Datenstrukturen und Algorithmen

Prof. Dr. M. Jarke Lehrstuhl Informatik V
Frontalübung für Technische Redakteure
Thomas von der Maßen

Ziel der Veralstaltung:
- Aufarbeiten der Vorlesung
- Übungen mit neuen Beispielen
- Diskussion und Fragen

Übungsschein
- nicht zwingend erforderlich
- 60% der erzielbaren Übungespunkte
- Vorrechnen mind. einer Lösung in den Gruppen

--> Übungsklausur am Ende des Semesters
Informationen, Folien, Skript und Musterlösungen:
http://www-i5.informatik.rwth-aachen.de/lehrstuhl/lehre/DA01/index.html

Ziele der Vorlesung:
- (effiziente) Datenstrukturen
- Algorithmen für allg. Probleme
- Theoretische Kenntnisse
- Praktische Kenntnisse

14.05.01

Algorithmen und Komplexität

Die Komplexitätsordnung eines Algorithmus bestimmt maßgeblich die Laufzeit und den Platzbedarf der konkreten Implementierung.

Komplexitätsmaße:
- Anzahl der einfachen Operationen
- Anzahl der Variablen

Größe der Eingabe:
- Anzahl der Elemente
- Anzahl der Parameter

Die Vorlesung beschäftigt sich Hauptsächlich mit der Laufzeit. Der Platzbedarf wird vernachlässigt.

Zeitkomplexität:
Sei A ein Algorithmus. Für all Eingaben w ist $T_A(w)$ die Anzahl der Operationen des Algorithmus bei der Bearbeitung der Eingaben w.

Komplexität im schlechtesten Fall (worst case)
$T_A(w) = \max\{ T_A(w) \mid w \text{ ist die Eingabe der Größe (bzw. Länge) } n \}$

Bsp.: Sortieren von n Zahlen mit einem bestimmten Algorithmus.

Komplexitätsklassen

1	konstant	(elementare Befehle)
log n	logarithmisch	(binäre Suche)
n	linear	(lineare Suche)
n log n	überlinear	(eff. Sortierverfahren)
n^k	Polynom zum Grad k	
2^n	exponentiell	(Backtracking)
n!	Fakultät	(Zahl d. Permutationen)

Beachte:
Abschätzung von n!:
$n! \sim \sqrt{2\pi n} * (n/e)^n$ (Stirlingsche Formel)

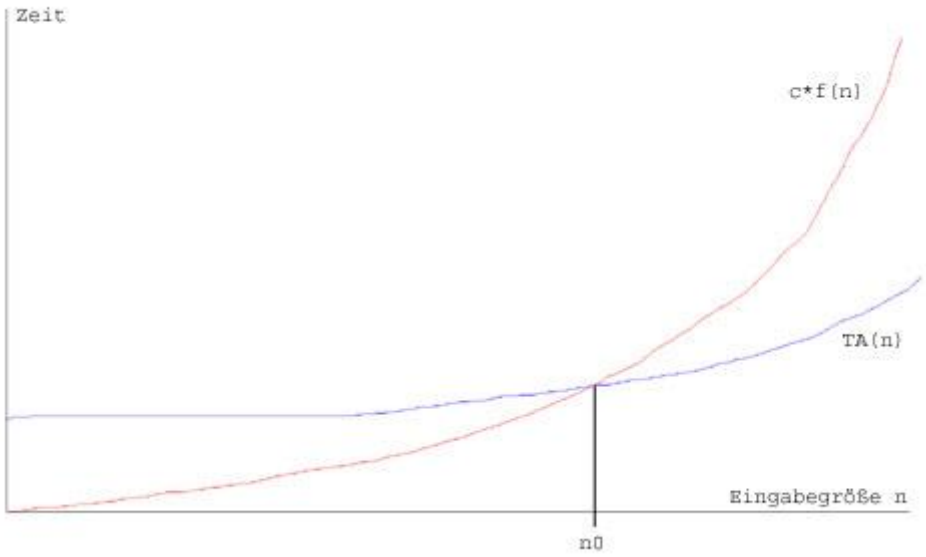
O-Notation:

Ausreichend: Asymptotisches Verhalten einer Funktion, welche die Laufzeit eines Programms beschreibt.

Definition:

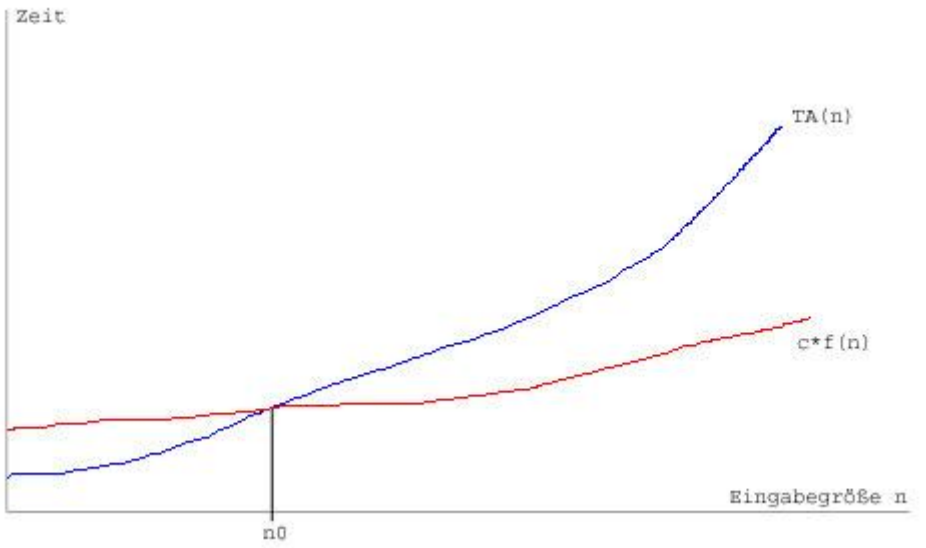
i) $T_A(n) \in O(f(n)) : \Leftrightarrow \exists\, c, n_o \in N, \forall\ n \geq n_o : T_A(n) \leq c * f(n)$

ii) (Im Skript: $O(f) := \{ g: N \rightarrow R^+ \mid \exists\ c, n_0 \geq 0 : g(n) \leq c * f(n) \ \forall\ n \geq n_o \}$)



=> f ist obere Schranke von $T_A(n)$

ii) $T_A(n) \in \Omega(f(n)) : \leftrightarrow \exists\, d, n_o \in N, \forall\ n \geq n_o : d * T_A(n) \geq f(n)$



=> f ist untere Schranke von $T_A(n)$

iii) $T_A(n) \in \Theta(f(n)) : \leftrightarrow\ T_A(n) \in O(f(n)) \wedge T_A(n) \in \Omega(f(n))$

=> f ist Wachstumsrate von $T_A(n)$

Rechenregeln:

f und g seien Funktionen mit $N \rightarrow R^+$, dann gilt:

1) $g(n) = \alpha * f(n) + \beta$ mit $\alpha, \beta \in R^+$ und $f \in \Omega(1) \Rightarrow g \in O(f)$
   konstante Faktoren und konstante Summanden können vernachlässigt werden.
2) $f + g \in O(\max\{f,g\}) = O(g)$, falls $f \in O(g)$
   $O(f)$, falls $g \in O(f)$
   Man betrachtet immer nur die Komplexität des schlechtesten Teils.
   Bsp.: Folgen von Anweisungen
   - asymptotisches Verhalten wird nur von dem Teil bestimmt, welcher den größten Aufwand hat.
   - Hintereinanderausführung von Anweisungen A und B mit
     $T_A \in O(f)$ und $T_B \in O(g)$
3) $a \in O(f) \wedge b \in O(g) \Rightarrow a * b \in O(f * g)$
   Bsp.: Schleifen
   - Laufzeit d. Schleifenkörpers mit Anzahl der Durchläufe multiplizieren.
4) Bedingte Anweisungen

```
IF b THEN            konstante Laufzeit
  A();                 g
ELSE
  B();                 f
END;
```

   Laufzeit: $O(1) + O(g+f)$ mit $T_A \in O(f) \wedge T_B \in O(g)$
   $\Rightarrow O(g+f)$

   Bsp.:
   1) Sei $f(n) = 6n + 4$
      $\Rightarrow 6n + 4 \leq c * n \ \forall \ n \geq n_0$
      mit $c = 7$ und $n_0 = 4$
      $\Rightarrow f(n) \in O(n)$

   2) Sei $g(n) = 4n^2 - 2$
      $\Rightarrow 4n^2 - 2 \leq c * n^2 \ \forall \ n \geq n_0$
      mit $c = 4$ und $n_0 = 1$

   3) Sei $g(n) = 7n^3 + 2n^2 - 8n + 2$
5) $\Rightarrow g(n) \in O(n^3)$

   4)
```
CONST n = 100
TYPE Feld = ARRAY [1..n] OF INTEGER;
VAR a: FELD;
(* Feldinitialisierung *)
FOR i := 1 TO n DO
  a[i] := 0;
END;
```

   $\Rightarrow T_A \in O(n)$

5)
```
FOR i:= 1 TO n DO
    IF i < 50 THEN
      a[i] := -1;
    ELSE
      a[i] := 1;
    END;
  END;
```
T_B | T_C | T_A

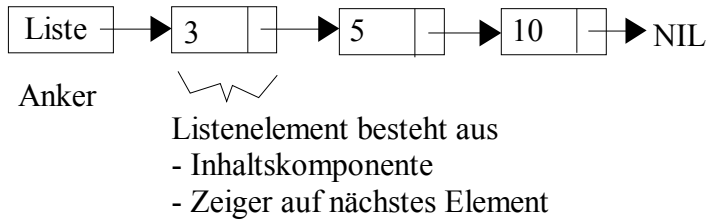
=> $T_A(n) \in O(n)$

6)
```
TYPE Feld = ARRAY [1..n, 1..m] OF INTEGER;
  VAR a : Feld;
  FOR i := 1 TO n DO
    FOR j := 1 TO m DO
      a[i,j] := 0;
    END;
  END;
```
=> $T_A(n) \in O(n^2)$

