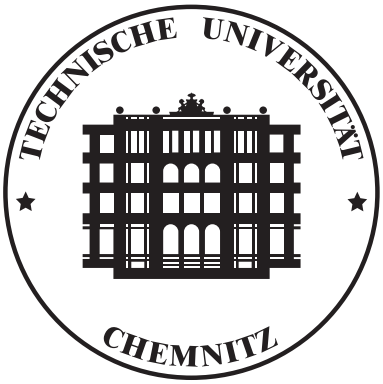
Andreas Goerdt

Theoretische Informatik III

Skriptum zur Vorlesung

Sommersemester 2007



Anstelle eines Vorwortes

Das vorliegende Skriptum entstand in einer Rohfassung als Arbeit studentischer Hilfskräfte aus Vorlesungsmitschriften der Sommersemester 1997 und 1998, wurde im Sommersemester 2001 von Olaf Hartmann und Frank Schädlich, sowie im Sommersemester 2007 von André Lanka überarbeitet.
Für Hinweise auf Fehler oder andere Verbesserungen sind wir dankbar. Bitte wenden Sie sich an die Professur.

Chemnitz, den 25.06.2007
André Lanka

Inhaltsverzeichnis

1 Binomiale Heaps **7**
- 1.1 Einführung . . . 7
- 1.2 Der binomiale Baum . . . 9
  - 1.2.1 Aufbau . . . 9
  - 1.2.2 Implementierung binomialer Bäume . . . 11
- 1.3 Der binomiale Heap . . . 11
  - 1.3.1 Struktur . . . 11
  - 1.3.2 Operationen auf dem binomialen Heap . . . 12
- 1.4 Binomialer Heap mit „lazy“ `meld`(h,h') . . . 16

2 Fibonacci-Heaps **21**
- 2.1 Einführung . . . 21
- 2.2 Die Datenstruktur des Fibonacci-Heap . . . 23
- 2.3 Laufzeiten . . . 27

3 Union-Find-Strukturen **29**
- 3.1 Einführung . . . 29
- 3.2 Partition als Menge von Bäumen . . . 30
- 3.3 Algorithmus Union-By-Size mit Wegkompression . . . 31

4 Selbstorganisierende Listen **39**
- 4.1 Datenstruktur . . . 39
- 4.2 Heuristiken . . . 40
- 4.3 Kosten . . . 41

5 Selbstorganisierende Bäume **47**
- 5.1 Algorithmus für geschicktes Rotieren, Splaying . . . 47
- 5.2 Datenstruktur Splaybaum . . . 49
- 5.3 Definitionen . . . 51
- 5.4 Schlüssellemma . . . 52

Kapitel 1

Binomiale Heaps

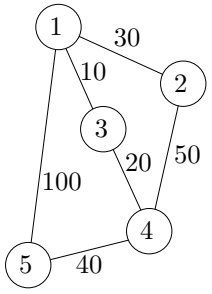
1.1 Einführung

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 Dijkstras Algorithmus betrachtet.

Dijkstras Algorithmus:

```
Array D := D[1..n]
D[1] := 0
for i := 2 to n do
   // ω(h,i) = ∞ wenn Kante (h,i) nicht 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] = ... = D[5] = ∞

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:

$n-1$ Knoten in Heap einfügen	=	$O(n)$
$(n-1)$-mal Minimum finden und löschen	=	$O(n \log n)$
$\|E\|$ (Anzahl der Kanten) mal $V\backslash S$ anpassen in $O(\log n)$	=	$O(\|E\| \log n)$
Laufzeit insgesamt bei $\|E\| \geq n-1$	=	$O(\|E\| \log n)$ [1]

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

$(n-1)$ mal Minimum finden und löschen	=	$O(n^2)$
D-Werte ändern	=	$O(\|E\|)$
Laufzeit insgesamt	=	$O(n^2)$

Der Heap ist also besser, solange

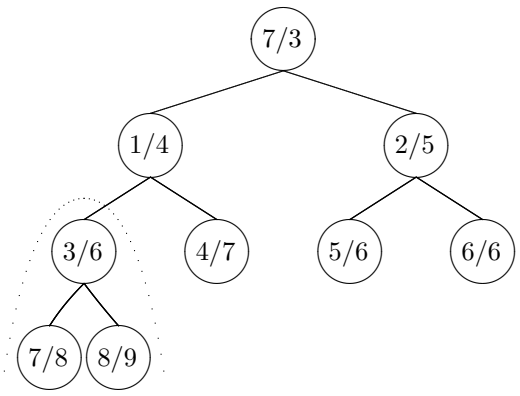
$$|E| = O\left(\frac{n^2}{\log n}\right), \quad \frac{n^2}{\log n} \geq n^{2-\epsilon} \quad \forall \epsilon > 0 \text{ und } n \to \infty.$$

Daraus ergibt sich die maximale Laufzeit für Dijkstras Algorithmus: $O((n + |E|) \cdot \log n)$. $O(n \log n)$ 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 keinen Fall 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 Dijkstras Algorithmus auf $O(n \log n + |E|)$ verbessern.

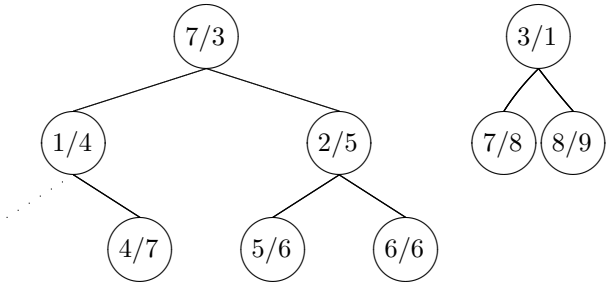
Bemerkung: Damit wird die Laufzeit nie schlechter als die der „naiven“ Implementation.

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 heraus. Dadurch erhalten wir 2 Bäume mit Heapeigenschaft:

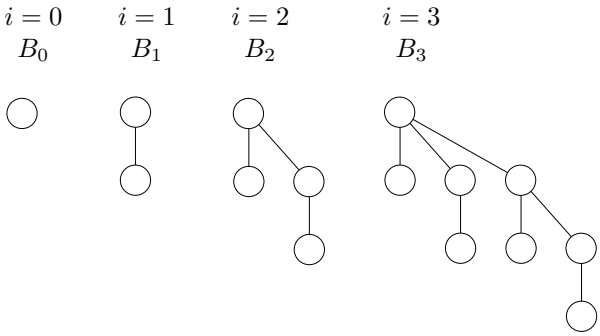


Man braucht also eine Möglichkeit, kontrolliert Bäume (=Heap) zu teilen und zusammenzusetzen.

1.2 Der binomiale Baum

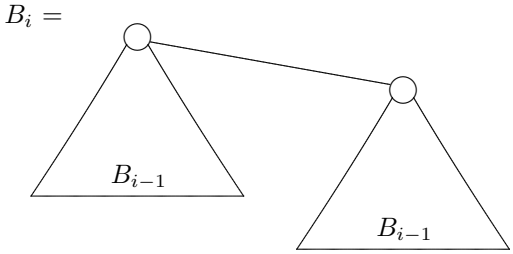
1.2.1 Aufbau

Induktiver Aufbau binomialer Bäume B_i für jedes $i \geq 0$



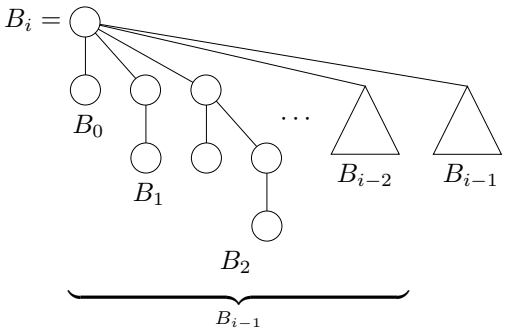
Definition 1.2.1. Der i-te binomiale Baum wird induktiv über i definiert:

Der 0-te binomiale Baum ist ein einzelner Knoten ◯.
Für $i > 0$ hat der i-te binomiale Baum die Struktur



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

Satz 1.2.2. (a) Sei $i > 0$. Dann hat der i-te binomiale Baum folgendes Aussehen:



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

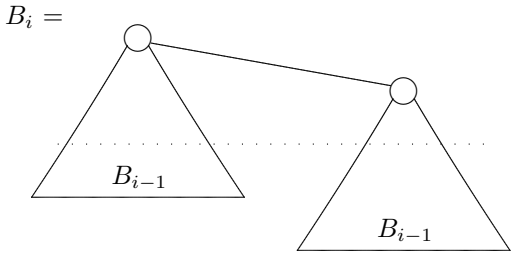
(b) Tiefe(B_i)$= i$ für alle $i \geq 0$.

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

(d) $|\{x|\ x \text{ ist Knoten von } B_i\ \wedge\ \text{Tiefe}(x) = j\}| = \binom{i}{j}$

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 $i - 1 \geq j \geq 1$

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

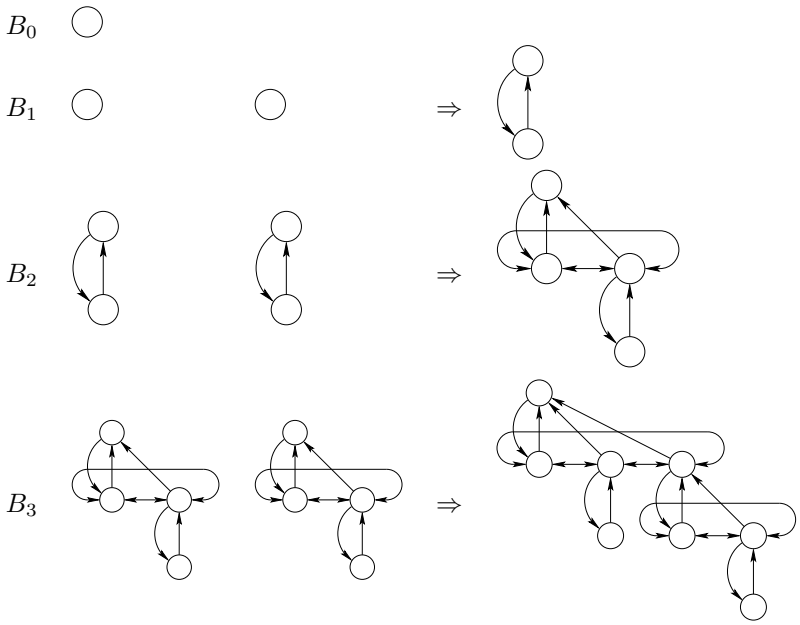
Gilt $i = j$, so hat der binomiale Baum B_{i-1} in Tiefe i nach (b) keinen Knoten. Es gilt also:

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

1.2.2 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
  Füge Wurzel(B) als weiteren Sohn von Wurzel(C) ganz
         rechts ein
else
  Füge Wurzel(C) als weiteren Sohn von Wurzel(B) ganz
         rechts ein
```

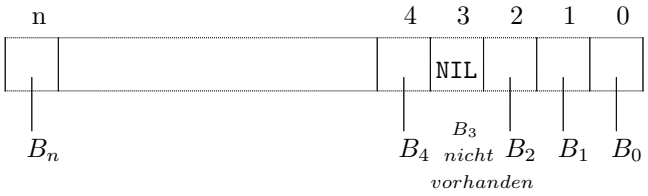
Die Operation `link`**(B, C)** hat wegen der Wahl der Pointer die Laufzeit O(1).

