Datenstrukturen und effiziente Algorithmen

Markus Vieth David Klopp Christian Stricker

22. Dezember 2015

Inhaltsverzeichnis

Teil I

Sortieren

1 Vorlesung 1

1.1 Bubblesort

1.1.1 Pseudocode

```
void bubblesort (int[] a) {
  int n = a.length;
  for (int i = 1; i < n; i++) {
    for (int j = 0; j < n-i; j++) {
      if ( a[j] < a[j+1])
        swap (a, j, j+1);
    }
  }
}
```

Schleifen-Invariante: Nach dem Ablauf der i-ten Phase gilt:

Die Feldpositionen n-i,...,n-i enthalten die korrekt sortierten Feldelemente

Beweis durch Induktion nach i $\overset{i=n-1}{\Longrightarrow}$ Sortierung am Ende korrekt.

1.1.2 Laufzeitanalyse

1.	Phase	n-1
2.	Phase	n-1
3.	Phase	n-1
	$\vdots$	
i.	Phase	n-1
	$\vdots$	
(n-1).	Phase	n-1
$1+2+3+\ldots+(n+1)$		

$$T(n) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \in O(n^2)$$

n	T_{real}
2^{10}	8ms
2^{11}	11ms
2^{12}	26ms
$\vdots$	
2^{16}	5,819s
2^{17}	23,381s
$\vdots$	
2^{20}	16min
$\vdots$	
2^{26}	52d

$$T_{real}(n) \approx cn^2 \quad c \approx 10^{-6}$$

1.2 Heapsort

z.B. 21 6 4 7 12 5 3 11 14 17 19 8 9 10 42

Skizze

1.2.1 Heap-Eigenschaft

2 Vorlesung 2

Heapsort (Fortsetzung)

2.0.1 Pseudocode

```
heapify ( int[] a, int i, int n) {
  while (2i + 1 < n) {          //linkes Kind von i existiert
    int j = 2i + 1;
    if ( 2i +2 < n)             //rechtes Kind von i existiert
      if ( a[j] < a[j+1])
        j = j + 1;              //j steht für Indes des größten Kindes
    if ( a[i] > a[j])           //Vater größer als Kind
      break;                    //Abbruch, weil heap bereits erfüllt
    swap(a,i,j);                //Tausch zwischen Vater und Kind
    i = j;
  }
}
```

1. Phase: Bottom-up Strategie zum Heapaufbau

```
for ( int i = n/2; i ≥ 0; i--)
  heapify(a,i,n);
```

2. Phase: Sortierphase

```
for ( int i = n-1; i ≥ 0; i--) {
  swap(a,0,i);
  heapify(a,0,i);
}
```

2.0.2 Korrektheitsbetrachtung

Invariante beim Heapaufbau: Beim Durchlauf der for-Schleife wird die Heapeigenschaft vom unteren Baumlevel bis zur Wurzel hergestellt.

Invariante für Sortierphase: Nach jedem weiteren Durchlauf der for-Schleife findet ein weiteres Element am Feldende seinen „richtigen Platz“.

2.0.3 Laufzeitanalyse

$T(n)$ = Zahl der Elementvergleiche.

Analyse Heapaufbau:

3 Vorlesung 3

3.1 Landau-Notation

3.1.1 O

3.1.2 Ω

3.1.3 Θ

3.1.4 o

3.1.5 Notation

3.2 Mergesort (Divide and Conquer)

3.2.1 Pseudo-Code

3.2.2 Laufzeitanalyse

4 Vorlesung 4

4.1 Master-Theorem

4.1.1 Fall 1

4.1.2 Fall 2

4.1.3 Fall 3

4.1.4 Beispiel: Mergesort

4.2 Schnelle Multiplikation langer Zahlen

4.2.1 Karazuba Ofman

4.2.2 Akra-Brazzi Theorem

5 Vorlesung 5

5.1 Akra-Brazzi

5.2 Lineare Rekursionsgleichungen

5.2.1 Methode der erzeugenden Funktionen

5.2.2 Einschub: Reihenentwicklung

5.2.3 Nullstellen des Nennerpolynoms

5.2.4 Partialbruchzerlegung

5.3 Quicksort (Divide and Conquer)

6 Vorlesung 6

6.1 Quicksort

6.1.1 Pseudo-Code

6.1.2 Zufallspermutation

6.1.3 Einschub: Stochastik

6.1.4 Laufzeitanalyse

6.2 Median in Linearzeit

7 Vorlesung 7

7.1 Quicksort

7.2 Quickselect

8 Vorlesung 8

8.1 Verallgemeinerung von Akra-Brazzi

$$T_n = [\sum_{i=1}^{k} a_i T(\frac{n}{b_i})] + g(n)$$

Beispiel

$$T_n = 1 \cdot T(\frac{n}{3}) + 1 \cdot T(\frac{2n}{3}) + n$$

$$T_n = \theta(n^\alpha(1 + \int_1^n \frac{g(x)}{x^{1+\alpha}} dx))$$

Klassisch $\alpha = \log_b(a)$, $\frac{a}{b^\alpha} = 1$

Jetzt Bestimmte α so, dass gilt:

$$\sum_{i=1}^{k} \frac{a_i}{b_i^\alpha} = 1$$

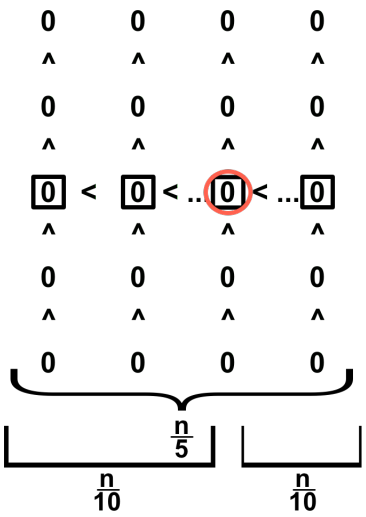
$$a_1 = a_2 = 1, \quad b_1 = 3, \quad b_2 = \frac{3}{2}, \quad g(n) = n$$

$$\frac{1}{3}^\alpha + \frac{2}{3}^\alpha \stackrel{!}{=} 1 \Rightarrow \alpha = 1$$

$$T(n) = \Theta(n(1 + \int_1^n \frac{x}{x^{1+1}} dx)) = \Theta(n \ln(n))$$

8.2 Median der Mediane

Gruppierung in 5er Päckchen



Median der Mediane

Abbildung 8.1

Wortlaut Teile die n Elemente in 5-er Gruppen. Bestimme innerhalb jeder Gruppe den Median. Bestimme nun den Median der Mediane. Wähle diesen Median als Pivot Element.

$$\exists \frac{3n}{10} \text{ Elemente } \leq p \geq \exists \frac{3n}{10} \text{ Elemente } (\pm 1 \text{ wegen p})$$

8.2.1 Deterministische Variante für k-Select

Wähle zu Beginn den Median der Mediane als Pivot Elemente. Unterteile nun die Folge anhand von p in zwei Teilfolgen und verfahre von nun an analog zur randomisierten Variante von k-Select.

8.2.2 Laufzeitanalyse für den worst-case

$$T(n) = T(\frac{n}{5}) + n + T(\frac{7n}{10})$$

$$A_1 = \frac{n}{5}, \quad A_2 = n, \quad A_3 = \frac{7n}{10}$$

A_1 = Laufzeit zur rekursiven Bestimmung des Medians der Mediane
A_2 = Laufzeit zur Aufteilung in Teilfolgen
A_3 = Laufzeit für den Aufruf von k-Select für größere Teilfolgen, die aber sicher $\leq n - \frac{3n}{10} - \frac{7n}{10}$ hat.

Wende die verallgemeinerte Form von Akra-Brazzi an:
$g(n) = n, \quad a_1 = a_2 = 1, \quad b_1 = 5, \quad b_2 = \frac{10}{7}$

Bestimme

$$\alpha = (\frac{1}{5})^\alpha + (\frac{7}{10})^\alpha = 1$$

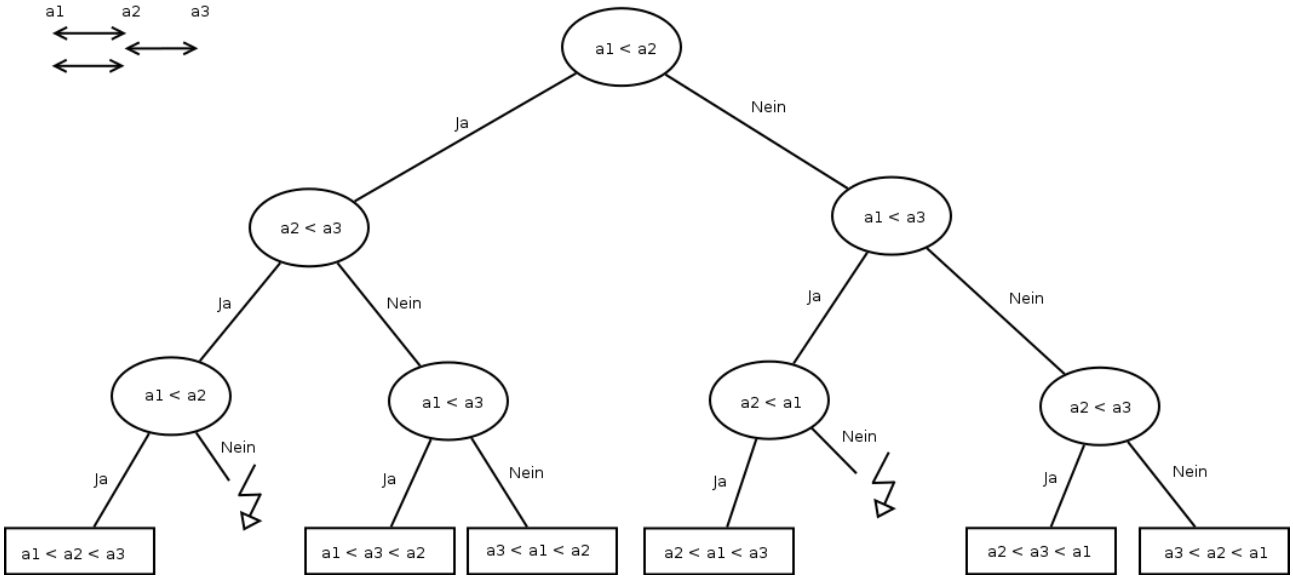
$$\Leftrightarrow (\frac{2}{10})^\alpha + (\frac{7}{10})^\alpha = 1$$

$$\Rightarrow 0 < \alpha < 1$$

$$n^\alpha(1 + \int_1^n \frac{x}{x^{1+\alpha}}dx) = n^\alpha(1 + \int_1^n x^{-\alpha}dx) = n^\alpha(1 + [\frac{1}{1-\alpha}x^{-\alpha+1}]_1^n) = n^\alpha(1 + \frac{1}{1-\alpha}(n^{-\alpha+1} - 1))$$

8.3 Untere Schranke für vergleichsbasierte Sortierverfahren

Entscheidungsproblem: (Bubbelsort)



Ein Entscheidungsbaum für einen vergleichsbasierten Sortieralgorithmus besteht aus inneren Knoten, die mit der Vergleichsoperation $a_i < a_j$ beschriftet sind, wobei sich die Indizes i, j auf die Position der Elemente in der Eingabefolge beziehen.
Die Blätter des Entscheidungsbaums sind mit den Permutationen beschriftet, die sich nach korrekter Sortierung ergeben.
Jeder korrekte Sortieralgorithmus muss zu einem Entscheidungsbaum mit mindestens $n!$ Blättern korrespondieren.

maximale Baumtiefe $\hat{=}$ maximale Anzahl durchgeführter Vergleichsoperationen

mittlere Baumtiefe $\hat{=}$ average-case Laufzeit

9 Vorlesung 9

9.1 Vergleichsbasierte Sortieralgorithmen

9.1.1 Worst-case Laufzeit

eines vergleichsbasierten Sortieralgorithmus
$\hat{=}$ maximale Tiefe des zugehörigen Entscheidungsbaums
$\hat{=}$ mittlere Tiefe der Blätter im zugehörigen Entscheidungsbaums

Sei T_{max} die maximale Baumtiefe in einem binären Baum. Betrachte nun zunächst den vollständigen binären Baum mit #Blätter ≤ 2.

