



Institut für Softwaretechnologie
Technische Universität Graz



VO Entwurf & Analyse von Algorithmen

Wintersemester 2009

Zusammenfassung der Theorie

Author:

Altinger Harald

HARALD.ALTINGER@STUDENT.TUGRAZ.AT

1 Grundlagen

1. Teilsummenproblem

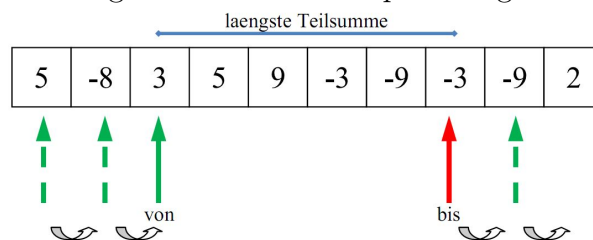
(a) Für jeden Index k die maximale Teilfolge T_k die bei k endet

- Laufzeit: $O(n)$

Listing 1: Scanline Pseudocode

```
1  i»; int max = 0, von = 0, bis = 0, v = 1, T = 0;
2  for(int k = 1; k <= n; k++)
3  {
4      T = T + A[k];
5      if(T < 0)
6      {
7          T = 0;
8          v = k + 1;
9      }
10     if(T > max)
11     {
12         max = T;
13         bis = k;
14         von = v;
15     }
16 }
```

Abbildung 1: Scanline: Prinzip des Algorithmus



2. Multiplikation langer Zahlen (Divide & Conquer)

(a) zwei Zahlen die zu lange sind um auf einmal berechnet zu werden

$$p = \sum_{n=1}^n P[i] \cdot 10^{n-i}$$
$$q = \sum_{n=1}^n Q[i] \cdot 10^{n-i} \tag{1}$$

- Divide & Conquer $\Rightarrow O(n^2)$

$$\begin{aligned} p &= a \cdot 10^{\frac{n}{2}} + b \\ p &= c \cdot 10^{\frac{n}{2}} + d \end{aligned} \tag{2}$$

$$\begin{aligned} p \cdot q &= (a \cdot 10^{\frac{n}{2}+b} \cdot c \cdot 10^{\frac{n}{2}} + d \\ &= a \cdot c \cdot 10^{\frac{n}{2}} + (a \cdot d + c \cdot b)10^{\frac{n}{2}} + b \cdot d \end{aligned} \tag{3}$$

- optimierter Divide & Conquer $\Rightarrow O(n^{ld(3)}) \sim O(n \cdot \sqrt{n})$

$$\begin{aligned} p &= a \cdot 10^{\frac{n}{2}} + b \\ p &= c \cdot 10^{\frac{n}{2}} + d \\ u &= a \cdot c \\ v &= b \cdot d \end{aligned} \tag{4}$$

$$w = (a + b)(c + d)$$

$$(a \cdot b + c \cdot d) = w - v - u \tag{5}$$

Es koennen alle 3 Additionen in $O(n)$ durchgefuehrt werden

3. Exponentiation

(a) Exponentiation $\Rightarrow \leq 2 \cdot \lfloor ld(2) \rfloor = O(n \cdot \log(n))$

$$\begin{aligned} x^2 &= x \cdot x \\ x^4 &= x^2 \cdot x^2 \\ \dots \\ x^{16} &= x^8 \cdot x^8 \\ x^{23} &= x^{16} \cdot x^4 \cdot x^2 \cdot x \end{aligned} \tag{6}$$

Listing 2: Exponentiation Code

```

1 double exp(int x, int n)
2 {
3     int e = 1;
4     while(n > 0)
5     {
6         if(n % 2 != 0)
7         {
8             e = e * x;
9             n = n >> 2;
10            x = x * x;
11        }
12    }
13    return e;
14 }
```

2 Rekursive Zeitgleichungen

4. Loesungsverfahren

(a) Substitutions-Methode

- Versuch der Abschaetzung mittels einer Bekannten Laufzeit, zeigen durch Induktion
- Zu zeigen ist, z.B.: $T(n)$ glaubt man sei $O(n) \Rightarrow \exists c > 0, n_0 \in \mathbb{N} : T(n) < c \cdot n \forall n \geq n_0$
- Bei vollstaendiger Induktion muss immer die exate, nicht die asymptotische Form beweisen werden.
- Ein Nachteil ist das die Schranke nicht scharf sein muss, d.h. oft sind mehrere Schranken zu testen.

