Kapitel 6

Manipulation von Mengen

Datenstrukturen zur Repräsentation einer Kollektion von Datenmengen, auf der gewisse Operationen ausgeführt werden sollen, wurden erstmals von Aho, Hopcroft und Ullman [3] systematisch behandelt. Die abstrakte Behandlung solcher Mengenmanipulationsprobleme erleichtert in vielen Fällen den Entwurf und die Analyse von Algorithmen aus verschiedenen Anwendungsgebieten. Man formuliert Algorithmen zunächst auf hohem Niveau unter Rückgriff auf Strukturen und Operationen zur Manipulation von Mengen, die in herkömmlichen Programmiersprachen üblicherweise nicht vorkommen. In einem zweiten Schritt überlegt man sich dann, wie die Kollektion von Datenmengen und die benötigten Operationen implementiert, also programmtechnisch realisiert werden können. Besonders erfolgreich war dieser Ansatz bei der Verbesserung und Neuentwicklung von Algorithmen auf Graphen. Beispiele sind Verfahren zur Berechnung spannender Bäume, kürzester Pfade und maximaler Flüsse, vgl. hierzu auch das Kapitel 8 und die Monographie von Tarjan [180].

Einen wichtigen Spezialfall eines Mengenmanipulationsproblems, das sogenannte Wörterbuchproblem, haben wir im Kapitel 1 und besonders im Kapitel 5 bereits ausführlich behandelt. Dort ging es um die Frage, wie eine Menge von Schlüsseln abgespeichert werden soll, damit die Operationen Suchen (Zugriff), Einfügen und Entfernen von Schlüsseln möglichst effizient ausführbar sind. Wir werden sehen, daß die im Kapitel 5 zur Lösung des Wörterbuchproblems benutzten Bäume auch für viele andere Mengenmanipulationsprobleme benutzt werden können.

Wir gehen in diesem Kapitel davon aus, daß die Datenmengen stets Mengen ganzzahliger Schlüssel sind, obwohl die Schlüssel in den meisten Anwendungen lediglich zur eindeutigen Identifizierung der „eigentlichen“ Information dienen.

Neben dem bereits genannten Wörterbuchproblem sind zwei Spezialfälle des Mengenmanipulationsproblems in der Literatur besonders ausführlich behandelt worden: Vorrangswarteschlangen (Priority Queues), die im Abschnitt 6.1 behandelt werden, und Union-Find-Strukturen, die im Abschnitt 6.2 diskutiert werden. Im Abschnitt 6.3 geben wir einen allgemeinen Rahmen zur Behandlung von Mengenmanipulationsproblemen an und zeigen Möglichkeiten zur Lösung solcher Probleme mit Hilfe verschiedener Klassen von Bäumen auf.

6.1 Vorrangswarteschlangen

Als *Vorrangswarteschlange* (englisch: *priority queue*) bezeichnet man eine Datenstruktur zur Speicherung einer Menge von Elementen, für die eine Ordnung (Prioritätsordnung) definiert ist, so daß folgende Operationen ausführbar sind: *Initialisieren* (der leeren Struktur), *Einfügen* eines Elementes (Insert), *Minimum Suchen* (Access Min), *Minimum Entfernen* (Delete Min). Wir nehmen der Einfachheit halber an, daß die Elemente ganzzahlige Schlüssel sind und die Prioritätsordnung die übliche Anordnung ganzer Zahlen ist.

Der Begriff Vorrangswarteschlange erinnert an offensichtliche Anwendungen für solche Strukturen. Man denke an Kunden, die vor Kassen warten, an Aufträge, die auf ihre Ausführung warten, an Akten, die im Bearbeitungsstapel eines Sachbearbeiters auf ihre Erledigung warten. Die Prioritätsordnung ist hier durch den Ankunftszeitpunkt oder die Dringlichkeit festgelegt; die zeitlich ersten (frühesten) oder dringendsten Ereignisse haben Vorrang vor anderen.

Der Begriff Priority Queue wurde von Knuth [89] eingeführt. Andere Autoren, z.B. [3] und [180], benutzen den Begriff *Heap* (Halde), den wir in Abschnitt 2.3 für eine spezielle Datenstruktur reserviert haben, die im Sortierverfahren Heapsort verwendet wurde. Selbstverständlich sind Heaps eine mögliche Implementation für Priority Queues. Ein Heap mit N Schlüsseln erlaubt das *Einfügen* eines neuen Elementes und das *Entfernen des Minimums* in $O(\log N)$ Schritten; da das Minimum stets am Anfang des Heaps steht, kann die Operation *Access Min* in konstanter Zeit ausgeführt werden. In Abschnitt 2.3 haben wir nicht das Minimum, sondern das Maximum aller Schlüssel am Anfang des Heaps gespeichert. Dies gibt einfach eine andere Prioritätsordnung über den Schlüsseln ($>$ statt $<$) wieder und kann offensichtlich ebenso leicht im Heap realisiert werden.

Neben den genannten Operationen wird häufig verlangt, daß für Priority Queues weitere Operationen ausführbar sind und zwar: das *Entfernen beliebiger Elemente*, also nicht nur des Minimums, und das *Herabsetzen* eines Schlüssels um einen vorgegebenen Wert (*Decrease-Key*-Operation). Hierbei wird allerdings in der Regel vorausgesetzt, daß die Position des Schlüssels, den man entfernen möchte oder dessen Wert man erniedrigen möchte, bekannt ist; d.h. den Aufwand, den betreffenden Schlüssel in der Priority Queue zu finden, läßt man bei diesen Operationen außer Betracht. Schließlich verlangt man häufig, daß das *Zusammenfügen* (*Verschmelzen*) zweier elementfremder Priority Queues (Operation *Meld* oder *Merge*) möglich ist.

Zwar lassen sich alle diese Operationen auch für die im Abschnitt 2.3 eingeführten Heaps ausführen. Weil Heaps aber eine sehr starre Struktur haben, ist es besonders schwierig, zwei Heaps schnell zu einem neuen zusammenzufügen. Zwei offensichtliche Möglichkeiten sind jedoch die folgenden. Erstens kann man sämtliche Elemente des kleineren Heaps in den größeren einfügen. Das ist in $O(k \log(N+k))$ Schritten ausführbar, wobei k die Anzahl der Elemente des kleineren Heaps und N die des größeren Heaps ist. Die zweite Möglichkeit ist, die vorhandenen Strukturen aufzulösen und einen neuen Heap mit allen $N+k$ Elementen aufzubauen. Der Aufbau ist in $O(N+k)$ Schritten durchführbar.

In Abschnitt 6.1.1 geben wir zunächst ein Beispiel für die Verwendung von Priority Queues an. In Abschnitt 6.1.2 zeigen wir dann, wie man Priority Queues mit Hilfe bereits bekannter Strukturen implementieren kann. Schließlich führen wir in den folgenden Abschnitten eine Reihe neuer Strukturen ein, die zeigen, daß Priority Queues sehr einfach und effizient implementiert werden können.

Man beachte, daß in keinem Fall die Operation des Suchens eines Schlüssels besonders unterstützt wird.

6.1.1 Dijkstras Algorithmus zur Berechnung kürzester Wege

Als einziges Beispiel eines Algorithmus, der mit Hilfe von Operationen für Priority Queues bequem formuliert werden kann, wollen wir ein Verfahren zur Berechnung kürzester Wege in gerichteten Graphen diskutieren. Eine ausführlichere Behandlung von Algorithmen für Graphen erfolgt im Kapitel 8.

Gegeben sei ein gerichteter Graph G mit Knotenmenge V und Kantenmenge E. Wir verzichten hier auf eine formale Definition von Graphen und von Begriffen im Zusammenhang mit Graphen; der interessierte Leser sei auf Kapitel 8 verwiesen. Man stelle sich einfach ein Netz von Einbahnstraßen zwischen Orten vor. Die Orte bilden die Knotenmenge V, und die Einbahnstraßen sind die gerichteten Kanten zwischen den Orten. Jede Kante e hat eine nichtnegative Länge $l(e)$.

Wir wollen ein Problem lösen, das in der Literatur unter dem Namen *Single-source-shortest-paths-Problem* (oder *one-to-all shortest paths*, vgl. [180]) bekannt ist. Gegeben sei ein Knoten (ein Startort) s. Die Aufgabe besteht darin, für jeden Knoten $v \in V$ des Graphen G den kürzesten Weg von s nach v zu bestimmen. Wir begnügen uns damit, nicht den Weg selbst, sondern nur seine Länge zu bestimmen. Wir setzen der Einfachheit halber voraus, daß jeder Knoten $v \in V$ auch durch wenigstens einen Weg von s aus erreichbar ist. Abbildung 6.1 zeigt ein Beispiel für einen solchen Graphen; die Länge $l(e)$ einer Kante e ist jeweils als Beschriftung der Kante e angegeben.

Seien also ein Graph G mit Knotenmenge V und ein Knoten $s \in V$ gegeben. Um zu jedem Knoten $v \in V$ die Länge eines kürzesten Weges von s nach v zu bestimmen, könnte man natürlich sämtliche Wege von s nach v betrachten und unter diesen den mit kürzester Länge auswählen. Das ist aber höchstens für Graphen mit sehr wenigen Knoten und Kanten noch praktikabel.

Bereits 1959 hat Dijkstra [35] ein wesentlich effizienteres Verfahren vorgeschlagen. Wir skizzieren das Verfahren jetzt, ohne auf alle Implementationsdetails einzugehen und ohne die Korrektheit des Verfahrens zu begründen.

Das Verfahren besteht darin, sukzessive für jeden Knoten $v \in V$ den kürzesten Weg von s nach v zu bestimmen. Dazu wird eine Menge S von Knoten betrachtet und schrittweise vergrößert, für die der kürzeste Weg von s aus bereits bekannt ist. Jedem Knoten $v \in V$ wird eine vorläufige Distanz $d(v)$ zugeordnet. Falls $v \in S$ ist, ist $d(v)$ auch bereits die Länge des kürzesten Weges von s nach v. Falls $v \notin S$ ist, so ist $d(v)$ die Länge eines kürzesten Weges der Form $s \ldots wv$, mit $w \in S$, d.h. $d(v) = min(\{d(w) + l(wv); w \in S\})$ bzw. $d(v) = \infty$, falls ein solcher Weg nicht existiert.

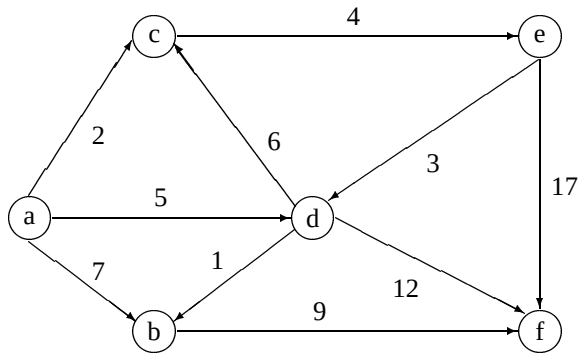

Abbildung 6.1

Anfangs ist $d(s) = 0$, und für alle von s verschiedenen Knoten $v \in V$ ist $d(v) = \infty$, und S ist leer. Dann wird S nach dem Prinzip „Knoten mit kürzester Distanz von s zuerst“ schrittweise wie folgt vergrößert, bis S alle Knoten V des Graphen enthält:

1. *Wähle Knoten $v \in V \setminus S$ mit minimaler Distanz $d(v)$.*
2. *Nimm v zu S hinzu.*
3. *Für jede Kante vw von v zu einem Knoten $w \notin S$ ersetze $d(w)$ durch $min(\{d(w), d(v) + l(vw)\})$.*

Wir implementieren $V \setminus S$ als Priority Queue und wählen als Prioritäten (Schlüssel) die Distanzen $d(v)$. Wir denken uns ferner für jeden Knoten $v \in V$ die Menge aller Knoten $w \in V$ mit $vw \in E$ in einer *Nachfolgermenge* $N(v)$ zusammengefaßt. Dann kann man das soeben informell beschriebene Verfahren von Dijkstra etwas genauer wie folgt formulieren:

```
procedure shortestpath ((V,E) : Graph; s : Knoten);
{berechnet zum Graphen mit Knotenmenge V und Kantenmenge E
 und zu s ∈ V für jeden Knoten v ∈ V die
 Länge d(v) des kürzesten Weges von s nach v}
begin
  {Initialisierung}
  for all v ∈ V \ {s} do d(v) := ∞;
  d(s) := 0; S := ∅;
  Initialisiere V \ S := V als Priority Queue, geordnet nach Distanzen
    d(v), v ∈ V;
  {anfangs ist also s minimales Element in V \ S; es folgen alle
   übrigen Elemente von V in beliebiger Reihenfolge}
  {vergrößern von S nach dem Prinzip: Knoten mit kürzester Distanz
   von s zuerst}
  while V \ S ≠ ∅ do
    begin
```

```
(∗)     v := min(V\S); deletemin(V\S);
        S := S∪{v};
        for all w ∈ N(v)\S do
          if d(v) + l(vw) < d(w)
(∗∗)        then decreasekey(w, d(v) + l(vw))
      end
    end
```

Wir verfolgen diesen Algorithmus am Beispiel des Graphen aus Abbildung 6.1 und Startknoten a. Dazu geben wir die Mengen S und $V\setminus S$, durch einen Strich getrennt, nach der Initialisierungsphase und nach jedem Durchlauf der **while**-Schleife an. $V\setminus S$ ist von links nach rechts nach aufsteigenden Distanzen geordnet. Ferner geben wir jeweils für die Knoten der Nachfolgermenge $N(v)$ des minimalen Elementes $v \in V\setminus S$ die Distanzen von s an. Der Ablauf des Verfahrens ist in Tabelle 6.1 zusammengefaßt. Das Ergebnis kann man in der letzten Zeile ablesen.

$(v,d(v))$ für $v \in S \mid (v,d(v))$ für $v \in V \setminus S$	für $v = \min(V \setminus S)$ und $w \in N(v) \setminus (S \cup \{v\})$: $\dfrac{(w,l(vw))}{\min(d(w), d(v)+l(vw))}$
$\emptyset \mid (a,0),(b,\infty),(c,\infty),(d,\infty),(e,\infty),(f,\infty)$	$\dfrac{(b,7),(c,2),(d,5)}{(b,7),(c,2),(d,5)}$
$(a,0) \mid (c,2),(d,5),(b,7),(e,\infty),(f,\infty)$	$\dfrac{(e,4)}{(e,6)}$
$(a,0),(c,2) \mid (d,5),(e,6),(b,7),(f,\infty)$	$\dfrac{(b,1),(f,12)}{(b,6),(f,17)}$
$(a,0),(c,2),(d,5) \mid (e,6),(b,6),(f,17)$	$\dfrac{(f,17)}{(f,17)}$
$(a,0),(c,2),(d,5),(e,6) \mid (b,6),(f,17)$	$\dfrac{(f,9)}{(f,15)}$
$(a,0),(c,2),(d,5),(e,6),(b,6) \mid (f,15)$	$\underline{\emptyset}$
$(a,0),(c,2),(d,5),(e,6),(b,6),(f,15) \mid \emptyset$	

Tabelle 6.1

Die Zeit, die das Verfahren für einen Graphen mit n Knoten und m Kanten benötigt, hängt offenbar ganz entscheidend davon ab, welche Zeit für die Bestimmung und das Entfernen des Minimums in Zeile $(*)$ und das Herabsetzen eines Schlüssels in Zeile

$(**)$ benötigt wird. Es ist klar, daß die Zeile $(*)$ höchstens n-mal ausgeführt wird. Weil die Anzahl aller Elemente in allen Nachfolgermengen $N(v)$ von Knoten $v \in V$ natürlich genau gleich der Anzahl m der Kanten des gegebenen Graphen ist, wird die Zeile $(**)$ insgesamt höchstens m-mal ausgeführt.

Bei geeigneter programmtechnischer Realisierung des Graphen ergibt sich dann als Laufzeit für den Algorithmus von Dijkstra die Größenordnung $O(t_{init} + n \cdot (t_m + t_{dm}) + m \cdot t_{dk})$. Dabei ist t_{init} die zur Initialisierung, also insbesondere zum Aufbau einer Priority Queue mit n Elementen erforderliche Schrittzahl; t_m, t_{dm} und t_{dk} bezeichnen jeweils die Zeit zur Ausführung einer Operation *Access Min*, *Delete Min* und *Decrease Key* auf der Priority Queue $V \setminus S$, die höchstens n Elemente enthält. Statt eine Abschätzung der Laufzeit über die im schlechtesten Fall zur Ausführung einer einzelnen Operation benötigten Zeit vorzunehmen, könnte man natürlich auch eine globalere, aber amortisierte Worst-case-Abschätzung für die Laufzeit des Verfahrens von Dijkstra vornehmen.

