Algorithmen und Datenstrukturen SS 2008

Clemens Ballarin

Inhaltsverzeichnis

Kapitel 1

Einführung

1.1 Algorithmenbegriff

Algorithmus (Merriam-Webster)

- Verfahren, um ein mathematisches Problem (z.B. Finden des größten gemeinsamen Teilers) in einer endlichen Zahl von Schritten zu lösen, das häufig die Wiederholung einer Operation beinhaltet.
- Allgemein: ein schrittweises Verfahren um ein Problem zu lösen oder ein Ziel zu erreichen.

Algorithmenentwurf altes Studiengebiet. Bsp. für Verfahren und deren Verbesserung:

- Feuer machen
- Pyramiden bauen
- Briefe sortieren

Entwurf von Computeralgorithmen neu.

Wodurch unterscheiden sich Computeralgorithmen von herkömmlichen Algorithmen?

- Computeranweisungen sind genau definierte, beschränkte primitive Operationen.
- Übersetzung von natürlicher Sprache in Computersprache ist schwierig (Programmierung).

- Problemgröße.
  Große Datenmengen erfordern andere Algorithmen als kleine. Umgang mit großen Datenmengen ist unintuitiv.

- Aufwand zum Entwurf des Algorithmus vs. Aufwand zum Lösen des Problems.

  Bsp.: Rasen mähen, Auspacken der Einkaufstüte, ...

1.2 Längste aufsteigende Teilsequenz

Definition 1 (Teilsequenz) *Gegeben ist eine Sequenz $x_1, x_2, \ldots, x_n$ von Zahlen. Durch Entfernen einiger Einträge erhält man eine* Teilsequenz.

Problem:

Bestimme die Länge der längsten aufsteigenden ($\leq$) Teilsequenz (LAT).

Bsp.:

```
1   3   4   6   2   4
```

eine Teilsequenz: `1 3 2`
keine Teilsequenz: `1 4 3`
eine aufsteigende Teilsequenz: `1 3 6`

Lösungsidee:

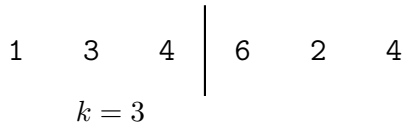
- Zähle alle Teilsequenzen auf.
- Bestimme Länge der längsten aufsteigenden Teilsequenz.

Alle 2^n Teilsequenzen müssen betrachtet werden! Das ist sehr langsam.

Verbesserung:

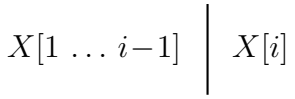
- Versuche, bereits gefundene LATs zu verlängern (Wiederverwenden von Zwischenergebnissen).
- Systematisches Vorgehen von links nach rechts.

1. Versuch: Merke Länge k der bisher gefundenen LAT.



Verlängert `6` die bisherige LAT?
Hierzu müssen wir den Endwert der LAT wissen!

2. Versuch: In Schritt i, merke Länge k einer LAT in $X[1 \ldots i-1]$ und m ist die kleinste Zahl, mit der eine solche LAT endet.

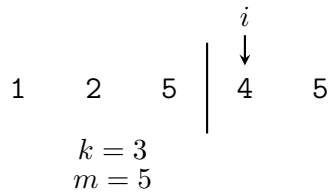


wenn X[i] $\geq$ m:

neue LAT der Länge $k+1$ in $X[1 \ldots i]$,
d. h. $k := k+1$; $m := X[i]$;

wenn X[i] $<$ m:

Bsp.:

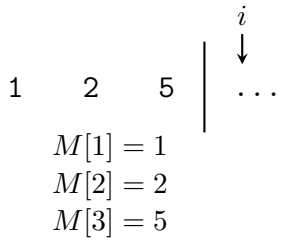


neue LAT der Länge k in $X[1 \ldots i]$,
aber mit niedrigerem Endwert `4`.

Benötigt: niedrigster Endwert von Sequenzen der Länge $k-1$ in $X[1 \ldots i-1]$.

3. Versuch: In Schritt i, sei k die Länge einer LAT in $X[1 \ldots i-1]$ und für alle j, $1 \leq j \leq k$ sei $M[j]$ der kleinste Endwert einer aufsteigenden Teilsequenz der Länge j in $X[1 \ldots i-1]$.

Bsp.:



wenn X[i] $\geq$ M[k]:

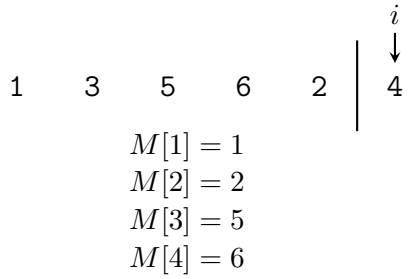
neue LAT, $k := k + 1$; $M[k] := X[i]$;

wenn X[i] $<$ M[1]:

neues Minimum, $M[1] := X[1]$;

wenn M[1] $\leq$ X[i] $<$ M[k]:

Bsp.:



neuer Endwert 4 für Teilsequenz der Länge 3.

NB: $M[1 \ldots k]$ ist sortiert.

finde j mit $M[j-1] \leq X[i] < M[j]$;
$M[j] := X[i]$;

Algorithmus 1 Längste Aufsteigende Teilsequenz

Eingabe: $X[1 \ldots n]$ (Feld von ganzen Zahlen, $n \geq 1$)
Ausgabe: Länge k der längsten aufsteigenden Teilsequenz

```
1: begin
2:   k := 1;
3:   M[1] := X[1];
4:   for i := 2 to n do
5:     if X[i] ≥ M[k] then
6:       k := k + 1;
7:       M[k] := X[i];
8:     else if X[i] < M[1] then
9:       M[1] := X[i];
10:    else
11:      bestimme j mit M[j−1] ≤ X[i] < M[j];
12:      M[j] := X[i];
13: end
```

Bsp.:

	1	4	7	3	5	6
k	1	2	3	3	3	4
$M[1]$	1	1	1	1	1	1
$M[2]$		4	4	3	3	3
$M[3]$			7	7	5	5
$M[4]$						6

Länge der LAT ist 4.

Anzahl der Rechenschritte (Zeitaufwand) proportional n^2.

1.3 EBNF-Notation

⟨`xyz`⟩	syntaktische Kategorie `xyz` (Nichtterminal)
::=	Definition einer syntaktischen Kategorie
|	Alternative
(...)	Klammerung
[...]	optional (null- oder ein-malige Wiederholung)
(...)$^+$	ein- bis n-malige Wiederholung
(...)*	null- bis n-malige Wiederholung
'[', ']', etc.	bezeichnet [,] respektive.

1.4 Pseudocode

- Zur informellen Beschreibung von Algorithmen
- Nicht direkt ausführbar

Schema einer Algorithmenbeschreibung:

Kopf

Algorithmus 2 ⟨*name*⟩

Eingabe: ⟨Spezifikation der Eingabe⟩
Ausgabe: ⟨Spezifikation der Ausgabe, Beschreibung des Zusammenhangs zwischen Eingabe und Ausgabe⟩

Rumpf

```
1: begin
2:   ⟨statement⟩+
3: end
```

⟨`variable`⟩

Variablenname, beginnt mit einem Buchstaben, z. B. x, x_0, *counter*, *a_very_long_name*

⟨`expression`⟩

Ausdruck, der zu einem Wert ausgewertet wird. In Pseudocode wird mathematische Notation verwendet.

Bsp.: $x + 1$, $\sin x$, x^2, $x = y$, $x \leq y$, $x \geq 2 \wedge x \leq 4$, $x \in S$, $S \cup U$

⟨`statement`⟩

Es gibt folgende Anweisungen:

Zuweisung:

- ⟨`variable`⟩ := ⟨`expression`⟩;
- ⟨`variable`⟩‘[’⟨`expression`⟩‘]’ := ⟨`expression`⟩;

Kontrollfluss:

- if ⟨`expression`⟩ then
  ⟨`statement`⟩$^+$
  [else ⟨`statement`⟩$^+$]
- while ⟨`expression`⟩ do
  ⟨`statement`⟩$^+$
- for ⟨`variable`⟩ := ⟨`expression`⟩ to/downto ⟨`expression`⟩ do
  ⟨`statement`⟩$^+$

  Blockstruktur von Anweisungen wird durch Einrückung verdeutlicht. Darüber hinaus sind textuelle Beschreibungen von Anweisungen zugelassen.
- for ⟨`variable`⟩ in ⟨`expression`⟩ do
  ⟨`statement`⟩$^+$
  - iteriert über alle Elemente von ⟨`expression`⟩
  - Reihenfolge ist nicht festgelegt
  - Bsp.:

    for x in S do
    ⟨`statement`⟩

    iteriert über alle Elemente der Menge S.

Kapitel 2

Übersicht über die Programmiersprache C

2.1 Struktur eines C-Programms

```
#include ⟨stdio.h⟩
#define SUCCESS 0

int main()
{
   ⟨declaration⟩+
   ⟨statement⟩+

   return SUCCESS;
}
```

`#include ⟨file.h⟩`

bindet Informationen über Bibliotheksfunktionen ein.

`#define NAME ⟨text⟩`

Makro, Präprozessor ersetzt alle Vorkommen von `NAME` durch ⟨`text`⟩ (außer in Strings).

2.2 Elementare Typen

`char`	Zeichen, üblicherweise ein Byte
`int`	ganze Zahl, Größe abhängig von der Architektur
`float`	Fließpunktzahl einfacher Genauigkeit
`double`	Fließpunktzahl doppelter Genauigkeit

2.3 Deklarationen

syntaktische Kategorie ⟨`declaration`⟩

Alle Variablen müssen vor Gebrauch mit ihrem Typ deklariert werden.

Bsp.:

```
int    lower, upper, step;
char   c, line[1000];
int    i = 0;                    // mit Initialisierung
```

2.4 Konstanten

syntaktische Kategorie ⟨`constant`⟩

Bsp.:

`char`	‘A’, ‘\n’, etc.
`int`	0, -10, etc.
`float`	1.0f, 1e-3f (= 0.001f), etc.
`double`	1.0, 1e-3 (= 0.001)

2.5 Ausdrücke

syntaktische Kategorie ⟨`expression`⟩

⟨`expression`⟩ ::= ⟨`variable`⟩
| ⟨`constant`⟩
| ‘(’ ⟨`expression`⟩ ‘)’
| ⟨`prefix-op`⟩ ⟨`expression`⟩
| ⟨`expression`⟩ ⟨`postfix-op`⟩
| ⟨`expression`⟩ ⟨`bin-op`⟩ ⟨`expression`⟩

- Arithmetische Operationen $*$, /, %, +, $-$ (binär)
- Vergleichsoperationen $<$, $\leq$, $>$, $\geq$, $==$, $\neq$ (binär)
- Logische Operatoren

  ! (prefix)
  &&, || (binär)

Vergleichs- und logische Operatoren werten zu 1 für „wahr“ und zu 0 für „falsch“ aus.

- Increment- und Decrementoperatoren ++, $--$
  - n++, ++n: erhöht Wert von n um 1.
  - n++ liefert ursprünglichen Wert von n zurück (post-increment), ++n liefert erhöhten Wert zurück (pre-increment).
  - $--$ dito, aber erniedrigt Wert um 1.
- Zuweisungsoperator = (binär)
  - $n = \langle$`expression`$\rangle$
    - Weist n den Wert von $\langle$`expression`$\rangle$ zu.
    - Wertet selbst zu diesem Wert aus.
  - $m = n = \langle$`expression`$\rangle$
    - Weist m und n den Wert zu.

Präzedenz

!, ++, $--$	binden am stärksten
$*$, /, %	
+, $-$	
$<$, $\leq$, $>$, $\geq$	
$==$, $\neq$	
&&	
||	
=	binden am schwächsten

Alle Operationen sind *linksassoziativ*, d. h. $x + y + z$ steht für $(x + y) + z$.

Ausnahme: $x = y = z$ steht für $x = (y = z)$.

Auswertungsreihenfolge

- &&, ||: von links nach rechts; nur solange, bis das Ergebnis feststeht.
- andere Operationen:

  nicht festgelegt (abhängig vom Compiler)

  Bsp.:

  $$A[i] = i{+}{+}$$

  Wird der Feldindex vor oder nach dem Inkrement ausgewertet?

2.6 Statements

Zuweisung

- ⟨`variable`⟩ = ⟨`expression`⟩;
- ⟨`variable`⟩‘[’⟨`expression`⟩‘]’ = ⟨`expression`⟩;

Blockstruktur

$$\langle \texttt{statement} \rangle \quad ::= \quad \{ \langle \texttt{statement} \rangle^{+} \}$$

If

if ‘(’⟨`expression`⟩‘)’
  ⟨$\texttt{statement}_1$⟩
[else
  ⟨$\texttt{statement}_2$⟩]

- Wertet ⟨`expression`⟩ zu Wert $\neq 0$ aus, so wird ⟨$\texttt{statement}_1$⟩ ausgeführt, sonst ⟨$\texttt{statement}_2$⟩.
- Bei geschachtelten ifs: else bezieht sich auf das innerste!
  Falls nicht erwünscht, { ... } verwenden.

While

while ‘(’⟨`expression`⟩‘)’
  ⟨`statement`⟩

- Wertet ⟨`expression`⟩ aus. Falls $\neq 0$ wird ⟨`statement`⟩ ausgeführt. Das wird solange wiederholt, bis ⟨`expression`⟩ zu 0 auswertet.

For

- for '('⟨expression$_1$⟩; ⟨expression$_2$⟩; ⟨expression$_3$⟩')'
  ⟨statement⟩

  entspricht

  ⟨expression$_1$⟩;
  while '('⟨expression$_2$⟩')' {
  ⟨statement⟩
  ⟨expression$_3$⟩;
  }

- for $(i = 0; i < n; i{+}{+})$
  ⟨statement⟩

  entspricht

  for $i := 0$ to $n - 1$ do
  ⟨statement⟩

  in Pseudocode.

Beispielprogramm in C

```
/* Convert positive number to binary representation. */

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

#define LENGTH 1000  /* Maximal length of binary number */

/*
    Convert positive number to binary representation

    Input: reads number from standard input
    Output: prints binary representation to standard output
*/

int main()
{
    int n, k, i;
    int B[LENGTH];

    /*  Read number from standard input, store in n */

    if (scanf("%d", &n) == 1)
    {
        k = 0;
        while (n > 0 && k < LENGTH) {
            B[k] = n % 2;
            n = n / 2;
            k = k + 1;
        }
        if (k < LENGTH) {
            printf("Binary representation: ");
            for (i = k-1; i >= 0; i--)
                printf("%d", B[i]);
            printf("\n");
            return 0;
        } else {
            printf("Binary representation longer than %d bits.\n", LENGTH);
            return -1;
        }
    } else {
        printf("No valid number read.\n");
        return -1;
    }
}
```

2.7 Funktionen

Gründe für die Verwendung von Funktionen

- Zerlegen eines großen Codestücks in kleinere Codestücke (Struktur)
- Ermöglichen, bereits bestehenden Code zu verwenden. (Wiederverwendung)
- Verstecken Implementierungsdetails vor Teilen des Programms welche diese nicht benötigen. (Kapselung)

  → Klären den Gesamtzusammenhang

  → Erleichtern Modifikationen am Programm.

