1 Einleitung

Ein Fibonacci-Heap ist eine Datenstruktur, die die unter Punkt 4 erklärten Operationen unterstützt. Wie binomial Heaps sind Fibonacci-Heaps nicht für das effektive Suchen eines Elements entwickelt. Deswegen bekommen Methoden, die auf einen bestimmten Knoten x angewendet werden, einen Zeiger auf x übergeben. Methoden ohne das Entfernen eines Elements laufen in amortisierter Zeit in $\mathcal{O}(1)$, daher sind Fibonacci-Heaps sehr wünschenswert, wenn die Methoden *EXTRACT-MIN* und *DELETE* im Gegensatz zu den anderen Operationen relativ selten ausgeführt werden. Probleme wie das *minimum-spanning-tree* oder das Finden von *single-shortest-path* machen essentiell Gebrauch von Fibonacci-Heaps. Praktisch gesehen sind Fibonacci-Heaps wegen den konstanten Faktoren und der Komplexität der Programmierung weniger erstrebenswert als binomial Heaps und folglich eher von theoretischem Interesse.

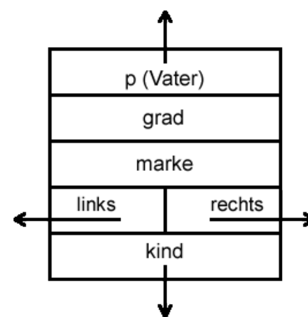
2 Die Struktur

Ein Fibonacci-Heap ist eine Menge von *min-heap-geordneten* Bäumen[1] (siehe binomial Bäume), die sich jedoch nicht auf binomial Bäume beschränkt. Anders als beim binomial-Heap sind die Bäume zwar gerichtet, jedoch ungeordnet. Ein *ungeordneter binomial Baum U* ist wie ein binomial Baum rekursiv definiert. U_0 besteht aus einem Knoten und U_k setzt sich aus den Bäumen $U_{k-1} + U_{k-1}$ zusammen, wobei U_{k-1} ein beliebiges Kind von der Wurzel (nicht zwingend ganz links) ist. Folglich hat die Wurzel von U_k den Grad k, welcher größer ist, als der jedes anderen Knotens und die Kinder $U_1, \ldots, U_{k-1}$ befinden sich in beliebiger Reihenfolge.

Durch die lockerere Struktur als bei binomial Heaps sind verbesserte asymptotische Zeitschranken möglich. Die Hauptidee ist, die Verwaltung der Struktur solange zu verschieben, bis es günstig ist diese auszuführen.

Jeder Knoten x besteht aus:

- einem Zeiger *p[x]* auf seinen Vater,
- einem Zeiger *kind[x]* auf eines seiner Kinder,
- einem Zeiger *links[x]* auf seinen linken Bruder,
- einem Zeiger *rechts[x]* auf seinen rechten Bruder,
- einem Wert *grad[x]* für die Anzahl der Kinder,
- einem Boolean *marke[x]*, der angibt ob x ein Kind verloren hat.



Die Kinder eines Knotens x sind in Form einer zirkularen-doppeltverketteten Liste verknüpft, auch *Kindliste* von x genannt. Die Reihenfolge in der die Kinder erscheinen ist beliebig. Es gilt: Ein Kind ist Einzelkind, wenn $links[x] = rechts[x] = x$ gilt.

Ein Fibonacci-Heap H hat einen Zeiger $min[H]$, der auf die Wurzel mit minimalen Schlüssel zeigt und einen Wert $n[H]$, welcher die Anzahl der gegenwärtigen Knoten in H speichert. Die Wurzeln aller Bäume in H sind ebenfalls durch eine zirkulare-doppeltverkettete Liste (*Wurzelliste*) verbunden.

