Chair of Embedded Systems
Department of Computer Science
University of Augsburg, Germany

Martin Frieb

Universität Augsburg University

Wintersemester 2020/2021

Informatik 1

Kapitel 15 – Dynamische Datenstrukturen

Inhaltsverzeichnis

15.1 Überblick

Was ist eine dynamische Datenstruktur?
Bis zum Kapitel 8 haben wir nur mit statischen Datenstrukturen gearbeitet: Wir haben z.B. am Anfang von Funktionen Variablen so deklariert, dass wir anschließend mit einem `int`, `double`, `char` oder Feld davon arbeiten können. Ansonsten mussten wir uns um nichts kümmern. Der Compiler hat automatisch Speicherplatz reserviert und wieder freigegeben und wir konnten genau so viel Speicherplatz nutzen, wie wir bei der Programmerstellung festgelegt hatten.

> Beispiel: Anlegen einer statischen Datenstruktur
>
> `char v[SIZE];` reserviert Platz für genau `SIZE` Zeichen inklusive der abschließenden `\0`. Der Compiler kümmert sich darum, dass wir das Feld `v` nutzen können.

Dies ist allerdings auch mit Nachteilen verbunden: Wenn wir `SIZE` zu klein wählen und gerne mehr Platz nutzen wollen würden, können wir `v` **nicht** nachträglich vergrößern. Wenn wir `SIZE` zu groß wählen, verschwenden wir unnötig Speicherplatz. Außerdem verbleibt die Frage wann `SIZE` groß genug ist.

Abhilfe schafft die dynamische Speicherverwaltung, wie wir sie im Kapitel 9 kennengelernt haben. Mit Hilfe von `malloc` können wir zur Laufzeit dynamisch Speicherplatz auf dem Heap reservieren.

> Beispiel: Anlegen einer dynamischen Datenstruktur
>
> `char *v = malloc(SIZE * sizeof(char));` reserviert Speicherplatz für `SIZE` Zeichen inklusive der abschließenden `\0`. Wir müssen den Speicherbereich des Feldes `v` mit `free` freigeben, wenn wir ihn nicht mehr brauchen, können aber auch seine Größe jederzeit via `realloc` ändern.

Die Speicherverwaltung übernimmt in diesem Fall **nicht** der Compiler, sondern wir kümmern uns selbst darum, Speicher zu reservieren, zu vergrößern oder zu verkleinern und ihn wieder freizugeben. Damit können wir den reservierten Speicherplatz immer entsprechend unserem aktuellen Bedarf anpassen.

In diesem Kapitel kombinieren wir dynamische Speicherverwaltung mit komplexen Datenstrukturen zu **dynamischen Datenstrukturen**. Diese sind in bestimmten Bereichen der Informatik von enormer Bedeutung und tauchen in verschiedensten Anwendungsbereichen auf. Beispielsweise werden *Stacks* (Kapitel 15.4) und *Queues* (Kapitel 15.5) für viele Algorithmen verwendet, um bestimmte Operationen möglichst effizient gestalten zu können. *Verkettete Listen* (Kapitel 15.3) können z.B. für hochgradig parallele Datenverarbeitung verwendet werden, wobei Wartezeiten minimiert werden.

Wir konzentrieren uns in diesem Kapitel auf einige ausgewählte, grundlegende Datenstrukturen. Ausführlichere und komplexere Datenstrukturen werden in der Vorlesung *Informatik 3* vorgestellt.

Anwendung dynamischer Datenstrukturen
Dynamische Datenstrukturen werden zur effizienten Verwaltung von sortierten und unsortierten Listen (und Mengen) von Datenwerten, die sich zur Laufzeit einer Anwendung häufig ändern, verwendet.

Typische Verwaltungsoperationen für solche Datenstrukturen sind:

- Eine leere Liste anlegen (`list_create`).
- Eine Liste löschen (`list_destroy`).
- Länge einer Liste berechnen (`list_size`).
- Eine Liste ausgeben (`list_print`).
- Testen, ob eine Liste leer ist (`list_isempty`).
- Einer Liste ein neues Element hinzufügen (`list_insert`).
- Ein vorhandenes Element aus einer Liste löschen `list_delete`).
- Testen, ob eine Liste ein Element enthält (`list_iselem`).

Im Folgenden werden verschiedene dynamische komplexe Datenstrukturen zur Verwaltung von Listen ganzer Zahlen vorgestellt. Diese lassen sich natürlich auch für andere Datentypen anpassen.

Hinweis zur Initialisierung der Datenstrukturen
Die dynamischen Datenstrukturen verwenden i.d.R. Zeiger. Dabei wird häufig unterschieden, ob bereits Elemente in der dynamischen Datenstruktur gespeichert sind oder ob diese noch leer ist. Diese Unterscheidung wird durch eine Überprüfung auf den Wert `NULL` vorgenommen. Achten Sie daher darauf, dass Sie Zeigern, die nirgendwohin zeigen, immer die Konstante `NULL` zuweisen.

15.2 Dynamische Felder

15.2.1 Grundlagen dynamischer Felder

Was ist ein dynamisches Feld?
Ein **dynamisches Feld** ist im Grunde ein Feld, dessen Größe erst zur Programmlaufzeit festgelegt wird. Im Kapitel 9.6 haben wir bereits mit Hilfe von `malloc` Felder von variabler Größe angelegt bzw. mit `realloc` deren Größe verändert. Nun fügen wir noch einen Zähler hinzu, der speichert, wie groß das Feld aktuell ist und fassen beides als komplexe Datenstruktur zusammen. Zusätzlich stellen wir Verwaltungsoperationen zur Verfügung, die den Zugriff vereinheitlichen und vereinfachen. Damit erhalten wir ein *dynamisches Feld*, in dem wir Elemente nicht nur einfach einfügen und löschen, sondern z.B. auch suchen können. Damit unser dynamisches Feld nicht zu kompliziert wird, speichert es die `int`-Werte in diesem Kapitel unsortiert.

> Beispiel:
>
> Beispielsweise wollen wir in einer Liste alle Namen der Studierenden speichern. Würden wir das mit einem statischen Feld realisieren, würde das so aussehen:

```
char namen[100][20];
```

Wir haben nun eine Liste, die 100 Namen speichern kann. Jeder Name kann dabei maximal 20 Buchstaben lang sein.
Doch was, wenn ein Name länger als 20 Buchstaben ist? Oder unsere Universität so populär wird, dass sie von mehr als 100 Studierenden besucht wird?

In diesem Fall ist eine dynamische Liste deutlich überlegen. Wir können zur Laufzeit unsere Feldgröße beliebig vergrößern (oder verkleinern), sodass auch ein Schwung an 400 neuen Studierenden mit sehr langem Namen uns nicht ins Schwitzen bringt.

Ein dynamisches Feld und seine Verwaltungsoperationen könnte folgendermaßen aussehen:

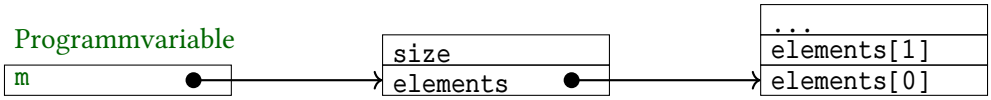
```
1  #ifndef ARRAYLIST_H_INCLUDED
2  #define ARRAYLIST_H_INCLUDED
3
4  typedef struct _arraylist {
5      int *elements;
6      int size;
7  } arraylist;
8
9  arraylist *arraylist_create(void);
10 void arraylist_destroy(arraylist *m);
11 int arraylist_size(const arraylist *m);
12 void arraylist_print(const arraylist *m);
13 int arraylist_isempty(const arraylist *m);
14 int arraylist_insert(arraylist *m, int n);
15 int arraylist_delete(arraylist *m, int n);
16 int arraylist_iselem(const arraylist *m, int n);
17
18 #endif
```

Das dynamische Feld selbst ist durch folgende komplexe Datenstruktur dargestellt:

```
1  typedef struct _arraylist {
2      int *elements;
3      int size;
4  } arraylist;
```

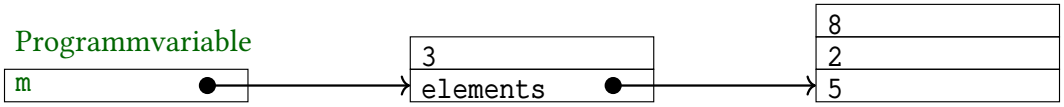
Das dynamische Feld selbst wird durch ein `struct` realisiert, das zwei Komponenten hat: Zum Einen einen Zeiger `elements` auf einen via `malloc` reservierten Speicherbereich., in dem die Daten als Feld abgelegt werden (in diesem Fall `int`-Werte). Die Zweite Komponente `int size` gibt die Länge dieses Feldes an, also die aktuelle Anzahl an gespeicherten Zahlen. Dieses `size` ist gleichzeitig auch die Datenstrukturinvariante. Es muss immer die aktuelle Größe des Feldes darin gespeichert sein.