(b) Iterative-Methode

- Versuch der Abschaetzung mittels staendigem wieder einsetzen der Bedingung bist Schranken fuer (un)endliche Summen gefunden werden
- Beispiel:

$$\begin{aligned}T(n) &= 2 \cdot T\left(\frac{n}{2}\right) + n^2 \\T\left(\frac{n}{2}\right) &= 2 \cdot T\left(\frac{n}{4}\right) + \left(\frac{n}{2}\right)^2 \\T(n) &= 2\left(2 \cdot T\left(\frac{n}{4}\right) + \frac{n^2}{2}\right) + n^2 \\T(n) &= 2^2 \cdot T\left(\frac{n}{2^2}\right) + \frac{n^2}{2^1} + n^2 \\&= \dots \\T(n) &= 2^k \cdot T\left(\frac{n}{2^k} + \sum_{i=1}^{k-1} \frac{n^2}{2^i}\right) \tag{7} \\T(1) \doteq 1 &\Rightarrow \frac{n}{2^k} = 1 \Rightarrow k = \log(n) \\T(n) &= O(n) + n^2 \cdot \sum_{i=1}^{k-1} \frac{1}{2^i} \\&\sum_{i=1}^{k-1} \frac{1}{2^i} = 2 \\T(n) &= O(n) + 2 \cdot n^2 \\T(n) &= O(n^2)\end{aligned}$$

(c) Master-Methode

- Gilt nur fuer Rekursionen der Form $T(n) = a \cdot T\left(\frac{n}{b} + f(n)\right)$, wenn gilt $a \geq 1; b > 1$

- dann gibt es drei Loesungsmoeglichkeiten

– Fall 1

$$\begin{aligned} f(n) &= O(n^{\log_b(a)-\epsilon}); \epsilon > 0 \\ \Rightarrow T(n) &= O(n^{\log_b(a)}) \end{aligned} \quad (8)$$

– Fall 2

$$\begin{aligned} f(n) &= O(n^{\log_b(a)}) \\ \Rightarrow T(n) &= O(n^{\log_b(a)} \cdot \log(n)) \end{aligned} \quad (9)$$

– Fall 3

$$\begin{aligned} f(n) &= \Omega(n^{\log_b(a+\epsilon)}); \epsilon > 0; \exists c < 1 : a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n), n \geq n_0 \\ \Rightarrow &= \theta(f(n)) \end{aligned} \quad (10)$$

Es gibt Faelle in denen die Mastermethode nicht funktioniert, z.B.: wenn $f(n)$ nicht groesser als eine Bedingung ist

3 dynamisches Programmieren

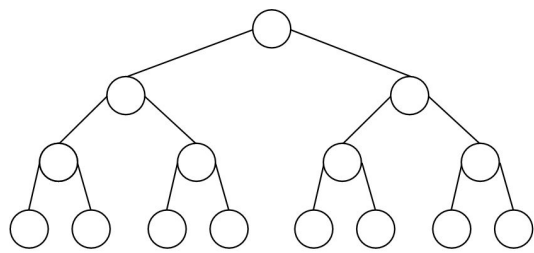
5. dynamisches Programmieren

(a) Grundlage

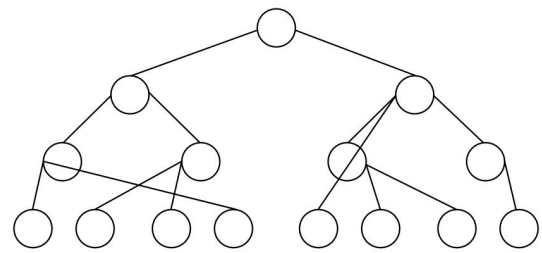
- Aehnliche und voneinander abhaengige Teilprobleme verursachen oft exponentioneller Laufzeit
- Kann man kleinere Funktionswerte zuerst berechnen und diese speichern vermeidet man doppelte Berechnungen
- dies bezeichnet man als dynamische Programmierung
- Der Nachteil ist ein hoehererer Speicherbedarf
- **Satz:** Die optimale Klammerung zur Multiplikation von n Matrizen A_1, \dots, A_n kann in $O(n^3)$ Zeit und $O(n^2)$ Speicher berechnet werden.