21.05.01

Dynamische Datenstrukturen

Lineare Listen
- Folge von Elementen
- Verkettung von Records mittels Zeiger



Deklaration in Modula3

```
TYPE Listenelement = REF RECORD
                           wert : INTEGER;
                           next: Listenelement;
                         END;
VAR liste : Listenelement;
```

Initialisierung:

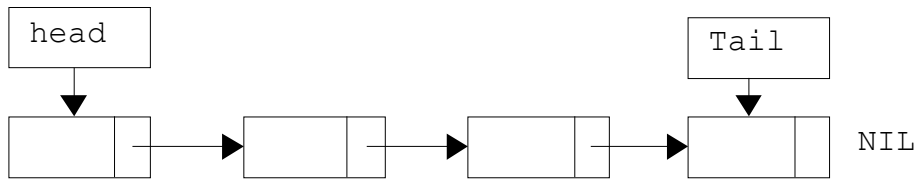
```
PROCEDURE Init(VAR l: Listenelement) =
BEGIN
  l := NIL;
END Init;
```

Anhängen eines Elements

```
PROCEDURE Append (zahl : INTEGER; VAR l : Listenelement)=
VAR hilf, vorgaenger, neu : Listenelement;
BEGIN
  neu := NEW(Listenelement);
  neu^.wert := zahl;
  neu^.next := NIL;
  IF l = NIL THEN (* leere Liste *)
    l := neu;
  ELSE
    hilf := l;
    vorgaenger := l;
    WHILE hilf # NIL
      vorgaenger := hilf;
      hilf := hilf^.next;
    END;
    vorgaenger^.next:= neu;
  END;
END Append;
```

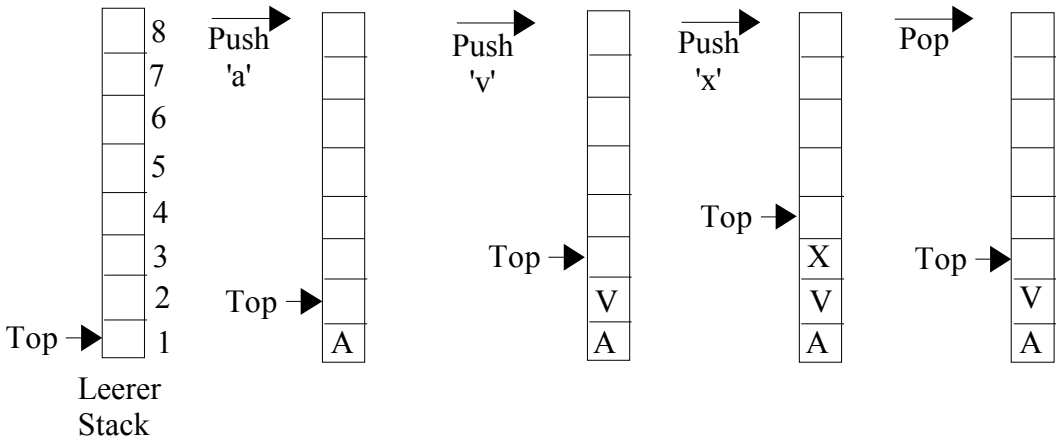
=> O(n)

Unterschiedliche Implementierung von Listen sind möglich. Beispiel: Verwaltung von Kopf- und Endzeigern.



Vorteil: Direkter Zugriff auf das letzte Element der Liste möglich! (Append ∈ O(1))

Keller (Stack)

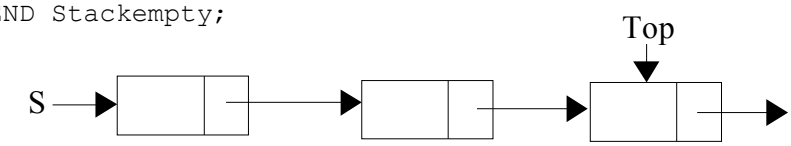


push: neues Element wird am oberen Ende eingefügt.
pop: obestes Element wird entfernt und zurückgeliefert.

- Arbeitet nach dem LIFO-Prinzip (Last In, First Out)
- Es kann nur auf das oberste Element zugegriffen werden
- es wird immer von oben „eingekellert“ (Wie bei einem Stapel Teller)

```
CONST MAX = 10;
TYPE Stack = ARRAY [0..Max-1] OF CHAR;
VAR s : Stack;

PROCEDURE Init()=
BEGIN
  top := 0;
END;

PROCEDURE Push(c : CHAR)=
BEGIN
  IF NOT Stackfull() THEN
    s[top] := c;
    INC(c);
  END;
END Push;
```

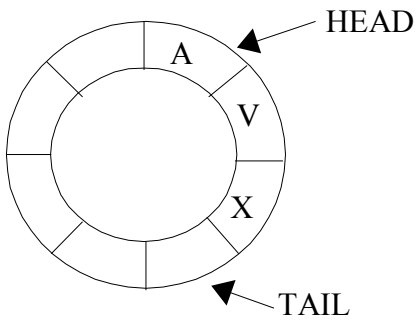
```
PROCEDURE Pop() : CHAR =
BEGIN
  IF NOT Stackempty() THEN
    DEC(c);
    RETURN s[top];
  END;
END Pop;

PROCEDURE Stackfull(): BOOLEAN =
BEGIN
  IF (top = MAX) THEN
    RETURN TRUE;
  ELSE
    RETURN FALSE;
  END;
END Stackfull;

PROCEDURE Stackempty() : BOOLEAN =
BEGIN
  IF (top = 0) THEN
    RETURN TRUE;
  ELSE
    RETURN FALSE;
  END;
END Stackempty;
```



Listenimplementierung:

push := „Einfügen vorne“
pop := „Löschen vorne“ + Zurückgeliefern des ersten Elements.
empty := Liste = NIL
full := FALSE (Liste wird nie voll)

Warteschlage (Queue)



put(a) -> Tail = 1
put(v) -> Tail = 2
put(x) -> Tail = 3
get() -> head = 2

put: neues Element wird am Ende angehängt
get: erstes Element wird entfernt und zurückgeliefern

- arbeitet nach dem FIFO-Prinzip (First In, First Out)
- es wird immer nur auf das erste Element zugegriffen
- es wird nur „hinten“ angehängt

Implementierung in Modula3

```
CONST Max = 10;
TYPE Queue = ARRAY [0..Max] OF CHAR;
VAR q : Queue;
    head, tail : CARDINAL;

PROCEDURE Init() =
BEGIN
  head := 0;
  tail := 0;
END Init;

PROCEDURE Put(c : CHAR) =
BEGIN
  IF NOT Full() THEN
    q[tail] := c;
    tail := (tail+1) MOD (MAX+1);
  END;
END Put;

PROCEDURE Get() : CHAR =
VAR c : CHAR;
BEGIN
  IF NOT Empty() THEN
    c := q[head];
    head := (head+1) MOD (MAX+1);
    RETURN c;
  END;
END Get;

PROCEDURE Empty() : BOOLEAN =
BEGIN
  IF head = tail THEN
    RETURN TRUE;
  ELSE
    RETURN FALSE;
  END;
END Empty;

PROCEDURE Full() : BOOLEAN =
BEGIN
  IF (head = tail + 1) OR (tail = MAX AND head = 1) THEN
    RETURN TRUE;
  ELSE
    RETURN FALSE;
  END;
END Full;
```

Problem: Es können maximal MAX Elemente gespeichert werden, obwohl das Feld aus MAX+1 Elementen besteht.

Listenimplementierung:

put := Append „Einfügen hinten“

get := „Löschen vorne“ + Zurückgeliefern des ersten Elements.

empty := Liste = NIL

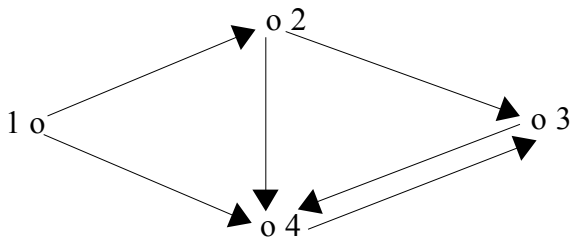
full := FALSE (Liste wird nie voll)

28.05.01

Bäume

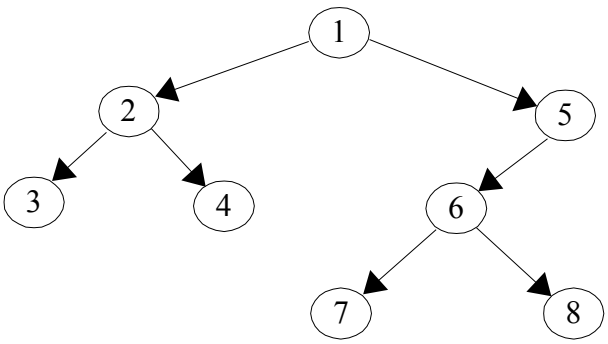
Graphen: G = (K, E) mit
- K endliche Menge von Knoten
- $E \subseteq K \times K$ (Menge von Knoten)
- Gerichtete Graph:
  - k_2 ist Nachfolger von $k_1 \leftrightarrow (k_1, k_2) \in E$
  - $(k_1, .. , k_{n+1})$ heßt gerichteter Knotenzug $\leftrightarrow (k_i, k_{i+1}) \in E$

Bsp.: G = (K, E) mi
K = {1, 2, 3, 4 }
E = {(1, 2), (2, 3), (2, 4), (1, 4), (3, 4), (4, 3)}



Baum: Zyklenfreier, gerichteter Graph, wobei jeder Knoten höchstens einen Vorgänger hat und bei dem jeder Knoten von bestimmten Knoten (Wurzel) erreichbar ist.

BspBaum.