Dynamische Felder im Arbeitsspeicher
Im Arbeitsspeicher würde eine Variable `arraylist m` folgendermaßen aussehen:



Beispiel:

Darstellung der Liste `5,2,8`:

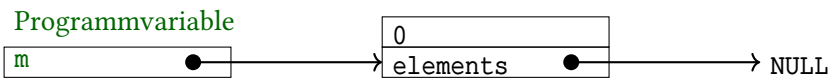


15.2.2 Anlegen und Löschen dynamischer Felder

Verwaltungsoperation: Leere Liste anlegen
Zuerst muss immer die Datenstruktur erzeugt bzw. deklariert werden. Dafür wird eine leere Liste ohne Elemente darin erzeugt (man beachte die Einhaltung der Datenstrukturinvariante). Dies wird durch folgende Funktion realisiert:

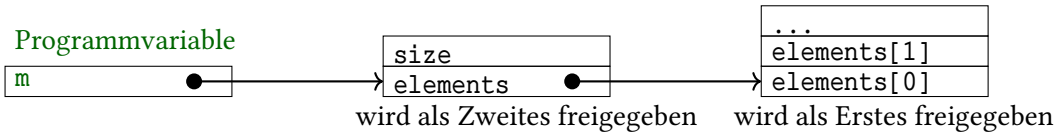
```
1 arraylist *arraylist_create(void)
2 {
3     arraylist *m = malloc(sizeof(arraylist));
4     if(m == NULL)
5         return NULL;
6     m->elements = NULL;
7     m->size = 0;
8     return m;
9 }
```

Wir reservieren Speicherplatz für die Verwaltungsdatenstruktur eine neuen dynamischen Feldes im Heap. Da wir damit noch keine Speicherplatz für die einzelnen Elemente reserviert haben, setzen wir den `elements`-Zeiger auf `NULL` und die Größe des dynamischen Feldes auf 0. Wir nutzen einen Zeiger, daher können wir nicht mit dem Punkt-Operator arbeiten, sondern müssen den Pfeil-Operator verwenden. Anschließend können wir das dynamische Feld zurück geben:



Verwaltungsoperation: Dynamisches Feld freigeben
Möchten wir das dynamische Feld freigeben (d.h. die Elemente und die obige Verwaltungs-Datenstruktur löschen), müssen wir das von hinten nach vorne tun. Das bedeutet: Zuerst die Elemente des dynamischen Feldes löschen, bevor wir die zugehörige komplexe Datenstruktur (`struct`) selbst löschen können.

```
1  void arraylist_destroy(arraylist *m)
2  {
3      if(!arraylist_isempty(m))
4          free(m->elements);
5      free(m);
6  }
```



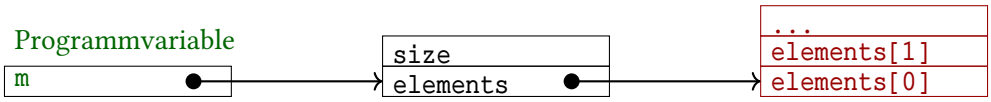
Würden wir zuerst die komplexe Datenstruktur löschen, also zuerst `free(m)` aufrufen, würde dadurch ein *Speicherleck* entstehen. Wir löschen dadurch `size` und den Zeiger `elements`, doch alle Werte in dem Feld, auf das `elements` zeigt, würden noch im Speicher existieren. Durch das Löschen von `elements` verlieren wir den Zeiger auf das eigentliche Feld und können so nicht mehr darauf zugreifen. Und ohne einen Zugriff auf die Werte können wir sie auch nicht löschen.

15.2.3 Verwaltungsoperationen zum Zugriff auf dynamische Felder

Verwaltungsoperation: Test auf Elementzugehörigkeit
Test auf Element-Zugehörigkeit mit sequentieller Suche:

```
1  int arraylist_iselem(const arraylist *m, int n)
2  {
3      int k;
4      for (k = 0; k < m->size; ++k) {
5          if ((m->elements)[k] == n)
6              return 1;
7      }
8      return 0;
9  }
```

Um zu testen, ob ein Wert bereits in der Liste vorhanden ist, muss die Liste solange durchgegangen werden, bis entweder das Ende erreicht ist oder eine entsprechende Variable gefunden wurde (da unser dynamisches Feld hier nicht sortiert ist). Das wird durch die Schleife erreicht. Mit `(m->elements)[k]` wird auf die `k`-te Feldkomponente von `elements` zugegriffen und mit dem gesuchten Wert verglichen. Da wir das dynamische Feld hierfür einmal durchlaufen müssen, ist die Zeitkomplexität linear zur Länge der Liste `k`, d.h. $O(k)$.



Verwaltungsoperation: Element einfügen
Neues Element an letzter Stelle einfügen:

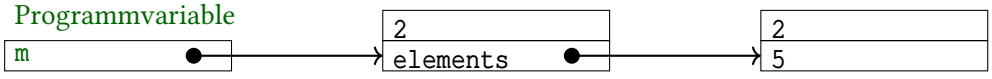
```
1  int arraylist_insert(arraylist *m, int n)
2  {
3      int *neu = NULL;
4      if (arraylist_isempty(m))
5          neu = malloc(sizeof(int));
6      else
7          neu = realloc(m->elements, (m->size + 1) * sizeof(int))
               ;
8      if(!neu)
9          return 0;
10     m->elements = neu;
11     (m->elements)[m->size] = n;
12     ++(m->size);
13     return 1;
14 }
```

Beim Einfügen eines Wertes werden zwei Fälle unterschieden:

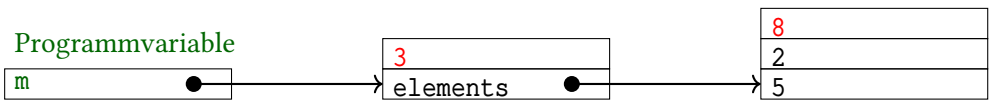
1. Die Liste ist leer. In diesem Fall wird neuer Speicherplatz via `malloc` reserviert (Zeile 5).
2. In der Liste sind bereits Elemente enthalten. Dann wird der reservierte Speicherplatz mit Hilfe von `realloc` vergrößert (Zeile 7).

Danach wird überprüft, ob die (zusätzliche) Speicherreservierung erfolgreich war (Zeile 8). Wenn nicht, bricht die Funktion mit einem Fehler ab (Zeile 9). Ansonsten kann der `elements`-Zeiger auf den neu reservierten Speicherbereich umgebogen werden (Zeile 10). Schließlich wird der neue Wert im neu reservierten Speicherbereich gespeichert (Zeile 11) und die Größe des dynamischen Feldes angepasst (Zeile 12). Das Einfügen in das dynamische Feld dauert unabhängig von dessen Länge immer gleich lange. Die Zeitkomplexität ist somit konstant, d.h. $O(1)$.

Beispiel:



Neuen Wert 8 einfügen (Aufruf: `status = arraylist_insert(m, 8)`):



Verwaltungsoperation: Element löschen

Möchte man ein Objekt löschen, dann muss die komplette Liste nach diesem Objekt durchsucht werden. Wenn das Element nicht gefunden wird, bricht man ab. Ansonsten kann das gefundene Element gelöscht werden, indem alle Elemente um eine Position nach vorne verschoben werden.

Eingabe: $x_1, \ldots, x_n \in \mathbb{Z}, x \in \mathbb{Z}, n \in \mathbb{N}$

1 Suche erstes k mit $x_k = x$;

2 **wenn** x *kommt nicht in* $x_1, \ldots, x_n$ *vor* **dann**

  Ausgabe: 1

3 Bewege alle Elemente nach x_k um eine Position nach vorne;

Ausgabe: 0

Bei der Implementierung in C müssen wir nach dem nach vorne schieben noch die Größe des reservierten Speicherbereichs verkleinern. Sollte dies fehlschlagen, muss der Zustand wiederhergestellt werden, der vor dem Löschen des Elements bestanden hat.

```
1  int arraylist_delete(arraylist *m, int n)
2  {
3      int *neu = NULL;
4      int k = 0, i;
5      /* n in arraylist suchen */
6      while (k < m->size) {
7          if ((m->elements)[k] == n) break;
8          ++k;
9      }
10     /* wenn n nicht in arraylist gefunden wurde */
11     if (k == m->size) return 1;
12     /* n aus arraylist entfernen */
13     /* alle weiteren Elements um 1 nach vorne schieben */
14     for (i = k; i < m->size - 1; ++i)
15         (m->elements)[i] = (m->elements)[i + 1];
16     /* reservierten Speicherbereich verkleinern */
17     neu = realloc(m->elements, (m->size - 1) * sizeof(int));
18     /* realloc fehlgeschlagen => bisherigen Zustand
          wiederherstellen */
19     if (neu == NULL && m->size != 1) {
20         for (i = m->size - 2; i > k; --i)
21             (m->elements)[i] = (m->elements)[i - 1];
22         (m->elements)[k] = n;
23         return 2;
24     }
25     /* realloc erfolgreich => Zeiger + Groesse anpassen */
26     m->elements = neu;
27     --(m->size);
```

```
28      return 0;
29  }
```

Das Löschen eines Elements läuft wie folgt ab:

1. Das erste Vorkommen der zu löschenden Zahl wird gesucht (erstes `k` mit `(m->elements)[k] == n`, `while`-Schleife in Zeilen 6-9).
   Wenn `n` nicht in `(m->elements)[1],...,(m->elements)[n]` vorkommt, gibt die Funktion in Zeile 11 den Wert 1 zurück und bricht ab.

2. Um `n` aus dem dynamischen Feld zu entfernen, wird es mit der Zahl überschrieben, die an der nachfolgenden Position steht (sonst würden wir eine Lücke erhalten). Ebenso werden alle weiteren Zahlen um eine Position nach vorne verschoben, damit es nirgendwo eine Lücke gibt (Zeilen 14-15).

3. Wenn alle nachfolgenden Zahlen um eine Position nach vorne verschoben wurden, wird die letzte Position nicht mehr benötigt. Das dynamische Feld wird daher in Zeile 17 um einen Eintrag verkleinert. Sollte es sich um das letzte Element handeln, wird der Speicher des dynamischen Feldes freigegeben. Wenn `realloc` fehlschlägt, wird als nächstes Schritt 4. ausgeführt, sonst Schritt 5.

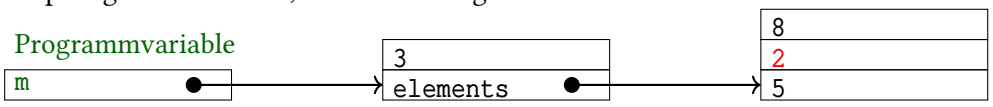
4. Zunächst wird im `if` unterschieden: Wenn `m->size` den Wert 1 hat, so wurde gerade das letzte Element gelöscht und daher das dynamische Feld freigegeben. In diesem Fall hat `realloc` ordnungsgemäß funktioniert und es geht weiter bei 5.
   In allen anderen Fällen ist `realloc` fehlgeschlagen und der ursprüngliche Zustand wird wiederhergestellt: Alle Elemente ab dem gelöschten Element werden in der Schleife in den Zeilen 20-21 wieder um eine Position nach hinten verschoben. Da das Element an der Position `m->size - 1` nicht verändert wurde (es wurde nur an die Position `m->size - 2` kopiert), reicht es aus, die Elemente ab der Position `m->size - 2` wiederherzustellen. Anschließend fügt man das gelöschte Element wieder an seiner ursprünglichen Position ein (Zeile 22) und gibt den Fehlercode 2 zurück (Zeile 23).

