Zürcher Hochschule
für Angewandte Wissenschaften

School of
Engineering

InIT Institut für angewandte
Informationstechnologie

Bachelorarbeit Informatik

Best-Practices zur Performance-Optimierung bei der Software-Entwicklung in Java

Autoren	Manuel Simbürger Roman Thöni
Hauptbetreuung	Mark Cieliebak Markus Thaler

Datum	05.06.2015

Zusammenfassung

Während lauffähiger Programmcode in der Regel vergleichsweise einfach geschrieben werden kann, ist das Schreiben von zeiteffizientem Code oft eine Herausforderung. Nicht nur die Wahl der Datenstrukturen, sondern auch die verwendeten Code-Konstrukte spielen dabei eine wichtige Rolle. Im Internet kursieren viele Meinungen zum Thema Performance, jedoch existieren nur wenige detaillierte Experimente oder Dokumentationen zu diesem Thema.
Da die vorhandenen Erkenntnisse meist veraltet sind, sollten diese im Rahmen dieser Arbeit überprüft und um neue erweitert werden. Der Hauptfokus liegt darauf, eine Sammlung von *Good-Practises* beim Entwickeln von Java Programmen zu erstellen.
Performancemessungen werden auf aktuellen Computersystemen immer schwerer, da allein schon durch die Laufzeitoptimierung des JIT der ursprüngliche Code drastisch verändert werden kann. Um dennoch möglichst aussagekräftige Resultate zu erhalten, wurde in einem ersten Schritt eine mehrere Rechner umfassende Testumgebung aufgebaut, wobei sich die Testrechner alle in Hardware oder Betriebssystem unterscheiden. In einem zweiten Schritt wurden zwei Benchmarking Tools für Java evaluiert und eine Reihe von Performancetests definiert, welche in einem dritten Schritt umgesetzt und ausgewertet wurden. Jeder Abschnitt im Kapitel der Resultate enthält am Ende zudem eine Empfehlung über zu verwendende Code-Konstrukte für effizienten Java Code.

Abstract

Although it is relatively easy to write functioning source code, writing time-efficient code is often on a completely different level of complexity. Not only which data structure is chosen, but also the general code constructs used play a key role. A wide range of differing ideas and standpoints on the issue of performance can be found on the internet, but detailed experiments or documentation are rare.
As most of the existing insights are outdated, they are to be examined and complemented with new concepts in this work. The main goal is to create a compilation of *best-practice* for writing Java source code.
Measuring performance on current systems has become more and more complex because the original code can be radically changed only by the runtime optimisation of the JIT. Therefore, in order to obtain the most meaningful results, the first step was to set up a test environment consisting of several computers, all of which differed as regards hardware or operating systems. A second step was to evaluate two different benchmarking tools for Java and to devise a series of performance tests, which were then implemented and analysed in a final step. Each part of the results chapter contains a recommendation on the use of code constructs for efficient Java code.

Vorwort

Da wir im Laufe unseres Informatikstudiums selbst sehr viel Code erstellen mussten und uns das Thema Performance auch persönlich sehr interessiert, haben wir uns für diese Bachelorarbeit entschieden. Mit dieser Arbeit möchten wir einige Vorteile überprüfen und so mit einer einfachen Übersicht den Entwicklungsalltag eines Software-Ingenieurs vereinfachen.
Wir möchten uns an dieser Stelle herzlich bei Mark Cieliebak und Markus Thaler bedanken, welche uns während der gesamten Arbeit stets mit kompetenten Antworten und konstruktivem Feedback unterstützt haben.

Erklärung betreffend das selbständige Verfassen einer Bachelorarbeit an der School of Engineering

Mit der Abgabe dieser Bachelorarbeit versichert der/die Studierende, dass er/sie die Arbeit selbständig und ohne fremde Hilfe verfasst hat. (Bei Gruppenarbeiten gelten die Leistungen der übrigen Gruppenmitglieder nicht als fremde Hilfe.)

