Inhaltsverzeichnis

1. Datenstrukturen in C....1
1.1. Ziele. Voraussetzungen....1
1.2. Listen....1
1.2.1. Das Einfügen in eine verkettete Liste....5
1.2.2. Das Löschen aus einer verketteten Liste....6
1.2.3. Das Demoprogramm t_minish.c....10
1.3. Aufgaben: Listen....13
Aufgabe: tminish.c, o_strlist.h, o_strlist.c....13
+Aufgabe: Dlist....13
Aufgabe: sort....13
Aufgabe: InsertSorted....13
1.4. Bäume....13
1.4.1. Die Haederdatei eines Binären Suchbaumes....14
1.4.2. Das Traversieren binärer Bäume....15
1.4.3. Suchen....17
1.4.4. Einfügen....17
1.4.5. Das Löschen eines Elementes....18
1.4.6. Aufgaben: Binäre Bäume....18
+Aufgabe: delBinSearch....18
Aufgabe: BinTreeADB....18
1.5. Zusammenfassung....19
1.6. Ausblick....19

1. Datenstrukturen in C

1.1. Ziele. Voraussetzungen

☑ Komplexe, flexible Datenhaltung im RAM einsetzen und erstellen können

☑ Voraussetzungen:

- ☐ Grundlagen der Programmierung in C

☑ Quellen:

- ☐ www.pronix.de

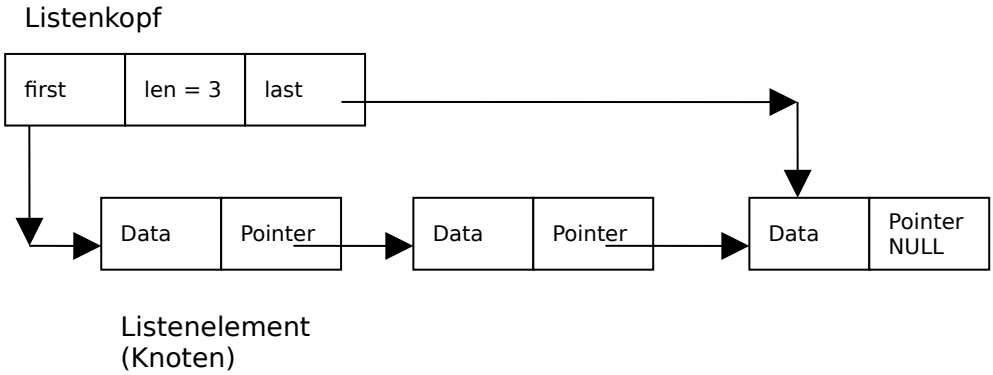
1.2. Listen

Eine **verkettete** Liste ist eine Datenstruktur, in der Objekte linear geordnet sind.

In Arrays ist dies durch die Indizierung der Arrayelemente gegeben. Da die Arraygröße aber zur Übersetzungszeit bekannt sein muss, kann dies u. U. zu Problemen führen (1. Array wird zur Laufzeit zu klein oder 2. Es wird zu viel Speicherplatz verschwendet).

Verkettete Listen sind hier flexibler, da von jedem Element auf den Nachfolger verwiesen wird. Unter C verwendet man dafür Zeiger.

Die folgende Grafik zeigt eine sog. einfach gekettete Liste mit einem Listenkopf:



Ein Listenelement (man sagt auch Knoten) besteht aus folg. 2 Komponenten:

- Das zu speichernde Datum (int, float, struct person , ..., char *void)
- Zeiger auf das nächste **Listenelement**
  Hinweis: Beim letzten Listenelement wird diese Komponente mit einem **NULL**-Pointer belegt.

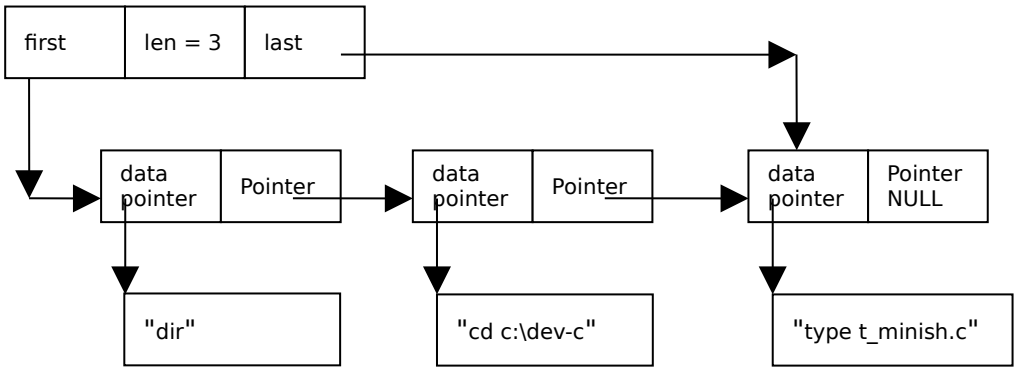
```
/* -- SLIST Knoten */
struct SList {
  char *Data;
  struct SList *Next;
};
typedef struct SList  SLIST;
```