(b) Laengste gemeinsame Teilfolge (lgT)

- Ermittelt ob eine Folge $X(x_1, \dots, x_n)$ in einer Folge $Z(z_1, \dots, z_m)$ wobei $(n \leq m)$ enthalten ist
- dabei muessen die einzelnen Elemente von X nur in ihrer Reihenfolge in Z enthalten sein, nicht jedoch durchgehend aneinander



(a) Rekursives Programmieren



(b) dynamisches Programmieren, es werden bereits berechnete Ergebnisse wiederverwendet

Abbildung 2: rekursives vs. dynamisches Programmieren

Listing 3: laengste gemeinsame Teilfolge Code: Zeit: $O(m \cdot n)$ und Speicher: $O(m \cdot n)$

```

1  i>>int lgt(int m, int n)
2  {
3      int l[m,n];
4      //Init
5      for(int i = 0; i < m; i++)
6      {
7          l[i,0] = 0;
8      }
9      for(int j = 1; j < n; j++)
10     {
11         l[0,j] = 0;
12     }
13     //calc
14     for(int i = 1; i < m; i++)
15     {
16         for(int j = 0; j < n; j++)
17         {
18             if(x(i) == y(j))
19             {
20                 l[i,j] = l[i-1, j-1] + 1; //case 1
21             }
22             else
23             {
24                 l[i,j] = max(l[i-1,j],l[i,j-1]); //case 2
25             }
26         }
27     }
28     printf("max lenght: %d\n", l[m,n]);
29     return l[m,n];
30 }

```

4 Randomisierung

6. Randomisierter Suchbaum

(a) Balancierungsschema - Prioritaet

- Jeder Wert x erhaelt eine Prioritaet $P(x)$
- Der Baum bildet eine Halde mit: $root \geq \max \text{Sohn}_{rechts}, \text{Sohn}_{rechts}$
- **Beobachtung:** Besitzen alle Knoten verschiedene Werte und unterschiedliche Prioritaeten, ist der Baum eindeutig!
- **Beweis:** Die Wurzel ist der maximale Wert, der Baum zerfaellt in Teile j Wurzel (linker Teilbaum) und i Wurzel (rechter Teilbaum); dies wiederholt sich rekursiv fuer alle Werte

5 Schnitt von Liniensegmenten

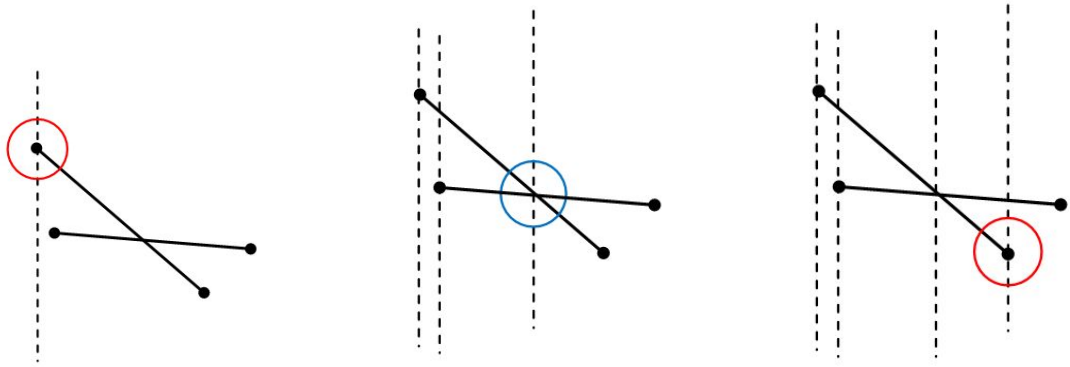
7. Plane Sweep Algorithmus

(a) Grundlagen

- Es gibt 3 Ereignisse: Beginn, Ende und Schnitt zweier Linien
- Die X-Koordinaten von Beginn und Ende jeder Linie werden sortiert gespeichert \Rightarrow Halde
- Die Y-Koordinaten werden vom aktuellen Ereignis aufgerufen, Einfuegen, Entfernen, Sweep \Rightarrow (2-4) Baum als Datenstruktur
- Berechnung der Schnittpunkte mittels Geradengleichung
- n Segmente mit k Schnitten wobei gilt: $0 \leq k \leq \binom{2n}{n} \Rightarrow$ Zeit: $O((n+k)\log(n))$ und Speicher: $O(n+k)$
- Der Algorithmus ist auch auf Kreise anwendbar, jedoch nur als Schnittdetektor geeignet