1.3 Der binomiale Heap

1.3.1 Struktur

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 1.3.1. 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$\{i \mid \mathrm{A}[i] \neq$ `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:

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

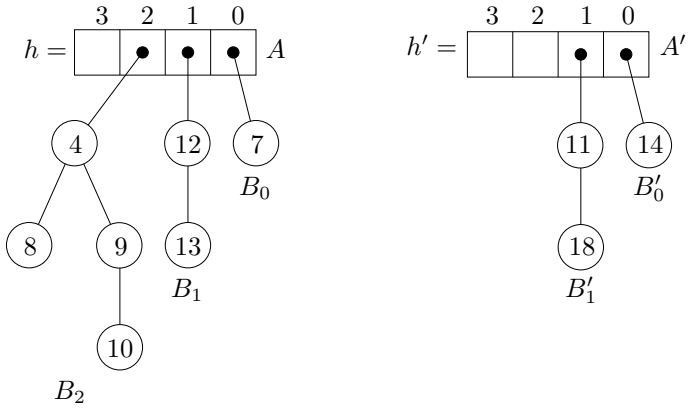
Die untere Schranke ist klar. Die obere folgt wegen:

$$\sum_{i=0}^{\texttt{heapsize}} 2^i = 2^{\texttt{heapsize}+1} - 1.$$

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

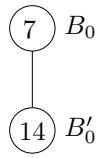
1.3.2 Operationen auf dem binomialen Heap

(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 1.2.2)
Am Beispiel:

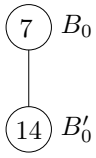


Vergleiche:
$$\begin{array}{r} 0111 \\ +0011 \\ \hline 1010 \end{array}$$

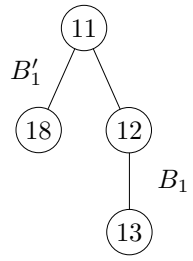
$B_0 + B_0'$ = `NIL` Übertrag:



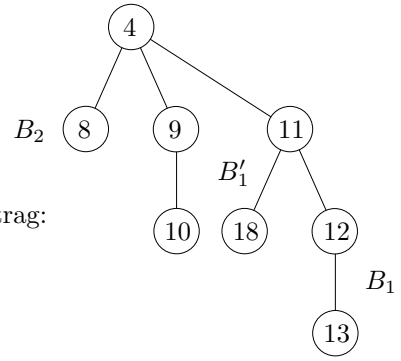
$B_1 + B_1' +$ Übertrag =



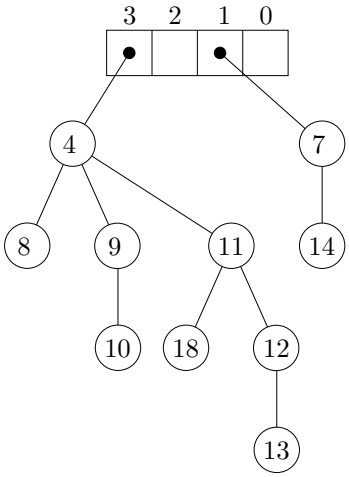
Übertrag:



$B_3 +$ Übertrag = `NIL` Übertrag:



`meld`(h, h') =



Die Laufzeit von `meld`(h,h') ist $O(\log|h| + \log|h'|)$, 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: ($n = |h|$)
1. $O(1)$
2. $O(\log n)$
3. $O(1)$
4. $O(\log n)$
5. $O(\log n)$

Insgesamt $O(\log n)$.

Satz 1.3.2. Das Einfügen von n Elementen in den anfänglich leeren binomialen Heap erfolgt in Zeit $O(n)$.

Beachte: Obwohl einzelne Einfügeoperationen $\Theta(\log n)$ dauern können, ist die Gesamtzeit für n `insert`s nur $O(n)$.

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
0 `link` ein `link` gezählt

`insert`(2,2) Zeit: 1 `makeheap`
1 Eintrag in h
1 `link` ein `link` gezählt

`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 u (ohne Finden)nd 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)

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

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

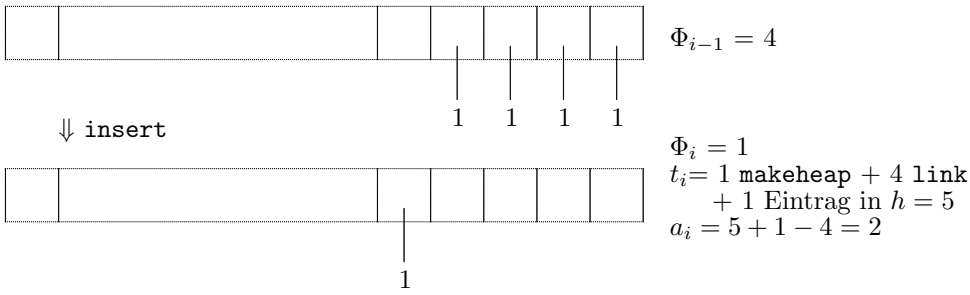
(Könnte $\Omega(\log i)$ sein.)
(Was ist $\Phi_i - \Phi_{i-1}$? Die Differenz der gespeicherten Zeiten.)

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

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

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 $a_i = t_i + \Phi_i - \Phi_{i-1}$



Satz 1.3.3. Führen wir eine Folge σ von m Operationen $\sigma_1, \sigma_2, \ldots, \sigma_m$ von m_1 `insert`s, m_2 `deletemin`s und m_3 `min`s ($m_1 + m_2 + m_3 = m$) 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($\log n$).

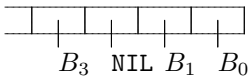
Beweis: Der Beweis dieses Falls erfolgt in der Übung.
Zusammenfassung

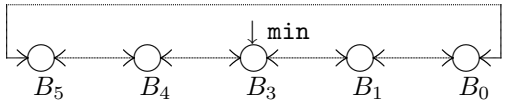
Operation	Direkte Adressen	Binärer Heap (Worst-Case)	Binomialer Heap (Worst-Case)	Binomialer Heap (amortisiert)
`insert`	O(1)	O($\log n$)	O($\log n$)	O(1)
`deletemin`	O(1)	O($\log n$)	O($\log n$)	O($\log n$)
`min`	O(n)	O(1)	O(1)	O(1)

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

(a) Statt eines Arrays wird nun eine doppelt verkettete zirkuläre Liste zum Verwalten der binomialen Bäume verwendet:

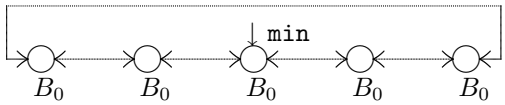
Vorher h= jetzt:





Beachte: *keine* Leerstelle für B_2

Sogar noch allgemeiner: ein i-ter binomialer 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 ($O(\log n)$).

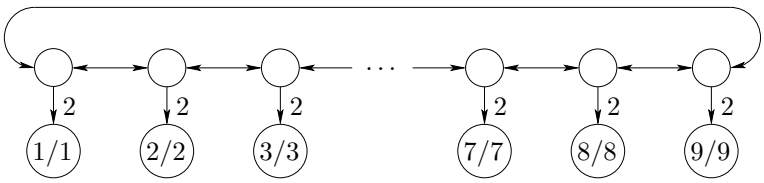
Zeit im Worst-Case: $O(n)$.

Idee: Zeit bei den Wurzeln der Bäume speichern, um die Schranke an der Worst-Case zu verbessern.

Satz 1.4.1. Beginnen wir mit leerem binomialen Heap (mit „lazy“ meld), haben wir

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

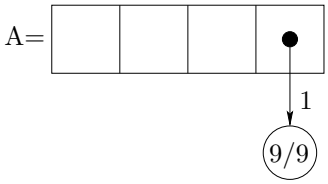
Beispiel:
Nach 9 inserts haben wir die Situation



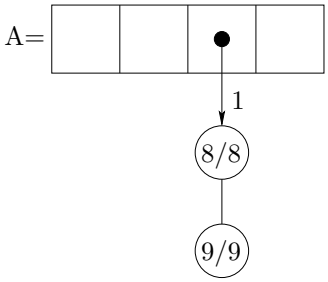
Was geschieht jetzt bei deletemin(h)?

1. 

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

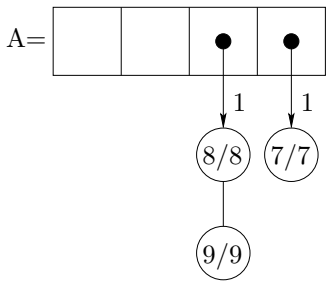
2. 

gebrauchte Zeit $O(1)$, zählen 0

3. 

