

Masterarbeit
Angewandte Informatik
Nr. AI-2024-MA-005

Dynamische Datenstrukturen unter Echtzeitbedingungen

B. Sc. Erik Bünnig

Abgabedatum: 09.10.2024

Prof. Dr. Kay Gürtzig
Dipl-Ing. Peter Brückner

Kurzfassung

Die Verwendung dynamischer Datenstrukturen unter Echtzeitbedingungen muss genau geprüft werden um sicher zustellen, dass ein Echtzeitsystem dessen vorgegebene Aufgaben in der erwarteten Zeit erfüllen kann. Ein solches Echtzeitsystem ist das Laufzeitsystem der T4gl-Programmiersprache, eine Domänenspezifische Sprache für Industrieprüfmaschinen. In dieser Arbeit wird untersucht, auf welche Weise die in T4gl verwendeten Datenstrukturen optimiert oder ausgetauscht werden können, um das Zeitverhalten unter Worst Case Bedingungen zu verbessern. Dabei werden vorallem persistente Datenstrukturen implementiert, getestet und verglichen.

Abstract

The usage of dynamic data structures under real time constraints must be analyzed precisely in order to ensure that a real time system can execute its tasks in the expected time. Such a real time system is the runtime of the T4gl programming language, a domain-specific language for industrial measurement machines. This thesis is concerned with the analysis, optimization and re-implementation of T4gl's data structures, in order to improve their worst-case time complexity. For this, various, but foremost persistent data structures are implemented, benchmarked and compared.

Inhaltsverzeichnis

Abbildungsverzeichnis	III
Tabellenverzeichnis	IV
Listingverzeichnis	V
1 Einleitung	1
1.1 Motivation	1
1.2 Aufbau der Arbeit	1
1.3 Komplexität	2
1.4 Legende & Konventionen	3
1.4.1 Texthervorhebungen	3
1.4.2 Grafiken	3
1.4.3 Notation	4
2 T4gl	5
2.1 Echtzeitanforderungen	5
2.2 T4gl-Arrays	6
3 Lösungsansätze	11
3.1 Move-Semantik	11
3.2 Annotationen	12
4 Persistente Datastrukturen	15
4.1 Kurzlebige Datenstrukturen	15
4.2 Persistenz und Kurzlebigkeit	16
4.3 Lösungsansatz	18
4.4 B-Bäume	18
4.5 2-3-Fingerbäume	19
4.6 Generische Fingerbäume	23
4.6.1 Baumtiefe	24
4.6.2 Über- & Unterlaufsicherheit	25
4.6.3 Suche	27
4.6.4 Push & Pop	28
4.6.5 Split & Concat	31
4.6.6 Insert & Remove	33
4.6.7 Echtzeitanalyse	34
4.7 SRB-Bäume	35
4.7.1 Schlüsseltypen	36
4.7.2 Zeitverhalten	36
4.7.3 Speicherauslastung	37
5 Implementierung	39
5.1 Von Haskell zu C++	39
5.1.1 Node	41
5.1.2 Digits	42
5.1.3 FingerTree	42

6 Analyse & Vergleich	45
6.1 Umgebung & Auswertung	45
6.2 Kontrollgruppen	45
6.2.1 QMap	45
6.2.2 Persistenter B-Tree	46
6.3 2-3-Fingerbäume	47
6.4 Vergleich	49
7 Fazit	51
7.1 Ergebnis	51
7.2 Optimierungen	51
7.2.1 Pfadkopie	51
7.2.2 Lazy-Evaluation	51
7.2.3 Generalisierung & Cache-Effizienz	51
7.2.4 Vererbung & Virtual Dispatch	52
7.2.5 Memory-Layout	53
7.2.6 Spezialisierte Allokatoren	53
7.2.7 Unsichtbare Persistenz	53
Literatur	VI
Danksagung	VIII
Eigenständigkeitserklärung	X

Abbildungsverzeichnis

Abbildung 2-1: Die drei Ebenen von T4gl-Arrays in verschiedenen Stadien der Datenteilung.	8
Abbildung 4-1: Der Aufbau eines Vektors in C++.	15
Abbildung 4-2: Eine Abfolge von Operationen auf persistenten verketteten Listen. .	16
Abbildung 4-3: Partielle Persistenz teilt zwischen mehreren Instanzen die Teile der Daten, welche sich nicht verändert haben, ähnlich der Persistenz in Abbildung 4-2. .	17
Abbildung 4-4: Ein interner B-Baum-Knoten eines 2-4-Baums.	19
Abbildung 4-5: Ein 2-3-Fingerbaum der Tiefe 3 und 21 Elementen.	20
Abbildung 4-6: Die möglichen Tiefen t für einen 2-3-Fingerbaum mit n Blattknoten. .	25
Abbildung 4-7: Ein SRB-Baum für die 16 bit Schlüssel und einen Zweigfaktor von 16. .	36
Abbildung 6-1: Vergleich der Lesezugriffe in Abhängigkeit der Anzahl der Elemente n . .	49
Abbildung 6-2: Vergleich der Schreibzugriffe in Abhängigkeit der Anzahl der Elemente n	50

Tabellenverzeichnis

Tabelle 1-1: Unvollständige Liste verschiedener Komplexitätsklassen in aufsteigender Reihenfolge.	2
Tabelle 1-2: Legende von Hervorhebungen im Lauftext.	3
Tabelle 1-3: Legende der Konventionen in Datenstrukturgrafiken.	3
Tabelle 1-4: Legende der Notationen in Pseudocode.	4
Tabelle 2-1: Semantische Analogien in C++ zu spezifischen Varianten von T4gl-Arrays.	6
Tabelle 4-1: Die Komplexitäten von 2-3-Fingerbäumen im Vergleich zu gewöhnlichen 2-3-Bäumen.	21
Tabelle 4-2: Beim Verpacken der Knoten ergibt sich eine Obergrenze für die Anzahl der verpackten Knoten.	32
Tabelle 6-1: Die verschiedenen Iterationen der QMap Benchmarks.	45
Tabelle 6-2: Die verschiedenen Iterationen der B-Baum Benchmarks.	46
Tabelle 6-3: Die verschiedenen Iterationen der 2-3-Fingerbaum Benchmarks.	48

Listingverzeichnis

Listing 2-1: Beispiele für Deklaration und Indizierung von T4gl-Arrays.	6
Listing 2-2: Demonstration von Referenzverhalten von T4gl-Arrays.	7
Listing 3-1: Demonstration der Move-Semantik in C++.	11
Listing 3-2: Demonstration der Move-Semantik in Rust.	11
Listing 3-3: Eine ArrayMax Annotation begrenzt das Array auf 1024 Elemente. . . .	12
Listing 4-1: Ein C++ Program, welches einen <code>std::vector</code> anlegt und mit Werten befüllt.	16
Listing 4-2: Die Definition von 2-3-Fingerbäumen in C++ übersetzt.	21
Listing 4-3: Die Definition von generischen Fingerbäumen in C++.	23
Listing 5-1: Nicht-reguläre Definition von Fingerbäumen.	40
Listing 5-2: Die Definition der Node-Klassen.	41
Listing 5-3: Die Definition der Digits-Klassen.	42
Listing 5-4: Die Definition der FingerTree-Klassen.	42

1 Einleitung

1.1 Motivation

In der Automobilindustrie gibt es verschiedene Systeme, welche die Automation von Prüfständen erleichtern oder ganz übernehmen. Eines dieser Systeme ist T4gl, eine Programmiersprache und gleichnamiges Laufzeitsystem zur Interpretation von Testskripten in Industrieprüfanlagen wie End of Line- aber auch Versuchsprüfständen. Bei diesen Anlagen werden verschiedene Tests und Messvorgänge durchgeführt, welche bestimmte Zeitanforderungen aufweisen. Können diese Zeitanforderungen nicht eingehalten werden, müssen Testvorgänge verworfen und wiederholt werden. Daher ist es essentiell, dass das T4gl-Laufzeitsystem in der Lage ist, die erwarteten Testvorgänge innerhalb einer festgelegten Zeit abzuarbeiten, ungeachtet der anfallenden Testdatenmengen. T4gl ist eine Hochlevel-Programmiersprache, Speicherverwaltung oder Synchronization werden vom Laufzeitsystem übernommen und müssen in den meisten Fällen nicht vom Programmierer beachtet werden. Wie in fast allen Hochlevel-Programmiersprachen, gibt es in T4gl dynamische Datenstrukturen zur Verwaltung von mehreren Elementen. Diese werden in T4gl als Arrays bezeichnet und bieten eine generische Datenstruktur für Sequenzen, Tabellen, ND-Arrays oder Queues. Intern wird die gleiche Datenstruktur für alle Anwendungsfälle verwendet, in welchen ein T4gl-Programmierer eine dynamische Datenstruktur benötigt, ungeachtet der individuellen Anforderungen. Aus der Interaktion verschiedener Features des Laufzeitsystems und der Standardbibliothek der Programmiersprache kommt es, zusätzlich dazu, zu unnötig häufigen Kopien von Daten. Schlimmer noch, durch die jetzige Implementierung wächst die Länge von Kopiervorgängen der T4gl-Arrays proportional zur Anzahl der darin verwalteten Elemente. Bei diese Kopien kann es sich um T4gl-Arrays mit fünfzig Elementen oder Arrays mit fünf-Millionen Elementen handeln. Durch diese unzureichende Flexibilität bei der Wahl der Datenstruktur ist es dem T4gl-Programmierer nicht möglich, die Nutzung der Datenstruktur hinreichend auf die jeweiligen Anwendungsfälle zu optimieren. Das Laufzeitsystem kann die gestellten Zeitanforderungen nicht garantiert einhalten, da die Anzahl der Elemente in T4gl-Arrays erst zur Laufzeit bekannt ist.

Das Hauptziel dieser Arbeit ist die Verbesserung der Latenzen des T4gl-Laufzeitsystems durch Analyse und gezielte Verbesserung der Datenverwaltung von T4gl-Arrays. Dabei werden verschiedene Lösungsansätze evaluiert, teilweise implementiert, getestet und verglichen.

1.2 Aufbau der Arbeit

Dieses Kapitel führt das zu lösende Problem ein und beschreibt die Arbeit selbst. Dabei werden Notation und Konventionen sowie fachspezifisches Grundwissen vermittelt. Kapitel 2 beschreibt T4gl und dessen Komponenten genauer, sowie die Implementierung der T4gl-Arrays. Darin wird erarbeitet, wie es zu den meist unnötigen Kopiervorgängen kommt und in welchen Fällen diese auftreten. In Kapitel 3 werden verschiedene Veränderungen an der Programmiersprache selbst erörtert und warum diese unzureichend oder anderweitig unerwünscht sind. Dazu zählen verschiedene neue syntaktische Konstrukte zur statischen Analyse oder das Optimierten der Implementierung, sowie die Veränderung bestehender Semantik existierender Sprachkonstrukte. Kapitel 4 führt einen Begriff der

Persistenz im Sinne von Datenstrukturen ein und befasst sich mit verschiedenen Datenstrukturen, durch welche die zeitliche Komplexität von Kopien der T4gl-Arrays reduziert werden kann. Im Anschluss wird in Kapitel 5 die Implementierung einer ausgewählten Datenstruktur beschreiben. Die daraus folgenden Implementierungen werden in Kapitel 6 getestet und verglichen. Zu guter Letzt wird in Kapitel 7 das Resultat der Arbeit evaluiert und weiterführende Aufgaben beschrieben.

1.3 Komplexität

Im Verlauf dieser Arbeit wird oft von Zeit- und Speicherkomplexität gesprochen. Komplexitätstheorie befasst sich mit der Komplexität von Algorithmen und algorithmischen Problemen, vor allem in Bezug auf Speicherverbrauch und Bearbeitungszeit. Dabei sind folgende Begriffe relevant:

Zeitkomplexität Zeitverhalten eines Algorithmus über eine Menge von Daten in Bezug auf die Anzahl dieser [Cor+09, S. 44].

Speicherkomplexität Der Speicherbedarf eines Algorithmus zur Bewältigung eines Problems [Cor+09, S. 44] im Bezug auf die Problemgröße. Wird auch für den Speicherbedarf von Datenstrukturen verwendet.

Amortisierte Komplexität Bezüglich einer Sequenz von n Operationen mit einer Gesamtdauer von $T(n)$, gibt die amortisierte Komplexität den Durchschnitt $T(n)/n$ einer einzelnen Operation an [Cor+09, S. 451].

LANDAU-Symbole umfassen Ausdrücke zur Klassifizierung der asymptotischen Komplexität von Funktionen und Algorithmen. Im folgenden werden Variationen der KNUTH'schen Definitionen verwendet [Knu76, S. 19] [Cor+09, S. 44-48], sprich:

$$\begin{aligned} O(f) &= \{g : \mathbb{N} \rightarrow \mathbb{N} \mid \exists n_0, c > 0 \quad \forall n \geq n_0 \quad 0 \leq g(n) \leq cf(n)\} \\ \Omega(f) &= \{g : \mathbb{N} \rightarrow \mathbb{N} \mid \exists n_0, c > 0 \quad \forall n \geq n_0 \quad 0 \leq cf(n) \leq g(n)\} \\ \Theta(f) &= \{g : \mathbb{N} \rightarrow \mathbb{N} \mid \exists n_0, c_1, c_2 > 0 \quad \forall n \geq n_0 \quad c_1 f(n) \leq g(n) \leq c_2 f(n)\} \end{aligned} \quad (1.1)$$

Bei der Verwendung von LANDAU-Symbolen steht die Variable n für die Größe der Daten, welche für die Laufzeit eines Algorithmus relevant sind. Bei Operationen auf Datenstrukturen entspricht die Größe der Anzahl der verwalteten Elemente in der Struktur.

Tabelle 1-1: Unvollständige Liste verschiedener Komplexitätsklassen in aufsteigender Reihenfolge, dabei steht α für ein Symbol aus Gleichung (1.1).

Komplexität	Beschreibung
$\alpha(1)$	Konstante Komplexität, unabhängig von der Menge der Daten n
$\alpha(\log n)$	Logarithmische Komplexität über der Menge der Daten n
$\alpha(n)$	Lineare Komplexität über der Menge der Daten n
$\alpha(n^k)$	Polynomialkomplexität des Grades k über der Menge der Daten n
$\alpha(k^n)$	Exponentialkomplexität über der Menge der Daten n zur Basis k

$O(f)$ beschreibt die Menge der Funktionen, welche die obere asymptotische Grenze f haben. Gleichmaßen gibt $\Omega(f)$ die Menge der Funktionen an, welche die untere asymptotische Grenze f haben. $\Theta(f)$ ist die Schnittmenge aus $O(f)$ und $\Omega(f)$ [Cor+09, S. 48, Theorem 3.1]. Trotz der Definition der Symbole in Gleichung (1.1) als Mengen schreibt man oft $g(n) = O(f(n))$ statt $g(n) \in O(f(n))$ [Knu76, S. 20]. Die Vorliegende Arbeit hält

sich an diese Konvention. Tabelle 1-1 zeigt verschiedene Komplexitäten in aufsteigender Ordnung der Funktion $f(n)$. Unter Betrachtung der asymptotischen Komplexität werden konstante Faktoren und Terme geringerer Ordnung generell ignoriert, sprich $g(n) = 2n^2 + n = O(n^2)$.

1.4 Legende & Konventionen

Die folgenden Konventionen und Notationen werden während der Arbeit verwendet. Sollten bestimmte Teile der Arbeit diesen Konventionen nicht folgen, sind diese Abweichungen im umliegenden Text beschrieben.

1.4.1 Texthervorhebungen

Wenn bestimmte Variablen, Werte, Typen oder Funktionen aus umliegenden Abbildungen referenziert werden, sind diese auf verschiedene Weise hervorgehoben. Tabelle 1-2 zeigt verschiedene Hervorhebungen.





Tabelle 1-2: Legende von Hervorhebungen im Lauftext.

Hervorhebung	Beschreibung
<code>func</code> , <code>nullptr</code> , <code>if</code>	Hervorhebungen von code- oder programmiersprachen-spezifischen Funktionen, Typen oder Variablen, verwendet Monospace-Schriftart und Syntax-Highlighting.
<code>FingerTree</code>	Typ oder Konstruktor in Pseudocode.
<code>concat</code>	Funktion in Pseudocode.

1.4.2 Grafiken

Tabelle 1-3 beschreibt die Konventionen für Grafiken, vor allem Grafiken zu Baum- oder Listenstrukturen. Diese Konventionen sollen vor allem Konzepte der Persistenz (siehe Kapitel 4) vereinfachen.

Tabelle 1-3: Legende der Konventionen in Datenstrukturgrafiken.

Umrandung	Beschreibung
	Geteilte Knoten, d.h. Knoten, welche mehr als einen Referenten haben.
	Kopierte (nicht geteilte) Knoten, d.h. Knoten, welche durch eine Operation kopiert wurden statt geteilt zu werden, z.B. durch eine Pfadkopie.
	Visuelle Orientierungshilfe für folgende Abbildungen, kann hypothetischen oder gelöschten Knoten darstellen. Die genaue Bedeutung ist im umliegenden Text beschrieben.
	Instanzknoten, d.h. Knoten, welche nur Verwaltungsinformationen enthalten, wie die Instanz eines <code>std::vector</code> .

1.4.3 Notation

In Pseudocode werden sowohl mathematische Symbolik, wie die zur Kardinalität von Mengen $|M|$, als auch Programmierkonventionen übernommen, zum Beispiel die Syntax für Feldzugriffe $x.y$.

Tabelle 1-4: Legende der Notationen in Pseudocode.

Syntax	Beschreibung
$x.y$	Feldzugriff, x ist eine Datenstruktur mit einem Feld y , dann gibt $x.y$ den Wert des Feldes y in x zurück.
$ x $	Längenzugriff, ist x ein Vektor, ein Baum, eine Hashtabelle, etc. (ein Kontainertyp), gibt $ x $ die Anzahl der Elemente in x zurück.
$[x, y, \dots]$	Sequenzkonstruktor, erzeugt eine Sequenz aus den Elementen x, y , usw. Der Typ von Sequenzen wird als $[\text{Typ}]$ geschrieben, wobei Typ der Typ der Elemente in der Sequenz ist.

2 T4gl

T4gl (*engl.* **T**esting **4**th **G**eneration **L**anguage) ist ein proprietäres Softwareprodukt zur Entwicklung von Testsoftware für Industrieprüfanlagen wie die LUB (**L**ow **S**peed **U**niformity and **B**alance), HSU (**H**igh **S**peed **U**niformity and **B**alance) oder vielen Weiteren. T4gl steht bei der Brückner und Jarosch Ingenieurgesellschaft mbH (BJ-IG) unter Entwicklung und umfasst die folgenden Komponenten:

- Programmiersprache
 - Anwendungsspezifische Features
- Compiler
 - Statische Analyse
 - Übersetzung in Instruktionen
- Laufzeitsystem
 - Ausführen der Instruktionen
 - Scheduling von Green Threads
 - Bereitstellung von maschinen- oder protokollspezifischen Schnittstellen

Wird ein T4gl-Script dem Compiler übergeben, startet dieser zunächst mit der statischen Analyse. Bei der Analyse der Skripte werden bestimmte Invarianzen geprüft, wie die statische Länge bestimmter Arrays, die Typsicherheit und die syntaktische Korrektheit des Scripts. Nach der Analyse wird das Script in eine Sequenz von *Microsteps* (atomare Instruktionen) kompiliert. Im Anschluss führt das Laufzeitsystem die kompilierten *Microsteps* aus, verwaltet Speicher und Kontextwechsel der *Microsteps* und stellt die benötigten Systemschnittstellen zur Verfügung.

2.1 Echtzeitanforderungen

Je nach Anwendungsfall werden an das T4gl-Laufzeitsystem Echtzeitanforderungen gestellt.

