Seminar Randomisierte Algorithmen WS 2002/03

Thema 5: Datenstrukturen

Frank Meyer Christoph Miebach Christian Westheide

28. April 2003

Inhaltsverzeichnis

1 Das Verwalten von Datenmengen

Die Herausforderung beim Verwalten von Datenmengen besteht darin, die Datensätze sortiert vorzuhalten und somit effiziente Operationen auf diesen Datensätzen zu ermöglichen. Als grundsätzliche Operationen stellt man sich zunächst das Einfügen und das Löschen von Datensätzen vor, sowie das Zugreifen auf die sortiert vorliegenden Datensätze. Zudem wünscht man sich ggf. Operationen auf ganzen Mengen von Datensätzen, wie z.B. Durchschnitt, Vereinigung usw.
Formal können wir die Voraussetzungen für unsere Aufgabe so beschreiben: Es liegen Datensätze i vor, die durch Schlüsselwerte $k(i)$ identifiziert werden können. Im Folgenden werden wir keine Unterscheidung zwischen Schlüsselwert und Datensatz machen. Eine Suche nach einem Datensatz ist also gleich der Suche nach seinem zugehörigen Schlüssel. Unsere Datenstruktur soll die folgenden Anforderungen erfüllen:

- MAKESET(S): initialisiert eine neue leere Menge S
- INSERT(i, S): fügt ein Element i in die Menge S ein
- DELETE(k, S): löscht das Element i mit $k = k(i)$ aus S
- FIND(k, S): sucht das Element mit dem Schlüssel k in S und liefert es zurück

1.1 binäre Suchbäume als Datenstruktur

Als sehr grundlegende Datenstruktur sind uns Binärbäume bekannt. Sei zeichnen sich durch folgendes Kriterium aus: Sei v ein Knoten in einem Binärbaum T. Dann gilt für alle Knoten u des linken Teilbaums von v, dass $k(u) < k(v)$, und für alle Knoten w des rechten Teilbaums von v, dass $k(w) > k(v)$. Betrachtet man Laufzeiten für das Einfügen oder das Suchen von Elementen, so ist klar, dass diese Operationen in Laufzeit $O(Hoehe(T))$ möglich sind. Im worstcase bedeutet dies, dass durch das sukzessive Einfügen von Elementen aus einer bereits sortierten Liste heraus der Binärbaum zu einer linearen Liste entartet. Damit ist die Laufzeit für Suchen etc. linear in der Anzahl der Anzahl der Datensätze.

Wünschenswert ist es nun, diese Fälle auszuschliessen und die Baumhöhe auf ein Minimum zu beschränken. Hierbei möchten wir zwei Varianten von binären Suchbäumen erwähnen.
Die erste Variante sind AVL-Bäume. Diese Bäume zeichnen sich dadurch aus, dass die Buch über die Balanceinformationen in jedem Knoten führen. Somit kann bei zu einseitigem Einfügen und damit Verletzen der Balancekriterien durch das Ausführen von Rotationen die Baumhöhe wirksam beschränkt werden. Ein AVL-Baum ist stets in Balance, weshalb man für diese Datenstruktur eine maximale Laufzeit für Suchen etc. in $O(\log n)$ garantieren kann.
Die zweite Variante sind Bäume, die mit der sogenannten *splaying*-Technik arbeiten. Sie basieren auf der Idee, dass ein Knoten, der mittels FIND gesucht

wurde, durch Rotationen nach oben bis zur Wurzel rotiert wird. Als Konsequenz verbleiben somit oft gesuchte Knoten in der Nähe der Wurzel, während selten besuchte Knoten sich im unteren Teil des Baumes ansammeln. Splaying trees garantieren eine *amortisierte* Laufzeit von $O(\log n)$ pro Operation. Das lässt sich anschaulich dadurch erklären, dass häufig gesuchte Knoten in der Nähe der Wurzel angesammelt werden und somit geringe Suchkosten verursachen. Selten besuchte Knoten hingegen leisten sowieso keinen großen Beitrag zu den Suchkosten, weshalb die hier als schlecht zu erwartende Laufzeit nicht so gravierend ist.
Die Vorteile der splaying trees liegen in der Einfachheit der Implementation. Sie benötigen keinerlei Balanceinformationen und garantieren trotzdem gleiche Laufzeiten (amortisiert) wie AVL-Bäume. Dies Entpuppt sich jedoch als Nachteil, wenn man für *jede* Operation die Kosten im Griff haben muss, und dort bei splaying trees im worst-case lineare Laufzeit angenommen werden muss. Ein weiterer Nachteil ist die ständige Rekonstruktion des Baumes, die nach jeder Suchoperation notwendig ist. Dies kann sich in der Praxis als unperformant erweisen, wenn Caching- oder Paging-Strategien eingesetzt werden. Zudem ist bei höherdimensionalen Datenstrukturen, bei denen sich Teile der Informationen aus den Daten in den Teilbäumen errechnen, zu Leistungseinbußen kommen.

Dies war eine kurze Diskussion von Binärbäumen zur Speicherung von Datensätzen. Es folgt nun eine Betrachtung einer Variante von Binärbäumen, der Treaps.

2 Treaps

Es seien wie zuvor unsere Datensätze i mit Schlüsselwerten $k(i)$ gegeben. Dazu habe jetzt jeder Datensatz zusätzlich noch eine Priority $p(i)$. Treaps sind binäre Suchbäume bezüglich den Schlüsseln $k(i)$ und Heaps bezüglich der Priorities $p(i)$. Wir gehen hier von Maximums-Heaps aus, d.h. auf einem Pfad von der Wurzel zu einem Blatt tragen die Knoten monoton fallende Priority-Werte.
Um die notwendigen Binärbaum-Operationen nun auf Treaps anzuwenden bedarf es kleinen Anpassungen. Das Einfügen von Datensätzen geschieht zunächst wie im normalen Binärbaum. Der einzufügende Schlüsselwert wird im Baum gesucht und an die Stelle, wo die FIND-Operation terminiert eingefügt. Wenn jetzt das Heap-Kriterium verletzt sein sollte, so muss der eingefügte Knoten durch Rotationen weiter nach oben bewegt werden.
Den umgekehrten Weg führt man bei der Löschoperation aus. In Binärbäumen kann man am unteren Ende des Baumes trivial löschen, indem das zu entfernende Blatt einfach abgeschnitten wird. Somit kann man in Treaps einfach Knoten löschen, indem zunächst den fraglichen Datensatz löscht, dann diesen durch Rotationen ans untere Ende des Baumes bewegt und dann einfach entfernt. Die Suchoperation FIND lässt sich ohne Änderungen übernehmen.

Der Zeitaufwand dieser Operationen auf einem Treap T liegt in $O(Hoehe(T))$.

Aber wie hoch kann so ein Treap werden? Dazu erst folgender Satz, der eine Aussage über die Struktur von Treaps macht.

Satz 1 *Sei $S = \{(k_1, p_1), \ldots, (k_n, p_n)\}$ eine beliebige Menge von Schlüssel-Priority-Paaren, so dass alle Schlüssel und alle Priorities paarweise verschieden sind.*
Dann existiert genau ein Treap $T(S)$ für die Menge S.

Beweis: Für den leeren Treap ($n = 0$) und den Treap mit einem Knoten ($n = 1$) ist die Aussage trivial wahr. Sei nun $n \geq 2$: Dann existiert ein Element i mit der größten Priorität: $\forall k \in S : p(i) > p(k)$. Zeichne diesen Knoten als Wurzel des Treaps aus. Alle Knoten l mit Schlüsseln $k(l) < k(i)$ gehören in den linken Teilbaum, alle Knoten r mit Schlüsseln $k(r) > k(i)$ gehören in den rechten Teilbaum. Das Verfahren wird nun für beide Teilbäume wiederholt und so oft ausgeführt, bis alle Datensätze einsortiert sind. □

Damit ist gezeigt: Das Aussehen des Treaps und damit die Höhe ist durch die Priorities bestimmt. Jede gewünschte Form des Treaps kann durch geeignete Wahl der Priorities erreicht werden. Aber was ist eine gute Wahl? Wir werden die Strategie untersuchen, Priorities randomisiert zu wählen.

