18. Dynamische Datenstrukturen

Dynamische Datenstrukturen ändern ihre Struktur und den von ihnen belegten Speicherplatz während der Programmausführung. Sie sind aus einzelnen Elementen, den so genannten 'Knoten', aufgebaut, zwischen denen üblicherweise eine bestimmte Nachbarschafts-Beziehung (Verweise) besteht. Die Dynamik liegt im Einfügen neuer Knoten, Entfernen vorhandener Knoten und Änderung der Nachbarschaftsbeziehung. Der Speicherplatz für einen Knoten wird erst bei Bedarf zur Programmlaufzeit allokiert. Es handelt sich hierbei um Strukturen, die als 'listen' oder 'Bäume' bezeichnet werden.

Weiterhin ist der Speicherort der einzelnen Knoten im voraus nicht bekannt (und interessiert auch nicht. Benachbarte" Knoten, d. h. Knoten, die logisch nebeneinander angeordnet sind, liegen in der Regel weit auseinander. Die Beziehung der einzelnen Knoten untereinander wird sinn voller weise über Zeiger hergestellt, die jeweils auf den Speicherort des " logischen Nachbarn" zeigen. Jeder Knoten wird daher neben den jeweils zu speichernden Nutzdaten mindestens einen Zeiger auf die jeweiligen "Nachbar"-Knoten enthalten. Man nenn solche Strukturen daher auch "verkettete Datenstrukturen" oder auch "selbstbezügliche (rekursive) Datenstrukturen". Die wichtigsten dieser Datenstrukturen sind :

- Lineare Listen
    - einfach verkette Listen
    - doppelt verkette Listen
    - Spezialformen (bezüglich der Anwendung)
    - Queue (Pufferspeicher, FIFO)
    - Stack (Kellerspeicher, LIFO)
- Bäume
    - Binärbäume (zwei Nachfolger/Nachbarn)
    - Vielweg-Bäume (mehr als zwei Nachfolger/Nachbarn)
- Allgemeine Graphen

Die wichtigsten Operationen mit dynamischen Datenstrukturen sind :

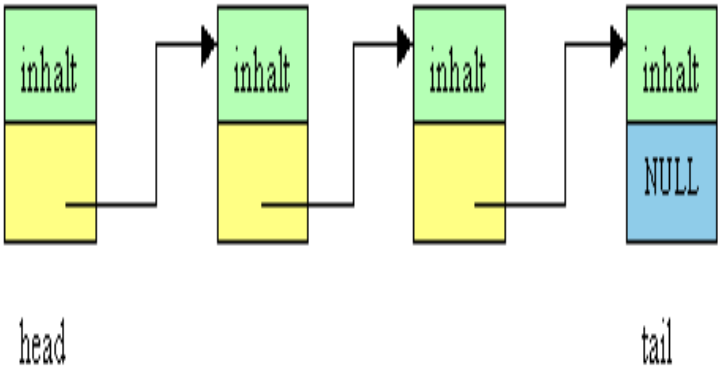
- Erzeugen eines neuen Elements
- Einfügen eines Elements
- Entfernen eines Elements
- Suchen eines Elements

In C lassen sich die einzelnen Elemente (Knoten) durch `structures` darstellen. Zur Realisierung verketteter Datenstrukturen müssen diese `structures` **Pointer** auf `structures` ihres eigenen Typs enthalten. Beispiel:

```
struct datum
  {
  int tag;
  int monat;
  int jahr;
  int jahrestag;
  char mon_name[4];
  struct datum *heute;
  };
```

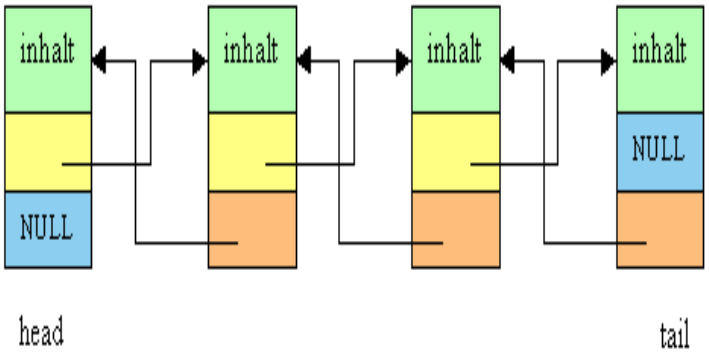
Aufbau verketteter Datenstrukturen mittels rekursiver Strukturen

Einfach verkettete Liste



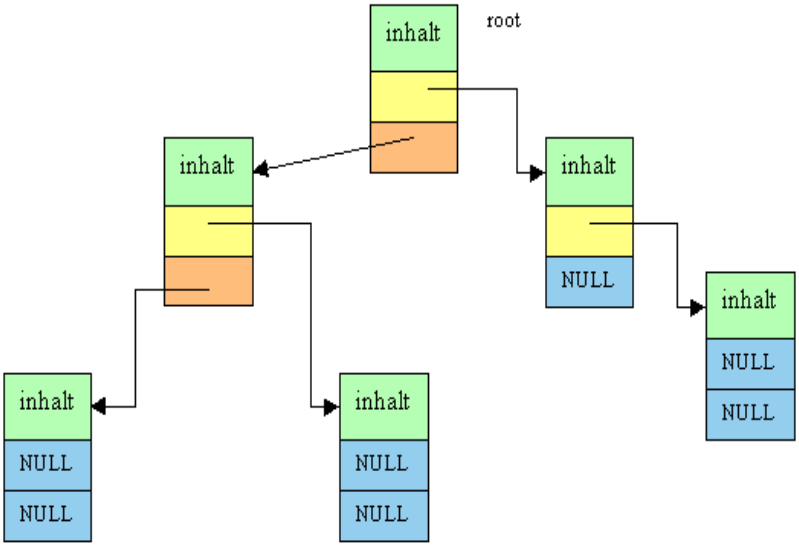

```
struct listelement
  {
  int inhalt;
  struct listelement *next;
  };
```