„Unter Echtzeit versteht man den Betrieb eines Rechensystems, bei dem Programme zur Verarbeitung anfallender Daten ständig betriebsbereit sind, derart, dass die Verarbeitungsergebnisse innerhalb einer vorgegebenen Zeitspanne verfügbar sind. Die Daten können je nach Anwendungsfall nach einer zeitlich zufälligen Verteilung oder zu vorherbestimmten Zeitpunkten anfallen.“

— Peter Scholz [[Sch05](#), S. 39] unter Verweis auf DIN 44300

Ist ein Echtzeitsystem nicht in der Lage, eine Aufgabe in der vorgegebenen Zeit vollständig abzuarbeiten, spricht man von Verletzung der Echtzeitbedingungen, welche an das System gestellt wurden. Je nach Strenge der Anforderungen lassen sich Echtzeitsysteme in drei verschiedene Stufen einteilen:

Weiches Echtzeitsystem Die Verletzung der Echtzeitbedingungen führt zu degenerierter, aber nicht zerstörter Leistung des Echtzeitsystems und hat *keine* katastrophalen Folgen [[LO11](#), S. 6].

Festes Echtzeitsystem Eine geringe Anzahl an Verletzungen der Echtzeitbedingungen hat katastrophale Folgen für das Echtzeitsystem [[LO11](#), S. 7].

Hartes Echtzeitsystem Eine einzige Verletzung der Echtzeitbedingungen hat katastrophale Folgen für das Echtzeitsystem [[LO11](#), S. 6].

T4gl ist ein weiches Echtzeitsystem. Für Anwendungsfälle, bei denen Echtzeitanforderungen an T4gl gestellt werden, darf die Nichteinhaltung dieser Anforderungen keine katastrophalen Folgen haben (Personensicherheit, Prüfstandssicherheit, Prüflingssicherheit). In diesem Fall sind im schlimmsten Fall nur die Testergebnisse ungültig und müssen verworfen werden.

2.2 T4gl-Arrays

Bei T4gl-Arrays handelt es sich um mehrdimensionale assoziative Arrays mit Schlüsseln, welche eine voll definierte Ordnungsrelation haben. Um ein Array in T4gl zu deklarieren, wird mindestens ein Schlüssel- und ein Werttyp benötigt. Auf den Werttyp folgt in eckigen Klammern eine komma-separierte Liste von Schlüsseltypen. Die Indizierung erfolgt wie in der Deklaration durch eckige Klammern, es muss aber nicht für alle Schlüssel ein Wert angegeben werden. Bei Angabe von weniger Schlüsseln als in der Deklaration, wird eine Referenz auf einen Teil des Arrays zurückgegeben. Sprich, ein Array des Typs `T[U, V, W]` welches mit `[u]` indiziert wird, gibt ein Unter-Array des Typs `T[V, W]` zurück, ein solches Unter-Array kann referenziert, aber nicht gesetzt werden. Wird in der Deklaration des Arrays ein Ganzzahlwert statt eines Typs angegeben (z.B. `T[10]`), wird das Array mit fester Größe und durchlaufenden Indizes (0 bis 9) als Schlüssel angelegt und mit Standardwerten befüllt. Für ein solches Array können keine Schlüssel hinzugefügt oder entnommen werden.

Listing 2-1: Beispiele für Deklaration und Indizierung von T4gl-Arrays.

```

1  String[Integer] map
2  map[42] = "Hello World!"
3
4  String[Integer, Integer] nested
5  nested[-1, 37] = "T4gl is wacky!"
6
7  String[10] staticArray
8  staticArray[9] = "Truly wacky!"

```

Bei den in Listing 2-1 gegebenen Deklarationen werden, je nach den angegebenen Typen, verschiedene Datenstrukturen vom Laufzeitsystem gewählt. Diese ähneln den C++-Varianten in Tabelle 2-1, wobei `T` und `U` Typen sind und `N` eine Zahl aus \mathbb{N}^+ . Die Deklaration von `staticArray` weist den Compiler an, ein T4gl-Array mit 10 Standardwerten für den `String` Typ (die leere Zeichenkette "") für die Schlüssel 0 bis einschließlich 9 anzulegen. Es handelt sich um eine Sonderform des T4gl-Arrays, welches eine dichte festgelegte Schlüsselverteilung hat (es entspricht einem gewöhnlichen Array).

Tabelle 2-1: Semantische Analogien in C++ zu spezifischen Varianten von T4gl-Arrays.

Signatur	C++ Analogie
<code>T[N]</code> name	<code>std::array<T, N></code>
<code>T[U]</code> name	<code>std::map<U, T></code> name
<code>T[U, N]</code> name	<code>std::map<U, std::array<T, N>></code> name
<code>T[N, U]</code> name	<code>std::array<std::map<U, T>, N></code> name

Signatur	C++ Analogie
...	...

Die Datenspeicherung im Laufzeitsystem kann nicht direkt ein statisches Array (`std::array<T, 10>`) verwenden, da T4gl nicht direkt in C++ übersetzt und kompiliert wird. Intern werden, je nach Schlüsseltyp, pro Dimension entweder ein Vektor oder ein geordnetes assoziatives Array angelegt. T4gl-Arrays verhalten sich wie Referenztypen, wird ein Array `a2` durch ein anderes Array `a1` initialisiert, teilen sich diese die gleichen Daten. Schreibzugriffe in einer Instanz sind auch in der anderen lesbar (demonstriert in Listing 2-2).

Listing 2-2: Demonstration von Referenzverhalten von T4gl-Arrays.

```

1  String[10] a1
2  String[10] a2 = a1
3
4  array1[0] = "Hello World!"
5  // array1[0] == array2[0]
```

Im Gegensatz zu dem Referenzverhalten der Arrays aus Sicht des T4gl-Programmierers steht die Implementierung durch QMap. Bei diesen handelt es sich um Copy-On-Write (CoW) Datenstrukturen, mehrere Instanzen teilen sich die gleichen Daten und erlauben zunächst nur Lesezugriffe darauf. Muss eine Instanz einen Schreibzugriff durchführen, wird vorher sichergestellt, dass diese Instanz der einzige Referent der Daten ist, wenn nötig durch eine Kopie der gesamten Daten. Dadurch sind Kopien von QMap initial sehr effizient, es muss lediglich die Referenzzahl der Daten erhöht werden. Die Kosten der Kopie zeigt sich erst dann, wenn ein Schreibzugriff nötig ist.

Damit sich T4gl-Arrays trotzdem wie Referenzdaten verhalten, teilen sie sich nicht direkt die Daten mit CoW-Semantik, sondern QMap durch eine weitere Indirektionsebene. T4gl-Arrays bestehen aus Komponenten in drei Ebenen:

1. T4gl: Die Instanzen aus Sicht des T4gl-Programmierers (z.B. die Variable `a1` in Listing 2-2).
2. Indirektionsebene: Die Daten aus Sicht des T4gl-Programmierers, die Ebene zwischen T4gl-Array Instanzen und deren Daten. Dabei handelt es sich um Qt-Datentypen wie QMap oder QVector.
3. Speicherebene: Die Daten aus Sicht des T4gl-Laufzeitsystems. Diese Ebene ist für den T4gl-Programmierer unsichtbar.

Zwischen den Ebenen 1 und 2 kommt geteilte Schreibfähigkeit durch Referenzzählung zum Einsatz, mehrere T4gl-Instanzen teilen sich eine Qt-Instanz und können von dieser lesen, als auch in sie schreiben, ungeachtet der Anzahl der Referenten. Zwischen den Ebenen 2 und 3 kommt CoW + Referenzzählung zum Einsatz, mehrere Qt-Instanz teilen sich die gleichen Daten, Schreibzugriffe auf die Daten sorgen vorher dafür, dass die Qt-Instanz der einzige Referent ist, wenn nötig durch eine Kopie. Wir definieren je nach Tiefe drei Typen von Kopien:

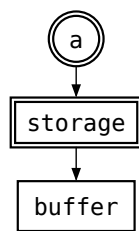
Typ-1 (flache Kopie) Eine Kopie der T4gl-Instanz erstellt lediglich eine neue Instanz, welche auf die gleiche Qt-Instanz zeigt. Dies wird in T4gl durch Initialisierung von Arrays durch existierende Instanzen oder die Übergabe von Arrays an normale Funktionen hervorgerufen. Eine flache Kopie ist immer eine $\Theta(1)$ -Operation.

Typ-2 Eine Kopie der T4gl-Instanz **und** der Qt-Instanz, welche beide auf die gleichen Daten zeigen. Wird eine tiefe Kopie durch das Laufzeitsystem selbst hervorgerufen, erfolgt die Kopie der Daten verspätet beim nächsten Schreibzugriff auf eine der Qt-Instanzen. Halbtiefe Kopien sind Operationen konstanter Zeitkomplexität $\Theta(1)$. Aus einer halbtiefen Kopie und einem Schreibzugriff folgt eine volltiefen Kopie.

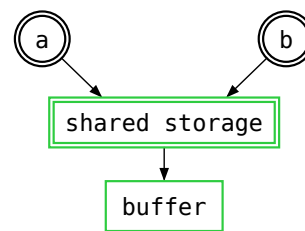
Typ-3 (tiefe Kopie) Eine Kopie der T4gl-Instanz, der Qt-Instanz **und** der Daten. Beim expliziten Aufruf der Methode `clone` durch den T4gl Programmierer werden die Daten ohne Verzögerung kopiert. Tiefe Kopien sind Operationen linearer Zeitkomplexität $\Theta(n)$.

Bei einer Typ-1-Kopie der Instanz a in Abbildung 2-1a ergibt sich die Instanz b in Abbildung 2-1b. Eine Typ-2-Kopie hingegen führt zur Instanz c in Abbildung 2-1c. Obwohl eine tiefe Kopie zunächst nur auf Ebene 1 und 2 Instanzen kopiert, erfolgt die Typ-3-Kopie der Daten auf Ebene 3 beim ersten Schreibzugriff einer der Instanzen (Abbildung 2-1d).

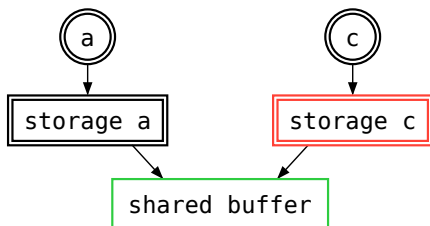
In seltenen Fällen kann es dazu führen, dass wie in Abbildung 2-1c eine Typ-2-Kopie angelegt wurde, aber nie ein Schreibzugriff auf a oder c durchgeführt wird, während beide Instanzen existieren. Sprich, es kommt zur Typ-2-Kopie, aber nicht zur Typ-3-Kopie. Diese Fälle sind nicht nur selten, sondern meist auch Fehler der Implementierung. Bei korrektem Betrieb des Laufzeitsystems sind Typ-2-Kopien kurzlebig und immer von Typ-3-Kopien gefolgt, daher betrachten wir im folgenden auch Typ-2-Kopien als Operationen linearer Zeitkomplexität $\Theta(n)$, da diese bei korrektem Betrieb fast immer zu Typ-3 Kopien führen.



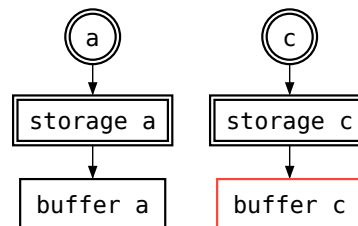
(a) Ein T4gl Array nach Initialisierung.



(b) Zwei T4gl-Arrays teilen sich eine C++ Instanz nach Typ-1 Kopie.



(c) Zwei T4gl-Arrays teilen sich die gleichen Daten nach Typ-2 Kopie.



(d) Zwei T4gl-Arrays teilen sich keine Daten nach Typ-2 Kopie und Schreibzugriff.

Abbildung 2-1: Die drei Ebenen von T4gl-Arrays in verschiedenen Stadien der Datenteilung.

Ein kritischer Anwendungsfall für T4gl-Arrays ist die Übergabe einer rollenden Historie von Werten einer Variable. Wenn diese vom Laufzeitsystem erfassten Werte an den T4gl-Programmierer übergeben werden, wird eine Typ-2-Kopie erstellt. Die T4gl-Instanz, welche an den Programmierer übergeben wird, sowie die interne Qt-Instanz teilen sich Daten, welche vom Laufzeitsystem zwangsläufig beim nächsten Schreibzugriff kopiert

werden müssen. Es kommt zur Typ-3-Kopie, und, daraus folgend, zu einem nicht-trivialem zeitlichem Aufwand von $\Theta(n)$. Das gilt für jeden ersten Schreibzugriff, welcher nach einer Übergabe der Daten an den T4gl-Programmierer erfolgt.

3 Lösungsansätze

Initial wurden zur Reduzierung der Latenzen im T4gl-Laufzeitsystem verschiedene Veränderungen auf Sprachebene, wie die Einführung von Move-Semantik oder neuer Syntax/Annotationen in T4gl-Arrays, in Erwägung gezogen. In den folgenden Abschnitten werden diese kurz umrissen, warum diese für das Problem unzureichend sind und wie diese Erkenntnisse zum Lösungsansatz in Kapitel 4 führten.

3.1 Move-Semantik

Eine der möglichen Lösungen zur Reduzierung von Array-Kopien in T4gl ist die Einführung einer Move-Semantik ähnlich derer in Rust oder `std::move` in C++. Durch explizite oder implizite *Moves* reduziert man die Anzahl an Kopien von Daten, in den Fällen, wo Daten explizit übergeben und nicht kopiert werden sollen. Soll zum Beispiel ein Array an eine Funktion übergeben und danach nicht außerhalb dieser Funktion verwendet werden, kann statt einer Kopie ein *Move* durchgeführt werden. Listing 3-1 zeigt die Initialisierung dreier Vektoren. Zuerst wird der Vektor `foo` angelegt. Danach wird der Vektor `foo` kopiert, um `bar` zu initialisieren. Darauf hin wird `baz` durch den *Move* von `bar` initialisiert. Im Anschluss sind `foo` und `baz` mit der Sequenz `[0, 1, 2]` belegt, während `bar` leer ist.

Listing 3-1: Demonstration der Move-Semantik in C++.

```
1  std::vector<int> foo = {0, 1, 2};
2  std::vector<int> bar = foo;           // copy!
3  std::vector<int> baz = std::move(bar); // no copy!
```

In C++ wird durch den Aufruf von `std::move` ein *MoveConstructor* aufgerufen, dessen Aufgabe es ist, aus einer alten Instanz eines Typs Daten in die neue Instanz zu übertragen, ähnlich wie bei einer Kopie. Im Gegensatz zu einer Kopie, wird die alte Instanz aber in einen undeterminierten aber validen Zustand gesetzt, dieser ist implementierungsdefiniert, aber meistens der Standardwert eines Typs. Bei Klassen wie `shared_ptr` oder `unique_ptr` ist das das Äquivalent zu `nullptr`, im Falle des Vektors wird beim *Move* der Daten von `bar` zu `baz` die Instanz `bar` zum leeren Vektor ohne dass die eigentlichen Elemente im Vektor verschoben werden.

Listing 3-2: Demonstration der Move-Semantik in Rust.

```
1  let foo = vec![0, 1, 2];
2  let bar = foo.clone(); // copy!
3  let baz = bar;         // no copy!
```

In Listing 3-2 ist die Rust-Variante von Listing 3-1 zu sehen. Der Unterschied liegt darin, dass `bar` danach nicht leer ist, sondern für die weitere Referenzierung ungültig. Zugriffe auf die Variable `bar` sind nach der dritten Zeile nicht mehr erlaubt und werden zur Kompilierzeit geprüft.

Beide Varianten ermöglichen es, potentiell teure Kopien zu vermeiden, indem die Verantwortung über Daten des Programms von einem Punkt zum anderem übertragen wird. Die C++-Variante kann in T4gl eingeführt werden, ohne das Verhalten alter Skripte zu

verändern. Zuweisungen in T4gl würden weiterhin intern Daten teilen, statt direkt Kopien anzulegen. Neue Skripte können durch explizitem *Move* von Instanzen Kopien vermeiden. Daraus folgt aber auch, das alte Skripte keinen Nutzen daraus ziehen, ohne dass diese überarbeitet werden müssen. Bei Unachtsamkeit durch den T4gl-Programmierer können fälschlicherweise Instanzen verwendet werden, welche durch einen *Move* deinitialisiert wurden. Die Rust-Variante verhindert durch die Invalidierung von Variablen, welche auf der rechten Seite eines *Move-Assignments* standen, dass solche Instanzen verwendet werden, kann aber durch diese Restriktion Migrationsaufwand von Projekten voraussetzen, welcher bei der C++-Variante optional wäre. In beiden Fällen sind also Codeänderungen bei T4gl-Projekten vonnöten, um von der verringerten Anzahl an Kopien zu profitieren. Ungeachtet dessen löst dieses veränderte Verhalten nicht das eigentliche Problem, wenn eine Kopie durchgeführt werden muss, ist diese immer noch teuer. Außerdem benötigt die Rust-Variante nicht-triviale Änderungen an der Analyse-Stufe des T4gl-Kompilers.

3.2 Annotationen

Wie im vorherigen Abschnitt beschrieben, müssen die Kosten von Kopien selbst reduziert werden. Kopien zu vermeiden bekämpft nur ein Symptom der Implementierung, ohne die Ursache zu beheben. Wenn T4gl-Arrays zu groß sind, sind Kopien rechentechnisch zu aufwendig.

Listing 3-3: Eine ArrayMax Annotation begrenzt das Array auf 1024 Elemente.

```
1 @ArrayMax(1024)
2 String[Integer] array
```

Ein alternativer Lösungsansatz zur Reduzierung der Kosten von Kopien sind Annotationen, welche die Größe von T4gl-Arrays begrenzen. Listing 3-3 zeigt, wie eine solche Annotation verwendet werden könnte. Es gibt verschiedene Möglichkeiten, solche Annotationen zu implementieren:

1. Überprüfung zur Analyse-Stufe: Ähnlich der statischen T4gl-Arrays (z.B. `String[10]`) müsste überprüft werden, dass die Länge des Arrays 1024 nicht übersteigt.
2. Überprüfung zur Laufzeit: Überschreitung der Maximallänge würde einen Laufzeitfehler hervorrufen.

Das Verhalten von statischen Arrays ist verhältnismäßig simpel, die Funktionen `insert` oder `remove` können zu Kompilizerzeit schon nicht aufgerufen werden und Elemente sind durchgängig initialisiert. Bei Dynamischen Arrays ist das anders. Um die Überschreitung der Maximallänge zur Analyse-Stufe zu überprüfen, müsste der Compiler zur Analyse-Stufe Annahmen über unbekannte Größen treffen, von welchen die Anzahl der Elemente abhängen. Hängen diese von Laufzeitwerten wie *I/O* ab, dann ist die Analyse in den meisten Fällen unmöglich. Es ist auch nicht trivial zu überprüfen, wann ein T4gl-Array durch ein anderes initialisiert wurde, welches eine solche Annotation hat. Die Zuweisung von verschiedenen T4gl-Instanzen kann ebenfalls von Laufzeitvariablen abhängen.

Bei Überprüfung zur Laufzeit wird zwar verhindert, dass T4gl-Arrays eine Maximallänge überschreiten, das erfolgt aber durch einen Laufzeitfehler. Wird ein Laufzeitfehler nicht verarbeitet, beendet dieser das Programm.

Ähnlich der *Move*-Semantik in Abschnitt 3.1 müssen Annotationen explizit zu alten Projekten hinzugefügt werden, um von diesen zu profitieren. Eine Analyse ist beinahe

unmöglich und Laufzeitfehler sind in den meisten Fällen schlimmer als Verzögerungen. Des Weiteren verhindern diese nur Kopien von Arrays mit mehr Elementen als die Annotation angibt, wird eine Annotation aber mit einem Wert wie 2^{64} verwendet, ist diese nutzlos.

4 Persistente Datastrukturen

Die in Kapitel 3 beschriebenen Lösungsansätze haben alle etwas gemeinsam, sie bekämpfen Symptome statt Ursachen. Die Häufigkeit der Kopien kann reduziert werden, aber nie komplett eliminiert, da es eine fundamentale Operation ist. Es gilt also, ungeachtet der Häufigkeit einer solchen Operation, auf Seiten der Implementierung von T4gl-Arrays, Kopien von Daten nur dann anzufertigen, wenn das vonnöten ist. In Abschnitt 2.2 wurde beschrieben, wie T4gl-Arrays bereits Daten teilen und nur bei Schreibzugriffen auf geteilte Daten Kopien erstellen. Das fundamentale Problem ist, dass bei Schreibzugriffen auf geteilte Daten, ungeachtet der Ähnlichkeit der Daten nach dem Schreibzugriff, alle Daten kopiert werden.

