4 Heaps

Es gibt eine Vielzahl von Anwendungen, die effizient das größte bzw. kleinste Element aus einer Menge von Elementen finden und extrahieren müssen. Eine Datenstruktur, die eine effiziente Maximumsextraktion (bzw. Minimumsextraktion), Einfügeoperation und Löschoperation anbietet, nennt man *Prioritätswarteschlange*.

Anwendungen von Prioritätswarteschlangen. Beispielsweise muss ein Betriebssystem ständig (und natürlich unter Verwendung von möglichst wenig Rechenressourcen) entscheiden, welcher Prozess als Nächstes mit der Ausführung fortfahren darf. Dazu muss der Prozess mit der höchsten Priorität ausgewählt werden. Außerdem kommen ständig neue Prozesse bzw. Tasks hinzu. Man könnte die entsprechende Funktionalität dadurch gewährleisten, dass die Menge von Tasks nach jedem Einfügen eines Elementes immer wieder neu sortiert wird, um dann das größte Element effizient extrahieren zu können; Heaps bieten jedoch eine effizientere Möglichkeit dies zu implementieren.

Auch einige Algorithmen, wie beispielsweise der Dijkstra-Algorithmus zum Finden kürzester Wege oder Prims Algorithmus zum Finden eines minimalen Spannbaums, verwenden Prioritätswarteschlangen und sind auf eine effiziente Realisierung der Einfügeoperation und der Minimumsextraktion angewiesen.

Heaps als Implementierungen von Prioritätswarteschlangen. Als *Heap* bezeichnet man in der Algorithmik einen Baum, dessen Knoten der sog. Min-Heap-Bedingung (bzw. Max-Heap-Bedingung – abhängig davon, ob man sich für die minimalen oder maximalen Werte interessiert) genügen. Ein Knoten genügt genau dann der (Min-)Heap-Bedingung, wenn sein Schlüsselwert kleiner ist als die Schlüsselwerte seiner Kinder.

Die in diesem Abschnitt vorgestellten Datenstrukturen stellen allesamt mögliche Implementierungen von Prioritätswarteschlagen dar, die die Operationen „Einfügen“, „Minimumsextraktion“, „Löschen“ und evtl. „Erniedrigen eines Schlüsselwerts“ effizient unterstützen. Die in Abschnitt 4.1 beschriebenen binären Heaps stellen hierbei die „klassische“ Implementierung von Prioritätswartschlangen dar. Binäre Heaps wurden eigentlich schon in Kapitel 2 bei der Beschreibung des Heapsort-Algorithmus verwendet, werden aber in diesem Kapitel der Vollständigkeit halber nochmals vorgestellt.

Binomial-Heaps (siehe Abschnitt 4.2), Fibonacci-Heaps (siehe Abschnitt 4.3) und Pairing-Heaps (siehe Abschnitt 4.4) sind zusätzlich in der Lage die Verschmelzung zweier Heaps effizient zu unterstützen. Eine solche Verschmelzungsoperation spielt beispielsweise beim Prozessmanagement von Rechnern mit parallelen Prozessoren bzw. parallelen Threads eine Rolle: Gibt ein Prozessor seine „Arbeit“ an einen anderen Prozessor ab, so erfordert dies u. A. die Verschmelzung der Prozesswarteschlangen beider Prozessoren.

4.1 Binäre Heaps

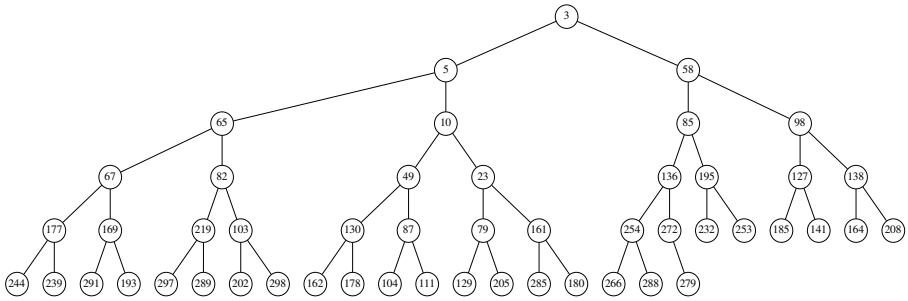


Abb. 4.1: *Beispiel eines binären Min-Heaps der $n = 50$ Elemente enthält.*

Binäre Heaps stellen wahrscheinlich die am häufigsten verwendete Art der Implementierung von Prioritätswarteschlangen dar. Wie für jeden anderen Heap auch, muss für jeden Knoten v eines binären Heaps die Min-Heap-Bedingung erfüllt sein, d. h. die Schlüsselwerte der Kinder von v müssen größer sein als der Schlüsselwert von v. Zusätzlich ist ein binärer Heap immer ein vollständiger Binärbaum, dessen Ebenen alle vollständig gefüllt sind; nur die unterste Ebene des Heaps ist, falls die Anzahl n der im Heap enthaltenen Elemente keine Zweierpotenz (minus Eins) ist, linksbündig unvollständig gefüllt. Abbildung 4.1 zeigt ein Beispiel eines Min-Heaps.

Obwohl einige Operationen (wie beispielsweise die Einfügeoperation oder das Erniedrigen eines Schlüsselwertes) für binäre Heaps eine schlechtere (asymptotische) Laufzeitkomplexität besitzen als für alternative Implementierungen, wie Fibonacci-Heaps oder Pairing-Heaps, stellen sie trotzdem in vielen Fällen die sinnvollste Implementierung dar: Zum Einen weil die in der O-Notation der Laufzeit versteckten Konstanten relativ „klein“ sind; zum Anderen weil wegen dessen fester Struktur ein binärer Heap in einem zusammenhängenden festen Speicherbereich gehalten werden kann. Zusätzlich werden wir sehen, dass die Implementierung der meisten Operationen relativ (zumindest im Vergleich zur Implementierung der entsprechenden Operationen für Binomial-Heaps und Fibonacci-Heaps) einfach ist.

4.1.1 Repräsentation binärer Heaps

Binäre Heaps sind laut Definition immer vollständige Binärbäume, haben also eine feste Struktur, die nicht explizit gespeichert werden muss. Es bietet sich daher eine „strukturlose“ Repräsentation als Liste an. Hierbei schreibt man die Einträge des Heaps von der Wurzel beginnend ebenenweise in die Liste, wobei die Einträge jeder Ebene von links nach rechts durchlaufen werden. Wir werden gleich sehen, dass es hier günstig ist, den ersten Eintrag der den Heap repräsentierenden Liste freizuhalten; konkret setzen wir diesen auf „*None*“. Der Min-Heap aus Abbildung 4.1 wird beispielsweise durch die folgende Liste repräsentiert:

$$[\mathit{None}\ ,3,5,58,65,10,85,98,67,82,49,23,136,195,127,\ \ldots\ ]$$

Repräsentiert man also einen Heap als Liste l, so ist leicht nachvollziehbar, dass das linke Kind von $l[i]$ der Eintrag $l[2*i]$ und das rechte Kind der Eintrag $l[2*i+1]$ ist.

Aufgrund der Struktur des binären Heaps gilt, dass die Höhe eines Heaps der n Elemente enthält immer $\lceil \log_2 n \rceil$ ist, also in $O(\log n)$ ist.

4.1.2 Einfügen eines Elements

Das in Listing 4.1 gezeigte Programm implementiert die Operation „Einfügen“ eines Elementes in einen als Liste repräsentierten Heap.

```
1 def insert(heap, x):
2   heap.append(x)
3   i = len(heap)-1
4   while heap[i/2]>heap[i]:
5     heap[i/2],heap[i] = heap[i],heap[i/2]
6     i = i/2
```

Listing 4.1: *Einfügen eines Elementes in einen als Liste repräsentierten Min-Heap*

Das einzufügende Element x wird zunächst hinten an den Heap angehängt; dies entspricht dem Kommando *heap.append*(x) in Zeile 2. Anschließend wird das eingefügte Element solange durch Tausch mit dem Vaterknoten die Baumstruktur „hoch“transportiert, bis die Heapbedingung erfüllt ist. Die **while**-Schleife wird solange durchlaufen wie der Wert des eingefügten Knotens kleiner ist als der Wert seines Vaterknotens, d. h. sie wird solange durchlaufen wie die Bedingung *heap*$[i/2]>$*heap*$[i]$ gilt.

Da die Anzahl der Tauschungen durch die Höhe des Heaps begrenzt ist, ist die Laufzeit dieser Operation offensichtlich in $O(\log n)$.

4.1.3 Minimumsextraktion

Das minimale Element eines binären Heaps wird wie folgt extrahiert: Das letzte Element aus einer den Heap repräsentierenden Liste *heap*, also *heap*$[-1]$, wird an die Stelle der Wurzel, also *heap*$[1]$, gesetzt. Dies verletzt i. A. die Heap-Bedingung. Die Heap-Bedingung kann wiederhergestellt werden, indem man dieses Element solange durch Tauschen mit dem kleineren der beiden Kinder nach „unten“ transportiert, bis die Heap-Bedingung wiederhergestellt ist.

Listing 4.2 zeigt eine Implementierung der Minimumsextraktion. In der Variablen n ist während des ganzen Programmablaufs immer der Index des „letzten“ Elements des Heaps gespeichert. In den Zeilen 3 und 4 wird das „letzte“ Element des Heaps an die Wurzel gesetzt. Die Durchläufe der **while**-Schleife transportieren dann das Wurzel-Element solange nach „unten“, bis die Heap-Bedingung wieder erfüllt ist. Am Anfang der **while**-Schleife zeigt die Variable i immer auf das Element des Heaps, das möglicherweise die Heap-Bedingung noch verletzt. In Zeile 9 wird das kleinere seiner beiden Kinder ausgewählt; falls dieses Kind größer ist als das aktuelle Element, d. h. falls $lst[i] \leq lst[k]$, so ist die Heap-Bedingung erfüllt und die Schleife kann mittels **break**

```
1  def minExtract(lst):
2    returnVal=lst[1]
3    lst[1]=lst[-1] # letztes Element an die Wurzel
4    del(lst[-1])
5    n=len(lst)-1   # n zeigt auf das letzte Element
6    i=1
7    while i≤n/2:
8      j=2*i
9      if j<n and lst[j]>lst[j+1]: j+=1 # wähle kleineres der beiden Kinder
10     if lst[i]≤lst[j]: break
11     lst[i], lst[j]=lst[j], lst[i]
12     i=j
13   return returnVal
```

Listing 4.2: *Implementierung der Minimumsextraktion, bei der das Wurzel-Element des Heaps entfernt wird.*

abgebrochen werden. Falls jedoch dieses Kind kleiner ist als der aktuelle Knoten, ist die Heapbedingung verletzt, und Vater und Kind müssen getauscht werden (Zeile 11). Durch die Zuweisung $i=j$ fahren wir im nächsten **while**-Schleifendurchlauf damit fort, den getauschten Knoten an die richtige Position zu bringen.

Die Höhe des Heaps begrenzt die maximal notwendige Anzahl der Vergleichs- und Tauschoperationen auch bei der Minimumsextraktion. Damit ist die Laufzeit der Minimumsextraktion auch in $O(\log n)$.

4.1.4 Erhöhen eines Schlüsselwertes

Soll ein Element *heap*[i] eines als Liste repräsentierten Heaps *heap* erhöht werden, so ist die Heap-Bedingung nach dem Erhöhen evtl. verletzt. Die Heap-Bedingung kann dadurch wiederhergestellt werden, indem man das Element soweit im Heap „sinken“ lässt (d. h. sukzessive mit einem der Kinder tauscht), bis die Heap-Bedingung wiederhergestellt ist. Die in Listing 4.3 gezeigte Funktion *minHeapify* implementiert dies.

