A Dynamic Lookup Scheme for Bursty Access Patterns

F. Ergun, S. Mittra, S. C. Sahinalp, J. Sharp, R. K. Sinha (2001)

Oliver Liegmann
Matrikelnummer: 1324345

24. Februar 2006

Seminar „Ausgewählte Algorithmen und Datenstrukturen“
Wintersemester 2005/06

Betreuer: Prof. Dr. Thomas Ottmann, Peter Leven
Institut für Informatik
Albert-Ludwigs-Universität, Freiburg i. Br.

Inhaltsverzeichnis

1 Einführung **1**
1.1 Problemstellung . . . 1
1.2 Bursty Access Patterns . . . 1
1.3 Voraussetzungen . . . 1

2 Skiplisten **2**
2.1 Einführung . . . 2
2.2 Konstruktion . . . 2
2.3 Suche nach einem Schlüssel . . . 4
2.4 Einfügen eines Schlüssels . . . 5
2.5 Entfernen eines Schlüssels . . . 6
2.6 Speicherbedarf . . . 6

3 Biased Skip List **7**
3.1 Einführung . . . 7
3.2 Definitionen . . . 7
3.3 Konstruktion . . . 7
3.4 Suche nach einem Schlüssel . . . 9
3.5 Einfügen und Entfernen eines Schlüssels . . . 10
3.6 Speicherbedarf . . . 10
3.7 Anwendungen . . . 10

4 Dynamic Biased Skip List **10**
4.1 Einführung . . . 10
4.2 Default Key . . . 11
4.3 Konstruktion . . . 11
4.4 Suche nach einem Schlüssel . . . 13
4.5 Speicherbedarf . . . 13

5 Anwendungen für DBSL **13**
5.1 Szenarien . . . 13
5.2 Dynamic Biased Skip List für MFA (DBSL-MFA) . . . 14
5.2.1 Suche nach einem Schlüssel . . . 14
5.2.2 Einfügen eines Schlüssels . . . 14
5.2.3 Entfernen eines Schlüssels . . . 15
5.3 Dynamic Biased Skip List für MRA (DBSL-MRA) . . . 15
5.3.1 Lazy Updating . . . 15
5.3.2 Suche nach einem Schlüssel . . . 16
5.3.3 Einfügen eines Schlüssels . . . 16
5.3.4 Entfernen eines Schlüssels . . . 17
5.4 Anmerkungen zur Implementierung von Ergun et al. . . . 17

6 Weitere Ansätze **19**
6.1 Sahni und Kim (2004) . . . 19
6.1.1 Problemstellung . . . 19
6.1.2 Biased Skip Lists with Prefix Trees . . . 19
6.1.3 Collection of Splay Trees . . . 21

7 Zusammenfassung **21**

1 Einführung

1.1 Problemstellung

Für das 1-dimensionale *Longest-Matching-Prefix-Problem (LMP)* existieren bereits eine Reihe von Ansätzen. Die häufigsten basieren auf Tries oder Suchbäumen. Diese Ansätze verfolgen im Allgemeinen das Ziel, die Zugriffszeit für die Suche eines individuellen Präfix zu optimieren. Dies impliziert jedoch die Annahme, dass alle Präfixe mit gleich hoher Wahrscheinlichkeit angefragt werden, was in der Praxis jedoch häufig nicht der Fall ist. Hier treten viel eher eine spezielle Art von dynamischen Anfragemustern, so genannte *Bursty Access Patterns*[1], auf. Ergun et al. [1] präsentieren in ihrer Veröffentlichung *A Dynamic Lookup Scheme for Bursty Access Patterns* einen auf Skiplisten basierenden Ansatz für den 1-dimensionalen Fall des LMP, der diese Anfragemuster berücksichtigt.
Die Hausarbeit wird zunächst Skiplisten einführen (Kap. 2), dann auf die von Ergun et al. [1] vorgeschlagenen Datenstrukturen Biased Skip List (Kap. 3) und Dynamic Biased Skip List (Kap. 4) eingehen. Das Kapitel 5 wird zwei Anwendungsfälle für die Dynamic Biased Skip List behandeln: DBSL-MFA (Kap. 5.2) für relativ statische Umgebungen und DBSL-MRA (Kap. 5.3) für Umgebungen mit Bursty Access Patterns. Auf die Vorstellung der experimentelle Resultate wurde zu Gunsten der Laufzeitbeweise verzichtet. Das Kapitel 6 wird weitere Ansätze für das LMP mit Bursty Access Patterns vorstellen.

1.2 Bursty Access Patterns

Definition 1 (Bursty Access Patterns) Sei die Eingabe Q, eine Sequenz von Adressen für LMP.
Für die Anzahl unterschiedlicher Adressen k in jeder Teilsequenz $q \subseteq Q$ gilt: $k \ll |q|$.

Bemerkung 1 Eine momentan angefragte Adresse d hat bei Bursty Access Patterns eine hohe Wahrscheinlichkeit in den nächsten Schritten wieder angefragt zu werden.

Umgebungen mit Bursty Access Patterns bilden meist Anfragemuster aus, in denen bestimmte Adressbereiche besonders häufig angefragt werden, andere hingegen nur selten. Diese Muster ändern sich sehr oft, so dass häufiges Einfügen und Entfernen von Adressbereichen notwendig ist. Ergun et al. [1] verweisen auf Veröffentlichungen von Lin und McKeown [2] sowie Mittra und Basu [3], in deren Zusammenhang dieses Szenario auftritt.

1.3 Voraussetzungen

Alle in der Hausarbeit verwendeten Listen sind doppelt verkettet. Soweit nicht anders angegeben, handelt es sich bei der Eingabe um eine Liste nicht-überlappender Intervalle von n aufsteigend sortierten Intervallen. Zur Vereinfachung werden die Intervalle in den Grafiken als einzelne Werte dargestellt, die beispielsweise dem Anfangswert der Intervalle entsprechen. Der Übersicht halber wurde zudem in den Analysen auf die Verwendung der oberen Gauß-Klammer verzichtet. Bei $\log n$ handelt es sich somit durchgehend um $\lceil \log_2 n \rceil$. Zur Kennzeichnung von Konstanten in den Laufzeitbeweisen werden c und d verwendet.

Bemerkung 2 Seien k Präfixregeln gegeben, so ergeben sich daraus höchstens $n = 2k - 1$ nicht-überlappende Adressbereiche (Intervalle).

Lampson et al. [4] geben einen Algorithmus an, mit dem eine aus n Einträgen bestehende sortierte Präfixtabelle mit $O(n)$ Laufzeit in Intervalle umgewandelt werden kann. Abbildung 1 zeigt ein Beispiel für eine solche Umwandlung.

[1] Bursty Access Patterns werden in Kap. 1.2 eingeführt.

Präfix	Zieladresse
*	a
0*	b
001*	c
100*	d

⇒

Bereich	Zieladresse
[0000 – 0001]	b
[0010 – 0011]	c
[0100 – 0111]	b
[1000 – 1001]	d
[1010 – 1111]	a

Abbildung 1: Umwandlung von Präfixen in Intervalle

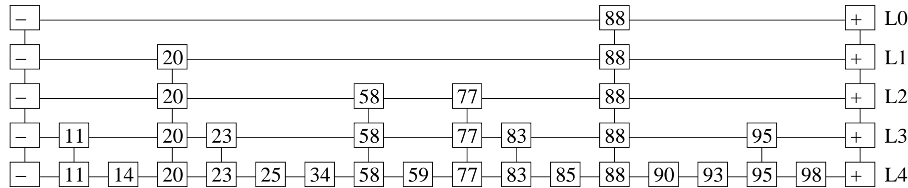


Abbildung 2: Skipliste

2 Skiplisten

2.1 Einführung

Skiplisten wurden 1990 von Pugh [5] entwickelt und gelten laut Ergun et al. [1] im Allgemeinen als die in der Praxis effizienteste Suchdatenstruktur. Es handelt sich hierbei um eine randomisierte Datenstruktur, welche Einfügen, Löschen und Suchen von Schlüsseln in erwarteter Laufzeit von $O(\log n)$ ermöglicht, sowie eine Initialisierung in einer erwarteten Laufzeit von $O(n)$, sofern die Eingabe in sortierter Reihenfolge vorliegt.
Im Gegensatz zu der von Ergun et al. [1] vorgeschlagenen Implementierung, werden in dieser Hausarbeit anstelle von *nil*-Zeigern an Anfang und Ende der Listen $+/-\infty$-Dummyschlüssel verwendet. Dies orientiert sich an der Veröffentlichung von Pugh [5]. Der Grund hierfür liegt darin, dass der Suchalgorithmus in der von Ergun et al.[1] vorgeschlagenen Implementierung andernfalls in einigen Fällen nicht funktioniert. Hierauf wird in Kapitel 2.3 näher eingegangen.

2.2 Konstruktion

Die Eingabe für Skiplisten ist eine doppelt verkette Liste von n Schlüsseln, die bezüglich einer lexikographischen Ordnung, in diesem Fall nach den Endpunkten der Intervalle, aufsteigend sortiert ist. Im ersten Schritt wird am Anfang der Liste ein Schlüssel $-\infty$ eingefügt, sowie am Ende ein Schlüssel $+\infty$ (in den Grafiken jeweils durch $-$ bzw. $+$ gekennzeichnet). Diese Liste bildet das unterste Level $L_{\log n}$ der Datenstruktur (in Abb. 2 durch L_4 gekennzeichnet). Von allen Schlüsseln eines Levels L_i werden mit Wahrscheinlichkeit $p = 0.5$ Kopien in das nächst höhere Level L_{i-1} erstellt und mit den Schlüsseln verlinkt. Alle Schlüssel des neuen Levels werden dann wiederum in einer doppelt verketteten Liste gehalten. Von diesen Schlüsseln werden wiederum mit Wahrscheinlichkeit $p = 0.5$ Kopien in das nächste Level erstellt. Dieser Vorgang wird rekursiv fortgesetzt, bis L_0 erreicht wird und die Höhe der Datenstruktur $\log n + 1$ beträgt.

