Informatik I

Algorithmen und Datenstrukturen

Inhalt

INHALT 2

VORWORT 6

0 EINFÜHRUNG 7

0.1 WAS IST INFORMATIK? 7
0.2 DIE ENTWICKLUNG VON RECHNERN UND RECHENMETHODEN 7

1 ALGORITHMEN UND DATENSTRUKTUREN 9

1.1 EINFÜHRUNG 9
1.1.1 DEFINITIONEN 9
1.1.2 EIN EINFACHES SORTIERPROBLEM – INSERTION SORT 9
- Code von Insertion Sort 9
- Laufzeitanalyse von Insertion Sort 10

1.2 ASYMPTOTISCHE EFFIZIENZ VON ALGORITHMEN 11
1.2.1 DEFINITIONEN 11
1.2.2 FOLGERUNGEN 12
1.2.3 MASTER-THEOREM (SATZ ZUR BERECHNUNG VON REKURSIV DEFINIERTEN LAUFZEITEN) 12

1.3 VORGEHENSWEISE VON ALGORITHMEN 12
1.3.1 MERGE SORT 13
- Code von Merge Sort 13
- Code von Merge 13
- Vergleich von Insertion Sort und Merge Sort 14

1.3.2 MAXIMALE TEILSUMME 14
- Naiver Algorithmus 14
- Divide&conquer 15
- Scanline 16

2 SORTIEREN 18

2.1 SELECTION SORT 18
- Code von Selection Sort 18

2.2 BUBBLE SORT 19
- Code einer einfachen Bubblesort – Variante 19
- Code einer schnellerer Bubblesort – Variante 19
- Vergleich Sortieralgorithmen 20

2.3 HEAP SORT 20
2.3.1 DEFINITIONEN 20
- Binärer Baum 20
- Heap 21

2.3.2 DIE BESTANDTEILE VON HEAP SORT 21
- Heapify 21

	- Build Heap	21
	- Heap Sort	22
2.4	**QUICKSORT**	**22**
	- Vorgehensweise	22
	- Code von Quicksort	23
	- Eigenschaften von Quicksort	23
2.5	**COUNTING SORT**	**24**
	- Code von Counting Sort	24
	- Eigenschaften von Counting Sort	25
2.6	**RADIX SORT**	**25**
2.7	**RECHENEXPERIMENTE**	**26**
	- Rechenzeiten und Laufzeiten	26
	- Schlußfolgerungen	26
3	**ELEMENTARE ABSTRAKTE DATENTYPEN**	**27**
3.1	**EINFÜHRUNG**	**27**
3.2	**DER STACK**	**27**
	- Struktur des Stacks	27
	- Code der Stack-Operationen	27
3.3	**DIE QUEUE**	**28**
	- Struktur der Queue	28
	- Code der Queue-Operationen	28
3.4	**LISTEN**	**29**
3.4.1	STRUKTUR VON LISTEN	29
3.4.2	SEQUENTIELLE LISTEN	29
3.4.3	DOPPELT VERKETTETE LISTEN	30
	- Struktur	30
	- Codes der Listenoperationen	30
3.5	**IMPLEMENTIERUNG VON ABSTRAKTEN DATENTYPEN**	**32**
3.5.1	IMPLEMENTATION IN FORTRAN77	32
3.5.2	IMPLEMENTATION IN C	32
3.5.3	IMPLEMENTATION IN C++	33
3.6	**BÄUME**	**33**
3.6.1	EINFÜHRUNG	33
3.6.2	DATENSTRUKTUR BINÄRER BÄUME	34
	- Ein typischer binärer Baum	34
3.6.3	VERALLGEMEINERUNG AUF BÄUME MIT MEHR ALS ZWEI SÖHNEN	34
	- Ein Baum mit mehr als zwei Söhnen	35
4	**ELEMENTARE SUCHVERFAHREN**	**36**
4.1	**AUSWAHLPROBLEM**	**36**
4.2	**SUCHE IN LINEAREN LISTEN**	**36**
4.2.1	SEQUENTIELLE SUCHE	36
4.2.2	BINÄRE SUCHE	37
4.2.3	INTERPOLATIONSSUCHE	37
5	**SUCHEN IN BÄUMEN**	**38**
5.1	**EINFÜHRUNG**	**38**
5.2	**BINÄRE SUCHBÄUME**	**38**
5.2.1	DEFINITION	38
5.2.2	CODES DER OPERATIONEN EINES BSB	39
	- Ausgabe des gesamten Baums	39

	- Suche	39
	- Finden des Minimums bzw. Maximums	39
	- Finden des Vorgängers und des Nachfolgers	40
	- Einfügen eines Knotens	40
	- Löschen eines Knotens	40
	- Einfügen	40
	- Löschen	41
5.2.3	KOMPLEXITÄT DER BASISOPERATIONEN BEI BINÄREN SUCHBÄUMEN	41
5.3	**ROT-SCHWARZ-BÄUME (RSB)**	**42**
5.3.1	ALLGEMEINES	42
	- Definitionen	42
	- Beispiel eines RSB	42
5.3.2	DIE MAXIMALE HÖHE EINES RSB	43
	- Satz	43
	- Beweis	43
5.3.3	EINFÜGEN IN EINEM RSB	43
	- Rotation	43
	- Einfügen	44
	- Laufzeitanalyse	45
5.3.4	ENTFERNEN EINES KNOTENS	46
5.3.5	BEISPIEL FÜR DAS SUKZESSIVE EINFÜGEN EINER FOLGE IN EINEN RSB	46
6	**HASHTABELLEN**	**47**
6.1	**DIREKTE ADRESSIERUNG**	**47**
6.2	**HASHTABELLEN**	**48**
6.2.1	EINFACHE HASHTABELLEN	48
6.2.2	VERKETTETE HASHTABELLEN („CHAINED HASHING“)	48
6.2.3	ANDERE HASHFUNKTIONEN	49
6.3	**OFFENE HASHFUNKTIONEN**	**49**
6.3.1	LINEARES SONDIEREN	49
6.3.2	QUADRATISCHES SONDIEREN	50
6.3.3	DOUBLE HASHING	50
6.3.4	LAUFZEITENANALYSEN	51
6.3.5	VERBESSERUNG VON BREUT	51
7	**GRAPHEN**	**52**
7.1	**EIGENSCHAFTEN, DARSTELLUNG UND SPEICHERUNG VON GRAPHEN**	**52**
7.1.1	DEFINITIONEN	52
7.1.2	SPEICHERUNG VON GRAPHEN	53
	- Adjanzenz-Matritzen	53
	- Adjazenz-Listen	54
7.2	**BREADTH-FIRST SEARCH, „BREITENSUCHE“**	**55**
7.2.1	HINTERGRUND	55
7.2.2	VORGEHEN	55
	- Beispiel	56
	- Der BFS-Baum	56
	- Aufwand	56
7.2.3	FINDEN DER KÜRZESTEN VERBINDUNG VON ZWEI KNOTEN	57
	- Code von print_path	57
7.2.4	SATZ	57
7.2.5	ENTDECKEN EINES GESCHLOSSENEN KREISES	57
7.3	**DEEP FIRST SEARCH, „TIEFENSUCHE“**	**58**

7.3.1 HINTERGRUND 58

7.3.2 VORGEHEN 58

- Algorithmus der sequenziellen Variante 58
- Code der rekursiven Variante 58

7.3.3 LAUFZEIT 59

7.3.4 BEISPIEL EINER DFS 59

7.3.5 DER DFS-WALD 59

- Satz 60

7.3.6 TOPOLOGISCHE SORTIERUNG VON GRAPHEN 60

- Beispiel 1 60
- Beispiel 2 – Das morgendliche Anziehen 61

7.4 MINIMAL SPANNING TREES 62

7.4.1 EINFÜHRUNG 62

- Definition 62
- Beispiel 1 62
- Beispiel 2 62

7.4.2 ERSTELLUNG EINES MST 63

- Vorgehen 63
- Finden einer sicheren Kante 63
- Satz 63
- Einfacher Algorithmus („greedy algorithm“) 64

LITERATUR 66

INDEX 67

Vorwort

0 Einführung

0.1 Was ist Informatik?

Informatik (engl.: computer science, information technology) ist ein Kunstwort, daß aus dem lateinischen Wort „Information“ (=Darstellung) und dem griechischen Wort „Automatik“ (=selbstständiger Ablauf) entstanden ist.
Informatik ist die Wissenschaft von Computern und ihrem Einsatz. Sie hat ein weites Spektrum von Teildisziplinen: von theoretischen Grundlagen bis zu praktischen Anwendungen, von hardwarenaher, systemnaher Software bis hin zu allgemeinen Berechnungsverfahren und Anwendungssoftware. Sie läßt sich in folgende Gebiete einteilen:

- **Technische Informatik** beschäftigt sich mit der Konstruktion der Hardware (Prozessoren, Speicher, Netzwerke etc.) und Rechnerarchitektur im Allgemeinen.
- **Praktische Informatik** schlägt die Brücke zwischen Hardware und Anwendungssoftware und setzt sich mit so grundlegender Software wie Betriebsystemen, Compilern und Datenbanksystemen auseinander.
- **Angewandte Informatik** hat Softwareentwicklung und den Einsatz der Software zum Inhalt. Die Themen erstrecken sich von Simulation und Optimierung über Steuerung und Visualisierung bis hin zu Computergraphik und –animation.
- **Theoretische Informatik** schließlich befaßt sich mit den abstrakten mathematischen Hintergründen der Informatik, z.B. mit formalen Sprachen, Automatentheorie und Komplexitätstheorie.

0.2 Die Entwicklung von Rechnern und Rechenmethoden

- Frühzeitlich: Rechenhilfen wie Abakus, Suanpan(China, 11. Jhd. v.Chr.), Soropan(Japan, 7. Jhd. v. Chr.), Stschoty(Rußland)
- 1624: Rechenschieber, nach der Entdeckung des Logarithmus durch Lord Napier im 16.Jhd.
- 16.-17. Jhd.: Mechanische Rechner, Multiplizierer, Addierer (Pascal, Leibnitz)
- 19. Jhd.: Addiermaschinen mit Tastatur
- 1946: Elektronische Rechner der ersten Generation, basierend auf Röhren. (Z22, ENIAC, IBM 650)
- 1957: Elektronische Rechner der zweiten Generation, basierend auf Transistoren. (IBM 1400, Siemens 2002)
- 1964: Elektronische Rechner der dritten Generation, basierend auf integrierten Schaltungen. (CDC 3000, Univac 9000, IBM 360, Siemens 4004)
- 1975: Elektronische Rechner der vierten Generation, mit Mehrprozessorarchitektur und Großintegration. (CDC Cyber, IBM 370, Siemens 7700)
- 1978: Mit dem Prozessor 8086 beginnt die Firma INTEL eine Serie von Prozessoren, die sich bis zum Pentium III im Jahr 1999 fortsetzt.
- 1981: IBM baut den ersten PC.
- 1993: neue und schnelle Prozessoren kommen auf den Markt: Der Pentium mit 66MHz, Der DEC Alpha mit 150MHz und der PowerPC mit 66MHz.
- 1995: Pentium6 mit 150MHz, Alpha 21164 mit 300MHz, PowerPC620 mit 133MHz und MIPS T5 mit 200MHz.

- Ende 90er: Der Trend geht mehr und mehr zu PC-Geräten. Große Rechenleistung wird durch Parallelisierung erreicht). Es entstehen mehr und mehr Spezialrechner, z.B. ausschließlich für Grafikanwendungen.

1 Algorithmen und Datenstrukturen - Grundlagen

1.1 Einführung

Kein Rechner arbeitet ohne Programm. Zitat Wirth[1]: “Programme sind letztendlich konkrete Formulierungen abstrakter **Algorithmen**, die sich auf bestimmte Darstellungen und **Datenstrukturen** stützen“.
Im Folgenden finden immer wieder einige grundlegende Begriffe der Informatik Verwendung, die jetzt definiert werden.

1.1.1 Definitionen

- **Daten und Datenstrukturen** sind die codierte Darstellung von Informationen, im Speziellen Input- und Outputdaten.
- Ein **Algorithmus** ist eine Berechnungsvorschrift oder Folge von Rechenoperationen, die aus einer Menge von Inputdaten eine andere Menge von Outputdaten erzeugt, z.B. Kochrezepte, Bastelanleitungen oder Partituren. Wenn nicht anders vermerkt, wird im folgenden unter Algorithmus immer ein Code verstanden, der von einem Computer ausgeführt werden kann, d.h. der auf konkrete Daten angewendet werden kann.
- **Operationen**: Man unterscheidet zwischen sog. „elementaren“ Operationen, wie z.B. Grundrechenarten, Abrufen und Speichern von Informationen (in beschränktem Maße auch Lesen und Schreiben von Daten), Vergleichsoperationen, logische Operationen und Sprungoperationen und den sog. „komplexen“ Operationen, die sich aus den elementaren Operationen zusammensetzen.
- Ein **Problem** bzw. eine **Problemklasse** spezifiziert den Output zu einem gegebenen Input in allgemeiner und impliziter Form.
- Von einer **Probleminstanz** oder einem **Problembeispiel** spricht man, wenn für ein Problem konkrete Daten vorliegen.
- Formalisierung: z.B. Rechner funktioniert als universeller Modellrechner (z.B. Turing-Maschine), findet vor allem in der theoretische Informatik Anwendung.
- Die **Laufzeit** ist der Zeitbedarf zum Ausführen eines Algorithmus in Abhängigkeit vom Umfang der Inputdaten.

1.1.2 Ein einfaches Sortierproblem – Insertion Sort

Input: Ganze Zahlen („integer“), die ein Array von ganzen Zahlen $a_1, a_2, \ldots, a_n$ bilden.
Output: Umsortierung der Folge zu $a_1', a_2', \ldots, a_n'$, so daß $a_1' < a_2' < \ldots < a_n'$.
Beispiel („Probleminstanz“): Folge <32, 25, 13, 48, 39>

<32, 25, 13, 48, 39> ⇒ <25, 32, 13, 48, 39> ⇒ <13, 25, 32, 48, 39> ⇒ <13, 25, 32, 39, 48>

Es gibt verschiedene Lösungsalgorithmen, der einfachste Ansatz nennt sich **Insertion Sort**.

Code von Insertion Sort

```
insertion sort(A,n) {
for j=2 to n
{
     key = A[j];
     i = j–1;
     while ( i≥1 && A[i]>key )
```

[1] Computerguru, entwickelte die Programmiersprachen Pascal, Modula und Oberon.

```
        {
             A[i+1] = A[i];
             i--;
        }
        A[i+1] = key;
  }}
```

Bei der Analyse von Algorithmen interessieren uns vor allem der Zeitbedarf (=Laufzeit) und der Speicherbedarf. Bei komplexen Algorithmen werden dann auch andere Kriterien relevant wie Zuverlässigkeit, universelle Einsetzbarkeit und Genauigkeit, auf die in dieser Vorlesung aber nicht eingegangen wird.

Laufzeitanalyse von Insertion Sort

Bei der Analyse der Laufzeit wird davon ausgegangen, daß jeweils nur eine Operation (=Instruktion) zu einer Zeit ausgeführt wird, und daß das Ausführen einer Operation immer gleich viel Zeit kostet. Ein System, daß diese Eigenschaften hat, wird auch als „random access machine“ (=RAM) bezeichnet.

Tatsächlich gibt es allerdings Mikroparallelität auf dem Chip, d.h. selbst auf nur einem Prozessor werden mehrere Operationen zur gleichen Zeit ausgeführt. Außerdem sind bei einem realen Prozessor die Ausführungszeiten sequenzabhängig(„pipelining“), d.h. je nach Reihenfolge der Instruktionen ergeben sich unterschiedliche Ausführungszeiten.

Zeile	**Anzahl**	**Kosten**
(1)	n	c_1
(2)	$(n-1)$	c_2
(3)	$(n-1)$	c_3
(4)	$\sum_{j=2}^{n} t_j$	c_4
(5)	$\sum_{j=2}^{n} (t_j - 1)$	c_5
(6)	$\sum_{j=2}^{n} (t_j - 1)$	c_6
(7)	$(n-1)$	c_7

c_k: Zeitaufwand für das Ausführen der k-ten Programmzeile

t_j: Anzahl der while-Schritte für jedes j

Somit ergibt sich für die Laufzeit von „Insertion Sort“ allgemein:

$$T(n) = c_1 n + (c_2 + c_3 + c_7)(n-1) + c_4 \sum_{j=2}^{n} t_j + (c_5 + c_6) \sum_{j=2}^{n} (t_j - 1)$$

Im besten Fall („best case“) sind alle Zahlen sind bereits richtig sortiert ($t_j = 1$):

$$T(n) = (c_1 + c_2 + c_3 + c_4 + c_7) \cdot n - (c_2 + c_3 + c_4 + c_7) \Rightarrow T(n) = \hat{a}n + \hat{b}$$

Dieser Ausdruck beschreibt eine Gerade. Die Laufzeit ist hier also „linear“ abhängig von n. Man spricht auch von einer Laufzeit der Ordnung $\Theta(n)$.

Im schlechtesten Fall dagegen(„worst case“) sind alle Zahlen in absteigender Reihenfolge sortiert ($t_j = j$). Unter Anwendung der Rechenregeln[1]:

$$\sum_{i=2}^{n} i = \frac{n(n-1))}{2} - 1 \qquad \sum_{i=2}^{n} (i-1) = \frac{n(n-1)}{2}$$

ergibt sich für die Laufzeit dann:

[1] nach Euler

$$T(n) = \left(\frac{c_4 + c_5 + c_6}{2}\right)n^2 + (c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7)n - (c_2 + c_3 + c_4 + c_7)$$

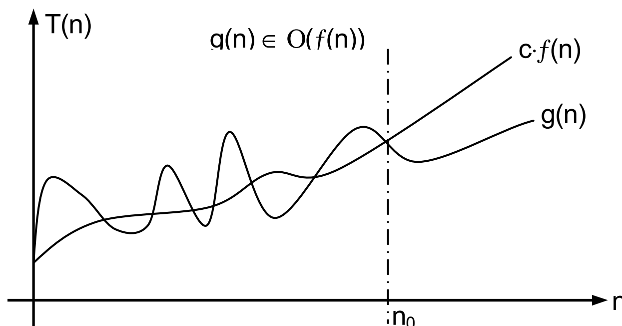
$\Rightarrow T(n) = an^2 + bn + c$ Dieser Ausdruck beschreibt eine Parabel bzw. ein Polynom zweiter Ordnung. Deshalb spricht man auch von „quadratischer “ Laufzeit bzw. von einer Laufzeit der Ordnung $\Theta(n^2)$.
Die durchschnittliche Laufzeit ist das Mittel über die Laufzeit aller möglichen Probleminstanzen der Problemgröße n. Die durchschnittliche Laufzeit kann manchmal erheblich besser sein als der worst case, in anderen Fällen ist sie ähnlich schlecht.
In diesem Fall ist im Mittel $t_j = j/2$ und es ergibt sich: $T(n) = \overline{a}n^2 + \overline{b}n + \overline{c}$, also ebenfalls eine quadratische Laufzeit!
Im Normalfall ist die Mittelung nicht so einfach und man muß sich einiger Methoden aus der Statistik bedienen, um zu einem Ergebnis zu kommen.

1.2 Asymptotische Effizienz von Algorithmen

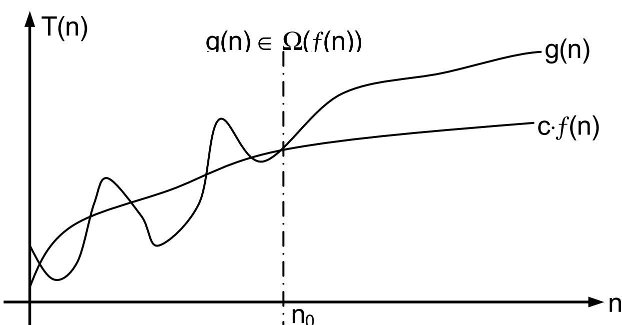
Asymptotische Effizienz bedeutet: Untersuchung des Verhaltens eines Algorithmus für $n \rightarrow \infty$. Ein asymptitisch effizienter Algorithmus stellt im Normalfall die beste Wahl dar, außer man verwendet eine sehr kleine Menge an Inputdaten[1].

