(M. Dietzfelbinger, 17. Dezember 2020)

4 Implementierung von Priority Queues: Fibonacci-Heaps

Die vom Datentyp „Priority Queue (PQ)“ (deutsch auch: „Vorrangwarteschlange“) zur Verfügung gestellte Funktionalität ist folgende:

Es werden Objekte („Einträge“) e mit einem Attribut $key(e) \in U$ verwaltet („Schlüssel“ oder „Priorität“, mit $x, y, z, \ldots$ bezeichnet), wobei $(U, <)$ eine Totalordnung ist, z. B. $U = \mathbb{N}$. Dabei entsprechen *kleinere* Schlüssel einer *höheren* Priorität. Typische Anwendungen finden sich in Algorithmen wie dem Algorithmus von Dijkstra, bei Discrete-Event-Simulation, bei der Prozessverwaltung in Rechensystemen (s. AuD-Vorlesung).

Eine einfache Priority Queue bietet mindestens die folgenden Operationen an:

Name(Parameter)	Effekt
new()	erzeuge neue leere PQ
makeHeap($\{e_1, \ldots, e_n\}$)	erzeuge neue PQ mit Einträgen $e_1, \ldots, e_n$
insert(e)	füge e als neuen Eintrag hinzu
min()	gib einen Eintrag mit minimalem Schlüssel aus
deleteMin()	gib einen Eintrag mit minimalem Schlüssel aus **und** entferne ihn aus der Datenstruktur[1]

Einige Algorithmen und Anwendungen (insbesondere der Algorithmus von Dijkstra, und mit ihm alle Arten von Discrete-Event-Simulation) benötigen weitere Operationen, bei denen Schlüssel von Einträgen verändert werden, die nicht das Minimum sind. Hierzu benötigt man den Zugriff auf durch Zeiger identifizierte Einträge. Die Verwendung von Zeigern auf Objekte in der Datenstruktur durch den Benutzer ist extrem schlechter Programmierstil, da es sämtlichen Kapselungsprinzipien widerspricht. Dennoch verwenden wir in der Diskussion den Begriff des Zeigers (Java-Bezeichnung: Referenz). In realen Implementierungen sind auch diese Zeiger lokal für die Datenstruktur („`private`“); der Benutzer benutzt *Namen*, um die gewünschten Einträge zu bezeichnen. Details zu solchen Kapselungstechniken wurden schon in der Vorlesung AuD betrachtet.

[1] Anstelle von *deleteMin*() findet man auch die Bezeichnung *extractMin*().

Zusätzliche „fortgeschrittene“ Operationen („adressierbare PQs“):

Name(Parameter)	Effekt
insert(e)	füge e ein; Rückgabewert: ein Zeiger p auf den Eintrag
makeHeap$(\{e_1, \ldots, e_n\})$	erzeuge neue PQ mit Einträgen $e_1, \ldots, e_n$; Rückgabewerte: n Zeiger $p_1, \ldots, p_n$ auf diese Einträge
decreaseKey(p, x)	verringere den Schlüssel im Eintrag, auf den p zeigt, auf x (wenn $x >$ aktueller Schlüssel: Fehler)
delete(p)	lösche den Eintrag, auf den p zeigt

Eine weitere interessante Operation ist

Name(Parameter)	Effekt
meld(H')	vereinige die PQ mit einer anderen, H', zu *einer* PQ

Adressierbare PQs, die auch diese Operation unterstützen, heißen „meldable PQs“ oder „PQs mit Verschmelzung“.

In der Vorlesung „Algorithmen und Datenstrukturen (AuD)“ (Bachelor) hat sich gezeigt, dass binäre Heaps alle Operationen von adressierbaren PSs (ohne *meld*) recht effizient implementieren. Die benötigten Zeiten bei Binärheaps im schlechtesten Fall sind, wenn die Anzahl der Einträge n ist:

Name(Parameter)	Zeitschranke
new()	$O(1)$
min()	$O(1)$
deleteMin()	$O(\log n)$
insert(e)	$O(\log n)$
makeHeap$(\{e_1, \ldots, e_n\})$	$O(n)$
delete(p)	$O(\log n)$
decreaseKey(p, x)	$O(\log n)$
meld(H')	$O(n)$

Dabei wird „*delete*(p)“ dadurch realisiert, dass zuerst *decreaseKey*$(p, -\infty)$ für einen künstlichen Schlüssel $-\infty$ ausgeführt wird, der kleiner als alle realen Schlüssel ist, und dann *deleteMin*. Die *meld*(H')-Operation fällt aus dem Rahmen: sie erfordert bei Binärheaps einen kompletten Neuaufbau.

