2.8) SQL-Datenbankzugriff mit PHP

Um auf eine MySQL-Datenbank zuzugreifen, muss zuerst eine Verbindung (***mysqli_connect***) hergestellt werden. Dazu ist der **Host**, auf dem der MySQL-Server läuft, der **Benutzername** sowie das **Kennwort** anzugeben. Anschließend muss noch eine Datenbank auf dem DBS ausgewählt werden (***mysqli_select_db***). Danach können die SQL-Anweisungen folgen (***mysqli_query***). Im Falle einer SQL-Abfrage liefert diese Funktion alle Tupel zurück, welche der Reihe nach mit ***mysqli_fetch_array*** abgearbeitet werden können.

Im Folgenden soll als Beispiel ein Gästebuch realisiert werden. Dazu wurde eine Tabelle **meldung** erstellt, welche die einzelnen Nachrichten aufnehmen soll:

```
CREATE TABLE meldung (
  id INT NOT NULL AUTO_INCREMENT,
  datum DATETIME DEFAULT NULL,
  name VARCHAR(200) DEFAULT NULL,
  eintrag TEXT,
  CONSTRAINT PK_id PRIMARY KEY (id)
);
```

Zusätzlich zu den Daten **datum, name** und **eintrag** wurde ein Attribut **id** eingeführt, welches als Schlüssel dient.

Das folgende Skript **show.php** liest alle Einträge der Datenbank aus und gibt sie in Tabellenform aus:

```
<?php
    $DBHost = "127.0.0.1";
    $DBUser = "root";
    $DBPasswd = "";
    $DBName = "db";

    //Verbindung zu DB-Server herstellen
    $con=mysqli_connect($DBHost, $DBUser, $DBPasswd, $DBName)
      OR die("Konnte DB-Server nicht erreichen!");
    mysqli_select_db($con,$DBName);
?>
<html>
    <head>
        <title>Alle Meldungen</title>
    </head>
    <body>
    <?php
        $erg=mysqli_query($con, "SELECT datum, name, eintrag FROM meldung
ORDER BY datum DESC");
        echo mysqli_error($con);

        while($row=mysqli_fetch_array($erg))
        {
            echo "<table bgcolor=\"#dddddd\" border=\"0\" width=\"600\">
```

```
\n";
            echo "<tr><td>Datum: </td><td>".$row["datum"]."</td></tr> \n";
            echo "<tr><td>Name:</td><td>" .$row ["name"]. "</td></tr> \n";
            echo "<tr><td valign=\"top\">Eintrag:</td> \n";
            echo "<td>".nl2br(htmlentities($row["eintrag"])."</td></tr>
\n";
            echo "</table>\n<br>\n";
        }
    ?>
    <hr><a href="insert.php">neuen Eintrag hinzufügen</a>
    </body>
</html>
```

Datum:	2018-11-15 20:19:20
Name:	Prof. Andreas Lahmer
Eintrag:	Hier sehen sie die SQL-Anbindung mittels PHP in Form eines Gästebuchs!

---

neuen Eintrag hinzufügen

Um neue Einträge für das Gästebuch zu erstellen, existiert ein weiteres Skript insert.php:

```
<?php
    $DBHost = "127.0.0.1";
    $DBUser = "root";
    $DBPasswd = "";
    $DBName = "db";

    //Verbindung zu DB-Server herstellen
    $con=mysqli_connect($DBHost, $DBUser, $DBPasswd)
      OR die("Konnte DB-Server nicht erreichen!");
    mysqli_select_db($con,$DBName);
?>
<html>
    <head>
        <title>Neuer Eintrag in unser Gästebuch</title>
    </head>
    <body>
        <?php

            if(isset($_GET["submit"]))
            {
                $submit = $_GET["submit"];
                $name = $_GET["name"];
                $eintrag = $_GET["eintrag"];

                //Der Submit-Button wurde gedrückt --> die Werte müssen
überprüft werden
                //und bei Gültigkeit in die DB eingefügt werden
```

```
                $DatenOK=1;         //wir gehen prinzipiell von der
Gültigkeit der Daten aus
                $error="";         //es gab noch keine Fehlermeldung bis hier
hier

                if($name=="")    //Kein Name eingegeben
                {
                    $DatenOK=0;
                    $error.="Es muss ein Name eingegeben werden!<br>\n";
                }
                if($eintrag=="")    //Kein Kommentar eingegeben
                {
                    $DatenOK=0;
                    $error.="Ein Eintrag ohne Kommentar macht nicht viel
Sinn!<br>\n";
                }
                if($DatenOK)        //Daten OK -> also in DB eintragen
                {
                                        $timestamp=date("Y-m-d h:i:s",
time());
                    mysqli_query($con,"INSERT INTO eintraege (datum, name,
eintrag) VALUES (\"$timestamp\", \"$name\", \"$eintrag\" );");
                    echo mysqli_error($con);
                    echo "<b>Daten wurden eingetragen.<b>";
                }
                else
                {
                    echo "<h2>Fehler: </h2>\n"; //Fehlermeldung
                    echo $error;
                }
            }

            //Formular
        ?>



        <form action="insert.php" method="GET">
            Name: <input type="text" name="name" size="30" maxlength="200"
value=""><br>
            Text: <br><textarea rows="10" cols="50" wrap="virtual"
name="eintrag"></textarea>
            <br><input type="submit" name="submit" value="Absenden">
        </form>
        <a href="show.php"> Alle Einträge anzeigen</a>
    </body>
</html>
```