Nehmen wir einmal an, daß die Initialisierung auf jeden Fall in Zeit $O(n+m)$ möglich ist. Dann benötigt das Verfahren von Dijkstra höchstens soviele Schritte, wie erforderlich sind, um insgesamt n Operationen *Access Min* und *Delete Min* auszuführen, plus die Gesamtanzahl von Schritten zur Ausführung von m *Decrease Key* Operationen; die Operationen betreffen dabei jeweils Priority Queues mit höchstens n Elementen.

Verschiedene Implementationen von Priority Queues liefern damit unmittelbar verschiedene Implementationen für das Verfahren von Dijkstra zur Berechnung kürzester Wege.

6.1.2 Implementation von Priority Queues mit verketteten Listen und balancierten Bäumen

Verkettet gespeicherte, nicht sortierte Listen bilden eine erste offensichtliche Möglichkeit zur Implementation von Priority Queues. Wir können wie im Abschnitt 1.5 willkürlich voraussetzen, daß die Liste stets je ein unechtes Dummy-Element am Anfang und am Ende hat und der *next*-Zeiger des letzten Elements auf dieses selbst zurückweist. Die Liste ist durch den Zeiger *head* auf das Dummy-Element am Anfang gegeben. Die Verwendung von Dummy-Elementen ist ein Implementationstrick, der sichert, daß sich das Einfügen in die leere Liste algorithmisch nicht vom Einfügen in die nichtleere Liste unterscheidet. Ein neues Element wird immer nach dem Dummy-Element des Listenanfangs eingefügt. Daher ist das *Einfügen* in konstanter Zeit ausführbar. Um das *Minimum suchen* zu können, durchläuft man die Liste vom Anfang an mit zwei Hilfszeigern Z_1 und Z_2. Während Z_1 die Liste durchläuft, markiert Z_2 das bis dahin kleinste gefundene Element. Offenbar benötigt diese Operation im schlechtesten Fall $\Omega(N)$ Schritte. Der minimale Schlüssel wird entfernt, indem der Zeiger des Vorgängers auf das dem Minimum nachfolgende Element gerichtet wird. Das *Entfernen* des bereits gefundenen minimalen Elements ist in konstanter Zeit durchführbar. Zwei unsortierte, verkettete Listen kann man einfach aneinanderhängen. Dazu durchläuft man eine der Listen mit einem Zeiger, um den Zeiger des letzten Elements auf das erste Element der anderen Liste zu richten. Dieser Vorgang benötigt im schlechtesten Fall $\Omega(N)$ Schritte, wenn N die Länge der durchsuchten Liste ist. Falls die Operation des *Zusammenfügens* häufig benötigt wird, ist es besser, Priority Queues als Listen mit Anfangs- und Endzeiger zu

implementieren. Dann muß man beim Zusammenfügen keine Liste mehr durchlaufen, und die Operation wird in konstanter Zeit ausführbar.

Natürlich kann man Priority Queues auch mit Hilfe verkettet gespeicherter, sortierter linearer Listen implementieren. Wir können wieder eine Implementation mit Dummy-Elementen verwenden. Wir achten aber darauf, daß die Schlüssel aufsteigend sortiert sind. Beim *Einfügen* ist jetzt darauf zu achten, daß das neue Element die aufsteigende Ordnung der Schlüssel nicht zerstört. Das Suchen der richtigen Einfügeposition und das anschließende Einfügen benötigen im schlechtesten Fall $\Omega(N)$ Schritte. Bei dieser Implementation steht der kleinste Schlüssel immer am Anfang der Liste (nach dem Dummy-Element). Daher ist die Operation *Minimum suchen* in konstanter Zeit ausführbar. Zum *Entfernen* des Minimums genügt es, den *next*-Zeiger des *head*-Elements umzuhängen. Um zwei Listen zusammenzufügen, durchläuft man mit einem Zeiger die eine Liste und fügt die Elemente in die andere Liste an der jeweils richtigen Stelle ein. Die dazu erforderliche Schrittzahl ist proportional zur Gesamtanzahl der Elemente in beiden Listen. Es ist nicht schwer, sich zu überlegen, wie die übrigen Operationen (Entfernen eines beliebigen Elements, Herabsetzen eines Schlüssels) bei dieser Implementation von Priority Queues mit linearen Listen ausgeführt werden können.

Um für alle Operationen Laufzeiten von der Größenordnung $O(\log N)$ zu erhalten, kann man Implementationen mit Baumstrukturen verwenden. Grundsätzlich eignet sich dazu jede Klasse balancierter Bäume. Wir skizzieren hier, wie eine Implementation mit Bruder-Bäumen, vgl. Abschnitt 5.2.2, aussehen könnte. Für die Implementation verwenden wir eine Variante von Bruder-Bäumen, bei der die Schlüssel in den Blättern der Bäume gespeichert werden; die inneren Knoten enthalten jeweils das Minimum im Teilbaum unterhalb des Knotens. Die in den Blättern gespeicherten Schlüssel müssen, anders als bei Suchbäumen, nicht sortiert vorliegen. Natürlich ist es überflüssig, in den unären Knoten eines Bruder-Baumes Schlüssel zu speichern; denn nach der gerade getroffenen Festlegung müssen die Werte unärer Knoten mit denen ihrer einzigen Söhne identisch sein.

Abbildung 6.2 zeigt eine als Bruder-Baum gespeicherte Priority Queue mit acht Schlüsseln.

Einen neuen Schlüssel kann man an beliebiger Stelle unter den Blättern *einfügen*, z.B. immer ganz rechts. Im allgemeinen ist es dann erforderlich, den Baum umzustrukturieren, damit nach dem Einfügen wieder ein Bruder-Baum entsteht. Auf die für Bruder-Bäume typischen Umstrukturierungsoperationen nach einer Einfügung sind wir in Abschnitt 5.2.2 eingegangen; im Unterschied zur dort angegebenen Beschreibung muß man in der Umstrukturierungsinvariante aber jetzt die Sortierung von Schlüsselwerten ignorieren. Wichtig ist hier außerdem, daß im ungünstigsten Fall der Baum längs eines Pfades vom neuen Blatt bis zur Wurzel umstrukturiert werden muß. Zugleich müssen die Minima längs dieses Pfades adjustiert werden. Man weiß, daß das Einfügen in einen Bruder-Baum in $O(\log N)$ Schritten ausführbar ist. Das Minimum kann stets an der Wurzel abgelesen werden. Die Operation *Access Min* ist deshalb in $O(1)$ Schritten ausführbar. Um das Minimum zu *entfernen*, muß man das Blatt mit dem minimalen Wert entfernen. Dazu folgt man auf dem Weg von der Wurzel zu den Blättern immer dem Knoten mit dem kleineren Wert. Nach dem Entfernen des Blattes, das das Minimum enthält, sind im allgemeinen Umstrukturierungen nötig, um wieder einen Bruder-Baum zu erhalten. Ferner müssen auch die Werte der inneren Knoten auf dem Pfad von dem Vater des entfernten Schlüssels bis zur Wurzel geändert werden. Das Entfernen eines

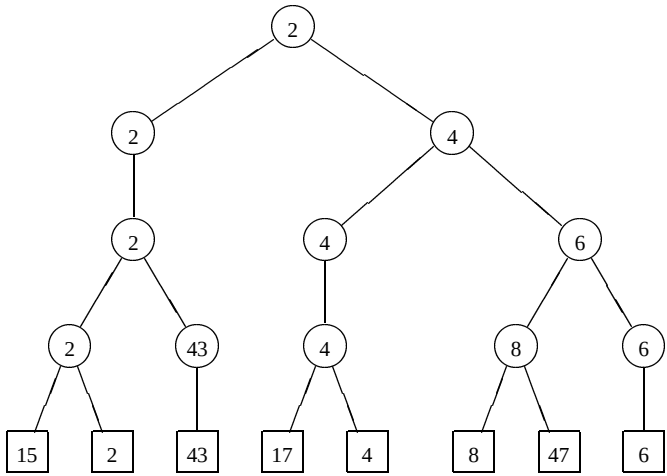

Abbildung 6.2

beliebigen Elementes ist ebenfalls in insgesamt $O(\log N)$ Schritten ausführbar, wenn man die Position des zu entfernenden Elements kennt. Dasselbe gilt für das Herabsetzen eines Schlüssels, das man als Entfernen des alten und Wiedereinfügen des neuen Schlüsselwertes realisieren kann.

Das *Zusammenfügen* zweier als balancierte Bäume implementierter Priority Queues läuft auf das Vereinigen zweier Bäume unterschiedlicher Höhe hinaus. Dazu suchen wir im Baum A den rechtesten Teilbaum mit gleicher Höhe wie Baum B. (Ohne Einschränkung nehmen wir an, daß A der höhere Baum ist.) Hat dieser Teilbaum einen unären Vater, können wir Baum B zum zweiten Teilbaum dieses Vaterknotens machen. Hat der Vater p des Teilbaums bereits zwei Söhne, so fügt man die Wurzel von B als dritten Sohn von p ein und strukturiert den Baum von p aufwärts so um, daß insgesamt wieder ein Bruder-Baum entsteht. Zum Schluß müssen noch die Werte innerer Knoten auf dem Pfad von der Stelle, an der der Baum eingefügt wurde, bis zur Wurzel überprüft und gegebenenfalls verändert werden. Diese Operation ist insgesamt in $O(\log N)$ Schritten ausführbar.

6.1.3 Linksbäume

Balancierte Bäume haben die Eigenschaft, daß *jeder* Pfad von der Wurzel zu einem Blatt des Baumes mit $N+1$ Blättern und N inneren Knoten eine Länge der Größenordnung $O(\log N)$ hat. Für die Implementation von Priority Queues reicht eine wesentlich schwächere Forderung aus, um zu sichern, daß die für Priority Queues typischen Operationen Access Min in Zeit $O(1)$ und das Einfügen, Entfernen des Minimums und das Verschmelzen zweier Priority Queues sämtlich in Zeit $O(\log N)$ ausführbar sind. Es genügt, dafür zu sorgen, daß Bäume verwendet werden, die zwei Bedingungen erfüllen.

Die Schlüsselwerte der Söhne müssen (wie bei Heaps) stets größer sein als der Schlüssel des Vaters. Ferner muß es wenigstens einen Pfad von der Wurzel zu einem Blatt mit Länge $O(\log N)$ geben. Wenn man dann Einfüge- und Verschmelze-Vorgänge längs eines solchen kurzen Pfades vornimmt, kann man ein Verhalten garantieren, das genauso gut ist wie bei der Verwendung einer Klasse balancierter Bäume. Diese Idee führt zur Definition von Linksbäumen. Ein binärer Baum heißt ein *Linksbaum*, wenn gilt: Jeder innere Knoten enthält neben dem Schlüssel und den zwei Zeigern auf die beiden Söhne noch ein sogenanntes Distanzfeld, in dem die Entfernung des Knotens zum nächstgelegenen Blatt festgehalten wird. Blätter haben die Distanz 0. Das Knotenformat kann also durch folgende Typvereinbarung beschrieben werden.

```
type Linksbaum = ↑Knoten;
     Knoten = record
                  Schlüssel : integer;
                  Dist : integer;
                  links, rechts : Linksbaum;
                  info : {infotype}
              end
```

Für einen Linksbaum wird nun zunächst gefordert, daß für jeden inneren Knoten p gilt: $p.Schlüssel < p.links\uparrow.Schlüssel$ und $p.Schlüssel < p.rechts\uparrow.Schlüssel$. Ferner verlangt man, daß für jeden inneren Knoten p gilt: $p.Dist = 1 + min\ (p.links\uparrow.Dist, p.rechts\uparrow.Dist), p.links\uparrow.Dist \geq p.rechts\uparrow.Dist$. Also muß stets $p.Dist = 1 + p.rechts\uparrow.Dist$ gelten.

Aufgrund der letzten Bedingung ist ein kürzester Pfad von der Wurzel zu einem Blatt immer der Pfad zum rechtesten Blatt. Die Abbildung 6.3 zeigt das Beispiel eines Linksbaumes, der die Schlüssel {2, 3, 4, 5, 7, 8, 9} speichert; Schlüssel sind links oben, Distanzen jeweils rechts oben in den Knoten eingetragen. Blätter sind durch **nil**-Zeiger in den Vätern repräsentiert.

Wie bei Heaps kann man das Minimum in konstanter Zeit an der Wurzel ablesen. Weil offenbar jeder Teilbaum eines Linksbaumes wieder ein Linksbaum sein muß, kann man alle anderen Operationen an Linksbäumen auf das Verschmelzen zweier Linksbäume zu einem neuen zurückführen. Das *Einfügen* eines Schlüssels k in den Linksbaum A kann man auffassen als Verschmelzen von A mit dem Linksbaum B, der einen einzigen inneren Knoten mit Schlüssel k und Distanz 1 enthält. Das *Entfernen des Minimums* aus einem Linksbaum bedeutet das Entfernen der Wurzel und Verschmelzen der beiden Teilbäume der Wurzel. Das *Entfernen eines beliebigen* inneren Knotens p eines Linksbaumes kann man wie folgt durchführen. Der Linksbaum zerfällt beim Wegnehmen von p in einen oberhalb von p liegenden Teilbaum A, den linken Teilbaum B und den rechten Teilbaum C von p. In A ersetzt man p durch ein Blatt, das man gegebenenfalls mit seinem Bruder vertauscht, um links den Teilbaum mit größerer Distanz anzubringen. Dann adjustiert man die Distanz des Vaters von p. Dies setzt man für die Knoten von A auf dem Pfad zur Wurzel fort. Schließlich verschmilzt man die drei Linksbäume A, B und C zu einem neuen. Schließlich kann man das *Herabsetzen* eines Schlüssels als Entfernen des Schlüssels und anschließendes Wiedereinfügen des neuen, herabgesetzten Schlüsselwertes auffassen. Man beachte aber, daß das Adjustieren der Distanzen und gegebenenfalls das erforderliche Vertauschen von Teilbäumen auf dem Pfad von p

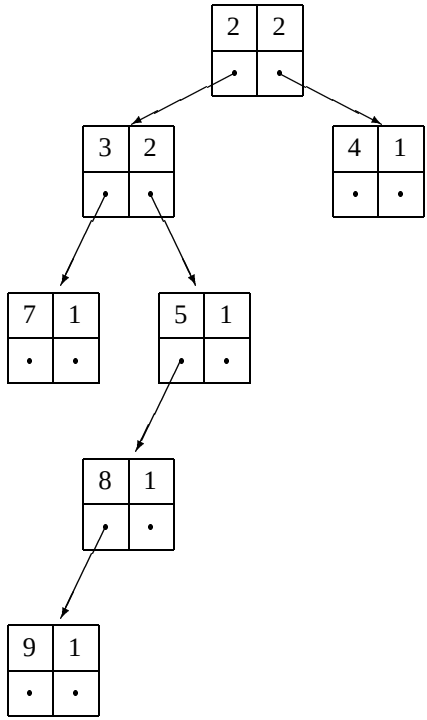

Abbildung 6.3

zur Wurzel von A eine Anzahl von Schritten erfordert, die proportional zur Höhe von A sein kann, weil der Pfad von p zur Wurzel im allgemeinen nicht der rechteste Pfad in A ist. Daher kann der Aufwand zur Ausführung der Operationen Entfernen eines beliebigen Knotens und Herabsetzen eines Schlüssels im schlechtesten Fall $\Omega(N)$ Schritte erfordern.

Dagegen können die Operationen des Einfügens und Entfernens des Minimums in logarithmischer Schrittzahl ausgeführt werden, wenn es gelingt, das *Verschmelzen* (Merge) zweier Linksbäume entsprechend effizient durchzuführen. Wir sorgen dafür, daß der dazu erforderliche Aufwand von der Größenordnung der Summe der Längen der Pfade von den Wurzeln der beteiligten Bäume zum jeweils rechtesten Blatt ist. Weil der *kürzeste* Pfad in einem beliebigen Binärbaum mit N inneren Knoten natürlich höchstens die Länge $\lceil \log_2(N+1) \rceil$ haben kann, folgt dann sofort, daß die Operationen Einfügen, Entfernen des Minimums und Verschmelzen in logarithmischer Zeit ausführbar sind.