2.1 Random Treaps

Wir kreieren einen Random Treap, indem wir für jeden Datensatz eine Priority randomisiert wählen. Dabei soll Gleichverteilung auf einem beliebigen beschränkten Intervall angenommen werden, wobei kein Wert mehr als einmal gezogen werden darf. Zusätzlich sollen die Priorities unabhängig von den Schlüsselwerten der Datensätze gezogen werden, genauer, dass die $p(i)$ und die $k(i)$ *unkorreliert* sein sollen. Die Priorities werden den Datensätzen vor dem Einfügen in den Treap zugewiesen und bleiben bis zur Löschung fixiert. Sollte ein gelöschtes Element erneut in den Treap eingefügt werden, so erhält es eine neue Priority.
Zentrale Frage ist nun, welche Kosten ein solcher Treap verursacht. Wenn wir also Laufzeiten für FIND, INSERT und DELETE analysieren wollen, so müssen wir wissen, wie groß der Treap werden kann. Wie bereits erläutert, ist der Treap immer noch ein Binärbaum, also sind die oben genannten Operationen linear in der Höhe des Baumes.
Wir werden uns dieser Frage nicht direkt zuwenden. Stattdessen analysieren wir zunächst ein paar Urnenspiele, um ein paar Hilfsmittel für unsere Argumentation zu sammeln.

2.1.1 Mulmuley Games

Mulmuley Games sind Betrachtungen diverser Urnenexperimenten, die uns eine nützliche Abstraktion der Wahl von Priorities in unserem betrachteten Treap sein können. Wir werden hier nur auf die Analyse von zwei Spielen eingehen, Game A und Game D, da diese uns zur Argumentation genügen.
Die Urne kann mit folgenden Elementen gefüllt werden:

- eine Menge $\mathcal{P} = \{P_1, \ldots, P_p\}$ mit *players*;
- eine Menge $\mathcal{T} = \{T_1, \ldots, T_t\}$ mit *triggers*;
- eine Menge $\mathcal{B} = \{B_1, \ldots, B_b\}$ mit *bystanders*;

Die Elemente sind wohlunterscheidbar. Zudem sind die Elemente aus P geordnet. Man kann sich vorstellen, dass die *bystanders* bloßes Füllmaterial darstellen, die *triggers* haben eine auslösende Funktion und beim Ziehen eines *players* passiert etwas.
Bevor wir uns dem ersten Spiel zuwenden, beweisen wir zunächst folgenden Hilfssatz über Harmonische Zahlen: Sei $H_k := \sum_{i=1}^{k} \frac{1}{i}$ die k-te Harmonische Zahl, dann gilt:

$$\sum_{k=1}^{n} H_k = (n+1)H_{n+1} - (n+1).$$

Beweis: Wir werden die Aussage mit Induktion über n beweisen. Im Induktionsanfang gilt $n = 1$ und somit

$$\sum_{k=1}^{1} H_k = H_1 = 1 = 2 * (1 + \frac{1}{2}) - 2 = 2 * H_2 - 2$$

Im Induktionsschluss von $n \rightsquigarrow n+1$ gilt:

$$\begin{aligned} \sum_{k=1}^{n+1} H_k &= (n+2)H_{n+2} - (n+2) \\ &= (n+2)(H_{n+2} - \frac{1}{n+2}) - (n+1) \\ &= (n+2)H_{n+2} - (n+2) \end{aligned}$$

□

Game A. Bei diesem Spiel ist die Urne mit Elementen aus $\mathcal{P} \cup \mathcal{B}$ gefüllt. Es werden einzelne Elemente ohne Zurücklegen gezogen. Eine Zufallsvariable V bestimmt die Anzahl der Züge, in denen ein P_i gezogen wird, das größer als alle zuvor gezogenen P_k ist. Das Ergebnis des Spiels bezeichnen wir mit $A_p := \mathbf{E}[V]$.

Satz 2 $\forall p \geq 0, A_p = H_p$.

Beweis: o.B.d.A. seien die P_i wiefolgt geordnet: $P_1 > P_2 > \ldots > P_p$. Bemerke, dass die B_i nicht zum Ergebnis beitragen und man daher $b = 0$ annehmen kann. Sei der erste gezogene Player gleich P_i. In diesem Fall gilt $\mathbf{E}[V] = 1 + A_{i-1}$, da nun alle Player $P_{i+1}, \ldots, P_p$ nicht mehr in die Zählung von V mit eingehen. P_i wird gleichverteilt unter allen Playern $P_1, \ldots, P_p$ bestimmt. Daher gilt:

$$A_p = \sum_{i=1}^{p} \frac{1 + A_{i-1}}{p} = 1 + \sum_{i=1}^{p} \frac{A_{i-1}}{p}$$

Ferner gilt: $A_0 = 0$, da kein Player zum Zählen da ist. Mit Indexverschiebung zu $A_p = 1 + \sum_{i=1}^{p-1} \frac{A_i}{p}$ folgt

$$\sum_{i=1}^{p-1} A_i = pA_p - p$$

und mit dem Hilfssatz über Harmonische Zahlen $\sum_{k=1}^{n} H_k = (n+1)H_{n+1} - (n+1)$ ist erkennbar, dass $A_p = H_p$ Lösung dieser Gleichung ist. □

Game D. Bei diesem Spiel ist die Urne mit Elementen aus $\mathcal{P} \cup \mathcal{B} \cup \mathcal{T}$ gefüllt. Wieder werden Elemente ohne Zurücklegen gezogen. Unterschied zu Game A ist, dass hier erst gezählt wird, nachdem ein Trigger gezogen wurde. Die Zufallsvariable V bezeichnet somit die Anzahl der Züge nach dem ersten T_i, in denen ein P_i gezogen wurde, der größer als alle anderen P_k ist. $D_p^t := \mathbf{E}[V]$. Es gilt: Wenn $p, t \geq 0 : D_p^t = H_p + H_t - H_{p+t}$. (Ohne Beweis)

2.1.2 Laufzeitanalyse von FIND

Wir wissen, dass die Priorities gleichverteilt und unabhängig von einander gezogen werden. Wir stellen uns vor, die Priorities werden bereits vor dem Einfüügen der Datensätze gewählt. Damit ist ein sortiertes Einfügen gemäß der Heap-Eigenschaft möglich, wodurch sämtliche Rotationen vermieden werden. Dies ist eine Idee, die unseren Beweis für die Laufzeitanalyse der Suche im Treap tragen wird. Wir werden über die Tiefe von einzelnen Blättern argumentieren, indem wir deren Erwartungswert berechnen. Somit stützt das Ausschliessen von Rotationen nach dem Einfügen unsere These und wir können sicher sein, dass die FIND-Operation einen Knoten dort findet, wo er auch eingefügt wurde. Wir formulieren folgenden Satz:

Satz 3 *Sei T ein Random Treap für die Menge S der Größe n. Sei $x \in S$ das k-größte Element in S. Dann gilt:*

$$\mathbf{E}[depth(x)] = H_k + H_{n-k+1} - 1$$

Beweis: Wir definieren die Mengen $S^- = \{y \in S \mid y \leq x\}$ und $S^+ = \{y \in S \mid y \geq x\}$. Da x das k-größte Element in S ist, folgt $|S^-| = k$ und $|S^+| = n - k + 1$.
Sei $Q_x \subseteq S$ die Menge der Knoten auf dem Pfad von der Wurzel des Treaps bis zu x einschließlich, also die Vorgänger von x. Definiere die Mengen $Q_x^- := Q_x \cap S^-$ und $Q_x^+ := Q_x \cap S^+$. Es gilt folgender Hilfssatz:

> $y \in Q_x^- \iff y$ war zum Zeitpunkt des Einfügens das größte Element aus S^- im Treap.

