Verkettete Listen
in
C++
und
Visual Basic 6

Stefan Buchgeher

8. März 2007

Inhaltsverzeichnis

1 Grundlegendes zu “Verketteten Listen“ (engl. linked list) 2
1.1 Was ist eine “Verkettete Liste“ . . . 2
1.2 Aufbau einer “Verketteten Liste“ . . . 2
1.3 Arten von “Verketteten Listen“ . . . 3

2 Hinweise zur Realisierung einer “Doppelt verketteten Liste“ 5
2.1 Einen Knoten hinzufügen . . . 5
2.2 Daten eines Knoten ändern (editieren) . . . 11
2.3 Einen Knoten löschen . . . 11
2.4 Gesamte Liste löschen . . . 18
2.5 In der Liste blättern (scrollen) . . . 19

3 Realisierung einer doppelt verketteten Liste in C++ 20
3.1 Klasse `CNode` . . . 21
3.2 Klasse `CLinkedList` . . . 24
3.3 Demonstrationsbeispiel . . . 34

4 Realisierung einer einfach verketteten Liste in Visual Basic 37
4.1 Klasse `clsNodeE` . . . 38
4.2 Klasse `clsLinkedListE` . . . 39
4.3 Demonstrationsbeispiel . . . 44

5 Realisierung einer doppelt verketteten Liste in Visual Basic 46
5.1 Klasse `clsNodeD` . . . 47
5.2 Klasse `clsLinkedListD` . . . 48
5.3 Demonstrationsbeispiel . . . 56

Kapitel 1

Grundlegendes zu “Verketteten Listen“ (engl. linked list)

1.1 Was ist eine “Verkettete Liste“

Eine “Verkettete Liste“ (engl. linked list) ist eine sehr einfache Möglichkeit zum Speichern einer umfangreichen Datenmenge und ist, im Sinne der Informatik, eine grundlegende dynamischen Datenstruktur.
Unter einer dynamischen Datenstruktur versteht man eine Datenstruktur die zwar vom Programmierer festgelegt wurde (z.B. ein Telefonbuch soll die Informationen Vorname, Nachname, Telefonnummer usw. enthalten) aber wie viele solcher Telefonnummern gespeichert werden soll ist zum Zeitpunkt der Programmerstellung nicht bekannt.

1.2 Aufbau einer “Verketteten Liste“

Die wesentliche Komponente einer verketteten Liste ist der so genannte Knoten (engl. node). Abbildung 1.1 zeigt den Aufbau eines solchen Knotens (Node).

Abbildung 1.1: Aufbau eines Knotens (Node)

Ein Knoten beinhaltet die zu speichernden **Nutzdaten** (im einfachsten Fall nur eine bestimmte Zahl, oder bei einer Anwendung als Telefonbuch sämtliche Informationen für eine Person, also Vorname, Nachname, Telefonnummer usw.) und zumindest einen **Zeiger** auf den nächsten Knoten. Die Zeiger stellen somit das Bindeglied zwischen den einzelnen Knoten dar, daher auch der Name “Verkettete Liste“.
Wichtig ist, dass die Adresse des ersten Knoten an irgendeiner Stelle im Programm bekannt sein muss.

1.3 Arten von "Verketteten Listen"

Bei den "Verketteten Listen" unterscheidet man grundsätzlich zwei Arten:

- Einfach verkettete Liste
- Doppelt verkettete Liste

Einfach verkettete Liste

Bei einer "Einfach verketteten Liste" entspricht der Aufbau eines Knotens Abbildung 1.1. Die Verknüpfung der Knoten erfolgt dabei mit nur einem Zeiger. In der Regel zeigt dieser Zeiger immer auf den folgenden Knoten. Abbildung 1.2 zeigt eine solche Liste mit drei Knoten. Die Nutzdaten sind in diesem Beispiel die Zahlen 27, 1234 und -89.

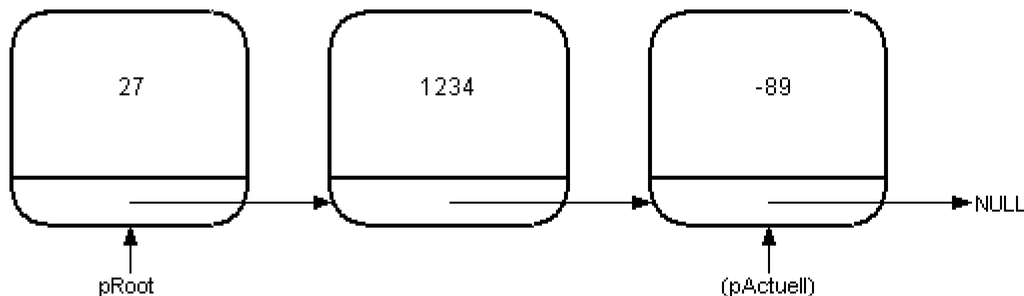

Abbildung 1.2: Einfach verkettete Liste mit drei Knoten

Wie schon erwähnt, muss der erste Knoten der "Verketteten Liste" im Programm bekannt sein. In Abbildung 1.2 übernimmt diese Aufgabe der Zeiger `pRoot`.

Bei den meisten Anwendungen ist es sehr hilfreich, wenn man einen Zeiger hat, der auf den Knoten zeigt, den man gerade bearbeiten möchte. (z.B. um den Inhalt dieses Knotens auszulesen, oder zu verändern, oder aber um an dieser Stelle einen neuen Knoten hinzufügen zu können). In Abbildung 1.2 übernimmt diese Aufgabe der Zeiger `pActuell`.

Das Ende der "Verketteten Liste" erkennt man dadurch, dass der Zeiger des letzten Knotens den Wert `NULL` beinhaltet, da ja kein weiterer Knoten mehr vorhanden ist. Siehe Abbildung 1.2.

Bei der "Einfach verketteten Liste" ist es nur möglich in der Richtung vom ersten Knoten zum letzten Knoten zu gelangen. Eine "Rückwärtsbewegung" ist hier nicht möglich. Dies ist der Nachteil der "Einfach verketteten Liste" gegenüber der "Doppelt verketteten Liste".

Doppelt verkettete Liste

Bei der "Doppelt verketteten Liste" hat jeder Knoten einen zusätzlichen Zeiger. Dieser Zeiger zeigt immer auf den vorhergehenden Knoten. Somit kennt jeder Knoten seinen "Vorgänger-Knoten" und seinen "Nachfolger-Knoten". Abbildung 1.3 zeigt eine solche Liste mit drei Knoten. Die Nutzdaten sind in diesem Beispiel die Zahlen 27, 1234 und -89.

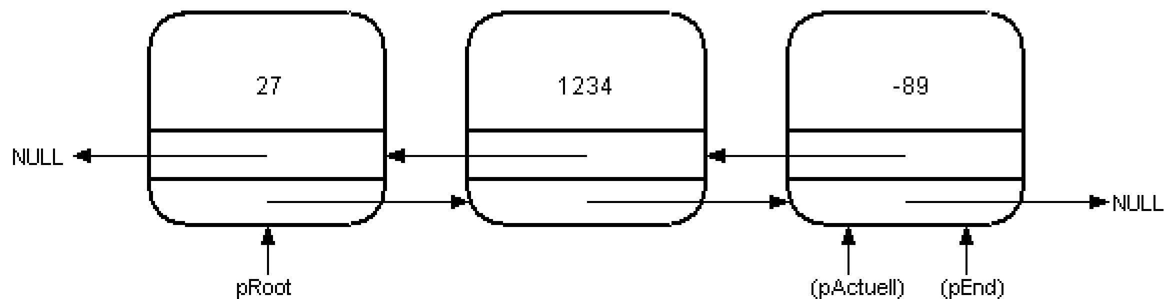

Abbildung 1.3: Doppelt verkettete Liste mit drei Knoten

Der Vorteil der "Doppelt verketteten Liste", nämlich die Bewegung in beiden Richtungen, wird durch den zusätzlichen Zeiger, was einen zusätzlichen Speicherbedarf bedeutet, erkauft.

Auch bei der "Doppelt verketteten Liste" muss der erste Knoten der "Verketteten Liste" im Programm bekannt sein. (Zeiger `pRoot` in Abbildung 1.3).
Dieser Zeiger ist der einzig notwendige. Weitere Zeiger sind zwar manchmal sehr hilfreich aber nicht unbedingt notwendig. Zum Beispiel könnte ein Zeiger hilfreich sein, der immer auf den letzten Knoten zeigt. In Abbildung 1.3 wäre dies der Zeiger `pEnd`. Ein weiterer Zeiger `pActuell`, der die selbe Funktion wie bei der "Einfach verketteten Liste" hat.

Dadurch, dass jeder Knoten auch einen Zeiger zu seinem Vorgänger-Knoten besitzt, muss dafür gesorgt werden, dass dieser Zeiger beim ersten Knoten den Wert `NULL` besitzt, so wie beim letzten Knoten der Zeiger auf den folgenden Knoten ebenfalls den Wert `NULL` beinhalten muss, da es ja in beiden Fällen keinen weiteren Knoten gibt.
Aus diesem Grund muss der Zeiger `pRoot` nicht unbedingt auf den ersten Knoten zeigen. Es reicht wenn er auf irgend einen beliebigen Knoten der "Verketteten Liste" zeigt.

Kapitel 2

Hinweise zur Realisierung einer “Doppelt verketteten Liste“

Dieses Kapitel soll nun einen prinzipiellen Überblick darüber geben, wie man einen “Doppelt verketteten Liste“ realisiert, unabhängig von der verwendeten Programmiersprache.

Speziell die folgenden Punkte möchte ich hier näher betrachten:

- Einen Knoten hinzufügen
- Die Daten eines Knotens ändern (editieren)
- Einen Knoten von der Liste entfernen (löschen)
- Die gesamte Liste löschen
- Sich In der Liste bewegen (scrollen)

Bei der Realisierung von Verketteten Listen spielen die Zeiger eine sehr wichtige Rolle. Daher ist der korrekte Umgang mit Zeigern eine Grundvoraussetzung.

2.1 Einen Knoten hinzufügen

Beim Hinzufügen eines neuen Knotens muss zwischen 3 Fällen unterschieden werden:

- Fall 1: Die Liste ist noch leer, es handelt sich hier also um den ersten Knoten. (Kennzeichen: Der Zeiger `pRoot` ist leer, beinhaltet also den Wert 0 bzw. `NULL`)
- Fall 2: Der neue Knoten befindet sich am Ende der Liste (Kennzeichen: Der Zeiger der auf den nächsten Knoten zeigt ist leer, beinhaltet also den Wert 0 bzw. `NULL`)
- Fall 3: Der neue Knoten befindet sich nicht am Ende der Liste, sondern mittendrin (Kennzeichen: Der Zeiger der auf den nächsten Knoten zeigt ist nicht leer, beinhaltet also einen Wert ungleich 0 bzw. `NULL`. (Ist also das Gegenteil von Fall 2)

a) FALL 1: Die Liste ist noch leer, es handelt sich hier also um den erste Knoten

Schritt 1:
Speicherplatz für den neuen Knoten (dynamisch) anfordern. In der Regel wird dazu ein Zeiger definiert. (Hier soll der Zeiger `pNewNode` heißen)
Ist noch genügend Arbeitsspeicher für den neuen Knoten vorhanden, so wird der Zeiger `pNewNode` mit der Adresse des Arbeitsspeichers, an welche dieser neue Knoten hinterlegt wurde geladen. Der Zeiger `pNewNode` zeigt somit auf jene Adresse im Arbeitsspeicher, wo sich dieser neue Knoten befindet. Ist jedoch nicht mehr ausreichend Arbeitsspeicher vorhanden so wird in der Regel der Zeiger `pNewNode` mit 0 (`NULL`) geladen.

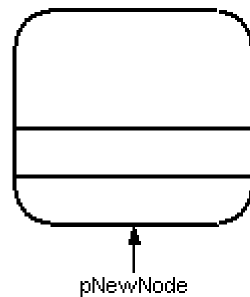

Abbildung 2.1: Ersten Knoten einfügen (Schritt 1)

Schritt 2:
Die zu speichernden Daten eingeben und im neuen (ersten) Knoten (Zeiger `pNewNode`) speichern (Abbildung 2.2).

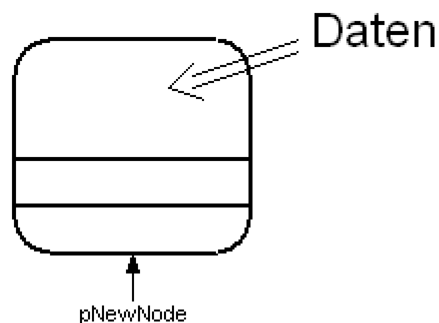

Abbildung 2.2: Ersten Knoten einfügen (Schritt 2)

Schritt 3:
Hier, beim ersten Knoten, die Zeiger auf den vorhergehenden Knoten und auf den folgenden Knoten mit dem Inhalt 0 (`NULL`) laden (Abbildung 2.3).

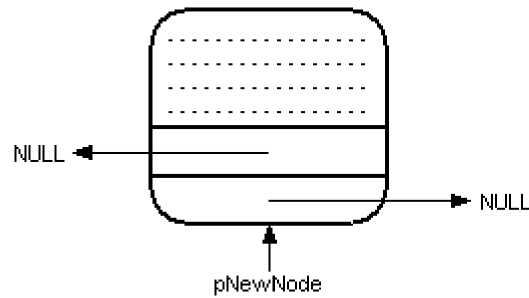

Abbildung 2.3: Ersten Knoten einfügen (Schritt 3)

Schritt 4:
Die Zeiger `pRoot` (für den ersten Knoten), `pEnd` (für den letzten Knoten) und `pActuell` (für den aktuellen Knoten) auf diesen ersten Knoten zeigen lassen. (Abbildung 2.4)

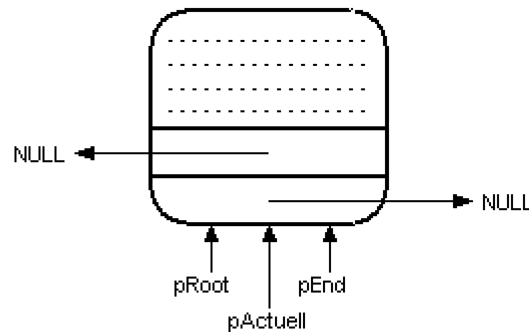

Abbildung 2.4: Ersten Knoten einfügen (Schritt 4)

b) FALL 2: Der neue Knoten befindet sich am Ende der Liste

Schritt 1:
Speicherplatz für den neuen Knoten anfordern. (wie im Fall 1; Abbildung 2.5).

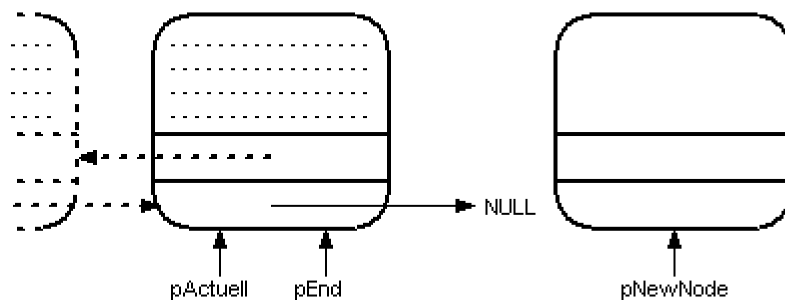

Abbildung 2.5: Knoten am Ende der Liste hinzufügen (Schritt 1)

Schritt 2:
Die zu speichernden Daten eingeben und im neuen Knoten (Zeiger `pNewNode`) speichern (wie im Fall 1; Abbildung 2.6).

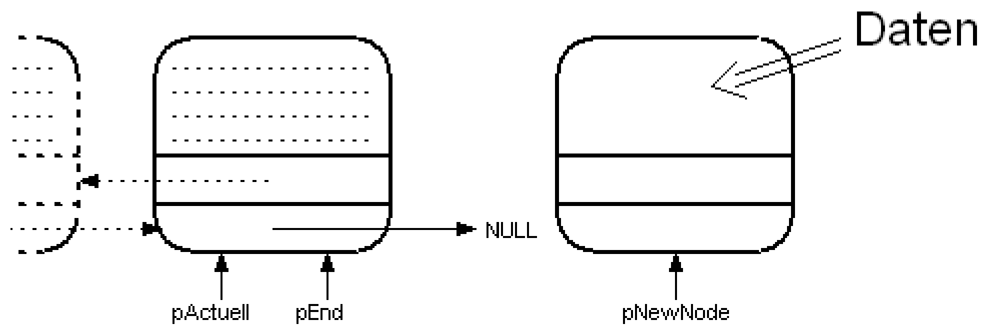

Abbildung 2.6: Knoten am Ende der Liste hinzufügen (Schritt 2)

Schritt 3:
Hier, beim letzten Knoten der Liste, den Zeiger auf den folgenden Knoten mit dem Inhalt 0 (`NULL`) laden. (Abbildung 2.7).

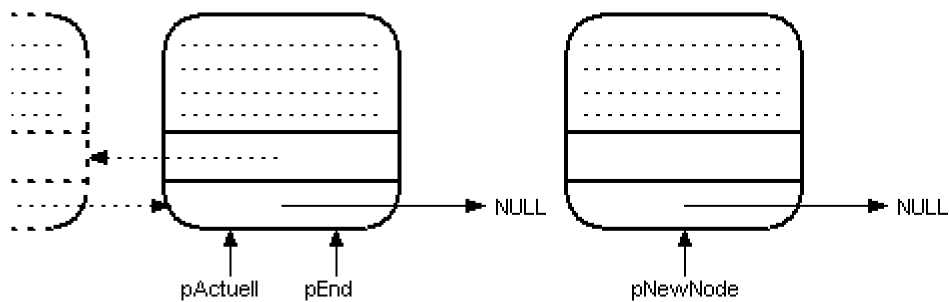

Abbildung 2.7: Knoten am Ende der Liste hinzufügen (Schritt 3)

Schritt 4:
Die Knoten auf denen die Zeiger `pActuell` und `pNewNode` zeigen miteinander verketten. (Abbildung 2.8).

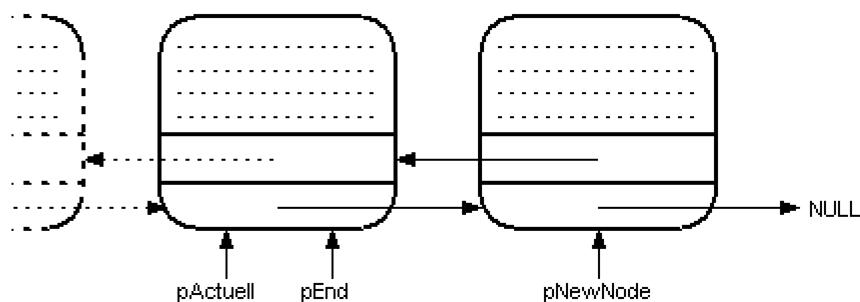

Abbildung 2.8: Knoten am Ende der Liste hinzufügen (Schritt 4)

Schritt 5:
Die Zeiger `pEnd` (für den letzten Knoten) und `pActuell` (für den aktuellen Knoten) auf diesen letzten Knoten zeigen lassen. (Abbildung 2.9).

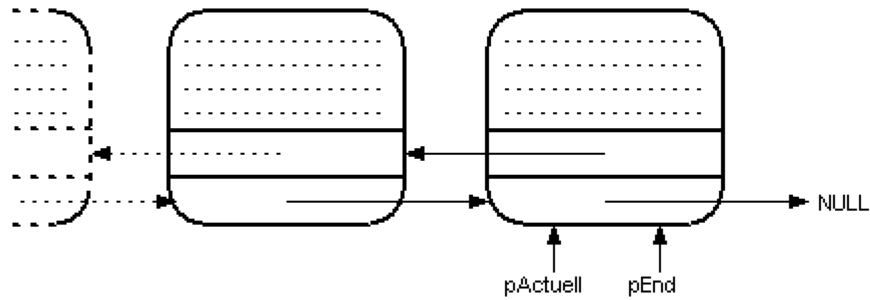

Abbildung 2.9: Knoten am Ende der Liste hinzufügen (Schritt 5)

c) FALL 3: Der neue Eintrag befindet sich nicht am Ende der Liste, sondern mittendrin

Schritt 1:
Speicherplatz für den neuen Knoten anfordern. (wie im Fall 1; Abbildung 2.10).

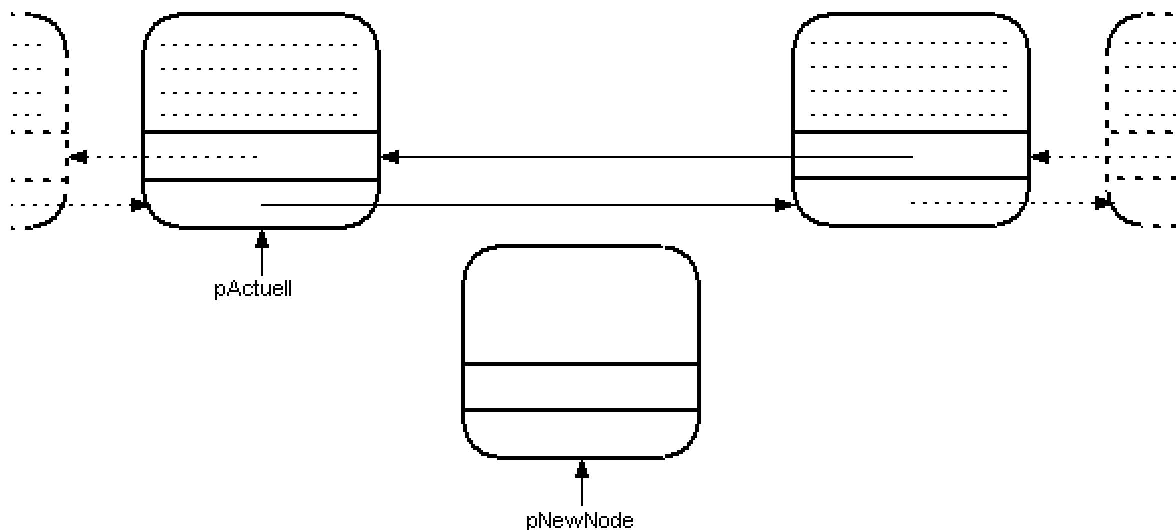

Abbildung 2.10: Knoten innerhalb der Liste hinzufügen (Schritt 1)

Schritt 2:
Die zu speichernden Daten eingeben und im neuen Knoten (Zeiger `pNewNode`) speichern (wie im Fall 1; Abbildung 2.11).

