Z E I G E R

Verkettete Listen und binäre Bäume

ZEIGERVERKETTETE LISTEN UND BÄUME

[01] Zeiger und dynamische Variable (*zeiger*)

Angenommen, wir befinden uns in einem großen Büroraum. Der Büroleiter erteilt dem Kanzleigehilfen zwei Aufträge:

Befehl 1: Hole die Akte "MÜLLER".
Befehl 2: Hole die oberste Akte der dritten Schublade aus dem vierten Kasten.

Der erste Befehl setzt offenbar 2 Dinge voraus: die Deklaration einer Akte als "MÜLLER" und die Kenntnis ihres Aufbewahrungsortes. Das entspricht in DELPHI einer wohldefinierten Variablen durch Designation und Typisierung. Dadurch wird im Datenspeicher ein Bereich mit bestimmter Größe und Struktur ab einer bestimmten Startadresse festgelegt.

Im zweiten Befehl hingegen ist der Name der Akte unbekannt. Auf diese Akte wird zugegriffen durch einen Verweis (Referenz) auf ihre Adresse. Solche Variable heißen "Anonyme Variable". Der zweite Befehl selbst entspricht einem ZEIGER (Pointer) auf die Adresse dieser Variablen. Erst durch das Setzen eines Zeigers wird der Zugriff auf die anonyme Variable ermöglicht. Diese wird in dem Zusammenhang auch als dynamische Variable bezeichnet. Für dynamische (anonyme) Variable steht der gesamte freie Arbeitsspeicher des Computers zur Verfügung, der auch als HEAP bezeichnet wird.

Um es exakt zu formulieren: Statische Variablen müssen im Definitionsteil eines Programmes festgelegt werden. Der für sie reservierte Speicherbereich kann später nicht mehr freigegeben werden. Dynamische Variablen werden erst zur Laufzeit des Programmes mit Hilfe von Zeigern erzeugt und können auch wieder vernichtet werden. Dadurch wird eine flexible und ökonomische Nutzung des Speichers ermöglicht.

Ein Zeiger selbst ist eine statische Variable im Datenspeicher. Die Länge einer solchen ZEIGER-Variablen beträgt genau vier Bytes (32 Bits), welche die Adresse einer Bezugsvariablen (z.B. einer dynamischen, anonymen Variablen am HEAP) beinhalten. Es muss sowohl der Typ der Bezugsvariablen als auch der Typ der zugehörigen Zeigervariablen definiert werden. Soll der Zeiger **P** auf eine Stringvariable verweisen, dann lautet seine Definition (auch "Referenzierung" genannt):

var P : ^String;

Die Bezugsvariable von **P** ist dann ein String, der am Heap liegt und wird im Programmverlauf mittels **P^** angesprochen wird. Der unitäre Referenzoperator ^ (Dach) bedeutet: "Durch einen Zeiger vermittelt". Es gelten dafür folgende zwei Notationsregeln:

(a) Steht er vor einem Datentyp, dann bezeichnet er einen Zeiger (^**Typname**).
(b) Steht er hinter einer Zeigervariablen (**Zeiger^**), dann bezeichnet er die entsprechende Bezugsvariable des Zeigers und er liefert jenen Wert, der an jener Adresse gespeichert ist, welche der Zeiger angibt. Das wird auch "Dereferenzierung" des Zeigers genannt.

Wie wird nun eine Bezugsvariable im Anweisungsteil eines Programmes erzeugt? Ganz einfach durch den Befehl ***New(P)***. Dadurch wird am Heap ein Speicherbereich belegt, dessen Größe durch die Typendeklaration von P vorgegeben ist, und außerdem wird der Zeigervariablen P die Startadresse dieses Bereiches zugewiesen. Der Programmbefehl ***P^ := 'Herbert'*** schreibt dann den Text 'Herbert' in diesen Speicherbereich.

Wie wird eine solche anonyme Bezugsvariable wieder vernichtet? Ganz einfach durch den Befehl ***Dispose(P).*** Damit wird der durch ***New(P)*** reservierte Speicherbereich am Heap freigegeben und steht somit anderweitig zur Verfügung. Durch einen neuerlichen ***New***-Befehl wird dieser Bereich überschrieben.

```
type Punkt = record
               X : Real;
               Y : Real;
             end;
var A: Punkt;
    P: ^Punkt;

begin
  New(P);
  P^.X := -2.50;
  P^.Y :=  7.25;
  A := P^;
  Dispose(P);
end;
```

Neben ***New*** und ***Dispose*** werden auch ***GetMem*** und ***FreeMem*** zur dynamischen Speicherverwaltung am Heap verwendet. ***GetMem(Zeiger,ByteAnzahl)*** reserviert unter dem Zeiger-Namen eine bestimmte Anzahl von Bytes am Heap. ***FreeMem(Zeiger,ByteAnzahl)*** hingegen gibt diesen Speicher wieder frei. Im nachfolgenden Beispiel gibt die systemdefinierte Funktion ***SizeOf*** die Anzahl der Bytes an, die für eine bestimmte Variable im Speicher reserviert sind.

```
begin
  GetMem(P,SizeOf(Punkt));
  P^.X  := -2.50;
  P^.Y  :=  7.25;
  FreeMem(P,SizeOf(Punkt));
end;
```

Zeiger müssen aber nicht nur auf dynamische Bezugsvariable am Heap verweisen, sondern können sich auch direkt auf schon voher definierte Variablen beziehen. Dazu muss ihnen nur explizit die Adresse dieser Variablen zugewiesen werden. Dies geschieht mit der Funktionsanweisung ADDR. (Dafür kann auch der Adressoperator @ verwendet werden.)

```
var S: String;
    Z: Integer;
    P: ^String;
    Q: ^Integer;

begin
  P  := Addr(S);
  P^ := 'Herbert';
  Q  := @Z;
  Q^ :=  16;
end;
```

Durch obige Anweisungen werden über dem Umweg von Zeigern den Variablen bestimmte Werte zugewiesen, was natürlich mit S := 'Herbert' und Z := 16 schneller geht.

Wird mit Dispose oder FreeMem versucht, einen Speicherbereich freizugeben, der vorher nicht mit New oder GetMem angefordert (alloziert) wurde, dann führt das zu einem schweren Fehler!

DELPHI stellt zusätzlich einen untypisierten Zeiger zur Verfügung, der mit allen Bezugstypen kompatibel ist. Die Zuweisung P := NIL (Not in List) bedeutet, dass der Zeiger P (noch) auf keine Bezugsvariable verweist. Seine Adresse ist daher Null. Außerdem gibt es einen, im System vordefinierten Zeiger ***PChar***, der automatisch auf Stringzeichen (Char) verweist.