5. Wenn `realloc` erfolgreich war, müssen noch die zwei Variablen in der komplexen Datenstruktur aktualisiert werden: Zum Einen der `elements`-Zeiger in Zeile 26 (da nach einem `realloc` auf eine andere Adresse gezeigt werden könnte als davor). Zum Anderen die Größe der Liste in Zeile 27, die nun 1 kleiner ist als davor. Abschließend wird die Funktion mit dem Erfolgscode 0 in Zeile 28 beendet.

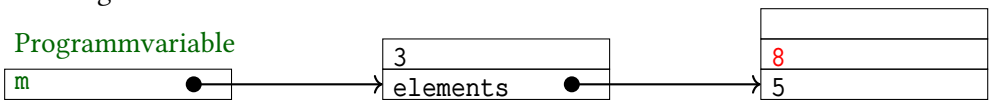
Da beim Löschen alle Elemente in der Liste durchsucht und verschoben werden müssen, ist die Zeitkomplexität linear zur Länge der Liste `k` – die Komplexität der Operation liegt somit in $O(k)$.

Beispiel: Wert 2 aus Liste 5, 2, 8 löschen (Aufruf `arraylist_delete(m, 2)`)

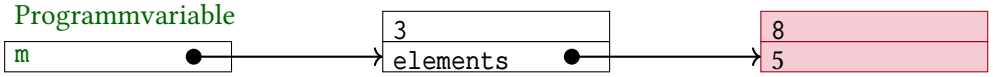
Ursprünglicher Zustand, Wert 2 wurde gefunden:



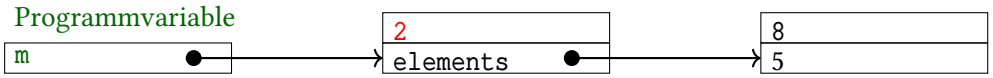
Nachfolgende Zahl nach vorne verschoben:



Feld verkleinert, aber `size` noch nicht angepasst:

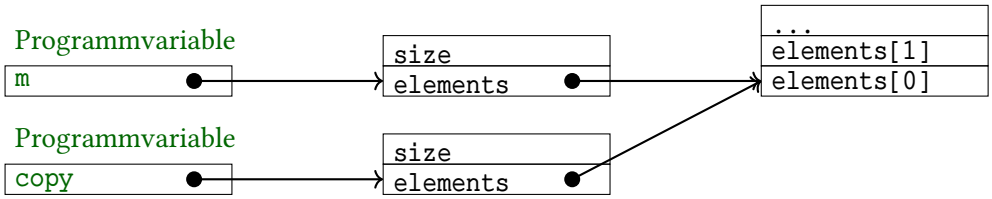


`size` angepasst, Funktion `arraylist_delete` erfolgreich beendet:



Kopien anlegen

Es ist **nicht** sinnvoll, eine flache Kopie eines dynamischen Feldes anzulegen. Dies kann man sich anhand der folgenden Grafik veranschaulichen:



Sobald ein Element hinzugefügt oder entfernt wird, würde die Angabe der Länge der Liste `size` im Original oder in der Kopie nicht mehr stimmen. Damit wäre die Datenstrukturinvariante verletzt, die maßgeblich zum Funktionieren des dynamischen Feldes beiträgt.

15.2.4 Programmbeispiel

Das folgende Hauptprogramm legt ein neues dynamisches Feld an und fügt ein Element ein. Anschließend wird das dynamische Feld ausgegeben und wieder freigegeben.

```
1 #include "arraylist-neu.h"
2 #include <stdio.h>
3
4 int main(void) {
5     int status;
6     arraylist *m = arraylist_create();
7     if (!m) return 1;
8     status = arraylist_insert(m, 5);
9     if (!status) { arraylist_destroy(m); return 1; }
```

```
10      printf("Ausgabe: \n");
11      arraylist_print(m);
12      arraylist_destroy(m);
13      return 0;
14  }
```

Nachdem das dynamische Feld in Zeile 6 angelegt wurde, muss in Zeile 7 überprüft werden, ob dies erfolgreich war (`malloc` in `arraylist_create` könnte `NULL` zurückgeliefert haben). Im Fehlerfall muss das Programm abgebrochen werden (Zeile 7). Auch wenn das Einfügen eines neuen Werts in Zeile 8 fehlschlägt, muss das Programm abgebrochen werden (Zeile 9). In diesem Fall muss zusätzlich der reservierte Speicherplatz wieder freigegeben werden (Zeile 9). Wenn alles erfolgreich war, wird das dynamische Feld in Zeile 11 ausgegeben und in Zeile 12 wieder freigegeben.

Dem gewitzten Studierenden fällt hierbei sicherlich auf: Es werden ausschließlich Verwaltungsoperationen verwendet. Somit muss nicht direkt auf die Komponenten der Datenstruktur zugegriffen werden. Damit ergeben sich vielerlei Vorteile. Zum einen muss dadurch der Benutzer / die Benutzerin nicht wissen, wie genau die Datenstruktur aufgebaut ist. Es reicht, nur die Operationen sowie eine kurze Beschreibung derer zu kennen. Außerdem ist das Programm dadurch weniger fehleranfällig, da wir davon ausgehen können, dass die Operationen korrekt arbeiten.

15.2.5 Erweiterungen

Wir haben eine einfache Umsetzung von dynamischen Feldern kennengelernt. Diese lässt sich z.B. durch folgende zusätzliche Datenstrukturinvarianten erweitern (diese führen allerdings zur Änderung der Implementierung mehrerer Verwaltungsoperationen):

- Wiederholungsfreie Listen zur Verwaltung von Mengen
- sortierte Liste
- verschiedene Lösungsstrategien wenn ein Wert mehrfach vorkommt, z.B. nur erstes oder alle Vorkommen löschen

Das Einfügen neuer Werte am Anfang des dynamischen Feldes wäre sehr aufwendig. Dadurch müssten alle anderen Werte nach hinten geschoben werden, was zu einer Verlängerung der Laufzeit führen würde. Eine Lösung hierfür sind *verkettete Listen*, die wir uns im folgenden Teilkapitel anschauen.

15.3 Einfach verkettete Listen

Dynamische Felder haben mehrere Nachteile: Zum Einen muss genügend Speicher am Stück verfügbar sein – im Falle einer Vergrößerung des dynamischen Feldes ggf. sogar fast doppelt so viel Speicher wie die das dynamische Feld benötigt: Dann wenn eine Vergrößerung nicht möglich ist, weil hinter dem letzten Eintrag des dynamischen Feldes bereits etwas anderes gespeichert ist, muss an einer anderen Stelle im Heap neuer Speicher reserviert werden und der Inhalt des dynamischen Feldes dorthin verschoben werden (das übernimmt alles die Funktion `realloc`). Zum Anderen

muss beim Löschen meist eine größere Zahl an Elementen verschoben werden. Sofern das dynamische Feld sortiert ist, gilt dies auch beim Einfügen von neuen Elementen. Daher lernen wir nun die *einfach verkettete Liste* kennen, die diese Probleme behebt.

15.3.1 Grundlagen einfach verketteter Listen

Was ist eine einfach verkettete Liste?
Einfach verkettete Listen werden ebenso wie dynamische Felder dazu verwendet, mehrere Werte zu speichern, die in unserem Fall nicht sortiert sind. Die Werte liegen dabei nicht hintereinander im Speicher, sondern für jeden Wert wird ein eigenes Stück Speicher im Heap reserviert. Zusätzlich zum Wert wird dort auch ein Zeiger zum nächsten Element hinterlegt. Wert und Zeiger bilden zusammen einen sogenannten *Knoten* (engl.: *node*). Um einen Wert einzufügen, erstellt man einen neuen Knoten, der auf den bisherigen neuesten Knoten zeigt und biegt den Zeiger, der auf den Beginn der verketteten Liste zeigt so um, dass er auf den neuen Knoten zeigt. Wenn man einen Wert sucht, muss man nacheinander die verschiedenen Knoten besuchen, solange bis man beim passenden Knoten angekommen ist. Im Gegensatz zum dynamischen Feld wird die die Länge der Liste **nicht** gespeichert.

Eine einfach verkettete Liste und ihre Verwaltungsoperationen könnte wie folgt aussehen:

```
1  #ifndef EVL_H_INCLUDED
2  #define EVL_H_INCLUDED
3
4  #define EMPTY_LIST NULL
5
6  typedef struct _node {
7      int element;
8      struct _node *next;
9  } node;
10 typedef node *list;
11
12 void list_destroy(list m);
13 int list_size(list m);
14 void list_print(const list m);
15 int list_isempty(const list m);
16 list list_insert(list m, int n);
17 list list_delete(list m, int n);
18 int list_iselem(list m, int n);
19
20 #endif
```

Das `#define EMPTY_LIST NULL` dient hierbei als Konstante, die anzeigen soll, wann man mit einer leeren verketteten Liste arbeitet. Der sprechende Name `EMPTY_LIST` soll dabei für mehr Klarheit sorgen als wenn man überall immer `NULL` hinschreibt.

Die einfach verkettete Liste selbst ist durch folgende komplexe Datenstruktur dargestellt:

```
1 typedef struct _node {
2     int element;
3     struct _node *next;
4 } node;
5 typedef node *list;
```

Man beachte, dass im `typedef` hier ein Zeiger auf ein `node` zu `list` umbenannt wird, d.h. überall, wo wir im Programm `list` schreiben, meinen wir damit einen Zeiger auf eine einfach verkettete Liste.

Die einfach verkettete Liste kann als Basis für andere Datenstrukturen dienen. Damit kann man beispielsweise eine *Last-In, First-Out* Datenstruktur erstellen. Also eine Liste, bei der das zuletzt hinzugefügte Element zurück gegeben wird.
In der Programmiersprache *LISP* spielen einfach verkettete Listen auch eine große Rolle, denn LISP-Programme selbst sind beispielsweise einfach verkettete Listen.