Es gibt verschiedene Arten der Realisierung. Hier wird ein sog. Listenheader verwendet, der aus 3 Komponenten besteht:

- Zeiger auf das erste Listenelement
- Zeiger auf das letzte Listenelement
- Anzahl der Listenelemente

```
/* -- Ein Kopf-Knoten, der die Listenlänge enthält */
struct SList_Header {
  int Len;
  SLIST *First, *Last;
};
typedef struct SList_Header SLIST_HEADER;
```

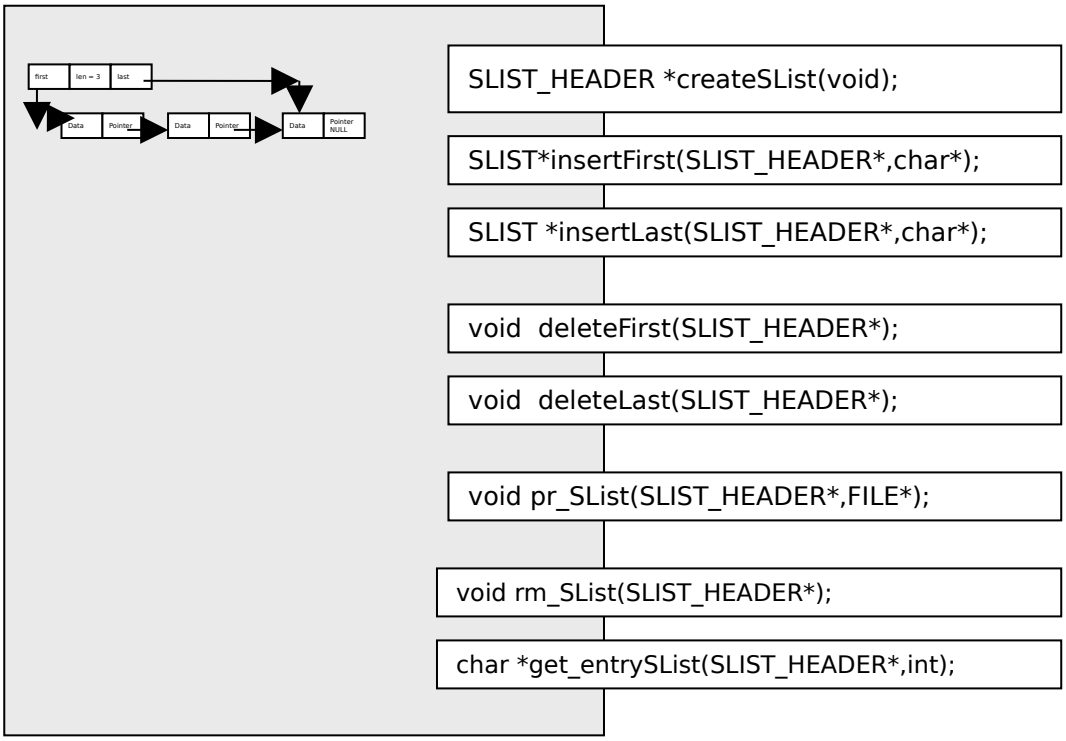
Wir wollen im folgenden das Programm *t_minish* entwickeln, das eine so genannte Historie der eingegebenen Befehle in Form einer Liste speichern soll. D.h. ein Listenknoten enthält einen String (=das eingegebene Kommando). Die folgende Grafik zeigt diesen Sachverhalt.




```
// Grobentwurf von t_minish

#include "o_strlist.h"
…
int main()
{
     SLIST_HEADER* cmdList;
     cmdList = createSList();

     printf(prompt);
     fgets (eingabe);

     while (eingabe != "quit")
          insertFirst (cmdList, eingabe);
          exec (eingabe);
….
}
```

Im Sinne des Information Hiding wollen wir mittels **Datenkapselung** den Zugriff auf die tatsächlichen Knoten und Listenheader nur mittels **Zugriffsfunktionen** realisieren. D.h. wir bauen einen ADT (Abstrakten DatenTyp) namens SLIST.