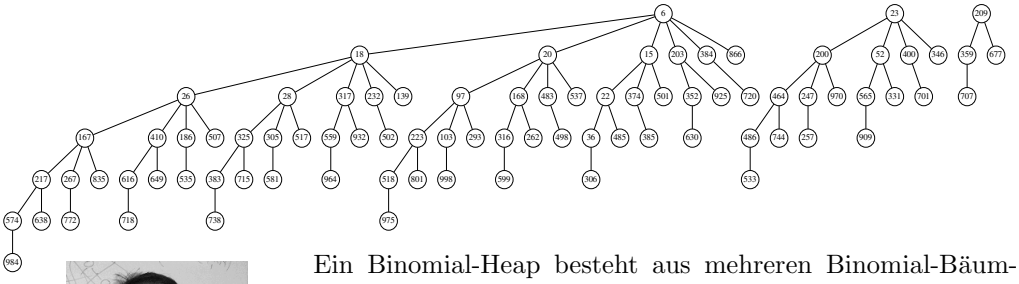
Die Funktion *minHeapify* stellt die Heap-Bedingung, falls diese verletzt ist, für den Knoten an Index i des Heaps *heap* wieder her, und zwar dadurch, dass der Knoten im Heap solange nach „unten“ gereicht wird, bis die Heap-Bedingung wieder erfüllt ist. Die in Zeile 2 und 3 definierten Variablen l und r sind die Indizes der Kinder des Knotens an Index i. In Zeile 5 wird mittels einer Listenkomprehension eine i. A. dreielementige Liste *nodes* aus den Werten des Knotens an Index i und seiner beiden Kinder erstellt. Um den Knoten mit kleinstem Wert zu bestimmen, wird *nodes* sortiert; danach befindet sich der *Wert* des kleinsten Knotens in *nodes*[0][0] und der Index des kleinsten Knotens in *nodes*[0][1]. Falls der Wert des Knotens i der kleinste der drei Werte ist, ist die Heap-Bedingung erfüllt und die Funktion *minHeapify* kann verlassen werden; falls andererseits einer der Kinder einen kleineren Wert hat (d. h. *smallestIndex*$\neq i$) so ist die Heap-Bedingung verletzt und der Knoten an Index i wird durch Tauschen mit dem kleinsten Kind nach „unten“ gereicht; anschließend wird rekursiv weiterverfahren.

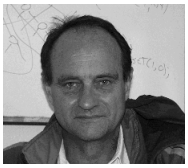
```
1  def minHeapify(heap,i):
2    l = 2*i
3    r = l+1
4    n = len(heap)-1
5    nodes = [(heap[v],v) for v in [i,l,r] if v≤n]
6    nodes.sort()
7    smallestIndex = nodes[0][1]
8    if smallestIndex ≠ i:
9      heap[i],heap[smallestIndex] = heap[smallestIndex],heap[i]
10     minHeapify(heap,smallestIndex)
```

Listing 4.3: *Die Funktion minHeapify, die den Knoten an Index i soweit sinken lässt, bis die Heap-Bedingung des Heaps „heap“ wiederhergestellt ist.*

Auch die Laufzeit dieser Operation ist durch die Höhe des binären Heaps begrenzt und liegt in $O(\log n)$.

4.2 Binomial-Heaps





Bildquelle:
http://www.di.ens.fr/ jv/

Ein Binomial-Heap besteht aus mehreren Binomial-Bäumen, deren Knoten jeweils die Heap-Bedingung erfüllen. Diese Bäume besitzen eine festgelegte rekursive Struktur, die eine einfache Verschmelzung zweier Bäume erlaubt.
Binomial-Heaps wurden 1978 [18] von Jean Vuillemin, Professor für Informatik an der an der Ecole Normale Superieure in Paris, eingeführt.

Wie schon zu Beginn des Kapitels erwähnt, gibt es einige Anwendungen, die eine effiziente Vereinigung zweier Heaps benötigen; man denke etwa an Mehrkern-Prozessorsysteme, die je nach Auslastung der Prozessoren Prozess-Prioritätswarteschlangen aufteilen bzw. vereinigen müssen. Während herkömmliche binäre Heaps keine „schnelle“ Vereinigung (in $O(\log n)$ Schritten) unterstützen, sind Binomial-Heaps gerade auf die Unterstützung einer effizienten Vereinigung hin entworfen.

Aufgabe 4.1

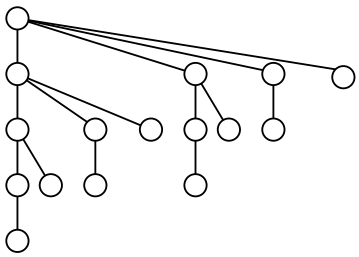
Implementieren Sie die Vereinigungs-Operation *mergeHeaps*, die zwei binäre Heaps miteinander vereinigt. Welche Laufzeit hat ihre Implementierung?

4.2.1 Binomial-Bäume

Ein Binomial-Heap besteht aus mehreren Binomial-Bäumen. Wir beginnen zunächst mit der Definition von Binomial-Bäumen. Die Struktur eines Binomial-Baums der Ordnung k kann folgendermaßen induktiv definiert werden:

- Ein Binomial-Baum der Ordnung „0“ besteht aus einem einzelnen Knoten.
- Ein Binomial-Baum der Ordnung k besteht aus einem Wurzelknoten mit k Nachfolgern: Der erste Nachfolger ist ein Binomial-Baum der Ordnung k, der zweite Nachfolger ist eine Binomial-Baum der Ordnung $k-1$, usw.; der letzte Nachfolger ist ein Binomial-Baum der Ordnung „0“, also ein einzelner Knoten.

Ein Binomial-Baum beispielsweise der Ordnung 4 hat folgende Struktur:



Ein Binomial-Baum der Ordnung k enthält genau 2^k Elemente; dies kann man einfach über vollständige Induktion zeigen – siehe hierzu Aufgabe 4.2.

Aufgabe 4.2

Wie viele Knoten hat ein Binomial-Baum der Ordnung k?

(a) Schreiben Sie eine rekursive Python-Funktion *anzKnotenBinomial*(k), die die Anzahl der Knoten eines Binomial-Baums der Ordnung k zurückliefert; diese Funktion sollte sich an der induktiven Definition eines Binomial-Baums orientieren.

(b) Zeigen Sie mit Hilfe der vollständigen Induktion, dass ein Binomial-Baum der Ordnung k genau 2^k Elemente enthält.

4.2.2 Repräsentation von Binomial-Bäumen

Es gibt – wie auch bei vielen anderen Datenstrukturen – mehrere Möglichkeiten der Repräsentation. Binomial-Bäume können in Python etwa als Klasse repräsentiert werden.

Legt man Wert auf eine klare Darstellung der Algorithmen, so scheint eine möglichst einfache Repräsentation am günstigsten, etwa die Repräsentation eines Binomial-Baums als Tupel. Die erste Komponente des Tupels enthält das Element an der Wurzel des Binomial-Baums und die zweite Komponente ist eine Liste der Unterbäume des Binomial-Baums. Die Repräsentation eines Binomial-Baums der Ordnung k hätte in Python also das folgende Aussehen (wobei x der an der Wurzel gespeicherte Wert und bt_i ein Binomial-Baum der Ordnung i darstellt):

$$(x, [\, bt_{k-1}\,, bt_{k-2}\,, bt_{k-3}\,, \;\; \ldots\;, bt_1\,, \;\; bt_0\,])$$

Ist bt ein Binomial-Baum der Ordnung k, so muss also immer $len(\,bt\,[1\,]) == k$ sein.

Zwei einfache Beispiele: Ein Binomial-Baum der Ordnung 0 dessen Wurzel die Zahl „13“ enthält entspricht somit dem Python-Wert $(13, [\,])$; der in Abbildung 4.2 gezeigte Binomial-Baum der Ordnung 2 entspricht dem folgenden Python-Wert:
$bt2 = (72,[(77,[(89,[\,])\,]\,),(91,\,[\,])\,])$

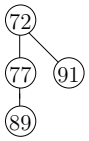


Abb. 4.2: *Binomial-Baum der Ord. 2.*

Aufgabe 4.3

Implementieren Sie eine Python-Funktion *isBinomial*(bt), die genau dann „*True*“ zurückliefert, wenn das Argument bt ein gültiger Binomial-Baum ist.

Aufgabe 4.4

Implementieren Sie eine Python-Funktion *bt2str*$(\,bt)$, die einen Binomial-Baum bt in eine geeignete Stringrepräsentation umwandelt.

4.2.3 Struktur von Binomial-Heaps

Jeder Binomial-Heap besteht aus mehreren Binomial-Bäumen verschiedener Ordnungen; für jeden der Binomial-Bäume muss zusätzlich die Heapbedingung erfüllt sein, d. h. (im Falle von Min-Heaps) muss ein Knoten immer einen größeren Wert gespeichert haben als seine Kinderknoten.

Wollen wir n Elemente in einem Binomial-Heap speichern, so ist die Struktur dieses Binomial-Heaps bestimmt durch die Binärdarstellung der Zahl n. Angenommen wir wollen 22 (in Binärdarstellung: „10110“) Elemente in einem Binomial-Heap speichern, so muss dieser Binomial-Heap genau einen Binomial-Baum der Ordnung 4 (das von rechts gesehen, von Null an gezählte Bit an Position „4“ von „10110“ ist gesetzt), einen Binomial-Baum der Ordnung 2 (das Bit an Position „2“ von „10110“ ist gesetzt) und einen Binomial-Baum der Ordnung 1 (das Bit an Position „1“ von „10110“ ist gesetzt) enthalten. Ebenso wie die Binärdarstellung der Zahl 22 eindeutig bestimmt ist, ist auch die Struktur des Binomial-Heaps (nicht jedoch notwendigerweise die Anordnung der Elemente im Heap) eindeutig bestimmt. Abbildung 4.3 zeigt ein Beispiel eines Binomial-Heap, der 22 Elemente enthält.

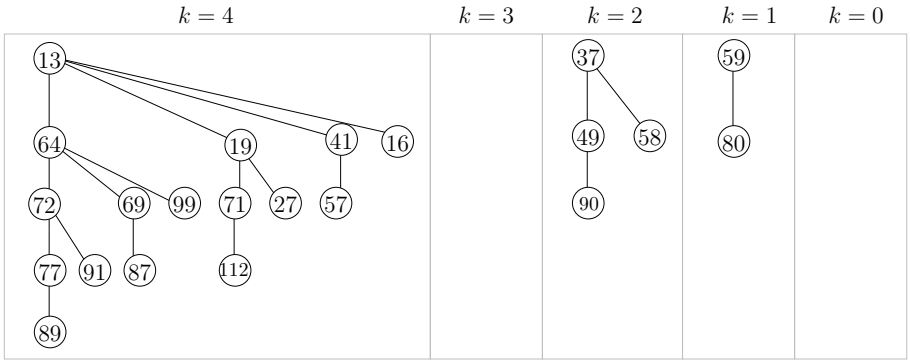


Abb. 4.3: *Beispiel eines Binomial-Heap mit 22 Elementen, dessen Knoten der Min-Heap-Bedingung genügen.*

4.2.4 Repräsentation von Binomial-Heaps

Offensichtlich kann man einen Binomial-Heap in Python einfach als Liste von Binomial-Bäumen repräsentieren; der in Abbildung 4.3 gezeigte Binomial-Heap beispielsweise wäre durch folgenden Python-Wert repräsentiert:

$$[bt4,\ None,\ bt2,\ bt1,\ None]$$

wobei *bt1*, *bt2* und *bt4* jeweils die in Abbildung 4.3 gezeigten Binomial-Bäume der Ordnung 1, 2 bzw. 4 darstellen.

Aufgabe 4.5

Geben Sie die Pythonrepräsentation des in Abbildung 4.3 gezeigten Binomial-Heaps an.

Aufgabe 4.6

Implementieren Sie eine Python-Funktion *isBinHeap*(*bh*), die genau dann „*True*“ zurückliefert, wenn *bh* ein gültiger Binomial-Heap ist.

4.2.5 Verschmelzung zweier Binomial-Bäume

Die Struktur der Binomial-Bäume ist genau deshalb algorithmisch so interessant, weil man zwei Binomial-Bäume *bt1* und *bt2* der Ordnung k sehr einfach in $O(1)$ Schritten zu einem Binomial-Baum der Ordnung $k+1$ verschmelzen kann. Angenommen *bt1*<*bt2* (d. h. der an der Wurzel von *bt1* gespeicherte Wert ist kleiner als der in der Wurzel von *bt2* gespeicherte Wert). Dann besteht die Verschmelzungsoperation einfach darin, *bt2* als linkesten Teilbaum unter den Binomial-Baum *bt1* zu hängen. Der Wurzelknoten dieses neuen Baums hat $k+1$ Kinder, die jeweils Binomial-Bäume der Ordnung k, $k-1$, ..., 0 darstellen – ist also ein Binomial-Baum der Ordnung $k+1$. Abbildung 4.4 zeigt

die Verschmelzung zweier Binomial-Bäume der Ordnung 3 zu einem Binomial-Baum der Ordnung 4.

