients gesendet.

**Auftrag** **`close()`**
Alle Verbindungen wurden getrennt. Der Server kann nicht mehr verwendet werden.

# 7. Beispiel für die Anwendung der Klasse BinTree in Java

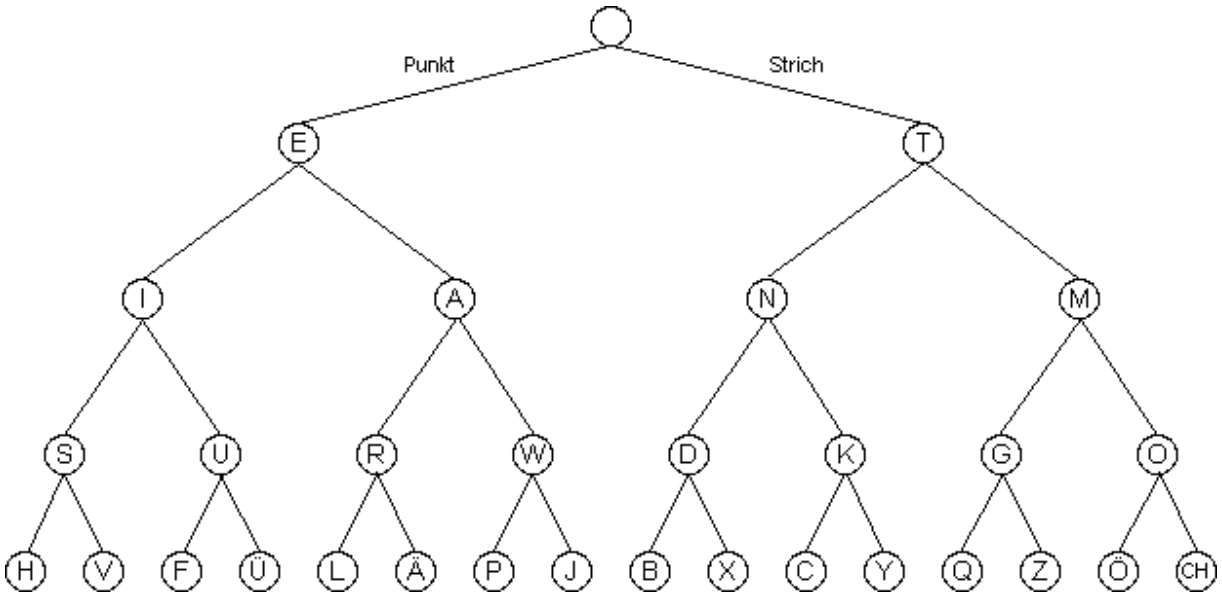
## a) Aufgabenstellung

Es soll ein Programm entwickelt werden, das in der Lage ist, eine Zeichenfolge aus Großbuchstaben in das Morsealphabet zu codieren und eine Folge von Morsezeichen, die durch „/“ getrennt sind, zu dekodieren.

