Prof. Andreas Goerdt
Frank Schädlich

TU Chemnitz
Sommersemester 2003

Theoretische Informatik III
2. Übung - Lösungsvorschläge

1. **Aufgabe**

   (a) Das Minimum des binomialen Heaps wird über den entsprechenden Zeiger in der Zeit $O(1)$ gefunden. Sei dieses Minimum die Wurzel eines binomialen Baums B_k.

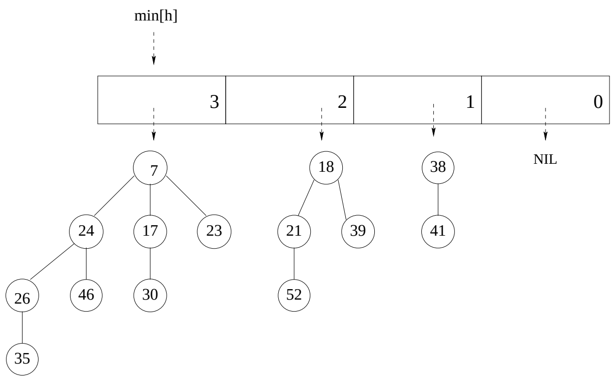


Abbildung 1:

Dann entfernt man diesen zugehörigen binomialen Baum B_k. (In unserer Aufgabe ist dieses der binomiale Baum B_3, wie man im Bild 1 erkennen kann.) Der binomiale Baum B_k $(k > 0)$ besteht wiederum aus den binomialen Bäumen $B_0, \cdots, B_{k-1}$, welche unter der Wurzel (hier der Knoten mit dem Wert 7) hängen. Diese „schmilzt“ man nun in den Heap ein.

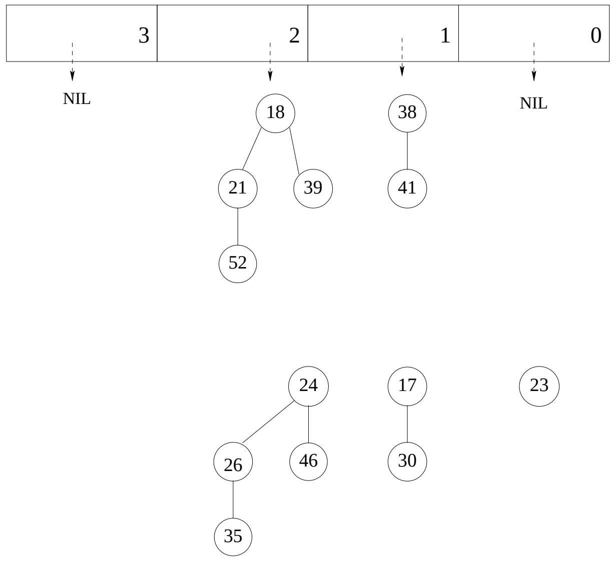


Abbildung 2:

Bild 2 zeigt die Situation vor dem Einschmelzen. Gleichzeitig mit dem Einschmelzen wird das neue Minimum bestimmt. Ein mögliches Resultat ist in Bild 3 dargestellt.

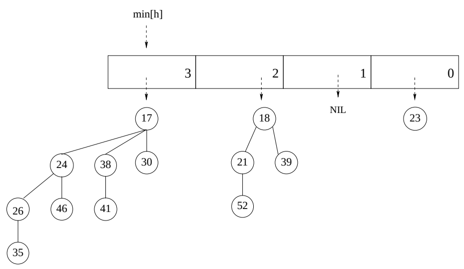


Abbildung 3:

Die möglichen Ergebnisse resultieren aus der Wahl des Einschmelzens der drei Bäume B_2.
(Ist das Minimum ein binomialer Baum B_0, so müssen natürlich keine Bäume mehr eingeschmolzen werden. Es ist dann nur noch die Bestimmung des neuen Minimums erforderlich.)