3 Potentialmethode

Die amortisierten Laufzeitanalysen beruhen auf der Potentialmethode. Für einen gegebenen Heap H ist $t(H)$ die Anzahl der Bäume in der Wurzelliste und $m(H)$ die Anzahl der markierten Knoten in H. Das Potential eines Fibonacci-Heaps H kann durch folgende Funktion berechnet werden:

$$\Phi(H) = t(H) + 2m(H).$$

Wir nehmen an, dass eine Potentialeinheit einen konstanten Arbeitsaufwand bezahlen kann. Dabei ist die Konstante hinreichend groß um für die Kosten jedes spezifischen Arbeitsschrittes in konstanter Zeit aufkommen zu können. Des Weiteren nehmen wir an, dass eine Fibonacci-Heap Anwendung ohne einen Heap startet und somit ein Anfangspotential von $\Phi(H) = 0$ besitzt. Die obere Schranke für amortisierte Kosten = obere Schranke für tatsächliche Gesamtkosten der Operationssequenz + Veränderung des Potentials.

Die bekannte obere Schranke $D(n)$ für den maximalen Grad eines beliebigen Knotens in einem Fibonacci-Heap mit n Knoten ist $D(n) \leq \lfloor \lg(n) \rfloor$.

[1] Angenommen die Operationen *Decrease-Key* und *Delete* werden nie aufgerufen, so verhalten sich die Bäume wie binomial Bäume.

4 Operationen

4.1 Erzeugen eines neuen Fibonacci-Heaps

Die Methode *MAKE-FIB-HEAP* allokiert ein Fibonacci-Heap Objekt H mit den Werten $n[H] = 0$ und $\min[H] = NIL$ und gibt es anschließend zurück. Da $t(H) = 0$ und $m(H) = 0$ ist das Potential $\Phi(H) = 0$. Wir erhalten eine Laufzeit von $\mathcal{O}(1)$.

4.2 Einfügen eines Knotens

Bei der Operation *FIB-HEAP-INSERT(H,x)* wird der Knoten x in die Wurzelliste eingefügt, $n[H]$ inkrementiert und gegebenenfalls der Zeiger $min[H]$ aktualisiert. Das Einfügen von x sowie das Aktualisieren von $min[H]$ und $n[H]$ geht in tatsächlicher Zeit $\mathcal{O}(1)$.

amortisierte Kosten:
Sei H der ursprüngliche Heap und H' der Heap nach dem Einfügen, so ist $t(H') = t(H) + 1$, da ein neuer Baum in der Wurzelliste eingefügt wurde und $m(H') = m(H)$, da kein Knoten markiert wurde. Dies ergibt eine Potentialerhöhung von:

$$\Phi(H') - \Phi(H) = (t(H) + 1) + 2m(H) - (t(H) + 2m(H)) = 1$$

Somit ergibt sich eine amortisierte Laufzeit von $\mathcal{O}(1) + 1 = \mathcal{O}(1)$. Im Gegensatz zum Einfügen eines Knotens in binomial-Heaps wird hier nicht versucht die einzelnen Bäume zu vereinigen.

4.3 Finden des minimalen Elements

Da der minimale Knoten eines Heaps durch den Zeiger $min[H]$ gegeben ist, beträgt die tatsächlich benötigte Zeit $\mathcal{O}(1)$. In dieser Operation ändert sich das Potential des Heaps nicht und somit sind die tatsächlichen Kosten = amortisierte Kosten = $\mathcal{O}(1)$.

4.4 Vereinigen zweier Fibonacci-Heaps

Die Operation *FIB-HEAP-UNION*(H_1, H_2) bekommt zwei Heaps H_1 und H_2 übergeben und verschmilzt die beiden Wurzellisten. Dabei werden beide Heaps in einem neuen Heap H eingefügt. Nach dem Verschmelzen wird der $min[H]$-Zeiger des neuen Heaps auf das kleinere Minima gesetzt.

amortisierte Kosten
Die Potentialänderung lässt sich wie folgt berechnen:

$$\Phi(H) - (\Phi(H_1) + \Phi(H_2)) = t(H) + 2m(H) - ((t(H_1) + 2m(H_1)) + t(H_2) + 2m(H_2)) = 0.$$

Demnach gilt auch hier: amortisierte Kosten = tatsächliche Kosten = $\mathcal{O}(1)$.