1.2.1 Definitionen

- **asymptotische obere Schranke**: $O(f(n)) = \{g: \angle \rightarrow 3^+ \mid \exists n_0 \exists c>0 : \forall n \geq n_0: g(n) \leq c \cdot f(n)\}$ „$O(f(n))$ ist die Menge aller Funktionen g(n), die durch $c \cdot f(n)$ nach oben beschränkt sind.“



- **asymptotische untere Schranke**: $\Omega(f(n)) = \{g: \angle \rightarrow 3^+ \mid \exists n_0 \exists c>0 : \forall n \geq n_0: g(n) \geq c \cdot f(n)\}$ „$\Omega(f(n))$ ist die Menge aller Funktionen g(n), die durch $c \cdot f(n)$ nach unten beschränkt sind.“



- **genaue Wachstumsrate**: $\Theta(f(n)) = \{g: \angle \rightarrow 3^+ \mid \exists n_0 \exists c_1, c_2>0 : \forall n \geq n_0: c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n)\}$ „$\Theta(f(n))$ ist die Menge aller Funktionen g(n), die durch $c_1 \cdot f(n)$ nach unten und durch $c_2 \cdot f(n)$ nach oben beschränkt sind.“

[1] s. 2.2 „Vergleich Sortieralgorithmen“

1.2.2 Folgerungen

$$g(n) \in \Theta(f(n)) \Leftrightarrow g(n) \in O(f(n)) \text{ and } g(n) \in \Omega(f(n))$$[1]

„Wenn $f(n)$ obere und untere Schranke von g(n) ist, dann geschreibt $f(n)$ auch die genaue Wachtumsrate von g(n).„
Oder äquivalent: „Wenn g(n) die genaue Wachstumsrate $f(n)$ hat, dann existieren sowohl eine obere wie auch eine untere Schranke mit der Ordnung $f(n)$."

Die obere und untere Schranken sind nicht eng, d.h. wenn $T(n) \in O(n^2)$ so gilt auch $T(n) \in O(n^3)$ oder wenn $T(n) \in \Omega(n^2)$ so gilt auch $T(n) \in \Omega(n)$.

Ist z.B. $T(n) = an^2 + bn + c$ (quadratisches Wachstum), dann gilt $T(n) \in \Theta(n^2)$ und deshalb auch $T(n) \in O(n^2)$ und $T(n) \in \Omega(n^2)$.

Was haben O, Ω und Θ mit best case und worst case zu tun? Aussagen über die obere und untere Schranke lassen sich relativ leicht machen. Die genaue Wachstumsrate dagegen bezieht sich immer auf einen speziellen Satz von Inputdaten (in unserem Beispiel die Annahme $t_j = 1$) und läßt sich deshalb i.d.R. nicht allgemeingültig bestimmen.

1.2.3 Master-Theorem (Satz zur Berechnung von rekursiv definierten Laufzeiten)

Zur Berechnung von rekursiv definierten Laufzeiten gilt folgender **Satz**[2]**:**

Sei $a \geq 1$, $b > 1$ und $\varepsilon > 0$. Der Aufwand genüge der Rekursion $T(n) = a \cdot T(n/b) + D(n)$. Dann gilt:

$$D(n) \in O\left(n^{\log_b(a)-\varepsilon}\right) \Rightarrow T(n) \in \Theta\left(n^{\log_b a}\right)$$

$$D(n) \in \Theta\left(n^{\log_b(a)}\right) \Rightarrow T(n) \in \Theta\left(n^{\log_b a} \cdot \log_b n\right)$$

$$D(n) \in \Omega\left(n^{\log_b(a)+\varepsilon}\right) \text{ und } a \cdot D\left(\tfrac{n}{b}\right) \leq c \cdot D(n) \text{ für ein } c < 1 \Rightarrow T(n) \in \Theta(D(n))$$

1.3 Vorgehensweise von Algorithmen

Als Beispiel für **inkrementelles Vorgehen** haben wie Insertion Sort kennen gelernt:

```
sortiere Feld A[1...j]
sortiere Feld A[1...j+1]
...
sortiere Feld A[1...j+n]
```

Diese Verfahren führte zu einer mittleren quadratischen Laufzeit, was relativ langsam ist. Eine Alternative dazu stellt das sogenannte **divide&conquer** Verfahren dar:

[1] Es wird auch oft $f(n) = \Theta(n)$ geschrieben, aber die mathematisdch korrekte Schreibweise lautet $f(n) \in \Theta(n)$, da $\Theta(n)$ eine Menge bezeichnet. In diesem Script wird diese Variante verwendet.
[2] Beweis siehe Cormen

1.3.1 Merge Sort

Merge Sort beruht auf dem Prinzip von „divide&conquer“.

Idee:

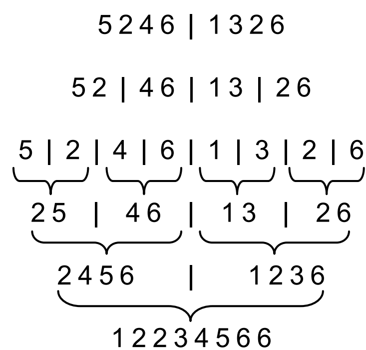
(1) teile die Folge in mehrere Teile (divide)

(2) sortiere jeden Teil für sich (conquer)

(3) ordne die Teile ineinander ein (merge)

Dieses Prinzip wird rekursiv angewendet, d.h. um die entstandenen Teilfolgen zu sortieren, ruft sich der Algorithmus selber auf!

Beispiel:
Zu sortieren sei die Folge <5, 2, 4, 6, 1, 3, 2, 6>. Dabei geht der Algorithmus in den folgenden Schritten vor:



Code von Merge Sort

```
merge_sort(A,p,r) {
if ( p < r )
     { q = (p+r)/2; } /*Abrunden erfolgt automatisch, wenn q integer*/
merge_sort(A,p,q);
merge_sort(A,q+1,r);
merge(A,p,q,r); }
```

Merge Sort benötigt merge als Unterprogramm. Der Aufwand für merge ist linear in r – p. $D(r-p) \in \Theta(r-p)$
Merge geht folgendermaßen vor:

Code von Merge

```
void merge (A,p,q,r) {
/* an dieser Stelle müßte der Speicher reserviert werden*/
l = 1;
i = p;
j = q + 1;
while ( l ≤ r-p+1 )
{
     if ( A[i] ≤ A[j] )
     {
          B[l] = A[i];
          i++;
     }
     else
     {
          B[l] = A[j];
          j++;
     }
```

```
        l++;
    }
    for ( l=p ; l≤r ; l++ )
    {
        A[l] = B[k-p+1]
    }/* an dieser Stelle müßte der Speicher freigegeben werden*/ }
```

Gesamtaufwand: Zur Vereinfachung gehen wir davon aus, daß $r-p=2^k$, d.h. es gibt höchstens $\log_2(2^k) = k$ Aufteilungsschritte und ebensoviele merge-Schritte. Es gilt: $T(1) = C$, $T(n) = 2T(n/2)+D(n)$
Man kann zeigen: $T(n) = O(n \cdot \log_2(n)) = \Theta(n \cdot \log_2(n))$[1].

Vergleich von Insertion Sort und Merge Sort

- Fall A ist ein Supercomputer mit 10^{10} Instruktionen pro Sekunde, auf dem ein äußerst guter Programmierer die Laufzeit von Insertion Sort auf $2n^2$ gedrückt hat.
- Fall B ist ein älterer ALDI-PC mit 10^8 Instruktionen pro Sekunde, auf dem ein sehr mäßiger Programmierer einen Merge Sort mit einer Laufzeit von $50n\log_2(n)$ laufen läßt.

n	**A**	**B**
10^3	10	$5 \cdot 10^{-3}$s
10^6	200s	10s
10^9	~6a	~4h

$\Rightarrow$ Algorithmenentwicklung ist eine Technologie! Die Rechnergeschwindigkeit verdoppelt sich gegenwärtig alle 18 Monate, bei geschickter Algorithmenwahl ist ein Faktor von 1000 oder mehr auf einen Schlag möglich!

1.3.2 Maximale Teilsumme

Gegeben: Folge $(x_1, x_2, \ldots, x_n)$
Gesucht: Teilfolge $(x_i, \ldots, x_j)$, so daß die Summe $\sum_{l=i}^{j} x_l$ maximal wird.
Folgen mit Null Elementen sind dabei zugelassen, deren maximale Teilsumme definiert man als Null. Somit ist sie maximale Teilsumme immer größer oder gleich Null.

Beispiel:

$$31 \quad -41 \quad \underbrace{59 \quad 26 \quad -53 \quad 58 \quad 97}_{\text{maximale Teilfolge, } \sum=187} \quad -93 \quad -23 \quad 84$$

Zur Lösung des Problems werden drei Algorithmen behandelt, die sich stark in ihrer Laufzeit unterscheiden.

Naiver Algorithmus

Dieser durchläuft von die Folge von x_1 bis x_n, errechnet alle möglichen Teilsummen, vergleicht diese und merkt sich die bisher Größte.

```
mts_naiv(X,n,mts) {
mts = 0;
for ( i=1 ; i≤ n ; i++ )
{
    for ( j=1 ; j≤n ; j++ )
    {
        sumij = 0;
        for ( l=i ; l≤j ; l++ )
        {
            sumij = sumij + X[l];
            if ( sumij>mts ) mts = sumij;
        }
    }
```

[1] siehe 1.3.3 (Master-Theorem)

```
}}
```

Laufzeit:

$$T(n) = c_1 n + (c_2 + c_3 + c_6) \cdot \frac{n(n+1)}{2} + (c_4 + c_5) \cdot \underbrace{\left(\sum_{i=1}^{n} \sum_{j=i}^{n} \sum_{l=i}^{j} 1 \right)}_{= \frac{1}{6}n^3 + \frac{1}{2}n^2 + \frac{1}{3}n}$$

Die asymptotische Laufzeit ist also $\Theta(n^3)$, der Algorithmus ist also sehr ineffizient. Die Laufzeit T(n) hängt nicht von der Probleminstanz ab, deshalb ist der worst case gleich dem best case. Auch im einfachsten Fall (nur positive Zahlen) testet der Algorithmus alle möglichen Teilsummen durch.

Divide&conquer

Idee: Teile den Array A[l]…A[r] „in der Mitte“ auf. Das mittlere Element hat den Index (l+r)/2. Es gibt dann drei Fälle:

(1) Die maximale Teilsumme (MTS) liegt „links“ von m: $i^* \le j^* \le m$

(2) Die MTS liegt „rechts“ von m: $m+1 \le i^* \le j^*$

(3) Die MTS geht über die Mitte hinweg: $i^* \le m < j^*$

Zu (3): Sind MTSL / MTSR „linke“ bzw. „rechte“ Teil der max. Teilsumme, dann gilt:

$MTSL = \sum_{l=i^*}^{m} A[l]$ ist optimal unter allen $\sum_{l=i^*}^{m} A[l]$, i=1…m („linkes Randmaximum“)

$MTSR = \sum_{l=m+1}^{j^*} A[l]$ ist optimal unter allen $\sum_{l=im+1}^{j^*} A[l]$, j=m+1…n („rechtes Randmaximum“).

Code:

```
mts_dc(A,l,r,mts) {
if ( l=r )
{
     if ( A[r]<0 )mts = 0;
     else mts = A[r];
}
else
{
     m = (l+r)/2;
     mts_2(A,l,m,mtslinks);
     mts_2(A,m+1,r,mtsrechts);
     mts_2_l(A,l,m,mtsl);
     mts_2_r(A,m+1,r,mtsr);
     mts = mtsl+mtsr;
     if ( mts<mtslinks ) mts = mtslinks;
     if ( mts<mtsrechts ) mts = mtsrechts;
}}
```

Unterprogramm zur Bestimmung des linken Randmaximums:

```
mts_dc_l(A,l,k,mtsl) {
mtsl = 0;
sum = 0;
for ( i=k ; i≥l ; i-- )
{
     sum = sum + A[i];
     if ( sum>mtsl ) mtsl = sum;
}
```

Analog ist das Unterprogramm zur Bestimmung des rechten Randmaximums definiert.

Beispiel: $n = 2^3 = 8$
Für jede Teilfolge muß berechnet werden:

- max. Teilsumme mit dem linken Randelement (Im Bsp. unten an erster Stelle)
- max Teilsumme mit beliebigen Elementen (Im Bsp. unten an zweiter Stelle)
- max. Teilsumme mit dem rechten Randelement (Im Bsp. unten an dritter Stelle)

$$\underbrace{31 \quad -41}_{31\ 31\ -10} \quad \underbrace{59 \quad 26}_{85\ 85\ 85} \quad \underbrace{-53 \quad 58}_{5\ 58\ 58} \quad \underbrace{97 \quad -93}_{97\ 97\ 4}$$

$$\underbrace{\qquad\qquad\qquad}_{75\ 85\ 85} \qquad \underbrace{\qquad\qquad\qquad}_{105\ 155\ -}$$

$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad}_{-\ 187\ 155}$$

Die **Laufzeit** ist hier offenbar $D(l,r) \in \Theta(r-l+1)$. Bei der Laufzeitanalyse nehmen wir wieder vereinfachend an, daß $n = 2^k$.

- Die Berechnung der Randmaxima hat in jedem Schritt einen Aufwand von $D(n) \in O(n)$.
- Ansonsten gilt $T(n) \in \begin{cases} \Theta(1), & \text{für } n = 1 \\ T(\frac{n}{2}) + D(n), & \text{sonst in jedem Einzeldurchlauf} \end{cases}$
- In jedem Schritt entsteht ein Aufwand vom Typ $2 \cdot T(n/2)$, $4 \cdot T(n/4)$, $8 \cdot T(n/8)$ etc., d.h. alle sind gleich bis auf Terme höherer Ordnung!
- Es gibt $k = \log_2(n)$ Zerlegungsstufen, je Stufe ein ist der Aufwand $D(n)\ \Theta(n) = k \cdot D(n/k)$.

sei die Anzahl der Ausführungen von MTSL bzw MTSR. T sei die Zeit pro Ausführung, dann berechnet sich der Gesamtaufwand wie folgt:

```
 1   1   1   1     1   1   1   1      #: 8, T ≤ 1·a ⇒ Σ ≤ a·8
   2       2         2       2        #: 4, T ≤ 2·a ⇒ Σ ≤ a·8
       4                 4            #: 2, T ≤ 4·a ⇒ Σ ≤ a·8
              8                       #: 1, T ≤ 4·a ⇒ Σ ≤ a·8
```

$\Rightarrow T(n) \leq an \cdot (\log_2(n)+1) = n + n \cdot \log(n) = O(n \cdot \log_2 n)$

Dieses Verfahren hat also eine Laufzeit von $O(n \cdot \log(n))$ und ist damit wesentlisch schneller als der naive Algorithmus. Es geht aber noch schneller, mit Hilfe des folgenden Scanline-Algorithmus.

Scanline

Grundidee:
Hier wird ausgenutzt, daß sich am Rand einer maximalen Teilsumme immer negative Zahlen befinden. (oder natürlich der Rand der Folge). Der Algorithmus geht den Array von A[1] bis A[n] durch, merkt sich in „bismax" die bisher größte gefundene Teilsumme und vergleicht diese mit „scanmax":

- Ist scanmax+A[i] > 0, kann aus dieser Teilfoge noch die max. Teilsumme werden
- Ist scanmax+A[i] ≤ 0, kann diese Folge keine max. Teilsumme mehr werden und wird weggelassen.

- Ist scanmax > bismax, ist eine neue, größere max. Teilsumme gefunden worden.

Code:

```
mts_scan(A,n,mts) {
scanmax = 0;
bismax = 0;
for ( i=1 ; i≤n ;i++ )
{
     s = A[i];
     if ( scanmax+s > 0 ) scanmax = scanmax + s;
     else scanmax = 0;
     if ( scanmax>bismax ) bismax = scanmax;
}}
```

Die **Laufzeit** ist offenbar linear in n, $T(n) \in \Theta(n)$. Da jedes Element mindestens einmal betrachtet werden muß, erfordert das Problem auch mindestens n Schritte. Der Algorithmus hat die optimale Ordnung, es kann kein Algorithmus mit einer niedrigeren Laufzeit als $\Theta(n)$ existieren.

Ein Zahlenbeispiel zum Vergleich der drei Algorithmen:

n	**naiver Algorithmus**	**divide&conquer**	**scanline**
10^3	21,3s	$\to 0$	$\to 0$
10^4	23866,1s	0,05s	$\to 0$
10^6	$\to \infty$	6,1s	0,33s

2 Sortieren

Problem: Mehr als 25% der kommerziell genutzten Rechenzeit entfällt auf Sortieren.
Input: Zahlenfolge ($a_1 \ldots a_n$)
Output: Umordnung (Permutation): $a_1' \ldots a_n'$, so daß $a_1' \leq \ldots \leq a_n'$

Dabei ist in der Praxis Folgendes zu beachten:

- Zahlen stehen i.A. nicht allein, sondern gehören zu einem Datensatz(engl.: record). Der Satz enthält einen Schlüssel (engl.: key), der die Ordnung definiert. Das Umspeichern von Datensätzen kann sehr aufwendig sein. Evtl. reicht es deshalb, nur eine Indexliste zu sortieren, die Pointer auf die Zahlen enthält.
- Auch Vergleichsoperationen können teuer sein, weil es auch sehr komplexe Vergleiche gibt, z.B. Daten auf der Übungsteilnehmerliste: zu sortieren nach Fachsemester (einfach) oder nach Fächern (in festgelegter Reihenfolge – schon komplizierter) und dann nach Semesteranzahl (noch komplexer). Oder Daten von einem Personalausweis: sortieren nach Geburtsjahr, nach Jahr, Monat und Tag, nach Familien- und Vornamen in alphabetischer Reihenfolge.

Die im Folgenden beschreibenen Sortieralgorithmen Selection Sort, Bubble Sort, Heapsort und Quicksort basieren wie die bereits behandelten Verfahren Insertion Sort und Merge Sort auf den direkten Vergleichen zweier Zahlen.
Am Ende dieses Abschnitts werden mit Countingsort und Radixsort zwei Verfahren vorgestellt, die keine direkten Vergleiche benötigen.

2.1 Selection Sort

Idee: Suche für jedes i an der Stelle A[i] das kleinste Element A[j] ≤ A[l], l = i…n und vertausche A[i] und A[j],
Beispiel:

<**7**, 6, 5, 3, 4, **1**>

<1, **6**, 5, **3**, 4, 7>

<1, 3, **5**, 6, **4**, 7>

<1, 3, 4, **6**, **5**, 7>

<1, 3, 4, 5, 6, 7>

Code von Selection Sort

```
selection_sort(A,n) {
for ( i=1 ; i≤n-1 ; i++ )
/* n-1, da die letze Zahl automatisch die Größte ist */
{
     key = A[i];
     jopt = i;
     for ( j=i+1 ; j≤n ; j++ )
     {
          if ( A[j]<key )
          {
               key = A[j];
               jopt = j;
          }
          A[jopt] = A[i];
          A[i] = key;
     }
}}
```

Laufzeit:

Insgesamt sind $\sum_{i=1}^{n-1}(n-i) = \frac{n(n-1)}{2}$ Vergleiche notwendig, d.h. im worst case ist der Aufwand $O(n^2)$.

Der Aufwand für die Datenbewegungen ist linear mit n $\Rightarrow T(n) \in O(n)$. Die asymptotische Laufzeit ist wie bei Insertion Sort $\Theta(n^2)$, aber es sind weniger Umspeicherungen nötig.

2.2 Bubble Sort

Bubble Sort ist ein beliebter Sortieralgorithmus, da er leicht zu implementieren ist.
Idee: Vertausche so lange fehlstehende Nachbarelemente, bis alles sortiert ist.
Beispiel: (bereits richtig sortierte Elemente sind unterstrichen)

Schritt #					
1	5	2	4	3	1
2	2	5	4	3	1
3	2	4	5	3	1
4	2	4	3	5	1
5	2	4	3	1	5
6	2	3	4	1	5
7	2	3	1	4	5
8	2	1	3	4	5
9	1	2	3	4	5

Code einer einfachen Bubblesort – Variante

```
bubble_1(a,n) {
for ( j=n ; j≥1 ; j-- )
{
     for ( i=1 ; i≤j-1 ; i++ )
     {
          if ( A[i] > A[i+1] )
          {
               t = A[i]; /* vertausche A[i] und A[i+1] */
               A[i] = A[i+1];
               A[i+1] = t;
          }
     }
}}
```

Man kann Bubble Sort beschleunigen, indem die zweite Schleife nicht ganz bis j–1 durchlaufen läßt, sondern an der Stelle stoppt, ab der die Elemente bereits sortiert sind. (siehe Beispiel)

Code einer schnellerer Bubblesort – Variante

```
bubble_2(A,n) {
j = n;
change = TRUE;
do {
     change = FALSE;
     for ( i=1 ; i≤j-1 ; i++ )
     {
          if ( A[i] > A[i+1]
          {
               t = A[i]; /* vertausche A[i] und A[i+1] */
               A[i] = A[i+1];
               A[i+1] = t;
               change = TRUE;
          }
```

```
        }
        j--;
    } while change = TRUE; }
```

Vergleich Sortieralgorithmen

Laufzeiten:

n	Insertion Sort (8n²)	Merge Sort (64n·log(n))
2	32	128
4	128	512
8	512	1536
16	2048	4096
32	2192	10240
64	32768	24600

⇒ Für kleine n können die sog. „langsamen" Algorithmen mit $O(n^2)$ also schneller sein als divide&conquer. Daraus folgern wir, daß man divide&conquer durch Einsatz von „langsamen" Algorithmen auf den unteren Stufen der Rekursionen beschleunigen kann!