Die Operation *Merge* kann auf einfache Weise rekursiv erklärt werden. Betrachten wir das Problem, zwei Linksbäume A und B zu verschmelzen. Wir können ohne Einschränkung annehmen, daß der Schlüssel in der Wurzel von Baum A kleiner als der Schlüssel in der Wurzel von Baum B ist. Baum A und B werden zusammengefügt, indem zunächst der rechte Teilbaum von A mit Baum B zu einem neuen Linksbaum C

verschmolzen wird. Dann wird C zum neuen rechten Teilbaum von A gemacht. Falls die Distanz der Wurzel des entstandenen Baumes im rechten Teilbaum, also die Distanz der Wurzel von C, größer ist als im linken, werden die beiden Teilbäume vertauscht. Die Distanz der Wurzel des neuen Baumes ist gleich der ihres rechten Sohnes plus 1. Das Zusammenfügen des rechten Teilbaums von A mit Baum B geschieht (rekursiv) nach derselben Vorschrift. Die Rekursion endet, wenn die Wurzel des Baumes mit dem kleineren Schlüssel keinen rechten Sohn mehr hat. Dann wird der andere Baum einfach als neuer rechter Teilbaum angehängt, und gegebenenfalls werden die beiden Teilbäume anschließend vertauscht.

Das Verfahren kann in Pascal wie folgt formuliert werden:

```
function Verschmelzen (A, B : Linksbaum) : Linksbaum;
begin
  if A = nil then Verschmelzen := B
  else if B = nil then Verschmelzen := A
    else begin
           if A↑.Schlüssel > B↑.Schlüssel
             then Vertausche A mit B;
           {jetzt gilt A↑.Schlüssel < B↑.Schlüssel}
           A↑.rechts := Verschmelzen(A↑.rechts, B);
           if A↑.rechts↑.Dist > A↑.links↑.Dist
             then vertausche A↑.rechts mit A↑.links in A;
           A↑.Dist := A↑.rechts↑.Dist +1;
           Verschmelzen := A
         end
end
```

Man kann aus dieser rekursiven Formulierung unmittelbar ablesen, daß die Laufzeit des Verfahrens proportional zur Summe der Längen des rechtesten Pfades in A und in B ist. Linksbäume wurden von Crane 1972 erfunden, vgl. dazu [89].

6.1.4 Binomial Queues

Wir definieren für jedes $n \geq 0$ die Struktur eines *Binomialbaumes* B_n wie folgt:

(i) B_0 ist ein aus genau einem Knoten bestehender Baum.

(ii) B_{n+1} entsteht aus zwei Exemplaren von B_n, indem man die Wurzel eines Exemplars von B_n zum weiteren Sohn der Wurzel des anderen macht.

Graphisch kann man diese Definition auch kurz so mitteilen, wie es Abbildung 6.4 zeigt.

Die Abbildung 6.5 zeigt die Struktur der Binomialbäume $B_0, \ldots, B_4$. Binomialbäume sind also keine Binärbäume. Wir haben in der Abbildung 6.5 alle Knoten, die denselben Abstand zur Wurzel haben, also alle Knoten gleicher Tiefe, nebeneinander gezeichnet. Aus der Definition kann man leicht die folgenden strukturellen Eigenschaften von Binomialbäumen ableiten:

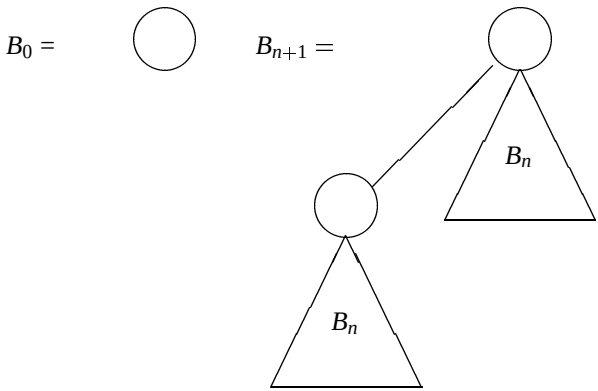

Abbildung 6.4

(1) B_n besteht aus genau 2^n Knoten.

(2) B_n hat die Höhe n.

(3) Die Wurzel von B_n hat die Ordnung n, d.h. sie hat genau n Söhne.

(4) Die n Teilbäume der Wurzel von B_n sind genau $B_{n-1}, B_{n-2}, \ldots, B_1, B_0$.

(5) B_n hat $\binom{n}{i}$ Knoten mit Tiefe i.

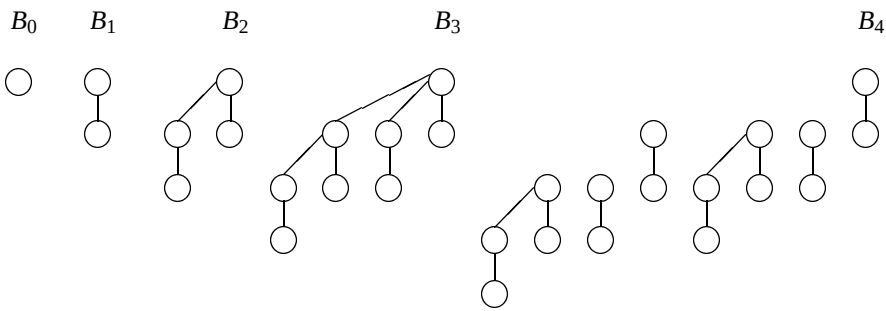

Abbildung 6.5

Wir wollen Binomialbäume zur Speicherung von Schlüsselmengen verwenden, so daß eine schwache Ordnungsbeziehung für die gespeicherten Schlüssel gilt, wie wir sie von Heaps kennen: Für jeden Knoten gilt, daß der in ihm gespeicherte Schlüssel kleiner ist als die Schlüssel seiner Söhne. Wir nennen einen Baum mit dieser Eigenschaft

heapgeordnet. Außerdem möchten wir nicht nur Mengen von N Schlüsseln speichern können, wenn $N = 2^n$, also eine Zweierpotenz ist. Dazu stellen wir N als Dualzahl dar: $N = (d_{n-1}d_{n-2}\dots d_0)_2$. Dann wählen wir für jedes j mit $d_j = 1$ einen Binomialbaum B_j; die Schlüsselmenge wird nun durch den *Wald* dieser Binomialbäume repräsentiert. Jeder Binomialbaum für sich muß heapgeordnet sein.

Beispiel: Gegeben sei die folgende Menge von elf Schlüsseln {2, 4, 6, 8, 14, 15, 17, 19, 23, 43, 47}. Weil $11 = (1011)_2$ ist, können die Schlüssel in einem Wald t_{11} von drei Binomialbäumen B_3, B_1, B_0 mit jeweils acht, zwei und einem Knoten gespeichert werden. Eine zulässige Speicherung, bei der die Werte der Söhne stets größer sind als die in den Vätern gespeicherten Schlüssel, zeigt Abbildung 6.6.

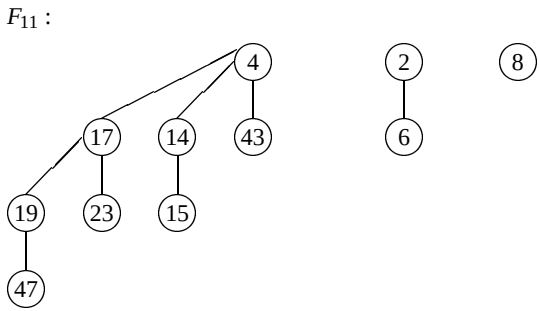

Abbildung 6.6

Man benötigt also zur Speicherung einer Menge von N Schlüsseln gerade so viele heapgeordnete Binomialbäume, wie Einsen in der Dualdarstellung von N auftreten. B_j wird genau dann benutzt, wenn an der j-ten Stelle in der Dualdarstellung von N die Ziffer 1 auftritt. Eine derartige Repräsentation einer Menge von N Schlüsseln nennen wir eine *Binomial Queue*. Denn wir werden jetzt zeigen, daß man alle für Priority Queues üblichen Operationen mit solchen Wäldern von Binomialbäumen durchführen kann.

Zunächst ist klar, wie man das Minimum einer in einer Binomial Queue F_N gespeicherten Menge von N Schlüsseln bestimmt. Man inspiziert die Wurzeln aller Binomialbäume des Waldes F_N, die die Queue bilden, und nimmt davon das Minimum. Da es natürlich höchstens $\lceil \log_2 N \rceil + 1$ Bäume in diesem Wald geben kann, ist klar, daß man das Minimum in $O(\log N)$ Schritten bestimmen kann.

Wir erklären jetzt, wie man zwei Binomial Queues zu einer neuen *verschmelzen* kann. (Dabei werden allerdings einige durchaus wesentliche Implementationsdetails zunächst offengelassen, die wir erst später angeben.) Das Verschmelzen zweier Binomialbäume B_n gleicher Größe mit jeweils genau 2^n Elementen ist ganz einfach. Die Struktur des durch Verschmelzen entstehenden Baumes ist ja bereits in der Definition festgelegt; wir müssen nur noch darauf achten, daß beim Zusammenfügen von zwei Exemplaren B_n zu B_{n+1} dasjenige Exemplar zur Wurzel von B_{n+1} wird, das den kleineren Schlüssel in der Wurzel hat.

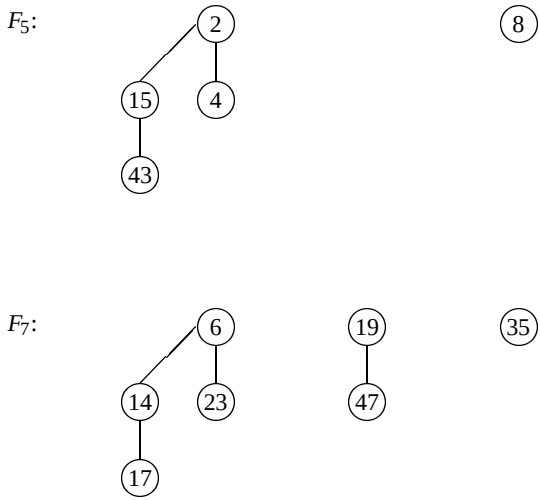

Abbildung 6.7

Das Zusammenfügen zweier Binomial Queues, die nicht genau aus zwei gleichgroßen Binomialbäumen bestehen, orientiert sich am bekannten Schulverfahren zur Addition zweier Dualzahlen. Seien also zwei Binomial Queues F_{N_1} und F_{N_2} mit N_1 und N_2 Elementen gegeben; sie bestehen jeweils aus Wäldern von höchstens $\lceil \log_2 N_1 \rceil + 1$ und $\lceil \log_2 N_2 \rceil + 1$ Binomialbäumen. Das Verfahren zum Verschmelzen der zwei Binomial Queues betrachtet die Binomialbäume der Wälder F_{N_1} und F_{N_2} der Reihe nach in aufsteigender Größe. Wie bei der Addition von Dualzahlen betrachtet man in jedem Schritt zwei Binomialbäume der gegebenen Queues und eventuell einen als *Übertrag* erhaltenen Binomialbaum. Anfangs hat man keinen Übertrag. Im i-ten Schritt hat man als Operanden einen Binomialbaum B_i der ersten Queue, wenn in der Dualdarstellung von N_1 an der i-ten Stelle eine 1 auftritt, ferner einen Binomialbaum B_i der zweiten Queue, wenn in der Dualdarstellung von N_2 an der i-ten Stelle eine 1 auftritt, und eventuell einen Binomialbaum B_i als Übertrag.

Ist keiner der drei Operanden vorhanden, ist auch die i-te Komponente des Ergebnisses nicht vorhanden; tritt genau einer der drei genannten Operanden auf, bildet er die i-te Komponente des Ergebnisses, und es wird kein Übertrag für die nächsthöhere Stelle erzeugt. Treten genau zwei Operanden auf, werden sie zu einem Binomialbaum B_{i+1} wie oben angegeben zusammengefaßt und als Übertrag an die nächsthöhere Stelle weitergegeben; die i-te Komponente des Ergebnisses ist nicht vorhanden. Sind schließlich alle drei Operanden vorhanden, wird einer zur i-ten Komponente des Ergebnisses; die beiden anderen werden zu einem Binomialbaum B_{i+1} zusammengefaßt und als Übertrag an die nächsthöhere Stelle übertragen.

Wir erläutern das Verfahren an folgendem Beispiel. Gegeben seien die Binomial Queues F_5 und F_7 mit $N_1 = 5$ und $N_2 = 7$ Elementen, vgl. Abbildung 6.7. Addition von N_1 und N_2 im Dualsystem ergibt:

N_1		1	0	1
N_2		1	1	1
Übertrag	1	1	1	0
Ergebnis	1	1	0	0

Das Verschmelzen von F_5 und F_7 zu F_{12} zeigt Abbildung 6.8.

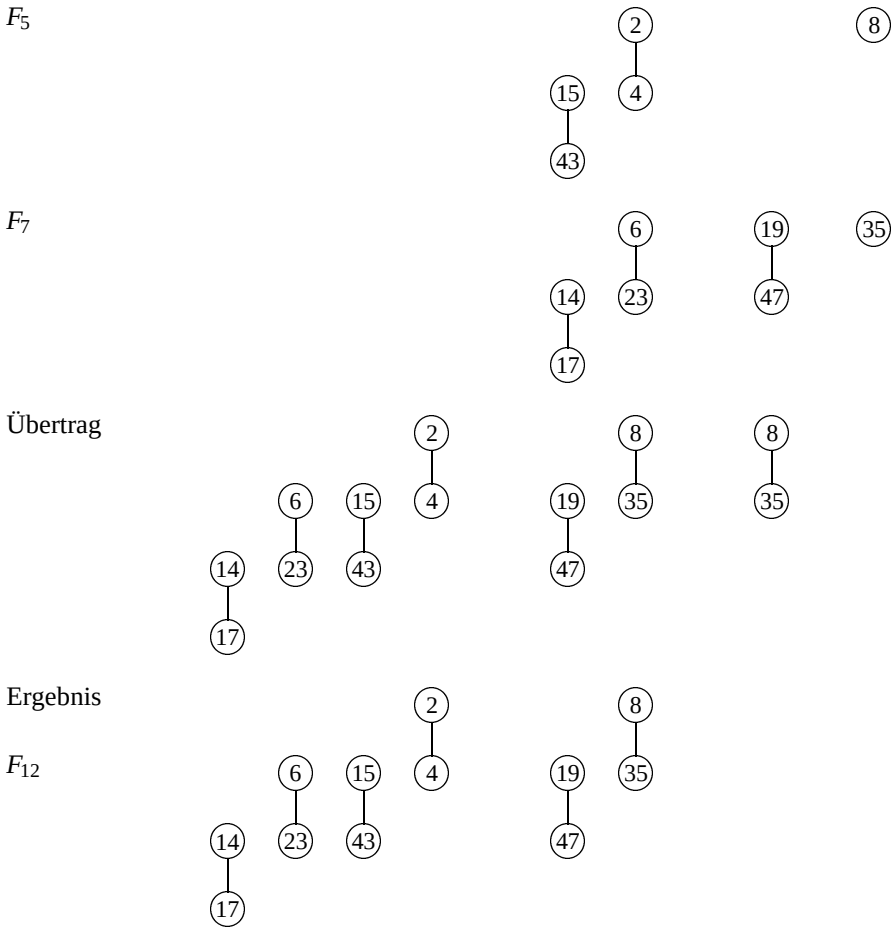

Abbildung 6.8

Es sollte klar sein, daß das Verschmelzen zweier Binomial Queues F_{N_1} und F_{N_2} mit N_1 und N_2 Elementen in $O(\log N_1 + \log N_2)$ Schritten ausführbar ist, wenn man voraussetzt, daß das Anhängen eines weiteren Sohnes an die Wurzel eines Binomialbaumes in konstanter Zeit möglich ist. Bevor wir auf diese Voraussetzung genauer eingehen, wollen wir uns zunächst überlegen, daß man die Operationen *Einfügen* eines neuen Elementes, *Entfernen des Minimums*, *Entfernen* eines beliebigen Elementes und *Herabsetzen* eines Schlüssels sämtlich auf das Verschmelzen von Binomial Queues zurückführen kann.

