Leistungserhebung im Wahlpflichtfach Informatik in den Schuljahrgängen 10 - 12

Abstrakte Datentypen und ihre Implementierung

Dr. Henry Herper

Otto-von-Guericke-Universität Magdeburg
Institut für Simulation und Graphik

Lisa-Weiterbildung – 18.09.2006

Suchen und Sortieren von Daten

Zielstellung:

Vertiefung der Kenntnisse über die Datenverwaltung in einem Computer, Erlernen der programmiersprachenunabhängigen Gemeinsamkeiten von Datentypen und Erweiterung und Festigung der Programmierfertigkeiten am Beispiel eindimensionaler Datenstrukturen.

Struktur der Rahmenrichtlinie

SJG 10	SJG 11	SJG 12
Grundlagen der Informationstechnik Projektarbeit unter Nutzung von Standardsoftware	Algorithmen-strukturen und ihre Implementierung	Wahlthema
Informatik und Gesellschaft Computer-Netzwerke	Datenstrukturen Informatisches Modellieren	Projektarbeit zur Software-entwicklung

Struktur der Rahmenrichtlinie

SJG 10	SJG 11	SJG 12
Grundlagen der Informationstechnik Projektarbeit unter Nutzung von Standardsoftware	Algorithmen-strukturen und ihre Implementierung	**Wahlthema**
Informatik und Gesellschaft Computer-Netzwerke	Datenstrukturen Informatisches Modellieren	**Projektarbeit zur Software-entwicklung**

Wahlthemen im SJG 12/1

(1) Modellbildung und Simulation
(2) Analyse und Design eines Informatiksystems
(3) Computergraphik
(4) **Abstrakte Datentypen und ihre Implementierung**
(5) Suchen und Sortieren von Daten
(6) Endliche Automaten und formale Sprachen
(7) Kryptologie
(8) Datenbankanwendungen zur dynamischen Webseitengenerierung

Abstrakte Datentypen und ihre Implementierung - Vorleistungen

Schuljahrgang 11/2

Thema: Strukturierte Datentypen ZRW: 10 Std.
Vorbemerkungen/Qualifikationen

Im Rahmen dieses Themas werden Datentypen zur Verwaltung von komplexen Datenstrukturen eingeführt. Hierzu werden Methoden zur permanenten Verwaltung auf externen Datenträgern vermittelt.

Die Schülerinnen und Schüler

- kennen Strukturen zur Verwaltung komplexer Daten und zur rechnerinternen Repräsentation,
- können Daten in Dateien verschiedenen Typs verwalten.

Abstrakte Datentypen und ihre Implementierung - Vorleistungen

Thema: Strukturierte Datentypen

Inhalte	Hinweise zum Unterricht
– Strukturierte Datentypen und ihre Implementierung • Reihung (Felder) • Datenverbund (Datensatz) • Dateien	– indizierter Zugriff auf Felder – Beispiel: Minimum, Maximum, Mittelwert einer Messwertfolge – Erstellen von Datensätzen aus einem Applikationsgebiet
– Operationen auf Dateien • Speichern und Öffnen von Dateien • Suchen und Editieren von Daten • Fehlerinterpretation	– Zusammenwirken von Betriebssystem und Dateizugriffskonzept – Verwaltung von Datensätzen aus einem Applikationsgebiet – Navigieren in einer Datei

W4 – Abstrakte Datentypen und ihre Implementierung (12/1)

Zeitrichtwert : 26 Stunden

Vorbemerkungen/Qualifikationen

Im Rahmen dieses Themas werden ausgewählte Datentypen ausführlich behandelt. Zur formalen Beschreibung werden abstrakte Datentypen verwendet. Die Implementierung erfolgt mit einer geeigneten Programmiersprache.

Ziele des Themas

Die Schülerinnen und Schüler

- kennen den **Begriff des abstrakten Datentyps**,
- können auf der Basis von abstrakten Datentypen **Grundfunktionen auf die dynamische Datenstruktur Liste beschreiben und implementieren**,
- kennen die Spezialformen **Stapel und Warteschlange**.

Unterrichtsinhalte – Abstrakte Datentypen und ihre Implementierung

Inhalte	Hinweise zum Unterricht
Begriff und Eigenschaften des abstrakten Datentyps (ADT)	• Universalität • Geheimnisprinzip

Datentyp

Datentyp:
Ein Datentyp ist durch einen

- **Wertebereich**, die Menge der Werte, die ein Datenelement des entsprechenden Typs annehmen kann, durch
- **Operationen**, die auf Werten dieses Bereiches ausgeführt werden können und durch
- die **Eigenschaften** der erlaubten Operationen

definiert.

Aus den in der Mathematik definierten Mengen (z.B. N, R) werden idealisierte Datentypen abgeleitet. Bei der Implementierung in eine konkrete Programmiersprache wird eine Menge von daraus abgeleiteten, konkreten Datentypen bereitgestellt.

Nachteil: vordefinierte Datentypen sind zur Lösung eines Problems nicht immer ausreichend

Abstrakte Datentypen - Entstehung

Abstrakter Datentyp (ADT) ist eine Bezeichnung in der Informatik für Datentypen, um diese unabhängig von einer konkreten Implementierung zu spezifizieren. Sie wurden von Barbara Liskov, Stephen Zilles 1974 und von John Guttag 1977 vorgestellt und einem breiten Publikum durch die „Communications of the ACM“ nahe gebracht.

Bei sogenannten primitiven Datentypen, den Basisdatentypen einer Programmiersprache, gibt es oft erhebliche Unterschiede zwischen der Datentypimplementierung in den verschiedenen Sprachen, obwohl sie ähnlich heißen. Mal wird ein ganzzahliger Datentyp (*Integer*) mit 8-Bit implementiert, mal mit 16, was zur Folge hat, dass der Datentyp unterschiedliche Wertebereiche umfasst.

/Quelle: www.wikipedia.de/

Abstrakter Datentyp (ADT)

Abstrakter Datentyp (ADT)

Ein abstrakter Datentyp beschreibt das Wesentliche, Konzeptionelle eines Datentyps, unabhängig von einer bestimmten Programmiersprache oder Implementation auf einem Rechner.

Aus den abstrakten Datentypen werden die konkreten Datentypen abgeleitet. Abstrakte Datentypen können für einfache oder strukturierte Datentypen definiert werden.

Abstrakte Datentypen zählen zu den theoretischen Wurzeln der objektorientierten Betrachtungsweisen.

Abstrakter Datentyp (ADT)

„Ein **abstrakter Datentyp**, abgekürzt ADT, ist bestimmt durch:

1. den Namen des ADT;
2. die Festlegung aller Operationen mit ihren Parametern und gegenseitigen Beziehungen, die auf den Instanzen (oft auch Objekte genannt) des ADT zulässig sind. Auf der Beschreibungsebene werden anstelle der algorithmisch orientierten Blockschnittstelle häufig Funktionen verwendet. Diese Funktionen erlauben einerseits eine einfache Beschreibung der Abhängigkeiten und andererseits eine präzise Beschreibung der eigentlichen Blockschnittstellen und
3. die Beschreibung der Eigenschaften der Operationen/Funktionen.“

/Horn, Christian; Immo O. Kerner; Forbrig, Peter; Lehr- und Übungsbuch Informatik; Grundlagen und Überblick; Fachbuchverlag Leipzig; 2003; ISBN 3-446-22543-9/

Abstrakter Datentyp - Spezifikationen

Operationelle Spezifikation

Die operationelle Spezifikation beschreibt die Wirkung der Operationen in einer algorithmischen Notation und entspricht damit einer Implementation auf hohem Abstraktionsniveau.

/Horn, Christian; Immo O. Kerner; Forbrig, Peter; Lehr- und Übungsbuch Informatik; Grundlagen und Überblick; Fachbuchverlag Leipzig; 2003; ISBN 3-446-22543-9/

Abstrakter Datentyp - Spezifikationen

Algebraische Spezifikation

ADT *Name*

Import – Liste von Datentypen und Parametern, die außerhalb deklariert sind, jedoch innerhalb einer Instanz verwendet werden.