Doppelt verkettete Liste




```
struct listelement
  {
  int inhalt;
  struct listelement *next;
  struct listelement *back;
  };
```

Binärer Baum




```
struct baumelement
  {
  int inhalt;
  struct baumelement *rechts;
  struct baumelement *links;
  };
```

18.1. Lineare Listen

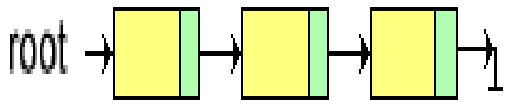
Die Operationen auf einer linearen Liste lassen sich am besten anhand eines Beispiels erarbeiten. Wir betrachten Elemente, in denen jeweils die relevanten Informationen in Form einer Zeichenkette enthalten sind. Die Listenelemente können wir uns dann anschaulich folgendermaßen vorstellen:

Information des Listenelements (hier: Zeichenkette, z. B. ein Name)	Zeiger auf nachfolgendes Element

Die Deklaration der entsprechenden Struktur sieht folgendermaßen aus:

```
struct LinListe
  {
     char inhalt[80];
     struct LinListe *next;
  };
```

Beim Aufbau der verketteten Liste können die einzelnen Listenelemente gleich nach einem bestimmten Ordnungskriterium (hier: Stringvergleich) hintereinander gehängt werden:



Das Symbol für elektrische Masse am Ende des letzten Elements deutet dabei den NULL-Zeiger an, der das Ende der Liste anzeigt.

Da jedes Element auf seinen Nachfolger zeigt, kann die ganze Liste an einem einzigen Zeiger hängen. Dieser Zeiger ist der Ursprung der Liste, oder auf englisch "root". root ist in den folgenden Beispielen eine globale Variable. Zeiger auf Listenelemente sind Zeiger auf Strukturvariable.

Wenn die Liste aufgebaut ist, so können in einer Schleife alle Elemente durchgegangen und der Inhalt ausgegeben oder geändert werden, wie das folgende Programmfragment, die Ausgabe der Liste, zeigt:

```
void DruckeListe(struct LinListe *root)
  {
     struct LinListe *Tail = root;
     while (Tail != NULL)
     {
       printf("%s\n", Tail->inhalt);
       Tail = Tail->next;
     }
   }
```

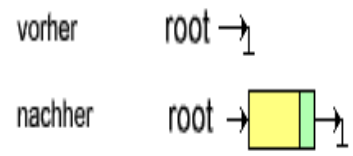
Der Zeiger Tail (Schwanz) wird benötigt, um an der Liste entlang zu gehen. Tail kann so lange auf das jeweils nächste Element gesetzt werden, bis das Listenende erreicht ist. Den NULL-Zeiger zu dereferenzieren, würde zu einem Laufzeitfehler führen.

Als nächstes wollen wir ansehen, wie ein neuer Auftrag in die Liste eingefügt wird. Wir nehmen jetzt an, daß die Liste so aufgebaut wird, daß die Elemente der lexikalischen Ordnung nach geordnet sind. Man nennt diesen Vorgang auch "Sortieren durch Einfügen".
Für ein neues Listenlelement wird zuerst eine neue Datenstruktur angelegt. Wenn kein Speicherbereich ausreichender Größe zur Verfügung steht, dann ist der Rückgabewert der NULL-Zeiger.

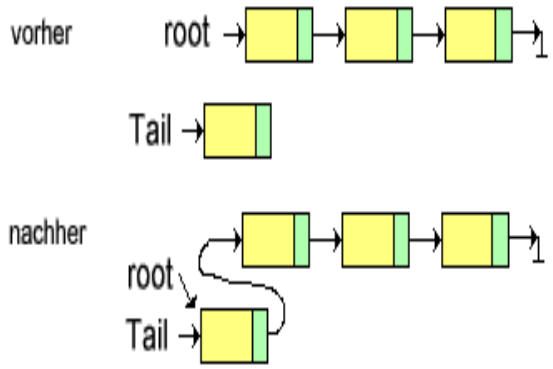
```
struct LinListe *Erzeuge(char *String)
  /* erzeugt und initialisiert neues Element in der Liste */
  {
    struct LinListe *Neu;
    Neu = (struct LinListe *) malloc(sizeof(struct LinListe));
    if (Neu == NULL)
    {
      printf("Speicher voll, Abbruch...\n");
      exit(1);
    }
    Neu->next = NULL;
    strcpy(Neu->inhalt,String);
    return(Neu);
  }
```

Um das neue Element einzufügen, wird ein weiterer Zeiger verwendet, der so lange von einem Element zum Nächsten bewegt wird, bis die Stelle gefunden ist, an der das neue Element einzufügen ist. Der Zeiger heißt im Beispiel "Tail", als Abkürzung für Rest der Liste ("Schwanz").
Ist die passende Stelle gefunden, so wird das neue Element durch Ändern von zwei Zeigern eingefügt. Dabei sind drei Fälle zu unterscheiden.