In diesem Kapitel besprechen wir eine Datenstruktur, die den abstrakten Datentyp „meldable PQ“ implementiert: Fibonacci-Heaps (Abschnitt 4.1), die amortisiert gesehen insbesondere *decreaseKey* besonders schnell erledigt und dadurch zu guten Laufzeiten des Algorithmus von Dijkstra beiträgt.

4.1 Fibonacci-Heaps

Fibonacci-Heaps implementieren *adressierbare Priority Queues*. Es werden alle dort angegebenen Operationen implementiert. Ein Fibonacci-Heap besteht aus *heapgeordneten Bäumen*. Die Datenstruktur ist so entworfen, dass eine amortisierte Analyse gut durchführbar ist.

Fibonacci-Heaps wurden 1984 von Michael L. Fredman und Robert Endre Tarjan vorgestellt.

4.2 Aufbau von Fibonacci-Heaps

Fibonacci-Heaps (oder F-Heaps) bestehen aus einer Menge von Bäumen. Elementarbausteine sind Baumknoten. Ein solcher Knoten v hat folgende Felder (siehe auch Abb. 1):

Zeiger zum Vorgänger: p

Zeiger auf linkes Geschwister: *prev*

Zeiger auf rechtes Geschwister: *next*

Kindzeiger: *child*

Schlüssel: *key* (aus totalgeordneter Menge U)

Informationsteil: *info* (anwendungsabhängig)

Rang: *rank*: integer (Anzahl der Kinder)

Markierung: *marked*: boolean (1: „markiert“, 0: “unmarkiert“.)

Schema:

Organisation in **einem Baum**: Für die Wurzel r ist der Vorgängerzeiger $p(r)$ gleich NIL, für Nichtwurzeln v zeigt $p(v)$ zum Vorgängerknoten. Der Zeiger $child(v)$ zeigt zu einem (beliebigen) Kindknoten von v (falls es einen gibt); die Kinder eines Knotens sind (mittels der *next*- und der *prev*-Zeiger) als zirkuläre doppelt verkettete Liste organisiert. Die Zahl $rank(v)$ gibt die Anzahl der Kinder von v an, die wir auch den *Rang* von v nennen. In Blättern ist der Kindzeiger NIL, der Rang 0.

In jedem Knoten v, der keine Wurzel ist, gilt die **Heapbedingung** $key(v) \geq key(p(v))$.

Ein ***Fibonacci-Heap*** ist eine Kollektion von solchen heapgeordneten Bäumen, wobei die Wurzeln mit Hilfe ihrer *next*- und *prev*-Zeiger als zirkuläre doppelt verkettete Liste

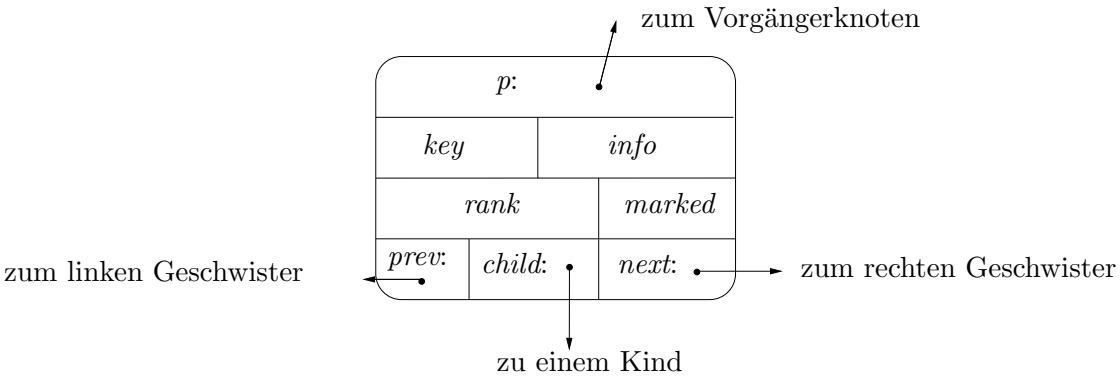

Abbildung 1: Fibonacci-Heaps: Das Format eines Knotens.

organisiert sind. Auf die gesamte Liste wird durch einen Ankerzeiger namens MIN zugegriffen, der auf eine Wurzel mit minimalem Wurzeleintrag zeigt. MIN hat Wert NIL genau dann wenn der Fibonacci-Heap leer ist.

Manche der Knoten sind „markiert“, manche „unmarkiert“. Dies wird explizit im Feld *marked*(v) angegeben. Später werden wir sehen, dass eine solche Markierung bedeutet, dass Knoten v irgendwann einen Kindknoten verloren hat. Markierte Knoten, bei denen ein weiterer Kindknoten entfernt werden soll, werden besonders behandelt.