4.5 Extrahieren des minimalen Knotens

Das Extrahieren des minimalen Knotens x ist eine der schwierigsten Operationen, da hier die bisher aufgeschobenen Vereinigungen der Bäume in der Wurzelliste ausgeführt werden. Zunächst werden alle Kinder von x in die Wurzelliste eingefügt und anschließend x aus der Wurzelliste entfernt. Der Zeiger $min[H]$ wird auf einen anderen Knoten in der Wurzelliste gesetzt, in unserem Fall auf den Knoten rechts neben dem ursprünglichen Minimum. Danach wird die Methode *CONSOLIDATE(H)* aufgerufen, welche Bäume gleichen Grades solange fusioniert, bis für jeden Grad höchstens eine Wurzel mit diesem Grad übrig bleibt. *CONSOLIDATE(H)* gliedert sich in zwei Arbeitsschritte:

1. Finde in der Wurzelliste zwei Knoten y und z mit dem gleichen Grad und *schlüssel[y]*$\leq$*schlüssel[z]*.

2. Verkette z mit y, dazu entferne z aus der Wurzelliste und mache ihn zu einem Kind von y. Der Grad von y wird inkrementiert und die Marke von z auf false gesetzt.

Die Methode *CONSOLIDATE* verwendet ein Hilfsarray $A[0..D(n(H))]$. Wenn $A[i] = z$ gilt, dann ist z gegenwärtig eine Wurzel mit $grad[z] = i$.

Betrachten wir die tatsächlichen Kosten für *FIB-HEAP-EXTRACT-MIN*. Da der zu entfernende Knoten maximal $D(n)$ Kinder hat, die bearbeitet werden müssen und das Hilfsarray in *CONSOLIDATE* die Größe $D(n)$ hat, ergibt sich ein Beitrag von $\mathcal{O}(D(n))$. Dazu kommen noch die Kosten für das Durchlaufen der

Wurzelliste (in *CONSOLIDATE*), welche maximal $D(n) + t(H) - 1$ groß ist, da die ursprüngliche Wurzelliste die Größe $t(H)$ hatte, zuzüglich der Kinder des entfernten Knotens und abzüglich des extrahierten Knotens. Somit ist der Gesamtarbeitsaufwand $\mathcal{O}(D(n) + t(h))$.

amortisierte Kosten

Das Potential des ursprünglichen Heaps ist $t(H) + 2m(H)$. Das Potential des neuen Heaps ist maximal $(D(n) + 1) + 2m(H)$, da während der Operation keine Knoten markiert werden und höchstens $D(n) + 1$ Wurzeln erhalten bleiben. Dies ergibt amortisierte Kosten von

$$\begin{aligned}\mathcal{O}(D(n) + t(H)) + ((D(n) + 1) + 2m(H)) - (t(H) + 2m(H)) &= \mathcal{O}(D(n)) + \mathcal{O}(t(H)) - t(H) \\ &= \mathcal{O}(D(n)).\end{aligned}$$

Die Einheiten des Potentials können so skaliert werden, dass die in $\mathcal{O}(t(H))$ versteckte Konstante dominiert wird. Die Kosten für eine Verkettung werden durch die Reduktion des Potentials um eins bezahlt. Da sich nach der Verkettung eine Wurzel weniger in der Wurzelliste befindet, reduziert sich auch das Potential des Heaps H um eins.

4.6 Verringern eines Schlüssels

Die Methode *FIB-HEAP-DECREASE-KEY(H,x,k)* verringert den Wert von x auf k. Dafür bekommt die Operation einen Zeiger auf den Knoten x übergeben. Zunächst wird geprüft ob der neue Schlüsselwert kleiner als der aktuelle Schlüssel ist. Ist dies nicht der Fall, so wird die Methode abgebrochen. Ist x eine Wurzel oder *schlüssel[x]≥schlüssel[y]* so müssen keine strukturellen Veränderungen vorgenommen werden, da die Min-Heap-Ordnung nicht verletzt wird. Andernfalls wird x mit Hilfe der Methode $CUT(H, x, y)$ vom Vater y getrennt, als Wurzel eingefügt und *marke[x]* auf $false$ gesetzt, da x nun eine Wurzel ist. Im Falle dessen, dass y bereits markiert ist und demnach x das zweite Kind ist, welches von seinem Vater y getrennt wird, wird die Methode *CASCADING-CUT(H,y)* aufgerufen.