Satz 1 *Sei die Eingabe eine doppelt verkettete sortierte Liste von n Schlüsseln. Die Initialisierung einer Skipliste kann in erwarteter Laufzeit $O(n)$ durchgeführt werden.*

Beweis Um die Aussage zu beweisen, wird der Beweis in zwei Teile gegliedert. Als erstes wird die Laufzeit zur Konstruktion des untersten Levels $L_{\log n}$ analysiert, dann die Laufzeit für die Konstruktion der restlichen Levels $L_{\log n-1}$ bis L_0.

$$\mathsf{E}[T_{\text{initialise}}] = \mathsf{E}[T_{\text{Konstruktion von } L_{\log n}}] + \mathsf{E}[T_{\text{Konstruktion von } L_{\log n-1},\ldots,L_0}]$$

Zu beachten ist, dass es sich um einen randomisierten Algorithmus handelt, weshalb es sich bei den Laufzeiten um Erwartungswerte handelt.
Wie bereits erwähnt, wird als erstes die Laufzeit für die Konstruktion von $L_{\log n}$ betrachtet. Hier werden jeweils die Schlüssel $+\infty$ und $-\infty$ eingefügt. Die Liste wird hierfür nur einmal traversiert wird, benötigt dies $O(n)$ Zeit.

$$\begin{aligned}\mathsf{E}[T_{\text{Konstruktion von } L_{\log n}}] &= T_{\text{Einfügen von Schlüsseln } +/-\infty}\\ &= O(n)\end{aligned}$$

Die Konstruktion der restlichen Levels lässt sich abschätzen durch die Zeit zur Erstellung eines einzelnen Levels L_i, summiert über alle i.

$$\mathsf{E}[T_{\text{Konstruktion von } L_{\log n-1},\dots,L_0}] = \sum_{i=0}^{\log n-1} \mathsf{E}[T_{\text{Konstruktion von } L_i}]$$

Die Zeit, die für ein festes L_i benötigt wird, lässt sich abschätzen durch die Anzahl der Schlüssel, die vom darunter liegenden Level L_{i+1} hoch kopiert werden, multipliziert mit der Zeit, die zum Erstellen der Kopie eines Schlüssels benötigt wird. Da das Erstellen einer einzelnen Kopie nur eine konstante Zeit benötigt, weil nur Zeiger zu den Nachbarn links und unten erstellt werden müssen, ergibt sich summiert über alle Level, eine Laufzeit von:

$$\begin{aligned}\mathsf{E}[T_{\text{Konstruktion von } L_{\log n-1},\dots,L_0}] &= \sum_{i=0}^{\log n-1} \mathsf{E}[T_{\text{Konstruktion von } L_i}]\\ &= c\cdot \sum_{i=0}^{\log n-1} \mathsf{P}[\text{Schlüssel wird von } L_{i+1} \text{ nach } L_i \text{ kopiert}]\\ &\quad \cdot \mathsf{E}[|\text{Schlüssel in } L_{i+1}|]\\ &= c\cdot \sum_{i=0}^{\log n-1} \frac{1}{2}\cdot \mathsf{E}[|\text{Schlüssel in } L_{i+1}|]\end{aligned}$$

Durch die Wahrscheinlichkeit $p = 0.5$ mit der die Schlüssel kopiert werden, existieren in jedem Level L_i erwartet halb so viele Schlüssel, wie in dem darunter liegenden Level L_{i+1} . Da bekannt ist, dass $L_{\log n}$ aus n Schlüsseln besteht, ergibt sich hieraus:

$$\begin{aligned}\mathsf{E}[T_{\text{Konstruktion von } L_{\log n-1},\dots,L_0}] &= c\cdot \sum_{i=0}^{\log n-1} \frac{1}{2}\cdot \mathsf{E}[|\text{Schlüssel in } L_{i+1}|]\\ &= c\cdot \frac{1}{2}\cdot \sum_{i=0}^{\log n-1} 2^{i+1}\\ &= c\cdot \sum_{i=0}^{\log n-1} 2^{i}\\ &= c\cdot (2^{\log n}-1)\\ &= O(n)\end{aligned}$$

Somit ergibt sich eine erwartete Gesamtlaufzeit für die Konstruktion von $O(n)$.

$$\begin{aligned}\mathsf{E}[T_{\text{initialise}}] &= \mathsf{E}[T_{\text{Konstruktion von } L_{\log n}}] + \mathsf{E}[T_{\text{Konstruktion von } L_{\log n-1},\dots,L_0}]\\ &= 2\cdot O(n)\\ &= O(n)\end{aligned}$$

□

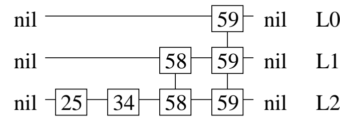


Abbildung 3: Implementierung der Skipliste nach Ergun et al. [1]

2.3 Suche nach einem Schlüssel

Sei der Schlüssel k gesucht. Ausgehend vom Schlüssel links oben, also dem obersten $-\infty$-Schlüssel, werden die Schlüssel des Levels so lange nach rechts traversiert, bis man zu einem Schlüssel $\geq k$ gelangt. Ist der Schlüssel gleich k, so endet die Suche. Andernfalls geht man einen Schritt nach links und folgt dem Zeiger nach unten in das nächste Level. Das Verfahren wird solange rekursiv weitergeführt, bis man den Schlüssel gefunden hat, oder kein Zeiger in das nächsttiefere Level mehr existiert. Sei beispielsweise in Abb. 2 der Schlüssel 23 gesucht. So werden die Schlüssel in folgender Reihenfolge traversiert (Der Index einer Zahl gibt das jeweilige Level an, in dem sich der Schlüssel befindet): $-\infty_0$, 88_0, $-\infty_0$, $-\infty_1$, 20_1, 88_1, 20_1, 20_2, 58_2, 20_2, 20_3, 23_3.
An dieser Stelle spielt die Implementierung eine Rolle. Ergun et al [1] verzichten in ihrer Implementierung auf die $-\infty$- und $+\infty$-Schlüssel. Diese Schlüssel sind jedoch wesentlich bei der Suche, wie man anhand eines Beispiels (Abb. 3) sehen kann. Da man in einem Level, wenn ein größerer Schlüssel gefunden wird, nur einen Schritt nach links geht, würde der Schlüssel 25 nicht gefunden werden und die Suche nach dem Traversieren von 59_0, 59_1, 58_1, 58_2, 34_2 erfolglos bei 34_2 abbrechen.

Satz 2 *Die Suche nach einem Schlüssel in einer Skipliste kann in erwarteter Laufzeit $O(\log n)$ durchgeführt werden.*

Beweis Die Laufzeit für die Suche lässt sich abschätzen durch die Summe der Zeit, die für die vertikalen Schritte benötigt wird und der Zeit, die für die horizontalen Schritte benötigt wird.
Für die vertikalen Schritte ist die Laufzeit jedoch einfach abzuschätzen. Es ist bekannt, dass man höchstens $\log n$ viele Schritte vertikal gehen kann, da die gesamte Höhe der Datenstruktur $\log n + 1$ beträgt und man immer nur nach unten geht, nie nach oben. Jeder vertikale Schritt benötigt nur konstant viel Zeit, da man nur einem Zeiger folgen muss. Somit werden höchstens $O(\log n)$ Schritte vertikal benötigt.

$$\begin{aligned}\mathsf{E}[T_{\text{search}}] &= \mathsf{E}[T_{\text{Vertikale Schritte}}] + \mathsf{E}[T_{\text{Horizontale Schritte}}] \\ &\leq O(\log n) + \mathsf{E}[T_{\text{Horizontale Schritte}}]\end{aligned}$$

Nun muss noch die Zeit für die horizontalen Schritte abgeschätzt werden. Da für jeden Schritt wieder nur konstant viel Zeit benötigt wird, da man bloß Zeigern folgen muss, entspricht dies einer konstanten Zeit mal der Anzahl der tatsächlichen Schritte. Diese Analyse wird getrennt geführt für das oberste Level L_0 und die restlichen Levels.

$$\begin{aligned}\mathsf{E}[T_{\text{Horizontale Schritte}}] &= \mathsf{E}[T_{\text{Horizontale Schritte in } L_0}] + \sum_{i=1}^{\log n - 1} \mathsf{E}[T_{\text{Horizontale Schritte in } L_i}] \\ &= c \cdot \left(\mathsf{E}[|\text{Horiz. Schr. in } L_0|] + \sum_{i=1}^{\log n - 1} \mathsf{E}[|\text{Horiz. Schr. in } L_i|] \right)\end{aligned}$$

Im untersten Level hat man, ohne die beiden Dummyschlüssel, n Schlüssel. Da die Schlüssel mit Wahrscheinlichkeit $p = 0.5$ nach oben kopiert werden, existieren im Level darüber im Erwartungswert halb so viele Schlüssel. Für das oberste Level gibt es dann im Erwartungswert noch einen

Schlüssel. Nach Einfügen der beiden Dummyschlüssel können erwartet höchstens noch 4 Schritte im obersten Level gemacht werden.

$$\begin{aligned}
\mathsf{E}[T_{\text{Horizontale Schritte}}] &= c \cdot \left(\mathsf{E}[|\text{Horiz. Schritte in } L_0|] + \sum_{i=1}^{\log n - 1} \mathsf{E}[|\text{Horiz. Schritte in } L_i|] \right) \\
&\leq c \cdot \left(4 + \sum_{i=1}^{\log n - 1} \mathsf{E}[|\text{Horizontale Schritte in } L_i|] \right)
\end{aligned}$$

Sei u_j die in Level j liegende Kopie des Schlüssels u. Analog für v_j. Betrachte nun für ein festes L_i die Anzahl der Schritte. Angenommen, man ist bei einem Schlüssel u_{i-1} vom darüber gelegenen L_{i-1} nach unten gekommen, dann muss der benachbarte Schlüssel v_{u-1} größer sein, als der gesuchte Schlüssel. Zu beachten ist, dass zwischen u_{u-1} und v_{u-1} kein weiterer Schlüssel liegt. In L_i müssen nun wieder zwei Schritte gemacht werden: Von u_i nach v_i und zurück. Es muss nun jedoch beachtet werden, dass zwischen u_i und v_i Schlüssel liegen können, die nach L_i, jedoch nicht nach L_{i-1} hoch kopiert wurden. Deren Anzahl entspricht jedoch im Erwartungswert 1. Der Grund hierfür ist, dass durch die Wahrscheinlichkeit für eine Kopie von $p = 0.5$ erwartet jeder zweite Schlüssel hoch kopiert wird.

$$\begin{aligned}
\mathsf{E}[T_{\text{Horizontale Schritte}}] &\leq c \cdot \left(4 + \sum_{i=1}^{\log n - 1} \mathsf{E}[|\text{Horizontale Schritte in } L_i|] \right) \\
&\leq c \cdot \left(4 + \sum_{i=1}^{\log n - 1} 2 \cdot (2 + \mathsf{E}[|\text{Neue Schlüssel zwischen 2 Schl. aus } L_{i-1}|]) \right) \\
&= c \cdot \left(4 + \sum_{i=1}^{\log n - 1} 2 \cdot (2 + 1) \right) \\
&= O(\log n)
\end{aligned}$$

Es ergibt sich somit für die Suche eine erwartete Gesamtlaufzeit von $O(\log n)$.

$$\begin{aligned}
\mathsf{E}[T_{\text{search}}] &= \mathsf{E}[T_{\text{Vertikale Schritte}}] + \mathsf{E}[T_{\text{Horizontale Schritte}}] \\
&\leq 2 \cdot O(\log n) \\
&= O(\log n)
\end{aligned}$$

□

2.4 Einfügen eines Schlüssels

Zum Einfügen eines Schlüssels k sucht man als erstes nach der Stelle in $L_{\log n}$, an der dieser eingefügt werden soll. Dort fügt man den Schlüssel ein, indem man wie bei doppelt verketten Listen üblich, die Zeiger neu setzt. Dann wird der Schlüssel solange rekursiv mit einer Wahrscheinlichkeit von $p = 0.5$ nach oben kopiert, bis entweder entschieden wird, dass keine Kopie mehr erstellt wird oder man in L_0 angelangt ist.