(b) Mit dem Einfügen eines neuen Elementes in den binomialen Heap wird dieses eingeschmolzen. Wir müssen also zuerst den binomialen Baum B_3, dann den Baum B_2 und letztendlich den Baum B_1 erzeugen. Dies kann z.B. durch das Einfügen von Elementen mit den Werten in der folgenden Reihenfolge erreicht werden:
26, 35, 24, 46, 17, 30, 7, 23,
21, 52, 18, 39,
38 und 41.

2. Aufgabe

Satz: Eine Folge $\sigma = \{\sigma_1, \sigma_2, .., \sigma_n\}$ von

- n_1-mal $Insert$,
- n_2-mal $DeleteMin$ und
- n_3-mal $Minimum$

in einer beliebigen Reihenfolge auf den leeren binomialen Heap ausgeführt und mit $n = n_1 + n_2 + n_3$, besitzt einen Zeitaufwand $O(n_1 + n_2 \cdot \log n_1 + n_3) = O(n \log n)$.

Beweis: Das Potential des Heaps H sei $\Phi(H) = \sharp\{\text{Bäume im Heap } H\}$. Für den anfänglich leeren Heap H_0 gilt $\Phi(H_0) = 0$. Sei T die Summe der tatsächlichen Zeiten t_i der Operationen σ_i, also $T = \sum_{i=1}^{n} t_i$. Die amortisierte Zeit a_i einer Operation σ_i, die den Heap H_{i-1} in den Heap H_i überführt ($H_{i-1} \xrightarrow{\sigma_i} H_i$), beträgt

$$a_i = t_i - \Phi(H_{i-1}) + \Phi(H_i).$$

Für die Summe der amortisierten Zeiten gilt:

$$\begin{aligned}\sum_{j=1}^{n} a_j &= \sum_{j=1}^{n} t_j - \sum_{j=1}^{n} \Phi(H_{j-1}) + \sum_{j=1}^{n} \Phi(H_j) \\ &= T - \Phi(H_0) - \sum_{j=1}^{n-1} \Phi(H_j) + \sum_{j=1}^{n-1} \Phi(H_j) + \Phi(H_n) \\ &= T + \Phi(H_n).\end{aligned}$$

Somit gilt für die Gesamtausführungszeit T:

$$T = \sum_{j=1}^{n} a_j - \Phi(H_n).$$

Bemerkung: Im allgemeinen wählt man das Potential Φ einer Datenstruktur A so, daß

$$\begin{aligned}\Phi(A_0) &= 0 \\ \Phi(A_i) &\geq 0\end{aligned}$$

gilt. Damit wird die amortisierte Laufzeit der Folge σ von Operationen zur oberen Schranke der tatsächlichen Laufzeit.

Um die Laufzeit der Folge von Operationen abschätzen zu können, betrachten wir nun die amortisierten Zeiten der drei möglichen Funktionen:

$Insert$: Das Erzeugen des Baumes mit einem Knoten benötigt einen Zeitaufwand von 1. Das Verschmelzen zweier binomialer Bäume benötigt ebenfalls einen Aufwand von 1. Der Verschmelzungsvorgang (Einschmelzen von B_0 und evtl. weitere Verschmelzungen) ist mit der binären Addition von 1 vergleichbar. Hatte der Heap vor dem Einschmelzen $x = \Phi(H_i)$ Bäume, so sind es danach noch $y = \Phi(H_{i+1})$. Der Zeitaufwand für das Einschmelzen ergibt sich also aus der Differenz von x und y. Der amortisierte Aufwand beträgt somit: $a_{Insert_i} = \underbrace{\Phi(H_i) - \Phi(H_{i+1})}_{\text{Verschmelzungen}} + \underbrace{1}_{\text{Einfügen}} \underbrace{- \Phi(H_i) + \Phi(H_{i+1})}_{\text{Potentialdifferenz}} = 1.$