Wir legen fest: *Wurzelknoten* sind immer *unmarkiert*.

4.3 Potenzialfunktion

Für die amortisierte Analyse der Operationen definieren wir das Potenzial $\Phi(H)$ eines Fibonacci-Heaps H, wie folgt:[1]

$$\Phi(H) := \#(\text{Wurzeln in } H) + 2 \cdot \#(\text{markierte Knoten in } H). \qquad (1)$$

Wenn eine Operation Op auszuführen ist, ist der F-Heap vorher H, der F-Heap nachher H'.

[1] „#“ bedeutet „Anzahl“.

4.4 Einfache Operationen

4.4.1 Erzeugen eines leeren Fibonacci-Heaps

new(): MIN $\leftarrow$ NIL.

Zeitaufwand: $O(1)$.
Echte Kosten: $c_{new} = 1$.
Amortisierte Kosten: $a_{new} = 1$.

Beachte: Das Potenzial des leeren Fibonacci-Heaps H_0 ist $\Phi(H_0) = 0$.

4.4.2 Minimum auslesen

min(): Gib den Eintrag zurück, auf den MIN zeigt, ohne H zu ändern.

Zeitaufwand: $O(1)$.
Echte Kosten: $c_{min} = 1$.
Amortisierte Kosten: $a_{min} = 1$.

4.4.3 Einfügen

insert(e):

Erzeuge neue Wurzel v mit Eintrag e;
falls MIN = NIL, wird v einziger Knoten in der Wurzelliste, sonst:
hänge v direkt neben Knoten MIN in Wurzelliste;
setze MIN auf v um, falls $key(v) < key(\text{MIN})$.

Zeitaufwand: $O(1)$.
Echte Kosten: $c_{insert} = 1$.
Amortisierte Kosten: $a_{insert}(H, H') = c_{insert}(H, H') + \Phi(H') - \Phi(H) = 1 + 1 = 2$
(Die neue Wurzel erhöht das Potenzial um 1.)

4.4.4 Heaperzeugung

makeHeap($\{e_1, \ldots, e_n\}$):
new();
for i **from** 1 **to** n **do** *insert*(e_i).

Zeitaufwand: $O(n)$.
Echte Kosten: $c_{makeHeap} = n + 1$.
Amortisierte Kosten: Sei H_0 der leere Heap, H der resultierende Heap. Dann:
$a_{makeHeap}(H_0, H) = c_{makeHeap}(H_0, H) + \Phi(H) - \Phi(H_0) = n + 1 + n = 2n + 1$.
($\Phi(H) = n$ wegen der n Wurzeln.)

4.4.5 Vereinigen zweier Heaps

meld(H'): Vereinigt zwei F-Heaps H und H'.

Die beiden Heaps sind durch Zeiger MIN und MIN$'$ gegeben.
Hänge die Wurzelliste von H' neben dem Objekt, auf das MIN zeigt, in die Wurzelliste von H ein;
wenn *key*(MIN$'$) $<$ *key*(MIN), setze MIN auf MIN$'$.

Zeitaufwand: $O(1)$.
Echte Kosten: $c_{meld} = 1$.
Amortisierte Kosten: $a_{meld} = c_{meld} = 1$, weil sich das Potenzial nicht ändert.

4.5 Hilfsoperationen *join* und *cleanup*

Wir beschreiben hier zwei für die Datenstruktur „private“ Operationen, die in der *deleteMin*-Operation benötigt werden. Sie können nicht vom Benutzer aufgerufen werden.

Die Hilfsoperation *join*(v, w) kann auf zwei Wurzeln v, w mit gleichem Rang *rank*(v) = *rank*(w) angewendet werden. Sie vereinigt die Bäume mit diesen Wurzeln zu einem Baum mit Wurzelrang *rank*(v)+1. Die Operation ist sehr natürlich: Wenn der Schlüssel in w kleiner ist als der in v, wird v zusätzliches Kind von w, sonst wird w zusätzliches Kind von v.

Prozedur 4.5.1 (*join*(v, w))**.**
Input: Zwei Wurzelknoten v, w von gleichem Rang.

```
(1)  if key(v) ≤ key(w)
(2)  then
(3)       mache w zu neuem Kind von v
(4)          // w wird aus der Wurzelliste ausgeklinkt
             // und in der Kindliste von v direkt neben child(v) eingefügt
(5)       erhöhe rank(v) um 1
(6)  else
(7)       mache v zu neuem Kind von w
(8)       erhöhe rank(w) um 1.
```

Der Zeitaufwand für *join*(v, w) ist $O(1)$.

