BADS'17

– Proceedings –

Preface

This volume contains the papers presented at BADS 2017, the 2nd Symposium on Breakthroughs in Advanced Data Structures held between July 31 and August 2 in Dortmund, Germany, as part of the MSc-seminar "Advanced Data Structures."

There were 17 submissions. Each submission was reviewed by at least 1, and on the average 1.9, program committee members. The committee decided to accept 16 papers.

We thank all authors and reviewers for their thorough work.

July 24, 2017
Dortmund

Johannes Fischer
Florian Kurpicz
Dominik Köppl

Program Committee

Ole Bergenholtz	TU Dortmund
Nils Bergmann	TU Dortmund
Nils Brinkmann	TU Dortmund
Patrick Dinklage	TU Dortmund
Johannes Fischer	TU Dortmund
Michael Dominik Görtz	TU Dortmund
Jens Knipper	TU Dortmund
Jakob Knorr	TU Dortmund
Daniel Korner	TU Dortmund
Florian Kurpicz	TU Dortmund
Dominik Köppl	TU Dortmund
Arno Pasternak	TU Dortmund
Fabian Pawlowski	TU Dortmund
Dennis Rohde	TU Dortmund
Lena Rolf	TU Dortmund
Tim Tannert	TU Dortmund
Uriel-Elias Wiebelitz	TU Dortmund
Roland Wyzgol	TU Dortmund
Jan-Gin Yuen	TU Dortmund
Jens Zentgraf	TU Dortmund

Table of Contents

Effiziente Datenstrukturen für Rank und Select auf Bitvektoren

Ole Bergenholtz[1]

1 **Lehrstuhl 11: Algorithm Engineering, Fakultät für Informatik, Technische Universität Dortmund, Deutschland**

Zusammenfassung

Rank und Select sind Operationen auf Bitvektoren. $\text{Rank}_1(x)$ berechnet dabei die Anzahl der Vorkommen gesetzter Bits bis zu der Stelle x, $\text{Select}_1(y)$ die Position des y-ten Vorkommens eines gesetzten Bits in dem Bitvektor B der Länge n. Mit diesen Operationen lassen sich verschiedene platzsparende Datenstrukturen realisieren. In dieser Seminararbeit werden bekannte statische Datenstrukturen für Rank und Select auf Bitvektoren vorgestellt, die mit $o(n)$ zusätzlichem Speicherplatz eine solche Anfrage in Laufzeit $\mathcal{O}(1)$ beantworten könne.

Durch Optimierungen hinsichtlich der Cache-Architektur und Instruktionssatz moderner Prozessoren lässt sich eine praktische Variante implementieren, welche gleichzeitig ähnliche Laufzeit und Speicherplatzverbrauch hat, wie die jeweiligen besten Varianten. Dafür wird der Bitvektor in Blöcke der Größe 512 Bit aufgeteilt, was der Größe einer Cache-Line entspricht. Der darüber aufgebaute Index besteht aus drei Ebenen. Die oberste Ebene speichert die absolute Anzahl gesetzter Bits aller vorhergehender Superblöcke der Größe 2^{32} Bits. In der zweiten Ebene werden die relativ zu Beginn des Superblocks gesetzten Bits jedes vierten Blocks gespeichert. Auf der dritten Ebene werden die Anzahl gesetzten Bits in den folgenden drei Blöcken gespeichert.

Die Datenstruktur für Select baut auf der Rank-Datenstruktur auf und wird unter gleichen Gesichtspunkten optimiert.

1998 ACM Subject Classification E.1 DATA STRUCTURES

Keywords and phrases bitvector, rank, select, data structure, optimization

Digital Object Identifier 10.4230/LIPIcs.BADS.2017.13

1 Einleitung

Viele platzsparende Datenstrukturen verwenden Rank- und Select-Datenstrukturen. Beispiele für diese platzsparenden Datenstrukturen sind der Wavelet-Tree [3] oder eine platzsparende Implementierung des Suffixbaums [7]. Operationen dieser platzsparenden Datenstrukturen werden reduziert auf die Operationen Rank und Select in Bitvektoren.

Datenstrukturen für Rank und Select ermöglichen es, entsprechende Anfragen effizient zu beantworten. $\text{Rank}(x)$ berechnet die Anzahl gesetzte Bits bis zur Position x im Bitvektor B der Länge n. $\text{Select}(y)$ berechnet dagegen, an welcher Position im Bitvektor sich das y-te gesetzte Bit befindet. Antworten auf diese Anfragen lassen sich mit einer simplen Iteration über den Bitvektor in linearer Zeit berechnen. Auf der anderen Seite ließen sich solche Anfragen in konstanter Laufzeit berechnen, wenn alle möglichen Anfragen im Voraus berechnet werden. Dies würde aber $\mathcal{O}(n \cdot \log(n))$ zusätzlichen Speicher belegen. Das Ziel von Rank- und Select-Datenstrukturen ist es, möglichst schnell solche Anfragen zu beantworten, mit möglichst wenig zusätzlichem Speicherbedarf. Die hier gezeigten theoretischen Datenstrukturen garantieren eine konstante Laufzeit, und belegen dabei $o(n)$ zusätzlichen Speicherplatz.

Diese theoretischen Schranken für Laufzeit und Speicher werden in dieser Arbeit für beide Datenstrukturen gezeigt.

Diese Arbeit stellt darüber hinaus eine Implementierung dieser Datenstrukturen vor, die zwar nicht die theoretisch optimale Laufzeit garantieren, aber durch Optimierungen eine geringe praktische Laufzeit haben. Darüber hinaus sind die verwendeten Datenstrukturen speicherplatzeffizient implementiert. Das Ergebnis sind Datenstrukturen für Rank- und Select-Anfragen, welche praktisch ähnliche Laufzeit und ähnlichen Speicherplatzbedarf wie die jeweils besten bestehenden Implementierungen bieten [9].

Da die hier gezeigte Implementierung der Select-Datenstruktur auf den Ergebnissen der Rank-Datenstruktur aufbaut, wird Rank-Datenstruktur zuerst beschrieben, gefolgt von einer Darstellung der Implementierung. Darauf folgt eine theoretische Betrachtung der Select-Datenstruktur, und wie die hier gezeigte Implementierung von dieser abweicht.

Bei der Implementierung der Datenstrukturen wurde das Design moderner Computer berücksichtigt. Moderne Rechner bieten verschiedene Operationen mit unterschiedlichen Laufzeiten. Theoretische Betrachtungen der Datenstrukturen lassen diese Unterschiede in der Laufzeit meist außen vor.

Die Operationen, die ein Computer ausführen kann, lassen sich in drei Kategorien unterteilen. Die schnellsten Operationen sind arithmetische Operationen, zum Beispiel das Addieren oder Multiplizieren von Werten, die im Speicher vorliegen. Etwas langsamer sind Operationen, die Bedingungen und Schleifen realisieren, sogenannte Branch-Operationen. Am teuersten sind Operationen, wenn Werte aus dem Speicher geladen werden müssen. Auf modernen Prozessoren gibt es mehrere Speicherebenen. Der schnellste Speicher ist der Cache. Auf Werte im Cache lässt sich sehr schnell zugreifen. Unter dem Cache liegt der Arbeitsspeicher. Programme agieren unter der Annahme, dass alle Werte aus dem Cache geladen werden. Da der Cache aber in der Größe sehr begrenzt ist, werden Werte aus dem Cache verdrängt. Dies bedeutet, dass diese Werte stattdessen in den Arbeitsspeicher abgelegt werden. Wenn ein Programm auf einen Wert zugreifen will, welcher nicht mehr im Cache vorliegt, tritt ein Cache-Miss auf, und das Betriebssystem muss den Wert, den das Programm braucht, aus dem sehr viel langsameren Arbeitsspeicher wieder nachladen.

Die Anzahl der Cache-Misses beeinflusst die Laufzeit von Programmen mit Datenstrukturen am meisten. Deswegen ist es sinnvoll, Programme dahingehend zu optimieren, dass möglichst wenig Cache-Misses auftreten.

2 Die Rank-Datenstruktur

Sei B ein Bitvektor der Länge n, und B_i das Bit an der Stelle i. Dann ist $\text{Rank}_1(x) = \sum_{0 \leq i < x} B_i$, wobei $0 \leq x \leq n-1$. Diese Definition zählt lediglich die Anzahl gesetzter Bits, und wird im Folgenden ohne Index verwendet. Wenn die Anzahl der ungesetzten Bits gezählt werden soll, reicht eine einfache Subtraktion des Rank-Werts von dem betrachteten Index, also $\text{Rank}_0(x) = x - \text{Rank}_1(x)$. Dies bedeutet, es reicht eine Datenstruktur, um beide möglichen Anfragen zu bedienen.

Die meisten Implementierungen für die Rank-Datenstruktur folgen der gleichen Methode. Der Bitvektor wird in Blöcke fester Größe eingeteilt. Innerhalb dieser Blöcke kann auf verschiedene Weisen effizient die Anzahl der gesetzten Bits gezählt werden. Oberhalb der Blöcke wird eine Index-Struktur aufgebaut, die Antworten über die Anzahl gesetzter Bits bis zu dem betrachteten Block liefert.

Im Folgenden wird die erste Variante beschrieben, die Rank in $o(n)$ Platz und $\mathcal{O}(1)$ Zeit berechnet hat [1]. Für asymptotische Betrachtungen wählen wir die Breite der Blöcke, in der

	Blockgröße	asymptotische Größe	Beschreibung
$P[V][i]$	$s = \lfloor \log(n)/2 \rfloor$	$\mathcal{O}(2^{\frac{\log(n)}{2}} \cdot s \cdot \log(s))$	Anzahl Bits in Block V bis Position i
L_1	s	$\mathcal{O}(n/s \cdot \log(s'))$	Anzahl Bits relativ zu Superblock
L_0	$s' = s^2 = \Theta(\log^2(n))$	$\mathcal{O}(n/s' \cdot \log(n))$	Anzahl Bits bis Beginn Superblock

Abbildung 1 Übersicht über die Elemente der klassischen Rank-Datenstruktur.

der Bitvektor aufgeteilt wird, mit $s = \lfloor \log(n)/2 \rfloor$. Innerhalb der Blöcke kann das Ergebnis einer Anfrage im Voraus berechnet werden und in einer Tabelle abgespeichert werden.

In dieser vorberechneten Tabelle werden alle möglichen Bit-Kombinationen indexiert, die in diesem Block auftreten können. Sei V der betrachtete Block der Länge s, und j die angefragte Position. Dann speichert die Tabelle P an Position $P[V][j]$ die Anzahl der 1-Bits von Anfang des Blocks V bis zur Position j. Insgesamt gibt es $2^s = 2^{\frac{\log(n)}{2}}$ mögliche unterschiedliche Blöcke der Länge s. Jede Zeile dieser Tabelle enthält s Einträge und benötigt $\log(s)$ Bits pro Eintrag. Der benötigte Speicherplatz berechnet sich also aus

$$(2^{\frac{\log(n)}{2}}) \cdot s \cdot \log(s) = \mathcal{O}(\sqrt{n} \log(n) \log(\log(n))) = o(n).$$

Die benötigte Zeit, um eine Rank-Anfrage innerhalb eines Blocks zu berechnen, ist $\mathcal{O}(1)$, da lediglich ein Eintrag in einer Tabelle abgerufen werden muss.

Zu diesem Wert innerhalb des Blocks muss nun lediglich die Anzahl der gesetzten Bits addiert werden, die sich in den vorherigen Blöcken befinden. Wenn lediglich die Laufzeit eine Rolle spielen würde, könnte dieser Wert einfach in einem Array gespeichert werden. Um eine bessere Schranke für den benötigten zusätzlichen Speicher zu finden, wird dieser Index aber aufgeteilt auf zwei Ebenen. Dafür werden Blocks zusammengefasst zu Superblocks. Die obere Ebene des Index speichert die Anzahl gesetzter Bits bis zum Beginn des Superblocks. Die zweite Ebene speichert lediglich die Anzahl gesetzter Bits relativ zum Anfang des Superblocks. Wenn diese beiden Ebenen summiert werden, erhält man die Anzahl der gesetzten Bits, die in den vorgehenden Blöcken aufgetreten sind.

Wir wählen für den Superblock die Länge $s' = s^2 = \Theta(\log^2(n))$. Das Array L_0 enthält an Position k die Anzahl der 1-Bits vom Start des Bitvektors B bis zum Beginn des k-ten Superblocks. Das Array L_1 enthält an Position l die Anzahl der 1-Bits des l-ten Blocks relativ zum Beginn des Superblocks, den diesen Block enthält. Für das Array L_0 ist der Speicherplatzverbrauch $n/s' \cdot \log(n) = \mathcal{O}(n/\log(n)) = \mathrm{o}(n)$. Das Array L_1 benötigt $n/s \cdot \log(s') = \mathcal{O}(\frac{n \log(\log(n))}{\log(n)}) = \mathrm{o}(n)$ Speicherplatz.

Eine Anfrage von Rank(x) lässt sich nun durch eine Anfrage auf diesem Index beantworten. Sei p die Anzahl der 1-Bits bis zum Beginn des Superblocks, in dem sich x befindet, q die Anzahl der 1-Bits bis zum Beginn des Blocks innerhalb des Superblocks, welches x enthält, und r die Anzahl der 1-Bits innerhalb des Blocks bis zur Position x. Nun lässt sich diese Anfrage berechnen mit Rank$(x) = p + q + r$. Die Werte p und q lassen sich durch L_0 und L_1 berechnen. Der Wert p entspricht $L_0[u']$, wobei $u' = \lfloor \frac{x}{s'} \rfloor$, und der Wert q entspricht $L_1[u]$, wobei $u = \lfloor \frac{x}{s} \rfloor$. Die Laufzeit dieser Datenstruktur für eine Anfrage ist $\mathcal{O}(1)$, da lediglich Werte in einem Array abgerufen werden.

▶ **Theorem 1.** *Eine Anfrage Rank$(x) = \sum_{0 \leq i < x} B_i$ auf einem Bitvektor B der Länge n lässt sich in Zeit $\mathcal{O}(1)$ und mit $o(n)$ zusätzlichen Speicherplatz beantworten.*

▶ **Beispiel 2.** Sei $B =$ `01001001101011010101` und $s = 3, s' = 9$. Der Index ist dargestellt in Abbildung 2. Beispiel für eine Anfrage ist Rank(13). Die Werte der Indexarrays berechnen

sich durch $u' = \lfloor 13/s' \rfloor = 1$ und $u = \lfloor 13/s \rfloor = 4$. Die Anzahl der gesetzten Bits bis Position 13 entspricht $L_0[u'] + L_1[u] + 2 = 4 + 1 + 2 = 7$.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
B	0	1	0	0	1	0	0	1	1	0	1	0	1	1	0	1	0	1	0	1
u	0			1			2			3			4			5			6	
L_1	0			1			2			0			1			3			0	
u'	0									1									2	
L_0	0									4									8	

Abbildung 2 Beispiel für den klassischen Aufbau einer Rank-Datenstruktur. L_0 und L_1 sind so eingerückt, dass sie den Anfängen der Blöcke entsprechen.

2.1 Praktische Optimierungen

Die obige Beschreibung der Rank-Datenstruktur zieht nicht den Aufbau moderner Computer in Betrachtung. Ein moderner Computer schafft in der Zeit, die es dauert, eine Cache-Line aus dem Speicher zu laden, viele arithmetische Instruktion. Befindet sich die entsprechende Cache-Line nicht im Speicher, tritt ein Cache-Miss auf, und die Cache-Line muss aus dem langsameren Speicher nachgeladen werden. Deshalb ist es effizienter, mit mehreren arithmetischen Instruktionen auf einem Wert im Cache die gesetzten Bits zu zählen, statt in einer vorberechneten Tabelle, die aus dem Speicher geladen werden muss, um den entsprechenden Wert zu berechnen.

2.1.1 Berechnung gesetzter Bits innerhalb eines Blocks

Die Befehlssatzerweiterung SSE4.2, kurz für Streaming SIMD Extensions, enthält die Instruktion `POPCNT`. Diese ermöglicht es, mit einer Instruktion die Anzahl gesetzter Bits in einem 64-Bit Wert zu zählen [4]. Ältere Erweiterungen enthalten ähnliche Instruktionen, die es dem Compiler ermöglichen, eine effiziente Implementierung einer Funktion zu generieren, die den gleichen Effekt hat wie die `POPCNT`-Instruktion [5]. Dies nennt sich dann Broadword-Programmierung.

Als Größe für einen Block wird 512 Bits gewählt. Dies hat den Vorteil, dass es der typischen Größe eine Cache-Line von 64 Bytes der Intel-CPU-Baureihen entspricht. Dass der Bitvektor in Blöcke unterteilt wird, die der Cache-Größe entsprechen, nennt sich Cache-Aligned. Bei einem Zugriff auf eine Position innerhalb eines solchen Block tritt somit lediglich ein einziger Cache-Miss auf. Keins der benachbarten Cache-Lines, und damit Blöcke, muss nachgeladen werden. Durch den Verzicht auf die vorberechnete Tabelle wird zudem Speicherplatz gespart.

Um die Anzahl gesetzter Bits bis zu einer bestimmten Position zu berechnen, wird der Block stückweise mit der `POPCNT`-Instruktion gezählt. Die einzelnen Stücke werden danach addiert und das Ergebnis wird ausgegeben. Als kleine Laufzeitverbesserung kann jedes 64-Bit Stück, welches gezählt werden muss, parallel berechnet werden.

2.1.2 Speicheroptimierung des Index

Für die Größe eines Superblock wird 2^{32} Bits gewählt. Durch die obere Grenze der Größe des Bitvektors von 2^{64} Bits, benötigen die einzelnen Elemente des Arrays L_0 maximal 64 Bit. Für ein Bitvektor der Länge 2^{34}, etwas mehr als zwei Gigabyte, sind lediglich $2^{34}/2^{32} = 2^2 = 4$

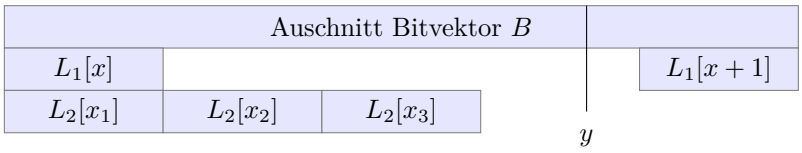


Abbildung 3 Schematischer Ausschnitt für L_1 und L_2 der poppy-Rank-Datenstruktur.

	Blockgröße	Bitbreite	Beschreibung
L_0	2^{32} Bits	64 Bits	Anzahl Bits bis Superblock
L_1	$4 \cdot 512$ Bits	32 Bits	Anzahl Bits relativ von Superblock Beginn
L_2	512 Bits	10 Bits	Anzahl Bits in den drei Blöcken nach Block in L_1

Abbildung 4 Übersicht über die Elemente der optimierten poppy-Rank-Datenstruktur.

Elemente in Array L_0 nötig. Damit belegt das Array 256 Bits im Speicher und kann im Cache vorgehalten werden. Für das Array L_1 reichen Elemente mit 32 Bits, um die Anzahl gesetzter Bits zu zählen, da in einem Superblock maximal 2^{32} gesetzte Bits vorkommen können.

Um den Speicherplatz weiter zu optimieren, wird das Array L_1 aufgeteilt in zwei Ebenen. Array L_1 speichert lediglich die Anzahl gesetzter Bits jedes vierten Blocks. In L_2 wird die Anzahl gesetzter Bits innerhalb des ersten, zweiten und dritten Blocks ausgehend vom Block, welches in L_1 vorgehalten wird, gespeichert. Da dies maximal 512 gesetzte Bits pro Block sind, reicht eine Größe von 10 Bits pro Eintrag im Array L_2. Das Schema einer solchen Folge von vier Blöcken, die durch L_1 und L_2 indiziert werden, ist dargestellt in Abbildung 3. Um die Anzahl gesetzter Bits bis zum Beginn des nicht indizierten Blocks, in dem sich y in der Abbildung befindet, zu berechnen, müssen die Werte aus $L_1[x], L_2[x_1], L_2[x_2]$ und $L_2[x_3]$ aufsummiert werden. Statt 128 Bits für vier Blöcke, die bei nur einer Ebene für L_1 nötig wären, wird mit diesem aufgespaltenen Index lediglich $32 + 3 \cdot 10 = 62 < 64$ Bits belegt, und somit die Hälfte des Speichers benutzt. Für vier Blöcke belegt der Index mit den Arrays L_1 und L_2 also $(2048 + 64/2048) = 3{,}125\%$ zusätzlichen Speicherplatz zum Bitvektor B. Dies war das Ziel der Implementierung.

Damit lediglich ein Cache-Miss auftritt, wenn auf den Index zugegriffen wird, werden L_1 und L_2 zusammen in einem Element abgespeichert. Die ersten 32 Bit eines Element dieses kombinierten $L_{1/2}$-Arrays entsprechen dem Wert des Arrays L_1. Die folgenden $3 \cdot 10$ Bit entsprechen den Werten des L_2. Damit das Array Cache-Aligned ist, enthalten die letzten 2 Bit eines Elements keine Informationen. Ein Element des Arrays $L_{1/2}$ ist also 64 Bit groß. Abbildung 4 gibt eine Übersicht über die verwendeten Arrays in der beschriebenen Index-Datenstruktur.

2.2 Vergleichbare Implementierungen

Wie oben angesprochen, gibt es verschiedene Methoden, die Anzahl gesetzter Bits innerhalb eines Blocks zu zählen. Die Anzahl der Stufen und die Breite eine Blocks lassen sich modifizieren, um Abwägungen zwischen Speicherplatzbedarf und maximaler Größe des Bitvektors zu treffen. Der Name für den Ansatz, der in dieser Arbeit beschrieben wird, ist poppy. Abbildung 5 gibt einen Überblick über Implementierungen der Rank-Datenstruktur.

Aus den dargestellten Ergebnissen leitet sich ab, dass das Ziel der Datenstruktur ist, einen Index zu implementieren, der Bitvektoren der Größe 2^{64} mit zusätzlichen Speicherplatzverbrauch von 3,125% zu implementieren. Gleichzeitig sollte die Laufzeit geringer als

Ansatz	Blockgröße	Methode	Index	Speicherplatz	Länge
Klassisch [2]	$\lfloor \log(n)/2 \rfloor$ Bits	Tabelle	2	66,85%	2^{32}
RG 37 [2]	256 Bits	Tabelle	2	37,5%	2^{32}
rank 9 [8]	64 Bits	Broadword	2	25%	2^{64}
combined-sampling [6]	1024 Bits	Tabelle	1	3,125%	2^{32}
poppy [9]	512 Bits	popcount	3	3,125%	2^{64}

Abbildung 5 Übersicht über verschiedene Ansätze, die die Rank-Datenstruktur implementieren. Die Spalte Methode beschreibt die Berechnungsweise der Anzahl gesetzter Bits innerhalb eines Blocks. Die Spalte Index beschreibt, wie viele Ebenen der Index über den Blocks aufweist. Die Spalte Speicherplatz beschreibt, wie viel zusätzlicher Speicherplatz für den Index relativ zur Größe des Bitvektors benötigt wird. Die Spalte Länge beschreibt die maximale Länge des Bitvektors, über den der Index aufgebaut werden kann.

die schnellsten Ansätzen bleiben.

3 Die Select-Datenstruktur

Sei B wieder ein Bitvektor der Länge n, und B_i das Bit an der Stelle i. Entsprechend ist $\text{Select}_1(y) = \min\{k | \text{Rank}(k) = y\}$ mit $1 \leq y \leq \text{Rank}(n)$.

Wenn $B_y = 1$ dann ist $\text{Rank}(\text{Select}_1(y)) = y$. Im Gegensatz zu Rank kann man nicht mit einer Select-Anfrage für gesetzte Bits die Select-Anfrage für ungesetzte Bits ausrechnen. Deswegen muss die gleiche Datenstruktur sowohl für 0 als auch für 1 erstellt werden. Im Folgenden wird der Index der Select-Operation weggelassen und bezieht sich immer auf gesetzte Bits. Die Datenstruktur für ungesetzte Bits funktioniert analog.

Ein weiterer Unterschied von Select zu Rank ist, dass nicht im Vorhinein bekannt ist, welche Position im Bitvektor benötigt wird. Beispielsweise kann sich das 15-te gesetzte Bit am Anfang des Bitevktors befinden, aber auch an einer Position sehr viel später. Deswegen ist es meist nicht effizient, einfach den Bitvektor in Blöcke zu unterteilen und Antworten bezogen auf Blöcke abzuspeichern.

Herangehensweisen für Select lassen sich in zwei Methoden unterteilen.

Die erste Methode zum Finden der Position ist Rank-basiert. Mit Hilfe der Datenstruktur für Rank und binärer Suche wird der Block bestimmt, in dem das gesuchte Bit vorliegt. Der Vorteil hiervon ist, dass dieser wenig zusätzlichen Speicherplatz benötigt, wenn die Rank-Datenstruktur wiederverwendet werden kann. Die Laufzeit hiervon ist dann $\mathcal{O}(\log(n) \cdot t_{\text{rank}}$, wobei t_{rank} der Laufzeit entspricht, die Rank benötigt. In der oben gezeigten Implementierung entspricht die Laufzeit von Select also $\mathcal{O}(\log(n))$.

Die andere Methode ist positionsbasiert. Die Ergebnisse für Select-Anfragen werden im Vorhinein berechnet. Dabei wird ein Array angelegt, welches Antworten auf bestimmte Anfragen sampled. Dies bedeutet, dass nur jede k-te Position eines gesetzten Bits in einem Array gespeichert wird. Um nun $\text{Select}(y)$ zu berechnen, muss das j gefunden werden, so dass $j \cdot k \leq y$. Wenn diese Position j gefunden wurde, wird mit weiteren Ebenen im Index die genaue Position bestimmt.

3.1 Klassische Variante

Im Folgenden wird eine Variante beschrieben, die Select in $o(n)$ Platz und $\mathcal{O}(1)$ Zeit berechnet [1]. Diese Variante ist positionsbasiert.

Für asymptotische Betrachtungen wird jede Antwort im Voraus berechnet, die ein Vielfaches von $k = \lfloor \log^2(n) \rfloor$ ist. Diese Werte werden im Array N gespeichert. An Position t im Array ist die Position des $t \cdot k$-ten gesetzten Bits gespeichert, also $\text{Select}(t \cdot k) = N[t]$. Dieses Array benötigt $\mathcal{O}((n/\log^2(n)) \cdot \log(n)) = \mathcal{O}(n/\log(n)) = o(n)$ zusätzlichen Speicherplatz.

Darüber hinaus zerteilt dieses Sampling-Array den Bitvektor in Blöcke verschiedener Längen, abhängig von der Dichte der gesetzten Bits in dem Abschnitt, der von den gespeicherten Werten beschrieben wird. Es sei angemerkt, dass die Anzahl der gesetzten Bits in diesen Intervallen immer gleich ist, und zwar k.

Die genaue Position des y-ten gesetzten Bits wird mit einer zweiten Ebene im Index bestimmt. Die zweite Ebene des Index wird je nach Länge dieses Intervalls auf unterschiedliche Weise implementiert. Wenn die Länge dieses Intervalls größer als k^2 ist, handelt es sich um einen langen Block, ansonsten um einen Miniblock.

Handelt es sich um einen langen Block, kann die Antwort für jede Anfrage explizit abgespeichert werden. Dieses Array belegt insgesamt $k \cdot \log(n)$ Speicherplatz. Von diesen langen Blöcken können maximal n/k^2 im Bitvektor auftreten (wenn jeder Block ein langer Block wäre). Insgesamt wird so also $\mathcal{O}((n/k^2) \cdot k \cdot \log(n)) = \mathcal{O}((n/\log^4(n)) \cdot \log^2(n) \cdot \log(n)) = O(n/\log n) = o(n)$ zusätzlicher Speicherplatz benötigt.

Wenn es sich bei dem durch das Array N beschriebenen Intervall um einen Miniblock handelt, wird eine weitere Ebene des Sampling-Array erstellt. Dieses Array N' speichert die Antwort auf jede Anfrage, welche ein Vielfaches von $k' = \lfloor \log^2(k) \rfloor$ ist. Die Werte sind dabei relativ zum Beginn des Miniblocks. Dieses Array benötigt maximal $\mathcal{O}((\frac{n}{k'}) \log(k^2) = \mathcal{O}((\frac{n}{\log^2(\log(n))}) \log(\log(n))) = o(n)$.

Ähnlich zu den Blöcken , welche durch N beschrieben werden, wird auch in der zweiten Ebene der Blocktyp nach Blockgröße unterschieden. Handelt es sich um einen langen Miniblock, ist dieser länger als $\frac{\sqrt{k}}{2}$. Für solche langen Miniblöcke werden die Antworten wieder explizit abgespeichert, relativ zum Beginn des Blocks. Auch diese verbrauchen wieder $\mathcal{O}((\frac{n}{k'}) \log(k^2)) = \mathcal{O}((\frac{n}{\log^2(\log(n))}) \log(\log(n))) = o(n)$ zusätzlichen Speicherplatz.

Sollte es sich um einen kurzen Miniblock handeln, wird ähnlich zur Rank-Datenstruktur eine vorberechnete Tabelle verwendet. Da diese Blöcke maximal $\frac{\sqrt{k}}{2} = \Theta(\log(n))$ lang sind, benötigt auch diese Tabelle lediglich $o(n)$ zusätzlichen Speicherplatz.

Eine Anfrage $\text{Select}(y)$ lässt sich durch eine Anfrage an das Array N an Position $j = \lfloor y/k \rfloor$ beantworten. Je nach Größe des Intervalls $N[j+1] - N[j]$ kann die Antwort direkt als expliziter Wert vorliegen, oder das Sampling-Array N' an Position $j' = \lfloor y/k' \rfloor$ kommt zum Tragen. Je nach Größe des Intervalls in Array N' liegt die Antwort explizit in einem Array vor, oder eine globale vorberechnete Tabelle liefert die gesuchte Antwort innerhalb des Blocks. Da lediglich Werte in Arrays oder Tabellen nachgeschaut werden, ist die Laufzeit dieses Algorithmus $\mathcal{O}(1)$.

▶ **Theorem 3.** *Eine Anfrage $\text{Select}(y) = \min\{k | \text{Rank}(k) = y\}$ mit $1 \leq y \leq \text{Rank}(n)$ auf einem Bitvektor B der Länge n lässt sich in Zeit $\mathcal{O}(1)$ und mit $o(n)$ zusätzlichen Speicherplatz beantworten.*

▶ **Beispiel 4.** Sei $B =$ `01001001101010110101011`. Die Idee des Sampling-Arrays N ist in Abbildung 6 dargestellt. Dabei wird jedes gesetzte Bit mit einem Vielfachen von $k = 3$ in dem Array gespeichert. Beispielsweise befindet sich das 6. gesetzte Bit an Position $j = \lfloor 6/3 \rfloor = 2$. Der Wert an Position 2 im Array N ist $N[2] = 12$. Das heißt, das sechste gesetzte Bit befindet sich an Position 12. Das Sampling-Array N' speichert alle Werte für $k' = 1$, relativ zum Beginn des Miniblocks, welches durch N bestimmt wird. Das 5. gesetzte Bit befindet sich an Position $N[1] + N'[5] = 7 + 3 = 10$.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
B	0	1	0	0	1	0	0	1	1	0	1	0	1	1	0	1	0	1	0	1
j	0							1					2					3		
N	0							7					12					17		
j'	0	1			2			3	4		5		6	7		8		9		
N'	0	1			4			0	1		3		0	1		3		0		

Abbildung 6 Beispiel für das Sampling-Array N einer Select-Datenstruktur. Der Wert k beträgt 3. Es handelt sich bei allen Intervallen um kurze Blöcke. Um das Beispiel anschaulich zu gestalten, sei $k' = 1$.

3.2 Praktische Optimierungen

Für die Select-Datenstruktur wird die Rank-Datenstruktur wiederverwendet. Für jeden Superblock der Größe 2^{64} Bits wird ein Sampling-Array angelegt. Dabei wird die Position jedes 8192. gesetzten Bits innerhalb dieses Superblocks abgespeichert, ähnlich zu dem Algorithmus combined-sampling [6].

Ein Aufruf von Select(y) nutzt dann dieses Sampling-Array und die Rank-Datenstruktur. Als erstes wird mit binärer Suche auf dem Array L_0 der Superblock bestimmt, in dem das gesuchte Bit liegen muss. Innerhalb dieses Superblocks gibt das Sampling-Array darüber Auskunft, an welcher Position j im Array L_1 das $j \cdot 8192 \leq y$ liegt. Darauf wird dann die Position im Array L_1 gesucht, die der gesuchten Position am nächsten liegt. Darauf folgt dann die Suche in dem Array L_2. Dabei wird einfach so lange über L_1 iteriert, bis der Block der Länge 2048 gefunden ist, in dem das gesuchte gesetzte Bit liegen muss. Mithilfe des Array L_2 wird dann die Position des gesuchten Bit auf ein Block der Länge 512 eingegrenzt.

Sobald dieser Block gefunden ist, in dem sich das gewünschte Bit aufhält, wird mit der `POPCNT`-Instruktion das 64 Bit Intervall lokalisiert, das die Position enthält. Innerhalb dieses Intervalls kommt dann ein Broadword-Algorithmus zum Einsatz, der die genaue Position bestimmt.

Eine kleine Optimierung zur Laufzeitverbesserung ist, dass bei der Lokalisierung der Position im Array L_1 beachtet wird, dass jeder Eintrag maximal 2048 Bits enthalten kann. Dies bedeutet, dass die ersten Einträge übersprungen werden können, welche auf keinen Fall die gewünschte Position enthalten können.

3.3 Vergleichbare Implementierungen

Abbildung 7 gibt eine Überblick über bestehende Ansätze, die Select-Datenstruktur zu implementieren. Der in dieser Arbeit beschriebene Ansatz nennt sich cs-poppy. Die beste Laufzeit von bestehenden Algorithmen bietet combined-sampling.

4 Evaluation

Die beschriebenen effizienteren Datenstrukturen sind nicht unter informationstechnischen Gesichtspunkten entwickelt worden, sondern unter praktischen. Dies bedeutet, dass beispielsweise die beschriebene Select-Datenstruktur keine theoretische Laufzeit von $\mathcal{O}(1)$ liefert. Auch der benötigte Speicherplatz ist nicht durch $o(n)$ beschränkt, sondern linear in der Größe der Eingabe, wenn auch mit einem sehr kleinen Faktor.

Bei einer praktischen Evaluierung aber wird klar, dass der klassische Ansatz keine Chance hat, mit diesem praktisch optimierten Ansatz verglichen zu werden. Während die Laufzeit

Ansatz	Blockgröße	Methode	Speicherplatz	Länge
Clark [1]	$\lceil \log(n) \rceil$ Bits	Tabelle	60%	2^{32}
Hinted bsearch [8]	64 Bits	Broadword	$\sim$ 37,38%	2^{64}
select 9 [8]	64 Bits	Broadword	50%	2^{64}
simple select [8]	64 Bits	Broadword	9,01% – 45,94%	2^{64}
combined-sampling [6]	1024 Bits	Tabelle und Scan	$\sim$ 0,39%	2^{32}
cs-poppy [9]	512 Bits	`POPCNT` und Broadword	$\sim$ 0,39%	2^{64}

Abbildung 7 Übersicht über verschiedene Ansätze, die die Select-Datenstruktur implementieren. Die Spalte Blockgröße beschreibt die gewählte Blockgröße. Die Spalte Methode beschreibt die Berechnungsweise der Position des gesuchten gesetzten Bits innerhalb eines Blocks. Die Spalte Speicherplatz beschreibt, wie viel zusätzlicher Speicherplatz für den Index relativ zur Länge des Bitvektors benötigt wird. Dabei wird nicht der Speicherplatz in Betracht gezogen, der für die Rank-Datenstruktur verwendet wird. Die Spalte Länge beschreibt die maximale Länge des Bitvektors, über den der Index aufgebaut werden kann.

des beschriebenen Ansatz ähnlich oder gleich mit dem klassischen Ansatz ist, ist der zusätzliche Speicherplatzbedarf um einiges geringer zum klassischen Ansatz. Vor allem bei kleinen Bitvektoren benötigt die Tabelle, welche alle Ergebnisse vorberechnet, ein Vielfaches der Eingabegröße zusätzlichen Speicherplatz [6]. Auch bei langen Bitvektoren ist der benötigte zusätzliche Speicherplatz um einiges höher, als der von der hier beschriebenen Variante.

Literatur

1 Clark, David: *Compact Pat Trees*. PhD Thesis, University of Waterloo, 1997.

2 González, Rodrigo, Szymon Grabowski, Veli Mäkinen und Gonzalo Navarro: *Practical Implementation of Rank and Select Queries*. In: *Poster Proceedings Volume of the 4th Workshop on Efficient and Experimental Algorithms (WEA)*, Seiten 27–38, 2005.

3 Grossi, Roberto, Ankur Gupta und Jeffrey Scott Vitter: *High-Order Entropy-Compressed Text Indexes*. In: *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, Seiten 841–850. Society for Industrial and Applied Mathematics, 2003.

4 Intel Corporation: *SSE4 Programming Reference*, 2007.

5 Ladra, Susana, Oscar Pedreira, Jose Duato und Nieves R Brisaboa: *Exploiting SIMD Instructions in Current Processors to Improve Classical String Algorithms*. In: *East European Conference on Advances in Databases and Information Systems*, Seiten 254–267. Springer, 2012.

6 Navarro, Gonzalo und Eliana Providel: *Fast, Small, Simple Rank/Select on Bitmaps*. In: *Proceedings of the 11th International Conference on Experimental Algorithms*, Band 7276 der Reihe *Lecture Notes in Computer Science*, Seiten 295–306. Springer, 2012.

7 Ohlebusch, Enno, Johannes Fischer und Simon Gog: *CST++*. In: *Proceedings of the 17th International Conference on String Processing and Information Retrieval (SPIRE)*, Seiten 322–333. Springer, 2010.

8 Vigna, Sebastiano: *Broadword Implementation of Rank/Select Queries*. In: *Proceedings of the 7th international conference on Experimental Algorithms*, Seiten 154–168. Springer, 2008.

9 Zhou, Dong, David G. Andersen und Michael Kaminsky: *Space-Efficient, High-Performance Rank and Select Structures on Uncompressed Bit Sequences*. In: *Proceedings of the 12th International Conference on Experimental Algorithms*, Band 7933 der Reihe *Lecture Notes in Computer Science*, Seiten 151–163. Springer, 2013.

Fast Compressed Tries through Path Decompositions

Nils Bergmann[1]

1 **Technische Universität Dortmund, Dortmund, Germany**
nils2.bergmann@udo.edu

Zusammenfassung

Tries, auch als Präfixbäume bekannt, sind Suchbäume, welche zur Speicherung von Strings verwendet werden. Bei ihnen wird jeder Kante ein Zeichen zugeordnet, sodass ein Pfad von der Wurzel zu einem Blatt genau einem der gespeicherten Strings entspricht. Tries sind keine balancierte Datenstruktur und können im schlechtesten Fall einen großen Speicherbedarf haben, was zusätzlich einen dementsprechenden schlechten Einfluss auf Zugriffszeiten haben kann. Mit Hilfe von Pfadzerlegung können komprimierte Tries erstellt werden, welche einen geringeren Speicherbedarf haben und, je nach Anwendung und Darstellung, einen gleich schnellen oder sogar schnelleren Zugriff auf Informationen ermöglichen. Es werden zwei Anwendungsfälle für dieses Verfahren betrachtet:

- String Dictionaries, welche die Funktionen Lookup(s) und Access(i) unterstützen.
- Monotone Hashfunktionen für Strings, also Hashfunktionen, welche die lexikographische Reihenfolge bewahren.

Für die Anwendungsfälle werden Darstellungen entwickelt, welche die jeweiligen Anforderungen und notwendigen Operationen unterstützen und dabei einen möglichst geringen Speicherbedarf haben. Der Vergleich mit gängigen Implementierungen lässt schließen, dass die durch Pfadzerlegung erzeugten Strukturen deutliche Verbesserung im Bezug auf Laufzeit der Operationen und Speicherbedarf aufweisen.

1998 ACM Subject Classification E.1 DATA STRUCTURES – please refer to `http://www.acm.org/about/class/ccs98-html`

Keywords and phrases Path Decomposition, Compression, Tries, Data Structure, Strings

Digital Object Identifier 10.4230/LIPIcs.BADS.2017.19

1 Einleitung / Motivation

Tries finden in vielen verschiedenen Bereichen Verwendung. Typische Anwendungsgebiete sind zum Beispiel in der Textverarbeitung, *Pattern Matching* oder *Data Mining*. Durch ihre simple Funktionsweise können sie sehr gut angepasst und erweitert werden, was sie in verschiedenen Anwendungsgebieten attraktiv macht. Sie sind jedoch durch ihre Baumstruktur im Bezug auf Speicherbedarf und Performance stark von den in ihnen gespeicherten Zeichenketten abhängig. Ziel dieser Arbeit ist es zu präsentieren wie mit der Hilfe von Pfadzerlegungen dieses Problem bewältigt werden kann. Hierfür werden zunächst die Grundlagen aufgearbeitet, dann zwei Anwendungsfälle betrachtet und zum Schluss ein Vergleich mit gängigen Lösungen der Anwendungsfälle durchgeführt.

2 Grundlagen

In diesem Abschnitt werden die benötigten Grundlagen eingeführt, welche die Basis für spätere Implementierungen bilden.

2.1 Tries

Ein Trie, auch Präfixbaum, ist eine auf Suchbäumen basierende Datenstruktur, in welcher Zeichenketten gespeichert werden. Sei $S = \{ S_1, \ldots, S_n \}$ eine Menge aus n Zeichnketten über ein Alphabet Σ mit σ verschiedenen Zeichen, dann ist ein Trie über diese Menge S ein Baum T, der aus den Mengen V (den Knoten) und E (den Kanten) besteht. Jeder Knoten in aus T entspricht einem Präfix einer oder mehrerer der Zeichenketten aus S. Hierfür erhält jede Kante $e \in E$ ein Zeichen aus Σ als Label, sodass für ein beliebiges Präfix p eines Strings $s \in S$ genau ein Knoten $v \in V$ existiert, für den gilt: Die Konkatenation der Kantenlabels auf dem Pfad von der Wurzel zu v bildet p. Somit existiert also auch für jede Zeichenkette $S_i \in S$ genau ein Knoten in T (vgl. Abbildung 1). Hier ist zu beachten, dass jedes Blatt von T einer Zeichenkette aus S entspricht. außerdem gilt: Wenn für zwei Zeichenketten $S_1 \in S$ und $S_2 \in S$ gilt, dass S_1 ein Präfix von S_2 ist, dann ist der Knoten, welcher S_1 entspricht, kein Blatt, sondern ein Knoten auf dem Pfad zwischen der Wurzel und S_2. Solche Knoten müssen gesondert markiert werden. Es findet bereits ein gewisses

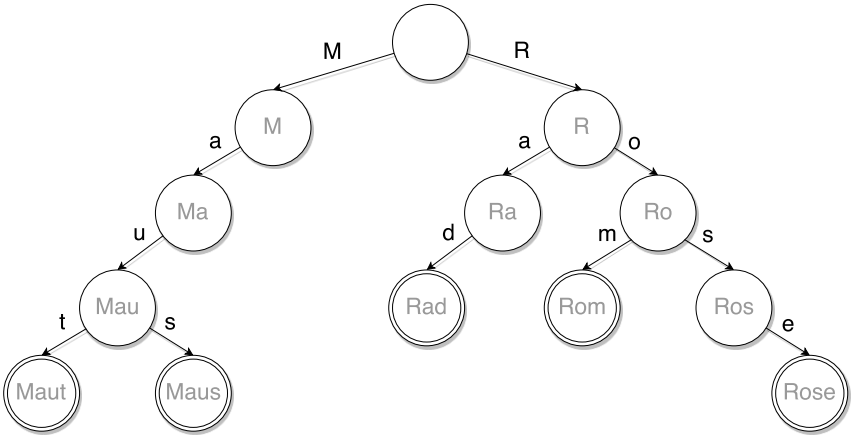


Abbildung 1 Ein typischer Trie, mit $S = \{ Maus, Maut, Rad, Rom, Rose \}$. Die hier grau eingetragenen Labels der Knoten werden **nicht** gespeichert und wurden hier nur zu Demonstrationszwecken der entsprechenden Zeichenketten ergänzt.

Maß an Kompression statt, da bei Zeichenketten mit gemeinsamen Präfixen das Präfix nur ein Mal gespeichert werden muss, allerdings ist dies auch die einzige Kompression, die stattfindet. Desweiteren können durch ihre Baumstruktur die Daten in ihnen, im Vergleich zu kompakteren Datenstrukturen, weit verbreitet im Speicher liegen, was bedeutet dass sehr viele *Cache Misses* auftreten können. Tries sind außerdem keine balancierte Datenstruktur, da ihre Höhe stark von den gespeicherten Zeichenketten abhängt, was einen zusätzlichen negativen Effekt auf den Speicherbedarf hat. Das Ziel ist es also einen Weg zu finden den Speicherbedarf zu verringern und schnelle Zugriffszeiten sicher zu stellen.

2.2 Kompakte Tries

Kompakte Tries sind, wie der Name besagt, eine kompakte Darstellung von Tries. Betrachtet man zum Beispiel in der Abbildung 1 den Subtrie der Wurzel, sieht man ein typisches Problem von Tries: Knoten, die genau ein Kind haben. Hier existieren keine Abzweigungen und es wird so so sowohl die Höhe des Baumes als auch den Speicherbedarf erhöht, ohne dass zusätzliche Informationen enthalten sind. In kompakten Tries werden Knoten und ihre dazugehörigen Kanten dieser Art zusammengefüg. In der Abbildung 2 ist der Trie aus der

Abbildung 1 in der kompakten Form dargestellt. Ein kompakter Trie kann aus einer Menge an Strings über einen rekursiven Algorithmus erzeugt werden [5, S. 3]:

- Der kompakte Trie eines einzelnen Strings ist ein einzelner Knoten, dessen Bezeichner der String selbst ist.
- Bei einer gegebenen Menge S mit $S \neq \emptyset$ wird die Wurzel des Baumes mit dem längsten gemeinsamen Präfix α aller Strings aus S als Bezeichner erstellt (Wobei $\alpha = \emptyset$ gelten kann, sollten nicht alle Strings aus S ein gemeinsames Präfix haben, vgl. Abbildung 2). Als nächstes wird für jedes Zeichen b, für den $S_b = \{ \beta \mid \alpha b \beta \in S \} \neq \emptyset$ gilt, der kompakte Trie über die Menge S_b als Kind an die Wurzel angehangen. b ist der *Branching Character* und wird als Bezeichner für die Kante zwischen der Wurzel und dem S_b-Trie verwendet. Die Länge von α wird *Skip* genannt und wird mit δ bezeichnet.

Wie in normalen Tries entspricht jede Zeichenkette $S_i \in S$ einem Pfad in dem Baum. Hier werden allerdings, anstatt die Kantenlabels auf dem Pfad zu konkatenieren, abwechselnd die Kantenlabels und die Knotenlabels konkateniert um S_i zu erhalten. Im Beispiel in der Abbildung 2 entspricht die Zeichenkette *Rose* dem Pfad $\emptyset \xrightarrow{R} \emptyset \xrightarrow{o} \emptyset \xrightarrow{s} e$. *Tries* bezeichnet ab jetzt, sofern nicht anders explizit erwähnt, *kompakte* Tries, da sie äquivalent zu den bekannten Tries sind und die verkleinerte Form spätere Änderungen vereinfacht.

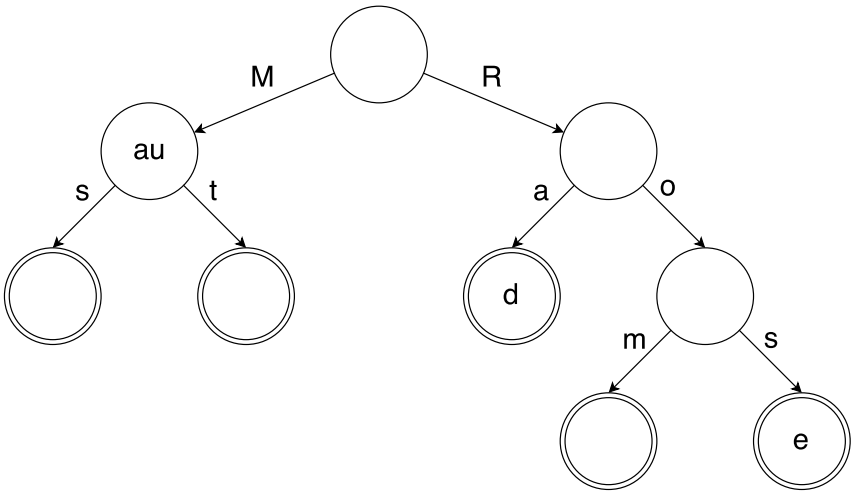


Abbildung 2 Der Trie aus Abbildung 1 als kompakter Trie. Zu beachten ist hier die stark verringerte Anzahl der Knoten und Kanten im Vergleich zu Abbildung 1

.

2.3 Pfadzerlegung

Eine Pfadzerlegung, oder auch *path decomposition*, T^c eines Tries T zu einer Zeichenkette S ist ein Baum, in dem jeder Knoten einem Pfad in T entspricht. T^c wird rekursiv erzeugt, indem man zu Beginn einen Pfad von der Wurzel von T zu einem Blatt wählt und ihn als Wurzel von T^c festlegt. Danach wird dieses Verfahren rekursiv auf die Subtries, die als Kinder von diesem Pfad abgehen, angewendet. Die entstehenden Bäume werden von links aus als Kinder dieser Wurzel eingefügt und die Kanten zu ihnen mit den entsprechenden *Branching Characters* zu dem entsprechenden Kindern des aktuellen Knotens aus T beschriftet. Zu beachten ist, dass T^c genau $|S|$ Knoten hat, da jeder Knoten einem Knoten-zu-Blatt-Pfad in T entspricht. Außerdem kann die Höhe von T^c nicht größer als die von T sein.

Das Auswahlverfahren der Reihenfolge der Pfade kann hier beliebig gewählt werden. Zwei solche Verfahren sind *leftmost path*, bei welchem immer das linkeste Kind zuerst gewählt wird, und *heavy path*, bei welchem immer das Kind mit dem meisten Blättern in seinem Subtrie gewählt wird (Bei gleich vielen Blättern wird beliebig gewählt). Wenn T lexikographisch geordnet ist, ist T^c mit dem *leftmost path*-Verfahren auch lexikographisch geordnet. Deshalb wird es auch *lexikographische Pfadzerlegung* genannt. Wenn *heavy path* gewählt wird, ist die Höhe von T^c maximal $O(log|S|)$, eine solche Zerlegung wird auch *zentroide Pfadzerlegung* genannt. **Beispiel:** Führt man eine lexikographische Pfadzerlegung auf den Trie in der Abbildung 2 aus, erhält man den in Abbildung 3 dargestellten Trie. Hierfür beginnt man mit dem Pfad $\emptyset \xrightarrow{M} au \xrightarrow{s} \emptyset$. Dieser wird als v_1 gewählt. Von ihm gehen an zwei Stellen Subtries ab, der Erste (v_2) geht am Knoten, welcher dem Präfix *Mau* entspricht, mit der Kante t ab, der Zweite (v_3) an der Wurzel mit der Kante R. v_2 entspricht einem Pfad, welcher nur ein Blatt ohne Label enthält. v_3 entspricht dem Pfad $\emptyset \xrightarrow{a} d$ von welchem an der Wurzel ein Subtrie v_4 mit der Kante o abgeht. v_4 entspricht dem Pfad $\emptyset \xrightarrow{m} \emptyset$ von welchem ein letzter Subtrie v_5 an der Wurzel mit der Kante s abgeht, welcher lediglich ein Blatt mit dem Label e ist.

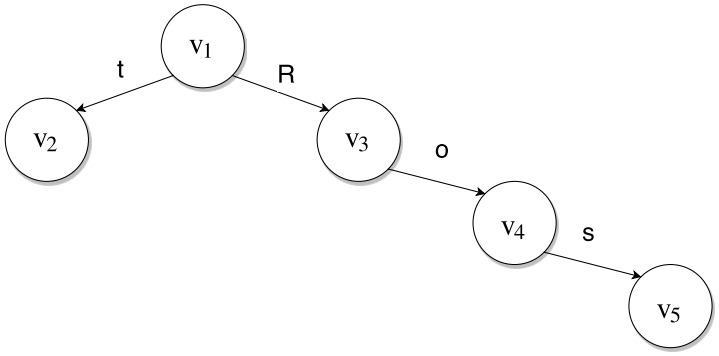
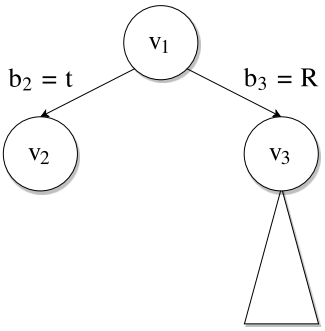


Abbildung 3 Der Trie aus Abbildung 1 und 2 nach einer lexikographischen Pfadzerlegung.

2.4 Mittel zur späteren Darstellung

Folgende Funktionen oder Datenstrukturen werden in späteren Abschnitten verwendet, um die entwickelten Datenstrukturen zu kodieren, speichern oder Operationen auf ihnen zu definieren: Für einen beliebigen Bitvektor[1] X werden folgende Funktionen definiert: $Rank_b(i)$ liefert die Häufigkeit des Bits b in den ersten i Stellen von X. $Select_b(i)$ liefert die Position des i-ten Vorkommens von b in X. Beide Operationen können nach [3] in konstanter Zeit durchgeführt werden, wenn im Bitvektor o(n) Bits zusätzliche redundante Informationen gespeichert werden. *Beispiel*: Sei $X = 1101010110$, dann ist $Rank_0(5) = 2$ und $Select_0(3) = 7$. Die *Elias-Fano*-Zeichenkodierung ist eine Zeichenkodierung, welche eine monoton steigende Folge von m Integern aus $[0, n)$ in $2m + m\lceil log\frac{n}{m}\rceil + o(m)$ Bits repräsentieren kann und dabei in konstanter Zeit auf den i-ten Integer zugreifen kann. *Balanced parentheses* (BP) ist eine Sequenz von ausgeglichenen Klammern, in der jede offene Klammer „(“ eine entsprechende schließende Klammer „)“ hat. Die Sequenz kann als ein Bitvektor dargestellt werden, indem man „(“ durch 1 und „)“ durch 0 darstellt. Die Operationen *FindOpen* und *FindClose*, welche den entsprechenden Partner einer Klammer liefern, können in nach [6] in konstanter Zeit

[1] Eine Zeichenkette mit dem Alphabet $\{0, 1\}$



Der Knoten v_1 entspricht dem Pfad $\emptyset \xrightarrow{M} au \xrightarrow{s} \emptyset$ in dem Trie in der Abbildung 2. Es befinden sich an zwei Stellen an dem Pfad abgehende Subtries. (wie in dem Beispiel im Abschnitt 2.3 erläutert). Somit folgt:

$$\begin{aligned} BP_{v_1} &= (() \\ B_{v_1} &= b_3 b_2 = Rt \\ L_{v_1} &= \emptyset\,\mathbf{1}\,M\,au\,\mathbf{1}\,s\,\emptyset = \mathbf{1}Mau\mathbf{1}s \end{aligned}$$

Abbildung 4 Das Ergebnis der lexikographischen Pfadzerlegung aus Abbildung 3 mit der entsprechenden Kodierung der Wurzel v_1 nach 3.1.1

durchgeführt werden, wenn für n Klammerpaare o(n) Bits ergänzt werden. Außerdem wird in [6] die Operation *enclose(i)* eingeführt, welche für eine offene Klammer o an der Stelle i die Position der offenen Klammer o_2 des Klammerpaares findet, die mit ihrer entsprechenden geschlossenen Klammer g_2 o und ihre entsprechende geschlossenene Klammer g umschließt. Für eine Sequenz in der Form $\ldots o_2 \ldots o \ldots g \ldots g_2 \ldots$, wobei o sich an der Position i befindet, liefert *enclose(i)* also die Position von o_2.

3 Anwendungen

Dieser Abschnitt behandelt mögliche Anwendungen der in Abschnitt 2 eingeführten Grundlagen. Diese Datenstrukturen und ihre Definitionen stammen aus [5].

3.1 String Dictionaries

Ein *String Dictionary* ist eine Datenstruktur, in der eine Menge von Strings, S, gespeichert wird. Sie besitzt zwei Funktionen: $Lookup(s)$ liefert, falls $s \in S$, einen Index $\in [0, |S|)$ und -1 wenn $s \notin S$. $Access(i)$ liefert die Zeichenkette mit dem Index i. Hier gilt: $s \in S \Rightarrow Access(Lookup(s)) = s$ [5, S. 5]. Da es sich hier um eine Datenstruktur handelt, in welcher Zeichenketten verwaltet werden, ist eine mögliche Implementierung die Verwendung eines Tries. Außerdem ist es möglich hier die Vorteile einer Pfadzerlegung aus zu nutzen. Wählt man die *lexikographische Pfadzerlegung* stellt man sicher, dass die Indizes, welche durch *Lookup* geliefert werden, lexikographisch geordnet sind. Wählt man die *zentroide Pfadzerlegung* erhält man die Sicherheit, dass die Höhe logarithmisch ist, was einen postiven Effekt auf den Speicherbedarf hat.

3.1.1 Repräsentation

Das Ergebnis einer Pfadzerlegung T^c eines solchen Tries T wird in drei Sequenzen repräsentiert. Hier werden zuerst die Teilsequenzen für jeden Knoten erzeugt (vgl. Abbildung 4) und dann der Tiefe nach konkateniert. Sei v ein beliebiger Knoten aus T^c:

- Der Bitvektor BP_v enthält n offene Klammern (gefolgt von einer geschlossenen Klammer), wobei n dem *Grad* von v entspricht. Ist v ein Blatt, ist demnach $BP_v =)$.
- Das Array B_v enthält die *Branching Characters* der ausgehenden Kanten zu Kindern des Knotens. Sie werden in umgekehrter Reihenfolge, also von rechts nach links, notiert. Dadurch entsteht eine eins-zu-eins Beziehung zwischen den offenen Klammern in BP und den *Branching Characters* in B_v durch die spätere Kodierung von BP_v.

- Die Sequenz L_v enthält das Label des Knotens. Das Alphabet Σ wird um $|\Sigma|-1$ zusätzliche Zeichen ergänzt, sodass $\Sigma' = \Sigma \cup \{ \mathbf{1}, \mathbf{2}, \ldots, |\Sigma| - 1 \}$ entsteht. Diese Sonderzeichen werden später verwendet um die Grade der Knoten (und somit abgehende Subtries auf dem Pfad) zu kodieren. Da der maximale Grad eines Knotens in T $|\Sigma| - 1$ ist, werden $|\Sigma| - 1$ Sonderzeichen benötigt. Zu beachten ist hier, dass das Sonderzeichen $\mathbf{i}$ dem $Grad - 1$ entspricht.
  v entspricht einem Pfad in T in der Form $w_1 \xrightarrow{c_1} w_2, \ldots, w_{k-1}, \xrightarrow{c_{k-1}}, w_k$, wobei w_i die Knoten und c_i die Kantenbezeichnungen sind. Dieser Pfad wird in L_v kodiert, indem $L_v = \alpha_{w1}\mathbf{d}_{w1}c_1 \ldots \alpha_{wk-1}\mathbf{d}_{wk-1}c_{k-1}\alpha_{wk}$ wobei α_{wi} das Label des Knoten w_i und $\mathbf{d}_{wi}$ das neue Sonderzeichen in Σ'ist, das $Grad_{wi} - 1$ repräsentiert.
  Ein Label wird also kodiert, indem abwechselnd das Label und der *Branching Character* der ausgehenden Kante eines jeden Knoten auf dem Pfad mit der Anzahl der anderen Subtries, welche an diesem Knoten als Kinder hängen, in der Sequenz L_v gespeichert werden.

Dadurch, dass die Teilsequenzen der Tiefe nach konkateniert werden, entstehen $BP = BP_{v_1}BP_{v_2}\ldots BP_{v_n}$, $B = B_{v_1}B_{v_2}\ldots B_{v_n}$ und $L = L_{v_1}L_{v_2}\ldots L_{v_n}$, diese drei Sequenzen repräsentieren zusammen T^c. Zu beachten ist hier, dass BP eine Sequenz von ausgeglichenen Klammern ist und auch eine Darstellung der Struktur des Baumes in der *depth-first unary degree sequence* (DFUDS)-Repräsenation nach [2], wenn man eine führende (ergänzt. Da die Teilsequenzen L_v von L eine variable Größe haben, wird die Endposition aller L_v in L in einer monoton steigenden Sequenz gespeichert, diese Sequenz kann dann mit der *Elias-Fano*-Zeichkodierung kodiert werden, was einen Zugriff auf die Endpositionen eines beliebigen L_{vi} und somit auch auf die Anfangspositionen in konstanter Zeit ermöglicht.
Beispiel: Dieses Beispiel basiert auf den Abbildungen 2 und 3. Wenn man das dort aufgeführte Beispiel nach dieser Kodierung kodiert, erhält man folgende Sequenzen:

		v_1	v_2	v_3	v_4	v_5
BP	= (	(( )	)	( )	( )	)
B	=	Rt	$\emptyset$	o	s	$\emptyset$
L	=	$\mathbf{1}Mau\mathbf{1}s$	$\emptyset$	$\mathbf{1}ad$	$\mathbf{1}m$	e

3.1.2 Implementierung der Operationen

Ein *String Dictionary* besitzt wie erwähnt zwei Operationen, *Lookup* und *Access*.
Lookup(s): Man erzeugt einen Akkumulator $m = 0$. Ausgehend von der Wurzel werden gleichzeitig das Label L_v und s durchgegangen, wenn das aktuelle Zeichen im Label ein Sonderzeichen ist ($\mathbf{1}$, $\mathbf{2}$, ...) entspricht dies der Anzahl der möglichen Abzweigungen von dem Pfad in T. Dieser Wert wird auf m addiert. Wenn das aktuelle Zeichen ein normales Zeichen ist, wird überprüft ob es mit dem aktuellen Zeichen in s übereinstimmt, wenn ja, wird mit dem nächsten Zeichen fortgefahren. Wenn sie unterschiedlich sind, gibt es zwei mögliche Fälle: Das vorherige Zeichen war ein Sonderzeichen d und es wurde die falsche Abzweigung genommen. Wenn dies nicht der Fall ist, gilt $s \notin S$ und es wird -1 zurück gegeben. Die Differenz zwischen m und $m + d$ entspricht der Anzahl der anderen Kinder des Knotens v, also liegt der korrekte *branching character* (wenn es ihn gibt) in B_v zwischen m und $m + d$. B entspricht, wie oben erwähnt, den offenen Klammern in BP, also kann mit $Rank_($ der aktuellen Position in BP der Anfang von B_v in B gefunden werden. Die Zeichen in B_v sind sortiert, womit eine binäre Suche möglich ist. Wird kein passendes Zeichen gefunden, existiert kein passender Pfad und s befindet sich somit nicht in S. Wenn eine passende Kante gefunden wird, wird die Suche mit dem Suffix von s in dem Kind fortgeführt. Wenn der

String auf diese Weise komplett durchlaufen ist und ein Blatt erreicht wurde, gilt $s \in S$ und der Index der) in BP, welcher dem gefundenen Knoten entspricht, wird zurückgegeben. Der Index kann mit $Rank_)$ der aktuellen Position in BP gefunden werden.
Access(i): $Select_)(i)$ liefert die Startposition in BP, danach wird rekursiv rückwärts der Pfad rekonstruiert, indem man zu dem Elternknoten springt[2] und alle nicht-Sonderzeichen im Label L_v bis zu der Stelle, die der Abzweigung zum Ursprungskind entspricht, vor den Rückgabestring hängt und dann die Rekursion weiter führt, bis die Wurzel von T^c erreicht wurde. Da die Rekursion von unten nach oben durchgeführt wird, ist, wenn man sich in einem Knoten v befindet, bekannt von welchem Kind man v erreicht hat. Das bedeutet, dass auch bekannt ist, bis zu welcher Stelle L_v gelesen werden muss. Außerdem ist auch bekannt welcher *Branching Character* aus B_v benötigt wird und somit an der Kante zwischen den beiden Knoten liegt.

Beispiele: Sei T^c gleich dem Baum in Abbildung 3 mit der in 3.1.1 vorgestellten Repräsentation, dann gilt:
Lookup(Rad): $m = 0$ wird erzeugt und man beginnt in der Wurzel, v_1. Ł$_{v1} = \mathbf{1}Mau\mathbf{1}s$ hat an erster Stelle ein Sonderzeichen (**1**), also wird m auf $m + 1 = 1$ gesetzt. Das nächste Zeichen ist $M \neq R$. Da das vorherige Zeichen ein Sonderzeichen $d = \mathbf{1}$ war, muss das korrekte Zeichen (R) zwischen $m = 1$ und $m + d = 2$ in B_{v1} liegen, was zutrifft. Jetzt wird die Suche mit dem Suffix von s, $s' = ad$ in dem Knoten v_3 fortgesetzt. Hier wird in $L_{v3} = \mathbf{1}ad$ zuerst die **1** gelesen und der Akkumulator erhöht, dann wird mit dem nächsten Zeichen fortgefahren, da $a = a$ wird der Zeiger in s auf d vorgeschoben. Das nächste Zeichen in L_{v3} ist auch ein d, womit s nun vollständig durchgangen ist, ist $s \in S$ und die entsprechende Position des Knotens kann mit einem $Rank_)$ zurückgegeben werden.
Access(3): $Select_)(3)$ liefert die Position der dritten) in BP und somit das Ende von BP_{v3} und somit den Knoten v_3, deshalb werden die Nicht-Sonderzeichen aus L_{v3} vor den Rückgabestring gestellt, somit erhält man das Zwischenergebnis $s = ad$. Dann wird ein Sprung zu dem Elternknoten von v_3, v_1 durchgeführt (zum Beispiel durch einen Aufruf von *enclose* auf die Startposition von BP_{v3} in BP, was die Startposition des Elternknotens von v_3 und somit die Startposition von BP_{v1} liefert) und der ensprechende *Branching Character* von v_1 zu v_3 vor s gestellt. Damit erhält man $s = Rad$. Als nächstes wird $L_{v1} = \mathbf{1}Mau\mathbf{1}s$ bis zu der Stelle, an der die Abzweigung zu v_3 befindet durchgangen. Diese befindet sich an der ersten Stelle, womit die Rekursion abgeschlossen ist und $s = Rad$ als Ergebnis zurück gegeben wird.

3.1.3 Komplexität der Operationen

Für einen String mit der Länge p und einen pfadzerlegten Baum T^c mit der Höhe h ist die Anzahl der Operationen von *Lookup* in $O(p + h\log|\Sigma|)$, da die einen String der Länge p maximal p Sonderzeichen gelesen werden und für jeden Knoten v aus dem Durchlauf eine sequentielle Suche über L_v und eine binäre Suche über B_v durchgeführt wird. Für Access ist die Anzahl der Operationen durch $O(p)$ beschänkt, da keine binäre Suche durchgeführt wird und $p \geq h$. Die Anzahl der wahlfreien Speicherzugriffe ist bei beiden Operationen von $O(h)$ beschränkt, wobei h die Höhe von T^c ist.

[2] BP ist nach [2] eine DFUDS-Repräsentation und es kann so mit *enclose* der Elternknoten des Knotens in konstanter Zeit gefunden werden.

3.2 Monotoner Minimaler Perfekter Hash für Strings

Eine minimale perfekte Hashfunktion bildet eine Menge von Strings S bijektiv auf $[0, |S|)$ ab. Eine *monotone* Hashfunktion behält dabei die lexikographische Reihenfolge bei. Zu beachten ist, dass eine Datenstruktur für eine Hashfunktion nicht unbedingt S speichern muss, sondern für Strings, die nicht in S liegen, beliebige Werte liefern kann. Diese Funktion eines Hashes verhält sich analog zum *Lookup* eines *String Dictionary.* Ähnlich wie bei der in dem Abschnitt 3.1 vorgestellten Struktur bietet sich deshalb auch hier eine auf Tries basierte Struktur an, wobei hier im Vergleich zu einem *String Dictionary* weniger Informationen gespeichert werden müssen, da lediglich *Lookup(s)* unterstützt werden muss. Diese Struktur nach [1] wird *Hollow Trie* genannt, wobei hier in einem binären Trie lediglich die Topologie des Baumes und die *Skips* gespeichert werden. Die *Branching Characters* der Kanten in diesem binären Baum sind 0 für ein linkes Kind und 1 für ein rechtes Kind. Um dann den Hashwert eines Strings s zu bestimmen, wird s bitweise durchgegangen und dabei eine blinde Suche in dem Baum durchgeführt. Hierbei wird beginnend mit der Wurzel das aktuelle Bit in s mit den *Branching Characters* der ausgehenden Kanten des Knotens verglichen und die Suche in dem entsprechenden Subtrie fortgeführt. Wenn $s \in S$ erreicht man so das entsprechende Blatt, wenn $s \notin S$ ist es das längste Präfix von s, welches in S enthalten ist. Ein großes Problem dieser *Hollow Tries* ist die Höhe, da das Alphabet Σ von S in ein binäres Alphabet umgewandelt werden muss, kann die Höhe des Tries im Vergleich zu einem Trie über Σ bis zu $O(log|\Sigma|)$ mal so groß sein. Aber eine zentroide Pfadzerlegung würde die lexikographische Reihenfolge nicht bewahren, welche für die Monotonie der Hashfunktion notwendig ist. Deshalb wird in [5] ein Pfadzerlegungsverfahren entwickelt, welches ausnutzt, dass ein binäres Alphabet verwendet wird und dass *Access* nicht benötigt wird. Zuerst werden die Subtries, welche vom Pfad abgehen lexikographisch sortiert, sodass die Subtries links vom Pfad von oben nach unten und die Rechts von unten nach oben angeordnet sind. Wenn man jetzt das *heavy path*-Verfahren so ändert, dass bei einem Gleichstand der Tiefe nicht beliebig gewählt wird, sondern bevorzugt linke Kinder gewählt werden, erhält man das *left-biased heavy path*-Verfahren. Dieses Verfahren ermöglicht es eine Pfadzerlegung eines *Hollow Tries* zu erstellen, die sowohl die logarithmische Höhe, als auch die Beibehaltung der lexikographischen Reihenfolge für *Lookup* ermöglicht.

3.2.1 Repräsentation

Der Trie wird in drei Sequenzen repräsentiert, BP, L^{high} und L^{low}. Der Bitvektor BP ist identisch zu BP in 3.1.1. L^{high} und L^{low} stellen die Labels der Knoten dar und sind speicherausgerichtete Bitvektoren. Zuerst werden alle *Skips* um 1 erhöht um 0 aus der Domäne zu entfernen. Dann werden die binären Repräsenationen der *Skips* mit den Pfadrichtungen verschachtelt und in L^{low} konkateniert. in L^{high} werden für jeden Knoten Nullfolgen mit der Länge der binären Repräsenationen der *Skips* gefolgt von einer 1 gespeichert. Das bedeutet, dass die Endpunkte der einzelnen (*skip*, Richtung)-Paare, welche in L^{low} konkateniert sind, passend zu den 1-en in L^{high} ausgerichtet sind. Nach [4] ist es dadurch möglich, über ein *Select Directory* über L^{high} wahlfreie Zugriffe auf die Paare in L^{low} durchzuführen.

3.2.2 Implementierung von Lookup(s)

Der Durchlauf wird ähnlich wie im Abschnitt 4 durchgeführt. Während des Durchlaufes wird die Anzahl der linken und rechten Kinder, welche passiert werden, gespeichert. Wenn das aktuelle Zeichen in s sich von der gewählten Kante unterscheidet, wird der Durchlauf im entsprechenden anderen Kind fortgesetzt. Durch die Sortierung der Kinder folgt: Wenn der

Fehler in ein linkes Kind führt, ist dessen Index die Anzahl der passierten linken Kinder. Wenn er in ein rechtes Kind führt ist es der Grad des Knotens minus der Anzahl der passierten rechten Kinder. Wenn s komplett durchgangen ist, ist nicht die Position des aktuellen Knotens in der Tiefensuche der gesuchte Wert, da alle durchgangenen Knoten, welche linke Kinder waren, zwar in der Tiefensuche vor dem aktuellen Knoten auftreten, aber sich lexikographisch hinter ihm befinden. Zusätzlich sind alle Strings in den links abgehenden Subtries des entsprechenden Pfades lexikographisch vor dem aktuellen String, sollte der gefundene Knoten kein Blatt sein. Deshalb muss eine Modifikation durchgeführt werden, bei welcher die Anzahl der passierten linken Kinder vom Index abgezogen wird. Die Anzahl der Blätter der links abgehenden Subtries kann bestimmt werden, indem man mit *FindClose* zum ersten rechten Kind springt, die Anzahl der übersprungenen Knoten ist dann äquivalent zu der Anzahl der Blätter in den linken Subtries.

3.2.3 Komplexität von Lookup

Die Laufzeit von *Lookup* verhält sich ähnlich zu der in Abschnitt 2.3 verwendeten Variante von *Lookup*. Für einen pfadzerlegten *Hollow Trie* T^c mit der Höhe h können während des Durchlaufes mit einem String s nicht mehr als $|s|$ Sprünge durchgeführt werden. Außerdem wird keine binäre Suche durchgeführt, was zur Folge hat, dass die Anzahl der Operationen $O(\min(|s|, h))$ ist und die Anzahl der wahlfreien Speicherzugriffe $O(h)$

4 Auswertung

Die hier aufgeführten Beobachtungen basieren auf den Ergebnissen der in [5, S. 15-18] durchgeführten Analyse.

4.1 Verfahren

Die in Abschnitt 3 eingeführten Datenstrukturen werden mit gängigen Implementierungen für die entsprechenden Probleme im Bezug auf Faktoren wie Speicherbedarf und Performance verglichen. Zum Vergleich werden fünf Datensätze mit jeweils 2.5 bis 114.3 Millionen Strings verwendet. Diese Datensätze sind `enwiki-titles` (alle Seitentitel der englischen Version von Wikipedia, 8.5 Millionen Strings), `aol-queries` (10.2 Millionen Suchanfragen an AOL, die 2006 veröffentlicht wurden), `uk-2002` (18.5 Millionen URLs), `webbase-2001` (114.3 Millionen URLs) und ein synthetischer Datensatz `synthetic`, der so entwickelt wurde, dass er möglichst unbalancierte Tries erzeugt, aber dennoch stark komprimierbar ist und 2.5 Millionen Strings enthält. In dem synthetischen Datensatz sind zufällig erzeugte Strings mit dem Muster $d^i c^j b^t X$, wobei $i, j \in [0, 500)$ und $t \in [0, 10)$. X ist ein 100 Zeichen langes Suffix, welches 100 unterschiedliche Zeichen enthält und für jeden String identisch ist.

4.2 Beobachtungen

- **Durchschnittliche Höhe:** In typischen realen Anwendungsfällen hat die zentroide Pfadzerlegung einen enormen Effekt auf die Höhe. Im Vergleich zu kompakten Tries sinkt hier die Höhe um einen Faktor von zwei bis drei. Bei *Hollow Tries* ist die Differenz noch stärker, da sie durch die binäre Darstellung zuerst enorm wachsen (Zur Erinnerung: Die Höhe erhöht sich um $\log|\Sigma|$), aber durch die Pfadzerlegung wird ein Baum erzeugt, der sogar sehr viel kleiner als $\log|S|$ ist. Auch bei einer lexikographischen Pfadzerlegung lassen sich durchweg Verbesserungen beobachten.

- **String Dictionaries:** Im Vergleich zu gängigen Implementierungen einer solchen Datenstruktur schneidet die in 3.1 eingeführte Datenstruktur sowohl im Bezug auf Speicherbedarf als auch auf Performance sehr gut ab. Die zentroide Variante hat in 4/5 Testfällen den geringsten Speicherbedarf und belegt in einem Testfall knapp den zweiten Platz. Außerdem hat sie die schnellsten *Lookup*-Aufrufe. Allerdings ist die Geschwindigkeit von *Access* langsamer als einzelne andere Lösungen. Die lexikographische Variante ist in realen Testfällen ähnlich schnell wie die zentroide Variante, aber sie schneidet bei dem synthetischen Datensatz sehr schlecht ab, da der entstehende Baum sehr unbalanciert ist.
- **Monotone Hashes:** Der pfadzerlegte *Hollow Trie*, welcher in 3.2 eingeführt wurde, ist zwei bis fünf mal so schnell wie ein normaler *Hollow Trie*. Er ist außerdem für den synthetischen Datensatz um einen Faktor von 13 bis zu 41 schneller als ein normaler *Hollow Trie*. Der Grund ist hier, dass *Hollow Tries* durch ihre Unbalanciertheit bei diesem Datensatz enorm leiden, während diese Pfadzerlegung es ermöglicht einen kompakten Baum zu erzeugen.