Das Programm "***zeiger***" demonstriert die Verwendung von Zeigervariablen.

[02] Grundlagen von Listen-Strukturen (*list01*)

Angenommen, ein Datenbestand von 200 Stringelementen zu je 80 Zeichen liegt vor und soll alphabetisch sortiert werden. Dafür gibt es verschiedene Lösungsmöglichkeiten.

(1) Die Datenelemente werden in einem statischen Array im Datenspeicher abgelegt und dort mit Hilfe eines Sortierverfahrens sortiert.

```
type Feld : array[1..200] of String[80];
var  Data : Feld;

begin
  Data[1] := 'Herbert';
  ........
end;
```

(2) Man definiert für jedes Datenelement eine Zeigervariable und speichert diese Zeiger in einem statischen Array im Datenspeicher. Die Bezugsdaten der Zeiger liegen anonym am Heap und die Sortierung erfolgt dann mit Hilfe des Zeiger-Arrays.

```
type PType : ^String;
var  PFeld : array[1..200] of PType;

begin
  New(PFeld[1]);
  PFeld[1]^ := 'Herbert';
  ........
end;
```

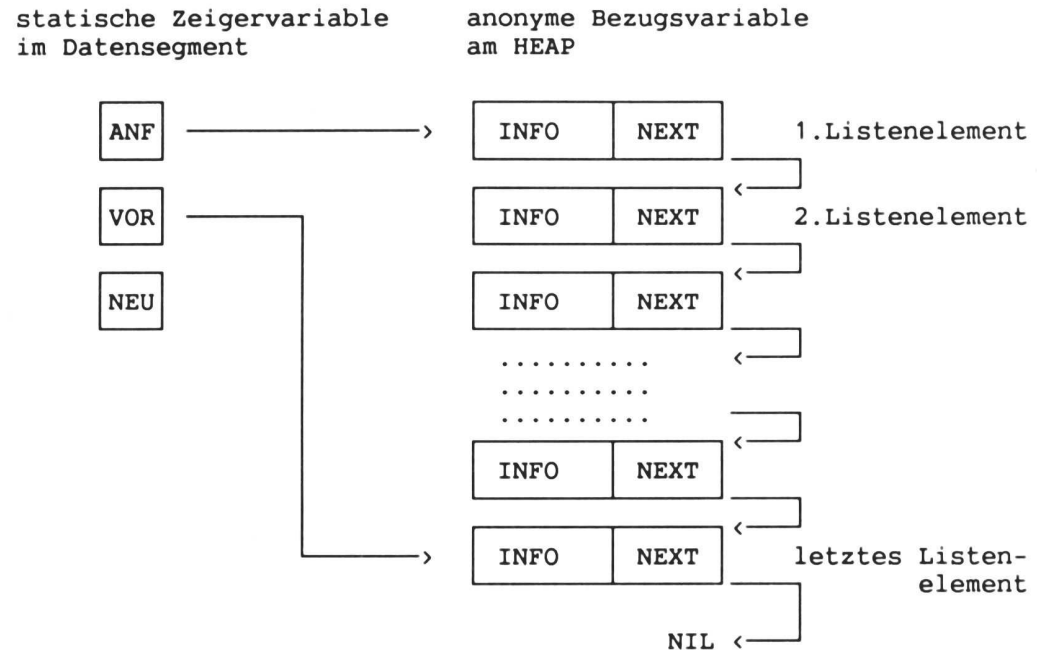
Die zweite Vorgangsweise hat den Vorteil, dass am Heap für die dynamischen Datenelemente mehr Speicherplatz zur Verfügung steht und bei der Sortierung mittels Zeiger nicht die langen Datenelemente, sondern nur die kurzen Zeiger vertauscht werden. Das bedeutet eine schnellere Ausführungszeit. Trotz der augenscheinlichen Vorteile des zweiten Lösungsweges ist auch hier die Speicherausnutzung nicht optimal. Ein großer Datenbestand kann zwar problemlos am Heap erzeugt werden, jedoch ist er dadurch begrenzt, dass das statische Array der zugehörigen Zeigervariablen beschränkt ist.

Im Folgenden soll **ein dritter Lösungsansatz** dargestellt werden, der zu einer optimalen Speicherausnutzung führt. Die dabei aufgebaute Datenstruktur wird eine zeigerverkettete, lineare Liste genannt (Linked List). Die dynamischen Datenelemente werden als Records typisiert, welche im einfachsten Fall aus zwei Komponenten bestehen: aus einer String-Komponente *INFO*, welche der eigentliche Informationsträger ist, und aus einer Zeiger-Komponente *NEXT*, die auf das nachfolgende dynamische Datenelement verweist, d.h. dessen Speicheradresse enthält.

```
type Zeiger = ^Element;
     Element = record
                 Info : String;
                 Next : Zeiger;
               end;

var  P,Q : Zeiger;

begin
  New(P);
  New(Q);
  P^.Info := 'Herbert';
  P^.Next := Q;                 //  P^.Next zeigt auf Q
  Q^.Info := 'Susanne';
  Q^.Next := NIL;               //  Q^.Next zeigt auf NIL
  ........
end;
```

Interessant ist die Tatsache, dass der Zeiger *NEXT* auf einen Record zeigt, in dessen Struktur-Definition er selbst steht und welche noch nicht abgeschlossenen ist. Solche Typendefinitionen heißen "Vorwärts-Deklarationen".

Eine ***Linked List*** besteht nun aus mehreren solchen anonymen Recordvariablen am Heap, die über ihre Zeigerkomponenten miteinander in Verbindung stehen. Zur ökonomischen Verwaltung einer solchen Liste werden am besten drei statische Zeigervariablen eingesetzt (*ANF*, *VOR* und *NEU*). *ANF* zeigt auf das erste Listenelement, *NEU* zeigt auf das aktuelle Listenelement und *VOR* auf dessen Vorgänger bzw. das vorläufige Listenende. Alle Listenelemente sind dynamisch am Heap abgespeichert.



