Kapitel 6

Fibonacci-Heaps und Amortisierte Analyse

Der Fibonacci-Heap dient als eine Implementierung für die abstrakte Datenstruktur *Priority Queue* und wurde von Michael L. Fredman and Robert E. Tarjan entwickelt. Im Vergleich zur einfachen Queue, die Sie aus Algorithmen und Datenstrukturen 1 kennen, speichert eine Priority Queue für jedes Element zusätzlich einen Prioritätswert ab und unterstützt folgende Funktionen:

- Insert: Ein neues Element wird in die Priority Queue eingefügt.
- ExtractMax: Das Element mit dem größten Prioritätswert wird ausgegeben und aus der Priority Queue entfernt.
- IncreaseKey: Der Prioritätswert eines Elements in der Priority Queue wird erhöht.

Das Einsatzgebiet von Priority Queues in der Informatik ist sehr vielfältig. Bei Ressourcenzuteilungen oder Scheduling-Anwendungen sollen z.B. Prozesse bevorzugt werden, die eine hohe Priorität besitzen. Bei der Berechnung von kürzesten Wegen mittels Dijkstras Algorithmus können die Distanzen zwischen den einzelnen Knoten in Priority Queues gespeichert werden. Prims Algorithmus zur Berechnung eines minimalen Spannbaums kann hiermit effizienter implementiert werden, und nicht zuletzt können Priority Queues auch zum effizienten Sortieren nützlich sein.

Eine einfache Implementierung einer Priority Queue haben Sie schon bei Heapsort kennengelernt. Dort wurde sie als ein binärer Heap realisiert. Zur Erinnerung: Ein (Maximum-)Heap ist ein spezieller binärer Baum, repräsentiert durch ein einfaches Feld, in dem die für jeden Knoten die Heapbedingung gilt: Ein eingetragene Schlüssel muss mindestens so groß wie die Schlüssel der beiden Kinder, falls diese existieren, sein. Damit ist es möglich, für einen

n-elementigen Heap die Methoden Insert, ExtractMax und IncreaseKey in $O(\log n)$ Zeit zu implementieren.

Ein Fibonacci-Heap besteht aus einer Menge von Bäumen, die jeweils die genannte Heapeigenschaft erfüllen. Allgemein kann nun ein Knoten auch mehr als zwei Nachfolger besitzen, d.h. wir beschränken uns nun nicht mehr auf binäre Bäume. Die Wurzelknoten befinden sich in einer gemeinsamen Wurzelebene. Ein globaler Max-Pointer verweist auf das größte Element (das Element mit der höchsten Priorität) in der Datenstruktur. Jeder Knoten in den Bäumen hat außerdem einen Zustandsfeld, der angibt, ob er *angeregt* ist oder nicht. Diese Eigenschaft wird in der IncreaseKey-Operation verwendet und ist in weiterer Folge für die Performanceanalyse wichtig.

Abbildung 6.1 zeigt ein Beispiel für einen Fibonacci-Heap bestehend aus drei Bäumen. Die Knoten 18 und 12 sind angeregt.

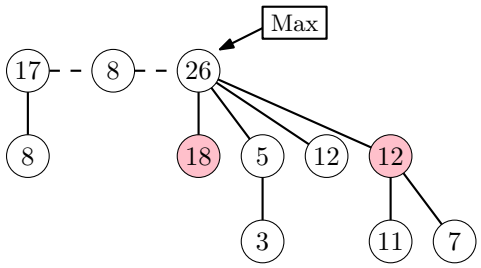

Abbildung 6.1: Beispiel für einen Fibonacci-Heap.

Durch seinen besonderen Aufbau ermöglicht der Fibonacci-Heap eine erhebliche Performancesteigerung. Tabelle 6.1 zeigt einen Laufzeitvergleich zwischen der Implementierung einer Priority Queue als binärer Heap und als Fibonacci-Heap.

Tabelle 6.1: Laufzeitvergleich zwischen binärem Heap und Fibonacci-Heap.

Operation	Binärer Heap	Fibonacci-Heap
Insert	$O(\log n)$	$O(1)$
ExtractMax	$O(\log n)$	$O(\log n)^*$
IncreaseKey	$O(\log n)$	$O(1)^*$

* Amortisierte Laufzeit, siehe Kapitel 6.2

6.1 Implementierung

In diesem Kapitel wird die konkrete Realisierung der drei Operationen erläutert und provisorisch analysiert.