5 Fazit

Pfadzerlegung ist ein nützliches Mittel um die Probleme von Tries auszugleichen, gerade die Unbalanciertheit von Tries kann, je nach Anwendungsfall, einen enormen Einfluss auf die Performance und den Speicherbedarf haben. Die entstehenden Strukturen sind sehr viel kompakter als gängige Lösungen und haben dennoch vergleichbare oder bessere Zugriffszeiten. Sie sind außerdem verlässlicher, da sie nicht so stark vom Datensatz abhängig sind. Durch geeignete Repräsentationen können schnelle Operationen ermöglicht werden, während der Speicherbedarf minimiert wird. Ein kleiner Nachteil ist die nötige Abwägung, ob die lexikographische oder die zentroide Pfadzerlegung am besten geeignet ist, idealerweise würde es ein Verfahren geben, welches die Vorteile der beiden kombiniert, dies ist zwar bei der Verwendung als *Hash* der Fall, ist aber bei der Verwendung als *String Dictionary* in der Form nicht möglich.

Literatur

1 Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. Theory and practice of monotone minimal perfect hashing. *Journal of Experimental Algorithmics (JEA)*, 16:3–2, 2011.

2 David Benoit, Erik D. Demaine, J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, Dec 2005. URL: `https://doi.org/10.1007/s00453-004-1146-6`, `doi:10.1007/s00453-004-1146-6`.

3 Francisco Claude and Gonzalo Navarro. Fast and compact web graph representations. *ACM Trans. Web*, 4(4):16:1–16:31, September 2010. URL: `http://doi.acm.org/10.1145/1841909.1841913`, `doi:10.1145/1841909.1841913`.

4 P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, March 1975. `doi:10.1109/TIT.1975.1055349`.

5 Roberto Grossi and Giuseppe Ottaviano. Fast compressed tries through path decompositions. *Journal of Experimental Algorithmics (JEA)*, 19:3–4, 2015.

6 J Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses, static trees and planar graphs. In *Foundations of Computer Science, 1997. Proceedings., 38th Annual Symposium on*, pages 118–126. IEEE, 1997.

Quake Heaps

Nils Brinkmann[1]

1 **Technische Universität Dortmund, Fachbereich Informatik**
`nils.brinkmann@tu-dortmund.de`, **Matr. Nr. 165867**

Zusammenfassung

Quake Heaps [1] von Timothy M. Chan sind Prioritätswarteschlangen, die sich durch eine besonders einfache Struktur und gleiche asymptotische Laufzeit in den drei Methoden `insert` ($O(1)$ amortisiert), `delete-min` ($O(\log n)$ amortisiert) und `decrease-key` ($O(1)$ amortisiert) wie die Fibonacci Heaps [2] auszeichnen. Anwendung finden Prioritätswarteschlangen beispielsweise in Greedy-Algorithmen wie dem Algorithmus von Dijkstra für kürzeste Pfade.

Ein Quake Heap besteht aus einer Menge von Tournament Trees, bei denen jeder Schlüssel in einem Blatt gespeichert wird und die Wurzel jeweils dem Minimum ihrer Blätter entspricht. Die Einfachheit der Quake Heaps liegt darin, dass lediglich eine einzige globale Invariante existiert, bei deren Nichterfüllung ein teilweises Verwerfen und Neuaufbauen des Quake Heaps stattfindet, was der Datenstruktur ihren Namen verleiht. Für alle Operationen wird stets die minimal erforderliche Anzahl an Schritten durchgeführt, um diese zu erfüllen. Dieses Vorgehen ermöglicht laut Chan ein besonders leichtes Verständnis der Funktionsweise von den Operationen und eine einfache Laufzeitanalyse der Methoden ohne Fallunterscheidungen.

1998 ACM Subject Classification E.1 Data structures

Keywords and phrases Quake Heaps, Datenstruktur, Tournament Trees

Digital Object Identifier 10.4230/LIPIcs.BADS.2017.7

1 Einführung

Eine Prioritätswarteschlange ist eine abstrakte Datenstruktur, die das Einfügen von einem Element mit seinem Schlüssel, das Entfernen und Ausgeben des Elements mit kleinstem Schlüssel und das Vermindern eines Schlüssels von einem Element unterstützen. Die entsprechenden Methoden werden im Folgenden `insert`, `delete-min` und `decrease-key` genannt.

Prioritätswarteschlangen werden für verschiedene algorithmische Probleme verwendet. Beispielsweise können Elemente durch Einfügen in eine Prioritätswarteschlange und anschließendem Ausgeben sortiert werden. Auch viele Greedy-Algorithmen verwenden Prioritätswarteschlangen, um immer das gewinnbringendste Element als nächstes zu betrachten. Ein Beispiel dafür ist der Dijkstra-Algorithmus für das kürzeste Pfade-Problem, in welchem stets der Knoten als nächstes betrachtet wird, der die geringste Distanz zum Startknoten aufweist (und nicht bereits betrachtet wurde).

Quake Heaps von Timothy M. Chan [1] sind eine vom Aufbau der Datenstruktur und Komplexität der Analyse besonders einfache Implementation von Prioritätswarteschlangen. Ihre Methoden besitzen eine gute amortisierte Laufzeit (`insert` $O(1)$, `delete-min` $O(\log\ n)$ amortisiert und `decrease-key` $O(1)$ amortisiert) und im Gegensatz zu anderen Varianten, wie zum Beispiel den Fibonacci Heaps [2], erfordern sie weder Fallunterscheidungen bei der Implementation noch eine komplizierte Laufzeitanalyse der Methoden.

2nd Symposium on Breakthroughs in Advanced Data Structures.
Editor: Prof. Dr. Johannes Fischer; Article No. 7; pp. 7:1–7:10

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

2 Funktionsweise

Bei einem Quake Heap handelt es sich um eine Menge von Bäumen, für die eine einzige globale Invariante aufrecht erhalten wird. Diese Menge von Bäumen wird im Folgenden mit S bezeichnet. Eine Besonderheit der Quake Heaps stellt die Art der Bäume, genannt Tournament Trees dar, die zum Speichern der Schlüssel verwendet werden.

2.1 Tournament Trees

Tournament Trees sind eine Art von Binärbäumen, bei denen der Schlüssel eines jeden inneren Knoten dem Minimum der Werte seiner Kinder entspricht, es gilt also die Heap Bedingung. Durch diese Struktur befindet sich jeder Schlüssel in einem der Blätter. Jeder Pfad von einem Knoten u zu einem beliebigen Blatt hat dieselbe Länge, welche als Höhe von u bezeichnet wird. Jeder innere Knoten besitzt den Grad eins oder zwei. In Abbildung 1 ist ein Beispiel für einen Tournament Tree zu sehen.

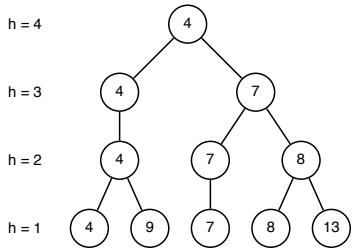


Abbildung 1 Ein Beispiel für einen Tournament Tree. Alle Schlüssel stehen in den Blättern und der Elternknoten entspricht immer dem Minimum seiner Kinder.

Tournament Trees unterstützen zwei Operationen, die in konstanter Zeit durchgeführt werden können. Zum einen können zwei Tournament Trees u und v gleicher Höhe h mithilfe der `link` Operation zu einem neuen Tournament Tree der Höhe $h+1$ vereinigt werden. Dafür erhält eine neue Wurzel Verweise auf u und v als ihre beiden Kinder. Das Vorgehen ist in Abbildung 2 beispielhaft für zwei Knoten der Höhe $h = 3$ dargestellt.

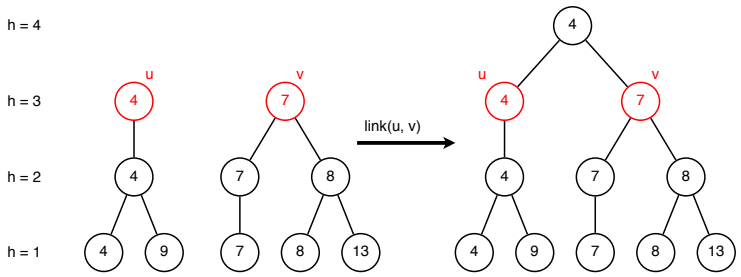


Abbildung 2 Zwei Tournament Trees der Höhe drei werden mithilfe der `link`-Operation zu einem Baum der Höhe vier vereinigt. Die Wurzel erhält den kleineren der beiden Schlüssel von seinen Kindern.

Die zweite Methode `cut` trennt einen Teilbaum v von seinem Elternknoten u ab, wenn der Schlüssel von v nicht dem Schlüssel von u entspricht (v also nicht der Kindknoten mit minimalem Schlüssel ist). Dies reduziert den Grad von Knoten u auf eins. Ein Beispiel hierfür ist in Abbildung 3 zu sehen.

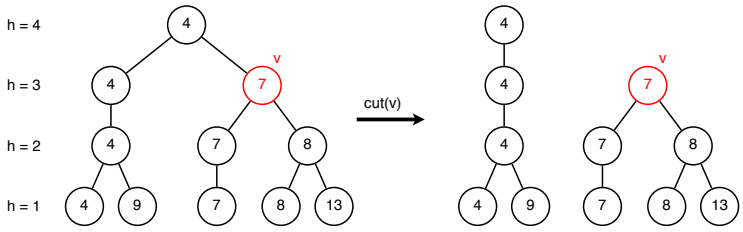


Abbildung 3 Ein Teilbaum, der nicht dem Schlüssel seines Elters entspricht, wird mithilfe der `cut`-Operation von seinem Elter abgetrennt und wird zu einer neuen Wurzel.

2.2 Quake Heaps

In den Quake Heaps werden Tournament Trees verschiedener Höhe für das Speichern der Schlüssel verwendet. Timothy Chan fasst das Vorgehen für die Operationen auf seiner Datenstruktur mit „be lazy during updates, and just rebuild when the structure gets ‚bad'" zusammen [1]. Alle Operationen auf der Quake Heap Datenstrukur werden mit möglichst wenig Berechnungsaufwand durchgeführt, ohne dabei im Detail beispielsweise darauf zu achten, ob einzelne Tournament Trees degenerieren, also ob viele Knoten mit Grad eins enthalten sind. Jede Operation muss lediglich die Gültigkeit der Invariante

$$n_{i+1} \leq \alpha n_i \qquad \forall i \geq 1 \tag{1}$$

sicherstellen. n_i steht dabei für die Anzahl der Knoten auf der Höhe i in *allen* Tournament Trees in der Menge S. $\alpha \in (\frac{1}{2}, 1)$ ist eine im angegebenen Intervall frei wählbare Konstante, welche die Häufigkeit des Neuaufbaus steuert, im Fall dass die Datenstruktur zu stark degeneriert ist. Ein Wert für α nahe der unteren Schranke $\frac{1}{2}$ bedingt einen häufigeren Neuaufbau, wohingegen ein Wert für α nahe 1 eine stärkere Degenerierung der Tournament Trees zulässt. Ein Neuaufbau wird gegebenenfalls in der Methode `delete-min` durchgeführt und im dazugehörigen Abschnitt näher erläutert.

Die Anzahl der Elternknoten n_{i+1} auf Höhe $i+1$, gezählt über alle Tournament Trees des Quake Heaps, darf nach der Invariante 1 maximal einem Bruchteil (abhängig von α) der Anzahl ihrer Kinder n_i entsprechen. Dadurch ist die maximale Höhe eines Tournament Trees im Quake Heap durch $\log_{\frac{1}{\alpha}}(n)$ beschränkt, wobei $n = n_0$ der Anzahl der im Quake Heap gespeicherten Schlüssel entspricht. Dies ermöglicht die gute amortisierte Laufzeit der `delete-min`-Operation. Der Parameter α wird in den folgenden Beispielen stets als $\frac{3}{4}$ angenommen. Ein Beispiel für einen Quake Heap ist in Abbildung 4 zu sehen.

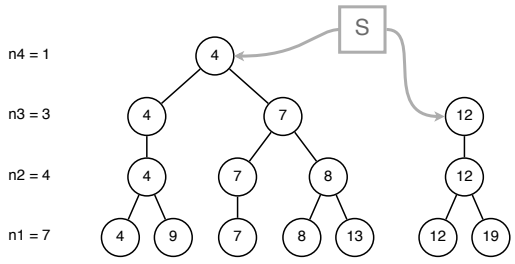


Abbildung 4 Zwei Tournament Trees bilden beispielhaft einen Quake Heap. Die Wurzelmenge S ist mit ihren Verweisen auf die verschiedenen Tournament Trees in grau dargestellt. Die Invariante ist für alle Höhen erfüllt, da $1 \leq \frac{3}{4} * 3 \wedge 3 \leq \frac{3}{4} * 4 \wedge 4 \leq \frac{3}{4} * 7$ gilt.

Die Methode `insert` kann unter Erfüllung der Invariante einfach durchgeführt werden, indem ein neuer Tournament Tree mit einem Knoten und dem neuen Schlüssel der Menge S hinzugefügt wird. Da somit lediglich n_1 um eins erhöht wird, ist dies jederzeit ohne eine Verletzung der Invariante möglich. Dieses Vorgehen wird in Abbildung 5 verdeutlicht.

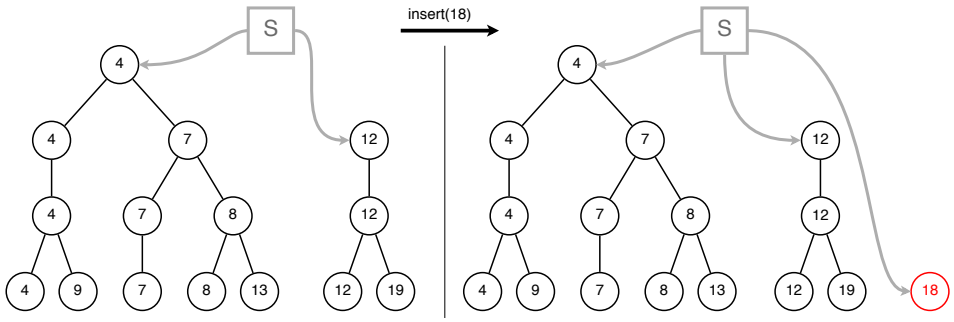


Abbildung 5 Ein Beispiel für eine Einfügeoperation in einem Quake Heap. Es wird ein neuer Tournament Tree bestehend aus einem Knoten mit dem neuen Schlüssel der Wurzelmenge hinzugefügt.

Auch die Methode `decrease-key` kann auf eine einfache Weise implementiert werden. Ist der zu vermindernde Schlüssel gleichzeitig der minimale Schlüssel in dem Tournament Tree der ihn enthält, so kann dieser Schlüssel ohne weitere Schritte auf den gewünschten Wert verringert werden ohne die Heap Eigenschaft des Tournament Trees zu verletzen. Andernfalls wird mithilfe der Operation `cut` der höchste Teilbaum des Tournament Trees abgetrennt, der den Schlüssel als Wurzelelement enthält. In diesem kann dann, ohne eine Verletzung der Heap Eigenschaft, der Schlüssel auf den gewünschten Wert verringert und der so veränderte Tournament Tree der Menge S hinzugefügt werden. Da sich durch diese Schritte die Anzahl der Knoten auf den verschiedenen Höhen nicht verändert, ändert sich an der Erfüllung von Invariante 1 nichts. Ein Beispiel, in dem ein Teilbaum mithilfe der `cut` Operation abgetrennt werden muss, um `decrease-key` durchzuführen, ist in Abbildung 6 zu sehen. Der Knoten v mit Schlüssel 8 ist nicht der kleinste Schlüssel im Tournament Tree, der ihn enthält. Aus diesem Grund wird v von seinem Elternknoten getrennt und als eigener Tournament Tree der Menge S hinzugefügt. Nun kann der Schlüssel wie gewünscht verringert werden.

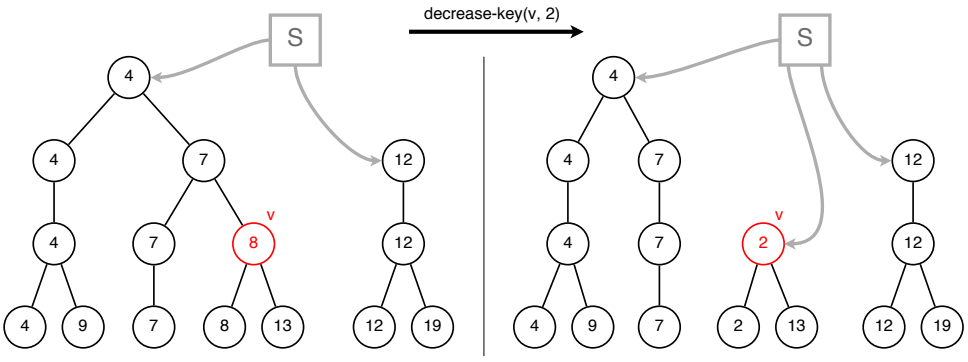


Abbildung 6 Eine `decrease-key` Operation auf einem Quake Heap. Da die erforderliche Verminderung des Schlüssels die Heap-Bedingung verletzen würde, muss der Teilbaum zunächst mithilfe der `cut`-Operation abgetrennt werden. Im neu entstandenen Tournament Tree kann der Schlüssel dann wie gefordert vermindert werden.

Die `delete-min` Operation erfordert eine Mehrzahl an Schritten. Zunächst wird unter allen Wurzeln der Menge S der Tournament Tree u mit dem kleinsten Schlüssel gesucht. Alle Knoten, die diesen Schlüssel enthalten, werden aus u entfernt. Teilbäume deren Eltern dabei entfernt wurden, werden einfach als neue Wurzel wieder der Menge S hinzugefügt. Im Anschluss werden Bäume mit gleicher Höhe mithilfe der Methode `link` zu einem neuen Tournament Tree verbunden, sodass keine zwei Bäume gleicher Höhe in der Menge S mehr existieren. Die Auswirkungen dieser Schritte sind in Abbildung 7 zu sehen. Hier verbleibt nach der Entfernung aller Knoten mit minimalem Schlüssel 2 der Teilbaum mit Schlüssel 13. Dieser wird der Menge S als neue Wurzel hinzugefügt. In diesem Beispiel haben alle Wurzeln in S verschiedene Höhen, weshalb keine `link` Operation durchgeführt werden muss.

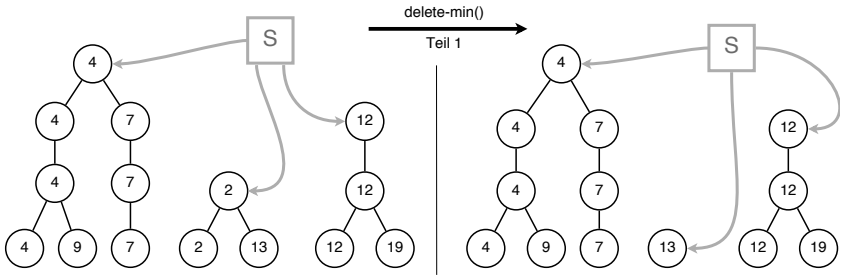


Abbildung 7 Teil 1 der `delete-min` Operation. Alle Knoten mit minimalem Schlüssel 2 werden entfernt, verbleibende Teilbäume der Menge S hinzugefügt und Tournament Trees gleicher Höhe mithilfe der `link` Operation verbunden.

Nun folgt der letzte und für die Datenstruktur namensgebende Schritt. Da durch das Entfernen der Knoten mit minimalem Schlüssel die Anzahl an Knoten n_i auf verschiedenen Höhen verändert werden, muss die Erfüllung der Invariante überprüft werden. Gesucht wird die kleinste Höhe i, für welche die Invariante 1 verletzt ist. Existiert ein solches i, werden alle Knoten oberhalb der Höhe i entfernt. Dies erzeugt gegebenenfalls viele kleinere Bäume, die im Anschluss zur Menge S hinzugefügt werden, um später einen Neuaufbau (im Rahmen der nächsten `delete-min` Operation) zu ermöglichen. Die Bezeichnung „Erdbebenhaufen“ referenziert auf diese Zerstörung hoher Strukturen und das Zurücklassen niedrigerer Teilbäume. Abschließend wird der minimale Schlüssel zurückgegeben. Eine umfassende Strukturveränderung mit geringer Höhe i sollte aufgrund des hohen Rechenaufwands für den folgenden Neuaufbau möglichst selten vorkommen. Der zweite Teil, der für die `delete-min` Operation erforderlichen Schritte, ist in Abbildung 8 dargestellt.

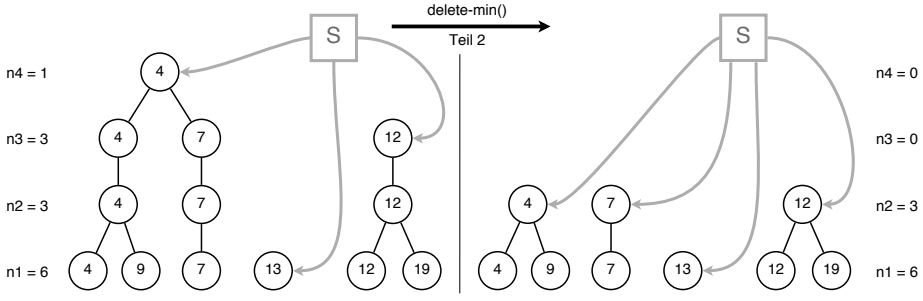


Abbildung 8 Teil 2 der `delete-min` Operation. Die Invariante 1 ist für Höhe $i = 2$ verletzt. Daher werden alle Knoten mit Höhe $i > 2$ entfernt und verbleibende Teilbäume in S aufgenommen.

3 Implementation

Für die effiziente Implementierung der Operationen müssen einige Anpassungen an der Struktur der Tournament Trees und Quake Heaps durchgeführt werden. Diese sind an die Vorgehensweise von Wolfgang Mulzer [3] angelehnt.

Die inneren Knoten der Tournament Trees speichern nicht den minimalen Schlüssel ihrer Kinder selbst, sondern einen Verweis `minleaf` auf das Blatt, das diesen enthält. Hierdurch muss für eine Veränderung eines Schlüssels innerhalb der `decrease-key` Operation lediglich der Schlüssel im Blatt selbst und nicht auf dem ganzen Pfad verändert werden. Jeder Knoten erhält außerdem einen Zeiger `parent` auf seinen Elternknoten. Im Falle einer Wurzel ist dieser Verweis leer. Dies ermöglicht unter anderem eine schnelle Überprüfung, ob es sich bei einem Knoten um eine Wurzel handelt und wird für die `cut` Operation der Tournament Trees benötigt. Um innerhalb der `decrease-key` Operation den höchsten Knoten u auffinden zu können, der auf den Schlüssel des Blatts v verweist, erhält jedes Blatt v einen zusätzlichen Zeiger `top` auf u. Außerdem kennt jeder Knoten seine Höhe `height`, damit im Rahmen der `decrease-key` und `delete-min` Operationen die Höhe der Bäume nicht durch eine Traversierung berechnet werden muss. Eine Visualisierung eines solchen Tournament Trees mit allen Zusatzinformationen ist in Abbildung 9 zu sehen.

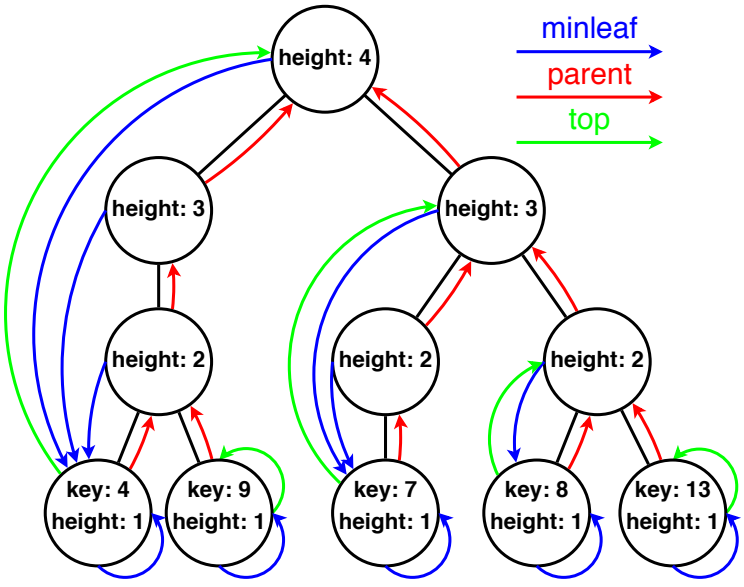


Abbildung 9 Der Tournament Tree aus Abbildung 1 mit den Zusatzinformationen zum Elter, minimalen Blatt und höchsten Knoten der den Schlüssel aus dem Blatt enthält. Diese sind für die effiziente Implementation der Quake Heaps erforderlich.

Anstatt einen zusätzlichen Zeiger für den kleineren Kindknoten einzuführen, wird dieser stets im linken Teilbaum gespeichert. Auch ein Knoten mit Grad eins besitzt stets nur einen linken Kindknoten. Somit ergibt sich folgender Pseudocode für die Implementation der beiden Methoden `link` (Listing 1) und `cut` (Listing 2):

Listing 1 Pseudocode für die `link`-Operation

```
link(u, v):
   // Links the two tournament trees u and v to form a higher one
   if u.minleaf.key <= v.minleaf.key:
      min = u; max = v
   else
      min = u; max = v
   new = TournamentTree()
   new.leftchild=min; new.rightchild=max
   new.height = min.height + 1
   new.minleaf = min.minleaf
   min.minleaf.top = new
   min.parent = new; max.parent = new
   return new
```

Listing 2 Pseudocode für die `cut`-Operation

```
cut(u):
   // Cuts the tournament tree u from its parent
   if(u.parent != null)
      u.parent.rightchild = null
      u.parent = null
```

Zur effizienten Implementierung benötigen auch die Quake Heaps einige Zusatzinformationen, die in zwei Arrays gespeichert werden. Die Einträge i des Arrays T enthalten Listen von Tournament Trees der Höhe i. Das Array T ersetzt die Wurzelmenge S, um einen effizienten Zugriff auf Bäume der Höhe i innerhalb von `delete-min` zu ermöglichen. Die Einträge i des Arrays n entsprechen der Knotenanzahl auf Höhe i aller Tournament Trees und werden für die Überprüfung der Erfüllung der Invariante benötigt.

Damit ergibt sich der Pseudocode für die Quake Heap Operationen `insert` (Listing 3), `decrease-key` (Listing 4) und `delete-min` (Listing 5) wie folgt:

Listing 3 Pseudocode für die `insert`-Operation

```
insert(k):
   // Insert a new key into the quake heap
   new = TournamentTree()
   new.key = k
   new.minleaf = new; new.top = new
   new.height = 1
   T[1].append(new)
   n[1] = n[1] + 1
   return new
```

Listing 4 Pseudocode für die `decrease-key`-Operation

```
decrease-key(s, k):
   // Decrease the key of node s to k
   t = s.minleaf.top
   if(t.parent) != null):
      cut(t)
      T[t.height].append(t)
   s.minleaf.key = k
```

Listing 5 Pseudocode für die `delete-min`-Operation

```
1  delete-min():
2     // Remove the minimal key of the quake heap and return it
3     // Step 1: Find tournament tree with minimal key
4     min = null
5     for(list in T):
6        for(u in list):
7           if(min == null or u.minleaf.key <= min.minleaf.key):
8              min = u
9
10    // Step 2: Remove all nodes with minimal key
11    T[min.height].remove(min)
12    current = min
13    while(current != null):
14       if(current.rightchild != null)
15          r = current.rightchild
16          cut(r)
17          T[r.height].append(r)
18       n[current.height] = n[current.height] - 1
19       current = current.leftchild
20
21    // Step 3: Link trees with equal height
22    for(i = 1 to T.size):
23       while(T[i].size > 1):
24          u = T[i].get(0); v = T[i].get(1)
25          T[i].remove(u); T[i].remove(v)
26          w = link(u, v)
27          T[i + 1].append(w)
28          n[i + 1] = n[i + 1] + 1
29
30    // Step 4: Check if invariant is violated
31    violation_i = 0
32    for(i = 1 to T.size - 1):
33       if(n[i + 1] > α * n[i]): // Invariant violated -> Earthquake!
34          violation_i = i
35
36    if(violation_i > 0)
37       // Remove all nodes at height violation_i + 1 and up
38       nodes_to_process = Queue()
39       for(i = violation_i + 1 to T.size)
40          for(u in T[i])
41             nodes_to_process.enqueue(u)
42          T[i] = [] // Empty treelists
43          n[i] = 0 // Reset node counter
44       while(nodes_to_process.size > 0)
45          u = nodes_to_process.dequeue()
46          if(u.height > violation_i + 1)
47             nodes_to_process.enqueue(u.leftchild)
48             nodes_to_process.enqueue(u.rightchild)
49          else // Tree u has height violation_i + 1
50             // add its children back to the quake heap
51             for(v in [u.leftchild, u.rightchild])
52                    v.parent = null
53                    v.minleaf.top = v
54                    T[v.height].add(v)
55    return min.minleaf.key
```

4 Analyse

Zur Laufzeitanalyse der Quake Heap Operationen werden die Laufzeiten der Methoden `link` und `cut` der Tournament Trees benötigt. `link` besitzt eine Laufzeit von $O(1)$, da in jedem Fall ein Vergleich durchgeführt und eine konstante Anzahl von Zeigern gesetzt wird. Gleiches gilt für die Operation `cut`. Hier werden lediglich zwei Zeiger verändert, sodass auch hier die Laufzeit $O(1)$ beträgt.

Für die amortisierte Analyse der Quake Heap Operationen wird im Folgenden die Potenzialmethode verwendet. Hier wird einer Datenstruktur ein Potenzial zugeordnet, das in den Operationen aufgebaut oder verbraucht werden kann. Die amortisierte Laufzeit a entspricht dann der tatsächlichen Laufzeit t plus der Potenzialdifferenz nach und vor der Operation. Sei die Potenzialfunktion:

$$\phi = \#N + 2\#T + \frac{2}{2\alpha - 1}\#B \tag{2}$$

Dabei entspricht $\#N$ der Anzahl von Knoten, $\#T$ der Anzahl von Bäumen und $\#B$ der Anzahl von Knoten mit Grad eins (also der Anzahl von Knoten mit nur einem Kind) im Quake Heap. Seien ϕ', $\#N'$, $\#T'$ und $\#B'$ jeweils die entsprechenden Werte nach der betrachteten Operation. Es folgt die Analyse der Operationen von den Quake Heaps im Detail.

Die Operation `insert` (Listing 3) besitzt eine tatsächliche Laufzeit von $t = O(1)$, da lediglich ein neuer Baum mit einem Knoten und dem gewünschten Schlüssel angelegt und dem Array T hinzugefügt wird. Darüber hinaus wird das Array n mit den Anzahlen von Bäumen der verschiedenen Höhen aktualisiert. Diese Schritte sind unabhängig von der Größe der Datenstruktur in konstanter Zeit $O(1)$ möglich. Durch `insert` wird die Anzahl der Knoten $\#N$ und Bäume $\#T$ im Quake Heap jeweils um 1 erhöht. Die Anzahl der Knoten mit Grad eins $\#B$ verändert sich hingegen nicht. Damit ist die Potenzialdifferenz $\phi' - \phi = 1 + 2 = 3$ und die amortisierte Laufzeit $a = O(1) + 3 = O(1)$.

Die tatsächliche Laufzeit der `decrease-key` Operation (Listing 4) ist $O(1)$, da die Laufzeit der Methode `cut` $O(1)$ ist und darüber hinaus lediglich der neue Baum in eine Liste des Arrays T eingefügt und der Schlüssel im Blatt aktualisiert wird. Hier zeigt sich die Notwendigkeit der in Kapitel 3 beschriebenen Anpassung der Tournament Trees. Der Zeiger top der Blätter ermöglicht das Auffinden des höchsten Knotens mit gleichem Schlüssel von der Wurzel aus in konstanter Zeit. Dadurch, dass die inneren Knoten anstelle des Schlüssels selbst lediglich einen Verweis auf den Schlüssel im Blatt speichern, muss auch nur dieser angepasst werden. Für die Potenzialdifferenz von `decrease-key` gilt $\phi' - \phi \leq 2 + \frac{2}{2\alpha-1}$, da sich die Anzahl an Bäumen $\#T'$ und die Anzahl an Knoten mit Grad eins $\#B'$ jeweils maximal um eins erhöhen. Da α eine Konstante im Intervall $\left(\frac{1}{2}, 1\right)$ ist, gilt damit $a = O(1) + O(1) = O(1)$.

Aufgrund der Länge des Programmcodes für die `delete-min` Operation (Listing 5) werden die einzelnen Schritte in der amortisierten Analyse zunächst getrennt betrachtet. Die tatsächlichen Kosten der Minimumsuche in Zeile 2 bis 9 betragen $O(\#T)$, da zur Minimumsuche einmal alle $\#T$ Bäume durchgegangen werden müssen.

Das Löschen des Minimums in Zeilen 10 bis 20 besitzt Kosten $O(\log(n))$, da hierzu der Tournament Tree von der Wurzel bis zum Blatt mit minimalem Schlüssel durchgegangen und alle Knoten auf dem Weg entfernt werden müssen. Die maximale Höhe eines Tournament Trees im Quake Heap ist wie in Kapitel 2.2 beschrieben $O(\log_{\frac{1}{\alpha}}(n))$ bei n Schlüsseln.

Das Verschmelzen gleich hoher Bäume in Zeile 21 bis 29 hat ebenfalls Laufzeit $O(\#T)$, da die `link` Operation konstante Laufzeit hat und diese maximal so oft ausgeführt wird, wie Tournament Trees im Quake Heap existieren. Da im Anschluss an das Verschmelzen nur ein

Baum pro Höhe existieren kann, ist die Anzahl der Bäume $\#T^1$ nach dem Verschmelzen auch auf $O(log(n))$ beschränkt.

Für die Programmcodezeilen 2 bis 29 (Teil 1) betragen die tatsächlichen Kosten also $t^1 = O(\#T) + O(\log\ n)$. Die Veränderung der Anzahl von Bäumen ist $\#T^1 - \#T = O(\log\ n) - \#T$ und da keine neuen Knoten mit Grad eins erzeugt werden, kann sich deren Anzahl $\#B^1$ nicht erhöhen. Durch das Löschen des Minimums wird minimal 1 Knoten entfernt und durch das Verschmelzen gleich hoher Bäume kommen maximal $\#T$ neue Knoten hinzu. Damit gilt für die Potenzialdifferenz der Zeilen 2 bis 29: $\phi^1 - \phi = \#T + 2(O(\log\ n) - \#T) = 2O(\log\ n) - \#T$ und für die amortisierten Kosten: $a^1 = \#T + O(\log\ n) + 2O(\log\ n) - \#T = O(\log\ n)$.

Das Überprüfen der Verletzung der Invariante und das darauffolgende Entfernen aller Knoten oberhalb der Höhe i, für die die Invariante verletzt wurde (Teil 2: Zeile 30 bis 55), besitzen tatsächliche Kosten in Höhe der maximalen Höhe der Bäume $O(\log\ n)$ plus der Anzahl der gelöschten Knoten $O(D)$, also $t^2 = O(D) + O(\log\ n) = O(\sum_{j>i} n_j^1) + O(log\ n)$, wobei n_j^1 der Anzahl der Knoten auf Höhe j unmittelbar vor Zeile 30 entspricht.

Für die Veränderung der Anzahl an Knoten gilt $\#N^2 - \#N^1 \leq -D$ und für die Anzahl an Bäumen $\#T^2 - \#T^1 \leq n_i^1$. Sei b_i^1 die Anzahl von Knoten mit Grad eins auf Höhe i vor Zeile 30, dann gilt $n_i^1 \geq 2n_{i+1}^1 - b_i^1$, da alle Knoten, abgesehen von den b_i^1 Einzelkindern, auf Höhe $i+1$ 2 Kinder besitzen. Daraus folgt $b_i^1 \geq 2n_{i+1}^1 - n_i^1 \geq 2(\alpha n_i^1) - n_i^1 = (2\alpha - 1)n_i^1$, da Invariante 1 auf der Höhe i verletzt ist (es gilt $n_{i+1}^1 > \alpha n_i^1$). Aus diesem Grund beträgt die Differenz der Anzahl an Knoten mit Grad eins vor und nach dem Erdbeben $\#B^2 - \#B^1 \leq -(2\alpha - 1)n_i^1$. Damit ist die Potenzialdifferenz von Zeile 30 bis 55:
$\phi^2 - \phi^1 \geq -\sum_{j>i} n_j^1 + 2n_i^1 + \frac{-2(2\alpha-1)n_i^1}{2\alpha-1} = -\sum_{j>i} n_j^1 = -D$
und die amortisierten Kosten $a^2 = t^2 + \phi^2 - \phi^1 \leq O(D) + O(\log\ n) - D = O(\log\ n)$.

Die amortisierten Kosten für die gesamte `delete-min` Operation (Zeile 2 bis 55) betragen somit: $a = a^1 + a^2 = O(\log\ n) + O(\log\ n) = O(\log\ n)$.

5 Fazit

Quake Heaps sind eine strukturell besonders einfache Implementation von Prioritätswarteschlangen. Sie erfordern keine lokalen Zähler oder komplexe Fallunterscheidungen und beruhen lediglich auf einer einzigen, für die gesamte Datenstruktur geltenden Invariante. Zusätzlich sind die Quake Heaps an verschiedenen Stellen anpassbar. Beispielsweise kann der Wert α der Invariante im gegebenen Intervall frei gewählt und das Verbinden von Bäumen gleicher Höhe auch an anderen Stellen im Algorithmus durchgeführt werden.

Zwar ermöglicht die Verwendung von Tournament Trees eine intuitive Betrachtung der Operationen, jedoch benötigen diese eine lineare Anzahl an zusätzlichen Knoten zum Speichern der Schlüssel. Aus diesem Grund existieren Variationen der Datenstruktur, welche heap-ordered oder half-ordered Trees einsetzen.

Insgesamt führt die Methodik der Quake Heaps laut Chan weder zum kürzesten Programmcode noch zur besten Performanz in der Praxis. Ihr einfacher Aufbau und die unkomplizierte Analyse zusammen mit der guten amortisierten Laufzeit machen jedoch eine theoretische Betrachtung interessant und geben ihnen einen pädagogischen Wert.

Literatur

1 Timothy M. Chan. Quake heaps: A simple alternative to fibonacci heaps.

2 Michael L Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3), 1987.

3 Wolfgang Mulzer. Quake Heaps – Erdbebenhaufen.

Effiziente Datenstruktur für statische Bäume basierend auf Range-Min/Max-Trees

Patrick Dinklage[1]

1 **TU Dortmund, Fakultät für Informatik, Lehrstuhl 11 – Algorithm Engineering, Otto-Hahn-Str. 14, 44227 Dortmund, Deutschland**
patrick.dinklage@tu-dortmund.de

Zusammenfassung

G. Navarro und K. Sadakane stellen in ihrem Papier [1] eine platzeffiziente Datenstruktur für geordnete statische Bäume vor, welche eine Vielzahl typischer Anfragen in konstanter Zeit beantworten kann. Die Datenstruktur besteht aus der BP-Darstellung („balanced parantheses") des Baumes, welche durch einen Range-Min/Max-Baum erweitert wird. Mit diesem können einige Grundanfragen auf Bitvektoren in konstanter Zeit beantwortet werden: Die Summe über ein Intervall sowie die Vorwärts- und Rückwärtssuche nach einem bestimmten Intervall mit dem gesuchten Summenwert. Bereits ohne Blick auf Bäume ist dies eine Verbesserung gegenüber den bisher besten bekannten effizienten Datenstrukturen zur Beantwortung dieser Anfragen. Die Autoren zeigen jedoch ferner, dass sich darauf aufbauend auch typische Anfragen auf Bäumen (z.B. Elternknoten oder Iteration über Kinder) in konstanter Zeit beantworten lassen - ohne, wie gemein üblich, noch weitere Datenstrukturen einzuführen. Um den benötigten Speicherplatz zu minimieren, stellen die Autoren zunächst die Datenstruktur für kleine Bäume, d.h. bis zu einer bestimmten Größe vor. Sie zeigen dann, dass die BP-Darstellung eines größeren Baums in Blöcke unterteilt werden kann, um die Lösung für kleine Bäume auf einzelne Blöcke anzuwenden. Für Block-übergreifende Anfragen erzeugen sie weitere Datenstrukturen, mit Hilfe derer sie in konstanter Zeit beantwortet werden können. Für Bäume beliebiger Größe n erreichen sie eine Speicherkomplexität von insgesamt $2n + \mathcal{O}(\frac{n}{\text{polylog}(n)})$, womit einige bekannte Datenstrukturen im Bezug auf ihre Größe deutlich unterboten werden.

1998 ACM Subject Classification E.1 Data Structures, E.2 Data Storage Representations

Keywords and phrases trees, succinct data structures, range min/max trees, tree navigation, range minimum / maximum queries

Digital Object Identifier 10.4230/LIPIcs.BADS.2017.15

1 Einführung

Diese Ausarbeitung stellt eine Zusammenfassung der Erkenntnisse von G. Navarro und K. Sadakane dar, die sie 2014 in ihrem Papier [1] veröffentlicht haben.

Wir betrachten eine effiziente Datenstruktur zur Darstellung von Bäumen, die einige häufig gebrauchte Anfragen (siehe Unterabschnitt 2.2) auf diesen in konstanter Zeit beantworten kann und dabei nur sublinear viel Speicher zusätzlich zur BP-Darstellung (siehe Unterabschnitt 2.1) benötigt.

Hierzu wird die BP-Darstellung von Bäumen beliebiger Größe in hinreichend kleine Blöcke unterteilt, auf denen effizient gearbeitet werden kann (siehe Abschnitt 4). Unter Verwendung zusätzlicher Datenstrukturen können dann auch Anfragen auf beliebig großen Bäumen effizient beantwortet werden (siehe Abschnitt 5).

2 Grundlagen

In diesem Abschnitt definieren wir die verwendeten Notationen und Darstellungen, sowie die Anfragen auf Bäumen, die von der später entwickelten Datenstruktur unterstützt werden.

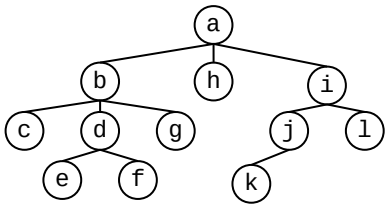
Ein String $B \in \{0,1\}^*$ heißt *Bitvektor*. Für $i, j \in \mathbb{N}$ bezeichne $B[i] \in \{0,1\}$ das i-te Bit in B und $B[i,j] \in \{0,1\}^*$ mit $i \leq j$ die Teilbitfolge von B von der Stelle i bis zur Stelle j (jeweils einschließlich). Für ein $i \geq 1$ definieren wir die Funktion $\text{rank}_1(B,i)$, die die Anzahl der 1-Bits in $B[1,i]$ zurückgibt ($\text{rank}_0(B,i)$ analog für 0-Bits). Die Beziehung $\text{rank}_0(B,i) = i - \text{rank}_1(B,i)$ ist wohlbekannt.

Ein *Integer-Array* ist eine endliche Folge natürlicher Zahlen $A = a_1, a_2, \ldots, a_n$. Es bezeichne $A[i] = a_i$ die i-te Zahl aus A.

2.1 Balanced Parantheses

Ein Baum mit n Knoten kann durch einen Bitvektor $B \in \{0,1\}^m$ der Länge $m = 2n$ eindeutig repräsentiert werden [2]. In dieser Darstellung wird jeder Knoten des Baums als Klammerpaar notiert, d.h. mit jeweils einer öffnenden und einer schließenden Klammer. Daher wird sie *balanced parantheses* genannt (im Folgenden kurz *BP-Darstellung*). Die Syntax der Darstellung ergibt sich aus der *In-Order*-Traversierung des Baums: Eine öffnende Klammer markiert das erste Besuchen eines Knotens und die dazugehörige schließende Klammer das letzte Verlassen des Knotens. Die schließende Klammer nennen wir *passend* zur öffnenden Klammer (umgekehrter Fall analog).

Für die praktische Kodierung dieser Darstellung eignen sich Bitvektoren, wobei öffnende Klammern üblicherweise als *1-Bits* und schließende Klammern als *0-Bits* kodiert werden. Abbildung 1 zeigt beispielhaft die BP-Darstellung und den dazugehörigen Bitvektor für einen Baum.



i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
B	(	(	(	)	(	(	)	(	)	)	(	)	)	(	)	(	(	(	)	)	(	)	)	)
Knoten	a	b	c		d	e		f			g			h		i	j	k			l			

Abbildung 1 Ein Baum der Größe $n = 12$ und die dazugehörige BP-Darstellung B der Länge $2n = 24$. Die alphabetische Auflistung der Knoten (`a`, `b`, ...) entspricht der In-Order-Traversierung des Baums. In der untersten Zeile („Knoten"), die nicht gespeichert wird und hier nur zur Hilfe dient, ist vermerkt, zu welchem Knoten die jeweilige öffnende Klammer gehört.

Wir definieren den *excess*-Wert für B und eine Position i wie folgt:

$$\text{excess}(B,i) \stackrel{\text{def}}{=} \text{rank}_1(B,i) - \text{rank}_0(B,i) = 2 \cdot \text{rank}_1(B,i) - i.$$

Dieser Wert entspricht der Anzahl der öffnenden Klammern im Teilstring $B[1,i]$, auf die im gleichen Teilstring keine passende schließende Klammer folgt. Für $B[i,j]$ gilt:

$$\text{excess}(B,i,j) \stackrel{\text{def}}{=} \text{excess}(B,j) - \text{excess}(B,i-1).$$

Wir definieren ferner die Funktionen *findclose*, *findopen* und *enclose* analog zu [3]: findclose(B, i) gibt für eine öffnende Klammer $B[i] =$ `'('` die Position der passenden schließenden Klammer in $B[i, m]$ zurück. Analog gibt findopen(B, i) für eine schließende Klammer $B[i] =$ `')'` die Position der passenden öffnenden Klammer in $B[1, i-1]$ zurück. enclose(B, i) ermittelt schließlich die rechteste Position $k < i$, so dass $i \in [k,\ \text{findclose}(B, k)]$ gilt; k entspricht dann also der Position der öffnenden Klammer des i umschließenden Klammerpaares.

2.2 Anfragen auf Bäumen

Abbildung 2 gibt eine Übersicht über die Anfragen, die in dieser Ausarbeitung betrachtet werden. Sie dient lediglich der Beschreibung der Anfragen und ist unabhängig von der tatsächlichen Darstellung von Bäumen und Knoten. Sie umfasst überdies nur eine Auswahl der Anfragen, die von der in [1] beschriebenen Datenstruktur unterstützt werden.

3 Anfragen auf Bäumen in BP-Darstellung

In diesem Abschnitt betrachten wir, wie die in Unterabschnitt 2.2 beschriebenen Anfragen auf Bäumen, die in BP-Darstellung vorliegen, auf einige grundlegende Anfragen heruntergebrochen werden können. Ziel ist es, die Anfragen durch möglichst wenige primitive Anfragen ausdrücken zu können, so dass nur für diese Datenstruktur aufgebaut und gespeichert werden muss.

Im Folgenden sei B ein Bitvektor, der eine Klammerndarstellung wie in Unterabschnitt 2.1 kodiert. Wir repräsentieren einen Knoten v_i durch die Position i seiner öffnenden Klammer in B.

3.1 Herunterbrechung der Anfragen

Die in Unterabschnitt 2.2 genannten Anfragen lassen sich für BP-Strings leicht auf die Anfragen *rank*, *select*, *findclose*, *findopen* und *enclose* herunterbrechen [1]. Seien i und j Positionen, die Knoten im dargestellten Baum repräsentieren (d.h. $B[i] = 1$ und $B[j] = 1$):

$$
\begin{aligned}
\text{parent}(i) &= \text{enclose}(B, i) \\
\text{isancestor}(i, j) &= \text{„ja“} \Leftrightarrow i \leq j \leq \text{findclose}(B, i) \\
\text{depth}(i) &= \text{rank}_1(B, i) - \mathit{rank}_0(B, i) \\
\text{isleaf}(i) &= \text{„ja“} \Leftrightarrow B[i+1] = 0 \\
\text{subtree\_size}(i) &= \frac{\text{findclose}(B, i) - i + 1}{2} \\
\text{first\_child}(i) &= i + 1 \text{ (falls } i \text{ kein Blatt ist)} \\
\text{last\_child}(i) &= \text{findopen}(B,\ \text{findclose}(B, i) - 1) \text{ (falls } i \text{ kein Blatt ist)} \\
\text{next\_sibling}(i) &= \text{findclose}(B, i) + 1 \text{ (falls } i \text{ nicht letztes Kind ist)} \\
\text{prev\_sibling}(i) &= \text{findopen}(B, i-1) \text{ (falls } i \text{ nicht erstes Kind ist)}
\end{aligned}
$$

Zeigt die Antwort von *next_sibling* auf eine schließende Klammer ($B[\text{next\_sibling}(i)] = 0$), dann ist i das letzte Kind seines Elternknotens. Für *prev_sibling* kann hingegen vorab überprüft werden, ob es sich bei i um das erste Kind handelt: In dem Fall ist $B[i-1] = 1$ (öffnende Klammer).

Anfrage	Ausgabe
parent(v)	Der direkte Elternknoten von v.
isancestor(u, v)	Ist u ein Vorfahre von v?
depth(v)	Die Tiefe von v im Baum (von der Wurzel aus).
isleaf(v)	Ist v ein Blatt?
subtree_size(v)	Die Größe des von v aufgespannten Teilbaums.
first_child(v)	Das erste Kind von v.
last_child(v)	Das letzte Kind von v.
next_sibling(v)	Der nächste Geschwisterknoten von v.
prev_sibling(v)	Der vorhergehende Geschwisterknoten von v.

Abbildung 2 Anfragen auf Bäumen und deren Ausgaben. u und v bezeichnen dabei Knoten.

3.2 Grundlegende Anfragen auf BP-Strings

Wir definieren nun einige grundlegende, sehr allgemeine Anfragen auf Bitvektoren, mit denen *findclose*, *findopen*, *enclose* und *rank* ausgedrückt werden können.

Eine Funktion $g : \{0, 1\} \to \{-1, 0, 1\}$ heißt $\pm$-*Funktion*. Für $i, j, d \in \mathbb{N}$ definieren wir:

$$\begin{aligned}
\text{sum}(B, g, i, j) &\stackrel{\text{def}}{=} \sum_{k=i}^{j} g(B[k]) \\
\text{fwd_search}(B, g, i, d) &\stackrel{\text{def}}{=} \min\{j \geq i \mid sum(B, g, i, j) = d\} \\
\text{bwd_search}(B, g, i, d) &\stackrel{\text{def}}{=} \max\{j \leq i \mid sum(B, g, j, i) = d\}
\end{aligned}$$

Die Funktion *sum* summiert die Bilder von g über ein Intervall von B. *fwd_search* sucht vorwärts, von der Startposition i im Bitvektor ausgehend, die nächste Position $j \geq i$, an welcher *sum* für das Intervall $[i, j]$ den Wert d ergibt. *bwd_search* funktioniert analog, aber rückwärts, d.h. es wird $j \leq i$ unter Betrachtung des Intervalls $[j, i]$ gesucht.

Für die $\pm$-Funktion π mit $\pi(1) = 1,\ \pi(0) = -1$ ergeben sich folgende Beobachtungen:

$$\begin{aligned}
\text{excess}(B, i, j) &= \text{sum}(B, \pi, i, j) \\
\text{findclose}(B, i) &= \text{fwd_search}(B, \pi, i, 0) \\
\text{findopen}(B, i) &= \text{bwd_search}(B, \pi, i, 0) \\
\text{enclose}(B, i) &= \text{bwd_search}(B, \pi, i, 2)
\end{aligned}$$

Die Eigenschaft für *excess* folgt sofort aus π: Eine öffnende Klammer erhöht einen Zähler um $\pi(1) = 1$, während eine schließende ihn um 1 erniedrigt ($\pi(0) = -1$). Die Summe über B ist demnach die Differenz zwischen öffnenden und schließenden Klammern.

Für *findclose* wird ausgehend von einer öffnenden Klammer an Position i die nächste Position j gesucht, so dass excess$(B, i, j) = 0$. Dies entspricht nach der obigen Beobachtung per Definition der Funktion von *fwd_search*. Für *findopen* gilt diese Beobachtung analog in die Gegenrichtung.

Die Eigenschaft für *enclose* lässt sich wie folgt veranschaulichen: Es wird die öffnende Klammer des Elternknotens gesucht. An der öffnenden Klammer des betrachteten Knotens i ist der *excess*-Wert bereits 1 (da die schließende Klammer im nachfolgenden Teil von B liegt). Gesucht wird also die nächste Stelle vor i, an welcher der *excess*-Wert (in Rückrichtung) exakt 2 ist. Dies wird in Abbildung 3 anhand eines Beispiels gezeigt.

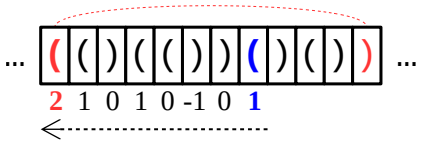


Abbildung 3 Beispiel für die Beantwortung der *enclose*-Anfrage für die blau markierte öffnende Klammer. Die Rückwärtssuche (*bwd_search*) bildet den *excess*-Wert für jede Position in Rückrichtung. An der rot markierten öffnenden Klammer ist der Wert genau 2 – hier befindet sich die öffnende Klammer des umschließenden Klammerpaares.

Mit der ±-Funktion ϕ mit $\phi(1) = 1$ und $\phi(0) = 0$ lässt sich auch *rank* simulieren:

$$\text{rank}_1(B, i) = \text{sum}(B, \phi, 1, i)$$

Man kann dies als einfaches Aufsummieren der vorkommenden 1-Bits in B interpretieren. Für $\text{rank}_0(B, i)$ findet die Beziehung aus Abschnitt 2 Anwendung.

4 Effiziente Datenstruktur für kleine Bäume

Aus den Beobachtungen in Abschnitt 3 ergibt sich, dass man lediglich die Anfragen *sum*, *fwd_search* und *bwd_search* effizient beantworten können muss, um damit alle in Unterabschnitt 2.2 genannten Anfragen ebenfalls effizient beantworten zu können.

In diesem Abschnitt führen wir eine Datenstruktur ein, die dies für kleine Bäume (die Spezifikation von „klein" erfolgt in Unterabschnitt 4.2) ermöglicht und betrachten, wie sie dazu verwendet wird.

Für einen Bitvektor B, eine Funktion g und ein Intervall $[i, j]$ definieren wir zunächst noch die Anfragen *range minimum query* (*rmq*) sowie *range maximum query* (*RMQ*) folgendermaßen:

$$\begin{aligned} \text{rmq}(B, g, i, j) &\stackrel{\text{def}}{=} \min\{sum(B, g, i, k) \mid i \leq k \leq j\} \\ \text{RMQ}(B, g, i, j) &\stackrel{\text{def}}{=} \max\{sum(B, g, i, k) \mid i \leq k \leq j\} \end{aligned}$$

Sie ermitteln demnach das Minimum bzw. Maximum der Summe der g-Bilder (z.B. im Falle von $g = \pi$ die *excess*-Werte) für alle Teilintervalle $[i, k]$ mit $k \leq j$ aus B.

4.1 Range-Min/Max-Bäume

Sei B ein Bitvektor der Länge n und $[\ell_1, r_1], [\ell_2, r_2], \ldots, [\ell_q, r_q]$ eine Partitionierung von $[1, n]$ mit $\ell_1 = 1$, $r_i + 1 = \ell_{i+1}$ und $r_q = n$. Für B und eine ±-Funktion g definieren wir den *Range-Min/Max-Baum* (kurz *rmM-Baum*) [1]. Das i-te Blatt (von links) speichert den Teilvektor $B[\ell_i, r_i]$ sowie folgende Werte:

$$\begin{aligned} e[i] &= \text{sum}(B, g, 1, r_i) \\ m[i] &= e[i-1] + \text{rmq}(B, g, \ell_i, r_i) \\ M[i] &= e[i-1] + \text{RMQ}(B, g, \ell_i, r_i) \end{aligned}$$

Jeder interne Knoten u speichert in $e[u]$ den letzten („rechtesten") Wert e seiner Kinder, in $m[u]$ und $M[u]$ das Minimum bzw. Maximum über die Werte m bzw. M der Kinder. Die Wurzel speichert demnach $e = \text{sum}(B, g, 1, n)$, $m = \text{rmq}(B, g, 1, n)$ und $M = \text{RMQ}(B, g, 1, n)$. Aus Gründen der Anschaulichkeit wird hier o.B.d.A. verlangt, dass der rmM-Baum ein

vollständiger Baum ist (d.h. jeder innere Knoten hat genau k Kinder, unabhängig von der Länge des Bitvektors). Abbildung 4 zeigt beispielhaft einen rmM-Baum für einen Bitvektor und $g = \pi$.

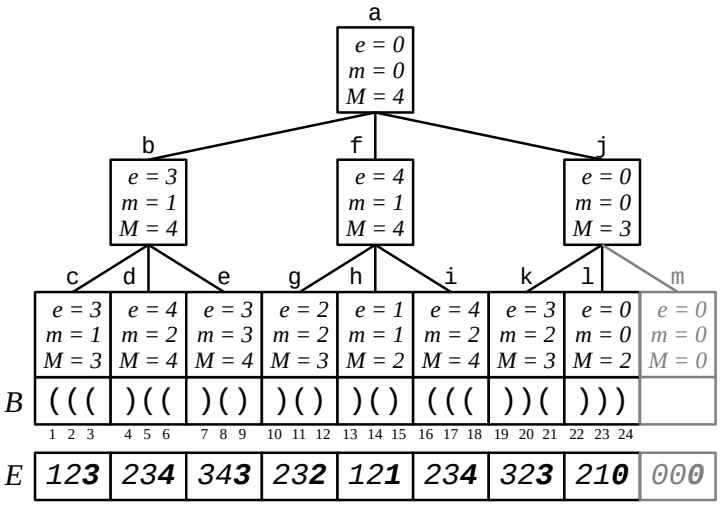


Abbildung 4 rmM-Baum für den Bitvektor aus Abbildung 1 und $g = \pi$, mit Blockgröße $s = 3$ und $k = 3$ Kindern je Knoten. In jedem Knoten werden die gespeicherten Werte e, m und M dargestellt, für jedes Blatt der entsprechende Block aus B. Der Knoten `m` wurde künstlich hinzugefügt, um den Baum zu vervollständigen. Das *excess-Array* E wird nicht gespeichert; es dient hier lediglich der Übersicht.

Folgendes Lemma aus [1] wird für die Abschätzungen unserer Laufzeiten bedeutend sein:

▶ **Lemma 1.** *Jedes Intervall $[i, j] \subseteq [1, n]$ in T_{mM} wird durch eine disjunkte Menge von $\mathcal{O}(ck)$ Teilintervallen abgedeckt, so dass das linkeste und rechteste jeweils Teilintervalle eines Blattes aus T_{mM} ist, während die dazwischenliegenden Teilintervalle vollständige Knoten in T_{mM} abdecken.*

Beweis. Jedes Blatt in T_{mM} wird durch ein Intervall der Länge s in B abgedeckt. Ist $|j - i| \leq ks$, so deckt das Intervall $[i, j]$ höchstens k Blätter ab und das Lemma folgt sofort. Wir betrachten also den Fall $|j - i| > ks$: Dann enthält $[i, j]$ mehr als k Blätter vollständig. Da T_{mM} k-adisch ist, können k Blätter zu einem Knoten zusammengefasst werden. Dieser Ansatz kann rekursiv für darüberliegende Ebenen verfolgt werden, d.h. k benachbarte Knoten können wiederum zu dem darüberliegenden gemeinsamen Elternknoten zusammengefasst werden. Da T_{mM} die Tiefe $\mathcal{O}(c)$ hat, kann das Intervall $[i, j]$ also durch $\mathcal{O}(ck)$ Knoten aus T_{mM} ausgedrückt werden und das Lemma folgt. ◀

▶ **Beispiel 2.** Wir betrachten den rmM-Baum in Abbildung 4 und das Intervall $[5, 23]$. Die Randintervalle $[5, 6]$ und $[22, 23]$ sind Teilintervalle der Blätter `d` bzw. `l`. Durch das dazwischenliegende Intervall $[7, 21]$ werden die fünf Blätter `e`, `g`, `h`, `i` und `k` abgedeckt, also mehr als $k = 3$. Offensichtlich haben die Blätter `g`, `h` und `i` jedoch den gemeinsamen Elternknoten `f`, zu dem sie zusammengefasst werden können. Das Intervall $[7, 21]$ deckt dann die vom Lemma vorausgesagten $k = 3$ Knoten `e`, `f` und `k` ab.

4.2 Aufbau der Datenstruktur

Sei w die Länge eines Maschinenwortes im RAM-Modell in Bits und $c \geq 1$ konstant. Für einen Bitvektor B der Länge $n < w^c$ und eine $\pm$-Funktion g konstruieren wir den k-adischen, vollständigen Range-Min/Max-Baum T_{mM}. Dabei wird B in logische *Blöcke* der Länge $s = \frac{1}{2}w$ unterteilt, welche die Partitionierung mit $\ell_i = s \cdot (i-1)$ bilden. Die Anzahl der Kinder je Knoten wird $k = \Theta(\frac{w}{c \log w})$ gewählt, was weiter unten verwendet wird.

Die Tiefe von T_{mM} ist $\lceil \log_k(\frac{n}{s}) \rceil = \mathcal{O}(c)$ (die asymptotische Abschätzung folgt aus $n < w^c$), die Anzahl der Knoten beträgt $\frac{n}{s} \cdot \sum_{i=0}^{\lceil \log_k(\frac{n}{s}) \rceil} k^{-i} \leq \frac{n}{s} \cdot \frac{k}{k-1} = \mathcal{O}(\frac{n}{s})$ (dies folgt aus der geometrischen Reihe).

Weil der Baum vollständig ist, kann er durch drei Integer-Arrays e', m' und M' dargestellt werden, die wie Heaps aufgebaut sind und demnach jeweils die Größe $\mathcal{O}(\frac{n}{s})$ haben. Wegen $n < w^c$ gilt für alle i: $-w^c \leq e'[i], m'[i], M'[i] \leq w^c$. Die Arrays e', m' und M' benötigen daher einen Speicherplatz von jeweils $\frac{n}{s} \cdot \frac{k}{k-1} \cdot \lceil \log(2w^c + 1) \rceil = \mathcal{O}\left(\frac{nc \log w}{w}\right)$ Bits.

4.3 Beantwortung von Anfragen

Wir betrachten nun die Beantwortung von *fwd_search* in konstanter Zeit. Die Beantwortung von *bwd_search* funktioniert analog in Gegenrichtung und wird nicht näher betrachtet. Die Beantwortung von *sum* in konstanter Zeit wird für die Beantwortung der Suchanfragen benötigt und im Zuge dessen beschrieben.

Wir setzen in diesem Abschnitt $g = \pi$, betrachten also die *excess*-Werte des Bitvektors B.

Die generelle Idee ist, dass die Blockgröße s klein genug gewählt ist, dass es sich speichertechnisch lohnt, für alle Bitfolgen der Länge s die Antworten auf alle gültigen Anfragen vorab zu ermitteln und zu speichern. Für längere Bitfolgen wird dann T_{mM} hinzugezogen.

Wir konstruieren also zunächst die universelle Tabelle fwd^s_π, welche für jeden möglichen s-Block $S = \{0,1\}^s$, jede Startposition $i \in [1, s]$ innerhalb des Blocks und jeden gültigen[1] Eingabewert $d \in [0, s]$ die Antwort für die Vorwärtssuche speichert:

$$\mathsf{fwd}^s_\pi[S, i, d] = \begin{cases} 0 & \text{falls } d \text{ nicht im Block vorkommt,} \\ \text{fwd\_search}(S, \pi, i, d) & \text{sonst.} \end{cases}$$

Diese Tabelle benötigt $\mathcal{O}(2^s s^2 \log s) = \mathcal{O}(\sqrt{2^w} w^2 \log w)$ Bits im Speicher.

Für die Beantwortung von fwd_search(B, π, i, d) sei nun $[\ell_k, r_k]$ mit $k = \lfloor \frac{i}{s} \rfloor$ der s-Block von B, der das i-te Bit enthält. Es wird nun zunächst $x = \mathsf{fwd}^s_\pi[B[\ell_k, r_k],\ i - \ell_k + 1,\ d]$ nachgeschlagen. Ist das gesuchte d im k-ten Block enthalten ($x > 0$), wird die Antwort $\ell_k + x - 1$ nach konstanter Zeit zurückgegeben.

▶ **Beispiel 3.** Wir betrachten den rmM-Baum in Abbildung 4 und die Anfrage findclose(8) = fwd_search$(B, \pi, 8, 0)$. Die Position 8 liegt im s-Block $[\ell_k, r_k] = [7, 9]$, welcher die Bitfolge $S = \texttt{010}$ enthält. Wir schlagen daher $\mathsf{fwd}^s_\pi[\texttt{010}, 2, 0]$ nach und finden das Ergebnis $x = 3$ in konstanter Zeit. Die Antwort ist dann $\ell_k + x - 1 = 7 + 3 - 1 = 9$.

[1] Die Einschränkung von d auf einen nicht-negativen Wertebereich erfolgt aus praktischen Gründen: Es gibt im Rahmen dieser Arbeit keinen Anwendungsfall, in welchem nach Werten mit $-s \leq d < 0$ gesucht werden müsste.

Andernfalls muss eine Block-übergreifende Suche durchgeführt werden. Unter Berücksichtigung der Bitfolge vor der Startposition i bestimmen wir den gesuchten *globalen Zielwert* $d' = e[k] - \text{sum}(B, g, i, r_k) + d$. Wir suchen nun das erste $j > i$, so dass $m[u_j] \leq d' \leq M[u_j]$ gilt (u_j bezeichne dabei einen Knoten, der die Position j enthält). Dieses kann sich frühestens im nächsten s-Block befinden, weil es sonst bereits zurückgegeben worden wäre. Das Intervall $[\ell_{k+1}, n]$ in B, in welchem sich j also befinden muss, wird nach Lemma 1 durch höchstens $\mathcal{O}(ck)$ Knoten $u_1, u_2, \ldots$ abgedeckt. Dies nutzen wir und konstruieren die universellen Tabellen m_k und M_k, die für jede mögliche Folge von k Werten $m[\cdot]$ bzw. $M[\cdot]$ und jeden möglichen gesuchten Wert d' (mit $-w^c \leq d' \leq w^c$) den Knoten u_j speichern, der die gesuchte Position j enthält. Ist $d' < m[u_1]$, so wird mit Hilfe von M_k nach dem u_j mit $m[u_j] \leq d'$ gesucht; ist analog dazu $d' > M[u_1]$, so wird u_j mit $M[u_j] \geq d'$ in M_k gesucht. Im Fall $m[u_1] \leq d' \leq M[u_1]$ ist die Antwort indes bereits $u_j = u_1$. Innerhalb von u_j kann die Suchanfrage wiederum in konstanter Zeit mit dem finalen Ergebnis beantwortet werden: Entweder ist u_j ein Blatt und die Antwort kann in der universellen Tabelle nachgeschlagen werden, oder die Kinder von u_j werden rekursiv bis zu einem Blatt durchlaufen (wegen der Tiefe $\mathcal{O}(c)$ ebenfalls konstanter Zeitaufwand).

Die Tabellen m_k und M_k haben jeweils $(2w^c+1)^{k+1}$ Einträge mit $\log(k+1)$ Bits je Eintrag (es wird lediglich die Position der Antwort innerhalb der Folge der k Knoten gespeichert oder 0, falls die Antwort nicht in der Folge vorkommt). Wegen $k = \Theta(\frac{w}{c \log w})$ beträgt der benötigte Speicher somit $\mathcal{O}(\sqrt{2^w} \log w)$ Bits.

Eine Summe $\text{sum}(B, \pi, i, j) = \text{sum}(B, \pi, 1, j) - \text{sum}(B, \pi, 1, i-1)$ wird wie folgt in konstanter Zeit ermittelt: Wir steigen in T_{mM} zu dem Blatt $[\ell_k, r_k]$ hinab, welches die Position j enthält und lesen $\text{sum}(B, g, 1, r_{k-1}) = e[k-1]$ (vgl. Unterabschnitt 4.1). Dazu addieren wir das Ergebnis von $\text{sum}(B, g, \ell_k, j)$, welches wir aus einer universellen Tabelle sum^s_π, ebenfalls der Größe $\mathcal{O}(\sqrt{2^w} w^2 \log w)$, ablesen können (denn $j - \ell_k \leq s$). Der Schritt wird für $i-1$ wiederholt und die Differenz schließlich ausgegeben. Weil T_{mM} die Tiefe $\mathcal{O}(c)$ hat, kostet die Operation insgesamt $\mathcal{O}(c)$ Zeit.

▶ **Beispiel 4.** Wir betrachten den rmM-Baum in Abbildung 4 und die Anfrage findclose(2) = fwd_search$(B, \pi, 2, 0)$. Es ist $[\ell_k, r_k] = [1, 3]$. Wir finden $\mathsf{fwd}^s_\pi[\mathtt{111}, 2, 0] = 0$ und können daher keine direkte Antwort liefern. Es ist $d' = e[\mathtt{c}] - \text{sum}(B, \pi, 2, 3) + d = 3 - 2 + 0 = 1$ der globale Zielwert. Das Intervall $[\ell_{k+1}, n] = [4, 24]$ wird durch die Knoten `d`, `e`, `f` und `j` abgedeckt. Wegen $d' < m[\mathtt{d}]$ suchen wir also nach dem ersten j in den Knoten `e`, `f` und `j` mit $m[\cdot] \leq d'$. Die Antwort $\mathsf{m}_k[(\mathtt{e}, \mathtt{f}, \mathtt{j}), 1] = \mathtt{f}$ wird in konstanter Zeit abgelesen; für die Kinder von `f` dann wiederum $\mathsf{m}_k[(\mathtt{g}, \mathtt{h}, \mathtt{i}), 1] = \mathtt{h}$. Innerhalb von `h` wird die Antwort $j = 13$ dann in fwd^s_π in konstanter Zeit nachgeschlagen.

5 Effiziente Datenstruktur für große Bäume

Die in Abschnitt 4 beschriebene Datenstruktur beantwortet die Anfragen *sum*, *fwd_search* und *bwd_search* lediglich für Bäume der Größe $n < w^c$ in konstanter Zeit. Wir betrachten nun, wie diese Anfragen auch auf größeren Bäumen effizient beantwortet werden können.

Wir unterteilen die BP-Darstellung B des Baums in τ Blöcke der Länge w^c und konstruieren für jeden dieser Blöcke einen rmM-Baum wie in Abschnitt 4. Die Strategie ist offensichtlich, alle Anfragen innerhalb eines Blocks mit der bereits beschriebenen Datenstruktur zu beantworten. Für Block-übergreifende Anfragen werden zusätzliche Datenstrukturen der Größe $\frac{n}{w^c}$ aufgebaut, mit Hilfe derer diese Anfragen auf Blockebene effizient beantwortet werden können.

5.1 LRM- und RLM-Bäume

Sei $m_1, m_2, \ldots, m_\tau$ eine Folge natürlicher Zahlen. Für ein $j \in [1, \tau]$ definieren wir die *von-links-nach-rechts-Minima (lrm) beginnend bei* j als Folge $\mathrm{lrm}(j) = \langle j_0, j_1, j_2, \ldots \rangle$, so dass $j_0 = j$ und für beliebige r gilt: $j_r < j_r + 1$ und $m_{j_{r+1}} < m_{j_r}$. Für jedes j ergibt sich also eine Folge von Indizes, wobei auf jeden Index der Index des nächsten (echt) kleineren Wertes in der ursprünglichen Folge zeigt, falls vorhanden (ansonsten endet die Index-Folge).

Treffen sich zwei *lrm*-Folgen $\mathrm{lrm}(j_1)$ und $\mathrm{lrm}(j_2)$ an einem Index j_r, sind deren Restfolgen identisch (mit $\mathrm{lrm}(j_r)$) [1]. Daher können die *lrm*-Folgen für alle $j \in [1, \tau]$ als *lrm-Baum* dargestellt werden: Jeder Index j ist darin Kind des nächsten Index j_1 in seiner *lrm*-Folge $\mathrm{lrm}(j)$. Hierdurch entsteht zunächst ein Wald. Um diesen zu einem Baum umzuformen, werden alle Wurzeln Kinder einer künstlich erzeugten Wurzel.

Diese Datenstruktur ist auch als *2d-Min-Heap* bekannt [4].

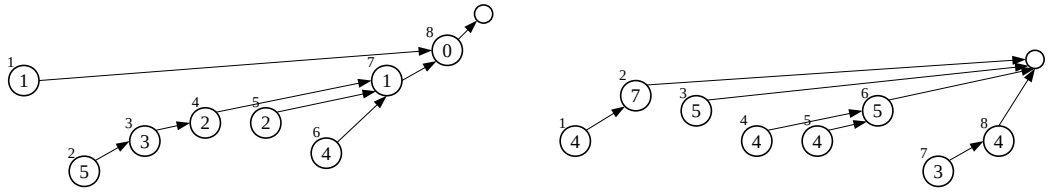


Abbildung 5 *lrm*-Baum (links) für die Folge $\langle m_1 \ldots m_8 \rangle = \langle 1, 5, 3, 2, 2, 4, 1, 0 \rangle$ und *lrM*-Baum (rechts) für die Folge $\langle M_1 \ldots M_8 \rangle = \langle 4, 7, 5, 4, 4, 5, 3, 4 \rangle$. Die Knoten sind mit den Elementen der Folgen beschriftet, oben links davon jeweils deren Indizes.

Analog zu *lrm* definieren wir *von-links-nach-rechts-Maxima (lrM)*, *von-rechts-nach-links-Minima (rlm)*, *von-rechts-nach-links-Maxima (rlM)* und die entsprechenden Bäume. Abbildung 5 zeigt beispielhaft einen *lrm*- sowie einen *lrM*-Baum.

5.2 Aufbau der Datenstruktur

Wir konstruieren den *lrm*-Baum T_{lrm}, den *lrM*-Baum T_{lrM}, den *rlm*-Baum T_{rlm} sowie den *rlM*-Baum T_{rlM} für die Folgen der Minima bzw. Maxima (in beide Richtungen) der τ Blöcke, in die B unterteilt ist. Die Minima und Maxima können jeweils aus den Wurzeln der *rmM*-Bäume der einzelnen Blöcke abgelesen werden.

Einer Kante (j, j_1) in T_{lrm} weisen wir ein Gewicht $m_j - m_{j_1}$ zu (zur Erinnerung: es ist $m_j < m_{j_1}$). Da jeder Block w^c Bits groß ist, betragen diese Gewichte jeweils höchstens w^c. Wir erweitern den Baum um eine Datenstruktur zur Beantwortung *gewichteter level-ancestor*-Anfragen, die auf dem *Ladder*-Algorithmus [5] basiert, hier jedoch nicht näher beschrieben wird (eine detaillierte Beschreibung befindet sich in [1]). Sie liefert für einen Knoten v den Vorfahren v' in konstanter Zeit, so dass der Pfad zwischen v und v' ein bestimmtes Gesamtgewicht besitzt. Dies werden wir in Unterabschnitt 5.3 nutzen. Die Datenstruktur benötigt einen Speicherplatz von $\mathcal{O}(\frac{n \log^2 n}{w^c} + \frac{n\, c^c}{\log^c n} + n^{3/4})$ Bits.

Für die anderen Bäume verfahren wir analog.

5.3 Beantwortung von Anfragen

Wir betrachten nun lediglich die Beantwortung von fwd_search(B, i, d) mit Hilfe der Bäume T_{lrm} und T_{lrM}. Die Beantwortung von *bwd_search* funktioniert analog mit T_{rlm} und T_{rlM}.

Sei $j = \lfloor \frac{i}{w^c} \rfloor$ der Block, in dem sich die Startposition i befindet. Zunächst führen wir die Suche innerhalb des Blocks wie in Unterabschnitt 4.3 aus und geben die Antwort, falls sie

sich dort befindet, nach konstanter Zeit aus.