Dazu überlegen wir zunächst, welche Operationen/Funktionen (=Verhalten der Liste) wir benötigen. D.h. wir spezifizieren das Interface des ADT (vgl. Software IC)



Nachdem das Interface festgelegt ist, können wir die C-Headerdatei erstellen:

```
/**
 * Datei: o_strlist.h   Hofmann Anton
 * Headerdatei fuer Typvereinbarungen.
 * Einfach verkettete Listen als ADTs.
 **/

/* -- SLIST Knoten */
struct SList {
  char *Data;
  struct SList *Next;
};
typedef struct SList  SLIST;


/* -- Ein Kopf-Knoten, der die Listenlaenge enthaelt */
struct SList_Header {
  int Len;
  SLIST *First, *Last;
};
typedef struct SList_Header SLIST_HEADER;

/* -- Zugriffsfunktionen */
extern SLIST_HEADER *createSList(void);

extern SLIST *insertFirst(SLIST_HEADER*,char*);
extern SLIST *insertLast(SLIST_HEADER*,char*);

extern void  deleteFirst(SLIST_HEADER*);
```

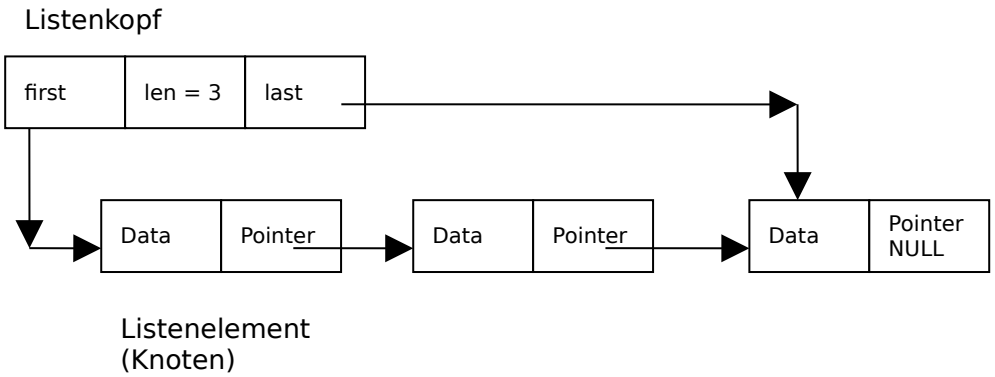
```
extern void  deleteLast(SLIST_HEADER*);

extern void pr_SList(SLIST_HEADER*,FILE*);

extern void rm_SList(SLIST_HEADER*);

extern char *get_entrySList(SLIST_HEADER*,int);
```

Bevor wir die Implementierungsdatei **o_strlist.c** erstellen, wollen wir uns das Einfügen und Löschen genauer ansehen. Wir beschränken uns auf das Einfügen/Löschen am Listenanfang/Listenende, weil dadurch auch andere Operationen realisiert werden können.
LIFO: Stack: push, pop
FIFO: Queue: enqueue, dequeue

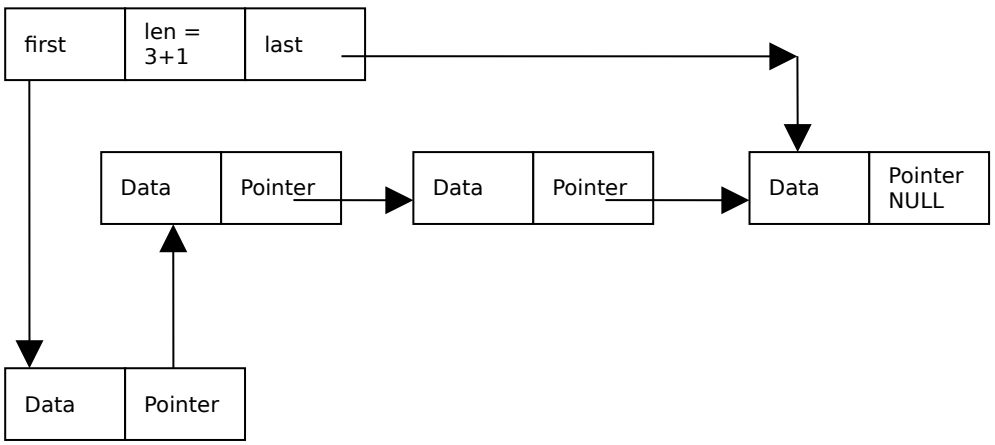
1.2.1. Das Einfügen in eine verkettete Liste

Hier muss man folgende Schritte realisieren:
1. **Speicherplatz** für neuen Knoten anlegen
2. Neuen Knoten mit den richtigen **Werten** belegen
3. **Listenhaeder** aktualisieren

