Rekurrenzen

$T(n) = T(n-1) + N$
beschreibt Laufzeitverhalten eines rekursiven Algorithmus

Bsp. Fibunacci
$F(n) = F(n-1) + F(n-2) \qquad N \geq 2$
$F(0) = F(1) = 1$
$\Rightarrow F(N)$ rekursive Aufrufe
$1,618^N \sim 2^N$

1) **Enfaltung**:
immer wieder einsetzen, bis man eine Regelmäßigkeit entdeckt
$T(N) = 2T(\frac{N}{2}) + N \qquad N \geq 2$
$T(1) = 1$

$$\begin{aligned} T(N) &= 2T\left(\frac{N}{2}\right) + N = \\ &= 2\left(2T\left(\frac{N}{4}\right) + \frac{N}{2}\right) + N = \\ &= 4T\left(\frac{N}{4}\right) + 2N = \\ &= 4\left(2T\left(\frac{N}{8}\right) + \frac{N}{4}\right) + 2N = \\ &= 8T\left(\frac{N}{8}\right) + 3N = \\ &= 2^i T\left(\frac{N}{2^i}\right) + iN \end{aligned}$$

$$\begin{aligned} N &= 2^i \\ i &= \mathrm{ld}\, N \end{aligned}$$

$i \,\mathrm{ld}\, N$ setzt man in T ein:

$$T(N) = 2^{\mathrm{ld}\, N} \underbrace{T\left(\frac{N}{2^{\mathrm{ld}\, N}}\right)}_{1} + \underbrace{\mathrm{ld}\, N * N}_{N * \mathrm{ld}\, N} =$$

$$\Rightarrow T(N) = N + N \mathrm{ld}\, N$$

$$\Rightarrow O(N + N \mathrm{ld}\, N)$$

2) **Raten und Beweisen**:

$T(N) = 2T(\frac{N}{2}) + N \qquad N \geq 2$

$T(1) = 1$

Guess: $T(N) = N + N\,\mathrm{ld}\,N$

Beweisen durch Induktion:

Basis: $T(1) = 1 + 1\,\mathrm{ld}\,1 = 1$

Schritt:

$$\begin{aligned} T(N) &= 2T\left(\frac{N}{2}\right) + N = \\ &= 2\left(\frac{N}{2} + \frac{N}{2}\,\mathrm{ld}\,\frac{N}{2}\right) + N = \\ &= N + N(\mathrm{ld}\,N - 1) + N = \\ &= N + N\,\mathrm{ld}\,N - N + N = \\ &= N + N\,\mathrm{ld}\,N \end{aligned}$$

$$\text{NR} : \ \mathrm{ld}\left(\frac{N}{2}\right) = \mathrm{ld}\,N - \mathrm{ld}\,2 = \mathrm{ld}\,N - 1$$

3) **Master Theorem**:

für Rekurrenzen der folgenden Art:

$T(n) = aT(\frac{n}{b}) + f(n)$

mit $a, b \geq 1$ und $f(n)$ asymptotisch positiv und $\frac{n}{b}$ kann entweder $\lceil \frac{n}{2} \rceil$ oder $\lfloor \frac{n}{2} \rfloor$ bedeuten

Bsp.

$T(N) = 9T(\frac{N}{3}) + N^{2.5}$

$a = 9$

$b = 3$

$f(n) = N^{2.5}$

$N^{\log_b a} = N^{\log_3 9} = N^2$

} immer machen

jetzt die Fälle betrachten:

$\varepsilon = 0.5$ (ist positiv, daher Fall 3) $\qquad (f(n) = N^{2+0.5} \rightarrow \varepsilon = 0.5)$

hier Fall 3:

$af(\frac{N}{b}) = 9(\frac{N}{3})^{2.5} = 9\frac{N^{2.5}}{3^{2.5}} = \frac{N^{2.5}}{3^{0.5}} = N^{2.5}\frac{1}{3^{\frac{1}{2}}} = f(n) * \frac{1}{3^{\frac{1}{2}}}$

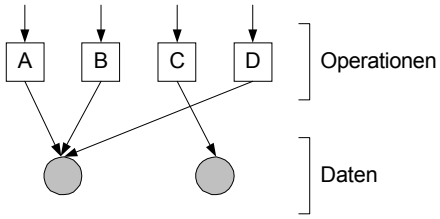
$c = \frac{1}{3^{\frac{1}{2}}}$

$\Rightarrow T(N) = O(N^{2.5})$

Abstrakte Datentypen

1) abstrakte Datenstrukturen (ADS)