Funktionsaufruf

- Argumentausdrücke werden ausgewertet, Werte werden an Funktion übergeben (kopiert) – (*call by value*).
- Bei Funktionsaufruf als Ausdruck

  $$f(\ldots)$$

  wertet der Ausdruck zum Rückgabewert der Funktion aus.
- Bei Funktionsaufruf als Anweisung

  $$f(\ldots);$$

  wird der Rückgabewert ignoriert.
- In beiden Fällen finden evtl. *Seiteneffekte* statt.
- Eine Funktion kann sich selbst aufrufen (Rekursion).

Funktionsdefinition

- Im Kopf: Parameter und deren Typen, Typ des Rückgabewerts
- Parameter(variablen) können modifiziert werden
- Rückgabewert wird mit

`return` ⟨`expression`⟩;

ausgegeben. Hiermit wird auch die Funktion verlassen.

- Bei Funktionen ohne Ergebnis:
  - Typ des Rückgabewerts: `void`
  - Verlassen mit: `return`;

Bsp. einer Funktionsdefinition in C

```
// Berechnet x^y
double power(double x, int y)
{
    double z = 1.0;

    if (y < 0) {
        x = 1.0 / x;
        y = -y;
    }

    while (y != 0) {
        z = z * x;
        y--;
    }

    return z;
}
```

Bsp. für einen Funktionsaufruf

```
int main()
{
    double a, b;
    int i;

    scanf("%lf%d", &a, &i);
    b = power(a, i);
    printf("%f", b);
}
```

Funktionsdefinition vs. Funktionsdeklaration

Variable und Funktionen müssen vor Verwendung *deklariert* werden, z. B.

```
double power(double x, int y);
```

Die Deklaration legt Parameter- und Ergebnistypen fest. Die Definition kann später erfolgen.

Funktionen in Pseudocode

Funktionsdefinition:

function ⟨name⟩(⟨parameter-list⟩) : ⟨return type⟩

begin
  ⋮
  return ⟨expression⟩;
end

wobei Rückgabewert und Parameter typisiert.

⟨parameter-list⟩ ::= ⟨name⟩ : ⟨type⟩ [, ⟨parameter-list⟩]

Funktionsaufruf:

ist ein Ausdruck. Mathematische Notation, z. B. $f(x)$, $find\_max(A)$, etc.

Ohne Rückgabewert:

Schlüsselwort **procedure**
Aufruf ist eine Anweisung: **call** $sort(A)$.

Globale vs. lokale Variablen

Variablen können sowohl außerhalb, als auch innerhalb von Funktionen deklariert werden.

Außerhalb

- Heißen *global* (in C auch *extern*)
- Von allen (nachfolgenden) Funktionsdefinitionen aus sichtbar.
- Werden beim Starten des Programms initialisiert.

Innerhalb

- Heißen *lokal* (in C auch *automatisch*)
- Deklaration zu Beginn jedes Blocks $\{\ldots\}$ möglich.
- Werden bei Betreten des Blocks initialisiert, falls Initialwerte angegeben, sonst Anfangswert zufällig.
- Verdecken Variablen gleichen Namens, die weiter außen deklariert sind (global oder lokal).

2.8 Implementierung einer FIFO-Warteschlange (Queue)

Definition 2 (FIFO) *Datenstruktur, in der Elemente in der Reihenfolge entfernt werden, in der sie eingefügt werden.*

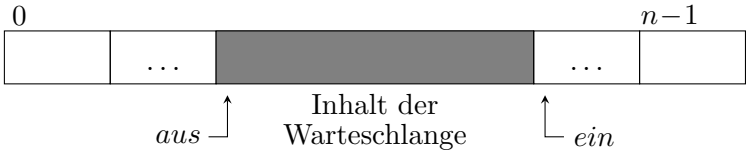
Operationen:

init:	löscht die Warteschlange
enqueue:	fügt neues Element ein
dequeue:	entfernt nächstes Element und gibt es zurück
empty:	testet, ob Datenstruktur leer
full:	testet, ob Datenstruktur voll

Implementierung als Ringpuffer:

Daten:

- A: `array`$[0 \ldots n-1]$
- *ein*: Position der nächsten Eingabe
- *aus*: Position der nächsten Ausgabe



Wenn $ein = aus$, dann ist die Warteschlange leer.
Wenn $ein > aus$, dann Inhalt von $aus, \ldots, ein - 1$.
Wenn $ein < aus$, dann Inhalt von $aus, \ldots, n$ und $1, \ldots, ein - 1$.

Algorithmus 3 INIT

Löscht die Warteschlange

```
1: begin
2:    ein := 0;
3:    aus := 0;
4: end
```

Algorithmus 4 ENQUEUE

Eingabe: x (Element, das in die Warteschlange eingefügt wird)

```
1: begin
2:    if ein = aus − 1 ∨ (ein = n − 1 ∧ aus = 0) then
3:       error „Warteschlange voll“;
4:    else
5:       A[ein] := x;
6:       ein := ein + 1;
7:       if ein ≥ n then
8:          ein := ein − n;
9: end
```

Algorithmus 5 DEQUEUE

Ausgabe: x (Element, das als erstes eingefügt wurde)

```
1: begin
2:    if ein = aus then
3:       error „Warteschlange leer“;
4:    else
5:       x := A[aus];
6:       aus := aus + 1;
7:       if aus > n then
8:          aus := aus − n;
9: end
```

Referenzübergabe – (*call by reference*)

Bei Parameterübergabe: Wert wird nicht kopiert, sondern Adresse der Variablen im Speicher wird übergeben.

- Modifikation des Parameters verändert das ursprüngliche Objekt
- Als Argumente sind nur Variablen, keine beliebigen Ausdrücke zugelassen
- Möglichkeit zur Rückgabe von Ergebnissen

In C:

Typmodifikator $*$:	⟨`type`⟩$*$ ist ein *Zeiger* auf ein Objekt (die *Adresse* eines Objekts) vom Typ ⟨`type`⟩.
Adressoperator &:	&⟨`expression`⟩ ist die Adresse von ⟨`expression`⟩.
Inhaltsoperator $*$:	$*$⟨`expression`⟩ ist das Objekt an Adresse ⟨`expression`⟩.

Bsp.:

Funktion *swap*, die die Werte zweier Variablen vertauscht.

```
void swap(int *px, int *py)
{
    int temp;

    temp = *px;
    *py = *px;
    *px = temp;
}
```

Aufruf: `swap(&x, &y);` vertauscht den Inhalt der Variablen x und y.

Funktionsdeklarationen für die FIFO

```
/** Globale Daten **/

int A[BUFSIZE];     // Ringpuffer
int aus, ein;


/** Operationen **/

/* Loeschen */
void init(void);

/* Test, ob Puffer leer */
/* Rueckgabe 1 falls leer, sonst 0 */
int empty(void);

/* Test, ob Puffer voll */
/* Rueckgabe 1 falls voll, sonst 0 */
int full(void);

/* Einfuegen */
/* Rueckgabe 1, falls Einfuegen von x erfolgreich, sonst 0 */
int enqueue(int x);

/* Entfernen */
/* Rueckgabe 1, falls Entfernen erfolgreich, sonst 0, */
/* Rueckgabewert in x */
int dequeue(int *x);
```

Kapitel 3

Datenstrukturen

3.1 Elemente

Element steht für nicht weiter spezifizierten Datentyp. z. B. Integer, Menge von Integers, Textdatei, etc. Wir machen folgende Annahmen über Elemente:

1. Man kann feststellen, ob zwei Elemente gleich sind.
2. Elemente gehören zu einer total (linear) geordneten Menge. Man kann feststellen, ob ein Element „kleiner“ als ein anderes ist.
3. Elemente können kopiert werden.

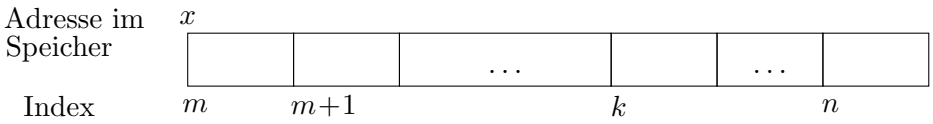
Diese Operationen verbrauchen jeweils eine Zeiteinheit. (In Wirklichkeit abhängig von der tatsächlichen Größe.)

3.2 Felder (Arrays)

Definition 3 (Feld)

- *Aneinanderreihung von Objekten*
- Größe *des Felds: Zahl der Objekte*
- *Speicherplatzbedarf: Größe* $\times$ *Speicherplatz pro Objekt*

Ein Feld belegt einen zusammenhängenden Block im Speicher.



Adresse des Objekts mit Index k im Speicher: $x + (k - m) \cdot d$ wobei d Speicherplatz pro Objekt.

Vorteile:

- Schneller Zugriff.

Nachteile:

- Nur Objekte gleichen Typs (gleicher Größe).
- Feldgröße nicht dynamisch veränderbar.

Notation in Pseudocode:

- Typ: `array`$[m \ldots n]$ of $\langle$`type`$\rangle$
  z. B. `array`$[1 \ldots 20]$ of `integer`,
  `array`$[1 \ldots 3]$ of `array`$[1 \ldots 3]$ of `integer`
- Zugriff: A`[`k`]`: Element von A an Stelle k.

3.3 Records

Definition 4 (Record)

- *Aneinanderreihung von Objekten verschiedenen Typs*
- *Abfolge der Typen ist fest*
- *Fester Speicherplatzbedarf*

Ein Record belegt einen zusammenhängenden Block im Speicher.

Beispiel:

record `Adresse`
begin

Vorname:	`array`[1 ... 10] of `character`;
Nachname:	`array`[1 ... 10] of `character`;
Titel:	`array`[1 ... 20] of `character`;
Straße:	`array`[1 ... 20] of `character`;
PLZ:	`integer`;
Ort:	`array`[1 ... 20] of `character`;

end

Feldnamen — Felder (fields)

Schneller Zugriff auf Felder (Relative Lage im Record vom Compiler berechnet)

Notation in Pseudocode:

- Typ: **record** ⟨`name`⟩
  begin
  (⟨`field-name`⟩ : ⟨`type`⟩;)$^+$
  end
- Zugriff: $A.n$ bezeichnet Feld n von A.

Records in C:

Typdeklaration:

struct ⟨`name`⟩
{(⟨`type`⟩ ⟨`field-name`⟩;)$^+$};

Variablendeklaration:

struct ⟨`name`⟩ ⟨`var-name`⟩ (,⟨`var-name`⟩)*;

Zugriff: $A.n$ bezeichnet Feld n von A.

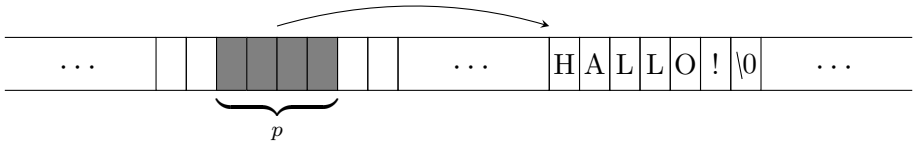
Records können kopiert aber nicht (als ganzes) verglichen werden. Records sind als Parameter und Rückgabewert von Funktionen erlaubt.

3.4 Zeiger

Organisation des Speichers

- in aufeinanderfolgend unnummerierten Speicherzellen (Bytes)
- Index einer Speicherzelle heißt *Adresse*
- Variable belegen zusammenhängende Blöcke von Speicherzellen.

Definition 5 (Zeiger) *Ein* Zeiger *ist eine Variable, die eine Adresse enthält.*



Sprechweise: *p zeigt auf* die Zeichenkette „HALLO!“.

Notation in Pseudocode:

- Typ: ^⟨type⟩ ist der Typ eines Zeigers auf ein Objekt vom Typ ⟨type⟩
- Inhaltsoperator: p^ ist der Inhalt des Objekts an Adresse p.
  Ist p vom Typ ^⟨type⟩, so ist p^ vom Typ ^⟨type⟩.
- Nullzeiger: `nil`
- Speicherverwaltung:
  - p := `new` ⟨type⟩, wobei p : ^⟨type⟩, reserviert Speicherplatz für ein Objekt vom Typ ⟨type⟩ im Speicher, schreibt dessen Adresse nach p
  - `free` p, wobei p : ^⟨type⟩, gibt den Speicherplatz bei p wieder frei

Notation in C:

- ⟨type⟩* ist der Typ eines Zeigers auf ein Objekt vom Typ ⟨type⟩
- Inhaltsoperator: $*p$ für Zeiger p
- Adressoperator: $\&c$ für Variable c

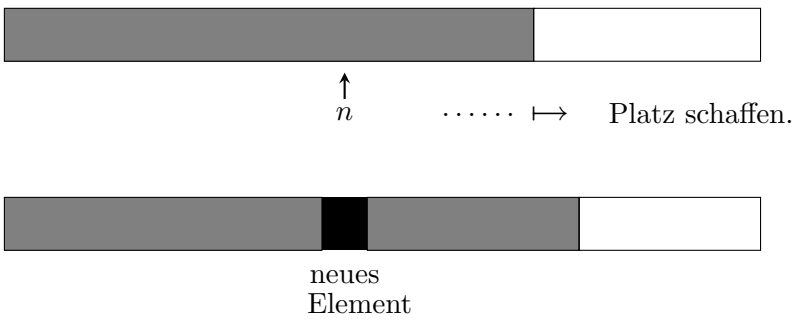
- Nullzeiger: `NULL`
- Speicherbeschaffung: $p = (\langle \texttt{type} \rangle *)$ `malloc(sizeof(`$\langle$`type`$\rangle$`))`;
- Speicherfreigabe: `free` p;

3.5 Dynamische Datenstrukturen

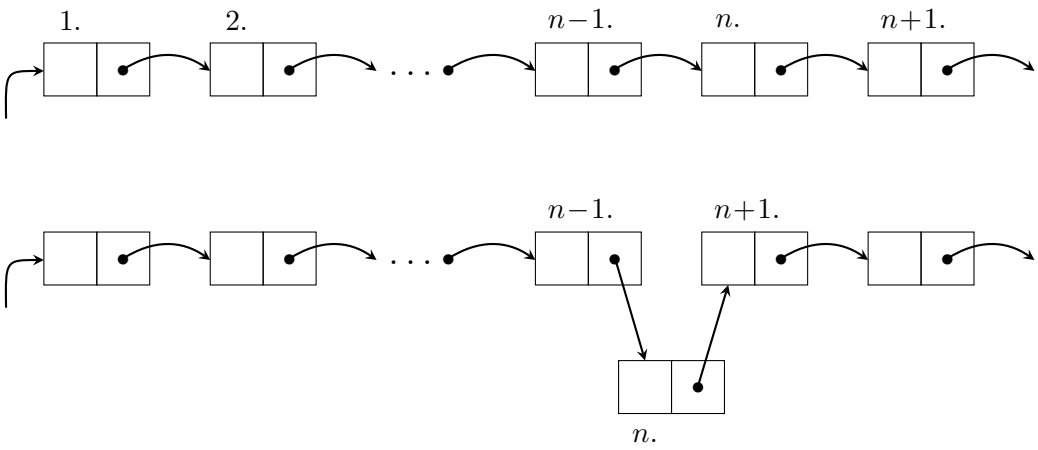
Implementiert mit Hilfe von Zeigern.