Rechenzeiten[1]:

n	Insertion Sort	bubble sort 1	bubble sort 2	Selection Sort
5000	5,1	19,4	12	5,8
10000	21,2	78,9	48,6	23,6
20000	84,9	345,5	193,8	95,5

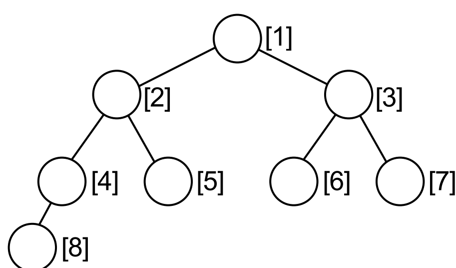
2.3 Heap Sort

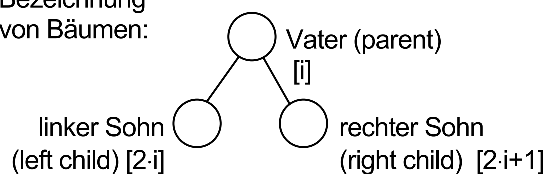
Idee: Kombination der Vorteile von Merge Sort und Insertion Sort. d.h. eine Laufzeit von $n \cdot \log_2(n)$ bei einem niedrigen Speicherverbrauch.

2.3.1 Definitionen

Binärer Baum

Ein binärer Baum ist eine Datenstruktur mit folgendem Aufbau[2]:





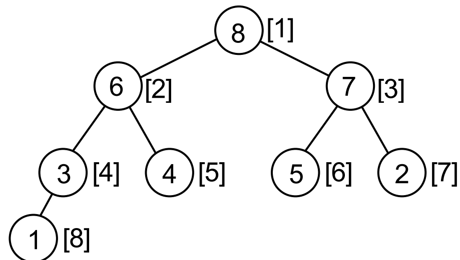
[1] berechnet auf einer Sparc SLC, einer inzwischen älteren Workstation:
[2] Die Indizierung der Elemente geht hier davon aus, das die Wurzel die Nummer 1 hat. Da in den meisten Programmiersprachen aber bei 0 mitt dem Zählen begonnen wird, sind die Indizierungen dort entsprechend abzuändern.

Heap

Ein Heap ist ein binärer Baum, der folgende Eigenschaft erfüllt:

$$A[parent[i]] \geq A[i] \text{ oder } a_{i/2} \geq a_i \; \forall i$$

Dies bedeutet, daß jedes Sohnelement kleiner oder gleich seinem Vaterelement ist. Die Wurzel des Baumes hat automatisch den größten Wert.



Das nebenstehende Bild zeigt den Array <8, 6, 7, 3, 4, 5, 2, 1> als Heap.

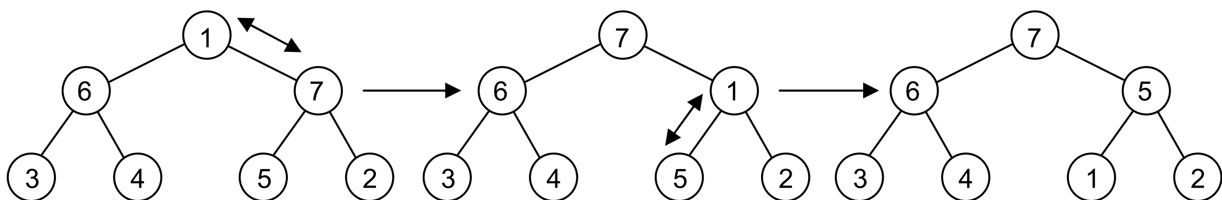
2.3.2 Die Bestandteile von Heap Sort

Heapify

Heap Sort benötigt die Unterroutine heapify. Heapify erzeugt eine Heapstruktur mit im Unterbaum mit der Wurzel i, wenn die Unterbäume [2i] und [2i+1] bereits ein Heapstruktur haben. Existiert ein Sohn oder beide Söhne nicht, so wird als deren Wert $-\infty$ angenommen.

```
heapify(A,n,i) {
suche k ∈ {i, 2i, 2i+1} mit größtem A[k]
     /* falls Söhne nicht vorhanden wird als Wert –∞ angenommen */
falls k ≠ i, vertausche A[i] und A[k] und führe heapify (A,n,k) aus. }
```

Beispiel: Heapify auf die Wurzel



Die Laufzeit beträgt $T(n) \leq \Theta(1) + T(3n/2)$, $T(n) = O(\log_2(n))$

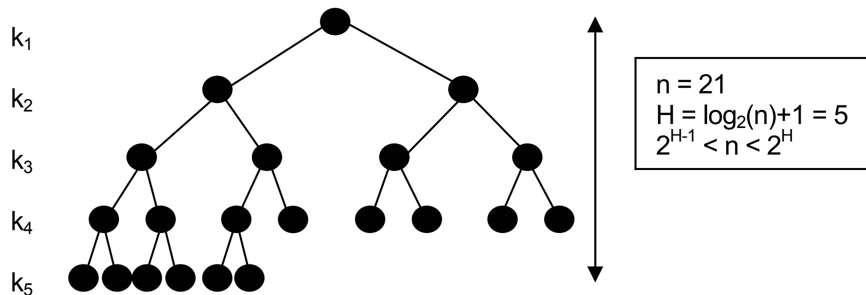
Build Heap

Die Unterroutine build Heap Sortiert einen Array in einen Heap um. Dabei werden alle Endpunkte des Baumes als triviale Unterbäume mit der Größe 1 betrachtet. DIe Routine startet Heapify von allen anderen Knoten in absteigender Reihenfolge n/2 bis 1. Deshalb sind die Unterbäume der Söhne bereits sortiert, wenn die Väter an die Reihe kommen.

```
build_heap(A,n) {
for ( i=n/2 ; i≥1 ; i-- ) heapify(A,n,i) }
```

Laufzeit:
Eine einfache Laufzeitabschätzung ergibt $O(n)$ Aufrufe von heapify à $O(\log_2(n)) \Rightarrow T(n) \in O(n \cdot \log_2(n))$.

Dies ist aber nur eine grobe obere Schranke. Eine genauere Laufzeitbetrachtung berücksichtigt, wie lange der Aufruf von heapify in Abhängigkeit von jeder Stufe im Baum benötigt.



Der Aufwand für jedes heapify ist $O(H–k)$:

$$T(n) \leq \sum_{k=1}^{H-1} 2^{k-1} \cdot C \cdot (H-k) = C \cdot \left(2^0(H-k) + 2^1(H-k) + K\ .2^{H-2} \cdot 1\right) = C \cdot \underbrace{2^{H-1}}_{<n} \left(\frac{H-1}{2^{H-1}} + \frac{H-2}{2^{H-2}} + K + \frac{1}{2^1}\right)$$

$= C \cdot \sum_{i=1}^{H-1} \frac{i}{2^i}$ Unter Verwendung von $\lim_{n \to \infty}\left(\sum_{i=1}^{H-1} \frac{i}{2^i}\right) = 2$ ergibt sich also eine Laufzeit von $T(n) \leq 2 \cdot Cn$

Heap Sort

```
heap_sort(A,n) {
build_heap(A,n);
size = n;
for ( i=n ; i>2 ; i--)
{
     vertausche A[i] und A[1];
     size--;
     heapify[A,size,1];
}}
```

Laufzeit: Die Gesamtlaufzeit ergibt sich also als Summe aus den Laufzeiten für build_heap und der Laufzeitz aller Auführungen von heapify.
$T(n) = O(n) + (n–1) \cdot O(\log_2(n)) = O(n \cdot \log_2(n))$

2.4 Quicksort

Vorgehensweise

Quicksort ist vom Typ divide&conquer und wird sehr häufig verwendet, da er schnell und leicht zu implementieren ist. Das hier behandelte Verfahren ist nur ein Vertreter einr Gruppe von Quicksortalgorithmen. Einige mögliche Variationen sind am Ende des Abschnittes aufgeführt.

Idee: $F = (a_1 \ldots a_n)$

- Falls $F = \emptyset$ oder $\#F = 1$ ist bereits alles sortiert.
- Wähle ein Element $k \in F$, das sog. „Pivot“, in diesem Fall das rechts stehende Element $k = a_n$
- Teile F auf in $F_1 = \{a_j < k\}, \{k\}, \{a_i \geq k \mid a_j \neq a_n\} = F_2$.
- Führe Quicksort aus auf F_1 und F_2. (F_1,k,F_2) ist dann das sortierte Ergebnis.

Es stellt sich natürlich die Frage, wie man eine Folge A[l]…A[r] effektiv aufteilt.

- Wähle A[r] = k als Pivot.
- Lasse zwei Indizes gleichzeitig laufen: i von l aus aufsteigend, j von r–1 aus absteigend.
- Teste, ob A[i] < k und A[j] ≥ k. (Hier wird festgestellt, ob A[i] ∈ F_1 und A[j] ∈ F_2, s.o.)
- Trifft eine diese Bedingungen nicht zu, halte den entsprechenden Index an.
- Sobald beide Indizes stehen, also ein „fehlstehendes" Element gefunden haben, vertausche beide Elemente.
- Ist i > j, so ist alles aufgeteilt.
- Bringe das Pivot wieder in „die Mitte": A[r] = A[i], A[i] = k.

Code von Quicksort

```
quicksort(A,l,r)
if r≤l return
k = A[r]
i = l-1
j = r
do
{
     do { i++ } while A[i]<k
     do { j-- } while (A[j]>k and j≥l)
     if i<j { A[i] ↔ A[j] }
     else
     {
          A[r] = A[i]}
          A[i] = k
     }
} while ( i < j)
quicksort (A,l,i-1)
quicksort (A,i+1,r)
```

Laufzeit: Im worst case ist die Folge bereits sortiert, so daß keine Aufteilung stattfindet. Dann wird der Prozeß n mal durchlaufen. Der Aufwand ist dabei jedes Mal n–1, n–2… ⇒ $T(n) \in O(n^2)$. Im best case ist die Folge immer in der Mitte halbierbar. ⇒ $T(n) \in O(n\cdot \log_2(n))$

Für den average case kann man folgendes zeigen: Falls alle Permutationen mit gleicher Wahrscheinlichkeit vorkommen, ist im Mittel $T(n) \in O(n\cdot\log_2(n))$ als Erwartungswert, dann sind aber ~38% mehr Vergleiche notwendig als im best case.

Eigenschaften von Quicksort

Der Vorteil von Quicksort liegt also darin, daß er im average case lediglich eine Laufzeit von $n\cdot\log_2(n)$ hat, ohne zusätzlichen Speicher zu belegen.
Der Nachteil ist, daß er etwas langsamer ist als Merge Sort und im ungünstigsten Fall eine Rekursionstiefe von n hat. Dies führt zu großem Speicherverbrauch, weil für jede Rekursion die Werte aller lokalen Variablen der Unterroutine zwischengespeichert werden müssen.
Der Algorithmus existiert in verschiedenen Varianten, die das Ziel haben, Vorsortierungen zu vermeiden:

- Bestimme Pivotindex zufällig und vertausche A[μ] und A[r].
- „median choice": Bestimme das Pivot aus mehreren Elementen: A[l], A[(l+r)/2], A[r] und nehme dann das Element mit dem mittleren Wert.
- Wie oben, jedoch mit zufälligen Indizes.
- Sortieren die Eingabefolge zufällig um. (kein Scherz!) Hierbei muß man beachten, daß sowohl die Erzeugung von Zufallszahlen sowie die Umsortierung ebenfalls Zeit kosten.

Um allzu große Rekursionstiefen zu vermeiden spart man sich am Ende des Programms einen Aufruf von Quicksort und führt an dieser Stelle die Sortierung direkt durch. So erreicht man Rekursionstiefe von $\log_2(n)$.

2.5 Counting Sort

Problem: Sortieren mit Schlüsselvergleichen kostet im worst case mindestens $O(\log(n))$ Vergleiche. Merge Sort und Heap Sort haben also schon die optimale Komplexitätsordnung im worst case unter Verfahren, die Schlüsselvergleiche durchführen.
Man kann aber im Spezialfall Schlüsselvergleiche umgehen und somit schneller als $O(\log(n))$ werden. Ein Beispiel für einen solchen Algorithmus ist Counting Sort.

Damit man Counting Sort anwenden kann, muß à priori die Anzahl der Möglichkeiten für einen Schlüssel sowie die obere und untere Grenze bekannt sein. Desweiteren läßt sich das Verfahren nur auf ganzzahlige Schlüssel anwenden, z.B. $a_i \in \{1, 2, \ldots, M\}$, $M \in \angle$

Idee:

(1) Zähle alle Elemente mit Schlüsselwert j für alle $j = 1, \ldots, M \Rightarrow N_j$ $(j = 1, \ldots, M)$

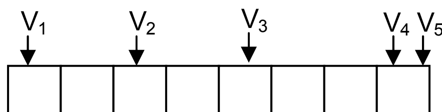
(2) Berechne Adressen in Array, indem sortierte Folge gespeichert wird. $V_1 = 1$, $V_2 = V_1+N_1$, $V_3 = V_2+N_2$

(3) Sortiere Elemente A[i] ein.

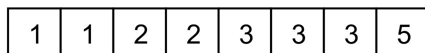
Beispiel: $M = 5$, $S = \{1, 2, 3, 4, 5\}$, $(a_1 \ldots a_n) = <3\ 2\ 5\ 3\ 1\ 2\ 1\ 3>$

(1) Zähle: $N_1 = 2$, $N_2 = 2$, $N_3 = 3$, $N_4 = 0$, $N_5 = 1$

(2) Anfangsindizes: $V_1 = 1$, $V_2 = 3$, $V_3 = 5$, $V_4 = 8$, $V_5 = 8$



(3) erzeuge Folge:



Bestimme Zahl μ_i aller Elemente $a_j < a_i$ zu jedem a.
Von $a_j = a_j$: zähle nur die mit dem kleineren Index $j < i$

Dies hat den Effekt, daß Elemente mit gleichem Schlüssel in der Reihenfolge unverändert bleiben.

Code von Counting Sort

Um Array A sortieren, führt man zwei neue Arrays B und C ein. B ist das Zielarray und hat die selbe Größe wie A. C ist das Array, in dem man sich die Anzahl der Schlüssel merkt und hat somit die Größe M.

```
counting_sort(M,n) {
for ( j=1 ; i≤n ;i++ )
{
    C[A[j]]++;
}/*Jetzt steht auf C[i] die Anzahl der Elemente mit dem Schlüssel i.*/

for ( i=2 ; i≤M ; i++ )
{
    C[i] = C[i]+C[i-1];
} /* Auf C[i] steht jetzt die Adresse Vi-1+Ni = Vi+1-1, d.h. hier steht
der letzte Index des Array-Segements*/
```

```
for ( j=n ; j≥1 ;j-- )
{
     B[C[A[j]]] = A[j];
     C[A[j]]--;
}}
```

Eigenschaften von Counting Sort

Satz:
Sei A eine Folge von n Elementen mit Schlüsseln aus S = {1, …, M}, dann sortiert Counting Sort in $O(M+n)$ Schritten.
Aber hierzu werden zwei zusätzlich Arrays B und C benötigt, die sehr viel Speicher belegen. Falls M konstant unabhängig von n, also M = $O(1)$, dann ist die Laufzeit $O(n)$. Falls M = $O(n)$, dann ist die Laufzeit $O(n)$, denn $O(O(n)+n) = O(n)$.

Satz:
Counting Sort ist „stabil“, d.h. Elemente a_i mit gleichem Schlüsselwert haben nach Sortierung gleiche Reihenfolge.

Die Eigenschaft „stabil“ ist natürlich nur wichtig, wenn der Schlüssel zusammen mit Daten steht, deren Ordnung erhalten bleiben soll oder bei sukzessiver Sortierung, z.B. wenn in einer bereits nach Geburtsjahr sortierten Adressenliste zusätzlich nach Geburtsmonat sortiert werden soll.

Beispiel:
In einer Adressenliste sortiert man zunächst nach Geburtsjahr. Sortiert man jetzt mit einem stabilen Algorithmus nach Geburtsmonat, so bleibt die Reihenfolge der Geburtsjahre erhalten, und eine im August 1978 geborene Person wird nicht direkt z.B. hinter einer im Juli 1963 geborenen Person aufgeführt, sondern erst nach den im Juli 1978 geborenen Personen.

2.6 Radix Sort

Radix Sort ist die Verallgemeinerung von Counting Sort für den Fall, daß die Schlüssel aus endlich vielen Komponenten besteht mit jeweils endlich vielen Elementen M_i.

Idee:

- Sortiere sukzessive nach Komponenten des Schlüsseln
- mit stabilem Algorithmus
- entgegen der Intuition von „hinten“, d.h. mit dem Schlüssel der geringsten Signifikanz beginnend

Beispiele:

- 6-stellige Dezimalzahlen: 6 Schlüssel à 10 Möglichke Schlüsselwerte

1	4	7	3	6	8

- Geburtsdaten: ein Schlüssel à 31, einer à 12 und einer à 100 Möglichkeiten

3	1	0	1	8	0

- Bytes (8stellig dual): 8 Schlüssel à 2 Möglichkeiten
- 3stellige Dezimalzahlen:

Satz: l sei die Anzahl der Schlüssel, M die maximal mögliche Anzahl der jeweiligen Schlüsselwerte, dann ist die Laufzeit $O(l(M+n))$, falls l mal Counting Sort durchgeführt wird.
Also ist die Laufzeit von Radix Sort linear in n, falls M = O(1) oder M = O(n).

2.7 Rechenexperimente

Folgende Werte beziehen sich auf Aufgabe n natürliche, positive Zahlen im Bereich von $0 \ldots 10^9$ zu sortieren, dies entspricht maximal 4 Byte pro Zahl.

Rechenzeiten und Laufzeiten

Algorithmus	**Laufzeit T(n)**
Insertion Sort	$0{,}22n^2$
bubble sort	$0{,}45n^2$
Selection Sort	$0{,}23n^2$
Heap Sort	$15{,}4n \cdot \log(n)$
Merge Sort	$0{,}96n \cdot \log(n)$
quicksort	$1{,}94n \cdot \log(n)$
Radix Sort	$9{,}56n$

Algorithmus	**Anzahl**	**Zeit[s]**
bubble sort	$n = 10^4$	t = 044,9
Heap Sort	$n = 4 \cdot 10^5$	t = 115,1
quicksort	$n = 10^6$	t = 019,1
Radix Sort	$n = 10^6$	t = 009,6

Schlußfolgerungen

Ein „bestes“ Verfahren existiert nicht!
Welcher Algorithmus zu einem gegebenen Problem die beste Lösung darstellt, sind viele Faktoren relevant:

- Größe der Datenmenge
- Typ der Schlüssel
- Speicherbedarf
- Rechenzeit
- Eigene Arbeitszeit
- Typ der Daten
- Verhältnis der Größen von Schlüssel und anhängenden Daten
- Bei sehr großen Datenmengen Zugriffszeiten auf die Speichermedien wie on-chip memory, Cache, RAM und Festplatte.

3 Elementare abstrakte Datentypen

3.1 Einführung

Bisher wurden nur „Arrays“ vom Typ A[1]…A[n] behandelt. Diese bilden eine Folge von Daten, so daß auf jedes Element A[i] mit Hilfe eines Index direkt zugegriffen werden kann. Für viele Zwecke sind aber andere Strukturen günstiger.

Definition:

Zu einem allgemeinen abstrakten Datentyp (ADT) gehören eine Menge von Objekten (Daten) und darauf ausführbare Operationen (Algorithmen).
Die konkrete Realisierung in einer Programmiersprache interessiert hier zunächst nicht.

3.2 Der Stack

Der Stack ist eine sehr einfache Datenstruktur, bei der nur auf das zuletzt eingefügte Element zugegriffen werden kann. Praktische Beispiele sind z.B. ein Aktenstapel auf einem Schreibtisch oder der Tablettstapel in der Mensa. Um auf das zweite, dritte etc. Element von oben zugreifen zu können, muß man erst alle darüberliegenden, später eingefügeten Elemente herunternehmen.

Struktur des Stacks

Objekte: Dynamische Folgen von Daten mit veränderlicher Länge
Operationen:

- Elemente einfügen (push)
- Zugreifen und entfernen des zuletzt eingefügten Elements (pop)
- Erzeugen eines leeren Stacks (init_stack)
- Feststellen, ob Stack leer ist (stack_empty)

Attribute: Um mit dem Stack arbeiten zu können, benötigen wir noch das Attribut top(s), das den Index des zuletzt eingefügten Elements liefert und damit die aktuelle Länge des Stacks angibt.

Eine möglichst einfache Implementierung des Stacks verwendet einen Array der Größe n, mit einem hinreichend großen n. Das erste Element des Stacks wird an die erste Stelle des Arrays geschrieben.

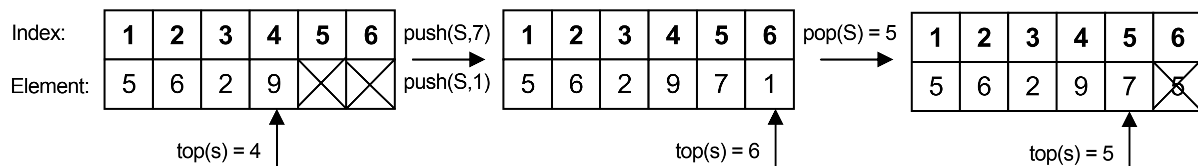
Der Stack hat die Elemente S[1] bis S[top(s)]. Wenn top(s) = 0, so ist der Stack leer und die Operation pop unzulässig und führt zu dem sog. Fehler „stack underflow“. Ist top(s) = n, so ist der Stack voll, die Operation push ist unzulässig und führt zum „stack overflow“.