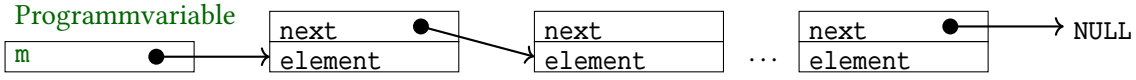
> Beispiel:
>
> Möchte man sich einfach verkettete Listen besser veranschaulichen, kann man sich diese Art der Datenstruktur als eine Art Schnitzeljagt veranschaulichen.
> Anfänglich besitzt man eine Referenz zum nächsten Ort, das man finden muss (das wäre `struct _node *next`).
> Ist man an diesem nächsten Ort angekommen, kann man Informationen finden (diese Informationen entsprechen dem `element`). Hier befindet sich auch eine Referenz zum nächsten Ort, das man finden muss.
> Aber erst einmal an einem beliebigen Ort angekommen, kann man daraus nicht mehr schließen, wo man hergekommen ist. Wir können also immer nur weiter zum nächsten Ort gehen, oder wieder zurück an den Anfang.
>
> Irgendwann ist man dann am Ziel angekommen. Das wäre dann die letzte Position der einfach verketteten Liste.

Einfach verkettete Listen im Arbeitsspeicher

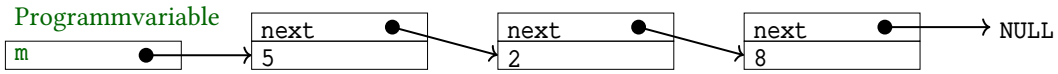
Die folgenden Datenstrukturinvarianten müssen gelten:

- Es gibt genau einen (ersten) Knoten ohne Vorgänger
- Es gibt genau einen (letzten) Knoten ohne Nachfolger
- Alle anderen Knoten haben jeweils genau einen Vorgänger und Nachfolger

Im Arbeitsspeicher würde die Datenstruktur `list m` folgendermaßen aussehen:



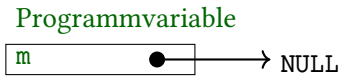
> Beispiel:
>
> Beispielhafte Darstellung der Liste `5,2,8`:



15.3.2 Anlegen und Löschen einfach verketteter Listen

Verwaltungsoperation: Leere Liste erstellen
Das anfängliche Erzeugen einer Liste ist relativ einfach. Hierfür muss nur eine Variable deklariert und mit `EMPTY_LIST` initialisiert werden.

```
list m = EMPTY_LIST;
```



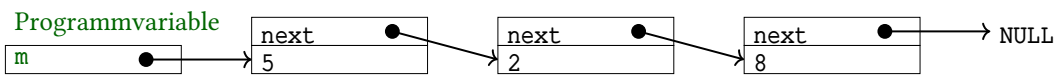
Verwaltungsoperation: Liste freigeben
Falls die Liste `m` nicht leer ist, muss sie rekursiv von hinten nach vorne freigegeben werden. Hier gilt das selbe wie für die dynamischen Felder: Würde man zuerst `m` freigeben, verliert man die Referenz zu den nächsten Objekten und kann sie nicht löschen. Es würde also auch hier wieder zu einem Speicherleck kommen.

```
1 void list_destroy(list m) {
2     if (m != EMPTY_LIST) {
3         list_destroy(m->next);
4         free(m);
5     }
6 }
```

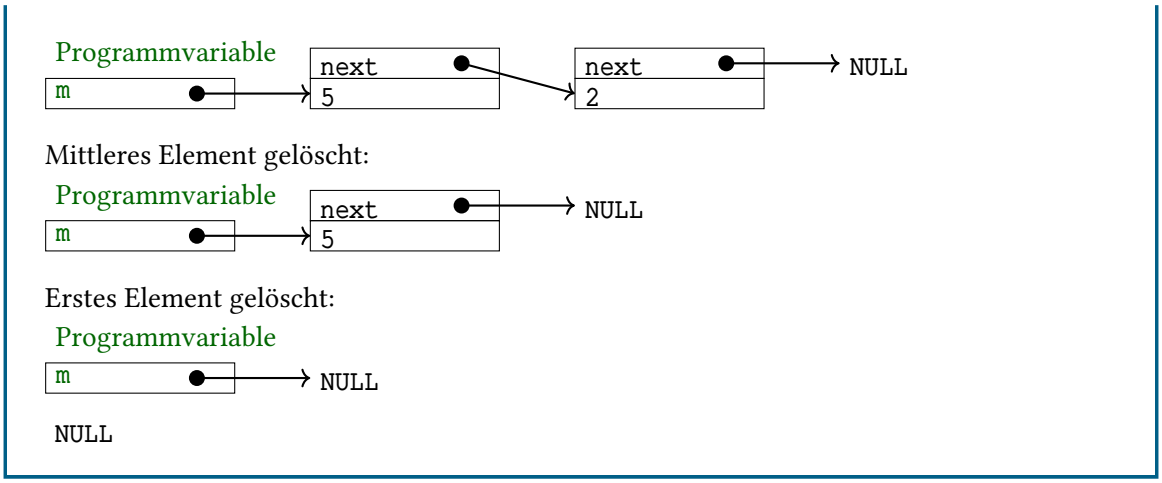
Wir rufen also so oft rekursiv `list_destroy(m->next)` auf, bis wir am Ende der Liste angelangt sind. Von da aus wird das letzte Element gelöscht und die zuletzt aufgerufene Funktion beendet sich. Danach sind wir wieder in der vorletzten aufgerufenen Funktion. Auch dort führen wir `free` aus, und gehen einen Schritt zurück.

Beispiel: Löschen der Liste 5, 2, 8

Ursprünglicher Zustand:



Letztes Element gelöscht:



15.3.3 Verwaltungsoperationen zum Zugriff auf einfach verkettete Listen

Verwaltungsoperation: Test auf Elementzugehörigkeit
Das Überprüfen auf Elementzugehörigkeit funktioniert ähnlich zum dynamischen Feld. Wir müssen alle Werte durchgehen und die Elemente darauf überprüfen, ob sie mit dem zu suchenden Wert übereinstimmen.
Dafür vergleichen wir den Wert `element` des aktuellen Knotens mit dem gesuchten Wert. Ist dieser nicht der gesuchten Knoten, gehen wir über den `next` Zeiger weiter zum nächsten Knoten. Das machen wir solange, bis wir den passenden Knoten gefunden haben oder am Ende der Liste angekommen sind. Wir durchlaufen also nacheinander Knoten für Knoten, denn wir besitzen keinen direkten Zugriff auf einzelne Knoten innerhalb der Liste.

```
1 int list_iselem(list m, int n) {
2     while (m != EMPTY_LIST && n != m->element) {
3         m = m->next;
4     }
5     return (m != EMPTY_LIST && n == m->element);
6 }
```

In der `while`-Schleife überprüfen wir einerseits, ob `m` ein gültiger Zeiger auf einen Knoten ist. Falls ja, dann prüfen wir, ob der in diesem Knoten gespeicherte Wert `m->element` nicht mit dem gesuchten Element übereinstimmt. Sofern `m` den Wert `EMPTY_LIST` hat, dürfen wir keine weitere Auswertung vornehmen und brechen die Schleife (und auch die Auswertung der Schleifenbedingung) ab. Dann wird in der `return`-Anweisung `EMPTY_LIST` auf Ungleichheit mit `EMPTY_LIST` verglichen, was falsch ist. Der zweite Teil der `return`-Anweisung wird damit nicht mehr ausgewertet, sondern gleich *falsch* zurückgegeben und damit angezeigt, dass `n` nicht gefunden wurde.

Falls `m` eine gültige Adresse ist, dürfen wir den Wert dieses Knotens via `m->element` auslesen und mit dem gesuchten `n` vergleichen. Wenn `n` nicht der gesuchte Wert ist, wird der Zeiger `m` umgebogen auf den nächsten Knoten, welcher in `m->next` hinterlegt ist und die Schleife erneut durchlaufen.

Sofern n und `m->element` übereinstimmen, haben wir den gesuchten Wert in unserer verketteten Liste gefunden. Daher verlassen wir die Schleife und geben in der `return`-Anweisung wahr und wahr = wahr zurück.

Die Zeitkomplexität dieser Funktion wächst linear zur Länge der verketteten Liste k – die zugehörige Komplexitätsklasse ist somit $O(k)$.

Verwaltungsoperation: Element einfügen

Elemente werden bei uns immer an erster Stelle eingefügt. Das heißt, sie werden am Anfang der Liste platziert.

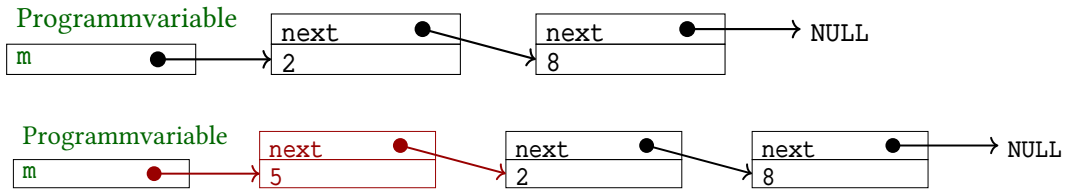
```
1  list list_insert(list m, int n) {
2      node *neu = malloc(sizeof(node));
3      if (neu == NULL)
4          return NULL;
5      neu->element = n;
6      neu->next = m;
7      return neu;
8  }
```

In Zeile 2 reservieren wir dynamisch den Speicherbereich für einen neuen Knoten auf dem Heap. Anschließend überprüfen wir in Zeile 3, ob dabei ein Fehler aufgetreten ist. Sollte das der Fall sein, brechen wir das Einfügen des neuen Wertes ab, die Liste kann somit unverändert weiterverwendet werden. Im Erfolgsfall (ab Zeile 5) speichern wir den `element`-Wert für unseren neuen Knoten und setzen seinen `next`-Zeiger auf den bisherigen Anfang der verketteten Liste. Anschließend geben wir die Adresse unseres neuen Knotens und somit der neuen Liste zurück.

Da unsere Einfügeoperation unabhängig von der Länge der verketteten Liste ist, haben wir eine konstante Zeitkomplexität $O(1)$.