Der/die unterzeichnende Studierende erklärt, dass alle zitierten Quellen (auch Internetseiten) im Text oder Anhang korrekt nachgewiesen sind, d.h. dass die Bachelorarbeit keine Plagiate enthält, also keine Teile, die teilweise oder vollständig aus einem fremden Text oder einer fremden Arbeit unter Vorgabe der eigenen Urheberschaft bzw. ohne Quellenangabe übernommen worden sind.

Bei Verfehlungen aller Art treten die Paragraphen 39 und 40 (Unredlichkeit und Verfahren bei Unredlichkeit) der ZHAW Prüfungsordnung sowie die Bestimmungen der Disziplinarmassnahmen der Hochschulordnung in Kraft.

Ort, Datum:

……

Unterschriften:

……...

……

……

Das Original dieses Formulars ist bei der ZHAW-Version aller abgegebenen Bachelorarbeiten zu Beginn der Dokumentation nach dem Abstract bzw. dem Management Summary mit Original-Unterschriften und -Datum (keine Kopie) einzufügen.

Inhaltsverzeichnis

1 Einleitung

1.1 Ausgangslage

Bei der Softwareentwicklung geht es oft in erster Linie darum fehlerfreien Code zu schreiben, der die geforderten Funktionalitäten erfüllt. Dem entsprechend überprüfen die Tests während der Entwicklung und für die Abnahme in der Regel lediglich die korrekte Ausführung, nicht jedoch die Performanz. Obwohl langsame Codeteile mithilfe von Java Mission Control oder gängigen Java Profilern schnell eingegrenzt werden können so ist das Beheben oft mit erheblichem Aufwand verbunden. Das bessere Vorgehen wäre, bereits von Beginn an effizienten Code zu schreiben und an diesem Punkt setzt diese Arbeit an.

Als Grundlage diente das Buch Effective Java (Bloch, 2011) von Joshua Bloch. In seinem Buch stellt Bloch 78 Items vor und zeigt anhand von Codebeispielen was getan oder vermieden werden sollte. Das 2008 erschienene Werk basiert auf Java 6, behandelt jedoch vor allem Features wie Autoboxing und Enumerations die mit Java 5 hinzugefügt wurden. Seither wurde die Buchreihe nicht weitergeführt.

Des weiteren existieren diverse Whitepaper und online Artikel zum Thema Java Performance, welche jedoch oft nur ältere Java Versionen behandeln. Aktuelle Literatur zum Thema ist kaum vorhanden.

1.2 Zieelsetzung / Aufgabenstellung / Anforderungen

Ziel dieser Arbeit ist es, experimentell die Performance von verschiedenen Java Code Konstrukten, welche die selbe Aufgabe lösen, zu messen und davon Erkenntnisse für das Entwickeln von effizienten Java Programmen abzuleiten. Analysiert werden vor allem die Java Libraries `java.lang` und `java.util`, jedoch keine grafischen Oberflächen oder Elemente aus `java.io`. Um diese Experimente durchführen zu können wird vorgängig eine geeignete Testumgebung aufgebaut und ein Benchmarksystem evaluiert. Die Benchmarks müssen einfach erweitert und beliebig oft ausgeführt werden können. Zudem sollen die Tests zu einem beliebigen späteren Zeitpunkt auch auf einer anderen Testumgebung ausgeführt werden können.

Die offizielle Aufgabenstellung befindet sich im Anhang A.1.

1.3 Abgrenzung

Im Rahmen dieser Arbeit beschränken sich die Benchmarks lediglich auf Zeitmessungen im Millisekundenbereich. Nicht gemessen werden Grössen wie verwendeter Arbeitsspeicher, die Anzahl der erstellten Objekte oder die effektive Auslastung des Prozessors. Die Resultate beschränken sich in erster Linie auf beobachtetes Laufzeitverhalten eines Code-Konstrukts.

2 Theoretische Grundlagen

Im folgenden Abschnitt werden die theoretischen Grundlagen für Performancemessungen in Java erläutert.