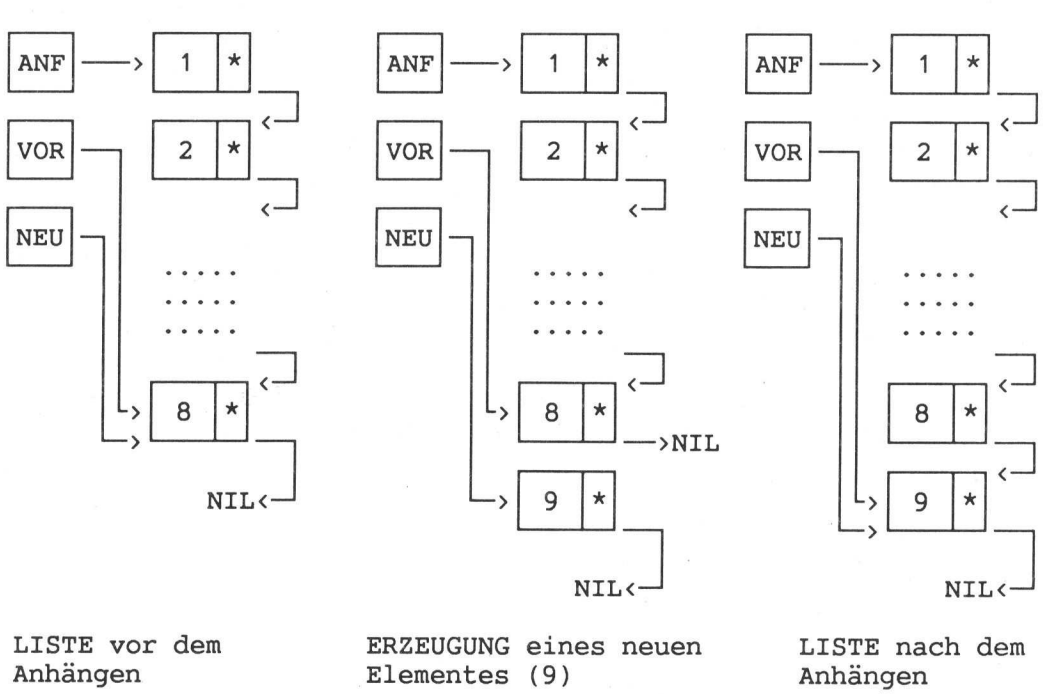
Grafische Darstellung einer einfach zeigerverketteten, offenen Liste.

Das Kernstück der Listenerzeugung ist das Anhängen eines neuen Datenelementes, was im nachfolgenden Listing durch die Prozedur *Eingeben* erreicht wird.

```
var ANF,VOR: Zeiger;
    EIN: String;

procedure Eingeben(var ANF,VOR: Zeiger; EIN: String);
var NEU: Zeiger;
begin
  New(NEU);
  NEU^.Info := EIN;
  NEU^.Next := NIL;
  if (ANF = NIL) then ANF := NEU
                 else VOR^.Next := NEU;
  VOR := NEU;
end;

begin
  ANF := NIL;
  VOR := NIL;
  EIN := 'Herbert';
  Eingeben(ANF,VOR,EIN);
end;
```

Der Zeigerkomponente des letzten Listenelementes *VOR* ist dabei immer auf NIL gesetzt.

Der Prozess des Anfügens kann grafisch in drei Phasen dargestellt werden.



Die Ausgabe einer Linked List ist denkbar einfach. Beginnend beim Zeiger *ANF* wird die Liste so lange durchklettert, bis NIL erreicht ist. Das Kernstück ist die Zuweisung *NEU := NEU^.NEXT*. Im nachfolgenden Listing werden die Daten der Liste zeilenweise in einer Memokomponente (*Memo2*) des Formulares ausgegeben.

```
procedure Ausgeben(ANF: Zeiger);
var NEU: Zeiger;
begin
  NEU := ANF;
  While NEU <> NIL do begin
    Form1.Memo2.Lines.Add(NEU^.Info);
    NEU := NEU^.Next;
  end;
end;
```

Das Programm "***list01***" behandelt einfach zeigerverkettete, offene Listen.

[03] Doppelt verkettete, geschlossene Listen (*list02*)

Im Gegensatz zu einer offenen Liste verweist bei einer geschlossenen Liste das letzte Element nicht auf NIL, sondern zurück auf das erste Listenelement. Dadurch ist eine Ringform entstanden.

Die Listenelemente des Programmes "***list02***" enthalten neben der *INFO*-Komponente noch zwei Zeiger, wobei *PREV* auf das vorangehende Element und *NEXT* auf das nachfolgende verweist. Der Vorteil einer solchen doppelt verketteten Liste liegt nun darin, dass im Unterschied zur einfach verketteten Liste von jedem Listenelement aus ohne Schwierigkeit auch auf den jeweiligen Vorgänger zugegriffen werden kann.

```
type Zeiger = record
                Info : String;
                Prev : Zeiger;
                Next : Zeiger;
              end;
```

Beim Anlegen einer solchen Liste ist zu beachten, dass jetzt immer zwei Zeiger entsprechend gesetzt werden müssen. Am Ende muss der Ring ordnungsgemäß geschlossen werden, indem das letzte Element auf das erste verweist (*NEU^.NEXT := ANF*) und das erste auf das letzte zurückzeigt (*ANF^.PREV := NEU*). Jetzt ist ein eigener Zeiger *VOR* auf das Listenende nicht mehr nötig, weil Anfang und Ende der geschlossenen Liste der Zeiger *ANF* ist. Eine doppelt zeigerverkettete Liste kann nun ohne Schwierigkeit in beiden Richtungen durchlaufen und ausgegeben werden. Nachfolgendes Listing beschreibt das *Eingeben*, *Ausgeben* und *Freigeben* der Liste.