Beispiel:

Möchten wir beispielsweise in die Liste 2, 8 das neue Element 5 einfügen, sähe das folgendermaßen aus:



Verwaltungsoperation: Element löschen (rekursiv)
Um ein spezifisches Element aus der Liste zu löschen, müssen wir es ersteinmal finden. Dazu gehen wir von Knoten zu Knoten, um das Element zu finden. Haben wir es gefunden, können wir es nicht einfach so löschen. Die Struktur der Liste muss weiterhin beibehalten werden. Dafür müssen die Zeiger neu gesetzt werden, damit nichts verloren geht.

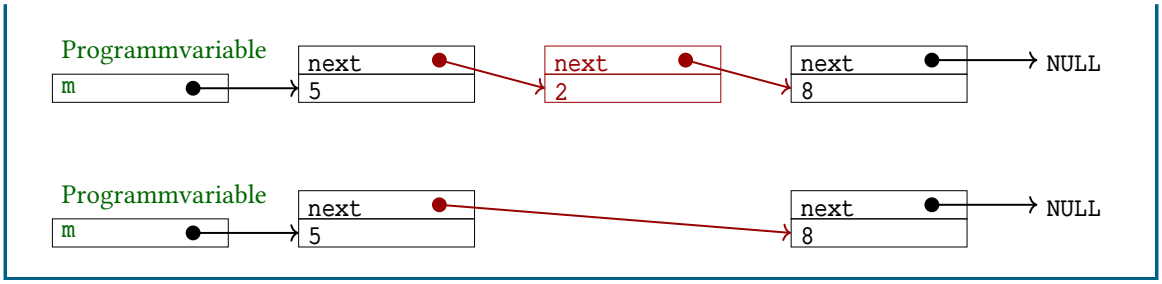
```
1  list list_delete(list m, int n) {
2      if (m == EMPTY_LIST) {
3          return EMPTY_LIST;
4      } else if (n != m->element) {
5          list neu = list_delete(m->next, n);
6          m->next = neu;
7          return m;
8      } else {
9          list rest = m->next;
10         free(m);
11         return rest;
12     }
13 }
```

Unsere Implementierung von `list_delete` funktioniert rekursiv: Es wird immer die Liste zurückgegeben, die der Aufrufer über seinen `next`-Zeiger verbinden soll. Dafür gibt es folgende Fälle:

1. `if (m == EMPTY_LIST)`: Dies ist ein Basisfall, es gibt keine weitere Liste, d.h. wir sind am Ende angelangt. Somit kann der Aufrufer auch nichts verbinden und soll seinen `next`-Zeiger auf `EMPTY_LIST` richten.
2. `else if (n != m->element)`: Der aktuelle Knoten entspricht **nicht** dem gesuchten Knoten, es soll also alles so bleiben wie es ist. Daher wird der rekursive Aufruf einfach durchgereicht, wobei der eigene `next`-Zeiger neu gesetzt wird auf die Rückgabe von `list_delete`. Sollte der nächste Knoten der zu löschende Knoten sein, wird der Aufruf von `list_delete` nämlich die Adresse des übernächsten Knotens zurückgeben (siehe nachfolgenden Fall). An den Aufrufer wird die eigene Adresse zurückgegeben, um weiterhin in der Liste zu bleiben.
3. `else`: Hier ist der aktuelle Knoten der Knoten der gelöscht werden soll. Er gibt dem Aufrufer daher die Adresse des nächsten Knotens zurück (wird zukünftig also übersprungen) und gibt seinen eigenen Speicherbereich frei. Da kein weiterer rekursiver Aufruf erfolgt, wird nur das erste Vorkommen des gesuchten Wertes gelöscht.

Somit kommen wir auf eine lineare Laufzeit für das Löschen eines Elementes.

> Beispiel:
>
> Wenn wir eine Liste 5, 2, 8 besitzen und das Element 2 löschen wollen, sähe das im Arbeitsspeicher folgendermaßen aus:



15.3.4 Programmbeispiel

Das folgende Hauptprogram legt zwei leere verkettete Listen an, erstellt einen neuen Knoten und nimmt diesen im Erfolgsfall in der verketteten Liste `m` auf. Nachdem die verkettete Liste ausgegeben und freigegeben wurde, wird das Programm wieder beendet.

```
1  #include "evl.h"
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  int main(void) {
6      list m = EMPTY_LIST, neu;
7      neu = list_insert(m, 5);
8      if (!neu) { list_destroy(m); return 1; }
9      m = neu;
10     printf("Ausgabe: \n");
11     list_print(m);
12     list_destroy(m);
13     return 0;
14 }
```

15.3.5 Bewertung und Ausblick

Im Vergleich zu dynamischen Feldern zeigt sich, dass auch verkettete Listen ihre Vor- und Nachteile haben: Beides sind Implementierungen zur Verwaltung dynamischer Listen und Mengen. Einfügen und Löschen von Elementen geht bei verketteten Listen schneller, die Ausgabe und der Test, ob ein Element in der Liste enthalten ist, bei dynamischen Feldern.

Wenn man zusätzlich fordert, dass die Liste wiederholungsfrei ist, muss man `list_insert` anpassen, da die einfach verkettete Liste dann bei jedem Einfügen durchlaufen werden muss, um sicherzustellen, dass es das einzufügende Element noch nicht gibt.

Leichter wird das, wenn man die zusätzliche Datenstrukturinvariante aufstellt, dass die Liste sortiert sein soll – dann muss man nicht immer bis zum Ende der Liste laufen, um sicher zu sein, dass es einen Wert nur einmal gibt. Hier hat die einfach verkettete Liste zusätzlich den Vorteil, dass nicht alle Elemente verschoben werden müssen, wenn ein Element eingefügt oder gelöscht wird. Stattdessen können neue Elemente an beliebigen Stellen in der Liste einfach hinzugefügt

oder entfernt werden, indem die entsprechenden Zeiger umgebogen werden. Allerdings müsste auch hier die Implementierung verschiedener Verwaltungsoperationen geändert werden. Die Vor- und Nachteile dieser Implementierungen von sortierten Listen werden im Kapitel 15.6.2 weiter diskutiert.

Mit einer simplen Ergänzung lassen sich neue Elemente auch am Ende in $O(1)$ einfügen. Bislang müsste man jedes mal alle Elemente durchlaufen, um ein neues Element am Ende hinzuzufügen. Dies ist wenig effizient. Stattdessen könnte man aber einen zusätzlichen Zeiger speichern, der auf das Ende der Liste zeigt. Dies setzen wir im Kapitel 15.5 um, wo wir mit Hilfe von einfach verketteten Listen eine Warteschlange realisieren.

Eine andere Erweiterung sind *doppelt verkettete Listen*, bei denen ein Knoten nicht nur auf den nächsten sondern auch auf den vorherigen Knoten zeigt, um die Navigation in beide Richtungen zu ermöglichen. Hierauf gehen wir kurz im Kapitel 15.6.1 ein.

15.4 Stacks

Wir haben bereits im Kapitel 2 einen Stack kennengelernt, dort im Zusammenhang mit der Übergabe von Funktionsparametern an eine Funktion. Nun arbeiten wir mit einer Datenstruktur, die ebenso einen Stack umsetzt.

15.4.1 Grundlagen von Stacks

Im Kontext dynamischer Datenstrukturen ist ein **Stack (Kellerspeicher)** eine Datenstruktur für dynamische Listen, die nur einen **eingeschränkten Zugriff** auf die Listenelemente erlaubt. Eingeschränkter Zugriff bedeutet in diesem Fall, dass der Datenzugriff nach dem **LIFO-Prinzip** (**L**ast **I**n **F**irst **O**ut) funktioniert. Das bedeutet, wir besitzen nur Zugriff auf das letzte Element, das hinzugefügt wurde.

Beispiel:

Den Stack kann man sich gut mithilfe einer Keksdose vorstellen.

Abbildung 1: Eine Keksdose

Stellen wir uns vor, wir haben extra breite, schmackhafte Kekse. Da wir viel Wert auf Ästhetik legen, bewahren wir sie in dieser Dose auf.

Unser Stack ist nun diese Keksdose. Die Kekse darin sind die Elemente, die wir gespeichert haben. Da unsere Kekse besonders breit sind, können wir immer nur den obersten Keks heraus nehmen (oder anschauen). Somit können wir auch einen neuen Keks nicht beliebig hineinlegen. Ein neuer Keks (bzw. neues Element) muss immer ganz oben hineingelegt werden.

Unsere Implementierung basiert auf der einfach verketteten Liste. Man kann Stacks aber auch mit ein dynamischen Feldern realisieren.

```
1  #ifndef STACK_H_INCLUDED
2  #define STACK_H_INCLUDED
3
4  #define EMPTY_STACK NULL
5
6  typedef struct _node
7  {
8      int element;
9      struct _node *next;
10 } node;
11 typedef node* stack;
12
13 void stack_destroy(stack m);
14 int stack_isempty(stack m);
15 int stack_push(stack *m, int n);
16 int stack_pop(stack *m);
17 int stack_top(stack m);
18
19 #endif
```

Für bessere Lesbarkeit unseres C-Quellcodes definieren wir, dass `EMPTY_STACK` der Konstante `NULL` entspricht.

Der Stack selbst ist durch folgende komplexe Datenstruktur dargestellt:

```
1  typedef struct _node
2  {
3      int element;
4      struct _node *next;
5  } node;
6  typedef node* stack;
```

Hier sehen wie abgesehen vom Namen keinen Unterschied zur einfach verketteten Liste. Es unterscheiden sich somit nur die Verwaltungsoperationen. Ein Stack besitzt immer mindestens folgende drei Operationen:

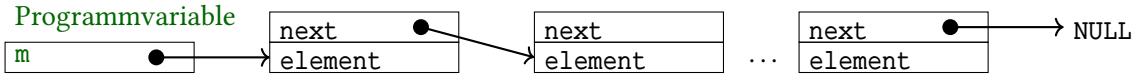
- `push`: ein neues Element an oberster Stelle einfügen
- `pop`: oberstes Element lesen / zurückgeben und löschen
- `top`: oberstes Element lesen / zurückgeben (und nicht löschen)

Stacks besitzen viele Anwendungsgebiete. Ein Beispiel wäre die Implementierung eines *Rückgängig*-Befehls. Also das Rückgängig-Machen von Änderungen, wie wir das z.B. aus Office-Programmen oder unserem Entwicklungs-Editor kennen. Hierbei können wir uns vorstellen, dass jede Änderung einen neuen Eintrag im Stack darstellt (`push`-Befehl; füge eine neue Änderung hinzu). Somit haben wir nur Zugriff auf die letzte Änderung. Möchten wir nun etwas rückgängig machen, führen wir einen `pop`-Befehl aus und wissen, wie unser Stand vor der letzten Änderung aussah. Diesen letzten Stand können wir nutzen, um etwas rückgängig zu machen.

