Fachhochschule Bingen

Programmieren

Dynamische Speicherbelegung: Datenobjekte

Prof. Dr. Maximilian Mengel,
Professur Programmiermethodik,
Grundlagen der Informatik und Multimedia
Gebäude 1, Raum 212
Tel.: 06721-409 152
E-Mail: mengel@fh-bingen.de

Dynamische Datenobjekte

- Jegliche Datenobjekte können
  - Statisch als Variablen angelegt werden
  - Dynamisch angelegt werden, die dann über Pointer benutzt werden
- Verschiedene Datenobjekte sind nur als dynamisch angelegte Datenobjekte wirklich sinnvoll zu nutzen
  - Listenelemente
  - Bäume

Dynamische Datenobjekte

- Dynamische Datenobjekte sind normalerweise als Strukturen aufgebaut:

```
typedef struct listInt {
   int Element;
   struct listInt *next;
   struct listInt *prev;
   } t_listInt;
// ab hier können struct listInt und t_listInt
// synonym füreinander benutzt werden; innerhalb der
// Vereinbarung kann nur struct listInt benutzt
// werden; t_listInt ist noch nicht bekannt!
```


Zugriff auf dynamische Strukturen

- Um auf dynamische Strukturen zuzugreifen braucht man den Wert der Variable auf die der Pointer zeigt; von diesem Wert kann dann das Strukturelement ausgewählt werden:

```
t_listInt * pList;
…
(* pList).Element = 0;
```

- Um dies einfacher zu schreiben existiert der Pfeil-Operator: ->

```
pList->Element = 0 // synonym zu obigem Ausdruck
```

Listen

- Eine Liste ist eine Folge von Elementen.
- Eine Liste kann erweitert werden, indem neue Elemente eingefügt werden
- Elemente einer Liste können gelöscht werden
- Eine Liste besitzt einen Anfang und ein Ende
  - Bei einer leeren Liste sind Anfang und Ende ebenfalls leer
  - Bei einer Liste mit einem Element zeigen Anfang und Ende auf dieses Element

Listen: Beispiel

- Beispiel:

```
typedef struct listInt {
   int Element;
   struct listInt *next;
   struct listInt *prev;
   } t_listInt;
...
t_listInt *pListenAnfang, *pListenEnde;
pListenAnfang = (t_listInt *) malloc(sizeof(t_listInt));
if (pListenAnfang == NULL)
   printf(„Fehler bei malloc!“);
else
{
   pListenEnde = pListenAnfang;
   (*pListenAnfang).Element = 0;  // pListenAnfang->Element = 0
   pListenAnfang->next = NULL;
   pListenAnfang->prev = NULL;
}
```


Eigenschaften von Listen

- Die Länge einer Liste L entspricht der Anzahl der Elemente
- Jedes Element hat eine Position P innerhalb der Liste mit:
  - $0 \leq P \leq L-1$
- Typische Listenoperationen
  - IstLeer / Laenge / Anfang / Suche / Einfuegen / Loesche / NaechstesElement / PositionVon / …

Einfach und doppelt verkettete Listen

- Einfach verkettete Liste
  - Bei der einfach verketteten Liste besitzt jedes Element einen Verweis zu seinem Nachfolger
  - Sequentielles durchsuchen ist möglich
  - Bei dem letzten Element der Liste ist der Verweis zum Nachfolger im Allgemeinen leer
- Doppelt verkettete Liste
  - Jedes Element besitzt Vorgänger und Nachfolger
  - Das erste Element besitzt keinen Vorgänger
  - Das letzte Element besitzt keinen Nachfolger

Bäume

- Bäume können als eine Art mehr-dimensionale Erweiterung von Listen angesehen werden
  - Bäume bestehen aus Knoten und aus Blättern
  - Ein Knoten hat mehrere Söhne
    - Wiederum Knoten
    - Blätter
  - Blätter besitzen keine Söhne
  - Jeder Knoten und jedes Blatt (bis auf die Wurzel) hat einen Vater

Arten von Bäumen

- Es existieren Bäume bei denen:
  - Daten nur in den Blättern gespeichert werden
  - Daten in Knoten und Blättern gespeichert werden
- Bäume mit jeweils (potentiellen) zwei Söhnen heißen Binäre Bäume
- In einem Baum können
  - Neue Elemente eingefügt werden
  - Elemente gesucht werden
  - Elemente gelöscht werden

Beispiel Bäume

- Beispiel

```
typedef struct treeInt {
   int Element;
   struct treeInt *left;
   struct treeInt *right;
   } t_treeInt;
...
t_treeInt *pWurzel;
pWurzel = (t_treeInt *) malloc(sizeof(t_treeInt));
if (pWurzel == NULL)
   printf(„Fehler bei malloc!“);
else
{
   (*pWurzel).Element = 0;  // pWurzel->Element = 0
   pWurzel->left = NULL;
   pWurzel->right = NULL;
}
```


Übung

- Ein Feld-basierter LIFO Stack hat normalerweise eine beschränkte Größe
- Implementieren Sie einen LIFO-Stack für Integer-Werte, der beliebig viele Elemente aufnehmen kann
  - Implementieren Sie den Stack Listen-basiert
  - Bei Push soll ein neues Listenelement angelegt werden, in dem dann der entsprechende Wert abgespeichert wird
  - Bei Pop soll der gespeicherte Wert zurückgegeben und das Listenelement freigegeben werden