Listing 4: Planesweep Code:

```
1  i>> void planeSweep(X, Y)
2  {
3      int x = 0, y = 0;
4      while(X[i] != 0)
5      {
6          //first
7          m = min(X); //and remove m from X
```



(a) Ereignis Beginn einer Linie, das Segment wird eingefuegt

(b) Ereignis Schnittpunkt zweier linien, die benachbarten Segmente werden getauscht (switch)

(c) Ereignis Ende einer Linie, das Segment wird entfernt

Abbildung 3: Ereignisse des PlaneSweep Algorithmus

```

8      //second
9      if(m == leftEnd)
10     {
11         //add Segment to Y
12     }
13     else if (m == rightEnd)
14     {
15         //remove Segment from Y
16     }
17     else
18     {
19         //switch neighbor segment
20     }
21     //third
22     for(all new neighbors in Y)
23     {
24         if(cross p != 0 exists && p right of L)
25         {
26             printf(p);
27             //add p to X
28         }
29     }
30     i++;
31 }
32 }

```

6 konvexe Huelen KH

8. Definition

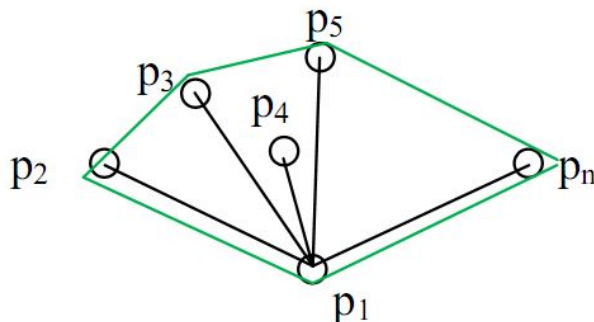
(a) Definition

- „Gummiband“ um eine Punktmenge
- Die Konvexe Huelle von S , kurz $KH(S)$, ist das kleinste konvexe Polygon das S enthaelt; $KH(S) = \bigcap_{P \supset S} P$
- **Durchmesser einer Punktmenge:** (=maximaler Abstand zweier Punkte): $d(s) = \max_{p, q \in S} \text{distanz}(p, q)$
- **Seperationsgerade:** Eine Seperationsgerade existiert genau dann wenn sich $KH(A)$ und $KH(B)$ nicht schneiden.

(b) Graham-Scan

- KH reflektiert eine syklische Orndung der extrmen Punkte \Rightarrow zyklisch sortiert, nicht extreme Punkte loeschen
- Laufzeit: $O(n \log(n))$ und Speicher $O(n)$

Abbildung 4: graham: Prinzip des Algorithmus



Listing 5: Graham-Scan Code:

```
1  i>>j void grahamScan(S)
2  {
3      //PUSH ... Put on top
4      //POP ... get from top
5      //TOP ... topmost element
6      //TOP2 ... second topmost element
7
8      sort(polar(S)); //polar coords depending to p[1]
9      PUSH(p[1], p[2], p[3]);
10     for(int i = 4; i <= n; n++)
```

```

11  {
12  while(innerAngle(TOP2, TOP, p[i]) > pi())
13  {
14      POP;
15  }
16  PUSH(p[i]);
17  }
18  }