Im Folgenden wird beschrieben, wie effiziente Datenteilung durch Persistenz [FK24] auf Speicherebene der T4gl-Arrays die Komplexität von Typ-3-Kopien auf logarithmische Komplexität senken kann. Es werden die Begriffe der Persistenz und Kurzlebigkeit im Kontext von Datenstrukturen eingeführt, sowie der aktuelle Stand persistenter Datenstrukturen beschrieben. Im Anschluss werden persistente Datenstrukturen untersucht, welche als Alternative der jetzigen Qt-Instanzen verwendet werden können.

4.1 Kurzlebige Datenstrukturen

Um den Begriff der Persistenz zu verstehen, müssen zunächst kurzlebige oder gewöhnliche Datenstrukturen entgegengestellt werden. Dafür betrachten wir den Vektor, ein dynamisches Array.

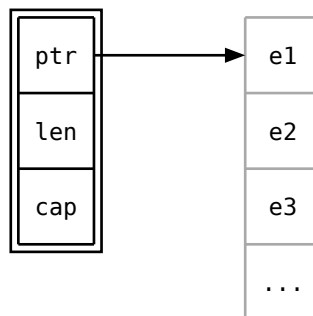


Abbildung 4-1: Der Aufbau eines Vektors in C++.

Ein Vektor besteht in den meisten Fällen aus 3 Komponenten, einem Pointer `ptr` auf die Daten des Vektors (in grau ■ umrandet), der Länge `len` des Vektors und die Kapazität `cap`. Abbildung 4-1 zeigt den Aufbau eines Vektors in C++¹. Operationen am Vektor wie `push_back`, `pop_back` oder Indizierung arbeiten direkt im Speicher, auf welchen `ptr` verweist. Reicht der Speicher nicht aus, wird er erweitert und die Daten werden aus dem alten Speicher in den neuen Speicher verschoben. Die in Listing 4-1 gezeigten Aufrufe zu `push_back` schreiben die Werte direkt in den Speicher des Vektors. Zu keinem Zeitpunkt gibt es mehrere Instanzen, welche auf die gleichen Daten zeigen. Wird der Vektor `vec` auf `other` zugewiesen, werden alle Daten kopiert, die Zeitkomplexität der Kopie selbst ist proportional zur Anzahl der Elemente $\Theta(n)$.

¹Der gezeigte Aufbau ist nicht vom Standard [JTC20] vorgegeben, manche Implementierungen speichern statt der Kapazität einen `end` Pointer, aus welchem die Kapazität errechnet werden kann. Funktionalität und Verhalten sind allerdings gleich.

Listing 4-1: Ein C++ Program, welches einen `std::vector` anlegt und mit Werten befüllt.

```

1  #import <vector>
2
3  int main() {
4      std::vector<int> vec;
5
6      vec.push_back(3);
7      vec.push_back(2);
8      vec.push_back(1);
9
10     std::vector<int> other = vec;
11
12     return 0;
13 }
```

4.2 Persistenz und Kurzlebigkeit

Wenn eine Datenstruktur bei Schreibzugriffen die bis dahin bestehenden Daten nicht verändert, gilt diese als persistent/langlebig [KT96, S. 202]. Daraus folgt, dass viele Instanzen sich gefahrlos die gleichen Daten teilen können. In den Standardbibliotheken verschiedener Programmiersprachen hat sich für dieses Konzept der Begriff *immutable* (engl. unveränderbar) durchgesetzt. Im Gegensatz dazu stehen Datenstrukturen, welche bei Schreibzugriffen ihre Daten direkt beschreiben. Diese gelten als kurzlebig. Dazu gehört zum Beispiel der Vektor, beschrieben in Abschnitt 4.1. Persistente Datenstrukturen erstellen meist neue Instanzen für jeden Schreibzugriff, welche die Daten der vorherigen Instanz wenn möglich teilen. Die bis dato verwendeten QMaps sind persistent in dem Sinne, dass keine Instanz die Schreibzugriffe anderer Instanzen sehen kann, da diese immer vorher ihre geteilten Daten kopieren. Allerdings können clever aufgebaute Datenstrukturen bei einem Schreibzugriff dennoch Teile ihrer Daten mit anderen Instanzen teilen. Ein gutes Beispiel dafür bietet die einfach verkettete Liste (Abbildung 4-2).

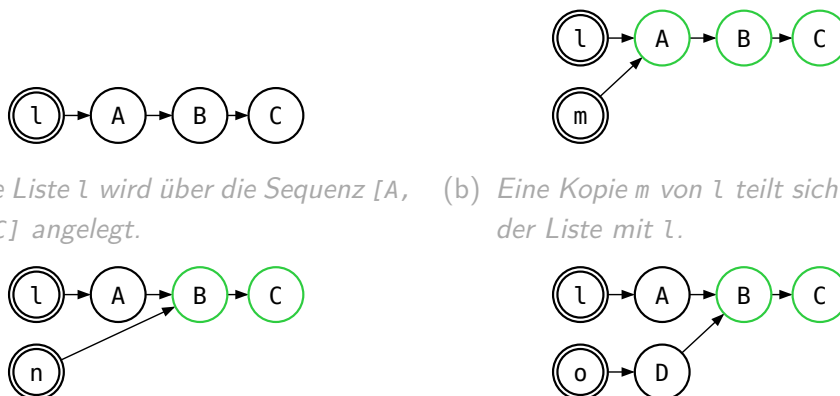


Abbildung 4-2: Eine Abfolge von Operationen auf persistenten verketteten Listen.

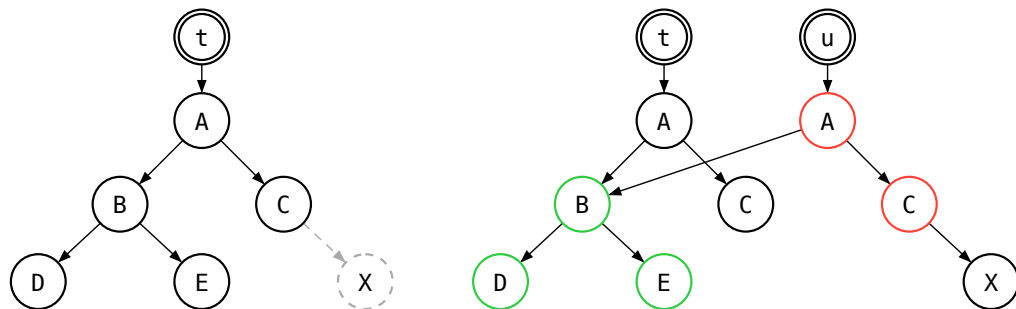
Trotz der Veränderung von m in Abbildung 4-2c teilen sich l und n weiterhin Daten, während QMaps in diesem Fall gar keine Daten mehr teilen würden.

Im Folgenden werden durch die Datenteilung *Instanzen* und *Daten* semantisch getrennt:

Instanzen Teile der Datenstruktur, welche auf die geteilten Daten verweisen und anderweitig nur Verwaltungsinformationen enthalten.

Daten Die Teile einer Datenstruktur, welche die eigentlichen Elemente enthalten, in Abbildung 4-2 beschreibt das die Knoten mit einfacher Umrandung, während doppelt umrandete Knoten die Instanzen sind.

Persistenz zeigt vor allem bei Baumstrukturen ihre Vorteile. Bei der Kopie der Daten eines persistenten Baums können je nach Tiefe und Balance des Baums große Teile des Baums geteilt werden. Ähnlich wie bei persistenten einfach verketteten Listen, werden bei Schreibzugriffen auf persistente Bäume nur die Knoten des Baums kopiert, welche zwischen Wurzel (Instanz) und dem veränderten Knoten in den Daten liegen. Das nennt sich eine Pfadkopie (*engl.* path copying). Betrachten wir partielle Persistenz von Bäumen am Beispiel eines Binärbaums, sprich eines Baums mit Zweigfaktoren (ausgehendem Knotengrad) zwischen 0 und 2. Abbildung 4-3 illustriert, wie am Binärbaum t ein Knoten X angefügt werden kann, ohne dessen partielle Persistenz aufzugeben. Es wird eine neue Instanz angelegt und eine Kopie der Knoten A und C angelegt, der neue Knoten X wird in C eingehängt und der Knoten B wird von beiden A Knoten geteilt. Durch die Teilung von B werden auch alle Kindknoten unter B geteilt.



(a) Eine Baumstruktur t , an welche ein neuer Knoten X unter C angefügt werden soll.

(b) Bei Hinzufügen des Knotens X als Kind des Knotens C wird ein neuer Baum u angelegt.

Abbildung 4-3: Partielle Persistenz teilt zwischen mehreren Instanzen die Teile der Daten, welche sich nicht verändert haben, ähnlich der Persistenz in Abbildung 4-2.

Für unbalancierte Bäume lässt sich dabei aber noch keine besonders gute Zeitkomplexität garantieren. Bei einem Binärbaum mit n Kindern, welcher maximal unbalanciert ist (äquivalent einer verketteten Liste), degeneriert die Zeitkomplexität zu $\Theta(n)$ für Veränderungen am Blatt des Baums. Ein perfekt balancierter Binärbaum hat eine Tiefe $d = \log_2 n$, sodass jeder Schreibzugriff auf einem persistenten Binärbaum maximal d Knoten (Pfad zwischen Wurzel und Blattknoten) kopieren muss. Je besser ein persistenter Baum balanciert ist, desto geringer ist die Anzahl der Knoten, welche bei Pfadkopien unnötig kopiert werden müssen. Die Implementierung von T4gl-Arrays durch persistente Bäume könnte demnach die Komplexität von Kopien drastisch verbessern.

4.3 Lösungsansatz

Zunächst definieren wir Invarianzen von T4gl-Arrays, welche durch eine Änderung der Implementierung nicht verletzt werden dürfen:

1. T4gl-Arrays sind assoziative Datenstrukturen.
 - Es werden Werte mit Schlüsseln adressiert, nicht nur Ganzzahl-Indizes.
2. T4gl-Arrays sind nach ihren Schlüsseln geordnet.
 - Die Schlüsseltypen von T4gl-Arrays haben eine voll definierte Ordnungsrelation.
 - Iteration über T4gl-Arrays ist deterministisch in aufsteigender Reihenfolge der Schlüssel.
3. T4gl-Arrays verhalten sich wie Referenztypen.
 - Schreibzugriffe auf ein Array, welches eine flache Kopie eines anderen T4gl-Arrays ist, sind in beiden Instanzen sichtbar.
 - Tiefe Kopien von T4gl-Arrays teilen sichtbar keine Daten, ungeachtet, ob die darin enthaltenen Typen selbst Referenztypen sind.

Die Ordnung der Schlüssel schließt ungeordnete assoziative Datenstrukturen wie Hash-tabellen aus. Das Referenztypenverhalten ist dadurch umzusetzen, dass wie bis dato drei Ebenen verwendet werden (siehe Abschnitt 2.2), es werden lediglich die Qt-Instanzen durch neue Datenstrukturen ersetzt. Essentiell für die Verbesserung des worst-case Zeitverhaltens bei Kopien und Schreibzugriffen ist die Reduzierung der Daten, welche bei Schreibzugriffen kopiert werden müssen. Hauptproblem bei flachen Kopien, gefolgt von Schreibzugriff auf CoW-Datenstrukturen, ist die tiefe Kopie *aller* Daten, selbst wenn nur ein einziges Element beschrieben oder eingefügt/entfernt wird. Ein Großteil der Elemente in den originalen und neuen kopierten Daten ist nach dem Schreibzugriff gleich. Durch höhere Granularität der Datenteilung müssen bei Schreibzugriffen weniger unveränderte Daten kopiert werden.

Ein Beispiel für persistente Datenstrukturen mit granularer Datenteilung sind RRB-Vektoren [BR11] [Bag+15] [Stu15], eine Sequenzdatenstruktur auf Basis von Radix-Balancierten Bäumen. Trotz der zumeist logarithmischen worst-case Komplexitäten können RRB-Vektoren nicht als Basis für T4gl-Arrays verwendet werden. Die Effizienz der RRB-Vektoren baut auf der Relaxed-Radix-Balancierung auf, dabei wird von einer Sequenzdatenstruktur ausgegangen. Da die Schlüsselverteilung in T4gl-Arrays nicht dicht ist, können die Schlüssel nicht ohne Weiteres auf dicht verteilte Indizes abgebildet werden. Etwas fundamentaler sind B-Bäume [BM70] [Bay71]. Bei persistenten B-Bäumen haben die meisten Operationen eine worst- und average-case Zeitkomplexität von $\Theta(\log n)$. Eine Verbesserung der average-case Zeitkomplexität für bestimmte Sequenzoperationen (push, pop) bieten 2-3-Fingerbäume [HP06]. Diese bieten sowohl theoretisch exzellentes Zeitverhalten, als auch keine Einschränkung auf die Schlüsselverteilung. Im Folgenden werden unter anderem B-Bäume, sowie 2-3-Fingerbäume als alternative Storage-Datenstrukturen für T4gl-Arrays untersucht.

4.4 B-Bäume

Eine für die Persistenz besonders gut geeignete Datenstruktur sind B-Bäume [BM70] [Bay71], da diese durch ihren Aufbau und Operationen generell balanciert sind. Schreiboperationen auf persistenten B-Bäumen müssen lediglich $\Theta(\log n)$ Knoten kopieren. B-Bäume können vollständig durch ihre Zweigfaktoren beschrieben werden, sprich, die

Anzahl der Kindknoten, welche ein interner Knoten haben kann bzw. muss. Ein B-Baum ist wie folgt definiert [Knu98, S. 483]:

1. Jeder Knoten hat maximal k_{\max} Kindknoten.
2. Jeder interne Knoten, außer dem Wurzelknoten, hat mindestens $k_{\min} = \lceil k_{\max}/2 \rceil$ Knoten.
3. Der Wurzelknoten hat mindestens 2 Kindknoten, es sei denn, er ist selbst ein Blattknoten.
4. Alle Blattknoten haben die gleiche Entfernung zum Wurzelknoten.
5. Ein interner Knoten mit k Kindknoten enthält $k - 1$ Schlüssel.

Die Schlüssel innerhalb eines internen Knoten dienen als Suchhilfe. Sie sind Typen mit fester Ordnungsrelation und treten aufsteigend geordnet auf. Abbildung 4-4 zeigt einen internen Knoten eines B-Baums mit $k_{\min} = 2$ und $k_{\max} = 4$, ein sogenannter 2-4-Baum. Jeder Schlüssel s_i mit $1 \leq i \leq k - 1$ dient als Trennwert zwischen den Unterbäumen in den Kindknoten k_j mit $1 \leq j \leq k$. Sei x ein Schlüssel im Unterbaum mit der Wurzel k_i , so gilt

$$\begin{aligned} x &\leq s_i && \text{wenn } i = 1 \\ x &> s_{i-1} && \text{wenn } i = k_{\max} \\ s_{i-1} &< x \leq s_i && \text{sonst} \end{aligned} \quad (4.1)$$

Die Schlüssel erlauben bei der Suche auf jeder Ebene den Suchbereich drastisch zu reduzieren. Die simpelste Form von B-Bäumen sind sogenannte 2-3-Bäume (B-Bäume mit $k_{\min} = 2$ und $k_{\max} = 3$), dabei ist die Angabe der Zweigfaktoren als Prefix des Baums eine übliche Konvention.

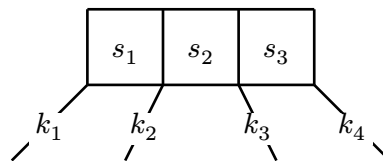


Abbildung 4-4: Ein interner B-Baum-Knoten eines 2-4-Baums.

Da die Schlüssel in B-Bäumen anhand ihrer Ordnungsrelation sortiert sind und demnach auch nicht auf Ganzzahlwerte beschränkt sind, können diese in persistenter Form durchaus eine gute Alternative zu QMap darstellen.

4.5 2-3-Fingerbäume

2-3-Fingerbäume wurden von HINZE und PATERSON [HP06] eingeführt und sind eine Erweiterung von 2-3-Bäumen, welche für verschiedene Sequenzoperationen optimiert wurden. Die Autoren führen dabei folgende Begriffe im Verlauf des Texts ein:

Spine Die Wirbelsäule eines Fingerbaums, sie beschreibt die Kette der zentralen Knoten, welche die sogenannten Digits enthalten.

Digit Erweiterung von 2-3-Knoten auf 1-4-Knoten, von welchen jeweils zwei in einem Wirbelknoten vorzufinden sind. Obwohl ein Wirbelknoten zwei Digits enthält, sind, statt dieser selbst, direkt deren 1 bis 4 Kindknoten an beiden Seiten angefügt. Das reduziert die Anzahl unnötiger Knoten in Abbildungen und entspricht mehr der späteren Implementierung. Demnach wird im Folgenden der Begriff Digits verwendet, um die linken und rechten Kindknoten der Wirbelknoten zu beschreiben.

Measure Wert, welcher aus den Elementen errechnet und kombiniert werden kann. Dieser dient als Suchhilfe innerhalb des Baums, durch welchen identifiziert wird, wo ein Element im Baum zu finden ist, ähnlich den Schlüsseln in Abbildung 4-4.

Safe Sichere Ebenen sind Ebenen mit 2 oder 3 Digits, ein Digit kann ohne Probleme entnommen oder hinzugefügt werden. Zu beachten ist dabei, dass die Sicherheit einer Ebene sich auf eine Seite bezieht, eine Ebene kann links sicher und rechts unsicher sein.

Unsafe Unsichere Ebenen sind Ebenen mit 1 oder 4 Digits, ein Digit zu entnehmen oder hinzuzufügen kann Über- bzw. Unterlauf verursachen (Wechsel von Digits zwischen Ebenen des Baums, um die Zweigfaktoren zu bewahren).

Der Name Fingerbäume rührt daher, dass imaginär zwei Finger an die beiden Enden der Sequenz gesetzt werden. Diese Finger ermöglichen den schnellen Zugriff an den Enden der Sequenz. Die zuvor definierten Digits haben dabei keinen direkten Zusammenhang mit den Fingern eines Fingerbaums, trotz der etymologischen Verwandtschaft beider Begriffe. Abbildung 4-5 zeigt den Aufbau eines 2-3-Fingerbaums, die in blau eingefärbten Knoten sind Wirbelknoten, die in türkis eingefärbten Knoten sind die Digits. In grau eingefärbte Knoten sind interne Knoten. Weiße Knoten sind Elemente, die Blattknoten der Teilbäume. Knoten, welche mehr als einer Kategorie zuzuordnen sind, sind geteilt eingefärbt.

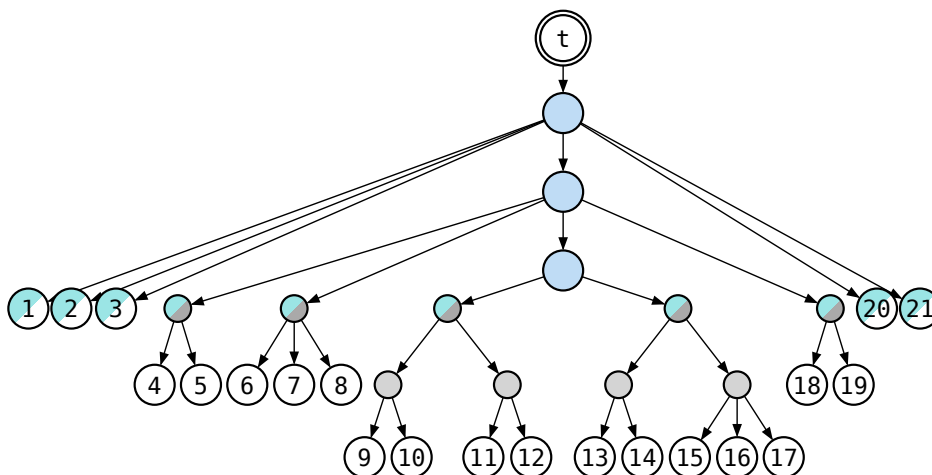


Abbildung 4-5: Ein 2-3-Fingerbaum der Tiefe 3 und 21 Elementen.