Ansonsten suchen wir den *globalen Zielwert* $d' = e_{j-1} + \mathrm{sum}(B, \pi, 1, i-1-w^c \cdot (j-1)) + d$ in den folgenden Blöcken. Dabei lesen wir e_{j-1} aus der Wurzel des rmM-Baums für den Block $j-1$ ab. Die Summe wird lokal im Block j in konstanter Zeit berechnet (das Intervall liegt also relativ zum Beginn des Blocks j). Die gesuchte Antwort liegt dann im ersten Block $r > j$, so dass $d' \in [m_r, M_r]$.

Das Problem ist es nun also, die Suche nach $r > j$ mit $m_r \leq d'$ bzw. $M_r \geq d'$ in einer Folge von τ Werten (den Minima bzw. Maxima der Blöcke) in konstanter Zeit durchzuführen. Wir betrachten ersteren Fall und suchen $r > j$ mit $m_r \leq d'$; für den anderen Fall verfahren wir analog. Wir ziehen den *lrm*-Baum T_{lrm} hinzu und suchen darin den ersten Vorfahren j_r des Knotens j, so dass die Summe der Kantengewichte auf dem Pfad $d'' = m_j - d'$ übersteigt (zur Erinnerung: das Gewicht einer Kante ist die Differenz der Minima der verbundenden Blöcke). Diesen finden wir durch die in Unterabschnitt 5.2 beschriebene *gewichtete level-ancestor*-Anfrage in konstanter Zeit und somit den Block r. Innerhalb des Blocks r kann die finale Antwort nun ebenfalls in konstanter Zeit ermittelt werden.

6 Fazit

Bei einer Wortgröße von w Bits im RAM-Modell können für eine Konstante $c \geq 1$ und einen Bitvektor B der Länge $n < w^c$ die Anfragen *sum*, *fwd_search* und *bwd_search* in $\mathcal{O}(c)$ Zeit beantwortet werden. Hierzu wird der rmM-Baum mit $\mathcal{O}(\frac{nc \log w}{w})$ Bits sowie universelle Tabellen der Größenordnung $\mathcal{O}(\sqrt{2^w} w^2 \log w)$ Bits benötigt.

Bitvektoren der Größe $n > w^c$ werden in Blöcke der Größe w^c unterteilt und für jeden dieser Block ein rmM-Baum aufgebaut. Für Block-übergreifende Anfragen werden der *lrm*-, *lrM*-, *rlm*- und *rlM*-Baum aufgebaut und um eine Datenstruktur zur Beantwortung gewichteter *level-ancestor*-Anfragen erweitert. Diese benötigen $\mathcal{O}(\frac{n \log^2 n}{w^c} + \frac{n\, c^c}{\log^c n} + n^{3/4})$ Bits im Speicher.

Bei einer Wortgröße von $w = \log n$ Bits ergibt sich demnach insgesamt ein Speicherplatzbedarf von $\mathcal{O}(\frac{nc \log w}{w} + \sqrt{2^w} w^2 \log w + \frac{n \log^2 n}{w^c} + \frac{n\, c^c}{\log^c n} + n^{3/4}) = \mathcal{O}(\frac{n}{\mathrm{polylog}(n)})$ Bits für alle nötigen Datenstrukturen zur Beantwortung von *sum*, *fwd_search* und *bwd_search* in konstanter Zeit. Unter Berücksichtigung des BP-Strings B selbst kommen wir für einen Baum der beliebigen Größe n somit zum finalen Ergebnis von $2n + \mathcal{O}(\frac{n}{\mathrm{polylog}(n)})$ Bits Speicher für die vorgestellte Datenstruktur.

Literatur

1 Gonzalo Navarro und Kunihiko Sadakane: *Fully Functional Static and Dynamic Succinct Trees.* In: *ACM Transactions on Algorithms*, Seiten 16:1–16:39. ACM, 2014.

2 Guy Jacobson: *Space-efficient Static Trees and Graphs.* In: *30th Annual Symposium on Foundations of Computer Science*, Seiten 549–554. IEEE Computer Society, 1989.

3 J. Ian Munro und Venkatesh Raman: *Succinct Representation of Balanced Parentheses, Static Trees and Planar Graphs.* In: *38th Annual Symposium on Foundations of Computer Science*, Seiten 118–126. IEEE Computer Society, 1997.

4 Johannes Fischer: *Optimal Succinctness for Range Minimum Queries.* In: *Latin American Symposium on Theoretical Informatics*, Seiten 158–169. Springer Berlin Heidelberg, 2010.

5 Michael A. Bender und Martin Farach-Colton: *The Level Ancestor Problem simplified.* In: *5th Latin American Symposium on Theoretical Informatics*, Seiten 508–515. Springer Berlin Heidelberg, 2002.

Accelerating Parallel Suffix Tree Construction using Cartesian Trees

Michael Dominik Görtz[1]

1 **TU Dortmund University, Otto-Hahn-Straße 14, Dortmund, Germany**
dominik.goertz@tu-dortmund.de

Abstract

By providing efficient support for a wide variety of string operations suffix trees have become one of the most important data structures in the field of string processing. While search operations on this structure are very efficient, the construction of suffix trees has been relativly slow. To accelerate the construction of suffix trees, Shun et al. presented an algorithm that constructs a suffix tree out of a suffix array based on a Cartesian tree algorithm in polylogarithmic time. They compared their new algorithm with existing sequential and parallel algorithms in a variety of experiments, both for the full construction of suffix trees from given input texts as well as focussing on the construction of the underlying cartesian trees out of suffix arrays. They proved that their algorithm is able to construct suffix trees in linear work and polylogartihmic time while still maintaining the versatility of handling texts from arbitrary alphabets. In their experiments they showed that their solution outperforms previous implementations and delivers significant speedups of up to 40 fold in certain workloads. While their implementation has not been very refined it proves to be an important change in direction for the efficient parallelized construction of suffix trees.

1998 ACM Subject Classification D.1.3 Concurrent Programming

Keywords and phrases suffix trees, cartesian trees, performance, string processing

Digital Object Identifier 10.4230/LIPIcs.BADS.2017.12

1 Introduction

Over the last 20 years the data sizes that are processed by computers every day have grown more and more. One of the most important type of workloads in modern data analysis is string processing, most often in the form of searching within large amounts of text. To accommodate this growing demand for high performance string processing techniques a whole research field has formed and produced a number of different approaches to various tasks in the string processing spectrum. While there have been many specialized algorithms that are able to solve specific problems very well there is one data structure that is able to provide very good performance and efficiency values for most of the typical processing types. A *suffix tree*, a data structure that stores all suffixes of a given text in the form of a tree, is able to speed up text searches, searches for longest common substrings, repetitions, palindromes and many more [5].

However, while the search within a given suffix tree provides great performance, the construction of these trees has traditionally been a costly procedure. The majority of suffix tree construction algorithms have been of a sequential nature and did not make use of the increasing performance of multiprocessor systems while the existing parallel construction algorithms were not suitable for efficient implementations. In order to overcome this lack of performance in the construction of suffix trees, Blelloch and Shun introduced a multi-stage

approach that makes use of existing techniques for generating suffix arrays [11]. These are then processed into suffix trees by generating a cartesian tree using the longest common prefixes between adjacent entries in the array.

2 Background

A basic data structure for efficiently saving multiple strings is the *prefix tree* also known as *trie*. It contains an edge for each letter of a word, forming the word layer by layer. Words that share common prefix share their path in the tree for the length of this prefix. As displayed in Fig. 1 each path from the root to a leaf in the *trie* corresponds to a stored word. E.g. the words `work` and `world` share their first three edges in the tree because their longest common prefix `wor` is three letters long.

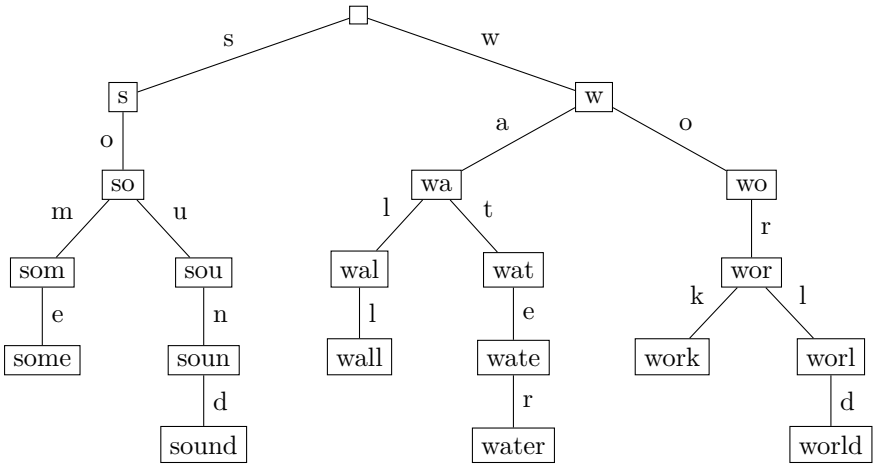


Figure 1 A prefix tree containing the strings `some`, `sound`, `wall`, `water`, `work` and `world`.

A *patricia trie* is a trie structure that compresses branch free paths and thus provides significant savings in required memory space as well as quicker access times since fewer nodes have to be traversed during search operations. If subsequent edges do not have a branch between them they are contracted into one edge. E.g. the three edges "u","n" and "d" of the word "sound" are contracted into one edge called "und". For the given example the edge count is reduced from 19 to 10 edges leaving little over half the original memory size required.

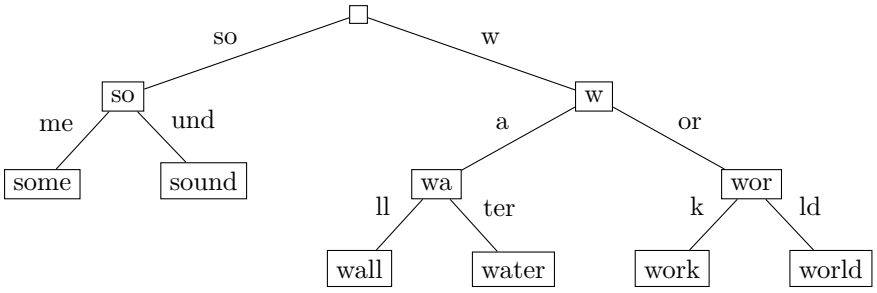


Figure 2 The patricia trie representation of the prefix tree in Fig. 1. Branch free paths are contracted into one edge.

A *suffix tree* is an advanced data structure that uses a patricia trie to save all suffixes of a given string and their position within the string. They can be used for pattern matching

in large texts and are especially efficient when the same text is searched multiple times since they only have to be generated once for a given text and can then be reused for all further search operations. Fig. 4 shows an example *suffix tree* containing all suffixes of the string "abacaba".

Suffix	ix_{start}	len_{LCP}
a	6	1
aba	4	3
abacaba	0	1
acaba	2	0
ba	5	2
bacaba	1	0
caba	3	

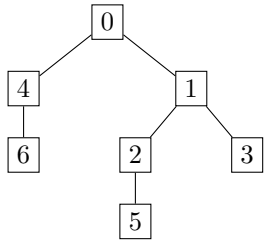


Figure 3 All prefixes of the string "abacaba" lexicographically ordered in a suffix array with the lengths of the LCP shared with each next element (left) and the cartesian tree generated from the suffix array (right)

Cartesian trees on the other hand are tree structures which represent elements of a sequence, enforcing a heap order with increasing values of the nodes from the root of the tree to the leaves. The order of the underlying sequence is represented by an in-order traversal of the tree. As displayed on the right in Fig. 3 the values of the nodes decrease towards the trees root.

A different approach to saving suffix information about a given text is the *suffix array*. Similar to the suffix tree it holds pointers to the suffixes of the text. All suffixes of the input string are saved sorted by lexicographic order. Since every suffix can be addressed by its index within the input string, only the indexes are saved in order to save memory space. As displayed in Fig. 3, the suffix array of the string "abacaba" is `{6, 4, 0, 2, 5, 1, 3}`. In addition to these suffix arrays, Blelloch and Shun use the lengths of the *longest common prefixes* between adjacent elements within the array as an additional meta information about the suffixes in order to generate suffix trees.

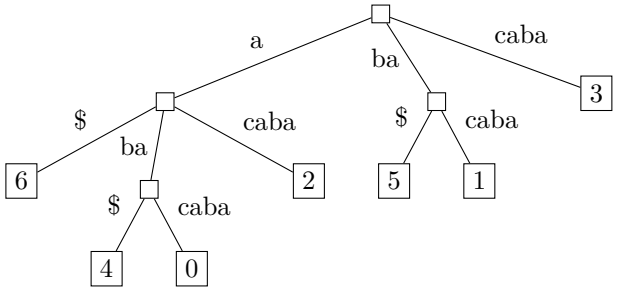


Figure 4 Suffix tree representing the suffix array of the example in Fig. 3. `$` symbols represent the end of suffixes that are also the prefixes of other suffixes.

There are different machine models used in order to categorize algorithms and their performance or their efficiency respectively. The most common one is the *random access machine* (RAM) which models a compute unit that can access any memory location at any given time but does not support concurrency, i.e. only one thread is executed. For modern multi-core computers there are various *parallel random access machine* (PRAM) models. They differ in regards to their specific behavior when memory is accessed by two or more

threads at once. E.g. concurrent read exclusive write (CREW) PRAMs allow multiple threads to read at the same location but only one thread is allowed to write to it at any given time. CRCW PRAMs on the other hand provide *concurrent write* capabilities, enabling multiple threads to write to the same memory location at the same time. Write collisions are then resolved depending on the specific model. In the context of this work it is assumed that write collisions are solved by randomly choosing a value from the colliding write operations.

For the complexity evaluation of parallelized algorithms there are different aspects of complexity. The usual metrics are the complexity for the *work*, *time* and *space* required by the algorithm. The *space* complexity covers the memory required to run the algorithm relative to the input size. The *time* complexity represents the time needed to solve the problem with the given algorithm relative to the input size. For parallel algorithms this time usually differs from the required *work* because multiple processors running in parallel are able to perform twice the work in the same time. I.e. a parallel algorithm may be able to do the same work like a sequential algorithm in a fraction of the time.

3 Related Work

The idea of suffix trees was first presented by Peter Weiner in 1973 along with a sequential algorithm for construction of these trees [12]. He proposed that a compressed *binary tree* can be used to store suffix information for fast access times supporting searches within texts. While Weiner's idea was revolutionary at its time it received many revisions in the next decades. In 1976 McCreight introduced a modification that moved from binary trees as an underlying data structure to trees that can have an arbitrary number of children per node. The children of each node are saved in a hash table to maintain constant access times to child nodes when traversing through the tree [10].

Due to the design of the computers of that time, the algorithms for constructing suffix trees were completely sequential. More than a decade later Apostolico et al. presented the concept of a parallel suffix tree construction algorithm [1]. Although the algorithm was fairly simple it did not provide the required efficiency. While Hariharan was able to improve the time needed for constructing a suffix tree to $O(\log^4 n)$ [6] and Farach et al. achieved a work optimal solution with a time efficiency of $O(\log n)$ [4] both solutions are complicated and hard to optimize for efficient implementations.

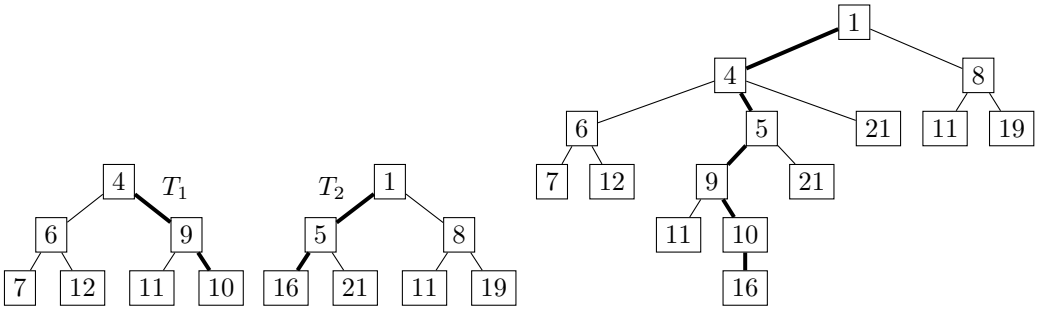
4 Parallel Construction of Cartesian Trees

In order to synthesize the suffix information and to make it more accessible Blelloch and Shun use an existing suffix array algorithm. While these suffix arrays only hold all suffixes of a given text in lexicographic order they make a good starting point for generating suffix trees from them.

First of all, the *longest common prefixes* (LCPs) are calculated for all adjacent elements in a suffix array and their lengths are saved to an additional array. Afterwards the lengths of the suffixes in the suffix array are written to an array which is then combined with the LCP lengths, interleaving them to form a new sequence. This sequence can now be used as a data base to construct a cartesian tree which is going to provide the structure for the resulting suffix tree. For the example in Fig. 3 the resulting array of interleaved suffix lengths and LCP lengths is `{1,1,3,3,7,1,5,0,2,2,6,0,3}`.

While the construction of the suffix array and the calculation of the longest common prefix have been problems that have already been solved in an efficient way, requiring linear

time and work, the construction of cartesian trees has been relatively inefficient. Because of this Blelloch and Shun introduce a parallel algorithm for constructing a cartesian tree from a given sequence of values. Their devide-and-conquer algorithm recursively splits up the input sequence into halfs and generates the cartesian tree for each half, merging them afterwards in order to create the full tree. While the splitting of an array is a trivial task, the merging of two cartesian trees is not.



(a) Example trees T_1 and T_2. The thicker edges are the spines the merging algorithm works along.

(b) T_1 and T_2 merged into one tree. The thick edges represent the path the algorithm took to weave the two trees together.

Figure 5 Example for the merge process of two cartesian trees T_1 and T_2. This is the last merge step for the cartesian tree construction from the input array `{7,6,12,4,11,9,10,16,5,21,1,11,8,19}`

In order to merge two cartesian trees Blelloch and Shun only use the spines of both trees. The spine of a tree is made up of the nodes on the way from the root of the tree to the leftmost or rightmost leave element. The way to the left leave forms the left spine of the tree and the path to the right leave represents the right spine. When merging two cartesian trees the spines facing each other have to be merged, weaving the two trees together as one bigger tree.

The generated cartesian trees are only connected via parent pointers in the nodes. Since every node only has one parent at a time this results in a linear space requirement. The merging of two trees starts with the two nodes at the bottom of the two adjacent spines. From there on, the nodes of the two spines are joined and sorted, changing each parent pointer to the node with the next smaller value. As shown in Fig. 5a and 5b the nodes of the two spines are woven together and ordered, descending in value towards the root of the resulting tree.

This process is similar to the way Iliopoulos and Rytter describe for merging cartesian trees, but while their algorithm traverses over the full length of the spines and is based on suffix arrays, Blelloch and Shun's solution only traverses the nodes up to the first root of the two spines that is reached [7]. This leads to a significant reduction of the nodes that have to be merged since the rest of the other spine does not have to be reordered and is left intact.

Blelloch and Shun's code for their semi parallelized algorithm 1a is shown in Alg. 1. The `cartesianTree` method splits up the given input array and starts parallel recursive calls for both halfs. The resulting cartesian trees are then merged together again using the rightmost node from the left half and the leftmost node from the right half as starting nodes for the spines (Lines 21–24).

For the `merge` process, the algorithm starts at the leaf nodes of the two trees' facing spines. The leaf with the higher value is selected as the initial `head` of the merging path and the pointer for the next parent candidate in the tree (`left` or `right`) is set to the parent

```
1  struct node { node* parent; int value; };
2  void merge ( node* left , node * right ) {
3    node* head;
4    if ( left->value > right->value ) {
5      head = left;
6      left = left->parent; }
7    else { head = right;
8      right= right ->parent; }
9    while ( 1 ) {
10     if ( left == NULL) { head->parent = right; break; }
11     if ( right == NULL) { head->parent = left; break; }
12     if ( left->value > right->value ) {
13       head->parent = left; left = left ->parent; }
14     else { head->parent = right;
15       right = right->parent; }
16     head = head->parent;
17   }
18 }
19 void cartesianTree ( node* Nodes , int n ) {
20   if ( n < 2 ) return;
21   cilk_spawn cartesianTree ( Nodes , n/2 );
22   cartesianTree ( Nodes+n/2 , n-n/2 );
23   cilk_sync;
24   merge ( Nodes+n/2 -1 , Nodes+n/2 ); }
```

Listing 1 Algorithm 1a of Blelloch and Shun written in C. The *cilk++* extension calls are used for parallel recursive execution. From [3].

node of the initial `head`.(Lines 4–8) From then on the two parent candidates from the left and right spine are compared and the `head` of the merge path is set to the one with the higher value, i.e. the node that is placed lower in the new path (Lines 12–16). This way the `head` bubbles up until it reaches the end of one of the two spines (Lines 10–11). On the way up the parent pointers of the visited nodes are always set to the new `head` node for the next iteration, forming a new path along the two spines, weaving them together as one (Line 16).

Additionally Blelloch and Shun describe a modified version `1b` of their merging algorithm. While the splitting of the suffix array is done in parallel, the merging of the two cartesion trees along their spines is still performed sequentially when using algorithm `1a`. The modified version makes use of binary search trees for each of the two spines. It determines which of the two spines has the root with the higher value and splits the other spine at this value. The first spine is then recursively merged with the lower half of the split spine and afterwards the upper half is added to it without any changes since there are no values for the first spine that need to be woven in anymore.

This provides a fully parallelized algorithm for the construction of the cartesian tree out of the interleaved suffix and LCP lengths. Furthermore Blelloch and Shun are able to prove that their algorithm 1a requires $O(n)$ work and $O(n)$ memory space on a (sequential) random access machine and their algorithm 1b is able to generate a cartesian tree with requirements of $O(n)$ work, $O(\log^2 n)$ time and $O(n)$ space on a CREW PRAM. With these requirements the algorithms are both great improvements over existing approaches to the construction of cartesian trees.

5 Enhancing a Cartesian Tree to a Pat Trie

After generating the cartesian tree out of the interleaved suffix array and the LCPs the structure for the finished suffix tree is created but it needs to be filled with the parts of the suffixes corresponding to the structure. These substrings from the suffixes are added to the edges of the cartesian tree, effectively enhancing it to a pat tree.

The information needed to add labels to the edges of the cartesian tree is already encoded within the tree itself. Since the tree is generated from the suffix lengths and the longest common prefixes of the suffixes in the suffix array the labels for each edge can be calculated using the nodes it connects. Blelloch and Shun point out that for a node with a value $v = LCP(s_i, s_{i+1})$ and parent value v' the edges label can be extracted from the suffix array as the substring of the element s_i starting from the position $v' + 1$ and ending at position v.

With the addition of the suffix substrings from the suffix array using the LCP and suffix length information the construction of the suffix tree is finished. Since every node only needs to be visited once for a full labeling of the trees edges, this task can be completed with linear work and within logarithmic time when the process is parallelized.

All in all the algorithms of Blelloch and Shun can be used to construct a suffix tree from a suffix array in linear work and polylogarithmic time.

6 Evaluation

During their evaluation Blelloch and Shun compared their own algorithms to existing algorithms and implementations for the construction of cartesian trees and suffix trees. They measured the execution times for both categories and examined the influence of different parameters, e.g. processor count and input size, on the performance of their algorithms.

6.1 Experimental Setup

The test platform used for their experiments was a 32-core Dell PowerEdge 910 equipped with 4 Intel Xeon X7560 Processors from the Nehalem Family running at 2.26GHz clock speed, providing 24MB L3 Cache. The system was fitted with 64GB of main memory. All parallel algorithms were compiled with the `cilk++` compiler build 8503 and the `-O2` optimization level.

6.2 Reference Algorithms and Additional Code

Since the goal of Blelloch and Shun's work was to create an algorithm for converting a given suffix array into a suffix tree, there is the need for an algorithm that is able to generate these suffix arrays from an input text in the first place. For this task they implemented a parallel version of the skew algorithm by Karkkainen et al. [8] and optimized it using a approach to the calculation of the LCPs that differs from the original version. With this modification they were able to achieve a significant performance gain over the original version by Karkkainen et al.

Futhermore Blelloch and Shun implemented an additional suffix tree construction algorithm by Berkman et al. [2] which is based on the *all nearest smaller values* problem (ANSV) and makes use of parallelization over many cores similar to their own algorithm. With small optimizations they managed to achieve significant performance improvements over the original version of Berkman et al.

Text	Size (MB)	Kurtz	Alg 1a 32-core	Alg 1a seq.	Alg 1a Speedup	Alg 2 32-core	Alg 2 seq.	Alg 2 Speedup
10Midentical	10	1.04	0.46	4.34	9.4	1.32	9.64	7.66
10Mrandom	10	10.6	0.39	7.58	19.4	1.01	11.9	11.8
$(a^{\sqrt{10^7}}b)^{\sqrt{10^7}}$	10	1.69	0.54	5.78	10.7	1.36	11.0	8.1
chr22.dna	34.6	31.1	2.23	37.4	16.8	5.22	53.9	10.3
etext99	105	128	8.71	135	15.5	15.7	191	12.2
howto	39.4	33.1	2.90	45.8	15.8	5.88	64.1	11.0
jdk13c	69.7	18.1	5.30	90.2	17.0	11.0	125	11.6
rctail96	115	65.5	9.54	153	16.0	17.1	216	11.4
rfc	116	89.6	8.96	150	16.7	17.4	220	12.6
sprot34.dat	110	89.8	9.42	143	15.2	15.2	202	13.3
thesaurus.doc	11.2	10.7	0.72	9.77	13.6	1.39	14.4	10.4
w3c2	104	33.4	8.87	138	15.6	15.8	201	12.7
wikisamp8.xml	100	35.5	8.42	128	15.2	13.5	173	12.8
wikisamp9.xml	1000	—	118	1840	15.6	—	—	—
10MrandomInts10K	10	—	0.44	7.74	17.6	0.91	10.1	11.1
10MrandomIntsImax	10	—	0.41	5.96	14.5	0.73	8.18	11.2

Table 1 Execution times for suffix tree construction with different algorithms and input files. From [3].

To compare their algorithm with existing sequential implementations Blelloch and Shun use Stefan Kurtz' suffix tree code from the MUMmer project [9]. This algorithm follows a completely different paradigm and is purely sequential, thus providing a very good comparison for the performance on systems with small processor counts.

6.3 Test Data

To conduct their experiments the authors had to find suitable test data with varying sized and content patterns. They used generated inputs with and without randomness, texts from different online sources, a word document (thesaurus.doc) and some of Wikipedia's XML samples. While their work was originally based on integer alphabets, any arbitrary alphabet can be used since it can be mapped to an integer alphabet by assigning each element of the alphabet a number.

6.4 Performance of the partially parallelized cartesian tree algorithm

Measuring the execution times of their algorithm 1a Blelloch and Shun found their algorithm to perform very well. Figure 6 shows the breakdown of the execution times of algorithm 1a for different inputs for the execution on 32 cores with hyper-threading. Their multiway-cartesian tree algorithm only takes up about 5% of the total execution time for all input files while the construction of the underlying suffix array takes 85% of the total execution time.

The suffix tree construction times on random character files shown in Fig. 6 are superlinear to the input file size with increasing execution times towards very big file sizes due to memory effects.

As expected the parallel merging implementation by Blelloch and Shun was able to provide a significant speedup in comparison to the sequential implementation by Stafan Kurtz. As displayed in Fig. 7 (left) the parallel algorithm is faster from 4 processor cores and more providing a speedup compared to the sequential implementation of up to 10 fold. While

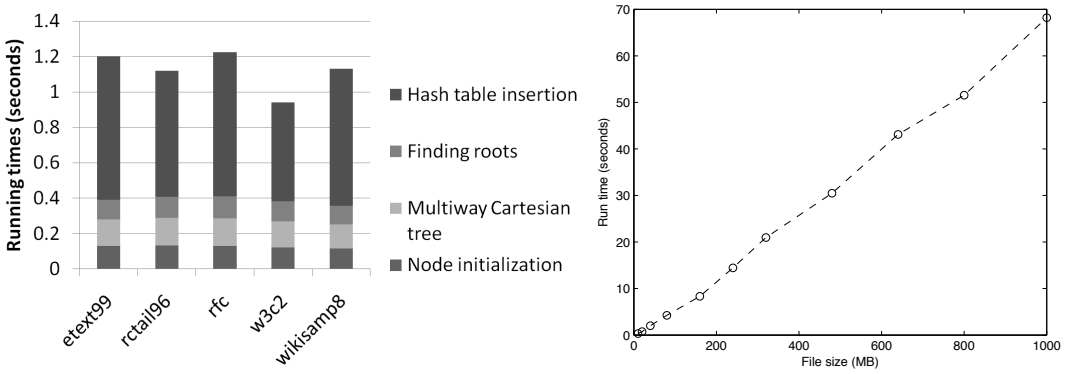


Figure 6 Breakdown of the execution times of Algorithm 1a (left). Construction times of Algorithm 1a on random character files depending on the file size (right). From [3].

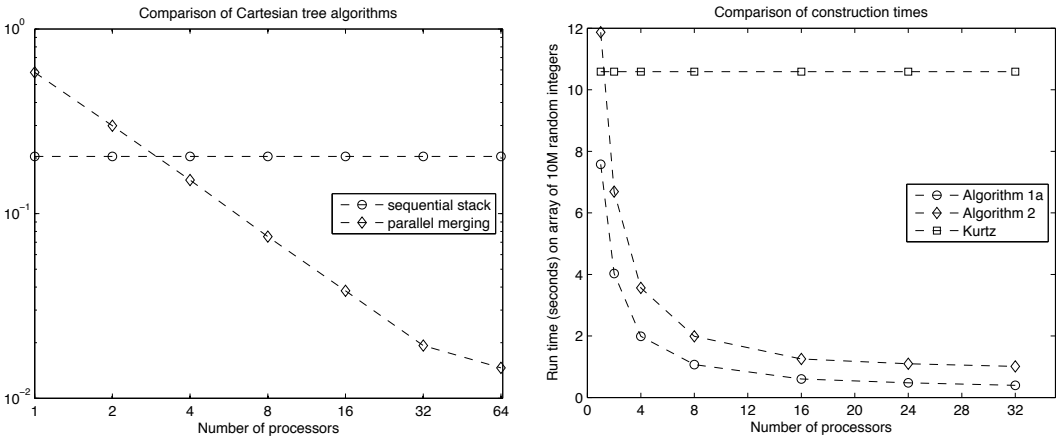


Figure 7 Execution times with different numbers of cores for the cartesian tree algorithms (left) and for different suffix tree construction algorithms (right). From [3].

6.5 Suffix Tree Construction Performance

When comparing their algorithm with the ANSV algorithm and Kurtz' sequential version Blelloch and Shun found that their implementation was able to achieve significant improvements over the other solutions. Even with only one core they were able to outperform both the parallel ANSV algorithm as well as the sequential algorithm already providing a speedup of 20 to 30%. Fig. 7 shows that with increasing core count the speedup to the single core algorithm increases and is up to 27 fold faster.

In comparison to the ANSV Blelloch and Shun's implementation of their algorithm 1a is able to provide between 20 and 60% speedup as displayed in Table 1. While this advantage is heavily dependant on the nature of the input files their algorithm is without a doubt superior to the ANSV approach.

7 Conclusion

In their paper Blelloch and Shun present algorithms for the partially as well as fully parallelized construction of cartesian trees based on given suffix arrays with their LCPs. Their solutions

are able to create suffix trees within polylogarithmic time and with linear work and heave use of existing research in the field of suffix array construction. As such they provide a modular approach that is simple to implement and has a high potential for optimizations both through improvements in their own algorithm stages as well as trough the improvement of suffix array construction algorithms. They are able to prove that their algorithms adhere to their specified time and work constraints and show through experimental evaluation that both of their algorithms provide a significant improvement over the existing suffix tree construction algorithms.

8 Acknowledgements

This work is a seminar paper based on the article "A simple parallel cartesian tree algorithm and its application to suffix tree construction" by Guy E. Blelloch and Julian Shun [3] and their paper "A simple parallel cartesian tree algorithm and its application to parallel suffix tree construction" [11].

References

1 Alberto Apostolico, Costas Iliopoulos, Gad M. Landau, Baruch Schieber, and Uzi Vishkin. Parallel construction of a suffix tree with applications. *Algorithmica*, 3(1):347–365, 1988.

2 Omer Berkman, Baruch Schieber, and Uzi Vishkin. Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values. *Journal of Algorithms*, 14(3):344–370, 1993.

3 Guy E Blelloch and Julian Shun. A simple parallel cartesian tree algorithm and its application to suffix tree construction. In *2011 Proceedings of the Thirteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 48–58. SIAM, 2011.

4 Martin Farach-Colton, Paolo Ferragina, and Shanmugavelayutham Muthukrishnan. On the sorting-complexity of suffix tree construction. *Journal of the ACM (JACM)*, 47(6):987–1011, 2000.

5 D Gussfield. Algorithms on strings, trees, and sequences. *Computer Science and Computional Biology (Cambrigde, 1999)*, 1997.

6 Ramesh Hariharan. Optimal parallel suffix tree construction. In *Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, pages 290–299. ACM, 1994.

7 Costas Iliopoulos and Wojciech Rytter. On parallel transformations of suffix arrays into suffix trees. In *Proceedings of the 15th Australasian Workshop on Combinatorial Algorithms (AWOCA'04)*, 2004.

8 Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In *ICALP*, volume 2719, pages 943–955. Springer, 2003.

9 Stefan Kurtz. Reducing the space requirement of suffix trees. *Software-Practice and Experience*, 29(13):1149–71, 1999.

10 Edward M McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM (JACM)*, 23(2):262–272, 1976.

11 Julian Shun and Guy E Blelloch. A simple parallel cartesian tree algorithm and its application to parallel suffix tree construction. *ACM Transactions on Parallel Computing*, 1(1):8, 2014.

12 Peter Weiner. Linear pattern matching algorithms. In *Switching and Automata Theory, 1973. SWAT'08. IEEE Conference Record of 14th Annual Symposium on*, pages 1–11. IEEE, 1973.

Hollow-Heaps

Jens Knipper 159585

Department of Computer Science, TU Dortmund, Germany,
`jens.knipper@tu-dortmund.de`

Abstract

Im Folgenden möchte ich den Hollow-Heap vorstellen, eine Spezialform des Fibonacci-Heaps. Ziel dieser ist es, die Operationen auf der Datenstruktur zu vereinfachen. Namensgebend für diese Form des Heaps sind die sogenannten leeren „hollow“ Knoten. Sie sind einer der markantesten Unterschiede zum klassischen Fibonacci-Heap. Als Einführung in diese Ausarbeitung sollen Fibonacci-Heaps grundlegend erläutert werden. Anhand der Unterschiede wird daraufhin zunächst die Struktur des *Multi-root Hollow-Heaps* erläutert. Verfeinerungen, wie der *One-Root Hollow-Heap* und der *Two-Parent Hollow-Heap* sollen nicht besprochen werden. Der Fokus dieser Arbeit liegt darin, die Funktionsweise und Stärken der beiden verwandten Heap-Strukturen herauszustellen, sowie deren Unterschiede und die damit verbundenen Auswirkungen auf das Verhalten und die Simplizität zu verdeutlichen. Desweiteren sollen die Methoden der Datenstruktur erläutert werden und markante Funktionen, wie zum Beispiel *delete-min* und *decrease-key* gesondert hervorgehoben werden.

1998 ACM Subject Classification E.1 Data Structures

Keywords and phrases Datenstrukturen, Heap, Fibonacci-Heap, Hollow-Heap

Digital Object Identifier 10.4230/LIPIcs.BADS.2017.5

1 Einleitung

Prioritätswarteschlangen sind eine abstrakte Datenstruktur, welche die Methoden *insert*, *delete-min* und *decrease-key* implementieren. Die Methoden ermöglichen das Einfügen eines Elementes mit einem Schlüssel, sowie das Löschen des Elementes mit minimalem Schlüssel in der Datenstruktur. Die dritte Operation erniedrigt den Schlüssel eines Elementes.
Eine Datenstruktur dieser Art, erlaubt eine besonders simple Art der Sortierung, durch Einfügen der Elemente und schrittweises extrahieren des Minimums.
Der Fibonacci-Heap ist eine Prioritätswarteschlange und wurde von Fredman und Tarjan im Jahre 1987 vorgestellt [1]. Im Gegensatz zu binären Heaps erlaubt die Struktur das Einfügen von Elementen in konstanter Zeit. Zudem ist das Löschen des Minimums in amortisierter Zeit von $\Theta(\log n)$ möglich. Das Erniedrigen des Schlüssel eines Elementes ist in amortisiert konstanter Zeit möglich. Dadurch ermöglichen Fibonacci-Heaps eine schnelle Implementierung von Greedy-Algorithmen, wie z.B. Dijkstras Kürzester-Wege Algorithmus. Darauf aufbauende Probleme, wie das *All Pair Shortest Path*-Problem, können durch diesen Fortschritt ebenfalls schneller gelöst werden.
Der Hollow-Heap ist eine auf dem Fibonacci-Heap beruhende Datenstruktur, welche seinen Vorgänger in der Einfachheit überbietet. Er wurde erstmals im Jahre 2015 vorgestellt. Ziel der Autoren ist es, die Datenstruktur bei gleichbleibender Performanz zu vereinfachen [2]. Namensgebend ist das verzögerte Löschen in der Datenstruktur, welches leere (bzw. hohle: engl. hollow) Knoten ermöglicht. Auf Basis des Hollow-Heaps mit leeren Knoten, wurden in der Veröffentlichung weitere Datenstrukturen vorgestellt, die Fibonacci-Heaps vereinfachen sollen. Im Rahmen dieser Arbeit soll allerdings nur die Erste der Datenstrukturen eine

größere Beachtung finden.
Im Verlauf dieser Arbeit sollen beide Heap-Strukturen vorgestellt und miteinander verglichen werden. Die expliziten Unterschiede sollen erläutert und auf ihre Wirkung hin analysiert werden.

2 Fibonacci-Heaps

Fibonacci-Heaps sind Prioritätswarteschlangen, die neben den drei bereits bekannten Funktionen noch die Methoden *make-heap*, *find-min*, *meld* und *delete* unterstützen. Diese dienen dazu einen leeren Heap zu erstellen, das Minimum zu suchen, zwei Heaps zu verknüpfen, sowie ein Element zu löschen. In Tabelle 1 sind die genannten Operationen, samt ihrer Parameter und Ausgaben, noch einmal genauer aufgeführt. Die Operationen sind bei Fibonacci- und Hollow-Heap identisch, werden allerdings teilweise unterschiedlich umgesetzt, wobei das Verhalten und die Bezeichnungen jeweils analog sind.

Operation	Beschreibung
make-heap()	Erstellt einen neuen, leeren Heap und gibt diesen zurück.
find-min(h)	Gibt das Element mit minimalem Schlüssel aus dem Heap h zurück. Ist der Heap leer, wird *null* zurückgegeben.
insert(e,k,h)	Rückgabe des Heaps h mit dem zusätzlichen Element e, welches den Schlüssel k enthält. Das Element darf sich in keinem weiteren Heap befinden.
delete-min(h)	Gibt den zuvor nicht leeren Heap h zurück, aus welchem das Element gelöscht wird, welches *find-min(h)* zurückgibt.
meld(h_1,h_2)	Gibt einen Heap zurück, welcher alle Elemente der Heaps h_1 und h_2 enthält.
decrease-key(e,k,h)	Falls Element e im Heap h enthalten ist und der Schlüssel größer als k ist, dann wird der Schlüssel von e auf k gesetzt (Beim Fibonacci-Heap wird der Schlüssel des Elements e um den übergebenen Wert k erniedrigt). Anschließend wird der Heap zurückgegeben.
delete(e,h)	Gibt einen Heap zurück, welcher dadurch gebildet wird, dass das Element e aus dem Heap h entfernt wird.

Table 1 Übersicht der Methoden von Fibonacci- und Hollow-Heaps.

Fibonacci-Heaps sind Bäume, die entsprechend der Heap-Bedingung aufgebaut sind. Daher gilt die Bedingung, dass der Schlüssel eines Knotens immer kleiner sein muss als die Schlüssel seiner Kinder. Ein einzelner Knoten des Fibonacci-Heaps ist wiederum ein eigenständiger Fibonacci-Heap für den die Heap-Bedingung gilt. Ein solcher Knoten enthält unter anderem einen Schlüssel, einen Rang und eine Markierung. Der Rang gibt an, wie viele Kinder der Knoten besitzt. Die Markierung wird im Zusammenhang mit der Methode *decrease-key* erläutert. Neben den Attributen ist ein Knoten durch eine doppelte Verkettung mit einem seiner Kinder verbunden. Diese sind wiederum durch eine doppelt verkettete zirkuläre Liste untereinander verbunden. Jedes Kind kennt seinen Elter-Knoten und enthält somit ebenfalls einen Zeiger auf diesen. Eine beispielhafte Darstellung der Verknüpfungen ist in Abbildung 1 zu finden. Die Wurzeln können als Kinder ohne Eltern interpretiert werden und sind somit ebenfalls untereinander durch eine doppelt verkettete zirkuläre Liste verbunden. Das kleinste Elemente der Wurzelliste wird genutzt um auf die Datenstruktur zuzugreifen. Der Zeiger auf das Element wird auch Min-Pointer genannt.

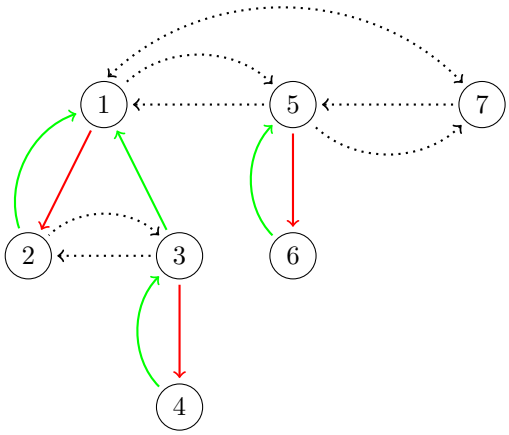


Figure 1 Verknüpfungen innerhalb eines Fibonacci-Heaps. Die grünen Kanten stellen die Verknüpfungen von Kindern zu Eltern dar. Die roten Kanten führen von Elter zu Kind und die schwarz gepunkteten stellen die Verknüpfungen von Kindern untereinander dar. Die drei Heaps unterscheiden sich anhand ihres Ranges.

In Abbildung 1 fällt auf, dass der Fibonacci-Heap aus mehreren Heaps besteht, welche sich durch den Rang ihrer Wurzel unterscheiden. Eine Erklärung dafür liefert die Methode *delete-min*. In dieser wird zunächst das kleinste Element aus der Wurzelliste entfernt, wodurch sämtliche Kinder des Elementes zu Wurzeln werden. Um die Datenstruktur nicht zu einer Liste entarten zu lassen muss nun die Baumstruktur wiederhergestellt werden. Dazu werden, wie in Abbildung 2 zwei Wurzeln gleichen Ranges miteinander verknüpft, indem das Element mit dem größeren Schlüssel Kind vom Element mit dem kleineren Schlüssel wird. Dies erhöht den Rang des kleineren Elementes um eins. Die Verknüpfungen werden solange durchgeführt, bis es keine zwei Elemente gleichen Ranges mehr gibt. Aus der Wurzelliste muss nun ein neues Minimum bestimmt werden und der Min-Pointer entsprechend gesetzt werden.

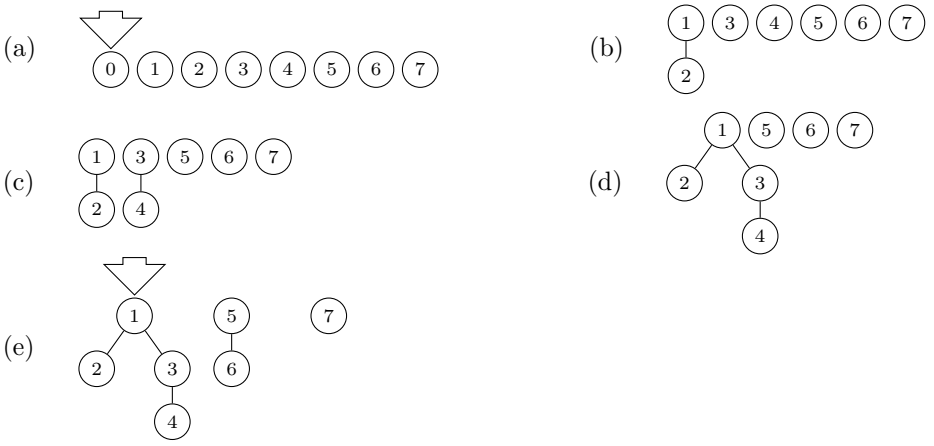


Figure 2 Ausführung der Methode *delete-min*. Verknüpfungen werden vereinfacht dargestellt. Ein Pfeil zeigt auf das kleinste Element. (a) Es wurden nacheinander Elemente in den Heap eingefügt. Danach erfolgen nur noch Schritte von *delete-min*. (b) Entfernen des Minimums und Verknüpfung zweier Wurzeln gleichen Ranges. (c) und (d) Verknüpfung zweier Wurzeln gleichen Ranges. (e) Es wurden zwei Wurzeln gleichen Ranges verknüpft wonach nur noch Wurzeln unterschiedlichen Ranges existieren. Das Minimum wurde in der Wurzelliste gesucht und der Min-Pointer gesetzt.

Um ein Element in einen Fibonacci-Heap einzufügen, wird ein leerer Heap mittels *make-heap* erstellt, in den daraufhin das entsprechende Element eingefügt wird. Die Operation *meld* verknüpft nun die beiden Heaps, indem der Wurzellisten miteinander verkettet werden.

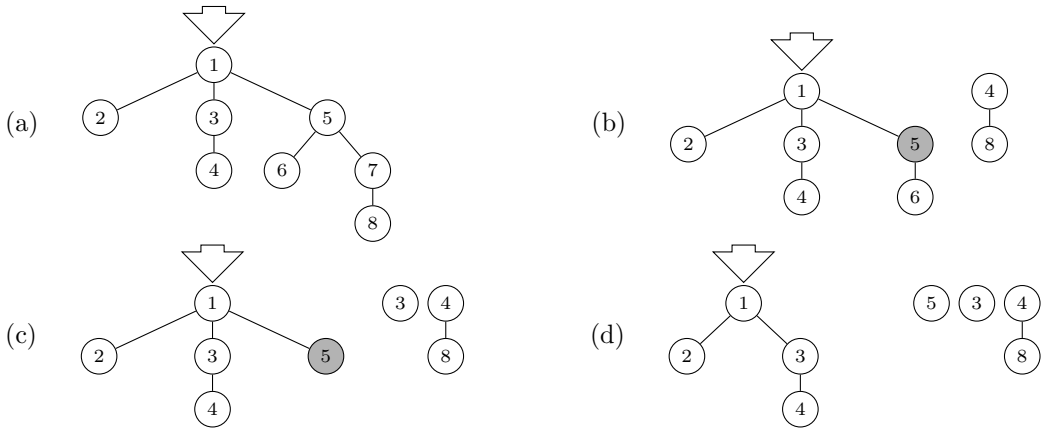


Figure 3 Ausführung der Methode *decrease-key*. Verknüpfungen werden vereinfacht dargestellt. Dunkel hinterlegte Elemente sind markiert. (a) Heap zu Beginn. (b) Heap, nachdem 7 auf 4 erniedrigt wurde. (c) 6 auf 3 erniedrigt, *decrease-key* noch nicht abgeschlossen. (d) Zweifach markierten Knoten 5 in Wurzelliste eingetragen.

Die Methode *decrease-key* erniedrigt den Schlüssel eines Elementes, indem es diesen auf einen übergebenen Wert setzt. Eine solche Operation kann die Heap-Bedingung verletzten, die daraufhin wiederhergestellt werden muss. Eine einfache Möglichkeit dies zu tun kann durch das Entfernen des veränderten Knotens erreicht werden. Er wird von seinem Elter-Knoten getrennt und in die Wurzelliste eingefügt. Man spricht dabei auch vom „Zerschneiden“ des Baumes. Beispielhaft ist dies in Abbildung 3 (b) ausgeführt. Dies kann allerdings zu einem sehr starken Ausdünnen des Baumes führen, weshalb das Konzept der Markierung eingeführt wird. Verliert ein unmarkierter Knoten durch *decrease-key* ein Kind, wird er markiert. Ein markierter Knoten, der ein weiteres Kind durch *decrease-key* verliert, wird ähnlich wie bei der Verletzung der Heap-Bedingung von seinem Elter-Knoten getrennt und in die Wurzelliste eingefügt. Danach wird die Markierung entfernt. Ein Beispiel dafür ist in 3 (c) und (d) zu finden. Dieses Verfahren kann zu einem kaskadierenden Zerschneiden des Baumes führen.

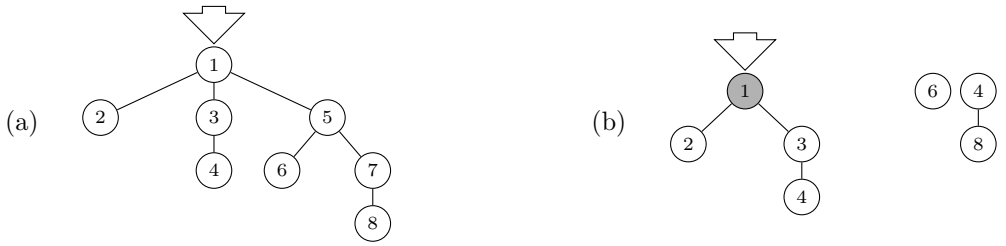


Figure 4 Ausführung der Methode *delete*. Verknüpfungen werden vereinfacht dargestellt. Dunkel hinterlegte Elemente sind markiert. (a) Der Heap zu Beginn. (b) Der Heap, nachdem der Knoten 5 gelöscht wurde. Die Kinder des Knotens wurden zu Wurzeln.

Die Operation *delete* funktioniert ähnlich, wie *decrease-key*. Lediglich wird hier der Knoten nicht erniedrigt, sondern gelöscht. Anschließend werden die Kinder des gelöschten Knotens der Wurzelliste angehängt und der Elter-Knoten des gelöschten Knotens markiert. Ist dieser bereits markiert, muss der Baum, wie in *decrease-key* zerschnitten werden.
Der Fibonacci-Heap erhält seinen Namen dadurch, dass ein Knoten mit Rang r mindestens F_{r+2} Nachfolger hat, wobei F_{r+2} die *r+2-te* Fibonacci-Zahl ist. Eine abgewandelte Bedingung gilt auch für die im folgenden Abschnitt erklärten Hollow-Heaps.

3 Multi-Root Hollow-Heaps

Multi-Root Hollow-Heaps sind eine Erweiterung der Fibonacci-Heaps, in denen Knoten ohne Inhalt erlaubt sind. Die leeren Knoten enthalten keinen Schlüssel, müssen aber traversiert werden können. Aus diesem Grund muss die Struktur des Heaps an die Gegebenheiten angepasst werden. Ein Knoten besteht nicht mehr, wie im Fibonacci-Heap aus Verknüpfungen zu Elter- und Kinder-Knoten, sowie den Attributen. Stattdessen werden die Attribute in ein Item ausgelagert, welches mit dem Knoten verknüpft ist. Dies ist vereinfacht in Abbildung 5 dargestellt. Einen leeren Knoten zu erstellen ist nun trivial. Es muss lediglich die Verknüpfung zu dem Item gelöscht werden. Die Datenstruktur erlaubt es nicht, leere Knoten erneut zu befüllen. Im Hollow-Heap ist es nicht nötig eine Markierung zu speichern. Die Gründe dafür werden im Zusammenhang mit der Methode *decrease-key* erläutert.
Das Löschen eines Knotens wird durch das Erlauben von leeren Knoten deutlich vereinfacht.

Figure 5 Vereinfachte Darstellung eines Knotens aus einem Hollow-Heap. (a) Der Knoten ist nicht leer und enthält ein Item mit den entsprechenden Attributen, wie dem Schlüssel. (b) Der Knoten ist leer und enthält kein Item mehr. Die Verknüpfungen zu Elter- und Kinder-Knoten bleiben erhalten.

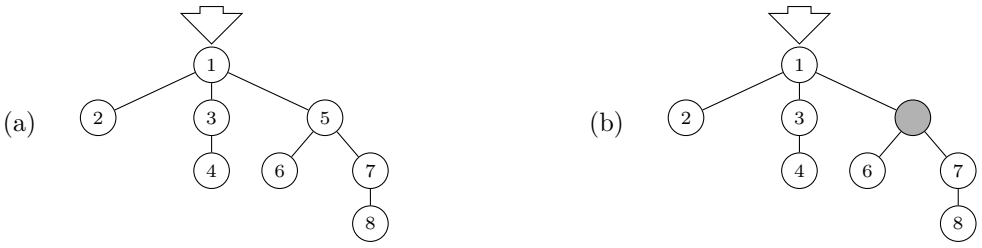


Figure 6 Ausführung der Methode *delete*. Verknüpfungen werden vereinfacht dargestellt. (a) Heap zu Beginn. (b) Heap nachdem 5 gelöscht wurde.

Es muss lediglich die Verknüpfung des Items zum Knoten aufgehoben werden. Die Operation hat keine weiteren Auswirkungen auf den Heap, wie auch in Abbildung 6 zu sehen ist.

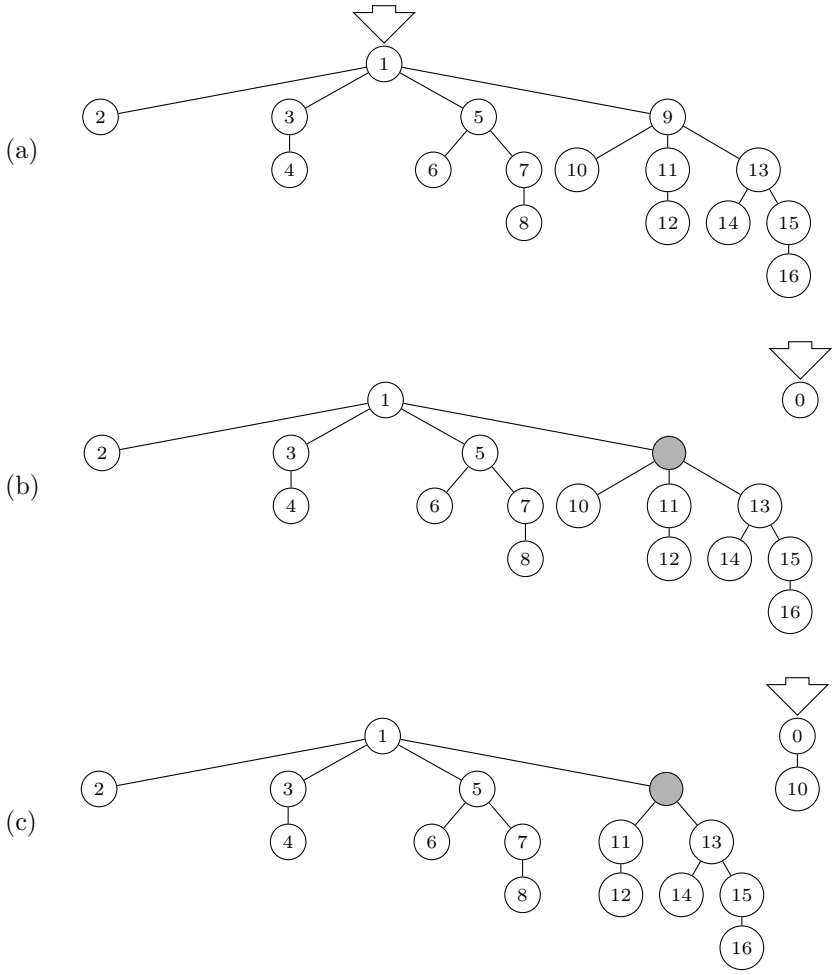


Figure 7 Ausführung der Methode *decrease-key*. Verknüpfungen werden vereinfacht dargestellt. (a) Heap zu Beginn. (b) Item in neue Wurzel verschoben. (c) Die zwei Kinder mit dem höchsten Rang verbleiben beim leeren Knoten, der Rest wird in die neue Wurzel verschoben.

Die Operation *decrease-key* des Hollow-Heaps geht ähnlich vor, wie die des Fibonacci-Heaps. Zunächst wird der Schlüssel erniedrigt. Falls danach die Heap-Bedingung verletzt ist, muss diese wiederhergestellt werden. Dazu wird ein neuer Wurzel-Knoten erstellt, in welchen das geänderte Item verschoben wird (vgl. Abbildung 7 (b)). Dadurch wird der alte Knoten leer. Anschließend werden alle Kinder des leeren Knotens an die neue Wurzel gehangen. Lediglich die zwei Kinder mit dem höchsten Rang verbleiben bei dem leeren Knoten. Hat der Knoten zwei oder weniger Kinder wird keines an die neue Wurzel abgegeben. Der gesamte Vorgang ist beispielhaft in Abbildung 7 dargestellt.
Anders als beim Fibonacci-Heap verbleiben die Kinder eines Knotens teilweise oder auch vollständig nach den Operationen *delete* und *decrease-key* an ihren ursprünglichen Plätzen. Dies verhindert ein ausdünnen des Baumes, wodurch die Markierung des Elter-Knotens nicht mehr notwendig ist. Ein Zeiger vom Kind- auf seinen Elter-Knoten ist somit ebenfalls obsolet.

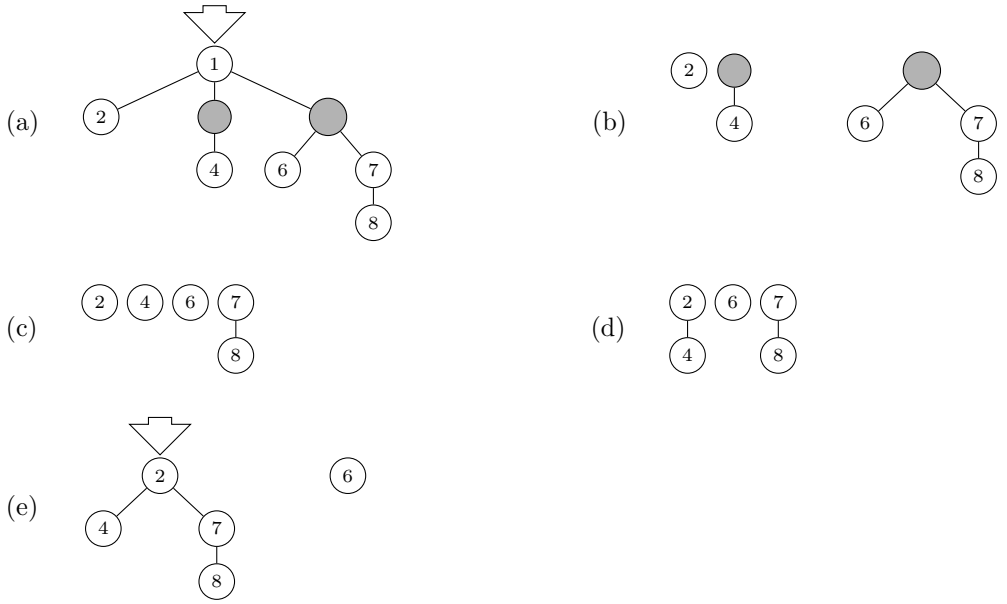


Figure 8 Ausführung der Methode *delete-min*. Verknüpfungen werden vereinfacht dargestellt. (a) Heap mit leeren Knoten zu Beginn. (b) Heap nach Entfernen des Minimums. (c) Heap nach Entfernen leerer Wurzeln. (d) Verknüpfung Knoten gleichen Ranges. (d) Verknüpfung Knoten gleichen Ranges und Setzen des Min-Pointers.

Die nächste Methode, welche durch die leeren Knoten beeinflusst wird ist *delete-min*. In dieser wird, wie beim Fibonacci-Heap, das Minimum gelöscht, wodurch alle Kinder zu Wurzeln werden. Anstatt direkt Wurzeln gleichen Ranges zu verknüpfen werden zunächst alle leeren Wurzeln gelöscht und deren Kinder wiederum zu Wurzeln gemacht. Beispielhaft ist der Vorgang in Abbildung 8 dargestellt.
Weitere Operationen des Hollow-Heaps, wie *make-heap*, *find-min*, *meld* und *insert* werden augenscheinlich von den Änderungen an der Datenstruktur in ihrer Methodik nicht beeinflusst.

4 Analyse

Bei Fibonacci-Heaps gilt die Bedingung, dass ein Knoten mit Rang r mindestens F_{r+2} Nachfolger besitzt. F_{r+2} ist dabei die *r+2-te* Fibonacci-Zahl.
Für Hollow-Heaps stellen wir die Induktionsannahme auf, dass ein Knoten, sich selbst eingeschlossen, mindestens $F_{r+3} - 1 = 1 + (F_{r+2} - 1) + (F_{r+1} - 1)$ Nachfolger besitzt. Es ist dabei irrelevant, ob der Knoten leer oder voll ist. Die rechte Seite der Gleichung setzt sich aus dem Knoten selbst, und den Kindern mit Rang $r - 1$, sowie $r - 2$ zusammen. Das führt daher, dass ein Knoten mit $r \geq 2$ mindestens zwei Kinder besitzt.
Der Anfang der Induktion wird mit $r = 0$ und $r = 1$ gemacht. Für $r = 0$ gilt $F_3 - 1 = 2 - 1 = 1$. In dem Fall besitzt der Knoten keine Kinder, weshalb nur der Knoten selbst zählt. Die Annahme ist für den Fall somit korrekt. Der zweite Fall der Induktionsannahme ist $r = 1$. Dafür gilt $F_4 - 1 = 3 - 1 = 2$. Da der Knoten ein Kind besitzt ist dieser Fall ebenfalls korrekt und die Induktionsannahme abgeschlossen.
Im Induktionsschritt erhöhen wir r um 1, woraus sich die folgende Gleichung ergibt: $F_{r+4} - 1 = 1 + (F_{r+3} - 1) + (F_{r+2} - 1)$. Mit dem Wissen, dass $F_r = F_{r-1} + F_{r-2}$ gilt, lässt sich die Gleichung zerlegen: $F_{r+3} + F_{r+2} - 1 = 1 + (F_{r+2} + F_{r+1} - 1) + (F_{r+2} - 1)$. Durch Kürzen von F_{r+2} kommen wir auf die Induktionsannahme zurück, womit die Aussage

bewiesen ist.

Der maximale Rang eines Knotens im Hollow-Heap beträgt $\log_\phi N$. $\phi = (1 + \sqrt{5})/2$ ist der goldene Schnitt.

Um eine amortisierte Analyse des Hollow-Heaps durchzuführen, wird die Potenzialmethode [3] angewendet. Bei dieser wird der Zustand einer Datenstruktur einem Potenzial zugeordnet. Durch Veränderungen der Datenstruktur kann das Potenzial auf- oder abgebaut werden. Dabei ist das es zu Beginn null und wird niemals negativ.
Jeder volle Knoten, welcher nicht Kind eines anderen vollen Knoten ist erhöht das Potenzial in der Datenstruktur um eins. Das bedeutet, dass Wurzeln und Kinder von leeren Knoten das Potenzial erhöhen, alle anderen dagegen nicht. Die Menge der Wurzeln und Kinder von leeren Knoten werden im folgenden auch als $\#K$ bezeichnet. Die Potenzialfunktion ist durch $\phi = \#K$ gegeben. Da sich das zerstören von Knoten und dem Erstellen von Knoten, in diesem Fall *insert* und *decrease-key*, kostenmäßig aufheben, reicht es die Anzahl der Verknüpfungen $\#L$ zu betrachten. Diese beeinflussen nur die Methoden *delete* und *delete-min*. Die amortisierten Kosten werden als $a = \#L_i - \#L_{i-1} + \phi_i - \phi_{i-1}$ definiert.
Die Funktion *insert* erstellt eine neue Wurzel und erhöht somit das Potenzial um eins. Verknüpfungen werden weder erstellt, noch entfernt. Aus diesem Grund ergeben sich amortisierte Kosten von $\Theta(1)$ für die Methode.
Find-min, *make-heap* und *meld* verändern weder das Potenzial, noch die Anzahl der Links und haben somit amortisierte Kosten $\Theta(1)$.
Die Methode *decrease-key* erstellt durch das Verschieben eines Items eine neue Wurzel und einen leeren Knoten. Die Wurzel erhöht das Potenzial um eins. Der leere Knoten erhöht das Potenzial, je nach Anzahl der Kinder, um maximal zwei. $\#L$ verändert sich nicht, da durch das Verschieben weder Verknüpfungen erstellt, noch gelöscht werden. Die amortisierten Kosten sind somit auch hier konstant.
Das Löschen eines Elementes im Heap verändert die Anzahl der Links nicht. Allerdings kann der Knoten bis zu $\log_\phi N$ volle Kinder haben, wodurch sich das Potenzial um die entsprechende Anzahl erhöht. Die amortisierten Kosten der Funktion betragen somit $\Theta(\log N)$.
Nach dem Löschen des Minimums werden Wurzeln gleichen Ranges verknüpft. Das Verknüpfen erhöht die amortisierten Kosten um eins, führt aber auch dazu, dass eine Wurzel aus der Wurzelliste entfernt wird, wodurch die amortisierten Kosten um eins sinken. Das Umbauen des Heaps ist somit mit amortisiert konstanten Kosten möglich.

5 Abschluss

Multi-Root Hollow-Heaps sind eine vereinfachte Variante der Fibonacci-Heaps. Anhand einer Gegenüberstellung der beiden Datenstrukturen wurde die Grundidee der „lazy deletion“ und ihre Vorteile erläutert. Insbesondere bei den Methoden *delete* und *decrease-key* wurde die vereinfachte Methodik der Operationen sichtbar.
Ziel der Datenstruktur ist es, bei ähnlicher Laufzeit wie sein Vorbild, leichter Verständlich zu sein. Die Autoren zielen somit weniger auf eine bessere Performanz in der Praxis, sondern mehr auf eine vereinfachte Betrachtung der Theorie ab.
Weitere Formen des Holow-Heaps, wie der One-Root Hollow-Heap und der Two-Parent Hollow-Heap wurden nicht besprochen, sind aber dennoch eine interessante Weiterführung des Themas, da diese weitere Vereinfachungen bieten, welche das Nachvollziehen der Methoden erneut vereinfachen.

References

1 Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, July 1987. URL: http://doi.acm.org/10.1145/28869.28874, doi:10.1145/28869.28874.

2 Thomas Dueholm Hansen, Haim Kaplan, Robert E. Tarjan, and Uri Zwick. Hollow heaps. *CoRR*, abs/1510.06535, 2015. URL: http://arxiv.org/abs/1510.06535.

3 Robert Endre Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318, 1985. URL: https://doi.org/10.1137/0606031, arXiv:https://doi.org/10.1137/0606031, doi:10.1137/0606031.

Worstcase-Effiziente dynamische Arrays in der Praxis

Jakob Knorr, Matr. Nummer: 148086[1]

1 Technische Universität Dortmund, Deutschland

Abstract

In der Studie *Worst-Case-Efficient Dynamic Arrays in Practice* [4] untersucht Jyrki Katajainen von der Universität Kopenhagen verschiedene Implementierungsvarianten dynamischer Arrays hinsichtlich Laufzeit und Speicherbelegung, um Nachteile und Hindernisse in der verwendeten Strategie der dynamischen Speicherallokierung zu erkennen. Behandelt werden dabei die Varianten *resizable array relying on doubling, halving, and incremental copying*, *Levelwise-Allocated Pile*, *Sliced Array with fixed-capacity slices* und *blockwise-allocated Pile*. Diese haben allesamt eine Worst-Case-Effizienz von $\mathcal{O}(1)$ für die drei grundlegenden Operationen *Zugriff* (*operator[]*), *push_back* (hinten Anhängen) und *pop_pack* (letztes Element entfernen). Die Implementierungen dieser Varianten in C++ werden dabei bezüglich Geschwindigkeit und Speicherkosten gemessen und mit der Standardimplementierung von C++ [2] verglichen. Unterschiede zeigen sich insbesondere in der Strategie der dynamischen Speicherallokierung und (damit zusammenhängend) in der Berechnung der Speicheraddresse eines Index. Diese Ausarbeitung wird die verschiedenen Implementierungen der dynamischen Arrays beschreiben und die Messergebnisse der Geschwindigkeits- und Speicherbedarfstests von Jyrki Katajainen zusammenfassen.

Digital Object Identifier 10.4230/LIPIcs.BADS2017.2017.1

1 Problemstellung

Ein Array ist eine Datenstruktur zum Speichern von Elementen des gleichen Datentyps. Die meisten gängigen Programmiersprachen beinhalten eine solche Datenstruktur in ihren Standardbibliotheken. Dabei muss zwischen einem statischen Array und einem dynamschen Array unterschieden werden. Ein statisches Array wird mit einer festen Größe initialisiert. Zum Zeitpunkt der Initialisierung wird dabei ein ausreichend großer Speicherblock allokiert, um alle Elemente ablegen zu können. Zugriffe auf ein Element an einer Stelle i können in konstanter Zeit geschehen, indem sich ein *Pointer*[1] auf das erste Element des Arrays gemerkt wird und die Addresse des *Pointers* um das i-Fache der Größe des Datentyps erhöht wird.
Ein dynamisches Array hingegen wird nicht zum Zeitpunkt der Initialisierung auf eine maximale Größe beschränkt, sondern zeichnet sich durch die Eigenschaft aus, dass es an einem Ende wachsen und schrumpfen kann, sollten neue Element hinzukommen oder entfernt werden. Die maximale Anzahl an zu speichernden Elementen muss hier also zum Zeitpunkt der Initialisierung nicht bekannt sein. Dieser Gewinn an Flexibilität während der Nutzung eines solchen dynamischen Arrays, wird mit der Hinzunahme von Komplexität bei der Implementierung des Arrays und einem höheren Speicherbedarf erzielt. Fünf dieser Strategien werden in der Studie „Worst-Case-Efficient Dynamic Arrays in Practice“ von Jyrki Katajainen verglichen und auf einem Testsystem auf Vor- und Nachteile überprüft, um die Frage zu beantworten, welche Variante sich am besten eignet, um ein dynamisches Array in einer Softwarebibliothek

[1] In einem *Pointer* ist die Addresse eines Speicherblocks enthalten. Dadurch können Elemente im Speicher auf einfache Art referenziert werden.

zu implementieren. Eine Variante ist bereits in der Standardbibliothek von C++ enthalten (*std::vector*). Die anderen vier Strategien sind in dem C++-Namensraum *CPH STL* [3] implementiert, welcher von der Universität Kopenhagen entwickelt wurde. Alle Bibliothekten dieses Namensraums stehen zum freien Donwload zur Verfügung (http://www.cphstl.dk - zuletzt gesehen am 19.07.2017).
In seiner Studie stellt Jyrki Katajainen einige Anforderungen an Implementierungen des dynamischen Arrays, welche die vier untersuchten Varianten aus dem C++-Namensraum *CPH STL* allesamt erfüllen:

- Der Zugriffsoperator *[]* soll im Worstcase eine konstante Laufzeit haben ($\mathcal{O}(1)$).
- Die Operation zum Hinzufügen eines Elements an das Ende des Arrays soll im schlechtesten Fall ebenfalls eine Laufzeit von $\mathcal{O}(1)$ betragen (*push_back*).
- Auch das Entfernen des letzten Elements (*pop_back*) soll immer in konstanter Zeit ablaufen ($\mathcal{O}(1)$).
- Alle Implementierungen des dynamischen Arrays benötigen mehr Speicherplatz als die in ihm gespeicherten Elemente in der Summe groß sind. Dieser Speicheroverhead soll möglichst gering ausfallen.
- Die Werte im Speicher sollen nicht verschoben werden. Alle Referenzen und Pointer sollen beibehalten werden.

Die Standardimplementierung von C++ (wie in section 2.1 beschrieben) erfüllt diese Anfordernungen nicht, da die Laufzeiten zum Hinzufügen und Entfernen von Elementen nur amortisiert in $\mathcal{O}(1)$ liegen.

2 Die Untersuchten Implementierungen

2.1 Die C++-Standardlösung

Die Standardlösung um aus einem statischen Array ein dynamisches Array zu machen basiert auf Duplizieren und Halbieren. Erreicht ein Array X bei einem Aufruf der Operation *push_back* das Speicherlimit, so wird ein neues Array Y im Speicher angelegt. Y wird dabei so initialisiert, dass es doppelt so viele Elemente wie X enthalten kann. Anschließend werden alle Elemente aus X nach Y verschoben und der Speicher von X freigegeben, wie in fig. 1 dargestellt. Sollte im Rahmen einer *pop_back*-Operation der genutzte Speicher des Arrays auf ein Viertel des gesamten Speichers fallen, so wird analog zum Verdopplungsvorgang ein halb so großes Array angelegt, die Elemente dorthin verschoben und der allokierte Speicherplatz des nun leeren Arrays freigegeben.

Die beiden Operationen *push_back* und *pop_back* haben im Worstcase eine lineare Laufzeit ($\mathcal{O}(n)$). Allerdings tritt der Fall, dass ein Verdopplungsvorgang stattfindet, frühstens nach $n/2$ *push_back*-Operationen ein (falls das Array zuvor halbiert wurde). Einer Halbierung des Arrays müssen mindestens $n/4$ *pop_back*-Operationen vorausgehen. Die amortisierten Kosten für die beiden Operationen sind deshalb konstant ($\mathcal{O}(1)$). Steht ein Array kurz vor einer Halbierung (enthält also ein Element mehr als ein Viertel der Gesamtgröße), so hat das Array zu diesem Zeitpunkt den größten Speicherbedarf (etwa 400 %). Während der Reorganisation (also der Halbierung) muss zudem ein weiteres Array angelegt werden, dass doppelt so viel Speicher allokiert, wie die Element, die derzeit im Array enthalten sind (zusätzlich 200 %). Für den Zeitraum der Reorganisation belegt das Array demnach bis zu $6n$ im Speicher.
Die Implementierung dieser Strategie in der C++-Standardbibliothek gibt zudem niemals

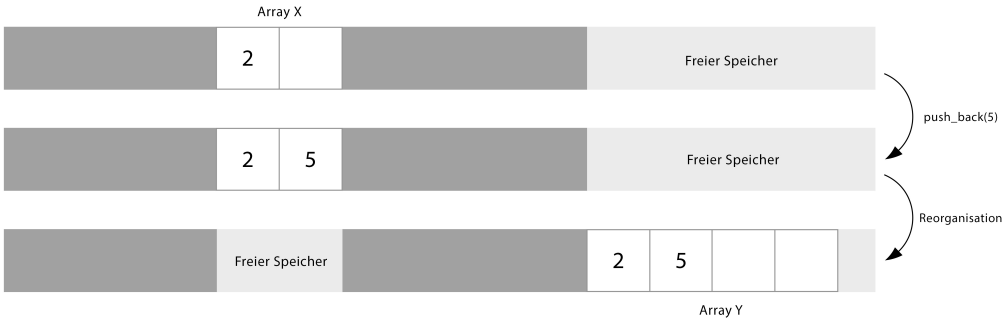


Figure 1 Die Reorganisation des Arrays nach einer *push_back*-Operation im Speicher, wenn das Speicherlimit erreicht wird bei der C++-Standardimplementierung.

Speicher frei. Das Array wird demnach nur beim Hinzufügen von Elementen reorganisiert, beim Entfernen jedoch nicht, sodass der Speicheroverhead beliebig groß sein kann, wenn ein Array erst mit n *push_back*-Operationen vergrößert wird und anschließend mit $n-1$ *pop_back*-Operationen soweit geleert wird, dass es nur noch ein Element enthält. Dies hat den Vorteil, dass *pop_back* in konstanter Zeit ausgeführt wird. Dafür wird einmal allokierter Speicher jedoch nicht wieder freigegeben (außer bei einem manuellen Aufruf der Operation *shrink_to_fit*).

2.2 Resizeable Array

Das *Resizeable Array* benutzt eine ähnliche Strategie wie die oben beschriebene Standardlösung, die auf Duplizieren und Halbieren basiert. Allerdings findet das Verschieben der Elemente nach einer Reorganisation des Arrays nicht sofort statt, sondern verteilt sich auf die nachfolgenden *push_back*- und *pop_back*-Operationen. Ist ein Array X nach dem Hinzufügen eines Elements voll, so wird auch hier ein neues Array Y im Speicher angelegt, das die doppelte Menge an Elementen speichern kann. Der Speicherplatz von X wird nun jedoch nicht freigegeben und die Elemente auch nicht von X nach Y verschoben. Neu hinzugefügte Element werden in Y gespeichert, wobei hier erst ab der zweiten Hälfte von Y die neuen Element platziert werden. Bei jedem Hinzufügen eines Elements (*push_back*) wird zusätzlich das letzte Element von X nach Y verschoben. Ist das Array X also mit n Element gefüllt, erfolgt eine Verdopplung und Y hat eine Kapazität von $2n$. Die nächste Verdopplung erfolgt nach n *push_back* Operationen. Bei jedem *push_back* wird das neue Element in Y hinzugefügt und ein Element aus X nach Y verschoben. Spätestens, wenn die nächste Verdopplung ansteht, befinden sich nun alle Elemente aus X in Y, sodass der Speicher von X freigegeben werden kann. fig. 2 zeigt diesen Vorgang bis zur Freigabe von X.