$DeleteMin$: Das Finden und Herausschneiden des Minimums geschieht in Zeit $O(1)$. Das Minimum sei die Wurzel eines binomialen Baums B_z mit $0 \leq z \leq \log_2 n_1 \leq \log_2 n$. Somit sind z Binomialbäume wieder einzufügen und evtl. einzuschmelzen. Das Einfügen bzw. Einschmelzen eines Baumes geschieht mit konstantem Aufwand. Nach dem Herausschneiden des Minimums und vor dem Einschmelzen haben wir offensichtlich $\Phi(H_i) - 1 + z$ Bäume. Mit jedem Einschmelzen „verschwindet" ein binomialer Baum. Es gibt also $(\Phi(H_i) - 1 + z) - \Phi(H_{i+1})$ Einschmelzungen. Nun müssen nur noch die $\Phi(H_{i+1})$ Bäume nach dem neuen Minimum durchsucht werden. Es ergibt sich also für die amortisierte Laufzeit eines DeleteMin():

$$\begin{aligned}
a_{DeleteMin_i} &= t_i - \Phi(H_i) + \Phi(H_{i+1}) \\
&= t(\text{Minimum löschen}) + t(\text{Teilbäume einfügen}) + \\
&\quad\ t(\text{Einschmelzen}) + t(\text{neues Minimum suchen}) - \\
&\quad\ \Phi(H_i) + \Phi(H_{i+1}) \\
&= 1 + z + (\Phi(H_i) - 1 + z) - \Phi(H_{i+1}) + \Phi(H_{i+1}) - \\
&\quad\ \Phi(H_i) + \Phi(H_{i+1}) \\
&= 2z + \Phi(H_{i+1}).
\end{aligned}$$

$$\begin{aligned}
\text{Da wir maximal } n_1 &\leq n \text{ Elemente im Heap speichern, gibt es} \\
\text{maximal } \Phi(H_{i+1}) &\leq \log_2 n_1 \leq \log_2 n \text{ binomiale Bäume im Heap.} \\
\text{Also gilt } a_{DeleteMin_i} &\leq 3 \cdot \log_2 n_1 = O(\log n).
\end{aligned}$$

(Jedes *Insert* „speichert" also Zeit in dem neu eingefügten Baum. Diese Zeit wird durch ein Einschmelzen wieder aufgebraucht.)

Minimum: Die Funktion Minimum ändert nichts am Potential, es gilt also $\Phi(H_i) = \Phi(H_{i+1})$. Der Zugriff auf das Minimum erfolgt über einen Zeiger in Zeit $O(1)$. Der amortisierte Aufwand beträgt somit $a_{Minimum_i} = O(1)$.

Für den Gesamtaufwand T von σ ergibt sich nun:

$$\begin{aligned}
T &= \sum_{i=1}^{n} t_i \\
&= \sum_{i=1}^{n} a_i - \Phi(H_n) \\
&\leq \sum_{i=1}^{n} a_i \\
&= \sum_{i=1}^{n_1} a_{Insert_i} + \sum_{i=1}^{n_2} a_{DeleteMin_i} + \sum_{i=1}^{n_3} a_{Minimum_i} \\
&= O(n_1 + n_2 \cdot \log n_1 + n_3) \\
&= O(n \log n).
\end{aligned}$$

3. **Aufgabe**

Sei $S_i = \underbrace{1 + 1 + \cdots + 1}_{i\times} = i$ die Summe nach der Addition der ersten i Einsen. Es ist also $S_{i+1} = S_i + 1$. Nach der Addition von n Einsen erhalten wir $S = S_n$. Wir wissen, daß S_i mit k, $k \leq \lceil \log_3 i \rceil + 1$, Ziffern (0-2) dargestellt werden kann. Sei $S_i = (s_{k-1}, \cdots, s_0)$ die Ternärdarstellung von $S_i = i$. Zur Vereinfachung nehmen wir führende Nullen an, so daß wir immer $k = \lceil \log_3 i \rceil + 1$ Ziffern haben. Für die Addition einer 1 zählen wir eine Zeiteinheit. Addieren wir zu einer 2 eine 1 ergibt sich natürlich 0 und wir müssen im Übertrag eine weitere 1 addieren, also brauchen wir 2 Zeiteinheiten. In einer ersten Abschätzung sehen wir, daß maximal $\lceil \log_3 n \rceil$ Zweien auftreten können und damit eine Addition die Kosten $\Theta(\log n)$ haben kann. Daraus ergibt sich als obere Schranke eine Laufzeitkomplexität von $O(n \log n)$ für die Addition der n Einsen im Ternärsystem.