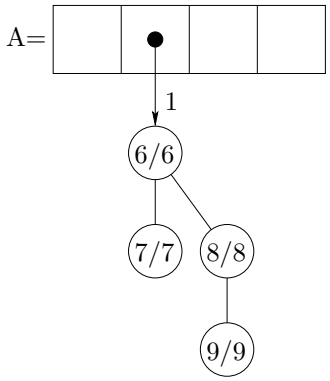
gebrauchte Zeit $O(1)$, zählen 0

4.



gebrauchte Zeit O(1), zählen 0

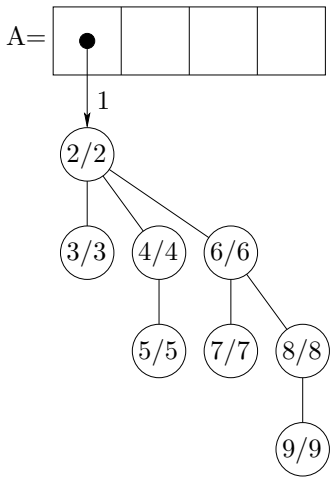
5.



gebrauchte Zeit O(2), zählen 0

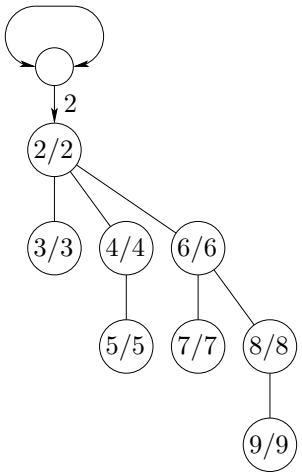
⋮ ⋮

9.



0 gezählt für den Aufbau des Heaps.

10.



Wir zählen jetzt 2. Beim Löschen zerfällt Heap in $\log n$ Teilbäume. An deren Wurzeln ist keine Zeit gespeichert. Diese $\log n$ Bäume werden in die zirkuläre Liste eingefügt, dies geschieht in $O(\log n)$. Um das Minimum zu finden, müssen sämtliche Wurzeln betrachtet werden, dies geschieht ebenfalls in $O(\log n)$.
Somit hat `deletemin` die amortisierte Laufzeit $O(\log n)$.

Beweis:
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.

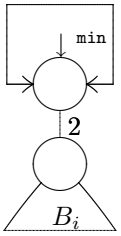
$$h_0 = 0 \xrightarrow[t_1]{\sigma_1} h_1 = 1 \xrightarrow[t_2]{\sigma_2} h_2 \xrightarrow[t_3]{\sigma_3} \cdots \xrightarrow[t_m]{\sigma_m} h_m$$

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

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

(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:

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

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 der Dijkstra-Algorithmus eine Laufzeit von O($|E| \log |V|$) wenn $|E| \geq |V|$ gilt.

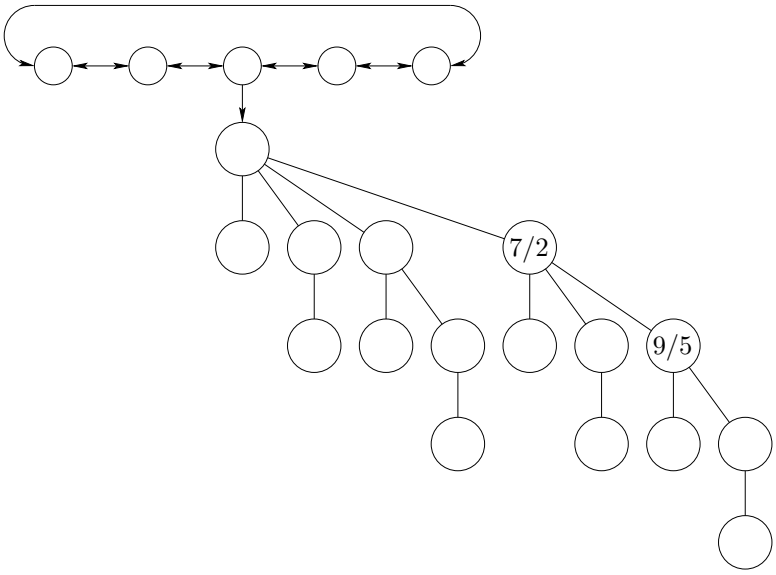
Kapitel 2

Fibonacci-Heaps

2.1 Einführung

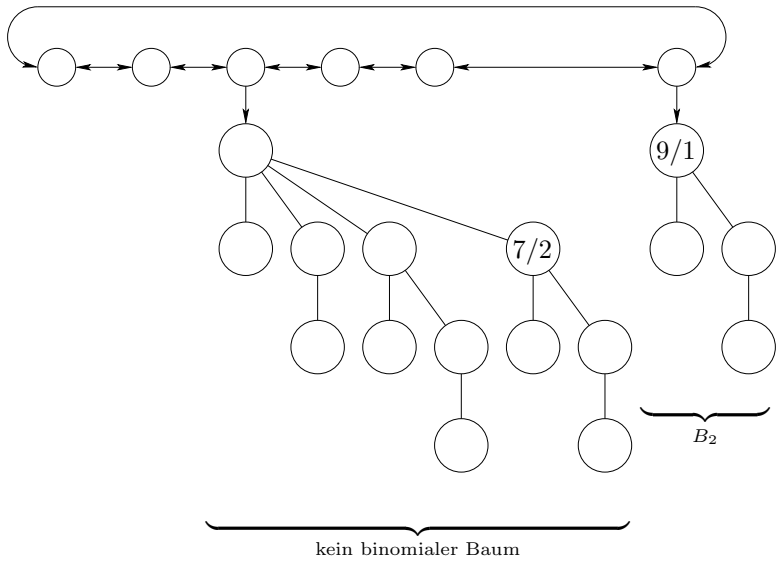
Unser Ziel ist es, den binomialen Heap so zu modifizieren, daß die amortisierte Laufzeit von `decreasekey`(i, j, h) O(1) ist.

Beispiel 1:



`decreasekey`(9, 1, h)

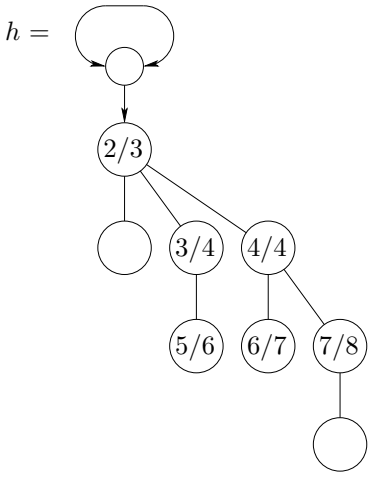
Idee: Herausschneiden und Einschmelzen (in O(1)) ergibt:



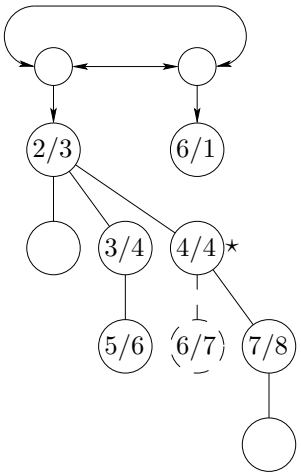
Problem: Es muß sichergestellt werden, daß durch iteriertes Herausschneiden die Kinderzahl im Verhältnis zu Elementanzahl nicht zu groß wird. Die Bäume dürfen also nicht zu breit und flach werden.

Beispiel 2:

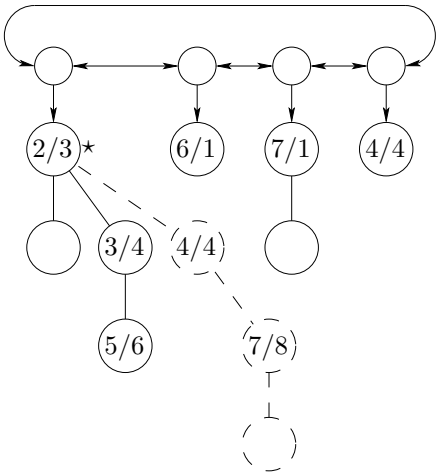
Wie soll das gehen? Die Vermeidung gar zu breiter flacher Bäume erfolgt, indem pro (nicht-Wurzel-) Knoten immer nur ein Kind abgeschnitten wird. Das geht so:



`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.

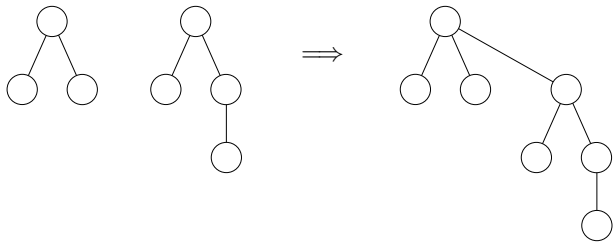
2.2 Die Datenstruktur des Fibonacci-Heap

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`. (1.4)
Da es sich im allgemeinen nicht mehr um binomiale Bäume handelt, wird statt des i-ten binomialen Baumes der Baum genommen, dessen Wurzel genau i Kinder hat.



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

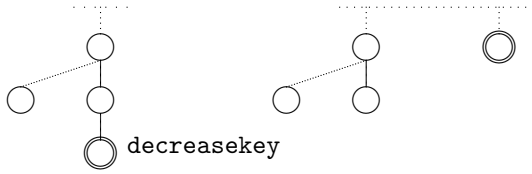
(d) `cut`(x, h)
Herauschneiden des Teilbaumes, dessen Wurzel der Knoten x ist.

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).

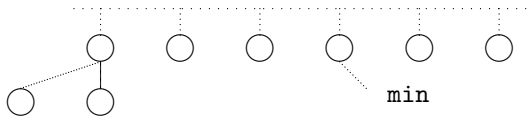
(e) `decreasekey`(x, j, h)
$j \leq$ alter Schlüssel von x.

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

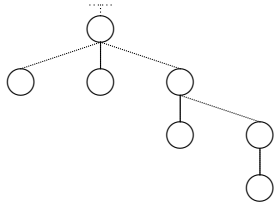
Beispiel
`deletemin`



4 mal `insert`



Nach `deletemin`:



Beachte: Es werden nur Bäume kombiniert, wenn ihre Wurzeln die gleiche Anzahl Kinder haben.

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

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

$$|\mathrm{T}| \geq 2^{c \cdot \text{Anzahl Kinder von } r}$$

für ein festes c $(0 < c < 1)$.

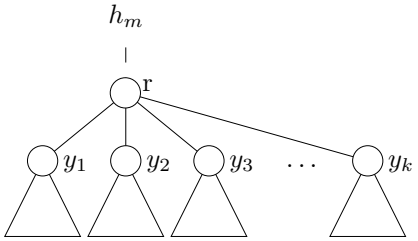
Also: Anzahl Kinder $\leq \frac{1}{c}(\log |\mathrm{T}|) = O(\log |\mathrm{T}|)$.