Der Stack im Arbeitsspeicher
Unser Stack besitzt die selben Datenstrukturinvarianten wie auch eine einfach verkettete Liste (da er eine einfach verkettete Liste ist):

- Es gibt genau einen (ersten) Knoten ohne Vorgänger
- Es gibt genau einen (letzten) Knoten ohne Nachfolger
- Alle anderen Knoten haben jeweils genau einen Vorgänger und Nachfolger

Im Arbeitsspeicher würde die komplexe Datenstruktur `stack m` folgendermaßen aussehen:



Auch hier wieder: Analog zur einfach verketteten Liste.

15.4.2 Anlegen und Löschen von Stacks

Das Anlegen und Löschen funktioniert genau wie bei der einfach verketteten Liste (siehe Kapitel 15.3.2). Möchte man einen neuen Stack erzeugen, deklariert man dafür eine Variable vom Typ `stack` und initialisiert sie mit `NULL`. Das Löschen muss wieder rekursiv erfolgen, damit es nicht zum Speicherleck kommt.

15.4.3 Verwaltungsoperationen zum Zugriff auf Stacks

Da man bei einem Stack immer nur auf das oberste Element zugreifen kann, werden abgesehen von `isempty` die Verwaltungsoperationen durch die drei Operationen `push`, `top` und `pop` ersetzt.

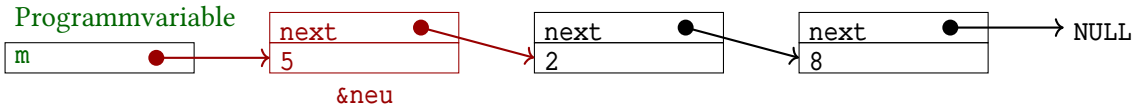
Verwaltungsoperation: Neues Element an oberster Stelle einfügen (push)

`stack_push` fügt ein neues Element an oberster Stelle ein.

```
1  int stack_push(stack *m, int n)
2  {
3      node *neu = malloc(sizeof(node));
4      if (neu == NULL)
5          return 0;
6      neu->element = n;
7      neu->next = *m;
8      *m = neu;
9      return 1;
10 }
```

Um ein neues Element im Stack einzufügen, gehen wir ähnlich bei bei `list_insert` vor (vgl. Seite 17). Der Unterschied liegt in dem, was übergeben und zurückgegeben wird:
Die Adresse des erweiterten Stacks wird nicht zurückgegeben, sondern der übergebene Stack wird überschrieben (hier gilt das Call-by-Reference-Prinzip). Der Rückgabewert unterscheidet allein zwischen Erfolgs- und Fehlerfall (0 wenn `malloc` einen Fehler verursacht hat und 1 wenn alles geklappt hat).

Beispielhafter Aufruf für die Programmvariable `stack m` und einen Stack, der bislang die Elemente 2 und 8 enthält: `int n = stack_push(&m, 5);`



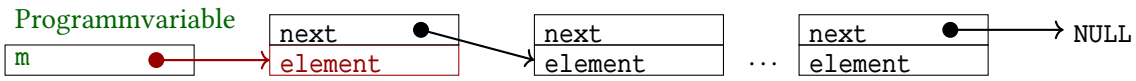
Verwaltungsoperation: Oberstes Element lesen (top)

`stack_top` liest das oberste Element und gibt es zurück, aber löscht es nicht.

```
1  int stack_top(stack m)
2  {
3      if (m == EMPTY_STACK) {
4          printf("\nstack_top called on empty stack...");
5          return 0;
6      }
7      return m->element;
8  }
```

Wir können nur auf das erste Element der Liste direkt zugreifen. Wir übergeben also das `stack`-Objekt und geben das erste Element darin zurück. Falls der leere Stack übergeben wird, geben wir eine Fehlermeldung aus.

Beispielhafter Aufruf für die Programmvariable `stack m`:
`int n = stack_top(m);`



Verwaltungsoperationen: Oberstes Element lesen und löschen (pop)

`stack_pop` liest das oberste Element, gibt es zurück und löscht es.

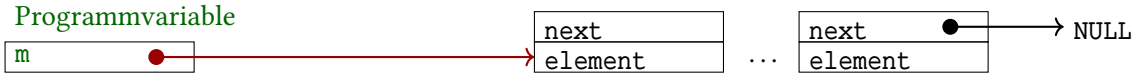
```
1  int stack_pop(stack *m)
2  {
3      int n = stack_top(*m);
4      stack next = (*m)->next;
5      free(*m);
6      *m = next;
7      return n;
8  }
```

Nachdem wir das Auslesen des obersten Elements bereits in `stack_top` implementiert haben, rufen wir als erstes diese Funktion auf. Danach biegen wir die Programmvariable `m` auf den nächsten Knoten um (Zeilen 4+6) und löschen den aktuellen Knoten (Zeile 5), bevor wir die gespeicherte Zahl zurückgeben (Zeile 7).

Ebenso wie bei `stack_push` wird direkt der übergebene Stack `m` verändert. Die Adresse des reduzierten Stacks wird nicht zurückgegeben, sondern an der Stelle `*m` gespeichert (hier gilt das Call-by-Reference-Prinzip). Es wird nur der oberste Wert gelöscht und zurück gegeben.

Man muss beachten, dass für `m` nicht `NULL` übergeben werden darf. Alternativ könnte man auch eine Fehlerbehandlung wie bei `stack_top` einbauen.

Zustand nach Aufruf für die Programmvariable `stack m`:
`int n = stack_pop(m);`



15.4.4 Programmmbeispiel

Hauptprogramm: Beispiel

```
1  #include <stdio.h>
2
3  #include "stack.h"
4
5  int main(void)
6  {
7      stack s = EMPTY_STACK;
```

```
 8      stack_push(&s, 5);
 9      stack_push(&s, 7);
10      stack_push(&s, 10);
11      printf("%i ", stack_pop(&s));
12      printf("%i ", stack_pop(&s));
13      stack_destroy(s);
14      return 0;
15  }
```

Wir legen statisch einen neuen Stack an (Zeile 7), in dem wir nacheinander die Werte 5, 7 und 10 ablegen (Zeilen 8-10). Anschließend nehmen wir in den Zeilen 11+12 die obersten beiden Werte heraus und geben diese aus, bevor wir den Stack in Zeile 13 wieder freigeben.

15.4.5 Fazit

Wie wir gesehen haben, bieten Stacks nur eingeschränkten Zugriff auf eine Datenstruktur: Es kann immer nur auf das zuletzt hinzugefügte Element zugegriffen werden, womit das LIFO-Prinzip umgesetzt wird (**L**ast **I**n, **F**irst **O**ut). Eine mögliche Anwendung wäre z.B. die Realisierung einer *Rückgängig*-Funktionalität, wie wir sie aus Office-Programmen oder unserem Entwicklungs-Editor kennen. Stacks lassen sich z.B. mit dynamischen Feldern oder einfach verketteten Listen implementieren.

Zum Zugriff verwenden wir die Operationen `push` (Element einfügen), `pop` (Element zurückgeben und löschen) und `top` (Element zurückgeben, aber im Stack lassen). Da diese alle immer nur auf den vordersten Knoten zugreifen, wird die Datenstruktur nie komplett durchlaufen. Die Stack-Operationen lassen sich daher in $O(1)$ realisieren.

15.5 Queues

15.5.1 Grundlagen von Queues

Eine **Queue (Warteschlange)** (sprich: [kju]) ist eine Datenstruktur für dynamische Listen, die nur einen eingeschränkten Zugriff auf die Listenelemente erlaubt. Das bedeutet, ähnlich wie beim Stack, kann nur auf ein bestimmtes Element in der Liste zugegriffen werden. Der Unterschied zum Stack ist, dass eine Queue den Datenzugriff nach dem **FIFO-Prinzip** realisiert: Dem **F**irst **I**n, **F**irst **O**ut) Prinzip. Das bedeutet, es kann immer nur auf das älteste Element zugegriffen werden.

> Beispiel:
>
> Die Queue kann man auch, wie der Name schon sagt, anhand einer Warteschlange (bspw. in der Mensa) verdeutlichen. Die Warteschlange selbst stellt dabei die Queue da. Die Menschen, die darin warten, stellen die Elemente der Queue dar.
> Hierbei gilt: Die Person, die zuerst die Warteschlange betreten hat, bekommt auch als erstes Essen. Danach kommt die Person dran, sie sich als zweites in die Warteschlange gestellt hat.

Betritt man die Warteschlange (man fügt also ein neues Element hinzu), muss man warten, bis alle anderen Personen dran gekommen sind (also alle vorherigen Elemente herausgenommen wurden).

Eine Queue kann intern als einfach verkettete Liste oder als dynamisches Feld realisiert werden. Wir verwenden dazu im Folgenden eine einfach verkettete Liste. Eine Implementierung könnte nun folgendermaßen aussehen:

```
1  #ifndef QUEUE_H_INCLUDED
2  #define QUEUE_H_INCLUDED
3
4  #include "evl.h"
5
6  #define EMPTY_QUEUE NULL
7
8  typedef struct _queue {
9      list first;
10     list last;
11 } * queue;
12
13 queue queue_create(void);
14 void queue_destroy(queue m);
15 int queue_isempty(queue m);
16 int queue_enter(queue m, int n);
17 int queue_remove(queue m, int *success);
18 int queue_first(queue m, int *success);
19
20 #endif
```

Diesmal binden wir die `.h`-Datei der einfach verketteten Liste ein und ergänzen diese um eine weitere komplexe Datenstruktur `struct _queue` und deren Verwaltungsoperationen. Die Queue selbst wird durch folgende komplexe Datenstruktur realisiert:

```
1  typedef struct _queue {
2      list first;
3      list last;
4  } * queue;
```

Da wir wieder immer nur mit dem Zeiger auf das `struct` arbeiten werden, haben wir das `typedef` diesmal gleich so geschrieben, dass es sich auf den Zeiger bezieht.

