FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Untersuchung der Implementierbarkeit eines lock-freien binären Suchbaumes

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Nick Mayerhofer, BSc
Matrikelnummer 0726179

an der Fakultät für Informatik
der Technischen Universität Wien
Betreuung: Prof. Dr. Jesper Larsson Träff

Wien, 20. August 2016

Nick Mayerhofer

Jesper Larsson Träff

Erklärung zur Verfassung der Arbeit

Nick Mayerhofer, BSc
Draschestrasse 59/9, 1230 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 20. August 2016

Nick Mayerhofer

Danksagung

Danke an die sozialstaatliche Leistung Österreichs zur Förderung Studierender, sowie an meine Partnerin Barbara für ihre Aufopferung.

[1] https://gitlab.com/nickma/LockFreeBST-Thesis

Kurzfassung

So wie wir Menschen, verbringen auch Computer eine beträchtliche Zeit mit Datenverarbeitung: Sie schlagen unter anderem „Kontostände, Flugreservierungen, Rechnungen und Zahlungen“ [DMS14, S.183] nach. Um diese Vorgänge für Computer zu beschleunigen, können mehrere Prozessoren parallel dieselben Datensätze bearbeiten, was in den Bereich des parallelen Rechnens fällt.
Die effiziente Datenmanipulation von mehreren Prozessoren gleichzeitig bedarf neuer Datenstrukturen, die für diesen Einsatz ausgelegt sind. Lock-freie Datenstrukturen tragen dieses Potential, da sie nicht auf blockierende Synchronisationstechniken zurückgreifen. In dieser Arbeit wird der Lock-freie Algorithmus von Chatterjee et al.[CNT14] zur parallelen Manipulation einer Baum-Datenstruktur untersucht.

Diese Diplomarbeit beschäftigt sich mit der Frage, ob der Algorithmus von Chatterjee et al. so ausführlich beschrieben wurde, dass er sich im Rahmen dieser Arbeit in ein ausführbares Programm implementieren lässt. Weiters sollte die Güte der Implementierung des parallelen Algorithmus anhand der geleisteten Performance untersucht werden [RR10, S.167]. Durch die Anwendung eines Test-Driven-Development Prozesses, in Kombination mit dem fein granularen Debugging Verfahren namens Tracing und einer eigens entwickelten Fehler-Injektion konnten die Kernelemente des Algorithmus implementiert werden.

Letztendlich konnte der Algorithmus von Chatterjee et al. bis auf einen parallelen Spezialfall vollständig und mit allen Funktionalitäten implementiert werden. Das Ergebnis deutet darauf hin, dass Pseudocode-Teile zur Verarbeitung eines speziellen Zustandes, die der Baum einnehmen kann, von Chatterjee et al. nicht ausgeführt wurden.
Abschließend sei erwähnt, dass die Anwendung des Trace-Debuggings Verfahrens als Erfolg zu verbuchen war.

Abstract

Like humans, computers also spend a considerably great amount of time on data processing: They look up account balances, flight reservations, invoices and payments [DMS14, S.183]. To speed up these processes for computers, multiple processors can process the same data in a parallel way, which corresponds to the scope of concurrent calculations. Efficient data manipulation of multiple processors at the same time requires new data structures which are designed for this use. Lock-free data structures carry this potential, as they do not rely on blocking synchronization techniques. In this work, the lock-free algorithm for parallel manipulation of the tree data structure by Chatterjee et al. is examined.

This thesis deals with the question of whether the algorithm of Chatterjee et al. is described in a way that it can be implemented to work as an executable program. Furthermore, the quality of the parallel implementation of the algorithm was examined with the help of an experimental performance analysis [RR10, S.167].

Due to the use of a test-driven development process, constraints were very closely monitored. Thereby it was possible to discover and fix errors in the course of the implementation process. By applying the finely grained debugging process called *tracing* and a specially developed error injection, it was possible to obtain a deep insight into the parallel flow of the application. Thus it was possible to realize important parts of the implementation.

Ultimately, the algorithm of Chatterjee et al. could not be implemented completely. However, it was possible to implement all functionalities but a parallel special case of the algorithm. The result denotes that parts of the pseudo code of Chatterjee et al., which process a particular condition, may be missing.

Finally, please note that the practice of trace debugging has been accredited as successful.

Inhaltsverzeichnis

KAPITEL 1

Einleitung

1.1 Grundlegende Problemstellung

In dieser Arbeit wird der Algorithmus zur Lock-freien parallelen Manipulation des binär geordneten Suchbaums von Chatterjee et al.[CNT14] untersucht, fortan wird dieser als untersuchter Algorithmus bezeichnet. Die Autoren stellen einen Lock-freien Algorithmus zur parallelen Manipulation ihrer Baum-Datenstruktur vor. Dies ist wichtig, da High-Performance Datenverarbeitungen auf parallele Abarbeitungen angewiesen sind.

Ebenfalls wichtig ist eine praktische Evaluierung neuer Verfahren, was für diesen Algorithmus nach Kenntnisstand des Autors noch nicht geschehen ist. Deshalb ist das Ziel dieser Arbeit, eine lauffähige Applikation auf Basis des vorgestellten Verfahrens zu implementieren und diese auf Funktionalität, die behaupteten Zusicherungen und Performance zu prüfen.

Die wichtigste der behaupteten Zusicherungen wird Lock-Free genannt, diese soll laufenden Fortschritt eines parallelen Systems sicherstellen. Weitere Besonderheiten des untersuchten Algorithmus sind eine Zeitkomplexitätsanalyse des Algorithmus, eine Durchführung mit ausschließlich Single-Word-CAS Operationen, welche als besonders performant gelten [Gra15, S.1,5], sowie eine neuartige Fehlerbehandlung in parallelen Konfliktfällen.

1.2 Motivation

So wie wir Menschen, verbringen auch Computer eine beträchtliche Zeit mit Datenverarbeitung: Sie schlagen unter anderem „Kontostände, Flugreservierungen, Rechnungen und Zahlungen“ [DMS14, S.183] nach. Um diese Vorgänge für Computer zu beschleunigen, können mehrere Recheneinheiten parallel dieselben Datensätze bearbeiten, was in den Bereich der parallelen Programme fällt.

Parallele Programmierung und die Gestaltung effizienter paralleler Programme sind im wissenschaftliches Bereich in Form von High-Performance Computing seit Jahren etabliert. Diese werden in Bereichen wie Datenbanken, Simulationen oder der Bearbeitung komplexer Probleme benötigt [TP14, S.1].

Auf herkömmliche Datenstrukturen kann ausschließlich sequenziell zugegriffen werden, weshalb auch keine Leistungssteigerung durch paralleles Rechnen erzielt werden kann. Diese Datenstrukturen sind damit an die Maximalleistung eines Rechenkerns gebunden. Für parallele Zugreifbarkeit von Datenstrukturen müssen gesondert Vorkehrungen getroffen werden. Dies betrifft im Speziellen Fortschrittsgarantien an die Zugreifbarkeit der Datenstruktur, da eine Blockade aller parallelen Zugriffe ein „übliches [..] Problem“ [RR10, S.6] darstellt. Lock-free stellt eine solche Fortschrittsgarantie dar.

In einer Menge von Threads ist Lock-Free eine strikte Fortschrittszusicherung, die besagt, dass zu jeder Zeit mindestens ein Thread Fortschritte verzeichnet. Lock-freier paralleler Datenverarbeitung liegen komplexe Verfahren zur Umgehung von Lock-basierten Synchronisationstechniken zugrunde, was Forschung auf diesem Gebiet *so wichtig macht.*

1.3 Erwartetes Resultat

Im Rahmen dieser Arbeit soll praktisch evaluiert werden, ob die Performance des Algorithmus den Aussagen und Behauptungen der Autoren entspricht oder nicht. Dies betrifft im Speziellen die Zeitkomplexitätsanalyse der Baum-Operationen *Insert*, *Remove* und *Contains*, über die Knoten in den Baum eingefügt, entfernt und aufgefunden werden können. Die Hypothese der Komplexität ist in Abhängigkeit der Knotenzahl n, der Baumhöhe $H(n)$ und der Anzahl der Konflikte c im Zuge eines Baumzugriffs mit der Obergrenze $O(H(n) + c)$ angegeben.
Probleme, die bei der Evaluierung auftreten, sollen festgehalten werden. Diese Ergebnisse sind für jede Form der weiteren theoretischen und praktischen Verwendung sowie im Sinne der Nachvollziehbarkeit relevant.

1.3.1 Fokus und Abgrenzung

Das Ziel ist eine funktionsfähige Implementierung des BST Algorithmus beschrieben von Chatterjee et al. im Werk [CNT14]. Die Implementierung soll sich aus Gründen der Nachvollziehbarkeit, der Vergleichbarkeit des Quellcodes und der Versuchsergebnisse, wie auch zur besseren Fehleranalyse nahe an den Pseudocode halten und in C++ ausgeführt werden.

Die primären Anforderungen an den vorliegenden Algorithmus, stellen die Operationen *Insert* und *Remove* dar, die Updates auf den BST ausführen. Ferner gibt es die Operation *Contains* um die Existens von Elementen im Baum zu prüfen.

Sekundäre Anforderungen betreffen sowohl die Umgebung, als auch die Testbarkeit der Implementierung. Darunter befindet sich das automatische Testen des Quellcodes,

eine Visualisierung der Baumelemente inklusive deren Zustände und der Anspruch an Portierbarkeit.

1.4 Methodische Vorgehensweise

Zur Erstellung der Applikation ist in einem ersten Schritt ein ausgiebiges Studium des Papers der Autoren notwendig, danach kann mit der Implementierung der Baum-Operation begonnen werden. Abschließend kann die Applikation hinsichtlich ihrer Performance analysiert werden.

Um das Forschungsziel zu erreichen und über die Performance Aussagen treffen zu können, werden die für die Baummanipulation nötigen Laufzeiten empirisch ermittelt und ausgewertet. Sollte sich der Algorithmus der Autoren nicht wie erwartet umsetzen lassen, hängt die Performanceanalyse von der erreichten Umsetzung ab, was detailliert im Benchmark (siehe Abschnitt 5.8) erwähnt werden wird.

1.5 Aufbau dieser Arbeit

Das Kapitel 2 *Grundlagen und Definitionen* erläutert Themen und Begriffe, die für diese Arbeit benötigt werden und führt in die relevanten Grundlagen ein.

Das Kapitel 3 *Stand der Forschung* analysiert bestehende Ansätze und Forschungsergebnisse zu paralleler Datenverarbeitung und beschreibt deren Herangehensweise. Weiters bringt es die analysierten Werke in Verbindung mit dieser Masterarbeit und gibt kurze Empfehlungen an Interessenten der Werke ab.

Das Kapitel 4 *Untersuchung des Lock-freien BST Algorithmus* beschreibt die abstrakten Eigenschaften des betrachteten Algorithmus. Darauf folgt eine Beschreibung der Knoten, Links und Daten, welche die Kernstücke des BST bilden und daher zum Verständnis des Algorithmus notwendig sind. Abschließend wird detailliert die Funktionsweise des Algorithmus beschrieben, in der die Knoten, Links und Daten Verwendung finden.

Das Kapitel 5 *Die Implementierung des Algorithmus* beschreibt Ansätze, die zur Implementierung des BST Algorithmus verwendet wurden. Es wird auch auf Probleme sowie Auffälligkeiten eingegangen, die bei der Untersuchung und Implementierung aufgetreten sind.

Das Kapitel 6 *Diskussion der Ergebnisse* beschreibt die primären Erkenntnisse dieser Arbeit und gibt einen Ausblick auf zukünftige Forschungsmöglichkeiten. Zusätzlich findet sich eine Konklusion, die über das geplante und das erreichte Ziel reflektiert.

1.6 Konklusion vorab

Das wichtigste Ziel dieser Arbeit war die Analyse der Implementierbarkeit des untersuchten Algorithmus. Dieses Ziel konnte nicht vollständig erreicht werden, da Probleme zu

Endlosschleifen führten. Das Ergebnis widerlegt jedoch *nicht* die Funktionsfähigkeit des vorgestellten Verfahrens selbst, sondern weist auf Lücken im Pseudocode hin.

Die Implementierung war soweit möglich, dass bis auf einen parallelen Spezialfall alle Funktionalitäten realisiert werden konnten. Gemessen und einem Benchmark unterzogen wurden ausschließlich die voll funktionsfähigen Teile des Algorithmus, nämlich die folgenden zwei Operationen des Baumes: *Insert* und *Contains*.

Abschließend sei gesagt, dass durchaus die Möglichkeit existiert, dass nicht funktionsfähige Teile des Pseudocodes in einer durch Chatterjee et al. überarbeiteten Variante erscheinen, da diesbezüglich schon mit den Autoren Kontakt aufgenommen wurde.

KAPITEL 2

Grundlagen und Definitionen

Dieses Kapitel erläutert Themen und Begriffe, die für diese Arbeit benötigt werden und führt in die relevanten Grundlagen ein. Es hält sich stark an die Begrifflichkeiten, welche in den folgenden Kapiteln *Untersuchung des Lock-freien BST Algorithmus*, sowie *Die Implementierung des Algorithmus* verwendet werden.

Hierzu gibt das Kapitel zunächst eine Einführung in das Konzept nebenläufiger Datenzugriffe beziehungsweise nebenläufiger Programmierung und die Arten von Herangehensweisen, um Nebenläufigkeit zu gewährleisten. Weiters wird an der aktuelle Bedarf an nebenläufigen Datenstrukturen gezeigt. Abschließend wird an die Begriffe *Lock-Free* und *Wait-Free* herangeführt, wobei ersteres eine zentrale Eigenschaft des Algorithmus von Chatterjee et al. darstellt.

2.1 Parallele Datenverarbeitung

Parallele Datenverarbeitung[1] bietet mehreren sequenziellen Programmen gleichzeitig die Möglichkeit, „eine wesentlich höhere Rechenleistung zu nutzen, als sie sequentielle Rechner bereitstellen, indem mehrere Prozessoren oder Verarbeitungseinheiten gemeinsam eine Aufgabe bearbeiten“ [RR10, S.2].

Der folgende Abschnitt vermittelt eine Übersicht über die Möglichkeiten zur Parallelisierung unter Verwendung von einerseits Hardware und andererseits Software. Sowohl der Unterabschnitt Hardware als auch der der Software Parallelisierung gehen nach einer allgemeinen Übersicht genauer auf die für die vorliegende Arbeit relevanten Spezialisierungen ein. Der Hardware Abschnitt zeigt eine Kategorisierung paralleler Prozessoren nach Flynn [Fly72] und geht genauer auf Hardware-Multi-Threading ein. Der Software Abschnitt geht auf die Bedeutung der Applikationsebene sowie des Betriebssystems ein und fokussiert auf das Betriebssystem-Threading.

[1] Englisch: Parallel Computing

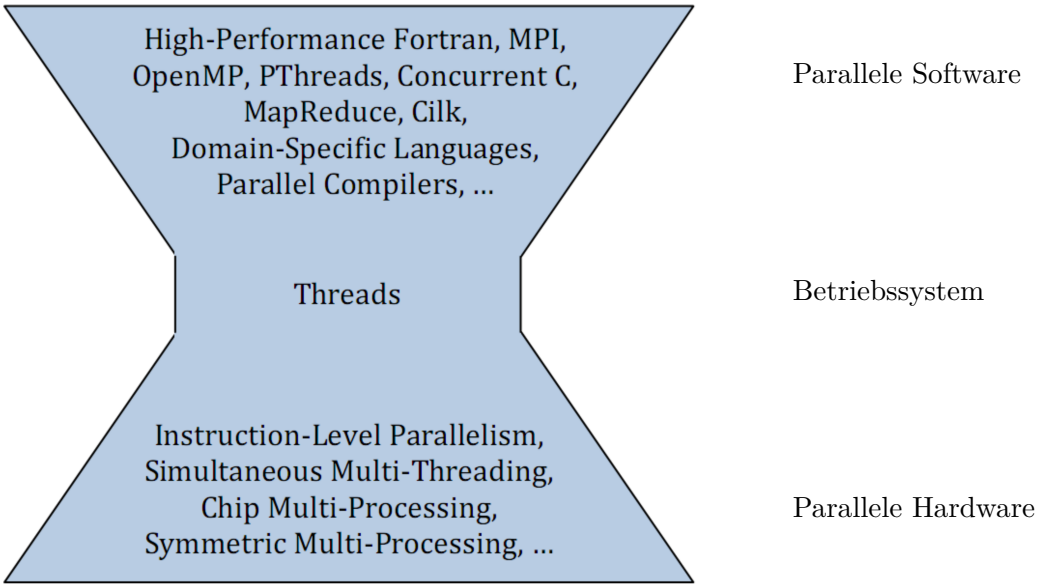


Abbildung 2.1: Sanduhr der Parallelitäten [Trö08, S.7]

Abbildung 2.1 zeigt eine grobe Strukturelle Anordnung, der unterschiedlichen paralleler Entitäten.

2.1.1 Hardware Parallelisierung

Moderne Rechner verfügen über unterschiedliche Arten von Parallelitäten. Unter paralleler Hardware versteht man die Komponenten, die zumindest zwei Threads (siehe Abschnitt 2.1.2) Parallelität zur Verfügung stellen.

Die erste und noch immer geläufige Kategorisierung paralleler Prozessoren von M. Flynn [Fly72] ordnet „Parallelrechner nach der Organisation der globalen Kontrolle und den resultierenden Daten- und Kontrollflüssen“ [RR10, S.18] ein:

- Single Instruction Single Data (SISD)
- Single Instruction Multiple Data (SIMD)
- Multiple Instruction Single Data (MISD)
- Multiple Instruction Multiple Data (MIMD)
- Hardware Multi-Threading

Diese Kategorisierungen werden im Folgenden definiert und erklärt.

Single Instruction bezieht sich auf die Existenz eines Programmspeichers, „auf den eine für die Steuerung des Kontrollflusses zuständige Kontrolleinheit zugreift“ [RR10, S.19].

Einer oder mehrere Prozessoren führen in einem Single Instruction Rechner die gleiche Instruktion aus dem Programmspeicher aus.
Single Instruction Single Data (SISD) sowie Single Instruction Multiple Data (SIMD) fallen in diese Kategorie und sind für diese Arbeit nicht relevant, da hier mehrere Prozessoren unterschiedliche Befehle ausführen können sollen. Single Data sind ebenfalls für diese Arbeit nicht relevant, da alle Prozessoren auf die Datenstruktur zugreifen können sollen.

Interessant hingegen sind die Multiple Instruction Multiple Data (MIMD) Rechner, welche folgend erläutert werde.

Multiple Instruction Multiple Data (MIMD)

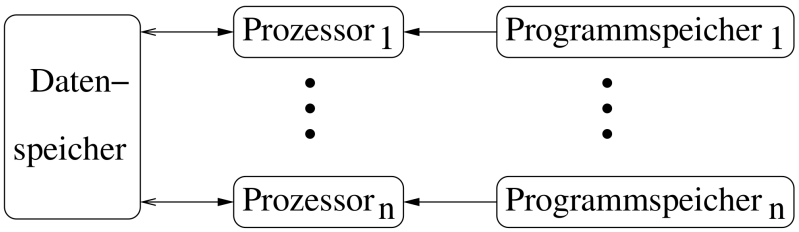


Abbildung 2.2: MIMD Modellrechner [RR10, S.18]

Der MIMD Rechner „besteht aus mehreren Verarbeitungseinheiten, von denen jede einen separaten Zugriff auf einen (gemeinsamen oder verteilten) Datenspeicher und auf einen lokalen Programmspeicher hat. Ein Verarbeitungsschritt besteht darin, dass jeder Prozessor eine separate Instruktion aus seinem lokalen Programmspeicher und Daten aus dem Datenspeicher lädt, die Instruktion auf die Daten anwendet und ein eventuell errechnetes Ergebnis in den Datenspeicher zurückschreibt. Dabei können die Prozessoren parallel zueinander arbeiten“ [RR10, S.19].

Eine Klassifizierung der MIMD Computer kann entsprechend ihrer Speicherorganisation durchgeführt werden. Zwei Aspekte werden dazu unterschieden: die physische Speicherorganisation und der Blick des Programmierers auf den Speicher. Für die physische Organisation können Computer mit einem physikalisch geteilten Speicher[2] und Computer mit einem physikalisch *ver*teilten Speicher[3] unterschieden werden.

Aus der Sicht des Programmierers kann zwischen Rechnern mit verteiltem Adressraum und jenen mit einem gemeinsam genutzten Adressraum unterschieden werden. Diese Ansicht muss nicht notwendigerweise der physikalischen Repräsentation entsprechen [TP14, S.12].

[2]Englisch: Shared Memory
[3]Englisch: Distributed Memory

Hardware Multi-Threading

Unter Multi-Threading versteht man das parallele Ausführen mehrerer Threads. Hardware unterstützt Multi-Threading durch eine Kombination aus dem Multicore Konzept und simultaner Multi-Threading Unterstützung [SA05, S.248].

Simultanes Multi-Threading [4] ist ein Verfahren, „bei dem mehrere Threads ohne explizites Umschalten ausgeführt werden“ [RR10, S.99]. Simultanes Multi-Threading wurde unter anderem von Intel® unter dem Namen Hyper-Threading® integriert.

Relevanz für die vorliegende Arbeit

Klassische Multicore-CPUs sind nach dem MIMD Modell aufgebaut.
Multicore Systeme sind vollwertige parallele Computer [Trö08, S.5] und die vorliegende Baumstruktur von Chatterjee et al. wird auf Multicore-Rechnersystemen der MIMD Kategorie untersucht.

2.1.2 Betriebssystem Parallelisierung

Wie Abbildung 2.1 illustriert ist, liegt das Betriebssystem die Parallelisierung betreffend zwischen Hardware und Software.
Alle modernen Betriebssysteme stellen unterbrechbare und planbare parallele Abarbeitungsmöglichkeiten zur Verfügung. Diese werden für gewöhnlich *Threads* genannt [Trö08, S.6].

Threads

Abgesehen von der Hardware Unterstützung für Multi-Threading (siehe Abschnitt 2.1.1) hat die Software Abstraktion eine größere Bedeutung für diese Arbeit. Folgend wird auf Software-Threads eingegangen.

Herlihy et al. definieren Threads wie folgend: „A shared-memory computation consists of multiple threads, each of which is a sequential program in its own right. These threads communicate by calling methods of objects that reside in a shared memory. Threads are asynchronous, meaning that they run at different speeds, and any thread can halt for an unpredictable duration at any time“ [HS08, S.71].

Ein Thread ist also ein Sub-Prozess des eigentlich ausgeführten Prozesses, durch den das Betriebssystem paralleles Abarbeiten von Aufgaben ermöglicht. Threads sind asynchron bezugnehmend auf ihre unterschiedlichen Ausführungsgeschwindigkeiten und darauf, dass alle Threads jederzeit für unbestimmte Zeit angehalten werden können. Diese Ansicht reflektiert die tatsächlichen Gegebenheiten von modernen Multiprozessor-Architekturen, wo Thread-Verzögerungen unvorhersehbar sind und im Bereich von Mikrosekunden bis in den Bereich von Sekunden liegen können. [HS08, S.71].

[4] Englisch: Simultaneous MultiThreading (SMT)

Threads befreien die Anwendungsebene jedoch nicht von der eigentlichen Arbeit, parallele Aktivitäten zu starten und zu verwalten. Sie erlauben der Anwendungsebene aber einen abstrakte Zugriff auf parallele Aktivitäten, unabhängig von der zugrunde liegenden Hardware [Trö08, S.6].

Werden mehrere Threads eingesetzt, bezeichnet man das als Multithreading. Multithreading ist ein aktuell wichtiges Parallelisierungs-Paradigma und wird von verschiedensten standardisierten Programmierbibliotheken unterstützt [Trö08, S.6]. Des Weiteren wird dieses Paradigma auch bei der Implementierung der vorliegenden Arbeit angewendet.

2.1.3 Software Parallelisierung

Software-seitig übernehmen mehrere unterschiedliche Teile die Parallelisierung. Wie in Abbildung 2.1 ersichtlich ist, wird neben paralleler Hardware-Unterstützung das Betriebssystem sowie die Unterstützung von der Programmiersprache selbst benötigt. Letztendlich muss der Programmierer zusätzlich auf der Applikationsebene parallele Aktivitäten starten und verwalten.

2.2 Parallele Applikationen

Um Aufgaben zu parallelisieren, müssen die Teile einer Applikation gefunden werden, die parallel ausführbar sind. Dieses Kapitel beschreibt eine Vorgehensweise, um Parallelisierung zu realisieren. Es kategorisiert Ebenen von Parallelitäten und geht speziell auf die Daten- und Thread-Parallelität ein, die für die vorliegende Arbeit relevant sind. Abschließend beschreibt es den Ursprung der Komplexität paralleler Datenzugriffe und deren Gefahren.

2.2.1 Metriken paralleler Performance

Die parallele Laufzeit $T_p(n)$ eines Programms ist die Zeit zwischen dem Start der Abarbeitung und Beendigung aller parallel durchgeführten Prozesse. Die Laufzeit wird meist in Abhängigkeit von p, der Anzahl der parallel ausführenden Threads, und einer Problemgröße n angegeben [RR10, S.167]. Die Problemgröße entspricht der Anzahl der Eingangswerte.

2.2.2 Performance

Die Performance ist „ein wesentliches Kriterium für die Güte einer parallelen Implementierung eines Algorithmus [...] auf einem gegebenen Rechner“ [RR10, S.167]. Diese Arbeit zieht zum Vergleich zwischen parallelen Applikationen die Laufzeit als Performanceindikator heran. Folgend werden Metriken zum Vergleich der Laufzeit, sowie der maximale Geschwindigkeitsgewinn gegenüber einer sequentiellen Applikation vorgestellt.

Es sei erwähnt, dass zur Aufrechterhaltung der maximalen Performance auch an die Bereitstellung der Eingangsdaten zu denken ist. Bildet sich ein Flaschenhals bei der

Bereitstellung, leidet die maximale Performance ebenfalls darunter. Diese Untersuchung sowie die zeitgerechte Aufbereitung der Eingangsdaten sind nicht teil dieser Arbeit.

Performance Gewinn Der Performance Gewinn $S_p(n)$ eines parallelen Programmes mit der Laufzeit $T_p(n)$ lässt sich definieren als

$$S_p(n) = \frac{T^*(n)}{T_p(n)}$$

wobei p die Anzahl der parallel ausführenden Threads „zur Lösung des Problems der Größe n bezeichnet. $T^*(n)$ ist die Laufzeit einer optimalen sequentiellen Implementierung zur Lösung desselben Problems“ [RR10, S.168].

2.2.3 Inhärente Komplexität und Risiken

Folgend soll auf die für diese Arbeit bedeutendsten Faktoren, die die Komplexität der Parallelisierung bestimmen, eingegangen werden.

Speicher

Speicherverwaltung richtig zu implementieren ist eine schwierige Aufgabe [Gra15, S.3]: Es müssen spezielle Vorkehrungen getroffen werden, um Speicher - nach der Verwendung in einer parallelen Datenstruktur - wieder frei zu geben. Die Realisierung der Speicherfreigabe ist nicht immer unmittelbar umsetzbar.

Race Condition

Bei der Verwendung von gemeinsam genutztem Speicher greifen mehrere Threads auf denselben Speicher zu. Dabei müssen Lese- wie Schreibzugriffe zum selben Zeitpunkt verhindert werden, weil dies zu *Race Conditions* führen kann. Der Begriff *Race Condition* beschreibt den Effekt von parallel ausgeführten Programmteilen durch multiple ausführende Einheiten auf das Resultat. Das Resultat ist abhängig von der Reihenfolge, in welcher die Programmteile durch die unterschiedlichen ausführenden Einheiten bearbeitet werden. Im Falle einer Race Condition kann es vorkommen, dass die Berechnung eines Programmteils zu unterschiedlichen Ergebnissen führt - je nachdem, ob Thread T_1 den Programmteil vor dem T_2 oder umgekehrt ausführt. In der Regel sind Race Conditions unerwünscht, da die relative Ausführungsgeschwindigkeit der Threads von vielen Faktoren, die vom Programmierer nicht beeinflusst werden können, abhängt. Dies kann zu nicht-deterministischem[5] Verhalten führen, da abhängig von der Ausführungsreihenfolge verschiedene Ergebnisse möglich sind und das genaue Ergebnis nicht vorhergesagt werden kann [Ray13, S.118].

[5] Englisch: Non-deterministic

Kritische Sektionen

In parallel arbeitenden Systemen ist eine kritische Sektion ein Programmabschnitt, der nicht parallel ausgeführt werden darf, da es sonst zu *Race conditions* und anderen unerwünschten Phänomenen kommen kann. Bei jeder Parallelisierung besteht die Gefahr, dass kritische Sektionen von Prozessen überlappen, was zu drei Hauptgruppen negativer Zustände führt:

Indeterminismus Mehrere datenverarbeitende sequenzielle Programme erbringen bei gleichen Eingaben unterschiedliche Ergebnisse.

Deadlock Die verarbeitenden Prozesse warten auf Ressourcen des jeweilig anderen und blockieren sich somit unwiderruflich gegenseitig.

Starvation Den Zustand, wenn Prozessen fortwährend die nötigen Ressourcen zur weiteren Abarbeitung ihrer Tätigkeit entzogen werden, nennt man Starvation.

