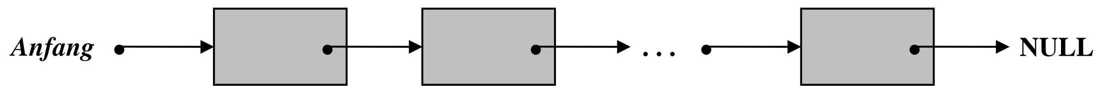
Inhalt

13 Listen und Bäume ... 13-2
13.1 Einfach verkettete lineare Listen ... 13-2
13.1.1 Vorbetrachtungen zum Programm ... 13-2
13.1.2 Programm zum Sortierproblem ... 13-4
13.2 Standardoperationen für einfach verkettete Listen ... 13-7
13.3 Spezielle verkettete lineare Listen und Bäume ... 13-9
13.3.1 Listen ... 13-9
13.3.2 Bäume ... 13-10

13 Listen und Bäume

13.1 Einfach verkettete lineare Listen



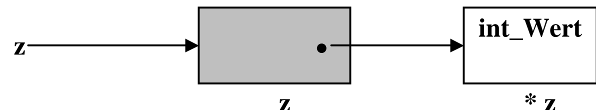
struct Punkt { struct kates kar ; struct polar pol ; struct Punkt * Nf ;}

Ein Nachteil des obigen Sortierbeispiels ist die Arbeit in einem Feld, damit muss die Länge des Feldes vereinbart werden, bevor die Daten eingelesen werden können und das aufwendige Umspeichern der Daten beim Einsortieren.

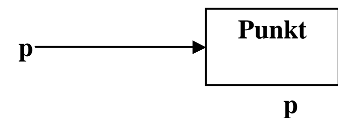
Besser wäre es, wenn jede Struktur direkt auf seinen Nachfolger verweist und nur dann eine neue Struktur und ein Verweis erzeugt wird, wenn noch Daten eingelesen werden. Beim Einsortieren müssen dann nur noch Zeiger, also Adressen umgespeichert werden.

13.1.1 Vorbetrachtungen zum Programm

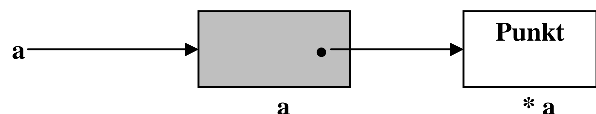
Verwendete Datentypen
int * z ;



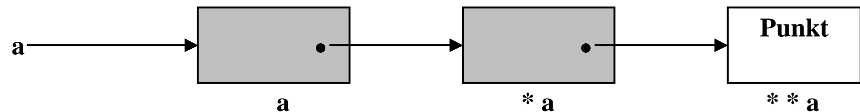
struct Punkt p ;



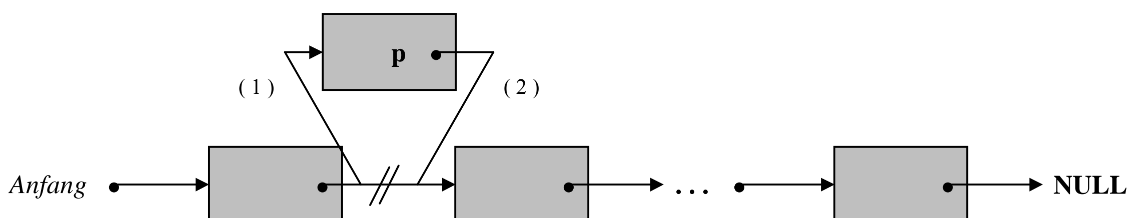
struct Punkt * a ;



struct Punkt * * a ;

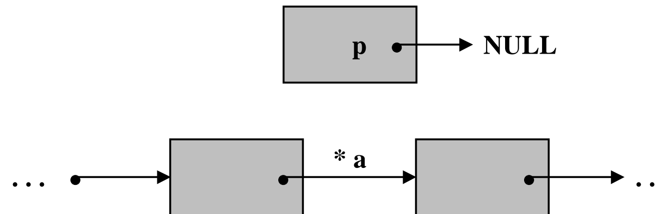


Funktion sortier_ein
Um einen neuen Punkt in eine gegebene Liste einzuordnen, sind in der Regel zwei Verweise einzutragen.