Die Tiefe ergibt sich bei 2-3-Fingerbäumen aus der Anzahl der zentralen Wirbel. Jeder Wirbelknoten beschreiben eine Ebene t . Der Baum in Abbildung 4-5 hat Ebene 1 bis Ebene 3. Aus der Tiefe der Wirbelknoten ergibt sich die Tiefe der Teilbäume (2-3-Bäume) in deren Digits. Die Elemente sind dabei nur in den Blattknoten vorzufinden. In Ebene 1 sind Elemente direkt in den Digits enthalten – die Teilbäume haben die Tiefe 1. In Ebene 2 sind die Elemente in Knoten verpackt, deren Digits enthalten Teilbäume der Tiefe 2. Die dritte Ebene enthält Knoten von Knoten von Elementen, sprich Teilbäume der Tiefe 3, und so weiter. Dabei ist zu beachten, dass der letzte Wirbelknoten einen optionalen mittleren Einzelknoten enthalten kann. Dieses Sonderdigit wird verwendet um die Bildung interner Ebenen zu überbrückung, da diese mindestens zwei Digits enthalten müssen. Die linken Digits jedes Wirbelknotens bilden dabei den Anfang der Sequenz, während die rechten Digits das Ende der Sequenz beschreiben. Der mittlere Teilbaum eines Wirbelknotens bildet die Untersequenz zwischen diesen beiden Enden ab. Die Nummerierung der Knoten in Abbildung 4-5 illustriert die Reihenfolge der Elemente. Interne Knoten und Wirbel-

knoten enthalten außerdem die Suchinformationen des Unterbaums, in dessen Wurzel sie liegen. Je nach Wahl des Typs dieser Suchinformationen kann ein 2-3-Fingerbaum als gewöhnlicher Vektor, geordnete Sequenz, Priority-Queue oder Intervallbaum verwendet werden.

Durch den speziellen Aufbau von 2-3-Fingerbäumen weisen diese im Vergleich zu gewöhnlichen 2-3-Bäumen geringere amortisierte Komplexitäten für Deque-Operationen auf. Tabelle 4-1 zeigt einen Vergleich der Komplexitäten verschiedener Operationen über n Elemente. Dabei ist zu sehen, dass push- und pop-Operationen auf 2-3-Fingerbäumen im average-case amortisiert konstantes Zeitverhalten aufweisen, im Vergleich zu dem generell logarithmischen Zeitverhalten bei gewöhnlichen 2-3-Bäumen. Fingerbäume sind symmetrische Datenstrukturen, push und pop kann problemlos an beiden Enden durchgeführt werden.

Tabelle 4-1: Die Komplexitäten von 2-3-Fingerbäumen im Vergleich zu gewöhnlichen 2-3-Bäumen.

Operation	2-3-Baum			2-3-Fingerbaum		
	worst	average	best	worst	average	best
search	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
insert	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
remove	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
push	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)^*$	$\Theta(1)$
pop	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)^*$	$\Theta(1)$

Ein wichtiger Bestandteil der Komplexitätsanalyse der push- und pop-Operationen von 2-3-Fingerbäumen ist die Aufschiebung der Bildung von Wirbelknoten durch Lazy Evaluation. Bei der amortisierten Komplexitätsanalyse werden diesen Aufschiebungen Credits in höhe ihrer sicheren Digits zugewiesen, welche für die Auswertung der aufgeschobenen Unterbäume in einem späteren Zeitpunkt verwendet wird.

Eine naive C++-Definition von 2-3-Fingerbäumen ist in Listing 4-2 beschrieben. T ist der Typ der im Baum gespeicherten Elemente und M der Typ der Suchinformationen für interne Knoten. Im Regelfall wären alle Klassendefinitionen über T und M per **template** parametrisiert, darauf wurde verzichtet, um die Definition im Rahmen zu halten.

Listing 4-2: Die Definition von 2-3-Fingerbäumen in C++ übersetzt.

```

1  using T = ...;
2  using M = ...;
3
4  class Node {};
5  class NodeDeep : public Node<M, T> {
6      M measure;
7      std::array<Node*, 3> children; // 2..3 children
8  };
9  class NodeLeaf : public Node<M, T> {
10     T value;

```

*Amortisiert

```

11 };
12
13 class Digits {
14     M measure;
15     std::array<Node*, 4> children; // 1..4 digits
16 };
17
18 class FingerTree {
19     M measure;
20 };
21 class Shallow : public FingerTree<M, T> {
22     Node* digit; // 0..1 digits
23 };
24 class Deep : public FingerTree<M, T> {
25     Digits<M, T> left; // 1..4 digits
26     FingerTree<M, T>* middle;
27     Digits<M, T> right; // 1..4 digits
28 };

```

Die Werte des Measures werden in Unterbäumen zwischengespeichert und muss dabei ein Monoid sein, sprich, er benötigt eine assoziative Operation $f : (M, M) \rightarrow M$, welche zwei Elemente des Typs M zu einem kombiniert und ein Identitätselement m_0 , sodass $f(m_0, m) = m \wedge f(m, m_0) = m$. Diese Kombinationsoperation wird verwendet um die zwischengespeicherten Werte der Unterbäume aus deren Werten zu kombinieren. Die Elemente des Typen T müssen dann eine Abbildung von $g : T \rightarrow M$ zur Verfügung stellen um die Measures der Blattknoten zu bestimmen.

Definieren wir M als \mathbb{N} mit f und m_0 als

$$\begin{aligned}
 f(l, r) &: (\mathbb{N}, \mathbb{N}) \rightarrow \mathbb{N} \\
 f(l, r) &= l + r \\
 m_0 &= 0
 \end{aligned}
 \tag{4.2}$$

und g als

$$\begin{aligned}
 g(t) &: T \rightarrow \mathbb{N} \\
 g(t) &= 1
 \end{aligned}
 \tag{4.3}$$

so kann dieses Measure die Größe von Unterbäumen beschreiben und für den Fingerbaum können Vektor-Operationen wie `split_at`, `index` oder `length` implementiert werden [HP06, S. 15].

Für die in T4gl verwendeten Arrays sind Schlüssel nach Ordnungsrelation sortiert, daher ergibt sich aus der inherenten Sortierung die Operation $f(l, r) = r$. Leere Unterbäume werden ignoriert, damit kein Sonderwert für m_0 reserviert werden muss. Des Weiteren wird der obengenannte Monoid mitgeführt, um auf die Länge der Unterbäume effektiv zugreifen zu können. Für die Implementierung in T4gl werden andere Measure ignoriert, da diese nicht für das Laufzeitsystem relevant sind und die Implementierung unnötig erschweren würden.

4.6 Generische Fingerbäume

Im Folgenden wird betrachtet, inwiefern die Zweigfaktoren von Fingerbäumen generalisierbar sind, ohne die in Tabelle 4-1 beschriebenen Komplexitäten zu verschlechtern. Höhere Zweigfaktoren der Teilbäume eines Fingerbaums reduzieren die Tiefe des Baums und können die Cache-Effizienz erhöhen.

Wir beschreiben die Struktur eines Fingerbaums durch die Minima und Maxima seiner Zweigfaktoren k (Zweigfaktor der internen Knoten) und d (Anzahl der Digits auf jeder Seite einer Ebene)

$$\begin{aligned} k_{\min} &\leq k \leq k_{\max} \\ d_{\min} &\leq d \leq d_{\max} \end{aligned} \quad (4.4)$$

Die Teilbäume, welche in den Digits liegen, werden auf B-Bäume generalisiert, daher gilt für deren interne Knoten außerdem $k_{\min} = \lceil \frac{k_{\max}}{2} \rceil$, sowie $k_{\min} = 2$ für deren Wurzelknoten.

Die in Listing 4-2 gegebene Definition lässt sich dadurch erweitern, das `std::array` durch `std::vector` ersetzt wird, um die wählbaren Zweigfaktoren zu ermöglichen. Diese können ebenfalls mit `template` Parametern zur Kompilierzeit gesetzt werden, um den Wechsel auf Vektoren zu vermeiden. Die Definition von `Shallow` wird dadurch erweitert, dass diese bis zu $2d_{\min}$ aufnimmt.

Listing 4-3: Die Definition von generischen Fingerbäumen in C++.

```

1  using V = ...;
2  using K = ...;
3
4  class Node {
5      K key;
6  };
7  class Internal : public Node<K, V> {
8      std::vector<Node*> children; // k_min..k_max children
9  };
10 class Leaf : public Node<K, V> {
11     V val;
12 };
13
14 class Digits {
15     K key;
16     std::vector<Node<K, V*>> children;
17 };
18
19 class FingerTree {
20     K key;
21 };
22 class Shallow : public FingerTree<K, V> {
23     std::vector<Node*> digits; // 0..2d_min digits
24 };

```

```

25  class Deep : public FingerTree<K, V> {
26      Digits<K, V> left; // d_min..d_max digits
27      FingerTree<K, V>* middle;
28      Digits<K, V> right; // d_min..d_max digits
29  };

```

Des weiteren betrachten wir für die Generalisierung lediglich Fingerbäume, welche für T4gl relevant sind. Statt generischen Measures für verschiedene Anwendungsfälle wird die Schnittstelle der Operationen auf geordnete einzigartige Schlüssel spezialisiert, aus measure wird der key, gespeicherte größten Schlüssel der Unterbäume. Kein Schlüssel kommt mehr als einmal vor. Während bei der generischen Variante in Listing 4-2 Schlüssel- und Wertetyp in T enthalten sein müssten, sind diese hier direkt in Node zu finden.

Die Operationen in den Abschnitten 4.6.3 bis 4.6.6 verwenden in den meisten Fällen mit Node Variablen als Eingabe oder Ausgabe. Dabei müssen alle Eingaben bei initialem Aufruf Leaf-Knoten sein, um die korrekte Tiefe der Unterbäume zu wahren. Gleichermäßen geben die initialen Aufrufe dieser Algorithmen immer Leaf-Knoten zurück. In beiden Fällen ermöglicht die Verwendung von Node statt der Typen K und V die Definition der Algorithmen durch simple Rekursion. In der Haskell-Definition ergibt sich zur Kompilierzeit aus durch den nicht-regulären Typen FingerTree a [HP06, S. 3] Funktionen, welche bei initialem Aufruf keine Knoten manuell ent- oder verpacken müssen. Eine äquivalente C++-Implementierung kann diese als interne Funktionen implementieren, welche die Eingaben in Knoten verpacken und die Ausgaben entpacken.

Da Wirbelknoten mindestens $2d_{\min}$ Digits enthalten, muss die Bildung dieser Minimalanzahl der Elemente im letzten Wirbelknoten überbrückt werden. Das erzielt man durch das Einhängen von Sonderdigits in den letzten Wirbelknoten, bis genug für beide Seiten vorhanden sind. Prinzipiell gilt für den letzten Wirbelknoten

$$0 \leq d < 2d_{\min} \quad (4.5)$$

Ein generischer Fingerbaum ist dann durch diese Minima und Maxima beschrieben. Im Beispiel des 2-3-Fingerbaums gilt

$$\begin{aligned} 2 &\leq k \leq 3 \\ 1 &\leq d \leq 4 \end{aligned} \quad (4.6)$$

4.6.1 Baumtiefe

Sei $n(t)$ die Anzahl n der Elemente einer Ebene t , also die Anzahl aller Elemente in allen Teilbäumen der Digits eines Wirbelknotens, so gibt es ebenfalls die minimale und maximale mögliche Anzahl für diesen Wirbelknoten. Diese ergeben sich jeweils aus den minimalen und maximalen Anzahlen der Digits, welche Teilbäume mit minimal bzw. maximal belegten Knoten enthalten.

$$\begin{aligned} n_{\min}(t) &= k_{\min}^{t-1} \\ n_{\min}(t) &= 2d_{\min} k_{\min}^{t-1} \\ n_{\max}(t) &= 2d_{\max} k_{\max}^{t-1} \end{aligned} \quad (4.7)$$

n_{\min} beschreibt das Minimum des letzten Wirbelknotens, da dieser nicht an die Untergrenze $2d_{\min}$ gebunden ist. n_{\min} ist das Minimum interner Wirbelknoten. Für die kumulativen Minima und Maxima aller Ebenen bis zur Ebene t ergibt sich

$$\begin{aligned} n'_{\min}(t) &= n_{\min}(t) + n'_{\min}(t-1) \\ n'_{\min}(t) &= n_{\min}(t) + n_{\min}(t-1) + \dots + n_{\min}(1) = \sum_{i=1}^t n_{\min}(i) \\ n'_{\max}(t) &= n_{\max}(t) + n_{\max}(t-1) + \dots + n_{\max}(1) = \sum_{i=1}^t n_{\max}(i) \end{aligned} \quad (4.8)$$

Das wirkliche Minimum eines Baums der Tiefe t ist daher $n'_{\min}(t) = n'_{\min}(t)$, da es immer einen letzten nicht internen Wirbelknoten auf Tiefe t gibt. Abbildung 4-6 zeigt die Minima und Maxima von n für die Baumtiefen $t \in [1, 8]$ für 2-3-Fingerbäume. Dabei zeigen die horizontalen Linien das kumulative Minimum n'_{\min} und Maximum n'_{\max} pro Ebene.

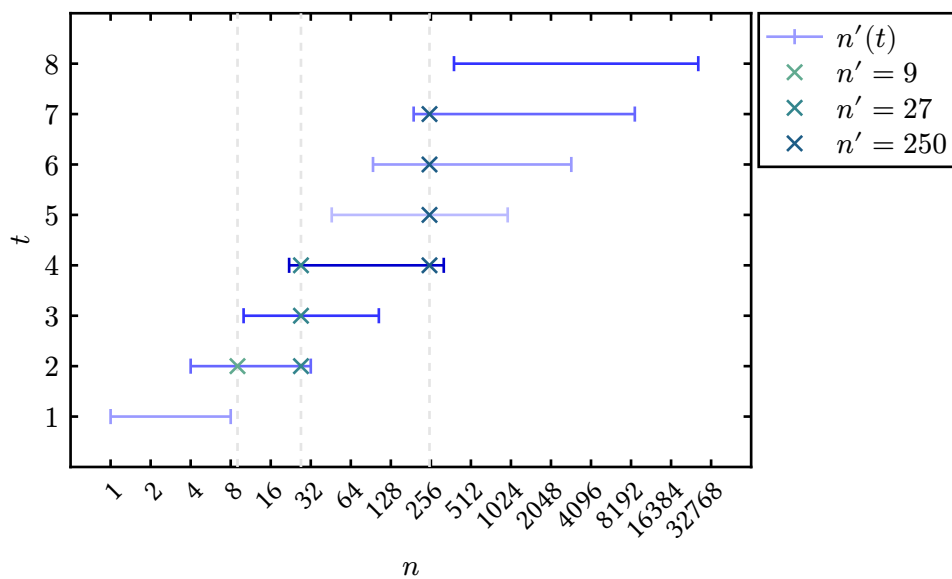


Abbildung 4-6: Die möglichen Tiefen t für einen 2-3-Fingerbaum mit n Blattknoten.

4.6.2 Über- & Unterlaufsicherheit

Über- bzw. Unterlauf einer Ebene t ist der Wechsel von Digits zwischen der Ebene t und der Ebene $t + 1$. Beim Unterlauf der Ebene t sind nicht genug Digits in t enthalten. Es wird ein Digit aus der Ebene $t + 1$ entnommen und dessen Kindknoten in t gelegt. Der Überlauf einer Ebene t erfolgt, wenn in t zu viele Digits enthalten sind. Es werden genug Digits aus t entnommen, sodass diese in einen Knoten verpackt und als Digit in $t + 1$ gelegt werden können. Das Verpacken und Entpacken der Digits ist nötig, um die erwarteten Baumtiefen pro Ebene zu erhalten, sowie zur Reduzierung der Häufigkeit der Über- und Unterflüsse je tiefer der Baum wird.

Die Anzahl der Digits hat dabei einen direkten Einfluss auf die amortisierten Komplexitäten von push- und pop-Operationen.

„We classify digits of two or three elements (which are isomorphic to elements of type Node a) as safe, and those of one or four elements as dangerous. A deque operation may only propagate to the next level from a dangerous digit, but in doing so it makes

that digit safe, so that the next operation to reach that digit will not propagate. Thus, at most half of the operations descend one level, at most a quarter two levels, and so on. Consequently, in a sequence of operations, the average cost is constant.

The same bounds hold in a persistent setting if subtrees are suspended using lazy evaluation. [...] Because of the above properties of safe and dangerous digits, by that time enough cheap shallow operations will have been performed to pay for this expensive evaluation.“

— HINZE und PATERSON [HP06, S. 7]

Die Analyse HINZE und PATERSON stützt sich darauf, dass persistente 2-3-Fingerbäume deren Unterbäume durch Lazy Evaluation erst dann bilden und auswerten, wenn eine weitere Operation bis in diese Tiefe vorgehen muss. Für die Debit-Analyse von push und pop werden dabei den Unterbäumen so viele Credits zugewiesen wie diese sichere Knoten haben. Diese Credits werden für die spätere Auswertung verwendet, sodass die amortisierten Kosten über den realen Kosten bleiben. Damit sich die gleiche Analyse auf generische Fingerbäume übertragen lässt, müssen generische Fingerbäume ebenfalls bei jeder Operation auf einer unsicheren Ebene eine sichere Ebene zurücklassen.

Wir erweitern den Begriff der Sicherheit einer Ebene t mit d Digits als

$$d_{\min} < d_t < d_{\max} \quad (4.9)$$

Ist eine Ebene sicher, ist das Hinzufügen oder Wegnehmen eines Elements trivial. Ist eine Ebene unsicher, kommt es beim Hinzufügen oder Wegnehmen zum Über- oder Unterlauf, Elemente müssen zwischen Ebenen gewechselt werden, um die obengenannten Ungleichungen einzuhalten. Erneut gilt zu beachten, dass die Sicherheit einer Ebene nur für eine Seite der Ebene gilt. Welche Seite, hängt davon ab, an welcher Seite eine Operation wie push oder pop durchgeführt wird. Wir betrachten den Über- und Unterlauf einer unsicheren Ebene t in eine bzw. aus einer sicheren Ebene $t + 1$. Über- oder Unterlauf von unsicheren Ebenen in bzw. aus unsicheren Ebenen kann rekursiv angewendet werden bis eine sichere Ebene erreicht wird. Ähnlich den 2-3-Fingerbäumen wird durch die Umwandlung von unsicheren in sichere Ebenen dafür gesorgt, dass nur jede zweite Operation eine Ebene in dem Baum herabsteigen muss, nur jede vierte zwei Ebenen, und so weiter. Dieses Konzept nennt sich implizite rekursive Verlangsamung (*engl.* implicit recursive slowdown) [Oka98] und ist Kernbestandteil der amortisierten Komplexität Deque-Operationen (push/pop).

Damit die Digits einer Ebene t in eine andere Ebene $t + 1$ überlaufen können, müssen diese in einen Digit-Knoten der Ebene $t + 1$ passen, es gilt

$$k_{\min} \leq \Delta d \leq k_{\max} \quad (4.10)$$

Dabei ist Δd die Anzahl der Digits in t , welche in $t + 1$ überlaufen sollen. Essentiell ist, dass eine unsichere Ebene nach dem Überlauf wieder sicher ist, dazu muss Δd folgende Ungleichungen einhalten. Die Ebene $t + 1$, kann dabei sicher bleiben oder unsicher werden.

$$\begin{aligned} t : d_{\min} &< d_{\max} - \Delta d + 1 < d_{\max} \\ t + 1 : d_{\min} &< d_{t+1} + 1 \leq d_{\max} \end{aligned} \quad (4.11)$$

Gleichermaßen gelten für den Unterlauf folgende Ungleichungen.

$$\begin{aligned} t : d_{\min} &< d_{\min} + \Delta d - 1 < d_{\max} \\ t + 1 : d_{\min} &\leq d_{t+1} - 1 < d_{\max} \end{aligned} \quad (4.12)$$

Betrachtet man die Extrema von Δd in Gleichung (4.10), folgt aus deren Substitution in den Gleichungen (4.11) und (4.12) eine Beziehung zwischen d und k .

$$d_{\max} - d_{\min} + 1 > k_{\max} \quad (4.13)$$

Für eine effiziente Implementierung von Deque-Operationen haben HINZE und PATERSON die Zweigfaktoren der Digits von denen der Knoten um ± 1 erweitert [HP06, S. 4]. Die Definitionen $d_{\min} = k_{\min} - 1$ und $d_{\max} = k_{\max} + 1$ halten nicht generell. Werden diese in Gleichung (4.13) eingesetzt, ergibt sich $k_{\min} < 3$ und folglich $\lceil k_{\max}/2 \rceil < 3$. Unter diesen Definitionen von d in Abhängigkeit von k können Zweigfaktoren ab $k_{\max} > 7$ nicht dargestellt werden. Wählt man stattdessen $d_{\min} = \lceil k_{\min}/2 \rceil$ (siehe Abschnitt 4.6.5, Gleichung (4.16)), ergibt sich

$$d_{\max} - \left\lceil \frac{\lceil \frac{k_{\max}}{2} \rceil}{2} \right\rceil + 1 > k_{\max} \quad (4.14)$$

Die Zweigfaktoren d_{\max} und k_{\max} sind so zu wählen, dass Werte für Δd gefunden werden können, für die zuvor genannten Ungleichungen halten. Betrachten wir 2-3-Fingerbäume, gilt $d_{\min} = 1$, $d_{\max} = 4$, $k_{\min} = 2$ und $k_{\max} = 3$, daraus ergibt sich trivialerweise $4 > 3$.