6.1.1 Insert

Die Implementierung von Insert ist sehr einfach: Der einzufügende Knoten v wird als ein neuer, zusätzlicher Wurzelknoten auf der Wurzelebene eingefügt und der Max-Pointer gegebenenfalls umgesetzt. Trivialerweise benötigt diese Funktion nur eine konstante Laufzeit. Diese einfache Herangehensweise bedeutet jedoch einen größeren Umorganisationsaufwand bei den anderen Funktionen.

6.1.2 ExtractMax

Bei der ExtractMax-Funktion ist es zwar einfach, mittels des Max-Pointers das größte Element zu finden, nach dessen Entfernen muss jedoch die Datenstruktur wie folgt reorganisiert werden. Alle Kinder des Max-Pointers werden in die Wurzelebene aufgenommen und bilden somit die Wurzeln ihrer Unterbäume. Dann ist es notwendig, alle Wurzeln zu durchlaufen, um das neue größte Element zu bestimmen und den Max-Pointer auf dieses zu setzen. Währenddessen wird die Datenstruktur aufgeräumt, sodass am Ende alle Wurzelknoten paarweise verschiedene Knotengrade besitzen. Diese Eigenschaft ist später für das garantieren einer guten Performance wichtig.

Algorithmus 32 Aufräumen (**var** H)

Eingabe: Fibonacci-Heap während ExtractMax nach dem Löschen des größten Elements.
Ausgabe: Aufgeräumter Fibonacci-Heap.

```
1:  für alle Wurzeln v {
2:    ende = false;
3:    wiederhole
4:      d = deg(v);
5:      falls B[d] == NULL dann {
6:        B[d] = v;
7:        ende = true;
8:      } sonst {
9:        falls B[d] kleiner als v in Bezug auf Priorität dann {
10:         mache B[d] zum Kind von v;
11:       } sonst {
12:         mache v zum Kind von B[d];
13:         v = B[d];
14:       } // deg(v) ist jetzt d + 1
15:       B[d] = NULL;
16:     }
17:   bis ende;
18: }
```

Algorithmus 32 zeigt, wie dieses Prinzip umgesetzt wird. Für jeden Wurzelknoten wird überprüft, ob sein Grad einzigartig ist. Wenn dies nicht der Fall ist, werden die beiden Wurzelknoten mit dem selben Grad und die von ihnen aufgespannten Bäume zusammengefügt. Der Grad des Wurzelknotens des neuen Baumes wird wieder überprüft. Für die Implementierung wird ein Hilfsarray B verwendet, das Zeiger zu den Wurzelknoten aufgegliedert nach ihren Graden abspeichert, d.h. $deg(v) = i \Leftrightarrow B[i] = v$, wobei $deg(v)$ den Grad des Knotens v darstellt. Damit lässt sich effizient erkennen bzw. verhindern, dass zwei Wurzelknoten den gleichen Grad annehmen würden. Abb. 6.2 und 6.3 illustrieren diesen Vorgang anhand eines Beispiels.

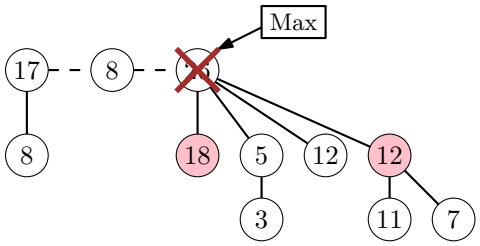

Abbildung 6.2: Beispiel für ExtractMax: Der Knoten 26 wird entfernt.

Es ist leicht zu erkennen, dass die Worst-Case Laufzeit für ExtractMax von der Anzahl der Wurzelknoten abhängt. Sei d der Knotengrad des gelöschten Wurzelknotens und m die Anzahl der Wurzeln vor dem Löschen, dann benötigt ExtractMax $O(d+m)$ Zeit. Dies bedeutet, dass diese Funktion im Worst-Case, nachdem n Mal ein neues Element eingefügt wurde und danach ExtractMax aufgerufen wird, eine Laufzeit von $O(n)$ benötigt.