Analog verhält es sich bei *pop_back*: Nach einer Reorganisation werden bei jedem Aufruf die Elemente von X nach Y verschoben. Enthält das Array Y nach einer Reorganisation jedoch n Elemente (bei einer Kapazität von $2n$), so kann eine erneute Reorganisation bereits nach $n/2$ *pop_back*-Operationen stattfinden. Deshalb werden bei jedem *pop_back* gleich zwei Elemente von X nach Y verschoben, bis X keine Elemente mehr enthält. Diese Lösung umgeht die schlechte Worstcase-Laufzeit, indem das Verschieben der Elemente nicht direkt nach einer Reorganisation stattfindet, sondern inkrementell auf die nachfolgenden Modifikationsoperationen aufgeteilt wird. Dadurch erhalten *push_back* und *pop_back* eine Worstcase-Laufzeit von $\mathcal{O}(1)$. Diese Strategie geht auf Kosten des Speicheroverheads. Sollte

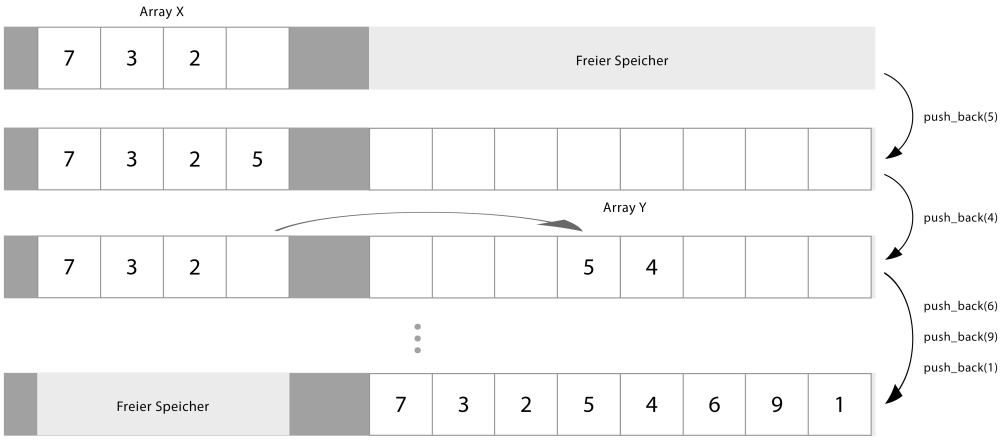


Figure 2 Resizeable Array: Die Elemente werden nach dem Verdopplungsvorgang schrittweise bei jedem *push_back* nach Y verschoben (Die Verdopplung nach dem letzten Schritt ist hier nicht abgebildet).

ein Array X nur zu einem Viertel gefüllt sein, so löst dies eine Halbierung aus. Anschließend hat X eine Kapazität von $4n$ und Y hat Platz für $2n$ Elemente. Es müssen mindestens $n/2$ *pop_back*-Operationen folgen, bis die nächste Reorganisation erfolgt. Zu diesem Zeitpunkt haben die beiden Arrays X und Y zusammen zwölfmal so viel Platz, wie eigentlich für die im Array enthaltenen Elemente benötigt. Das *Resizeable Array* wurde in dem Paket *cphstl::resizeable_array* implementiert.

2.3 Levelwise-Allocated Pile

Diese Lösung basiert auf dem *Levelwise-Allocated Pile*, welcher in [5] beschrieben wird.Zuerst betrachten wir die Eigenschaften eines *Heap*[7]:

- *Form:* Ein *Heap* ist ein linksvollständiger Binärbaum.
- *Kapazität:* Jeder Knoten in dem Baum enthält genau ein Element eines gegebenen Typs.
- *Repräsentation:* Der Binärbaum ist als Array repräsentiert. Die Wurzel des Baumes wird an Stelle 0 des Arrays gespeichert. Die beiden direkten Nachfolger an den Stellen 1 und 2. Dies wird für alle weiteren Knoten fortgesetzt, sodass die beiden Kindknoten des Index i an den Stellen $2i + 1$ und $2i + 2$ zu finden sind.
- *Reihenfolge:* Die Elemente, die in den Baumknoten gespeicherten werden, unterliegen einer Ordnung und keiner der Kindknoten eines Knotens enthält ein Element, das größer (bzw. kleiner) ist, als das Element des Vaters bezüglich der gegebenen Ordnung.

Als *static Pile* definieren wir die Datenstruktur, welche die drei Eigenschaften *Form*, *Kapazität* und *Repräsentation* des *Heap* übernimmt, wobei die *Reihenfolge* nicht berücksichtigt wird und die Element keiner Ordnung unterliegen müssen. Auch ein *static Pile* ist ein statisches Array, was bedeutet, dass die Anzahl der maximal enthaltenen Elemente bereits zum Zeitpunkt der Initialisierung bekannt sein muss. Allerdings dient es als Grundlage um daraus das *Levelwise-Allocated Pile* zu erstellen. Ein statisches Array dient als *Header*. In diesem *Header* werden *Pointer* auf *static Piles* gespeichert, welche die eigentlichen Elemente des dynamischen Arrays enthalten. Ein *static Pile* an der Stelle i im *Header*-Array wird mit der Größe 2^i initialisiert. Auf Level 0 enthält das *Levelwise-Allocated Pile* demnach genau ein Element, auf dem Level 1 zwei Elemente, auf dem Level 2 enthält es vier Elemente usw.. Ist

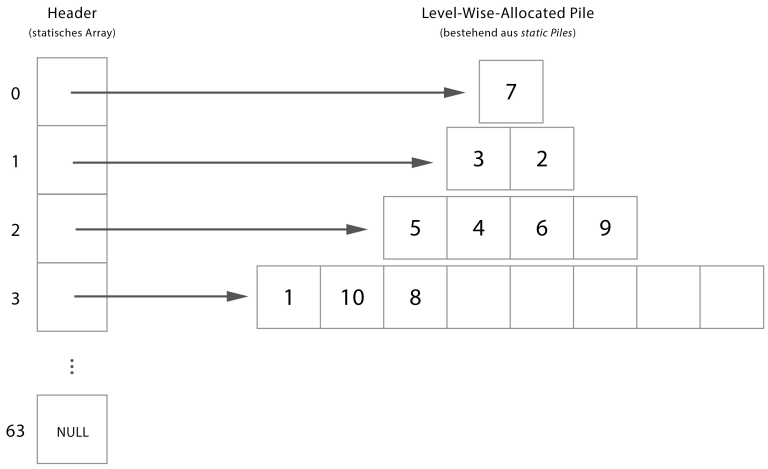


Figure 3 Ein dynamisches Array mit den Werten [7,3,2,5,4,6,9,1,10,8], das als *Levelwise-Allocated Pile* im Speicher liegt.

ein *static Pile* komplett gefüllt, so wird ein neues Array angelegt und der *Pointer* davon im *Header*-Array am Index des nächsten Level gespeichert. Wird ein Array auf einem Level leer (durch eine *pull_back*-Operation), so kann der Speicherplatz für dieses Array freigebeben und der *Pointer* aus dem *Header* gelöscht werden. Die Größe des *Header*-Array, in dem die *Pointer* der einzelnen Level als *static Pile* gespeichert werden, sollte ausreichend groß gewählt werden, dass es für alle praktischen Zwecke ausreicht (64 sollte genügen). Um nun auf ein Element in dem dynamischen Array zuzugreifen, wird das Level und der Index in dem *static Pile* benötigt, das von dem Level referenziert wird. Das Level für den Index k kann einfach durch die Formel $\lfloor \log_2(k+1) \rfloor$ berechnet werden. Zur Berechnung des lokalen Index im *static Pile* des Level kann die Formel $k - 2^{\lfloor \log_2(k+1) \rfloor}$ benutzt werden. Am meisten Speicheroverhead hat diese Strategie zu dem Zeitpunkt, wenn in einem Level nur ein Element enthalten ist. Befindet sich dieses Element auf dem Level k, so können die darunter liegen Level genau $2^k - 1$ Elemente enthalten. Zu diesem Zeitpunkt befinden sich (mit dem ersten Element aus dem k-ten Level) genau 2^k Elemente im Array, der reservierte Speicher hat jedoch Platz für $2^{k+1} - 1$ Elemente. Bei n Elementen hat diese Strategie demnach einen maximalen Speicheroverhead von $n - 1$ (also etwa 100 %). Das *Levelwise-Allocated Pile* wurde im Namensraum *CPH STL* unter dem Namen *cphstl::pile* implementiert.

2.4 Sliced Array

Bei dem *Sliced Array* wird der Speicher in gleich große Teile aufgeteilt (in der Implementierung wird eine Größe von 512 benutzt). Für diese gleich großen Teile werden statische Arrays benutzt und für jedes dieser statischen Arrays wird ein *Pointer* in einem *Resizeable Array* abgelegt. Dieses dient als Verzeichnis, um auf die statischen Arrays zuzugreifen. Wird das letzte statische Array bei einem *push_back* komplett gefüllt, so wird ein neues Array der gleichen Größe im Speicher allokiert und der zugehörige *Pointer* hinten im *Resizeable Array* hinzugefügt. Sollte das letzte statische Array im Rahmen einer *pop_back*-Operation geleert werden, so wird der Speicher freigegeben und der *Pointer* auf das Verzeichnis gelöscht. Die Laufzeit für *push_back* und *pop_back* ist konstant (da keine Elemente verschoben werden) und der indexbasierte Zugriff auf ein Element an Position k des dynamischen Arrays lässt sich

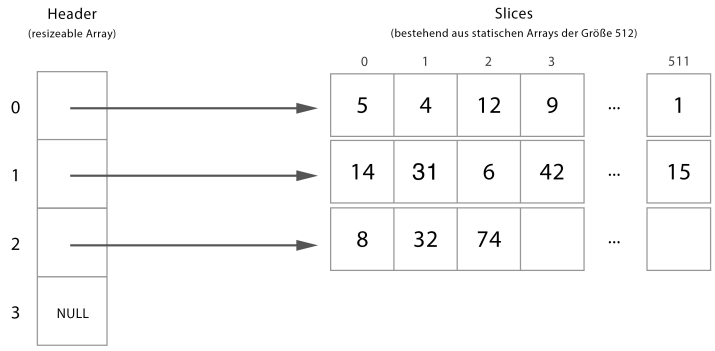


Figure 4 Ein *Sliced Array* mit einer Kapazität von 512 Element pro *Slice*. Enthält hier zurzeit 1027 Element bei einer Gesamtkapazität von 1536.

ebenfalls in $\mathcal{O}(1)$ berechnen: Für den Index im Verzeichnis (also dem *Resizeable Array*) wird die Formel $\lfloor k/512 \rfloor$ benutzt. Für den Index im statischen Array genügt es, den Rest dieser Division zu nehmen ($k \mod 512$). Der Speicheroverhead bei den statischen Arrays beträgt hier die maximale Größe eines solchen Arrays (also 512 in der Implementierung). Dazu kommt der ungenutzte Speicher des *Resizeable Array*, welcher zuvor mit $6n$ während einer Reorganisation angegeben wurde. Da in dieser Implementierung auf alle 512 Elemente ein *Pointer* kommt, ist der Speicheroverhead hier mit $6 * \lceil n/512 \rceil$ zu berechnen, was zwar ebenfalls in $\mathcal{O}(n)$ liegt, in der Praxis dennoch Auswirkungen auf den Speicherbedarf haben kann (da hier zudem nur *Pointer* und keine Elemente gespeichert werden). Die Implementierung des *Sliced Array* mit einer Blockgröße von 512 lässt sich aus dem Namensraum *CPH STL* unter dem Namen *cphstl::sliced_array* benutzen.

2.5 Space-Efficient Array

Das *Space-Efficient Array* ist die Variante mit dem geringsten Speicheroverhead. Hier wird das *Levelwise-Allocated Pile* so erweitert, dass jeder *Pointer* im *Header* auf ein *Hashed Array Tree* zeigt. Ein *Hashed Array Tree* [6] besteht aus einem Verzeichnis-Array fixer Größe, das *Pointer* auf die sogenannten *Leaf*-Arrays enthält, welche zum Speichern der eigentlichen Werte dienen. Die *leafs* haben dabei die gleiche Größe, wie das Verzeichnis selbst (bezüglich der Anzahl der maximal speicherbaren Elemente). Kann das Verzeichnis-Array also n Elemente enthalten, so auch alle Arrays in den *leafs*. Die Gesamtkapazität eines *Hashed Array Tree* ist dann n^2. Um die Berechnung des Index einfach zu halten, sollte die Größe des Verzeichnis-Array (und somit auch die Größe der *leafs*) immer eine Zweierpotenz sein. In fig. 5 ist ein *Hashed Array Tree* visualisiert, das bis zu 16 Element speichern kann. Die Datenstruktur des *Space-Efficient Array* nennt der Autor in [5] auch *blockwise-allocated Pile*. In dieser Implementierung liegt hinter dem Level i des *Directory* (welches hier konkret als *Sliced Array* umgesetzt wurde) ein *Hashed Array Tree* der Größe i, worin i^2 Elemente gespeichert werden können. Eine Ausnahme bildet das Level 0, welches ganz konkret 2 Werte enthält (Index 0 und 1) um die Berechnung des Levels bei gegebenem Index einfach durch eine Bitverschiebung machen zu können. Wird in einem dynamischen Array, das als *Space-Efficient Array* implementiert ist, nun der Index k angefragt, so lässt sich das Level im *Directory* leicht durch die Formel $\lfloor \log_2(k) \rfloor$ berechnen. Der lokale Index für den *Hashed Array Tree* errechnet sich durch $k - 2^{\lfloor \log_2(k) \rfloor}$. Durch die Verschiebung um 1 (in Level 0) kann dies in C++ sehr performant durch das bitweise Verschieben einer 1 nach links um

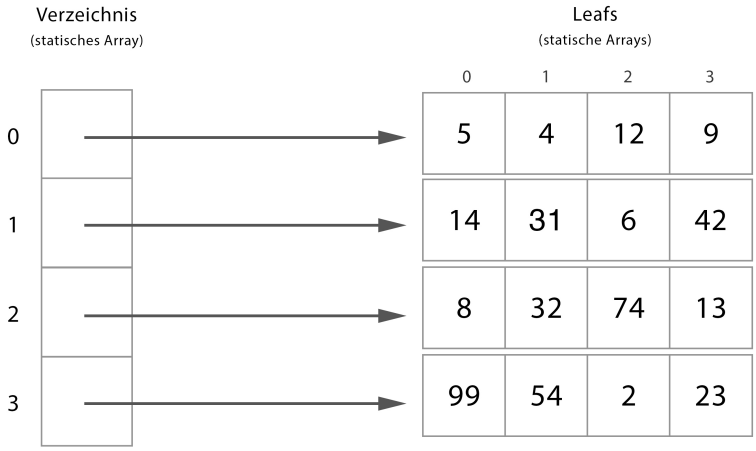


Figure 5 Ein *Hashed Array Tree* mit einer Verzeichnisgröße von 4 und einer Kapazität von 16 Elementen.

$\lfloor \log_2(k) \rfloor$ Stellen geschehen. Diese Strategie ist sehr speichereffizient, da der Speicher eines *Leaf* in einem *Hashed Array Tree* erst allokiert wird, sobald er benötigt wird (das vorherige *Leaf* also komplett gefüllt ist). Ein solches *Leaf* hat die Größe $\mathcal{O}(\sqrt{n})$, weshalb auch der Speicheroverhead höchstens $\mathcal{O}(\sqrt{n})$ beträgt. Dieser Wert ist nach einem Beweis in [1] optimal für Strategien, die den Speicher blockweise allokieren. Die Laufzeit für den indexbasierten Zugriff, *push_back* und *pop_back* sind allesamt konstant, da sich die Indexaddresse durch einfache mathematische Operationen berechnen lässt und beim Hinzufügen und Entfernen keine Reorganisation der Elemente im Speicher stattfindet, sondern lediglich neuer Speicher allokiert wird oder ungenutzter Speicher freigegeben wird. Auch das *Space-Efficient Array* ist im *CPH STL* Namensraum implementiert (*cphstl::space_efficient_array*).

3 Die verschiedenen Implementierungen in der Praxis

In der Theorie werden niedrige Worstcase-Werte immer den niedrigen amortisierten Kosten vorgezogen. Doch in der Praxis können hier unerwartete Effekte entstehen. Da die Zeiten für das Allokieren und Freigeben von Speicherplatz immer als konstant betrachtet werden, ist es nicht klar, welche der vorliegenden Lösungen sich am besten für welchen Einsatzzweck eignet, ohne Tests auf einem realen System durchgeführt zu haben. Deswegen hat Jyrki Katajainen die fünf verschiedenen Implementierungen in einer Reihe von Tests auf einem Testsystem[2] untersucht. Die nachfolgenden Ergebnisse (Grafiken und Tabellen) entstammen alle aus den Tests von Jyrki Katajainen und sind aus [4] übernommen.

3.1 Speichertest

Ein Programm führt n *push_back*-Operationen aus. Anschließend wird der belegte Platz im Arbeitsspeicher gemessen. Der Test wurde mehrfach für verschieden große n wiederholt. Die Ergebnisse sind in fig. 6 visualisiert. Es ist deutlich zu sehen, dass für speicherkritische Programme die beiden Varianten *cphstl::sliced_array* und *cphstl::space_efficient_array* klar

[2] Intel Core i5-2520M @ 2.50GHz x 4; 3.8GB Ram; Ubuntu 14.04LTS; Kernel 3.13.0-83-generic, Compiler g++ v4.8.4

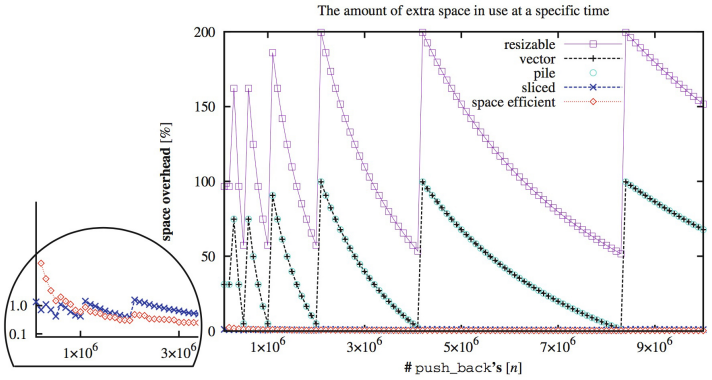


Figure 6 Die Ergebnisse des Speichertests: Nach n *push_back*-Operationen lassen sich starke Unterschiede hinsichtlich des Speicheroverheads bei den verschiedenen Implementierungen erkennen. Quelle: [4]

zu empfehlen sind, da sie während des Test nie mehr als 4 % Speicheroverhead hatten. Die Standardimplementierung *std::vector* und *cphstl::pile* lagen bei jedem Test gleich auf und erreichten einen maximalen Wert von etwa 100 % Speicheroverhead. Weit abgeschlagen mit bis zu 200 % hingegen war die Implementierung *cphstl::resizeable_array*.

3.2 Sortierungstest

Um zu testen, wie schnell die verschiedenen Implementierungen bei den Zugriffen sind, wurden alle Implementierungen sortiert. Dies wurde für verschiedene Arraygrößen (n) wiederholt und dabei jeweils einmal der Algorithmus *Introsort*(aus *std::sort*), der viele sequentielle Zugriffe auf ein Array macht, und ein zweites mal *Heapsort* (aus *std::partial_sort*), der vor allem randomisierte Zugriffe auf das zu sortierende Array macht, benutzt. Die Ergebnisse sind in table 1 und table 2 dargestellt. Es ist deutlich zu erkennen, dass die Standardimplementierung bei beiden Sortierungsalgorithmen die schnellste Lösung ist. Währen die Ergebnisse bei den anderen drei Varianten (*Resizeable Array*, *Levelwise-Allocated Pile* und *Sliced Array*) nicht gravierend schlechter sind, ist das *Space-Efficient Array* bei beiden Tests deutlich langsamer. Die Berechnung des Indexes mithilfe des Logarithmus scheint in der Praxis einiges an Zeit zu kosten.

n	Vector	Resizeable	Pile	Sliced	Space-Efficient
2^{10}	3.56	6.18	9.31	8.35	12.0
2^{15}	3.56	5.96	8.99	8.05	11.6
2^{20}	3.48	5.84	8.80	7.91	11.3
2^{25}	3.48	5.79	8.67	7.80	11.2

Table 1 Resultat des *Introsort* Tests. Die Ergebnisse sind in ns pro n lg n. Quelle: [4]

3.3 Modifikationstest

Bei dieser Untersuchung wurde beobachtet, in welcher Geschwindigkeit sich die verschiedenen dynamischen Arrays in ihrer Größe manipulieren lassen. Beim Wachstumstest wurden n

n	Vector	Resizeable	Pile	Sliced	Space-Efficient
2^{10}	4.83	8.89	17.1	12.5	20.3
2^{15}	4.94	8.47	16.6	12.3	19.8
2^{20}	7.18	10.7	17.8	15.7	21.8
2^{25}	23.5	27.7	33.3	37.0	39.8

Table 2 Resultat des *Heapsort* Tests. Die Ergebnisse sind in ns pro n lg n. Quelle: [4]

Elemente an das Array angehangen. Die nachfolgende Tabelle zeigt die Zeit in Nanosekunden, die durchschnittlich pro *push_back* benötigt wurde.

n	Vector	Resizeable	Pile	Sliced	Space-Efficient
2^{10}	4.23	5.18	5.65	4.65	10.3
2^{15}	3.52	6.39	5.16	4.63	7.35
2^{20}	4.78	8.48	5.12	4.60	6.92
2^{25}	4.15	8.42	4.55	4.58	6.75

Table 3 Resultat des Wachstumstests. Die Ergebnisse sind in ns pro n. Quelle: [4]

Beim Schrumpftest wurden die dynamischen Arrays mit n Elementen gefüllt und anschließend die Zeit gemessen um diese mit *pop_back*-Operationen wieder zu entfernen.

n	Vector	Resizeable	Pile	Sliced	Space-Efficient
2^{10}	0.0	3.62	3.08	2.56	8.15
2^{15}	0.0	2.99	2.15	2.60	5.55
2^{20}	0.0	2.86	2.27	2.41	5.17
2^{25}	0.0	2.91	2.11	2.43	5.07

Table 4 Resultat des Schrumptests. Die Ergebnisse sind in ns pro n. Quelle: [4]

Den Testergebnissen aus table 3 und table 4 ist zu entnehmen, dass das *Resizeable Array* durch die Verschiebung der Elemente langsamer ist, als das *Levelwise-Allocated Pile* oder das *Sliced Array*. Beim dem *Space-Efficient Array* hingegen vermutet Jyrki Katajainen, dass der Grund für die schlechten Werte an der komplizierten Implementierung liegen könnte. Die Werte der C++-Standardlösung beim Schrumpftest sind deshalb so niedrig, weil der reservierte Speicher nie freigebenen wird und so auch keine Reorganisation des Arrays stattfindet.

3.4 Robustheitstest

In dem letzten Test wurde die Robustheit der verschiedenen Strategien untersucht. Im Robustheitstest wurden 2^{20} Elemente in ein leeres Array hinzugefügt und anschließend $2^{20} - 1$ Elemente entfernt. Dieser Vorgang wurde so oft wiederholt, bis ein *out-of-memory Signal* aufgetreten ist. Gezählt wurde dabei, wie viele Wiederholungen es bis zu diesem Signal benötigte.

Dadurch, dass die Standardimplementierung den bereits allokierten Speicher nicht wieder freigibt, schneidet sie in diesem Test um mehrere Größenordnungen schlechter ab (804 Wiederholungen) als alle anderen Implementierungen. Auch das *Sliced Array* kommt im Vergleich zu den drei übrigen Implementierungen auf keinen guten Wert (1048448 Wiederholungen).

Vector	Resizeable	Pile	Sliced	Space-Efficient
804	33 554 432	16 777 216	1 048 448	8 388 473

Table 5 Resultat des Robustheitstests. Zeigt die Zahl der Wiederholungen des Test, bis ein *out-of-memory Signal* aufgetreten ist. Quelle: [4]

In einem Programm, dass ein dynamisches Array häufig stark wachsen und schrumpfen lässt, kann es bei der Implementierung aus *std::vector* schnell zu Speicherproblemen kommen. In solchen Fällen sollte eine der anderen Lösungen bevorzugt werden.

4 Fazit

Ein Array dynamisch wachsen und schrumpfen zu lassen ist keine triviale Aufgabe. Der Vorteil bei der Benutzung, die maximale Größe des Arrays bei der Initialisierung nicht angeben zu müssen, wird durch Leistungs- und Speicherplatznachteile erkauft. Diese können je nach verwendeter Implementierung variieren. Die Worstcase-Laufzeiten lassen sich allerdings nicht auf die Praxis beziehen. In den Tests zeigt sich, dass eine Implementierung, die in der Theorie eine Laufzeit von $\mathcal{O}(n)$ hat (z.B. *std::vector* bei *push_back*), in der Praxis deutlich schneller sein kann, als eine Strategie, die eine konstante Worstcase-Laufzeit hat. Dies liegt vor allem daran, dass bei der Ermittlung dieser Werte nicht berücksichtigt wird, dass auch Operationen die eine konstante Laufzeit haben sehr viel Zeit in Anspruch nehmen können (z.B. Speicherallokierung oder die Indexberechnung bei dem *Space-Efficient Array*). Bei der Benutzung von dynamischen Arrays sollte darauf geachtet werden, ob für den eingesetzten Zweck die Geschwindigkeit ausschlaggebend ist oder ob es sich eher um eine speicherkritische Anwendung handelt. Für Letzteres kann die Standardimplementierung von C++ schnell nicht mehr ausreichen.

References

1 Andrej Brodnik, Svante Carlsson, Erik D. Demaine, J. Ian Munro, and Robert Sedgewick. Resizable arrays in optimal time and space. In *Proceedings of the 6th International Workshop on Algorithms and Data Structures*, WADS '99, pages 37–48, London, UK, UK, 1999. Springer-Verlag. URL: `http://dl.acm.org/citation.cfm?id=645932.673194`.

2 The C++ Standard Committee. Standard for programming language c++, 2017.

3 University of Copenhagen Department of Computer Science. The cph stl, 2000 - 2016.

4 Jyrki Katajainen. Worst-case-efficient dynamic arrays in practice. In *Proceedings of the 15th International Symposium on Experimental Algorithms - Volume 9685*, SEA 2016, pages 167–183, New York, NY, USA, 2016. Springer-Verlag New York, Inc. URL: `http://dx.doi.org/10.1007/978-3-319-38851-9_12`, `doi:10.1007/978-3-319-38851-9_12`.

5 Jyrki Katajainen and Bjarke Buur Mortensen. Experiences with the design and implementation of space-efficient deques. In *Proceedings of the 5th International Workshop on Algorithm Engineering*, WAE '01, pages 39–50, London, UK, UK, 2001. Springer-Verlag. URL: `http://dl.acm.org/citation.cfm?id=647258.720793`.

6 E. Sitarski. Algorithmen alley: Hats: hashed array trees: fast variable-length arrays. 1996.

7 J. W. J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, 1964.

Implementierung der Burrows–Wheeler–Transformation in DYNAMIC

Daniel Korner (Mat.Nr. 147952)

Abstract

Diese Ausarbeitung behandelt eine Implementierung der Burrows–Wheeler–Transformation aus der Open Source C++ Library DYNAMIC. Dabei dient eine Implementierung einer Searchable Partial Sums With Inserts Datenstruktur als Basis, um darauf aufbauend Gap–Encoded Bitvector, Wavelet tree und Run–Length Encoded Strings Datenstrukturen zu erzeugen, welche die BWT Implementierung auf verschiedene Weisen nutzt. Die in der Library implementierten Datenstrukturen sind nah am theoretischen Optimum im Bezug auf den Speicherverbrauch. Im Vergleich mit bekannten Implementierungen ist der Speicherverbrauch zwar signifikant geringer, die Laufzeit jedoch höher.

1998 ACM Subject Classification E.4 CODING AND INFORMATION THEORY

Keywords and phrases C++, DYNAMIC, Burrows–Wheeler–Transformation, Searchable Partial Sums With Inserts, Gap–Encoded Bitvector, Wavelet tree, Run–Length Encoded Strings

Digital Object Identifier 10.4230/LIPIcs.BADS.2017.21

1 Einführung

Mittels der Burrows–Wheeler–Transformation (BWT) können Daten mit redundaten Informationen effizient gespeichert werden, weswegen die BWT die Basis von Kompressionsalgorithmen wie z.B. `bzip2` [5] oder `BWO` [11] bildet. Ein gängiges Verfahren ist, die BWT eines Strings mithilfe eines Suffixarrays oder eines Suffixbaumes zu berechnen [8]. Die Größe des Suffixarrays oder des Suffixbaumes kann deutlich größer werden als die des BWTs, weswegen Libraries wie `bwte` [8] einen modifizierten Ansatz verwenden.

Bestehende Libraries wie `bwte` sind beschränkt in der Größe des Strings oder erlauben nach dem Aufbau der BWT keine Erweiterung dieser. Im Gegensatz dazu ist die C++ Library DYNAMIC lediglich durch den Addressierraum der Maschine und dessen Speicher beschränkt und ermöglicht das Erweitern einer BWT.

Um die BWT Implementierung von DYNAMIC in Abschnitt 7 nachzuvollziehen, gehen wir zunächst auf für die BWT notwendigen Datenstrukturen ein. Wir beginnen mit den Packed Vectors in Abschnitt 2. Diese Datenstruktur ermöglicht das kompakte Speichern von Zahlen. Darauf aufbauend erläutern wir in Abschnitt 3 eine Datenstruktur zur Speicherung von Sequenzen von Zahlen die Funktionen wie `sum` oder `search` definiert. Mithilfe dieser Datenstruktur wird in Abschnitt 4 eine Bitvektor Implementierung realisiert, mit dem wir in Abschnitt 5 eine String Datenstruktur aufbauen. Mit dieser String Datenstruktur und dem Bitvektor bauen wir eine weitere String Datenstruktur in Abschnitt 6 auf. Anschließend nutzen wir die String Datenstrukturen für die BWT Implementierung.

Falls nicht anders angegeben, beginnt die Indizierung in dieser Ausarbeitung bei 0. Bitvektoren werden in dieser Arbeit little-endian angegeben. Das heißt, dass das least significant bit das rechteste Bit ist.

2nd Symposium on Breakthroughs in Advanced Data Structures.
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Des Weiteren gelten folgende Definitionen:

▶ **Definition 1.** Sei S eine Datenstruktur, k, i, x, n und δ eine Zahl und σ eine Zahl oder ein Buchstabe.

- $S.\texttt{at}(i)$ bzw. $S[i]$: Gibt das i–te Element von S zurück.
- $S.\texttt{insert}(i, \sigma)$: Fügt das Element σ an Position i in S ein und verschiebt vorher jedes Element mit Position k mit $k \geq i$ nach $k + 1$.
- $S.\texttt{assign}(i, \sigma)$ bzw. $S[i] = \sigma$: Ersetzt das Element an Position i in S mit σ.
- $S.\texttt{update}(i, \delta)$ bzw. $S[i]$ += δ: Addiert δ auf das Element an Position i in S, dabei muss $0 \leq S[i] + \delta$ gelten.
- $S.\texttt{select}(i, \sigma)$: Gibt die Position des i–ten σ in S zurück.
- $S.\texttt{rank}(i, \sigma)$: Gibt an wie häufig σ bis exklusive der i–ten Position in S vorkommt.
- $S.\texttt{sum}(i)$: $\sum_{m=0}^{i}(S.\texttt{at}[m])$.
- $S.\texttt{search}(x)$: Gibt $\min\{\ i\ |S.\texttt{sum}(i) \geq x\ \}$ zurück. Also die kleinsten Position i in S ab dem die partielle Summe x übersteigt oder gleich ist.
- $S.\texttt{size}()$: Gibt die Anzahl der Elemente in S zurück.
- $S.\texttt{pushBack}(\sigma)$: $S.\texttt{insert}(S.\texttt{size}(), \sigma)$
- Anstelle von $(\sigma_0, \sigma_1, \ldots, \sigma_n)$ schreiben wir auch $\sigma_0\sigma_1 \ldots \sigma_n$ oder $\sigma_0 \cdot \sigma_1 \ldots \sigma_n$
- Für alle $n \in \mathbb{Z}$ gibt $(n)_b$ die binäre Darstellung im 2er–Komplement zurück.
- $(k)_b \ll n$: $(k * 2^n)_b$
- $(k)_b \gg n$: $(k/2^n)_b$
- $\texttt{bitsize}(k)$ gibt an, wie viele Bits notwendig sind, um die Zahl k binär als 2er–Komplement zu speichern, Bsp. $\texttt{bitsize}(124) = 7$.
- $\sim (k)_b$ invertiert die Bits eines Bitvektors. $\sim (k)_b[i]$ ist 1, wenn $(k)_b[i] = 0$, sonst 1.

2 Packed Vector (PV)

Ein normaler Vektor geht über einen Type T mit einer festen Größe von sizeof(T). Unter Verwendung eines 64–Bit unsigned integers benötigt die Sequenz $S = (1, 1, 2, 0, 1, 3, 1)$ in einem Vektor $|S| * 64 = 448$ Bit. Da alle Zahlen in S kleiner sind als 2^2, reichen, unter Verwendung eines 2–Bit unsigned integers, $|S| * 2 = 14$ Bit aus, würden aber aufgrund der festen Größe von 2–Bit verhindern Zahlen größer als $2^2 - 1 = 3$ zu speichern.

Die Packed Vector (PV) Implementierung der Library DYNAMIC (`dyn::packed_vector`) adressiert genau dieses Problem und ermöglicht die dynamische Anpassung der Größe, entsprechend der für die Sequenz notwendigen Bits. Dafür verwendet der PV intern 64–Bit Blöcke und merkt sich, wie viele Werte pro Block gespeichert sind. Dabei besteht `dyn::packed_vector` aus einem Vektor `words` von 64–Bit Blöcken, der Anzahl an Elementen im PV `size_` und der Anzahl von Elementen `intPerWord` pro 64–Bit Block. Dabei unterstützt `dyn::packed_vector` die Funktionen $\texttt{assign}(i, x)$, $\texttt{sum}(i)$, $\texttt{search}(x)$, $\texttt{size}()$, $\texttt{insert}(i, x)$, und $\texttt{at}(i)$ mit folgenden Implementierungen.

- $\texttt{assign}(i, x)$:

```
if (bitsize(x) > max{ bitsize( at(k) ) | 0 ≤ k ≤ size()})
words = Ein neuer Vektor der alle Werte von words enthält,
        jedoch mit bitsize(x) vielen Bits pro Wert. Werte die
        weniger Bits benötigen erhalten führende 0–en.
MASK = (2^intPerWord − 1)_b
words[i/intPerWord] &= ~(MASK ≪ (i % intPerWord))
words[i/intPerWord] |= ((x)_b ≪ (i % intPerWord))
```

- `sum`(i): **return** $\sum_{k=0}^{i}$(`at`(i))
- `search`(x): **return** $\min\{\ i \mid \sum_{k=0}^{i}$ `at`$(i)\ \geq x\}$
- `size`(): **return** `size_`
- `insert`(i, x):

```
if (bitsize(x) > max{ bitsize( at(k) ) | 0 ≤ k ≤ size()})
words = Ein neuer Vektor der alle Werte von words enthält,
        jedoch mit bitsize(x) vielen Bits pro Wert. Werte die
        weniger Bits benötigen erhalten führende 0–en.
Kopier alle Werte words[k] mit k ≥ i absteigend nach words[k+1]
assign(i, x)
```

- `at`(i):

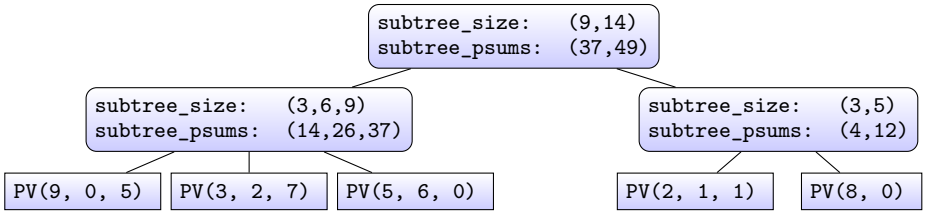
```
64Bit_Block = words.[⌊i/intPerWord⌋]
MASK = (2^intPerWord − 1)_b
64Bit_Block = 64Bit_Block & ( MASK ≪ (i % intPerWord) )
return 64Bit_Block ≫ (i % intPerWord)
```

3 Searchable partial sums with insert (SPSI)

Das Searchable partial sums with insert (SPSI) Problem beschreibt das Problem eine Sequenz $S = (s_0, s_1, \ldots, s_n)$ von natürlichen Zahlen zu speichern und dabei gleichzeitig die Funktionen `sum`(i), `update`(i, δ), `search`(x), `insert`(i, x) zu unterstützen [4]. Hierbei erweitert das SPSI Problem das searchable partial sums (SPS) Problem (`sum`(i), `update`(i, δ), `search`(x)) und das partial sums (PS) (`sum`(i), `update`(i, δ)) [4]. Datenstrukturen, die das SPS Problem lösen, finden breite Anwendung auf dem Gebiet der succinct Datenstrukturen [7].

Die SPSI Implementierung (`dyn::spsi`) von DYNAMIC nutzt intern einen B–Tree. Das ist ein selbst balancierender Baum mit einem beliebig, aber festen maximalen Fanout pro Knoten. Nicht zuletzt wird ein B–Tree verwendet, weil dieser sich ausgezeichnet zur Speicherung eindimensionaler Daten eignet [2]. Die Wurzel des Baums wird in `dyn::spsi` als `root` gespeichert. Im Gegensatz zum klassischen B–Tree von Bayer [1] werden die Werte der Sequenz S lediglich in den Blättern des Baums, die Packed Vectoren sind, abgelegt wie in Beispiel 2 zu sehen.

▶ **Beispiel 2.** `B-Tree` für die Sequenz `(9,0,5,3,2,7,5,6,0,2,1,1,8,0)`, mit einem maximalen Fanout von 3, wie ihn `dyn::spsi` aufbauen könnte.



Eine weitere Besonderheit sind die Vektoren `subtree_size` und `subtree_psum` der inneren Knoten. Hierbei handelt es sich um Vektoren über die summierte Anzahl der Knoten bis zum i–ten Unterbaum (`subtree_sizes`) und die summierten Summen der Zahlen bis zum i–ten Unterbaum (`subtree_psums`). Beide Vektoren dienen zur Laufzeitoptimierung der `sum` und `select` Funktionen.

Dabei unterstützt `dyn::spsi` die Funktionen `at`(i), `sum`(i), `search`(x), `insert`(i, x), `size`(), `update`(i, δ) und `assign`(i, x). Die Funktionen `insert`(i, x) und `at`(i) entsprechen

dabei den Algorithmen aus Organization and Maintenance of Large Ordered Indexes [1] mit der Ausnahme, dass nach dem `insert` zusätzlich die Vektoren `subtree_psums` und `subtree_size` neu berechnet werden. Die restlichen Funktionen sind wie folgt implementiert.

- `sum`(i): `at`(i): `root.btreeSum`(i) mit `btreeSum`(i):

```
j = 0
while(subtree_sizes[j] <= i)
  j++
previous_size = (j==0 ? 0 : subtree_sizes[j-1])
previous_psum = (j==0 ? 0 : subtree_psums[j-1])
child = das j-te Kind
if( Kinder des aktuellen Knotens sind Blätter )
  return previous_psum + child.sum(i-previous_size)
return previous_psum + child.btreeSum(i-previous_size)
```

- `search`(x): `root.btreeSearch`(x) mit `btreeSearch`(x):

```
j = 0
while(subtree_psums[j] < x OR (subtree_psums[j] == 0 AND x == 0))
  j++
previous_size = (j==0 ? 0 : subtree_sizes[j-1])
previous_psum = (j==0 ? 0 : subtree_psums[j-1])
child = das j-te Kind
if( Kinder des aktuellen Knotens sind Blätter )
  return previous_size + child.search(x-previous_psum)
return previous_size + child.btreeSearch(x-previous_psum)
```

- `size`(): `return root.subtree_size[root.subtree_size.size`()-1]
- `update`(i, δ): `root.btreeUpdate`(i, δ) mit `btreeUpdate`(i, δ):

```
j = 0
while(subtree_sizes[j] <= i)
  j++
previous_size = (j==0 ? 0 : subtree_sizes[j-1]);
child = das j-te Kind
if( Kinder des aktuellen Knotens sind Blätter ){
  child.update(i-previous_size , δ)
else
  child.btreeUpdate(i-previous_size , δ)
for( k = j; k< subtree_sizes.size ; k = k + 1)
  subtree_psums[k] += δ
```

- `assign`(i, x): `update`($i, -$`at`$(i) + x$)

4 Gap–Encoded Bitvector (GEB)

Eine binäre Zahl ist einer Sequenz $B \in \{0,1\}^*$ und lässt sich als Tupel (`size`, S) darstellen mit der Länge `size` $= |B|$ und einer Sequenz S von Zahlen $S = (s_0, s_1, \ldots, s_{n-1}, s_n)$ mit $\forall s \in S : s \geq 0$, so dass $B = 0^{\texttt{size}-s'}10^{s_n}10^{s_{n-1}}\cdots 10^{s_1}10^{s_0}$ mit $s' = \sum_{s\in S}(s+1)$ gilt. Die binäre Zahl 001 0010 0011 (291) lässt sich so auf $0^2 10^2 10^3 10^0 10^0$ bzw. die Sequenz $S = (0, 0, 3, 2)$ mit `size` $= 11$ reduzieren.

Genau diese Reduzierung nimmt die GEB Implementierung (`dyn::gap_bitvector`) von DYNAMIC vor und nutzt zur Speicherung der Sequenzen S die `dyn::spsi` Datenstruktur.

Dabei unterstützt `dyn::gap_bitvector` die Funktionen `at`(i), `select`(i, σ), `insert`(i, σ), `size`(), `assign`$(i, 1)$ und `rank`(i, σ) mit folgenden Implementierungen.

- `at`(i): **`return`** `rank`$(i+1, 1) -$ `rank`$(i, 1)$
  Die Bestimmung des Wertes des Bits an i–ter Position lässt sich reduzieren auf die Frage, ob bis zur $(i-1)$–ten Position weniger 1–sen liegen als bis zur i–ten Position. Entsprechend wurde `at`(i) auf die `rank` Funktion des GEB reduziert.
- `select`(i, σ):
  - Für $\sigma = 1$: **`return`** S.`sum`$(i) + i$
    Da bis zur i–ten 1 genau i 1–sen liegen, hat die i–te 1 die Position $\sum_{k=0}^{i}(s_k) + i$.
  - Für $\sigma = 0$: **`return`** S.`search`$(i+1) + i$
    Da die Zahlen der Sequenz S jeweils für die Anzahl 0–en vor einer 1 stehen, zählen wir die 1–sen, indem wir das kleinste k finden für das $\sum_{m=0}^{k}(s_m) \geq (i+1)$ gilt. Da bis zur i–ten 0 genau i 0–en liegen, hat die i–te 0 die Position $k + i$.

  Entsprechend wurde die `select`(i, σ) auf die `sum` bzw. `search` Funktion der internen SPSI reduziert.
- `insert`$(i, 1)$:
  - Für $\sigma = 1$:

```
j = (i, 1)
cnt = S[j]
newsize =  i - ( j == 0 ? 0 : select(j-1, 1)+1 )
delta = cnt - newsize
S[j] += -delta
S.insert(j+1,delta)
```

    Das Einfügen einer 1 an der i–ten Position sorgt für eine Aufspaltung des Summanden s_m in $s_{m'}$ und $s_{m''}$. Um m zu ermitteln, werden die 1–sen bis zur i–ten Position gezählt. Als nächstes ermitteln wir, wie viele 0–len zwischen der einzufügenden 1 an Position i und der darauf folgenden 1 liegen und nennen es delta. Daraus ergibt sich, dass $s_{m'} = s_m -$ delta und $s_{m''} =$ delta gilt. Dabei darf gelten $m = m'$.
  - Für $\sigma = 0$: $S[k] += 1$ mit k als kleinste Zahl für die $\sum_{m=0}^{k}(s_m) \geq (i+1)$ gilt.
    Im Gegensatz zum Einfügen einer 1 reicht es beim Einfügen einer 0, s_k um eins zu erhöhen. Dabei entspricht k der Anzahl der 1–sen bis exklusive der i–ten Position.

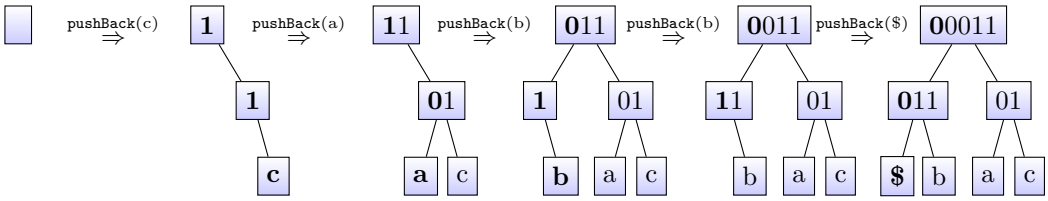
  Entsprechend wurde `insert`(i, σ) auf `update`, `select` und `rank` Funktionen der internen SPSI reduziert.
- `size`(): Die Größe des Bitvektors lässt sich mit $\sum_{i=0}^{|S|}(s_i) + |S|$ berechnen und somit auf die `size` und `sum` Funktionen der internen SPSI reduzieren. Zur Optimierung der Laufzeit wurde jedoch ein Zähler implementiert, der bei der `insert` Funktion des GEB jeweils um eins erhöht wird.
- `assign`$(i, 1)$: Wenn an i–ter Position eine 1 steht, ändert sich nichts. Andernfalls entspricht die Funktion weitestgehend der `insert`$(i, 1)$ Funktion mit dem Unterschied, dass $s_{m'} = s_m - delta - 1$ gilt. Die zusätzliche -1 entfernt die 0, für die eine 1 eingefügt wurde. Analog zu `insert`$(i, 1)$ reduziert sich `assign`$(i, 1)$ auf `update`, `select` und `rank` Funktionen der internen SPSI.
  Zu beachten ist dabei, dass `assign`$(i, 0)$ nicht in der Implementierung vorkommt.
- `rank`(i, σ):
  - Für $\sigma = 1$: **`return`** k mit k als kleinste Zahl für die $\sum_{m=0}^{k}(s_m) \geq (i+1)$ gilt.
  - Für $\sigma = 0$: **`return`** $i - k$ mit k als kleinste Zahl für die $\sum_{m=0}^{k}(s_m) \geq (i+1)$ gilt.

5 Wavelet Tree (WT) String

Wie der Name suggeriert, handelt es sich bei einem Wavelet Tree um einen Binärbaum $wt = (\texttt{INODE} \cup \Sigma, E)$ mit $E \subseteq \texttt{INODE} \times (\texttt{INODE} \cup \Sigma)$ und $\texttt{INODE} = \{0,1\}^*$. Dabei werden innere Knoten mit `INODE`s und Blätter mit Σ modelliert. Zusätzlich wird eine bijektive Funktion $\texttt{enc} : \Sigma \to \texttt{INODE}$ benötigt, die jedem $\sigma \in \Sigma$ einen Binärvektor zuordnet. Eine mögliche Funktion für `enc` wäre die normalisierte UTF–8 Kodierung.

Um einen gegeben String $S = (\sigma_0, \sigma_1, \ldots, \sigma_{n-1}, \sigma_n) \in \Sigma^*$ in einem WT zu kodieren, beginnt man, wie in Beispiel 3 zu sehen, mit einem leeren Baum. Für jeder Buchstabe σ_k, der hinzugefügt werden soll, wird zunächst der $\texttt{enc}(\sigma_k)$ Bitvektor ermittelt. Nun geht man den Bitvektor durch und beginnt in jeder Ebene i das i–te Bit an den inneren Knoten anzuhängen. Handelt es sich bei dem i–ten Bit um eine 0, macht man weiter beim linken Teilbaum, andernfalls beim rechten Teilbaum. Existiert ein Teilbaum noch nicht, wird ein leerer, innerer Knoten angelegt oder ist man am letzten Bit des Bitvektors angekommen, fügt man den Buchstaben σ_k als linkes Blatt, wenn das letzte Bit eine 0 ist, andernfalls als rechtes Blatt ein.

▶ **Beispiel 3.** Schrittweise Aufbau des WT von $S = \text{cabb\$}$ unter Verwendung der Kodierung $\texttt{enc}(\$) = 00$, $\texttt{enc}(\text{a}) = 01$, $\texttt{enc}(\text{b}) = 10$ und $\texttt{enc}(\text{c}) = 11$.



Die WT String Implementierung `dyn::wt_string` von DYNAMIC fasst $\texttt{INODE} \cup \Sigma$ und E zusammen in einen Type. Dieser Type besteht aus einem boolean `leaf`, der angibt, ob der Knoten ein Blatt ist oder nicht, einen Pointer `parent` zum Vaterknoten, einen Pointer `right` zum rechten und `left` zum linken Kind, ein Buchstaben σ und einen Gap Encoded Bitvektor `bv`. Die Wurzel des über diesen Type aufgebauten Baums nennen wir im folgenden `root`. Die Implementierung unterstützt die Funktionen $\texttt{at}(i)$, $\texttt{size}()$, $\texttt{rank}(i, \sigma)$, $\texttt{insert}(i, \sigma)$, und $\texttt{select}(i, \sigma)$. Dabei entspricht das Vorgehen von $\texttt{select}(i, \sigma)$ dem von Grossi et al. [6].

- $\texttt{at}(i)$: `return root.wt_at`(i) mit $\texttt{wt\_at}(i)$:

```
if (leaf)
  return σ
return (bv[i]==0)?left.wt_at(bv.rank(i,0)):right.wt_at(bv.rank(i,1))
```

- $\texttt{size}()$: `return root.bv.size`$()$
- $\texttt{rank}(i, \sigma)$: $\texttt{root.wt\_rank}(i, \texttt{enc}(\sigma), 0)$ mit $\texttt{root.wt\_rank}(i, B, j)$:

```
if (j == B.size())
  return i
else if (B[j] == 0 AND Linkes Kind existiert)
  left.wt_rank(bv.rank(i, 0), B, j+1)
else if (B[j] == 1 AND Rechtes Kind existiert)
  right.wt_rank(bv.rank(i, 1), B, j+1)
return 0
```

- `insert`(i, σ): `root.wt_insert`$(i, \texttt{enc}(\sigma), \sigma, 0)$ mit `root.wt_insert`(i, B, c, j):

```
if(j == B.size())
if(!leaf)
  leaf = true
  σ = c
else
bv.insert(i,b)
if (B[j] == 1)
  Falls noch kein rechtes Kind existiert, lege ein leeres Kind an.
  right.wt_insert( bv.rank(i, 1), B, c, j+1 )
else
  Falls noch kein linkes Kind existiert, lege ein leeres Kind an.
  left.wt_insert( bv.rank(i, 0), B, c, j+1 )
```

.

6 Run–length encoded (RLE) String

Beim Run–length encoding (RLE) handelt es sich um eine verlustfreie Kodierung, die einen String $S = (\sigma_0, \sigma_1, \ldots, \sigma_{n-1}, \sigma_n)$ in eine Sequenz von Läufen (σ_k, j) bzw. $(\sigma_k)^j$, auch runs genannt, kodiert.

▶ **Definition 4.** Ein String $S = (\sigma_0, \ldots, \sigma_n)$ lässt sich wie folgt mittels Run-length encodings `rle` in einen Run-length encoded String umwandeln.

$$\begin{aligned}
\texttt{rle}(()) &= () \\
\texttt{rle}((\sigma, k)) &= (\sigma, k) \\
\texttt{rle}((\sigma_0, \ldots, \sigma_n)) &= \begin{cases} \texttt{rle}((\sigma, k+1) \cdot (\sigma_2, \ldots, \sigma_n)) & \sigma_0 \in \Sigma \times \mathbb{N} \text{ mit } \sigma_0 = (\sigma, k) \wedge \sigma = \sigma_1 \\ \sigma_0 \cdot \texttt{rle}((\sigma_1, \ldots, \sigma_n)) & \sigma_0 \in \Sigma \times \mathbb{N} \text{ mit } \sigma_0 = (\sigma, k) \wedge \sigma \neq \sigma_1 \\ \texttt{rle}((\sigma_0, 1) \cdot (\sigma_1, \ldots, \sigma_n)) & \text{sonst} \end{cases}
\end{aligned}$$

Die RLE String Implementierung `dyn::rle_string` von DYNAMIC beinhaltet einen String `heads`, der die Buchstaben der Läufe enthält, einen Bitvektor `runs`, der die Länge der Läufe unär kodiert enthält, und eine Map `runsPerLetter`, die für jeden Buchstaben die Länge der Läufe des jeweiligen Buchstabens kodiert enthält, mit folgendem Inhalt:
Es gelte $\texttt{rle}(S) = ((\sigma_0, i_0), \ldots, (\sigma_k, i_k))$.

$$\begin{aligned}
\texttt{heads} &= (\sigma_0, \ldots, \sigma_k) \\
\texttt{runs} &= 10^{i_k} \ldots 10^{i_0} \\
\forall \sigma \in \Sigma : \texttt{runsPerLetter}[\sigma] &= 10^{i_{t_k}} \ldots 10^{i_{t_0}} \text{ mit } \forall j < k : t_j < t_{j+1} \text{ und } \forall j \leq k : \sigma_{t_k} = \sigma
\end{aligned}$$

Weiter unterstützt `dyn::rle_string` die Funktionen `at`(i), `select`(i, σ), `insert`(i, σ), `size`$()$, und `rank`(i, σ) mit folgenden Implementierungen.

- `at`(i): `return heads`$[$`runs.rank`$(i, 1)]$
- `select`(i, σ):

```
σ_run = runsPerLetter[σ].rank(i,1)
sel = i - (σ_run == 0 ? 0 : runsPerLetter[σ].select(σ_run-1, 1)+1)
this_run = heads.select(σ_run, σ)
return sel + (this_run == 0 ? 0 : runs.select(this_run-1)+1, 1)
```

- `insert`(i, σ):

```
if ( String ist leer )
  Setze heads = σ und runs = 1 und runsPerLetter[σ] = 1
if ( at[i] == σ OR at[i+1] == σ )
  Wenn at[i] == σ bestimme den run an i–ter Position und verlängere diesen um eins.
  Danach bestimmte den dazu korrespondierenden run in runsPerLetter und
  verlängere diesen ebenfalls. Analog für den Fall, dass at[i+1] == σ gilt.
else
  if (i == 0)
    Füge am Anfang von runs, runsPerLetter[σ] und heads eine 1 bzw. ein σ hinzu.
  else if (i == size)
    Füge am Ende von runs, runsPerLetter[σ] und heads eine 1 bzw. ein σ hinzu.
  else if ( at(i) != at(i+1) )
    Bestimmte den run an i–ter Position. Füge nach diesem run einen neuen run der
    Länge eins über den Buchstaben σ ein. Anschließend bestimme vom letzten σ run
    bis zur i–ten Position und füge in runsPerLetter[σ] den neuen run danach ein
  else
    Bestimmte den run an i–ter Position. Spalte den run über den beliebigen Buchstaben x
    an Position i auf und füge einen neuen run der länge eins über den Buchstaben σ
    zwischen den aufgespaltenen run. Spalte auch den zugehörigen run in runsPerLetter[x].
    Füge danach den zwischen den runs eingefügten σ run in runsPerLetter[x] ein.
```

- `size`(): `return runs.size()`
- `rank`(i, σ):

```
if (Es kommt kein σ in heads vor)
  return 0
this_run = runs.rank(i,1)
this_σ_run = heads.rank(this_run,σ)
rk = 0
if ( this_run != heads.size() && heads[this_run] == σ )
  rk = i − (this_run == 0 ? 0 : runs.select(this_run−1, 1)+1 )
return rk+(this_σ_run==0?0:runsPerLetter[σ].select(this_σ_run−1,1)+1)
```

7 Burrows–Wheeler–Transformation

Die BWT eines Strings $S = (\sigma_0, \ldots, \sigma_n) \in \Sigma^*$ mit $\forall k < n : \sigma_k \neq \$ \wedge \sigma_n = \$$ ist eine reversible Umsortierung der Buchstaben von S die sich aus der letzten Spalte der Burrows–Wheeler–Matrix (BWM) von S ablesen lässt. Hierbei handelt es sich um eine $|S| \times |S|$–Matrix.

Zur Berechnung wird in jede Zeile der $|S| \times |S|$–Matrix der String S entsprechend der Zeilennummer k nach rechts um k viele Buchstaben rotiert eingefügt, wie in der zweiten Matrix in Beispiel 5 zu sehen. Anschließend wird die Matrix, zu sehen in der letzten Matrix des Beispiels, zeilenweise entsprechen der totalen Ordnung $\leq$ sortiert. Dabei muss für die Ordnung zusätzlich gelten: $\forall \sigma \in \Sigma : \$ \leq \sigma$.

▶ **Beispiel 5.** Abfolge zur Berechnung der BWM vom String *cabb*$.

c	a	b	b	$

$\overset{*}{\Rightarrow}$

c	a	b	b	$
$	c	a	b	b

$\overset{\vdots}{\Rightarrow}$

c	a	b	b	$
$	c	a	b	b
b	$	c	a	b
b	b	$	c	a
a	b	b	$	c

$\overset{**}{\Rightarrow}$

$	c	a	b	b
a	b	b	$	c
b	$	c	a	b
b	b	$	c	a
c	a	b	b	$

* Der String *cabb*$ wird in Zeile eins um eins nach rechts rotiert eingefügt.

** Die $|W| \times |W|$–Matrix wird mit der Ordnung $\leq$ zeilenweise sortiert.

First

BWT von cabb$

Wie in dem Beispiel 5 zu sehen, kommen in der BWM gleiche Buchstaben in der letzten Spalte (der BWT) häufig nacheinander vor. Dies kommt durch die Rotation des Strings und die Sortierung der Matrix. Buchstaben, auf die der gleiche Buchstabe im ursprünglichen String folgt, werden in der BWT durch die Sortierung zusammengefasst [3].

Hat man die BWM zu dem String $S \in \Sigma^*$, lässt sich die BWM für den String σS mit $\sigma \in \Sigma$ erzeugen, in dem in jeder Zeile der BWM nach dem \$ eine Spalte mit dem Buchstaben σ hinzugefügt wird. So wird aus der Zeile $(a, b, \$)$ die Zeile $(a, b, \$, \sigma)$. Anschließend wird der String σS als Zeile der Matrix entsprechend der Ordnung $\leq$ eingefügt. Betrachtet man dabei lediglich die erste Spalte, auch First oder F genannt, und die letzte Spalte, auch Last oder L genannt, stellt man fest, dass diese sich nur an zwei Stellen ändern: Wo zuvor in L das Terminalsymbol \$ war, ist danach der neu hinzugefügte Buchstabe σ und dort, wo die Zeile σS hinzugefügt wurde, ist nun das Terminalsymbol \$. Weiter kann auf die Speicherung des Terminalsymbols in F verzichtet werden, da dieses aufgrund der Ordnung immer an 0–ter Position steht. Ebenfalls reicht es aus einzig die Position des Terminalsymbols in L zu speichern, anstelle das Terminalsymbol direkt in L zu speichern, wodurch die Datenstrukturen für L auf die Funktionen `assign`(i, σ) verzichten kann.

Die BWT Implementierung `dyn::bwt` von DYNAMIC nutzt das beschriebene Vorgehen um den BWT für einen String aufzubauen. Dabei besteht `dyn::bwt` aus einem String F für die erste Spalte der BWM, einem String L für die letzte Spalte der BWM, einem Alphabet Σ und der Position $\texttt{pos}_\$$ für \$ im String L und unterstützt die Funktionen `at`(i), `extend`(σ) und `size`() mit folgenden Implementierungen.

- `at`(i):

```
if (i < pos$)
  return L[i]
if (i == pos$)
  return $
return L[i-1]
```

- `extend`(σ): Erweitert die BWT Datenstruktur für den String S zur BWT Datenstruktur für den String σS mit

```
if (σ ∈ Σ)
  num_of_σ = Zähle wie häufig σ in L vorkommt
  pos_in_F = F.select(0,σ) + num_of_σ
else
  pos_in_F = 1 + |{ a ∈ Σ | a < c }|
  Σ = Σ ∪ {c}
F.insert(pos_in_F,Σ);
L.insert(pos$,Σ);
pos$ = pos_in_F+1;
```

- `size`(): `return` L.`size`$() + 1$

8 Fazit

Die DYNAMIC Library bietet viele Vorteile gegenüber bestehenden Libraries wie z.B. dbwt oder bwte, nicht zuletzt, weil DYNAMIC dynamische Datenstrukturen zur Verfügung stellt. Die Datenstrukturen von DYNAMIC sind einzig durch den Adressraum und Speicher der Maschine limitiert und benötigen im Vergleich mit gängigen Libraries deutlich weniger Speicher, wenn auch häufig auf Kosten der Laufzeit [10].

In dieser Ausarbeitung wurden dynamische Datenstrukturen für Vektoren von Zahlen, für das SPSI Problem, für Bitvektoren, für Strings und für die BWT vorgestellt. Im Bezug auf den Speicherverbrauch sind diese Datenstrukturen mit Ausnahme der BWT nah am Speicheroptimum [10]. Weitergehend bietet DYNAMIC noch die dynamische Datenstrukturen

- `dyn::succinct_bitvector`, eine succincte Bitvektor Implementierung auf Basis der SPSI Datenstruktur.
- `dyn:fm_index`, eine komprimierte Stringindizierung Datenstruktur auf Basis einer BWT, Bitvektor und Vektor Datenstruktur.
- `dyn::sparse_vector`, ein speichereffizienter Vektor für Zahlen auf Basis der SPSI und GEB Datenstruktur.
- `bwtil::cw_bwt`, eine speichereffiziente BWT Implementierung

und bietet Implementierung für diverse auf lz77 [12] aufbauende Kompressionsalgorithmen wie h0-lz77 [9].

References

1 Rudolf Bayer. Organization and maintenance of large ordered indexes. *Acta informatica*, 1(3):173–189, 1972.

2 Rudolf Bayer. The universal b-tree for multidimensional indexing: General concepts. In *International Conference on Worldwide Computing and Its Applications*, pages 198–209. Springer, 1997.

3 Michael Burrows and David J Wheeler. A block-sorting lossless data compression algorithm. 1994.

4 Frank Dehne, Jörg-Rüdiger Sack, and Roberto Tamassia. *Algorithms and Data Structures: 7th International Workshop, WADS 2001 Providence, RI, USA, August 8-10, 2001 Proceedings*, volume 2125. Springer, 2003.

5 Jeff Gilchrist. Parallel data compression with bzip2. In *Proceedings of the 16th IASTED international conference on parallel and distributed computing and systems*, volume 16, pages 559–564, 2004.

6 Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 841–850. Society for Industrial and Applied Mathematics, 2003.

7 Wing-Kai Hon, Kunihiko Sadakane, and Wing-Kin Sung. Succinct data structures for searchable partial sums with optimal worst-case performance. *Theoretical Computer Science*, 412(39):5176–5186, 2011.

8 Daisuke Okanohara and Kunihiko Sadakane. A linear-time burrows-wheeler transform using induced sorting. In *International Symposium on String Processing and Information Retrieval*, pages 90–101. Springer, 2009.

9 Alberto Policriti and Nicola Prezza. Fast online lempel-ziv factorization in compressed space. In *International Symposium on String Processing and Information Retrieval*, pages 13–20. Springer, 2015.

10 Nicola Prezza. A framework of dynamic data structures for string processing. *arXiv preprint arXiv:1701.07238*, 2017.

11 Anthony Ian Wirth and Alistair Moffat. Can we do without ranks in burrows wheeler transform compression? In *Data Compression Conference, 2001. Proceedings. DCC 2001.*, pages 419–428. IEEE, 2001.

12 Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977.

Eine experimentelle Untersuchung von Prioritätswarteschlangen für Externspeicher-Algorithmen

Fabian Pawlowski, 127443[1]

1 **TU Dortmund, Otto-Hahn-Str. 12, 44227 Dortmund, Deutschland**
`fabian.pawlowski@tu-dortmund.de`

Zusammenfassung

Diese Seminararbeit befasst sich mit den Erkenntnissen und Ergebnissen des Papiers *„An Experimental Study of Priority Queues in External Memory“* [3]. Priority Queues, oder zu Deutsch Prioritätswarteschlangen (im Folgenden *PQs*), sind eine erweiterte Form einer Datenstruktur zur Realisierung einer Warteschlange. Elementen der *PQ* wird ein Schlüssel zugeordnet, der die Reihenfolge der Abarbeitung vorgibt.
PQs finden in der Informatik eine weite Verbreitung, da sie bei vielen verschiedenen Problemen eingesetzt werden können. Viele der *PQ* Implementierungen haben gemeinsam, dass sie für kleine Datensätze, die den internen Speicher nicht übersteigen, geschaffen wurden. Für Instanzen, die die Größe des internen Speichers überschreiten, ist zu beobachten, dass *PQs* die Performance des Algorithmus signifikant verschlechtern.
Diese Arbeit stellt mit dem *externen Radix-Heap* und dem *externen Array-Heap* zwei Heap-Varianten zur Realisierung von effizienten Prioritätswarteschlangen für Externspeicher-Algorithmen aus [3] vor.

1998 ACM Subject Classification E.1 Data Structures

Keywords and phrases Externspeicher-Algorithmen, Prioritätswarteschlangen, Heaps

Digital Object Identifier 10.4230/LIPIcs.BADS.2017.9

1 Einleitung

Eine Priority Queue, zu Deutsch *Prioritätswarteschlange* (im Folgenden *PQ*) ist eine Datenstruktur, die Tupel von Inhalten (*Values*) und Prioritätsschlüsseln (*Keys*) speichert. Sie bieten die folgenden Operationen an: **extract_min** (liefert das Element mit dem niedrigsten Schlüssel zurück), **decrease_key** (verringert den Schlüsselwert eines Elements und erhöht damit seine Priorität) und **insert** (fügt ein neues Element der *PQ* hinzu).
Prioritätswarteschlangen finden eine breite Anwendung in vielen verschiedenen Bereichen der Informatik, wie z.B. bei der Suche nach kürzesten Pfaden mit Dijkstras Algorithmus [6]. Viele Implementierung von *PQs* beschränken sich dabei allerdings auf kleine Datensätze, die nicht größer als der interne Speicher sind. Es ist zu beobachten, dass diese Implementierungen bei Externspeicher-Algorithmen, bei denen Datensätze verarbeitet werden, die nicht mehr vollständig in den internen Speicher passen, zu einer Verlangsamung der Rechenzeit bei jeder *PQ*-Operation zum Faktor 10^5 bis 10^6 führen [3].
Diese Arbeit stellt zwei *PQ*-Implementierungen vor, die diesem Problem entgegen kommen und *PQs* auch effizient für Externspeicher-Algorithmen einsetzbar machen sollen. Die erste *PQ*-Variante adaptiert den *Two-Level-Radix-Heap* aus [1] für Externspeicher-Algorithmen, während die zweite Variante eine Vereinfachung der *PQ*-Variante auf Basis von *Array-Heaps*

aus [4] darstellt. Um die Effizienz der vorgestellten Varianten bewerten zu können benutzen *Brengel et al.* das *Parallel Disk Model* von *Vitter und Shriver* [10].
Im *Parallel Disk Model* wird der Speicher eines Computers in zwei verschiedene Bereiche eingeteilt. Einmal ein kleiner und schneller *interner Speicher* der Größe M, sowie einen größeren und langsameren *externen Speicher*, mit D unabhängigen *Disks* (beispielsweise die Platten einer mechanischen Festplatte). Daten zwischen dem *internen Speicher* und den *Disks* werden in *Blöcken* der Größe B (auch *Disk Pages* oder *Seiten* genannt) übertragen. Jeder Plattenzugriff überträgt dabei DB viele Elemente, beim Abrufen einer *Seite* von jeder Platte. Die Performance wird anhand der ausgeführten Plattenzugriffe (kurz *I/Os*), durch Messen der internen CPU-Zeit und durch Messen der Anzahl der betroffenen *Disk Pages* gemessen.

2 Radix Heaps

Die erste *PQ*-Variante geht zurück auf den *Two-Level-Radix-Heap* für Internspeicher-Algorithmen, wie in [1] vorgestellt. Bevor wir uns mit der Externspeichervariante dieses *Radix-Heaps* aus [3] beschäftigen, schauen wir uns die Grundlagen von *Radix-Heaps* einmal an. Abbildung 1 skizziert die Idee des einfachen *Radix-Heap*.

2.1 Interner Radix Heap

Ein *Radix-Heap* ist eine Datenstruktur, die hauptsächlich aus sogenannten *Buckets* oder auch *Eimern* besteht, deren Größe exponentiell zunehmen. Dabei müssen einige Voraussetzungen erfüllt sein. Alle Schlüssel müssen natürliche Zahlen seien und es muss $max(key) - min(key) \leq C$ für eine Konstante C gelten. Des Weiteren geben *Radix-Heaps* Werte, die mit aufeinander folgenden **extract_min**-Aufrufen ausgegeben werden, in monotoner Reihenfolge aus. Es handelt sich bei *Radix-Heaps* also auch um eine sogenannte *monotone priority queue*. Definition 1 gibt eine mögliche Definition eines *Radix-Heaps* an.

▶ **Definition 1.** Ein *Radix-Heap* ist ein Container, bestehend aus $B = \lceil log_2(C+1) \rceil + 2$ *Buckets*, mit einer initialen Größe von $1, 1, 2, 4, 8, ..., 2^{B-2}$. Jeder *Bucket* i besitzt eine untere Schranke $u(i)$, die gegeben ist durch:

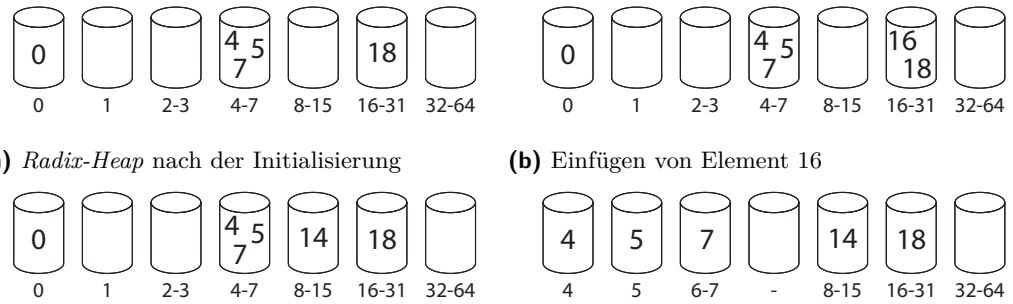
$$u(0) = 0;$$
$$u(i) = u(i-1) + 2^{i-1} \text{ für } 1 \leq i \leq B-1$$

Die Größe eines *Buckets* i ist gegeben durch $size(i)$ und ist definiert als:

$$size(0) = 1;$$
$$size(i) = 2^{i-1} \text{ für } 1 \leq i \leq B-1;$$

Dabei gelten immer die folgenden Invarianten:
1. $u(i) \leq$ Schlüssel in $b(i) < u(i+1)$
2. $u(0) = 0, u(1) = u(0) + 1, u(B) = \infty$ und $0 \leq u(i+1) - u(i) \leq 2^{i-1}$ für $i = 1, ..., B-1$

Mit Hilfe von Definition 1 kann die Funktionsweise des *Radix-Heaps* mit Abbildung 1 verdeutlicht werden. In dem Beispiel wird angenommen, dass Elemente mit den Werten $\{0, 4, 5, 7, 18\}$ in den *Radix-Heap* einsortiert werden sollen. Da zu jedem Zeitpunkt $max(key) - min(key) \leq C$ für eine Konstante C gelten muss, kann die Konstante in diesem Beispiel minimal den Wert 18 besitzen. Abbildung 1a zeigt den *Radix-Heap* der Größe $B = 7$ für die Elemente $\{0, 4, 5, 7, 18\}$, bei dem das exponentielle Wachstum der *Bucket*-Größe



(a) *Radix-Heap* nach der Initialisierung

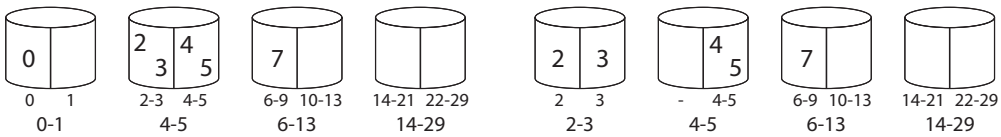
(b) Einfügen von Element 16

(c) Verringern von Element 16 auf 14

(d) Extrahieren des Minimums

Abbildung 1 Beispiel der *Radix-Heap* Operationen. Zur Vereinfachung des Beispiels wird angenommen, dass der *Inhalt* und der *Schlüssel* eines Elements den selben Wert haben.

ersichtlich wird. Zur Vereinfachung nehmen wir an, dass der Wert des Inhalts und der Wert des Schlüssels jedes Elements identisch sind.
Abbildung 1b veranschaulicht die **insert**-Operation, bei der die *Buckets* von rechts nach links durchlaufen werden und das neue Element $x = 16$ in das *Bucket* gespeichert wird, so dass gilt $u(i) \geq key(x)$.
Bei der **decrease_key**-Operation werden analog zur **insert**-Operation die *Buckets* linear von rechts nach links durchlaufen, bis der *Bucket* gefunden wurde, in den das Element mit dem veränderten Schlüssel eingefügt werden muss. In Abbildung 1c wird das Element 16 aus Abbildung 1b auf den Wert 14 verringert und in den *Bucket* mit dem Bereich $8 - 15$ einsortiert.
Abbildung 1d veranschaulicht das Vorgehen bei der **extract_min**-Operation. Hierbei wird ein Element aus dem *Bucket* $b(0)$ entfernt und zurückgegeben. Sofern $b(0)$ dadurch leer geworden ist, wird nach dem ersten nicht leeren *Bucket* $b(i)$ mit $i > 0$ gesucht und dessen kleinstes Element k bestimmt. $u(0)$ wird anschließend auf k gesetzt und die Bucketgrenzen werden gemäß der Invarianten neu gesetzt. Abschließend werden die Elemente des *Buckets* $b(i)$ auf die neuen davor liegenden *Buckets* verteilt.
Es ist zu beobachten, dass das Neu-Einfügen von Elementen in der **extract_min**-Operation am teuersten ist [1]. Um die benötigte Zeit zu reduzieren stellen *Ahuja et al.* in [1] eine erweiterte Variante des zuvor vorgestellten *Radix-Heap* vor, der in Abbildung 2 beispielhaft dargestellt wird. Im *Two-Level-Radix-Heap* aus [1] wird jeder *Bucket* in gleich große *Segmente* eingeteilt. Die *Segmente* unterteilen den Bereich eines *Buckets* dann in mehrere kleinere Bereiche, damit während der Anpassung der unteren Schranken u in der **extract_min**-Operation, weniger Elemente neu eingefügt werden müssen. Für eine detaillierte Beschreibung des *Two-Level-Radix-Heap* von *Ahuja et al.* sei an dieser Stelle auf [1] verwiesen.



(a) *2-Level-R-Heap* nach der Initialisierung

(b) Extrahieren des Minimums

Abbildung 2 *Two-Level-Radix-Heaps*: Unterteilung der *Buckets* in *Segmente*. In diesem Beispiel zwei *Segmente* pro *Bucket*.