Mit Hilfe von *join* realisieren wir *cleanup*, eine weitere Hilfsoperation. Zweck dieser Prozedur ist es, die (eventuell sehr lange) Wurzelliste eines F-Heaps zu „kompaktieren“, und zwar so, dass alle verbleibenden Wurzeln verschiedene Ränge haben.

Die Idee ist sehr einfach: Solange man zwei Bäume findet, deren Wurzeln den gleichen Rang haben, wendet man auf diese die *join*-Operation an. Mit jedem *join* verringert sich die Anzahl der Wurzeln um 1, also muss der Prozess irgendwann anhalten. Dann haben alle Wurzeln verschiedene Ränge. Wir müssen dann nur noch die Wurzel mit dem kleinsten Eintrag suchen und den MIN-Zeiger auf diese Wurzel richten.

Die Zahl $D(n) \in \mathbb{N}$ bezeichnet eine (später zu berechnende) obere Schranke für den Rang von Knoten, die in einem F-Heap mit n Einträgen auftreten können. Zur vorläufigen Orientierung: Wir werden sehen, dass $D(n) = O(\log n)$ gewählt werden kann. Das hat den Effekt, dass das Suchen des Minimums am Schluss nur noch Zeit $O(D(n)) = O(\log n)$ dauert.

Das einzige kleine Problem ist, den Ablauf so zu organisieren, dass man nicht lange suchen muss, bis zwei Wurzeln mit dem gleichen Rang gefunden worden sind. Dazu benutzt man einen netten Trick: Man stellt ein Array $A[0..D(n)]$ von Zeigern bereit. Eintrag $A[i]$ kann NIL oder ein Zeiger auf eine Wurzel mit Rang i sein. Wenn man jetzt irgendeine Wurzel r mit Rang j hat, prüft man (in Zeit $O(1)$), ob $A[j] = \text{NIL}$ ist oder auf eine Wurzel zeigt. Im ersten Fall lässt man $A[j]$ auf r zeigen, und behandelt als nächstes irgendeine bislang noch nicht betrachtete Wurzel. Im zweiten Fall wendet man *join* auf r und $A[j]$ an (dadurch wird $A[j]$ wieder NIL), und man hat eine Wurzel mit Rang $j+1$, mit der man genauso weiter verfährt. Wenn man diese Idee konsequent durchführt, erhält man folgendes *cleanup*-Verfahren.

Prozedur 4.5.2 (*cleanup*(L))**.**
Input: Liste L von Wurzeln. // Ein Zeiger MIN wird nicht benötigt
Ausgabe: F-Heap H mit den gleichen Einträgen, Ränge aller Wurzeln verschieden.
Methode:
(1) $A[0..D(n)]$: Array von Zeigern, die anfangs alle NIL sind.
(2) **while** L ist nicht leer **do**
(3) entnehme nächste Wurzel r aus L; $i \leftarrow \mathit{rank}(r)$; $w \leftarrow r$
(4) **while** $A[i] \neq$ NIL **do**
(5) $w \leftarrow \mathit{join}(w, A[i])$; $A[i] \leftarrow$ NIL;
(6) $i \leftarrow i + 1$; (neuer Rang von w)
// der Baum unter w enthält alle Knoten der in (3)–(6) bearbeiteten Bäume
(7) $A[i] \leftarrow w$;
(8) Füge Wurzeln in $A[0..D(n)]$ in neue Wurzelliste ein;
(9) durchlaufe Wurzelliste, um Wurzel r mit minimalem Schlüssel zu finden;
(10) lasse MIN auf r zeigen;
(11) Ausgabe: MIN.

Amortisierte Kostenanalyse: Sei ℓ die anfängliche Länge der Wurzelliste L, $\ell' \leq D(n) + 1$ die Anzahl der Wurzeln in H am Ende. Wir definieren das Potenzial $\Phi(L)$ einer Wurzelliste L analog zu dem Potenzial eines Fibonacci-Heaps wie in (1).

Der Zeitaufwand ist $O(D(n) + \ell + 1)$, denn: Es wird Zeit $O(D(n) + 1)$ für die Initialisierung des Arrays $A[0..D(n)]$, das Durchmustern von $A[0..D(n)]$ in Zeile (8), und für das Finden des Minimums in Zeile (9) benötigt. Es werden ℓ Wurzeln betrachtet und $\ell - \ell'$ *join*-Operationen durchgeführt, die jeweils Zeit $O(1)$ kosten.