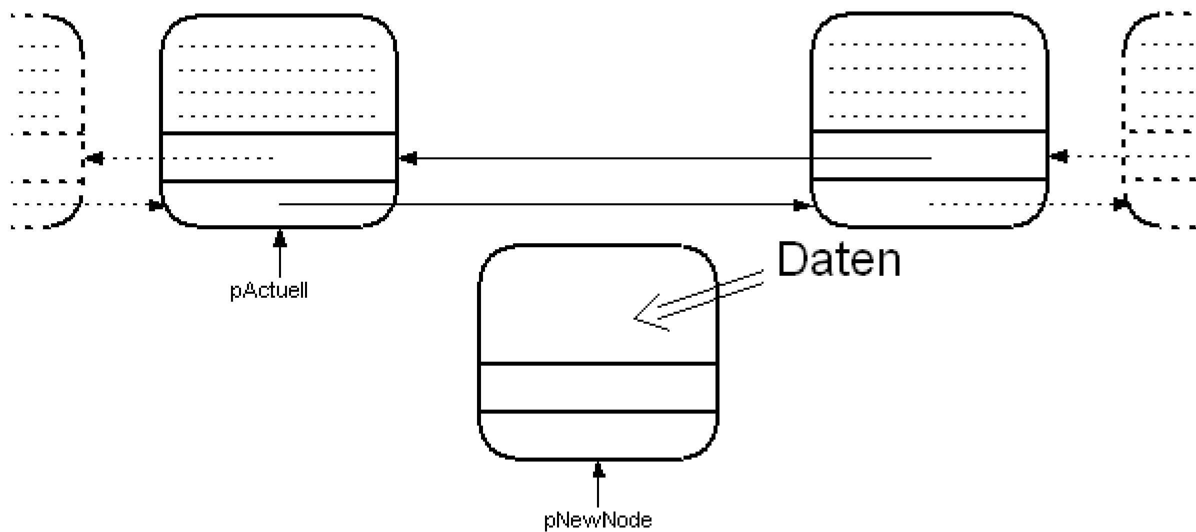

Abbildung 2.11: Knoten innerhalb der Liste hinzufügen (Schritt 2)

Schritt 3:
Die Knoten auf denen die Zeiger `pNewNode`, `pActuell` und den auf `pActuell` folgenden miteinander verketten. (Abbildung 2.12).

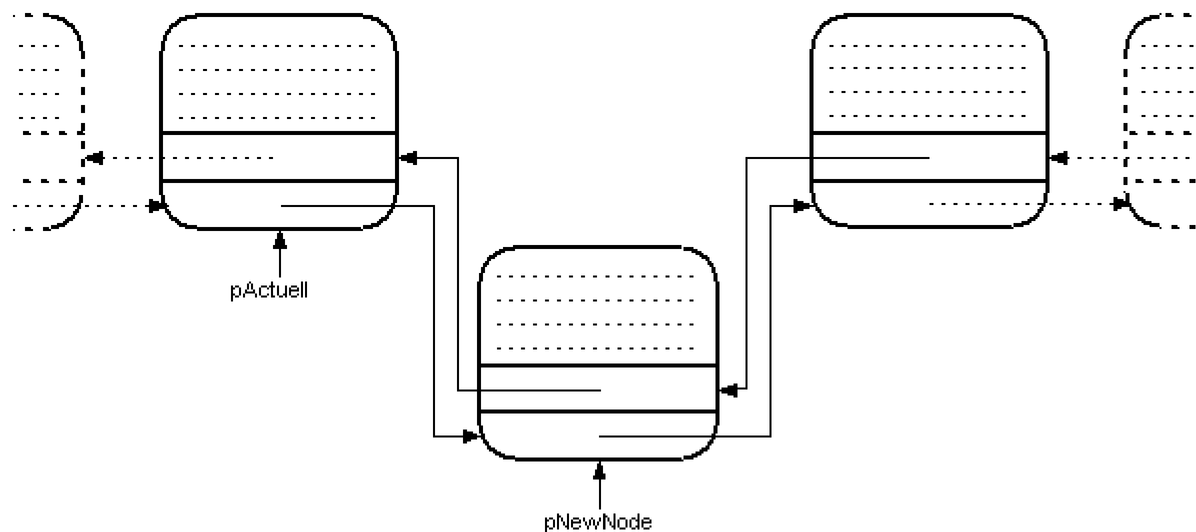

Abbildung 2.12: Knoten innerhalb der Liste hinzufügen (Schritt 3)

Schritt 4:
Den Zeiger `pActuell` (für den aktuellen Knoten) auf diesen neuen Knoten zeigen lassen. (Abbildung 2.13).

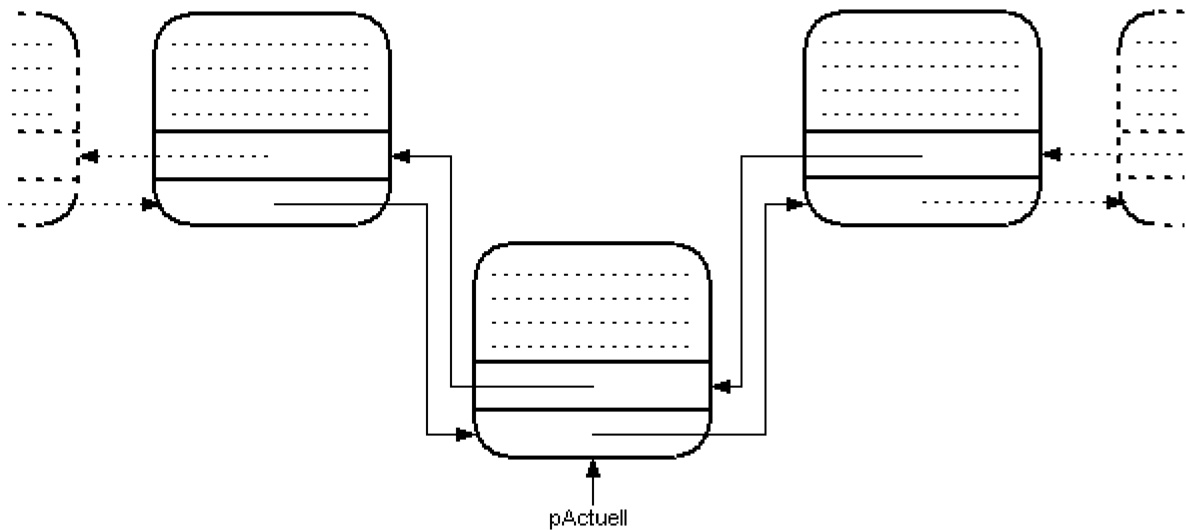

Abbildung 2.13: Knoten innerhalb der Liste hinzufügen (Schritt 4)

2.2 Daten eines Knoten ändern (editieren)

Das ändern (editieren) der Daten die in einem Knoten abgelegt sind ist besonders einfach. Es müssen einfach nur die Daten auf denen der Zeiger `pActuell` zeigt mit den "neuen" Werten (Daten) überschrieben werden.

Wichtig: Dieser Knoten muss natürlich "existieren", also im Speicher vorhanden sein. Dies erkennt man dadurch dass der Zeiger `pActuell` einen Wert besitzt der größer als Null ist.

2.3 Einen Knoten löschen

Beim Löschen eines Knotens muss zwischen 4 Fällen unterschieden werden:

- Fall 1: Die Liste besteht nur aus **einem einzigen** Knoten. Dieser Knoten soll nun gelöscht werden. (Kennzeichen: Der Zeiger der auf den nächsten Knoten zeigt ist leer, und auch der Zeiger der auf den vorhergehenden Knoten zeigt ist leer. Beide Zeiger beinhalten also den Wert 0 bzw. `NULL`)
- Fall 2: Die Liste besteht aus **mehreren** Knoten, davon wird der erste Knoten gelöscht. (Kennzeichen: Nur der Zeiger der auf den vorhergehenden Knoten zeigt ist leer, beinhaltet also den Wert 0 bzw. `NULL`)
- Fall 3: Die Liste besteht aus **mehreren** Knoten, davon wird der letzte Knoten gelöscht. (Kennzeichen: Der Zeiger der auf den nächsten Knoten zeigt ist leer, beinhaltet also den Wert 0 bzw. `NULL`)
- Fall 4: Die Liste besteht aus **mehreren** Knoten, davon wird weder der erste Knoten noch der letzte Knoten gelöscht. Sondern ein Knoten irgendwo zwischen dem Ersten und dem Letzten. (Kennzeichen: Der Zeiger der auf den vorhergehenden Knoten

zeigt und auch der Zeiger der auf den nächsten Knoten zeigt ist nicht leer. Beide Zeiger beinhalten also einen Wert ungleich 0 bzw. `NULL`.)

a) FALL 1: Die Liste besteht nur aus einem einzigen Knoten. Dieser Knoten soll nun gelöscht werden

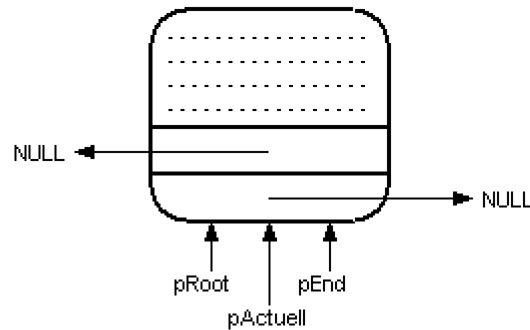

Abbildung 2.14: Den einzigen Knoten löschen

Schritt 1:
Den Speicherplatz auf den der Zeiger `pActuell` zeigt freigeben. Dadurch wird dieser Knoten aus dem Speicher gelöscht.

Schritt 2:
Die Zeiger `pRoot` (für den ersten Knoten), `pEnd` (für den letzten Knoten) und `pActuell` (für den aktuellen Knoten) mit dem 0 bzw. `NULL` laden, da ja nun kein Knoten mehr vorhanden ist. Die Liste ist somit wieder leer.

b) FALL 2: Die Liste besteht aus mehreren Knoten, davon wird der erste Knoten gelöscht

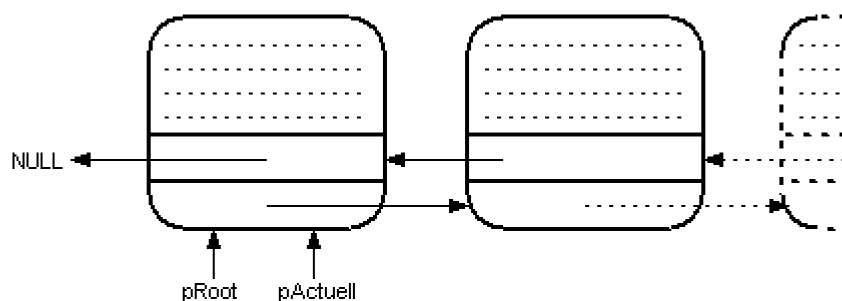

Abbildung 2.15: Den ersten Knoten löschen

Schritt 1:
Den Zeiger `pActuell` auf den auf `pActuell` folgenden Knoten setzen (Abbildung 2.16).

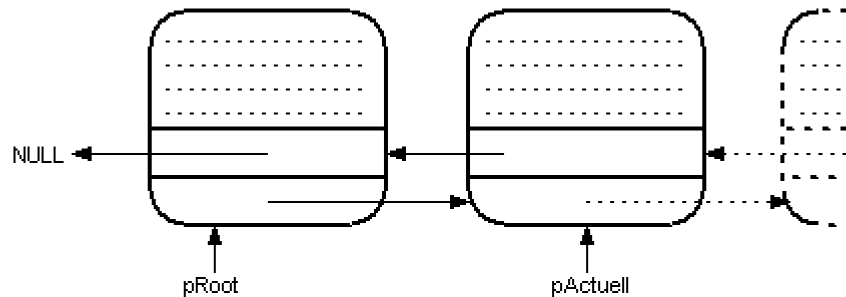

Abbildung 2.16: Den ersten Knoten löschen (Schritt 1)

Schritt 2:
Den Speicherplatz für den ersten Knoten freigeben (z.B. mit Hilfe des Zeigers `pRoot`). Dadurch wird dieser (erste) Knoten aus dem Speicher gelöscht (Abbildung 2.17).

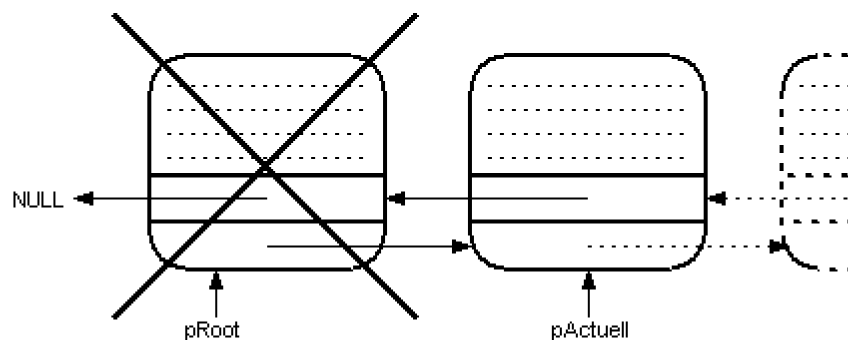

Abbildung 2.17: Den ersten Knoten löschen (Schritt 2)

Schritt 3:
Beim nun aktuellen Knoten den Zeiger, der auf den vorhergehenden Knoten zeigt mit dem Inhalt 0 bzw. `NULL` laden. Da es ja keinen vorhergehenden Knoten mehr gibt. (Abbildung 2.18).

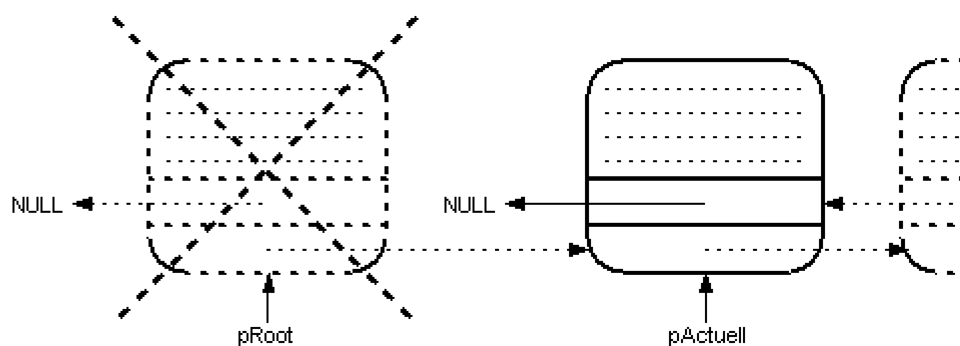

Abbildung 2.18: Den ersten Knoten löschen (Schritt 3)

Schritt 4:
Den Zeiger `pRoot` (für den ersten Knoten) auf den nun aktuellen Knoten setzen. (Abbildung 2.19).

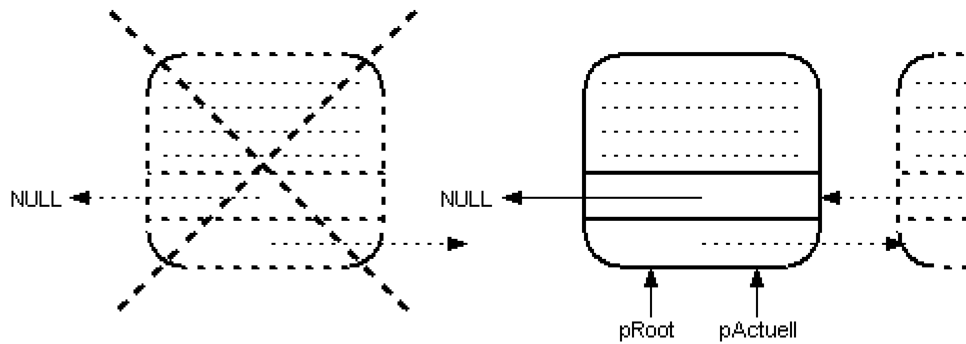

Abbildung 2.19: Den ersten Knoten löschen (Schritt 4)

c) FALL 3: Die Liste besteht aus mehreren Knoten, davon wird der letzte Knoten gelöscht

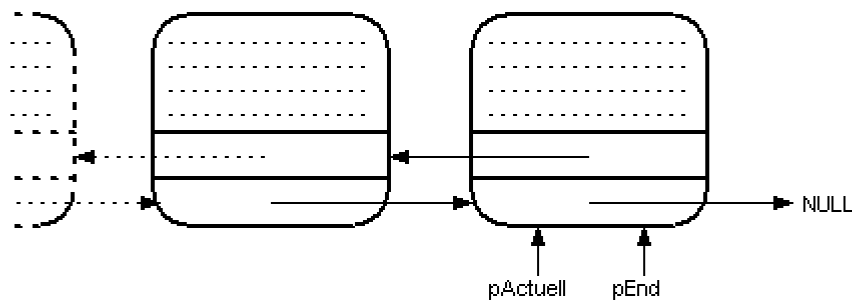

Abbildung 2.20: Den letzten Knoten löschen

Schritt 1:
Den Zeiger `pActuell` auf den von `pActuell` vorhergehenden Knoten setzen (Abbildung 2.21).

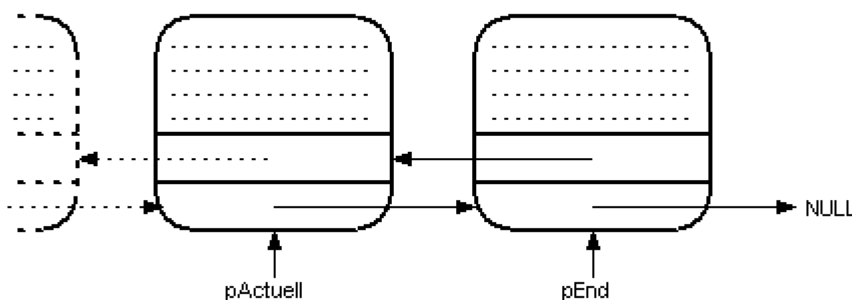

Abbildung 2.21: Den letzten Knoten löschen (Schritt 1)

Schritt 2:
Den Speicherplatz für den letzten Knoten freigeben (z.B. mit Hilfe des Zeigers `pEnd`). Dadurch wird dieser (letzte) Knoten aus dem Speicher gelöscht (Abbildung 2.22).

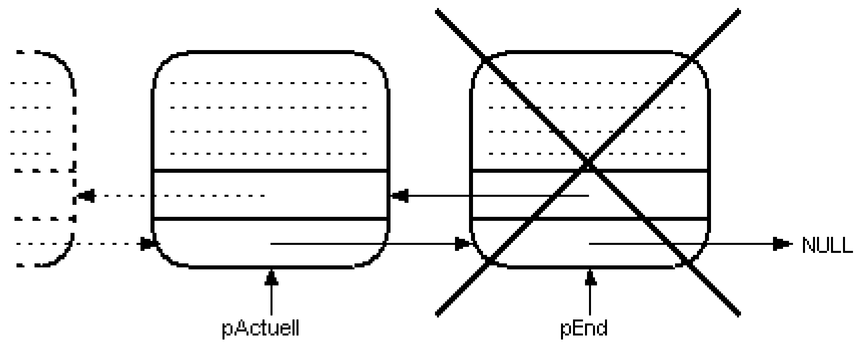

Abbildung 2.22: Den letzten Knoten löschen (Schritt 2)

Schritt 3:
Beim nun aktuellen Knoten den Zeiger, der auf den nachfolgenden Knoten zeigt mit dem Inhalt 0 bzw. `NULL` laden. Da es ja keinen nachfolgenden Knoten mehr gibt. (Abbildung 2.23).

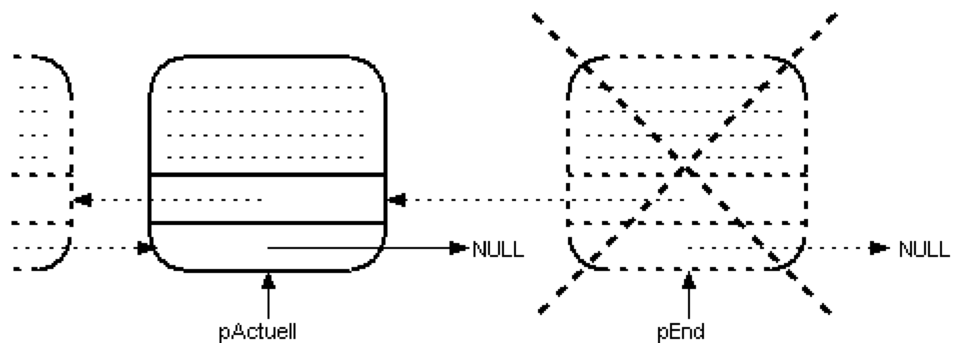

Abbildung 2.23: Den letzten Knoten löschen (Schritt 3)

Schritt 4:
Den Zeiger `pEnd` (für den letzten Knoten) auf den nun aktuellen Knoten setzen. (Abbildung 2.24).