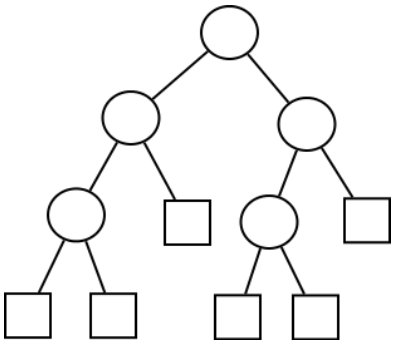


Abbildung 9.1

Untere Schranke $t_{max} \geq \log_2(n!) = \Omega(n \log n) = \log_2(n!) \leq t_{mean}$

Herleitung

$$\ln(n!) = \ln(n(n-1) \cdot (n-2) \cdot ... \cdot 2 \cdot 1) = \ln(n) + \ln(n-1) + ... + 1$$

$$= \sum_{i=1}^{n} ln(i) \geq \int_1^n \ln(x)dx = [x \ln(x) - x]_1^n = n \ln(n) - n + 1$$

$$\Rightarrow n! \geq e^{n \ln(n) - n + 1} = e \cdot e^{-n} \cdot (e^{\ln(n)})^n = e \cdot e^{-n} \cdot n = e(\frac{n}{e})^n$$

Stirling $n! \approx \sqrt{2\pi n}(\frac{n}{e})^n$

9.1.2 Lemma: Mittlere Tiefe der Blätter in einem Entscheidungsbaum $> \log_2(n)n$

Beweis Induktion nach m (Blattanzahl)

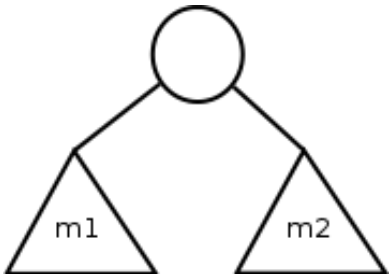


Abbildung 9.2

Untere Schranke $m_1, m_2 \hat{=}$ Blattanzahl im linken bzw. rechten Teilbaum der Wurzel

Induktions Anfang: $m = 1 \quad t_{mean} = \log_2(1) = 0$

Induktions Behauptung: $t_{mean} \geq \log_2(m)$

Induktions Schritt: Sei $m_1 < m, m_2 < m \quad (1)$ und $m_1 + m_2 = m \quad (2)$

b $\hat{=}$ Blatt im Entscheidungsbaum T_b
l $\hat{=}$ Blatt im linken Teilbaum T_l
r $\hat{=}$ Blatt im rechten Teilbaum T_r

$$t_{mean}^{links} \geq \log_2(m_1) \text{ und } t_{mean}^{rechts} \geq \log_2(m_2)$$

$$\frac{1}{m} \sum_l \cdot t_l = t_{mean}^{links} \geq \log_2(m_1)$$

Verfahre analog für rechts.

$$\sum_b T_b = \sum_l (T_l + 1) + \sum_r (T_r + 1) \geq m_1 + m_2 + m_1 \log_2(m_1) + m_2 \log_2(m_2)$$

Unter der Annahme, dass das Minimum bei $\frac{m}{2}$ liegt:

$$m_1 \log_2(m_1) + m_2 \log_2(m_2) \geq \frac{m}{2} \log_2(\frac{m}{2}) \cdot 2 = m \log_2(\frac{m}{2}) \quad \text{mit } (2)$$

Es folgt somit:

$$t_{mean} = \frac{1}{m} \sum_b T_b \geq \frac{1}{m}(m + m \log_2(\frac{m}{2})) = 1 + \log_2(\frac{m}{2}) = 1 + \log_2(m) - 1 = \log_2(m)$$

$$q.e.d$$

9.2 Radix-Sort

9.2.1 Beispiel:

10**1**	0**1**0	**1**00	001
01**0**	1**0**0	**1**01	010
00**1**	1**1**0	**0**01	011
11**1**	1**0**1	**0**10	100
10**0**	0**0**1	**1**10	101
01**1**	1**1**1	**1**11	110
11**0**	0**1**1	**0**11	111

Wichtig Beginne die Sortierung mit dem niedrigsten Bit

9.2.2 Pseudo-Code

```
void radixsort(int[] a) { // positives Element
    int n = a.length;
    int[] b0 = new int[n];
    int[] b1 = new int[n];
    int n0, n1;

    for (int i=0; i<32; i++) {
        n0 = n1 = 0;
        for (int j=0; j<n; j++) {
            if (a[j] & (1<<i)) { // i-tes Bit von a[j]
                b1[n1] = a[j];
                n1 = n1+1;
            } else {
                b0[n0] = a[j];
                n0 = n0+1;
            }
        }
    }
    for (int j=0; j<n0; j++)
        a[j] = b0[j];
    for (int j=0; j<n1; j++)
        a[n0+j] = b1[j];
}
```

9.3 Binäre Suchbäume

Zahlen 12, 8, 3, 16, 24, 17, 10, 21, 14, 9

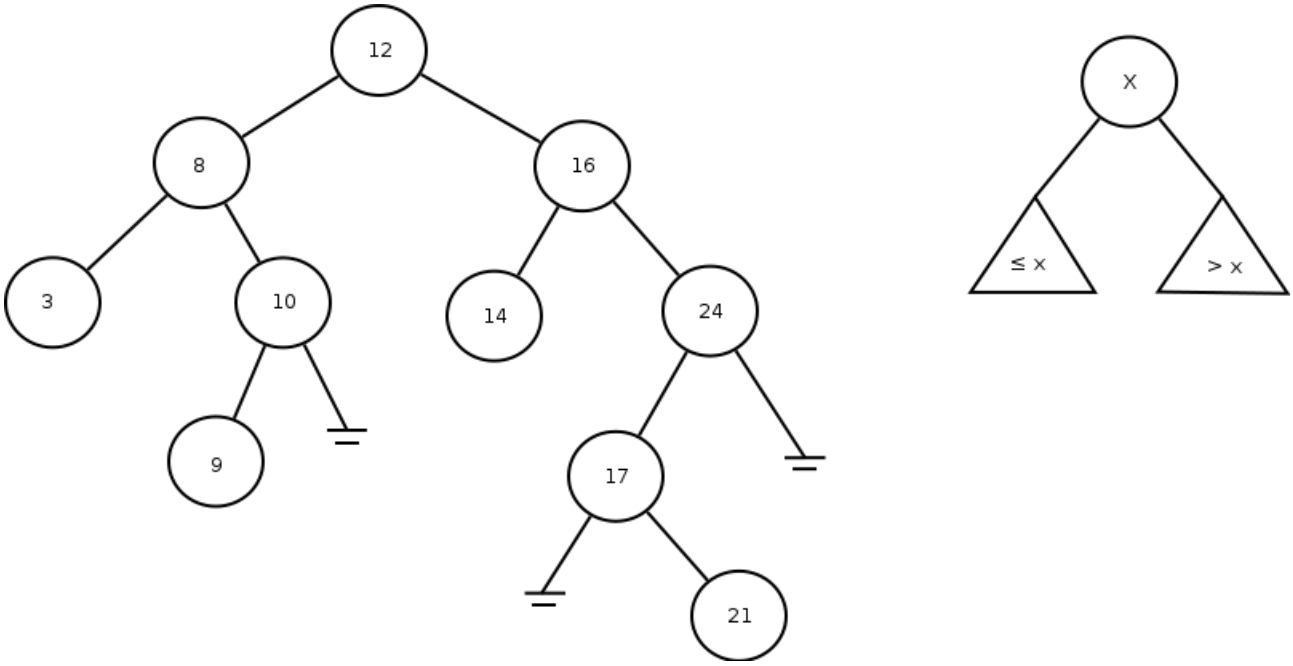


Abbildung 9.3: Knotenorientierte Speicherung

10 Vorlesung 10

10.1 Binärer Suchbaum

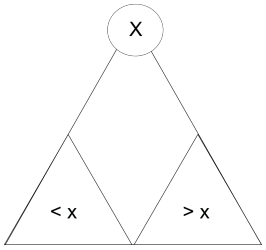


Abbildung 10.1: Binärer Suchbaum

10.2 Pseudo-Code

```
class Node {
    int key, info;              // info ist optional
    Node left, right, parent; // parent ist optional
}

int height(Node node) {
        if (node = NULL) return 0;
        return height;
}

Node insert(Node node, int x) {
    if (node == NULL)
        return new Node(x, NULL, NULL);

    if (node.key > x)
        node.left = insert(node.left, x);
    else
        node.right = insert(node.right, x);
    return node;
}

void inorder(Node node) {
    if (node == NULL) return;
    inorder(node.left) // linke Hälfte
    print(node)
    inorder(node.right) // rechte Hälfte
}
```

10.3 AVL-Bäume

Ziel Binärer Suchbaum mit garantierter Such-, Einfüge- und Löschzeit $O(\log n)$

Idee Definiere eine Balancebedingung, die dafür sorgt, dass die Baumstruktur möglichst nahe an der Idealstruktur eines vollständigen binären Baumes liegt.
Aber gleichzeitig soll es möglich sein "schnell" Strukturänderungen beim Einfügen und Löschen vorzunehmen.

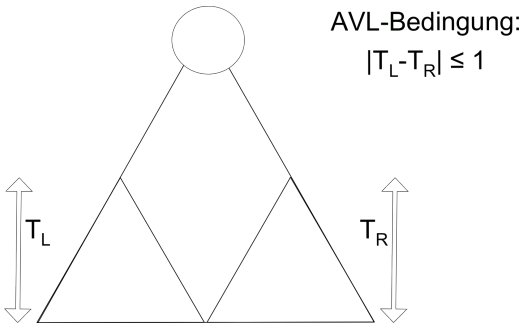
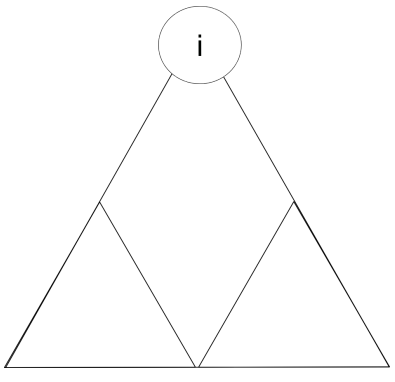


Abbildung 10.2: AVL-Baum

10.4 Laufzeitanalyse

Ziel Analyse der erwarteten maximalen Tiefe randomisierter binärer Suchbäume

Sei der Schlüssel der Wurzel das i-kleinste Element



i-1 Elemente m-i Elemente

Abbildung 10.3

$T_n \mathrel{\hat{=}}$ maximale Tiefe eines randomisierten Suchbaums mit $\{1, ..., n\}$ Elementen