Name:

Text:

Absenden

Alle Einträge anzeigen

2.7.3) DML (Data Manipulation Language)

Unsere Tabellen sind angelegt und warten darauf, mit Daten gefüttert bzw. durchsucht zu werden. Lernen Sie im Folgenden, wie das gemacht wird.
Die Data Manipulation Language umfasst insgesamt 3 Arten von Statements:

2.7.3.1) Einfügen von Datensätzen

Bevor Sie mit einer Datenbank arbeiten, sollten zunächst einmal Daten in der Datenbank vorhanden sein.
Um Daten in eine Datenbank einzufügen, benutzen wir den Befehl INSERT. Dessen Aufbau ist recht einfach:

```
INSERT INTO <Tabelle> [(Spalte1, Spalte2,...)] VALUES (<wert1>,
<wert2>,...);
```

Als Beispiel wollen wir einen Datensatz in die Tabelle Administratoren einfügen:

```
INSERT INTO Administratoren (UserID, USER, Passwort) VALUES
(1,'Meister','geheim');
```

Die Zuordnung, welcher Wert in welche Spalte eingefügt wird, geschieht über die jeweiligen Positionen innerhalb der Klammern;
⇒ der erste Wert „Meister“ wird in die erste Spalte „User“ eingefügt.

Wenn wir einen neuen Datensatz anlegen, müssen wir nicht alle Spalten sofort füllen. Die Tabelle Administratoren hat eigentlich drei Spalten. Damit das aber funktioniert und sich das DBMS nicht beschwert, darf die Spalte nicht als **NOT NULL** definiert sein. Oder, wenn sie es ist, muss sie einen Defaultwert aufweisen.

2.7.3.2) Ändern von Datensätzen

Einen Datensatz können wir mit UPDATE ändern - Vorsicht, es ist nicht möglich, damit einen Datensatz einzufügen.

Hier die vollständige Syntax:

```
UPDATE Relation
   SET <Spalte1>=<Wert> [, <Spalte2>=<Wert>, ...]
     [WHERE <Bedingung>]
```

Die WHERE-Bedingung ist optional, Sie haben dieselben Möglichkeiten zum Formulieren wie bei der SELECT-Anweisung. Geben Sie keine Bedingung an, wirkt sich ein Update auf alle aufgezählten Spalten aus.

Wollen wir die Spalte User aller Datensätze auf einen neuen Wert setzen, schreiben wir:

```
UPDATE Administratoren SET Passwort='noch geheimer';
```

Wollen wir nur einen einzelnen Datensatz ändern bzw. nur einige, so müssen wir eine entsprechende WHERE-Klausel formulieren.

```
UPDATE Administratoren
   SET Passwort='ganz geheim'
   WHERE UserID='1';
```

Sie sollten für das Update eines einzelnen Datensatzes unbedingt dessen Primärschlüssel als Bedingung nehmen, wie in diesem Beispiel. Die Bedingung über die Spalte User zu formulieren, würde nur sicher sein, wenn wir beim Einfügen des Datensatzes darauf achten, in dieser Spalte keine Werte doppelt zu haben.

2.7.3.3) Löschen von Datensätzen

Was wir zum Abschluss des Abschnitts über das UPDATE gesagt haben, gilt auch für die DELETE-Anweisung. Benutzen Sie zum Löschen mit DELETE immer eine WHERE-Klausel mit Bezug auf den Primärschlüssel, außer Sie wollen wirklich alle Datensätze einer Tabelle löschen.

