4.4.2 Dynamisches perfektes Hashing

Sei $U = \{0, \ldots, p-1\}$ für eine Primzahl p. Zunächst einige mathematische Grundlagen.

Definition 36

$\mathcal{H}_{k,n}$ bezeichne in diesem Abschnitt die Klasse aller Polynome $\in \mathbb{Z}_p[x]$ vom Grad $< k$, wobei mit $\vec{a} = (a_0, \ldots, a_{k-1}) \in U^k$

$$h_{\vec{a}}(x) = \left(\left(\sum_{j=0}^{k-1} a_j x^j \right) \bmod p \right) \bmod n \text{ für alle } x \in U\,.$$

Definition 37

Eine Klasse $\mathcal{H}$ von Hashfunktionen von U nach $\{0, \ldots, n-1\}$ heißt (c, k)-universell, falls für alle paarweise verschiedenen $x_0, x_1, \ldots, x_{k-1} \in U$ und für alle $i_0, i_1, \ldots, i_{k-1} \in \{0, \ldots, n-1\}$ gilt, dass

$$\Pr[h(x_0) = i_0 \wedge \cdots \wedge h(x_{k-1}) = i_{k-1}] \leq \frac{c}{n^k} ,$$

wenn $h \in \mathcal{H}$ gleichverteilt gewählt wird.

Satz 38

$\mathcal{H}_{k,n}$ ist (c,k)-universell mit $c = (1+\frac{n}{p})^k$.

Beweis:

Da $\mathbb{Z}_p$ ein Körper ist, gibt es für jedes Tupel $(y_0,\ldots,y_{k-1}) \in U^k$ genau ein Tupel $(a_0,\ldots,a_{k-1}) \in \mathbb{Z}_p^k$ mit

$$\sum_{j=0}^{k-1} a_j x_r^j = y_r \bmod p \qquad \text{für alle } 0 \le r < k.$$

Damit folgt, dass

$$\begin{aligned}
&|\{\vec{a};\ h_{\vec{a}}(x_r) = i_r \text{ für alle } 0 \le r < k\}| \\
&= |\{(y_0,\ldots,y_{k-1}) \in U^k;\ y_r = i_r \bmod n \text{ für alle } 0 \le r < k\}| \\
&\le \left\lceil \frac{p}{n} \right\rceil^k .
\end{aligned}$$

Beweis (Forts.):

Da es insgesamt p^k Möglichkeiten für $\vec{a}$ gibt, folgt

$$\begin{aligned}\Pr[h(x_r) = i_r \text{ für alle } 0 \leq r < k] &\leq \left\lceil \frac{p}{n} \right\rceil^k \cdot \frac{1}{p^k} \\ &= \left(\left\lceil \frac{p}{n} \right\rceil \cdot \frac{n}{p} \right)^k \cdot \frac{1}{n^k} \\ &< \left(1 + \frac{n}{p} \right)^k \cdot \frac{1}{n^k} \,.\end{aligned}$$

□

Kuckuck-Hashing für dynamisches perfektes Hashing

Kuckuck-Hashing arbeitet mit zwei Hashtabellen, T_1 und T_2, die je aus den Positionen $\{0, \ldots, n-1\}$ bestehen. Weiterhin benötigt es zwei $(1+\delta, \mathcal{O}(\log n))$-universelle Hashfunktionen h_1 und h_2 für ein genügend kleines $\delta > 0$, die die Schlüsselmenge U auf $\{0, \ldots, n-1\}$ abbilden.

Jeder Schlüssel $x \in S$ wird entweder in Position $h_1(x)$ in T_1 oder in Position $h_2(x)$ in T_2 gespeichert, aber nicht beiden. Die *IsElement*-Operation prüft einfach, ob x an einer der beiden Positionen gespeichert ist.

Die *Insert*-Operation verwendet nun das Kuckucksprinzip, um neue Schlüssel einzufügen. Gegeben ein einzufügender Schlüssel x, wird zunächst versucht, x in $T_1[h_1(x)]$ abzulegen. Ist das erfolgreich, sind wir fertig.