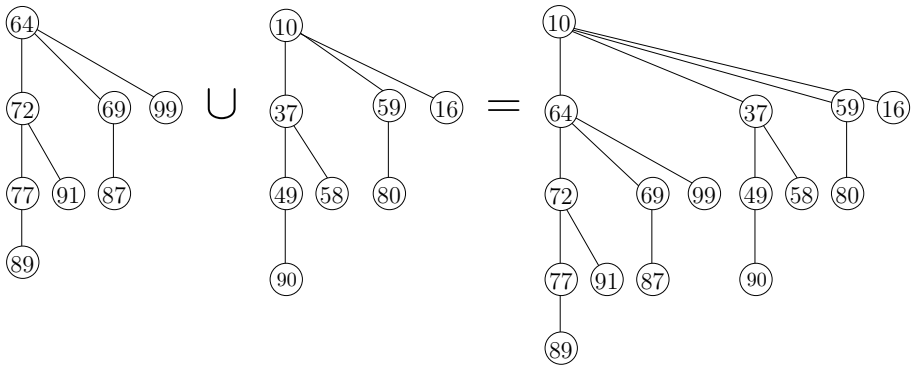


Abb. 4.4: *Verschmelzung zweier Binomial-Bäume der Ordnung k zu einem Binomial-Baum der Ordnung $k+1$ – hier ist $k=3$.*

Auch die entsprechende in Listing 4.4 gezeigte Implementierung in Python ist relativ simpel. Die Funktion *meltBinTree* liefert einen neuen Binomial-Baum zurück, der die

```
1 def meltBinTree(bt0,bt1):
2   # Voraussetzung: bt0<bt1
3   root = lambda x : x[0]
4   subtrees = lambda x : x[1]
5   return ( root(bt0), [bt1] + subtrees(bt0) )
```

Listing 4.4: *Verschmelzung zweier Binomial-Bäume*

Verschmelzung der beiden Binomial-Bäume *bt0* und *bt1* darstellt; dieser wird direkt nach dem **return**-Kommando generiert und besteht einfach aus dem Wurzelknoten *root*(*bt0*) des Baumes *bt0*; der linkeste Unterbaum ist der komplette Binomial-Baum *bt1*; die weiteren Unterbäume sind die Unterbäume des Binomial-Baums *bt0*, nämlich *subtrees*(*bt0*).

4.2.6 Vereinigung zweier Binomial-Heaps

Die Verschmelzung zweier Binomial-Heaps hat große Ähnlichkeit mit der Addition zweier Binärzahlen: Ein gesetztes Bit an der k-ten binären Stelle entspricht dem Vorhandensein eines Binomial-Baums der Ordnung k im Binomial-Heap; ein nicht-gesetztes Bit an der k-ten binären Stelle entspricht dagegen einem *None*-Eintrag an der von rechts gesehen k-ten Stelle der Python-Liste, die den Binomial-Heap repräsentiert. Auch das Verwenden eines Carry-Bits und die bitweise Berechnung der einzelnen Stellen durch einen Volladdierer hat eine Entsprechung bei der Vereinigung zweier Binomial-Heaps.

Abbildung 4.5 zeigt ein Beispiel für die Vereinigung zweier Binomial-Heaps; auch die Darstellung in dieser Abbildung ist angelehnt an die Addition zweier Binärzahlen. Während der Vereinigung entstehen zwei Carry-Bäume, die genau wie ein Carry-Bit in den für die nächste Stelle zuständigen Volladdierer einfließen.

Listing 4.5 zeigt die Implementierung eines „Volladdierers“, der zwei Binomial-Bäume und einen Carry-Baum addiert und ein Tupel bestehend aus einem dem Summen-Bit entsprechenden Binomial-Baum und einem dem Carry-Bit entsprechenden Binomial-Baum zurückliefert. Ein nicht-gesetztes Bit entspricht wiederum dem Wert „*None*“, ein gesetztes Bit entspricht einem Binomial-Baum der Ordnung k.

```
1 def fullAddB(bt0,bt1,c):
2   bts = sorted([b for b in [bt0,bt1,c] if b])
3   if len(bts)≥2:
4     c = meltBinTree(bts[0],bts[1])
5     return (None if len(bts)==2 else bts[2], c)
6   else:
7     return (None if len(bts)==0 else bts[0], None)
```

Listing 4.5: *Implementierung des Pendants eines Volladdierers zur Vereinigung zweier Binärbäume und eines Carry-Baums.*

Zunächst werden in Zeile 2 die *None*-Werte mittels der Bedingung „**if** b“ in der Listenkomprehension ausgefiltert und die übergebenen Binomial-Bäume der Größe nach sortiert in der Liste *bts* abgelegt. Da die Sortierung stets lexikographisch erfolgt (siehe auch Anhang A.6 auf Seite 295) erhält man dadurch in $bts[0]$ den Baum mit dem kleinsten Wurzelelement und in $bts[2]$ den Baum mit dem größten Wurzelelement; diese Information ist für die Verschmelzungsoperation in Zeile 4 wichtig. Immer dann, wenn der Funktion *fullAddB* zwei oder mehr Binomial-Bäume der Ordnung k übergeben werden, wird in Zeile 4 ein Carry-Baum der Ordnung $k+1$ erzeugt, der dann zusammen mit dem Summenbaum in Zeile 5 zurückgeliefert wird. Wurden weniger als zwei Binomial-Bäume übergeben, so wird als Carry-Baum „*None*“ und als Summen-Baum der eine übergebene Binomial-Baum übergeben (bzw. „*None*“ falls nur „*None*“-Werte übergeben wurden).

Die Verschmelzung zweier Binomial-Heaps erfolgt nun einfach durch die stellenweise Ausführung von *fullAddB*. Listing 4.6 zeigt eine Implementierung.

```
1 def merge(h1,h2):
2   h1 = [None]*(len(h2)-len(h1)) + h1
3   h2 = [None]*(len(h1)-len(h2)) + h2
4   erg=[None]*(len(h1)+1) ; c = None
5   for i in range(len(h1))[::-1]:
6     (s,c) = fullAddB(h1[i],h2[i],c)
7     erg[i+1]=s
8   erg[0]=c
9   return erg
```

Listing 4.6: *Verschmelzung zweier Binomial-Heaps*

Abb. 4.5: *Vereinigung zweier Binomial-Heaps. Der obere Heap enthält* $22 = 10110\,b$ *Elemente, der untere Heap enthält* $12 = 01100\,b$ *Elemente. Die Vereinigung der beiden Heaps hat Ähnlichkeit mit der binären Addition von* $10110\,b$ *und* $01100\,b$*. Betrachten wir die stellenweise Addition beginnend mit dem niederwertigsten (rechten) in zu höherwertigsten (linken) Bit. Anfänglich ist – wie bei jeder Addition – das Carry-Bit „0" und es wird zunächst* $fullAdd(0,0,0)$ *berechnet – dies entspricht der Berechnung von* $fullAddB(None,None,None)$ *bei der Heap-Vereinigung. Zur Berechnung der zweiten Stelle der Addition wird* $fullAdd(1,0,0)$ *berechnet – dies entspricht der Berechnung* $fullAddB((59,[(80,[])]),None,None)$*; die Summe entspricht hierbei einfach dem ersten Argument, das neue Carry-Bit bleibt None. Bei der Addition an Stelle* $k = 2$ *entsteht ein Carry-Baum (dargestellt in einem weißen Kasten), der in die Addition an Stelle* $k = 3$ *wieder einfließt. Auch bei der Addition an Stelle* $k = 3$ *entsteht wieder ein Carry-Baum der seinerseits in die Addition an Stelle* $k = 4$ *einfließt. Ergebnis ist schließlich ein Binomial-Heap der einen Eintrag mehr besitzt als seine beiden Summanden.*

In den Zeilen 2 und 3 werden die beiden übergebenen Heaps auf die gleiche Länge gebracht, indem gegebenenfalls „*None*“-Werte links (also an den höherwertigen „Bit“-Positionen) eingefügt werden. In der Variablen *erg* speichern wir das Ergebnis der Verschmelzung und füllen diese zunächst mit $len(h1)+1$ „*None*“-Werten auf, also einer Stelle mehr, als der längere der beiden übergebenen Binomial-Heaps. Analog zur bitweisen Addition zweier Binärzahlen, setzen wir anfänglich den Carry-Baum auf „*None*“. Die **for**-Schleife ab Zeile 5 läuft über die Stellen der Binomial-Heaps und führt für jede Stelle eine Volladdition durch. Schließlich wird der zuletzt entstandene Übertrag der höchstwertigen Stelle von *erg* zugewiesen. Man beachte, dass in der **for**-Schleife ab Zeile 5 die Binomial-Baum-Listen von „hinten“ nach „vorne“ durchlaufen werden, also tatsächlich von der niederwertigsten Stelle $h1[-1]$ bzw. $h2[-1]$ bis zur höchstwertigsten Stelle $h1[0]$ bzw. $h2[0]$.

Die Laufzeit der Verschmelzung zweier Binomial-Heaps mit jeweils n bzw. m Elementen liegt offensichtlich in $O(\log(n+m))$: Die **for**-Schleife ab Zeile 5 wird $len(h) \leq \log_2(n+m)$-mal durchlaufen und die Ausführung der Funktion *fullAddB* benötigt $O(1)$ Schritte.

4.2.7 Einfügen eines Elements

Man kann ein Element x einfach dadurch in einen Binomial-Heap bh einfügen, indem man aus x einen einelementigen Binomial-Heap (bestehend aus einem einelementigen Binomial-Baum der Ordnung 0) erzeugt und diesen dann mit bt verschmilzt.

Aufgabe 4.7

Implementieren Sie eine Funktion *insertBinomialheap*(bh,x) die als Ergebnis einen Binomial-Heap zurückliefert, der durch Einfügen von x in bh entsteht.

Die Einfügeoperation hat offensichtlich eine Worst-Case-Laufzeit von $O(\log n)$, die etwa dann eintritt, wenn durch den Verschmelzungsprozess alle „Bits“ des Binomial-Heap bh von Eins auf Null „umkippen“, wenn also in einen Binomial-Heap mit 2^n-1 enthaltenen Elementen ein neues Element hinzugefügt wird. Da dieser Fall jedoch selten eintritt, kann man zeigen, dass die amortisierte Laufzeit in $O(1)$ liegt. Um diese theoretisch mögliche (amortisierte) Laufzeit voo $O(1)$ zu erreichen, müsste jedoch die in Listing 4.6 gezeigte Implementierung angepasst werden; siehe hierzu auch die folgende Aufgabe 4.8.

Aufgabe 4.8

...

4.2.8 Extraktion des Minimums

Ein Heap, der dazu verwendet wird, eine Prioritätswarteschlange zu implementieren, sollte effizient das Finden und die Extraktion des minimalen Elements unterstützen.

Zunächst können wir feststellen, dass das Finden des minimalen Elements $O(\log n)$ Schritte benötigt: Alle Wurzelelemente der $O(\log n)$ Binomial-Bäume müssen hierfür verglichen werden.

Nehmen wir an, das minimale Wurzelelement ist das Wurzelelement eines Binomial-Baums bt_k der Ordnung k. Das anschließende Löschen dieses Elements erzeugt k „freie“ Binomial-Bäume. Diese werden dann in einem Binomial-Heap (der $2^k - 1$ Elemente enthält) zusammengefasst und mit dem ursprünglichen Binomial-Heap (ohne bt_k) verschmolzen.