```

(c) Iteratives Einfuegen

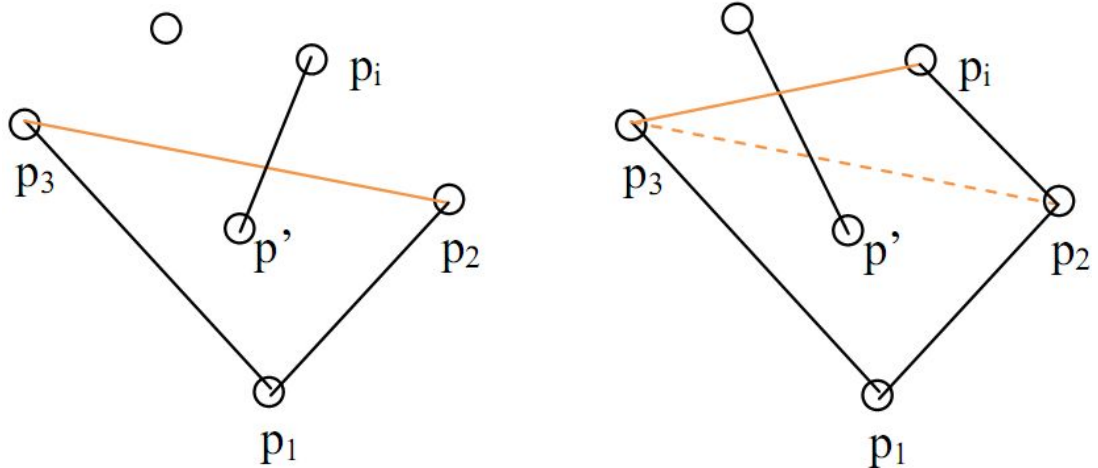
- Die KH wird Schritt fuer Schritt aufgebaut, durch hinzufuegen der unsortierten Punkte zur bestehenden KH
- Start mit einem Dreieck (p_1, p_2, p_3) ; jeder hinzugefuegte Punkt p_i ist entweder ein Teil der KH oder nicht
- Ist p_i ein Teil wird er der $KH(R)$ hinzugefuegt, eine Tangente an die bestehende $KH(R)$ gelegt und alle von p_i aus sichtbaren Kanten entfernt
- Die Entscheidung ob $p_i \in KH(R)$ liegt kann durch Schnittberechnung erfolgen, dabei wird ein Punkt p' innerhalb der bestehenden $KH(R)$ (z.B.: Schwerpunkt) gewaehlt. Fuer jeden Punkt p_k wird fuer jede bestehene Kante berechnet ob sie mit der Verbindungslinie $\overline{p'p_i}$ schneidet. Schneidet sie nicht, liegt p_i innerhalb der $KH(R) \Rightarrow p_i \notin KH(R)$, schneidet sie mit einer Kante, ist diese Zeugenkante von $p_k \Rightarrow O(n)$ Laufzeit
- wird ein neuer Punkt in die $KH(R)$ aufgenommen, werden Kanten geloescht. Jede geloeschte Kannte war ev. Zeugenkante. Fuer die betroffenen Punkte muss die Zeugeninformation erneut gerechnet werden mit den neuen Kanten von $KH(R)$
- die Wahrscheinlichkeit das sie die Zeugenkannte fuer einen Punkt aendert ist $O(\log(n))$, da dies fuer jeden Punkt waehrend des Einfuegens gerechnet werden muss ergibt sich eine Laufzeit von $O(n \log(n))$ und ein Speicherbedarf von $O(n)$.
- **Achtung:** der Algorithmus kann nicht online arbeiten, zu beginn muessen alle Punkte bekannt sein!

7 Optimale Triangulierung von Polygonen

9. Allgemein

(a) Definition

- **simples Polygon:** Von $n \geq 3$ geraden begrenzte, zusammenhaengende Menge in der Ebene
- **konvexes Polygon:** ein simples Polygon das keine einspringenden Ecken besitzt



(a) iteratives einfügen: $\overline{p'p_i}$ schneidet die Zeugenlinie $\overline{p_2p_3}$ von p_i

(b) iteratives einfügen mit Zeugenupdate: p_i wurde in $KH(R)$ aufgenommen und seine Zeugenkante gelöscht. Alle anderen Punkte die sie als Zeugenkante hatten müssen upgedated werden da sie eine neue Zeugenkante erhalten.

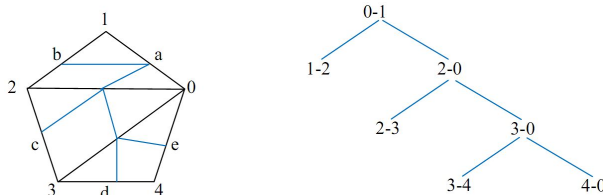
Abbildung 5: Iteratives Einfügen zur Berechnung der $KH(R)$

- **Satz:** Ist das Polygon P n -seitig, so besitzt jede Triangulierung von P genau $n - 3$ Diagonalen und $n - 2$ Dreiecke
 - eine Triangulierung einer Punktmenge S ist der maximale, planare und geradlinige Graph auf S .
 - optimale Triangulierung ist jene Triangulierung die die minimale Diagonalenlänge liefert
- (b) Anzahl Triangulierungen eines konvexen Polygons
- Wurzel = Kante e von P
 - $\{e, d_1, d_2\}$ ist ein eindeutiges Dreieck das P in zwei konvexe Polygone P_1, P_2 teilt
 - ist d_1 (bzw. d_2) eine Diagonale von P , so ist d_1 (bzw. d_2) die Wurzel des Binaerbaumes fuer P_1 (bzw. P_2)
 - ansonsten ist d_1 (bzw. d_2) eine Kante von P und ein Blatt im Binaerbaum fuer P
- (c) Algorithmus zur optimalen Triangulierung
- new item