Deadlock

Nicht-deterministisches Verhalten und Race Conditions können durch Synchronisations-Mechanismen wie der Lock-Synchronisation vermieden werden. Jedoch kann (auch, aber nicht nur) die Verwendung von Locks zu Deadlocks führen. Deadlock bezeichnet die Situation, wenn die Programmausführung in einen Zustand gelangt, in dem jeder Thread auf ein Ereignis wartet, das ausschließlich von den jeweils anderen Threads entspringen kann.
Im Allgemeinen tritt ein Deadlock für eine Reihe von Aktivitäten auf, wenn jede der Aktivitäten auf Ereignisse wartet, die nur von einem der anderen Aktivitäten verursacht werden können. So tritt ein Zyklus des gegenseitigen Wartens auf [Ray13, S.140].

Thread T_1	Thread T_2
$lock(l1)$	$lock(l2)$
$lock(l2)$	$lock(l1)$
$do\_work()$	$do\_work()$
$unlock(l2)$	$unlock(l1)$
$unlock(l1)$	$unlock(l2)$

Tabelle 2.1: Deadlock-behaftete Lock-Synchronisation [Ray13, S.140]

Tabelle 2.1 illustriert eine Abfolge der Ausführung zweier Threads T_1 und T_2, bei denen im Zuge der Verwendung der Locks $l1$ und $l2$ ein Deadlock auftreten kann. Dieser entsteht bei der folgenden Ausführungsreihenfolge:

- Thread T_1 versucht zuerst Lock $l1$, danach Lock $l2$ zu gewinnen. T_1 wird jedoch nach erfolgreichem Antrag auf $l1$ unterbrochen.

- Thread T_2 versucht zuerst Lock $l2$, danach Lock $l1$ zu gewinnen. T_2 wartet jedoch nach erfolgreichem Antrag auf $l2$ auf die Freigabe von $l1$.

In dieser Situation ist $l1$ gesperrt durch T_1 und $l2$ gesperrt durch T_2. Beide Threads warten auf die Freigabe des fehlenden Locks, das vom jeweilig anderen Thread gehalten wird. Jedoch tritt diese Freigabe nicht auf, weil der jeweilig andere Thread ebenfalls wartet [Ray13, S.140].

2.2.4 Lock-Free und Wait-Free

Lock-Free[6] und Wait-Free[7] sind Fortschrittszusicherungen, die ein System und deren Threads machen müssen. Garantiert ein Programm diese Zusicherungen, sind keine weiteren Vorkehrungen notwendig, um diese Zusicherungen zu erhalten [HS08, S.99].
Die beiden Eigenschaften *garantieren Fortschritt* einer Multi-Thread Umgebung in unterschiedlichen Ausprägungen:

Lock-Free wird eine strikte Fortschrittszusicherung bezeichnet, die besagt, dass zu jedem Zeitpunkt mindestens ein Thread Fortschritte verzeichnet. Diese Zusicherung garantiert, dass ein System als Ganzes jederzeit Fortschritte macht [BER14, S.329] und folgend daraus, dass Zugriffe auf eine Ressource unendlich lange erfolgreich sind [HS08][S.24, S.99]. Das bedeutet jedenfalls, dass ein solches System Deadlock frei ist, schließt *Starvation* jedoch nicht aus. *Starving* einzelner Threads ist dadurch nicht ausgeschlossen, auch wenn das System als Ganzes jederzeit Fortschritt zeigt.

Wait-Free wird eine Fortschrittszusicherung bezeichnet, bei der *alle Threads* zu jedem Zeitpunkt *Fortschritt verbuchen*. Eine Ressource ist Wait-Free, wenn jeder Zugriff auf diese Ressource nach einer *endlichen* Anzahl an Schritten erfolgreich ist [HS08, S.99]. Dies garantiert nicht nur, dass das System Deadlock frei ist, sondern zusätzlich, dass keine Starvation auftreten kann.

Die stärkere dieser beiden Zusicherungen stellt also Wait-Free dar [TP14, S.357]. Ein Wait-Free Algorithmus ist schwerer zu entwerfen und geht nahezu immer mit Performance-Einbußen (wegen steigendem Overhead) [TP14, S.357] einher.

Der in dieser Arbeit untersuchte Algorithmus sichert Lock-Freiheit, jedoch nicht Wait-Freiheit, zu.

2.2.5 Linearisierbarkeit

Ein Möglichkeit, die Korrektheit von parallelen Applikationen zu zeigen, stellt die Linearisierung dar. „Linearizability is a correctness condition for concurrent objects [...]. [It] provides the illusion that each operation applied by concurrent processes takes

[6] Anglizismus für Sperren-Freiheit oder Lock-Freiheit
[7] Anglizismus für Warte-Freiheit

effect instantaneously at some point between its invocation and its response“ [HW90, S.463]. Dieser Zeitpunkt heißt Linearisierungspunkt. Eine Abfolge valider, paralleler, linearisierbarer Operationen rufen denselben Effekt hervor wie eine valide sequenzielle Abfolge dieser Operationen.
So sind also parallele Sequenzen von Operationen formal rückführbar auf eine äquivalente sequenzielle Operationsabfolge.

Dieses Verfahren wird in der untersuchten Arbeit zur Beweisführung der Validität verwendet, was im Unterabschnitt 4.1.3 weiter untersucht wird.

2.2.6 C++11 Erweiterung für Multi-Threading

In dieser Arbeit kommt die Erweiterung der sequentiellen Programmiersprache C++ in der Version C++11 zum Einsatz. Die für diese Arbeit wichtigen Erweiterungen von C++11 wurden in den Bereichen der Threads [Kor13, S.198] und einer klar definierten Memory Reihenfolge durch `std::memory_order` implementiert.

So kann die Reihenfolge der nicht atomischen Zugriffe vor und nach einer atomischen Operation verändert werden (siehe [cpp16b]).

2.3 Parallele Datenstrukturen

Datenstrukturen stellen eine Schnittstelle für den Zugriff auf Daten dar, unabhängig davon, ob das Lese- oder Schreibzugriffe sind.

Datenstrukturen beschreiben die Art der digitalen Abbildung von Daten, womit sie eine Basis für Forschung (beziehungsweise Diskussionen) hinsichtlich ihrer Zusicherungen und Limits bilden. Datenstrukturen abstrahieren die Verwendung von Daten und deren inhärente unterliegende Art das zu tun. Das macht sie für Datenzugriffe aller Art zu einem wichtigem Gebiet in der modernen Informationstechnologie.

Datenstrukturen wie *Listen, Stacks, Queues* oder *Bäume* bilden „die Bausteine für die Implementierung komplexer Algorithmen“ [Güt92, S.49]. Sie unterscheiden sich durch ihre „Modelle (Algebren)“ [Güt92, S.49], ihre Implementierungsoptionen („einfach und doppelt verkettete Listen, Einbettung in ein Array“ [Güt92, S.49]) und durch ihren Aufbau.

2.3.1 Parallele Datenzugriffe

Unter paralleler oder nebenläufiger[8] Datenverarbeitung versteht man das gleichzeitige Ausführen mehrere sequenzieller Berechnungen (zum Beispiel Zugreifen, Editieren oder Löschen) auf ein und die selben Daten.

Nebenläufige Datenstrukturen sind so entworfen, dass sie alle zur Verfügung stehenden Ressourcen gleichzeitig nutzen, um dadurch eine bessere Performance zu erreichen [TP14,

[8] Englisch: Concurrency

S.357].
Das ist in vielen Programmen sinnvoll, zum Beispiel wenn dieselbe Operation auf unterschiedliche Elemente einer große Datenstruktur angewendet werden soll. Wenn die anzuwendenden Operationen unabhängig voneinander sind, können sie zur Parallelisierung vorgesehen werden. Datenparallelität nennt den Fall, wenn sich die Elemente einer Datenstruktur gleichmäßig über den Prozessor verteilt befinden und jeder Prozessor in der Lage ist, seine Operationen auf die ihm zugewiesenen Elemente anzuwenden [TP14, S.100].

2.3.2 Bedarf an Lock-Free Datenstrukturen

Mit der stark wachsenden Verfügbarkeit von Multi-Core Prozessoren wird es zwingend, effizientere parallele Datenstrukturen zu designen und zu entwickeln. Was Lock-Free Datenstrukturen attraktiver macht als ihre blockierend ausgeführte Gegenstücke, ist, dass sie Deadlocks über die Kontrolle des Datenstruktur Designers hinausgehend verhindert [CNT14, S.322]. Weiters bringen Datenstrukturen die Lock-Freiheit zusichern weniger Overhead mit sich als jene, die noch striktere Zusagen wie Wait-Free treffen, was sie zu einem guten Kompromiss macht.

2.3.3 Suchbäume

Die Repräsentation von zu speichernden Keys in einer Baumstruktur passiert durch Knoten, welche durch Kanten zu einem Suchbaum verbunden werden.

Bäume speichern Keys aus einem Universum ab und um als Suchbaum funktionieren zu können müssen a) Keys von Knoten größer sein als ihre Nachfolgeknoten und b) alle nachfolgenden Keys in linken Unterbäumen müssen kleiner und alle nachfolgenden Keys in rechten Unterbäumen müssen größer sein.

Zu den drei wichtigsten Operationen, die auf einen Suchbaum angewendet werden können, zählen das Einfügen, Auffinden und Entfernen von Keys.

Digitale Abbildung von Bäumen

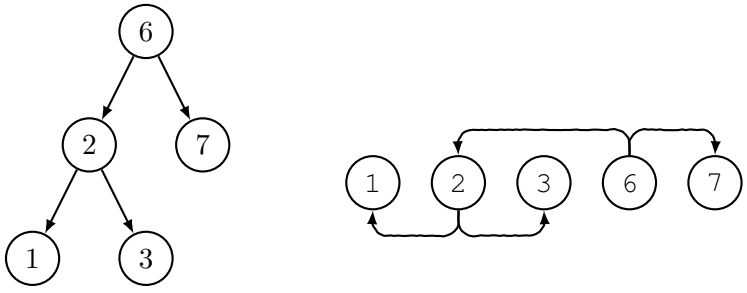


Abbildung 2.3: (a) BST (b) Repräsentation als geordnete Liste

Zur Abbildung einer Kante auf einen Folgeknoten verwendet man Zeiger im Speicher [BER14, S.332] wie dies in Abbildung 2.3(a) zu sehen ist. Solch ein Aufbau eines Baumes, in Form einer im Speicher abgebildeten Liste ist in Abbildung 2.3(b) zu sehen. Im Speicher bietet sich hierfür, je nach Implementierung, eine Referenz beziehungsweise ein Pointer an.
Für den Bezug auf die Baumstruktur von Chatterjee et al. sei auf den Abschnitt 2.3.3 Balancierung und Baumtiefe verwiesen.

Bedarf an Baumstrukturen

Bäume finden in der praktischen Programmierung viele Anwendungsfälle [KS07, S.153]. Sie bilden oft „die Grundlage für schnelle Suchverfahren in Datenbanken und einige Datenkompressionsverfahren. Viele dieser Anwendungen bauen auf [...] den binären Bäumen“ [KS07, S.153] auf.

Binäre Suchbäume

Ein Spezialfall eines gewöhnlichen Suchbaums ist der *binäre* Baum beziehungsweise Binary Search Tree (BST). Er ist ein Baum, bei dem jeder Knoten höchstens zwei Nachfolger beziehungsweise Kinder besitzt. Diese werden im Fall binärer Bäume als linkes beziehungsweise rechtes Kind[9] bezeichnet [SG00, S.118][Güt92, S.76,83].

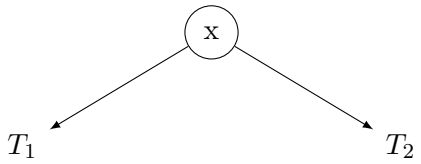


Abbildung 2.4: Definition binärer Bäume

Für BST lassen sich folgende Eigenschaften definieren:
a) Ein leerer Baum ist ein binärer Baum und
b) Wenn x ein Knoten ist und T_1,T_2 binäre Bäume sind, dann ist auch das Tripel (T_1, x, T_2) ein binärer Baum T; dies wird von Abbildung 2.4 illustriert [Güt92, S.77].

Abbildung 2.5 zeigt ein Beispiel eines BSTs.

Mit Ausnahme des Wurzelknotens besitzt jeder Knoten genau eine eingehende Kante [SG00, S.118]. Diese Eigenschaft ist für das Durchsuchen eines binären Baumes besonders wichtig, weil diese Eigenschaft dem Durchlaufen des Baums seine Geschwindigkeit verleiht.

Die hier untersuchte Datenstruktur von Chatterjee et al. stellt einen BST dar, auf dessen Besonderheiten im Kapitel 4~Untersuchung des Lock-freien BST Algorithmus weiter eingegangen wird.

[9] Englisch: Left and Right child

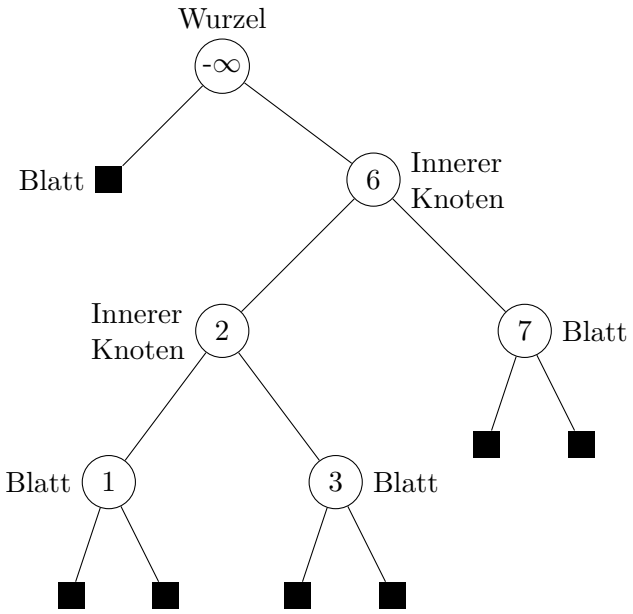


Abbildung 2.5: Binärbaum mit Blättern

Weitere Spezialfälle von Suchbäumen

Es sei auf die Existenz der folgenden *Spezialfälle* von Suchbäumen hingewiesen, namentlich:

B-Bäume Sind selbst-balancierte Baumstrukturen und bilden eine generalisierte Form der BSTs, bei denen jeder Knoten mehr als zwei Kinder besitzen kann.

2-3-4 Bäume sind selbst-balancierte Bäume, mit zwei bis vier Keys je Knoten.

B+ Bäume haben, im Gegensatz zu BST, eine hohe Anzahl an Links zu ihren Kindern pro Knoten [Com79, S.129], was den Suchaufwand reduziert.

Rot-Schwarz Bäume „Ein Rot-Schwarz-Baum ist ein binärer Suchbaum, in dem jede Kante rot oder schwarz gefärbt ist“ [DMS14, S.196]. Die Färbung wird verwendet, um den Baum balanciert zu halten.

Vorteile binärer Bäume gegenüber anderen Datenstrukturen

Folgend werden sortierte Reihen und verkettete Listen den BSTs gegenübergestellt. Die großen implementationsunabhängigen Unterschiede ruhen in der Zeitkomplexität der verschiedenen Datenstrukturen, welche sich wie folgend ergeben:

Sortierte Reihe Suchvorgänge in sortierten Reihen besitzen im Worst-Case eine Zeitkomplexität von $O(\log\ n)$. Sortierte Reihen besitzen jedoch den Nachteil der

ungünstigen Zeitkomplexität von $O(n)$ für *insert* Operationen und den Bedarf an a priori Wissen über die maximale Größe der Datenstruktur.

Verkettete Liste In einer verketteten Liste muss Speicher erst dann reserviert werden, wenn er auch tatsächlich gebraucht wird. Der *insert* Vorgang, exklusive dem Suchvorgang, kann in einem Schritt erfolgen, hat jedoch den Nachteil der unerwünscht großen Zeitkomplexität von $O(n)$, bei jeder Suche von Elementen.
[SG00, S.87-89, S.121]

Der binäre Suchbaum kombiniert die Vorteile sowohl von verketteten Listen, als auch die von sortierten Reihen, indem er „die Objekte zu einem sortierten Binärbaum zusammenfügt“ [SG00, S.121]. Der Worst-Case von $O(n)$ verbessert sich dadurch nicht, da der Baum zu einer sortierten Reihe mit der Tiefe n entarten kann. Bei Betrachtung aller sortierten Binärbäume mit den Schlüsseln 1 bis n, liegt die mittlere Baumtiefe bei $2 * log_2(n)$ [SG14, S.114].

Balancierung und Baumtiefe

Zusätzlich zu den Balancierungseigenschaften gewöhnlicher Suchbäume, die diese Arbeit nicht näher beschreibt, seien einige wesentliche Spezifika binärer Bäume erwähnt.

Ein besonders schlecht balancierter Baum hat die Tiefe n und damit die gleiche Struktur wie eine verkettete Liste [SG00, S.121]. Diese der balancierten Hierarchie entgegengesetzte Anordnung von Knoten wird *entartete Ordnung* beziehungsweise *Entartung* genannt.

Um der Entartung Herr zu werden, können Balancierungsmaßnahmen[10] getroffen werden. Diese Maßnahmen sortieren die innere Struktur des Baumes so um, dass dieser nach dem Prozess ähnlicher einer balancierten Hierarchie ist als zuvor. Diese Vorgehensweise wird auch Re-Balanierung[11] genannt.

Selbst-balancierend Eine Baumstruktur, bei der ein Balancierungsvorgang nicht getrennt angestoßen werden muss, sondern der dies selbst überwacht, wird selbst-balancierend genannt. Als Beispiele seien die folgenden selbst-balancierenden Suchbäume genannt: AVL-Baum, Rot-Schwarz-Baum oder der 2-3-Baum.

Durchnittstiefe „Wenn man alle sortierten Binärbäume mit den Schlüsseln 1 bis n betrachtet, dann haben sie im Durchschnitt eine Tiefe von $2 * \log n$“ [SG00, S.122]. Diese Tatsache macht sie im Schnitt performanter als sortierte Reihen beziehungsweise gewöhnliche Suchbäume.

[10] Englisch: Balancing
[11] Englisch: Re-Balancing

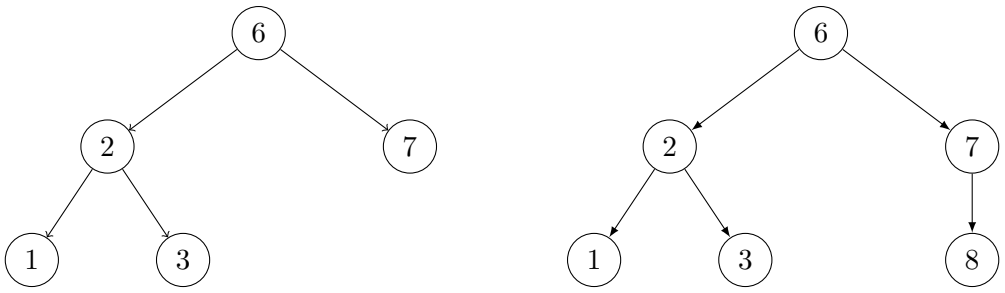
Suchen in Binärbäumen

Zusammen mit der Invariante, dass ein sortierter Baum vorliegt, kann der Binärbaum gleich schnell wie eine sortierte mathematische Reihe durchsucht werden. Das bedeutet im Worst-Case eine Zeitkomplexität von $O(\log\ n)$, wie Abschnitt 2.3.3 zeigt.

Folgend wird auf den Vorgang der Suche selbst eingegangen. Um einen Schlüssel x zu lokalisieren, wird an der Wurzel mit der Suche begonnen und man folgt dem eindeutigen Weg zu dem Blatt mit dem kleinsten Schlüssel, der mindestens so groß wie x ist. Das bedeutet, dass der Aufwand, jede einzelne Verzweigung zu durchsuchen, aufgrund des eindeutigen Pfades entfällt. Dies wird erreicht, indem in den *inneren Knoten* des Baums ebenfalls Folge-Schlüssel gespeichert werden, die die Suche bestimmen. In einem binären Suchbaum mit mindestens zwei Blättern besitzt jeder innere Knoten genau zwei Kinder, ein linkes und ein rechtes Kind. Für die Kinder des Knotens mit dem Schlüssel s gilt, dass alle Schlüssel x im linken Unterbaum $x \leq s$ erfüllen und alle Schlüssel im rechten Unterbaum $x > s$ erfüllen [DMS14, S.186].

Insert und *Remove* Vorgänge

Die gängigen Operationen von Datenstrukturen *Insert* sowie *Remove* fügen ein neues Element zur Datenstruktur hinzu beziehungsweise entfernen ein bestehendes Element aus der Datenstruktur. Sollte dies nicht möglich sein, weil für *Insert* der einzufügende Schlüssel bereits vorhanden ist, beziehungsweise für *Remove* der zu entfernende Schlüssel nicht vorhanden ist, werden keine Änderungen an der Datenstruktur vorgenommen [CGR13, S.231]. Beide Operationen bilden die wichtigsten *Update* Maßnahmen und sind zusammen mit dem Suchvorgang die wichtigsten Operationen, die auf Datenstrukturen ausgeführt werden können.



(a) Ursprungszustand (b) Nach dem Einfügen von $x = 8$

Abbildung 2.6: Einfügen eines Knotens

Um einen Schlüssel in einen Suchbaum **einzufügen**, wird zuerst dessen Existenz im Baum geprüft. Abhängig von der Existenz des einzufügenden Schlüssels wird einer der folgenden Fälle ausgeführt: a) Er existiert, dann wird erfolglos abgebrochen oder b) Er existiert nicht, dann wird er durch Manipulation der Kanten in den Baum eingefügt.

Um einen Schlüssel aus einem Suchbaum zu **entfernen**, wird ebenfalls zuerst dessen Existenz im Baum geprüft. Abhängig von der Existenz des zu entfernenden Schlüssels wird einer der folgenden Fälle ausgeführt: a) Er existiert nicht, dann wird erfolglos abgebrochen oder b) Er existiert, dann werden seine rechten und linken Teilbäume durch Manipulation der Kanten an anderen Stellen im Baum angehängt.

Auf diese Weise erreicht man, dass der resultierende Baum wieder ein Suchbaum ist, wie Abbildung 2.6 illustrativ darstellt [OW12, S.267].

Komplexität

Um der inhärenten Komplexität und den Risiken im Zusammenhang mit Datenstrukturen entgegenzuwirken, wurden Lock-Free sowie Wait-Free Datenstrukturen entwickelt.
Die Schwierigkeit einen Lock-Free Baum zu designen, liegt jedoch an der Komplexität multiple Referenzen (Links) atomisch zu modifizieren. Die untersuchte Baumstruktur von Chatterjee et al. stellt einen Lock-Free Algorithmus dar. Wie die Autoren diese Probleme handhaben, stellt das folgende Kapitel der Implementierung (Kapitel 4) vor.

KAPITEL 3

Stand der Forschung

Dieses Kapitel analysiert bestehende Ansätze und Forschungsergebnisse zu paralleler Datenverarbeitung und beschreibt deren Herangehensweise. Das Kapitel reflektiert zu großen Teilen den aktuellen Stand des Forschungsgebiets sowie Meinungen und Erkenntnisse anderer Wissenschafter.

Jede der vorgestellten Publikationen geht auf unterschiedliche Besonderheiten oder Neuigkeiten im betrachteten Forschungsgebiet ein und zeigt die Relevanz für die vorliegende Arbeit auf.

3.1 More Than You Ever Wanted to Know About Synchronization

Gramoli stellt einen umfassenden Vergleich der Performance unterschiedlicher Datenstrukturen zur Verfügung. Er vergleicht unter anderem die Performance von Locks gegenüber CAS und stellt eine Benchmark-Umgebung zum Vergleich unterschiedlicher Datenstrukturen für unterschiedliche Programmiersprachen zur Verfügung.

Weiters werden Vor- und Nachteile von CAS und anderen Synchronisationstechniken verglichen. Für die betrachtete Arbeit von Chatterjee et al., die ebenfalls auf CAS Operationen aufbaut, sind die Vergleiche des Autors in Hinblick auf deren Performance interessant [Gra15].

3.1.1 Besonderheiten

Mithilfe der von dem Autor vorgestellten (Open-Source) *Micro-Benchmark* Umgebung stellt er extensiv Vergleiche zwischen mehr als dreißig unterschiedlichen Datenstrukturen an. Diese Arbeit stellt, zum Zeitpunkt der Untersuchung, den umfangreichsten Vergleich von Synchronisationstechniken dar.

Der Autor trifft die folgenden Aussagen, die auch für das vorliegende Werk relevant sind.

Speicher Wiederverwendung Dass Speicher-Management in parallelen Datenstrukturen eine komplexe Aufgabe darstellt, wurde bereits im vorigen Kapitel (Abschnitt 2.2.3) näher gebracht. Zusätzlich dazu hält der Autor fest, dass die Krux des Problems in der Speicherwiederverwendung[1] liegt. Threads müssen nämlich die Möglichkeit haben, auf Speicherbereiche zuzugreifen, die zukünftig nicht mehr genutzt werden können [Gra15, S.3] und damit ab einem Zeitpunkt nicht mehr valide sind.

CAS Performance CAS basierte Implementierungen sind typischerweise schneller als alle anderen Synchronisationstechniken. Im Speziellen sind CAS Implementierungen in Konfliktsituationen zwischen mehreren zugreifenden Parteien und bei Zugriffen ohne effektive Manipulation der Daten schneller als alle anderen verglichenen Konzepte [Gra15, S.5].

Skip Liste vs. Suchbäume Skip Listen bestehen aus Türmen, die auf ihre Nachfolger zeigen und sie beinhalten Knoten, von denen jeder auf den ihm direkt untergeordneten Knoten zeigt. Sie werden oft mit Suchbäumen verglichen, weil jeder Knoten einen Nachfolger in seinem Nachfolger-Turm besitzt. Ein bedeutender Unterschied liegt darin, dass die Abwärts-Zeiger in den Skip Listen 'Säulen' (oder Nachfolger Turm) unverändert bleiben und somit einer einfacheren atomischen Modifikation zugrunde liegen. Dies ist laut dem Autor wahrscheinlich der Grund dafür, dass Skip Listen unter hoher Konfliktlast eine höhere Leistung erbringen als Suchbäume [Gra15, S.7].

Einen Vergleich zwischen Linked Listen und den anderen von den Autoren untersuchten Datenstrukturen ergibt sich nicht, da Linked Listen laut ihren empirischen Messungen um einen Faktor von ungefähre 500 schlechter performen.

3.1.2 Relevanz für die vorliegende Arbeit

Diese Arbeit dient als Bestätigung der Verwendung von Read-Modify-Write Techniken wie CAS, gegenüber den Techniken der Locks, transaktioneller Speicher-Synchronisierung oder Read-Copy-Update als Synchronisationstechniken. Weiters finden sich in dieser Arbeit interessante Erweiterungsmöglichkeiten der Speicherwiederverwendung für den vorliegenden BST. Leider wurde die betrachtete Arbeit nicht von den Autoren implementiert, was für zukünftige Vergleiche mit der zur Verfügung stehenden Benchmark-Umgebung interessant ist.

3.1.3 Ergebnisse

Der Autor zeigt mit Analysen die Flaschenhälse unterschiedlicher Synchronisationstechniken auf. Er greift dieses Thema auf, da er einen Mangel an Vergleichbarkeit zwischen

[1] Englisch: Memory Reclamation

Datenstrukturen festgestellt hat. Er hebt die Bedeutung von CAS Operationen hervor, aber auch die Nachteile wie „the lock-free use of compare-and-swap makes the design of [...] data structures, and especially the ones with non-trivial mutations, extremely difficult“ [Gra15, S.1].

Es werden Schnittstellen sowohl für Java, als auch für C++ zur Verfügung gestellt, um weitere Implementierungen mit der vorgestellten Benchmark-Umgebung vergleichen zu können. Ein beträchtlicher Aufwand wurde in die Implementierung von mehr als dreißig Datenstrukturen gesteckt, die mit der vorgestellten Benchmark-Umgebung kompatibel gemacht wurden.

Jede/m die/der an vergleichbaren Lock-freien Datenstrukturen arbeitet, sei dieses Werk nahegelegt.

3.2 A General Technique for Non-Blocking Trees

Das Werk von Trevor Brown, Faith Ellen und Eric Ruppert beschreibt eine generelle Technik, um beweisbare Lock-Free Datenstrukturen zu erstellen. Die Autoren spezialisieren sich auf Datenstrukturen, die vom Elternknoten zum Kindknoten gerichtet sind. Die Autoren beschreiben eine generelle Strategie, um eine Lock-Free Implementierung zu erlangen [BER14, S.329].

Um diese Technik zu illustrieren, stellen sie eine abgewandelte Variante eines Red Black Trees (Chromatische Bäume wurden von Nurmi und Soisalon-Soininen [NSS96] als ein neuer Typ von BSTs für Datenbanken definiert) vor, einen so genannten chromatischen Baum. Bei diesem wurden rot und schwarz des Red Black Tree durch nicht-negative Integer-Gewichte ersetzt [BER14, S.335].

3.2.1 Besonderheiten

Der von den Autoren beschriebene chromatische Baum ist träge balanciert. Er besitzt eine garantierte Tiefe von $O(c+\log n)$ [EFHR14, S.340], wobei n die Anzahl der Schlüssel[2] und c die Anzahl der in Bearbeitung befindlichen Updates ist. Laut den Autoren stellt dies den ersten beweisbar korrekten Lock-Free BST dar [BER14, S.330].