3.5.1 Listen

Implementierung mit Feld:



Implementierung mit verketteter Liste:



Jeder Knoten ist eine Datenstruktur vom Typ `list`.

record `list`
begin
    data: `element`;
    next: `^list`;
end

Listenende: Nullzeiger (nil) $\perp$

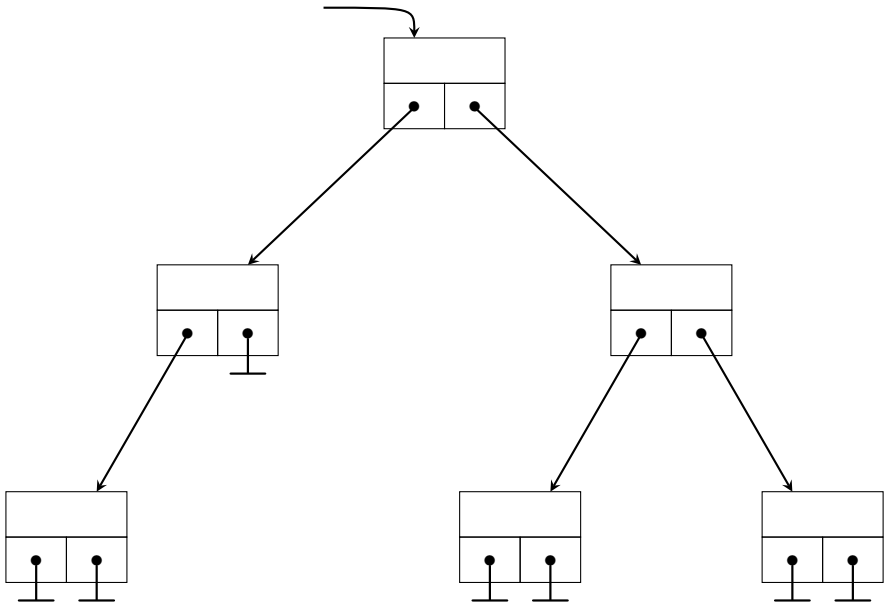
Algorithmus 6 PRINT LIST

Eingabe: l vom Typ `^list`;
Ausgabe: gibt die data-Felder der Liste aus

```
1: begin
2:   while l ≠ nil do
3:     call print(l^.data);
4:     l := l^.next;
5: end
```

3.5.2 Binärbäume

Repräsentation von Binärbäumen:



record `tree`
begin
    data: `element`;
    left, right: `^tree`;
end

Algorithmus 7 PRINT TREE

Eingabe: t vom Typ `^tree`;
Ausgabe: gibt die data-Felder des Baums in Infix-Ordnung aus

```
 1: begin
 2:   call PRINT_T(t);
 3: end
 4:
 5: procedure PRINT_T(t : ^tree)
 6: begin
 7:   if t ≠ nil then
 8:     call print_t(t^.left);
 9:     call print(t^.data);
10:     call print_t(t^.right);
11: end
```

3.5.3 Vergleich statischer und dynamischer Datenstrukturen

Statische Datenstrukturen
(Felder, Records)

+ Konstante Zugriffszeit (unabhängig von Größe der Datenstruktur)
− Größe muss von vorne herein bekannt sein

Dynamische Datenstrukturen
(mit Zeigern)

+ Größe kann dem Bedarf entsprechend angepasst werden
+ Können flexibler verändert werden
− Zugriffszeit abhängig von Größe
− Höherer Platzverbrauch

Algorithmus 8 N-TES ELEMENT

Eingabe: L (Liste, d. h. Zeiger auf verkettete Liste oder nil)
n (positive, ganze Zahl)
Ausgabe: $success$ (Boolean, zeigt an ob n-tes Element vorhanden)
x (wenn $success = true$, das n-te Element von L)

```
1: begin
2:    i := 1;
3:    while i ≠ n ∧ L ≠ nil do
4:       /* L zeigt auf i-ten Eintrag der Eingabeliste */
5:       L := L^.next;
6:       i := i + 1;
7:    if L = nil then
8:       success := false;
9:    else
10:      success := true;
11:      x := L^.data;
12: end
```

Algorithmus 9 EINFÜGEN AN N-TER STELLE

Eingabe: L (Zeiger auf Liste)
n (positive, ganze Zahl)
x (Element)

Ausgabe: *success* (Boolean, zeigt an ob n-tes Element eingefügt werden konnte)
L (wenn $success = false$, die Liste aus der Eingabe, sonst die Liste aus der Eingabe, wobei an n-ter Stelle x eingefügt wurde.)

```
1:  begin
2:    K := L;
3:    i := 1;
4:    while i ≠ n ∧ K ≠ nil do
5:      /* K zeigt auf i-ten Eintrag; K' zeigt (außer für i = 1) auf i−1-ten
        Eintrag */
6:      K' := K;
7:      K := K^.next;
8:      i := i + 1;
9:    if i = n then
10:     success := true;
11:     M := new list;
12:     M^.data := x;
13:     M^.next := K;                    /* K zeigt auf Restliste oder nil */
14:     if i = 1 then                                  /* am Anfang der Liste */
15:       L := M;
16:     else
17:       K'^.next := M;
18:   else
19:     success := false;
20: end
```

Kapitel 4

Analyse von Algorithmen

Vorhersage des Verhaltens von Algorithmen:

- Laufzeit
- Speicherplatz.

Definition 6

Aufwand, Komplexität *eines Algorithmus:*
Laufzeit.

Platzaufwand, Platzkomplexität *eines Algorithmus:*
Speicherplatzbedarf.

Definition 7

Komplexität *eines Problems:*
Algorithmen welcher Komplexität können für ein Problem bestenfalls existieren.

4.1 Näherungsweise Vorhersage des Verhaltens von Algorithmen

- Asymptotisches Verhalten

Konstante Faktoren ignoriert.
Verhalten für große Eingaben ($n \to \infty$).

	Schritte	Näherung	
Alg. 1	$100n$	n	schneller
Alg. 2	$2n^2 + 50$	n^2	
			(für $n \geq 50$)

Konstanten sind in der Regel klein.

- Verhalten bezogen auf Größe n der Eingabe
  - Größe $\widehat{=}$ Platzbedarf der Eingabe.
  - Laufzeit kann für Eingaben gleicher Größe stark variieren.
- Welche der Eingaben gleicher Größe maßgebend für das Verhalten?

  Bester Fall

  → meist nicht repräsentativ

  Durchschnittlicher Fall

  → was heißt durchschnittlich?

  → mathematisch schwierig
  (werden wir nur selten analysieren)

  Schlechtester Fall

  → „auf der sicheren Seite“

  → oft ist der schlechteste Fall nahe am durchschnittlichen Fall

4.2 $\mathcal{O}$-Notation (Landau-Symbole)

Mathematisches Werkzeug zur näherungsweisen Analyse.

Definition 8 ($\mathcal{O}$-Notation) *Seien* $f, g \in \mathbb{R} \to \mathbb{R}^+$.

$$\mathcal{O}(f) = \{g \mid \text{es gibt } c > 0 \text{ und } N \text{ so dass für alle } n > N \text{ gilt } g(n) \leq c\, f(n)\}$$

- $\mathcal{O}(f)$ ist die Menge der durch f von oben beschränkten Funktionen.
- Notation: statt $g \in \mathcal{O}(f)$ oft $g(n) = \mathcal{O}(f(n))$.

Bsp.:

$$5n^2 + 15 \in \mathcal{O}(n^2) \qquad 5n^2 + 15 \leq 6n^2 \quad \text{für } n \geq 4$$
$$5n^2 + 15 \in \mathcal{O}(n^3) \qquad 5n^2 + 15 \leq n^3 \quad \text{für } n \geq 6.$$

Bemerkung: Konstante Faktoren können weggelassen werden.

Bsp.:

$$\mathcal{O}(5n + 4) = \mathcal{O}(n)$$
$$\mathcal{O}(\log_b n) = \mathcal{O}\left(\tfrac{\ln n}{\ln b}\right) = \mathcal{O}(\log n)$$
$\mathcal{O}(1)$ bezeichnet Konstante

Exponentielle Funktionen wachsen schneller als polynomielle.

Lemma 1 *Für Konstanten $c > 0$ und $a > 1$ und monoton wachsende Funktionen f gilt*

$$(f(n))^c \in \mathcal{O}\left(a^{f(n)}\right).$$

(Ohne Beweis)

Bsp.:

Für $f(n) = n$: $\quad n^c \in \mathcal{O}(a^n)$
Für $f(n) = \log_a n$: $\quad \log_a n^c \in \mathcal{O}\left(a^{\log_a n}\right) = \mathcal{O}(n)$.

Lemma 2 *Sei $f \in \mathcal{O}(s(n))$ und $g \in \mathcal{O}(r(n))$. Dann gilt*

$$f(n) + g(n) \in \mathcal{O}(s(n) + r(n))$$
$$f(n) \cdot g(n) \in \mathcal{O}(s(n) \cdot r(n)).$$

Beweis: Nach Definition existieren c_1, N_1, c_2, N_2 so dass $f(n) \leq c_1\, s(n)$ für $n > N_1$, $g(n) \leq c_2\, r(n)$ für $n > N_2$. Behauptung folgt mit $max(c_1, c_2)$ und $max(N_1, N_2)$.

(Für $f(n) - g(n)$ und $f(n)/g(n)$: allgemein nicht gültig.)

□

Lemma 3 *Seien* $f,\ g \in \mathbb{R} \to \mathbb{R}^+$.

$$f \in \mathcal{O}(g) \quad \text{g. d. w.} \quad \limsup_{n\to\infty} \frac{f(n)}{g(n)} < \infty.$$

(Ohne Beweis)

Bsp.:

$$\lim_{n\to\infty} \frac{2n^3}{n^3 - n^2} = 2 < \infty$$
$$\text{Also} \quad 2n^3 \in \mathcal{O}(n^3 - n^2).$$

Weitere asymptotische Notation

Für $f,\ g \in \mathbb{R} \to \mathbb{R}^+$.

- $\Omega(f) = \{g \mid \text{es gibt } c > 0 \text{ und } N \text{ so dass für alle } n > N \text{ gilt } g(n) \geq c\, f(n)\}$

$$f \in \Omega(g) \quad \text{g. d. w.} \quad \liminf_{n\to\infty} \frac{f(n)}{g(n)} > 0.$$

- $\Theta(f) = \mathcal{O}(f) \cap \Omega(f)$.

4.3 Bestimmen des Aufwands eines Algorithmus

Laufzeit

Zählen aller Schritte

- Schritte können verschieden lang dauern (Addition / Division)
- Abhängig von Hardware, Compiler, etc.

Stattdessen: zählen der *wesentlichen Schritte*

Bsp.: Vergleichsoperationen in einem Sortieralgorithmus
(Restliche Operationen proportional zu Zahl der Vergleiche.)

Speicherplatz

Größe des temporären Speicherplatzes während der Ausführung des Algorithmus.
(Hier: Vergleich verschiedener Algorithmen für das selbe Problem. Speicherplatz für Ein- und Ausgabe in beiden Fällen gleich.)

Beispiel: Aufwand zur Berechnung von Binomialkoeffizienten

$$\binom{n}{0} = 1, \quad \binom{n}{n} = 1, \qquad n \geq 0$$

$$\binom{n+1}{k} = \binom{n}{k} + \binom{n}{k+1}, \qquad n \geq 0, \quad 0 < k \leq n$$

Algorithmus 10 BINOMIALKOEFFIZIENTEN

Eingabe: m, ganze Zahl ≥ 0
Ausgabe: B : `array`$[0 \ldots m]$ of `array`$[0 \ldots m]$ of `integer`,
wobei $B[n,k] = \binom{n}{k}$ für $0 \leq k \leq n \leq m$

```
1: begin
2:   B[0][0] := 0;
3:   for n := 1 to m do
4:     B[n][0] := 1;
5:     B[n][n] := 1;
6:     for k := 1 to n − 1 do
7:       B[n][k] := B[n − 1][k] + B[n − 1][k + 1];
8: end
```

Laufzeit:

$$
\begin{array}{ll}
 & 1 \\
n = 1\text{:} & +\,2 \\
n = 2\text{:} & +\,2 + 1 \\
\quad\vdots & \quad\vdots \\
n = i\text{:} & +\,2 + \underbrace{1 + \ldots + 1}_{i-1} \\
\quad\vdots & \quad\vdots \\
n = m\text{:} & +\,2 + \underbrace{1 + \ldots\ldots\ldots + 1}_{m-1} \\
 & = 2m + 1 + \sum_{i=1}^{m-1} i \\
 & = 2m + 1 + \frac{m(m-1)}{2} \\
 & = \frac{1}{2}m^2 + \frac{3}{2}m + 1 \in \mathcal{O}(m^2)
\end{array}
$$

also $\mathcal{O}(m^2)$

Speicherplatz:

zwei Zählvariable,
Konstant, $\mathcal{O}(1)$.

Kapitel 5

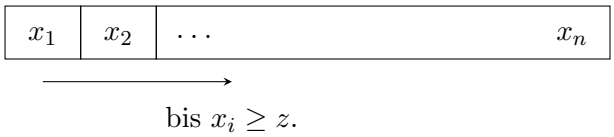
Suchen und Sortieren

5.1 Binärsuche

Problem:

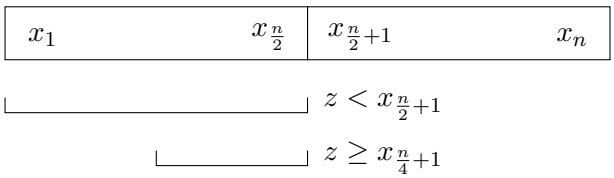
> $x_1, \ldots, x_n$: aufsteigend sortierte Sequenz von Zahlen.
>
> Kommt z in der Sequenz vor; wenn ja, an welcher Stelle?

