Andreas Goerdt

Theoretische Informatik III

Skriptum zur Vorlesung

Sommersemester 2001

Version 0.1.1-5

TECHNISCHE UNIVERSITÄT CHEMNITZ

Anstelle eines Vorwortes

Das vorliegende Skriptum entstand in einer Rohfassung als Arbeit studentischer Hilfskräfte aus Vorlesungsmitschriften der Sommersemester 1997 und 1998. Die Kapitel 1 bis 5 dieser Version wurden von Olaf Hartmann und Frank Schädlich an Hand der Vorlesung im Sommersemester 2001 überarbeitet.

Die hier vorliegende Fassung erhebt keinen Anspruch auf Fehlerfreiheit oder Vollständigkeit. Sie ist vielmehr als eine Ergänzung zu den eigenen Aufzeichnungen zu sehen.

Wir haben uns bemüht, die Kapitel 1 bis 5 so gut wie möglich auf einen aktuellen Stand zu bringen. Sollten sich hier dennoch Fehler eingeschlichen haben, so sind diese oder andere Hinweise bitte an olaf.hartmann@s1998.tu-chemnitz.de oder frank.schaedlich@informatik.tu-chemnitz.de zu senden.

Die Kapitel 6 bis 11 werden derzeit überarbeitet. Wann diese aktualisiert zur Verfügung stehen werden ist leider noch nicht abzusehen. In der jetzigen Form sollten sie jedoch sehr kritisch betrachtet werden.

Chemnitz, den 12. Juli 2001

Olaf Hartmann

Frank Schädlich

Inhaltsverzeichnis

Kapitel 1

Binomiale Heaps

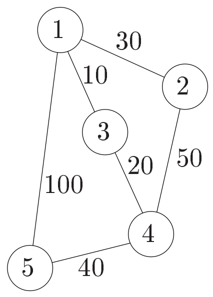
1.1 Einführung

Binomiale Heaps und auch die im nächsten Kapitel behandelten Fibonacci-Heaps sind Datenstrukturen für Priority-Queues (Vorrangswarteschlangen). Im Folgenden wird der Binomiale Heap am Beispiel von Dijkstra's Algorithmus betrachtet.

Dijkstra's Algorithmus:

```
Array D := D[1..n]
D[1] := 0
for i := 2 to n do
  // ω(h,i) = ∞ wenn keine Kante (h,i) existiert
  D[i] := ω(1,i)
S := {1}
for i := 1 to n-1 do
  Suche Knoten k ∈ V\S mit D[k] minimal
  S := S ∪ {k}
  for alle Kanten (k,j) des Graphen mit j ∈ V\S do
    if D[j] > D[k] + ω(k,j) then
      D[j] := D[k] + ω(k,j)
      V\S anpassen
```

Beispiel:



Start: Knoten 1

D = $[1 \dots n]$
S = Menge der Knoten mit kürzesten Wegen

$n = |V| = 5$
S = $\{1\}$

$V\backslash S = \{2,3,4,5\}$
$D[1] = 0$, $D[2] = \ldots = D[5] = \infty$

Zwischenergebnisse der For-Schleife:

S	$V\backslash S$	D[1]	D[2]	D[3]	D[4]	D[5]
{1}	{2,3,4,5}	**0**	30	10	∞	100
{1,3}	{2,4,5}	**0**	30	**10**	30	100
{1,2,3}	{4,5}	**0**	**30**	**10**	30	100
{1,2,3,4}	{5}	**0**	**30**	**10**	**30**	70
{1,2,3,4,5}	{}	**0**	**30**	**10**	**30**	**70**

Laufzeit, wenn $V\backslash S$ als Heap implementiert wird:

$$\begin{array}{rcl}
|E| \text{ mal Kanten in Heap einfügen} & = & O(|E|) \\
(n-1) \text{ mal Minimum finden und löschen} & = & O(n \log n) \\
|E| \text{ (Anzahl der Kanten) mal } V\backslash S \text{ anpassen in } O(\log n) & = & O(|E| \log n) \\
\hline
\text{Laufzeit insgesamt bei } |E| \geq n-1 & = & O(|E| \log n)
\end{array}$$ [1]

Laufzeit mit $V\backslash S$ als boolsches Array:

$$\begin{array}{rcl}
(n-1) \text{ mal Minimum finden und löschen} & = & O(n^2) \\
\text{D-Werte ändern} & = & O(|E|) \\
\hline
\text{Laufzeit insgesamt} & = & O(n^2)
\end{array}$$

Der Heap ist also besser, solange

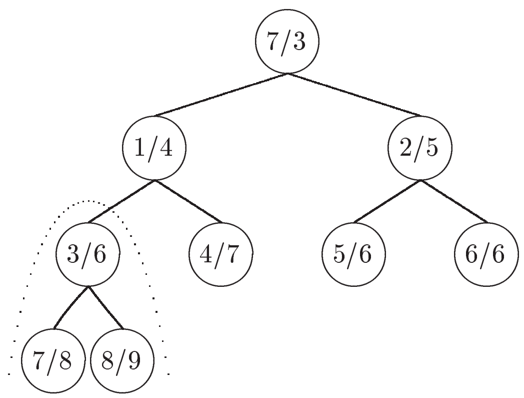
$$|E| = O\left(\frac{n^2}{\log n}\right), \quad \frac{n^2}{\log n} \leq n^2 - \epsilon \;\; \forall \epsilon > 0 \text{ und } n \to \infty.$$

Daraus ergibt sich die maximale Laufzeit für Dijkstra's Algorithmus: $O((n + |E|) \cdot \log n + |E|)$. $O(n \log n)$ ist dabei die Zeit zum Löschen des Minimums aus $V\backslash S$ und $O(|E| \cdot \log n)$ die Zeit zum Anpassen des Heaps.

Beachte: $O(n \log n + |E|)$ ist $O(|E|)$, wenn $|E| \geq n \log n$. $O(|E|)$ ist auf jeden Fall nicht zu verbessern, da man jede Kante einmal ansehen muß.
Unser Ziel ist es nun den Heap so anzupassen, daß eine Änderung des D-Wertes möglichst in $O(1)$ erfolgt. ($|E|$-mal `decreasekey` in Zeit $O(|E|)$) Damit würde sich die Laufzeit von Dijkstra's Algorithmus auf $O(n \log n + |E|)$ verbessern.

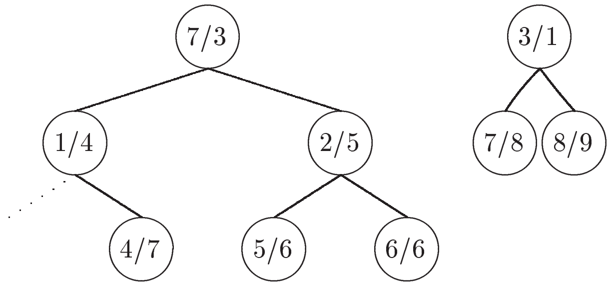
Bemerkung: Damit wird die Laufzeit nie schlechter als die der „naiven“ Implementation.

Grundidee dazu ist:



[1] Geht man von einem zusammenhängenden Graphen aus, gilt natürlich $|E| \geq n-1$.

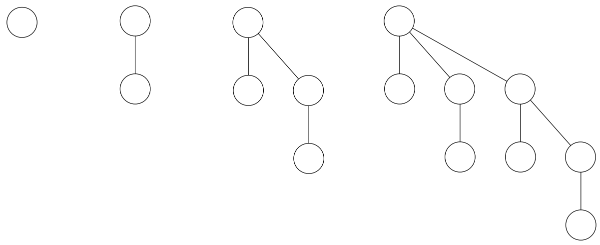
`decreasekey`(3, 1) (Name, neuer D-wert)
Da die Heapeigenschaft verletzt ist, schneidet man den Teilbaum mit 3/1 als Wurzel herraus. Dadurch erhalten wir 2 Bäume mit Heapeigenschaft:



Man braucht also eine Möglichkeit, kontrolliert Bäume (=Heap) zu teilen und zusammenzusetzen.

Dazu eine induktive Definition (binomialer Baum B_i für jedes $i \geq 0$)

$i = 0$	$i = 1$	$i = 2$	$i = 3$
B_0	B_1	B_2	B_3



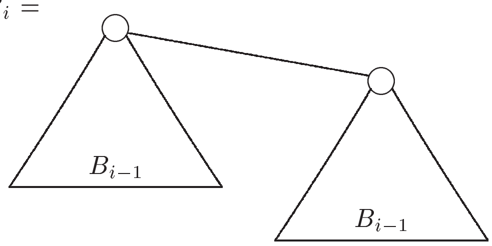
1.2 Der binomiale Baum

Definition 1.2.1. Der i-te binomiale Baum wird induktiv über i definiert:

$i = 0$  0-ter binomiale Baum.

$i > 0$, Der i-te binomiale Baum hat die Struktur

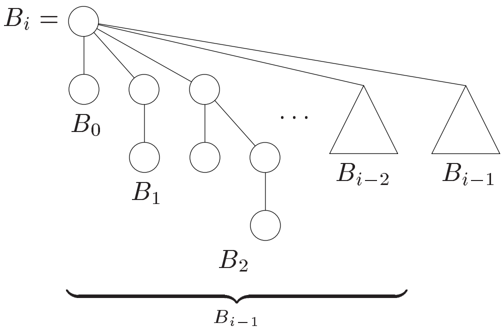
$B_i =$



Ein i-ter binomialer Baum ist also aus zwei $(i-1)$-ten binomialen Bäumen zusammengesetzt.
Bezeichnung: B_i = i-ter binomialer Baum.

Satz 1.2.2.

(a) Sei $i > 0$. Dann hat ein i-ter binomialer Baum folgendes Aussehen:



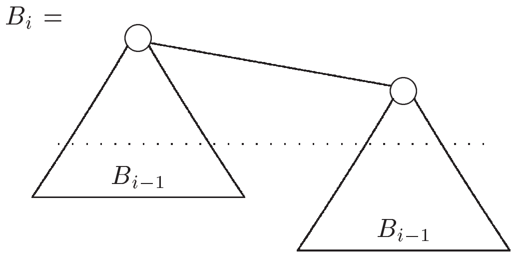
Unter der Wurzel von B_i hängen also $B_0,\ B_1,\ \dots\ , B_{i-1}$.

(b) Tiefe(B_i)$= i$ für alle $i \geq 0$.

(c) $|B_i| = 2^i$ für alle $i \geq 0$. ($|B_i|$ = Anzahl Knoten von B_i)

(d) $|\{x|\ x$ ist Knoten von $B_i\ \wedge$ Tiefe$(x) = j\}| = \binom{i}{j}$
für alle i, j mit $i \geq 0$ und $i \geq j \geq 0$.
(In Tiefe j des binomialen Baumes B_i gibt es genau $\binom{i}{j}$ Knoten)

Beweis:
(a), (b), (c) siehe Übungsblatt
(d) Der Beweis erfolgt durch Induktion über i sowie für alle j mit $i \geq j \geq 0$.
B_i die Form



Induktionsanfang: $i = 0 \Rightarrow j = 0 \Rightarrow \binom{0}{0} = 1$
Induktionsschluß: Es gelte die Behauptung für $i-1$ und alle j mit $i-1 \geq j \geq 0$.
Dann ist für j beliebig mit $i-1 \geq j \geq 1$

$$
\begin{aligned}
&|\{x|\ x \text{ Knoten von } B_i\ \wedge\ \text{Tiefe}(x) = j\}| \\
&\quad = \quad |\{x|\ x \text{ Knoten von } B_{i-1}\ \wedge\ \text{Tiefe}(x) = j\}| \\
&\qquad\quad + \ |\{x'|\ x' \text{ Knoten von } B_{i-1}\ \wedge\ \text{Tiefe}(x') = j-1\}| \\
&\quad \overset{\text{Ind. Vor}}{=} \binom{i-1}{j} + \binom{i-1}{j-1} = \binom{i}{j}
\end{aligned}
$$

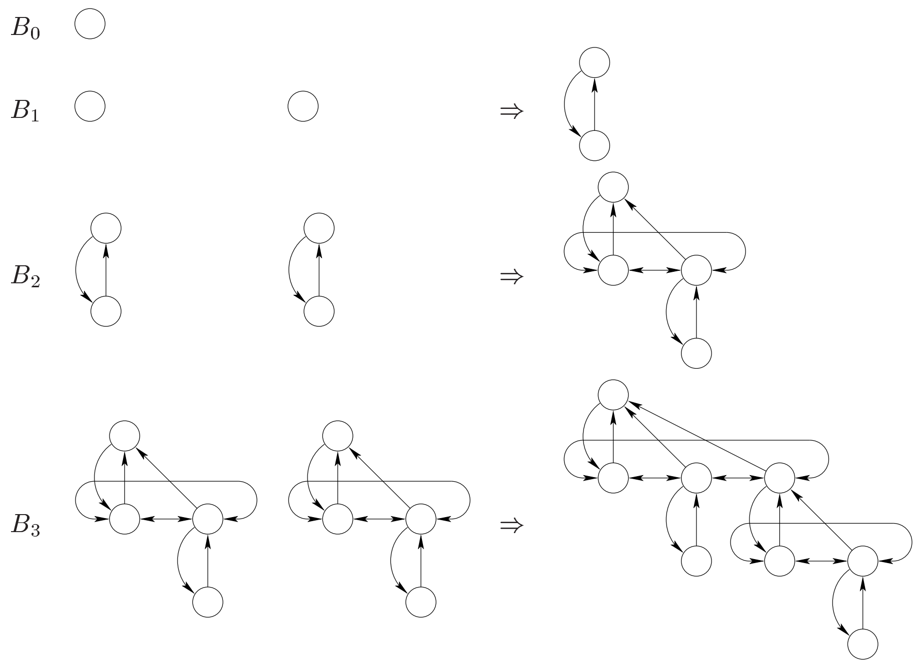
Gilt $i = j$, so hat der binomiale Baum B_{i-1} in Tiefe i nach (b) keinen Knoten. Es gilt also:

$$
\begin{aligned}
&|\{x|\ x \text{ Knoten von } B_i \ \wedge\ \text{Tiefe}(x) = i\}| \\
=\quad & \underbrace{|\{x|\ x \text{ Knoten von } B_{i-1} \ \wedge\ \text{Tiefe}(x) = i\}|}_{=0} \\
& +\ |\{x'|\ x' \text{ Knoten von } B_{i-1} \ \wedge\ \text{Tiefe}(x') = i-1\}| \\
\overset{\text{Ind. Vor}}{=}\quad & \binom{i-1}{i-1} = 1 = \binom{i}{i} = \binom{i}{j}
\end{aligned}
$$

1.3 Implementierung binomialer Bäume

Wir gehen von der Heapeigenschaft der Binomialen Bäume aus.

(a) Implementierung mit Zeigern (Pointer)



Vom Vater geht ein Zeiger zum ersten Sohn. Die Söhne sind ringartig doppelverkettet. Von allen Söhnen geht ein Pointer zum Vater.
Die triviale Ringliste der Blätter wurde zur besseren Übersicht weggelassen.

(b) Sind B und C zwei i-te binomiale Bäume die die Heapeigenschaft erfüllen, dann ergibt `link`(B, C) einen $(i+1)$-ten binomialen Baum.

`link`(B, C):

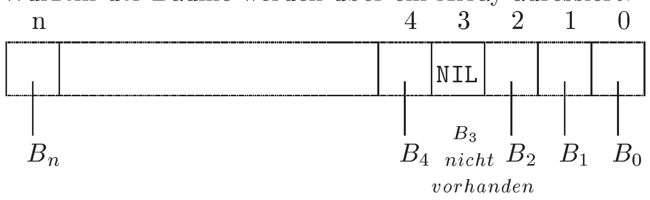
```
if Wert von Wurzel(B) ≥ Wert von Wurzel(C) then
  Suche rechtesten Sohn von Wurzel(C)
  Füge Wurzel(B) als neuen rechtesten Sohn von Wurzel(C) ein
else
  Suche rechtesten Sohn von Wurzel(B)
  Füge Wurzel(C) als neuen rechtesten Sohn von Wurzel(B) ein
```

Die Operation **`link`(B, C)** hat Laufzeit O(1) wegen der Wahl der Pointer.

1.4 Struktur des binomialen Heaps

Um beliebige Elementarzahlen darstellen zu können, ist der binomiale Heap eine Menge von binomialen Bäumen B_i für verschiedene i.
Im binomialen Heap ist eine beliebige Mächtigkeit darstellbar, da jede Zahl eine Summe von Zweier-Potenzen ist.

Definition 1.4.1. Ein binomialer Heap ist eine Menge von binomialen Bäumen, wobei es 0 (keinen) oder 1 (genau einen) binomialen Baum B_i für jedes $i \geq 0$ gibt. Die Wurzeln der Bäume werden über ein Array adressiert.



Zusätzlich werden immer zwei Variablen mitgeführt:

- `heapsize` = Max$\{i \mid \mathrm{A}[i] \neq \texttt{NIL}\}$
- `min` = Zeigt auf auf die Wurzel des Baumes, dessen Wurzel den minimalen D-Wert enthält.

Ist h ein binomialer Heap, so ist $|h|$ = Anzahl der Knoten in allen Bäumen (= Anzahl gespeicherter Elemente).

Es gilt immer:

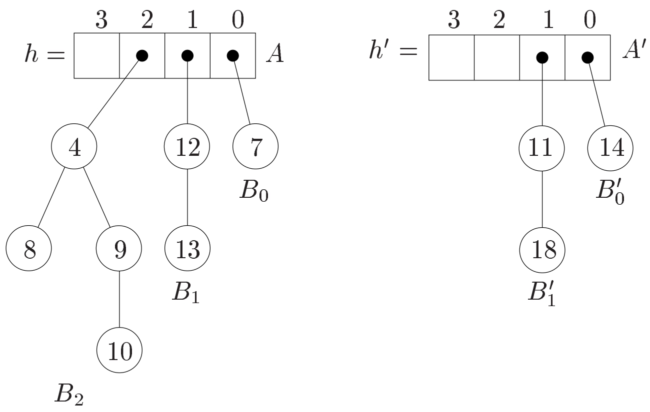
$$2 \cdot 2^{\texttt{heapsize}} - 1 \geq |h| \geq 2^{\texttt{heapsize}}$$

$$\left(\sum_{i=0}^{n-1} 2^i = 2^n - 1, \quad 2^n + (2^n - 1) = 2 \cdot 2^{\texttt{heapsize}}\right)$$

$$\text{alternativ:} \quad \texttt{heapsize} + 1 \geq \log_2 |h| \geq \texttt{heapsize}$$

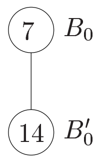
1.5 Operationen auf dem binomialen Heap

(a) `meld`(h,h') (`meld`=Verschmelzen)
Eingabe: 2 binomiale Heaps h, h'.
Ausgabe: 1 binomialer Heap, wobei h und h' vereinigt wurden.
Vorgehensweise: analog der Addition von Binärzahlen von rechts nach links (unter Verwendung von `link`(B, C) aus 1.3)
Am Beispiel:

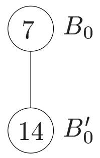


$$\text{Vergleiche:}\quad \begin{array}{r} 0111 \\ +0011 \\ \hline 1010 \end{array}$$

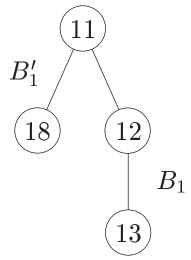
$B_0 + B_0' \;=\;$ `NIL` Übertrag:



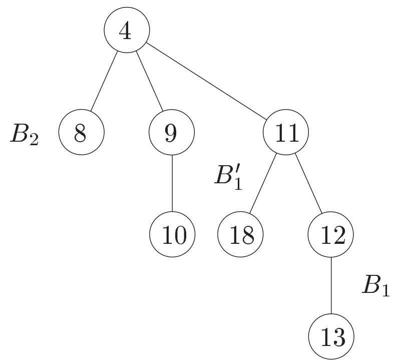
$B_1 + B_1' + \text{Übertrag} \;=\;$



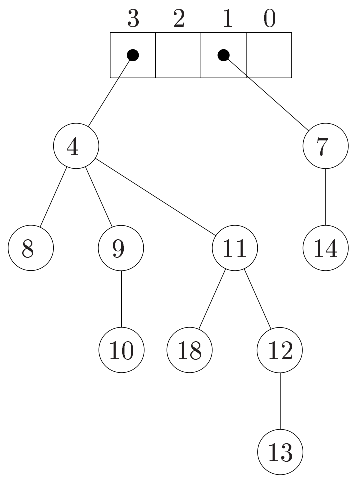
Übertrag:



$B_3 + \textit{Übertrag} \;=\;$ `NIL` Übertrag:



`meld`$(h,\, h') \;=\;$



Die Laufzeit von `meld`(h,h') ist $\mathrm{O}(\log|h| + \log|h'|)$, also logarithmisch in der Gesamtzahl von Elementen.
Das ist bei den bisher betrachteten Heaps nicht so schnell möglich, da im wesentlichen die Bäume neu strukturiert werden müssen. (Neuaufbau des Heaps „von unten“ ist in Linearzeit möglich)

(b) `makeheap`(i,j) (i = Element, j = Schlüsselwert)
Es wird wird ein Array mit B_0 (= Knoten i mit j als Schlüsselwert) gebildet.
Laufzeit O(1).

(c) `insert`(i,j,h)
`meld`(`makeheap`$(i,j),h$)
Laufzeit $O(\log |h|)$.

(d) `deletemin`(h)

Minimum löschen:

1. Minimum suchen mittels `min`, ist `min` $= k$, dann hängen unter der Wurzel (=Minimum) von B_k die Bäume B_0, B_1, ... ,B_{k-1}.
2. Wir erzeugen Heap h' aus B_0, B_1, ... ,B_{k-1}.
3. Man löscht B_k aus h durch das Setzen von $A[k]$ auf `NIL`.
4. `meld`(h,h').
5. Neues Minimum suchen im Ergebnis von `meld`(h, h').

Laufzeiten:	1. O(1)
($n = \|h\|$)	2. $O(\log n)$
	3. O(1)
	4. $O(\log n)$
	5. $O(\log n)$

Insgesamt $O(\log n)$.

Satz 1.5.1. Das Einfügen von n Elementen in den anfänglich leeren binomialen Heap erfolgt in Zeit $\mathbf{O(n)}$ (!) (nicht $O(n \log n)$).

Das einzelne Einfügen kann Zeit $\Omega(\log n)$ erfordern, also gilt unter der Voraussetzung im obigen Satz sicher die Zeitabschätzung $O(n \log n)$.

Beweis:
Zuerst ein Beispiel:
`insert`$(1,1,h)$; `insert`$(2,2,h)$; ... ; `insert`$(9,9,h)$
h = leer (am Anfang)

`insert`(1,1)	Zeit: 1 `makeheap`	
	1 Eintrag in h	
	1 `link`	ein `link` gezählt
`insert`(2,2)	Zeit: 1 `makeheap`	
	1 Eintrag in h	kein `link` gezählt
	1 `link`	ein `link` gespeichert
`insert`(3,3)	Zeit: 1 `makeheap`	
	1 Eintrag in h	
	0 `link`	ein `link` gezählt
`insert`(4,4)	Zeit: 1 `makeheap`	
	1 Eintrag in h	
	2 `link`	ein `link` gezählt

`insert`(5,5)	Zeit: 1 `makeheap`	
	1 Eintrag in h	
	0 `link`	ein `link` gezählt
`insert`(6,6)	Zeit: 1 `makeheap`	
	1 Eintrag in h	
	1 `link`	ein `link` gezählt
`insert`(7,7)	Zeit: 1 `makeheap`	
	1 Eintrag in h	
	0 `link`	ein `link` gezählt
`insert`(8,8)	Zeit: 1 `makeheap`	
	1 Eintrag in h	
	3 `link`	ein `link` gezählt

Formal: Induktiv über der Anzahl der Elemente, die man speichern will mit folgender Zeitinvariante:
An jedem Baum ist eine „Zeiteinheit" gespeichert. Dann läßt sich als Induktionsschluß zeigen:
Das Einfügen eines weiteren Elementes benötigt Zeit O(1), wobei die Zeitinvariante bestehen bleibt.

Bemerkung:

(a) Man sagt bei n Einfügungen in den leeren Heap benötigt ein `insert` die *amortisierte* Zeit O(1) (dies ist sozusagen der Mittelwert: $\frac{\text{Gesamtzeit}}{\text{Anzahl d. Operationen}} = \frac{\mathrm{O}(n)}{n} = \mathrm{O}(1)$).

(b) Wie kann man die amortisierte Zeit beispielsweise noch sehen?
Sei t_i die eigentliche Zeit für das i-te Einfügen, sei a_i die Zeit, die wir gezählt haben und sei Φ_{i-1} die Zeit, die in dem Heap vor der i-ten Operation gespeichert ist.

Situation: `insert`(1,1,h), `insert`(2,2,h), ... , `insert`(i,i,h)

$$\Phi_0 = 0 \xrightarrow[t_1/a_1]{} \Phi_1 = 1 \xrightarrow[t_2/a_2]{} \Phi_2 \xrightarrow[t_3/a_3]{} \quad \dots \quad \xrightarrow[t_{i-1}/a_{i-1}]{} \Phi_{i-1} \xrightarrow[t_i/a_i]{} \Phi_i$$

$a_1 = a_2 = \cdots = a_i = \mathrm{O}(1)$

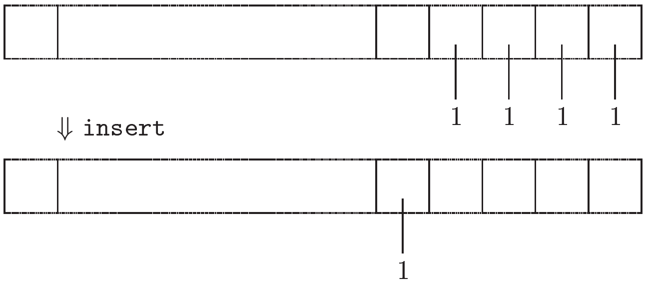
(Könnte $\Omega(\log i)$ sein.)
(Was ist $\Phi_i - \Phi_{i-1}$? Die Differenz der gespeicherten Zeiten.)

Jetzt ist $a_i = t_i + (\Phi_i - \Phi_{i-1})$.

$$\begin{aligned}
\sum_{i=1}^{n} a_i &= \underbrace{t_1 + (\Phi_1 - \overbrace{\Phi_0}^{=0})}_{=a_1} + \underbrace{t_2 + (\Phi_2 - \Phi_1)}_{=a_2} + \cdots + \underbrace{t_n + (\Phi_n - \Phi_{n-1})}_{=a_n} \\
&= \sum_{i=1}^{n} t_i + \Phi_n \\
\Longrightarrow & \sum_{i=1}^{n} t_i \leq \sum_{i=1}^{n} a_i - \Phi_n \leq \sum_{i=1}^{n} a_i
\end{aligned}$$

Φ_i = Potential der i-ten Struktur
$\Phi_i - \Phi_{i-1}$ = Potentialdifferenz, die bei i-ter Operation erzeugt wird
(Hier: Potential = Anzahl der Bäume)

(c) Zu $a_i = t_i + \Phi_i - \Phi_{i-1}$



$\Phi_{i-1} = 4$
Dann ist
$t_i + \Phi_i - \Phi_{i-1}$
$= 1 + 1 + 1$

$\Phi_i = 1$
t_i= 1 `makeheap` + 4 `link`
+ 1 Eintrag in h

Satz 1.5.2. Führen wir eine Folge σ von m Operationen $\sigma_1, \sigma_2, \ldots, \sigma_m$ von m_1 `insert`'s, m_2 `deletemin` und m_3 `min` $(m_1 + m_2 + m_3 = m)$ auf dem anfangs leeren Heap aus und ist n die maximale Anzahl von Elementen im Heap, so ist die Zeit für die Ausführung gleich $O(m_1) + O(m_2 \cdot \log n) + O(m_3)$.

D. h. `deletemin` benötigt ebenfalls Zeit $O(\log n)$.

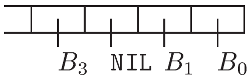
Beweis: Der Beweis dieses Falls erfolgt in der Übung.