Für den Fall, dass i als Wurzelknoten gewählt wird gilt:

$$T_n = max\{T_{i-1}, T_{n-i}\} + 1$$

$$X_n = 2^{T_n} \text{ exponentielle Tiefe}$$

$$2^{T_n} = 2^{1+max\{T_{i-1},T_{n-1}\}} = 2 \cdot 2^{max\{T_{i-1},T_{n-1}\}} = 2 \cdot max\{2^{T_{i-1}}, 2^{T_{n-1}}\}$$

$$\Rightarrow X_n = 2 \cdot max\{X_{i-1}, X_{n-1}\}$$

Mit der Abschätzung: $max\{2^{T_1}, 2^{T_2}\} \leq 2^{T_1} + 2^{T_2}$ **folgt:**

$$E(X_n) = E(\sum_{i=1}^{n} \frac{1}{n} \cdot 2 \cdot max\{X_{i-1}, X_{n-1}\})$$

$$= \frac{2}{n} \sum_{i=1}^{n} E(max\{X_{i-1}, X_{n-1}\}) \leq \frac{2}{n} \sum_{i=1}^{n} E(X_{i-1} + X_{n-1}) = \frac{2}{n} \sum_{i=1}^{n} [E(X_{i-1}) + E(X_{n-i})] \leq \frac{4}{n} \sum_{i=0}^{n-1} E(X_i)$$

$$n \cdot E(X_n) = 4 \cdot \sum_{i=0}^{n-1} E(X_i) \quad (1)$$

$$(n-1) \cdot E(X_{n-1}) = 4 \cdot \sum_{i=0}^{n-2} E(X_i) \quad (2)$$

$$nE(X_n) - (n-1)E(X_{n-1}) = 4E(X_n) \quad (1) - (2)$$

$$\Leftrightarrow nE(X_n) = (n+3)E(X_{n-1})$$

$$E(X_n) = \frac{n+3}{n} E(X_{n-1}) = \frac{n+3}{n} \cdot \frac{n+2}{n-1} E(X_{n-2}) = \prod_{i=0}^{n-1} \frac{n+3-i}{n-i} = \frac{n+3}{n} \cdot \frac{n+2}{n-1} \cdot \frac{n+1}{n-2} \cdot \frac{n}{n-3} \cdot \ldots \cdot \frac{6}{3} \cdot \frac{8}{2} \cdot \frac{4}{1}$$

Mit der "Jensenschen Ungleichung" folgt:

$$\sum_i Pr(T = t_i) \cdot f(t_i) \geq f\left(\sum_i Pr(T = t_i) \cdot t_i\right) = \frac{(n+3)(n+2)(n+1)}{3!} \cdot c \Rightarrow E(X_n) \in O(n^3)$$

$$X_n = 2^{T_n}, E(X_n) = E(2^{T_n})$$

$$E(f(T)) \geq f(E(T)) \Leftrightarrow fkonvex$$

$$c \cdot n^3 \geq 2^{E(T_n)}, E(T_n) \leq \log_2(c \cdot n^3) \in O(\log n)$$

11 Vorlesung 11

11.1 AVL-Bäume von Adelson-Velskii and Landis

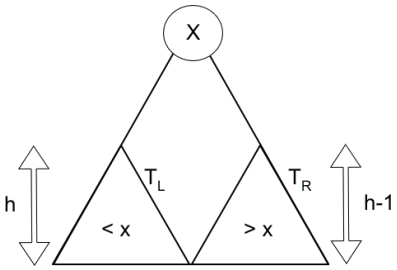


Abbildung 11.1

Ziel: Zeige, dass die maximale Tiefe eines AVL-Baums mit n Knoten ($\hat{=}$ n gespeicherten Schlüsseln) $O(\log(n))$ beträgt.

11.1.1 AVL-Eigenschaft:

$|h(T_L) - h(T_R)| \leq 1$ muss für jeden Knoten des Baums gelten. $\Rightarrow$ Suchzeit $O(\log(n))$ im worst case.

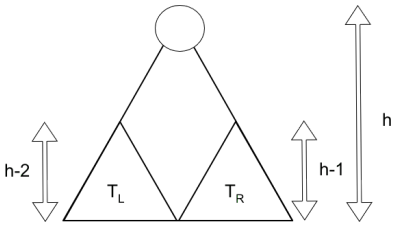


Abbildung 11.2

$n(h) =$ minimale Anzahl von Knoten in AVL-Baum der Tiefe h

$$n(h) \geq 1 + n(h-2) + n(h-1) \text{ mit } n(0) = 0 \text{ und } n(1) = 1$$

$$n \geq f(h)^{\text{I}} = \frac{1}{\sqrt{5}} \cdot (\phi^h - \phi^{-h}) \text{ mit}$$

$$\phi = \frac{1+\sqrt{5}}{2} \approx 1,61\ldots$$

$$\Rightarrow n \geq c \cdot \phi^h$$

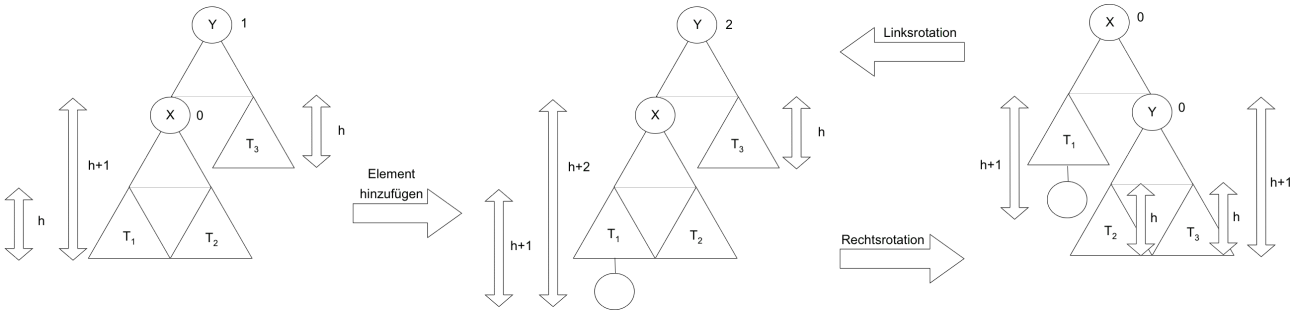
$$\Leftrightarrow h \leq \log(\frac{n}{c})$$

$$\Rightarrow h \in O(\log n)$$

q.e.d.

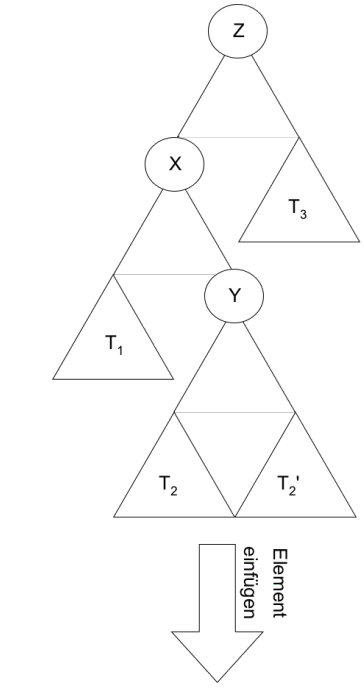
[I] $f(h)$ meint hierbei die h-te Fibonacci-Zahl

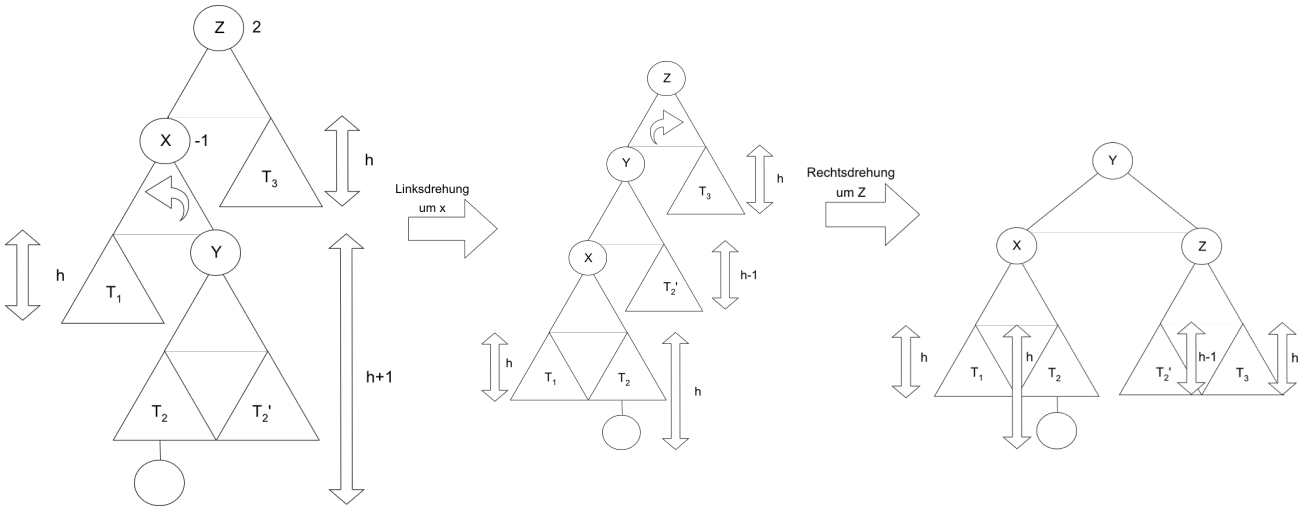
11.2 Rotationen



$Keys(T_1) < Key(X) < Keys(T_2) < Key(Y) < Keys(T_3)$
balance(Y) = height(Y.left)−height(Y.right)

$Keys(T_1) < Key(X) < Keys(T_2) < Key(Y) < Keys(T_2^{'}) < Key(Z) < Keys(T_3)$





11.3 Pseudo-Code

```
class Node {
        int key;
        Node left, right;
        int height;
}

int height(Node node) {
        if (node = null) return 0;
        return height;
}

Node rotateRight(Node y) {
        Node x = y.left;
        Node T2 = x.right;
        y.left = T2;
        T2.right = y;
        y.height = 1+max(height(y.left), height(y.right));
        x.height = 1+max(height(x.left), height(x.right));
        return x;
}

Node rotateLeft(Node y) { //analog }

Node insert(Node node, int key) {
        if (node == null) return new Node(key);
        if (key < node.key)
                node.left = insert(node.left, key);
        else
                node.right = insert(node.right, key);

        if (balance(node)>1 && key < node.left.key)
                return rotateRight(node);
        if (balance(node)<-1 && key > node.right.key)
                return rotateLeft(node);
        if (balance(node)>1 && key > node.left.key) {
                node.left = rotateLeft(node.left);
                return rotateRight(node);
        }
        if (balance(node)<-1 && key < node.right.key) {
                node.right = rotateRight(node.right);
                return rotateLeft(node);
        }
        return node;
}
```

Anmerkung: Die Laufzeit des Einfügens bleibt in $O(\text{Baumtiefe}) = O(\log n)$. Nur einer der vier Fälle ist notwendig, um die Balance herzustellen.

12 Vorlesung 12

12.1 (a,b)-Suchbäume

Blattorientierte Speicherung der Elemente
Innere Knoten haben mindestens a und höchstens b Kinder und tragen entsprechende Schlüsselwerte, um die Suche zu leiten.