6.1.3 IncreaseKey

Zuerst wird der Prioritätswert des betroffenen Knotens v auf den neuen Wert $newKey$ erhöht. Wenn die Heapeigenschaft zwischen v und seinem Vorgänger u nicht verletzt ist, muss nichts weiter unternommen werden. Ist diese jedoch verletzt, kommt v mit seinem Unterbaum in die Wurzelebene. Der Max-Pointer wird gegebenenfalls umgesetzt. Hier benötigen wir den Zustand, ob ein Knoten angeregt ist, um zu kennzeichnen, ob der Knoten schon mal ein Kind verloren hat. Ist u noch nicht angeregt, wird er angeregt. War u bereits angeregt, d.h. er hat schon einmal ein Kind verloren, kommen er und der von ihm aufgespannte Unterbaum ebenfalls in die Wurzelebene und der Prozess wird mit seinem Vorgänger fortgesetzt. Dieser relativ aufwendige Prozess ist eine gute Performancegarantie von großer Bedeutung. Der detaillierte Ablauf ist in Algorithmus 33 angegeben. Abb. 6.4 und 6.5 illustrieren diesen Vorgang anhand von zwei Beispielen.

Offensichtlich hängt die Worst-Case Laufzeit für IncreaseKey von der Anzahl der angeregten Knoten ab. Sind alle Knoten, die sich auf dem Weg vom Knoten v bis zur Wurzel befinden,

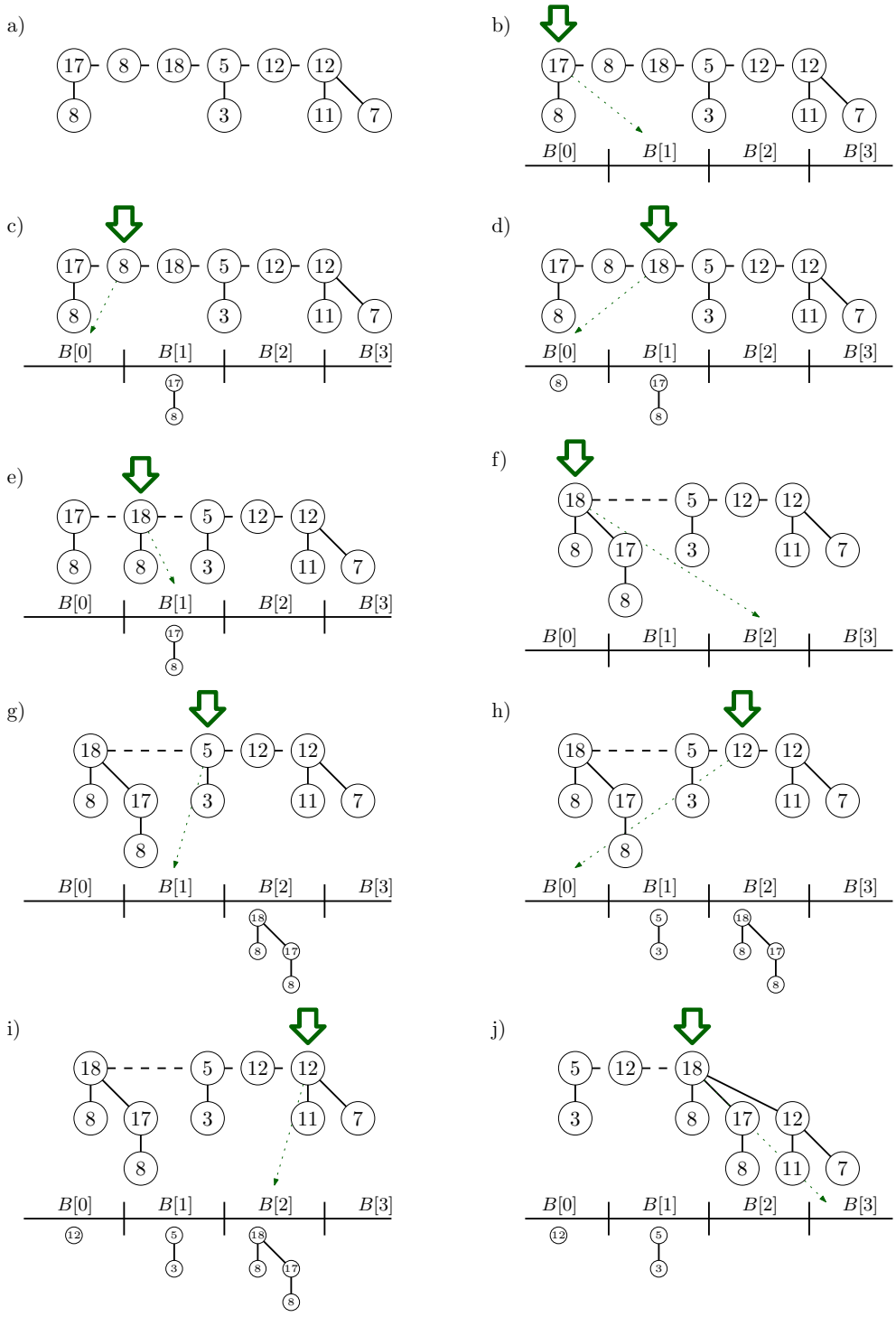

Abbildung 6.3: Beispiel für ExtractMax: Der Heap wird nach dem Entfernen reorganisiert.