Export – Liste der Operationen bzw. Funktionen mit ihren Definitions- und Wertebereichen. Hierüber kann von außen auf die innere Struktur zugegriffen werden.

Axiome – Liste von Beziehungen zwischen den Operationen bzw. Funktionen (in axiomatischer Form), die den Charakter der Datenstruktur (z.B. lineare Liste) und das Verhalten der Operationen/Funktionen festlegen.

/Horn, Christian; Immo O. Kerner; Forbrig, Peter; Lehr- und Übungsbuch Informatik; Grundlagen und Überblick; Fachbuchverlag Leipzig; 2003; ISBN 3-446-22543-9/

Eigenschaften abstrakter Datentypen

Universalität:
ADT kann in unterschiedlichen Programmiersprachen implementiert werden.

Spezifikation:
Vollständige und eindeutige Beschreibung der Schnittstelle zwischen Implementierung und Anwendung.

Geheimnisprinzip (Verbergen von Informationen):
Der Anwender kennt die Wirkungsweise eines ADT, nicht aber die interne Form der Verarbeitung.

Geschütztheit:
Der Zugriff auf einen ADT erfolgt nur über die Schnittstelle der Operationen.
➔ Kapselung des ADT

Grundlagen zur Implementierung von Listen

Die Implementierung des ADT Liste kann durch

- dynamische Datenstrukturen oder
- Felder erfolgen.

Werden dynamische Datenstrukturen verwendet, so muss das **Zeigerkonzept** als Grundlage eingeführt werden.

Vorteil: besseres Verständnis für die rechnerinterne Datenverwaltung

Nachteil: nicht alle Programmiersprachen unterstützen dynamische Datenstrukturen

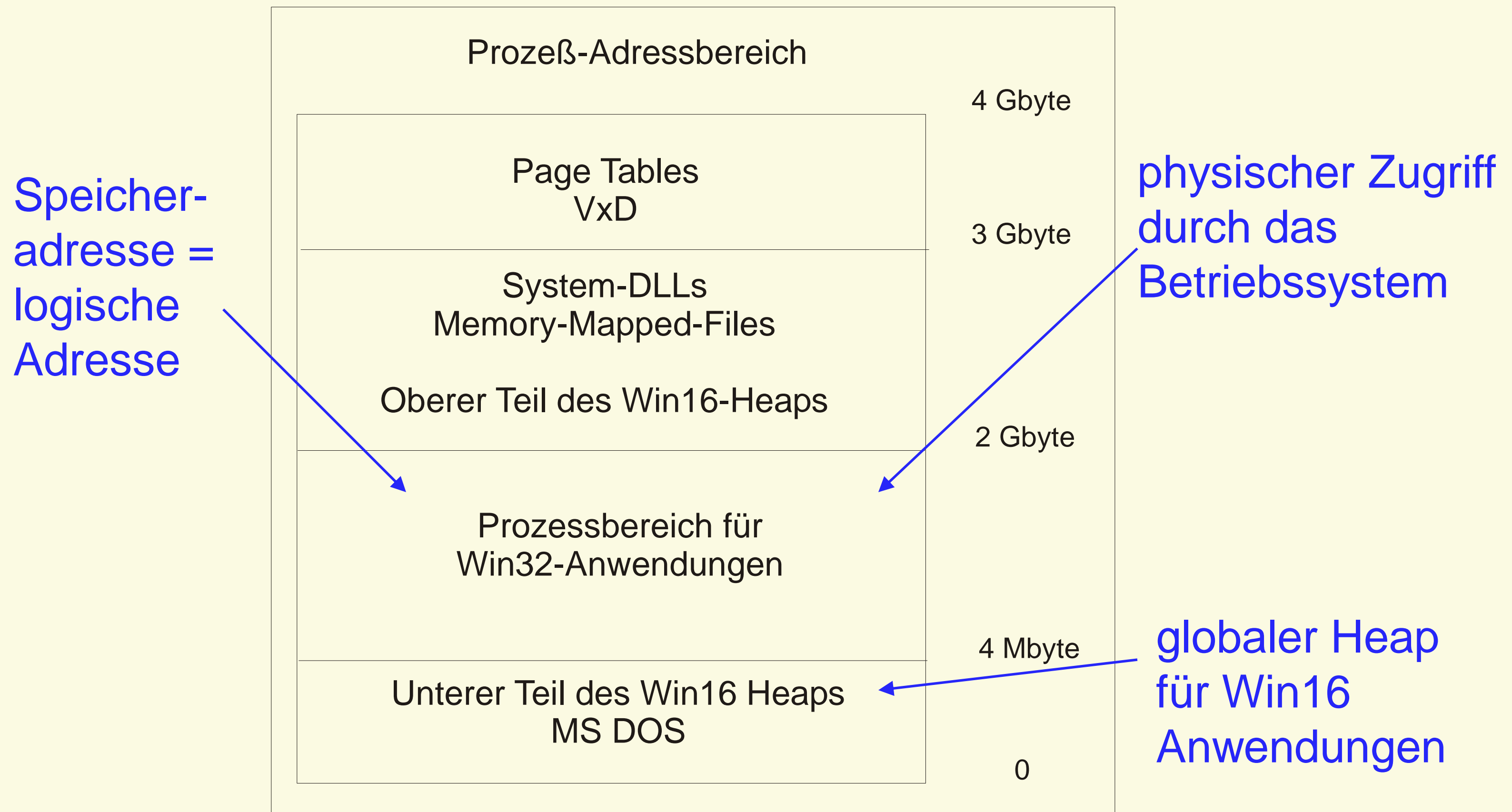
Dynamische Datenstrukturen

Der Datentyp zur Verwaltung von Speicheradressen wird als **Zeiger** oder **Pointer** bezeichnet. Damit wird auch die Nutzung von **dynamischen Variablen** möglich. Der Speicherplatz wird erst nach dem Programmstart verbraucht, vor dem Programmende wieder freigegeben. Datenmengen unbekannter Größe können effektiv verarbeitet werden.

Die so erzeugten Variablen werden nicht durch einen Namen, sondern **nur durch ihre Lage im Speicher beschrieben.**

Zeiger sind notwendig, um komplexe Datenstrukturen, wie Listen und Bäume, zu erzeugen und zu verwalten. Weiterhin werden sie benötigt, um die Methode der objektorientierten Programmierung nutzen zu können.

Speicheraufteilung im 32-Bit-Betriebssystem



Datentyp Zeiger (Pointer)

Ein **Zeiger** ist ein **Verweis auf eine Adresse** im Hauptspeicher.
Auf dieser können z.B. Daten oder Programmcode gespeichert sein. Zeiger sind Elemente, die eine Speicheradresse angeben. Sie können als Zeiger auf einen Datentyp (**typisierte Zeiger**) oder allgemein als Zeiger (**generische Zeiger** bzw. **untypisierte Zeiger**) deklariert werden. Für generische Zeiger steht der Typ **Pointer** zur Verfügung.
Variablen vom Typ Pointer enthalten eine Adresse, jedoch keine Informationen darüber, worauf sie zeigen.

Routinen zur Arbeit mit typisierten Zeigern

new (laufzeiger)	Bereitstellen des Speicherplatzes für einen Eintrag im Heap, entsprechend des Datentyps von Laufzeiger
lz := new (zeiger)	Verwendung von new als Funktion, um Speicherplatz entsprechend des Typs von zeiger bereitzustellen
dispose (zeiger)	Freigabe des Speicherplatzes an der Adresse von zeiger mit der Größe entsprechend des Datentyps von zeiger

Routinen zur Arbeit mit untypisierten Zeigern

getmem (zeiger,wert)	Bereitstellung des in wert angegebenen Speicherplatzes in Byte auf dem Heap, ab einer Adresse, die in zeiger übergeben wird.
freemem (zeiger,wert)	Freigabe des in wert angegebenen Speicherbereiches ab der Adresse, die in zeiger angegeben ist

Zeigerarithmetik

Für Rechenoperationen mit Zeigern besteht die Möglichkeit, diese zu **inkrementieren** bzw. zu **dekrementieren**.
Diese Rechenoperationen sind **nur auf typisierte Zeiger anwendbar**. Sie werden z.B. zur Arbeit mit Zeigern auf Felder genutzt. Für das Hochzählen (Inkrementieren) steht die Prozedur **inc(p)** und für das Herunterzählen (Dekrementieren) die Prozedur **dec(p)** zur Verfügung.
Die Veränderung der Adresse erfolgt jeweils um die Größe des Datentyps, auf die der Zeiger verweist.

