3.4 Hashing

Hash- oder auch Streuspeicherverfahren sind dadurch gekennzeichnet, dass jedem Schlüssel aus einem Universum U durch eine (effizient berechenbare) Funktion h (die Hashfunktion) eine Adresse $\in \{0, \ldots, m-1\}$ zugeordnet wird. Der Schlüssel k wird dann im Prinzip im Element $A[h(k)]$ des m-elementigen Feldes $A[0..m-1]$ gespeichert.

Im Idealfall sind dann die Wörterbuchoperationen effizient ausführbar, bestimmt durch den Aufwand für die Berechnung von $h(k)$.

Da $|U| \gg m$, kann h natürlich nicht injektiv sein, und es kommt zu Kollisionen:

$$h(x) = h(y) \text{ für gewisse } x \neq y\,.$$

Normalerweise sind wir nur an einer kleineren Teilmenge $S \subset U$ von Schlüsseln interessiert ($|S| = n$), und wir haben, der Speichereffizienz halber, z.B.

$$m \in \{n, \ldots, 2n\}$$

Selbst in diesem Fall sind Kollisionen jedoch zu erwarten, wie z.B. das so genannte Geburtstagsparadoxon zeigt:

In einer Menge von mindestens 23 zufällig gewählten Personen gibt es mit Wahrscheinlichkeit größer als $\frac{1}{2}$ zwei Personen, die am gleichen Tag des Jahres Geburtstag haben (Schaltjahre werden hier nicht berücksichtigt).

Das Geburtstagsparadoxon ist äquivalent dazu, dass in einer Hashtabelle der Größe 365, die mindestens 23 Einträge enthält, wobei die Hashadressen sogar zufällig gewählt sind, bereits mit einer Wahrscheinlichkeit von mehr als $\frac{1}{2}$ eine Kollision vorkommt.

Satz 167

In einer Hashtabelle der Größe m mit n Einträgen tritt mit einer Wahrscheinlichkeit von mindestens $1 - e^{-n(n-1)/(2m)}$ mindestens eine Kollision auf, wenn für jeden Schlüssel die Hashadresse gleichverteilt ist.

Beweis:

Die Wahrscheinlichkeit, dass auch der i-te eingefügte Schlüssel keine Kollision verursacht, ist

$$\frac{m-(i-1)}{m}.$$

Die Wahrscheinlichkeit, dass für alle n Schlüssel keine Kollision eintritt, ist daher

$$\Pr\{\text{keine Kollision}\} = \prod_{i=1}^{n} \frac{m-(i-1)}{m} = \exp\left(\sum_{i=0}^{n-1} \ln\left(1-\frac{i}{m}\right)\right)$$

Satz 167

In einer Hashtabelle der Größe m mit n Einträgen tritt mit einer Wahrscheinlichkeit von mindestens $1 - e^{-n(n-1)/(2m)}$ mindestens eine Kollision auf, wenn für jeden Schlüssel die Hashadresse gleichverteilt ist.

Beweis:

Da $\ln(1+x) \leq x$ für alle $x \in \mathbb{R}$, folgt damit

$$\begin{aligned}
\Pr\{\text{keine Kollision}\} &= \exp\left(\sum_{i=0}^{n-1} \ln\left(1 - \frac{i}{m}\right)\right) \\
&\leq \exp\left(\sum_{i=0}^{n-1} \left(-\frac{i}{m}\right)\right) \\
&= \exp\left(-\frac{n(n-1)}{2m}\right) .
\end{aligned}$$

□

Korollar 168

Hat eine Hashtabelle der Größe m mindestens $\omega(\sqrt{m})$ Einträge, wobei für jeden Schlüssel die Hashadressen gleichverteilt sind, so tritt mit Wahrscheinlichkeit $1 - o(1)$ eine Kollision auf.

Definition 169

Eine Hashfunktion h heißt ideal, wenn gilt

$$(\forall i \in \{0, \ldots, m-1\}) \left[\sum_{\substack{k \in U \\ h(k)=i}} \Pr\{k\} = \frac{1}{m} \right] ,$$

wobei $\Pr\{k\}$ die Wahrscheinlichkeit ist, mit der der Schlüssel $k \in U$ vorkommt.