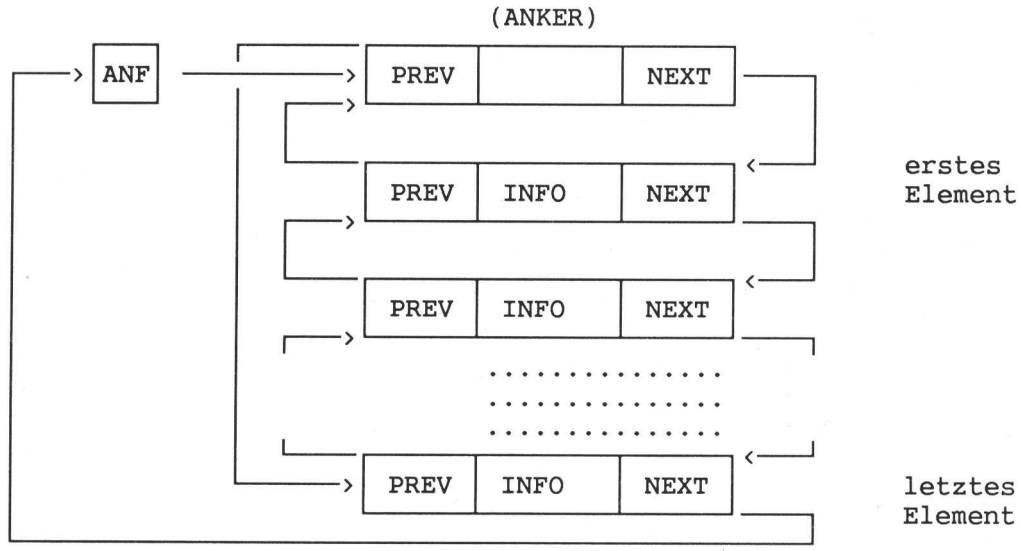
```
var ANF: Zeiger;
    EIN: String;

procedure Eingeben(var ANF: Zeiger; EIN: String);
var NEU: Zeiger;
begin
  New(NEU);
  NEU^.Info := EIN;
  NEU^.Prev := NIL;
  NEU^.Next := NIL;
  if (ANF = NIL) then ANF := NEU
  else begin
    NEU^.Next := ANF;
    NEU^.Prev := ANF^.Prev;
    ANF^.Prev^.Next := NEU;
    ANF^.Prev := NEU;
  end;
end;

procedure Ausgeben_1(ANF: Zeiger);
var NEU: Zeiger;
begin
  NEU := ANF;
  repeat
    Form1.Memo2.Lines.Add(NEU^.Info);
    NEU := NEU^.Next;
  until NEU = ANF;
end;

procedure Ausgeben_2(ANF: Zeiger);
var NEU: Zeiger;
begin
  NEU := ANF;
  repeat
    NEU := NEU^.Prev;
    Form1.Memo2.Lines.Add(NEU^.Info);
  until NEU = ANF;
end;

procedure Freigeben(var ANF: Zeiger);
var VOR,NEU: Zeiger;
begin
  VOR := ANF;
  while VOR.Next <> ANF do begin
     NEU := VOR^.Next;
     Dispose(VOR);
     VOR := NEU;
  end;
end;

// Hauptprogramm
begin
  ANF := NIL;
  EIN := 'Herbert';
  Eingeben(ANF,EIN);
  EIN := 'Susanne';
  Eingeben(ANF,EIN);
  Ausgeben(ANF);
  Freigeben(ANF);
end;
```

[04] Vollständige Verwaltung von Listen (*list03*)

Im Programm "***list03***" wird eine komplette Listen-Verwaltung durchgeführt. Dabei werden alle wichtigen Verwaltungsroutinen demonstriert:

- ANLEGEN - EINGEBEN - SORTIEREN - SUCHEN - LÖSCHEN - AUSGEBEN -

Die ***Linked List*** besteht aus einer linearen, doppelt verketteten, geschlossenen Liste mit Anker. Das Ankerelement besteht aus Zeichen mit niedrigstem druckbaren ASCII-Code (Leerzeichen) und dient nur der Verankerung einer neu angelegten Liste. Auf das Ankerelement verweist immer der Zeiger *ANF*. Die Struktur einer doppelt verketteten, geschlossenen Liste mit Anker hat folgendes Aussehen:



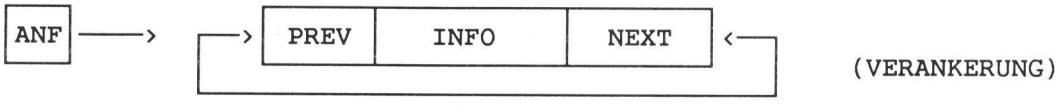
Im Allgemeinem genügen drei Zeiger *ANF*, *VOR* und *NEU*, um in einer solchen Liste sämtliche Verwaltungsroutinen durchzuführen. Der Zeiger *ANF* verweist immer auf das Ankerelement.

```
const Anker = #32;

type  Zeiger = ^Element;
      Element = record
                  Info : String;
                  Prev : Zeiger;
                  Next : Zeiger;
                end;

var   ANF,VOR,NEU: Zeiger;
```

Im Folgenden sollen die wichtigsten Verwaltungsroutinen kurz beschrieben werden:

(a) ANLEGEN der Liste

Dabei wird nur das Ankerelement erzeugt, und zwar mit Zeigerkomponenten, die rückbezüglich auf das Ankerelement selbst verweisen.

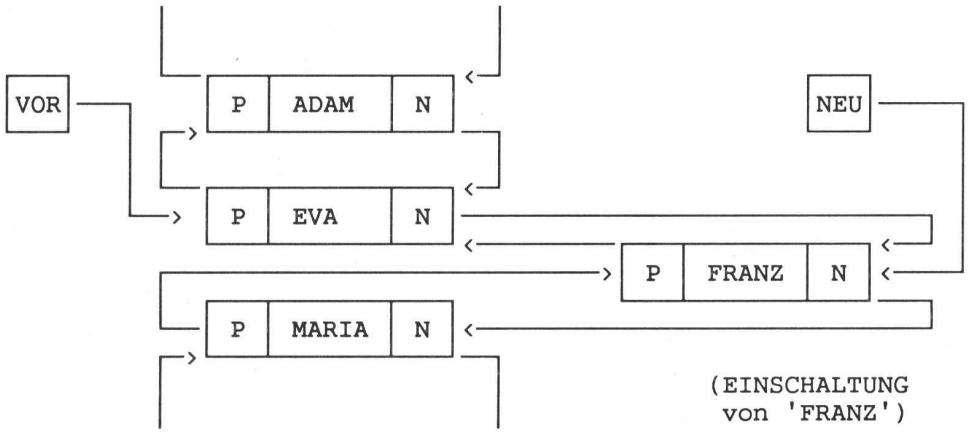


```
procedure Anlegen(var ANF: Zeiger);
// Ankerelement der Liste anlegen
begin
  New(ANF);
  ANF^.Info := Anker;
  ANF^.Prev := ANF;
  ANF^.Next := ANF;
end;
```

(b) EINGEBEN von neuen Datenelementen

Das neue Datenelement, bestimmt durch den Zeiger *NEU*, wird dabei zwischen dem letzten Listenelement (*ANF.^Prev*) und dem Ankerelement *ANF* eingehängt. Zu diesem Zweck müssen die beteiligten Zeiger entsprechend verbogen werden.

Die folgende Abbildung zeigt das allgemeines Schema des Einhängen in eine Liste, und zwar hinter jener Stelle, auf die der Zeiger *VOR* gerade verweist. Das neue Element wird dabei durch den Zeiger *NEU* bestimmt.




```
procedure Eingeben(var ANF: Zeiger; EIN: String);
// Einschalten eines neuen Knotens NEU zwischen ANF^.Prev und ANF
var NEU: Zeiger;
begin
  if ANF = NIL then Anlegen(ANF);
  New(NEU);
  NEU^.Info := EIN;
  NEU^.Prev := ANF^.Prev;
  NEU^.Next := ANF;
  ANF^.Prev^.Next := NEU;
  ANF^.Prev := NEU;
end;
```

(c) SUCHEN eines Listenelementes

Beim Suchen eines Datenelementes wird die Liste, beginnend beim ersten Element *ANF^.Next*, über die Zeigerkomponenten der einzelnen Elemente so lange durchklettert, bis die entsprechende Infokomponente gefunden wurde oder wieder das erste Listenelement erreicht ist. Das folgende Schema des Kletterns ist so aufgebaut, dass auch auf den Vorgänger des gefundenen Listenelementes zugegriffen werden kann (*VOR*). Das ist deswegen wichtig, weil dann an dieser Stelle problemlos eine Einschaltung oder eine Entfernung vorgenommen werden kann. Die Variable EIN soll dabei der gesuchte String sein.

```
VOR := ANF^.Next;
while (EIN <> VOR^.Info) and (VOR<>ANF) do VOR := VOR^.Next;
```

Wenn in obigem Schema *VOR* wieder *ANF* ist, dann wurde die ganze Liste ohne Erfolg durchlaufen und der Suchvorgang bricht erfolglos ab.

