# Vorlesung Informatik 1 (Wintersemester 2020/2021)

## Kapitel 15: Dynamische Datenstrukturen

Martin Frieb
Johannes Metzger

Universität Augsburg
Fakultät für Angewandte Informatik

03. Februar 2021

# Was ist eine dynamische Datenstruktur?

## Statische Datenstruktur

- Speicherbedarf steht schon zur Übersetzungszeit fest.
- Die Speicherverwaltung (Reservieren und Freigeben von Speicher) erfolgt automatisch.
- Beispiel: `char v[SIZE];`

## Dynamische Datenstruktur

- Speicherbedarf steht erst zur Laufzeit fest und kann sich zur Laufzeit ändern.
- Die Speicherverwaltung (Reservieren und Freigeben von Speicher) erfolgt dynamisch durch explizite Anweisungen im Programm.
- Beispiel:

```
char *v = malloc(SIZE * sizeof(char));
```

# 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**: Verschiedene komplexe dynamische Datenstrukturen zur Verwaltung von Listen ganzer Zahlen

# Was ist ein dynamisches Feld?

Ein **dynamisches Feld** zur Verwaltung einer unsortierten Liste ganzer Zahlen mit Verwaltungsoperationen:

```
#ifndef ARRAYLIST_H_INCLUDED
#define ARRAYLIST_H_INCLUDED

typedef struct _arraylist {
  int *elements;
  int size;
} arraylist;

arraylist *arraylist_create(void);
void arraylist_destroy(arraylist *m);
int arraylist_size(const arraylist *m);
void arraylist_print(const arraylist *m);
int arraylist_isempty(const arraylist *m);
int arraylist_insert(arraylist *m, int n);
int arraylist_delete(arraylist *m, int n);
int arraylist_iselem(const arraylist *m, int n);

#endif
```

# Dynamische Felder im Arbeitsspeicher

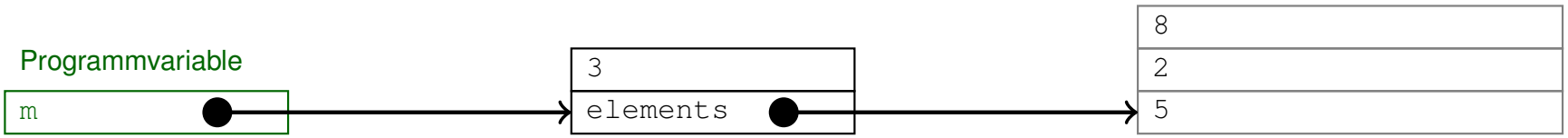
Datenstrukturinvarianten:

- Der Wert von `size` entspricht der Anzahl der Komponenten des dynamischen Feldes `elements`

```
typedef struct _arraylist {
  int *elements;
  int size;
} arraylist;
```

Allgemeine Übersicht:


Darstellung der Liste `5,2,8`:

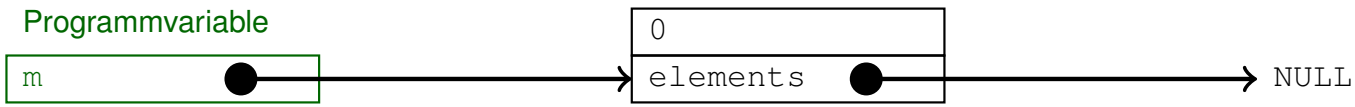


# Verwaltungsoperationen: Leere Liste anlegen

Erzeugung einer leeren Liste ohne Elemente (beachte die Einhaltung der Datenstrukturinvariante)

```
arraylist *arraylist_create(void)
{
  arraylist *m = malloc(sizeof(arraylist));
  if(m == NULL)
    return NULL;
  m->elements = NULL;
  m->size = 0;
  return m;
}
```

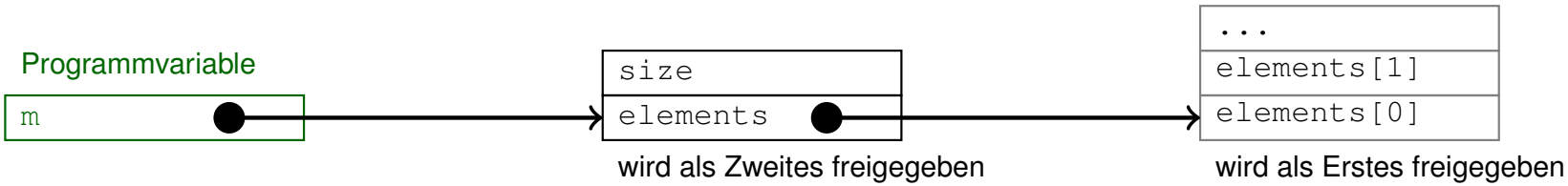
- `m->elements`: Zugriff auf die Komponente `elements` der Liste
- `m->size`: Zugriff auf die Komponente `size` der Liste



# Verwaltungsoperationen: Liste freigeben

Freigabe des Speicherbereichs erfolgt rückwärts, damit keine Speicherlecks entstehen:

```
void arraylist_destroy(arraylist *m)
{
  if(!arraylist_isempty(m))
    free(m->elements);
  free(m);
}
```



# Verwaltungsoperationen: Test auf Elementzugehörigkeit

Test auf Element-Zugehörigkeit mit sequentieller Suche:

```
int arraylist_iselem(const arraylist *m, int n)
{
  int k;
  for (k = 0; k < m->size; ++k) {
    if ((m->elements)[k] == n)
      return 1;
  }
  return 0;
}
```

- `(m->elements)[k]`: Zugriff auf die `k`-te Feldkomponente von `elements`
- Zeitkomplexität: linear in der Länge der Liste.