Weiters stellen die Autoren ein Template vor, dass die Integration von Update-Prozessen (wie *Insert*, *Remove* oder das *Re-Balancierungen*) vereinfacht. Es kann durch Einsetzen in das Template automatisch ein linearisierbarer und Lock-freier Baum erstellt werden, dessen Inhalt durch jeweils exakt einen atomischen Zeigertausch verändert werden kann. Eine solche Veränderung illustriert die Abbildung 3.1 [BER14, S.332].

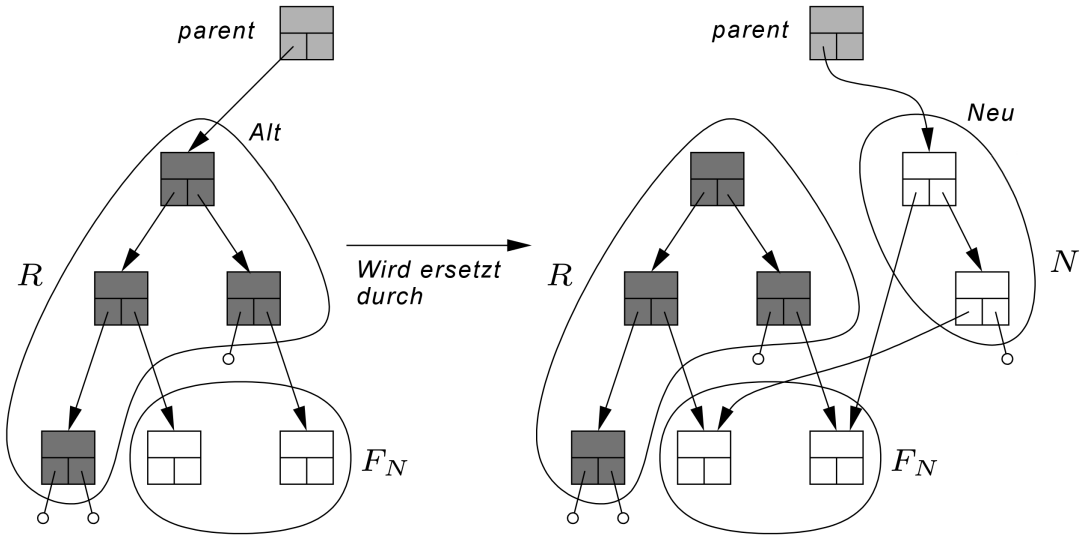
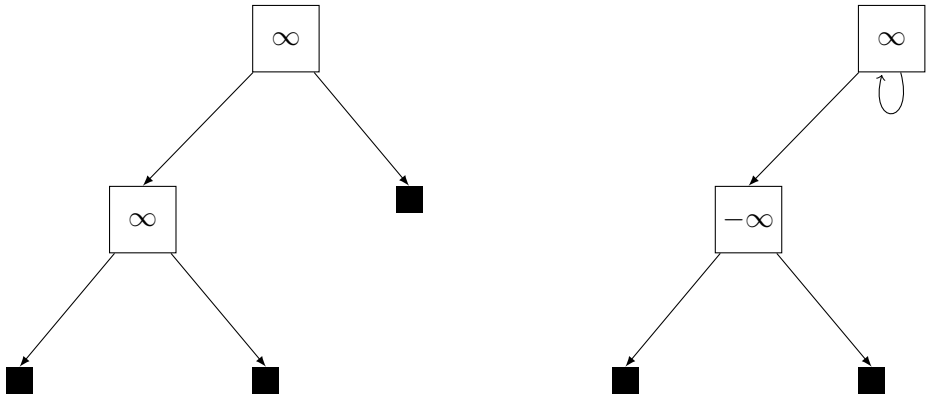


Abbildung 3.1: Atomisches Updated auf der Baumstruktur von Brown, Ellen und Ruppert



(a) Brown et al. (b) Chatterjee et al.

Abbildung 3.2: Unterschiedliche Implementierungen der Wächter Knoten

3.2.2 Relevanz für die vorliegende Arbeit

Die Autoren verwenden das Konzept der „Wächter-Knoten“[3] [BER14, S.335]. Wächter-Knoten sind Knoten an oberster hierarchischer Stelle von Datenstrukturen, die jederzeit garantiert verfügbar sind. Dadurch lassen sich Zusagen zum Zeitpunkt der Implementierung treffen, die diesen Vorgang zumindest vereinfachen.
Die verwendeten Wächter-Knoten ähneln stark der untersuchten Datenstruktur von Chatterjee et al.. Beide verwenden zumindest ein paar Wächter-Knoten an der Wurzel

[2]Englisch: Keys
[3]Englisch: Centinel Nodes

des Baumes, wobei sich bei Chatterjee et al. der zweiten Knoten durch ein negatives Vorzeichen unterscheidet (siehe Abbildung 3.2).

3.2.3 Ergebnisse

Die Autoren haben die Performance einer ihrer Implementationen in einer vielseitigen Analyse gegen eine Reihe von Datenstrukturen anderer Forscher und Entwickler getestet. In ihren Performance-Tests verhält sich die Lock-Free Datenstruktur der Autoren in einigen Fällen gleich gut, in anderen sogar besser als die Vergleichsgruppe.
Weiterführend könnte daran geforscht werden, ob die Analyse auf komplexere Datenstrukturen erweiterbar ist, um $O(c + \log n)$ auch für Lock-Free Sets zu beweisen [EFHR14, S.340].

Sie greifen das Thema auf, um den Weg zu neuen Lock-free Datenstrukturen zu erleichtern, indem sie eine Basis dafür vorgeben. Jede/r, die/der im Begriff ist, solche eine neue Datenstruktur zu erstellen, sei dieses Werk nahegelegt.

3.3 A Practical Wait-free Simulation for Lock-free Data Structures

Timnat und Petrank präsentieren in ihrer Arbeit eine generische Transformationsmethode von Lock-Free Datenstrukturen zu einer, die Wait-Free ist [TP14].

Die ursprünglichen Lock-Free Datenzugriffe werden dabei in einer leicht veränderten Variante so lange eingesetzt, bis ein definierbares Konfliktniveau[4] überschritten wird. Danach wird ein langsamerer Wait-Free Pfad beschritten, der andere Threads dazu auffordert, die fehlschlagende Operation gemeinsam zu vollenden. Diese Methode bietet im Lock-Free Pfad verglichen mit der ursprünglichen Implementierung nahezu gleiche Performance und bietet im Wait-Free Pfad eine Obergrenze der Ausführungszeit.
Hierfür definieren sie eine *normalisierte Repräsentation* von Algorithmen und eine Möglichkeit jeden normalisierten Algorithmus in einen Wait-Free Algorithmus zu transformieren. Laut Aussagen der Autoren lässt sich jede linearisierbare Lock-Free Datenstruktur in die *normalisierte Repräsentation* bringen.

3.3.1 Besonderheiten

Der Schritt-für-Schritt Ansatz zur Überführung eines Lock-Free in einen Wait-Free Algorithmus ähnelt einer Anleitung, der auch ein Beispiel beigelegt ist. Das Ziel der Autoren ist es, Personen die Anwendung ihres Verfahrens zu ermöglichen, die keine Experten auf diesem Gebiet sind [TP14, S.357].
Die Anwendung des Verfahren setzt jedoch detaillierte Kenntnisse über den zu bearbeitenden Algorithmus voraus und verlangt ein hohes Maß an praktischer Umsetzungsfähigkeit ab.

[4] Threshold

Mechanismen zur Detektion des fehlenden Fortschritts werden vorgestellt. Darauf aufbauend wird von der fehlschlagende Operation Hilfe aller anderen Threads angefordert. Dies geschieht über eine Wait-Free Datenstruktur. Die Autoren verwenden dazu die Wait-Free Queue von Kogan und Petrank [KP11]. Die einzige Änderung, die dazu im schnellen Lock-Free Verfahren eingepflegt werden muss, ist ein Check, ob Hilfe angefordert wurde. Sollte dies der Fall sein, leistet der jeweilige Thread Unterstützung bei der Ausführung der fehlgeschlagenen Operation [TP14, S.359]. Dabei gilt es Details im Zuge der Synchronisation der helfenden Threads zu beachten, die nicht trivial erscheinen.

Die normalisierte Repräsentation Sie ist erreicht, wenn eine Operation in drei Stufen unterteilt werden kann. Das bedeutet, dass die mittlere Stufe die CAS Operation ausführt, die erste Stufe diese vorbereitet und die letzte Stufe die Nachbereitungen übernimmt.

Konflikt-Zählung Zur Detektion des fehlenden Fortschritts verwenden die Autoren ein Zählwerk (auch *Contention Counter* genannt), das sich bei jedem Fehlversuch einer CAS Operation erhöhen lässt. Sobald das Zählwerk die definierte Schwelle überschreitet, kann der hilferufende Pfad eingeschlagen werden [TP14, S.361].

Operation Record Um einer Operation Hilfe zu leisten, werden alle nötigen Informationen in den so genannten *Operation Record* gespeichert und in der Wait-Free Queue abgelegt.

3.3.2 Relevanz für die vorliegende Arbeit

Zur Überführung des Lock-Free Suchbaumes in einen Wait-Free Suchbaum, ist dieses Verfahren sehr interessant, da der Aufbau der vorliegenden Arbeit bereits der normalisierten Form ähnelt. Dies sollte eine Überführung nach dem Schritt-für-Schritt Ansatz ermöglichen.

Mangels zeitlicher Ressourcen im Rahmen der Masterarbeit wurde der Versuch den Algorithmus in einen Wait-Free Algorithmus zu konvertieren jedoch nicht unternommen. Übernommen wurde der Ansatz eines *Contention Counters*, der jedoch nicht wie von Timnat et al. vorgeschlagen fehlschlagende CAS Operationen zählt, sondern wiederkehrende While-Loops.

3.3.3 Ergebnisse

Abschließend sei erwähnt, dass die Überführung in eine gewünschte Wait-Free Datenstruktur die Verwendung einer beliebigen, bereits vorhandenen Wait-Free Datenstruktur voraussetzt. Diese wird als Zwischenspeicher für die *Operation Records* benötigt und erhöht potentiell die Komplexität der Implementierung entscheidend.

Die Autoren greifen das Thema auf, da sie auf die Existenz einer wesentlich größeren Menge an Lock-Free Datenstrukturen, als auf Wait-Free Datenstrukturen gestoßen

sind. Sie möchten damit eine Hilfestellung für zukünftiges Entstehen neuer Wait-Free Datenstrukturen leisten. Jede/r, die/der eine neue Wait-Free Datenstruktur erstellen möchte, sei diese Werk nahegelegt. Im Speziellen gilt dies, wenn der Protagonist bereits tief gehendes Wissen über Lock-Free Datenstrukturen besitzt.

3.4 The Amortized Complexity of Non-blocking Binary Search Trees

Das Werk von Ellen, Fatourou, Helga und Ruppert stellt im Jahr 2014 nach eigenen Aussagen die erste Implementierung eines Lock-Free Suchbaumes mit einer definierten Obergrenze der Zeitkomplexität vor.
Dies ist besonders wertvoll, um Aussagen darüber treffen zu können, wie effizient sich eine Datenstruktur bei unregelmäßiger (anstelle von kontinuierlicher) Verwendung verhalten wird [EFHR14, S.332].

3.4.1 Besonderheiten

Die Autoren stellen erstmals eine Obergrenze der Zeitkomplexität für einen Lock-Free Suchbaum vor. Dazu analysieren sie anhand einer ihrer Implementationen die amortisierte Zeitkomplexität eines Lock-freien Suchbaumes. Daraus wird der Big-Oh Beweis abgeleitet, dass die Funktionen *Search*, *Insert* sowie *Delete* als Obergrenzen $O(h(op)+\dot{c}(op))$ besitzen. Wobei $h(op)$ die Höhe des Baumes zu Beginn einer Operation op darstellt, wobei $\dot{c}(op)$ die maximale Anzahl an möglichen Zugriffen während der Abarbeitung von op repräsentiert.

3.4.2 Relevanz für die vorliegende Arbeit

Die Autoren verwenden in ihrer Implementation einen ähnlichen Ansatz wie jener in der vorliegenden Arbeit. Zur Gewährleistung, dass ein aktiver Teil des Baumes jederzeit erreicht werden kann (auch nach dem Löschen eines Knotens), stellen Ellen et al. eine Stack Datenstruktur vor, die sich die besuchten Knoten merkt und somit einen Rückzug[5] möglich macht. Das hat den Vorteil, dass keine zusätzliche Verwaltung von Back-Links notwendig ist [EFHR14, S.334-335] und der Rückzug nicht bis zur Wurzel, sondern darunter, möglich ist. Das stellt eine Verbesserung der Performance per Operation dar.

3.4.3 Ergebnisse

Es wird bewiesen, dass für endliche Prozesslaufzeiten α die folgende Obergrenze gilt: $O(\sum_{op\in\alpha}(\dot{c}(op) + h(op))$.
Abschließend wird eine Änderung im Rückzugsverhalten vorgestellt, die eine Verbesserung dieses Konzeptes sowohl gegenüber dem Referenzwerk der Autoren Ellen et al. [EFRvB10], als auch gegenüber der vorliegenden Arbeit darstellt. In Zukunft kann angedacht werden, dieses als Verbesserungsmaßnahme in die vorliegende Arbeit zu integrieren.

[5] Englisch: Backtracking

Die Autoren greifen das Thema auf, da Obergrenzen für Lock-Free Datenstrukturen bis zu diesem Zeitpunkt nicht untersucht wurden. Dies ist der Fall, weil Lock-Freiheit (siehe Unterabschnitt 2.2.4) keine zeitliche Obergrenzen einzelner Threads garantiert, sondern viel mehr einen Fortschritt des Gesamtsystems gewährleistet.

Dies stellt ein sehr theoretisches Feld dar, da eine Kontrolle der Parameter (zum Beispiel Baumhöhe oder die maximale Zugriffsanzahl) in einer sich ändernden Baumstruktur wohl kaum praktikabel ist. Darum wird die Praxis eher auf die aufwändigere Wait-Free (siehe Unterabschnitt 2.2.4) Variante zurückgreifen; im theoretischen Bereich erscheint dies jedoch für eine besser Vergleichbarkeit von Algorithmen sinnvoll.

3.5 A Contention-Friendly Binary Search Tree

Die Autoren Crain, Gramoli und Raynal implementieren einen modernen BST mit dem Ziel einen hohen Grad an Konfliktzugriffen zu erlauben Limitierung der Anzahl pro Zeiteinheit der selbigen, um die Performance zu erhöhen [CGR13]. Sie halten die strengen Garantien der asymptotischen (Big-Oh) Komplexität nur dann ein, wenn keine Konflikte bei der Bearbeitung des Baumes auftreten. Im parallelen Fall werden Konfliktfälle zu Kosten der asymptotischen Obergrenze vermieden.

3.5.1 Besonderheiten

Abstrakte Modifikationen Anders als Nurmi et al. (Unterabschnitt 3.6.1) führen die Autoren in diesem Werk ihre Forschungen anhand eines BST vor und implementieren zwei Arten von Entkoppelungen. Die erste Art, nämlich das Entkoppeln von *Update* und *Re-Balancing* Prozessen, wird auch im Werk von Nurmi et al. verwendet. Zusätzlich machen sich die Autoren das zweite und neuere Verfahren der Entkoppelung von logischem und physischem Entfernen von Knoten aus der Datenstruktur [CGR13, S.230] zunutze.

Träge strukturelle Anpassung Das Ziel der Entkoppelung ist es, den *Re-Balancing* Prozess auslagern zu können und komplett von den *Update* Operationen zu trennen.
Das hat laut Aussage der Autoren mehrere Vorteile: Sie können idempotente (mehrfaches Anwenden derselben Operationen führt zum selben Ergebnis) strukturelle Anpassungen, die parallelen Prozessen entspringen, zusammenfassen und somit den *Re-Balancing* Aufwand reduzieren. Weiters werden *Remove* Prozesse selektiv durchgeführt.

Selektives *Remove* Dabei wird in der Baumstruktur ein zu entfernender Knoten vorerst nur als entfernt markiert. Wann und ob dieser entfernt wird, wird am Grad der Auswirkung des Entfernens bemessen. Ein Knoten, auf den oft zugegriffen wird, hat einen potenziell größeren Effekt im Zuge dieser Operation, als einer auf den nur selten zugegriffen wird.

Dies ist eine effektive Maßnahme zur Verminderung der Anzahl der konfliktbehafteten Zugriffe, dabei gehen jedoch strikte asymptotische Garantien in Form von Invarianten verloren [CGR13, S.231].

3.5.2 Relevanz für die vorliegende Arbeit

Einerseits können die Feststellungen der Autoren bezüglich Lock-basierter Datenstrukturen als Bestätigung der Relevanz des vorliegenden Ansatzes gesehen werden, andererseits kann für zukünftige Forschung angedacht werden, das *Balancing* Verfahren als Erweiterung heranzuziehen.

3.5.3 Ergebnisse

Der untersuchte BST vereint mehrere Konzepte zur Hochlast-Handhabung in einer Datenstruktur. Die Autoren stellen fest, dass Lock-basierte Datenstrukturen gut von ihrem Hochlast-freundlichen Ansatz profitieren können und dass im Speziellen die Trennung der Modifikationen (*Abstrakte Modifikationen*) zu guter Skalierbarkeit führt [CGR13, S.239].

Die Autoren greifen dieses Thema auf, da sie Performance Verbesserungsmöglichkeiten suchen. Sie zielen auf eine Optimierung ab, bei der sie die möglichen Konflikte auf den BST limitieren (Big-Oh wird gebrochen), um in der Praxis eine höhere Performance zu erreichen.

Dieser sehr interessante Ansatz wird allen Lock-Free Datenstruktur Entwicklern nahegelegt. Weiters sei auf die Existenz des Buches [Ray13] von dem Co-Autor Michel Raynal hingewiesen.

3.6 Weitere Werke

In diesem Kapitel sollen für die vorliegende Arbeit relevante Werke aufgelistet werden, die Datenverarbeitung im Allgemeinen aber auch BSTs betreffen.

3.6.1 Uncoupling Updating and Rebalancing in Chromatic Binary Search Trees

Anhand eines chromatischen Baums stellen die Autoren Nurmi und Soisalon-Soininen ein Verfahren zur zeitlichen und strukturellen Entkoppelung der Update-Vorgänge vor, sodass *Re-Balancing* unabhängig von *Insert* und *Remove* Operationen durchgeführt werden können.
Die Methode hat den Vorteil, dass ein höherer Grad an Nebenläufigkeit ermöglicht wird [BER14, S.330] [GS78] und dass der *Re-Balancing* Prozess sich in einen eigenen Thread auslagern lässt [NSS96, S.192].

Vorbedingung ist jedoch, dass der Ziel-Baum ein Red Black Tree ist (ein selbst balancierende Spezialform eines BST, siehe siehe Abschnitt 2.3.3). Dieser Red Black Tree

muss so weit bekannt sein, dass er in einen Chromatischer Baum überführbar ist. Ein Chromatischer Baum ist eine abgewandelte Variante eines Red Back Trees, bei dem rot und schwarz durch nicht-negative Integer-Gewichte ersetzt [BER14, S.335] werden.

Zur Trennung der Änderungszugriffe lockern sie die Balancierungsbedingung von balancierten Suchbäumen so auf, dass nicht mehr direkt nach den *Insert* und *Remove* Vorgängen ein *Re-Balancing* notwendig ist [NSS91]. Anstatt *Re-Balancing* als zugehörig zu *Insert* beziehungsweise *Remove* zu sehen und als einen einzigen großen Schritt durchzuführen, teilen die Autoren diese Schritte in mehrere kleinere auf. Diese kleineren Schritte dürfen in sich zeitlich überlappen, was einen höheren Grad an Nebenläufigkeit ermöglicht.

Relevanz für die vorliegende Arbeit

Diese hier beschriebenen Techniken können bei der Implementierung eines *Re-Balancing* Verfahrens für die betrachtete Arbeit verwendet werden, das von Chatterjee et al. gänzlich vernachlässigt wird.

Ergebnisse

Die Autoren präsentieren den Grundstein zur strukturellen Entkoppelung der Update-Vorgänge und reduzieren dadurch die Anzahl an Locks entscheidend. Als zusätzliche Referenz kann die von den Autoren implementierte chromatische Baumstruktur [NSS96] angeführt werden, in der dieses Verfahren zum Einsatz kommt.

Sie greifen dieses Thema auf, da es ein Problem darstellen kann, zeitliche Zusicherungen für einen selbst-balancierten Baum zu tätigen. Das ist der Fall, weil die Komplexität des Umsortierens bei den *Insert* oder *Remove* Operationen zu tragen kommt. Diesen Vorgang entkoppeln die Autoren von den Operationen, um zeitliche Zusagen halten zu können.

3.6.2 A practical concurrent binary search tree

Bogué et al. [BCCO10] stellen einen Lock-basierenden Georgy Adelson-Velsky and Evgenii Landis' (AVL) Baum vor, der zu jedem *Insert* und *Remove* mit einer linearen Anzahl an *Rebalance*-Schritten auskommt, um Balancing zu erhalten.

Besonderheiten

Die Autoren haben ein stark optimiertes Fine grained locking (FGL) Verfahren entwickelt. Sie setzen dieses Verfahren in Kombination mit Techniken der optimistischen Nebenläufigkeit ein, um einen verbesserten Durchsatz bei dem Durchsuchen ihres Baumes zu erreichen [BER14, S.331].

Weitere Referenzwerke

Als Lock-basierende Referenz eines Suchbaum seien zwei Implementationen erwähnt: nämlich „A Practical Concurrent Self-Adjusting Search Tree“ [AKK+12], der sich kontinuierlich selbstständig balanciert (siehe Abschnitt 2.3.3), ist und „A practical concurrent binary search tree“ [BCCO10].

Des weiteren benutzen Howley und Jones [HJ12] einen model checker um Korrektheit zu beweisen, taten das aber laut Aussagen von Brown et al. [BER14, S.330] nicht richtig.

Abschließend sei die verlinkte Liste des Werks *A Pragmatic Implementation of Non-blocking Linked-Lists* erwähnt, bei der die Autoren [Har01] auf die Entkoppelung von logischem Entfernen und physischem Entfernen von Knoten im Zuge des *Remove* Prozesses setzen [CGR13, S.230].

KAPITEL 4

Untersuchung des Lock-freien BST Algorithmus

Das Paper „Efficient Lock-free Binary Search Trees“ von Chatterjee et al. [CNT14] befasst sich mit einem neuen Verfahren zur Implementierung eines Lock-Free BST. Chatterjee et al. stellen einen entwickelten Algorithmus vor, welcher Lock-Free Zugriffe auf einen BST gewährleisten soll.

In diesem Kapitel sollen in einer **abstrakten Beschreibung** die Eigenschaften des Algorithmus beschrieben werden. Darauf folgt eine **Beschreibung der Knoten, Links und Daten**, welche die Kernstücke des BST bilden und daher zum Verständnis des Algorithmus notwendig sind. Abschließend wird detailliert die Funktionsweise des Algorithmus beschrieben, in der die Knoten, Links und Daten Verwendung finden.

4.1 Abstrakte Beschreibung des Algorithmus

Das folgende Kapitel geht auf die Besonderheiten beziehungsweise Errungenschaften der betrachteten Arbeit ein.

Die wichtigsten Beiträge des **Lock-freien** Baums sind eine **Komplexitätsanalyse** der zur Verfügung stehenden Operationen, das Vorführen der aktiven **Speicherung von Markern in Links** anstelle der Speicherung in Knoten und eine Entscheidungsfindung für unterstützende Löschvorgänge - das sogenannte *Helping*.

4.1.1 Lock-freier BST

Lock-Freiheit ist die wichtigste Eigenschaft, die von Chatterjee et al. angestrebt wird. Die Beweisführung im Kapitel 4.2 [CNT14, S.329-330] für das korrekte Verhalten der *Remove*-Operation, zeigt alle möglichen zeitlichen Sequenzen von Konfliktfällen auf und definiert

das Lock-freie Soll-Verhalten. Weiters trifft *Lemma 11* [CNT14, S.330] Aussagen über eine konstante Abarbeitungszeit im Falle eines Konflikts, in dem ein Thread von einem anderen unterbrochen wird. Dabei werden ausschließlich Single Word CAS-Operationen von dem Algorithmus verwendet. Außerdem wissen wir:

- „that a concurrent object is linearizable if each method call appears to take effect instantaneously at some moment between that method's invocation and return events." [HS12, Kapitel 3.6]
- Dass ausschließlich CAS-Operationen zur Manipulation verwendet werden und diese sicherstellen, dass Änderungen zu einem Zeitpunkt für alle sichtbar werden.

Durch dieses Wissen und dadurch, dass kein Anzeichen im Zuge der Implementierung gegen Lock-Freiheit gesprochen haben, wird Lock-Freiheit als plausibel erachtet.

4.1.2 Komplexitätsanalyse

Die primäre Leistung des entwickelten Verfahrens liegt in der Zeitkomplexität, die sowohl analysiert als auch zugesichert wird. Andere Forschungen betreffend Lock-Free Suchbäumen von Natarajan und Mittal [NM14], Brown, Ellen und Ruppert [BER14], sowie das Werk von Drachsler, Vechev und Yahav [DVY14] führten keine theoretische Komplexitätsanalyse durch [CNT14, S.322]. Stadtessen führen sie Benchmarks durch, welche den Nachteil haben, dass keine Informationen über die Effizienz außerhalb der gebenchmarkten Parameter vorliegen [EFHR14, S.332].

Komplexität

Chatterjee et al. zeigen eine maximale Zeitkomplexität aller Operationen von $O(H(n) + c)$ auf. Wobei $H(n)$ die Höhe des Baums darstellt, n die Knotenanzahl und c die Anzahl an Konflikten im Zuge der ausgeführten Operation [CNT14, S.322]. Wobei das Hauptaugenmerk auf dem additiven Charakter der Konfliktanzahl c liegt.

Dies wurde sichergestellt indem in jedem Pfad im Quellcode ein Hilfsmechanismus etabliert wurde, welcher von allen Threads zu Beginn jeder Funktion auf den BST ausgeführt wird. Dies geschieht über die Methode (namentlich *Locate*) die für das Iterieren durch den Baum zuständig ist. Ein so entstandener Konflikt wird von allen darauf folgenden Operationen zu beheben versucht, was die Komplexitätsanalyse plausibel erscheinen lässt.

Da der Algorithmus nicht vollständig implementiert werden konnte (siehe Abschnitt 6.1 Zentrale Ergebnisse), wurde dies jedoch nicht empirisch überprüft.

4.1.3 Linearisierbarkeit

„Proving that an algorithm implements an atomic object to this end, we need to prove that all histories generated by the algorithm are linearizable, i.e. identify a linearization

of its operations that respects the 'real-time' occurrence order of the operations and that is consistent with the sequential specification of the object“ [Ray13, S.121].

Die beschriebene History bezogen auf den untersuchten Algorithmus kann die Operationen *Insert*, *Remove* und *Contains* enthalten. Wobei sich der Linearisierungspunkt von Operation zu Operation wie folgend unterscheidet:

Insert besitzt zwei Linearisierungspunkte, jeweils für einen positiven und negativen Operationsausgang. Im positiven Fall geschieht dies durch das einzige *CAS*, das von *Insert* benötigt wird (siehe Pseudocode Zeile 173 aus Abbildung 4.9). Im negativen Fall liegt der Linearisierungspunkt nach dem Suchen, ob der Key bereits im Baum vorhanden ist (siehe Pseudocode Zeile 10 aus Abbildung 4.5).

Remove besitzt einen Linearisierungspunkt für den positiven und zumindest zwei Linearisierungspunkte im Falle eines negativen Operationsausgangs. Im positiven Fall erfolgt dies mit dem *CAS*, der den Back-Link (siehe Abschnitt 4.2.2) umhängt. Welche *CAS* Operation das ist, hängt von der Knoten-Kategorie ab: Kategorie 1 mit Schritt XXV, Kategorie 2 mit XXII, Kategorie 3 mit XVI (siehe Tabelle 4.1). Der negative Fall zwei teilt sich in a) Der zu löschende Knoten ist nicht im Baum existent, dann gleicht der Linearisierungspunkt dem der folgenden *Contains* Operation und b) Der zu löschende Knoten existiert, sein OrderLink (siehe Abschnitt 4.2.2) wurde aber bereits von einem parallelen *Remove* mit einem Flag-Marker (siehe Abschnitt 4.2.2) versehen, dann liegt der Linearisierungspunkt bei dem parallelen *Remove* Vorgang.

Contains besitzt ebenfalls mehrere Linearisierungspunkte. Für den positiven Operationsausgang liegt der Linearisierungspunkt (gleich wie bei einem fehlgeschlagenen *Insert*) nach dem Suchen, ob der Key bereits im Baum vorhanden ist (siehe Pseudocode Zeile 10 aus Abbildung 4.5). Für den negativen Operationsausgang gibt es zwei Szenarien: a) Der gesuchte Knoten war zur Zeit der Operation keinesfalls im Baum vorhanden, dann befindet sich der Linearisierungspunkt zu Beginn der Operation selbst (siehe Zeile 26 Abbildung 4.10) und b) Der gesuchte Knoten befand sich zu Beginn der *Contains* Operation im Baum, wurde aber während der Traversierung von einem parallelen *Remove* gelöscht; dann liegt der Linearisierungspunkt bei dem parallelen *Remove* Vorgang.