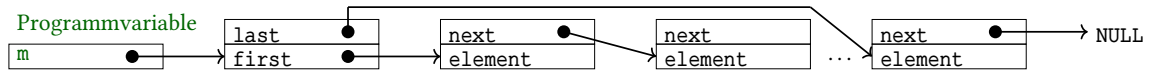
Die Queue selbst besitzt zwei Elemente, mit jeweils folgenden Datenstrukturinvarianten:

- `first` zeigt auf das erste Element der Liste
- `last` zeigt auf das letzte Element der Liste

Außerdem besitzt sie, ähnlich zum Stack, immer mindestens folgende drei Operationen:

- `enter`: ein neues Element an letzter Stelle einfügen
- `remove`: erstes Element lesen und löschen
- `first`: erstes Element lesen (und nicht löschen)

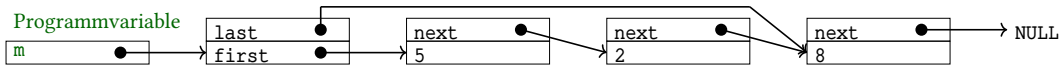
Queues im Arbeitsspeicher
Allgemeine Übersicht für eine Programmvariable `queue m` im Arbeitsspeicher:



Durch diese Darstellung wird auch ersichtlich: Der hintere Teil einer Queue wird durch eine bereits bekannte, einfach verkettete Liste dargestellt. Darin befinden sich die Werte, die in dieser Liste gespeichert werden.
Einzig der vordere Teil unterscheidet sich: Unsere Programmvariable zeigt auf ein `struct`, welches die Elemente `first` und `last` besitzt. `first` zeigt auf das erste Element der einfach verketteten Liste, während `last` auf das letzte Element dieser Liste zeigt.

Beispiel:

Darstellung der Liste `5,2,8`:



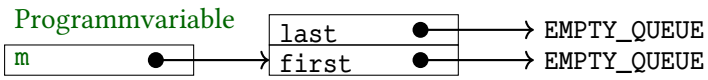
15.5.2 Anlegen und Löschen von Queues

Verwaltungsoperation: Queue erzeugen
Möchte man eine Queue erzeugen, muss das Objekt sowie dessen Komponenten `last` und `first` initialisiert werden. Dies geschieht mit `EMPTY_QUEUE`:

```
1  queue queue_create(void)
2  {
3      queue q = malloc(sizeof(struct _queue));
4      if (q == NULL)
5          return NULL;
6      q->first = EMPTY_QUEUE;
7      q->last = EMPTY_QUEUE;
8      return q;
9  }
```

Da wir mit dem `typedef` in der `.h`-Datei nur einen Zeiger auf ein `struct _queue` bezeichnen, müssen wir in Zeile 3 Speicher für ein `struct _queue` reservieren (und nicht für ein queue!). Die Zeiger `first` und `last` initialisieren wir mit `EMPTY_QUEUE`, da die Queue ja noch leer ist.

Nach dem Anlegen schaut unsere Queue im Arbeitsspeicher folgendermaßen aus:



Verwaltungsoperation: Queue freigeben

Möchte man eine Queue freigeben, muss die Operationsreihenfolge beachtet werden. Ansonsten kann es auch hier wieder zum Speicherleck führen.
Da der hintere Teil der Queue eine einfach verkettete Liste ist, kann dafür auch die Operation zum Löschen einer einfach verketteten Liste verwendet werden (sofern sie nicht leer ist). Anschließend lässt sich problemlos die Queue-Verwaltungsdatenstruktur löschen.

```
1  void queue_destroy(queue m)
2  {
3      if (m != NULL) {
4          if (!queue_isempty(m))
5              list_destroy(m->first);
6          free(m);
7      }
8  }
```

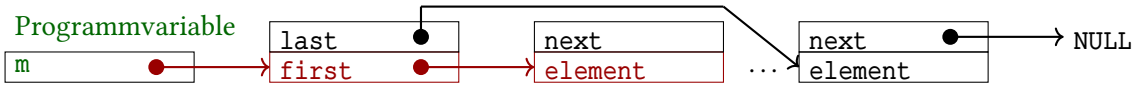
Wie schon beschrieben wird zuerst die einfach verkettete Liste `m->first` über die Freigabe-Funktion für einfach verkettete Listen `list_destroy` in Zeile 5 freigegeben. Anschließend kann man die Queue-Verwaltungsdatenstruktur löschen (Zeile 6).

15.5.3 Verwaltungsoperationen zum Zugriff auf Queues

Verwaltungsoperation: Erstes Element lesen (first)

```
1  int queue_first(queue m, int *success)
2  {
3      if (queue_isempty(m)) {
4          if (success != NULL)
5              *success = 0;
6          return 0;
7      }
8      if (success != NULL)
9          *success = 1;
10     return (m->first)->element;
11 }
```

Bevor wir auf das erste Element in der Queue zugreifen, prüfen wir in Zeile 3, ob die Queue leer ist. Dann gibt es kein erstes Element und wir brechen die Funktion ab (Zeile 6). Falls für `success` etwas anderes als `NULL` übergeben wurde, wird in dieser Variable gespeichert, ob der Zugriff erfolgreich war oder nicht – in diesem Fall also 0 für fehlerhaften Zugriff. Ansonsten speichern wir 1 in Zeile 9 und geben den Wert des ältesten Knotens zurück. Der Zugriff auf diesen Knoten erfolgt dabei über die Komponente `first`.



Verwaltungsoperation: Neues Element an letzter Stelle einfügen (enter):

Das Einfügen ähnelt dem Einfügen in eine einfach verkettete Liste:

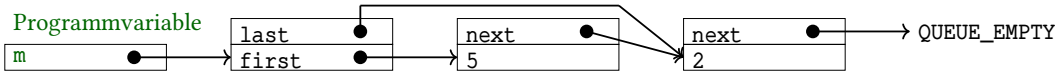
```
1  int queue_enter(queue m, int n)
2  {
3      node *neu = malloc(sizeof(node));
4      if (neu == NULL)
5          return 0;
6      neu->element = n;
7      neu->next = EMPTY_QUEUE;
8      if (m->last != EMPTY_QUEUE)
9          (m->last)->next = neu;
10     if (m->first == EMPTY_QUEUE)
11         m->first = neu;
12     m->last = neu;
13     return 1;
14 }
```

Zunächst reservieren wir Speicher (Zeile 3), brechen im Fehlerfall ab (Zeilen 4+5) und initialisieren den neuen Knoten (Zeilen 6+7). Danach (ab Zeile 8) wird der neue Knoten in die Queue eingereiht. Hierbei muss beachtet werden, dass wir zwei Komponenten behandeln müssen: `first` und `last`. Der Knoten wird immer an der letzten Stelle hinzugefügt. Deswegen müssen wir unseren bisher letzten Knoten ansprechen und mit dem neuen Knoten verlinken (Zeile 9). Dies dürfen wir aber nur tun, wenn es bisher auch einen letzten Knoten gegeben hat, d.h. die Warteschlange nicht leer war (Zeile 8).

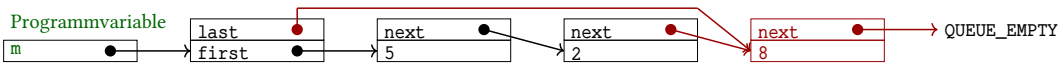
Sollte vor dem Hinzufügen des neuen Knotens die Queue davor leer gewesen sein, so ist unser neuer Knoten der letzte und gleichzeitig auch der erste Knoten in der Queue. Somit müssen wir auch `first` auf den neuen Knoten setzen (Zeilen 10+11). In Zeile 12 wird schließlich noch der Zeiger `last` auf unseren neuen Knoten aktualisiert. Das muss unabhängig von der bisherigen Länge der Queue immer passieren, da der neue Knoten stets der letzte ist.

Beispiel:

Wir wollen nun beispielsweise unseren neuen Wert 8 einfügen:



Nach dem Einfügen sieht unsere Queue folgendermaßen aus:



Verwaltungsoperation: Erstes Element lesen und löschen (remove)

Diese Funktion ist analog aufgebaut zur pop-Operation vom Stack (vgl. Seite 23):

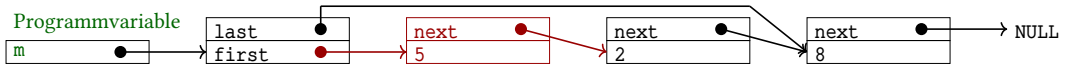
```
1  int queue_remove(queue m, int *success)
2  {
3      int success_first;
4      list next;
5      int n = queue_first(m, &success_first);
6      if (!success_first) {
7          if (success != NULL)
8              *success = 0;
9          return 0;
10     }
11     next = (m->first)->next;
12     free(m->first);
13     m->first = next;
14     if (next == EMPTY_QUEUE)
15         m->last = EMPTY_QUEUE;
16     if (success != NULL)
17         *success = 1;
18     return n;
19 }
```

In Zeile 5 lesen wir mit `queue_first` den Wert des ersten Knotens aus. Dabei speichern wir in `success_first`, ob dies erfolgreich war und werten es in Zeile 6 aus. Falls für `success` etwas anderes als `NULL` übergeben wurde, wird darin gespeichert ob der Zugriff erfolgreich war. Unabhängig davon, ob für `success` eine Variable übergeben wurde, wird die Funktion im Fehlerfall abgebrochen (Zeile 9).