# Verwaltungsoperationen: Element einfügen

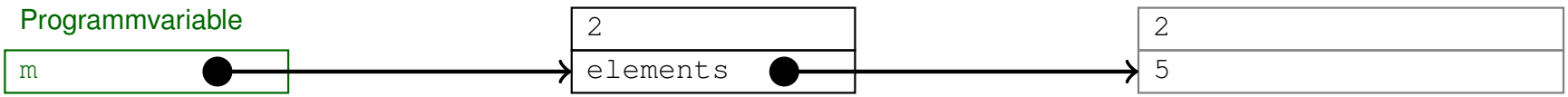
Neues Element an letzter Stelle einfügen:

```
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 }
```

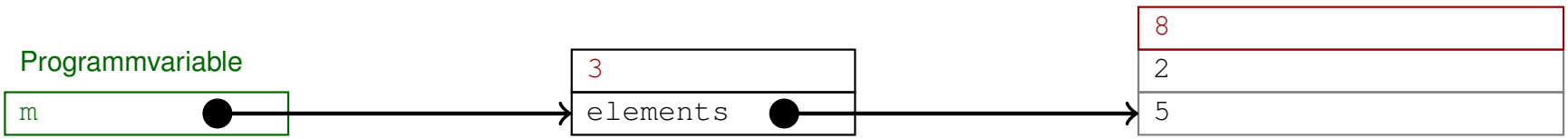
- Datenstrukturinvariante: Anpassung von `size` nach erfolgreichem Einfügen (Zeile 12)
- Zeitkomplexität: konstant.

# Verwaltungsoperationen: Element 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 }
```




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



# Verwaltungsoperationen: Element löschen

Grundidee:

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

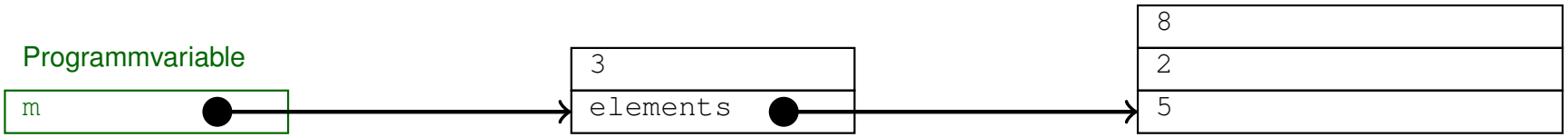
1 Suche erstes $k$ mit $x_k = x$;

2 **wenn** *$x$ kommt nicht in $x_1, \ldots, x_n$ vor* **dann**
  **Ausgabe :** 1

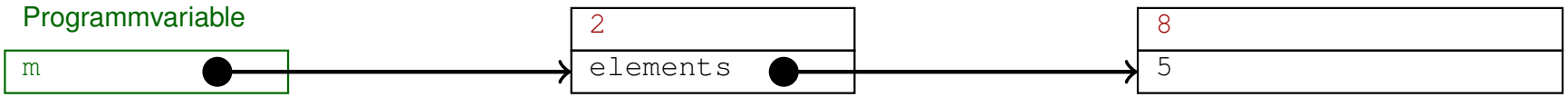
3 Bewege alle Elemente nach $x_k$ um eine Position nach vorne;

**Ausgabe :** 0

Zeitkomplexität: linear in der Länge der Liste.




Wert 2 löschen (Aufruf `status = arraylist_delete(m, 2)`):

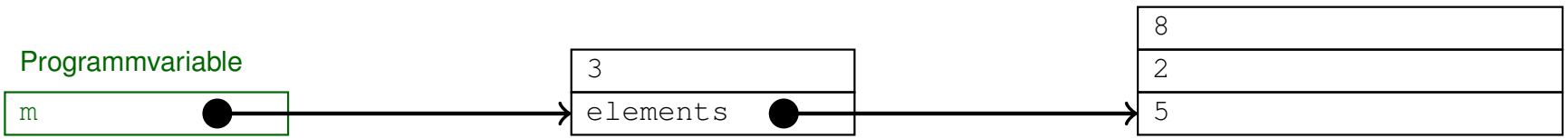


# Verwaltungsoperationen: Element löschen

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

Implementierung in C:

- Suche erstes `k` mit `(m->elements)[k] = n`.
- Wenn `n` nicht in `(m->elements)[1],...,(m->elements)[n]` vorkommt, gibt `1` aus.
- Bewege alle Elemente nach `(m->elements)[k]` um eine Position vor.
- Verringere den Speicherplatz von `m->elements` (auch dabei kann es zu einem Speicherfehler kommen!).
  Falls das nicht klappt: Stelle alte Liste wieder her und gib `2` aus.
  Falls es klappt: Verringere `m->size` um `1` (Datenstrukturinvariante).

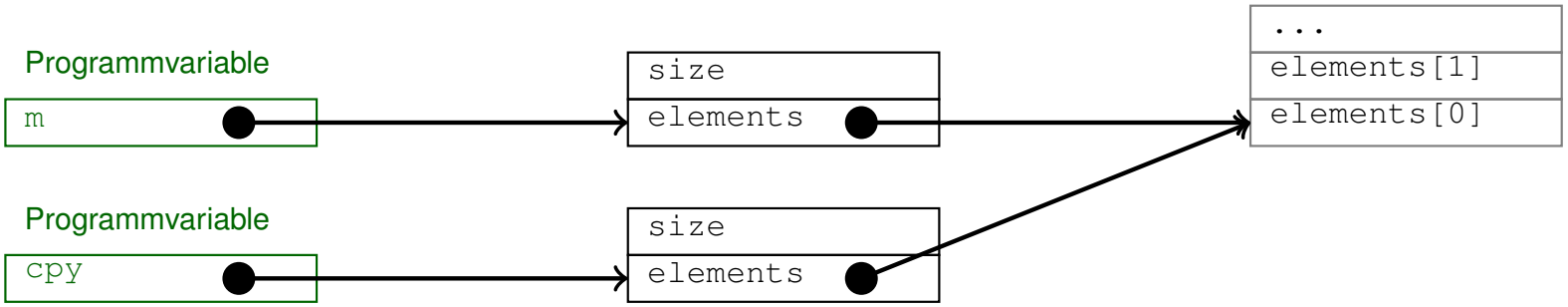


# Verwaltungsoperationen: Element löschen