2.1 Benchmarks

Beim Schreiben von Benchmarks muss eine Reihe von Punkten beachtet werden. Nachdem definiert wurde was getestet werden soll muss sichergestellt werden, dass der Benchmark dies auch wirklich tut. Um dies zu erreichen muss der Code entweder das Programm in irgendeiner anderen Weise verändern (wie zum Beispiel das Hinzufügen eines Wertes zu einer Collection) oder einen Rückgabewert haben (Decker und Kick, o.J[a]). Letzteres ist notwendig, da von Java an mehreren Stellen exzessiv optimiert wird und allfälliger "toter"-Code entfernt werden. Dies gilt insbesondere auch für die Benchmark-Methoden, weshalb Rückgabewerte mindestens in einem Array gespeichert oder an das Benchmark-Framework zurückgegeben werden sollten.

Der Just-In-Time Compiler (JIT) von Java optimiert je nach ausführendem System mit unterschiedlicher Hardware / Java Version anders und somit variieren die gemessenen Werte mitunter stark. Um dieser Fehlerquelle entgegenzuwirken sollte der zu testende Code vor der eigentlichen Messung mehrere Male ausgeführt werden (oft "WarmUp"-Phase genannt).

Um vergleichbare Werte zu bekommen sollte das Benchmarking auf einem Computersystem mit möglichst geringer Arbeitslast ausgeführt werden. Da die Effizienz der Optimierungen stark von der Hard- und Software des jeweiligen Systems abhängt, sollten für allgemeine Aussagen zur Performance eines Codeabschnitts mehrere verschiedene Testsysteme benutzt werden.

2.2 Zeitmessung

Die Zeitmessung während des Programmablaufs hängt unter anderem von Hyperthreading, Processswapping und generellen Messungungenauigkeiten, welche zum Beispiel durch die Garbage Collection verursacht werden, ab. Hinzu kommt, dass für jede Zeitmessung ein Systemcall durchgeführt werden muss. Da das Betriebssystem diese Anfragen nicht direkt bearbeiten muss, kann dies zu weiteren Verzögerungen und somit Messungenauigkeiten führen. Um dem entgegenzuwirken sollte die Laufzeit eines Benchmarks im Millisekunden Bereich liegen. Um dies zu erreichen ist in der Regel eine Schleife notwendig, was zusätzlichen Aufwand für das System bedeutet und die Messung zusätzlich leicht verfälscht. Wird für sämtliche Benchmarks die selbe Schleife verwendet, so bleiben die Resultate jedoch vergleichbar.

3 Vorgehen / Methoden

In den folgenden Abschnitten werden die verwendeten Benchmark Frameworks kurz vorgestellt und deren Vor- und Nachteile aufgezeigt. Des weiteren werden sämtliche Experimente sowie deren Durchführung und Auswertung beschrieben.
Der vollständige Code sämtlicher erarbeiteter Programme befindet sich auf der CD im Anhang.

3.1 Caliper

Caliper ist ein opensource Framework zum Schreiben, Ausführen und Auswerten von Java Microbenchmarks (Decker und Kick, o.J[b]). Das Framework wurde hauptsächlich von Colin Decker und Gregory Kick, beide Softwareingenieure bei Google, entwickelt und wurde auf Google Code[1] veröffentlicht und später auf Github[2] migriert. Der letzte offizielle Release ist Beta 1.0, jedoch ist über Github eine aktuellere Betaversion verfügbar.

```
@Benchmark
long factorialRecursive(long reps) {
    long dummy = 0L;
    for (long i = 0; i < reps; i++) {
        dummy += Factorial.recursive(20);
    }
    return dummy;
}
```