Code der Stack-Operationen

```
stack_empty(S) {
if ( top(s)=0 ) return(TRUE);
else return(FALSE); }

stack_full(S,n) {
if ( top(s)==n ) return(TRUE);
else return (FALSE); }
```

```
push(S,x) {
if ( stack_full == TRUE ) return(error_overflow);
top(s)++;
S[top(s)] = x; }

pop(S) {
if ( stack_empty == TRUE ) return(error_underflow);
top(s)--;
return S[top(s)+1]; }
```

Beispiel:



3.3 Die Queue

Bei der Queue kann man auf das zuerst eingefügte Element zugreifen. Im Alltag findet man das z.B. in einer Supermarktschlange: wer zuerst in der Schlange stand, kommt auch zuerst raus.

Struktur der Queue

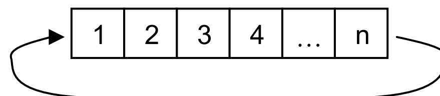
Objekte: Dynamische Folgen von Daten mit veränderlicher Länge.
Operationen:

- Elemente hinzufügen (enqeue)
- Zuerst eingefügtes Element abrufen und entfernen (dequeue)
- Leere Queue erzeugen (init_queue)
- Feststellen, ob queue leer ist (queue_empty)
- Feststellen, ob queue voll ist (queue_full)

Attribute:

- Head(Q) ist der Index des zuerst eingefügten Elements.
- Tail(Q) ist der Index des zuletzt eingefügten Elements +1, also der die Position, an der das nächste Elements eingefügt wird.

Die einfachste Implementierung verwendet ein normales Array Q[1]…Q[n] mit zirkulärer Indizierung, d.h. ist man bei n angelangt, ist der Index des nächsten Elements wieder die 1.



Code der Queue-Operationen

```
queue_full(Q,n) {
if ( (tail(Q)+1)mod(n) == head(Q) ) return(TRUE);
else return (FALSE); }

queue_empty(Q) {
if ( tail(Q) == head(Q) ) return(TRUE);
else return(FALSE); }
```

```
enqueue(Q,x) {
if ( queue_full == TRUE ) return(error_overflow);
Q[tail(Q)] = x;
if ( tail(Q) == n ) tail(Q) = 1;
else tail(Q)++; }

dequeue(Q) {
if ( queue_empty == TRUE ) return(error_underflow);
x = Q[head(s)];
if ( head(Q) == n ) head(Q) = 1;
else head(Q)++; }
```

3.4 Listen

3.4.1 Struktur von Listen

Objekte: Folgen eines Grundtyps (z.B.: Integer-Zahlen) bzw. Datensätzen inkl. eines Schlüssels, dynamisch, d.h. mit variabler Anzahl von Elementen.
Notation:

- $L = \langle a_1 \ldots a_n \rangle$
- L = < > bez. ∅: leere Liste
- Attribut lengh[l]=n

Operationen:

- Erzeuge leere Liste
- Element einfügen
- Element löschen
- Zugriff auf Elemente
- Liste L_1 an Liste L_2 anhängen
- Abfragen, ob Liste leer
- Herausgreifen von Teillisten

Es gibt zwei verschiedene Listentypen: sequentielle Listen und verkettete Listen.

3.4.2 Sequentielle Listen

Dieser Listentyp stellt nichts anders dar als ein Array, dessen Elemente beliebige Daten enthalten, und wird normalerweise nicht verwendet.
Wegen unvermeidbarer Verschiebungen haben die Oparationen Einfügen und Entfernen im worst case eine Laufzeit von $O(n)$. Dies gitl auch im Durchschnitt, falls Positionen p alle gleich wahrscheinlich sind aus <1..n>.
Dieses Problem umgeht man mit der Verwendung von verketteten Listen.

3.4.3 Doppelt verkettete Listen

Idee: Bei den verketteten Listen wird das Prinzip der physikalischen Nachbarschaft der Listenelemente aufgegeben, d.h. statt direkt hintereinander befinden sich die einzelnen Elemente der Liste jetzt „irgendwo" im Speicher.

Struktur

Die Datensätze werden um zwei Zeiger ergänzt, die eine Reihenfolge zwischen den Elementen herstellen: prev(x) zeigt auf den Vorgänger (auf NULL, wenn erstes Element), next(x) zeigt auf den Nachfolger (auf NULL, wenn letztes Element). Ein Element in einer Liste (Knoten) hat folgende Struktur:

Außerdem werden folgende Hilfselemente benötigt: head(L) zeigt auf das erste Element der Liste und tail(L) auf das letzte. Eine komplette verkettete Liste hat also diese Struktur:

Die hier gezeigt Liste ist eine sogenannte "doppelt verkettete Liste", da sie sowohl Zeiger auf das Vorgänger wie auch auf das Nachfolgerelement besitzt. Es werden aber auch einfach verkettete Listen verwendet, die nur einen Zeiger, entweder auf den Nachfolger oder auf den Vorgänger besitzen.
Head(L) und tail(L) werden in einem ADT "Liste" als Pointer auf die Listenelemente implememtiert.

Vorteile:

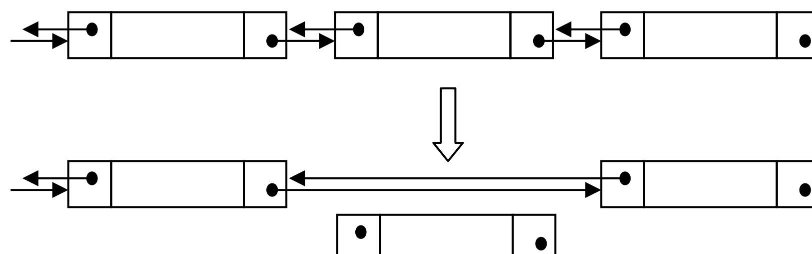
- Das Einfügen/Entfernen/Verbinden benötigt kein Umspeichern mehr.

Nachteile:

- Kein direkter Zugriff auf Elemente mehr.
- Der direkte Zugriff ist nur durch eine zusätzliche Liste der next- und prev-Pointer möglich, diese ist aber wieder sequentiell mit allen damit verbundenen Nachteilen.
- Da die Elemente der Liste über den Speicher „verstreut" liegen, kann der Rechner keine Beschleunigungen aus seiner Caching- oder Pipelining-Architektur ziehen. Somit steigen die Zugriffszeiten.

Codes der Listenoperationen

Entfernen: Das Entfernen geschieht durch umsetzen der Pointer.



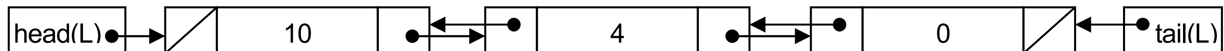
```
list_delete(L,x) {
if ( prev(x) ≠ NULL )
{
     next(prev(x)) = next(x);
}
else head(L) = next(x);

if ( next(x) ≠ NULL )
{
     prev(next(x)) = prev(x);
}
/* bei der späteren Implementation muß unbedingt noch der für das
Element x verwendete Speicher freigegeben werden! */
else tail(x) = prev(x); }
```

Laufzeit: Der Aufwand beträgt $O(1)$, gleiches gilt für einfügen und verbinden.

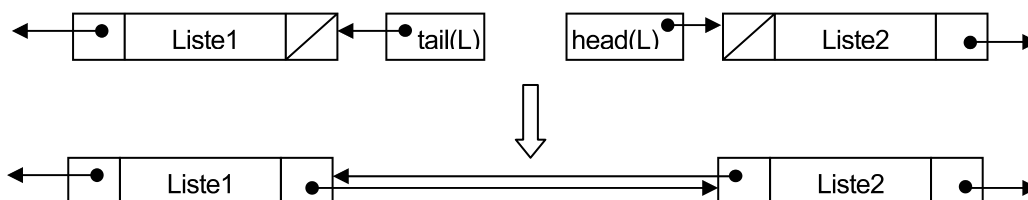
Suche: Suche erstes Element in der Liste L, welches den Schlüssel k hat und gib Pointer auf Element zurück. Falls nicht vorhanden, übergebe NULL-Pointer.

```
int* list_search(L,k) {
x = head(L);
while ( ( x ≠ NULL) && (key[x] ≠ k))
{
     x = next(x);
}
return x; }
```

Beispiel: list_search(L,4) gibt Pointer auf Element 2 zurück. list_search(L,7) gibt NULL-Pointer zurück.



Verbinden: Das Verbinden von Listen geschieht wieder müber das Umsetzen von Pointern




```
void concat_list(Liste1, Liste2, ListeNeu) {
next(tail(Liste1)) = head(Liste2);
prev(head(Liste2)) = tail(Liste1);
head(ListeNeu) = head(Liste1);
tail(ListeNeu) = tail(Liste2); }
```

Laufzeit: Auch hier beträgt die Laufzeit nur $O(1)$.

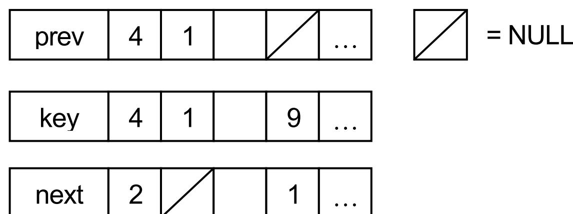
3.5 Implementierung von abstrakten Datentypen

Anhand der doppelt verketten Liste wird hier exemplarisch gezeigt, wie man die Datentypen tatsächlich in Programmiersprachen implementiert.

3.5.1 Implementation in FORTRAN77

Da einige Programmiersprachen wie z.B. FORTRAN77 die Definitionen eigener Datentypen nicht zulassen, muß man eine Umweg gehen, um verkettete Listen zu verwenden:
Man initialisiert drei Felder mit fester Dimension und eine Variable, die den Index des ersten Listenelements enthält.

Beispiel: L=4



3.5.2 Implementation in C

In C hat man die Möglichkeit, sich mit Hilfe von Strukturen (keyword struct) eigene Datentypen zu definieren.

```
struct listnode //Struktur eines Listenknotens
{
     int key;
     /* an der Stelle von der einfachen Intergervariablen „key“ stehen
     normalerweise umfangreichere Daten */
     struct listnode *prev;
     struct listnode *next;
};
```

In den Zeilen zwei und drei wird ein Pointer definiert, der auf eine Struktur vom Typ listnode zeigt. Im Hauptprogramm kann man auf die Elemente der Struktur auf folgende Weise zugreifen:

```
struct listnode KnotenA;
/* Initialisierung einer Variablen vom Typ „listnode“ mit dem Namen
KnotenA */
KnotenA.key liefert Schlüssel von KnotenA
KnotenA.next liefert Zeiger auf nächsten Knoten
KnotenA.prev liefert Zeiger auf vorhergehenden Knoten
```

Hat man statt des Knotens nur einen Zeiger auf diesen, so lassen sich die Elemente wie folgt ansprechen:

```
struct listnode *p_KnotenA;
/* Initialisierung eines Pöointers auf eine Variablen vom Typ listnode
mit Namen p_KnotenA */
p_KnotenA->key liefert Schlüssel von KnotenA
p_KnotenA->next liefert Zeiger auf nächsten Knoten
p_KnotenA->prev liefert Zeiger auf vorhergehenden Knoten
```

Für die Liste selber kann man sich ebenfalls eine Struktur definieren:

```
struct list
/* Struktur der „Liste“, diese enthält aber keine Schlüsseldaten! */
{
     struct listnode *head;
     struct listnode *tail;
     int number; //Anzahl der Listenelemente
};
```

Will man auf die Liste zugreifen, geschieht das auf folgendem Weg:

```
struct list L;
L.head liefert einen Pointer auf das erste Element der Liste
```

3.5.3 Implementation in C++

In C++ kann man sich eigene Datentypen als sog. Klassen definieren. In der C++ Standardbibliothek sind aber bereits viele der häufig verwendeten abstrakten Datentypen implementiert, z.B.

- „list“, eine doppelt verkettete Liste
- „vector“, ein ADT ähnlich dem Stack, bei dem aber zusätzlich direkter Zugriff auf ein Element möglich ist
- „deque“, eine queue, bei der an beiden Enden eingefügt und ausgelesen werden kann

3.6 Bäume

3.6.1 Einführung

Bäume lassen sich mittels verketteter Datenstrukturen darstellen.

Nomeklatur:

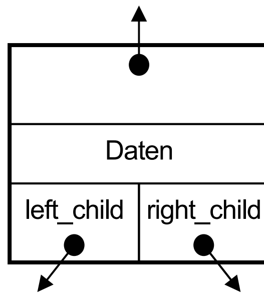
- Die Baumelemente werden Knoten (engl.: node) genannt.
- Das „oberste“ Element eines Baumes heißt Wurzel (engl.: root).
- Als Baumhöhe bezeichnet man die Anzahl der Stufen oder auch Generationen eines Baumes. Die Baumhöhe ist die Länge des längsten Weges von der Wurzel zu einem Knoten + 1.
- Ein binärer Baum ist ein Baum, dessen Knoten jeweils höchstens zwei Knoten haben.
- Eine Menge von Bäumen heißt Wald[1].

Jeder Knoten kann auf eindeutigem Weg über Kanten, die Verbindungen zwischen den Knoten, von der Wurzel aus erreicht werden.

[1] Sag bloß! Echt?

3.6.2 Datenstruktur binärer Bäume

Schematische Struktur eines Knotens in einem binären Baum:



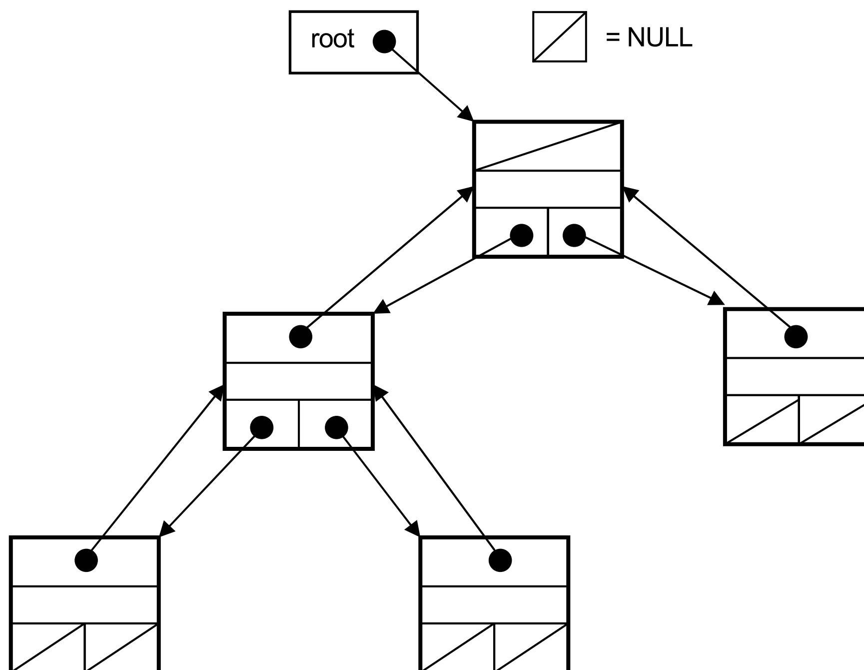
Ein Knoten besteht aus drei Pointern: einer auf den Vater(parent) und zwei auf die Söhne(left_child und right_child) sowie den Daten.

Wenn parent(x) = NULL, so ist x die Wurzel. Wenn left_child(x) = NULL oder right_child(x) = NULL, so ist der entsprechende Sohn nicht vorhanden.

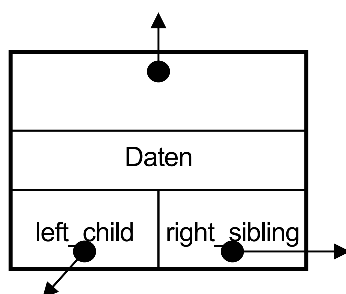
Sind sowohl left_child(x) wie auch right_child(x) = NULL, so ist x ein Blatt des Baumes.

Die Variable root(x) enthält einen Zeiger auf die Wurzel des Baumes. Ist root = NULL, so ist der entsprechende Baum leer.

Ein typischer binärer Baum



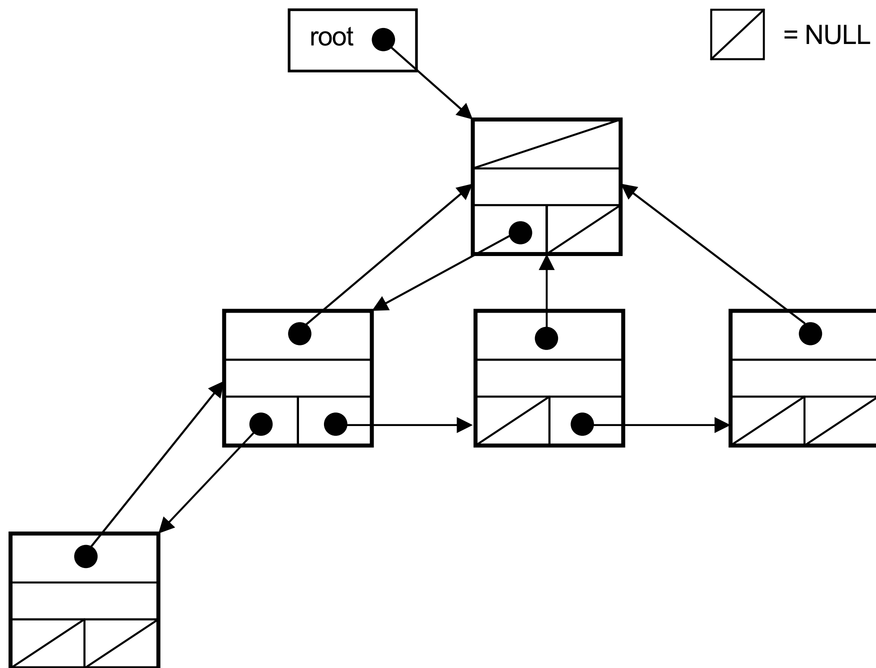
3.6.3 Verallgemeinerung auf Bäume mit mehr als zwei Söhnen



Will man einen Baum mit k Söhnen implementieren, kann man zunächst wie beim binären Baum vorgehen: Wenn max. k Söhne vorhanden sind, kann man statt den zwei Pointern left_child und right_child k Pointer child_1 bis child_k verwenden.
Diese Methode hat den Nachteil, daß bei großem k und stark schwankender Anzahl von Söhnen viele unnötige Pointer deklariert werden. Außerdem funktioniert dieser Ansatz überhaupt nicht, wenn k unbekannt ist.
Um diese Probleme zu umgehen, wendet man folgende Struktur an: Jeder Knoten besitzt einen Pointer auf seinen Vater, einen Pointer auf den linken Sohn und einen auf seinen rechten „Bruder“ (right_sibling), also auf den Knoten mit dem selben Vater, der rechts von ihm steht.

Bei dieser Variante benötigt man nicht mehr Zeiger als bei einem binären Baum und kann doch beliebig viele Söhne pro Vater zulassen, wobei die Anzahl gar nicht bekannt sein muß!

Ein Baum mit mehr als zwei Söhnen



Hier ist left_child = NULL, falls es keine Söhne gibt und right_sibling = NULL, wenn der Knoten bereits „rechtester“ Bruder ist.

Natürlich sind auch andere Darstellungen von Bäumen möglich, z.B. wie von Heap Sort bekannt in einem eindimensionalen Feld oder, wie für manche Anwendungen ausreichend, nur mit einem Pointer auf den Vater oder ohne die Pointer auf die Brüder.

4 Elementare Suchverfahren

Problem: Suche in großem Mengen von Datensätzen nach einem Schlüssel, z.B. Stichwörter in einem Wörterbuch, Telefonnummern in einem Telefonbuch.

Zunächst verwenden wir nur Systeme mit Schlüsselvergleichen und keine speziellen Datenstrukturen, sondern nur lineare Listen.

4.1 Auswahlproblem

Input: Datensätze mit Schlüsseln $<k_1, \ldots, k_n>$, eine natürliche Zahl l
Output: Element mit l-kleinstem Schlüssel, d.h. Datensatz mit kleinsten, zweitkleinsten… Schlüssel.
Spezialfall: Finde den Median, d.h. das mittlere Element, also das Element n/2, falls n gerade oder die Elemente (n/2) und (n/2)+1, falls n ungerade.

Die einfachste Lösung für diese Problem ist natürlich, die Datensätze mittels Heap Sort oder Merge Sort zu sortieren und dann das **l**-te Element der Menge zu suchen. Die Laufzeit ist bei beiden $O(n \cdot \log(n))$.
Es geht aber auch schneller, und zwar in $O(n)$!