```
1  int arraylist_delete(arraylist *m, int n)
2  {
3    int *neu = NULL;
4    int k = 0, i;
5    while (k < m->size) {
6      if ((m->elements)[k] == n) break;
7      ++k;
8    }
9    if (k == m->size) return 1;
10   for (i = k; i < m->size - 1; ++i)
11     (m->elements)[i] = (m->elements)[i + 1];
12   neu = realloc(m->elements, (m->size - 1) * sizeof(int));
13   if (neu == NULL && m->size != 1) {
14     for (i = m->size - 2; i > k; --i)
15       (m->elements)[i] = (m->elements)[i - 1];
16     (m->elements)[k] = n;
17     return 2;
18   }
19   m->elements = neu;
20   --(m->size);
21   return 0;
22 }
```

# Kopien anlegen

Ist es sinnvoll eine Kopie der folgenden Form anzulegen?




Überlege: Kann die Manipulation der Kopie eine Datenstrukturinvariante beim Original verletzten (oder umgekehrt)?

# Hauptprogramm: Beispiel

```
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 }
```

- Fehlerbehandlungen nicht vergessen (Zeilen 7 und 9)
- Speicherfreigabe nicht vergessen bei Fehler (Zeile 9) und am Ende (Zeile 12)
- Verwende ausschließlich die Verwaltungsoperationen ohne Benutzung der Implementierung der Liste (also ohne direkt auf die Komponenten der Datenstruktur zuzugreifen)

# Bewertung

Weitere mögliche Datenstrukturinvarianten (Achtung: Implementierung der Verwaltungsoperationen ändert sich!)

- Liste wiederholungsfrei: zur Verwaltung von Mengen.
- Liste sortiert.

Löschen eines Wertes:

- Verschiedene Strategien, falls ein Wert mehrmals vorkommt: erstes Vorkommen löschen / alle Vorkommen löschen

Wieso fügt man neue Werte nicht zu Beginn ein?

- Unter Beibehaltung des reservierten Speicherbereichs müssten alle anderen Werte nach hinten geschoben werden

# Was ist eine einfach verkettete Liste?

**Einfach verkettete Liste** zur Verwaltung einer unsortierten Liste ganzer Zahlen:

```
#ifndef EVL_H_INCLUDED
#define EVL_H_INCLUDED

#define EMPTY_LIST NULL

typedef struct _node {
  int element;
  struct _node *next;
} node;
typedef node *list;

void list_destroy(list m);
int list_size(list m);
void list_print(const list m);
int list_isempty(const list m);
list list_insert(list m, int n);
list list_delete(list m, int n);
int list_iselem(list m, int n);

#endif
```

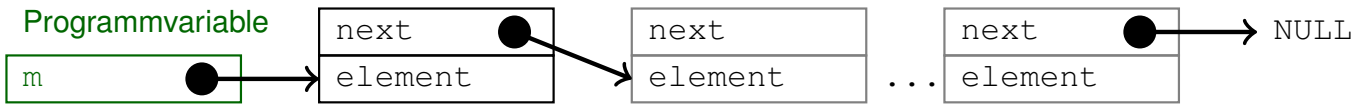
# Einfach verkettete Listen im Arbeitsspeicher

Datenstrukturinvarianten:

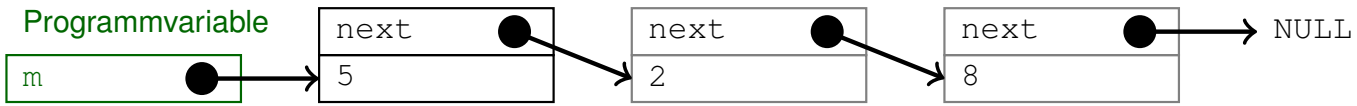
- Es gibt genau einen (ersten) Knoten ohne Vorgänger
- Es gibt genau einen (letzten) Knoten ohne Nachfolger
- Alle anderen Knoten haben jeweils genau einen Vorgänger und Nachfolger