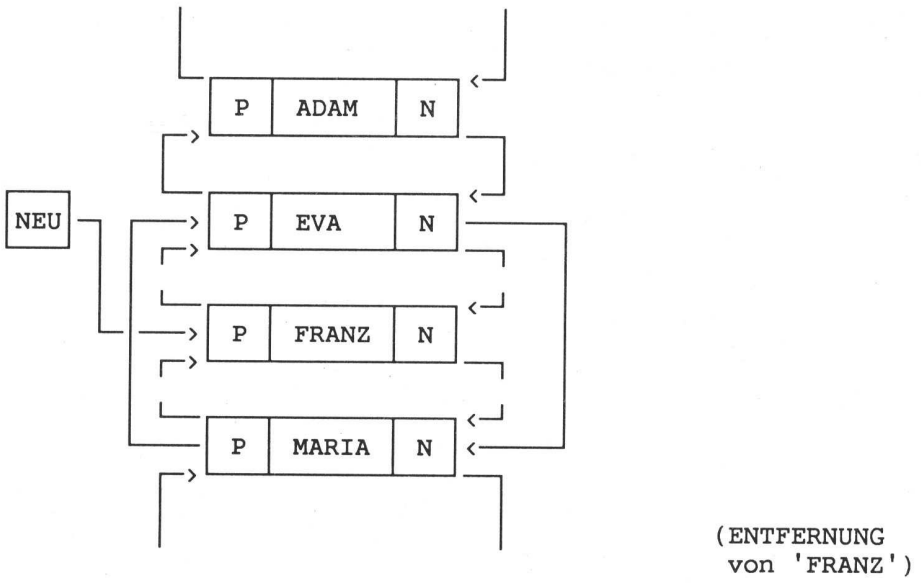
Im nachfolgenden Listing übergibt die Funktion *Suchen* bei erfolgreicher Suche den Zeiger auf das gesuchte Listenelement, andernfalls NIL.

```
function Suchen(ANF: Zeiger; EIN: String): Zeiger;
// Suchen in der Liste nach dem Schlüsselbegriff EIN
var VOR,NEU : Zeiger;
begin
  if ANF = NIL then Exit;
  VOR:= ANF^.Next;
  while (EIN <> VOR^.Info) and (VOR <> ANF) do VOR := VOR^.Next;
  NEU := VOR;
  if NEU <> ANF then Result := NEU
                else Result := NIL;
end;
```

(d) LÖSCHEN eines Listenelementes

Beim Entfernen eines Elementes aus der Liste muss dieses zuerst gesucht werden. Um das Element aus der Liste zu entfernen, müssen nur sein Vorgänger und sein Nachfolger kurzgeschlossen werden.

```
NEU^.Prev^.Next := NEU^.Next;
NEU^.Next^.Prev := NEU^.Prev;
Dispose(NEU);
```




```
function Loeschen(ANF: Zeiger; EIN: string): Zeiger;
// Listenelement löschen, d.h. Kurzschließen von NEU^.Prev und NEU^.Next
var NEU: Zeiger;
begin
  if ANF = NIL then Exit;
  NEU := Suchen(ANF,EIN);
  Result := NEU;
  if NEU <> NIL then begin
     NEU^.Prev^.Next := NEU^.Next;
     NEU^.Next^.Prev := NEU^.Prev;
     Dispose(NEU);
  end;
end;
```

(e) AUSGEBEN von Listenelementen

Bei der Ausgabe der Liste wird diese nur einmal ringförmig durchlaufen.

```
procedure Ausgeben(ANF: Zeiger);
// Die ganze Liste vorwärts ausgeben
var NEU : Zeiger;
begin
  if ANF = NIL then Exit;
  NEU := ANF;
  repeat
    NEU := NEU^.Next;
    Form1.Memo2.Lines.Add(NEU^.Info);
  until NEU = ANF;
end;
```

(f) SORTIEREN der Liste

Unsere Liste soll zunächst unsortiert eingegeben werden. Natürlich könnte man das Programm so gestalten, dass bei jeder Eingabe eines neuen Elementes die Liste durchklettert wird und das Element an die richtige Stelle eingeschaltet wird (Sortieren beim Eingeben). Das könnte sehr einfach mit Hilfe eines Ankers realisiert werden.

Ich habe bewusst die Liste beim Eingeben nicht bereits sortiert, weil dadurch eine sehr interessante Aufgabenstellung entsteht, wenn man die Liste im Nachhinein sortieren will. Im Folgenden soll eine Lösung vorgeschlagen werden, bei der aus der alten unsortierten Liste schrittweise eine neue, sortierte Liste erzeugt wird.

Mittels des Hilfzeigers *HELP* werden der alten, unsortierten Liste schrittweise die einzelnen Elemente entnommen und in eine neue Liste einsortiert. Ausgangspunkt ist dabei immer der Zeiger *ANF*, der unveränderlich auf das Ankerelement verweist. Am Ende liegt dann eine neue, sortierte Zeigerverkettung vor.

```
procedure Sortieren(var ANF: Zeiger);
// Die ganze Liste neu sortieren
var VOR,NEU,HELP : Zeiger;
begin
    if ANF = NIL then Exit;
    NEU := ANF^.Next;
    ANF^.Prev := ANF;
    ANF^.Next := ANF;
    repeat
       HELP := NEU^.Next;
       ;
       VOR  := ANF;
       while (NEU^.Info > VOR^.Next^.Info) and (VOR^.Next <> ANF) do
              VOR := VOR^.Next;
       NEU^.Prev := VOR;
       NEU^.Next := VOR^.Next;
       VOR^.Next^.Prev := NEU;
       VOR^.Next := NEU;
       ;
       NEU := HELP;
    until NEU=ANF;
end;
```