CASCADING-CUT(H,y) prüft die Markierung von y und versucht gegebenenfalls eine *Kaskadentrennung* auf y auszuführen. Ist y unmarkiert so wird die Markierung des Knotens auf *true* gesetzt, da y eben ein Kind verloren hat, andernfalls wird *CUT(H,y,z)* und *CASCADING-CUT(H,z)* aufgerufen, wobei z der Vater von y ist. Die Methode *CASCADING-CUT* ruf sich solange rekursiv auf, bis entweder eine Wurzel oder ein unmarkierter Knoten gefunden wird.

Nachdem die Methode *CASCADING-CUT* terminiert hat, endet *FIB-HEAP-DECREASE-KEY* indem $min[H]$ aktualisiert wird. Der einzige Knoten, dessen Schlüssel sich verändert hat, ist x, da der Schlüsselwert verringert wurde. Demnach ist entweder x ist das neue Minimum oder der Zeiger muss nicht verändert werden.

amortisierte Kosten

Betrachten wir zunächst die tatsächlichen Kosten. Die Methode *FIB-HEAP-DECREASE-KEY* benötigt $\mathcal{O}(1)$ zuzüglich der Zeit für die Kaskadentrennungen. Angenommen *CASCADING-CUT* wird c-mal rekursiv aufgerufen, davon kostet jeder Aufruf, ohne Berücksichtigung der weiteren rekursiven Aufrufe, $\mathcal{O}(1)$ Zeit. Betrachtet man alle Kosten, so ergibt dies tatsächliche Kosten von $\mathcal{O}(c)$ für *FIB-HEAP-DECREASE-KEY*.

Betrachten wir nun die Potentialänderung. Sei H der ursprüngliche Heap. Jeder rekursive Aufruf von *CASCADING-CUT*, bis auf der Letzte, trennt einen markierten Knoten vom Vater, löscht das markierte Bit und fügt den Knoten als Wurzel ein. Demnach befinden sich nach der Methode $t(H) + c$ Bäume in der Wurzelliste ($t(H)$ ursprüngliche Bäume, der Baum mit Wurzel x, sowie die $c - 1$ erzeugten Bäume durch Kaskadentrennung). Die Anzahl der markierten Knoten reduziert sich auf maximal $m(H) - c + 2$, da $c - 1$ Knoten durch Kaskadentrennung unmarkiert wurden und der letzte Aufruf eventuell einen Knoten markiert hat. Dies ergibt eine maximale Potentialänderung von

$$((t(H) + c) + 2(m(H) - c + 2)) - (t(H) + 2m(H)) = 4 - c.$$

Die amortisierten Kosten betragen folglich höchstens

$$\mathcal{O}(c) + 4 - c = \mathcal{O}(1),$$

da wir die Einheiten des Potentials so verändern können, dass die in $\mathcal{O}(c)$ verborgenen Konstanten ausgeglichen werden.

Es ist nun erkennbar warum in der Potentialfunktion der Term für markierte Knoten doppelt genommen wurde. Wird ein markierter Knoten durch Kaskadentrennung gelöscht, so wird das markierte Bit gelöscht und folglich das Potential um 2 verringert. Dies ergibt sich daraus, dass eine Potentialeinheit für das Trennen des markierten Knotens und Löschen des markierten Bits bezahlt werden muss, und eine weitere Einheit für das Einfügen des Knotens als Wurzel und den damit verbundenen Anstieg des Potentials.

4.7 Entfernen eines Knotens

Die Methode *FIB-HEAP-DELETE(H,x)* funktioniert wie das Entfernen eines Knotens in einem binomial Heap. Mit Hilfe der Methode *FIB-HEAP-DECREASE-KEY(H,x,$-\infty$)* wird der Wert von x auf $-\infty$ gesetzt und anschließend mit *FIB-HEAP-EXTRACT-MIN(H)* aus dem Heap entfernt.

amortisierte Kosten

Die amortisierte Zeit für *FIB-HEAP-DELETE* ergeben sich aus der Summe der amortisierten Zeit von *FIB-HEAP-DECREASE-KEY* ($\mathcal{O}(1)$) und *FIB-HEAP-EXTRACT-MIN* ($\mathcal{O}(D(n))$). Dies ergibt eine Gesamtzeit von $\mathcal{O}(D(n)) = \mathcal{O}(\lg n)$.