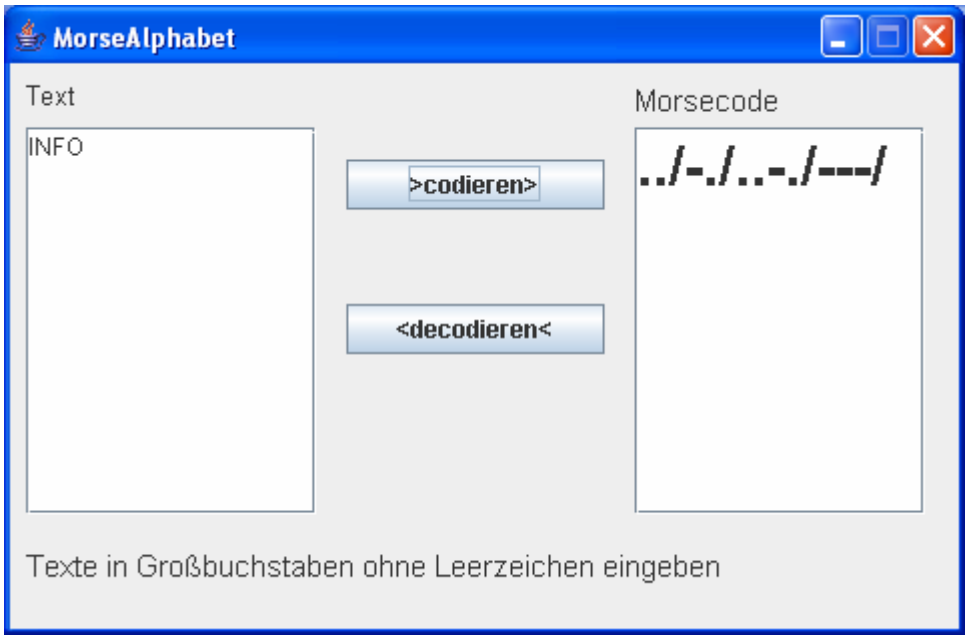
**Das Morsealphabet**

| | | | | | |
|---|---|---|---|---|---|
| A ·- | F ···· | K -·- | P ·--· | U ··- | Z --·· |
| B -··· | G --· | L ·-·· | Q --·- | V ···- | Ä ·-·- |
| C -·-· | H ···· | M -- | R ·-· | W ·-- | Ö ---· |
| D -·· | I ·· | N -· | S ... | X -··- | Ü ··-- |
| E · | J ·--- | O --- | T - | Y -·-- | CH ---- |

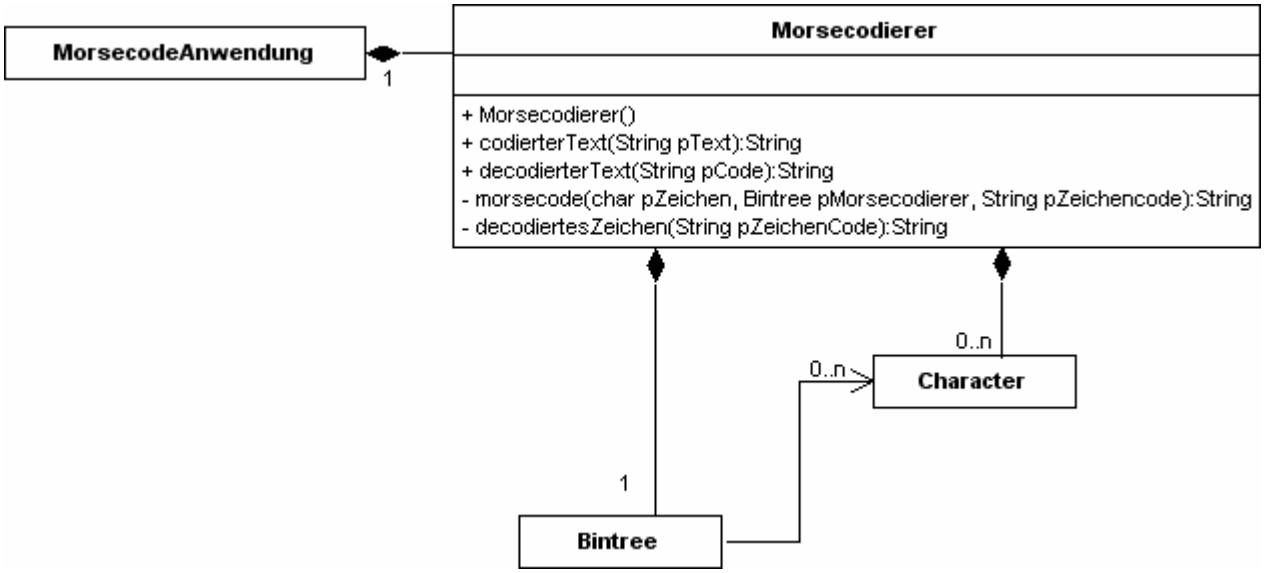
Da der Morsecode nur aus zwei Zeichen besteht, lässt er sich in einem Binärbaum darstellen:



## b) Benutzerschnittstelle




## c) UML-Diagramm (vereinfacht)



### d) Implementation einiger Methoden

Der Konstruktur ***Morsecodierer*** der Klasse ***Morsecodierer*** erzeugt den Morsebaum. Um den ASCII-Code verwenden zu können, wurde auf die Umlaute und das Zeichen „CH“ verzichtet. Man beachte die Verwendung verschiedener Konstruktoren der Klasse *BinTree.*

```
public MorseCodierer(){
  BinTree lBaum4Links = new BinTree(new Character('H'));
  BinTree lBaum4Rechts = new BinTree(new Character('V'));
  BinTree lBaum3Links = new BinTree(new Character('S'),lBaum4Links, lBaum4Rechts);
  lBaum4Links = new BinTree(new Character('F'));
  lBaum4Rechts = new BinTree(new Character('#'));
  BinTree lBaum3Rechts = new BinTree(new Character('U'), lBaum4Links, lBaum4Rechts);
  BinTree lBaum2Links = new BinTree(new Character('I'), lBaum3Links, lBaum3Rechts);
  lBaum4Links = new BinTree(new Character('L'));
  lBaum4Rechts = new BinTree(new Character('#'));
  lBaum3Links = new BinTree(new Character('R'), lBaum4Links, lBaum4Rechts);
  lBaum4Links = new BinTree(new Character('P'));
  lBaum4Rechts = new BinTree(new Character('J'));
  lBaum3Rechts = new BinTree(new Character('W'), lBaum4Links, lBaum4Rechts);
  BinTree lBaum2Rechts = new BinTree(new Character('A'), lBaum3Links, lBaum3Rechts);
  BinTree lBaum1Links = new BinTree(new Character('E'), lBaum2Links, lBaum2Rechts);
  lBaum4Links = new BinTree(new Character('B'));
  lBaum4Rechts = new BinTree(new Character('X'));
  lBaum3Links = new BinTree(new Character('D'), lBaum4Links, lBaum4Rechts);
  lBaum4Links = new BinTree(new Character('C'));
  lBaum4Rechts = new BinTree(new Character('Y'));
  lBaum3Rechts = new BinTree(new Character('K'), lBaum4Links, lBaum4Rechts);
  lBaum2Links = new BinTree(new Character('N'), lBaum3Links, lBaum3Rechts);
  lBaum4Links = new BinTree(new Character('Q'));
  lBaum4Rechts = new BinTree(new Character('Z'));
  lBaum3Links = new BinTree(new Character('G'), lBaum4Links, lBaum4Rechts);
  lBaum4Links = new BinTree(new Character('#'));
  lBaum4Rechts = new BinTree(new Character('#'));
  lBaum3Rechts = new BinTree(new Character('O'),lBaum4Links, lBaum4Rechts);
  lBaum2Rechts = new BinTree(new Character('M'), lBaum3Links, lBaum3Rechts);
  BinTree lBaum1Rechts = new BinTree(new Character('T'), lBaum2Links, lBaum2Rechts);
  morsebaum = new BinTree(new Character('#'), lBaum1Links, lBaum1Rechts);
}
```

Die Methode ***codierterText*** liefert den in den Morsecode übersetzten Text ***pText***. Die von ihr aufgerufene private Methode ***morsecode*** sucht den Morsecode eines einzelnen Zeichens im Morsebaum.

```
public String codierterText (String pText) {
  int lZaehler=0;
  String lCode="";
  while ((lZaehler<pText.length()) && (pText.charAt(lZaehler)>='A') &&
          (pText.charAt(lZaehler)<='Z')){
    lCode = lCode + this.morsecode(pText.charAt(lZaehler),morsebaum,"");
    lCode = lCode + "/";
    lZaehler=lZaehler+1;
  }
  return lCode;
}

public String morsecode(char pZeichen, BinTree pMorsebaum, String pZeichencode){
  String lErgebnisLinkerTeilbaum="", lErgebnis="";
  Character charObjekt;
  if (pMorsebaum.isEmpty())
    return "";
  else {
    if (((Character)pMorsebaum.getRootItem()).charValue()==pZeichen)
      return pZeichencode;
    else {
      lErgebnisLinkerTeilbaum=this.morsecode(pZeichen,pMorsebaum.getLeftTree(),
                                             pZeichencode+".");
      if (lErgebnisLinkerTeilbaum.length()==0)
       return this.morsecode(pZeichen,pMorsebaum.getRightTree(),pZeichencode+"-");
      else
        return lErgebnisLinkerTeilbaum;
    }
  }
}
```

Die Methode ***decodierterText*** liefert die Decodierung des Morsecodes ***pCode***. Die von ihr aufgerufene private Methode ***decodierteZeichen*** decodiert ein einzelnes Zeichen mit Hilfe des Morsebaums.