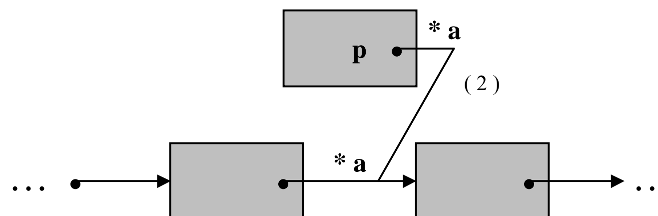


Sei **a** der Zeiger, der die Liste durchläuft. Da sich auch die *Anfangsadresse* der Liste ändern kann, muss **a** auf den *Anfang* zeigen. **a** ist also ein Zeiger auf einen Zeiger.
Angenommen wir haben die Stelle erreicht, nach der eingefügt werden muss und es ist eine Stelle mitten in der Liste (Einfügen weder am *Anfang* noch an Ende). Dann sind folgende Schritte notwendig:

1. Der Nachfolger des Vorgängerpunktes wird bestimmt. **a = & (* a) -> Nf;**

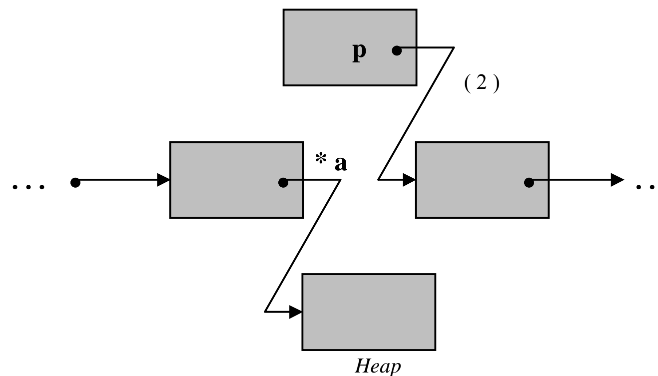


2. Der so bestimmte Nachfolger wird Nachfolger des neuen Punktes. **p . Nf = * a ;**

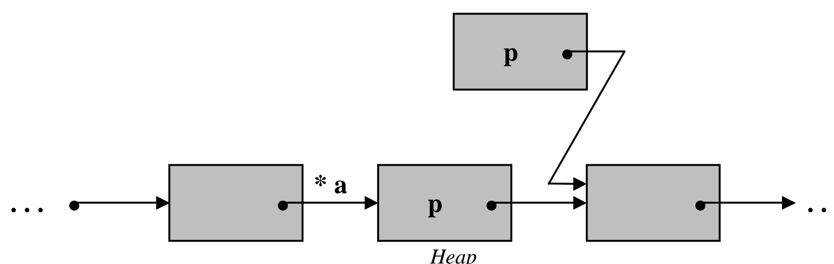


3. Es wird Speicherplatz für das neue Listenelement aus dem Heap zur Verfügung gestellt und der Zeiger des Vorgängers auf diesen Speicherplatz gestellt.

*** a = (struct Punkt *) malloc (sizeof (struct Punkt)) ;**



4. Die Daten des Punktes werden in den Heap gespeichert. *** * a = p ;**



13.1.2 Programm zum Sortierproblem

liste.h

```
/*
 * Listenoperationen
 */
# define M_PI 3.14159265359

struct kartes{ float x, y;};    /* kart. Koordinaten */
struct polar{ float rho, phi;};  /* Polarkoordinaten */
struct Punkt  /* Punkt in kart. und Polarkoordinaten */
{
  struct kartes kar;
  struct polar pol;
  struct Punkt *Nf;         /* Zeiger auf Nachfolger */
};

/*
 * Wandelt Polarkoordinaten in kartesische um
 * Parameter: struct polar    Punkt in Polarkoordinaten
 * Rueckgabe: struct kartes   Punkt in kart. Koord.
 */
struct kartes kartesisch( struct polar);

/*
 * Sortiert, geordnet nach kart. Koordinaten,
 * in eine einfach verkettete Liste ein
 * Parameter: struct Punkt ** Anfangsadresse der Liste
 *            struct Punkt    Punkt
 */
void sortier_ein( struct Punkt **, struct Punkt);
```