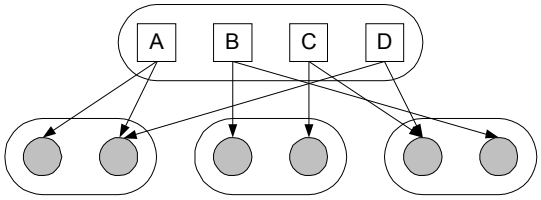
= Menge von Daten und den zugehörigen Operationen



¾ nur über diese Operationen auf die Daten Zugriff

¾ nur 1 Exemplar

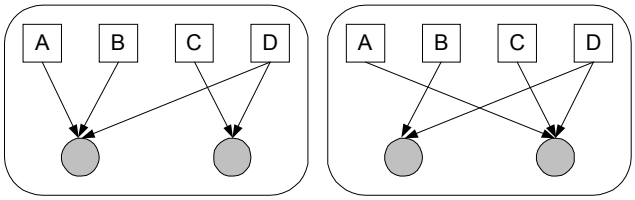
2) Abstrakter Datentyp



beliebig viele Exemplare

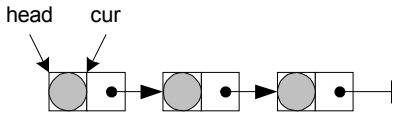
3) Klasse

Operationen sind bei den Daten eingebaut



wieder: beliebig viele Exemplare

zur Übung:



Einfügen:

```
public class LinkedList implements List{
    private Node head;
    public void add(Object o){
        Node cur;
        if (head==null) head=new Node(o);
        else {
            cur=head;
            while (cur.next!=null) cur=cur.next;
```

```
                cur.next = new Node(o);
            }
        }
    }
```

Dummy-Element: immer ein Element da, d.h. Sonderfälle, wenn Liste leer, fallen weg

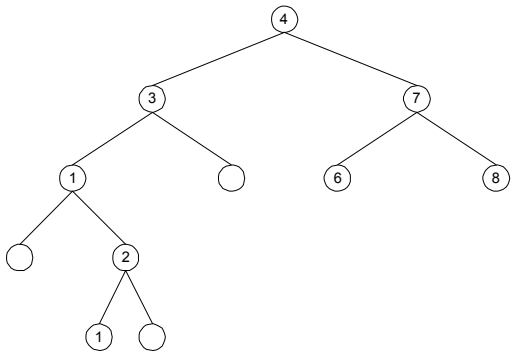
Löschen (mit Dummy-Element):
(erstes Vorkommen des Elements wird gelöscht)

```
public boolean remove(Object o){
    Node cur=head;
    while ((cur.next!=null) && (!cur.next.elem.equals(o))) cur=cur.next;
    if (cur.next!=null) return false;
    cur.next=cur.next.next;
    return true;
}
```

Bäume

Binärbäume

4 3 7 1 2 6 8 1
if kleiner 4 dann links else rechts



inOrder-Durchlauf:

1. linker Teilbaum
2. Knoteninhalt
3. rechter Teilbaum

} diese Folge bei jedem Knoten von vorne anfangen

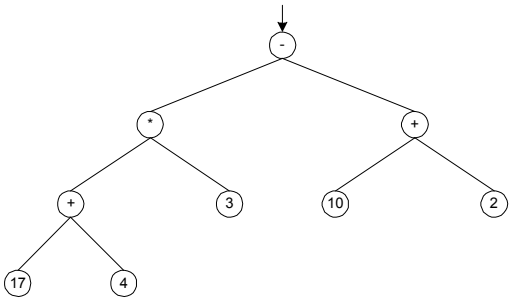
im Bsp: 1 1 2 3 4 6 7 8

preOrder-Durchlauf:

1. Knoteninhalt
2. linker Teilbaum
3. rechter Teilbaum

} diese Folge bei jedem Knoten von vorne anfangen

$x = (17 + 4) * 3 - (10 + 2)$



- * +17 4 3 +10 2

⇒ =Präfix-Notation

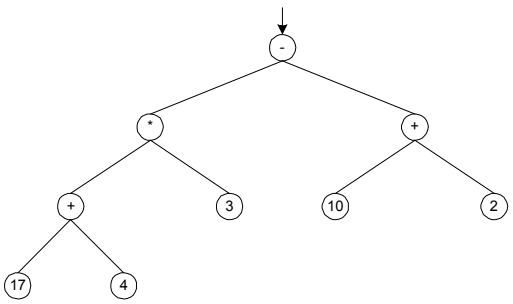
postOrder-Durchlauf:
Sinn:
Stack:
$x = (17 + 4) * 3 - (10 + 2)$

