

Teil 4: Verwaltung von Mengen

Prioritäts-Warteschlangen

Priority-Queues

Eine **Min-Priority-Queue** oder **Min-Prioritäts-Warteschlange** ist eine Datenstruktur zur Speicherung einer linear geordneten Menge M , die die folgenden Operationen unterstützt:

insert(x): $M = M \cup \{x\}$

min: Gibt das minimale Element in M zurück.

delete(x): $M = M \setminus \{x\}$

Häufig schränkt man die Funktion `delete` auf das Minimum ein:

delete_min: Gibt das Minimum zurück und entfernt es aus M .

Max-Priority-Queues sind analog für den Zugriff auf das Maximum definiert.

Heaps

HEAPSORT verwendete bereits eine solche Datenstruktur: **binäre Max-Heaps**.

Diese lassen sich leicht verallgemeinern und für den Zugriff auf das Minimum umdefinieren.

Definition (Allgemeine Min-Heaps)

$M = \{x_1, \dots, x_n\}$ sei eine Menge und $\text{key}: M \rightarrow K$ eine Abbildung von M in eine linear geordnete Menge (z.B. \mathbb{N}). Der Wert $\text{key}(x)$ heißt **Schlüssel von x** .

Ein (gerichteter) Baum T mit Knotenmenge M genügt der **Min-Heap-Bedingung**, wenn für jedes $x \in M$ und jedes Kind y von x **$\text{key}(x) \leq \text{key}(y)$** gilt.

Priority Queues - Weitere Operationen

In vielen Anwendungen sind die Schlüssel nicht statisch, sondern müssen im Laufe des Algorithmus angepasst werden. Um diese Dynamik zu ermöglichen, fordern wir noch die folgende zusätzliche Operation:

`decrease_key(x, k)`: $\text{key}(x) := \min(k, \text{key}(x))$.

Dabei setzen wir voraus, dass auf den Schlüssel und die Position des Elementes in der Priority Queue in **konstanter Zeit** zugegriffen werden kann.

Dies erreichen wir dadurch, dass zusammen mit den eigentlichen Objekten auch Verweise / Zeiger auf die entsprechenden Objekte in dem Priority Queue gespeichert werden.

Bemerkung

*Priority Queues erlauben **nicht** das Suchen nach einem bestimmten Schlüssel. Lediglich der Zugriff auf alle Elemente mit **minimalem** / **maximalem** Schlüssel ist möglich.*

Priority Queues - Weitere Operationen

Zusätzlich verwenden wir die folgende Operation:

`union(x, y)`: Vereinigung zweier **disjunkter** Priority Queues zu einer.

Im Zusammenhang mit Priority Queues wird `union` auch häufig `merge` oder `meld` genannt.

Ein einfaches Beispiel ist z.B. das Mischen zweier sortierter Listen mittels MERGESORT.

Binäre Heaps

Ein **binärer Heap**, wie er von HEAPSORT verwendet wird, verursacht bei n Elementen die folgenden Kosten:

Aufbau des Heaps: $O(n)$ (Aufbauphase von HEAPSORT)

min: $O(1)$ (Zugriff auf die Wurzel)

delete_min: $O(\log n)$ (Löschen der Wurzel + Heapify)

insert: $O(\log n)$ (Anhängen + korrekte Position suchen, Übungsaufgabe)

decrease_key: $O(1)$ für die Wurzel, $O(\log n)$ sonst (Position muss aktualisiert werden)

delete: $O(\log n)$ (Wie Wegnahme bei HEAPSORT)

Binäre Heaps

$\text{union}(S, T)$ für zwei Heaps S und T , mit $|S| = k \leq n = |T|$, kann auf zwei Arten durchgeführt werden:

- 1 Lösche nach und nach das Minimum aus S und füge es in T ein. Dies führt zu einer Laufzeit von $O(k \log(n + k))$
- 2 Baue den Heap vollständig neu auf. Dies führt zu einer Laufzeit von $O(n + k)$.