Satz 3 *Das Einfügen eines Schlüssel in eine Skipliste kann in erwarteter Laufzeit $O(\log n)$ durchgeführt werden.*

Beweis Die Laufzeit für das Einfügen eines neuen Schlüssel kann abgeschätzt werden durch die Zeit für die Suche nach der Stelle, an der der Schlüssel eingefügt werden muss und die zusätzliche Zeit, die benötigt wird, um den Schlüssel an der gefundenen Stelle einzufügen und seine Kopien zu erstellen.

Es wurde bereits gezeigt, dass die Zeit für die Suche erwartet $O(\log n)$ beträgt. Das Einfügen selbst ist in konstanter Zeit möglich, da nur die Zeiger von und zu den Nachbarn neu gesetzt werden müssen. Also muss nur noch die Zeit abgeschätzt werden, die benötigt wird um die Kopien des Schlüssels in die höheren Levels zu erstellen.

$$\begin{aligned}\mathsf{E}[T_{\text{insert}}] &= \mathsf{E}[T_{\text{search}}] + \mathsf{E}[T_{\text{Einfügen im untersten Level}}] + \mathsf{E}[T_{\text{Erstellung der Kopien}}] \\ &= O(\log n) + c + \mathsf{E}[T_{\text{Erstellung der Kopien}}]\end{aligned}$$

Es ist klar, dass die Anzahl der Kopien beschränkt ist, durch die Anzahl der Levels, also $O(\log n)$. Das Kopieren benötigt konstant viel Zeit pro Level. Somit kommt man auf eine ewartete Gesamtlaufzeit von $O(\log n)$.

$$\begin{aligned}\mathsf{E}[T_{\text{insert}}] &= O(\log n) + c + \mathsf{E}[T_{\text{Erstellung der Kopien}}] \\ &= O(\log n) + c + d \cdot O(\log n) \\ &= O(\log n)\end{aligned}$$

□

2.5 Entfernen eines Schlüssels

Zum Entfernen eines Schlüssels wird dieser zunächst gesucht. Bei der Suche findet man das Vorkommen des Schlüssels, welches sich in der Skipliste am weitesten oben befindet. Nachdem man den Schlüssel gefunden hat, wird der Schlüssel aus der Liste des jeweiligen Levels gelöscht, sowie rekursiv nach unten hin, alle seine Kopien, die durch Folgen des Zeigers nach unten direkt gefunden werden.

Satz 4 *Das Löschen eines Schlüssels aus einer Skipliste kann in erwarteter Laufzeit $O(\log n)$ durchgeführt werden.*

Beweis Die Laufzeit für das Löschen setzt sich zusammen aus der Suche nach dem Schlüssel und dem Löschen des Schlüssels mit seinen Kopien. Die Suche hat, wie bereits gezeigt, eine erwartete Laufzeit von $O(\log n)$. Das Löschen eines einzelnen Schlüssels geht in konstanter Zeit, da nur die Zeiger der Nachbarn neu gesetzt werden müssen und dem Zeiger nach unten gefolgt wird. Da die Datenstruktur durch eine erwartete Höhe von $O(\log n)$ beschränkt ist die erwartete Laufzeit somit $O(\log n)$.

$$\begin{aligned}\mathsf{E}[T_{\text{remove}}] &= \mathsf{E}[T_{\text{search}}] + \mathsf{E}[T_{\text{Löschen des Schlüssels und der Kopien}}] \\ &= 2 \cdot O(\log n) \\ &= O(\log n)\end{aligned}$$

□

2.6 Speicherbedarf

Satz 5 *Eine Skipliste mit n Schlüsseln als Eingabe benötigt einen erwarteten Speicherplatz von $O(n)$.*

Beweis Der Speicherverbrauch ist beschränkt durch die Laufzeit für die Konstruktion. Diese beträgt im Erwartungswert $O(n)$. Da jeder Schlüssel nur konstant viel Platz benötigt (der Schlüssel und vier Zeiger), benötigt die gesamte Datenstruktur erwartet $O(n)$ Platz. □

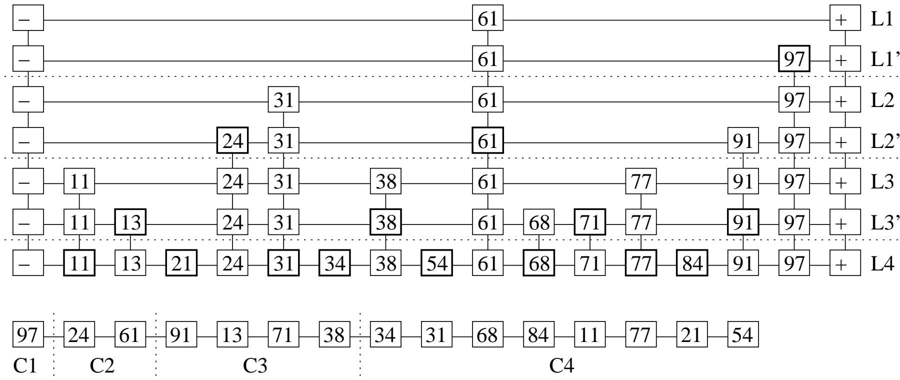


Abbildung 4: Biased Skip List mit Masterliste

3 Biased Skip List

3.1 Einführung

Problematisch bei den normalen Skiplisten ist die Voraussetzung, dass alle Schlüssel mit gleicher Wahrscheinlichkeit angefragt werden. Die Verwendung von normalen Skiplisten bringt also keinen entscheidenden Vorteil gegenüber bestehenden baumbasierten Datenstrukturen für das LMP-Problem. Eine gewünschte Eigenschaft wäre, Schlüssel die häufig angefragt werden, möglich weit oben in der Skipliste anzuordnen, um sie dadurch schneller finden zu können. Dies ist die Motivation für die *Biased Skip List (BSL)*, welche von Ergun et al. [1] vorgestellt wird.

3.2 Definitionen

Definition 2 (Rang) Der Rang $r(k)$ eines Schlüssels k ist eine natürliche Zahl mit der Eigenschaft $r(k) \in [1, \ldots, n]$, wobei keine zwei unterschiedliche Schlüssel den gleichen Rang haben.

Definition 3 (Masterliste) Eine Masterliste ist eine doppelt verkettete Liste aller Schlüssel, sortiert nach aufsteigendem Rang.

Definition 4 (Klasse) Eine Klasse C_i ist eine Teilliste der Masterliste, wobei $i \in \mathbb{N}$ und $1 \leq i \leq \log n$ mit folgenden Eigenschaften:

1. Die Größe der Klassen beträgt $|C_i| = 2^{i-1}$.

2. Für zwei unterschiedliche Klassen C_i, C_{i+1} gilt: $x \in C_i$, $y \in C_{i+1} \Rightarrow r(x) < r(y)$.

3.3 Konstruktion

Die Eingabe ist die aufsteigend sortierte doppelt verkettete Liste von Schlüssel mit den beiden Dummyschlüsseln, wie bei den normalen Skiplisten, sowie die Masterliste, welche die Schlüssel in aufsteigendem Rang enthält.
Die Level sind von unten nach oben wie folgt gekennzeichnet: $L_{\log n}$, $L'_{\log n-1}$, $L_{\log n-1}$,..., L_2, L'_1, L_1.
Das unterste Level $L_{\log n}$ besteht aus allen Schlüsseln der Eingabeliste. Für die Konstruktion der restlichen Levels wird wie folgt vorgegangen: L'_i besteht aus allen Schlüsseln aus den Klassen $C_i, C_{i-1}, \ldots, C_1$ (*automatische Kopien*). Aus den nicht automatisch von L_{i+1} kopierten Schlüsseln werden mit Wahrscheinlichkeit $p = 0.5$ die Schlüssel nach L'_i kopiert (*randomisierte Kopien*).
In das Level L_i kopiert man daraufhin alle Schlüssel aus den Klassen $C_{i-1}, C_{i-2}, \ldots, C_1$ (*automatische Kopien*). Wiederum werden von den nicht automatisch kopierten Schlüsseln mit Wahrscheinlichkeit $p = 0.5$ Schlüssel aus L'_i nach L_i kopiert.

Bemerkung 3 Sei k ein Schlüssel in der Masterliste. Um in konstanter Zeit von k in der Masterliste zu einer Kopie von k in der BSL und umgekehrt zu gelangen, existieren Zeiger zwischen k in der Masterliste und k in der BSL. Diese sind aus Gründen der Übersichtlichkeit nicht in den Abbildungen vorhanden. Ergun et al. [1] geben keine Implementierung für diese Zeiger an, man kann sich jedoch vorstellen, dass die in der Skipliste am weitesten oben liegende Kopie von k einen Zeiger in die Masterliste beinhaltet, sowie ein Zeiger von der Masterliste zurück in die BSL. Es genügt laufzeittechnisch, nur von einer der Kopien von k einen Zeiger zu setzen, da nur eine erwartet konstante Anzahl von Kopien eines Schlüssels existieren, wie in Lemma 1 gezeigt wird.