Beispiel: Um festzustellen, ob ein Element das Größte oder das Kleinste ist, sind nur n–1 Schlüsselvergleiche nötig. Einen allgemeinen Algorithmus dazu wollen wir hier nicht behandeln[1].

4.2 Suche in linearen Listen

Input: Liste $<a_1, \ldots, a_n>$
Output: Ist Schlüssel enthalten und wenn ja, an welcher Stelle?

4.2.1 Sequentielle Suche

```
sequential_search(a,key) {
for ( i=n ; i≥1 ; i-- )
{
    if ( a[i] == key ) return i;
}
return („nicht vorhanden““); }
```

Laufzeit:

- Im best case $O(1)$, z.B. a(n) = key.
- Im worst case $O(n)$, z.B. a[1] = key.
- Im Durchschnitt, falls key dabei an allen Stellen in der Liste gleich häufig vorkommt:

$$T(n) = \frac{1}{n}\sum_{i=1}^{n} i = \frac{n+1}{2}, \text{ also } T(n) \in O(n)!$$

[1] genaueres dazu im Buch von Cormen

4.2.2 Binäre Suche

Bei der binären Suche geht man davon aus, daß alle k_i verschiedene, natürliche, sortierte Zahlen sind. Der Algorithmus ist vom Typ divide&conquer. Er geht nach folgendem Schema vor:

- vergleiche das mittlere Element mit dem key
- bei Treffer STOP, ansonsten links weitersuchen wenn key kleiner und rechts weitersuchen wenn key größer war als das mittlere Element.

Laufzeit: Im worst case und im Durchschnitt ist die Laufzeit $O(\log(n))$, da $T(n) = T(n/2) + 1\Theta(1)$.

4.2.3 Interpolationssuche

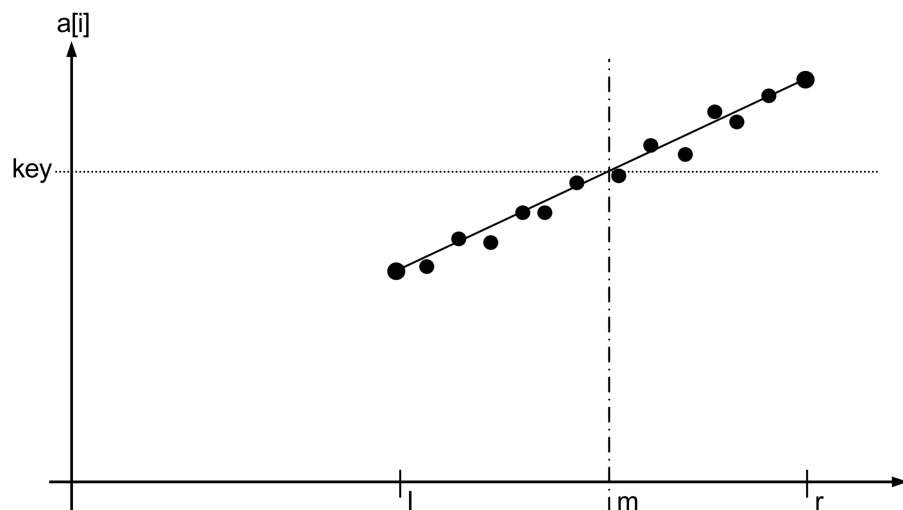
Dies ist eine verbesserte binäre Suche. Man geht davon aus, daß in a[1]…a[r] ein lineares Wachstum herrscht.

Wähle dann $m = l + \frac{key - a[l]}{a[r] - a[l]}(r - l)$. m ist der Index, für den gilt: $c_1 + c_2 \cdot i = key$

Dies ist eine bessere Wahl als ½.

Satz: Sind alle Schlüssel unabhängig und gleich verteilt, dann ist die Laufzeit im Durchschnitt $O(\log(\log(n)))$. Dies ist eine sehr langsam wachsende Funktion!

Illustration:



5 Suchen in Bäumen

5.1 Einführung

Bäume sind besonders für eine Implementierung von „Wörterbüchern“ geeignet.

Es gibt drei wichtige **Operationen:**

- suchen
- löschen
- einfügen

Nomenklatur:

- Als „innerer Knoten“ bezeichnen wir einen Knoten mit mindestens einem Sohn.
- Ein „Blatt“ oder „äußerer Knoten“ hat keine Söhne.
- Die Ordnung eines Baumes ist d, wenn pro Vater maximal d Söhne existieren.

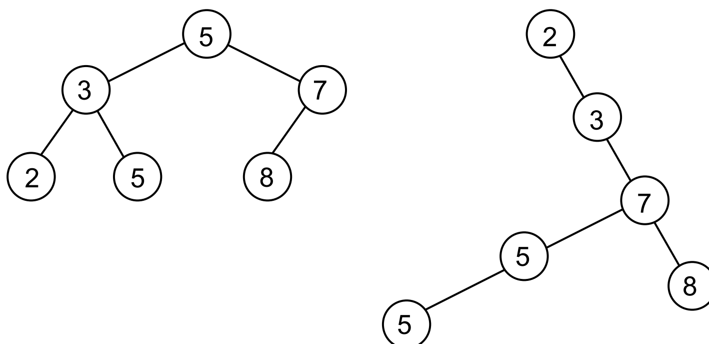
5.2 Binäre Suchbäume

5.2.1 Definition

Binäre Suchbäume haben die Ordnung 2. Jedem Knoten ist ein Schlüssel key(x) zugeordnet. Wenn x ein beliebiger Vaterknoten ist muß ein binärer Suchbaum folgende Eigenschaften (binary search tree properties) erfüllen:

- Für alle Knoten y im linken Teilbaum (subtree) gilt: key(y) ≤ key(x)
- Für alle Knoten z im rechten Teilbaum (subtree) gilt: key(z) ≥ key(x)

Beispiel:



Achtung: Diese beiden Bäume haben die gleiche Schlüsselmenge und erfüllen auch beide die Kriterien eines binären Suchbaums!

5.2.2 Codes der Operationen eines BSB

Ausgabe des gesamten Baums

Aufgabe: Sortierte Ausgabe aller Schlüssel ab Knoten x abwärts, also im Teilbaum unter x.

```
inorder_tree_walk(x) {
if ( x ≠ NULL )
{
     inorder_tree_walk(left(x));
     print key(x);
     inorder_tree_walk(right(x));
} }
```

Die Beispiele in 5.2.1 liefern beide <2, 3, 5, 5, 7, 8>.

Die **Laufzeit** ist $O(n)$, denn jeder Knoten ruft 2 mal inorder_tree_walk auf.
Daraus kann man folgern, daß das Sortieren einer beliebigen Menge mit Schlüsselvergleichen im binären Suchbaum mindestens $\Omega(n \cdot \log(n))$ kostet.

Suche

Input: Pointer auf die Wurzel und Schlüsselwert k
Output: Pointer auf den Knoten, der Schlüssel k enthält oder NULL, falls Schlüssel im Baum nicht vorhanden.

```
x = root(T)
tree_search(x,k) {
if ( x == NULL || k == key(x) ) return(x);
if ( k<key(x) ) return(tree_search(left(x),k));
else return(tree_search(right(x),k)); }
```

Der **Aufwand** ist hier $O(h)$, $h \geq \log_2(n)+1$. Es gibt auch eine nicht rekursive Vorgehensweise:

```
iterative_tree_search {
while ( x ≠ NULL && k ≠ key(x) )
{
     if ( k<key(x) ) x = left(x);
     else x = right(x);
}
return(x); }
```

Finden des Minimums bzw. Maximums

```
tree_minimum(T) {
x = root(T)
while ( left(x) ≠ NULL ) x = left(x);
return(x); }
```

Der **Aufwand** beträgt $O(h)$.

Finden des Vorgängers und des Nachfolgers

```
tree_sucessor(T,x) {
if ( right(x) ≠ NULL )return(tree_minimum(right(x));
else y = parent(x);
while ( y ≠ NULL && x > right(y) )
{
     x = y;
     y = parent(x);
}
return(y); }
```

Einfügen eines Knotens

```
tree_insert(T,z) {
y = NULL;
x = root(T);
while ( x ≠ NULL )
{
     y = x;
     if ( key(z) < key(x) ) x = left(x);
     else x = right(x);
}
parent(z) = y;
if ( y = NULL ) root(T) = z;
else
{
     if ( key(z) < key(y) ) left(y) = z;
     else right(y) = z;
} }
```

Löschen eines Knotens

Beim Löschen eines Knotens k aus einem binären Suchbaum können drei verschiedene Fälle auftreten:

(1) k hat keine Söhne ⇒ k löschen und beim Vater von k den Pointer auf k = NULL setzen

(2) k hat einen Sohn ⇒ k löschen und Verbindung zwischen Vater und Sohn von k herstellen. Der Unterbaum des Sohnes rück hier eine Generation nach oben

(3) k hat zwei Söhne ⇒ Nachfolger y im rechten Unterbaum von k suchen. y löschen und in den Baum an Stelle von k einfügen (Fall 1 oder 2).

```
tree_delete(T,2) {
if ( left(z) == NULL || right(z) == NULL ) y = 2;
else y = tree_sucessor(z);
if ( left(y) ≠ NULL ) x = left(y);
else x = right(y);
if ( x ≠ NULL ) parent(x) = parent(y);
if ( parent(y) == NULL ) root(T) = x;
else if ( y == left(parent(y) ) left(parent(y)) = x;
else right(parent(y))=x;
if ( y ≠ z ) key(z) = key(y);

return(y); }
```

Die **Laufzeit** ist $O(h)$.

Einfügen

```
tree_insert(T,Z) {
```

```
y = NULL;
x = T->root;
while ( x ≠ NULL) {
     y = x;
     if (z->key < x->key) x = x->left;
     else  x = x->right; }
z->parent = y;
if ( y == NULL ) T->root = Z;
else if (z->key < y->key ) {
     y->left = z;
     else y->right = z; }
}
```

Diese Funktion deckt folgende Fälle ab:

- Baum war bisher leer, dann ist T->root = NULL, die while-Schleife wird gar nicht durchlaufen, und z wird als root des Baums eingefügt.
- Der Baum enthält bereits Elemente, dann führt die Funktion an das Ende eines Astes.

Die **Laufzeit** beträgt $O(h)$.

Löschen

Aufgabe: Lösche Knoten z aus dem binären Baum. Die Funktion berücksichtigt folgende Fälle:

(1) Z hat keine Söhne ⇒ Knoten löschen und den Pointer des Vates auf NULL setzen

(2) Z hat einen Sohn ⇒ Z löschen und Pointer des Vaters auf den Sohn von z zeigen lassen.

(3) Z hat zwei Söhne ⇒ Nachfolger y von z suchen, der keinen linken Sohn hat. y dann gemäß Fall 1 oder 2 löschgen und statt z in den Baum einfügen.

5.2.3 Komplexität der Basisoperationen bei binären Suchbäumen

Suchen, Minimum, Maximum, Nachfolger, Vorgänger, einfügen und entfernen haben $O(h)$, wobei $h \geq \log_2(n)$ die Baumhöhe bezeichnet. Dies ist schnell für kleines h, d.h. $h \approx \log_2(n)$. Wie bereits bekannt ist im optimalen Fall die Höhe des Baumes $\log_2(n)+1$. Im Extremfall kann der binäre Baum aber zu einer linearen Kette entarten (z.B. wenn eine sortierte Folge in den Baum einsortiert wird), dann ist h = n. In diesem Fall sinkt die Laufzeit für Basisoperationen auf $O(n)$.
Es gibt verschiedene Methoden, die Suchbäume so zu entwerten, daß sie ihr vorteilhaftes Verhalten nicht verlieren. Man spricht dann von ausgeglichenen oder balancierten Bäumen. Ein Beispiel dafür sind die Rot-Schwarz-Bäume.

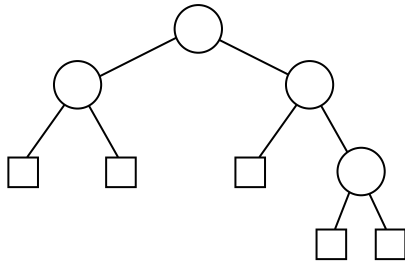
5.3 Rot-Schwarz-Bäume (RSB)

5.3.1 Allgemeines

Ziel: Erhaltung von balancierten binären Suchbäumen mit niedriger Höhe, h möglichst nahe bei $2 \cdot \log_2(n)$.

Vorgehensweise: Jeder Knoten wird um ein Bit erweitert: color(x) = rot oder schwarz. Dabei gilt folgende Konvention: Der Baum soll (formal) in leeren Blättern enden, d.h. in Knoten ohne Daten. Damit sind die datentragenden Knoten immer innere Knoten.

Die allgemeine Struktur sieht also folgendermaßen aus:



Definitionen

Sei T ein binärer Suchbaum. T ist ein RSB, wenn er die folgenden Bedingungen erfüllt:

(1) Jeder Knoten ist entweder rot oder schwarz.

(2) Jedes Blatt ist schwarz.

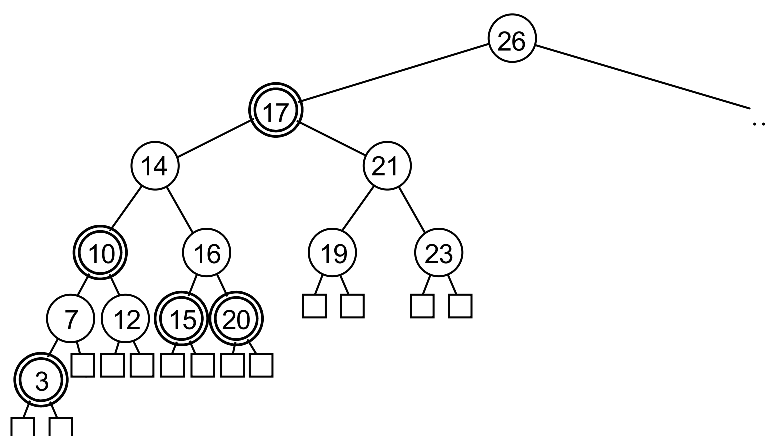
(3) Söhne von roten Knoten sind immer schwarz.

(4) Alle Wege von einem beliebigen Knoten x im Baum zu den Blättern haben gleich viele schwarze Knoten.

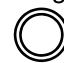
(5) Die Wurzel ist immer schwarz.

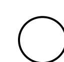
Folgerungen:
(3) besagt, daß auf jedem Weg von einem Knoten zu einem Blatt mindestens so viele schwarze wie rote Knoten sind.
Die „Schwarzhöhe" sh(x) ist die Anzahl der schwarzen Knoten auf irgendeinem Weg von einem Knoten x zu einem Blatt (ohne x). Nach Punkt 4 ist diese eindeutig.
h(x) ist die maximale Anzahl von Knoten auf allen möglichen Wegen von x zu einem Blatt (ohne x).
$\Rightarrow$ Nach (3) gilt dann $sh(x) \geq h(x)/2$!

Beispiel eines RSB



Farben werden folgendermaßen dargestellt:

 ein "roter" Knoten

 ein „schwarzer" Knoten

Man beachte, daß auf schwarze Knoten sowohl rote wie auch schwarze Knoten folgen können.

5.3.2 Die maximale Höhe eines RSB

Satz

Ein RSB mit n inneren Knoten hat eine Höhe von $h \leq 2 \cdot \log_2(n+1)$

Beweis

Lemma: Sei x ein beliebiger Knoten und T(x) der zugehörige Unterbaum, dann hat T(x) hat mindestens $2^{sh(x)}-1$ innere Knoten.

Beweis per Induktion:
Induktionsanfang: Sei h(x) = 0, d.h. x ist leeres Blatt. Der Unterbaum T(x) hat $2^0-1 = 0$ innere Knoten.

Sei h(x) > 0, x sei innerer Knoten mit den Söhnen y und z.
Induktionsvoraussetzung: # bezeichne die Anzahl der inneren Knoten. Es gilt dann:

$$\#T(y) \geq 2^{sh(y)}-1 \text{ und } \#T(z) \geq 2^{sh(z)}-1.$$

Hier muß man zwei Fälle unterscheiden:

(1) Sohn ist rot: sh(y) = sh(x) bzw. sh(z) = sh(x)

(2) Sohn ist schwarz: sh(y) = sh(x)–1 und sh(z) = sh(x)–1 (klar, da $h(y) \leq h(x)-1$ bzw. $h(z) \leq h(x)-1$)

Nach Annahme sind in jedem Unterbaum mindestens $2^{sh(x)-1}-1$ innere Knoten, also sind in T(x) mindestens $2 \cdot (2^{sh(x)-1}-1)+1 = 2^{sh(x)}-1$ innere Knoten. q.e.d.

Sei r die Wurzel des Baumes und h(r) die Baumhöhe.
Dann ist nach dem eben bewiesenen Lemma $n \geq \#T(r) \geq 2^{sh(x)}-1$. Nach Punkt 3 gilt, daß $sh(r) \geq h(r)/2$.
Damit ist $n \geq 2^{h(r)/2}-1$, also gilt $\log_2(n+1) \geq h(r)/2$ bzw. $h(r) \leq 2 \cdot \log_2(n+1)$. q.e.d.

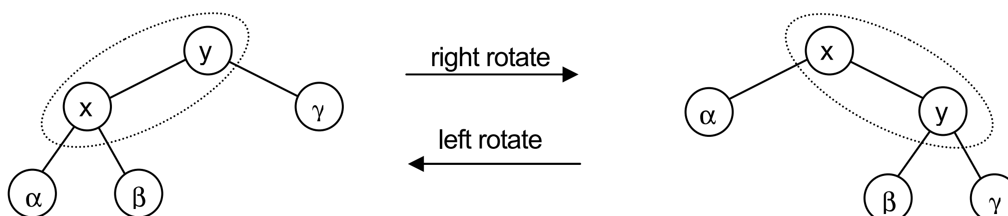
Folgerungen: Im RSB gilt also immer, daß alle elementaren Operationen $O(\log(n))$ kosten. Aber sowohl das Einfügen wie auch das Entfernen können die RSB-Struktur zerstören, deshalb müssen diese Routinen für einen RSB neu entwickelt werden.

5.3.3 Einfügen in einem RSB

Die neue Einfügen-Operation im RSB benötigt eine Hilfoperation:

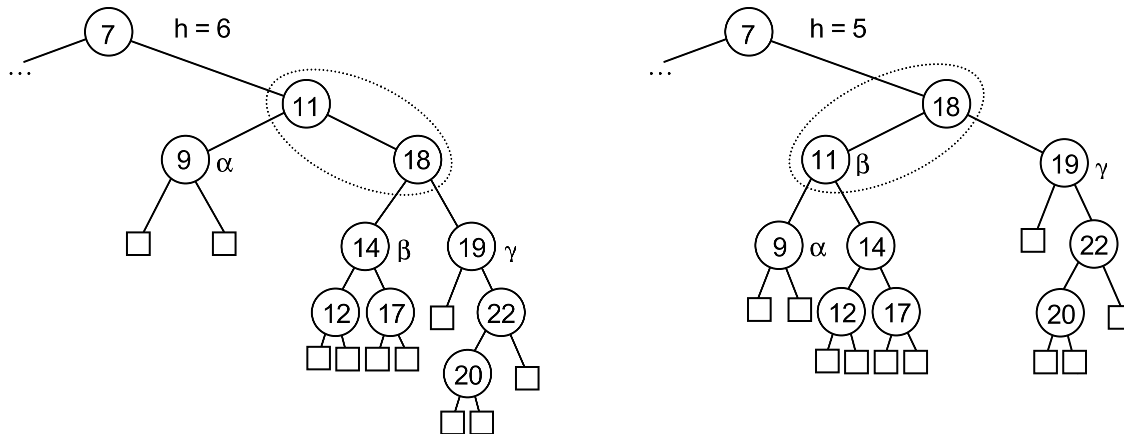
Rotation

Der Algorithmus zum Einfügen in einem RSB benötigt die Hilfsoperation Rotation. Sie erhält die BSB-Eigenschaften und stellt die RSB-Eigenschaften her.



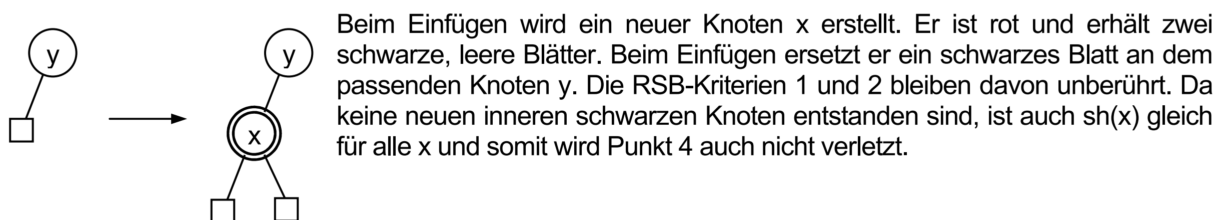
α, β und γ bezeichnen hier beliebige Unterbäume. Offensichtlich bleibt hier die RSB-Struktur erhalten. Der Aufwand ist gering, es werden nur Pointer umgestellt, die Laufzeit ist $O(1)$.