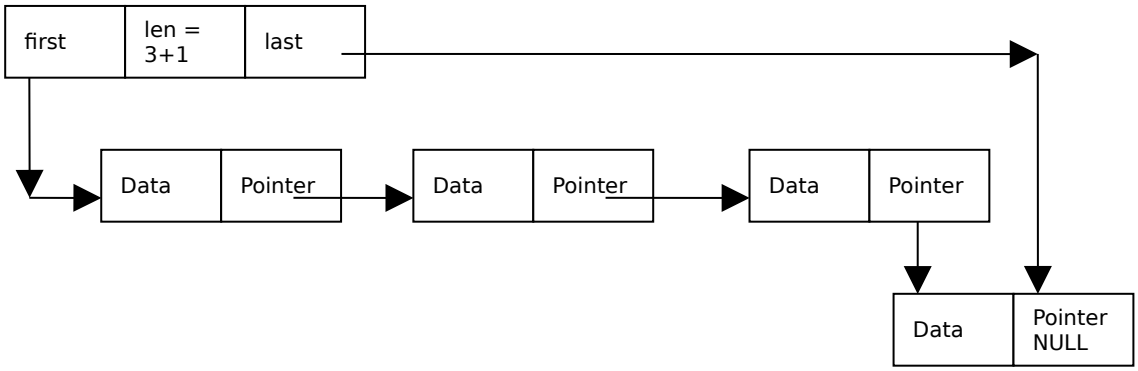
Vor dem Einfügen:



Nach dem Einfügen am Listenbeginn:

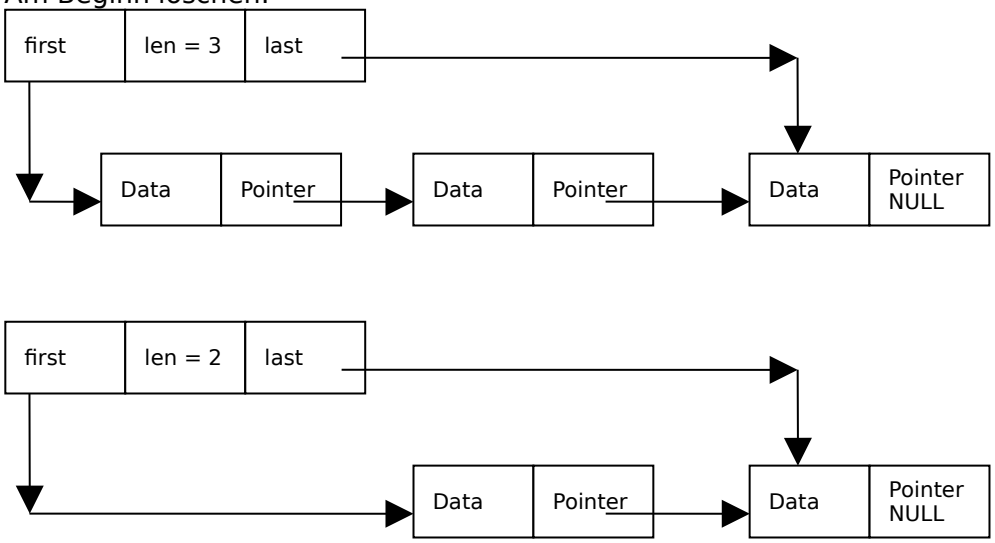


Nach dem Einfügen am Listenende:



1.2.2. Das Löschen aus einer verketteten Liste

Am Beginn löschen:



Damit können wir die tatsächliche Implementierung erstellen:

```
/**
 * Datei: o_strlist.c    Hofmann Anton
 * Einfach verkettete Listen als ADTs.
 **/

#include <stdio.h>
#include <stdlib.h>
#include "o_strlist.h"

/**
 * Lokale OPERATIONEN, die privat gehalten werden.
 *-----------------------------------------------*/
```

```
static SLIST *makeSListNode (char * userData, SLIST * aPtr)
{
SLIST *newPtr;    /* Zeiger auf zugewiesenen Speicher */

if ((newPtr=(SLIST *) malloc(sizeof(SLIST))) != NULL)
{
  newPtr->Data= (char *) malloc(strlen(userData)+1);
  if (newPtr->Data == (char *) NULL)
  {
     free (newPtr);
     return((SLIST *)NULL);
  }
  else
  {
     strcpy(newPtr->Data, userData);
     newPtr->Next= aPtr;
  }
}
return (newPtr);
}


/**
 * Von aussen sichtbare ZUGRIFFS-OPERATIONEN
 * ----------------------------------------*/

/* -- Eine neue Liste aufbauen */
SLIST_HEADER * createSList ()
{
SLIST_HEADER * aList;

if ((aList= (SLIST_HEADER *) malloc(sizeof(SLIST_HEADER))) != NULL)
{
  aList->Len= 0;
  aList->First= (SLIST *) NULL;
  aList->Last= (SLIST *) NULL;
}
return (aList);
}


/* -- Am Beginn der Liste eintragen */
SLIST *insertFirst(SLIST_HEADER* aList, char* userData)
{
SLIST *newPtr;
if ((newPtr= makeSListNode(userData, aList->First))!= NULL)
{
  aList->First= newPtr;
  if (aList->Len == 0)  /* -- erster Eintrag ? */
    aList->Last= newPtr;

  aList->Len++;
}
return (newPtr);
```

```
}


/* -- Am Ende der Liste eintragen */
SLIST *insertLast(SLIST_HEADER* aList, char* userData)
{
SLIST *newPtr;
if ((newPtr= makeSListNode(userData, (SLIST*) NULL))!= NULL)
{
  if (aList->Len != 0)  /* -- Liste nicht leer ? */
    aList->Last->Next= newPtr;
  else
    aList->First= newPtr;

  aList->Last= newPtr;
  aList->Len++;
}

return (newPtr);
}


/* -- Loeschen des ersten Listeneintrages */
void  deleteFirst(SLIST_HEADER* aList)
{
SLIST *temp;
char *userData;

if (aList->Len > 0)   /* Liste nicht leer */
     {
     temp= aList->First;
     userData= temp->Data;

     aList->First= temp->Next;
     aList->Len--;

     if (aList->Len == 0) /* Wenn die Liste leer ist, Last-Zeiger auf
                             NULL */
       aList->Last= (SLIST *) NULL;

     free( (void *) temp->Data);   /* String freigeben */
     free( (void *) temp);         /* Knoten freigeben */
     }
}


/* -- Loeschen des letzten Listeneintrages */
void  deleteLast(SLIST_HEADER* aList)
{
SLIST *temp, *help;
char *userData;
int i;

if (aList->Len > 0)   /* Liste nicht leer */
```

```
     {
     temp= aList->Last;
     userData= temp->Data;

     /* suche den Vorletzen, aList->Last soll darauf zeigen */
     help= aList->First;
     aList->Last= help;
     for (i=0; i<aList->Len- 1 ; i++){
      aList->Last= help;
      help= help->Next;
     }
     aList->Len--;

     if (aList->Len == 0) /* Wenn die Liste leer ist, Last-Zeiger auf
                             NULL*/
     {
       aList->Last= (SLIST *) NULL;
       aList->First= (SLIST *) NULL;
     }
     free( (void *) temp->Data);   /* String freigeben */
     free( (void *) temp);         /* Knoten freigeben */
     }
}


/* -- Die Liste ausgeben */
void pr_SList(SLIST_HEADER * aList, FILE* device)
{
int i;
SLIST *node;

node= aList->First;
for (i=1; i<= aList->Len;i++){
   fprintf(device,"%6d\t%s\n",i,node->Data);
   node= node->Next;
   }
}

/* -- Loeschen der gesamten Liste */
void rm_SList(SLIST_HEADER* aList)
{
while (aList->Len)
  deleteLast(aList);

free (aList); /* alle Knoten geloescht, Header noch freigeben */
}



/* -- Das Element an der n-ten Stelle der Liste erhalten */
char *get_entrySList(SLIST_HEADER* aList, int number)
{
if (number>=1 && number<=aList->Len)
{
```

```
   int i;
   SLIST *node;

   node= aList->First;
   for(i=1;i<number;i++)
     node= node->Next;
   return(node->Data);
}
else
   return((char *) NULL);
}
```