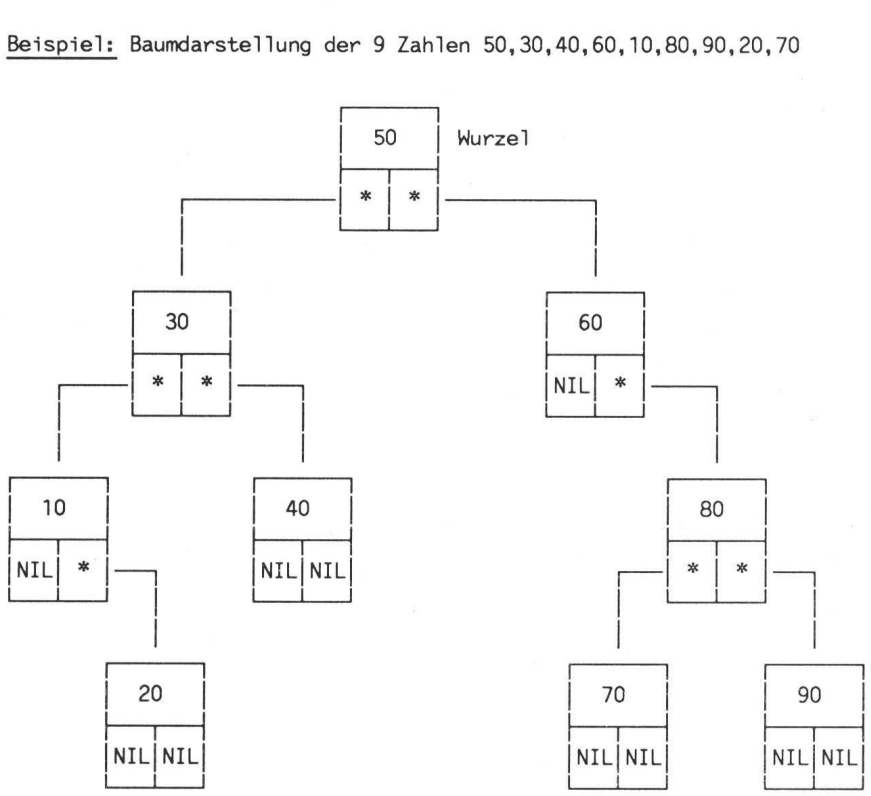
(g) FREIGEBEN der Liste

Am Abschluss des Programmes muss die ganze Liste aus dem Speicher entfernt werden, damit der Speicherplatz wieder zur Verfügung steht. Das entsprechende Listing wurde bereits im Abschnitt [14.03] besprochen.

[05] Grundlagen von Baum-Strukturen (*baum01*)

Sowie die ***Linked List*** gehört auch der ***Binäre Baum*** zu den zeigerverketteten dynamischen Datenstrukturen. Während eine Liste aber linear angeordnet ist, so ist ein Baum hierarchisch gegliedert. Jedes Baumelement (Knoten) enthält neben dem Datenteil (Schlüssel) mindestens zwei Zeiger - einen auf den linken (L) und einen auf den rechten Nachfolger (R).

Ein direkter Nachfolger wird als Sohn, ein direkter Vorgänger als Vater bezeichnet. Der oberste Knoten im Baum heißt Wurzel. Die Knoten heißen Blätter. Der Pfad eines Knotens ist der Weg von der Wurzel bis zum Knoten. Die Pfadlänge ist dann die Anzahl der auf diesem Weg passierten Knoten. Unter der Baumhöhe versteht man die längste Pfadlänge. Die Endknoten auf den Baumästen haben NIL als Zeigerkomponenten.



Unter einem Unterbaum versteht man alle Knoten, die von einem Ausgangsknoten erreicht werden können. Von einem Knoten ausgehend, kann man somit einen linken Unterbaum (LUB) und einen rechten Unterbaum unterscheiden (RUB). Beginnend bei der Wurzel kann der Baum als rekursive Struktur "***LUB - Knoten - RUB***" aufgefasst werden. Das rekursive Bildungsgesetz, nach dem der binäre Baum aufgebaut ist, kann unschwer mit folgender Formel ausgedrückt werden:

Alle Schlüssel im LUB sind kleiner/gleich und alle Schlüssel im RUB sind größer als der Schlüssel im Knoten selbst.

Wenn in einem Baum ein Vater drei Söhne hat, dann spricht man von einem triären Baum. In diesem Kapitel werden nur binäre Bäume behandelt.

Die Knoten des oben dargestellten Baumes haben genau zwei Zeiger auf ihre beiden Nachfolger. Dadurch kann der Baum immer nur von oben nach unten durchlaufen werden, d.h., ein direkter Zugriff von einem Knoten auf seinen Vorgänger ist nicht möglich.

Um den Baum auf direktem Weg von unten nach oben zu durchlaufen, kann zu jedem Knoten noch ein zusätzlicher Zeiger (*V*) installiert werden, der auf den Vater verweist. Dann spricht man von einer Zweirichtungsverkettung, was natürlich den Speicherbedarf erhöht.

Unterscheiden sich die Knotenanzahlen im linken und rechten Unterbaum unter jedem Baumknoten um höchstens eins, dann spricht man von einem ausbalancierten Baum. Ein vollständig unbalancierter, rechtslastiger Baum entsteht, wenn man eine bereits vorsortierte Datenfolge ablegt. Der Baum ist dann eine linear verkettete Liste. Um die Entartung eines Baumes in eine Liste zu vermeiden, wird man beim Aufbau trachten, diesen möglichst symmetrisch anzulegen, was natürlich von der Wahl des Wurzelknotens abhängt. Zwei russische Mathematiker, Adelson-Velskii und Landis, haben ein Verfahren entwickelt, welches einen binären Baum ausbalanciert (AVL-Bäume).

Das Programm "***baum01***" zeigt eine grafische Darstellung eines binären Baumes.

[06] Vollständige Verwaltung von Bäumen (*baum02*)

Die grundlegenden Operationen in Bäumem sind:

EINFÜGEN eines Knotens (Einsortieren)
AUSGABE des Baumes (Durchklettern)
SUCHEN eines Knotens
ENTFERNEN eines Knotens

Das Programm "***baum02***" demonstriert die komplette Verwaltung von binären Bäumen. Ein Baumelement wird dabei durch einen Rekord definiert:

```
type Zeiger  = ^Element;
     Element = record
                  Info: string;
                  L,R : Zeiger;
               end;
```

Zur Verwaltung des Baumes genügen vier statische Zeiger und eine boolesche Variable:

```
var WURZEL,ANF,VOR,NEU: ZEIGER;
    Gefunden: Boolean;
```

NEU und *ANF* verweisen dabei auf beliebige Knoten, *VOR* zeigt auf den direkten Vorgänger eines Knotens und *WURZEL* verweist auf den Ausgangsknoten des Baumaufbaues.