Unterrichtsinhalte – Abstrakte Datentypen und ihre Implementierung

Inhalte	Hinweise zum Unterricht
Prinzipien der Verkettung des ADT-Liste ❍ einfach verkettete Liste ❍ doppelt verkettete Liste ❍ Ringliste	● Implementierung durch dynamische Datenstrukturen oder Reihung (Felder)

Listen - Datenmodell

Liste:

Eine **Liste** ist eine endliche Folge von null oder mehr Elementen eines gegebenen Typs. Sind die Elemente vom Typ ETYPE, so wird der Typ der Liste als „Liste von ETYPE“ bezeichnet.

Länge einer Liste:

Die **Länge einer Liste** gibt die Anzahl der Positionen, nicht die Anzahl der verschiedenen Elemente an.

Teilliste:

Wenn die Liste $L = (a_1,a_2,...,a_n)$ gegeben ist, so bezeichnet man für jedes i und j mit $1 <= i <= j <= n$, $(a_i,a_{i+1},...,a_j)$ als **Teilliste** von L. (Zusammenhängende Struktur der ursprünglichen Liste)

Listen - Datenmodell

Teilsequenz:

Eine **Teilsequenz** (subsequenz) der Liste $L = (a_1,a_2,...,a_n)$ ist eine Liste, die durch Entfernen von null oder mehr Elementen aus L entsteht.

Die Reihenfolge der Elemente bleibt erhalten.

Vorkommen:

Die Position, an der ein Element steht, wird auch als **Vorkommen** von a bezeichnet.

Operationen auf Listen

- Einfügen
- Löschen
- Suchen
- Konkatenation
- spezielle Listenoperationen

  first - gibt das erste Element einer Liste an (Kopf)

  last - gibt das letzte Element einer Liste an (Ende)

  retrieve (i,L) - gibt das i-te Element einer Liste an

  length(L) - gibt die Länge einer Liste an (Anzahl der belegten Plätze)

  isEmpty(L) - gibt true an, wenn L eine leere Liste ist

  isNotEmpty(L) - gibt true an, wenn L keine leere Liste ist

Spezifikation des ADT Liste

Die Beschreibung des ADT Liste erfolgt exakt über Import, Export, Wertebereich und Operationen.

Import:
Typen: TInhalt (als Inhaltstyp der Elemente)

Export:
Typen: TListe

Wertebereich:
Die Elemente der Datenstruktur sind linear angeordnet, d.h. jedes Element außer dem ersten hat einen direkten Vorgänger und jedes Element außer dem letzten hat einen direkten Nachfolger.
Falls die Liste nicht leer ist, existiert ein aktuelles Element. Die Anzahl der Elemente ist durch den zur Laufzeit verfügbaren Speicherplatz begrenzt.
Die Elemente enthalten den importierten Inhaltstyp.

Beispiel: Deklaration in Object-Pascal

```
type T_Zeiger = ^T_Liste;

     T_inhalt = record
       name   : string [20];
       nummer : string[10];
       end;

     T_Liste = record
       inhalt : T_inhalt;
       next   : T_Zeiger;
       end;
```

Listen

Listen sind durch folgende Eigenschaften charakterisiert:

• die **Art und Weise der Verkettung** (in der Regel durch Zeiger beschrieben),

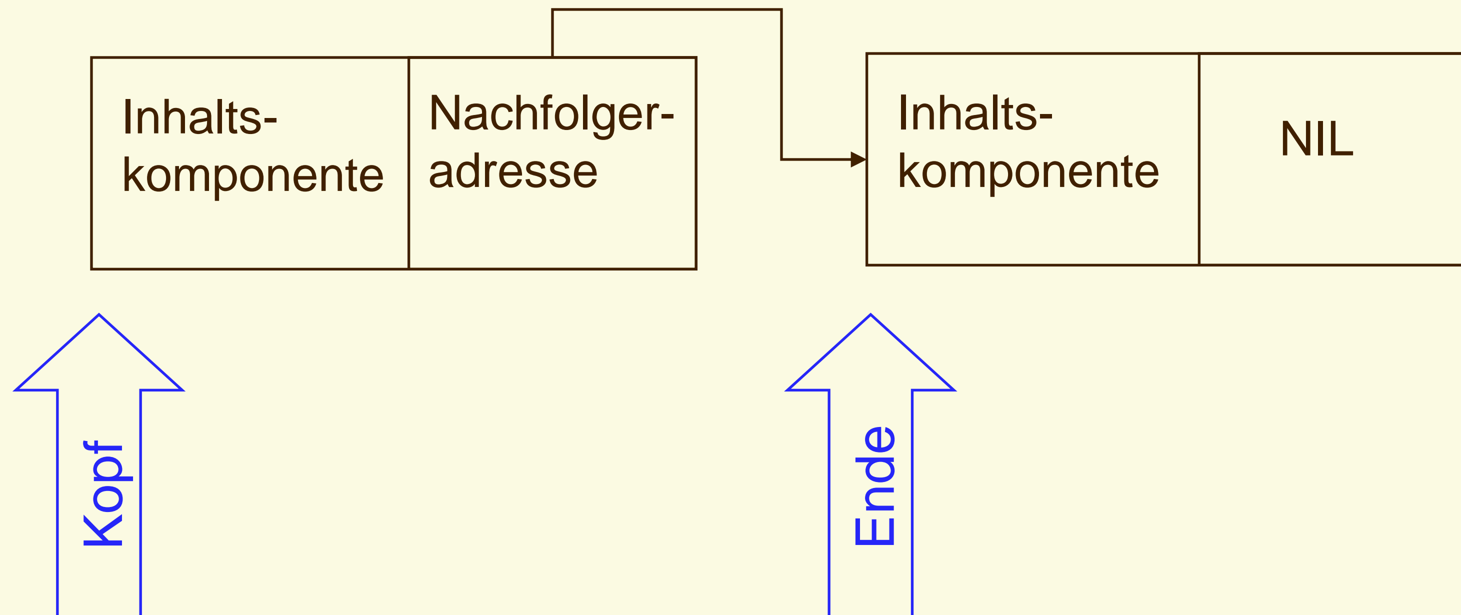
• **wie und wo Elemente eingefügt werden können** (üblicherweise an beliebigen Stellen der Liste, da die Daten im Hauptspeicher nicht geordnet liegen),

• die **Anzahl der Vorgänger und der Nachfolger**, die ein Listenelement haben kann.

Um eine allgemeine Listenstruktur angeben zu können, muss das System in der Lage sein, auch nicht zusammenhängenden Speicherplatz zu nutzen. Dazu ist es mindestens erforderlich, dass jedes Listenelement die Speicheradresse seines Nachfolgers kennt (Vorwärtsverkettung). Instrument zur Realisierung dieser Struktur ist der Zeiger.

Listen

Einfach verkettete Listen – jedes Element kennt seinen Nachfolger.



Unterrichtsinhalte – Abstrakte Datentypen und ihre Implementierung

Inhalte	Hinweise zum Unterricht
Implementierung der Grundfunktionen - Methoden auf Listen ❍ Initialisierung ❍ Einketten ❍ Ausketten ❍ Suchen ❍ Speichern in Dateien ❍ Laden aus Dateien	• Realisierung an einem komplexen Beispiel

Operationen auf den ADT Liste

Erzeugen:

PROC Erzeuge (TListe [aus])
vor Es existiert keine Liste.
nach Es existiert eine leere Liste.

```
procedure p_init(var kopf,ende,aktuell:t_zeiger);
    begin
    kopf     := nil;
    ende     := nil;
    aktuell := nil;
    end;  // of procedure p_init
```

Operationen auf den ADT Liste – Einfügen vor dem aktuellen Element

Einfügen (vor):

PROC FuegeEinVor (TListe [ein/aus]; TInhalt[ein])

vor	Die Liste ist erzeugt und nicht voll.
nach	Der **Inhalt ist als neues Element vor dem aktuellen Element in die Liste eingefügt**. War die Liste zuvor leer, ist der Inhalt das erste (und letzte) Element der Liste. Das neue Element ist das aktuelle.