Anwendungen von Priority Queues

Priority Queues werden in vielen Zusammenhängen benutzt. Als Beispiel sollen hier nur vier Probleme auf Graphen genannt werden, die wir noch genauer betrachten werden:

- 1 Der Algorithmus von Prim für minimale Spannbäume.
- 2 Der Algorithmus von Kruskal für minimale Spannbäume (verwendet auch Union-Find).
- 3 Der kürzeste Wege Algorithmus von Dijkstra.
- 4 Der Algorithmus zur Bestimmung von minimalen Schnitten nach Stoer und Wagner.

Teil 4: Verwaltung von Mengen

Binomial Queues

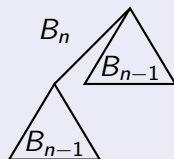
Binomialbäume

Im folgenden stellen wir eine Realisierung einer Priority Queue mittels so genannter **Binomialbäume** vor.

Definition (Binomialbäume)

Für $n \geq 0$ wird der n -te **Binomialbaum** B_n induktiv definiert:

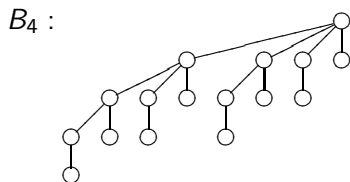
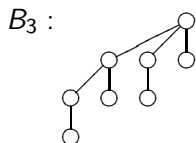
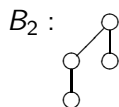
- B_0 besteht lediglich aus einem Knoten.
- B_n , für $n \geq 1$, besteht aus zwei Binomialbäumen B_{n-1} , wobei die Wurzel des einen als Kind an die Wurzel des anderen gehängt wird.



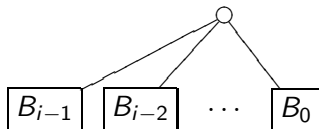
Binomialbäume

Damit ergeben sich B_0, \dots, B_4 als:

B_0 : ○



Insgesamt ergibt sich B_i als



Binomialbäume

Lemma

Für $n \geq 0$ hat der n -te Binomialbaum B_n genau 2^n Knoten und die Tiefe n .

Beweis.

Wir führen eine vollständige Induktion über n durch.

Für $n = 0$ ist die Behauptung offensichtlich erfüllt.

Für $n \geq 1$, ergibt sich die Anzahl der Knoten in B_n als:

$$|B_n| = 2|B_{n-1}| = 2 \cdot 2^{n-1} = 2^n.$$

Da eine Kopie von B_{n-1} an die Wurzel der anderen Kopie gehängt wird, ergibt sich für die Tiefe von B_n :

$$d(B_n) = d(B_{n-1}) + 1 = n - 1 + 1 = n.$$



Binomialbäume - Der Ursprung des Namens

Lemma

Für $n \geq 0$ haben genau $\binom{n}{k}$ Knoten von B_n die Tiefe k .

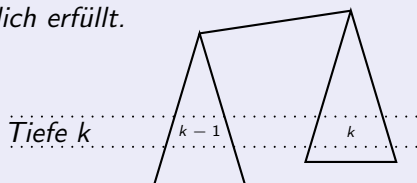
Beweis

Wir führen eine vollständige Induktion über n durch.

Für $n = 0$ ist die Behauptung offensichtlich erfüllt.

Für $n \geq 1$ haben die folgenden Knoten die Tiefe k in B_n :

- Alle Knoten der Tiefe k in der "oberen,, Kopie von B_{n-1}
- Alle Knoten der Tiefe $k - 1$ in der "unteren,, Kopie von B_{n-1}



Beweis (Fortsetzung)