```
typedef struct _node {
  int element;
  struct _node *next;
} node;
typedef node *list;
```

Allgemeine Übersicht für Programmvariable `list m`:

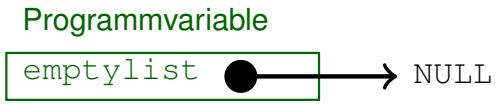



Darstellung der Liste `5,2,8`:



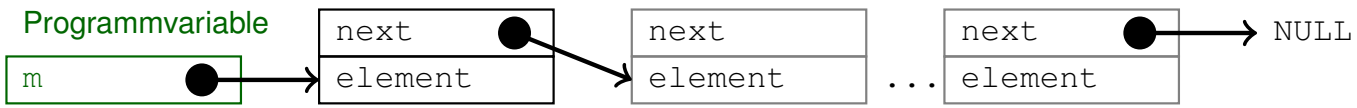
# Verwaltungsoperationen: Leere Liste erstellen und Liste freigeben

**Leere Liste** erstellen: `list m = EMPTY_LIST;`




**Rekursive Freigabe des Speicherbereichs**: Falls die Liste `m` nicht leer ist, wird zuerst rekursiv die Liste `m->next` freigegeben mit `list_destroy(m->next)`, und dann das Element, auf das `m` zeigt, mit `free(m)`.

```
void list_destroy(list m) {
  if (m != EMPTY_LIST) {
    list_destroy(m->next);
    free(m);
  }
}
```

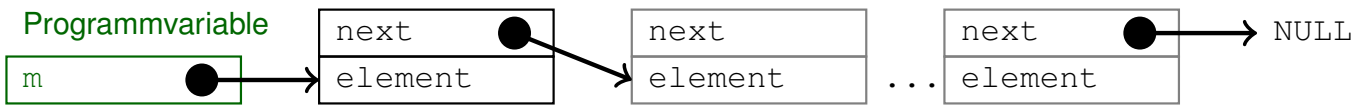


# Verwaltungsoperationen: Test auf Elementzugehörigkeit

Überprüfe Wert eines Knotens und gehe über den zugehörigen Zeiger zum nächsten Knoten, solange das Element nicht gefunden wurde. Man muss sequentiell Knoten nach Knoten durchlaufen (**kein direkter** Zugriff auf einzelne Knoten)

```
int list_iselem(list m, int n) {
  while (m != EMPTY_LIST && n != m->element) {
    m = m->next;
  }
  return (m != EMPTY_LIST && n == m->element);
}
```

- `m->element`: Zugriff auf den im Knoten gespeicherten Wert
- `m->next`: Zugriff auf den nächsten Knoten in der Liste
- Zeitkomplexität: linear in der Länge der Liste.

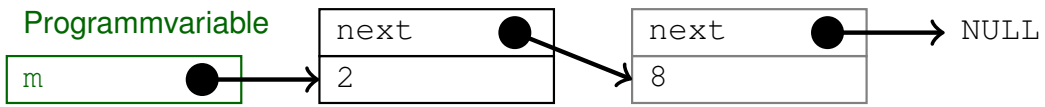


# Verwaltungsoperationen: Element einfügen

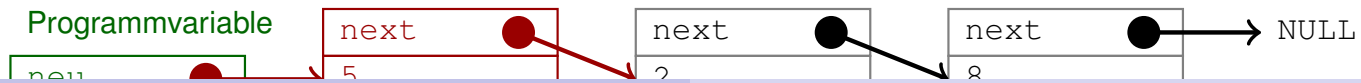
**Neues Element an erster Stelle einfügen**:

```
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  }
```

- Speicherbereich für neuen Knoten wird dynamisch neu erzeugt (Zeile 2)
- Fehlerfall: Rückgabe `NULL` (Zeile 4, alte Liste kann weiter benutzt werden)
- Erfolgsfall: Rückgabe der Adresse der neuen Liste (Zeile 7)
- Zeitkomplexität: konstant.




Neuen Wert `5` einfügen (Aufruf: `neu = list_insert(m, 5)`):



# Verwaltungsoperationen: Element löschen (rekursiv)

```
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 }
```

- Falls erstes Element **ungleich** `n` (Zeile 4): Rufe `list_delete` rekursiv für nächstes Element `m->next` auf (Z. 5), verbinde erstes Element mit der neuen Liste, die dabei zurückgegeben wird (Z. 6), gib erstellte Liste zurück (7)
- Falls erstes Element **gleich** `n` (ab Zeile 8): Gibt dieses Element frei (Zeile 10), gib nachfolgende Liste zurück (Zeile 11)
- Zeitkomplexität: linear in der Länge der Liste.

# Verwaltungsoperationen: Element löschen (rekursiv)

```
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 }
```


Wert `2` löschen (Aufruf: `neu = list_delete(m, 2)`):

# Hauptprogramm: Beispiel