Beispiele für gebräuchliche Hashfunktionen:

1. **Additionsmethode:**

$$h(k) = \left(\sum_{i=0}^{r} \left((k \bmod 2^{\alpha_i}) \operatorname{div} 2^{\beta_i} \right) \right) \bmod m$$

für $\alpha_i \geq \beta_i \in \{0, \ldots, \ell(|U|) - 1\}$

2. **Multiplikationsmethode:** Sei $a \in (0, 1)$ eine Konstante.

$$h(k) = \lfloor m \cdot (ka - \lfloor ka \rfloor) \rfloor$$

Eine gute Wahl für a ist z.B.

$$a = \frac{\sqrt{5} - 1}{2} = \frac{1}{\varphi} \approx 0,618\ldots,$$

wobei $\varphi = \frac{1+\sqrt{5}}{2}$ der Goldene Schnitt ist.

Beispiele für gebräuchliche Hashfunktionen:

3. **Teilermethode:**

$$h(k) = k \bmod m\,,$$

wobei m keine Zweier- oder Zehnerpotenz und auch nicht von der Form $2^i - 1$ sein sollte. Eine gute Wahl für m ist i.A. eine Primzahl, die nicht in der Nähe einer Zweierpotenz liegt.

3.4.1 Kollisionsauflösung

Hashing durch Verkettung

In jedem Feldelement der Hashtabelle wird eine lineare Liste der Schlüssel abgespeichert, die durch die Hashfunktion auf dieses Feldelement abgebildet werden. Die Implementierung der Operationen is_member, insert und delete ist offensichtlich.

Sei $\alpha = \frac{n}{m}$ der Füllgrad der Hashtabelle. Dann beträgt, bei Gleichverteilung der Schlüssel, die Länge einer jeden der linearen Listen im Erwartungswert α und der Zeitaufwand für die Suche nach einem Schlüssel im

$$\text{erfolgreichen Fall } 1 + \frac{\alpha}{2}$$

$$\text{erfolglosen Fall } 1 + \alpha$$

Für die Operationen insert und delete ist darüber hinaus lediglich ein Zeitaufwand von $O(1)$ erforderlich.

Hashing durch Verkettung

In jedem Feldelement der Hashtabelle wird eine lineare Liste der Schlüssel abgespeichert, die durch die Hashfunktion auf dieses Feldelement abgebildet werden. Die Implementierung der Operationen is_member, insert und delete ist offensichtlich.