5 Beschränkung des maximalen Grades

Angenommen die Methoden *FIB-HEAP-DECREASE-KEY* und *FIB-HEAP-DELETE* werden nicht benutzt, so sind nur ungeordnete binomial Bäume in unserem Fibonacci-Heap und wir haben die uns bekannte obere Schranke für den maximalen Grad jedes Knotens, der sich in einen Fibonacci-Heap mit n Knoten befindet, von $D(n) = \lfloor \lg n \rfloor$ (siehe binomial Baum). Die in *FIB-HEAP-DECREASE-KEY* auftretenden Trennungen können jedoch dazu führen, dass unvollständige ungeordnete binomial Bäume entstehen.

Kommen wir nun zu dem Teil, der erklärt wie *Fibonacci-Heaps* zu ihrem Namen kommen. Die $k-te$ Fibonaccizahl F_k kann durch $F_k = F_{k-1} + F_{k-2}$ berechnet werden, wobei $F_0 = 0$ und $F_1 = 1$.

Lemma 1. *Für alle ganzen Zahlen $k \geq 0$ gilt:* $F_{k+2} = 1 + \sum_{i=0}^{k} F_i$

Beweis.
Induktionsanfang: $1 + \sum_{i=0}^{0} F_i = 1 + F_0 = 1 + 0 = F_2$
Induktionsschritt: $F_{k+2} = F_k + F_{k+1} = F_k + (1 + \sum_{i=0}^{k-1} F_i) = 1 + \sum_{i=0}^{k} F_i$ □

Wir wissen, dass $F_{k+2} \geq \phi^k$ mit $\phi = \frac{1+\sqrt{5}}{2}$

Lemma 2. *Sei x ein beliebiger Knoten im Fibonacci-Heap und sei $k = grad[x]$. Dann ist größe(x)$\geq F_{k+2} \geq \phi^k$.*

Beweis. Sei s_k die minimale Größe eines beliebigen Knotens mit Grad k, so gilt: $s_0 = 1, s_1 = 2, s_2 = 3$. Die Anzahl s_k ist höchstens *größe(x)*, wobei s_k monoton mit k steigt. Betrachten wir einen Knoten z für den $grad[z] = k$ und *größe(z)=*s_k gilt. Sei $y_1, y_2, \ldots, y_k$ die Kinder von z in der Reihenfolge ihrer Verkettung. Um eine untere Schranke für s_k zu berechnen zählen wir z selbst, das erste Kind y_1 (größe(y_1)≥ 1) sowie die minimal möglichen Größen der restlichen Kinder von z.

$$\text{größe}(x) \geq s_k \geq 2 + \sum_{i=2}^{k} s_{i-2}$$ □

Wir zeigen nun per Induktion, dass $s_k \geq F_{k+2}$ für alle ganzzahligen, nichtnegativen Zahlen k gilt. Der Induktionsanfang für $k = 0$ und $k = 1$ ist trivial. Für den Induktionsschritt nehmen wir an, dass $k \geq 2$ und $s_i \geq F_{i+2}$ für $i = 0, 1, \ldots, k-1$. Somit gilt:

$$s_k \geq 2 + \sum_{i=2}^{k} s_{i-2} \geq 2 + \sum_{i=2}^{k} F_i = 1 + \sum_{i=0}^{k} F_i = F_{k+2}$$

Damit haben wir gezeigt, dass größe(x)$\geq s_k \geq F_{k+2} \geq \phi^k$.

Korollar:
Der maximale Grad $D(n)$ eines beliebigen Knotens in einem Fibonacci-Heap mit n Knoten ist $\mathcal{O}(\lg n)$.

Beweis. Sei x ein Knoten in einem Fibonacci-Heap mit n Knoten und $k = grad[x]$. Auf Grund von Lemma 2 gilt: $n \geq \text{größe}(x) \geq \phi^k$. Bilden wir auf beiden Seiten den Logarithmus zur Basis ϕ, so erhalten wir $\log_\phi n \geq k$. Da k eine ganze Zahl ist gilt $\lfloor \log_\phi n \rfloor \geq k$. Der maximale Grad $D(n)$ eines beliebigen Knotens ist demnach $\mathcal{O}(\lg n)$. □