Beachte: Im binomialen Baum ist $c = 1$.

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:



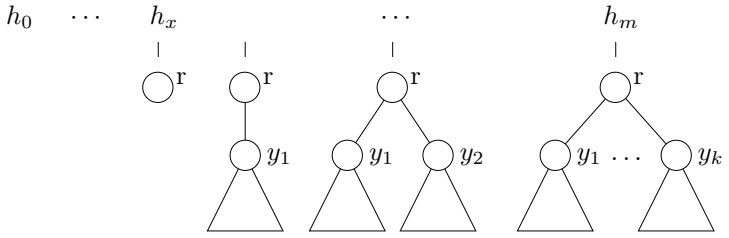
$$k \geq 0$$

(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

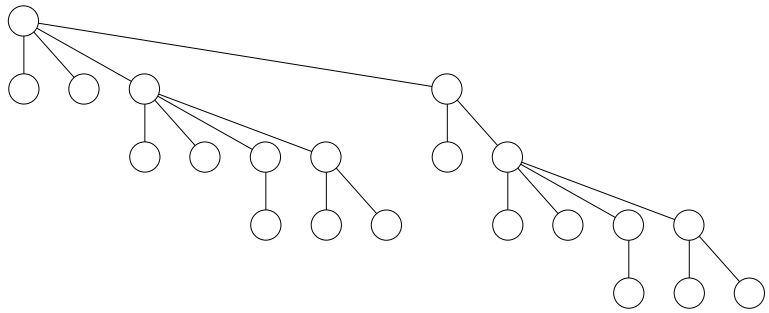
Zeitverlauf:



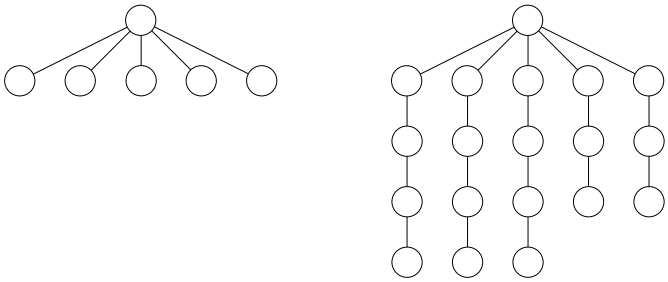
r kann noch weitere Kinder gehabt haben, die aber zwischendurch wieder gelöscht wurden.
Nun gilt die Schlüsselaussage:

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

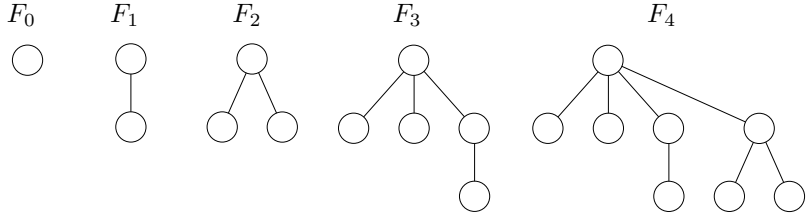
Beweis (der Schlüsselaussage):
Wenn y_i an r kommt, hat r mindestens die Kinder $y_1, \ldots, y_{i-1}$. Wegen `link` hat y_i dann auch $\geq i - 1$ Kinder. Bis zum Ende kann y_i wegen `cut` maximal ein Kind verlieren. Somit hat y_i am Ende mindestens $i - 2$ Kinder.
Also haben die Bäume in etwa die folgende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

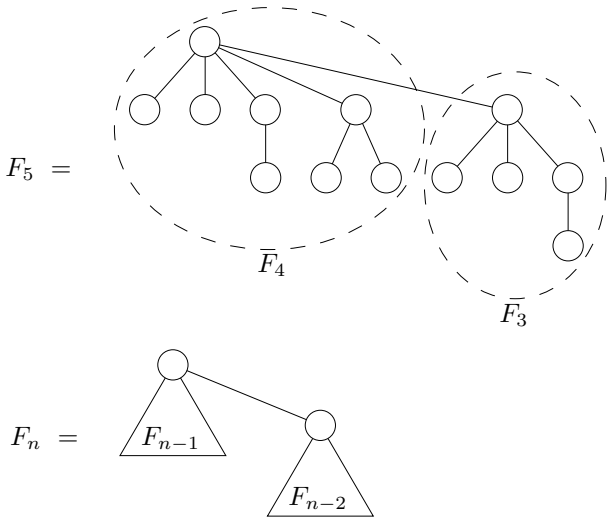


Derartige Bäume sind ausgeschlossen:



Die Frage ist: Wieviel Bäume können nur auftreten?
Sei $n \geq 0$ und sei F_n ein kleinster Baum, ein Baum der möglichst wenig Elemente enthält, 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:





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 1.4.1.
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 aufsteigend nach ihrem Gewicht sortieren.
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`)

⇒ 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 $\mathrm{A}[1] = \mathrm{A}[2] = 1$

A[3] = A[4] = A[5] = 2 | 1 1 2 2 2 3 3 |
A[6] = A[7] = 3

⇒ `find`: O(1) `union`: im Worst-Case $\Theta(n)$
Kruskal mit der Partition als Array benötigt Laufzeit:

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

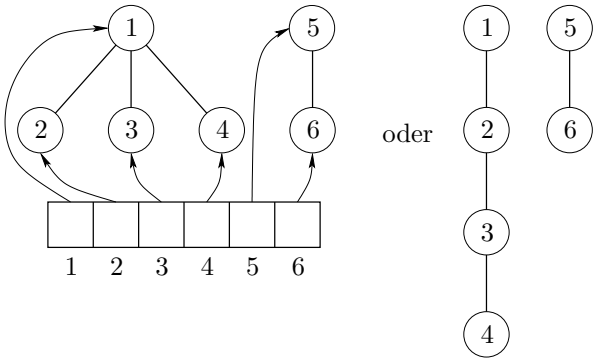
Das ist $O(|E| \cdot \log|V| + |V|^2)$.
Ist $|E| = O(\frac{|V|^2}{\log|V|})$, dann $O(|V|^2)$ (gilt nur nicht für sehr dichte Graphen).

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:

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

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 $O(\log|T|)$ beschränkt.

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 $O(\kappa + (n + \lambda) \cdot \log^* n)$. Dabei ist $\log^*$ definiert durch:

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

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

$\log^* n$ ist fast O(1). Beispiel: $\log^* 2^{16} = \log^* 65\,536 = 4$.
$\log^* 2^{65\,536} = \log^* 10^{19728} = 5$.

Beachte: Für Kruskals Algorithmus ist $\kappa = n - 1 = |V| - 1$ und $\lambda = 2|E|$. Bei vorsortierten Kanten ergibt sich eine Laufzeit von $O(|E| \log^* |V|)$.

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) (2) (3) ... (n) die anfängliche Union-Find-Struktur.

Sei $\sigma = \sigma_1, \sigma_2, \sigma_3, \ldots, \sigma_m$ eine Folge von `union`- und `find`-Operationen. Sei weiterhin $\lambda =$ 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

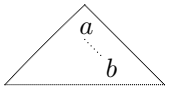
$$S'_0, S'_1, \ldots, S'_m \quad \text{mit } S_0 = S'_0.$$

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) > Level(b).

(b) Ist b Nachfolger von a in S_i, so gilt Level(a) > Level(b) .

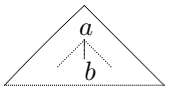
Beachte: Level bezieht sich *immer* auf S'_m, also auf die Union-Find-Datenstruktur, die entsteht, wenn $\sigma_1, ..., \sigma_m$ *ohne* Wegkompression ausgeführt würden.

Beweis:

(a) Der Teilbaum unter a ist zum Zeitpunkt i natürlich mindestens 1 tiefer als der unter b.

Ist a nicht die Wurzel, dann kann sich unter a später nichts mehr ändern und die Behauptung gilt.
Ist a die Wurzel, so kann der Baum unter a durch spätere Unions höchstens noch tiefer werden. Wieder gilt Level(a) > Level(b)

(b) Sei also in S_i die Situation:

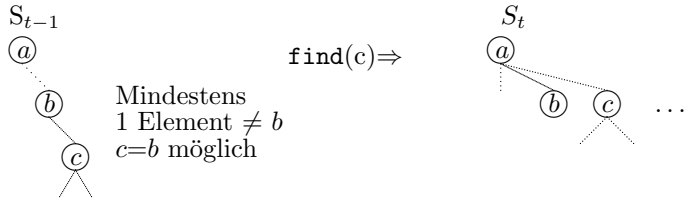


(b ist Sohn von a)

Wir zeigen die Behauptung für die direkten Nachfolger von a. Für tiefere Knoten folgt die Behauptung induktiv. Wir betrachten den Zeitpunkt $t \leq i$ zu dem die Kante $a \leftarrow b$ durch σ_t entstanden ist.

1. Fall: Die Kante $a \leftarrow b$ ist durch $\sigma_t = \texttt{union}(a, b)$ entstanden. Dann ist in S_{t-1} die Anzahl der Elemente unter $b \leq$ der Anzahl der Elemente unter a. Da S_{t-1} und S'_{t-1} dieselben Partitionen darstellen, gilt dies genauso in S'_{t-1}. Die Kante $a \leftarrow b$ entsteht also ebenfalls in S'_i. 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. Also ist a zum Zeitpunkt $t-1$ Wurzel des Baums.



Genauso ist a in S'_{t-1} Wurzel und b in jedem Fall unter a. Wegen (a) ist Level(a) > 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, ... , A_k, A_{k+1} mit:

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

Da die Tiefe der Bäume in S'_i durch $\log n$ beschränkt ist, gibt es für jeden Knoten a ein i, so daß $A_i \leq$ Level(a) $< A_{i+1}$. Da die Folge der A_i streng monoton wächst, gibt es zu jedem Knoten a auch nur maximal ein i mit $A_i \leq$ Level(a) $< A_{i+1}$. Das führt uns zum Begriff des Niveaus.