Lineare Suche:



Komplexität: $\mathcal{O}(n)$

Binäre Suche:



Intervall wird in jedem Schritt halbiert.
Intervallbreite nach k Schritten: $\frac{n}{2^k}$

$$\frac{n}{2^k} = 1 \qquad \rightsquigarrow \quad k = \log_2 n$$

Nach $\log_n$ Schritten ist die Intervallbreite = 1.

Komplexität: $\mathcal{O}(\log n)$

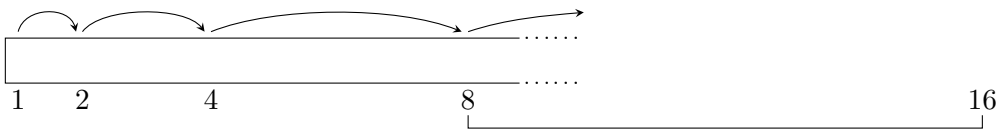
Algorithmus 11 BINÄRSUCHE

Eingabe: X (sortiertes Feld im Bereich $1 \ldots n$)
z (das gesuchte Element)

Ausgabe: pos (Index, so dass $X[pos] = z$ oder 0, falls kein solcher Index existiert)

```
1: begin
2:   pos := FIND(z, 1, n);
3: end
4:
5: function FIND(z : element; Left, Right : integer) : integer
6: begin
7:   if Left = Right then
8:     if X[Left] = z then
9:       Find := Left;
10:    else
11:      Find := 0;
12:  else
13:    Middle := ⌈1/2 (Left + Right)⌉;
14:    if z < X[Middle] then
15:      Find := FIND(z, Left, Middle − 1);
16:    else
17:      Find := FIND(z, Middle, Right);
18: end
```

Binärsuche in Sequenzen unbekannter Größe:



Idee: Schrittweise Verdoppelung des Bereichs, in dem gesucht wird.
Dann Binärsuche.

5.2 Binäre Suchbäume

Definition 9 (Baum) *Ein* Baum *ist eine Datenstruktur, die aus* Knoten *und* Kanten *besteht, und wobei*

- *jede Kante ein geordnetes Paar aus zwei verschiedenen Knoten ist.*
- *Ist* (A, B) *eine Kante, so heißt* A Elternknoten *von* B, B Kind *von* A.
- *Es gibt genau einen Knoten ohne Elternknoten. Dieser ist die* Wurzel *des Baums.*
- *Alle anderen Knoten haben jeweils genau einen Elternknoten.*
- *Ein* Pfad *von Knoten* A *nach Knoten* B *ist eine Folge von Kanten* $(V_1, V_2), (V_2, V_3), \ldots, (V_{n-2}, V_{n-1}), (V_{n-1}, V_n)$ *wobei* $A = V_1, B = V_n$.
- *Für jeden Knoten* A *gibt es genau einen Pfad von der Wurzel nach* A.

Definition 10 *Weitere Begriffe*

- *Der* Grad eines Knotens *ist die Zahl seiner Kinder.*
- *Der* Grad eines Baums *ist der maximale Grad seiner Knoten.*
- *Ein* Binärbaum *ist ein Baum vom Grad* 2.
- *Gibt es einen Pfad von* A *nach* B, *so heißt* A Vorfahre *von* B, B Nachkomme *von* A.
- *Ein Knoten* A *ist in* Ebene k, *wenn der Pfad von der Wurzel zu* A *aus* k *Kanten besteht.*
- *Die* Höhe *des Baums ist die maximale Ebene eine Knotens.*
- *Ein Knoten ohne Kinder heißt* Blatt, *ein Knoten mit Kindern* innerer Knoten.

Ziel: Datenstruktur für Index-Wert-Paare, wobei zu jedem Index höchstens ein Wert vorkommt (Wörterbuch).

Operationen:

- Einfügen eines Paars (k, d)
- Suchen des Werts d zu Index k
- Löschen des Paars mit Index k

Datenstruktur

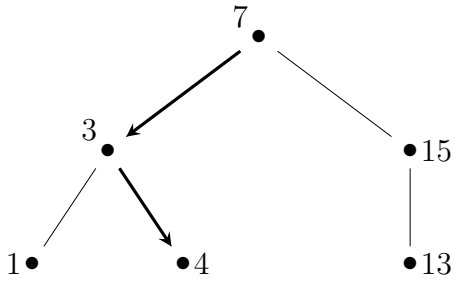
```
record tree
begin
    key:          element;
    data:         ...;
    left, right:  ^tree;
end
```

Konsistenzbedingung:

Für jeden Knoten gilt:

- Der Schlüssel ist größer als die Schlüssel im linken Unterbaum.
- Der Schlüssel ist kleiner als die Schlüssel im rechten Unterbaum.

Bsp.: suche $k = 6$.



$k = 6$ kommt nicht vor.

Vorgehen bei der Suche:

Vergleiche k mit Schlüssel k' an der Wurzel:

- $k = k'$: Element gefunden.
- $k < k'$: Rekursive Suche im linken Teilbaum.
- $k > k'$: Rekursive Suche im rechten Teilbaum.

Algorithmus 12 BST-SUCHE

Eingabe: T (Zeiger auf einen binären Suchbaum)
k (Schlüssel)
Ausgabe: N (Zeiger auf Knoten mit Schlüssel k,
oder *nil* falls nicht vorhanden)

```
1: begin
2:   N := FIND(T, k);
3: end
4:
5: function FIND(T : ^tree; k : element) : ^tree
6: begin
7:   if T = nil ∨ T^.key = k then
8:     return T;
9:   else
10:    if k < T^.key then
11:      return FIND(T^.left, k);
12:    else
13:      return FIND(T^.right, k);
14: end
```

Einfügen:

Nur wenn Schlüssel noch nicht vorhanden.

- Suche nach Schlüssel k endet erfolglos an Knoten B mit Schlüssel k'. B ist entweder Blatt oder innerer Knoten von Grad 1.
- Neuer Knoten mit Schlüssel k als linker ($k < k'$) oder rechter ($k > k'$) Nachfolger von B.

Löschen:

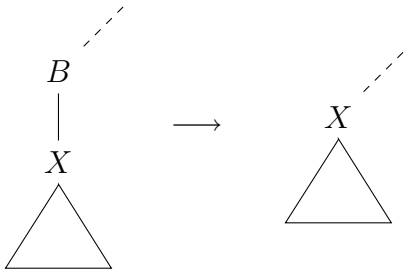
Schlüssel k gehört zu Knoten B.

- 1. Fall: B ist Blatt: direkt löschen
- 2. Fall: B hat nur ein Kind:

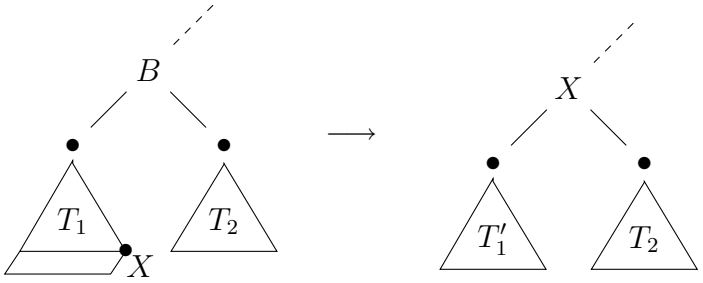
Algorithmus 13 BST-EINFÜGEN

Eingabe: T (Zeiger auf binären Suchbaum)
k (Schlüssel)
d (Wert)
Ausgabe: T (zeigt auf modifizierten Suchbaum)
C (zeigt auf den neuen Knoten, `nil`, falls schon ein Knoten mit Schlüssel k vorhanden)

```
 1: begin
 2:   if T = nil then
 3:     C := new tree;
 4:     C^.key := k;
 5:     C^.data := d;
 6:     C^.left := nil;
 7:     C^.right := nil;
 8:     T := C;
 9:   else
10:     N := T;
11:     C := T;                                   /* C ≠ nil */
12:     while N ≠ nil ∧ C ≠ nil do
13:       if N^.key = k then
14:         C := nil;
15:       else
16:         P := N;                               /* Schleppzeiger */
17:         if k < N^.key then
18:           N := N^.left;
19:         else
20:           N := N^.right;
21:     if C ≠ nil then                           /* Schlüssel k ist neu. */
22:       C := new tree;
23:       C^.key := k;
24:       C^.data := d;
25:       C^.left := nil;
26:       C^.right := nil;
27:       if k < P^.key then
28:         P^.left := C;
29:       else
30:         P^.right := C;
31: end
```



- 3. Fall: B hat zwei Kinder:



Knoten X in T_1: Vorgänger von B in der Schlüsselordnung.

- X hat kein rechtes Kind
- Lösche X (1. oder 2. Fall)
- Ersetze B durch X.

Komplexität

Abhängig von Form des Baums und Lage des gesuchten Knotens.

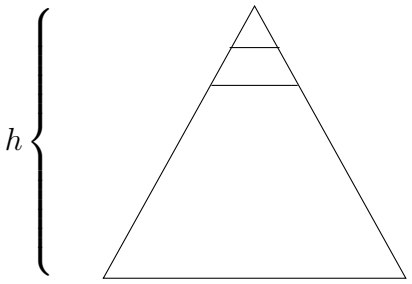
$\rightsquigarrow$ Laufzeit im schlechtesten Fall:

> Länge des längsten Pfads von der Wurzel.
> (gilt für Suchen, Einfügen, Löschen).

Länge des längsten Pfads im schlechtesten Fall $n - 1$ für n Elemente.
$\rightsquigarrow$ nichts erreicht.

5.3 Balancierte Binärbäume

Voll besetzter Baum der Höhe h:



Ebene		Knoten	
	0		1
	1		2
	2		4
	⋮		⋮
	i		2^i
	⋮		⋮
	h		2^h
			$2^{h+1}-1$

Enthält $2^{h+1}-1$ Knoten

Ziel: Suchbaum der Höhe $\mathcal{O}(\log n)$

Adelson-Velskii, Landis (russ. Math.): 1962: AVL-Bäume

Definition 11 (AVL-Baum) *Ein AVL-Baum ist ein binärer Baum, sodass für jeden Knoten die Höhen des linken und des rechten Teilbaums sich maximal um* 1 *unterscheiden.*

Höhe des leeren Baums: -1.

Theorem 1 *Für die Höhe h eines AVL-Baums mit n Knoten gilt*

$$h = \mathcal{O}(\log n)\,.$$

Beweis (Idee):

Betr. AVL-Bäume bestimmter Höhe mit möglichst wenigen Knoten.

Damit: Suche in AVL-Bäumen: $\mathcal{O}(\log n)$.

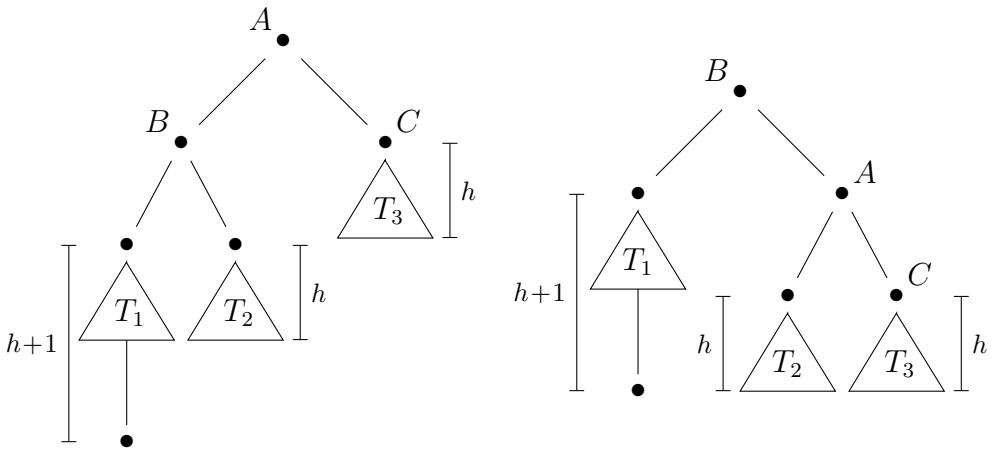
Einfügen und Löschen:

- Wie in normalen Binärbäumen.
- Ggf. muss AVL-Bedingung wiederhergestellt werden.

Einfügen

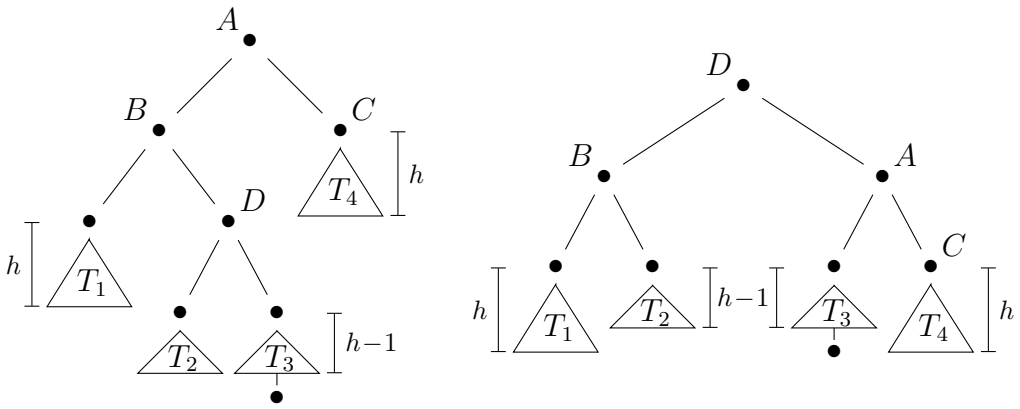
Betr. kleinsten Teilbaum, dessen AVL-Eigenschaft verletzt ist.

Einfache Rotation:



$h(T_1) = h(T_2)$ weil *kleinster* Teilbaum betr. wird.

Doppelte Rotation:



- AVL-Eigenschaft kann mit einer Rotation wiederhergestellt werden.
- Rotationen erhalten die Sortierung des Suchbaums.

Wie findet man diesen Teilbaum?

Definition 12 *Für einen Knoten A heißt*

Höhe des rechten Teilbaums minus
Höhe des linken Teilbaums

Balancefaktor *von A.*

Bemerkung: In einem AVL-Baum gilt für alle Knoten:

$$\text{Balancefaktor} \in \{-1, 0, 1\}.$$

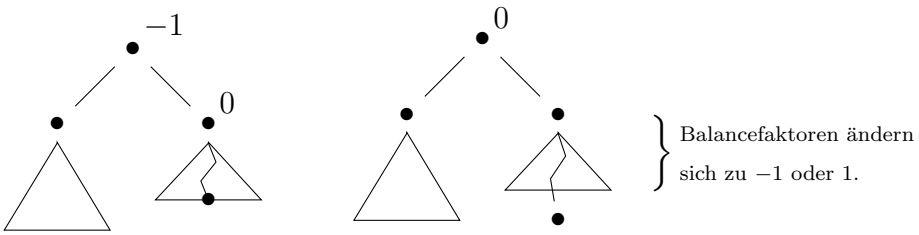
Definition 13 *Der niedrigste Vorfahr des eingefügten Knotens mit Balancefaktor* $\neq 0$ *(vor dem Einfügen) heißt* kritischer Knoten.