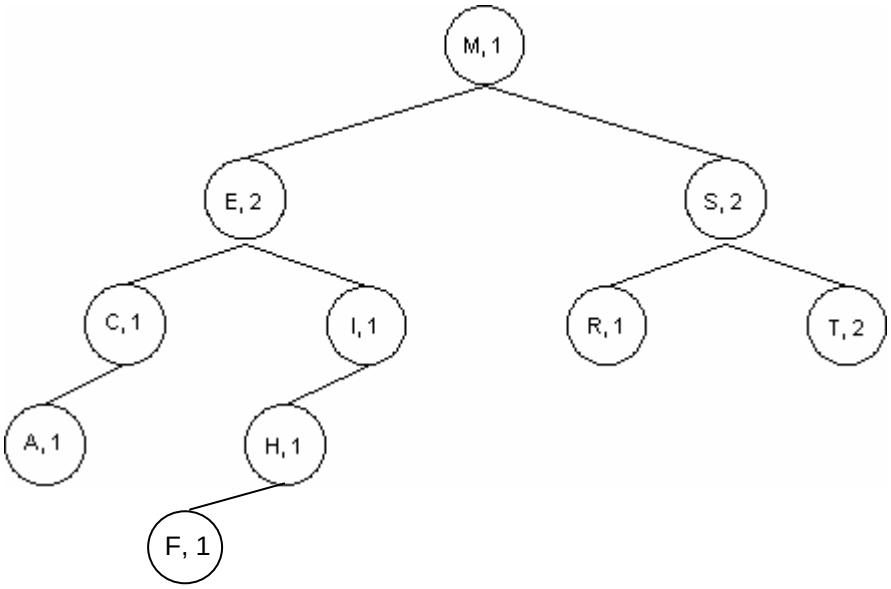
```
public String decodierterText (String pCode) {
  int lZaehler=1;
  String lText="";
  String lMorsezeichen;
  do {
    lMorsezeichen=pCode.substring(0,pCode.indexOf('/'));
    pCode=pCode.substring(pCode.indexOf('/')+1,pCode.length());
    if (!lMorsezeichen.equals("")) {
      lText=lText+this.decodiertesZeichen(lMorsezeichen);
    }
  } while (!pCode.equals(""));
  return lText;
}

public Character decodiertesZeichen( String pZeichenCode){
  String lText="";
  int lZaehler=0;
  BinTree lBaum=morsebaum;
  while (lZaehler<pZeichenCode.length()) {
    if (pZeichenCode.charAt(lZaehler)=='.')
      lBaum=lBaum.getLeftTree();
    else
      lBaum=lBaum.getRightTree();
    lZaehler+=1;
  }
  return (Character)lBaum.getRootItem();
}
```

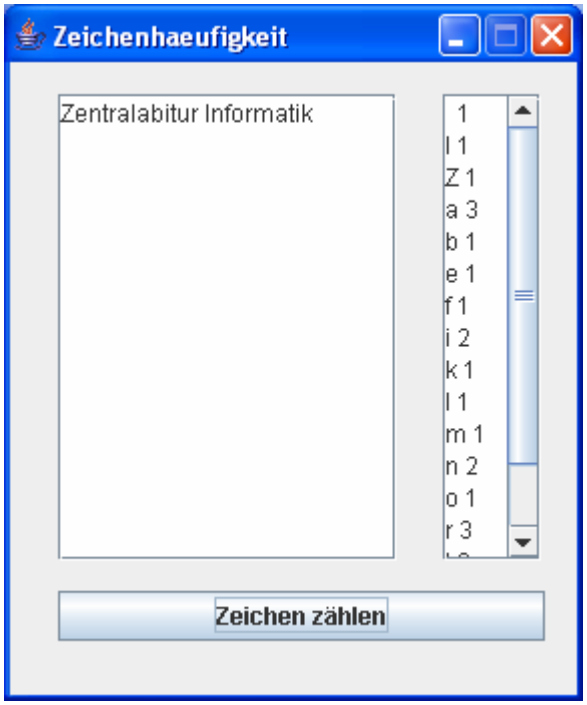
## 8. Beispiel für die Anwendung der Klasse OrderedTree in Java