Als tatsächliche Kosten setzen wir daher $c_{\mathit{cleanup}}(L, H) = D(n) + \ell + 1$ an.

Amortisierte Kosten: Da $\ell - \ell'$ Wurzeln verschwinden und sich an den Markierungen nichts ändert, ist die Potenzialdifferenz $\Phi(H) - \Phi(L) = \ell' - \ell$. Dabei ist $\ell' \leq D(n) + 1$, und wir erhalten:

$$\begin{aligned} a_{\mathit{cleanup}}(L, H) &= c_{\mathit{cleanup}}(L, H) + \Phi(H) - \Phi(L) \\ &= (D(n) + \ell + 1) + \ell' - \ell \leq 2(D(n) + 1). \end{aligned}$$

(Intuition: Die negative Potenzialdifferenz, die sich durch die verschwindenden Wurzeln ergibt, genügt, um für die *join*-Operationen zu bezahlen.)

Lemma 4.5.3. *Die Operation cleanup auf einer Wurzelliste hat amortisierte Kosten höchstens* $2(D(n) + 1)$*, wobei* n *die Anzahl der Einträge (Knoten) bezeichnet.* □

4.6 Komplexere Operationen

Dieser Abschnitt befasst sich mit der Implementierung und der amortisierten Analyse der Operationen *deleteMin* und *decreaseKey*.

4.6.1 Minimum entnehmen

Prozedur 4.6.1 (*deleteMin*()).
Input: F-Heap H.
Ausgabe: F-Heap H', enthält alle Knoten außer dem Knoten v mit minimalem Eintrag; v.
Methode:
(1) Sei v die Wurzel, auf die MIN zeigt.
(2) Klinke v aus der Wurzelliste aus; Restliste: L; // kein Minimum bekannt!
(3) Durchlaufe Kindliste L' von v, dabei:
(4) für jeden Knoten w in L':
(5) $p(w) \leftarrow$ NIL; *marked*$(w) \leftarrow 0$; hänge w in L ein;
(6) *cleanup*(L); // liefert F-Heap H'
(7) **return**(H', v).

Der Zeitaufwand zur Erstellung von L einschließlich L' (also bis Zeile (5)), ist $O(rank(v)+1) = O(D(n)+1)$.

Für Zeilen (1)–(5) setzen wir echte Kosten $c_{1-5} = rank(v) + 1 \leq D(n) + 1$ an.

Amortisierte Kosten für Zeilen (1)–(5):
$a_{1-5} = c_{1-5} + \Phi(L) - \Phi(H) \leq 1 + 2 \cdot rank(v) \leq 1 + 2D(n)$.
(Die $rank(v)$ neuen Wurzeln erhöhen das Potenzial um je 1. Dies ist nur eine Abschätzung nach oben, da das Potenzial auch geringer sein kann, wenn Kinder von v markiert waren und nun die Markierung verlieren.)

Amortisierte Kosten inklusive *cleanup*(L):

$a_{deleteMin}(H, H') = a_{1-5} + a_{cleanup}(L, H') \leq 3 + 4D(n) = O(D(n))$.

4.6.2 Verringern eines Schlüssels

Der Zweck der Operation *decreaseKey*(v, x) ist, in einem Knoten v den dort vorhandenen Schlüssel y durch einen neuen Schlüssel $x \leq y$ zu ersetzen. Der Knoten v ist hierfür über einen Zeiger (bzw. über eine Referenz) gegeben.

Problem: Wenn $x < key(p(v))$ ist, kann man den Schlüssel nicht einfach verringern, weil dann die Heapbedingung verletzt ist.

Idee: Hänge dann v von seinem Vorgänger $p(v)$ ab und mache v zu einer neuen

Wurzel. Dann kann man $key(v)$ beliebig verringern, ohne dass die Heapbedingung verletzt wird. Man mache sich klar, dass das Abschneiden von v nur konstante Zeit erfordert (man benutzt den Vorgängerzeiger und die Tatsache, dass die Kinder von $p(v)$ als zirkuläre doppelt verkettete Liste organisiert sind), ebenso das Einfügen in die Wurzelliste. Als Ergebnis kennt man auch $p(v)$.

Durch dieses Vorgehen können Knoten Kinder verlieren.

Für die Analyse ist es nötig, dass die Ränge nicht zu groß werden; die Rangschranke $D(n)$ soll logarithmisch in n bleiben. Um zu vermeiden, dass ein Knoten einen großen Rang hat, obwohl es in seinem Unterbaum nicht viele Knoten gibt, werden die Markierungen verwendet. Die allgemeine Strategie ist folgende: Wenn ein Knoten w, der keine Wurzel ist, erstmals ein Kind verliert, wird er markiert. Wenn er nochmals ein Kind verliert, wird er selbst zu einer neuen Wurzel gemacht.