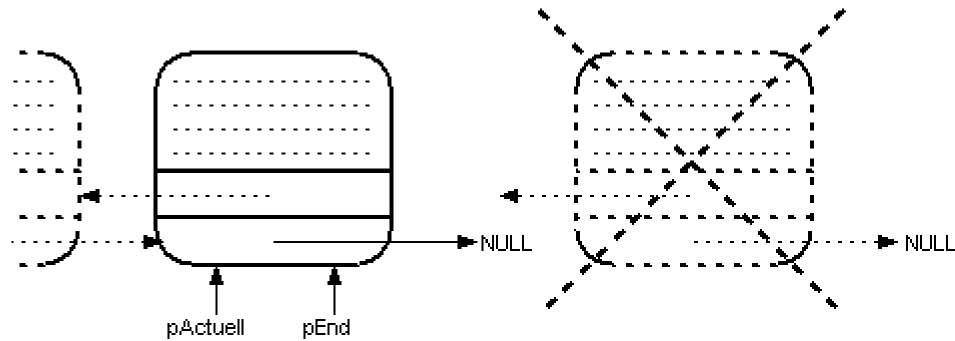

Abbildung 2.24: Den letzten Knoten löschen (Schritt 4)

d) FALL 4: Die Liste besteht aus mehreren Knoten, davon wird weder der erste Knoten noch der letzte Knoten gelöscht. Sondern ein Knoten irgendwo zwischen Ersten und Letztem

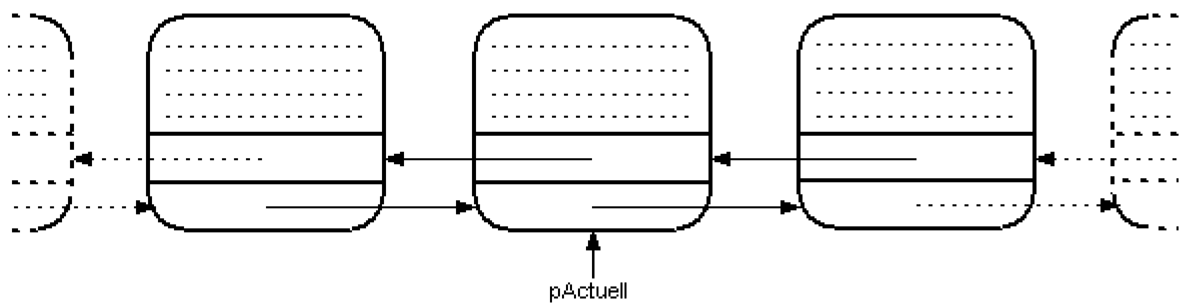

Abbildung 2.25: Einen Knoten mittendrin löschen

Schritt 1:
Die Adresse des auf den Zeiger `pActuell` folgenden Knoten in `pTempNach` sichern, und den zu `pActuell` vorherigen Knoten in `pTempVor` sichern (Abbildung 2.26).

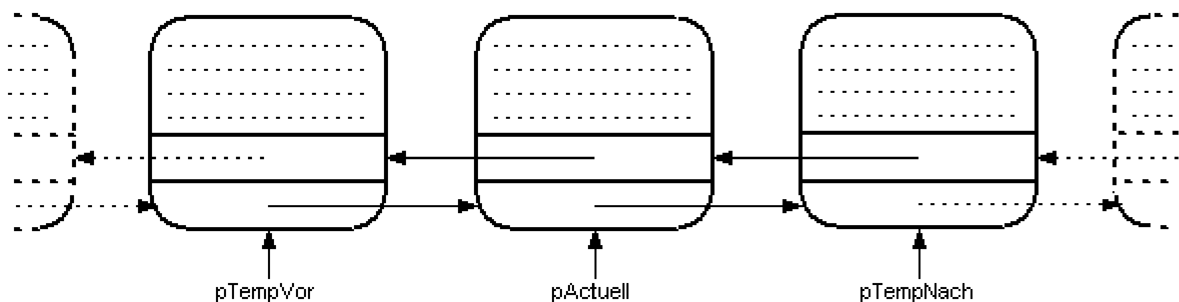

Abbildung 2.26: Einen Knoten mittendrin löschen (Schritt 1)

Schritt 2:
Den Knoten auf den der Zeiger `pActuell` zeigt entfernen (löschen) und den Speicherplatz für diesen Knoten wieder freigeben (Abbildung 2.27).

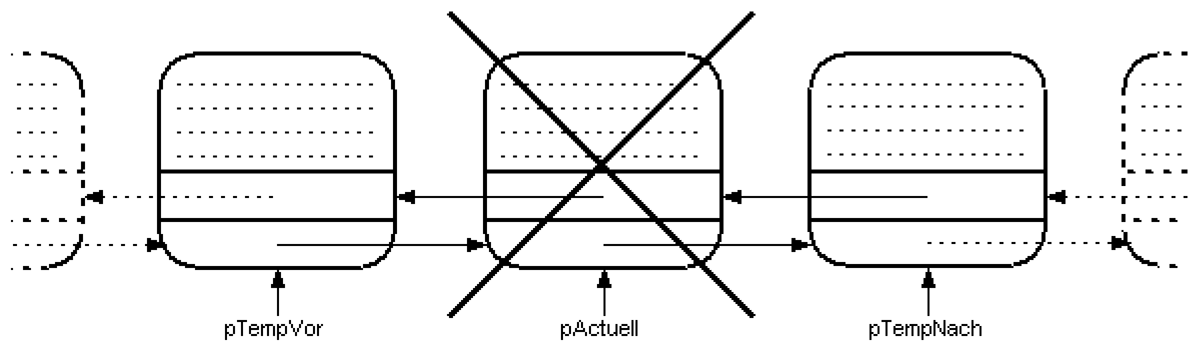

Abbildung 2.27: Einen Knoten mittendrin löschen (Schritt 2)

Schritt 3:
Die beiden Knoten auf denen die Zeiger `pTempVor` und `pTempNach` zeigen miteinander verketten (Abbildung 2.28).

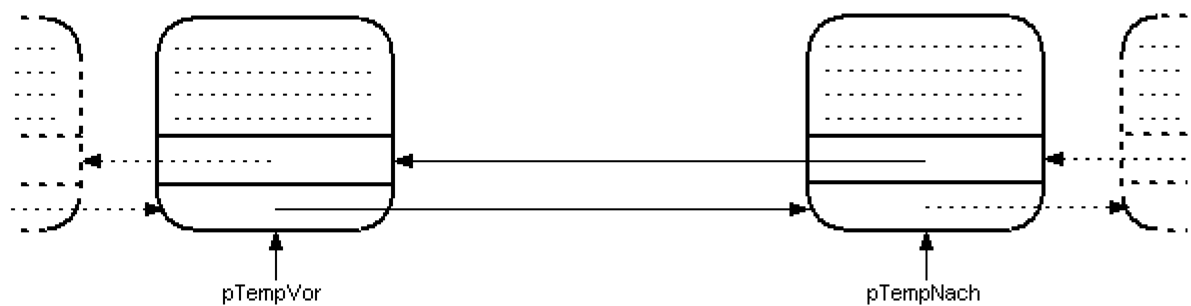

Abbildung 2.28: Einen Knoten mittendrin löschen (Schritt 3)

Schritt 4:
Den Zeiger `pActuell` (für den aktuellen Knoten) auf den Knoten setzen auf den der Zeiger `pTempVor` zeigt. Dieser Knoten ist nun der "neue" aktuelle Knoten (Abbildung 2.29).

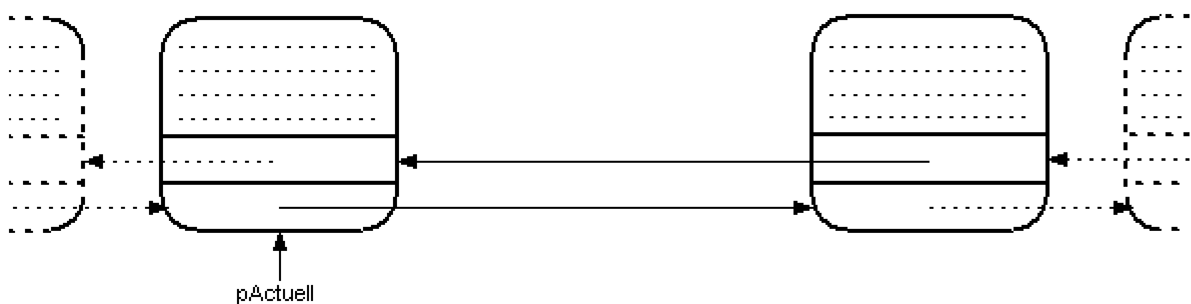

Abbildung 2.29: Einen Knoten mittendrin löschen (Schritt 4)

2.4 Gesamte Liste löschen

Beim Löschen der gesamten Liste ist darauf zu achten das jeder Knoten für sich gelöscht werden muss, und dass anschließend die Zeiger `pRoot` (für den ersten Knoten), `pEnd` (für den letzten Knoten) und `pActuell` (für den aktuellen Knoten) mit dem Wert 0 bzw. `NULL` geladen werden müssen, da ja nun kein Knoten mehr vorhanden ist.

Als Vorgehensweise empfiehlt sich folgende Methode: Mit Hilfe zweier Hilfszeiger (z.B. `pDeleteNode` und `pDeleteNextNode`) beginnend beim ersten Knoten diesen mit Hilfe einer Schleife Knoten für Knoten vom Speicher entfernen. Dabei zeigt der Zeiger `pDeleteNode` immer auf den zu entfernenden Knoten und der Zeiger `pDeleteNextNode` auf den darauf folgenden Knoten. Die Abbruchbediengung für die Schleife ist wenn der Zeiger `pDeleteNextNode` ein "Null-Zeiger" ist, d.h. der Zeiger den Wert Null beinhaltet. Denn dann ist er am Ende der Liste angekommen.
Abbildung 2.30 zeigt diesen Vorgang in Form eines Flussdiagramms.

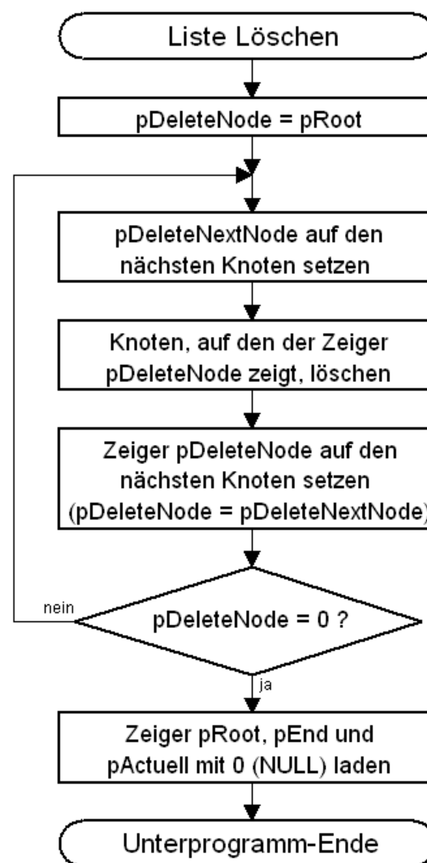

Abbildung 2.30: Flussdiagramm (Löschen der gesamten Liste)

2.5 In der Liste blättern (scrollen)

Unter "scrollen" versteht man das sich bewegen innerhalb der Liste. Also vom aktuellen Knoten zu dessen vorhergehenden Knoten, oder zum nachfolgenden Knoten.
Oft soll es auch möglich sein direkt zum ersten Knoten zu gelangen, und natürlich auch zum letzten Knoten.

Das "scrollen" ist notwendig um dem Benutzer die gewünschten Daten (in den Knoten) sichtbar z.B. am Bildschirm anzeigen zu können.

Bei einer doppelt verketteten Liste ist die Realisierung des "scrollen" sehr einfach. Aufgrund der Verkettung zeigt ja ein Zeiger immer auf den nachfolgenden Knoten und bei der doppelt verketteten Liste ein Zeiger auf den vorhergehenden Knoten. Weiters gibt es noch den Zeiger der auf den ersten Knoten der Liste zeigt (Zeiger `pRoot`) und in vielen Fällen (so wie in dieser Dokumentation) auch einen Zeiger auf den letzten Knoten der Liste (Zeiger `pEnd`).
Es ist daher nur notwendig den Zeiger, der auf den aktuellen Knoten zeigt (hier der Zeiger `pActuell`), mit dem je nach "Scrollart" (also einen Knoten nach vor oder zurück, zum Beginn der Liste, oder ans Ende der Liste) erforderlichen Zeiger zu laden. Tabelle 2.1 zeigt wie der Zeiger `pActuell` geladen werden muss.

Scrollart	**Symbol**	**Zeigerzuweisung**
Zum Beginn der Liste	$\ll$	`pActuell = pRoot`
Zum vorhergehenden Knoten	$<$	`pActuell = pActuel.pLast`
Zum folgenden Knoten	$>$	`pActuell = pActuel.pNext`
Zum Ende der Liste	$\gg$	`pActuell = pEnd`

Tabelle 2.1: Zeigerzuweisung beim Scrollen

Kapitel 3

Realisierung einer doppelt verketteten Liste in C++

Da die verkettete Liste eine sehr einfache und grundlegende Datenstruktur ist, gibt es auch sehr viele unterschiedliche Realisierungmöglichkeiten. Diese sind natürlich auch von der verwendeten Programmiersprache abhängig. Da sie so grundlegend ist kommt sie auch in sehr vielen Büchern vor, und auch im Internet gibt es unzählige Seiten zu diesem Thema.

Meine Realisierung ist daher nur eine von vielen Möglichkeiten.

In C++ ist es zweckmäßig die gesamte Liste mit Hilfe von zwei Klassen zu implementieren (Abbildung 3.1):

- Klasse `CNode`: Diese bildet nur den eigentlichen Knoten mit den zu speichernden Daten ab.
- Klasse `CLinkedList`: Diese beinhaltet die gesamte Liste, so wie sie in Kapitel 2 erläutert wurde.

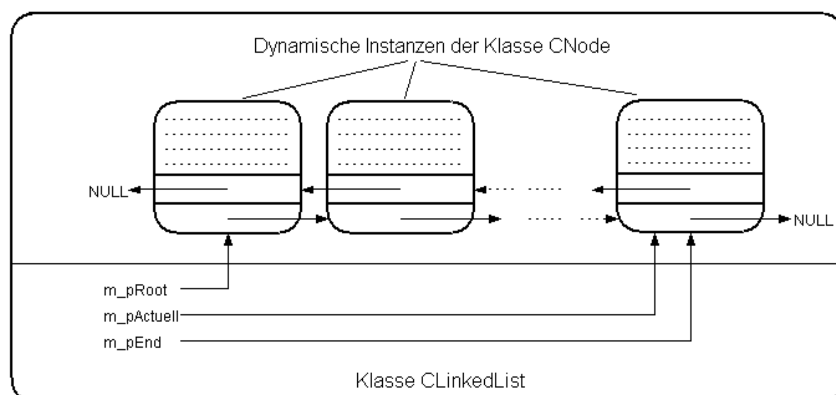

Abbildung 3.1: Realisierung einer doppelt verketteten Liste in C++

Anmerkung: Dies hier ist kein C++-Kurs. Ich beschreibe hier nur kurz die wesentlichen Punkte für eine mögliche Realisierung der doppelt verketteten Listen in C++.

Ich gehe davon aus, dass der/die Leser(in) die Grundlagen von C++ kennt und diese auch beherrscht. Begriffe wie Klasse, Konstruktor, Destruktor, Member-Variablen usw. sind ihnen also bekannt. Wenn nicht so empfehle ich das Buch *C++ in 21 Tagen* [2].

C++ stellt keine speziellen Befehle und Unterprogramme dafür zur Verfügung. Es muss also alles selbst programmiert werden!

3.1 Klasse `CNode`

In C++ lässt sich ein einzelner Knoten sehr gut durch eine Klasse beschreiben. Diese Klasse kann z.B. wie folgt deklariert werden (Datei `CNode.h`, Listing 3.1).

Anmerkung zu den Zeilennummern: Die Zeilennummern sind **nicht** Bestandteil des Quellcodes. Sie dienen hier nur der besseren Orientierung. In vielen Entwicklungsumgebungen können diese so eingestellt werden, dass sie automatisch vom Editor hinzugefügt werden.

Listing 3.1: CNode.h (Klassendefinition)

```
1  /*******************************************************************************/
2  /* Header zur Klasse CNode     (CNode.h)                                      */
3  /*                                                                             */
4  /* Kurzbeschreibung                                                            */
5  /*  Klasse fuer einen Knoten einer verketteten Liste (engl. linked list)       */
6  /*                                                                             */
7  /* Aenderungen/Ergaenzungen                                                    */
8  /*                                                                             */
9  /* Entwickler: Buchgeher Stefan                                                */
10 /* Entwicklungsbeginn dieser Klasse: 10. Oktober 2006                          */
11 /* Funktionsfaehig seit: 10. Oktober 2006                                      */
12 /* Letzte Bearbeitung: 24. Januar 2007                                         */
13 /*******************************************************************************/
14
15 #ifndef __CNode
16 #define __CNode
17
18
19 /***************** Include-Dateien ***********************************************/
20 // keine Include-Dateien notwendig
21
22
23 /***************** Konstanten ****************************************************/
24 // keine Konstanten
25
26
27 /***************** Klassendefinition *********************************************/
28 class CNode
29 {
30 public:
31     // Konstruktoren
32     CNode();
33     CNode(int nValue);
34
35     // Kopierkonstruktor
36     CNode (const CNode &rhs);
37
38     // Destruktor
39     virtual ~CNode();
40
41     // oeffentliche Member-Funktionen
42     void SetValue(int nValue) {m_nValue = nValue;}
43     void SetLast(CNode *pLast) {m_pLast = pLast;}
44     void SetNext(CNode *pNext) {m_pNext = pNext;}
```

```
45
46      int GetValue() const {return m_nValue;}
47      CNode* GetLast() const {return m_pLast;}
48      CNode* GetNext() const {return m_pNext;}
49
50
51  protected:
52      // geschuetzte Member-Variablen
53      int m_nValue;               // den zu speichernden Wert
54      CNode *m_pLast;             // Zeiger auf den vorhergehenden Knoten
55      CNode *m_pNext;             // Zeiger auf den folgenden Knoten
56  };
57
58  #endif // __CNode
```

Da diese Klasse nur einen Knoten beinhaltet, ist sie auch sehr einfach. Hier beinhaltet dieser Knoten als Daten nur eine Integer-Variable (hier `m_nValue`, Zeile 53) in Form einer Member-Variable und die Zeiger zum vorhergehenden und folgenden Knoten (`m_pLast` und `m_pNext`, Zeilen 53 und 54).

Da die Funktionen zum Setzen und Auslesen dieser Membervariablen sehr einfach sind wurden diese gleich in der Header-Datei ausgeführt, Zeilen 42 bis 48.
Die Realisierung der restlichen Funktionen (hier eigentlich nur die Konstruktoren) erfolgt in der Datei `CNode.cpp`, Listing 3.2. Auch diese sind sehr einfach, da nur die Membervariablen initialisiert werden müssen.

Listing 3.2: CNode.cpp

```
1   /*****************************************************************************/
2   /* Klasse CNode       (CNode.cpp)                                             */
3   /*                                                                            */
4   /* Kurzbeschreibung                                                           */
5   /*  Klasse fuer einen Knoten einer verketteten Liste (engl. linked list)      */
6   /*                                                                            */
7   /* Aenderungen/Ergaenzungen                                                   */
8   /*                                                                            */
9   /* Entwickler: Buchgeher Stefan                                               */
10  /* Entwicklungsbeginn dieser Klasse: 10. Oktober 2006                         */
11  /* Funktionsfaehig seit: 10. Oktober 2006                                     */
12  /* Letzte Bearbeitung: 24. Januar 2007                                        */
13  /*****************************************************************************/
14
15  /***************** Include-Dateien *******************************************/
16  #include "CNode.h"
17
18
19  /***************** Konstruktoren *********************************************/
20
21  /*****************************************************************************/
22  /* CNode (Default-Konstruktor):                                               */
23  /*                                                                            */
24  /* Aufgaben:                                                                  */
25  /*  - Alle Member-Variablen und -Zeiger mit 0 initialisieren                  */
26  /*                                                                            */
27  /* Veraenderte Member-Variablen:                                              */
28  /*  - m_nValue                                                                */
29  /*  - m_pLast                                                                 */
30  /*  - m_pNext                                                                 */
31  /*****************************************************************************/
32  CNode::CNode()
33  {
34      m_nValue = 0;
35      m_pLast = 0;
36      m_pNext = 0;
```

```
37 }
38
39
40 /*****************************************************************************/
41 /* CNode(int nValue):                                                        */
42 /*                                                                           */
43 /* Aufgaben:                                                                 */
44 /*  - Die Member-Variable m_nValue mit dem uebergebenen Parameter initialisieren, und */
45 /*    die Zeiger m_pLast und m_pNext mit 0 initialisieren                    */
46 /*                                                                           */
47 /* Uebergabeparameter:                                                       */
48 /*  - nValue (Die zu speichernde Zahl)                                       */
49 /*                                                                           */
50 /* Veraenderte Member-Variablen:                                             */
51 /*  - m_nValue                                                               */
52 /*  - m_pLast                                                                */
53 /*  - m_pNext                                                                */
54 /*****************************************************************************/
55 CNode::CNode(int nValue)
56 {
57     m_nValue = nValue;
58     m_pLast = 0;
59     m_pNext = 0;
60 }
61
62
63 /*****************************************************************************/
64 /* CNode (const CNode &rhs):     (Kopierkonstruktor)                         */
65 /*                                                                           */
66 /* Aufgabe:                                                                  */
67 /*  - Kopierkonstruktor                                                      */
68 /*                                                                           */
69 /* Veraenderte Member-Variablen:                                             */
70 /*  keine                                                                    */
71 /*****************************************************************************/
72 CNode::CNode (const CNode &rhs)
73 {
74     m_nValue = rhs.m_nValue;
75     m_pLast = rhs.m_pLast;
76     m_pNext = rhs.m_pNext;
77 }
78
79
80 /***************** Destruktor ************************************************/
81
82 /*****************************************************************************/
83 /* ~CNode:                                                                   */
84 /*                                                                           */
85 /* Aufgabe:                                                                  */
86 /*  Hat hier keine Aufgabe                                                   */
87 /*                                                                           */
88 /* Veraenderte Member-Variablen:                                             */
89 /*  keine                                                                    */
90 /*****************************************************************************/
91 CNode::~CNode()
92 {
93 }
```

3.2 Klasse `CLinkedList`

Die KLasse `CLinkedList` ist schon etwas umfangreicher. Diese Klasse realisiert die verkettete Liste, so wie sie in Kapitel 2 ab Seite 5 beschrieben wurde, daher ist eine ausführliche Beschreibung hier nicht mehr notwendig. Auch hier dient wieder eine Header-Datei zur Deklaration (`CLinkedList.h`, Listing 3.3) und eine Datei, welche den Quellcode beinhaltet (`CLinkedList.cpp`, Listing 3.4).

