

Inhalte Informatik

I1 Grundprinzip des objektorientierten Modellierens **I3 Modellieren von Netzwerkanwendungen**

II.0 Grundlegende Programmstrukturen und Algorithmen

- Sortier- und Suchalgorithmen auf Arrays
 - Sortieren durch direktes Einfügen
 - Sortieren durch direktes Auswählen
 - Quicksort (nur Leistungskurs)
- Lineare Suche
- Binäre Suche
- Rekursion

II.1 Datenstrukturen

- Lineare Strukturen mit den Akzenten
 - Schlange und Stapel
 - Anwendung der Standardoperationen
 - Implementation der Standardoperationen
 - Lineare Liste
 - Anwendung der Standardoperationen
- Baumstrukturen mit den Akzenten
 - Binärbaum
 - Anwendung der Standardoperationen
 - Traversierungsverfahren
 - Geordneter Baum als Spezialfall des Binärbaums
 - Anwendung der Standardoperationen

II.3 Client-Server-Strukturen

- Schichtenmodell: TCP/IP-Referenzmodell
- Protokolle: TCP/IP, http, Mail-Protokolle
- Client-Anwendungen (z. B. Time-Client oder E-mail-Client)
- Client-Server-Anwendungen (z.B. Chat-Programm oder Netzwerk-Spiel)

Basis-Sprachelemente und -Datentypen

Sprachelemente

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

| 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 | | |

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 enqueue (Object pObject)

Vorher Die Schlange ist erzeugt.

Nachher *pObject* ist als letztes Element in der Schlange abgelegt.

Auftrag dequeue ()

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 Stapелеlement. 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.

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* sowie 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. Der Binärbaum ist unverändert.

Anfrage *getRightTree*(): BinTree

vorher Der Binärbaum ist nicht leer.

nachher Diese Anfrage liefert den rechten Teilbaum der Wurzel des Binärbaums. 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 *OrderedTree* ()

nachher Der geordnete Baum existiert und ist leer.

Anfrage *isEmpty*(): *boolean*

nachher Diese Anfrage liefert den Wahrheitswert *true*, wenn der geordnete Baum leer ist, sonst liefert sie den Wert *false*.

Auftrag *insertItem*(*Item pItem*)

nachher *pItem* ist entsprechend der Ordnungsrelation in den Baum eingeordnet.

Anfrage *searchItem* (*Item pItem*): *Item*

nachher Falls ein bezüglich der verwendeten Ordnungsrelation mit *pItem* übereinstimmendes Objekt im geordneten Baum enthalten ist, liefert die Anfrage dieses, ansonsten wird *null* zurückgegeben.

Auftrag *deleteItem*(*Item pItem*)

nachher 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.

Client-Server

Die Klasse Connection

Objekte der Klasse *Connection* ermöglichen eine Netzwerkverbindung mit dem TCPProtokoll.

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 *Connection* (String pServerIP, int pServerPort)

nachher 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 *send* (String pMessage)

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

Anfrage *receive* () :String

nachher 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 pMessage)
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 pMessage)`

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, Nachrichtempfang 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.