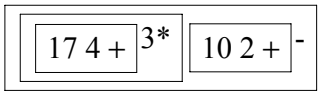
	Addition ausführen ⇒							
4								
17		21						
	Multiplikation ausführen ⇒							
3								
21		63						
2	Addition ausführen ⇒			Subtraktion ausführen ⇒				
10		12						
63		63						51

also

1. linker Teilbaum
2. rechter Teilbaum
3. Knoteninhalt

} diese Folge bei jedem Knoten von vorne anfangen





⇒ = Postfix-Notation

zur Übung:

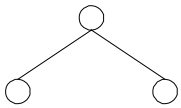
zu b)

```
public void inOrder(){
    myInOrder(root);
}

private void myInOrder(Node n){
    ...
    myInOrder(n.left);
    // Ausgabe von Knoteninhalt von n
    myInOrder(n.right);
}
```

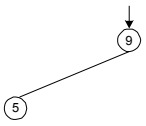
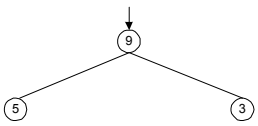
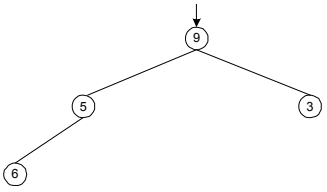
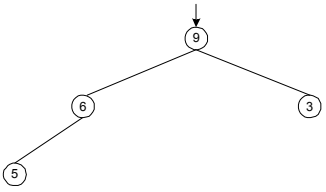
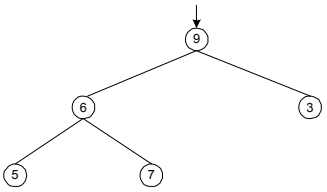
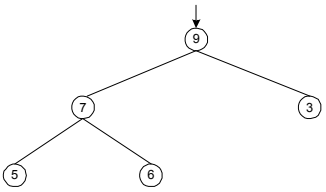
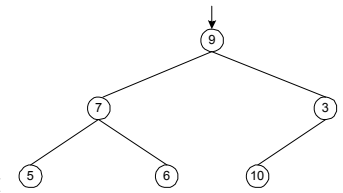
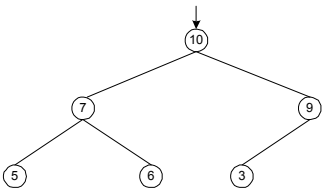
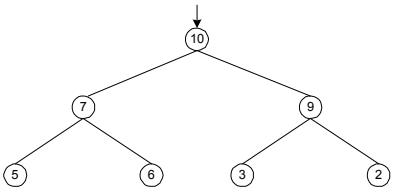
NumberOfNodes: wirklich durchzählen