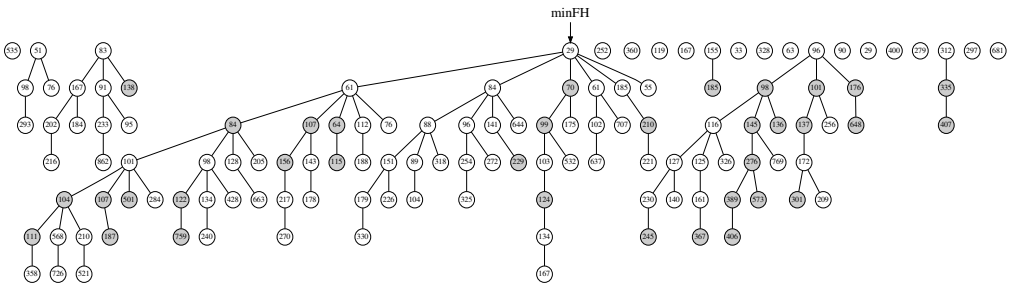
Listing 4.7 zeigt die Implementierung der Extraktion des minimalen Elements.

```
1 def minExtractB(bh):
2   (bt,k) = min([(bt,k) for k,bt in enumerate(bh) if bt ≠ None])
3   bh2 = [None if i==k else bt2 for i,bt2 in enumerate(bh)]
4   return minEl,merge(bh2,bt[1])
```

Listing 4.7: *Implementierung der MinExtract-Funktion*

Zunächst wird in Zeile 2 derjenige Binomial-Baum bt gesucht, der das minimale Element als Wurzelelement besitzt – dies kann in $O(\log n)$ Schritten erfolgen. Das Wurzelelement $bt[0]$ dieses Binomial-Baums wird dann in Zeile 4 zusammen mit dem Binomial-Heap zurückgegeben, der durch Löschen von $bt[0]$ entsteht.

4.3 Fibonacci Heaps



Michael Fredman und Robert Tarjan entwickelten im Jahr 1984 die Fibonacci-Heaps und publizierten ihre Entdeckung im Jahre 1987 [9].

Fibonacci-Heaps sind Binomial-Heaps ähnlich, und tatsächlich waren Fibonacci-Heaps von Tarjan und Fredman auch als eine Art „Verbesserung“ von Binomial-Heaps gedacht. Wie an obiger Abbildung eines Fibonacci-Heaps schon zu erkennen, sind sie etwas weniger strukturiert als Binomial-Heaps. Sie besitzen für einige wichtige Operationen wie die Verschmelzung und die Minimumsbestimmung eine bessere (amortisierte) Laufzeit als Binomial-Heaps.

Ebenso wie Binomial-Heaps bestehen auch Fibonacci-Heaps aus einer Menge von einzelnen Bäumen, die jeweils der Heap-Bedingung genügen. Jedoch ist die Struktur eines

Fibonacci-Heaps flexibler und einige notwendige Restrukturierungs-Operationen etwa bei der Verschmelzung zweier Fibonacci-Heaps werden geschickt auf einen späteren Zeitpunkt verschoben; durch dieses „Verschieben“ kann eine erstaunlich gute amortisierte Laufzeit vieler Operationen erreicht werden: Die Verschmelzung zweier Heaps, etwa, ist so in einer amortisierten Laufzeit von $O(1)$ möglich; das Erniedrigen eines Schlüsselwertes ist ebenfalls in $O(1)$ möglich.

4.3.1 Struktur eines Fibonacci-Heaps

Ein Fibonacci-Heap besteht aus einer Liste einzelner Bäume, die jeweils die (Min-)Heap-Bedingung erfüllen – also ihrerseits Heaps sind. Es gilt also, dass der Schlüsselwert eines Knotens immer kleiner sein muss als die Schlüsselwerte seiner Kinder. Diese Bäume, aus denen ein Fibonacci-Heap besteht, bezeichnen wir im Folgenden auch als *Fibonacci-Bäume*. Genau wie im Falle der Binomial-Heaps definiert man die Ordnung eines Fibonacci-Baums als die Anzahl der Kinder, die das Wurzelelement besitzt.

Zusätzlich wird ein Zeiger auf den Fibonacci-Baum mitgeführt, dessen Wurzel das minimale Element des Fibonacci-Heaps enthält. Dies ermöglicht etwa eine Implementierung der *getMin*-Funktion in $O(1)$ Schritten.

Aufgabe 4.9

Erklären Sie, warum der Knoten mit minimalem Schlüsselwert sich immer an der Wurzel eines Fibonacci-Baums befinden muss.

Einige Knoten des Fibonacci-Heaps sind markiert – in Abbildung 4.6 sind dies die grau-gefüllten Knoten.

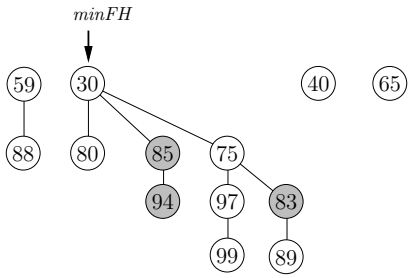


Abb. 4.6: *Beispiel eines Fibonacci-Heaps, der aus vier Fibonacci-Bäumen besteht: einem Fibonacci-Baum der Ordnung 1, einem Fibonacci-Baum der Ordnung 3 und zwei Fibonacci-Bäumen der Ordnung 0. Der Fibonacci-Heap enthält einen Zeiger, der auf den Fibonacci-Baum zeigt, der das minimale Element Fibonacci-Heaps als Wurzelelement enthält.*

Wie wir in Abschnitt 4.3.8 zeigen werden, stellen alle Operationen auf Fibonacci-Heaps sicher, dass der maximale Grad aller Knoten in $O(\log n)$ ist. Genauer: Der Grad aller Knoten eines Fibonacci-Heaps mit n Elementen ist immer $\leq \log_\phi(n)$ mit $\phi = (1+\sqrt{5})/2$.

4.3.2 Repräsentation in Python

Es gibt viele mögliche Arten Fibonacci-Heaps in Python zu repräsentieren:

- Der klassische objektorientierte Ansatz besteht darin, eine Klasse (etwa mit Namen *FibonacciHeap*) zu definieren, alle Komponenten der Datenstruktur (also die einzelnen Bäume, der Zeiger auf den Baum, der das minimale Element enthält, Information darüber, ob ein Knoten markiert ist) als Attribute der Klasse zu definieren und alle Operationen auf Fibonacci-Heaps als Methoden der Klasse *FibonacciHeap* zu definieren. Zwar hat diese Art der Repräsentation in Python einige Vorteile; beispielsweise man kann sich einfacher mittels der *__str__*-Methode eine String-Repräsentation definieren; man kann typsicherer programmieren, usw. Wir bevorzugen jedoch eine andere Art der Repräsentation, die eine knappere und damit prägnantere Formulierung der meisten hier beschriebenen Algorithmen erlaubt.
- Eine Repräsentation ähnlich der für Binomial-Heaps wäre denkbar; im Gegensatz zu einem Binomial-Heap besteht jedoch ein Fibonacci-Heap aus mindestens zwei Komponenten (die Liste der Bäume und der Zeiger auf den Baum, der das minimale Element enthält). Auch ein einzelner Knoten müsste noch die Zusatzinformation mit sich führen, ob er markiert ist und – wie wir später sehen werden – benötigt er einen Zeiger auf seinen Elternknoten.
- Auch eine Repräsentation unter Verwendung von Pythons *dict*-Typs ist möglich. Diese ist der Art der Repräsentation, die wir bei den Binomial-Heaps im letzten Kapitel verwendet haben, ähnlich; jedoch lassen sich so die einzelnen Komponenten eines Fibonacci-Heaps bzw. eines Fibonacci-Baums expliziter benennen. Wir verwenden für die Repräsentation von Fibonacci-Heaps im Weiteren diese Art der Repräsentation.

Ein Fibonacci-Heap besteht aus zwei Komponenten:

- Der „*treesFH*“-Eintrag enthält die Liste der Fibonacci-Bäume, aus denen der Fibonacci-Heap besteht
- Der „*minFH*“-Eintrag enthält den Index desjenigen Fibonacci-Baums, der das minimale Element des Fibonacci-Heaps enthält.

Ein Fibonacci-Baum besteht seinerseits aus vier Komponenten:

- Der „*rootFT*“-Eintrag enthält den im jeweiligen Knoten gespeicherten Schlüsselwert.
- Der „*subtreesFT*“-Eintrag enthält die Liste der Kinder des Knotens.
- Der „*markedFT*“-Eintrag enthält einen booleschen Wert, der anzeigt, ob der jeweilige Knoten markiert ist.
- Der „*parentFT*“-Eintrag enthält den Verweis auf den Elternknoten bzw. den Wert *None*, falls es sich um einen Wurzelknoten handelt.

Ein Fibonacci-Heap *fibonacciHeap* und ein Fibonacci-Baum *fibonacciTree* kann man sich also (schemahaft) wie folgt definiert denken, wobei die Variablen *ft*, *ft0*, *ft1*, usw. Fibonacci-Bäume, *b* einen booleschen Wert und *i* einen Indexwert enthalten sollten.

$$
\begin{aligned}
\mathit{fibonacciHeap} &= \{\ \mathit{treesFH}: [\mathit{ft0}, \mathit{ft1}, \ldots]\ ,\ \mathit{minFH}: i\ \} \\
\mathit{fibonacciTree} &= \{\ \mathit{rootFT}: x\ ,\ \mathit{subtreesFT}: [\mathit{ft0}, \mathit{ft2}, \ldots], \\
&\qquad \mathit{markedFT}: b,\ \mathit{parentFT}: \mathit{ft}\}
\end{aligned}
$$

Die Schlüsselwerte *treesFH*, *minFH*, *rootFT* und *subtreesFT*, *markedFT* und *parentFT* der *dict*-Objekte *fibonacciHeap* und *fibonacciTree* können etwa folgendermaßen vordefiniert werden:

treesFH, *minFH*, *rootFT*, *subtreesFT*, *markedFT*, *parentFT* = *range*(6)

Der in Abbildung 4.6 gezeigte Fibonacci-Heap hätte somit die folgende Python-Repräsentation, wobei an der mit „ [...]“ markierten Stelle noch die Repräsentation der Teilbäume des zweiten Fibonacci-Baums einzusetzen wäre; die „ {..}“-Einträge stellen Verweise auf den Elternknoten dar.

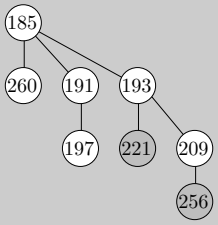
{*treesFH*: [{*rootFT*: 59, *subtreesFT*: [{*rootFT*: 88, *subtreesFT*: [ ], *markedFT*: *False*, *parentFT*: {..}}], *markedFT*: *False*, *parentFT*: *None*},
{*rootFT*: 30, *subtreesFT*: [...], *markedFT*: *False*, *parentFT*: {..}},
{*rootFT*: 40, *subtreesFT*: [ ], *markedFT*: *False*, *parentFT*: {..}},
{*rootFT*: 65, *subtreesFT*: [ ], *markedFT*: *False*, *parentFT*: {..}}],
minFH: 1}

Aufgabe 4.10

Vervollständigen Sie den oben gezeigten Wert so, dass er den in Abbildung 4.6 gezeigten Fibonacci-Heap vollständig repräsentiert.

Aufgabe 4.11

(a) Implementieren Sie eine Funktion *FT2str*(*ft*), die aus einem Fibonacci-Baum eine gut lesbare Stringform produziert. Schreiben Sie die Funktion so, dass etwa aus dem rechts dargestellten Fibonacci-Baum der folgende String produziert wird:




```
'185-(260 ; 191-197 ; 193-(#221 ; 209-#256))'
```

Die Liste der Teilbäume soll also immer in runden Klammern eingeschlossen sein; die einzelnen Teilbäume sollen durch `';'` getrennt sein; markierten Knoten soll ein `'#'` vorangestellt werden.

(b) Implementieren Sie eine Funktion *FH2str*(*fh*), die aus einem Fibonacci-Heap eine gut lesbare Stringform produziert; verwenden Sie hierzu die in der letzten Teilaufgabe beschriebene Funktion *FT2str*.

Aufgabe 4.12

Schreiben Sie eine Funktion *FH2List*, die die in einem Fibonacci-Heap enthaltenen Elemente als Liste zurückliefert.