```
1  #include "evl.h"
2  #include <stdio.h>
3  #include <stdlib.h>

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 }
```

- Fehlerbehandlungen nicht vergessen (Zeile 8)
- Speicherfreigabe nicht vergessen bei Fehler (Zeile 8) und am Ende (Zeile 12)
- Beachte, dass überall wo `list` steht, ein Zeiger auf eine einfach verkettete Liste gemeint ist (vgl. **`typedef`** in Definition von `list`)

# Bewertung

Weitere mögliche Datenstrukturinvarianten (Achtung: Implementierung der Verwaltungsoperationen ändert sich!)

- Liste sortiert.

Vergleich mit dynamischen Feldern:

- Einfach verkettete Listen und dynamische Felder sind alternative Implementierungen zur Verwaltung dynamischer Listen und Mengen
- Einfügen, Löschen: geht bei einfach verketteten Listen schneller
- Ausgabe, Elementtest: geht bei dynamischen Feldern schneller

Wieso fügt man neue Werte nicht am Ende ein?

- Man müsste alle Element zuvor durchlaufen, um das bisher letzte Element zu finden

# Was ist ein Stack?

Ein **Stack** ist eine Datenstruktur für dynamische Listen, die nur einen eingeschränkten Zugriff auf die Listenelemente erlaubt:

- Er realisiert einen Datenzugriff nach dem **LIFO-Prinzip** (Last-in-First-out)
- Auf die Listenelemente kann nur in umgekehrter Reihenfolge, in der sie im Stack abgelegt wurden, zugegriffen werden.
- `push`: ein neues Element an oberster Stelle einfügen
- `pop`: oberstes Element lesen / zurückgeben und löschen
- `top`: oberstes Element lesen / zurückgeben (und nicht löschen)

Ein Stack kann intern als einfach verkettete Liste oder als dynamisches Feld realisiert werden (wir verwenden dazu im Folgenden eine einfach verkettete Liste)

# Was ist ein Stack?

Ein **Stack (Kellerspeicher)** zur Verwaltung einer unsortierten Liste ganzer Zahlen:

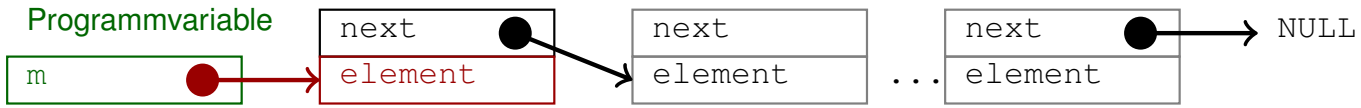
```
#ifndef STACK_H_INCLUDED
#define STACK_H_INCLUDED

#define EMPTY_STACK NULL

typedef struct _node
{
  int element;
  struct _node *next;
} node;
typedef node* stack;

void stack_destroy(stack m);
int stack_isempty(stack m);
int stack_push(stack *m, int n);
int stack_pop(stack *m);
int stack_top(stack m);

#endif
```

# Verwaltungsoperationen: Element lesen (top)

**Oberstes Element lesen** (`top`):

```
int stack_top(stack m)
{
  if (m == EMPTY_STACK) {
    printf("\nstack_top called on empty stack...");
    return 0;
  }
  return m->element;
}
```

- Nur auf das erste Element der Liste kann direkt zugegriffen werden
- **Aufruf** für Programmvariable `stack m`:
  `int n = stack_top(m);`



# Verwaltungsoperationen: Element einfügen (push)

**Neues Element an oberster Stelle einfügen** (`push`):

```
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 }
```

**Unterschiede** zur Implementierung von `insert` für einfach verkettete Listen:

- Adresse des erweiterten Stacks wird nicht zurückgegeben, sondern an der Stelle `*m` gespeichert (Call-by-Reference-Prinzip)
- Rückgabewert unterscheidet allein zwischen Erfolgs- und Fehlerfall

**Aufruf** für eine Stack-Variable `m`:

```
int status = stack_push(&m, 5);
```

# Verwaltungsoperationen: Element löschen

**Oberstes Element lesen und löschen** (`pop`):

```
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  }
```

**Unterschiede** zur Implementierung von `delete` für einfach verkettete Listen:

- Adresse des reduzierten Stacks wird nicht zurückgegeben, sondern an der Stelle `*m` gespeichert (Call-by-Reference-Prinzip)
- Zurückgegeben wird der oberste Wert
- Es kann nur das oberste Element gelöscht werden (keines weiter unten im Stack)

**Aufruf** für eine Stack-Variable `m`: `int n = stack_pop(&m);`

# 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 }
```

# Stacks: Fazit

- Stacks bieten nur eingeschränkten Zugriff auf eine Datenstruktur
- Es kann immer nur auf das zuletzt hinzugefügte Element zugegriffen werden (LIFO-Prinzip)
- Anwendung z.B. Realisierung einer *Rückgängig*-Funktionalität
- Implementierung z.B. mit dynamischen Feldern oder einfach verketteten Listen möglich
- Operationen `push`, `pop` und `top` in $O(1)$ realisierbar

# Was ist eine Queue?

Eine **Queue (Warteschlange)** ist eine Datenstruktur für dynamische Listen, die nur einen eingeschränkten Zugriff auf die Listenelemente erlaubt:

- Sie realisiert einen Datenzugriff nach dem **FIFO-Prinzip** (First-in-First-out)
- Auf die Listenelemente kann nur in derselben Reihenfolge, in der sie in der Queue abgelegt wurden, zugegriffen werden.
- `enter`: ein neues Element an letzter Stelle einfügen
- `remove`: erstes Element lesen und löschen
- `first`: erstes Element lesen (und nicht löschen)

Eine Warteschlange kann intern als einfach verkettete Liste oder als dynamisches Feld realisiert werden (wir verwenden dazu im Folgenden eine einfach verkettete Liste)