HINZE und PATERSON entschieden sich bei Überlauf für $\Delta d = 3$, gaben aber an, dass $\Delta d = 2$ ebenfalls funktioniert [HP06, S. 8]. Für generische Fingerbäume zeigt sich das in Gleichung (4.10). Beim Unterlauf hängt Δd von dem Zustand der Datenstruktur ab und kann nicht frei gewählt werden. Aus den oben genannten Ungleichungen lassen sich Fingerbäume mit anderen d_{\max} und k_{\max} wählen, welche die gleichen amortisierten Komplexitäten für Deque-Operationen aufweisen.

4.6.3 Suche

Um ein gesuchtes Element in einem Fingerbaum zu finden, werden ähnlich wie bei B-Bäumen Hilfswerte verwendet, um die richtigen Teilbäume zu finden. In Fingerbäumen sind das die sogenannten Measures. Ein Unterschied zu B-Bäumen ist, wo diese Werte vorzufinden sind. In B-Bäumen sind direkt in den internen Knoten bei k Kindknoten $k - 1$ solcher Hilfswerte zu finden, bei Fingerbäumen sind Measures stattdessen einmal pro internem Knoten vorzufinden. Des Weiteren sind die Measures in Fingerbäumen die Kombination der Measures ihrer Kindknoten.

Für die Anwendung in T4gl werden geordnete Sequenzen benötigt, sodass der Measure interner Knoten den größten Schlüssel des Unterbaums angibt. Ist der Measure des Unterbaums kleiner als der Suchschlüssel, so ist der Schlüssel nicht in diesem Unterbaum enthalten.

Algorithmus 4-1 zeigt, wie durch Measures ein Fingerbaum durchsucht werden kann. Dabei ist **digit-search** Binärsuche anhand der Measures über eine Sequenz von Digits, gefolgt von gewöhnlicher B-Baumsuche (nach Abschnitt 4.4) der gefundenen Digit. Bevor aber **digit-search** angewendet werden kann, muss der richtige Wirbelknoten gefunden werden.

Algorithmus 4-1: $\text{ftree-search}(t, m) : (\text{FingerTree}, \text{Key}) \rightarrow \text{Node}$

```

1  if  $t$  is Shallow
2    if  $m \leq t.\text{key}$ 
3      return  $\text{digit-search}(t.\text{digits}, m)$ 
4    else ▷ key not in this subtree
5      return None
6  else ▷ find suitable branch to descend
7    if  $m \leq t.\text{left}.\text{key}$ 
8      return  $\text{digit-search}(t.\text{left}, m)$ 
9    else if  $m \leq t.\text{middle}.\text{key}$ 
10     return  $\text{ftree-search}(t.\text{middle}, m)$ 
11   else if  $m \leq t.\text{right}.\text{key}$ 
12     return  $\text{digit-search}(t.\text{right}, m)$ 
13   else ▷ key not in this subtree
14     return None

```

Algorithmus 4-1: Die search Operation auf einem Fingerbaum.

Das Zeitverhalten von Algorithmus 4-1 hängt von der Tiefe des Baums ab, da im worst-case der gesuchte Schlüssel im letzten Wirbelknoten vorzufinden ist. Die Baumtiefe steigt logarithmisch mit der Anzahl der Elemente in Abhängigkeit von den Zweigfaktoren (siehe Abschnitt 4.6.1). Je weiter ein Schlüssel von den beiden Enden entfernt ist, desto tiefer muss die Suche in den Baum vordringen. Dabei ist ein Schlüssel, welcher d Positionen vom nächsten Ende entfernt ist $\Theta(\log d)$ Ebenen tief im Baum vorzufinden [HP06, S. 5], also effektiv $\Theta(\log n)$.

4.6.4 Push & Pop

Kernbestandteil von Fingerbäumen sind push und pop an beiden Seiten mit amortisierter Zeitkomplexität von $\Theta(1)$. Die Algorithmen 4-2 und 4-3 beschreiben die Operationen an der linken Seite eines Fingerbaums, durch die Symmetrie von Fingerbäumen lassen sich diese auf die rechte Seite übertragen.

Algorithmus 4-2 ist in zwei Fälle getrennt, je nach der Art des Wirbelknotens t , in welchen der Knoten e eingefügt werden soll. Ist der Wirbelknoten *Shallow*, wird der Knoten e direkt als linkes Digit angefügt, sollten dabei genug Digits vorhanden sein, um einen Wirbelknoten des Typs *Deep* zu erstellen, wird dieser erstellt, ansonsten bleibt der Fingerbaum *Shallow*. In beiden Fällen kommt es nicht zum Überlauf. Sollte der Wirbelknoten *Deep* sein, wird der neue Knoten e als linkes Digit angefügt. Wird die maximale Anzahl der Digits d_{\max} überschritten, kommt es zum Überlauf. Beim Überlauf werden Δd Digits in einen Knoten verpackt und in die nächste Ebene verschoben. Bei push Operationen muss außerdem beachtet werden, dass der Schlüssel des eingefügten Knotens die Ordnung der Schlüssel einhält. In der Implementierung kann diese Operation entweder als *private* markiert werden oder die Knoten vor dem Einfügen validieren.

Algorithmus 4-2: $\text{ftree-push-left}(t, e) : (\text{FingerTree}, \text{Node}) \rightarrow \text{FingerTree}$

```

1  if  $t$  is Shallow                                ▷ overflow by split into
2  | let digits = push-left( $t$ .digits,  $e$ )
3  | if  $|\text{digits}| \leq 2d_{\min}$                         ▷ no split
4  | | return Shallow(digits)
5  | else
6  | | let left, right := split(digits.children,  $d_{\min}$ )
7  | | return Deep(left, None, right)
8  else
9  | let left := push-left( $t$ .left.children,  $e$ )
10 | if  $|\text{left}| \leq d_{\max}$                             ▷ no overflow
11 | | return Deep(left,  $t$ .middle,  $t$ .right)
12 | else
13 | | let rest, overflow := split(left,  $|\text{left}| - \Delta d$ )    ▷ overflow by descent
14 | | let middle := ftree-push-left( $t$ .middle, Node(overflow))
15 | | return Deep(rest, middle,  $t$ .right)

```

Algorithmus 4-2: Die push-Operation an der linken Seite eines Fingerbaumes.

Algorithmus 4-3 ist auf ähnliche Weise wie Algorithmus 4-2 in zwei Fälle getrennt, **Shallow** und **Deep**. Ist der Wirbelknoten **Shallow**, wird lediglich ein Digit von links abgetrennt und zurückgegeben. Bei null Digits wird der Wert **None** zurückgegeben, in einer Sprache wie C++ könnte das durch einen **nullptr** oder das Werfen einer Exception umgesetzt werden. Bei einem **Deep** Wirbelknoten wird ein linkes Digit entfernt. Bleiben dabei weniger als d_{\min} Digits vorhanden, erzeugt das entweder Unterlauf oder den Übergang zum **Shallow** Fingerbaum. Ist die nächste Ebene selbst **Shallow** und leer, werden die linken und rechten Digits zusammengefasst und diese Ebene selbst wird **Shallow**. Ist die nächste Ebene nicht leer, wird ein Knoten entnommen und dessen Kindknoten füllen die linken Digits auf.

Sowohl push als auch pop gehen nach dem gleichen rekursiven Über/Unterfluss-Prinzip vor und hängen daher von der Baumtiefe ab, im worst-case ist jede Ebene unsicher und sorgt für einen Über- oder Unterfluss, daher ergibt sich eine Komplexität von $\Theta(\log n)$. Die amortisierte Komplexität stützt sich auf die gleiche Analyse wie diese von 2-3-Fingerbäumen, insofern Unterbäume mit Lazy Evaluation aufgeschoben werden und korrekte Zweigfaktoren gewählt wurden, können auch generische Fingerbäume push und pop in amortisiert $\Theta(1)$ ausführen.

Algorithmus 4-3: $\text{ftree-pop-left}(t) : \text{FingerTree} \rightarrow (\text{Node}, \text{FingerTree})$

```

1  if  $t$  is Shallow
2  | if  $|\text{t.digits}| = 0$ 
3  | | return (None,  $t$ )
4  | else
5  | | let  $e$ , rest := pop-left( $t$ .digits)
6  | | return ( $e$ , Shallow(rest))

```

Algorithmus 4-3: $\text{ftree-pop-left}(t) : \text{FingerTree} \rightarrow (\text{Node}, \text{FingerTree})$

```

7  else
8  | let  $e, \text{rest}_l := \text{pop-left}(t.\text{left.digits})$ 
9  | if  $|\text{rest}_l| \geq d_{\min}$  ▷ no underflow
10 |   return  $(e, \text{Deep}(\text{rest}_l, t.\text{middle}, t.\text{right}))$ 
11 | else if  $|t.\text{middle}| = 0$  ▷ underflow by left+right merge
12 |   let  $\text{middle} = \text{concat}(t.\text{left}, t.\text{right})$ 
13 |   if  $|\text{middle}| \geq 2d_{\min}$ 
14 |   | let  $\text{left}, \text{right} = \text{split}(\text{middle}, \lfloor |\text{middle}|/2 \rfloor)$ 
15 |   | return  $(e, \text{Deep}(\text{left}, \text{Shallow}(\text{None}), \text{right}))$ 
16 |   else
17 |   | return  $(e, \text{Shallow}(\text{middle}))$ 
18 | else ▷ underflow by descent
19 |   let  $\text{node}, \text{rest}_m := \text{ftree-pop-left}(t.\text{middle})$ 
20 |   return  $(e, \text{Deep}(\text{concat}(\text{rest}_l, \text{node.children}), \text{rest}_m, t.\text{right}))$ 

```

Algorithmus 4-3: Die pop-Operation an der linken Seite eines Fingerbaumes.

Außerdem definieren wir append und take als wiederholte Versionen von push und pop. Diese sind gleicherweise symmetrisch für die rechte Seite zu definieren. Die Operation take nimmt dabei so viele Elemente aus t wie möglich, aber nicht mehr als n .

Algorithmus 4-4: $\text{ftree-append-left}(t, \text{nodes}) : (\text{FingerTree}, [\text{Node}]) \rightarrow \text{FingerTree}$

```

1  for  $e$  in rev(nodes) ▷ simply push all values one by one
2  |  $t = \text{ftree-push-left}(t, e)$ 
3  return  $t$ 

```

Algorithmus 4-4: Die append-Operation an der linken Seite eines Fingerbaumes.

Die worst-case Komplexität von Algorithmus 4-6 ergibt sich aus der Anzahl der Elemente im Baum n und derer in der Sequenz nodes Algorithmus 4-2 als $\Theta(|\text{nodes}| \log n)$.

Algorithmus 4-5: $\text{ftree-take-left}(t, \text{count}) : (\text{FingerTree}, \text{Int}) \rightarrow ([\text{Node}], \text{FingerTree})$

```

1  let  $n' = \emptyset$ 
2  for _ in 1..count
3  | if  $|t| = 0$  ▷ no more values left
4  |   break
5  | else
6  |   let  $e, t' = \text{ftree-pop-left}(t)$ 
7  |    $t = t'$ 
8  |    $n' = \text{push-right}(n', e)$ 
9  return  $t$ 

```

Algorithmus 4-5: Die take-Operation an der linken Seite eines Fingerbaumes.

Da Algorithmus 4-5 nur so viele Elemente entfernen kann wie im Baum vorhanden sind, ist der Faktor das Minimum aus der Anzahl der Elemente n und dem Argument `count`, die Komplexität von `take` ist demnach $\Theta(\min(\text{count}, n) \log n)$.

4.6.5 Split & Concat

Das Spalten und Zusammenführen zweier Fingerbäume sind fundamentale Operationen, welche vor allem für die Implementierung von `insert` und `remove` in Abschnitt 4.6.6 relevant sind. Algorithmus 4-6 beschreibt, wie zwei Fingerbäume zusammengeführt werden, dabei muss in jeder Ebene eine Hilfssequenz m übergeben werden, welche die Zusammenführung der inneren Digits der Bäume beschreibt, nachdem deren Kindknoten entpackt wurden. Beim initialen Aufruf wird die leere Sequenz \emptyset übergeben. Algorithmus 4-6 ruft sich selbst rekursiv auf, bis einer der Bäume `Shallow` ist, in diesem Fall degeneriert der Algorithmus zu wiederholtem `push` und terminiert. Ähnlich wie bei Algorithmus 4-2 muss beachtet werden, dass die Zusammenführung der zwei Bäume die Ordnung der Schlüssel einhält.

Algorithmus 4-6: `ftree-concat`(l, m, r) : (`FingerTree`, [`Node`], `FingerTree`) → `FingerTree`

```

1 if  $l$  is Shallow
2   | return ftree-append-left( $r$ , push-left( $m$ ,  $l$ .digits))
3 else if  $r$  is Shallow
4   | return ftree-append-right( $l$ , push-right( $m$ ,  $r$ .digits))
5 else
6   | let  $m' := \text{pack-nodes}(\text{concat}(l.\text{right}, m, r.\text{left}))$     ▷ pack nodes for the next layer
7   | return Deep( $l.\text{left}$ , ftree-concat( $l.\text{middle}$ ,  $m'$ ,  $r.\text{middle}$ ),  $r.\text{right}$ )

```

Algorithmus 4-6: Das Zusammenfügen zweier Fingerbäume zu einem.

Der Hilfsalgorithmus 4-7 verpackt Knoten für die Rekursion in die nächste Ebene in Algorithmus 4-6. Die Größe der zurückgegebenen Sequenz von verpackten Knoten hat dabei direkten Einfluss auf die eigene Komplexität (durch den rekursiven Aufruf in Algorithmus 4-6) und auf die Komplexität der nicht rekursiven Basisfälle in Algorithmus 4-6.

Algorithmus 4-7: `pack-nodes`(n) : [`Node`] → [`Node`]

```

1 if  $|n| < k_{\min}$                                           ▷ can't form a single node
2   | return None
3 else
4   | let  $n' := \emptyset$ 
5   | while  $|n| - k_{\max} \geq k_{\min}$                           ▷ push max size nodes as long as it goes
6     | let  $\text{children}, \text{rest} = \text{split}(n, k_{\max})$ 
7     |  $n = \text{rest}$ 
8     |  $n' = \text{push-right}(n', \text{Node}(\text{children}))$ 
9   | if  $|n| - k_{\min} \geq k_{\min}$                               ▷ push the remaining 1 or 2 nodes
10  | let  $a, b = \text{split}(n, k_{\min})$ 

```

Algorithmus 4-7: $\text{pack-nodes}(n) : [\text{Node}] \rightarrow [\text{Node}]$

```

11 |   |  $n' = \text{push-right}(n', \text{Node}(a))$ 
12 |   |  $n' = \text{push-right}(n', \text{Node}(b))$ 
13 | else
14 |   |  $n' = \text{push-right}(n', \text{Node}(n))$ 
15 | return  $n'$ 

```

Algorithmus 4-7: Ein Hilfsalgorithmus zum Verpacken von Knoten.

Bei 2-3-Fingerbäumen lässt sich zeigen, dass die mittlere Sequenz m eine Obergrenze von 4 und eine Untergrenze von 1 hat. Zunächst betrachtet man, wie das Verpacken der Knoten sich auf deren Anzahl auswirkt.

Sei m_t die Anzahl der verpackten Knoten bei Rekursionstiefe t , gilt

$$n_t = k_t + m_{t-1}$$

$$m_t = \begin{cases} n_t/k_{\max} & \text{wenn } n_t \bmod k_{\max} = 0 \\ \lfloor n_t/k_{\max} \rfloor + 1 & \text{wenn } n_t \bmod k_{\max} \geq k_{\min} \\ \lfloor (n_t - k_{\min})/k_{\max} \rfloor + 2 & \text{wenn } n_t \bmod k_{\max} < k_{\min} \end{cases} \quad (4.15)$$

dabei ist k_t die Summe der Knoten links und rechts und m_{t-1} die Größe der Sequenz aus der vorherigen Tiefe. Beim initialen Aufruf ist die Sequenz leer, daher gilt $m_0 = 0$.

Für 2-3-Fingerbäume verhält sich die Größe der mittleren Sequenz wie folgt. Auf jeder Ebene können zwischen $2d_{\min} = 2$ und $2d_{\max} = 8$ Elemente zu den verpackten Elementen der vorherigen Ebene hinzukommen.

Tabelle 4-2: Beim Verpacken der Knoten ergibt sich eine Obergrenze für die Anzahl der verpackten Knoten.

t	$\min n_t \Rightarrow \min m_t$	$\max n_t \Rightarrow \max m_t$
0	$0 \Rightarrow 0$	$0 \Rightarrow 0$
1	$2 + 0 \Rightarrow 1$	$8 + 0 \Rightarrow 3$
2	$2 + 1 \Rightarrow 1$	$8 + 3 \Rightarrow 4$
3	$2 + 1 \Rightarrow 1$	$8 + 4 \Rightarrow 4$
4	$2 + 1 \Rightarrow 1$	$8 + 4 \Rightarrow 4$
...	ad infinitum	ad infinitum

Damit n_t Knoten verpackt werden können, muss die übergebene Sequenz mindestens k_{\min} Knoten enthalten. Bei 2-3-Fingerbäumen reicht $k_{\min} = 2d_{\min}$ aus, um in jeder Ebene beim Zusammenführen der inneren Digits mindestens einen Knoten bilden zu können. Tabelle 4-2 zeigt, dass die Länge der mittleren Sequenz, je nach Belegung der inneren Digits, zwischen 1 und 4 liegt. Im generischen Fall muss aus $2d_{\min}$ mindestens ein Knoten gebildet werden. Daraus folgt

$$k_{\min} \leq 2d_{\min} \quad (4.16)$$

Andernfalls können Fingerbäume nicht zusammengefügt werden. Experimentelle Ergebnisse deuten darauf hin, dass bei $k_{\max} > 5$ and $d_{\max} = k_{\max} + 1$ eine Obergrenze ab $m_2 =$

3 existiert, aber ein Beweis konnte nicht formuliert werden. Ohne einen Beweis für die Obergrenze dieser Sequenz kann nicht nachgewiesen werden, dass Algorithmus 4-6 generell in logarithmischer Zeit läuft, da über diese Sequenz im Basisfall der Rekursion iteriert wird. Es ist unklar, ob die Sequenz, welche durch Gleichung (4.15) erzeugt wird, eine obere asymptotische Grenze von $O(1)$ hat. Stellt sich heraus, dass diese Sequenz sich in die Größenordnung von $\Theta(n)$ bewegt, tut das auch die Komplexität von Algorithmus 4-6. Wird allerdings ein t gefunden, ab welchem diese Sequenz für gewählte Zweigfaktoren nicht weiter ansteigt, ist der Fingerbaum valide.

Einen Fingerbaum zu teilen ist essentiell, um einzelne Elemente mit bestimmten Eigenschaften zu isolieren, wie es für `insert` und `remove` der Fall ist. Algorithmus 4-8 zeigt, wie ein Fingerbaum t so geteilt wird, dass alle Schlüssel, welche größer als k sind, im rechten Baum und alle anderen im linken Baum landen. Das folgt daraus, dass jeder Schlüssel nur einmal im Baum vorkommen darf und alle Schlüssel sortiert sind. Die Implementierung in [HP06] kann weder garantieren, dass die zurückgegebene Teilung die einzige, noch dass diese die erste Teilung ist, da diese nicht von monoton ansteigenden Schlüsseln ausgehen kann.