Mittels des Zeigers auf den Fibonacci-Baum, der das minimale Element enthält, kann die Operation *getMin* offensichtlich in konstanter Zeit implementiert werden:

```
def getMinFH(fh):
  return fh[treesFH][fh[minFH]][rootFT]
```

4.3.3 Amortisierte Laufzeit und Potenzialfunktion

Die amortisierte Laufzeit einer bestimmten Operation bezieht sich nicht auf die einmalige Ausführung dieser Operation, sondern entweder auf die wiederholte Ausführung dieser Operation oder auf die wiederholte Ausführung der Operation in Kombination mit der Ausführung weiterer Operationen auf der Datenstruktur.

Eine Möglichkeit, die amortisierte Laufzeit verschiedener Operationen einer Datenstruktur in Kombination zu bestimmen, besteht in der Verwendung einer sog. Potential-Funktion. Wir verwenden hier die Potential-Funktion $\Phi(fh)$, wobei fh ein Fibonacci-Heap ist. Die Potential-Funktion ist folgendermaßen definiert:

$$\Phi(fh) = t(fh) + 2 \cdot m(fh) \tag{4.1}$$

Hierbei ist $t(fh)$ die Anzahl der Fibonacci-Bäume aus denen fh besteht, und $m(fh)$ bezeichnet die Anzahl der markierten Knoten in fh.

Die amortisierte Laufzeit einer Operation auf einem Fibonacci-Heap setzt sich nun zusammen aus der klassisch bestimmten Laufzeit plus der durch diese Operation bewirkten Potential-Änderung.

4.3.4 Verschmelzung

Das Verschmelzen zweier Fibonacci-Heaps *fh1* und *fh2* ist denkbar einfach: Die Listen *fh1*[*treesFH*] und *fh2*[*treesFH*] der Fibonacci-Bäume der beiden Heaps werden einfach vereinigt, und der Zeiger auf den Baum, der das minimale Element enthält, wird ggf. angepasst. Eventuell notwendige Restrukturierungsmaßnahmen werden auf „später“ verschoben. Durch wiederholte Ausführung von Verschmelzungsoperationen kann man so Fibonacci-Heaps erzeugen, die aus sehr vielen Fibonacci-Bäumen bestehen. Das Mitführen des Zeigers auf den Fibonacci-Baum der das minimale Element enthält, stellt jedoch immer ein effizientes Finden des minimalen Elements sicher. Listing 4.8 zeigt die Implementierung der Verschmelzungsoperation in Python.

```
1 def mergeFH(fh1,fh2):
2   if getMinFH(fh1) < getMinFH(fh2):
3     i = fh1[minFH]
```

```
4   else:
5     i = len(fh1[treesFH]) - 1 + fh2[minFH]
6   return { treesFH : fh1[treesFH] + fh2[treesFH] , minFH : i }
```

Listing 4.8: *Implementierung der Verschmelzung zweier Fibonacci-Heaps*

Die Fibonacci-Bäume des Ergebnis-Heaps sind einfach die Vereinigung der Fibonacci-Bäume von *fh1* mit den Fibonacci-Bäumen von *fh2*, also $fh1[treesFH] + fh2[treesFH]$. Der Zeiger auf das minimale Element des Ergebnis-Heaps ist entweder der in $fh1[minFH]$, falls das minimale Element von *fh1* kleiner ist als das minimale Element von *fh2* – falls also $getMinFH(fh1) < getMinFH(fh2)$; oder $fh2[minFH]$ zeigt auf den Heap, der das minimale Element des Fibonacci-Heaps enthält.

Aufgabe 4.13

Die in Listing 4.8 gezeigte Implementierung stellt eine nicht-destruktive Realisierung der Verschmelzungs-Operation dar. Implementieren Sie eine destruktive Version *mergeFHD*(*fh1*,*fh2*), die keinen „neuen“ Fibonacci-Heap als Rückgabewert erzeugt, sondern nichts zurückliefert und stattdessen den Parameter *fh1* (destruktiv) so verändert, dass dieser nach Ausführung von *mergeFHD* den Ergebnis-Heap enthält.

4.3.5 Einfügen

Um eine neues Element x in einen Fibonacci-Heap *fh* einzufügen, erzeugt man zunächst einen Fibonacci-Baum, der lediglich den Wert x enthält; dies geschieht in Listing 4.9 in Zeile 2 mittels der Funktion *makeFT*. Dieser einelementige Fibonacci-Baum wird dann der Liste der Fibonacci-Bäume von *fh* angefügt – dies geschieht in Zeile 3. In Zeile 5 wird der $fh[minFH]$ ggf. angepasst.

```
1 def insert(x,fh): # O(1)
2   ft = makeFT(x)
3   fh[treesFH].append(ft)
4   if getMinFH(fh) > x: # min−Pointer anpassen
5     fh[minFH] = len(fh[treesFH]) -1
```

Listing 4.9: *Implementierung der Einfügeoperation.*

Amortisierte Laufzeit. Die einfache Laufzeit der *insert*-Funktion ist in $O(1)$, denn sowohl Generierung eines einelementigen Fibonacci-Baums als auch das Anfügen und die erneute Minimumsbestimmung (die ja nur den bisherigen Minimumswert und den neu eingefügten Knoten in Betracht zieht) benötigen eine konstante Laufzeit. Die durch *insert*-Funktion bewirkte Potenzialveränderung ist

$$\Delta\Phi = 1 = O(1)$$

Die amortisierte Laufzeit ist somit in $O(1) + O(1) = O(1)$.

Aufgabe 4.14

Implementieren Sie die Funktion *makeFT*, die in Zeile 2 in Listing 4.9 benötigt wird.

Aufgabe 4.15

Die *insert*-Funktion aus Listing 4.9 ist destruktiv, d. h. sie verändert ihr Argument *fh* und liefert keinen Wert zurück. Implementieren Sie eine nicht-destruktive Variante dieser *insert*-Funktion, die ihr Argument *fh* nicht verändert und stattdessen einen neuen Fibonacci-Heap zurückliefert, in den das Element x eingefügt wurde.

4.3.6 Extraktion des Minimums

Die Extraktion des minimalen Elements eines Fibonacci-Heaps verläuft in zwei Phasen.

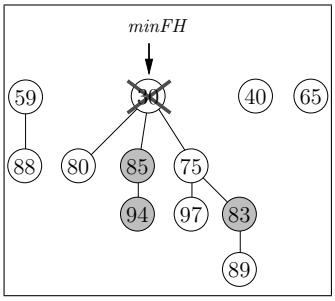
Phase 1: Das minimale Element des Fibonacci-Heap *fh* wird zunächst gefunden und gelöscht (dargestellt in Abbildung 4.7(a)); dadurch zerfällt der Fibonacci-Baum *ft*, dessen Wurzel dieses minimale Element war, in *len*(*ft* [*subtreesFT*]) Unterbäume. Diese Unterbäume werden zunächst dem Fibonacci-Heap *fh* angefügt (dargestellt in Abbildung 4.7(b)).

Phase 2: Nun werden die Bäume des Fibonacci-Heaps sukzessive so miteinander verschmolzen, dass am Ende keine zwei Bäume dieselbe Ordnung haben (dargestellt in Abbildungen 4.7(c) bis 4.7(h)).

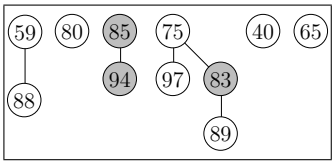
Listing 4.10 zeigt die Implementierung der Extraktion des minimalen Elements.

```
1  def extractMin(fh):
2    m = getMinFH(fh)
3    newsubtrees = fh[treesFH][fh[minFH]][subtreesFT]
4    del fh[treesFH][fh[minFH]]
5    ordTab = {}
6    for t in newsubtrees + fh[treesFH]:
7      o = len(t[subtreesFT])
8      while o in ordTab:
9        t = mergeFT(t,ordTab[o])
10       del ordTab[o]
11       o += 1
12     ordTab[o] = t
13   fh[treesFH] = ordTab.values()
14   fh[minFH] = min([(t[rootFT],i) for i,t in enumerate(fh[treesFH])])[1] #O(log n)
15   return m
```

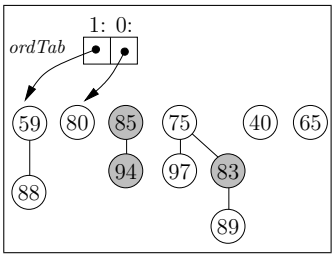
Listing 4.10: *Implementierung der Extraktion des minimalen Elements eines Fibonacci-Heaps.*



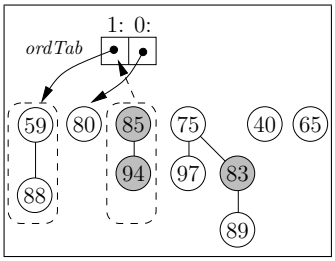
(a) Zunächst wird das minimale Element gelöscht, ...



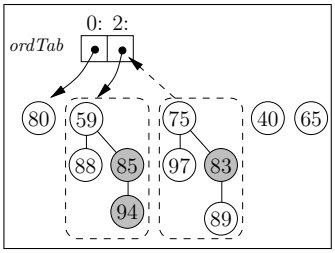
(b) ... die Unterbäume dem Fibonacci-Heap hinzugefügt, ...



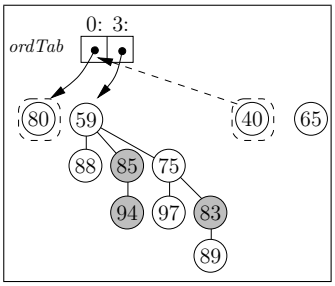
(c) ... dann die einzelnen Fibonacci-Bäume geordnet nach ihrem jeweiligen Rang in ein *dict*-Objekt *ordTab* gespeichert, ...



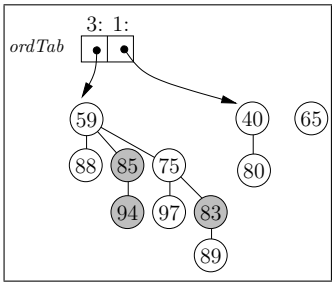
(d) ... und dabei Bäume gleicher Ordnung verschmolzen; bei Untersuchung des dritten Baumes (der die Ordnung 1 hat) wird – da *ordTab*[1] bereits einen Eintrag besitzt – erkannt, dass es schon einen Baum dieser Ordnung gibt, ...



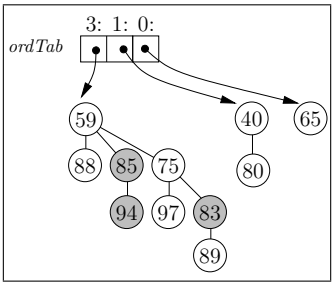
(e) ... und diese beiden Bäume werden miteinander verschmolzen, wodurch ein Fibonacci-Baum der Ordnung 2 entsteht.



(f) Auch der als Nächstes zu untersuchende Baum der Ordnung zwei wird mit dem bereits in *ordTab* existierenden (im letzten Schritt entstandenen) Baum der Ordnung 2 verschmolzen; es entsteht ein Baum der Ordnung 3.



(g) Der als Nächstes zu untersuchende Baum der Ordnung null wird mit dem bereits in *ordTab* befindlichen Baum der Ordnung null zu einem Baum der Ordnung eins verschmolzen.



(h) Schließlich befinden sich nur noch drei Fibonacci-Bäume unterschiedlicher Ordnung im Fibonacci-Heap.