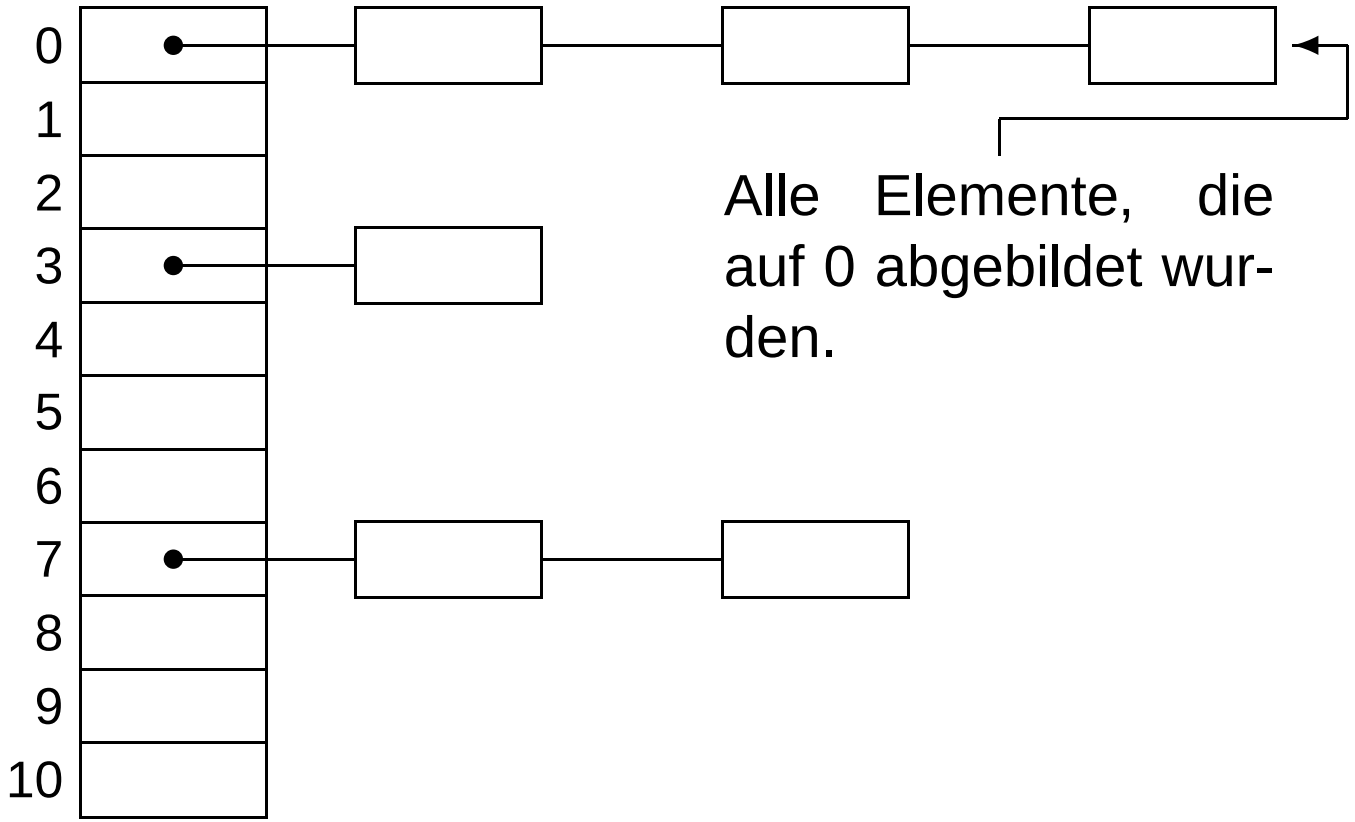
Sei $\alpha = \frac{n}{m}$ der Füllgrad der Hashtabelle. Dann beträgt, bei Gleichverteilung der Schlüssel, die Länge einer jeden der linearen Listen im Erwartungswert α und der Zeitaufwand für die Suche nach einem Schlüssel im

$$\text{erfolgreichen Fall } 1 + \frac{\alpha}{2}$$

$$\text{erfolglosen Fall } 1 + \alpha$$

Hashing durch Verkettung (engl. chaining) ist ein Beispiel für ein geschlossenes Hashverfahren.

Beispiel 170 (Hashing durch Verkettung)



Lineare Sondierung

Man führt eine *erweiterte Hashfunktion* $h(k, i)$ ein. Soll der Schlüssel k neu in die Hashtabelle eingefügt werden, wird die Folge $h(k, 0), h(k, 1), h(k, 2), \ldots$ durchlaufen, bis die erste freie Position in der Tabelle gefunden wird.

Beim Verfahren des linearen Sondierens (engl. linear probing) ist

$$h(k, i) = (h(k) + i) \pmod{m}.$$

Für den Zeitaufwand für die Suche nach einem Schlüssel kann man zeigen im

$$\text{erfolgreichen Fall: } \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$$

$$\text{erfolglosen Fall: } \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

Lineare Sondierung

Man führt eine *erweiterte Hashfunktion* $h(k, i)$ ein. Soll der Schlüssel k neu in die Hashtabelle eingefügt werden, wird die Folge $h(k, 0), h(k, 1), h(k, 2), \ldots$ durchlaufen, bis die erste freie Position in der Tabelle gefunden wird.