Beispiel:

$$h \hat{=} \text{Tiefe} \Rightarrow \quad a^h \leq n \leq b^h \;\Rightarrow\; \log_b n \leq h \leq \log_a n$$

12.1.1 Aufspaltung bei Einfügen

12.1.2 Verschmelzen von Knoten beim Löschen

Aufspalte- und Verschmelze-Operationen können sich von der Blattebene bis zur Wurzel kaskadenartig fortpflanzen. Sie bleiben aber auf den Suchpfad begrenzt.
$\Rightarrow$ Umbaukosten sind beschränkt durch die Baumtiefe $= O(\log n)$

12.2 Amortisierte Analyse

Beispiel: Binärzähler

000		
001	Kosten(1)	$= 1$
010		$= 2$
011		$= 1$
100		$= 3$
101		$= 1$
110		$= 2$
111		$= 1$
		$\overline{11}$

Kosten der Inkrement-Operation $\hat{=}$ Zahl der Bit-Flips

Naive Analyse $2^k = n$

$$1 \cdot \frac{n}{2} + 2 \cdot \frac{n}{4} + 3 \cdot \frac{n}{8} + \ldots + k \cdot \frac{n}{2^k} = \frac{n}{2} \sum_{i=1}^{k} i(\frac{1}{2})^{i-1} = 2^{k+1} - k - 2 = 2n - k - 2$$

Von 0 bis n im Binärsystem zu zählen kostet $\leq 2n$ Bit-Flips

Sprechweise: amortisierte Kosten einer Inkrement-Operation sind 2
Folge von n-Ops kostet $2n$

12.2.1 Bankkonto-Methode

$$\text{Konto}(i+1) = \text{Konto}(i) - \text{Kosten}(i) + \text{Einzahlung}(i)$$

$$\sum_{i=1}^{n} \text{Kosten}(i) = \text{tatsächliche Gesamtkosten} = \sum_{i=1}^{n} (\text{Einzahlung}(i) + \text{Konto}(i - \text{Konto}(i+1))$$

$$= \sum_{i=1}^{n} \text{Einzahlung}(i) + \text{Konto}(1) - \text{Konto}(n+1)$$

000	
$001_{€}$	Kosten(1) $= 1$
$01_{€}0$	$= 2$
$01_{€}1_{€}$	$= 1$
$1_{€}00$	$= 3$
$1_{€}01_{€}$	$= 1$
$1_{€}1_{€}0$	$= 2$
$1_{€}1_{€}1_{€}$	$= 1$
	$\overline{11}$

Kontoführungsschema: für Binärzähler

$1€$ pro 1 in der Binärdarstellung
Jeder Übergang $1_{€} \to 0$ kann dann mit dem entsprechenden Euro Betrag auf dieser 1 bezahlt werden.
Es gibt pro Inkrement Operation nur einen $0 \to 1$ Übergang
$2€$ Einzahlung für jede Inc-Operation reichen aus um:

1. diesen $0 \to 1$ Übergang zu bezahlen
2. die neu entstandene $1_{€}$ mit einem Euro zu besparen.

$$\text{GK} = 2(2^k - 1) + 0^{\text{I}} - k^{\text{II}} = 2n - k - 2$$

[I] Zählerstand(000)

[II] Zählerstand($\overbrace{111\ldots1}^{k}$)

13 Vorlesung 13

Satz: Ausgehend von einem leeren 2-5-Baum betrachten wir die Rebalancierungskosten C (Split- und Fusionsoperationen) für eine Folge von m Einfüge- oder Löschoperationen. Dann gilt: $C \in O(m)$
d.h. Amortisierte Kosten der Split- und Fusionsopeartionen sind konstant.
! Dies bezieht sich nicht auf die Suchkosten, die in $O(\log n)$ liegen.

Beweisidee:

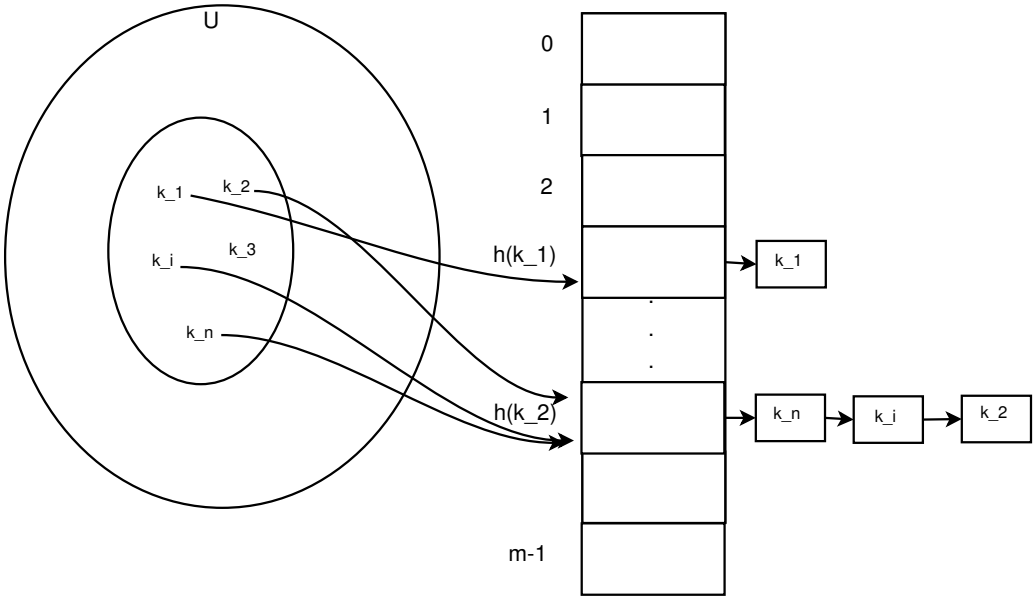
Kontoführung:

1	2	3	4	5	6
2€	1€	0€	0€	1€	2€

regelmäßige Einzahlung: 1€
Durch eine Einfüge- oder Löschoperation steigt oder fällt der Knotengrad des direkt betroffenen Knotens um höchstens 1. $\Rightarrow$ 1€ Einzahlung reicht zur Aufrechterhaltung dieses Sparplanes.
Jetzt Beseitigung der temporären 1- und 6-Knoten:
Ein 6-Knoten nutzt 1€ um seinen Split zu bezahlen. Die beiden neu entstehenden 3-Knoten benötigen kein Kapital. Der Vaterknoten des gesplitteten 6-Knotens benötigt ggf. den zweiten verfügbaren €.
Analoge Betrachtung für Fusion eines temp. 1-Knotens.

13.1 Hashing

Abbildung 13.1: Universum und Hashtabelle der Größe m



$U \subseteq \mathbb{N}$ z.B. 64-Bit-Integer
n = Zahl dr zu verwaltenden Schlüssel

$$|U| >> n$$

Hashfunktion h:

$$h : U \to [0, \ldots, m-1]$$

$$\text{z.B. } k \mapsto k \mod m$$

Einfache Annahme: (einfaches uniformes Hashing)

$$\forall\ k_i, k_j \in U : Pr(h(k_i) = h(k_j)) = \frac{1}{m}$$

Analyse der Laufzeit zum Einfügen eines neuen Elementes k

- $h(k)$ berechnen $\longrightarrow O(1)$
- Einfügen am Listenanfang in Fach $h(k)$. $\longrightarrow O(1)$

Analyse der Suchzeit für einen Schlüssel k

- $h(k) \longrightarrow O(1)$
- Listenlänge zum Fach $h(k)$ sei $n_{h(k)}$ also beim Durchlauf der kompletten Liste $\longrightarrow O(n_{h(k)})$

$$E(n_{h(k)}) = \frac{n}{m} = \alpha^{\text{I}}$$

$$\text{Suchzeit(Einfügen)} \in O(1+\alpha)$$

Laufzeit beim Löschen von Schlüssel k

- $h(k) \longrightarrow O(1)$
- Durchlaufen der Liste $\longrightarrow 0(n_{h(k)})$
- Löschen durch „Pointer-Umbiegen“ $\longrightarrow O(1)$

13.1.1 Universelles Hashing

Idee Arbeite nicht mit einer festen Hashfunktionm sondern wähle am Anfang eine zufällige Hashfunktion aus einer Klasse von Hashfunktionen aus.

z.B.

$$h_{a,b}(k) = ((a \cdot k + b) mod p) mod m$$

p sei eine hinreichend große Primzahl $\quad 0 < a < p, 0 \leq b < p$

$$\mathcal{H}_{p,m} = \{h_{a,b}(k) | 0 < a < p,\ 0 \leq b < p\}$$

$$|\mathcal{H}_{p,m}| = p(p-1)$$

Definition $\mathcal{H}$ heißt universell $\Leftrightarrow \quad \forall\ k, l \in U :\ Pr(h(k) = h(l)) \leq \frac{1}{m}$

[I] Belegungsfaktor

Suchzeit

$$\mathcal{X}_{k,l} = \begin{cases} 1 & \text{für } h(k) = h(l) \\ 0 & \text{sonst} \end{cases}$$

$$E(n_{h(k)}) = E\left(\sum_{l \in T, l \neq k}\right) = \sum_{l \in T, l \neq k} E(X_{k,l}) = \sum_{l \in T, l \neq k} Pr(h(k) = h(l)) = \sum_{l \in T, l \neq k} \frac{1}{m} = \frac{n-1}{m} = \alpha$$

14 Vorlesung 14

Universelles Hashing (Fortsetzung)

Könnte ein boshafter Mitspieler n Schlüssel bei gegebener fester Hashfunktion wählen, so würde er solche wählen, die auf den gleichen Slot unter gegebener Hashfunktion abgebildet werden. $\rightsquigarrow$ Durchschnittliche Ablaufzeit von $O(n)$

Idee zufällige Wahl der Hashfunktion aus einer Familie von Funktionen derart, dass die Wahl unabhängig von den zu speichernden Schlüssel ist (universelles Hashing).

14.0.1 Definition

Sei $\mathcal{H}$ eine endliche Menge von Hashfunktionen, welche ein gegebenes Universum U von Schlüsseln auf $\{0, \ldots, m-1\}$ abbildet. Sie heißt universell, wenn für jedes Paar von Schlüsseln $k, l \in U \quad l \neq k$ die Anzahl der Hashfunktionen $h \in \mathcal{H}$ mit $h(l) = h(k)$ höchstens $\frac{|\mathcal{H}|}{m}$. Anders: Für ein zufälliges $h \in \mathcal{H}$ beträgt die Wahrscheinlichkeit, dass zwei unterschiedliche Schlüssel k, l kollidieren nicht mehr als $\frac{1}{m}$ ist.

14.0.2 Beispiel

p Primzahl, so groß, dass alle möglichen Schlüssel $k \in U$ im $0, \ldots, p-1$ liegen. $\mathbb{Z}/p\mathbb{Z}$ bezeichnet den Restklassenring $\mod p$ (weil p prim, ist $\mathbb{Z}/p\mathbb{Z}$ ein Körper). $\mathbb{Z}/p\mathbb{Z}^*$ ist die Einheitengruppe.

Annahme: Die Menge der Schlüssel im Universum U ist größer als die Anzahl der Slots in der Hashtabelle. Für $a \in \mathbb{Z}/p\mathbb{Z}^*$ und $b \in \mathbb{Z}/p\mathbb{Z}$ betrachte:

$$h_{a,b}(k) := (a \cdot k + b \mod p) \mod m \quad (*)$$

Damit ergibt sich die Familie

$$\mathbb{Z}/p\mathbb{Z}^* = \{1, \ldots, p-1\} \quad \mathbb{Z}/p\mathbb{Z} = \{0, \ldots, p-1\} \quad \mathcal{H}_{p,m} = \{h_{a,b} | a \in \mathbb{Z}/p\mathbb{Z}^*, b \in \mathbb{Z}/p\mathbb{Z}^{(*)} \quad |\mathcal{H}| = p(p-1)\}$$

Satz Die in $(*)$ eingeführte Klasse von Hashfunktionen ist universell.