Da diese maximal $\lceil \log_3 n \rceil$ Zweien offensichtlich nur einmal auftreten, ist dieses eine relativ grobe Abschätzung. Betrachten wir nun die Addition einer 1 einmal im Detail. Die Summe S_i ende O.B.d.A. mit z vielen Zweien, $0 \leq z \leq \lceil \log_3 i \rceil + 1$, also: $S_i = (s_{k-1}, s_{k-2}, \cdots, s_z, \underbrace{2, \cdots, 2}_{z\times})$. Die Ziffer s_z sei damit die erste Ziffer ungleich Zwei (gesehen von rechts), d.h. die Addition des Übertrages endet hier. Offensichtlich gibt es z Überträge. Wir wählen nun die Anzahl der Zweien als Potentialfunktion: $\Phi(S_i) = \sharp\{\text{Zweien in der Ternärdarstellung von } S_i\}$. Da $\Phi(S_0) = 0$ und $\Phi(S_i) \geq 0$ gilt, ist dies eine prinzipiell geeignete Potentialfunktion. Für die amortisierten Kosten a_i der Addition einer Eins zu S_i erhalten wir $a_i = \underbrace{1}_{\text{Addition einer 1}} + \underbrace{z}_{z \text{ Überträge}} \underbrace{-\Phi(S_i) + \Phi(S_{i+1})}_{\text{Potentialdifferenz}}$. Ist die Ziffer s_z eine 0, so erhalten wir keine neue 2, damit ist in diesem Fall $\Phi(S_i) - \Phi(S_{i+1}) = z$. Ist dagegen $s_z = 1$, so erzeugen wir eine neue Zwei. Damit ist $\Phi(S_i) - \Phi(S_{i+1}) = z - 1$ und es ergibt sich $a_i = \begin{cases} 1 \text{ falls } s_z = 0 \\ 2 \text{ falls } s_z = 1. \end{cases}$ Es gilt also immer $a_i \leq 2$. Somit hat die Addition von n Einsen im Ternärsystem eine Zeitkomplexität von $O(n)$.

An dieser Aufgabe sieht man sehr schön die Vorteile einer Analyse unter Verwendung von amortisierten Zeiten. Zum Vergleich gibt es hier noch einmal die Musterlösung der analogen Übungsaufgabe aus der Vorlesung „Theoretische Informatik 1“:

Haben wir Ternärzählen der Länge n, so bestimmt die Anzahl der konsekutive Zweien am Ende die Anzahl der Takte des von Neumann Addierers, denn diese Zweien werden alle durch Nullen ersetzt und die vorhergehende Ziffer (0 oder 1) um 1 erhöht.

Hat ein Ternärzähler am Ende genau k konsekutive Zweien, so ist die Anzahl der Takte des von Neumann Addierers $(1 + k)$, $k = 0, \ldots, n$. Es gibt $\left(3^{n-(k+1)} \cdot 2\right)$ Ternärzahlen mit genau k konsekutive Zweien am Ende für $k = 0, \ldots, n-1$, da für die vorhergehende Position nur die Einträge 0 und 1 möglich sind, die verbleibenden $n-(k+1)$ Positionen jedoch beliebig mit 0, 1 oder 2 besetzt werden können.

Für $k = n$ gibt es jedoch nur eine derartige Ternärzahl, nämlich $(2, 2, \ldots, 2)$.

Also ist im `average case` die Anzahl der Takte eines von Neumann Addierers:

$$\begin{aligned} T_{VN}^{ac}(n) &= \frac{\sum\limits_{k=0}^{n-1} \left(3^{n-(k+1)} \cdot 2 \cdot (k+1)\right) + (n+1)}{3^n} \\ &= \frac{n+1}{3^n} + 2 \cdot \sum_{k=1}^{n} k \cdot 3^{-k} \end{aligned}$$