"⇒": Aus $y \in Q_x^-$ wissen wir, dass y Vorgänger von x ist, also dass $p(x) < p(y)$. Ausserdem gilt: $y < x \Rightarrow x$ liegt im rechten Teilbaum von y. Dies gilt jedoch für alle Elemente z mit $y < z < x$: Betrachten wir die Suche im Treap nach x

und nach y. Beide Operationen besuchen den Pfad von der Wurzel bis zu y, da x ja im rechten Teilbaum von y liegt. Wenn dem aber so ist, dann kann es kein solches Element z geben, das vor y auf dem Suchpfad liegt. Denn sonst läge y im linken Teilbaum von z und x im rechten Teilbaum. Somit wäre $y \notin Q_x^-$.
Also folgt, dass y Vorgänger jeden Knotens z ist, mit $y < z < x$. Mit unserer Annahme über das Einfügen nach Festlegen der Priorities folgt $p(y) > p(z)$ und damit wurden alle Elemente aus $S^- > y$ nach y eingefügt.

"$\Leftarrow$": y war zum Zeitpunkt des Einfügens das größte Element aus S^- im Treap. Demnach besucht die FIND(x)-Operation den Knoten y, denn die einzige Chance, y nicht zu besuchen, wäre vor y auf einen Knoten z zu treffen mit $y < z < x$ (Stichwort: Dort gabelt sich der Weg im Baum).
Dieses $z \in S^-$ wäre somit vor y eingefügt worden, was nach Voraussetzung nicht möglich ist. Es folgt $y \in Q_x^-$.
Um jetzt den Bezug zu Game A herzustellen, bemerken wir folgendes: Der Knoten y trägt zu Q_x^- genau dann bei, wenn er beim Zeitpunkt des Einfügens einen größeren Wert hatte, als alle vorher aus S^- eingefügten Datensätze. Somit entspricht die Verteilung von $|Q_x^-|$ der von Game A mit $\mathcal{P} = S^- \quad$ und $\quad B = S \backslash S^-$.
Aus $|S^-| = k$ folgt $\mathbf{E}[|Q_x^-|] = H_k$.
Der Pfad von der Wurzel zu x besteht ja auch noch aus Elementen $y \in Q_x^+$. Die Argumentation über die erwartete Anzahl solcher Elemente ist komplett symmetrisch, weshalb aus $|S^+|$ folgt, dass $\mathbf{E}[|Q_x^+|] = H_{n-k+1}$. Das Element x wurde in beiden Mengen mitgezählt, also muss es in der Gesamtbetrachtung noch einmal abgezogen werden.

$$\Rightarrow \mathbf{E}[depth(x)] = H_k + H_{n-k+1} - 1$$

□

Unter Ausnutzung von $H_k = \ln k + O(1)$ ergeben sich Kosten von $O(\log n)$ für jede Operation auf dem Treap bei einer Knotenzahl von n.

2.1.3 Zur Anzahl der erwarteten nötigen Rotationen

Wir werden unser oben gezeigtes Ergebnis über die Kostenanalyse noch verfeinern können. Es ist klar, dass die Anzahl der Rotationen bei Einfüge- oder Löschoperationen höchstens gleich der Länge eines Pfades von der Wurzel zu einem Blatt sein kann, also ebenfalls logarithmisch in der Anzahl der Knoten. Jedoch lassen sich mit dem folgenden Satz die benötigten Rotationen besser nach oben abschätzen:

Satz 4 *Sei T ein beliebiger Treap. Die Anzahl der Rotationen in T während DELETE(v, T) bzw. INSERT(v, T) ist gleich der Länge der left spine des rechten Teilbaums von v und der right spine des linken Teilbaums von v.*

Dazu machen wir folgende Definition:

Definition 1 *Die left spine eines Baums T enthält alle Knoten, die man erhält wenn man von der Wurzel von T aus startet und soweit möglich immer wieder den linken Sohn besucht. Die right spine definiert sich analog.*

Um nun die Anzahl der erwarteten Rotationen während DELETE und INSERT nach oben beschränken zu können, machen wir folgende Aussage: Sei mit L_x die Länge der left spine des rechten Teilbaums von x bezeichnet. Analog dazu Sei R_x die Länge der right spine des linken Teilbaums von x. Dann gilt folgender Satz:

Satz 5 *Sei x das k-größte Element in S und T ein Random Treap für S der Größe n.*

$$\begin{aligned} \mathbf{E}[R_x] &= 1 - \frac{1}{k} \\ \mathbf{E}[L_x] &= \frac{1}{n-k+1} \end{aligned}$$

Beweis: Wir werden nur die erste Aussage beweisen, die zweite folgt aus Symmetriegründen. Invertiert man die Ordnung auf den Schlüsseln, so vertauschen sich nämlich R_x und L_x, wobei x dann das $n-k+1$-größte Element in S ist. Um die erste Aussage zu beweisen, zeigen wir, dass die Verteilung der Zufallsvariable R_x gleich der Zufallsvariable V aus Game D ist. Mit $\mathcal{T} = \{x\}, \mathcal{P} = S^-\backslash\{x\}$ und $\mathcal{B} = S^+\backslash\{x\}$, wobei S^-, S^+ wie zuvor, gilt wegen $|P| = k-1, |T| = 1$:

$$\mathbf{E}[R_x] = D^1_{k-1} = H_{k-1} + H_1 - H_k = 1 - \frac{1}{k}$$

Doch wie zeigen wir die Verbindung zu Game D? Wir werden folgende Behauptung beweisen, die aussagt, dass ein Element genau dann auf der right spine des linken Teilbaums von x liegt, wenn die Bedingungen gelten, unter denen ein Element in Game D zum Zählen der Zufallsvariable beiträgt.

> Beh.: Ein Element $z < x$ liegt genau dann auf der right spine des linken Teilbaums von x, wenn z nach x eingefügt wurde und $\forall y \in S^-, z < y < x : y$ wurde nach z eingefügt.

Beweis: "$\Rightarrow$" z liegt im linken Teilbaum von x, also ist $z < x$ und z wurde nach x eingefügt. Wenn z auf der right spine des linken Teilbaums von x liegt, dann muss es vor allen Elementen y mit $z < y < x$ eingefügt worden sein. Ansonsten würde z links von einem solchen y liegen und somit nicht auf der right spine. Also wurden alle Elemente y mit $z < y < x$ nach z eingefügt.
$''\Leftarrow$" Betrachte den Pfad von FIND(z, S). Der Pfad besucht x, denn die einzige Möglichkeit, einen Weg an x vorbei zu nehmen, sind Elemente y mit $z < y < x$. Diese Elemente sind gemäß Voraussetzung beim Einfügen von z nicht vorhanden. Weil $z < x$ liegt es auch im linken Teilbaum von x. Da z zum Zeitpunkt des Einfügens auch das größte Element aus S^- ist, liegt es im linken Teilbaum auf jeden Fall auf der right spine. □

3 Skip Lists

Im Folgenden verwende ich die Notation dieser Definitionen:

Definition 2 *Ein Leveling mit r Leveln von einer geordneten Menge S ist eine Sequenz von ineinander verschachtelten Teilmengen (den Leveln)*

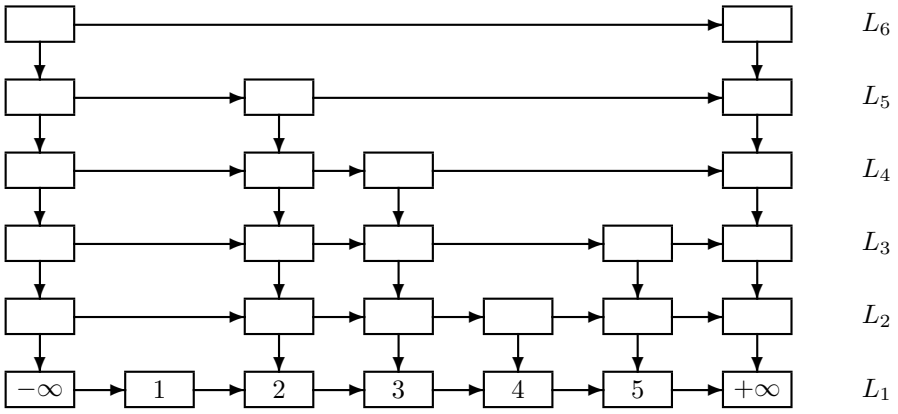
$$L_r \subseteq L_{r-1} \subseteq ... \subseteq L_2 \subseteq L_1$$

so, dass $L_r = \emptyset$ *und* $L_1 = S$.