Definition 3.3.4. Für $0 \leq i \leq k$ ist

$$\text{Niveau}(a) = i \quad \Longleftrightarrow \quad A_i \leq \text{Level}(a) < A_{i+1}$$

Bemerkung: Ist Level(a) = 0, dann gilt Niveau(a) = 0. Ist Level(a) = $\lfloor \log_2 n \rfloor$, dann ergibt sich Niveau(a) = k.

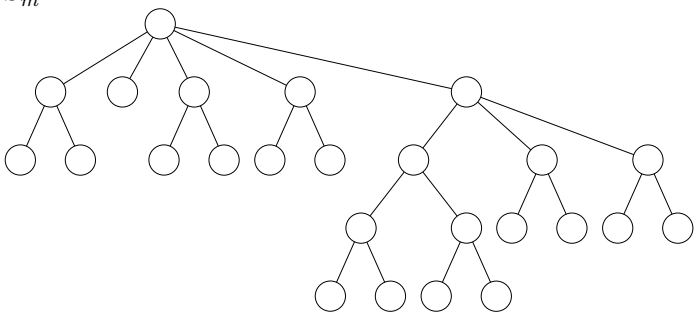
Anhand der A_i-Werte kann später die amortisierte Laufzeit berechnet werden. Die Wahl der A_i hat Einfluß auf die Laufzeitabschätzung.

Beispiel:

(a) $k = 0, \quad A_0 = 0, \quad A_1 = \lfloor \log_2 n \rfloor + 1$

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

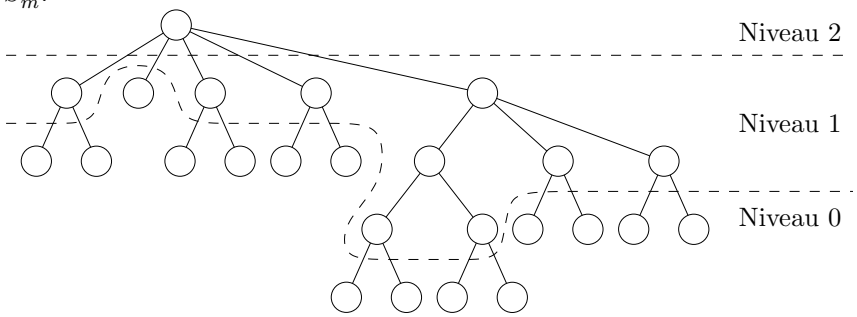
(c) S'_m:



$\lfloor \log_2 25 \rfloor = 4$

wähle $k = 2$ und $A_0 = 0$, $A_1 = 1$, $A_2 = 4$, $A_3 = 5$

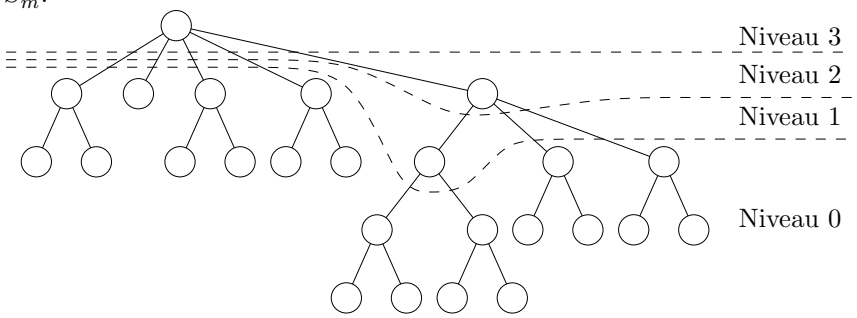
S'_m:



(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:



Wieviele Elemente existieren auf einem Niveau?

Lemma 3.3.5.

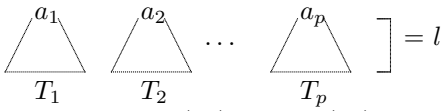
Es gilt:

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

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

Beweis:

(a) Es seien $a_1, a_2, \ldots, a_p$ alle Elemente vom Level l. Die Struktur S'_m enthält nun folgende (Teil-) Bäume:



Ferner gilt Tiefe(T_1) = Tiefe(T_2) = $\cdots$ = Tiefe(T_p) = l.
Aufgrund der gleichen Tiefe sowie der unterschiedlichen Wurzeln sind die Bäume T_i disjunkt. Wegen der Heuristik Union-By-Size enthält jeder Baum mindestens 2^l Elemente.
Also gilt:

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

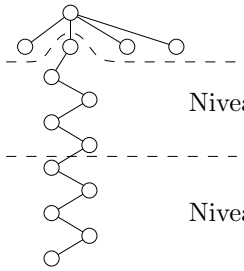
(b) Wir summieren über alle Level innerhalb des Niveaus i.

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

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

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

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:



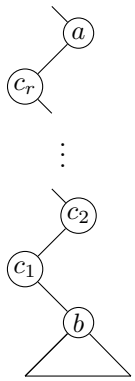
Niveau 1 = Level 5, ... , Level 9

Niveau 0 = Level 0, ... , Level 4

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

Beweis: Die Operationsfolge $\sigma = \sigma_1, \ \ldots \ , \sigma_m$ sei fest. Die Operation σ_i = `union`(a,b) benötigt Zeit O(1), während σ_i = `find`(b) die Zeit O(1 + Anzahl Kanten von b bis zur Wurzel a) benötigt.

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



Nach Lemma 3.3.3 gilt für die Level:

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

also auch:

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

Für jede Kante l auf dem Weg gilt eine der beiden Möglichkeiten:

(a) l verbindet 2 Knoten desselben Niveaus, oder

(b) l verbindet 2 Knoten mit verschiedenen Niveaus.

Für Kanten vom Typ (a) verwenden wir gespeicherte Zeit, falls der obere Knoten *keine* Wurzel ist. Wir nehmen die gespeicherte Zeit vom unteren Knoten der Kante.

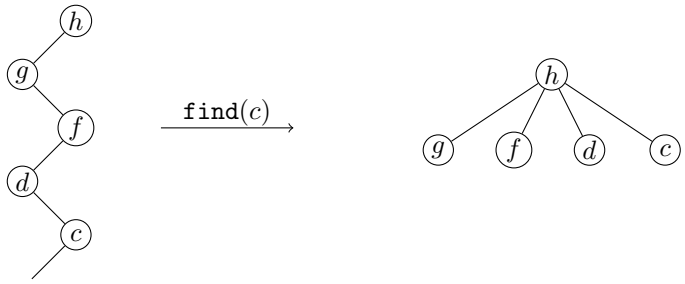
Für Kanten vom Typ (b) und die Kanten zur Wurzel benutzen wir keine gespeicherte Zeit. Diese Schritte zählen wir direkt beim find. Das sind maximal # Niveauübergänge+1 $= k + 1$ viele.

Daraus ergibt sich folgende Zusammensetzung:

$$\text{Zeit von } \sigma_1, \ \ldots \ , \sigma_m \ \leq \ \underbrace{O(\kappa)}_{\text{für } \texttt{union}} \ + \ \underbrace{O(\lambda \cdot (k+1))}_{\text{direkt bei } \texttt{find} \text{ gezählt}} \ + \text{gespeicherte Zeit} \quad (3.1)$$

Wieviel Zeit muß gespeichert sein?

Sei c ein beliebiger Knoten mit Niveau i. Jedes Mal, wenn bei c ein O(1) entnommen wird, liegt folgende Situation vor (weder c noch dessen Vater sind Wurzel):



Nach Lemma 3.3.3 ist Level(f) $\geq$ Level(d) + 1 (f existiert, da d keine Wurzel ist). Also gilt: Wenn bei c Zeit entnommen 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 neuen Vaters von $\geq A_{i+1}$, dann ist das Niveau des neuen Vaters von $\geq i+1$. Dann ist die Kante von c zu seinem Vater ein Niveauübergang. Das Benutzen dieser Kante wird bei zukünftigen `find`s direkt bei diesen gezählt.
Daraus folgt: Bei c wird maximal soviel Zeit entnommen, wie es Level-Übergänge im Niveau i gibt, also $A_{i+1} - A_i$. Es genügt für jeden der $\leq 2n/2^{A_i}$ Knoten im Niveau i die Zeit $A_{i+1} - A_{i-1}$ zurücklegen.
Damit reicht es, die Zeit

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

zu speichern. Wir erhalten das Lemma mit Ungleichung (3.1).
Wir versuchen jetzt, k und die A_i so zu wählen, daß die Abschätzung aus Lemma 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

$$O(\lambda \cdot (k+1) + \kappa + 2 \cdot n \cdot (A_1 - A_0 - 1)) = O(\lambda + \kappa + n \cdot \log n)\,.$$

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

Die Laufzeiten in (a) und (b) sind noch nicht optimal.
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 $A_0 = 0$, $A_1 = 1$, $A_2 = 2^{A_1} = 2$, $A_3 = 2^{A_2} = 4$, $A_4 = 2^{A_3} = 16$, $A_5 = 2^{A_4} = 2^{16}$ $\ldots$
Wegen $A_k \leq \lfloor \log_2 n \rfloor < A_{k+1}$ ist $k = \log^* n$.
Dann hat die Operationsfolge σ die Laufzeit

$$O\left(\lambda \cdot \log^* n + \kappa + \sum_{i=0}^{k} \frac{2n}{2^{A_i}} \cdot (A_{i+1} - A_i)\right),$$

wobei

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

Wir erhalten (wie im Satz 3.3.1 behauptet) die Laufzeit $O((\lambda + n) \cdot \log^* n + \kappa)$.

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 enthalten 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 Hashing). 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

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

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

`start` $= 1$

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:
$3 \to 2 \to 5 \to 7 \quad \Rightarrow \quad 3 \to \underbrace{5 \to 2}_{\text{Transposition}} \to 7$

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. (Vier Heuristiken)

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:

σ = `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:

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

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

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

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. Zu den Transpositonen, wo x nicht nach links geht, gehören auch die, an denen x unbeteiligt ist.

Satz 4.3.2 (Optimalität von MF)**.** Für jede Folge σ von Operationen auf einer anfangs leeren Liste gilt:

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

wobei A eine beliebige Heuristik 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:

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

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:

$$\{x, y\} \text{ stellt eine } \textit{Inversion} \text{ von } S,\ T \text{ dar}$$

$$\Longleftrightarrow$$

$$((x \text{ vor } y \text{ in } S) \text{ und } (x \text{ nach } y \text{ in } T)) \text{ oder } ((x \text{ nach } y \text{ in } S) \text{ und } (x \text{ vor } y \text{ in } T))$$

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.