Für das Einfügen eines neuen Elementes ist dies offensichtlich.

Der minimale Schlüssel einer Binomial Queue F_N ist Schlüssel der Wurzel eines Binomialbaumes B_i im Wald von Binomialbäumen, die F_N bilden. Entfernt man diese Wurzel, zerfällt B_i in Teilbäume $B_{i-1}, B_{i-2}, \ldots, B_0$; sie bilden einen Wald F_{2^i-1}. Läßt man B_i aus dem Wald F_N weg, bleibt ein Wald F_{N-2^i} übrig. Verschmelzen dieser beiden Wälder liefert das gewünschte Ergebnis.

Das Entfernen eines Schlüssels k, der nicht in der Wurzel eines Binomialbaumes B_i im die Binomial Queue bildenden Wald F_N auftritt, ist schwieriger. Wir können aber annehmen, daß k in B_i auftritt (allerdings nicht an der Wurzel), B_i Binomialbaum im Wald F_N. Wir entfernen B_i aus F_N und erhalten einen Wald F_{N_1} mit $N_1 = N - 2^i$.

B_i besteht aus zwei Exemplaren B_{i-1}, einem linken Teilbaum B^l_{i-1} und einem rechten Teilbaum B^r_{i-1}, vgl. Abbildung 6.9.

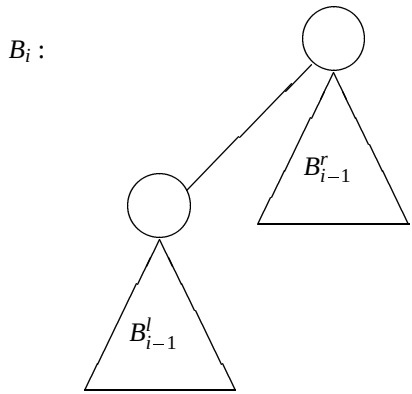

Abbildung 6.9

Kommt k in B^l_{i-1} vor, bilden wir einen neuen Wald F_{N_2}, in den wir zunächst B^r_{i-1} aufnehmen; kommt k in B^r_{i-1} vor, nehmen wir in F_{N_2} zunächst B^l_{i-1} auf. Dann zerlegen wir B_{i-1} auf dieselbe Weise und nehmen immer wieder kleinere Binomialbäume zu F_{N_2} hinzu, bis wir bei einem Binomialbaum B_j angekommen sind, der k als Schlüssel der Wurzel hat. Dann entfernen wir diese Wurzel und nehmen die Teilbäume $B_{j-1}, \ldots, B_0$ der Wurzel noch zu F_{N_2} hinzu. Insgesamt erhalten wir so zwei Wälder F_{N_1} und F_{N_2}, die nach dem oben angegebenen Verfahren verschmolzen werden können.

Entfernt man z.B. aus dem in Abbildung 6.6 gezeigten Wald F_{11} den Schlüssel 14, so zerfällt F_{11} zunächst in die in Abbildung 6.10 gezeigten Bäume F_3 und F_7, die anschließend verschmolzen werden müssen.

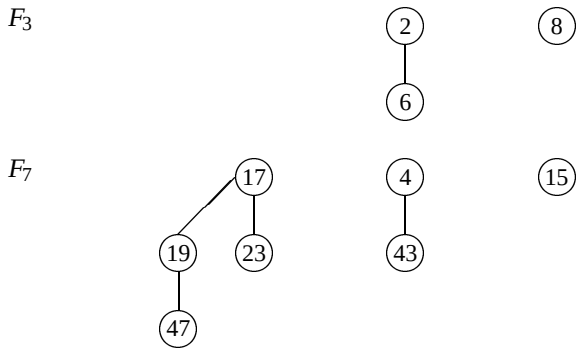

Abbildung 6.10

Das Herabsetzen eines Schlüssels kann man, wie bisher stets, auf das Entfernen des Schlüssels und das anschließende Wiedereinfügen des herabgesetzten Schlüssels zurückführen. Alternativ kann man auch den erniedrigten Schlüssel so oft mit seinem Vater vertauschen, bis die Heapordung wiederhergestellt ist.

Eine *Implementation* dieser Verfahren verlangt es, Bäume mit unbeschränkter Ordnung programmtechnisch zu realisieren. Denn Binomialbäume B_n sind Bäume der Ordnung n, weil die Wurzel n Söhne hat. Man könnte natürlich einen maximalen Knotengrad als Obergrenze vorsehen und jedem Knoten erlauben, soviele Söhne zu haben, wie dieser Knotengrad angibt. Das hätte aber eine enorme Verschwendung von Speicherplatz zur Folge, die weder sinnvoll noch nötig ist.

Vuillemin [190] schlägt vor, Binomialbäume, und damit Binomial Queues, als Binärbäume wie folgt zu repräsentieren: Jeder Knoten eines Binomialbaumes enthält genau zwei Zeiger, einen Zeiger *llink* auf den linkesten Sohn und einen Zeiger *rlink* auf seinen rechten Nachbarn. Hat ein Knoten keinen rechten Nachbarn, kann man den Zeiger *rlink* auf den Vater des Knotens zurückweisen lassen. Nach diesem Prinzip kann man beliebige Vielwegbäume als Binärbäume repräsentieren, also nicht nur Binomialbäume.

Abbildung [illegible] zeigt als Beispiel eine Binärbaum-Repräsentation des Binomialbaumes B_3 aus dem Wald F_{11} von Abbildung 6.6.

Sollen zwei als Binärbäume repräsentierte Binomialbäume zu einem neuen verschmolzen werden, muß man den *llink*-Zeiger der Wurzel des einen Baumes auf die Wurzel des anderen umlegen und den *rlink*-Zeiger der Wurzel des zweiten Baumes auf den linkesten Sohn der Wurzel des ersten Baumes zeigen lassen, falls dieser einen Sohn hatte; sonst läßt man den *rlink*-Zeiger auf die Wurzel des neuen Baumes zurückweisen. Es ist klar, daß diese Operationen in konstanter Zeit ausführbar sind. Diese Ope-

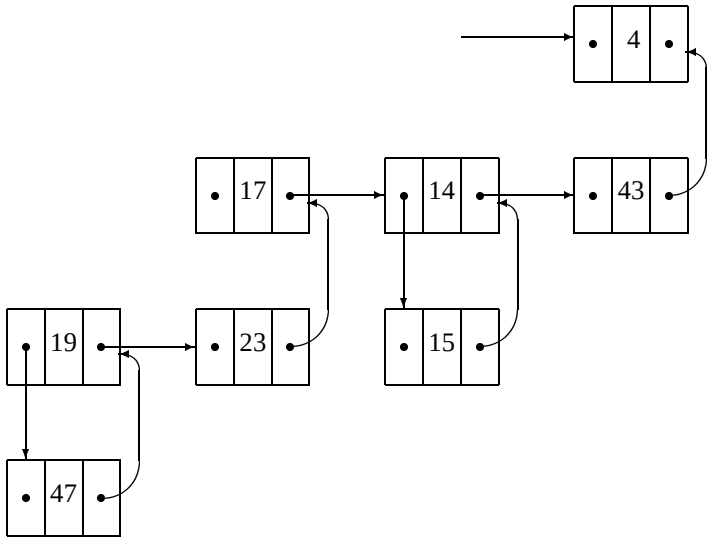

Abbildung 6.11

rationen bilden die Grundlage für eine Prozedur zum Verschmelzen zweier Binomial Queues. Für weitere Einzelheiten der programmtechnischen Realisierung der Algorithmen dieses Abschnitts konsultiere man [190]. Insgesamt ergibt sich, daß alle genannten Operationen Access Min, Einfügen, Meld, Minimum Entfernen, Decrease Key, Delete in Zeit $O(\log N)$ ausführbar sind für eine Binomial Queue mit N Elementen.

6.1.5 Fibonacci-Heaps

Die Struktur von Binomialbäumen und Binomial Queues ist ebenso starr wie die von Heaps. Für eine gegebene Zahl N gibt es jeweils nur eine einzige Struktur mit N Knoten. Lediglich die Verteilung der Schlüssel ist nicht eindeutig bestimmt, weil nur verlangt wird, daß die Bäume heapgeordnet sein müssen. Fibonacci-Heaps sind wesentlich weniger starr. Ein *Fibonacci-Heap* (kurz: *F-Heap*) ist eine Kollektion heapgeordneter Bäume mit jeweils disjunkten Schlüsselmengen. Es wird keine weitere Forderung an die Struktur von F-Heaps gestellt. Dennoch haben F-Heaps eine implizit durch die für F-Heaps erklärten Operationen festgelegte Struktur. Die Klasse der F-Heaps ist die kleinste Klasse von heapgeordneten Bäumen, die gegen die später erklärten Operationen *Initialisieren* (des leeren F-Heaps), *Einfügen* eines Schlüssels, *Access Min*, *Delete Min*, *Decrease Key*, *Delete* und *Meld* abgeschlossen ist. Wir werden sehen, daß F-Heaps eng mit den im Abschnitt 6.1.4 behandelten Binomial Queues zusammenhängen.

Die genannten Operationen für F-Heaps verändern die Kollektion heapgeordneter Bäume. Es können neue heapgeordnete Bäume in die Kollektion aufgenommen werden oder zwei (oder mehrere) heapgeordnete Bäume zu einem neuen heapgeordneten Baum verschmolzen werden. Diese Operation des Verschmelzens von zwei heapgeordneten Bäumen ist genau die von Binomialbäumen bekannte Operation. Zwei heapgeordnete Bäume, deren Wurzeln denselben Rang r haben, können zu einem heapgeordneten Baum mit Rang $r+1$ verschmolzen werden, indem man die Wurzel des Baumes mit dem größeren Schlüssel zum weiteren, $(r+1)$-ten Sohn der Wurzel des Baumes macht, der den kleineren Schlüssel in der Wurzel hat. Anders als bei Binomialbäumen und Binomial Queues kann es bei F-Heaps jedoch vorkommen, daß Bäume verschmolzen werden, die nicht dieselbe Knotenzahl haben. (Das gilt aber höchstens dann, wenn die Operationen Decrease Key und Delete in einer Operationsfolge für F-Heaps vorkommen.) Bevor wir jetzt der Reihe nach die oben genannten Operationen für F-Heaps erklären, wollen wir angeben, wie F-Heaps implementiert werden, damit wir die Zeit zur Ausführung der Operationen abschätzen können.

Ein F-Heap besteht aus einer Kollektion heapgeordneter Bäume; die Wurzeln dieser Bäume sind Elemente einer doppelt verketteten, zyklisch geschlossenen Liste. Diese Liste heißt die *Wurzelliste* des F-Heaps. Der F-Heap ist gegeben durch einen Zeiger auf das Element mit minimalem Schlüssel in der Liste. Dieses Element heißt das *Minimalelement* des F-Heaps. Jeder Knoten eines heapgeordneten Baumes hat einen Zeiger auf seinen Vater (wenn er einen Vater hat, und sonst einen **nil**-Zeiger) und einen Zeiger auf einen seiner Söhne. Ferner sind alle Söhne eines Knotens untereinander doppelt, zyklisch verkettet. Außerdem hat jeder Knoten ein Rangfeld, das die Anzahl seiner Söhne angibt, und ein Markierungsfeld, dessen Bedeutung später erklärt wird. Das Knotenformat eines in einem F-Heap auftretenden Baumes kann also durch folgende Typvereinbarung beschrieben werden:

```
type heap-ordered-tree = ↑Knoten;
     Knoten = record
                  links, rechts : ↑Knoten;
                  vater, sohn : ↑Knoten;
                  key : integer;
                  rank : integer;
                  marker : boolean
              end
```

Natürlich kann man jede Binomial Queue auch als F-Heap auffassen und wie soeben angegeben implementieren. Abbildung 6.12 zeigt F_7 aus Abbildung 6.7 als F-Heap; wir haben allerdings die Rang- und Markierungsfelder weggelassen.

Wir erklären jetzt die Operationen für F-Heaps. Die Operationen Initialisieren, Einfügen, Access Min und Verschmelzen (Meld) ändern weder die Rang- noch die Markierungsfelder von bereits existierenden Knoten; sie sind wie folgt erklärt.

Initialisieren des leeren F-Heaps: Liefert einen **nil**-Zeiger.

Einfügen eines Schlüssels k in einen F-Heap h: Bilde einen F-Heap h' aus einem einzigen Knoten, der k speichert. (Dieser Knoten ist unmarkiert und hat Rang 0.) Verschmilz h und h' zu einem neuen F-Heap, vgl. unten.

Access Min: Das Minimum eines F-Heaps h ist im Minimalknoten von h gespeichert.

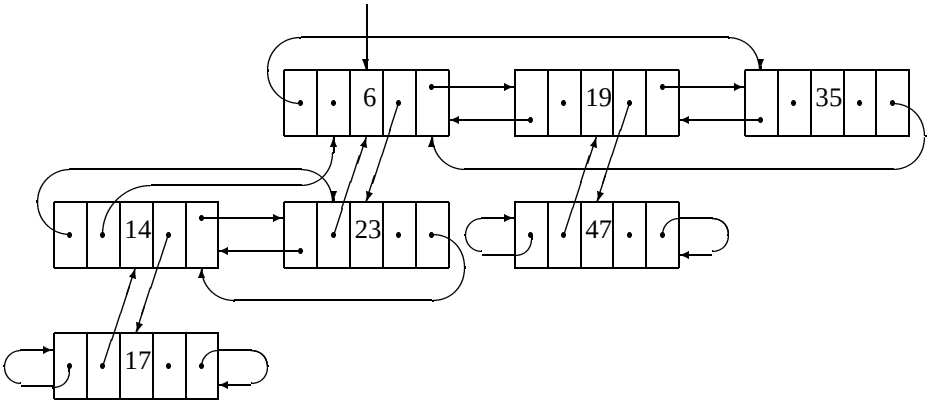

Abbildung 6.12

Das *Verschmelzen* (Meld) zweier F-Heaps h_1 und h_2 mit disjunkten Schlüsselmengen geschieht durch Aneinanderhängen der beiden Wurzellisten von h_1 und h_2. Minimalelement des resultierenden F-Heaps ist das kleinere der beiden Minimalelemente von h_1 und h_2; als Ergebnis der Verschmelze-Operation wird ein Zeiger auf dieses Element abgeliefert.

Offenbar sind alle diese Operationen in Zeit $O(1)$ ausführbar, wenn man F-Heaps wie oben angegeben implementiert. Man beachte den Unterschied zwischen der Verschmelze-Operation (Meld-Operation) für F-Heaps und der entsprechenden Operation für Binomial Queues: Die Verschmelze-Operation für F-Heaps sammelt nur die den F-Heap bildenden heapgeordneten Bäume in der Wurzelliste, ohne diese Bäume zu größeren zu verschmelzen; die entsprechende Operation für Binomial Queues fügt die Bäume analog zur Addition zweier Dualzahlen zusammen. Dies einer Dualzahladdition entsprechende Zusammenfügen von heapgeordneten Bäumen erfolgt bei F-Heaps immer dann, wenn eine Delete-Min-Operation ausgeführt wird.

Das *Entfernen* des Minimalknotens (*Delete Min*) eines F-Heaps h geschieht folgendermaßen: Entferne den Minimalknoten aus der Wurzelliste von h und bilde eine neue Wurzelliste durch Einhängen der Liste der Söhne des Minimalknotens an Stelle des Minimalknotens in die Wurzelliste. (Das ist in konstanter Zeit möglich, wenn man die Vaterzeiger der in die Wurzelliste neu aufgenommenen Knoten erst beim anschließenden Durchlaufen der Wurzelliste adjustiert.) Anschließend werden so lange je zwei heapgeordnete Bäume, deren Wurzeln denselben Rang haben, zu einem neuen heapgeordneten Baum verschmolzen, bis eine Wurzelliste entstanden ist, deren sämtliche heapgeordneten Bäume verschiedenen Rang haben. Beim Verschmelzen zweier Bäume entsteht ein heapgeordneter Baum, dessen Wurzel einen um eins erhöhten Rang hat und dessen Markierungsfeld auf „unmarkiert“ gesetzt wird. Beim Durchlaufen der Wurzelliste und Verschmelzen von Bäumen merkt man sich zugleich die Wurzel des Baumes mit dem bislang minimalen Schlüssel. Am Ende wird dieser der Minimalknoten des resultierenden F-Heaps; man liefert als Ergebnis einen Zeiger auf diesen Knoten ab.