Tiefe eines Knotens: Level(k) = 1 , falls k Wurzel
(level(vorgaenger(k) + 1) , sonst

Höhe eines Baums: max(level(k)) , $k \in K$

Durchlaufen eines Baums

1) Tiefensuche
   a) PreOrder
      - betrachte n
      - durchlaufe Teilbäume n_1, .. , n_k
      - KLR

      BspBaum.: 1 – 2 – 3 – 4 – 5 – 6 – 7 – 8

   b) InOrder
      - durchlaufe n_1
      - betrachte n
      - durchlaufe n_2,..., n_k
      - LKR

      BspBaum.: 3 – 2 – 4 – 1 – 7 – 6 – 8 – 5

   c) PostOrder
      - durchlaufe Teilbäume n_1,...,n_k
      - betrachte n
      - LRK

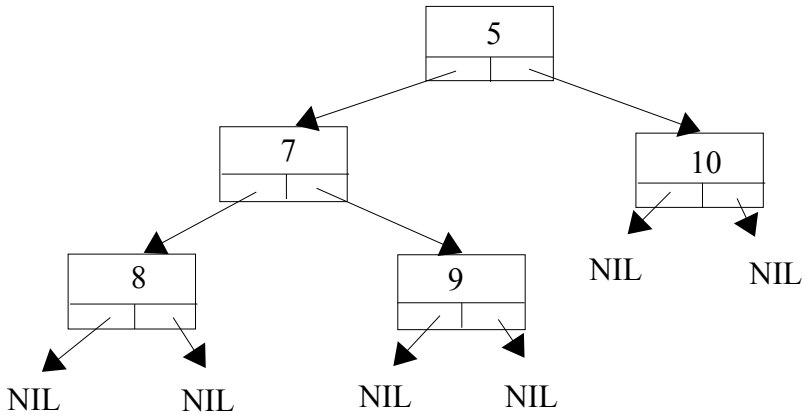
      BspBaum.: 3 – 4 – 2 – 7 – 8 – 6 – 5 – 1

2) Breitensuche
   Knoten werden nach ihrer Tiege gelistet und zwar bei gleicher Tiefe von links nach rechts
   Bsp.: 1, 2, 5, 3, 4, 6, 7, 8

   Verschiedene Strategien haben unterschiedliche Vor- und Nachteile.

Realisierung in Modula3

```
TYPE Tree = REF RECORD
                        key : INTEGER;
                        left, right : Tree;
                    END;
VAR root : Tree;
```

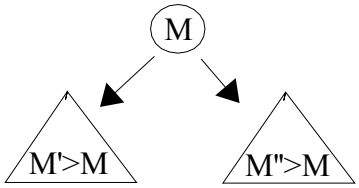



```
PROCEDURE Init(VAR t : Tree) =
BEGIN
  t := NIL;
END Init;

PROCEDURE PrintPre(t:Tree) =
BEGIN
  IF (t # NIL) THEN
      SIO.PutInt(t^.key); SIO.PutText(' ');
      PrintPre(t^.left);
      PrintPre(t^.right);
  END;
END PrintPre;

PROCEDURE PrintIn(t:Tree) =
BEGIN
  IF (t # NIL) THEN
      PrintIn(t^.left);
      SIO.PutInt(t^.key); SIO.PutText(' ');
      PrintIn(t^.right);
  END;
END PrintPre;

PROCEDURE PrintPost(t:Tree) =
BEGIN
  IF (t # NIL) THEN
      PrintPost(t^.left);
      PrintPost(t^.right);
      SIO.PutInt(t^.key); SIO.PutText(' ');
  END;
END PrintPost;
```

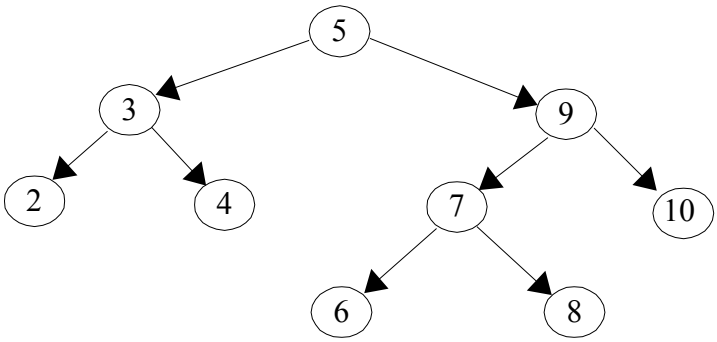
Binärer Suchbaum

Baum T = (K, E) $\quad \forall\ k \in K$

Markierung des linken Teilbaums sind kleiner als Markierung des Knotens k.
Markierungendes rechten Teilbaums sind größer als Markierungdes Knotens k.



Bsp.:

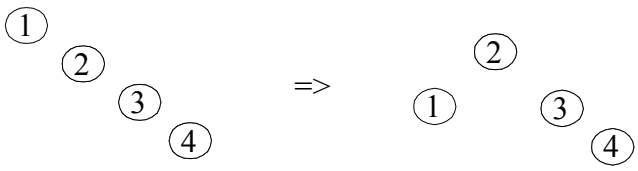


Suchen in binären Suchbäumen

```
PROCEDURE Search(x : INTEGER; t : Tree) : BOOLEAN =
BEGIN
   IF (t = NIL) THEN
       RETURN FALSE;
  ELSIF (x = t^.key) THEN
       RETURN TRUE;
  ELSIF (x < t^.key) THEN
       RETURN (Search(x, t^.left);
   ELSE (*  x > t^.key *)
       RETURN (Search(x, t^.right);
   END;
END Search;
```

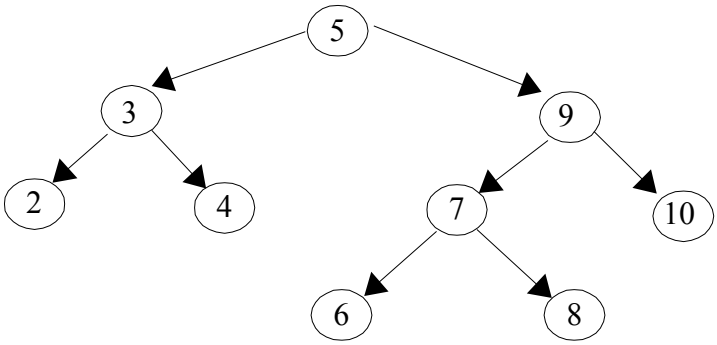
Laufzeitanalyse

- für ausgeglichene Suchbäume $\in$ $O(\log_2 n)$
- für degenerierte Suchbäume $\in$ $O(n)$



Aufbau eines Suchbaums

root -> NIL;
Einfügen von 5, 3, 4, 9, 2, 10, 7, 8 , 6




```
PROCEDURE Insert(x : INTEGER; VAR t : Tree) =
BEGIN
  IF (t = NIL) THEN
      t := NEW(Tree);
      t^.key := x;
      t^.left := NIL;
      t^.right := NIL;
  ELSIF (x < t^.key) THEN
      Inser(x, t^.left);
  ELSE (* x > t^.key *)
      Inser(x, t^.right);
  END;
END Insert;
```

11.06.01

Divide & Conquer Strategie

Divide: Zerlegung des Problems in Teilprobleme bis es „elementar“ gelöst werden kann.
Conquer: setzt die Teillösungen zur Gesamtlösung zusammen.

Bsp.: Gleichzeitiges bestimmen von Maximum und Minimum einer Folge
Idee:

- Zerlege die Folge in zwei gleichgroße Teilfolgen bis eine Teilfolge nur noch aus 2 Elementen besteht.
- Bestimme Max und Min
- Setze Teilfolgen wieder zusammen und berechne Max und Min.

Bsp.:

7 12 18 4 1 8 5 9

7 12 | 18 4 | 1 8 | 5 9

Min7 Min 4 Min 1 Min 5
Max 12 Max 18 Max 8 Max 9

Min 4 Min 1
Max 18 Max 9
Min 1
Max 18

Allgemein im Skript Seite 51.

$T(n) = 3/2\ n - 2$

Sei T(n) die Zahl d Vergleich, dann gilt

$T(n) = 1$, falls $n = 2$
$2 * T(n/2) + 2$, falls $n > 2$

$T(2) = 1$
$T(4) = 2 * T(2) + 2 = 2 *1 + 2 = 4$
$T(8) = 2 * T(4) + 2 = 2 * (2 * T(2) + 2) + 2 = 2 * 4 + 2 = 10$
$T(16) = 2 * T(8) + 2 = 2 * (2 * T(4) + 2) + 2 = 2 * (2 * (2 * T(2) + 2) + 2) + 2 = 22$

Behauptung: $T(n) \in O(n)$

Master-Theorem für Rekursionsgleichungen

$T(n) = 1$, für „n = 1“
$a * T(n/b) + d(n)$, für $n > 1$

n: Größe der Eingabe
a : Wieviele Teilprobleme müssen gelöst werden
b (bzw n/b): Größe der Teilprobleme
d(n): Funktion, die zusätzlichen Aufwand berechnet

Dann gilt für $d(n) \in O(n^{\gamma})$

$T(n) = O(n^{\gamma})$ $a < b^{\gamma}$
$O(n^{\gamma} * \log_b n)$ $a = b^{\gamma}$
$O(n^{\log_b a})$ $a > b^{\gamma}$

im Beispiel: $a = 2, b = 2, \gamma = 0$ $\Rightarrow 2 > 2^0$
$\Rightarrow T(n) \in O(n^{\log_2 2})$
$= T(n) \in O(n)$

i) $T(n) = 8 * T(n/3) + n^2$
$a = 8, b = 3, \gamma = 2$ $\Rightarrow 8 < 3^2 \iff 8 < 9$
$\Rightarrow T(n) \in O(n^2)$

ii) $T(n) = 9 * T(n/3) + n^2$
$a = 9, b = 3, \gamma = 2$ $\Rightarrow 9 = 3^2 \iff 9 = 9$
$\Rightarrow T(n) \in O(n^2 * \log_3 n)$

iii) $T(n) = 10 * T(n/3) + n^2$
$a = 10, b = 3, \gamma = 2$ $\Rightarrow 10 > 3^2 \iff 10 > 9$
$\Rightarrow T(n) \in O(n^{\log_3 10})$