Listing 3.3: CLinkedList.h (Klassendefinition)

```
1  /*****************************************************************************/
2  /* Header zur Klasse CLinkedList      (CLinkedList.h)                        */
3  /*                                                                           */
4  /* Kurzbeschreibung                                                          */
5  /*  Klasse fuer eine doppelt verkettete Liste (engl. linked list)            */
6  /*                                                                           */
7  /* Aenderungen/Ergaenzungen                                                  */
8  /*                                                                           */
9  /* Entwickler: Buchgeher Stefan                                              */
10 /* Entwicklungsbeginn dieser Klasse: 10. Oktober 2006                        */
11 /* Funktionsfaehig seit: 10. Oktober 2006                                    */
12 /* Letzte Bearbeitung: 24. Januar 2007                                       */
13 /*****************************************************************************/
14
15 #ifndef __CLinkedList
16 #define __CLinkedList
17
18
19 /***************** Include-Dateien ******************************************/
20 #include "CNode.h"
21
22
23 /***************** Konstanten ***********************************************/
24 enum SCROLL_DIRECTION
25 {
26     SCROLL_BACK,
27     SCROLL_FORWARD,
28     SCROLL_BEGIN,
29     SCROLL_END
30 };
31
32
33 /***************** Klassendefinition ****************************************/
34 class CLinkedList
35 {
36 public:
37     // Konstruktoren
38     CLinkedList();
39     CLinkedList(int nValue);
40
41     // Kopierkonstruktor
42 //    CLinkedList (const CLinkedList &rhs);
43
44
45     // Destruktor
46     virtual ~CLinkedList();
47
48
49     // oeffentliche Member-Funktionen
50     int AddNode(int nValue);
51
52     int ShowList(void);
53
54     int RemoveNode(void);
55     int DeleteList(void);
56
57     int Scroll(int nDirection);
```

```
58
59      //CLinkedList::operator=( const CLinkedList &rhs);
60
61      // ACHTUNG: noch nicht realisiert , da beim Demo keine Zuweisung zwischen zwei Listen
62      // vorkommt! Ist aber bei einer Zuweisung in einer Applikation (z.B. Liste2 = Liste1)
63      // absolut notwendig und muss daher geeignet realisiert werden!!!
64
65
66  protected:
67      // geschuetzte Member-Variablen
68      CNode *m_pRoot;              // Zeiger auf den ersten Knoten der Liste
69      CNode *m_pEnd;               // Zeiger auf den letzten Knoten der Liste
70      CNode *m_pActuell;           // Zeiger auf den aktuell "bearbeiteten" Knoten der Liste
71  };
72
73  #endif // __CLinkedList
```

Da diese Klasse auch die `CNode`-Klasse benötigt muss dies hinzugefügt werden (siehe Zeile 20).

Das Scrollen, also das Blättern innerhalb der Liste, wurde mit nur einer Memberfunktion realisiert (Zeile 57). Da es aber 4 Scrollmöglichkeiten gibt, ist ein Übergabeparameter der die Scrollart bestimmt notwendig. Dies wird hier mit einer so genannten Aufzählungsvariable realisiert (Zeilen 24 bis 30).

Das Hinzufügen eines neuen Knotens, das Anzeigen der gesamten Liste und das Löschen eines Knotens oder der gesamten Liste ist mit je einer eigenen Memberfunktion realisiert (Zeilen 50 bis 55).
Da bei diesen Funktionen während der Laufzeit Fehler entstehen können, z.B. wenn kein neuer Knoten mehr hinzugefügt werden kann, oder wenn versucht wird eine leere Liste zu löschen, dann sollte das aufrufende Programm über diesen Fehler informiert werden. Daher ist es gängige Praxis, Funktionen, die einen Fehler verursachen können mit einem Rückgabewert zu versehen. Dieser Rückgabewert gibt dann den Fehler in Form einer Zahl (Fehlercode) an das aufrufende Programm zurück. Tritt während der Ausführung dieser Funktion kein Fehler auf, so ist es üblich den Wert 0 zurückzugeben. Dieser Fehlermechanismus muss natürlich selbst ausprogrammiert werden, und ist nur eine von mehreren Möglichkeiten zur Fehlererkennung. Hier wurde diese einfache Methode für sämtliche Memberfunktionen gewählt (Zeilen 50 bis 57).

Listing 3.4: CLinkedList.cpp

```
 1  /*****************************************************************************/
 2  /* Klasse CLinkedList      (CLinkedList.cpp)                                 */
 3  /*                                                                           */
 4  /* Kurzbeschreibung                                                          */
 5  /*  Klasse fuer eine doppelt verkettete Liste (engl. linked list)            */
 6  /*                                                                           */
 7  /* Aenderungen/Ergaenzungen                                                  */
 8  /*                                                                           */
 9  /* Entwickler: Buchgeher Stefan                                              */
10  /* Entwicklungsbeginn dieser Klasse: 10. Oktober 2006                        */
11  /* Funktionsfaehig seit: 11. Oktober 2006                                    */
12  /* Letzte Bearbeitung: 24. Januar 2007                                       */
13  /*****************************************************************************/
14
15  /***************** Include-Dateien *****************************************/
16  #include "CLinkedList.h"
17  #include "CNode.h"
```

```
18 #include <iostream.h>
19
20
21 /***************** Konstruktoren ****************************************************/
22
23 /**********************************************************************************/
24 /* CLinkedList (Default-Konstruktor):                                             */
25 /*                                                                                */
26 /* Aufgaben:                                                                      */
27 /*  - Die Member-Variablen mit 0 initialisieren                                   */
28 /*                                                                                */
29 /* Veraenderte Member-Variablen:                                                  */
30 /*  - m_pRoot                                                                     */
31 /*  - m_pEnd                                                                      */
32 /*  - m_pActuell                                                                  */
33 /**********************************************************************************/
34 CLinkedList::CLinkedList()
35 {
36     m_pRoot = 0;
37     m_pEnd = 0;
38     m_pActuell = 0;
39 }
40
41
42 /**********************************************************************************/
43 /* CLinkedList(int nValue):                                                       */
44 /*                                                                                */
45 /* Aufgaben:                                                                      */
46 /*  - Zunaechst alle Member-Variablen initialisieren und                          */
47 /*  - Den ersten Knoten erzeugen und darin den uebergebenen Wert sichern          */
48 /*                                                                                */
49 /* Uebergabeparameter:                                                            */
50 /*  - nValue (Die zu speichernde Zahl)                                            */
51 /*                                                                                */
52 /* Veraenderte Member-Variablen:                                                  */
53 /*  - m_pRoot                                                                     */
54 /*  - m_pEnd                                                                      */
55 /*  - m_pActuell                                                                  */
56 /**********************************************************************************/
57 CLinkedList::CLinkedList(int nValue)
58 {
59     m_pRoot = 0;
60     m_pEnd = 0;
61     m_pActuell = 0;
62
63     AddNode(nValue);
64 }
65
66
67 /**********************************************************************************/
68 /* CLinkedList (const CLinkedList &rhs):     (Kopierkonstruktor)                  */
69 /*                                                                                */
70 /* Aufgabe:                                                                       */
71 /*  - Kopierkonstruktor                                                           */
72 /*                                                                                */
73 /* Veraenderte Member-Variablen:                                                  */
74 /*  keine                                                                         */
75 /**********************************************************************************/
76 //CLinkedList::CLinkedList (const CLinkedList &rhs)
77 //{
78 //    noch nicht vorhanden
79 //}
80
81
82 /***************** Destruktor *******************************************************/
83
84 /**********************************************************************************/
85 /* ~CLinkedList:                                                                  */
86 /*                                                                                */
87 /* Aufgabe:                                                                       */
88 /*  Die gesamte Liste (falls vorhanden) vom Speicher entfernen.                   */
```

```
89  /*                                                                              */
90  /* Veraenderte Member-Variablen:                                                */
91  /*  - m_pRoot                                                                   */
92  /*  - m_pEnd                                                                    */
93  /*  - m_pActuell                                                                */
94  /********************************************************************************/
95  CLinkedList::~CLinkedList()
96  {
97      DeleteList();
98  }
99
100
101 /***************** Member-Funktionen ***********************************************/
102
103 /********************************************************************************/
104 /* int AddNode(nValue):                                                         */
105 /*                                                                              */
106 /* Aufgabe:                                                                     */
107 /*  Neuen Knoten (mit dem Wert nValue) an die bestehende Liste anhaengen.       */
108 /*                                                                              */
109 /* Uebergabeparameter:                                                          */
110 /*  nValue: Dieser Wert soll in den neu hinzugefuegten Knoten gespeichert werden. */
111 /*                                                                              */
112 /* Rueckgabewert:                                                               */
113 /*  - Fehlercode:   0 ... kein Fehler                                           */
114 /*                  1 ... Es kann kein Knoten hinzugefuegt werden, weil kein Speicher */
115 /*                        alloziiert werden kann!                               */
116 /*                                                                              */
117 /* Vorgehensweise:                                                              */
118 /*  - Speicherplatz fuer den neuen Knoten dynamisch anfordern. Der Zeiger pNewNode */
119 /*    zeigt dabei auf die Adresse im Arbeitsspeicher wo sich dieser neue Knoten be- */
120 /*    findet. Ist der Inhalt des Zeigers pNewNode aber 0 (NULL) so konnte kein  */
121 /*    Speicherplatz angefordert werden.                                         */
122 /*  - War das Speicheranfordern nicht erfolgreich (pNewNode = 0), den entsprechenden */
123 /*    Fehlercode dem aufrufenden Programm zurueckgeben.                         */
124 /*  - War das Speicheranfordern erfolgreich (pNewNode > 0):                     */
125 /*      - Die zu speichernden Daten im neuen Knoten (pNewNode) speichern (mit Hilfe */
126 /*        der Memberfunktion SetValue der Klasse CNode)                         */
127 /*      - Das Verketten der Knoten ist nun abhaengig davon, an welcher Position sich */
128 /*        der neue Knoten befindet:                                             */
129 /*                                                                              */
130 /*          - FALL 1: Der neue Knoten (pNewNode) befindet sich am Anfang der Liste, */
131 /*            es ist daher auch das erste Element der Liste                     */
132 /*            (Kennzeichen: m_pRoot = 0 (NULL)                                  */
133 /*            Da es das erste Element der Liste ist gilt fuer diesen Knoten:    */
134 /*              - Der Zeiger auf den vorhergehende Knoten muss den Wert 0 (NULL) */
135 /*                beinhalten (pNewNode->SetLast(0))                             */
136 /*              - Der Zeiger auf das naechste Element muss ebenfalls den Wert 0 (NULL) */
137 /*                beinhalten (pNewNode->SetNext(0))                             */
138 /*              - Die (Member-)Zeiger m_pRoot, m_pEnd und m_pActuell zeigen auf diesen */
139 /*                ersten Knoten.                                                */
140 /*                                                                              */
141 /*          - FALL 2: Der neue Knoten (pNewNode) befindet sich am Ende der Liste, */
142 /*            (Kennzeichen: m_pActuell->GetNext() == 0 (NULL)                   */
143 /*            Da es der letzte Knoten der Liste ist gilt fuer diesen Knoten:    */
144 /*              - Der Zeiger von pNewNode auf den naechsten Knoten muss den Wert 0 */
145 /*                (NULL) beinhalten (pNewNode->SetNext(0))                      */
146 /*              - Der Zeiger von pNewNode auf den vorhergehenden Knoten muss den */
147 /*                Zeiger pActuell beinhalten (pNewNode->SetLast(m_pActuell))    */
148 /*              - Der Zeiger von pActuell auf den naechste Knoten muss den Wert des */
149 /*                Zeigers pNewNode beinhalten (m_pActuell->SetNext(pNewNode))   */
150 /*              - Die (Member-)Zeiger m_pEnd und m_pActuell zeigen auf diesen letzten */
151 /*                Knoten.                                                       */
152 /*                                                                              */
153 /*          - FALL 3: Der neue Knoten befindet sich nicht am Ende der Liste, sondern */
154 /*            mittendrin                                                        */
155 /*            Da es ein Knoten innerhalb der Liste ist gilt fuer dieses Knoten: */
156 /*              - Den auf pActuell folgenden Knoten in pTemp sichern            */
157 /*              - Den neuen Knoten pNewNode nun so mit pActuell und pTemp verketten, */
158 /*                dass es sich anschliessend zwischen pActuell und pTemp befindet */
159 /*                  - Verkettung zwischen pActuell und pNewNode herstellen      */
```

```
160 /*                  - Verkettung zwischen pNewNode und pTemp herstellen              */
161 /*              - Der (Member-)Zeiger m_pActuell zeigt auf den eingefuegten Knoten     */
162 /*                                                                                     */
163 /* Veraenderte Member-Variablen:                                                       */
164 /*  - m_pRoot                                                                          */
165 /*  - m_pEnd                                                                           */
166 /*  - m_pActuell                                                                       */
167 /**************************************************************************************/
168 int CLinkedList::AddNode(int nValue)
169 {
170     CNode *pNewNode;
171     CNode *pTemp;        // nur fuer Fall 3 notwendig
172 
173 
174     pNewNode = new CNode();
175 
176     if (pNewNode)
177     {
178         // Ist der Ruechgabewert pNewNode groesser als 0, so war die Anforderung
179         // des Speicherplatz erfolgreich. pNewNode beinhaltet nun die Startadresse
180         // des neuen Knotens.
181         pNewNode->SetValue(nValue);
182 
183         if (m_pRoot == 0)
184         {
185             // Fall 1: Erster Knoten
186             pNewNode->SetLast(0);
187             pNewNode->SetNext(0);
188 
189             m_pRoot = pNewNode;
190             m_pEnd = pNewNode;
191         }
192         else if (m_pActuell->GetNext() == 0)
193         {
194             // Fall 2: Knoten am Ende der Liste anhaengen
195 
196             // Die Knoten m_pActuell und pNewNode miteinander verbinden (Neuen Knoten
197             // an das Ende der Liste, also an den "aktuellen" Knoten anhaengen
198             pNewNode->SetNext(0);
199             pNewNode->SetLast(m_pActuell);
200 
201             m_pActuell->SetNext(pNewNode);
202 
203             m_pEnd = pNewNode;
204         }
205         else
206         {
207             // Fall 3: Knoten irgendwo zwischen Anfang und Ende
208 
209             // Adresse des auf pActuell folgenden Knoten in pTemp zwischenspeichern
210             pTemp = m_pActuell->GetNext();
211 
212             // Den neuen Knoten zwischen pActuell und pTemp einfuegen (verketten)
213             // pActuell <-> pNewNode
214             m_pActuell->SetNext(pNewNode);
215             pNewNode->SetNext(m_pActuell);
216 
217             // pNewNode <-> pTemp
218             pNewNode->SetNext(pTemp);
219             pTemp->SetLast(pNewNode);
220         }
221 
222         // (Member)-Zeiger m_pActuell auf den neuen Knoten setzen
223         m_pActuell = pNewNode;
224 
225         return (0); // Rueckgabewert 0 (alles okay, kein Fehler)
226 
227     }
228     else
229     {
230         // Ist der Ruechgabewert pNewNode 0, so konnte kein Speicherplatz fuer
```

```
231         // den neuen Knoten angefordert werden.-> Fehlercode (1) zurueckgeben
232         return (1);
233     }
234 }
235
236
237 /*****************************************************************************/
238 /* int ShowList(void):                                                       */
239 /*                                                                           */
240 /* Aufgabe:                                                                  */
241 /*  - Die gesamte Liste ausgeben                                             */
242 /*                                                                           */
243 /* Uebergabeparameter:                                                       */
244 /*  - keiner                                                                 */
245 /*                                                                           */
246 /* Rueckgabewert:                                                            */
247 /*  - Fehlercode:   0 ... kein Fehler                                        */
248 /*                  1 ... Es ist keine Liste vorhanden                       */
249 /*                                                                           */
250 /* Vorgehensweise:                                                           */
251 /*  - Ist die Liste leer (m_pRoot = 0) nur den entsprechenden Fehlercode zurueckgeben  */
252 /*  - Andernfalls mit Hilfe eines Hilfszeigers (hier pShowNode) und einer Schleife be- */
253 /*    ginnend beim ersten Knoten dessen Inhalt mit Hilfe von GetValue ausgeben.        */
254 /*                                                                           */
255 /* Veraenderte Member-Variablen:                                             */
256 /*  - keine                                                                  */
257 /*****************************************************************************/
258 int CLinkedList::ShowList(void)
259 {
260     CNode *pShowNode;
261
262
263     if (m_pRoot)
264     {
265         // Wenn eine Liste vorhanden ist (m_pRoot > 0)
266         pShowNode = m_pRoot;
267         do
268         {
269             cout << pShowNode->GetValue() << endl;
270
271             pShowNode = pShowNode->GetNext();
272         }
273         while (pShowNode != 0);
274
275         return (0); // Rueckgabewert: alles okay
276     }
277     else
278     {
279         // Wenn keine Liste vorhanden ist (m_pRoot = 0)
280         return (1); // Rueckgabewert: Fehler
281     }
282 }
283
284
285 /*****************************************************************************/
286 /* int RemoveNode(void):                                                     */
287 /*                                                                           */
288 /* Aufgabe:                                                                  */
289 /*  Den Knoten auf welches der (Member-) Zeiger (m_pActuell) zeigt, aus der Liste ent- */
290 /*  fernen, und den Speicherplatz fuer diesen Knoten wieder freigeben.      */
291 /*                                                                           */
292 /* Uebergabeparameter:                                                       */
293 /*  - keiner                                                                 */
294 /*                                                                           */
295 /* Rueckgabewert:                                                            */
296 /*  - Fehlercode:   0 ... kein Fehler                                        */
297 /*                  1 ... Es ist kein Knoten vorhanden                       */
298 /*                                                                           */
299 /* Vorgehensweise:                                                           */
300 /*  - Pruefen, ob ueberhaupt schon ein Eintrag existiert (m_pRoot > 0)       */
301 /*  - Ist kein Knoten vorhanden (m_pRoot = 0), den entsprechenden Fehlercode dem auf-  */
```

```
302 /*    rufendem Programm zurueckgeben.                                          */
303 /*  - Ist zumindest ein Knoten vorhanden (m_pRoot > 0) den Knoten, auf den der */
304 /*    Zeiger pActuell zeigt loeschen. Beim Entfernen (loeschen) eines Knotens muss */
305 /*    zwischen 4 Faellen unterschieden werden:                                 */
306 /*                                                                             */
307 /*      - FALL 1: Die gesamte Liste besteht nur aus einem einzigen Knoten. Dieser */
308 /*        Knoten soll nun geloescht werden.                                    */
309 /*        Kennzeichen: m_pActuell -> GetLast = 0                               */
310 /*                     m_pActuell -> GetNext = 0                               */
311 /*          - Den Speicherplatz mit der Adresse m_pActuell freigeben.          */
312 /*          - Die (Member-)Zeiger m_pRoot, m_pEnd und m_pActuell mit 0 laden, da die */
313 /*            Liste nun wieder leer ist.                                       */
314 /*                                                                             */
315 /*      - FALL 2: Die Liste besteht aus mehreren Knoten, davon wird der erste Knoten */
316 /*        geloescht.                                                           */
317 /*        Kennzeichen: m_pActuell -> GetLast = 0                               */
318 /*                     m_pActuell -> GetNext ungleich 0                        */
319 /*          - Den (Member-)Zeiger m_pActuell auf den auf m_pActuell folgenden Knoten */
320 /*            setzen.                                                          */
321 /*          - Den ersten Knoten (= m_pRoot) entfernen (loeschen) und den Speicherplatz */
322 /*            fuer diesen Knoten wieder freigeben.                             */
323 /*          - Den Zeiger auf den vor dem aktuellen Knoten (m_pActuell) zeigenden mit 0 */
324 /*            (NULL) laden.                                                    */
325 /*          - Den (Member-)Zeiger m_pRoot mit m_pActuell laden, da die Liste nun einen */
326 /*            neuen Beginn (Wurzel) besitzt.                                   */
327 /*                                                                             */
328 /*      - FALL 3: Die Liste besteht aus mehreren Knoten, davon wird der letzte Knoten */
329 /*        geloescht.                                                           */
330 /*        Kennzeichen: m_pActuell -> GetLast ungleich 0                        */
331 /*                     m_pActuell -> GetNext = 0                               */
332 /*          - Den (Member-)Zeiger m_pActuell auf den von m_pActuell vorhergehenden */
333 /*            Knoten setzen.                                                   */
334 /*          - Den letzten Knoten (= m_pEnd) entfernen (loeschen) und den Speicherplatz */
335 /*            fuer diesen Knoten wieder freigeben.                             */
336 /*          - Den vom aktuellen Knoten (m_pActuell) aus auf den naechsten Knoten */
337 /*            zeigenden Zeiger mit 0 (NULL) laden, da es keinen nachfolgenen Knoten */
338 /*            gibt, bzw. der aktuelle Knoten auch gleichzeitig der letzte Knoten ist. */
339 /*          - Den (Member-)Zeiger m_pEnd mit m_pActuell laden, da die Liste nun ein */
340 /*            neues Ende besitzt.                                              */
341 /*                                                                             */
342 /*      - FALL 4: Die Liste besteht aus mehreren Knoten, davon wird weder der erste */
343 /*        Knoten noch der letzte Knoten geloescht. Sondern ein Knoten irgendwo */
344 /*        zwischen Ersten und Letztem.                                         */
345 /*        Kennzeichen: m_pActuell -> GetLastEntry ungleich 0                   */
346 /*                     m_pActuell -> GetNextEntry ungleich 0                   */
347 /*          - Die Adresse des auf m_pActuell folgenden Knotens in pTempNach sichern, */
348 /*            und den zu m_pActuell vorherigen Eintrag in pTempVor sichern.    */
349 /*          - Den Knoten auf den der Zeiger m_pActuell zeigt entfernen (loeschen) und */
350 /*            den Speicherplatz fuer diesen Knoten wieder freigeben.           */
351 /*          - Die Knoten pTempVor und pTempNach miteinander verketten.        */
352 /*          - Den (Member-)Zeiger m_pActuell mit pTempVor laden                */
353 /*                                                                             */
354 /* Veraenderte Member-Variablen:                                               */
355 /*  - m_pRoot                                                                  */
356 /*  - m_pEnd                                                                   */
357 /*  - m_pActuell                                                               */
358 /*****************************************************************************/
359 int CLinkedList::RemoveNode(void)
360 {
361     CNode *pTempVor;     // Fuer Fall 4 (das zu loeschende Entry befindet sich
362     CNode *pTempNach;    //   in der Mitte der Liste)
363
364
365     if (m_pRoot == NULL)
366     {
367         // Wenn kein Knoten vorhanden ist (m_pRoot = 0)
368         return (1); // Rueckgabewert: Fehler
369     }
370     else
371     {
372         if (m_pActuell->GetLast() == NULL)
```

```
373         {
374             if (m_pActuell->GetNext() == NULL)
375             {
376                 // Fall 1: Die Liste besteht nur aus einem einzigen Knoten
377                 // Dieser Knoten wird nun geloescht, der Speicherplatz fuer
378                 // diesen Knoten wird wieder freigegeben und die Zeiger m_pRoot,
379                 // m_pActuell und m_pEnd werden geloescht
380                 delete (m_pActuell);
381 
382                 m_pRoot = 0;
383                 m_pActuell = 0;
384                 m_pEnd = 0;
385             }
386             else
387             {
388                 // Fall 2: Die Liste besteht aus mehreren Knoten.
389                 // Der erste Knoten wird nun geloescht, und der Speicherplatz fuer
390                 // dieses Knoten wird wieder freigegeben
391 
392                 // Zeiger pActuell auf den naechsten Knoten setzen
393                 m_pActuell = m_pActuell->GetNext();
394 
395                 // Den ersten Knoten (m_pRoot) loeschen und den Speicherplatz fuer
396                 //  diesen Knoten wieder freigeben
397                 delete (m_pRoot);
398 
399                 // Den Zeiger m_pActuell->SetLast() mit NULL laden, da ja nun
400                 // dieser Knoten das erste Entry der Liste ist
401                 m_pActuell->SetLast(0);
402 
403                 // Den Zeiger m_pRoot mit m_pActuell laden, da die Liste nun einen
404                 // neuen Beginn (Wurzel) besitzt
405                 m_pRoot = m_pActuell;
406             }
407         }
408         else
409         {
410             if (m_pActuell->GetNext() == NULL)
411             {
412                 // Fall 3: Die Liste besteht aus mehreren Knoten.
413                 // Der letzte Knoten wird nun geloescht, und der Speicherplatz
414                 // fuer diesen Knoten wird wieder freigegeben
415 
416                 // Zeiger m_pActuell auf den vorherige Knoten setzen
417                 m_pActuell = m_pActuell->GetLast();
418 
419                 // Den letzten Knoten (m_pEnd) loeschen,
420                 // den Speicherplatz fuer dieses Knoten wieder freigeben
421                 delete (m_pEnd);
422 
423                 // Den Zeiger m_pActuell->SetNext mit NULL laden, da ja nun
424                 // dieser Knoten der letzte Knoten der Liste ist.
425                 m_pActuell->SetNext(0);
426 
427                 // Den Zeiger m_pEnd mit m_pActuell laden, da die Liste nun
428                 // ein neues Ende besitzt
429                 m_pEnd = m_pActuell;
430             }
431             else
432             {
433                 // Fall 4: Die Liste besteht aus mehreren Knoten.
434                 // Es wird weder der erste noch der letzte Knoten geloescht,
435                 // sondern ein Knoten irgendwo in der Mitte der Liste, der
436                 // Speicherplatz fuer diesen Knoten wird wieder freigegeben.
437 
438                 // Die Adresse des vorhergehenden Knotens in pTempVor sichern,
439                 // und die Adresse des naechsten Knotens in pTempNach sichern.
440                 // (Anm.: Bezueglich m_pActuell)
441                 pTempVor = m_pActuell->GetLast();
442                 pTempNach = m_pActuell->GetNext();
443 
```

```
444                 // Den Knoten m_pAktuell loeschen, der Speicherplatz fuer
445                 // diesen Knoten wird wieder freigegeben
446                 delete (m_pActuell);
447
448                 // Die Knoten pTempVor und pTempNach verketten
449                 pTempVor->SetNext(pTempNach);
450                 pTempNach->SetLast(pTempVor);
451
452                 // Den Zeiger m_pActuell mit pTempVor laden
453                 m_pActuell = pTempVor;
454             }
455         }
456     }
457
458     return (0); // Rueckgabewert: alles okay
459 }
460
461
462 /******************************************************************************/
463 /* int DeleteList(void):                                                      */
464 /*                                                                            */
465 /* Aufgabe:                                                                   */
466 /*  - Die gesamte Liste vom Speicher entfernen                                */
467 /*                                                                            */
468 /* Uebergabeparameter:                                                        */
469 /*  - keiner                                                                  */
470 /*                                                                            */
471 /* Rueckgabewert:                                                             */
472 /*  - Fehlercode:    0 ... kein Fehler                                        */
473 /*                   1 ... Es ist keine Liste vorhanden                       */
474 /*                                                                            */
475 /* Vorgehensweise:                                                            */
476 /*  - Ist die Liste leer (m_pRoot = 0) muss sie auch nicht geloescht werden!! - Was   */
477 /*    nicht vorhanden ist, kann auch nicht geloescht werden!!                 */
478 /*  - Andernfalls mit Hilfe zweier Hilfszeigers (hier pDeleteNode und pDeleteNext)    */
479 /*    beginnend beim ersten Knoten diesen mit Hilfe einer Schleife Knoten fuer Knoten */
480 /*    vom Speicher entfernen. Dabei zeigt der Zeiger pDeleteNode immer auf den zu ent- */
481 /*    fernenden Knoten und der Zeiger pDeleteNext auf den darauf folgenden Knoten.    */
482 /*    Dieser Vorgang wird solange fortgesetzt, bis pDeleteNext ein "Null-Zeiger" ist, */
483 /*    D.h. der Zeiger den Wert Null beinhaltet. Denn dann ist er am Ende der Liste an- */
484 /*    gekommen.                                                               */
485 /*                                                                            */
486 /* Veraenderte Member-Variablen:                                              */
487 /*  - m_pRoot                                                                 */
488 /*  - m_pEnd                                                                  */
489 /*  - m_pActuell                                                              */
490 /******************************************************************************/
491 int CLinkedList::DeleteList(void)
492 {
493     CNode *pDeleteNode;
494     CNode *pDeleteNext;
495
496
497     // Liste nur dann loeschen, wenn eine vorhanden ist!
498     if (m_pRoot != 0)
499     {
500         // Zeiger pDeleteNode auf den ersten Knoten setzen
501         pDeleteNode = m_pRoot;
502
503         do
504         {
505             // Zeiger pDeleteNext auf den naechsten Knoten setzen
506             pDeleteNext = pDeleteNode->GetNext();
507
508             // Knoten auf den der Zeiger pDeleteNode zeigt loeschen
509             delete (pDeleteNode);
510
511             // Zeiger pDeleteNode auf den naechsten Knoten setzen
512             pDeleteNode = pDeleteNext;
513         } while (pDeleteNode);
514
```

```
515         // Die Membervariablen ebenfalls loeschen
516         m_pRoot = 0;
517         m_pEnd = 0;
518         m_pActuell = 0;
519
520         return (0); // Rueckgabewert: alles okay
521     }
522     else
523     {
524         // Wenn keine Liste vorhanden ist (m_pRoot = 0)
525         return (1); // Rueckgabewert: Fehler
526     }
527 }
528
529
530 /***********************************************************************************/
531 /* int Scroll (int nDirection):                                                    */
532 /*                                                                                 */
533 /* Aufgabe:                                                                        */
534 /*  - In der Liste entweder um einen Knoten vor oder zurueck scrollen, oder an den */
535 /*    Beginn oder das Ende der Liste scrollen. Der Uebergabeparameter gibt dabei die */
536 /*    Scrollrichtung an..                                                          */
537 /*                                                                                 */
538 /* Uebergabeparameter:                                                             */
539 /*  - nDirection: gibt die Scrollrichtung an                                       */
540 /*                - zum Anfang der Liste                                           */
541 /*                - zum Ende der Liste                                             */
542 /*                - zum vorhergehenden Knoten                                      */
543 /*                - zum naechsten Knoten                                           */
544 /*                                                                                 */
545 /* Rueckgabewert:                                                                  */
546 /*  - Fehlercode:  0 ... kein Fehler                                               */
547 /*                 1 ... Es ist keine Liste vorhanden                              */
548 /*                 2 ... Scrollbewegung nach vor nicht moeglich                    */
549 /*                 3 ... Scrollbewegung zurueck nicht moeglich                     */
550 /*                                                                                 */
551 /* Vorgehensweise:                                                                 */
552 /*  - Fuer diese Funktionalitaet eignet sich sehr gut ein "switch"-Anweisung, wobei */
553 /*    die einzelnen Faelle mit je einer Konstante - in Form einer enum-Konstante   */
554 /*    siehe Beginn des Codes) - realisiert wurde.                                  */
555 /*  - Zu Beginn muss aber ueberprueft werden, ob ueberhaupt eine Liste vorhanden ist. */
556 /*  - Befindet sich der zu scrollende Zeiger m_pActuell am Ende der Liste, so kann sie */
557 /*    nicht mehr weiter nach vor gescrollt werden. Fuer diesen Fall muss ein Fehler- */
558 /*    code an das aufrufende Programm zurueckgegeben werden. Analoges gilt auch, wenn */
559 /*    der Zeiger m_pActuell am Beginn der Liste befindet.                          */
560 /*                                                                                 */
561 /* Veraenderte Member-Variablen:                                                   */
562 /*  - m_pActuell                                                                   */
563 /***********************************************************************************/
564 int CLinkedList::Scroll(int nDirection)
565 {
566     int ErrorCode;
567
568
569     if (m_pRoot)
570     {
571         // Es ist eine Liste vorhanden
572         switch(nDirection)
573         {
574         case SCROLL_BACK:
575             if (m_pActuell->GetLast())
576             {
577                 m_pActuell = m_pActuell->GetLast();
578                 ErrorCode = 0;
579             }
580             else
581             {
582                 // Fehler:
583                 ErrorCode = 2;
584             }
585             break;
```

```
586
587             case SCROLL_FORWARD:
588                 if (m_pActuell->GetNext())
589                 {
590                     m_pActuell = m_pActuell->GetNext();
591                     ErrorCode = 0;
592                 }
593                 else
594                 {
595                     // Fehler:
596                     ErrorCode = 1;
597                 }
598                 break;
599
600             case SCROLL_BEGIN:
601                 m_pActuell = m_pRoot;
602                 ErrorCode = 0;
603                 break;
604
605             case SCROLL_END:
606                 m_pActuell = m_pEnd;
607                 ErrorCode = 0;
608                 break;
609             }
610         }
611         else
612         {
613             // es ist keine Liste vorhande, Fehlercode (1) zurueckgeben
614             ErrorCode = 1;
615         }
616
617         return (ErrorCode);
618 }
```

