Lösungen zu „Fundamente der Informatik – Ablaufmodellierung, Algorithmen und Datenstrukturen“

Im Lehrbuch „Fundamente der Informatik – Ablaufmodellierung, Algorithmen und Datenstrukturen“ von Peter Hubwieser und Gerd Aiglstorfer (erschienen im Oldenbourg Verlag München, 2004, ISBN 3-486-27572-0) schließt jedes Kapitel mit einigen Aufgaben.

Die folgenden Lösungsvorschläge wurden von Gerd Aiglstorfer erstellt. Weitere Informationen und Hilfestellungen zum Lehrbuch finden Sie unter http://www.aigl.de.

Inhalt

1 Lösungsvorschläge zu „Bäume“

1.1 Lösungsvorschlag Aufgabe 13.1

Ein AVL-Baum ist nach Definition ausgeglichen, wenn für alle (internen) Knoten v des Baumes gilt:

$$|\text{Höhe linker Teilbaum von } v - \text{Höhe rechter Teilbaum von } v| \leq 1.$$

Es wäre nun möglich, zur Feststellung der AVL-Ausgeglichenheit jedes Mal die Höhen der beiden Teilbäume zu berechnen. Dies erscheint in Anbetracht des Rechenaufwandes jedoch wenig praktikabel. Als weitere Möglichkeit bietet sich an, in jedem Knoten die absolute Höhe des Baumes zu speichern, mit der Folge, dass bei jeder Änderung im Baum die Höhe in den entsprechenden Knoten angepasst werden muss (diese Lösung verfolgen wir jedoch nicht weiter).

Die folgenden Ausführungen dienen zur vertiefenden Information und müssen nicht Teil der Lösung zu dieser Aufgabe sein.

Um den Aufwand zur Bestimmung der AVL-Ausgeglichenheit zu minimieren, wird an jedem internen Knoten die Differenz zwischen linkem und rechtem Teilbaum gespeichert. Diese Differenz nennen wir *Balancefaktor balance* (an einem Knoten v sei der Wert durch *balance(v)* gegeben). Die folgenden AVL-Bäume sind mit diesem Faktor versehen (für die Berechnung des Balancefaktors wird die tatsächliche Höhe nicht benötigt – genaueres weiter unten):

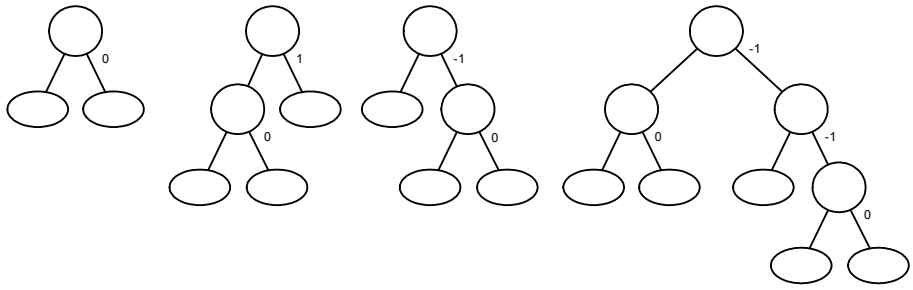

Abb. 1.1 *Balancefaktor*

Ein binärer Suchbaum ist AVL-ausgeglichen, wenn die Balancefaktoren im gesamten Baum nur die Werte –1, 0 oder 1 annehmen. Die Werte sind für Bäume geringer Höhe schnell und ohne bestimmtes Verfahren zu berechnen. Bei vielen Werten in einem Baum ist dies jedoch nicht mehr möglich.

Der Balancefaktor ändert sich beim Einfügen, Löschen und Ausführen von Rotationen und Doppelrotationen. Wir betrachten Einfüge- und Lösch-Operationen, da Rotationen auf diese folgen.