# Was ist eine Queue?

Eine **Queue (Warteschlange)** zur Verwaltung einer unsortierten Liste von Zahlen:

```
#ifndef QUEUE_H_INCLUDED
#define QUEUE_H_INCLUDED

#include "evl.h"

#define EMPTY_QUEUE NULL

typedef struct _queue {
  list first;
  list last;
} * queue;

queue queue_create(void);
void queue_destroy(queue m);
int queue_isempty(queue m);
int queue_enter(queue m, int n);
int queue_remove(queue m, int *success);
int queue_first(queue m, int *success);

#endif
```

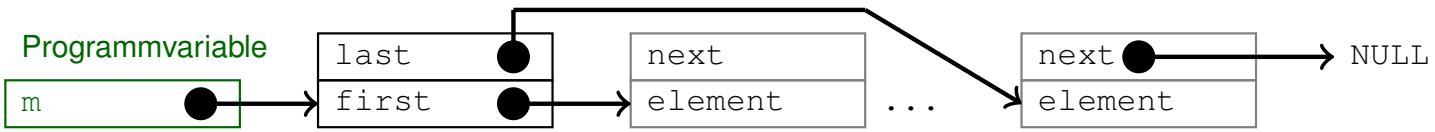
# Queues im Arbeitsspeicher

Datenstrukturinvarianten:

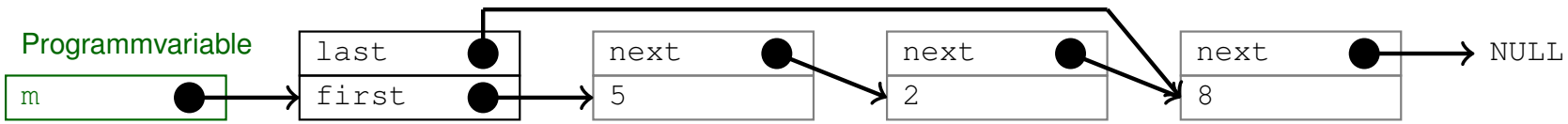
- `first` zeigt auf das erste Element der Liste
- `last` zeigt auf das letzte Element der Liste

```
typedef struct _queue {
  list first;
  list last;
} * queue;
```

Allgemeine Übersicht für eine Programmvariable `queue m`:




Darstellung der Liste `5,2,8`:

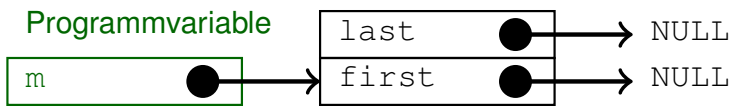


# Verwaltungsoperationen: Queue erzeugen

Erzeugung einer leeren Liste ohne Elemente:

```
queue queue_create(void)
{
  queue q = malloc(sizeof(struct _queue));
  if (q == NULL)
    return NULL;
  q->first = EMPTY_QUEUE;
  q->last = EMPTY_QUEUE;
  return q;
}
```

Die Komponenten `first` und `last` werden als Zeiger auf `NULL` angelegt.

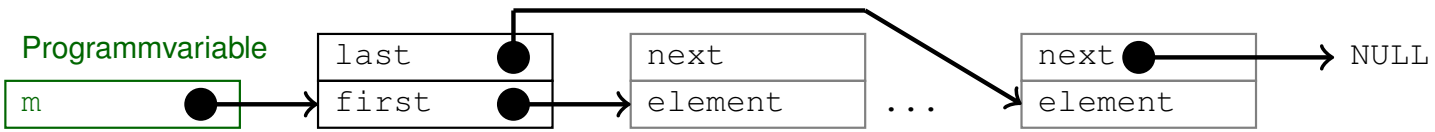


# Verwaltungsoperationen: Queue freigeben

Eine Liste freigeben:

```
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  }
```

- Zuerst wird die Liste `m->first` freigegeben (Zeile 4); Benutze dazu die `destroy`-Operation für einfach verkettete Listen.
- Dann wird die Queue `m` freigegeben (Zeile 5).

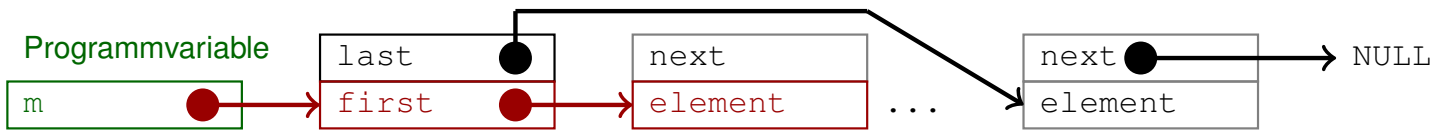


# Verwaltungsoperationen: Element lesen

**Erstes Element lesen** (`queue_first`):

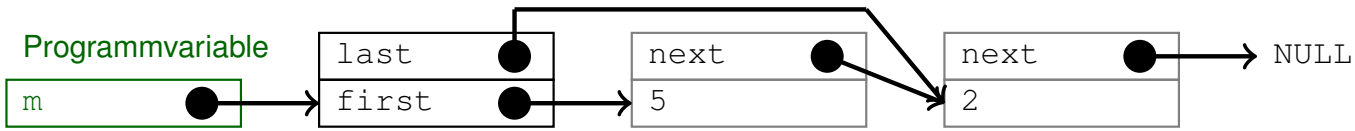
```
int queue_first(queue m, int *success)
{
  if (queue_isempty(m)) {
    if (success != NULL)
      *success = 0;
    return 0;
  }
  if (success != NULL)
    *success = 1;
  return (m->first)->element;
}
```