Aufgrund der ausgiebigen Kommentare und der ausführlichen Abhandlung in Kapitel 2 (ab Seite 5) ist hier eine weitere Beschreibung überflüssig!

Zu Beachten ist vielleicht, dass hier Speicher dynamisch angefordert wird. Daher ist es auch ganz wichtig dass der Destruktur diesen Speicher wieder freigibt. Hier, indem er einfach die Memberfunktion `DeleteList()` aufruft. Denn diese Memberfunktion erfüllt genau diesen Zweck (Zeilen 96 bis 98).

3.3 Demonstrationsbeispiel

Anhand eines einfachen Demonstrationsbeispiels soll nun die Klasse `CLinkedList` ausprobiert werden. Listing 3.5 zeigt dieses sehr einfache und selbsterklärende Demonstrationsbeispiel. Es sollen nur die realisierten Funktionen angewendet werden, also Knoten hinzufügen, Liste anzeigen, Knoten löschen und Scrollen. Auf eine Werteingaben wurde bewusst verzichtet, da dies das Demonstrationsbeispiel nur unnötig komplex machen würde.

Abbildung 3.5 (auf Seite 35) zeigt einen Bildschirmausdruck des Ergebnisses des Demonstrationsbeispiels (Konsolenanwendung).

Listing 3.5: CLinkedList_Demo.cpp (Demonstrationsbeispiel)

```
/*****************************************************************************/
/* Demonstrationsprojekt zur eigenen Klasse CLinkedList                      */
/*                                                                           */
/* (doppelt) verkettete Liste                                                */
/*                                                                           */
/* Entwickler: Buchgeher Stefan                                              */
/* Entwicklungsbeginn dieser Klasse: 10. Oktober 2006                        */
/* Funktionsfaehig seit: 10. Oktober 2006                                    */
/* Letzte Bearbeitung: 25. Januar 2007                                       */
/*****************************************************************************/

/***************** Include-Dateien *******************************************/
#include "CLinkedList.h"
#include <iostream.h>


/***************** Hauptprogramm *********************************************/
void main(void)
{
    int ErrorCode;


    CLinkedList  Liste1(4);


    cout << "Fall 1 - nur ein Knoten vorhanden" << endl;
    ErrorCode = Liste1.ShowList();

    cout << endl << "... einzigen Wert entfernen ..." << endl;
    ErrorCode = Liste1.RemoveNode();

    cout << endl << "... leere Liste ausgeben ..." << endl;
    ErrorCode = Liste1.ShowList();



    cout << endl << "Fall 2" << endl;
    cout << "Liste ausgeben ..." << endl;
    ErrorCode = Liste1.ShowList();

    cout << endl << "... Einen Wert (3) hinzufuegen ..." << endl;
    ErrorCode = Liste1.AddNode(3);

    cout << endl << "... Liste ausgeben ..." << endl;
    ErrorCode = Liste1.ShowList();

    cout << endl << "... Weitere Werte (24, -454, 1033, -1124) hinzufuegen ..." << endl;
    ErrorCode = Liste1.AddNode(24);
    ErrorCode = Liste1.AddNode(-454);
    ErrorCode = Liste1.AddNode(1033);
    ErrorCode = Liste1.AddNode(-1124);

    cout << endl << "... Liste ausgeben ..." << endl;
    ErrorCode = Liste1.ShowList();

    cout << endl << "... Letzten beiden Wert entfernen ..." << endl;
    ErrorCode = Liste1.RemoveNode();
    ErrorCode = Liste1.RemoveNode();

    cout << endl << "... Liste ausgeben ..." << endl;
    ErrorCode = Liste1.ShowList();

    cout << endl << "... Einen Wert (-33) hinzufuegen ..." << endl;
    ErrorCode = Liste1.AddNode(-33);

    cout << endl << "... Liste ausgeben ..." << endl;
    ErrorCode = Liste1.ShowList();

    cout << endl << "... Einen Knoten zurueck scrollen ..." << endl;
    ErrorCode = Liste1.Scroll(SCROLL_BACK);
```

```
71
72      cout << endl << "... Einen Wert (100) hinzufuegen ..." << endl;
73      ErrorCode = Liste1.AddNode(100);
74
75      cout << endl << "... Liste ausgeben ..." << endl;
76      ErrorCode = Liste1.ShowList();
77
78
79      cout << endl << "... Zum Beginn der Liste scrollen ..." << endl;
80      ErrorCode = Liste1.Scroll(SCROLL_BEGIN);
81
82      cout << endl << "... Einen Wert (-100) hinzufuegen ..." << endl;
83      ErrorCode = Liste1.AddNode(-100);
84
85      cout << endl << "... Liste ausgeben ..." << endl;
86      ErrorCode = Liste1.ShowList();
87
88      cout << endl << "... Liste entfernen ..." << endl;
89      ErrorCode = Liste1.DeleteList();
90  }
```

```
"E:\CPP linked list\LinkedList mit Fehlercode\Debug\LinkedList.exe"
Fall 1 - nur ein Knoten vorhanden
4

... einzigen Wert entfernen ...

... leere Liste ausgeben ...

Fall 2
Liste ausgeben ...

... Einen Wert (3) hinzufuegen ...

... Liste ausgeben ...
3

... Weitere Werte (24, -454, 1033, -1124) hinzufuegen ...

... Liste ausgeben ...
3
24
-454
1033
-1124

... Letzten beiden Wert entfernen ...

... Liste ausgeben ...
3
24
-454

... Einen Wert (-33) hinzufuegen ...

... Liste ausgeben ...
3
24
-454
-33

... Einen Knoten zurueck scrollen ...

... Einen Wert (100) hinzufuegen ...

... Liste ausgeben ...
3
24
-454
100
-33

... Zum Beginn der Liste scrollen ...

... Einen Wert (-100) hinzufuegen ...

... Liste ausgeben ...
3
-100
24
-454
100
-33

... Liste entfernen ...
Press any key to continue
```

Abbildung 3.2: Realisierung einer doppelt Verketteten Liste in C++ (Demo)

Kapitel 4

Realisierung einer einfach verketteten Liste in Visual Basic

(Einfach) Verkettete Listen lassen sich auch in Visual Basic sehr gut realisieren. Hier soll nun die grundsätzliche Vorgehensweise an einem sehr einfachen Beispiel gezeigt werden.

In dieses Beispiel soll eine Textliste (z.B. eine Namensliste) erstellt werden, wobei ein neuer Knoten immer nur an das Ende der Liste angehängt werden soll. Das Löschen eines einzelnen Knotens und das Scrollen wurde hier bewusst (noch) nicht realisiert. Abbildung 4.3 (Seite 45) zeigt das ausgeführte Programm.

Da auch Visual Basic eine objektorientierte Sprache ist, und daher das Konzept der Klassen sehr gut unterstützt, wurde auch dieses Beispiel mit Klassen realisiert. Auch hier wurde die gesamte Liste wieder mit zwei Klassen realisiert (Abbildung 4.1):

- Klasse `clsNodeE`: Diese bildet nur den eigentlichen Knoten mit den zu speichernden Daten ab.
- Klasse `clsLinkedListE`: Diese beinhaltet die gesamte Liste, ähnlich, wie sie im Kapitel 2 erläutert wurde. Nur mit dem Unterschied, dass es sich nun um eine einfach verkettete Liste handelt.

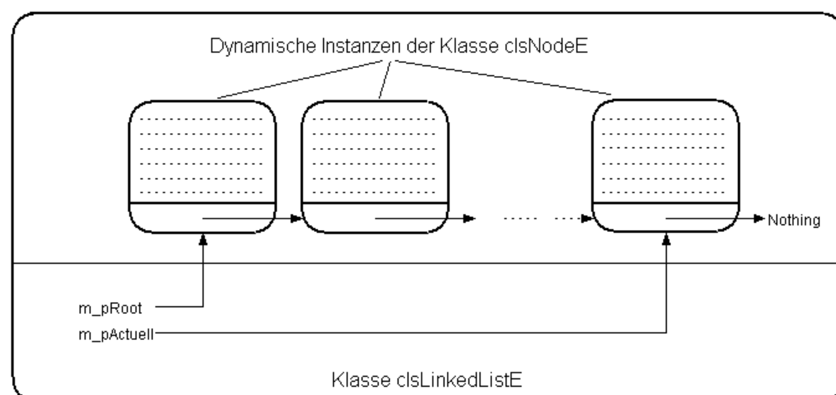

Abbildung 4.1: Realisierung einer einfach verketteten Liste in Visual Basic

Anmerkung: Auch hier gilt, dass dies hier kein Visual-Basic-Kurs ist. Ich beschreibe hier nur kurz die wesentlichen Punkte für eine mögliche Realisierung der einfach verketteten Listen in Visual Basic (Version 6).
Ich gehe daher auch hier davon aus, dass der/die Leser(in) die Grundlagen von Visual Basic kennt und diese auch beherrscht. Wenn nicht so empfehle ich die Bücher *Jetzt lerne ich Visual Basic* [3], *Visual Basic 6 - Grundlagen und Profiwissen* [4] und als Nachschlagerefernez das Buch *Visual Basic 6 Referenz* [5]. Auch im Internet gibt es eine Fülle von Informationen zu Visual Basic. Hier sei nur [6] erwähnt.