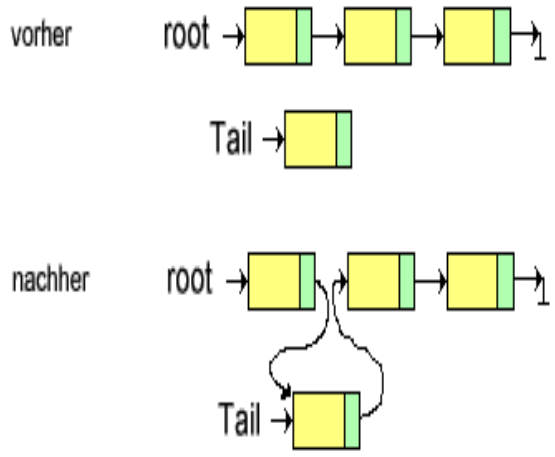
Die Liste ist noch leer:
`root` zeigt auf den Nullpointer. In diesem Fall muss `root` auf das erste neu erzeugte Element zeigen:



Es wird am Anfang der nichtleeren Liste eingefügt:
In diesem Fall ändert sich der Wert von `root`.



Einfügen innerhalb der Liste:
Hier ist es egal, ob irgendwo zwischen zwei Elementen eingefügt wird oder ob das am Listenende geschieht.



Für alle drei Einfügevarianten schreiben wir eine einzige Funktion:

```
struct LinListe *ElementEinfuegen(struct LinListe *root, struct LinListe
*Neu)
  {
  struct LinListe *Tail;
  /* Wenn Neu das erste Listenelement werden muss, */
  /*dann ist die globale Variable root zu aendern */
```

```
  if (root == NULL)
    /*Liste noch leer, erstes Element, root wird neues Element */
    root = Neu;
  else if (strcmp(Neu->inhalt, root->inhalt) < 0)
    /* Einfuegen vor dem ersten Element, root aendert sich */
    {
    Tail = root;
    root = Neu;
    Neu->next = Tail;
    }
  else
    {
    Tail = root;
    /* Element suchen, nach dem Neu einzuhaengen ist */
    while ((Tail->next != NULL) && (strcmp(Tail->next->inhalt, Neu->inhalt)
< 0))
      /*Reihenfolge beachten: erst auf NULL testen */
      Tail = Tail->next;
    /* Neu nach Tail einhaengen */
    Neu->next = Tail->next;
    Tail->next = Neu;
    }
  return(root);
  }
```

Wird kein Sortierkriterium benötigt, wird immer am Ende ein neues Listenelement angehängt. Die Funktion vereinfacht sich dann entsprechend:

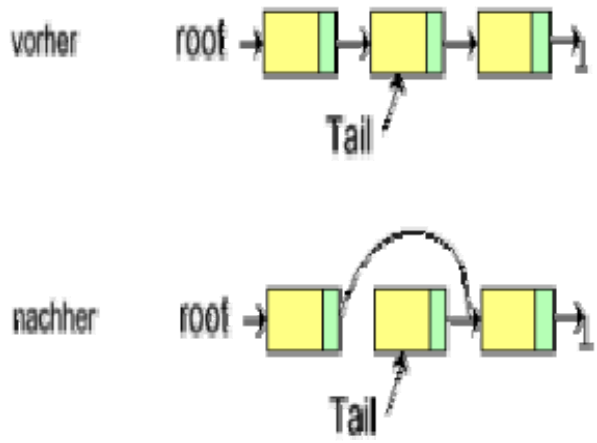
```
struct LinListe *ElementAnhaengen(struct LinListe *root, struct LinListe
*Neu)
  {
  struct LinListe *Tail;
  /* Wenn Neu das erste Listenelement werden muss, */
  /*dann ist die globale Variable root zu aendern */

  if (root == NULL)
    /*Liste noch leer, erstes Element, root wird neues Element */
    root = Neu;
  else
    {
    Tail = root;
    /* Element suchen, nach dem Neu einzuhaengen ist */
    while (Tail->next != NULL)
      Tail = Tail->next;
    /* Neu nach Tail einhaengen */
    Neu->next = Tail->next;
    Tail->next = Neu;
    }
  return(root);
  }
```

Zum Schluss noch das Testprogramm (ohne Funktionsdefinitionen):

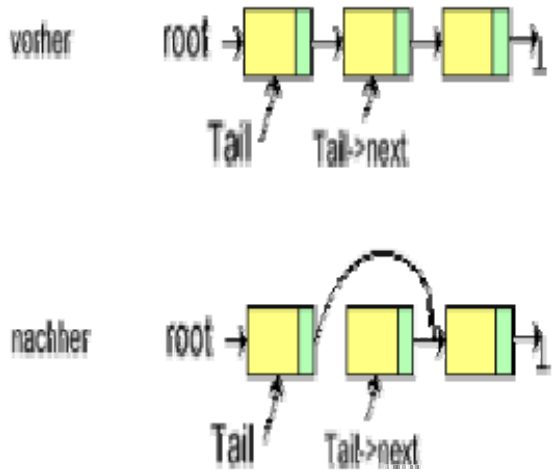
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct LinListe
  {
  char inhalt[80];
  struct LinListe *next;
```

```
  };

struct LinListe *root = NULL;