Die Syntax zum Löschen eines oder mehrerer Datensätze lautet:

```
DELETE FROM Relation
   [WHERE <Bedingung>]
```

Auch hier ist die WHERE-Klausel optional, wird keine angegeben, werden alle Datensätze gelöscht - die Tabelle ist danach leer.

Um eine Eintrag aus der Administrator-Tabelle zu löschen, schreiben wir:

```
DELETE FROM Administratoren WHERE UserID=1;
```

2.7.1) DDL - Data Definition Language

2.7.1.1) Datentypen

In SQL stehen folgende Datentypen zur Auswahl:

- exakt numerisch
  - INTEGER
  - SMALLINT
  - NUMERIC (p,q)
  - DECIMAL (p,q)
- angenähert numerisch
  - REAL
  - DOUBLE
  - FLOAT
- Zeichenketten
  - CHAR
  - VARCHAR
- Bitketten
  - BLOB (Binary Large Objects)
- Datum und Uhrzeit
  - DATE
  - TIME
  - TIMESTAMP
- Logischer Datentyp
  - BOOLEAN

2.7.1.2) ERZEUGEN von Relationenschemata

Mit dem Befehl „**CREATE TABLE *relationenname* (...)**“ wird ein Relationenschema definiert. Zu jedem Attribut wird ein Typ angegeben. Optional können eine oder mehrere Integritätsbedingungen angegeben werden. Mögliche Integritätsbedingungen sind:

- die Angabe, dass das Attribut einen Primärschlüssel darstellt (**PRIMARY KEY**)
- dass das Attribut keinen Nullwert annehmen darf (**NOT NULL**)
- dass ein Attribut eindeutig sein muss (**UNIQUE**) - diese Bedingung gilt automatisch für Primärschlüsselattribute
- oder eine durch die Klausel „**CHECK (*bedingung*)**“ formulierte Wertebereichseinschränkung

Weiters kann für Attribute durch die Klausel „**DEFAULT *wert***“ ein Vorgabewert angegeben werden.

Beispiel

Es soll eine leere Relation mit dem Namen **Restaurant** erstellt werden. Es wird festgelegt, dass das Attribut **rnr** ein Primärschlüssel ist, dass die Attribute **name** und **adresse** keine Nullwerte annehmen

dürfen, und dass **haube** nur die Werte (0-4) annehmen darf. Für den Typ eines Restaurants ist als Defaultwert „österreichisch“ festzulegen.

```
CREATE TABLE restaurant
(
    rnr INT NOT NULL,
    name VARCHAR(100) NOT NULL,
    adresse    VARCHAR(100) NOT NULL,
    haube INT CHECK (0<=haube AND haube <=4),
           /*oder CHECK (haube IN (0,1,2,3,4)),*/
           /*oder CHECK (haube BETWEEN 0 AND 4),*/
    typ VARCHAR(100) DEFAULT 'österreichisch',
    CONSTRAINT PK_Restaurant PRIMARY KEY(rnr)
);
```

Eine spezielle Integritätsbedingung (zu Attributen oder zu Relationen) ist die mittels einer **REFERENCES-Klausel** angegebene Fremdschlüsselbedingung **FOREIGN KEY**, die eine Abhängigkeit repräsentiert. Zu jeder Fremdschlüsselbedingung kann durch **„ON UPDATE *action*„** und **„ON DELETE *action*„** angegeben werden, wie auf Verletzungen durch Änderung oder Löschen des referenzierten Schlüsselwertes reagiert werden soll. Mögliche Aktionen sind:

- die Änderung bzw. das Löschen zu verhindern (**NO ACTION**)
- fortzusetzen (**CASCADE**) oder
- den Wert des referenzierenden Attributes auf einen Nullwert (**SET NULL**) bzw. den Defaultwert des Attributs (**SET DEFAULT**) zu setzen

Beispiel

Der nächste SQL-Befehl erzeugt das Relationenschema **Speise**. Die angeführten Integritätsbedingungen legen u.a. fest, dass die Attribute **rnr** und **name** gemeinsam den Primärschlüssel der Relation **Speise** bilden und dass das Attribut **rnr** der Relation **Speise** ein Fremdschlüssel ist, der sich auf das Schlüsselattribut **rnr** der Relation **Restaurant** bezieht (**Speise[rnr] ⊆ Restaurant[rnr]**). Die Angabe **ON UPDATE CASCADE** legt fest, dass eine Änderung der Nummer eines Restaurants bei den entsprechenden Speisen mitgezogen wird, und die Angabe **ON DELETE NO ACTION** legt fest, dass ein Restaurant nicht gelöscht werden darf, solange noch Speisen für dieses Restaurant vorhanden sind.

```
CREATE TABLE Speise
(
    rnr INTEGER,
    name VARCHAR(150),
    CONSTRAINT PK_Speise PRIMARY KEY (name),
    CONSTRAINT FK_Speise FOREIGN KEY (rnr) REFERENCES restaurant(rnr) ON
UPDATE CASCADE ON DELETE NO ACTION
);
```