4.1 Klasse `clsNodeE`

Auch in Visual Basic ist diese Klasse sehr einfach. Sie beinhaltet nur die Membervariablen für die Daten (hier also `m_sValue`, Zeile 18, für einen String) und den Zeiger zum folgenden Knoten (`m_pNext`, Zeile 20), einen Konstruktor zur Initialisierung der Membervariablen (Zeilen 42 bis 47) einen Destruktor, der hier eigentlich keine Aufgabe hat (Zeilen 61 und 62), und die Funktionen zum Setzen und Auslesen dieser Membervariablen (Zeilen 66 bis 83). Listing 4.1 zeigt die Realisierung dieser Klasse.

Anmerkung zu den Zeilennummern: Die Zeilennummern sind auch hier **nicht** Bestandteil des Quellcodes. Sie dienen auch hier nur der besseren Orientierung.

Listing 4.1: clsNodeE.cls

```
1  '*****************************************************************************
2  '* Klasse clsNodeE       (clsNodeE.cls)                                      *
3  '*                                                                           *
4  '* Kurzbeschreibung                                                          *
5  '*  Klasse fuer einen Knoten einer einfach verketteten Liste (engl. linked list)  *
6  '*                                                                           *
7  '* Aenderungen/Ergaenzungen                                                  *
8  '*                                                                           *
9  '* Entwickler: Buchgeher Stefan                                              *
10 '* Entwicklungsbeginn dieser Klasse: 23. Februar 2007                        *
11 '* Funktionsfaehig seit: 23. Februar 2007                                    *
12 '* Letzte Bearbeitung: 5. Maerz 2007                                         *
13 '*****************************************************************************
14 Option Explicit
15
16
17 '***************** Member-Variablen ******************************************
18 Private m_sValue As String      'Diese Variable beinhaltet den zu speichernden
19                                 'Inhalt (Daten)
20 Private m_pNext As clsNodeE     'Zeiger auf den naechsten Knoten
21
22
23 '***************** Member-Funktionen *****************************************
24 'Public Property Let SetValue(ByVal sValue As String)
25 'Public Property Get GetValue() As String
26 'Public Property Set SetNext(ByVal pNext As clsNodeE)
27 'Public Property Get GetNext() As clsNodeE
28
29
30 '***************** Konstruktor ***********************************************
31
32 '*****************************************************************************
33 '* Private Sub Class_Initialize():                                           *
34 '*                                                                           *
35 '* Aufgaben:                                                                 *
36 '*  - Die Member-Variablen initialisieren                                    *
```

```
37  '*                                                                          *
38  '* Veraenderte Member-Variablen:                                            *
39  '*  - m_sValue                                                              *
40  '*  - m_pNext                                                               *
41  '****************************************************************************
42  Private Sub Class_Initialize()
43      'Daten initialisieren (hier mit einem Leerzeichen)
44      m_sValue = ""
45      'Zeiger auf den naechsten Knoten initialisieren
46      Set m_pNext = Nothing
47  End Sub
48
49
50  '***************** Destruktor ***********************************************
51
52  '****************************************************************************
53  '* Private Sub Class_Terminate():                                           *
54  '*                                                                          *
55  '* Aufgabe:                                                                 *
56  '*  Hat hier keine Aufgabe!                                                 *
57  '*                                                                          *
58  '* Veraenderte Member-Variablen:                                            *
59  '*  keine                                                                   *
60  '****************************************************************************
61  Private Sub Class_Terminate()
62  End Sub
63
64
65  '***************** Member-Funktionen (Realisierung) *************************
66  Public Property Let SetValue(ByVal sValue As String)
67      m_sValue = sValue
68  End Property
69
70
71  Public Property Get GetValue() As String
72      GetValue = m_sValue
73  End Property
74
75
76  Public Property Set SetNext(ByVal pNext As clsNodeE)
77      Set m_pNext = pNext
78  End Property
79
80
81  Public Property Get GetNext() As clsNodeE
82      Set GetNext = m_pNext
83  End Property
```

4.2 Klasse `clsLinkedListE`

Die Klasse `clsLinkedListE` (Listing 4.2) ist auch hier schon etwas umfangreicher. Diese Klasse realisiert die eigentliche verkettete Liste, wobei hier nur das Hinzufügen eines neuen Knotens ans Ende der Liste (Zeilen 114 bis 140), das Anzeigen der gesamten Liste in einer so genannten List-Box (Zeilen 167 bis 188) und das Entfernen der gesamten Liste (Zeilen 225 bis 260) realisiert sind. Weiters den Konstruktor zur Initialisierung der Membervariablen (Zeilen 40 bis 43) und einen Destruktor zum Entfernen der gesamten Liste (Zeilen 58 bis 63).
Dieses Beispiel wurde bewusst so gewählt, da es nur die allerwichtigsten Funktionen beinhalten soll, um das Prinzip der verketteten Liste auch in Visual Basic zu demonstrieren.

Listing 4.2: clsLinkedListE.cls

```
'*****************************************************************************
'* Klasse clsLinkedListE        (clsLinkedListE.cls)                         *
'*                                                                           *
'* Kurzbeschreibung                                                          *
'*  Klasse fuer eine einfach verkettete Liste (engl. linked list)            *
'*                                                                           *
'* Aenderungen/Ergaenzungen                                                  *
'*                                                                           *
'* Entwickler: Buchgeher Stefan                                              *
'* Entwicklungsbeginn dieser Klasse: 23. Februar 2007                        *
'* Funktionsfaehig seit: 23. Februar 2007                                    *
'* Letzte Bearbeitung: 5. Maerz 2007                                         *
'*****************************************************************************
Option Explicit


'***************** Member-Variablen *****************************************
Private m_pRoot As clsNodeE       'Zeiger auf den ersten Knoten der Liste
Private m_pActuell As clsNodeE    'Zeiger auf den aktuellen Knoten der Liste


'***************** Member-Funktionen ****************************************
'Public Function AddNode(ByVal sValue As String) As Integer
'Public Function ShowList(ListBox As Object) As Integer
'Public Function RemoveList() As Integer


'***************** Konstruktor **********************************************

'*****************************************************************************
'* Private Sub Class_Initialize():                                           *
'*                                                                           *
'* Aufgaben:                                                                 *
'*  - Die Member-Variablen mit 0 (= Nothing) initialisieren                  *
'*                                                                           *
'* Veraenderte Member-Variablen:                                             *
'*  - m_pRoot                                                                *
'*  - m_pActuell                                                             *
'*****************************************************************************
Private Sub Class_Initialize()
    Set m_pRoot = Nothing
    Set m_pActuell = Nothing
End Sub


'***************** Destruktor ***********************************************

'*****************************************************************************
'* Private Sub Class_Terminate():                                            *
'*                                                                           *
'* Aufgabe:                                                                  *
'*  Die gesamte Liste (falls vorhanden) vom Speicher entfernen.              *
'*                                                                           *
'* Veraenderte Member-Variablen:                                             *
'*  - m_pRoot                                                                *
'*  - m_pActuell                                                             *
'*****************************************************************************
Private Sub Class_Terminate()
    Dim ErrorCode As Integer


    ErrorCode = RemoveList()
End Sub


'***************** Member-Funktionen (Realisierung) *************************

'*****************************************************************************
'* Public Function AddNode(sValue As String) As Integer:                     *
'*                                                                           *
```

```
71  '* Aufgabe:                                                                    *
72  '*  Neuen Knoten (mit dem String sValue) ans Ende der bestehenden Liste anhaengen.  *
73  '*                                                                             *
74  '* Uebergabeparameter:                                                         *
75  '*  sValue: Dieser Wert (String) soll in den neu hinzugefuegten Knoten gespeichert  *
76  '*          werden.                                                            *
77  '*                                                                             *
78  '* Rueckgabewert:                                                              *
79  '*  - Fehlercode:   0 ... kein Fehler                                          *
80  '*                  1 ... Es kann kein Knoten hinzugefuegt werden, weil kein Speicher *
81  '*                        alloziiert werden kann!                              *
82  '*                                                                             *
83  '* Vorgehensweise:                                                             *
84  '*  - Speicherplatz fuer den neuen Knoten dynamisch anfordern.                 *
85  '*  - Die zu speichernden Daten im neuen Knoten (pNewNode) speichern (mit Hilfe der  *
86  '*    Memberfunktion SetValue der Klasse clsNode)                              *
87  '*  - Das Verketten der Knoten ist nun abhaengig davon, an welcher Position sich der  *
88  '*    neue Knoten befindet:                                                    *
89  '*                                                                             *
90  '*          - FALL 1: Es ist noch keine Liste Vorhanden, der "neue" Knoten ist daher  *
91  '*            der erste und befindet sich daher am Anfang der Liste.           *
92  '*            (Kennzeichen: m_pRoot = 0 (NOTHING)                              *
93  '*            Da es das erste Element der Liste ist gilt fuer diesen Knoten:   *
94  '*              - Der Zeiger auf den naechsten Knoten muss den Wert 0 (NOTHING)  *
95  '*                beinhalten (pNewNode.SetNext = NOTHING)                      *
96  '*              - Die (Member-)Zeiger m_pRoot und m_pActuell zeigen auf diesen ersten *
97  '*                Knoten                                                       *
98  '*                                                                             *
99  '*          - FALL 2: Es ist schon eine Liste vorhanden, der neue Knoten (pNewNode)  *
100 '*            wird an das Ende der Liste angehängt. Der Member-Zeiger m_pActuell zeigt *
101 '*            hier immer das Ende der Liste an.                                *
102 '*            (Kennzeichen: m_pRoot > 0                                        *
103 '*            Da es der letzte Knoten der Liste ist gilt fuer diesen Knoten:   *
104 '*              - Der Zeiger von pNewNode auf den naechsten Knoten muss den Wert 0  *
105 '*                (NOTHING) beinhalten (pNewNode.SetNext = NOTHING)            *
106 '*              - Der Zeiger von pActuell auf den naechste Knoten muss den Wert des  *
107 '*                Zeigers pNewNode beinhalten (m_pActuell.SetNext = pNewNode   *
108 '*              - Der (Member-)Zeiger m_pActuell zeigt auf diesen letzten Knoten.  *
109 '*                                                                             *
110 '* Veraenderte Member-Variablen:                                               *
111 '*  - m_pRoot                                                                  *
112 '*  - m_pActuell                                                               *
113 '*******************************************************************************
114 Public Function AddNode(sValue As String) As Integer
115     Dim pNewNode As clsNodeE
116
117
118     'Knoten erzeugen
119     Set pNewNode = New clsNodeE
120
121     'Daten im  Knoten speichern
122     pNewNode.SetValue = sValue
123
124     Set pNewNode.SetNext = Nothing
125
126     If m_pRoot Is Nothing Then
127         'Fall 1: Erster Knoten der Liste
128         'Set pNewNode.SetNext = Nothing
129         Set m_pRoot = pNewNode
130     Else
131         'Fall 2: neuen (weiteren) Knoten an das Ende der Liste anhaengen
132         'Set pNewNode.SetNext = Nothing
133         Set m_pActuell.SetNext = pNewNode
134     End If
135
136     Set m_pActuell = pNewNode
137
138     'Rueckgabewert
139     AddNode = 0      'kein Fehler
140 End Function
141
```

```
142
143 '*************************************************************************************
144 '* Public Function ShowList(ListBox As Object) As Integer:                            *
145 '*                                                                                    *
146 '* Aufgabe:                                                                           *
147 '*  - Die gesamte Liste in der uebergebenen List-Box anzeigen                         *
148 '*                                                                                    *
149 '* Uebergabeparameter:                                                                *
150 '*  - Listbox (Objekt)                                                                *
151 '*                                                                                    *
152 '* Rueckgabewert:                                                                     *
153 '*  - Fehlercode:   0 ... kein Fehler                                                 *
154 '*                  1 ... Es ist keine Liste vorhanden                                *
155 '*                                                                                    *
156 '* Vorgehensweise:                                                                    *
157 '*  - ListBox loeschen                                                                *
158 '*  - Ist die Liste leer (m_pRoot = Nothing) den Text "Leer!" in die Listbox schreiben *
159 '*    und den entsprechenden Fehlercode zurueckgeben.                                 *
160 '*  - Andernfalls mit Hilfe eines Hilfszeigers (hier pShowNode) und einer Schleife be- *
161 '*    ginnend beim ersten Knoten dessen Inhalt mit Hilfe von GetValue in die Listbox  *
162 '*    schreiben.                                                                      *
163 '*                                                                                    *
164 '* Veraenderte Member-Variablen:                                                      *
165 '*  - keine                                                                           *
166 '*************************************************************************************
167 Public Function ShowList(ListBox As Object) As Integer
168     Dim pShowNode As clsNodeE
169
170
171     Set pShowNode = m_pRoot
172
173     ListBox.Clear
174     If pShowNode Is Nothing Then
175         ListBox.AddItem "Leer!"
176
177         'Rueckgabewert = Fehler
178         ShowList = 1
179     Else
180         While Not pShowNode Is Nothing
181             ListBox.AddItem pShowNode.GetValue
182             Set pShowNode = pShowNode.GetNext
183         Wend
184
185         'Rueckgabewert = kein Fehler
186         ShowList = 0
187     End If
188 End Function
189
190
191 '*************************************************************************************
192 '* Public Function RemoveList() As Integer:                                           *
193 '*                                                                                    *
194 '* Aufgabe:                                                                           *
195 '*  - Die gesamte Liste vom Speicher entfernen                                        *
196 '*                                                                                    *
197 '* Uebergabeparameter:                                                                *
198 '*  - keiner                                                                          *
199 '*                                                                                    *
200 '* Rueckgabewert:                                                                     *
201 '*  - Fehlercode:   0 ... kein Fehler                                                 *
202 '*                  1 ... Es ist keine Liste vorhanden                                *
203 '*                                                                                    *
204 '* Vorgehensweise:                                                                    *
205 '*  - Ist die Liste leer (m_pRoot = Nothing) muss sie auch nicht geloescht werden!! - *
206 '*    Was nicht vorhanden ist, kann auch nicht geloescht werden!!                     *
207 '*  - Andernfalls mit Hilfe zweier Hilfszeiger (hier pRemoveNode und pRemoveNext)     *
208 '*    beginnend beim ersten Knoten diesen mit Hilfe einer Schleife Knoten fuer Knoten *
209 '*    vom Speicher entfernen. Dabei zeigt der Zeiger pRemoveNode immer auf den zu ent- *
210 '*    fernenden Knoten und der Zeiger pRemoveNext auf den darauf folgenden Knoten.    *
211 '*    Dieser Vorgang wird solange fortgesetzt, bis pRemoveNext ein "Null-Zeiger" ist, *
212 '*    D.h. der Zeiger den Wert Nothing beinhaltet. Denn dann ist er am Ende der Liste *
```

```
213 '*     angekommen.                                                                  *
214 '*                                                                                  *
215 '* Anmerkung zum Entfernen von referenzierten Objekten:                             *
216 '*     Visual Basic entfernt automatisch allozierte Objekte, wenn keine Referenz auf *
217 '*     dieses Objekt zeigt. Anders ausgedrueckt: Wenn alle Referenzen auf ein dynamisch *
218 '*     erzeugtes Objekt den Wert Nothing beinhalten, entfernt Visual Basic automatisch *
219 '*     dieses Objekt und gibt den Speicher wieder frei.                             *
220 '*                                                                                  *
221 '* Veraenderte Member-Variablen:                                                    *
222 '*  - m_pRoot                                                                       *
223 '*  - m_pActuell                                                                    *
224 '***********************************************************************************
225 Public Function RemoveList() As Integer
226     Dim pRemoveNode As clsNodeE
227     Dim pRemoveNext As clsNodeE
228 
229 
230     'Liste nur dann loeschen, wenn eine vorhanden ist!
231     If m_pRoot Is Nothing Then
232          'Wenn keine Liste vorhanden ist (m_pRoot = nothing)
233         RemoveList = 1 ' Rueckgabewert: Fehler
234     Else
235         'Zeiger pRemoveNode auf den ersten Knoten setzen
236         Set pRemoveNode = m_pRoot
237 
238         'alle moeglichen Referenzen auf den ersten Knoten loeschen
239         Set m_pRoot = Nothing
240         Set m_pActuell = Nothing
241 
242         'Mit einer Schleife alle Knoten loeschen
243         Do
244             'Zeiger pRemoveNext auf den naechsten Knoten setzen
245              Set pRemoveNext = pRemoveNode.GetNext()
246 
247             'Auch den Zeiger, der vom zu loeschenden Knoten weggeht loeschen
248             Set pRemoveNode.SetNext = Nothing
249 
250             'Knoten auf den der Zeiger pRemoveNode zeigt loeschen
251             Set pRemoveNode = Nothing
252 
253             'Zeiger pRemoveNode auf den naechsten Knoten setzen
254             Set pRemoveNode = pRemoveNext
255         Loop Until pRemoveNode Is Nothing
256 
257         'Rueckgabewert = Liste erfolgreich geloescht (kein Fehler)
258         RemoveList = 0
259     End If
260 End Function
```

Aufgrund der ausgiebigen Kommentar verzichte ich auch hier auf eine weitere Beschreibung.

Zu Beachten ist vielleicht, dass auch hier der Speicher für jedes Objekt, also für jeden Knoten, dynamisch angefordert wird, und dieser daher im Destruktor wieder an das Betriebssystem zurückgegeben werden muss. Im Gegensatz zu C++ gibt es in Visual Basic keine vordefinierte Funktion zum Entfernen von Objekten. In Visual Basic werden Objekte automatisch entfernt, wenn es zu diesem Objekt keine Referenzen mehr gibt. D.h. Wenn in Visual Basic ein Objekt gelöscht werden soll, müssen alle Referenzen, die auf dieses Objekt zeigen mit “Nothing“ geladen werden.

4.3 Demonstrationsbeispiel

Auch das Demonstrationsbeispiel ist sehr einfach. Dazu wird ein einfaches Formular (hier: `frmLinkedListE`) mit den folgenden (Standard-)Steuerelemente benötigt:

- 1 TextBox (Name: `txtNeueDaten`)
- 1 ListBox (Name: `lstListe`)
- 2 CommandButton (Namen: `cmdNeueDaten` und `cmdListeAnzeigen`)
- Optional: 2 Frames (ohne Namen)

Abbildung 4.2 zeigt die Anordnung dieser Steuerelemente.

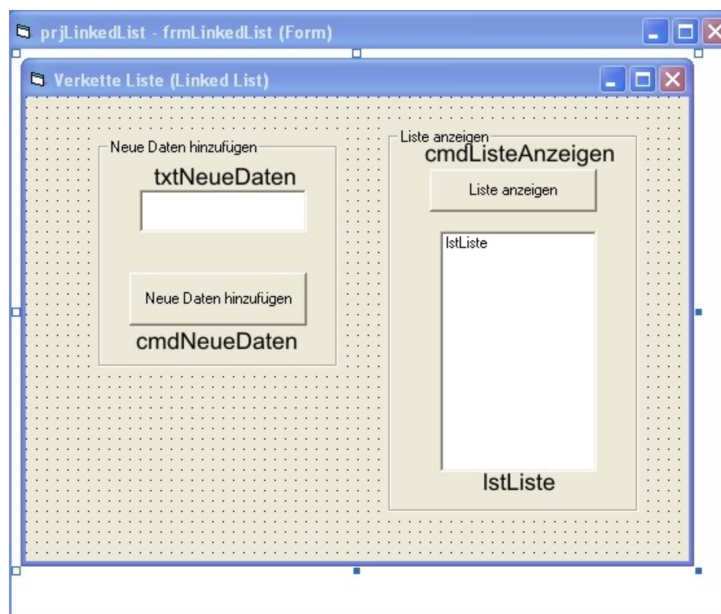

Abbildung 4.2: Steuerelemente für eine einfach verkettete Liste (Visual Basic)

So einfach wie das Formular ist, so einfach ist auch der Quellcode der dahintersteckt. Listing 4.3 zeigt das relativ kurze Programm.