Mit $\sum\limits_{k=1}^{n} k \cdot x^k = \frac{n \cdot x^{n+2} - (n+1) \cdot x^{n+1} + x}{(x-1)^2}$ (siehe Vorlesung) folgt für $x := \frac{1}{3}$:

$$
\begin{aligned}
T_{VN}^{ac}(n) &= \frac{n+1}{3^n} + 2 \cdot \sum_{k=1}^{n} k \cdot 3^{-k} \\
&= \frac{n+1}{3^n} + 2 \cdot \frac{n \cdot \left(\frac{1}{3}\right)^{n+2} - (n+1) \cdot \left(\frac{1}{3}\right)^{n+1} + \frac{1}{3}}{\left(\frac{2}{3}\right)^2} \\
&= \frac{n+1}{3^n} + \frac{9}{2} \cdot \left(n \cdot \left(\frac{1}{3}\right)^{n+2} - (n+1) \cdot \left(\frac{1}{3}\right)^{n+1} + \frac{1}{3} \right) \\
&= \frac{n+1}{3^n} + \frac{1}{2} \cdot \left(n \cdot \left(\frac{1}{3}\right)^{n} - 3 \cdot (n+1) \cdot \left(\frac{1}{3}\right)^{n} + 3 \right)
\end{aligned}
$$

$$
\begin{aligned}
&= \frac{n}{3^n} + \frac{1}{3^n} - \frac{n}{3^n} - \frac{3}{2} \cdot \left(\frac{1}{3}\right)^{n} + \frac{3}{2} \\
&= \frac{3}{2} - \frac{1}{2} \cdot 3^{-n}.
\end{aligned}
$$

4. **Aufgabe**

(a) Im Fibonacci-Heap werden einzelne Bäume nur dann zusammengefaßt, wenn die Funktion `DeleteMin()` aktiviert wurde. Den zur Liste entarteten Baum erzeugen wir rekursiv. Zunächst erzeugen wir einen Baum bei dem die Wurzel nur einen Sohn s hat. Diesen können wir dann mit einer schon gegebenen Liste L verschmelzen. Aus dem so entstandenen Baum T kann mittels der Funktion `DecreaseKey()` s abgeschnitten und so wieder eine Liste erzeugt werden. Den erforderlichen Hilfsbaum gewinnt man etwa durch das Einfügen von drei Elementen in den Heap, welche das neue Minimum, z.B. $m = 0$, enthalten. Löscht man m, setzt der geforderte Schmelzvorgang ein. Bild 4 zeigt schematisch die Vorgehensweise. Es ergibt

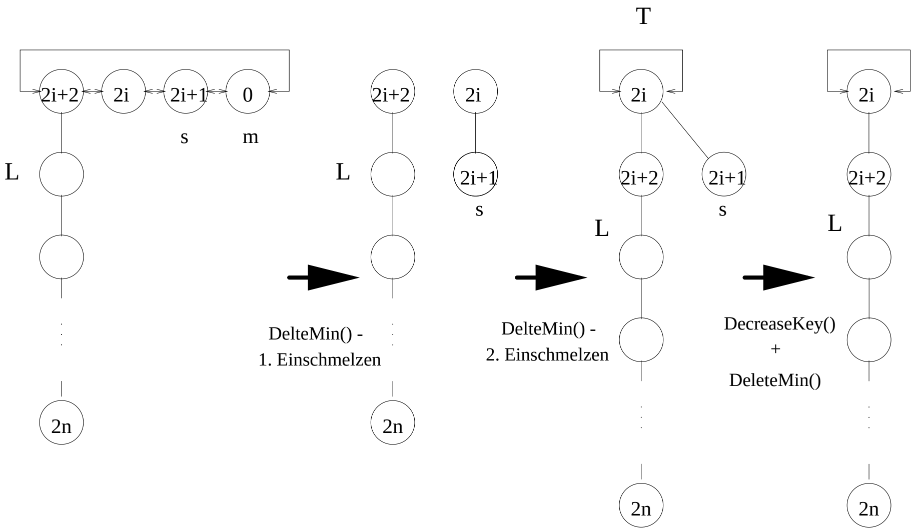


Abbildung 4:

sich also folgende Folge von Operationen:

```
1 for i:=n DOWNTO 1 DO

2      Insert(2i+1,2i+1)
3      Insert(2i,2i)
4      Insert(0,0)

5      DeleteMin()

6      DecreaseKey(2i+1,0)
7      DeleteMin()
```