Definition 3 *Gegeben sei eine geordnete Menge S und ein Leveling für S, dann ist das Level eines Elementes* $x \in S$ *definiert als*

$$l(x) = \max\{i | x \in L_i\}.$$

Dabei stammen die Elemente von S aus einem Universum mit totaler Ordnung. Für jedes gegebene Leveling für die Menge S lässt sich eine Skip List konstruieren. Dazu fügt man alle Elemente aus S auf der untersten Ebene (Level 1) ein und behält sie soviele Ebenen bei, wie es ihrem Level entspricht. Ausserdem fügen wir noch die Elemente $-\infty$ und $+\infty$ hinzu, denen wir Level r zuweisen und die damit also bis nach ganz oben bleiben.



Eine Skip List

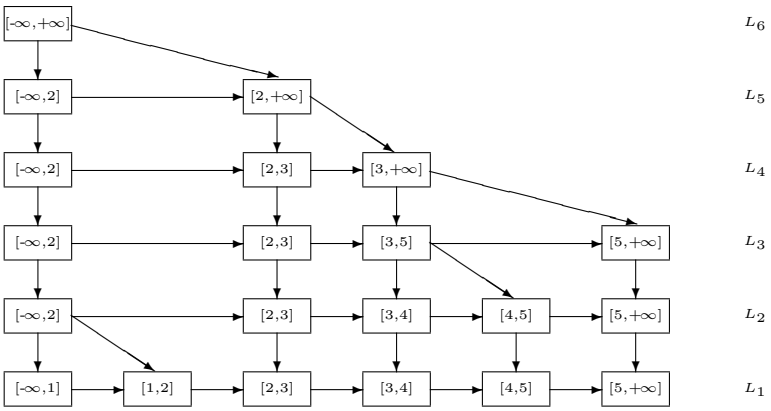
Dies ist die Skip List für $S = \{1, 2, 3, 4, 5\}$ und das Leveling bestehend aus $L_6 = \emptyset$, $L_5 = \{2\}$, $L_4 = \{2, 3\}$, $L_3 = \{2, 3, 5\}$, $L_2 = \{2, 3, 4, 5\}$ und $L_1 = \{1, 2, 3, 4, 5\}$.

Definition 4 *Ein Intervall auf Level i ist die Menge von Elementen von S, die von einem einzelnen horizontalen Pfeil in Level i überspannt wird.*

Die Einteilung in Level kann man auch als immer gröber werdende Partitionierung von S in eine Sammlung von Intervallen betrachten. Unser Beispiel würde dann so aussehen:

$$
\begin{aligned}
L_1 &= [-\infty, 1] \cup [1, 2] \cup [2, 3] \cup [3, 4] \cup [4, 5] \cup [5, +\infty] \\
L_2 &= [-\infty, 2] \cup [2, 3] \cup [3, 4] \cup [4, 5] \cup [5, +\infty] \\
L_3 &= [-\infty, 2] \cup [2, 3] \cup [3, 5] \cup [5, +\infty] \\
L_4 &= [-\infty, 2] \cup [2, 3] \cup [3, +\infty] \\
L_5 &= [-\infty, 2] \cup [2, +\infty] \\
L_6 &= [-\infty, +\infty]
\end{aligned}
$$

Weil wir die Darstellung der Skip List als Baum später noch verwenden, gebe ich sie für unser Beispiel auch explizit an:



Die Baum-Darstellung einer Skip List

3.1 Random Skip Lists

Ein zufälliges Leveling für eine Skip List kann wie folgt festgelegt werden:
Auf Level 1 sind alle Elemente aus S vorhanden.
Jedes Element wird mit Wahrscheinlichkeit $p = \frac{1}{2}$ in das nächsthöhere Level übernommen. Sobald auf einem Level keine Elemente mehr da sind, ist natürlich Schluss.
Anders ausgedrückt: Die l_i sind unabhängige, geometrisch verteilte Zufallsvariablen (mit $p = \frac{1}{2}$).

Für solche Skip-Lists gilt dann das folgende

Lemma 1 *Die Anzahl an Leveln r eines zufälligen Levelings einer Menge der Größe n hat Erwartungswert $E[r] = O(\log n)$. Ferner gilt $r = O(\log n)$ mit hoher Wahrscheinlichkeit.*

Beweis Die Anzahl an Leveln ist $r = 1 + \max_{x \in S} l(x)$. Die einzelnen Levels $l(x)$ sind unabhängig geometrisch verteilt mit Parameter $p = 1/2$ und können als Zufallsvariablen $X_1, ..., X_n$ aufgefasst werden. Es gilt $Pr[X_i > t] \leq (1-p)^t$ und daher

$$Pr[\max_i X_i > t] \leq n(1-p)^t = \frac{n}{2^t}$$

(Wir berechnen die Gegenwahrscheinlichkeit zu „jedes $X_i \leq k$“ mit $p = \frac{1}{2}$. Die Abschätzung gilt, weil Wahrscheinlichkeiten stets ≤ 1 sind)

Durch definieren von $t = \alpha \log n$ und $r = 1 + \max_i X_i$ erhalten wir beim Einsetzen:

$$Pr[r > \alpha \log n] \leq \frac{1}{n^{\alpha-1}}$$

für jedes $\alpha > 1$. □

Eine Implementation von FIND geht, von oben an, jede Ebene soweit nach rechts durch, bis der gesuchte Wert errreicht ist. Und von dort dann eine Ebene weiter nach unten.
Die Laufzeit hängt damit also von sowohl von der Anzahl der Leveln r als auch von den jeweils besuchten Intervallen ab. Weil wir noch keine Aussage über die besuchten Intervalle haben brauchen wir noch

Lemma 2 *Die Kosten von FIND liegen in* $O(\log n)$

Beweis Selbst wenn wir alle Intervalle abklappern müssen, aus denen sich das darüberliegende zusammensetzt hat sind das, vom Erwartungswert her, nur 2, also $O(1)$. Dazu zählen wir bei der geometrischen Verteilung einfach waagerecht: Für jedes der Intervalle gilt ja, dass sie auf der unteren Ebene vorhanden waren und das sie auf dem darüberliegenden jetzt nicht mehr da sind. Und dafür hatten wir ja genau eine Wahrscheinlichkeit $p = 1/2$.
Leider sind wir immer noch nicht ganz fertig. Denn die beiden Variablen sind nicht unabhängig.
Aus dem vorigen Lemma wissen wir aber, dass die Wahrscheinlichkeit für $r > \alpha \log n$ kleiner ist als $1/n^{\alpha-1}$. Da wir auf jeder Ebene aber nur n(+2) Intervalle haben (ganz unten sind alle drin; und mehr werden es ja nicht), deshalb hat dieser Fall keine große Auswirkung auf die Laufzeit. □

Theorem 1 *In einer Random Skip List für eine Menge S der Mächtigkeit n lassen sich FIND INSERT und DELETE in erwarteter Zeit* $O(\log n)$ *durchführen.*

Dazu folgende Überlegung:
Für das INSERT muss entweder auf jeder Ebene jeweils ein Intervall aufgespalten werden, oder jeweils eine leere Ebene generiert werden (das Level des neue Elements ist größer als r). DELETE beginnt mit einem FIND und dann umgekehrt mit dem Zusammenfassen der beiden Nachbarn.
Das ist also nur abhängig von der Höhe.

4 Hash Tables

1. Beim *Static Dictionary Problemend* ist eine Menge von Schlüsselwerten S gegeben, und wir müssen sie in eine Datenstruktur packen, die effiziente FIND-Anfragen unterstützt.