Algorithmus 33 IncreaseKey(**var** $H, v, newKey$)

Eingabe: Fibonacci-Heap und Knoten v, für den der Prioritätswert auf $newKey$ erhöht werden soll.

Ausgabe: Fibonacci-Heap nach IncreaseKey.

```
1:  erhöhe den Wert von v auf newKey;
2:  falls v.key > Max.key dann {
3:     Max = v;
4:  }
5:  falls Heapeigenschaft zwischen v uns seinem Vorgänger verletzt dann {
6:     ende = false;
7:     wiederhole
8:        u = Vorgänger von v;
9:        mache v zu nicht angeregter Wurzel;
10:       falls u angeregt dann {
11:          v = u;
12:       } sonst {
13:          rege u an, wenn er kein Wurzelknoten ist;
14:          ende = true;
15:       }
16:    bis ende;
17: }
```

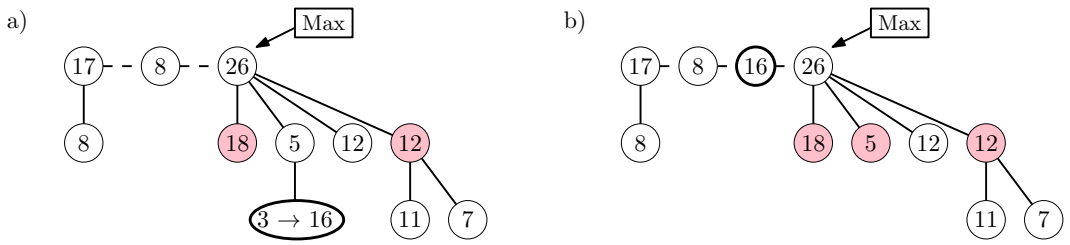

Abbildung 6.4: Beispiel für IncreaseKey: Wenn die Priorität von Knoten 3 auf 16 erhöht wird, wird die Heapeigenschaft verletzt und dieser Knoten kommt in die Wurzelebene. Sein Vorgänger wird angeregt.

angeregt, wird in jeder Iteration ein Teilbaum abgespaltet und in die Wurzelebene eingefügt. Damit steigt der Aufwand von IncreaseKey im Worst-Case proportional zur Höhe des Heaps.

Basierend auf den bisher angestellten Überlegungen lassen sich die die Worst-Case Laufzeiten in Tabelle 6.2 wie gefolgt zusammenfassen. Wir sehen, dass der Fibonacci-Heap nicht mit dem binären Heap mithalten kann. Wurde anfangs zu viel versprochen?

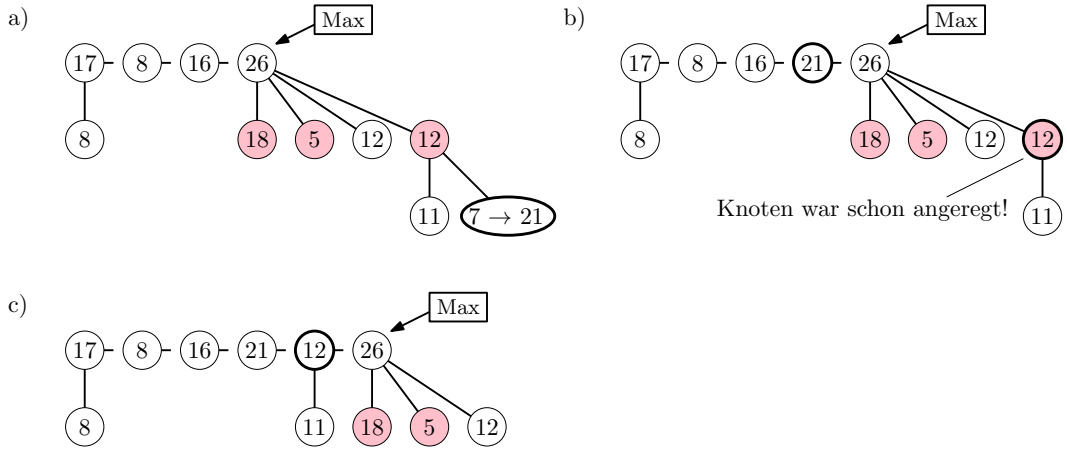

Abbildung 6.5: Beispiel für IncreaseKey: Wenn die Priorität von Knoten 7 auf 21 erhöht wird, kommt neben diesem Knoten sein Vorgänger ebenfalls in die Wurzelebene, da er bereits angeregt war.