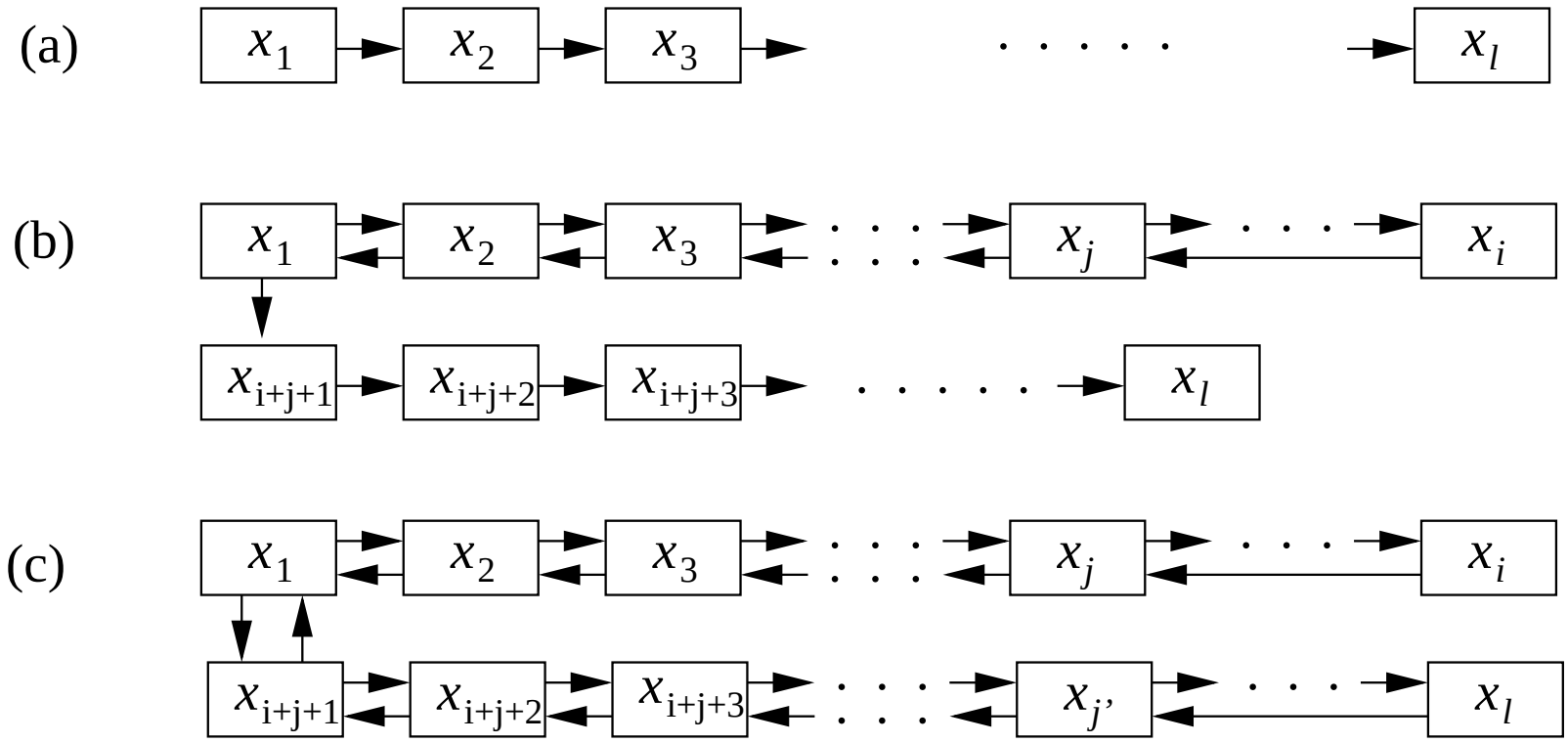
Falls aber $T_1[h_1(x)]$ bereits durch einen anderen Schlüssel y besetzt ist, nehmen wir y heraus und fügen stattdessen x in $T_1[h_1(x)]$ ein. Danach versuchen wir, y in $T_2[h_2(y)]$ unterzubringen. Gelingt das, sind wir wiederum fertig. Falls $T_2[h_2(y)]$ bereits durch einen anderen Schlüssel z besetzt ist, nehmen wir z heraus und fügen stattdessen y in $T_2[h_2(y)]$ ein. Danach versuchen wir, z in $T_1[h_1(z)]$ unterzubringen, und so weiter, bis wir endlich den zuletzt angefassten Schlüssel untergebracht haben. Formal arbeitet die Insert-Operation wie folgt:

if $T_1[h_1(x)] = x$ **then return fi**
repeat MaxLoop times
(a) exchange x und $T_1[h_1(x)]$
(b) **if** $x =$ NIL **then return fi**
(c) exchange x und $T_2[h_2(x)]$
(d) **if** $x =$ NIL **then return fi**
od
rehash(); *Insert*(x)

Für die Analyse der Zeitkomplexität nehmen wir an, dass die Schleife t-mal durchlaufen wird (wobei $t \leq$ MaxLoop).