Aus der Sicht einer $decreaseKey(v, x)$-Operation sieht das dann so aus: Wenn $x < key(p(v))$ ist, wird v von seinem Vorgänger $v_1 = p(v)$ abgehängt. Wenn v_1 markiert war, wird v_1 von seinem Vorgänger $v_2 = p(v_1)$ abgehängt und zur Wurzel gemacht. Wenn v_2 markiert war ... . Abgebrochen wird, wenn ein unmarkierter Knoten erreicht wird, was spätestens dann passiert, wenn die Wurzel erreicht wird. Auf diese Weise kann eine ganze Folge $v, v_1, \ldots, v_k$ von Knoten zu neuen Wurzeln werden. Man nennt diesen Vorgang auf englisch „*cascading cut*“ (etwa: „wiederholtes Abschneiden“).

Prozedur 4.6.2 ($decreaseKey(v, x)$)**.**
Input: F-Heap H, (Zeiger auf) Knoten v, Schlüssel $x \leq key(v)$.
Ausgabe: F-Heap H' mit denselben Einträgen, Schlüssel in v auf x erniedrigt.
Methode:

(1) **if** v Wurzel **then**
    $key(v) \leftarrow x$; setze MIN-Zeiger auf v um, falls $x < key(\text{MIN})$; **return**.
(2) **if** $x \geq key(p(v))$ **then** $key(v) \leftarrow x$; **return**.
(3) // Verfolge die Folge $v_0 = v, v_1 = p(v_0), v_2 = p(v_1), \ldots$
(4) $i \leftarrow 1$; $v_1 \leftarrow p(v)$;
(5) Füge v als neue (unmarkierte) Wurzel neben dem MIN-Eintrag ein;
        // v_1 verliert Kind v
(6) $key(v) \leftarrow x$;
(7) setze MIN-Zeiger auf v um, falls $x < key(\text{MIN})$;
(8) **while** v_i ist markiert **do** // v_i ist keine Wurzel
(9) $v_{i+1} \leftarrow p(v_i)$;
(10) Füge v_i als neue Wurzel (unmarkiert) neben dem MIN-Eintrag ein
        // v_{i+1} verliert Kind v_i
(11) $i \leftarrow i + 1$;
(12) **if** v_i ist keine Wurzel **then** markiere v_i.

Die *decreaseKey*-Operation kann eventuell viele neue Wurzeln erzeugen und dabei hohen Zeitaufwand haben. Dabei verschwinden aber Markierungen, wodurch das Potenzial sinkt. Der (negative) Potenzialunterschied wird benutzt, um für diese Kosten zu bezahlen. Man beachte: Es wird jetzt *nicht* versucht, „aufzuräumen“. Vielmehr bleiben die neuen Wurzeln einfach in der Wurzelliste stehen. (*cleanup* würde zu amortisierten Kosten $O(\log n)$ führen!)

Es seien $v_0 = v, v_1, \ldots, v_k$ die Knoten, die zu neuen Wurzeln werden. Möglicherweise wird v_{k+1} markiert (wenn es keine Wurzel ist).

Wir analysieren den Zeitaufwand. Wenn v Wurzel ist oder $x \geq key(p(v))$ gilt, sich also die Struktur nicht ändert, sind die echten Kosten $O(1)$ und das Potenzial ändert sich nicht. Wir betrachten den Fall, wo Knoten umgehängt werden. Der Zeitaufwand ist dann $O(k+1)$ für das Umhängen der Knoten $v_0, v_1, \ldots, v_k$ und das Ändern der Vaterzeiger und Markierungen.

Als echte Kosten setzen wir an: $c_{decreaseKey}(H, H') = k + 1$.

Amortisierte Kosten: Das Potenzial verändert sich wie folgt: $v_1, \ldots, v_k$ sind nicht mehr markiert (Änderung -2), aber werden zu Wurzeln (Änderung $+1$), das Potenzial sinkt also um $k \cdot (2-1) = k$; Knoten v wird zu neuer Wurzel, dadurch steigt das Potenzial sicher um 1. Falls v_{k+1} neu markiert wird, steigt das Potenzial zusätzlich um 2.
Für die Potenzialänderung gilt also: $\Phi(H') - \Phi(H) \leq -k + 3$.

Die amortisierten Kosten erfüllen demnach

$$a_{decreaseKey}(H, H') = c_{decreaseKey}(H, H') + \Phi(H') - \Phi(H) \leq k + 1 + (-k + 3) = 4,$$

sie sind also durch eine Konstante beschränkt.

