Fibonacci Heaps

1 Einleitung

Fibonacci Heaps sind für viele Anwendungen, insbesondere für Graphenalgorithmen von theoretischem Interesse. In der Praxis werden sie jedoch nur selten eingesetzt, wegen ihrer Komplexität und weil sie sich hinter grossen Konstanten (Big Oh Notation) verstecken. Der Hauptsächliche Vorteil ist die decrease_key Funktion, welche amortisiert in O(1) läuft.

2 Struktur

Ein Fibonacci Heap besteht aus einer Menge von min-heap geordneter Bäume, die in einer ungeordneten, doppelt verketteten Wurzelliste zusammengefasst sind. Jeder Knoten hat vier Zeiger: Es gibt einen Zeiger auf den Elternknoten, und einen auf ein beliebiges seiner Kinder. Zusätzlich sind auch alle Kinder eines Knotens in einer doppelt verketteten Liste verbunden, weshalb jeder Knoten zusätzlich noch zwei Zeiger zu seinen Nachbarn hat. Hat ein Knoten nur ein Kind x so gilt für dieses x.left = x.right = x. Der Elternzeiger von Knoten in der Wurzelliste wird auf NULL gesetzt. Doppelt verkettete Listen geben uns zwei Vorteile: Es ist möglich, Knote aus der Liste zu löschen oder zwei Listen zusammenzuhängen, und zwar in konstanter Zeit.

Weiter gibt es bei jedem Knoten noch zwei Felder: Zum einen wird der Grad, also die Anzahl Kinder in der Kindliste gespeichert. Zudem gibt es ein Feld „mark“, welches angibt, ob ein Knoten ein Kind verloren hat seit dem letzen Mal, als es Kind eines anderen Knotens wurde. Neue Knoten erhalten für dieses Feld *false*.

Der Fibonacci Heap wird über einen Zeiger *min* angesprochen, der auf das minimale Element in der Wurzelliste zeigt, und damit auf das globale Minimum. Ist der Heap leer, ist *min* einfach NULL.

3 Operationen auf Fibonacci Heaps

3.1 Heap erstellen, Minimum finden, O(1) real und amortisiert

Diese beiden Operationen sind trivial und werden nicht genauer erklärt

3.2 Ein Element einfügen, O(1) real und amortisiert

Es wird ein neuer Knoten alloziert, sein Grad auf 0 gesetzt, der Eltern- und Kindzeiger seien NULL, er wird nicht markiert und seinen beiden Nachbarzeiger zeigen auf ihn selbst. Nun kann der Knoten in die Wurzelliste eingefügt werden, und gegeben falls der *min*-Zeiger auf das neue Element gesetzt werden.

3.3 Union, O(1) real und amortisiert

Zwei Fibonacci Heaps können einfach vereinigt werden, indem die Wurzellisten der beiden Heaps zusammengehängt werden und das Minimum angepasst wird.

3.4 extractMin, O(D(n)) = 0(log n) amortisiert

Die Funktion *extractMin* oder *deleteMin* funktioniert in drei Phasen

1. Das minimale Element wird gespeichert (um es später zurückzugeben) und aus der Wurzelliste entfernt. Alle Kinder dieses Knotens werden in die Wurzelliste aufgenommen und deren Elternzeiger wird angepasst.
2. Nun müssen wir den *min*-Zeiger anpassen. Da in der Wurzelliste aber bist zu *O(n)* Elemente zu finden sind, werden wir die Wurzelliste zuerst restrukturieren. Wir fordern, dass keine zwei Knoten in der Wurzelliste denselben Grad haben. Dazu benutzen wir ein Array *A* der Länge *D(n)*, wenn *D(n)* der maximale Grad eines Knotens im Fibonacci Heaps mit *n* Elementen ist. Die Wurzelliste wird nun traversiert, und bei jedem Knoten x wird dieser in A[Grad(x)] gespeichert. Ist dort bereits ein Knoten, verletzt dieser unsere Forderung, und wir verbinden die beiden Knoten zu einem min-Heap. Dazu wird einfach der Knoten mit dem grösseren Schlüssel als Kind unter den anderen Knoten gehängt.
3. Nun muss nur noch aufgeräumt und das neue Minimum bestimmt werden. Dazu wird die Wurzelliste geleert (*min := NULL*) und mit dem Array A wieder neu aufgebaut. In selben Durchgang kann auch gleich das neue Minimum gesetzt werden.

3.5 decreaseKey, O(1) amortisiert

Um einen Schlüssel zu verkleinern, gehen wir zu diesem Knoten und verändern den Schlüssel. Wird dadurch die Heapbedingung verletzt und wird der Schlüssel kleiner als der Schlüssel des Elternknoten, so trennen wir den Knoten und hängen ihn in die Wurzelliste, wobei wir „mark“ auf *false* setzen. Der Elternknoten wird nun markiert, ausser diese befindet sich in der Wurzelliste. War er bereits markiert, so trennen wir diesen Knoten ebenfalls; so gehen wir rekursiv bis möglicherweise ganz nach oben. Wenn nötig wird nun noch der *min*-Zeiger aktualisiert, wobei diese entweder unverändert bleibt, oder der Knoten wird, dessen Schlüssel wir gerade verändert haben.

3.6 delete, O(D(n)) = 0(log n) amortisiert

Einen Schlüssel zu Löschen ist einfach: Wir verändern seinen Schlüssel einfach auf minus Unendlich und entfernen dann das Minimum.

4 Analyse

Für die amortisierte Analyse wird eine Potenzialfunktion benötigt. Hierfür sei *t(H)* gleich die Anzahl Elemente in der Wurzelliste und mit *m(H)* werde die Anzahl markierter Elemente im gesamten Heap angegeben. Dann ist die Potenzialfunktion definiert als:

$$\varphi(H) = t(H) + 2m(H)$$

Zu Beginn haben wir einen leeren Heap, das Potenzial ist also 0. Danach ist gemäss Definition nur ein nicht-negatives Potenzial möglich, was uns mit den amortisierten Kosten eine obere Schranke für die Tatsächlichen Kosten gibt.

Es bleibt noch zu zeigen, dass D(n) auch wirklich in O(log(n)) liegt. Dazu schaut man in der Literatur nach.