Gegeben sei der folgende AVL-Baum (Achtung: in den folgenden Grafiken sind die leeren Blätter der Knoten von AVL-Bäumen nicht angegeben):

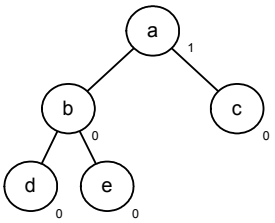

Abb. 1.2 *AVL-Baum (ohne Blätter)*

Wir fügen nun – jeweils vom aktuellen Baum ausgehend – an Knoten *c* ein rechtes Kind *f*, ein linkes Kind *g* und an Knoten *e* ein linkes Kind *h* ein:

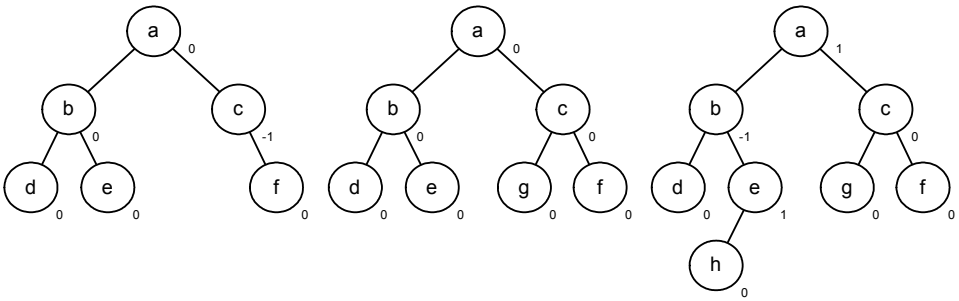

Abb. 1.3 *AVL-Baum: Balancefaktor nach Einfüge-Operation*

Dabei ergeben sich Änderungen in den Balancefaktoren:

- Beim Einfügen von *f* wächst die Höhe dieses Teilbaumes um 1, wodurch der Balancefaktor an Knoten *c* von 0 auf –1 wechselt (als direkte Konsequenz aus der Differenz von linkem und rechtem Teilbaum). Die Änderung wirkt sich aber nicht nur an *c*, sondern auch am Vater von *c* aus, dessen rechter Teilbaum um 1 wächst. Der Balancefaktor von *a* wechselt von 1 auf 0 und es sind keine Anpassungen mehr erforderlich (da *a* die Wurzel des Baumes ist).

- Nach dem Einfügen von *g* haben beide Teilbäume von *c* wieder die gleiche Höhe. *balance* von *c* wechselt von –1 auf 0. Auf die Gesamthöhe des Baumes hat dies jedoch keine Auswirkungen, wodurch am Vater von *c* keine Änderungen nötig werden.
- Durch das Einfügen von *h* wechselt *balance* an Knoten *e* von 0 auf 1, da der linke Teilbaum in der Höhe um 1 wächst. Die Höhe des rechten Teilbaumes von *b* (der Vaterknoten von *e*) wächst ebenfalls um 1: *balance(b)* wechselt hier von 0 auf –1. Dadurch vergrößert sich die Höhe des linken Teilbaumes der Wurzel *a* (*balance(a)* wechselt von 0 nach 1).

Wir verallgemeinern dieses Verfahren: Nach dem Einfügen muss zuerst der Knoten *v* betrachtet werden, an dem sich an einem der beiden Kinder auf erster Ebene etwas geändert hat. Der Balancefaktor von *v* muss in jedem Fall aktualisiert werden:

Tab. 1.1 *Änderung des Balancefaktors (Einfügen)*

Knoten v	Änderung des Balancefaktors
Einfügen linkes Kind	balance(v) + 1
Einfügen rechtes Kind	balance(v) – 1

balance(v) war vor Einfügen des neuen Knoten –1, 0 oder 1 (es muss ein AVL-Baum vorgelegen haben). Das weitere Vorgehen hängt nun vom neuen Wert *balance(v)* ab (oder anders: es hängt vom Verlauf des Wertes ab):

Tab. 1.2 *Vorgehen nach Änderung des Balancefaktors (Einfügen)*

Änderung balance(v) von x nach y	weiteres Vorgehen	Begründung
$0 \rightarrow -1, 1$	Betrachte Vater von v (falls vorhanden).	Der (Teil-) Baum t mit Wurzel v ist um 1 gewachsen. Dies hat Auswirkungen auf die Ausgeglichenheit des Baumes, in dem t Teilbaum ist (außer v war die Wurzel des gesamten Baumes). Darum muss der Balancefaktor im Vater von v angepasst werden. Dies geschieht mit demselben Verfahren (beginnend in der folgenden Tabelle), abhängig davon, ob v linkes oder rechtes Kind ist. Danach wird wieder nach dem Vorgehen dieser Tabelle entschieden (rekursiv).
$-1, 1 \rightarrow 0$	fertig	Die Gesamthöhe des (Teil-) Baumes mit Wurzel v hat sich nicht geändert. Die Höhe des linken oder rechten Teilbaumes hat sich lediglich der Höhe des anderen angeglichen. Der Vater von v muss nicht betrachtet werden und das Verfahren endet.
$-1 \rightarrow -2$	Führe Einfach- oder Doppelrotation aus.	Der (Teil-) Baum ist nicht mehr AVL-ausgeglichen.
$1 \rightarrow 2$	Führe Einfach- oder Doppelrotation aus.	Der (Teil-) Baum ist nicht mehr AVL-ausgeglichen.