2. Beim *Dynamic Dictionary Problem* ist die Menge S nicht von Anfang an gegeben, sondern sie wird durch INSERT und DELETE Befehle, gemischt mit den FIND-Anfragen, konstruiert.

Definition 5 *Eine Hash-Funktion* h : M $\rightarrow$ N *heißt* perfekt *für eine Menge* S $\subseteq$ M *wenn* h *keine Kollisionen verursacht, bei Eingabe von* S.

Wirklich perfekte Hashfunktionen sind eher langweilig, da sie offensichtlich nur solange funktionieren, wie die Anzahl der Daten kleiner ist als die Anzahl der Felder.

4.1 Universale Hash Familien

Definition 6 *Sei M =*{*0,1,...,m-1*} *und N=*{*0,1,...,n-1*}*, mit* $m \geq n$*. Eine Familie* H *von Funktionen von* M *nach* N *heißt* 2-universal *wenn, für alle* $x, y \in M$*, mit* $x \neq y$*, und zufällig (gleichverteilt) gewähltem* h *aus* H *gilt*

$$Pr[h(x) \quad = \quad h(y)] \leq \frac{1}{n}$$

Diese Bedingung erfüllt eine völlig zufällige Funktion auch. Die sind aber in diesem Zusammenhang nicht wirklich gut, weil sie erheblich mehr Platzbedarf haben: Es muss ja für jeden Wert aus dem Schlüsseluniversum abgespeichert werden, wo er zu finden ist, sonst kann man sich das ja gleich sparen. Da aber in vielen Fällen $m \gg n$ lohnt es sich, dafür diese andere Möglichkeit zu implementieren. Bei den 2-universalen Funktionen geht zwar „Zufälligkeit“ verloren, jedoch verhalten sich diese Funktionen immer noch wie paarweise unabhängig zufällig Variblen. Daher kommt auch die 2 in 2-universal.
Die Notation von Definition 6 wird bei den folgenden Sätzen erstmal beibehalten. Für die Bewertung von unseren Hash-Familien brauchen wir noch ein Kostenmaß. Dafür definieren wir uns

$$\delta(x, y, h) \quad := \quad \begin{cases} 1 & \text{für } h(x) = h(y) \text{ und } x \neq y \\ 0 & \text{sonst} \end{cases}$$

Und als Erweiterung für alle $X, Y \subseteq M$:

$$\begin{aligned}
\delta(x,y,H) &:= \sum_{h\in H}\delta(x,y,h)\\
\delta(x,Y,h) &:= \sum_{y\in Y}\delta(x,y,h)\\
\delta(X,Y,h) &:= \sum_{x\in X}\delta(x,Y,h)\\
\delta(x,Y,H) &:= \sum_{y\in Y}\delta(x,y,H)\\
\delta(X,Y,H) &:= \sum_{h\in H}\delta(X,Y,h)
\end{aligned}$$

Für jede 2-universale Familie H und beliebige $x \neq y$ gilt dann $\delta(x,y,H) \leq |H|/n$.
Dass es nicht viel besser geht, zeigt das folgende Theorem. Denn m kann ja beliebig groß werden.

Theorem 2 *Für jede Familie* H *von Funktionen von* M *nach* N*, gibt es* $x, y \in M$*, so dass*

$$\delta(x,y,H) > \frac{|H|}{n} - \frac{|H|}{m}$$

Beweis Wähle eine beliebige Funktion $h \in H$ und definiere für jedes $z \in N$ die Menge von Elementen aus M, die auf z abgebildet wird als

$$A_z := \{x \in M | h(x) = z\}.$$

Die Mengen A_z, für $z \in N$, bilden eine Partitionierung von M. Es gilt

$$\delta(A_w, A_z, h) = \begin{cases} 0 & w \neq z \\ |A_z|(|A_z|-1) & w = z \end{cases}$$

da Kollisionen nur auftreten, wenn beide Elemente in der gleichen Menge A_z liegen.
Die Gesamtzahl der Kollisionen ist minimal, wenn alle A_z gleich groß sind. Wenn wir das einsetzen ergibt sich

$$\begin{aligned}
\delta(M,M,h) &= \sum_{z\in N}|A_z|(|A_z|-1)\\
&\geq n(\frac{m}{n}(\frac{m}{n}-1))\\
&= m^2(\frac{1}{n}-\frac{1}{m})
\end{aligned}$$

Weil das für jede Wahl von $h \in H$ gilt, ergibt sich jetzt

$$\delta(M, M, H) = \sum_{h \in H} \delta(M, M, h) \geq |H| m^2 (\tfrac{1}{n} - \tfrac{1}{m})$$

Nach dem Schubfachschluss gibt es $x, y \in M$ so, dass

$$\begin{aligned} \delta(x, y, H) &\geq \frac{\delta(M, M, H)}{m^2} \\ &= \frac{|H| \delta(M, M, h)}{m^2} \\ &\geq \frac{|H| m^2 (1/n - 1/m)}{m^2} \\ &= |H| (\frac{1}{n} - \frac{1}{m}) \end{aligned}$$

□

4.2 Anwendung auf Dynamic Dictionary

Lemma 3 *Für alle $x \in M$, $S \subseteq M$, und zufälliges $h \in H$ gilt*

$$E[\delta(x, S, h)] \leq \tfrac{|S|}{n}$$

Beweis

$$\begin{aligned} E[\delta(x, S, h)] &= \sum_{h \in H} \frac{\delta(x, S, h)}{|H|} \\ &= \frac{1}{|H|} \sum_{h \in H} \sum_{y \in S} \delta(x, y, h) \\ &= \frac{1}{|H|} \sum_{y \in S} \sum_{h \in H} \delta(x, y, h) \\ &= \frac{1}{|H|} \sum_{y \in S} \delta(x, y, H) \\ &\leq \frac{1}{|H|} \sum_{y \in S} \frac{|H|}{n} \\ &= \frac{|S|}{n} \end{aligned}$$

□

Betrachten wir eine Anfrage-Sequenz $R = R_1 R_2 ... R_r$ von Update- und Suchanfragen und eine, zu Anfang leere Hash-Tabelle. Angenommen R enthält s INSERT-Operationen. Dann sind in der Hash-Tabelle niemals mehr als s Werte gespeichert. Bezeichne ρ die Gesamtkosten, die bei der Bearbeitung dieser Anfragen mit der Hash-Funkfion $h \in H$ und einer einfach verketteten Liste für Kollisionen anfallen.

Theorem 3 *Für jede Sequenz* R *der Länge* r *mit* s *INSERTs und zufällig gewählten* h *aus einer 2-universal Familie* H *gilt*

$$E[\rho(h, R)] \quad \leq \quad r(1 + \frac{s}{n})$$

Die 1 sind die Kosten für die Auswertung der Hash-Funktion der Rest stammt von den Kollisionen. Die werden in dieser Implementation als einfach verkette Liste gespeichert und sind deshalb genauso groß wie die Anzahl der Kollisionen. Falls vorausgesetzt werden kann, dass unsere Tabelle größer ist, als die maximale Anzahl von gespeicherten Werten, folgt daraus, dass die erwarteten Kosten pro Operation höchstens 2 betragen. Mach der Markov-Ungleichung ist dann die Wahrscheinlichkeit für „$Gesamtkosten > 2rt$“höchstens 1/t. Dabei sind keine Einschränkungen bezüglich R gemacht, außer dem begrenzten Platzbedarf.

4.3 Konstruktion von Familien universaler Hashfunktionen

Wir wenden uns nun der Aufgabe zu, eine 2-universale Familie von Hashfunktionen explizit zu konstruieren. Hierzu definieren wir folgende Familie von Hashfunktionen, um anschließend deren 2-Universalität nachzuweisen.