Es gilt, die folgenden zwei Fälle zu betrachten:

1. Die *Insert*-Operation gerät während der ersten t Runden in eine Endlosschleife
2. Dies ist nicht der Fall



Insert bei Kuckuck-Hashing; Endlosschleife im Fall (c)

Erster Fall: Sei $v \leq l$ die Anzahl der verschiedenen angefassten Schlüssel. Dann ist die Anzahl der Möglichkeiten, eine Endlosscheife zu formen, höchstens

$$v^3 \cdot n^{v-1} \cdot m^{v-1} ,$$

da es maximal v^3 Möglichkeiten für die Werte i, j und l gibt, n^{v-1} viele Möglichkeiten für die Positionen der Schlüssel, und m^{v-1} viele Möoglichkeiten für die Schlüssel außer x_1.

Angenommen, wir haben $(1, v)$-universelle Hashfunktionen, dann passiert jede Möglichkeit nur mit einer Wahrscheinlichkeit von n^{-2v}. Falls $n \geq (1+\delta)m$ für eine Konstante $\delta > 0$, dann ist die Wahrscheinlichkeit für den Fall 1 höchstens

$$\sum_{v=3}^{l} v^3 n^{v-1} m^{v-1} n^{-2v} \leq \frac{1}{nm} \sum_{v=3}^{\infty} v^3 (m/n)^v = \mathcal{O}(\frac{1}{m^2}) .$$

Zweiter Fall:

Lemma 39

Im zweiten Fall gibt es eine Schlüsselfolge der Länge mindestens $l/3$ in $x_1, \ldots, x_l$, in der alle Schlüssel paarweise verschieden sind.

Beweis:

Nehmen wir an, dass die Operation zu einer bereits besuchten Position zurückkehrt, und seien i und j so definiert wie in der Abbildung. Falls $l \leq i + j$, dann bilden die ersten $j - 1 \geq (i + j - 1)/2 \geq l/2$ Schlüssel die gesuchte Folge.

Für $l \geq i + j$ muss eine der Folgen $x_1, \ldots, x_{j-1}$ und $x_{i+j-1}, \ldots, x_l$ die Länge mindestens $l/3$ haben.

Beweis (Forts.):

Sei also $x'_1, \ldots, x'_v$ eine solche Folge verschiedener Schlüssel in $x_1, \ldots, x_{2t}$ der Länge $v = \lceil (2t-1)/3 \rceil$. Dann muss entweder für $(i_1, i_2) = (1, 2)$ oder für $(i_1, i_2) = (2, 1)$ gelten, dass

$$h_{i_1}(x'_1) = h_{i_1}(x'_2), h_{i_2}(x'_2) = h_{i_2}(x'_3), h_{i_1}(x'_3) = h_{i_1}(x'_4), \ldots$$

Gegeben x'_1, so gibt es m^{v-1} mögliche Folgen von Schlüsseln $x'_2, \ldots, x'_v$. Für jede solche Folge gibt es zwei Möglichkeiten für (i_1, i_2). Weiterhin ist die Wahrscheinlichkeit, dass die obigen Positionsübereinstimmungen gelten, höchstens $n^{-(v-1)}$, wenn die Hashfunktionen aus einer $(1, v)$-universellen Familie stammen. Also ist die Wahrscheinlichkeit, dass es irgendeine Folge der Länge v gibt, so dass Fall 2 eintritt, höchstens

$$2(m/n)^{v-1} \leq 2(1+\delta)^{-(2t-1)/3+1}.$$

Diese Wahrscheinlichkeit ist polynomiell klein in m, falls $t = \Omega(\log m)$ ist. □

Beweis (Forts.):

Zusammen ergibt sich für die Laufzeit von *Insert*:

$$1 + \sum_{t=2}^{\mathrm{MaxLoop}} (2(1+\delta)^{-(2t-1)/3+1} + \mathcal{O}(1/m^2))$$

$$\leq 1 + \mathcal{O}\left(\frac{\mathrm{MaxLoop}}{m^2}\right) + \mathcal{O}\left(\sum_{t=0}^{\infty}((1+\delta)^{-2/3})^t\right)$$

$$= \mathcal{O}\left(1 + \frac{1}{1-(1+\delta)^{-2/3}}\right) = \mathcal{O}(1+1/\delta)\,.$$