Tab. 1.3 *Änderung des Balancefaktors im Vater von v (Einfügen)*

Vater von v	Änderung des Balancefaktors
Einfügen im linken Teilbaum	balance(Vater von v) + 1
Einfügen im rechten Teilbaum	balance(Vater von v) – 1

Aus dieser Beschreibung geht hervor, dass nach dem Einfügen unter Umständen der Balancefaktor an allen Knoten auf dem Pfad vom Knoten, an dem eingefügt wurde, bis zur Wurzel des Baumes zu aktualisieren ist. Es kann aber auch nur ein Teil des Pfades sein, genauso wie Rotationen auf Teilbäumen oder dem gesamten Baum nötig werden können. Darüber hinaus wird klar, dass es keiner Kenntnis der absoluten Höhe der Teilbäume zur Sicherstellung der AVL-Ausgeglichenheit bedarf.

Im Falle der Lösch-Operation verhält es sich ähnlich. Zunächst ein Beispiel (wir betrachten hier nur Lösch-Operationen auf äußeren Knoten):

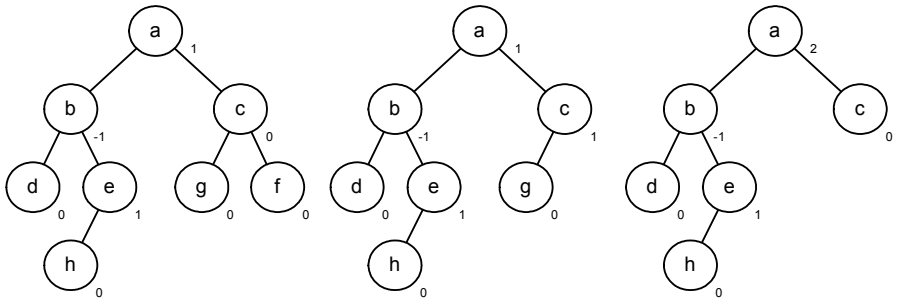

Abb. 1.4 *AVL-Baum: Balancefaktor nach Lösch-Operation*

Die Änderungen der Balancefaktoren ergeben sich wie folgt:

- Beim Löschen von *f* wird *balance(c)* um 1 erhöht. Der Wechsel von 0 nach 1 hat jedoch keine Auswirkungen auf *balance(a)*, da sich die Gesamthöhe des rechten Teilbaumes nicht ändert.
- Das Löschen von *g* rebalanciert den Knoten *c* (*balance(c)* wechselt von 1 nach 0), es verringert jedoch die Höhe des Teilbaumes, wodurch *balance(a)* um 1 erhöht werden muss.

Offensichtlich gilt, dass das Verhalten bei der Lösch-Operation genau umgekehrt zum Vorgehen beim Einfügen ist. Nach dem Löschen gilt für den Balancefaktor des Knoten *v*, der unmittelbar von der Änderung betroffen war:

Tab. 1.4 *Änderung des Balancefaktors (Löschen)*

Knoten v	Änderung des Balancefaktors
Löschen linkes Kind	balance(v) – 1
Löschen rechtes Kind	balance(v) + 1

Wiederum hat *balance(v)* aufgrund der AVL-Ausgeglichenheit vor dem Löschen einen der Werte –1, 0 oder 1. Für das weitere Vorgehen aufgrund des Wertverlaufes von *balance(v)* gilt:

Tab. 1.5 *Vorgehen nach Änderung des Balancefaktors (Löschen)*

Änderung balance(v) von x nach y	weiteres Vorgehen	Begründung
$0 \rightarrow -1, 1$	fertig	Die Gesamthöhe des (Teil-) Baumes mit Wurzel v hat sich nicht geändert. Die Höhen des linken und rechten Teilbaumes waren vor dem Löschen identisch. Der Vater von v muss nicht betrachtet werden und das Verfahren endet.
$-1, 1 \rightarrow 0$	Betrachte Vater von v (falls vorhanden).	Der (Teil-) Baum t mit Wurzel v ist um 1 geschrumpft. Dies hat Auswirkungen auf die Ausgeglichenheit des Baumes, in dem t Teilbaum ist (außer v war die Wurzel des gesamten Baumes). Darum muss der Balancefaktor im Vater von v angepasst werden. Dies geschieht mit demselben Verfahren (beginnend in der folgenden Tabelle), abhängig davon, ob v linkes oder rechtes Kind ist. Danach wird wieder nach dem Vorgehen dieser Tabelle entschieden (rekursiv).
$-1 \rightarrow -2$	Führe Einfach- oder Doppelrotation aus.	Der (Teil-) Baum ist nicht mehr AVL-ausgeglichen.
$1 \rightarrow 2$	Führe Einfach- oder Doppelrotation aus.	Der (Teil-) Baum ist nicht mehr AVL-ausgeglichen.

Tab. 1.6 *Änderung des Balancefaktors im Vater von v (Löschen)*

Vater von v	Änderung des Balancefaktors
Löschen im linken Teilbaum	balance(Vater von v) – 1
Löschen im rechten Teilbaum	balance(Vater von v) + 1

Die Untersuchungen zur Veränderung der Balancefaktoren bei Rotationen bzw. die Identifikation der Art der Rotation anhand der Verteilung der Balancefaktoren seien nun dem Leser überlassen (dabei wäre auch noch zu untersuchen, wie sich Rotationen auf Teilbäumen auf deren Vaterknoten auswirken).

1.2 Lösungsvorschlag Aufgabe 13.2

Der zu erzeugende AVL-Baum mit Wurzel 16 (die leeren Blätter sind jeweils nicht angegeben):

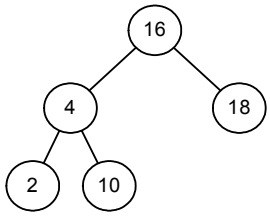

Abb. 1.5 *AVL-Baum*

Der Leser verdeutliche sich, dass dies der einzig korrekte AVL-Baum mit Wurzel 16 und diesen Schlüsseln ist.

Teilaufgabe a)
Nach dem Einfügen der 1 (als linker Teilbaum von Knoten 2 – nach der Bedingung für binäre Suchbäume ist dies die einzig mögliche Stelle), wird an der Wurzel des Baumes eine Rotation nach rechts erforderlich.

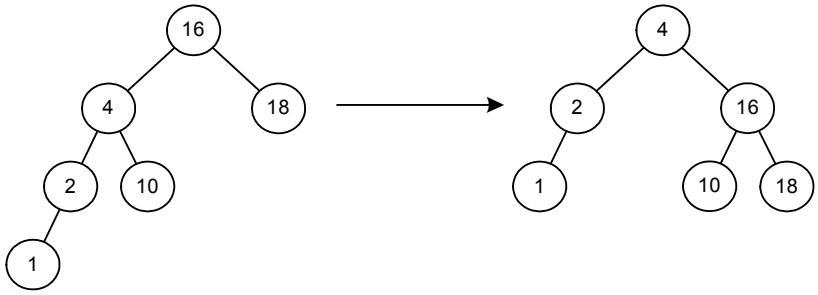

Abb. 1.6 *Rotation nach rechts*

Teilaufgabe b)
Das Einfügen der 7 hat einen validen AVL-Baum zur Folge. Nach dem Hinzufügen der 6 ist die AVL-Ausgeglichenheit im linken Teilbaum des Knotens 16 verletzt, d.h. die 10 muss mit einer Doppelrotation nach rechts-links in die Wurzel transportiert werden.