1.2.3. Das Demoprogramm t_minish.c

Somit können wir zum Schluss noch das Programm t_minish erstellen, dass die Verwendung des Listen ADTs zeigen soll:

```
/*
 * Datei: t_minish.c     Hofmann Anton
 * gcc t_minish.c o_strlist.c -o t_minish
 * Nutzt die verkettete Liste (o_strlist.h o_strlist.c)
 * Implementiert eine minishell
 */

#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include "o_strlist.h"

enum BOOL {false,true};
typedef enum BOOL BOOLEAN;


char help[]=
  "\nMINISHELL::HELP\n \
  hi   ............. history\n \
  !#   ............. execute #th command\n \
  !save Filename ... save history list\n \
  !load Filename ... load history list from Filename\n \
  !do  ............. execute current history list\n \
  quit ............. quit MINISHELL\n\n";

char prompt[]="\nminish-> ";


#define MAXCMD   128
char cmd[MAXCMD];           /* enthaelt eingelesenes Kommando */
SLIST_HEADER * cmdSList;    /* History Liste                  */
void do_history();
```

```
main()
{
BOOLEAN end=false;


/* ------------------------------------------
 * Liste zur Speicherung der Kommandos erzeugen
 * ------------------------------------------*/
cmdSList= createSList();
if (cmdSList == (SLIST_HEADER *) NULL)
{
  fprintf(stderr,"Not enough memory\n");
  exit(1);
}


/* ----------------------
 * Logo und help ausgeben
 * ----------------------*/
printf("\nMINISHELL:: (Hofmann Anton, ::Demo fuer verkettete Listen\n");
printf(help);


/* ------------------------------------------
 * Hauptschleife zur Verarbeitung der Eingaben
 * ------------------------------------------*/
while (end==false){
   /* -- Prompt ausgeben und Kommando einlesen */
   printf(prompt);
   fgets(cmd,MAXCMD,stdin);
   cmd[strlen(cmd)-1]='\0';

   /* -- Kommando verarbeiten */
   if (strcmp(cmd, "hi")==0)        /* -- history ausgeben */
      pr_SList(cmdSList, stdout);

   else if (strcmp(cmd, "help")==0) /* -- Hilfe ausgeben   */
      printf(help);

   else if (cmd[0] == '!')          /* -- Kommando aus history */
      {
      do_history();
      }

   else if (strcmp(cmd, "quit")==0) /* -- MINISHELL beenden  */
      {
      rm_SList(cmdSList); /* Liste freigeben */
      end= true;
      }
   else                             /* -- Kommando in history eintragen
                                          u. ausfuehren */
      {
      insertLast(cmdSList, cmd);
```

```
      system(cmd);
      }
}/* end_while */
}/*end_main*/



void do_history()
{
int number;
FILE *fp;

if (strncmp(cmd+1, "save", 4) == 0)   /* -- SAVE history */
{
   fp= fopen(cmd + 6, "w");
   if (fp==(FILE *) NULL)
      perror(cmd+6);
   else
   {
      pr_SList(cmdSList, fp);
      fclose(fp);
   }
}
else if (strncmp(cmd+1, "load", 4) == 0) /* -- LOAD history list */
{
   fp= fopen(cmd + 6, "r");
   if (fp==(FILE *) NULL)
      perror(cmd+6);
   else
   {
      rm_SList(cmdSList);     /* remove history list and build a
                                 new one from file */
      cmdSList= createSList();
      if (cmdSList == (SLIST_HEADER *) NULL)
      {
         fprintf(stderr,"Not enough memory\n");
      }
      else
      {
      while(fgets(cmd, MAXCMD, fp))
            {
           cmd[strlen(cmd)-1]='\0';  /* loesche '\n'*/
           insertLast(cmdSList, cmd+7);
         }
      }
   }
}
else if (strncmp(cmd+1, "do", 2) == 0) /* -- EXECUTE current hi list */
{
   int i;
   char * cmdPtr;

   for (i=1; cmdPtr=get_entrySList(cmdSList,i); i++)
   {
```

```
      printf("%s%s\n",prompt, cmdPtr);
      system(cmdPtr);
   }
}
else if (isdigit(cmd[1]))       /* -- EXECUTE #th history command */
   {
     number= atoi(cmd+1);
     system(get_entrySList(cmdSList, number));
   }
}/* end_do_history */
```

1.3. Aufgaben: Listen

Aufgabe: tminish.c, o_strlist.h, o_strlist.c

Bringen Sie das Minishell Programm zum Laufen.

+Aufgabe: Dlist

Erstellen Sie einen Modul für eine doppelt gekettete Liste

Aufgabe: sort

Fügen Sie zu obiger einfach geketteten Liste noch die Zugriffsfunktion sort(), die alle Listeneinträge sortiert.

Aufgabe: InsertSorted

Fügen Sie zu obiger einfach geketteten Liste noch die Zugriffsfunktion, um Einträge in sortierter Reihenfolge eingeben zu können.

1.4. Bäume

Unter Bäumen versteht man folgende Datenstrukturen:

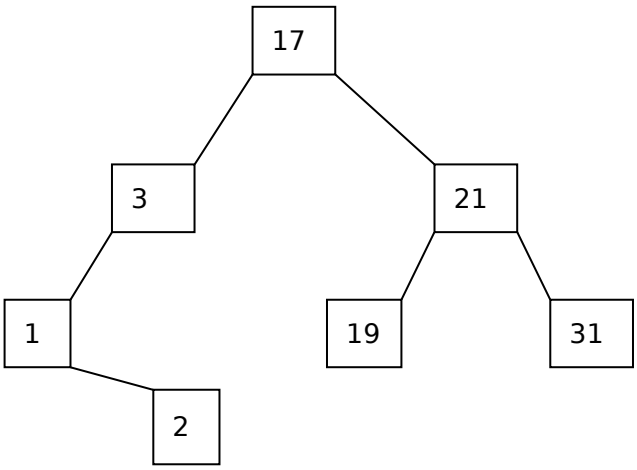
- Bäume bestehen aus **Knoten**.
- Jeder Knoten hat keine oder mehrere **Nachfolgerknoten**.
- Jeder Knoten, bis auf einen, hat einen **Vorgängerknoten**.
- Der Knoten ohne Vorgänger ist der Wurzelknoten (**root**).
- Ein Knoten ohne Nachfolger ist ein Blatt (**leaf**)

Ein Spezialfall der Bäume sind die *binären Bäume*. Hier ist die Maximalzahl der Nachfolgerknoten 2.

Ein Spezialfall der binären Bäume sind **Binäre Suchbäume**:

- Jeder Knoten besitzt einen so genannten Schlüsselwert (**key**) und es gilt:
- Jeder Knoten im **linken** Teilbaum ist **kleiner** als die Wurzel und jeder Knoten im rechten

Teilbaum ist größer als die Wurzel.



