# 15 Dynamische Datenstrukturen

Bei dynamischen Datenstrukturen handelt es sich um Speichermechanismen, die sich flexibel dem momentanen Speicherplatzbedarf eines Programms anpassen. Bei einem Array muss die Größe zu Beginn definiert werden und kann nicht mehr verändert werden. Das verschwendet entweder Speicherplatz oder erzeugt einen Fehler, wenn das Array überläuft, weil es voll ist!

## 15.1 Einfach verkettete Listen

### 15.1.1 Anwendungsgebiete

Wofür kann diese Datenstruktur eingesetzt werden? Prädestiniert ist sie als Stack und (in nachfolgender Implementierung mit der Referenz listEnd) als Queue.
Ein Stack ist eine FILO (First In, Last Out) Datenstruktur. Das bedeutet, dass das erste Datenelement, das gespeichert wird, das letzte ist, das wieder aus der Datenstruktur herausgenommen werden kann. Man kann sich das, wie der Name schon andeutet, als einen Stapel z.B. Bücher auf einem Tisch vorstellen. Das Buch, das man als erstes auf die Tischplatte gelegt hat, bekommt man erst (ohne Probleme) wieder, wenn all die Bücher, die man auf dieses gestapelt hatte, wieder entfernt wurden. Der Stack ist in der Informatik eine sehr wichtige Datenstruktur.
Er wird z.B. im Bereich des Compilerbaus und, während der Ausführung eines Programms, bei jeder Parameterübergabe an eine Funktion / Prozedur / Methode eingesetzt. Die Implementierung mit Hilfe einer linearen Liste sieht so aus: Neue Elemente werden vorne in die Liste eingefügt und auch von dort wieder weggenommen (lesen und löschen des ersten Knotens in der Liste).

Die Queue ist das Gegenstück zum Stack, eine FIFO (First In, First Out) Datenstruktur. Sie wird z.B. als Puffer für eintreffende Nachrichten verwendet: die erste Nachricht, die in den Puffer geschrieben wird, ist auch die erste, die aus diesem gelesen wird. Bei der Implementierung würde am nächsten liegen, neue Datenelemente vorne in die Liste einzufügen und am Ende zu entnehmen.

Ein weiteres Einsatzgebiet besitzt die lineare Liste überall dort, wo Daten zunächst einmal gesammelt und dann immer wieder komplett durchlaufen werden müssen.

### 15.1.2 Aufbau

Einfach verkettete Listen erzeugt man, indem man eine Struktur für ein einzelnes Listenelement erschafft und diese dann je nach Bedarf aneinander hängt.
Zuerst benötigt man also eine Struktur in C für ein solches Listenelement:

```
typedef struct LISTELEMENT{
      int data;
      struct LISTELEMENT *next;
}ListElement;
```

Die Struktur trägt den Namen LISTELEMENT und durch den Befehl typedef erzeugen wir aus dieser Struktur den Datentyp ListElement. Dieser enthält nur zwei Felder, nämlich eines, das die Daten trägt, die gespeichert werden sollen und ein weiteres Feld, das ein Pointer auf die Struktur LISTELEMENT ist.

Grafisch sieht eine solche Liste dann wie folgt aus:

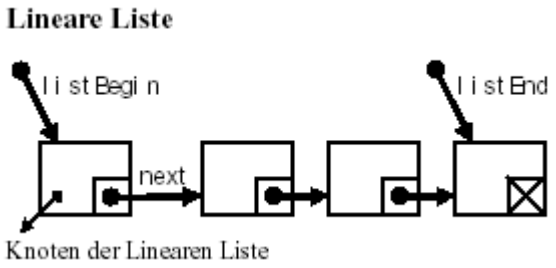
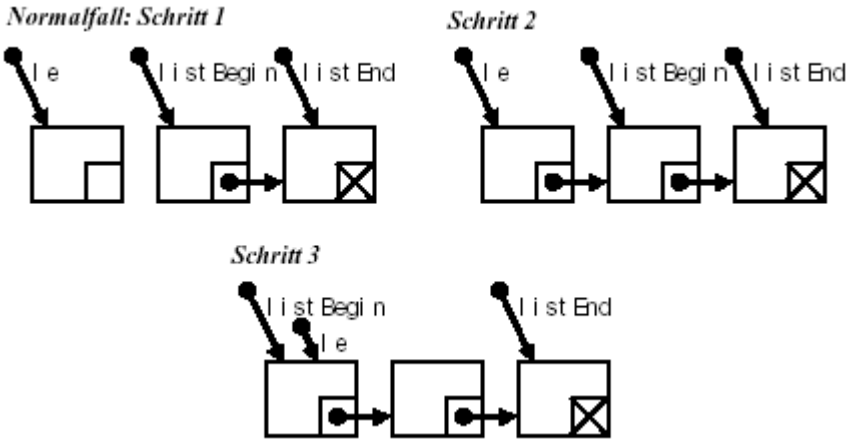