4.1.4 Einschränkungen des Baumes

Der entworfene BST ist nicht selbst balancierend (siehe Abschnitt 2.3.3), was das Potential für Entartungen mit sich bringt.

4.2 Beschreibung der Knoten, Links und Daten

Dieses Kapitel beschreibt angewandte Techniken, um einen Aufbau des Baumes zu ermöglichen. Der Lock-freie Baum besteht aus unterschiedlichen Kategorien von **Knoten im Baum**. Er verfügt über unterschiedliche Arten von **Links zwischen Knoten**, welche die Knoten verbinden, und eine spezielle Art Daten in Links zu speichern.

4.2.1 Knoten im Baum

Die zentralen Elemente des Baumes bilden die Knoten, die im Speicher wie in Algorithmus 4.2 zu sehen ist abgebildet sind. Der Algorithmus implementiert einen abstrakten Datentyp (ADT), der einen binärer Suchbaum abbildet, in dem jeder Knoten durch einen einzigartigen Key k identifiziert wird und einer totalen linearen Ordnung unterliegt.

Jeder Knoten besteht neben seinen Nutzdaten auch noch aus einem Quadrupel aus den Links *LeftChild*, *RightChild*, *BackLink* und *Prelink*. Diese sind Zeiger auf einen beliebigen Knoten im BST, den Ursprungsknoten miteinbezogen.

Die Zahl der Speicherwörter, die bei dieser Implementierung verwendet werden, ist $5n$, wobei n die Anzahl an Knoten angibt.

Knotenkategorisierung

Knoten mit und ohne Child-Knoten benötigen im Falle einer Baummanipulation unterschiedliche Schritte. Um diese logisch zu trennen, werden Knoten anhand ihrer Child-Knoten wie folgend ein **drei** Kategorien unterteilt. Knotenkategorisierung passiert über den speziellen Ursprung eines Links, der sich Order-Link (siehe: Abschnitt 4.2.2 Order-Links) nennt. Knoten werden in die Kategorien eins bis drei unterteilt und unterscheiden den zu tätigenden Vorgang beim Entfernen von Knoten aus dem Baum. Je nach Kategorisierung unterscheiden sowohl der Quellcode als auch das Paper unterschiedliche Vorgehensweisen bei verschiedenen Operationen. Dies betrifft eine Unterscheidung aller *Update* Prozesse und im Speziellen der *Remove*-Operation. Folgend werden die drei untersuchten Kategorien von Knoten definiert:
Knoten vom Typ eins sind jene, deren *Order-Link* (siehe Abschnitt 4.2.2) ihrer selbst entspringt; Jene vom Typ zwei besitzen einen *Order-Link*, der dem linken *Child* entspringt; In Kategorie drei besitzen Knoten einen eingehenden *Order-Link*, der dem ganz rechten Knoten in seinem linken sub-Baum entspringt (siehe Abbildung 4.1).

Ein Algorithmus zur Ermittlung der Knotenkategorie kann wie in Algorithmus 4.3 aussehen.

Sentinel-Knoten

Um garantierte Bezugspunkte im Baum zu besitzen, werden Sentinel- beziehungsweise „Dummy“-Knoten [CNT14, S.325] verwendet. Dieses Konzept umfasst zwei fortwährend

Algorithmus 4.1 : Flagged Node Implementierung

```
1  class FlagNodePtr {
2  public:
3  enum flagID : size_t { flag = 61, mark  = 62, thread = 63};
4  FlagNodePtr(ANodePtr rawPointer,bool flag,bool mark,bool thread) {
5    // detect upper three bits "active low" or "active high" to react
6    //  appropriatly (e.g. SPARC stack vs. heap allocated memory)
7    const uint64_t clearFlagsMask = ~upperThreeBits_mask;
8    uint64_t flaggedPtr = (uint64_t) rawPointer & clearFlagsMask;
9    flaggedPtr |= (uint64_t) flag << flagID::flag;
10   flaggedPtr |= (uint64_t) mark << flagID::mark;
11   flaggedPtr |= (uint64_t) thread << flagID::thread;
12   _flaggedPtr = (T*) flaggedPtr;
13 }
14
15 /** returns the clean pointer */
16 ANodePtr get() const {
17   // Sparc specific detection
18   const uint64_t upperThreeResetVal =
19       ((uint64_t)_flaggedPtr & beforeLastThreBits_mask) << 3;
20   const uint64_t clearFlagsMask = ~upperThreeBits_mask;
21   return (ANodePtr) (((uint64_t)_flaggedPtr & clearFlagsMask)
22       | upperThreeResetVal);
23 }
24
25 /** flag getter */
26 ANodePtr bool isFlagged() const {
27   return (uint64_t)_flaggedPtr & ((uint64_t)1 << flagID::flag);
28 }
29 ANodePtr bool isMarked() const {
30   return (uint64_t)_flaggedPtr & ((uint64_t)1 << flagID::mark);
31 }
32 ANodePtr bool isThreaded() const {
33   return (uint64_t)_flaggedPtr & ((uint64_t)1 << flagID::thread);
34 }
35
36 private:
37 static constexpr uint64_t upperThreeBits_mask =
38                 (uint64_t)0xE0 << (sizeof(void*)*8-8);
39 static constexpr uint64_t beforeLastThreBits_mask =
40                 (uint64_t)0x1C << (sizeof(void*)*8-8);
41
42 /** flag encoded pointer type T */
43 ANodePtr _flaggedPtr {nullptr};
44 };
```

Algorithmus 4.2 : Aufbau der Knoten mit atomischen Links

```
1 struct AtomicNode_s {
2     IDType value;
      /* Es folgen atomische Links zu weiteren Knoten        */
3     atomic<FlagNodePtr> leftChild;
4     atomic<FlagNodePtr> rightChild;
5     atomic<ANodePtr> backLink;
6     atomic<ANodePtr> preLink;
7 }
```

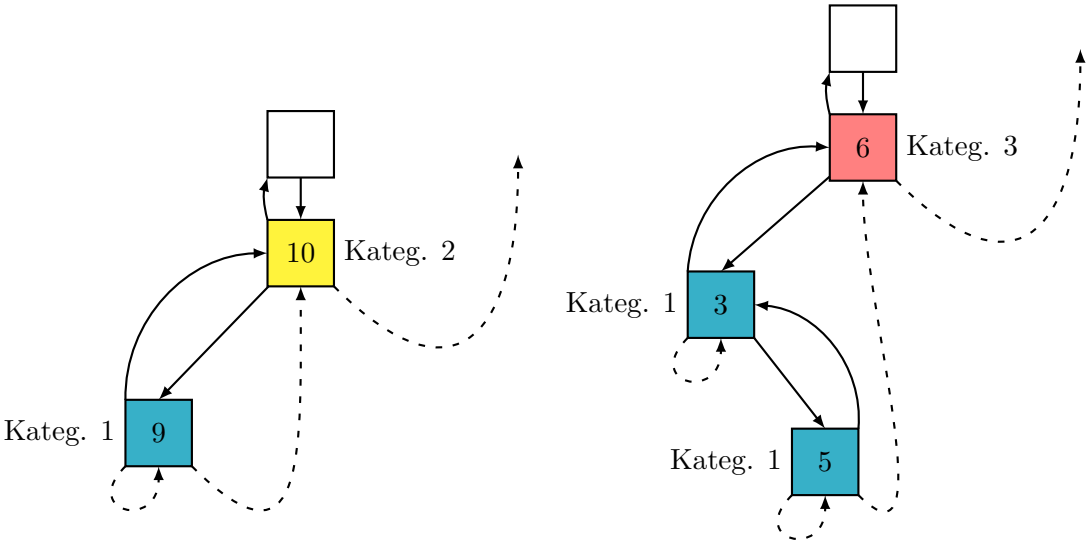


Abbildung 4.1: Kategorisierungen von Knoten

Algorithmus 4.3 : Schema zur Ermittlung der Knotenkategorie

```
1  if (node.leftChild == node)
2    return 1;
3  else if (node.leftChild.righChild == node)
4    return 2;
5  else
6    return 3;
```

verfügbaren Knoten ∞ und $-\infty$ (beziehungsweise *inf* und *-inf*), welche die Wurzel des BSTs bilden. Algorithmus 4.4 veranschaulicht eine mögliche Art der Umsetzung einer solchen Struktur.

Algorithmus 4.4 : Atomische Knoten und Sentinel Knoten

```
1  //Interface zur Initialisierung von ANode_t
2  struct ANode_s( IDType k,
3    FlagNodePtr leftChild, FlagNodePtr rightChild,
4    ANodePtr    backLink,  ANodePtr    preLink
5  );
6  //Initialisierung der Sentinel Knoten
7  struct ANode_s root[2] = {
8   {IDType::NEG_INFITY,
9    {&root[0], 0,0,1}, {&root[1], 0,0,1}, &root[1], nullptr},
10  {IDType::POS_INFITY,
11   {&root[0], 0,0,0}, {nullptr , 0,0,1}, nullptr , nullptr}
12 };
```

4.2.2 Links zwischen Knoten

Links sind Verbindungen zwischen zwei Konten, die als Pointer im Computerspeicher realisiert werden.

Den Links zwischen den Knoten kommen in dieser Arbeit eine besondere Bedeutung zu, da sie zusätzlich zu ihren Zeigern Informationen enthalten. Es existieren unterschiedliche Link-Arten, wobei die komplexeren unter ihnen Informationen über die Art der Verbindung (Threaded oder nicht) und über deren Zustand (Marked, Flagged) halten. Dies benötigt drei extra Bits, welche im Pointer abgelegt werden.

Dieses Kapitel erläutert Unterscheide zwischen den Arten der Links, sowie die Semantik der Informationen im Detail.

Arten von Links

In diesem Abschnitt folgt eine Übersicht der von Chatterjee et al. verwendeten Links zwischen Knoten, sowie eine Erläuterung ihrer speziellen Verwendungen im BST. Alle Link-Typen werden im Speicher als Pointer repräsentiert.

Threaded-Links werden zur Bestimmung von Vorgängerknoten und der Knotenkategorie verwendet. Die entstehende Baumstruktur hat ihren Ursprung im Werk von Perlis und Thornton [PT60], die diese Struktur das Threaded-Format nannten.

Diese Threaded-Links sind linke und rechte Child-Links eines Knotens, die in der jeweiligen Richtung keinen hierarchischen Folgeknoten besitzen, wie dies Abbildung 4.2 illustriert. Beim Threading werden ausgehende *LeftChild* Links zu dem Ursprungsknoten selbst verbunden. Threaded *Right-Child Link*s werden zu dem logischen Vorgängerknoten verbunden, was in Abbildung 4.2 und Abbildung 4.1 anhand der rechten strichlierten Links ersichtlich ist.

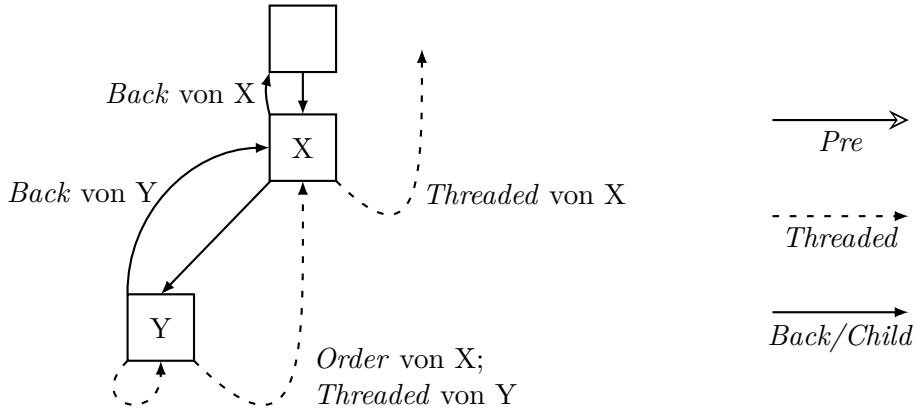
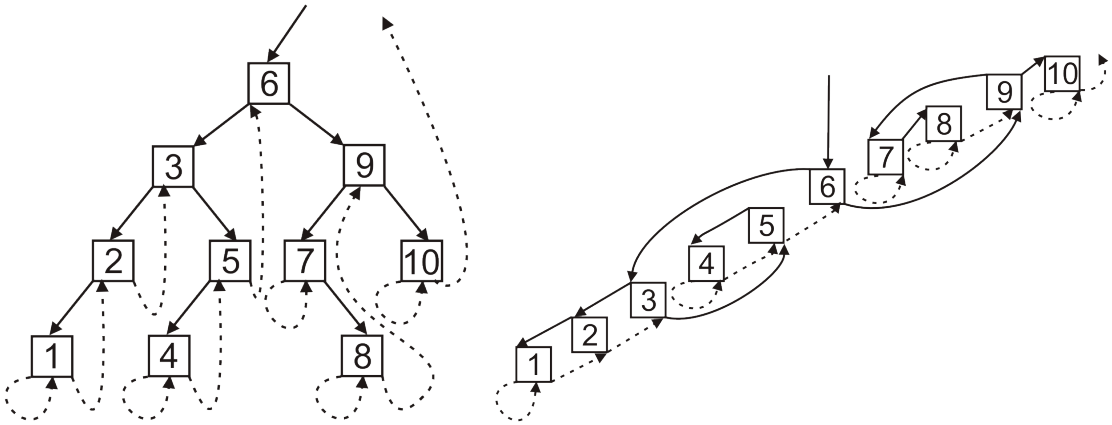


Abbildung 4.2: Bestehende Link-Arten



(a) Threaded BST (b) Repräsentative als geordnete Liste

Abbildung 4.3: Repräsentation des implementierten BSTs als geordnete Liste [CNT14]

Aufgrund diesen speziellen Aufbaus:

- gleicht das Durchsuchen dem einer geordneten Liste, was in Abbildung 4.3 beispielhaft gezeigt wird.
- existiert maximal ein eingehender *Threaded-Link* je Knoten, der im Zielknoten den *Order-Link* darstellt.

Order-Links werden für jeden Knoten durch den einzigen eingehenden *Threaded-Link* gebildet. Man verwendet sie um die Knoten einer der **drei** Knotenkategorien zuzuordnen, wobei der Link-Ursprung ausschlaggebend für diese Zuordnung ist.

Knoten der Kategorie eins verfügen als einzige über einen *Order-Link*, der ihnen selbst entspringt (siehe Knoten 3, 5 und 9 in Abbildung 4.1), nämlich ihr linker ausgehender *Threaded-Link*. Kategorie 2-Knoten besitzen einen *Order-Link*, der ihrem linken Child entspringt (siehe Knoten 10 in Abbildung 4.1). Knoten der Kategorie drei besitzen die komplexeste Konstellation des Ursprungs des *Order-Links*, dieser entspringt nämlich einem beliebig weit entferntem rechten *Child* des direkten linken *Child*s des Ursprungsknotens (siehe Knoten 6 in Abbildung 4.1).

Back-Links werden verwendet, um fehlschlagende Operationen nach der benötigten Hilfestellung bei ihren Nachbarknoten erneut starten zu lassen. Sie werden jedoch nicht verwendet, um den Baum zu durchsuchen. Back-Links zeigen immer auf Knoten, die genau einen Link entfernt sind. Jeder Knoten hat einen ausgehenden Back-Link, der zusichert, auf einen existierenden Knoten im Baum zu zeigen, der einen einzigen Link vom fehlschlagenden Punkt entfernt ist. Back-Links werden jedoch weder zum Durchforsten der Baumstruktur, noch für *Insert* oder *Remove*-Operationen verwendet [CNT14, S.224].

Pre-Links werden ausschließlich während einer laufenden *Remove*-Operation gesetzt und dienen der Wiederaufnahme der gerade ausgeführten Update Operation im Konfliktfall. Sie verbinden den zu entfernenden Knoten mit ihrem Order-Knoten (siehe Abschnitt 4.2.2) bevor einer der ausgehenden Links mit dem *Mark* Indikator versehen wird [CNT14, S.324].

Speicherung von Informationen in Links

Ein gängiges Verfahren ist das Speichern benötigten Informationen in den Knoten selbst, wie dies Ellen et al. [EFRvB10] sowie Howley und Jones [HJ12] einsetzen. Dieses Verfahren birgt Nachteile beim Einsatz von Single Word CAS-Operationen, da beim Manipulieren des Baumes faktisch ganze Knoten blockiert werden müssen, um diese einer Änderung zu unterziehen.

Chatterjee et al. behaupten durch die Speicherung von Informationen in den Links anstelle von Knoten eine signifikante Verbesserung der parallelen Zugreifbarkeit erreicht zu haben. Dies wurde nicht kontrolliert, da keine Referenzimplementierung vorliegt oder gefunden wurde.

Link-Zustände

Zur Speicherung von Zuständen werden **drei Bits** direkt im Pointer des jeweiligen Links abgelegt, was ein gängiges Verfahren darstellt [CNT14, S.325] und unter dem Namen Bit Diebstahl[1] bekannt ist.
Diese drei Bits verwenden die Indikatoren: 1) *Threaded* der logisch den Link-Typen (und keinem Zustand) zugeordnet ist und schon in Abschnitt 4.2.2 beschrieben wurde, 2&3) *Marked* und *Flagged*, welche in diesem Kapitel beschrieben werden sollen.

[1] Englisch: Bit Stealing

Die Speicherung beruht darauf, dass die oberen Bits nicht Teil des adressierbaren Adressraumes sind sondern ungenutzt bleiben. Deshalb ist es möglich, die überschüssigen Bits als *Marked* Indikator zu verwenden, sodass das *Flag* und der Pointer in einem gemeinsamen CAS atomisch getauscht werden können.

Diese Implementierung verfügt über zwei separate Zustände für *Child-Links* die eingesetzt werden, um im Gang befindliche Veränderungen der Links zu kennzeichnen. Bevor Links im Baum verändert werden, markiert der Algorithmus diese mit einer der beiden Indikator-Bits *Flag* oder *Mark*.
Sollte einer dieser beiden Zustände auf einen Link zutreffen, kann er nicht mehr für *neue Insert* oder *Remove* verwendet werden, solang der aktuelle Vorgang nicht abgeschlossen ist [CNT14, S.324]. Abbildung 4.4 zeigt die verwendete Visualisierung der beiden Zustände *Flag* und *Mark*.

Abbildung 4.4: Grafische Darstellung der Link Zustände

Flag Abhängig von der Knotenkategorie werden die folgenden eingehenden Links zu einem Knoten, der verändert werden soll, mit dem *Flag* Indikator gekennzeichnet. Für alle drei Knotenkategorien gilt, dass als Erstes der eingehende *Order-Link* ge-flagged wird und nach einigen Zwischenschritten dann der *Child-Link* des Vorgängerknotens mit einem *Flag* Indikator versehen wird.

Mark Der *Mark*-Indikator wird verwendet, um das Löschen eines Knotens anzuzeigen oder ihn als entfernt zu markieren. Ein Knoten versteht sich als gelöscht, wenn sein rechter *Child-Link* mit dem *Mark* Indikator versehen ist.

4.3 Funktionsweise des Algorithmus

Dieses Kapitel beschreibt wie die Links, Knoten und Daten verwendet werden müssen, um die primären Operationen *Contains*, *Insert* und *Remove* (siehe Abschnitt 2.3.3) ausführen zu können. Die primären Operationen bilden das User-Application Programming Interface (API), beziehungsweise sind sie jene Methoden, die der Baum vorrangig zur Verfügung stellt.

Die drei primären Operationen kapseln ihre Funktionalitäten in fünf Hilfsmethoden, weshalb diese vor den primären Operationen erläutert werden. Bei der Beschreibung der primären Operationen liegt der Fokus auf einer **exemplarischen Erläuterung** der *Remove* Operation.

LeserInnen, die an der generellen Funktionalität des Algorithmus interessiert sind, können direkt zu den primären Operationen (beginnend mit Unterabschnitt 4.3.2 auf Seite 50) springen.

4.3.1 Hilfsmethoden

Die verwendeten fünf Hilfsmethoden *Locate*, *TryFlag*, *TryMark*, *CleanFlag* und *CleanMark* sollen folgend ausführlich erklärt werden.

Die vier Hilfsmethoden *TryFlag*, *TryMark*, *CleanFlag* und *CleanMark* werden ausschließlich während des Entfernens eines Knotens benötigt, nicht jedoch bei einer *Contains* oder *Insert* Operation. Das Entfernen von Knoten kann aber sehr wohl im Zuge einer Hilfestellung für einen anderen Thread, indirekt während eines *Contain* oder *Insert*, angestoßen werden.
TryFlag bemüht sich darum, den Link-Zustand *Flag* zu setzen, wobei *TryMark* sich um das Setzen des Link-Zustandes *Mark* kümmert. *CleanFlag* und *CleanMark* verarbeiten den kompletten Löschvorgang mit der Ausnahme des ersten Schrittes, dieser wird von *Remove* selbst vorgenommen. Die beiden Methoden *CleanFlag* und *CleanMark* rufen sich rekursiv gegenseitig auf, was sie zu den komplexesten Methoden dieses Algorithmus macht.

Den Erläuterungen der *Remove* Operation aus Unterabschnitt 4.3.4 vorgreifend sei erwähnt, dass sich die *Remove* Operation in das Pre-Processing und in finale Schritte unterteilen lässt. Diese Unterteilung wird im Folgenden verwendet, wobei sich **römische Zahlen** auf die jeweiligen Schritte aus Tabelle 4.1 der *Remove* Operation beziehen.

Locate

Locate wird zur Auffindung von Knoten im Baum von allen drei der primären Operationen verwendet.

Locate sucht im Baum nach einem Key k und retourniert zwei Knoten $prev = x(k_i)$ und $curr = x(k_j)$ im Intervall $[k_i, k_j]$, sodass entweder $\{k_i \leq k < k_j\}$ oder $\{k_i \geq k > k_j\}$ gilt. Befindet sich k im Baum, stellt das Intervall den Zielknoten und dessen Vorgänger dar. Befindet sich k nicht im Baum, stellt das Intervall die Knoten vor und nach der Zielposition dar.

Aufrufende *Locate* wird von allen drei primären Operationen *Contains*, *Insert* und *Remove* aufgerufen, da diese bevor sie mit ihrer eigentlichen Tätigkeit beginnen, erst die zu bearbeitenden Knoten finden müssen.

Funktionsweise Folgend wird die Funktionsweise anhand des Pseudocodes aus Abbildung 4.5 erläutert. Nach dem Aufruf beginnt die While-Loop aus Zeile 9 durch den Baum zu iterieren. Die Iteration geschieht über die Zeile 25, wobei sich die Schleife durch eine Manipulation der übergebenen Pointer *prev* und *curr* vorwärts bewegt. In Zeile 10 und 11 wird ein Vergleich zwischen dem gesuchten k und *curr* angestellt.

```
   // Locate returns 2 if key is found otherwise 0 or 1 if the last visited link is
      left or right respectively.
 8 int Locate(NPtr& prev, NPtr& curr, KType k)
 9 while true do
       // Function cmp returns 2 := equal, 1 := greater than and 0 := less than
10     dir = cmp(k, curr→e);
11     if (dir = 2) then return dir;
12     else
13         (R, ∗, m, t) = curr→child[dir];
           /* If eager helping is required then help cleaning the current node if its
              right child pointer has been marked; this
           cleaning can not fail.  */
14         if ((m = 1) and (dir = 1)) then
15             newPrev = prev→backLink;
16             CleanMarked(prev, curr, dir);
17             prev = newPrev;
18             pDir =cmp(k, prev→k);
19             curr = (prev→child[pDir])·ref;
20             continue;
21         if t then
               // Check the stopping criterion.
22             nextE = R→k;
23             if ((dir = 0) or (k < nextE)) then
24                 return dir;
25             else prev = curr; curr = R;
```

Abbildung 4.5: Pseudocode der Methode *Locate* [CNT14, S.326]

Als Rückgabewert von *Locate* kommen drei Werte in Frage: 0 & 1) Der gesuchte Knoten k wurde nicht im Baum gefunden (Zeile 24 quittiert), 2) k im Baum vorhanden und der *curr* Pointer zeigt auf ihn (Zeile 10 quittiert).

TryFlag

Diese Hilfsmethode setzt selbständig das Flag-Bit des so genannten Desired-Link. Der Desired-Link ist jener, der die Knoten $x(prev)$ und $x(curr)$ verbindet (siehe Abbildung 4.6). *TryFlag* wird für die *Remove* Operation verwendet, um die Pre-Processing Schritte I sowie V auszuführen.

Aufrufende *TryFlag* wird ausschließlich von *Remove* und *CleanMark_(right)* aufgerufen.

Funktionsweise Folgend wird die Funktionsweise anhand des Pseudocodes aus Abbildung 4.6 erläutert. *TryFlag* „returns true only if the operation performing it could successfully flag the desired link at line 45. If the CAS step to atomically flag the link fails then it checks the reason of failure. If it fails because some other thread already had successfully flagged the link, false is returned, line 50“ [CNT14, S.326].

```
41 bool TryFlag(NPtr& prev, NPtr& curr, NPtr& back, bool isThread)
42 while true do
43     pDir = cmp(curr→k, prev→k) & 1;
       // Try atomically flagging the parent link.
44     t = isThread;
45     result = CAS(prev→child[pDir], (curr, 0, 0, t), (curr, 1, 0, t));
46     if result then return true;
47     else
           /* The CAS fails, check if the link has been marked, flagged or the curr
           node got deleted. If flagged, return false; if
           marked, first clean it; else just proceed */
48         (newR, f, m, t) = prev→child[pDir];
49         if (newR = curr) then
50             if f then return false;
51             else if m then
52                 CleanMarked(prev, pDir);
53             prev = back;
54             pDir =cmp(curr→k, prev→k);
55             newCurr = (prev→child[newPDir])·ref; Locate(prev, newCurr, curr→k);
56             if (newCurr ≠curr ) then
57                 return false;
58             back = prev→backLink;
```

Abbildung 4.6: Pseudocode der Methode *TryFlag* [CNT14, S.326]

Fehlschlagen kann die Operation aus drei Gründen:

1. Ein anderer Thread hat den Link bereits erfolgreich ge-flagged, dann wird *false* retourniert.
2. Wenn der zu markierende Link ge-marked wurde, dann wird die Operation, welche den *Mark*-Indikator gesetzt hat, zuerst abgearbeitet. Dies beginnt mit Zeile 51.
3. Ein neuer Knoten wurde in den Baum eingefügt.

In den Fällen zwei und drei bewegt sich der Algorithmus via der gespeicherten Variable *back* (Zeile 58) einen Schritt im Baum zurück und beginnt die Operation erneut.

TryMark

Diese Methode setzt atomisch den *Mark* Indikator in dem Link, der dem Knoten $x(curr)$ in die Richtung *dir* entspringt.

Aufrufende *TryMark* wird von den folgenden Methoden heraus aufgerufen: Von *CleanFlag* im Zuge des Schritts VI (siehe Zeile 95, Abbildung 4.8) und von *CleanMark* während des Pre-Processings zum Entfernen eines Kategorie 3-Knotens im Zuge von Schritt VII (siehe Zeile 144, Abbildung 4.9). *TryMark* wird jedoch nicht im Zuge des

```
59 void TryMark(NPtr& curr, int dir)
60 while true do
61     back = curr→backLink;
62     (next, f, m, t) = curr→child[dir];
63     if (m == 1) then break;
64     else if (f == 1) then
65         if (t == 0) then
66             CleanFlag(curr, next, back, false); continue;
67         else if ((t == 1) and (dir == 1)) then
68             CleanFlag(curr, next, back, true); continue;
69     result = CAS(curr→child[dir], (next, 0, 0, t), (next, 0, 1, t));
70     if result then break;
```

Abbildung 4.7: Pseudocode der Hilfsmethode *TryMark* [CNT14, S.326]

Schrittes III des Pre-Processings verwendet, dieser Schritt besitzt seine eigene CAS Operation (siehe Zeile 86, Abbildung 4.8).

Funktionsweise Der aus dem Knoten $x(curr)$ ausgehende Link in Richtung *dir* wird vor dem Setzen des *Mark* Zustandes auf *Flag* überprüft. Ist der *Flag* Zustand gesetzt, hilft *TryMark* dem Thread, der den *Flag* Marker gesetzt hat.

Fehlschlag Sollte das Setzen von *Mark* durch ein paralleles *Flagging* fehlschlagen, hilft *TryMark* dem Thread, der den *Flag* Marker gesetzt hat (siehe Zeile $64 - 68$, Abbildung 4.7); Mit der Ausnahme, dass es sich um einen *Threaded*-Left Link handelt, da dies nämlich auf eine laufende *Remove* Operation eines Kategorie 1-Knoten hindeutet. Dieser Fall wird von *CleanFlag* in den Zeilen $100 - 101$ bearbeitet.

CleanFlag

Diese Methode hat mehrere Funktionen: neben den *Remove* Pre-Processing Schritten V und VI ist *CleanFlag* hauptverantwortlich für die finalen *Remove* Schritte von Kategorie 1- und 2-Knoten verantwortlich, welche in Unterabschnitt 4.3.4 *Remove*-Operation detailliert erklärt werden.