2.2 Externer Radix Heap

An den *externen Radix-Heap* von *Brengel et al.* werden natürlich die selben Voraussetzungen gestellt, wie für den ganz einfachen *Radix-Heap* aus Abschnitt 2.1, das heißt, dass die Schlüssel natürliche Zahlen sein müssen, C eine Konstante, die mindestens die maximale Differenz zwischen dem größten und kleinsten Schlüssel angibt, sowie die Einhaltung der Monotonie bei **extract_min**-Operationen.
Abbildung 3 verdeutlicht die Struktur des *externen Radix-Heaps*. Sei r (auch *radix* genannt) eine willkürlich gewählte natürliche Zahl und h eine minimal große natürliche Zahl, so dass $r^h > C$ gilt (h lässt sich also bestimmen durch $h = \lceil log_r(C+1) \rceil$). Sei außerdem k die Priorität eines willkürlichen Elements der Warteschlange und sei $k_h k_{h-1} ... k_0$ dessen Repräsentation zur Basis r (hier weiterhin bezeichnet als $(k)_r$). Sei min die Priorität des zuletzt extrahierten Elements, oder 0, sofern noch kein Element extrahiert wurde. Sei also $m_h m_{h-1} ... m_0$ die Repräsentation von min zur Basis r (hier auch weiterhin bezeichnet als $(min)_r$). Sollte ein Element mit Priorität k zur Warteschlange gehören gilt $k - min < r^h$. Folglich gilt $k_h = m_h$ oder $k_h = (m_h + 1) \mod r$.
Der *externe Radix-Heap*, wie in Abbildung 3 zu sehen, besteht aus den folgenden Teilen:

- Aus h *Arrays*, wobei jedes Größe r hat. Jedes Array-Element speichert eine lineare Liste von Blöcken, bezeichnet als *Bucket*. Sei $B(i, j)$ das *Bucket*, das mit der j-ten Stelle des i-ten *Arrays* assoziiert wird, für $0 \leq i < h, 0 \leq j < r$. Für jedes *Bucket* wird der erste *Block* im Hauptspeicher gespeichert. Hierdurch wird r beschränkt durch die Beziehung $h \cdot r \cdot B \leq M$.
- Ein *Bucket* N, welches alle Elemente mit Priorität k mit $k_h = (m_h + 1) \mod r$ enthält und ebenfalls seinen ersten *Block* im Hauptspeicher speichert.
- Einer Internspeicher-*PQ* Q, die alle Indizes von nichtleeren *Buckets* enthält. Diese Indizes sind lexikographisch sortiert, das heißt $(i, j) < (i', j')$ wenn entweder $i < i'$ oder $i = i'$ und $j < j'$ gilt. Q speichert nie mehr als $h \cdot r$ Indizes.

Die entscheidende Idee des *externen Radix-Heaps* ist die Verwendung der Repräsentationen $(k)_r$ und $(min)_r$. Elemente werden nämlich entsprechend ihrer Differenz zwischen diesen beiden Repräsentationen gespeichert. Ein Element k wird dabei in *Bucket* N gespeichert wenn $k_h = (m_h + 1) \mod r$ gilt, in allen anderen Fällen wird es in *Bucket* $B(i, j)$ gespeichert, wobei i und j gegeben sind durch $i = \max(\{l | m_l \neq k_l, 0 \leq l \leq h\} \cup \{0\})$ und $j = k_i$.

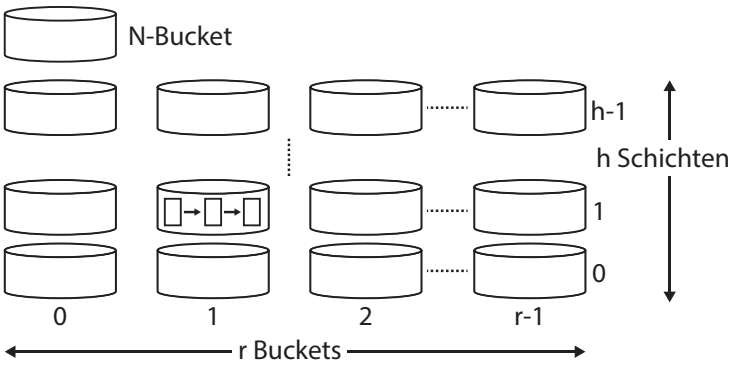


Abbildung 3 Der externe *Radix-Heap* besteht aus h *Schichten* mit jeweils r *Buckets*, einem zusätzlichen *Bucket* N und einer internen *Prioritätswarteschlange* Q.

▶ **Example 2.** Überlegen wir kurz ein Beispiel, um dieses Vorgehen zu veranschaulichen. Nehmen wir an, wir haben einen *externen Radix-Heap* gegeben durch $r = 3$. Nehmen wir weiter an, dass noch kein Element entfernt wurde, *min* also den Wert 0 hat und ein Element mit dem Schlüssel 140 den größten Schlüssel in dem *Radix-Heap* angibt und somit die Konstante C mindestens den Wert 140 hat. Außerdem ist h gegen durch 5 da $r^h > C$ gilt. Abbildung 4 verdeutlicht das Vorgehen beim Einfügen eines neuen Elements mit Schlüssel 120. Die Repräsentation von *min* zur Basis r ist in diesem Fall also 000000 während $(k)_r$, wobei k in diesem Fall 120 entspricht gegeben ist durch 011110. Da $k_h = (m_h + 1) \mod r$ nicht gilt, wird das neue Element in *Bucket* $B(i, k_j)$ gespeichert. Hier schauen wir uns also die Differenz zwischen $(k)_r$ und $(min)_r$ an, um i mit $i = \max(\{l | m_l \neq k_l, 0 \leq l \leq h\} \cup \{0\})$ zu bestimmen. In diesem Beispiel ist i und j also 1. Das Element mit Schlüssel 120 würde also in *Bucket* $B(1, 1)$ eingefügt werden.

Es ist zu beobachten, dass Elemente mit Schlüsseln gleich *min* in *Bucket* $B(0, m_0)$ eingefügt werden. Außerdem haben Elemente die in $B(0, j)$ eingefügt werden den selben Schlüsselwert j und die *Buckets* $B(0, j)$ mit $y < m_0$ sind leer.
Im Folgenden werden die Operationen **insert**, **decrease_key** und **extract_min** beschrieben, sowie die Performanzen hinsichtlich des *Parallel Disk Models* [10] angegeben.

Insert: Wie aus Beispiel 2 zu sehen, wird zuerst überprüft, ob $k_h = (m_h + 1) \mod r$ gilt. Sollte dies der Fall sein, wird das einzufügende Element in *Bucket* N eingefügt, andernfalls in *Bucket* $B(i, k_j)$ mit $i = \max(\{l | m_l \neq k_l, 0 \leq l \leq h\} \cup \{0\})$. Sollte *Bucket* $B(i, k_j)$ vor dem Einfügen leer sein, wird der Index (i, k_i) zur *PQ* Q hinzugefügt. *Bucket* $B(0, m_0)$ kommt eine besondere Rolle zu. Da in diesem *Bucket* alle Elemente den selben Schlüssel haben, werden von hier zwei *Blöcke* im Hauptspeicher gespeichert. Sollte der zweite *Block* durch neu eingefügte Elemente vollständig gefüllt werden sein, wird dieser Festplatte verschoben, damit die Inhalte des ersten *Blocks* in den zweiten *Block* verschoben werden können.

Extract_min: Sofern *Bucket* $B(0, m_0)$ nicht leer sein sollte, wird ein zufälliges Element aus diesem *Bucket* entfernt, da alle Elemente den selben Schlüssel haben. Sollte $B(0, m_0)$ leer sein, wird die interne *PQ* Q verwendet, um den ersten nicht-leeren *Bucket* zu finden. Entweder ist das *Bucket* $B(i, j)$ oder *Bucket* N. In beiden Fällen wird das neue Minimum gesucht und *min* entsprechend angepasst. Wie beim Reorganisieren des normalen *Radix-Heaps* ist es nun notwendig die Elemente des aktuellen *Buckets* $B(j, i)$ wie bei der *insert*-Operation in *Buckets* auf niedrigeren Ebenen zu verteilen. Da hierdurch Elemente in leere *Buckets* eingefügt werden können muss zusätzlich Q angepasst werden. Abschließend kann jedes Element des *Buckets* $B(0, m_0)$ ausgegeben werden.

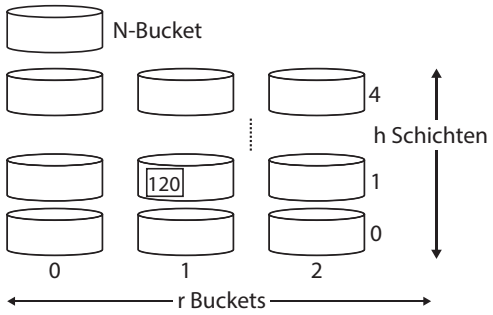


Abbildung 4 Beispiel eines *externen Radix-Heaps* nach dem Einfügen eines *Blocks* mit Schlüsselwert 120.

Decrease_key: Die *decrease_key*-Operation funktioniert genauso wie in Abschnitt 2.1 für den einfachen *Radix-Heap*. Der Schlüssel eines Elements wird verringert und anschließend wird das Element wie bei der *insert*-Operation neu einsortiert.

Damit die Datenstruktur effizient arbeiten kann, müssen r und h geeignet gewählt werden. Bisher ist unsere Forderung, dass der erste *Block* jedes *Buckets*, sowie die interne *PQ* Q im internen Speicher M Platz haben müssen, also $h \cdot r \cdot B \leq M$. Wie zuvor gesehen ist h beschränkt durch $\Theta(log_r C)$. Der maximal Wert für r kann also unter Einhaltung der obigen Bedingung berechnet werden, wenn wir zusätzlich annehmen, dass $m = M/B$ gilt. Daraus ergibt sich:

$$r \cdot log_r C \leq m \Rightarrow \frac{r}{log\, r} \leq \frac{m}{log\, C} \Rightarrow r = \Theta(\frac{m}{log\, C} \frac{m}{log\, C})$$

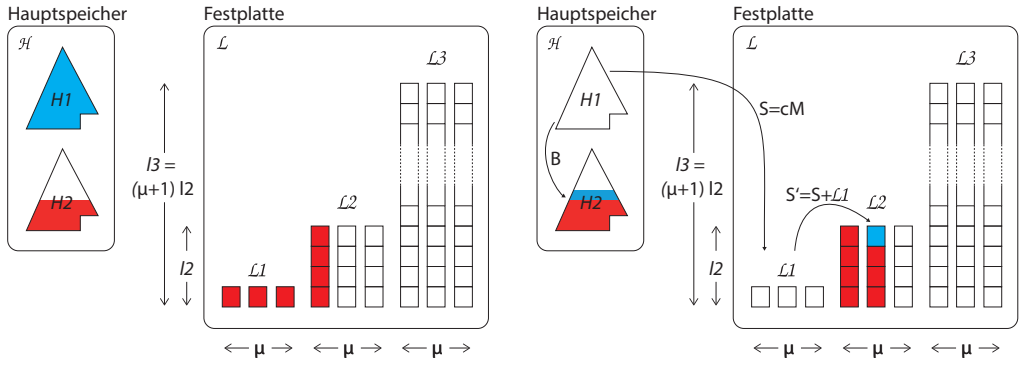
Nach *Brengel et al.* in [3] erhalten wir für den *externen Radix-Heap*:

▶ **Theorem 3.** *Sei* $m = M/B$ *und sei* $r = \Theta(\frac{m}{log\, C} \frac{m}{log\, C})$. *Eine* ***insert****-Operation in den externen Radix-Heap kostet* $O(1/B)$ *amortisierte I/Os und* $O(log(r\, log_r\, C))$ *CPU-Zeit. Eine* ***extract_min****-Operation kostet* $O((1/B)log_r C)$ *amortisierte I/Os und* $O(log_r C \cdot log(r log_r C))$ *amortisierte CPU-Zeit.*

In Anwendungen, in denen C klein ist, kann der *externe Radix-Heap* effizient eingesetzt werden. Für typische M, B und C ist $log_r(C)$ etwa zwei oder drei.

3 External Array-Heaps

Die zweite Heap-Struktur aus *„An Experimental Study of Priority Queues in External Memory“* [3] ist eine Vereinfachung aus [4]. Dieser *externe Array-Heap* besteht aus einem internen Heap $\mathcal{H}$ und einer externen Datenstruktur $\mathcal{L}$, die aus einer Menge von sortierten *Arrays* unterschiedlicher Länge besteht. Diese *Arrays* sind in L Schichten $\mathcal{L}_i$, $1 \leq i \leq L$ eingeteilt. Jede Schicht besteht aus $\mu = (cM/B) - 1$ Feldern (auch *Slots* genannt) der Länge $l_i = (cM)^i/B^{i-1}$ für $c < 1$. Jeder dieser Slots ist entweder leer oder er enthält eine sortierte Folge von höchstens l_i Elementen.
Elemente werden zuerst zu $\mathcal{H}$ hinzugefügt. Sollte $\mathcal{H}$ dabei komplett gefüllt werden, werden cM dieser Elemente in den externen Speicher verschoben. Dabei wird versucht, sie zuerst in sortierter Folge in die erste Schicht $\mathcal{L}_1$ einzutragen. Sollte das nicht möglich sein, werden alle Elemente aus der Schicht $\mathcal{L}_1$ mit den cM Elementen aus $\mathcal{H}$ zu einer sortierten Liste gemischt, die dann in einem freien Slot in Schicht $\mathcal{L}_2$ gespeichert werden soll. Sollte auch da kein freier Slot existieren, wird das Verfahren solange wiederholt, bis ein freier Slot gefunden wurde. Abbildung 5a veranschaulicht den Aufbau des vereinfachten *externen Array-Heap*, welcher im Folgenden beschrieben wird. Der Aufbau des zuvor beschriebenen normalen *externen Array-Heaps* sieht sehr ähnlich aus. Der einzige Unterschied liegt in der Unterteilung des internen Heap $\mathcal{H}$ in zwei interne Heaps $\mathcal{H}_1$ und $\mathcal{H}_2$.
Es gilt die Invariante, dass sich das kleinste Element immer in $\mathcal{H}$ befindet. Bei der *extract_min*-Operation muss $\mathcal{H}$ also nach dem Entfernen wieder mit passenden Blöcken aus den sortierten Schichten von $\mathcal{L}$ aufgefüllt werden.
Brengel et al. beschreiben in [3] zwei Varianten dieses *externen Array-Heap*. Eine vereinfachte Variante, die in der Praxis schnelle Berechnungszeiten ermöglicht und hier vorgestellt werden soll, sowie eine zweite Variante, die I/O-Optimalität erreicht und in [3] wird.



(a) Bereits befüllter *Array-Heap* **(b)** Einfügen eines Elements

Abbildung 5 Die Skizze veranschaulicht den Aufbau des vereinfachten *externen Array-Heap*. Abbildung 5a zeigt beispielhaft eine bestehende Füllung der Datenstruktur, während Abbildung 5b das Vorgehen der *insert*-Operation veranschaulicht.

3.1 Vereinfachter externer Array-Heap

Bei der vereinfachten Version des *externen Array-Heaps* wird der interne Heap $\mathcal{H}$ in zwei Heaps aufgeteilt, $\mathcal{H}_1$ und $\mathcal{H}_2$. Dabei speichtert $\mathcal{H}_1$ die neu hinzugefügten Elemente und enthält maximal $2cM$ Elemente. $\mathcal{H}_2$ speichert maximal die kleinsten B Elemente jedes belegten Slots j in jeder Schicht $\mathcal{L}_i$. Es befinden sich also maximal $cM(2 + L)$ Elemente im internen Speicher.

Insert: Ein neues Element wird zuerst in $\mathcal{H}_1$ eingefügt. Sollte $\mathcal{H}_1$ dabei komplett gefüllt werden, werden die größten cM Elemente aus $\mathcal{H}_1$ nach $\mathcal{L}_1$ verschoben und bilden die Folge S. Die kleinsten B Elemente verbleiben im Hauptspeicher und werden nach $\mathcal{H}_2$ verschoben. Sei nun $i = 1$. Sollte $\mathcal{L}_i$ einen leeren Slot besitzen, wird die Folge S in diesem gespeichert. Andernfalls ist in $\mathcal{L}_i$ nicht ausreichend Platz und es müssen zwei Fälle unterschieden werden: (i) mindestens 2 Slots können zu einem einzelnen Slot vereint werden, damit ein Slot frei wird und die Folge S gespeichert werden kann, andernfalls (ii) werden alle Slots von $\mathcal{L}_i$, sowie die Elemente der Folge S zu einer neuen sortierten Folge S' verschmolzen und zur nächsten Schicht $\mathcal{L}_{i+1}$ verschoben, um dort wie oben wieder zu prüfen, ob es einen freien Slot gibt.

▶ **Example 4.** Abbildung 5b verdeutlicht das Vorgehen bei der *insert*-Operation. In dem Beispiel ist der zweite Fall zu beobachten. $\mathcal{H}_1$ wird durch das Hinzufügen eines Elements vollständig gefüllt, also bilden die größten cM Elemente die Folge S, welche in die erste Schicht $\mathcal{L}_1$ verschoben wird. Da diese keine freien Slots enthält, werden die Slots der Schicht $\mathcal{L}_1$ mit der Folge S zu einer neuen Folge S' verschmolzen und in einem freien Slot der Schicht $\mathcal{L}_2$ einsortiert. Die kleinsten B Elemente aus $\mathcal{H}_1$ werden zusätzlich nach $\mathcal{H}_2$ verschoben.

Extract_min: Das kleinste Element befindet sich entweder in $\mathcal{H}_1$ oder $\mathcal{H}_2$. In beiden Fällen wird das Element entfernt und ausgegeben. Sollte das Element in $\mathcal{H}_1$ gewesen sein, ist nichts weiter zu tun. War das Element in $\mathcal{H}_2$, korrespondiert es zu einem Slot j einer Schicht $\mathcal{L}_i$. Sollte es das letzte Element in $\mathcal{H}_2$ gewesen sein, das mit Slot j in Schicht $\mathcal{L}_i$ assoziiert wird, wird $\mathcal{H}_2$ mit den nächsten B Elementen aus Slot j aufgefüllt. Anschließend wird geprüft, ob durch diese Verschiebung Slots entstanden sind, die weniger als $l_i/2$ Elemente enthalten. Sollte das der Fall sein, werden sie verschmolzen, um neue Slots freizumachen.

▶ **Example 5.** Das Vorgehen der *extract_min*-Operation kann ebenfalls mit Abbildung 5b verdeutlicht werden. Nehmen wir an, dass von den zuvor kleinsten B Elementen, die von $\mathcal{H}_1$ nach $\mathcal{H}_2$ verschoben wurden, nur noch ein Element x in $\mathcal{H}_2$ ist, also das letzte Element darstellt, dass mit dem neu gefüllten Slot j in Schicht $\mathcal{L}_2$ assoziiert wird und ebenfalls das zu extrahierende Minimum in $\mathcal{H}_2$ ist, da $\mathcal{H}_1$ aktuell leer ist. Bei der *extract_min*-Operation würde das Element x nun entfernt und ausgegeben und $\mathcal{H}_2$ würde mit den nächsten B Elementen aus Slot j aufgefüllt werden.

Nach *Brengel et al.* in [3] ergibt sich für die I/O Kosten und den Speicherplatzbedarf des *externen Array-Heaps* das folgende Bild:

▶ **Theorem 6.** *Unter der Annahme, dass L konstant ist, benötigen die* ***insert*** *und* ***extract_min*** *Operationen $O(1/B)$ amortisierte I/Os. Der benötigte externe Speicherplatz beträgt $(2X/B) + L$ Pages, wobei X die aktuelle Anzahl an Elemente im Heap ist. Der benötigte interne Speicherplatz ist $cM(2 + L)$.*

4 Experimente

Brengel et al. haben die zuvor vorgestellten Externspeicher-*PQs* in [3] ausführlichen Tests unterzogen und mit anderen bekannten Implementationen verglichen. Dafür haben sie *external Radix-Heaps*, *Array-Heaps*, *Buffer Trees* [2] und *B-trees* [7] mit der **LEDA-SM** [5] Bibliothek implementiert, die eine Erweiterung der **LEDA** [9] Bibliothek ist und vergleichen diese mit *Fibonacci Heaps* [8], *k-ary Heaps* [11], *Pairing Heaps* und *internal Radix-Heaps* [1].
Dabei unterziehen sie diese acht *PQ*-Varianten den folgenden Tests:

1. **Insert-All-Delete-All.** Bei diesem Test werden N Elemente eingefügt und anschließend N Elemente entfernt. Dieser Test soll die reine I/O-Performanz der verschiedenen *PQs* bewerten. Die Schlüssel der *PQs* sind dabei natürliche Zahlen aus dem Bereich $[0, 10^7]$.
2. **Intermixed insertions and deletions.** Bei diesem Test sollen die Geschwindigkeit der I/Os, sowie die *CPU*-Geschwindigkeit der internen Speicherbestandteile der externen *PQs* getestet werden. Dafür werden zuerst 20 Millionen Schlüssel zu den Warteschlangen hinzugefügt, um anschließend eine zufällige Folge von **insert** und **extract_min** Operationen auszuführen. Dabei wird ein Einfügen mit Wahrscheinlichkeit 1/3 durchgeführt und dementsprechend ein Extrahieren mit 2/3.
3. **Dijkstra's shortest-path algorithm.** Bei diesem Test wird der *Dijkstra* Algorithmus für kürzeste Pfade für internen Speicher auf sehr großen Pfaden simuliert. Eine Folge von Einfüge- und Extraktionsoperationen des Algorithmus einer Internspeicher-*PQ* wird zum Testen der Externspeicher-*PQs* verwendet, um deren Verhalten zu beobachten.

Am interessantesten sind die Ergebnisse für die in dieser Arbeit vorgestellten Externspeicher-*PQs* aus [3]. Für detaillierte Ergebnisse sei an das Paper [3] von *Brengel et al.* verwiesen. Der *externe Radix-Heap* ist nach den Ergebnissen der drei Tests die schnellste *PQ* basierend auf natürlichen Zahlen. Da hier keine zusätzliche Datenstruktur die kleinsten Elemente speichern muss, wie es bei *Array-Heaps* und *Buffer-Heaps* der Fall ist, wird die CPU-Zeit deutlich reduziert. Bezogen auf die Zeit, die die Einfüge- und Extraktionsoperationen verbrauchen, ist der *externe Radix-Heap* etwa $2,5$ mal schneller als der *Array-Heap* und 6 mal schneller als der *Buffer-Heap* (siehe Abbildung 1). Zusätzlich benötigen sie am wenigsten I/Os, von denen nur etwa 15% zufällig sind. Allerdings ist ihre Verwendung etwas beschränkter, da sie nur auf monotonen Warteschlangen eingesetzt werden können.
Für *Array-Heap* ergeben die Tests, dass die **insert** Operation eine enorme Verlangsamung der CPU-Zeit mit sich bringt. Diese sind bis zu zehn mal langsamer als die des *externe Radix-Heap*. Dafür erreichen sie eine leicht bessere Performanz bei **extract_min** Operationen und

sind im Vergleich zu *Buffer-Heaps* fast neun mal schneller. In der Summe der Einfüge- und Extraktionsoperationen ergibt das eine CPU-Zeit etwa drei mal schneller als *Buffer-Heaps* und $2,5$ langsamer als *externe Radix-Heap*. Da sie allerdings vielseitiger einsetzbar sind als *externe Radix-Heap*, sind sie die schnellsten *general-purpose PQs*. Bezogen auf das I/O-Verhalten, arbeiten sie nur leicht schlechter als *externe Radix-Heap* und nur 2% der I/Os sind zufällig.

5 Fazit

Die Testdaten aus Tabelle 1 [3] zeigen, dass Internspeicher-*PQs* versagen, wenn die Daten zu groß werden, um komplett im internen Speicher verarbeitet werden zu können. Des Weiteren hat sich herausgestellt, dass in allen Testfällen der *radix heap* die schnellste *PQ* darstellt. Die Struktur ist einfach zu implementieren und bietet effektive I/O und CPU-Zeit Performanz. Allerdings sind *radix heaps* auf natürliche Zahlen und monotone heaps beschränkt. Im Allgemeinfall werden dadurch *array heaps* zu einer besseren Wahl.

Insert/Extract_Min time performance of the external queues (in secs)				
N [$*10^6$]	radix heap	array heap	buffer tree	B-Tree
1	6/24	18/11	56/34	11287/259
5	17/97	74/63	148/309	66210/1389
10	35/178	353/89	201/882	-
25	85/372	724/295	311/2833	-
50	164/853	1437/645	445/6085	-
75	246/1416	2157/1005	569/9880	-
100	325/1957	2888/1408	*	-
150	478/3084	4277/2297	*	-
200	628/4036	5653/3234	*	-
Insert/Extract_Min time performance of the internal queues (in secs)				
N [$*10^6$]	Fibonacci heap	k-ary heap	pairing heap	radix heap
1	3/32	4/33	3/19	3/11
2	6/73	8/75	6/45	5/27
5	17/208	21/210	14/126	11/71
7.5	172800*/-	32/344	22/207	18/124
10	-/-	43/482	30/291	23/162
20	-/-	172800*/-	172800*/-	172800*/-
Random/Total I/Os fpr external queues				
N [$*10^6$]	radix heap	array heap	buffer tree	
1	44/420	24/720	228/668	
5	422/3550	120/4560	16722/21970	
10	1124/9620	168/9440	35992/47297	
25	2780/21820	570/29520	93789/123285	
50	7798/56830	1288/66160	190147/249955	
75	12466/89370	2016/102480	286513/376625	
100	17736/124740	2776/139760	*	
150	27604/192500	4216/210080	*	
200	38284/211570	5712/284320	*	

Tabelle 1 Ergebnisse des **inster-all-delete-all** Tests [3].

Time performance on mixed operations			
N [$*10^6$]	radix heap	array heap	buffer tree
50	544	770	4996
75	609	945	5862
100	619	1027	6029
Random/Total I/Os on mixed operations			
50	2935/19615	22325/26997	153321/177201
75	5128/26752	24256/28384	171615/196647
100	5782/30094	24220/28380	171658/196578
Time performance on the Dijkstra's test			
15	441	812	1524
20	1266	2030	5602
25	2750	4795	-
Total/Random I/Os on the Dijkstra's test			
15	422/5054	160/7800	33359/40887
20	2630/20790	528/23920	156155/176275
25	5810/48650	1600/80640	-

Tabelle 2 Ergebnisse der **intermixed insertions and deletions** und **Dijkstra's shortest-path algorithm** Tests [3].

Literatur

1 R. Ahuja, K. Mehlhorn, J.B. Olrin, and R.E. Tarjan. Faster algorithms for the shortest path problem. In *Journal of the ACM*, pages 213–223, 1990.

2 L. Arge. The buffer tree: A new technique for optimal i/o-algorithms. In *Workshop on Algorithms and Data Structures*, pages 334–345, 1995.

3 Klaus Brengel, Andreas Crauser, Paolo Ferragina, and Ulrich Meyer. An Experimental Study of Priority Queues in External Memory. In *Algorithm Engineering*, pages 345–359. Springer-Verlag Berlin Heidelberg, Juli 1999. doi:10.1007/3-540-48318-7.

4 G.S. Brodal and J. Katajainen. Worst-case efficient external memory priority queues. In *Scandinavian Workshop on Algorithm Theory*, pages 107–117, 1998.

5 A. Crauser and K. Mehlhorn. Leda-sm, extending leda to secondary memory. In *Workshop on Algorithmic Engineering*, 1999.

6 E. Dijkstra. A note on two problems in connection with graphs. In *Num. Math. 1*, pages 269–271, 1959.

7 R. Fadel, K.V. Jakobsen, J. Katajainen, and J. Teuhola. External heaps combined with effective buffering. In *Proc. Computing: The Australasian Theory Symposium*, 1997.

8 M.L. Fredman and R.E. Tarjan. Fibonacci heaps and their use in improved network optimization algorithms. In *Journal of the ACM*, pages 596–615, 1987.

9 K. Mehlhorn and S. Näher. Leda: A platform for combinatorial and geometric computing. In *Communications of the ACM*, pages 96–102, 1995.

10 J.S. Vitter and E.A.M. Shriver. Optimal algorithms for parallel memory i: Two-level memories. In *Algorithmica*, pages 110–147, 1994.

11 J.W.J. William. Algorithm 232 (heapsort). In *Communications of the ACM*, pages 347–348, 1964.

Entropy Compressed Rank/Select Dictionary

Dennis Rohde[1]

1 **TU Dortmund, Germany**
dennis.rohde@tu-dortmund.de

Abstract

Rank/Select dictionaries, that are data structures for *ordered sets* $S \subset U = \{0, \ldots, n-1\}$, have many applications as components of *succinct data structures* for functions, strings, trees, graphs etc. The main purpose of rank/select dictionaries is to provide *time-efficient* $rank(x, S)/select(i, S)$ queries, that are function evaluations which return the number of elements in S that are not greater than x and the position of the i-th smallest element in S respectively, while occupying as little *space* as possible. Previously *plain* (size: $n + o(n)$ bits) implementations (with uncompressed auxiliary data structures) of rank/select dictionaries were common, but showed poor performance to real data sizes, especially when the set is *sparse*. Four newly proposed rank/select dictionaries are examined: esp, recrank, vcode and sdarray. With a size of $n \cdot H_0(S) + o(n)$ bits (H_0: *zeroth order empirical entropy*), these implementations approach the lower bound of $n \cdot H_0(S)$ bits of size, as experimental results show. Further all dictionaries provide time-efficient rank/select-queries in the average case. In practice as in theory, these dictionaries improve size and query-time compared to previous proposed ones.

Keywords and phrases Entropy Compressed, Rank/Select Dictionaries, Practical Focus, Succinct Data Structures, Constant Time Operations

Digital Object Identifier 10.4230/LIPIcs.BADS.2017.14

1 Introduction

In this work four implementations of Rank/Select Dictionaries, presented in [5], are described and analysed. We are mainly focussing on the underlying ideas and the theoretical concepts, rather than real implementations.
We first introduce the concepts of Rank/Select Dictionaries and give detailed descriptions of previously common implementations, namely *Succinct Implementations*.
Further we describe a simple *Entropy Compressed Implementation*, namely ent, which is based on the former. In the following the four new implementations from [5], named esp, recrank, vcode and sdarray are described in detail as well as their benefits and disadvantages are discussed.

2 Preliminaries

In this work the *word RAM model* is assumed, here arithmetical and logical operations can be performed on two $\mathcal{O}(\log_2(n))$ integers in constant time, as well as consecutive $\mathcal{O}(\log_2(n))$ bits can be saved/loaded to/from memory in constant time. For space-efficient computations in this model, *succinct data structures* are crucial. These data structures are using $(1 + o(1)) \cdot \log_2(L)$ bits to represent an object from a universe with cardinality L.
Here a data structure of interest is the Rank/Select-Dictionary which represents an ordered set S, that is a subset of an universe $U := \{0, \ldots, n-1\}$, and performs *time-efficient* operations rank and select. These operations are defined as follows:

- rank(x,S): Number of elements in S that are less than or equal to x.
- select(i,S): Index of the i-th smallest element in S.

A simple representation of such an ordered set is a bit vector B of length n, indicating the elements of the universe U that are members of S.

$$B[i] = \begin{cases} 1 & \text{if } U[i] \in S \\ 0 & \text{otherwise} \end{cases}.$$

This *plain representation* uses $n = \log_2(2^n)$ bits for representing one of 2^n possible sets S. The function rank(x,S) is the number of ones in $B[0, x]$ and select(i,S) is the position of the i-th one from the left in B. To evaluate these functions in *constant time* auxiliary data structures proposed in [4] can be utilized.

When S has cardinality m (B has m ones) there are only $\binom{n}{m}$ possible sets, hence at minimum $\mathcal{B}(n, m) := \lceil \log_2(\binom{n}{m}) \rceil$ bits are needed for representing S through B, what is less than the size of the *plain representation* when $m < n$. Note that this lower bound is approached from above by n times the *zero-th order empirical entropy* $H_0(B) := \frac{m}{n} \log_2(\frac{n}{m}) + \frac{n-m}{n} \log_2(\frac{n}{n-m})$.

Typically there are two applications for Rank/Select Dictionaries, one is for *dense sets*, where $m \approx \frac{n}{2}$ and the other one is for *sparse sets*, where $m \ll n$ or $n - m \ll n$. Because in the word RAM model $\mathcal{O}(\log_2(n))$ consecutive bits can be accessed in constant time, we often partition data into blocks of size $\Theta(\log_2(n))$ bits and compress pointers to these blocks utilizing sparse sets, which lie in the focus of this work.

3 Succinct Implementations

Plain representations of the Rank/Select Dictionary that are using $n + o(n)$ bits were previously common in succinct data structures. In these implementations the bit vector B is partitioned into *large blocks* of size $l := \log_2^2(n)$, which are then partitioned into *small blocks* of size $s := \frac{\log_2(n)}{2}$.

A *rank directory* $R_l[0 \dots \frac{n}{l}]$ stores the results of the rank function at every boundary of a large block using $\mathcal{O}\left(\frac{n}{\log_2(n)}\right)$ bits. An additional rank directory $R_s[0 \dots \frac{n}{s}]$ stores the rank results, relative to the corresponding entries in R_l at the boundaries of the small blocks, using $\mathcal{O}\left(n \frac{\log_2(\log_2(n))}{\log_2(n)}\right)$ bits. Let $popcount(i, j)$ define the number of ones in $B[i \dots i + j]$, which can be computed in constant time for $1 \leq j \leq s$ using a pre-computed byte-aligned table of size $\mathcal{O}\left(\sqrt{n} \cdot \log_2^2(n)\right)$ (see [2]).

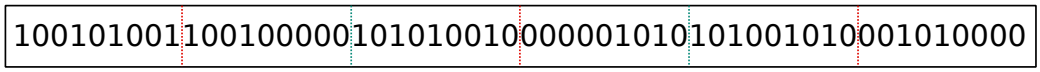


Figure 1 Bitvector B with large block boundaries in green and small block boundaries in red for $l := 18, s := 9$. The rank directories are $R_{18} := [0, 6, 12]$ and $R_9 := [0, 4, 0, 4, 0, 4]$.

The function rank can be calculated in constant time in the following way:

$$rank(x, S) = R_l\left[\left\lfloor \frac{x}{l} \right\rfloor\right] + R_s\left[\left\lfloor \frac{x}{s} \right\rfloor\right] + popcount\left(\left\lfloor \frac{x}{s} \right\rfloor \cdot s, x \mod s\right)$$

The select function is implemented as binary search on the rank function and therefore has worst case running-time of $\mathcal{O}(\log_2(n))$.

4 Entropy Compressed Implementations

The following implementation of the Rank/Select Dictionary is called ent. Here the small blocks (see section 3) are encoded by *enumerative code* (see [1]) to save space.

Let $s \in \Theta(\log_2(n))$ denote the length of a small block B_i, for $1 \leq i \leq \frac{n}{s}$ and u_i denote the number of ones in the block. We can identify each possible block by a number $id(B_i) \in [0, \left\lceil \binom{s}{u_i} \right\rceil - 1]$, for $id(B_i) := \sum_{j=1}^{u_i} \binom{j}{s-p_j-1}$ where $p_j = select(j, B_i)$ is the position of the j-th one in the block. This function can be calculated using the precomputed *popcount* table and looking up the positions where the number of set bits increases through forward search. For every possible B_s the code word $id(B_s)$ can be stored using $\mathcal{B}(s,u) = \left\lceil \log_2(\binom{s}{u}) \right\rceil$ bits. Let u_i denote the number of ones in small block i. Because $\sum_{i=1}^{\frac{n}{s}} s = n$ and $\sum_{i=1}^{\frac{n}{s}} u_i = m$, less than $\log_2\left(\binom{n}{m}\right) + \frac{n}{s}$ bits are needed for all code words (see [6, Lemma 4.1]). These are less than or equal to $n \cdot H_0(B) + o(n)$ bits, therefore we call this implementation *entropy compressed*.

Unfortunately additional pointers to the code words must be stored, since they have different sizes. These pointers need $\mathcal{O}\left(n \frac{\log_2(\log_2(n))}{\log_2(n)}\right) = o(n)$ bits, therefore $n \cdot H_0(B) + o(n)$ bits are needed for the complete Rank/Select Dictionary.

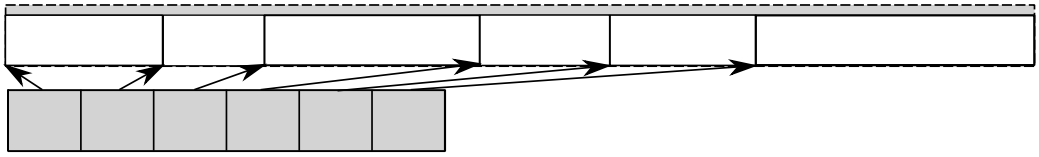


Figure 2 Above: Bitvector B consisting of compressed small blocks in white. Underneath: Array of pointers to small block's starting positions.

In [5] four new entropy compressed implementations of Rank/Select Dictionaries are introduced which focus on practical data sizes. In the following we describe the functionality of every of those implementations.

4.1 Estimating Pointer Information

The idea behind the following implementation, which is based on the ent implementation described above, is to get rid of the pointers to the code words (see section 4) by *estimating* code word sizes on the fly from *rank information*. Since the $\mathcal{O}\left(n \frac{\log_2(\log_2(n))}{\log_2(n)}\right)$ bits for pointers to encoded small blocks are $\Theta\left(n \cdot H_0(S)\right)$ bits for practical (small) data sizes, this makes a noteworthy saving of space.
Let B be a bit vector of length n with m ones and $B_1, \ldots, B_{\frac{n}{s}}$ be a partition of B into blocks of size $s \leq n$. We describe a way of storing all code words representing the blocks of B with $n \cdot H_0(B) + \frac{n}{s} + 1 = nH_0(B) + o(n)$ bits, storing code words at positions obtained through size estimation using H_0. Since we obtain an upper bound for the size of code words, it is sometimes necessary to insert gap bits to estimate the right starting positions. Every Block of the partition is encoded by *enumerative code* (see [1]).

▶ **Theorem 1** (see [5]). *Let $L(B)$ be the length of the code word for B using* enumerative code *(see [1]). It holds that $L(B) \leq n \cdot H_0(B) + 1$.*

▶ **Theorem 2** (see [5]). *The following holds for the cumulated size of code words:* $\sum_{i=1}^{\frac{n}{s}} L(B_i) \leq n \cdot H_0(B) + \frac{n}{s} + 1$

Let $B' := B[0 \dots u]$ be a prefix of B, for $0 \leq u \leq n$. Using theorem 1 and theorem 2 we know, that we can store the code word for B' with up to $u \cdot H_0(B') + o(u)$ bits. Through rank queries the number of ones in B' can be obtained and since the length of B' is also known the function $H_0(B')$ can be calculated.

The data structure esp is obtained as follows: We partition B into *super large blocks* of size $k := \log_2^3(n)$, which are partitioned into *large blocks* of size $l := \log_2^2(n)$, which are then partitioned into *small blocks* of size $s := \frac{\log_2(n)}{2}$. Each small block is encoded by *enumerative code*.
The position for the i-th small block SB_i can be calculated as follows: Let x be the position in B, such that SB_i starts at $B[x]$. Let SLB_i denote the superlarge block containing SB_i and LB_i denote the large block SB_i lies in. The infix of B that contains all large blocks from the boundary of SLB_i up to LB_i is denoted LB'_{x_l} and the infix of B where all small blocks from the boundary of LB_i up to SB_i lie in is denoted SB'_{x_s}.
Further define $x_l := \left\lfloor \frac{x}{l} \right\rfloor$ as well as $x_s := \left\lfloor \frac{x}{s} \right\rfloor$, such that $l_r := R_l[x_l]$, respective $l_s := R_s[x_s]$ (see Section 3) denote the results of rank for LB_i and SB_i. Let slp be the absolute starting position of SLB_i and lp be the starting position of LB_i relative to slp, further sp the starting position of SB_i relative to lp. We calculate lp and sp as follows (see [5], Figure 3):

$$lp = l \cdot x_l \cdot H_0(LB'_{x_l}) = \underbrace{l_r \cdot \log_2\left(\frac{l \cdot x_l}{l_r}\right)}_{\text{space for ones in } B[slp \dots lp]} + \underbrace{(l \cdot x_l - l_r) \cdot \log_2\left(\frac{l \cdot x_l}{l \cdot x_l - l_r}\right)}_{\text{space for zeros in } B[slp \dots lp]}$$

$$sp = s \cdot x_s \cdot H_0(SB'_{x_s}) = \underbrace{l_s \cdot \log_2\left(\frac{s \cdot x_s}{s_r}\right)}_{\text{space for ones in } B[lp \dots sp]} + \underbrace{(s \cdot x_s - s_r) \cdot \log_2\left(\frac{s \cdot x_s}{s \cdot x_s - s_r}\right)}_{\text{space for zeros in } B[lp \dots sp]}$$

The starting position of SB_i is then at:

$$slp + lp + sp$$

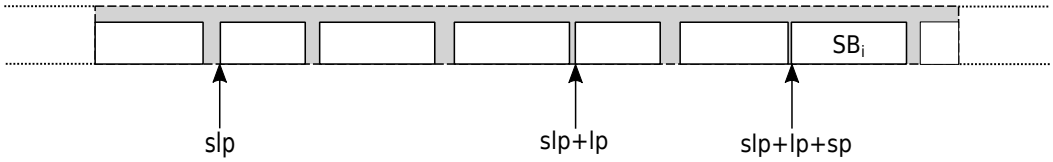


Figure 3 Compressed representation of B with enumerative encoded small blocks, aligned to estimated pointers.

We store the pointers slp explicitly to frequently reset estimation errors. The rank directories for large blocks and small blocks are also stored explicitly. To calculate $rank$, R_l and R_s are queried, then the position of the small block is calculated, which is then decoded so *popcount* can be applied (see Section 3). For *select* an approach described in [3] can be applied which leads to constant running-time using $o(n)$ bits auxiliary data structures.

4.2 Recursive Rank

The following idea is based on the observation, that sparse bit vectors can be contracted by partitioning and leaving out every block that contains only 0s. This technique works even better when it is applied recursively.
Let $B[0 \dots n-1]$ be a bit vector with m ones, that is partitioned into blocks $B_0 \dots B_{\frac{n}{s}}$ of size s. A block, where every element is 0 is called *zero block*, whereas every other block is called *non-zero block*. Let $B_c[0 \dots \frac{n}{s} - 1]$ be the *contracted bit vector* of B, where:

$$B_c[i] = \begin{cases} 1 & \text{if } B_i \text{ is a non-zero block} \\ 0 & \text{else} \end{cases}.$$

The *extracted bit vector* B_e is the concatenation of all non-zero blocks in original order.
We apply this reduction recursively, with B_e as the new input bit vector, until the resulting extracted bit vector is dense – more than every fourth bit is 1. The resulting contracted and extracted bit vectors in the i-th iteration are denoted B_c^i and B_e^i respectively, where B_c^t and B_e^t are the final results.
Let $p(B) := \frac{m}{n}$ denote the probability for an element of B to be a 1, when every element is picked at same chance. If $p(B) = 1$ every element is one and if $p(B) = 0$ every element is zero. The reduction can not be applied then.
Else the probability for a block to be a *zero block* is $(1-p(B))^s$. If $(1-p(B))^s = \frac{1}{2}$, half of all blocks are *zero blocks*, therefore we set $s := \frac{1}{-\log_2(1-p(B^i))}$ in each iteration, where B^i is the input bit vector and $B^1 = B$. The expected length of B_e is then $\frac{n}{2}$, hence $p(B_e) \approx 2 \cdot p(B)$. The expected length of B_c is $-n \log_2(1-p(B))$.
This reduction is applied $t := \lfloor -\log_2(p(B)) \rfloor$ times, such that $p(B_e^t) > \frac{1}{4}$ with high probability. While B_e^i shrinks in every iteration, B_c^i slowly grows, due to decreasing block sizes (see Figures 4, 5).

B^1	01000	00000	10001	00000	00010	00000
B_c	1	0	1	0	1	0

Figure 4 First iteration in RecRank. There are 4 ones in the input bit vector of length 30, therefore $p(B^1) = \frac{4}{30}$, $s = \frac{1}{-\log_2(1-\frac{4}{30})} \approx 5$ and $t = \lfloor -\log_2(\frac{4}{30}) \rfloor = 2$. The extracted bit vector is $B_e^1 = [010001000100010]$.

B^2	01	00	01	00	01	00	01	0
B_c^2	1	0	1	0	1	0	1	0

Figure 5 Last iteration in RecRank. There are 4 ones in the input bit vector $B^2 = B_e^1$ of length 15, therefore $p(B^2) = \frac{4}{15}$, $s = \frac{1}{-\log_2(1-\frac{4}{15})} \approx 2$. The extracted bit vector is $B_e^2 = [01010101]$ with $p(B_e^2) = \frac{4}{8} = \frac{1}{2} > \frac{1}{4}$.

Let T be the expected size of the resulting structure.

▶ **Theorem 3** (see [5]). *It holds that* $T \leq 1.44m \log(\frac{n}{m}) + m$ *bits.*

In [5] it is mentioned, that $m \log(\frac{n}{m}) + 1.44m$ is approximately $n \cdot H_0(S)$ for $m \ll n$, or $n - m \ll n$, therefore the expected size of RecRank is nearly entropy compressed for sparse sets.
To calculate Rank we apply the following equation recursively for every iteration:

$$rank(x, B) = rank\left(rank\left(\left\lfloor \frac{x}{s} \right\rfloor\right) \cdot s + (x \mod t) \cdot B_c\left[\left\lfloor \frac{x}{s} \right\rfloor\right], B_e\right)$$

Since every rank query takes constant time this takes time $\mathcal{O}\left(\log\left(\frac{n}{m}\right)\right)$ (note that $t = -log(p) = \log\left(\frac{n}{m}\right)$). For select we use the same approach taking time $\mathcal{O}\left(\log\left(\frac{n}{m}\right)\right)$.

4.3 Vertical Code

This data structure is vastly different from every data structure that was described here before. Here byte-based operations are used to maintain fast queries while occupying as little space as possible. Though this is not a real entropy compressed rank/select dictionary at worst case, in most cases its size is close to H_0.
Let $B[0 \dots n-1]$ be a bit vector with m ones, the *gap-sequence* of B is $d[0 \dots m-1]$, where:

$$d[i] := \begin{cases} select(i+1, B) - select(i, B) & \text{if } 1 \leq i \leq m-1 \\ select(1, B) & \text{if } i = 0 \end{cases}.$$

Let $p \in \mathcal{O}(\log(n))$ and $B_1 \dots B_{\frac{m}{p}}$ be a partition of d. Let $k \in \{0, \dots, \frac{m}{p} - 1\}, i \in \{0, \dots, m-1\}$, we define arrays S, T, V as follows:

- Let S be an array with $S[k] := select(B, k \cdot p)$, that holds evaluations of select.
- Let T be an array with $T[k] := \lfloor \log_2(\max_{j \in \{0,\dots,p-1\}} d[k \cdot p + j]) \rfloor$, such that all values in B_k can be represented by $T[k]$ bits.
- Let V_k be an array, such that $V_k[j]$ is an array of the $j+1-$th bits of the binary sequences of $B_k[1], \dots, B_k[p]$, for $0 \leq j \leq p-1$.

Let $b := \frac{i}{p}$ and $q := i \mod p$, then:

$$select(i, S) := S[b] + q + \sum_{j=b \cdot p}^{b \cdot p + q} d[j]$$

To calculate $\sum_{j=b \cdot p}^{b \cdot p + q} d[j]$ we count the ones obtained by the first q bits of each $V_k[0], \dots, V_k[T[i]]$ through *popcount* (see Figure 6) and sum them up using a bit-shift operation.

d	2	0	2	0	0	2	4	5
T	2				3			
0	0	0	0	0	0	0	0	1
1	1	0	1	0	0	1	0	0
2					0	0	1	1
			V_1				V_2	

Figure 6 Example of *VCode* for $B = [00110011100100001000001]$. Here $S = [0, 7]$.

If p is a multiple of eight, all operations are byte-aligned. Further Select has running-time $\mathcal{O}(\log_2(n))$, since it holds that $\forall i : d[i] \leq n$ and therefore $\forall i : T[i] \leq \log_2(n)$, further all other operations have constant running-time. Rank can be implemented using binary search on Select with running-time $\mathcal{O}(\log_2(m) \cdot \log_2(n))$.
The expected size of V, which dominates the size of the structure, is close to $m \log_2\left(\frac{n}{m}\right)$, therefore the complete size is close to $nH_0(S)$.

4.4 Sparse/Dense Array

Here we describe a method where input sets are distinguished between sparse sets and dense sets, then for each case a specialised data structure is utilized. If some preprocessing is done, one can calculate whether the input set is dense or sparse based on the number of ones in the set. If this is done, for the former darray constitutes a good data structure while sarray is an appropriate data structure for the latter.

4.4.1 Sparse Array

Let $B[0 \dots n]$ be a bit vector with m ones and $m \ll n$ or $n - m \ll n$. For $i \in \{0, \dots, m-1\}$ define:

$$x[i] := select(i+1, B)$$

Let $t = 1.44m$ be a parameter and $\mathbb{B}(x[i])$ define the binary sequence of $x[i]$. The following array stores the lower $w := \left\lfloor \log_2\left(\frac{n}{t}\right)\right\rfloor$ bits of the $x[i]$ explicitly:

$$L[i] := \mathbb{B}\left(x[i]\right)[0 \dots w]$$

The upper $z := \lceil \log_2(t) \rceil$ bits of the $x[i]$ are represented through a bit array of length $m + t$, defined as follows:

$$H[k] = \begin{cases} 1 & \text{if } k = \frac{x[i]}{2^w} + i, i \in \{0, \dots, m-1\} \\ 0 & \text{else} \end{cases}.$$

This is an unary coding of gaps between the values represented by the upper bits of the $x[i]$ (see Figure 7), with m ones and t zeros.

B	01001001000000000010000010100011							
x[i]	1	4	7	18	24	26	30	31
$\frac{x[i]}{2^1}$	0	2	3	9	12	13	15	15
$\frac{x[i]}{2^1}$+i	0	3	5	12	16	18	21	22
L	10100001							
H	10101000000100010100011							

0000	1
0010	0
0011	1
1001	0
1100	0
1101	0
1111	0
1111	1

Figure 7 Example of *SArray* with $z = \lceil \log_2(1.44 \cdot 8) \rceil = 4$ and $w = 1$.

Select can be calculated as follows, in constant time:

$$select(i, B) = (select(i, H) - i) \cdot 2^w + L[i]$$

Rank can be implemented as binary search on Select with running-time $\mathcal{O}(\log_2(\frac{n}{m}))$.
SArray uses size $m+t$ bits for H and $m \cdot \log_2\left(\frac{n}{t}\right)$ bits for L. When $t = m \cdot \log_2(e) \approx 1.44m$, the size of SArray is minimal and close to $H_0(B)$.

4.4.2 Dense Array

Let $B[0 \dots n-1]$ be a bit vector with $m \approx \frac{n}{2}$ ones. B is partitioned into blocks with $L \in \mathcal{O}(\log_2^2(n))$ ones each.
Let $L_2 \in \mathcal{O}(\log_2^4(n))$, $L_3 \in \mathcal{O}(\log_2(n))$ be parameters and S_l, S_s be arrays.
For $i \in \{0, \dots, \frac{n}{L} - 1\}$ we define:

$$P_l[i] := select(i \cdot L, B)$$

We distinguish between two types of blocks using the following criteria:

- If $(P_l[i] - P_l[i-1]) > L_2$ store select results explicitly in S_l.
- If $(P_l[i] - P_l[i-1]) < L_2$ store select results of the $j \cdot L_3$-th ones, relative to $P_l[i]$ in S_s, for $j \in \{0, \dots, \frac{L_2}{L_3}\}$.

While the former uses up to $\frac{n}{L_2} \cdot L \log_2(n)$ bits the latter uses up to $\frac{n}{L_3} \log_2(L_2)$ bits. Additionally $\mathcal{O}\left(\frac{n}{L} \log_2(n)\right)$ are needed for P_l. Therefore P_l, S_l and S_s need $o(n)$ bits, hence DArray uses $n + o(n)$ bits.
To calculate Select, we query $P_l\left[\left\lceil \frac{i}{L} \right\rceil\right]$ and $P_l\left[\left\lceil \frac{i}{L} - 1 \right\rceil\right]$ and determine whether the block is greater than L_2 or not. If it is, we query the value from S_l. Otherwise we query the position of the $i \cdot L_3$-th one from S_s and start a sequential search in the corresponding block in $\mathcal{O}\left(\frac{L_2}{\log_2(n)}\right)$ time.
Rank can be implemented by a binary search on select in $\mathcal{O}\left(\log_2(n)\right)$ time.

5 Discussion

In [5] the results of an experiment are presented, that is comparing these four implementations to similar ones. The *sizes* of these data structures can be compared to n times the *zero-th order entropy*. Here it can be observed, that for smaller input sizes these implementations are close to $n \cdot H_0(S)$.
For larger input only Esp stays close to $n \cdot H_0(S)$, while the other implementations sizes grow stronger. In conclusion most of the presented data structures benefit only from small inputs. Furthermore *running-time* of Rank and Select are analysed. It can be observed, that running-times are nearly constant on the input length for all conducted data structures. Running-times of SDArray show to be superior, while running-times of Esp, RecRank and VCode show to be similar to the ones of the common implementations.
In conclusion these four Rank/Select Dictionaries show to be superior to common implementations for small input sizes, but mostly are very specialised and do not guarantee superior sizes and/or running-times in general, but for their specific applications.

References

1 T. Cover. Enumerative source encoding. *IEEE Transactions on Information Theory*, 19(1):73–77, January 1973.

2 R. González, Sz. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Poster Proc. Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA)*, pages 27–38, Greece, 2005. CTI Press and Ellinika Grammata.

3 Dong Kyue Kim, Joong Chae Na, Ji Eun Kim, and Kunsoo Park. *Efficient Implementation of Rank and Select Functions for Succinct Representation*, pages 315–327. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

4 J. Ian Munro. *Tables*, pages 37–42. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996.

5 Daisuke Okanohara and Kunihiko Sadakane. *Practical Entropy-Compressed Rank/Select Dictionary*, pages 60–70.

6 Rasmus Pagh. Low redundancy in static dictionaries with constant query time. *SIAM Journal on Computing*, 31(2):353–363, 2001.

CSA++: Schnelle Suche in Texten mit großen Alphabeten

Lena Rolf (156072)[1]

1 **Technische Universität Dortmund, Dortmund, Deutschland**
lena.rolf@tu-dortmund.de

Zusammenfassung

Das *Compressed Suffix Array* (CSA) ist ein etablierter Volltextindex. Zwar benötigt die CSA-Indizierung in der Regel mehr Speicherplatz als andere Textindices, allerdings wird die zur Pattern Suche benötigte Zeit kaum von der Größe des dem Text zugrundeliegenden Alphabets beeinflusst. Wegen dieser Eigenschaft diente CSA als Basis für den in dieser Arbeit behandelten Index CSA++ von Gog, Moffat und Petri, der für Texte mit großen Alphabeten konstruiert wurde. Um den Speicherplatzverbrauch zu reduzieren, werden selten im Text vorkommende Symbole in einer separaten Struktur gespeichert. Für die restlichen Zeichen wird das *Successor Array* mithilfe einer erweiterten Version der *Uniform Elias-Fano Codierung* gespeichert. Diese enthält einen zusätzlichen Blocktyp, der Sequenzen über einem großen Wertebereich effizient codiert. Beim experimentellen Vergleich mit vorherigen CSA-Implementierungen und verschiedenen Varianten des FM-Index zeigt sich, dass CSA++ für große Alphabete einen ähnlichen Speicherbedarf wie speicherplatzeffiziente FM-Index Varianten hat, während die Zeit für das Zählen der Vorkommen des Suchstrings im Vergleich geringer ist. Die Analyse der Struktur eines dabei konstruierten CSA++ Index zeigt, dass die vorgenommenen Veränderungen für einen großen Anteil des Alphabets genutzt werden.

1998 ACM Subject Classification H.3.1 Content Analysis and Indexing – please refer to `http://www.acm.org/about/class/ccs98-html`

Keywords and phrases Suffix Array, Textindizierung, Pattern Search

Digital Object Identifier 10.4230/LIPIcs.BADS.2017.21

1 Einleitung und Motivation

In vielen verschiedenen Bereichen ist es hilfreich durch Strings repräsentierte Daten nach einem bestimmten Substring oder *Pattern* zu durchsuchen. Von alltäglichen Aufgaben wie der Suche eines Wortes in einem Textdokument bis zu spezifischen Anwendungen wie der Suche nach einem Pattern in einer DNA-Sequenz. Gog, Moffat und Petri stellen in [4] einen Textindex vor, der speziell für Texte mit großen Alphabeten, z.B. aus Wörtern oder asiatischen Schriftzeichen, konzipiert ist. In dieser Arbeit werden zunächst das dem CSA++ Index zugrundeliegende *Compressed Suffix Array* von Sadakane [11] beschrieben und anschließend verschiedene Aspekte des CSA++ vorgestellt, die zur Verringerung des Speicherbedarfs und der Suchzeit für Texte mit großen Alphabeten beitragen.

2 Das Compressed Suffix Array

Sei T ein Text der Länge n über einem Alphabet Σ mit σ verschiedenen Zeichen und einem *Stoppzeichen* $\$ \notin \Sigma$ an letzter Stelle des Textes und P ein Pattern der Länge m mit

$P = p_0 \dots p_{m-1}$ und $p_i \in \Sigma$. Dann ist das Ergebnis einer *Counting Query* (CQ) für T und P die Anzahl der Vorkommen von P in T.

Für jede Suchanfrage an einen Text muss dieser linear nach dem Pattern durchsucht werden. Durch die einmalige Konstruktion eines *Textindex* kann die Suche nach einem beliebigen Pattern ohne Überprüfung jedes Textzeichens durchgeführt werden. Das *Compressed Suffix Array* (CSA) ist ein solcher Textindex und basiert auf der Idee des *Suffix Array*s.

In Abschnitt 2.1 wird erläutert wie ein *Suffix Array* konstruiert und zur Suche genutzt werden kann. In den darauf folgenden Abschnitten 2.2 und 2.3 werden Idee und Funktionsweise des CSA sowie Aspekte der Speicherung behandelt.

2.1 Das Suffix Array

Für die Konstruktion eines *Suffix Arrays* (SA) werden alle Suffixe $T[i, n-1]$ mit $i \in \{0, \dots, n-1\}$ lexikographisch sortiert und ihre Position im Text in dieser Reihenfolge mit $\mathcal{O}(n \log(n))$ Bits gespeichert. Die Sortierung der Suffixe hat zur Folge, dass alle Suffixe, die mit dem gleichen String anfangen, an aufeinanderfolgenden Stellen im SA gespeichert sind. Um zu bestimmen, wie oft ein Pattern P in T vorkommt, reicht es also den Bereich im SA zu bestimmen, in dem alle Suffixe mit P beginnen. Dieser Bereich kann mit Hilfe des SA und dem Text T durch zwei binäre Suchen eingegrenzt werden. Eine solche Vorgehensweise erfordert jedoch eine Laufzeit von $\mathcal{O}(m \log(n))$.

2.2 Idee & Funktionsweise des komprimierten Suffix Arrays

Um ein SA zu einem CSA nach Sadakane zu komprimieren wird ein *Successor Array* Ψ konstruiert (vgl. [11]). In diesem Array wird für jedes Suffix $S = T[i, \dots, n-1]$ von T gespeichert, an welcher Stelle des SA die Position des auf S folgenden Suffix $T[i+1, \dots, n-1]$ enthalten ist. Zwischen SA und Ψ besteht also folgender Zusammenhang:

$$\Psi[i] = SA^{-1}[(SA[i] + 1) mod(n)]$$

Anhand von Beispiel 1 lässt sich diese Beziehung nachvollziehen: Die Position des Suffix „banana$“ ist im SA an Stelle 4 gespeichert. Der Eintrag für das Suffix „anana$“ befindet sich an Stelle 3. Demnach enthält Ψ einen Eintrag mit $\Psi[4] = 3$.

▶ **Beispiel 1.** Die folgenden Tabellen enthalten das SA bzw. das *Successor Array* für den Text „banana$“. Die Spalte „Suffix“ soll lediglich das Verständnis erleichtern, ist aber nicht Teil des SA.

SA

Index	Position	Suffix
0	6	$
1	5	a$
2	3	ana$
3	1	anana$
4	0	banana$
5	4	na$
6	2	nana$

Ψ :

Index	Ψ
0	4
1	0
2	5
3	6
4	3
5	1
6	2

Es kann gezeigt werden, dass es in einem *Successor Array* Ψ für jedes Zeichen aus $\Sigma \cup \{\$\}$ einen Bereich gibt, der eine aufsteigende Zahlenfolge enthält (vgl. [6]). Das *Successor Array* aus Beispiel 1 hat vier solcher Bereiche $\Psi[0], \Psi[1, 2, 3], \Psi[4]$ und $\Psi[5, 6]$. Durch diese Eigenschaft ist Ψ besser komprimierbar als das SA (Details zur Speicherung von Ψ in Abschnitt 2.3).

Um die Speichervorteile von Ψ ausnutzen zu können, muss die Suche ohne das SA und den Text T funktionieren. Dazu wird eine weitere Datenstruktur benötigt: Sei n_c die Anzahl der Vorkommen eines Zeichens $c \in \Sigma \cup \{\$\}$ in Text T. Das Array C enthält für jedes Zeichen c die Summe der Vorkommen der lexikographisch kleineren Zeichen, also $C[c] = \sum_{c' <_{lex} c} n_{c'}$. Zusätzlich enthält C einen Eintrag $C[EOF] = n$ für ein neues Zeichen $EOF \notin \Sigma$, wobei n gleich der Anzahl der Zeichen in T ist. Zur Speicherung dieses Arrays werden $\mathcal{O}(\sigma \log(n))$ Bits benötigt.

Mithilfe der Arrays Ψ und C kann in T durch den *Backwards Search*-Algorithmus von Sadakane die Anzahl der Vorkommen eines Pattern im Text, also das Ergebnis einer CQ, bestimmt werden (vgl. [10]). Das Ergebnis des Algorithmus ist ein Bereich $R(P)$ von Ψ bzw. SA, in dem die entsprechenden Suffixe mit dem gesuchten Pattern $P = p_0 \dots p_{m-1}$ beginnen. Dieser Bereich wird rekursiv berechnet, indem zunächst $R(p_{m-1}) = [l_{m-1}, r_{m-1}]$ bestimmt wird. Nach Konstruktion von C ist $R(p_i) = [C[p_i], C[p_i + 1] - 1]$, dabei ist $C[p_i]$ der Index des lexikographisch kleinsten Suffix, das mit p_i beginnt und $C[p_i + 1] - 1$ der Index des lexikographisch größten Suffix, das mit p_i beginnt.[1]

Nach der Bestimmung von $R(p_{m-1})$ wird $R(p_{m-2}p_{m-1}) = [l_{m-2}, r_{m-2}]$ berechnet. Da der String $p_{m-2}p_{m-1}$ mit dem Zeichen p_{m-2} beginnt, muss $R(p_{m-2}p_{m-1})$ in $R(p_{m-2})$ enthalten sein. Zusätzlich müssen die den Indices aus $R(p_{m-2}p_{m-1})$ zugehörigen Suffixe mit p_{m-1} fortgesetzt werden. Demnach liegt eine Stelle $j \in R(p_{m-2})$ genau dann in $R(p_{m-2}p_{m-1})$, wenn das nachfolgende Suffix in $R(p_{m-1})$ liegt, also $\Psi[j] \in R(p_{m-1})$ ist. Die Grenzen des gesuchten Bereichs $R(p_0 \dots p_{m-1}) = [l_0, r_0]$ werden also rekursiv durch

$$[l_i, r_i] = \begin{cases} [C[p_i], C[p_i + 1] - 1] & \text{wenn } i = m - 1 \\ [\operatorname{argmin}_{j \in R(p_i)} \Psi[j] \geq l_{i+1}, \operatorname{argmax}_{j \in R(p_i)} \Psi[j] \leq r_{i+1}] & \text{sonst} \end{cases}$$

bestimmt (vgl. [10, Lemma 4.1]). Das Ergebnis der CQ ist dann $r_0 - l_0 + 1$. Der Algorithmus wird im folgenden Beispiel 2 illustriert.

▶ **Beispiel 2.** Für den Text $T =$ „banana\$“ ist das Array C gegeben durch $C[\$] = 0, C[a] = 1, C[b] = 4, C[n] = 5, C[\text{EOF}] = 7$. Wird nach dem Pattern $P =$ „ana“ gesucht, ist der erste zu bestimmende Bereich $R(a) = [l_2, r_2] = [C[a], C[b] - 1] = [1, 3]$. In der zweiten Iteration müssen die Grenzen l_1 und r_1 von $R(na)$ innerhalb von $R(n) = [C[n], C[\text{EOF} - 1] = [5, 6]$ bestimmt werden. Da sowohl $\Psi[5] = 1 \geq l_2$ als auch $\Psi[6] = 2 \leq r_2$ gilt, ist $R(na) = [l_1, r_1] = [5, 6]$. Im letzten Schritt wird $R(ana)$ nach dem gleichen Schema bestimmt, nach dem auch $R(na)$ bestimmt werden konnte. Es ist $R(p_0) = R(a) = [1, 3]$. Da $\Psi[1] = 0 \ngeq l_1$, aber $\Psi[2] = 5 \geq l_1$ ist, ist die untere Grenze des gesuchten Bereichs $l_0 = 5$. Die obere Grenze ist $r_0 = 6$, da $\Psi[3] = 6 \leq r_1$ ist. Das Pattern „ana“ kommt also zweimal im Text vor.

Da die Werte innerhalb von $R(p_i)$ aufsteigend sind, können Minimum und Maximum durch zwei binäre Suchen innerhalb von $R(p_i)$ ermittelt werden. Die dazu benötigte Zeit liegt in $\mathcal{O}(log(n_{p_i}))$, da es n_{p_i} Suffixe mit dem Anfangsbuchstaben p_i gibt. Da $R(p_i \dots p_{m-1})$ für alle $i \in \{0, \dots, m-1\}$ bestimmt werden muss, müssen beide Suchen jeweils m mal durchgeführt werden. Damit hängt die Zeit für den *Backward Search*-Algorithmus hauptsächlich von der Länge des Patterns und den Werten n_{p_i} für die Zeichen $p_i \in P$ ab. Die Größe des Alphabets beeinflusst lediglich die Größe des Arrays C, ist also für die Suchzeit nicht relevant. Aus diesem Grund scheint die Wahl des CSA als Basis für einen Textindex für große Alphabete sinnvoll.

[1] $p_i + 1$ bezeichnet dabei das lexikographisch auf p_i folgende Zeichen $c \in \Sigma$, wobei *EOF* das größte und \$ das kleinste aller Zeichen ist

2.3 Speicherung des CSA-Index mit Uniform Elias-Fano Codierung

Der wichtigste Faktor bei der Speicherung des CSA Index ist das *Successor Array* Ψ. In diesem Abschnitt wird erläutert, wie Ψ mit Hilfe der *Uniform Elias-Fano Codierung* (UEFC) [8] codiert werden kann, auf der auch die Speicherung des CSA++ Index basiert.

Mit der UEFC können Sequenzen aufsteigender Integer codiert werden. Zur Codierung werden die Segmente aus Ψ, die je einem Zeichen $c \in \Sigma$ zugeordnet werden können, als einzelne Sequenzen betrachtet. Eine Sequenz $S[0, m-1]$ über einem Universum $U = \{0, 1, 2, \ldots, S[m-1]\}$ wird in $c = \lceil m/b \rceil$ Blöcke $b_0, \ldots, b_{c-1}$ mit je b Integern aufgeteilt, die getrennt behandelt und durch einen von drei Blocktypen codiert werden. Die Speicherung erfolgt auf zwei Ebenen: Die erste Ebene $L1$ enthält das erste Element jeden Blocks $L1 = [S[0], S[b], \ldots, S[(c-1) \cdot b]]$, die zweite Ebene enthält die restlichen Werte.[2]

Vor der Codierung eines Blocks wird das Universum angepasst um den Speicherplatzbedarf zu reduzieren. Da Ψ keinen Wert doppelt enthält und die Sequenz S aufsteigend ist, muss für jedes Element $S[i]$ mit $i > 0$ in Block b_j gelten: $L1[j] + 1 \leq S[i] < L1[j+1]$. Wenn von jedem Element $S[i]$ mit $i > 0$ aus Block b_j der Wert des kleinstmöglichen Elements $L1[j] + 1$ subtrahiert wird, kann die Größe des Universums U_j für den entsprechenden Block auf $|U_j| = L1[j+1] - L1[j] - 1$ beschränkt werden. Zur Codierung der zweiten Ebene des Blocks wird also aus S eine Sequenz $S' = [S[1] - L1[j] + 1, \ldots, S[m-1] - L1[j] + 1]$ mit $|S'| = b - 1$ konstruiert. Die Sequenz S' wird dann, abhängig vom Verhältnis von $|U_j|$ zur Blocklänge b, als NIL-, BV- oder EF-Block codiert:

NIL-Block Enthält ein Block genau b aufeinanderfolgende Integer ($|U_j| = |S'|$), so ist die Sequenz durch $L1[j]$ und die feste Blocklänge b ausreichend bestimmt.

EF-Block Wenn $|S'| \leq |U_j|/4$ ist wird der Block mithilfe des *Elias-Fano Codes* (EFC) [1] durch zwei Bitvektoren H und L repräsentiert. Dazu wird jeder Integer $s_i \in S'$ zunächst in seine Binärrepräsentation mit $\lceil \log |U_j| \rceil$ Bits umgewandelt. Anschließend werden die Binärzahlen in $l = \lfloor \log |U_j|/|S'| \rfloor$ niederwertigen Bits (l_i) und $h = \lceil \log |U_j| \rceil - l$ höherwertige Bits (h_i) getrennt. Alle Integer, deren höherwertige Bits gleich sind, werden einer Gruppe zugeordnet. Dadurch entsteht eine Aufteilung der Integer aus S' in 2^h Gruppen G_{h_i}.

Bitvektor H enthält die Anzahl der Elemente $|G_{h_i}|$ jeder Gruppe G_{h_i}, aufsteigend sortiert nach h_i. Dazu wird jede Gruppengröße durch $|G_{h_i}|$ Einsen und eine abschließende Null repräsentiert. Die l niederwertigen Bits jedes Integers werden nacheinander in Bitvektor $L = l_0, \ldots, l_{m-1}$ gespeichert.

▶ **Beispiel 3.** Die Integer-Sequenz $S' = [3, 5, 6, 15]$ über einem Universum $U = \{0, 1, \ldots, 15\}$ mit $|U| = 16$ wird durch einen EF-Block codiert, da $|S'| = 4 \leq 16/4$ ist. Die Binärrepräsentationen der Integer werden wie folgt aufgeteilt:

0	0	0	1	} Höherwertige Bits
0	1	1	1	
1	0	1	1	} Niederwertige Bits
1	1	0	1	

$x =$	0	1	2	3
h_x	00	01	01	11
l_x	11	01	10	11

Die Gruppengrößen der vier Gruppen zu S' betragen $|G_{00}| = 1, |G_{01}| = 2, |G_{10}| = 0$ und $|G_{11}| = 1$, daher ist Bitvektor $H =$ 10 110 0 10. [3] Die niederwertigen Bits werden durch $L = 11\,01\,10\,11$ repräsentiert.

[2] In der Originalarbeit [8] wird der letzte Wert jeden Blocks gespeichert, das Prinzip ist aber analog.
[3] Die Umrandungen dienen lediglich der Visualisierung und sind nicht Teil der Codierung.

BV-Block In einem BV-Block wird eine Sequenz von Integern durch ihren *charakteristischen Vektor*, einen Bitvektor bv der Länge $|U_j|$ mit $bv[e] = 1 \Leftrightarrow e \in b_j$ codiert. Da alle Elemente im Intervall $[0, |U_j| - 1]$ liegen und nur einmal vorkommen, kann mit bv und somit $|U_j|$ Bits der gesamte Block dargestellt werden. Ein Block wird als BV-Block codiert, wenn $|S'| > {}^{|U_j|}/_4$ ist und somit die Elias-Fano Repräsentation mehr als $|U_j|$ Bits benötigen würde.

▶ **Beispiel 4.** Für die Codierung von $S' = [0, 2, 3, 16]$ über dem Universum $U = \{0, 1, \ldots, 16\}$ mit $|U| = 17$ wird ein BV-Block verwendet, da $b - 1 = 5 - 1 = 4 > {}^{17}/_4$ ist. Somit wird S' durch $bv = 10110000000000001$ repräsentiert.

Insgesamt wird ein Ψ-Segment also durch $L1$ und die einzelnen Codierungen jedes Blocks b_j repräsentiert. Für eine spätere Decodierung muss zusätzlich für jeden Block durch ein *flag* kenntlich gemacht werden, welcher Blocktyp zur Codierung verwendet wurde. Da $L1$ ebenfalls eine aufsteigende Sequenz von verschiedenen Zahlen ist, kann auch hier der EFC zur Codierung verwendet werden. Im folgenden Beispiel werden alle benötigten Komponenten zur Codierung eines Ψ-Segment bestimmt.

▶ **Beispiel 5.** Sei $S = [1, 2, 4, 5, 18, 20, 24, 26, 27, 36, 38, 39, 40, 41, 42]$ die zu codierende Integer-Sequenz. Die Blockgröße ist durch $b = 5$ gegeben. Die zu codierenden Blocks sind $b_0 = [1, 2, 4, 5, 18], b_1 = [20, 24, 26, 27, 36]$ und $b_2 = [38, 39, 40, 41, 42]$ und somit ist $L1 = [1, 20, 38]$. $L1$ wird wie ein EF-Block mit Hilfe des EFC codiert. [4] Für Block b_0 wird mit $L1[0]+1 = 1+1 = 2$ die zu codierende Sequenz $S' = [0, 2, 3, 16]$ gebildet. S' entspricht der Sequenz aus Beispiel 4 und wird somit durch $bv = 10110000000000001$ als BV-Block repräsentiert. Für Block b_1 ist $S' = [3, 5, 6, 15]$ und wird als EF-Block durch $H = 10110010$ und $L = 11011011$ codiert (siehe Beispiel 3). Der letzte Block b_2 ist eine Sequenz aufeinanderfolgender Integer (SaI) der Länge b und wird somit als NIL-Block gespeichert.

Für die Suche nach einem Pattern ist es nötig auf einzelne Elemente des codierten Segments zuzugreifen. Ein Vorteil der UEFC ist, dass für einen Zugriff auf ein Element an Stelle i der codierten Sequenz nicht alle vorhergehenden Elemente eines Blocks decodiert werden müssen. Um $S[i]$ zu bestimmen, muss zunächst der entsprechende Block $b_j = b_{\lfloor i/b \rfloor}$ und die genaue Stelle $k = i \mod b$ innerhalb des Blocks identifiziert werden, an der $S[i]$ codiert gespeichert ist. Ist $k = 0$, so ist $S[i]$ das erste Element des Blocks b_j und somit in $L1$ gespeichert. Zur Decodierung von $S[i]$ muss also lediglich $L1[j]$ decodiert werden. Andernfalls wird zur Bestimmung von $S[i]$ zunächst der lokalisierte Wert aus der zweiten Ebene des Blocks b_j $S'[k-1]$ dem angegebenen Blocktypen entsprechend decodiert. Anschließend wird $L1[j]$ decodiert und $L1[j] + 1$ zu $S'[k-1]$ addiert um das ursprüngliche Universum wieder herzustellen.

Ist b_j ein EF-Block, so kann jede Stelle der zugehörigen Sequenz S' aus L und H mit Hilfe der $Select_1$ Operation rekonstruiert werden. Für einen Index p gibt $\text{Select}_1(p)$ die Position der $p+1$-ten Eins im Bitvektor zurück. Die Bits h_{k-1} entsprechen dann der Binärcodierung von $Select_1(k-1) - (k-1)$. Wird H in einer entsprechenden Datenstruktur gespeichert, die den Speicherplatzverbrauch minimal erhöht, kann diese Operation in konstanter Zeit durchgeführt werden (vgl. [5]). Die niederwertigen Bits von $S'[k-1]$ sind im zugehörigen Array L an den Stellen $(k-1)l, \ldots, kl-1$ enthalten.

Die Decodierung einer Stelle eines NIL-Blocks ist trivial. Bei dem entsprechenden *flag* wird $S'[k-1]$ des Blocks b_j zu $k-1$ decodiert. Im Fall eines BV-Blocks muss der charakteristische Vektor decodiert werden. Um den Wert an Stelle $k-1$ zu erhalten wird bestimmt an welcher

[4] Anstelle der Blocklänge wird für die Trennung der Bits die Länge von $L1$, also c, genutzt.

Stelle k' des Vektors bv die k. Eins steht. Diese Stelle kann mit der Select_1 Funktion bestimmt werden. Der gesuchte Wert $S'[k-1]$ ist also $k' = \text{Select}_1(k-1)$. In [12] wird beschrieben, wie die Select Funktion effizient in einem charakteristischen Vektor realisiert werden kann.

3 CSA++

Die CSA++ Codierung, die Gog, Moffat und Petri in [4] vorgestellt haben, basiert auf der in Abschnitt 2.2 vorgestellten Idee des CSA mit einem *Successor Array* und der Speicherung von Ψ mit Hilfe der UEFC. Für die Speicherung von Ψ wird das Array in die den verschiedenen Zeichen zugehörigen Segmente zerlegt, die separat behandelt werden. Um das für die Suche relevante Segment identifizieren zu können, wird ein invertierter Index angelegt. Dieser *Segment Index* enthält eine Zuordnung von Zeichen zu den entsprechenden Segmenten. Ebenfalls erforderlich für die Suche ist das in Abschnitt 2.2 beschriebene Array C.