Implementierung von Einfügen vor

Erzeugen eines Listenelementes und Einketten vor dem aktuellen Element

1. Anfordern des benötigten Speicherplatzes mit **new**
2. Test, ob Element erstes Element in der Liste ist **(kopf = nil**), wenn ja, dann für das erste Element den Anfangspunkt der Liste (z.B. Kopf bzw. Wurzel) auf Adresse des Elementes setzen
3. **Suchen des Vorgängers des aktuellen Elementes**
4. Herstellen der Verkettung, indem dem Vorgänger des aktuellen Elementes die Adresse des neuen Elementes als Nachfolger eingetragen wird. Dem neu erzeugten Element wird die Adresse des aktuellen Elementes als Nachfolger eingetragen.
5. Übertragen der Inhaltskomponente in das Listenelement

Operation: Einfügen vor dem aktuellen Element

```
procedure p_fuegeeinvor(var kopf,ende,aktuell: T_Zeiger; satz : T_inhalt);
   var akt,vor : t_zeiger;
   begin
    new(akt);
    if kopf = nil then
      begin              // Sonderfall: leere Liste
      kopf := akt; ende := akt; aktuell := akt;
      akt^.next := NIL;
      end
     else
       begin
        if kopf = aktuell then
           kopf := akt // Sonderfall: Einfuegen vor dem 1. Element
          else
           begin        // Element in der Liste
             vor := kopf;
             while vor^.next <> aktuell do  // Vorg. von akt. Element suchen
                vor := vor^.next;
             vor^.next := akt;
           end;
          akt^.next := aktuell;   //  Vorwaertsverkettung
       end;
      akt^.inhalt := satz;       // Inhalt übertragen
      aktuell := akt;    // das aktuelle Element ist das neue
   end; // of procedure p_fuegeeinvor
```

Operationen auf den ADT Liste – Einfügen nach dem aktuellen Element

Einfügen (nach):

PROC FuegeEinNach (TListe [ein/aus]; TInhalt[ein])

vor	Die Liste ist erzeugt und nicht voll.
nach	Der Inhalt ist als neues Element nach dem aktuellen Element in die Liste eingefügt. War die Liste zuvor leer, ist der Inhalt das erste (und letzte) Element der Liste. Das neue Element ist das aktuelle.

Implementierung von Einfügen nach

Erzeugen eines Listenelementes und Einketten nach dem aktuellen Element

1. Anfordern des benötigten Speicherplatzes mit **new**
2. Test, ob Element erstes Element in der Liste ist **(kopf = nil**), wenn ja, dann für das erste Element den Anfangspunkt der Liste (z.B. Kopf bzw. Wurzel) auf Adresse des Elementes setzen
3. Eintragen der Inhaltskomponente in das Listenelement
4. Herstellen der Verkettung, indem dem neuen Element die Nachfolgeradresse des aktuellen Elementes in die Nachfolgerkomponente, dem aktuellen Element die Adresse des neuen Elementes in die Nachfolgerkomponente eingetragen wird
5. Ist das neue Element letztes Element, so wird der Ende-Zeiger auf den Wert des neuen Elementes gesetzt.

Operation: Einfügen nach dem aktuellen Element

```
procedure p_fuegeeinnach(var kopf,ende,aktuell: T_Zeiger; satz :
T_inhalt);
   var akt : t_zeiger;
   begin
    new(akt);
    if kopf = nil then
      begin                 // Sonderfall: leere Liste
      kopf := akt; ende := akt; aktuell := akt;
      akt^.next := NIL;
      end
     else
       begin
          akt^.next := aktuell^.next;      //Vorwaertsverkett.
          aktuell^.next := akt;
       end;
      akt^.inhalt := satz;         // Inhalt übertragen
      if aktuell = ende then       // ggf. Ende aktualisieren
        ende := akt;
      aktuell := akt;     // das aktuelle Element ist das neue
   end; // of procedure p_fuegeeinnach
```

Operationen auf den ADT Liste – Ändern des Inhaltes eines Elementes

Ändern:

PROC Aendere (TListe [ein/aus]; TInhalt[ein])

vor Die Liste ist erzeugt und nicht leer.

nach Der Inhalt des aktuellen Elementes ist mit dem übergebenen (neuen) Inhalt überschrieben, die Struktur der Liste ist unverändert. Ist die Liste leer, so wird der Inhalt als neues Listenelement eingefügt.

Operation: Einfügen nach dem aktuellen Element

```
procedure p_aendern(var kopf,ende,aktuell:T_Zeiger;
satz : T_inhalt);
    begin
      if aktuell <> nil then
       aktuell^.inhalt := satz  // Inhalt übertragen
       else
       p_fuegeeinnach(kopf,ende,aktuell,satz); // Wenn
Liste Leer, Element einfügen
     end; // of procedure p_aendern
```

Operationen auf den ADT Liste – Löschen des aktuellen Elementes

Löschen:

PROC Loesche (TListe [ein/aus])

vor Die Liste ist erzeugt und nicht leer.

nach Das aktuelle Element ist nicht mehr in der Liste enthalten. Falls die Liste nicht leer geworden ist, ist das erste Element das aktuelle.

Implementierung: Löschen

Löschen des aktuellen Elementes aus einer Liste

1. Suchen des Vorgängers des zu löschenden Elementes durch Durchlaufen der Liste.
2. Die im aktuellen Element eingetragene Nachfolgeradresse wird dem Vorgänger als Nachfolgeradresse zugewiesen (z.B. vorg^.next := aktuell^.next). Hat das aktuelle Element keinen Vorgänger (erstes Element), so wird die Nachfolgeradresse als Kopfadresse eingetragen.
3. Der vom Listenelement belegte Speicherplatz wird mit dispose freigegeben.
4. Ist das zu löschende Element das letzte Element, so wird der Ende-Zeiger auf das Vorgängerelement positioniert.

Operationen auf den ADT Liste – Löschen des aktuellen Elementes

```
procedure p_loesch(var kopf,ende,aktuell:T_Zeiger);
     var vorg : T_zeiger;
     begin
       if aktuell <> nil then   // Gibt es das zu loeschende Element?
         begin
         if aktuell = kopf then
            if kopf = ende then    // Das einzige Element der Liste 
wird gelöscht
               begin
               dispose(aktuell);
               p_init(kopf,ende,aktuell); // Zeiger Initialisieren
               end
               else
                 begin    // Kopfelement ist zu loeschen
                 kopf := aktuell^.next;
                 dispose(aktuell);
                 aktuell := kopf;
                 end
```

Operationen auf den ADT Liste – Löschen des aktuellen Elementes

```
          else          // Element aus der Liste ist zu loeschen
              begin
                 vorg := kopf;
                 while vorg^.next <> aktuell do
                   vorg := vorg^.next;
                 vorg^.next := aktuell^.next;   // Verketten
                 dispose(aktuell);
                 if vorg^.next <> NIL then
                   aktuell := vorg^.next
                   else
                     begin
                     aktuell := vorg;
                     ende := vorg;
                     end;
              end;
         end;
     end; // of procedure p_loesch
```

Operationen auf den ADT Liste – Ausgabe des Inhaltes des akt. Elementes

Inhalt ausgeben:

PROC Lies (TListe [ein/aus]; TInhalt[aus])

vor Die Liste ist erzeugt und nicht leer.

nach Im Ausgangsparameter steht der Inhalt des aktuellen Elements.

```
procedure p_lies(var aktuell:T_Zeiger;var satz:
T_inhalt);
     begin
     if aktuell <> nil then
        satz := aktuell^.inhalt;
     end; // of procedure p_lies
```

Operationen auf den ADT Liste – Suchen eines Elementes

Element suchen:

PROC Lies (TListe [ein/aus]; TInhalt[ein])

vor	Die Liste ist erzeugt und nicht leer.
nach	Das aktuelle Element ist das gesuchte (erste Übereinstimmung). Wird kein Element mit Übereinstimmung gefunden, so ist das aktuelle Element nil.