Seien t_i die Kosten(σ_i) unter MF und t'_i die Kosten(σ_i) unter A.

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

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

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

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

amortisierte Kosten(σ) unter MF:

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

Bemerkung: Die amortisierten Gesamtkosten sind immer eine obere Schranke an die tatsächlichen Kosten dar, solange $\Phi_i \ge 0$ für alle i gilt.

Es genügt nun zu zeigen: Für alle i ist $a_i \le 2 \cdot t'_i$. Daraus folgt sofort

$$\sum t_i \le \sum a_i \le 2 \cdot \sum t'_i.$$

Für die Idee der Potentialfunktion kann man zunächst folgenden Spezialfall betrachten:

Sei $\sigma_i = \texttt{find}(x)$, $S_{i-1} = \ldots \to x$ und $S'_{i-1} = x \to \ldots$. Hier ist also t_i lang, während t'_i kurz ist. Während $\texttt{find}(x)$ wandert x in S_{i-1} an den Kopf der Liste und es verschwinden $t_i - 1$ Inversionen. Damit ist $a_i = t_i + \Phi_i - \Phi_{i-1} = t_i - (t_i - 1) = 1 \le 2 \cdot t'_i$.

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:

$$S_{i-1} = \underbrace{\cdots}_{k\text{-1 Elemente}} \rightarrow \underbrace{x}_{k\text{-te Stelle}} \rightarrow \cdots$$

$$S'_{i-1} = \underbrace{\cdots}_{j\text{-1 Elemente}} \rightarrow \underbrace{x}_{j\text{-te Stelle}} \rightarrow \cdots$$

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

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

$$\begin{aligned} \text{MF:} \quad & S_{i-1} \xrightarrow[a_{i,1}]{} S_i \xrightarrow[a_{i,2}]{} S_i \\ \text{A:} \quad & \underbrace{S'_{i-1}}_{\Phi_{i-1}} \xrightarrow[a_{i,1}]{} \underbrace{S'_{i-1}}_{\Phi'} \xrightarrow[a_{i,2}]{} \underbrace{S'_i}_{\Phi_i} \end{aligned}$$

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

Es sei $a_{i,1} = t_i + \Phi' - \Phi_{i-1}$. 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 Verschieben von x mittels der Heuristik MF (Laufzeit $a_{i,1}$) betrachtet:

$$\text{MF:} \quad S_{i-1} = \underbrace{\cdots y \cdots}_{k-1} \rightarrow x \rightarrow \cdots z \cdots \qquad S_i = x \rightarrow \cdots y \cdots z \cdots$$

$$\text{A:} \quad S'_{i-1} = \underbrace{\cdots z \cdots}_{j-1} \rightarrow x \rightarrow \cdots y \cdots \qquad S'_{i-1} = \underbrace{\cdots z \cdots}_{j-1} \rightarrow x \rightarrow \cdots y \cdots$$

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.

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

p... Anzahl verschwindender Inversionen beim Übergang

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.

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

r... Anzahl hinzukommender Inversionen beim Übergang: $r = k - 1 - p$.

Damit ist

$$a_{i,1} = t_i + r - p = k + r - p \leq j - 1 + k - p\,,$$

da $r \leq j-1$ gelten muß. Aus $r \leq j-1$ und $r = k-1-p$ folgt $k-p \leq j$, woraus sich

$$a_{i,1} \leq 2 \cdot j - 1$$

ergibt.

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

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

Da von der Zeit t'_i genau j Schritte darauf entfallen, x zu finden, können noch maximal $t'_i - j$ Inversionen dazukommen. Damit ist $a_{i,2} \leq t'_i - j$.

Insgesamt ist nun

$$a_i = a_{i,1} + a_{i,2} \leq (2j-1) + (t'_i - j) = t'_i + j - 1 \leq 2 \cdot t'_i\,, \quad \text{da } j \leq t'_i\,.$$

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

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

Es erfolgt wieder die Aufteilung in:

$$\begin{array}{rlllll} \text{MF:} & S_{i-1} & \longrightarrow & (S_{i-1} \to x) & \longrightarrow & S_i = (x \to S_{i-1}) \\ \text{A:} & \underbrace{S'_{i-1}}_{\Phi_{i-1}} & \longrightarrow & \underbrace{(S'_{i-1} \to x)}_{\Phi'} & \longrightarrow & \underbrace{S'_i}_{\Phi_i} \end{array}$$

k =Länge der Liste
$\Phi' = \Phi(S_{i-1} \to x, S'_{i-1} \to x)$
$a_{i,1} = t_i + \Phi' - \Phi_{i-1}$
$a_{i,2} = 0 + \Phi_i - \Phi'$
(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 $a_{i,2}$:

$$\text{MF:}\quad S_{i-1} = \underbrace{\cdots}_{S_{i-1}} \to x \qquad S_i = x \to \underbrace{\cdots}_{S_{i-1}}$$

$$\text{A:}\quad S'_{i-1} = \underbrace{\cdots}_{S'_{i-1}} \to x \qquad S'_i = \underbrace{\cdots}_{j-1} \to \underbrace{x}_{j\text{-te Stelle}} \to \cdots$$

j = Position von x in S'_i

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

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:

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

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

$$\begin{aligned}\text{Kosten}(\sigma) \text{ in MF} &= 1 + (n-1) + 2 + (n-2) + 3 + (n-3) + \ldots + (n-1) + 1 + n\\ &\le n\cdot(n-1)\\ &= O(n^2)\end{aligned}$$

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).

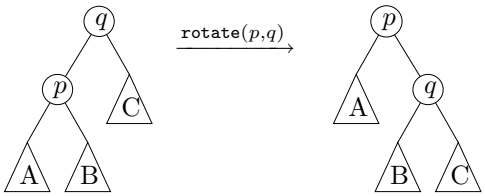
`splaying`(p)**:**

```
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)
```

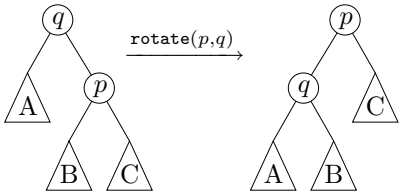
Fälle von `splaying`(p):

Sei $q = \texttt{vater}(p)$ und $r = \texttt{vater}(q)$.

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

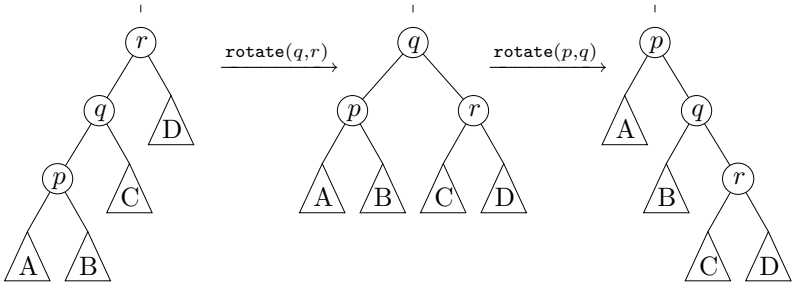


   (b) q ist Wurzel, p ist rechter Sohn von q

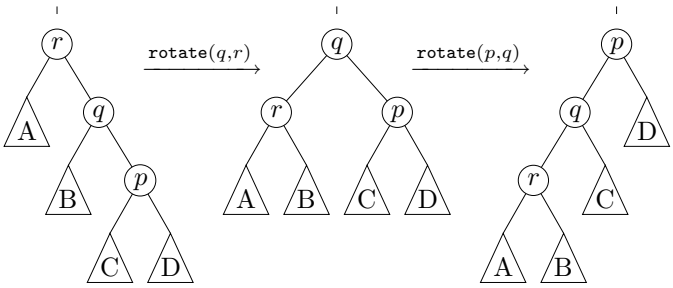


2. q ist nicht die Wurzel

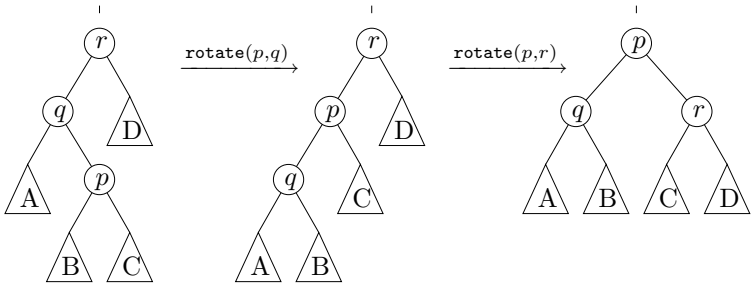
   (a) „zig-zig“ nach links



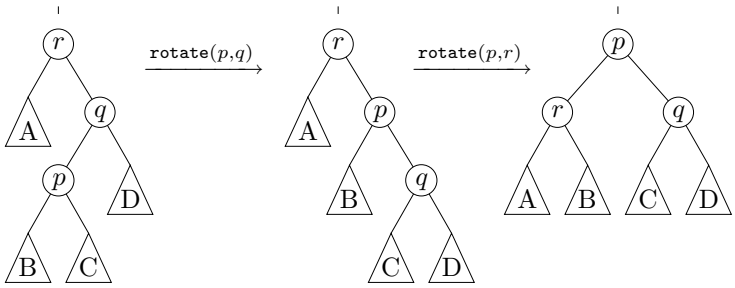
   (b) „zig-zig“ nach rechts



(a) „zig-zag“ links-rechts



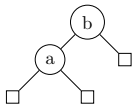
(b) „zig-zag“ rechts-links



5.2 Datenstruktur Splaybaum

Die Struktur ist ein einfacher binärer Suchbaum mit leeren Blättern. D. h. sind die eigentlichen Elemente des Baumes so angeordnet: 

, so hat der Baum in unserem Sinne noch drei weitere (leere) Blätter:



(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 Suche nach x geendet hat
while p nicht die Wurzel **do**
  `splaying(`p`)`

(b) `find`(x)

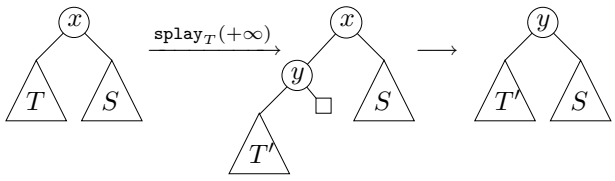
`splay(`x`)`
if x ist Wurzel **then**
  return x
else
  return „nicht vorhanden“

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

(c) `delete`(x)

```
splay(x)
if x ist Wurzel then        // ⇔ x ist im Baum enthalten
   T := linker Teilbaum
   S := rechter Teilbaum
   if T ist leer then
      S wird neuer Baum
   else
      splay_T(+∞)
      Hänge S unter die neue Wurzel von T.
```

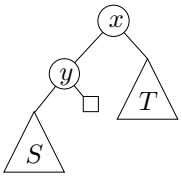
Dabei wird durch $\texttt{splay}_{\texttt{T}}(+\infty)$ der symmetrische Vorgänger y von x zum linken Sohn von x. (Da die Schlüssel eindeutig sind, hätte $\texttt{splay}_{\texttt{T}}(x)$ dieselbe Wirkung.) Der Knoten y hat keinen rechten Sohn, da y das größte Element im linken Teilbaum ist.