Hinweise:

- Die Vorgehensweise ist noch nicht optimal. Das in Zeile 7 gelöschte Minimum könnte beispielsweise im nächsten Schleifendurchlauf eines der drei Inserts aus den Zeilen 2-4 ersetzen. Optimieren Sie den Algorithmus entsprechend.
- Untersuchen Sie, ob bzw. wie der Algorithmus geändert werden muß, damit alle Elemente der Liste (evtl. ohne die Wurzel) markiert sind.

(b) Nachfolgend werden die drei Funktionen im Detail betrachtet:

- `DeleteMin(h)`: Im `worst case` ist der Heap in einer Form organisiert, in der alle n Knoten listenartig aneinandergereiht sind, d.h. keiner der Bäume hat eine Tiefe > 1. Das ist zum Beispiel nach dem Einfügen von n Knoten in einen leeren Heap der Fall.
  Wird das Minimum gelöscht, müssen alle Knoten auf der Suche nach dem neuen kleinsten Wert durchforscht werden. Das dauert Zeit $\Omega(n)$, wobei `DeleteMin()` nebenher sogar noch den Heap reorganisiert.
- `Cut(i,h)`: Der schlimmste Fall tritt ein, wenn der Heap eine Struktur wie in Teilaufgabe (a) besitzt und alle Knoten bis auf die Wurzel und den am tiefsten liegenden Knoten bereits markiert sind. Ist letzterer der herauszuschneidene Knoten i, so setzt sich `Cut(i,h)` bis zur Wurzel fort und bearbeitet insgesamt $n-1$ Elemente jeweils in Zeit $\Omega(1)$. Dann ist die Gesamtlaufzeit gleich $\Omega(n)$.
- `DecreaseKey(i,j,h)`: Da diese Funktion `Cut(i,h)` aufruft, wenn der neue Wert j von i die Heap-Eigenschaft verletzt und `Cut(i,h)` eine Laufzeit von $\Omega(n)$ haben kann, gilt eine Zeitschranke von mindestens $\Omega(n)$ im `worst case` auch für `DecreaseKey(i,j,h)`.

5. **Aufgabe**

Sei

$$\Phi(h) = 1 \cdot \text{Anzahl Bäume in } h + 2 \cdot \text{Anzahl markierte Knoten in } h$$

das Potential von h. Betrachten mit $\sigma = \sigma_1, \ldots, \sigma_m$ eine Folge von Operationen, die ihrerseits eine Folge von Heaps $h_0 = \emptyset, h_1, \ldots, h_m$ erzeugt, wobei $\Phi_i = \Phi(h_i)$ sei. Wenn $t_i = \text{Zeit}(\sigma_i)$ ist, folgt für die amortisierte Zeit $a_i = t_i + \Delta\Phi = t_i + \Phi_i - \Phi_{i-1}$ und es gilt:

$$\sum_{i=1}^{m} a_i = \sum_{i=1}^{m} (t_i + \Phi_i - \Phi_{i-1}) = \Phi_m + \sum_{i=1}^{m} t_i \geq \sum_{i=1}^{m} t_i.$$

Es wird nun mit diesem Potential und einer Fallunterscheidung gezeigt, daß die behaupteten Laufzeiten erreicht werden:

- $\sigma_i =$ `Min(h)`
  Mit dem direkten Zeiger auf das Minimum beträgt die Laufzeit $O(1)$, zumal keine Potentialänderung stattfindet.
- $\sigma_i =$ `Insert(i,j,h)`
  Pro eingefügtem Knoten wird $\Delta\Phi = 1$ zum Potential hinzugefügt, um die korrekte Zählung von Φ nach obiger Forderung zu gewährleisten. Da 1 eine additive Konstante ist, bleibt die Laufzeit von `Insert(i,j,h)` $\Theta(1)$.
- $\sigma_i =$ `DeleteMin(h)`
  Betrachten die Ausführung dieser Operation im Detail:
  1 Das Minimum des Heaps wird in $\Theta(1)$ gefunden und der zugehörige Baum aus dem Heap entfernt
  2 Die maximal $\log n$ vielen Kinder des Baumes werden mit einer Zeiteinheit versehen und in das Feld A eingeschmolzen. Die Laufzeit beträgt hier $O(\log n)$.
  3 Das Feld A wird weiter aufgebaut, indem jeder Baum des Heaps eingeschmolzen wird. Hierbei wird eine Einheit vom Potential abgezogen. Dies verbraucht (im amortisierten Sinne) keine Zeit.
  3 Minimum suchen in $O(\log n)$. Der logarithmische Aufwand ergibt sich aus der Konstruktion des Fibonacci-Heaps.
  4 Liste aufbauen in $(2+1) \cdot O(\log n) = O(\log n)$, wobei der erste Teil des Potentials wiederhergestellt wird.

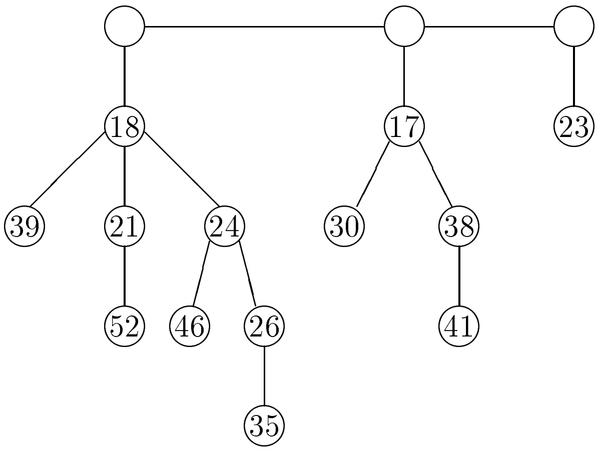
  Insgesamt braucht `DeleteMin(h)` eine Zeit von $O(\log n)$, wie behauptet.
- $\sigma_i =$ `Cut(i,h)`
  Betrachten die Ausführung dieser Operation im Detail:
  1 Wenn Knoten i (anfangs) noch nicht markiert ist, wird er in $O(1)$ herausgeschnitten und in die Liste eingeschmolzen. Beachte, daß dieser Fall während der Rekursion nur einmal auftreten kann, denn im weiteren sind die Knoten, wenn sie herausgeschnitten werden, bereits markiert. In diesem Fall benutzen wir eine Einheit des Potentials, um den Knoten herauszuschneiden. Die restliche Einheit verbleibt ihm, da er jetzt Wurzel eines neuen Teilbaumes ist.
  2 Ist der Vater des Knotens i noch nicht markiert, so tun wir das und zählen zwei Einheiten zu seinem Potential hinzu. Die Zeit dafür ist $O(2+1) = O(1)$. Beachte: Diese Zeit wird nur einmal gebraucht, denn nach dem Markieren, oder wenn die Wurzel erreicht ist, wird die Rekursion abgebrochen! War der Vater schon markiert, gehe zu Schritt 1.

  `Cut(i,h)` dauert im amortisierten Sinne nur $O(1)$!
- $\sigma_i =$ `DecreaseKey(i,j,h)`
  Verletzt i durch seinen neuen Wert j die Heap-Eigenschaft, so wird `Cut(i,h)` aufgerufen, das in $O(1)$ läuft, was dann auch für `DecreaseKey(i,j,h)` der Fall ist.

Damit ist die Behauptung für alle Operationen des Fibonacci-Heaps gezeigt. Ein ähnlicher Weg für den Nachweis der amortisierten Zeiten ist das direkte Verwenden der Definition $a_i = t_i + \Phi_i - \Phi_{i-1}$, woraus sich die Laufzeit nach kurzer Überlegung zu den drei Größen auf der rechten Seite unmittelbar ableiten läßt.