Damit ergibt sich die Gesamtzahl aller Knoten der Tiefe k als:

$$\begin{aligned}\binom{n-1}{k} + \binom{n-1}{k-1} &= \frac{(n-1)!}{k!(n-k-1)!} + \frac{(n-1)!}{(k-1)!(n-k)!} \\ &= \frac{(n-1)!}{(k-1)!(n-k-1)!} \left(\frac{1}{k} + \frac{1}{n-k} \right) \\ &= \frac{(n-1)!}{(k-1)!(n-k-1)!} \frac{n-k+k}{k(n-k)} \\ &= \frac{(n-1)!}{(k-1)!(n-k-1)!} \frac{n}{k(n-k)} \\ &= \frac{n!}{k!(n-k)!} \\ &= \binom{n}{k}\end{aligned}$$



Binomial Queues

Wir wollen jetzt insgesamt N Schlüssel in möglichst wenig Binomialbäumen speichern, ohne dabei **leere** Positionen zu haben.

Da B_n genau 2^n Knoten hat, können wir die Binärdarstellung $(b_0 \dots b_k)_2$ von N nutzen, denn

$$N = \sum_{i=0}^l b_i 2^i.$$

D.h. unser Wald von Binomialbäumen für N Schlüssel enthält genau dann B_j , wenn das j -te Bit in der Binärdarstellung von N eine **Eins** ist.

Definition (Binomial Queue)

Sei $M = \{x_1, \dots, x_N\}$ eine geordnete Menge. Ein **Binomial Queue** ist eine Menge \mathcal{F}_N von Binomialbäumen über der Knoten Menge M , so dass:

- Genau eine Kopie von B_j in \mathcal{F}_N liegt, wenn die j -te Stelle der Binärdarstellung von N eine 1 ist, und
- Alle Bäume in \mathcal{F}_N die Heapbedingung erfüllen.

Binomial Queue - min

Um das Minimum zu bestimmen, muss lediglich das Minimum aller Wurzeln ermittelt werden.

Da höchstens $\lfloor \log N \rfloor + 1$ Bäume in \mathcal{F}_N sind, benötigen wir hierfür höchstens $O(\log N)$ zeit.

Prozedur (min auf Binomial Queues)

Eingabe: *Ein Binomial Queue \mathcal{F} mit N Schlüsseln*

Ausgabe: *Das Minimum aller Schlüssel in \mathcal{F}*

$x := \infty$

für $B \in \mathcal{F}$ **tue**

wenn $\text{root}(B) < x$ **dann** $x := \text{root}(B)$

Ende

Binomial Queues - union

Für die Vereinigung zweier Binomial Queues \mathcal{F}_{N_1} und \mathcal{F}_{N_2} , führen wir im Prinzip eine Addition der beiden Binärdarstellungen von N_1 und N_2 durch. Dazu beobachten wir, dass zwei Binomialbäume $B_n^{(1)}$ und $B_n^{(2)}$ gleicher Größe, zu einem Binomialbaum B_{n+1} zusammengefasst werden. Dabei muss der Baum mit der größeren Wurzel an den Baum mit der kleineren Wurzel gehängt werden. Dadurch erfüllt der neu entstandenen Binomialbaum offensichtlich weiterhin die Heapbedingung. Die Kosten für dieses Zusammenhängen sind $O(1)$.

Prozedur (union - Vereinigung zweier Bäume)

Eingabe: Zwei Binomialheaps $B_n^{(1)}$ und $B_n^{(2)}$ gleicher Größe.

Ausgabe: Der vereinigte Binomialheap B_{n+1} .

$x := \text{root}(B_n^{(1)})$, $y := \text{root}(B_n^{(2)})$

wenn $x < y$ **dann** $\text{pred}(y) := x$

sonst $\text{pred}(x) := y$

Binomial Queues - union

Um nun zwei Binomial Queues \mathcal{F}_{N_1} und \mathcal{F}_{N_2} zu einer $\mathcal{F} := \mathcal{F}_{N_1+N_2}$ vereinigen, führen wir im Prinzip eine Addition der Binärdarstellungen der beiden Zahlen N_1 und N_2 durch.