Abbildung: Einfach-verkettete Liste

Das erste Element enthält Daten und einen Zeiger auf das folgende Element. So geht das immer weiter, bis schließlich das Ende der Liste erreichet wird. Hier befindet sich ein Element vom Typ ListElement, bei dem next auf NULL initialisiert wurde, um das Ende der Liste zu markieren.

### 15.1.3 insertElement()

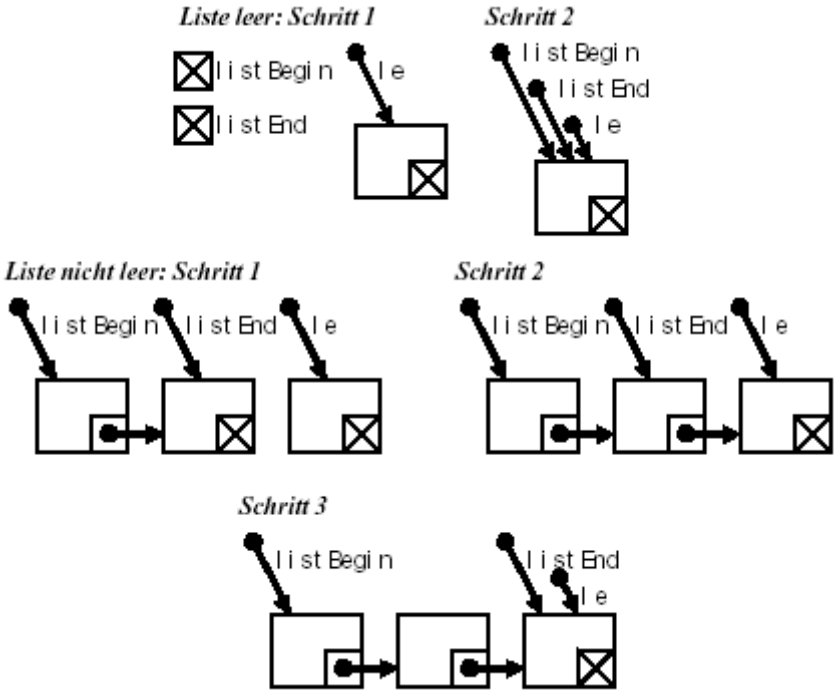
Diese Methode gehört wohl zu den Grundfunktionen einer Datenstruktur. Für die lineare Liste bedeutet diese Operation, dass ein neuer Knoten am Beginn der Liste eingefügt wird.
Der Aufbau von insertElement() ist relativ einfach. Das in dem Knoten zu speichernde Datenelement wird der Methode als Parameter übergeben (int data). Jetzt wird ein neues ListElement benötigt, da ja ein neuer Knoten in die Liste eingehängt werden soll (malloc()). Danach folgen die Operationen zum Einhängen des neuen Knotens in die Liste. Dessen Referenz next muss auf den bisherigen Listenbeginn verweisen. Mit der Aktualisierung von listBegin ist das neue Listenelement korrekt eingefügt.




```
void insertElement(int data, ListElement **listB, ListElement **listE)
{       ListElement *le=NULL;

        if ((le=(ListElement *)malloc(sizeof(ListElement)))==NULL){
                fprintf(stderr,"Nicht genuegend Speicher verfuegbar!"); return ;
        }

        le->data=data;
        le->next=*listB;
        *listB=le;

        if (*listE==NULL){
                *listE=le;
        }
}
```

### 15.1.4 appendElement()

Mit dieser Funktion wird ein neuer Knoten am Ende der linearen Liste angehängt. Diese Operation wird z.B. bei der Implementierung einer Queue benötigt.