Da CSA++ speziell für Texte mit großen Alphabeten konstruiert wurde, haben Zeichen, die selten im Text vorkommen, einen großen Anteil am Alphabet. Im CSA++ Index werden die Ψ-Segmente solcher Zeichen in einer separaten Struktur gespeichert (siehe Abschnitt 3.2). Die restlichen Segmente werden in einer UEF-Struktur mithilfe der erweiterten Version der UEFC aus Abschnitt 3.1 gespeichert.

3.1 Erweiterung der Uniform Elias-Fano Codierung

In einer UEF-Struktur wird jedes Ψ-Segment durch die erweiterte UEFC codiert. Wie in Abschnitt 2.3 beschrieben werden die ersten Elemente der einzelnen Blöcke separat gespeichert und mit dem EFC codiert. Dieses Array wird in [4] und im Folgenden als *Sample Index* bezeichnet. Wie bei der UEFC können die einzelnen Blöcke dann als NIL-, BV- oder EF-Block codiert werden. Zusätzlich zu diesen Blocktypen gibt es bei CSA++ die Möglichkeit einen Block als *Run-Length Encoded Block* (RL Block) zu repräsentieren. Dieser Blocktyp dient der effizienten Darstellung von Sequenzen über einem großen Universum, indem Sequenzen aufeinanderfolgender Integer (SaI) und die Abstände zwischen diesen codiert werden. Ist das Alphabet eines Textes groß, gibt es Zeichen, für die ein großer Anteil des zugehörigen Ψ-Segments in einem Block gespeichert wird. Da die Werte eines Ψ-Segments in $[0, \ldots, n-1]$ liegen können, kann das Universum der Werte innerhalb eines solchen Blocks groß sein.

Um einen RL-Block aus einer zu codierenden Sequenz $b_j = S$ zu bilden wird die Sequenz zunächst in Subsequenzen S_i unterteilt, sodass S_i inklusionsmaximale SaI sind. Jede SaI S_i der Länge n_i wird durch höchstens drei Werte $d, 1, n_i - 1$ repräsentiert. Der Wert d gibt die Differenz von $S_i[0]$ und dem letzten Wert der vorherigen Subsequenz S_{i-1} an. Ist $S_i = S_0$, also die erste Subsequenz von S', so ist d überflüssig, da der erste Wert der Subsequenz $S_0[0] = S[0]$ ist und im *Sample Index* gespeichert ist. Durch das Tupel $(1, n_i - 1)$ wird angegeben, dass die Subsequenz S_i die Länge n_i hat. Für einzelne Werte, also Subsequenzen S_i der Länge $n_i = 1$, wird lediglich d gespeichert. Da in diesem Fall $d \neq 1$ ist, ist die Codierung dennoch eindeutig. Zusammen mit $S[0]$ kann die ursprüngliche Sequenz iterativ rekonstruiert werden. Die einzelnen Integer der so entstehenden Sequenz werden mithilfe des präfixfreien *Elias δ Code*s aus [2] codiert.

▶ **Beispiel 6.** Sei $S = [3, 4, 5, 10, 11, 13, 14, 15, 17]$ die zu codierende Integer Sequenz mit $S_0 = [3, 4, 5]$, $S_1 = [10, 11]$, $S_2 = [13, 14, 15]$ und $S_3 = [17]$. Mithilfe der RL-Block Codierung entsteht aus S die Integer-Sequenz $RL =$ [1,2,5,1,1,2,1,2,2]. Wenn die einzelnen Zahlen aus RL mit Hilfe des *Elias δ Code*s codiert werden, entsteht der Bitstring 1 0100 0110 1 1 1

0100 1 0100 0100.[5]

Je länger die zu codierenden Subsequenzen sind, desto weniger Bits werden offensichtlich zur Codierung benötigt. Allerdings ist der Speicherplatzverbrauch durch eine RL-Block Repräsentation für Sequenzen über einem großen Universum verglichen mit dem Speicherplatzverbrauch durch einen EF- oder BV-Block relativ gering. Der Nachteil dieser Codierung besteht darin, dass die Rekonstruktion einer beliebigen Stelle der ursprünglichen Sequenz S zunächst die Berechnung aller vorheriger Stellen erfordert. Daher wird in [4] festgelegt, dass ein Block nur dann als RL-Block codiert wird, wenn die Codierung als EF- oder BV-Block mehr als doppelt so viel Speicherplatz erfordern würde.

3.2 Speicherstruktur für Segmente seltener Symbole

Die Struktur von CSA++ sieht vor, dass für jedes Zeichen aus $\Sigma \cup \{\$\}$ eine separate Struktur aus codiertem Ψ-Segment und *Sample Index* gebildet wird. Im schlimmsten Fall sind die im entsprechenden Ψ-Segment enthaltenen Werte sehr weit verteilt, das zur Codierung benötigte Universum also groß. In diesem Fall kann das Segment durch keinen Blocktypen der erweiterten UEFC effizient dargestellt werden. Zusätzlich wird für jede UEF-Struktur, unabhängig von der Länge des Segments, eine konstante Anzahl Bits benötigt. Für Zeichen, die nur wenige Ψ-Einträge haben, ist dieser Speicherplatz im Vergleich zur Anzahl der codierten Ψ-Einträge relativ groß und der Speicherplatzverbrauch pro Eintrag steigt.

Bei einem großen Alphabet ist es deutlich wahrscheinlicher, dass ein Zeichen selten vorkommt, als bei einem kleinen Alphabet. In [4] wird eine separate Struktur für die Speicherung seltener Symbole vorgestellt. Seltene Symbole sind die Zeichen c des Alphabets, für die n_c unter einem gegebenen Grenzwert L liegt: $\Sigma' = \{c | n_c \leq L\}$. Alle Werte aus Ψ, die innerhalb eines Segments von $c' \in \Sigma'$ liegen, werden nicht als UEF-Struktur mit *Sample Index* sondern in L Arrays $A_1, \ldots, A_L$ gespeichert. Kommt ein seltenes Symbol n_c mal im Text vor, so gibt es auch genau n_c Ψ-Einträge im entsprechenden Segment. Diese werden in unveränderter Reihenfolge in A_{n_c} gespeichert. Zur Konstruktion der Arrays $A_1, \ldots, A_L$ werden die Symbole $c' \in \Sigma'$ in lexikographischer Reihenfolge abgearbeitet und die Binärrepräsentationen der Werte des entsprechenden Segments in Ψ werden an das Array $A_{n'_c}$ angehängt.

▶ **Beispiel 7.** Sei $\Sigma' = \{\$, b, n\}$ mit $n_b = 1, n_n = 2$. Das *Successor Array* entspricht dem Array Ψ aus Beispiel 1. Die relevanten Segmente sind also $\Psi[0]$, $\Psi[4]$ und $\Psi[5, 6]$. Die Arrays A_1 und A_2 sind dann:

$$A_1 = [\Psi[0], \Psi[4]] = [100, 011] \text{ und } A_2 = [\Psi[5], \Psi[6]] = [001, 010]$$

Um eine Stelle von Ψ zu bestimmen, muss bei der Decodierung ermittelbar sein, ob das Segment eines Zeichens als UEF-Struktur oder separat in einem A_i gespeichert ist. Dazu wird dem Index ein Bitvektor D der Länge $|D| = \sigma + 1$ mit $c \in \Sigma' \Leftrightarrow D[c] = 0$ hinzugefügt. Wird nach einem Zeichen c gesucht und ist $D[c] = 1$, so wird der Segment Index verwendet, um das entsprechende Segment zu lokalisieren. Die darin enthaltene UEF-Struktur kann dann wie beschrieben verwendet werden. Ist dagegen $D[c] = 0$, so wird zunächst das relevante Array A_{n_c} mit Hilfe des C Arrays bestimmt. Um die Positionen der Ψ Werte in A_{n_c} zu bestimmen, wird ermittelt wie viele lexikographisch kleinere Symbole ebenfalls n_c mal im Text vorkommen und somit in A_{n_c} vor den gesuchten Werten stehen. Mit einer geeigneten Datenstruktur kann diese Anzahl in konstanter Zeit ermittelt werden.

[5] Die Umrandungen zeigen welche Zahlen zu einer Subsequenz gehören, sind aber nicht Teil der Codierung.

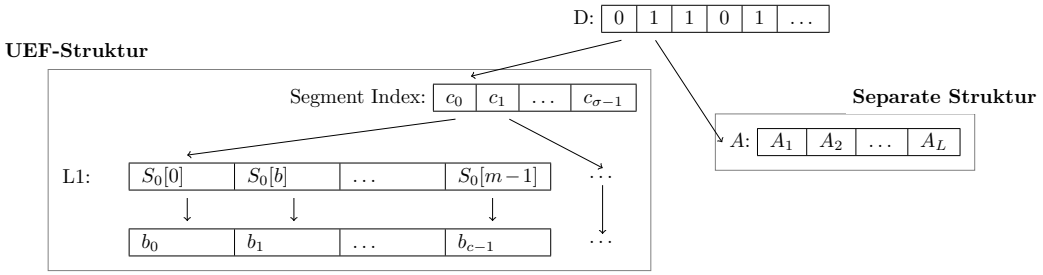


Abbildung 1 Struktur der Codierung von Ψ im CSA++ Index.

Der benötigte Speicherplatz pro Ψ-Eintrag ist in $\mathcal{O}(\log(n))$, da keine Reduktion des Universums vorgenommen wird. Es ist jedoch schwierig diesen Wert mit dem Speicherplatzverbrauch einer UEFC des gleichen Segments zu vergleichen, da die Länge dieser Codierung vom block-spezifischen Faktor u_j und dem gewählten Blocktypen abhängt. Die Verwendung der speziellen Struktur für ausgewählte Zeichen hat jedoch den Vorteil, dass der Speicher *Overhead* verhältnismäßig gering ist. Wenn eine UEF-Struktur angelegt wird, wird eine Erweiterung des Segment Index ebenso nötig wie ein neues Array $L1$ als *Sample Index* und mindestens eine Codierung eines Blocks. Wird dagegen die separate Struktur genutzt, wird lediglich eines der vorhandenen Arrays $A_1, \ldots, A_L$ erweitert. Der Speicherplatz, der unabhängig von der Anzahl der Ψ-Einträge im betrachteten Segment benötigt wird, ist also in der UEF-Struktur größer. Zusätzlich dazu erfordert die Decodierung der Ψ-Werte in A lediglich die Bestimmung der richtigen Positionen des relevanten Arrays, da die Werte in ihrer Binärrepräsentation gespeichert werden. Es muss also abgewägt werden, ab wie vielen Einträgen innerhalb eines Segments eine mögliche Speicherplatzreduktion durch Verwendung der UEF-Struktur die Vorteile der schnelleren Suche überwiegt. In [4] wird ein Schwellwert von $L = b$ verwendet, sodass für ein Zeichen nur dann eine UEF-Struktur erstellt wird, wenn die entsprechenden Ψ-Einträge mindestens einen vollen Block einnehmen.

Im folgenden Abschnitt wird erläutert, wie das Ergebnis einer CQ im CSA++ Index bestimmt werden kann.

3.3 Pattern Suche mit CSA++

Die Struktur des CSA++ Index unterscheidet sich deutlich von der eines CSA. In Abbildung 1 werden die verschiedenen Ebenen der Struktur dargestellt: Die oberste Ebene enthält das Array D, in der nachfolgenden Ebene wird zwischen der UEF-Struktur und der separaten Speicherung für seltene Zeichen unterschieden. In der UEF-Struktur auf der linken Seite von Abbildung 1 wird durch den Segment Index auf die *Sample Indexes* $L1$ und die zugehörigen Blöcke b_i der verschiedenen Ψ-Segmente verwiesen. Die Struktur für seltene Zeichen auf der rechten Seite der Abbildung enthält die Arrays $A_1, \ldots, A_L$.

Die veränderte Struktur erfordert eine angepasste Vorgehensweise bei der Suche. Das Ergebnis einer CQ kann weiterhin mit dem *Backward Search*-Algorithmus berechnet werden. Es müssen also die Bereiche $R(p_{m-1}), R(p_{m-2}p_{m-1}), \ldots R(p_0 \ldots p_{m-1})$ bestimmt werden. Anders als bei dem in Abschnitt 2.2 vorgestellten Algorithmus werden jedoch nur die unteren Grenzen l_i über eine binäre Suche berechnet. Die oberen Grenzen r_i werden in jedem Schritt iterativ bestimmt.

Um die untere Grenze eines Bereichs zu berechnen, muss bestimmt werden welche Indices von Ψ im p_i-Segment liegen. Dazu kann weiterhin das C Array verwendet werden. Anschließend muss der kleinste Index j in diesem Bereich bestimmt werden, der $\Psi[j] =$

$w \geq l_{i+1}$ erfüllt (siehe Abschnitt 2.2). Da $\Psi[j]$ im p_i-Segment liegt, muss nach der Struktur des Index zunächst bestimmt werden, ob das benötigte Ψ-Segment für das Zeichen p_i als UEF-Struktur oder separat gespeichert wurde. Ist $D[p_i] = 0$, so werden über C das benötigte Array A_x und dessen relevanten Stellen bestimmt (siehe Abschnitt 3.2). Da die Werte in den Arrays A_i in ihrer Binärrepräsentation gespeichert sind, ist keine weitere Decodierung nötig und der Wert w kann mittels Binärsuche gefunden werden.

Ist $D[p_i] = 1$, so wird über den Segment Index bestimmt, wo das benötigte Ψ-Segment von Symbol p_i gespeichert ist. Da ein w mit $w \geq l_{i+1}$ gesucht wird, kann zunächst über eine Binärsuche in $L1$ der Block b_j bestimmt werden, der die kleinsten Werte enthält, die größer als l_{i+1} sind. Es gilt also $L_1[j-1] \leq l_{i+1}$ und $L1[j] \geq l_{i+1}$. Die einzelnen Werte in $L1$ können wie in Abschnitt 2.3 beschrieben einzeln decodiert werden. Innerhalb des relevanten Blocks wird dann der kleinste Wert $w \geq l_{i+1}$ gesucht. Ist dieser Block als RL-Block codiert, so muss der gesamte Block decodiert werden, bevor der gesuchte Wert gefunden werden kann. Andernfalls wird erneut durch eine Binärsuche nach w gesucht, wobei nur die benötigten Werte decodiert werden.

Bisher wurde lediglich die Suche nach $\Psi[j] = w$ beschrieben, für die Bestimmung des Bereichs $R(p_i \dots p_{m-1})$ wird jedoch der Index j benötigt. Dieser Index kann bestimmt werden, indem bei der beschriebenen Suche nach w stets der „aktuelle“ Index mitgeführt wird.

Wenn Index l_i gefunden wurde, kann davon ausgehend mit Hilfe eines *Finger Search* auch das Ende des gesuchten Bereichs r_i gefunden werden. Iterativ wird für Indices $l_i \leq k \leq r_{i+1}$ geprüft, ob $k \in R(p_i)$ ist. Wird kein solcher Index gefunden, so kommt das gesuchte Pattern nicht im Text vor. Bei dieser linearen Suche müssen maximal $r_i - l_i + 1$ aufeinanderfolgende Stellen von Ψ überprüft werden. Für CSA++ wird diese Form der Suche bevorzugt, da die Suchzeit im Vergleich zur binären Methode, analog zur Bestimmung von l_i, von einer geringen Anzahl von Patternvorkommen stärker profitiert. Bei der binären Suche muss dagegen unabhängig von der Größe des Bereichs zunächst der relevante Block bzw. das relevante Array bestimmt werden. Je größer das Alphabet eines Textes im Verhältnis zu dessen Länge ist, desto seltener kommt ein Zeichen durchschnittlich im Text vor. Für die Vorkommen eines Pattern, als Kombination mehrerer Zeichen, verstärkt sich dieser Effekt. Die erwartete Anzahl der Vorkommen eines Pattern $r_i - l_i + 1$ und damit die erwartete Anzahl der nötigen Iterationen des *Finger Search* ist also für den Anwendungsfall von CSA++ gering.

Soll die Suche nicht nur die Vorkommen eines Pattern zählen sondern auch deren Position im Text bestimmen, so muss man auf diese vom bestimmten Index in Ψ schließen können. Eine gängige Methode um solche sogenannte *Locating Queries* zu beantworten basiert auf der teilweisen Speicherung des SA. Mit Hilfe dieser SA Einträge und Ψ können dann Textpositionen beliebiger Suffixe bestimmt werden (vgl. [3]).

4 Zusammenfassung & Fazit

In dieser Arbeit wurde die grundlegende Funktionsweise eines CSA und des *Backward Search*-Algorithmus erläutert und beschrieben, wie das *Successor Array* mithilfe der UEFC codiert werden kann. Darauf aufbauend wurde der CSA++ Index vorgestellt.

In [4] werden Textindices hinsichtlich der Indexgröße relativ zur Textgröße und der Zeit zur Beantwortung einer CQ pro Patternzeichen verglichen. Für die getesteten Texte mit großen Alphabeten erzielt CSA++ die geringste Indexgröße bei schnellster Suche im

Vergleich zu verschiedenen CSA Implementierungen und FM-Index Varianten[6]. Auch bei kleinen Alphabeten ist CSA++ den betrachteten CSA Implementierungen überlegen, was darauf schließen lässt, dass die Erweiterungen auch in diesem Fall vorteilhaft sein können.

Die Analyse eines der konstruierten Indices zeigt wie häufig die Erweiterungen des CSA++ genutzt werden: Für mehr als 50% des Alphabets wird die Struktur für seltene Symbole genutzt und damit $4,4\%$ von Ψ in Binärrepräsentation gespeichert. Der RL-Blocktyp wird für $3,8\%$ der Ψ-Einträge genutzt. Für diese Blocks wäre eine Codierung durch einen anderen Blocktyp per Konstruktion mindestens doppelt so groß, es wurde also Speicherplatz gespart. Es bleibt jedoch offen wie sich dieser Aspekt auf die Suchzeit auswirkt. Die Modifikation der Suche erscheint für viele CQs sinnvoll, da viele Zeichen selten im Text vorkommen, allerdings gibt es auch wenige Zeichen, die einen großen Teil von Ψ ausmachen. Ob die Verwendung des *Finger Search* sinnvoll ist, hängt also stark von der Struktur der Pattern ab. Für die Weiterentwicklung des Index könnte geprüft werden, ob die Suchzeit verringert werden kann indem die Verwendung des *Finger Search* in jedem Schritt vom aktuellen Zeichen und dem zu durchsuchenden Bereich abhängig gemacht wird.

Literatur

1 Peter Elias. Efficient Storage and Retrieval by Content and Address of Static Files. *Journal of the ACM*, pages 246–260, 1974.

2 Peter Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, pages 194–203, 1975.

3 Paolo Ferragina, Rodrigo González, Gonzalo Navarro, and Rossano Venturini. Compressed text indexes: From Theory to Practice. *Journal of Experimental Algorithmics*, 2009.

4 Simon Gog, Alistair Moffat, and Matthias Petri. CSA++: Fast Pattern Search for Large Alphabets. In *Proceedings of the 19th Workshop on Algorithm Engineering and Experiments*, pages 73–82, 2017.

5 Rodrigo González, Szymon Grabowski, Veli Mäkinen, and Gonzalo Navarro. Practical implementation of rank and select queries. In *Poster Proceedings Volume of the fourth Workshop on Efficient and Experimental Algorithms*, pages 27–38, 2005.

6 Roberto Grossi and Jeffrey Scott Vitter. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. *SIAM Journal on Computing*, pages 378–407, 2005.

7 Gonzalo Navarro and Veli Mäkinen. Compressed Full-Text Indexes. *ACM Computing Surveys*, 2007.

8 Giuseppe Ottaviano and Rossano Venturini. Partitioned Elias-Fano indexes. In *Proceedings of the 37th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 273–282, 2014.

9 Giulio Ermanno Pibiri and Rossano Venturini. Clustered Elias-Fano Indexes. *ACM Transactions on Information Systems*, pages 2:1–2:33, 2017.

10 Kunihiko Sadakane. Succinct Representations of lcp Information and Improvements in the Compressed Suffix Arrays. In *Proceedings of the 13th annual ACM-SIAM Symposium on Discrete Algorithms*, pages 225–232, 2002.

11 Kunihiko Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, pages 294–313, 2003.

12 Sebastiano Vigna. Quasi-Succinct Indices. In *Proceedings of the sixth ACM International Conference on Web Search and Data Mining*, pages 83–92, 2013.

[6] Der FM-Index ist ein gängiger Textindex, Information zu Konstruktion und CQ-Algorithmen in [7].

Effiziente und speicherplatzsparende Implementierung von Arrays dynamischer Größe*

Tim Tannert, 168292[1]

1 **Informatik TU-Dortmund, OH12, Dortmund, Deutschland**
`tim.tannert@tu-dortmund.de`

Abstract

Diese Ausarbeitung betrachtet eine optimale Datenstruktur für Arrays dynamischer Größe aus [1]. Diese Arrays können eine beliebige Anzahl an Elementen fester Größe beinhalten und benötigen dabei wenig Speicherplatz. Ein solches Array hat die Form $A[0..n-1]$ und kann dabei am Ende wachsen oder schrumpfen. Alle Operationen der betrachteten Datenstruktur benötigen $O(1)$ Zeit und $O(\sqrt{n})$ Extraspeicherplatz. Dabei werden die Laufzeitschranken der Operationen und die Extraspeicherplatzschranke bewiesen. Zusätzlich wird gezeigt, dass jede beliebige Datenstruktur mit dynamischer Größe $\Omega(\sqrt{n})$ Extraspeicherplatz benötigt und die betrachtete Datenstruktur somit im Hinblick auf ihren Extraspeicherplatz optimal ist. Ein besonderer Fokus liegt dabei auf dem Aufbau der Datenstruktur, der besonders wichtig für die Realisierung der Operationen in konstanter Laufzeit ist.

Keywords and phrases Optimal Resizeable Arrays – Optimal Dynamic Arrays – Advanced Data Structures

Digital Object Identifier 10.4230/LIPIcs.BADS.2017.1

1 Problemstellung

Dynamische Arrays sind in vielen Programmiersprachen bereits in der Standardbibliothek enthalten. Sie können für die Implementierung einer Vielzahl von Datenstrukturen verwendet werden und bieten viele Vorteile im Vergleich zu statischen Arrays. Aus diesem Aspekt ist eine optimale Implementierung wünschenswert.

Ein dynamisches Array ist dabei ein Array, das im Gegensatz zu einem statischen Array zur Laufzeit (theoretisch) beliebig viele Elemente aufnehmen kann. Zum Initialisierungszeitpunkt muss dabei nicht feststehen, wie viele Elemente bzw. Daten im Array gespeichert werden sollen. Ein dynamisches Array kann am Ende wachsen und schrumpfen. Dabei hat die Datenstruktur zu jedem Zeitpunkt die Form $A[0..(n-1)]$ (wobei n nicht fest ist) und besitzt folgende Operationen:

1. **Lese**(i): Gibt das Element mit Index i aus. Es gilt $0 \leq i \leq n-1$.
2. **Schreibe**(i,x): Schreibt an Index i, den Wert x. Es gilt $0 \leq i \leq n-1$.
3. **Wachse()**: Setzt $n = n+1$ und erzeugt ein neues Element mit Index $i = n-1$.
4. **Schrumpfe()**: Löscht das n-te Element und setzt $n = n-1$.

Bei späterer Laufzeitbetrachtung werden die Operationen *Lese(i)* und *Schreibe(i,x)* nicht getrennt betrachtet. Die Komplexität beider Funktionen ist auf die Lokalisierung (*Lokalisiere(i)*) der zu lesenden oder in die zu schreibenden Speicherstelle reduzierbar. Somit muss lediglich die Laufzeit dieser Funktion betrachtet werden.

* This work was partially supported by someone.

Eine weitverbreitete Implementierungsmöglichkeit von dynamischen Arrays ist die sogenannte *Doubling-Methode*. Dabei wird zunächst ein Array fester Größe erzeugt. Falls die Anzahl der Elemente im Array einen zuvor festgesetzten Grenzwert (z. B. indem es voll wird) überschreitet, dann wird im Speicher ein weiteres Array doppelter Größe angelegt und anschließend alle Werte vom alten Array in das neue kopiert. Beim Löschen verhält sich das Array analog. Dieses Verfahren ist recht intuitiv, hat allerdings einige Nachteile: Zum einen liegt die Worstcaselaufzeit nicht in $O(1)$, da in einigen Fällen das komplette Array kopiert werden muss. Zusätzlich gibt es Ausführungszeitpunkte, an denen der benötigte Extraspeicherplatz $O(n)$ entspricht.

Die im folgenden betrachtete Datenstruktur ist eine optimale Datenstruktur für dynamische Arrays. Mit optimal sind zum einen die Operationskosten in $O(1)$ und der Extraspeicherplatzverbrauch in $O(\sqrt{n})$ gemeint. In Abschnitt 2.6 wird gezeigt, dass jede dynamisch wachsende Datenstruktur mindestens $\Omega(\sqrt{n})$ Extraspeicherplatz benötigt. Das bedeutet, dass der angestrebte Extraspeicherplatz optimal ist. In den folgenden Abschnitten wird der Entwurf einer solchen Datenstruktur betrachtet. Alle folgenden Informationen über die Datenstruktur (z. B. Beweise, Lemma, Theoreme) stammen aus [1].

2 Datenstruktur

In diesem Abschnitt wird die Datenstruktur für dynamische Arrays aus [1] beschrieben. Dabei werden zuerst in Unterabschnitt 2.1 Grundideen für die Datenstruktur dargestellt, die anschließend verwendet werden, um die in Unterabschnitt 2.2 vorgestellte Datenstruktur zu entwickeln. Anschließend wird in Unterabschnitt 2.3 die Implementierung der benötigten Datenstrukturoperationen besprochen. Danach werden in Unterabschnitt 2.4 und 2.5 die Laufzeit- und Extraspeicherplatzschranke analysiert. Zum Schluss wird in 2.6 die Optimalität der Datenstruktur (bezüglich des Extraspeicherplatzes) bewiesen.

2.1 Grundidee

Eine fundamentale Eigenschaft der vorgestellten Datenstruktur ist die Optimalität sowohl in der Operationslaufzeit als auch im verwendeten Extraspeicherplatz. Im folgenden werden zwei Ansätze für die Datenstruktur betrachtet, die beide alleine nicht in Frage kommen, aber kombiniert die gewünschten Laufzeit- und Speicherplatzschranken einhalten. Wird im folgenden von Elementen gesprochen, dann sind immer Daten, die gespeichert werden sollen, gemeint. Die Größe eines Elements ist fest. Die Größe eines Datenblocks beschreibt, wie viele Elemente potenziell in diesen passen. Der entsprechende Speicherplatz ist somit reserviert, aber noch nicht belegt.

Wie bereits besprochen, soll die konstruierte Datenstruktur $o(n)$ Extraspeicherplatz verwenden, um besser als die *Doubling-Methode* zu sein. Eine Möglichkeit wäre es beispielsweise einen Extraspeicherplatz von $O(\sqrt{n})$ anzustreben. Somit ist es naheliegend, alle n Elemente eines Arrays in genau $\sqrt{n}$ Blöcken der Größe $\sqrt{n}$ zu speichern. Da n allerdings dynamisch wächst, nehmen die Speicherblöcke Größen von 1 bis $\sqrt{n}$ an. Unter diesen Umständen passen in die ersten k Speicherblöcke $n = \frac{k(k+1)}{2}$ Elemente. Daraus folgt, dass $k = |\frac{\sqrt{1+8n}-1}{2}|$ Blöcke benötigt werden um n Elemente zu speichern. Da jeder Block in der Datenstruktur h *Headerinformationen* benötigt, wobei h konstant ist, folgt daraus, dass der Extraspeicherbedarf in $\Theta(\sqrt{n})$ liegt.

Dieser Ansatz bietet bereits eine Möglichkeit, ein dynamisches Array mit $\Theta(\sqrt{n})$ Extraspeicherplatz zu realisieren. Allerdings muss hierbei während eines Zugriffs (zum Schreiben oder Lesen) auf ein Element unter anderem der Speicherblockindex des dazugehörigen

Speicherblocks berechnet werden. Alleine hierfür muss $\sqrt{1+8i}$ berechnet werden, was mit *Newtons-Methode* in $O(\log(\log(i))$ Laufzeit möglich ist [4, Seite 274-292].

Das vorherige Problem beim Berechnen des gesuchten Blockindexes ist zu umgehen, indem versucht wird Blöcke mit der Größe von Zweierpotenzen zu verwenden. Mit diesen Ansatz muss lediglich $\lfloor \log_2(1+i) \rfloor$ berechnet werden, um den Blockindex zu bestimmen. Dies ist in konstanter Laufzeit möglich. Bei diesem Ansatz gilt bezüglich der Elemente n, die in k Blöcke passen: $n = 2^k - 1$. Sind nun alle Blöcke voll und ein Element wird eingefügt, dann wird im schlimmsten Fall ein neuer Speicherblock erzeugt, der um Eins größer ist als die bisherigen Speicherblöcke zusammen. Es wird somit im Worstcase $O(n)$ Extraspeicher verwendet, was nicht wünschenswert ist.

Um eine effiziente Datenstruktur zu finden, können die obigen Ideen kombiniert werden. Die Speicherblöcke können in Superblöcke geteilt werden, die wiederum aus $2^{i/2}$ Datenblöcken bestehen, welche ungefähr $2^{i/2}$ groß sind. Interessanterweise existieren dann zu jedem Zeitpunkt nur $O(\sqrt{n})$ Datenblöcke. Der genaue Aufbau dieser Datenstruktur folgt im folgenden Abschnitt.

2.2 Aufbau

Die Datenstruktur besteht aus einem Indexblock und mehreren Datenblöcken. Der Indexblock beinhaltet Zeiger auf jeden Datenblock. Er wird benötigt, um einen konstanten Zugriff auf jeden Datenblock zu ermöglichen. Ein Datenblock beinhaltet mehrere Elemente. Zusätzlich können die Datenblöcke verschiedenen Superblöcken zugeordnet werden. Ein Superblock beschreibt dabei eine Menge von Datenblöcken. Für alle Datenblöcke innerhalb eines Superblocks gilt, dass diese die selbe Größe haben. Superblöcke werden im weiteren Verlauf als $SB_0, ..., SB_{s-1}$ und Datenblöcke als $DB_0, ..., DB_{d-1}$ bezeichnet. Die Datenstruktur beinhaltet Informationen über die Anzahl der Elemente, die Anzahl und Position der Superblöcke (des ersten Datenblocks), die Anzahl der nicht-leeren Datenblöcke und zusätzlich die Größe und die Anzahl der Elemente im letzten nicht-leeren Datenblock. Der k-te Superblock SB_k besteht aus $2^{\lfloor k/2 \rfloor}$ Datenblöcken. Ein Datenblock beinhaltet die Nutzdaten, also die Elemente des dynamischen Arrays. Wenn ein neuer Datenblock erzeugt wird, er allerdings nicht mehr einem bestimmten Superblock zugeordnet werden kann (falls zu viele Datenblöcke der selben Größe existieren), dann erhält der neue Datenblock eine größere Größe. Die Datenblöcke werden so erzeugt, dass aus dem k-ten Superblock alle Datenblöcke die Größe $2^{\lceil k/2 \rceil} + h$ (er benötigt nämlich h *Headerinformationen*) haben. Folglich kann der k-te Superblock bis zu $2^{\lceil k/2 \rceil} \cdot 2^{\lfloor k/2 \rfloor} = 2^k$ Elemente besitzen. Datenblöcke werden nur gelöscht, wenn zwei leere Datenblöcke existieren. Somit kann bis zu ein leerer Datenblock existieren. Der Schematische Aufbau der Datenstruktur wird in Figure 1 dargestellt.

▶ **Theorem 1** (Optimalität der Datenstruktur). *Die Operationen der beschriebenen Datenstruktur benötigen $O(1)$ Laufzeit und die Datenstruktur benötigt $O(\sqrt{n})$ Extraspeicherplatz. Sie ist auf jeder Maschine implementierbar, die Bitschiebeoperationen um k Bit auf Worte der Größe $\lceil \log_2(n+1) \rceil$ und das Reservieren und Freigeben von Speicher in konstanter Zeit unterstützen. Des Weiteren ist die Anzahl der Operationen zwischen Speicherbelegung und Speicherfreigabe vorhersagbar. Wird eine dieser Funktionen zum Zeitpunkt $t = n$ aufgerufen, dann wird der nächste Aufruf derselben Funktion nach $\Omega(\sqrt{t})$ Operationen geschehen.*

2.3 Datenstrukturoperationen

Im folgenden wird die Implementierung der bereits aus Abschnitt 1 bekannten Operationen *Wachse()*, *Schrumpfe()* und *Lokalisiere(i)* besprochen.

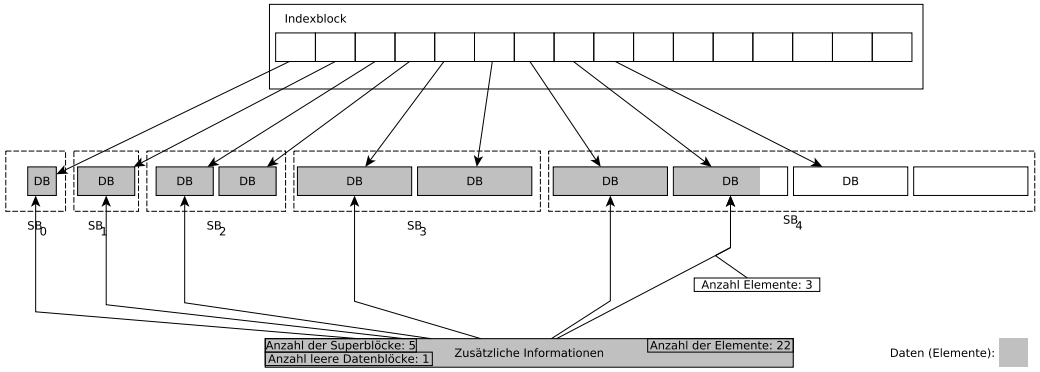


Figure 1 Ein dynamisches Array, dass mit 22 Elementen befüllt ist. Dabei waren zuvor 27 Elemente in der Datenstruktur und es wurden 5 entfernt. Jeder Superblock beinhaltet Datenblöcke einer Größe.

Wachse

Die Operation *Wachse()* ermöglicht es dem dynamischen Array ein Element mehr aufzunehmen. Folglich muss mehr Platz für ein weiteres Element geschaffen werden. Dabei ist es abhängig von der genauen Implementierung, ob der Platz für das Element vorinitialisiert wird, oder ob die neue Speicherzelle unangetastet bleibt und den unbekannten alten Wert behält.

Entweder wird der letzte Datenblock um ein Element erweitert und die Anzahl der Elemente d dieses Datenblocks wird erhöht oder der letzte Datenblock ist voll und es muss ein neuer Datenblock mit reserviert werden. Der Indexblock muss dafür einen Zeiger auf den neuen Datenblock erhalten und kann dabei voll werden. Ist der Indexblock durch eine frühere Operation voll, so kann kein weiterer Datenblock eingefügt werden. Somit wird die Größe des Indexblocks verdoppelt. Zusätzlich kann es geschehen, dass der letzte Superblock voll war und der neue Datenblock zu einem neuen Superblock gehört. In diesem Fall wird vor der Erzeugung des Datenblocks die Anzahl der Superblöcke erhöht. Anschließend wird der neue Datenblock mit der entsprechenden Größe erzeugt und das neue Element wird hinzugefügt. Zum Schluss muss die globale Variable, welche die Anzahl der Elemente im Array speichert, erhöht werden. Eine detailliertere Beschreibung ist in Algorithm 1 zu finden. Zusätzlich wird in Abbildung 2 der Zustand der Datenstruktur vor und nach dem Aufruf der Operation *Wachse()* dargestellt.

Schrumpfe

Die Anzahl der Elemente in der Datenstruktur soll bei einem Aufruf von *Schrumpfe()* um Eins verringert werden. Hierfür wird die Anzahl der Elemente im letzten nicht-leeren Datenblock verringert. Die Daten des letzten Elements werden explizit aus der Datenstruktur gelöscht. Sollte der letzte nicht-leere Datenblock $DB_{d'}$ durch diese Aktion völlig leer werden, dann wird geprüft, ob noch ein weiterer (jüngerer (in Abbildung 1 weiter rechts stehender)) Datenblock leer ist. Denn ein Datenblock wird erst gelöscht, wenn ein weiter Datenblock leer geworden ist. Durch dieses Verhalten können zu keinem Zeitpunkt mehr als Zwei leere Datenblöcke existieren. Da ein Eintrag aus dem Indexblock gelöscht wird, muss geprüft werden, ob dieser anschließend nur noch ein Viertel befüllt ist. Ist dies der Fall, muss seine Größe

Algorithm 1 Pseudocode: Wachse

```
1. Wenn der letzte nicht-leere Datenblock DB_{d-1} voll ist:
        1.1 Wenn der letzte Superblock SB_{s-1} voll ist:
                1.1.1 Erhöhe s
                1.1.2 Wenn s gerade ist, dann verdoppel
                      die Anzahl der Datenblöcke innerhalb des
                      Superblocks.
                1.1.3 Falls s ungerade ist, dann verdoppel
                      die Größe der Datenblöcke in dem neuen
                      Superblock.
                1.1.4 Setze SB_{s-1} auf frei/leer.
        1.2 Wenn kein leerer Datenblock existiert:
                1.2.1 Wenn der Indexblock voll ist, dann verdoppel
                      seine Größe
                1.2.2 Reserviere Speicher für einen neuen letzten
                      Datenblock, der zu SB_{s-1} hinzugefügt wird.
        1.3 Erhöhe d und die Anzahl der Datenblocks in SB_{s-1}.
        1.4 Setze DB_{d-1} auf leer.
2. Erhöhe n und die Anzahl der belegten Elemente in DB_{d-1}.
```

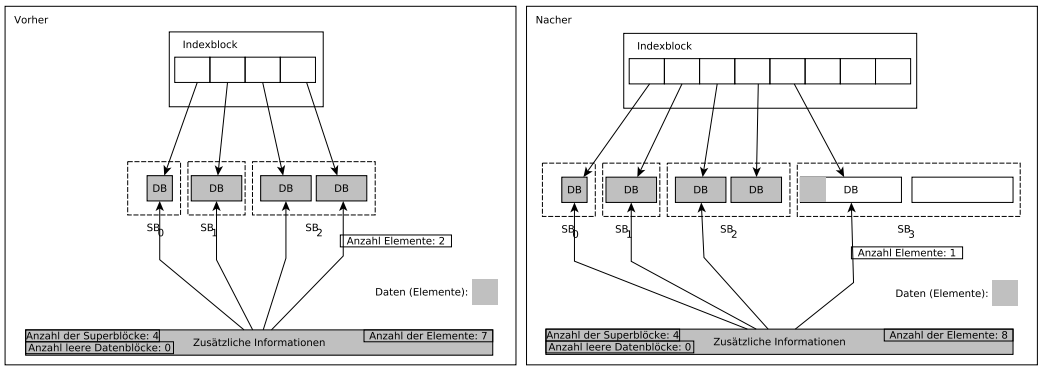


Figure 2 In dieser Abbildung ist auf der linken Seite die Datenstruktur mit 7 Elementen abgebildet. Dabei ist der letzte Datenblock und der letzte Superblock voll. Zusätzlich ist der Indexblock voll. Auf der Rechten Seite ist die Datenstruktur nach einer *Wachse()*-Operation abgebildet. Der Indexblock wurde verdoppelt, ein neuer Datenblock wurde angelegt (somit ext. auch ein neuer Superblock), die Anzahl der Superblöcke und die Anzahl der Elemente wurde erhöht und es wurde ein neuer Zeiger auf den letzten nicht-leeren Datenblock eingefügt.

Algorithm 2 Pseudocode: Schrumpfe

```
1. Verringere n und die Anzahl der Elemente des letzten nicht
   leeren Datenblocks DB_{d'}.
2. Wenn DB_{d'} leer ist:
        2.1 Wenn ein jüngerer Datenblock bereits leer ist, gebe
            diesen jüngeren Datenblock frei.
        2.2 Wenn der Indexblock ein Viertel gefüllt ist, dann
            halbiere seine Größe.
        2.3 Verringere d und die Anzahl der Datenblöcke, die sich im
            letzten Superblock SB_{s-1} befinden.
        2.4 Wenn SB_{s-1} leer ist, verringere s.
        2.5 Markiere DB_{d-1} als voll.
```

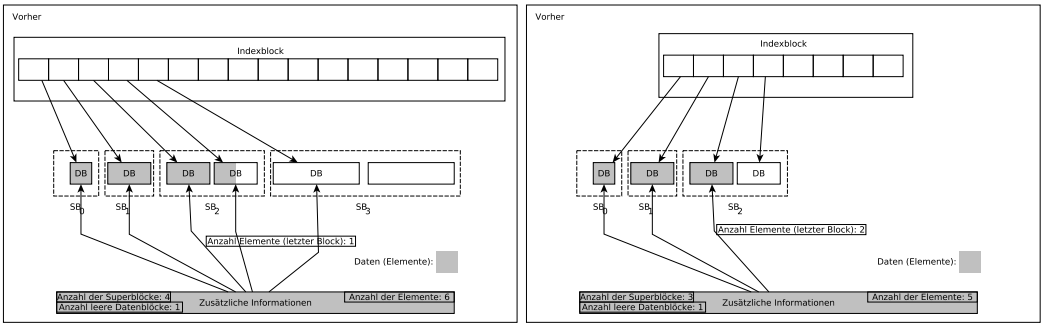


Figure 3 In dieser Abbildung ist auf der linken Seite die Datenstruktur mit 6 Elementen abgebildet. Dabei besaß das Array zu einem früherem Zeitpunkt 11 Elemente und es wurde 5-mal *Schrumpfe()* aufgerufen. Somit hat die linke Abbildung einen leeren Datenblock. Auf der rechten Seite wurde auf das Array ein weiteres Mal *Schrumpfe()* aufgerufen. Da der Indexblock nur noch zu einem Viertel gefüllt war, wurde der Indexblock verkleinert, der jüngste nicht leere Datenblock wurde entfernt (somit ext. der letzte Superblock nicht mehr), die Anzahl der Superblöcke wurde verringert, der Zeiger auf den letzten nicht-leeren Datenblock wurde angepasst, die Anzahl der Elemente des letzten nicht-leeren Datenblocks wurde angepasst, die Anzahl der Elemente im Array wurde angepasst und diverse Zeiger wurden entfernt.

halbiert werden. Daraufhin wird die Variable d, welche die Anzahl der Datenblöcke angibt, verringert. Zusätzlich muss ggf. die Anzahl der Superblöcke verringert werden. Der nun neue letzte Datenblock (falls dieser existiert) wird zum Schluss als voll markiert, da dieser voll sein muss, wenn der darauf folgende Datenblock entfernt wurde. Dies liegt daran, dass Datenblöcke vollständig befüllt werden bevor ein neuer Datenblock erzeugt wird. Wird dieser neue Datenblock daraufhin durch *Schrumpfe* wieder gelöscht, ist der vorige Datenblock noch immer voll. Der Pseudocode der Funktion *Schrumpfe* kann in Algorithm 2 betrachtet werden. Zusätzlich wird in Abbildung 3 der Zustand der Datenstruktur vor und nach dem Aufruf der Operation *Schrumpfe()* dargestellt.

Lokalisiere

Die Aufgabe der Funktion *Lokalisiere(i)* ist es die Position des i-ten Elements, also des Elements mit dem Index i innerhalb der Datenstruktur zu finden. Die Hauptaufgabe liegt dabei auf der Konstruktion der Datenstruktur, damit diese eine Funktion *Lokalisiere(i)* in $O(1)$ ermöglicht. Sofern die Hardware Zweierlogarithmen in konstanter Zeit berechnen kann,

Algorithm 3 Pseudocode: Lokalisiere

```
1. Sei r die binäre Repräsentation von i+1 ohne führende Nullen.
2. Das gesuchte Element i, ist Element e aus Datenblock b von
   Superblock k, wobei:
        2.1 k = |r| − 1.
        2.2 b entspricht den ⌊k/2⌋ Bits von r, die direkt auf
            das führende 1-bit folgen.
        2.3 e entspricht den letzten ⌈k/2⌉ von r.
3. Gebe den Datenblockindex b und die innere Position e zurück.
```

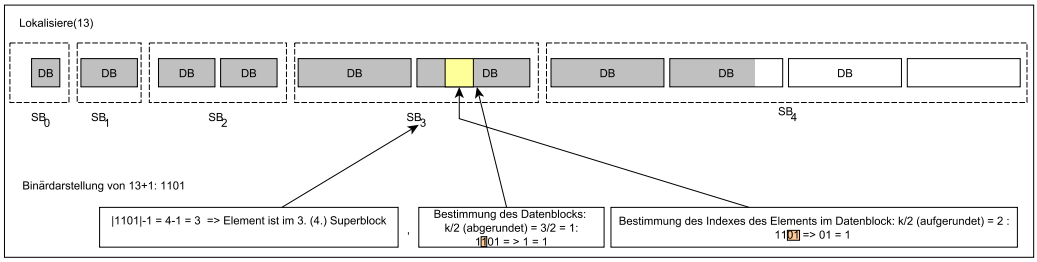


Figure 4 Diese Abbildung beschreibt die Bestimmung der Position des 12-ten Elements in einem optimalen dynamischen Array. Dabei wird nach Algorithm 3 die Position ermittelt.

benötigt auch die Funktion *Lokalisiere(i)* lediglich $O(1)$ Laufzeit.

Zuallererst muss der Superblock SB_k bestimmt werden, indem sich das Element befindet, anschließend wird die Position des Datenblocks innerhalb des Superblocks b bestimmt. Zum Schluss wird die Position des Elements innerhalb des Datenblocks bestimmt.

Da jeder Superblock 2^i Elemente besitzt, haben die ersten k Superblöcke $\sum_{i=i}^{k} 2^i = 2^{k+1}-1$ Elemente. Somit gilt, dass sich das i-te Element im $\lceil \log i \rceil = |(i+1)_2| - 1$-ten[1] Superblock befindet. Da in jedem Superblock $2^{\lfloor k/2 \rfloor}$ Datenblöcke existieren und in jedem Datenblock bis zu $2^{\lceil k/2 \rceil}$ Elemente vorhanden sind (vgl. Unterabschnitt 2.2), kann mit den $\lfloor k/2 \rfloor$ Bits nach der führenden Eins von $(i+1)_2$ die Position des Datenblocks innerhalb des Superblocks und mit den niederwertigsten $\lfloor k/2 \rfloor$ Bits von $(i+1)_2$ die Position des Elements innerhalb dieses Datenblocks bestimmt werden. Die Funktion *Lokalisiere()* wird in Algorithm 3 beschrieben. Zusätzlich wird zur Veranschaulichung die Lokalisierung von Index 12 in Abbildung 4 dargestellt.

2.4 Speicherplatzschranke

▶ **Lemma 1.** *Die Anzahl s der existierenden Superblöcke entspricht immer* $s = \lceil \log_2(1+n) \rceil$.

Proof. Nach Konstruktion der Datenstruktur beinhaltet jeder Superblock i genau 2^i Elemente. Die Anzahl der Elemente in der gesamten Datenstruktur ist einfach zu berechnen:

$$n = \sum_{i=0}^{s-1} 2^i = 2^s - 1$$

[1] Die Betragsstriche stehen für die Länge der Binärdarstellung (Anzahl der benötigten Bits).

Wird dies nach s umgestellt, folgt daraus, dass die Datenstruktur $s = \log_2(n+1)$ Superblöcke besitzt. Dabei muss allerdings zusätzlich aufgerundet werden, da die Datenstruktur nur ganze Superblöcke besitzt. Es folgt, dass die Datenstruktur $\log_1(n+1)$ Superblöcke besitzt. [2] ◀

▶ **Lemma 2.** *Zu jedem Zeitpunkt entspricht die Anzahl der Datenblöcke $O(\sqrt{n})$.*

Proof. Jeder Superblock i besitzt $2^{\lfloor i/2 \rfloor}$ Datenblöcke. Daraus folgt für die gesamte Datenstruktur [2]:

$$d = \sum_{i=0}^{s-1} 2^{\lfloor i/2 \rfloor} \leq \sum_{i=0}^{s-1} 2^{i/2} = \frac{2^{s/2}-1}{\sqrt{2}-1} = \frac{\sqrt{2^s}-1}{\sqrt{2}-1} \overset{\text{Lemma 1}}{=} \frac{\sqrt{n+1}-1}{\sqrt{2}-1}$$

$$\textbf{Somit gilt: } d \in O(\sqrt{n})$$

◀

▶ **Lemma 3.** *Die letzte Datenblock (egal ob leer oder nicht-leer) hat die Größe $\Theta(\sqrt{n})$.*

Proof. Die Anzahl der Datenblöcke im letzten Superblock SB_{s-1} beträgt nach Konstruktion:

$$d_{s-1} = 2^{\lceil (s-1)/2 \rceil} \in \Theta(2^{s/2}) \overset{\text{Lemma 1}}{=} \Theta(\sqrt{2^{\lceil \log_2(1+n) \rceil}}) = \Theta(\sqrt{2^{\log_2(n)}}) = \Theta(\sqrt{n})$$

Da alle Datenblöcke in diesem letzten Superblock die selbe Größe haben, besitzen auch die leeren Datenblöcke $\Theta(\sqrt{n})$ Elemente. [2] ◀

Mit Hilfe dieser Lemmas ist der Beweis des benötigten Extraspeicherplatzes trivial. Nur die letzten beiden Datenblöcke können nicht voll sein. Diese haben laut Lemma 3 die Größe $\Theta(\sqrt{n})$. Daraus folgt, dass $O(\sqrt{n})$ Extraspeicherplatz durch leere Datenblöcke verbraucht wird. Weiterhin verschwendet der Indexblock im Worstcase viermal die Indexblockgröße (durch den Doppel-, Halbindexblock und konstante Erweiterungen). Daraus folgt, dass auch der Indexblock $O(\sqrt{n})$ Extraspeicherplatz verbraucht. Alle Datenblöcke haben *Headerinformationen* der Größe h. Da nach Lemma 2 $O(\sqrt{n})$ Datenblöcke existieren, folgt derselbe Extraspeicherplatzverbrauch. Weiterhin verbrauchen alle weiteren Zeiger lediglich konstant viel Speicherplatz.

Alles in allem verwendet somit die gesamte Datenstruktur $O(\sqrt{n})$ Extraspeicherplatz und ist somit in dieser Hinsicht nach Abschnitt 2.6 optimal.

2.5 Laufzeitschranke

Um die Zeitschranken von $O(1)$ zu beweisen, wird zuerst die Gültigkeit der Schranke für *Lokalisiere* gezeigt und anschließend für *Schrumpfe* und *Wachse*.

Die Laufzeit der Funktion *Lokalisiere* ist abhängig von der verwendeten Hardware. Kann diese 2-er Logarithmen in konstanter Zeit berechnen, dann sind alle Instruktionen aus dem Pseudocode in konstanter Zeit ausführbar und die Funktion hat auch eine konstante Laufzeit. Dies sollte auf sehr viele Rechnerarchitekturen zutreffen. [3] nennt eine zusätzliche Implementierungsmöglichkeit, die das gesamte Problem mit boolscher Logik und Arithmetik löst.

Für die Laufzeituntersuchung von *Wachse* und *Schrumpfe* wird zunächst die Anzahl der Operationen zwischen zwei Aufrufen von *Reserviere* und/oder *Freigeben* aus Theorem 1 bewiesen. Dabei beschreibt die Operation *Reserviere* die Reservierung von Speicherblöcken und *Freigeben* das Freigeben von Speicherblöcken. Anschließend wird mit diesem Wissen

die Laufzeit der Größenänderung des Indexblocks untersucht, denn alle anderen Teile sind offensichtlich in konstanter Laufzeit berechenbar. Ein neuer Datenblock wird nur belegt, wenn der letzte Datenblock voll ist. Da der letzte volle Datenblock laut Lemma 3 $\Theta(\sqrt{n})$ Elemente besitzt bedeutet das im speziellen, dass alle diese Elemente vor dem Aufruf von *Reserviere* durch *Wachse* hinzugefügt wurden. Deshalb gab es zwischen dem ersten *Wachse* und dem aktuellen *Reserviere* kein anderes *Reserviere* oder *Freigeben*. Daraus folgt, dass $\Omega(\sqrt{n})$ andere Operationen vor dem *Reserviere* Aufruf ausgeführt wurden. Das Selbe gilt vor jeden Aufruf von *Freigeben*. Ein Datenblock wird nur gelöscht, wenn ein weiterer leer ist. Da der letzte Datenblock somit nur angelegt werden konnte, wenn der vorletzte Datenblock voll war, folgt aus Lemma 3, dass der vorletzte Datenblock im Worstcase Größe $\Theta(\sqrt{n}/2) = \Theta(\sqrt{n})$ besitzt und dieser einmal durch Schrumpfe komplett geleert werden muss, bevor der letzte Datenblock entfernt werden darf. Somit werden vor jedem Freigeben $\Omega(\sqrt{n})$ Operationen ausgeführt. Es wird davon ausgegangen (vgl. Theorem 1), dass das Freigeben und das Reservieren von $\sqrt{n}$ Speicherplatz in konstanter Laufzeit möglich ist.

Der wichtigste zu untersuchende Teil der Laufzeiten von *Wachse* und *Schrumpfe* ist die Größenänderung des Indexblocks. Dies geschieht wie zuvor betrachtet entweder, wenn der Indexblock voll ist und somit laut Lemma 2 $O(\sqrt{n})$ Einträge besitzt, oder nach unten hin, wenn der Indexblock nur noch ein Viertel seiner Einträge besitzt. Da ein Indexblock zum Zeitpunkt seiner Erzeugung von der Funktion *Wachse* immer zur Hälfte befüllt wird[2], ist erkennbar, dass asymptotisch $\Omega(\sqrt{n})$ *Reserviere*- oder *Freigebe*-Operationen ausgeführt werden, bevor seine Größe angepasst wird. Die Größenänderung des Indexblocks benötigt im Worstcase $O(\sqrt{n})$ Laufzeit (durch das kopieren). Wie bereits gezeigt, werden zwischen zwei *Reserviere*- und *Freigebe*-Operationen $\Omega(\sqrt{n})$ Datenstrukturänderungsoperationen ausgeführt. Nun ist es einfach, mittels einer amortisierten Analyse (z. B. mit der Kontenmethode) für die Größenänderung des Indexblocks die Laufzeit $O * (1)$ zu zeigen. Dafür müssen die zu hohen Kosten dieser Operation ($O(\sqrt{n})$) durch eine konstante Erhöhung der Laufzeitkosten der Datenstrukturänderungsoperationen kompensiert werden, was problemlos möglich ist.

Durch eine Anpassung der Datenstruktur ist allerdings auch eine echte Worstcaselaufzeit in $O(1)$ möglich. Hierfür wird das Kopieren des gesamten bzw. ein Viertel des Indexblocks verhindert. Dies geschieht, indem immer drei Indexblöcke gespeichert werden. Einen mit normaler Größe, einen mit doppelter Größe und einen mit halber Größe. Zusätzlich existieren zwei Zähler, die jeweils immer erhöht werden, wenn aus dem Hauptindexblock ein Element in den Doppelindexblock oder in den Halbindexblock kopiert werden. Immer wenn ein Datenblock erzeugt wird, wird dieser im Indexblock eingetragen und die letzten zwei noch nicht kopierten Datenblockzeiger in den Doppelindexblock kopiert. Wird ein Datenblock entfernt, werden zwei nicht kopierte Blöcke in den Halbindexblock kopiert. Zusätzlich werden ggf. die Kopien des Datenblocks gelöscht. Durch dieses Vorgehen wird erreicht, dass bei einem vollen Indexblock bereits alle Datenblockzeiger im Doppelindexblock sind und bei einem ein Viertel vollen Indexblock, alle Datenblöcke im Halbindexblock sind. Anschließend wird bei vollem Indexblock der alte Indexblock zum Halbindexblock, der alte Doppelindexblock zum Indexblock, ein neuer Doppelindexblock erzeugt und der alte Halbindexblock freigegeben. Ist der Indexblock dagegen ein Viertel voll, dann wird der alte Halbindexblock zum Indexblock, der alte Indexblock zum Doppelindexblock, der alte Doppelindexblock freigegeben und ein neuer Halbindexblock reserviert. Diese Änderung der Datenstruktur ist sehr nahe an der Umverteilung der Kosten, die in der amortisierten Analyse gemacht wird. Somit wurde die $O(1)$ Laufzeit aller Operationen (vgl. Theorem 1) gezeigt.

[2] Eine Ausnahme hierbei ist die Erzeugung eines leeren Arrays.

2.6 Optimalität

In diesem Abschnitt wird eine untere Extraspeicherplatzschranke für beliebige dynamisch wachsende Datenstrukturen bewiesen. Diese Schranke gilt offensichtlich auch für dynamisch wachsende Arrays.

▶ **Theorem 4** (Untere Extraspeicherplatzschranke für dynamische Datenstrukturen)**.** *Jede Datenstruktur, die das Hinzufügen und Löschen von Elementen in irgendeiner Reihenfolge unterstützt, benötigt* $\Omega(\sqrt{n})$ *Extraspeicher.*

Proof. Wir betrachten eine beliebige Einfüge und Löschsequenz mit folgendem Schema:

$$Einfügen(a_1), ..., Einfügen(a_n), \underbrace{Lösche, ..., Lösche}_{n-mal}$$

Diese Sequenz kann für beliebige n betrachtet werden. Besonders wird der Zustand der Datenstruktur zwischen den Einfüge- und Löschoperationen betrachtet: Sei $f(n)$ die Größe des größten Speicherblocks und sei $g(n)$ die Anzahl aller Speicherblöcke nach der n-ten Einfügeoperation. Da auf jedes Element zugegriffen werden kann, muss $f(n) \cdot g(n)$ mindestens n entsprechen. Anderenfalls wären nicht alle n Elemente in der Datenstruktur gespeichert. Denn im Worstcase hat jeder Speicherblock Größe $f(n)$ und jedes Element hat mindestens Größe Eins. Es gilt allerdings $f(n) \cdot g(n) \geq n$, da ein Element auch größer Eins sein kann und da nicht jeder Block die maximale Größe haben muss.

Weiterhin benötigt jeder Speicherblock *Headerinformationen*, die eine untere Schranke $h * g(n)$ für den Extraspeicherplatz liefern. Somit muss der benötigte Worstcaseextraspeicherplatz mindestens $g(n)$ betragen (falls die *Headerinformationen* Größe Eins haben). Wird der Speicherblock der Größe $f(n)$ reserviert und noch nicht direkt verwendet, dann wird in diesem Augenblick zusätzlich $f(n)$ Extraspeicherplatz verwendet.

Aus diesen beiden Überlegungen folgt, dass der benötigte Extraspeicherplatz mindestens $max\{f(n), g(n)\}$ beträgt. Da allerdings $f(n) \cdot g(n) \geq n$ gilt, folgt dass $max\{f(n), g(n)\}$ mindestens $\sqrt{n}$ beträgt. ◀

3 Bewertung

Die Vorteile dieser Datenstruktur wurden implizit bereits des öfteren erwähnt. Sie ist optimal sowohl im verbrauchten Extraspeicherplatz, als auch in der Laufzeit ihrer Operationen. Ein weiterer großer Vorteil ist, dass bei der Vergrößerung der Datenstruktur (des Indexblocks) keine Nutzdaten verschoben werden müssen, sondern nur die Zeiger auf die Datenblöcke. Diese sind kleiner als die Nutzdaten und zeigen auf $(O(\sqrt{n}))$ Elementen. Somit ist das Kopieren aller Indexzeiger günstiger als das Kopieren aller Elemente. Das macht diese Datenstruktur auch für große Rechnersysteme mit großen dynamischen Datensätzen interessant.

Ein entscheidender Nachteil dieser Datenstruktur ist die Zugriffszeit auf die Elemente. Die Lokalisierung benötigt zwar konstante Laufzeit, die Konstante ist allerdings durchgehend schlechter als die von üblichen Implementierungen. Zusätzlich muss auf zwei Speicherstellen zugegriffen werden. Einmal auf den Indexblock und einmal auf das Element im Datenblock. Hierbei kann es zu zwei teuren *Cach-Misses* kommen. Durch die deutlich kleinere Größe des Indexblocks im Vergleich zur Anzahl der Elemente, ist die Wahrscheinlichkeit eines *Cache-Miss* im Indexblock niedriger als innerhalb der Datenblöcke. Im Vergleich zu einer Datenstruktur, die alle Elemente hintereinander im Array speichert, ist die *Cache-Miss*--Wahrscheinlichkeit

etwas höher, da die Datenblöcke nicht zwingend in einem zusammenhängenden Speicherbereich liegen müssen. Auch diese *Cacheproblematik* kann einen negativen Einfluss auf die Laufzeit der Zugriffsoperationen haben.

References

1 Andrej Brodnik, Svante Carlsson, Erik D Demaine, J Ian Ian Munro, and Robert Sedgewick. Resizable arrays in optimal time and space. In *Workshop on Algorithms and Data Structures*, pages 37–48. Springer, 1999.

2 Andrej Brodnik, Svante Carlsson, Erik D Demaine, J Ian Ian Munro, and Robert Sedgewick. Resizable arrays in optimal time and space. In *Technical Report CS-99-09*, 1999.

3 Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer auF der Heide, Hans Rohnert, and Robert E Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–761, 1994.

4 William H Press, Saul A Teukolsky, and William T Vetterling. Bp flannery numerical recipes in c: the art of scientific computing. *Cambridge University Press, Cambridge, UK*, 1992.

Funnel Heap*

Uriel Elias Wiebelitz[1]

1 **Fakultät für Informatik, TU Dortmund, 44227 Dortmund**
Matrikelnummer: 162556
uriel-elias.wiebelitz@tu-dortmund.de

Zusammenfassung

Die meisten modernen Vielzwecksysteme verfügen über ein mehrschichtiges Cache-System, welches aktuell bearbeitete Daten möglichst direkt verfügbar hält. Bereits seit einiger Zeit wird beim Entwurf von Algorithmen das Externspeichermodell von Aggarwal und Vitter herangezogen. Dieses sog. I/O-Modell setzt voraus, dass den Algorithmen und Datenstrukturen Informationen über die Block- und Speichergröße zur Verfügung stehen. Im Gegensatz dazu haben die Algorithmen im Cache-oblivious-Modell keine Kenntnis davon, ob das System über Caches verfügt daher natürlich auch nicht über die oben genannten Parameter. In dieser Ausarbeitung wird die Cache-oblivious Prioritätswarteschlange *Funnel Heap* von Brodal und Fagerberg vorgestellt. Bei dieser Datenstruktur handelt es sich um eine Heap-Struktur, die auf binären Verschmelzungen und k-Wege Verschmelzungen basiert. Zu letzteren stellen wir die Datenstruktur *Lazy Funnelheap* vor, die ausschließlich durch binäre Merger umgesetzt ist, sodass die gesamte Datenstruktur als die geschickte Kombination von binären Verschmelzungen verstanden werden kann. Neben den Grundlagen des cache-oblivious Sortierens wird außerdem die Struktur des Funnel Heap erörtert, wie auf ihm die Operationen DeleteMin und Insert umgesetzt werden können und abschließend die I/O-Komplexität der Datenstruktur betrachtet.

Keywords and phrases Funnel Heap, Prioritätswarteschlange, Cache-oblivious, Lazy Funnelsort

Digital Object Identifier 10.4230/LIPIcs.BADS.2017.11

1 Cache-Systeme

Heutige Vielzwecksysteme verfügen über mehrschichtige Cache-Systeme. Bei solchen Caches handelt es sich um extrem schnelle Speichereinheiten, die jedoch auf Grund des Platzbedarfs und ihrer speziellen Architektur verhältnismäßig klein ausfallen. Sie dienen dazu, häufig verwendete Daten möglichst verfügbar zu halten, um nicht ständig auf den Arbeitsspeicher zugreifen zu müssen, da ein solcher Zugriff trotz der mittlerweile schnellen Einheiten ca. 100 Taktzyklen in Anspruch nimmt [4], wie in Abbildung 1 dargestellt. Bei Cache-Systemen wird grundsätzlich angenommen, dass Daten, die in zeitlicher Nähe benötigt werden, auch im Speicher beieinander liegen. Dazu werden bei einem Speicherzugriff nicht nur einzelne Bits oder Bytes geladen, sondern immer Speicherblöcke (sog. *Cachelines*). Dadurch können viele Speicherzugriffe eingespart werden, weil folgend angefragte Daten vermutlich schon mit einer vorherigen Anfrage in den Cache geladen wurden. Sollte ein angefragtes Datum nicht im Cache gefunden werden können, kommt es zu einem Cache-Miss und ein neuer Block muss aus der nächst langsameren Cache-Ebene geladen werden.

Zum Scheduling der Cache-Inhalte werden in der Regel Algorithmen verwendet, die keine Kenntnis über die aktuell ausgeführten Algorithmen haben, was ein übliches Merkmal von Vielzwecksystemen ist.

* Diese Arbeit basiert im Wesentlichen auf der Vorstellung der Datenstruktur in [3] von Brodal und Fagerberg.

2nd Symposium on Breakthroughs in Advanced Data Structures.
Editors: Johannes Fischer and Dominik Köppl and Florian Kurpicz; Article No. 11; pp. 11:1–11:10

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

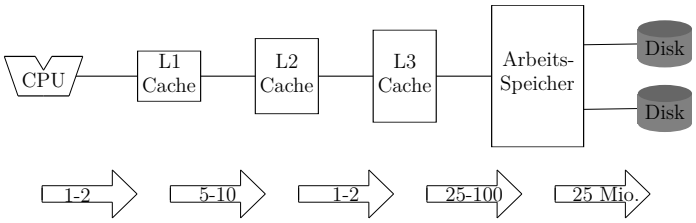


Abbildung 1 Darstellung eines mehrschichtigen Chache-Systems. Die Pfeilbeschriftungen geben die ungefähr benötigten Taktzyklen für den Zugriff an. (In Anlehnung an [4])

1.1 Externe Datenstrukturen

Auch der Arbeitsspeicher selbst lässt sich als eine Cache-Ebene verstehen. Bei sehr speicherbedürftigen Datenstrukturen reicht die Kapazität des Arbeitsspeichers alleine nicht aus, um sämtliche Daten vorzuhalten. In solchen Fällen müssen große Teile der Daten auf Speichermedien wie Festplatten ausgelagert werden. Bekannte Beispiele sind etwa Datenbanken oder Navigationsgeräte. Um diese Daten verwenden zu können müssen, diese wiederum in den Arbeitsspeicher geladen werden, was noch deutlich mehr Zeit in Anspruch nimmt, als ein Arbeitsspeicherzugriff (HDD: 25 Mio. Taktzyklen, bei 2,5 Ghz und einer Zugriffszeit von 10 ms). Um dennoch möglichst effizient mit solchen Daten arbeiten zu können entwickelten Aggarwal und Vitter in [1] das Externspeichermodell, welches von den Hardwareaspekten weitestgehend abstrahiert, allerdings die notwendigen Informationen für den Algorithmenentwurf berücksichtigt: Den gesamt verfügbaren Speicher im Cache (bezeichnet als M) sowie die Speichergröße, die in einem Block geladen wird (bezeichnet als B). Ziel der Entwicklung von Externspeicher-Datenstrukturen und -Algorithmen ist es, trotz der Nutzung externen und langsameren Speichers optimale Performance zu erreichen.

1.2 Cache-Oblivious

Auch wenn das von Aggarwal und Vitter [1] vorgestellte Modell ein problemspezifischeres Algorithmen-Design erlaubt, erfordert es doch spezifische Kenntnisse über die Hardwareeigenschaften. Das Cache-oblivious-Modell versucht genau diese Schwierigkeit auszuräumen: Die Algorithmen haben keine Kenntnis, wie der Cache geartet ist, sie verfügen weder über Informationen hinsichtlich der Cachegröße noch über die Blockgrößen. Dementsprechend werden Algorithmen und Datenstrukturen so ausgelegt, dass sie unabhängig von diesen Kenngrößen effizient arbeiten.

2 Funnelsort

Bei der in dieser Ausarbeitung beschriebenen Datenstruktur *Funnel Heap* handelt es sich um eine optimale Min-Heap-Datenstruktur in Bezug auf I/Os im Cache-Oblivious-Modell.

Ein Heap stellt die beiden Operationen Insert und DeleteMin zur Verfügung. Letztere gibt stets das kleinste im Heap befindliche Element aus. Durch leichte Abänderung der Datenstruktur kann in vergleichbarer Weise ein Max-Heap aufgebaut werden. Die Ausführung von DeleteMin in konstanter Zeit setzt voraus, dass die Elemente derart organisiert sind, dass das kleinste Element unabhängig von der Anzahl der enthaltenen Elemente gefunden werden kann. Der *Funnel Heap* hält die eingefügten Elemente dazu stets sortiert. Dies wird ausschließlich durch binäre Merge-Sortierung erreicht. Dabei wird auf sog. k-Merger zurück gegriffen, die im Folgenden erläutert werden.

2.1 Merge-Sort

Ein weit verbreiteter Sortieralgorithmus ist der Merge-Sort. Er zerteilt die zu sortierenden Daten in Teilmengen, sortiert diese (rekursiv) in sich und verschmilzt (merged) die sortierten Daten schließlich wieder. Dabei werden in jedem Merge-Schritt die ersten Elemente der (sortierten) Teilmengen verglichen und das kleinste (bzw. größte) Element in die Ausgabe geschrieben. Dieser Algorithmus erreicht in Worst-, Best- und Average-Case stets eine Laufzeit von $O(n \cdot \log(n)))$. Auch wenn dieser Algorithmus Cache-oblivious implementiert werden kann und seine Laufzeit dem aktuell bekannten Optimum für vergleichsbasiertes Sortieren ohne Parallelisierung auf beliebigen Eingaben entspricht [6], ist er dennoch nicht optimal in Hinsicht auf Cache-Misses.

2.2 Lazy Funnelsort

Basierend auf der Idee des Merge-Sorts stellten Figro et al. [5] den Funnel-Sort-Algorithmus vor, der asymptotisch optimal in Hinsicht auf Laufzeit und Cache-Misses ist. Im Unterschied zum Merge-Sort wird beim Funnel-Sort nicht davon ausgegangen, dass in einem Rekursionsschritt sämtliche Daten des vorherigen Schritts verarbeitet werden, sondern dass die Daten jeweils „auf Nachfrage“ erzeugt werden.

Brodal et al. [2] entwickelten eine abgewandelte Form des Funnelsort, die ausschließlich auf binären Mergern basiert. Diese Datenstruktur ist binär gefaltet, lässt sich also als Binärbaum repräsentieren, wobei die Knoten die binären Merger sind und die Puffer entlang der Kanten zwischen den Mergern liegen (siehe Abbildung 3).

Die Daten werden von sog. k-Mergern bereitgestellt. Ein k-Merger merged Daten von k sortierten Eingängen zu einer sortierten Ausgabe. Abbildung 2 veranschaulicht den Aufbau eines solchen Mergers. Er lässt sich als balancierter Binärbaum interpretieren, bei dem die binären Merger die Knoten darstellen und die Puffer entlang der Kanten liegen. Balanciert bedeutet, dass die Länge alle Wege von der Wurzel zu den Blättern um maximal 1 abweicht.

Die Eingänge eines k-Mergers werden aufgeteilt und an interne Merger als Input übergeben. Die Aufteilung in Kind-Merger erfolgt rekursiv und abhängig von der Höhe i des Binärbaumes: Die Ebenen von der Wurzel bis zur Ebene $\lceil i/2 \rceil$ gehören zum top-Merger. Die Merger der Ebene $\lceil i/2 \rceil + 1$ bilden die Wurzelknoten der bottom-Merger. Die Größe aller Puffer zwischen top-Merger und den bottom-Mergern beträgt $\lceil k^{3/2} \rceil$ Elemente. Der Ausgangspuffer des k-Mergers ist k^3 Elemente groß. In den Abb. 2 und 3 ist ein 8-Merger mit den entsprechenden Puffergrößen dargestellt.

Werden nun Daten vom k-Merger angefordert und der Ausgangspuffer ist leer, so erzeugt er eine Ausgabe von k^3 Elementen – diese Operation wird als Fill bezeichnet. Der Aufruf von Fill erfolgt für den rechtesten (letzten) Merger innerhalb des k-Mergers, der ggf. wiederum Fill seine verknüpften Eingangs-Merger (die Kind-Merger) aufruft:

Es werden wiederholt Mergeschritte ausgeführt, bis der Ausgabepuffer voll ist oder beide Eingangspuffer leer gelaufen sind. Zu Beginn eines jeden Mergeschrittes wird geprüft, ob einer der Eingangspuffer leer ist; sollte dies der Fall sein, wird Fill für den entsprechenden Kind-Merger aufgerufen. Kann dieser keine Daten mehr liefern, wird Fill beendet und markiert, dass keine weiteren Daten mehr verfügbar sind (*exhausted*). Daraufhin wird das eigentliche Zusammenführen vollzogen, d.h. die jeweils ersten Elemente der Eingangspuffer werden verglichen und das kleinere in den Ausgangspuffer geschrieben.

Der Algorithmus sortiert in die Eingangsdaten in $O(n \cdot \log(n))$ [5], was wie oben beschrieben dem bisher bekannten Minimum entspricht. Außerdem lässt sich beweisen, dass der Algorithmus maximal $O\left(3(n/B)\log_M(n)\right)$ Cache-Misses und somit I/Os hervorruft, um n

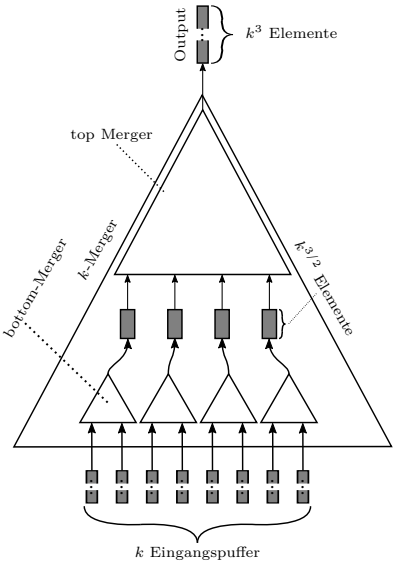


Abbildung 2 Darstellung eines k-Mergers für $k = 8$. Ein k-Merger verfügt über k Eingänge, die wiederum auf Kind-Merger aufgeteilt werden. Deren Ausgänge führen in Puffer der Größe $k^{3/2}$, die wiederum in einem top-Merger verschmolzen werden, der den Ausgabepuffer versorgt. Dies wird rekursiv so lange wiederholt, bis nur noch zwei Puffer mit einem binären Merer zusammengeführt werden.

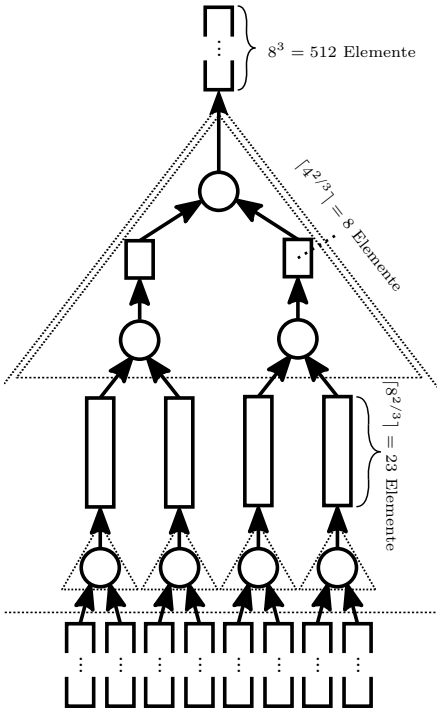


Abbildung 3 K-Merger im Lazy Funnelsort sind perfekt balancierte Binärbäume. Die Knoten (Kreise) sind binäre Merger. Entlang der Kanten liegen die Puffer (Rechtecke).

Elemente zu sortieren (siehe [2, S. 431]). Nach Aggarwal und Vitter [1] existiert eine untere Grenze für die Cache-Misses eines jeden Sortieralgorithmus von $\Omega((n/B)\log_{M/B}(n/M))$. Zu beachten ist hierbei, dass vorausgesetzt wird, dass $M = \Omega(B^2)$ ist, was bedeutet, dass der Cache mindestens eine Größe von B^2 hat (bezeichnet als *tall cache assumption*).