- Zugriff auf das erste Element der Liste über die Komponente `first`.
- Überprüfung, ob Queue leer ist. Falls für `success` etwas anderes als `NULL` übergeben wurde entsprechende Rückmeldung.
- **Aufruf** für eine Queue `m`: `int n = queue_first(m, &status);`

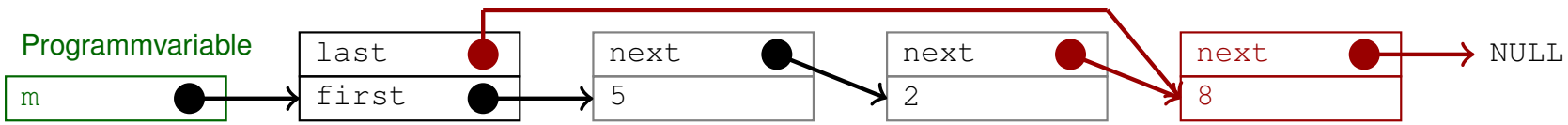


# Verwaltungsoperationen: Element einfügen

```
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;
```




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



# Verwaltungsoperationen: Element löschen

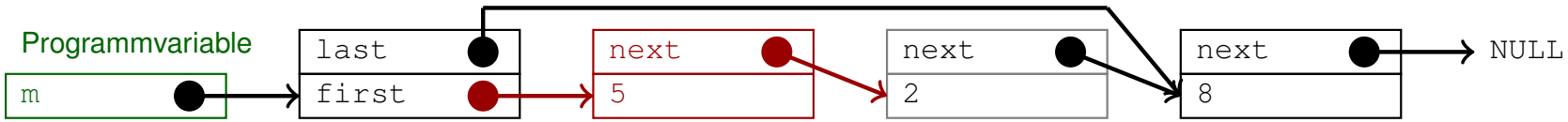
**Erstes Element lesen und löschen** (`queue_remove`):

```
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 }
```

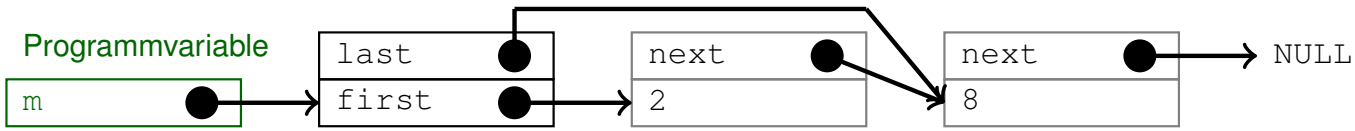
# Verwaltungsoperationen: Element löschen

- Implementierung analog zu `pop` bei einem Stack
- Leere Queue wird in `queue_first` behandelt
- Falls `sucess != NULL`, wird Status darin gespeichert

Welches Element wird gelöscht?




Zustand nach Aufruf `int n = queue_remove(m, &status);`:




Erster Wert 5 wurde zu `n` gelesen und gelöscht

# Hauptprogramm: Beispiel

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

# Queues: Fazit

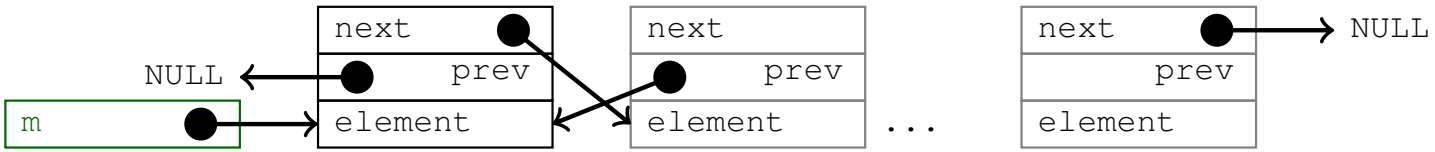
- Queues bieten nur eingeschränkten Zugriff auf eine Datenstruktur
- Es kann immer nur auf das älteste Element zugegriffen werden (FIFO-Prinzip)
- Anwendung alles, wo etwas nacheinander verarbeitet werden soll
- Zur Implementierung eignen sich v.a. einfach verkettete Listen
- Operationen `enter`, `remove` und `first` in $O(1)$ realisierbar

# Weitere Datenstrukturen: Doppelt verkettete Listen

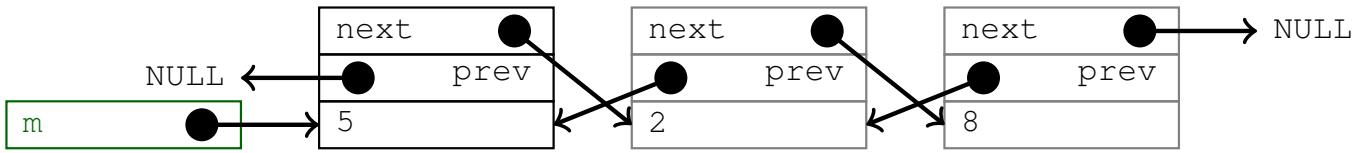
Jedes Element ist mit seinem Nachfolger **und Vorgänger** verbunden:

```
typedef struct _node {
  int element;
  struct _node *prev;
  struct _node *next;
} node;
typedef node* list;
```

Allgemeine Übersicht:




Darstellung der Liste `5,2,8`:



# 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.
- 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 (warum?)