Für die j -te Stelle betrachten wir bis zu zwei Bäume $B_j^{(1)}$ und $B_j^{(2)}$ aus \mathcal{F}_{N_1} und \mathcal{F}_{N_2} , sowie möglicherweise einen Übertrag $B_j^{(3)}$ von der vorhergegangenen Stellen.

- Liegt keiner dieser drei Bäume vor, passiert nichts.
- Liegt nur einer dieser drei Bäume vor, füge ihn in \mathcal{F} ein.
- Liegen zwei vor, vereinige Sie und speichere sie als Übertrag $B_{j+1}^{(3)}$.
- Liegen drei vor, füge einen in \mathcal{F} ein, vereinige die beiden anderen und speichere sie als Übertrag $B_{j+1}^{(3)}$.

Da wir für jede Stelle der Binärdarstellungen konstante Zeit benötigen, erhalten wir insgesamt eine Laufzeit von $O(\log N_1 + \log N_2)$.

Binomial Queues - union

Prozedur (min auf Binomial Queues)

Eingabe: Zwei Binomial Queues \mathcal{F}_{N_1} und \mathcal{F}_{N_2} .

Ausgabe: Die vereinigte Binomial Queue $\mathcal{F} := \mathcal{F}_{N_1+N_2}$.

```
 $B_3 := \emptyset;$  // Initialisiere den Übertrag
für  $j = 1, \dots, \lfloor \max(N_1, N_2) \rfloor + 1$  tue
     $B_1 := B(\mathcal{F}_{N_1}, j)$   $B_2 := B(\mathcal{F}_{N_2}, j);$  // Wähle die Bäume der Größe  $j$ 
    wenn  $B_1 \neq \emptyset$  und  $B_2 \neq \emptyset$  dann
        Füge  $B_3$  in  $\mathcal{F}$  ein
         $B_3 := \text{union}(B_1, B_2)$ 
    sonst wenn  $B_1 = \emptyset$  und  $B_2 = \emptyset$  dann Füge  $B_3$  in  $\mathcal{F}$  ein
    sonst wenn  $B_1 \neq \emptyset$  dann
        wenn  $B_3 = \emptyset$  dann Füge  $B_1$  in  $\mathcal{F}$  ein sonst  $B_3 = \text{union}(B_1, B_3)$ 
    sonst
        wenn  $B_3 = \emptyset$  dann Füge  $B_2$  in  $\mathcal{F}$  ein sonst  $B_3 = \text{union}(B_2, B_3)$ 
    Ende
Ende
wenn  $B_3 \neq \emptyset$  dann Füge  $B$  in  $\mathcal{F}$  ein.
```

Binomial Queues - union - ein Beispiel

	3	2	1	0
F_5				
F_7				
Übertrag				
Ergebnis				

Binomial Queues - insert und delete_min

`insert(x)` ist einfach die Vereinigung zweier Binomial Queues \mathcal{F}_N und \mathcal{F}_0 , wobei die zweite lediglich x enthält. Damit ist die Laufzeit von `insert` $O(\log N)$.

`delete_min` verwendet ebenfalls `union`.

Wir nehmen an, dass das Minimum in der Wurzel eines Baumes $B_i \in \mathcal{F}_n$ steht. Durch das Löschen dieses Knotens, zerfällt der Baum in Teilbäume der Form B_{i-1}, \dots, B_0 , d.h. je einer von allen Größen bis $i-1$. Diese bilden eine Binomial Queue \mathcal{F}_{2^i-1} mit $2^i - 1$ Schlüsseln.

Um die resultierende Binomial Queue zu erhalten wird B_i aus \mathcal{F}_N entfernt, d.h. man erhält \mathcal{F}_{N-2^i} und dieser Wald wird mit dem entstehenden Wald \mathcal{F}_{2^i-1} mittels `union` zu \mathcal{F}_{N-1} vereinigt.

Da B_i 2^i Knoten enthält und Tiefe i hat, ist die Tiefe jedes Baumes in \mathcal{F}_N durch $O(\log(N))$ beschränkt. Die Laufzeit dieser Prozedur ist somit $O(\log N)$.