/* Hier Funktionen einfuegen, siehe oben */

int main(void)
  {
  char str[80];
  struct LinListe *Neu;

  while(1)
    {
    if (gets(str) != NULL)
      {
      Neu = Erzeuge(str);
      root = ElementEinfuegen(root, Neu);
      }
   else break;
    }

  printf("\n");
  DruckeListe(root);

  return(0);
  }
```

Bei linearen Listen ist das Löschen von Einträgen auch noch recht einfach. Es muss lediglich der Zeiger des Vorgängers auf den Nachfolger gesetzt werden.



Da jedoch bei einer einfach verketteten Liste kein Verweis auf den Vorgänger existiert (da bräuchte man doppelt verkettete Listen), wird ein Trick verwendet. Beim durch die Liste laufende Zeiger `Tail` wird nicht der Inhalt des durch den Zeiger referierten Objektes untersucht, sondern der Inhalt des Nachfolgers. Dadurch zeigt `Tail` immer auf den Vorgänger des zu löschenden Elements und das "Ausklinken" ist ganz einfach.



Da nun erst ab dem zweiten Element der Liste nach dem zu l&öuml;schenden Element gesucht wird, muß das erste Element getrennt untersucht werden. Das ist aber sowieso notwendig, da sich dann der Wert von `root` ändert. Die Funktion sieht dann so aus:

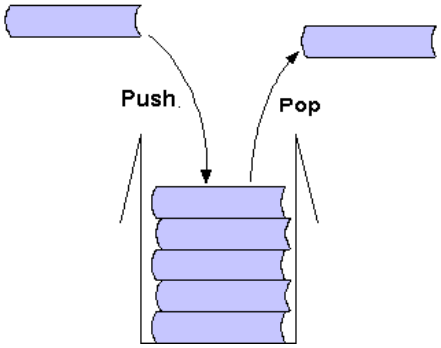
```
struct LinListe *ElementLoeschen(struct LinListe *root, char *key)
  {
  struct LinListe *Tail, *Hilf;

  if (strcmp(key, root->inhalt) == 0)
    /* Erstes Element loeschen, root aendert sich */
    {
    Hilf = root;
    root = root->next;
    free(Hilf);
    }
  else
    {
    Tail = root;
    while ((Tail->next != NULL) && (strcmp(Tail->next->inhalt, key) != 0))
      /*Reihenfolge beachten: erst auf NULL testen */
      Tail = Tail->next;
    if (Tail->next != NULL)
      {
      Hilf = Tail->next;
      Tail->next = Tail->next->next;
      free(Hilf);
      }
    else
      {
      printf("%s not found!\n",key);
      }
    }
  return(root);
  }
```

Stack, Stapel oder Keller, realisiert mit linearer Liste

Ein Keller, auch Stapel oder Stack genannt, ist eine besondere lineare Liste, bei der die Daten eine Folge bilden, die nur durch die Ein- und Ausfügeoperationen bestimmt wird. Das Prinzip eines Kellers besteht darin, dass stets das zuletzt eingefügte Element eines Kellers als erstes wieder entfernt werden muss. Die Kellerverwaltung arbeitet nach dem LIFO-Prinzip: *Last In First Out*. Was zuletzt in den Keller kam, kommt zuerst aus dem Keller auch wieder raus.

Man kann sich einen Keller als eine Bücherkiste vorstellen, in die man einzeln Bücher legen und wieder herausnehmen kann. Üblicherweise stehen folgende fünf Kelleroperationen zur Verfügung:

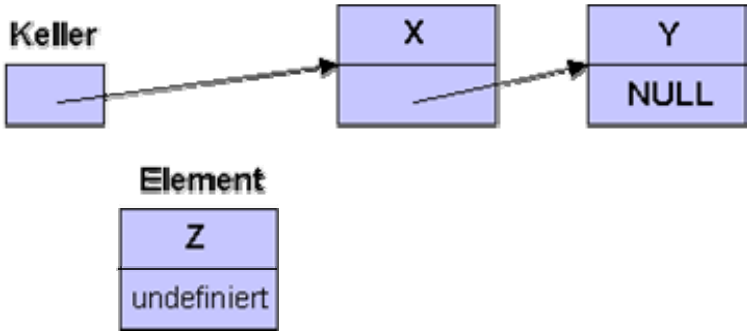
Init	Initialisiert einen neuen Keller.
Push	Legt ein Element auf dem Keller ab.
Pop	Holt ein Element aus dem Keller.
Top	Liest das oberste Element im Keller.
Empty	Prüft, ob ein Keller leer ist.



Datenstruktur für einen Keller

Man nutzt auch hier dynamisch verkettete lineare Listen. Die Elemente einer solchen Liste müssen außer der Nutzinformation noch einen Zeiger auf das nachfolgende Kellerelement speichern.