Es folgt ein Beispiel für einen Knoten eines bin. Suchbaumes:

```
typedef struct Bnode {
  int key;                //Datensatz
  struct Bnode *Left;     //Zeiger auf linken Teilbaum
  struct Bnode *Right;    //Zeiger auf rechten Teilbaum
}BNODE;
```

Wenn es sich um ein Blatt handelt sind Left und Right **NULL**-Pointer.

Wir wollen nun einen BinSearchTree-Modul entwickeln.

1.4.1. Die Haederdatei eines Binären Suchbaumes

```
// BinSearchTree.h
// A.hofmann sept. 2002

//1. Knoten definieren
typedef struct Bnode {
  int key;           //Datensatz
  struct Bnode *Left;     //Zeiger auf linken Teilbaum
  struct Bnode *Right;    //Zeiger auf rechten Teilbaum
}BNODE;


//2. interface (Zugriffsfunktionen/Methoden) definieren
typedef BNODE* BinSearchTree;

// Traversieren
void inorder (BinSearchTree root, FILE *stream);
void postorder (BinSearchTree root, FILE *stream);
void preorder (BinSearchTree root, FILE *stream);
```

```
// Suchen
BinSearchTree search (int key, BinSearchTree root);

//Einfügen
BinSearchTree insert (int key, BinSearchTree *root);

//Löschen
void delete (int key, BinSearchTree *root);
```

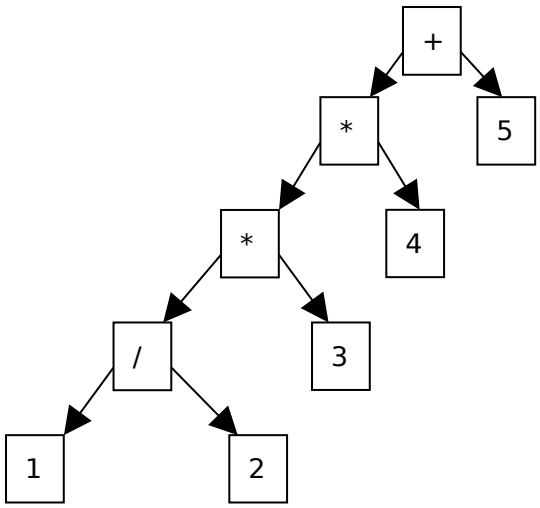
1.4.2. Das Traversieren binärer Bäume

Traversieren bedeutet jeden Knoten genau einmal aufsuchen.

Sei:
L/R Bewegung nach Links/Rechts
V einen Knoten besuchen (visit)

Dann ist:
L V R inorder
L R V postorder
V L R preorder

Beispiele:




```
void inorder ( BinSearchTree ptr, FILE *stream){

     if (ptr){
          inorder (ptr->Left, stream);

          fprintf (stream, "%d", ptr->key);

          inorder (ptr->Right, stream);
     }
```

```
}

// 1 / 2 * 3 * 4 + 5
```

Spur des Programmes inorder():

Aufruf inorder()	Wert in der Wurzel	Aktion
1	+	
2	*	
3	*	
4	/	
5	1	
6	NULL	
5	1	printf
7	NULL	
4	/	printf
8	2	
9	NULL	
8	2	printf
10	NULL	
3	*	printf
11	3	
12	NULL	
11	3	printf
13	NULL	
2	*	printf
14	4	
15	NULL	
14	4	printf
16	NULL	
1	*	printf
17	5	
18	NULL	
17	5	printf
19	NULL	

```
void postorder ( BinSearchTree ptr, FILE *stream)
{
     if (ptr){
          postorder (ptr->Left, stream);
          postorder (ptr->Right, stream);
          fprintf (stream, ”%d”, ptr->key);
     }
}
// 1  2 / 3 * 4 * 5 +
```

```
void preorder ( BinSearchTree ptr, FILE *stream)
{
     if (ptr){
          fprintf (stream, ”%d”, ptr->key);
          preorder (ptr->Left, stream);
```

```
            preorder (ptr->Right, stream);
      }
}

// + * * / 1  2  3  4  5
```

1.4.3. Suchen

```
BinSearchTree search (int key, BinSearchTree root)
{
      if (root == NULL)
            return NULL;
      else if (key == root->key)
            return root;
      else if (key < root->key)
            return search (key, root->Left);
      else
            return search (key, root->Right);
}
```

1.4.4. Einfügen

```
BinSearchTree insert (int key, BinSearchTree *root)
{
      BinSearchTree curr= *root;
      BinSearchTree prev= *root;

      // 1. Wenn Baum leer ist
      if (*root == NULL)
      {
            *root = make_node(key);
            return *root;
      }

      //2. Suche Platz zum Einfügen
      while (curr != NULL)
      {
            prev= curr;
            if (key > curr->key)
                  curr= curr->Right;
            else if ( key < curr->key)
                  curr= curr->Left;
            else //Element bereits vorhanden
                  return NULL;
```

```
     }

     //3. Bei prev einfügen
     if (key > prev->key)
     {
          prev->Right= make_node(key);
          return prev->Right;
     }
     else
     {
          prev->Left= make_node(key);
          return prev->Left;
     }
}
```

1.4.5. Das Löschen eines Elementes

wird hier nicht behandelt.