Operationen auf den ADT Liste – Suchen eines Elementes

Suchen in einer Liste

1. Setzen des aktuellen Zeigers auf den Kopf der Liste
2. Vergleich der Inhaltskomponente des aktuellen Elementes (oder Teilen davon) mit dem Suchdatensatz. Bei Übereinstimmung ggf. Ende der Suche und Positionieren des aktuellen Zeigers auf das gefundene Element.
3. Setzen des aktuellen Zeigers auf das Nachfolgerelement und Fortsetzung bei 2. bis Eintragung für das Nachfolgerelement = nil.

Operationen auf den ADT Liste – Suchen eines Elementes

```
procedure p_such(var kopf,aktuell:T_zeiger;
      satz:T_inhalt);
    var switch : boolean;
    begin
      switch := false;
      aktuell := kopf;
      while (aktuell <> NIL) and not(switch) do
         begin
         if satz.name <> aktuell^.inhalt.name then
           aktuell := aktuell^.next
           else
           switch := true;
         end; // of while
    end; // of procedure p_such
```

Operationen auf den ADT Liste – Positionieren auf den Listenanfang

Anfang finden:

PROC FindeErstes(TListe [ein/aus])

vor Die Liste ist erzeugt und nicht leer.

nach Das erste Element ist das aktuelle.

```
procedure p_anfang (kopf,ende:T_Zeiger; var
aktuell:T_Zeiger);
    begin
     aktuell := kopf;
    end; // of procedure p_anfang
```

Operationen auf den ADT Liste – Positionieren auf das Nachfolgerelement

Nachfolger finden:

PROC FindeNaechstes(TListe [ein/aus])

vor Die Liste ist erzeugt und nicht leer und das aktuelle Element ist nicht das letzte in der Liste.

nach Das aktuelle Element ist der Nachfolger des zuvor aktuellen.

```
procedure p_next (kopf,ende:T_Zeiger; var
aktuell:T_Zeiger);
    begin
     aktuell := aktuell^.next;
    end; // of procedure p_next
```

Operationen auf den ADT Liste – Test: Ist Liste leer?

Liste leer?:

FUNC Leer(TListe [ein]):Boolean

vor Die Liste existiert.

nach Falls die Liste kein Element enthält ist das Funktionsergebnis *True*, ansonsten *False*.

```
function f_listeleer(kopf:T_Zeiger): boolean);
    begin
      if kopf = NIL then
        f_listeleer := true
        else
        f_listeleer := false;
    end; // of function f_listeleer
```

Operationen auf den ADT Liste – Test: Ist Liste voll?

Liste voll?:

FUNC Voll:Boolean

vor Die Liste existiert.

nach Falls im Speicher kein Platz mehr für ein neues Listenelement ist, ist das Funktionsergebnis *True*, ansonsten *False*.

Operationen auf den ADT Liste – Anzahl der Elemente

Anzahl der Elemente?:

FUNC ElementZahl(TListe [ein]):Integer

vor Die Liste existiert.

nach Das Funktionsergebnis enthält die Anzahl der Listenelemente.

Operationen auf den ADT Liste – Anzahl der Elemente

```
function f_elementzahl(kopf:T_Zeiger): integer;
    var akt     : T_zeiger;
        counter : integer;
    begin
      akt := kopf;
      counter := 0;
      while akt <> nil do
        begin
        inc(counter);
        akt := akt^.next;
        end;
      f_elementzahl := counter;
    end; // of function f_elementzahl
```

Operationen auf den ADT Liste – Ist das aktuelle Element das letzte Element?

Letztes Element?:

FUNC Letztes(TListe [ein]):Boolean

vor Die Liste existiert und ist nicht leer.

nach Falls das aktuelle Element das letzte der Liste ist, ist das Funktionsergebnis *True*, ansonsten *False*.

```
function f_letzteselement(aktuell,ende:T_Zeiger):
boolean;
    begin
      f_letzteselement := false;
      if aktuell = ende then
        f_letzteselement := true
     end; // of function f_letztes Element
```

Operationen auf den ADT Liste – Liste vollständig auf Datenträger speichern

Liste Speichern:

PROC Speichere (TListe [ein]; String[ein]):Integer

vor	Die Liste existiert und auf dem Datenträger ist hinreichend Platz.

nach	Die Liste ist unter dem als zweiten Parameter angegebenen Namen auf dem Datenträger gespeichert.

Operationen auf den ADT Liste – Liste vollständig auf Datenträger speichern

Liste Laden:

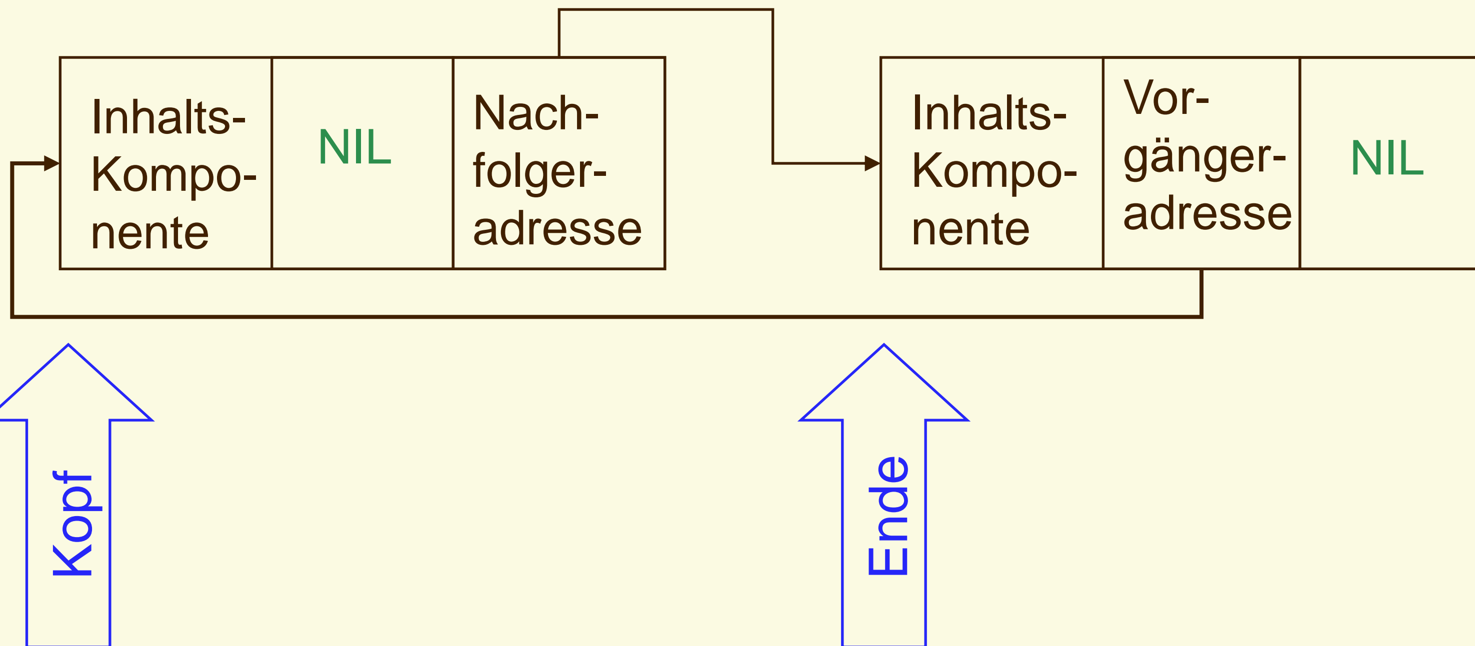
PROC Lade (TListe [ein/aus]; String[ein]):Integer

vor Die Liste existiert und ist nicht leer. Auf dem Datenträger existiert ein unter dem als zweiten Parameter übergebenen Namen mit Hilfe der Operation *Speichere* abgespeicherte Datei.

nach Die Liste ist vom Datenträger geladen und unter dem ersten Parameter verfügbar. Falls die Liste während des Ladevorganges voll geworden ist, so sind die nachfolgenden Elemente nicht enthalten.