```
struct TStack
  {
  char Daten[100];
  struct TStack *Next;
  };
```



Implementierung der Kelleroperationen

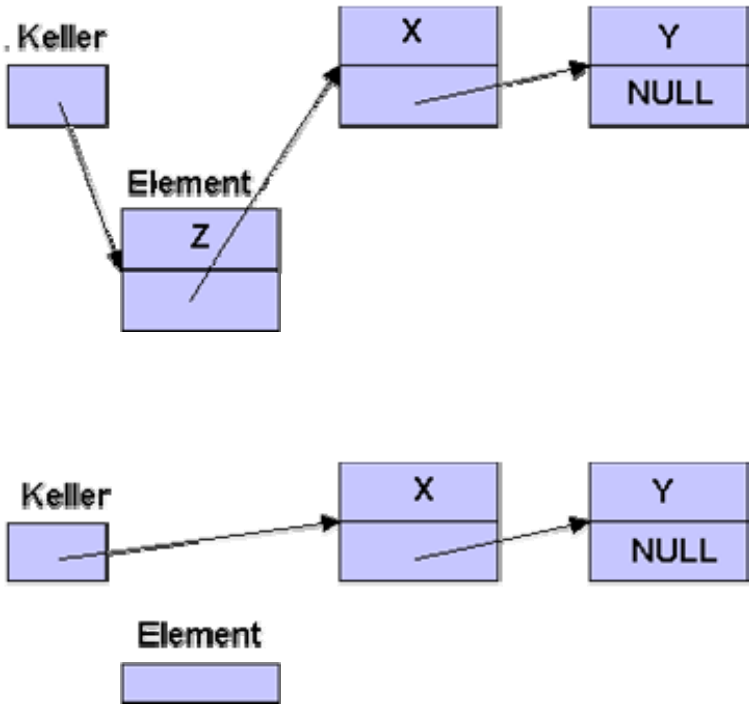
Die Implementierung der Kelleroperationen beginnen wir mit den einfachen Funktionen *Init* und *Empty.*

Init erledigt sich durch die Definition der Kellervariablen:

```
struct TStack *Keller = NULL;
```

Empty degeneriert zur Abfrage `Keller == NULL`. Interessanter sind *Push* und *Pop*:

Etwas schwieriger ist die Implementierung der *Push*-Operation. Bevor die Daten im Keller abgelegt werden können, müssen Sie in einen Element-Verbund verpackt werden. Dazu erzeugen wir mittels *New* ein neues Element und legen in ihm die Daten ab. Im Bild liegen die beiden Werte 'X' und 'Y' auf dem Keller. Das 'Z' soll als nächstes auf den Keller gelegt werden. Das neue Element muß das oberste Kellerelement werden. Dies gelingt durch zwei



Zeigeroperationen:

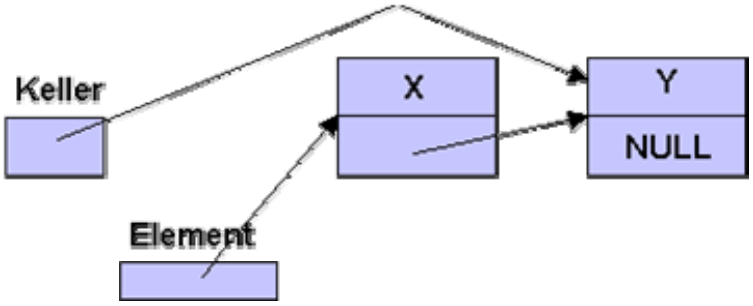
1. Durch Setzen des bislang undefinierten Zeigers auf das bislang oberste Kellerelement werden die alten Kellerelemente Nachfolger des neuen Elements.
2. Durch Umsetzen des Kellerzeigers vom alten Kelleranfang auf den neuen Kelleranfang wird das neue Element in den Keller aufgenommen.

```
void Push(char *Daten)
  {
  struct TStack *Element;

  Element = (struct TStack *) malloc(sizeof(struct TStack));
  strcpy(Element->Daten, Daten);
  Element->Next = Keller;
  Keller = Element;
  }
```

Pop muß das oberste Kellerelement vom Keller wegnehmen und dessen Daten zurückliefern. Wenn der Keller leer ist, geben wir eine Fehlermeldung aus. Bei Bedarf kann der Programmierer mit *Empty* Auskunft über den Keller erhalten. Wenn er dennoch durch falsche Programmierung eine *Pop*-Operation auf einem leeren Keller ausführt, wird er auf seinen Programmierfehler durch eine Fehlermeldung hingewiesen.

Das Abhängen des obersten Kellerelements erfolgt durch drei Zeigeroperationen:

1. Kellerelement an Hilfsvariable *Element* binden.
2. Keller-Zeiger auf nächstes Kellerelement verbiegen.
3. Speicher des alten Kellerelements freigeben.




```
void Pop(char *Resultat)
  {
  struct TStack *Element;

  if (Keller == NULL)
    printf("Der Keller ist leer!\n");
  else
    {
    strcpy(Resultat, Keller->Daten);
    Element = Keller;
    Keller = Keller->Next;
    free(Element);
    }
  }
```

Ein Testprogramm sieht dann z. B. so aus:

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

struct TStack
  {
  char Daten[100];
  struct TStack *Next;
  };

struct TStack *Keller = NULL;

void Push(char *Daten);

void Pop(char *Resultat);

int main(void)
  {
  char str[100];

  Push("Hallo");
  Push("Sie da!");
```