Die Operation Delete Min verlangt, daß man in einer Liste von Wurzeln von heapgeordneten Bäumen immer wieder Knoten vom selben Rang findet, die dann verschmolzen werden. Das kann man mit Hilfe eines Rang-Arrays erreichen, d.h. eines linearen Feldes, das mit den Rängen von 0 bis zum maximal möglichen Rang indiziert ist und Zeiger auf die Wurzeln heapgeordneter Bäume enthält. Zu jedem Rang enthält das Rang-Array höchstens einen Zeiger; anfangs ist das Rang-Array leer, d.h. es enthält noch keinen Zeiger. Dann durchläuft man die Wurzelliste, also die Liste der heapgeordneten Bäume, die verschmolzen werden sollen. Trifft man in dieser Liste auf einen Baum B mit Wurzel vom Rang r, versucht man, im Rang-Array einen Zeiger auf diesen Baum B an Position r einzutragen. Ist dort bereits ein Zeiger auf einen Baum B' (mit Wurzel vom gleichen Rang r) eingetragen, fügt man B und B' zu einem Baum mit Wurzel vom Rang $r+1$ zusammen und versucht, einen Zeiger auf diesen Baum an Position $r+1$ im Rang-Array einzutragen; der Eintrag an Position r im Rang-Array wird gelöscht. Jedes Element der Wurzelliste wird so genau einmal betrachtet, und am Ende enthält das Rang-Array für jeden Rang höchstens einen Zeiger auf eine Wurzel eines heapgeordneten Baumes. (Das Rang-Array kann dann wieder gelöscht werden.) Jetzt sollte auch der Zusammenhang mit den im Abschnitt 6.1.4 behandelten Binomial Queues klar sein. Man verschiebt einfach die der Addition von Dualzahlen entsprechenden Operationen an heapgeordneten Bäumen von der Verschmelze-Operation zur Delete-Min-Operation. Das hat den großen Vorteil, daß man zugleich mit der Ausführung der notwendigen Verschmelze-Operationen an heapgeordneten Bäumen auch das neue Minimalelement bestimmen kann.

Genauer gilt offenbar folgendes: Beginnt man mit einem anfangs leeren F-Heap und führt eine beliebige Folge von Einfüge-, Access-Min-, Meld- und Delete-Min-Operationen aus, so sind die Bäume in den Wurzellisten sämtlicher durch die Operationsfolge erzeugten F-Heaps stets Binomialbäume. Am Ende einer Delete-Min-Operation bilden die Bäume in der Wurzelliste des F-Heaps sogar eine Binomial Queue.

Bevor wir die Anzahl der zur Ausführung einer Delete-Min-Operation erforderlichen Schritte bestimmen, geben wir noch an, wie der Schlüssel eines Elementes *herabgesetzt* und wie ein Element aus einem F-Heap *entfernt* werden kann, das nicht das Minimalelement ist.

Um einen Schlüssel eines Knotens p eines F-Heaps h *herabzusetzen*, trennen wir p von seinem Vater φp ab und nehmen p mit dem herabgesetzen Schlüssel in die Wurzelliste des F-Heaps auf. Natürlich müssen wir auch den Rang von φp um 1 erniedrigen. Ist der herabgesetzte Schlüssel von p kleiner als der des Minimalelementes von h, machen wir p zum neuen Minimalelement.

Diese Veränderungen sind sämtlich in konstanter Zeit ausführbar. Im allgemeinen ist damit die Operation des Herabsetzens oder Entfernens eines Schlüssels aber noch nicht zu Ende. Wir wollen nämlich verhindern, daß ein Knoten mehr als zwei Söhne verliert, wenn auf diese Weise ein Knoten abgetrennt wird. (Denn dann könnte der heapgeordnete Baum zu „dünn“ werden.) Um das zu erreichen, benutzen wir die Markierung. Wir hatten einen Knoten als unmarkiert gekennzeichnet, wenn er Wurzel eines heapgeordneten Baumes geworden war, der durch Verschmelzen zweier Bäume mit Wurzeln vom gleichen Rang entstand. Wird nun im Verlauf einer Decrease-key- oder Delete-Operation p von seinem Vater φp abgetrennt und ist φp unmarkiert, so setzen wir φp auf markiert. Ist aber φp bereits markiert, so bedeutet das: φp hat bereits einen seiner Söhne verloren. In diesem Fall trennen wir nicht nur p von φp ab, sondern trennen auch

φ*p* von dessen Vater φφ*p* ab, usw., bis wir auf einen unmarkierten Knoten stoßen, der dann markiert wird, falls er nicht in der Wurzelliste auftritt. Alle abgetrennten Knoten werden in die Wurzelliste des F-Heaps aufgenommen. Obwohl wir, um den Schlüssel eines Knotens *p* herabzusetzen oder *p* zu entfernen, eigentlich nur *p* von seinem Vater abtrennen wollten, weil an dieser Stelle ein Verstoß gegen die Heap-Ordnung vorliegen könnte, kann das Abtrennen von *p* von φ*p* eine ganze Kaskade von weiteren Abtrennungen auslösen. Bevor wir uns überlegen, wieviele solcher indirekter Abtrennungen von Knoten (*cascading cuts*) vorkommen können, betrachten wir ein Beispiel. Nehmen wir an, daß in dem heapgeordneten Baum von Abbildung 6.13 der Schlüssel 31 auf 5 herabgesetzt werden soll und daß in dem Baum die Knoten 17, 13 und 7 (durch einen $*$) markiert sind, also bereits einen Sohn verloren haben. Dann führt das Abtrennen des Knotens 31 von seinem Vater dazu, daß auch 17, 13 und 7 abgetrennt werden, und man erhält die in Abbildung 6.14 gezeigte Liste von Bäumen.

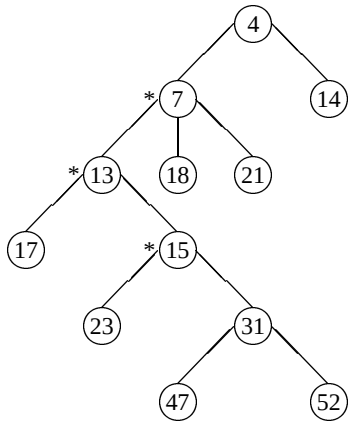

Abbildung 6.13

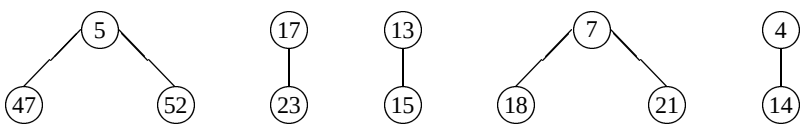

Abbildung 6.14

Das *Entfernen* eines Knotens *p*, der nicht das Minimalelement von *h* ist, kann wie folgt durchgeführt werden: Zunächst wird der Schlüssel von *p* auf einen Wert her-

abgesetzt, der kleiner als alle übrigen Schlüsselwerte in h ist. Anschließend wird die Operation Delete Min ausgeführt.

Daß die über die Markierung von Knoten gesteuerte Regel „Mache Knoten, die zwei Söhne verloren haben, zu Wurzeln“ wirklich verhindert, daß die in Wurzellisten von F-Heaps auftretenden Bäume zu „dünn“ werden, zeigen die folgenden Sätze.

Lemma 6.1 *Sei p ein Knoten eines F-Heaps h. Ordnet man die Söhne von p in der zeitlichen Reihenfolge, in der sie an p (durch Zusammenfügen) angehängt wurden, so gilt: Der i-te Sohn von p hat mindestens Rang $i-2$.*

Zum Beweis nehmen wir an, p habe r Söhne. Es ist möglich, daß p schon mehr als r Söhne gehabt hat und davon einige wieder durch Abtrennen verloren hat. Ordnet man die noch vorhandenen r Söhne von p der zeitlichen Reihenfolge nach, in der sie an p angehängt wurden, so muß gelten: Als der i-te Sohn an p angehängt wurde (durch Verschmelzen zweier Wurzeln vom gleichen Rang), müssen sowohl p als auch sein i-ter Sohn wenigstens Rang $i-1$ gehabt haben, und beide natürlich denselben Rang. Der i-te Sohn kann später höchstens einen Sohn verloren haben, denn andernfalls wäre er von p nach der oben angegebenen Regel abgetrennt worden. □

Lemma 6.2 *Jeder Knoten p vom Rang k eines F-Heaps h ist Wurzel eines Teilbaumes mit wenigstens F_{k+2} Knoten.*

Zum Beweis definieren wir

$$S_k \quad = \quad \text{Minimalzahl von Nachfolgern eines Knotens } p \text{ vom Rang } k \text{ in einem F-Heap (einschließlich } p\text{)}.$$

Ein Knoten mit Rang 0 hat keinen Sohn, ein Knoten mit Rang 1 hat mindestens einen Sohn, also $S_0 = 1$, $S_1 = 2$. Betrachten wir jetzt also einen Knoten p vom Rang k. Wir können die k Söhne von p in der Reihenfolge ordnen, in der sie an p angehängt wurden. Der erste Sohn von p kann Rang 0 haben; für alle anderen gilt Lemma 6.1; zählt man noch p selbst hinzu, so folgt:

$$S_k \geq 2 + \sum_{i=0}^{k-2} S_i, \text{ für } k \geq 2. \tag{6.1}$$

Aus der Definition der Fibonacci-Zahlen ($F_0 = 0, F_1 = 1, F_{k+2} = F_{k+1} + F_k$) folgt sofort:

$$F_{k+2} = 2 + \sum_{i=2}^{k} F_i, \text{ für } k \geq 2. \tag{6.2}$$

Aus (6.1) und (6.2) leitet man durch vollständige Induktion über k her:

$$S_k \geq F_{k+2}, \text{ für } k \geq 0.$$

□

Aufgrund von Lemma 6.2 haben Fredman und Tarjan [60] den Namen Fibonacci-Heap eingeführt. Wir wissen bereits, vgl. Abschnitt 3.2.3, daß die Fibonacci-Zahlen exponentiell (mit dem Faktor $1.618\ldots$) wachsen. Vergleichen wir nun F-Heaps und Binomial Queues: Binomial Queues bestehen aus Binomialbäumen; jeder Binomialbaum B_j mit Wurzel vom Rang j hat 2^j Knoten. Ein in der Wurzelliste eines F-Heaps auftretender Baum muß ebenfalls eine Anzahl von Knoten haben, die exponentiell mit dem Rang, d.h. mit der Anzahl der Söhne der Wurzel wächst. Genauer kann man aus Lemma 6.2 folgern, daß F-Heaps mit Wurzeln vom Rang 0, 1, 2, 3, 4 ... und minimaler Knotenzahl die in Abbildung 6.15 gezeigte Struktur haben müssen. (Der in Abbildung 6.13 gezeigte heapgeordnete Baum kann also in der Wurzelliste eines F-Heaps nicht auftreten!)

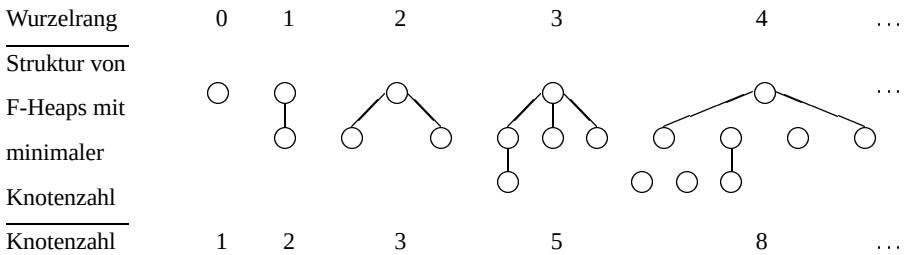

Abbildung 6.15

Umgekehrt folgt aus Lemma 6.2 natürlich auch, daß jeder Knoten eines F-Heaps mit insgesamt N Knoten einen Rang $k \leq 1.44\ldots \log_2 N$ hat. Das hat insbesondere zur Folge, daß durch Entfernen des Minimalknotens eines F-Heaps mit N Knoten die Wurzelliste höchstens um $O(\log N)$ Wurzeln heapgeordneter Bäume verlängert wird.

Wir wollen jetzt die Anzahl der Schritte (die Zeit oder die Kosten) nach oben hin abschätzen, die zur Ausführung der Operationen an F-Heaps erforderlich sind. Dabei interessieren wir uns für die Kosten pro Operation, gemittelt über eine beliebige Operationenfolge, beginnend mit einem anfangs leeren F-Heap. Schwierig ist allein die Abschätzung der Zahl der Verschmelze-Operationen nach Entfernen des Minimalknotens bei einer Delete-Min-Operation und der Zahl der indirekten Abtrennungen (cascading cuts) von Knoten nach einer Decrease-Key- oder Delete-Operation.

Es ist intuitiv klar, daß die Zahl der Verschmelze-Operationen mit der Zahl der Knoten in der Wurzelliste eines F-Heaps zusammenhängt. Jede Verschmelze-Operation verkürzt die Wurzelliste. Ebenso ist klar, daß die Zahl der markierten Knoten und damit die Zahl der indirekten Abtrennungen mit der Zahl der Decrease-Key- und Delete-Operationen zusammenhängen muß. Eine Markierung ist stets Folge einer solchen Operation.

Zur Abschätzung der wirklichen Gesamtkosten für eine Folge von Operationen an F-Heaps führen wir eine amortisierte Worst-case-Analyse durch und benutzen das Bankkonto-Paradigma aus Abschnitt 3.3. Wir ordnen jedem Bearbeitungszustand, der

nach Ausführung eines Anfangsstücks einer gegebenen Folge von Operationen erreicht wird, einen nichtnegativen *Kontostand* und der i-ten Operation der Folge eine *amortisierte Zeit* a_i zu: a_i ist die wirkliche Zeit t_i zur Ausführung der i-ten Operation zuzüglich dem Kontostand nach Ausführung der i-ten Operation minus dem Kontostand vor Ausführung der i-ten Operation. Die zur Durchführung einer Folge von Operationen erforderliche Gesamtzeit kann dann durch die gesamte amortisierte Zeit minus Nettozuwachs des Kontos abgeschätzt werden (vgl. dazu Abschnitt 3.3). Man kann den Kontostand als eine Menge von Zahlungseinheiten auffassen, mit denen man die zur Ausführung von Operationen anfallenden Kosten begleichen kann.

Wir ordnen einem aus dem anfangs leeren F-Heap durch eine Folge von Operationen erzeugten F-Heap h einen Kontostand $bal(h)$ wie folgt zu:

$$\begin{aligned} bal(h) \quad = \quad & \text{Anzahl Bäume in der Wurzelliste von } h \\ & + 2\cdot(\text{Anzahl markierter Knoten in } h, \text{ die} \\ & \qquad \text{nicht in der Wurzelliste auftreten}) \end{aligned}$$

Die amortisierte Zeit zur Ausführung einer Einfüge-, Access-Min- und Meld-Operation ist $O(1)$. Denn die Einfüge-Operation erhöht lediglich die Zahl der Bäume in der Wurzelliste um 1; Access Min und Meld lassen die Gesamtzahl der Bäume und der markierten Knoten unverändert.