Bemerkung:

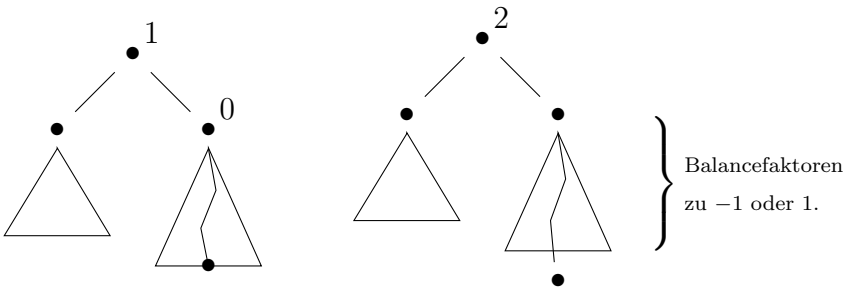
1. Wenn die AVL-Eigenschaft verletzt wird, ist der kritische Knoten die Wurzel des kleinsten Teilbaums, wo dies der Fall ist.
2. Balancefaktoren oberhalb des kritischen Knotens müssen nicht verändert werden.

zu 1.

- 1. Fall: Balancefaktor wird 0:



- 2. Fall: Balancefaktor wird -2 oder 2:



zu 2.

Höhe des Teilbaums bleibt unverändert. (ggf. nach Rotation)

Implementierung:

- Speichern der Balancefaktoren in den Knoten
- Beim Abstieg in den Baum merke kritischen Knoten
- Anpassen der Balancefaktoren auf Pfad von kritischem Knoten zu Blatt
- Ggf. Rotation

Aufwandsanalyse: Rotation konstant.

$\mathcal{O}(\log n)$ gesamt.

Löschen

U. U. mehrere Rotationen nötig, beschränkt durch $\mathcal{O}(\log n)$.

5.4 Sortieren

Grundlage für viele Algorithmen

- sehr ausführlich untersucht
- viel Rechenzeit wird auf Sortieren verwendet
- Dutzende von Algorithmen

Problem:

Gegeben n Zahlen $x_1, x_2, \ldots, x_n$. Ordne diese aufsteigend.

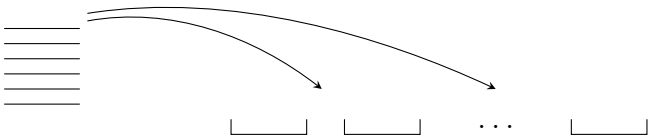
Hierzu äquivalent:

Finde eine Sequenz paarweise verschiedener Indizes $1 \leq i_1, i_2, \ldots, i_n \leq n$, so dass $x_{i_1} \leq x_{i_2} \leq \ldots \leq x_{i_n}$.

Definition 14 *Ein Sortieralgorithmus heißt*

- stabil*, wenn die Reihenfolge gleich großer Elemente nicht verändert wird.*
- in-place*, wenn neben dem Eingabe-Array kein Arbeitsspeicher benötigt wird.*

5.4.1 Postraumsortierung



Bucket-Sort

Briefe werden in korrespondierende Fächer eingeordnet.

Analyse: n Elemente, Zahlen im Bereich $1, \ldots, m$.

- Verteilen der Eingabe auf die Fächer.
- Abfragen der Fächer der Reihe nach und Aufsammeln der Elemente

  $\rightsquigarrow \mathcal{O}(m+n)$

D. h. linear falls $m = \mathcal{O}(n)$, sehr ineffizient für große m.

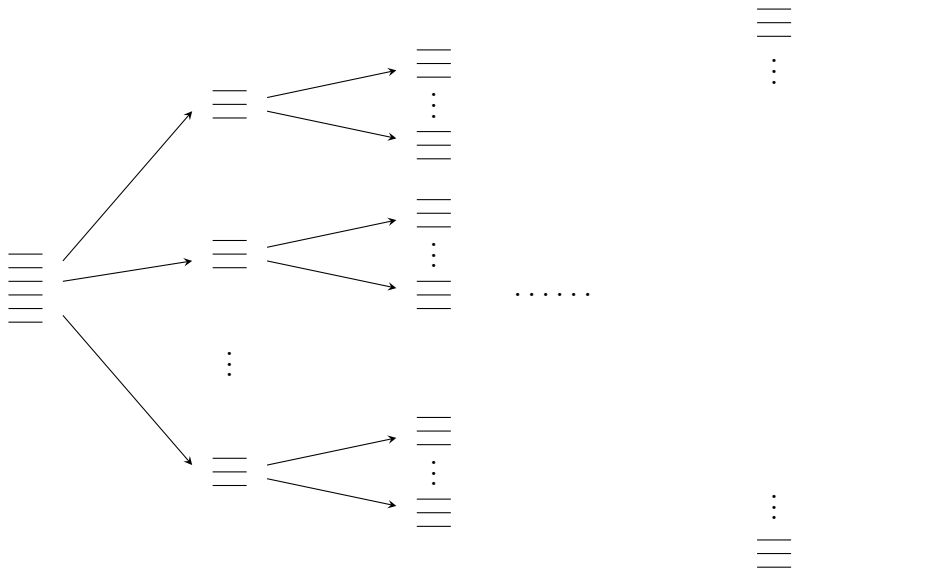
Verbesserung: Radix-Sort

Voraussetzung:

Elemente sind k-stellige Zahlen, jede Ziffer $\in \{0, \ldots, d-1\}$.

Idee:

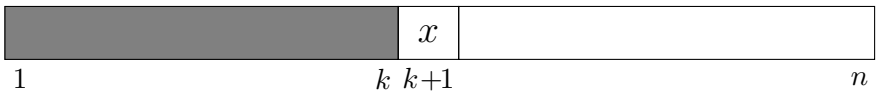
- Bucket-Sort zunächst nach der führenden Ziffer
- Dann rekursiv nach den weiteren Ziffern



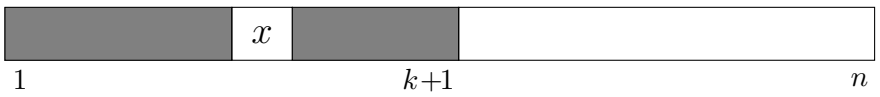
Zahl der jeweils gleichzeitig benötigten Fächer: $k \cdot d$.

5.4.2 Sortieren durch Einfügen

Voraussetzung: Die ersten k Elemente sind sortiert.



Dann können wir die ersten $k + 1$ Elemente sortieren.

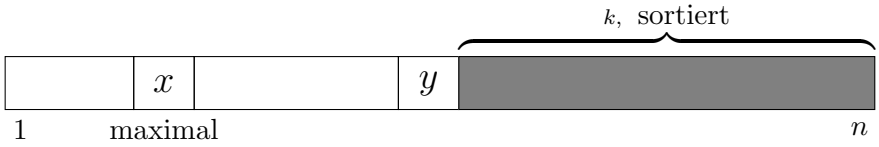


Aufwandsabschätzung:

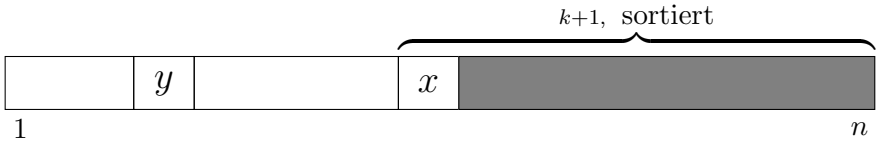
Aufwand	k. **Schritt**	**gesamt**
Vergleiche:	k	$\mathcal{O}(n^2)$
mit Binärsuche:	$\log k$	$\mathcal{O}(n \log n)$
Kopien:	k	$\mathcal{O}(n^2)$
Insgesamt:		$\mathcal{O}(n^2)$

5.4.3 Sortieren durch Auswählen

Voraussetzung: Die k größten Elemente sind sortiert und befinden sich am Ende des Felds.



Dann können wir die $k+1$ größten Elemente sortieren, und diese befinden sich wieder am Ende des Felds.



Aufwandsabschätzung:

Aufwand	k. **Schritt**	**gesamt**
Vergleiche:	$n-k$	$\mathcal{O}(n^2)$
Kopien:	2	$\mathcal{O}(n)$
Insgesamt:		$\mathcal{O}(n^2)$

mit AVL-Bäumen: $\mathcal{O}(n \log n)$

Aufwandsanalyse mit rekursiven Gleichungen

Bsp.: Sortieren durch Einfügen

$T(k)$ Zeitbedarf zum Sortieren der ersten k Elemente

$$T(1) = 0$$
$$T(k+1) \leq 2k + T(k)$$

mit

$$T'(1) = 0$$
$$T'(k+1) = 2k + T'(k)$$

gilt $T(k) \leq T'(k)$ für alle $k \in \mathbb{N}$

Lösung der Gleichung: $T'(k) = k^2 - k$.

Bsp.: Sortieren durch Auswählen

$T(k)$ Zeitbedarf zum Finden und Sortieren der k größten Elemente

$$T(0) = 0$$
$$T(k+1) = n - k + 2 + T(k)$$

Lösung:

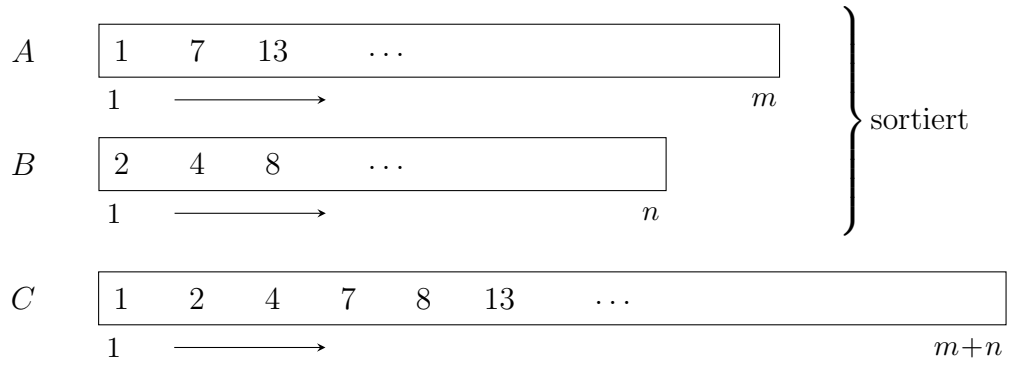
$$T(k) = -\tfrac{1}{2}k^2 + \left(\tfrac{5}{2} + n\right)k$$
$$T(n) = \tfrac{1}{2}n^2 + \tfrac{5}{2}n$$

5.4.4 Mergesort (Sortieren durch Verschmelzen)

Idee: In Sortieren durch Einfügen:

Ein Suchdurchlauf (durch die sortierten Elemente) kann die Position für mehrere Elemente finden.

Merge-Operation:



Ein simultaner Durchlauf durch A und B liefert sortiertes Feld C.

Aufwand: $m+n$ Kopien
$\leq m+n$ Vergleiche

Platzbedarf: $m+n$

Der Sortieralgorithmus

- Teile Feld in zwei Hälften.
- Sortiere Hälften rekursiv; bei Feldgröße 1 ist das Feld trivial bereits sortiert
- Verschmelze sortierte Hälften mit Merge-Operation zu sortiertem Feld.

Aufwand:

$T(n)$ ist der Aufwand zum Sortieren eines Felds der Größe n

$T(2n) \;=\; 2\,T(n) + 2n,\;\; T(1) \;=\; 0$

Lösung: $\quad T(n) \;\in\; \mathcal{O}(n \log n)$

Platzaufwand:

Merge-Operation benötigt zusätzlichen Speicherplatz $\quad\rightsquigarrow\quad \mathcal{O}(n)$.

Master-Theorem

Zum Lösen von Rekursionsgleichungen für Teile-und-Herrsche Algorithmen.

Theorem 2 (Master Theorem) *Sei $a \geq 1, b > 1$ und*

$$T(n) \;=\; a\,T\big(\tfrac{n}{b}\big) + f(n).$$

Dann ist $T(n)$ asymptotisch beschränkt wie folgt

a) wenn $f \in \mathcal{O}(n^s)$ für $s < \log_b a$, dann

$$T(n) \in \Theta\big(n^{\log_b a}\big)$$

b) wenn $f \in \Theta\big(n^{\log_b a}\big)$, dann

$$T(n) \in \Theta\big(n^{\log_b a} \cdot \log n\big)$$

c) wenn $f \in \Omega(n^s)$ für $s > \log_b a$ und es existiert ein $c < 1$ und ein $k \in \mathbb{N}$, so dass für alle $n > k$ gilt $a\,f\big(\frac{n}{b}\big) \leq c\,f(n)$ dann

$$T(n) \in \Theta(f)\,.$$

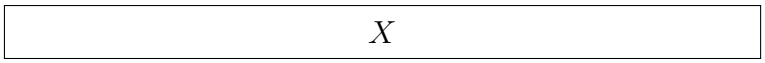
Ohne Beweis.

5.4.5 Quicksort (Hoare 1961)

- Teile-und-Herrsche: $\quad \mathcal{O}(n \log n)$
- Mergesort: Aufteilung ist beliebig, und man kann nicht vorhersagen, wo ein Element nach dem Verschmelzen landet.

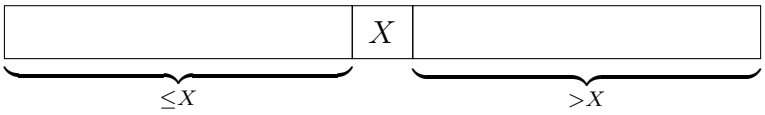
Idee: aufwendige Aufteilung, einfaches Zusammenfügen.

Partition



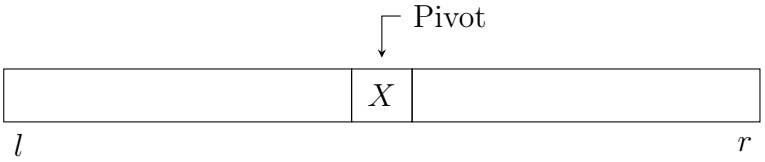
Angenommen, wir kennen den Median X.

Umsortieren, so dass

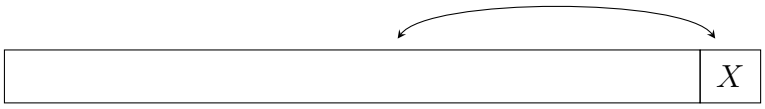


Jetzt kann man rekursiv die linke und die rechte Hälfte sortieren.