Beweis Seien k, l Schlüssel auf $\mathbb{Z}/p\mathbb{Z}$ mit $k \neq l$
Für $h_{a,b} \in \mathcal{H}_{p,m}$ betrachten wir

$$r = (a \cdot k + b) \mod p$$

$$s = (a \cdot l + b) \mod p$$

Es ist $r \neq s$
Dazu:

$$r - s = a \cdot (k - l) \mod p \quad (*2)$$

Angenommen $r - s = 0$

$$0 = a \cdot (k - l) \mod p, \text{ aber } a \in \mathbb{Z}/p\mathbb{Z}^* \Rightarrow a \neq 0 \text{ und } k \neq l \Rightarrow k - l \neq 0$$

Da p prim ist $\mathbb{Z}/p\mathbb{Z}$ ein Körper $\Rightarrow$ kein Nullteiler $\Rightarrow a \cdot (k - l) \neq 0 \Rightarrow r \neq s$
Daher bilden $h_{a,b} \in \mathcal{H}_{p,m}$ unterschiedliche Schlüssel k, l auf unterschiedliche Elemente ab. („Auf dem level $\mod p$“ gibt es keine Kollisionen).
Aus $(*2)$ folgt:

$$(r - s)(k - l)^{-1} = a \mod p$$

$$r - a \cdot k = b \mod p \text{ Bijektion zwischen (k,l) und (a,b)}$$

Daher ist die Wahrscheinlichkeit, dass zwei Schlüssel $h \neq l$ kollidieren, gerade die Wahrscheinlichkeit, dass $r \equiv s \mod m$, falls $r \neq S$ zufällig gewählt (aus $\mathbb{Z}/p\mathbb{Z}$).
Für gegebenes r gibt es unter den übrigen $p-1$ Werten für s höchstens $\lceil \frac{p-1}{m} \rceil \leq \lceil \frac{p}{m} \rceil - 1$ Möglichkeiten, sodass $s \neq r \mod p$ aber $r = s \mod m$

14.0.3 Abschätzung nach oben

$$\lceil \frac{p}{m} \rceil - 1 \leq \frac{(p + m - 1)}{m} - 1 = \frac{p - 1}{m} \text{ Kollisionsmöglichkeiten}$$

Die Wahrscheinlichkeit, dass r und s kollidieren $\mod m$ Kollisionsmöglichkeiten / Gesamtzahl der Werte

$$= \frac{p - 1}{m} \cdot \frac{1}{p - 1} = \frac{1}{m}$$

$\Rightarrow$ Für ein Paar von Schlüsseln $k, l \in \mathbb{Z}/p\mathbb{Z}$ mit $k \neq l$

$$P[h_{a,b}(k) = h_{a,b}(l)] \leq \frac{1}{m} \Rightarrow \mathcal{H}_{p,m} \text{ universell!}$$

14.1 Perfektes Hashing

Wichtig Menge der Schlüssel ist im Vorhinein bekannt und ändert sich nicht mehr.

Beispiele reserved words bei Programmiersprachen, Dateinamen auf einer CD

14.1.1 Definition

Eine Hashmethode heißt perfektes Hashing, falls $O(1)$ Speicherzugriffe benötigt werden, um die Suche nach einem Element durchzuführen.

Idee Zweistufiges Hashing mit universellen Hashfunktionen.

1. Schritt n Schlüssel, m Slots durch Verwendung der Hashfunktion h, welche aus einer Familie universeller Hashfunktionen stammt.

2. Schritt Statt einer Linkedlist im Slot anzulegen, benutzen wir eine kleine zweite Hashtabelle S_j mit Hashfunktion h_j

Bild Schlüssel $k = \{10, 22, 37, 49, 52, 60, 72, 75\}$
Äußere Hashfunktion $h(k) = ((a \cdot b) \mod p) \mod m$

$$a = 3, \quad b = 42, \quad p = 101, \quad m = 9$$

$$h(10) = \underbrace{(3 \cdot 10 + 42 \mod 101)}_{=72} \mod 9 = 0$$

Um zu garantieren, dass keine Kollision auf der zweiten Ebene auftreten, lassen wir die Größe von S_j

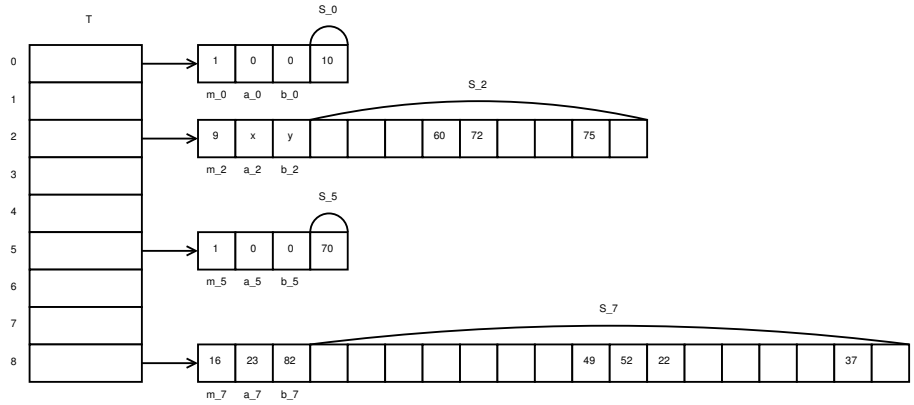


Abbildung 14.1: Perfekte Hashtabelle

gerade n_j^2 sein ($n_j \neq$ #Schlüssel$\mapsto j$Slot).
Wir verwenden für die Hashfunktion der ersten Ebene eine Funktion aus $\mathcal{H}_{p,m}$. Schlüssel die im j-ten Slot werden in der sekundären Hashtabelle S_j der Größe m_j mittels h_j gehasht. $h_j \in \mathcal{H}_{p,m}$

Wir zeigen: 2 Dinge:

1. Wie versichern wir, dass die zweite Hashfunktion keine Kollision hat.
2. Der erwartete Speicherbedarf ist $O(n)$

zu 1.

Satz Beim Speichern von n Schlüsseln in einer Hashtabelle der Größe $m = n^2$ ist die Wahrscheinlichkeit, dass eine Kollision auftritt $< \frac{1}{2}$

Beweis: Es gibt $\binom{n}{2}$ mögliche Paare, die kollidieren können. Jedes kollidiert mit der Wahrscheinlichkeit $\leq \frac{1}{m}$, falls $h \in \mathcal{H}$ zufällig gewählt wurde.
Sei X eine zufallsvariable(ZV), X zählt Kollisionen:
Für $m = n^2$ ist die erwartete Zahl der Kollisionen:

$$E[X] = \binom{n}{2} \cdot \frac{1}{m} = \binom{n}{2} \cdot \frac{1}{n^2} = \frac{n!}{2!(n-2)!n^2} = \frac{(n-1)}{2n} \leq \frac{1}{2}$$

Anwenden der Markow-Ungleichung (a=1):

$$P[X \geq 1] \leq \frac{E[X]}{1} = \frac{1}{2} \Rightarrow \text{ Wahrscheinlichkeit für irgendeine Kollision ist } < \frac{1}{2}$$

q.e.d

14.1.2 Nachteil

Für große n ist $m = n^2$ nicht haltbar!

zu 2. Wenn die Größe der primären Hashtabelle $m = n$ ist, dann ist der Platzverbrauch in $O(n) \curvearrowright$ Betrachte Platzverbrauch der sekundären Hashtabellen.

Satz Angenommen wir wollen n Schlüssel in einer Hashtabelle der Größe $m = n$ mit Hashfunktion $h \in \mathcal{H}$. Dann gilt:

$$E\left[\sum_{j=0}^{m-1} n_j^2\right] < 2n$$

Beweis

Betrachte

$$a^2 = a + 2 \cdot \binom{a}{n} = a + 2 \cdot \frac{a^2 - a}{2} \quad (*3)$$

Betrachte

$$E\left[\sum_{j=0}^{m-1} n_j^2\right] \overset{(*3)}{=} E\left[\sum_{j=0}^{m-1} \left(n_j + 2\binom{n_j}{2}\right)\right]$$

$$\overset{lini.desEW}{=} E\left[\underbrace{\sum_{j=0}^{m-1} n_j}_{=n}\right] + 2E\left[\sum_{j=0}^{m-1} \binom{n_j}{2}\right] = n + 2E\left[\sum_{j=0}^{m-1} \binom{n_j}{2}\right] \#\text{ der Kollisionen}$$

Da unsere Hashfunktion universell ist, ist die erwartete Zahl dieser Paare:

$$\binom{n}{2}\frac{1}{m} = \frac{n(n-1)}{2m} = \frac{n-1}{2}, \text{ da } m = n$$

Somit

$$E\left[\sum_{j=0}^{m-1} n_j^2\right] \leq n + 2\frac{n-1}{2} = 2n - 1 < 2n$$

Korollar Speichern wir n Schlüssel in einer Hashtabelle der Größe $m = n$ mit einer zufälligen universellen Hashfunktion und setzen die Größe der Hashtabellen der zweiten Ebene auf $m_j = n_j^2$ für $j = 0, m = 1$, so ist der Platzverbrauch des perfekten Hashings weniger als $2n$. Die Wahrscheinlichkeit, dass der Platzverbrauch der zweiten Hashtabellen $\geq 4n$ ist, ist $\leq \frac{1}{2}$ ohne Beweis.

15 Vorlesung 15

Bei n Elementen sollte die Hashtabelle $m = n^2$ groß sein.
Für die universellen Hashfunktionen

$$\mathcal{H}_{p,m} = \{h_{a,b}(k) = (a \cdot k + b) \mod p \mod m | 0 < a < p,\ 0 \leq b < p\}$$

$\binom{n}{1}$ Schlüsselpaare (k, l) mit $k \neq l$

$$E(\#\text{Kollisionen}) \leq \binom{n}{2} \cdot \frac{1}{m}^{\text{I}} = \frac{n(n-1)}{2} \cdot \frac{1}{n^2} \leq \frac{1}{2}$$

Idee Zweistufiges Verfahren:

- primäre Hashfunktion für Tabelle der Größe $m = n$

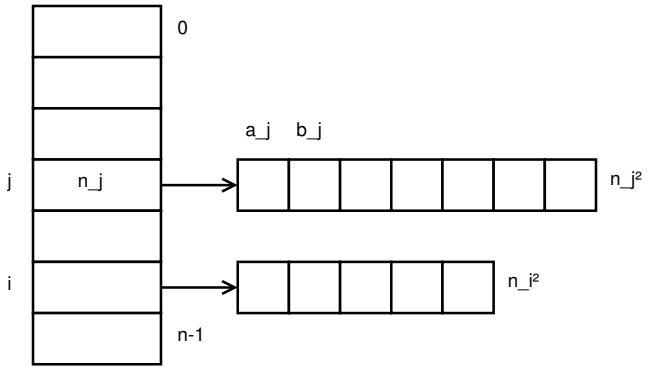


Abbildung 15.1: Perfektes Hashing

[I] Universalität von $\mathcal{H}$

Teil II

Graphen-Algorithmen

15.0.1 Einführung

$$G = (V, E) \quad V \text{ vertices, } E \text{ edges} \quad E \subseteq V \times V$$

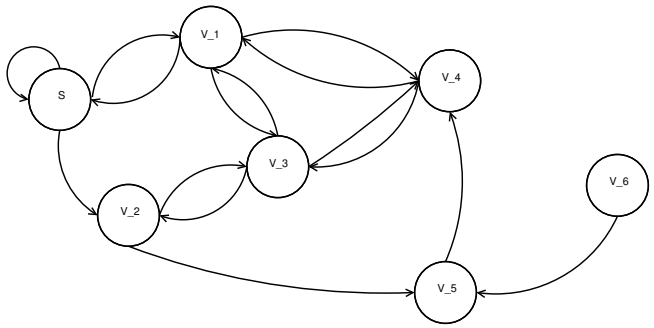


Abbildung 15.2: Gerichteter Graph

Planare Graphen können ohne Überkreuzung der Kanten in die Ebene eingebettet werden.