4.6.3 Löschen

Die Operation *delete*(v), die einen Knoten entfernt, der durch einen Zeiger gegeben ist, kann wie folgt realisiert werden: Man verringert mittels *decreaseKey* den Schlüssel in v auf einen Wert $< key(\text{MIN})$ (liefert H') und führt dann auf H' die Operation *deleteMin*() aus. Die amortisierten Kosten hierfür sind $a_{decreaseKey}(H, H') + a_{deleteMin}(H', H'') \leq 6 + 4D(n) = O(D(n))$.

Damit sind alle Operationen implementiert und die amortisierten Kosten sind ermittelt – bis auf den Schönheitsfehler, dass in den Formeln die obere Schranke $D(n)$ für den maximalen Rang bei n Einträgen vorkommt. Im nächsten und letzten Abschnitt bestimmen wir eine passende Zahl $D(n)$. Hier erklärt sich dann auch endlich der Name „Fibonacci-Heap“.

4.7 Analyse des maximalen Rangs in Fibonacci-Heaps

Wir zeigen, dass es eine Schranke $D(n)$ für den maximalen Rang (Grad, Anzahl der Kinder) eines Knotens in einem F-Heap mit n Einträgen gibt, die $D(n) = O(\log n)$ erfüllt.

Definition 4.7.1. *Die Fibonaccizahlen sind wie folgt definiert:*
$F_0 = 0$, $F_1 = 1$, $F_i = F_{i-2} + F_{i-1}$ *für* $i \geq 2$.
Bekanntlich sind folgendes die ersten Fibonaccizahlen:
$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$.

Betrachte die bekannte Zahl $\Phi = \frac{1}{2}(1 + \sqrt{5}) = 1.618\ldots$.
(Das Verhältnis $1 : \Phi$ heißt „goldener Schnitt“.)
Die Fibonaccizahlen wachsen exponentiell, im wesentlichen mit der Basis Φ:

Fakt 4.7.2. (a) $1 + \Phi = \Phi^2$. (b) *Für* $i \geq 0$ *gilt:* $F_{i+2} \geq \Phi^i$.

Beweis: (a) Nachrechnen.

(b) Vollständige Induktion.

$i = 0 : F_2 = 1 = \Phi^0$.

$i = 1$: $F_3 = 2 > \Phi^1$.

Nun sei $i \geq 2$, die Behauptung gelte für $i - 2$ und $i - 1$. Wir rechnen mit Hilfe der Definition und der Induktionsvoraussetzung:

$$F_{i+2} = F_i + F_{i+1} \overset{\text{I.V.}}{\geq} \Phi^{i-2} + \Phi^{i-1} = \Phi^{i-2}(1 + \Phi) \overset{\text{(a)}}{=} \Phi^i. \qquad \square$$

Fakt 4.7.3. *Für* $i \geq 1$ *gilt:* $1 + F_1 + \ldots + F_i = F_{i+2}$.

(Beispiel: $1+1+1+2+3+5+8 = 21$.) Allgemein ist die Gleichung ganz leicht durch Induktion zu beweisen. Für $i = 1$ ist sie richtig: $1 + F_1 = 1 + 1 = 2 = F_3$. Wenn sie für $i \geq 1$ stimmt (I.V.), folgt:

$$1 + F_1 + \ldots + F_i + F_{i+1} \overset{\text{I.V.}}{=} F_{i+2} + F_{i+1} \overset{\text{Def.}}{=} F_{i+3},$$

das ist die Induktionsbehauptung.

Wir benutzen die Fibonaccizahlen, um auszudrücken, dass die Anzahl der Nachfahren eines Knotens in einem Fibonacci-Heap exponentiell mit seinem Rang wächst.

Lemma 4.7.4. *Betrachte einen Fibonacci-Heap* H. *Dann gilt*:

(a) *Wenn* v *ein Knoten mit* $\text{rank}(v) = i$ *Kindern ist, dann hat der Unterbaum unter* v *mindestens* F_{i+2} *Knoten.*

(b) *Wenn H genau n Einträge hat, dann können die Rangwerte in H nicht größer als $D(n) = \lfloor 1.4405 \log_2 n \rfloor$ sein.*

Beweis: (a) Wir benutzen Induktion über die *Tiefe* des Baums T_v unter dem Knoten v. (Achtung: Man führt nicht Induktion über den Rang!)

Ind.-Anfang: T_v hat Tiefe 0, d. h. er besteht nur aus dem Knoten v. Weil $F_2 = 1$, stimmt die Behauptung.

Ind.-Schritt: Sei v Knoten mit Rang $i \geq 1$.