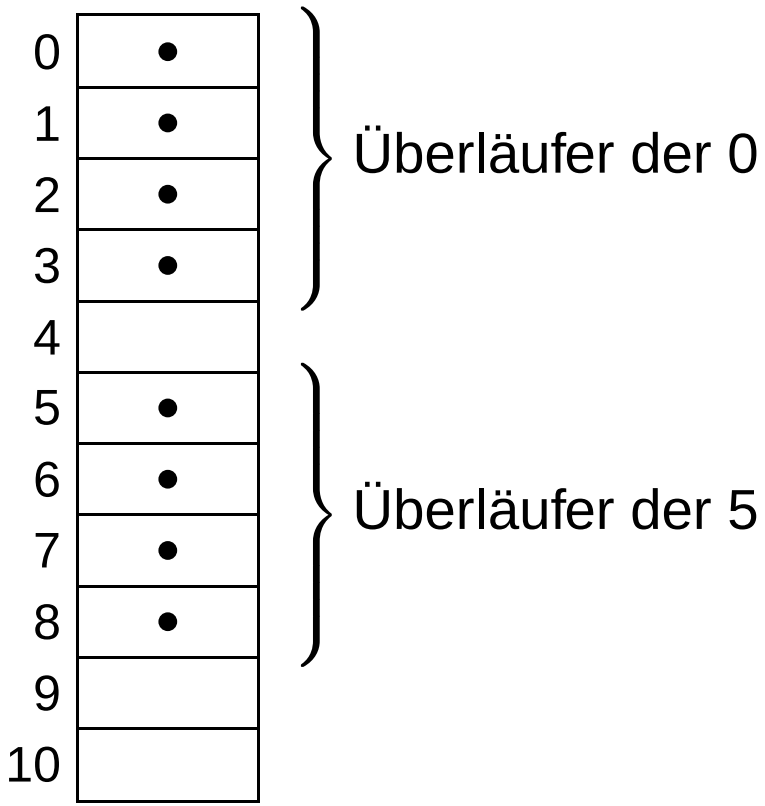
Beim Verfahren des linearen Sondierens (engl. linear probing) ist

$$h(k, i) = (h(k) + i) \pmod{m}.$$

Die Implementierung von insert ist kanonisch, bei delete ist jedoch Sorge zu tragen, dass keine der Sondierungsfolgen $h(k, 0), h(k, 1), h(k, 2), \ldots$ unterbrochen wird. Oft werden gelöschte Elemente daher nicht wirklich entfernt, sondern nur als *gelöscht* markiert.

Ein großes Problem der linearen Sondierung ist, dass sich durch Kollisionen große zusammenhängende Bereiche von belegten Positionen der Hashtabelle ergeben können, wodurch die Suchfolge bei nachfolgenden Einfügungen sehr lang werden kann und auch solche Bereiche immer mehr zusammenwachsen können. Dieses Problem wird als primäre Häufung (engl. primary clustering) bezeichnet.

Beispiel 171 (Lineare Sondierung)



Quadratische Sondierung

Beim Verfahren des quadratischen Sondierens (engl. quadratic probing) ist

$$h(k,i) = (h(k) - (-1)^i(\lceil i/2 \rceil^2)) \pmod m .$$

Surjektivität der Folge $h(k,0), h(k,1), h(k,2), \ldots$ kann garantiert werden, wenn z.B. m prim und $m \equiv 3 \bmod 4$ ist. Für den

Zeitaufwand für die Suche nach einem Schlüssel kann man zeigen im

$$\text{erfolgreichen Fall: } 1 - \frac{\alpha}{2} + \ln\left(\frac{1}{1-\alpha}\right)$$

$$\text{erfolglosen Fall: } 1 + \frac{\alpha^2}{1-\alpha} + \ln\left(\frac{1}{1-\alpha}\right)$$

Für die Implementierung von delete gilt die gleiche Bemerkung wie beim linearen Sondieren.

Doppeltes Hashing

Beim Verfahren des doppelten Hashings (engl. double hashing) ist

$$h(k,i) = (h(k) + i \cdot h'(k)) \pmod m .$$

Dabei sollte $h'(k)$ für alle k teilerfremd zu m sein, z.B.

$h'(k) = 1 + (k \bmod (m-1))$ oder $h'(k) = 1 + (k \bmod (m-2))$.