3 Funnel Heap

Die von Brodal und Fagerberg vorgestellte Cache-oblivious Heap-Datenstruktur Funnel Heap [3] basiert ausschließlich auf binären Mergern, die von dem oben beschriebenen Funnelsort-Algorithmus inspiriert wurden. Sie unterstützt die üblichen Heap-Funktionen Insert und DeleteMin.

3.1 Aufbau der Datenstruktur

Die Datenstruktur ist in Abbildung 4 dargestellt. Wie schon in der Darstellung der k-Merger des Lazy Funnelsort, sind auch hier einfache binäre Merger durch Kreise repräsentiert. Durch die Verwendung von Lazy Funnelsort für die k-Merger (dargestellt als Dreiecke), der selbst nur auf binären Mergern basiert, kommt diese Datenstruktur allein mit binären Mergern und Puffern aus. Während der Lazy Funnelsort auf perfekt ausbalancierten Bäumen basiert, ist direkt zu erkennen, dass der Binärbaum, der aus dieser Datenstruktur resultiert als ganzes nicht länger balanciert ist. Stattdessen sind die wachsen die k-Merger K_i und ihre Eingangspuffer mit steigendem i. Außerdem werden erweiternde Links stets im rechten

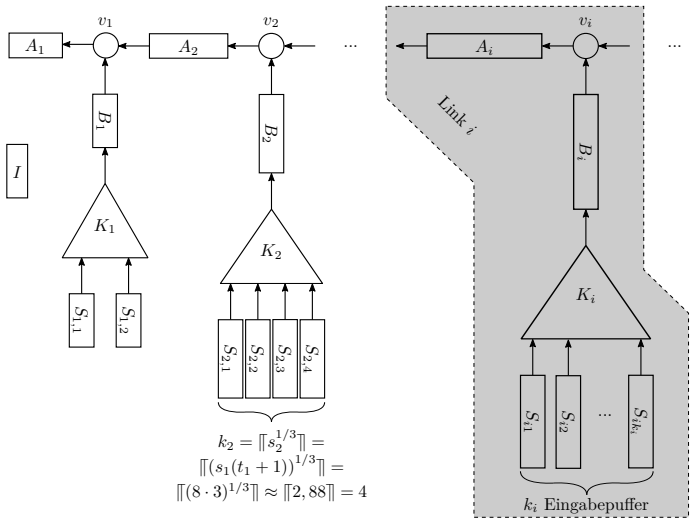


Abbildung 4 Funnel-Heap – in Anlehnung an [3, Fig. 3]. Er besteht aus einem Eingabepuffer I, der jedes eingefügte Element zunächst zwischenspeichert. Ist dieser voll, werden die Elemente in den Baum mit der Wurzel A_i eingefügt. Der Baum besteht aus erweiterbar vielen Links, wobei jeder Link i aus Eingabepuffern $S_{i1}, ..., S_{ik_i}$ der Größe s_i, einem k_i-Merger K_i gemäß *Lazy Funnelsort*, dem Puffer B_i, sowie einem binären Merger v_i, der Elemente aus dem Link i und den Links $> i$ in A_i vereint.

Teilbaum von v_i eingeordnet. Ein solcher Link verfügt über einen Ausgabepuffer A_i, einem binären Merger v_i, einem K_i-Merger mit dessen Eingabepuffern $S_{i1}, ..., S_{ik_i}$, sowie dessen Ausgabepuffer B_i. Außerdem beinhaltet jeder Link einen Zähler c_i, der später zum Einfügen benötigt wird. Dieser Zähler gibt an, ab welchem Eingabepuffer des k-Mergers nur noch leere Puffer folgen. Es gilt die Invariante, dass die Puffer $S_{ic_i}, ..., S_{ik_i}$ leer sind. Es gilt $1 \leq c_i \leq k_i + 1$, wobei c_i mit 1 für den leeren Heap initialisiert wird. Im Arbeitsspeicher werden die einzelnen Teilelemente eines Links in der Reihenfolge c_i, A_i, v_i, B_i, K_i, $S_{i1}, ..., S_{ik_i}$ vorgehalten. Diese Struktur im Speicher ist maßgeblich für die benötigten I/Os.

Ein weiterer sortierter Puffer I wird für das Einfügen, das in Abschnitt 3.3 betrachtet wird, benötigt. Die gesamte Datenstruktur wird in der Abfolge I, Link 1, Link 2, ... im Speicher abgelegt.

3.1.1 Wachstum der Datenstruktur

Wie bereits genannt, wächst die Größe der Links. Um dieses Wachstum zu betrachten, definieren wir die Größen der Einzelnen Puffer, sowie der k-Merger.

Die Ausgabegröße eines k-Mergers richtet sich nach den Elementen, die beim Aufruf des k-Mergers erzeugt werden. Somit muss B_i die Größe k^3 haben. Auch die Größe von A_i wird als k^3 gewählt. Die Größe der Eingabepuffer $S_{i1}, ..., S_{ik_i}$ wird mit s_i bezeichnet. Dem Eingabepuffer I wird die Größe der Eingabepuffer des ersten k-Mergers zugeordnet: s_i.

k_i und s_i sind wie folgt induktiv definiert:

▶ **Definition 1.**

$$\begin{aligned}(k_1, s_1) &= (2, 8)\\ s_{i+1} &= s_i(k_i + 1)\\ k_{i+1} &= \lceil\!\lceil {s_{i+1}}^{1/3} \rceil\!\rceil = 2^{\lceil \log({s_{i+1}}^{1/3}) \rceil}\end{aligned}$$

Wobei $\lceil\!\lceil x \rceil\!\rceil$ die kleinste Zweierpotenz bezeichnet, die größer als x ist, d.h. $\lceil\!\lceil x \rceil\!\rceil = 2^{\lceil \log(x) \rceil}$, und $i \geq 1$ gefordert wird. Dadurch wird gewährleistet, dass die Anzahl der Eingangspuffer eines k-Mergers stets einer Zweierpotenz entspricht, was die binäre Faltung ermöglicht.

3.2 Heapeigenschaften

Der Funnel Heap kann durch einfache Abänderung der Merger kleinste oder größte Elemente ausgeben. Im Folgenden wird ausschließlich die Ausgabe der kleinsten Elemente betrachtet.

Die Daten der gesamten Datenstruktur befinden sich gemäß ihres Aufbaus im Puffer I, in den Puffern entlang der Kanten des Baums, sowie in den Eingangspuffern der k-Merger. Die k-Merger sortieren ihre jeweiligen Eingangsdaten. Dementsprechend befinden sich in ihren Ausgangspuffern stets kleinere Elemente, als in den Eingabepuffern verbleiben. Selbiges gilt für die Merger v_i in den Links. Somit fallen die Werte der Elemente in den Puffern in Richtung A_1 monoton. Anders ausgedrückt sind die Elemente entlang jedes möglichen Weges von A_1 zu den Blättern in aufsteigender Folge sortiert. Die Elemente in den Eingabepuffern $S_{i1}, ..., S_{ik_i}$ sind bereits beim Einfügen sortiert, was im nächsten Abschnitt erklärt wird.

Da der einzige Puffer außerhalb dieses sortierten Baums der Puffer I ist, folgt, dass sich das kleinste Element entweder in diesem Puffer I oder in A_1 befindet. Sollte der Puffer A_i leer sein, muss zunächst Fill(v_1) aufgerufen werden.

Beim Aufruf von DeleteMin muss schließlich lediglich das kleinste Element aus den sortierten Puffern I und A_1 entfernt und zurückgegeben werden.

3.3 Einfügen

Um eine Einfügung zu vollziehen (Insert) wird das Element in den Puffer I eingefügt, wobei es der Sortierung entsprechend einzufügen ist, damit I sortiert bleibt. Es wird immer sichergestellt, dass I nicht voll ist und das Element somit eingefügt werden kann. Sollte I nach dem Einfügen voll sein (der Puffer enthält s_1 Elemente) (siehe Abb. 5 (1)), rufen wir Sweep(i) auf, was die Elemente aus den niedrigeren Links gesammelt in einen freien größeren Eingabepuffer eines höheren Links verschiebt. Dadurch skalieren der Aufwand und die notwendigen I/Os mit der Anzahl der Elemente in der Datenstruktur. Dass dies platztechnisch möglich ist, wird im Abschnit 3.4 „Korrektheit“ näher betrachtet.

Im Folgenden betrachten wir den Aufruf von Sweep(i). Der Parameter i gibt den niedrigsten Index an, bei dem $c_i \leq k_i$. Zur Erinnerung: Es wird der kleinste und erste Link angegeben, in dem mindestens der letzte Eingabepuffer leer ist. Ziel der Operation ist es, alle Inhalte der Links $1, ..., i-1$ (inklusive der Elemente in den k-Mergern) in diesen freien Puffer S_{ic_i} (in Abb. 5 als $S_{3,2}$ gekennzeichnet) zu verschieben. Dazu werden insgesamt drei Hilfspuffer σ_1, σ_2 und σ angelegt.

Zunächst durchlaufen wir den Weg p von A_1 nach S_{ic_i} und speichern die Anzahl der enthaltenen Elemente jedes Puffers auf dem Weg. Dabei speichern wir die Elemente auf auf dem Teilweg von A_1 bis S_{ic_i}, d.h. die Puffer A_j für $j = 1, ..., i$, B_i, sowie die Puffer auf dem Weg innerhalb des k-Mergers K_i in σ_1. Dies kann bereits während des Zähldurchlaufs erledigt werden. In σ_2 werden alle verbleibenden Elemente der Links $1, ..., i-1$ sowie aus dem Puffer I gespeichert. Dazu wird der Puffer A_i als *leer gelaufen (exhausted)* gekennzeichnet und DeleteMin wiederholt aufgerufen, wodurch die Elemente sortiert in σ_2 abgelegt werden. Durch die Markierung von A_i als *leer gelaufen* vermeiden wir, dass dabei keine Links $\geq i$ aktiviert werden (vgl. Abb. 5 (2)). Daraufhin führen wir die beiden Puffer σ_1 und σ_2 zu einem sortierten Puffer σ zusammen (Abb. 5 (3)), was mit Hilfe eines binären Mergers in $O(|\sigma_1| + |\sigma_2|)$ möglich ist, da σ_1 und σ_2 bereits sortiert sind. Bei einem zweiten Durchlauf des Pfades p fügen wir die ersten Elemente von σ wieder in die Puffer von p ein, sodass diese wieder dieselbe Anzahl von Elementen enthalten, wie vor dem Aufruf von Sweep (siehe Abb. 5 (4)). Die restlichen Elemente aus σ fügen wir in den gewählten Eingabepuffer S_{ic_i} (in der Abb. $S_{3,2}$). Abschließend setzen wir sämtliche Zähler c_l der Links $l = 1, 2, ..., i-1$ auf 1

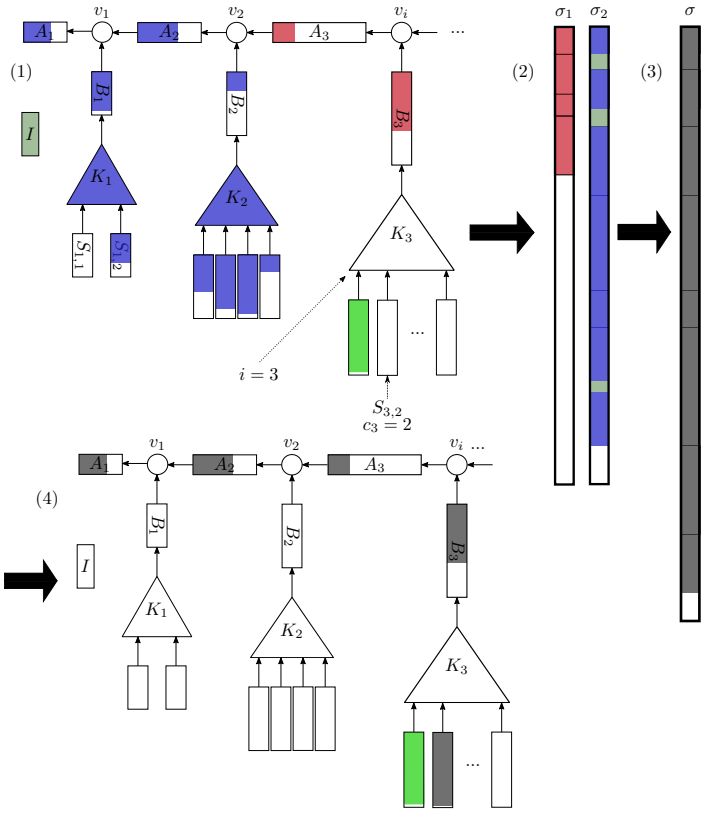


Abbildung 5 Ausführung von SWEEP während INSERT in Funnel-Heap. Die rot markierten Elemente werden zunächst sortiert in σ_1 verschoben (sie sind von A_1 in Richtung $S_{3,2}$ sortiert) (2). Außerdem werden die blau und olivgrün hervorgehobenen Elemente sortiert in σ_2 verschoben, wobei die Sortierung durch Kennzeichnung von A_3 als *leer gelaufen* und wiederholte Aufrufe von DELETEMIN erreicht wird. In (3) werden σ_1 und σ_2 sortiert in σ zusammengeführt und schließlich wieder in den Funnel Heap eingefügt (4). Der erste Puffer von K_1 ($S_{1,1}$) ist zu Beginn von SWEEP (1) zwar leer, allerdings ist $c_1 = 2$, da der zweite Puffer ($S_{1,2}$) Elemente enthält. Die grün gekennzeichneten Daten werden nicht verschoben.

zurück, wodurch gekennzeichnet wird, dass die Eingabepuffer dieser Links wieder leer sind. Außerdem wird c_i um eins erhöht.

3.4 Korrektheit

Die Korrektheit von DELETEMIN ist schnell ersichtlich: Die Korrektheit der Ausgabe des Baums ergibt sich unmittelbar aus der Korrektheit des Lazy Funnelsort, die in [2] gezeigt wurde. Der Puffer I ist der einzige Teil der Datenstruktur, der nicht direkt vom Baum erfasst wird. Daher ergibt sich die Korrektheit von DELETEMIN auf der gesamten Datenstruktur daraus, dass immer das kleinste Element aus dem Baum und I gewählt, gelöscht und zurückgegeben wird.

Die Korrektheit von INSERT wird im Folgenden genauer betrachtet. Beim Einfügen wird das Element in I der Sortierung entsprechend eingefügt. Sollte I noch nicht voll sein, ist damit die Invariante, dass I sortiert ist weiterhin erfüllt. Außerdem wird sicher gestellt, dass in I immer mindestens ein Element eingefügt werden kann, da I sofort wieder geleert wird, sobald der Puffer beim Befüllen voll wird. Für den Fall, dass I damit gefüllt ist, werden die Daten wie oben beschrieben umorganisiert. Die Eingabepuffer der Links $1, 2, ..., i-1$ sind im Anschluss leer. Die Daten in den Puffern A_l für $l = 1, 2, ..., i-1$ sind auf Grund der Sortierung von σ stets sortiert. Dementsprechend ist die Invariante der Sortiertheit der Daten in den Links $1, 2, ..., i-1$ erfüllt. Die in S_{ic_i} eingefügten Elemente sind ebenfalls sortiert, da sie in derselben Reihenfolge eingefügt werden, wie sie in σ standen, das bereits sortiert war.

Die Anzahl der Elemente in der Datenstruktur wird durch den Aufruf von SWEEP nicht verändert, da alle Daten, die aus Puffern entnommen werden, in σ zwischen gespeichert und wieder eingefügt werden. Nach einem Aufruf von SWEEP ist I leer, sodass künftige

INSERT-Operationen erfolgreich durchgeführt werden können und I sortiert ist.

Es bleibt noch zu zeigen, dass die Puffer tatsächlich auch die ihnen zugeordnete Menge an Elementen aufnehmen können. Bei den Puffern auf dem Pfad p ist dies leicht ersichtlich: Es werden lediglich so viele Elemente eingefügt, wie auch vor der INSERT-Operation enthalten waren. Die restlichen Elemente werden in S_{ic_i} eingefügt. Dass dies möglich ist, wird bei näherer Betrachtung des Wachstums deutlich: In einem k-Merger und seinem Ausgabepuffer können insgesamt nur so viele Elemente existieren, wie alle seine Eingabepuffer fassen. Es ist in der hier vorgestellten Implementierung nicht vorgesehen, dass ein k-Merger-Eingabepuffer erneut befüllt wird, bevor er zuvor gänzlich geleert wurde. Anhand des oben geschilderten Wachstums (siehe Abschnitt 3.1.1) für $s_{i+1} = s_i(k_i + 1) = s_i k_i + s_i$ ist unmittelbar zu erkennen, dass in einem Puffer der Größe s_{i+1} sämtliche Elemente aus den Eingabepuffer des nächstkleineren k-Mergers gespeichert werden können $(s_i \cdot k_i)$. Das Hinzuaddieren eines weiteren s_i erlaubt rekursiv gefolgert, dass auch alle Elemente aus dem $i-1$-ten k-Merger aufgenommen werden können, der ja wiederum Platz für alle seine Vorgänger bietet, und so weiter. Der Puffer I wird bei s_2 berücksichtigt, wobei $s_2 = s_1 k_1 + s_1$ ist. Das hinzuaddierte s_1 bietet den benötigten Platz für I, da keine kleineren k-Merger existieren, die diesen Platz in Anspruch nehmen.

Formaler betrachtet lässt sich mittels Induktion, die wir hier leider nicht ausführen, zeigen, dass für das oben beschriebene Wachstum $s_i = s_1 + \sum_{j=1}^{i-1} k_j s_j$ gilt. Daraus folgt unmittelbar, dass alle Elemente aus den k-Mergern der Links $i = 1, 2, ..., i-1$ sowie der Eingabepuffer I in einem Puffer der Größe s_i Platz finden.

3.5 Komplexitätsanalyse

Im Folgenden betrachten wir die Cache-Effizienz der Datenstruktur.

Dazu verwenden wir für die k-Merger folgendes Lemma, das aus [2, Lemma 1] folgt:

▶ **Lemma 2.** *Unter der tall-cache-assumption ($B^2 \leq M$) benötigt der Aufruf von* FILL *für die Wurzel eines k-Mergers $O\left(k + \frac{k^3}{B} \log_M \left(k^3\right)\right)$ I/Os. Der Speicherbedarf eines k-Mergers ist ohne Betrachtung der Ein- und Ausgabepuffer in $O(k^2)$.*

Die Operation INSERT greift intern auf DELETEMIN (wenn SWEEP ausgeführt wird) zurück. DELETEMIN lässt die Elemente sortiert im Baum aufsteigen, was somit sowohl von INSERT als auch SWEEP benötigt wird. Daher betrachten wir zunächst die Komplexität im Bezug auf I/Os für dieses Aufsteigen in der Datenstruktur bzw. im Binärbaum, als der die Datenstruktur interpretiert werden kann. Abschließend werden wir die Operationen INSERT und DELETEMIN amortisiert analysieren.

Als erstes betrachten wir, welche Teile der Datenstruktur überhaupt I/Os verursachen und welche permanent im Cache gehalten werden können: Der Speicherbedarf eines Links i setzt sich zusammen, aus den Puffern des Mergers K_i, den Eingabepuffern und den Puffern A_i und B_i. Letztere sind im Speicherbedarf ${k_i}^3$. Für den Speicherbedarf der Eingangspuffer $S_{i1}, ..., S_{ik_i}$ zeigt sich, dass $\Theta(s_i k_i) = \Theta({k_i}^4)$: Der Speicherbedarf der Eingangspuffer betragt gemäß dem Aufbau der Datenstruktur $k_i \cdot s_i$. Aus Definition 1 und insbesondere der dort gegebenen Definition von $\lceil\!\lceil x \rceil\!\rceil$ folgt, dass ${s_i}^{1/3} \leq k_i < 2{s_i}^{1/3}$, was ingesamt zu $\Theta(s_i k_i) = \Theta({k_i}^4)$ führt. Mit Lemma 2 (Speicherbedarf von K_i in k_i^2) und k^3 für die beiden Puffer A_i und B_i folgt dementsprechend, dass die Größe eines einzelnen Links durch die Größe seiner Eingabepuffer dominiert wird.

Sei Δ_i die Summe des Speicherbedarfs der Links $1, ..., i$. Da aus Definition 1 auch ${s_i}^{4/3} \leq s_{i+1} < 3{s_i}^{4/3}$ folgt, wachsen k_i und s_i doppelt exponentiell.

Sei Δ_i die Summe des Speicherbedarfs der Links $1, ..., i$. Durch das doppelt exponentielle Wachstum von k_i und s_i wird Δ_i von der Größe des Links i dominiert, was bedeutet, dass $\Delta_i = \Theta({k_i}^4)$ ist. Wir bezeichnen i_M als das größte i, für das $\Delta_i \leq M$ gilt. Dies bedeutet dass alle Links $1, ..., i_M$ vollständig in den Cache passen. Wir gehen davon aus, dass diese Links dauerhaft im Cache gehalten werden. Damit müssen wir im Folgenden das Aufsteigen der Elemente lediglich bis zum Link i_M betrachten.

Für das Durchlaufen des k-Mergers K_i besagt das Lemma 2, dass jeder Aufruf des k-Mergers K_i $O\big(k_i + \frac{{k_i}^3}{B} \log_{M/B}\left({k_i}^3\right)\big)$ I/Os verursacht. Beachten wir hierbei nun, dass dabei jeweils ${k_i}^3$ Elemente ausgegeben werden und $k_i < 2{s_i}^{1/3}$ gilt, so führt dies zu $O\Big(\big(\frac{{k_i}^3}{B} \log_{M/B}\left({k_i}^3\right)\big)/{k_i}^3\Big) = O\big(\frac{1}{B} \log_{M/B}\left({k_i}^3\right)\big) = O\big(\frac{1}{B} \log_{M/B}\left(s_i\right)\big)$ I/Os für jedes ausgegebene Element. Dabei haben wir außer Acht gelassen, dass es vorkommen kann, dass B_i nicht gänzlich gefüllt wird, da keine Elemente mehr verfügbar sind, wodurch k_i hinfällig wird. Diesem Fall berücksichtigen wir am Ende der Analyse.

Da für jedes Element bisher lediglich der Weg bis zum Ausgabepuffer B_i des k-Mergers K_i ermittelt wurde, muss für den vollständigen Aufstieg durch den Baum noch der Weg über die Puffer A_j mit $j = i, i-1, ..., i_M$ betrachtet werden.

Im Folgenden greifen wir auf folgende mehrfache Anwendung der beiden zu Anfang eingeführten Ungleichungen zurück: $O(k_{i_M+1}) = O({s_{i_M+1}}^{1/3}) = O({s_{i_M}}^{4/9}) = O({k_{i_M}}^{4/3})$. Das Befüllen eines Puffers A_j verursacht $O(1 + |A_j|/B)$ I/Os. Für B gilt $B = O({k_{i_M+1}}^2) = O({k_{i_M}}^{8/9})$. Für $|A_j|$ gilt $|A_j| = {k_j}^3$, womit das Befüllen von A_j durch $|A_j|$ dominiert wird. Jedes Element aus einem Link $\geq j$ erzeugt also $O(1/B)$ I/Os für jeden Puffer A_j, wenn der Puffer vollständig gefüllt wird. Außerdem folgt aus der obigen Definition von Δi_M, dass $M < \Delta_{i_M+1}$ und somit $M = O({k_{i_M+1}}^4) = O({s_{i_M}}^{16/9})$ ist. Zusammen mit dem doppelt exponentiellen Wachstum von s_i erhalten wir für $i - i_M : i - i_M = O(\log\log_M(s_i)) = O(\log_M(s_i)) = O(\log_{M/B}(s_i))$. Dementsprechend dominiert das Aufsteigen innerhalb des k-Mergers K_i über das Aufsteigen innerhalb der Puffer A_j.

Wir halten also fest, dass wir für das Aufsteigen eines Elements im Baum vom Eingabepuffer S_{ic_i} bis A_1 $O\Big(\frac{1}{B} \log_{M/B}\left(s_i\right)\Big)$ I/Os benötigen!

Für die Analyse der vollständigen Datenstruktur ist das Traversieren der Elemente durch den Baum alleine jedoch nicht ausreichend. Im Folgenden betrachten wir die amort. Analyse von Insert bzw. dem daraus resultierenden Sweep-Aufruf. Wie bereits unter dem Abschnitt 3.4 „Korrektheit“ ausgeführt, werden bei einem solchen Aufruf von Sweep(i) maximal s_i Elemente in S_{ic_i} eingefügt, wobei alle Links $1, ..., i-1$ geleert werden. Bis zum nächsten Sweep(i)-Aufruf müssen also wieder s_i Elemente eingefügt worden sein. Wir können die Kosten für das spätere Aufsteigen innerhalb des Baums für die Elemente also auf die letzten s_i Einfügungen vor dem Aufruf von Sweep verteilen. Daher werden für das Einfügen eines Elements $O\big(\frac{1}{B} \log_{M/B}(s_i)\big)$ I/Os eingerechnet. Genauso können wir, ohne diese Berechnung zu verändern, die Kosten für die Umstrukturierung während einer Durchführung von Sweep $(\sigma, \sigma_1, \sigma_2)$ auf dieselben Einfügungen umverteilen.

Diese Umverteilung ohne ein Abändern der Beschränkung wird durch die Art der Umstrukturierung ermöglicht: Für σ_1 wird der Pfad p von A_1 bis S_{ic_i} linear abgelaufen. Dies entspricht einem linearen Speicherscan, der $O((\Delta_{i-1} + |A_i| + |B_i| + |K_i|)/B) = O({k_i}^3/B) = O(s_i/B)$ I/Os erfordert. Dabei werden aus Δ_{i-1} offensichtlich nicht alle Inhalte tatsächlich in σ_1 übernommen. Das Befüllen von σ_2 ruft für jedes Element DeleteMin auf. Dies resultiert im Aufsteigen der Elemente im Baum, was bereits bei den vorherigen s_i Insert-Operationen „bezahlt“ wurde. Somit bleibt noch das Befüllen und wieder Rausschreiben von σ, was letztlich ein einfaches Mergen von σ_1 und σ_2 und Einfügen in den Weg p darstellt. Der

Aufwand wird vom Pfad p dominiert, sodass auch hier wieder für alle Elemente zusammen $O(s_i/B)$ I/Os anfallen, pro Element also $O(1/B)$ I/Os.

Da wir bisher immer davon ausgegangen sind, dass wir beim Befüllen eines Puffers diesen gänzlich füllen, müssen wir noch betrachten, dass es auch vorkommen kann, dass ein Puffer nicht gänzlich gefüllt wird, da seine Eingabepuffer leer gelaufen sind. Dies kann jedoch zwischen zwei Aufrufen von Sweep(i) nur einmal für einen Puffer A_i bzw. B_i geschehen. Die Größe dieser Puffer ist $|A_i| = |B_i| = {k_i}^3 = \Theta(s_i)$. Daher reicht es aus, jedem Aufruf von Sweep weitere Kosten von $O\big(\frac{s_i}{B} \log_{M/B}(s_i)\big)$ I/Os zuzuordnen. Wie zuvor können wir diese Kosten auf die letzten s_i Einfügungen vor einem Aufruf von Sweep verteilen, womit für jede Einfügung zusätzliche $O\big(\frac{1}{B} \log_{M/B}(s_i)\big)$ I/Os fällig werden. Auch diese Umverteilung verändert die Beschränkung von Insert also nicht.

Betrachten wir nun abschließend eine Folge von Operationen auf einem zu Beginn leeren Funnel Heap, wobei n die Anzahl der Einfügungen ist. Dabei kann mit der Einfügung eines Elements höchstens ein Aufruf von Sweep(i) beglichen werden. Gemeinsam mit dem doppelt exponentiellen Wachstum von s_i fallen für eine Einfügung folgende I/Os an:

$$O\left(\sum_{k=0}^{\infty} \frac{1}{B} \log_{M/B}\left(n^{(3/4)^k}\right)\right) = O\left(\frac{1}{B} \log_{M/N}(n)\right)$$

Die Analyse von DeleteMin fällt schlicht aus: Amortisiert fallen für einen Aufruf von DeleteMin keine weiteren I/Os an, da alle notwendigen I/Os für das Aufsteigen im Baum bereits beim Einfügen beglichen wurden. Das Nachschlagen in I und A_1 erfordert unter der Annahme, dass diese permanent im Cache gehalten werden, keine weiteren I/Os.

4 Fazit

Der von Brodal und Fagerberg entwickelte *Funnel Heap* stellt eine effiziente Umsetzung eines Heaps dar, die insbesondere Cache-oblivious ist, d.h. unabhängig von den Cache-Parametern die minimal notwendigen Cache-Misses für sortierbasierte Datenstrukturen hervorruft. Die Datenstruktur ist dynamisch in ihrer Größe und verursacht gemäß der amort. Analyse für einen Insert-Aufruf $O\big(\frac{1}{B} \log_{M/N}(n)\big)$ I/Os. Für DeleteMin fallen in der amort. Analyse keine weiteren I/Os an.

Literatur

1 Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.

2 Gerth Stølting Brodal and Rolf Fagerberg. *Cache Oblivious Distribution Sweeping*, pages 426–438. Springer-Verlag, 2002.

3 Gerth Stølting Brodal and Rolf Fagerberg. Funnel heap - a cache oblivious priority queue. In *Proceedings of the 13th International Symposium on Algorithms and Computation*, pages 219–228. Springer-Verlag, 2002.

4 Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Addison-Wesley Publishing Company, 2nd edition, 2010.

5 Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. *ACM Trans. Algorithms*, 8(1), 2012.

6 Christos H. Papadimitriou. Computational complexity. In *Encyclopedia of Computer Science*, pages 260–265. John Wiley and Sons Ltd., 2003.

Der Buffer-Tree für externe Datenstrukturen

Roland Christian Wyzgol[1]

1 **TU Dortmund**
`roland.wyzgol@tu-dortmund.de`
`Matrikelnummer: 130917`

Zusammenfassung

In dieser Ausarbeitung wird eine Variante einer Baumstruktur vorgestellt, welche entworfen wurde, um die Entwicklung von Input/Output effizienten Algorithmen unter Nutzung des externen Hauptspeichers zu unterstützen: Der *Buffer Tree*. Dabei handelt es sich um einen (a, b)-Baum, bei welchem der Fan-Out und die Höhe von den Parametern des Hauptspeichers (Anzahl der Blöcke) abhängen. Das Besondere des Buffer-Tree ist, dass jeder Knoten einen *Puffer* erhält, der Anfragen aufnimmt bevor sie verarbeitet werden. Damit wird das Ziel verfolgt, den internen Speicher des Prozessors zu entlasten und bestimmte Aufgaben in den Hauptspeicher auszulagern. Im Rahmen dieser Ausarbeitung wird die Idee für die Struktur des Baumes sowie das Vorgehen zur Lösung verschiedener Probleme erklärt. Es wird dargelegt, wie der Buffer-Tree mit Hilfe der Operationen „Einfügen“ und „Löschen“ aufgebaut wird und zum Sortieren verwendet werden kann. Darauf aufbauend wird veranschaulicht, wie der Buffer Tree durch die Erweiterung um eine Funktion „deletemin“ als Prioritätswarteschlange eingesetzt werden kann. Für die Varianten wird die Laufzeit angegeben und gezeigt, dass sich diese im Rahmen der bereits existierenden Lösungen bewegt.

1998 ACM Subject Classification E.1 Data Structures

Keywords and phrases buffer-tree, a,b-tree variant, I/O, external memory

Digital Object Identifier 10.4230/LIPIcs.BADS.2017.13

1 Ausgangssituation

Die grundlegende Ursache, die zu der Entwicklung des Buffer-Tree geführt hat, liegt in einem Problem, das seit den späten 1970er Jahren diskutiert wird und seinen Ursprung in der Struktur der Von-Neumann-Architektur hat. Das Problem, welches auch als „Von-Neumann-Flaschenhals“ [2] bekannt ist, liegt in der unterschiedlichen Datentransferrate des Prozessors und dem verwendeten Speicher. Durch die in den vergangenen Jahrzehnten stetig gewachsenen Datenmengen und deren Verarbeitungsgeschwindigkeit, haben Input/Output effiziente Algorithmen an Relevanz gewonnen. Die Entwicklung, Prozessoren mit mehreren Kernen auszustatten, verschärft dieses Problem nur. Ein Ansatz, der häufig verfolgt wird, ist die „Neu-Entwicklung“ von Datenstrukturen für externe Speicher, welche ursprünglich für interne Speicher entwickelt wurden. In dieser Ausarbeitung wird ausgehend von [1] ein weiterer Ansatz vorgestellt, der wie bereits erwähnt, eine neue Datenstruktur zu Grunde legt. Die Grundlage dieses Ansatzes ist das Standard I/O Modell von Aggawal und Vitter [1]. Teil dieses Modells sind die folgenden Parameter:

- N: Anzahl der Elemente in der Probleminstanz
- M: Anzahl der Elemente, die im Hauptspeicher Platz finden
- B: Anzahl der Elemente pro Block

Eine „I/O-Operation“ ist nun das Austauschen (englisch: „swap“ [1]) von B Elementen (eines Blockes) des internen Speichers mit B aufeinanderfolgenden Elementen des externen Speichers.

2nd Symposium on Breakthroughs in Advanced Data Structures.
Editors: Johannes Fischer; Article No. 13; pp. 13:1–13:10

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Damit wird auch eine Möglichkeit gegeben die Leistungsfähigkeit von Datenstrukturen zu bewerten: Diese können anhand der Anzahl der benötigten Operationen zur Lösung eines Problems bewertet werden. Mit Bezug zu einer Probleminstanz können aus den obigen Variablen zwei Kenngrößen abgeleitet werden, die für die folgende Betrachtung besonders interessant sind:

- Anzahl der Blöcke einer Probleminstanz: $n = \frac{N}{B}$
- Anzahl der Blöcke, die in den internen Speicher passen: $m = \frac{M}{B}$

Benötigt ein Algorithmus $O(n)$ solcher Operationen zur Lösung eines Problems, ist die Anzahl der I/O Operationen linear.

2 Der Buffer-Tree

In diesem Kapitel wird der Buffer-Tree und die dafür verwendete Puffer-Technik [1] vorgestellt. Im weiteren Verlauf werden das Sortieren und die Verwendung des Buffer-Tree als Prioritätswarteschlange anhand von Beispielen erläutert.
Beim Buffer-Tree handelt es sich, wie eingangs erwähnt wurde, um einen (a, b)-Baum. Ein (a, b)-Baum ist ein spezieller Binärbaum, welcher bestimmten Einschänkungen unterliegt und damit auch zuverlässig bestimmte Eigenschaften erfüllt [4]. So müssen sich alle Blätter des Baumes auf einer Ebene befinden. Außerdem gilt, dass alle internen Knoten zwischen a und b Kinder haben (auch: Der *Fan-Out* liegt zwischen a und b). Für die Wurzel gilt, dass sie zwischen 2 und b Kinder besitzt. Des Weiteren erfüllen a und b die Bedingung, dass sie natürliche Zahlen sind und die Eigenschaft $2 \leq a \leq \frac{b+1}{2}$ erfüllen. Anhand von Abbildung 1 wird ein einfacher $(2, 3)$-Baum erklärt. Aus den obigen Regeln folgt, dass in diesem Beispiel jeder Knoten mindestens 2 und höchstens 3 Kinder haben darf. In den Blättern werden die Schlüssel in sortierter Reihenfolge gespeichert. In den internen Knoten werden Hilfschlüssel abgelegt, damit innerhalb des Knotens navigiert werden kann. Eine Suche nach dem Element 14 würde anhand der Hilfschlüssel in der Wurzel ($7 < 14 < 15$) in den mittleren Knoten der ersten Ebene führen. Die darauf folgende Auswertung ($11 < 14$) würde in den rechten Knoten führen. Schließlich würde festgestellt werden, dass das Element 14 nicht im Baum enthalten ist ($13 \neq 14$). Der Vorteil dieser Baumstruktur ist, dass der Baum bei großen Werten für a und b eine Vielzahl an Schlüssel enthalten kann, dabei aber nur von geringer Tiefe ist. Zur Veranschaulichung: Ein Baum mit $a = 400$ kann bei einer Tiefe von 3 bereits $400^3 = 64.000.000$ Schlüssel speichern.

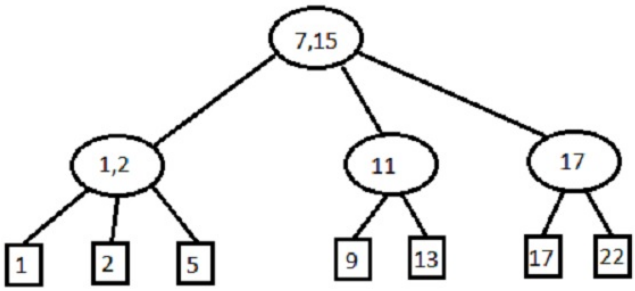


Abbildung 1 Ein einfacher (2,3)-Baum

Diese Eigenschaft wird im Fall des Buffer-Tree ausgenutzt. Hierfür werden a und b anhand der Anzahl der Blöcke, die in den internen Speicher passen, gewählt: $a = \frac{m}{4}$ und $b = m$. Abbildung 2 zeigt den Aufbau im Detail. Der Buffer-Tree ist unterteilt in Wurzel, interne Knoten, Blattknoten und Blätter. Alle Knoten haben einen Fan-Out zwischen $\frac{1}{4} \cdot m$ und m. In jedem Blatt können B Elemente gespeichert werden. Dies entspricht der Anzahl der Elemente in jedem Block. Ein so konstruierter Baum hat eine Tiefe von $O(\log_m n)$. Durch den hohen Fan-out wird die Grundlage für effiziente Operationen auf der Datenstruktur gelegt [1]. Außerdem ist der benötigte Speicherplatz dieser Datenstruktur konstant, da die Größe des Baumes von den Hardwarestrukturen abhängt und somit keiner laufzeitbedingten Variation unterliegt.

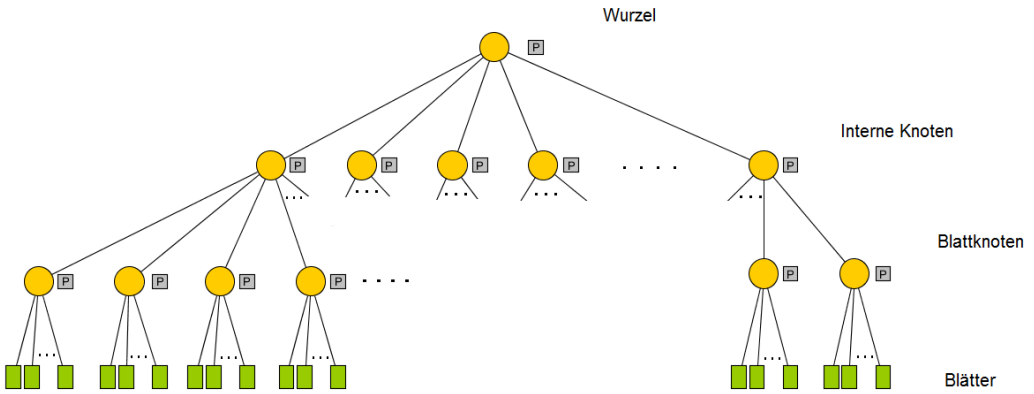


Abbildung 2 Der Aufbau des Buffer-Tree (P = Puffer) (Nach [1], erstellt mit yEd Graph Editor Version 3.17 (www.yworks.com))

Die Puffer-Technik beschreibt nun, wie auf der Datenstruktur mit Daten gearbeitet wird. Die grundlegende Idee ist zum einen, mit einem Puffer (siehe P in Abbildung 2) zu arbeiten. Jedem Knoten des Buffer-Tree wird ein Puffer zugewiesen. Dieser Puffer hat die Größe m, entspricht also der Anzahl der Blöcke im internen Speicher. Zum anderen werden Operationen (zum Beispiel Einfügen und Löschen) nach dem Prinzip der *Lazy Evaluation* ausgewertet. Anstatt eine Operation direkt abzuarbeiten, werden B Operationen (ein Block) im internen Speicher gesammelt und anschließend in den Puffer der Wurzel eingefügt. Ist dieser Puffer voll, werden die Elemente des Puffers eine Ebene tiefer in die Puffer der nächsten Knoten einsortiert. Dieser Vorgang wird *Pufferentleerungsprozess* (nach [1]: *buffer emptying process*) genannt. Das Einfügen eines Elements in den Baum geschieht demnach auf folgende Weise: Im ersten Schritt wird aus dem Element, welches eingefügt werden soll, aus einem Zeitstempel und einer Information darüber, ob es sich über einen Einfüge- oder Löschvorgang handelt, ein neues Element erzeugt. Es wird also nicht nur ein Wert übergeben, sondern ein gesamter Befehl. Auf diese Weise erzeugte Elemente könnten zum Beispiel so aussehen:

- (insert, 8, 10:22:54)
- (delete, 4, 08:03:12)

Diese Befehle werde im internen Speicher gesammelt und der Wurzel des Buffer-Tree übergeben, sobald B dieser Elemente zusammengekommen sind. Das Prinzip der *Lazy Evaluation* hat für die Elemente im Buffer-Tree zwei Konsequenzen. Dadurch, dass die Lösch- und Einfüge-Operationen nicht unmittelbar ausgewertet, sondern gepuffert werden, können

zeitgleich mehrere dieser Operationen für dasselbe Element in Puffer oder Baum vorkommen. Aus diesem Grund werden Elemente bzw. Befehle beim Einfügen in die Wurzel mit einem Zeitstempel versehen. So können sie chronologisch abgearbeitet werden. Außerdem führt dieses Vorgehen dazu, dass Operationen gestapelt (nach [1]: „batched“) und damit zeitverzögert ausgeführt werden. Es dauert mehrere Pufferentleerungsprozesse bis die eingefügten Befehle in den Blättern ankommen. Dieser Prozess wird von einem Knoten angestoßen, dessen Puffer voll ist und wird dann nach unten weiter für den Teilbaum ausgeführt. Im Detail läuft der Prozess wie folgt ab [1]:

- Es werden die ersten m Elemente des Puffers in den internen Speicher geladen, sortiert und mit den restlichen Elementen des Puffers zusammengeführt (nach [1]: „merge“). Dabei werden Operationen, die sich gegenseitig aufheben (Lösch- und Einfüge-Operationen des gleichen Elements), entsprechend ihrer Zeitstempel entfernt.
- Anschließend wird die sortierte Liste der Elemente schrittweise auf die unteren Puffer verteilt. Dabei gilt, dass höchstens ein Block eines Puffers nicht voll sein darf.
- Enthalten nun weitere Knoten mehr als m Elemente im Puffer und handelt es sich bei diesen Knoten um interne Knoten, wird für diese Knoten ebenfalls der Pufferentleerungsprozess angestoßen.

Nachdem dieser Prozess für alle internen Knoten durchgeführt wurde, werden die Puffer der Blattknoten entleert. Beginnend mit einem Blattknoten mit vollem Puffer (mehr als m Elemente) wird folgende Prozedur durchgeführt, bis kein voller Blattknoten mehr existiert.

- Im ersten Schritt wird, wie bei den internen Knoten, der Puffer sortiert und Operationen, die sich gegenseitig aufheben (Lösch- und Einfüge-Operation des gleichen Elements), werden zusammengeführt.
- Im zweiten Schritt wird diese sortierte Liste mit der Liste der Elemente aus den Blättern des Knotens zusammengeführt. Hierbei werden ebenfalls diejenigen Operationen entfernt, die sich gegenseitig aufheben.
- Wenn die Anzahl der Elemente in dieser Liste nun kleiner als k (Anzahl der Blätter des Knotens) ist, wird folgendes getan:
  a) Füge die Elemente in sortierter Reihenfolge in die Blätter des Knotens ein.
  b) Hänge „Dummy-Blätter“ (leere Blätter) an den Knoten an, bis der Knoten k Blätter hat.
- Sonst (Anzahl der Elemente in der Liste ist größer als k):
  a) Füge die k kleinsten Elemente aus der Liste in die Blätter des Knotens ein.
  b) Füge solange die restlichen Elemente ein und balanciere den Baum aus, bis alle Elemente eingefügt wurden.

Im Anschluss an diesen Vorgang werden schrittweise die „Dummy-Blätter“ gelöscht und der Baum jeweils ausbalanciert. Bevor ein Knoten ausbalanciert wird, wird für die benachbarten Knoten eine Pufferentleerung durchgeführt. Wird der Puffer eines Blattes durch diese Pufferentleerung gefüllt, wird er geleert, bevor das nächste „Dummy-Blätter“ entfernt wird.
Die Eigenschaft der Puffer, dass sie im Zuge von Entleerungsprozessen mehr als m Elemente bearbeiten können, sorgt dafür, dass es nur bei der Entleerung von Blattknoten zu einem Ausbalancieren kommen kann. Das Ausbalancieren funktioniert für den Buffer-Tree wie für einen (a,b)-Baum: Sollte ein Blattknoten nach dem Einfügen ausbalanciert werden müssen (der Blattknoten hat zu viele Blätter), werden einer oder mehrere sogenannte *Splits* [1] durchgefürt, bis die Anzahl der Blätter wieder zu den Vorgaben des Buffer-Tree passt. Sollte ein Blatt nach einer Lösch-Operation leer sein, wird das Ausbalancieren durch das Zusammenführen von Blattknoten und gegebenfalls dem Aufteilen von Blättern durchgeführt. In [1] werden Algorithmen für das Ausbalancieren vorgestellt.

2.1 Sortieren

Damit der Buffer-Tree nun zum Sortieren verwendet werden kann, wird eine weitere Operation benötigt (zum Beispiel „empty/write“ [1]). Diese gibt die enthaltenen Elemente in sortierter Reihenfolge aus. Dies ist, nach obiger Beschreibung, keine unlösbare Aufgabe, da die Elemente in den Puffern des Buffer-Tree bereits sortiert sind. Daher müssen nur mehrere Pufferentleerungsprozesse in bestimmter Reihenfolge angestoßen werden. Diese Reihenfolge entspricht der Breitensuche (auch „BFS“ für „breadth-first search“, [1]). Bei der Breitensuche werden zuerst alle Knoten einer Ebene durchsucht, bevor ihre Kindsknoten besucht werden. Im Fall des Buffer-Tree wird zuerst ein Entleerungsprozess in der Wurzel angestoßen. Anschließend im am weitesten links liegenden Kindsknoten, dann der Knoten rechts davon und so weiter. Auf diese Weise steht am Ende des Prozesses im linken Blatt des Buffer-Tree das kleinste Element an erster Stelle. Das besondere an diesem Vorgehen ist, dass nicht alle Elemente zu Beginn bekannt sein müssen. Die Elemente können an belieber Stelle im Buffer-Tree liegen und gegebenenfalls durch Löschoperationen im Laufe des Algorithmus entfernt werden. Im Folgenden wird dieses Vorgehen für einen einfachen Buffer-Tree gezeigt.

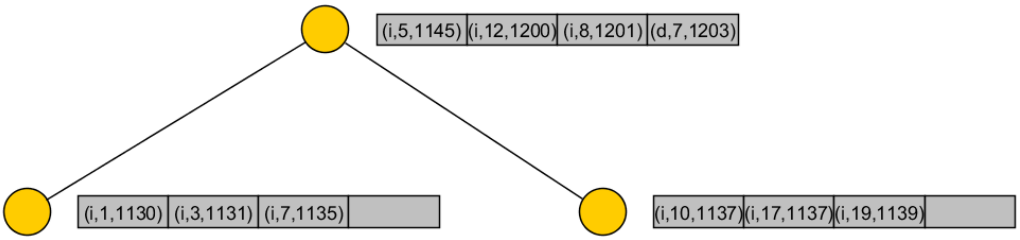


Abbildung 3 Der Buffer-Tree in der Ausgangssituation vor dem Sortieren. Die Elemente wurden vorher durch Einfüge- und Löschoperationen in den Baum geladen. (Erstellt mit yEd Graph Editor Version 3.17 (www.yworks.com).)

Dem Buffer-Tree werden $M = 8$ (Anzahl der Elemente im internen Speicher) und $B = 2$ zu Grunde gelegt. Das führt zu einem möglichen Fan-Out m zwischen 1 und 4 (zur Erinnerung: $m = \frac{M}{B}$). Es wird angenommen, dass der Buffer-Tree bereits mit Daten gefüllt ist (siehe Abbildung 3). Der dargestellte Baum enthält keine Blätter, da diese noch nicht angelegt wurden. Blätter werden erst angelegt, wenn der Puffer eines Blattknotens geleert wird. Die die Notation der Elemente (o, x, y) ist wie folgt zu verstehen: o steht für Operation (insert, delete), x steht für den Wert des eigentlichen Elements und y ist der Zeitstempel. Für den Zeitstempel wurde eine kurze Schreibweise der Uhrzeit ohne Sekunden gewählt. Nun wird schrittweise vorgegangen, um die Einträge in sortierter Reihenfolge auszugeben:

- Alle Einfüge- und Löschoperationen wurden dem Puffer der Wurzel übergeben. Nun wird die *empty* Operation aufgerufen.
- Beginnend in der Wurzel wird ein Pufferentleerungsprozess angestoßen. Als erstes werden die Elemente anhand ihrer Werte im Puffer der Wurzel sortiert (siehe Abbildung 4, Schritt 1).
- Nun werden die Elemente des Puffers auf die Puffer der darunter liegenden Knoten verteilt (siehe Abbildung 4 Schritt 2). Dabei wird immer darauf geachtet, dass höchstens ein Block eines Puffers leer bleibt. Sollte ein Puffer dabei voll werden, werden diese Elemente an den Puffer angehängt und im nächsten Pufferentleerungsprozess verarbeitet. Die Verteilung läuft so ab, dass jeder der Wert des kleinsten Elements in der Liste mit den

Elementen im Puffer der darunterliegenden Ebene verglichen und an der geeigneten Stelle eingefügt werden. Das erste Element ist (i,5,1145) und wird nach (i,3,1131) eingefügt. Das Element (i,7,1135) wird ein Feld nach hinten geschoben. Nun steht mit (d,7,1203) eine Löschoperation an. Da der Zeitstempel dieses Elements aktueller ist, werden (i,7,1135) und (d,7,1203) aus dem Puffer entfernt. Das nächste Element (i,8,1201) wird an den Puffer angehängt. Dieser ist nun voll und es wird überprüft, ob im nächsten Puffer noch Platz ist. Dies ist der Fall und das Element (i,12,1200) wird nach (i,10,1137) einsortiert (siehe Abbildung 4, Schritt 3).

- Dem Muster der Breitensuche folgend wird zuerst im linken Knoten ein Pufferentleerungsprozess angestoßen. Da es sich bei diesem Knoten um einen Blattknoten handelt, werden die Elemente des Puffers, beginnend mit dem Linken, in die Blätter einsortiert (siehe Abbildung 4 Schritt 4). Für den rechten Knoten wird ebenso verfahren. Da die Größe der Blätter durch B festgelegt wird, werden pro Blatt zwei Elemente gespeichert.
- Abschließend stehen in den Blättern des Buffer-Tree die übergebenen Elemente in sortierter Reihenfolge (siehe Abbildung 4 Schritt 4) und werden durch die *empty* Operation ausgegeben. An dieser Stelle ist noch zu beachten, dass jeweils zwei Dummy-Blätter erstellt werden (siehe oben). Diese werden im darauffolgenden Pufferentleerungsprozess für den benachbarten Knoten allerdings wieder entfernt (siehe Abbildung 5).

Die Kosten zum Sortieren einer Menge von N Elementen sind (nach [1], "Corollary 1") $O(n \cdot \log_m n)$ I/O Operationen. Damit reiht sich das Sortieren mit dem Buffer-Tree bezüglich der Laufzeit in die Reihe vieler Sortieralgorithmen ein. Viele haben im *Average-Case* eine Laufzeit von $O(N \cdot \log N)$ (Quicksort, Heapsort) [4]. Es handelt sich um einen I/O effizienten Algorithmus [1].

2.2 Gepufferte Prioritätswarteschlange

In diesem Abschnitt wird dargelegt, wie der Buffer-Tree als Prioritätswarteschlange verwendet werden kann. Im Allgemeinen kann ein klassischer Suchbaum als Prioritätswarteschlange eingesetzt werden, da das kleinste Element immer im am weitesten links liegenden Blatt gefunden wird. Dies gilt für den Buffer-Tree nicht zwangsläufig, da das Element sich auch in einem Puffer zwischen Wurzel und dem am weitesten links gelegenen Blattknoten befinden kann. Im Folgenden wird ein Ansatz für eine „deletemin“ Operation gegeben, die das kleinste Elemente aus dem Buffer-Tree ausgibt und entfernt. Dafür wird als erstes ein Pufferentleerungsprozess für die Knoten angestoßen, die auf dem Pfad von der Wurzel zum am weitesten links liegenden Blatt liegen (siehe Abbildung 6). Nun werden die kleinsten $\frac{1}{4} \cdot m \cdot B$ Elemente gelöscht. (Zur Erinnerung: Ein Blattknoten hat $\frac{1}{4} \cdot m$ Blätter und jedes Blatt hat B Elemente.) Dadurch landen sie in den Blättern des am weitesten links gelegenen Blattknoten und werden in den internen Speicher geladen. Damit können die nächsten $\frac{1}{4} \cdot m \cdot B$ „deletemin“ Operationen der Warteschlange direkt aus dem internen Speicher heraus ausgeführt werden. Im Hintergrund werden die kleinsten Elemente im Buffer-Tree aktualisiert (Ausbalancieren durch Aufsplitten der Knoten), da die Blätter des am weitesten links gelegenen Blattknoten leer sind. Das folgende Beispiel soll diesen Prozess veranschaulichen.

Sei für das Beispiel ein Buffer-Tree mit beliebigem m und n gegeben (siehe Abbildung 6) und die Werte x der Elemente (o, x, y) entsprechen der Priorität. Das zuletzt eingefügte Elemente $(i, 1, 1315)$ ist im Puffer der Wurzel und hat die höchste Priorität. Ein Aufruf der *deletemin* Operation hat den folgenden Effekt:

- Für die Wurzel wird eine Pufferentleerung angestoßen. Dabei wird das Element an der ersten Stelle des Puffers einsortiert und anschließend an den linken Knoten unter der Wurzel übergeben. Dort wird er wieder an der erste Stelle des Puffers einsortiert.

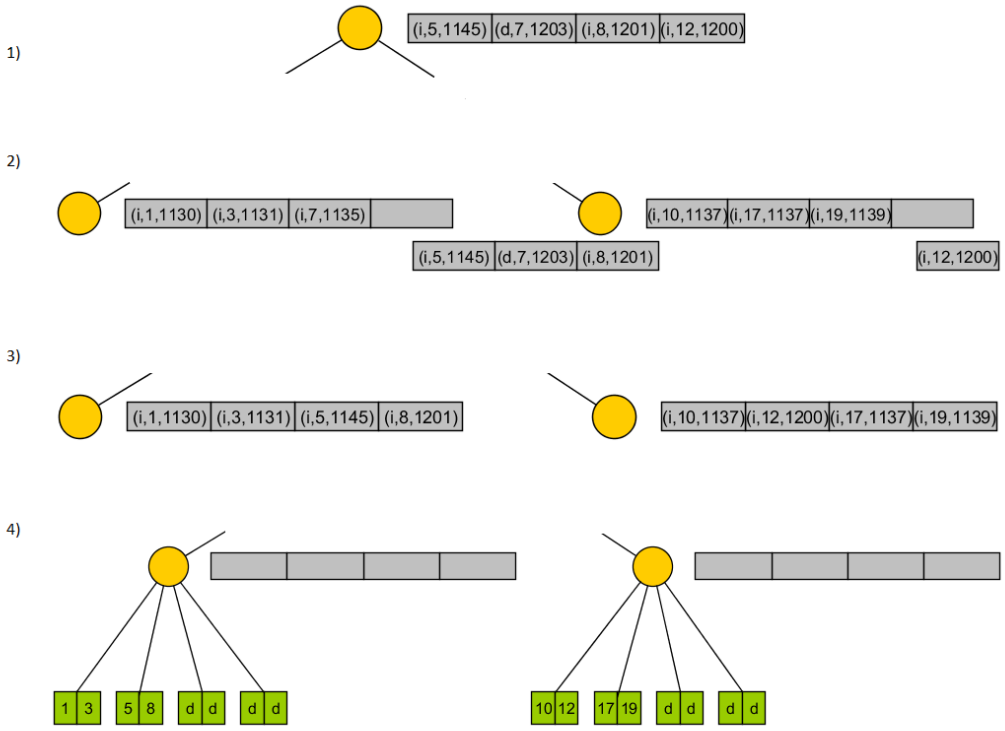


Abbildung 4 Der Sortiervorgang im Buffer-Tree in vier Schritte aufgeteilt. 1) Sortieren 2) Verteilung auf Kindsknoten 3) Knoten nach dem Zusammenführen 4) Pufferentleerung der Knoten (Erstellt mit yEd Graph Editor Version 3.17 (www.yworks.com).)

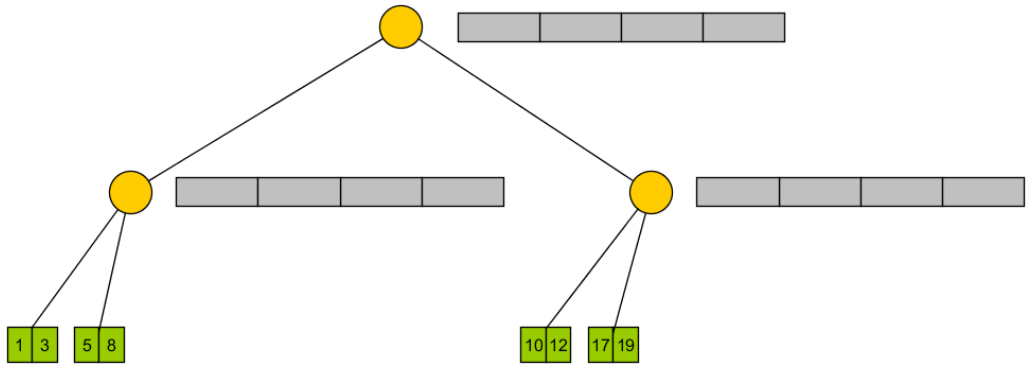


Abbildung 5 Der Endzustand des Buffer-Tree, nachdem die Dummy-Blätter entfernt wurden. (Erstellt mit yEd Graph Editor Version 3.17 (www.yworks.com).)

- Nun wird die Pufferentleerung für diesen Knoten angestoßen und das Element wird an den darunter liegenden linken Knoten übergeben und an erster Stelle des Puffers einsortiert. Dieser Prozess wird bis zum Blattknoten wiederholt.
- Das Element mit der höchsten Priortität wurde auf diese Weise von der Wurzel in das am weitesten links liegenden Blatt bewegt. Darüber hinaus wurden bei diesem Vorgehen alle Elemente mit vergleichbar hoher Priorität (z.B. $(i, 4, 1530)$) entlang dieses Pfades verschoben. Diese Elemente befinden sich ebenfalls in den Blättern.
- Im nächsten Schritt werden nun die $\frac{1}{4} \cdot m \cdot B$ kleinsten Elemente des Buffer-Tree gelöscht und im internen Speicher behalten. Damit sind alle Elemente des linken Blattknotens im internen Speicher und können ohne I/O Operation abgerufen werden.

Die Kosten der *deletemin* Operation belaufen sich auf $O(m \cdot \log_m n)$ [1]. Der Unterschied zum Sortieren ist, dass nicht alle Puffer geleert werden müssen, sondern nur einer auf jeder Ebene des Baumes.

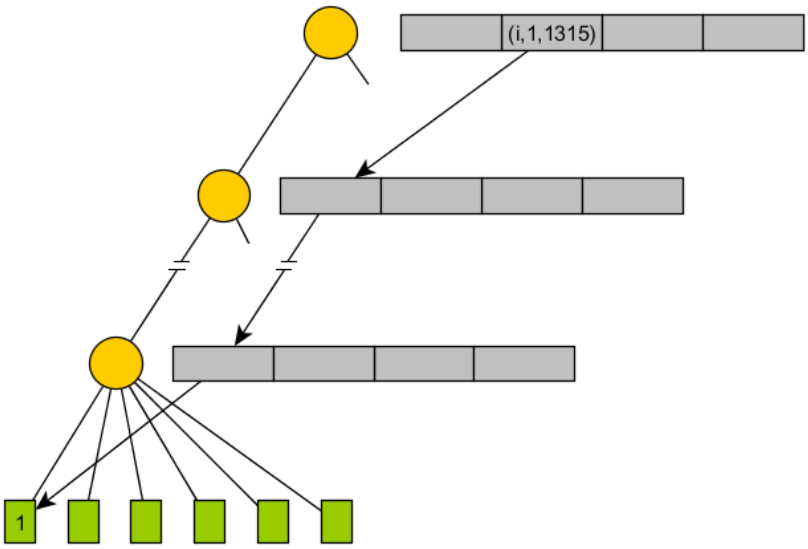


Abbildung 6 Darstellung des Pfades eines kleinsten Elements beim Aufruf der *deletemin* Operation. (Erstellt mit yEd Graph Editor Version 3.17 (www.yworks.com).)

2.3 Gepufferter Bereichsbaum

Ein Bereichsbaum ist ein nicht ausbalancierter Suchbaum, in welchem die Einträge nach Bereichen eingeordnet werden. Eine Bereichssuche auf einem solchen Baum gibt alle Elemente aus, die im Anfrageintervall $[x_1, x_2]$ liegen. Im Falle eines Suchbaums ist der normale Ablauf wie folgt: Es werden die Werte x_1 und x_2 gesucht, um anschließend alle Elemente der Blätter zwischen den Blättern von x_1 und x_2 auszugeben. Ein alternatives Vorgehen ist, die Elemente schon während der Suche nach x_1 und x_2 auszugeben. Dieser Ansatz wird auch für den Buffer-Tree verfolgt [1].

Dafür wird als erstes ein Suchelement in den Puffer der Wurzel eingefügt. Darin enthalten ist das Intervall $[x_1, x_2]$ sowie ein Zeitstempel. Für die Bereichssuche mit dem Suchelement wird ein modifizierter Pufferentleerungsprozess angestoßen [1]. Bevor dieser durchgeführt wird, wird für einen internen Knoten v überprüft, ob eine Suche nach den Elementen

des Intervalls im selben Unterbaum aller Unterbäume des Knotens v landet. Es wird also überprüft, ob x_1 und x_2 im selben Unterbaum des Knotens liegen. Gilt das, wird das Suchelement in den Knoten v eingefügt. Andernfalls wird das Suchelement in die Elemente x_1 und x_2 aufgeteilt und alle Unterbäume, deren Elemente *alle* im Intervall $[x_1, x_2]$ liegen werden ausgegeben. Die Elemente werden fortan wie Einfüge-/Löschelemente behandelt und während des Pufferentleerungsprozesses im Baum nach unten geschoben. Für eine detaillierte Beschreibung des gepufferten Bereichsbaums sei an dieser Stelle auf die Beschreibung in [1] verwiesen.

2.4 Gepufferter Segmentbaum

Die letzte Variante des Buffer-Tree, die in diesem Rahmen vorgstellt wird, ist der gepufferte Segmentbaum. Ein klassischer Segmentbaum ist mit einem Bereichsbaum vergleichbar. Es handelt sich ebenfalls um einen binären Suchbaum, in dem Intervalle gespeichert werden. Den wesentlichen Unterschied stellt allerdings die Anfrage an den Baum dar. Während einem Bereichsbaum ein Intervall übergeben wird, wird einem Segment für eine Anfrage ein Punkt übergeben. Die gewünschte Antwort ist die Beantwortung der Frage, in welchen Segmenten der Punkt q enthalten ist. Daraus ergeben sich auch die unterschiedlichen Anwendungsfälle. Ein klassischer Segmentbaum ist statisch aus einer sortierten Menge von Elementen aufgebaut. Ein Segment ist ein Intervall, dessen Eckpunkte (Anfangs- und Endpunkt der Intervallmenge) die Blätter des Baumes bilden. Die Informationen zu den Segmenten können in bis zu zwei Knoten gespeichert werden. Zum einen in den Knoten, deren Blätter die Eckpunkte des Intervalls enthalten und zum anderen in dem Knoten, dessen Unterbaum das komplette Segment enthält. Zur Beantwortung einer Anfrage nach einem Punkt q wird im Baum von oben nach q gesucht und alle Segmente, die dabei in den besuchten Knoten gefunden werden, werden ausgegeben. Da eine ausführliche Beschreibung den Rahmen dieser Ausarbeitung sprengen würde, sei ebenfalls auf die Beschreibung in [1] verwiesen.

3 Anwendungsfall für gepufferte Prioritätswarteschlange

Zum Abschluss wird nun ein Anwendungsbeispiel für die gepufferte Prioritätswarteschlange gegeben. Das „Time-Forward Processing“ wird zur Bewertung von Schaltungen verwendet [3]. Gegeben ist dabei ein verbundener, gerichteter, azyklischer Graph G mit N Kanten und einem Eingangsgrad $v_d \leq \frac{M}{2}$ für jeden Knoten v. Gesucht werden die Werte $f_v : R^{v_d} \to R$ für alle Senkknoten. Senkknoten haben einen Ausgangsgrad von 0.
Die Idee, die zur Lösung verfolgt wird, ist auszunutzen, dass die Knoten im Graphen G topologisch sortiert sind und eine totale Ordnung haben. Das problematische an der Auswertung ist, dass zur Auswertung der Funktion f_v an einem bestimmten Knoten die Werte der Inputs dieses Knotens im internen Speicher sein müssen. Stellt man sich vor, dass Knoten v zum „Zeitpunkt v“ ausgewertet wird, wird der Ausdruck „time forward“ daher motiviert, dass die Ergebnisse bei Auswertung in topologischer Ordnung der Reihe nach über die Kanten „in der Zeit vorwärts“ verschickt werden, damit diese zur weiteren Auswertung verfügbar sind.
Für die Lösung dieses Problems existieren Algorithmen [3] mit verschiedenen Einschränkungen. Nach [1] ist der Hauptkritikpunkt demnach, dass beide eine Einschränkung auf m haben. Dieses darf eine bestimmte Größe nicht über- oder unterschreiten. Diese Einschränkung soll mit der gepufferten Prioritätswarteschlange umgangen werden. Der Ablauf ist dabei folgender: Anfangs ist die Prioritätswarteschlange leer. Wird nun Knoten u berechnet, enthält sie ein Element für jede Kante in der Form (v, u). Die zeitliche Priorität der Elemente steckt in den

Knoten: v hat eine höhere Priorität als u. Damit können für alle direkten Vorfahren u_d die Ergebnisse durch deletemin-Operationen bestimmt werden. Schließlich werden anhand der Liste der Kanten die weiteren Elemente (zum Beispiel mit der Priorität w die Elemente (u, w)) eingefügt. Die Laufzeit dieser Lösung wird nach [1] insgesamt mit $O(n \cdot \log_m n)$ angegeben. Im Vergleich zu den vorhandenen Lösungen ergibt sich daraus zwar kein direkter Vorteil in der Laufzeit, allerdings können die Einschränkungen auf m umgangen werden.

4 Fazit

In dieser Ausarbeitung wurde ein Ansatz zur Strukturierung von externen Daten vorgestellt. Das verfolgte Ziel war, Datenmengen auf einen Speicher ablegen zu können, der nicht zum internen Kreis (RAM, L1-L2-Cache) des Prozessors gehört. Diese externen Datenträger zeichnen sich durch ihre Größe aus, erlauben aber nur eine geringere Übertragungsrate. Aus diesem Grund war es wichtig, dass mindestens die Laufzeiten von bestehenden Algorithmen eingehalten werden. Der Buffer-Tree liefert diese Eigenschaft. Die Lösungen für Datenstrukturen und die dazugehörigen Algorithmen zeichnen sich durch ihre Einfachheit aus. Insbesondere die Verfahren zum Sortieren und Priorisieren von Daten funktionieren auf dem Buffer-Tree intuitiv. Darüber hinaus kann der Buffer-Tree auch für komplexere Probleme verwendet werden, ohne dabei übermäßig komplexe Algorithmen erforderlich zu machen (siehe [1] zum gepufferten Bereichs- und Segmentbaum).
Gleichzeitig bleibt anzumerken, dass die Datenstruktur auch ihre Schwächen hat. Das Prinzip der *Lazy Evaluation* kann zu Problemen bei teilweise geordneten Mengen führen [1] und alle Anfragen werden gestapelt ausgeführt. Es gibt also immer eine Zeitverzögerung bei der Beantwortung von Anfragen. Das bedeutet, dass der Buffer-Tree als Datenstruktur für Echtzeitsysteme mit strengen Vorgaben hinsichtlich der Bearbeitungszeit von Anfragen nicht in Frage kommt. Insgesamt stellen die Varianten des Buffer-Tree brauchbare Lösungen zu bekannten Problemen bereit.

Literatur

1 Lars Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003. URL: https://doi.org/10.1007/s00453-003-1021-x, doi:10.1007/s00453-003-1021-x.

2 John Backus. Can programming be liberated from the von neumann style?: A functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, August 1978. URL: http://doi.acm.org/10.1145/359576.359579, doi:10.1145/359576.359579.

3 Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory graph algorithms. pages 139–149, 1995. URL: http://dl.acm.org/citation.cfm?id=313651.313681.

4 Thomas Ottmann and Peter Widmayer. *Algorithmen und Datenstrukturen, 5. Auflage.* Spektrum Akademischer Verlag, 2012. URL: https://link.springer.com/book/10.1007/978-3-8274-2804-2.

Entwicklung einer Sortierte-Liste-Datenstruktur für 32-Bit Integer Schlüssel

Jan-Gin Yuen (Matrikelnummer 165542)[1]

1 **TU Dortmund, Emil-Figge-Straße 50, 44227 Dortmund, Germany**
`https://www.tu-dortmund.de//`

Zusammenfassung

Obwohl schon gute Implementationen von vergleichbasierten Datenstrukturen existieren, die sich auch in der Praxis als effizient bewiesen haben, sind Suchbaumdatenstrukturen, wie die von van Emde Boas (vEB), eine attraktive Alternative. Sie besitzen für kleinere Zahlenwerte und viele Operationen eine bessere asymptotische Leistung, da sie Zahlen nach oben beschränken und diese Einschränkung nutzen, um die Laufzeit der Operationen zu verbessern. Jedoch zeigt sich in der Praxis, dass sie trotzdem nicht mit den anderen Datenstrukturen mithalten können.

Das Paper stellt eine Datenstruktur vor, die vom gleichen Ansatz ausgeht, und für 32-Bit Integer ausgelegt ist. Ausgehend vom vEB-Baum werden einige Eigenschaften modifiziert und hinzugefügt, um eine Datenstruktur zu erstellen, die vergleichsbasierte Datenstrukturen im Suchen übertrifft und mindestens vergleichbare Leistungen im Einfügen und Löschen aufweist. Dies wird erreicht durch das Speichern der Elemente in einer doppelt verketteten Liste und das zusätzliche Nutzen eines speziellen Suchbaums, welcher in drei Ebenen aufgespalten ist. Der Suchbaum erlaubt es, schneller auf die Elemente in der Liste zuzugreifen.

1998 ACM Subject Classification E.1 Data Structures

Keywords and phrases advanced data structures, sorted list, van Emde Boas

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Einleitung

In der Informatik beschreibt eine Datenstruktur, wie Daten gespeichert und organisiert werden. Sie wird sowohl durch die enthaltenden Daten charakterisiert, als auch durch die Weise, wie sie festgehalten und verwaltet wird. Die interessanten Eigenschaften einer Datenstruktur sind die benötigte Laufzeit für das Durchführen einer Operation und der Speicherplatzbedarf.

Die Laufzeiten der Standardoperationen INSERT, DELETE und SEARCH von einigen verschiedenen Datenstrukturen sind in Tabelle 1 aufgeführt. Darin sind zwei vergleichsbasierte Datenstrukturen aufgelistet: das sortierte Array und der balancierter Baum. Die Hashtabelle

	Array, sortiert	balancierter Baum	Hashtabelle	van-Emde-Boas-Baum
INSERT	$O(n)$	$O(\log(n))$	$O(1)$	$O(\log(\log(M)))$
DELETE	$O(n)$	$O(\log(n))$	$O(1)$	$O(\log(\log(M)))$
SEARCH	$O(\log(n))$	$O(\log(n))$	$O(1)$	$O(\log(\log(M)))$
PREDECESSOR	$O(\log(n))$	$O(\log(n))$	$O(n)$	$O(\log(\log(M)))$
SUCCESSOR	$O(\log(n))$	$O(\log(n))$	$O(n)$	$O(\log(\log(M)))$

Tabelle 1 Laufzeiten der Operationen von verschiedenen Datenstrukturen. n ist die Anzahl der eingefügten Elemente, und M die maximale Schlüsselgröße

.

ist eine besondere Datenstruktur. Sie verwendet eine Hashfunktion, um die Position eines Objektes in der Tabelle zu bestimmen, was zu einer üblicherweise konstanten Laufzeit der Standardoperationen führt. Jedoch ist der Nachteil dieser Datenstruktur, dass der Kontext zwischen den einzelnen Daten verloren geht. Beispielsweise ist eines der Nachbareinträge in einem sortierten Array das nächstgrößere oder kleinere Element, je nach dem wie das Array sortiert ist. Jedoch kann man dies in einer Hashtabelle nicht ohne durchlaufen der Einträge herausfinden. Dadurch werden einige andere Operationen, wie beispielsweise das nächstgrößere Element in der Datenstruktur zu finden, erschwert. Der van-Emde-Boas-Baum ist eine Baumdatenstruktur, die nicht vergleichsbasiert ist. Auf diese Datenstruktur wird in Abschnitt 2 eingegangen. Sie besitzt nicht nur für die Standardoperationen, sondern auch für *MINIMUM*, *MAXIMUM*, *PREDECESSOR* und *SUCCESSOR* eine Laufzeit von $O(\log(\log(M)))$, wobei M die maximale Schlüsselgröße ist. Demnach kann man M als $M = 2^m$ definieren, wobei m die maximale Anzahl der verwendeten Bits sind. Diese Datenstruktur zeigt in der Praxis jedoch, dass sie langsamer als effiziente vergleichsbasierte Datenstrukturen ist[2].

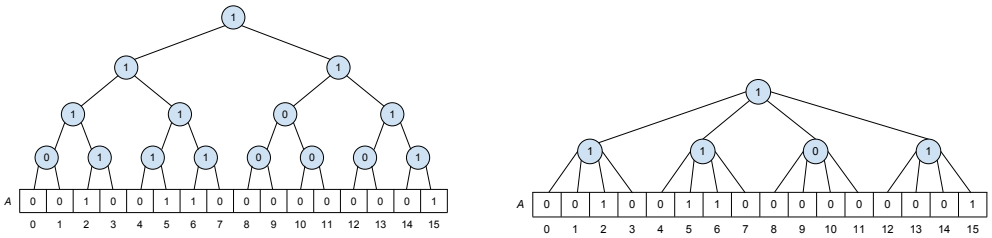
Das Paper präsentiert eine effiziente praktische Implementierung der van-Emde-Boas-Datenstruktur.

2 van-Emde-Boas-Datenstruktur

Zum Verständnis der eigentlich vorzustellenden Datenstruktur wird an dieser Stelle die Datenstruktur erläutert, auf der sie basiert.

Die ursprüngliche Idee ist es, anstatt Werte zu vergleichen und sie demnach einzuordnen, sie in einer schon festen Ordnung zu halten. Das bedeutet, dass für jeden möglichen Schlüssel ein fester Platz reserviert ist. Die einfachste Lösung dafür ist ein Bit-Vektor, wobei jeder Feldeintrag eine Zahl repräsentiert. Dazu beschränkt man sich auf eine Eingabegröße von höchstens u, und legt folglich einen Bit-Vektor der Größe u an. Diese Lösung erlaubt keine Duplikate. Die Standardoperationen sind dann zwar mit Laufzeit $O(1)$ lösbar, jedoch steigt die Laufzeit von anderen Operationen wie SUCCESSOR auf $O(u)$, da man im schlechtestem Fall den ganzen Vektor durchlaufen muss. Um dem entgegenzuwirken, wird ein Hilfsbaum auf den Bit-Vektor gesetzt. Jeder Knoten in diesem Baum enthält das Ergebnis des logischem OR seiner Kinderknoten. Eine Veranschaulichung mit Knotengrad 2 wird in Abbildung 1a dargestellt.