Um die amortisierten Kosten einer Delete-Min-Operation zu bestimmen, setzen wir zunächst voraus, daß jedes Verschmelzen zweier Bäume der Wurzelliste zu einem Baum genau eine Kosteneinheit verursacht, also durch das Verschwinden eines Baumes aus der Wurzelliste aufgewogen wird. Wir berücksichtigen daher bei der weiteren Analyse die Kosten des Verschmelzens nicht mehr. Die Anzahl der nicht in der Wurzelliste auftretenden markierten Knoten bleibt bei einer Delete-Min-Operation unverändert oder nimmt sogar ab, nämlich dann, wenn markierte Knoten in die Wurzelliste aufgenommen werden. Wir können uns daher bei der Untersuchung der Kontostandsänderung auf die Änderung der Anzahl Bäume in der Wurzelliste von h beschränken. Sei $w(h)$ diese Anzahl vor Entfernen des Minimums. Dann betragen die tatsächlichen Kosten der Delete-Min-Operation (ohne Berücksichtigung des Verschmelzens) gerade $O(\log N + w(h))$, da die — um maximal $O(\log N)$ Knoten vergrößerte — Wurzelliste von h einmal durchlaufen wird, um Bäume gleichen Ranges zu verschmelzen. Nach dem Verschmelzen enthält die Wurzelliste von h höchstens noch $O(\log N)$ Knoten. (Nach Ausführen einer Delete-Min-Operation ist h eine Binomial Queue; da h N Knoten enthält, besteht h aus höchstens $O(\log N)$ Bäumen.) Also sinkt der Kontostand von $O(w(h)) + 2\cdot$Anzahl markierter Knoten auf $O(\log N) + 2\cdot$Anzahl markierter Knoten. Damit sind die amortisierten Kosten einer Delete-Min-Operation, also die tatsächlichen Kosten plus die Kontostandsänderung, gerade $O(\log N + w(h)) + O(\log N) - O(w(h)) = O(\log N)$.

Um die amortisierten Kosten einer Decrease-Key-Operation zu bestimmen, setzen wir voraus, daß jedes direkte und indirekte Abtrennen eines Knotens eine Kosteneinheit verursacht. Wird ein Knoten von seinem unmarkierten Vater abgetrennt, in die Wurzelliste aufgenommen und der Vater markiert, so verursacht dies eine Kosteneinheit. Zugleich nimmt der Kontostand um drei Einheiten zu. Die amortisierten Kosten dieser Operation sind also in $O(1)$. Nehmen wir nun an, ein Knoten p wird von einem markierten Vater φp abgetrennt; dann muß auch φp von dessen Vater $\varphi\varphi p$ abgetrennt werden usw., bis schließlich ein markierter Knoten von einem unmarkierten abgetrennt

wird. Jede Abtrennoperation, außer der letzten, verursacht eine Kosteneinheit, erhöht die Zahl der Bäume in der Wurzelliste um 1 und vermindert die Zahl der markierten Knoten, die zu $bal(h)$ beitragen, um 1; die amortisierten Kosten dafür sind also 0. Die letzte Abtrennoperation erhöht die Zahl der markierten Knoten um 1 und die Zahl der Bäume in der Wurzelliste um 1; sie verursacht ebenfalls eine Kosteneinheit. Insgesamt sind auch in diesem Fall die amortisierten Kosten in $O(1)$.

Weil eine Delete-Operation eine Decrease-Key-Operation mit anschließender Delete-Min-Operation ist, folgt sofort, daß auch die amortisierten Kosten einer Delete-Operation in $O(\log N)$ sind. Wir können unsere Überlegungen damit in folgendem Satz zusammenfassen.

Satz 6.1 *Führt man, beginnend mit dem anfangs leeren F-Heap, eine beliebige Folge von Operationen an Priority Queues aus, dann ist die dafür insgesamt benötigte Zeit beschränkt durch die gesamte amortisierte Zeit; die amortisierte Zeit einer einzelnen Delete-Min- und Delete-Operation ist in $O(\log N)$, die amortisierte Zeit aller anderen Operationen in $O(1)$.*

Wir können F-Heaps verwenden zur Implementation von Dijkstras Algorithmus zur Lösung des Single-source-shortest-paths-Problems für einen Graphen mit n Knoten und m Kanten. Der Algorithmus hat dann die Laufzeit $O(n \log n + m)$. Auch zur Implementation vieler anderer Algorithmen kann man F-Heaps verwenden.

Kürzlich wurden Relaxed Heaps in [39] als Alternative zu F-Heaps angegeben. Für sie gelten dieselben Schranken für die amortisierten Worst-case-Kosten zur Ausführung der Operationen an Priority Queues wie für F-Heaps. Für eine Variante von Relaxed Heaps erhält man aber dieselben Zeitschranken sogar für jeweils eine einzelne Operation im schlechtesten Fall. Die Struktur von Relaxed Heaps und die für sie erklärten Algorithmen zur Ausführung der Operationen an Priority Queues sind jedoch erheblich komplexer als für F-Heaps und übersteigen den Rahmen dieses Buches.

6.2 Union-Find-Strukturen

In einer ganzen Reihe von Algorithmen insbesondere aus dem Bereich der Algorithmen auf Graphen tritt als Teilaufgabe das Problem auf, für eine Menge von Objekten, z.B. für die Knoten oder Kanten eines Graphen, eine Einteilung in Äquivalenzklassen vorzunehmen. Man beginnt mit einer sehr feinen Einteilung, die sukzessive durch Vereinigen der Mengen vergröbert wird. Man kann diese Teilaufgabe als einen Spezialfall des Mengenmanipulationsproblems auffassen, der dadurch charakterisiert ist, daß auf einer Kollektion von Mengen die folgenden Operationen ausführbar sind.

Make-set(e,i) schafft eine neue Menge i mit e als einzigem Element; i ist also der Name der Menge; es wird vorausgesetzt, daß das Element e neu ist, also in keiner anderen Menge der Kollektion vorkommt.

Find(x) liefert den Namen der Menge, die das Element x enthält.

Union(i,j,k) vereinigt die Mengen i und j zu einer neuen Menge mit Namen k. i und j werden aus der Kollektion von Mengen entfernt und k aufgenommen; es wird angenommen, daß i und j verschieden sind.

Wegen der bei der Operation Make-set gemachten Voraussetzung besteht die durch eine beliebige Folge dieser Operationen erzeugte Kollektion von Mengen stets aus paarweise disjunkten Mengen. Da es auf die Namen der Mengen nicht ankommt, kann man sie auch ganz unterdrücken und jeder Menge einen eindeutig bestimmten Repräsentanten, ein sogenanntes *kanonisches Element*, zuordnen. Das kanonische Element der durch *Make-set*(e,i) geschaffenen Menge ist natürlich e. Die *Find*(x)-Operation liefert das kanonische Element der Menge, in der x liegt. Der durch Vereinigung von zwei Mengen i und j entstehenden Menge kann man willkürlich ein neues kanonisches Element zuordnen, z.B. immer das kanonische Element von i. Wir verwenden daher in der Regel einfach die Operationen *Make-set*(e), *Find*(x), *Union*(e,f) statt der oben angegebenen mit der offensichtlichen Bedeutung.

Das Problem, eine Datenstruktur zur Repräsentation einer Kollektion von paarweise disjunkten Mengen und Algorithmen zur Ausführung der Operationen *Make-set*, *Find* und *Union* auf dieser Kollektion zu finden, heißt das *Union-Find-Problem*.

Bevor wir mögliche Lösungen des Union-Find-Problems diskutieren, wollen wir ein einziges Beispiel für einen Algorithmus angeben, bei dessen Implementation man Lösungen des Union-Find-Problems verwenden kann.

6.2.1 Kruskals Verfahren zur Berechnung minimaler spannender Bäume

Wir lösen das Problem der Berechnung *minimaler spannender Bäume* für zusammenhängende, ungerichtete, gewichtete Graphen. Für eine ausführliche Behandlung dieses Problems verweisen wir auf das Kapitel 8.

Gegeben sei ein Graph G mit Knotenmenge V und Kantenmenge E. Jeder Kante $e \in E$ sei eine reelle Zahl $c(e)$ als Kosten (engl.: cost) zugeordnet. Der Graph sei ungerichtet und zusammenhängend, d.h. je zwei Knoten des Graphen seien durch mindestens einen (ungerichteten) Kantenzug miteinander verbunden. Wir verzichten wieder auf eine genaue, formale Definition. Man stelle sich den Graphen G einfach als Menge von Orten vor, die durch in beide Richtungen befahrbare Straßen miteinander verbunden sind. Die Kosten einer Kante $e = (v,w)$ ist dann die Länge der Straße e, die die Orte v und w miteinander verbindet.

Ein minimaler spannender Baum T (*minimum spanning tree*, kurz: *MST*) für G besteht aus allen Knoten V von G, enthält aber nur eine Teilmenge E' der Kantenmenge E von G, die alle Knoten des Graphen miteinander verbindet und die Eigenschaft hat, daß die Summe aller Kantengewichte den minimal möglichen Wert hat unter allen Teilmengen von E, die alle Knoten des Graphen G miteinander verbinden.

Im Bild der Orte und Straßen bedeutet die Konstruktion eines *MST* das Herausfinden eines Teilstraßennetzes kürzester Gesamtlänge, das noch alle Orte miteinander verbindet.

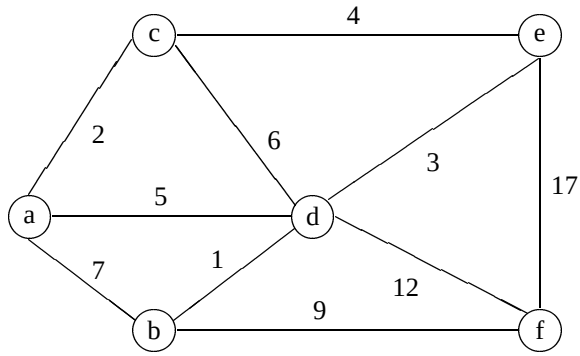

Abbildung 6.16

Als Beispiel betrachten wir den Graphen in Abbildung 6.16; das ist derselbe Graph wie in Abbildung 6.1, jedoch sind jetzt alle Kanten ungerichtet.

Abbildung 6.17 zeigt einen *MST* für diesen Graphen.

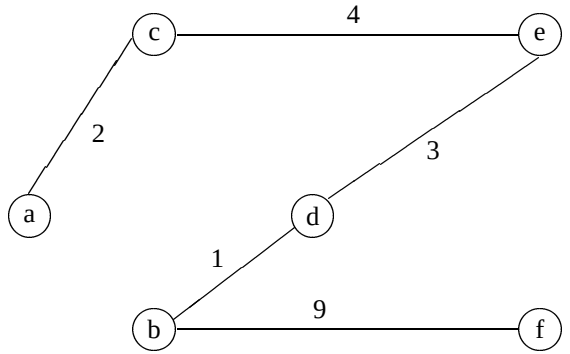

Abbildung 6.17

Es gibt zahlreiche Verfahren zur Konstruktion eines *MST*. Wir skizzieren ein Verfahren, das auf J. Kruskal zurückgeht [96]. Die Idee des Verfahrens von Kruskal besteht darin, einen Wald von Teilbäumen des *MST* sukzessive zum *MST* zusammenwachsen zu lassen. Man beginnt mit Teilbäumen, die sämtlich nur aus je genau einem Knoten des gegebenen Graphen $G = (V, E)$ bestehen. Dann werden immer wieder je zwei verschiedene Teilbäume durch Hinzunahme einer Kante minimalen Gewichts zu einem verbunden, bis schließlich nur noch ein einziger Baum, eben der *MST*, übrigbleibt. Wir wollen hier wieder nicht die Frage der Korrektheit des Verfahrens diskutieren (siehe

dazu Abschnitt 8.6), sondern nur zeigen, wie Lösungen des Union-Find-Problems zur Implementation des Verfahrens verwendet werden können.

Das Verfahren von Kruskal geht aus von einer Kollektion K von einelementigen Knotenmengen. Die Knotenmengen werden sukzessive vergrößert, indem je zwei Mengen der Kollektion vereinigt werden, wenn sie durch eine Kante minimalen Gewichts miteinander verbunden werden können. Das Verfahren endet, wenn die Kollektion nur noch aus einer einzigen Menge (der Knotenmenge V des gegebenen Graphen) besteht. Etwas genauer kann das Verfahren wie folgt beschrieben werden:

```
procedure MST ((V,E) : Graph);
{berechnet zu einem zusammenhängenden, ungerichteten, gewichteten
 Graphen G = (V,E) einen minimalen spannenden Baum T = (V,E')}
begin
  E' := ∅;
  K := ∅;
  bilde Priority Queue Q aller Kanten in E mit den Kantengewichten
   als Prioritätsordnung;
  for all v ∈ V do Make-set (v);
  {jetzt besteht K aus allen Mengen {v}, v ∈ V}
  while K enthält mehr als eine Menge do
    begin
      (v,w) := min(Q); deletemin(Q);
      if Find(v) ≠ Find(w) then
        begin
          Union(v0,w0), mit v0 = Find(v), w0 = Find(w);
          E' := E' ∪ {(v,w)}
        end
    end
end
```

Wir verfolgen den Ablauf des Verfahrens am Beispiel des Graphen aus Abbildung 6.16. Anfangs besteht die Kollektion K aus den einelementigen Mengen $\{a\}$, $\{b\}$, $\{c\}$, $\{d\}$, $\{e\}$, $\{f\}$. Die Kante mit kleinstem Gewicht ist (b,d). Also wird diese Kante zum Baum T hinzugenommen, und die zwei Mengen, die b und d enthalten, werden zu $\{b,d\}$ vereinigt. Dann wird die Kante (a,c) gewählt, $\{a\}$ und $\{c\}$ werden zu $\{a,c\}$ vereinigt und (a,c) wird zu T hinzugenommen. Als nächste wird die Kante (d,e) gewählt; weil d und e in verschiedenen Mengen der Kollektion K sind, werden die Mengen zu $\{b,d,e\}$ vereinigt und (d,e) in T aufgenommen. Dann wird die Kante (c,e) ausgewählt; wieder sind c und e in verschiedenen Mengen der Kollektion, so daß durch Vereinigung dieser Mengen $\{a,b,c,d,e\}$ entsteht und (c,e) in T aufgenommen wird. Die noch nicht betrachtete Kante mit kleinstem Gewicht ist (a,d); a und d liegen aber bereits in derselben Menge der Kollektion, so daß (a,d) nicht in T aufgenommen wird und keine Mengen von K vereinigt werden. Das entsprechende gilt für (c,d) und (a,b). Die nächste betrachtete Kante ist (b,f); sie wird in T aufgenommen und die beiden Mengen, die b und f enthalten, zu einer Menge (der gesamten Knotenmenge) verschmolzen. Es müssen also keine weiteren Kanten mehr betrachtet werden. Tabelle 6.2 faßt alle Schritte nochmals zusammen.

Kollektion K	nächste betrachtete Kante	Hinzunahme zu T
$\{a\},\{b\},\{c\},\{d\},\{e\},\{f\}$	(b,d)	*ja*
$\{a\},\{b,d\},\{c\},\{e\},\{f\}$	(a,c)	*ja*
$\{a,c\},\{b,d\},\{e\},\{f\}$	(d,e)	*ja*
$\{a,c\},\{b,d,e\},\{f\}$	(c,e)	*ja*
$\{a,b,c,d,e\},\{f\}$	(a,d)	*nein*
	(c,d)	*nein*
	(a,b)	*nein*
	(b,f)	*ja*
$\{a,b,c,d,e,f\}$		

Tabelle 6.2

6.2.2 Vereinigung nach Größe und Höhe

Die einfachste Möglichkeit zur Lösung des Union-Find-Problems besteht darin, jede Menge der Kollektion K durch einen (nichtsortierten) Baum beliebiger Ordnung zu repräsentieren; die Knoten des Baumes sind die Elemente der Menge. Es genügt, zu verlangen, daß die Wurzel des Baumes das kanonische Element der Menge enthält oder, falls man explizit mit Namen operiert, daß an der Wurzel der Name der Menge vermerkt ist. Jeder Knoten im Baum enthält einen Zeiger auf seinen Vater; die Wurzel zeigt auf sich selbst und enthält gegebenenfalls den Namen der Menge. Abbildung 6.18 zeigt ein Beispiel für eine Kollektion von zwei Mengen, die im Verlauf des Verfahrens von Kruskal auftritt.

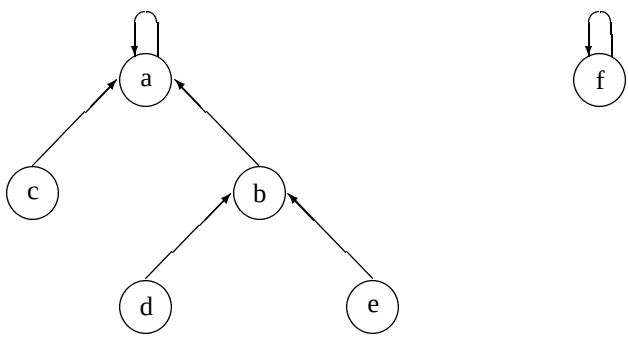

Abbildung 6.18

Wir nehmen an, daß man auf die Elemente der Mengen, also auf die Knoten in den die Menge repräsentierenden Bäumen, direkt zugreifen kann. Es liegt nahe, dazu einfach ein mit sämtlichen Elementen indiziertes Array zu verwenden, das zu jedem Element einen Verweis auf dessen Vater enthält. Diese Idee liefert eine sehr kompakte, zeigerlose Realisierung von Wäldern von Bäumen.