Abb. 4.7: *Extraktion des Minimums eines Fibonacci-Heaps. Im Zuge dieser Operation werden auch Restrukturierungsmaßnahmen durchgeführt und Fibonacci-Bäume gleicher Ordnung zusammengefügt.*

Das minimale Element wird in Zeile 2 in der Variablen m gespeichert und am Ende in Zeile 15 zurückgeliefert. In der Variablen *newsubtrees* werden die Unterbäume des minimalen Elements gespeichert; in Zeile 4 wird der komplette Baum, der das minimale Element enthält, aus der Liste der Fibonacci-Bäume des Fibonacci-Heaps *fh* gelöscht. Die **for**-Schleife ab Zeile 6 durchläuft nun alle Fibonacci-Bäume (inklusive der durch die Löschung hinzugekommenen). Die Variable o enthält immer die Ordnung des Baumes der gerade bearbeitet wird. Gibt es bereits einen Eintrag „o“ in *ordTab* (d. h. gibt es unter den bisher untersuchten Bäumen bereits einen Fibonacci-Baum t der Ordnung o), so wird dieser mit dem aktuellen Baum verschmolzen (diese Verschmelzung wird in Zeile 9 durchgeführt) und der Eintrag „o“ aus *ordTab* gelöscht. Durch diese Verschmelzung entsteht ein Fibonacci-Baum der Ordnung $o+1$; o wird entsprechend um Eins erhöht. Die **while**-Schleife ab Zeile 8 prüft nun, ob es auch schon einen Baum der Ordnung $o+1$ in *ordTab* gibt, usw. Die **while**-Schleife bricht erst dann ab, wenn es keinen Eintrag „o“ in *ordTab* mehr gibt. Dann wird der aktuelle Fibonacci-Baum t in $ordTab[o]$ gespeichert und mit dem nächsten Baum fortgefahren.

Nach Abbruch der **for**-Schleife haben die in *ordTab* gespeicherten Fibonacci-Bäume (also *ordTab.values*()) alle unterschiedliche Ordnung; es sind genau die „neuen“ Fibonacci-Bäume, aus denen der Fibonacci-Heap nach Extraktion des minimalen Elements bestehen soll. Jetzt muss nur noch der Zeiger auf das minimale Element ggf. angepasst werden – dies geschieht in Zeile 14.

Aufgabe 4.16

Implementieren Sie die in Zeile 9 in Listing 4.10 benötigte Funktion *mergeFT*, die zwei Fibonacci-Bäume *ft1* und *ft2* so verschmilzt, dass die Heap-Bedingung erhalten bleibt.

Amortisierte Laufzeit. Sei $Ord(n)$ der maximale Grad eines Fibonacci-Baums in einem Fibonacci-Heap mit insgesamt n Knoten; nach Löschen des minimalen Elements werden dem Fibonacci-Heap also $O(Ord(n))$ Fibonacci-Bäume hinzugefügt. In Abschnitt 4.3.8 zeigen wir, dass $Ord(n) = O(\log n)$. Die einfache Laufzeit der in Listing 4.10 gezeigten Implementierung hängt entscheidend ab von der Anzahl der Schleifendurchläufe der **for**-Schleife ab Zeile 6; diese wird $t(fh) + O(Ord(n))$ mal durchlaufen. Innerhalb der **for**-Schleife werden Fibonacci-Bäume verschmolzen; aber auch hier gibt es höchstens $O(t(fh)) + O(Ord(n))$ Verschmelzungsoperationen. Somit ist die einfache Laufzeit in $O(t(fh)) + O(Ord(n))$.

Am Ende der Verschmelzungsphase gibt es $O(\log n)$ Fibonacci-Bäume (denn jeder Baum hat eine unterschiedliche Ordnung). An den Knotenmarkierungen ändert sich nichts. Es gilt also $\Delta\Phi = t(fh) - O(\log n)$. Insgesamt erhalten wir also eine amortisierte Laufzeit von

$$O(t(fh)) + O(Ord(n)) - (t(fh) - O(\log n)) = O(Ord(n)) + O(\log n) = O(\log n)$$

4.3.7 Erniedrigen eines Schlüsselwertes

Das Erniedrigen eines Schlüsselwertes ist vor allem deshalb eine wichtige Operation, weil man darüber in der Lage ist, einen Knoten aus einem Fibonacci-Heap zu löschen. Man braucht den Schlüsselwert eines Knotens lediglich auf $-\infty$ zu erniedrigen und anschließend den Knoten mit minimalem Schlüssel mittels der *minExtract*-Funktion aus Listing 4.10 aus dem Heap zu entfernen.

Jeder Knoten muss einen Zeiger auf seinen Elternknoten mitführen; nur so kann überprüft werden, ob durch das Erniedrigen die Heap-Bedingung verletzt wird und nur so können die im Folgenden beschriebenen Operationen durchgeführt werden. Wir erweitern hierfür die Repräsentation eines Fibonacci-Baums um einen Eintrag „*parentFT*“, der auf den Elternknoten eines Fibonacci-Teilbaums zeigt. Wir können uns also ab sofort einen Fibonacci-Baum (schemahaft) wie folgt definiert denken:

$$\begin{aligned} \mathit{fibonacciTree} = \{\ & \mathit{rootFT} : x\ ,\ \mathit{subtreesFT} :\ [\mathit{ft0}, \mathit{ft2}, \ldots\ ]\ , \\ & \mathit{markedFT} : b\ ,\ \mathit{parentFT} : \mathit{ft} \\ \} & \end{aligned}$$

Bei allen Operationen auf Fibonacci-Heaps muss man sicherstellen, dass die „Vorwärts“verzeigerung mittels *subtreesFT* mit der „Rückwärts“verzeigerung mittels *parentFT* übereinstimmt, dass also immer ft **in** $\mathit{ft}\,[\mathit{parentFT}]\,[\mathit{subtreesFT}]$ gilt.

Aufgabe 4.17

Schreiben Sie eine Python-Funktion *isConsistent*(*fh*), die überprüft, ob die Vorwärts- und Rückwärtsverzeigerung in allen Bäumen eines Fibonacci-Heaps *fh* konsistent ist.

Angenommen, wir wollen den Schlüsselwert der Wurzel eines Teilbaums *ft* eines Fibonacci-Heaps erniedrigen. Wird dadurch die Heap-Bedingung nicht verletzt, d. h. gilt weiterhin, dass $\mathit{ft}\,[\mathit{parentFT}]\,[\mathit{rootFT}] < \mathit{ft}[\mathit{rootFT}]$, so ist nichts weiter zu tun – ein Beispiel einer solchen Situation ist in Abbildung 4.8 dargestellt.

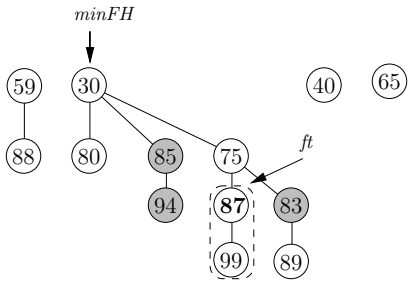


Abb. 4.8: *Die Min-Heap-Bedingung wird durch das Erniedrigen des Schlüsselwertes der Wurzel des Teilbaums ft (von 97 auf 87) nicht verletzt; in diesem Fall ist nichts weiter zu tun.*

Wird die Min-Heap-Bedingung durch Erniedrigen des Schlüsselwertes der Wurzel von *ft* verletzt, gilt also $\mathit{ft}\,[\mathit{parentFT}]\,[\mathit{rootFT}] > \mathit{ft}[\mathit{rootFT}]$ dann muss der Fibonacci-Heap

so modifiziert werden, dass die Min-Heap-Bedingung wiederhergestellt wird. Es sind zwei Fälle zu unterscheiden:

Fall 1: *Der Elternknoten von ft ist nicht markiert.* Abbildung 4.9 zeigt ein Beispiel einer solchen Situation.

In diesem Fall wird der Teilbaum *ft* einfach vom Elternkoten getrennt, der Elternknoten markiert und anschließend *ft* an Liste der Fibonacci-Bäume des Fibonacci-Heaps angehängt. Ist $ft\,[rootFT]$ kleiner als das bisher minimale Element, so muss der $fh\,[minFH]$ angepasst werden.

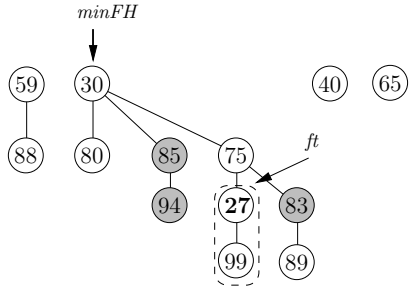
Fall 2: *Der Elternknoten von ft ist bereits markiert*, d. h. es gilt $ft\,[parentFT]\,[markedFT]$. Abbildung 4.10 zeigt ein Beispiel einer solchen Situation.
In diesem Fall wird der Teilbaum *ft* ebenfalls vom Elternknoten $ft\,[parentFT]$ getrennt und an die Liste der Fibonacci-Bäume des Fibonacci-Heaps angefügt. Anschließend wird auch der Elternknoten $ft\,[parentFT]$ von seinem Elternknoten getrennt und an die Liste der Fibonacci-Bäume des Fibonacci-Heaps hinzugefügt. Ist der Elternknoten des Elternknotens nicht markiert – d. h. gilt $ft\,[parentFT]\,[parentFT]\,[markedFT]$ – so wird dieser markiert. Ist auch dieser bereits markiert, so wird auch der Elternknoten des Elternknotens von dessen Elternknoten getrennt, usw..

Listing 4.11 zeigt eine Implementierung der Erniedrigung eines Schlüsselwertes um *delta* eines durch *pos* spezifizierten Knotens eines Fibonacci-Heaps *fh*. Die Positionsangabe *pos* ist ein Tupel. Die erste Komponente, also $pos[0]$ spezifiziert den Fibonacci-Baum des Fibonacci-Heaps *fh*, in dem sich der zu erniedrigende Knoten befindet. Die zweite Komponente, also $pos[1]$ enthält eine Liste von Zahlen, die einen von der Wurzel beginnenden Pfad spezifizieren. Die Liste „ [ ]“ (also der leere Pfad) beispielsweise spezifiziert die Wurzel des Fibonacci-Baums. Die Liste „ [1,0,2]“ beispielsweise spezifiziert von der Wurzel ausgehend den 1-ten Teilbaum, davon den 0-ten Teilbaum und davon wiederum den 2-ten Teilbaum, also $ft\,[subtreesFT]\,[1]\,[subtreesFT]\,[0]\,[subtreesFT]\,[1]$, wobei *ft* der $pos[0]$-te Teilbaum des Fibonacci-Heaps *fh* sei. Genau genommen spezifiziert *pos* einen Teilbaum, der sich nach Ausführung der **for**-Schleife in Zeile 4 in der Variablen *ft* befindet. In Zeile 6 wird die Wurzel dieses Teilbaums, also $ft\,[rootFT]$ und den Betrag *delta* erniedrigt.

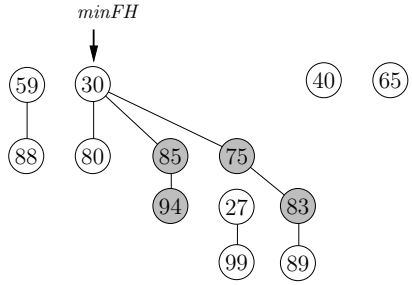
Die Struktur des Fibonacci-Heaps muss nun genau dann angepasst werden, wenn die Heap-Bedingung durch dieses Erniedrigen verletzt wird; dies prüft die **if**-Abfrage in Zeile 7.

Muss die Struktur angepasst werden, so hängt die „**while** *True*“-Schleife ab Zeile 8 solange den Knoten, den Elternknoten, den Eltern-Elternknoten usw. vom aktuellen Baum ab und fügt diesen Knoten als weiteren Fibonacci-Baum in $fh\,[treesFH]$ ein, bis entweder ein nicht-markierter Knoten gefunden wird – dieser Fall wird in Zeile 21 und 22 abgehandelt, oder bis ein Wurzelknoten erreicht wird – dieser Fall wird in Zeile 11 und 12 abgehandelt.

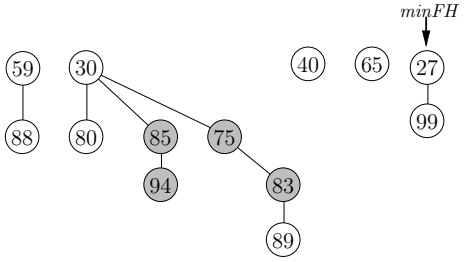
Das Aushängen eines Teilbaums *ft* geschieht folgendermaßen: Zunächst wird eine eventuelle Markierung von *ft* gelöscht (Zeile 14), da *ft* zu einem neuen Baum des Fibonacci-Heaps gemacht wird und die Wurzeln der Bäume grundsätzlich nicht markiert sein



(a) Der Schlüsselwert an der Wurzel des Teilbaums *ft* wird von 97 auf 27 erniedrigt. Dadurch wird die Min-Heap-Bedingung verletzt. Diese muss wiederhergestellt werden.