Sortieren

- Prozess des Anordnens einer gegebenen Menge von Objekten in einer bestimmten Ordnung.
- Ordnungsfunktion f

  $f(a[k_1]) \leq f(a[k_2]) \leq ... \leq f(a[k_n])$

Bsp.: 44 – 55 – 12 – 42 – 94 – 18 – 6 – 67

Maße für Effizenz:

C := Anzahl der Vergleiche
M := Anzahl der Bewegungen (Umstellungen) der Elemente

1) Selection Sort (Sortieren duch Auswahl)
   Idee: 1. Auswahl des Elements mit dem kleinsten Schlüssel
   2. Austauschen gegen das erste Element der u betrachtenden Folge (unsortierte Teilfolge)

Bsp.:

44 55 12 42 94 18 **6** 67

6 | 55 **12** 42 94 18 44 67

6 12 | 55 42 94 **18** 44 67

6 12 18 | **42** 94 55 44 67

6 12 18 42 | 94 55 **44** 67

6 12 18 42 44 | **55** 94 67

6 12 18 42 44 55 | 94 **67**

6 12 18 42 44 55 67 | **94**

Algorithmus

FOR i := 1 TO n-1 DO
- finde Index mit dem kleinsten Element von a[i] ... a[n]
- weise Index min zu
- vertausche a[i] und a[min]

END;

Realisierung in Modula3

```
CONST N = 10;
TYPE Feld = ARRAY [1..N] OF INTEGER;

PROCEDURE SelectionSort (VAR f : Feld)=
VAR min : CARDINAL;
    help : INTEGER;
BEGIN
  FOR i := 1 TO N-1 DO
      min := i;
      FOR j := i+1 TO N DO
            IF f[j] < f[min] THEN
                  min := j;
            END;
      END;
      help := f[min];
      f[min] := f[i];
      f[i] := help;
  END;
END SelectionSort;
```

Komplexität: $O(n^2)$

2) Insertion Sort (Sortieren duch Einfügen)

Idee:
FOR i:= 2 TO n DO
- x := a[i]
- füge x am entsprechenden Platz in a[1]..a[i] ein

END;

Bsp.:

44 | **55** 12 42 94 18 6 67

44 55 | **12** 42 94 18 6 67

12 44 55 | **42** 94 18 6 67

12 42 44 55 | **94** 18 6 67

12 42 44 55 94 | **18** 6 67

12 18 42 44 55 94 | **6** 67

6 12 18 42 44 55 94 | **67**

6 12 18 42 44 55 67 94 |

Realisierung in Modula3

```
CONST N = 10;
TYPE Feld = ARRAY[0..N] OF INTEGER;

PROCEDURE InsertionSort(VAR f : Feld) =
VAR   j : CARDINAL;
      x : INTEGER;
BEGIN
  FOR i := 2 TO N DO
    x := f[i];
    f[0] := x;
    j := i-1;
    WHILE x < f[j] DO
      f[j+1] := f[j];
      j := j-1;
    END;
    f[j+1] := x;
  END;
END InsertionSort;
```

Komplexität: $O(n^2)$

3) Bubble Sort

Idee: Durch wiederholtes Vertauschen von benachbarten Elementen wandert pro Durchlauf das größte Element an das rechte Ende des Feldes.

I = 1	*44*	**55**	**12**	42	94	18	6	67
	44	*12*	**55**	**42**	94	18	6	67
	44	*12*	*42*	*55*	**94**	**18**	6	67
	44	*12*	*42*	*55*	*18*	**94**	**6**	67
	44	*12*	*42*	*55*	*18*	*6*	**94**	**67**
	44	*12*	*42*	*55*	*18*	*6*	*67*	*94*
I = 2	12	42	44	18	6	55	67	94
I = 3	12	42	18	6	44	55	67	94
I = 4	12	18	6	42	44	55	67	94
I = 5	12	6	18	42	44	55	67	94
I = 6	6	12	18	42	44	55	67	94

Realisierung in Modula3

```
CONST N = 10;
TYPE Feld = ARRAY [1..N] OF INTEGER;

PROCEDURE BubbleSort (VAR f : Feld) =
VAR help := INTEGER;
BEGIN
  FOR i := N TO 1 BY -1 DO
    FOR j := 2 TO i DO
      IF f[j-1] > f[j] THEN
        help := f[j-1];
        f[j-1] := f[j];
        f[j] := help;
      END;
    END;
  END;
END BubbleSort;
```

Komplexität: $O(n^2)$

Vergleich der einfachen Sortierverfahren:

	Vergleich C			Bewegungn M		
	Best	Average	Worst	Best	Average	Worst
SelectionSort	$N^2 / 2$	$N^2 / 2$	$N^2 / 2$	3(N-1)	3(N-1)	3(N-1)
InsertionSort	N	$N^2 / 4$	$N^2 / 2$	2(N-1)	$N^2 / 4$	$N^2 / 2$
BubbleSort	$N^2 / 2$	$N^2 / 2$	$N^2 / 2$	0	$3\ N^2 / 4$	$3\ N^2 / 2$

18.06.01

Höhere Sortierverfahren

Vorteil: (Viel) bessere Laufzeit als einfache Sortierverfahren
Nachteil: Algorithmen sind komplexer
eignen sich nur für große Mengen

1) Quicksort

- wurde bereits 1962 vib C.A.R. Hoane entwickelt
- gebräuchlichste interne Sortiermethode

Idee:
1) Man wählt willkürlich ein Element X der Folge aus (Pivot-Element)
2) Durchlaufe das Feld von links, bis ein Element a[i] > X gefunden wird.
3) Durchlaufe das Feld vin rechts, bis ein Element a[j] < X gefunden wird.
4) Vertausche diese beiden Elemente a[i] und a[j]. Setze i := i+1 und j := j-1
5) Wiederhole die Schritte 2 bis 4 bis sich die Zeiger i und j kreuen (also j < i)

==> Partitionierung

Bsp.:

44$_{i}$	55	12	**42**$_{x}$	94	6	18	*67*$_{j}$
44$_{i}$	55	12	**42**$_{x}$	94	6	*18*$_{j}$	67
18	*55*$_{i}$	12	**42**$_{x}$	94	*6*$_{j}$	44	67
18	6	*12*$_{i}$	**42**$_{x}$	*94*$_{j}$	55	44	67
18	6	12	**42**$_{xij}$	94	55	44	67
18	6	*12*$_{j}$	**42**$_{x}$	*94*$_{i}$	55	44	67
$\forall_{k=1}^{i-1} a[k] \le x$				$\forall_{k=j+1}^{n} x \le a[k]$			

Aber: Ziel ist nicht das Finden von Zerlegungn, sonder das Sortieren!
=> Lösung: Nach der Zerlegung wendet man den gleichen Prozeß auf die beiden Teile, dann auf deren neuen Zerlegungen, usw.... bis jeder Teilnur noch ein Element umfaßt. (=> Rekursion)

Realisierung in Modula3

```
PROCEDURE QuickSort (VAR a : Feld) =

  PROCEDURE Sort (l, r : CARDINAL) =
  VAR i, j : CARDINAL;
       x, w : INTEGER;
  BEGIN
      i := l;
      j := N;
      x := a[l + r DIV 2]; (* Pivot Element *)
      REPEAT
            WHILE a[i] < x DO
                  i := i+1;
            END;
            WHILE a[j] > x DO
                  j := j-1;
            END;
            IF (i <= j) THEN
                  w := a[i];
                  a[i] := a[j];
                  a[j] := w;
                  i := i+1;
                  j := j-1;
            END;
      UNTIL i > j;
      IF l < j THEN
            Sort(l, j);
      END;
      IF r > i THEN
            Sort(i, r);
      END;
  END Sort;

BEGIN
  Sort(1,N);
END QuickSort;
```

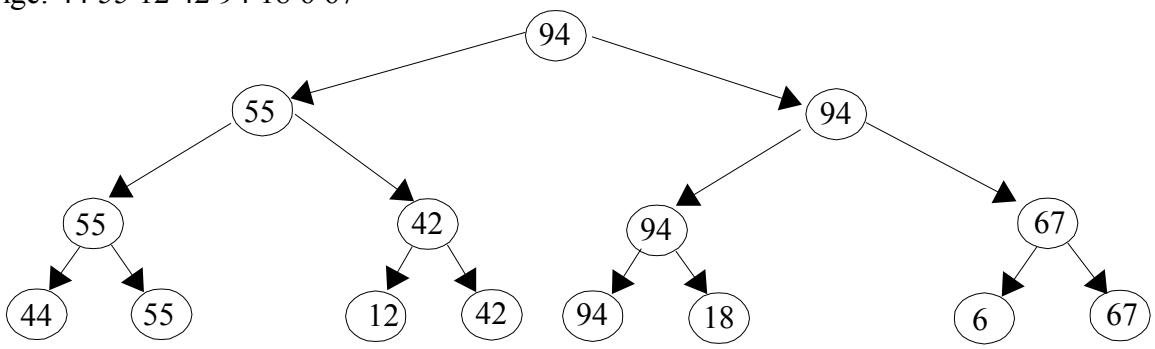
Analyse
Auswahl von x ist entscheidend.
Best Case: x ist immer „mittleres“ Element der Folge
=> optimale Partitionierung |---M---| X |---M---|
=> O (n log n)
Worst Case: x ist immer größtes Element der Folge
=> keine partitionierung |------M------| X
=> n statt log n Zerlegungen sind notwendig
=> insgesammt $O(n^2)$

Ziel: gutes Pivot Element finden.