### a) Aufgabenstellung

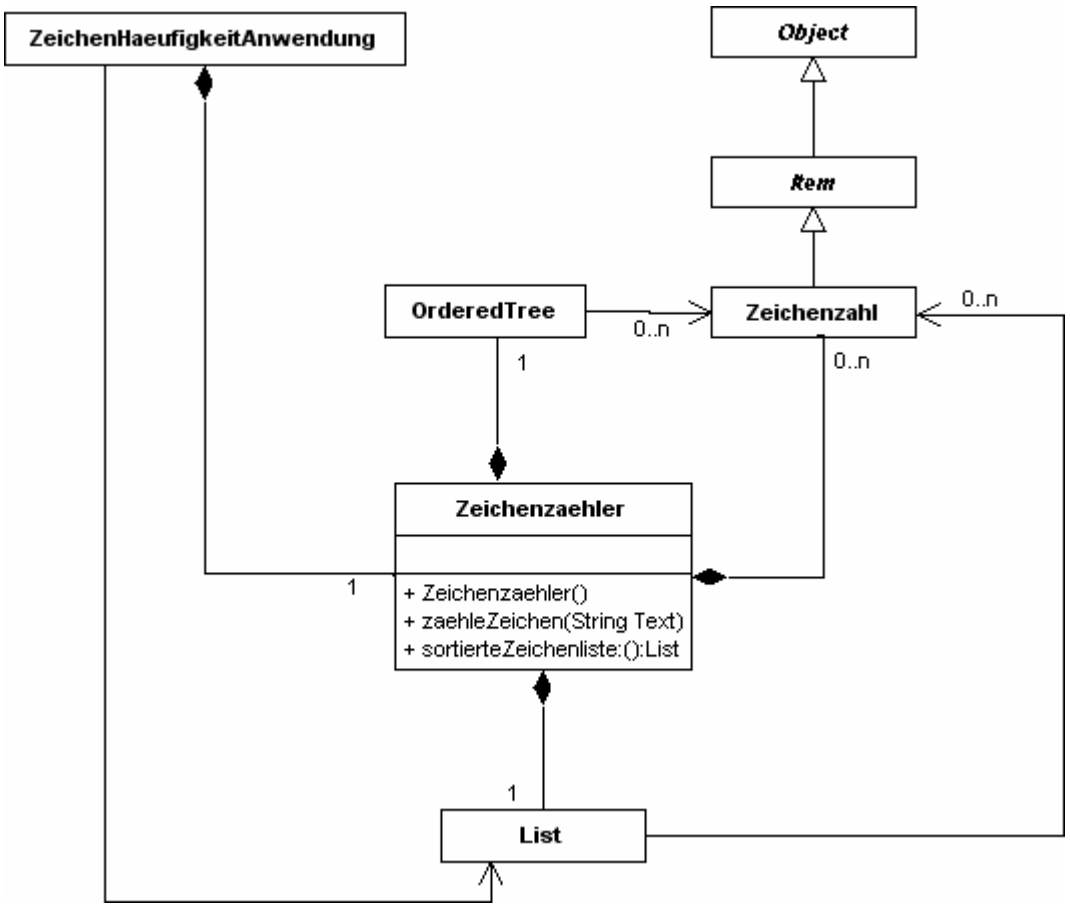
Es soll ein Programm entwickelt werden, das in der Lage ist zu zählen, wie oft jedes verwendete Zeichen in einem beliebigen Text vorkommt.
Die Zeichen mit ihrer Häufigkeit werden in einem geordneten Baum gespeichert.
So ergibt sich für das Wort **MEISTERSCHAFT** folgender geordnete Baum:




### b) Benutzerschnittstelle



**c) UML-Diagramm (vereinfacht)**



**d) Die Klasse *Zeichenzahl***

Eine Klasse, deren Objekte in den Knoten eines geordneten Baumes gespeichert werden, muss immer eine Unterklasse von ***Item*** sein. Die Ordnungsrelation für den sortierten Baum wird durch Überschreiben der abstrakten Methoden ***isEqual***, ***isLower*** und ***isGreater*** festgelegt.