Seien m und n wie zuvor. Man wählt eine Primzahl $p \geq m$. Definiere nun folgende Funktionen:

$$\begin{aligned} g :\ & \mathbb{Z}_p \to N, && g(x) \ = \ x \bmod n \\ f_{a,b} : & \mathbb{Z}_p \to \mathbb{Z}_p, && f_{a,b}(x) \ = \ (ax \ + \ b) \bmod p \\ h_{a,b} : & \mathbb{Z}_p \to N, && h_{a,b} = g(f_{a,b}(x)) \end{aligned}$$

Wir kommen jetzt zur zentralen Aussage dieses Abschnitts.

Satz 6 *Die Familie* $H = \{h_{a,b} \mid a, b \in \mathbb{Z}_p \text{ mit } a \neq 0\}$ *ist eine 2-universale Hash-Familie.*

Wenn diese Familie von Hashfunktionen, die auf $\mathbb{Z}_p$ definiert sind, 2-universal ist, so ist sie dies offensichtlich auch für jede Untermenge von $\mathbb{Z}_p$, insbesondere also

auch für unsere Menge $M = \{0, 1, ..., m-1\} \subseteq \mathbb{Z}_p = \{0, 1, ..., p-1\}$, da $m \leq p$.

Um Satz 6 beweisen zu können, benötigen wir folgendes Lemma.

Lemma 4 *Für alle x, $y \in \mathbb{Z}_p$, $x \neq y$, gilt $\delta(x, y, H) = \delta(\mathbb{Z}_p, \mathbb{Z}_p, g)$.*

Beweis von Lemma 4 Wir werden zeigen, dass die Zahl der Hashfunktionen, die eine Kollision zwischen x und y verursachen, durch die Größe der Restklasse $\mathbb{Z}_p$ mod n bestimmt wird. Angenommen, x und y kollidieren bei einer spezifischen Hashfunktion $h_{a,b}$. Sei $f_{a,b}(x) = r$ und $f_{a,b} = s$. Aus $a \neq 0$ und $x \neq y$ folgt $r \neq s$.

$$a \neq 0 \wedge x \neq y \Rightarrow r \neq s$$
$$\text{Kollision von x und y} \Rightarrow g(r) = g(s) \Leftrightarrow r \equiv s \, mod \, n$$

Bei gegebenen x und y werden für jede Wahl von $r \neq s$ die Werte a und b eindeutig als Lösung des folgenden Gleichungssystems über $\mathbb{Z}_p$ bestimmt:

$$ax + b \equiv r \, (mod \, p)$$
$$ay + b \equiv s \, (mod \, p)$$

Daraus folgt, dass die Anzahl der Hashfunktionen, die eine Kollision von x und y verursachen, die Anzahl der Wahlen für $r \neq s$ ist, so dass $r \equiv s \, (mod \, n)$. Diese ist aber gegeben durch $\delta(\mathbb{Z}_p, \mathbb{Z}_p, g)$.

□

Beweis von Satz 6 Für jedes $z \in N$ sei $A_z = \{x \in \mathbb{Z}_p | g(x) = z\}$. Offenbar ist $|A_z| \leq \lceil p/n \rceil$. Das bedeutet, dass es für jedes $r \in \mathbb{Z}_p$ maximal $\lceil p/n \rceil$ verschiedene $s \in \mathbb{Z}_p$ gibt, so dass $g(r) = g(s)$. Da es p Möglichkeiten gibt, $r \in \mathbb{Z}_p$ zu wählen, folgt

$$\delta(\mathbb{Z}_p, \mathbb{Z}_p, g) \leq p \, (\left\lceil \frac{p}{n} \right\rceil - 1) \leq \frac{p \, (p-1)}{n}$$

Aus Lemma 4 folgt nun für je zwei verschiedene x und y aus $\mathbb{Z}_p$ $\delta(x, y, H) \leq p \, (p-1)/n$. Da die Größe der Familie H genau $p \, (p-1)$ beträgt, ergibt sich das Resultat $\delta(x, y, H) \leq |H|/n$.

□

Ein Ergebnis der Zahlentheorie, Bertrand‘s Postulat, besagt, dass es für jede natürliche Zahl m eine Primzahl zwischen m und $2m$ gibt. Deshalb kann man ein $p = O(m)$ wählen, was zur Folge hat, dass die Zahl der Zufallsbits, die

benötigt werden, um eine Hashfunktion aus H zu erstellen, $2 \log p = O(\log m)$ beträgt. Die Auswahl, Speicherung und Auswertung einer Hashfunktion aus H ist daher erstaunlich einfach und effizient. a und b werden gleichverteilt und unabhängig aus $\mathbb{Z}_p$ gewählt, die Speicherung benötigt nur vier Speicherzellen der Grösse der Zellen zur Speicherung der Elemente aus M, ist also konstant, und die Auswertung lässt sich aufgrund der Einfachheit der Funktion ebenfalls in konstanter Zeit durchführen. Dies sind enorme Vorteile gegenüber einer vollständig zufällig gewählten Hashfunktion.

4.4 Stark universale Hashfamilien

Die Definition der 2-Universalität beschränkt lediglich die Wahrscheinlichkeit, dass zwei unterschiedliche Schlüssel auf den selben Speicherplatz abgebildet werden, was die paarweise Unabhängigkeit, die durch die Konstruktion der im vorherigen Abschnitt konstruierten 2-universalen Hashfunktionen gegeben ist, nicht vollständig umfasst. Diese erfüllen nämlich auch die folgend definierte stärkere Einschränkung.

Definition 7 *Seien $M = \{0, 1, ..., m-1\}$ und $N = \{0, 1, ..., n-1\}$ mit $m \geq n$. Eine Hashfamilie H, bestehend aus Funktionen von M nach N, heisst stark 2-universal, falls für alle $x_1 \neq x_2 \in M$ und $y_1, y_2 \in N$, bei gleichverteilter zufälliger Wahl von h aus H, gilt:*

$$Pr[h(x_1) = y_1 \text{ und } h(x_2) = y_2] = \frac{1}{n^2}$$

Die meisten bekannten Konstruktionen von 2-universalen Hashfamilien erzeugen zugleich auch stark 2-universale Hashfamilien, was dazu geführt hat, dass die beiden Definitionen normalerweise nicht unterschieden werden. Die obige Definition lässt sich auf stark k-universale Hashfamilien verallgemeinern:

Definition 8 *Seien $M = \{0, 1, ..., m-1\}$ und $N = \{0, 1, ..., n-1\}$ mit $m \geq n$. Eine Hashfamilie H, bestehend aus Funktionen von M nach N, heisst stark k-universal, falls für jede Menge S von k unterschiedlichen Elementen aus M und jede Menge T von k Elementen aus N, bei gleichverteilter zufälliger Wahl von h aus H, die Wahrscheinlichkeit dafür, dass jeweils das i-te Element aus S auf das i-te Element aus T abgebildet, $\frac{1}{n^k}$ beträgt.*

Die Definitionen in diesem Abschnitt sind jeweils verwandt mit der Definition der paarweisen Unabhängigkeit, siehe hierzu die Ausarbeitung zum 3. Seminarthema.

5 Hashing mit O(1) Suchzeit

Das im vorherigen Abschnitt beschriebene Hashing-Verfahren hatte zwar für $|S| = O(n)$ eine beschränkte erwartete Suchzeit, jedoch ist sie im Worst-Case unbeschränkt. In diesem Abschnitt wird nun ein Hashing-Verfahren vorgestellt, welches im Worst-Case $O(1)$ Suchzeit benötigt. Dafür müssen wir in Kauf nehmen, dass diese Lösung nur für das statische Dictionary-Problem funktioniert. Das bedeutet, dass die Menge S der Größ s im voraus feststeht und nur die $FIND$-Operation unterstützt werden muss. Offenbar gibt es hierfür die triviale Lösung einer Hash-Tabelle der Größe m - jedoch hat dies den Nachteil von Preprocessing-Kosten der Größenordnung $O(m)$. Wir wollen jedoch ein Hashing-Verfahren, das nur linearen Speicherplatz sowie polynomielle Preprocessing-Kosten benötigt, d.h., dass die Hashtabelle die Größenordnung $n = O(s)$ aufweisen soll.