Aufrufende *CleanFlag* wird von den folgenden Methoden heraus aufgerufen: Von *Add* selbst, nach dem Fehlschlag bei der Überprüfung, ob ein Link mit dem *Flag* Indikator versehen wurde (siehe Zeile 181, Abbildung 4.8). Von *Remove* aus Zeile 39, aus einem normalen Ablauf jeder Knotenkategorie heraus, um die auf CAS_I folgenden Schritte in den Weg zu leiten. *TryMark* aus den Zeilen 66 und 68 im Falle, dass ein Link mit einem *Flag* Indikator gefunden wurde. Von *CleanFlag* selbst aus Zeile 118 heraus, nach dem Schritt V eines *Remove* Knotenkategorie 3 Pre-Processings. Von *CleanMark* nach Schritt V eines Kategorie 1- oder 2-Knotens aus der Zeile 130. *CleanMark* handhabt weiters den Schritt IV von Kategorie 3-Knoten. Sollte hier schon ein *Flag* gefunden werden,

```
71 void CleanFlag(NPtr& prev, NPtr& curr, NPtr& back, bool isThread)
72 if (isThread ) then
       // Cleaning a flagged order-link
73     while true do
           // To mark the right-child link
74         (next, f, m, t) = curr→child[1];
75         if (m) then break;
76         else if (f) then
               // Help cleaning if right-child is flagged
77             if (back = next) then
                   // If back is getting deleted then move back.
78                 back = back→backLink;
79             backNode = curr→backLink;
80             CleanFlag(curr, next, backNode, t);
81             if (back = next) then
82                 pDir =cmp(prev→k, backNode→k);
83                 prev = back→child[pDir];
84         else
85             if (curr→preLink ≠ prev) then curr→preLink = prev;
86             result = CAS(curr→child[1], (next, 0, 0, t), (next, 0, 1, t)) ;
87             if result then break;
88     CleanMark(curr, 1);
89 else
90     (right, rF, rM, rT) = curr→child[1];
91     if (rM) then
92         (left, lF, lM, lT) = curr→child[0];
93         preNode = curr→preLink;
94         if (left ≠ preNode) then // A category 3 node
95             TryMark(curr, 0);
96             CleanMark(curr, 0);
97         else
98             pDir =cmp(curr→k,prev→k);
99             if (left = curr) then
100                CAS(prev→child[pDir], (curr , f, 0, 0), (right, 0, 0, rT));
101                if (!rT) then CAS(right→backLink, (curr, 0, 0, 0), (prev, 0, 0, 0));
102            else
103                CAS(preNode→child[1], (curr, 1, 0, 1), (right, 0, 0, rT));
104                if (!rT) then CAS(right→backLink, (curr, 0, 0, 0), (prev, 0, 0, 0));
105                CAS(prev→child[pDir], (curr, 1, 0, 0), (preNode, 0, 0, rT));
106                CAS(preNode→backLink, (curr, 0, 0, 0), (prev, 0, 0, 0));
107    else if (rt and rF) then
           // The node is moving to replace its successor
108        delNode = right;
109        while true do
110            parent = delNode→backLink;
111            pDir =cmp(curr→k, prev→k);
112            (∗, pF, pM, pT) = parent→child[pDir];
113            if (pM) then CleanMark(parent, pDir);
114            else if (pF) then break;
115            else if (CAS(parent→child[pDir], (curr, 0, 0, 0), (curr, 1, 0, 0))) then
116                break;
117        backNode = parent→backLink;
118        CleanFlag(parent, curr, backNode, true);
```

Abbildung 4.8: Pseudocode der Methode *CleanFlag* [CNT14, S.327]

wird *CleanFlag* ebenfalls aus Zeile 138 heraus aufgerufen, beziehungsweise nach einem fehlgeschlagenen CAS_{IV} in der Zeile 140.

Funktionsweise Primär muss man die Funktion nach dem Zustand des übergebenen Arguments **`bool`** `isThread` teilen. Grundsätzlich versucht sie, den *Flag*-Status des Order-Links (Argument $isThread = \top$) beziehungsweise den des Parent-Links (Argument $isThread = \bot$) zu löschen.

Ist $isThread = \top$ wird die Durchführung der *Remove* Pre-Processing Schritte II und III angestrebt (siehe Zeile $85 - 86$, Abbildung 4.8). Diese Schritte löschen den *Flag* Status. Je nach Zustand des Parent-Links, wird im Falle von $isThread = \bot$ einer der folgenden Fälle ausgeführt:

- Der *Remove* Pre-Processing Schritt V (siehe Zeile 115)
- Der *Remove* Pre-Processing Schritt VI (siehe Zeile 95)
- Die finalen *Remove* Schritte von Kategorie 1-Knoten mit den Schritten XXV-XXVI (siehe Zeilen $100 - 101$)
- Die finalen *Remove* Schritte von Kategorie 2-Knoten mit den Schritten XX-XXIII (siehe Zeile $103 - 106$)

CleanMark

Diese Methode übernimmt zwei *Remove* Pre-Processing Schritte sowie die finalen Remove Schritte von Kategorie 3-Knoten.

Aufrufende *CleanMark* wird von den folgenden Methoden heraus aufgerufen: Von *Add* selbst nach dem Fehlschlag bei der Überprüfung, ob ein Link mit dem *Mark* Indikator versehen wurde (siehe Zeile 179). Von *Locate* in der Zeile 16, wenn bei der Traversierung durch den Baum ein Link mit dem Zustand *Mark* entdeckt wird. Von *TryFlag* in der Zeile 52, wenn die CAS Operation wegen eines *Mark*-Flags fehlgeschlagen ist. Von *CleanFlag* in der Zeile 88 (siehe Abbildung 4.8), nach dem erfolgreichen Abschluss der Schritte II-III des *Remove* Pre-Processings. In Zeile 96 nach dem Schritt VI und vor dem Schritt V aus Zeile 113. Weiters ruft sich *CleanMark* auch selbst auf, nämlich vor Schritt IV aus Zeile 136 wenn der BackOrderNode.RightLink mit dem *Mark* Indikator versehen ist, sowie nach Anstoß eines **Final Remove**s eines Kategorie 3-Knotens aus Zeile $144 - 145$.

Funktionsweise Die Funktion teilt sich nach dem Zustand des übergebenen Arguments **`int`** `markDir`.
Der Bereich markDir=Right ist in Zeile $122 - 140$ zu finden, wobei sich markDir=Left in den Zeilen $141 - 160$ findet.

```
119 void CleanMark(NPtr& curr, int markDir)
120 (left, lF, lM, lT) = curr→child[0];
121 (right, rF, rM, rT) = curr→child[1];
122 if (markDir == 1) then
        /* The node is getting deleted itself.  if it is category 1 or 2 node, flag
        the incoming parent link; if it is a category 3
        node, flag the incoming parent link of its predecessor.  */
123     while true do
124         preNode = delNode→preLink;
125         if (preNode == left) then
126             parent = delNode→backLink;
127             back = parent→backLink;
128             TryFlag(parent, curr, back, true);
129             if (parent→child[pDir].ref == curr) then
130                 CleanFlag(parent, curr, back, true);
131                 break;
132         else
                // Category 3 node.
133             preParent = preNode→backLink;
134             (*, pPF, pPM, pPT) = preParent→child[1];
135             backNode = preParent→backLink;
136             if (pM) then CleanMark(preParent, 1);
137             else if (pF) then
138                 CleanFlag(preParent, preNode, backNode, true);break;
139             else if (CAS(parent→child[pDir], (curr, 0, 0, 0), (curr, 1, 0, 0))) then
140                 CleanFlag(preParent, preNode, backNode, true);break;
141 else
        // The node is getting deleted (†) or moved to replace its successor (‡).
142     if (rM) then
            // (†) clean its left marked link.
143         preNode = curr→preLink;
144         TryMark(preNode, 0);
145         CleanMark(preNode, 0);
146     else if (rt and rF) then
            // (‡) change links accordingly
147         delNode = right;
148         delNodePa = delNode→backLink;
149         preParent = curr→backLink;
150         pDir =cmp(delNode→k, delNodePa→k);
151         (delNodeL, *, *, dlT) = delNode→child[0];
152         (delNodeR, *, *, drT) = delNode→child[1];
153         CAS(preParent→child[1], (curr, f, 0, 0), (left, lF, 0, lT));
154         if (!lT) then CAS(left→backLink, (curr, 0, 0, 0), (preParent, 0, 0, 0));
155         CAS(curr→child[0], (left, 0, 1, lT), (delNodeL, 0, 0, 0));
156         CAS(delNodeL→backLink, (delNode, 0, 0, 0), (curr, 0, 0, 0));
157         CAS(curr→child[1], (right, 1, 0, 1), (delNodeR, 0, 0, drT));
158         if (!drT) then CAS(delNodeR→backLink, (delNode, 0, 0, 0), (curr, 0, 0, 0));
159         CAS(delNodePa→child[pDir], (delNode, 1, 0, 0), (curr, 0, 0, 0));
160         CAS(curr→backLink, (preParent, 0, 0, 0), (delNodePa, 0, 0, 0));
```

Abbildung 4.9: Pseudocode der Methode *CleanMark* [CNT14, S.327]

markDir=Right ist im Falle von Kategorie 1- und 2-Knoten für den Schritt V verantwortlich, was in der Zeile 128 veranlasst wird.
Im Falle eines Kategorie 3-Knotens ruft sich *CleanMark* auch selbst auf, nämlich vor Schritt IV aus Zeile 136, sollte der BackOrderNode.RightLink mit dem *Mark* Indikator versehen sein, um Hilfe beim Beseitigen des *Mark*-Indikators zu leisten. Weiters ist markDir=Right für den Schritt IV des Remove Pre-Processing von Kategorie 3-Knoten verantwortlich (siehe Zeile 139).

markDir=Left ist für die letzten Schritte eines Knotenkategorie 3 *Removes* verantwortlich. Dazu zählt der Schritt VII des Pre-Processings in den Zeilen 144 – 145 und die Schritte X-XVII in den Zeilen 153 – 160 des finalen Removes.

4.3.2 *Contains*-Operation

Contains lässt eine Aussage über die Existenz des Knoten mit dem Index k im BST zu. Die einfache Funktionsweise wird in Abbildung 4.10 illustriert und es sei auf die

```
26 bool Contains(KType k)
   // Initialize the location variables with the global variables.
27 prev = &root[1]; curr = &root[0];
28 dir = Locate(prev, curr, k);
29 if (dir = 2) then return true;
30 else return false;
```

Abbildung 4.10: Pseudocode der Methode *Contains* [CNT14, S.326]

Erläuterung der Hilfsoperation *Locate* in Abschnitt 4.3.1 verwiesen.

4.3.3 *Insert*-Operation

Die *Insert*-Operation fügt einen neuen Knoten mit dem Index k in den BST ein. Zum Einfügen eines neuen Knotens wird ein Speicher für einen neuen Knoten angelegt (Zeilen 163 – 164 aus dem Pseudocode) und es werden seine ausgehenden Links vorbereitet (Zeilen 170 – 172).

Zum Abschluss einer *Add* Operation wird mit einem einzigen CAS ein Link so manipuliert, dass er auf einen neuen Knoten zeigt, wie dies in Abbildung 4.12 zu sehen ist. Dieser Vorgang fügt den neuen Knoten final in den Baum ein.

4.3.4 *Remove*-Operation

Diese Operation gliedert sich immer in zwei Phasen. Während der ersten Phase, dem **Pre-Processing**, werden Linkzustände rund um den zu entfernenden Knoten gesetzt und markieren so eine folgende Manipulation dieser Links. In der zweiten Phase, dem **finalen Remove**, werden die zuvor markierten Links so umgehängt, dass sie nicht mehr auf den zu entfernenden Knoten verweisen.

```
161 bool Add(KType k)
162 prev = &root[1]; curr = &root[0];
    /* Initializing a new node with supplied key and left-link threaded and pointing
    to itself.  */
163 node = new Node(k);
164 node→child[0] = (node, 0, 0, 1);
165 while true do
166     dir = Locate(prev, curr, k);
167     if (dir = 2) then // key exists in the BST
168         return false;
169     else
170         (R, *, *, *) = curr→child[dir];
            /* The located link is threaded.  Set the right-link of the adding node
            to copy this value */
171         node→child[1] = (R, 0, 0, 1);
172         node→backLink = curr;
173         result = CAS(curr→child[dir], (R, 0, 0, 1), (node, 0, 0, 0));
174         if result then return true;
175         else
                /* If the CAS fails, check if the link has been marked, flagged or a
                new node has been inserted.  If marked or
                flagged, first help cleaning.  */
176             (newR, f, m, t) = curr→child[dir];
177             if (newR = R) then
178                 newCurr = prev;
179                 if m then CleanMarked(curr, dir);
180                 else if f then
181                     CleanFlagged(curr, R, prev, true);
182                 curr = newCurr;
183                 prev = newCurr→backLink;
```

Abbildung 4.11: Pseudocode der Methode *Add* [CNT14, S.328]

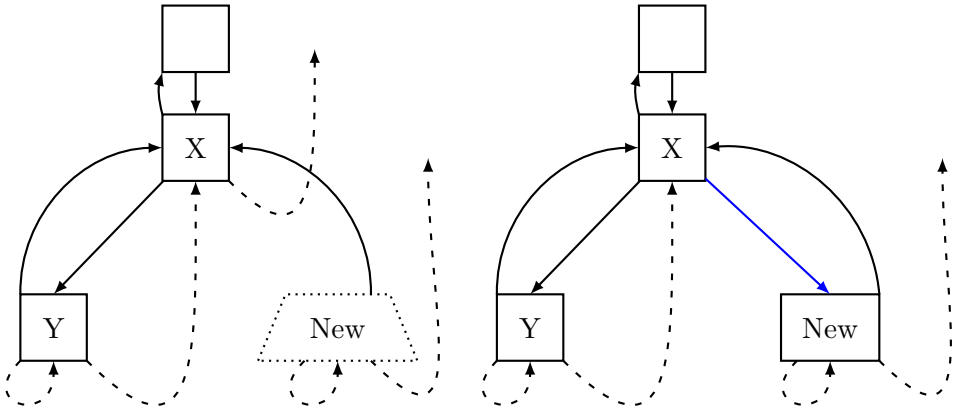
Schritt-für-Schritt Beschreibung

Folgend soll generell und anhand von Beispielen der Schritt-für-Schritt Ablauf der Löschvorgänge nachvollziehbar gemacht werden. Die Schritte sind mit **römischen Zahlen** nummeriert, diese dienen als Referenz für jegliche hier erklärten CAS-Schritte.

In der Tabelle 4.1 werden die notwendigen CAS-Operationen beschrieben, die zur Durchführung des Löschvorgang notwendig sind. Diese werden folgend als Schritte bezeichnet. In der ersten Spalte findet sich die Zugehörigkeit der Schritte zur jeweiligen Operationsart. Diese Zugehörigkeit unterteilt sich in das Pre-Processing, was für alle *Remove*-Operationen notwendig ist, sowie in die Knotenkategorien (siehe Abschnitt 7). In der Spalte *Ziel-Link des CAS* findet sich jeweils der Link, der Ziel der CAS-Operation ist. In der Spalte *Pre-Condition* finden sich Vorbedingungen zur Durchführung der CAS-Operation. In der letzten Spalte *gewünschter Zielwert* finden sich die Werte, die durch ein erfolgreiches CAS gesetzt werden. Im Falle des Misserfolges werden keine Werte manipuliert. Es

Teil von	#	Ziel-Link des CAS	Pre-Condition	Gewünschter Zielwert
Pre-Prozess	I	OrderNode.RightLink		**F**:= $\top$
	II	DelNode.PreLink		Link→OrderNode
	III	DelNode.RightLink		Mark
	IV	BackOrderNode.RightLink	DelNode.Kategorie=3	**F**:= $\top$
	V	BackNode.linkToDelNode		**F**:= $\top$
	VI	DelNode.LeftLink	DelNode.Kategorie=3	**M**:= $\top$
	VII	OrderNode.LeftLink	DelNode.Kategorie=3	**M**:= $\top$
Kateg. 3	X	BackOrderNode.RightLink	$\neg$**M** $\wedge\neg$**T**	Link:=OrderNode.LeftNode **F**:=OrderNode.LeftLink.Flaged **T**:=OrderNode.LeftLink.Threaded
	XI	OrderNode.LeftLink	$\neg$**F** $\wedge$ $\neg$**M** $\wedge$ $\neg$**T**	Link:=BackOrderNode
	XII	OrderNode.LeftLink	$\neg$**F** $\wedge$ **M**	Link:=LeftNode **M** := $\bot$, **T** := $\bot$
	XIII	LeftNode.BackLink	$\neg$**F** $\wedge$ $\neg$**M** $\wedge$ $\neg$**T**	Link:=OrderNode
	XIV	OrderNode.RightLink	**F** $\wedge$ $\neg$**M** $\wedge$ **T**	Link:=RightNode, **F** := $\bot$ **T** :=DelNode.RightLink.Threaded
	XV	RightNode.BackLink	$\neg$**F** $\wedge$ $\neg$**M** $\wedge$ $\neg$**T** Indirekt:$\neg$delNode.RightLink.**T**	Link:=RightNode
	XVI	BackNode.linkToDelNode	**F** $\wedge$ $\neg$**M** $\wedge$ $\neg$**T**	Link:=OrderNode, F:= $\bot$
	XVII	OrderNode.BackLink	$\neg$**F** $\wedge$ $\neg$**M** $\wedge$ $\neg$**T**	Link:=BackDelNode
Kateg. 2	XX	OrderNode.RightLink	**F** $\wedge$ $\neg$**M** $\wedge$ **T**	Link→DelNode.RightChild, $F := \bot$ T :=DelNode.RightLink.Threaded
	XXI	RightChild.BackLink	Indirekt:$\neg$delNode.RightLink.**T**	Link→OrderNode
	XXII	BackNode.linkToDelNode	**F** $\wedge$ $\neg$**M** $\wedge$ $\neg$**T**	Link→OrderNode, $F := \bot$ T :=DelNode.LeftLink.Threaded
	XXIII	OrderNode.BackLink		Link→BackNode
Kateg. 1	XXV	BackNode.linkToDelNode	**F** $\wedge$ $\neg$**M** $\wedge$ $\neg$**T**	Link→DelNode.RightChild, **F** := $\bot$ T :=DelNode.RightLink.Threaded
	XXVI	RightChild.BackLink	Indirekt:$\neg$delNode.RightLink.**T**	Link→BackDelNode
				F... Flag, **M**... Mark, **T**... Threaded

Tabelle 4.1: *Remove* Prozedur Schritt für Schritt, I-VII stellen das Pre-Processing dar, die restlichen Schritte bilden die finalen Remove-Schritte der Knoten der jeweiligen Kategorie



(a) Vor dem einfügen (b) Nach dem Einfügen

Abbildung 4.12: *Insert*-Operation mit nur einem Link-Tausch

wird zwischen den Schritten I-VII, die das Pre-Processing darstellen, und den finalen Manipulationen der Links mit den Schritten X-XVII unterschieden.

Die Knoten in der Tabelle 4.1 beziehen sich auf folgende Knoten im BST: **DelNode** bezeichnet den zu entfernenden Knoten auf den sich der gesamte Löschvorgang bezieht; **OrderNode** bezeichnet den Order-Knoten, dem der *Order-Link* (siehe Abschnitt 4.2.2) entspringt; **BackOrderNode** bezeichnet den Knoten, der über den *Back-Link* (siehe Abschnitt 4.2.2) des *OrderNode*s zu erreichen ist; **BackDelNode** bezeichnet den Knoten, der über den *Back-Link* des *Del-Node*s zu erreichen ist; **RightNode/LeftNode** bezeichnet das rechte beziehungsweise linke Child des *Remove-Node*s.

Die *Remove*-Operation selbst muss je nach Kategorie (siehe Unterabschnitt 4.2.1) des zu entfernenden Knotens nochmals in drei unterschiedliche Operationsarten gegliedert werden. Sie unterscheiden sich durch die Art des Pre-Processings, wobei die nötigen Schritte für das Entfernen von Knoten der Kategorie eins und zwei eine Teilmenge derer von Kategorie drei darstellen. So werden bei Löschvorgängen der Kategorie eins und zwei die Schritte IV, VI und VII nicht benötigt, da der *OrderNode* nicht weiter als einen Link entfernt ist.

Pseudocode der *Remove* Operation

In Zeile 37 der Abbildung 4.13 wird CAS_I angestoßen. Dies ist der einzige Ort im Code von dem ein CAS_I ausgehen kann. Das ist deshalb wichtig, da ein Order-Link nur von diesem CAS aus manipuliert wird, was als Indikator für den Zielknoten des Order-Links gilt, dass dieser entfernt wird.

```
31 bool Remove(KType k)
   // Initialize the location variables as before.
32 prev = &root[1]; curr = &root[0];
33 dir = Locate(prev, curr, k− ϵ);
34 (next, f, ∗, t) = curr→child[dir];
35 if (k ≠ next→k) then return false;
36 else
   |  // flag the order-link
37 |  result = TryFlag(curr, next, prev, true);
38 |  if (prev→child[dir].ref = curr) then
39 |  |  CleanFlag(curr, next, prev, true);
40 return result;
```

Abbildung 4.13: Pseudocode der Methode *Remove* [CNT14, S.326]

Beispiel eines Kategorie 3-Removes

Die unten beschriebenen Vorgänge zeigen das Entfernen eines Kategorie 3-Knotens.

Wie zu Beginn des Kapitels *Remove*-Operation erläutert, teilt sich dieser Vorgang in zwei wesentliche Schritte auf, dem **Pre-Processing** und den Schritten des **finalen Remove**s.

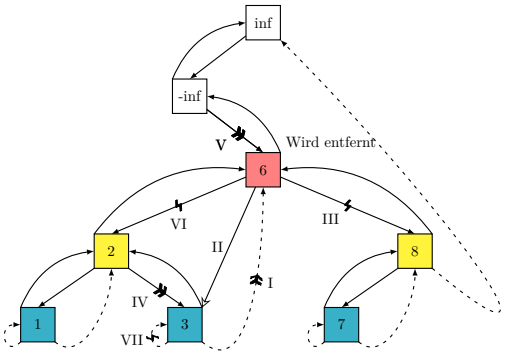
Pre-Processing Im Zuge des Pre-Processings werden die Status-Indikatoren jedes zu verändernden Links manipuliert, sodass ein exklusiver Zugriff für eine Löschoperation gewährleistet werden kann. Eine detaillierte Aufschlüsselung der gesetzten Indikatoren findet sich in der Spalte *gewünschter Zielwert* der Tabelle 4.1. Grafisch sind die notwendigen Schritte I-VII in der Abbildung 4.14a veranschaulicht.

Pre-Processing arbeitet nicht streng sequenziell im Sinne der Schritte I-VII aus Tabelle 4.1, sondern kann auch dazu übergehen, anderen Prozessen bei der Durchführung ihres Löschvorgangs zu helfen. Dabei unterbricht es den Markiervorgang unter Verwendung der Methoden *TryFlag* und *cleanFlag*, um dem konkurrierenden Löschvorgang Hilfestellung zu leisten. Hierfür finden sich Beispiele im folgenden Abschnitt Remove-*Operation im Konfliktfall*.

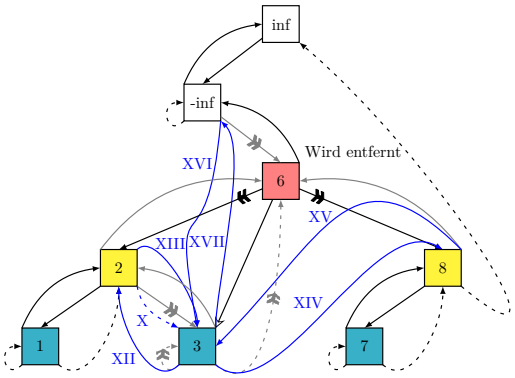
Finales *Remove* Schritte, die nach dem Pre-Processing folgen, bilden den eigentlichen Vorgang des Entfernens. Sie manipulieren die Links im Baum so, dass der zu entfernende Knoten nicht mehr wie zuvor über die Child-Links erreichbar ist. Die notwendigen Schritte für den finalen *Remove* Prozess sind je nach Knotenkategorie unterschiedlich und werden in Tabelle 4.1 abhängig von den Kategorien dargestellt.

Folgend soll beispielhaft aus einem Baum ein Kategorie 3-Knoten entfernt werden, wobei sich die Schritte der römisch markierten Zahlen immer in Tabelle 4.1 (S.52) wieder finden lassen.

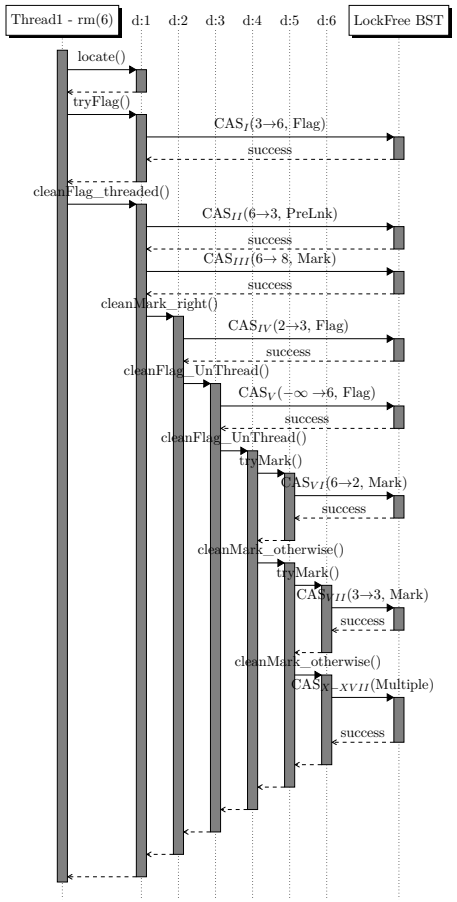
Aus dem in Abbildung 4.14b gezeigten Baum soll ein Kategorie drei Knoten (Farblich rot markiert) mit dem Index $k = 6$ entfernt werden. Zuerst wird in Abbildung 4.14a das Pre-Processing mit den Schritten I-VII durchgeführt. Die Abbildung 4.14b zeigt



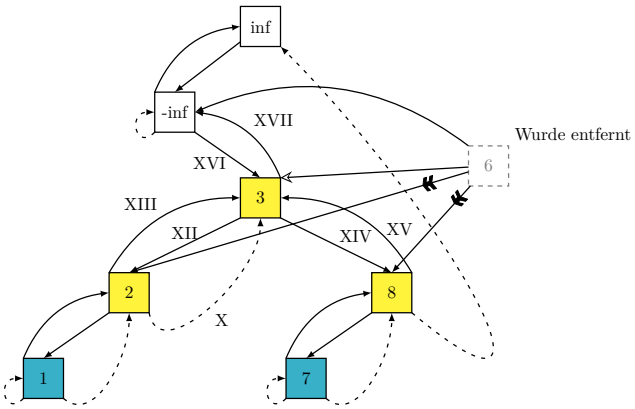
(a) Pre-Prozess I - VII



(b) Finale Remove Schritte X - XVII



(c) *Remove* Prozedur (Call Graph)



(d) Post *Remove*; Knoten 6 wurde entfernt

Abbildung 4.14: Exemplarisches *Remove* eines Kategorie 3 Knotens

anhand von blauen Links X-XVII die Nachfolger der hellgrau unterlegten Links, die für das finale *Remove* eines Kategorie 3-Knotens manipuliert werden. Das Ergebnis der Manipulation stellt Abbildung 4.14d dar, wobei erkennbar ist, dass kein Child-Link mehr auf den entfernten Knoten mit $k = 6$ zeigt.

Remove-Operation im Konfliktfall

Konfliktfälle treten dann auf, wenn Veränderungen der Baumstruktur durch einen anderen Thread B vorgenommen werden und dies eine bestehende Abarbeitung von Thread A, wie von diesem ursprünglich vorgesehen, behindert. Sie äußern sich in fehlschlagenden CAS-Operationen.
Im Konfliktfall versucht der unterbrechende Thread B, Thread A zu unterstützen, indem er parallel dieselbe Sequenz an CAS-Operationen durchführt und so versucht, den von Thread B angestrebten Zustand zu erreichen. Sollte dies nicht gelingen, beginnt A von Neuem mit seiner aktuellen Operation. Um nicht den gesamten Baum neu durchsuchen zu müssen, arbeitet sich dieser über die *Back-Links* nach oben zurück und findet so einen validen Einstiegspunkt, der zugesichert noch im Speicher vorhanden ist.

Das Verhalten von *Remove*-Abfolgen im Konfliktfall wird durch das Regelwerk von *LEMMA 10* [CNT14, S.329] festgelegt. Darin ist für alle möglichen Konfliktfälle festgehalten, in welcher Reihenfolge konkurrierende Operationen durchgeführt werden müssen, um valide beendet zu werden. Das geschieht an mehreren Stellen des Pseudocodes [CNT14, S.326-328], nämlich in den Zeilen $14 - 19$, 52, 113 sowie in der Zeile 145 (siehe Abbildung 4.9). Hilfe für andere Löschvorgänge wird parallel geleistet, indem der Thread, der den Hilfsbedarf entdeckt, ebenfalls die finalen *Remove*-Schritte ausführt.