Listing 3.1: Caliper Benchmark Beispiel

In der verwendeten Version können Benchmarks einfach per Annotation geschrieben werden (siehe Listing 3.1). Der Parameter `reps` bestimmt, wie oft eine bestimmte Messung durchgeführt werden soll. Caliper bestimmt diesen Wert automatisch und erhöht ihn, sollte ein Methodenaufruf die vorgesehene Mindestlaufzeit unterschreiten.

```
Starting Caliper benchmark
C:\Users\Roman\Documents\GitHub\CaliperBenchmarks\caliper\1432800239662
    _Caliper_Benchmark.log
Experiment selection:
  Benchmark Methods:   [factorialRecursive]
  Instruments:   [allocation, runtime]
  User parameters:   {}
  Virtual machines:  [default]
  Selection type:    Full cartesian product

This selection yields 2 experiments.
Trial Report (1 of 2):
  Experiment {instrument=allocation, benchmarkMethod=factorialRecursive, vm=
      default, parameters={}}
  Results:
    bytes(B): min=0.00, 1st qu.=0.00, median=0.00, mean=0.00, 3rd qu.=0.00,
        max=0.00
    objects: min=0.00, 1st qu.=0.00, median=0.00, mean=0.00, 3rd qu.=0.00, max
        =0.00
Trial Report (2 of 2):
  Experiment {instrument=runtime, benchmarkMethod=factorialRecursive, vm=
      default, parameters={}}
```

[1] URL: https://code.google.com/p/caliper/ [Stand 31.05.2015]
[2] URL: https://github.com/google/caliper/ [Stand 31.05.2015]

```
  Results:
    runtime(ns): min=21.99, 1st qu.=22.18, median=22.51, mean=22.51, 3rd qu
        .=22.77, max=23.05
Collected 27 measurements from:
  2 instrument(s)
  2 virtual machine(s)
  1 benchmark(s)

Execution complete: 21.37 s.
Results have been uploaded. View them at: https://microbenchmarks.appspot.com/
    runs/3ba9bc68-11cc-453c-97aa-a665c789d9c6
```

Listing 3.2: Caliper Benchmark Ausgabe

Ausgaben in der Konsole sind auf ein Minimum reduziert. Stattdessen werden die Ergebnisse der Benchmarks von Caliper aufbereitet und direkt auf `http://microbenchmarks.appspot.com/` hochgeladen. Ausgewertet werden neben der Laufzeit auch der Speicherverbrauch und die Anzahl der erstellten Objekte (vgl. Abbildung 3.1). Zudem speichert Caliper auch sämtliche JVM Argumente. Standardmässig werden alle Benchmarks einzeln und unabhängig voneinander gespeichert. Durch die Angabe eines API-Keys ist es jedoch möglich auch später auf alle Messungen zuzugreifen.

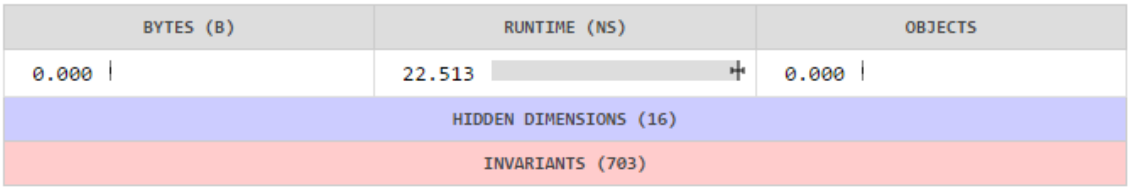

Abbildung 3.1: Beispiel Caliper Webinterface

Nachteile von Caliper sind jedoch, dass das Projekt sehr viel Arbeitsspeicher benötigt um kompiliert werden zu können. Mit lediglich 4 Benchmark Klassen wurden bereits bis zu 10 GB RAM belegt. Während der Ausführung lagen die Spitzen bei knapp 3 GB, welche von den meisten Testrechnern nicht alloziert werden konnten. Des weiteren stürzte Caliper während der Ausführung regelmässig ab oder konnte die Ergebnisse nicht hochladen. Aus diesen Gründen wurde entschieden, Caliper nicht weiter für Experimente zu nutzen.