2.7.1.3) ÄNDERN von Relationenschemata

Die Änderung der Struktur einer Tabelle wird mit dem Befehl „**ALTER TABLE *relationenname…***“ durchgeführt.

Beispiel - Hinzufügen eines Attributes KALORIEN zur Relation SPEISE

```
ALTER TABLE Speise ADD Kalorien INT;
```

Beispiel - Hinzufügen einer CHECK-Klausel für Kalorien

```
ALTER TABLE Speise ADD CONSTRAINT ck CHECK (Kalorien>=0);
```

Beispiel - Entfernen eines Attributes

```
ALTER TABLE Restaurant DROP Adresse;
```

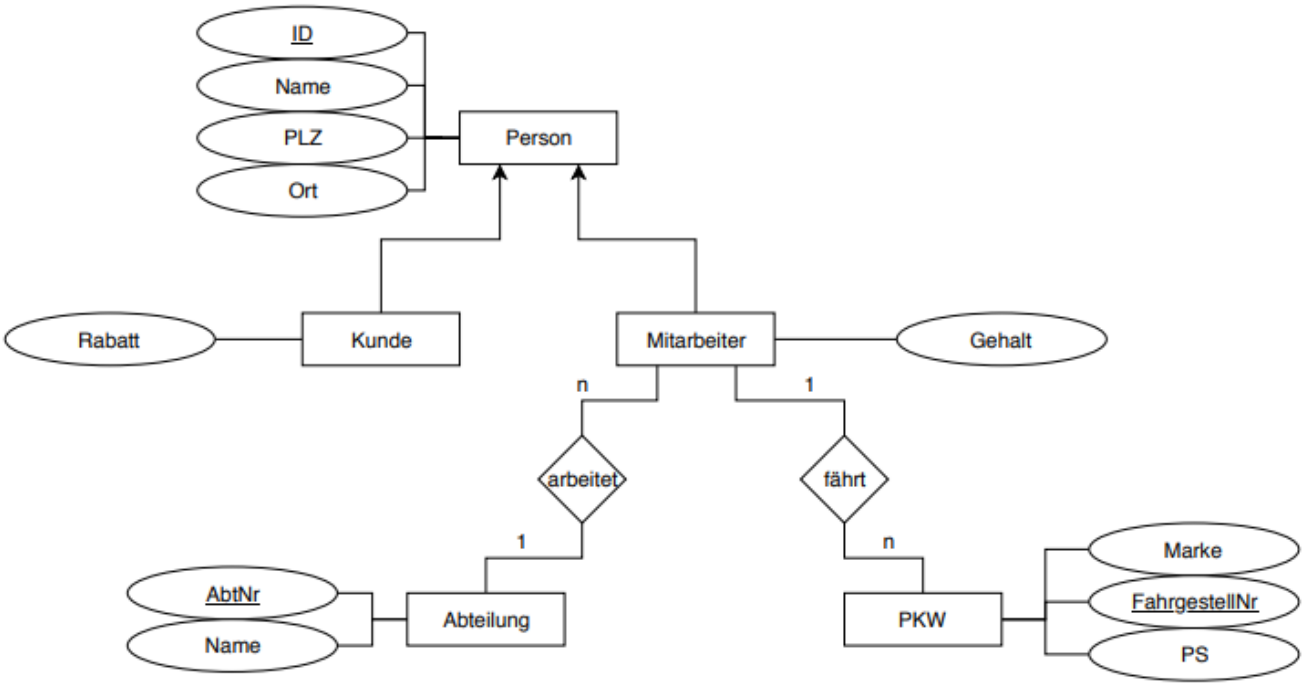
2.7.1.4) ENTFERNEN von Relationenschemata

Mit dem Befehl „**DROP TABLE *relationenname***“ wird eine Relation wieder gelöscht.

```
DROP TABLE Restaurant;
```

2.7.1.5) Übungsbeispiel

Gegeben ist folgendes ER-Modell:



Gesucht sind die SQL-Statements zur Erstellung der Relationen

```
CREATE TABLE Orte (
    Plz INT NOT NULL,
    Ort VARCHAR(100),
    CONSTRAINT PK_Plz PRIMARY KEY(Plz)
);

CREATE TABLE Person (
    ID INT NOT NULL,
    Name VARCHAR(100),
    Plz INT NOT NULL,
    CONSTRAINT FK_Orte FOREIGN KEY (Plz) REFERENCES Orte(Plz),
    CONSTRAINT PK_ID PRIMARY KEY (ID)
);

CREATE TABLE Kunde (
    ID INT NOT NULL,
    Rabatte INT NOT NULL,
    CONSTRAINT PK_ID PRIMARY KEY (ID),
    CONSTRAINT FK_ID FOREIGN KEY (ID) REFERENCES Person(ID) ON DELETE
CASCADE
);

CREATE TABLE Abteilung(
    AbtNr INT NOT NULL,
    Name VARCHAR(140),
    CONSTRAINT PK_AbtNr PRIMARY KEY(AbtNr)
);

CREATE TABLE Mitarbeiter(
    ID INT NOT NULL,
```

```
    Gehalt INT NOT NULL,
    AbtNr INT NOT NULL,
    CONSTRAINT PK_AbtNr PRIMARY KEY(ID),
    CONSTRAINT FK_AbtNr FOREIGN KEY(AbtNr) REFERENCES Abteilung(ID),
    CONSTRAINT FK_ID FOREIGN KEY(ID) REFERENCES PERSON(ID)
);

CREATE TABLE PKW(
    Marke VARCHAR(100),
    PS INT NOT NULL,
    FahrgestellNr INT NOT NULL,
    ID INT NOT NULL,
    CONSTRAINT PK_fgsNr PRIMARY KEY (FahrgestellNr),
    CONSTRAINT FK_ID FOREIGN KEY (ID) REFERENCES Mitarbeiter(ID)
);
```

Doppelt verkettete Listen

Doppelt verkettete Listen (oder doubly linked lists) sind häufig benutze Datenstrukturen und eine Verallgemeinerung der einfach verketteten Listen, die ich hier anhand von Beispielen in der Programmiersprache C++ vorstellen will.

Sie funktioniert in jeder Programmiersprache genauso, die Zeiger oder Referenzen und so etwas wie Klassen oder Structs zur Verfügung stellt.

Wozu?

Doppelt verkettete Listen, oder „double linked lists“ braucht man immer dann, wenn man sich in einer Liste leicht vorwärts und rückwärts bewegen können muss, und wenn man schnell und einfach Elemente der Liste an beliebigen Positionen löschen und neue einfügen muss.

Denn einfach verkettete Listen haben den Nachteil, dass man sie nur in einer Richtung durchlaufen kann.

Darüber hinaus kann man ein referenziertes Listenelement nicht unmittelbar löschen, da man von diesem Element keinen Zugriff auf das Vorgängerelement hat.

Doppelt verkettete Listen umgehen dieses Problem, indem sie in jedem Knoten zusätzlich noch eine Referenz auf den Vorgängerknoten speichern.

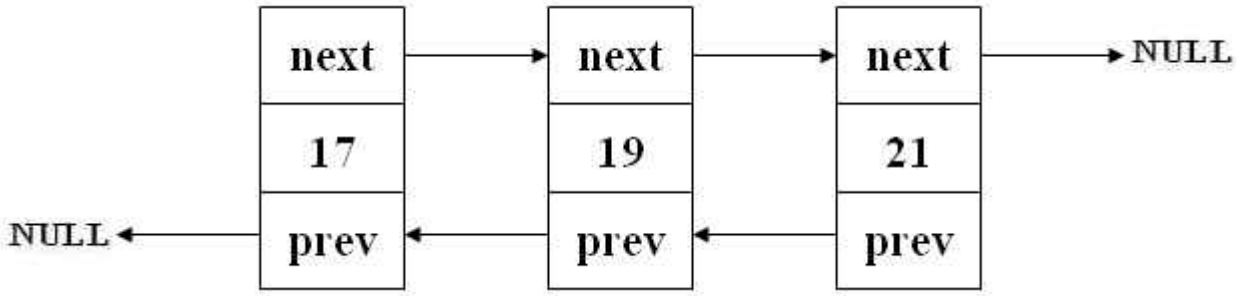
Die Grundidee

Im Gegensatz zu einfach verketteten Listen haben doppelt verkettete Listen in den Knoten eine zusätzliche Instanzvariable für die Referenz auf den Vorgängerknoten

Die Idee einer doppelt verketteten Liste ist es also, für jedes Element zwei Zeiger zu speichern, einen nach links und einen nach rechts:

```
struct DList{
    DList*left;
    DList *right;
    int data;
};
```

Die eigentlichen Daten, die in der Liste gespeichert werden sollen, stehen in data, das hier als int definiert ist. Jeder andere Datentyp, z.B. ein Zeiger auf beliebige andere Strukturen, funktioniert genau so.



