zu 4. 1. Dynamische Speicherverwaltung, Listen

Mit Hilfe der dynamischen Bereitstellung von Speicherplatz zur Laufzeit des Programms lassen sich *Listen* aus einzelnen **`struct`**-Elementen erzeugen, die durch Zeiger verkettet sind und nach Bedarf um neue Elemente erweitert werden können.

Beispielprogramm:
Eine Liste mit Daten verschiedener Personen soll angelegt, ausgedruckt und wieder gelöscht werden.

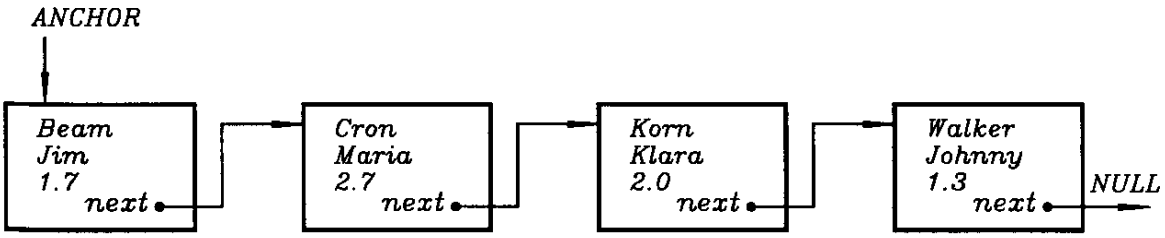
Realisierung:

- als "vorwärtsverkettete Liste"
- jedes Listenelement enthält einen Zeiger auf das nächste
- wenn kein nächstes Element existiert: **`NULL`**
- gebraucht wird im Programm auch ein Zeiger auf das Startelement (**`anchor`**) und ein Zeiger auf das aktuell letzte Element der Liste (**`aktuell`**).

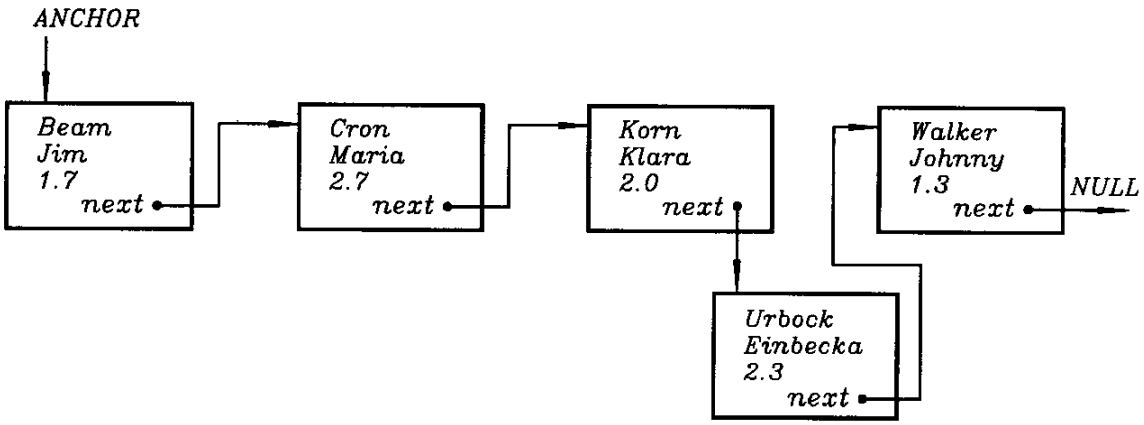
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct Student
   {
   char name[40];
   char vorname[40];
   float note;
   struct Student *next;
   };

struct Student *neues_element(char *name,
       char *vorname, float note)
  /* Funktion, die Speicherplatz für neues Element bereitstellt */
  {
    struct Student *stud_p;
    if ((stud_p = (struct Student *)
       malloc(sizeof(struct Student))) == NULL)
```