Satz 6 *Die Initialisierung einer BSL kann in erwarteter Laufzeit $O(n)$ durchgeführt werden.*

Beweis $L_{\log n}$ kann in $O(n)$ erstellt werden, da nur die Dummyschlüssel eingefügt werden müssen. Zu analysieren sind noch die Laufzeit zur Erstellung der randomisierten und der automatischen Kopien.

$$\begin{aligned}\mathsf{E}[T_{\text{initialise}}] &= \mathsf{E}[T_{L_{\log n}}] + \mathsf{E}[T_{\text{Randomisierte Kopien}}] + \mathsf{E}[T_{\text{Automatische Kopien}}] \\ &= O(n) + \mathsf{E}[T_{\text{Randomisierte Kopien}}] + \mathsf{E}[T_{\text{Automatische Kopien}}]\end{aligned}$$

Die Analyse der randomisierten Kopien bei BSL verläuft analog zur Analyse der randomisierten Kopien bei den normalen Skiplisten, da der Algorithmus analog arbeitet mit dem einzigen Unterschied, dass die Datenstruktur jetzt keine Höhe von $\log n + 1$ sondern $2 \cdot \log n - 1$ hat. Somit wird der Beweis analog geführt:

$$\begin{aligned}\mathsf{E}[T_{\text{Randomisierte Kopien}}] &\le \sum_{i=1}^{2\cdot\log n-1} \mathsf{E}[T_{\text{Kopien in Tiefe } i}] \\ &= c \cdot \sum_{i=1}^{2\cdot\log n-1} \mathsf{P}[\text{Schüssel wird von Tiefe } i+1 \text{ nach } i \text{ kopiert}] \\ &\quad \cdot \mathsf{E}[|\text{Schlüssel in Tiefe } i+1|] \\ &= c \cdot \sum_{i=1}^{2\cdot\log n-1} \frac{1}{2} \cdot \mathsf{E}[|\text{Schlüssel in Tiefe } i+1|] \\ &= c \cdot \sum_{i=1}^{2\cdot\log n-1} 2^i \\ &= O(n)\end{aligned}$$

Für automatischen Kopien gilt, dass die Zeit für eine einzelne Kopie konstant ist. Somit muss noch die Gesamtzahl der automatischen Kopien berechnet werden. Diese entspricht der Anzahl von Schlüsseln, die von L_{i+1} nach L'_i kopiert werden plus der Kopien von L'_i nach L_i.

$$\begin{aligned}\mathsf{E}[T_{\text{Automatische Kopien}}] &= c \cdot |\text{Automatische Kopien}| \\ &= c \cdot \sum_{i=1}^{\log n-1} (|\text{Kopie von } L_{i+1} \text{ nach } L'_i| + |\text{Kopie von } L'_i \text{ nach } L_i|)\end{aligned}$$

Die Anzahl der Kopien von L'_i nach L_i sind höchstens so viele wie von L_{i+1} nach L'_i, da nur eine Auswahl von Schlüssel aus L'_i nach L_i weiter kopiert wird. Somit lässt sich diese Anzahl durch das Doppelte der Kopien von L_{i+1} nach L'_i abschätzen.

$$\begin{aligned}\mathsf{E}[T_{\text{Automatische Kopien}}] &= c \cdot \sum_{i=1}^{\log n - 1} (|\text{Kopie von } L_{i+1} \text{ nach } L'_i| + |\text{Kopie von } L'_i \text{ nach } L_i|) \\ &\leq 2 \cdot c \sum_{i=1}^{\log n - 1} |\text{Kopie von } L_{i+1} \text{ nach } L'_i|\end{aligned}$$

Man betrachte ein festes Level L'_i: Nach L'_i werden alle Schlüssel aus den Klassen C_i, $C_{i-1}, \ldots, C_1$ kopiert.

$$\begin{aligned}\mathsf{E}[T_{\text{Automatische Kopien}}] &\leq 2 \cdot c \cdot \sum_{i=1}^{\log n - 1} |\text{Kopie von } L_{i+1} \text{ nach } L'_i| \\ &= 2 \cdot c \cdot \sum_{i=1}^{\log n - 1} \sum_{j=1}^{i} |C_j|\end{aligned}$$

Eine Klasse C_j besteht, nach Definition 4, aus 2^{j-1} Schlüsseln.

$$\begin{aligned}\mathsf{E}[T_{\text{Automatische Kopien}}] &\leq 2 \cdot c \cdot \sum_{i=1}^{\log n - 1} \sum_{j=1}^{i} |C_j| \\ &= 2 \cdot c \cdot \sum_{i=1}^{\log n - 1} \sum_{j=1}^{i} 2^{j-1} \\ &= 2 \cdot c \cdot \sum_{i=1}^{\log n - 1} (2^i - 1) \\ &= 2 \cdot c \cdot \sum_{i=1}^{\log n - 1} O(2^i) \\ &= 2 \cdot c \cdot O(n) \\ &= O(n)\end{aligned}$$

Somit beträgt die Gesamtzeit für die Konstruktion $O(n)$.

$$\begin{aligned}\mathsf{E}[T_{\text{initialise}}] &= \mathsf{E}[T_{L_{\log n}}] + \mathsf{E}[T_{\text{Randomisierte Kopien}}] + \mathsf{E}[T_{\text{Automatische Kopien}}] \\ &\leq 3 \cdot O(n) \\ &= O(n)\end{aligned}$$

□

3.4 Suche nach einem Schlüssel

Der Suchalgorithmus ist der gleiche, wie bei den normalen Skiplisten.

Lemma 1 *Sei k ein Schlüssel in der BSL. Dann existieren nur erwartet konstant viele Kopien von k in der BSL.*

Beweis Um die Anzahl der Kopien zu einem festen Schlüssel k zu berechnen, werden alle in der BSL enthaltenen Elemente (Schlüssel und deren Kopien) durch die Anzahl der Schlüssel in der Eingabe dividiert. Die Anzahl der Elemente ist nach oben hin beschränkt durch die Konstruktionszeit der BSL, welche erwartet $O(n)$ beträgt. Da die Eingabe aus n Schlüsseln besteht, ist der Quotient $\frac{O(n)}{n} = O(1)$. Somit ist die erwartete Anzahl von Kopien von k konstant. □

Satz 7 *Die Suche nach einem Schlüssel in einer BSL kann in erwarteter Laufzeit $O(\log r(k))$ durchgeführt werden. Existiert der Schlüssel nicht, so beträgt die erwartete Suchzeit $O(\log n)$.*

Beweis Nach höchstens $O(\log r(k))$ vertikalen Schritten ist der Schlüssel gefunden, da ein Schlüssel mit Rang $r(k)$ sich durch die automatischen Kopien in der Klasse $C_{\log r(k)}$ befindet.

$$\begin{aligned}\mathsf{E}[T_{\text{search}}] &= \mathsf{E}[|\text{Vertikale Schritte}|] + \mathsf{E}[|\text{Vertikale Schritte}|] \cdot \mathsf{E}[|\text{Horizontale Schritte}|] \\ &\leq O(\log(r(k)) + \mathsf{E}[|\text{Vertikale Schritte}|] \cdot \mathsf{E}[|\text{Horizontale Schritte}|]\end{aligned}$$

Nun muss für diese $O(log(k))$ Levels die Anzahl der Schritte pro Level berechnet werden. Durch die automatischen Kopien, die zu den randomisierten Kopien hinzukommen, ist diese doppelt so groß, wie bei den normalen Skiplisten.

$$\begin{aligned}\mathsf{E}[T_{\text{search}}] &\leq O(\log(r(k)) + \mathsf{E}[|\text{Vertikale Schritte}|] \cdot \mathsf{E}[|\text{Horizontale Schritte}|] \\ &\leq O(\log(r(k)) + O(\log(r(k)) \cdot \left(2 \cdot \mathsf{E}[|\text{Horizontale Schritte}|]^{\textit{Skip List}}\right) \\ &= O(\log(r(k)) + O(\log(r(k)) \cdot (2 \cdot O(1)) \\ &= O(\log(r(k))\end{aligned}$$

Die Suchzeit beträgt somit erwartet $O(\log r(k))$.
Wird der gesuchte Schlüssel nicht gefunden, so endet die Suche im untersten Level. Somit benötigt die Suche in dem Fall höchstens erwartet $O(\log n)$. □

3.5 Einfügen und Entfernen eines Schlüssels

Die Datenstruktur ist nicht als dynamische Datenstruktur ausgelegt, weshalb Einfügen und Entfernen nicht behandelt wird.

3.6 Speicherbedarf

Satz 8 *Eine BSL mit n Schlüsseln als Eingabe benötigt einen erwarteten Speicherplatz von $O(n)$.*

Beweis Der Speicherverbrauch ist beschränkt durch die Laufzeit für die Konstruktion. Diese ist erwartet $O(n)$. Die Masterliste benötigt zusätzlich $O(n)$ Speicherplatz. Somit hat eine BSL mit n Schlüsseln einen erwarteten Speicherbedarf von $O(n)$. □

3.7 Anwendungen

Man kann BSL als Suchdatenstruktur für statische Router verwenden. Wenn man den Rang beispielsweise bezüglich der Zugriffshäufigkeiten der Intervalle sortiert, dann sind häufig angefragte Intervalle weiter oben in der BSL, als seltener angefragte. So kann die Suchzeit verringert werden.

4 Dynamic Biased Skip List

4.1 Einführung

Die *Dynamic Biased Skip List* (im folgenden *DBSL* genannt) ist eine verbesserte Variante der BSL, die Einfügen und Entfernen von Schlüsseln erlaubt. Bei BSL müssen beim Einfügen und Entfernen von Schlüsseln mit geringem Rang bis zu $O(\log n)$ Kopien erstellt werden. Dies ist für die Anwendung bei Bursty Access Patterns problematisch. In diesem Fall sind häufige Einfügen- und Entfernen-Operationen durchzuführen und der Rang der eingefügten Schlüssel bleibt oft niedrig. Dadurch befinden sie sich in ihrer Lebensspanne oft ausschließlich weit oben in der BSL. Die Idee ist, bei Schlüsseln die weit oben in der Liste sind, nur bis zu einer bestimmten Tiefe Kopien zu erstellen, um auf diese Weise Einfügen und Entfernen zu optimieren. Außerdem wird dadurch der

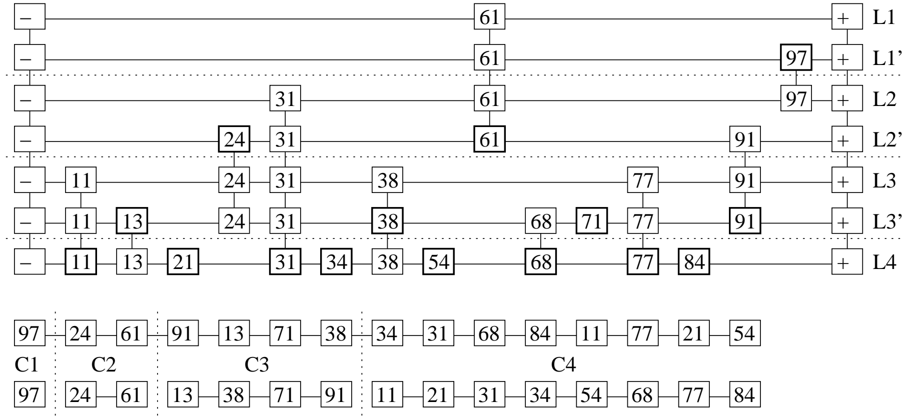


Abbildung 5: Dynamic Biased Skip List mit Masterliste und sortierten Teillisten

Speicherplatzbedarf verringert. Er ist zwar weiterhin asymptotisch im Erwartungswert $O(n)$, da aber bei Hardwareroutern der Speicherplatz häufig stark beschränkt ist, ist jegliche Verringerung des Speicherbedarfs vorteilhaft.

4.2 Default Key

Definition 5 (Default Key) Alle Schlüssel in Level L'_i, die der Klasse C_i angehören werden *Default Keys* genannt.