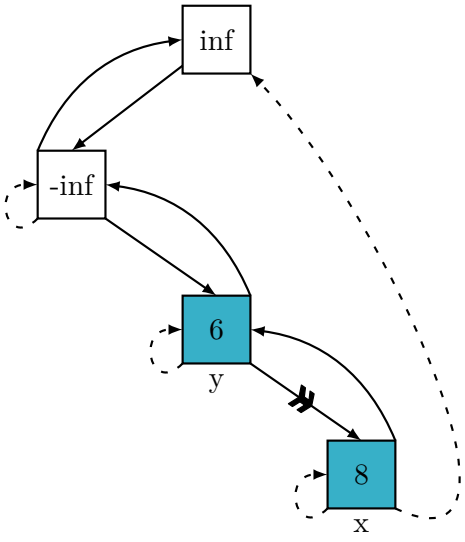


Abbildung 4.15: Baum für das Beispiel *LEMMA 10_a Flagged*

Folgend soll anhand eines Beispiels die parallele Abarbeitung des *LEMMA 10.a Flagged* im Konfliktfall gezeigt werden. Dieses *LEMMA 10_a Flagged* ist einer von vier Fällen, welcher die Abarbeitungsreihenfolge im Konfliktfall definieren. Das *LEMMA* setzt zwei Konten x und y voraus, die parallel durch zwei Threads entfernt werden, wobei x ein Kind von y sein muss und der Link $[y, x]$ ge-*Flagged* sein muss. In diesem Fall muss *Remove*(x) sich vor *Remove*(y) beenden. Sei ein beispielhafter Baum wie in Abbildung 4.15 aufgebaut mit $x == 8$ und $y == 6$ und existiere weiters ein Thread1, der ein *Remove(*$x == 8$*)*, und ein Thread2, der ein *Remove(*$y == 6$*)* ausführt. Um das Szenario aus *LEMMA 10.a Flagged* herbeizuführen, wird Thread1 nach dem *Remove* Schritt V (siehe Tabelle 4.1) unterbrochen, was in Abbildung 4.16 zu sehen ist. In der Grafik sind die *CAS* Schritte mit dem jeweiligen Index aus Tabelle 4.1 versehen, was erkennbar macht, dass Thread1 nach dem CAS_V von Thread2 unterbrochen wird und zu einem Konflikt führt. Thread2 erkennt diesen Konflikt in der Funktion *CleanFlag_unThreaded()* und vollendet den von Thread1 begonnenen *Remove*-Vorgang durch das Ausführen von CAS_{XXV}.

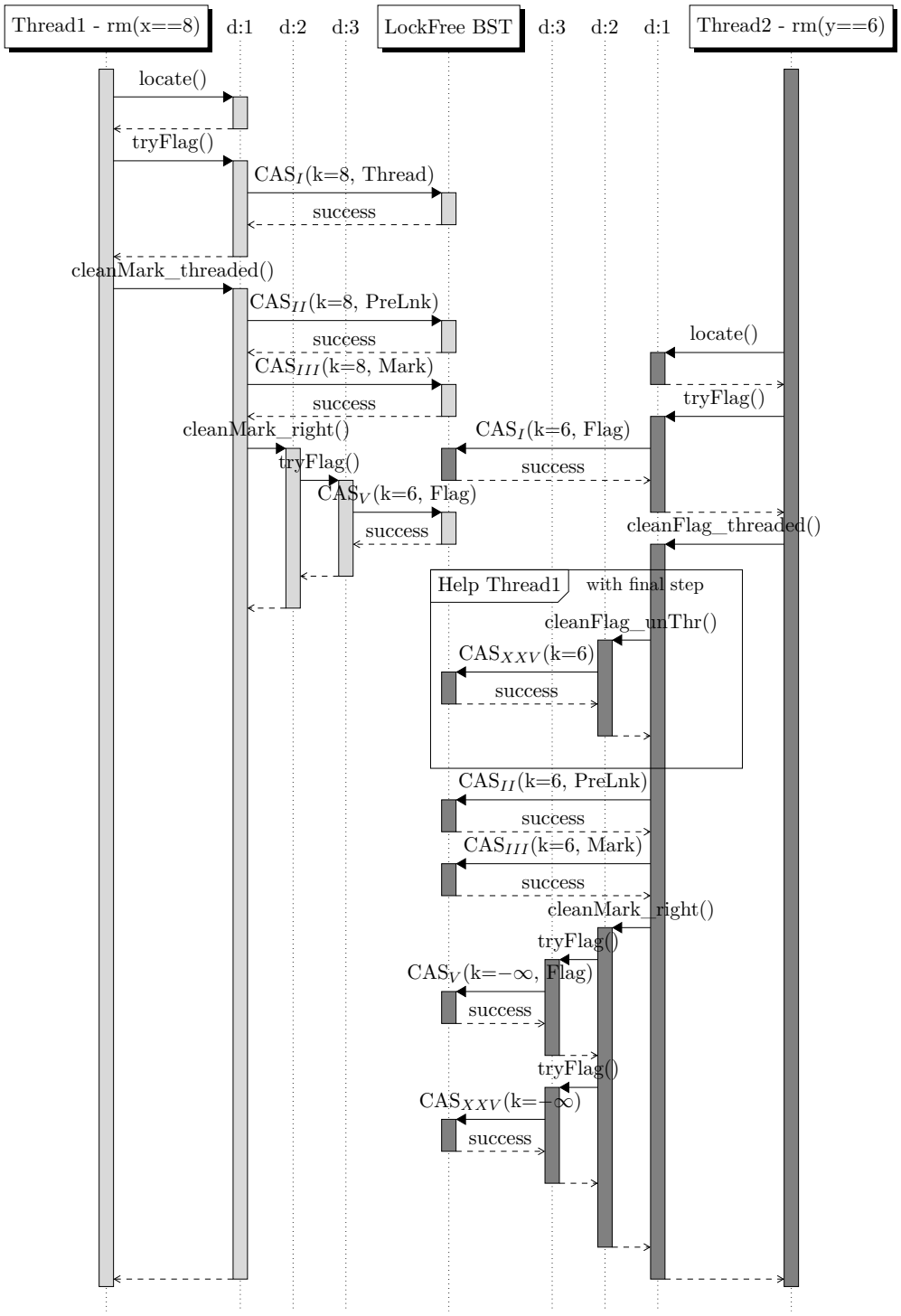


Abbildung 4.16: *Remove* Schritte LEMMA 10a Flagged, mit der Aufruftiefe d und dem Zugriff auf den Speicher des BSTs in der Mitte.

KAPITEL 5

Die Implementierung des Algorithmus

Dieses Kapitel beschreibt Ansätze, die zur Implementierung des BST Algorithmus verwendet wurden. Weiters werden Probleme sowie Auffälligkeiten erläutert, die bei der Untersuchung und Implementierung aufgetreten sind.

Unter der Implementierung versteht sich die Umsetzung des in C++11 programmierten Algorithmus in eine ausführbare Applikation. Dies bedurfte eines ausgiebigen Studiums des vorgestellten Verfahrens und eine Interpretation des Pseudocodes, um ihn in Form von Quellcode in eine Applikation überführen zu können. Dieser Vorgang wird im folgenden Kapitel beschrieben. Weiters werden entwickelte Zusatzfunktionalitäten und Visualisierungskonzepte beschrieben.

Um eine Evaluierung des Algorithmus zu ermöglichen, soll ein Programm erstellt werden, das den Pseudocode [CNT14, S.326-328] der Autoren abbildet und in C++11 [Hal13] umsetzt. Durch Test-Driven-Development mit dem *Google Test Framework*[1] können von Beginn an Zusicherungen überprüft werden und ein Benchmark der implementierten Funktionen erstellt werden. Wenn notwendig, wird zur Fehlersuche auf das *Linux Tracing Tool next generation LTTng*[2] zurückgegriffen, das selbst in einer parallelen Umgebung einen feingranularen Einblick in den Applikationsablauf gewährt.

Folgend sind die Teile beschrieben, in welche sich der Quellcode aufteilt.

- Das **primäre User-Interface** des BST repräsentiert durch die drei Operationen *Insert*, *Remove* und *Contains* (siehe Algorithmus 5.2), die gleichzeitig von mehreren Threads aufrufbar sein sollen.

[1] https://github.com/google/googletest
[2] https://lttng.org

- **BST Hilfsoperationen** namentlich *Locate*, *TryFlag*, *TryMark*, *CleanFlag* und *CleanMark* welche von den Funktionen des primären User Interfaces benötigt werden.

- Die **automatisierte Testumgebung** bestehend aus dem Unit-Test Werkzeug *googletest* [git16b], das Google C++ Testing Framework [Bre15, S.605] sowie *greatest* [git16a], einem C89 kompatiblen, minimalistischen Testing Framework.

- Der **Benchmark-Umgebung**, die in C++ entwickelt wurde, um Ausführungszeiten zu messen.

Gesondert betrachten Abschnitt 5.6 und Abschnitt 5.8 jene Teile der Implementierung, die Testverfahren des BSTs betreffen, beziehungsweise jene Teile, die auf Performance-Messungen eingehen.

5.1 Quellcode und Entwicklungsumgebung

Der gesamte Quellcode wurde in der Hochsprache *C++11* programmiert. Diese Sprache wurde aufgrund der Verwendung spezieller Pointer-Manipulationen (siehe Abschnitt 4.2.2) und wegen der Ähnlichkeit des Pseudocodes mit *C++* gewählt. Zusätzlich wird wegen umfangreicher Unterstützung von Multithreads sowie CAS auf den Dialekt C++11 zurückgegriffen [Hal13, Kapitel 24.3].
Aufgrund der speziellen Verwendung von Pointern im BST (siehe Abschnitt 4.2.2) ist die Ausführung der Implementierung ausschließlich auf 64Bit kompatiblen Architekturen möglich, was auch dem jeweiligen Compiler mitgeteilt werden muss. Dies geschieht für die zwei eingesetzten Compiler folgendermaßen: Mit **GNU Compiler Collection) (GCC)** durch `gcc -std=c++0x -m64` und unter Verwendung von **clang** mit `clang++ -std=c++0x -m64`.

Zur Verwaltung des Quellcodes wurde *CMake* eingesetzt, welches die C++11 Kompatibilität automatisch prüft. Algorithmus 5.1 zeigt die wichtigsten CMake Einstellungen, wie das Setzen der Compiler-Optimierung in Zeile $4-6$, als die Aktivierung der `c++11` Kompatibilität in den Zeilen $8-9$. Weiters wurden die Optionen *INCLUDE_GMOCK* sowie *INCLUDE_LTTNG_UST* (siehe Algorithmus 5.1 Zeilen $11-12$) integriert, um das Kompilieren der *googletest* Umgebung sowie das Linux Trace Toolkit: next generation (LTTng) zu aktivieren (siehe: Abschnitt 5.6). Das optionale Aktivieren wurde zur Aufwandsreduktion eingesetzt, indem nur ein reduziertes Set an Funktionalitäten für alle zu testenden Architekturen zur Verfügung gestellt wurde.

Um den Quellcode zum Kompilieren vorzubereiten, ist `$cmake <pathToCMakeList.txt>` aufzurufen. Anschließend kann mit `$make` kompiliert werden.

Algorithmus 5.1 : Die wichtigsten CMakeLists Einstellungen

```
1  set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall -m64 -g3")
2
3  #compiler optimization
4  if(CMAKE_COMPILER_IS_GNUCXX)
5      set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -O3")
6  endif()
7
8  set_property(TARGET main PROPERTY CXX_STANDARD 11)
9  set_property(TARGET main PROPERTY CXX_STANDARD_REQUIRED ON)
10
11 option(INCLUDE_GMOCK "Include the google mocking library" OFF)
12 option(INCLUDE_LTTNG_UST "Include lttng_ust tracing tools" OFF)
```

5.2 Vorgehensweise: Vom Algorithmus zur Implementierung

Nach der grundlegenden Einarbeitungsphase in das Paper wurde mit dem schrittweisen Aufbau des Quellcodes begonnen. Dazu wurde die Build-Umgebung eingerichtet und das Pointerverhalten auf unterschiedlichen Architekturen erforscht (siehe: Unterabschnitt 5.2.1 und Abschnitt 4.2.2). Daraufhin wurde der Fokus auf die *Insert* und *Contains*-Operationen gelegt, um initial einen Baum erstellen und testen zu können. Diese zwei Operationen wurden als erstes implementiert, da diese ausschließlich von der sekundären Operation (siehe Unterabschnitt 4.3.1) *Locate* abhängen.
Dies markierte auch den Entscheidungszeitpunkt zum Test Driven Development (TDD), da sich die zu überprüfenden Bedingungen mehrten. Zu Beginn wurde mit der minimalistischen Testumgebung *greatest* begonnen, welche später aufgrund weiterer Fähigkeiten des *googletest* Frameworks um dieses erweitert wurde.

5.2.1 Pointer auf unterschiedlichen Architekturen

Untersucht wurde die Verwendbarkeit von drei Bits in den Pointern der jeweiligen Rechner-Architektur, wie dies in Abschnitt 4.2.2 Link-Zustände vorausgesetzt worden war. Dafür wurden drei Architekturen untersucht: Intel Xeon, Intel Core i5 und SPARC T5. Zu beobachten war, dass die beiden Intel Architekturen nicht verwendete Bits mit 0 ausfüllen, wobei die SPARC Architektur Abhängig vom Speicherort unterschiedliche Werte verwendete. Abhängig ob im Stack oder Heap gespeichert wird, padded SPARC mit 0 oder 1. Um dies zu kompensieren wurde zu Beginn einer solchen Pointer-Verwendung ein Reset Value ermittelt (siehe Zeilen 18 – 19 Algorithmus 4.1, Seite 37).

Pointer auf AMD64 basierten CPUs besitzen 48Bit Größe, die im Speicher auf 64Bit abgebildet werden [Tec12, S.120]. Da aktuelle C++11 Compiler auf *AMD64* sowie *x86_64* Prozessoren standardmäßig für 64Bit kompilieren, jedoch nicht immer auf der Solaris

64Bit SPARC, muss der Compiler auf SPARC Architekturen explizit zur Verwendung von 64Bit aufgefordert werden. Dies geschieht mit dem GCC Flag *-m64*.

5.2.2 Implementierte Operationen

Algorithmus 5.2 : Schnittstelle der primären Baumoperationen

```
1 class LockFreeBST {
2 public:
3   bool contains(const IDType k);
4   bool remove(const IDType k);
5   bool add(const IDType k);
6 }
```

Nach dem Einarbeiten wurde die *CAS* Reihenfolge untersucht und ihre Funktionsweise zusammengefasst (siehe Tabelle 4.1). Basierend darauf konnten die primären Baumoperationen, wie sie in Algorithmus 5.2 zu sehen sind, Implementiert werden. Darauf folgend wurden die restlichen zugrundeliegenden sekundären Funktionen (siehe: Unterabschnitt 4.3.1) *CleanFlag*, *CleanMark* und *TryFlag* Implementiert.

5.3 Zusätzlich entwickelte Features

5.3.1 BST Visualisierung

Zusätzlich wurde eine Baum Visualisierung implementiert, die einen im Speicher befindlichen BST ausliest und in das *DOT Format* [GN00] überführt. Das geschieht indem dem Visualisierer ein Pointer zu einem der *Sentinel Knoten* übergeben wird, der daraufhin eigenständig anhand der ausgehenden Links durch den Baum traversiert. Dieses Verfahren ist nicht Thread-sicher und unterstützt daher keine parallele Baummanipulation. Durch die Applikation lässt sich das *DOT Format* in Postscript konvertieren.
Die Abbildung 5.1 zeigt einen so automatisiert erstellte Baumvisualisierung. Blaue Kanten kennzeichnen Back-Links, alle weiteren sind gewöhnliche Child-Links, die ihren Status durch das daneben stehende Quadrupel kennzeichnen. Im Quadrupel befinden sich von links nach rechts folgende Informationen über den jeweiligen Link: {L,R} ob es ein linker oder rechter Link ist, die Stellen zwei bis vier kennzeichnen (durch binäres {0,1}) ob ein Link *Flagged*,*Marked* und/oder *Threaded* ist.

5.4 Behobene Probleme aus dem Paper

Im Zuge der Implementierung des Pseudocodes sind einige Probleme aufgetreten. Die folgend beschriebenen Probleme werden in Bezug auf den Pseudocode angegeben, welcher im *Abschnitt 4.3 Funktionsweise des Algorithmus* zu finden ist.

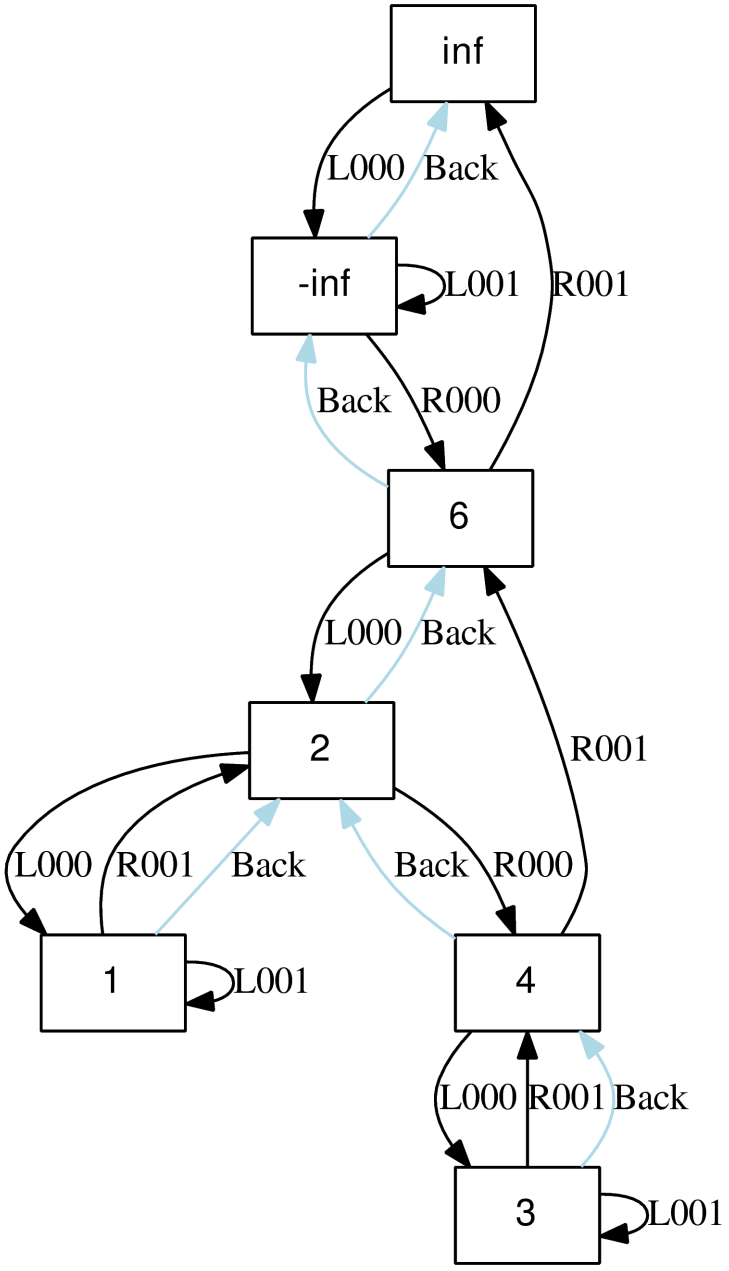


Abbildung 5.1: Automatisch erstellte Grafik eines BST, mit den Knoten {1,2,3,4,6}; sowie den *Sentinel Knoten* -inf & inf

5.4.1 *Locate* Endlosschleife

Mit der im Paper beschriebenen Abbruchbedingung [CNT14, Zeile 23] (siehe Abbildung 4.5, Seite 44) kam es zu Endlosschleifen beim Auffinden von Knoten. Dies lag

Algorithmus 5.3 : Verbesserte Zeilen 21 bis 25 (siehe Abbildung 4.5)

```
1  int k = <id_to_be_located>;
2  if (t) {
3    const int nextE = R->k;
4    const bool isThreadedLeftLink = (dir == cmpRC_e::LESS);
5    if (isThreadedLeftLink or (k < nextE))
6      return dir;
7    // Fehlerhaft urspruenglich hier:
8    // else prev = curr; curr = R;
9  }
10 prev = curr; curr = R;
```

daran, dass die zwei Zeiger *prev* und *curr* nicht manipuliert wurden, da sie falsch verschachtelt waren. Diese vormals falsche Position findet sich in den Zeilen $7-8$ und die funktionsfähige Variante in der Zeile 10 des Algorithmus 5.3.

5.4.2 *Locate* ϵ Logik

Die in Zeile 33 (siehe Abbildung 4.13, Seite 54) verwendete ϵ-Offset Logik wird beim Aufruf der *Locate*-Operation verwendet.
Diese Logik soll sicherstellen, dass `Locate(prev, curr, (k-epsilon))` nicht den gesuchten Knoten selbst findet, sondern die Stelle, an der dieser eingefügt werden soll. Wird *Locate* also ein ϵ übergeben, liefert diese Funktion die Knoten vor oder nach dem gesuchten k, je nach Vorzeichen des ϵ. Der zugrunde liegende Pseudocode ist im Paper nicht weiter ausgeführt, sondern wird ausschließlich durch *LEMMA 1* [CNT14, S.328-329] beschrieben. Um diese Funktionalität zu implementieren, wurde von der Knotenidentifikation (beziehungsweise dem *IDType*) ein *EpsilonedIDType* abgeleitet, wie dies Algorithmus 5.4 illustriert.

5.4.3 *CleanFlag* is-Threaded Argument

Die Methode *CleanFlag* (siehe Abschnitt 4.3.1 Seite 46) wird von mehreren Orten aus dem Quellcode heraus aufgerufen. Sie bereinigt einen Link von dem Flag-Indikator (siehe Abschnitt 4.2.2). Welchen Link die Methode bereinigt, ist abhängig von dem übergebenen Argument `bool isThread`. Im Falle von *isThread*$=\top$ wird von einem Threaded Link ausgegangen, wobei es sich um einen Order-Link (siehe Abschnitt 4.2.2) handeln muss (da nur diese Threaded sind), im Falle von *isThread*$=\bot$ wird ein Parent-Link von dem Flag-Indikator befreit.

Im Pseudocode waren mehrere der *isThread* Argumente invertiert gesetzt, was zu einer falschen Abfolge des *Remove* Pre-Processings führte. Deshalb wurden diese invertiert, dazu wurde das fehlerhafte Verhalten des bereits implementierten Sourcecodes mit dem gewünschten Verhalten der Autoren abgeglichen. Die größten Erkenntnisse waren der

Algorithmus 5.4 : Knoten ID Implementierung und deren ϵ Implementierung

```
1  class IDType {
2    int _value; //!< the held value
3  public:
4    IDType(const int value);
5    /** paper conform compare operation
6    * @return {0,1,2} depending on whether lhs is less than,
7    *              greater than or equal to rhs       */
8    int cmp(const IDType& rhs) const;
9    friend int cmp(const IDType& lhs, const IDType& rhs);
10 };
11
12 enum epsilon_t { NEGATIVE = -1, ZERO = 0, POSITIVE = 1 };
13 class EpsilonedIDType : public IDType {
14   const epsilon_t _e; //!< offset epsilon
15 public:
16   EpsilonedIDType(int value, epsilon_t epsilon = ZERO);
17   friend cmpRC_e cmp(const EpsilonedIDType& lhs, const IDType& rhs);
18 };
```

Aufschlüsselung der *Remove* Prozedur Schritt für Schritt (siehe Tabelle 4.1) zu verdanken, da dadurch kontrolliert werden konnte, ob diese Schritte im Sourcecode auch tatsächlich erreicht wurden.

5.4.4 Flag vs Mark Definition

Der Unterschied zwischen den Link-Zuständen *Flag* und *Mark* (siehe Abschnitt 4.2.2), die beide für die *Remove* Operation notwendigen sind, wird nicht deutlich definiert. Dies soll mit dem *Abschnitt 4.2.2 Link-Zustände* hier nachgeholt werden. Ihnen kommt dieselbe Aufgabe zu, nämlich die Kennzeichnung, dass sich ein Link in einer Kette von Manipulationen befindet (siehe Unterabschnitt 4.3.4). Die Unterscheidbarkeit zwischen zwei Zuständen *Flag* und *Mark* kommt ausschließlich dem Ablauf des Algorithmus zugute.

5.5 Offene Probleme

Dieses Kapitel erläutert Probleme, die gefunden wurden, jedoch nicht behoben wurden oder nicht behoben werden konnten.

Das größte ungeklärte Problem ist, dass keine Aussage darüber getroffen werden kann, ob die bestehenden Fehler vom Pseudocode selbst ausgehen oder ob diese der Implementierung (zum Beispiel Folgefehlern) entspringen. Das ist der Fall da a) der rekursive Aufbau den Pseudocode unübersichtlich werden lässt b) zur initialen Funktion vom Pseudocode abgewichen werden, musste sodass c) die negativen Wechselwirkungen dieser Änderungen nicht mehr von den Fehlern aus dem Pseudocode unterschieden werden konnten.

Dieser Ansatz war nötig, da nach initialer Implementierung des Pseudocodes an verschiedenen Stellen Endlosschleifen entdeckt wurden. Ein daraus folgendes Symptom war, dass nach erfolgter Problembehebung an einer Stelle, ein neues Problem an einer anderen Stelle auftrat.

Abhilfe geschaffen hätte eine Implementierung, die sich ausschließlich an die formalen Regeln hält, die im Werk von Chatterjee et al. beschrieben wurden, sich jedoch nicht auf den Pseudocode stützt. Dies hatte aufgrund des zusätzlich notwendigen Arbeitsaufwands den Rahmen dieser Arbeit überschritten.

5.5.1 Unbehandelter Zustand des Baumes

Die *TryFlag* Funktion endet in einer Endlosschleife mit der im Paper beschriebenen Methode. Hintergrund ist die fehlerhafte Abfolge: (thr:0) rm Schritt (V) -> (thr:1) CLRMARK_O_XII -> (thr:1) TryMark via cleanMark-Codezeile 144 setzt das *Mark* Flag auf den bereits frisch umgehängten Pointer. Das manifestiert sich in einer Endlosschleife der Funktion *TryFlag* mit dem Aufruf der Unterfunktion *CleanMark* in Codezeile 52, mit $markDir = 0$. Dies beruht auf der Tatsache, dass für die Methode *CleanMark* im Falle von *markDir==0, lM==1, rM==0* keine Bearbeitung vorgesehen ist (siehe Codezeile 142-145 [CNT14, p.327]. Gelöst werden könnte dies durch eine zusätzliche Absicherung, dass Schritt (V) nicht durch CLRMARK_O_XII unterbrochen wird.

5.5.2 Speicherbereinigung

Die betrachtete Datenstruktur geht nicht auf das Problem der Speicherbereinigung ein (siehe Abschnitt 2.2.3). So werden nach der Verwendung von Knoten keinerlei Maßnahmen getroffen, um den benötigten Speicher wieder freizugeben. Diese Tatsache spielt in einer Umgebung mit *Garbage Collector* keine Rolle, führt aber anderenfalls zu einem *Memory Leak*, was in dieser Implementierung auch der Fall ist.

Bronson et al. [BCCO10] verwenden eine Technik der Speicher-Wiederverwendung, bei der die Autoren gelöschte Knoten markieren, um diese Knoten beziehungsweise denselben Speicher bei einem zukünftigen *Insert* wieder zu verwenden [BER14, S.331].

Crain, Gramoli und Raynal verwenden Lock-basierende Software Transactional Memory (STM) [CGR12] und Locks [CGR13], bei denen der gesamte Löschvorgang durch Markieren der Knoten stattfindet. Indes übernimmt das physisches Entfernen der Knoten ein separater Thread, was zu einer temporären aber starken Entartung der Baumstruktur führen kann [BER14, S.331], jedoch zu keinem Memory Leak führt.

Vergleich zwischen den Programmiersprachen *Java* und *C*

Wenngleich die *C* Implementationen Bits in den verwendeten Pointern benützen (siehe Abschnitt 4.2.2) um Markierungen zu speichern, benötigen *Java* Implementationen basierend auf dem Pendant *AtomicMarkableReference* einen extra Lesebefehl um an dieselben Markierungen zu gelangen [Gra15, S.5].

Java hingegen besitzt einen Garbage Collector, was die Speicherbereinigung nach dem Löschen eines Knotens im Baum für den Softwareentwickler vereinfacht. Auf die Speicherfreigabe gehen die Autoren Chatterjee et al. nicht näher ein.

5.6 Tests und Validierung

Das folgende Kapitel geht auf Methoden ein, die während der Entwicklung verwendet wurden, um den Quellcode zu testen sowie zu validieren. Darunter fallen zwei **automatisierte Test**-Umgebungen, ein **Trace Level Debugging** Tool, sowie ein eigens entwickelter **CAS Fehler Injektor**.