Als Beispiel wird die Implementation der Klasse ***Zeichenzahl*** angegeben:

```
class Zeichenzahl extends Item{
  private char zeichen;
  private int anzahl=0;

  public Zeichenzahl(char pZeichen){
    zeichen = pZeichen;
    anzahl = 1;
  }

  public char getZeichen(){
    return zeichen;
  }

  public int getAnzahl(){
    return anzahl;
  }

  public void erhoeheAnzahl(){
    anzahl = anzahl+1;
  }

  public boolean isGreater (Item pItem){
    return (this.zeichen>((Zeichenzahl)pItem).getZeichen());
  }

  public boolean isLower (Item pItem){
   return (this.zeichen<((Zeichenzahl)pItem).getZeichen());
  }
  public boolean isEqual (Item pItem){
    return (this.zeichen==((Zeichenzahl)pItem).getZeichen());
  }
}
```

**e) Implementation einiger Methoden der Klasse *Zeichenzaehler***

Die Methode ***zaehleZeichen*** durchläuft den Text ***pText*** Zeichen für Zeichen. Jedes Zeichen, das zum ersten Mal vorkommt, wird mit der Häufigkeit eins im geordneten Baum gespeichert. Bei jedem weiteren Auftreten wird lediglich die Häufigkeit des bereits gespeicherten Zeichens um eins erhöht.

```
public void zaehleZeichen(String pText){
  char lZeichen;
  Zeichenzahl lAnzahlAlt;
  for (int lZaehler=0; lZaehler<pText.length();lZaehler++ ) {
    lZeichen = pText.charAt(lZaehler);
    Zeichenzahl lAnzahlNeu=new Zeichenzahl(lZeichen);
    lAnzahlAlt=((Zeichenzahl)zeichenZaehlbaum.searchItem(lAnzahlNeu));
    if (lAnzahlAlt==null)
       zeichenZaehlbaum.insertItem(lAnzahlNeu);
    else {
      lAnzahlAlt.erhoeheAnzahl();
    }
  }
}
```

Die Methode ***sortierteZeichenliste*** liefert die nach dem ASCII-Code geordnete Liste der Zeichen mit ihren Häufigkeiten.

```
public List sortierteZeichenliste(){
  return zeichenZaehlbaum.getSortedList();
}
```

## 9. Beispiel für die Anwendung der Klasse Connection

### a) Aufgabenstellung

Besonders einfache Server sind Daytime-Server. Der Dienst "Daytime" wird mit einem sehr einfachen Protokoll genutzt. Nach Herstellen der Verbindung sendet der Server eine Zeichenkette an den Client, die das Datum und die aktuelle Uhrzeit in englischer Sprache angibt.
Es soll ein Daytime-Client entwickelt werden, der unter Nutzung eines öffentlich zugänglichen Daytime-Servers das aktuelle Datum und die aktuelle Zeit anzeigt. Mögliche Daytime-Server, die über das Internet zu erreichen sind, sind z. B *time.fu-berlin.de* oder *cs.columbia.edu* für eine andere Zeitzone Die Port-Nummer des Servers ist jeweils 13.

### b) Benutzeroberfläche


Im Textfeld wird der Name des *Time-Servers* eingegeben, nach Betätigen der Schaltfläche Zeit wird die Verbindung zum Server hergestellt. Dieser sendet als String das aktuelle Datum mit der aktuellen Zeit an den TimeClient, der Datum und Uhrzeit im Formular anzeigt. Anschließend wird die Verbindung zum Server geschlossen.

### c) Implementation der Methode `holeZeit`

```
String holeZeit(String pServer){
  Connection lTimeClient=new Connection(pServer,13);
  String lTime=lTimeClient.receive();
  lTimeClient.close();
}
```

## 10. Beispiel für die Anwendung der Klassen Client und Server

### a) Aufgabenstellung

Auf den Computern einer Schule soll ein interner Chat-Dienst eingerichtet werden. Dazu müssen ein Chat-Client und ein Chat-Server-Programm entwickelt werden. Zunächst muss ein Protokoll *Chat-Dienst* erstellt werden, mit dem alle Teilnehmer Nachrichten austauschen, die dann auf jedem angeschlossenen Computer angezeigt werden. Dabei sollen in der ersten Version keine Spitznamen, sondern zur Identifikation der Chat-Teilnehmer nur die IP- und Port-Nummern verwendet werden. Eine weitere Vereinfachung ergibt sich dadurch, dass zunächst alle Chat-Nachrichten an alle Teilnehmer geschickt werden und Nachrichten an einzelne Teilnehmer noch nicht möglich sein sollen.

