Über Arrays und verkettete Listen

Listen in Delphi

Michael Puff

mail@michael-puff.de

2010-03-26

Inhaltsverzeichnis

1 Einführung **3**

2 Arrays **4**

3 Einfach verkettete Listen **7**

4 Doppelt verkettete Listen **11**

Literaturverzeichnis **14**

1 Einführung

Listen gehören zu einer der elementarsten Datenstrukturen. Das liegt unter anderem daran, dass man Listen im richtigen Leben überall begegnet. Sei es in der einfachen Form einer Einkaufsliste oder sei es in Form einer Bundesligatabelle mit mehreren Spalten. In diesem Artikel soll es um die verschiedenen Möglichkeiten gehen solche Listen zu implementieren.

2 Arrays

Eine der verbreitesten Möglichkeiten eine Liste abzubilden ist wohl das *Array* bzw. Feld. Ein Array besteht aus einer Reihe von Elementen, die im Speicher direkt hintereinander liegen. Man kann also die einzelnen Elemente „anspringen“, indem man einfach einen Zeiger immer um die Größe eines Elementes im Speicher vorrückt oder zurückgeht. Dies ist ein recht einfaches und leicht verständliches Prinzip, da die einzelnen Elemente schön geordnet hintereinander im Speicher liegen.

Das hat nun allerdings auch ein paar Nachteile. Überlegen wir mal, welche Operationen erforderlich sind, um zum Beispiel ein neues Elemente in ein Array einzufügen. Als erstes muss man das Array um die Größe eines Elementes am Ende verlängern. Dann muss man alle Elemente, einschließlich des Elementes vor dem ein neues Element eingefügt werden soll, um einen Platz nach hinten kopieren. Dann kann man das neue Elemente an der gewünschten Stelle einfügen. Siehe dazu auch Skizze c) auf der Grafik am Ende der Seite. Alternativ kann man auch nur dasjenige Element nach hinten kopieren, an dessen Stelle das neue Element eingefügt werden soll. Dabei geht aber die eventuell vorhanden Sortierung verloren.

Das Löschen eines Elementes erfordert einen ähnlichen Aufwand. Entweder man kopiert das letzte Element auf den Platz des zu löschenden Elementes und verkürzt das Array um ein Elemente am Ende. Dann geht allerdings, wie beim Einfügen, die Sortierung verloren - falls vorhanden. Oder man kopiert alle nachfolgenden Elemente um einen Platz nach vorne. Beides ist aufwendig und wenn die Liste sortiert bleiben muss eventuell noch mit zusätzlicher Arbeit verbunden. Siehe Skizze c) und d).

Dazu etwas Beispielcode:

```
// Dynamische Arrays - Beispielprogramm
// Michael Puff [http://www.michael-puff.de]

program DynArrays;

{$APPTYPE CONSOLE}

type
  TDynArray = array of Integer;

var
  DynArray: TDynArray;

procedure AddElement(data: Integer);
begin
  SetLength(DynArray, Length(DynArray) + 1);
  DynArray[Length(DynArray) - 1] := data;
end;
```

```
procedure InsertElementAfter(Element: Integer; data: Integer);
var
  i: Integer;
begin
  SetLength(DynArray, Length(DynArray) + 1);
  for i := Length(dynArray) - 2 downto Element do
  begin
    DynArray[i+1] := DynArray[i];
  end;
  DynArray[Element] := data;
end;

procedure DeleteNextElement(Element: Integer);
begin
  DynArray[Element+1] := DynArray[Length(Dynarray) - 1];
  SetLength(DynArray, Length(DynArray) - 1);
end;

procedure WalkTheArray;
var
  i: Integer;
begin
  Writeln;
  for i := 0 to Length(DynArray) - 1 do
    Writeln(DynArray[i]);
end;

var
  i: Integer;
begin
  for i := 0 to 5 do
  begin
    AddElement(i);
  end;
  InsertElementAfter(4, 9);
  WalkTheArray;

  SetLength(DynArray, 0);
  for i := 0 to 5 do
  begin
    AddElement(i);
  end;
  DeleteNextElement(3);
  WalkTheArray;

  Readln;
end.
```