Zusammenfassung

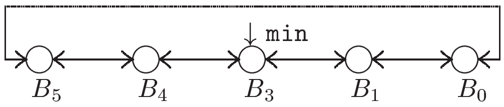
Operation	Direkte Adressen	Binärer Heap (Worst-Case)	Binomialer Heap (Worst-Case)	Binomialer Heap (amortisiert)
`insert`	O(1)	$O(\log n)$	$O(\log n)$	O(1)
`deletemin` (ohne Finden)	O(1)	$O(\log n)$	$O(\log n)$	$O(\log n)$
Minimum finden (`min`)	O(n)	O(1)	O(1)	O(1)

1.6 Binomialer Heap mit „lazy“ `meld`(h,h')

(a) Statt einem Array wird nun eine doppelt verkettete zirkuläre Liste zum Verwalten der binomialen Bäume verwendet:
Vorher h=

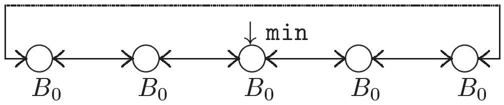


jetzt:



Beachte: *keine* Leerstelle für B_2

sogar noch allgemeiner: ein i-ter bimomialer Baum kann beliebig häufig auftreten, zum Beispiel:



(b) `meld`(h,h')
Einfaches Zusammenhängen der Listen in O(1) („lazy“).
`min` anpassen, auch in O(1).

(c) `insert`(i,j,h)
`meld`$(h,$`makeheap`$(i,j))$ und `min` anpassen. Zeit O(1)

(d) `deletemin`(h)

Eingabe: h als Liste von binomialen Bäumen. Man kann an der Wurzel den Typ (i von B_i) ablesen (d. h. das i von B_i ist immer mitzuführen)
Ausgabe: Modifikation mit gelöschtem Minimum.

Minimum löschen:

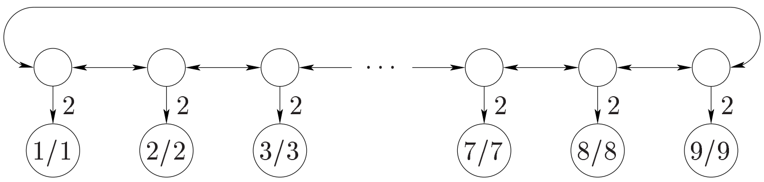
1. Generiere Array A mit $\lfloor \log_2 |h| \rfloor + 1$ `NIL`-Pointern.
2. Gehe die Liste h durch. Trage jeden Baum mit `makeheap` und `meld` (dem ursprünglichen `meld`) in das Array A ein. Nehme beim Minimum die Kinder der Wurzel. Laufzeit: O(n).
3. Suche Minimum in A und erzeuge doppelt verkettete zirkuläre Liste (O($\log n$)).
   Zeit im Worst-Case O(n).

Idee: Diese Zeit bei den Wurzeln der Bäume speichern.

Satz 1.6.1. Beginnen wir mit leerem binomialen Heap (mit „lazy“ `meld`), so haben wir

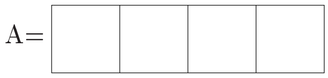
	`deletemin`	`insert`	`min`
Worst-Case	O(n)	O(1)	O(1)
amortisiert	O($\log n$)	O(1)	O(1)

Beweis:
Vorab ein Beispiel:
nach 9 `insert`'s:



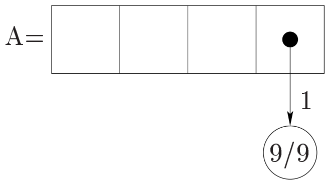
Was geschieht jetzt bei `deletemin`(h)?

1.



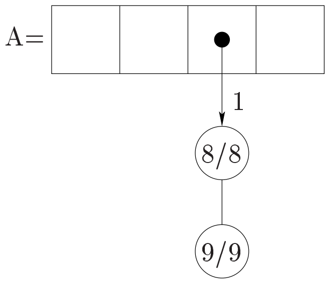
$4 = \lfloor \log_2 9 \rfloor + 1$, $O(\log n)$ zum Aufbau von A

2.



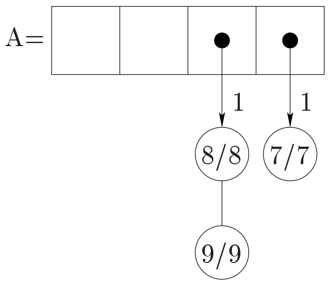
gebrauchte Zeit O(1), zählen 0

3.



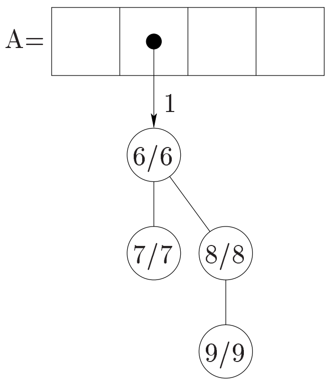
gebrauchte Zeit O(1), zählen 0

4.



gebrauchte Zeit O(1), zählen 0

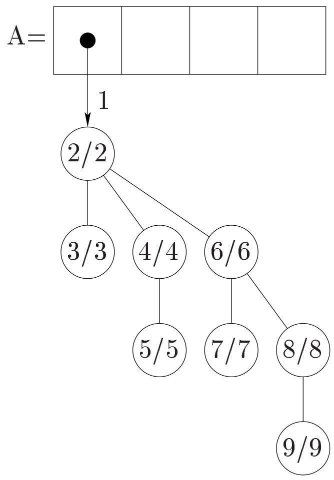
5.



gebrauchte Zeit O(2), zählen 0

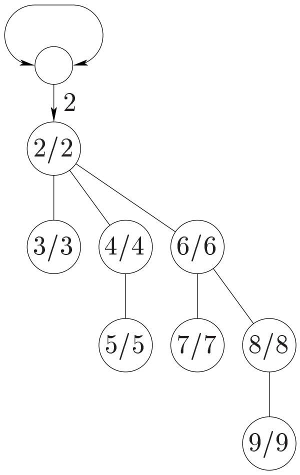
⋮

9.



0 gezählt für den Aufbau des Heaps.

10.



zählen jetzt 2
damit zählen wir insgesamt $O(\log n)$
Neues Minimum suchen benötigt $O(\log n)$
Also `deletemin` in $O(\log n)$ amortisiert.

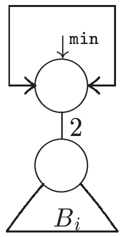
Bemerkung:
Formalisiert verbirgt sich dahinter folgender Beweis: Sei $\sigma_1, \sigma_2, \ldots, \sigma_m$ eine Folge von Operationen auf den binomialen Heap. Seien $h_0, h_1, \ldots, h_m$ die Heaps und $t_1, t_2, \ldots, t_m$ die *echten* Laufzeiten.

$$h_0 = 0 \xrightarrow[t_1]{\sigma_1} h_1 = 1 \xrightarrow[t_2]{\sigma_2} h_2 \xrightarrow[t_3]{\sigma_3} \cdots \xrightarrow[t_m]{\sigma_m} h_m$$

$\Phi(h) = 2 \cdot$ Anzahl Bäume von h.
$a_i = t_i + (\Phi_i - \Phi_{i-1}) \qquad$ für $1 \leq i \leq n$.
Jetzt können wir dir amortisierten Laufzeiten der Operationen beweisen:

(a) $\sigma_i = \mathtt{min}(h)$ $\quad a_i = O(1)$
(b) $\sigma_i = \mathtt{insert}(i, j, h)$ $\quad a_i = O(1)$
(c) $\sigma_i = \mathtt{deletemin}(h)$ $\quad a_i = O(\log n)$

(a), (b) gilt.
(c) Sei einmal das Minimum Wurzel eines größeren Baumes:



Unser Beweis gilt, wenn an jeder Wurzel 2 Zeiteinheiten gespeichert sind.
Für die Kinder des Minimums ist keine Zeit gespeichert, aber durch Zählen weiterer $2 \cdot \log n$ wird die Situation wieder hergestellt. Die amortisierte Zeit bleibt dabei bei $O(\log n)$.
Also bei m_1 `deletemin`, m_2 `insert` und m_3 `delete` gilt:

$$\begin{aligned}\sum_{i=1}^{n} a_i &= m_1 \cdot O(\log n) + m_2 \cdot O(1) + m_3 \cdot O(1) \\ &= \sum_{i=1}^{n} \underbrace{(t_i + \Phi_i - \Phi_{i-i})}_{a_i} \\ &= \sum_{i=1}^{n} t_i + \underbrace{\Phi_n}_{\geq \sigma} \qquad \text{da } \Phi_0 = 0 \\ \Rightarrow \sum_{i=1}^{n} t_i &\leq \sum_{i=1}^{n} a_i\end{aligned}$$

Zusammenfassung

	binomialer Heap („lazy" `meld`)		binomialer Heap („eager" `meld`)	
	Worst-Case	amortisiert	Worst-Case	amortisiert
`insert`	$O(1)$	$O(1)$	$O(\log n)$	$O(1)$
`deletemin`	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
`min`	$O(1)$	$O(1)$	$O(1)$	$O(1)$
`decreasekey`	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$

Mit „lazy" Meld hat Dijkstra's Algorithmus eine Laufzeit von $O(|E| \log |V|)$ wenn $|E| \geq |V|$ gilt.

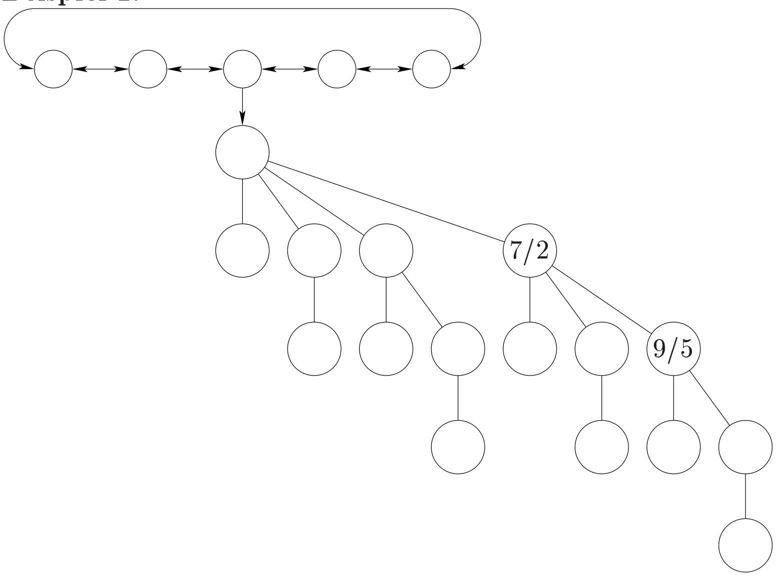
Kapitel 2

Fibonacci-Heaps

2.1 Einführung

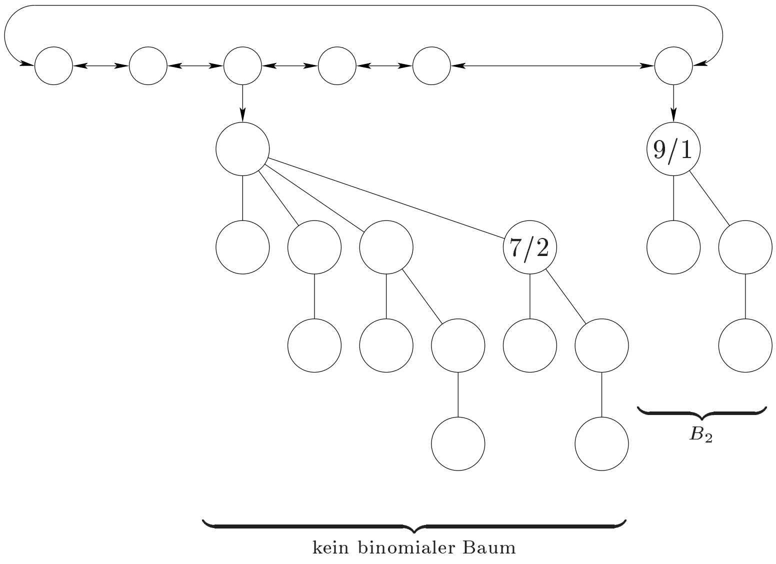
Ziel: Modifikation des binomialen Heaps, so daß `decreasekey`(i, j, h) amortisiert in O(1) läuft.

Beispiel 1:



`decreasekey`(9, 1, h)

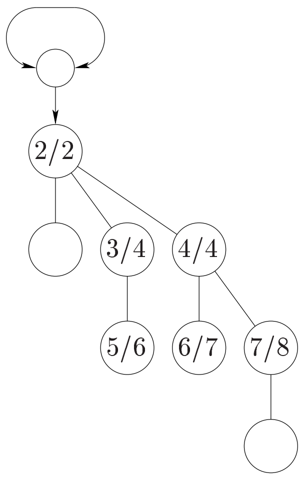
Idee: Herausschneiden und Einschmelzen (in O(1)) ergibt:



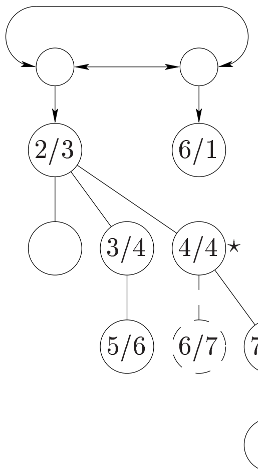
Problem: Es muß sichergestellt werden, daß durch iteriertes Herausschneiden die Kinderzahl im Verhältnis zu Elementanzahl nicht zu groß wird. Die Bäume dürfen also nicht zu breit und flach werden.

Beispiel 2:
Wie soll das gehen? Die Vermeidung gar zu breiter flacher Bäume erfolgt, indem pro (nicht-Wurzel-) Knoten immer nur ein Kind abgeschnitten wird. Das geht so:

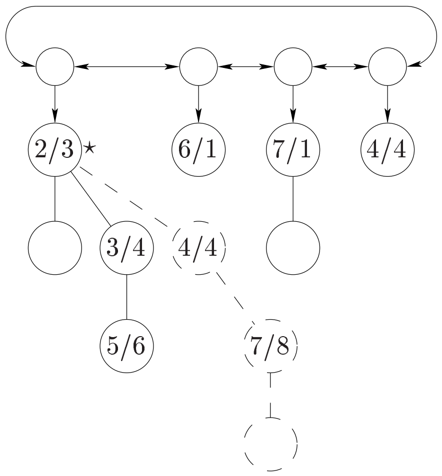
$h =$



`decreasekey`(6, 1, h)



`decreasekey`(7, 1, h)



Cascading-Cuts: Wurde vom Vater des abgeschnittenen Knotens bereits ein Sohn entfernt, so wird auch der Vater herausgeschnitten. Das wird solange fortgesetzt, bis ein Knoten erreicht ist, der selbst eine Wurzel ist oder bei dem noch kein Kind weggeschnitten wurde.

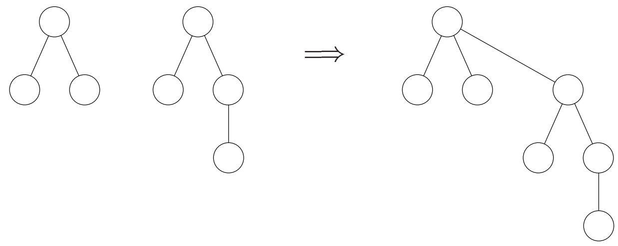
2.2 Die Datenstruktur des Fibonacci-Heap

Struktur wie binomialer Heap mit „lazy" `meld`.

(a) `insert`(i, j, h)
1. `meld`(h, `makeheap`(i, j)
2. Minimum anpassen
O(1)

(b) `min`(h)
Minimum finden in O(1)

(c) `deletemin`(h)
Wie beim binomialen Heap mit „lazy" `meld`. (1.6)
Da es sich im allgemeinen nicht mehr um binomiale Bäume handelt wird statt dem i-ten binomialen Baum der Baum, dessen Wurzel genau i Kinder hat genommen.



Bäume können nur dann verschmolzen werden, wenn ihre Wurzeln gleich viele Kinder (dirkete Nachfolger) haben. (Wir führen die Kinderzahl bei jedem Knoten mit)

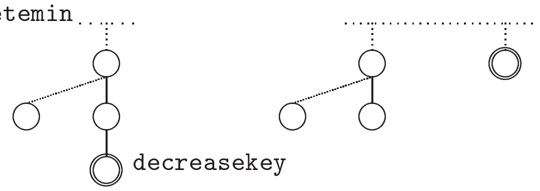
(d) `cut`(x, h)
Herauschneiden des Teilbaumes, dessen Wurzel der Knoten x ist.

1. Schneide Teilbaum x und schmelze mit „lazy“ `meld` ein.
   Falls x markiert (siehe 2.) Löschen der Markierung.
2. Ist der Vater von x nicht die Wurzel und nicht markiert, dann markiere ihn mit $\star$ (markiert $\Leftrightarrow$ 1 Kind fehlt).
3. Sonst: Ist der Vater y von x bereits (von vorher) markiert, dann `cut`(y, h).

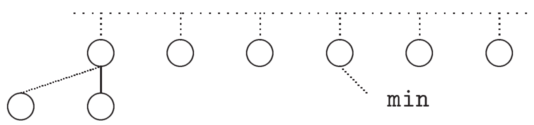
(e) `decreasekey`(x, j, h)
$j \leq$ alter Schlüssel von x.

1. Gehe zum Knoten x, der i enthält.
2. Setze Schlüessel von i auf j.
3. Ist die Heapeigenschaft verletzt, so führe `cut`(x, hx) aus.

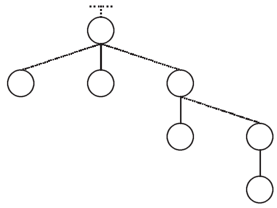
Beispiel
`deletemin`



4 mal `insert`



Nach `deletemin`:



Beachte: kombiniert werden Bäume nur, wenn die gleiche Anzahl Kinder an der Wurzel hängt.

Ziel: `decreasekey` amortisiert in Zeit O(1) (d. h. iteriertes Herausschneiden muß ohne Zeitaufwand zu zählen sein) und `deletemin` amortisiert in Zeit $\mathrm{O}(\log n)$ (Anzahl der Kinder an der Wurzel darf höchsten $\mathrm{O}(\log n)$ sein).

Satz 2.2.1. Ist r die Wurzel eines Teilbaumes im Fibonacci-Heap, so gilt:

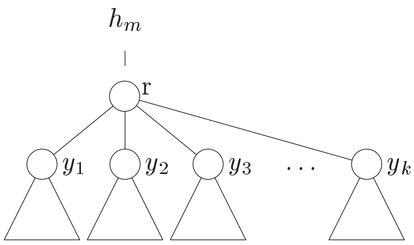
$$|\mathrm{T}| \geq s^{c \cdot \text{Anzahl Kinder von } r}$$

für ein festes c $(0 < c < 1)$.

Im binomialen Baum ist c=1
Also: Anzahl Kinder $\leq \frac{1}{c}(\log \mathrm{T}) = \mathrm{O}(\log \mathrm{T})$.

Beweis:
Vorab: Der Fibonacci-Heap wird gemäß unseren Operationen aus dem leeren Heap aufgebaut.Mittels Link(B, C) (beim Aufruf von `deletemin`) entstehen kompliziertere Bäume.
Link(B, C) wird ausgeführt $\Leftrightarrow$ Grad von B = Grad von C
(Grad = Anzahl der Kinder der Wurzel)

Seien h_0, h_1, h_2, h_3 $\ldots$, h_m mit $|h_0| = 0$ eine Folge von Fibonacci-Heaps, die mit den Operationen `insert`, `delete` und `min` auseinander hervorgehen. Sei r ein Knoten in h_m:



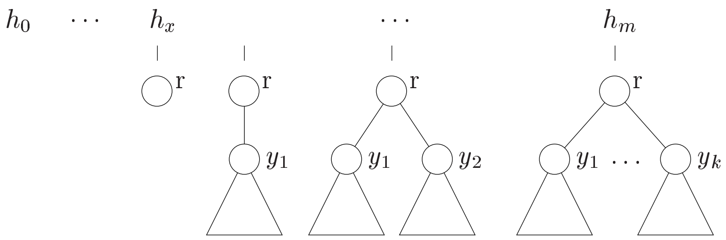
$k \geq 0$

(bei $k = 0$ gilt die Behauptung automatisch)

Jedes y_i ist durch ein geeignetes `link` an r gehängt worden.

Sei o. B. d. A. y_1 = das Kind, das am längsten an r hängt
y_2 = das zweite
$\vdots$
y_k = das Kind, das am kürzesten an r hängt

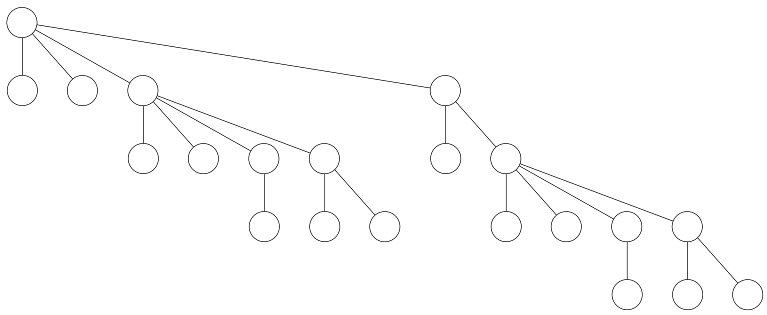
Zeitverlauf:



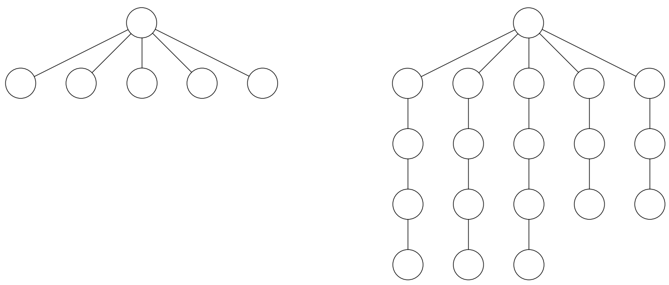
r kann noch weitere Kinder gehabt haben, die aber zwischendurch wieder gelöscht wurden.
Nun gilt die Schlüsselaussage:

Satz 2.2.2. In h_m ist der Grad von $y_i \geq i - 2$ für alle i.

Beweis (der Schlüsselaussage):
Wenn y_i an r kommt, hat r mindestens die Kinder $y_1, \ldots, y_{i-1}$. Wegen `link` hat y_i dann auch $\geq i-1$ Kinder. Bis zum Ende kann y_i wegen `cut` maximal ein Kind verlieren. Somit hat y_i am Ende mindestens $i-2$ Kinder.
Also haben die Bäume in etwa die folgende Struktur:

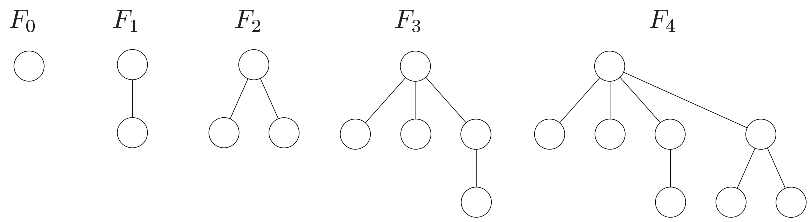


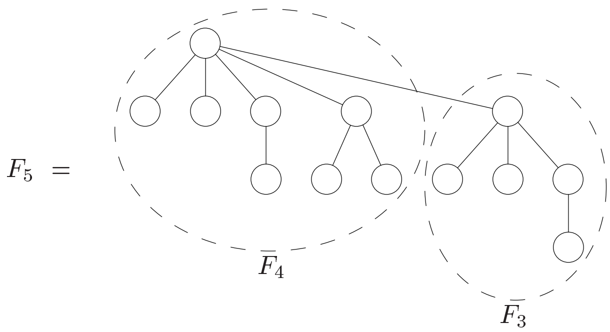
Derartige Bäume sind ausgeschlossen:

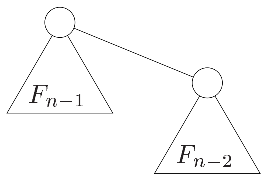


Die Frage ist: Wieviel Bäume können nur auftreten?

Sei $n \geq 0$ und sei F_n ein kleinster Baum, dessen Wurzel genau n Kinder hat und für jeden Knoten r von F_n gilt unsere Bedingung von oben.
Die kleinsten F_n haben folgendes Aussehen:





$F_n =$ 

Sei $f_n = |F_n|$ (die Anzahl der Knoten in F_n). Dann ist:

$$f_0 = 1,\ f_1 = 2,\ f_2 = f_0 + f_1 = 3$$
$$f_n = f_{n-1} + f_{n-2} \quad \text{für } n > 2 \qquad (= n\text{-te Fibonacci-Zahl})$$

Es gilt $f_n \geq \varphi^n$ wobei $\varphi = \frac{1+\sqrt{5}}{2} \approx 1{,}618$ der Goldene Schnitt ist.
(Goldener Schnitt: Eine Strecke der Länge x wird in zwei Teile mit den Längen a und b ($a \geq b$) geteilt, so daß $\frac{x}{a} = \frac{a}{b}$) gilt. Man berechnet leicht, daß dies für $a = \frac{x}{\varphi}$ gilt.)

2.3 Laufzeiten

Satz 2.3.1. Bei anfänglich leerem Heap benötigt der Fibonacci-Heap folgende Zeiten:

Operation	Binomialheap („eager“)		Binomialheap („lazy“)		Fibonacci-Heap	
	Worst-Case	amortisiert	Worst-Case	amortisiert	Worst-Case	amortisiert
`insert`	$O(\log n)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
`deletemin`	$O(\log n)$	$O(\log n)$	$O(n)$	$O(\log n)$	$O(n)$	$O(\log n)$
`min`	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
`decreasekey`	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$	$O(1)$
`cut`	-	-	-	-	$O(n)$	$O(1)$

Beweis: Im Worst-Case ist keine Verbesserung möglich. (siehe Übung)
Für die amortisierten Zeiten definieren wir die Potentialfunktion:

$$\Phi(h) = \text{Anzahl Bäume von } h \ + \ 2 \cdot \text{Anzahl markierte Knoten.}$$

Dann analog zu Satz 1.6.1

Folgerung: Mit $V \backslash S$ im Fibonacci-Heap läuft Dijkstras Algorithmus in Zeit $O(|V| \cdot \log |V| + |E|\ )$ ($|V|$ mal `deletemin`, $|E|$ mal `decreasekey`).

Kapitel 3

Union-Find-Strukturen

3.1 Einführung

Sei G ein ungerichteter, gewichteter Graph mit $G = (V, E, c)$ mit $c \colon E \to \mathbb{R}$. Ein Teilgraph $H = (V, F, c)$ mit $F \subseteq E$ ist ein Spannbaum von G genau dann, wenn H ein zusammenhängender Graph ist und und H nach dem Entfernen einer beliebigen Kante nicht mehr zusammenhängend ist.

Beachte: Es muß $|F| = |V| - 1$ gelten.

Gesucht ist ein Spannbaum mit minimaler Kostensumme der Kanten.
Der Kruskal-Algorithmus findet einen minimalen Spannbaum durch schrittweisen Aufbau von Kanten in den anfänglich leeren Graphen, d. h. ohne Kanten.

Bemerkung: Ein Spannbaum kann nur gefunden werden, wenn der Graph G zusammenhängend ist. Dies kann in Linearzeit mittels der Tiefen- oder Breitensuche festgestellt werden (siehe Vorlesung Theoretische Informatik I). Es sei also im Weiteren G ein zusammenhängender Graph.

Kruskal-Algorithmus:

1. Kanten nach ihrem Gewicht c ordnen.
2. Kanten der Reihe nach untersuchen.
   (a) Sind die beiden Knoten der Kante in verschiedenen Teilen? (dazu Operation `find`)
   (b) Wenn ja, dann vereinige die beiden Teile und füge die Kante dem Spannbaum hinzu. (dazu Operation `union`)

$\Rightarrow$ Man muß eine Partition der Knotenmenge mitführen.
Am Anfang:

$$(\{v_1\}, \{v_2\}, \{v_3\}, \ldots, \{v_n\})$$

Kante (v_2, v_5) in den (zukünftigen) Spannbaum einfügen:

$$(\{v_1\}, \{v_2, v_5\}, \ldots)$$

Eine erste Möglichkeit eine Partition P darzustellen besteht darin, ein Feld A zu verwenden. Dabei bedeutet $\mathrm{A}[i] = x$ das der Knoten i in der Menge x liegt.