Im Falle des Beispiels aus Abbildung 6.18 nehmen wir also an, daß folgende Vereinbarungen gegeben sind:

type *element* = (*a*,*b*,*c*,*d*,*e*,*f*);
var *p* : **array** [*element*] **of** *element*

Die in Abbildung 6.18 gezeigte Situation wird durch folgende Belegung des Arrays *p* realisiert:

x	:	a	b	c	d	e	f
$p[x]$	:	a	a	a	b	b	f

Es ist klar, wie man die gewünschten Operationen ausführen kann:

Make-set(x) liefert einen Baum mit einem einzigen Knoten x, dessen Vaterverweis auf sich selbst zurückweist.

Zur Ausführung von *Find*(x) folgt man ausgehend vom Knoten x Vaterverweisen, bis man bei der Wurzel angelangt ist. Das merkt man daran, daß sich in der durchlaufenen Knotenfolge ein Knoten wiederholt. Sobald man bei der Wurzel angelangt ist, gibt man das Wurzelelement als kanonisches Element der Menge aus, oder, falls man explizit mit Namen operiert, den bei der Wurzel gespeicherten Namen.

Zur Ausführung einer Vereinigungsoperation *Union*(e, f) schaffen wir einen neuen Baum dadurch, daß wir (willkürlich) den Knoten f auf e zeigen lassen, also e zum kanonischen Element der durch Vereinigung neu entstehenden Menge machen.

Denken wir uns ein mit allen Elementen indiziertes Array p als global vereinbarte Variable gegeben, so kann man die Operationen wie folgt programmtechnisch realisieren.

```
var p: array [element] of element;

procedure Make-set (x : element);
begin p[x] := x end

procedure Union (e, f : element);
  begin p[f] := e end

function Find (x : element) : element;
var y : element;
begin y := x;
  while p[y] ≠ y do y := p[y];
  Find := y
end
```

Make-set und *Union* sind in konstanter Zeit ausführbar; die Anzahl der Schritte zur Ausführung einer *Find*(x)-Operation ist proportional zur Anzahl der Knoten auf dem Pfad vom Knoten x zur Wurzel des Baumes. Weil wir keinerlei Bedingung an die Vereinigung zweier Bäume gestellt haben, kann der Aufwand für eine einzelne Find-Operation groß werden. Man betrachte dazu die folgende Operationsfolge:

$$\begin{array}{ll} \textit{Make-set}(i); & i = 1,\ldots,N \\ \textit{Union}(i-1,i); & i = N,\ldots,2 \\ \textit{Find}(N) & \end{array}$$

Offenbar wird ausgehend von N Bäumen mit je einem Knoten zunächst ein degenerierter Baum der Höhe N erzeugt, so daß die Find-Operation $\Omega(N)$ Schritte benötigt.

Es gibt zwei naheliegende Strategien, mit denen man verhindern kann, daß durch iteriertes Vereinigen von Bäumen zu linearen Listen degenerierte Bäume entstehen können: Vereinigung nach *Größe* und Vereinigung nach *Höhe*.

Wir haben nämlich beim oben angegebenen naiven Vereinigungsverfahren willkürlich festgesetzt, daß die durch eine Vereinigungsoperation *Union*(e,f) entstehende Menge e als kanonisches Element haben soll. Natürlich hätten wir ebensogut f als kanonisches Element wählen können und dazu den Knoten e auf f zeigen lassen. Man merkt sich nun jeweils an der Wurzel die Größe, d.h. die gesamte Knotenzahl, bzw. die Höhe des Baumes und verfährt wie folgt.

Um zwei Bäume mit Wurzeln e und f zu vereinigen, macht man die Wurzel des Baumes mit kleinerer Größe (bzw. geringerer Höhe) zum direkten weiteren Sohn des Baumes mit der größeren Größe (bzw. Höhe). Falls die Größen von e und f (bzw. die Höhen) gleich sind, kann man e oder f zur Wurzel machen. Je nachdem, ob e oder f die Wurzel geworden ist, wird e oder f kanonisches Element der durch Vereinigung entstandenen Menge.

Es dürfte klar sein, wie man diese Strategien programmtechnisch realisieren kann. Die Funktion *Find* bleibt in jedem Fall unverändert. Wir geben die geänderten Prozeduren zur Ausführung einer Make-set- und Union-Operation für den Fall der Vereinigung nach Größe an. Dazu setzen wir voraus, daß ein weiteres Array *Größe* vereinbart ist, das zu jedem kanonischen Element eines Baumes die Anzahl der Elemente im Baum liefert.

```
procedure Make-set (x : element);
begin
  p[x] := x;
  Größe[x] := 1
end

procedure Union (e, f : element);
begin
  if Größe[e] < Größe[f] then vertausche(e,f);
  {jetzt ist e kanonisches Element der größeren Menge}
  p[f] := e;
  Größe[e] := Größe[f] + Größe[e]
end
```

Make-set und Union sind natürlich immer noch in konstanter Zeit ausführbar.

Lemma 6.3 *Das Verfahren Vereinigung nach Größe konserviert die folgende Eigenschaft von Bäumen: Ein Baum mit Höhe h hat wenigstens 2^h Knoten.*

Zum Beweis nehmen wir an, daß T_1 und T_2 Bäume mit den Größen $g(T_1)$ und $g(T_2)$ sind, die vereinigt werden sollen; h_1 und h_2 seien die Höhen von T_1 und T_2. Der durch Vereinigung von T_1 und T_2 entstehende Baum $T_1 \cup T_2$ hat die in Abbildung 6.19 dargestellte Gestalt. D.h. wir nehmen ohne Einschränkung an, daß $g(T_1) \geq g(T_2)$ ist. Nach Voraussetzung hat T_i wenigstens 2^{h_i}, $i = 1,2$, Knoten.

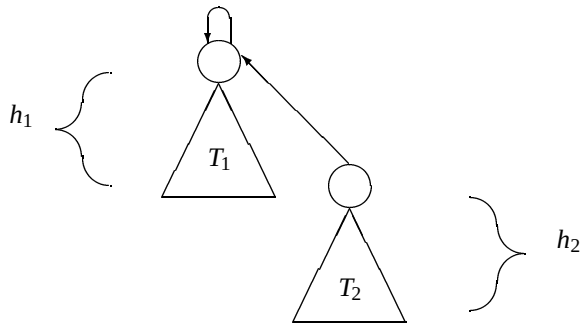

Abbildung 6.19

Fall 1: Höhe$(T_1 \cup T_2) = \max(\{h_1, h_2\})$.
Dann hat $T_1 \cup T_2$ trivialerweise wenigstens $2^{\text{Höhe}(T_1 \cup T_2)}$ Knoten.
Fall 2: Die Höhe des durch Vereinigung entstandenen Baumes ist gegenüber $max(\{h_1, h_2\})$ um 1 gewachsen. Aufgrund der von uns getroffenen Annahmen ist das nur möglich, wenn Höhe$(T_1 \cup T_2) = h_2 + 1$ ist. Wir müssen die Größe $g(T_1 \cup T_2)$ des durch Vereinigung von T_1 und T_2 entstandenen Baumes abschätzen. Es gilt:

$$g(T_1) \geq g(T_2) \geq 2^{h_2}, \text{ also}$$
$$g(T_1 \cup T_2) = g(T_1) + g(T_2) \geq 2 \cdot 2^{h_2} = 2^{\text{Höhe}(T_1 \cup T_2)}$$

□

Als unmittelbare Folgerung aus Lemma 6.3 erhält man: Wird das Verfahren Vereinigung nach Größe iteriert angewandt, beginnend mit einer Folge von N Bäumen mit je genau einem Knoten, die N einelementige Mengen repräsentieren, so haben alle entstehenden Bäume eine Höhe $h \leq \log_2 N$.

Vereinigung nach Größe garantiert also, daß eine Find-Operation höchstens $O(\log N)$ Schritte kosten kann. Dasselbe gilt auch für die Strategie der Vereinigung nach Höhe. Denn auch für dieses Verfahren gilt die Aussage von Lemma 6.3 entsprechend, wie man leicht nachprüft. Das Verfahren Vereinigung nach Höhe hat gegenüber dem Verfahren Vereinigung nach Größe den (kleinen) Vorteil, daß die für die kanonischen Elemente mitzuführende Höheninformation nicht so stark wächst wie die Größe der Bäume; man kommt mit $\log\log N$ statt $\log N$ Bits Zusatzinformation für jeden Baum aus, um diese Strategie zu implementieren.

6.2.3 Methoden der Pfadverkürzung

Vereinigung nach Größe oder Höhe garantiert, daß die zur Ausführung einer Find-Operation zu durchlaufende Folge von Kanten (Vaterverweisen) nicht zu lang wird. Eine sehr drastische weitere Verkürzung dieser Pfade würde man dadurch erhalten, daß man alle Knoten des einen Baumes direkt auf die Wurzel des anderen zeigen läßt. Das ist natürlich nicht besonders effizient, weil dann die Vereinigungsoperation nicht mehr in konstanter Zeit ausführbar ist, sondern so viele Schritte benötigt, wie die (zweite) Menge Knoten hat.

Eine andere Möglichkeit zur Verkürzung von Pfaden, die bei Find-Operationen durchlaufen werden müssen, ist, die bei einer Find-Operation durchlaufenen Knoten unmittelbar oder zumindest näher an die Wurzel zu hängen. Das verteuert zwar die gerade durchgeführte Find-Operation, zahlt sich aber für künftige Find-Operationen aus, weil die dann noch zu durchlaufenden Pfade kürzer werden. Die naheliegendste Methode dieser Art ist die *Kompressionsmethode*: Sämtliche bei Ausführung einer Find-Operation durchlaufenen Knoten werden direkt an die Wurzel gehängt.

Diese Methode verlangt aber, daß man bei Ausführung von $Find(x)$ den von x zur Wurzel führenden Pfad zweimal durchläuft, weil man einen Knoten natürlich erst dann an die Wurzel anhängen kann, wenn man die Wurzel kennt. Die Kompressionsmethode kann wie folgt implementiert werden.

```
function Find(x : element) : element;
var y,z,t : element;
begin
  y := x;
  while p[y] ≠ y do y := p[y];
  {jetzt ist y die Wurzel; alle Knoten auf dem Pfad von x nach y
   werden direkt an y angehängt}
  z := x;
  while p[z] ≠ y do
    begin t := z; z := p[z]; p[t] := y end;
  Find := y
end
```

Ein Beispiel für die Wirkung der Kompressionsmethode zeigt Abbildung 6.20. Dort sind die vor Ausführung von $Find(x)$ vorhandenen Vaterverweise durchgezogen und die danach vorhandenen für die Knoten auf dem Pfad von x zur Wurzel gestrichelt gezeichnet.

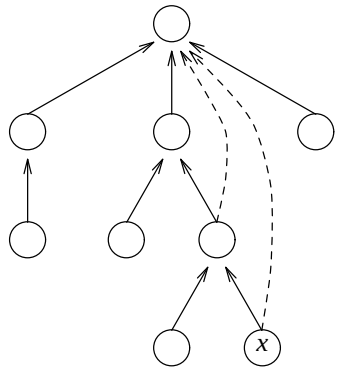

Abbildung 6.20

Die Analyse der Kompressionsmethode in Verbindung mit der Strategie Vereinigung nach Größe oder Vereinigung nach Höhe ist deshalb schwierig, weil die Kosten der Operationen von der Reihenfolge, in der sie ausgeführt werden, abhängen. Wir verweisen daher auf die Arbeit [182], in der die Kompressionsmethode und andere Methoden der Pfadverkürzung analysiert werden. Die Herleitung der kleinsten oberen Schranke für die amortisierten Worst-case-Kosten der Kompressionsmethode findet man auch in der Monographie von Tarjan [180].

Wir geben hier nur das Ergebnis der Analyse an. Sei m die Anzahl der Operationen und n die Anzahl der Elemente in allen Mengen. D.h. es werden n Make-set-Operationen und höchstens $n-1$ Union-Operationen ausgeführt und es ist $m \geq n$. Die Aussage über die zur Ausführung der m Operationen benötigte Anzahl von Schritten macht Gebrauch von einer sehr schwach wachsenden Funktion, der Inversen der Ackermannfunktion. Die Ackermannfunktion $A(i,j)$ ist für $i,j \geq 1$ wie folgt definiert:

$$\begin{aligned} A(1,j) &= 2^j, \text{ für } j \geq 1, \\ A(i,1) &= A(i-1,2), \text{ für } i \geq 2 \\ A(i,j) &= A(i-1,A(i,j-1)), \text{ für } i,j \geq 2 \end{aligned}$$

Die Inverse der Ackermannfunktion $\alpha(m,n)$ ist für $m \geq n \geq 1$ wie folgt definiert:

$$\alpha(m,n) = \min\{i \geq 1 \mid A(i, \lfloor m/n \rfloor) > \log n\}$$

Die bemerkenswerteste Eigenschaft der Ackermannfunktion ist ihr „explosives“ Wachstum. (Häufig wird in der ersten Definitionszeile der Ackermannfunktion $A(1,j) = j+1$ gesetzt und nicht, wie oben angegeben $A(1,j) = 2^j$. Das explosive Wachstum tritt jedoch auch dann ein, nur etwas später.) Weil A sehr schnell wächst, folgt umgekehrt, daß α sehr langsam wächst. Es ist beispielsweise $A(3,1) = A(2,2) = A(1,A(2,1)) = A(1,A(1,2)) = A(1,4) = 16$; also ist $\alpha(m,n) \leq 3$ für $n < 2^{16} = 65536$. $A(4,1) = A(2,16)$ ist bereits so riesig groß, daß $\alpha(m,n) \leq 4$ ist für alle praktisch auftretenden Werte von n und m.

Tarjan hat nun gezeigt: Benutzt man die Strategie Vereinigung nach Größe oder Vereinigung nach Höhe und benutzt man bei der Ausführung von Find-Operationen die Kompressionsmethode, so benötigt man zur Ausführung einer beliebigen Folge von $m \geq n$ Operationen $\Theta(m \cdot \alpha(m,n))$ Schritte. Die zur Ausführung einer einzelnen Operation in einer beliebigen Folge von Operationen erforderliche Schrittzahl ist also praktisch konstant.

Neben der Kompressionsmethode gibt es noch eine Reihe anderer Methoden zur Pfadverkürzung, die das Ziel verfolgen, bei Ausführung einer *Find*(x)-Operation den Pfad von x zur Wurzel nicht zweimal durchlaufen zu müssen. Wir geben zwei Methoden an, die asymptotisch dieselbe Laufzeit haben wie die Kompressionsmethode, vgl. [182].

Aufteilungsmethode (Splitting): Während der Ausführung einer Find-Operation teilt man den Suchpfad dadurch in zwei Pfade von etwa halber Länge auf, daß man jeden Knoten (mit Ausnahme des letzten und vorletzten) statt auf seinen Vater auf seinen Großvater zeigen läßt. Ein Beispiel zeigt Abbildung 6.21.

Die Funktion *Find* kann also wie folgt implementiert werden:

```
function Find(x : element) : element;
var x,t : element;
begin
  y := x;
  while p[p[y]] ≠ p[y] do
    begin
      t := y; y := p[y]; p[t] := p[p[t]]
    end
end
```

Halbierungsmethode (Halving): Während der Ausführung einer Find-Operation läßt man jeden zweiten Knoten auf seinen Großvater zeigen (mit Ausnahme der eventuell letzten Knoten). Man ändert also die Verweise für den 1., 3., 5., ... Knoten, und läßt die Verweise für den 2., 4., 6., ... unverändert. Auf diese Weise wird die Länge der Suchpfade für nachfolgende Find-Operationen etwa halbiert. Ein Beispiel zeigt Abbildung 6.22.