Heap

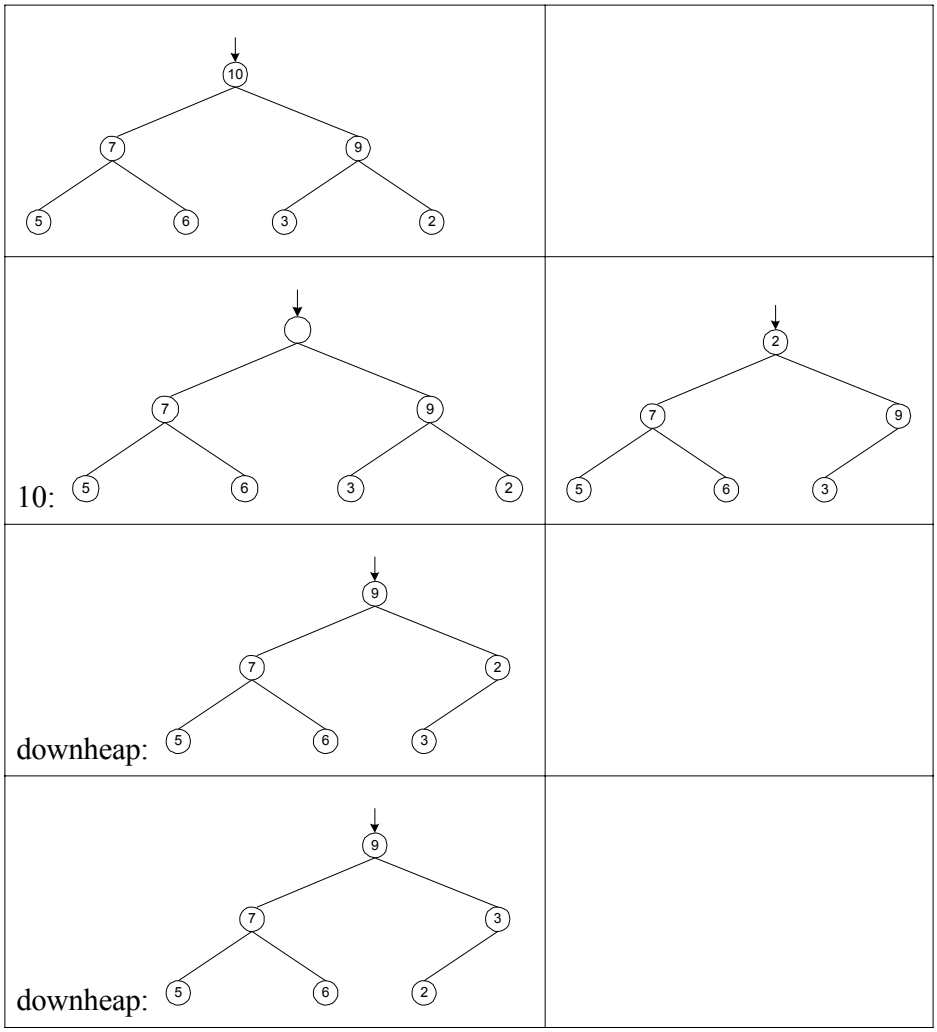


Key (Parent)≥Key(Children)
= Ordnungsrelation

Einfügen:
9 5 3 6 7 10 2

9: 	
5: 	
3: 	
6: 	upheap: 
7: 	upheap: 
10: 	2*upheap: 
2: 	

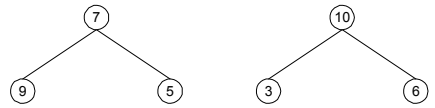
Löschen:



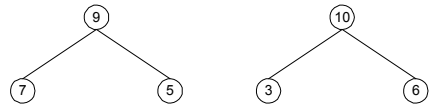
Einfügen mit Bottom-Up-Verfahren:

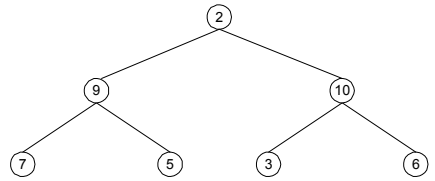
9	5	3	6	7	10	2

$\frac{n+1}{2}$ Blätter

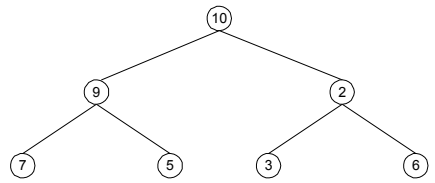


downheap in beiden Bäumen

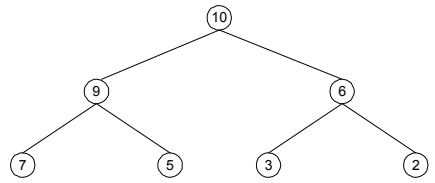




downheap



downheap



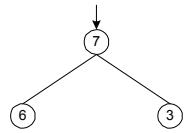
Beim Einfügen in eine Datenstruktur:
Baum ineinem Array abspeichern (zeilenweise)

⇒

9	8	7	6	5	4	3		

linker Sohn: $2i$ (i ist Index des Knotens)
rechter Sohn: $2i + 1$
Parent: $\frac{i}{2}$
Einfügen immer an der letzten Position

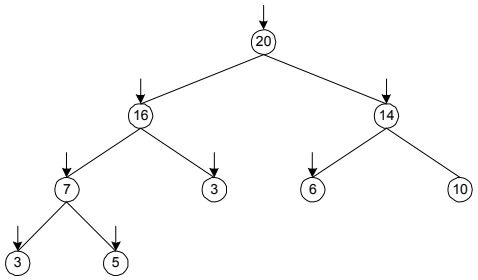
Suchen im Heap:



im Array:

7	6	3

Wegen der Ordnungsrelation kann man im Heap z.B. sofort sagen, daß 8 nicht enthalten ist.