Beispiel:



Bemerkung: left_rotate und right_rotate sind spiegelsymmetrisch oder dual, d.h.. hintereinander ausgeführt erhält man wieder den unveränderten Baum.

Einfügen



Beim Einfügen wird ein neuer Knoten x erstellt. Er ist rot und erhält zwei schwarze, leere Blätter. Beim Einfügen ersetzt er ein schwarzes Blatt an dem passenden Knoten y. Die RSB-Kriterien 1 und 2 bleiben davon unberührt. Da keine neuen inneren schwarzen Knoten entstanden sind, ist auch sh(x) gleich für alle x und somit wird Punkt 4 auch nicht verletzt.

Nur bei Punkt 3 kann es zu Komplikationen kommen:

- Ist y schwarz, so bleibt die RSB-Struktur erhalten, Punkt 3 wird nicht verletzt
- Ist y dagegen rot, so ist der Sohn eines roten Knotens rot, was nach Punkt 3 nicht erlaubt ist.

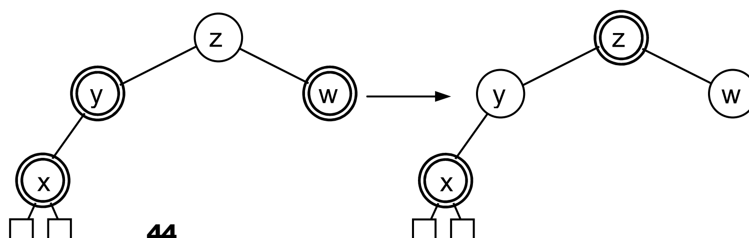
Um diese Problem zu beheben, muß man jetzt umfärben und ggf. rotieren. Dabei entstehen 6 Fälle, wobei drei davon spiegelsymmetrisch sind und mit 1‘, 2‘ und 3‘ bezeichnet werden.

Fall 1(a):

- x ist linker Sohn von y und beide sind rot.
- w sei rechter Sohn von z, w ist rot.

Man muß den Baum nach oben umfärben:

- y wird schwarz
- z muß rot werden wegen RSB-Eigenschaft (4).
- w muß schwarz werden wegen RSB-Eigenschaft (3) und (4).



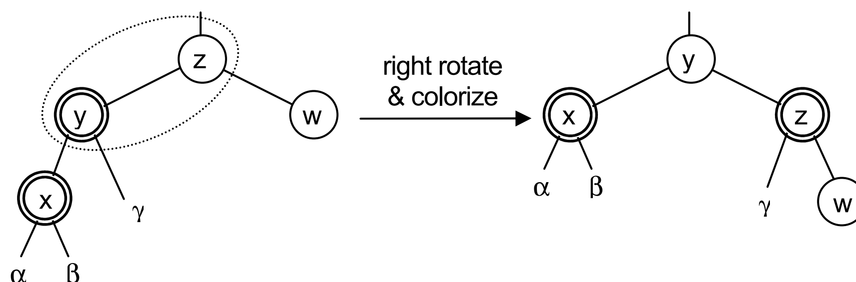
Bei z muß man eine Fallunterscheidung durchführen:

- Ist der Vater von z schwarz, treten keine weiteren Widersprüche auf.
- Ist z die Wurzel, kann man z schwarz färben und es tritt kein Konflikt auf.
- Ist aber der Vater von z rot, hat man wieder Fall 1, nur zwei Generationen weiter oben.

Fall 2:

- x ist linker Sohn von y, beide sind rot
- w ist rechter Sohn von z, w ist schwarz

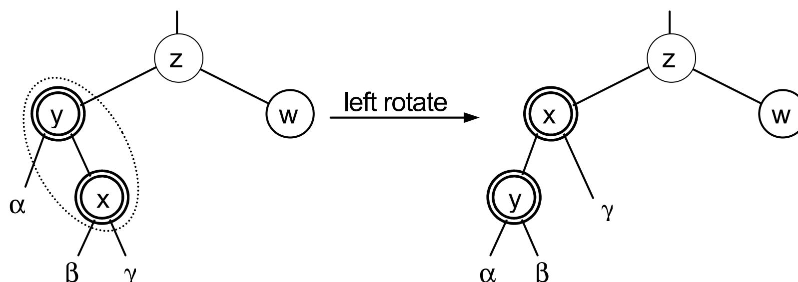
In diesem Fall existiert eine einfachere Lösung. Man führt right rotate auf z aus und färbt anschließend y schwarz und z rot.



Fall 3:

- x ist rechter Sohn von y, beide sind rot.
- w ist rechter Sohn von z. w ist schwarz

Man führt hier left rotate auf y aus und erhält eine Konstellation, auf die man Fall 2 anwenden kann.



Hierzu gibt es drei symmetrische Fälle, die sich aus folgenden Konstellationen ergeben:

- y ist rechter Sohn von z
- w ist linker Sphn von z

Laufzeitanalyse

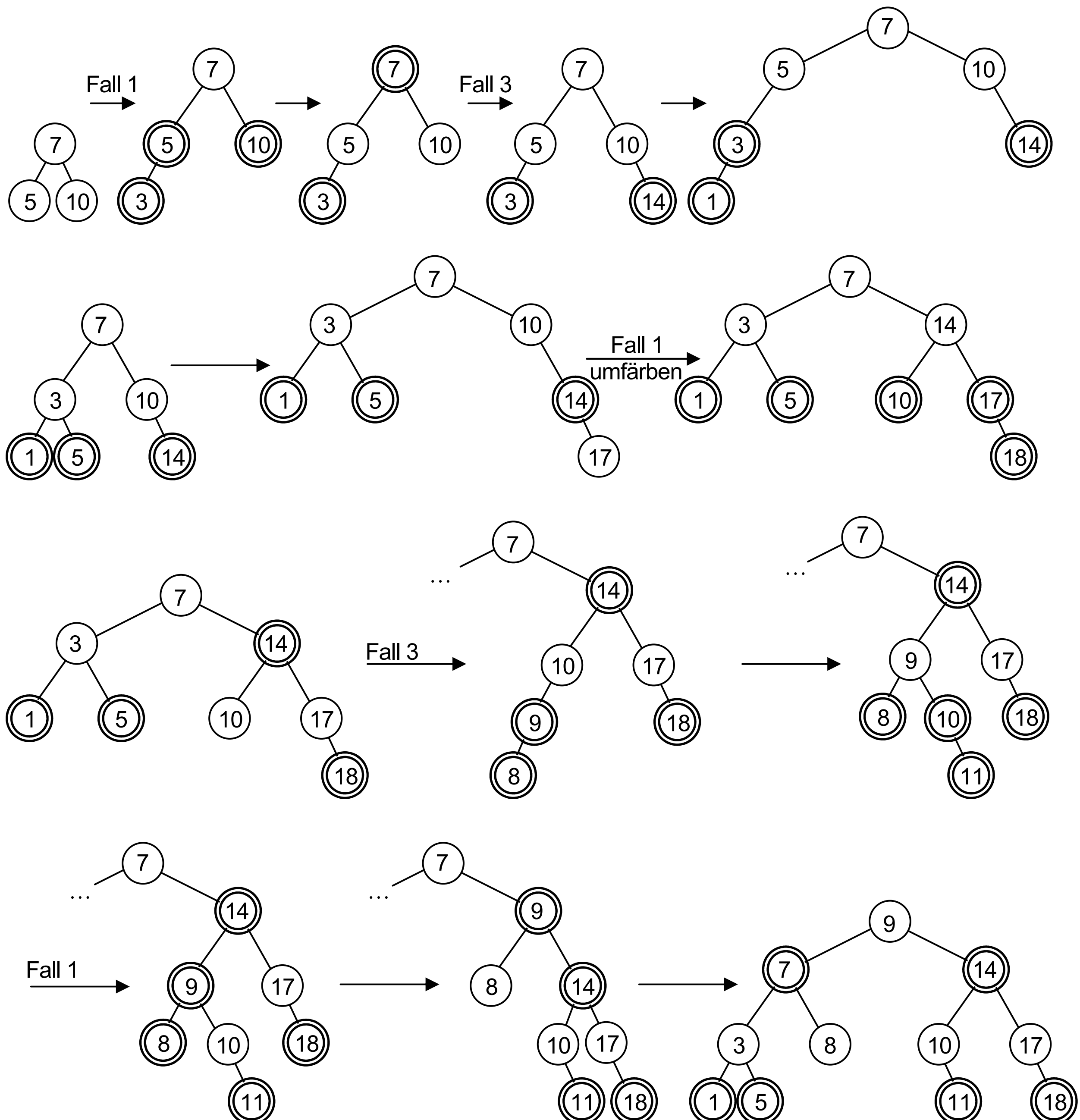
Die Höhe des RSB ist $O(\log_2(n))$, d.h. im Fall 1 sind maximal $\log_2(n)$ Wiederherstellungen nötig, um die RSB-Kriterien zu erfüllen.
Normales Einfügen im binären Suchbaum kostet $O(\log_2(n))$. Um zusätzlich die RSB-Eigenschaften zu erfüllen, sind im Fall 1 $O(\log_2(n))$ oder in den Fällen 2 und 3 $O(1)$ Operationen notwendig.

5.3.4 Entfernen eines Knotens

Das Entfernen eines Knotens verläuft analog zum Einfügen. Ist x rot wird die RSB-Struktur gar nicht gestört. Ist x schwarz, gibt es sogar 8 verschiedene Fälle, die wir hier allerdings nicht behandeln. Interessierte seine hier auf das Buch von Cormen verwiesen.

5.3.5 Beispiel für das sukzessive Einfügen einer Folge in einen RSB

Es wird die Folge <3,14,1,17,18,9,8,11> in einen RSB eingefügt. Aus Gründen der Übersichtlichkeit werden die schwarzen, leeren Blätter weggelassen.



Wie man klar erkennen kann, erzeugen die RSB-Kriterien einen ausgewogenen Baum.

6 Hashtabellen

Einen völlig anderen Ansatz zur Verwaltung von Daten verwenden die Adresstabellen, die völlig ohne Schlüsselvergleiche auskommen.

Idee:
Den Schlüssel eines Datensatzes wird direkt oder indirekt als Index einer Tabelle verwendet. Dazu müssen die Schlüssel allerdings ganzzahlig sein.

Operationen:

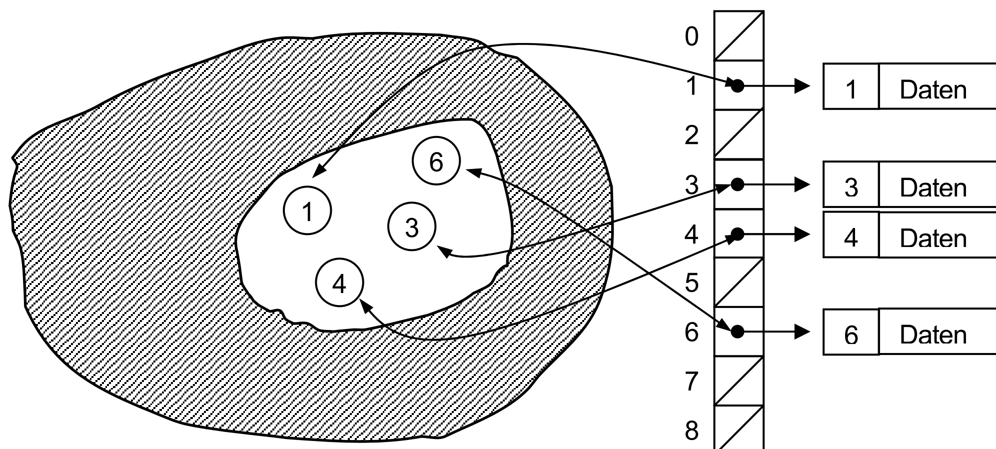
- suchen
- entfernen
- einfügen

6.1 Direkte Adressierung

Die direkte Adressierung (direct adress tables) ist eine einfach Variante, die allerdings nur eingesetzt werden kann, wenn sehr spezielle **Voraussetzungen** erfüllt sind:

- Alle Schlüssel sind ganzzahling und liegen zwischen 0 und m–1.
- m ist nicht zu groß.
- Jeder Schlüssel kommt nur einmal vor.

Idee:
Es wird eine Adresstabelle T angelegt, so daß T(k) auf den Datensatz mit dem Schlüssel k zeigt. Falls nicht k existiert ist T(k) = NULL.



Operationen:
Die Operationen sind sehr schnell und einfach implementierbar.

```
direct_address_search(k) {
return(T(k)) }
/* liefert Adresse von */

direct_address_insert(x) {
T(key(x)) = x }
/* erzeugt Adresse x in Tabelle */
```

```
direct_address_delete(x) {
T(key(x)) = NULL }
/* entfernt Adresse x aus Tabelle */
```

Die **Laufzeit** ist hier immer O(1), auch im worst case!

Problem:
Es muß für alle möglichen Schlüssel Speicherplatz zur Verfügung gestellt werden, auch wenn nur sehr wenige davon wirklich auftreten. In diesen Fällen wird sehr viel Speicherplatz verschwendet. Dieses Problem lösen die Hashtabellen.

6.2 Hashtabellen

6.2.1 Einfache Hashtabellen

Idee:

- Verwende nur eine kleine Tabelle T[0]…T[l–1] ; $l \approx n$
- Verwende eine Hashfunktion k(0)…k(m-1), die jedem möglichen Schlüssel $k \in S$ einen Tabellenplatz zuweist. Die Adresse wäre dann idealerweise T[k(k)].

Beispiel einer Hashfunktion:

h(k) = kmod(l) „Rest der ganzzahligen Division von k durch l"

l = 8: h(8) = 0
h(i) = i, $i \in \{0,\dots,7\}$
h(9) = 1

Vorteile:

- Laufzeit aller elementaren Operationen ist O(1)
- Speicherverbauch nur O(n)

Problem:
$n/l = \alpha$, α wird Lastfaktor genannt. Für $\alpha < 1$ treten keine Probleme auf. Ist jedoch $\alpha \geq 1$, so ist die Liste überbelegt und es lassen sich keine weiteren Daten dort unterbringen. Durch die Wahl von l kann man bei Implementation Einfluß darauf nehmen, wie schnell dieser Fall eintritt.

Bei Hastabellen kann aber auch folgendes Problem auftreten, da m<n:
Zwei Schlüssel $k_1 \neq k_2$ weisen auf den gleichen Platz: $k(k_1) = k(k_2)$! Dieses Problem läßt sich vermeiden, indem man T[i] zu einer verketteten Liste macht. („chained hashing")

6.2.2 Verkettete Hashtabellen („chained hashing")

Idee:
Alle Elemente mit gleichem Wert von $k(k_i)$ werden in einer einfach verketten Liste gespeichert. T(k(k)) verweist dann auf den Anfang der Liste.

```
chained_hash_insert(x) {
/* füge x am Kopf von T[key(x)] hinzu */

chained_hash_search(k) {
/* suche Element mit Schlüssel k in der Liste T[k(k)] */
```

```
chained_hash_delete(x) {
/* suche Adresse x in der Liste T[k(key(x))] und entferne */
```

Laufzeit:
Der Aufwand für Insert ist $O(1)$, falls man nicht überprüfen muß, ob schon ein Element in der Liste ist. Für search und delete ist der Aufwand abhängig von der Listenlänge.

Der worst case tritt ein, wenn eine schlechte Hashfunktion alle n Schlüssel dem gleichen Tabellenplatz zuordnet. In diesem Fall dauert das Suchen und Entfernen $O(n)$.

Zur Berechnung der durchschnittlichen Laufzeit geht man davon aus, daß alle n Schlüssel mit gleicher Wahrscheinlichkeit jedem Platz $i \in \{0...l-1\}$ zugeordnet werden. („simple uniform hashing") In diesem Fall ist jede Liste im Durchschnitt α lang.
Als Aufwand ergeben sich Fixkosten von $O(1)$, bei erfolgloser Suche $O(\alpha)$, bei erfolgreicher ca. $O(\alpha/2) = O(\alpha)$, also insgesamt $O(1) + O(\alpha)$.
Sorgt man bei der Implementation dafür, daß l proportional zu m ist, (α Proportionalitätsfaktor) folgt also: Aufwand konstant $O(1)$ für $n \to \infty$.

6.2.3 Andere Hashfunktionen

Multiplikationsvariante:
$h(k) = \lfloor l \cdot (\beta \cdot k \bmod (1)) \rfloor = 0{,}61803\ldots$
$0 < \beta < 1$: wähle $\left(\sqrt{5} - 1\right)/2 \Rightarrow \beta \cdot k \bmod (1) = \beta \cdot k - \lfloor \beta \cdot k \rfloor \Rightarrow$ goldener Schnitt
„Nachkommazahl von $\beta \cdot k$"

Random Hashfunktion:
Ziele bei der Wahl einer Hashfunktion sind:

- Alle Elemente von T sind erreichbar
- Die Elemente sind möglichst gleichverteilt
- Elemente sind möglichst gleich (zufällig) über die Liste verteilt

$\Rightarrow$ Wähle h(k) durch einen Pseudozufallszahlengenerator.

Eine ganz andere Variante zum Auffinden eines freien Listeplatzes verwendet das offene hashing.

6.3 Offene Hashfunktionen

Offen Hashfunktionen suchen anhand eines eindeutigen Algorithmus freie Plätze in der Tabelle und speichern die Adresse und den Schlüssel dort ab.
Dazu verwendet man eine allgemeine Hashfunktion, die einen Startplatz $h_1(k)$ bereitstellt und sucht dann systematisch weitere Kandidatenplätze.

Die Hashfunktion benötigt zwei Inputwerte: $h(k,i) \in \{0...l-1\}$. k ist der Schlüssel und i die Nummer des Versuchs. Die Aufrufe bei der Suche eines freien Platzes sehen dann so aus:
$h(k,0)$, falls belegt $\to h(k,1)$, falls belegt $\to h(k,2) \ldots$

6.3.1 Lineares Sondieren

Sei h eine Hashfunktion (z.B. kmod(l)). Definiere dann eine Erweiterung:

$$h(k,i) = (h(k)+i) \bmod (l)$$

Beispiel: (die gefundenen Listenplätze sind unterstrichen)

k	**10**	**19**	**31**	**22**	**14**	**16**
i=0	2	3	7	6	6	0
i=1	3	4	0	7	7	1
i=2	4	5	1	0	0	2

Ergebnistabelle bei linearem Sondieren:

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
14	16	10	19	–	–	22	31

Es wurden im Durchschnitt 1,5 Versuche (Vergleiche) gebraucht, bis ein leerer Listenplatz gefunden wurde.

Zum Auffinden eines Schlüssels durchsucht man auf die selbe Weise die Liste, wobei man darauf achten muß, daß die Suchfunktion folgendes Problem behebt:
Sei k_1 an der Stelle j_1 in der Tabelle gespeichert. Jetzt soll der Schlüssel k_2 eingeordnet werden und $k(k_2) = j_1$. Da j_1 belegt ist, wird durch das Sondieren ein neuer Listenplazu $j_2 \neq j_1$ für k_2 gefunden.
Wird jetzt zuerst k_1 aus der Liste entfernt und dann k_2 gesucht, so findet die die Hashfunktion $k(k_2)$ an der Stelle j_1 einen leeren Tabellenplatz vor und könnte irrtümlich zurückmelden, daß j_2 nicht in derTabelle existiert.

Das eigentliche Problem beim linearen Sondieren besteht darin, daß die belegten Teilstücke schnell anwachsen. („clustering“) Alle elementaren Funktionen müssen erst den gesamten „Cluster“ durchsuchen.

6.3.2 Quadratisches Sondieren

Wie beim linearen Sondieren wird hier an einer Stelle $h_1(k)$ mir der Suche begonnen, jedoch wird die Schrittweite für das Sondieren durch eine quadratische Funktion bestimmt:

Allgemeine Form:

$$h(k,i) = (h'(k)+c_1 \cdot i+c_2 \cdot i^2)\mathrm{mod}(l)$$

z.B. l=8, c_1=0,5, c_2=0,5 $\Rightarrow$ h'(k) = +1, +3, +6, +10…

k	**10**	**19**	**31**	**22**	**14**	**16**
i=0	<u>2</u>	<u>3</u>	<u>7</u>	<u>6</u>	6	<u>0</u>
i=1	3	4	0	7	7	1
i=2	5	6	2	1	<u>1</u>	3

Ergebnistabelle beim quadratischen Sondieren:

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
16	14	10	19	–	–	22	31

Hier wurden im Durchschnitt 1,33 Vergleiche gebraucht, bis ein leerer Tabellenplatz gefunden wurde. Ein Nachteil ist aber, daß bei gleichem Startplatz der Schlüssel die Folge der Sondierungsschrittweiten gleich bleibt. („secondary clustering“)

Dieses Problem kann man damit beheben, indem man auch die Schrittweite durch den Schlüssel bestimmt.

6.3.3 Double Hashing

Diese Variante kombiniert zwei Hashfunktionen:

$$h(k,i) = (h_1(k) + i \cdot h_2(k))\mathrm{mod}(l)$$

Es hängen also sowohl der Startplatz wie auch die Schrittweite von vom Schlüssel k ab.