Die Funktion *Find* kann jetzt wie folgt implementiert werden:

```
function Find(x : element) : element;
var y,t : element;
begin
  y := x;
  while p[p[y]] ≠ p[y] do
    begin
```

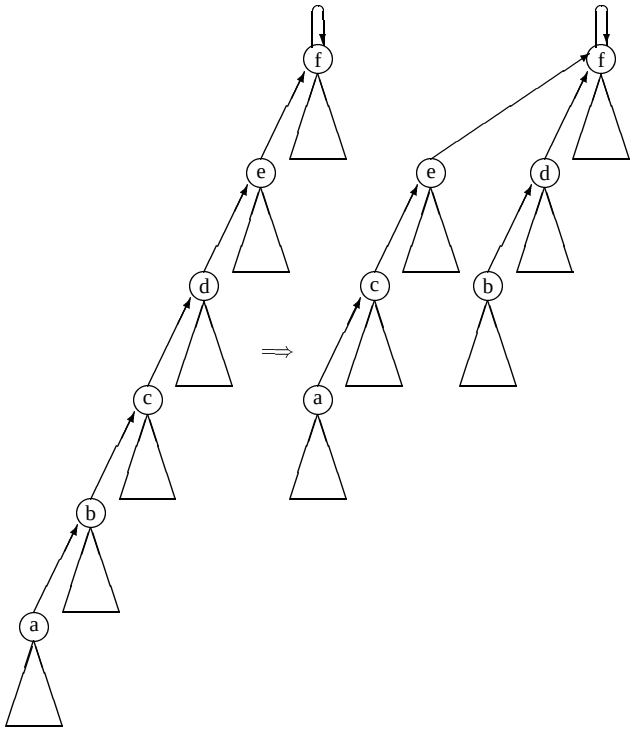

Abbildung 6.21

$t := p[p[y]];\ p[y] := t;\ y := t$

end

end

Es ist klar, daß damit das Spektrum der möglichen Methoden zur Pfadverkürzung keineswegs erschöpft ist. Beispielsweise könnte man einen Suchpfad ebensogut in drei, vier, usw. statt zwei etwa gleichlange Pfade aufteilen. In der Literatur sind eine Reihe weiterer Methoden vorgeschlagen und untersucht worden; man vergleiche dazu [182].

6.3 Allgemeiner Rahmen

Wörterbücher (Dictionaries), Priority Queues und Union-Find-Strukturen kann man als Spezialfälle eines allgemeinen Mengenmanipulationsproblems auffassen, das wie folgt beschrieben werden kann. Gegeben ist eine Kollektion K von paarweise disjunkten

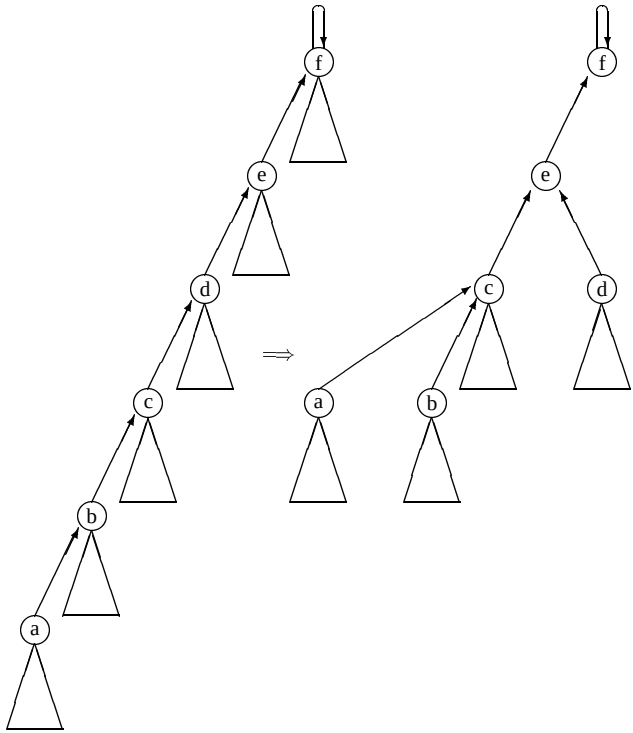

Abbildung 6.22

Mengen, deren Elemente zu einem Universum U gehören und deren Namen zu einer Menge N von Namen gehören.

$$\begin{aligned} K &= \{S_{n_1}, \ldots, S_{n_t}\}, \quad S_{n_i} \cap S_{n_j} = \emptyset, \text{ für } i \neq j. \\ U &\supseteq \bigcup K = \{x \in S \mid S \in K\} \\ N &\supseteq \{n_i \mid S_{n_i} \in K\} \end{aligned}$$

Das Universum sei eine geordnete Menge von Elementen. (Häufig nimmt man sogar an, daß das Universum U und die Namensmenge N die Menge der positiven ganzen Zahlen sind.)

Auf der Kollektion K soll eine beliebige Folge von Operationen, wie sie in Tabelle 6.3 angegeben sind, ausführbar sein. Diese Liste möglicher und sinnvoller Operationen für eine Kollektion K von Mengen ist keineswegs vollständig, sondern soll das breite Spektrum derartiger Operationen illustrieren.

Eine Lösung des Mengenmanipulationsproblems sollte natürlich berücksichtigen, welche Operationen mit welcher Häufigkeit, in welcher Reihenfolge ausgeführt werden. In vielen Fällen kann man jedoch eine Lösung wählen, deren Grobstruktur wie

Make-set(x,n):	Bilde eine Menge mit einzigem Element x und gebe ihr den Namen n. (Dabei wird vorausgesetzt, daß x und n neu sind.)
Suche(x,n):	Suche x in der Menge mit Namen n.
Einfüge(x,n):	Füge x in die Menge mit Namen n ein. (Dabei wird vorausgesetzt, daß x neu ist.)
Entferne(x,n):	Entferne x aus der Menge mit Namen n.
Find(x):	Bestimme den Namen der Menge, die x enthält.
Union(i,j,k):	Vereinige die Mengen mit Namen i und j zu einer Menge mit Namen k.
Access-Min(n):	Bestimme das Minimum in der Menge mit Namen n.
Delete-Min(n):	Entferne das Minimum in der Menge mit Namen n.
Nachfolger(x,n):	Bestimme das zu x nächstgrößere Element in der Menge mit Namen n.
Vorgänger(x,n):	Bestimme das zu x nächstkleinere Element in der Menge mit Namen n.
(k)*-tes Element*:	Bestimme das k-größte Element in $\bigcup K$

Tabelle 6.3

folgt beschrieben werden kann. Repräsentiere $\bigcup K \subseteq U$ durch einen balancierten, sortierten Binärbaum, den $\bigcup$*-Baum*. Wenn man die Operation *k-tes Element* unterstützen möchte, ist es sinnvoll, an jedem Knoten p noch einen Zähler $z(p)$ mitzuführen, der die Anzahl der Schlüssel im Teilbaum mit Wurzel p angibt. Stelle jede Menge S_i der Kollektion K durch einen nichtsortierten *Mengenbaum* dar, den S_i*-Baum*. Der Knoten x im $\bigcup$-Baum ist durch einen Zeiger mit dem Knoten x im S_i-Baum verbunden, wenn $x \in S_i$ ist. Die Menge aller Namen von Mengenbäumen ist in einem sortierten, balancierten N*-Baum* gespeichert. Die Wurzel eines jeden Mengenbaums ist durch je einen Verweis in beiden Richtungen mit seinem Namen im N-Baum verbunden.

Sollen Find-Operationen unterstützt werden, zeigt jeder Knoten eines Mengenbaumes auf seinen Vater. Sollen Access-Min- und Delete-Min-Operationen unterstützt werden, sind die Mengenbäume heapgeordnet. Falls die Union-Operation als Vereinigung nach Größe oder Höhe ausgeführt werden soll, muß man an den Wurzeln der Mengenbäume die Größe oder Höhe mitführen. Die in Abbildung 6.23 gezeigte Struktur der Lösung muß also auf den jeweils aktuell vorliegenden Fall zugeschnitten werden.

Wir geben an, wie einige der genannten Operationen ausgeführt werden können.

Einfüge(x,i): Füge x im $\bigcup$-Baum ein; suche i im N-Baum, folge Zeiger zur Wurzel des S_i-Baumes, füge x in diesen Baum ein. (Ist beispielsweise S_i heapgeordnet, so beinhaltet das Einfügen von x in S_i auch die Wiederherstellung der Heapordnung.)

Entferne(x,i): Die Ausführung dieser Operation verlangt, x im Mengenbaum S_i zu finden. Da wir im allgemeinen nicht voraussetzen, daß diese Bäume Suchbäume sind, sucht man x zunächst im $\bigcup$-Baum, folgt dem Zeiger von x zum Knoten gleichen Namens in einem der Mengenbäume, läuft dort zur Wurzel und stellt über den Verweis in

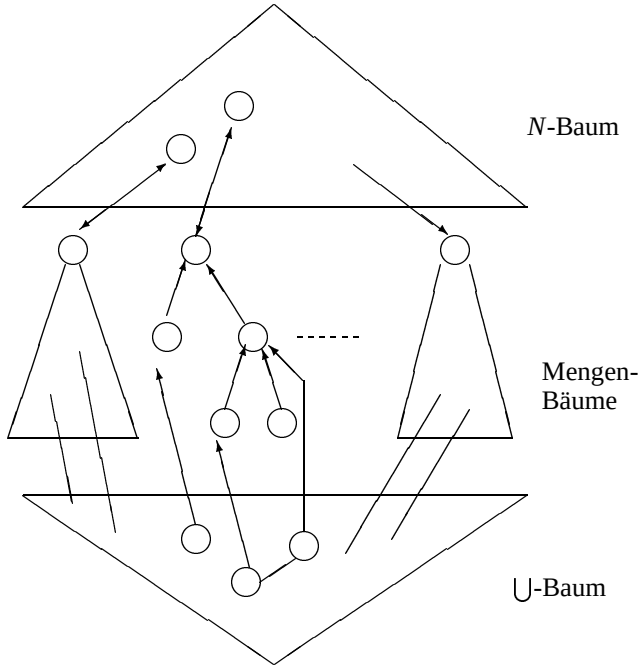

Abbildung 6.23

den N-Baum fest, ob x in S_i auftritt. Dann entfernt man gegebenenfalls x aus S_i und aus dem $\bigcup$-Baum.

k-tes Element: Man beginnt bei der Wurzel p des $\bigcup$-Baumes. Falls $z(p) < k$ ist, gibt es kein k-tes Element im $\bigcup$-Baum. Sonst inspiziert man den linken Sohn λp und dessen Zähler $z(\lambda p)$. Falls $k \leq z(\lambda p)$ ist, setzt man die Suche nach dem k-ten Element rekursiv beim linken Sohn fort. Falls $k = z(\lambda p) + 1$ ist, ist das in p gespeicherte Element das gesuchte. Falls schließlich $k > z(\lambda p) + 1$ ist, setzt man die Suche rekursiv beim rechten Sohn von p fort, sucht dort aber nach dem $(k - z(\lambda p) - 1)$-ten Element.

Es ist nicht schwer, sich in allen anderen Fällen zu überlegen, wie die Operationen ausgeführt werden können und welche zusätzlichen Voraussetzungen man gegebenenfalls über die Struktur der Mengenbäume usw. benötigt, um die gewünschten Operationen effizient ausführen zu können.

Die im vorigen Abschnitt 6.2 angegebenen Lösungen des Union-Find-Problems kann man folgendermaßen unter den hier angegebenen Rahmen subsummieren: Im Falle des Union-Find-Problems können $\bigcup$-Baum und N-Baum jeweils zu Arrays vereinfacht werden. Falls man Namen unterdrücken möchte und mit kanonischen Elementen operiert, kann man auf den N-Baum (oder ein N-Array) sogar ganz verzichten.

6.4 Aufgaben

Aufgabe 6.1
Eine Vorrangswarteschlange für ganzzahlige Schlüssel soll als Bruder-Baum realisiert werden, wobei die Schlüssel in den Blättern gespeichert werden. Als Wegweiser soll an jedem binären inneren Knoten der kleinste Schlüsselwert seines Teilbaumes stehen.

a) Geben Sie ein Einfüge-Verfahren für beliebige Schlüssel an und beschreiben Sie dieses, zusammen mit dem Knotenformat, in Pascal.

b) Geben Sie je eine Umstrukturierungs-Invariante und eine Umstrukturierungs-Operation für den Fall des Einfügens eines beliebigen Schlüssels und des Entfernens des Minimums an. Beachten Sie, daß Schlüssel nicht unbedingt sortiert in symmetrischer Reihenfolge auftreten, und daß Wegweiser angepaßt werden müssen.

c) Beschreiben Sie die beiden Umstrukturierungs-Operationen aus b) als Pascal-Prozeduren.

d) Beschreiben Sie die Priority-Queue-Operationen Access Min und Delete Min ebenfalls in Pascal.

Aufgabe 6.2
Ein Linksbaum, der als Priority Queue für eine Menge ganzzahliger Schlüssel dient, kann konstruiert werden, indem man diese Schlüssel in einer beliebigen Reihenfolge in den anfangs leeren Linksbaum unter Zuhilfenahme der Funktion *Verschmelzen* einfügt.

a) Beschreiben Sie eine Folge von N Schlüsseln (für beliebiges, natürliches N), für die der durch fortgesetztes Einfügen entstehende Linksbaum zu einer linearen Liste degeneriert.

b) Beschreiben Sie eine Folge von $2^k - 1$ Schlüsseln ($k \geq 1$, beliebig), für die der durch fortgesetztes Einfügen entstehende Linksbaum ein vollständiger Binärbaum ist.

c) Wieviele strukturell verschiedene Linksbäume für vier Schlüssel gibt es? Geben Sie für jeden dieser Bäume alle Reihenfolgen der Schlüssel 1, 2, 3, 4 an, durch die er bei fortgesetztem Einfügen in den anfangs leeren Linksbaum erzeugt werden kann.

Aufgabe 6.3
Eine Binomial Queue, also ein Wald von Binomialbäumen, soll durch fortgesetztes Einfügen ganzzahliger Schlüssel in die anfangs leere Queue erzeugt werden.

a) Geben Sie eine Folge von vier Schlüsseln an, für die die entstehende Binomial Queue strukturell gleich (gleich, wenn man keine Reihenfolge der Söhne unterstellt) ist mit dem entstehenden Linksbaum.

b) Verfolgen Sie die Entwicklung einer anfangs leeren Binomial Queue beim Einfügen der Schlüssel 17, 9, 12, 8, 15, 6 und beim anschließenden Entfernen des Schlüssels 9.

c) Definieren Sie das Knotenformat von Binomialbäumen für ganzzahlige Schlüssel in Pascal. Schreiben Sie in Pascal Prozeduren und Funktionen für die Operationen Access Min, Einfügen, Entfernen, Minimum Entfernen, Herabsetzen und Verschmelzen.

Aufgabe 6.4

Verfolgen Sie im einzelnen, wie sich der anfangs leere Fibonacci Heap verändert, wenn er als Priority Queue für das in Abschnitt 6.1.1 beschriebene Verfahren von Dijkstra zur Berechnung kürzester Pfade für den in Abbildung 6.1 gezeigten Graphen eingesetzt wird. Verfolgen Sie inbesondere für jede Operation die Änderung von Markierungen und des Kontostandes. Vergleichen Sie als alternative Implementierungen der Priority Queue für dieses Beispiel Binomial Queues und Linksbäume.

Aufgabe 6.5

Verfolgen Sie im einzelnen die Veränderungen einer Union-Find-Struktur zur Berechnung eines minimalen, spannenden Baumes nach Kruskal für das in Abbildung 6.16 angegebene Beispiel. Welche Pfadlängen ergeben sich für die einzelnen *Find*-Operationen, wenn man sich bei der Vereinigung nach der Höhe von Bäumen richtet? Welchen Effekt hat im Beispiel die Kompressionsmethode zur Pfadverkürzung?

Aufgabe 6.6

Bei der Kompressionsmethode zur Pfadverkürzung haben nach Erledigung einer Operation $Find(x)$ alle ursprünglich auf dem Pfad von x zur Wurzel des Baumes gelegenen Knoten die Entfernung 1 zur Wurzel. Entwerfen und implementieren Sie eine Pfadverkürzungsmethode, bei der diese Entfernung höchstens 2 beträgt, bei der man aber den Pfad von x zur Wurzel nur einmal durchläuft.