Noch ein paar Worte zum Speichermanagement. Wenn ein dynamisches Array vergrößert wird, passiert folgendes: Da alle Element hintereinander im Speicher liegen müssen, reserviert der Speichermanager neuen Speicherplatz, der um die Anzahl der Elemete, um die das Array vergrößert werden soll, größer ist. Dann werden alle Elemente von dem alten Speicherplatz für das Array in den neu reservierten Speicherplatz kopiert. Dann werden die neuen Elemente in den neu reservierten Speicherplätze abgelegt. Dies ist natürlich nicht

sehr effizient. Will man also mehrere Elemente in einer Schleife hinzufügen, sollte man entweder, die Länge des Arrays vorher setzen oder, wenn das nicht möglich ist, die Länge des Arrays auf die ungefähr zu erwartende Länge setzen. Und dann entweder das Array auf die tatsächliche Länge verkürzen oder das Array noch mals entsprechend verlängern.

Eine Alternative bieten sogenannte *einfach verkettete Listen*.

3 Einfach verkettete Listen

Einfach verkettete Listen zeichnen sich dadurch aus, dass ihre Elemente, bei Listen spricht man meist von Knoten, nicht unbedingt hinter einander im Speicher liegen und zusätzlich zu den eigentlichen Daten noch zusätzlich gespeichert haben, welcher Knoten der nächste in der Liste ist. Sie besitzen also eine Art Zeiger der auf das nächste Element/Knoten in der Liste zeigt. Die einzelnen Knoten sind mit einander verkettet.

Damit vereinfachen sich nun bestimmte Operationen, die bei Arrays etwas umständlich waren. Will man einen neuen Knoten einfügen, lässt man den neune Knoten auf den Nachfolger zeigen von dem Knoten nach dem eingefügt werden soll und lässt den Knoten, nach dem eingefügt werden soll auf den neuen Knoten zeigen. Skizze 3).

Auch das Löschen ist einfach. Man lässt einfach den Knoten, nach dem gelöscht werden soll, auf den übernächsten Knoten zeigen. Es wird also einfach der Knoten, der gelöscht werden soll, übersprungen.

Man merkt schon an der Kürze der Beschreibung, dass dieses Vorgehen vom Prinzip einfacher ist als bei den Arrays. Allerdings bei der Implementierung ist etwas mehr Abstraktionsvermögen gefragt. Deswegen eine beispielhafte Implementierung einer einfach verketteten Liste in Delphi:

```
// Einfach verkettete Listen - Beispielprogramm
// Michael Puff [http://www.michael-puff.de]

program SingleLinkedList;

{$APPTYPE CONSOLE}

type
  PNode = ^TNode;
  TNode = record
    data: Integer;
    next: PNode;
  end;

var
  FirstNode: PNode; // Hilfsknoten
  LastNode: PNode; // Hilfsknoten
  CurrentNode: PNode; // aktueller Knoten

procedure InitList;
begin
  FirstNode := nil;
  LastNode := nil;
end;

procedure ClearList;
```

```
var
  TempNode: PNode;
begin
  CurrentNode := FirstNode;
  while (CurrentNode <> nil) do
  begin
    TempNode := CurrentNode.Next;
    Dispose (CurrentNode);
    CurrentNode := TempNode;
  end;
  FirstNode := nil;
  LastNode := nil;
end;

procedure AddNode(data: Integer);
begin
  New(CurrentNode);
  CurrentNode.data := data;
  CurrentNode.next := nil;
  // Liste leer
  // Ersten und letzten Knoten mit neuen Knoten gleichsetzen
  // Ein Knoten in der Liste
  if LastNode = nil then
  begin
    FirstNode := CurrentNode;
    LastNode := CurrentNode;
  end
  // Liste nicht leer
  // Letzten Knoten auf neuen Knoten zeigen lassen
  // Letzten Knoten zum aktuellen Knoten machen
  else
  begin
    LastNode.next := CurrentNode;
    LastNode := CurrentNode;
  end;
end;

procedure InsertNodeAfter(AfterNode: PNode; data: Integer);
var
  NewNode: PNode;
begin
  // neuen Knoten erzeugen
  New(NewNode);
  NewNode.data := data;
  // Neuer Knoten übernimmt Nachfolger
  // vom Knoten nach dem eingefügt werden soll
  NewNode.next := AfterNode.next;
  // Knoten nach dem eingefügt werden soll,
  // zeigt auf neuen Knoten
  AfterNode.next := NewNode;
end;

procedure DeleteNextNode(Node: PNode);
var
  TempNode: PNode;
begin
  if Node.next <> nil then
```

```
  begin
    TempNode := Node.next;
    // Vorheriger Knoten über nimmt übernächste Knoten als Nachfolger
    // (Überspringen des zu löschenden Knotens)
    Node.next := Node.next.next;
    Dispose(TempNode);
  end;
end;

procedure WalkTheList;
begin
  Writeln;
  CurrentNode := FirstNode;
  while CurrentNode <> nil do
  begin
    Writeln(CurrentNode.data);
    CurrentNode := CurrentNode.next;
  end;
end;

var
  i: Integer;
  TempNode: PNode = nil;

begin
  // Test AddNode und InsertNodeAfter
  InitList;
  for i := 0 to 5 do
  begin
    AddNode(i);
    if i mod 3 = 0 then
      TempNode := CurrentNode;
  end;
  WalkTheList;
  InsertNodeAfter(TempNode, 9);
  WalkTheList;
  // Test DeleteNextNode
  InitList;
  for i := 0 to 5 do
  begin
    AddNode(i);
    if i mod 3 = 0 then
      TempNode := CurrentNode;
  end;
  WalkTheList;
  DeleteNextNode(TempNode);
  WalkTheList;
  // Liste leeren
  ClearList;
  Readln;
end.
```