Listing 4.3: frmLinkedListE.frm

```
'****************************************************************************
'* Demonstrationsprojekt zur einfach verketteten Liste (mit der Klasse clsLinkedListE) *
'*                                                                          *
'* Aenderungen/Ergaenzungen                                                 *
'*                                                                          *
'* Entwickler: Buchgeher Stefan                                             *
'* Entwicklungsbeginn dieser Klasse: 23. Februar 2007                       *
'* Funktionsfaehig seit: 23. Februar 2007                                   *
'* Letzte Bearbeitung: 4. Maerz 2007                                        *
'****************************************************************************
Option Explicit


'Instanz der verketteten Liste erzeugen
Dim DemoListe As clsLinkedListE
```

```
16
17
18 Private Sub Form_Load()
19     'Eine Instanz der (Einfach) verketteten Liste erstellen
20     Set DemoListe = New clsLinkedListE
21 End Sub
22
23
24 ' Neuen Knoten (mit dem Inhalt der Textbox) ans Ende der Liste anhaengen
25 Private Sub cmdNeueDaten_Click()
26     Dim ErrorCode As Integer
27
28     'Diesen Eintrag in die Liste hinzufuegen
29     ErrorCode = DemoListe.AddNode(Me.txtNeueDaten.Text)
30 End Sub
31
32
33 ' Gesamte Liste in einer Listbox anzeigen
34 Private Sub cmdListeAnzeigen_Click()
35     Dim ErrorCode As Integer
36
37     'Die gesamte Liste in der Listbox (lstListe) anzeigen
38     ErrorCode = DemoListe.ShowList(Me.lstListe)
39 End Sub
```

Hier nur kurz die wesentlichen Punkte:
In der Ereignisprozedur `Form_Load` muss eine Instanz der Verketteten Liste erzeugt werden (Zeilen 18 bis 21).
Das Hinzufügen eines neuen Knotens erfolgt durch das Anklicken der Taste `cmdNeueDaten` (Zeilen 25 bis 30), und das Anzeigen der gesamten Liste durch Anklicken der Taste `cmdListeAnzeigen` (Zeilen 34 bis 39).

Abbildung 4.3 zeigt das erfolgreich kompilierte Programm in Aktion.

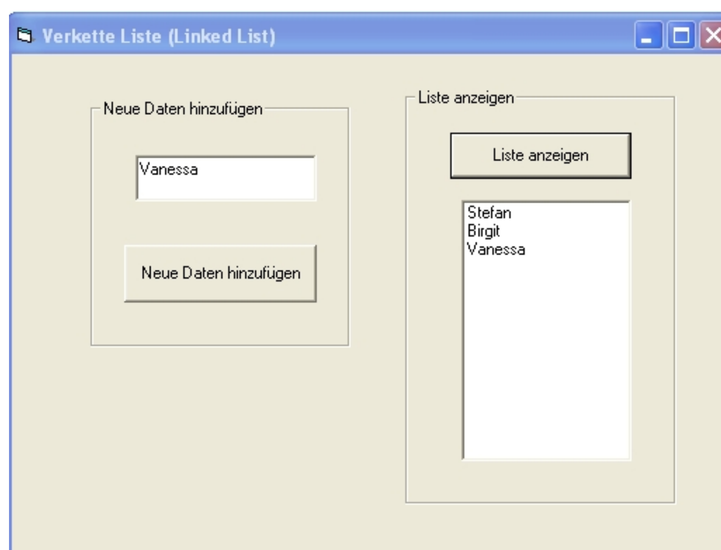

Abbildung 4.3: Einfaches Demo für eine einfach verkettete Liste in Visual Basic

Kapitel 5

Realisierung einer doppelt verketteten Liste in Visual Basic

In diesem Kapitel wird nun eine doppelt verkettete Liste in Visual Basic (Version 6) realisiert. Auch hier soll nur die grundsätzliche Vorgehensweise an einem sehr einfachen Beispiel gezeigt werden.

In dieses Beispiel soll wieder eine Textliste (z.B. eine Namensliste) erstellt werden, wobei zur Demonstration auch hier ein neuer Knoten immer nur an das Ende der Liste angehängt werden soll. Das Löschen des letzten Knotens wurde hier ebenfalls realisiert. Abbildung 5.3 (Seite 58) zeigt das ausgeführte Programm.

Auch hier wurde die gesamte Liste wieder mit zwei Klassen realisiert (Abbildung 5.1):

- Klasse `clsNodeD`: Diese bildet nur den eigentlichen Knoten mit den zu speichernden Daten ab.
- Klasse `clsLinkedListD`: Diese beinhaltet die gesamte Liste, wie sie im Kapitel 2 erläutert wurde, wobei aber nicht alle Funktionen realisiert wurden (z.B. das Scrollen wurde hier nicht realisiert).

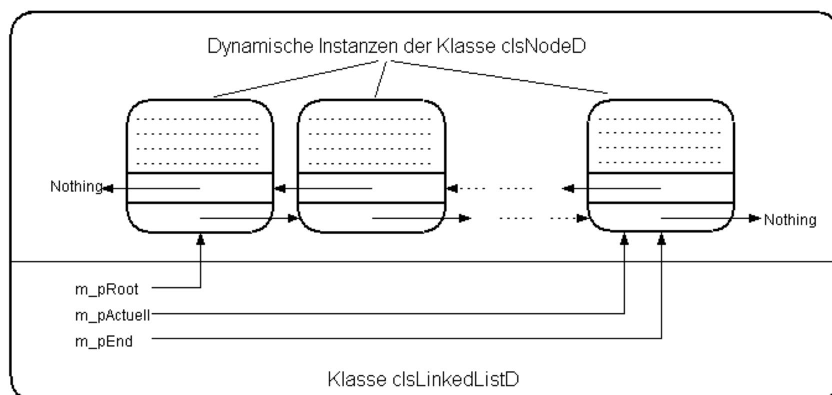

Abbildung 5.1: Realisierung einer doppelt verketteten Liste in Visual Basic

5.1 Klasse `clsNodeD`

Neu in dieser Klasse ist die Membervariable, die auf den vorhergehenden Knoten zeigt (`m_pLast`, Zeile 21), die dazugehörigen Funktionen zum Setzen und Lesen dieser Membervariable (Zeilen 91 bis 98) und auch die Initialisierung im Konstruktor (Zeile 51). Alles andere ist gleich wie die Klasse für einen einfach verketteten Knoten (Abschnitt 4.1 ab Seite 38). Listing 5.1 zeigt die Realisierung dieser Klasse.

Die Zeilennummern dienen auch hier nur der besseren Orientierung.

Listing 5.1: clsNodeD.cls

```
1  '*******************************************************************************
2  '* Klasse clsNodeD       (clsNodeD.cls)                                       *
3  '*                                                                             *
4  '* Kurzbeschreibung                                                            *
5  '*  Klasse fuer einen Knoten einer doppelt verketteten Liste (engl. linked list) *
6  '*                                                                             *
7  '* Aenderungen/Ergaenzungen                                                    *
8  '*                                                                             *
9  '* Entwickler: Buchgeher Stefan                                                *
10 '* Entwicklungsbeginn dieser Klasse: 28. Februar 2007                         *
11 '* Funktionsfaehig seit: 28. Februar 2007                                     *
12 '* Letzte Bearbeitung: 6. Maerz 2007                                          *
13 '*******************************************************************************
14 Option Explicit
15
16
17 '***************** Member-Variablen *********************************************
18 Private m_sValue As String       'Diese Variable beinhaltet den zu speichernden
19                                  'Inhalt (Daten)
20 Private m_pNext As clsNodeD      'Zeiger auf den naechsten Knoten
21 Private m_pLast As clsNodeD      'Zeiger auf den vorhergehenden Knoten
22
23
24 '***************** Member-Funktionen ********************************************
25 'Public Property Let SetValue(ByVal sValue As String)
26 'Public Property Get GetValue() As String
27 'Public Property Set SetNext(ByVal pNext As clsNodeD)
28 'Public Property Get GetNext() As clsNodeD
29 'Public Property Set SetLast(ByVal pNext As clsNodeD)
30 'Public Property Get GetLast() As clsNodeD
31
32
33 '***************** Konstruktor **************************************************
34
35 '*******************************************************************************
36 '* Private Sub Class_Initialize():                                             *
37 '*                                                                             *
38 '* Aufgaben:                                                                   *
39 '*  - Die Member-Variablen initialisieren                                      *
40 '*                                                                             *
41 '* Veraenderte Member-Variablen:                                               *
42 '*  - m_sValue                                                                 *
43 '*  - m_pNext                                                                  *
44 '*  - m_pLast                                                                  *
45 '*******************************************************************************
46 Private Sub Class_Initialize()
47     'Daten initialisieren (hier mit einem Leerzeichen)
48     m_sValue = ""
49     'Beide Zeiger initialisieren
50     Set m_pNext = Nothing
51     Set m_pLast = Nothing
52 End Sub
53
54
```

```
55 '***************** Destruktor ******************************************************
56
57 '**********************************************************************************
58 '* Private Sub Class_Terminate():                                                  *
59 '*                                                                                 *
60 '* Aufgabe:                                                                        *
61 '*  Hat hier keine Aufgabe!                                                        *
62 '*                                                                                 *
63 '* Veraenderte Member-Variablen:                                                   *
64 '*  keine                                                                          *
65 '**********************************************************************************
66 Private Sub Class_Terminate()
67 End Sub
68
69
70 '***************** Member-Funktionen (Realisierung) *********************************
71 Public Property Let SetValue(ByVal sValue As String)
72     m_sValue = sValue
73 End Property
74
75
76 Public Property Get GetValue() As String
77     GetValue = m_sValue
78 End Property
79
80
81 Public Property Set SetNext(ByVal pNext As clsNodeD)
82     Set m_pNext = pNext
83 End Property
84
85
86 Public Property Get GetNext() As clsNodeD
87     Set GetNext = m_pNext
88 End Property
89
90
91 Public Property Set SetLast(ByVal pLast As clsNodeD)
92     Set m_pLast = pLast
93 End Property
94
95
96 Public Property Get GetLast() As clsNodeD
97     Set GetLast = m_pLast
98 End Property
```

5.2 Klasse `clsLinkedListD`

Die Klasse `clsLinkedListD` (Listing 5.2) ist auch hier wieder etwas umfangreicher. Hier wurden wieder alle wichtigen Funktionen mit sämtlichen Fällen, so wie sie in Kapitel 2 (ab Seite 5) beschrieben wurde, realisiert, mit Ausnahme des Scrollens. Listing 5.2 zeigt die Realisierung dieser Klasse.

Aufgrund der ausgiebigen Kommentar verzichte ich auch hier auf eine weitere Beschreibung. Für das Entfernen eines Knotens bzw. der gesamten Liste gilt auch hier dass Visual Basic ein dynamisch erzeugtes Objekt erst dann vom Speicher löscht, wenn keine Referenzen (Zeiger) auf dieses Objekt zeigen.

Listing 5.2: clsLinkedListD.cls

```
'*****************************************************************************
'* Klasse clsLinkedListD        (clsLinkedListD.cls)                         *
'*                                                                           *
'* Kurzbeschreibung                                                          *
'*  Klasse fuer eine doppelt verkettete Liste (engl. linked list)            *
'*                                                                           *
'* Aenderungen/Ergaenzungen                                                  *
'*                                                                           *
'* Entwickler: Buchgeher Stefan                                              *
'* Entwicklungsbeginn dieser Klasse: 28. Februar 2007                        *
'* Funktionsfaehig seit: 28. Februar 2007                                    *
'* Letzte Bearbeitung: 6. Maerz 2007                                         *
'*****************************************************************************
Option Explicit


'***************** Member-Variablen ******************************************
Private m_pRoot As clsNodeD       'Zeiger auf den ersten Knoten der Liste
Private m_pEnd As clsNodeD        'Zeiger auf den letzten Knoten der Liste
Private m_pActuell As clsNodeD    'Zeiger auf den aktuellen Knoten der Liste


'***************** Member-Funktionen *****************************************
'Public Function AddNode(ByVal sValue As String) As Integer
'Public Function ShowList(ListBox As Object) As Integer
'Public Function RemoveNode() As Integer
'Public Function RemoveList() As Integer


'***************** Konstruktor ***********************************************

'*****************************************************************************
'* Private Sub Class_Initialize():                                           *
'*                                                                           *
'* Aufgaben:                                                                 *
'*  - Die Member-Variablen mit 0 (= Nothing) initialisieren                  *
'*                                                                           *
'* Veraenderte Member-Variablen:                                             *
'*  - m_pRoot                                                                *
'*  - m_pEnd                                                                 *
'*  - m_pActuell                                                             *
'*****************************************************************************
Private Sub Class_Initialize()
    Set m_pRoot = Nothing
    Set m_pEnd = Nothing
    Set m_pActuell = Nothing
End Sub


'***************** Destruktor ************************************************

'*****************************************************************************
'* Private Sub Class_Terminate():                                            *
'*                                                                           *
'* Aufgabe:                                                                  *
'*  Die gesamte Liste (falls vorhanden) vom Speicher entfernen.              *
'*                                                                           *
'* Veraenderte Member-Variablen:                                             *
'*  - m_pRoot                                                                *
'*  - m_pEnd                                                                 *
'*  - m_pActuell                                                             *
'*****************************************************************************
Private Sub Class_Terminate()
    Dim ErrorCode As Integer


    ErrorCode = RemoveList()
End Sub