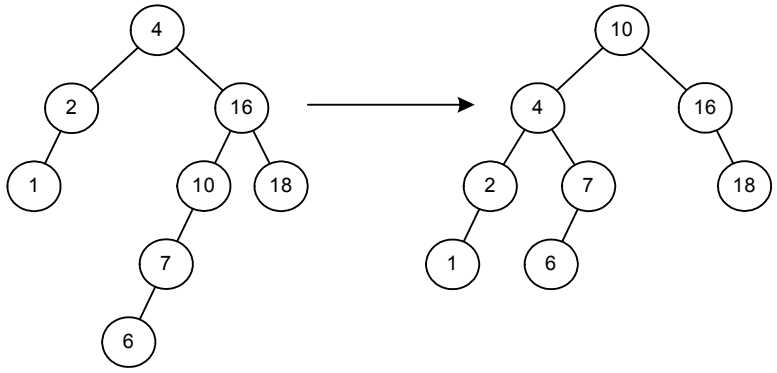

Abb. 1.7 *Doppelrotation nach rechts-links*

Es existiert noch eine weitere Möglichkeit (rotiere nur den Teilbaum mit der Wurzel 10 einmal nach rechts):

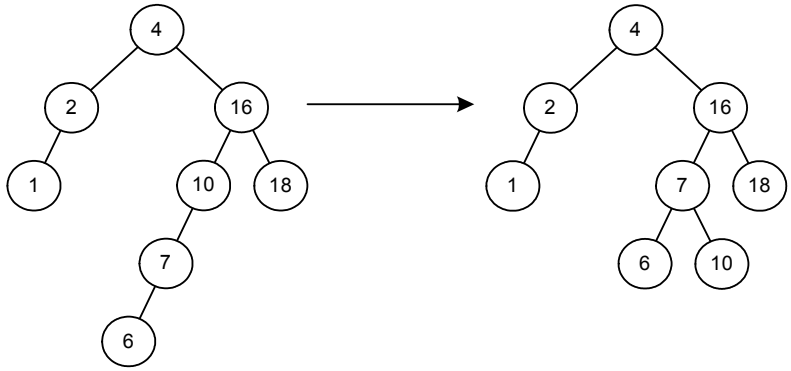

Abb. 1.8 *Rotation eines Teilbaumes nach rechts*

Wir betrachten AVL-Bäume in dieser Aufgabe ohne die Erkenntnis aus dem Lösungsvorschlag von Aufgabe 13.1. Hätten wir einen Balancefaktor für jeden Knoten eingeführt, so wäre nach Einfügen der 6 *balance(10) = 2*. Und somit würde sofort die Rotation des Teilbaumes ausgeführt. Je nachdem wie stark der Baum gefüllt ist, bestehen beim bloßen Betrachten oft mehrere Möglichkeiten zum Rotieren (wie in dieser Aufgabe). Ein implementierter Algorithmus (welcher aus dem Lösungsvorschlag von Aufgabe 13.1 hergeleitet werden kann) hat jedoch keine „globale“ Sicht auf den Baum und muss daher „lokal“ optimieren, wenn sich die Notwendigkeit ergibt.

1.3 Lösungsvorschlag Aufgabe 13.3

Teilaufgabe a)
Es sind $N = 8 = (1000)_2$ Elemente einzufügen. Es wird ein B_3 benötigt. Dieser könnte die folgende Schlüsselverteilung aufweisen (dies ist nur eine von mehreren Möglichkeiten – wir betrachten nicht, wie sie entstanden ist):

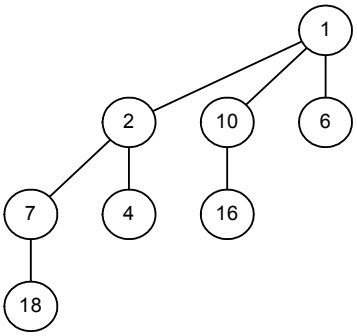

Abb. 1.9 *Binomial Queue*

Teilaufgabe b)
Nach dem Löschen von 18 sind noch $N = 7 = (111)_2$ Elemente in der Binomial Queue. Das Ergebnis wäre dann (mittels B_2, B_1 und B_0):

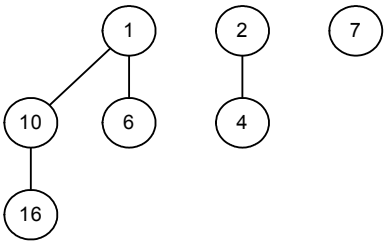

Abb. 1.10 *Binomial Queue nach dem Löschen von 18*

Teilaufgabe c)
Der Lösch-Vorgang kann nicht so einfach realisiert werden, wie es Teilaufgabe b) vermuten lässt. Es ist nicht immer der Fall, dass nur ein Binomialbaum in der Binomial Queue vorhan-

den ist. Das zu löschende Element könnte außerdem auch im Inneren eines Binomialbaumes zu finden sein.