4.3 Konstruktion

Der grundlegende Unterschied bei der Konstruktion von DBSL gegenüber der BSL ist, dass bei DBSL keine automatischen Kopien in die höheren Levels existieren. Bei der Konstruktion der DBSL wird als erstes das Level $L_{\log n}$ erstellt, indem alle Schlüssel, die der Klasse $C_{\log n}$ angehören sortiert eingefügt werden.
Für $i < \log n$ wird L'_i erstellt, indem aus dem darunter liegenden Level L_{i+1} die Schlüssel mit Wahrscheinlichkeit $p = 0.5$ kopiert werden. Zusätzlich werden die Default Keys der Klasse C_i eingefügt. Von den Default Keys werden rekursiv mit Wahrscheinlichkeit $p = 0.5$ Kopien nach unten erstellt, bis höchstens zu $L_{\log n}$.
Die Levels L_i mit $i < \log n$ werden dadurch erstellt, dass alle Schlüssel aus L'_i mit Wahrscheinlichkeit $p = 0.5$ nach L_i kopiert werden.
Bei Ergun et al. [1] ist die Eingabe die sortierte Liste aller Schlüssel und die Masterliste wie bei BSL. Da die Default Keys in der Masterliste jedoch unsortiert vorliegen, würde die Laufzeit für die Konstruktion durch zusätzliche Sortieroperationen verschlechtert werden. Daher schlage ich vor, zusätzlich jeder Klasse C_i eine Teilliste zu übergeben, die alle Default Keys aus C_i nach ihrem Schlüsselwert sortiert enthält. Diese zusätzlichen Listen der werden nur für die Konstruktion benötigt und können danach gelöscht werden.

Satz 9 *Eine DBSL kann in erwarteter Laufzeit $O(n)$ erstellt werden.*

Beweis Die Laufzeitanalyse teilt sich in drei Teile auf: Die Laufzeit für das Einfügen der Default Keys, die Kopien nach oben und die Kopien nach unten.

$$\mathsf{E}[T_{\text{initialise}}] = \mathsf{E}[T_{\text{Kopien nach oben}}] + \mathsf{E}[T_{\text{Default Keys}}] + \mathsf{E}[T_{\text{Kopien nach unten}}]$$

Da für die Kopien nach oben genau so viel Zeit benötigt wird wie bei den BSL, lässt sich dies durch die Laufzeit für die Konstruktion der BSL abschätzen, welche erwartet $O(n)$ beträgt.

$$\mathsf{E}[T_{\text{Kopien nach oben}}] = O(n)$$

Bei der Analyse der Default Keys werden auch die Schlüssel von $L_{\log n}$ mitberücksichtigt, obwohl sie zwar streng nach Definition keine Default Keys sind, aber im Grunde genommen die gleichen Eigenschaften besitzen. Diese können durch $O(n)$ abgeschätzt werden. Für die restlichen Levels muss die Zeit abgeschätzt werden, um die Default Keys einzufügen. Zu beachten ist, dass in L'_i die Schlüssel aus Klasse C_i enthalten sind, von denen 2^{i-1} Stück existieren. Diese liegen bereits in sortierter Reihenfolge vor, so dass sie in Zeit $O(|C_i|) = O(2^i)$ eingefügt werden können. Zusätzlich müssen noch die Kopien der Schlüssel aus dem Level darunter betrachten werden. Von diesen existieren erwartet wiederum $O(2^i)$ Stück.

$$\begin{aligned}
\mathsf{E}[T_{\text{Default Keys}}] &\leq \mathsf{E}[T_{\text{Schlüssel in } L_{\log n}}] + \sum_{i=1}^{\log n} \mathsf{E}[T_{\text{Default Keys in } L'_i}] \\
&= O(n) + \sum_{i=1}^{\log n} \Pr[\text{Kopie eines Schlüssels}] \cdot \mathsf{E}[|L_{i+1}|] + O(|C_i|) \\
&= O(n) + \sum_{i=1}^{\log n} \frac{1}{2} \cdot O(2^i) + O(2^i) \\
&= O(n) + \frac{3}{2} \sum_{i=1}^{\log n} O(2^i) \\
&= O(n)
\end{aligned}$$

Für die Anzahl der Kopien nach unten müssen ausgehend von jedem Default Key berechnet werden, wie viele Kopien erstellt werden. Es existieren $O(n)$ Default Keys. Bei der Konstruktion der Kopien nach unten muss jetzt jedoch beachtet werden, dass man nicht sofort vom linken Nachbarn aus über einen Zeiger in das darunter liegende Level (im Folgenden *Down-Link* genannt) gelangen kann, weil dort eventuell keine Kopie nach unten existiert, so dass man weiter nach links gehen muss, um einen Down-Link zu finden. Daher lässt sich die Analyse aufteilen in die Schritte nach links (bis zu einem Schlüssel mit Down-Link), einem Schritt nach unten und die Schritte nach rechts (zur Einfügestelle).

$$\begin{aligned}
\mathsf{E}[T_{\text{Kopien nach unten}}] &= O(n) \cdot \mathsf{E}[|\text{Schritte pro Kopie}|] \\
&= O(n) \cdot (\mathsf{E}[|\text{Schritte links}|] + 1 + \mathsf{E}[|\text{Schritte rechts}|])
\end{aligned}$$

Nach links muss auf jeden Fall ein Schritt gemacht werden. Hinzu addieren muss man die erwartete Anzahl der Schlüssel ohne Down-Link die traversiert werden. Für die Schritte nach rechts gilt wieder, dass ein Schritt auf jeden Fall gemacht werden muss, aber noch zusätzliche Schritte nötig sind, die durch Kopien aus dem darunter liegenden Level entstehen (analog zu den BSL). Da sowohl für die Kopien nach unten als auch für die Kopien nach oben gilt, dass sie mit einer Wahrscheinlichkeit von $p = 0.5$ erstellt werden, gilt im Erwartungswert, dass jeder zweite Schlüssel einen Down-Link hat und analog dazu von unten nach oben jeweils ein neuer Schlüssel hinzukommt.

$$\begin{aligned}
\mathsf{E}[T_{\text{Kopien nach unten}}] &= O(n) \cdot (\mathsf{E}[|\text{Schritte links}|] + 1 + \mathsf{E}[|\text{Schritte rechts}|]) \\
&= O(n) \cdot (1 + \mathsf{E}[|\text{Schlüssel ohne Down-Link}|] + 1 + 1 \\
&\quad + \mathsf{E}[|\text{Neue Schlüssel von nächstem Level}|]) \\
&= O(n) \cdot (1 + 1 + 1 + 1 + 1) \\
&= O(n)
\end{aligned}$$

Somit ergibt sich insgesamt eine erwartete Laufzeit für die Konstruktion von $O(n)$.

$$\begin{aligned}
\mathsf{E}[T_{\text{initialise}}] &\leq \mathsf{E}[T_{\text{Kopien nach oben}}] + \mathsf{E}[T_{\text{Default Keys}}] + \mathsf{E}[T_{\text{Kopien nach unten}}] \\
&= 3 \cdot O(n) \\
&= O(n)
\end{aligned}$$

□

4.4 Suche nach einem Schlüssel

Auch bei der Suche muss nun beachtet werden, dass nicht jeder Schlüssel einen Down-Link hat. Dadurch muss in einem solchen Fall weiter zum nächsten linken Nachbarn gegangen werden, bis schließlich ein Down-Link gefunden wird. Auch hier werden wieder die $-\infty$-Schlüssel benötigt. Dies sind nämlich die Schlüssel bei denen spätestens ein Down-Link gefunden werden kann.

Lemma 2 *Sei k ein Schlüssel in der DBSL. Dann existieren nur erwartet konstant viele Kopien von k in der DBSL.*

Beweis Der Beweis ist analog zum Beweis der Aussage für BSL (siehe Lemma 1) □

Satz 10 *Ein Schlüssel in einer DBSL kann in erwarteter Laufzeit $O(\log r(k))$ gesucht werden. Existiert der Schlüssel nicht, so beträgt die erwartete Suchzeit $O(\log n)$.*

Beweis Auch hier werden wieder die vertikalen und horizontalen Schritte separat betrachtet. Die vertikalen sind wieder beschränkt durch $O(\log r(k))$, mit dem gleichen Argument, wie bei den BSL. Dort waren für die Suche pro Level nur konstant viele horizontale Schritte nötig. Zusätzlich müssen jetzt aber zusätzliche Schritte gemacht werden, da es Schlüssel ohne Down-Link gibt.

$$\begin{aligned} \mathsf{E}[T_{\text{search}}] &= \mathsf{E}[|\text{Schritte vertikal}|] + \mathsf{E}[|\text{Schritte horizontal}|] \\ &= O(\log r(k)) + O(\log r(k)) \cdot \mathsf{E}[|\text{Schritte vertikal pro Level}|] \\ &\leq O(\log r(k)) + O(\log r(k)) \cdot (c + \mathsf{E}[|\text{Schlüssel ohne Down-Link}|]) \end{aligned}$$

Aus dem Beweis für die Konstruktion ist jedoch bereits bekannt, dass die Anzahl der hierdurch zusätzlich entstehenden Schritte konstant ist.

$$\begin{aligned} \mathsf{E}[T_{\text{search}}] &\leq O(\log r(k)) + O(\log r(k)) \cdot (c + \mathsf{E}[|\text{Schlüssel ohne Down-Link}|]) \\ &\leq O(\log r(k)) + O(\log r(k)) \cdot (c + d) \\ &= O(\log r(k)) \end{aligned}$$

□

4.5 Speicherbedarf

Satz 11 *Eine DBSL mit n Schlüsseln als Eingabe benötigt erwarteten Speicherplatz von $O(n)$.*

Beweis Der Speicherbedarf der DBSL ist beschränkt durch den Speicherverbrauch der BSL und somit erwartet $O(n)$. □

5 Anwendungen für DBSL

5.1 Szenarien

Das Einfügen und Löschen von Schlüsseln in DBSL hängt nicht mehr alleine von der DBSL selbst ab, sondern auch davon, in welcher Weise die Masterliste verwaltet wird. Daher soll hier zunächst die Grundidee der Updatestrategie für die Masterliste betrachtet werden. In der Vorstellung der Algorithmen werden diese Ideen dann verfeinert.
Es gibt zwei häufige Anwendungsfälle, die betrachtet werden: Zum einen den Fall, dass eine relativ statische Umgebung vorliegt in der Schlüssel eingefügt und gelöscht werden können, wobei jedoch davon ausgegangen wird, dass dies relativ selten geschieht und darüber hinaus die Zugriffswahrscheinlichkeiten auf die verschiedenen Schlüssel über die Zeit hinweg stabil bleibt. Dieses Szenario kann man sich in der Praxis beispielsweise vorstellen für einen Router, der IP-Pakete von Übersee empfangt und diese auf die verschiedenen europäischen Länder routen soll. In diesem Fall bietet

es sich eine *Most Frequently Accessed*-Strategie für die Sortierung der Masterliste an, so dass die Masterliste nach der Häufigkeit der Anfragen sortiert ist. Für den Fall, dass Bursty Access Patterns vorliegen bietet sich hingegen eine *Most Recently Accessed*-Strategie an, welche die Schlüssel der Masterliste nach dem Zeitpunkt sortiert, an dem sie zuletzt angefragt wurden. Da bei Bursty Access Patterns eine geringe Anzahl von Schlüsseln kurz hintereinander häufig angefragt werden, bleiben diese dadurch in der DBSL möglichst weit oben.