Surjektivität der Folge $h(k,0), h(k,1), h(k,2), \ldots$ kann dann garantiert werden, wenn m prim ist. Für den Zeitaufwand für die Suche nach einem Schlüssel kann man zeigen im

$$\text{erfolgreichen Fall: } \frac{1}{\alpha} \ln \left(\frac{1}{1-\alpha} \right)$$

$$\text{erfolglosen Fall: } \frac{1}{1-\alpha}$$

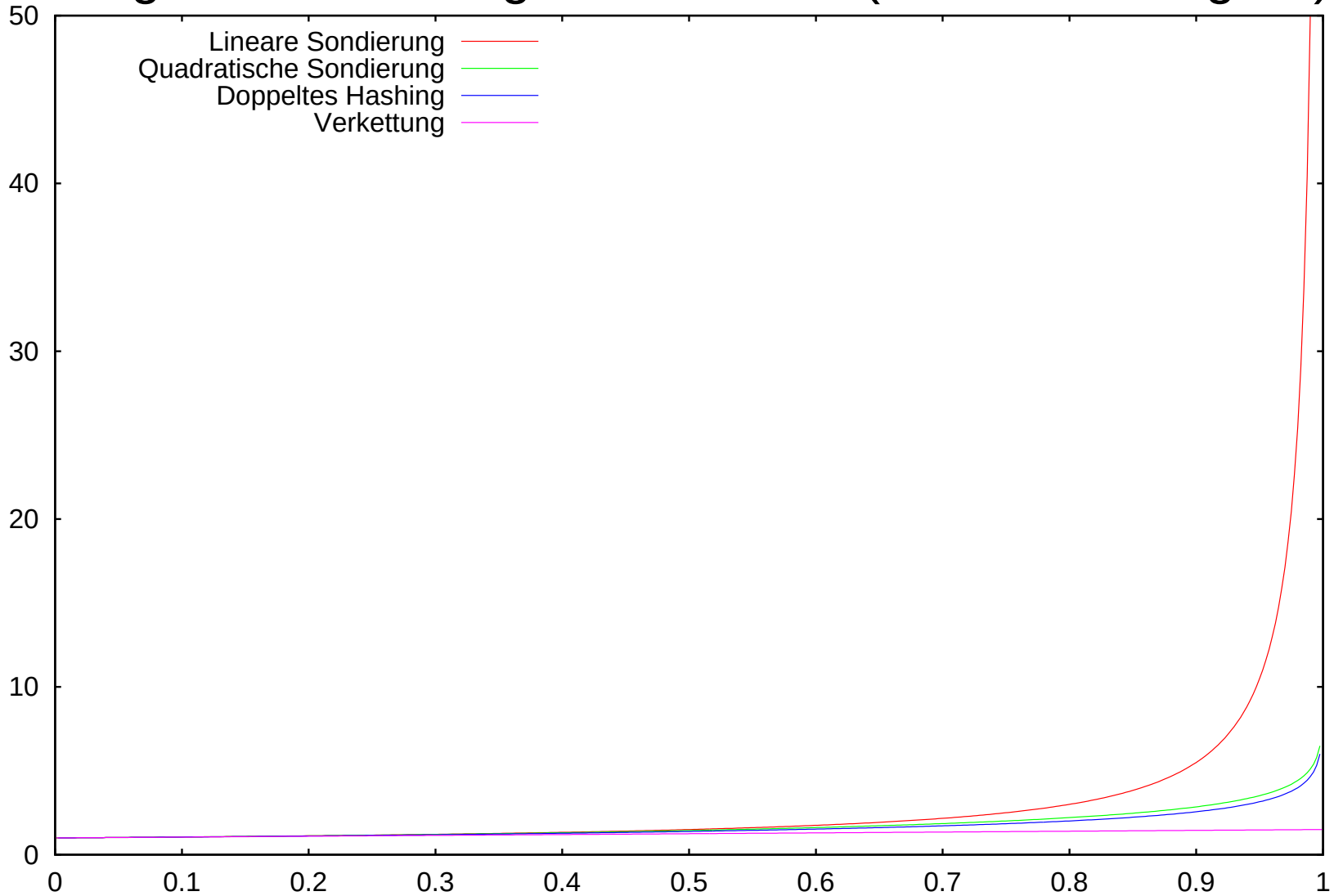
Für die Implementierung von delete gilt die gleiche Bemerkung wie beim linearen Sondieren.