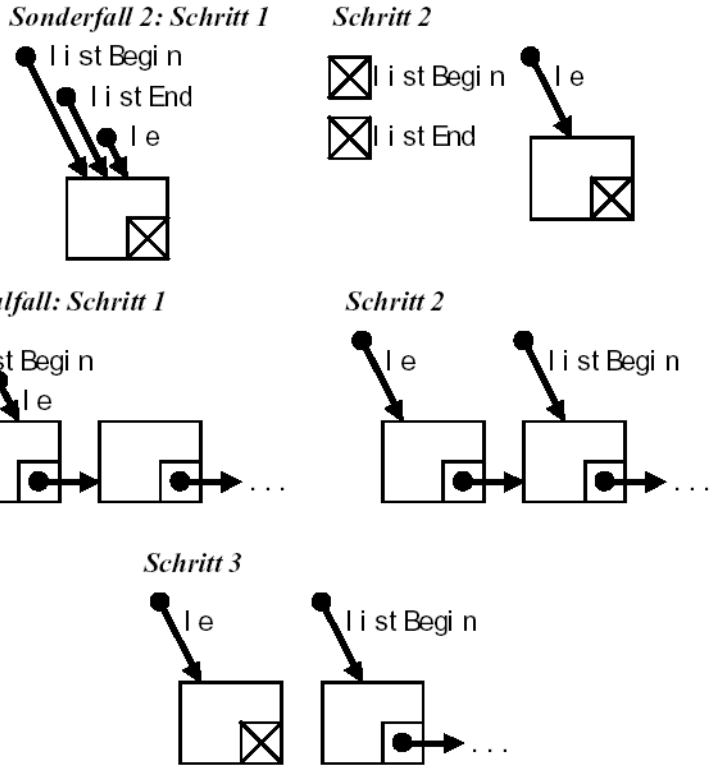

Quellcode:

```
void appendElement(int data, ListElement **listB, ListElement **listE){

	ListElement *le=NULL;

	if ((le=(ListElement *)malloc(sizeof(ListElement)))==NULL){
		fprintf(stderr,"Nicht genuegend Speicher verfuegbar!"); return;
	}
	le->data=data;
	le->next=NULL;

	if (*listB==NULL){
		*listB=le;
		*listE=le;
	}
	else{
		(*listE)->next=le;
		*listE=le;
	}
}
```

### 15.1.5 deleteFirstElement()

Grundsätzlich benötigt man, falls man einen Stack oder eine Queue implementieren möchte, nur die Entfernung des ersten Listenelementes zu programmieren.
Sonderfälle:

1. die Liste ist leer
2. es befindet sich nur ein Element in der Liste
3. Normalfall




```
int deleteFirstElement(int *data, ListElement **listB, ListElement
**listE){
      ListElement *temp;

      // Elementarprüfung: Ist die Liste leer?
      if (*listB==NULL){
            fprintf(stderr,"%s","Fehler: Liste ist noch leer\n");
            *listE=NULL;
            *listB=NULL;
            return EXIT_FAILURE;
      }
      //ist nur ein Element in der Liste?
      if ((*listB)->next==NULL){
            *data=(*listB)->data;
            free(*listB);
            *listE=NULL;
            *listB=NULL;
      }
      else{ // 1.Element übergeben, Speicher freigeben
                  *data=(*listB)->data;
                  temp=(*listB)->next;
                  free(*listB);
                  *listB=temp;
      }
      return EXIT_SUCCESS;
}
```

### 15.1.6 ClearAllElements()

Diese Funktion dient zum vollständigen Entfernen der Liste aus dem Speicher.

```
void ClearAllElements(ListElement **listBegin, ListElement **listEnd){

	ListElement *le;

	while(*listBegin!=NULL){
		le=(*listBegin)->next;
		free(*listBegin);
		*listBegin=le;
	}
	*listEnd=NULL;
}
```

### 15.1.7 deleteElement()

In dem nachfolgenden Programmfragment wird allerdings auch die beliebige Entfernung eines innerhalb der Liste vorkommenden Elementes gezeigt.
Wenn man ein Element innerhalb der Liste löschen möchte, so muss man einfach dafür sorgen, dass dieses Element in der Liste übersprungen wird.
Man setzt also den Zeiger des Vorgängerelementes auf das Nachfolgeelement. Das zu löschende Element besitzt zwar noch einen Zeiger auf seinen Nachfolger, das spielt aber keine Rolle, da auf das Element selbst kein Zeiger mehr existiert, es also nie erreicht werden kann.

Allerdings gilt es einige Sonderfälle zu beachten:

1. die Liste ist leer
2. es befindet sich nur ein Element in der Liste
3. soll das erste Element gelöscht werden
4. soll das letzte Element gelöscht werden
5. Normalfall

### 15.1.8 dataOutput()

Das folgende Programmfragment soll der Vollständigkeit dienen und zeigt die einfache Ausgabe der gesamten Liste.

```
void dataOutput(ListElement *listB, ListElement *listE){

	ListElement *le=NULL;

	le=listB;
	while (le!=NULL){
		printf("%d \n",le->data);
		le=le->next;
	}
}
```