```
      {
      puts("Fehler beim Bereitstellen von"
           " Speicherplatz\n");
      exit(1);
      }
   strcpy(stud_p->name, name);
   strcpy(stud_p->vorname, vorname);
   stud_p->note = note;
   stud_p->next = (struct Student *) NULL;
   return stud_p;
   }

int main()
{
struct Student *anchor, *aktuell;

anchor = neues_element("Beam", "Jim", 1.7);
aktuell = anchor->next =
   neues_element("Cron", "Maria", 2.7);
aktuell = aktuell->next =
   neues_element("Korn", "Klara", 2.0);
aktuell = aktuell->next =
   neues_element("Walker", "Johnny", 1.3);

/* Abarbeiten der Liste: */
for (aktuell = anchor; aktuell != NULL;
        aktuell = aktuell->next)
   printf("%s, %s\t%f\n", aktuell->name,
          aktuell->vorname, aktuell->note);
/* Löschen der Liste, Wiederfreigabe des
   Speicherplatzes: */
while (anchor != NULL)
   {
   aktuell = anchor->next;
   free(anchor);
   anchor = aktuell;
   }
return 0;
}
```

- Es muss nicht vorher feststehen, wie lang die Liste wird; jederzeit kann ein neues Element (eine Struktur) ergänzt werden. Speicherplatz wird genau dann angefordert (mit **`malloc`**), wenn er benötigt wird.
- Im Beispielprogramm werden die neuen Elemente jeweils am Ende der Liste angefügt. Vor dem Abarbeiten und Ausdrucken der Liste sieht diese so aus:

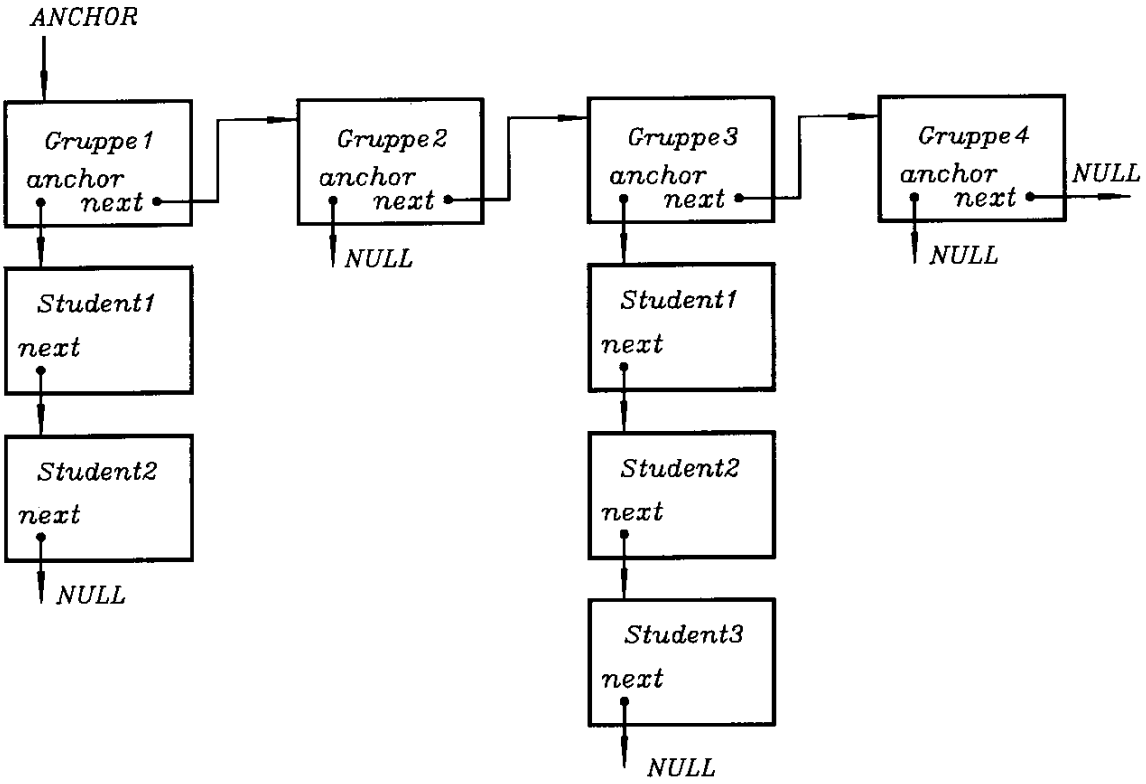


- Die gesamte Liste wird über einen einzigen Pointer, **`anchor`**, verwaltet: "Einstieg" in die Liste, die "Fortsetzungs-Informationen" befinden sich innerhalb der Liste.
- Durch die Verkettung wird eine Reihenfolge festgelegt (wie bei Arrays durch die Indizierung). Im Gegensatz zu Arrays kann aber mit geringem Aufwand ein Element an einer beliebigen Stelle in die Liste eingefügt werden: es sind dazu nur 2 Zeiger-Zuweisungen nötig.



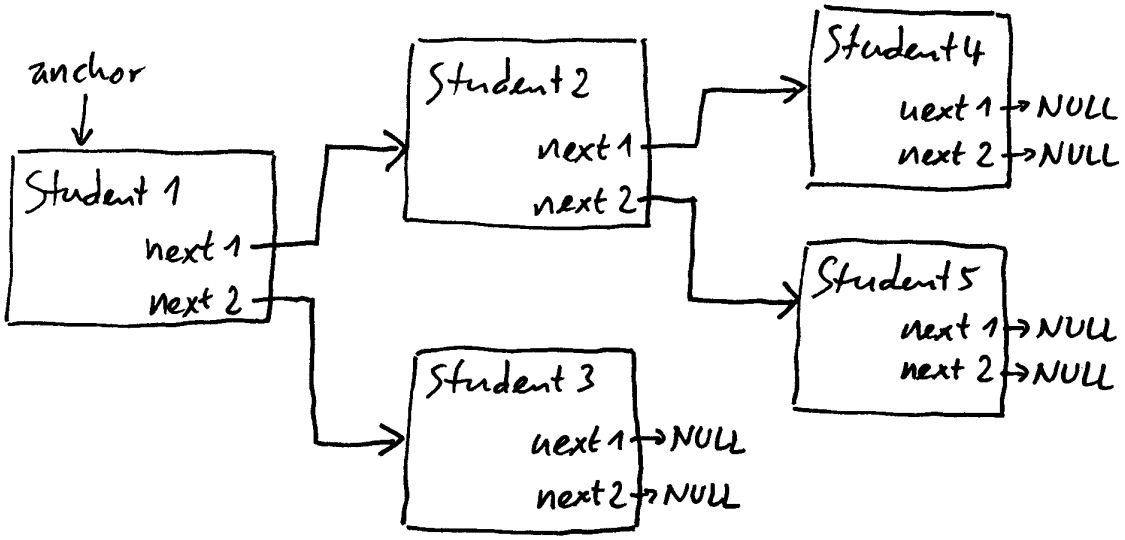
- Nachteil dieser Art von Verkettung: Die Liste kann direkt nur von vorn nach hinten durchsucht werden. Die Abarbeitung in umgekehrter Reihenfolge ist schwierig. Abhilfe: "doppelt verkettete Liste" konstruieren, wo jedes Element zusätzlich einen Zeiger auf das Vorgängerelement enthält.

Beachte: Ein Listenelement kann wieder zum Ausgangspunkt einer neuen Liste werden
⇒ es lassen sich Strukturen beliebiger Komplexität erzeugen.



(Abbildungen und Programmbeisp. nach Dankert 1997)

Wenn *jedes* Element 2 "Nachfolger-Zeiger" enthält, gelangt man zur Struktur eines *Binärbaumes*:



Anwendungen: schnelle Suchstrategien in solchen Bäumen, Repräsentation von Stammbäumen, von syntaktischen Strukturen, von Hierarchien...