5.1 Fast perfekte Hashfamilien

Definition 9 *Eine Familie von Hashfunktionen $H = \{h : M \rightarrow N\}$ heißt perfekte Hashfamilie, wenn es für jede Menge $S \subseteq M$ der Grösse $s < m$ eine perfekte Hashfunktion $h \in H$ existiert.*

Offensichtlich existieren perfekte Hashfamilien - z.B. die Familie aller möglichen Funktionen von M nach T. Wenn man eine perfekte Hashfamilie gegeben hat, löst man das statische Dictionary-Problem auf folgende Weise: Man sucht eine für S perfekte Funktion $h \in H$ und speichert jeden Schlüssel $x \in S$ an der Stelle $T[h(x)]$. Die Preprocessing-Kosten hängen von den Kosten zur Identifizierung der perfekten Hashfunktion, die Suchkosten von den Kosten zur Auswertung der Hashfunktion ab. Da die Wahl der Hashfunktion von der Menge S abhängt, muss sie ebenfalls in der Tabelle gespeichert werden. Wir nehmen an, dass hierzu einige Hilfszellen bereitgestellt werden. Wenn die Größe der Hashfamilie r ist, benötigt die Speicherung einer Funktion $h \in H$ $\Omega(log\, r)$ Bits. Da die Suchzeit in $O(1)$ sein soll, ist es notwendig, dass die Hashfunktion in $O(1)$ Zellen der Tabelle T paßt. Da eine Zelle gerade groß genug ist, einen Schlüssel aus M zu speichern, beträgt ihre Größe maximal $log\, m$ Bits Speicherplatz. Daher muss die Größe der Hashfamilie r polynomiell in m beschränkt sein. Ebenfalls ist es notwendig, die in $O(log\, m)$ Bits gespeicherte Hashfunktion für beliebige Schlüssel effizient auswerten zu können.

Kommen wir noch einmal auf die in Abschnitt 4.3 definierte Hashfamilie h zurück. Jede Hashfunktion $h_{a,b}$ wird durch $a, b \in \mathbb{Z}_p$ bestimmt. Nimmt man nun eine Wahl von p an, die nahe genug an m liegt, dann kann die Funktion $h_{a,b}$ in $O(1)$ Zellen gespeichert werden, die Auswertung ist ebenfalls in $O(1)$ möglich. Das Problem ist jedoch, dass diese universale Hashfamilie nicht perfekt ist.

Satz 7 *Man nehme $n = s$ an. Dann hat jede perfekte Hashfamilie Größe $2^{\Omega(s)}$ und es existiert eine perfekte Hashfamilie der Größe $2^{O(s)}$.*

Daraus folgt, dass $s = O(log\, m)$ gelten muss, damit überhaupt eine in m polynomielle perfekte Hashfamilie existiert. Dies ist jedoch für gewöhnliche m in der Praxis nicht der Fall.

Definition 10 *Sei $S \subset M$ und $h : M \to N$. Für jede Tabellenzelle $0 \leq i \leq n-1$ definiere den Bin*

$$B_i(h, S) = \{x \in S | h(x) = i\}.$$

Bezeichne die Größe eines Bin mit $b_i(h, S) = |B_i(h, S)|$.

Eine perfekte Hashfunktion garantiert also, dass die Größe jedes Bins maximal 1 beträgt.

Definition 11 *Eine Hashfunktion heißt b-perfekt für S, falls $b_i(h, S) \leq b$ für alle i. Eine Hashfamilie heißt b-perfekt, falls für jede $S \subseteq M$ eine b-perfekte Hashfamilie $h \in H$ existiert.*

Satz 8 *Für jedes $m \geq n$ existiert eine b-perfekte Hashfamilie, so dass $b = O(log\, n)$ und $|H| \leq m$.*

Basierend auf diesem Satz kommen wir nun zum Double-Hashing. Hierbei werden in dem Fall, dass mehrere Elemente auf dieselbe Tabellenposition abgebildet werden, diese mittels einer neuen Hashfunktion auf eine zweite, mit der Tabellenposition assoziierte Hashtabelle abgebildet.

Für die erste Stufe verwenden wir eine $(log\, m)$-perfekte Hashfunktion h, um S auf die primäre Tabelle T abzubilden. Betrachten wir nun den Bin B_i, der alle Schlüssel enthält, die auf eine bestimmte Zelle $T[i]$ abgebildet wurden. In dieser Zelle speichern wir nun die Beschreibung einer sekundären Hashfunktion h_i, die benutzt wird, um die Elemente aus B_i auf die mit dieser Zelle assoziierte sekundäre Hashtabelle T_i abzubilden. Da $b = O(log\, m)$, ist 2^b polynomiell in m beschränkt. Daher gibt es eine für B_i perfekte Hashfunktion h_i mit Speicherbedarf $O(log\, m)$. Daraus folgt, dass Double-Hashing für beliebiges $m \geq n$ mit $O(1)$ Suchzeit implementiert werden kann.

Ein Problem bei diesem Ansatz ist, dass es $\Omega(s\, log\, m)$ Speicherplatz benötigt, da für jede der $n = O(s)$ Speicherplätze in der primären Hashtabelle eine sekundäre Tabelle der Größe $O(log\, m)$ benötigt wird. Ein größeres Problem ist die

effiziente Konstruktion und Auswertung der benutzten Hashfunktionen. Das Ziel der primären Hashfunktionen sollte es sein, genügend kleine Bins zu erstellen, um einige perfekte Hashfunktionen als sekundäre Hashfunktionen verwenden zu können.

Satz 9 *Gegeben sei eine Tabelle der Größe* $r = \Omega(s^2)$ $R = \{0, ..., r-1\}$ *und* $m \geq s$. *Dann existiert eine perfekte Hashfamilie* $H = \{h : M \to R\}$ *mit* $|H| \leq m$.

Nun können wir die endgültige Lösung beschreiben. Wir benutzen eine primäre Hashtabelle der Größe $n = s$ und eine primäre Hashfunktion, die garantiert, dass die Bins klein sind. Dann verwenden wir die perfekten Hashfunktionen aus Satz 9 als sekundäre Hashfunktionen sowie sekundäre Hashtabellen der Größe $O({b_i}^2)$ und garantieren damit perfektes Hashing auf der sekundären Ebene. Der gesamte benötigte Speicherplatz für Double-Hashing ist somit

$$s + O\Big(\sum_{i=0}^{s-1} {b_i}^2\Big).$$

Dies ist linear, vorrausgesetzt, dass die Summe der Quadrate der Bin-Größen linear in s beschränkt ist. Die Suchzeit für eine Suchanfrage ist offenbar auch $O(1)$.

5.2 Erreichen der Schranke der Zugriffszeit

In diesem Abschnitt wollen wir die für die primäre Tabelle Hashfunktionen finden, die garantieren, dass die Summe der Quadrate der Bin-Größen linear ist und für die sekundären Tabellen perfekte Hashfunktionen, die maximal quadratischen Platz benötigen, finden.

Für die im Folgenden definierten Hashfunktionen nehmen wir der Einfachheit halber, dass $p = m + 1$ eine Primzahl ist.

Definition 12 *Seien* $V \subseteq M$ *mit* $|V| = v$ *und* $R = \{0, ..., r-1\}$ *mit* $r \geq v$. *Definiere für* $1 \leq k \leq p-1$ *die Funktion* $h_k : M \to R$ *als*

$$h_k(x) = (kx \bmod p) \bmod r.$$

Die Bins seien für alle $i \in R$ *bezeichnet als*

$$B_i(k, r, V) = \{x \in V | h_k(x) = i\}$$

und ihre Größen als

$$b_i(k, r, V) = |B_i(k, r, V)|.$$

Wir fügen r als Parameter der Bin-Größen hinzu, da wir hier, im Gegensatz zu Definition 10, wo wir $n = O(s)$ angenommen hatten, nicht davon ausgehen, dass r linear in Bezug zu v ist. Da die Funktionen allein durch den Wert k bestimmt werden und $k \in 1, ..., m$, kann die Funktionsbeschreibung in einer einzelnen Zelle der Tabelle gespeichert werden. Das folgende Lemma faßt die kritische Eigenschaft dieser Hashfunktionen, die ihre Benutzung motiviert, zusammen.