Da jedoch jeder Binomialbaum komplett mit Schlüsseln gefüllt ist, können bei geschickter Realisierung der Lösch-Operation nach dem Entfernen des Elements aus dem B_n die Binomialbäume B_{n-1}, B_{n-2}, ..., B_1, B_0 entstehen (wenn es anders umgesetzt wird, so hat dies schlechtere Laufzeiten der Operation zur Folge). Die Binomial Queue wird dann durch die Operation *merge* wieder an die Bedingungen angepasst.

Wir nehmen für den Algorithmus an, dass die Kanten zu den Kindern eines Knotens v von links nach rechts mit 1 bis n nummeriert sind. Dabei ist n der jeweils Grad des B_n, dessen Wurzel v ist (zur Erinnerung: jeder Teilbaum eines Binomialbaumes ist aufgrund der induktiven Definition wiederum ein Binomialbaum).

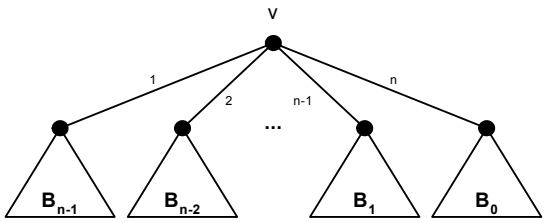

Abb. 1.11 *Kantennummerierter Binomialbaum B_n*

Wenn nun beispielsweise das Kind von v an Kante 2 entfernt wird (heiße es w), so liegt kein Binomialbaum mehr vor (dem B_n fehlt der B_{n-2}). Alle Kinder von w sind Binomialbäume, und zwar sind dies nach Definition: B_{n-3}, B_{n-4}, ..., B_0. Vom Baum mit Wurzel v wird nun noch die Kante 1 entfernt, wodurch ein B_{n-1} (der Baum an Kante 1) und B_{n-2} (der Baum mit Wurzel v ohne die Teilbäume an Kante 1 und 2) entsteht. Falls v nun Wurzel des ursprünglichen Baumes war, so ist die Lösch-Operation beendet. Andernfalls muss v von seinem Vater und dieser ebenfalls wie eben von den linken Kindern getrennt werden – außer v war an Kante 1 (und so fort, bis die Wurzel erreicht ist). Wenn nicht so am Vater weiter verfahren wird, so liegen keine Binomialbäume vor. Der Leser kann sich dies durch Löschen der 4 im Baum aus Teilaufgabe a) verdeutlichen.

Der informelle Algorithmus lautet somit (die Wurzel aller Binomialbäume werden in einer Wurzelliste verwaltet):

```
procedure delete (nat x):
begin
suche das Element x an Knoten v;
entferne alle Kanten zu den Kindern von v (falls vorhanden);
setze die Kinder mit ihren Unterbäumen in die Wurzelliste;
lösche den Knoten v;
solange v einen Vater u hatte, wiederhole:
```

```
      gehe zu u über die Kante i;
      lösche die Kante i;
      setze alle Teilbäume von u an den Kanten 1, …, i-1 in
      die Wurzelliste und entferne diese Kanten;
      setze u mit Kindern in die Wurzelliste;
      setze v := u
ende wiederhole
führe die Operation merge aus
endproc
```

1.4 Lösungsvorschlag Aufgabe 13.4

Nach 8-maligem *insert* erhalten wir den folgenden Fibonacci-Heap:

Abb. 1.12 *Fibonacci-Heap*

Darauf wird nun *deleteMin* ausgeführt (Löschen der 1 und solange *merge* ausführen, bis keine zwei Bäume mit identischem Wurzel-Rang mehr vorhanden sind):