Beispiel:

$$\mathrm{P} = (\{1, 2\}, \{3, 4, 5\}, \{6, 7\})$$

Array A[1] = A[2] = 1
A[3] = A[4] = A[5] = 2 $\boxed{1\ 1\ 2\ 2\ 2\ 3\ 3}$
A[6] = A[7] = 3

$\Rightarrow$ `find`: O(1) `union`: (n)
Kruskal mit der Partition als Array benötigt Laufzeit:

$$O(|E| \cdot \underbrace{\log|E|}_{O(\log|V|} + \underbrace{|E|}_{2\cdot|E| \,\times\, \texttt{find}} + \underbrace{|V|^2}_{|V| \,\times\, \texttt{union}})$$

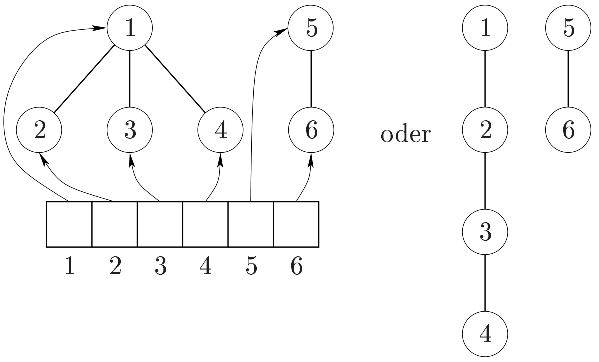
Das ist $O(|E| \cdot \log|V| + |V|^2)$.
Ist $|E| \subseteq O\left(\frac{|V|^2}{\log|V|}\right)$, dann $O(|V|^2)$ (gilt nur nicht für sehr dichte Graphen).

Ziel: Es soll eine Datenstruktur gefunden werden, so daß n `union` und m `find` möglichst schnell sind. Die untere Schranke für jede Union-Find-Struktur ist offensichtlich $\Omega(n+m)$. Diese untere Schranke wird amortisiert „fast“ erreicht.

3.2 Partition als Menge von Bäumen

Beim Einsatz von Bäumen anstelle von Listen in Datenstrukturen kann oft eine Laufzeitverbesserung erreicht werden. Die Partition P wird nun als eine Menge von Bäumen dargestellt.

Beispiel: P = $(\{1,2,3,4\},\{5,6\})$



`union`(i, j) (i, j sind die Namen der Wurzeln) in O(1)
`find`(i) in O(Tiefe(i))
Wie tief kann ein Knoten i liegen?

Die Partition P wird geschickt durch ein Vorgängerarray gespeichert:

	1	2	3	4	5	6
A	1	1	1	1	5	5

Dabei enthält A[i] den Vater des Knotens i im Baum. Gilt A[i] $= i$, so ist der Knoten i die Wurzel eines Baumes in der Partition P.

Beachte: Es kann passieren, daß `find`(v) Zeit $\Omega(n)$ dauert.

Diese schlechte Laufzeit des `find` kann durch die Heuristik Union-By-Size verbessert werden:
Bei jeder Wurzel wird die Anzahl der Elemente mitgeführt. Bei der Operation `union` wird dann immer der kleinere unter den größeren Baum gehängt.

Folgerung: Bei Union-By-Size gilt für jeden Baum T, daß Tiefe(T) $\leq \log_2 |T|$ ist. Damit wird die Laufzeit eines `find` durch $O(\log_2 |T|)$ beschränkt.

Anmerkung: Mit Union-By-Size ist der Baum mit dem ungünstigstem Verhältnis von Kinderzahl zu Tiefe (große Tiefe bei wenigen Elementen) ein binomialer Baum.

Laufzeit von Kruskal mit Union-By-Size:

$$O(\underbrace{|E| \cdot \log|V|}_{\text{sortieren}} + \underbrace{|E| \cdot \log|V|}_{\texttt{find}} + \underbrace{|V|}_{\texttt{union}}) = O(|E| \cdot \log|V|).$$

Bei vorsortierten Kanten bleibt es bei $O(|E| \cdot \log|V|)$, die Hauptzeit fällt dann in den `find`-Operationen an.

3.3 Algorithmus Union-By-Size mit Wegkompression

Während eines `find`(i) werden alle Knoten von i zur Wurzel durchlaufen. Damit hat man aber für diese Knoten ebenfalls die Wurzel gefunden. Diese zusätzlich gewonnenen Informationen werden in der Heuristik Wegkompression ausgenutzt:

(a) `union`(v, w) mit Heuristik Union-By-Size:

```
if Anz[v] ≤ Anz[w] then
  P[v]:= w
  Anz[w] := Anz[w] + Anz[v]
else
  P[w] := v
  Anz[v] := Anz[v] + Anz[w]
```

Zeit: O(1)

(b) `find`(v) mit Heuristik Wegkompression:

```
Speichere v auf Keller
while P[v] ≠ v do
  v := P[v]
  Speichere v auf Keller
Gebe v aus
for each w auf dem Keller do P[w] := v
```

Alternativ mit rekursivem Aufruf:

```
if P[v] ≠ v then       // P[v] ist keine Wurzel
  P[v] := find(P[v])
return P[v]
```

Zeit: Worst-Case $O(\log n)$

Satz 3.3.1. Mit Union-By-Size und Wegkompression benötigt man für λ `find` und κ `union`-Operationen bei der anfänglichen Partition P=$(\{1\}, \{2\}, \ldots, \{n\})$ die Zeit $O(\kappa + (n + \kappa) \cdot \log^* n)$. Dabei ist $\log^*$ definiert durch:

$$\log^* = \text{Min}\{s \mid \log^{(s)} n \leq 1\}$$

$$\text{mit } \log^{(s)} n = (\log \circ \log \circ \log \circ \cdots \circ \log)n = \log(\log(\ldots(\log(n))\ldots)).$$

$\log^* n$ ist fast O(1). Beispiel: $\log^* 2^{16} = \log^* 65\,536 = 5$.

Beachte: Die offensichtliche untere Schranke ist $\Omega(\kappa + \lambda)$, die obere Schranke ist $O(\kappa + \lambda \cdot \log n)$. Ein einzelnes `find` kann jedoch $\Omega(\log n)$ dauern.

Bei linear vielen `find`-Operationen ist die mittlere (und auch die amortisierte) Zeit für ein `find` $O(\log^* n)$.

Was heißt „eine lineare Anzahl von `find`-Operationen"?
Das bedeutet: Es gibt ein c, so daß $\lambda \leq c \cdot n$ ist. Ferner werden maximal n Elemente eingefügt. Damit ist $\kappa \leq n$.
Dann ist:

$$O(\underbrace{n \cdot \log^* n}_{=\kappa \leq n} + \underbrace{\lambda}_{\leq c \cdot n} \cdot \log^* n) \;=\; O((1+c) \cdot n \cdot \log^* n)$$

Damit ist die mittlere Zeit für ein `find` $\frac{1}{c \cdot n} \cdot O((1+c) \cdot n \cdot \log^* n)$, das ist $O((\frac{1}{c} + 1) \cdot \log^* n)$, also $O(\log^* n)$.

Beweis:
Der Beweis des Satzes Satz 3.3.1 (Union-By-Size mit Wegkompression) erfolgt durch die folgenden Definitionen und Lemmata.

Sei $S_0 =$ 1/1 2/1 3/1 … n/1 die anfängliche Union-Find-Struktur.
Sei $\sigma = \sigma_1, \sigma_2, \sigma_3, \ldots, \sigma_m$ eine Folge von `union`- und `find`-Operationen.
Sei $\lambda =$ Anzahl `find`, $\kappa =$ Anzahl `union` und $m = \lambda + \kappa$.

Es seien die Strukturen $S_0, \ldots, S_m$ gegeben durch:

$$S_0 \xrightarrow{\sigma_1} S_1 \xrightarrow{\sigma_2} S_2 \xrightarrow{\sigma_3} S_3 \cdots S_{m-1} \xrightarrow{\sigma_m} S_m.$$

D. h. S_i ist die Union-Find-Struktur die nach der Abarbeitung der Operation σ_i mit Wegkompression entstanden ist. S_0 ist dabei die anfänglich leere Struktur.
Die Idee liegt in der zusätzlichen Betrachtung der Strukturen

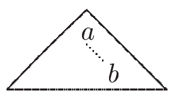
$$S'_0,\ S'_1,\ \ldots,\ S'_m \quad \text{mit } S_0 = S'_0.$$

S'_i entsteht, indem $\sigma_1, \ldots, \sigma_i$ ohne Wegkompression auf S'_0 ausgeführt werden. S_i und S'_i stellen die selbe Partition dar. Die Wurzeln der einzelnen Bäume in S_i und S'_i sind ebenfalls gleich.

Definition 3.3.2. Für ein Element a mit $1 \leq a \leq n$ ist:
Level(a) = Tiefe des Teilbaumes mit Wurzel a in der Struktur S'_m, also in der Struktur ohne Wegkompression nach Abarbeitung der letzten Operation σ_m.

Lemma 3.3.3.

(a) Hat S'_i mit $1 \leq i \leq m$ folgende Struktur:



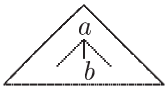
b ist Nachfolger von a in S'_i, a kann, muß aber nicht, die Wurzel sein, so gilt: Level(a) $\gneqq$ Level(b).

(b) Hat S_i die selbe Struktur (b ist Nachfolger von a in S_i) wie im Fall (a), so gilt ebenfalls: Level$(a) \gneqq$ Level(b).

Beweis:

(a) Ist a nicht die Wurzel, dann kann sich unter a nichts mehr ändern und die Behauptung gilt.
Ist a die Wurzel, so kann der Level(a) höchstens noch größer werden.

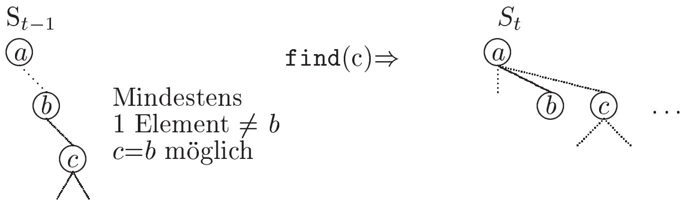
(b) Sei also in S_i die Situation:



(b ist Sohn von a)

1. Fall: Die Kante $a \leftarrow b$ ist durch $\sigma_t = \texttt{union}(a, b)$ mit $t \leq i$ entstanden. Dann ist in S_{t-i} die Anzahl der Elemente über $b \leq$ der Anzahl der Elemente über a. Da S_{t-1} und S'_{t-1} die selben Partitionen darstellen gilt das auch in S'_{t-1}. Also gilt in S'_t auch: a und b sind Wurzeln und es gilt die Elementbeziehung. Also ist die ist die Kante $a \leftarrow b$ auch in S'_i entstanden.
Wegen (a) folgt dann die Behauptung.

2. Fall: Die Kante $a \leftarrow b$ ist durch Wegkompression aus S_{t-1} mit $\sigma_t = \texttt{find}(c)$ entstanden.



Dann ist auch b in S'_{t-1} in jedem Fall unter a, damit ist (mit (a)) Level$(a) \gneqq$ Level(b).

In Abhängigkeit von Level(a) wird der Wert Niveau(a) definiert. Dazu wählt man ein k mit $0 \leq k \leq \lfloor \log n \rfloor$ und $k+2$ natürliche Zahlen $A_0, A_1, A_2, \ldots, A_k, A_{k+1}$ mit:

$$A_0 = 0 \lneqq A_1 \lneqq A_2 \lneqq \cdots \lneqq A_k \leq \lfloor \log n \rfloor \lneqq A_{k+1},$$

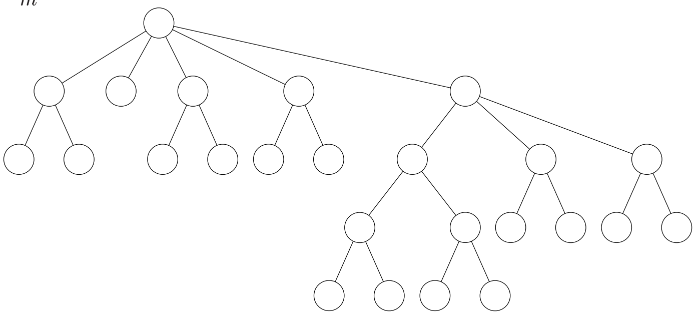
Die Tiefe der Bäume in S'_i ist logarithmisch beschränkt. Man kann somit Knoten suchen, die mindestens eine Tiefe A_l und maximal die Tiefe A_{l+1} haben. Diese Knoten werden später zu Niveaus zusammengefaßt.
An Hand der A_i-Werte kann später die amortisierte Laufzeit berechnet werden. Die Wahl der A_i hat Einfluß auf eine möglichst geringe Laufzeitabschätzung.

Beispiel:

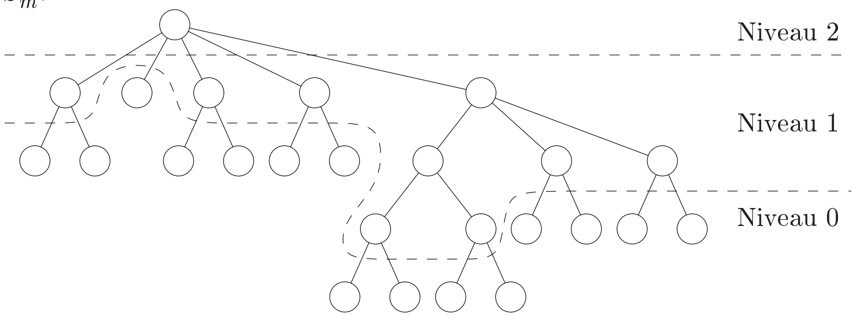
(a) $k = 0, \quad t_0 = 0, \quad A_1 = \lfloor \log_2 n \rfloor + 1$

(b) $k = \lfloor \log_2 n \rfloor$, dann $A_0 = 0$, $A_1 = 1, \ldots, A_k = \lfloor \log_2 n \rfloor$, $A_{k+1} = A_k + 1$

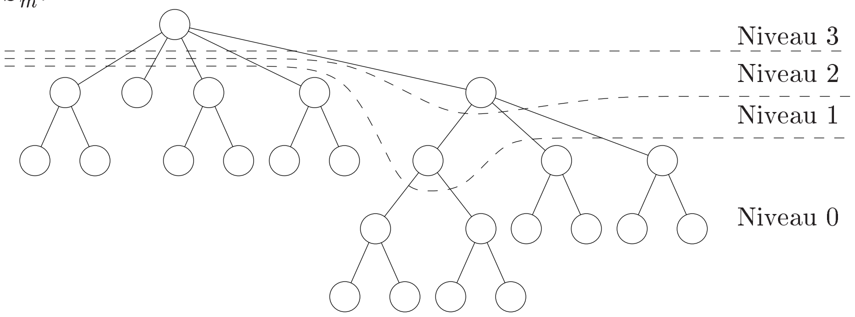
(c) S'_m:



$\lfloor \log_2 25 \rfloor = 4$
wähle $k = 2$ und $A_0 = 0$, $A_1 = 1$, $A_2 = 4$, $A_3 = 5$
S'_m:



(d) S'_m wie bei (c)
wähle $k = 3$ und $A_0 = 0$, $A_1 = 2$, $A_2 = 3$, $A_3 = 4$, $A_4 = 5$
S'_m:



Definition 3.3.4. Für $0 \leq i \leq k$ ist
Niveau$(a) = i \quad \iff \quad A_i \leq \text{Level}(a) \lneqq A_{i+1} \qquad$ Intervall $[A_i,\ A_{i+1})$

Bemerkung: Es ist jedes Element von genau einem Niveau, da $0 \leq \text{Level}(a) \leq \lfloor \log_2 n \rfloor$ gilt.
Ist Level$(a) = 0$, dann gilt Niveau$(a) = 0$
Ist Level$(a) = \lfloor \log_2 n \rfloor$, dann ergibt sich Niveau$(a) = k$.

Wieviel Elemente existieren auf einem Niveau?

Lemma 3.3.5.
(a) Sei $l \in \mathbb{N}$, dann gilt:

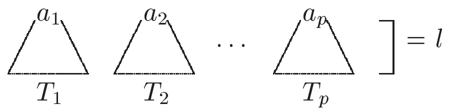
$$|\{a \mid \text{Level}(a) = l\}| \;\leq\; \frac{n}{2^l}$$

(b)

$$|\{a \mid \text{Niveau}(a) = i\}| \;\leq\; \frac{2 \cdot n}{2^{A_i}}$$

Beweis:

(a) Es seien $a_1, a_2, \ldots, a_i$ alle Elemente vom Level l. Somit ist $|\{a \mid \text{Level}(a) = l\}| = i$. Die Struktur S'_m enthält nun folgende (Teil-) Bäume:



Ferner gilt $\text{Tiefe}(T_1) = \text{Tiefe}(T_2) = \cdots = \text{Tiefe}(T_p) = l$.
Aufgrund der gleichen Tiefe sowie der unterschiedlichen Wurzeln sind alle Bäume T_i disjunkt. Wegen der Heuristik Union-By-Size ist $|T_i| \geq 2^l \quad (\Leftrightarrow\ l \leq \log_2 |T_i|)$.
Also gilt:

$$n \ \geq\ \sum_{i=1}^{p} |T_i| \ \geq\ p \cdot 2^l \text{ und damit } p \leq \frac{n}{2^l}.$$

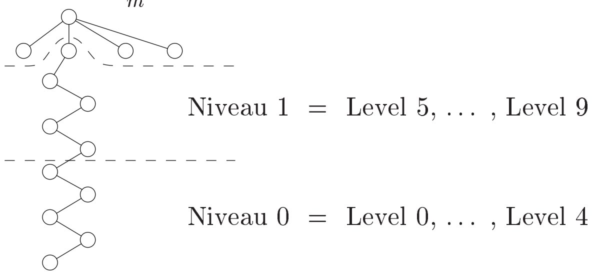
(b)

$$\begin{aligned}
|\{a \mid \text{Niveau}(a) = i\}| &= |\{a \mid \text{Level}(a) = A_i\}| \ + \ |\{a \mid \text{Level}(a) = A_i + 1\}| \\
&\quad + \ |\{a \mid \text{Level}(a) = A_i + 2\}| \ + \ \ldots \\
&\quad + \ |\{a \mid \text{Level}(a) = A_{i+1} - 1\}| \\
&\leq n \cdot \frac{1}{2^{A_i}} + \frac{1}{2^{A_i+1}} + \frac{1}{2^{A_i+2}} + \ \cdots \ + \frac{1}{2^{A_{i+1}-1}} \\
&\leq n \cdot \frac{1}{2^{A_i}} \cdot (1 + \frac{1}{2} + \frac{1}{4} + \ldots) \\
&= \frac{n}{2^{A_i}} \sum_{i=0}^{\infty} \frac{1}{2^i} \qquad \text{geometrische Reihe} \\
&= \frac{n}{2^{A_i}} \cdot 2 \\
&= \frac{2 \cdot n}{2^{A_i}}
\end{aligned}$$

Lemma 3.3.6. Eine Folge $\sigma = \sigma_1, \sigma_2, \ldots, \sigma_n$ von λ `union`- und κ `find`-Operationen benötigt eine Zeit von maximal

$$c \cdot \Big(\lambda \cdot (k+1) + \kappa + \sum_{i=0}^{k} \frac{2 \cdot n}{2^{A_i}} (\underbrace{A_{i+1} - A_i - 1}_{\text{Anzahl Levelübergänge im Niveau } i})\Big).$$

Beispiel:
Sei $n = 2000$ und damit $\lfloor \log_2 n \rfloor = 10$. Es wird k=2 gewählt. Ferner wählt man z. B. folgende $(k+2)$ Zahlen: $A_0 = 0$, $A_1 = 5$, $A_2 = 10$ $A_3 = 11$
Baum in S'_m:

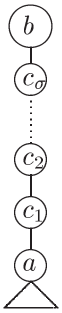


Die Anzahl der Levelübergänge im Niveau 0 ist 4 = 5 - 0 - 1 = $A_1 - A_0 - 1$

Beweis: Die Operationsfolge $\sigma = \sigma_1, \ \dots \ ,\sigma_m$ sei fest. Die verwendete Zeit wird nun wie folgt verteilt:
σ_i = `union`(a,b) benötigt Zeit O(1). Diese Zeit wird für σ_i als eine Zeiteinheit gezählt, d. h. es wird keine Zeit gespeichert.

σ_i = `find`(a): benötigt Zeit O(1 + Anzahl Kanten von a bis zur Wurzel b).

Nun betrachtet man die Datenstruktur S_{i-1}, bevor also die Operation σ_i = `find` ausgeführt wird. Zwischen der Wurzel b und dem Knoten a liegen $v \geq 0$ Knoten c_1, ... , c_v:



Nach Lemma 3.3.3 gilt für die Level:

$$\text{Level}(b) \gneqq \text{Level}(c_v) \gneqq \text{Level}(c_{v-1}) \gneqq \ \cdots \ \gneqq \text{Level}(c_1) \gneqq \text{Level}(a)$$

also auch:

$$\text{Niveau}(b) \geq \text{Nivea}(c_v) \geq \text{Niveau}(c_{v-1}) \geq \ \cdots \ \geq \text{Niveau}(c_1) \geq \text{Niveau}(a).$$

Für jede Kante l auf dem Weg gilt eine der beiden Möglichkeiten:
(a) l verbindet 2 Knoten des selben Niveaus, oder
(b) nicht des selben Niveaus.

Zur Verteilung der Zeit von `find`(a):

(a) Für Kanten innerhalb eines festen Niveaus wird die Zeit O(1) bei dem unteren Knoten der Kante gespeichert, sofern der obere Knoten der Kante nicht die Wurzel ist.

(b) Die Zeit für Kanten mit Niveauübergang und für die Kanten zur Wurzel wird als Zeit von σ_i = `find`(a) verrechnet ($\leq$ Anzahl Niveauübergänge + 1 = k + 1)

Daraus ergibt sich folgende Zusammensetzung:

$$\text{Zeit von } \sigma_1, \ \dots \ ,\sigma_m \ \leq \ \underbrace{O(\kappa)}_{\text{für union}} \ + \ \underbrace{O(\lambda \cdot (k+1))}_{\text{direkt bei find gezählt (b)}} \ + \ \underbrace{\text{gespeicherte Zeit}}_{\text{(a)}}$$

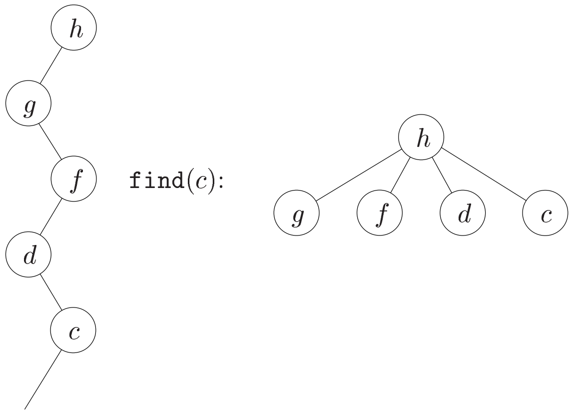
Wieviel Zeit wird bei (a) gespeichert?

Behauptung: Es wird in (a) maximal die Zeit

$$\sum_{i=0}^{k} \frac{2 \cdot n}{2^{A_i}}(A_{i+1} - A_i - 1)$$

gespeichert.

Beweis: Sei Niveau(c) = i, so ist am Ende bei c die Zeit $A_{i+1} - A_i - 1$ gespeichert. Jedes Mal, wenn bei c ein O(1) gespeichert wird, liegt folgende Situation vor (c und der Vater von c sind keine Wurzel):



Nach Lemma 3.3.3 ist Level(f) $\geq$ Level(d) + 1 (f existiert, da d keine Wurzel ist). Also gilt: Wenn in c gespeichert wird, bekommt c einen neuen Vater. Das Level des neuen Vaters ist mindestens um 1 größer als das des alten Vaters. Ist irgendwann das Level des Vaters von $c \geq A_{i+1}$, dann ist das Niveau des Vaters von $c \geq i + 1$. Dann ist die Kante von c zu seinem Vater ein Niveauübergang und wird nicht mehr bei c gezählt.
Daraus folgt: Bei c wird maximal die Anzahl der Levelübergänge im Niveau von c, $[A_i, A_{i+1})$ gezählt. Das ist gerade $A_{i+1} - A_i - 1$.
Die Anzahl der Elemente im Niveau i ist $\leq \frac{2 \cdot n}{2^{A_i}}$. Damit ist die maximal gespeicherte Zeit:

$$\sum_{i=0}^{k} \frac{2 \cdot n}{2^{A_i}}(A_{i+1} - A_i - 1).$$

Ziel: Wahl der A_i so, daß die Zeit nach Lemma 3.3.6 möglichst klein wird.

Beispiel:

(a) Es sei $k = 0$, $A_0 = 0$, $A_1 = \lfloor \log_2 n \rfloor + 1$. Daraus ergibt sich eine maximale Laufzeit

$$\begin{aligned} &\leq c \cdot (\lambda \cdot (k+1) + \kappa + 2 \cdot n \cdot (A_1 - A_0 - 1)) \\ &= c \cdot (\lambda + \kappa + 2 \cdot n \cdot \log_2 n) \\ &= O(n \cdot \log n). \end{aligned}$$

(b) Es sei $k = \lfloor \log_2 n \rfloor$, $A_0 = 0$, $A_1 = 1$, $A_2 = 2, \ldots, A_k = \lfloor \log_2 n \rfloor$, $A_{k+1} = A_k + 1$. Es ergibt sich ebenfalls für die Laufzeit $O(n \cdot \log n + \kappa) = O(n \cdot \log n)$.

Die Laufzeiten in (a) und (b) sind noch nicht optimal.

Folgerung 3.3.7. Die A_i werden für ein möglichst schnelles Wachstum wie folgt rekursiv definiert: $A_0 = 0$, $A_1 = 1$ und $A_{i+1} = 2^{A_i}$. Damit ist $A_0 = 0$, $A_1 = 1$, $A_2 = 2^{A_1} = 2$, $A_3 = 2^{A_2} = 4$, $A_4 = 2^{A_3} = 16$, $A_5 = 2^{A_4} = 2^{16} \ldots$, d. h. k ist so zu wählen, so daß $A_k \leq \lfloor \log_2 n \rfloor < A_{k+1}$ gilt, es ist also $k = \log^* n$.
Dann beträgt die Laufzeit für die Operationsfolge

$$\sigma \;=\; O(\lambda \cdot \log^* n + \kappa + 2 \cdot n \cdot \log^* n).$$

Beweis: Was ist $k = k(n)$? Es gilt $k(n) = \log^* n$.
Es ist (vgl. Lemma 3.3.6):

$$\begin{aligned}
\sum_{i=0}^{k(n)} \frac{2 \cdot n}{2^{A_i}}(A_{i+1} - A_i - 1) &\leq \sum_{i=0}^{k(n)} \frac{2 \cdot n}{2^{A_i}} A_{i+1} \\
&= 2n \cdot \underbrace{(\underbrace{\frac{A_1}{2^{A_0}}}_{=1} + \underbrace{\frac{A_2}{2^{A_1}}}_{=1} + \;\cdots\; + \underbrace{\frac{A_{k+1}}{2^{A_k}}}_{=1})}_{\log^* n} \\
&= O(n \cdot \log^* n)
\end{aligned}$$

$\Rightarrow$ Es gilt Satz 3.3.1.

Kapitel 4

Selbstorganisierende Listen

Selbstorganisierende Listen können zur Realisierung der Datenstruktur Wörterbuch („Dictionary“) verwendet werden. Zu den typischen Wörterbuchoperationen gehören die drei Operationen Einfügen, Löschen und Finden:

`find`(x)	Liefert einen Zeiger auf x, falls x enthalen ist.
`insert`(x)	Einfügen von x, falls x nicht enthalten ist.
`delete`(x)	Löschen von x, falls x enthalten ist.