(a) EINFÜGEN eines Knotens

Am Anfang wird das erste Datenelement als Wurzelknoten genommen, auf das der Zeiger *WURZEL* verweist. Dabei werden die beiden Zeigerkomponenten *WURZEL^.L* und *WURZEL^.R* auf NIL gesetzt. Bei der Eingabe der nachfolgenden Elemente geht man mit Hilfe des so genannten "Abwärtsklettern" folgendermaßen vor:

Wenn der Schlüssel des neuen Elementes kleiner oder gleich dem Schlüssel der *WURZEL* ist, dann wird im linken Baumast abgestiegen. Bei jedem Knoten wird nach links verzweigt, wenn der neue Schlüssel kleiner/gleich dem Knotenschlüssel ist, andernfalls wird nach rechts abgebogen. Erst wenn ein Endknoten erreicht ist, wird das Datenelement eingesetzt, d.h., der erreichte Endknoten wird nun zum Vater des neuen Elementes, welches selbst zum Endknoten wird. Zu diesem Zweck müssen die beteiligten Zeiger entsprechend umgeändert werden. Dieser Prozess des Einfügens wird am besten rekursiv programmiert.

```
procedure Einsort(var ANF: Zeiger; NEU: Zeiger);
// Sortiert den Knoten NEU in den binären Baum richtig ein
begin
  if ANF = NIL then ANF := NEU
  else if NEU^.Info <= ANF^.Info then Einsort(ANF^.L,NEU)
                                 else Einsort(ANF^.R,NEU);
end;

procedure Eingabe(var ANF: Zeiger; EIN: String);
// Fügt einen neuen Knoten mit dem Schlüsselbegriff EIN in den binären Baum
begin
  if ANF = NIL then begin
     New(ANF);
     ANF^.Info := EIN;
     ANF^.L := NIL;
     ANF^.R := NIL;
  end
  else begin
     New(NEU);
     NEU^.Info := EIN;
     NEU^.L := NIL;
     NEU^.R := NIL;
     Einsort(WURZEL,NEU);
  end;
end;
```

Zur Illustration soll im vorangehenden Baumbeispiel die neue Zahl 35 eingefügt werden:

35 < 50, also nach links.
35 > 30, also nach rechts.
35 < 40, also nach links. NIL erreicht, also 35 als neuen Endknoten einsetzen.

(b) AUSGABE des Baumes

Zum Zweck der Ausgabe der Knoten eines Baumes kann dieser in verschiedenster Weise durchklettert werden. Im Folgenden soll der binäre Baum so durchwandert werden, dass entsprechend dem Bildungsgesetz die Knoten auch sortiert ausgegeben werden. Beginnend mit der *WURZEL*, wird rekursiv die Prozedur *SORTAUS* aufgerufen. Es wird so lange nach links abwärts gestiegen, bis ein NIL erreicht wird. Dann wird zum letzten Knoten zurückgegangen und dieser ausgegeben. Anschließend wird nach rechts abgestiegen. Die Schlüsselwerte werden in einem Memo angezeigt.

```
procedure Sortaus(ANF: Zeiger);
// Gibt den gesamten binären Baum aus
begin
  if ANF <> NIL then begin
     Sortaus(ANF^.L);
     Form1.Memo2.Lines.Add(ANF^.Info);
     Sortaus(ANF^.R);
  end;
end;
```

(c) SUCHEN eines Knotens

Beim Suchen wird ebenfalls bei der Baumwurzel begonnen und genauso wie beim Einfügen abwärts geklettert. Wenn ein Knotenschlüssel mit dem Suchbegriff übereinstimmt, dann wird die boolesche Variable *GEFUNDEN* wahr und die Suche beendet. Andernfalls wird weitergeklettert bis ein NIL erreicht ist, worauf die Suche abgebrochen und *GEFUNDEN* auf falsch gesetzt wird. Die Prozedur *SUCHEN* realisiert dieses Durchsuchen mit Hilfe einer einfachen Iteration. Dabei ist *ANF* der Zeiger auf den jeweils erreichten Knoten und der mitgeführte Zeiger *VOR* verweist auf den *VATER* des Knotens. Das ist beim reinen Suchen unnötig, aber beim *ENTFERNEN* eines Knotens unbedingt erforderlich, weil ja zur Entfernung des *SOHNES* die Verbindung zwischen *VATER* und dessen *ENKEL* kurzgeschlossen werden muss.

```
procedure Suchen(var ANF: Zeiger; EIN: string);
// Durchsucht den binären Baum nach dem Schlüsselbegriff EIN
begin
  Gefunden := False;
  VOR := ANF;
  repeat
    if ANF^.Info = EIN then Gefunden := True
    else begin
       VOR := ANF;
       if ANF^.Info >= EIN then ANF := ANF^.L;
       if ANF^.Info <  EIN then ANF := ANF^.R;
    end;
  until Gefunden or (ANF=NIL);
end;
```

(d) ENTFERNEN eines Knotens

Zum Entfernen eines Knotens wird dieser zuerst mit der Prozedur *SUCHEN* gesucht. Falls er gefunden wird, kann er aus dem Baum entfernt werden. Am Ende der Suchprozedur verweist der Zeiger *ANF* auf den gefundenen Knoten und der Zeiger *VOR* auf seinen Vater. Der Kern des Entfernens eines Knotens ist immer der Kurzschluss zwischen *VATER* und *ENKELN*. Dadurch wird der Sohn aus der Zeigerverkettung herausgenommen. Er kann dann mittels *Dispose(ANF)* physisch vom Heap gelöscht werden. Dies muss nach der unten stehenden Prozedur *ENTFERNEN* erfolgen.

Beim Herausnehmen eines Knotens sind jetzt vier Fälle zu unterscheiden, welche in der Prozedur *ENTFERNEN* getrennt behandelt werden:

Fall 1: Der Knoten *ANF^* hat keine Söhne. Der Zeiger des Vaters, der auf *ANF^* verweist, wird auf NIL gesetzt.

Fall 2: Der Knoten *ANF^* hat nur einen linken Sohn. Der Zeiger des Vaters, der auf *ANF^* verweist, wird auf diesen Enkel gesetzt (*ANF^.L^*). Dadurch wird der Knoten aus der Verkettung ausgeschlossen.

Fall 3: Der Knoten *ANF^* hat nur einen rechten Sohn. Der Zeiger des Vaters, der auf *ANF^* verweist, wird auf diesen Enkel gesetzt (*ANF^.R^*). Dadurch wird der Knoten aus der Verkettung ausgeschlossen.

Fall 4: Der Knoten *ANF^* hat zwei Söhne. Der Zeiger des Vaters, der auf *ANF^* verweist, wird auf den linken Enkel *ANF^.L^* gesetzt. Der rechte Teilbaum von ANF^ wird an den linken Teilbaum von ANF^ gehängt, wobei rechts eine freie Stelle gesucht wird.