Eulerische Polyederformel

$$|V| + |F| = |E| + 2$$

$$8 + 6 = 12 + 2$$

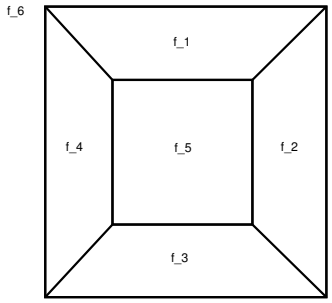


Abbildung 15.3: Würfel

Es gilt:

$$2 \cdot |E| \geq 3 \cdot |F|$$

$$\#\text{gerichtete Kanten} = 2 \cdot |E| = \sum_{i=1}^{|F|} \#\text{Kanten}(f_i)^{\text{II}} \geq 3 \cdot |F|$$

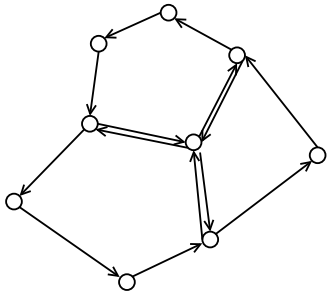


Abbildung 15.4: Placeholder

$$|F| \leq \frac{2}{3}|E|, \quad |V| + |F| = |E| + 2 \leq |V| + \frac{2}{3}|E| \Rightarrow \frac{1}{3}|E| + 2 \leq |V|$$

$$\boxed{\Rightarrow |E| \leq 3 \cdot |V| - 6}$$

[II] Jedes f_i hat mindestens 3 Kanten

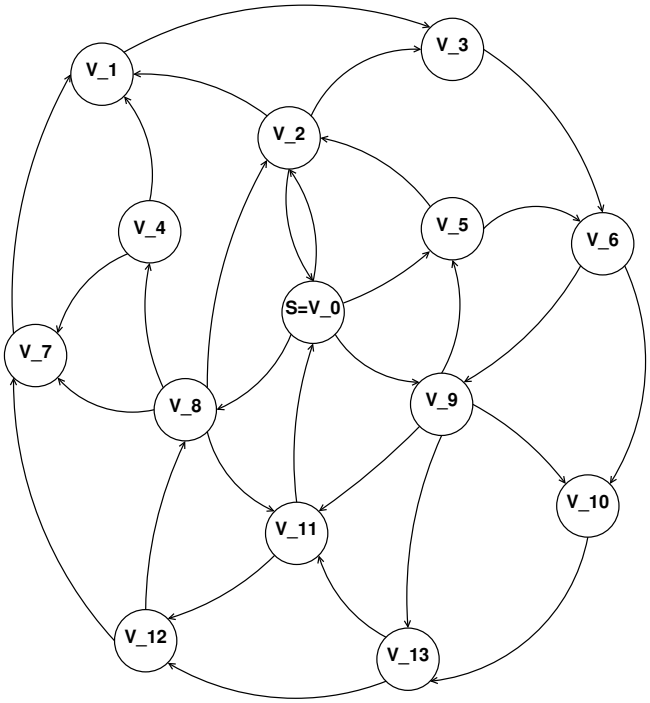


Abbildung 15.5: Beispiel

Adjazenzmatrix

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	1	0	1	0	0	1	0	0	1	1	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
2	1	1	1	1	0	0	0	0	0	0	0	0	0	0
3	0	0	0	1	0	0	1	0	0	0	0	0	0	0
4	0	1	0	0	1	0	0	1	0	0	0	0	0	0
5	0	0	1	0	0	1	1	0	0	0	0	0	0	0
6	0	0	0	0	0	0	1	0	0	1	1	0	0	0
7	0	1	0	0	0	0	0	1	0	0	0	0	0	0
8	0	0	1	0	1	0	0	1	1	0	0	0	0	0
9	0	0	0	0	0	1	0	0	0	1	1	1	0	1
10	0	0	0	0	0	0	0	0	0	0	1	0	0	1
11	1	0	0	0	0	0	0	0	0	0	0	1	1	0
12	0	0	0	0	0	0	0	1	1	0	0	0	1	0
13	0	0	0	0	0	0	0	0	0	0	0	1	1	1

$= A$

$$a \in B^{|V| \times |V|}$$

falls G ungerichtet $\Rightarrow A = A^T$

Adjazenzlisten Repräsentation

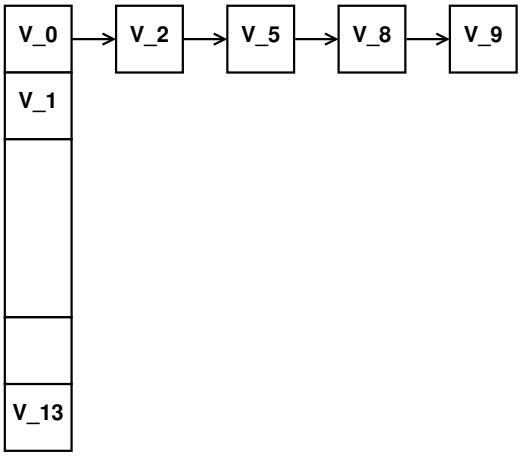


Abbildung 15.6: Adjazenzliste

Platzbedarf

$$\mathcal{O}(|V| + |E|) = \mathcal{O}\left(|V| + \sum_{i=0}^{|V|-1} \text{outdeg}(v_i)\right)$$

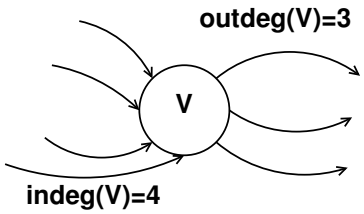


Abbildung 15.7: indeg und outdeg

15.0.2 BFS (Breadth-First Search) Breitensuche

```
forall ( v in V\{S}) {
  col[v]=white;     // Farbe  weiß = unbekannt, grau = bekannt, schwarz = vollkomm
  d[v] = infinity; // Distanz
  pi[v] = NULL;     // pi ist Vorgänger
}
col[s] = grey;      // s ist Startknoten
d[s] = 0;
pi[s] = null;
```

Queue	vs	Stack
Schlange		Stapel
empty()		”
push()		”
pop()		
FIFO		FILO
First-In-First-Out		First-In-First-Out

```
Queue Q;
Q.push(s);
while(!Q.empty()) {
  u = Q.pop();
  forall( (u,v) in E) {
    if (col[v] == white) {
      col[v] == grey;
      d[v] = d[u]+1;
      pi[v] = u;
      Q.push(v);
    }
  }
  col[u] = black;
}
```

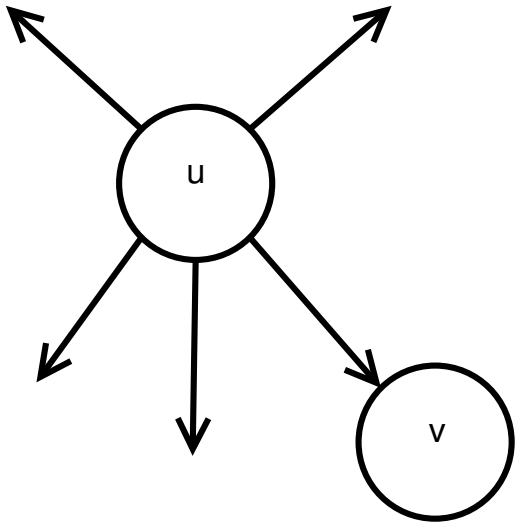


Abbildung 15.8: Grafik zum Beispielcode

Laufzeit

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

Begründung: Jeder von s aus erreichbare Knoten wird nur einmal in die Queue aufgenommen und auch ihr entfernt. Für jeden Knoten muss nur einmal seine Adjazenzliste durchlaufen werden.

$$\Rightarrow \mathcal{O}\left(|V| + \sum_{v \in V} \text{outdeg}(v)\right)$$

16 Vorlesung 16

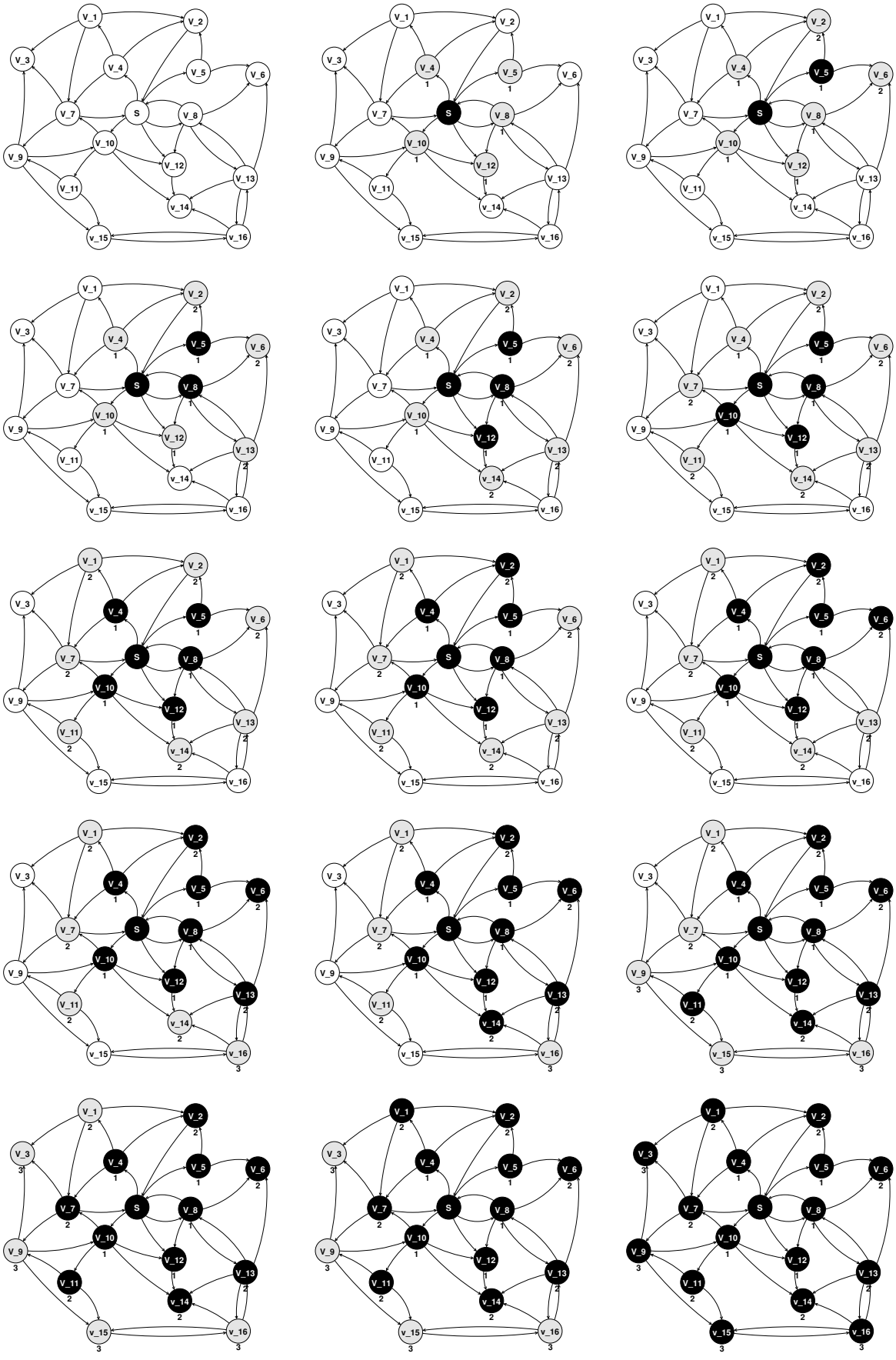


Abbildung 16.1: Beispiel

Definition: Länge kürzesten Weges

$\delta(s, v)$ = Länge eines kürzesten Weges vom Startknoten s zum Knoten v.
Setze $\delta(s, v) = \infty$, falls v nicht erreichbar von s aus.