3.2 JMH

Eine Alternative zu Caliper für das Schreiben, Ausführen und Auswerten von Java Microbenchmarks ist JMH[3], welches von Oracle entwickelt wurde. JMH wird laufend erweitert und läuft zuverlässig und stabil. Dies waren die Hauptgründe für die Verwendung in dieser Arbeit. Die Ausgabe der Resultate (siehe Listing 3.5) beschränken sich bei JMH lediglich auf die Konsole und werden nicht anderweitig gespeichert.

3.2.1 Einrichtung von JMH

Für die Erstellung von Benchmarks mit Hilfe von JMH ist zwingend ein Maven Projekt nötig. Nach dem Hinzufügen der notwendigen Dependencies für JMH kann ohne weitere Vorbereitungen mit dem Schreiben der Benchmarks begonnen werden.

[3]URL: `http://openjdk.java.net/projects/code-tools/jmh/` [Stand 31.05.2015]

3.2.2 Verwendung von JMH

In der Hauptklasse muss in der JMH Runner gestartet und mit entsprechenden Optionen ausgestattet werden. Die Optionen umfassen mindestens die auszuführende Benchmark Klasse und die `.build()` Anweisung. Zusätzlich können beispielsweise die Anzahl der Methodeniterationen, JVM Argumente, die Zeiteinheit oder der Modus festgelegt werden (siehe Listing 3.3).

```
    private static void main(String[] args) throws RunnerException {
        Options opt = new OptionsBuilder()
                .include(ch.zhaw.bafs15.jmhbenchmark.
                    IterativeVersusRecursiveTest.class.getSimpleName())
                .warmupIterations(10)
                .measurementIterations(10)
                .operationsPerInvocation(1)
                .timeUnit(TimeUnit.MILLISECONDS)
                .mode(Mode.AverageTime)
                .jvmArgs(References.JVM_OPTIONS.STANDARD)
                .forks(1)
                .build();
        new Runner(opt).run();
    }
```

Listing 3.3: JMH Benchmark starten

Ähnlich wie bei Caliper können auch bei JMH per `@Benchmark`-Annotation sehr einfach neue Benchmarkmethoden geschrieben werden (siehe Listing 3.4). Anders als Caliper liefert JMH keine Variable für die Anzahl der Iterationen mit. Um geeignete Messwerte zu erhalten sollte jedoch trotzdem eine Schleife mit einer für die Benchmarks geeigneten Anzahl Repetitionen eingebaut werden.

```
    @Benchmark
    public long recursive() {
        long dummy = 0L;
        for(long i = 0; i < References.BENCHMARK.REPETITIONS; i++) {
            dummy += IterativeVersusRecursive.recursive(20);
        }
        return dummy;
    }
```

Listing 3.4: JMH Benchmark Beispiel

Für jede Klasse kann zusätzlich noch der Scope festgelegt werden, mit welchem definiert wird wo die Instanz der Klasse zur Laufzeit verfügbar ist. Der Standardwert ist hier `Scope.Thread`, das heisst eine Instanz pro Thread der die Klasse testet.