(b) Da der Elternknoten (mit Schlüsselwert 75) nicht markiert ist, kann der Teilbaum *ft* einfach vom Elternknoten getrennt werden. Der Elternknoten wird danach markiert.



(c) Der abgetrennte Teilbaum *ft* wird nun einfach an die Teilbaumliste des Fibonacci-Heaps angefügt. Der Zeiger auf das minimale Element muss danach nach einem Vergleich mit dem bisherigen Minimum *getMinFH*(*fh*) ggf. angepasst werden.

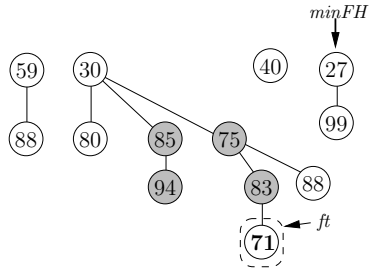
Abb. 4.9: *Erniedrigen eines Schlüsselwertes, das die Min-Heap-Bedingung verletzt. Es handelt sich hier um den einfacheren Fall: Der Elternknoten des Knotens, dessen Schlüsselwert erniedrigt werden soll, ist nicht markiert.*

dürfen. In Zeile 15 wird der Rückwärtszeiger von *ft* gelöscht, in Zeile 16 wird *ft* aus der Liste der Teilbäume seines Elternknotens gelöscht. In Zeile 17 wird *ft* der Baumliste der Fibonacci-Heaps hinzugefügt. In Zeile 19 wird (falls erforderlich) der Zeiger auf den Baum angepasst, der das minimale Element des Fibonacci-Heaps enthält.

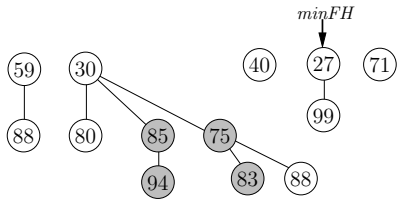
Es gibt noch einen Sonderfall, der nicht vergessen werden darf: Soll der Schlüsselwert der Wurzel eines Fibonacci-Baums erniedrigt werden, gilt also $pos[1]==[]$, und ist dieser neue Schlüsselwert kleiner als das bisherige Minimum, so muss der Zeiger $fh[minFH]$ angepasst werden.

Aufgabe 4.18

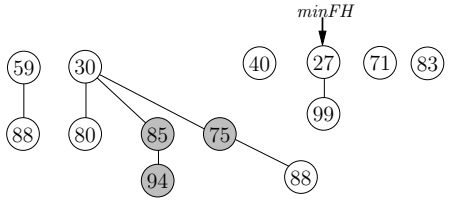
Erstellen Sie eine Funktion *allPaths*(*fh*), die die Liste aller gültigen Pfade eines Fibonacci-Heaps erzeugt – und zwar so, dass jeder dieser Pfade als möglicher zweiter Parameter der in Listing 4.11 gezeigten Funktion *decKey* dienen könnte.



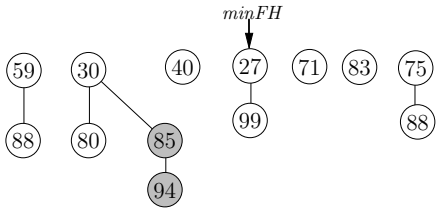
(a) Durch Erniedrigen von $ft\,[rootFT]$ wird die Heap-Bedingung verletzt.



(b) Der Teilbaum wird vom Elternknoten getrennt und der Liste der Fibonacci-Bäume hinzugefügt.



(c) Da der Elternknoten (mit Schlüsselwert 83) markiert ist, wird auch dieser von seinem Elternknoten getrennt und der Liste der Fibonacci-Bäume hinzugefügt.



(d) Auch dessen Elternknoten (mit Schlüsselwert 75) ist markiert und darum wird auch dieser von seinem Elternknoten getrennt und der Liste der Fibonacci-Bäume hinzugefügt.

Abb. 4.10: *Erniedrigen eines Schlüsselwertes, das die Min-Heap-Bedingung verletzt. Der Elternknoten des Knotens, dessen Schlüsselwert erniedrigt werden soll. ist bereits markiert.*

Aufgabe 4.19

Verwenden Sie die eben implementierte Funktion *allPaths*, um ein zufällig ausgewähltes Element eines Fibonacci-Heaps um einen bestimmten Betrag zu erniedrigen – dies ist etwa zu Testzwecken hilfreich; auch der am Anfang dieses Abschnitts gezeigte Fibonacci-Heap wurde (neben zufälligen Einfügeoperationen und Minimumsextraktionen) so erzeugt.

Amortisierte Laufzeit. Angenommen *fh* sei der Fibonacci-Heap vor Erniedrigung des Schlüsselwerts und fh' der Fibonacci-Heap nach Erniedrigung des Schlüsselwerts. Nehmen wir an, die **while**-Schleife in Listing 4.11 (ab Zeile 8) wird c mal durchlaufen, d. h. es werden c Knoten von ihren jeweiligen Elternknoten getrennt. Dann gilt, dass

- $t(fh') = t(fh) + c$ denn jeder der c Knoten wird an die Liste der Fibonacci-Bäume von *fh* angehängt.
- $m(fh') = m(fh) - (c - 2)$ denn die Markierung jedes Knotens der von seinem Elternknoten getrennt wird, wird gelöscht – denn dieser Knoten wird ja zur Wurzel eines Fibonacci-Baums und alle Wurzeln müssen grundsätzlich unmarkiert sein. Auf diese Weise wird die Markierung von $c - 1$ gelöscht. Der Elternknoten des

```
def decKey(fh,pos,delta):
  # pos = (x,[x0,x1,x2,  ...])
  ft = fh[treesFH][pos[0]]
  for x in pos[1]:
    ft = ft[subtreesFT][x]
  ft[rootFT] -= delta
  if ft[parentFT] and ft[parentFT][rootFT] > ft[rootFT]:
    while True:
      ftParent = ft[parentFT]
      if not ftParent: # ft ist Wurzel
        ft[markedFT] = False
        break
      else:
        ft[markedFT] = False
        ft[parentFT] = None # ft trennen
        ftParent[subtreesFT].remove(ft)
        fh[treesFH].append(ft) # ... ft wird neue Wurzel
        if ft[rootFT]<getMinFH(fh):
          fh[minFH] = len(fh[treesFH]) -1
        if not ftParent[markedFT]:
          if ftParent[parentFT]≠None: ftParent[markedFT] = True
          break
        ft = ftParent # weiter mit Elternknoten
    elif ft[rootFT]<getMinFH(fh): fh[minFH] = pos[0]
```

Listing 4.11: *Implementierung der Erniedrigung des Schlüsselwertes an der Wurzel des (Teil)-Fibonacci-Baums.*

zuletzt getrennten Knotens wird markiert und darum verändert sich die Zahl der markierten Knoten um $c - 2$.

Somit ergibt sich folgende Potenzialveränderung:

$$\begin{aligned}\Delta\Phi &= t(fh) + 2 \cdot m(fh) - (t(fh) + c + 2(m(fh) - (c-2))) \\ &= 4 - c\end{aligned}$$

Insgesamt ergibt sich also eine amortisierte Laufzeit von

$$O(c) + 4 - c = O(1)$$

An diesem Punkt sehen wir klarer, warum die Anzahl der markierten Knoten in der Potenzialfunktion mit dem Faktor „2“ auftaucht.

- Der eine markierte Knoten verrechnet sich mit dem Trennen des Knotens von seinem Elternknoten und dem nachfolgenden Löschen der Markierung.

- Der andere markierte Knoten verrechnet sich mit dem Potenzialanstieg aufgrund des zusätzlich eingefügten Fibonacci-Baums.

Aufgabe 4.20

Die in Zeile 18 in Listing 4.11 durchgeführt Überprüfung, ob $ft\,[rootFT]$ kleiner ist als das bisherige Minimum des Fibonacci-Heaps braucht eigentlich nicht in jedem Durchlauf der äußeren **while**-Schleife durchgeführt werden. Passen Sie die in Listing 4.11 gezeigte Implementierung so an, dass diese Überprüfung nur einmal stattfindet.

4.3.8 Maximale Ordnung eines Fibonacci-Baums

Woher Fibonacci-Heaps ihren Namen haben, sehen wir in diesem Abschnitt. Es bleibt noch zu zeigen, dass die maximale Ordnung – im Folgenden als $Ord(n)$ bezeichnet – eines in einem n-elementigen Fibonacci-Heap befindlichen Fibonacci-Baums in $O(\log n)$ ist. Wir werden im Speziellen zeigen, dass gilt:

$$Ord(N) \leq \log_\phi n, \quad \text{mit} \quad \phi = \frac{1+\sqrt{5}}{2}$$

Wir bezeichnen als $s(ft)$ die Anzahl der im Fibonacci-Baum ft befindlichen Elemente. Sei o die Ordnung dieses Fibonacci-Baums – also $o = len(ft\,[subtreesFT])$. Wir zeigen, dass

$$s(ft) \geq F_{o+2}$$

gelten muss. Hierbei ist F_k ist die k-te Fibonacci-Zahl[1]. Um diese Aussage zu zeigen, verwenden wir vollständige Induktion[2] über die Höhe h von ft:

$h = 0$: In diesem Fall ist $s(ft) = 1 \geq F_2$.

$< h \to h$: Wir nehmen also an, ft besitzt eine Höhe $h > 0$ und muss damit eine Ordnung $o > 0$ haben. Seien $ft_0, ft_1, \ldots, ft_{o-1}$ die Teilbäume von ft, geordnet nach dem Zeitpunkt zu dem diese ft hinzugefügt wurden. Sei $o_i = len(ft_i\,[subtreesFT])$ – d. h. o_i ist die Ordnung von ft_i. Man kann zeigen, dass $o_i \geq i - 1$:

Z. z. $o_i \geq i - 1$: Als ft_i zu ft hinzugefügt wurde, waren also $ft_0, \ldots, ft_{i-1}$ bereits Teilbäume von ft und ft hatte somit eine Ordnung von i. Da Bäume nur dann verschmolzen werden, wenn sie gleiche Ordnung besitzen, muss ft_i auch eine Ordnung von i gehabt haben. Seit dem Zeitpunkt dieser Verschmelzung wurde höchstens ein Teilbaum von ft_i entfernt (aufgrund der Handhabung von Markierungen ist es nicht möglich, mehr als einen Teilbaum zu entfernen); die momentane Ordnung von ft_i ist also $\geq i - 1$.

[1] Siehe auch Anhang B.2 auf Seite 307
[2] Siehe auch Anhang B.1.4 auf Seite 306

Da die Höhen der ft_i kleiner sind als die Höhe h von ft, können wir auf diese die Induktionshypothese anwenden und annehmen, dass $s(ft_i) \geq F_{o_i+2} = F_{i+1}$. Der Induktionsschritt lässt sich dann folgendermaßen zeigen:

$$\begin{array}{rcl} s(ft) & = & \overbrace{1}^{\text{Wurzel von } ft} + s(ft_0) + s(ft_1) + \ldots + s(ft_{o-1}) \\ & \geq & 1 + F_{o_0+2} + F_{o_1+2} + \ldots + F_{o_{o-1}+2} \\ & = & 1 + F_1 + F_2 + \ldots + F_o \\ & \overset{\text{Satz 2}}{=} & F_{o+2} \end{array}$$

Nach Satz 3 (aus Anhang B.2) gilt, dass $F_{o+2} \geq \varphi^o$ und damit $s(ft) \geq \varphi^o$, wobei – wir erinnern uns – o die maximale Ordnung eines Knotens in ft bezeichnet. Aufgelöst nach o gilt somit

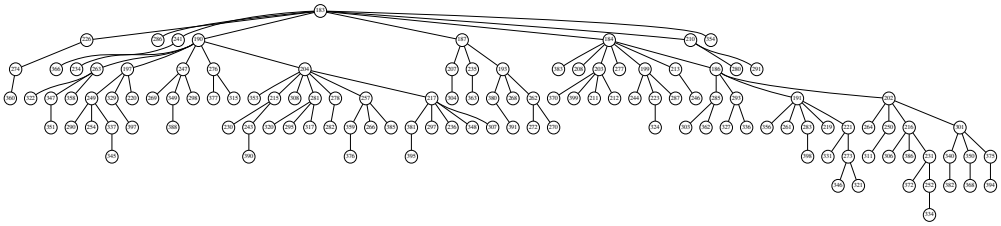
$$o \leq \log_\varphi s(ft)$$

oder anders ausgedrückt

$$Ord(n) \leq \log_\varphi(n)$$

wobei $Ord(n)$ die maximale Ordnung eines Fibonacci-Baums mit n Elementen bezeichnet.