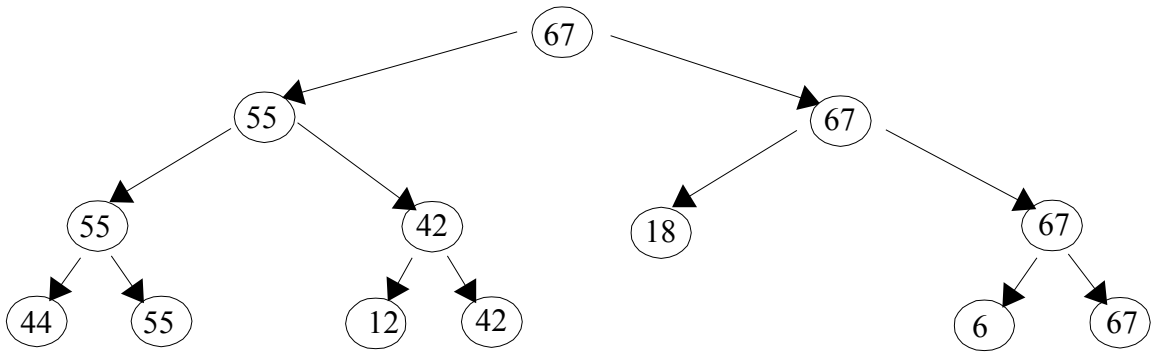
2) Heap Sort

- Erinnerung: Sortieren durch Auswählen
- Ziel: kann von jedem Durchlauf mehr Informationen gehalten werden, als nur das kleinste Element?
- Idee: Größeren Schlüssel von jedem Paar identifizieren
  => aus n-1 Vergleichen Auswahlbaum konstruieren.

Folge: 44 55 12 42 94 18 6 67



nach der Ausgabe des größten Schlüssels (94):

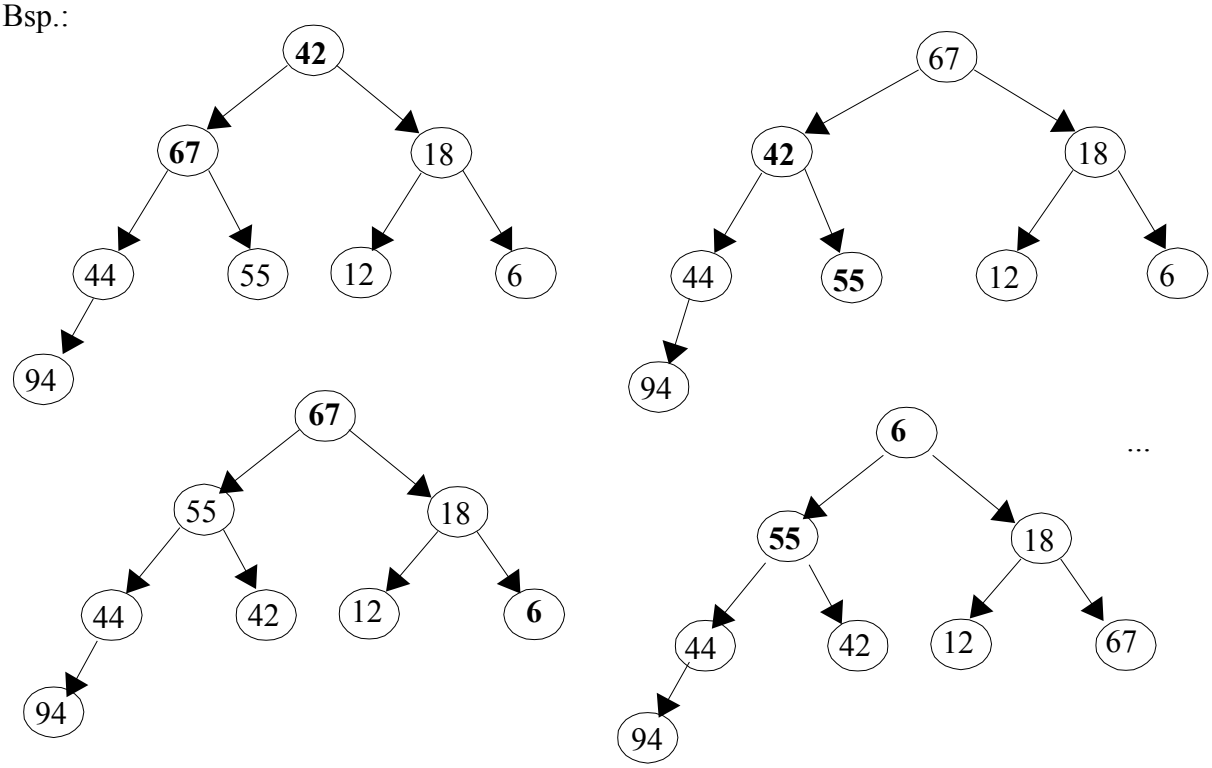


Baum wird nicht jedes mal nach Ausgabe verworfen, sonder neu konstruiert.
Jetzt: Baumverwaltung ohne 'Löcher'
n statt 2n Speicherplätze
=> Heapsort

Def.: Eine Folge von Schlüsseln $h_l, h_{l+1}, \ldots, h_{r-1}, h_r$ heißt genau dann Heap, wenn
$h_i >= h_{2i}$ und $h_i >= h_{2i+1}$ für i = L ... R/2
=> Der Schlüssel jedes inneren Knotens ist größer oder gleich dem Schlüssel seiner beiden Söhne.

Jetzt: Abspalten des größten Elements (Wurzel / a[i])
d.h. Vertauschen von a[i] und a[R] (letztes Element)
und stelle erneut Heapeigenenschaften her.

Bsp.:



Als Feld:

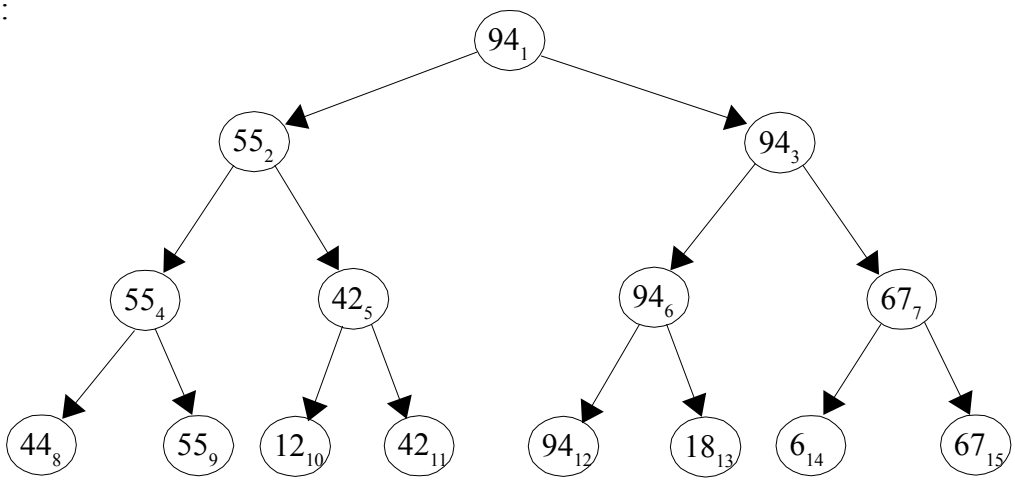
44	55	12	42	94	18	6	67		Heapaufbau
94	67	18	44	55	12	6	42		
42	67	18	44	55	12	6	94		Heapaufbau
67	55	18	44	42	12	6	94		
6	55	18	44	42	12	67	94		
...									
6	12	18	42	44	55	67	94		

Realisierung in Modula3

```
PROCEDURE HeapSort (VAR a : Feld) =
VAR l, r : CARDINAL;
       x : INTEGER;

PROCEDURE Sift (l,r : CARDINAL) =
VAR i, j : CARDINAL;
        x : INTEGER;
BEGIN
  i := l;
  j := 2*l;
  x := a[l];
  IF ((j < R) AND (a[j] < a[j+1])) THEN
      j := j+1;
  END;
  WHILE ((j <= r) AND (x < a[j])) DO
      a[i] := a[j];
      i := j;
      j := 2*i;
      IF ((j < R) AND (a[j] < a[j+1])) THEN
            j := j+1;
      END;
  END;
  a[i] := x;
END Sift;

BEGIN
  l := (N DIV 2) + 1;
  r := N;
  WHILE (l > 1) DO
      l := l-1;
      Sift(l,r);
  END;
  WHILE (r>1) DO
      x := a[1];
      a[i] := a[r];
      a[N] := x;
      r := r-1;
      Sift(l,r);
  END;
END HeapSort;
```