Tabelle 6.2: Worst-Case-Analyse zwischen binärem Heap und Fibonacci-Heap.

Operation	Binärer Heap	Fibonacci-Heap
Insert	$O(\log n)$	$O(1)$
ExtractMax	$O(\log n)$	$O(n)$
IncreaseKey	$O(\log n)$	$O(h(\text{Heap}))$

6.2 Amortisierte Analyse

Im vorigen Kapitel haben wir gesehen, dass der Fibonacci-Heap eine schlechte Worst-Case-Performance aufweist. Bis jetzt haben wir uns primär mit Best-Case-, Worst-Case- und Average-Case-Analysen beschäftigt. Best-Case und Worst-Case Analysen liefern lediglich untere bzw. obere Schranken für den allgemein zu erwartenden Fall und sind daher in der Praxis häufig wenig aussagekräftig. Die Average-Case-Analyse bietet zwar in diesem Sinne eine gute Laufzeitabschätzung, ist aber in vielen Fällen sehr aufwendig bzw. in der Praxis häufig auch nicht durchführbar.

Anstatt Algorithmen anhand einzelner Operationen zu analysieren, werden bei der amortisierten Analyse *Sequenzen von Operationen* betrachtet. Die Motivation liegt darin, dass bei manchen Algorithmen einzelne Operationen zwar teuer sein können, aber diese Fälle nur relativ selten auftreten. Wenn der Algorithmus in den überwiegend mehrheitlichen Situationen effizient ist, wäre es ungerecht, seine Leistung nur anhand eines einzelnen Falles im Worst-Case zu bewerten. Die amortisierte Analyse ist daher die Worst-Case-Analyse für eine Sequenz von Operationen. In Algorithmen und Datenstrukturen 1 wurde diese Thematik bereits kurz im Zusammenhang mit der Union-Find Datenstruktur angeschnitten. Hier wollen

wir nun genauer auf die Analyse eingehen.

Es gibt verschiedene Möglichkeiten, die amortisierte Analyse durchzuführen:

- Aggregat-Methode
- Bankkonto-Methode
- Potenzialfunktion-Methode

Das Grundprinzip ist bei allen gleich: Die billigen Operationen werden systematisch verteuert, damit die *amortisierten Kosten* der teuren Operationen gesenkt werden können. Um diese Methoden besser zu veranschaulichen, werden sie am folgenden einfachen Beispiel illustriert.

Beispiel: Binärzähler

Der Binärzähler enthält einen Zählerstand, der im Dualsystem arbeitet. Jedes Mal wenn die Funktion „Increase“ aufgerufen wird, erhöht sich der Zählerstand um eins. Der Aufwand dieser Funktion wird an der Anzahl der Bit-Flips gemessen.

Verwendet man die klassische Worst-Case-Analyse, liegt der Aufwand in $O(n)$, wenn der Zählerstand durch n Bits dargestellt wird. Dieser Fall tritt ein, wenn an jeder Stelle im Zählerstand bis auf die vorderste eine Eins steht und dann „Increase“ aufgerufen wird. Im weiteren Verlauf werden wir sehen, wie die amortisierten Kosten berechnet werden können.

6.2.1 Aggregat-Methode

Die Aggregat-Methode ist die primitivste Form, um die amortisierten Kosten zu berechnen. Für eine Sequenz von Operationen werden einfach die Kosten aufsummiert und durch die Anzahl der Operationen dividiert. Somit erhält man die durchschnittlichen Kosten pro Operation.

Tabelle 6.3 zeigt ein Beispiel, wie dieses Verfahren auf den Binärzähler angewendet wird. Die Kosten für sechs Operationen betragen in Summe zehn. Das ergibt einen Durchschnitt von $\frac{10}{6}$ pro Operation. Naheliegenderweise ist die Aggregat-Methode in vielen Fällen ungeeignet. Die Durchschnittskosten sind im Allgemeinen stark von der Anzahl und Wahl der Operationen abhängig. Es ist schwierig, „alle“ Kombinationen sinnvoll abzudecken.

Tabelle 6.3: Beispiel für die Aggregat-Methode angewendet auf den Binärzähler.