Einfügen, Löschen und Ausgeben von Elementen

```
#include <iostream>

using namespace std;

struct DListe{
    DListe *next;
    DListe *prev;
    int zahl;
};


int main(int argc, char** argv) {

    DListe *head=NULL;        //Anlegen des Head-Pointers
    DListe *tail=NULL;        //Anlegen des Tail-Pointers
    DListe *help=NULL;        //Anlegen des Help-Pointers (für das Löschen
und Ausgeben)


    //Einfügen von 10 Elementen in eine doppelt verkettete Liste
    for(int i=0;i<10;i++)
    {
        DListe *elem=new DListe();        //Element erzeugen
        elem->zahl=i;                     //Attribut zahl befüllen
        if(head==NULL && tail==NULL)    //noch kein Element in der Liste
        {
            head=elem;                    //Head zeigt auf das neue Element
            tail=elem;                    //Tail zeigt auf das neue Element
            elem->next=NULL;            //Next zeigt auf NULL
            elem->prev=NULL;            //Prev zeigt auf NULL
        }
        else                            //Elemente sind bereits vorhanden &
Element wird am Ende eingefügt
        {
            tail->next=elem;            //tail->next auf das neue Element
            elem->prev=tail;            //elem->prev zeigt auf das
ursprünglich letzte Element
            elem->next=NULL;            //elem->next zeigt auf NULL
```

```
            tail=elem;                    //tail zeigt auf das neue letzte
Element
        }
    }

    cout << "Geben Sie an, welches Element (0-9) Sie loeschen wollen!" <<
endl;
    int loesche=0;
    cin >> loesche; //zB. Element mit der zahl=2 wird gelöscht

    help=head;                    //help zeigt auf head (Anfang der Liste)
    while(help->zahl!=loesche)    //Wenn help->zahl!=loesche, dann wandert
help in der Liste weiter
    {
        help=help->next;
    }
    //Überprüfe nochmals mit Ausgabe, ob beim richtigen Element (=2)
angekommen?
    cout << help->zahl;
    //Sonderfall 1. Element soll gelöscht werden (zahl=0)
    if(help==head)
    {
        head->next->prev=NULL;
        head=head->next;
        head->prev=NULL;
    }
    else if(help==tail)  //Sonderfall letztes Element soll gelöscht werden
    {
        tail=help->prev;
        help->prev->next=help->next;
    }
    else    //Element ist zwischen 2 anderen Elementen
    {
        help->prev->next=help->next;
        help->next->prev=help->prev;
    }

    //Ausgabe der doppelt verketteten Liste
    cout << endl << endl;
    help=head;
    while(help!=NULL)
    {
        cout << help->zahl << " -> ";
        help=help->next;
    }

    return 0;
}
```

Sortiertes Einfügen

```
      do{
        DListe *elem=new DListe();
        cout << endl << "Geben Sie ein Element ein, das sie einfuegen
wollen!" << endl;
        cin >> elem->zahl;
        elem->next=NULL;
        elem->prev=NULL;
        help=head;

        while((elem->zahl>help->zahl) && (help->next!=NULL))
        {
            help=help->next;
        }

        if(head==help)    //1. Fall => Element wird an erster Stelle
eingefügt
        {
            elem->next=help;
            help->prev=elem;
            head=elem;
        }
        else if(tail==help) //2. Fall => Element wird an letzter Stelle
eingefügt
        {
            elem->prev=help;
            help->next=elem;
            tail=elem;
        }
        else    //3. Fall => Element wird in der Mitte, also zwischen 2
anderen Elemente eingefügt
        {
            elem->next=help;
            help->prev->next=elem;
            elem->prev=help->prev;
            help->prev=elem;
        }

        //Ausgabe der Doppelt verketteten Liste
        ausgabe(head);

        cout << "Wollen Sie noch ein Element einfuegen (j/n)" << endl;
    }while(getch()=='j');
```