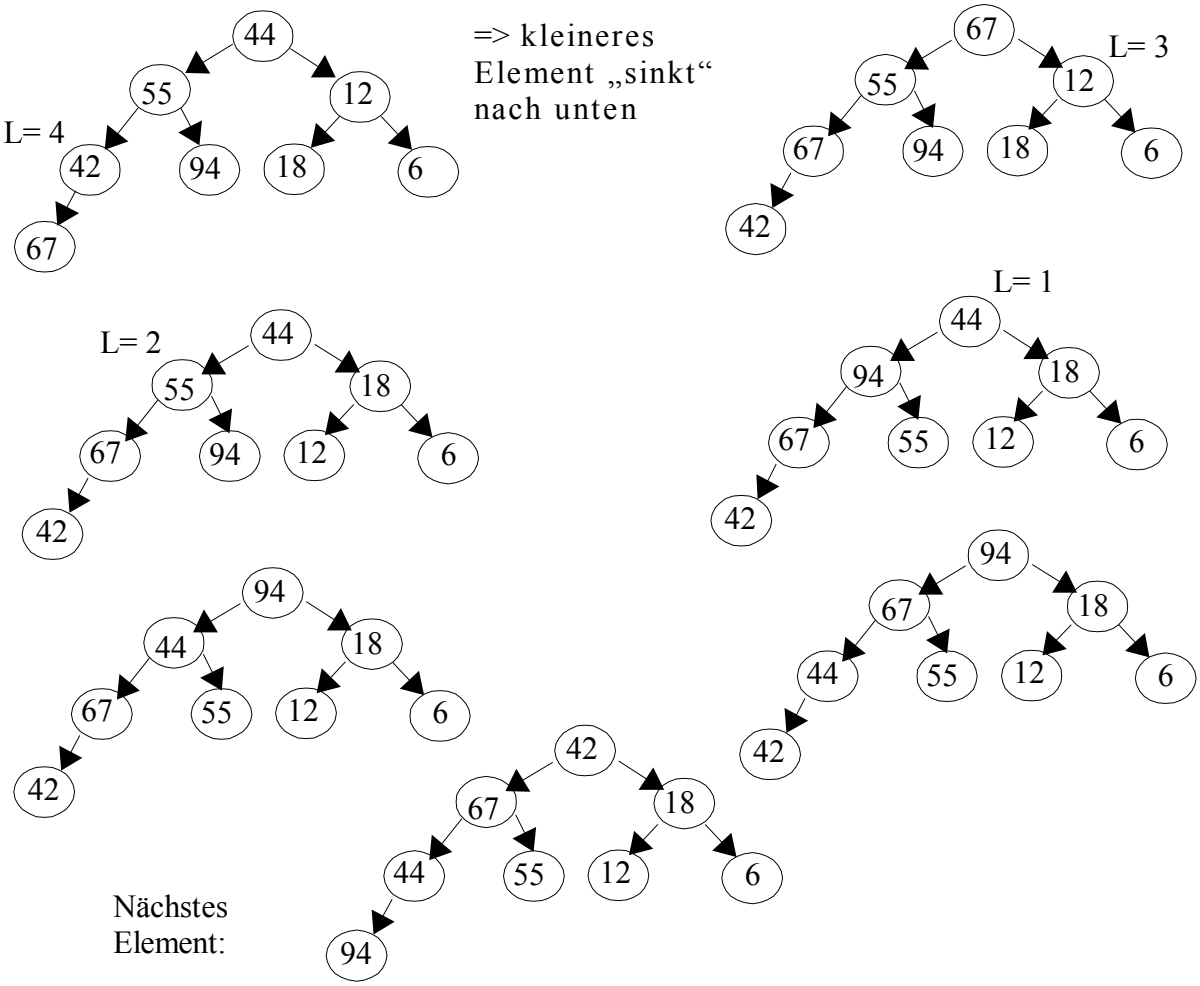
Laufzeit: O (n log n)

Bsp.:



94	55	94	55	42	94	67	44	55	12	42	94	18	6	67

=> Binärbaum mit Bedingung H (Heap)
=> Schichtenweise Abspeicherung liefert Heap
Betrachte Elemente $h_{(n/2)+1}$, ... , h_n haben bereits Heapeigenenschaften.
Konstruktion des anfänglichen Heaps zu einer Folge
Eingabe: 44 55 12 42 34 18 6 67



25.06.01
Suchen in Mengen

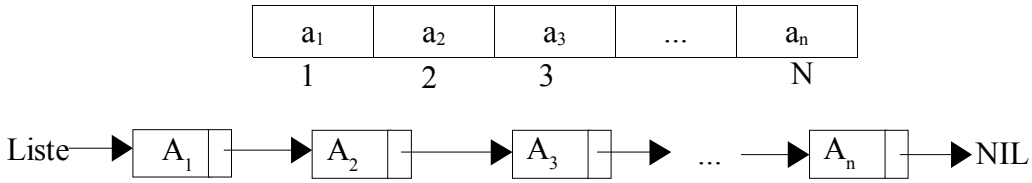
Gegeben: Eine Menge von Elementen mit Schlüssel- und Inhaltskomponente. (keine Duplikate)
Aufgabe: Finde zu einem gegebenen Schlüsselwert das zugehörige Element und führe u.U. eine Operation aus.

Operationen:

Search(x,S):	Falls $x \in S$, gib dieses Element zurück, sonst Meldung, daß $x \notin S$ (Alternativer Rückgabewert: TRUE / FALSE)
Insert(x,S):	Füge Element x zur Menge S hinzu. $S := S \cup \{x\}$ Fehlermeldung, falls x schon in S enthalten ist.
Delete(x,S):	Entferne Element x aus Menge S. $S := S \setminus \{x\}$ Fehlermeldung, falls x nicht in S enthalten ist.

Laufzeit der Operationen hängen zum einen von ihrer Implementierung ab und zum anderen, ob eine Ordnung auf der menge S existiert.

1) ungeordnete Arrays und Listen



Lineare Suche

```
PROCEDURE LineareSuche (x : INTEGER; a : Feld) : BOOLEAN =
VAR i : CARDINAL;
    gefunden : BOOLEAN;
BEGIN
  i := 1;
  gefunden = FALSE;
  WHILE (i <= N) AND (NOT gefunden) DO
      IF (a[i] = x) THEN
            gefunden = TRUE;
      END;
      INC(i);
  END;
  RETURN gefunden;
END LineareSuche;
```

Komplexität
n/2 Vergleiche bei erfolgreicher Suche (im Schnitt)
n Vergleiche bei erfolgloser Suche
=> O(n)

Einfügen: O(n) da zunächst überprüft werden muß, ob das Element schon in S enthalten ist.
=> das ist unabhängig davon, ob die Menge als Array oder als Liste implementiert ist

Löschen: O(n) da zunächst überprüft werden muß, ob das Element in S enthalten ist oder nicht.

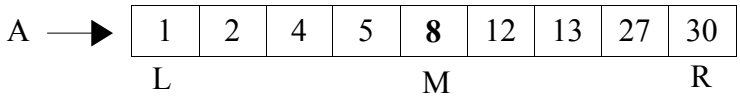
2) Geordnete Menge (Arrays)

- lineare Suche (sieh ungeordnete Arrays)
- Binärsuche
- Interpolationssuche

Binärsuche

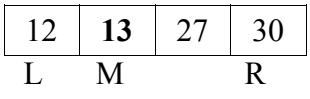
Idee: sukzessives Halbieren

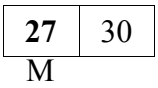
Bsp.:



Search(27,a): Prüfe mittleres Element

Falls zu suchendes Element kleines als mittleres Element ist, dann duchsuche „linken Teil“, sonst „rechten Teil“





Gefunden: Anstatt 8 Vergleiche bei linearer Suche nur 3 Vergleiche.

Implementierung in Modula3

```
PROCEDURE Binaersuche(x : INTEGER; a : FELD) : BOOLEAN =
VAR links, rechts, pos : CARDINAL;
    gefunden : BOOLEAN;
BEGIN
  gefunden := FALSE;
  links := 1;
  rechts := N;
  WHILE (links <= rechts) AND (NOT gefunden) DO
     pos := (links + rechts) DIV 2;
     IF (a[pos] = x) THEN
           gefunden := TRUE;
     ELSIF (a[pos] < x) THEN
           links := pos+1;
     ELSE (*  a[pos] > x *)
           rchts := pos-1;
     END;
  END;
  RETURN gefunden;
END Binaersuche;
```

Komplexität: $O(\log_2 n)$

Interpolationssuche

Idee: - Schnellere lokalisierung des Suchbereichs, indem Schlüsselwerte selbst betrachtet werden, um „Abstand“ zum Element x abzuschätzen.

– Nächste Suchposition „pos“ wird aus den Werten „links“ und „rechts“ der jeweiligen Unter- und Obergrenze dse aktuellen Surchbereichs berechnet.

$$pos := links + \lceil \frac{x - a[links]}{a[rechts] - a[links]} \cdot (rechts - links) \rceil$$

Komplexität

Average Case: O ($\log_2 \log_2 n+1$)

Schlüsselwert im betreffenden Bereich gleich verteilt

Worst Case: ungleichemäßige Wertverteilung -> O(n)

Bsp.:

A ⟶

1	2	4	5	8	12	13	27	30

Links := 1; rechts := 9; Search (30,A);

$$pos := 1 + \lceil \frac{30-1}{30-1} \cdot (9-1) \rceil = 1+8 = 9$$

=> Element mit nur einem Vergleich gefunden.

Search(5,A);

$$pos := 1 + \lceil \frac{5-1}{30-1} \cdot (9-1) \rceil = 1+2 = 3$$

neue Grenzen 3 und 9

$$pos := 3 + \lceil \frac{5-4}{30-4} \cdot (9-3) \rceil = 3+1 = 4$$

=> gefunden.

Sehr schnelle Verfahren bei gleichmäßig verteilten Werten.

Realisierung: (ähnlich Binärsuche)

```
pos := links + (((x-a[links]) DIV (a[recht] - a[links]))
                  + (rechts - links))
```

Zusätzliche Prüfung:

```
IF (pos >= 1) AND (pos <= N) THEN
    ....
END;
```

03.07.07
Hashing

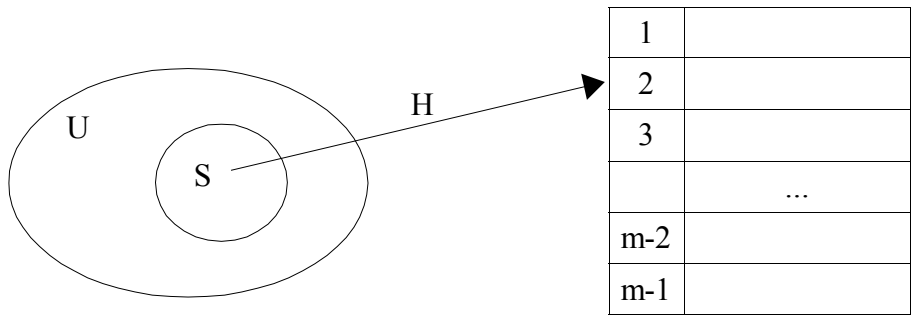
Hashing ist eine Methode zum dynamischen Verwalten von Daten, wobei diese durch einen Schlüssel angesprochen werden.

Opterationen:
- Suchen nach einem Datensatz, bei gegebenem Schlüssel.
- Einfügen eines neuen Datensatzes (mit Schlüssel)
- Löschen eines Datensatzes, bei gegebenem Schlüssel.