Operation	Zählerstand	Kosten
Init.	000000	
1	000001	1
2	000010	2
3	000011	1
4	000100	3
5	000101	1
6	000110	2

6.2.2 Bankkonto-Methode

Die Bankkonto-Methode berechnet die amortisierten Kosten mit Hilfe eines virtuellen Kontos. Stellen Sie sich vor, dass Sie für jede billige Operation absichtlich mehr als die tatsächlich notwendigen Kosten ausgeben. Das „Guthaben“ wird am Konto aufbewahrt und später für die teuren Operationen ausgegeben, damit diese günstiger ausfallen.

Tabelle 6.4 zeigt wieder das Binärzähler Beispiel, das diesmal mit der Bankkonto-Methode analysiert wird. Jede Increase-Operation, die die letzte Null in eine Eins verwandelt, erfordert tatsächlich nur einen Bit-Flip. Stattdessen bezahlen wir in einem solchen Fall die Kosten von zwei Bit-Flips, damit ein Guthaben von einem Bit-Flip am Konto landet. Teure Operationen treten dann auf, wenn durch eine Increase-Operation viele Einser in Nullen verwandelt werden. In diesen Fällen bezahlen wir ebenfalls nur die Kosten von zwei Bit-Flips und die Differenz auf die tatsächlichen Kosten wird mit dem Guthaben am Konto beglichen. Damit betragen die amortisierten Kosten für jede Operation konstant zwei Bit-Flips.

Tabelle 6.4: Beispiel für die Bankkonto-Methode angewendet auf den Binärzähler.

Operation	Zählerstand	„bezahlt“	tatsächl. Kosten	Konto
Init.	000000			0
1	000001	2	1	1
2	000010	2	2	1
3	000011	2	1	2
4	000100	2	3	1
5	000101	2	1	2
6	000110	2	2	2

6.2.3 Potenzialfunktion-Methode

Die Potenzialfunktion-Methode ist das mächtigste der drei Analyseverfahren. Sie betrachtet die Konfiguration der Datenstruktur D, auf der operiert wird. Die Potenzialfunktion $\Phi(D)$ kennzeichnet das Potenzial von D, d.h. den Grad der Unordnung im Vergleich zum Idealzustand. Je höher das Potenzial desto höher ist die Wahrscheinlichkeit, dass die darauffolgenden Operationen teuer ausfallen werden. Der Zustand von D und sein Potenzial können durch jede Operation verändert werden.

Sei $\langle o_1, \ldots, o_n \rangle$ eine Folge von n Operationen. Wir definieren folgende Notationen:

- a_i sind die amortisierten Kosten für die Operation o_i.
- t_i sind die tatsächlichen Kosten für die Operation o_i.
- D_i ist der Zustand der Datenstruktur nach der i-ten Operation.
- $\Phi_i = \Phi(D_i)$ ist das Potenzial der Datenstruktur nach der i-ten Operation.
- Φ_0 ist das Potenzial des Initialzustands von D vor der ersten Operation.

Die amortisierten Kosten $a_i,\ i = 1, \ldots, n,$ werden durch

$$a_i = t_i + \Phi_i - \Phi_{i-1} \text{ oder } a_i = t_i + \Delta\Phi$$

ausgedrückt, wobei $\Delta\Phi = \Phi_i - \Phi_{i-1}$ die Differenz des Potenzials zwischen den Zuständen i und $i-1$ darstellt. Folglich betragen die amortisierten Kosten für eine Folge von Operationen

$$\sum_{i=1}^{n} a_i = \sum_{i=1}^{n} (t_i + \Phi_i - \Phi_{i-1}) = \sum_{i=1}^{n} t_i + \Phi_n - \Phi_0.$$

Wählen wir eine nichtnegative Potenzialfunktion mit $\Phi_0 = 0$, gilt

$$\sum_{i=1}^{n} t_i \leq \sum_{i=1}^{n} a_i$$

und wir erhalten somit eine obere Schranke für die tatsächlichen Kosten.