Wenn die Schlüssel aus einer sehr großen Menge (etwa alle Namen) gewählt werden ist keine direkte Adressierung möglich.
Eine Lösung ist das Benutzen von:

(a) Ausgeglichenen Bäumen (z. B. AVL-Bäume, B-Bäume). Die Laufzeit für jede Operation ist dann im Worst-Case $O(\log n)$.

(b) Listen (z. B. Überlauflisten beim im Hash). Jede Operation braucht dann im Worst-Case die Zeit $\Omega(n)$. Listen sind dennoch sinnvoll, wenn extrem wenig Speicherplatz zur Verfügung steht.

4.1 Datenstruktur

Definition 4.1.1. (Selbstorganisierende Liste): Die Liste hat folgende Gestalt

$$\to x_1 \to x_2 \to x_3 \to \quad \cdots \quad \to x_n$$

Dies kann z. B. über folgende Array-Darstellung realisiert werden:

`start` = 5

x_1	3	x_4	5	x_2	4	x_3	2	x_5	0
1		2		3		4		5	

Folgende Arten der Implementierung sind zulässig:

(a) `find`(x)
1. Suchen vom Anfang bis x.
2. Eine Menge von Transpositionen.

(b) `insert`(x)
1. Suchen nach x (bzw. sicherstellen, daß x noch nicht vorhanden ist).
2. Einfügen von x am Ende der Liste.
3. Transpositionen.

(c) `delete`(x)
1. Suchen vom Anfang nach x (bzw. sicherstellen, daß x vorhanden ist).
2. Löschen von x.
3. Transpositionen.

Unter einer Transposition versteht man das Vertauschen zweier benachbarter Listenelemente.

Beispiel für eine Transposition:
$3 \to 2 \to 5 \to 7 \quad \Rightarrow \quad 3 \to \underbrace{5 \to 2}_{\left(\substack{\text{eine}\\ \text{Transposition}}\right)} \to 7$

Bemerkung: In den folgenden Abschnitten werden verschiedene Heuristiken vorgestellt bzw. miteinander verglichen. Deshalb ist es notwendig, die zulässigen Implementierungen zu beschränken. Beachtet man, daß durchaus doppelt verkettete Listen mit Verweisen auf den Listenanfang bzw. das Listenende verwendet werden können und es bei Transposition im Prinzip egal ist, ob sie am Anfang oder am Ende durchgeführt werden, stellen obige Implementierungen keine wirkliche Einschränkung dar.

4.2 Heuristiken

Definition 4.2.1. (Drei Heuristiken für Transposition)

1. Heuristik **B**: keinerlei Transposition.
2. Heuristik **MF** (move to front): bei `find`(x) und `insert`(x) kommt x durch Transpositionen an die Spitze. `delete`(x) ohne Transposition.
3. Heuristik **TR** (transpose): In `find`(x) und `insert`(x) gelangt x wenn möglich eins nach links (also in Richtung Listenanfang).
4. Heuristik **FC** (frequency counter): Merken im `Zähler`(x), wie oft x nach dem Einfügen gesucht wurde. Die Liste wird nach `Zähler` geordnet.

Beispiel: $\sigma =$ `insert`(1), `insert`(2), `insert`(3), `insert`(4), `find`(1), `find`(4), `find`(3), `find`(4), `find`(2), `find`(4)

Operation	Heuristik			
	B	MF	TR	FC
	()	()	()	()
`insert`(1)	1	1	1	1_1
`insert`(2)	1 2	2 1	2 1	$1_1\ 2_1$
`insert`(3)	1 2 3	3 2 1	2 3 1	$1_1\ 2_1\ 3_1$
`insert`(4)	1 2 3 4	4 3 2 1	2 3 4 1	$1_1\ 2_1\ 3_1\ 4_1$
`find`(1)	1 2 3 4	1 4 3 2	2 3 1 4	$1_2\ 2_1\ 3_1\ 4_1$
`find`(4)	1 2 3 4	4 1 3 2	2 3 4 1	$1_2\ 4_2\ 2_1\ 3_1$
`find`(3)	1 2 3 4	3 4 1 2	3 2 4 1	$1_2\ 4_2\ 3_2\ 2_1$
`find`(4)	1 2 3 4	4 3 1 2	3 4 2 1	$4_3\ 1_2\ 3_2\ 2_1$
`find`(2)	1 2 3 4	2 4 3 1	3 2 4 1	$4_3\ 1_2\ 3_2\ 2_2$
`find`(4)	1 2 3 4	4 2 3 1	3 4 2 1	$4_4\ 1_2\ 3_2\ 2_2$

4.3 Kosten

Definition 4.3.1. Ist A irgendeine Heuristik (d. h. irgendeine Vorschrift die Transpositionen zu setzen), dann ist:

$$\begin{aligned}\text{Kosten}(\texttt{find}_A(x)) = {} & \text{Position von } x \text{ in der Liste} \\ & + \text{Anzahl Transpositionen, wo } x \text{ nicht nach links geht.}\end{aligned}$$

$$\begin{aligned}\text{Kosten}(\texttt{insert}_A(x)) = {} & \text{Anzahl Elemente in Liste} + 1 \\ & + \text{Anzahl Transpositionen, wo } x \text{ nicht nach links geht.}\end{aligned}$$

$$\begin{aligned}\text{Kosten}(\texttt{delete}_A(x)) = {} & \text{Position von } x \text{ in der Liste} \\ & + \text{Anzahl Transpositionen.}\end{aligned}$$

Bemerkung: Die Kosten der Transpositionen von x nach links bei $\texttt{find}(x)$ und $\texttt{insert}(x)$ sind bereits in den Kosten zum Auffinden von x enthalten und werden daher nicht angerechnet.

Satz 4.3.2. Optimalität von MF
Für jede Folge σ von Operationen auf eine leere Liste gilt:

$$\text{Kosten}(\sigma) \text{ unter MF } \leq 2 \cdot \text{Kosten}(\sigma) \text{ unter A},$$

wobei A eine beliebige Heuristik (nach Definition 4.2.1, Definition 4.3.1) ist.

Beweis:
Sei $\sigma = \sigma_1, \sigma_2, \ldots, \sigma_m$ eine Folge von Operationen. Wir betrachten die (beliebige) Heuristik A und MF. Es ergeben sich folgende Listen:

$$\begin{array}{rl}\text{MF:} & S_0 = () \xrightarrow{\sigma_1} S_1 \xrightarrow{\sigma_2} S_2 \xrightarrow{\sigma_3} \cdots \xrightarrow{\sigma_m} S_m \\ \text{A:} & S'_0 = () \xrightarrow{\sigma_1} S'_1 \xrightarrow{\sigma_2} S'_2 \xrightarrow{\sigma_3} \cdots \xrightarrow{\sigma_m} S'_m\end{array}$$

In S_i und S'_i sind dieselben Elemente, höchstens in verschiedener Reihenfolge. Hauptidee ist es, die Inversionen bezüglich S_i und S'_i zu betrachten.

Definition 4.3.3. Sind S und T zwei Listen mit gleichen Elementen. Für $x, y \in S$ (auch $x, y \in T$) und $x \neq y$ sagt man:

$$\begin{aligned}x, y \text{ stellt eine } I\text{nversion von } S, T \text{ dar} \iff {} & ((x \text{ vor } y \text{ in } S) \text{ und } (x \text{ nach } y \text{ in } T) \\ & \text{oder } (x \text{ nach } y \text{ in } S) \text{ und } (x \text{ vor } y \text{ in } T))\end{aligned}$$

Beispiel:
$S = 1 \to 2 \to \cdots \to n$
$T = n \to n-1 \to \cdots \to 1$
hat genau $\binom{n}{2} = \frac{n(n-1)}{2}$ Inversionen.

Zum Vergleich der Kosten betrachtet man das Potential $\Phi(S, T) =$ Anzahl Inversionen von S und T.
Sei t_i die Kosten(σ_i) unter MF und t'_i die Kosten(σ_i) unter A.

$$\begin{array}{rl}\text{MF:} & S_0 = () \xrightarrow[t_1]{\sigma_1} S_1 \xrightarrow[t_2]{\sigma_2} S_2 \xrightarrow[t_3]{\sigma_3} \cdots \xrightarrow[t_m]{\sigma_m} S_m \\ \text{A:} & S'_0 = () \xrightarrow[t'_1]{\sigma_1} S'_1 \xrightarrow[t'_2]{\sigma_2} S'_2 \xrightarrow[t'_3]{\sigma_3} \cdots \xrightarrow[t'_m]{\sigma_m} S'_m\end{array}$$

$$\begin{aligned} \Phi_i &= \Phi(S_i, S_i') \\ a_i &= t_i + (\Phi_i - \Phi_{i-1}) \qquad \text{für } 1 \leq i \leq n \end{aligned}$$

Es ist:
(nicht amortisierte) Kosten(σ) unter MF:

$$\sum_{i=1}^{m} t_i$$

amortisierte Kosten(σ) unter MF:

$$\begin{aligned} \sum_{i=1}^{m} a_i &= \sum_{i=1}^{m} (t_i + (\Phi_i - \Phi_{i-1})) \\ &= t_1 + (\Phi_1 - \underbrace{\Phi_0}_{=0}) + t_2 + (\Phi_2 - \Phi_1) + \quad \cdots \quad + t_m + (\Phi_m - \Phi_{m-1}) \\ &= \sum_{i=1}^{m} t_i + \underbrace{\Phi_m}_{\geq 0} \end{aligned}$$

$$\Rightarrow \sum_{i=1}^{m} t_i \leq \sum_{i=1}^{m} a_i.$$

Bemerkung: Die amortisierten Gesamtkosten stellen immer eine obere Schranke der tatsächlichen Kosten dar, solange für alle i $\Phi_i \geq 0$ gilt.

Es genügt nun zu zeigen: Für alle i ist $a_i \leq 2 \cdot t_i'$. Daraus folgt sofort

$$\sum t_i \leq \sum a_i \leq 2 \cdot \sum t_i'.$$

Für die Idee der Potentialfunktion kann man zunächst folgenden Spezialfall betrachten:
Wenn t_i lang ist und t_i' kurz, z. B. $\sigma_i = \texttt{find}(x)$, $S_{i-1} = \cdots x$ und $S_{i-1}' = x \cdots$, dann verschwinden viele Inversionen bei der Operation σ_i. Damit $a_i \leq 2 \cdot t_i'$ gilt, muß $\underbrace{t_i + (\Phi_i - \Phi_{i-1})}_{a_i} \leq 2 \cdot t_i'$ gelten.

Es ist $\Phi_{i-1} \geq n - 1$ allein wegen der Position von x. Durch das Setzen von x nach vorn bei der Heuristik MF verschwinden $n - 1$ Inversionen. Damit wird a_i klein.

Im Weiteren werden die möglichen Operationen betrachtet (Fallunterscheidung nach σ_i):

1. Fall: $\sigma_i = \texttt{find}(x)$

$$\begin{aligned} \text{MF:} \quad & S_{i-1} \xrightarrow{\sigma_i} S_i \\ \text{A:} \quad & S_{i-1}' \xrightarrow{\sigma_i} S_i' \end{aligned}$$

Es seien k, $j \geq 1$ gegeben durch:

$$\begin{aligned} S_{i-1} \quad &= \quad \underbrace{\cdots}_{k\text{-1 Elemente}} \rightarrow \underbrace{x}_{k\text{-te Stelle}} \rightarrow \qquad \cdots \\ S_{i-1}' \quad &= \quad \underbrace{\cdots}_{j\text{-1 Elemente}} \qquad \rightarrow \underbrace{x}_{j\text{-te Stelle}} \rightarrow \cdots \end{aligned}$$

Dann ist $t_i = k$, $t'_i = j$ + Anzahl Transpositionen, wo x nicht nach links geht.

Zur Ermittlung von a_i wird der Übergang $S_{i-1} \to S_i$, $S'_{i-1} \to S'_i$ aufgeteilt in:

$$\begin{array}{rl} \text{MF:} & S_{i-1} \xrightarrow[a_{i,1}]{} S_i \xrightarrow[a_{i,2}]{} S_i \\ \text{A:} & S'_{i-1} \xrightarrow[a_{i,1}]{} S'_{i-1} \xrightarrow[a_{i,2}]{} S'_i \end{array}$$

1. Finden von x in beiden Listen
2. x in S nach vorn schieben
3. Transpositionen der Heuristik A in S' durchführen

Es sei $a_{i,1} = t_1 + \Phi' - \Phi_{i-1}$. Dann ist $\Phi' - \Phi_{i-1}$ die Potentialänderung, die durch Verschieben von x in S_i nach vorn bewirkt wird.
Es sei nun $a_{i,2} = 0 + \Phi_i - \Phi'$ $(a_i = a_{i,1} + a_{i,2})$.

Es wird zuerst das Verchieben von x mittels der Heuristik MF (Laufzeit $a_{i,1}$) betrachtet:

$$\begin{array}{rll} \text{MF:} & S_{i-1} = \underbrace{\cdots y \cdots}_{k-1} \to x \to \cdots z \cdots & S_i = x \to \cdots y \cdots z \cdots \\ \text{A:} & S'_{i-1} = \underbrace{\cdots z \cdots}_{j-1} \to x \to \cdots y \cdots & S'_{i-1} = \underbrace{\cdots z \cdots}_{j-1} \to x \to \cdots y \cdots \end{array}$$

Wann verschwindet ein Inversion? Die Inversion $\{x, y\}$ verschwindet genau dann, wenn y vor x in S_{i-1} und y nach x in S'_{i-1} ist.

$$p = |\{y \mid y \text{ vor } x \text{ in } S_{i-1},\ y \text{ nach } x \text{ in } S'_{i-1}\}|$$

Wann kommt eine Inversion hinzu? Die Inversion $\{x, z\}$ kommt genau dann hinzu, wenn z vor x in S_{i-1} und S'_{i-1} steht.

$$r = |\{z \mid z \text{ vor } x \text{ in } S_{i-1},\ z \text{ nach } x \text{ in } S'_{i-1}\}|$$

Man kann r ausrechnen:

$$r = k - 1 - p \quad \text{oder auch} \quad r + p = k - 1$$

Damit ist $a_{i,1} = t_i + r - p$. Außerdem gilt $k - 1 - p = r \leq j - 1$, also $k - p \leq j$. Daraus folgt:

$$a_{i,1} \leq 2 \cdot j - 1 \leq 2 \cdot t'_i - 1$$

$$\text{denn} \qquad \begin{aligned} a_{i,1} &= t_1 + r - p \\ &= k + r - p \\ &\leq j + r \\ &\leq 2 \cdot j - 1 \quad (\leq 2 \cdot t'_i - 1). \end{aligned}$$

Jetzt betrachtet man den zweiten Teil, die Transpositionen von A in S' (Laufzeit $A_{i,2}$):

$$\begin{array}{rll} \text{MF:} & S_i = x \to \cdots & S_i = x \to \cdots \\ \text{A:} & S'_{i-1} = \underbrace{\cdots}_{j-1} \to x \to \cdots & S'_i = \cdots \to x \to \cdots \end{array}$$

$$\begin{aligned} a_{i,2} = 0 + \Phi_i - \Phi' \\ &\leq \text{Anzahl Transpositionen, wo } x \text{ nicht beteiligt} \\ &\quad - \text{Anzahl Transpositionen, wo } x \text{ nach links geht} \\ &\quad + \text{Anzahl Transpositionen, wo } x \text{ nach rechts geht} \\ &\leq t'_i - (j-1) \end{aligned}$$

Beachte: x kann $\leq (j-1)$ mal *echt* nach links gehen.

Insgesamt ist nun

$$\begin{aligned} a_i &= a_{i,1} + a_{i,2} \\ &\leq \underbrace{2j-1}_{=a_{i,1}} + \underbrace{t'_i - (j-1)}_{=a_{i,2}} \\ &= t'_i + j \\ &\leq 2 \cdot t'_i \qquad \text{, da } j \leq t'_i \end{aligned}$$

2\. Fall: $\sigma_i = \texttt{insert}(x)$

$$\begin{aligned} \text{MF:} \quad & S_{i-1} \xrightarrow{\sigma_i} S_i \\ \text{A:} \quad & S'_{i-1} \xrightarrow{\sigma_i} S'_i \end{aligned}$$

Es erfolgt wieder die Aufteilung in:

$$\begin{aligned} \text{MF:} \quad & S_{i-1} \longrightarrow (S_i \to x) \longrightarrow S_i = (x \to S_{i-1}) \\ \text{A:} \quad & S'_{i-1} \longrightarrow (S'_i \to x) \longrightarrow S'_i \end{aligned}$$

k =Länge der Liste
$\Phi' = \Phi(S_{i-1}, S'_{i-1})$
$a_{i,1} = t_i + \Phi' - \Phi_{i-1}$
$a_{i,2} = 0 + \Phi_i - \Phi'$
(damit gilt wieder $a_i = a_{i,1} + a_{i,2}$)

zu $a_{i,1}$:

Es ist $a_{i,1} = t_i + \underbrace{(\overbrace{\Phi_{i-1}}^{=\Phi'} - \Phi_{i-1})}_{=0} = k \leq t'_i$.

zu $a_{i,2}$:

$$\begin{aligned} \text{MF:} \quad & S_{i-1} = \underbrace{\cdots}_{S_{i-1}} \to x & & S_i = x \to \underbrace{\cdots}_{S_{i-1}} \\ \text{A:} \quad & S'_{i-1} = \underbrace{\cdots}_{S'_{i-1}} \to x & & S'_i = \underbrace{\cdots}_{j-1} \to \underbrace{x}_{j\text{-te Stelle}} \to \cdots \end{aligned}$$

j = Position von x in S'_i

$$\begin{aligned} a_{i,2} &= 0 + \Phi_i - \Phi' \\ &\leq (j-1) + \text{Transpositionen von } A \text{, wo } x \text{ nicht berührt wird} \\ &= (j-1) + (t'_i - k) \\ &\leq t'_i \qquad \text{, da } (j-1) \leq k \end{aligned}$$

Also $a_i = a_{i,1} + a_{i,2} \leq 2 \cdot t'_i$.

3. Fall: $\sigma_i = \texttt{delete}(x)$ (siehe Übung)

Satz 4.3.4. Es gibt keine Konstante $c > 0$, so daß für jede Heuristik A und jede Operationsfolge σ gilt:

$$\begin{aligned}
\text{Kosten}(\sigma) \text{ der Heuristik B} &\leq c \cdot \text{Kosten}(\sigma) \text{ der Heuristik A,}\\
\text{Kosten}(\sigma) \text{ der Heuristik FC} &\leq c \cdot \text{Kosten}(\sigma) \text{ der Heuristik A oder}\\
\text{Kosten}(\sigma) \text{ der Heuristik TR} &\leq c \cdot \text{Kosten}(\sigma) \text{ der Heuristik A.}
\end{aligned}$$

Beweis:
Der Satz soll für jede Heuristik A gelten. Es genügt also für eine spezielle Heuristik, z. B. MF, ein Beispiel zu finden, wo die Laufzeiten asymptotisch verschieden sind.
zu B:
$\sigma = \texttt{insert}(1), \texttt{insert}(2), \texttt{insert}(3), \ldots, \texttt{insert}(n), \underbrace{\texttt{find}(n), \ \ldots, \texttt{find}(n)}_{m\text{-mal}}$

$$\begin{aligned}
\text{Kosten}(\sigma) \text{ in B} &= 1 + 2 + 3 + \ \cdots \ + (n-1) + n + n \cdot m\\
&= \frac{n(n+1)}{2} + n \cdot m\\
\text{Kosten}(\sigma) \text{ in MF} &= \frac{n(n+1)}{2} + m
\end{aligned}$$

Mit $m = n^2$ folgt:

$$\begin{aligned}
\text{Kosten}(\sigma) \text{ in B} &= \frac{n(n+1)}{2} + n^3\\
\text{Kosten}(\sigma) \text{ in MF} &= \frac{n(n+1)}{2} + n^2 \leq 2 \cdot n^2
\end{aligned}$$

zu FC:

$$\begin{aligned}
\sigma = \texttt{insert}(1),\ &\underbrace{\texttt{find}(1),\ \ldots,\ \texttt{find}(1)}_{(n-1)\text{-mal}},\\
\texttt{insert}(2),\ &\underbrace{\texttt{find}(2),\ \ldots,\ \texttt{find}(2)}_{(n-2)\text{-mal}},\\
\texttt{insert}(3),\ &\underbrace{\texttt{find}(3),\ \ldots,\ \texttt{find}(3)}_{(n-3)\text{-mal}},\\
&\vdots\\
\texttt{insert}(n-1),\ &\texttt{find}(n-1),\\
\texttt{insert}(n)&
\end{aligned}$$

$$\begin{aligned}
\text{Kosten}(\sigma) \text{ in FC} &= 1 + 1 \cdot (n-1) + 2 + 2 \cdot (n-2) + 3 + 3 \cdot (n-3) + \ \cdots\\
&\quad + (n-1) + (n-1) \cdot 1 + n + n \cdot 0\\
&= \sum_{i=1}^{n} i + \sum_{i=1}^{n} i \cdot n - \sum_{i=1}^{n} i^2\\
&= \frac{n(n+1)}{2} + n \cdot \frac{n(n+1)}{2} - \frac{1}{6} n(n+1)(n+2)\\
&= \Omega(n^3)
\end{aligned}$$

$$\begin{aligned}\text{Kosten}(\sigma) \text{ in MF} &= 1 + (n-1) + 2 + (n-2) + 3 + (n-3) + \quad \cdots \qquad +(n-1) + 1 + n + 0 \\ &\leq n \cdot (n-1) \\ &= O(n^2)\end{aligned}$$

zu TR: siehe Übung.

Kapitel 5

Selbstorganisierende Bäume

Die Operationen der Datenstruktur Wörterbuch („Dictionary“ siehe Kapitel 4) benötigen bei einer Implementation mit AVL- oder B-Bäumen nur die Zeit $O(\log n)$. Der AVL-Baum benutzt dazu an den Knoten gespeicherte Zusatzinformationen, z. B. Balancewerte. Der B-Baum verbraucht mehr Platz als nötig, er kann bis zur Hälfte leer sein. Ziel ist es nun einen binären Baum ohne Zusatzinformationen zu schaffen, der die Wörterbuch Operationen amortisiert auch in $O(\log n)$ ausführt und das Zugriffsverhalten irgendwie lernt (ähnlich der MF-Heuristik). Analog zur Transposition bei Listen kann man die Rotation bei Bäumen verwenden.

Problem: Gibt es eine analog zu der Heuristik MF implementierte Operation `find`(y), mit amortisierter Zeit $O(\log n)$?

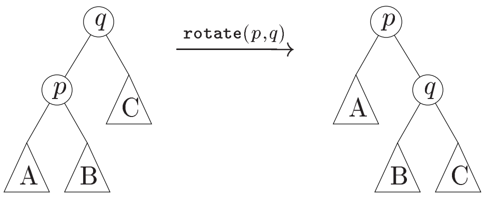
5.1 Algorithmus für geschicktes Rotieren, Splaying

Ist p ein Knoten im Suchbaum SB, der nicht die Wurzel ist. Sei q = `vater`(p).

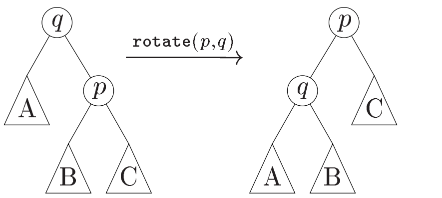
```
splaying(p):

   q := vater(p)
   if q ist Wurzel then
     rotate(p, q)
   else
     r := vater(q)
     if (p, q sind linke Söhne) oder (p, q sind rechte Söhne) then
       rotate(q, r)       // „zig-zig“ Fall
       rotate(p, q)
     else
       rotate(p, q)       // „zig-zag“ Fall
       rotate(p, r)
```

Fälle von `splaying`(b)
Sei q = `vater`(p) und r = `vater`(q)

1. (a) q ist Wurzel, p ist linker Sohn von q

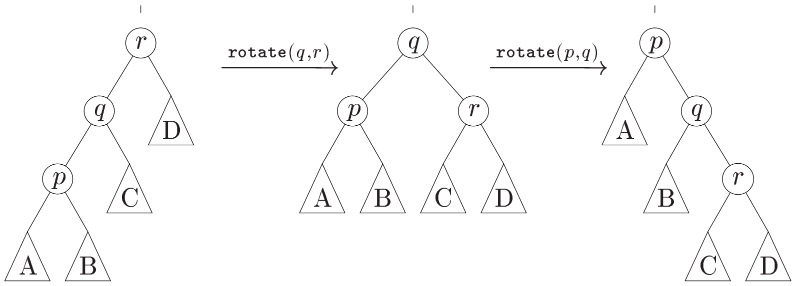


(b) q ist Wurzel, p ist rechter Sohn von q

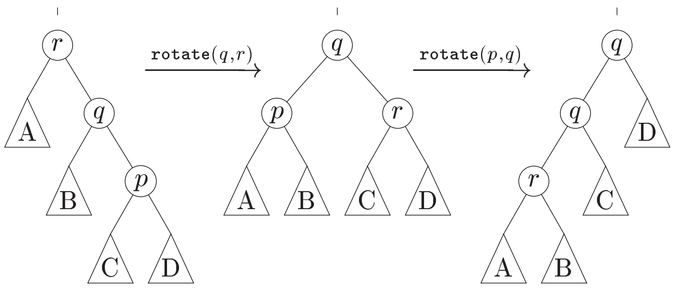


2. q ist nicht die Wurzel

   (a) „zig-zig“ nach links

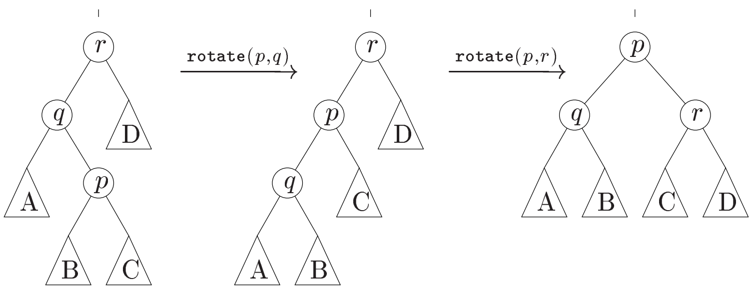


   (b) „zig-zig“ nach rechts

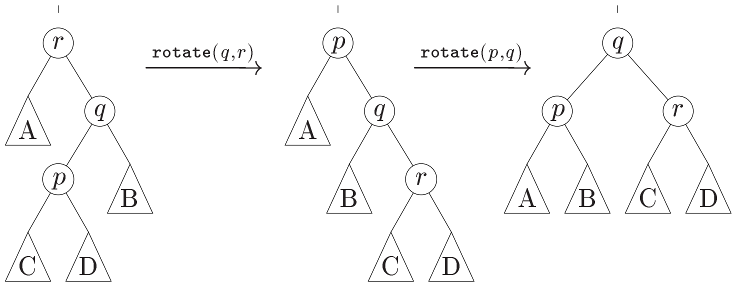


q ist nicht die Wurzel

   (a) „zig-zag“ links-rechts



   (b) „zig-zag“ rechts-linls



5.2 Datenstruktur Splaybaum

Die Struktur ist ein einfacher binärer Suchbaum.

(a) `splay`(x) (dabei ist x ein möglicher Schlüssel, der gespeichert sein kann)

> Suche x im Baum.
> **if** x kommt vor **then** // Dann ist x ein Knoten im Baum
> p := x
> **else**
> p := Vater des Blattes, wo die binäre Suche nach x geendet hat
> **while** x nicht die Wurzel **do**
>   `splaying`(x)

(b) `find`(x)

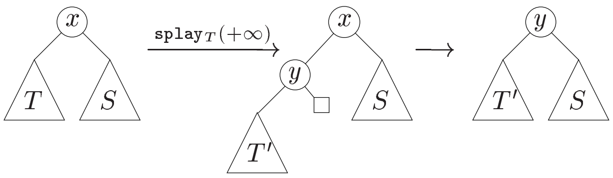
> `splay`(x)
> **if** x ist Wurzel **then**
>   **return** x
> **else**
>   **return** „nicht vorhanden“

Der Baum wird auch durch `splay`(x) umstrukturiert, wenn x nicht gefunden wurde.