Algorithmus 4-8: `ftree-split(t, k) : (FingerTree, Key) → (FingerTree, Node, FingerTree)`

```

1  if  $t$  is Shallow
2    let  $l, v, r := \text{digit-split}(t.\text{digits}, k)$ 
3    return (Shallow( $l$ ),  $v$ , Shallow( $r$ ))
4  else
5    if  $k \leq t.\text{left.key}$ 
6      let  $l, v, r := \text{digit-split}(t.\text{left.children}, k)$ 
7      return (Shallow( $l$ ),  $v$ , Deep( $r, t.\text{middle}, t.\text{right}$ ))
8    else if  $k \leq t.\text{middle.key}$ 
9      let  $l, v, r := \text{ftree-split}(t.\text{middle}, k)$                                 ▷ descend and unpack
10     let  $l', v', r' := \text{digit-split}(v.\text{children}, k)$ 
11     return (Deep( $t.\text{left}, l, l'$ ),  $v', \text{Deep}(r', r, t.\text{right})$ )
12   else if  $k \leq t.\text{right.key}$ 
13     let  $l, v, r := \text{digit-split}(t.\text{right.children}, k)$ 
14     return (Deep( $t.\text{left}, t.\text{middle}, l$ ),  $v$ , Shallow( $r$ ))
15   else
16     return ( $t$ , Shallow( $\emptyset$ ))

```

Algorithmus 4-8: Die Teilung eines Fingerbaumes um einen Schlüssel.

4.6.6 Insert & Remove

HINZE und PATERSON entschieden sich, `insert` und `remove` für geordnete Sequenzen durch `split`, `push` und `concat` zu implementieren.

Da in Abschnitt 4.6.5 kein Beweis vorliegt, dass Algorithmus 4-6 in logarithmischer Komplexität läuft, können `insert` und `remove` ebenfalls nicht nachweislich auf diese Art mit logarithmischer Laufzeit implementiert werden. Unter der Annahme, dass so ein Beweis

möglich ist, könnten insert und remove wie folgt implementiert werden. Algorithmus 4-9 zeigt die Implementierung von insert. Dabei ist zu beachten, dass wenn ein Schlüssel bereits in t existiert, dessen Wert ersetzt werden muss, da jeder Schlüssel nur maximal einmal vorhanden sein darf.

Algorithmus 4-9: $\text{ftree-insert}(t, e) : (\text{FingerTree}, \text{Node}) \rightarrow \text{FingerTree}$

```

1 let  $l, v, r := \text{ftree-split}(t, e.\text{key})$ 
2 if  $v$  is None ▷  $e.\text{key}$  not in  $t$ 
3   | let  $l' := \text{ftree-push-right}(l, e)$ 
4   | return  $\text{ftree-concat}(l', r)$ 
5 else ▷  $k$  in  $t$ 
6   | let  $l' := \text{ftree-push-right}(l, v)$ 
7   | let  $l'' := \text{ftree-push-right}(l', e)$ 
8   | return  $\text{ftree-concat}(l'', r)$ 

```

Algorithmus 4-9: insert als Folge von split, push und concat.

In Algorithmus 4-10 ist zu sehen, wie remove zu implementieren ist. Wie auch bei Insert wird geprüft, ob der erwünschte Schlüssel im linken Baum ist. Ähnlich der Implementierung von Algorithmus 4-1 und 4-3 muss bei Nichtvorhandensein des Schlüssels k ein **None**-Wert oder eine Fehler zurückgegeben werden.

Algorithmus 4-10: $\text{ftree-remove}(t, k) : (\text{FingerTree}, \text{Key}) \rightarrow (\text{FingerTree}, \text{Node})$

```

1 let  $l, v, r := \text{ftree-split}(t, k)$ 
2 if  $v$  is None ▷  $k$  not in  $t$ 
3   | return  $(t, \text{None})$ 
4 else ▷  $k$  in  $t$ 
5   | let  $t' := \text{ftree-concat}(l, r)$ 
6   | return  $(t', v)$ 

```

Algorithmus 4-10: remove als Folge von split und concat.

Alternative Implementierungen wurden nicht untersucht, aber je nach Besetzung der Unterbäume können insert und remove auch diese Lücken ausnutzen um statt split und concat auf simple Pfadkopie zurückzugreifen. Die Häufigkeit der Lücken hängt dabei von der Wahl von Δd ab. Liegt Δd näher an k_{\min} , können Blattknoten öfter im Nachhinein befüllt werden.

4.6.7 Echtzeitanalyse

Die Einhaltung verschiedener Ungleichungen in Abschnitt 4.6.2 hat vor allem Einfluss auf die amortisierten Komplexitäten der Deque-Operationen. Das amortisierte Zeitverhalten hat für die Echtzeitanalyse keinen Belang. Allerdings kann ohne einen Beweis in Abschnitt 4.6.5 keine Aussage darüber gemacht werden, ob die Operationen bei generischen Zweigfaktoren in ihren Komplexitätsklassen bleiben oder sogar schlechteres Verhalten als persistente B-Bäume vorweisen.


Sollte ein Beweis für logarithmische Laufzeit von Algorithmus 4-6 gefunden werden, können Fingerbäume exzellentes Zeitverhalten im Vergleich zu gewöhnlichen CoW-Datenstrukturen ohne granulare Persistenz vorweisen. Je nach Anforderungen des Systems könnten für verschiedene Operationen zulässige Höchstlaufzeiten festgelegt werden. Granular persistente Baumstrukturen wie Fingerbäume könnten prinzipiell mehr Elemente enthalten, bevor diese Höchstlaufzeiten pro Operationen erreicht werden. Aus Sicht der Echtzeitanforderungen an die Operationen auf der Datenstruktur selbst, ist jede Datenstruktur mit logarithmischem Zeitverhalten vorzuziehen. Die Wahl von Fingerbäumen über B-Bäumen ist daher eher eine Optimierung als eine Notwendigkeit.

Ohne den zuvor genannten Beweis für die Obergrenze der inneren Sequenz bei concat sind jedoch entweder 2-3-Fingerbäume oder B-Bäume generischen Fingerbäumen vorzuziehen.

4.7 SRB-Bäume

RRB-Bäume (*engl.* relaxed radix balanced trees) sind beinahe perfekt balancierte Suchbäume, welche für Sequenzdatenstrukturen wie Vektoren verwendet werden und wurden von BAGWELL *et al.* eingeführt [BR11] [Bag+15] [Stu15]. Um das Element an einem bestimmten Index zu finden, wird bei perfekter Balancierung eines Knotens der Teilindex direkt errechnet. Bei nicht perfekter Balancierung stützt sich die Suche auf Längeninformationen der Kindknoten. Dazu enthält jeder interne Knoten ein Array der kumulativen Anzahlen von Werten in dessen Kindknoten. Dabei wird nach Indizes gesucht, sprich, fortlaufende Ganzzahlwerte zwischen 0 und der Anzahl der Elemente n in der Sequenz. Im Folgenden wird eine Erweiterung von RRB-Bäumen vorgestellt, SRB-Bäume (*engl.* sparse radix balanced trees), welche für nicht fortlaufende Schlüssel verwendet werden können. Ein SRB-Baum ist aufgebaut wie ein perfekt balancierter voll belegter RRB-Baum über alle Indizes, welche vom Schlüsseltyp dargestellt werden können, aber ohne die Knoten, die nicht belegt sind. Im Gegensatz zu RRB-Bäumen können Teilindizes auf jeder Ebene direkt errechnet werden, ohne dass die Länge der Kindknoten bekannt sein muss. Da aber nicht alle Knoten besetzt sind, muss auf jeder Ebene geprüft werden, ob der errechnete Teilindex auf einen besetzten Knoten zeigt. Durch die perfekte Balancierung ist die Baumtiefe d nur von der Schlüsselbreite b (die Anzahl der Bits im Schlüssel) und dem Zweigfaktor k abhängig.

$$d = \log_k 2^b \quad (4.17)$$

Abbildung 4-7 zeigt einen SRB-Baum über die Sequenz [0, 16, 65296, 65519] für uint16 Schlüssel und mit einem Zweigfaktor von 16. Die belegten Knoten sind in türkis  hervorgehoben.

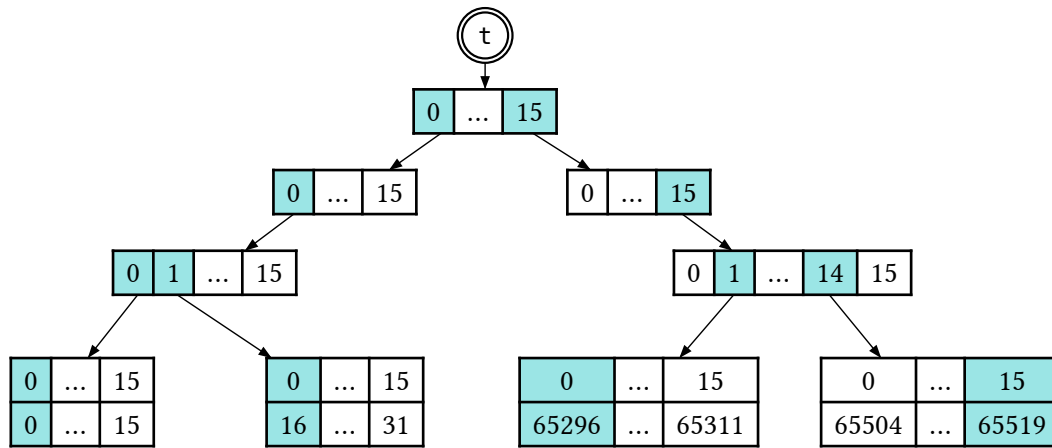


Abbildung 4-7: Ein SRB-Baum für die 16 bit Schlüssel und einen Zweigfaktor von 16.

4.7.1 Schlüsseltypen

Durch die spezifische Art der Suche können nur Schlüsseltypen verwendet werden, welche anhand von Radixsuche Teilindizes errechnen lassen. Das beschränkt sich generell auf nicht vorzeichenbehaftete Ganzzahltypen. Für diese Schlüssel u ergibt sich ein Teilindex i auf Ebene t wie folgt:

$$\frac{u}{k^{d-t}} \bmod k \quad (4.18)$$

Bei Zweigfaktoren $k = 2^l$ ergibt sich eine optimierte Rechnung, welche verschiedene Bitshifting Operationen verwendet:

$$(u \gg (l \cdot (d - t))) \& (k - 1) \quad (4.19)$$

Sollte für einen Schlüsseltyp T eine bijektive Abbildung f und deren Inverses f^{-1} existieren und es gilt

$$\forall t_1, t_2 \in T : t_1 < t_2 \Leftrightarrow f(t_1) < f(t_2) \quad (4.20)$$

dann kann durch diese Abbildungen auch der Schlüsseltyp T durch einen SRB-Baum verwaltet werden. Ein simples Beispiel sind vorzeichenbehaftete Schlüssel, welche einfach um ihr Minimum verschoben werden, so dass alle möglichen Werte vorzeichenunbehaftet sind. Das setzt natürlich voraus, dass ein Minimum existiert, wie es bei Ganzzahlrepräsentationen in den meisten Prozessoren der Fall ist. Ein Minimum und Maximum für die Schlüssel dieser Bäume bedeutet auch, dass die maximale Tiefe vor der Laufzeit bekannt ist.

4.7.2 Zeitverhalten

Um zur Laufzeit den Blattknoten zu finden, in welchem sich ein Schlüssel befindet, wird pro Ebene per Radixsuche der Index dieser Ebene errechnet und geprüft ob dieser Index besetzt ist. Ist der Index nicht besetzt, endet die Suche, ansonsten fährt diese fort, bis der Blattknoten erreicht ist. Durch die perfekte Balancierung ist die Baumtiefe nur von der Schlüsselbreite abhängig, nicht von der Anzahl der gespeicherten Elemente. Operationen wie insert, remove oder search haben daher eine Zeitkomplexität von $\Theta(d)$ mit d aus Gleichung (4.17).

Gerade unter Echtzeitbedingungen erleichtert das die Analyse enorm, durch die Wahl von Höchstschlüsselbreiten kann SRB-Bäumen konstantes worst-case Zeitverhalten für alle Operationen garantiert werden. Diese Schlüsselbreiten sind in den meisten Fällen von der Hardware vorgegeben und können in den üblichen Größen von 8, 16, 32, 64 und in experimentellen CPUs bis zu 128.

4.7.3 Speicherauslastung

Seien v der benötigte Speicher eines Werts und p der benötigte Speicher eines Pointers in einem SRB-Baum. Wir definieren die Speicherauslastung s_U als das Verhältnis zwischen den belegten Speicherplätzen s_S und den gesamten Speicherplätzen s_A , welche von der Datenstruktur angelegt wurden.

$$s_U(p, v) = \frac{s_S(p, v)}{s_A(p, v)} \quad (4.21)$$

Im Idealfall liegt s_U nahe 1, sprich 100%. Der in Abbildung 4-7 gegebene SRB-Baum verbraucht pro Blattknoten $16s_v$ Speicherplatz, ungeachtet der Belegung. Interne Knoten benötigen $16s_p$ Speicherplatz. Bei ungünstiger Belegung, zum Beispiel einem Element pro möglichem Blattknoten, ergibt sich ein Speicherplatzverbrauch von $16^4s_v + 16^3s_p + 16^2s_p + 16s_p$ bei nur 16^3 belegten Elementen. Bei Pointern mit $p = 8$ Byte und Pointern als Werten ($v = p$) folgt eine Speicherauslastung von

$$\begin{aligned} s_U(p, v) &= \frac{16^3v}{16^4v + 16^3p + 16^2p + 16p} \\ &= 0.058... \\ &\approx 6\% \end{aligned} \quad (4.22)$$

Dabei sind Verwaltungsstrukturen ignoriert, welche die Belegung der Elemente enthalten, da diese verhältnismäßig klein sind. Für einen SRB-Baum mit einem Zweigfaktor k und einer Schlüsselbreite b und der sich daraus ergebenden Baumtiefe $d = \lceil \log_k 2^b \rceil$, ergibt sich eine Speicherauslastung im worst-case von:

$$s_U(p, v) = \frac{k^{d-1}v}{k^d v + \sum_{n=1}^{d-1} k^n p} \quad (4.23)$$

Gerade wegen dieser potentiell hohen Speicherverschwendung und der Einschränkung auf die Schlüsseltypen wurden SRB-Bäume als Implementierung für T4gl-Arrays nicht weiter verfolgt.

5 Implementierung

Da keine eindeutigen Beweise gefunden werden konnten, welche eine sichere API der C++-Implementierung generischer Fingerbäume auf Typ-Ebene umsetzen können, wurden 2-3-Fingerbäume in C++ übersetzt um deren Verhalten mit dem von QMap zu vergleichen. Auf eine Implementierung von SRB-Bäumen wird Anhand des Speicherverbrauchs verzichtet.

5.1 Von Haskell zu C++

Die Implementierung von 2-3-Fingerbäumen, sowie deren Benchmarks mit Tests von QMap und einer persistenten B-Baum-Implementierung zur Kontrolle können unter `GitHub:tingerr/FingerTree` eingesehen werden. Verweise auf die Implementierung beziehen sich auf den letzten Stand dieses Repositories, welches mit Abgabe dieser Arbeit archiviert wird.

Die ursprüngliche Implementierung von 2-3-Fingerbäumen hat drei verschiedene Zustände, welche verschiedene Invarianzen sichern:

- **Empty**: Ein leerer Zustand ohne Felder.
- **Single**: Ein Überbrückungszustand zum Speichern eines einzelnen Knotens in einer Ebene, dieser hat einen einzelnen Knoten als Feld.
- **Deep**: Die rekursive Definition der Datenstruktur. Dieses hat drei Felder, die Digits auf beiden Seiten und einen weiteren Fingerbaum mit nicht-regulärer Rekursion auf Typenebene [BM98]. Die Definition von `FingerTree a` verwendet nicht erneut `FingerTree a`, sondern `FingerTree (Node a)` (analog zu `FingerTree<Node<T>>` statt `FingerTree<T>`), daraus wird, zusammen mit der Definition von `Node` die Tiefe der Knoten auf jeder Ebene automatisch durch den Typ gesichert.

Ohne die Typrekursion in der Deep-Variante würde sich ein Fingerbaum wie eine einfach verkettete Liste verhalten, die Rekursion ist essentiell für die Performance der Datenstruktur. In C++ ist die Definition von rekursiven Typen möglich, diese Definitionen müssen aber regulär erfolgen und wenn es um die Definition von den Feldern einer Datenstruktur handelt müssen Selbstverweise durch Indirektion erfolgen. Eine Klasse `C<T>` kann in der eigenen Definition auf folgende Versionen von `C<T>` zurückgreifen:

- Voll-instanzierte Typen wie `C<int>` oder `C<std::string>`.
- Neu parametrisierte Typkonstruktoren wie `template<typename U> class C<U>`, sofern diese nicht von `T` abhängen.
- Eigenverweise wie `C<T>` ohne Verpacken von Typen wie `C<A<T>>`.

Die Definition in Listing 5-1 wäre eine 1-1 Übersetzung der Haskell-Definition (Initialdefinition ohne Measures), diese wäre nach den oben genannten Regeln in C++ illegal. Zunächst müssen Eigenverweise durch Indirektion erfolgen, das Feld `middle` in `Deep` müsste also ein Pointer-Typ sein. Allerdings ist der Eigenverweis dann immernoch nicht möglich, da `Deep` auf den Typen `FingerTree<Node<T>>` verweist – ein nicht-regulärer Eigenverweis. Die Balancierung der Unterbäume kann in C++ nicht zur Kompilierzeit garantiert werden. Stattdessen muss in `middle` auf `FingerTree<T>` verwiesen werden und die Baumstruktur der `Node`-Typen wird durch `Internal` und `Leaf`-Typen erzeugt. Diese Neudefinition der `Node`-Typen hat auch einen Einfluss auf die API der Algorithmen. Während die gleichen Algorithmen in der Haskell-Definition verwendet werden können, um einzelne Elemente vom Typ `a`, sowie deren verpackete Knoten (`Node a`, `Node (Node a)`),

etc.) einzufügen oder abzutrennen, muss die C++-Definition immer Knoten einfügen und zurückgeben. Die internen rekursiven Algorithmen werden als `private` markiert und nur von speziellen Hilfsfunktionen aufgerufen, welche die Knoten entpacken oder verpacken.

Listing 5-1: Nicht-reguläre Definition von Fingerbäumen.

```

1  template <typename T>
2  class Node {};
3
4  template <typename T>
5  class Node2 : public Node<T> { T a; T b; };
6
7  template <typename T>
8  class Node3 : public Node<T> { T a; T b; T c; };
9
10 template <typename T>
11 class Digits : { std::vector<T> digits; };
12
13 template <typename T>
14 class FingerTree {};
15
16 template <typename T>
17 class Empty : public FingerTree<T> {};
18
19 template <typename T>
20 class Single : public FingerTree<T> {
21     T node;
22 };
23
24 template <typename T>
25 class Deep : public FingerTree {
26     Digits<T> left;
27     FingerTree<Node<T>> middle;
28     Digits<T> right;
29 };

```

In der simplifizierten Definition in Listing 4-3 ist bereits zu sehen, dass die Elemente `a` und deren generische Measures `v` durch die expliziten Typen `K` und `V` ersetzt wurden. Für `T4gl` sind andere Measures als `Key` über Schlüssel-Wert-Paare nicht relevant und wurden daher direkt eingesetzt. Diese Ersetzung zeigt sich auch in den Algorithmen, welche einerseits direkt auf die Invarianzen von `Key` und dessen Ordnungsrelation zurückgreifen, sowie auch diese Algorithmen enthalten, welche nur für geordnete Sequenzen sinnvoll sind (`insert` und `remove`).

Da die Implementierung des 2-3-Fingerbaums persistent sein soll, werden statt gewöhnlichen Pointern `std::shared_ptr` verwendet, um auf geteilte Knoten und Unterbäume zu zeigen. Jeder in Haskell definierte Typ `T` wird hier durch eine Klasse `T` direkt abgebildet, welche zwei Felder enthält:

- `_repr`: Einen `std::shared_ptr<TBase>`.