Am besten man veranschaulicht sich die einzelnen Prozeduren, in dem man auf einem Blattpapier die Operationen einfach mal Schritt für Schritt nachvollzieht.

Bei genauerem Hinsehen stellt man allerdings fest, dass es ein Problem oder Nachteil gibt: Die Liste lässt sich nur in einer Richtung druchwandern. In unserem Beispiel nur von vorne

nach hinten. Das liegt daran, dass ein Knoten nur seinen Nachfolger kennt, nicht aber seinen Vorgänger.

4 Doppelt verkettete Listen

Dies kann man beheben, indem man auf eine *doppelt verkettete Liste* zurückgreift. Bei einer doppelt verketteten Liste kennt ein Knoten nicht nur seinen Nachfolger, sonder auch seinen Vorgänger. Man kann also in der Liste vor und zurück gehen.

Bei dem Beispiel für die doppelt verkettete Liste habe ich mich nur auf das Hinzufügen von Knoten beschränkt:

```
// Doppelt verkettete Listen - Beispielprogramm
// Michael Puff  [http://www.michael-puff.de]>

program DoubleLinkedList;

{$APPTYPE CONSOLE}

type
  PNode = ^Tnode;
  TNode = record
    data: Integer;
    prev: PNode;
    next: Pnode;
  end;

var
  FirstNode: PNode;
  LastNode: PNode;
  CurrentNode: PNode;

procedure Init;
begin
  FirstNode := nil;
  LastNode := nil;
end;

procedure AddNode(data: Integer);
begin
  New (CurrentNode);
  CurrentNode.data := data;
  CurrentNode.prev := nil;
  CurrentNode.next := nil;
  if LastNode = nil then
  begin
    FirstNode := CurrentNode;
    LastNode := CurrentNode;
  end
  else
  begin
    LastNode.next := CurrentNode;
    CurrentNode.prev := LastNode;
```

```
    LastNode := CurrentNode;
  end;
end;

procedure WalkForward;
begin
  Writeln;
  CurrentNode := FirstNode;
  while CurrentNode <> nil do
  begin
    Writeln(CurrentNode.data);
    CurrentNode := CurrentNode.next;
  end;
end;

procedure WalkBackward;
begin
  Writeln;
  CurrentNode := LastNode;
  while (CurrentNode <> nil) do
  begin
    Writeln(CurrentNode.data);
    CurrentNode := CurrentNode.prev;
  end;
end;

var
  i: Integer;
begin
  Init;
  for i := 0 to 5 do
  begin
    AddNode(i);
  end;
  WalkForward;
  WalkBackward;
  Readln;
end.
```