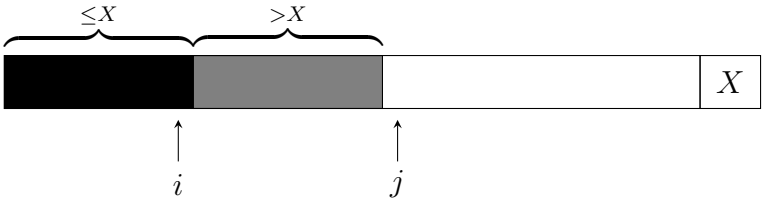
Partitionierung in-place:



Pivot an den Rand bewegen:



Voraussetzung: Bereich $X[l \ldots j-1]$ ist partitioniert



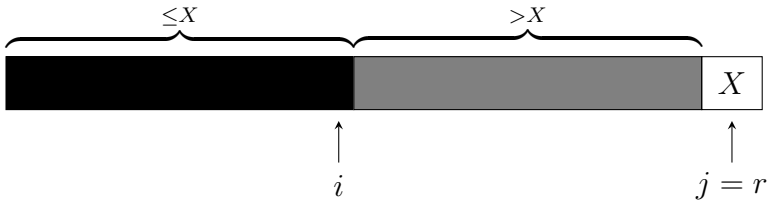
Partitioniert heißt:

- für $l \leq k \leq i$ gilt $A[k] \leq X$
- für $i+1 \leq k \leq j$ gilt $A[k] > X$

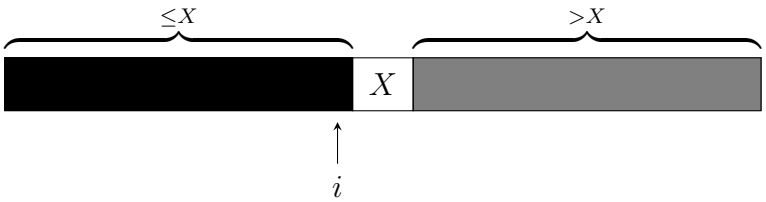
Vergrößern der Partition $j \mapsto j+1$:

- falls $A[j] > X$: nichts zu tun
- falls $A[j] \leq X$: $i := i+1$; vertausche $A[i]$ und $A[j]$

Zum Schluss:



Pivot in die Mitte:



Wahl des Pivots:

- Suche des Medians: aufwendig, $\mathcal{O}(n)$
- Heuristik: Median des linken, mittleren und rechten Elements
- Wahl eines zufälligen Elements
- Wenn Eingabe zufällig verteilt: Element ganz rechts

function PARTITION (A : `array`$[1 \ldots n]$ of `element`; l, r : `integer`)
: `integer`

/*
Partitioniert den Bereich $A[l \ldots r]$, d. h. ist p der Rückgabewert so gilt

$$A[i] \leq A[p] \quad \text{für} \quad l \leq i \leq p \quad \text{und}$$
$$A[i] \leq A[p] \quad \text{für} \quad p < i \leq r$$

*/

```
1:  begin
2:    x := A[r];
3:    i := l − 1;
4:    for j := l to r − 1 do
5:      if A[j] ≤ x then
6:        i := i + 1;
7:        vertausche A[i] und A[j];
8:    vertausche A[i + 1] und A[r];
9:    return i + 1;
10: end
```

Algorithmus 14 QUICKSORT

Eingabe: Feld A im Bereich $1 \ldots n$
Ausgabe: Feld A in sortierter Reihenfolge

```
1:  begin
2:    Q_Sort(1, n);
3:  end
4:
5:  procedure Q_Sort(left, right)
6:  begin
7:    if left < right then
8:      middle := Partition(A, left, right);
9:      Q_Sort(left, middle − 1);
10:     Q_Sort(middle + 1, right);
11: end
```

Komplexität

- Falls Partitionierung das Feld immer halbiert:

$$\begin{aligned} T(n) &= 2\,T(\tfrac{n}{2}) + \mathcal{O}(n)\,, \quad T(2) = 1 \\ T(n) &\in \mathcal{O}(n \log n) \end{aligned}$$

- Falls der Pivot nahe am größten oder kleinsten Element liegt, z. B. Pivot ist immer das kleinste Element:

$$T(n) = \mathcal{O}\big(n^2\big)$$

  z. B., wenn das Feld schon sortiert ist.

Analyse im durchschnittlichen Fall:

Laufzeit, wenn der Pivot das i.-kleinste Element ist (Zahl der Vergleiche):

$$T(n) = n-1+T(i-1)+T(n-i)$$

Durchschnittliche Laufzeit, wenn Wahrscheinlichkeit für alle Elemente gleich:

$$\begin{aligned} T(n) &= n-1 + \frac{1}{n}\sum_{i=1}^{n}\left(T(i-1)+T(n-i)\right) \\ &= n-1 + \frac{2}{n}\sum_{i=0}^{n-1} T(i) \end{aligned}$$

Lösung der Rekurrenz:

$$T(n) \in \mathcal{O}(n \log n)$$

Lösung der Rekursionsgleichung für Quicksort

$$\begin{aligned} T(n) &= n-1 + \frac{2}{n}\sum_{i=0}^{n-1} T(i) \qquad (n \geq 2) \\ T(1) &= 0 \end{aligned}$$

Elimination der Summe: $(n \geq 2)$

$$\begin{aligned} (n+1)\,T(n+1) - n\,T(n) &= (n+1)n + 2\sum_{i=0}^{n} T(i) - n(n-1) - 2\sum_{i=0}^{n-1} T(i) \\ &= 2n + 2\,T(n) \end{aligned}$$

$$T(n+1) = \frac{n+2}{n+1}\,T(n) + \frac{2n}{n+1} \leq \frac{n+2}{n+1}\,T(n) + 2$$

Definition 15

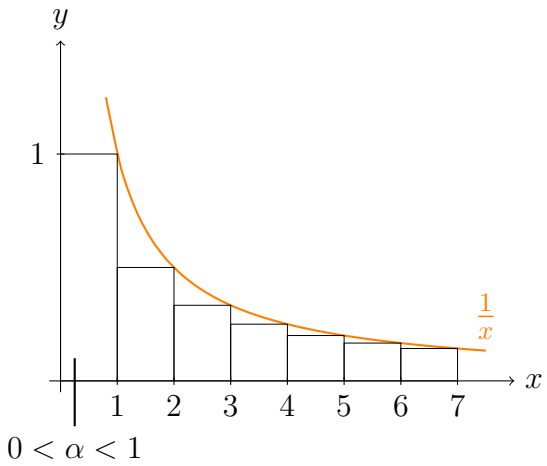
$$\begin{aligned} T'(n+1) &= \frac{n+2}{n+1}\,T(n) + 2 \qquad (n \geq 2) \\ T'(1) &= T(1) = 0 \\ T'(2) &= 2 > 1 = T(2) \end{aligned}$$

Damit $T(n) \leq T'(n)$.

$$\begin{aligned} T'(n) &= 2 + \frac{n+1}{n}\left(2 + \frac{n}{n-1}\left(2 + \frac{n-1}{n-2}\left(\ldots + \frac{5}{4}\left(2 + \frac{4}{3}\underbrace{T'(2)}_{2}\right)\ldots\right)\right)\right) \\ &= 2 + 2\,\frac{n+1}{n} + 2\,\frac{n+1}{n-1} + 2\,\frac{n+1}{n-2} + \ldots + 2\,\frac{n+1}{4} + 2\,\frac{n+1}{3} \\ &= 2\,(n+1)\Bigg(\underbrace{\frac{1}{n+1} + \frac{1}{n} + \ldots + \frac{1}{3} + \frac{1}{2} + 1}_{H(n+1)} - \frac{3}{2}\Bigg) \\ &= \mathcal{O}(n \log n) \end{aligned}$$

Harmonische Zahlen

$$
\begin{aligned}
H(n) &= \sum_{i=1}^{n} \frac{1}{i} \\
&\leq \int_{\alpha}^{n} \frac{1}{x}\,\mathrm{d}x \\
&= [\ln x + C]_{x=\alpha}^{n} \\
&= \ln n - \ln \alpha \\
&= \mathcal{O}(\log n)
\end{aligned}
$$



Speicherplatzkomplexität

Rekursionstiefe:

Im schlechtesten Fall: $\mathcal{O}(n)$

Im Durchschnitt: $\mathcal{O}(\log n)$

In der Praxis:

Quicksort sehr schnell, da viele Vergleiche mit dem gleichen Wert. Dieser kann in einem Prozessorregister gehalten werden und muss nicht nachgeladen werden.

5.4.6 Heapsort

Definition 16 (Heap) *Ein* Heap *(Halde) ist ein Binärbaum, in dem für alle Knoten n die* Heap-Eigenschaft *gilt:*

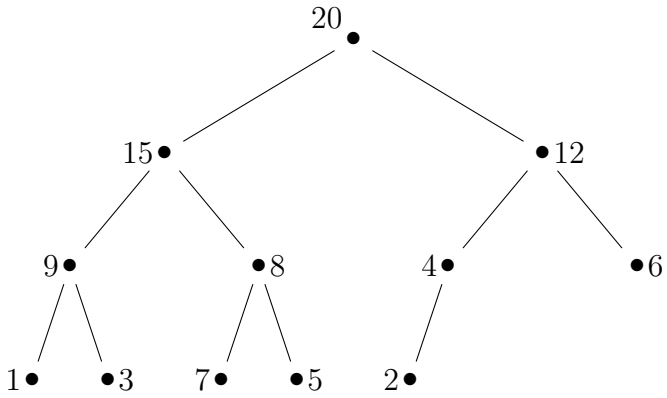
- *Für alle Kinder m von n gilt:*

$$\textit{Schlüssel von } n \quad \geq \quad \textit{Schlüssel von } m.$$

Nützlich zur Implementierung einer Prioritätswarteschlange:

Einfügen(x): Schlüssel x einfügen
EntferneMax: Größten Schlüssel aus der Struktur entfernen.

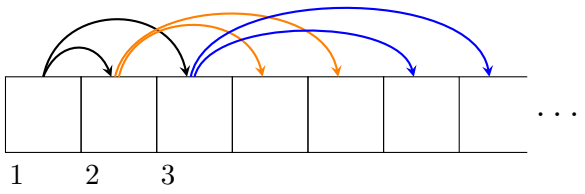
Bsp.:



Werte in einem Heap

Implizite Repräsentation: im Feld

Feld enthält alle Knoten.

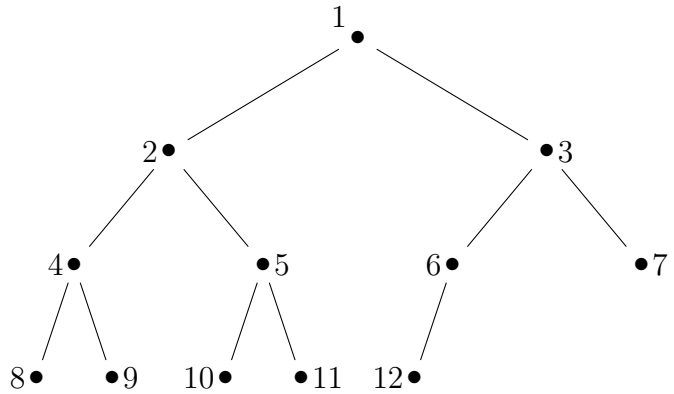


Induktive Beschreibung der Darstellung

- Wurzel in $A[1]$

- Die Kinder des Knotens in $A[i]$ befinden sich in $A[2i]$ und $A[2i+1]$.

Ablage der Ebenen im Feld:

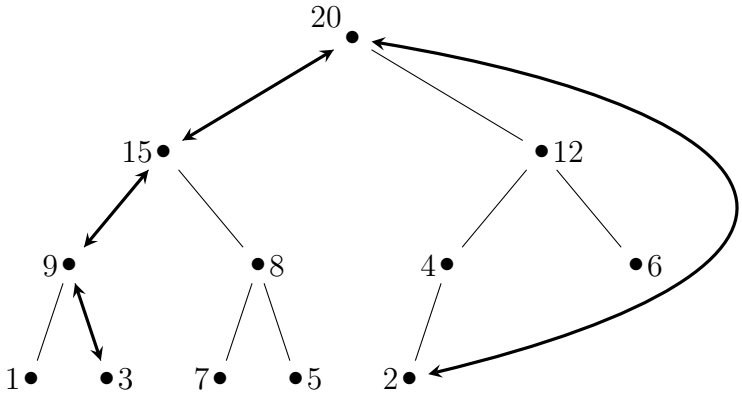


Bei Feldgröße n: Höhe des Baums $\mathcal{O}(\log n)$

Entfernen aus dem Heap: $A[1 \ldots n]$

Größtes Element: Wurzel $A[1]$

Bsp.:



$A[1] := A[n]; \quad n := n - 1;$

Einsinken für $A[j]$:

- $A[j] \geq A[2j]$ und $A[j] \geq A[2j+1]$: Heapeigenschaft wiederhergestellt
- Sonst: wähle $k \in \{2j, 2j+1\}$, so dass $A[k] \geq A[2j]$ und $A[k] \geq A[2j+1]$.
- Vertausche $A[j]$ und $A[k]$.
- Rekursives Einsinken für $A[k]$.

Einfügen in Heap: $A[1 \ldots n]$

Analog: Einfügen in $A[n+1]$ und nach oben tauschen bis Heap-Eigenschaft erfüllt.

Heapsort

Phase I: Konstruiere Heap aus Eingabe

Phase II: Die Soriterung ergibt sich durch sukzessives Entfernen des aktuellen Maximums aus dem Heap. $\Theta(n \log n)$

Phase I:

1. Variante: top down

Heap wird im vorderen Teil von A aufgebaut.

- $A[1]$ ist ein Heap.
- Sei $A[1 \ldots k-1]$ ein Heap. Durch Einfügen von $A[k]$ (nach oben tauschen) erhält man einen Heap $A[1 \ldots k]$.

Aufwand: $\Theta(n \log n)$

2. Variante: bottom up

Heap wird im hinteren Teil von A aufgebaut.

Beachte:

$A[i+1 \ldots n]$ ist im Allgemeinen kein Heap. $A[i+1 \ldots n]$ besteht aus Teilbäumen des Gesamtbaums.

Voraussetzung:

Alle Teilbäume in $A[i+1 \ldots n]$ erfüllen die Heap-Bedingung.

- Gilt für $A[n]$.
  Gilt aber auch für $A[\lfloor \frac{n}{2} \rfloor + 1 \ldots n]$! (Die Teilbäume in $A[\lfloor \frac{n}{2} \rfloor + 1 \ldots n]$ sind alles einzelne Knoten.)

- Die Voraussetzung gelte für $A[i{+}1 \ldots n]$. $A[i]$ hat bis zu zwei Kinder ($A[2i]$ und $A[2i+1]$). Nach Voraussetzung sind diese die Wurzeln gültiger Heaps.

  $\rightsquigarrow$ Einsinken von $A[i]$ in den Teilbaum mit der größeren Wurzel.