5.7 Automatisierte Tests

Wie schon in Abschnitt 5.2 begründet, wurde nach dem Programmierkonzept des Test Driven Development (TDD) entwickelt, wo Testfälle vor dem Produktivcode erstellt werden [Lat14, S.381]. Dies hat sich in der fortschreitenden Entwicklung als besonders nützlich erwiesen, da die implementierten Errungenschaften jederzeit automatisiert kontrolliert werden konnten. Zu diesem Zweck wurden zwei Unit-Testing-Frameworks eingesetzt:

Algorithmus 5.5 : Mocking CAS Wrapper

```
1  #include ...
2  #include "gmock/gmock.h"
3  class MockedLockFreeBST : public LockFreeBST {
4  // create a mock-method called casMock_ANodePtr
5  MOCK_CONST_METHOD3(casMock_ANodePtr, void(
6       ANodePtr casObject, ANodePtr expected, ANodePtr desired));
7
8  // create a mocking wrapper around the cas
9  virtual bool cas(std::atomic<ANodePtr>& casObject,
10       ANodePtr expected, ANodePtr desired) {
11      // call the mock
12      casMock_ANodePtr(originalObject, expected, desired);
13      // call the actual cas in the super class
14      bool rc = LockFreeBST::cas(casObject, expected, desired);
15      return rc;
16 }};
```

greatest ein minimalistisches Unit-Test System [git16a], das wegen seiner geringen Anforderungen (C-Standard C89 kompatibel) ohne Änderungen auf den Testsystemen eingesetzt wurde. Die auf *greatest* basierenden Unit-Tests bilden das Basisset an Tests, welche über die Applikation *main* aufrufbar sind.

googletest ist das von Google gewartete und eingesetzte C++ Unit-Testing-Framework [git16b][Bre15, S.605], welches umfangreiche Validierungs- und Testmöglichkeiten bietet. Weiters inkludiert es ein Objekt-Mock Framwork, eine gut bekannte Technik, um die Teile eines Programms, die in der Testsituation irrelevant sind, zu substituieren [TS06, S.1]. Diese Technik wurde verwendet, um Zusagen über die Ausführungsreihenfolgen treffen und kontrollieren zu können. Dazu wurde ein Mocking Wrapper um die eigentlichen CAS-Operationen entwickelt, wie er in Algorithmus 5.5 zu sehen ist.

5.7.1 Trace Level Debugging

Die Tatsache, dass parallele Programme potentiell schwer zu debuggen sind, hat mehrere Ursachen. Zwei davon sind, dass erstens Bugs in parallelen Umgebungen oftmals schwer zu reproduzieren sind und zweitens, dass Bugs von minimalen Laufzeitunterschieden abhängen [JSRV, S.57]. So manipuliert ein Debugger generell und auch der eingesetzte GNU Debugger (GDB) den eigentlichen Programmablauf. In der parallelen Entwicklung führte dies dazu, dass Fehlersymptome wie Race-Conditions, welche ohne Einsatz des Debuggers in wenigen Sekunden reproduzierbar waren, nach der Aktivierung von GDB nicht mehr auftraten.
Dieser Umstand führte zum Einsatz eines *tracing* Tools, welches nur minimal auf den Programmablauf einwirken soll. Wegen des Open-Source Charakters fiel die Wahl auf den Tracer Linux Trace Toolkit: next generation (LTTng)[3].

LTTng gewährt die Möglichkeit Thread-Safe zu loggen, sowie Daten des Programmablaufes inklusive Zeitstempel abzuspeichern. Die folgenden LTTng Interfaces wurden zum Loggen verwendet:

tracepoint(..) , welches sich ähnlich wie `printf(..)` verhält, jedoch fix definierte Argumente besitzt. Die Typen und Anzahl der Argumente werden via des Macros `TRACEPOINT_EVENT(..)` fix definiert (siehe Algorithmus 5.6).

tracef(..) , welches sich genau gleich wie `printf(..)` verwenden lässt, besitzt keine fixierten Werte und beeinflusst den Programmablauf mehr als `tracepoint(..)`.

Während des Programmablaufes können die beiden Interfaces aufgerufen werden und deren Ergebnisse werden im Speicher von LTTng abgelegt.

LTTng muss vor Ablauf des zu loggenden Programmes gestartet und danach wieder gestoppt werden. Dafür hat sich eine Resource Acquisition Is Initialization (RAII) (siehe Algorithmus 5.7) Implementierung bewährt. Diese automatisierende Klasse aktiviert LTTng bei Instanzierung und stoppt LTTng im Destruktor wieder, wie dies Algorithmus 5.7 in Zeile fünf illustriert.

[3]Siehe: `https://lttng.org/` (Lizenzen LGPLv2.1 und GPLv2)

Algorithmus 5.6 : LTTng Initialisierung

```
1  #undef  TRACEPOINT_PROVIDER
2  #define TRACEPOINT_PROVIDER tpProvider
3
4  #if !defined(_HELLO_TP_H) || defined(TRACEPOINT_HEADER_MULTI_READ)
5  #define _HELLO_TP_H
6  #include <lttng/tracepoint.h>
7  TRACEPOINT_EVENT(
8  tpProvider, //!< tracepoint provider name (from above)
9  casLog,     //!< tracepoint/event name
10 TP_ARGS(    //!<  list of tracepoint arguments
11         const char*, position,
12         const int, rc,
13         const int, threadAlias,
14         const int, k,
15         const char*, casData
16 ),
17 TP_FIELDS(
18         ctf_string(position, position)
19         ctf_integer(int, rc, rc)
20         ctf_integer(int, threadAlias, threadAlias)
21         ctf_integer(int, k, k)
22         ctf_string(casData, casData)
23 ))
24 #endif /* _HELLO_TP_H */
25
26 #undef TRACEPOINT_INCLUDE
27 #define TRACEPOINT_INCLUDE "hello_lttng.h"
28 #include <lttng/tracepoint-event.h>
```

Abschließend wurde die *python*-Bibliothek *babeltrace* verwendet, welche mit dem Interface *babeltrace.TraceCollection()* die Möglichkeit zur individuellen Auswertung der geloggten Daten ermöglicht.

5.7.2 CAS Fehler Injektion

Zur Analyse der Funktionalität der Baum-Operationen wurde ein *Fault Injector* implementiert. Dieser besitzt die Möglichkeit, *Race Condition*-spezifische CAS-Operationen herbeizuführen. Dazu werden die betreffenden CAS Operationen anhand einer ihnen fix beigefügten ID und einer Thread ID ausgewählt. Durch Injektion von Delays können in einer vorher bekannten Sequenz von Operationen Fehlschläge einzelner Operationen forciert werden.

Sei *Thread 1* mit CAS_N die zu störende CAS Operation, mit $t[ms]$ als Basiszeit für die eingesetzten Delays und beliebig vielen anderen Threads subsumiert als *Thread M*.

Algorithmus 5.7 : LTTng RAII Implementierung

```
1  #include <lttng/tracef.h>
2  #include <vector>
3
4  // @usage simply define the following variable in your
5  // code:    auto setupLTTNG = LttngUST_RAII();
6  class LttngUST_RAII {
7      const std::string _sessionName = "myConsoleSession";
8      const std::string _tracePointName = "tpProvider";
9  public:
10 LttngUST_RAII() {
11   std::vector <std::string> initSequence {
12     "lttng destroy " + _sessionName,
13     "lttng create " + _sessionName,
14     "lttng enable-event --userspace --tracepoint " +
15           _tracePointName + ":*",
16     //register tracef listener
17     "lttng enable-event --userspace \
18           --tracepoint lttng_ust_tracef:*",
19     "lttng add-context --userspace --type pthread_id",
20     "lttng start"
21   };
22   // Start LTTng
23   for (auto each: initSequence) {
24       std::system(each.c_str());
25   }
26 }
27 LttngUST_RAII::~LttngUST_RAII() {
28   // Stop LTTng
29   std::system(std::string("lttng stop "
30       + _sessionName).c_str());
31 }
32 };
```

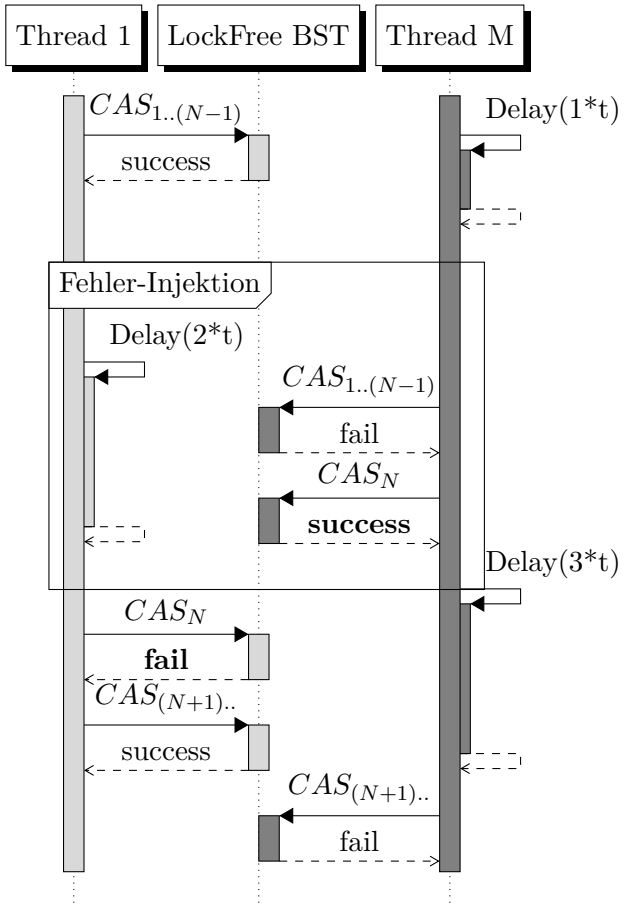


Abbildung 5.2: Konzept der Fehler-Injektion

Abbildung 5.2 illustriert einen möglichen Ablauf einer Fehler-Injektion, wobei folgende Regeln gelten:

CAS Sequenz Die CAS Operationssequenz S aller Prozesse/Theads muss denselben Ablauf besitzen. Dies wird durch den Einsatz eines Testing-Frameworks und die Delegation derselben Befehle an mehrere Threads ermöglicht.

$t_\Delta \leq t$ Das zeitliche Delta der CAS Operationen zwischen den Threads t_Δ muss kleiner t sein.

$t_{CAS} \leq t$ Die Zeit, die von einer konfliktfreien CAS Operation benötigt wird, muss kleiner t sein.

$Thread_M$ **pre delay** Alle Threads außer $Thread_1$ werden zu Beginn um die Zeit $1 * t$ verzögert.

T_1 **pre delay** Thread T_1 wird vor CAS_N um die Zeit $2 * t$ verzögert.

$Thread_M$ **post delay** Alle Threads außer $Thread_1$ werden nach ihrem Ausführen von CAS_N um $3 * t$ verzögert.

Algorithmus 5.8 : Interface des Fault-Injectors

```
1  class FaultInjector {
2  public:
3  void enable();
4
5  // injects the fault production if enabled. To be called
6  //  from every CAS Operation
7  void handle(const FaultPos& pos) const;
8
9  // stores an alias mapped to a thread::id
10 void rememberAlias(const thread::id& threadID, int alias);
11
12 // add's a fault condition/filter to the matching list, to
13 //  react on if matched during a handle(..) call
14 void addFaultCase(const FaultCase& faultEntity);
15 }
```

Implementiert wurde das Interface `rememberAlias` um Thread Aliasse hinzuzufügen. Thread Aliasse sind eindeutige Identifier, die Threads bei der Erstellung zugeordnet werden. Speichert man diese Identifier gemeinsam mit den `std::thread::ids`, erfüllen diese den Zweck, Threads in einer menschlich lesbaren Form wieder erkennen zu können, was vom Fault-Injector verwendet wird. Weiters wurde das Interface `addFaultCase` Implementiert, welches die Möglichkeit bietet Fehlerfälle hinzuzufügen. Ein eigenes Interface unterstützt dann automatisiertes Fehler-Injizieren vor und nach CAS Operatoren (siehe Algorithmus 5.8).

Algorithmus 5.9 : CAS Operations Wrapper

```
1  bool LockFreeBSTHelper::cas(
2          CASObject& casObject,
3          const CASValue expected,
4          const CASValue desired,
5          const CASUId& casPos) {
6
7  faultInjector.handle({casPos, CASPrefix::PRE_CAS});
8  bool rc=casObject.compare_exchange_strong(expected, desired);
9  faultInjector.handle({casPos, CASPrefix::POST_CAS});
10 return rc;
11 }
```

Abschließend muss nur noch ein Wrapper um die eigentliche CAS Operation erstellt werden, damit bei jedem Aufruf einer CAS Operation die `FaultInjectors::handle` Methode aufgerufen wird, was wie im Algorithmus 5.9 dargestellt aussehen kann.

5.8 Benchmark

In diesem Kapitel werden die Erkenntnisse der Messungen des implementierten Algorithmus beschrieben. Hierzu wird zuerst auf die **Metriken** des Benchmarks eingegangen und folgend die **Resultate** in Form der Erwartungen und der Messergebnisse festgehalten.

Untersucht wurden die Operationen *Insert* sowie *Contains*, jedoch nicht die Operation *Remove* mangels ihrer vollständigen Implementierung (siehe Abschnitt 5.5).

5.8.1 Metriken und Parameter

Beim Benchmark wurde der Durchsatz[4] von Operationen in einer Zeitspanne mit unterschiedlichen Parametern gemessen.

Zu den Parametern zählen: Die verwendete **Key-Range**, die **Thread Anzahl**, sowie drei unterschiedliche **Computersysteme**, auf denen die Messung durchgeführt wurde.

Key-Range

Als Key-Range wird die Größe des Intervalls verstanden, dass zur Generierung von **Zufallswerten** herangezogen wird. Diese so generierten Zufallswerte wurden den zu untersuchenden Operationen *Insert* und *Contains* zugeführt. Messungen wurden mit zwei unterschiedlichen Key-Ranges durchgeführt, eine kleine mit $[0, 1 * 10^6)$ und eine große mit $[0, 5 * 10^7)$.

Zufallswerte wurden für jede Operation in jedem Thread neu generiert. Dabei wurden dieselben Key-Ranges für alle Threads und Operationen verwendet.

Dieses Konzept wurde gewählt, um einen zufälligen Zugriff auf die Datenstruktur zu simulieren.

Generierung der Zufallswerte wurde mit Hilfe des in C++11 enthaltenen Zufallszahlengenerators *std::uniform_real_distribution* (gleich Verteilung) bewerkstelligt, wie das in Algorithmus 5.10 Zeile 5 zu sehen ist.

Thread Anzahl

Die Wahl der Thread Anzahl, die für das Benchmark herangezogen wurde, richtet sich nach den verwendeten Computersystemen.

[4] Englisch: Throughput

Algorithmus 5.10 : C++11 Random Number Generator

```
1 #include <random>
2 #include <cmath>
3 random_device rd;
4 mt19937 mt(rd());
5 uniform_real_distribution<> dist(1., (double)<keyRange>);
6
7 const double rndm_dbl_value = dist(mt);
8 const int    rndm_int_value = (int)round(rndm_dbl_value);
```

Computersysteme

Auf diesen drei Computersystemen (folglich auch Maschinen genannt) mit den teilweise unterschiedlichen Prozessor-Architekturen wurde gemessen:

- Mars
  8 Intel Xeon E7-8850 CPUs mit 10 Cores @ 2GHz
  160 Threads, Westmere microarchitecture
  1 TiB RAM
  Debian GNU/Linux 3.14-1

- Ceres
  4 SPARC T5 CPUs mit 16 Cores @ 3.6GHz
  512 Hardware Threads
  1 TiB RAM
  Oracle Solaris 11

- Home
  1 Intel Core i5 750 mit 4 Cores @ 2.67 GHz 8 Threads, Nehalem Microarchitecture
  Ubuntu GNU/Linux 16.04 x86_64

Eine Thread Range von $[10, 150]$ wurde gewählt, da die verwendeten Computersysteme *Mars* und *Ceres* jeweils mindestens so viele Threads unterstützen.

Die Hardware des Rechensystems *Home* unterstützt hingegen lediglich acht Threads. Dieses Rechensystem wurde trotz der hohen Thread-Überbelegung mit bis zu 150 Threads überbeansprucht, um Vergleichswerte mit den anderen Computersystemen zu erhalten. Bei der Messung auf *Home* unter Verwendung von 150 Threads muss durch die Überbeanspruchung mit grundsätzlichen Performance-Einbußen gerechnet werden.

Obwohl *Ceres* 512 Threads unterstützen würde und eine Messung mit dieser Anzahl möglich wäre, wurde der Maximalwert im Sinne der Vergleichbarkeit mit *Mars* gleich 150 gewählt.

Einstellungen

Folgende Einstellungen wurden für den Benchmark gewählt:

Algorithmus 5.11 : Anhalten aller Threads zu Arbeitsbeginn

```
1  #include <chrono>
2  #include <mutex>
3  #include <condition_variable>
4
5  struct ThreadSharedData {
6    mutex cv_m;              //!< mutex provider
7    condition_variable cv; //!< notify channel
8
9    void golbalReleaseHalt() {
10     cout << "notifying all to start" << endl;
11     cv.notify_all();
12   }
13 };
14
15 void haltAtStart(ThreadSharedData &input) {
16   unique_lock<mutex> lk(input.cv_m);
17
18   //wait until notified (or timeouted)
19   auto timeout = chrono::system_clock::now() + chrono::seconds(10);
20   input.cv.wait_until(lk, timeout);
21 }
```

- *Insert* und *Contains* wurden parallel im Verhältnis 10 : 90 ausgeführt, um Kollisionen zu provozieren und so näher an reale Verhältnisse zu gelangen. Dieses Verhältnis stützt sich auf die Annahme, dass *Contains* Operationen grundsätzlich häufiger als andere Operationsarten auftreten. Wobei diese Werte auch von Gramoli et al. [Gra15, S.8] verwendet wurden.
- Die Messunschärfe wurde reduziert, indem jeweils 20 autonome Messungen pro Messparameter durchgeführt wurden. Alle Threads wurden mit einem Lock für die Dauer der Thread-Erzeugung angehalten, um sie dann gemeinsam starten zu lassen, wie dies in Algorithmus 5.11 zu sehen ist.
- Jedem Thread wurden 10^4 zufällige Werte aus dem Intervall der Key-Ranges (siehe Key-Range in diesem Kapitel) übergeben.

Referenz Messungen Weiters wurden alle Messungen auf einer lock-basierenden Referenz-Implementierung ausgeführt, welche auf ein C++ `std::set` Container baut und selbst entwickelt wurde. Ein `std::set` Container wurde gewählt, da es Elemente ebenfalls sortiert speichert und für gewöhnlich als Red-Black-Tree (eine Spezialform

eines BST, siehe Abschnitt 2.3.3) ausgeführt ist. Damit kommt dieser Container der untersuchten Datenstruktur sehr ähnlich.

Zeitmessung und Auflösung

Zur Messung der Zeit wurde die `std::chrono::high_resolution_clock` aus C++11 verwendet, da sie „the clock with the smallest tick period provided by the implementation“[cpp16a] repräsentiert. Eingesetzt wurde dieser Timer als Start Stop Timer, wie dies im Algorithmus 5.12 gezeigt wird.

Algorithmus 5.12 : C++11 Naiver Start Stop Timer

```
1  #include <chrono>
2  class StartStopTimer { public:
3  chrono::time_point<chrono::high_resolution_clock> _startTime;
4  chrono::nanoseconds duration = chrono::nanoseconds::zero();
5
6  void start() {
7    _startTime = chrono::high_resolution_clock::now();
8  }
9  void stop() {
10   auto tmpTimeNow = chrono::high_resolution_clock::now();
11   duration = (tmpTimeNow - _startTime);
12 } };
```

Der Durchsatz ergibt sich aus der Aufzeichnung aller durchgeführten Operationen über die gemessene Zeit. Diese Werte wurden im Sinne der Vergleichbarkeit durch eine Division der gemessenen Zeit auf einen Zeitraum von einer Sekunde herab skaliert.

Compiler

Alle Tests und Benchmark Programme wurden mit der GNU-Compiler-Collection (Version 5.4.0 auf Mars, 4.8.2 auf Ceres und 5.4.0 auf Home) nach dem C++11 Standard (`-std=c++11`) mit höchstem Optimierungslevel (`-O3`) kompiliert.

5.8.2 Messungen und deren Interpretation

Im Folgenden werden die Messungen anhand der Abbildung 5.7 (Seite 83) erläutert.

Jeweils ausschlaggebend ist der *Durchsatz*, welcher in allen hier folgenden Grafiken gezeigt wird und worauf sich auch die kommenden Analysen beziehen.

Allgemeine Erwartungen

Erwartet wird eine steigende Performance mit steigender Thread-Anzahl, es gilt $S_p(n) > 1$ (aus $S_p(n) = \frac{T^*(n)}{T_p(n)}$, siehe Unterabschnitt 2.2.2 Performance), da eine geringere parallele

Laufzeit $T_p(n)$ als sequentielle Laufzeit $T^*(n)$ erwartet wird. Dies hat mehrere Limits, das entscheidendste trifft auf das *Home* Computersystem zu, welches von der Thread-Anzahl her überbelegt wurde.

Erwartet wurde weiters, dass ein wesentlich höherer Durchsatz bei der *Contains* Operation als bei der *Insert* Operation zu erkennen ist. Dies ist der Fall, da *Insert* zuerst einen *Contains* Check durchführen muss.

Für die Referenzmessung wurde erwartet, dass sie durchwegs schlechtere Ergebnisse erzielt als die Lock-freie Implementierung. Der Grund dafür liegt in der naiven Implementierung der Lock-basierten Datenstruktur, die einen gewöhnlichen `lock_guard<std::mutex>` aus C++11 für jeden Zugriff auf die Datenstruktur verwendet.

Key-Range

In den Messungen sind die unterschiedlichen Key-Ranges für zwischen Abbildung 5.7a und Abbildung 5.7b sowie für zwischen Abbildung 5.7c und Abbildung 5.7d zu sehen.

Erwartet wurde, dass weniger Kollisionen beim *Insert* der größeren $(5 * 10^7)$ Key-Range auftreten. Also von Abbildung 5.7a auf Abbildung 5.7b ein Anstieg der Performance.

Auffällig bei der Generierung war, dass diese auf der SPARC Architektur wesentlich länger dauerte als auf den beiden Intel Architekturen. Dies wurde jedoch nicht näher untersucht, da die Zeitmessung das Generieren der Zufallszahlen nicht inkludiert.

Ergebnis ist, dass die Key-Range in den gemessenen Bereichen keine signifikanten Unterschiede verursacht. Anzufügen ist, dass sich die beiden Messungen wesentlich ähnlicher verhalten haben als dies erwarte wurde (siehe Abbildung 5.7).

Thread Anzahl

Erwartet wurde ein Anstieg des Durchsatzes mit steigender Thread Anzahl. Nicht jedoch auf dem Computersystem *Home*, wegen dessen Thread-Überbelegung, und nicht für die Lock-basierte Referenzmessung.

Das **Ergebnis** zeigt keinen Anstieg im Falle der Lock-freien Implementierung, im Gegenteil fällt diese mit steigender Thread Anzahl ab (siehe Abbildung 5.7). Dies war bei der Lock-basierenden Implementierung und für *Home* zu erwarten, nicht jedoch für die Lock-freie Implementierung.

Computersysteme

Ergebnis war, dass *Home* bei allen Messungen auffällig gut abschnitt, obwohl dieses System eine wesentlich geringere Hardware-Thread Unterstützung besitzt als die anderen Systeme.

Ceres besitzt als einziges System bei der Lock-basierten Referenzimplementierung annähernd gleichen *Insert* und *Contains* Durchsatz. Zwischen *Insert* und *Contains* wurde

jedoch eine große Diskrepanz erwartet (wie dies bei den anderen Architekturen zu sehen ist).

Mars zeichnet sich durch wesentlich geringeren *Insert* und *Contains* Durchsatz der Referenzimplementierung aus, als dies auf allen anderen Architekturen zu beobachten war.

Weitere Ergebnisse

Contains* zu *Insert hat die erwartete Diskrepanz gezeigt. Es ist ein signifikant höherer Durchsatz bei *Contains* als bei *Insert* zu erkennen (siehe Abbildung 5.7). Die Lock-basierte Referenzmessung war auf den Intel Architekturen wie erwartet langsamer, jedoch nicht auf *Ceres* (SPARC).

Messungsergebnisse

Wie folgend werden die Achsen aus den Grafiken Abbildung 5.7 verwendet:

y-Achse zeigt den Durchsatz in Tausend Operationen pro Sekunde $[kOps/s]$ auf einer log_2 Skala.

x-Achse steht für die Anzahl an parallel eingesetzten Threads.

Beide Achsen sind auf allen Grafiken gleich gewählt, um eine bessere Vergleichbarkeit zu erzielen.

Weiters ist anhand der Form der Messpunkte die Datenstruktur (LockFree BST oder Referenz Messung) zu erkennen. Anhand der Farbe der jeweiligen Linie ist das ausführende Computersystem zu erkennen.

Interpretation

Die Implementierung ließ keine Steigerung der Performance (siehe Unterabschnitt 2.2.2) mit steigender Thread Anzahl zu, was ursprünglich erwartet worden war.

Weiters konnte die langsamste der drei Baum-Operationen, namentlich *Remove* (siehe Unterabschnitt 4.3.4) nicht gemessen werden, was einen klaren Einschnitt in der Aussagekraft der Messungen darstellt.

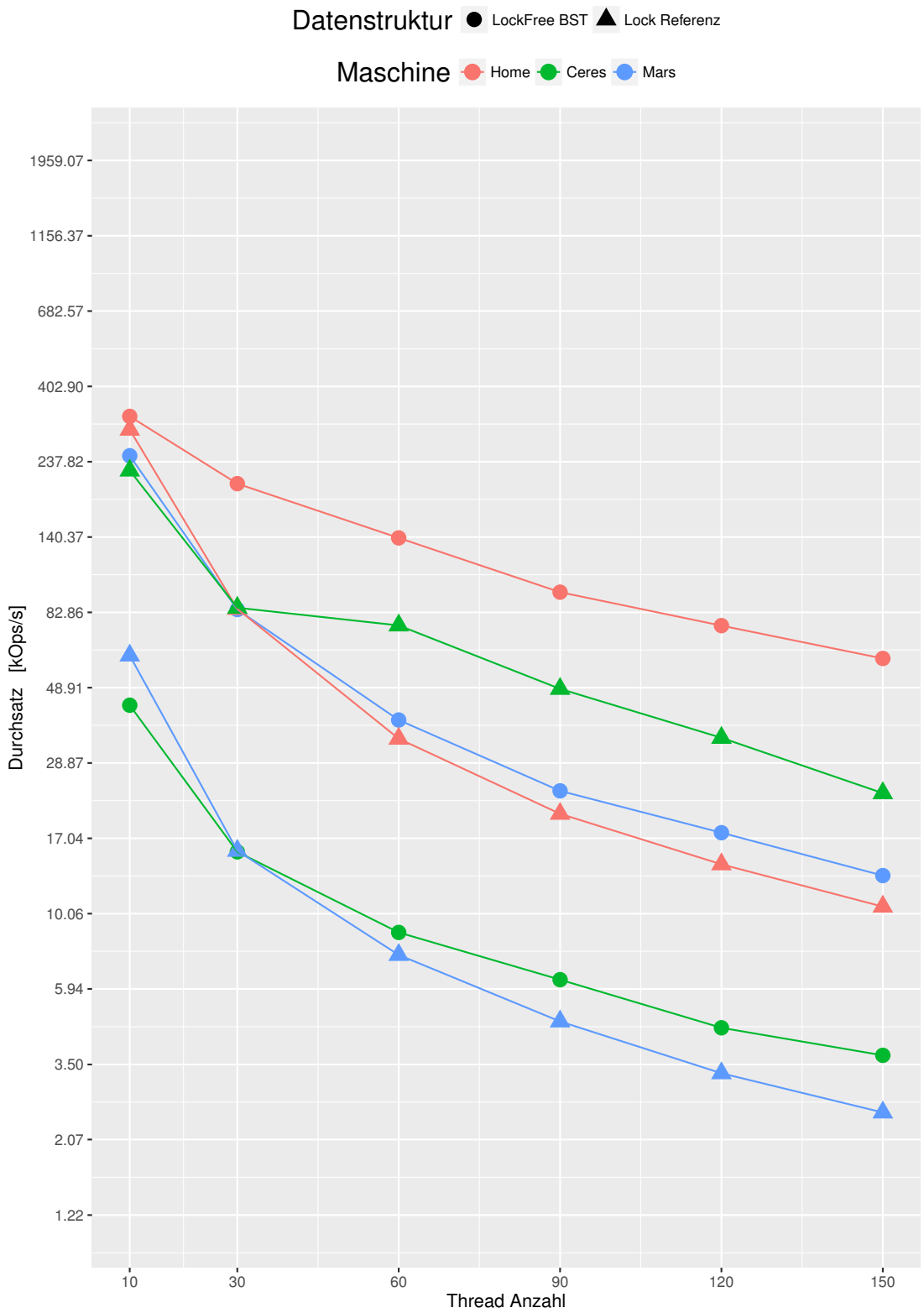


Abbildung 5.3: Benchmark der *Insert* Operation, aus einer Messung mit 10% Update (*Insert*) Last und 90% *Locate*; Key-range gleich $1 * 10^6$