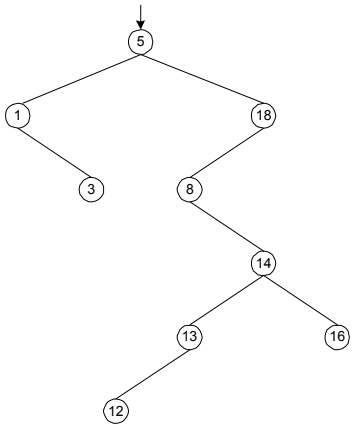


= wieder a rekursive gesucht

Change Priority:
wenn man oben im Beispiel z.B. 16er durch einen 25er ersetzt

Binärer Suchbaum

5 18 1 8 14 16 13 3 12
≥ ⇒ rechter Teilbaum
< ⇒ linker Teilbaum



Suchen im Binärbaum:
rekursiv:

```
Node find(int key, Node n){
    if (n==null) return null;
    if (key==n.key) return n;
    if (key<n.key) return find(key, n.left);
    return find(key,  n.right);
}
```

iterativ:

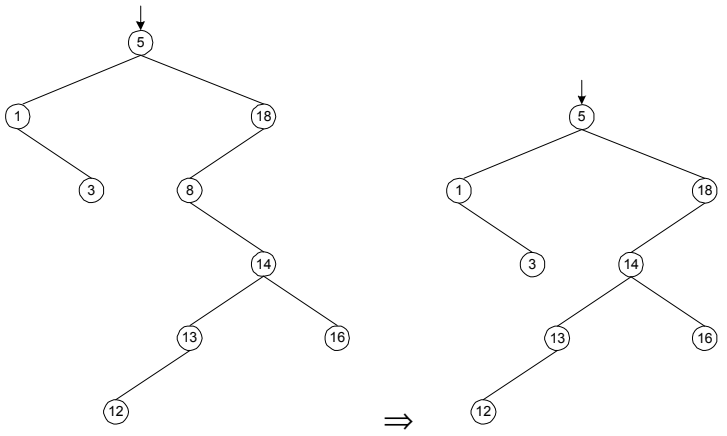
```
Node find(int key, Node n){
    while ((n!=null)&&(key!=n.key)){
        if (key<n.key) n=n.left;
```

```
            else n=n.right;
        }
    }
```

Löschen:

1.) Löschen eines Blattes (trivial)

2.) Löschen eines Knotens mit nur einem Sohn: z.B. löschen von Knoten 8



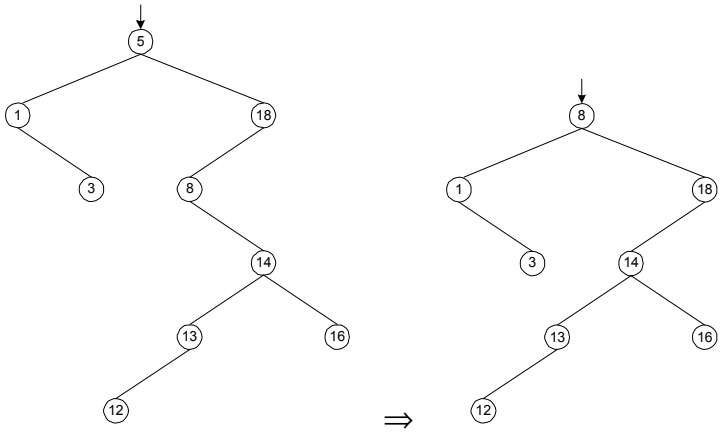
3.) Löschen eines Knotens mit 2 Söhnen:

a) einer der Söhne hat keine Söhne

b) rechter Sohn hat keinen linken Sohn oder linker Sohn hat keinen rechten Sohn ⇒ den jeweiligen Knoten nach oben ziehen

c) beide Söhne haben zwei Söhne oder wiedersprechen Punkt b) z.B. Löschen von Knoten 5 (im ursprünglichen Baum):

man bestimme den Nachfolgerknoten: dieser ist im rechten Teilbaum und innerhalb von dem, im linken Teilbaum → suchen, bis irgendwo der Links-Zeiger null ist. Im Bsp. ist 8 der Nachfolger



also:

1.) Suche Nachfolgeknoten

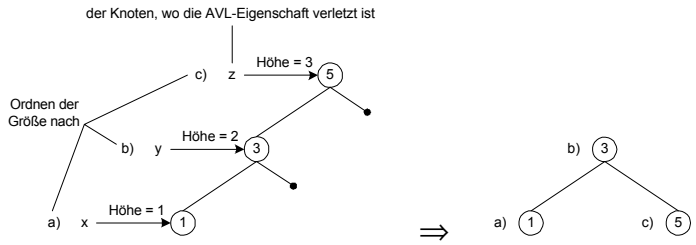
2.) Linkszeiger vom Vaterknoten (NF) zeigt auf rechten Sohn von NF