Listing 6: optimale Triangulierung:

```
1 i>>void optTriangle(L,S,u)
```

Abbildung 6: Triangulierung: Generierung eines Binaerbaumes aus der Triangulierung; es entstehen $n-1$ Blaetter und $n-2$ innere Knoten und $n-3$ Diagonalen; es gilt $\#Baeume \geq \#Triangulierungen$; Baut man aus dem Baum die Triangulierung gild $\#Triangulierungen \geq \#Baeume$, wobei zu beachten ist das die Wurzelkante zwar kein Blatt darstellt, aber vorhanden ist!



```

2 {
3   for(int i = 1; i <= (n-1); i++)
4   {
5     L[i,i+1]=0;
6   }
7   for(int l = 2; l <= (n-1); l++)
8   {
9     for(int i = l; i <=(n-1); l++)
10    {
11      j = i+1;
12      L[i,j] = inf;
13      for(int k = (i+1); k <= (j-1); k++)
14      {
15        x = L[i,k] + L[k,j] + circumference(i,j,k);
16        if(x < L[i,j])
17        {
18          L[i,j] = x;
19          S[i,j] = k;
20        }
21      }
22    }
23  }
24 }
```

8 new chapter

10. new problem

(a) new part

- new item

9 new chapter

11. new problem

(a) new part

- new item

10 new chapter

12. new problem

(a) new part

- new item

11 bekannte Laufzeiten

Algorithmus	Laufzeit	Speicherbedarf
Scanline, siehe 1	$O(n)$	$O(n)$
Exponentiation, siehe 2	$\leq 2 \cdot \lfloor ld(2) \rfloor = O(n \cdot \log(n))$	
Binaersuche, siehe Script [1]	$T(n) = T(\frac{n}{2}) + O(1)$	
Sortierverfahren, siehe Script [1]	$O(n \cdot \log(n))$	
lgt, siehe 3	$O(n \cdot m)$	$O(n \cdot m)$
Planesweep, siehe 4	$O((n+k)\log(n))$	$O(n+k)$
Graham-Scan, siehe 5	$O((n)\log(n))$	$O(n)$
iteratives einfügen, siehe 5	$O((n)\log(n))$	$O(n)$
optimale triangulierung konvexer und beliebiger Polygone, siehe [1] , Kapitel 7.1.1	$O(n^3)$	
beliebige triangulierung konvexer und beliebiger Polygone, siehe [1]	$O((n))$	
beliebige triangulierung allgemeiner Punktmengen, siehe [1] , Kapitel 7.1.1	$O(n\log(n))$	
optimale Triangulierung konvexer Polygone, siehe 6	$O(n^3)$	$O(n^2)$
beliebige triangulierung allgemeiner Punktmengen, siehe [1] , Kapitel 7.1.1	$O(n\log(n))$	

Tabelle 1: Laufzeiten und Speicherbedarf Uebersicht diverser Algorithmen

12 diverses Bekanntest

13. diverse Formeln

(a) Catalan-Zahlen:

$$\begin{aligned}P(n) &= C(n-1) \\C(n) &= \frac{1}{n+1} \cdot \binom{2n}{n} \\ &= \theta\left(\frac{4^n}{n^{1.5}}\right)\end{aligned}\tag{11}$$

(b) Woerterbuchproblem Das Woerterbuchproblem besteht aus folgenden 3 Operationen:

- Suchen: Finde einen Eintrag
- Einfuegen: Fuege einen neuen Eintrag hinzu
- Entfernen: entferne einen Eintrag

Satz: Alle 3 Operationen koennen mit einem randomisiertem Suchbaum mit hoher Wahrscheinlichkeit in je $O(\log(n))$ Zeit durchgefuehrt werden.

13 xxx

14. xxx

(a) yyy

- zzz, see [1]

(b) aaa

- bbb

Literatur

[1] Oswin Aichholzer. Vorlesungsmitschrift aus entwurf und analyse von algorithmen. Pdf, Institut für Softwaretechnologie, Oktober 2008. [12](#), [13](#)