Im Fall des Binärzählers ist D die Binärdarstellung des Zählers. Teure Operationen können dann auftreten, wenn der Zähler sehr viele Einser enthält. Daher wählen wir für die Potenzialfunktion der i-ten Operation $\Phi(D_i) = B_i$, wobei B_i der Anzahl der Einser in D_i entspricht. Nehmen wir an, dass D_i mit b_i aufeinanderfolgenden Einsern endet. Diese werden bei einer Increase-Operation zu Nullen konvertiert und die davorstehende Null wird zu einer Eins. Wir erhalten $\Phi_{i-1} = B_{i-1}$, $\Phi_i = B_{i-1} - b_i + 1$ und $t_i = b_i + 1$. Damit ergibt sich

$$a_i = (b_i + 1) + (B_{i-1} - b_i + 1) - B_{i-1} = 2 \qquad \Rightarrow \qquad \sum_{i=1}^{n} t_i \leq 2n.$$

6.3 Amortisierte Analyse von Fibonacci-Heaps

Wir werden nun die Potenzialfunktion-Methode anwenden, um die amortisierten Kosten für die Operationen auf einem Fibonacci-Heap zu analysieren. Dafür müssen wir zuerst zeigen, dass die Knotengrade in einem Fibonacci-Heap mit n Knoten durch $O(\log n)$ beschränkt sind. Folgende Lemmata werden benötigt:

Lemma 6.1 *Sei x ein Knoten vom Grad d und seien $y_1, \ldots, y_d$ dessen Kinder gemäß der Reihenfolge, in der sie durch ExtractMax-Operationen zu x verlinkt wurden. Dann ist der Grad für alle y_i entweder $i-1$ oder $i-2$ für $i = 1, \ldots, d$.*

Beweis: Zu dem Zeitpunkt, zu dem y_i zu x gelinkt wurde, hatte x bereits $y_1, \ldots, y_{i-1}$ als Kinder.

$\Rightarrow deg(x) = i - 1$.

$\Rightarrow deg(y_i) = i-1$, da wir während ExtractMax nur Knoten gleichen Grades verlinken.

Seitdem hat y_i höchstens nur ein Kind durch IncreaseKey-Operationen verloren, denn ansonsten wäre y_i von x abgeschnitten worden.

$\Rightarrow deg(y_i) = i - 1$ oder $deg(y_i) = i - 2$.

□

Lemma 6.2 *Sei $\varphi \in \mathbb{R}$ eine Zahl mit $1 < \varphi \leq 2$ und $1+\varphi \geq \varphi^3$. Sei x ein Knoten vom Grad d und $size(x)$ die Anzahl der Knoten, die im von x aufgespannten Unterbaum enthalten sind. Dann gilt $size(x) \geq \varphi^d$.*

Beweis: Vollständige Induktion

- Induktionsanfang: Lemma ist wahr für $d = 0$.
- Induktionsannahme: Lemma ist wahr für alle $d < k$.
- Induktionsbehauptung: $size(x) \geq \varphi^k$ mit $d = k$:

$$size(x) = 1 + \sum_{i \leq k} size(y_i) \geq$$

$$\geq size(y_{k-1}) + size(y_k) \geq$$

$$\geq \varphi^{k-3} + \varphi^{k-2} =$$
$$= \varphi^{k-3}(1+\varphi) \geq$$
$$\geq \varphi^{k-3}\varphi^3 = \varphi^k$$

□

Korollar 6.1 *Der Grad d jedes Knotens x ist beschränkt durch $O(\log n)$.*

Beweis:

$$n \geq size(x) \geq \varphi^d$$
$$\Rightarrow \log_\varphi n \geq d$$

□

6.3.1 Potenzialfunktion für ExtractMax

Mittels klassischer Worst-Case-Analyse haben wir gesehen, dass ExtractMax bis zu $O(n)$ Zeit benötigen kann. Dieser Fall tritt ein, wenn n Mal eingefügt wurde und danach ExtractMax aufgerufen wird. Intuitiv betrachtet baut Insert über n Operationen langsam ein Potenzial auf, sodass der nächste Aufruf von ExtractMax verlangsamt wird. Dieser baut jedoch das Potenzial wieder ab, wodurch darauffolgende ExtractMax-Operationen wieder schneller arbeiten können. Es ist daher höchst unwahrscheinlich, dass über eine Sequenz von Operationen betrachtet der Worst-Case wiederholt auftritt.

Wir suchen eine Potenzialfunktion, die besagt, wie schlecht die aktuelle Konfiguration des Fibonacci-Heaps D ist. Wie bereits im Kapitel 6.1.2 festgestellt wurde, hängt die Performance von ExtractMax stark von der Anzahl der Wurzelknoten ab. Daher wählen wir für die Potenzialfunktion

$$\Phi(D) = \alpha W,$$

wobei α eine Konstante und W die Anzahl der Wurzelknoten sind.