Datenstruktur: doppelt verkettete Liste

Doppelt verkettete Listen – jedes Element kennt seinen Vorgänger und Nachfolger.



Datenstruktur doppelt verkettete Liste in Objekt-Pascal

```
type T_Zeiger = ^T_Liste;

     T_inhalt = record
       name   : string [20];
       nummer : string[10];
       end;

     T_Liste = record
       inhalt : T_inhalt;
       prev   : T_zeiger;
       next   : T_Zeiger;
       end;
```

Ausgewählte Operationen auf doppelt verkettete Listen – Einfügen vor

Einfügen (vor):

PROC FuegeEinVor (TListe [ein/aus]; TInhalt[ein])

vor Die Liste ist erzeugt und nicht voll.

nach Der **Inhalt ist als neues Element vor dem aktuellen Element in die Liste eingefügt**. War die Liste zuvor leer, ist der Inhalt das erste (und letzte) Element der Liste. Das neue Element ist das aktuelle.

Ausgewählte Operationen auf doppelt verkettete Listen – Einfügen vor

Erzeugen eines Elementes einer doppelt verketteten Liste und Einfügen vor dem aktuellen Element

1. Anfordern des benötigten Speicherplatzes mit **new**
2. Test, ob Element erstes Element in der Liste ist **(kopf = nil**), wenn ja, dann für das erste Element den Anfangspunkt der Liste (z.B. Kopf bzw. Wurzel) auf Adresse des Elementes setzen
3. Eintragen der Inhaltskomponente in das Listenelement

Ausgewählte Operationen auf doppelt verkettete Listen – Einfügen vor

Erzeugen eines Elementes einer doppelt verketteten Liste und Einfügen vor dem aktuellen Element

1. Anfordern des benötigten Speicherplatzes mit **new**
2. Test, ob Element erstes Element in der Liste ist **(kopf = nil**), wenn ja, dann für das erste Element den Anfangspunkt der Liste (z.B. Kopf bzw. Wurzel) auf Adresse des Elementes setzen
3. Eintragen der Inhaltskomponente in das Listenelement

Ausgewählte Operationen auf doppelt verkettete Listen – Einfügen vor

Erzeugen eines Elementes einer doppelt verketteten Liste und Einfügen vor dem aktuellen Element

4. Herstellen der Vorwärtsverkettung, indem dem Vorgängerelement des aktuellen Elementes die aktuelle Adresse in die Nachfolgerkomponente eingetragen wird und dem neuen Element die Adresse des aktuellen Elementes in die Nachfolgerkomponente eingetragen wird.

Beispiel:

```
aktuell^.prev^.next := akt;
akt^.next := aktuell;
```

Ausgewählte Operationen auf doppelt verkettete Listen – Einfügen vor

Erzeugen eines Elementes einer doppelt verketteten Liste und Einfügen vor dem aktuellen Element

5. Herstellen der Vorwärtsverkettung, indem dem einzukettenden Element die Adresse des Vorgängers des aktuellen Elementes in die Vorgängerkomponente eingetragen wird und dem aktuellen Element die Adresse des einzukettenden Elementes in die Vorgängerkomponente eingetragen wird (dem ersten Element wird der Wert nil eingetragen)

Beispiel:

```
akt^.prev := aktuell^.prev;
aktuell^.prev := akt;
```

Ausgewählte Operationen auf doppelt verkettete Listen – Einfügen vor

Erzeugen eines Elementes einer doppelt verketteten Liste und Einfügen vor dem aktuellen Element

6. Wird das neue Element vor dem Kopfelement eingefügt, so muss der Kopfzeiger auf das neue Element gesetzt werden.
7. Das eingekettete Element wird zum aktuellen Element.

Ausgewählte Operationen auf doppelt verkettete Listen – Einfügen vor

```
procedure p_fuegeeinvor_d(var kopf,ende,aktuell: T_Zeiger;
satz : T_inhalt);
   var akt : t_zeiger;
   begin
    new(akt);
    if kopf = nil then
      begin                   // Sonderfall: leere Liste
      kopf := akt; ende := akt;
      akt^.next := NIL; akt^.prev := NIL;
      end
     else
      begin
        if kopf = aktuell then
          begin // Sonderfall: Element wird Kopfelement
          kopf := akt;
          akt^.prev := NIL;
          akt^.next := aktuell;
          aktuell^.prev := akt;
          end
```

Ausgewählte Operationen auf doppelt verkettete Listen – Einfügen vor

```
         else  // allgemeiner Fall
           begin
           akt^.next := aktuell;   //Vorwärtsv.
           aktuell^.prev^.next := akt;
           aktuell^.prev := akt;   // Rueckwaertsv.
           akt^.prev := aktuell^.prev;
           end;
       end;
    akt^.inhalt := satz;        // Inhalt übertragen
    aktuell := akt;    // das aktuelle Element ist das neue

  end; // of procedure p_fuegeeinvor_d
```

Ausgewählte Operationen auf doppelt verkettete Listen – Einfügen nach

Erzeugen eines Elementes einer doppelt verketteten Liste und Einfügen nach dem aktuellen Element

1. Anfordern des benötigten Speicherplatzes mit **new**
2. Test, ob Element erstes Element in der Liste ist **(kopf = nil)**, wenn ja, dann für das erste Element den Anfangspunkt der Liste (z.B. Kopf bzw. Wurzel) auf Adresse des Elementes setzen
3. Eintragen der Inhaltskomponente in das Listenelement

Ausgewählte Operationen auf doppelt verkettete Listen – Einfügen nach

Erzeugen eines Elementes einer doppelt verketteten Liste und Einfügen nach dem aktuellen Element

4. Herstellen der Vorwärtsverkettung, indem dem neuen Element die Nachfolgeradresse des aktuellen Elementes in die Nachfolgerkomponente eingetragen wird und dem aktuellen Element die Adresse des einzukettenden Elementes in die Nachfolgerkomponente eingetragen wird.

Beispiel:

```
akt^.next := aktuell^.next;
aktuell^.next := akt;
```

Ausgewählte Operationen auf doppelt verkettete Listen – Einfügen nach

Erzeugen eines Elementes einer doppelt verketteten Liste und Einfügen nach dem aktuellen Element

5. Herstellen der Rückwärtsverkettung, indem dem einzukettenden Element die Adresse des aktuellen Elementes in die Vorgängerkomponente eingetragen wird und dem Nachfolgerelement des aktuellen Elementes die Adresse des einzukettenden Elementes in die Vorgängerkomponente eingetragen wird (dem ersten Element wird der Wert nil eingetragen)

Beispiel:

```
akt^.prev := aktuell;
aktuell^.next.^prev := akt;
```

Ausgewählte Operationen auf doppelt verkettete Listen – Einfügen nach

Erzeugen eines Elementes einer doppelt verketteten Liste und Einfügen nach dem aktuellen Element

6. Wird das einzufügende Element nach dem letzten Element eingefügt, so muss der Endezeiger auf das neue Element gesetzt werden.
7. Das eingekettete Element wird zum aktuellen Element.