3.) Links- und Rechts-Zeiger von NF korrigieren

4.) Vaterknoten (Wurzel) muß auf NF zeigen

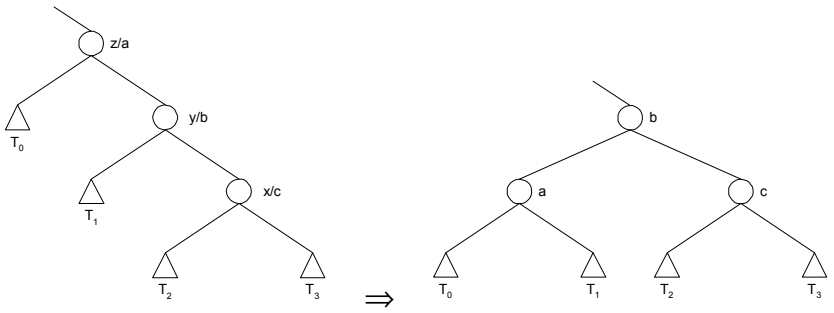
AVL-Bäume

= Balancierte Binärbäume
für jeden Knoten gilt: die Höhen seiner Teilbäume sind um maximal 1 verschieden

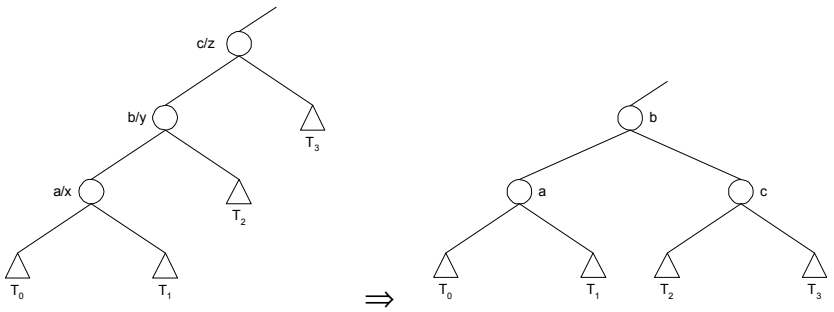


wenn noch Teilbäume daran hängen ⇒ 4 verschiedene Fälle:

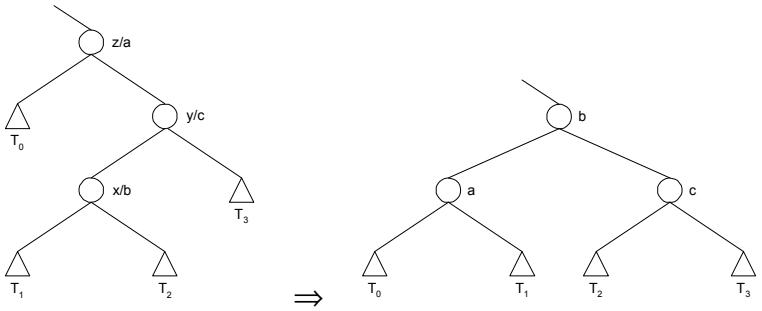
1)



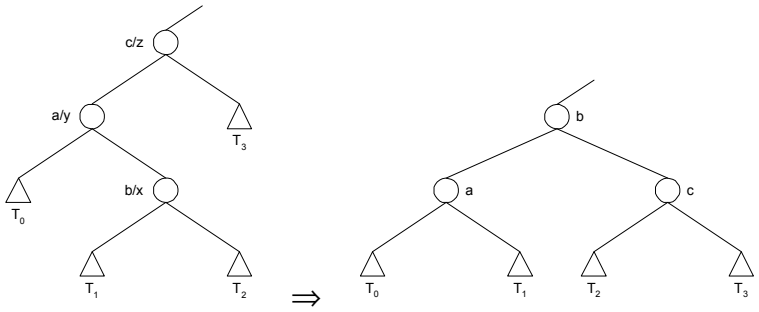
2)



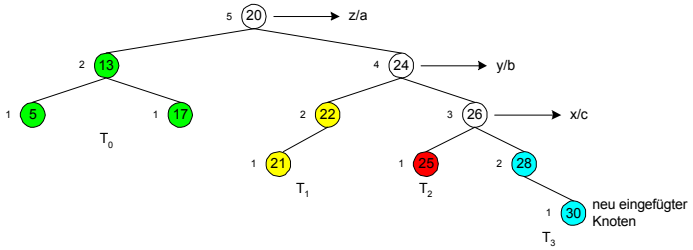
3)



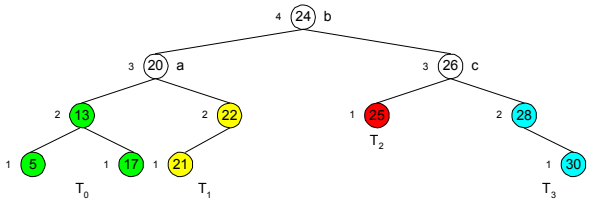
4)



Beispiel:

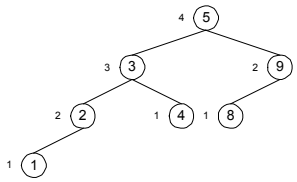


⇓

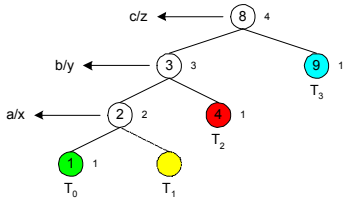


Löschen:
zuerst ganz normal löschen, dann Knoten z suchen, der unbalanciert ist
- y = Sohn von z mit größerer Höhe
- x = Sohn von y mit größerer Höhe

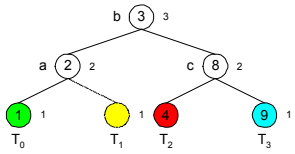
Beispiel:



5 löschen:



⇓



Hashing

Hashfunktion: meist Divisionsrestmethode

0	1	2	3	4	5	6	
21				11		27	

Elem mit Key 11 einsetzen: $11 \bmod 7 = 4$ 7 ... Listenlänge

z.B.

$27 \bmod 7 = 6$

$21 \bmod 7 = 0$

$18 \bmod 7 = 4$ ⇒ ist schon besetzt

$32 \bmod 7 = 4$

⇒ Lineare Kollisionsstrategie

man geht immer ein Feld weiter, bis man ein freies findet ⇒ dort einfügen.

Stößt man dabei auf das Ende der Liste, dann beginnt man wieder am Anfang zu schauen; d.h. immer $(\text{Feld}+1) \bmod 7$.

Wenn man was sucht: wenn man auf ein leeres Feld trifft, weiß man, daß das gesuchte Element nicht enthalten ist.

0	1	2	3	4	5	6	
21	32			11	18	27	

Löschen von 18:

0	1	2	3	4	5	6	
21	32			11		27	

Suche von 32 ⇒ stößt auf leeres Feld, obwohl 32 enthalten ist
Lösung: Flags setzen die sagen, ob ein Feld leer, voll oder ggelöscht wurde

0	1	2	3	4	5	6	
E	E	E	E	E	E	E	

E ... EMPTY

0	1	2	3	4	5	6	
21	32			11	18	27	
O	O	E	E	O	O	O	

O ... OCCUPIED

0	1	2	3	4	5	6	
21	32			11		27	
O	O	E	E	O	D	O	

D ... DELETED

⇒ Beim Suchen schaut man dann auch den Inhalt der DELETED-Flags an
In einer Hash-Table können keine zwei Einträge mit gleichem Schlüssel sein.

Quadratische Kollisionsstrategie
d.h. bei Kollisionen geht man in quadratischen Abständen nach rechts bzw. nach links

0	1	2	3	4	5	6	
				x			

also: $+1^2, -1^2, +2^2, -2^2, \ldots$

am Bsp.

0	1	2	3	4	5	6	
21			32	11	18	27	

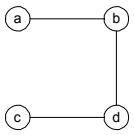
18 einfügen: $+1^2$
32 einfügen: $+1^2$ besetzt → -1^2

Größe einer Hashliste sollte eine Primzahl sein. Hashliste sollte zu max. 80% ausgelastet werden

Graphen

G=(V,E)
V ... Menge der Knoten (Vertex)
E ... Menge der Kanten (Edge)

V={a, b, c, d}
E={[a,b], [b,d], [c,d]}



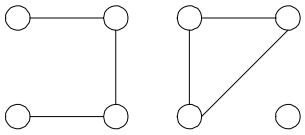
Adjazenz: 2 Knoten durch eine Kante verbunden

Grad: Anzahl der Kanten, die von diesem Knoten ausgehen

Pfad: Folge von adjazenten Knoten

Verbundener Graph: kein Knoten im Graph, der nicht ereichbar ist

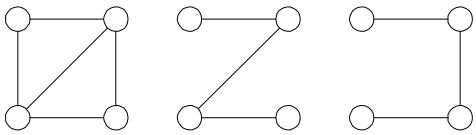
Komponente:
maximal verbundener Subgraph



Baum: verbundener Graph ohne Zyklen

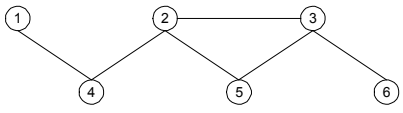
Wald: Menge von Bäumen (d.h. jede Komponente ist ein Baum)

Spannender Baum: (ST, spanning tree)



kein ST STs
d.h. alle Knoten erreichbar, keine Kante redundant
also:
ST ist ein Baum
ST enthält alle Knoten