□

Beweis (Forts.):

Überschreitet m irgendwann einmal die Schranke $n/(1+\delta)$, so wird n hochgesetzt auf $(1+\delta)n$ und neu gehasht. Unterschreitet auf der anderen Seite m die Schranke $n/(1+\delta)^3$, so wird n verringert auf $n/(1+\delta)$ und neu gehasht. Auf diese Weise wird die Tabellengröße linear zur Anzahl momentan existierender Schlüssel gehalten. Der Aufwand für ein komplettes Rehashing ist $\mathcal{O}(n)$, so dass amortisiert über $\Theta(n)$ Einfügungen und Löschungen der Aufwand nur eine Konstante ist. □

Originalarbeiten zu Hashverfahren:

J. Lawrence Carter, Mark N. Wegman:
Universal Classes of Hash Functions,
Proc. STOC 1977, pp. 106–112 (1977)

Gaston H. Gonnet:
Expected Length of the Longest Probe Sequence in Hash Code Searching,
Journal of the ACM **28**(2), pp. 289–304 (1981)

Martin Dietzfelbinger et al.:
Dynamic Perfect Hashing: Upper and Lower Bounds,
SIAM J. Comput. **23**(4), pp. 738–761 (1994)

Und weiter:

Rasmus Pagh, Flemming Friche Rodler:
Cuckoo Hashing,
Proc. ESA 2001, LNCS **2161**, pp. 121–133 (2001)

Luc Devroye, Pat Morin, Alfredo Viola:
On Worst Case Robin-Hood Hasing,
McGill Univ., TR **0212** (2002)

5. Vorrangwarteschlangen - Priority Queues

Priority Queues unterstützen die Operationen
Insert(), *Delete()*, *ExtractMin()*, *FindMin()*, *DecreaseKey()*, *Merge()*.

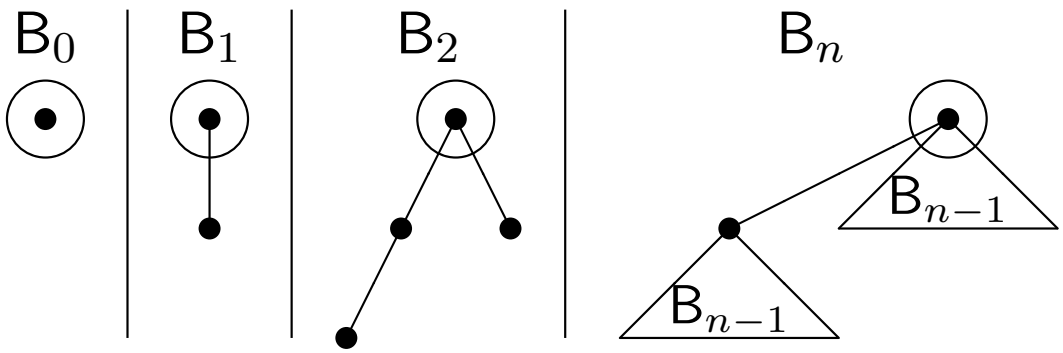
Priority Queues per se sind nicht für *IsElement()*-Anfragen, also zum Suchen geeignet. Falls benötigt, muss dafür eine passende Wörterbuch-Datenstruktur parallel mitgeführt werden.

5.1 Binomial Queues (binomial heaps)

Binomialwarteschlangen (Binomial Queues/Binomial Heaps) werden mit Hilfe von Binomialbäumen konstruiert.