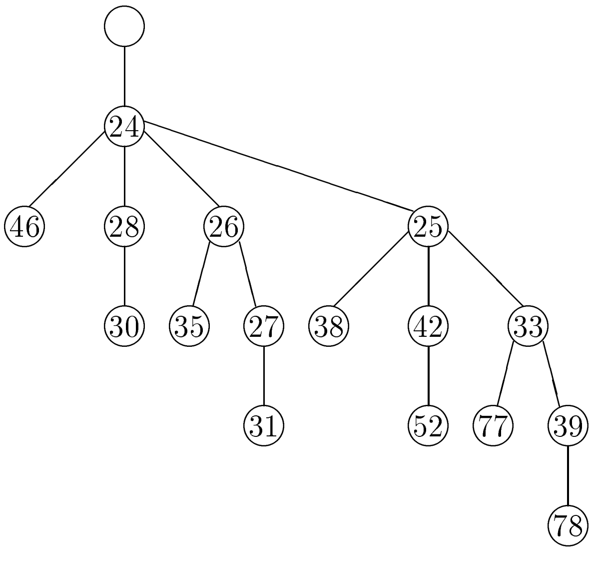
6. **Aufgabe**

(a) Nach dem Entfernen des Minimums mit `DeleteMin(h)` besitzt der Fibonacci-Heap folgendes Aussehen:

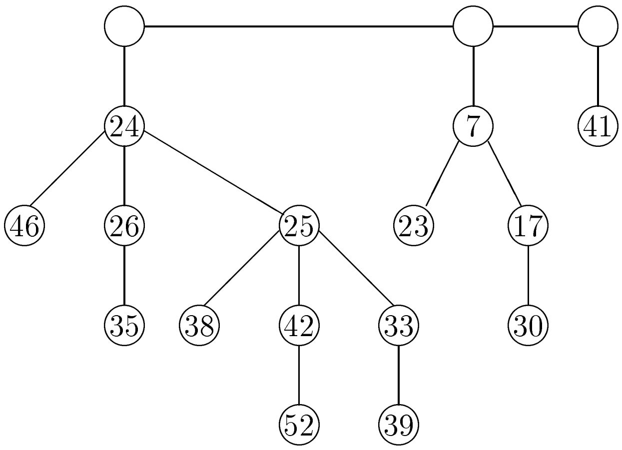


Neues Minimum ist der Knoten mit dem Wert 17.

(b) Der vorliegende Heap kann einschließlich der korrekten Markierungen wie folgt erstellt werden: Zunächst wird ein Binomialbaum folgenden Aussehens aufgebaut, indem die betreffenden Knoten einer passenden Reihenfolge (etwa von „links" nach „rechts") mit `Insert` eingefügt werden und zwischendurch immer wieder `CompressHeap` ausgeführt wird:



Nun operiert man wie folgt auf dem Heap: `DecreaseKey(28,17)` hängt einen Teilbaum heraus, `DecreaseKey(27,7)` und `DecreaseKey(31,23)` erzeugen einen zweiten und sorgen dafür, daß Knoten 26 markiert wird. Danach wird der Knoten 77 auf 0 erniedrigt und mit `DeleteMin` gelöscht. Das führt einerseits zum markieren von Knoten 33, andererseits zur Kompression des Heaps durch Zusammenhängen der oben erzeugten neuen Bäume. Selbige Operationsfolge wird auch auf Knoten 78 angewendet, um Knoten 39 zu markieren, wobei die Reorganisation diesmal überflüssig ist. Nun wird noch ein Knoten mit dem Wert 41 eingefügt. Es ergibt sich folgendes Bild:



Nun werden `DecreaseKey(25,0)` (Teilbaum heraushängen), `Decrease-Key(42,21)` und `DecreaseKey(33,18)` (richtige Werte einstellen) durchgeführt und anschließend löscht man das Minimum mittels `DeleteMin`. Damit werden drei neue Bäume freigesetzt, die durch die Reorganisation, wenn sie in geeigneter Weise durchgeführt wird (keine Einschränkung der Allgemeinheit, ggf. Reihenfolge der Kinder im Ausgangsbaum anpassen: 38 als rechtestes Kind von 25), zu dem gewünschten Ergebnis verschmolzen werden.