Bemerkung:
Hashing mit Kollisionsauflösung durch lineares oder quadratisches Sondieren, wie auch doppeltes Hashing, sind Beispiele für so genannte

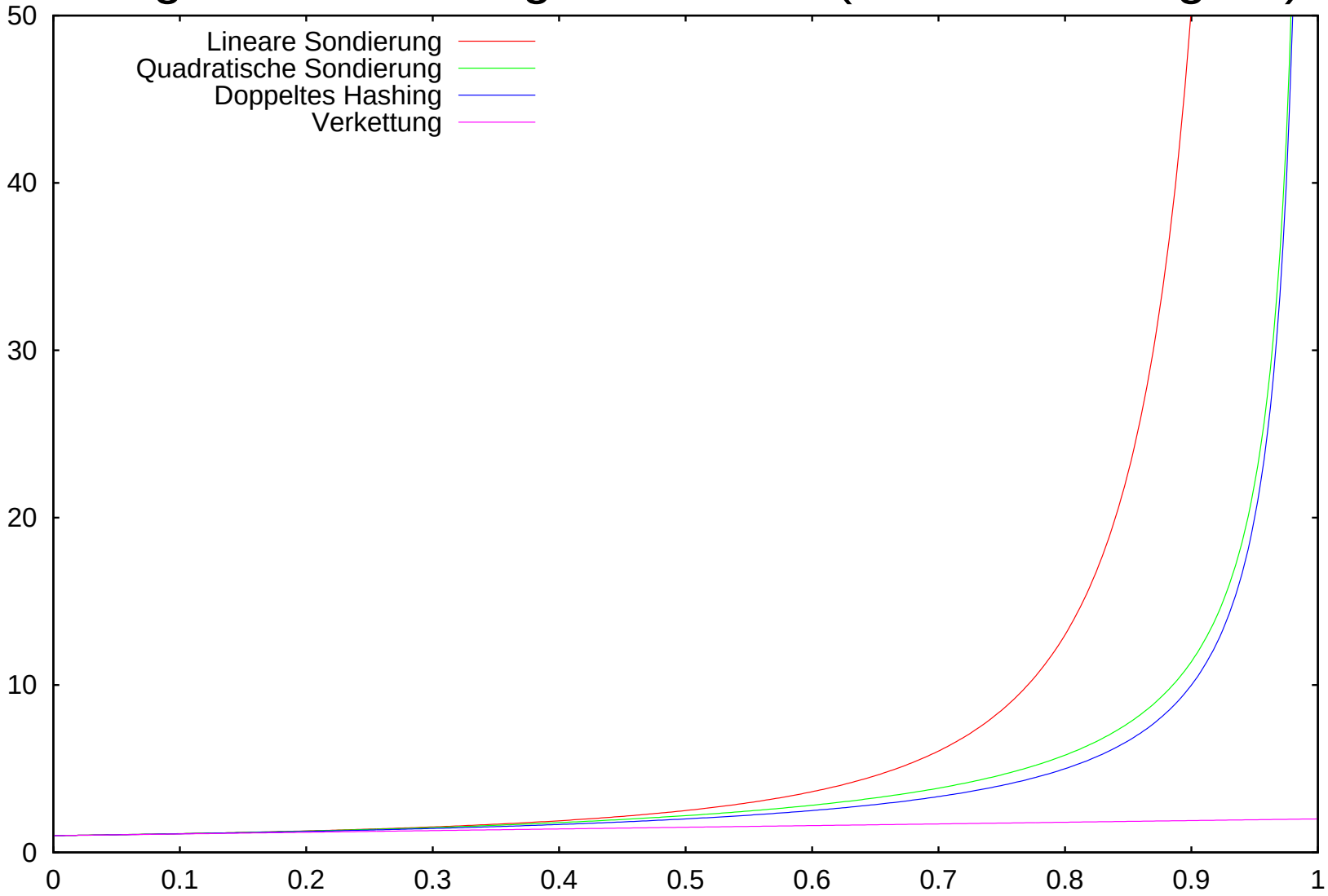
offene Hashverfahren,

da die Schlüssel nicht unbedingt an ihrer (ersten) Hashadresse abgespeichert werden.