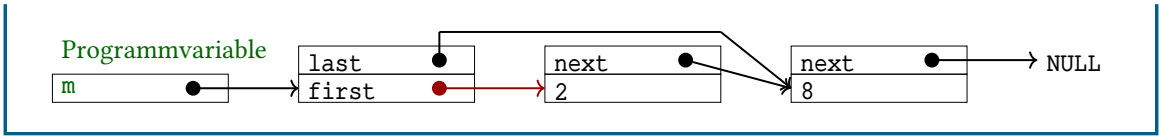
Im Erfolgsfall (ab Zeile 11) merken wir uns die Adresse des zweiten Knotens in `next`. Anschließend kann der bisher erste Knoten gelöscht werden (Zeile 12) und der Zeiger `first` auf den zweiten Knoten umgebogen werden (Zeile 13). Falls die Warteschlange nun leer ist, wird in Zeile 15 der Zeiger auf den letzten Knoten `last` noch auf die leere Warteschlange `EMPTY_QUEUE` gesetzt. Am Ende (ab Zeile 16) wird der `success`-Wert auf den Erfolgsfall gesetzt, falls dieser übergeben wurde, abschließend gibt die Funktion die gelesene Zahl zurück.

Beispiel:

Wir wollen nun beispielsweise aus unserer Queue mit den Elementen 5, 2, 8 die `remove` Funktion anwenden:



Nach dem Aufruf sieht unsere Queue folgendermaßen aus:



15.5.4 Programmmbeispiel

Hauptprogramm: Beispiel

```
1  #include "queue.h"
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  int main(void) {
6      int status, zahl;
7      queue m = queue_create();
8      if (m == NULL) {
9          return 1;
10     }
11
12     status = queue_enter(m, 5);
13     if (!status) {
14         queue_destroy(m);
15         return 1;
16     }
17
18     zahl = queue_remove(m, &status);
19     if(status != 0) {
20         printf("%i\n", zahl);
21     }
22
23     queue_destroy(m);
24     return 0;
25 }
```

Wir erstellen eine neue Queue in Zeile 7 und brechen unser Programm im Fehlerfall ab (Zeilen 8-10). Danach reihen wir die Zahl 5 neu ein (Zeile 12) und brechen das Programm im Fehlerfall ab (Zeilen 13-16). Schließlich entfernen wir unser vorderstes Element aus der Queue und geben dieses aus (Zeilen 18-21). Abschließend geben wir die Queue in Zeile 23 wieder frei und beenden dann unser Programm.

15.5.5 Fazit

Ebenso wie Stacks bieten Queues nur eingeschränkten Zugriff auf eine Datenstruktur. Während Stacks aber das LIFO-Prinzip umsetzen, setzen Queues das FIFO-Prinzip (**F**irst **I**n, **F**irst **O**ut) um.

Damit kann immer nur auf das älteste Element zugegriffen werden. Queues lassen sich somit überall einsetzen, wo etwas nacheinander verarbeitet werden soll. Zur Implementierung eignen sich v.a. einfach verkettete Listen.

Zum Zugriff verwenden wir die Operationen `enter` (in Warteschlange anstellen), `remove` (denjenigen, der am längsten wartet, dran nehmen) und `first` (denjenigen, der am längsten wartet, fragen, welchen Wert er gespeichert hat, aber in der Warteschlange stehen lassen). Dabei erstellt die Operation `enter` einen neuen Knoten, der immer hinten in der einfach verketteten Liste eingereiht wird. Durch den `last`-Zeiger muss die einfach verkettete Liste dabei nie durchlaufen werden. Die anderen beiden Operationen `remove` und `first` fragen immer nur den vordersten Knoten ab, so dass auch hier die einfach verkettete Liste nie durchlaufen werden muss. Alle Operationen sind somit in $O(1)$ realisierbar.

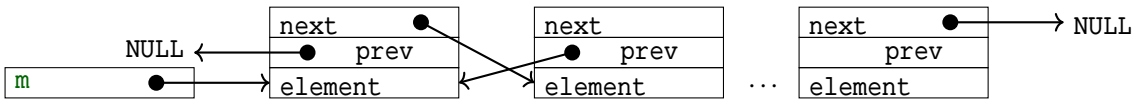
15.6 Ausblick

15.6.1 Weitere Datenstrukturen: Doppelt verkettete Listen

Eine doppelt verkettete Liste ist ähnlich der einfach verketteten Liste. Der große Unterschied hierbei ist, dass jedes Element mit seinem Nachfolger **und Vorgänger** verbunden ist:

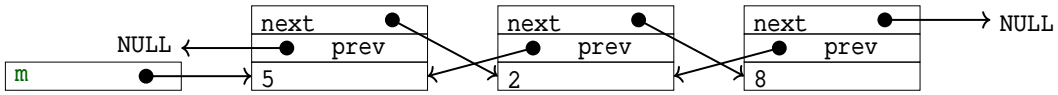
```
1 typedef struct _node {
2     int element;
3     struct _node *prev;
4     struct _node *next;
5 } node;
6 typedef node* list;
```

Allgemeine Übersicht:



Beispiel:

Darstellung der Liste `5,2,8`:



15.6.2 Implementierung von sortierten Listen

Will man mit den vorgestellten Datenstrukturen **sortierte Listen** verwalten, so ändert sich die Implementierung der meisten Verwaltungsfunktionen:

- Die Verwaltungsfunktionen **zur Änderung der Liste** (Einfügen, Löschen) müssen die Sortierung erhalten und brauchen dadurch mehr Rechenzeit.
- Die Verwaltungsfunktionen **zum Suchen und Lesen in der Liste** können die Sortierung für eine schnellere Implementierung benutzen.

Beispiele:

- Das **Einfügen eines neuen Werts** in ein sortiertes dynamisches Feld ist viel aufwändiger als bei einem unsortierten dynamischen Feld, da man zuerst die richtige Einfügeposition in der Sortierung finden muss und dann alle nachfolgenden Werte verschieben muss.
- Die **binäre Suche** in einem sortierten dynamischen Feld ist wesentlich schneller als die sequentielle Suche in einem unsortierten dynamischen Feld.
- Aber: Einfach verkettete Listen erlauben keine binäre Suche (denken Sie selbst darüber nach, warum das so ist – Lösung auf Seite 34).

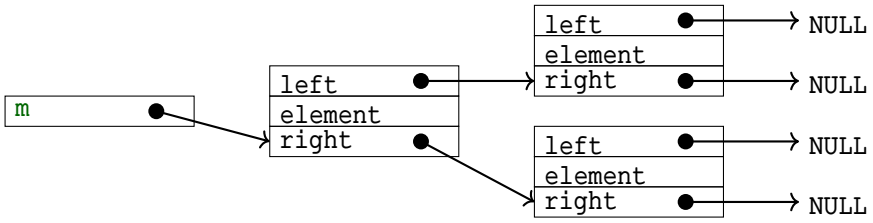
Die schnellsten Verwaltungsfunktionen zur Verwaltung einer sortierten Liste haben **Suchbäume**.

15.6.3 Weitere Datenstrukturen: Binäre Suchbäume

Binäre Suchbäume setzen eine sortierte dynamische Liste um. Sie kombinieren dabei die Systematik einer einfach verketteten Liste mit der Effizienz der binären Suche. Jeder Knoten hat dabei zwei Nachfolger: Einen linken (`left`) und einen rechten (`right`):

```
1 typedef struct _node {
2     struct _node *right;
3     int element;
4     struct _node *left;
5 } node;
6 typedef node* tree;
```

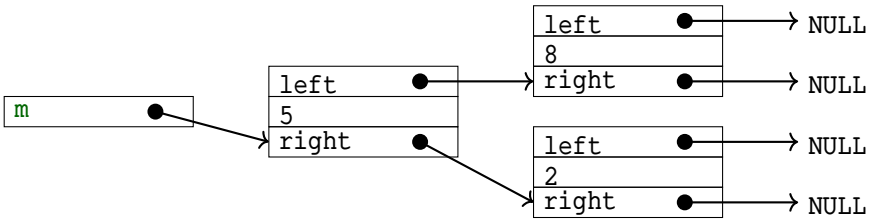
Allgemeine Übersicht für einen Baum mit 3 Knoten:



Die Sortierung erfolgt über eine kluge Ausnutzung der linken und rechten Teilbäume. Generell gilt, dass in den linken Teilbäumen größere Zahlen und im rechten Teilbaum kleinere Zahlen als in der Wurzel vorkommen. Genauere Details dazu würden hier jedoch den Rahmen sprengen. In der Vorlesung *Informatik 3* wird der binäre Suchbaum jedoch ausführlich erklärt.

Beispiel:

Beispiel eines Suchbaumes zur **sortierten** Speicherung der Zahlen `5,2,8`: Gemäß der Sortierung befinden sich im **linken Teilbaum** größere Zahlen und im **rechten Teilbaum** kleinere Zahlen als in der Wurzel.



- In einem Suchbaum mit Höhe m (= maximale Länge eines Pfades von der Wurzel zu einem Blatt) haben bis zu $n = 2^{m+1} - 1$ Knoten Platz.
- Ein Suchalgorithmus muss nur den richtigen Pfad entlang laufen, benötigt also maximal $O(\log n)$ Schritte, falls der Suchbaum voll befüllt ist.

15.6.4 Verwaltung beliebiger Daten

Alle bisher vorgestellten Datenstrukturen können leicht verallgemeinert werden, um allgemeine Daten anstelle von ganzen Zahlen zu verwalten:

- Statt der `int`-Komponente verwendet man einen Zeiger auf die gewünschten Daten(strukturen).

Beispiel: Verwaltung einer dynamischen Liste von Adressen als einfach verkettete Liste

```
1 typedef struct _node {
2     ADDRESS *element;
3     struct _node *next;
4 } node;
5 typedef node* list;
```

Warum funktioniert die binäre Suche nicht in einer einfach verketteten Liste?

Die binäre Suche (siehe Kapitel 11) halbiert immer wieder den Suchbereich, bis man bei der gewünschten Zahl/Position angekommen ist. Einfach verkettete Listen bieten aber keine Möglichkeit, effizient auf die Mitte zuzugreifen. Man müsste also bei jeder Iteration einen zusätzlichen

Aufwand im Umfang von $O(n)$ berücksichtigen, um den Knoten in der Mitte zu suchen. Die binäre Suche müsste also in der einfach verketteten Liste mehrfach sequentiell suchen, wohingegen die sequentielle Suche nur einmalig suchen muss.