5.2 Dynamic Biased Skip List für MFA (DBSL-MFA)

5.2.1 Suche nach einem Schlüssel

Die Suche wird wie bei DBSL durchgeführt. Nun muss jedoch für einen gefundenen Schlüssel der Zugriffszähler[2] erhöht werden. Sei x dieser Schlüssel und Rang des gefundenen Schlüssels $r(x)$. Durch Erhöhung des Zugriffszählers kann es geschehen, dass die Zugriffshäufigkeit von x höher ist, als für den Schlüssel mit Rang $r(x) - 1$. Sei dieser Schlüssel y. Befinden sich x und y in der gleichen Klasse, so werden diese beiden Schlüssel in der Masterliste vertauscht und die Suche ist beendet. Sei x jedoch in Klasse C_i und y in der Klasse C_{i-1}, so muss nach dem Vertauschen der beiden Schlüssel in der Masterliste sichergestellt werden, dass sich die Schlüssel in den richtigen Klassen befinden. Hierfür werden der Schlüssel x und seine Kopien um zwei Levels nach oben, sowie der Schlüssel y und seine Kopien um zwei Levels nach unten verschoben. Hierdurch wird sichergestellt, dass sie sich wieder in den richtigen Klassen befinden.
Ergun et al. [1] verzichten auf Laufzeitbeweise für DBSL-MFA. Der Vollständigkeit halber, werden diese jedoch hier geführt.

Satz 12 *Ein Schlüssel in einer DBSL-MFA kann in erwarteter Laufzeit $O(\log r(k))$ gesucht werden. Existiert der Schlüssel nicht, so beträgt die erwartete Suchzeit $O(\log n)$.*

Beweis Nehme oBdA. an, die Schlüssel müssen vertauscht werden. Die Suchzeit kann abgeschätzt werden durch die Suchzeit in DBSL, welche erwartet $O(\log r(k))$ beträgt, und der Zeit, die benötigt wird um die Default Keys mit ihren Kopien zu verschieben.

$$\begin{aligned} \mathsf{E}[T_{\text{search}}] &\leq \mathsf{E}[T_{\text{search}}^{DBSL}] + \mathsf{E}[T_{\text{Verschieben}}] \\ &= O(\log r(k)) + \mathsf{E}[T_{\text{Verschieben}}] \end{aligned}$$

Die beiden Default Keys und ihre nach Lemma 2 erwartet konstant vielen Kopien müssen um zwei Levels verschoben werden. Die Verschiebung benötigt also nur erwartet konstante Zeit.

$$\begin{aligned} \mathsf{E}[T_{\text{search}}] &\leq O(\log r(k)) + \mathsf{E}[T_{\text{Verschieben}}] \\ &= O(\log r(k)) + c \\ &= O(\log r(k)) \end{aligned}$$

Die Suchzeit beträgt somit erwartet $O(\log r(k))$ Existiert der Schlüssel nicht, so endet die Suche im untersten Level. Somit sind in dem Fall erwartet $O(\log n)$ Schritte nötig. □

5.2.2 Einfügen eines Schlüssels

Beim Einfügen eines Schlüssels muss dieser aufgrund der MRA-Strategie als letztes in die Masterliste eingefügt werden. Ist die Klasse in die der Schlüssel eingefügt wird noch nicht voll, so fügt man den Schlüssel in das unterste Level der DBSL ein und erstellt randomisierte Kopien in höher gelegene Levels. Ist die Klasse jedoch bereits voll, so wird für den neu eingefügten Schlüssel eine neue Klasse erstellt. Die DBSL wird um zwei Levels erweitert. Auch jetzt wird der Schlüssel in das unterste Level (und somit in die neu erstellte Klasse) eingefügt und randomisierte Kopien erstellt.

[2] Der Zugriffszähler wird hier nicht explizit angegeben. Er kann aber in den Knoten der Masterliste gespeichert werden.

Satz 13 *Das Einfügen eines Schlüssels in eine DBSL-MFA ist in erwarteter Laufzeit $O(\log n)$ möglich.*

Beweis OBdA. falle der neue Schlüssel in die neue Klasse. Die Laufzeit setzt sich zusammen aus der Suche, der Erstellung zusätzlicher Levels und den Kopien des Schlüssels. Die Suche ist beschränkt durch $O(\log n)$ erwarteter Laufzeit. Die neue Klasse besteht aus zwei Levels, die in konstanter Zeit erstellt werden können. Für den neuen Schlüssel müssen nach Lemma 2 nur erwartet konstant viele Kopien erstellt werden.

$$\begin{aligned}\mathsf{E}[T_{\text{insert}}] &\leq \mathsf{E}[T_{\text{Suche}}] + \mathsf{E}[T_{\text{Zusatzlevels}}] + \mathsf{E}[T_{\text{Kopien}}] \\ &= O(\log n) + c + d \\ &= O(\log n)\end{aligned}$$

□

5.2.3 Entfernen eines Schlüssels

Zum Entfernen eines Schlüssels wird dieser zunächst gesucht und samt seiner Kopien aus der Datenstruktur gelöscht. Nun ist jedoch zu beachten, dass die Klasse C_i, in der sich der Schlüssel zuvor befand, wieder auf 2^{i-1} Schlüssel aufgefüllt werden muss. Hierzu wird der Schlüssel mit dem kleinsten Rang aus der Klasse C_{i+1} der Klasse C_i zugeordnet. In der DBSL wird der Schlüssel nun samt seiner Kopien um zwei Levels verschoben. Rekursiv wird mit dieser Methode dann die Klasse C_{i+1} auf seine reguläre Größe gebracht.

Satz 14 *Das Entfernen eines Schlüssels aus einer DBSL-MFA ist in erwarteter Laufzeit $O(\log n)$ möglich.*

Beweis Die Suche geht, wie bereits bewiesen, in $O(\log r(k))$ erwarteter Laufzeit. Das Löschen des Schlüssels und seiner Kopien in erwartet konstanter Zeit.

$$\begin{aligned}\mathsf{E}[T_{\text{remove}}] &= \mathsf{E}[T_{\text{Suche}}] + \mathsf{E}[T_{\text{Löschen}}] + \mathsf{E}[T_{\text{Verschieben}}] \\ &= O(\log r(k)) + c + \mathsf{E}[T_{\text{Verschieben}}]\end{aligned}$$

Nun ist noch die Zeit zu betrachten, die zum Verschieben der Schlüssel an den Klassengrenzen nötig ist. Da die Masterliste aus $O(n)$ Schlüsseln und jede Klasse aus 2^{i-1} Schlüsseln besteht, existieren $O(\log n)$ Klassen. Somit befinden sich $O(\log n)$ Schlüssel mit ihren Kopien an den Klassengrenzen, die um zwei Levels hoch verschoben werden müssen. Somit ergibt sich eine erwartete Laufzeit von $O(\log n)$.

$$\begin{aligned}\mathsf{E}[T_{\text{remove}}] &= O(\log r(k)) + c + \mathsf{E}[T_{\text{Verschieben}}] \\ &= O(\log r(k)) + c + d \cdot O(\log n) \\ &= O(\log n)\end{aligned}$$

□

5.3 Dynamic Biased Skip List für MRA (DBSL-MRA)

5.3.1 Lazy Updating

Um Einfüge- und Entferne-Operationen noch weiter zu optimieren wird für die DBSL-MRA jetzt ein Lazy-Updating-Schema eingeführt. Bei den DBSL-MFA hatte man statische Klassengrößen $|C_i| = 2^{i-1}$. Dies führte dazu, dass durch das Verschieben der Schlüssel an den Klassengrenzen bei jeder Einfügen- oder Entfernen-Operation $O(\log n)$ Zeit benötigt wurde. Da bei Bursty Access Patterns eine Vielzahl von Einfüge- und Entferne-Operationen entstehen, ist das Ziel, dies weiter

C_1	1	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
C_2	2	**3**	**2**	**3**	**2**	**3**	**2**	**3**	**2**
C_3	4	4	**6**	6	**4**	4	**6**	6	**4**
C_4	8	8	8	8	**12**	12	12	12	**8**
C_5	16	16	16	16	16	16	16	16	**24**

Tabelle 1: Klassengrößen nach 8 Einfügungen

optimieren. Deshalb sind die Klassengrößen nun nicht mehr statisch, sondern dynamisch $|C_i| \in (2^{i-2}, \ldots, 2^i - 1)$.
Durch die dynamischen Klassengrößen wird erreicht, dass die Verschiebungen, die nötig sind um die Klassengrößen nach Einfügen oder Entfernen von Schlüsseln wiederherzustellen, in vielen Fällen nur die oberen Klassen betreffen. In Tabelle 1 sieht man dies anhand eines Beispiels, bei dem in ein bestehendes DBSL 8 Schlüssel hintereinander eingefügt werden. Die Klassen in denen es zu Verschiebungen kommt sind hervorgehoben.

5.3.2 Suche nach einem Schlüssel

Aufgrund der MRA-Strategie bekommt ein Schlüssel in der Masterliste, nachdem er gefunden wurde den Rang 1. Um die Analyse zu vereinfachen wird davon ausgegangen, dass die Klassengrößen konstant bleiben. Sei der gesuchte Schlüssel in Klasse C_i, so wird nach dem Verschieben des Schlüssels auf Rang 1 für die Klassen C_j mit $j \in [1, \ldots, i-1]$ jeweils der Schlüssel an der Klassengrenze in die nächste Klasse C_{j+1} verschoben.

Satz 15 *Ein Schlüssel in einer DBSL-MRA kann in erwarteter Laufzeit $O(\log r(k))$ gesucht werden. Existiert der Schlüssel nicht, so beträgt die erwartete Suchzeit $O(\log n)$.*

Beweis Die Suchzeit in DBSL beträgt erwartet $O(\log r(k))$. Zusätzlich muss nun noch die Zeit betrachtet werden, die benötigt wird, um den Default Key mit seinen Kopien in das oberste Level zu verschieben und das Verschieben der Schlüssel an den Klassenrändern.
Der gesuchte Schlüssel lässt sich mit seinen erwartet konstant vielen Kopien in $O(\log r(k))$ erwarteter Zeit in das oberste Level verschieben, da sich der Schlüssel in der Klasse $C_{\log r(k)}$ befindet und somit nur um $O(\log r(k))$ Levels verschoben werden muss. Für die Schlüssel an den Klassengrenzen gilt, dass nur maximal $O(\log r(k))$ viele solcher Schlüssel zu verschieben sind. Diese werden um zwei, somit konstant viele Levels verschoben, so dass sich Gesamtlaufzeit von erwartet $O(\log r(k))$ ergibt.

$$\begin{aligned} \mathsf{E}[T_{\text{search}}] &= \mathsf{E}[T_{\text{search}}^{DBSL}] + \mathsf{E}[|\text{Verschiebungen}|] \\ &= 2 \cdot O(\log r(k)) \\ &= O(\log r(k)) \end{aligned}$$

□

5.3.3 Einfügen eines Schlüssels

Definition 6 ($r_{max}(k)$) $r_{max}(k)$ bezeichnet den maximalen Rang, den ein Schlüssel k in seiner Lebensspanne erhält.