(c) `delete`(x)

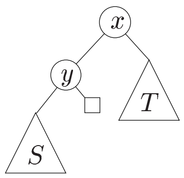
> `splay`(x)
> **if** x ist Wurzel **then** // $\Leftrightarrow$ x ist im Baum enthalten
>   `T` := linker Teilbaum
>   **if** `T` nicht leer **then**
> $\texttt{splay}_\texttt{T}(+\infty)$
>   Hänge rechten Teilbaum unter Wurzel des linken Teilbaumes.

Dabei wird durch $\texttt{splay}_\texttt{T}(+\infty)$ der symmetrische Vorgänger y von x zum linken Sohn von x. (Da die Schlüssel eindeutig sind hatte $\texttt{splay}_\texttt{T}(x)$ dieselbe Wirkung.) Der Knoten y hat keinen rechten Sohn, da y das größte Element im linken Teilbaum ist.

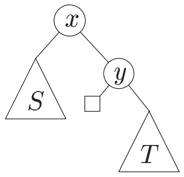


(d) `insert`(x)

> `splay`(x)
> **if** x ist die Wurzel **then**
>   **return** „x vorhanden"
> **else**
> y `:=` Schlüssel der Wurzel   `//` $x \neq y$
> T `:=` rechter Teilbaum
> S `:=` linker Teilbaum
>   **if** $x \gneqq y$ **then**
>
>   Bilde den Baum:



>   **else**   `//` $x \lneqq y$
>
>   Bilde den Baum:

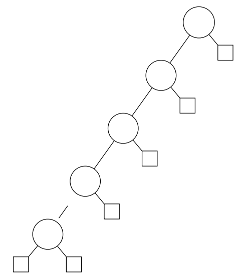


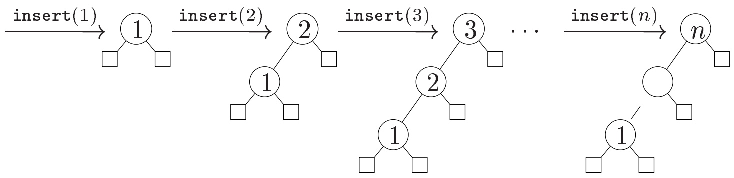
Für eine der obigen Operationen σ auf den Baum S ist

$$K_S(\sigma) = \text{Anzahl der Aufrufe von } \texttt{splaying}.$$

Beachte: Ein `splaying` in Zeit O(1).

Beispiel: Kann bei Start mit dem leeren Baum folgender Baum entstehen?



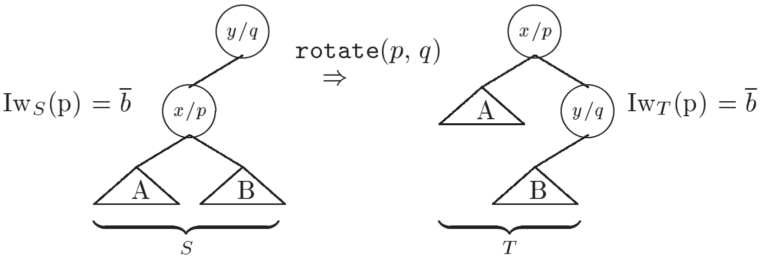


`find`(1) benötigt Zeit O(n), aber die vorherigen n Einfügungen nur O(1). Eine amortisierte Zeit von O($\log n$) ist also möglich.
Ziel: m Dictionary-Operationen auf einen anfangs leeren Baum sollen in Zeit O($m \cdot \log n$) abgearbeitet werden, wobei n ($\gneqq 1$) die Maximalzahl der Elemente im Baum ist.

5.3 Definitionen

Sei S ein binärer Suchbaum. Im folgenden werden die Begriffe Einzelgewicht, totales Gewicht und Rang eines Knotens sowie das Potential eines Suchbaums definiert.

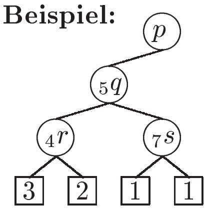
Definition 5.3.1. (Einzelgewicht)
Das *Einzelgewicht* eines Knotens (oder Blattes) p von S ist eine gegebene reelle Zahl $Iw_S(p) \gneqq 0$ (individual weight).
Beachte: $Iw_S(p)$ ist am Knoten fixiert, d. h. insbesondere:



Definition 5.3.2. (Totales Gewicht)
Sei p Knoten eines Suchbaumes S mit Einzelgewichten. Das *totale Gewicht* von p ist:

$$Tw_S(p) = \sum_q Iw_S(q) \qquad (q \text{ ist Knoten im Teilbaum mit der Wurzel } p).$$

Beispiel:

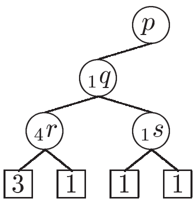


$$Tw_S(q) = 5 + 4 + 7 + 3 + 2 + 1 + 1 = 23$$

Definition 5.3.3. (Rang)
Sei S ein mit Einzelgewichten versehener Baum, dann ist der *Rang* eines Knotens p in S gegeben durch

$$R_S(p) = \lfloor \log_2(Tw_S(p)) \rfloor \ \in \mathbb{N}.$$

Beispiel:



$Tw_S(q) = 12 \quad R_S(q) = 3$
$Tw_S(r) = 8 \quad R_S(r) = 3$
$Tw_S(s) = 3 \quad R_S(s) = 1$

Definition 5.3.4. (Potential)
Das *Potential* eines mit Individualgewichten versehenen Suchbaumes S ist gegeben durch

$$\Phi(S) = \sum_p R_S(p) = \sum_p \lfloor \log_2 Tw_S(p) \rfloor.$$

(p ist ein Knoten von S)

Die amortisierte Zeit einer Operation σ_i $\big(\sigma_i = \texttt{splay}_S(x)$, $\texttt{find}_S(x)$, $\texttt{insert}_S(x)$, $\texttt{delete}_S(x)\big)$ ist gegeben durch:

$$a_S(\sigma_i) = K_S(\sigma_i) + \Phi(S') - \Phi(S) \qquad \text{mit } S \xrightarrow{\sigma_i} S'.$$

S' ist also der Baum, der durch σ_i ausgeführt auf S entsteht.

5.4 Schlüssellemma

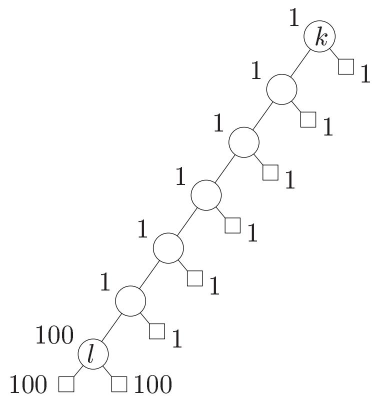
Lemma 5.4.1. (Schlüssellemma)
Sei S ein binärer Suchbaum mit der Wurzel k und sei S' definiert durch $S \xrightarrow{\texttt{splay}_S(x)} S'$. Sei l die Wurzel von S' (kommt x nicht vor, dann ist $l \neq r$). Dann gilt:

$$a_S(\texttt{splay}_S(x)) \leq 1 + 3 \cdot \big(R_S(k) - R_S(l)\big).$$

Beachte:

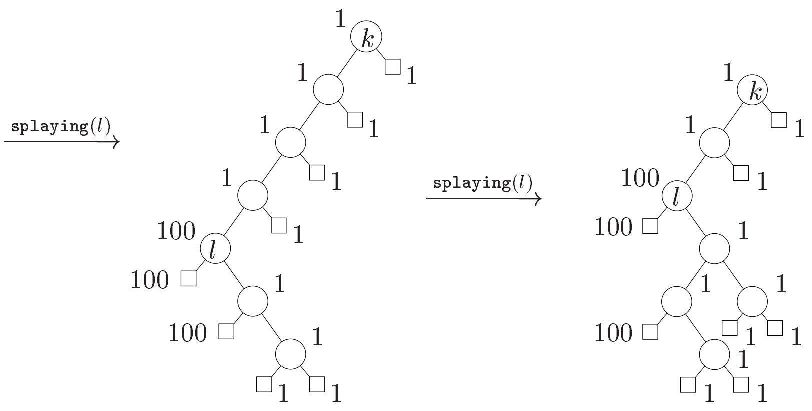
$$a_S(\texttt{splay}_S(x)) = K_S(\texttt{splay}_S(x)) + \Phi(S') - \Phi(S)$$
$$R_S(k) - R_S(l) \geq 0$$

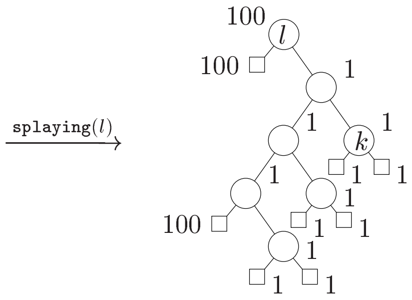
Beispiel: Im folgenden Beispiel sind die tatsächlichen Kosten hoch, die amortisierten Kosten von $3 \cdot (R_S(k) - R_S(l))$ jedoch klein.
S:



$R_S(k) = \lfloor \log_2(300 + 12) \rfloor = 8$
$R_S(l) = \lfloor \log_2 300 \rfloor = 8$
dann ist $R_S(k) - R_S(l) = 0$.

Sei x der Schlüsselwert von l. $\texttt{splay}_S(x)$ führt zu:





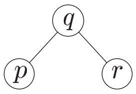
Es ist $K_S(\texttt{splay}_S(x)) = 3$. Es muß $a_S(\texttt{splay}_S(x)) \leq 1$ gelten, also ist $\Phi(S') - \Phi(S) \leq 2$.

$$\begin{aligned}
\Phi(S) &= \sum_p R_S(p) \\
&= \underbrace{\lfloor \log_2 100 \rfloor + \lfloor \log_2 100 \rfloor + 6 \cdot \lfloor \log_2 1 \rfloor}_{\text{Blätter von } S} \\
&\quad + \underbrace{\lfloor \log_2 300 \rfloor + \lfloor \log_2 302 \rfloor + \;\cdots\; + \lfloor \log_2 312 \rfloor}_{\text{alle inneren Knoten}} \\
&= 6 + 6 + 0 + 7 \cdot 8 \\
&= 68
\end{aligned}$$

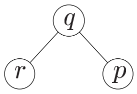
$$\begin{aligned}
\Phi(S') &= \underbrace{2 \cdot \lfloor \log_2 100 \rfloor + 6 \cdot \lfloor \log_2 1 \rfloor}_{\text{Blätter}} \\
&\quad + \underbrace{3 \cdot \lfloor \log_2 3 \rfloor + \lfloor \log_2 104 \rfloor + \lfloor \log_2 108 \rfloor + \lfloor \log_2 112 \rfloor + \lfloor \log_2 312 \rfloor}_{\text{alle inneren Knoten}} \\
&= 2 \cdot 6 + 0 + 3 + 18 + 8 \\
&= 41
\end{aligned}$$

Also ist $a(\texttt{splay}_S(x)) = 3 + \underbrace{41}_{\Phi(S')} - \underbrace{68}_{\Phi(S)} \leq 1$.

Satz 5.4.2. (Eigenschaften des Ranges)
Ist q Vater von p im Baum T mit Individualgewichten und r Bruder von p, also:



oder



, dann gilt:

- Eigenschaft 1 des Ranges:
  (i) $R_T(q) \geq R_T(p)$, $R_T(q) \geq R_T(r)$
  (ii) $R_T(q) \gneqq R_T(p) \;\Leftrightarrow$
  Es gibt ein $n \geq 0$, so daß $Tw_T(p) \lneqq 2^n$ und $Tw_T(q) \geq 2^n$.
- Eigenschaft 2:
  Ist $R_T(p) \geq R_T(r)$, dann ist $R_T(q) \geq 1 + R_T(r)$.

Beweis:
Eigenschaft 1 (i) und (ii) gelten offensichtlich.

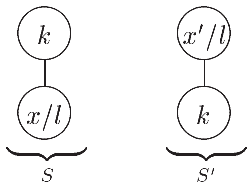
Eigenschaft 2:
Dann ist $Tw_T(p) \geq Tw_T(r)$

Dann $Tw_T(q) = \underbrace{Iw_T(q)}_{\geq 0} + Tw_T(p) + Tw_T(r) \geq 2 \cdot Tw_T(r)$.

$$\Rightarrow \quad \log_2 Tw_T(q) \geq \log_2 2 \cdot Tw_T(r) = 1 + log_2 Tw_T(r).$$

$$\begin{aligned} \Rightarrow \quad R_T(q) &= \lfloor \log_2 Tw_T(q) \rfloor \\ &\geq \lfloor 1 + \log_2 Tw_T(r) \rfloor \\ &= 1 + \lfloor \log_2 Tw_T(r) \rfloor \\ &= 1 + R_T(r) \end{aligned}$$

Nun kann der eigentliche Beweis von $a(\texttt{splay}_S(x)) \leq 1 + 3 \cdot (R_S(k) - R_S(l))$ betrachtet werden:



$x' = x$ oder $x' =$ ein nächster Schlüssel zu x in S

Zunächst untersucht man den Sonderfall $k = l$, d. h. es hat sich nichts geändert, dann ist $\Phi(S') = \Phi(S)$.
$K(\texttt{splay}_S(x)) = 1$, dann $a(\text{Splay}_S(x)) = 1 = 1 + 3 \cdot \underbrace{(R_S(k) - R_S(l)}_{=0,\text{ da } k=l}$.

Sei ab jetzt $k \neq l$, dann ist $\texttt{splay}_S(x)$ eine Folge von m (≥ 1) $\texttt{splaying}$-Operationen. Seien die Bäume $S_0 (= S)$, $S_1, \ldots, S_m$ gegeben durch:

$$S_0 \xrightarrow{1.\ \texttt{splaying}(l)} S_1 \xrightarrow{2.\ \texttt{splaying}(l)} S_2 \xrightarrow{3.\ \texttt{splaying}(l)} \cdots \xrightarrow{m.\ \texttt{splaying}(l)} S_m$$

$a(\texttt{splay}_S(x))$ wird jetzt auf die einzelnen $\texttt{splaying}$-Operationen aufgeteilt:

$$\begin{aligned} a(\texttt{splay}_S(x)) &= \sum_{i=1}^{m} a_i \\ &= \sum_{i=1}^{m} \big(1 + \Phi(S_i) - \Phi(S_{i-1})\big) \end{aligned}$$

Jetzt ist zu zeigen:

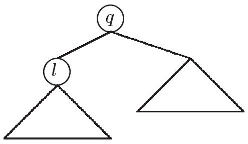
$$\begin{aligned} a_i &\leq 3 \cdot (R_{S_i}(l) - R_{S_{i-1}}(l)) \qquad \text{für alle } i \text{ mit } 1 \leq i \leq m-1 \\ a_m &\leq 3 \cdot (R_{S_m}(l) - R_{S_{m-1}}(l)), \end{aligned}$$

denn daraus folgt die Behauptung.

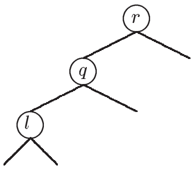
$$\begin{aligned} a(\texttt{splay}(x)) &= \sum_{i=1}^{m} a_i \\ &\leq 3 \cdot (R_{S_1}(l) - R_{S_0}(l)) + \;\cdots\; + 3 \cdot (R_{S_{m-1}}(l) - R_{S_{m-2}}(l)) \\ &\quad + 1 + 3 \cdot (R_{S_m}(l) - R_{S_{m-1}}(l)) \\ &= 1 + 3 \cdot (R_{S_m}(l) - R_{S_0}(l)) \\ &= 1 + 3 \cdot (R_S(l) - R_S(l)) \end{aligned}$$

Sei also i mit $1 \leq i \leq m$ fest. Wir betrachten also das i-te `splaying`(l). Folgende Fälle in S_{i-1} sind zu unterscheiden:

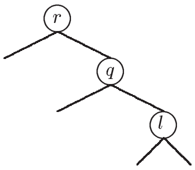
1. Fall:



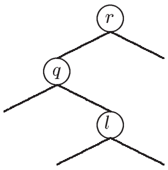
2. Fall:



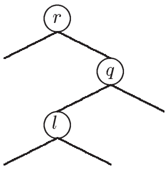
oder



3. Fall:



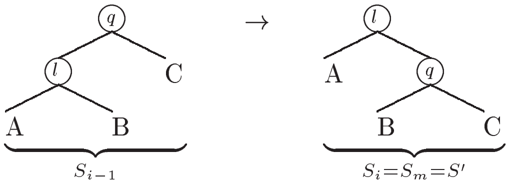
oder



Beweis:
Sei ab jetzt:
$q =$ Vater von l
$r =$ Grossvater von l (falls ein Großvater von l existiert).
einige Ränge:

$$\underbrace{\begin{matrix} R_l = R_{S_{i-1}}(l) \\ R_q = R_{S_{i-1}}(q) \\ R_r = R_{S_{i-1}}(r) \end{matrix}}_{\text{alte Ränge}} \qquad \underbrace{\begin{matrix} R'_l = R_{S_i}(l) \\ R'_q = R_{S_i}(q) \\ R'_r = R_{S_i}(r) \end{matrix}}_{\text{neue Ränge}}$$

1. Fall: Hier ist $i = m$ (r nicht definiert). Was geschieht?

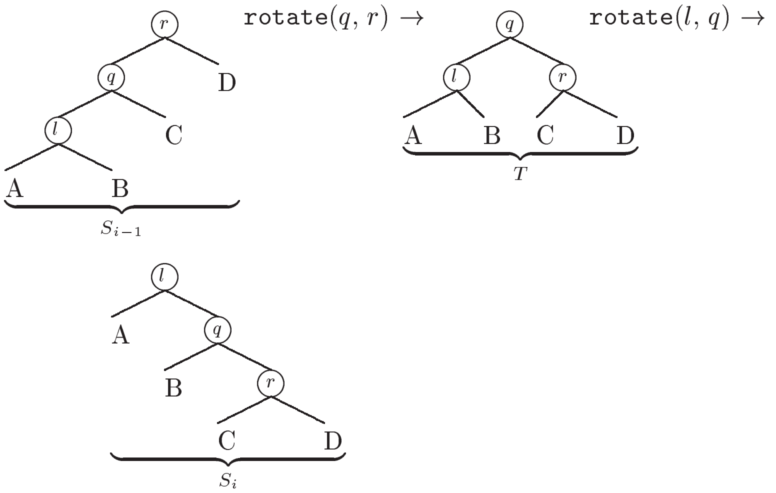


Es ist:
$R'_l = R_q$
$\Phi(S_{m-1}) = R_q + R_l+$ seine Ränge in A B C
$\Phi(S_m) = R'_l + R'_q+$ seine Ränge in A B C

Weiter folgt:

$$
\begin{aligned}
a_i &= a_m \\
&= 1 + \Phi(S_m) - \Phi(S_{m-1}) \\
&= 1 + R'_l + R'_q - R_l - R_q && \text{innere Teilbäume ändern sich nicht} \\
&= 1 + R'_q - R_l && \text{da } R'_l = R_q \\
&\le 1 + R'_l - R_l && \text{da } R'_l \ge R'_q \\
&\le 1 + 3 \cdot \underbrace{(R'_l - R_l)}_{>0} && \text{mit } R'_l \ge R_l \text{, da } R'_l = R'_q \ge R_l
\end{aligned}
$$

2. Fall:



$R_T(l) = R_l$
$R_T(q) = R_r = R'_l$
$R_T(r) = R'_r$

$$
\begin{aligned}
a_i &= 1 + \Phi(S_i) - \Phi(S_{i-1}) \\
&= 1 + R'_l + R'_q + R'_r - R_l - R_q - R_r \\
&= 1 + R'_q + R'_r - R_l - R_q && R'_l = R_r
\end{aligned}
$$

(a) $R'_l = R_r \gneqq R_l$
Dann folgt:

$$
\begin{aligned}
a_i &= 1 + R'_q + R'_r - R_l - R_q && \text{siehe oben} \\
&\le 1 + R'_l + R'_l - R_l - R_l && \text{da } R'_q \le R'_l,\ R'_r \le R'_l \text{ und } R_l \le R_q \\
&= 1 + 2 \cdot (R'_l - R_l) \\
&\le 3 \cdot (R'_l - R_l)
\end{aligned}
$$

Denn es ist nach Annahme $R'_l \gneqq R_l$, also $R'_l \ge R_l + 1$ da Ränge ganze Zahlen sind.

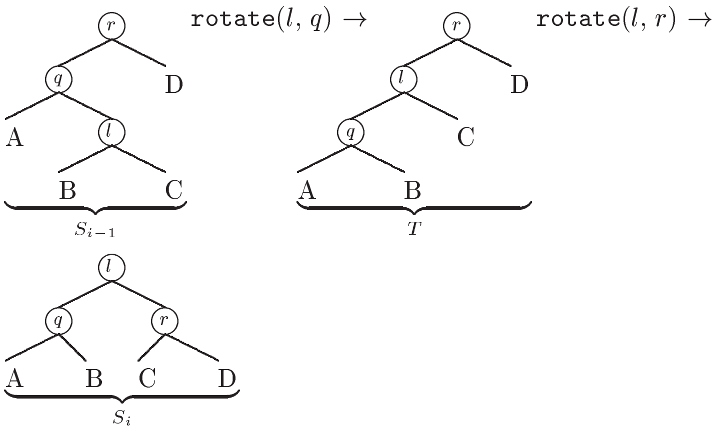
(b) $R'_l = R_r = R_l$ (Fall 2c: $R'_l \lneqq R_l$ kann nicht sein)
Zunächst gilt nach Eigenschaft (i): $R_l \le R_q \le R_r = R'_l$, also ist $R_l = R_q = R_r = R'_l$ nach Annahme.
Nun gilt $R'_r \lneqq R_l$, denn wenn $R'_r \ge R_l$ gelten würde, dann wäre $R'_l = R_T(q) \gneqq R_l$. Wegen Eigenschaft (ii) ist das aber nicht möglich.

Dann folgt:

$$\begin{aligned}
a_i &= 1 + R'_q + R'_r - R_l - R_q \\
&= 1 + \underbrace{(R'_q - R_q)}_{\leq 0} + (R'_r - R_l) && \text{da } R'_q \leq R'_l = R_l = R_q \text{ ist } R'_q - R_q \leq 0 \\
&\leq 1 + 0 + \underbrace{(R'_r - R_l)}_{\leq -1} && \text{da } R'_r \lneqq R_l \text{ ist } R'_r - R_l \lneqq 0\text{, also } \leq -1 \\
&\leq 1 + 0 - 1 = 0 \\
&= 3 \cdot (R'_l - R_l)
\end{aligned}$$

Der Beweis für den Baum in andere Richtung erfolgt analog.

3. Fall:



(a) $R'_l = R_r \gneqq R_l$

Dann ist

$$\begin{aligned}
a_i &= 1 + R'_q + R'_r - R_q - R_l \\
&\leq R'_l + R'_l - R_l - R_l \\
&= 1 + 2\underbrace{(R'_l - R_l)}_{\geq 1} \\
&\leq 3 \cdot (R'_l - R_l).
\end{aligned}$$

(b) $R'_l = R_r = R_l$ Dann gilt: $R'_l = R_l \leq R_q \leq R_r = R'_l$ also $R'_l = R_l = R_q = R_r$.
Dann folgt: $R'_q \lneqq R_q$ oder $R'_r \lneqq R_r$

Auf jeden Fall gilt $R'_q \leq R_q$ und $R'_r \leq R_r$.
Würde $R'_q \lneqq R_q$ oder $R'_r \lneqq R_r$ nicht gelten, dann wäre $R'_q = R_q$ oder $R'_r = R_r$. Aus $R'_q = R_q$ oder $R'_r = R_r$ folgt aber $R'_r = R'_q$, da $R'_l = R_q = R_r = R_l$. Nach Eigenschaft (ii) ist jetzt $R'_l \gneqq R'_r = R_r = R_l$. Das ist ein Widerspruch zu $R'_l = R_l$.

Also gilt $R'_q \lneqq R_q$ oder $R'_r \lneqq R_r$.

$$\begin{aligned} a_i &= 1 + R'_q + R'_r - R_q - R_l \\ &= 1 + (R'_q - R_q) + (R'_r - R_l) \\ &= 1 + \underbrace{\underbrace{(R'_q - R_q)}_{\le 0} + \underbrace{(R'_r - R_r)}_{\le 0}}_{\le 1,\text{ da } R'_q \lneqq R_q \text{ oder } R'_r \lneqq R_r} \\ &= 3 \cdot (R'_l - R_l). \end{aligned}$$

Satz 5.4.3.

(a) Sei $\sigma = \sigma_1, \sigma_2, \ldots, \sigma_m$ eine Folge von m Operationen `delete`, `insert` und `find`. σ wird auf S_0, den anfangs leeren Baum ausgeführt. Sei n die maximale Anzahl von Elementen im Baum, dann gilt:

$$\text{Kosten}(\sigma) = O(m \cdot \log n) \qquad (= \text{ Summe der Einzelkosten}).$$

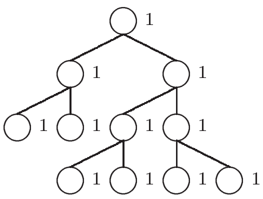
(b) Ist S_0 schon mit n Elementen ausgestattet, so ist:

$$\text{Kosten}(\sigma) = O((m+n) \cdot \log n) \qquad (= O(m \cdot \log n), \text{ falls } m \ge n).$$

Beweis:

(a) Wir wählen als individuelles Gewicht immer 1. Es ist also $Iw(p) = 1$ für alle Knoten p. Dann gilt für jeden Suchbaum S mit $\le n$ Elementen und p Knoten von S, daß $Tw_S(p) \le 2n + 1$ (n innere Knoten und $n+1$ Blätter).

Beispiel mit $n = 5$:



Also 5 innere Knoten und 6 Blätter.

Deshalb gilt:

$$\begin{aligned} R_S(p) &= \lfloor \log_2(Tw(p)) \\ &\le \log_2(2n+1) \\ &\le \log_2(2n) + 1 \\ &\le \log_2(n) + 2. \end{aligned}$$

Nach Lemma 5.4.1 ist

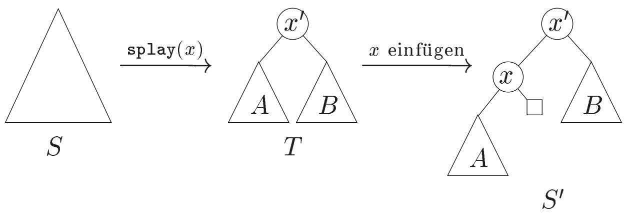
$$\begin{aligned} a(\mathtt{splay}_S(x)) &\le 1 + 3 \cdot (R_S(k) - \underbrace{R_S(l)}_{\le 1}) \\ &\le 1 + 3 \cdot (R_S(k) - 1) \\ &\le 1 + 3 \cdot (\log(n) + 1) \\ &= 4 + 3 \cdot \log n. \end{aligned}$$

Also:

$$a(\texttt{find}_S(x)) \leq 1 + 3 \cdot (\log(n) + 1)$$

nach Definition von $\texttt{find}(x)$.