Adjazenzmatrix:



	1	2	3	4	5	6
1	0	0	0	1	0	0
2	0	0	1	1	1	0
3	0	1	0	0	1	1
4	1	1	0	1	0	0
5	0	1	1	0	0	0
6	0	0	1	0	0	0

Depth-First-Search Algorithmus(DFS)

Hilfsarray:

1	2	3	4	5	6		
x							

1 markieren, prüfen: **4** (weil dieser als einziger von 1 aus erreichbar ist)

1	2	3	4	5	6		
x			x				

4 markieren, prüfen: 1,**2**

1	2	3	4	5	6		
x	x		x				

2 markieren, prüfen: **3**, 4, 5

1	2	3	4	5	6		
x	x	x	x				

3 markieren, prüfen: 2, **5**, 6

1	2	3	4	5	6		
x	x	x	x	x			

5 markieren, prüfen: 2, 3

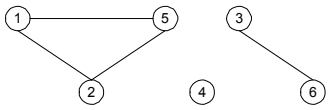
1	2	3	4	5	6		
x	x	x	x	x	x		

6 markieren, prüfen: 3

also: Rekursion immer zuerst in die Tiefe (Depth-First)

Statt eines Hilfsarrays könnte man auch die Hauptdiagonale der Matrix verwenden, da diese ohnehin nicht gebraucht wird

Bsp. eines nicht zusammenhängenden:



zuerst:

1	2	3	4	5	6		
x	x			x			

nrC=1 (nrC ... number of components)

dann DFS beginnend bei 3:

1	2	3	4	5	6		
x	x	x		x	x		

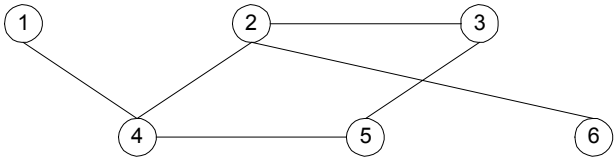
nrC=2

dann noch DFS beginnend bei 4:

1	2	3	4	5	6		
x	x	x	x	x	x		

nrC=3

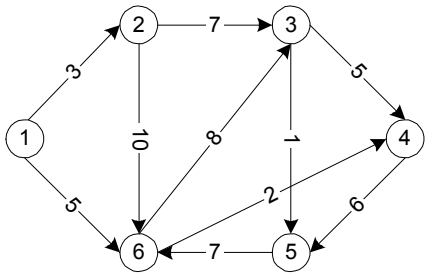
Breadth-First Search (Breitensuche)



	1	2	3	4	5	6
1	0	0	0	1	0	0
2	0	0	1	1	0	1
3	0	1	0	0	1	0
4	1	1	0	0	1	0
5	0	0	1	1	0	0
6	0	1	0	0	0	0

Besuchte Knoten	Queue
1	4
4	2, 5
2	5, 3, 6
5	3, 6
3	6
6	-

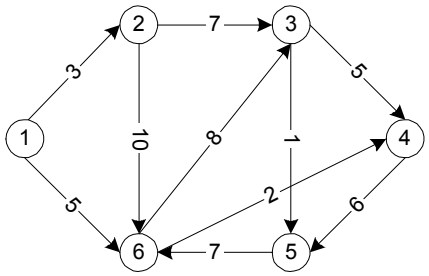
Minimale Wege (Dijkstra)



	1	2	3	4	5	6
1	0	3	∞	∞	∞	5
2	∞	0	7	∞	∞	10
3	∞	∞	0	5	1	∞
4	∞	∞	∞	0	6	∞
5	∞	∞	∞	∞	0	7
6	∞	∞	8	2	∞	0

BK	NK	S(1)	S(2)	S(3)	S(4)	S(5)	S(6)	min. Weg
{1}	{2,3,4,5,6}	(0)	3	∞	∞	∞	5	
{1,2}	{3,4,5,6}	(0)	(3)	3+7	∞	∞	5	5
{1,2,6}	{3,4,5}	(0)	(3)	3+7	5+2	∞	(5)	7
{1,2,6,4}	{3,5}	(0)	(3)	3+7	(5+2)	7+6	(5)	3+7
{1,2,6,4,3}	{5}	(0)	(3)	(3+7)	(5+2)	10+1	(5)	10+1
[1,2,6,4,3,5}	{}	0	3	3+7	5+2	10+1	5	

Kruskal (minimal spannender Baum)



ausgehend vom leeren Graphen; Kante mit geringstem Gewicht eintragen; dann immer mit dem nächst größten Gewicht; wenn eine Kante nicht paßt (wegen Zyklus) → auslassen