Es seien $w_1, \ldots, w_i$ die aktuell vorhandenen Kinder von v in der Reihenfolge, in der sie zu Kindern von v gemacht wurden (in *join*-Operationen). Der Unterbaum mit Wurzel w_j sei T_{w_j}. Betrachte ein w_j mit $j \in \{2, \ldots, i\}$. Als w_j Kind von v wurde, waren $w_1, \ldots, w_{j-1}$ schon da, also hatte v zu diesem Zeitpunkt Rang mindestens $j-1$. Nach den Regeln der *join*-Operation (die beiden Knoten haben gleichen Rang) hatte auch w_j zu diesem Zeitpunkt Rang mindestens $j-1$. Nachher kann sich der Rang von w_j höchstens um 1 verringert haben (wenn ein zweites Kind von w_j abgetrennt worden wäre, wäre w_j nach den Regeln für die Behandlung markierter Knoten zur Wurzel gemacht worden). Also gilt aktuell:

$$\mathit{rank}(w_j) \geq j - 2, \text{ für } 2 \leq j \leq i.$$

Nach Induktionsvoraussetzung (T_{w_j} hat geringere Tiefe als T_v) hat T_{w_j} mindestens $F_{(j-2)+2} = F_j$ Knoten, für $2 \leq j \leq i$.

Wir schließen: T_v hat als Knoten mindestens v und w_1 und die Knoten in $T_{w_2}, \ldots, T_{w_i}$, zusammen mindestens

$$1 + F_1 + F_2 + \ldots + F_i = F_{i+2}$$

viele (mit Fakt 4.7.3).

(b) Sei v Knoten mit Rang i in einem Fibonacci-Heap mit n Einträgen. Nach (a) hat T_v mindestens F_{i+2} Knoten, also ist $F_{i+2} \leq n$. Mit Fakt 4.7.2: $n \geq F_{i+2} \geq \Phi^i$. Durch Logarithmieren: $i \leq \log_\Phi n$, oder $i \leq (\log_\Phi 2) \cdot \log_2 n$. Dabei ist $\log_\Phi 2 = (\ln 2)/(\ln \Phi) = 1.4404\ldots < 1.4405$. □

4.8 Zusammenfassung

Fibonacci-Heaps sind eine Implementierung des Datentyps „Adressierbare Priority Queue“. Die Operationen und ihre amortisierten Kosten sind wie folgt, wobei n die Anzahl der Einträge ist:[2]

[2] In der Praxis sind gewöhnliche *quaternäre* Heaps, also solche, die wie Binärheaps über Arrays implementiert sind, bei denen aber die Knoten der gedachten Bäume vier Kinder haben, sehr schnell,

Operation	amortisierte Kosten
$new()$	$O(1)$
$insert(e)$	$O(1)$
$makeHeap(\{e_1, \ldots, e_n\})$	$O(n)$
$meld(H')$	$O(1)$
$min()$	$O(1)$
$deleteMin()$	$O(\log n)$
$delete(v)$	$O(\log n)$
$decreaseKey(v, x)$	$O(1)$

Anwendung: In der Vorlesung „Algorithmen und Datenstrukturen“ im Bachelorstudium wurden die Algorithmen von **Jarník/Prim** (für Minimale Spannbäume) und der Algorithmus von **Dijkstra** (für kürzeste Wege von einem Startknoten aus) vorgestellt. Beide benutzen eine adressierbare Priority Queue. Wenn man diese mit einem Fibonacci-Heap implementiert, und der eingegebene Graph $G = (V, E)$ $n = |V|$ Knoten und $m = |E|$ Kanten hat, erhält man Laufzeiten von

$$O(n \log n + m).$$

Dies liegt daran, dass n *insert*-Operationen und $\leq n$ *deleteMin*-Operationen sowie höchstens m *decreaseKey*-Operationen ausgeführt werden müssen, und dass maximal n Einträge in der Priority Queue liegen. Die gesamten *amortisierten* Kosten sind daher $O(n \log n + m)$, und nach dem allgemeinen Resultat zur Potenzialmethode aus Kapitel 3 gilt dies dann auch für die gesamten *tatsächlichen* Kosten.

Für alle Graphen mit mindestens $n \log n$ Kanten haben diese beiden Algorithmen also lineare Laufzeit!

meld ausgenommen. Fibonacci-Heaps sind theoretisch besser. Andere modernere Heapimplementierungen, die mit Fibonacci-Heaps vergleichbar sind, heißen „Pairing heaps“, „Relaxed heaps“, „Hollow heaps“. Beschreibungen findet man in in der Originalliteratur.