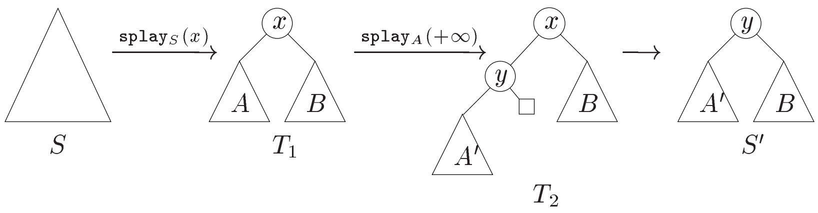
$\texttt{insert}_S(x)$



Also

$$\begin{aligned}
a(\texttt{insert}_S(x)) &= \text{Kosten}(\texttt{insert}_S(x)) + \Phi(S') - \Phi(S) \\
&= \text{Kosten}\underbrace{(\texttt{splay}_S(x)) + \Phi(T) - \Phi(S)}_{a(\texttt{splay}_S(x)}+\Phi(S') - \Phi(T) \\
&\leq 4 + 3 \cdot \log n + \Phi(S') - \Phi(T) \\
&\leq 4 + 3 \cdot \log n + R_{S'}(x) \qquad \text{nach Definition 5.3.4} \\
&\leq 6 + 4 \cdot \log n.
\end{aligned}$$

$\texttt{delete}_S(x)$



Also

$$\begin{aligned}
a(\texttt{delete}_S(x)) &= \text{Kosten}(\texttt{delete}_S(x)) + \Phi(S') - \Phi(S) \\
&= \text{Kosten}(\texttt{splay}_S(x)) + \Phi(T_1) - \Phi(S) \\
&\quad + \text{Kosten}(\texttt{splay}_A(+\infty)) + \Phi(T_2) - \Phi(T_1) \\
&\quad + \underbrace{\Phi(S') - \Phi(T_2)}_{\leq 0} \\
&\leq 4 + 3 \cdot \log_2(n) + 4 + 3 \cdot \log_2(n) \\
&= 8 + 3 \cdot \log_2 n.
\end{aligned}$$

Also insgesamt

$$\begin{aligned}
\text{Kosten}(\sigma) &= \sum_{i=1}^{m} \text{Kosten}(\sigma_i) \\
&= \text{Kosten}(\sigma_1) + \Phi(S_1) - \Phi(S_0) + \text{Kosten}(\sigma_2) + \Phi(S_2) - \Phi(S_1) + \cdots + \text{Kosten}(\sigma_m) + \Phi(S_m) - \Phi(\\
&= \sum_{i=1}^{m} (a(\sigma_i)) - \Phi(S_m) + \Phi(S_0) \\
&\leq \sum_{i=1}^{m} (a(\sigma_i)) \\
&\leq m \cdot (8 + 6 \cdot \log n) \\
&\leq 7m \cdot \log n \qquad \text{für } m,\, n \text{ groß genug.}
\end{aligned}$$

(b) siehe Übung.

Satz 5.4.4. (Lernfähigkeit der Splay-Bäume, vgl. optimale Suchbäume)
Seien S, S_0 zwei beliebige binäre Suchbäume für dieselbe Menge von n Elementen. Sei σ eine Folge von `find`-Operationen (wobei die gesuchten Elemente im Baum enthalten sind). St-Kosten$_S(\sigma)$ sind die Kosten der Ausführung der `find`-Operationen im normalen (statischen) Suchbaum S (etwa auch der optimale).
Dann gilt:

$$(\texttt{splay-})\text{Kosten}_{S_0}(\sigma) \;=\; O(\text{St-Kosten}_S(\sigma) + \underbrace{h \cdot n}_{\text{„Lernbarkeit“}}),$$

wobei h die Höhe von S_0 ist. (Falls σ sehr lang ist, dann ist $h \cdot m \leq$ St-Kosten$_S(\sigma)$ und man erhält O(St-Kosten$_S(\sigma)$).

Kapitel 6

Diskrete Fourier Transformation

Komlexe Zahlen: $z \in \mathbb{C}$,
$z = \underbrace{x}_{Realteil} + \underbrace{i \cdot y}_{Imaginärteil}$, wobei x,y$\in \mathbb{R}$
Darstellung:
im karthesischen Koordinatensystem (Ebene der komplexen Zahlen)

als Polarkoordinaten

Polarkoordinaten nach karthesische Koordinaten:
$z = x + i \cdot y = r \cdot (\cos\varphi + i \cdot \sin\varphi)$
Man fordert $z = 0$, $\varphi :\quad 0 \leq \varphi \leq 360$ (oder im Bogenmaß $0 \leq \varphi \leq 2\pi$)
Man nennt $\varphi = \arg z$, $\quad r = |z|$ der Betrag von z
In $\mathbb{R}$ gilt für $e^x \quad (x \in \mathbb{R})$
$(e^1 = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + ...)$

$$e^x = 1 + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + ...$$

also stellt x daß Bogenmaß eines Winkels dar, so ist:

$$\begin{aligned}
\sin x &= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - ... = \sum_{i \geq 0} \frac{(-1)^i \cdot x^{2i+1}}{(2i+1)!} \\
\cos x &= 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - ... = \sum_{i \geq 0} \frac{(-1)^i \cdot x^{2i}}{(2i)!}.
\end{aligned}$$

Schematische Übertragung auf komplexe Zahlen:
$e^z = \sum_{j \geq 0} \frac{z^j}{j!}$
$\sin z = \sum_{j \geq 0} \frac{(-1)^j \cdot x^{2j+1}}{(2j+1)!}$
$\cos z = \sum_{j \geq 0} \frac{(-1)^j \cdot x^{2j}}{(2j)!}$

Betrachten einmal für $z \in \mathbb{C}$:

$e^{i \cdot z} = \cos z + i \cdot \sin z$
Es ist immer $z' = i \cdot \underbrace{\frac{z'}{i}}_{=z}$
z.B.: $z \in \mathbb{R}$, (also $i \cdot z = 0 + i \cdot z$)

$$\begin{aligned}
e^{i \cdot z} &= 1 + \frac{i \cdot z}{1!} - \frac{z^2}{2!} - \frac{i \cdot z^3}{3!} + \frac{z^4}{4!} + \frac{i \cdot z^5}{5!} - ... \\
\cos z &= 1 - \frac{z^2}{2!} + \frac{z^4}{4!} - \frac{z^6}{6!} + \frac{z^8}{8!} - ... \\
i \cdot \sin z &= i \cdot (z - \frac{z^3}{3!} + \frac{z^5}{5!} - \frac{z^7}{7!} + \frac{z^9}{9!} - ...) \\
&= i \cdot z - \frac{i \cdot z^3}{3!} + \frac{i \cdot z^5}{5!} - \frac{i \cdot z^7}{7!} + \frac{i \cdot z^9}{9!} - ...)
\end{aligned}$$

Also $e^{i \cdot z} = \cos z + i \cdot \sin z$ gilt für $z \in \mathbb{R}$ aber auch $z \in \mathbb{C}$
Dann folge:

$$\begin{aligned}
\cos z &= \frac{1}{2}(e^{i \cdot z} + e^{-i \cdot z}) \\
\sin z &= \frac{1}{2 \cdot i}(e^{i \cdot z} - e^{-i \cdot z})
\end{aligned}$$

für alle $z \in \mathbb{C}$.

Zusammenhang Polar- und karthesische Koordinaten

$$z = x + i \cdot y = r \cdot e^{i \cdot \overbrace{\varphi}^{Bogenmaß}} = r \cdot (\cos \varphi + i \cdot \sin \varphi)$$

$\varphi = \arg z$, $r = |z| = \sqrt{x^2 + y^2}$

Multiplikation: $z \cdot z' = (x + i \cdot y) \cdot (x' + i \cdot y') = (r \cdot e^{i \cdot \varphi}) \cdot (r' \cdot e^{i \cdot \psi}) = r \cdot r' \cdot e^{i \cdot (\varphi + \psi)}$
Es ergeben sich die Additionetheoreme für sin und cos:
$e^{i \cdot \varphi} \cdot e^{i \cdot \psi} = e^{i \cdot (\varphi + \psi)}$, dann folgt:
$(\cos \varphi + i \cdot \sin \varphi) \cdot (\cos \psi + i \cdot \sin \psi) = \cos(\varphi + \psi) + i \sin(\varphi + \psi))$
Dann folgt:
$\cos \varphi \cdot \cos \psi - \sin \psi \cdot \sin \varphi = \cos(\varphi + \psi)$
Dann folgt:
$\cos \varphi \cdot \cos \psi + \cos \varphi \cdot \sin \varphi + i \cdot \sin \psi \cdot \cos \psi - \sin \varphi \cdot \sin \psi = \cos(\varphi + \psi) + i \cdot \sin(\varphi + \psi)$
Betrachten wir weiter: Für $z \in \mathbb{C}$
$e^{z + 2\pi i} = e^z + e^{2\pi i = e^z} = \underbrace{\cos 2\pi}_{=1} + \underbrace{i \cdot \sin 2\pi}_{=0}$
(weil die e-Funktion periodisch ist, wenn wir in die imaginäre Richtung gehen).

Die Schnelle Fourier-Transformation:
3 Arten der Darstellung von komplexen Zahlen

1. $z = x + i \cdot y$ karthesische Darstellung
   $x, y \in R$

2. $z = r \cdot (\cos\varphi + i \cdot \sin\varphi)$ Polarkoordinaten
   $z \neq 0, \quad 0 \leq \varphi \leq 2\pi, \quad r = |z|, \quad \varphi = \arg z$

3. $z = r \cdot e^{i\cdot\varphi}$
   $r = |z| \quad \varphi = \arg z$

Wenn wir haben $e^{i\cdot\varphi} = \cos\varphi + i \cdot \sin\varphi$
zur e-Funktion im Komplexen: $z \to e^z$ wobei $z \in \mathbb{C}$
Ist $z = x + i \cdot y$, dann ist
$e^z = e^{x+i\cdot y} = e^x \cdot e^{i\cdot y}$
$= e^x \cdot (\cos y + i \cdot \sin y)$
$= e^x \cdot \cos y + e^x \cdot i \cdot \sin y)$ (Beachte: $e^{z+2\pi i} = e^{x+i(y+2\pi)}$)
$= e^x \cdot \cos y + e^x \cdot i \cdot \sin y = e^z$
da sin und cos periodisch

$e^z = e^{x+i\cdot y} = e^x(\cos y + i \cdot \sin y)$

Definition (n-te Einheitswurzel) Ein $z \in \mathbb{C}$ ist eine n-te Einheitswurzel $\Leftrightarrow z^n = 1$.

Bsp: (Einheitskreis in der Ebene der komplexen Zahlen)
1-te Einheitswurzel $1^1 = 1$
2-te Einheitswurzeln $1^2 = 1$, $(-1)^2 = 1$
4-te Einheitswurzeln $i^4 = i^2 \cdot i^2 = (-1) \cdot (-1) = 1$
weitere 4-te Einheitswurzeln $(-1)^4 = 1$, $1^4 = 1$, $-i^4 = (-1)^4 \cdot i^4$.

Erinnerung: Multiplikation bei Exponentialdarstellung
$e^{i\cdot\varphi} * e^{i\cdot\psi} = e^{i\cdot(\varphi+psi)}$
$i = e^{i\cdot\frac{\pi}{2}} = \cos\frac{\pi}{2} + i \cdot \sin\frac{\pi}{2}$
$-1 = e^{i\cdot\pi} = \underbrace{\cos\pi}_{=-1} + i \cdot \underbrace{\sin\pi}_{=0}$
Multiplikation von Zahlen mit Betrag gleich 1 führt zur Addition der Winkel.

Satz:
für jedes $n \in \mathbb{N}$ gibt es genau n viele n-te Einheitswurzeln. Diese sind:
$e^0,\ e^{i\frac{2\pi}{n}},\ e^{i\frac{2\cdot 2\pi}{n}} = (e^{i\frac{2\pi}{n}})^2, ...,\ e^{i\frac{n-1\cdot 2\pi}{n}}$.
Das sind n-te Einheitswurzeln, denn $(e^{i\frac{k\cdot 2\pi}{n}})^n = e^{i\cdot k\cdot 2\pi} = 1$.
$\omega_n = e^{i\frac{2\pi}{n}}$ ist die primitive n-te Einheitswurzel (alle anderen lassen sich durch Potenzierung gewinnen). Das sind alle Einheitswurzeln, denn Einheitswurzeln und Nullstellen von $P(z) = z^n - 1$ welches genau n Nullstellen hat.

Satz: Komplexe n-te Einheitswurzeln (n fest) bilden eine Gruppe unter Multiplikation dar.

Beweis:
Multiplikation ist so definiert:
$\omega_n^j \cdot \omega_n^k = \omega_n^{j+k} = \omega_n^{(j+k) mod\ n}$.
Damit Abgeschlossenheit der n-ten Einheitswurzeln unter Multiplikation.
Einselement 1
$\omega_n^{-1} = \omega_n^{n-1} =$
Dann ist $\omega_n \cdot \omega_n^{n-1} = \omega_n^n = 1$
$(\omega_n^k)^{-1} = (\omega_n^{-1})^k = (\omega_n^{n-1})^k$
also $\omega_n^k \cdot (\omega_n^k)^{-1} = \omega_n^k \cdot (\omega_n^{n-1})^k = \omega_n^{(1+n-1)\cdot k} = 1$

Lemma: (Cancellation Lemma) Für alle ganzen Zahlen $n \geq 0,\ k \geq 0,\ d <\neq 0$ ist
$(\omega_{d\cdot n}^d)^k = \omega_{d\cdot n}^{d\cdot k} = \omega_n^\varphi$
Bsp: $d = 2$
$\omega_{dn}^{dk} = \omega_n^k$

Beweis: Es ist $\omega_{d\cdot n} = e^{i\frac{2\pi}{d\cdot n}}$ also:
$\omega_{d\cdot n}^{d\cdot k} = e^(i\frac{2\pi\cdot d\cdot k}{d\cdot n}) = e^(i\frac{2\pi\cdot k}{n}) = \omega_n^k$
Korollar: Ist $n > 0$ gerade Zahl, so $\omega_n^{\frac{n}{2}} = \omega_2 = -1$
Beweis: $\omega_n^{\frac{n}{2}} = (e^{\frac{2\pi i}{n}})^{\frac{n}{2}} = e^{\pi i} = e^{\frac{2\pi i}{2}} = \omega_2$
Bemerkung: Bei n gerade ist -1 eine n-te Einheitswurzel.
(Beachte bei 3 haben wir: $\omega_1 = e^{\frac{0\cdot 2\pi}{3}} = e^{\frac{3\cdot 2\pi}{3}},\quad \omega_2 = e^{\frac{1\cdot 2\pi}{3}},\quad \omega_3 = e^{\frac{2\cdot 2\pi}{3}},\quad$).

Lemma: (Halbierungslemma)
Ist $n > 0$, n gerade, dann sind die Quadrate der n'ten Einheitswurzeln gerade die $\frac{n}{2}$ vielen $\frac{n}{2}$-ten Einheitswurzeln. Jede dann genau 2 mal:
Beweis: (mit Cancelation Lemma)
Es ist $(\omega_n^k)^2 = (\omega_n^{2k}) = (\omega_{2\frac{n}{2}}^{2k}) = (\omega_{\frac{n}{2}}^k)$
für $k \geq 0$: Es ist $(\omega_n^{k+\frac{n}{2}})^2 = (\omega_n^k)^2$ (d.h. sie haben das gleiche Quadrat).

Lemma: (Summationslemma)
Für jedes $n \geq 1,\ k \geq 0,\ k$ nicht Vielfaches von n $(k \neq n,\ k \neq 2n,\ k \neq 3n,\ ...)$ gilt:
$\sum_{j=0}^{n-1} (\omega_n^k)^j = 0.$
$h = 1 \Rightarrow$ Summe aller Einheitswurzeln ist 0.
Bsp: n=2, k=1, 1−1=0 ; n=2, k=2, ! ; n=2, k=3, 1−1=0 ; n=4, i−1−i+1=0
Beweis: (geometrische Reihe)
$\sum_{n=0}^m x^n = \frac{x^{m+1}-1}{x-1} = \frac{1-x^{m+1}}{1-x}$ (für alle $x \neq 1$)

$\sum_{j=0}^{n-1} (\omega_n^k)^j = \frac{(\omega_n^k)^n-1}{\omega_n^k-1} = \frac{1-1}{\omega_n^k-1} = 0$ ($\omega_n^k \neq 1$ da k nicht Vielfaches von n)

Definition: (Diskrete Fourier Transformation – DFT)
Die Diskrete Fourier Transformationist eine lineare Abbildung mit

$$\underbrace{\begin{pmatrix} z_1 \\ z_2 \\ z_3 \\ \vdots \\ z_n \end{pmatrix}}_{\in\mathbb{C}^n} \Longrightarrow \mathcal{A} \underbrace{\begin{pmatrix} z_1 \\ z_2 \\ z_3 \\ \vdots \\ z_n \end{pmatrix}}_{\in\mathbb{C}^n}$$

wobei $\mathcal{A}$ eine $(n \times n)$ Matrix über $\mathbb{C}$ ist mit:

$$\mathcal{A} = \begin{pmatrix} 1 & 1 & 1 & 1 & \dots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \dots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \dots & \omega_n^{2(n-1)} \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \dots & \omega_n^{(n-1)(n-1)} \end{pmatrix}$$

d.h. $\mathcal{A} = (a_{i,j})$ mit $0 \leq i \leq n-1, \quad 0 \leq j \leq n-1$ (i=Zeile, j=Spalte)
$a_{i,j} = \omega_n^{i \cdot j}$.
Berechnung von $\mathcal{A}$z in Zeit $O(n^2)$.

Definition: inverse DFT

$$\mathcal{A}^{-1} = \frac{1}{n} \begin{pmatrix} 1 & 1 & 1 & 1 & \dots & 1 \\ 1 & \frac{1}{\omega_n} & \frac{1}{\omega_n^2} & \frac{1}{\omega_n^3} & \dots & \frac{1}{\omega_n^{n-1}} \\ 1 & \frac{1}{\omega_n^2} & \frac{1}{\omega_n^4} & \frac{1}{\omega_n^6} & \dots & \frac{1}{\omega_n^{2(n-1)}} \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ 1 & \frac{1}{\omega_n^{n-1}} & \frac{1}{\omega_n^{2(n-1)}} & \frac{1}{\omega_n^{3(n-1)}} & \dots & \frac{1}{\omega_n^{(n-1)(n-1)}} \end{pmatrix}$$

Eintrag 1,1 von $\mathcal{A} \cdot \mathcal{A}^{-1} = 1$
Eintrag 2,2 von $\mathcal{A} \cdot \mathcal{A}^{-1} = 1$, d.h Diagonale 1
Eintrag 1,2 $= \frac{1}{n} \sum_{j=0}^{n-1} 1 \cdot \frac{1}{\omega_n^j} = \frac{1}{n} \sum_{j=0}^{n-1} (\omega_n^j)^{-1} = 0$.

Kapitel 7

Die schnelle Fourier Transformation (FFT)

Ziel: Berechnung der DFT und DFT^{-1} in Zeit $\mathrm{O}(n \log n)$
Annahme: n Zweierpotenz

Effiziente Berechnung der DFT von $a = \begin{pmatrix} a_0 \\ \vdots \\ a_{n-1} \end{pmatrix}$ durch Divide and Conquer:

Zerlegen von a in : $a^{(0)} = \begin{pmatrix} a_0 \\ a_2 \\ a_4 \\ \vdots \\ a_{n-2} \end{pmatrix} \qquad a^{(1)} = \begin{pmatrix} a_1 \\ a_3 \\ a_5 \\ \vdots \\ a_{n-1} \end{pmatrix} \qquad a = \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-2} \\ a_{n-1} \end{pmatrix}$

Wir wenden DFT mit $\omega_{\frac{n}{2}}$ auf $a^{(0)}$ an und dann auf $a^{(1)}$. Wir bekommen dann:

$y^{(0)} = \begin{pmatrix} y_0^{(0)} \\ y_1^{(0)} \\ \vdots \\ y_{\frac{n}{2}-1}^{(0)} \end{pmatrix}$ für $a^{(0)}$ $\qquad y^{(1)} = \begin{pmatrix} y_0^{(1)} \\ y_1^{(1)} \\ \vdots \\ y_{\frac{n}{2}-1}^{(1)} \end{pmatrix}$ für $a^{(1)}$, die in rekursiver Form weiter aufgeteilt werden.

Wir können $y = \begin{pmatrix} y_0 \\ \vdots \\ y_{n-1} \end{pmatrix}$, die DFT (mit ω_n) in Zeit $\mathrm{O}(n)$ (Linearzeit!) aus $y^{(0)}$ $y^{(1)}$ gewinnen.

Für $0 \leq k \leq \frac{n}{2} - 1$ gilt:

$y_k = \sum_{j=0}^{n-1} \omega^{kj} - a_j$

$= \sum_{j=0}^{\frac{n}{2}-1} \underbrace{\omega_n^{k \cdot 2j}}_{gerade\ Spalten} - \underbrace{a_{2j}}_{gerade\ Komponenten} + \sum_{j=0}^{\frac{n}{2}-1} \omega_n^{k \cdot (2j+1)} - a_{2j+1}$

$= \sum_{j=0}^{\frac{n}{2}-1} \omega_n^{k \cdot 2j} - a_{2j} + \omega_n^k \sum_{j=0}^{\frac{n}{2}-1} \omega_n^{k \cdot 2j} - a_{2j+1}$

$= \sum_{j=0}^{\frac{n}{2}-1} \omega_{\frac{n}{2}}^{k \cdot j} - a_{2j} + \omega_n^k \sum_{j=0}^{\frac{n}{2}-1} \omega_{\frac{n}{2}}^{k \cdot j} - a_{2j+1}$

$= y_k^{(0)} + \omega_n^k y_k^{(1)}$

(für $0 \leq k \leq (\frac{n}{2}-1)$ so haben wir eine Fouriertransformation mit $\frac{n}{2}$).

Für die anderen k, $\frac{n}{2} \leq k \leq (n-1)$, haben wir: Sei $k = \frac{n}{2}+k^{'}$, wobei $0 \leq k^{'} \leq \frac{n}{2}-1$
$y_k = \sum_{j=0}^{\frac{n}{2}-1} \omega_n^{k2j} \cdot a_{2j} + \omega_n^k \sum_{j=0}^{\frac{n}{2}-1} \omega_n^{k2j} \cdot a_{2j+1}$

$= \sum_{j=0}^{\frac{n}{2}-1} \omega_{\frac{n}{2}}^{kj} \cdot a_{2j} + \omega_n^k \sum_{j=0}^{\frac{n}{2}-1} \omega_{\frac{n}{2}}^{kj} \cdot a_{2j+1}$

$(\omega_{\frac{n}{2}}^{kj} = \omega_{\frac{n}{2}}^{(\frac{n}{2}+k^{'})j} = \omega_{\frac{n}{2}}^{k^{'}j})$

$= \sum_{j=0}^{\frac{n}{2}-1} \omega_{\frac{n}{2}}^{k^{'}j} \cdot a_{2j} + \omega_n^k \sum_{j=0}^{\frac{n}{2}-1} \omega_{\frac{n}{2}}^{k^{'}j} \cdot a_{2j+1}$

$(\omega_n^{\frac{n}{2}+k^{'}} = \omega_n^{\frac{n}{2}} \cdot \omega_n^{k^{'}} = -\omega_n^{k^{'}})$

$= y_{k^{'}}^{(0)} - \omega_n^{k^{'}} \cdot y_{k^{'}}^{(1)}$.
Laufzeit: O$(\log n)$

Anwendung der schnellen Fourier Transformation: Polynommultiplikation in Zeit O$(n \log n)$,
Polynom in x ist eine Funktion $t(x) : \mathbb{C} \to \mathbb{C}$ mit $t(x) = \sum_{j=0}^{n-1} a_j \cdot x^j$ und $a_{n-1} \neq 0$, so ist (n-1) der Grad von $t(x)$ ($a_j \in \mathbb{C}$ oder $a_j \in \mathbb{R}$).

Darstellung in Koeffizientendarstellung t durch $\begin{pmatrix} a_0 \\ \vdots \\ a_{n-1} \end{pmatrix}$.

Auswertung von t linear nach Hornerschema:
$t(x) = a_0 + x \cdot (a_1 + x \cdot (a_2 + x \cdot (a_3 + ...x \cdot (a_{n-2} + x \cdot (a_{n-1}))...)))$
Addition zweier Polynome linear
Multiplikation $A = \sum_{j=0}^{n-1} a_j \cdot x^i$, $B = \sum_{j=0} n - 1 b_i \cdot x^j$, dann $A \cdot B = \sum c_j \cdot x^j$,
$c_j = \sum_{k=0}^{j} a_k \cdot b_{j-k}$ ($c_{2n-2} = a_{n-1} \cdot b_{n-1}$, $\quad c_{2n-3} = a_{n-1} \cdot b_{n-2} + a_{n-2} \cdot b_{n-1}$)

$\begin{pmatrix} c_0 \\ \vdots \\ c_{n-1} \end{pmatrix}$ ist Faltung von $\begin{pmatrix} a_0 \\ \vdots \\ a_{n-1} \end{pmatrix} \begin{pmatrix} b_0 \\ \vdots \\ b_{n-1} \end{pmatrix}$
Zeit: O(n^2)

Darstellung: Punkt-Wert Darstellung
Punkt-Wert Darstellung ist Darstellung von $t(x)$ durch n Punkt-Wert-Paare
$\{(x_0, y_0), ..., (x_{n-1}, y_{n-1})\}$, wobei $y_k = t(x_k)$ und x_k verschieden.
Annahme: Hier sei $x_k = \omega_n^k$.
Satz: Eindeutigkeit der Punkt-Wert-Darstellung.

Beweis: hat $t(x)$ die Koeffizienten $\begin{pmatrix} a_0 \\ \vdots \\ a_{n-1} \end{pmatrix}$, dann ist

$$\begin{pmatrix} y_0 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \cdots & \omega_n^{2(n-1)} \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \cdots & \omega_n^{(n-1)(n-1)} \end{pmatrix} \cdot \begin{pmatrix} a_0 \\ \vdots \\ a_{n-1} \end{pmatrix}$$

wobei $\{(1, y_0), (\omega_n^1, y_1), \ \ldots \ , (\omega_n^{n-1}, y_{n-1})\}$ Punkt-Wert-Darstellung. Wegen invertierbarkeit der Fourier-Transformation sind $y_0, ..., y_{k-1}$ eindeutig.
Addition: O(n) , Multiplikation O(n) (für Werte)
Falls in Koeffizientendarstellung, Multiplikation in Zeit O$(n \log n)$

Kapitel 8

Grundlagen der Wahrscheinlichkeitslehre

8.1 Einführung

Beispiel: 10 Münzwürfe, Ergebnissen folgen aus der Menge
$\Omega = \{(a_1,\ a_2,\ ...\ ,a_{10})\ ,\ \ a_i \in \{K, Z\}\ \}$ Ω = Menge der Elementarereignisse.
Wahrscheinlichkeit von $(K, K, K, K,\ ...\ K, K) = (\frac{1}{2})^{10} = \frac{1}{1024}$.
Wahrscheinlichkeit von $(K, Z, Z, K,\ ...\ Z, Z) = (\frac{1}{2})^{10} = \frac{1}{1024}$.
Alle Elementarereignisse gleichwahrscheinlich:

Laplace Verteilung: (uniform) Für $\omega \in \Omega$ ist
$\text{Prob}(\omega) = \frac{1}{|\Omega|}$ (Prob – Probability = Wahrscheinlichkeit)
Dann Fortsetzung auf $\mathcal{P}(\omega)$ mit
Prob $\mathcal{P}(\Omega) \to R^{\geq 0}$
Prob $(A) = |A| \cdot \frac{1}{|\Omega|}$ $|A \cup B| = |A| + |B|$
Dann gilt:
$\text{Prob}(\varnothing) = 0$
$\text{Prob}(\Omega) = 1$
$\text{Prob}(A \cup B)) = \text{Prob}(A) + \text{Prob}(B)$, falls $A \cap B = \varnothing$

Definition: Ein W-Baum $\mathcal{P}$ besteht aus 3 Komponenten:
Ω = Menge der Elementarereignisse (bei uns endlich)
$\mathbf{a} = \mathcal{P}(\Omega)$ = Menge der beobachtbaren Ereignisse
Prob $\mathcal{A} \to \mathbf{R}^{\geq 0}$ mit
(a) $0 \leq Prob(A) \leq 1$
(b) $\sum_{\omega \in \Omega} Prob(\omega) = 1$
(c) $Prob(A) = \sum_{\omega \in A} Prob(\omega)$.