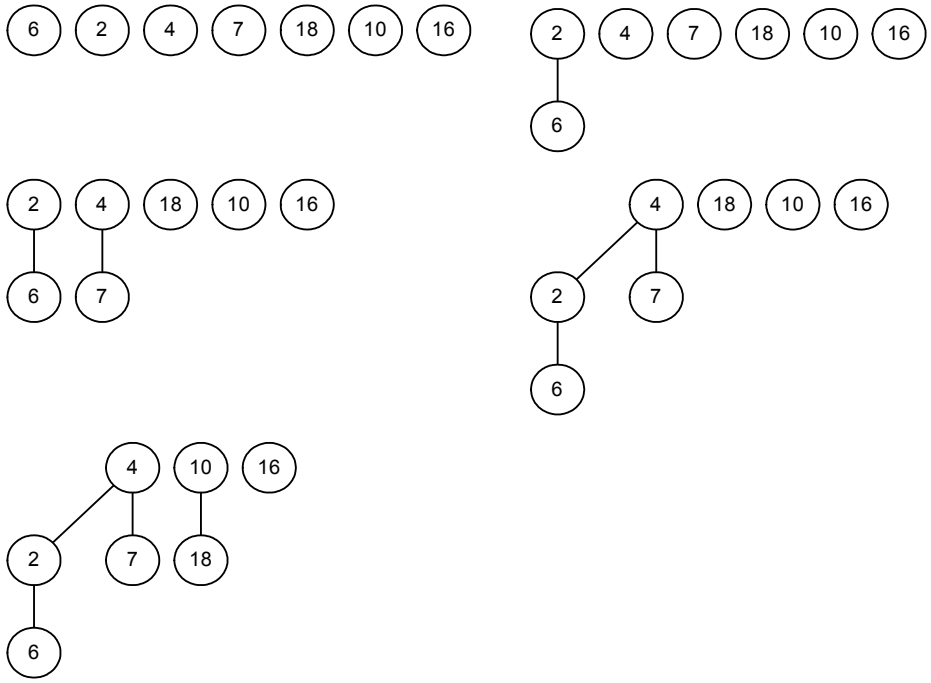

Abb. 1.13 *Fibonacci-Heap: deleteMin*

1.5 Lösungsvorschlag Aufgabe 13.5

Ein möglicher (2, 4)-Baum mit den gegebenen Schlüsseln:

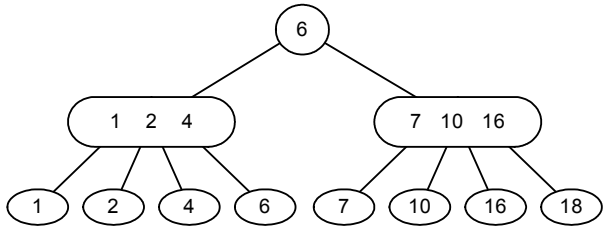

Abb. 1.14 *(2, 4)-Baum*

In einem (a, b)-Baum habe alle Blätter identische Tiefe, d.h. es kann eine minimale und eine maximale Anzahl der Blätter eines (a, b)-Baumes in Abhängigkeit von der Tiefe angegeben werden:

- Ein (2, 4)-Baum mit Tiefe 1 kann 2 bis 4 Schlüssel speichern.
- Ein (2, 4)-Baum mit Tiefe 2 hat 2 – 4 Knoten in Höhe 1, welche wiederum jeweils 2 – 4 Blätter haben. Diese Version speichert also 4 bis 16 Datenelemente.
- Bei genauerer Überlegung erhalten wir für den (2, 4)-Baum der Tiefe 3 ein Speichervolumen von 8 – 64 Schlüsseln.

Es existiert also exakt 1 Baum der Tiefe 3 zur Speicherung der 8 Elemente (jeder Knoten hat die minimal nötige Anzahl der Kinder). Alle anderen Varianten des (2, 4)-Baumes zur Speicherung von 8 Elementen haben die Tiefe 2. Da die Schlüssel sortiert abgelegt werden, gilt (für 2, 3 bzw. 4 Knoten an der Wurzel, die Additionen zeigen die möglichen Permutationen):

- 1 Variante mit 4 Kindern an 2 Knoten (siehe Abbildung oben): 4 + 4.
- 3 Varianten mit 3 Kindern an 2 Knoten und 2 Kindern an einem Knoten: 2 + 3 + 3, 3 + 2 + 3, 3 + 3 + 2.
- 3 Varianten mit 2 Kindern an 2 Knoten und 4 Kindern an einem Knoten: 4 + 2 + 2, 2 + 4 + 2, 2 + 2 + 4.
- 1 Variante mit 2 Kindern an 4 Knoten: 2 + 2 + 2 + 2.

Insgesamt sind also 9 Varianten möglich.