Ausgewählte Operationen auf doppelt verkettete Listen – Einfügen nach

```
procedure p_fuegeeinnach_d(var kopf,ende,aktuell: T_Zeiger; satz :
T_inhalt);
   var akt : t_zeiger;
   begin
    new(akt);
    if kopf = nil then
      begin              // Sonderfall: leere Liste
      kopf := akt;  ende := akt; aktuell := akt;  akt^.next := NIL;
      end
     else
       begin
          akt^.prev := aktuell;           // Ruechwaertsverkettung
          if aktuell^.next <> nil then
            aktuell^.next^.prev := akt;
          akt^.next := aktuell^.next;     //  Vorwaertsverkettung
          aktuell^.next := akt;
       end;
      akt^.inhalt := satz;        // Inhalt übertragen
      if aktuell = ende then      // ggf. Ende aktualisieren
        ende := akt;
      aktuell := akt;    // das neue Element ist das aktuelle
   end; // of procedure p_fuegeeinnach_d
```

Doppelt verkettete Listen – Löschen des aktuellen Elementes

Löschen des aktuellen Elementes aus einer doppelt verketteten Liste

1. Ist das Element einziges Element der Liste, so werden die Kopf- und Endezeiger auf NIL gesetzt.
2. Dem Vorgänger des aktuellen Elementes, falls vorhanden, wird in die Nachfolgerkomponente der Wert eingetragen der beim aktuellen Element in der Nachfolgerkomponente steht.

```
aktuell^.prev^.next := aktuell^.next;
```

Doppelt verkettete Listen – Löschen des aktuellen Elementes

3. Wenn das zu löschende Element einen Nachfolger hat, so wird diesem die Adresse des Vorgängers des aktuellen Elementes in die Vorgängerkomponente eingetragen.

```
if aktuell^.next <> NIL then
      begin
      hilf := aktuell^.next;
      aktuell^.next^.prev := aktuell^.prev;
      end
       else
        begin
        hilf := aktuell^.prev; ende := hilf;
        end;
```

4. Der vom Listenelement belegte Speicherplatz wird durch dispose(aktuell) freigegeben.
5. Der aktuelle Zeiger wird auf das Vorgängerelement gesetzt.

Doppelt verkettete Listen – Löschen des aktuellen Elementes

```
procedure p_loesch_d(var kopf,ende,aktuell:T_Zeiger);
    var hilf : T_zeiger;
     begin
       if aktuell <> nil then   // Gibt es das zu loeschende Element?
         begin
         if aktuell = kopf then
            if kopf = ende then    // Das einzige Element der Liste
wird gelöscht
               begin
               dispose(aktuell);
               p_init_d(kopf,ende,aktuell); // Zieger Initialisieren
               end
               else
                 begin    // Kopfelement ist zu loeschen
                 kopf := aktuell^.next;
                 aktuell^.next^.prev := NIL;
                 dispose(aktuell);
                 aktuell := kopf;
                 end
```

Doppelt verkettete Listen – Löschen des aktuellen Elementes

```
else          // Element aus der Liste ist zu loeschen
                begin
                   aktuell^.prev^.next := aktuell^.next;
                   if aktuell^.next <> NIL then
                       begin
                       hilf := aktuell^.next;
                       aktuell^.next^.prev := aktuell^.prev;
                       end
                       else
                         begin
                         hilf := aktuell^.prev;
                         ende := hilf;
                         end;
                   dispose(aktuell);
                   aktuell := hilf;
                end;
           end;
     end; // of procedure p_loesch_d
```

Unterrichtsinhalte – Abstrakte Datentypen und ihre Implementierung

Inhalte	Hinweise zum Unterricht
Anwendung von Listen ❍ Stack (Stapel) ❍ Queue (Warteschlange)	• LIFO-Prinzip • FIFO-Prinzip • Nachbildung eines Bediensystemmodells

Warteschlangen - Queues

Warteschlange

Eine Warteschlange ist eine eingeschränkte Form des abstrakten Datentyps Liste. Elemente dürfen nur hinten, am Ende, eingefügt und vorn, am Anfang, entfernt werden (Umkehrung auch zulässig).

Warteschlangen werden auch als fifo (first in - first out)-Speicher bezeichnet.

Spezifikation des ADT - Queue

Die Beschreibung des ADT Queue erfolgt exakt über Import, Export, Wertebereich und Operationen.

Import:

Typen: TInhalt (als Inhaltstyp der Elemente)

Export:

Typen: TQueue

Wertebereich:

Die Elemente der Datenstruktur sind linear angeordnet. Ein neues Element kann nur am Ende eingefügt werden, entfernt werden kann ein Element nur am Anfang. (FiFo-Struktur)
Die Anzahl der Elemente ist durch den zur Laufzeit verfügbaren Speicherplatz begrenzt.
Die Elemente enthalten den importierten Inhaltstyp.

ADT – Queue – Typendeklaration in Object-Pascal

```
type T_Zeiger = ^T_Liste;

     T_inhalt = record
       name    : string [20];
       zeit    : Tdatetime;
       end;

     T_Liste = record
       inhalt : T_inhalt;
       next   : T_Zeiger;
       end;
```

Operationen auf den ADT – Queue - Initialisieren

Initialisieren:

PROC Init (TQueue [aus])
vor
nach Es existiert eine leere Schlange (als Ausgangsparameter).

```
procedure p_init (var kopf,ende:t_zeiger);
    begin
    kopf     := nil;
    ende     := nil;
    end;  // of procedure p_init
```

Operationen auf den ADT – Queue - Einfügen

Einfügen:

PROC Put (TQueue [ein/aus]; TInhalt[ein])
vor Die Schlange existiert und nicht voll.
nach Der Inhalt ist als letztes Element der Schlange angefügt.

Operationen auf den ADT – Queue - Einfügen

```
procedure p_put(var kopf,ende: T_Zeiger; satz :
      T_inhalt)};
   var akt : t_zeiger;
   begin
    new(akt);
    if kopf = nil then
      begin               // Sonderfall: leere Queue
      kopf := akt;  ende := akt;
      end
     else
       begin
          ende^.next := akt; //  Vorwaertsverkettung
       end;
     akt^.next := NIL;
     akt^.inhalt := satz;       // Inhalt übertragen
     ende := akt;
   end; // of procedure p_put
```

Operationen auf den ADT – Queue - Entnehmen

Entnehmen:

PROC Get (TQueue [ein/aus]; TInhalt[aus])

vor	Die Warteschlange existiert und ist nicht leer.
nach	Das erste Element der Schlange ist entfernt, eine Kopie seines Inhaltes steht im zweiten Parameter.

Operationen auf den ADT – Queue - Entnehmen

```
procedure p_get(var kopf,ende: T_Zeiger; var satz :
      T_inhalt; var error : byte);
  var hilf : t_zeiger;
  begin
   if kopf <> nil then
     begin
     satz := kopf^.inhalt;
     hilf := kopf;
     if ende = kopf  then // Entfernen des letzten E.
       ende := nil;
     kopf := kopf^.next;
     dispose(hilf);
     error := 0;
     end
    else
      error := 1;
  end; // of procedure p_get
```

Operationen auf den ADT – Queue – Entnehmen (Funktion)

```
function f_get(var kopf,ende: T_Zeiger):T_zeiger;
  var hilf : T_zeiger;
  begin
   if kopf <> nil then
     begin
     hilf := kopf;
     if ende = kopf  then // Entfernen des letzten
      Elementes
       ende := nil;
     kopf := kopf^.next;
     f_get:= hilf;
     end
    else
      f_get := NIL;
  end; // of procedure p_get
```

Operationen auf den ADT – Queue – Anzahl der Elemente in der Queue

Anzahl?:

FUNC Count (TQueue [ein]): Integer

vor Die Schlange existiert.

nach Die Anzahl der Elemente der Schlange wird als Funktionsergebnis zurückgegeben.

```
function f_count(kopf:T_Zeiger):integer;
    var hilf  :T_Zeiger;
        count : integer;
    begin
    hilf := kopf; count := 0;
    while hilf <> nil do
      begin
      hilf := hilf^.next; inc(count);
      end;
    f_count := count;
    end; // of function f_count
```

Operationen auf den ADT – Queue – Schlange leer

Ist die Schlange leer?:

FUNC Empty (TQueue [ein]):Boolean

vor Die Schlange existiert.

nach Falls die Schlange kein Element enthält, ist das Funktionsergebnis True, ansonsten False.

```
function f_empty(kopf:T_Zeiger): boolean;
   begin
     if kopf = NIL then
       f_empty := true
       else
       f_empty := false;
   end; // of function f_empty
```

Stapel - Stack

Stack

Ein Stack ist ein abstrakter, auf einem Listenmodell basierender Datentyp. Alle Operationen werden an einem Ende der Liste durchgeführt. Das Element wird als oberstes Element (top) der Liste bezeichnet.