Lemma 5 *Für alle $V \subseteq M$ der Größe v und alle $r \geq v$ gilt*

$$\sum_{k=1}^{p-1} \sum_{i=0}^{r-1} \binom{b_i(k, r, V)}{2} < \frac{(p-1)v^2}{r} = \frac{mv^2}{r}$$

Beweis von Lemma 5 Die linke Seite gibt die Anzahl an Tupeln $(k, \{x, y\})$ an, bei denen h_k eine Kollision von x und y verursacht. Äquivalent dazu ist es die Anzahl an Tupeln, die die folgenden zwei Bedingungen erfüllen:

$$x, y \in V \; mit \; x \neq y \tag{1}$$

$$(kx \; mod \, p) \; mod \, r) = ((ky \; mod \, p) \; mod \, r). \tag{2}$$

Man nehme ein festes Paar $\{x, y\} \subseteq V$ mit $x \neq y$. Der Beitrag dieses Paar zur Gesamtsumme ist die Anzahl an Wahlmöglichkeiten für k, die die zweite Bedingung erfüllen. Es ist also die Anzahl an Möglichkeiten, k so zu wählen, dass

$$k(x - y) \; mod \, p \in \{\pm r, \pm 2r, ..., \pm \lfloor (p-1)/r \rfloor r\}.$$

Da p eine Primzahl ist, gibt es für einen festen Wert von $x - y$ für jedes j genau ein k, dass die Gleichung

$$k(x - y) \; mod \, p = jr$$

erfüllt. Daraus folgt, dass die Anzahl an Werten für k, die eine Kollision von x und y verursachen, durch $2(p-1)/r$ beschränkt ist. Da die Anzahl an Wahlen für $\{x, y\}$ $\binom{v}{2}$ beträgt, erhalten wir

$$\sum_{k=1}^{p-1} \sum_{i=0}^{r-1} \binom{b_i(k, r, V)}{2} \leq \binom{v}{2} \frac{2(p-1)}{r} < \frac{(p-1)v^2}{r}.$$

□

Mit dem Schubladenprinzip ergibt sich das folgende Korollar.

Korollar 1 *Für alle $V \subseteq M$ der Größe v und alle $r \geq v$ existiert ein $k \in \{1, ...m\}$, so dass*

$$\sum_{i=0}^{r-1} \binom{b_i(k, r, V)}{2} < \frac{v^2}{r}.$$

Die primäre Hashfunktion h_k bildet eine Menge $S \subseteq M$ der Größe s auf eine Hashtabelle T der Größe $n = s$ ab. Die Schlüssel in $B_i(k, r, V)$ werden dann in eine sekundäre Hashtabelle der Größe $b_i(k, r, V)^2$ mittels einer perfekten sekundären Hashfunktion h_{k_i} abgebildet.

Satz 10 *Für jede Menge $S \subseteq M$ mit $|S| = s$ und $m \geq s$ existiert eine Hashtable-Repräsentation von S, die $O(s)$ Speicherplatz benötigt und die Suche in $O(1)$ Zeit durchführt.*

Beweis von Satz 10 Zu zeigen ist, dass es Wahlmöglichkeiten für die primäre Hashfunktion h_k und die sekundären Hashfunktionen $h_{k_1}, ..., h_{k_s}$ gibt, die die zuvor genannten Schranken erfüllen.

Betrachten wir zunächst die primäre Hashfunktion h_k. Hier muss die Summe der quadrierten Bingrößen linear in n sein, um sicher zu stellen, dass der für die sekundären Hashtabellen benötigte Platz in $O(s)$ ist. Wenn wir Korollar 1 auf den Fall $V = S$, $R = T$, und damit $v = r = s$ anwenden, erhalten wir als Ergebnis, dass es ein $k \in \{1, ..., m\}$ gibt mit

$$\sum_{i=0}^{s-1} \binom{b_i(k, s, S)}{2} < s$$

oder dass

$$\sum_{i=0}^{s-1} b_i(k, s, S)[b_i(k, s, S) - 1] < 2s.$$

$$Da \bigcup_{i=0}^{s-1} B_i(k, s, S) = S$$

$$und \sum_{i=0}^{s-1} b_i(k, s, S) = s, \; ergibt \; sich$$

$$\sum_{i=0}^{s-1} b_i(k, s, S)^2 < 2s + \sum_{i=0}^{s-1} b_i(k, s, S) = 3s.$$

Betrachten wir nun die sekundäre Hashfunktion h_{k_i} für die Menge $S_i = B_i(k, s, S)$ der Größe s_i. Wenden wir das Korollar auf den Fall mit $V = S_i$ bzw. $v = s_i$ an und benutzen eine sekundäre Hashtabelle der Größe $r = {s_i}^2$, folgt, dass ein $k_i \in \{1, ..., m\}$ existiert, so dass

$$\sum_{j=0}^{{s_i}^2} \binom{b_j(k_i, {s_i}^2, S_i)}{2} < 1.$$

Dabei ist $b_j(k_i, {s_i}^2, S_i)$ die Anzahl der Kollisionen an der j-ten Position in der sekundären Hashtabelle für $T[i]$. Dies ist nur möglich, wenn jeder Summenterm 0 ist, was bedeutet, dass $b_j(k_i, {s_i}^2, S_i) \leq 1$ für alle j. Daher existiert eine perfekte Hashfunktion h_{k_i}.

Insgesamt benötigt man für dieses Verfahren $6s + 1$ Zellen: $s + 1$ Zellen für die primäre Hashtabelle und ihre Hashfunktion, $3s$ Zellen für die sekundären Hashtabellen und $2s$ Zellen, um die Größen der sekundären Hashtabellen und ihre Hashfunktionen zu speichern. Bei der Bearbeitung einer Anfrage werden 5 Zellen verwendet: der Wert von k und eine Zelle der primären Hashtabelle, die Zellen, die die Größe der sekundären Hashtabelle und ihre Hashfunktion beinhalten, sowie die tatsächliche Position in dieser Tabelle. Eine beschränkte Anzahl arithmetischer Operationen genügt, um die Hashfunktionen auszuwerten. Die ganze Datenstruktur kann also in einem Array der Größe $6s + 1$ gespeichert werden, vorausgesetzt, dass $m > 6s + 1$, um sicherzustellen, die Zeiger auf die sekundären Tabellen wie einen gewöhnlichen Schlüssel in einer Zelle speichern zu können.

□

Bislang haben wir mit einer Ausnahme alle Aspekte betrachtet, die den Realismus und die Effizienz betreffen. Satz 10 garantiert lediglich die Existenz geeigneter primärer und sekundärer Hashfunktionen, jedoch wurde noch nicht darauf eingegangen, wie diese Funktionen gefunden werden können. Wenn wir alle möglichen Funktionen ausprobieren müssten, kämen wir auf eine Laufzeit, die zumindest linear in m wäre. Aber m könnte bezüglich s superpolynomial sein, was die Preprocessing-Kosten praktisch unhandbar machen würde.

Korollar 2 *Für alle $V \subseteq M$ der Größe v und alle $r \geq v$ gilt*

$$\sum_{i=0}^{r-1} \binom{b_i(k, r, V)}{2} < 2\frac{v^2}{r}$$

für mindestens die Hälfte der Wahlen für $k \in \{1, ..., m\}$.

Ein Wert für k, der die Ungleichung erfüllt, kann in erwarteter Zeit $O(v)$ gefunden werden, da auch in $O(v)$ verifiziert werden kann, dass die Ungleichung erfüllt ist.

Satz 11 *Die Hashtabellen-Repräsentation aus Satz 5 kann mit erwarteter Preprocessing-Zeit von $O(s^2)$ bei $13s+1$ genutzten Speicherzellen konstruiert werden und dabei die selbe Suchzeit benötigen.*