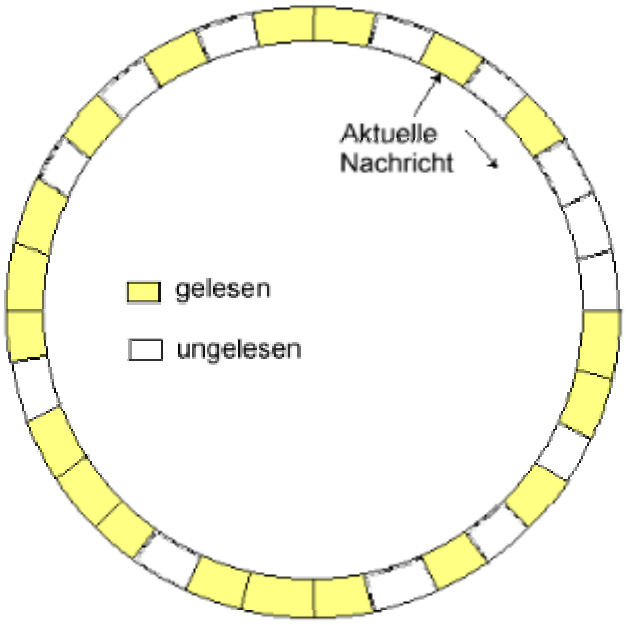
```
  Push("Ja, Sie!");

  while (Keller != NULL)
    {
    Pop(str);
    printf("%s\n",str);
    }

  return(0);
  }
```

Das Ergebnis ist dann:
```
Ja, Sie!
Sie da!
Hallo
```

Queue, realisiert mit linearer Liste

Ähnlich aufgebaut ist eine Queue, bei der die Elemente in der Reihenfolge ausgegeben werden, in der Sie eingegeben wurden, FIFO –Prinzip.

Ein Beispiel für eine Queue ist ein Ringpuffer. Ein Ringpuffer ist ein nach dem FIFO-Prinzip (first-in-first-out) organisierter Speicher zur Pufferung von Daten, die z. B. von einem Meßdatenerfassungsprogramm "angeliefert" werden, und zeitversetzt von einem Auswertungsprogramm weiterverarbeitet werden. Auch in der Betriebssystemsoftware für Netzwerkinterfaces und für die Datenfernübertragung finden Ringpuffer Verwendung.



Der Puffer muss groß genug sein, damit er auch bei maximalem Zeitversatz nicht "überläuft". Durch Realisierung als lineare Liste ist dieser Fall immer gegeben, da man zu der Liste immer neue Elemente hinzufügen kann (jedenfalls, solange noch Arbeitsspeicher vorhanden ist).

Es muss dafür gesorgt werden, dass aus einem leeren Puffer nicht mehr gelesen wird -> Meldung "Puffer leer".

Die Abbildung des Rings in einem linear adressierten Speicher geschieht dadurch, dass ein am Pufferende angekommener Zeiger wieder an den Anfang gesetzt wird. Das Letze Element der Lister verweist also wieder auf das erste.

Der Ringpuffer soll für die Verwaltung einer ständig wechselnden Anzahl von Nachrichten verwendet werden, wie es z. B. bei E-Mails der Fall ist. Die Nachrichten treffen als Texte unterschiedlicher Länge ein. Um sie effektiv zu verwalten und zu speichern, liegt die Verwendung von *verketteten Listen* nahe. Um aus diesen Listen einen Ringpuffer zu machen, muss folgendermaßen vorgegangen werden:

- Eine neue Nachricht hängt sich an die vorherige "dran" - die Kette wird länger.
- Die Kette soll immer zu einem Ring geschlossen sein, damit z. B. nach dem Lesen der neuesten Nachricht als nächstes eine Wiederholung alter Nachrichten bei der ältesten beginnen kann.
- Zu einer effektiven Speicherverwaltung gehört auch das Beschaffen von Speicherplatz erst dann wenn er benötigt wird (`malloc()`) und Freigeben beim Löschen von Einträgen (`free()`).

Es wird bei der ältesten Nachricht begonnen. Im Verlauf des Programms wird die jeweils aktuelle Nachricht angezeigt. Zur Interaktion mit dem Benutzer bietet das Hauptprogramm folgendes Menü an:

- "D": Nachricht löschen (Delete)
- " ": (Leertaste) Nächste Nachricht
- "Q": Programm verlassen (Quit)
- "X": alle Nachrichten löschen

Nur zum Zweck der **Simulation** des Nachrichtenempfangs wird noch ein weiterer Menüpunkt benötigt:

- "E": Nachricht eingeben. Nun kann eine Textzeile eingegeben werden. Das neue Listenelement wird an der aktuellen Listenposition eingefügt.

Der **Pufferzustand** soll ständig erkennbar sein, d. h. in einer sog. Statuszeile wird die Zahl alter/neuer (= gelesener/ungelesener) Nachrichten angezeigt.

Baumstrukturen

Baumstrukturen spielen in der Datenorganisation eine wichtige Rolle. Auch im Alltag sind diese Strukturen allgegenwärtig:

- beim Familienstammbaum;
- als Organisationsstruktur eines Unternehmens
- bei der Einteilung von Texten in Kapitel, Abschnitte, Absätze, ...

Ein Baum ist folgendermaßen definiert:

- Baumstrukturen haben Verzweigungen, die vom Ursprung (Anfangselement) und von Verzweigungen (Teilbäumen) selbst ausgehen können, bis schließlich Endpunkte erreicht werden.
- Die Elemente des Baums heißen Knoten, wobei das Anfangselement Wurzel heißt.

- Die Endpunkte heißen Blätter.
- Die Wurzel ist der einzige Knoten der keinen Vorgänger hat.
- Knoten, die weder Wurzel noch Blatt sind, heißen innere Knoten.
- Die Verweislinie zwischen zwei Knoten wird Kante genannt.
- Die Folge der Kanten von der Wurzel bis zu einem beliebigen Knoten heißt Pfad.
- Die Weglänge wird bestimmt durch die Anzahl der Kanten.
- Knoten, die dieselbe Weglänge von der Wurzel entfernt liegen, bilden jeweils eine Ebene.
- Die Gesamtzahl der Ebenen gibt die Tiefe des Baumes an. Anders ausgedrückt ist dies die Anzahl der Knoten an dem längsten Pfad.
- Falls ein Baum keinen Knoten besitzt, wird er leerer Baum genannt.
- Die maximale Anzahl der direkten Nachfolger, die ein Knoten hat, heiß Grad des Baumes.