liste.c

```
/*
 * Listenoperationen Header liste.h
 */
# include <stdio.h>
# include <math.h>
# include <stdlib.h>
# include "liste.h"

/*
 * Wandelt Polarkoordinaten in kartesische um
 */
struct kartes kartesisch( struct polar p)
{
  struct kartes k;

  p.phi *= M_PI / 180;   /* Winkelmass --> Bogenmass */
  k.x = p.rho * cos( p.phi);           /* Umrechnung */
```

```
    k.y = p.rho * sin( p.phi);

    return k;
  }


  /*
   * Sortiert, geordnet nach kart. Koordinaten,
   * in eine einfach verkettete Liste ein
  */
  void sortier_ein( struct Punkt **a, struct Punkt p)
  {
    while( *a != NULL && ( (*a) -> kar.x < p.kar.x ||
           (*a) -> kar.x == p.kar.x &&(*a) -> kar.y < p.kar.y))
      a = &( *a) -> Nf;

    p.Nf = *a;
    *a =( struct Punkt *) malloc( sizeof( struct Punkt));
    **a = p;

    return;
  }
```

punktListe.c

```
  /*
   * Sortieren von Punkten mit Listen von Strukturen
   */
  # include <stdio.h>
  # include "liste.h"

  int main()
  {                  /* Anfangsadresse ist NULL-Zeiger */
    struct Punkt *Anfang = NULL, *a, p;

    printf( "Polarkoordinaten (Abstand/Winkel):\n");
    printf( "Schliessen Sie mit ^d ab!\n");
    printf( "\n");

  /* Einlesen der Elemente der geschachtelten Struktur */
    while( scanf( "%f%f", &p.pol.rho, &p.pol.phi) != EOF)
    {
      p.kar = kartesisch( p.pol);
      sortier_ein( &Anfang, p);
    }
    printf( "sortiert (polar/kartesisch):\n");
    for ( a = Anfang; a != NULL; a = a -> Nf)
    {
      printf( "(%f,%f) ", a -> pol.rho, a -> pol.phi);
      printf( "(%f,%f)\n", a -> kar.x, a -> kar.y);
    }
```

```
  return 0;
}
```

makefile

```
# makefile fuer punktListe

CC = gcc
CFLAGS = -ansi -Wall -O -c
LDFLAGS = -lm
OBJ = punktListe.o liste.o

punktListe: $(OBJ)
	$(CC) $(OBJ) -o punktListe $(LDFLAGS)

punktListe.o: liste.h punktListe.c
	$(CC) $(CFLAGS) punktListe.c

liste.o: liste.h liste.c
	$(CC) $(CFLAGS) liste.c

clean:
	rm -f $(OBJ)
```

13.2 Standardoperationen für einfach verkettete Listen

Zu den Standardoperationen von solchen Datenstrukturen, wie die einfach verketteten Listen, gehören:

1. **Element einfügen**
2. **Element mit bestimmten Werten suchen**
3. **Element entfernen**

1. Element einfügen
Mit der Funktion **sortier_ein** wurde das **Elementeinfügen** realisiert.

2. Element mit bestimmten Werten suchen
Das **Elementsuchen** kann analog dem Elementeinfügen programmiert werden, da dem Elementeinfügen ein Suchen der Einfügestelle vorausgeht. Ein Zeiger auf die gefundene Stelle oder, im Fall eines Misserfolges der Nullzeiger, wird zurückgegeben. Es werden aber keine Änderungen ausgeführt.

Man hat zwei Möglichkeiten:

(a) Suchen in einer sortierten Liste ⇒ Suchen vom Listenanfang an bis der Suchwert überschritten wird.
(b) Suchen in einer unsortierten Liste ⇒ Durchsuchen der ganzen Liste.

(a) Suchen in einer sortierten Liste
Dieser Fall lässt sich, ausgehen davon, dass unsere betrachtete Liste bzgl. der kartesischen Koordinaten sortiert ist, folgendermaßen lösen:

```
struct Punkt *kar_suchen
  ( struct Punkt *a, struct Punkt p)
{
  while(a != NULL && ( a -> kar.x < p.kar.x ||
                       a -> kar.x == p.kar.x &&
                       a -> kar.y < p.kar.y))
    a = a -> Nf;

  if( a != NULL && a -> kar.x == p.kar.x &&
                   a -> kar.y == p.kar.y) return a;
  else return NULL;
}
```

(b) Suchen in einer unsortierten Liste
Dieser Fall lässt sich, ausgehen davon, dass unsere betrachtete Liste bzgl. der Polarkoordinaten nicht sortiert ist, hingegen so lösen:

```
struct Punkt *pol_suchen
  ( struct Punkt *a, struct Punkt p)
{
  while(a != NULL && ( a -> pol.rho != p.pol.rho ||
```