Satz: Richtigkeit des Algorithmus

Nach Ablauf von BFS[I] gilt

$$\forall v \in V : \ d[v] = \delta(s, v)$$

Lemma 1: Dreiecksungleichung für kürzeste Wege

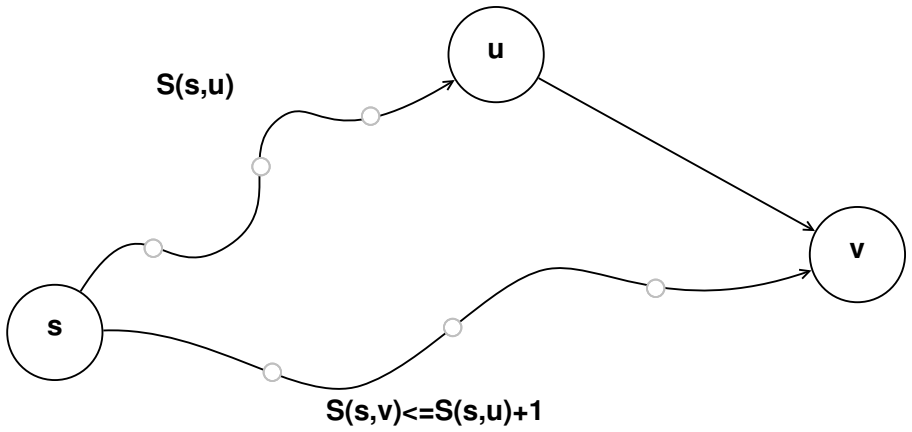


Abbildung 16.2

Lemma 2

Zu jedem Zeitpunkt im Verlauf von BFS gilt:

$$\forall v \in V : \ d[v] \geq \delta(s, v)$$

Beweis (induktiv über Zahl der Operationen, die d-Wert verändern)

Induktions-Anfang

$$d[s] = 0 \surd$$

Induktions-Schritt Knoten v wird von u aus neu entdeckt

$$d[u] \geq \delta(s, u)$$

$$d[v] = d[u] + 1 \geq \delta(s, u) + 1 \overset{D.U.}{\geq} \delta(s, v)$$

Lemma 3

Sei $Q = (v_1, v_2, \ldots, v_k)$ eine Queue, dann gilt stets:

$$d[v_1] \leq d[v_2] \leq \ldots \leq d[v_k] \leq d[v_1] + 1$$

[I] Breitensuche

Beweis (induktiv über die Zahl der push- und pop-Operationen)

Induktions-Anfang

$$d[s] = 0 \surd$$

Induktions-Schritt

pop

$$\cancel{d[v_1]} \leq d[v_2] \leq \ldots \leq d[v_k] \leq d[v_1] + 1 \overset{!}{\leq} d[v_2] + 1$$

push

$$d[u] = d[v_1] \leq d[v_2] \leq \ldots \leq d[v_k] \leq d[u] + 1$$

Beachte Kante (u, v) v ist weiß

$$v = v_{k+1} \text{ wird gepushed}$$

$$d[v_{k+1}] = d[v_1] + 1$$

Zustand von Q nach push

$$d[v_2 \leq d[v_3] \leq \ldots \leq d[v_k] \leq d[v_1] + 1 = d[v_{k+1}] \quad \surd$$

Satz: Richtigkeit des Algorithmus

Nach Ablauf von BFS[II] gilt

$$\forall v \in V : \; d[v] = \delta(s, v)$$

Beweis durch Widerspruch

Sei $v \in V$, so dass $d[v] \neq \delta(s, v)$ am Ende des Algorithmus $\overset{Lemma2}{\Longrightarrow} d[v] > \delta(s, V)$
Sei v so gewählt, dass es der erste knoten ist mit der Eigenschaft, dass sein d-Wert flasch gesetzt wird. d.h. Alle d-Werte bis zu diesem Zeitpunkt sind korrekt.
Sei $s \mapsto u' \to v$ ein kürzester Weg s ui v
Betrachte die Situation bei Bearbeitung von u':

1. Fall v ist in diesem Moment schwarz.

$$d[v] > \delta(s, v) = \delta(s; u') + 1 \geq {}^{\text{III}} d[v] \lightning$$

2. Fall v ist in diesem Moment weiß.

$$d[v] > \delta(s, u') + 1 = d[u'] + 1 = {}^{\text{IV}} d[v] \lightning$$

[II] Breitensuche
[III] v vor u' aus Q entfernt und Lemma 3.
[IV] wegen Wahl von v; d-Wert von u' muss also korrekt sein

3. Fall v ist grau.

$$d[v] > \delta(s,u') + 1 = d[u'] + 1 \geq d[u] + 1 = d[v]$$

$d[u] \leq d[u']$, weil u vor u' aus Q entfernt $\lightning$

q.e.d.

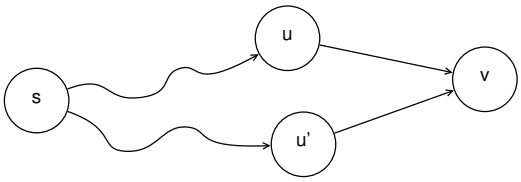


Abbildung 16.3

16.1 Kürzeste Wege Algorithmen

16.1.1 Dijkstra-Algorithmus

$$G = (V, E) \quad w : E \to \mathbb{R}_0^+$$

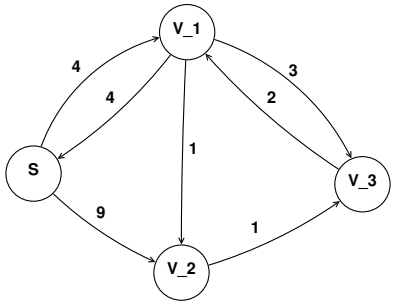


Abbildung 16.4

Sei $p = (s = v_0, v_1, v_2, \ldots, v_k)$

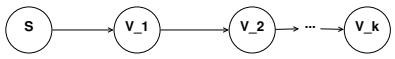


Abbildung 16.5

$$w(p) = \sum_{i=0}^{k-1} w(v_i, v_{i+1}) = \delta(s, v_k)$$

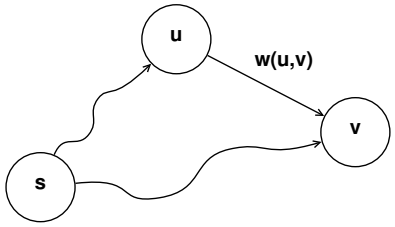


Abbildung 16.6

$$\delta(s,v) \leq \delta(s,U) + w(u,v)$$

```
relax(u,v,w) {
  if(d[v] > d[u]+w(u,v) ) {
    d[v] = d[u] + w(u,v);
    Π[v] = u;
  }
}
```

Betrachte Algorithmen zur kürzesten Wege Berechnung, die Distanzwerte nur mit Hilfe dieser relax-Funktion verändern, dann gilt:

$$d[v] \geq \delta(s, v) \quad \forall v \in V$$

Beweis

$$d[v] = d[u] + w(u, v) \overset{I.A.}{\geq} \delta(s, u) + w(u, v) \geq \delta(s, v)$$

Induktion über Zahl der reflex-Aufrufe

17 Vorlesung 17

Dijkstra Algorithmus (Fortsetzung)

$$G = (V, E) \quad w : E \to \mathbb{R}^{\geq 0}$$

```
forall (v ∈ V) {
  d[v] = ∞;
  Π[v] = NULL;
}
d[s] = 0;
S = ∅;
PriorityQueue PQ;
forall (v ∈ V)
  PQ.insert((d[v],v));
while(!PQ.empty()) {
  u = PQ.deleteMin();
  forall( (u,v) ∈ E) {
    if ( d[v] > d[u] + w(u,v)) {
      d[v] = d[u] + w(u,v);
      Π[v] = u;
      PQ.decreaseKey((d[v],v));
    }
  }
  S = S∪{u};
}
```

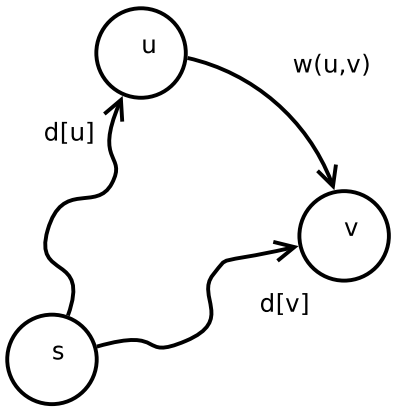


Abbildung 17.1

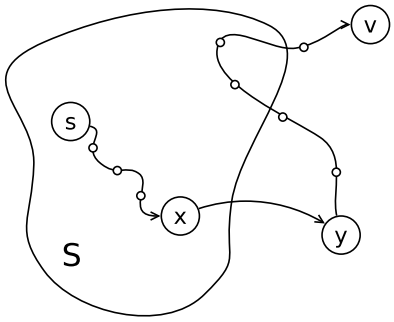
Satz: Der Dijkstra Algorithmus berechnet alle d-Werte, so dass nach Ablauf des Algo $\forall\ v \in V$ gilt: $d[v] = \delta(s, v)$.

Beweis:

Annahme:

$$\exists v \in V : \; d[v] \neq \delta(s,v)$$

$$\overset{LemmaRelax}{\Longrightarrow} \; d[v] > \delta(s,v)$$



Sei v so gewählt, das v der erste Knoten mit der Eigenschaft ist, der mit deleteMin der PQ entnommen wird und nach Relaxation aller von ihm ausgehenden Kanten der Menge S hinzugefügt wird.
Betrachte einen kürzesten Weg $s \rightsquigarrow v$

$$d[v] > \delta(s,v) \geq {}^{\text{I}}\delta(s,y) = d[y] = {}^{\text{II}}d[x] + w(x,y) = d[y] \geq {}^{\text{III}}d[v] \quad \lightning$$

Abbildung 17.2: Skizze

17.0.1 Vorläufige Laufzeitanalyse von Dijkstra

PQ.insert	x	$\lvert V\rvert$
PQ.empty	x	$\lvert V\rvert$
PQ.deleteMin	x	$\lvert V\rvert$
PQ.decreaseKey	x	$\lvert E\rvert$

Mit balanciertem Suchbaum oder mit binärem Heap (siehe Heapsort) können diese Opeartionen alle in Zeit $\mathcal{O}(\log |V|)$ realisiert werden.
$\Rightarrow$ Gesamtlaufzeit: $\mathcal{O}((|V| + |E|) \log |V|)$
Wir werden später zeigen, dass Laufzeit $\mathcal{O}(|V| \log |V| + |E|)$ möglich ist.

17.1 Bellman-Ford-Algorithmus

$$G = (V, E) \quad w : \; E \to \mathbb{R}$$

Voraussetzung G enthält keine negativen Zyklen

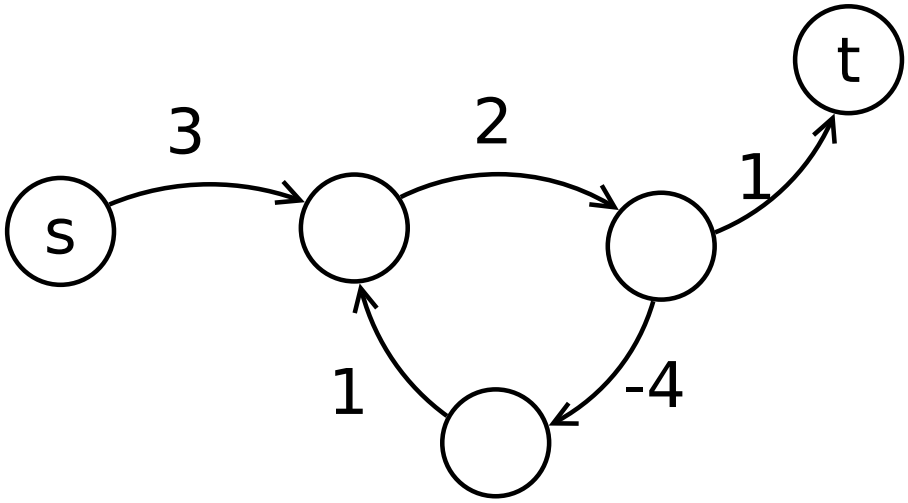


Abbildung 17.3: Ein verbotener, negativer Zyklus

[I] weil Kantengewichte nicht negativ sein dürfen
[II] x wurde schon zu S hinzugefügt, hat also Korrekten d-Wert $d[x] = \delta(s,x)$
[III] weil v vor y aus der PQ entnommen wird.