Ähnlich wie bei JUnit kann zudem eine `@Setup` und `@TearDown` Methode definiert werden, welche zu verschiedenen Zeiten gestartet werden können. Der Standardwert wäre `Level.Trial`, vor bzw. nach dem Durchlaufen des kompletten Benchmarks für eine Methode.

```
# JMH 1.9.2 (released 18 days ago)
# VM invoker: C:\Program Files\Java\jre1.8.0_45\bin\java.exe
# VM options: -Xms512m -Xmx1536m
# Warmup: 10 iterations, 1 s each
# Measurement: 10 iterations, 1 s each
# Timeout: 10 min per iteration
# Threads: 1 thread, will synchronize iterations
# Benchmark mode: Average time, time/op
# Benchmark: ch.zhaw.bafs15.jmhbenchmark.IterativeVersusRecursiveTest.
    recursive

# Run progress: 50.00% complete, ETA 00:00:20
# Fork: 1 of 1
# Warmup Iteration   1: 38.335 ms/op
# Warmup Iteration   2: 37.073 ms/op
# Warmup Iteration   3: 37.171 ms/op
# Warmup Iteration   4: 37.876 ms/op
# Warmup Iteration   5: 37.111 ms/op
# Warmup Iteration   6: 37.209 ms/op
```

```
# Warmup Iteration   7: 37.177 ms/op
# Warmup Iteration   8: 37.183 ms/op
# Warmup Iteration   9: 37.494 ms/op
# Warmup Iteration  10: 37.640 ms/op
Iteration   1: 37.124 ms/op
Iteration   2: 37.647 ms/op
Iteration   3: 37.642 ms/op
Iteration   4: 37.313 ms/op
Iteration   5: 36.885 ms/op
Iteration   6: 37.139 ms/op
Iteration   7: 37.038 ms/op
Iteration   8: 36.940 ms/op
Iteration   9: 37.498 ms/op
Iteration  10: 37.206 ms/op

Result "recursive":
  37.243 ±(99.9%) 0.416 ms/op [Average]
  (min, avg, max) = (36.885, 37.243, 37.647), stdev = 0.275
  CI (99.9%): [36.827, 37.659] (assumes normal distribution)

# Run complete. Total time: 00:00:20

Benchmark                                  Mode  Cnt   Score   Error  Units
IterativeVersusRecursiveTest.recursive     avgt   10  37.243 ± 0.416  ms/op
```

Listing 3.5: JMH Benchmark Ausgabe

Erklärung der Ausgabe
Neben der JMH Version, der verwendeten Java Version sowie allen gesetzten Optionen werden auch sämtliche Laufzeiten der einzelnen Iterationen ausgegeben. Zum Schluss werden alle getesteten Methoden noch einmal mit einer Durchschnittszeit zusammengefasst. Der Wert unter Score ist hier der gemessene Wert, Error bezeichnet die Hälfte des Konfidenzintervalls.

Der Hauptnachteil von JMH ist sicherlich, dass die Daten lediglich auf die Konsole ausgegeben werden und somit noch anderweitig gespeichert und ausgewertet werden müssen. Des weiteren kann nur eine Klasse pro Durchlauf getestet werden. JMH bietet keine Möglichkeit beispielsweise ein ganzes Package auf einmal zu testen. Ein weiterer Nachteil ist, dass es zu JMH keine detaillierte Dokumentation gibt. Es existieren jedoch diverse Beispiele und Tutorials welche jedoch in der Regel nicht vom Entwickler Oracle selbst stammen.

3.2.3 Konkrete Implementierung in dieser Arbeit

Zur besseren Übersicht über einzelnen Experimente wurden diese in separate Benchmark Klassen aufgeteilt. Auf Grund der Tatsache, dass mit JMH nur jeweils eine Klasse pro Durchgang getestet werden kann, wurden die einzelnen Benchmarkklassen mittels Reflection dynamisch gesucht und an JMH übergeben. Diese Variante hat den Vorteil, dass zukünftige Experimente einfach durch das Schreiben einer weiteren Benchmarkklasse hinzugefügt werden können.

3.3 Durchführung

Die Experimente, die im Rahmen dieser Arbeit erstellt wurden, bestehen jeweils aus mehreren unterschiedlichen Code-Konstrukten, welche dieselbe Aufgabe erledigen. Alle Experimente sind zu einem einzigen Benchmarkprogramm zusammengefasst, welches insgesamt 27 mal auf den zur Verfügung stehenden 9 Testrechnern (siehe Tabelle 3.2) mit einer durchschnittlichen Laufzeit von 2 Stunden 45 Minuten ausgeführt wurde. Die erhaltenen Ergebnisse wurden eingelesen, verglichen und grafisch aufbereitet (siehe Abschnitt 3.4).