Beispiel: Eingaben von Sortieralgorithmus als W-Raum
$\Omega = \Omega_n = \{(a_1, ..., a_n) \mid (a_1, ..., a_n)$ Permutation von $\{1, .., n\}\}$

Worst case Zeit:
$\mathrm{T}_{wc} : \mathbf{N} \to \mathbf{N}$
$\mathrm{T}_{wc}(n) :\ Max\{$ Zeit auf $\omega \mid \omega < \Omega_n\}$

Mittlere Zeit:
$\mathrm{T}_{av} : \mathbf{N} \to \mathbf{N}$
$\mathrm{T}_{av}(n) :$ mittlere Zeit über Ω_n

Was ist $\text{Prob}\big((a_1, ..., a_n)\big)$?
$\text{Prob}\big((a_1, ..., a_n)\big) = \frac{1}{n!}$
$\text{T}_{av}(n) = \sum_{\omega \in \Omega_n} (\text{ Laufzeit auf } \omega) \cdot \frac{1}{n!}$

Dann ist $\text{T}_{av}(n) \leq \text{T}_{wc}(n)$, denn
$\text{T}_{av}(n) = \sum_{\omega \in \Omega_n} (\text{ Laufzeit auf } \omega) \cdot \frac{1}{n!}$
$\leq \sum_{\omega \in \Omega} \text{T}_{wc}(n) \cdot \frac{1}{n!}$
$= \text{T}_{wc}(n)$.

Folgerung: $\mathcal{P} = (\Omega,\ \mathcal{A},\ \text{Prob})$ sei ein W-Raum. Dann gelten die üblichen Gesetze:
(a) $\text{Prob}(\varnothing) = 0$
(b) $\text{Prob}(\Omega) = 0$
(c) $\text{Prob}(\Omega \setminus A) = 1 - \text{Prob}(A)$
(d) $B \subseteq A$, dann $\text{Prob}(A \setminus B) = \text{Prob}(A) - \text{Prob}(B)$
(e) $\text{Prob}(A \cup B) = \text{Prob}(A) + \text{Prob}(B) - \text{Prob}(A \cap B)$.

Beweis:
(a) $\text{Prob}(\varnothing) = \sum_{\omega \in \varnothing} \text{Prob}(\omega) = 0$ (Def. der Wahrscheinlichkeit).

(b) $\text{Prob}(\Omega) = \sum_{\omega \in \Omega} \text{Prob}(\omega) = 1$.

(c) Es ist $A \cup (\Omega \setminus A) = \Omega$
$1 = \text{Prob}(\Omega) = \sum_{\omega \in A} \text{Prob}(\omega) + \sum_{\omega \in \Omega \setminus A} \text{Prob}(\omega)$
$= \text{Prob}(A) + \text{Prob}(\Omega \setminus A)$
$\Rightarrow \text{Prob}(\Omega \setminus A) = 1 - \text{Prob}(A)$.

(d) $B \subseteq A$, dann $A = B \cup \ \ A \setminus B)$
Also $\text{Prob}(A) = \text{Prob}(B) + \text{Prob}(A \setminus B)$.
(e) Es ist $A \cup B = A \setminus (A \cap B) \cup B \setminus (A \cap B) \cup (A \cap B)$
Dann gilt: $\text{Prob}(A \cup B) = \text{Prob}(A \setminus (A \cap B)) + \text{Prob}(B \setminus (A \cap B)) + \text{Prob}(A \cap B)$
$= \text{Prob}(A) - \text{Prob}(A \cap B) + \text{Prob}(B) - \text{Prob}(A \cap B) + \text{Prob}(A \cap B)$.

Beispiel: Ω_n = Permutation auf $(1, ..., n)$, Laplace Verteilung.
$\text{Prob}\{\omega \in \Omega_n, a_1 = 1,\ \omega_n\}$ Laplace-Fall
$= \frac{1}{n!} |\{\omega \in \Omega_n(a_1)\} = \frac{1}{n!} \cdot (n-1)! = \frac{1}{n}$ (Ebenso $\text{Prob}\{\omega \in \Omega \mid a_1 = i\}$)

Haben Zerlegung von Ω_n in $\Omega_n = \Omega^{(1)} \cup \Omega^{(2)} \cup \ldots \cup \Omega^{(n)}$, (alle $\Omega^{(i)}$ sind disjunkt) wobei $\Omega^{(i)} \mathrel{\widehat{=}} a_1 = i$.

Welche W-Räume induzieren Ω_n auf $\Omega^{(i)}$?
$\Omega_n \mathrel{\widehat{=}}$ ziehe n Zahlen aus $\{1, ..., n\}$ hintereinander, ohne Zurücklegen
$\Omega^{(i)} \mathrel{\widehat{=}}$ ziehe n Zahlen aus $\{1, ..., n\}$ hintereinander, ohne Zurücklegen und berücksichtige nur die Fälle mit $a_1 = i$
$\text{Prob}_{\Omega^{(i)}}(i, a_2, ..., a_n) = \frac{1}{(n-1)!} = \frac{1}{|\Omega^{(i)}|}$
$\text{Prob}_{\Omega^{(i)}}(i, a_2, ..., a_n) = \frac{n}{n!} = \frac{\frac{1}{n!}}{\frac{1}{n}} = \dfrac{\text{Prob}_{\Omega_n}(i, a_2,\ \ldots\ , a_n)}{\text{Prob}_{\Omega_n}(\Omega^{(i)})}$
Relative W-keit auf $\Omega^{(i)}$

Definition:
$\mathcal{P} = (\Omega,\ \underbrace{\mathcal{P}(\Omega)}_{\mathcal{A}},\ \text{Prob}),\quad A \leq \Omega,\quad \text{Prob}(A) >\neq 0$, dann $\text{Prob}_A :\ \mathcal{A} \to \mathbb{R}^2$

mit $\text{Prob}_A(B) = \dfrac{\text{Prob}_\Omega B \cap A}{\text{Prob}_\Omega A}$

Folgerung:
Ist $\mathcal{P} = (\Omega,\ \mathcal{P}(\Omega),\ \text{Prob})$ ein W-Raum, dann auch $\mathcal{P}_A = (A,\ \mathcal{P}(A),\ \text{Prob}_A)$.
Beweis: bei $B \subseteq A$ ist $\text{Prob}_A(B) = \dfrac{\text{Prob}_\Omega(B)}{\text{Prob}_\Omega(A)}$
$0 \leq \text{Prob}_A(B) \leq 1$, da $B \subseteq A$ $\big(\ \text{Prob}\Omega(A) = \sum_{\omega\in A} \text{Prob}\Omega(\omega)\ \big)$
$\text{Prob}_A(A) = 1$
$\text{Prob}_A(B) = \sum_{\omega\in B} \text{Prob}_A(\omega)$ ergibt sich auch direkt
Bsp: Ω mit Laplace Verteilung, $\text{Prob}(A) = \frac{|A|}{|\Omega|}$, dann $\text{Prob}_A(B) = \frac{|B\cap A|}{|A|}$
Für $B \subseteq A$ $\text{Prob}_A(B) = \frac{|B|}{|A|}$ (Bedingte Räume wieder Laplace).

Beispiel: $\Omega = \big\{(a_1,\ \ldots, a_n) \mid a_i \in \{1...n\}$ alle verschiden$\big\}$
Sortieralgorithmus, Eingabe zufällig gemäß W-keit aus Ω.
Laufzeit ist Funktion $T : \Omega \to \mathbf{N}$
Man interessiert sich für Prob $\underbrace{\{\omega | T(\omega) = 1000}_{=T^{-1}(1000)\ =\ [T=1000]}\ \}$ $\text{Prob}[T = 1000]$
$\text{Prob}[T \geq 100]$ analog T ist Zufallsvariable

Definition:
Eine Zufallsvariable X von einem endlichen W-Raum
$\mathcal{P} = (\Omega,\ \mathcal{P}(\Omega), Prob)$ ist eine Funktion $X : \Omega \to \mathbb{R}$

Beispiel: Sei $A \subset \Omega$, dann $I_A : \Omega \Rightarrow \mathbb{N}$
$I_A(\omega) = \begin{Bmatrix} 1 & \text{falls } \omega \in A \\ 0 & \text{sonst} \end{Bmatrix}$
$\text{Prob}[I_a = 1] \ = \ \text{Prob}\{\omega \mid I_A(\omega) = 1\} \ = \ \text{Prob}(A).$

Definition:
X eine Zufallsvariable, Ω endlich
$X(\Omega) = \{x_1,\ \ldots, x_n\}$, alle x_i verschieden
definieren:$\mathcal{P}_X = \big(X(\Omega),\ \mathcal{P}\big(X(\Omega)\big),\ \text{Prob}_x\big)$ mit
$\text{Prob}_X(x_i) \ = \ \text{Prob}[X = x_i] \ = \ \text{Prob}\big\{\omega \in \Omega \mid X(\omega) = x_i\big\}$
Man bekommt dann wirklich W-Raum auf $\{(x_1,\ \ldots, x_n)\}$. Die Wahrscheinlichkeiten $(\text{Prob}_X(x_1),\ \text{Prob}_X(x_2),\ \ldots, \text{Prob})_X(x_n)$ sind die Verteilung von X.

Beispiel:
(a) $\mathcal{P}$ sei W-Raum für Münzwurf. $\text{Prob(K)} = p,\ \text{Prob(Z)} = q = 1 - p$
$\mathcal{P} = \big(\{\text{K,Z}\},\ \mathcal{P}\big(\{\text{K,Z}\}\big), \text{Prob}\big)$ (Bernoulli Experiment)
Zufallsvariable $X : \{\text{K,Z}\} \Rightarrow \mathbb{R}$, $X(\text{K}) = 1, X(\text{Z}) = 0$
$\text{Prob}[X = x] = p^x \cdot q^{1-x}$ $(x \in \{0, 1\},\quad x = 0,$ dann q , $\quad x = 1,$ dann p)
(Bernoulli ZV)
(b) n Münzwürfe hintereinander, so $\Omega = \big\{(a_1,\ \ldots,\ a_n) | a_i \in \{\text{K,Z}\}\big\}$
$\text{Prob}(a_1,\ \ldots,\ a_n) = p^{\text{Anzahl Kopf in } (a_1,\ \ldots a_n)} q^{\text{Anzahl Zahl in } (a_1,\ \ldots a_n)}$
$q = 1 - p \quad p = q = \frac{1}{2}$, dann $\left(\frac{1}{2}\right)^n$ Laplace Verteilung
Die Definition von Prob gibt W-Raum:
$\sum_{\omega\in\Omega} \text{Prob}(\omega) = \sum_{k=0}^{n} \binom{n}{k} \cdot p^k \cdot q^{n-k} = (p + q)^n = 1$ (Binomial Satz)
Eine Zufallsvariable $X : \Omega \to \mathbb{R}$, $X(\omega) =$ Anzahl Kopf in ω
$\text{Prob}[X = 0] = p^0 \cdot q^n$
$\text{Prob}[X = 1] = n \cdot p^1 \cdot q^{n-1}$ $\text{Prob}(A) = \sum_{\omega\in A} \text{Prob}(\omega)$
$\text{Prob}[X = 2] = p^2 \cdot q^{n-2}$
$p_i = \text{Prob}[X = i]$ dann $(p_1,\ \ldots, p_n)$ Binomial-Verteilung.

Definition(Erwartungswert)
$X : \Omega \to \mathbb{N}$ Zufallsvariable, so EX $= \sum_{\omega\in\Omega} \mathrm{Prob}(\omega) \cdot X(\omega)$
Bsp:
(a) $\Omega = \{(a_1, \ ... \ a_n) \mid a_i \in \{1...n\}\}$ alle verschieden, Laplace Verteilung
$T : \Omega \to \mathbb{N}$ Laufzeit eines Sortieralgorithmus, dann
ET $= \sum_{\omega\in\Omega} T(\omega) \cdot \mathrm{Prob}(\omega) = \frac{1}{n!} \cdot \sum T(\omega)$

(b) X Zufallsvariable mit Binomialverteilung mit n,p , dann:
EX $= \sum_{i=0}^{n} \mathrm{Prob}[X = i] \cdot i = \sum_{\omega\in\Omega} X(\omega) \cdot \mathrm{Prob}(\omega)$
$= \sum_{i=0}^{n} i \cdot \binom{n}{i} \cdot p^i \cdot q^{n-1}$

$= \sum_{i=0}^{n} i \cdot \frac{n!}{i! \cdot (n-i)!} \cdot p^i \cdot q^{n-1}$

$= \sum_{i=\mathbf{1}}^{n} \frac{n!}{(i-1)! \cdot (n-i)!} \cdot p^i \cdot q^{n-1}$

$= \sum_{i=1}^{n} \frac{n!}{(i-1)! \cdot \big((n-1)-(i-1)\big)!} \cdot p^i \cdot q^{n-1}$

$= n \cdot p \cdot \sum_{i=1}^{n} \frac{n!}{(i-1)! \cdot \big((n-1)-(i-1)\big)!} \cdot p^{i-1} \cdot q^{(n-1)-(i-1)}$

$= n \cdot p \cdot \sum_{i=0}^{n} \frac{(n-1)!}{i! \cdot \big((n-1)-i\big)!} \cdot p^i \cdot q^{(n-1)-i}$

$= n \cdot p \cdot \sum_{i=0}^{n} \binom{n-1}{i} \cdot p^i \cdot q^{(n-1)-i} = n \cdot p \cdot \underbrace{(p+q)}_{=1}{}^{n-1}$

($n = 3$, $p = \frac{1}{2}$ dann EX=1,5)

Kapitel 9

Untere Schranke für die mittlere Laufzeit beim Sortieren

Sortieren (nur mit Vergleichen) worst case untere Schranke: $\Omega(n \log n)$
Zeigen jetzt Erwartungswert ist $\Omega(n \log n)$

Beispiel: Entscheidungsbaum, Algorithmus nur mit Vergleichen
Eingabe ist: $(a_1,\ a_2,\ a_3)$
mögliche Ausgabe: $(a_1,\ a_2,\ a_3),\ (a_1,\ a_3,\ a_2),\ \ldots\ ,\ (a_3,\ a_2,\ a_1)$

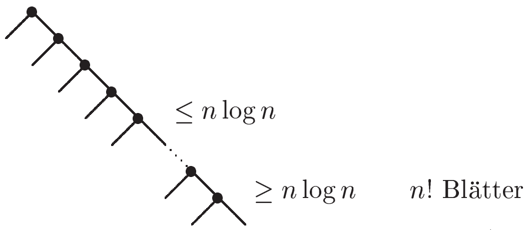
$$a_1 < a_2$$

ja $\swarrow$	$\searrow$ *nein*
(a_1, a_2, a_3)	(a_2, a_1, a_3)
(a_3, a_1, a_2)	(a_3, a_2, a_1)
(a_1, a_3, a_2)	(a_2, a_3, a_1)

$\geq n!$ Blätter , Tiefe $\geq \frac{1}{4} n \log n$.

Satz:
Wir betrachten jetzt $\mathcal{P}_n = (\Omega_n,\ \mathcal{P}(\Omega_n),\ \text{Prob}_n)$
$\mathcal{P}_n$ = Permutationen auf$\{1...n\}$
$\text{Prob}(\omega) = \frac{1}{n!}$
Sei T ein fester Entscheidungsbaum zum Sortieren, dann Zufallsvariable $X : \Omega \to \mathbb{N}$ mit $X(\omega)$ = Tiefe der Blätter zu denen T mit ω geht. Es gilt: $\text{EX} = \Omega(n \log n)$!

(Die Behauptung bedeutet etwa: viele Blätter liegen in Tiefe $\geq c \cdot n \cdot \log n$. Es könnte so sein :



Hier ist $\text{EX} = \frac{1}{n!}(1 + 2 + 3 + \ldots + n) = \frac{1}{n!} \cdot \frac{n! \cdot (n!+1)}{2} = \frac{n!+1}{2} \geq n \log n$).

Beweis: Zunächst Annahme: Jedes Blatt des Baumes wird genau einmal erreicht. Also Anzahl Blätter = $n!$

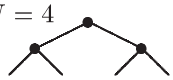
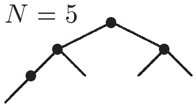
$\mathrm{EX} = \sum_{\omega \in \Omega} \mathrm{Prob}(\omega) X(\omega)$

$= \frac{1}{n!} \sum_{\omega \in \Omega} X(\omega)$

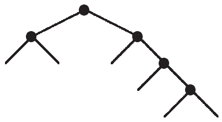
$= \frac{1}{n!} \sum_{b} \mathrm{Tiefe}(b) \qquad (\text{ b = Blatt von T})$

$\sum_{b} \mathrm{Tiefe}(b) =$ externe Weglänge von T

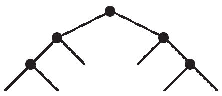
Suchen untere Schranke an externer Weglänge bei $n!$ Blätter.
Bezeichnung: Ein binärer Baum mit N Blätter ist maximal ausgeglichen $\Leftrightarrow$ Alle Blätter in Tiefe $\lfloor \log N \rfloor$ oder $\lfloor \log N \rfloor + 1$.
Beispiel:

$N = 4$  $N = 5$ 

Zeigen: Minimale externe Weglänge wird bei maximaler Ausgeglichenheit erreicht.
Sei N Bläterzahl, sei $\lfloor \log_2 N \rfloor = k$
Sei T irgend ein Baum mit genau N Blättern, dann Tiefe(T) $\geq k$
Elimination der Blätter in Tiefe $\geq k+2$

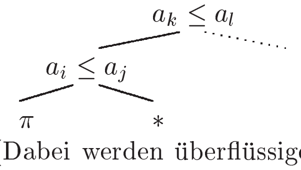


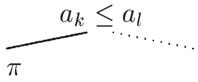
Da $\lfloor \log N \rfloor = k$ gibt es ein Blatt in Tiefe k ohne Kinder, dann einfach umhängen:



Wenden das auf den Entscheidungsbaum T an, so:
Maximale Weglänge $\geq \underbrace{n!}_{\substack{\text{Anzahl}\\ \text{Blätter}}} \cdot \underbrace{(\log n!)}_{\text{Tiefe}}$

Also ist $\mathrm{EX} \geq \log(n!) \geq \frac{1}{n} \cdot \log n$
$\geq \log n \cdot (n-1) \cdot \ \ldots \ \cdot \left(\frac{n}{2}\right)$
$\geq \log \left(\frac{n}{2}\right)^{\frac{n}{2}} = \frac{n}{2} \log \frac{n}{2}$
Es bleibt noch die Annahme: Keine nicht erreichbaren hinzustellen.
1. Markiere nicht erreichbare Blätter (*)
2. Transformation





(Dabei werden überflüssige Vergleiche beseitigt, d.h. Blätter rausschneiden, wenn sie keine neuen Informationen liefern).

Kapitel 10

Erwartungswert von Quicksort

Funktion von Quicksort:

$$\underbrace{3}_{\text{A[1]}} \quad 1 \quad 4 \quad 5 \quad 9 \quad 2 \quad 6 \quad 5 \quad \underbrace{3}_{\text{A[9]}}$$

A[1] ist Pivotelement (Splitter)

$$3 \quad 1 \quad \underline{4} \quad 5 \quad 9 \quad 2 \quad 6 \quad 5 \quad \underline{3}$$

$$3 \quad 1 \quad 3 \quad \underline{5} \quad 9 \quad \underline{2} \quad 6 \quad 5 \quad 4$$

$$\underline{3} \quad 1 \quad 3 \quad \underline{2} \quad 9 \quad 5 \quad 6 \quad 5 \quad 4$$

$$\underbrace{2 \quad 1 \quad 3 \quad 3}_{\text{linkes Teilarray}} \quad \underbrace{9 \quad 5 \quad 6 \quad 5 \quad 4}_{\text{rechtes Teilarray}}$$

Das linke und rechte Teilarray werden rekursiv weiter sortiert.

worst-case von Quicksort: (Eingabe schon sortiert)
$\underline{1}, 2, 3, 4, 5, \ \dots \ , n$
$\underline{1}, \underline{2}, 3, 4, 5, \ \dots \ , n$
$\vdots$

Die Anzahl der Vergleiche ist: $n + (n-1) + (n-2) + \ \dots \ + 3 + 2 + 1 \ = \ \frac{n \ \cdot (n+1)}{2}$.
Also ist die untere Schranke $\Omega(n^2)$.

Satz über die totale Wahrscheinlichkeit
Seien die Wahrscheinlichkeitsräume $W_i = (\Omega_i, \ \mathcal{A}_i, \ \mathrm{P}_{\Omega_i})$ $(1 \leq i \leq n)$ (P = Prob) und $W = (\Omega, \ \mathcal{A}_\Omega, \ \mathrm{P}_\Omega)$ mit $\Omega = \Omega_1 \cup \Omega_2 \cup \dots \cup \Omega_n$ wobei alle Ω_i paarweise disjunkt sind $(\Omega_i \cap \Omega_k = \varnothing$ für $i \neq k)$
Betrachten Ereignis $A \in \Omega$
$A = A \cap \Omega$
$A = A \cap (\Omega_1 \cup \Omega_2 \cup \dots \cup \Omega_n)$
$A = (A \cap \Omega_1) \cup (A \cap \Omega_2) \cup \ \dots \ \cup (A \cap \Omega_n)$
$\mathrm{P}(A) = p(A \cap \Omega_1) + p(A \cap \Omega_2) + \ \dots \ + p(A \cap \Omega_n)$
$\mathrm{P}(A) = \sum_{i=1}^{n} [\mathrm{P}_\Omega(\Omega_i) \cdot \mathrm{P}_\Omega(A/\Omega_i)]$
$\mathrm{P}(A) = \sum_{i=1}^{n} p(\Omega_i) \cdot p_{\Omega_i}(A)$

Berechnung von bedingten Erwartungswerten:
$W_1 = (\Omega_i,\ \mathcal{A}_i,\ p_{\Omega_i})$
$W = (\Omega,\ \mathcal{A},\ p)$
Zufallsvariable: $X : \Omega \to \mathbb{N}$
$X_i : \Omega_i \to \mathbb{N}$
$X(\omega) = X_i(\omega)$ für alle $\omega \in \Omega_i \subseteq \Omega$

$$\begin{array}{rcl}
\mathrm{EX} & = & \sum_{\omega\in\Omega} p(\omega)\cdot X(\omega) \\
 & = & \sum_{i=1}^{n}\sum_{\omega\in\Omega_i} p(\omega\cap\Omega_i)\cdot X(\omega) \\
 & = & \sum_{i=1}^{n}\sum_{\omega\in\Omega_i} \mathrm{P}(\omega/\Omega_i)\cdot \mathrm{P}(\Omega_i)\cdot X(\omega) \\
 & = & \sum_{i=1}^{n} \mathrm{P}(\Omega_i)\cdot\sum_{\omega\in\Omega_i} \mathrm{P}_{\Omega_i}(\omega)\cdot X_i(\omega) \\
\mathrm{EX} & = & \sum_{i=1}^{n} \mathrm{P}(\Omega_i)\cdot \mathrm{EX}_i \\
\mathrm{EX}_i & = & \sum_{i=1}^{n} E(X|\Omega_i)
\end{array}$$

10.1 Algorithmus Quicksort

− divide and conquer -Strategie
− Eingabe: Feld A[1 ... n], $(1 \le l, r \le n)$
− Ausgabe: Sortierung vom A[l..r]

Quicksort(A,l,r)
1.) q:=Parition(A,l,r)
- x=A[1] „Pivot Element”
  x wird richtig einsortiert, d.h. steht an Position q
- Umsortierung von A
  A[i] $\le x$ für $i \le q$
  A[k] $\le x$ für $k \ge q$

2.) if $l < q-1$ then Quicksort(A, l, q-1)
3.) if $q+1 > r$ then Quicksort(A, q+1, r)

(Bemerkung: If-Anweisung in Zeile 2 und 3 stellt sicher, daß Felder der Größe $\ge$2 sortiert werden)

Partition(A,l,r)
1.) x:=A[l]
2.) j:=r+1
3.) i:=l
4.) while true do
5.) repeat j:=j-1 until (A[j] $\le x$ or $j = l$)
6.) repeat i:=i+1 until (A[i] $\le x$ or $i = r$)
7.) if $i < l$ then vertausche A[i] und A[j]
8.) else vertausche A[j] und A[l]
9.) return

10.2 Beispiel

$$\begin{array}{lccccccccl}
A = (& 3 & 7 & 6 & 2 & 1 & 5 & 4 &) \\
 & i \rightarrow & \uparrow & & & \uparrow & & \leftarrow j & \\
 & & i & & & j & & & \\
 & & & & & & & & \\
A = (& 3 & 1 & 6 & 2 & 7 & 5 & 4 &) \\
 & & & \uparrow & \uparrow & & & & \\
 & & & & & & & & \\
A = (& 3 & 1 & 2 & 6 & 7 & 5 & 4 &) \\
 & & & & & & & & \\
A = (& 2 & 1 & \underbrace{3}_{x} & 6 & 7 & 5 & 4 &)
\end{array}$$

10.3 Bemerkung zur Laufzeit

(a) Laufzeit von Partition = Anzahl der Vergleiche (A$[k] \leq x$)
Laufzeit von Quicksort= Anzahl der Vergleiche in den Partitionen
(b) worst-case Laufzeit von Quicksort = $\Theta(n^2)$
best-case Laufzeit von Quicksort= $\Theta(n^2)$
(c) Anzahl Vergleiche in Partition: Vergleiche in Zeile 5, 6
Lauftzeit von Partition(A,l,r) $\in$ { r-l+1,r-l+2 } (eines der beiden Ereignisse)

10.4 Satz: Erwartungswert von Quicksort

− benötigen Wkt.-Raum
− Sortierung eines Feldes von n Elementen
− Gleichverteilung (Laplace) aller Felder mit n Elementen
$p(A[1,2]) = p(A[2,1]) = \frac{1}{2}$

A=$(a_1,\ a_2,\ ...,\ a_n)$
$\Omega_n = \big\{(a_1...a_n) \mid$ Permutationen von $\{1,2,...,n\}\big\}$ $\qquad (a_i \neq a_k$ für $i \neq k)$
$|\Omega_n| = n!$
$\mathcal{A} = p(\Omega)$
$p_n(\omega) = \frac{1}{n!}$ und $\omega \in \Omega$
$W_n = (\Omega_n,\ \mathcal{A}_n), \mathrm{P}_n)$

− Zufallsvariable X_n zum Messen der Laufzeit
$X_n : \Omega \rightarrow \mathbb{N}$
$X_n(A)$ = Anzahl der Vergleiche in Quicksort zum Sortieren des Feldes A=$(a_1,\ \dots, a_n)$
Satz: Quicksort hat im Mittel die Laufzeit $\Theta(n \log n)$.
Beweis: 1) $\mathrm{EX}_n = \Omega(n \log n)$
2) $\mathrm{EX}_n = O(n \log n)$
$\mathrm{EX}_n = \sum_{A \in \Omega_n} \frac{1}{n!} \cdot X_n(A)$
Bsp: Elemente (1,2,3,4)
A=$[1, ...]$
A=$[2, ...]$
A=$[3, ...]$
A=$[4, ...]$

Definieren E_k für beliebiges, festes n. E_k: Feld A beginnt mit A[1]=k
(Bsp: $E_3 \leftrightarrow$ A[3,...]) $E_i \cap E_i = 0$ für $i \neq k$

$$\begin{aligned} \mathrm{P}(E_k) &= \frac{\binom{n}{1}}{n!} \\ &= \frac{1}{(n-1)!} \end{aligned}$$

$\mathrm{EX}_n = \mathrm{E}(X/E_1) + \mathrm{E}(X/E_2) + ... + \mathrm{E}(X/E_n)$

1.Fall : $n = 2$
$\mathrm{EX}_2 = \mathrm{E}(X/\underbrace{E_1}_{a}) + \mathrm{E}(X/\underbrace{E_2}_{b})$
(für (a) A[1,2] und (b) A[2,1] jeweils 3 Vergleiche)
$\mathrm{EX}_2 = \frac{1}{2} \cdot 3 \ + \ \frac{1}{2} \cdot 3$
$\mathrm{EX}_2 = 3$

Satz: E-Wert von Quicksort $\mathrm{O}(n \log n)$
Beweis: $\Omega_n = \big\{(a_1 ... a_n) \mid (a_1, ..., a_n) \text{ Permutationen auf } \{1, ..., n\}\big\}$
$\mathrm{Prob}(a_1, ..., a_n) = \frac{1}{n!}$
$X_n : \Omega : n \to \mathbb{N}$
$X_1(a_1, ..., a_n) =$ Anzahl der Vergleiche der a_i
Suchen $\mathrm{EX}_n = \sum_{A \in \Omega_n} \mathrm{Prob}(A) \cdot X_n(A) = \sum \frac{1}{n!} \cdot X_n(A)$

Zwei wichtige Formeln:
- Formel von der totalen Wkt:
  Ist $\Omega = \Omega_1 \cup \Omega_2 \cup ... \cup \Omega_n$, dann ist $A \subseteq \Omega$.
  $\mathrm{Prob}_\Omega(A) = \mathrm{Prob}_\Omega(\Omega_A) \cdot \mathrm{Prob}(A/\Omega_1) + \ ... \ + \mathrm{Prob}_\Omega(\Omega_A) \cdot \mathrm{Prob}(A/\Omega_n)$
  wobei $\mathrm{Prob}\Omega_i) > 0$
- Formel der Berechnung von Erwartungswerten durch Bedingung:
  $\mathrm{EX} = \mathrm{Prob}_\Omega(\Omega_1) \cdot \mathrm{E}[X|\Omega_1] + \ ... \ + \mathrm{Prob}_\Omega(\Omega_n) \cdot \mathrm{E}[X|\Omega_n]$

Zu Quicksort: Ω_n und Wahrscheinlichkeitsräume
1. Ziehe zufällig 1 Element aus {1, ... ,n}
2. Ziehe zufällig 1 Element aus dem Rest
3. $\vdots$

Sei $n \geq 2$ fest. Für k mit $1 \leq k \leq n$
$E_k = \{A \in \Omega_n\} \quad A[1] = k$
dann $\Omega_n = E_1 \cup E_2 \cup \ ... \ \cup E_n$
dann auf jeden Fall (Bedingung)
$\mathrm{EX}_n = \sum_{k=1}^{n} \underbrace{\mathrm{Prob}(E_k)}_{=\frac{1}{n}} \cdot \mathrm{E}[X_n|E_k] = \frac{1}{n} \cdot \sum_{k=1}^{n} \mathrm{E}[X_n|E_k]$

Was ist $\mathrm{E}[X_n|E_k]$?