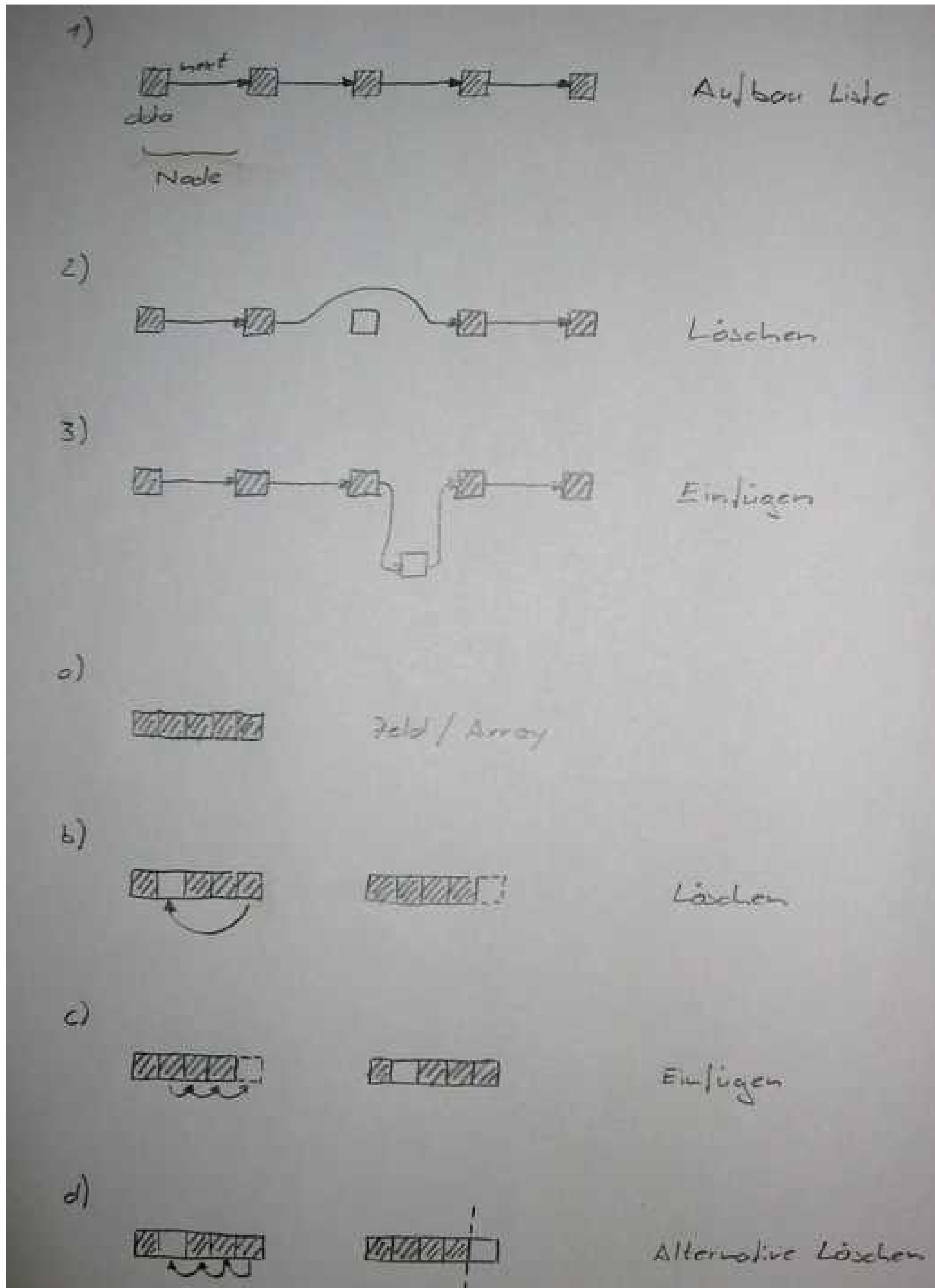

Abb. 4.1: Einfach verkettete Listen

Literaturverzeichnis

[1] Autor: Robert Sedgewick: *Algorithmen*. Addison-Wesley, 2. Auflage, 2002, 3-8273-7032-9