Ziel: Möglichst „schnelle“ Realisierung der Operationen

Bisher: Binärbäume
-> Operationen in $O(\log_2 n)$ (best case) bzw. O(n) (worst case)
Ziel: Mittlerer Aufwand von O(1)

Szenario:
- Große Menge U von potentiellen Schlüsseln (Universum)
- relativ kleine Menge tatsächlich verwendeter Schlüssel $S \subseteq U$
- Verwendung einer sog. Hashfunktion h, welche auf einen Schlüssel angewedent wird und damit eine Adresse in der Hashtabelle T liefert. Indizierung von 0...m-1




```
TYPE T = ARRAY [0..m-1] OF „Element“
```

Hashfunktion

h: U -> {0,1,...,m-1}
x -> h(x)

Annahme: Je zwei verschiedene tatsächlich verwendeter Schlüssel s, s' ∈ S haben verschiedene Hashwerte, also $h(s) \neq h(s')$

Operation	Implementierung	Komplexität
Suchen S	`IF T[h(s)] # 'leer' THEN...`	O(1)
Einfügen S	`T[h(s)] := 'Daten' zu Schlüssel S`	O(1)
Löschen S	`T[h(s)] := 'leer'`	O(1)

Hashfunktion

Anforderungen:
- ganze Hashtabelle sollte abgedeckt werden.
- H(x) soll Schlüssel gleichmäßig verteilen
- Berechnung sollte effizient sein.

Division-Rest-Methode
m = Größe der Hashtabelle (möglichst Primzahl)

h(x) = x MOD m

Bsp.: m=7 U = {1, ... , 100}
Einfügen:
2, 12, 36, 74

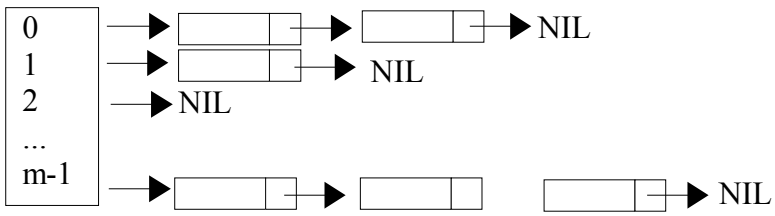
43 MOD 7 = 1 – dort ist aber bereits ein Wert gespeichert => Kollision

0	
1	36
2	2
3	
4	74
5	12
6	

43

Problem: Kollisionen
Frage: Wie löst man Kollisionen h(s) = h(s') ?

1) Hashing mit Verkettung (offenes Hashing)
Jedes Element der Hashtabelle dient als Ausgangspunkt für alle Schlüssel mit dem selben Hashwert



Komplexität:
- Adressbrechnung: O(1)
- Behälter aufsuchen: O(1)
- Aber: Liste durchsuchen O(n) im worst case

2) Open Hashing
Falls h(s) = h(s'), s bereits in T und s' neu, dann wird für s' ein alternativer freier Platz in T gesucht.
=> Maximal m Elemente speicherbar.

Achtung: Sonderbehandlung beim Löschen: Mögliche nachfolgende Elemente müssen erreichbar bleiben. Markierung mit 'empty' und 'deleted'

Strategien:
1) Lineare Sondierung

Idee:
- Im Kollisionsfall wird als alternativer Platz der benachbarte Platz in der Hashtabelle gesucht.
- Falls h(s) = h(s'), s' neu, dann (h(s)+1) MOD m, wenn der auch voll ist, dann h(s)+2 MOD m, etc. (h(s)+i) MOD m mit i=0,...,m-1
- Falls s gesucht wird, dan sondiert man h(s), h(s)+1 MOD m, h(s)+2 MOD m, ...
- Falls s gelöscht werden soll, muß man nach der Suche von s den schlüssel löschen und die Position mit der Markierung 'deleted' markieren.

Nachteil: Clustering – die Tendenz, daß immer längere zusammenhängende, belegte Abschnitte in der Hashtabelle entstehen.

Bsp.: Einfügen von Werten:

12 MOD 7 = 5
8 MOD 7 = 1
50 MOD 7 = 1
120 MOD 7 = 1
65 MOD 7 = 2

0	
1	8
2	50
3	120
4	65
5	12
6	

2) Quadratisches Sondieren
Häufung vermeiden, durch quadratisches Sondieren.
$h(x,i) = (h(x) + i^2) \text{ MOD } m$, i = 0,..., m-1
Bsp.:
8 MOD 7 = 1
50 MOD 7 = 1 (Kollision) = $1 + 1^2 = 2$
120 MOD 7 = $1 + 1^2 = 2$ (Kollision) = $1 + 2^2 = 5$
65 MOD 7 = 2 (Kollision) = $2 + 1^2 = 3$

0	
1	8
2	50
3	65
4	
5	120
6	

3) Double Hashing
Häufung vermeiden, durch Cerwendung einer zweiten Hashfunktion h'. Diese bestimmt im Kollisionsfall für den jeweiligen Schlüssel die Schrittweite mittels der dann in der Hashtabelle weitersondiert werden soll.
Sondierreihenfolge:
$h_1(x) = h(x)$
$h_2(x) = h(x) + h'(x) \text{ MOD } m$
$h_3(x) = h(x) + 2 * h'(x) \text{ MOD } m$
....
$h_n(x) = h(x) + (n-1)*h'(x) \text{ MOD } m$

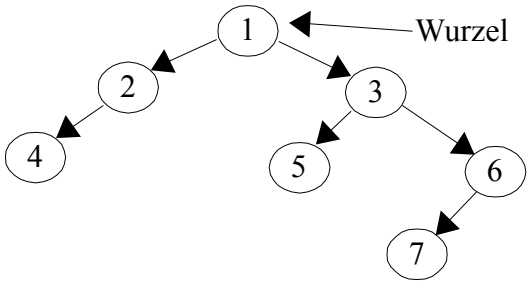
Beachte
- h' möglichst unabhängig von h definieren.
- Tabellengröße sollte eine Primzahl sein, denn wenn m und h'(x) nicht teilerfremd sind, dann wird beim Sondieren nicht die gesamte Tabelle durchlaufen.
- Wert von h'(x) darf nicht Null sein!

Zusammenfassung: average case: Hashverfahren zeigen effizientes Verhalten (O(1))
worst case: O(n)

09.07.01

Balancierte Suchbäume (AVL-Bäume)

Erinnerung:
Baum ist ein zyklenfreier, gerichteter Graph, wobei jeder Knoten höchstens einen Vorgänger hat und bei dem jeder Knoten des Baumes von der Wurzel erreichbar ist.

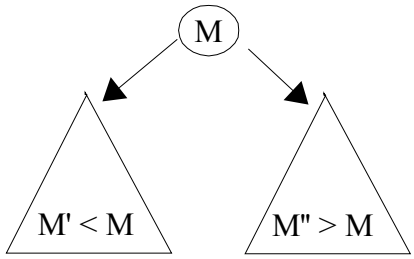


Tiefe eines Knotens: $level(k) = \begin{cases} 1, falls\ k\ Wurzel \\ level(Vorg.(k)) + 1, sonst \end{cases}$

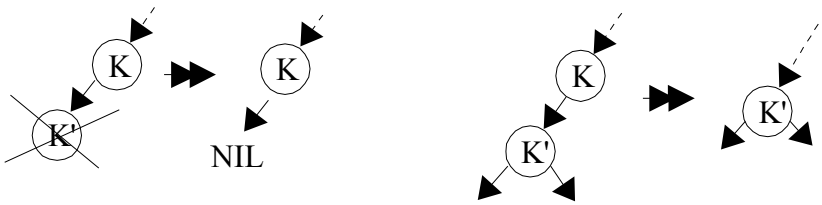
Höhe eines Baums: max{level(k)} $\forall$ k $\in$ K

Binärer Suchbaum

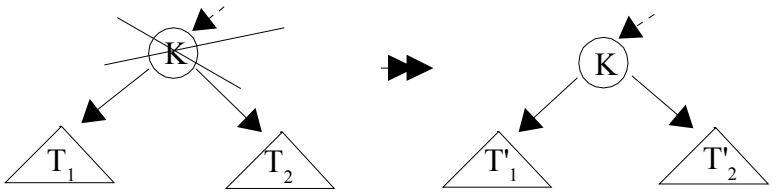
Baum T = (K, E) mit



Suchen + Einfügen in binären Suchbaum siehe Skript.
Löschen eines Knotens in binären Suchbäumen
Vorüberlegung:
a) Löschen eines Blattes oder eines Knotens mit nur einem Nachfolger (einfach!)



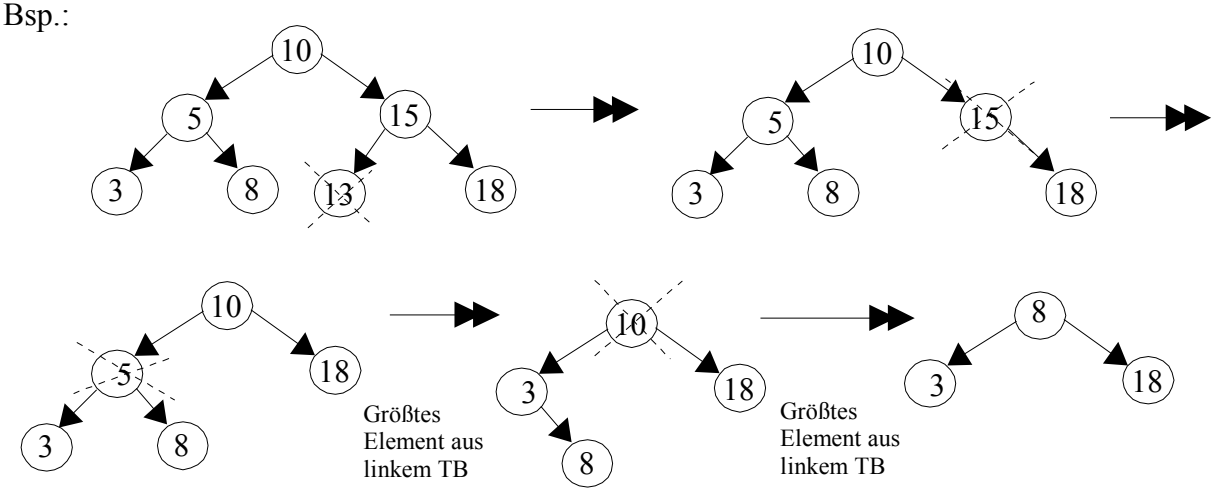
b) Knoten mit zwei Nachfolgern



2 Möglichkeiten

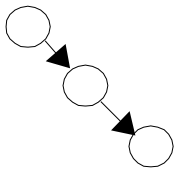
i) k' ist größtes Element aus T_1 dann ist $T_2' = T_2$
ii) k' ist kleinstes Element aus T_2 dann ist $T_1' = T_1$