Definition 40

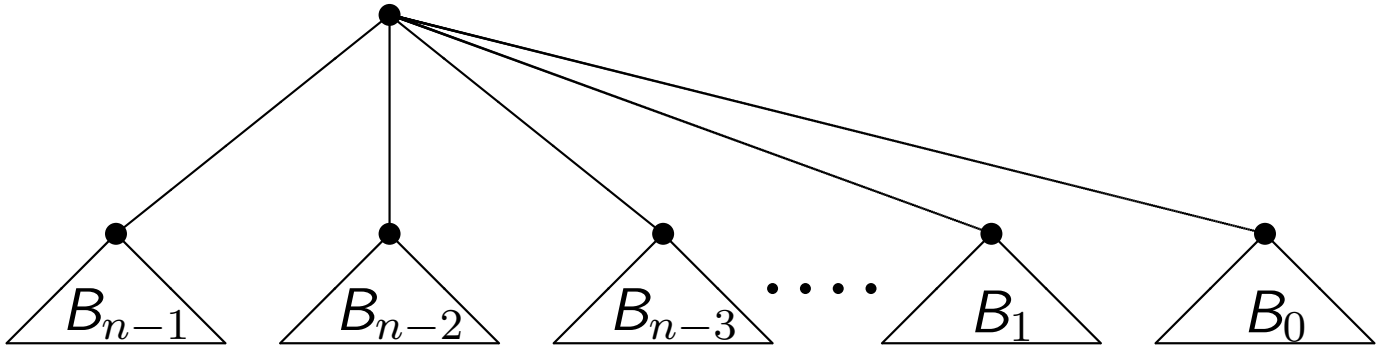
Die Binomialbäume B_n, $n \geq 0$, sind rekursiv wie folgt definiert:



Achtung: Binomialbäume sind offensichtlich keine Binärbäume!

Lemma 41

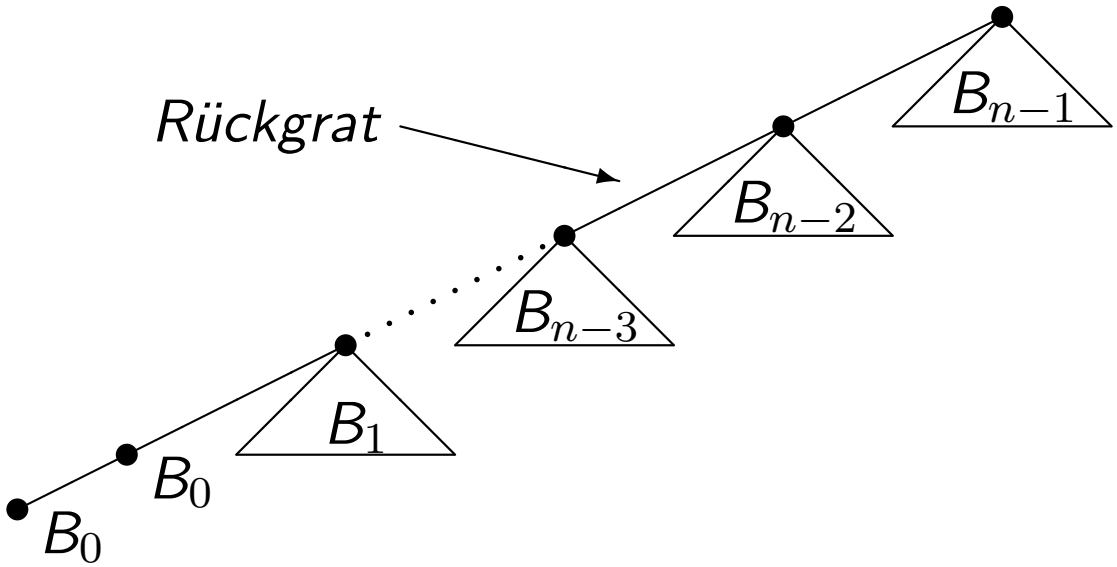
Ein B_n lässt sich wie folgt zerlegen:



Erste Zerlegung von B_n: der Breite nach

Lemma 41

Ein B_n lässt sich wie folgt zerlegen:



Zweite Zerlegung von B_n: der Tiefe nach

Satz 42

Für den Binomialbaum B_n gilt:

1. *B_n hat 2^n Knoten.*
2. *Die Wurzel von B_n hat Grad n.*
3. *B_n hat Höhe/Tiefe n.*
4. *B_n hat $\binom{n}{i}$ Knoten in Tiefe i.*

Beweis:

- zu 1: Induktion unter Verwendung der Definition
- zu 2: Siehe erste Zerlegung von Binomialbäumen
- zu 3: Siehe zweite Zerlegung von Binomialbäumen
- zu 4: Induktion über n:

  (I) $n = 0 : B_0$ hat 1 Knoten in Tiefe $i = 0$ und 0 Knoten in Tiefe $i > 0$, also $\binom{0}{i}$ Knoten in Tiefe i.