Um die Anzahl der Ebenen zu verringern, erhöht man den Grad jedes Knoten auf $\sqrt{u}$,



(a) Grafisches Beispiel eines Bit-Vektors der Länge 16 mit einem Hilfsbaum des Knotengrades 2

(b) Darstellung eines Beispiels mit einem Bit-Vektor der Länge 16 zusammen mit einem Hilfsbaum des Knotengrades $\sqrt{16}$

Abbildung 1 Darstellungen von Bit-Vektoren der Länge 16, mit Bäumen unterschiedlicher Knotengrade

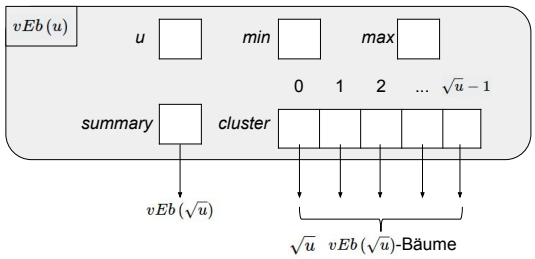


Abbildung 2 Veranschaulichung der $vEB(u)$ Struktur.

wie es in Abbildung 1b zu sehen ist. Man nimmt also an, dass $u = 2^{2k}$ für eine ganze Zahl k sei, sodass $\sqrt{u}$ eine ganze Zahl ist.

Als nächstes wird diese Vorgehensweise rekursiv angewendet. Mit dem anfänglichen Datensatz der Länge u erstellt man eine Struktur der Länge $\sqrt{u} = u^{\frac{1}{2}}$, dessen Einträge selber jeweils wieder auf eine Struktur der Länge $u^{\frac{1}{4}}$ zeigen, dessen Einträge wiederum auf eine kleinere Struktur der Länge $u^{\frac{1}{8}}$ zeigen, und so weiter, bis die Struktur eine Länge von 2 besitzt.

An dieser Stelle wird die finale Struktur vorgestellt: ***vEB(u)***, wobei u die Länge ist. Sie enthält, wie in Abbildung 2 dargestellt, folgende Informationen:

- ***u***: enthält die Länge der Struktur.
- ***min***: beinhaltet den Index des kleinsten Elementes.
- ***max***: zeigt den Index des größten Elementes.
- ***cluster***: ein Array der Länge $\sqrt{u}$, wobei jeder Feldeintrag auf eine $vEB(\sqrt{u})$ Struktur verweist.
- ***summary***: repräsentiert eine $vEB(\sqrt{u})$ Struktur, die das logische OR über die Elemente in *cluster* darstellt.

Zusätzlich wird das *min* Element in keinem der rekursiven vEB-Bäume gespeichert, sondern nur im *min* Attribut selbst. Dies verringert die Anzahl der rekursiven Aufrufe von Operationen, weil es dadurch möglich ist, in konstanter Zeit in eine leere vEB-Struktur einzufügen.

Da die kleinste Länge (Basislänge) 2 ist, benötigt ein $vEB(2)$ kein *summary* und *cluster*. Stattdessen kann man die Elemente anhand der Attribute *min* und *max* bestimmen. Es sind also nur u, *min* und *max* enthalten. Besitzt ein vEB-Baum keine Elemente, so sind *min* und *max* NIL.

An dieser Stelle werden kurz die einzelnen Operationen im Detail erläutert. Folgende Hilfsfunktionen werden verwendet:

- $high(x) = \lfloor x/\sqrt{u} \rfloor$, die Zahl, die entsteht, wenn man die $\sqrt{u}/2$ höchstwertigen Bits einer Zahl als Binärzahl interpretiert.
- $low(x) = x \bmod \sqrt{u}$, die Zahl, die entsteht, wenn man die $\sqrt{u}/2$ niedrigstwertigen Bits einer Zahl als Binärzahl interpretiert.
- $index(x, y) = x\sqrt{u} + y$, die Zahl, die entsteht, wenn man die Bits von y an die von x anhängt.

$high(x)$ wird dafür verwendet, um den vEB-Baum im *cluster* zu finden, in dem die Zahl x gespeichert ist. $low(x)$ bestimmt dann die Zahl, die in diesem Unterbaum gesucht wird. $index(x, y)$ wird verwendet, um die ursprüngliche Zahl aus $high(x)$ und $low(x)$ zu rekonstruieren.

Da das Minimum und das Maximum in den Attributen *min* und *max* gespeichert ist, brauchen die beiden Operationen MINIMUM und MAXIMUM jeweils nur konstante Zeit.

Die SEARCH Operation gibt einen Boolean zurück, welcher Auskunft darüber gibt, ob ein Wert in der Datenstruktur vorhanden ist oder nicht. Zeile 2-3 überprüft, ob x gleich dem *min* oder *max* Attribut ist, und gibt TRUE zurück, wenn das der Fall ist. Andernfalls fragt Zeile 4 ab, ob die jetzige vEB-Struktur der Zeile die kleinste Größe hat. Da ein $vEB(2)$-Baum nur *min* und *max* Attribute hat, gibt Zeile 5 FALSE zurück. Sind beide Abfragen negativ, wird die Methode rekursiv im korrespondierenden vEB-Baum aufgerufen.

Algorithm 1 vEB-SEARCH(V, x)

```
1: procedure vEB-SEARCH(V, x)
2:     if x == V.min or x == V.max then
3:         return TRUE
4:     else if V.u == 2 then
5:         return FALSE
6:     else
7:         return vEB-SEARCH(V.cluster[high(x)], low(x))
```

Die Abbruchkriterien der Rekursion für die PREDECESSOR Operation sind in Zeile 2 und 7 dargestellt und sind trivial. Sind die Abbruchkriterien nicht erfüllt, ist der Baum V nicht 2 groß und x ist kleiner oder gleich dem maximalem Wert in V. Enthält zusätzlich das zu x korrespondiere *cluster* einen kleineren Wert als x, liegt der Vorgänger von x noch im selbem *cluster* und es wird darin nach ihm gesucht. Andernfalls wird das Maximum des vorherigen *cluster* zurückgegeben, das Werte enthält. Gibt es dieses *cluster* nicht, so wird zunächst noch im *min* Attribut nachgesehen, da der Wert, der in *min* steht, nicht auch im *cluster* steht. Ist dann der Wert in *min* nicht kleiner als x, so gibt es keinen Vorgänger und es wird NIL zurückgegeben.

Die SUCCESSOR Operation funktioniert symmetrisch zur PREDECESSOR Operation. Es gibt eine Asymmetrie zwischen *min* und *max*, die dadurch entsteht, dass *min* nicht auch im *cluster* gespeichert wird, aber *max* schon. Dadurch fallen für SUCCESSOR die Zeilen 17 und 18 weg.

Algorithm 2 vEB-PREDECESSOR(V, x)

```
1: procedure vEB-PREDECESSOR(V, x)
2:     if V.u == 2 then
3:         if x == 1 and V.min == 0 then
4:             return 0
5:         else
6:             return NIL
7:     else if V.max ≠ NIL and x > V.max then
8:         return V.max
9:     else
10:        min-low = vEB-MINIMUM(V.cluster[high(x)])
11:        if min-low≠ NIL and low(x) > min-low then
12:            offset =vEB-PREDECESSOR(V.cluster[high(x)], low(x))
13:            return index(high(x), offset)
14:        else
```

```
15:         pred-cluster = vEB-PREDECESSOR(V.summary, high(x))
16:         if pred-cluster == NIL then
17:             if V.min ≠ NIL and x > V.min then
18:                 return V.min
19:             else
20:                 return NIL
21:         else
22:             offset = vEB-MAXIMUM(V.cluster[pred-cluster])
23:             return index(pred-cluster, offset)
```

INSERT lässt sich in zwei verschiedene Operationen spalten. Wird ein Element in einen leeren vEB-Baum eingefügt, so geschieht das in konstanter Zeit, da nur die min und max Attribute gesetzt werden müssen. Ist der Baum jedoch nicht leer, wird zunächst abgefragt, ob der einzufügende Wert x kleiner ist, als das jetzige Minimum im Baum. Ist dies der Fall, wird das min Attribut auf x gesetzt, aber der Wert, der vorher drin stand, muss nun eingefügt werden. Im Falle eines vEB-Baumes größer als 2 ist es notwendig, einen rekursiven Aufruf zu tätigen. Es wird in das dazugehörige *cluster* eingefügt. Ist dieses leer, so muss auch *summary* aktualisiert werden. In beiden Fällen ist höchstens ein Aufruf rekursiv. Wenn notwendig, wird anschließend das max Attribut aktualisiert.

Die DELETE Operation arbeitet wie folgt: Enthält der vEB-Baum V lediglich ein Element, ist es genauso einfach es zu löschen, wie es einzufügen ist. Man setzt dazu das min und max Attribut auf NIL, was in Zeile 2-4 geschieht.

Hat der Baum mehr als nur ein Element und ist die Größe des Baumes gleich 2, so enthält der Baum offensichtlich zwei Elemente. Es werden dementsprechend min und max in Zeile 6-10 angepasst.

Die restlichen Zeilen gehen davon aus, dass in V zwei oder mehr Elemente enthalten sind. In diesem Fall wird ein Element aus einem *cluster* gelöscht. Dieses Element ist in dem Fall, dass $x = V.min$ ist, jedoch nicht x, sondern das Minimum im ersten *cluster*, das ein Minimum hält, da $V.min$ dieses übernimmt. Dieser Austausch wird in Zeile 12-16 ausgeführt.

Anschließend wird in Zeile 16 das Element im *cluster* gelöscht. Ist das *cluster* nach dem Löschen leer, so muss auch *summary* angepasst werden. Dieser Vorgang ist in Zeile 17 und 18 beschrieben. Nach der Aktualisierung muss möglicherweise auch max aktualisiert werden. Zeile 19 überprüft, ob das Maximum gelöscht wird, und ist dies der Fall, so wird *summary-max* auf den Index des letzten nicht leeren *clusters* in *summary* gesetzt (Zeile 20).

Danach wird $V.max$ in Zeile 21-24 aktualisiert.

Anschließend muss der Fall behandelt werden, in welchem das *cluster* von x nicht leer wurde und x aber gleich dem maximalem Element in V entspricht. Demnach muss $V.max$ gleich dem max des aus dem *cluster* gelöschtem vEB-Baum gesetzt werden. Dieser Fall wird in Zeile 25 und 26 abgedeckt.

Auf dem ersten Blick sieht es so aus, als würden potenziell zwei rekursive Aufrufe geschehen. Bei genauerem Untersuchen stellt man jedoch fest, dass, falls Zeile 18 erreicht wird, der Aufruf in Zeile 16 konstante Zeit kostet, da dieser nur die Zeilen 2-4 aufruft, weil das *cluster* nur ein Element enthielt.

Algorithm 3 vEB-DELETE(V, x)

```
1: procedure vEB-DELETE(V, x)
2:     if V.min == V.max then
3:         V.min = NIL
```

```
4:          V.max = NIL
5:      else if V.u == 2 then
6:          if x == 0 then
7:              V.min = 1
8:          else
9:              V.min = 0
10:         V.max = V.min
11:     else
12:         if x == V.min then
13:             first-cluster = vEB-MINIMUM(V.summary)
14:             x = index(first-cluster, vEB-MINIMUM(V.cluster[first-cluster]))
15:             V.min = x
16:         vEB-DELETE(V.cluster[high(x)], low(x))
17:         if vEB-MINIMUM(V.cluster[high(x)], low(x)) = NIL then
18:             vEB-DELETE(V.summary, high(x))
19:             if x == V.max then
20:                 summary-max = vEB-MAXIMUM(V.summary)
21:                 if summary-max == NIL then
22:                     V.max = V.min
23:                 else
24:                     V.max = index(summary-max,
                            vEB-MAXIMUM(V.cluster[summary-max]))
25:         else if x == V.max then
26:             V.max = index(high(x),
                            vEB-MAXIMUM(V.cluster[high(x)])
```

Jede Operation ruft sich höchstens einmal rekursiv auf. Mit jedem rekursiven Aufruf reduziert sich die zu durchsuchende Datenmenge um $\sqrt{u}$, wobei u die Größe der jeweiligen vEB-Struktur ist. Zusätzlich dazu werden nur Funktionen verwendet, die konstante Zeit ($O(1)$) benötigen. Dies lässt sich zusammenfassen als Rekursionsgleichung $T(u) = T(\sqrt{u}) + O(1)$. Sei also $m = \log(u)$ und $S(m) = T(2^m)$. Daraus ergibt sich $S(m) = T(2^m) = T(u) = T(\sqrt{u}) + O(1) = T(2^{\frac{m}{2}}) + O(1) = S(m/2) + O(1)$. Unter Anwendung des Master-Theorems ergibt sich $S(m) = O(\log(m))$, womit $T(u) = O(\log(\log(u)))$ gilt. Die Laufzeiten der MINIMUM und MAXIMUM Operation betragen $O(1)$, und die der restlichen Operationen $O(\log(\log(u)))$.

Eine nähere Erklärung der van-Emde-Boas-Datenstruktur ist in [1] zu finden.

3 Die Datenstruktur

Die zu beschreibende Datenstruktur, die im Paper als *Stree* referenziert wird, ist eine effiziente Implementierung der vEB-Datenstruktur, die auch in der Praxis bessere Laufzeiten als vergleichsbasierte Datenstrukturen erzielt, wie sich später zeigt. Sie beschränkt sich bezüglich der Eingabewerte auf 32-Bit Integer-Schlüssel und erlaubt wie der vEB-Baum keine Duplikate. Somit ist die Eingabe auf ganzzahlige Werte im Intervall $[0, 2^{32} - 1]$ begrenzt und die höchstmögliche Anzahl der gespeicherten Werte beträgt 2^{32}.

Im Folgenden wird diese Notation verwendet:

- $x[i]$ repräsentiert das i-te Bit der ganzzahligen Zahl x, sodass $x = \sum_{i=0}^{31} 2^i x[i]$.
- $x[i..j]$, mit $i \leq j + 1$, stellt die Zahl dar, die entsteht, wenn man nur die Bits ab i bis j übernimmt, sodass $x[i..j] = \sum_{k=i}^{j} 2^{k-i} x[i]$.

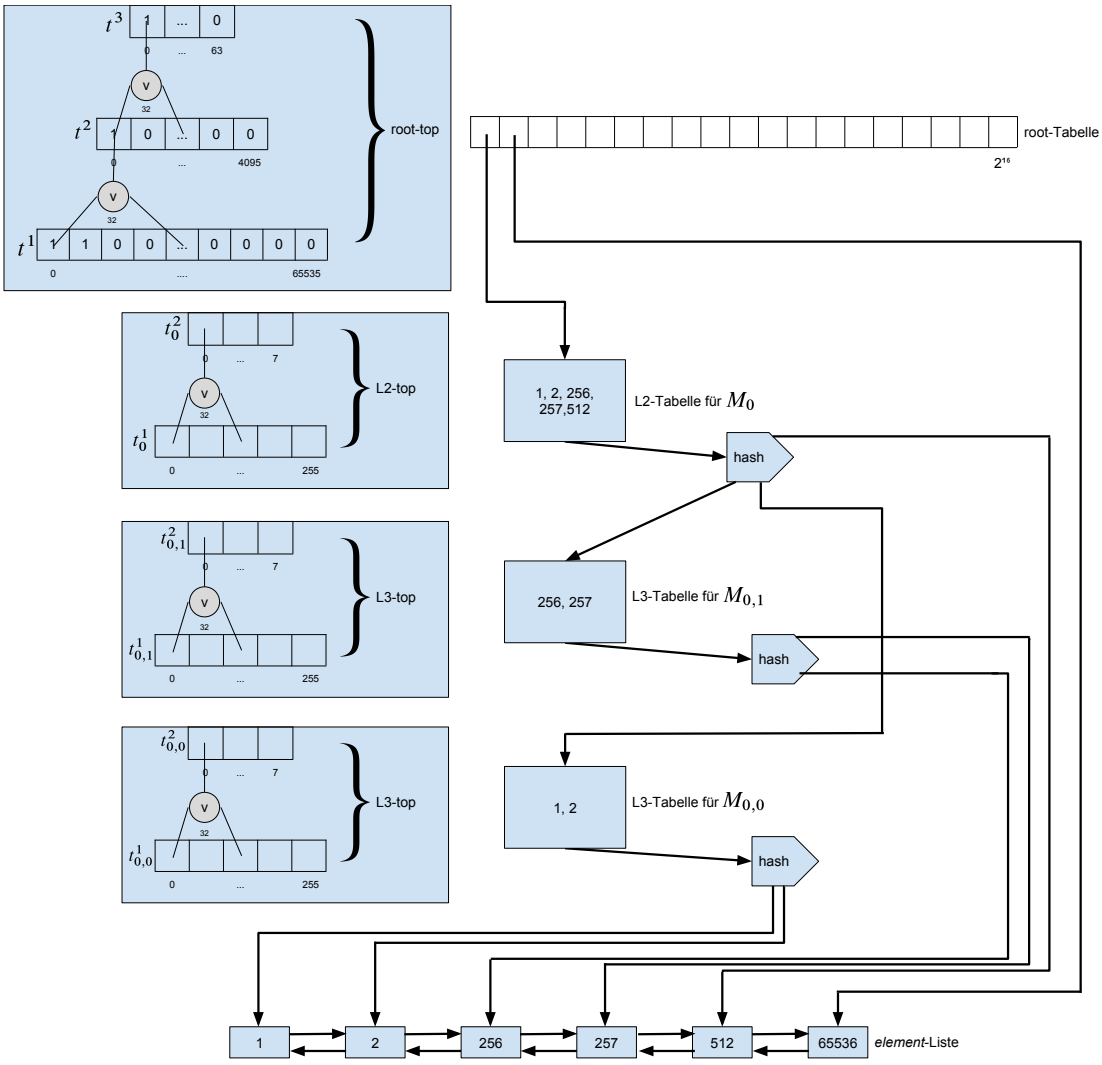


Abbildung 3 Die *Stree*-Datenstruktur für $M = \{1, 2, 256, 257, 512, 65536\}$.

- M bezeichnet die gesamte Menge der schon in der Datenstruktur gespeicherten Elemente
- M_i bezeichnet die Teilmenge von M, in der die höchstwertigen 16 Bits die Zahl i bilden. Es gilt also $M_i = \{x \in M : x[16..31] = i\}$.
- $M_{i,j}$ bezeichnet die Teilmenge von M, in der die höchstwertigen 16 Bits die Zahl i und die 8 nächsten höchstwertigen Bits die Zahl j bilden. Es gilt also $M_{i,j} = \{x \in M : x[8..15] = j, x[16..31] = i\}$.
- $|M|$ ist die Anzahl der Elemente in der Menge M.
- $msbPos(z)$ gibt die Position des größten Bit ungleich 0 in z zurück, sodass $msbPos(z) = \lfloor \log_2(z) \rfloor = \max\{i : x[i] \neq 0\}$

Stree speichert Elemente in der doppelt verketteten sortierten ***element*-Liste** ab und verwendet eine geschichtete Index-Baum-Struktur als Hilfsdatenstruktur, um einen schnellen Zugriff auf die Elemente der Liste zu gewährleisten. Mithilfe dieses Hilfsbaumes können neben den Standardoperationen auch Operationen wie SUCCESSOR, PREDECESSOR oder Bereichsabfragen effizient implementiert werden. Die Index-Datenstruktur besteht aus den 3

Ebenen *root*, Level 2 (*L2*) und Level 3 (*L3*):

Die ***root*-Tabelle** $r[0..2^{16}-1]$ ist ein Zeiger-Array, in dem der i-te Eintrag zu M_i korrespondiert: Ist $r[i] = \texttt{null}$, gilt $|M_i| = 0$. Falls $|M_i| = 1$ ist, enthält $r[i]$ einen Zeiger zum korrespondierendem Element in der *element*-Liste. Bei $|M_i| > 1$ zeigt $r[i]$ auf eine *L2*-Tabelle, die zu M_i korrespondiert.

Eine ***L2*-Tabelle** $r_i[0..2^8-1]$ speichert Elemente in M_i. Sie verwendet eine Hashtabelle, um einen Eintrag mit dem Schlüssel j zu speichern, falls $\exists x \in M_i : x[8..15] = j$. Ist $|M_{i,j}| = 1$, zeigt der Eintrag $r_i[j]$ auf das korrespondierende Element in der *element*-Liste. Bei $|M_{i,j}| \geq 2$ zeigt er auf eine $L3$-Tabelle, die $M_{i,j}$ repräsentiert.

Eine ***L3*-Tabelle** $r_{i,j}[0..2^8-1]$ ist ähnlich wie die $L2$-Tabelle aufgebaut und verwendet eine Hashtabelle, um einen Eintrag mit dem Schlüssel k zu speichern, wenn $\exists x \in M_{i,j} : x[0..7] = k$. Der Eintrag $r_{i,j}[k]$ zeigt auf ein Element x in der *element*-Liste mit $x[0..7] = k$, $x[8..15] = j$, $x[16..31] = i$.

Außerdem wird in jeder der drei genannten Tabellen das **Minimum** und **Maximum** gespeichert. Das bedeutet, dass der Schlüssel und der Zeiger zum Element in der *element*-Liste des kleinsten und größten Elements der jeweiligen Teilmenge von M gespeichert wird.

Es werden zusätzlich noch die 3 Datenstrukturen ***root-top***, ***L2-top*** und ***L3-top*** definiert, welche an das *summary*-Array des vEB-Baums erinnern:

Die ***root-top*** Datenstruktur t besteht aus den drei Bit-Arrays $t^1[0..2^{16}-1]$, $t^2[0..2^{12}-1]$ und $t^3[0..2^6-1]$. Es gilt $t^1[i] = 1$, falls $M_i \neq \emptyset$ ist. Somit enthält t^1 ein 1-Bit für jeden nicht-leeren Eintrag in der *root*-Tabelle r. $t^2[j]$ bildet das logische OR über $t^1[32j]..t^1[32j+31]$, sodass $t^2[j] = 1$ gilt, wenn $\exists i \in \{32j..32j+31\} : M_i \neq \emptyset$. Ähnlich dazu bildet $t^3[k]$ das logische OR über $t^2[32k]..t^2[32k+31]$, sodass $t^3[k] = 1$ gilt, wenn $\exists i \in \{1024k..1024k+1023\} : M_i \neq \emptyset$.

Die ***L2-top*** Datenstruktur t_i besteht aus den beiden Bit-Arrays $t_i^1[0..2^8-1]$ und $t_i^2[0..2^3-1]$ und hat einen ähnlichen Aufbau wie die *root-top* Datenstruktur t. t_i^i enthält einen 1-Bit für jeden nicht leeren Eintrag in r_i und die acht Bits in t_i^2 enthalten das logische OR über jeweils 32 unterschiedliche Bits in t_i^1. Diese Datenstruktur wird nur dann angelegt, wenn $|M_i| \geq 2$ gilt.

Die ***L3-top*** Datenstruktur $t_{i,j}$ mit den beiden Bit-Arrays $t_{i,j}^1[0..2^8-1]$ und $t_{i,j}^2[0..2^3-1]$ geben die Einträge von $M_{i,j}$ auf der Art wieder, wie es die *L2-top* Datenstruktur t_i für M_i tut. t_{ij}^1 enthält also wieder einen 1-Bit für jeden nicht leeren Eintrag in $r_{i,j}$ und $t_{i,j}^2$ bildet das logische OR über jeweils 32 Bits in $t_{i,j}^1$.

Die verwendeten Hashtabellen benutzen offene Adressierungen mit linearem Sondieren [3]. Ihre Größe beträgt immer eine Potenz von 2 zwischen 4 und 256. Sie wird zunächst so klein gehalten wie möglich, aber ihre Größe k wird verdoppelt, sobald sie mehr als $\frac{3}{4}k$ Einträge hält, solange $k < 256$ gilt. Sie wird halbiert, falls sie weniger als $k/4$ Einträge hält. Da alle Schlüssel zwischen 0 und 255 sind, wird die Hashfunktion als eine Nachschlagetabelle (*lookup table*) h implementiert, welche von allen Tabellen verwendet wird.

Sei beispielsweise $M = \{1, 2, 256, 257, 512, 65536\}$. Es gilt dann $M_0 = \{1, 2, 256, 257, 512\}$ und $M_1 = \{65536\}$, da die höchstwertigen 16 Bits der ersten fünf Zahlen die dezimale Zahl 0 und die der letzten Zahl 1 bildet. Fortführend gilt $M_{0,2} = 512$, wegen $512[8..15] = 2$, $M_{0,1} = \{256, 257\}$, wegen $256[8..15] = 257[8..15] = 1$, und $M_{0,0} = \{1, 2\}$, wegen $x[8..15] = 0$ für alle $x \in M_{0,0}$. $|M_1| = 1$ gilt, also zeigt der *root*-Tabelleneintrag $r[1]$ auf das Element mit dem Wert 65536 in der *element*-Liste. Da $|M_0| > 0$, zeigt $r[0]$ auf die $L2$-Tabelle r_0. Wegen $|M_{0,2}| = 1$ zeigt der Eintrag $r_0[2]$ auf das Element 512 in der *element*-Liste. Da $|M_{0,0}| > 2$, zeigt $r_0[0]$ auf die $L3$-Tabelle $r_{0,0}$. $|M_{0,1}| > 2$ gilt ebenfalls, somit zeigt $r_0[1]$ auf die $L3$-Tabelle $r_{0,1}$. Schließlich zeigen die Einträge $r_{0,0}[1]$, $r_{0,0}[2]$, $r_{0,1}[0]$ und $r_{0,1}[1]$ auf das jeweils korrespondierende Element in der *element*-Liste. Dieses spezifische Beispiel ist in

Abbildung 3 grafisch dargestellt. Es werden zusätzlich also nur eine $L2$- und zwei $L3$-Tabellen verwendet.

4 Die Operationen

An dieser Stelle werden die Operationen auf der Datenstruktur grob beschrieben, da sie im Prinzip durch den der *vEB*-Datenstruktur ähnelnden Aufbau abgeleitet werden können:

SEARCH(x) wandert den Baum, der durch die Tabellen impliziert wird, hinunter, bis das Listenelement zu x gefunden wird. Ist jedoch $x \notin M$, wird ein `null`-Zeiger zurückgegeben. Es werden keine Zugriffe auf die *top*-Datenstrukturen benötigt. Ist beispielsweise die Zahl 256 aus dem Beispiel 3 gesucht, würde man folgendermaßen vorgehen: Zunächst fragt man den Eintrag in der *root*-Tabelle ab. Zunächst werden die höchstwertigen 16 Bits in der Zahl 256, also 256[16..31], verwendet, um den Zeiger auf die $L2$-Tabelle r_0 zu finden. Dann wird anschließend mit dem Schlüssel $j = 256[8..15]$ für die Hashtabelle r_0 der Zeiger auf die $L3$-Tabelle $r_{0,1}$ abgerufen. Schließlich findet man mit dem Schlüssel $k = 256[0..7]$ den Zeiger, der auf das Element in der *element*-Liste zeigt.

SUCCESSOR(y) findet den kleinsten Wert $x \in M$ mit $y \leq x$. Zunächst verwendet SUCCESSOR $y[16..31]$, also die größten 16 Bits in y, um einen Zeiger auf r_i in der *root*-Tabelle r zu finden. Falls $r[i] ==$ `null` ist, womit M_i leer ist, oder das Minimum-Attribut kleiner ist als y, wird das nächste 1-Bit i' in der *root-top* Datenstruktur t gesucht und das kleinste Element von $M_{i'}$ wird zurückgegeben. $j = y[8..15]$ dient nun als Schlüssel für die Hashtabelle r_j. Dieses Muster wiederholt sich für die Level 2 und möglicherweise 3, bis der Zeiger auf ein Element in der *element*-Liste zeigt. SUCCESSOR geht also die Baumstruktur so lange entlang, bis die exakte Position gefunden ist. Die **PREDECESSOR**-Operation funktioniert symmetrisch zu SUCCESSOR.

INSERT(x) geht ähnlich wie SUCCESSOR vor, jedoch verändert die Operation die Datenstrukturen, durch welche sie wandert. Die Minima und Maxima werden aktualisiert und die passenden Bits in den *top*-Datenstrukturen gesetzt. Schließlich wird ein Zeiger auf den *element*-Listeneintrag vom Nachfolger von x verfügbar, so dass x davor eingefügt werden kann. Wenn ein M_i oder $M_{i,j}$ mehr als ein Element hält, werden neue $L2/L3$-Tabellen mit zwei Elementen angelegt. Soll die Zahl 513 in das Beispiel in 3 eingefügt werden. wird zunächst der Nachfolger und Vorgänger gesucht. Dadurch, dass eine doppelt verkettete Liste verwendet wird, muss nur einer davon gesucht werden. Der Vorgänger in diesem Beispiel ist 512 und der Nachfolger 65536. Es wird dann die Zahl 513 in die *element*-Liste eingefügt, indem man die Verkettung von 512 und 65537 anpasst. Außerdem muss der Zeiger in die Tabellenstruktur gebracht werden. Beim Hinunterwandern der Baumstruktur landet man in der $L2$-Tabelle r_0, dessen Eintrag mit dem Schlüssel $j = 513[8..15]$ auf die 512 zeigt. Also muss eine neue $L3$-Tabelle $r_{0,2}$ angelegt werden. Der Eintrag von r_0 zeigt dann auf die neu angelegte Tabelle. Danach werden die Zeiger auf 512 und 513 in $r_{0,2}$ mit dem Schlüssel $k = y[0..7]$, mit $y \in \{512, 513\}$, gespeichert. Schließlich wird die *top*-Datenstruktur $t_{0,2}$ angelegt und den Änderungen entsprechend gesetzt.

DELETE(x) wandert den Baum wie FIND hinunter, löscht das Listenelement x und passt die Minima und Maxima beim Hinaufwandern an. Falls eine $L2$- oder $L3$-Tabelle deshalb auf ein Element schrumpft, werden die korrespondierenden Hashtabellen und *top*-Datenstrukturen gelöscht. Beim Löschen eines Elements oder einer $L2/L3$-Tabelle wird die *top*-Datenstruktur durch das Löschen des Bit, welches zu dem gelöschten Eintrag korrespondiert, aktualisiert. Wird aus dem Beispiel in Abbildung 3 die 1 gelöscht, so zeigt danach die $L2$-Tabelle r_0 für den Schlüssel 0 direkt auf das Element 2. Außerdem werden die $L3$-*top* und -Tabelle für $M_{0,0}$

ebenfalls gelöscht werden. Die 2 wäre dann das erste Element in der *element*-Liste.

Eine Analyse der praxisbezogenen Laufzeit einiger Operationen ist in [2] zu finden.

5 Vergleich mit der van-Emde-Boas-Datenstruktur

Implementiert man eine vEB-Datenstruktur mit dem Bereich der möglichen Werte $[0..2^{32}]$ und $u = 2^{32}$, erhält man einen 6 Ebenen tiefen vEB-Baum, da die Datenmenge mit jeder Ebene auf $\sqrt{u}$ schrumpft, bis die Basisgröße 2 erreicht ist ($2^{32} \to 2^{16} \to 2^8 \to 2^4 \to 2^2 \to 2$). Der *Stree* lässt zwar die Datenmenge mit jeder Tabelle (*root*, $L2$ und $L3$) um die Wurzel schrumpfen, jedoch ist die "Basisgröße" dabei nicht 2, sondern 2^8. Zudem speichert sie, anders als die vEB-Datenstruktur, die Werte nicht in der Struktur selbst, sondern in der sortierten doppelt verketteten *element*-Liste. Es werden Zeiger in den Index-Tabellen verwendet, um auf die Elemente zu zeigen. Dadurch, dass Zeiger verwendet werden, ist es außerdem nicht mehr zwangsläufig notwendig, die Tabellenstruktur bis zum Ende zu verfolgen. Auch mit dem Verwenden von Hashtabellen werden Laufzeiten reduziert.

Die *top*-Datenstrukturen korrespondieren zu den *summary*-Arrays in jedem $vEB(u)$. Dabei wird aber die Datenmenge nicht auf $\sqrt{u}$ reduziert, sondern auf $\frac{1}{32}u$. Dies entspricht in $L2$- und $L3$-*top*, sowie $t^2 \to t^3$ von *root-top* $\log(u)$.

Außerdem ist der vEB-Baum statisch: Der gesamte Baum wird am Anfang aufgebaut und nur die Attribute verändern sich bei den Operationen. Dagegen ist der *Stree* dynamisch, da nur die leere *root*-Tabelle, die *root-top* und die sortierte doppelt verkettete *element*-Liste am Anfang vorhanden sind, und die $L2$ und $L3$ Strukturen nach Bedarf angelegt und gelöscht werden.

6 Fazit

Der *Stree* implementiert die van-Emde-Boas-Datenstruktur auf eine praktische und effiziente Art. Sie beschränkt sich auf 32-Bit Integer Schlüssel und erlaubt keine Duplikate. Jedoch lässt sich letzteres durch Anpassungen lösen. Die Datenstruktur kann vergleichsbasierte Datenstrukturen in der Laufzeit unterbieten, jedoch ist sie durch ihre Architektur nicht sonderlich platzeffizient.

Literatur

1 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

2 Roman Dementiev, Lutz Kettner, Jens Mehnert, and Peter Sanders. Engineering a sorted list data structure for 32 bit key. In Lars Arge, Giuseppe F. Italiano, and Robert Sedgewick, editors, *Proceedings of the Sixth Workshop on Algorithm Engineering and Experiments and the First Workshop on Analytic Algorithmics and Combinatorics (ALENEX-04)*, pages 142–151, New Orleans, LA, USA, January 2004. SIAM.

3 Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

Cache-, hash- und speicher-effiziente Bloomfilter*

Jens Zentgraf (156458)[1]

1 **Technische Universität Dortmund, Otto-Hahn-Straße 14, Dortmund, Deutschland**
jens.zentgraf@tu-dortmund.de

Zusammenfassung

Ein *Bloomfilter* ist eine Datenstruktur, mit der festgestellt werden kann, ob ein Element Teil einer Menge ist. Bei Anfragen, ob ein Element ein Teil der Menge ist, ist es sicher, dass eine negative Antwort immer korrekt ist. Allerdings ist es möglich, dass eine positive Antwort zurückgegeben wird, obwohl das angefragte Element nicht in der Menge enthalten war. Dies wird als *false positive* Antwort bezeichnet. Zunächst werden die Funktionsweise von Bloomfiltern und anschließend mehrere davon abgewandelte Varianten vorgestellt. Teilweise bieten diese eine größere Funktionalität als einfache Bloomfilter oder haben eine bessere Cache-Effizienz beziehungsweise verwenden weniger Hashfunktionen, als die normalen Bloomfilter. Betrachtet werden beispielsweise *Blocked Bloomfilter* alleine und in Kombination mit vorberechneten Bitmustern. Zudem werden Möglichkeiten zur Verbesserungen des Speicherplatzes aufgezeigt. Um Verbesserungen in der Cache-Effizienz, Hash-Effizienz oder der Speichereffizienz zu erreichen, müssen gleichzeitig Abstriche in einem der anderen Bereiche oder der false positive Rate akzeptiert werden. Bei jeder Variante wird explizit die Änderung der Häufigkeit der false positive Antworten analysiert.

Keywords and phrases Bloomfilter, Blocked Bloomfilter, Cache-Effizienz, Hash-Effizienz, Speichereffizienz

Digital Object Identifier 10.4230/LIPIcs.BADS.2017.23

1 Einleitung

Ein Bloomfilter beschreibt eine Datenstruktur, mit der festgestellt werden kann, ob ein Element ein Teil einer Menge ist. Sie wurde im Jahr 1970 von Burton Bloom vorgestellt. Durch die Nutzung mehrerer zufälliger und perfekter Hash-Funktionen kann es dazu kommen, dass für Elemente, die nicht in der Menge enthalten sind, eine positive Antwort gegeben wird. Eine perfekte Hashfunktion bildet für zwei Elemente $x \neq y$ immer auf verschiedene Positionen $h(x) \neq h(y)$ ab. Folglich können nur unterschiedliche perfekte Hashfunktionen für Elemente $x \neq y$ auf die selbe Position abbilden. Diese Antworten werden als false positive Antworten bezeichnet und die Wahrscheinlichkeit, mit der diese auftreten als *false positive* Rate. Weil nicht festgestellt werden kann, ob ein Element wirklich in der Menge enthalten ist, ist es auch nicht möglich, diese zu entfernen. Was hingegen bei einfachen Hashfunktionen möglich ist. Da jedes Element der Menge nur eine feste Anzahl an Bit im Bloomfilter benötigt, während die false positive Rate, unabhängig von der Größe des Eingabeuniversums, konstant bleibt, ist der Speicherbedarf von Bloomfiltern geringer als der vergleichbarer Datenstrukturen.

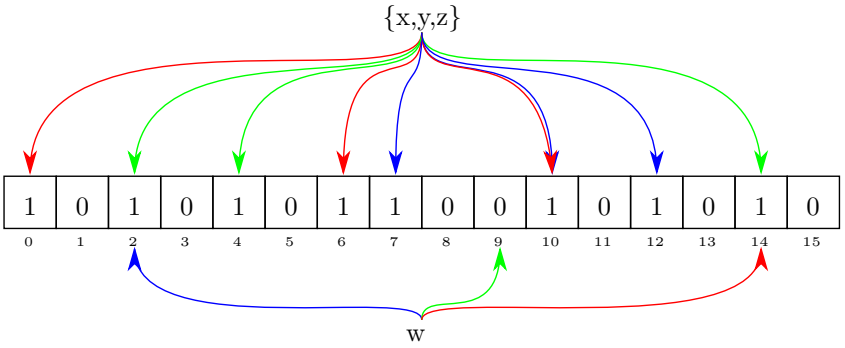
In Abschnitt 2 werden zuerst Standard-Bloomfilter vorgestellt. Diese bestehen aus einem Bitvektor der Länge m. In diesem werden mit k Hashfunktionen k Bit gesetzt. Darauf folgend wird in Abschnitt 3 eine Variante vorgestellt, mit der die Cachegröße effizient genutzt werden kann. Hier werden zwei Ansätze betrachtet und analysiert. Zum Einen ist es möglich, die

* Die Ausarbeitung basiert hauptsächlich auf der Veröffentlichung [6] von Putze, Sanders und Singler.

k zu setzenden Bit in einem kleineren Bitvektor zu setzen. Dieser passt vollständig in den Cache oder ist nicht sehr viel größer, um häufiges Nachladen zu vermeiden. Zum Anderen gibt es eine Variante, bei der die Bit auf x unterschiedliche Blöcke verteilt werden. Bei all diesen Varianten können vorberechnete Bitmuster benutzt werden, womit sich die Anzahl der zu berechnenden Hashfunktionen reduziert und damit auch die Zugriffszeit verbessert. Nachdem mit diesen Varianten die Zugriffszeiten auf Kosten des Speichers und der false positive Rate verbessert wurden, wird in Abschnitt 4 eine Speicher- und Cache-effiziente Variante vorgestellt. Abschließend wird in Abschnitt 5 ein Fazit zu den einzelnen Varianten gezogen.

2 Standard-Bloomfilter

Ein Standard-Bloomfilter setzt sich aus einem Bitvektor der Länge m und einer Menge von n Elementen zusammen. Hierbei beschreibt $c := m/n$ die Anzahl der Bit, die für jedes Element zur Verfügung stehen. Wird nun ein Element in den Bloomfilter eingefügt, werden mit Hilfe von $h_1, .., h_k$ Hashfunktionen k Positionen im Bitvektor berechnet. Initial sind alle Positionen in diesem auf 0 gesetzt und werden nun entsprechend der Hashfunktionen auf 1 gesetzt. Soll überprüft werden, ob ein Element x eingetragen ist, werden $h_1(x), .., h_k(x)$ berechnet. Gilt für eine der Hashfunktionen h_i, dass $h_i(x) = 0$, ist das Element x nicht eingetragen. Für alle Einträge im Bitvektor, die auf 1 gesetzt wurden, ist das Element entweder enthalten oder es ist ein false positive Antwort. Dies ist, wenn nur der Bloomfilter betrachtet wird und die Ursprungsmenge nicht zusätzlich vorhanden ist, nicht feststellbar, da Hashfunktionen auch bei unterschiedlichen Eingaben auf dieselbe Position verweisen können.



	h_1	h_2	h_3
x	0	14	7
y	10	4	12
z	6	2	10
w	14	9	2

Abbildung 1 Ein Standard-Bloomfilter mit einem Bitvektor der Länge $m = 16$. In diesen wurden mit Hilfe von $k = 3$ Hashfunktionen die Elemente x, y und z eingefügt und es wird abgefragt, ob ein Element $w \notin \{x, y, z\}$ in diesem enthalten ist. In der Matrix auf der rechten Seite ist zu erkennen, auf welchen Index des Bitvektors die Hashfunktionen für die einzelnen Elemente abbilden. Das Element w ist nicht im Bloomfilter enthalten, da die Hashfunktionen $h_2(w) = 9$ auf eine 0 abbildet.

In Abbildung 1 sieht man einen Standard-Bloomfilter mit der Länge $m = 16$ und $k = 3$ Hashfunktionen. In diesen wurden die Elemente x, y und z eingefügt. Die Hashfunktionen haben für x die Positionen $h_1(x) = 0$, $h_2(x) = 14$ und $h_3(x) = 7$, für y die Positionen $h_1(y) = 10$, $h_2(y) = 4$ und $h_3(y) = 12$ und für das Element z die Positionen $h_1(z) = 6$, $h_2(z) = 2$ und $h_3(z) = 10$ berechnet. Die Positionen wurden auf 1 gesetzt. Wird nun überprüft, ob das Element $w \notin \{x, y, z\}$ im Bloomfilter enthalten ist, wird auch für dieses mit jeder Hashfunktion eine Position berechnet, die hier $h_1(w) = 14$, $h_2(w) = 9$ und $h_3(w) = 2$ sind. Da an der Stelle 9 eine 0 steht, kann sicher gesagt werden, dass w nicht enthalten ist. Würde die Hashfunktion für w nicht auf Position 9 abbilden, sondern auf 10, wäre an allen

Positionen eine 1 gesetzt. Hieraus würde folgen, dass w entweder eingefügt wurde oder eine false positiv Antwort ist, was hier zutreffend wäre.

Die false positive Rate ist am niedrigsten unter der Annahme, dass die Wahrscheinlichkeit, dass ein Bit auf 1 gesetzt wird, nahe $\frac{1}{2}$ ist und die Anzahl der Hashfunktionen bei $k = \ln 2 * c \approx 0.693 \frac{m}{n}$ ist. Die Wahrscheinlichkeit, dass ein Bit nach dem Einfügen von n Elementen auf 0 bleibt, setzt sich wie folgt zusammen: Die Wahrscheinlichkeit, dass ein Bit nach dem Berechnen einer Hashfunktion für ein Element auf 1 gesetzt wird, beträgt $\frac{1}{m}$. Hieraus ergibt sich die Wahrscheinlichkeit, dass ein Bit auf 0 bleibt, von $1 - \frac{1}{m}$. Für jedes der n Elemente werden k Hashfunktionen berechnet. Folglich ist die Wahrscheinlichkeit, dass ein Bit nach der Berechnung von k Hashfunktionen auf 0 bleibt, $(1 - \frac{1}{m})^k$. Daraus folgt die Wahrscheinlichkeit von $(1 - \frac{1}{m})^{kn}$, dass ein Bit, nach dem Einfügen von n Elementen unter der Verwendung von k Hashfunktionen, auf 0 verbleibt. Dies lässt sich durch

$$p' := (1 - \frac{1}{m})^{kn} \approx e^{-\frac{kn}{m}} \tag{1}$$

abschätzen [1].

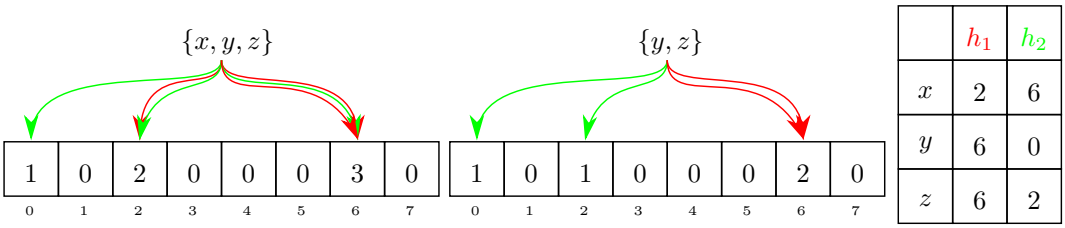
Die false positive Rate für Anfragen von Elementen, die nicht enthalten sind aber positiv beantwortet werden, besteht aus der Wahrscheinlichkeit $(1 - p')$, dass ein Bit, nach dem Einfügen von n Elementen mit k Hashfunktionen auf 1 gesetzt wurde. Die Wahrscheinlichkeit, dass alle Bit, auf die die k Hashfunktionen abbilden auf 1 gesetzt sind, beträgt folglich $(1 - p')^k$. Dies ergibt als false positive Rate für Standard-Bloomfilter

$$f_{std}(m, n, k) = (1 - p')^k \approx (1 - e^{-\frac{kn}{m}})^k \approx \frac{1}{2^k} \ . \tag{2}$$

Datenstrukturen lassen sich in statische, dynamische und teilweise dynamische Datenstrukturen unterteilen. Die Standard-Bloomfilter fallen in den Bereich der teilweise dynamischen Datenstrukturen, da nach dem Aufbau noch die Möglichkeit besteht, neue Elemente einzufügen. Das Entfernen eines Elements ist nicht möglich, da nicht feststellbar ist, ob ein Bit nur durch dieses Element auf 1 gesetzt wurde oder durch mehrere. Im Folgenden wird eine Variante betrachtet, die auch das Löschen von Elementen ermöglicht und damit voll dynamisch ist.

Counting Bloomfilter

Die *Counting Bloomfilter* stellen eine Variante dar, die das Löschen von vorher eingefügten Elementen ermöglicht, wodurch sie zu einer dynamischen Datenstruktur werden. Anstelle eines Bitvektors werden einzelne Zähler für jede Position verwendet. Dadurch kann nachgehalten werden, wie oft eine bestimmt Position auf 1 gesetzt wurde. Jedes Mal, wenn ein Element eingefügt wird und eine der Hashfunktionen auf die entsprechende Position abbildet, wird der Zähler um 1 erhöht. Wird nun ein Element entfernt, werden die einzelnen Positionen berechnet und die entsprechenden Zähler um 1 verringert. Bei Anfragen, ob ein Element im Bloomfilter enthalten ist, wird nun nicht überprüft, ob die Position genau 1 ist, sondern ob der Zähler > 0 ist, was sich in Abbildung 2 zeigt. Hier wurden die Elemente der Menge $\{x, y, z\}$ in einen Counting Bloomfilter eingefügt. Dabei wurden die Zähler an den Positionen 2 und 6 von $h_1(x) = 2$ und $h_2(x) = 6$ auf 1 gesetzt. Danach wurde das Element y eingefügt. Da $h_1(y) = 6$ gilt, wird der Zähler an Position 6 um eins erhöht und liegt somit bei 2. Der Zähler an Position 0 wird von $h_2(y) = 0$ auf 1 gesetzt. Als letztes wird z in den Bloomfilter eingefügt. $h_1(z) = 6$ bildet auch auf Position 6 ab und erhöht somit den Zähler noch einmal



	h_1	h_2
x	2	6
y	6	0
z	6	2

Abbildung 2 Hier sind zwei Counting Bloomfilter und die dazugehörige Matrix dargestellt. Im linken wurden die Elemente der Menge $\{x, y, z\}$ eingefügt. Im rechten wurde das Element x entfernt und die Zähler an den Stellen 2 und 6 um jeweils 1 reduziert.

um 1 auf 3. Da $h_2(z) = 2$ gilt wird auch der Zähler an Position 2 um eins erhöht und hat folglich den Wert 2. In dem zweiten dargestellten Counting Bloomfilter wurde das Element x wieder entfernt. Dabei wurden an den Positionen $h_1(x) = 2$ und $h_2(x) = 6$ die Zähler jeweils um eins reduziert.

Das Entfernen von Elementen aus dem Counting Bloomfiltern kann problematisch werden. Da nicht zusätzlich gespeichert wird, welche Elemente in den Bloomfilter eingefügt wurden, ist es beim Entfernen nicht möglich zu überprüfen, ob das Element vorher eingefügt wurde. So werden vielleicht Elemente entfernt, die nicht eingefügt wurden. Beispielsweise wurde der Zähler an Position 2 durch das Element z auf 1 erhöht und später durch das Entfernen des Elements $v \neq z$ auf 0 reduziert. Wird nun überprüft, ob das Element z enthalten ist, wird hier eine falsche negative Antwort (*false negative* Antwort) zurückgegeben, da der Zähler auf 0 ist obwohl z nicht entfernt wurde [2].

Da nun zum Speichern der eingefügten Elemente nicht mehr ein Bitvektor verwendet wird, sondern ein Vektor von Zählern, wird auch der Speicherverbrauch deutlich vergrößert. Wo vorher $m \cdot 1\text{Bit}$ verwendet wurde, muss jetzt, je nach Größe der Zähler, $m \cdot \lceil \log(i) \rceil \text{Bit}$ verwendet werden, wobei i den maximalen Wert des Zählers repräsentiert. Für $i = 16$ ergibt sich somit eine Vergrößerung um den Faktor 4.

3 Blocked Bloomfilter

Nachdem in Abschnitt 2 die Standard-Bloomfilter vorgestellt wurden, wird nun mit den *Blocked Bloomfiltern* eine Variante erläutert, die den Cache effizient nutzt. Der Cache ist ein schneller, aber sehr kleiner Zwischenspeicher, in dem die aktuell von der CPU genutzten Daten abgelegt werden. Wird nun für einen Bloomfilter ein Bitvektor verwendet, der größer ist als der Cache, kann dieser nicht vollständig in diesen geladen werden. Folglich wird immer nur ein Teil des Bitvektors geladen. Soll nun auf einen Wert zugegriffen werden, der nicht im Cache vorhanden ist, muss dieser Teil des Bitvektors neu geladen werden. Dies wird als *Cache-Miss* bezeichnet. Da der Teil von einem langsameren Speicher geladen werden muss, dauert dieser Vorgang lange und jeder Cache-Miss hat deutliche Auswirkungen auf die endgültige Laufzeit. Bei der Abfrage eines Werts, der nicht im Bloomfilter enthalten ist, werden im Durchschnitt zwei Cache-Misses verursacht. Dies ergibt sich daraus, dass bei einer möglichst kleinen false positive Rate die Wahrscheinlichkeit, dass ein Bit gesetzt ist, $q = \frac{1}{2}$ ist. Dies wird in [1] detaillierter beschrieben. Das bedeutet, dass Bloomfilter für Anfragen, die eindeutig nicht enthalten sind, bereits sehr wenige Cachezugriffe benötigen. Im Vergleich brauchen Anfragen für Elemente, die enthalten sind oder eine false positive Antwort erzeugen, deutlich mehr Cachezugriffe. Hier entsteht im schlimmsten Fall für jede berechnete Hashfunktion ein Cache-Miss, da eventuell für die Überprüfung jedes der k Bit

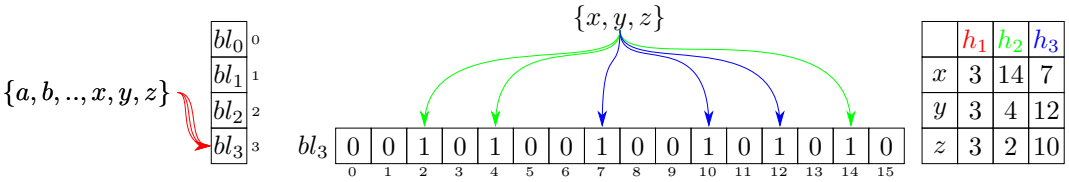


Abbildung 3 Darstellung der einzelnen Abläufe eines Blocked Bloomfilters. Zuerst wird mit Hilfe der Hashfunktion h_1 der Bloomfilter ausgewählt, in den ein Element eingefügt wird. Für die Elemente x, y und z ist dies der Bloomfilter bl_3. In diesen werden die Elemente wie bei einem Standard-Bloomfilter eingefügt.

ein anderer Teil geladen werden muss. Dasselbe gilt beim Einfügen von neuen Elementen.

Blocked Bloomfilter bestehen nicht aus einem großen Bitvektor, sondern unterteilen sich in b Blöcke in denen Standard-Bloomfilter der gleichen Größe verwendet werden. Diese werden aneinandergereiht. Jeder davon ist kleiner als eine Cache-Line. Eine Cache-Line ist meistens 64 Byte = 512 Bit groß und beschreibt die Menge an Daten, die in einer Operation in den Cache geladen werden können. Somit kann ein Block vollständig und in einer Operation geladen werden. Daraus ergibt sich, dass jeder verwendete Standard-Bloomfilter einen Bitvektor der Länge 512 verwenden kann. Um eine optimale Nutzung zu gewährleisten, werden die einzelnen kleinen Bloomfilter *Cache-Line-aligned* abgespeichert. Ist eine Datenstruktur größer als so ein Block, müssen davon mehrere geladen werden. Also bedeutet Cache-Line-aligned, dass die Daten im Speicher so abgelegt werden, dass nur auf möglichst wenige Blöcke zugegriffen werden muss.

Bei jeder Anfrage legt der Wert der ersten Hashfunktion den jeweiligen Block fest. Die folgenden Hashfunktionen setzten dann, wie bei einem Standard-Bloomfilter, in diesem Block die einzelnen Bit. Da alle Bit auf diesen einen Block begrenzt sind, wird nur ein Cache-Miss verursacht, was Abbildung 3 verdeutlicht. Hier werden die Elemente x, y und z in den Blocked Bloomfilter eingefügt. Dabei wird zuerst für jedes Element mit Hilfe der ersten Hashfunktion der entsprechende Block ausgewählt. Hiernach werden die einzelnen Elemente in den geladenen Block eingefügt. Dieser funktioniert wie ein Standard-Bloomfilter.

Weil die einzelnen Blöcke aus Standard-Bloomfiltern bestehen, kann davon ausgegangen werden, dass auch die false positive Rate der Blocked Bloomfilter von der false positive Rate der Standard-Bloomfilter abhängig ist. Der Blocked Bloomfilter kann allerdings nicht einfach als ein Standard-Bloomfilter betrachtet werden. Es würden mehrere Probleme auftreten. Die false positive Rate des Standard-Bloomfilters ist nur von k, der Anzahl der Hashfunktionen, n, der Anzahl der eingefügten Elemente und m, der Größe des Bitvektors abhängig, wobei k und n gleich bleiben. Die Länge der einzelnen Standard-Bloomfilter m_1 bis m_b kann addiert als die Länge eines Standard-Bloomfilters betrachtet werden, was zu Annahme von $m = \sum_{i=0}^{b} m_i$ führt. Allerdings handelt es sich bei m_1 bis m_b um deutlich kleineren Werte. Die in Gleichung 2 vorgestellte Abschätzung zur false positive Rate für Standard-Bloomfilter wird deshalb ungenau. Zudem sind die n Elemente nicht gleichmäßig über alle Blöcke verteilt, weil durch die erste Hashfunktion der entsprechende Block festgelegt wird und dies zufällig geschieht. So kann es passieren, dass in einigen Blöcken sehr viele Elemente eingetragen werden, wodurch diese überfüllt wären, und in anderen wenige bis keine. Die Verteilung der Elemente auf die einzelnen Blöcke kann durch eine Binominal-Verteilung $B(n, \frac{1}{b})$ beschrieben werden. Die Variable n beschreibt die eingefügten Elemente. Da die erste Hashfunktion aus den b Blöcken einen zufällig auswählt, ergibt sich die Wahrscheinlichkeit $\frac{1}{b}$, dass ein bestimmter Block ausgewählt wird. Diese lässt sich mit Hilfe einer Poisson-Verteilung mit den Parametern $\frac{n}{b} = \frac{B}{c}$ annähern. Im Allgemeinen ist die Poisson-Approximation [3] bei

großen Stichproben mit einer kleinen Wahrscheinlichkeit möglich. Durch die Änderung des Grenzwerts zu ∞ ergibt die Poisson-Verteilung eine Annäherung an die Binominal-Verteilung. Diese Approximation ist hier möglich, da n meistens sehr groß und $\frac{B}{c}$ eine kleine Konstante ist. Hieraus resultiert eine false positive Rate für Blocked Bloomfilter von

$$f_{blo}(B, C, k) := \sum_{i=0}^{\infty} \mathbf{Poisson}_{\frac{B}{c}}(i) * f_{\mathbf{inner}}(B, i, k). \tag{3}$$

Durch die Nutzung von Blocked Bloomfiltern wird somit die Häufigkeit von Cache-Misses reduziert, was eine deutliche Verbesserung der Laufzeit bewirkt. Ein weiterer für die Laufzeit ausschlaggebender Punkt ist die Anzahl der notwendigen Berechnungen, die gemacht werden müssen. Ein Teil der Zeit wird darauf verwendet bei jedem Einfügen und bei jeder positiven bzw. false positive Antwort k Hashfunktionen zu berechnen.

3.1 Vorberechnete Bitmuster

Die Anzahl der Berechnungen lässt sich reduzieren, indem eine Tabelle mit vorberechneten Bitmustern verwendet wird. Jedes Bitmuster hat die Länge B und k gesetzte Bit. Mit Hilfe einer Hashfunktion wird nun für jedes Element ein Bitmuster aus der Tabelle ausgewählt und auf den Bitvektor angewendet. Da diese Tabellen meistens vollständig in den Cache passen, tritt so kein Cache-Miss mehr auf. Weiterhin muss nur noch ein Hashwert berechnet werden, da durch diesen alle k Bit gesetzt werden, wie Abbildung 4 zeigt. Hier wird mit Hilfe von h_1 für x (grün), y (blau) und z (rot) das entsprechende Bitmuster ausgewählt. Beim Einfügen der Elemente wird mit x begonnen. Hierbei wird der mit Nullen befüllte Bitvekor mit dem Bitmuster verodert. Dasselbe Vorgehen wird für y und z angewandt.

Als größtes hier auftretendes Problem erweist sich, dass zwei unterschiedliche Elemente auf das gleiche Bitmuster abbilden, wodurch sich die false positive Rate erhöht. Da bei den Standard-Bloomfiltern k Hashfunktionen berechnet werden, ist es nicht ausschlaggebend, dass zwei Elemente auf den gleichen Wert abgebildet werden. Da nun aber nur noch eine Hashfunktion verwendet wird, gibt es keine Möglichkeit mehr, Fehler dadurch zu reduzieren. In einer Tabelle von Bitmustern mit l Einträgen lässt sich die Wahrscheinlichkeit, dass es eine Kollision bei n Einfügeoperationen gibt, mit $p_{coll}(n, l) := 1 - (1 - \frac{1}{l})^n$ berechnen. Hierbei beschreibt $(1 - \frac{1}{l})^n$ die Wahrscheinlichkeit, dass nach n Operationen eine bestimmte Position nicht genutzt wurde. Falls es keine Kollision bei der Auswahl des Bitmusters gibt, können innerhalb eines Blockes immer noch false positive Antworten auftreten, da diese Standard-Bloomfilter sind, wie in Abschnitt 2 beschrieben. Die Wahrscheinlichkeit, dass keine Kollision bei der Auswahl der Bitmuster auftraten, ist durch $1 - p_{coll}(n, l)$ zu beschreiben. Hierzu wird die Wahrscheinlichkeit multipliziert, dass beim Standard-Bloomfilter eine false positive Antwort auftritt. Diese wurde in Gleichung 2 als $f_{std}(m, n, k)$ eingeführt. Die false positive Rate für einen Block ist nun mit

$$f_{pat}(m, n, k, l) \leq p_{coll}(n, l) + (1 - p_{coll}(n, l)) * f_{std}(m, n, k) \tag{4}$$

abzuschätzen. Wird dies als f_{inner} in Gleichung 3 eingesetzt, ergibt sich hieraus die gesamte false positive Rate für Blocked Bloomfilter mit vorberechneten Bitmustern. Im Allgemeinen kann gesagt werden, dass sich die Kombination von vorberechneten Bitmustern mit Blocked Bloomfiltern in der Laufzeit auszahlt, da durch die Verwendung des Blocked Bloomfilters kaum Cache-Misses entstehen und durch die Nutzung der vorberechneten Bitmuster nur eine Hashfunktion zur Wahl des Blocks und eine zur Bestimmung des Bitmusters berechnet werden muss. Dies funktioniert allerdings nur unter der Bedingung, dass die Tabelle mit

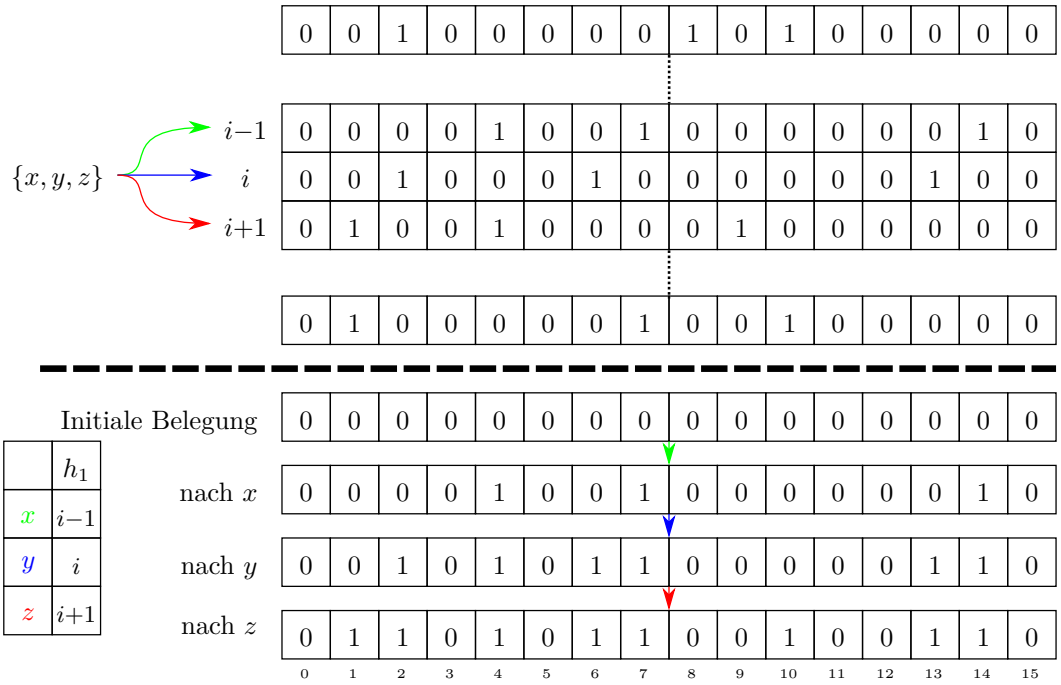


Abbildung 4 Der obere Bereich zeigt eine Tabelle von möglichen Bitmustern, aus denen mit Hilfe einer Hashfunktion jeweils ein vorberechnetes Bitmuster ausgewählt wird. In diesem Beispiel wird zuerst x in einen noch leeren, also mit 0en befüllten, Bitvektor eingefügt. Als nächstes wird für y und zuletzt z ein Bitmuster ausgewählt und hinzugefügt. Im unteren Bereich ist der Bitvektor nach jedem Einfügen eines Elements zu sehen.

den vorberechneten Bitmustern klein genug ist, um in den Cache zu passen, da sonst wieder teure Cache-Misses auftreten. Weiterhin muss sie auch entsprechend viele unterschiedliche vorberechnete Bitmuster enthalten, um das Risiko von Kollisionen zu minimieren. Ist dies nicht der Fall, folgt daraus eine Erhöhung der false positive Rate.

Diese Variante kann noch weiter entwickelt werden, um eine größere Vielfalt in den Mustern zu erhalten. So ist es möglich, an Stelle eines einzelnen Bitmusters mit k Bit x Bitmuster auszuwählen, in denen immer k/x Bit gesetzt sind. Wird nun jedes der x Bitmuster miteinander verodert, ergibt sich so das finale Bitmuster. Unter Außerachtlassung von Rundungsproblemen und Ähnlichem ergibt sich hierfür

$$f_{pat[x]}(m,n,k,l) \approx f_{pat}(m, xn, \frac{k}{x}, l)^n \tag{5}$$

als false positive Rate für einen einzelnen Block. $[x]$ beschreibt die Anzahl der Bitmuster pro Element. Da nun für jedes Element x Bitmuster ausgewählt werden müssen, werden $n * x$ Elemente bei der Berechnung von f_{pat} übergeben. Allerdings werden nur noch $\frac{k}{x}$ Bit pro Element gesetzt.

3.2 Multiblocking

Eine andere Variante, die die in Abschnitt 2 vorgestellten Blocked Bloomfilter verwendet und dadurch eine Verbesserung der false positive Rate erzielt, ist das *Multiblocking*. Hierbei werden nicht k Bit in einen einzelnen Block eingeordnet, sondern auf x unterschiedliche Blöcke verteilt. In jedem Block werden $\frac{k}{x}$ einzelne Bit gesetzt. Sollte k nicht ohne Rest

durch x teilbar sein, wird in den ersten $k \ mod \ x$ Blöcken jeweils ein zusätzliches Bit gesetzt. Hierdurch werden in den ersten $k \ mod \ x$ Blöcken $\lfloor \frac{k}{x} \rfloor + 1$ Bit gesetzt und in den restlichen $\lfloor \frac{k}{x} \rfloor$. Im Bezug auf die false positive Rate ergibt sich hier eine deutliche Verbesserung gegenüber einem einzelnen Bloomfilter mit der Länge $b * x$, da durch die Aufteilung deutlich mehr unterschiedliche Möglichkeiten entstehen. Die Gleichung 2 wird folglich zu

$$f_{std[x]}(m,n,k) = (1 - (1 - \frac{1}{m})^{\frac{kn}{x}})^k \tag{6}$$

erweitert. Die Wahrscheinlichkeit, dass ein Bit gesetzt wurde, bleibt bei $1 - \frac{1}{m}$. Dieses wird dann für $\frac{k}{x} * n$ Elemente berechnet, da für jedes Element $\frac{k}{x}$ einzelne Einfügeoperationen durchgeführt werden müssen. Mit Hilfe von 6 ergibt sich eine false positive Rate von

$$f_{blo[x]}(B,c,k) := \sum_{i=0}^{\infty} \text{Poisson}_{x*\frac{B}{c}}(i) * f_{std[x]}(B, \frac{i}{x}, k)^x. \tag{7}$$

Auch hier kann das Setzen der einzelnen Bit in den Blöcken durch vorberechnete Bitmuster beschleunigt werden. Allerdings müssen x einzelne Blöcke geladen werden, was wieder Cache-Misses erzeugt. Folglich ist das Multiblocking ein Kompromiss zwischen einer guten false positive Rate und wenigen Cache-Misses.

4 Speichereffiziente Bloomfilter

Bei allen bisher vorgestellten Varianten werden Verbesserungen bei der Cache- und Hash-Effizienz erarbeitet, um die Laufzeit zu verbessern. Dies wurde meistens auf Kosten des Speichers oder der false positive Rate erzielt. Nun wird eine Art von Boomfiltern betrachtet, die speichereffizient und gleichzeitig auch Cache-effizient ist. Dies wird durch eine Erhöhung der Laufzeit und einen additiven Teil beim Speicherplatz erreicht.

Anfangs wird eine statische Datenstruktur vorgestellt. Dies bedeutet, dass bereits vor der Nutzung alle zur Menge gehörigen Elemente bekannt sein müssen und vorher in die Datenstruktur eingefügt worden sind. Danach ist es nicht mehr möglich, weitere Elemente einzufügen. Am Ende wird eine dynamische Variante vorgestellt.

In [5] wird gezeigt, dass Standard-Bloomfilter zwar speichereffizient, aber nicht optimal sind. Ohne Berücksichtigung der Laufzeit bei Anfragen, ist es möglich, wenn nur ein Hashwert im Bereich $1, .., \frac{n}{f}$ gesetzt wird, eine false positive Rate von f zu erzielen. Hierbei ergibt sich ein Speicherverbrauch von $log(\binom{\frac{n}{f}}{b}) \approx n * log(\frac{e}{f})$ Bit. Im Vergleich hierzu benötigen Standard-Bloomfilter das $\frac{1}{\ln 2} \approx 1,44$-fache an Speicher. Dies ist im Bezug auf das theoretische Minimum von $n * log(\frac{1}{f})$ Bit nicht optimal.

Um weiterhin eine konstante Zugriffszeit zu gewährleisten, werden für jedes Element zusätzliche Bit benötigt. Diese dienen dazu, die Position des Elements einzuschränken. Je mehr Bit vorhanden sind, desto schneller wird der Zugriff. Weiterhin ist die Anzahl der Bit unabhängig von n und beeinflusst auch nicht die false positive Rate. Diese zusätzlichen Bit nehmen weitere Speicherkapazität in Anspruch. Ebenso sollte eine Hashstruktur nicht vollständig befüllt sein, sondern nur bis zu einem Maximum von α, was ebenfalls zusätzlichen Speicher verbraucht. Zusammengefasst ergibt sich ein Kompromiss zwischen Laufzeit und Speicherplatz, wobei der Speicherverbrauch nahe am Optimum liegt.

4.1 Golomb-Codierte Bloomfilter

In [6] wird ein anderer Ansatz verfolgt. Hier wird eine Methode von [7] zum Speichern von sortierten Integern verwendet. Ein Standard-Bloomfilter mit genau einer Hashfunktion

$k = 1$, der einer normalen *Hashbitmap* entspricht, hat eine false positive Rate von $\frac{1}{c}$. Da im Vergleich nur selten eine 1 gesetzt wird, ist hier Kompression gut anwendbar. Allerdings wird nicht der Bitvektor selbst komprimiert, sondern es werden für alle eingefügten Elemente die berechneten sortierten Hashwerte zur Komprimierung verwendet. Da die Werte eine gleichmäßige Verteilung haben, folgt daraus, dass die Differenzen von aufeinanderfolgenden Elementen ungefähr einer geometrischen Verteilung mit dem Faktor $p = \frac{1}{c}$ entsprechen. Diese lassen sich aufgrund der geometrischen Verteilung sehr gut mit Hilfe des *Golomb-Codes* [4] komprimieren. Bei diesem werden kleine Zahlen mit wenigen Bit und große Zahlen mit mehreren Bit kodiert. Mit einem Parameter kann gesteuert werden, wie schnell die Anzahl der Bit steigt. Wird ein guter Parameter p, der in diesem Fall $p = \frac{1}{c}$ ist, gewählt, erhöht sich der Speicher nur um ca. ein halbes Bit pro Element. Bei der Golomb-Codierung wird die

n	0	1	2	3	4	5	6	7	8	9	10	....	30
b=3	0 0	0 10	0 11	10 0	10 10	10 11	110 0	110 10	110 11	1110 0	1110 10	....	11111111110 0
b=7	0 00	0 010	0 011	0 100	0 101	0 110	0 111	10 00	10 010	10 011	10 100	....	11110 011

Abbildung 5 Die Zahlen 0 bis 10 in Golom-Codierung mit den Parametern $b = 3$ und $b = 7$.

Zahl in zwei Teile unterteilt. Der erste ist der Quotient q. Der zweite ist r, der den Rest der Division beschreibt. Für die Division wird ein Parameter b verwendet, der beliebig aber fest gewählt werden kann. Für q gilt $q = \lfloor \frac{n}{b} \rfloor$. Da r den Rest dieser Division beschreibt, lässt es sich mit $r = n \; mod \; q = n - qb$ berechnen. In der Kodierung wird q in unärer Darstellung ausgegeben. Dies bedeutet, dass q 1en ausgegeben werden und dann eine abschließende 0. Für den zweite Teil r wird eine gekürzte binäre Darstellung (truncated binary encoding) verwendet. In Abbildung 5 sind die Kodierungen für die Zahlen 1 bis 10 für die Parameter $b = 3$ und $b = 7$ dargestellt. Hierbei ist zu erkennen, dass die Kodierung für den Parameter $b = 3$ für die Zahl 0 zwei Bit verwendet und die Kodierung mit dem Parameter $b = 7$ bereits mit 3 Bit beginnt. Allerdings steigt die benötigte Zahl an Bit für größer werdende Zahlen bei $b = 3$ schneller als bei $b = 7$. So werden bei der Kodierung der Zahl 30 mit $b = 3$ bereits 12 Bit verwendet und be $b = 7$ nur 8 Bit. So beginnt $b = 7$ zwar bereits mit mehr Bit, aber die Anzahl nimmt mit größeren Zahlen langsamer zu.

Auf die komprimierten Daten kann nicht direkt zugegriffen werden, da die Größen der Golomb-Codes von den Größen der komprimieren Daten abhängen. Um einen schnellen Zugriff zu ermöglichen, wird noch eine Index-Struktur benötigt. Hierzu wird der Bereich, in den die Hashfunktionen abbilden in I gleiche Teile zerlegt. Zusätzlich wird für jeden dieser Bereiche ein Pointer abgespeichert, der auf das Bit zeigt mit dem die Teilsequenz beginnt, in dem der aktuelle Wert gespeichert wird. Wird nun ein Hashwert berechnet, kann mit Hilfe dieses Pointers direkt der Bereich gefunden werden, in dem das Element komprimiert gespeichert ist. Folglich wird für einen schnellen Zugriff ein möglichst kleines I benötigt. Im Gegensatz hierzu wird mit einem möglichst großen I jedoch der Speicherverbrauch reduziert. Die Wahl der Größe von I ist folglich ein Kompromiss zwischen der Zugriffszeit und dem Speicherverbrauch.

Die Datenstruktur, die aus mit dem Golomb-Code komprimierten Sequenzen besteht, ist statisch, was bedeutet, dass alle Elemente bereits zu Beginn bekannt sein müssen und die Datenstruktur vollständig erstellt wurde, bevor Anfragen bearbeitet werden können.

4.2 Dynamische Golomb-Codierung

Es ist möglich, das Einfügen und Löschen von Elementen durch die Verwendung von Hashfunktionen zu unterstützen. Dadurch wird die Datenstruktur vollständig dynamisch. Das Einfügen wird durch eine weitere kleine dynamische Hashtabelle T_i realisiert. In diese werden die neu Elemente eingefügt. Es ist ausreichend in T_i die bitgenauen Positionen in der Haupttabelle zu speichern. Das Einfügen in T_i ist solange möglich, bis T_i eine bestimmte Größe erreicht. Ist dies der Fall, werden die Elemente in die Haupttabelle übernommen. Dies geschieht durch das neue Aufbauen der Haupttablle mit den neuen Elementen. Das Löschen lässt sich durch das Nutzen eines Buffers ermöglichen, in dem die Positionen der zu löschenden Elemente gespeichert werden. Dieser findet Berücksichtigung beim Neuaufbau der Haupttabelle.

5 Fazit

Wenn man die hier vorgestellten Varianten betrachtet, stellt man fest, dass die Wahl sehr von der geplanten Anwendung abhängt. So sind selbst Standard-Bloomfilter meistens noch eine solide Wahl. Diese erzeugen bei negativen Antworten aufgrund der Cache-Effizienz kurze Laufzeiten und sind selbst bei positiven Anfragen und dem Einfügen von Elementen noch verhältnismäßig schnell. Sind das Einfügen von Elementen und das Beantworten von positiven Anfragen eine der späteren Hauptaufgaben, bieten sich Blocked Bloomfilter an. Diese haben bei diesen Operationen wenig Cache-Misses und so eine deutlich kürzere Laufzeit und können zudem beispielsweise noch mit vorberechneten Bitmustern erweitert werden. Ist allerdings eine möglichst niedrige false positive Rate erwünscht, bietet sich die Multiblocking Variante an. Weiterhin existieren speichersparende Varianten, wie die in Abschnitt 4 vorgestellten Golomb-Codierten Bloomfilter. Hierbei ändert sich die Laufzeit nur minimal. Im Endeffekt hat jede Variante ihre Vor- und Nachteile.

Literatur

1 Andrei Broder and Michael Mitzenmacher. Network Applications of Bloom Filters: A Survey. *Internet Mathematics*, 1(4):485–509, 2004.

2 Deke Guo, Yunhao Liu, Xiangyang Li, and Panlong Yang. False negative problem of counting bloom filter. *IEEE Transactions on Knowledge and Data Engineering*, 22(5):651–664, 2010.

3 Achim Klenke. *Wahrscheinlichkeitstheorie*. Springer-Verlag, 1 edition, 2006.

4 Alistair Moffat and Andrew Turpin. *Compression and coding algorithms*. Springer Science & Business Media, 1 edition, 2002.

5 Anna Pagh, Rasmus Pagh, and S. Srinivasa Rao. An Optimal Bloom Filter Replacement. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '05, pages 823–829. Society for Industrial and Applied Mathematics, 2005.

6 Felix Putze, Peter Sanders, and Johannes Singler. *Cache-, Hash- and Space-Efficient Bloom Filters*, pages 108–121. Springer-Verlag, 2007.

7 Peter Sanders and Frederik Transier. Intersection in integer inverted indices. In *Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX)*, 1, pages 71–83. Society for Industrial and Applied Mathematics, 2007.