- `_kind`: Einen Diskriminator, welcher angibt welche Variante in `_repr` enthalten ist (insofern nötig).

Varianten werden als ableitende Klassen von `TBase` implementiert. Alle Schreibzugriffe auf `T` sorgen dann für eine flache Kopie der im `_repr`-Felds verwiesenen Instanz. Damit wird sicher gegangen, dass Persistenz erhalten wird (siehe Abschnitt 7.2.7 für mögliche Optimierungen). Diese Trennung ist auch nötig, um den Übergang einer Variante in die andere möglich zu machen, ohne das der Nutzer der Klasse diese selbst verwalten muss.

Die folgenden Definitionen und Codeausschnitte verzichten aus Platzgründen auf redundante Sprachkonstrukte wie etwa die Template-Parameterdeklarationen `template <typename K, typename V>`, da diese prinzipiell an fast jeder Klasse vorkommen. Es wird außerdem darauf verzichtet für jede Klasse die Getter der Felder zu definieren. Für alle Felder aller Typen werden Getter wie folgt definiert: Sei `_field` ein Feld vom Typen `Type` der Klasse `Class`, so existiert eine Methode `auto Class::_field() const -> const& Type`.

5.1.1 Node

Damit die rekursiven Definitionen der Algorithmen in der Lage sind einen einzelnen Wert in Form eines Knotens zurück zu geben, muss es eine `NodeLeaf`-Variante geben, welche nur einen Wert und dessen Schlüssel enthält. Zur Bildung der rekursiven Struktur gibt es eine `NodeDeep`-Variante, welche sowohl `Node2` als auch `Node3` als `std::vector` abbildet. Wie zuvor bereits erklärt, erben diese von einer gemeinsamen Klasse `NodeBase`, welche durch `Node` persistent verwaltet wird.

Für die Knoten ist kein Übergang zwischen verschiedenen Zuständen nötig, Blattknoten werden durch die Hilfsfunktionen in `FingerTree` erstellt oder aufgelöst und interne Knoten werden innerhalb der rekursiven Algorithmen erstellt oder aufgelöst. Dadurch ist die API und Implementierung von Knoten verhältnismäßig simpel.

Listing 5-2: Die Definition der Node-Klassen.

```

1  enum class Kind { Deep, Leaf };
2  class NodeBase {};
3  class Node {
4      Kind _kind;
5      std::shared_ptr<NodeBase<K, V>> _repr;
6  };
7  class NodeDeep : public NodeBase<K, V> {
8      uint _size;
9      K _key;
10     std::vector<Node<K, V>> _children;
11 };
12 class NodeLeaf : public NodeBase<K, V> { K _key; V _val; };

```

Listing 5-2 zeigt die Definition der Node-Klassen. Die Felder `_size` und `_key` in `NodeDeep` sind die akkumulierten Werte der Kinder dieses Knotens, sprich die Summe von `_size` und der größte Schlüssel der Kindknoten. Für `LeafNode` liefert der Getter für `_size` direkt 1 und durch die inherente Sortierung der Schlüssel ist der größte Schlüssel immer der im rechten Kindknoten.

5.1.2 Digits

Die Digits eines tiefen 2-3-Fingerbaums könnten, ähnlich wie in [HP06, S. 8] vorgeschlagen, als vier Varianten mit jeweils 1-4 Knoten als Feldern umgesetzt werden. Das würde allerdings nur unnötig die Implementierung erschweren. Daher gibt es für Digits nur eine Variante, `DigitsBase`. Listing 5-3 zeigt die Definition von `Digits` und `DigitsBase`. Da `Node` selbst bereits Persistenz umsetzt, könnte `Digits` darauf verzichten und diese Knoten direkt als `std::vector` speichern und diesen kopieren. Auf eine solche Optimierung wurde aus Zeitgründen vorerst verzichtet, da ständiges Kopieren von Vektoren, wenn auch klein (1-4 Elemente), teurer sein kann als die geringere Verzögerung durch weniger Indirektion. Dadurch ist die Definition auch einfacher mit denen der anderen Klassen zu vergleichen.

Listing 5-3: Die Definition der Digits-Klassen.

```

1  class DigitsBase {
2      uint _size;
3      K _key;
4      std::vector<Node<K, V>> _digits;
5  };
6  class Digits {
7      std::shared_ptr<DigitsBase<K, V>> _repr;
8  };

```

Ähnlich wie auch in Listing 5-2, gibt es akkumulierte `_size`- und `_key`-Felder, diese haben den gleichen Zweck wie die in `NodeDeep`. Da es nur eine Variante gibt, kann auf das `_kind`-Feld in `Digits` verzichtet werden. Die API von `Digits` verhält sich ähnlich wie die einer `Deque`, mit spezialisierten Funktionen für `push` und `pop` an beiden Seiten. Desweiteren werden Funktionen bereitgestellt, um den Über- und Unterlauf zu vereinfachen; diese entpacken oder verpacken mehrere Knoten.

5.1.3 FingerTree

Listing 5-4 zeigt die gleiche Struktur wie die vorherigen Definitionen und baut auf denen von `Node` und `Digits` auf. Direkt fällt auf, dass die Variante `Empty` nutzlos erscheint, die Abwesenheit von Knoten könnte auch durch `_repr == nullptr` abgebildet werden. Damit wird auch verhindert, dass `Move`-Konstruktoren von `FingerTree` alte Instanzen in einem uninitialisierten Stand lassen, wie es bei `Digits` und `Node` der Fall ist. Ähnlich wie bei `Digits` wird auf solch eine Optimierung verzichtet. Das vereinfacht den Vergleich zu den Haskell-Definitionen.

Listing 5-4: Die Definition der FingerTree-Klassen.

```

1  enum class Kind { Empty, Single, Deep };
2  class FingerTreeBase {};
3  class FingerTree {
4      Kind _kind;
5      std::shared_ptr<FingerTreeBase<K, V>> _repr;
6  };

```

```
7  class FingerTreeEmpty : public FingerTreeBase<K, V> {};  
8  class FingerTreeSingle : public FingerTreeBase<K, V> {  
9      Node<K, V> _node;  
10 };  
11 class FingerTreeDeep : public FingerTreeBase<K, V> {  
12     Digits<K, V> _left;  
13     FingerTree<K, V> _middle;  
14     Digits<K, V> _right;  
15 };
```

6 Analyse & Vergleich

Zusätzlich zur theoretischen Analyse wurden zu verschiedenen verwendeten Datenstrukturen Benchmarks durchgeführt, um zu testen, ob eine simple Implementierung die theoretischen Erwartungen bestätigen kann. Zur Kontrolle der Ergebnisse der neuen Implementierung wurden verschiedene Benchmarks mit den bis dato in T4gl verwendeten Datenstrukturen, sowie einer naiven persistenten B-Baum-Implementierung durchgeführt. Die Tests beschränken sich auf die für die Arbeit relevanten Szenarien. Dazu zählen Lesezugriffe und verschiedene Szenarien, in welchen worst-case Verhalten getestet wurde. Dadurch sollen vor allem die Vor- und Nachteile granular-persistenter Datenstrukturen wie 2-3-Fingerbäumen gegenüber grob-persistenten wie QMap hervorgehoben werden.

6.1 Umgebung & Auswertung

Die Benchmarks wurden mit `google/benchmark v1.9.0` und `Qt 5.15.15` durchgeführt. Kompilation erfolgte mit `g++ (gcc) 14.2.1 20240910` und `C++ 20`. Für die Kompilation wurde Optimierungslevel 2 (`-O2`), sowie Link-Time-Optimization (`-lto, -fno-fat-lto-objects`) verwendet. Sowohl Benchmarking und Kompilation erfolgten auf Arch Linux `6.10.10-arch1-1` mit einem 11th Gen Intel® Core™ i7-11 auf einem Lenovo ThinkPad E15 G2 i7-1165G7 TS.

Bei der Darstellung der Zeiten wird die gemessene Systemzeit verwendet und auf 2 Kommastellen genau angezeigt. Die Iterationen-Spalte gibt die Anzahl der getesteten Aufrufe an, jede Iteration wurde mit den gleichen Ursprungsbedingungen getestet. Bei höheren Zeiten senkt `google/benchmark` die Anzahl der Iterationen automatisch ab, sodass pro Szenario etwa eine Sekunde lang Werte erfasst werden. Die Werte der Zeit-Spalte ergeben sich aus dem Durchschnitt aller Iterationen. Da es sich lediglich um Datenstrukturen im Arbeitsspeicher handelt, weicht die CPU-Zeit nicht weit von der gemessenen Systemzeit ab und wurde daher nicht mit in die Tabellen aufgenommen. Ähnlich wie auch beim Source Code werden die Daten im archivierten Repo vollständig mit aufgeführt und können unter `src/benchmarks.json` gefunden werden.

6.2 Kontrollgruppen

6.2.1 QMap

Zunächst werden zwei Szenarien getestet, welche das Kernproblem der QMap-Implementierung hervorheben sollen: wiederholte Schreibzugriffe bei nicht-einzigartigen Referenzen. Zusätzlich dazu wird auch ein durchschnittlicher Lesezugriff gemessen.

Tabelle 6-1: Die verschiedenen Iterationen der QMap Benchmarks.

Szenario	Größe	Zeit	Iterationen
get	2048	49.78 ns	13405287
	4096	61.14 ns	11300528
	8192	74.97 ns	8966928
	16384	89.30 ns	7655355
	32768	105.45 ns	6439869

Szenario	Größe	Zeit	Iterationen
	65536	129.32 ns	5280055
	131072	154.42 ns	4548232
	262144	234.40 ns	3118914
	524288	434.27 ns	1606491
insert_unique	2048	679.69 ns	1690672
	4096	672.05 ns	1000000
	8192	677.02 ns	1720568
	16384	673.36 ns	1683651
	32768	698.51 ns	1000000
	65536	696.99 ns	1613490
	131072	779.39 ns	1261392
	262144	779.30 ns	1076524
	524288	819.62 ns	940456
insert_shared	2048	231542.59 ns	13114
	4096	193917.76 ns	5695
	8192	254928.99 ns	3150
	16384	503994.78 ns	1000
	32768	1157190.05 ns	587
	65536	2559225.59 ns	269
	131072	8174316.06 ns	82
	262144	31706858.23 ns	22
	524288	88892347.42 ns	12

Tabelle 6-1 zeigt die zuvor genannten Szenarien. Das erste Szenario `get` zeigt einen durchschnittlichen Lesezugriff auf QMaps verschiedener Größen. Die Szenarien `insert_unique` und `insert_shared` testen das Einfügen von Werten für eine QMap mit einem Referenten (`unique`) und mehreren Referenten (`shared`). Dabei ist ein klarer Sprung der benötigten Zeit zwischen beiden Szenarien zu sehen. Sobald eine QMap-Instanz nicht der einzige Referent ist, muss der gesamte Speicher der QMap kopiert werden, um ein einziges Element hinzuzufügen.

6.2.2 Persistenter B-Tree

Da 2-3-Fingerbäume von 2-3-Bäumen abgeleitet wurden und keine Generalisierung der Zweigfaktoren gelungen ist, ist als Kontrolle auch eine simple B-Baum Implementierung vorhanden. Ähnlich wie die der 2-3-Fingerbäume, ist diese Implementierung granular-persistent, die Persistenz erfolgt auf jeder Knotenebene.

Tabelle 6-2: Die verschiedenen Iterationen der B-Baum Benchmarks.

Szenario	Größe	Zeit	Iterationen
get	2048	67.02 ns	10043411
	4096	73.37 ns	8937358
	8192	83.26 ns	7740631

Szenario	Größe	Zeit	Iterationen
	16384	92.13 ns	7267631
	32768	106.57 ns	6555362
	65536	115.42 ns	5944763
	131072	129.41 ns	5405401
	262144	144.63 ns	4742297
	524288	168.70 ns	4125653
insert	2048	477.33 ns	1496740
	4096	491.39 ns	1387006
	8192	564.65 ns	1190522
	16384	648.50 ns	1035268
	32768	759.21 ns	954489
	65536	773.03 ns	908970
	131072	823.41 ns	856259
	262144	889.31 ns	777519
	524288	1076.68 ns	658446

Tabelle 6-2 zeigt zwei Szenarien. Ähnlich wie bei QMap wird der durchschnittliche Lesezugriff sowie das Einfügen von Werten gemessen. Da die B-Baum Implementierung generell eine Pfadkopie bei einem Schreibzugriff erzeugt, selbst wenn diese Instanz der einzige Referent ist, ist für das Einfügen nur ein Szenario vorhanden.

6.3 2-3-Fingerbäume

Für 2-3-Fingerbäume wurden verschiedene Szenarien getestet, aber nicht direkt Szenarien für das Einfügen von Werten in den Baum. Da die Implementierung von 2-3-Fingerbäumen als geordnete Sequenzen insert durch split, gefolgt von push und dann concat umgesetzt, kann dabei nicht ohne Weiteres ein worst-case für alle drei Operationen erzeugt werden, da diese voneinander abhängen. Stattdessen wird für alle drei Operationen das worst-case Szenario gemessen, um daraus Schlüsse auf die worst-case Performance von insert zu ziehen.

Tabelle 6-3 enthält fünf Szenarien:

- get wie zuvor als durchschnittlicher Lesezugriff,
- split für das Trennen von 2-3-Fingerbäumen,
- concat für das Zusammenführen von 2-3-Fingerbäumen,
- push_worst als worst-case push-Szenario und
- push_avg als durchschnittliches push-Szenario.

Die Szenarien split und concat testen dabei mit Bäumen, welche gezielt aufgebaut wurden um den Worst Case zu simulieren. Bei split werden zufällige Werte zwischen INT_MIN und INT_MAX durch std::rand() generiert und eingefügt. Gleichmaßen werden diese Werte behalten und deren Median verwendet, um bei einem möglichst tiefen Punkt im Baum die Trennung zu erzwingen. Das geschieht unter der Annahme, dass die gleichmäßig verteilten Werte von std::rand() einen relativ balancierten Baum erzeugen, in welchem der Median im tiefsten Knoten liegt. Die Schlüssel für das Szenario get werden ebenfalls auf diese Weise ausgesucht, um möglichst tief in den Baum gehen zu müssen.

Im Szenario `concat` wird ein 2-3-Fingerbaum mit sich selbst verknüpft. Dabei wird zwar zwangsläufig die Ordnungsrelation der Schlüssel ignoriert, diese hat aber keinen Einfluss auf die Implementierung von `concat` oder anderweitige Implementierungsdetails des Benchmarkszenarios. Die Verknüpfung mit sich selbst ermöglicht dabei ein Szenario, in welchem kein seichter Baum existiert, welcher die Rekursion frühzeitig stoppen würde.

Die Szenarios `push_worst` und `push_avg` zeigen jeweils die worst-case und average-case Szenarien von `push`. Zur Erstellung des worst-case Szenarios wurden Bäume erstellt, bei welchen ein weiterer `push` bis in die tiefste Ebene überläuft.

Das Szenario `push_avg` zeigt, warum es nicht einfach ist, direkt `insert` zu testen. Es müssten Bäume erstellt werden, welche nach ihrer Trennung genau so aufgebaut sind, dass auch die `push`-Operation danach den Schlimmstfall zeigt, genau so wie `concat` nach dieser. Es ist nicht trivial, die genaue Struktur der Bäume zu kontrollieren, vor allem bei hohen Datenmengen. Bei Versuchen, die `insert`-Operation direkt zu messen, zeigte sich das vor allem in stärkerem Rauschen der erfassten Zeiten. Daher werden zum Einordnen von `insert` die individuellen worst-case Ergebnisse von `split`, `push_worst` und `concat` verwendet, auch wenn diese möglicherweise nie zusammen auftreten können.

Tabelle 6-3: Die verschiedenen Iterationen der 2-3-Fingerbaum Benchmarks.

Szenario	Größe	Zeit	Iterationen
get	2048	59.42 ns	13791018
	4096	78.39 ns	8538748
	8192	88.15 ns	6328726
	16384	150.61 ns	7606444
	32768	111.74 ns	6082595
	65536	139.48 ns	5814516
	131072	212.29 ns	7083590
	262144	285.92 ns	4634707
	524288	431.31 ns	1993965
push_worst	1820	1273.95 ns	529834
	5465	1496.93 ns	459721
	16400	1771.21 ns	395180
	49205	1939.19 ns	350308
	147620	2192.56 ns	318731
	442865	2413.30 ns	290844
push_avg	2048	92.98 ns	7139394
	4096	94.41 ns	7227637
	8192	92.73 ns	7581923
	16384	92.39 ns	6902359
	32768	91.24 ns	7370532
	65536	98.77 ns	6880169
	131072	94.12 ns	7581663
	262144	93.54 ns	7190740
	524288	103.23 ns	7271701

Szenario	Größe	Zeit	Iterationen
concat	1820	2573.52 ns	266980
	5465	3083.24 ns	215827
	16400	3522.57 ns	199929
	49205	4004.89 ns	175535
	147620	4509.77 ns	155332
	442865	5032.79 ns	138996
split	1820	1562.11 ns	443562
	5465	1885.92 ns	370596
	16400	2093.28 ns	322924
	49205	2396.90 ns	289782
	147620	2681.93 ns	258808
	442865	3018.57 ns	233736

Die zuvor genannten Szenarien sind in Tabelle 6-3 zu sehen. Auffällig sind dabei direkt die Sprünge in get bei 65536 und 524288, es ist unklar, ob es sich um ein Implementierungsproblem handelt oder ein inherentes Problem mit 2-3-Fingerbäumen. Mehrere Durchläufe der Benchmarks zeigten ähnliche Sprünge bei den gleichen Werten auf.

6.4 Vergleich

Bei Lesezugriffen steigt t für alle Datenstrukturen etwa logarithmisch zu n , jedoch mit verschiedenen Faktoren. 2-3-Fingerbäume zeigen dabei einen besonders hohen Faktor und schneiden schlechter ab. Bei QMap gibt es allerdings bei 200.000 bis 500.000 eine Verschlechterung, dort gibt es einen linearen Sprung (zu sehen in Tabelle 6-1, Szenario get). Das stimmt mit den theoretischen Erwartungen überein, während die zuvor erwähnten Sprünge der 2-3-Fingerbäume noch unerklärt bleiben.

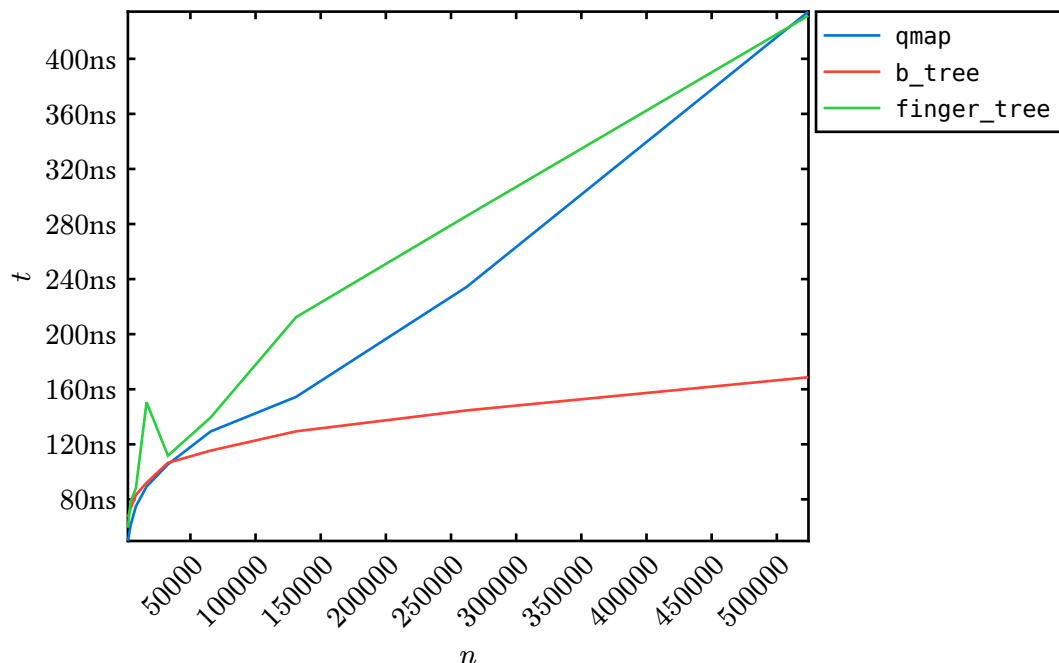


Abbildung 6-1: Vergleich der Lesezugriffe in Abhängigkeit der Anzahl der Elemente n .