  (II) Ist N_i^n die Anzahl der Knoten in Tiefe i im B_n, so gilt unter der entsprechenden Induktionsannahme für B_n, dass

$$N_i^{n+1} = N_i^n + N_{i-1}^n = \binom{n}{i} + \binom{n}{i-1} = \binom{n+1}{i},$$

  woraus wiederum per Induktion die Behauptung folgt.

□

Bemerkung:

Eine mögliche Implementierung von Binomialbäumen ist das Schema „Pointer zum ersten Kind und Pointer zum nächsten Geschwister“.

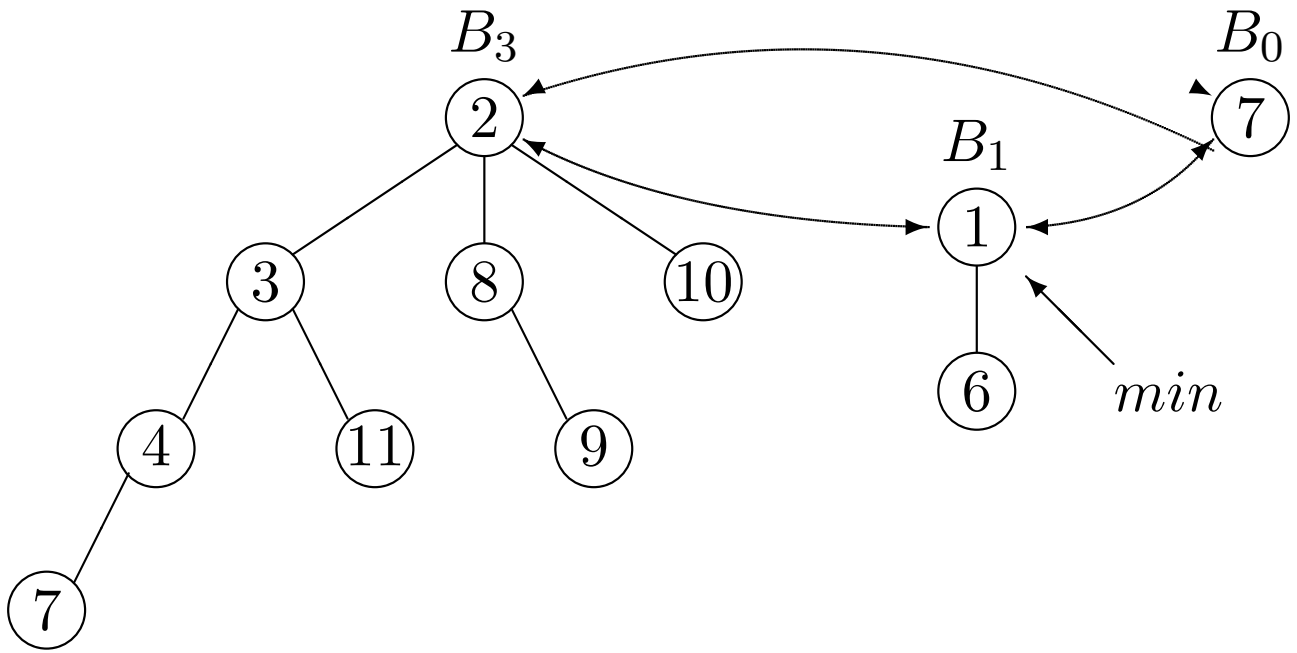
Definition 43

Eine Binomial Queue mit n Elementen wird wie folgt aufgebaut:

1. Betrachte die Binärdarstellung von n.
2. Für jede Position i mit einem 1-Bit wird ein Binomialbaum B_i benötigt (der 2^i Knoten hat).
3. Verbinde die (Wurzeln der) Binomialbäume in einer doppelt verketteten zirkulären Liste.
4. Beachte, dass innerhalb jedes Binomialbaums die Heap-Bedingung erfüllt sein muss. Dadurch enthält die Wurzel eines jeden Binomialbaums gleichzeitig sein minimales Element.
5. Richte einen Min-Pointer auf das Element in der Wurzel-Liste mit minimalem Schlüssel ein.

Beispiel 44

Beispiel: $n = 11 = (1011)_2$**:**



Operationen für Binomial Queues:

- *IsElement*: Die Heap-Bedingung wirkt sich nur auf die Anordnung der Datenelemente innerhalb jedes einzelnen Binomialbaums aus, regelt aber nicht, in welchem Binomialbaum ein gegebenes Element gespeichert ist. Tatsächlich kann ein Element in jedem der vorhandenen Binomialbäume stehen. Das Suchen ist hier nicht effizient implementierbar, denn im worst-case müsste jedes Element der Binomialbäume angeschaut werden.