Für den Stack ist auch die Bezeichnung lifo (last in - first out) - Speicher gebräuchlich.

Spezifikation des ADT - Stack

Die Beschreibung des ADT Stack erfolgt exakt über Import, Export, Wertebereich und Operationen.

Import:
Typen: TInhalt

Export:
Typen: TStack

Wertebereich:
Die Elemente der Datenstruktur sind linear angeordnet, nur das jeweils zuletzt hinzugefügte Element (Top-of-Stack) ist erreichbar.
Die Anzahl der Elemente ist durch den zur Laufzeit verfügbaren Speicherplatz begrenzt.
Die Elemente enthalten den importierten Inhaltstyp.

ADT – Stack – Typendeklaration in Object-Pascal

```
type T_Zeiger = ^T_Liste;

     T_inhalt = record
       name   : string [20];
       zeit   : Tdatetime;
       end;

     T_Liste = record
       inhalt : T_inhalt;
       next   : T_Zeiger;
       end;
```

Operationen auf den ADT – Stack – Initialisieren

Initialisieren:

PROC Init (TStack [aus])
vor
nach Es existiert ein leerer Stack (als Ausgangsparameter).

```
procedure p_init (var kopf:t_zeiger);
   begin
   kopf    := nil;
   end;  // of procedure p_init
```

Operationen auf den ADT – Stack – Push

Einfügen:

PROC Push (TStack [ein/aus]; TInhalt[ein])

vor Der Stack existiert und ist nicht voll.

nach Der im zweiten Parameter übergebene Inhalt ist der Top-of-Stack.

```
 procedure p_push(var kopf: T_Zeiger; satz :
T_inhalt);
   var akt : t_zeiger;
   begin
    new(akt);
    akt^.next := kopf;
    kopf := akt;
    akt^.inhalt := satz;
   end; // of procedure p_push
```

Operationen auf den ADT – Stack – Pop

Entnehmen:

PROC Pop (TStack [ein/aus]; TInhalt[aus])

vor	Der Stack existiert und ist nicht leer.
nach	Der Top-of-Stack ist entfernt, eine Kopie seines Inhaltes steht im zweiten Parameter zur Verfügung.

Operationen auf den ADT – Stack – Pop

```
 procedure p_pop(var kopf: T_Zeiger; var satz :
T_inhalt; var error : byte);
   var hilf : t_zeiger;
   begin
    if kopf <> nil then
      begin
      satz := kopf^.inhalt;
      hilf := kopf;
      kopf := kopf^.next;
      dispose(hilf);
      error := 0;
      end
     else
       begin
          error := 1;
       end;
   end; // of procedure p_pop
```

Operationen auf den ADT – Stack – Pop

```
function f_pop{(var kopf: T_Zeiger):T_zeiger};
  var hilf : T_zeiger;
  begin
      hilf := kopf;
      kopf := kopf^.next;
      f_pop:= hilf;
  end; // of procedure p_pop
```

Operationen auf den ADT – Stack – Anzeigen

Anzeigen:

PROC Top (TStack [ein/aus]; TInhalt[aus])

vor Der Stack existiert und ist nicht leer.

nach Eine Kopie des Inhaltes des Top-of-Stack steht im zweiten Parameter zur Verfügung.

```
procedure p_Top(kopf: T_Zeiger; var satz:T_inhalt;
var error : byte);
   begin
    if kopf <> nil then
      begin
      satz := kopf^.inhalt;
      error := 0;
      end
     else
      error := 1;
    end; // of procedure p_Top
```

Operationen auf den ADT – Stack – Test auf leer

Ist Stack leer?:

FUNC Empty (TStack [ein]):Boolean
vor Der Stack existiert.
nach Falls sich kein Element im Stack befindet, ist das Funktionsergebnis True, ansonsten False.

```
function f_empty(kopf:T_Zeiger): boolean;
   begin
     if kopf = NIL then
       f_empty := true
       else
       f_empty := false;
   end; // of function f_empty
```

Aufgabe 1

Schreiben Sie eine Delphi-Anwendung, welche die Verwaltung von Büchern unterstützt. Jedes Buch wird durch

- Titel,
- Autor,
- Verlag und
- Erscheinungsjahr

beschrieben.

Das Programm soll in der Lage sein, Buchtitel einzulesen und auf Anfrage nach Autor, Titel oder Verlag auszugeben. Die Daten sind in einer einfach verketteten Liste zu speichern. Es ist die Möglichkeit anzubieten, die Liste in einer Datei zu speichern und aus der Datei zu laden.

Aufgabe 2

Es ist eine Delphi-Anwendung zur Verwaltung von Teilnehmerdatensätzen eines Sportwettkampfes zu erstellen.

Die Daten sollen folgende Struktur haben:

```
- Name        : string[20],
- Vorname     : string[20],
- Nation      : string[3],
- Zeit_1_lauf : real,
- Zeit_2_lauf : real,
- Gesamtzeit  : real,
```

Die Informationen über Name, Vorname und Nation können aus einer Datei eingelesen oder über die Tastatur eingegeben werden.

Aufgabe 2

Während der Bearbeitung sind die Daten in einer Liste zu verwalten. Dabei sind zwei Verkettungen zu realisieren.

- Verkettung zur Realisierung einer alphabetischen Ordnung und

- Verkettung zur Realisierung einer Bestenliste.

Ausgeschiedene Teilnehmer sind aus der Liste zu löschen.

Es ist auf Wunsch des Nutzers ständig eine aktuelle Bestenliste auszugeben.

Vor Ende der Bearbeitung sind die Daten in alphabetischer Ordnung auf einem Datenträger zu speichern.

Aufgabe 3

Schreiben Sie eine Object-Pascal-Prozedur, welche einen Kinder-Abzähl-Reim simuliert. Die Namen der Kinder sind in einer Liste zu speichern. Die Liste soll eine doppelt verkettete Ringliste sein (letztes Element zeigt auf die Wurzel).

Der Reim ist in eine Zeichenkette einzulesen. Jedes Leerzeichen im Reim setzt den Zeiger ein Element weiter. Ist der Reim einmal durchlaufen, wird das entsprechende Element ausgekettet und in eine einfach verkettete Liste eingefügt.

Diese Liste ist während der Abarbeitung auszugeben. Das Programm ist beendet, wenn nur noch ein Element in der Ringliste ist. Dieser Name ist auszugeben.

Aufgabe 4

Schreiben Sie eine Delphi-Anwendung zur Überwachung Ihrer EC-Card-Ausgaben. Die Ausgaben eines Jahres werden in einer einfach verketteten Liste gespeichert. Die gespeicherten Datensätze sollen veränder- und löschbar sein.

Es ist die Möglichkeit zu schaffen, die Datensätze in eine Datei zu speichern, bzw. aus einer Datei zu lesen.

Es sind die Gesamtausgaben und die monatlichen Ausgaben zu berechnen und anzuzeigen.

Literatur

/DUDEN06/	Informatik Lehrbuch S II DUDEN - PAETEC GmbH Berlin, 2006 ISBN 3-89818-065-4
/GOLD90/	Goldschlager, Les; Lister, Andrew Informatik: eine moderne Einführung Carl Hanser Verlag, München, Wien, 1990 ISBN 3-446-15766-2
/HORN03/	Horn, Christian; Immo O. Kerner; Forbrig, Peter Lehr- und Übungsbuch Informatik Grundlagen und Überblick Fachbuchverlag Leipzig; 2003; ISBN 3-446-22543-9
/ROLLKE94/	Rollke, Karl-Hermann; Klaus Sennholz Grund- und Leistungskurs Informatik Cornelsen-Verlag-Berlin 1994; ISBN 3-464-57312-5