17.1.1 Pseudocode

```
forall(v ∈ V) {
  d[v] = ∞;
  Π[v] = NULL;
}
d[s] = 0;
for(i = 1; i < |V|; i++ )
  forall((u,v) ∈ E)
    if( d[v] > d[u] + w(u,v)) {
      d[v] = d[u] + w(u,v);
      Π[v] = u;
    }
```

17.1.2 Laufzeit: Bellman-Ford

$$\mathcal{O}(|V| \cdot |E|)$$

17.1.3 Korrektheitsbeweis: Bellman-Ford

Invariante: Nach den i-ten Schleifendurchlauf sind alle Kürzesten Wege korrekt berechnet, die $\leq i$ Kanten benutzen.

Beweis: Induktion über i

Induktionsanfang

$i = 0$ $d[s] = 0 = \delta(s, s)$, da keine negativen Zyklen vorliegen.

17.1.4 Induktionsschritt: $i \to i + 1$

Betrachte kürzesten Weg mit $i + 1$ Kanten:

$$s = v_0 \to v_1 \to v_2 \to \ldots \to v_i \to v_{i+1}$$

Aufgrund der Induktionsannahme[IV] gilt: $d[v_i] = \delta(s, v_i)$, weil $s = v_0 \to v_1 \to \ldots \to v_i$ ein kürzester Weg $s \rightsquigarrow v_i$ mit i Kanten ist. Da alle Kanten in der inneren Schleife einmal relaxiert werden, trifft dies insbesondere auf die Kante (v_i, v_{i+1}) zu:

$$d[v_{i+1}] = d[v_i] + w(v_i, v_{i+1}) = \delta(s, v_i) + w(v_i, v_{i+1}) = \delta(s, v_{i+1})$$

Frage: Warum folgt aus der Gültigkeit dieser Invariante die Korrektheit des Algo?

Antwort Alle kürzesten Wege benutzen höchstens $|V| - 1$ Kanten, ansonsten hätten sie einen Zyklus mit Gewicht ≥ 0, den man auch weglassen kann.

```
//Erkennung der Existenz negativer Zyklen
forall((u,v) ∈ E)
  if(d[v] > d[u] + w(u,v))
    negativer Zyklus
```

[IV] Die Invariante

18 Vorlesung 18

18.1 All-Pairs-Shortest Path Algorithmen

Distanzmatrix D für einen Graphen $G = (V, E)$ $V = v_1, v_2, \ldots, v_n$, $w : E \mapsto \mathbb{R}$

$$d_{ij} = \begin{cases} 0 & \text{für } i = j \\ w(v_i, v_j) & \text{für } (v_i, v_j) \in E \\ \infty & \text{sonst} \end{cases}$$

$$D = (d_{ij})_{\substack{i=1,\ldots,n \\ j=1,\ldots,n}} \in \mathbb{R}^{n \times n}$$

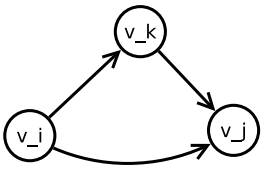


Abbildung 18.1: Grafik

$$d_{ij}^{(2)} = \min(d_i^{(1)} j,\ \min_{k=1,\ldots,n}(d_{ik}^{(1)} + d_k^{(1)} j))$$

$$D^{(2)} = D^{(1)} \circ D^{(1)} \qquad = \min(d_{ik}^{(1)} + d_k^{(1)} j)$$

Vergleich zu Matrixmultiplikation

$$C = A \circ B \quad \text{, mit } A, B \in \mathbb{R}^{n \times n}$$

$$C_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

im Ring $(\mathbb{R}, +, \cdot)$

$$C_{ij} = \min_{k=1,\ldots,n}(\quad A_{ik} + B_{kj})$$

Kommutativgesetz

$$\min(\min(a, b), c) = \min(a, b, c)$$

im „Ring“[I] $(\mathbb{R}, \min, +)$

Distributivgesetz

$$a + \min(b, c) = \min(a + b, a + c)$$

[I] der keiner ist

Assoziativgesetz

$$A \circ (B \circ C) = (A \circ B) \circ C$$

Ziel: $D^{(n)\text{II}} = D^{(1)} \circ D^{(1)} \circ \ldots \circ D^{(1)}$

Es gilt: $D^{(n)} = D^{(n+m)}$ für $m \geq 1$

18.1.1 Laufzeit zur Berechnung von $D^{(n)}$

Naiv: $\mathcal{O}(n^4)$

$$D^{(2)} = D^{(1)} \circ D^{(1)}$$

$$D^{(4)} = D^{(2)} \circ D^{(2)}$$

$$D^{(8)} = D^{(4)} \circ D^{(4)}$$

$$\vdots$$

$$D^{(2^i)} = D^{(2^{i-1})} \circ D^{(2^{i-1})}$$

Schrittzahl i so wählen, dass $2^i \geq n$

sukzessives Quadrieren: $\mathcal{O}(n^3 \log n)$

18.2 Floyd-Warshall-Algorithmus

```
for ( k = 1; k ≤ n; k++)
  for( i = 1; i ≤ n; i++)
    for ( j = 1; j ≤ n; j++)
      d[i][j] = min(d[i][j], d[i][k]+d[k][j])
```

Laufzeit $\mathcal{O}(n^3)$

18.2.1 Korrektheitsbeweis:

Invariante Nach dem k-ten Schleifendurchlauf entspricht d_{ij} der Weglänge eines kürzesten Weges p von v_i nach v_j, wobei nur Zwischenknoten erlaubt sind, mit Index $\leq k$

$$p : v_i \to v_{l_1} \to v_{l_2} \to \ldots \to v_{l_m} \to v_j$$

d.h. $1 \leq l_1, l_2, \ldots, l_m \leq k$

[II] In der Potenz stehen die Anzahl der betrachteten Kanten. n entspricht allen Kanten

18.2.2 Beweis der Invariante durch Induktion nach k

$k = 0$: Nach der Initialisierung von D, also vor dem 1. Schleifendurchlauf, gilt obige Invariante.

$k - 1 \to k$:

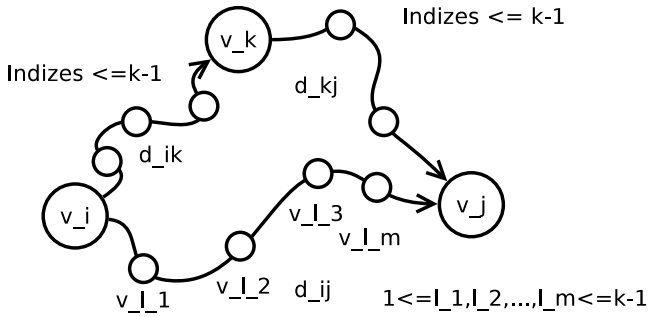


Abbildung 18.2: Beweis der invariante

Durch die Operation $d_{ij} = \min(d_{ij}, d_{ik} + d_{kj})$ wird die Invariante sichergestellt.

18.3 Naive Lösung des All-Pairs Problems durch $|V|$-malige Anwendung von Bellman-Ford oder Dijksta-Algorithmus

Bellman-Ford $\mathcal{O}(|V| \cdot |V| \cdot |E|) = \mathcal{O}(|V|^2 \cdot |E|)$

Dijksra $\mathcal{O}(|V| \cdot (|V| \cdot \log |V| + |E|)) = \mathcal{O}(|V| \cdot |E| + |V|^2 \cdot \log |V|)$

18.4 Johnson-Algorithmus

Idee: Neugewichtung der Kanten, so dass keine negativen Kantengewichte mehr vorhanden sind. Anschließend $|V|$-mal Dijkstra-Algorithmus ausführen.

Naiver Ansatz

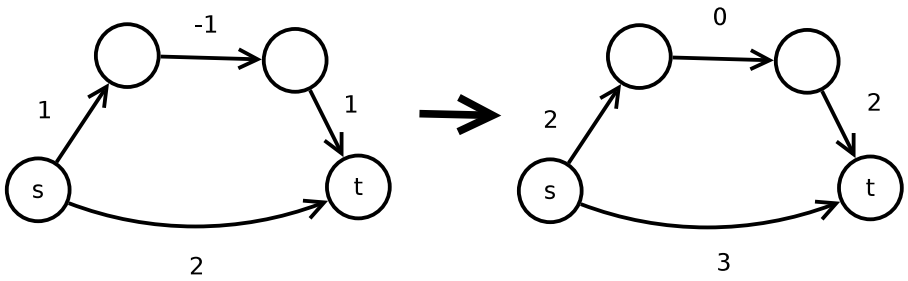


Abbildung 18.3: Naiver Ansatz, kürzester Weg wird zerstört

Neuer Ansatz

$$w'(u,v) = \text{pot}^{\text{III}}(u) - \text{pot}(v) + w(u,v) \geq 0$$

Mit dieser Neugewichtung gilt, dass kürzeste Wege bzgl. w den kürzesten Wegen bzgl. w' entsprechen.

$$p : s = v_0 \to v_1 \to v_2 \to \ldots v_i \to v_{i+1} \to \ldots v_k = t$$

$$w'(p) = \sum_{i=0}^{k-1} w'(v_i, v_{i+1}) = \sum_{i=0}^{k-1} [\text{pot}(v_i) - 1\text{pot}(v_{i+1}) + w(v_i, v_{i+1})]$$

$$\overset{Teleskopsumme}{=} \text{pot}(v_0) - \text{pot}(v_k) + \sum_{i=1}^{k-1} w(v_i, v_{i+1}) = \text{pot}(s) - \text{pot}(t) + w(p)$$

d.h. Alle kürzesten Wege $s \rightsquigarrow t$ unterscheiden sich bzgl. w' im Vergleich zu w nur um eine feste additive Konstante $\text{pot}(s) - \text{pot}(t)$

$$\text{pot}(u) - \text{pot}(v) + w(u,v) \geq 0$$

$$\text{pot}(v) \leq \text{pot}(u) + w(u,v)^{\text{IV}}$$

$$\text{pot}(v) = \delta(z,v)$$

$$G' = (V', E') \quad V' = V \cup z, E' = E \cup (z,v) | v \in V \quad \text{mit } w'(z,v) = 0$$

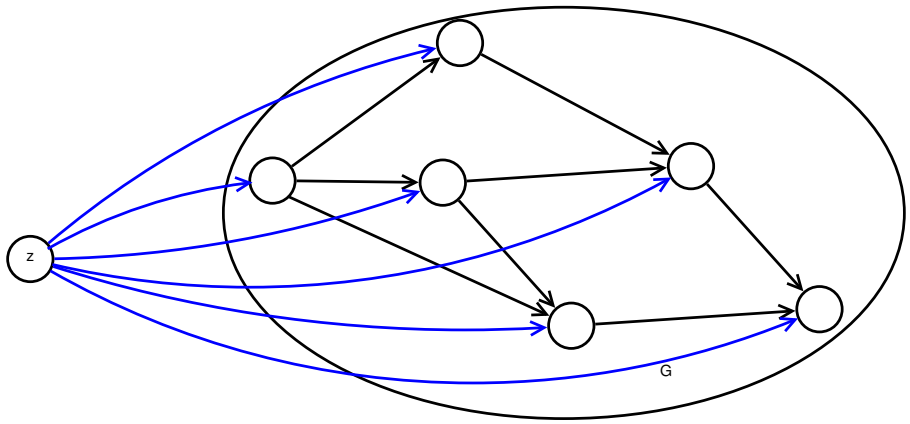


Abbildung 18.4: Die blau makierten Kanten haben die Länge 0

- Löse single-source-shortest-Path Problem in G' mit z als Startknoten
- setze $\text{pot}(v) = \delta_{G'}(z,v)^{\text{V}}$
- Neugewichtung
- $|V|$-mal Djikstra

18.4.1 Laufzeit des Johnson-Algorithmus

$$\mathcal{O}(|V| \cdot |E| + |V| \cdot (|V| \cdot \log|V| + |E|)) = \mathcal{O}(|V| \cdot |E| + |V|^2 \cdot |V|)$$

[III] Potentialfunktion
[IV] Dreiecksungleichung
[V] berechnet mit Bellman-Ford