(d) `insert`(x)

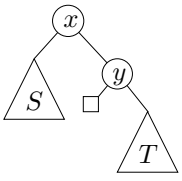
```
splay(x)
if x ist die Wurzel then
   return „x vorhanden“
else
   y := Schlüssel der Wurzel        // x ≠ y
   T := rechter Teilbaum
   S := linker Teilbaum
   if x > y then
```

Bilde den Baum:




```
   else        // x < y
```

Bilde den Baum:

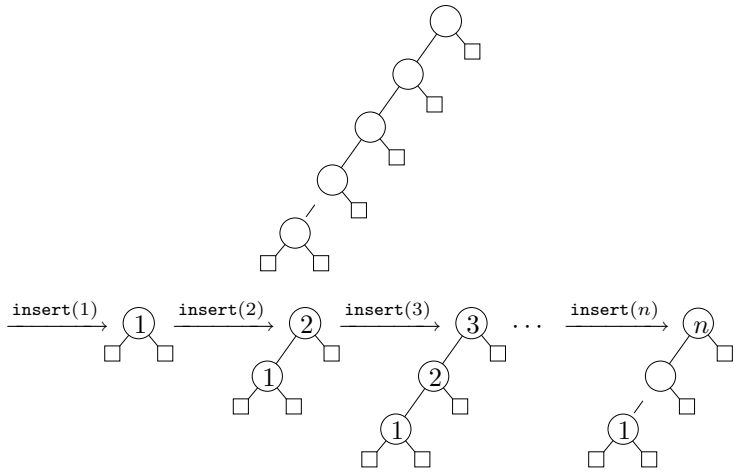


Für eine der obigen Operationen σ auf den Baum S ist

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

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

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



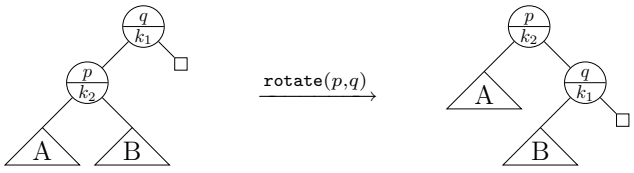
`find`(1) benötigt Zeit O(n), aber die vorherigen n Einfügungen nur O(1). Eine amortisierte Zeit von O($\log n$) ist also möglich.
Ziel: m Dictionary-Operationen auf einen anfangs leeren Baum sollen in Zeit O($m \cdot \log n$) abgearbeitet werden, wobei n (> 1) die Maximalzahl der Elemente im Baum ist.

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 $Iw_S(p) > 0$ (individual weight).

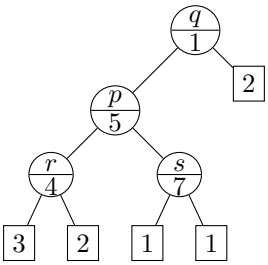
Beachte: Das Individualgewicht ist am Knoten fixiert. Es wird also mitrotiert:



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:

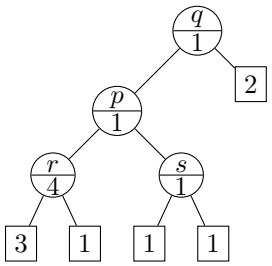


$$Tw_S(p) = 5 + 4 + 7 + 3 + 2 + 1 + 1 = 23$$

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:



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

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

$$\Phi(S) = \sum_{p \in S} R_S(p) = \sum_{p \in S} \lfloor \log_2 Tw_S(p) \rfloor.$$

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.

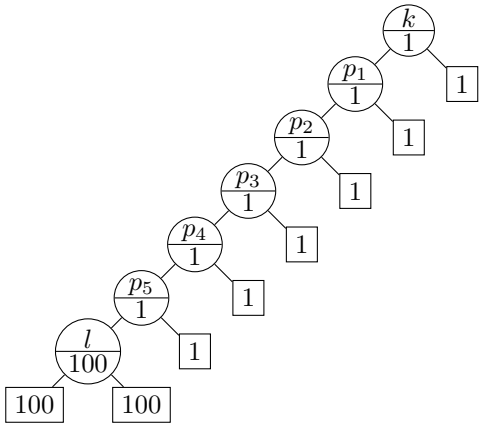
5.4 Schlüssellemma

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' . Dann gilt:

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

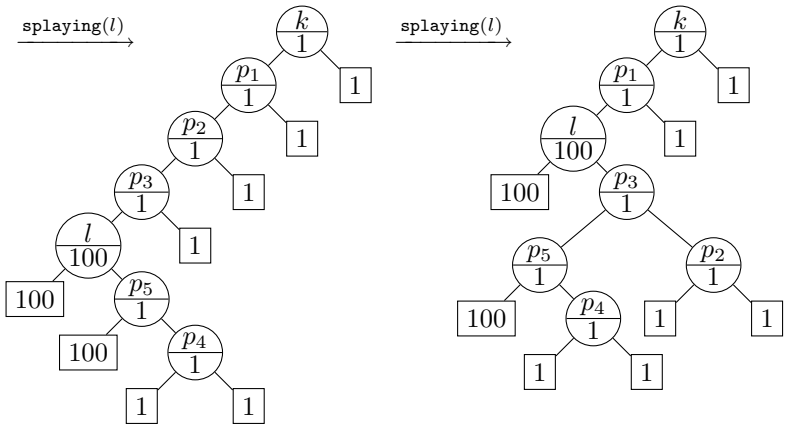
Beachte: $R_S(k) - R_S(l) \geq 0.$

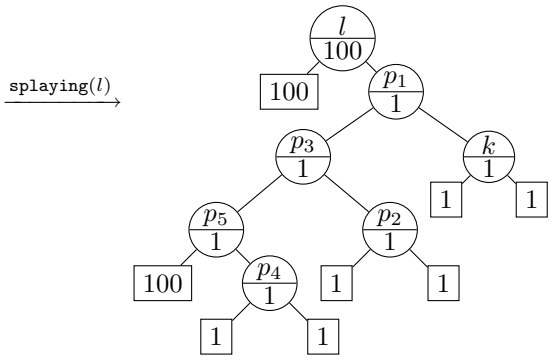
Beispiel: Im folgenden Beispiel sind die tatsächlichen Kosten hoch, die amortisierten Kosten von $3 \cdot (R_S(k) - R_S(l))$ jedoch klein.
S:



Wir haben $R_S(k) = \lfloor \log_2(300 + 12) \rfloor = 8$, $R_S(l) = \lfloor \log_2 300 \rfloor = 8$. Also ist $R_S(k) - R_S(l) = 0$.

$\texttt{splay}_S(l)$ führt zu:





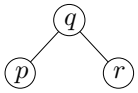
Damit ist $K_S(\texttt{splay}_S(x)) = 3$. Wie hoch sind die amortisierten Kosten?

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

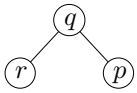
$$\begin{aligned}
\Phi(S') &= \underbrace{2 \cdot \lfloor \log_2 100 \rfloor + 6 \cdot \lfloor \log_2 1 \rfloor}_{\text{Blätter}} \\
&\quad + \underbrace{3 \cdot \lfloor \log_2 3 \rfloor + \lfloor \log_2 104 \rfloor + \lfloor \log_2 108 \rfloor + \lfloor \log_2 112 \rfloor + \lfloor \log_2 312 \rfloor}_{\text{alle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

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

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) $R_T(q) > R_T(p) \ \Leftrightarrow$
  Es gibt ein $n \geq 0$, so daß $Tw_T(p) < 2^n$ 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. Da l die neue Wurzel ist, ist $l = x$ oder l ist ein zu x nächster Schlüssel.
Falls $k = l$ ist, hat sich der Baum nicht geändert, also ist $\Phi(S') = \Phi(S)$. Wegen $K(\texttt{splay}_S(x)) = 1$ ist $a(\text{Splay}_S(x)) = 1 = 1 + 3 \cdot \underbrace{(R_S(k) - R_S(l))}_{=0,\text{ da } k=l}$.

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

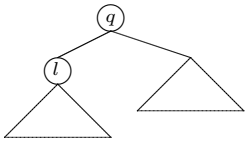
$$\begin{aligned} a_i &\leq 3 \cdot (R_{S_i}(l) - R_{S_{i-1}}(l)) \qquad \text{für alle } i \text{ mit } 1 \leq i \leq m-1 \\ a_m &\leq 1 + 3 \cdot (R_{S_m}(l) - R_{S_{m-1}}(l)), \end{aligned}$$

denn daraus folgt die Behauptung.