Damit die gesamte Tabelle durchsucht wird, dürfen die Werte von $h_2(k)$ keinen gemeinsamen Teiler d mit l haben, sonst werden nur der l/d Teil der Tabelle durchsucht. Dies kann man auf folgende Weisen sicherstellen:

Beispiel:

- $l=2^p$, $h_2(k)$ ungerade, z.B. 2h–1
- l ist Primzahl, $0 < h_2(k) < l$, z.B. $h_2(k)$ = 1+kmod(T), T = l–1 oder l–2

6.3.4 Laufzeitenanalysen

Es gibt l! verschiedene Sondierungsreihenfolgen.
Ideal wäre das sog. „uniform hashing", d.h. für einen zufällig gewählten Schlüssel wird jede Sondierungsreihenfolge mit der Wahrscheinlichkeit 1/l! gewählt.

Sei α = m/l. Man kann dann zeigen, daß folgende Laufzeiten für erfolgreiche (ER) und erfolglose (EL) Suche gelten.

- lineares Sondieren:

  EL: $\approx \frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$ ER: $\approx \frac{1}{2}\left(1+\frac{1}{(1-\alpha)}\right)$

- quadratisches Sondieren:

  EL: $\approx \frac{1}{1-\alpha}-\alpha+\ln\left(\frac{1}{1-\alpha}\right)$ ER: $\approx 1+\ln\left(\frac{1}{1-\alpha}\right)-\frac{\alpha}{2}$

- uniform hashing: (wird von double hashing fast erreicht)

  EL: $\approx \frac{1}{1-\alpha}$ EL: $\approx \frac{1}{\alpha}\ln\left(\frac{1}{1-\alpha}\right)$

α	**Verkettung**		**lin. Sondieren**		**quad. Sondieren**		**uniform hashing**	
	EL	ER	EL	ER	EL	ER	EL	ER
0,5	1,25	0,5	2,5	1,5	2,2	1,44	2,0	1,39
0,9	1,45	0,9	50,5	5,5	10,4	2,85	10	2,56
0,95	1,475	0,95	200,5	10,5	22,0	3,52	20	3,15
1,0	6	10	-	-	-	-	-	-

6.3.5 Verbesserung von Breut

Das double hashing läßt sich durch folgendes Verfahren noch weiter beschleunigen:

Der Schlüssel k soll eingefügt werden. Die Stelle j = k_1(k) ist schon mit Schlüssel k' belegt.
Definiere jetzt:
j_1 = j + h_2(k) = nächster Platz für k, j_2 = j+h_2(k') = nächster Platz für k'

Gehe dann wie folgt vor:

(1) j_1 ist frei: k dort abspeichern

(2) j_1 ist nicht frei und j_2 ist frei: k' nach j_2 kopieren und k in j speichern

(3) j_1 und j_2 nicht frei: mit normalem double hashing fortfahren

Laufzeit:

EL: $\approx \frac{1}{1-\alpha}$ (wie bei uniform hashing)

ER: $\approx 1+\frac{\alpha}{2}+\frac{\alpha^3}{4}+\frac{\alpha^4}{15}-\frac{\alpha^5}{18}\ldots$

Der Zugriff auf die zuletzt eingefügten Elemente wird beschleunigt zulasten der Zugriffszeit auf die älteren Elemente.

7 Graphen

Viele Probleme in der Informatik und der Mathematik lassen sich mit „gerichteten“ oder „ungerichteten“ Graphen darstellen.

7.1 Eigenschaften, Darstellung und Speicherung von Graphen

7.1.1 Definitionen

Ein Graph G besteht aus zwei Mengen V und E:

- $V = \{v_1, v_2, \ldots v_n\}$ Eine Menge von „Knoten“
- $E = \{ (v_i,v_j), (v_k,v_l), \ldots.(v_x,v_y) \}$; $i, j, k, l, x, y \in V$. Eine Menge von „Kanten“. Eine Kante ist eine Verbindung von zwei Knoten aus V.

Mit |E| bzw. |V| (Betrag V bzw. E) wird die Anzahl der Kanten bzw. Knoten der Menge bezeichnet.

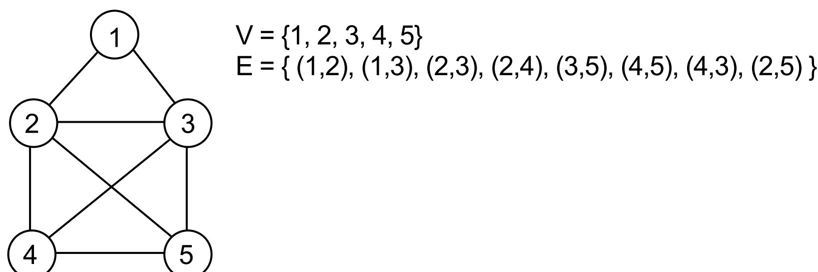
Der **Grad** eines Knotens ist die Anzahl der von ihm ausgehenden Kanten und somit die Anzahl der Nachbarknoten. Schreibweise: grad(i)

Ein **v_1-v_k-Pfad** in G = (V,E) ist eine Folge von Knoten $v_1 \ldots v_k$, die verbunden sind. z.B. $(v_1,v_2), (v_2,v_3), \ldots (v_{k-1},v_k)$

Es gibt zwei Graphentypen:

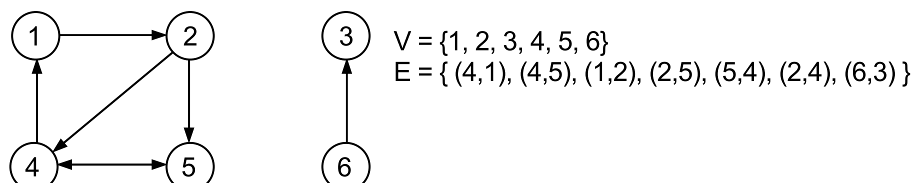
- Gerichtete Graphen: die Kanten haben eine Richtung: $(v_y,v_x) \neq (v_x,v_y)$
- Ungerichtete Graphen: Reihenfolge der Knoten irrelevant: $(v_y,v_x) = (v_x,v_y)$.

Beispiel eines ungerichteten Graphen:



V = {1, 2, 3, 4, 5}
E = { (1,2), (1,3), (2,3), (2,4), (3,5), (4,5), (4,3), (2,5) }

Beispiel eines gerichteten Graphen:



V = {1, 2, 3, 4, 5, 6}
E = { (4,1), (4,5), (1,2), (2,5), (5,4), (2,4), (6,3) }

Ein Graph ist **zusammenhängend**, wenn je zwei beliebige Knoten $u,v \in V$ durch einen u-v-Pfad verbunden werden können.

Ein Graph G‘ = (V‘,E‘) mit $E' \subset E$ und $V' \subset V$ heißt **Untergraph** von G.

Die **Zusammenhangskomponente** eines Graphen G ist der größte zusammenhängende Untergraph $G' \subset G$.

Ein Graph mit n Knoten heißt **dicht** oder **dick**, wenn er $\approx O(n^2)$ Kanten hat und licht oder dünn, wenn er $O(n)$ Kanten hat.

7.1.2 Speicherung von Graphen

Die beste Form für die Spiecherung eines Graphen hängt ab von den Operationen, die auf den Graphen ausgeführt werden sollen und von der Anzahl der vorhandenen Kanten (dicht oder licht).

Adjanzenz-Matritzen

G = (V,E) sei ein Graph mit n Knoten. Die zugehörige Adjazenzmatrix hat dann folgende Form:

$$A = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix}$$

Sie entspricht einem zweifach indizierten Array mit folgender Knonvention:

gerichtete Graphen: $a_{ij} = \begin{cases} 1, \text{falls } (i,j) \in E \\ 0, \text{sonst} \end{cases}$

ungerichtete Graphen: $a_{ij} = \begin{cases} 1, \text{falls } (i,j) \text{ oder } (j,i) \in E \\ 0, \text{sonst} \end{cases}$

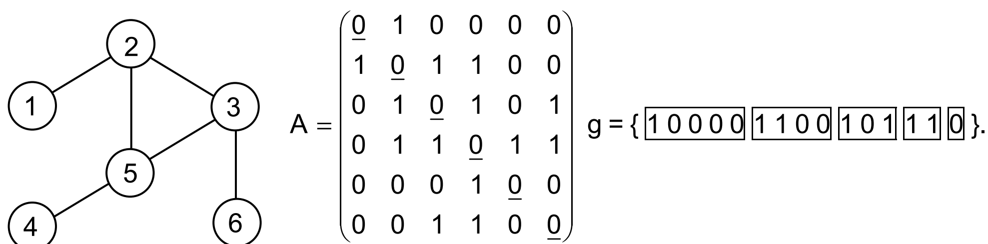
Vorteile:

- Suche nach einer Kante hat eine Laufzeit von $O(1)$
- Zum Darstellen einer Kante wird nur ein Bit benötigt.

Nachteile:

- Der Speicheraufwand ist $O(n^2)$.
- Speicherverbrauch hängt nicht von der wirklich vorhandenen Anzahl der Kanten ab.

Ist G ungerichtet, so ist $a_{kj} = a_{jk}$, d.h. die Matrix ist symmetrisch. Dann läßt sich Speicher einsparen, indem man nur die Hälfte der Matrix (alles rechts oben der Diagonale) speichert. Man legt die Daten dann in einem Vektor ab: g: g[0]...g[l].



Dieses Verfahren halbiert den Speicheraufwand, er ist aber weiterhin $O(n^2)$. Um auf den Vektor zuzugreifen, benötigt man aber einen zusätzlichen Indexvektor I(1...n–1), der die Anfangsadresse der Restzeilen berücksichtigt.
Er wird auf folgende Weise eingesetzt: A[i,j] = g[index(i)+j]. Im obigen Besispiel ist I = (-2, 2, 5, 7, 8). Man berechnet den Indexvektor auf folgende Weise:

- Finde den ersten Indexwert: I[1] = x

- Berechne alle folgenden Indexwerte nach der Rekursionsformel: **I**[k] = **I**[k–1] + (n–k)

Durchlaufen und bearbeiten aller Nachbarn:

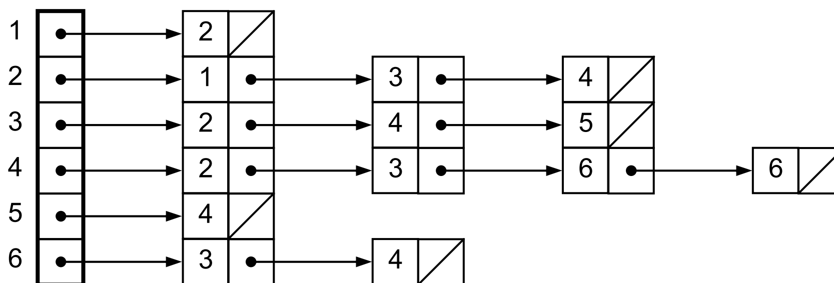
```
neighbour(G,i) {
for ( j=1 ; j≤i-1 ; j++)
{
     if ( g[index[j]+i] == 1 ) {…};
}
for ( j=i+1 ; j≤n ; j++)
{
     if ( g[index[i]+j] == 1 ) {…};
} }
```

Aufwand:
O(n) Operationen, unabhängig von grad(i).

Adjazenz-Listen

Für dünne Graphen ist der Speicheraufwand der Adjazenzmatrix inakzeptabel. Eine Abhilfe kann man schaffen, indem man für jeden Knoten eine verkettete Liste aller Nachbarknoten anlegt, die sog. Adjazenzliste.

Für das obige Beispiel hätten die Listen folgende Form:



Einfügen und Löschen von Kanten hat hier einen Aufwand von O(1). Der Speicheraufwand ist proportional zu der Anzahl der Kanten. Der minimale Speicherverbrauch ist |E|.

Wird keine dynamische Verwaltung des Graphen benötigt, d.h. sollen zu einem gegebenen Graphen keine Kanten hinzugefügt oder entfernt werden, so reicht ein Vektor n (=neighbour) aus, um die Indizes der Nachbarknoten zu speichern. Auch hier ist wieder ein Indexvektor I nötig, da die Anzahl der Nachbarn eines Knotens nicht aus n entnommen werden können:

n = { 2 | 1 3 4 | 2 4 6 | 2 3 5 6 | 4 | 3 4 }
I = { 0 1 4 7 11 12 14 }

Durchlaufen und bearbeiten aller Nachbarn:

```
neighbour2(G,i) {
for ( i=I[i] ; i≤I[i+1]-1 ; i++ )
{
     js = n[j];
     {…};
} }
```

Laufzeit:
O(grad(i)), d.h. bestmögliche Laufzeit, da jeder Nachbar ja mindestens einmal bearbeitet werden muß.

Suche nach einer Kante:

```
search(s,t) {
found = FALSE;
l = I[i]; /*oder I[j], falls grad(j)<grad(i)*/
while ( found ≠ TRUE && l < I[i+1] )
{
     if ( n[l]==j ) found = TRUE;
     else  l++;
} }
```

Aufwand:
Die Suche nach einer bestimmten Kante (s,t) kostet allerdings mehr als bei der Adjazenzmatrix: $O(\min(grad(s), grad(t)))$

7.2 Breadth-first search, „Breitensuche“

7.2.1 Hintergrund

Viele Aufgaben im Zusammenhang mit Graphen verlangen es, daß alle Knoten eines Graphen systematisch bearbeitet werden, z.B. um alle Daten der Knoten auszugeben oder um einen bestimmten Knoten zu finden.

Gegeben sei ein Graph G = (V,E) und ein beliebiger Startknoten $s \in V$ (Wurzel)

Mit der BFS werden alle Knoten des Graphen ausgehend von s durchlaufen. Dabei werden erst alle Knoten in der Entfernung k entdeckt, bevor Knoten in der Entfernung k+1 entdeckt werden.

7.2.2 Vorgehen

Idee: Färbe alle Knoten ein, abhängig vom Stand ihrer Bearbeitung:

- color(v) = weiß: noch nicht bearbeitet
- color(v) = grau: schon bearbeitet, aber es gibt noch weiße Nachbarn
- color(v) = schwarz: bearbeitet und alle nachbarn sind grau oder schwarz

Algorithmus:

(1) Färbe alle Knoten weiß.

(2) Färbe den Startknoten s grau und stecke s in eine queue. (s wird Wurzel des Baumes)

(3) Gehe Adjazenzliste von v durch, färbe weiße Nachbarn grau, stecke sie in queue und füge sie in den Baum ein

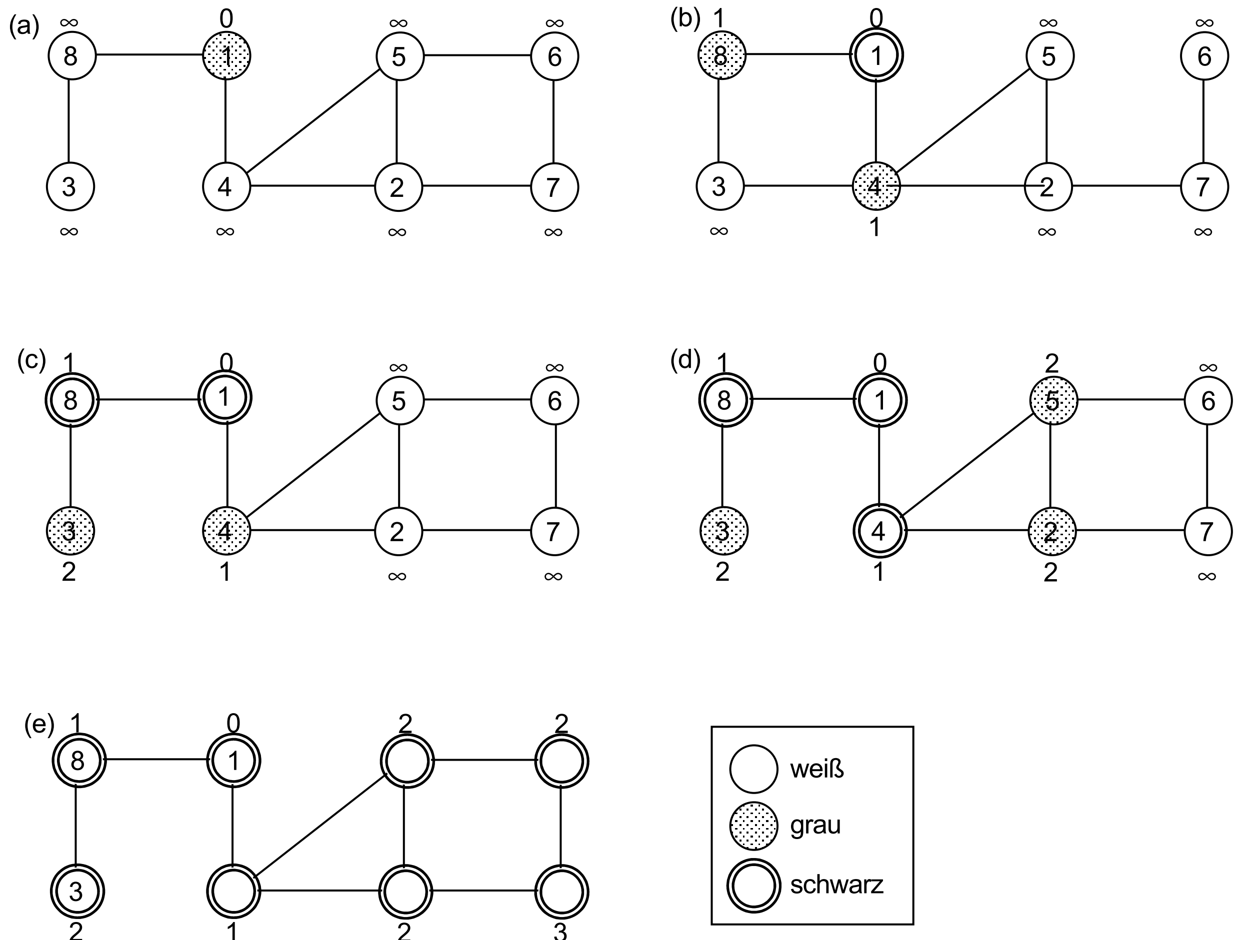
(4) Färbe s schwarz.

(5) Hole den nächsten Knoten aus der queue und fahre bei (3) fort.

Es werden also folgende zusätzliche Daten benötigt:

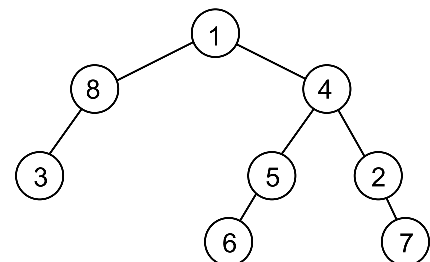
- color(v): weiß, grau, schwarz
- parent(v): Adresse des Vorgängers, NULL falls v Wurzel oder weiß
- dist(v): Abstand s von v
- Q: eine FIFO-Schlange

Beispiel



(e) ist das letzte Stadium der Suche.

Der BFS-Baum



Das Resultat von BFS läßt sich in einem Baum darstellen. (s. rechts) Der BFS-Baum stellt jeweils die kürzeste Verbindung zwischen s und einem beliebigen Knoten v aus dem Graphen her! Für jeden Knoten v aus dem Baum können wir also einen s-v-Pfad der Länge dist(v) ausschreiben.
Da der Baum (natürlich) kreisfrei ist und die minimale Anzahl an Kanten enthält, kann man mit BFS also auch einen Graphen kreisschlüsse eliminieren. (s. unten)

Aufwand

- Initialisierung: $O(|V|)$
- Jeder Knoten wird nur einmal grau (kommt in Q), dehalb Aufwand für enqeue, dequeue: $O(|E|)$
- Die Adjazenz-Liste für jedemn Knoten wird genau einmal durchgegangen, alle zusammen haben die Länge $O(|E|)$.

⇒ Gesamtzaufwand: $O(|V|) + O(|E|)$, d.h. die Laufzeit ist linear in der Darstellung des Graphen.

7.2.3 Finden der kürzesten Verbindung von zwei Knoten

Da BFS alle „nahen“ Knoten vor den „fernen“ bearbeitet, eignet sie sich auch, um den kürzesten Abstand zweier Knoten zu berechnen.
Zur Berechnung von Knoten s zu Knoten v zu berechnen, ruft man BFS auf den Knoten s auf, läßt einen BFS-Baum erstellen und ruft dann die Routine print_path auf den Knoten v auf.