4.4 Pairing-Heaps



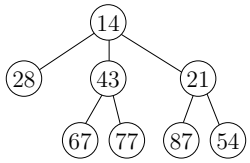
Pairing-Heaps wurden ursprünglich von Tarjan, Fredman, Sedgewick und Sleator [8] als eine einfachere Variante von Fibonacci-Heaps vorgeschlagen. Sie sind einfacher zu implementieren als Binomial-Heaps und Fibonacci-Heaps. Noch dazu zeigen Pairing-Heaps in den meisten praktischen Anwendungen eine hervorragende Performance. Experimente zeigen, dass Pairing Heaps etwa verwendet in Prim's Algorithmus zur Berechnung des minimalen Spannbaums, tatsächlich schneller zu sein scheinen, als *alle* anderen bekannten Alternativen. Trotz ihrer einfachen Funktionsweise stellt sich eine Laufzeitanalyse als äußert schwierig heraus: Bis heute ist eine abschließende Laufzeitanalyse noch ein offenes Problem der Informatik.

4.4.1 Struktur und Repräsentation in Python

Ein Pairing-Heap ist entweder leer oder besteht aus einem Wurzelelement zusammen mit einer Liste von Teilbäumen; jeder Knoten muss zusätzlich die (Min-)Heap-Bedingung erfüllen, d. h. sein Schlüsselwert muss kleiner sein als die Schlüsselwerte seiner Teilbäume.

Eine solche Struktur kann in Python am einfachsten als Tupel repräsentiert werden[3].

Der folgende Python-Ausdruck repräsentiert hierbei etwa den rechts davon abgebildeten Pairing-Heap.




```
(14, [(28, [] ),  \
      (43, [(67, [] ),(77, [] )] ),  \
      (21, [(87, [] ),(54, [] )] ))
```

Zusätzlich gehen wir im Folgenden davon aus, dass ein leerer Heap durch den Wert *None* repräsentiert ist.

Für die Lesbarkeit der in diesem Abschnitt präsentierten Algorithmen ist es günstig, wenn wir definieren:

$rootPH,\ subtreesPH = 0{,}1$

Um auf das Wurzelelement eines Pairing-Heaps *ph* zuzugreifen, schreiben wir im Folgenden statt $ph[0]$ der Lesbarkeit halber besser $ph[rootPH]$. Um auf die Liste der Teilbäume zuzugreifen, schreiben wir im Folgenden statt $ph[1]$ besser $ph[subtreesPH]$.

Aufgabe 4.21

Schreiben Sie eine Funktion *ph2str*, die einen Pairing-Heap als Argument übergeben bekommt und eine gut lesbare String-Repräsentation dieses Pairing-Heaps zurückliefert. Die String-Repräsentation des Pairing-Heaps aus Abbildung 4.11(a) sollte hierbei beispielsweise folgende Form haben:

```
'26-[48-49-[99,95],74,50-61,73,31-[39,69]]'
```

Die Teilbaumlisten sollten also – vorausgesetzt sie bestehen aus mehr als einem Baum – in eckige Klammern eingeschlossen werden; das Wurzelelement sollte mit einem `'-'` von seiner Teilbaumliste getrennt sein.

4.4.2 Einfache Operationen auf Pairing-Heaps

Die Implementierung der meisten Operationen auf Pairing-Heaps ist sehr simpel, insbesondere verglichen mit der Implementierung der entsprechenden Operationen auf Binomial-Heaps oder Fibonacci-Heaps und sogar auf binären Heaps.

Zunächst befindet sich das minimale Element immer an der Wurzel des Pairing-Heaps. Entsprechend einfach ist die Implementierung der *getMin*-Funktion auf Pairing-Heaps:

```
1 def getMin(ph):
2     if ph: return ph[rootPH]
```

Durch die **if**-Abfrage wird hier sichergestellt, dass kein Laufzeitfehler entsteht, wenn *getMin* auf einen leeren Heap angewendet wird.

[3]Selbstverständlich ist auch eine Repräsentation über eine Klasse möglich; siehe Aufgabe 4.23.

Zwei Pairing-Heaps werden verschmolzen, indem einfach der Heap mit dem größeren Wurzelelement als neuer Teilbaum unter den Heap mit dem kleineren Wurzelelement gehängt wird. Listing 4.12 zeigt eine Implementierung der Verschmelzungsoperation.

```
1 def merge(ph1,ph2):
2   if not ph1: return ph2
3   if not ph2: return ph1
4   if ph1<ph2:
5     return (ph1[rootPH], ph1[subtreesPH] + [ph2])
6   else:
7     return (ph2[rootPH], ph2[subtreesPH] + [ph1])
```

Listing 4.12: *Verschmelzung zweier Pairing-Heaps*

Aufgabe 4.22

Die oben gezeigte Implementierung der *merge*-Operation ist nicht-destruktiv implementiert: Die übergebenen Parameterwerte werden (durch Zuweisungen bzw. destruktive Listenoperationen) nicht verändert; als Rückgabewert wird eine neuer Pairing-Heap konstruiert.

Erstellen Sie nun eine destruktive Implementierung, in dem der *ph1*-Parameter destruktiv so verändert wird, dass er nach Ausführung der Funktion das gewünschte Ergebnis enthält. Erklären Sie, warum und wie sie die oben beschriebene Repräsentation von Pairing-Heaps hierfür anpassen müssen.

4.4.3 Extraktion des Minimums

Tatsächlich stellt die Extraktion des Minimums die einzige nicht-triviale Operation auf Pairing-Heaps dar. Durch das Löschen des Wurzelelements $ph[rootPH]$ entstehen $len(ph[subtreesPH])$ „freie“ Bäume. Es gibt mehrere sinnvolle Möglichkeiten, in welcher Weise diese Bäume wieder zu einem Pairing-Heap zusammengefügt werden. Eine häufig verwendete Möglichkeit wollen wir hier vorstellen: das paarweise Verschmelzen der „freien“ Bäume von links nach rechts in $ph[subtreesPH]$ und das anschließende Verschmelzen der so entstandenen Bäume von rechts nach links.

Listing 4.13 zeigt eine funktionale (d. h. nicht-destruktive) Implementierung der Minimumsextraktion. Die Funktion *extractMinND* verändert also ihr Argument *ph* nicht sondern konstruiert stattdessen mittels der Funktion *pairmerge* einen neuen Pairing-Heap der durch Extraktion des minimalen Elements entsteht und liefert diesen als Ergebnis zurück. Die Funktion *extractMinND* liefert also ein Tupel zurück dessen erste Komponente das minimale Element ist und dessen zweite Komponente der neue Pairing-Heap ist der durch Löschen des minimalen Elements entsteht.

Die erste **if**-Abfrage in Zeile 2 deckt den einfachsten Fall ab: Ein leerer Heap liefert das Tupel (*None*,*None*) zurück, gibt also kein minimales Element zurück und liefert wiederum den leeren Heap. Zeile 3 behandelt einen weiteren Sonderfall, den einelementigen

```
def extractMinND(ph):
    if not ph: return (None,None)
    if not ph[subtreesPH]: return ph[rootPH], None
    return ph[rootPH],pairmerge(ph[subtreesPH])

def pairmerge(phs):
    if len(phs)==0: return None
    if len(phs)==1: return phs[0]
    return merge(merge(phs[0],phs[1]),pairmerge(phs[2:]))
```

Listing 4.13: *Implementierung der Minimums-Extraktion.*

Heap. Hier wird der Wert an der Wurzel (also $ph[rootPH]$) und der leere Heap zurückgeliefert. Andernfalls werden die Teilbäume von *ph* mittels der Funktion *pairmerge* zu einem neuen Heap verschmolzen.

Die Implementierung von *pairmerge* ist rekursiv, rein funktional (d. h. verwendet keine Zuweisungen) und erstaunlich einfach. Besteht die der Funktion *pairmerge* übergebene Liste von Pairing-Heaps *phs* aus nur einem Baum, so wird dieser eine Baum einfach zurückgeliefert – dies ist der Rekursionsabbruch. Andernfalls werden die ersten beiden Pairing-Heaps $phs[0]$ und $phs[1]$ verschmolzen und der resultierende Pairing-Heap mit dem mittels *pairmerge* auf den restlichen Pairing-Heaps erstellten Heap verschmolzen. Der rekursive Abstieg führt die paarweisen Verschmelzungen von links nach rechts durch. Der darauf folgende rekursive Aufstieg führt die abschließenden Verschmelzungen von rechts nach links durch.

Abbildung 4.11 veranschaulicht den Ablauf einer Minimumsextraktion anhand eines Beispiel-Heaps. Man kann zeigen, dass die Minimums-Extraktion $O(\log n)$ Schritte benötigt; die Herleitung dieser Tatsache ist jedoch nicht trivial, und wir verzichten hier auf eine entsprechende Darstellung.

Aufgabe 4.23

Repräsentieren Sie einen Pairing-Heap durch eine Klasse *PairingHeap* und implementieren Sie die beschriebenen Funktion als Methoden dieser Klasse.

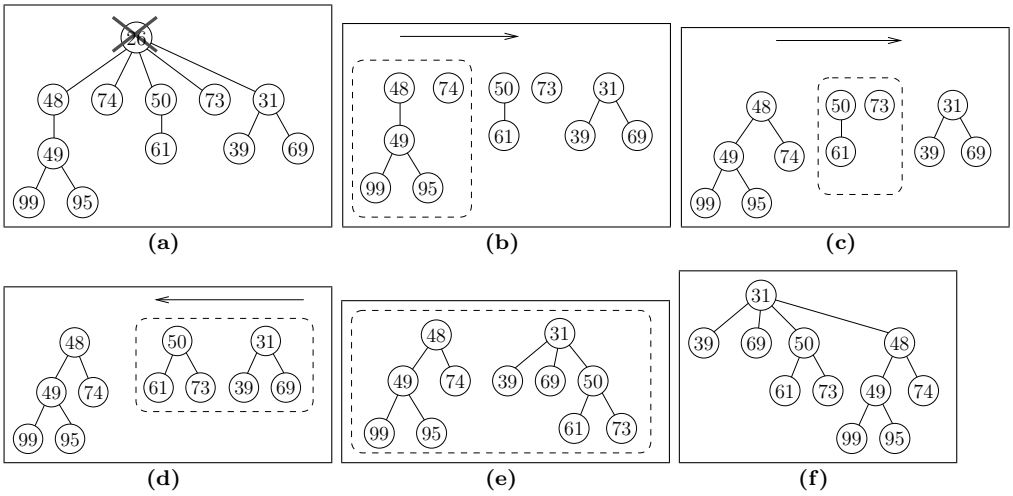


Abb. 4.11: *Darstellung der Funktionsweise der Minimumsextraktion anhand eines Beispiel-Heaps. Nach Löschen des Wurzelelements (Abbildung 4.11(a)) entstehen im Beispiel 5 lose Bäume; diese werden zunächst paarweise von links nach rechts verschmolzen (Abbildungen 4.11(b) und 4.11(c)) und anschließend die so entstandenen Bäume von rechts nach links verschmolzen (Abbildungen 4.11(d) und 4.11(e)). Aufgrund der Funktionsweise der Verschmelzungs-Operation erfüllen die Knoten des so entstandenen Baums (siehe Abbildung 4.11(f)) wieder die Min-Heap-Bedingung.*