Der Sonderfall *ANF = WURZEL* wird jeweils extra abgefragt.

```
PROCEDURE ENTFERNEN(VAR WURZEL,ANF :ZEIGER);
// entfernt das Element ANF aus dem binären Baum
VAR HELP : ZEIGER;
BEGIN
  IF (ANF^.L=NIL) AND (ANF^.R=NIL)
     THEN IF ANF<>WURZEL
             THEN IF ANF=VOR^.L
                     THEN VOR^.L := NIL
                     ELSE VOR^.R := NIL
             ELSE WURZEL := NIL;
  IF (ANF^.L=NIL) AND (ANF^.R<>NIL)
     THEN IF ANF<>WURZEL
             THEN IF ANF=VOR^.L
                     THEN VOR^.L := ANF^.R
                     ELSE VOR^.R := ANF^.R
             ELSE WURZEL := ANF^.R;
```

```
  IF (ANF^.L<>NIL) AND (ANF^.R=NIL)
     THEN IF ANF<>WURZEL
             THEN IF ANF=VOR^.L
                     THEN VOR^.L := ANF^.L
                     ELSE VOR^.R := ANF^.L
             ELSE WURZEL := ANF^.L;
  IF (ANF^.L<>NIL) AND (ANF^.R<>NIL) THEN
     BEGIN IF ANF<>WURZEL
              THEN IF ANF=VOR^.L
                      THEN VOR^.L := ANF^.L
                      ELSE VOR^.R := ANF^.L
             ELSE WURZEL := ANF^.L;
           HELP := ANF^.L;
           WHILE HELP^.R<>NIL DO HELP:=HELP^.R;
           HELP^.R := ANF^.R;
     END;
END;
```

[07] Ein objektorientierter Stapelspeicher (*stack*)

Im Programm "***stack***" simuliert das Objekt STACK einen dynamischen Stapelspeicher. Auf diesem STACK werden die einfachen Record-Elemente NODE abgespeichert. Diese bestehen aus einer Nummer, einem String und einem Zeiger, welcher auf das vorangehende Record-Element verweist. Dadurch wird eine offene, rückwärts zeigerverkettete Liste gebildet.

Das Objekt STACK selbst enthält das Datenfeld TOP, welches einen Zeiger auf das immer zuletzt eingegebene Listenelement darstellt. Die Constructor-Methode INIT setzt TOP auf NIL, wodurch der STACK initialisiert wird. Die Methode PUSH fügt ein neues Listenelement an. Die Methode POP entfernt das zuletzt angefügte Element vom STACK. Dadurch wird ein dynamischer Stapel erzeugt, der nach dem Prinzip "***Last in - First out***" funktioniert.

Die Methode SHOW gibt sämtliche Listenelemente in einem Memofeld aus und die Methode FREE entfernt das ganze Objekt mitsamt seiner Liste aus dem Speicher.

```
unit stack_u;
// Objektorientierte zeigerverkettete Simulation eines Stackspeichers
// (c) H.Paukert

interface
uses Windows, Messages, SysUtils, Classes, Graphics,
     Controls, Forms, Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    Label1: TLabel;
    Memo1: TMemo;
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    Button4: TButton;
    procedure FormActivate(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure Button4Click(Sender: TObject);
    private { Private-Deklarationen }
    public  { Public-Deklarationen }
  end;

var  Form1: TForm1;
```

```
implementation
{$R *.DFM}

type NodePtr = ^Node;

     Node = record
       Number : Integer;
       Item   : String;
       Prev   : NodePtr;
     end;

     StackPtr = ^Stack;

     Stack = class(TObject)
       Top : NodePtr;
       constructor Init; VIRTUAL;
       procedure Push(S: String);
       function  Pop: String;
       procedure Show;
       destructor Free; VIRTUAL;
     end;

     constructor Stack.Init;
     // Initialisiert die Stackspitze mit NIL
     begin
       Top := NIL;
     end;

     procedure Stack.Push(S: String);
     // Fügt einen neuen Knoten an die Stackspitze
     var Node: NodePtr;
     begin
       New(Node);
       Node^.Item := S;
       Node^.Prev := Top;
       if Node^.Prev = NIL then Node^.Number := 1
                           else Node^.Number := Node^.Prev^.Number + 1;
       Top := Node;
     end;

     function Stack.Pop: String;
     // Entfernt den Knoten an der Stackspitze
     var Node: NodePtr;
     begin
       if Top <> NIL then begin
          Node := Top;
          Result := Top^.Item;
          Top := Top^.Prev;
          Dispose(Node);
       end
       else Result := ' Stack ist leer ';
     end;

     procedure Stack.Show;
     // Gibt den gesamten Stack in einem Memofeld aus
     var Node: NodePtr;
     begin
       Form1.Memo1.Clear;
       Form1.Memo1.Lines.Add(#32);
       Node := Top;
       while Node <> NIL do begin
         Form1.Memo1.Lines.Add(#9 + IntToStr(Node^.Number) + '. ' + Node^.Item);
         Node := Node^.Prev;
       end;
       Form1.Memo1.Lines.Add(#9 + '----------');
     end;
```

```
    destructor Stack.Free;
    // Entfernt den gesamten Stack aus dem Speicher
    var Node: NodePtr;
    begin
      while Top <> NIL do begin
        Node := Top;
        Top := Node^.Prev;
        Dispose(Node);
      end;
    end;

var aStack : Stack;

function HeapStatus: String;
// liefert den Heapstatus
var HS : THeapStatus;
begin
  HS := GetHeapStatus;
  Result := ' MemAlloc: ' + IntToStr(HS.TotalAllocated) + ' Bytes ';
end;

procedure TForm1.FormActivate(Sender: TObject);
// Stack INIT
begin
  aStack := Stack.Init;
  Label1.Caption := HeapStatus;
end;

procedure TForm1.Button1Click(Sender: TObject);
// Stack PUSH
var S: String;
begin
  S := InputBox('PUSH eines neuen Element','','Element');
  aStack.Push(S);
  aStack.Show;
  Label1.Caption := HeapStatus;
end;

procedure TForm1.Button2Click(Sender: TObject);
// Stack POP
var S : String;
begin
  S := aStack.Pop;
  aStack.Show;
  Label1.Caption := HeapStatus;
end;

procedure TForm1.Button3Click(Sender: TObject);
// Stack FREE and Stack INIT
begin
  aStack.Free;
  aStack := Stack.Init;
  aStack.Show;
  Label1.Caption := HeapStatus;
end;

procedure TForm1.Button4Click(Sender: TObject);
// Quit programm
begin
  Application.Terminate;
end;

end.
```