Beachte: kleinstes Element aus T_2 oder größtes Element aus T_1 haben höchstens einen Nachfolger!
Bsp.:



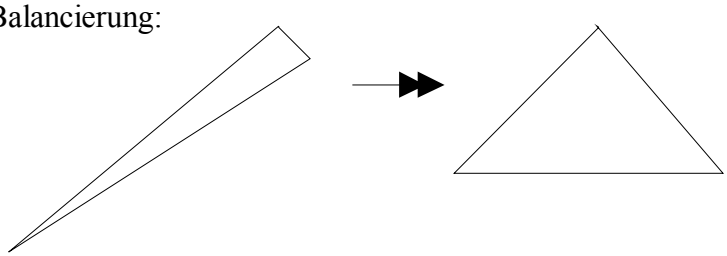
Beachte: Binäre Suchbaum-Eigenschaft bleibt immer erhalten!
Realisierung in Modula3 siehe Musterlösung zur Testklausur (Übungsblatt 7)

Problem: Man kann zeigen, daß Binäre Suchbäume im Mittel logarithmischen Aufwand für Suche, Einfügen und Löschen haben. O ($\log_2$ n)
Voraussetzung: Baum ist ausgeglichen.
Bei ungünstiger Eingabesequenz kann Baum zu linearer Liste degenerieren.



Dann Operationen in
O(n)

Lösung: Balancierung:



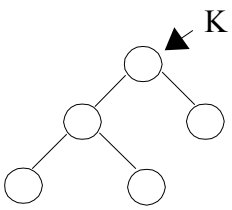
Garantierung von O (log n)

AVL-Bäume

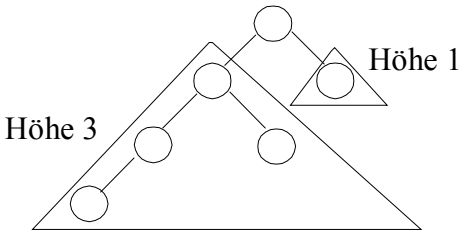
- entwickelt 1962 von Adelson-Velskij und Landis
- historisch erste Version eines balancierten Baums

Def.: Ein AVL-Baum ist ein binärer Suchbaum mit einer Struktur-Invarianten:
Für jeden Knoten gilt, daß sich dir Höhe seiner beiden Teilbäume höchsten um eins unterscheidet.

Bsp.:



AVL-Baum



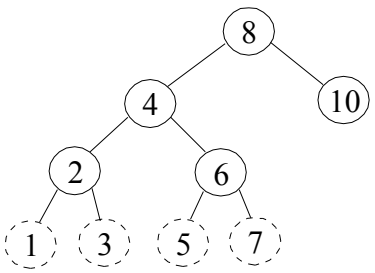
Kein AVL-Baum

Einfügen in AVL Baum (in linken Teilbaum – rechter Teilbaum analog)

1. $h_l = h_r$: Höhen werden verschieden, aber AVL-Eigenschaften wird nicht verletzt (Höhenunterschied von 1 ist erlaubt)
2. $h_l < h_r$: Höhen werden gleich – AVL-Eigenschaft bleibt erhalten.
3. $h_l > h_r$: AVL-Eigenschaft wird verletzt! => Baum muß umstrukturiert werden.

1. Fall: 1 oder 3 einfügen
2. 2. Fall: 5 oder 7 einfügen

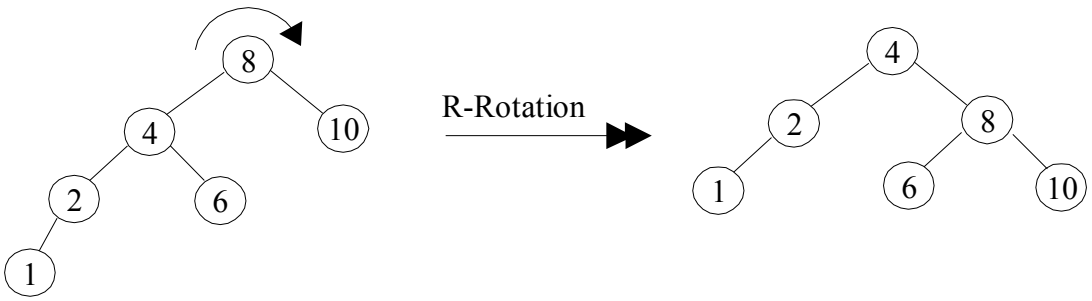
alle anderen Fälle hierzu symmetrisch



Operationen:
Standard-Operationen, wie beim Binären Suchbaum.
Zusätzlich Balanceüberprüfung vom eingefügten Knoten rückwärts bis zur Wurzel und testen für jeden Knoten, ob der Unterschied der Teilbäume größer als 1 ist.

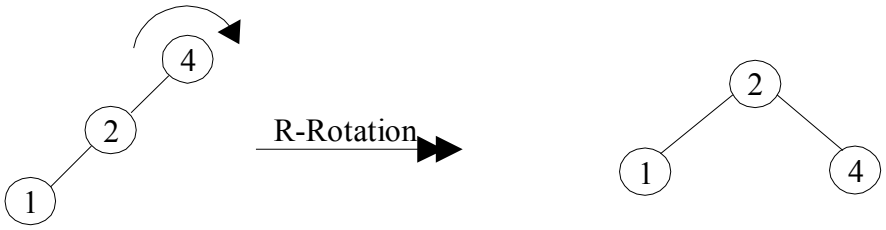
Rebalancierung durch Rotation:
a) Einfache Rotation
liegt nur vor, wenn der betroffene Teilbaum „außen“ liegt (Im Bsp. Fall 1). Dann rotiert der betroffene Knoten zum kürzeren Teilbaum hinunter und übernimmt den inneren Sohn des heraufrotierenden Knotens als inneren Sohn (es müssen nur 2 Knoten umgehängt werden).

Bsp.:



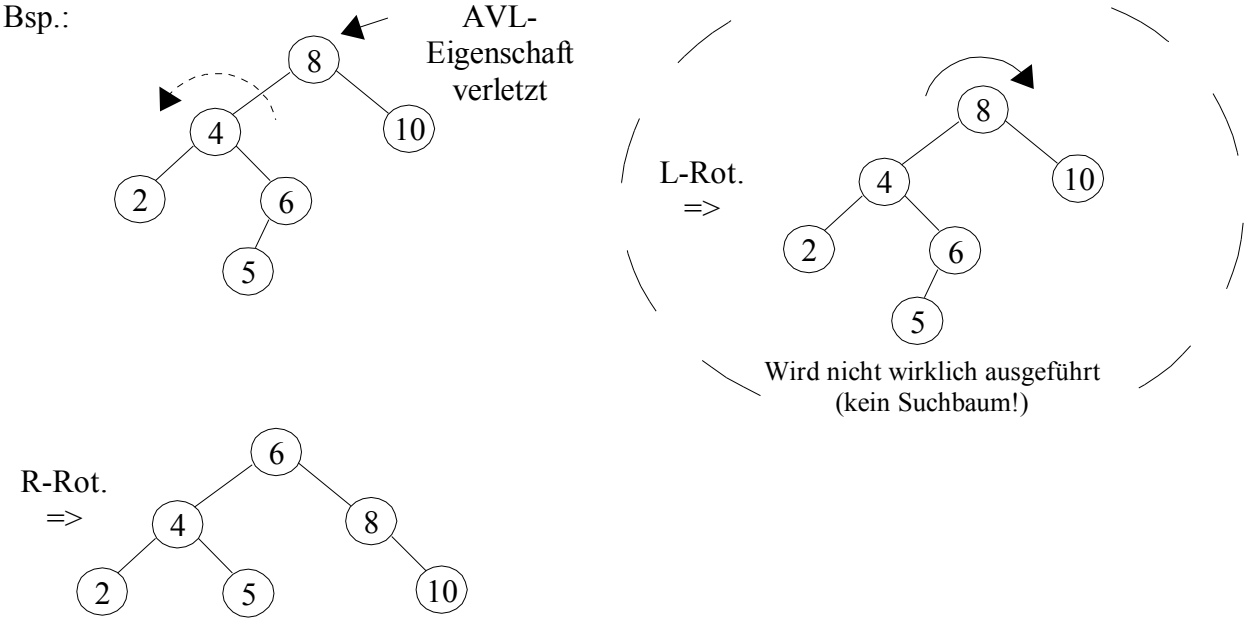
Analog: L-Rotation

Beachte: Funktioniert auch, wenn kein innerer Knoten existiert.
Bsp.:



b) Doppel-Rotation

liegt vor, wenn der betroffene Teilbaum „innen“ liegt. Dann wird eine Außenrotation im Vaterknoten der Wurzel des betroffenen Teilbaums durchgeführt und anschließend eine Rotation in entgegengesetzter Richtung im Vaterknoten dieses Knotens (es müssen 4 Knoten umgehängt werden).
Bsp.:



sogenannte LR-Rotation
Analog: RL-Rotation
Beachte: Bei allen Rotationen bleib Suchbaumeigenschaft erhalten!

16.07.01
(a, b) Bäume

1) Alle Blätter haben gleiche Tiefe
2) $\forall$ v gilt: $p(v) \leq s$
3) $\forall$ v außer Wurzel: $a \leq p(v)$
4) Wurzel hat mind. 2 Söhne