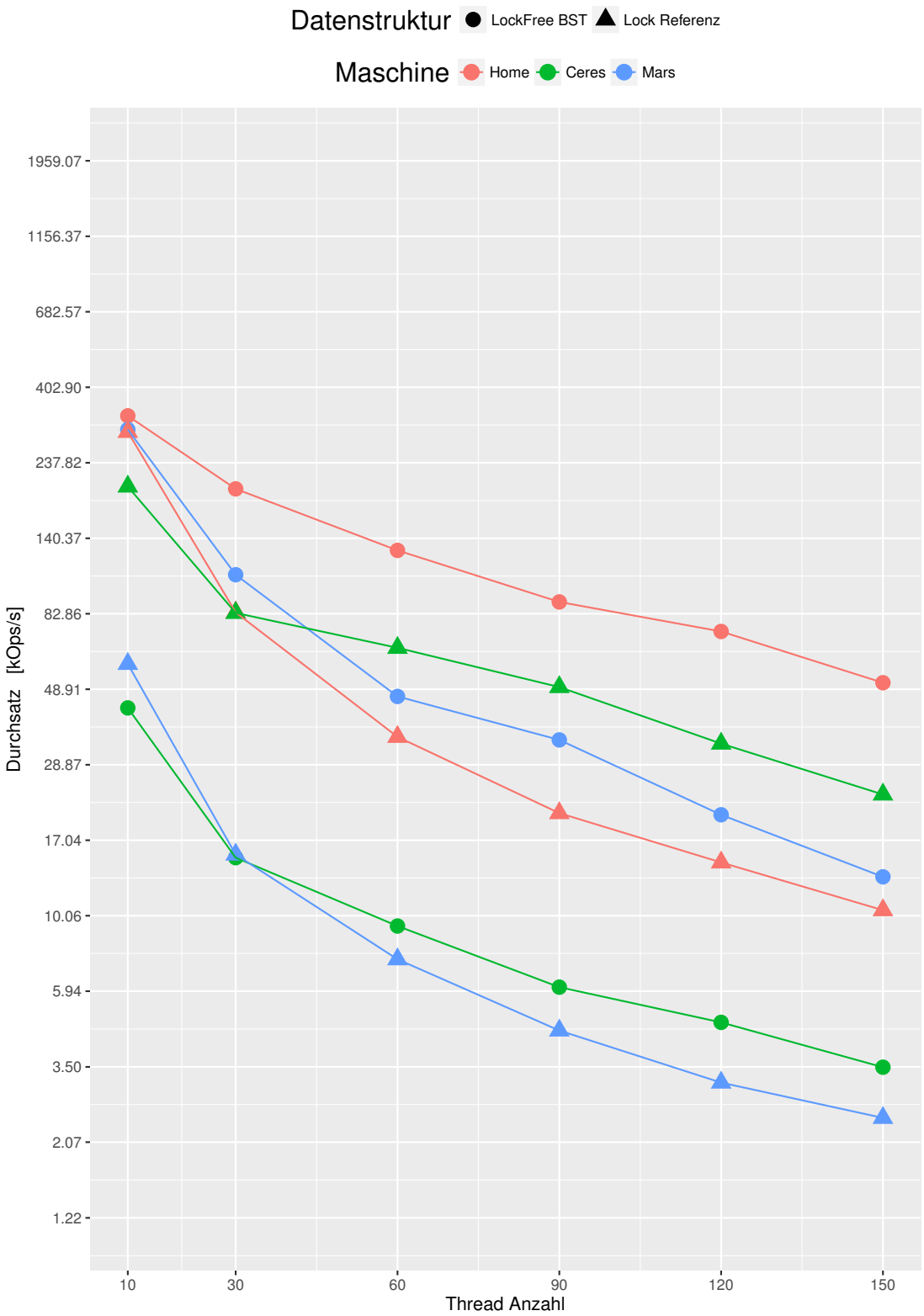


Abbildung 5.4: Benchmark der *Insert* Operation, aus einer Messung mit 10% Update (*Insert*) Last und 90% *Locate*; Key-range gleich $5 * 10^7$

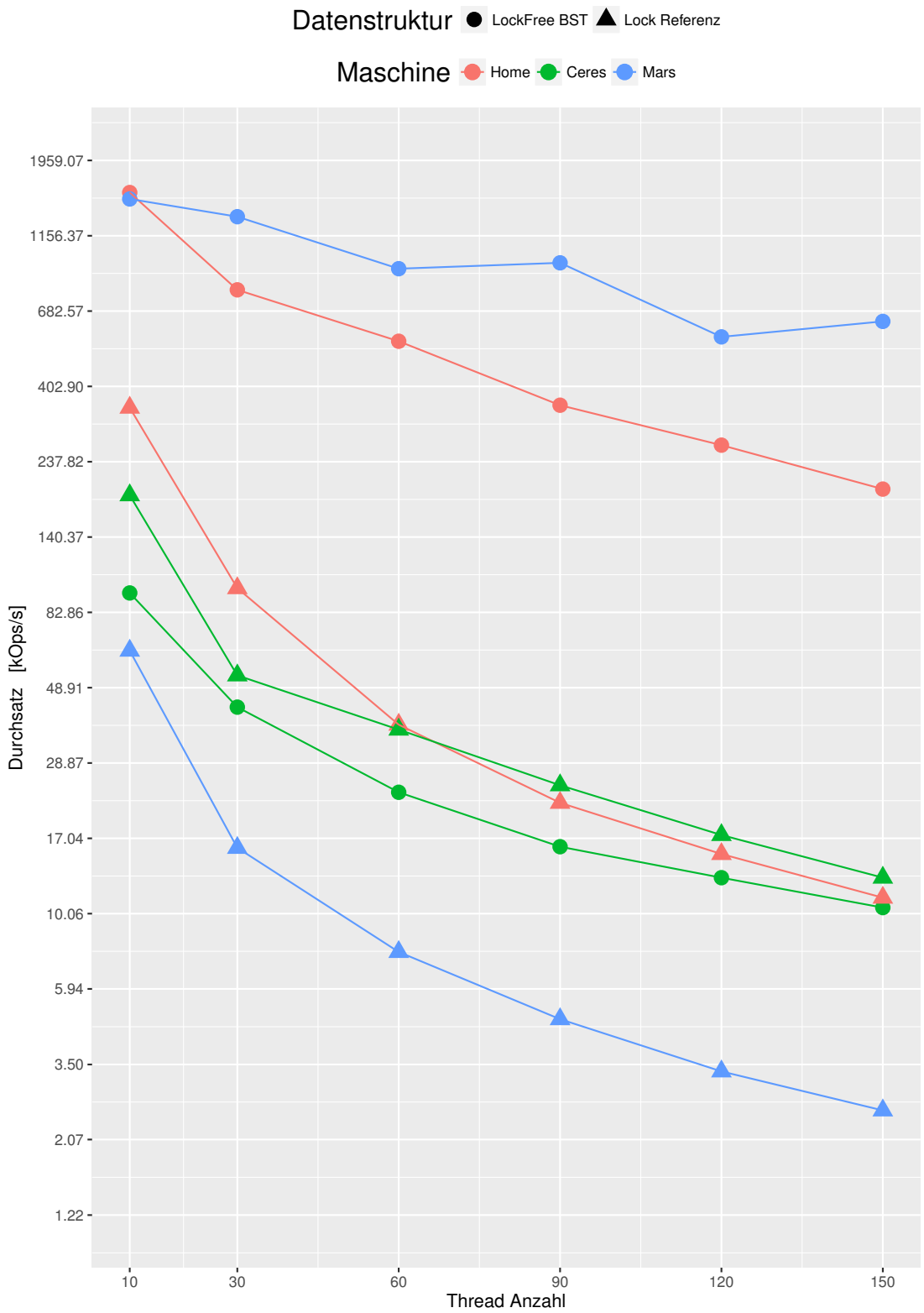


Abbildung 5.5: Benchmark der *Contains* Operation, aus einer Messung mit 10% Update (*Insert*) Last und 90% *Locate*; Key-range gleich $1 * 10^6$

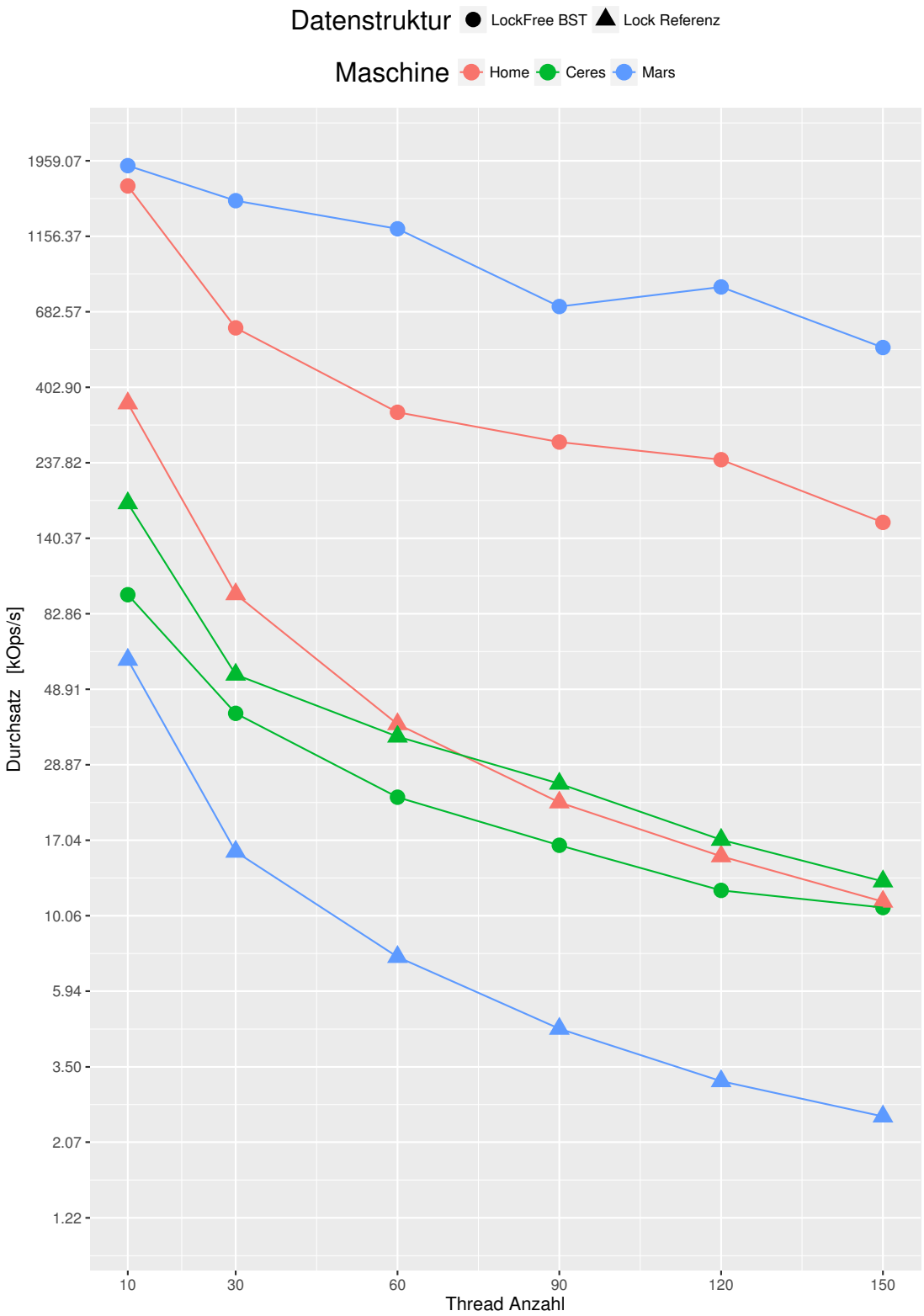
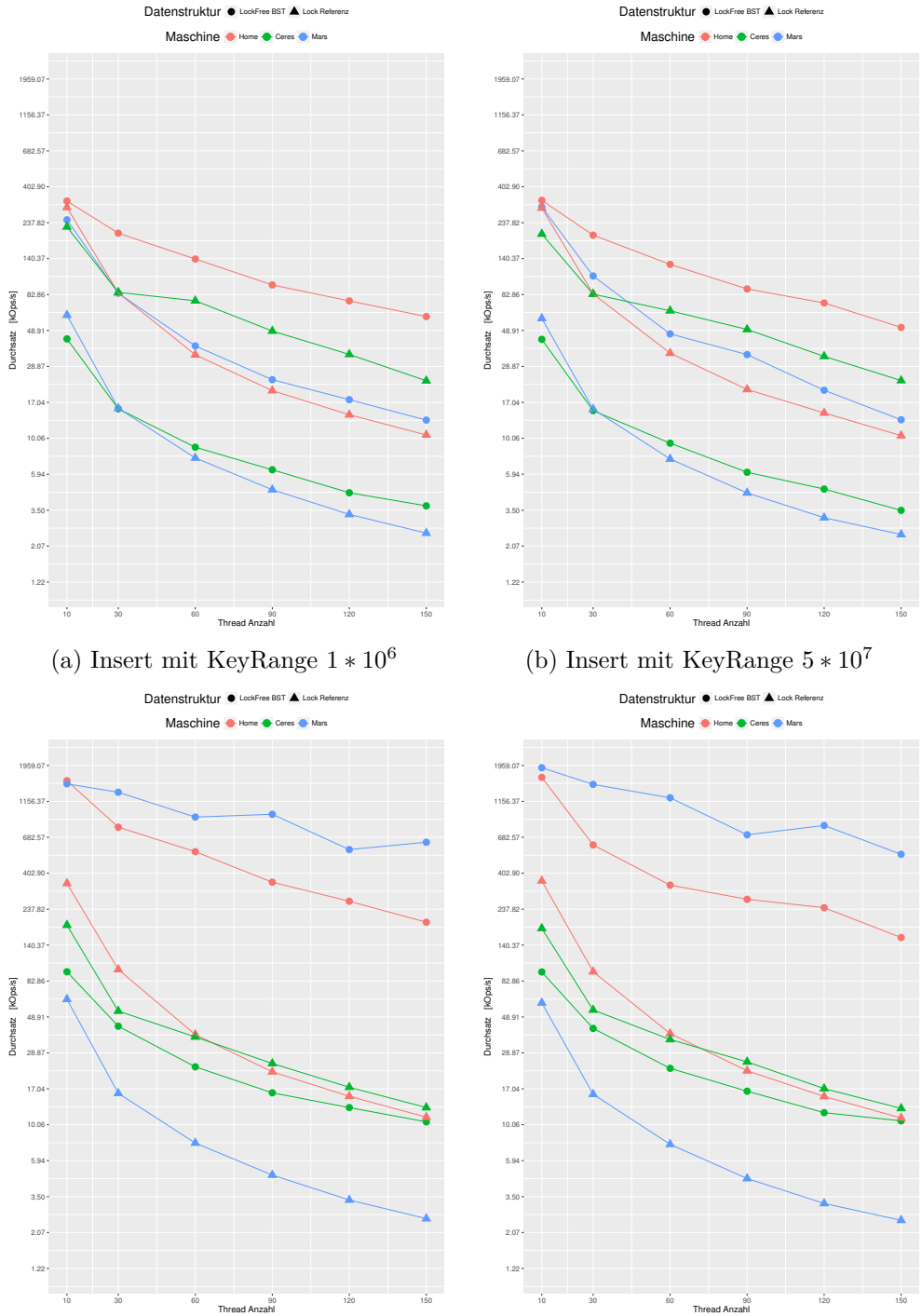


Abbildung 5.6: Benchmark der *Insert* Operation, aus einer Messung mit 10% Update (*Insert*) Last und 90% *Locate*; Key-range gleich $5 * 10^7$



(a) Insert mit KeyRange $1 * 10^6$

(b) Insert mit KeyRange $5 * 10^7$

(c) Continas mit KeyRange $1 * 10^6$

(d) Cointains mit KeyRange $5 * 10^7$

Abbildung 5.7: Übersicht der Benchmarks von *Insert* sowie *Contains*, mit 10% Update (*Insert*, ausschließlich erfolgreiche) Last und unterschiedlichen Key-Ranges mit log_2 y-Achse

KAPITEL 6

Diskussion der Ergebnisse

Dieses Kapitel beschreibt die primären Erkenntnisse dieser Arbeit und gibt einen Ausblick auf zukünftige Forschungsmöglichkeiten. Zusätzlich findet sich eine Konklusion, die über das geplante und das erreichte Ziel reflektiert.

6.1 Zentrale Ergebnisse

Der Algorithmus von Chatterjee et al. konnte nicht vollständig implementiert werden. Dies ist primär auf Lücken im Pseudocode zurückzuführen und widerlegt *nicht* die Funktionsfähigkeit des vorgestellten Verfahrens selbst.

Die Implementierung war so weit möglich, dass bis auf einen parallelen Spezialfall alle Funktionalitäten realisiert werden konnten. Gemessen und einem Benchmark unterzogen wurden ausschließlich die voll funktionsfähigen Teile des Algorithmus, nämlich die folgenden zwei von drei Operationen des Baumes: *Insert* und *Contains*.

Ausgiebiges Untersuchen des Pseudocodes deutet darauf hin, dass Pseudocode-Teile zur Verarbeitung spezieller Zustände von Chatterjee et al. nicht ausgeführt wurden. Diese Zustände sind unzulässige Kombinationen von Link-Zuständen (siehe Abschnitt 4.2.2), die erst bei Hochlast auftreten, wie dies im Abschnitt 5.5 erläutert wird. So konnten nicht alle Fehler beseitigt werden, um zu einem einwandfreien Zustand zu gelangen. Im Speziellen manifestierte sich dies in Endlosschleifen bei der Verwendung des Algorithmus.

6.1.1 Limitationen des Algorithmus

Eine Einschränkung stellt der unbalancierte Charakter der untersuchten Datenstruktur dar, da diese die Möglichkeit der Entartung besitzt, im Abschnitt 5.8 wird ein Beispiel dazu angeführt. Weiters bleibt das Problem der Speicherbereinigung nach dem logischen Löschen eines Knotens unbehandelt, was im Falle der eingesetzten Programmiersprache C++11 einer Lösung bedarf.

6.1.2 Errungenschaften dieser Arbeit

Eine wesentliche Errungeschaft stellt die Validierung des Konzepts eines Fault-Injectors (siehe Unterabschnitt 5.7.2) zur Simulation von parallelen Konflikten dar. Dieser ermöglicht, beliebige Race-Conditions zwischen CAS Operationen zu forcieren, indem ausgewählte Operationen vor oder nach ihrer Ausführung zu einer zeitlichen Unterbrechung gezwungen werden. Dadurch konnten auch rare Fälle im auszuführenden Programmpfad getestet werden.

Bei der Kontrolle des richtigen Programmablaufs lag die größte Errungenschaft beim Einsatz des Open Source Tracing Tools LTTng (siehe Unterabschnitt 5.7.1), das Debugging der parallel arbeitenden Applikation ermöglichte.

6.1.3 Rekursionen im Pseudocode

Die verwendeten Rekursionen im Pseudocode erschweren die Auffassung des Pseudocodes erheblich.
Über 50% der Implementierungstätigkeiten wurden mit Debugging verbracht, da eine direkte Umsetzung des Pseudocodes aus mehreren Gründen (siehe Abschnitt 5.4) in Fehlern wie Endlosschleifen mündete. Der Verständnisaufbau und die Fehlerbehebung haben unerwartet viel Zeit eingenommen.

So sei als Beispiel das gewöhnliche Entfernen eines Kategorie 3-Knotens genannt, wie es in Unterabschnitt 4.3.4 exemplarisch ausgeführt ist. Die Aufruftiefe liegt hier bei sechs verschachtelten Methodenaufrufen. Zur Vereinfachung hätten die zwei im Pseudocode der Autoren gezeigten Methoden *CleanFlag* sowie *CleanMark* in mehrere kleine Teile aufgespalten werden können. Eine andere Möglichkeit wäre gewesen, den Quellcode nach den CAS Operationen logisch auszurichten, was nicht teil dieser Arbeit war.

6.2 Zukünftige Forschungen

Weiterführende Arbeiten könnten sich mit dem vollständigen Implementieren der *Remove* Operation sowie dem daraus folgenden Benchmarks beschäftigen.

Zur besseren Vergleichbarkeit der Benchmark-Ergebnisse könnte zukünftig das Benchmark Framework *Synchrobench* von Gramoli und Vincent [Gra15] eingesetzt werden. Dies hätte den Vorteil, dass sogleich Vergleiche mit anderen Datenstrukturen vorliegen.

Intensivierungen des Einsatzes des Fault-Injectors (siehe Unterabschnitt 5.7.2) könnten zusätzlich weitere Fehler zu Tage bringen. Im Speziellen kann nochmaliges genaues Prüfen der aufgestellten *LEMMA*ta [CNT14, S.328-329] mit dem Fault-Injector zu neuen Sichtweisen auf bestehende Probleme (siehe Abschnitt 5.5) führen.

6.3 Konklusion

Das wichtigste Ziel dieser Arbeit war die Analyse der Implementierbarkeit des untersuchten Algorithmus. Dieses Ziel konnte nicht vollständig erreicht werden, da Probleme zu Endlosschleifen führten. Die wichtigste persönliche Erkenntnis des Autors dieser Masterarbeit ist, dass bestehenden Pseudocode in eine funktionsfähige Applikation zu überführen, größere Probleme als vermutet mit sich bringen kann. Die Komplexität der Umsetzung steigt, wenn a) Unklarheiten im Pseudocode vorhanden sind und b) der Pseudocode auf Rekursionen setzt, was ein Nachvollziehen des richtigen Programmablaufs erschwert.

Abschließend sei gesagt, dass durchaus die Möglichkeit existiert, dass nicht funktionsfähige Teile des Pseudocodes in einer durch Chatterjee et al. überarbeiteten Variante erscheinen, da diesbezüglich schon mit den Autoren Kontakt aufgenommen wurde.

Abbildungsverzeichnis

Tabellenverzeichnis

Liste der Algorithmen

Index

Akronyme

API Application Programming Interface. 42

AVL Georgy Adelson-Velsky and Evgenii Landis'. 30

BST Binary Search Tree. 2, 3, 14–16, 22, 23, 28, 29, 33–36, 38–40, 42, 44, 46, 48, 50, 52–54, 56, 57, 59, 60, 62, 63, 75, 78

CAS Compare and Swap. 21–23, 26, 34, 41, 46, 48, 51, 55, 60, 67–69, 72, 86

CPU Central Processing Unit. 8, 61

FGL Fine grained locking. 30

GCC GNU Compiler Collection). 60, 61

GDB GNU Debugger. 68

LTTng Linux Trace Toolkit: next generation. 61, 68

MIMD Multiple Instruction Multiple Data. 6–8

MISD Multiple Instruction Single Data. 6

RAII Resource Acquisition Is Initialization. 68, 70

SIMD Single Instruction Multiple Data. 6

SISD Single Instruction Single Data. 6

SMT Simultaneous MultiThreading. 8

STM Software Transactional Memory. 66

TDD Test Driven Development. 61, 67

Literaturverzeichnis

[AKK+12] Yehuda Afek, Haim Kaplan, Boris Korenfeld, Adam Morrison, and Robert Endre Tarjan. A Practical Concurrent Self-Adjusting Search Tree. In *Distributed Computing - 26th International Symposium (DISC)*, pages 1–15, 2012.

[BCCO10] Nathan Grasso Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 257–268, 2010.

[BER14] Trevor Brown, Faith Ellen, and Eric Ruppert. A General Technique for Non-blocking Trees. *SIGPLAN Not.*, 49(8):329–342, February 2014.

[Bre15] Ulrich Breymann. *Der C++-Programmierer: C++ lernen – professionell anwenden – Lösungen nutzen.* Carl Hanser Verlag GmbH & Company KG, 2015.

[CGR12] Tyler Crain, Vincent Gramoli, and Michel Raynal. A speculation-friendly binary search tree. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 161–170. ACM, 2012.

[CGR13] Tyler Crain, Vincent Gramoli, and Michel Raynal. A Contention-Friendly Binary Search Tree. In *Euro-Par*, volume 8097 of *Lecture Notes in Computer Science*, pages 229–240. Springer, 2013.

[CNT14] Bapi Chatterjee, Nhan Nguyen, and Philippas Tsigas. Efficient Lock-free Binary Search Trees. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 322–331. ACM, 2014.

[Com79] Douglas Comer. The Ubiquitous B-Tree. *ACM Comput. Surv.*, 11(2):121–137, 1979.

[cpp16a] cppreferences.com. `http://en.cppreference.com/w/cpp/chrono/high_resolution_clock`, [Online; zugegriffen am 13.Aug.2016].

[cpp16b] cppreference.com. cppreferences std::memory_order. http://en.cppreference.com/w/cpp/atomic/memory_order, June [Online; zugegriffen am 14.Aug.2016].

[DMS14] Martin Dietzfelbinger, Kurt Mehlhorn, and Peter Sanders. *Algorithmen und Datenstrukturen - die Grundwerkzeuge.* eXamen.press. Springer, 2014.

[DVY14] Dana Drachsler, Martin T. Vechev, and Eran Yahav. Practical concurrent binary search trees via logical ordering. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 343–356. ACM, 2014.

[EFHR14] Faith Ellen, Panagiota Fatourou, Joanna Helga, and Eric Ruppert. The Amortized Complexity of Non-blocking Binary Search Trees. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 332–340. ACM, 2014.

[EFRvB10] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 131–140. ACM, 2010.

[Fly72] Michael J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, September 1972.

[git16a] greatest - A testing system for C, contained in 1 file. https://github.com/silentbicycle/greatest, [Online; zugegriffen am 29.Mai.2016] 2016.

[git16b] Google Test, Google's C++ test framework. https://github.com/google/googletest, [Online; zugegriffen am 29.Mai.2016].

[GN00] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software: Practice and Experience, Vol.30(11), pp.1203-1233*, 9 2000.

[Gra15] Vincent Gramoli. More Than You Ever Wanted to Know About Synchronization: Synchrobench, Measuring the Impact of the Synchronization on Concurrent Algorithms. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 1–10. ACM, 2015.

[GS78] Leonidas J. Guibas and Robert Sedgewick. A Dichromatic Framework for Balanced Trees. In *FOCS*, pages 8–21. IEEE Computer Society, 1978.

[Güt92] Ralf H. Güting. *Datenstrukturen und Algorithmen.* B.G. Teubner Stuttgart, 1992.

[Hal13] Prentice Hall. *C++11 for Programmers.* Deitel Developer Series. Prentice Hall, 2 edition, 2013.

[Har01] Timothy L. Harris. A Pragmatic Implementation of Non-blocking Linked-Lists. In *DISC*, volume 2180 of *Lecture Notes in Computer Science*, pages 300–314. Springer, 2001.

[HJ12] Shane V. Howley and Jeremy Jones. A non-blocking internal binary search tree. In *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 161–171. ACM, 2012.

[HS08] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann, April 2008.

[HS12] Maurice Herlihy and Nir Shavit. Art of Multiprocessor Programming, Revised Reprint, 2012.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[JSRV] Nicholas Jalbert, Koushik Sen, Gruia-catalin Roman, and André Van Der Hoek. A trace simplification technique for effective debugging of concurrent programs.

[Kor13] Christopher Kormanyos. *Real-Time C++ - Efficient Object-Oriented and Template Microcontroller Programming.* Springer, 2013.

[KP11] Alex Kogan and Erez Petrank. Wait-free queues with multiple enqueuers and dequeuers. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 223–234. ACM, 2011.

[KS07] Herbert Klaeren and Michael Sperber. *Die Macht der Abstraktion. Einführung in die Programmierung.* B. G. Vieweg+Teubner, 1. Aufl. edition, 2007.

[Lat14] Roberto Latorre. Effects of Developer Experience on Learning and Applying Unit Test-Driven Development. *IEEE Transactions on Software Engineering*, 40(4):381–395, April 2014.

[NM14] Aravind Natarajan and Neeraj Mittal. Fast Concurrent Lock-free Binary Search Trees. *SIGPLAN Not.*, 49(8):317–328, February 2014.

[NSS91] Otto Nurmi and Eljas Soisalon-Soininen. Uncoupling Updating and Rebalancing in Chromatic Binary Search Trees. In *Proceedings of the Tenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '91, pages 192–198. ACM, 1991.

[NSS96] Otto Nurmi and Eljas Soisalon-Soininen. Chromatic Binary Search Trees. A Structure for Concurrent Rebalancing. *Acta Inf.*, 33(6):547–557, 1996.

[OW12] Thomas Ottmann and Peter Widmayer. *Algorithmen und Datenstrukturen, 5. Auflage.* Spektrum Akademischer Verlag, 2012.

[PT60] A. J. Perlis and Charles Thornton. Symbol Manipulation by Threaded Lists. *Commun. ACM*, 3(4):195–204, April 1960.

[Ray13] Michel Raynal. *Concurrent Programming - Algorithms, Principles, and Foundations.* Springer, 2013.

[RR10] Thomas Rauber and Gudula Rünger. *Parallel Programming - for Multicore and Cluster Systems.* Springer, 2010.

[SA05] Lawrence Spracklen and Santosh G. Abraham. Chip Multithreading: Opportunities and Challenges. In *HPCA*, pages 248–252. IEEE Computer Society, 2005.

[SG00] Andreas Solymosi and Ulrich Grude. *Grundkurs Algorithmen und Datenstrukturen - eine Einführung in die praktische Informatik mit Java.* Vieweg, 2000.

[SG14] Andreas Solymosi and Ulrich [author] Grude. Grundkurs Algorithmen und Datenstrukturen in JAVA; Eine Einführung in die praktische Informatik, 2014.

[Tec12] AMD64 Technology. AMD64 Architecture Programmer's Manual. `http://developer.amd.com/wordpress/media/2012/10/24593_APM_v2.pdf`, September 2012.

[TP14] Shahar Timnat and Erez Petrank. A Practical Wait-free Simulation for Lock-free Data Structures. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 357–368. ACM, 2014.

[Trö08] Peter Tröger. The Multi-Core Era - Trends and Challenges. *The Computing Research Repository (CoRR)*, abs/0810.5439, 2008.

[TS06] N. Tillmann and W. Schulte. Mock-object generation with behavior. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 365–368, Sept 2006.