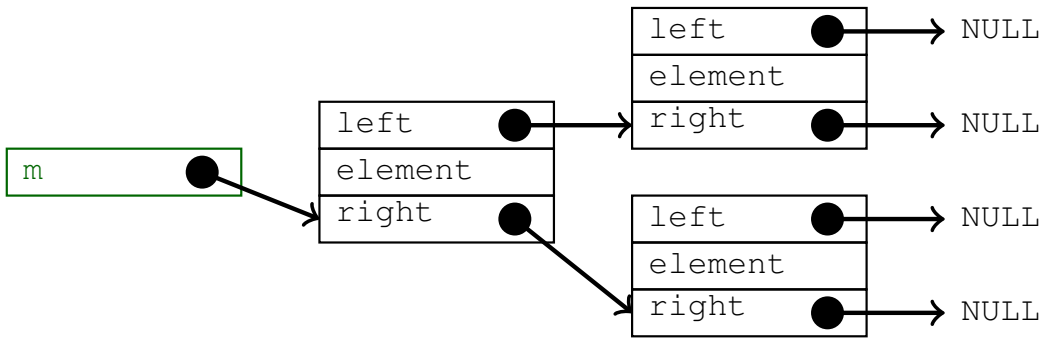
Die schnellsten Verwaltungsfunktionen zur Verwaltung einer sortierten Liste haben **Suchbäume** (siehe Informatik 3)

# Weitere Datenstrukturen: Binäre Suchbäume

Jeder Knoten des Baums hat zwei Kindknoten `left` und `right`:

```
typedef struct _node {
  struct _node *right;
  int element;
  struct _node *left;
} node;
typedef node* tree;
```

Allgemeine Übersicht für einen Baum mit 3 Knoten:

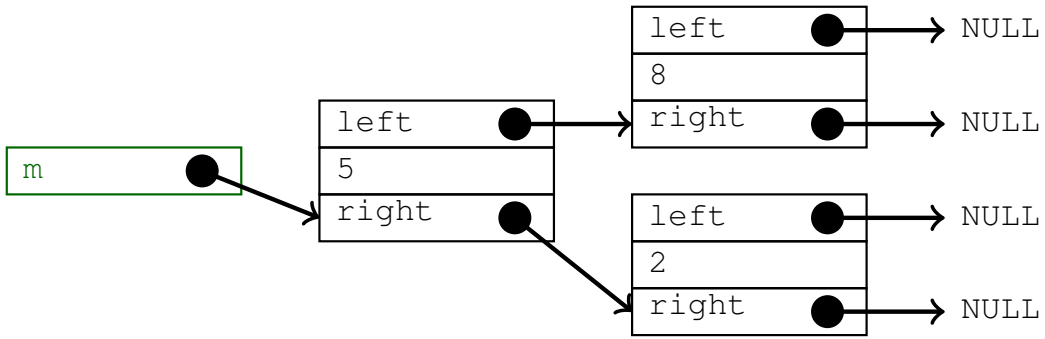


# Weitere Datenstrukturen: Binäre Suchbäume

Jeder Knoten des Baums hat zwei Kindknoten `left` und `right`:

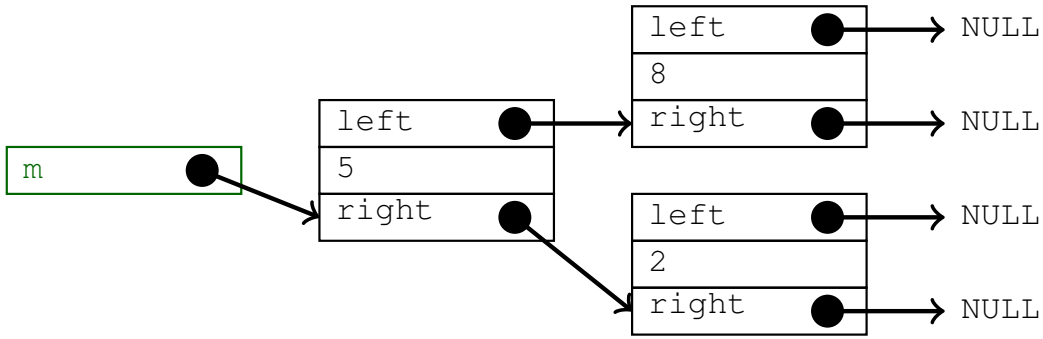
```
typedef struct _node {
  struct _node *right;
  int element;
  struct _node *left;
} node;
typedef node* tree;
```

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.



# Weitere Datenstrukturen: Binäre Suchbäume

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.

# 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 15.1 (Verwaltung einer dynamischen Liste von Adressen als einfach verkettete Liste)**

```
typedef struct _node {
  ADDRESS *element;
  struct _node *next;
} node;
typedef node* list;
```

# Ende

**Viel Erfolg bei den Prüfungen!**

wünscht das Team der Informatik 1

Dr. Martin Frieb
Johannes Metzger
Marius Brendle
und alle Tutoren