### b) Das Protokoll für den Chat-Dienst

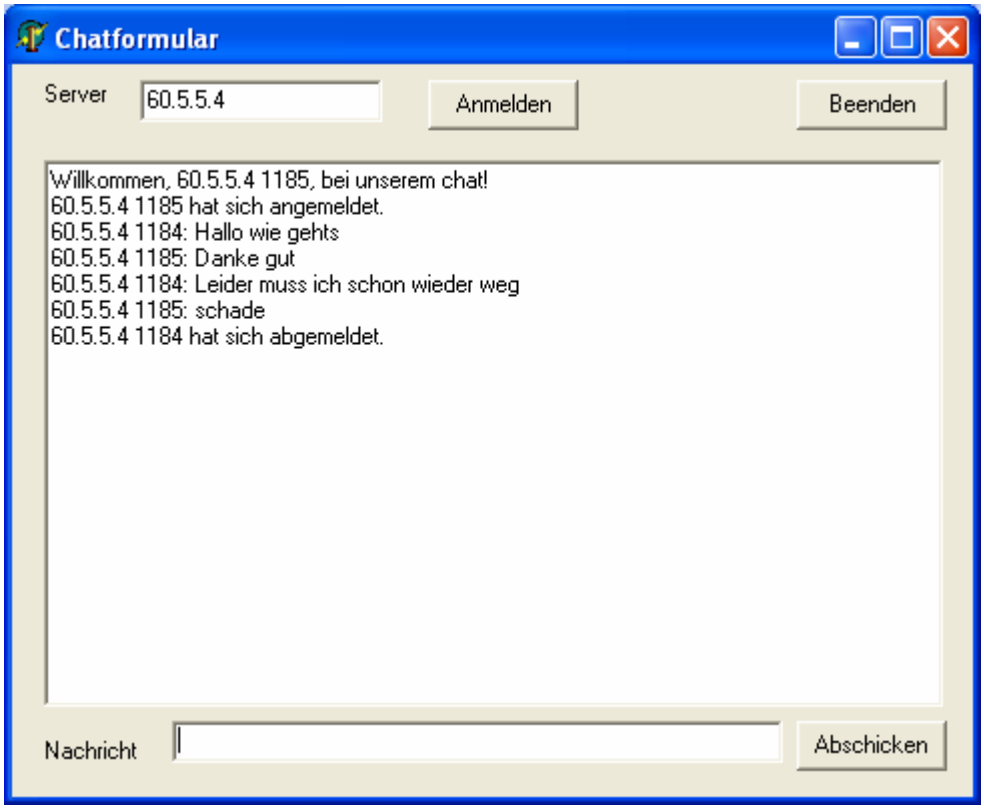
**Anmeldephase:**
Nach der Anmeldung eines neuen Clients schickt ihm der Server die Nachricht *"Willkommen, " + IP-Nummer und Port des neu angemeldeten Clients + ", bei unserem Chat!".* Danach schickt er allen Chat-Teilnehmern die Nachricht *IP-Nummer und Port des neu angemeldeten Clients + " hat sich angemeldet."*