Binomial Queues - decrease_key

Um den Schlüssel $\text{key}(x)$ eines Elementes x auf k herunterzusetzen, wird einfach das Element x mit erniedrigtem Schlüssel solange mit seinem direkten Vorgänger verglichen und nach oben getauscht, bis die Heapbedingung wieder hergestellt ist.

Damit ergibt sich die Laufzeit als $O(\log N)$.

Prozedur (decrease_key auf Binomial Queues)

Eingabe: Eine Binomial Queue \mathcal{F} , ein Wert x und ein Schlüssel k

Ausgabe: Eine Binomial Queue \mathcal{F} mit $\text{key}(x) = \min(k, \text{key}(x))$

wenn $k < \text{key}(x)$ **dann**

$\text{key}(x) := k$

solange $\text{key}(x) < \text{key}(\text{pred}(x))$ **tu** Tausche $\text{pred}(x)$ und x

Ende

Binomial Queues - delete

Um einen beliebigen Eintrag x zu löschen gehen wir folgendermaßen vor:

- 1 Senke $\text{key}(x)$ mittels `decrease_key` auf $-\infty$.
- 2 Lösche mittels `delete_min` das Minimum.

Damit benötigt das Entfernen eines beliebigen Schlüssels $O(\log N)$ Zeit.

Binomial Queues - Implementation

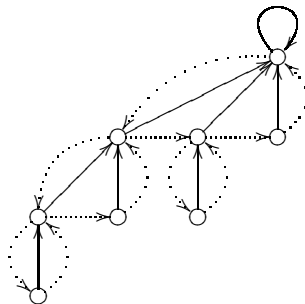
Bislang haben wir recht abstrakt über die Binomial Queues und die Operationen auf ihnen gesprochen. Im Folgenden wollen wir uns mit ihrer Implementation beschäftigen.

Jean Vuillemin schlug 1978 vor, jeden Knoten durch eine Struktur mit folgendem Inhalt darzustellen:

- Dem Wert/Schlüsselpaar $(x, \text{key}(x))$.
- Einen Zeiger $\text{pred}(x)$ auf den direkten Vorgänger.
- Einen Zeiger $\text{llink}(x)$ auf den linkesten Sohn mit $\text{llink}(x) = x$ falls x ein Blatt ist.
- Einen Zeiger $\text{rlink}(x)$ auf das nächste Kind des Vorgängers. Folgen x keine weiteren Kinder, gilt $\text{rlink}(x) = \text{pred}(x)$.

Binomial Queues - Implementation

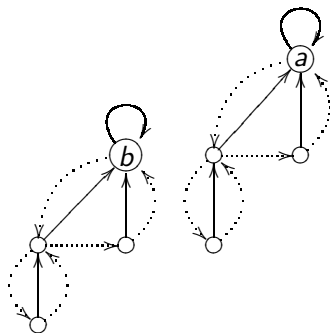
Damit ergibt sich für B_3 das folgende Bild:



Binomial Queues - Implementation

Die Vereinigung zweier B_2 ergibt sich dann folgendermaßen:

- 1 $\text{pred}(b) := a$
- 2 $\text{rlink}(b) := \text{llink}(a)$
- 3 $\text{llink}(a) := b$



Binomial Queues - Zusammenfassung

Mit der Implementation nach Vuillemin können die Operationen mit den oben beschriebenen Verfahren mit den jeweiligen Kosten implementiert werden

Operation	Binary Heaps	Binomial Queues
min	$O(1)$	$O(\log N)$
union	$O(N_1 \log(N_1 + N_2))$	$O(\log N_1 + \log N_2)$
insert	$O(\log N)$	$O(\log N)$
delete_min	$O(\log N)$	$O(\log N)$
decrease_key	$O(\log N)$	$O(\log N)$
delete	$O(\log N)$	$O(\log N)$