Wir nehmen an, dass der Heap am Anfang W_1 Wurzelknoten und nach der ExtractMax-Operation W_2 Wurzelknoten besitzt. Der gelöschte Wurzelknoten hatte Grad d. Die tatsächlichen Kosten betragen

$$t_i = c(d + W_1),$$

wobei c eine Konstante ist. Die amortisierten Kosten betragen

$$a_i = t_i + \Delta\Phi = c(d + W_1) + \alpha(W_2 - W_1) = (c - \alpha)W_1 + cd + \alpha W_2.$$

Nachdem für die beiden Konstanten α und c noch keine Werte festgelegt wurden, wählen wir $\alpha = c$. Damit vereinfachen sich die amortisierten Kosten auf

$$a_i = cd + \alpha W_2.$$

Da wir gezeigt haben, dass der Knotengrad durch $O(\log n)$ beschränkt ist, gilt $d = O(\log n)$. Weiters stellen wir fest, dass $W_2 = O(\log n)$, da alle Wurzelknoten am Ende der ExtractMax-Operation unterschiedliche Grade besitzen. Folglich gilt $a_i = O(\log n)$.

6.3.2 Potenzialfunktion für IncreaseKey

Nun wollen wir eine geeignete Potenzialfunktion für IncreaseKey finden. Wir nehmen an, dass während diesem Prozess insgesamt k neue Wurzeln geschaffen werden. Wenn viele neue Wurzeln erschaffen werden, bedeutet das nicht nur einen hohen Aufwand, sondern erhöht auch das Potenzial. Für die Potenzialfunktion müssen wir daher zusätzlich auch die angeregten Knoten heranziehen, da diese für die Erzeugung neuer Wurzelknoten verantwortlich sind. Nehmen wir an, dass vor der Operation insgesamt A angeregte Knoten im Heap D existierten. Als Potenzialfunktion wählen wir

$$\Phi(D) = \alpha' k + \beta' A,$$

wobei α' und β' wieder Konstanten sind, deren Werte wir später bestimmen wollen.

Die tatsächlichen Kosten betragen

$$t_i = c'(k+1),$$

wenn k Knoten abgespaltet werden, wobei c' eine weitere Konstante ist. Die Differenz des Potenzials beträgt

$$\Delta\Phi = \alpha' k + \beta'(2-k),$$

denn alle neuen Wurzeln waren angeregt bis auf eventuell den Knoten, dessen Prioritätswert erhöht und direkt abgespaltet wurde. Während dieses Prozesses wird höchstens ein neuer Knoten angeregt.

Die amortisierten Kosten betragen damit

$$a_i = t_i + \Delta\Phi = c'(k+1) + \alpha' k + \beta'(2-k) = (c' + \alpha' - \beta')k + c' + 2\beta'.$$

Wählen wir $\beta' = c' + \alpha'$, lässt sich der Ausdruck auf

$$a_i = 3c' + 2\alpha'$$

vereinfachen und dieser liegt in $O(1)$, d.h. er ist konstant.

Wir müssen uns nun noch darüber Gedanken machen, ob ExtractMax neu abgeschätzt werden muss, da die Komponente der angeregten Knoten hinzugekommen ist. In diesem Fall können wir darauf verzichten, da bei ExtractMax keine neuen Knoten angeregt werden und somit das Potenzial für IncreaseKey sich nicht verschlechtern kann.

Schlussfolgerung

Eine Folge von n Insert- bzw. IncreaseKey- und m ExtractMax-Operationen auf einem anfänglich leeren Fibonacci-Heap können in der Zeit $O(n + m \log n)$ ausgeführt werden. Mittels klassischer Worst-Case-Analyse würde man auf $O(n^2 + mn)$ kommen, was zu pessimistisch wäre.

Anmerkungen

Der Name „Fibonacci-Heap“ kommt von einer verschärften Version des Lemmas, das wir bewiesen haben. Dieses besagt, dass ein Knoten vom Grad d im Heap mindestens $F_{d+2} \geq \varphi^d$ Nachkommen besitzt, wobei F_{d+2} die $(d+2)$-te Fibonacci-Zahl und $\varphi = \frac{1+\sqrt{5}}{2}$ den Goldenen Schnitt darstellen.

Der Fibonacci-Heap spielt vor allem bei theoretischen Analysen eine wichtige Rolle. Man kann z.B. für Dijkstras Algorithmus zum Finden kürzester Pfade zeigen, dass dieser eine Laufzeit von $O(|E| + |V| \log |V|)$ erreichen kann, wenn die Priority Queue durch einen Fibonacci-Heap realisiert wird. In der Praxis werden jedoch häufig alternative Datenstrukturen verwendet, die einfacher aufgebaut und leichter zu implementieren sind.