  Also wären zum Suchen eines Elements schlimmstenfalls $2^0 + 2^1 + \cdots + 2^{\lfloor \log n \rfloor - 1} = \Theta(n)$ Elemente zu betrachten.

  Daher wird eine gesonderte Datenstruktur, etwa ein Suchbaum, für die *IsElement*-Operation verwendet. Damit ist Suchen mit Zeitaufwand $\mathcal{O}(\log n)$ möglich.

Operationen für Binomial Queues:

- *Merge*: Das Vorgehen für das Merge (disjunkte Vereinigung) zweier Binomial Queues entspricht genau der Addition zweier Binärzahlen: Ein einzelnes B_i wird übernommen, aus zwei B_i's wird ein B_{i+1} konstruiert. Damit die Heap-Bedingung erhalten bleibt, wird als Wurzel des entstehenden B_{i+1} die Wurzel der beiden B_i mit dem kleineren Schlüssel genommen.

  Allerdings kann ein solcher „Übertrag“ dazu führen, dass im nächsten Verschmelzungsschritt drei B_{i+1} zu verschmelzen sind. Dann wird unter Beachtung obiger Regel ein B_{i+2} gebildet und einer der B_{i+1} unverändert übernommen.

Algorithmus:

for $i := 0, 1, 2, 3, \ldots$ **do**
  if ($\exists$ genau 3 B_is) **then**
    verbinde zwei der B_i's zu einem B_{i+1} und behalte das dritte B_i
  elif ($\exists$ genau 2 B_is) **then**
    verbinde sie zu einem B_{i+1}
  elif ($\exists$ genau ein B_i) **then**
    übernimm es
  fi
od

Zeitkomplexität:

$$\boxed{\mathcal{O}(\log n) = \mathcal{O}(\log(n_1 + n_2))}$$

Operationen für Binomial Queues:

- *Insert*: Die *Insert*-Operation wird einfach durch eine *Merge*-Operation mit einem B_0 implementiert.

Beispiel 45

1	0	0	1	0	1	1	1
(B_7)			(B_4)		(B_2)	(B_1)	(B_0)
							1
1	0	0	1	1	0	0	0

Zeitkomplexität: $\mathcal{O}(\log n)$

Operationen für Binomial Queues:

- Initialisierung einer BQ durch n sukzessive *Insert*-Operationen: Hier ergäbe die obige Abschätzung einen Gesamtaufwand von $\mathcal{O}(n \log n)$, was allerdings schlecht abgeschätzt ist.

Wir sind an den Kosten zum sukzessiven Aufbau einer Binomial Queue mit n Elementen interessiert, die übrigens identisch sind zum Aufwand für das binäre zählen von 0 bis n, wenn jeder Zählschritt und jeder Übertrag jeweils eine Zeiteinheit kosten:

$$\underbrace{\underbrace{0+1}_{1}+1}_{2\text{ Schritte}}+1+1+1$$

$$\underbrace{\underbrace{\underbrace{\underbrace{\underbrace{0+1}_{1}+1}_{2\text{ Schritte}}+1}_{3}+1}_{4}+1}_{5}$$

Sei a_n die Anzahl der Schritte (Einfügen des neuen Elements und anschließende Verschmelzungen) beim Mergen eines Knotens zu einer Queue mit $n-1$ Elementen. Dann gilt für a_n:

n	a_n
1	1
2	2
... 4	1 3
... 8	1 2 1 4
... 16	1 2 1 3 1 2 1 5
... 32	1 2 1 3 1 2 1 4 1 2 1 3 1 2 1 6

Man erkennt sofort, dass jede Zeile (außer den beiden ersten) doppelt so lange ist wie die vorhergehende und dabei die Folge aller vorhergehenden Zeilen enthält, wobei das letzte Element noch um eins erhöht ist.

Damit ergibt sich für den Gesamtaufwand der sukzessiven Erzeugung einer Binomial Queue mit n Elementen

$$\begin{aligned} T_n &= \sum_{i=1}^{n} a_i \\ &= n + \left\lfloor \frac{n}{2} \right\rfloor + \left\lfloor \frac{n}{4} \right\rfloor + \left\lfloor \frac{n}{8} \right\rfloor + \cdots \\ &\leq 2n\,. \end{aligned}$$