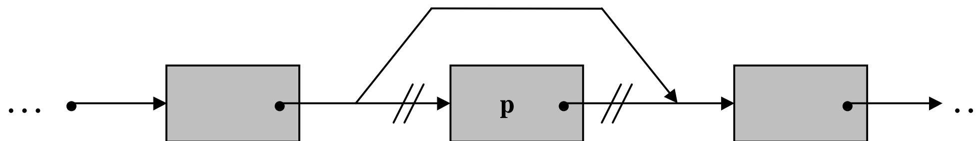
```
                                a -> pol.phi != p.pol.phi))
      a = a -> Nf;

    return a;
  }
```

Der Vorteil von **kar_suchen** ist, dass diese Funktion nicht die ganze Liste durchsucht, sondern ggf. bereits vorher abbricht. Folglich sollte man im 2. Fall bei sehr groβen Listen besser die gesuchten Polarkoordinaten in kartesische umrechnen und **kar_suchen** nutzen.

3. Element entfernen

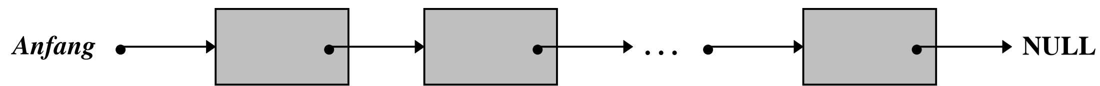
Das **Elemententfernen** soll hier nicht weiter ausgeführt werden. Es wird ähnlich dem Elementeinfügen durch Zeigermanipulationen mit anschließender Freigabe des Speicher des zu entfernenden Elements realisiert. Hier ein Schema:



13.3 Spezielle verkettete lineare Listen und Bäume

13.3.1 Listen

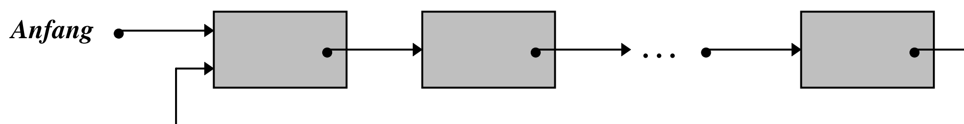
<u>Einfach verkettete lineare Listen</u>



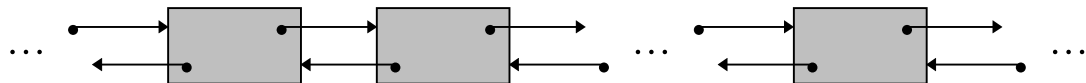
Keller (Stack, lifo-Prinzip, last in first out):
Elemente können nur am Anfang angehängt und entfernt werden.

Warteschlange (Queue, fifo-Prinzip, first in first out):
Elemente werden stets am Ende angehängt und am Anfang entfernt.

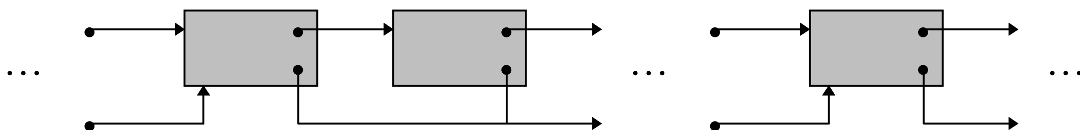
<u>Zyklisch verkettete lineare Listen</u> (Listenende zeigt auf den Anfang):



<u>Doppelt verkettete lineare Listen</u> (Verweis auf Vorgänger und Nachfolger):



<u>Mehrfach verkettete lineare Listen</u> (Verweis auf die nächste Gruppe von Elementen):



<u>Allgemeine Listen</u> erhält man, wenn man als Datentyp der Elemente wiederum Listen zulässt.

13.3.2 Bäume

<u>Bäume:</u>
Ein Baum hat ein ausgezeichnetes Element, die **Wurzel**. Diese hat *keinen* Vorgänger. Jedes weitere Element eines Baumes besitzt *genau* einen Vorgänger und *beliebig viele* Nachfolger. Elemente *ohne* Nachfolger nennt man **Blätter**.

<u>Binärbäume:</u>
Ein Binärbaum ist ein Baum, dessen Elemente *höchstens* zwei Nachfolger besitzen.

Auch hier gibt es gewisse Strategien zum Suchen, Einfügen und Entfernen von Elementen, insbesondere um große Datenmengen schnell zu verarbeiten, die aber programmiertechnisch wesentlich aufwendiger sind.