```

```
71  '****************** Member-Funktionen (Realisierung) **********************************
72
73  '*************************************************************************************
74  '* Public Function AddNode(sValue As String) As Integer:                             *
75  '*                                                                                   *
76  '* Aufgabe:                                                                          *
77  '*  Neuen Knoten (mit dem String sValue) ans Ende der bestehenden Liste anhaengen.   *
78  '*                                                                                   *
79  '* Uebergabeparameter:                                                               *
80  '*  sValue: Dieser Wert (String) soll in den neu hinzugefuegten Knoten gespeichert   *
81  '*          werden.                                                                  *
82  '*                                                                                   *
83  '* Rueckgabewert:                                                                    *
84  '*  - Fehlercode:   0 ... kein Fehler                                                *
85  '*                  1 ... Es kann kein Knoten hinzugefuegt werden, weil kein Speicher *
86  '*                        alloziiert werden kann!                                    *
87  '*                                                                                   *
88  '* Vorgehensweise:                                                                   *
89  '*  - Speicherplatz fuer den neuen Knoten dynamisch anfordern.                       *
90  '*  - Die zu speichernden Daten im neuen Knoten (pNewNode) speichern (mit Hilfe der  *
91  '*    Memberfunktion SetValue der Klasse clsNode)                                    *
92  '*  - Das Verketten der Knoten ist nun abhaengig davon, an welcher Position sich der *
93  '*    neue Knoten befindet:                                                          *
94  '*                                                                                   *
95  '*      - FALL 1: Der neue Knoten (pNewNode) befindet sich am Anfang der Liste, es ist *
96  '*        daher auch das erste Element der Liste                                     *
97  '*        (Kennzeichen: m_pRoot = 0 (Nothing)                                        *
98  '*        Da es das erste Element der Liste ist gilt fuer diesen Knoten:             *
99  '*           - Der Zeiger auf den vorhergehende Knoten muss den Wert 0 (Nothing)     *
100 '*             beinhalten                                                            *
101 '*           - Der Zeiger auf das naechste Element muss ebenfalls den Wert 0 (Nothing) *
102 '*             beinhalten                                                            *
103 '*           - Die (Member-)Zeiger m_pRoot, m_pEnd und m_pActuell zeigen auf diesen  *
104 '*             ersten Knoten.                                                        *
105 '*                                                                                   *
106 '*      - FALL 2: Der neue Knoten (pNewNode) befindet sich am Ende der Liste,        *
107 '*        (Kennzeichen: m_pActuell.GetNext = 0 (Nothing)                             *
108 '*        Da es der letzte Knoten der Liste ist gilt fuer diesen Knoten:             *
109 '*           - Der Zeiger von pNewNode auf den naechsten Knoten muss den Wert 0      *
110 '*             (Nothing) beinhalten                                                  *
111 '*           - Der Zeiger von pNewNode auf den vorhergehenden Knoten muss den Zeiger *
112 '*             Zeiger pActuell beinhalten                                            *
113 '*           - Der Zeiger von pActuell auf den naechste Knoten muss den Wert des     *
114 '*             Zeigers pNewNode beinhalten                                           *
115 '*           - Die (Member-)Zeiger m_pEnd und m_pActuell zeigen auf diesen letzten   *
116 '*             Knoten.                                                               *
117 '*                                                                                   *
118 '*      - FALL 3: Der neue Knoten befindet sich nicht am Ende der Liste, sondern     *
119 '*        mittendrin                                                                 *
120 '*        Da es ein Knoten innerhalb der Liste ist gilt fuer dieses Knoten:          *
121 '*           - Den auf pActuell folgenden Knoten in pTemp sichern                    *
122 '*           - Den neuen Knoten pNewNode nun so mit pActuell und pTemp verketten, dass *
123 '*             es sich anschliessend zwischen pActuell und pTemp befindet            *
124 '*           - Verkettung zwischen pActuell und pNewNode herstellen                  *
125 '*           - Verkettung zwischen pNewNode und pTemp herstellen                     *
126 '*           - Der (Member-)Zeiger m_pActuell zeigt auf den eingefuegten Knoten      *
127 '*                                                                                   *
128 '* Veraenderte Member-Variablen:                                                     *
129 '*  - m_pRoot                                                                        *
130 '*  - m_pEnd                                                                         *
131 '*  - m_pActuell                                                                     *
132 '*************************************************************************************
133 Public Function AddNode(sValue As String) As Integer
134     Dim pNewNode As clsNodeD
135
136     Dim pTemp As clsNodeD        'Hilfszeiger, fuer Fall 3 notwendig
137
138
139     'Knoten erzeugen
140     Set pNewNode = New clsNodeD
141
```

```
142         'Daten im  Knoten speichern
143         pNewNode.SetValue = sValue
144
145         'Verketten
146         If m_pRoot Is Nothing Then
147             'Fall 1: Erster Knoten der Liste
148             Set pNewNode.SetNext = Nothing
149             Set pNewNode.SetLast = Nothing
150             Set m_pRoot = pNewNode
151             Set m_pEnd = pNewNode
152         Else
153             If m_pActuell.GetNext Is Nothing Then
154                 'Fall 2: Knoten am Ende der Liste anhaengen
155
156                 'Die Knoten m_pActuell und pNewNode miteinander verbinden (Neuen Knoten
157                 'an das Ende der Liste, also an den "aktuellen" Knoten anhaengen
158                 Set pNewNode.SetNext = Nothing
159                 Set pNewNode.SetLast = m_pActuell
160
161                 Set m_pActuell.SetNext = pNewNode
162
163                 Set m_pEnd = pNewNode
164             Else
165                 'Fall 3: Knoten irgendwo zwischen Anfang und Ende
166
167                 'Adresse des auf pActuell folgenden Knoten in pTemp zwischenspeichern
168                 Set pTemp = m_pActuell.GetNext
169
170                 'Den neuen Knoten zwischen pActuell und pTemp einfuegen (verketten)
171                 'pActuell <-> pNewNode
172                 Set m_pActuell.SetNext = pNewNode
173                 Set pNewNode.SetLast = m_pActuell
174
175                 'pNewNode <-> pTemp
176                 Set pNewNode.SetNext = pTemp
177                 Set pTemp.SetLast = pNewNode
178             End If
179         End If
180
181         'Zeiger m_pActuell auf den neuen Knoten referenzieren
182         Set m_pActuell = pNewNode
183
184         'Rueckgabewert
185         AddNode = 0      'kein Fehler
186     End Function
187
188
189     '*****************************************************************************
190     '* Public Function ShowList(ListBox As Object) As Integer:                    *
191     '*                                                                            *
192     '* Aufgabe:                                                                   *
193     '*  - Die gesamte Liste in der uebergebenen List-Box anzeigen                 *
194     '*                                                                            *
195     '* Uebergabeparameter:                                                        *
196     '*  - Listbox (Objekt)                                                        *
197     '*                                                                            *
198     '* Rueckgabewert:                                                             *
199     '*  - Fehlercode:   0 ... kein Fehler                                         *
200     '*                  1 ... Es ist keine Liste vorhanden                        *
201     '*                                                                            *
202     '* Vorgehensweise:                                                            *
203     '*  - ListBox loeschen                                                        *
204     '*  - Ist die Liste leer (m_pRoot = 0) den Text "Leer!" in die Listbox schreiben und  *
205     '*    den entsprechenden Fehlercode zurueckgeben                              *
206     '*  - Andernfalls mit Hilfe eines Hilfszeigers (hier pShowNode) und einer Schleife be- *
207     '*    ginnend beim ersten Knoten dessen Inhalt mit Hilfe von GetValue in die Listbox  *
208     '*    schreiben.                                                              *
209     '*                                                                            *
210     '* Veraenderte Member-Variablen:                                              *
211     '*  - keine                                                                   *
212     '*****************************************************************************
```

```
213 Public Function ShowList(ListBox As Object) As Integer
214     Dim pShowNode As clsNodeD
215
216
217     Set pShowNode = m_pRoot
218
219     ListBox.Clear
220     If pShowNode Is Nothing Then
221         ListBox.AddItem "Leer!"
222
223         'Rueckgabewert = Fehler
224         ShowList = 1
225     Else
226         While Not pShowNode Is Nothing
227             ListBox.AddItem pShowNode.GetValue
228             Set pShowNode = pShowNode.GetNext
229         Wend
230
231         'Rueckgabewert = kein Fehler
232         ShowList = 0
233     End If
234 End Function
235
236
237 '**********************************************************************************
238 '* Public Function RemoveNode() As Integer:                                       *
239 '*                                                                                *
240 '* Aufgabe:                                                                       *
241 '*  Den Knoten auf welches der (Member-) Zeiger (m_pActuell) zeigt, aus der Liste ent- *
242 '*  fernen, und den Speicherplatz fuer diesen Knoten wieder freigeben.            *
243 '*                                                                                *
244 '* Uebergabeparameter:                                                            *
245 '*  - keiner                                                                      *
246 '*                                                                                *
247 '* Rueckgabewert:                                                                 *
248 '*  - Fehlercode:   0 ... kein Fehler                                             *
249 '*                  1 ... Es ist kein Knoten vorhanden                            *
250 '*                                                                                *
251 '* Vorgehensweise:                                                                *
252 '*  - Pruefen, ob ueberhaupt schon ein Eintrag existiert (m_pRoot > Nothing)      *
253 '*  - Ist kein Knoten vorhanden (m_pRoot = Nothing), den entsprechenden Fehlercode dem *
254 '*    aufrufendem Programm zurueckgeben.                                          *
255 '*  - Ist zumindest ein Knoten vorhanden (m_pRoot > 0) den Knoten, auf den der    *
256 '*    Zeiger pActuell zeigt loeschen. Beim Entfernen (loeschen) eines Knotens muss *
257 '*    zwischen 4 Faellen unterschieden werden:                                    *
258 '*                                                                                *
259 '*      - FALL 1: Die gesamte Liste besteht nur aus einem einzigen Knoten. Dieser *
260 '*        Knoten soll nun geloescht werden.                                       *
261 '*        Kennzeichen: m_pActuell.GetLast = Nothing                               *
262 '*                     m_pActuell.GetNext = Nothing                               *
263 '*          - Die (Member-)Zeiger m_pRoot, m_pEnd und m_pActuell mit Nothing laden, da *
264 '*            die Liste nun wieder leer ist. Dadurch wird der einzige zu loeschende *
265 '*            Knoten aus dem Speicher geloescht.                                  *
266 '*                                                                                *
267 '*      - FALL 2: Die Liste besteht aus mehreren Knoten, davon wird der erste Knoten *
268 '*        geloescht.                                                              *
269 '*        Kennzeichen: m_pActuell.GetLast = Nothing                               *
270 '*                     m_pActuell.GetNext ungleich Nothing                        *
271 '*          - Den (Member-)Zeiger m_pActuell auf den auf m_pActuell folgenden Knoten *
272 '*            setzen.                                                             *
273 '*          - Den Zeiger auf den vor dem aktuellen Knoten (m_pActuell) zeigenden mit 0 *
274 '*            (Nothing) laden.                                                    *
275 '*          - Den (Member-)Zeiger m_pRoot mit m_pActuell laden, da die Liste nun einen *
276 '*            neuen Beginn (Wurzel) besitzt.                                      *
277 '*          - Es zeigt keine Referenz mehr auf den ersten Knoten der Liste. Dieser *
278 '*            Knoten wird daher von Visual Basic automatisch vom Speicher entfernt. *
279 '*                                                                                *
280 '*      - FALL 3: Die Liste besteht aus mehreren Knoten, davon wird der letzte Knoten *
281 '*        geloescht.                                                              *
282 '*        Kennzeichen: m_pActuell.GetLast ungleich Nothing                        *
283 '*                     m_pActuell.GetNext = Nothing                               *
```

```
284 '*          - Den (Member-)Zeiger m_pActuell auf den von m_pActuell vorhergehenden     *
285 '*            Knoten setzen.                                                          *
286 '*          - Den vom aktuellen Knoten (m_pActuell) aus auf den naechsten Knoten      *
287 '*            zeigenden Zeiger mit 0 (Nothing) laden, da es keinen nachfolgenen Knoten *
288 '*            gibt, bzw. der aktuelle Knoten auch gleichzeitig der letzte Knoten ist.  *
289 '*          - Den (Member-)Zeiger m_pEnd mit m_pActuell laden, da die Liste nun ein    *
290 '*            neues Ende besitzt.                                                     *
291 '*          - Es zeigt keine Referenz mehr auf den letzten Knoten der Liste. Dieser    *
292 '*            Knoten wird daher von Visual Basic automatisch vom Speicher entfernt.    *
293 '*                                                                                    *
294 '*      - FALL 4: Die Liste besteht aus mehreren Knoten, davon wird weder der erste    *
295 '*        Knoten noch der letzte Knoten geloescht. Sondern ein Knoten irgendwo         *
296 '*        zwischen Ersten und Letztem.                                                *
297 '*        Kennzeichen: m_pActuell.GetLastEntry ungleich Nothing                       *
298 '*                     m_pActuell.GetNextEntry ungleich Nothing                       *
299 '*          - Die Adresse des auf m_pActuell folgenden Knotens in pTempNach sichern,   *
300 '*            und den zu m_pActuell vorherigen Eintrag in pTempVor sichern.            *
301 '*          - Alle Zeiger, die vom zu entfernenden Knoten weggehen ebenfalls loeschen  *
302 '*          - Die Knoten pTempVor und pTempNach miteinander verketten.                 *
303 '*          - Den (Member-)Zeiger m_pActuell mit pTempVor laden                        *
304 '*          - Es zeigt keine Referenz mehr auf den letzten Knoten der Liste. Dieser    *
305 '*            Knoten wird daher von Visual Basic automatisch vom Speicher entfernt.    *
306 '*                                                                                    *
307 '* Anmerkung zum Entfernen von referenzierten Objekten:                               *
308 '*    Visual Basic entfernt automatisch allozierte Objekte, wenn keine Referenz auf    *
309 '*    dieses Objekt zeigt. Anders ausgedrueckt: Wenn alle Referenzen auf ein dynamisch *
310 '*    erzeugtes Objekt den Wert Nothing beinhalten, entfernt Visual Basic automatisch  *
311 '*    dieses Objekt und gibt den Speicher wieder frei.                                 *
312 '*                                                                                    *
313 '* Veraenderte Member-Variablen:                                                      *
314 '*  - m_pRoot                                                                         *
315 '*  - m_pEnd                                                                          *
316 '*  - m_pActuell                                                                      *
317 '*************************************************************************************
318 Public Function RemoveNode() As Integer
319     Dim pTempVor As clsNodeD
320     Dim pTempNach As clsNodeD
321
322
323     Set pTempVor = New clsNodeD      'Fuer Fall 4 (das zu loeschende Entry befindet sich
324     Set pTempNach = New clsNodeD     'in der Mitte der Liste)
325
326     If m_pRoot Is Nothing Then
327
328         'Wenn kein Knoten vorhanden ist (m_pRoot = Nothing)
329         RemoveNode = 1 ' Rueckgabewert: Fehler
330     Else
331         If m_pActuell.GetLast Is Nothing Then
332             If m_pActuell.GetNext Is Nothing Then
333                 'Fall 1: Die Liste besteht nur aus einem einzigen Knoten
334                 'Dieser Knoten wird nun geloescht, der Speicherplatz fuer
335                 'diesen Knoten wird wieder freigegeben und die Zeiger m_pRoot,
336                 'm_pActuell und m_pEnd werden geloescht
337                 Set m_pRoot = Nothing
338                 Set m_pActuell = Nothing
339                 Set m_pEnd = Nothing
340             Else
341                 'Fall 2: Die Liste besteht aus mehreren Knoten.
342                 'Der erste Knoten wird nun geloescht, und der Speicherplatz fuer
343                 'dieses Knoten wird wieder freigegeben
344
345                 'Zeiger m_pActuell auf den naechsten Knoten setzen
346                 Set m_pActuell = m_pActuell.GetNext
347
348                 'Den Zeiger m_pActuell.SetLast mit Nothing laden, da ja nun
349                 'dieser Knoten das erste Entry der Liste ist
350                 Set m_pActuell.SetLast = Nothing
351
352                 'Den Zeiger m_pRoot mit m_pActuell laden, da die Liste nun einen
353                 'neuen Beginn (Wurzel) besitzt
354                 Set m_pRoot = m_pActuell
```

```
355
356                     'Es zeigt keine Referenz mehr auf den ersten Knoten der Liste. Dieser
357                     'Knoten wird daher von Visual Basic automatisch vom Speicher entfernt.
358                 End If
359             Else
360                 If m_pActuell.GetNext Is Nothing Then
361                     'Fall 3: Die Liste besteht aus mehreren Knoten.
362                     'Der letzte Knoten wird nun geloescht, und der Speicherplatz
363                     'fuer diesen Knoten wird wieder freigegeben
364
365                     'Zeiger m_pActuell auf den vorherige Knoten setzen
366                     Set m_pActuell = m_pActuell.GetLast
367
368                     'Den Zeiger m_pActuell->SetNext mit NULL laden, da ja nun
369                     'dieser Knoten der letzte Knoten der Liste ist.
370                     Set m_pActuell.SetNext = Nothing
371
372                     'Den Zeiger m_pEnd mit m_pActuell laden, da die Liste nun
373                     'ein neues Ende besitzt
374                     Set m_pEnd = m_pActuell
375
376                     'Es zeigt keine Referenz mehr auf den letzten Knoten der Liste. Dieser
377                     'Knoten wird daher von Visual Basic automatisch vom Speicher entfernt.
378                 Else
379                     'Fall 4: Die Liste besteht aus mehreren Knoten.
380                     'Es wird weder der erste noch der letzte Knoten geloescht,
381                     'sondern ein Knoten irgendwo in der Mitte der Liste, der
382                     'Speicherplatz fuer diesen Knoten wird wieder freigegeben.
383
384                     'Die Adresse des vorhergehenden Knotens in pTempVor sichern,
385                     'und die Adresse des naechsten Knotens in pTempNach sichern.
386                     '(Anm.: Bezueglich m_pActuell)
387                     Set pTempVor = m_pActuell.GetLast
388                     Set pTempNach = m_pActuell.GetNext
389
390                     'Alle Zeiger, die vom zu entfernenden Knoten weggehen ebenfalls
391                     'loeschen
392                     Set m_pActuell.SetLast = Nothing
393                     Set m_pActuell.SetNext = Nothing
394
395                     'Die Knoten pTempVor und pTempNach verketten
396                     Set pTempVor.SetNext = pTempNach
397                     Set pTempNach.SetLast = pTempVor
398
399                     'Den Zeiger m_pActuell mit pTempVor laden
400                     Set m_pActuell = pTempVor
401
402                     'Es zeigt keine Referenz mehr auf den ersten Knoten der Liste. Dieser
403                     'Knoten wird daher von Visual Basic automatisch vom Speicher entfernt.
404                 End If
405             End If
406
407             RemoveNode = 0 'Rueckgabewert: alles okay
408         End If
409     End Function
410
411
412     '*****************************************************************************
413     '* Public Function RemoveList() As Integer:                                  *
414     '*                                                                           *
415     '* Aufgabe:                                                                  *
416     '*  - Die gesamte Liste vom Speicher entfernen                               *
417     '*                                                                           *
418     '* Uebergabeparameter:                                                       *
419     '*  - keiner                                                                 *
420     '*                                                                           *
421     '* Rueckgabewert:                                                            *
422     '*  - Fehlercode:    0 ... kein Fehler                                       *
423     '*                   1 ... Es ist keine Liste vorhanden                      *
424     '*                                                                           *
425     '* Vorgehensweise:                                                           *
```

```
426 '*  - Ist die Liste leer (m_pRoot = Nothing) muss sie auch nicht geloescht werden!! -  *
427 '*    Was nicht vorhanden ist, kann auch nicht geloescht werden!!                       *
428 '*  - Andernfalls mit Hilfe zweier Hilfszeiger (hier pRemoveNode und pRemoveNext)       *
429 '*    beginnend beim ersten Knoten diesen mit Hilfe einer Schleife Knoten fuer Knoten   *
430 '*    vom Speicher entfernen. Dabei zeigt der Zeiger pRemoveNode immer auf den zu ent-  *
431 '*    fernenden Knoten und der Zeiger pRemoveNext auf den darauf folgenden Knoten.      *
432 '*    Dieser Vorgang wird solange fortgesetzt, bis pRemoveNext ein "Null-Zeiger" ist,   *
433 '*    D.h. der Zeiger den Wert Nothing beinhaltet. Denn dann ist er am Ende der Liste   *
434 '*    angekommen.                                                                       *
435 '*                                                                                      *
436 '* Anmerkung zum Entfernen von referenzierten Objekten:                                 *
437 '*    Visual Basic entfernt automatisch allozierte Objekte, wenn keine Referenz auf     *
438 '*    dieses Objekt zeigt. Anders ausgedrueckt: Wenn alle Referenzen auf ein dynamisch  *
439 '*    erzeugtes Objekt den Wert Nothing beinhalten, entfernt Visual Basic automatisch   *
440 '*    dieses Objekt und gibt den Speicher wieder frei.                                  *
441 '*                                                                                      *
442 '* Veraenderte Member-Variablen:                                                        *
443 '*  - m_pRoot                                                                           *
444 '*  - m_pEnd                                                                            *
445 '*  - m_pActuell                                                                        *
446 '****************************************************************************************
447 Public Function RemoveList() As Integer
448     Dim pRemoveNode As clsNodeD
449     Dim pRemoveNext As clsNodeD
450
451
452     'Liste nur dann loeschen, wenn eine vorhanden ist!
453     If m_pRoot Is Nothing Then
454          'Wenn keine Liste vorhanden ist (m_pRoot = nothing)
455         RemoveList = 1 ' Rueckgabewert: Fehler
456     Else
457         'Zeiger pRemoveNode auf den ersten Knoten setzen
458         Set pRemoveNode = m_pRoot
459
460         'alle moeglichen Referenzen auf den ersten Knoten loeschen
461         Set m_pRoot = Nothing
462         Set m_pEnd = Nothing
463         Set m_pActuell = Nothing
464
465         'Mit einer Schleife alle Knoten loeschen
466         Do
467             'Zeiger pRemoveNext auf den naechsten Knoten setzen
468              Set pRemoveNext = pRemoveNode.GetNext()
469
470             'Auch den Zeiger, der vom zu loeschenden Knoten weggeht loeschen
471             Set pRemoveNode.SetNext = Nothing
472
473             'Und auch den Zeiger, der vom naechsten Knoten auf den zu loeschenden
474             'zurueck zeigt loeschen, wenn es noch einen naechsten Knoten gibt
475             If Not pRemoveNode.GetNext Is Nothing Then
476                 Set pRemoveNext.SetLast = Nothing
477             End If
478
479             'Knoten auf den der Zeiger pRemoveNode zeigt loeschen
480             Set pRemoveNode = Nothing
481
482             'Zeiger pRemoveNode auf den naechsten Knoten setzen
483             Set pRemoveNode = pRemoveNext
484         Loop Until pRemoveNode Is Nothing
485
486         'Rueckgabewert = Liste erfolgreich geloescht (kein Fehler)
487         RemoveList = 0
488     End If
489 End Function
```

5.3 Demonstrationsbeispiel

Auch hier ist das Demonstrationsbeispiel wieder sehr einfach. Dazu wird ein einfaches Formular (hier: `frmLinkedListD`) mit den folgenden (Standard-)Steuerelemente benötigt:

- 1 TextBox (Name: `txtNeueDaten`)
- 1 ListBox (Name: `lstListe`)
- 4 CommandButton (Namen: `cmdNeueDaten`, `cmdDatenLoeschen`, `cmdListeAnzeigen` und `cmdListeLoeschen`)
- Optional: 2 Frames (ohne Namen)

Abbildung 5.2 zeigt die Anordnung dieser Steuerelemente.

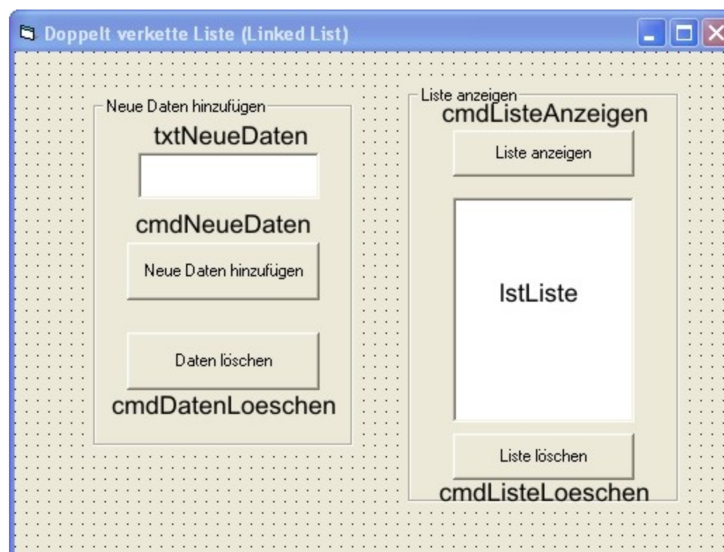

Abbildung 5.2: Steuerelemente für eine doppelt verkettete Liste (Visual Basic)

So einfach wie das Formular ist, so einfach ist auch der Quellcode der dahintersteckt. Listing 5.3 zeigt das relativ kurze Programm.

Listing 5.3: frmLinkedListD.frm

```
1  '*******************************************************************************
2  '* Demonstrationsprojekt zur doppelt verketteten Liste (mit der Klasse clsLinkedListD) *
3  '*                                                                             *
4  '* Aenderungen/Ergaenzungen                                                    *
5  '*                                                                             *
6  '* Entwickler: Buchgeher Stefan                                                *
7  '* Entwicklungsbeginn dieser Klasse: 28. Februar 2007                          *
8  '* Funktionsfaehig seit: 28. Februar 2007                                      *
9  '* Letzte Bearbeitung: 6. Maerz 2007                                           *
10 '*******************************************************************************
11 Option Explicit
12
13
14 'Instanz der verketteten Liste definieren
15 Dim DemoListe As clsLinkedListD
16
```

```
17
18  Private Sub Form_Load()
19     'Eine Instanz der (Doppelt) verketteten Liste erstellen
20      Set DemoListe = New clsLinkedListD
21  End Sub
22
23
24  ' Neuen Knoten (mit dem Inhalt der Textbox) ans Ende der Liste anhaengen
25  Private Sub cmdNeueDaten_Click()
26      Dim ErrorCode As Integer
27
28      'Diesen Eintrag in die Liste hinzufuegen
29      ErrorCode = DemoListe.AddNode(Me.txtNeueDaten.Text)
30  End Sub
31
32
33  ' Gesamte Liste in einer Listbox anzeigen
34  Private Sub cmdListeAnzeigen_Click()
35      Dim ErrorCode As Integer
36
37      'Die gesamte Liste in der Listbox (lstListe) anzeigen
38      ErrorCode = DemoListe.ShowList(Me.lstListe)
39  End Sub
40
41
42  ' aktuellen Knoten loeschen
43  Private Sub cmdDatenLoeschen_Click()
44      Dim ErrorCode As Integer
45
46      'Die gesamte Liste in der Listbox (lstListe) anzeigen
47      ErrorCode = DemoListe.RemoveNode
48  End Sub
49
50
51  'Den Inhalt der List-Box (lstListe) loeschen
52  Private Sub cmdListeLoeschen_Click()
53      Me.lstListe.Clear
54  End Sub
```

Auch hier nur kurz die wesentlichen Punkte:
In der Ereignisprozedur `Form_Load` muss eine Instanz der Verketteten Liste erzeugt werden (Zeilen 18 bis 21).
Das Hinzufügen eines neuen Knotens erfolgt durch Anklicken der Taste `cmdNeueDaten` (Zeilen 25 bis 30), und das Anzeigen der gesamten Liste durch Anklicken der Taste `cmdListeAnzeigen` (Zeilen 34 bis 39).
Neu in diesem Demonstrattionsprojekt ist, dass der letzte Knoten der Liste gelöscht werden kann. Dies erfolgt mit der Taste `cmdDatenLoeschen` (Zeilen 43 bis 48).
Achtung: Die Taste `cmdListeLoeschen` löscht nur den Inhalt der List-Box `lstListe`. Sie entfernt nicht die verkettet Liste vom Speicher (Zeilen 52 bis 54)!

Abbildung 5.3 zeigt das erfolgreich kompilierte Programm in Aktion.

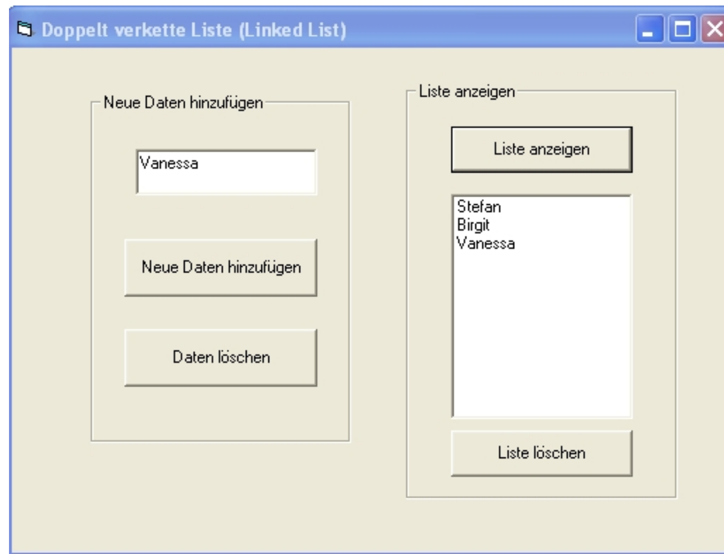

Abbildung 5.3: Einfaches Demo für eine doppelt verkettete Liste in Visual Basic

Literaturverzeichnis

[1] Niklaus Wirth. *Algorithmen und Datenstrukturen.* Teubner Verlag, Stuttgart/Leipzig/Wiesbaden, 1998.

[2] Jesse Liberty. *C++ in 21 Tagen.* Markt+Technik Verlag, München.

[3] Peter Monadjemi. *Jetzt lerne ich Visual Basic.* Markt+Technik Verlag, München, 2003.

[4] Walter Doberenz und Thomas Kowalsi. *Visual Basic 6 - Grundlagen und Profiwissen.* Carl Hanser Verlag München Wien, 2003.

[5] Rudolf Huttary. *Visual Basic 6 Referenz.* Markt+Technik Verlag, München, 2000.

[6] Dieter Otter. `http://www.vbarchiv.net/` *(Stand: 21. Februar 2007).*