$n = 2: \quad \mathrm{E}[X_2|\underbrace{E_1}_{(1,2)}] = 3$

$\quad \mathrm{E}[X_2|\underbrace{E_2}_{(2,1)}] = 3$

$n = 3: \quad \mathrm{E}[X_n|E_2] = 4 \qquad E_2 = (2,3,1)\text{und}(2,1,3)$

$n \geq 3: \quad \mathrm{E}[X_n|E_1] = \sum_{A\in E_1} \mathrm{Prob}(A|E_1) \cdot X_n(A)$
$= \sum_{A\in E_1} \frac{1}{(n-1)!} \cdot X_n(A)$
$= \sum \frac{1}{(n-1)!} \cdot \big(n+1 \ + \ X_{n-1}(A[2], ..., A[n])\big)$
$A = \big[A[1], ..., A[n]\big]$
$= \Big(\sum_{A\in E_1} \frac{1}{(n-1)!}\big(X_{n-1}(A[2]...A[n])\big)\Big) + n + 1$
$= \sum_{B\in\Omega_{n-1}} \frac{1}{(n-1)!} \ X_{n-1}(B) \ + n + 1$
$= \mathrm{EX}_{n-1} \ + n + 1$
Man sieht (bei E_1), daß die uniforme Verteilung auf Ω_n die uniforme Verteilung auf Ω_{n-1} ergibt. Analog ist $\mathrm{E}[X_n|E_n] = \mathrm{EX}_{n-1} \ + n + 1$

Ist $n \geq 4$ dann:
$\mathrm{E}[X_n|E_2] = \sum_{A\in E_2} \frac{1}{(n-1)!}(X_n|A)$
$= \sum_{A\in E_2} \frac{1}{(n-1)!}\big(n+1 \ + X_{n-2}(\text{rechtes Teilarray})\big)$
nach Partition(1,<u>2</u>,rechtes Teilarray)
$= \sum_{A\in E_2} \frac{1}{(n-1)!}\big(X_{n-2}(\text{rechtes Teilarray})\big) \ + n + 1$
Wie oft tritt rechtes Teilarray (3, ... , n) auf ?
2 1 3 ... n
2 3 1 ... n
2 4 3 1 ... n
2 5 3 4 1 ... n
(3, ... , n) wird genau (n-1)-mal getroffen
Damit ist $\sum_{A\in E_2} \frac{1}{(n-1)!} \ X_{n-2}(\text{rechtes Teilarray})$
$= \sum_{B\in\Omega_{n-2}} \frac{1}{(n-1)!}(n-1) \cdot X_{n-2}(B)$
$= \sum_{B\in\Omega_{n-2}} \frac{1}{(n-2)!} X_{n-2}(B) = \mathrm{EX}_{n-2}$
Man beachte daß für rechtes Teilarray $(a_1, \ ... \ , \ a_{n-2})$ gilt: Es wird getroffen von:
2 1 a_1 ...a_{n-2}
2 a_1 1 ...a_{n-2}
2 a_2 a_3 1 ...a_{n-2}
Ebenso $\mathrm{E}[X_n|E_{n-1}] = \mathrm{EX}_{n-2} \ + n + 1$

Jetzt der Fall zwweier rekursiver Aufrufe, d.h. $n \geq 5$ mit $3 \leq k \leq n-2$. Dann gilt
$\mathrm{E}(X_n|E_k) = \mathrm{EX}_{k-1} + \mathrm{EX}_{n-k} \ + n + 1$
es ist $\mathrm{E}[X_n|E_k]$
$= \sum_{A\in E_k} \frac{1}{(n-1)!}\big(X_{k-1}(\text{linkes Teilarray von A}) + X_{n-k}(\text{rechtes Teilarray von A}) \ + n + 1\big)$
$= \sum_{A\in E_k} \frac{1}{(n-1)!}\big(X_{k-1}(\text{linkes Teilarray von A}) + X_{n-k}(\text{rechtes Teilarray von A})\big) + n + 1$
Situation ist: $A = (k, A[2], \ ... \ , A[n])$
$A' = (\underbrace{A'[1], A'[2], \ ... \ , A'[k-1]}_{\in\Omega_{k-1}}, b, \underbrace{A'[k+1], \ ... \ , A'[n]}_{\in\Omega_{n-k}})$

Wie oft wird $(a_1, \ ... \ , a_k)$ als linkes Teilarray getroffen ?
von $(k, \underbrace{a_2, ..., a_{k-1}, a_1}_{<k}, \ \underbrace{A[k+2], ..., A[n]}_{\text{alle Möglichkeiten hier}} \ )$

von $(k, a_2, ..., a_{k-1}, A[k+1], a_1, A[k+3], ..., A[n])$
für$A[k+1], A[k+3]...A[n]$ alle Möglichkeiten
Ebenfalls von $(k, A[2], A[3], ..., A[n-k], a_1, a_k, a_{k-1}, ..., a_3, a_2)$ bei$k < n-k$
für$A[2]...A[n-k]$ alle $(n-k)!$ Möglichkeiten
Sei jetzt $(a_1, ..., a_{k-1})$ fest. Permutation von $\{1, ..., k\}$. Wie alle A $\in E_k$ die $(a_1, \ ... \ , \ a_{k-1})$ als linkes Teilarray haben ?
1. Wählen die Positionen in A, wo $\{a_1, \ ... \ , a_{k-1}\}$ stehen $\binom{n-1}{k-1}$.
2. Ordne 1, ... , k-1 so auf Positionen an, daß hinterher $(a_1, \ ... \ , \ a_{k-1})$ rauskommt.
3. Bestimme Anordnung des Restes: $(n-k)!$ Möglichkeiten.
Insgesamt $\binom{n-1}{k-1} \cdot (n-k)!$ Möglichkeiten
$\sum_{A \in E_k} X_{k-1}($ linkes Teilarray von A $) = \sum_{A \in \Omega_{k-1}} X_{n-k}(B) \cdot \binom{n-1}{k-1} \cdot (n-k)!$
$\sum_{A \in E_k} X_{n-k}($ rechtes Teilarray von A $) = \sum_{A \in \Omega_{n-k}} X_{n-k}(B) \cdot \binom{n-1}{n-k} \cdot (k-1)!$
Also ist $\sum_{A \in E} \frac{1}{(n-1)!} X_{k-1}($ linkes Teilarray von A$)$
$= \sum_{B \in \Omega_{k-1}} \frac{1}{(n-1)!} \binom{n-1}{k-1} \cdot (n-k)! \ X_{k-1}(B)$
$= \sum_{B \in \Omega_{k-1}} \frac{1}{(k-1)!} \ X_{k-1}(B)$
$= \mathrm{EX}_{k-1}$

$\mathrm{E}[X_n | E_k] = \mathrm{E}[X_{k-1}] + \mathrm{E}[X_{n-k}] \ + n + 1$
$\mathrm{EX}_n = \frac{1}{n}\big(\mathrm{EX}_{n-1} + \mathrm{EX}_{n-1} + (\mathrm{EX}_2 + \mathrm{EX}_{n-3}) + (\mathrm{EX}_3 + \mathrm{EX}_{n-4}) + \ ...$
$+ (\mathrm{EX}_{n-3} + \mathrm{EX}_2) + \mathrm{EX}_{n-2} + \mathrm{EX}_{n-1}\big) \ + n + 1$
$= \frac{2}{n}\big(\sum_{k=2}^{n-1} \mathrm{EX}_k \big) \ + n + 1$

Sie $n \geq 2$ fest, dann
(1) $\mathrm{EX}_n = \frac{2}{n}\big(\sum_{k=2}^{n-1} \mathrm{EX}_k \big) \ + n + 1$
(2) $\mathrm{EX}_{n+1} = \frac{2}{n+1}\big(\sum_{k=2}^{n} \mathrm{EX}_k \big) \ + n + 2$
(3) $(1) \cdot n :$ $n \cdot \mathrm{EX} = 2\big(\sum_{k=2}^{n-1} \mathrm{EX}_k \big) \ + n^2 + n$
(4) $(2) \cdot (n+1) :$ $(n+1)\mathrm{EX}_{n+1} = 2\big(\sum_{k=2}^{n} \mathrm{EX}_k \big) \ + n^2 + 3n + 2$
(5) $(4) - (3) :$ $(n+1) \cdot \mathrm{EX}_{n+1} = (n+2) \cdot \mathrm{EX}_n + 2(n+1)$
(6) $(5) \cdot \frac{1}{n+1} :$ $\mathrm{EX}_{n+1} = \frac{n+2}{n+1} \cdot \mathrm{EX}_n + 2$
(6) gilt für alle $n \geq 2$
Also $\mathrm{EX}_2 = 3, \ \mathrm{EX}_{n+1} = \underbrace{\frac{n+2}{n+1}}_{c_n+1} \cdot \mathrm{EX}_n + \underbrace{2}_{b_{n+1}}$
Es gilt (siehe Übungsaufgabe)
$\mathrm{EX}_2 = 3$, $\mathrm{EX}_{n+1} = \sum_{i=2}^{n+1} \big(\prod_{j=i+1}^{n+1} c_j \big) \cdot b_i$
Produkt bei $c = 3$ ist 1 $i = 2$ ist c_3
$\mathrm{EX}_3 = c_3 \cdot \mathrm{EX}_2 + b_3$
$= \sum_{i=2}^{3} \big(\prod_{j=i+1}^{3} c_j \big) \cdot b_i$
$= c_3 \cdot b_2 + b_3$

Jetzt $\prod j = i + 1^{n+1} \cdot c_j = \prod_{j=i+1}^{n+1} \frac{j+1}{j}$
$= \frac{i+2}{i+1} \cdot \frac{i+3}{i+2} \cdot \ ... \ \cdot \frac{n+2}{n+1}$
$= \frac{n+2}{i+1}$
Also $\mathrm{EX}_{n+1} = \sum_{i=2}^{n+1} \frac{n+2}{i+1} \cdot b_i$ $(b_i \in \{2, 3\})$
$\leq 3 \cdot \sum_{i=2}^{n+1} \frac{n+2}{i+1}$
$= 3(n+2) \cdot \sum_{i=2}^{n+1} \frac{1}{i+1}$
$= 3(n+2) \cdot \big(\sum_{i=1}^{n+1} \frac{1}{i+1} - \frac{1}{2} \big)$
$= 3(n+2) \cdot \big(\sum_{i=2}^{n+2} \frac{1}{i} - \frac{1}{2} \big)$
$= 3(n+2) \cdot \big(\sum_{i=2}^{n+2} \frac{1}{i} - 1 - \frac{1}{2} \big)$

$\leq 3(n+2) \cdot (\underbrace{\sum_{i=1}^{k} \frac{1}{i}}_{=H_k}) = 3(n+2)\mathrm{H}_n$ (n-te Harmonische Zahl)

$= 3(n+2)O(\ln n) = O(n \log n)$!

$\mathrm{EX}_n = O(n \log n)$ im Mittel optimal

Rekursionsgleichung:
$\mathrm{EX}_n = \sum_{k=1}^{n} \mathrm{Prob}(E_k) \cdot \mathrm{E}[X_n|E_k]$. Berechnung von Erwartungswerten durch Bedingung

$\mathrm{E}[X_n|E_k] = \mathrm{EX}_k + \mathrm{E}_{n-k} + \underbrace{n+1}_{\text{max Anz. von Vergleichen}}$

Beachte uniforme Verteilung auf $\mathrm{E}_k \leq \Omega_n$ wird transformiert durch Partition in uniforme Verteilung auf Ω_{k-1} (linkes Teilarray) und Ω_{n-k} (rechtes Teilarray).

Kapitel 11

Randomisiertes Quicksort

Vermeidung des worst-case von Quicksort durch Randomisieren

11.1 Algorithmus Randpartition Randquicksort

Randpartition(A, l, r)
1. i:=Random(l, r)
2. Vertausche A[l] und A[i]
3. Partition(A, l, r)
Random(l, r) liefert eine Zahl aus $\{l, l+1, l+2, ..., r\}$
dabei ist Random(l, r): $\Omega \rightarrow \{l, ..., r\}$ eine Zufallsverteilung mit uniformer Verteilung, d.h. Prob[Random(l, r)=i]=$\frac{1}{l-r+1}$ für jedes i
Randquicksort - Quicksort mit Randpartition

Bemerkung:
(1) Was ist ein Randomisierter Algorithmus ? Für jede feste Eingabe bekommen wir einen Wahrscheinlichkeitsraum von Berechnungen, dargestellt als Wahrscheinlichkeitsbaum
(2) Wahrscheinlichkeit einer Berechnung = Produkt der Wahrscheinlichkeiten an den Kanten
(3) Eine RAM (Randon-Access-Machine) die Randquicksort implementiert macht ein Zufallsexperiment in dem Baum
Random(l, r) hat Zeit O(1)
(4) Laufzeit ? Für feste Eingabe A haben wir eine Zufallsverteilung
X_A Raum der Berechnungen mit $A \rightarrow \mathbb{N}$
X_A(Berechnung)= Anzahl der Vergleiche von Array-Elementen, die in der Berechnung gemacht werden
Interessant wäre: EX_A für jedes A, auch Max$\{EX_A|$ A Eingabe$\}$
Hier Prob$[X_A \geq c \cdot n \cdot \log n]$ für geeigntes c
Prob$\{$Berechnungen$vertX_A$(Berechnungen)$\geq c \cdot n \cdot \log n$ $\}$

11.2 Definition (Berechnungsbaum von A)

Sei $A = (a_1, ..., a_n)$ Permutationen auf {1...n}
Der Wahrscheinlichkeitsbaum T der Berechnungen von A ist definiert durch Knoten von $T : 1, .., n$ auf den Plätzen von A mit einigen Elementen fixiert
Also z.B. $(\underline{1}, 2, \underline{3}, ...\underline{\frac{n}{2}+1}, ...a_n)$
die anderen liegen richtig zwischen den Fixierten

Ein Knoten K bestimmt folgendes Experiment: Suche das am weitesten links stehende Teilarray, nicht sortiert, der Länge $k \geq 2$

Fälle:

(a) $\underline{i}$, ... , _

(b) ... , $\underline{i}$

(c) ... , $\underline{i}$, ... (schon sortiert)

Dann bekommt K im Baum k Kinder gemäß

... , $\underline{i}$, ... i_k , $\underline{j}$

$j \equiv i_{k+1}$

Die Wahrscheinlichkeit, daß Partition mit einem Element zwischen i und j ausgefuehrt wird ist $\frac{1}{k}$

Hat K kein nicht finites Stück, der Länge $k \geq 2$, so ist K Blatt von T

Wurzel ist $A(a_1, ..., a_n)$

Jeder Knoten K bestimmt Wahrscheinlichkeitsraum $P_k = (\Omega_k, p(\Omega_k), \mathrm{Prob}_k)$ mit

Ω_k = Menge der Wege von K zu einem Blatt

$\mathrm{Prob}_k(\underbrace{K0}_{=K}, K_1, ..., K_m)$ = Produkt der Übergangsmöglichkeiten

$(\mathrm{Prob}_k(A) = \sum_{\omega \in A} \mathrm{Prob}_K(\omega))$

Das ist Wahrscheinlichkeitsraum, zeigen dazu $\sum_{\omega \in \Omega_k} \mathrm{Prob}_K(\omega) = 1$

Induktion über die längste Berechnung von K aus (=Maximale Weglänge bis zu einem Blatt)

Sei maximale weglänge =1, dann Situation

K

$\frac{1}{k}$ $\frac{1}{k}$

K_1 ... K_k

$\mathrm{Prob}_k(K, K_i) = \frac{1}{k}$ für $1 \leq i \leq k$

Behauptung gilt. Sei maximale Weglange $\gneqq 1$

K

K_1 ... K_k

Dann

$\Omega_k = \{\underbrace{K_0}_{=K}\ K_1\ \omega_{1_1}, ..., K_0\ K_1\ \omega_{1_{l_1}}; K_0, K_2\ \omega_{2_1}, ..., K_0\ K_2\ \omega_{2_{l_2}}; ...; K_0, K_k\ \omega_{k_1}, ..., K_0\ K_k\ \omega_{k_{l_k}}\}$

$K_0 = K$ Rechnungen der Länge ≥ 0 Dann ist $\mathrm{Prob}_K(K_0, K_1, \omega_{1,1}) + ... + \mathrm{Prob}_K(K_0, K_1, \omega_{k,l_k}) +$
$= \frac{1}{k}\big(\mathrm{Prob}_{K_1}(K_1, \omega_{1,1}) + ... + \mathrm{Prob}_{K_1}(K_1, \omega_{1,l_1})\big) + ... + \frac{1}{k}\big(\mathrm{Prob}_{K_k}(K_k, \omega_{k,1}) + ... +$
$\mathrm{Prob}_{K_k}(K_k, \omega_{k,l_k})\big)$
$= k\frac{1}{k} = 1$

11.3 Satz

Sei $A = (a_1 ... a_n)$ fest, beliebig, dann gibt es ein festes c mit:

$\mathrm{Prob}[X_A \geq c \cdot n \cdot \log n] = O(\frac{1}{n^6})$

(X_A ist $O(n \cdot \log n)$ mit Wahrscheinlichkeit $\rightarrow 1$)

Beweis: Sei $A = (a_1 ... a_n)$ fest gewählt ! Müssen X_a irgendwie verfolgen. Idee ist so: verfolgen die

Anzahl von Vergleichen, an denen a_1 als Nicht-Pivot-Element teilhat = Y_1

(Y_1 = Zufallsverteilung auf Raum von A)

Anzahl von Vergleichen, an denen a_2 als Nicht-Pivot-Element teilhat = Y_2

Haben also $Y_1, Y_2, ..., Y_n$: Berechnungen$\rightarrow \mathbb{N}$

Für alle Berechnungen von A, ω gilt:
$X_A(\omega) = Y_1(\omega) + ... + Y_n(\omega)$
Die Idee ist jetzt weiterhin so: Sei also $a = a_i$ fest. Wenn Y_i einen Vergleich mit a_i zählt, sind wir in einem Übergang $K = ...j...a...l...$
Im "Normalfall" wird, wenn Y_i einen Vergleich mit $a = a_i$ zählt, das Teilarray von a um einen linearen Faktor kleiner (etwa halbiert).Nehmen wir den günstigsten Fall an, daß wir eine Rechnung ω vorliegen haben, wo das Teilarray von a jedesmal halbiert wird. Dann ist $Y_i(\omega) = \log(n)$
Wir nehmen jetzt folgendes an: Für geeignetes c, konstant, ist $\text{Prob}[Y_i \geq c \cdot \log n] \to 0$
Dann bekommen wir
$\text{Prob}[X_A \geq c \cdot n \cdot \log n]$

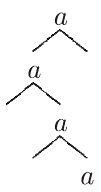
Bem: $X_A(\omega) = Y_1(\omega) + ... + Y_n(\omega)$

$= \text{Prob}\{\omega \in \Omega_A | Y_1(\omega) + Y_2(\omega) + ... + Y_n(\omega) \geq c \cdot \log n\}$

(d.h.: mindestens ein $Y_i(\omega) \geq c \cdot \log n$)

$\leq \text{Prob}\{\omega \in \Omega_A | Y_1(\omega) \geq c \cdot \log n \text{ oder } Y_2(\omega) \geq c \cdot \log n \text{ oder...oder } Y_n(\omega) \geq c \cdot \log n\}$

$= \text{Prob}(\{\omega \in \Omega_A | Y_1(\omega) \geq c \cdot \log n\} \cup \{\omega \in \Omega_A | Y_2(\omega) \geq c \cdot \log n\} \cup ... \{\omega \in \Omega_A | Y_n(\omega) \geq c \cdot \log n\})$
$\text{Prob}(A \cup B) \leq \text{Prob}(A) + \text{Prob}(B)$
$\leq \underbrace{\text{Prob}[Y_1 \geq c \cdot \log n]}_{\to 0 \text{ nach Annahme}} + ... + \text{Prob}[Y_n \geq c \cdot \log n]$
Wissen wir nur, daß für alle i $\text{Prob}[Y_i \geq c \cdot \log n] \leq \frac{1}{n}$, dann wissen wir nur $\text{Prob}[X_A \geq c \cdot n \cdot \log n] \leq n \cdot \frac{1}{n} = 1$
Ist aber $\text{Prob}[Y_i \geq c \cdot \log n] \leq \frac{1}{n^2}$, so bekommen wir:
$\text{Prob}[X_A \geq c \cdot n \cdot \log n] \leq n \cdot \frac{1}{n^2} = \frac{1}{n} \to 0$
Sei $a = a_i$ irgendwie fest aus A (A ist auch fest) Sei $Y = Y_i$ $Y(\omega) = 0$ auf den Rechnungen, wo a direkt Pivot-Element ist
Wie ist $Y(\omega)$ am günstigsten (wenn a bis zum Ende dabei bleibt)



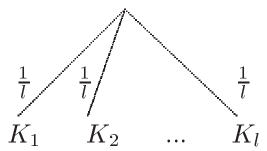
1 Verleich mit a

1 Verleich mit a

Insgesamt $\log n$ Vergleiche mit a

Wir betrachten folgende Situation: Irgendwo (im Baum)
Rechnung:

$K = (\overbrace{.....}^{fertig} \overbrace{a_i ... a ... a_j}^{nicht\ sortiert}$



Es nimmt also a entweder als Nicht-Pivot-Element an einem Vergleich teil, oder es wird Pivot
Wir sagen der Übergang $K - K_j$ ist gut $\Leftrightarrow$ das Teilarray von a wird in 2 Teile geteilt,

so daß jedes von diesen $\leq \frac{7}{8} \cdot l$ Elemente hat (l = Anzahl Elemente im Teilarray)
Bsp:
(1 . . . . a=50 . . . . . . 100)
(1 . . . 29 <u>30</u> . . . . a=50. . . . . . 100) ist gut, da $70 \leq \frac{7}{8} \cdot 100$, nicht gut dagegen ist:
(1 . . . . a=50 . . . . . . 100)
(1 . . . 9 <u>10</u> . . . . 100) denn $91 \geq \frac{7}{8} \cdot 100$.
Ist k wie oben, dann ist in Wahrscheinlichkeitsraum der (Einzel-)übergänge
$\text{Prob}\{K - K_j | K - K_j \text{ist nicht gut}\} \leq \frac{1}{8} + \frac{1}{8} = \frac{1}{4}$
Betrachten wieder ganze Rechnung $(\underbrace{A_0}_{=A} \; A_1 \; \ldots \; \underbrace{A_m}_{\text{sortiert}} \;)$
Wo ist die Anzahl guter Übergänge mit $a \leq \log_{\frac{8}{7}} n$ (Haben also feste Grenze an die Anzahl guter Übergänge wo a daran teilhat) Was ist nun $\text{Prob}[Y \geq 20 \cdot \log_{\frac{7}{8}} n]$?
Ist ω so, daß $Y(\omega) \leq 20 \cdot \log_{\frac{7}{8}} n$ ist, so nimmt a an $20 \log_{\frac{8}{7}} n$ Aufteilungen, also nicht-Pivot, teil. Also muß a an mindestens $19 \cdot \log_{\frac{8}{7}} n$ schlechten Übergängen teilnehmen. (Wahrscheinlichkeit für schlechten Übergang $\leq \frac{1}{4}$, $19 \cdot \log_{\frac{8}{7}} n$ mal Wahrscheinlichkeit $\leq \frac{1}{4}$ ist sehr unwahrscheinlich)
Problem zum Berechnen der Wahrscheinlichkeit, wissen zunächst nicht, an welchen Übergängen das a teilhat. Gehen davon aus, daß unsere Rechnungen so sind:

$$\underbrace{(\underbrace{A_0}_{=A}, \; \underset{a}{\underset{|}{A_1}}, \; \underset{a}{\underset{|}{A_2}}, \; \ldots \; \underbrace{, A_m}_{\text{a nicht mehr}}}_{\geq 20 \cdot \log_{\frac{8}{7}} n}$$

Nun ist, wenn alle Vergleiche mit a vorne sind (etwa wenn $a = 1$)
$\text{Prob}[Y_a \geq 20 \cdot \log_{\frac{8}{7}} n]$
$\leq \sum_{j \leq 19 \cdot \log_{\frac{8}{7}} n} \binom{20 \cdot \log_{\frac{8}{7}} n}{j} \cdot (\frac{1}{4})^j \cdot (\frac{3}{4})^{20 \cdot \log_{\frac{8}{7}} n - j}$
$\leq \sum \binom{20 \cdot \log_{\frac{8}{7}} n}{j} \cdot (\frac{1}{4})^j$
$\leq \sum \binom{20 \cdot (\log_{\frac{8}{7}} n) \cdot e}{j} \cdot (\frac{1}{4})^j$

$\leq \sum_{j \leq \ldots} \left(\frac{20 \log n \cdot e}{19 \log n}\right)^j \cdot (\frac{1}{4})^j$

$\leq \sum_{19 \cdot \log_{\frac{8}{7}} n} \left(\frac{5 \cdot e}{19}\right)^j \cdot (\frac{1}{4})^j \;\; \leq O(\frac{1}{n^7})$ wegen der geometrischen Reihe
Wie behandeln wir die Situation, wo die Vergleiche von a nicht vorne stehen? Mit der Formel von der totalen Wahrscheinlichkeit
$\big(\text{Prob}B = \text{Prob}(\Omega_1) \cdot \text{Prob}(B|\Omega_1) + \text{Prob}(\Omega_2) \cdot \text{Prob}(B|\Omega_2)\big)$
(wobei Zerlegung $\Omega = \Omega_1 \cup \Omega_2$)
Wir nehmen die Zerlegung des Raumes der Rechnungen
Ω_1 = Alte Vergleiche mit a ganz vorne
Ω_2 = Möglichkeit 2 der Vergleiche mit a
Ω_3 = Möglichkeit 3 der Vergleiche mit a
... ...
Dann $\sum \text{Prob}(\Omega_i) = 1$ und die relative Häufigkeit ist jedesmal gleich abzuschätzen.