Beim Einfügen eines Schlüssels wird dem neuen Schlüssel gemäß der MRA-Strategie der Rang 1 zugeordnet. Dementsprechend wird der Schlüssel an den Anfang der Masterliste eingefügt und der Rang aller weiteren Schlüssel um 1 erhöht. Darüber hinaus wird der Schlüssel in das Level L_1' eingefügt und wie bereits in den bisherigen Algorithmen betrachtet, randomisierte Kopien erstellt. Nun wird rekursiv zunächst für die erste Klasse geprüft, ob für die Klasse die Anzahl der enthaltenen Schlüssel innerhalb der durch das Lazy Updating Schema gegebenen Grenzen liegt. Ist dies der Fall so ist nichts weiter zu tun. Sind zu viele Schlüssel in der Klasse enthalten, so

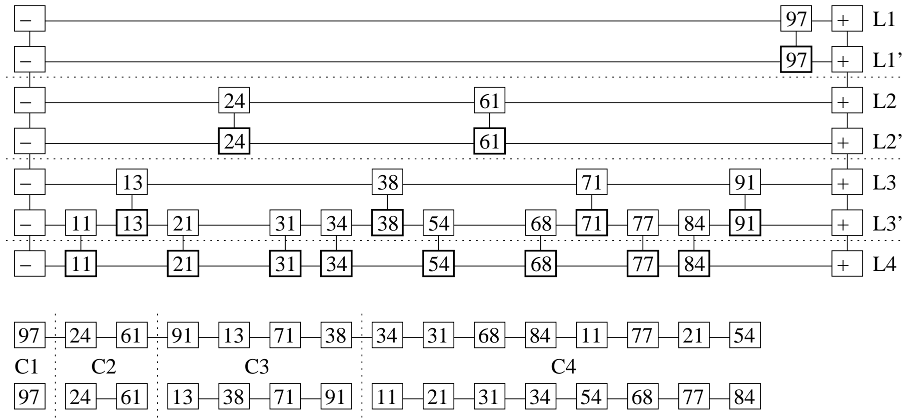


Abbildung 6: Speziell modifizierte DBSL-Implementation

wird die ranghöhere Hälfte der vorhandenen Schlüssel in das nächste Level verschoben und die Klassengröße überprüft, bis alle Klassengrößen in den gegebenen Intervallen liegt.

Satz 16 *Das Einfügen eines Schlüssels k in eine DBSL-MRA ist in amortisierter erwarteter Laufzeit $O(\log r_{max}(k))$ möglich.*

5.3.4 Entfernen eines Schlüssels

Das Entfernen eines Schlüssels geht auf ähnliche Weise, wie das Einfügen. Zunächst wird der bereits beschriebene Suchalgorithmus angewendet. Hierdurch wird dem Schlüssel der Rang 1 zugeordnet. Dann wird der Schlüssel samt seiner Kopien aus der DBSL, sowie der Masterliste entfernt. Nun werden, wie beim Einfügen, rekursiv die Klassen wieder aufgefüllt, bis die Klassengrößen in den gültigen Grenzen liegen. Hierfür wird aus dem nächsten Level jeweils die rangniedrigere Hälfte der enthaltenen Schlüssel verschoben.

Satz 17 *Das Löschen eines Schlüssels k aus einer DBSL-MRA ist in amortisierter erwarteter Laufzeit $O(\log r_{max}(k))$ möglich.*

Auf die Laufzeitbeweise für Einfügen und Entfernen von Schlüsseln kann aus Platzgründen nicht näher eingegangen werden. Wie in Abb. 1 zu sehen, werden in vielen Fällen bei Update-Operationen nur wenige Schlüssel verschoben, in manchen Fällen jedoch viele. Die Idee besteht nun darin, für die billigeren Einfügeoperation mehr Kosten zu berechnen um diese dann mit den teuren Operationen zu verrechnen. Der formale Laufzeitbeweis wird in Ergun et al.[6] ausführlich beschrieben.

5.4 Anmerkungen zur Implementierung von Ergun et al.

Um auf einige Probleme in der Implementierung der DBSL-Algorithmen von Ergun et al. [1] einzugehen, wird ein Lemma bewiesen.

Lemma 3 *DBSL bei denen von allen Default Keys in L'_i immer genau eine Kopie in das nächsthöhere Level L_i erstellt wird und keine Kopien in untere Level ergeben eine Datenstruktur mit Suchzeit $O(n)$. (Vergleiche Abb. 6)*

Beweis In dem Fall müsste bei einer Suche für alle L_i mit $i = 1, \dots, \log n - 1$ jeweils $O(2^i)$ Schritte bei der Suche gemacht werden, da man immer wieder zurück zum $-\infty$-Schlüssel muss. Für die restlichen Levels benötigt die Suche pro Level nur konstant viele Schritte.

Pro Level sind dann die Anzahl der Schritte erwartet:

$$\frac{1}{O(\log n)} \cdot \sum_{i=1}^{\log n-1} O(2^i) + c = O\left(\frac{n}{\log n}\right)$$

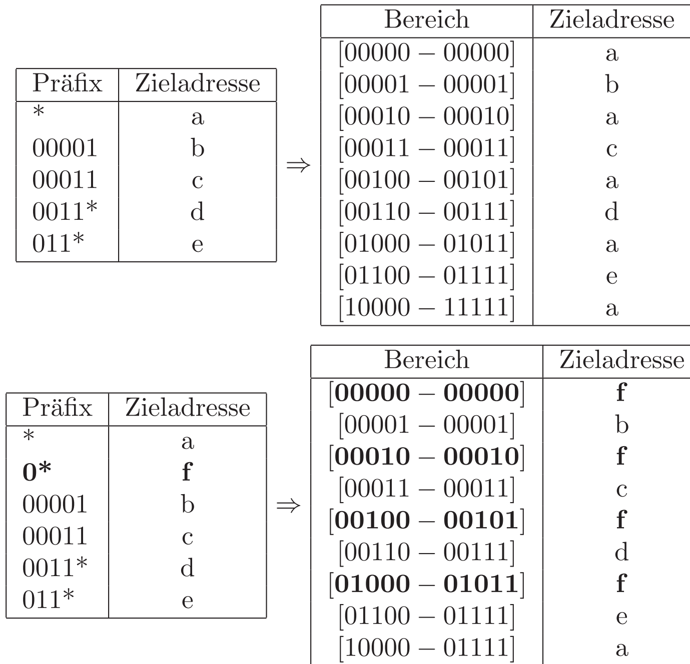
Somit benötigt in diesem Fall eine Suche nach einem Schlüssel in Klasse C_i die Laufzeit:

$$O\left(\frac{n}{\log n}\right) \cdot O(\log r(k)) = O(n)$$

□

Bei Ergun et al. [1] wird nicht angegeben, wie vorzugehen ist, wenn die Kopien der Schlüssel aus der Datenstruktur oben oder unten aufgrund von Verschiebungen über die Klassengrenzen hinaus ragen. Nimmt man an, dass die Höhe der Datenstruktur bei Verschiebungen der Schlüssel konstant bleibt, so müssen diese über die Grenzen ragenden Schlüssel gelöscht werden.
Werden über einen längeren Zeitraum bei einer DBSL mit MRA-Strategie viele Schlüssel gleichwahrscheinlich angefragt, aber keine Einfüge-Operationen durchgeführt, so führt dies dazu, dass im Laufe der Zeit die Datenstruktur immer mehr an Kopien verliert, was auf Dauer einen negativen Effekt auf die Laufzeit für die Suche hat.
Die lässt sich gut anhand des Worst Case sehen: Man habe die DBSL, wie in Abb. 5 vorliegen mit DBSL-MRA (Kap. 5.2) als Updatestrategie. Man frage sukzessive die Elemente der Masterliste vom größten Rang zum kleinsten Rang an. In dem gegebenen Beispiel also 54, 21, . . . , 97. Dann wird jeder Schlüssel nach der Anfrage an Position 1 der Rangliste verschoben. Der Default Key wird somit in L'_1 verschoben und es gibt noch höchstens eine Kopie nach oben. Da dies für alle Schlüssel durchgeführt wird, haben nach der Anfragesequenz alle Schlüssel nur noch eine Kopie nach oben. Da jedoch jeder Schlüssel durch diese Anfragesequenz einmal in $L_{\log n}$ verschoben wird, müssen alle Kopien nach unten gelöscht werden. Somit liegt am Ende der Anfragesequenz eine DBSL vor, wie in Abbildung 6. Von dieser wurde jedoch gezeigt, dass diese eine Suchlaufzeit von $O(n)$ besitzt. Bis zur nächsten Einfügeoperation (bei der ein neuer Schlüssel mit randomisierten Kopien nach unten erstellt wird) wird jede Anfrage eine Laufzeit von $O(n)$ besitzen.
Eine solch geartete Anfragesequenz ist in der Praxis für Bursty Access Patterns sehr unwahrscheinlich, weshalb es im Allgemeinen keinen Einfluss auf die erwartete Suchlaufzeit bei DBSL-MRA hat. Trotzdem möchte ich an dieser Stelle eine möglichen Ansatz aufzeigen, um das Problem zu umgehen. Anstatt, wie bei Ergun et al. [1] angegeben, wenn Schlüssel verschoben werden müssen, diese samt ihrer Kopien zu verschieben, wird wie folgt vorgegangen: Vor dem Verschieben werden bis auf den Default Key alle Kopien gelöscht. Erst dann wird der Default Key verschoben und danach rekursiv mit Wahrscheinlichkeit $p = 0.5$ wieder randomisierte Kopien sowohl nach oben, als auch nach unten bis höchstens zum oberen bzw. unteren Rand der Datenstruktur erstellt. Nach Lemma 2 existieren nur erwartet konstant viele Kopien eines Default Key, so dass dies keinen negativen Einfluss auf die Laufzeit hat.
Das Neuerstellen der Kopien hat neben der Lösung für die Problematik zusätzlich noch den Vorteil, dass durch häufigere Zufallsprozesse die DBSL die gesamte DBSL-Struktur zufälliger wird. Dies ist eine für randomisierte Datenstrukturen wünschenswerte Eigenschaft.
Man kann sich auch für DBSL-MFA (Kap. 5.2) spezielle Sequenzen überlegen, die zu einer Konfiguration der der DBSL wie in Abbildung 6 führen. Da hier jedoch die Schlüssel immer nur um einen Rang in der Masterliste verschoben werden, benötigt es deutlich mehr Suchoperationen, bis eine Laufzeitverschlechterung eintritt, weshalb darauf hier nicht näher eingegangen wird.
Eine weitere Möglichkeit, diesem Problem zu entgehen wäre, anstatt bei jeder Verschiebeoperation neue Kopien zu erstellen die Datenstruktur nach unten hin um das Doppelte zu vergrößern. Dieser zusätzliche Raum würde ausschließlich dazu genutzt, die Schlüssel, welche über $L_{\log n}$ hinausragen aufzunehmen. Hierdurch verändern sich die erwarteten Laufzeiten nicht und der Effekt der Ausdünnung würde nur die nach oben gerichteten Kopien betreffen.[3]

[3] Aufgrund der Suchzeit von $O(\log r(k))$ ist es nicht möglich, nach oben hin die Datenstruktur um $O(\log n)$ zu erhöhen, ohne die Suchalgorithmen ein weiteres Mal anzupassen.