1.4.6. Aufgaben: Binäre Bäume

+Aufgabe: delBinSearch

Schreiben Sie die Funktion void delete (int key, BinSearchTree *root), die das Element (int key) aus dem Baum löscht.

Aufgabe: BinTreeADB

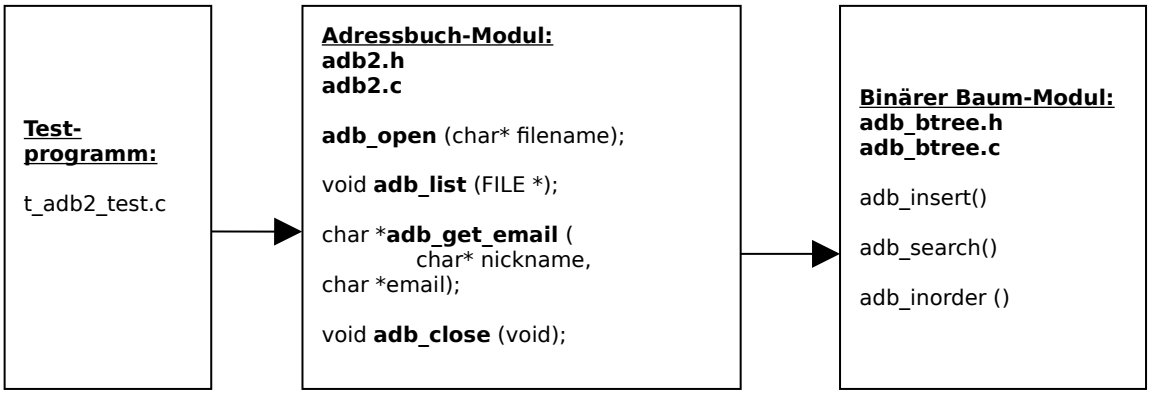
Im folgenden wollen wir einen Binären Suchbaum für das Email-Adressbuchbeispiel entwickeln. Ein Adressbuch-Datensatz (ADB_RECORD) besteht aus folgenden Komponenten:

```
typedef struct {
     char nickname[128];   // key
     char email[128];      //value1
     char comment[128];    //value2
} ADB_RECORD;              //Typ: Adressdatenbank RECORD
```

Einen Knoten für einen Binären Baum definieren wir:

```
typedef struct Bnode {
  ADB_RECORD entry;        //Datensatz
  struct Bnode *Left;//Zeiger auf linken Teilbaum
  struct Bnode *Right;     //Zeiger auf rechten Teilbaum
}BNODE;
```

Ein Vorschlag für einen Modul-Entwurf.



Schreiben Sie analog zu der in Arbeitsblatt 2 gestellten Aufgabe, den Zugriff auf die Email-Adressdatenbank der artum, dass ein Binärer Suchbaum verwendet wird.

1.5. Zusammenfassung

Wir haben in diesem Kapitel die Grundlagen der Datenstrukturen in C kennen gelernt.

1.6. Ausblick

In den folgenden Kapiteln wollen wir ähnliche Funktionalitäten bei objekt-orientierten Programmiersprachen wie C++ und Java kennen lernen.

Aufgabenverzeichnis

Aufgabe: tminish.c, o_strlist.h, o_strlist.c....13
Bringen Sie das Minishell Programm zum Laufen....13
+Aufgabe: Dlist....13
Erstellen Sie einen Modul für eine doppelt gekettete Liste....13
Aufgabe: sort....13
Fügen Sie zu obiger einfach geketteten Liste noch die Zugriffsfunktion sort(), die alle Listeneinträge sortiert....13
Aufgabe: InsertSorted....13
Fügen Sie zu obiger einfach geketteten Liste noch die Zugriffsfunktion, um Einträge in sortierter Reihenfolge eingeben zu können....13
+Aufgabe: delBinSearch....18
Schreiben Sie die Funktion void delete (int key, BinSearchTree *root), die das Element (int key) aus dem Baum löscht....18
Aufgabe: BinTreeADB....18
Im folgenden wollen wir einen Binären Suchbaum für das Email-Adressbuchbeispiel entwickeln. Ein Adressbuch-Datensatz (ADB_RECORD) besteht aus folgenden Komponenten:
....18