Code von print_path

```
print_path(G,s,v) {
if ( v==s ) { print(s); }
else
     {
     if ( parent(v) == NULL ) { print(„Kein s-v-Pfad!“); }
     else
          {
               print_path(G,s,parent(v));
               print(v);
          }
     }
}
```

7.2.4 Satz

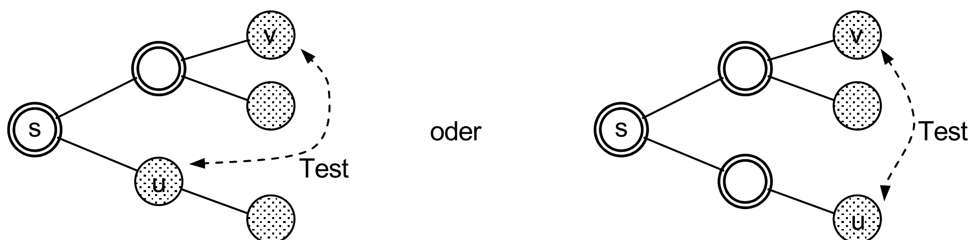
Der von print_path berechnete Pfad ist der kürzeste zwischen s und v, aber nicht unbedingt eindeutig.

Der zugehörige Beweis ist sehr umfangreich und würde den Umfang dieses Skriptes sprengen. Interessierte seinen auf das Buch von Cormen verwiesen.

Bemerkung:
Bei mehreren Wegen gleicher Länge hängt das Ergebnis von der Reihenfolge der Knoten in den Adjazenzlisten ab.
Ist G nicht zusammenhängend, so wird nur die Zusammenhangskomponente von s durchsucht. Um ganz G zu durchsuchen, muß man die Suche erneut bei einem Knoten starten, der noch nicht bearbeitet wurde.

7.2.5 Entdecken eines geschlossenen Kreises

Eine weitere nützliche Eigenschaft von BFS ist es, kreisschlüsse im Graühen entdeck zu können.
Falls G nicht kreisfrei ist, findet man ausgehend von u eine Kante u-v gefunden, für die v nicht mehr weis ist, d.h. der Knoten v ist schon früher erreicht worden. ⇒ hier existiert ein Kreisschluß



BFS erlaubt also:

- die Bestimmung aller Komponenten von G.
- die Bestimmung der kürzesten Wege zwischen zwei gegebenen Knoten.
- eine Überprüfung der Kreisfreiheit des Graphen.

7.3 Deep first search, „Tiefensuche“

7.3.1 Hintergrund

Bei der BFS werden alle Knoten gleicher Tiefe nacheinander aus der queue abgearbeitet.

DFS geht anders vor: sobald ein Knoten v von u aus grau gefärbt wird, wird die Adjazenzliste von u nicht weiter bearbeitet, es wird sofort v weiterbehandelt. Die Adjazenztliste von u wird später fortgesetzt.

7.3.2 Vorgehen

Man kann den Algorithmus sowohl rekursiv wie auch sequenziell implementieren. Die rekursive Variante ist einsichtiger, aber wegen der bekannten Probleme bei tiefen Rekursionen für große Bäume unbrauchbar, die sequenzielle Variante ist etwas schwieriger zu verstehen, ist aber in jedem Fall in der Praxis einsetzbar und zeigt außerdem die Ähnlichkeit zur BFS.

Algorithmus der sequenziellen Variante

(1) Färbe alle Knoten weiß.

(2) Mache den Satrtknoten zum aktuellen Knoten und färbe ihn grau.

(3) Gehe alle Nachbarn des aktuellen Knotens durch, färbe sie grau und lege sie auf den Stack.

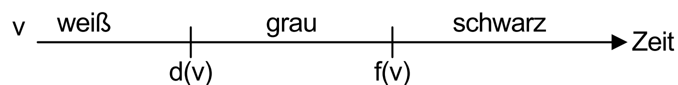
(4) Wurde kein weißer Nachbarknoten gefunden, färbe den aktuellen Knoten schwarz.

(5) Entnehme ein Element aus dem Stack (= zuletzt gefundener Knoten) und mache es zum aktuellen Knoten.

(6) Mache weiter mit Punkt (3)

Es werden jetzt zwei „Zeiten“ benötigt:
d(v): v wird entdeckt und grau gefärbt, f(v): alle Nachbarn von v sind grau, v wird schwarz

Es ist immer d(v) < f(v)



Code der rekursiven Variante

```
DFS(V) {
for ( alle Knoten v ∈ V )
{
     color(v) = WHITE;
     parent(v) = NULL;
}
time = 0;
for ( alle Knoten v ∈ V )
{
     if ( color(v) == WHITE ) DFS_visit(v);
}


DFS_visit(u) {
color(u) = GREY;
time++;
d(u) = time;
for ( alle v ∈ Adj(u) ) /* exploriere Kante (u,v) */
```

```
{
     if ( color(v) == WHITE )
     {
          parent(v) = u;
          DFS_visit(v);
     }
}
color(u) = BLACK;
time++;
f(u)=time;
}
```

7.3.3 Laufzeit

Für jeden Knoten u werden genau folgende Operationen durchgeführt:

- 3x färben
- f(u), d(u) einmal setzen
- einmal DFS_visit aufrufen

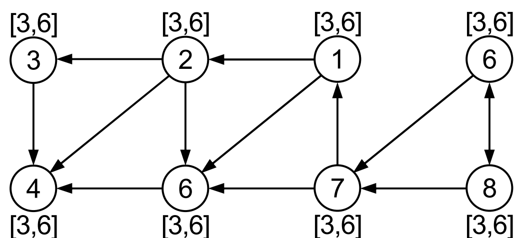
$\Rightarrow O(|V|)$

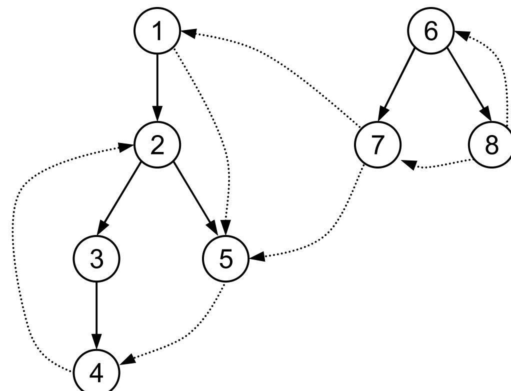
Jede Kante (u,v) wird genau einmal betrachtet: $O(|E|)$

$\Rightarrow$ Gesamtlaufzeit: $O(|V|) + O(|E|)$

Bei ungerichteten Graphen ist die Gesamtlaufzeit ebenfalls $O(|V|) + O(|E|)$, aber jede Kante wird 2x betrachtet.

7.3.4 Beispiel einer DFS

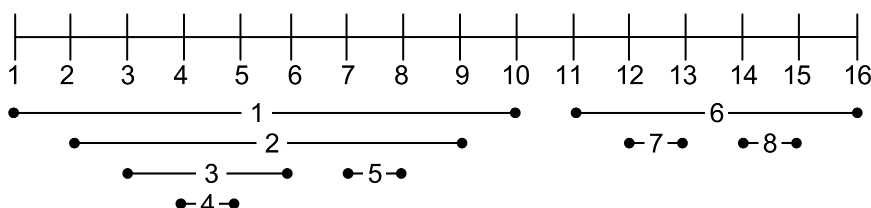




7.3.5 Der DFS-Wald

Die erzeugte Baumstruktur heißt „DFS-Wald". Sie ist nicht eindeutig (z.B. erzeugt der Staet bei Knoten 6 im obigen Beispiel nur einen Baum.) und enthält keine Kreise.

Folgendes Diagramm zeigt die Verteilung der [d,f]-Intervalle:



Man kann gut die rekursive Struktur des Algorithmus erkennen. (Schachtelungseigenschaft)

Satz

Ein Knoten v ist genau dann Nachfolger eines Knotens u im DFS_Wald, wenn zum Zeitpunkt d(u) (wenn u grau gefärbt wird) gibt es einen Pfad von u nach v, der nur aus weißen Knoten besteht.

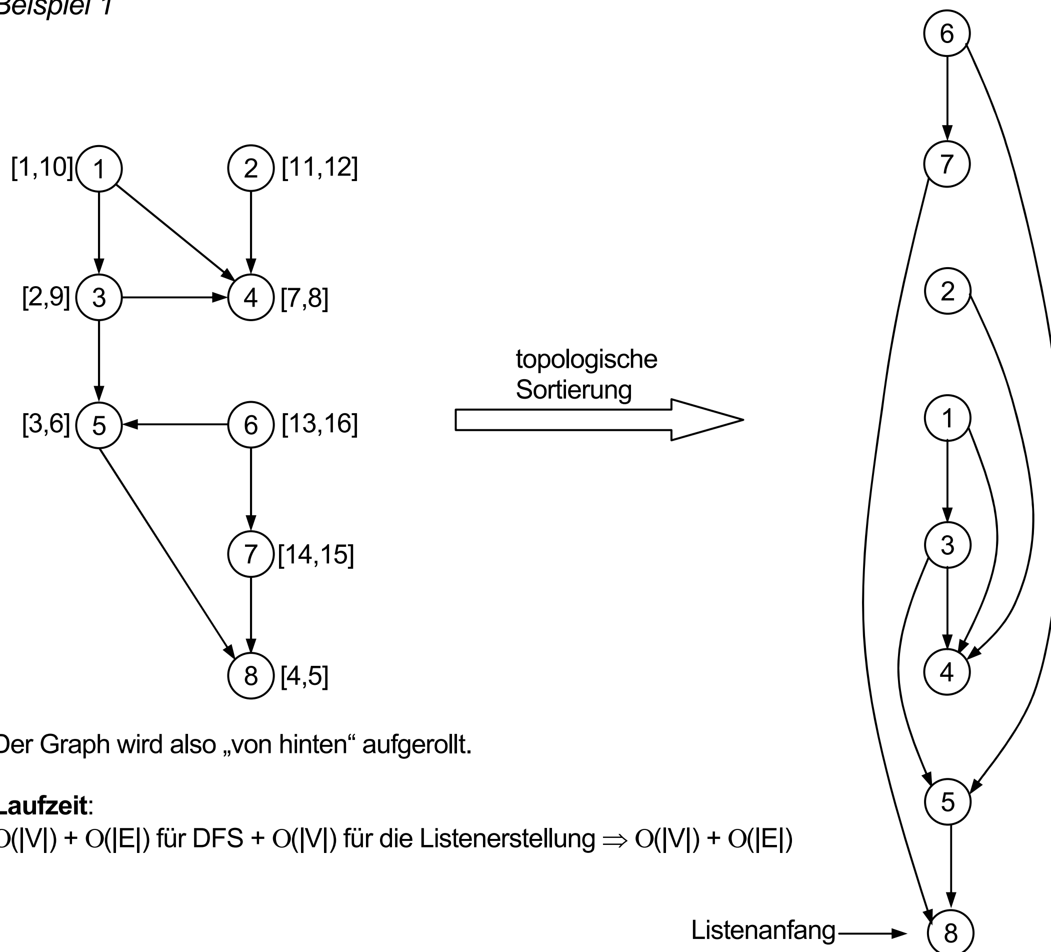
7.3.6 Topologische Sortierung von Graphen

Durch einen gerichtetenm kreisfreien Graphen lassen sich z.B. Arbeitsprozesse darstellen, die aufeinander aufbauen. Praktische Beispiele sind eine Fertigungsstraße im Automobilbau oder ein Ablaufplan für eine Baustelle. Eine Kante (x,y) interpretiert man dann so, daß x abgeschlossen sein muß, bevor y in Angriff genommen werden kann bzw. x ist Vorsussetzung für y.
In einem solchen Fall interessiert natürlich die Frage, in welcher Reihenfolge man die Arbeitsschritte ausführen soll, damit beider Ausführung eines Schritt bereits alle Vorsusstzungen erfüllt sind. Eine solche Darstelllung des Graphen nennnt man topologische Sortierung. Die topologische Sortierung muß nicht eindeutig sein und ist von der Reihenfolge der Knoten in der Adjazenzliste abhängig.

Die topologische Suche geht folgendermaßen vor:

- Intialisiere eine leere verkettete Liste L = ∅.
- Führe DFS(G) aus.
- Sobald f(v) für einen Knoten berechnet ist, füge v am Listenanfang ein.

Beispiel 1



Der Graph wird also „von hinten“ aufgerollt.

Laufzeit:
O(|V|) + O(|E|) für DFS + O(|V|) für die Listenerstellung ⇒ O(|V|) + O(|E|)

Beispiel 2 – Das morgendliche Anziehen

[11,16] Unterhose
Socken [17,18]
Uhr [9,10]
[12,15] Hose
Schuhe [13,14]
[6,7] Gürtel
Hemd [1,8]
Krawatte [2,5]
Jackett [3,4]
Ausgangsgraph

Schuhe
Hemd
Gürtel
Krawatte
Jackett
8
7
5
4
14
Hose
15
Unterhose
Uhr
16
Socken
topologisch sortiert – 2. Variante

Krawatte
Unterhose
Hose
Schuhe
Gürtel
12
10
9
8
6
Hemd
Jackett
13
5
Socken
Uhr
topologisch sortiert – 2. Variante

7.4 Minimal spanning trees

7.4.1 Einführung

Für viele Probleme sind die Eigenschaften der bisher behandelten Graphen nicht ausreichend und man muß sie mit weiteren Eingenschaften versehen:

Bei den gewichteten Graphen wird jeder Kante eine Zahl, daß sog. Gewicht, zugeordnet. Dieses Gewicht kann verschiedene Daten repräsentieren: Strecken (z.B. bei Wassernetzen), Zeiten (z.B. bei Bahnverbindungen), Wiederstände (bei Leiterplatten) oder ganz abstrakte Werte.
Ein ungewichteter Graph kann auch als gewichteter Graph aufgefaßt werden, bei dem jede Kante das Gewicht eins hat.

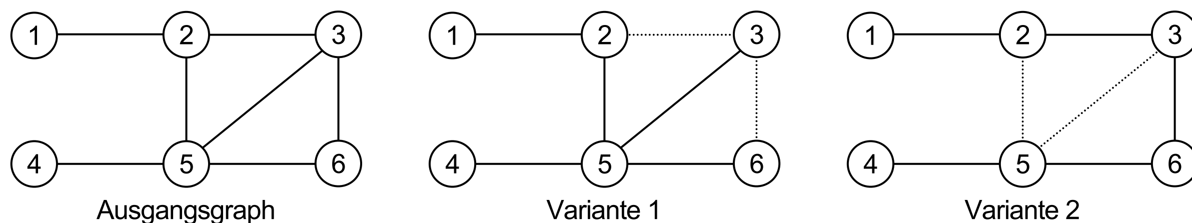
Eine häufiges Problem bei gewichteten Graphen ist die Suche nach dem Baum mit dem minimalen Gewicht, der alle Knoten enthält:

Definition

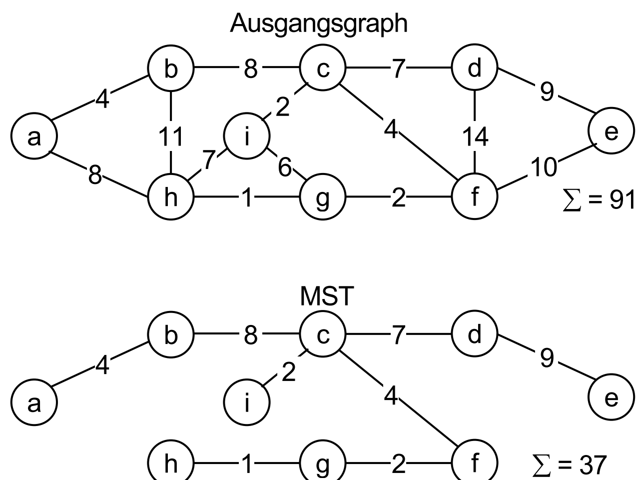
Ein minimal aufspannender Baum (MST) ist ein Untergraph G'(V,E') von G(V,E), der folgende Eingenschaften hat:

- zusammenhängend
- kreisfrei
- enthält alle Knoten $v \in V$

Beispiel 1



Beispiel 2



Wie man in diesem Beispiel erkennen kann, muß ein MST nicht immer eindeutig sein.
Man kann z.B. (b,c) durch (g,h) ersetzen und die minimale Summe $\sum = 37$ würde sich nicht verändern.

MST werden in vielen Bereichen zur Optimierung eingesetzt:

- Kommunikationsnetzwerke
- elektrische, elektronische Netzwerke
- Wassernetzwerke
- Straßennetz (z.B. Tourenplanung von Speditionen)
- Transportnetzwerke
- Erstellung von Leiterplatten (Verbinden n Pins mit n–1 Leitungen bei möglichst geringer Leitungslänge)

7.4.2 Erstellung eines MST

Vorgehen

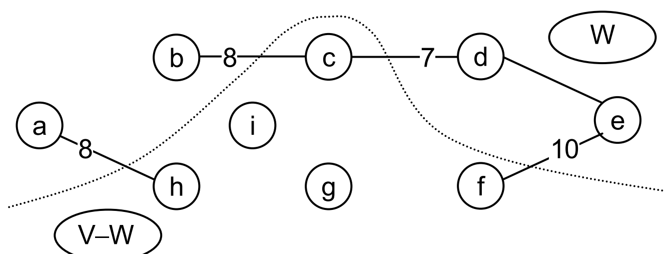
(1) Beginne zunächst mit A = leere Menge von Kanten
(2) Füge dann in jedem Schritt zu A ein Kante hinzu, so daß folgende Invariante gilt:
(3) „In jedem Schritt ist A Teilmenge eines MST."

Eine Kante aus E, die Eigenschaft (3) erhällt, heißt „sichere Kante". Wegen (3) muß eine solche Kante immer existieren, da immer ein MST existiert.

Finden einer sicheren Kante

Ein prinzipielles Verfahren zum Erkennen sicherer Kanten benutzt „Schnitte":

(1) Ein Schnitt S beruht auf der Zerlegung von V in zwei Mengen W, V-W:



$S = \{ (u,v) \mid u \in W; v \in V–W \}$

(2) Ein Schnitt S „respektiert" $A \subset E$, wenn er keine Kante von A enthält, d.h. keine A-Kante verbindet W und V–W. Die Teilbäume von A liegen dann also entweder in W oder in V–W.
(3) Die Kanten e = (u,v) in S mit minimiertem Gewicht heißen „Minimalkanten".

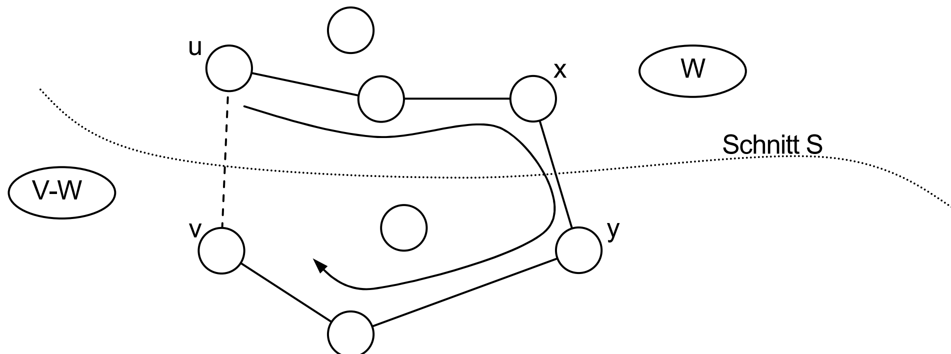
Satz

A sei Teilmenge von T, eines MST von G, S sei ein Schnitt, der A respektiert.

Dann ist jede Minimalkante $(u,v) \in S$ eine sichere Kante $\Leftrightarrow T_1 = A \cup (u,v) \in T$.

Beweis:
Fall 1: Die Minimalkante (u,v) ist aus T. Dann ist $T_1 = T \Rightarrow T_1$ ist MST.
Fall 2: Die Minimalkante (u,v) ist nicht aus T. Dann

- gibt es genau einen u-v-Pfad in T, da in einem Baum jeder Knoten über genau einen Pfad mit einem anderen Knoten verbunden ist.
- gibt es eine Kante (x,y) aus S die den Schnitt überquert.
- ist (x,y) nicht in A, da $(x,y) \in S$



w bezeichne das Gewicht einer Kante, bzw. das Gesamtgewicht einer Menge von Kanten.

Behauptung: Sei $T_1 = A \cup (u,v)$, $T_2 = A \cup (x,y)$. T_1 und T_2 sind MST. Dann gilt:

$$w(x,y) = w(u,v) \Leftrightarrow w(T_1) = w(T_2)$$

D.h. durch streichen von (x,y) und ersetzen durch (u,v) erzeugt auch einen minimalen Baum. (u,v) ist also in jedem Fall eine sichere Kante!

Annahme: Sei $w(u,v) < w(x,y) \Leftrightarrow w(T_1) < w(T_2)$.
Widerspruchsbeweis: T_2 kann kein MST sein, da sein Gewicht nicht minimal ist $\Rightarrow$ Annahme ist falsch.

Einfacher Algorithmus („greedy algorithm“)

Intuitiv wird man immer die Kante mit dem geringsten Gewicht wählen. Diese Vorgehensweise nennt man „greedy“ Algorithmus.

```
greedy_MST {
A = ();
while ( |A| < n-1 )
{
     suche sichere Kante (u,v) ∈ E;
     A = A ∪ (u,v);
     return A;
} }
```

Aufwand: A besteht aus einem Wald einzelner Bäume. In jedem Schritt werden 2 davon miteinander verbunden, d.h. es werden n–1 Schritte durchgeführt.

Literatur

Index