Ein Baum vom Grad 2 ist der binäre Baum. Er stellt einen wichtigen Sonderfall dar. Gegenüber dem allgemeinen Baum zeichnet er sich folgendermaßen aus:

- Jedes Element außer dem Anfangselement hat genau einen direkten Vorgänger.
- Jedes Element hat null bis zwei direkte Nachfolger, die als linker bzw. rechter Nachfolger unterschieden werden.
- Der Binärbaum besteht also aus einer Wurzel und zwei Teilbämen, wobei ein Baum und damit auch ein Teilbaum leer sein kann.
- Eine Sonderform ist der vollständige Binärbaum, bei dem jeder Knoten außer den Blättern zwei nicht leere Teilbäume besitzt und jeder Pfad die gleiche Länge hat.

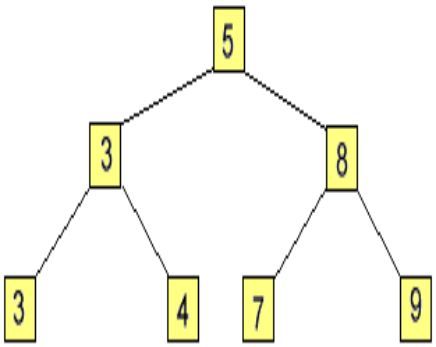
Offensichtlich kann man in einem binären Baum der Tiefe n höchstens 2^n-1 Knoten unterbringen. Bei einem unvollständigen Baum sind es entsprechend weniger. Im ungünstigsten Fall lassen sich nur n Elemente unterbringen, genau dann, wenn jeder Knoten nur einen Nachfolger besitzt. In diesem Fall degeneriert der Baum zu einer linearen Liste.

Die Baumstruktur ist rekursiver Natur, da die Nachfolger eines Knotens jeweils wieder als Wurzeln von Bäumen aufgefasst werden können.

Wenn die Knoteninhalte untereinander keine Ordnung besitzen ist der Baum ungeordnet. Bei geordneten Bäumen ist eine vollständige Ordnungsrelation zwischen den Knoten Voraussetzung. Sie wird oft über einen Schlüssel realisiert. Beispiele für Ordnungsschlüssel sind:

- Enthalten die Knoten nur natürliche Zahlen, so ist die numerische Ordnung eine derartige Relation.
- Falls die Knoten Zeichenketten enthalten, so ist die lexikographische Ordnung geeignet.

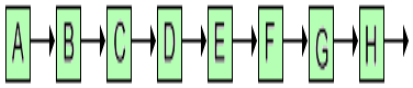
Lege nun "<" eine vollständige Ordnungsrelation zwischen den Knoten eines Binärbaums fest, dann ist jeder Knoten des Baums kleiner als alle Knoten seines rechten Teilbaums und zugleich größer oder gleich als alle Knoten seines linken Teilbaums.



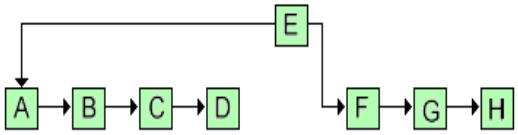
Offensichtlich gilt für jeden Knoten, daß ein kleinerer Nochfolgeknoten links und ein größerer rechts unterhalb anzuordnen ist. Nach der verwendeten Ordnungsrelation ist im Fall der Gleichheit ein Nachfolgeknoten ebenfalls links unterhalb anzuordnen.

Der Suchbaum

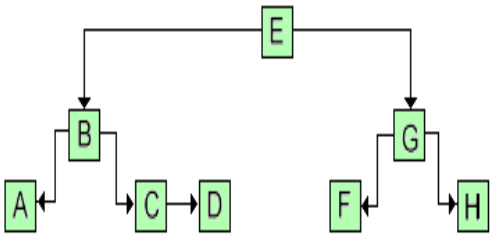
Der geordnete Baum erlaubt eine schnelle Suche von Informationen, ähnlich wie das binäre Suchverfahren in linearen Strukturen. Der geordnete Baum heißt deshalb auch Suchbaum. Dazu ein Beispiel. Gegeben sei folgende lineare Liste:



Der direkte Zugriff auf die Elemente der Liste sei möglich. Bei der binären Suche wird auf das mittlere Element zugegriffen. Das Anfangselement ist somit nicht mehr A sondern E, das die Liste in zwei Teile teilt:



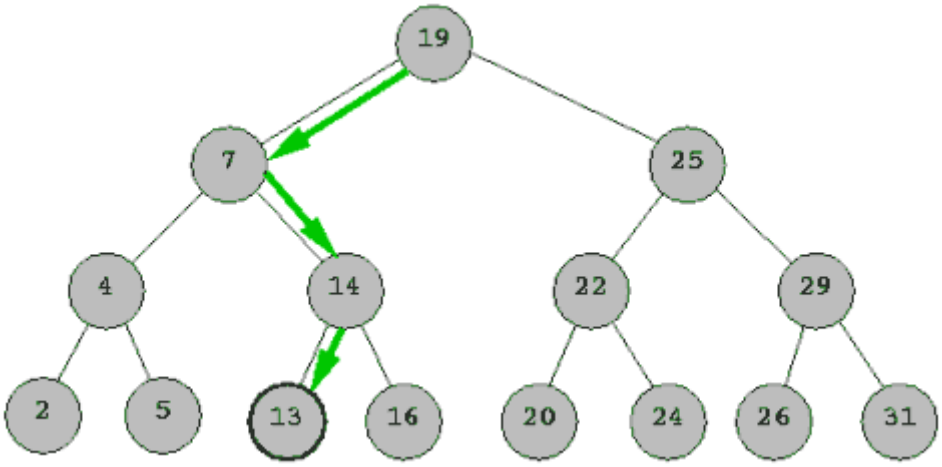
Ist E nicht das gesuchte Element, wird in der linken oder rechten Teilliste weitergesucht.



Teilt man die Liste solange nach diesem Verfahren, bis diese nur noch aus je einem Element bestehen, so liegt ein geordneter Baum in der bekannten Darstellung vor.

Dieser Baum ist so geordnet, dass alle Elemente im rechten Teilbaum einen lexikalisch größeren Wert besitzen als die Wurzel, und alle Elemente im linken Teilbaum einen kleineren. Dies gilt auch für alle Teilbäume. Einen derart georneten Baum bezeichnet man als binären Suchbaum. Damit wurde die linear verkettete Struktur, auf der im Prinzip nur sequentiell gesucht werden kann, durch Umorganisation in eine binäre Baumstruktur überführt, auf der der Suchvorgang aufgrund der Ordnungsrelation binär verläuft.

Auch bei der Datenstruktur Baum besteht die Notwendigkeit, Elementinhalte dauerhaft zu speichern und bei Bedarf zu laden. Dies führt zu dem grundsätzlichen Problem, die binär verzweigte Struktur so in eine lineare Struktur zu überführen, dass die Daten so als Datei gespeichert werden können, dass später daraus der ursprüngliche Binärbaum wieder rekonstruiert werden kann. Gegeben sei folgender Binärbaum:



- Inorder Traversierung
  Die Inorder-Traversierung hat stets die Reihenfolge *linker Teilbaum - Wurzel - rechter Teilbaum*. Die Wurzel steht in der Mitte. Der Baumdurchlauf nach dem Inorder-Algorithmus hat die besondere Eigenschaft, die nach einem bestimmten Kriterium in den Baum eingefügten Informationen der Knoten in ihrer sortierten Reihenfolge auszugeben.

  2 4 5 7 13 14 16 19 20 22 24 25 26 29 31

```
inorder (baumtyp *baum)
  {
  if (baum != NULL)
    {
    inorder (linker_teilbaum);
    verarbeite Wurzelinformation von baum;
    inorder (rechter_teilbaum);
    }
  }
```

  Anwendungsbeispiel:
  Dies ist die typische Traversierungs-Strategie, um eine vollständig sortierte Liste aus

einem Suchbaum zu bekommen: Besuche den linken Teilbaum (in dem alle Elemente kleiner als das Wurzelelement sind), dann die Wurzel, dann den rechten Teilbaum (in dem alle Elemente größer als die Wurzel sind).

- Postorder Traversierung
  Ändert man den Traversierungsalgorithmus dahingehend ab, dass der Baum in der Reihenfolge *linker Teilbaum - rechter Teilbaum - Wurzel* durchlaufen wird, spricht man von Postorder-Traversierung.

  2 5 4 13 16 14 7 20 24 22 26 31 29 25 19

  Dieses Ergebnis erzielt man, wenn man von der Wurzel ausgehend links herum alle Knoten längs der Kanten besucht und jedesmal, wenn man rechts neben einem Knoten vorbeikommt, den Knoteninhalt aufschreibt.

```
postorder (baumtyp *baum)
  {
  if (baum nicht leer)
    {
    postorder (linker_teilbaum);
    postorder (rechter_teilbaum);
    verarbeite Wurzelinformation von baum;
    }
  }
```

  Anwendungsbeispiel:
  Bei einem Taschenrechner oder anderen einfachen Automaten müssen zunächst intern die Argumente einer Rechenoperation in Register geladen werden. Danach wird die verknüpfende Operation aufgerufen (umgekehrt polnische Notation). POST-Order liefert dazu die passende Besuchs-Reihenfolge: *(Argument1, Argument2) Operation*.

- Preorder Traversierung
  Eine dritte Möglichkeit der Traversierung besteht in der Reihenfolge *Wurzel - linker Teilbaum - rechter Teilbaum*:

  19 7 4 2 5 14 13 16 25 22 20 24 29 26 31

  Dieses Ergebnis erzielt man, wenn man von der Wurzel ausgehend links herum alle Knoten längs der Kanten besucht und jedesmal, wenn man links neben einem Knoten vorbeikommt, den Knoteninhalt aufschreibt.

```
preorder (baumtyp *baum)
  {
  if (baum nicht leer)
    {
    verarbeite Wurzelinformation von baum;
    preorder (linker_teilbaum);
    preorder (rechter_teilbaum);
    }
  }
```

Anwendungsbeispiel:
Im Zusammenhang mit Funktionen kann in der Wurzel der Funktionsaufruf, in den Teilbäumen die Argumente stehen. PRE-Order ergibt dann die "gewohnte" Schreibweise *Funktion (Argument1, Argument2)*