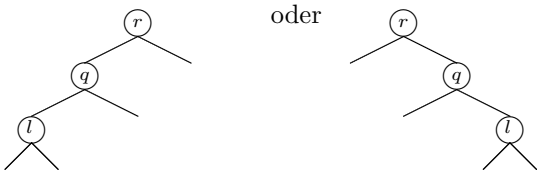
$$\begin{aligned} a(\texttt{splay}(x)) &= \sum_{i=1}^{m} a_i \\ &\leq 3 \cdot (R_{S_1}(l) - R_{S_0}(l)) + \ \cdots \ + 3 \cdot (R_{S_{m-1}}(l) - R_{S_{m-2}}(l)) \\ &\quad + 1 + 3 \cdot (R_{S_m}(l) - R_{S_{m-1}}(l)) \\ &= 1 + 3 \cdot (R_{S_m}(l) - R_{S_0}(l)) \\ &= 1 + 3 \cdot (R_{S'}(l) - R_S(l)) \\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

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:

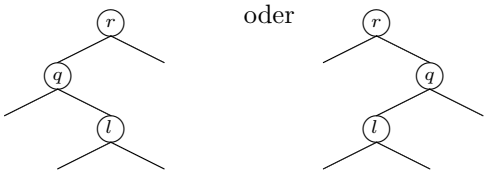
1. Fall:



2. Fall:



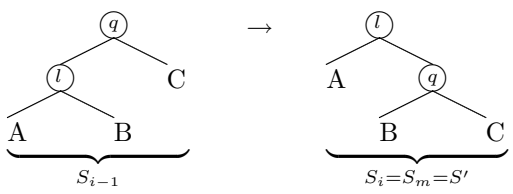
3. Fall:



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

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

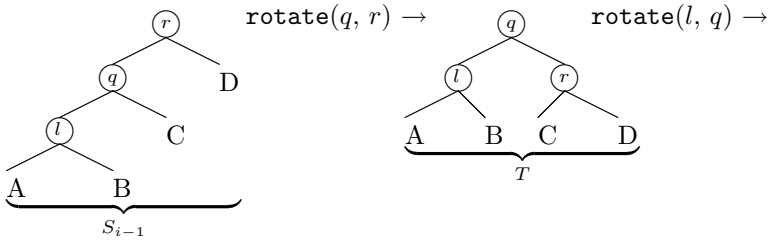
1. Fall: Hier ist $i = m$ (r nicht definiert). Was geschieht?

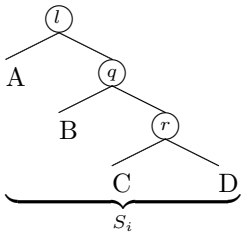


Es ist:
$R'_l = R_q$
$\Phi(S_{m-1}) = R_q + R_l +$ die Ränge in A B C
$\Phi(S_m) = R'_l + R'_q +$ die Ränge in A B C
Weiter folgt:

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

2. Fall:





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

$$\begin{aligned} a_i &= 1 + \Phi(S_i) - \Phi(S_{i-1}) \\ &= 1 + R'_l + R'_q + R'_r - R_l - R_q - R_r \\ &\overset{R'_l = R_r}{=} 1 + R'_q + R'_r - R_l - R_q \end{aligned} \tag{5.1}$$

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

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

Denn es ist nach Annahme $R'_l > R_l$, also $R'_l - R_l \geq 1$ da Ränge ganze Zahlen sind.

(b) $R'_l = R_r = R_l$ (Fall 2c: $R'_l < R_l$ kann nicht sein)
Zunächst gilt nach Eigenschaft 1 (i): $R_l \leq R_q \leq R_r = R'_l = R_l$, also ist $R_q = R'_l$ nach Annahme. Wegen $R'_q \leq R'_l$ ist $R'_q \leq R_q$. Wir vereinfachen (5.1) zu

$$a_i \leq 1 + R'_r - R_l.$$

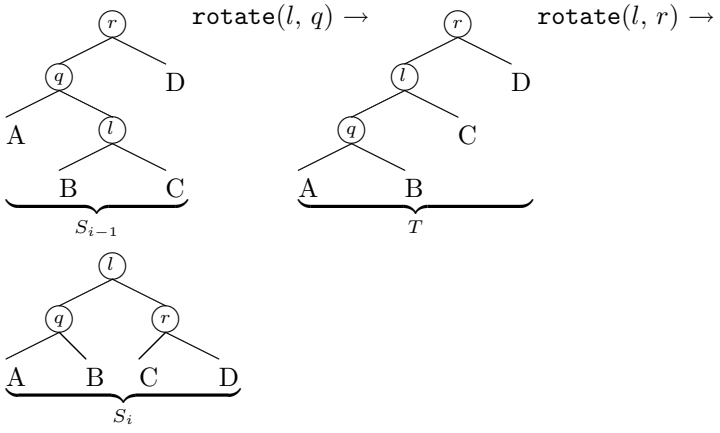
Angenommen, $R'_r \geq R_l$ würde gelten. Wenden wir Eigenschaft 2 im Baum T an, folgt $R_T(q) > R_l$. Wir haben bereits weiter oben festgestellt: $R_T(q) = R'_l$. Also wäre $R'_l > R_l$, was Annahme (b) widerspricht. Also gilt $R'_r < R_l$, bzw. $R'_r - R_l \leq -1$.

Es folgt:

$$a_i \leq 0 = 3 \cdot (R'_l - R_l).$$

Der Beweis für den Baum in andere Richtung erfolgt analog.

3. Fall:



Gleichung (5.1) gilt auch in diesem Fall.

(a) $R'_l = R_r > R_l$ Dann ist mit (5.1)

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

(b) $R'_l = R_r = R_l$
Wegen (5.1) und $R_r = R_l$ ist

$$a_i = 1 + (R'_q - R_q) + (R'_r - R_r)\,.$$

Man kann leicht sehen, daß sowohl $(R'_q - R_q)$ als auch $(R'_r - R_r) \leq 0$ sind. Darüberhinaus ist sogar einer der beiden Summanden < 0, also ≤ -1. Wir führen das Gegenteil ($(R'_q - R_q) = 0$ und $(R'_r - R_r) = 0$) zum Widerspruch. Wegen $R_r \geq R_q \geq R_l$ folgt $R_q = R_r$ also auch $R'_q = R'_r$. Eigenschaft 2 im Baum S_i gibt sofort $R'_l > R'_r = R_r$, was ein Widerspruch zur Annahme (b) ist.
Also ist $(R'_q - R_q) \leq -1$ oder $(R'_r - R_r) \leq -1$. Weil keiner der beiden Summanden positiv ist, folgt $(R'_q - R_q) + (R'_r - R_r) \leq -1$. Wir sehen

$$a_i = 1 + (R'_q - R_q) + (R'_r - R_r) \leq 1 - 1 = 0 = 3 \cdot (R'_l - R_l).$$

□

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:

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

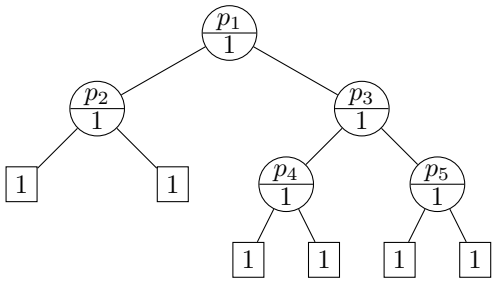
(b) Ist S_0 schon mit n Elementen ausgestattet, so ist:

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

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 $\leq n$ Elementen und jeden Knoten p von S, daß $Tw_S(p) \leq 2n + 1$ (n innere Knoten und $n + 1$ Blätter).

Beispiel mit $n = 5$:



Also 5 innere Knoten und 6 Blätter.

Deshalb gilt:

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

Nach Lemma 5.4.1 ist

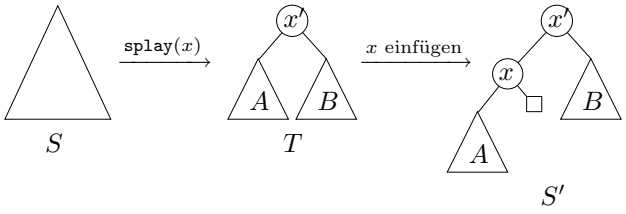
$$\begin{aligned} a(\texttt{splay}_S(x)) &\leq 1 + 3 \cdot (R_S(k) - \underbrace{R_S(l)}_{\geq 1}) \\ &\leq 1 + 3 \cdot (R_S(k) - 1) \\ &\leq 1 + 3 \cdot (\log(n) + 1) \\ &= 4 + 3 \cdot \log n. \end{aligned}$$

Also:

$$a(\texttt{find}_S(x)) \leq 4 + 3 \cdot \log(n)$$

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

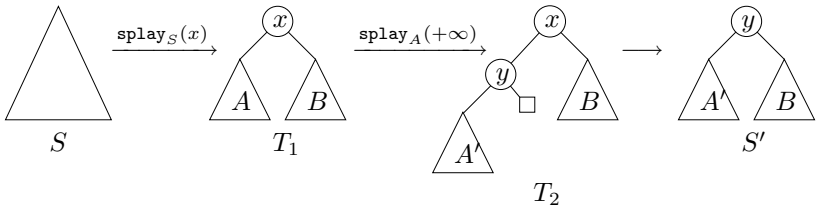
$\texttt{insert}_S(x)$



Also

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

$\texttt{delete}_S(x)$



Also

$$\begin{aligned} a(\texttt{delete}_S(x)) &= \text{Kosten}(\texttt{delete}_S(x)) + \Phi(S') - \Phi(S) \\ &= \text{Kosten}(\texttt{splay}_S(x)) + \Phi(T_1) - \Phi(S) \\ &\quad + \text{Kosten}(\texttt{splay}_A(+\infty)) + \Phi(T_2) - \Phi(T_1) \\ &\quad + \underbrace{\Phi(S') - \Phi(T_2)}_{\leq 0} \\ &\leq 4 + 3 \cdot \log n + 4 + 3 \cdot \log n \\ &= 8 + 6 \cdot \log n. \end{aligned}$$

Also insgesamt

$$\begin{aligned} \text{Kosten}(\sigma) &= \sum_{i=1}^{m} \text{Kosten}(\sigma_i) = \sum_{i=1}^{m} a(\sigma_i) \quad - \Phi(S_m) + \Phi(S_0) \\ &\leq \sum_{i=1}^{m} (8 + 6 \cdot \log n) - \Phi(S_m) + 0 \\ &\leq m \cdot (8 + 6 \cdot \log n) \\ &\leq 7m \cdot \log n \qquad \text{für } m, n \text{ groß genug.} \end{aligned}$$

(b) siehe Übung.

Satz 5.4.4. (Lernfähigkeit der Splay-Bäume, vgl. optimale Suchbäume)
Seien S, S' 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:

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

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