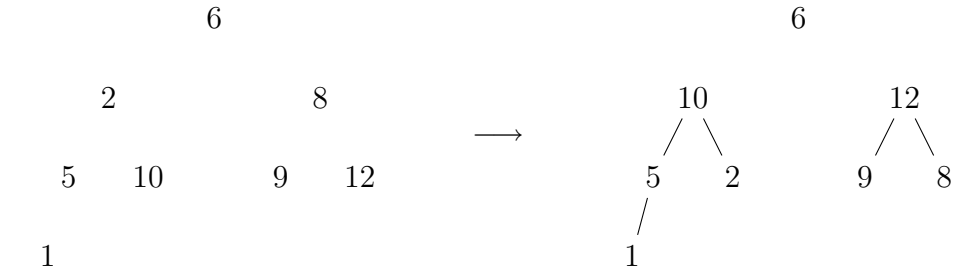
  Damit erfüllt $A[i \ldots n]$ auch die Heap-Bedingung.

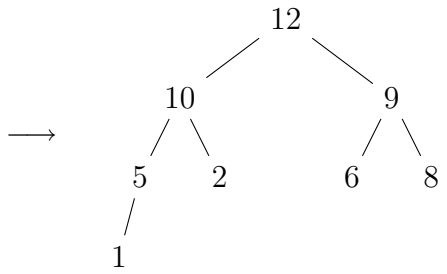
Aufwand: $\mathcal{O}(n)$

Bsp.:

6 2 8 5 10 9 12 1

Phase I, bottom up: Aufbau von Heaps





Phase II: Entfernen der Maxima ergibt das sortierte Feld

Eigenschaften:

$\Theta(n \log n)$

in-place

nicht stabil

5.4.7 Untere Schranke für Sortieralgorithmen

Argument über beliebige Sortieralgorithmen.

Benötigt: Methode mögliche Berechnungen zu beschreiben (*Berechnungsmodell*)

- Turing Maschine
- Registermaschine
- Entscheidungsbäume

Definition 17 (Entscheidungsbaum) *Ein Entscheidungsbaum ist ein Binärbaum, der einen Algorithmus modelliert.*

- *Jeder innere Knoten stellt eine Anfrage an die Eingabe dar und hat zwei Kinder. Das Ergebnis der Anfrage ist eine von zwei Möglichkeiten, jede assoziiert mit einem Kind.*
- *Jedes Blatt repräsentiert eine mögliche Ausgabe*

Pfad $\widehat{=}$ einer möglichen Berechnung

Höhe des Baums $\widehat{=}$ Laufzeit im schlechtesten Fall.

Sortieralgorithmus:

Eingabe: Sequenz $x_1, \ldots, x_n$
Ausgabe: sortierte Sequenz

- Ausgabe ist Permutation der Eingabe.
- Als Ausgabe ist jede beliebige Permutation möglich.
- Es gibt $n!$ Permutationen von $(1 \ldots n)$.
- Entscheidungsbaum für Sortieralgorithmus hat $n!$ Blätter.
- Baumhöhe h ist am geringsten, wenn alle Pfade gleich lang.

$$n! = \#\text{Blätter} \leq 2^h$$
$$\rightsquigarrow \quad h \geq \log_2 n!$$

Stirling'sche Formel: $n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \mathcal{O}\left(\frac{1}{n}\right)\right)$

$$\rightsquigarrow \quad h = \Omega(n \log n).$$

Theorem 3 *Die Höhe des Entscheidungsbaums für einen Sortieralgorithmus ist* $\Omega(n \log n)$.

- Die Schranke gilt nur für vergleichsbasierte Sortierverfahren. Andere Verfahren, z. B. Bucket-Sort, können besseren Aufwand haben.
- Das vergleichsbasierte Sortierproblem hat Komplexität $\Omega(n \log n)$.
- Informationstheoretische Schranke:

  Wie viel Information (Vergleiche) ist nötig, um das Ergebnis zu bestimmen.

5.5 Hash-Tabellen (Streutabellen)

Engl. to hash = zerstückeln

Bsp.:

Datenbank der immatrikulierten Studenten,

- Matrikelnummer ist Schlüssel
- Operationen zum Einfügen, Löschen und Suchen nach dem Schlüssel

Lösung: AVL-Bäume, $\mathcal{O}(\log n)$ bei n Studenten.

Mit Hash-Tabellen durchschnittliche Laufzeit $\Theta(1)$ möglich!

Idee: Zugriff auf Feldelement $\Theta(1)$

Sei U Menge der Schlüssel, K Menge der tatsächlich eingefügten Schlüssel.

- Feld der Größe $|U|$ wäre Platzverschwendung
- Feld in der Größenordnung von $|K|$ ist ok (weil $K \subseteq U$ klein).
- Wähle Feld mit Indexbereich $0 \ldots m-1$, wobei $m \sim |K|$.

Welcher Index $\in \{0, \ldots, m-1\}$ wird Schlüssel k zugeorndet?

- Hashfunktion h: $U \to \{0, \ldots, m-1\}$
- $h(k)$ heißt *Hashwert* von k

- Speichere Eintrag mit Schlüssel k an Position $h(k)$

Da $m \ll |U|$, h nicht injektiv, d. h. ex. $k_1 \neq k_2 \in U$ mit $h(k_1) = h(k_2)$. (*Kollision*).

Kollisionsvermeidung

Bsp.:

Matrikelnummern $U = \{10^7 \ldots 10^8 - 1\}$
$|K| < 10^4$ Studenten.

Wahl der Hashfunktion:

$h_1(k) = k \text{ div } 10^4$
$h_2(k) = k \bmod 10^4 \quad \leftarrow$ besser

Allgemein:

$m \geq |K|$ und h: $K \to \{0, \ldots, m-1\}$ gleichverteilt

Wenn K nicht bekannt und $U = \mathbb{N}$:

- $h(k) = k \bmod m$, wenn m prim
- $h(k) = (k \bmod p) \bmod m$, wobei p große Primzahl und m beliebig

Bsp.:

$m = 11$, $h(k) = k \bmod 11$
Einfügen von 15, 13, 4, 14, 25

0	1	2	3	4	5	6	7	...
		13	14	15	4	25		

Kollisionsauflösung in der Tabelle

Erweiterte Hashfunktion

h: $U \times \{0, \ldots, m-1\} \to \{0, \ldots, m-1\}$

maximal m Versuche, um Element einzufügen.

Einfügen von Schlüssel k:

Durchsuche Indizes $h(k,0), h(k,1), \ldots, h(k,m-1)$ bis freier Eintrag gefunden.

$h(k,0), h(k,1), \ldots, h(k,m-1)$ heißt *Sondierungssequenz*

Suchen von Schlüssel k:

Durchsuche Indizes anhand Sondierungssequenz bis Schlüssel k gefunden.

Lineares Sondieren

für gegebene Hashfunktion $h_1(k)$ wähle erweiterte Hashfunktion $h(k,i) = (h_1(k) + i) \bmod m$.

Problem: Clusterbildung

Quadratisches Sondieren

Vermeide Cluster durch besseres Verteilen der Einträge für Schlüssel mit Hashwert $h(k)$ Tabelle.

$$h(k,i) = (h_1(k) + i^2) \bmod m$$

Problem: k_1 und k_2 mit $h_1(k_1) = h_1(k_2)$ haben gleiche Sondierungssequenz (sekundäre Cluster).

Doppeltes Hashing

zweite Hashfunktion h_2: $U \to \mathbb{N}$

$$h(k,i) = (h_1(k) + i\, h_2(k)) \bmod m$$

Sondierungssequenz hängt von Schlüssel ab; vermeidet sekundäre Cluster

Kollisionsauflösung in der Tabelle heißt auch *offene Adressierung*.

Alternative: jeder Tabelleneintrag zeigt auf eine verkettete Liste.
$\rightsquigarrow$ vermeidet Clusterbildung

Analyse

Definition 18 *Sei* $n = |K|$. $\alpha := \frac{n}{m}$ *heißt* Belegungsgrad.

Kollisionsauflösung mit Verkettung

Die durchschnittliche Listenlänge ist α.

- Im schlechtesten Fall: $h(k) = \texttt{const}$ für alle $k \in K$,
  Laufzeit für suchen, einfügen, löschen: $\Theta(n)$.

Definition 19 *Man spricht von* einfachem uniformem Hashing*, wenn für* h *gilt:*

- *Wahrscheinlichkeit, dass* $h(k) = i$*, ist* $\frac{1}{m}$ *für alle* $k \in U$*,*
  $i \in \{0, \ldots, m-1\}$.
- *Wahrscheinlichkeiten für* $h(k_1) = i_1$ *und* $h(k_2) = i_2$ *sind unabhängig.*

Theorem 4 *Bei einfachem uniformem Hashing und Kollisionsauflösung mit Verkettung ist die durchschnittliche Laufzeit*

- *für erfolglose Suche:* $\Theta(1 + \alpha)$
- *für erfolgreiche Suche:* $\Theta(1 + \alpha)$

Kollisionsauflösung mit offener Adressierung

$\alpha \leq 1$

Definition 20 *Bei* erweitertem uniformem Hashing *gilt:*

- *Sondierungssequenz ist eine Permutation von* $0, \ldots, m-1$ *für alle* $k \in U$*.*
- *Wahrscheinlichkeit, dass die Sondierungssequenz eine bestimmte Permutation ist, ist für alle* $k \in U$ *und Permutationen gleich.*

Theorem 5 *Bei erweitertem uniformem Hashing, offener Adressierung und* $\alpha < 1$ *ist der Erwartungswert für die Zahl der Sondierungen*

- *bei erfolgloser Suche:* $\leq \frac{1}{1-\alpha}$
- *bei erfolgreicher Suche:* $\leq \frac{1}{\alpha} \ln \frac{1}{1-\alpha}$

D. h. für festes α: $\mathcal{O}(1)$

Bsp.:

	erfolglos: $\frac{1}{1-\alpha}$	erfolgreich: $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$
$\alpha = 20\%$	$1,25$	$\approx 1,12$
$\alpha = 50\%$	2	$\approx 1,39$
$\alpha = 90\%$	10	$\approx 2,56$

Kapitel 6

Dynamisches Programmieren: Rucksackproblem

Bsp.: Bepacken eines LKW, Schiffs, Silizium-Chips

Rucksackproblem:

> Gegeben $K \in \mathbb{N}$ und n Gegenstände $1, \ldots, n$, wobei der Gegenstand i die Größe $k_i \in \mathbb{N}$ hat. Bestimme eine Teilmenge der Gegenstände deren Größe zu exakt K summiert, oder entscheide, dass eine solche Teilmenge nicht existiert.

Bezeichnung: $P(n, K)$

Allgemein bezeichnen wir mit $P(i, k)$, dass das Rucksackproblem für die Gegenstände $1, \ldots, i$ und Rucksack der Größe k lösbar ist.

1. Versuch:

IV: Wir können $P(n-1, K)$ lösen.

- Basisfall: $P(1, K)$
  Lösung exakt g. d. w. $k_1 = K$.
- Induktionsschritt: $P(n, K)$
  - $P(n-1, K)$ hat Lösung
    $\rightsquigarrow$ $P(n, K)$ hat Lösung
    (Gegenstand n wird nicht verwendet)

– $P(n-1, K)$ hat keine Lösung
  $\rightsquigarrow$ Gegenstand n muss verwendet werden
  $P(n, K)$ hat Lösung g. d. w. $P(n-1, K-k_n)$ hat Lösung.

2. Versuch:

IV: Wir können $P(n-1, k)$ lösen für alle $0 \leq k \leq K$.

- Basisfall: $P(1, k)$
  - für $k = 0$ immer trivial lösbar
  - für $k > 0$: $P(1, k)$ lösbar g. d. w. $k_1 = k$
- Induktionsschritt: $P(n, k)$
  Reduktion auf $P(n-1, k)$ und $\underbrace{P(n-1, k-k_n)}_{\text{wenn } k-k_n \geq 0}$.

  Beide Teilprobleme ergeben sich aus der IV.

Effizienz:

Ein Problem der Größe n reduziert auf zwei Probleme der Größe $n-1$.

$\rightsquigarrow$ exponentieller Aufwand!

Aber: es gibt nur $n\,K$ verschiedene Probleme

$\rightsquigarrow$ Speichere einmal berechnete Lösungen in einer Tabelle

$\rightsquigarrow$ Aufwand: $\mathcal{O}(n\,K)$

Technik: *dynamisches Programmieren*

Bsp.:

Rucksack der Größe $K = 16$
Gegenstände: $k_1 = 2, k_2 = 3, k_3 = 5, k_4 = 6$

Tabelleneinträge:

–	$\widehat{=}$	keine Lösung existiert
O	$\widehat{=}$	Lösung ohne i-ten Gegenstand
I	$\widehat{=}$	Lösung mit i-ten Gegenstand existiert.

$k =$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$k_1 = 2$	O	–	I	–	–	–	–	–	–	–	–	–	–	–	–	–	–
$k_2 = 3$	O	–	O	I	–	I	–	–	–	–	–	–	–	–	–	–	–
$k_3 = 5$	O	–	O	O	–	O/I	–	I	I	–	I	–	–	–	–	–	–
$k_4 = 6$	O	–	O	O	–	O	I	O	O/I	I	O	I	–	I	I	–	I

Lösung existiert.
Enthaltene Gegenstände der Größen $2, 3, 5, 6$.

Algorithmus 15 RUCKSACK

Eingabe: S (Feld der Größe n, welches die Größe der Gegenstände enthält)
K (Größe des Rucksacks)
Ausgabe: P (2-dim Feld, wobei
$P[i][k].exists = true$ wenn Lösung für $P(i,k)$ existiert,
$P[i][k].belong = true$ wenn Lösung für $P(i,k)$ den i-ten Gegenstand enthält.)

```
1: begin
2:   P[0][0].exists := true;
3:   for k := 1 to K do
4:     P[0][k].exists := false;
5:   for i := 1 to n do
6:     for k := 0 to K do
7:       P[i][k].exists := false;                    /* als default */
8:       if P[i − 1][k].exists then
9:         P[i][k].exists := true;
10:        P[i][k].belong := false;
11:      else if k − S[i] ≥ 0 then
12:        if P[i − 1][k − S[i]].exists then
13:          P[i][k].exists := true;
14:          P[i][k].belong := true;
15: end
```

Kapitel 7

Graphenalgorithmen

Definition 21 Graph $G = (V, E)$, *wobei*

- V *Menge von* Knoten *(beliebig)*
- E *Menge von* Kanten (v, w), *wobei* $v, w \in V$, $v \neq w$.

Besteht E aus ungeordneten Paaren (d. h. (v, w) und (w, v) werden nicht unterschieden), so heißt G ungerichtet. *Besteht E aus geordneten Paaren, so heißt G* gerichtet.