Präfix	Zieladresse
*	a
00001	b
00011	c
0011*	d
011*	e

⇒

Bereich	Zieladresse
[00000 – 00000]	a
[00001 – 00001]	b
[00010 – 00010]	a
[00011 – 00011]	c
[00100 – 00101]	a
[00110 – 00111]	d
[01000 – 01011]	a
[01100 – 01111]	e
[10000 – 11111]	a

Präfix	Zieladresse
*	a
0*	**f**
00001	b
00011	c
0011*	d
011*	e

⇒

Bereich	Zieladresse
[**00000 – 00000**]	**f**
[00001 – 00001]	b
[**00010 – 00010**]	**f**
[00011 – 00011]	c
[**00100 – 00101**]	**f**
[00110 – 00111]	d
[**01000 – 01011**]	**f**
[01100 – 01111]	e
[10000 – 01111]	a

Abbildung 7: Einfügen eines neuen Präfix

6 Weitere Ansätze

6.1 Sahni und Kim (2004)

6.1.1 Problemstellung

Lampson et al. [4] weisen in einer Arbeit die 1999, vor Ergun et al. [1], veröffentlicht wurde, darauf hin, dass die Einfügung eines neuen Präfix in die Präfixtabelle im Worst Case dazu führt dass die Zieladresse in $O(n)$ Intervallen verändert werden müssen. Vergleiche hierzu Abbildung 7, in der die veränderten Werte beim Einfügen eines neuen Präfix hervorgehoben sind. Sahni und Kim [7] greifen dieses Problem auf und weisen darauf hin, dass hierdurch eine Einfügung eines neuen Präfix in eine BSL zu einer Laufzeit von $O(n)$ führen kann, da die Präfixinformation in der Masterliste gespeichert vorliegen und somit $O(n)$ viele Schlüssel der Masterliste angepasst werden müssen. Sie stellen in ihrer Arbeit zwei Datenstrukturen für Bursty Access Patterns vor, von der eine die BSL erweitert.

6.1.2 Biased Skip Lists with Prefix Trees

In Sahni und Kim [8] wird die Datenstruktur *Collection of Red-Black Trees (CRBT)* eingeführt. Diese Datenstruktur verwendet einen *Basic Interval Tree (BIT)*[4] um das Intervall zu finden und verzweigt dann in einen Knoten eines Präfixbaums. Für k Präfixe existieren k Präfixbäume. Diese Menge von Präfixbäumen wird *Collection of Prefix Trees (CPT)* genannt. Sie werden durch Red-Black Trees [9] implementiert und sind Suchbäume für die Präfixe, die durch die verschachtelten Intervalle repräsentiert werden. Abbildung 8 zeigt beispielsweise Präfixe Pi und ihre dazugehörigen Intervalle r_i in grafischer Darstellung. Hieraus wird die dazugehörige CPT aus Abbildung 9 erstellt. CRBTs haben eine Suchlaufzeit von $O(\log n)$ sind jedoch nicht für Bursty Access Patterns entwickelt worden.
Sahni und Kim [7] kombinieren daher die CPTs mit den BSLs von Ergun et al. [1], indem die ihre BIT durch die BSL ersetzen. Diese neue Datenstruktur nennen sie *Biased Skip Lists with*

[4]Für Beispiele siehe Sahni und Kim [7, 8]

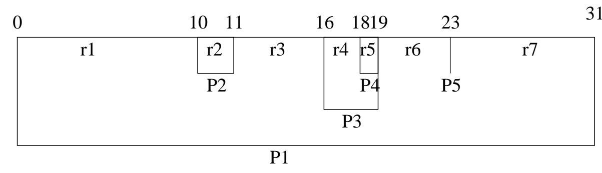


Abbildung 8: Grafische Darstellung von Präfixen und dazugehörigen Intervallen

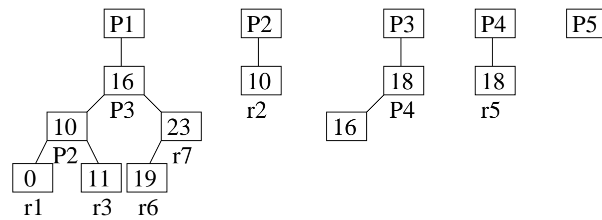


Abbildung 9: Collection of Prefix Trees für die Präfixe aus Abb. 8

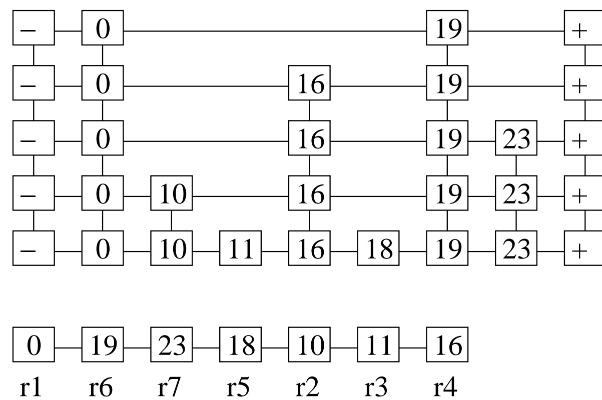


Abbildung 10: Biased Skip List

Prefix Trees (BSLPT). Die Schlüssel der Masterliste enthalten jetzt keine Präfixinformationen mehr, sondern nur die Intervallgrenzen und einen Zeiger zu einem Knoten eines Präfixbaums. Diese sind in Abbildung 10 als Werte unter den Knoten der Masterliste angegeben. Beim Einfügen eines neuen Präfix wird nun, wenn sich die Intervallgrenzen eines Knotens der Masterliste nicht ändern, bloß die Information im Präfixbaum geändert. Falls durch den neuen Präfix ein Intervall aufgeteilt werden muss, so werden in der Masterliste der Knoten in zwei aufgeteilt und die Zeiger zum Präfixbaum angepasst. Diese neue Datenstruktur hat auch dann eine erwartete Such- und Updatelaufzeit von $O(\log n)$, wenn $O(n)$ Intervalle neue Präfixinformationen erhalten, da sie sich nur im CPT verändern. Sahni und Kim [7] merken an, dass zur Implementierung der BSLPT anstelle einer BSL auch eine DBSL verwendet werden kann.

6.1.3 Collection of Splay Trees

Sahni und Kim [7] stellen darüber hinaus noch eine weitere Datenstruktur für Bursty Access Patterns vor. Diese verwendet eine alternative Implementierung der BIT ($ABIT$) und kombinieren diese mit einer Menge von Splay Trees [10] zu einer *Collection of Splay Trees (CST)*. Auf disese Datenstruktur wird in dieser Hausarbeit nicht näher eingegangen. Zu erwähnen bleibt jedoch, dass diese Datenstruktur eine amortisierte Such- und Updatelaufzeit von $O(\log n)$ hat.

7 Zusammenfassung

In dieser Hausarbeit wurde auf die von Ergun et al. [1] vorgestellten Datenstrukturen für das Longest-Matching-Prefix-Problem eingegangen. Das Hauptaugenmerk liegt in der DBSL-MRA, welche für Bursty Access Patterns eine erwartete Suchlaufzeit von $O(\log r(k)$ und eine amortisierte erwartete Laufzeit von $O(\log r_{max}(k))$ für Update-Operationen hat. Darüber hinaus wurde kurz auf eine Erweiterung durch Sahni und Kim [7] eingegangen, welche die Laufzeit in der Praxis von $O(n)$ auf erwartet $O(\log n)$ verbessert, wenn linear viele Präfixinformationen verändert werden. Bei allen Ansätzen für LMP muss jedoch immer beachtet werden, dass mit steigender Komplexität der Datenstrukturen die Implementierung in Hardwareroutern schwieriger wird, weshalb sich diese Datenstrukturen eher für den Einsatz in Softwareroutern eignen.

Literatur

[1] Ergun, Funda ; Mittra, Suvo ; Sahinalp, Suleyman C. ; Sharp, Jonathan ; Sinha, Rakesh K.: A Dynamic Lookup Scheme for Bursty Access Patterns. In: *INFOCOM*, IEEE, 2001, S. 1444–1453

[2] Lin, Steven ; McKeown, Nick: A simulation study of IP switching. In: *SIGCOMM '97: Proceedings of the ACM SIGCOMM '97 conference on Applications, technologies, architectures, and protocols for computer communication.* New York, NY, USA : ACM Press, 1997, S. 15–24

[3] Mittra, Suvo ; Basu, Anindya: Packet classification: An argument for a working set model. 1999. – Forschungsbericht

[4] Lampson, Butler ; Srinivasan, Venkatachary ; Varghese, George: IP lookups using multiway and multicolumn search. In: *IEEE/ACM Trans. Netw.* 7 (1999), Nr. 3, S. 324–334

[5] Pugh, William: Skip Lists: A Probabilistic Alternative to Balanced Trees. In: *Workshop on Algorithms and Data Structures*, 1989, S. 437–449

[6] Ergun, Funda ; Sahinalp, Suleyman C. ; Sharp, Jonathan ; Sinha, Rakesh: Biased dictionaries with fast insert/deletes. In: *STOC '01: Proceedings of the thirty-third annual ACM symposium on Theory of computing.* New York, NY, USA : ACM Press, 2001, S. 483–491

[7] Sahni, Sartaj ; Kim, Kun S.: Efficient Dynamic Lookup For Bursty Access Patterns. In: *Int. J. Found. Comput. Sci.* 15 (2004), Nr. 4, S. 567–591

[8] Sahni, Sartaj ; Kim, Kun S.: O(log n) Dynamic Packet Routing. In: *ISCC '02: Proceedings of the Seventh International Symposium on Computers and Communications (ISCC'02).* Washington, DC, USA : IEEE Computer Society, 2002, S. 443

[9] Guibas, Leonidas J. ; Sedgewick, Robert: A Dichromatic Framework for Balanced Trees. In: *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, IEEE, 1978, S. 8–21

[10] Sleator, Daniel D. ; Tarjan, Robert E.: Self-adjusting binary search trees. In: *J. ACM* 32 (1985), Nr. 3, S. 652–686