Für den Vergleich der insert-Operation werden für 2-3-Fingerbäume die Datenpunkte der split-, push_worst- und concat-Operationen aufsummiert. Es ist möglich, dass selbst im realen worst-case solche Werte nie gemeinsam vorkommen. So ist zum Beispiel nach dem worst-case push der Baum dort minimal auf jeder Ebene besetzt, wo dieser für den concat worst-case maximal besetzt sein müsste. Die Summierung aller worst-case Werte gibt daher eine besonders pessimistische, aber sichere Aussage über die worst-case Performance von insert.

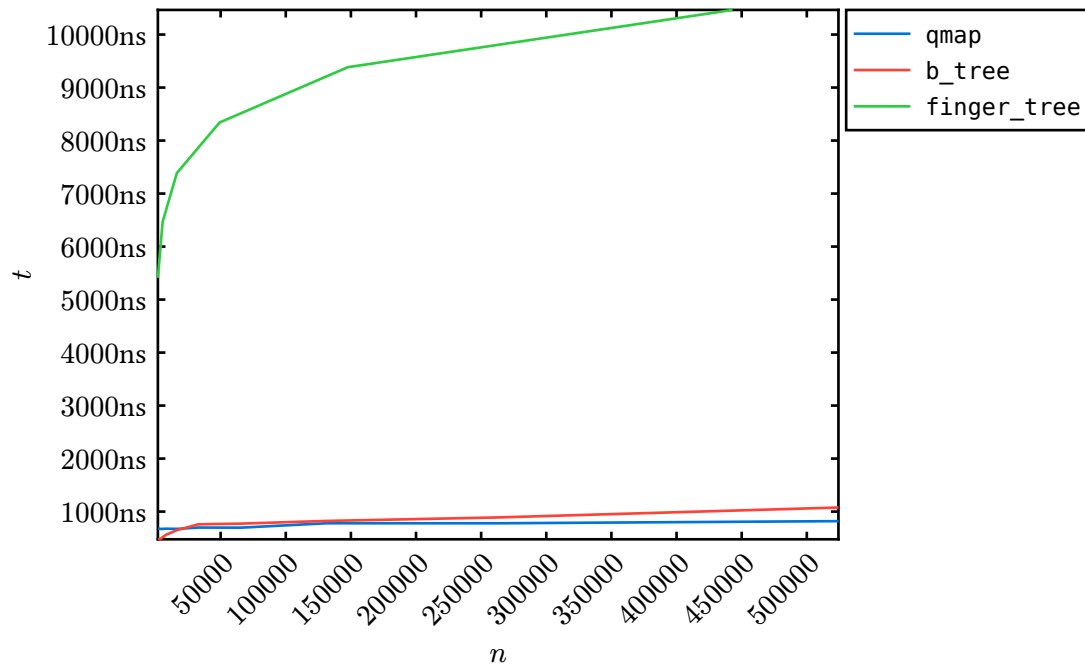


Abbildung 6-2: Vergleich der Schreibzugriffe in Abhängigkeit der Anzahl der Elemente n .

Abbildung 6-2 zeigt für B-Baum das Szenario insert, für QMap das Szenario insert_unique und für 2-3-Fingerbaum die Summe der Szenarien spit, push_worst und concat. Dabei ist zu beachten, dass das worst-case QMap Szenario so hohe Werte erzielt, dass die Werte von B-Baum und 2-3-Fingerbaum gleich aussehen (siehe Tabelle 6-1, Szenario insert_shared). Die Abbildung zeigt, dass 2-3-Fingerbäume in ihrer jetzigen Implementierung deutlich schlechter abschließen als QMap im best-case Szenario und die B-Baum-Implementierung generell. Allerdings schließen beide Baum-Implementierungen weitaus besser als QMap ab sobald mehr als ein Referent existiert, wie es in T4gl oft der Fall ist.

7 Fazit

7.1 Ergebnis

Aus den Benchmarks in Kapitel 6 kann geschlossen werden, dass persistente Baumdatenstrukturen vor allem im Worst Case bessere Performance liefern können. In ihrer jetzigen Implementierung sind 2-3-Fingerbäume keine gute Wahl für die Storage-Datenstruktur von T4gl-Arrays. Eine simple B-Baum Implementierung ohne Optimierung war in der Lage in den untersuchten Szenarios vergleichbare oder bessere Performance als QMap zu bieten. Unter Betrachtung weiterer Szenarien wird davon ausgegangen, dass dieser Trend fortbesteht und das worst-case Zeitverhalten von T4gl-Arrays drastisch verbessern kann.

7.2 Optimierungen

Verschiedene Optimierungen können die Performance der 2-3-Fingerbaum-Implementierung verbessern. Allerdings ist unklar, ob diese ausreichen, um die Performance der persistenten B-Baum-Implementierung zu erreichen, welche ähnlich unoptimiert implementiert wurde.

7.2.1 Pfadkopie

Die Implementierung von insert der 2-3-Fingerbäume stützt sich auf eine simple, aber langsame Abfolge von split, push und concat. Es ist allerdings möglich stattdessen eine Variante mit Pfadkopie und internem Überlauf zu implementieren. Bei Einfügen eines Blattknotens in einer unsicheren Ebene, kann das maximal zu einem neuen Knoten pro Ebene führen, ähnlich dem worst-case von push.

7.2.2 Lazy-Evaluation

Das Aufschieben von Operationen durch Lazy Evaluation hat einen direkten Einfluss auf die amortisierten Komplexitäten der Deque-Operationen [HP06, S. 7]. Da für die Echtzeitanalyse der Datenstruktur nur die worst-case Komplexitäten relevant sind, wurde diese allerdings vernachlässigt.

Zur generellen Verbesserung der durchschnittlichen Komplexitäten der Implementierung ist die Verwendung von Lazy Evaluation unabdingbar.

7.2.3 Generalisierung & Cache-Effizienz

Der Cache einer CPU ist ein kleiner Speicher, zwischen CPU und RAM, welcher generell schneller zu lesen und schreiben ist. Ist ein Wert nicht in dem CPU-Cache, wird in den meisten Fällen beim Lesen einer Adresse im RAM der umliegende Speicher mit in den Cache gelesen. Das ist einer der Gründe, warum Arrays als besonders schnelle Datenstrukturen gelten. Wiederholte Lese- und Schreibzugriffe im gleichen Speicherbereich können häufig auf den Cache zurückgreifen. In Präsenz von Indirektion, also der Verwendung von Pointern wie bei Baumstrukturen, können Lese- und Schreibzugriffe in den Speicher öfter auf Bereiche zeigen, welche nicht in dem Cache liegen, dabei spricht man von einem Cache-Miss.

Abschnitt 4.6 beschreibt einen Versuch die Cache-Effizienz von Fingerbäumen zu erhöhen, indem durch höhere Zweigfaktoren die Tiefe der Bäume reduziert wird. Durch die geringere Tiefe sollen die rekursiven Algorithmen welche den Baumknoten folgen weniger oft Cache-Misses verursachen.

Für verschiedene Teile der generalisierten Zweigfaktoren von Fingerbäumen konnten keine Beweise vorgelegt werden. Es wurden allerdings auch keine Beweise gefunden oder erarbeitet, welche die Generalisierung auf höhere Zweigfaktoren gänzlich ausschließen. Je nach Stand der Beweise könnten generalisierte Varianten von Fingerbäumen in Zukunft in T4gl eingesetzt werden. Unklar ist, ob der Aufwand der Generalisierung sich mit der verbesserten Cache-Effizienz aufwiegen lässt.

Die schlechten Ergebnisse der 2-3-Fingerbaum scheinen eine direkte Folge der naiven Implementierung zu sein, da die in [HP06, S. 20] gegebenen Benchmarks exzellente Performance vorweisen. Dabei ist allerdings unklar, wie stark der Einfluss von Lazy Evaluation in Haskell sich auf die Ergebnisse der Benchmarks auswirkt.

7.2.4 Vererbung & Virtual Dispatch

Wird eine Klasse in C++ vererbt und besitzt überschreibbare Methoden, gelten diese als virtuell. Hat eine vererbte Klasse eine Methode ohne eine Implementierung, gilt die Methode, sowie die Klasse selbst als abstrakt. Virtuelle Methoden müssen bei ihrem Aufruf zur Laufzeit zunächst die richtige Implementierung der Methode im Virtual Table finden. Das erfolgt durch eine sogenannte Virtual Table, auf welchen jede abstrakte Klasse und deren erbende Klassen verweisen. Dabei werden für die CPU wichtige Optimierungen erschwert, wie Branch Prediction, Instruction Caching oder Instruction Prefetching.

Die in Listing 4-2 gegebene Definition verwendet Vererbung der Klasse `FingerTree` zur Darstellung der verschiedenen Varianten. Daraus folgt, dass Fingerbäume nicht mehr direkt verwendet werden können, eine `FingerTree`-Instanz selbst ist nutzlos ohne die Felder und Implementierung der vererbenden Varianten. Instanzen von `FingerTree` müssen durch Indirektion übergeben werden, da diese generell auf deren erbende Variante verweisen. Die Operationen auf den verschiedenen Varianten von `FingerTree` müssten entweder durch vorsichtiges casten der Pointer oder durch einheitliche API anhand virtueller Methoden erfolgen. Ersteres ist unergonomisch und fehleranfällig, `FingerTree` wird zwangsläufig zur virtuellen Klasse, daraus folgt:

- dass für viele Methoden Virtual Dispatch verwendet werden muss
- und dass jeder Zugriff auf einen `FingerTree` zunächst die Indirektion auflösen muss (Pointerdereferenzierung).

Zweiteres ist nicht für alle Operationen sinnvoll, manche Operationen wie `split` sind nur für eine Variante sinnvoll implementiert.

Um zu vermeiden, dass jeder Aufruf essentieller Funktionen wie `pop` und `pop` auf Virtual Dispatch zurückgreifen muss, können statt virtuellen Methoden durch gezieltes casten auf die korrekte Variante Cache Misses vermieden werden. Die Auswahl der Klasse kann durch das Mitführen eines Diskriminators erfolgen, welcher angibt, auf welche Variante verwiesen wird. Damit wird sowohl die Existenz des Virtual Table Pointers in allen Instanzen, sowie auch die doppelte Indirektion dadurch vermieden. Besonders häufige Pfade, wie die der Deep-Variante, können dem CPU als heiß vorgeschlagen werden, um diese bei der Branch Prediction zu bevorzugen.

7.2.5 Memory-Layout

Bei C++ hat jeder Datentyp für den Speicher zwei relevante Metriken, dessen Größe und dessen Alignment. Das Alignment eines Datentyps gibt an, auf welchen Adressen im Speicher ein Wert gelegt werden darf. Ist das Alignment eines Typs T 8, kann dieser nur auf die Adressen gelegt werden, welche Vielfache von 8 sind, also 0×0 , 0×8 , 0×10 , 0×18 und so weiter. Da komplexe Datentypen aus anderen Datentypen bestehen, müssen auch diese im Speicher korrekt angelegt werden, deren Alignment wirkt sich auf das des komplexen Datentyps aus. Des Weiteren werden Felder in Deklarationreihenfolge angelegt. Damit die Alignments der einzelnen Feldertypen eingehalten werden, werden wenn nötig vom Compiler unbenutzte Bytes zwischen Feldern eingefügt. Das nennt sich Padding. Durch clevere Sortierung der Felder können Paddingbytes durch kleinere Felder gefüllt werden. Padding zu reduzieren, reduziert die Größe des komplexen Datentyps. Reduziert man die Größe des Datentyps, erhöht man die Anzahl der Elemente, welche von der CPU in deren Cache geladen werden können.

7.2.6 Spezialisierte Allokatoren

Eine mögliche Darstellung von Graphen ist es, Knoten in Arrays zu speichern und deren Verbindungen in Adjazenzlisten zu speichern. Dadurch können mehr Knoten in den CPU-Cache geladen werden als durch rekursive Definitionen. Da Bäume lediglich Sonderformen von Graphen sind, kann das auch auf die meisten Baumdatenstrukturen angewendet werden. Das bedeutet aber auch, dass alle Knoten kopiert werden müssten, welche von einem Graph erwartet werden wenn dieser kopiert wird. Das steht gegen das Konzept der Pfadkopie in persistenten Bäumen.

Eine Alternative, welche die Knoten eines Baums nah beieinander im Speicher anlegen könnten ohne die Struktur der Bäume zu zerstören, sind Allokatoren, welche einen kleinen Speicherbereich für die Knoten der Bäume verwalten. Somit könnte die Cache-Effizienz von 2-3-Fingerbäumen erhöht werden, ohne besonders große Änderungen an deren Implementierung vorzunehmen.

7.2.7 Unsichtbare Persistenz

Da die Persistenz von 2-3-Fingerbäumen durch die API der Klassen versteckt wird, können diese auch auf Persistenz verzichten, wenn es sich nicht auf andere Instanzen auswirken kann. Wenn eine Instanz des Typs T der einzige Referent auf die in `_repr` verwiesene Instanz ist, kann diese problemlos direkt in diese Instanz schreiben, ohne vorher eine Kopie anzufertigen. Das ist das Kernprinzip von vielen Datenstrukturen, welche auf CoW-Persistenz basieren. Dazu gehören auch die Qt-Datenstrukturen, welche ursprünglich in T4gl zum Einsatz kamen.

Das hat allerdings einen Einfluss auf die Art, auf welche eine solche Klasse verwendet werden kann. Wird eine Instanz `t1` angelegt und eine Referenz oder ein Pointer `&t1` wird an einen anderen Thread übergeben, kann zwischen dem Abgleich von `_repr.use_count() == 1` und der direkten Beschreibung der Instanz in `*_repr` eine Kopie von `t1` auf dem anderen Thread angelegt werden. Sprich, der `use_count` kann sich zwischen dem Abgleich und dem Schreibzugriff verändern. Das hat zur Folge, dass der Schreibzugriff in beiden Instanzen `t1` und `t2` sichtbar wird, obwohl der Schreibzugriff nach der Kopie von `t1` stattfand. Um auszunutzen zu können, dass eine Instanz der einzige Referent der Daten in `*_repr` ist, dürfen keine Referenzen oder Pointer zu Instanzen von T an andere Threads

übergeben werden. Stattdessen sollten diese Instanzen im Ursprungsthread kopiert werden, bevor sie an einen anderen Thread übergeben werden.

Literatur

- [Cor+09] T. H. Cormen, C. E. Leiserson, R. L. Rivest, und C. Stein, *Introduction to Algorithms*, 3. Aufl. MIT Press & McGraw-Hill, 2009.
- [Knu76] D. E. Knuth, „Big Omicron and big Omega and big Theta“, *ACM SIGACT News*, Bd. 8, Nr. 2, S. 18–24, Apr. 1976, doi: 10.1145/1008328.1008329.
- [Sch05] P. Scholz, *Softwareentwicklung eingebetteter Systeme*, 1. Aufl. Berlin, Heidelberg: Springer, 2005. doi: 10.1007/3-540-27522-3.
- [LO11] P. A. Laplante und S. J. Ovaska, *Real-Time Systems Design and Analysis*, 4. Aufl. Wiley & IEEE Press, 2011.
- [FK24] A. Fiat und H. Kaplan, „Making data structures confluent persistent“, *Journal of Algorithms*, Bd. 48, Zugegriffen: 28. August 2024. [Online]. Verfügbar unter: <https://www.sciencedirect.com/science/article/pii/S0196677403000440>
- [JTC20] JTC1/SC22/WG21 - The C++ Standards Committee - ISO C++, „Working Draft, Standard for Programming Language C++“, 14. Januar 2020. Zugegriffen: 23. April 2024. [Online]. Verfügbar unter: <https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/n4849.pdf>
- [KT96] H. Kaplan und R. E. Tarjan, „Purely Functional Representations of Catenable Sorted Lists“, in *STOC '96: Proceedings of the twenty-eighth annual ACM symposium on Theory of Computing*, New York, NY, United States: Association for Computing Machinery, Juni 1996, S. 202–211. doi: 10.1145/237814.237865.
- [BR11] P. S. Bagwell und T. Rompf, „RRB-Trees: Efficient Immutable Vectors“. 2011. Zugegriffen: 23. April 2024. [Online]. Verfügbar unter: <https://api.semanticscholar.org/CorpusID:15763144>
- [Bag+15] P. S. Bagwell, T. Rompf, N. Stucki, und V. Ureche, „RRB Vector: A Practical General Purpose Immutable Sequence“, in *ICFP 2015: Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, New York, NY, United States: Association for Computing Machinery, Aug. 2015, S. 342–354. doi: 10.1145/2784731.2784739.
- [Stu15] N. Stucki, „Turning Relaxed Radix Balanced Vector from Theory into Practice for Scala Collections“, 2015. Zugegriffen: 23. April 2024. [Online]. Verfügbar unter: <https://api.semanticscholar.org/CorpusID:60518521>
- [BM70] R. Bayer und E. M. McCreight, „Organization and maintenance of large ordered indices“, in *SIGFIDET '70: Proceedings of the 1970 ACM SIGFIDET Workshop on Data Description, Access and Control*, New York, NY, United States: Association for Computing Machinery, Nov. 1970, S. 107–141. doi: 10.1145/1734663.1734671.
- [Bay71] R. Bayer, „Binary B-trees for virtual memory“, in *SIGFIDET '71: Proceedings of the 1971 ACM SIGFIDET Workshop on Data Description, Access and Control*, New York, NY, United States: Association for Computing Machinery, Nov. 1971, S. 219–235. doi: 10.1145/1734714.1734731.

- [HP06] R. Hinze und R. Paterson, „Finger trees: a simple general-purpose data structure“, *Journal of Functional Programming*, Bd. 16, Nr. 2, S. 197–217, März 2006, doi: 10.1017/S0956796805005769.
- [Knu98] D. E. Knuth, *The Art of Computer Programming Volume 3: Sorting and Searching*, 2. Aufl., Bd. 3. Reading, Massachusetts, USA: Addison-Wesley, 1998.
- [Oka98] C. Okasaki, *Purely Functional Data Structures*. The Pitt Building, Trumpington Street, Cambridge, United Kingdom: Cambridge University Press, 1998. doi: 10.1017/CBO9780511530104.
- [BM98] R. Bird und L. Meertens, „Nested datatypes“, in *Mathematics of Program Construction*, J. Jeuring, Hrsg., in Lecture Notes in Computer Science, vol. 1422. Springer Berlin Heidelberg, 1998, S. 52–67. doi: 10.1007/BFb0054285.

Danksagung

Ich bedanke mich bei Kay Gürtzig für die wissenschaftliche Genauigkeit seines Feedbacks und die viele Zeit die er trotz vieler andere Pflichten in zahllose Rücksprachetermine investiert hat. Durch sein genaues Hinsehen wurden viele Fehler gefunden, welche gerade bei Änderungen schnell vergessen werden. Gleichmaßen bedanke ich mich bei Peter Brückner und Ralf Müller für deren Betreuung von seiten der Firma Brückner und Jarosch Ingenieuresellschaft mbH (BJ-IG). Ihre Unterstützung, Zeit und Vorschläge haben dann geholfen wenn Ergebnisse unrealistisch oder Beweise unmöglich erschienen. Desweiteren bedanke ich mich bei allen Problelesern, welche mir die Fehler gezeigt haben, welche man als Autor nach dem 30. mal Lesen des eigenen Texts nicht mehr sieht. Ohne die Unterstützung meiner Kollegen bei BJ-IG, meine probelesenden Freunde und Betreuer wäre ich nicht so weit gekommen. Ich bedanke mich bei allen Freunden und Familienmitgliedern, welche einfach nur da waren, wenn ich an etwas anderes denken wollte als der herannahende Abgabe Termin.

In der Hoffnung, dass ich schon morgen anderen so helfen kann wie sie mir geholfen haben, danke!

Eigenständigkeitserklärung

Ich, Erik Bünnig, versichere hiermit, dass ich die vorliegende Masterarbeit mit dem Titel

Dynamische Datenstrukturen unter Echtzeitbedingungen

selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel
angefertigt habe.

Erfurt, 09.10.2024

Erik Bünnig