Vergleiche für erfolgreiche Suche (abh. vom Füllgrad)



Vergleiche für erfolglose Suche (abh. vom Füllgrad)



3.4.1 Universelle Hashfunktionen

Definition 172

Eine Menge $\mathcal{H}$ von Hashfunktionen heißt universell, falls gilt:

$$\forall x, y \in U, x \neq y \underbrace{\frac{|\{h \in \mathcal{H};\ h(x) = h(y)\}|}{|\mathcal{H}|}}_{\Pr\{h(x)=h(y)\}} \leq \frac{1}{m}$$

Satz 173

Sei $\mathcal{H}$ eine universelle Familie von Hashfunktionen (für eine Hashtabelle der Größe m), sei $S \subset U$ eine feste Menge von Schlüsseln mit $|S| = n \leq m$, und sei $h \in \mathcal{H}$ eine zufällig gewählte Hashfunktion (wobei alle Hashfunktionen $\in \mathcal{H}$ gleich wahrscheinlich sind). Dann ist die erwartete Anzahl von Kollisionen eines festen Schlüssels $x \in S$ mit anderen Schlüsseln in S kleiner als 1.

Beweis:

Es gilt

$$\mathbb{E}[\#\text{ Kollisionen mit } x] = \sum_{\substack{y \in S \\ y \neq x}} \frac{1}{m} = \frac{n-1}{m} < 1\,.$$

Sei nun $m = p$ eine Primzahl. Dann ist $\mathbb{Z}_p = \{0, 1, \ldots, p-1\}$ ein Körper.

Wir betrachten im Folgenden $U = \{0, 1, \ldots, p-1\}^{r+1}$.

Sei die Familie $\mathcal{H} = \{h_\alpha;\ \alpha \in U\}$ von Hashfunktionen definiert durch

$$h_\alpha(x_0, \ldots, x_r) = \left(\sum_{i=0}^{r} \alpha_i \cdot x_i \right) \bmod p$$

Satz 174

$\mathcal{H}$ ist universell.

Beweis:

Sei $x \neq y$, o.B.d.A. $x_0 \neq y_0$. Dann gilt $h_\alpha(x) = h_\alpha(y)$ gdw

$$\alpha_0(y_0 - x_0) = \sum_{i=1}^{r} \alpha_i(x_i - y_i)\,.$$

Für jede Wahl von $(\alpha_1, \ldots, \alpha_r)$ gibt es daher genau ein α_0, das zu einer Kollision führt.

Es gilt daher

$$\frac{|\{h \in \mathcal{H};\ h(x) = h(y)\}|}{|\mathcal{H}|} = \frac{p^r}{p^{r+1}} = \frac{1}{p} = \frac{1}{m}$$

4. Vorrangwarteschlangen (priority queues)

Definition 175

Eine Vorrangwarteschlange (priority queue) ist eine Datenstruktur, die die folgenden Operationen effizient unterstützt:

1. Insert
2. ExtractMin Extrahieren und Löschen des Elements mit dem kleinsten Schlüssel
3. DecreaseKey Verkleinern eines Schlüssels
4. Union (meld) Vereinigung zweier (disjunkter) Priority Queues

Wir besprechen die Implementierung einer Vorrangwarteschlange als Binomial Heap. Diese Implementierung ist (relativ) einfach, aber asymptotisch bei weitem nicht so gut wie modernere Datenstrukturen für Priority Queues, z.B. Fibonacci Heaps (die in weiterführenden Vorlesungen besprochen werden).

Hier ist ein kurzer Vergleich der Zeitkomplexitäten:

	BinHeap	FibHeap
Insert	$O(\log n)$	$O(1)$
ExtractMin	$O(\log n)$	$O(\log n)$
DecreaseKey	$O(\log n)$	$O(1)$
Union	$O(\log n)$	$O(1)$
	worst case	amortisiert