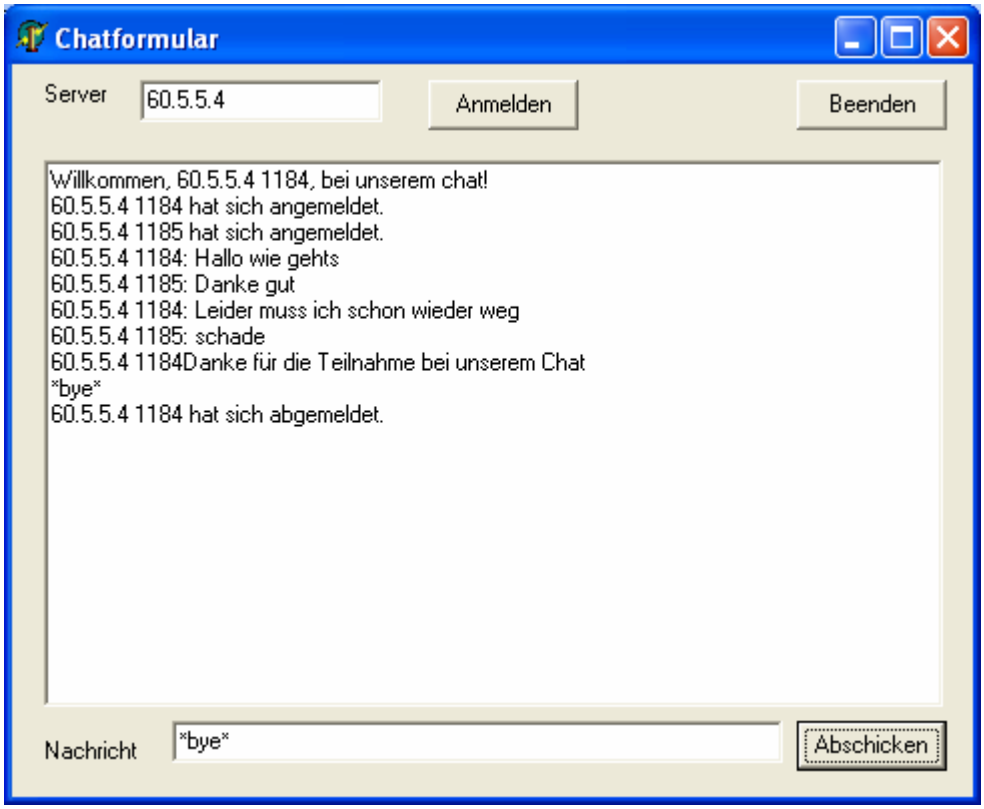
**Arbeitsphase:**
Die Clients schicken ihre Chatbeiträge als Nachricht an den Server.
Der Server verteilt jeden Beitrag an alle Chat-Teilnehmer und setzt dabei die IP-Nummer und den Port des Absenders, gefolgt von ": ", davor.

**Abmeldephase:**
Der Client beendet die Verbindung, indem er dem Server die Nachricht *"*bye**"* schickt. Daraufhin antwortet der Server dem Client mit *"Danke für die Teilnahme bei unserem Chat!"*, schickt an alle Chat-Teilnehmer die Nachricht *IP-Nummer und den Port des Clients, der sich abmeldet + " hat sich abgemeldet."* Zum Abschluss sendet er dem abmeldenden Client die Nachricht *"*bye**"*, worauf beide jeweils die Verbindung trennen.

Zum technischen Teil des Protokolls gehört die Festlegung auf eine Portnummer. Bei selbstdefinierten Diensten sollte sie größer als 1023 sein, hier wird 2000 benutzt.

**c) Benutzeroberfläche für zwei Clients, die miteinander chatten**



## d) Implementationen

### Die Klasse ChatClient

```
import
javax.swing.*;

public class ChatClient extends Client {

  final String ENDE="*bye*";
  JTextArea hatTextbereich;

  public ChatClient(String pIPAdresse, int pPortNR, JTextArea pTextbereich){
    super("127.0.0.1",2000,true);
    hatTextbereich=pTextbereich;
  }

  public void processMessage(String pMessage){
     if (!pMessage.equals(ENDE))
      hatTextbereich.setText(hatTextbereich.getText()+pMessage);
  }

  public void disconnect() {
    this.send(ENDE);
  }
}
```

### Senden einer Nachricht

```
public void jNachrichtButtonActionPerformed(ActionEvent evt) {
    String lNachricht=jTextField2.getText();
    if (lNachricht.length()>0) {
      hatChatClient.send(lNachricht);
    }
}
```

### Die Klasse ChatServer

```
public  class ChatServer extends Server {
   final String ENDE="*bye*";

   public ChatServer(int pPortNr, boolean pTestModus){
     super (2000,true);
   }

   public void processNewConnection (String pClientIP,int pClientPort){
     this.send(pClientIP,pClientPort,"Willkommen, "+pClientIP+" "+
               Integer.toString(pClientPort)+", bei unserem chat!");
     this.sendToAll(pClientIP+" "+ pClientPort + "hat sich angemeldet.");
   }

   public void processMessage(String pClientIP, int pClientPort, String pMessage){
     if (pMessage.equals(ENDE))
       this.closeConnection(pClientIP,pClientPort);
     else
       this.sendToAll(pClientIP+" "+ pClientPort +": "+pMessage);
   }

   public void processClosedConnection(String pClientIP, int pClientPort){
     this.send(pClientIP,pClientPort, pClientIP+" "+ pClientPort +
                      "Danke für die Teilnahme bei unserem Chat.");
     this.send(pClientIP,pClientPort,ENDE);
     this.sendToAll(pClientIP+" "+ pClientPort + " hat sich abgemeldet");
   }
}
```