Bemerkung: Eine Kante (v, v) heißt *Schleife*. Unsere Graphen sind schleifenfrei.

Wichtige Begriffe:

Sei $G = (V, E)$ Graph.

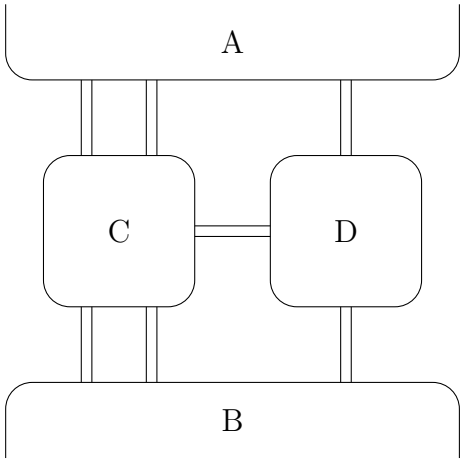
- $(v, w) \in E$ heißt *Kante von v nach w*.
- Für $v \in V$ ist $d(v) = |\{(u, w) \in E \mid u = v \vee u = w\}|$ der *Grad* von v. Im gerichteten Graphen:

  $|\{(w, v) \in E\}|$ *Eingangsgrad* von v
  $|\{(v, w) \in E\}|$ *Ausgangsgrad* von v

- Folge von Knoten $v_1, \ldots, v_k \in V$ heißt *Pfad von v_1 nach v_k*, wenn $(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k) \in E$. Ein Pfad ist *geschlossen* oder *zyklisch*, wenn $v_1 = v_k$; *einfach* wenn $v_1, \ldots, v_k$ paarweise verschieden (Ausnahme: $v_1 = v_k$ erlaubt).
- Ein Graph $H = (U, F)$ ist ein *Teilgraph* von G wenn $U \subseteq V$, $F \subseteq E$.

- Ein ungerichteter Graph heißt *zusammenhängend*, wenn es für alle $v, w \in V$, $v \neq w$ einen Pfad von v nach w gibt.
  Ein gerichteter Graph heißt *stark zusammenhängend*, wenn es für alle $v, w \in V$, $v \neq w$ einen Pfad von v nach w gibt.
- Ein gerichteter Graph heißt *zusammenhängend*, wenn er als ungerichteter Graph zusammenhängend ist.

7.1 Eulersche Graphen



Königsberger Brückenproblem (Euler 1736): Gibt es einen Weg, der zum Ausgangspunkt zurückführt, und auf dem jede Brücke genau einmal betreten wird?

Definition 22 *Ein Pfad ist ein* Eulerscher Pfad*, wenn jedes $e \in E$ genau einmal vorkommt. Dito* Eulerscher Kreis*, wenn der Pfad geschlossen ist.*
Ein Graph heißt Eulerscher Graph*, wenn es einen Eulerschen Kreis gibt.*

Theorem 6 (Euler) *In einem ungerichteten, zusammenhängenden Graphen G gibt es einen Eulerschen Kreis P genau dann, wenn $d(v) =$* gerade *für alle $v \in V$.*

Beweis:

$\Rightarrow$: Jeder Knoten wird so oft betreten wie verlassen, und jeweils über verschiedene Kanten. $\rightsquigarrow d(v)$ gerade für alle $v \in V$.

$\Leftarrow$: Beweis durch Konstruktion eines Eulerschen Kreises.

Beweis durch Induktion über Zahl m der Kanten.

$m = 0$: Nichts zu tun.

$m > 0$: Induktionsvoraussetzung: für alle ungerichteten, zusammenhängenden Graphen $G' = (V', E')$ mit $|E'| < m$ und $d(v)$ gerade für alle $v \in V'$ können wir einen Eulerschen Kreis konstruieren.

Sei $G = (V, E)$ ungerichteter, zusammenhängender Graph, $|E| = m$, $d(v)$ gerade für alle $v \in V$.

Wähle bel. $v_1 \in V$. Da G zusammenhängend, existiert $v_2 \in V$ mit $(v_1, v_2) \in E$. Da $d(v_2)$ gerade, existiert weitere Kante $(v_2, v_3) \in E$. Da $d(v_3)$ gerade, existiert weitere Kante $(v_3, v_4) \in E$, etc. Erhalte Pfad $v_1, \ldots, v_i$ auf dem jede Kante nur einmal vorkommt. Entweder $v_i = v_1$, oder es existiert weitere Kante $(v_i, v_{i+1}) \in E$.

Ergebnis: geschlossener Pfad $P = v_1, \ldots, v_j$, in dem keine Kante doppelt vorkommt.

Sei $G' = (V, E')$, wobei $E' = E \setminus$ Kanten in P.

- Knoten in G' haben geraden Grad.
- G' nicht notwendig zusammenhängend.

Betr. zusammenhängende Komponenten $G_1, \ldots, G_k$ von G'. Nach Induktionsvoraussetzung existieren geschlossene Eulersche Pfade P_i in $G_i = (V_i, E_i)$, $i = 1 \ldots k$.

Zusammensetzen von $P, P_1, \ldots, P_k$ zu Eulerschem Kreis von G: Ausgehend von v_1 folge Kanten in P bis Knoten v_i erreicht mit $v_{i_1} \in G_{j_1}$ für ein j_1. Ausgehend von v_{i_1} folge dem gesamten Pfad P_{j_1}. Von v_{i_1} folge Kanten in P bis $v_{i_2} \in G_{j_2}$ erreicht. Folge P_{j_2}. Etc. Schließlich wird über P v_1 erreicht. Der Pfad ist geschlossen.

Es wurden alle Komponenten $G_1, \ldots, G_k$ erreicht, demnach, da G zusammenhängend, enthält jedes G_i einen Knoten aus P.

Also enthält der konstruierte Pfad alle Kanten aus $E \quad \rightsquigarrow$ Eulerscher Kreis.

□

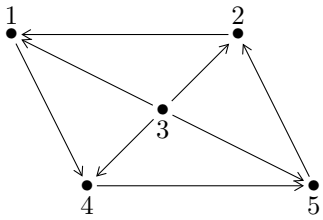
Bemerkung:

Als Algorithmus ist die Konstruktion noch unvollständig. Es fehlen:

- Identifizieren von Zusammenhangskomponenten
- Systematisches Durchlaufen eines Graphen.

7.2 Repräsentation von Graphen

Bsp.:



$G = (V, E)$
$|V| = n$
$|E| = m$

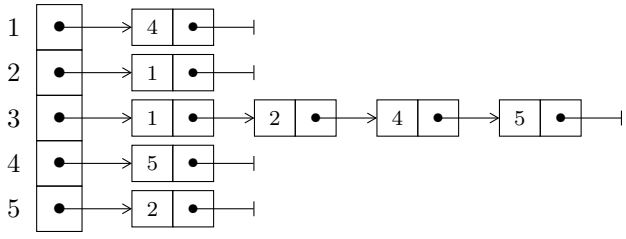
Adjazenzmatrix $n \times n$-Matrix A, $A_{ij} \in \{0, 1\}$

$A_{ij} = 1$ g. d. w. $(i, j) \in E$

	1	2	3	4	5
1				1	
2	1				
3	1	1		1	1
4					1
5		1			

Adjazenzliste Für jedes $v \in V$ eine Liste

der Knoten w mit $(v, w) \in E$.



7.3 Durchlaufen von Graphen

$\rightsquigarrow$ Durchlaufen aller Kanten

7.3.1 Tiefensuche (depth-first search)

Algorithmus 16 TIEFENSUCHE (EINFACHE VERSION)

Eingabe: $G = (V, E)$, ein Graph; v, ein Knoten von G
Ausgabe: abhängig von der Anwendung

```
1: begin
2:    markiere v;
3:    preWork auf v;
4:    for each (v, w) ∈ E do
5:       if w nicht markiert then
6:          TIEFENSUCHE(G, w);
7:       postWork für (v, w);
8: end
```

Bemerkung:

Manchmal `postWork` für (v, w) nur falls w nicht markiert war.

Strukturieren des Graphen mit Tiefensuche

$\rightsquigarrow$ Nummerierung der Knoten in Reihenfolge der Tiefensuche

$i := 1$;
TIEFENSUCHE(G, v);
(mit preWork:
    $v.DFS := i$;
    $i := i + 1$;)

$\rightsquigarrow$ Berechnung des Tiefensuchbaums

$F :=$ leere Menge von Kanten;
TIEFENSUCHE(G, v);
(mit `postWork`:
    if w war unmarkiert
    then füge Kante (v, w) zu F hinzu;)

Laufzeit jeweils $\mathcal{O}(|V| + |E|)$

Bemerkung:

Einfache Tiefensuche erreicht nicht notwendig alle Knoten. Wiederhole mit bisher unmarkiertem Knoten als Starknoten.

$\rightsquigarrow$ *Tiefensuche* meint iterierte Variante der einfachen Tiefensuche.

Tiefensuchbaum $\rightsquigarrow$ Menge von Tiefensuchbäumen (Wald)

Lemma 4 *Ist G ein ungerichteter oder gerichteter Graph, so werden durch Tiefensuche*

a) alle Knoten markiert und

b) `postWork` *für alle Kanten wenigstens einmal ausgeführt*

Beweis:

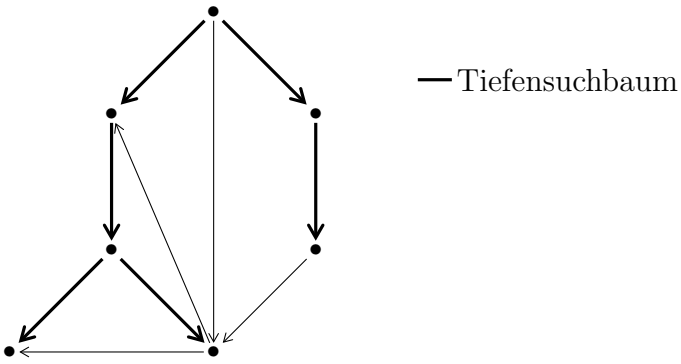
a) Nach Konstruktion

b) Für jeden Knoten wird `postWork` für alle abgehenden Kanten ausgeführt.

□

Tiefensuche in gerichteten Graphen

Sei $G = (V, E)$ gerichteter Graph und $T = (V, F)$ sein Tiefensuchbaum.



Definition 23 *Sei $(v, w) \in E$. (v, w) heißt*

- Baumkante*, wenn $(v, w) \in F$*
- Vorwärtskante*, wenn $(v, w) \notin F$, v und w auf dem gleichen Ast von T und $v.DFS < w.DFS$*
- Rückwärtskante*, wenn $(v, w) \notin F$, v und w auf dem gleichen Ast von T und $v.DFS > w.DFS$*
- Querkante*, wenn v und w auf verschiedenen Ästen von T liegen.*

Lemma 5 *Ist $(v, w) \in E$ mit $v.DFS < w.DFS$, dann ist w ein Nachkomme von v in T.*

Beweis:

In Tiefensuche wird v bevor w markiert. Nachdem v markiert, erfolgt Tiefensuche ausgehend von v. Da $(v, w) \in E$ wird w während dieser Tiefensuche markiert.
Damit liegen v und w in T auf dem gleichen Ast.

□

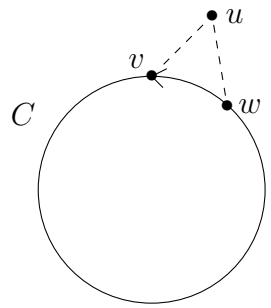
Hieraus folgt, dass für eine Querkante $(v, w) \in E$ gilt $v.DFS > w.DFS$.

Finden von Zyklen in gerichteten Graphen

Lemma 6 *$G = (V, E)$ gerichteter Graph, T ein Tiefensuchbaum von G. G enthält einen Zyklus genau dann, wenn G eine Rückwärtskante (in Bezug auf T) enthält.*

Beweis:

$\Leftarrow$: Rückwärtskante bildet Zyklus mit Teilast von T.
$\Rightarrow$: Sei C der Zyklus, v der Knoten in C mit der kleinsten Tiefensuchnummer.



Betr. Kante (w, v) aus C:

- $v.DFS < w.DFS$, also ist (w, v) keine Baum- oder Vorwärtskante
- Ann.: (w, v) ist Querkante. v und w in verschiedenen Tiefensuch-Teilbäumen von u.
  Aber: es gibt einen Pfad von v nach w.

Also ist (w, v) eine Rückwärtskante in Bezug auf T.

□

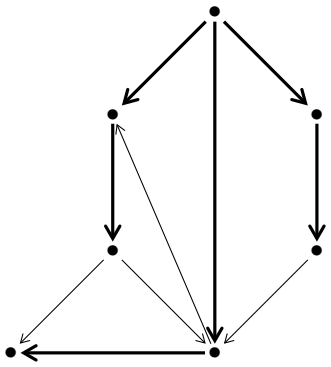
Algorithmus 17 ZYKLUS

Eingabe: $G = (V, E)$, ein gerichteter Graph
Ausgabe: Zyklus, Zyklus = true g. d. w. G einen Zyklus enthält

```
1: begin
2:   for each x ∈ V do
3:     x.on_the_path := false;
4:   Zyklus := false;

5:   Tiefensuche auf G, ausgehend von einem beliebigen Knoten, mit folgendem preWork und postWork:

6:   preWork:
7:     v.on_the_path := true;
8:     /* x.on_the_path ist true, wenn x auf dem Pfad von der Wurzel zum aktuellen Knoten liegt */

9:   postWork:
10:    if w.on_the_path then
11:      Zyklus := true;
12:      halt;
13:    if w ist der letzte Knoten, der von v ausgeht then
14:      v.on_the_path := false;
15: end
```

7.3.2 Breitensuche (breadth-first search)

Graph wird „ebenenweise“ durchlaufen.



Breitensuche

Ebene eines Knotens v ist die Länge des Pfads von der Wurzel nach v im Breitensuchbaum.

Algorithmus 18 BREITENSUCHE

Eingabe: $G = (V, E)$, zusammenhängender Graph, $v \in V$
Ausgabe: abhängig von der Anwendung; Breitensuchbaum T

```
1: begin
2:   markiere v;
3:   füge v in FIFO-Warteschlange Q ein;
4:   while Q nicht leer do
5:     entferne ersten Knoten w aus Q;
6:     preWork auf w;
7:     for each (w, x) ∈ E do
8:       if x unmarkiert then
9:         markiere x;
10:        füge (w, x) zu T hinzu;
11:        füge x in Q ein;
12: end
```

Literaturverzeichnis

[1] Udi Manber. *Introduction to Algorithms: A Creative Approach*, Addison-Wesley, 1989

[2] Thomas H. Cormen et al. *Introduction to Algorithms*, Second Edition, MIT Press, 2001

[3] Brian W. Kernighan, Dennis M. Ritchie. *Programmieren in C*, Zweite Ausgabe ANSI C, Hanser, München, 1990