PC	OS	Java Version	Hardware
1	Windows 7 x64	1.8.0_31	CPU: 3.0GHz RAM: 16GB
2	Ubuntu 14.04 x64	1.8.0_45	CPU: 2.8GHz RAM: 4GB
3	Windows 8 x64	1.8.0_45	CPU: 3.4GHz RAM: 16GB
4	Windows 8 x64	1.8.0_45	CPU: 2.5GHz RAM: 4GB
5	Ubuntu 15.04 x64	1.8.0_45	CPU: 3.4GHz RAM: 8GB
6	Windows 7 x64	1.8.0_40	CPU: 3.4GHz RAM: 8GB
7	Windows 7 x86	1.8.0_45	CPU: 2.7GHz RAM: 4GB
8	Ubuntu 15.04 x64	1.8.0_45	CPU: 3.0GHz RAM: 2GB + 2GB Swap
9	Debian 7 x64	1.8.0_31	CPU: 3.4GHz RAM: 16GB

Tabelle 3.1: Spezifikation Testrechner

Um allgemeine Aussagen über das Verhalten der getesteten Codekonstrukte machen zu können wurden sämtliche Benchmarks unter Windows 7 (32bit, 64bit), Windows 8 (64bit), Ubuntu (64bit) und Debian 7 (64bit) mit den unterschiedlichsten Hardware Spezifikationen ausgeführt. Da während der Testphase kein Rechner mit MAC OS zur Verfügung stand, konnten mit diesem Betriebssystem keine Benchmarks durchgeführt werden.

3.4 Auswertung

Wie in Abschnitt 3.2 bereits erwähnt wird von JMH lediglich eine Konsolenausgabe erstellt. Diese musste in einem ersten Schritt in Textdateien abgespeichert werden, um die Daten zu einem späteren Zeitpunkt noch verfügbar zu haben. Nachfolgend wird erklärt wie die automatisierte Auswertung dieser Daten umgesetzt wurde.

3.4.1 Resultparser und Datenbank

Für die Erhaltung und einfache Auswertung der Daten wurde eine MySQL Datenbank eingesetzt, welche sich aus drei Tabellen zusammensetzt. Die verwendeten Computer in der Tabelle `machine`, die einzelnen Benchmarkdurchgänge in `benchmark` und die eigentlichen Resultate pro Benchmark in `results` (siehe Abbildung 3.2).

Die vom Benchmarkprogramm generierten Textdateien wurden mittels eines Dateiparsers automatisiert gelesen und in die Datenbank abgespeichert. Der Parser traversiert ein angegebenes Verzeichnis rekursiv und durchsucht vorhandene Textdateien nach Resultaten. Sobald alle Ergebnisse einer Datei gelesen wurden, wird eine SQL Abfrage erstellt und die Datenbank aktualisiert.

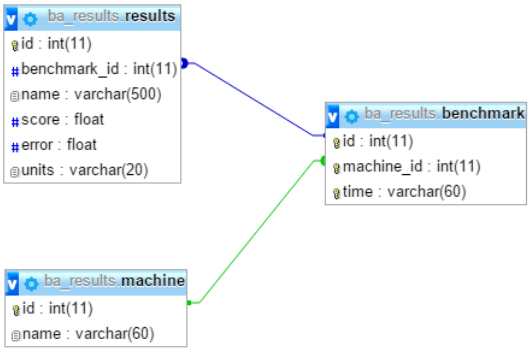

Abbildung 3.2: Design Datenbank

3.4.2 Diagrammerstellung

Für die grafische Aufbereitung der Daten wurde die opensource Library jFreeChart(JFree, o.J) Version 1.0.13 verwendet. Mit jFreeChart lassen sich einfach diverse Diagramme erstellen und wahlweise als Jpeg oder PNG speichern. Einziger Nachteil ist, dass der Developer Guide kostenpflichtig ist. Es existieren jedoch diverse online Anleitungen und Beispiele. Listing 3.6 ist ein minimales Codebeispiel für die Erstellung eines horizontalen Balkendiagramms (siehe Abbildung 3.3). Für die Auswertung wurde die Generierung der Diagramme soweit automatisiert, dass lediglich weitere SQL Statements geschrieben werden müssen.

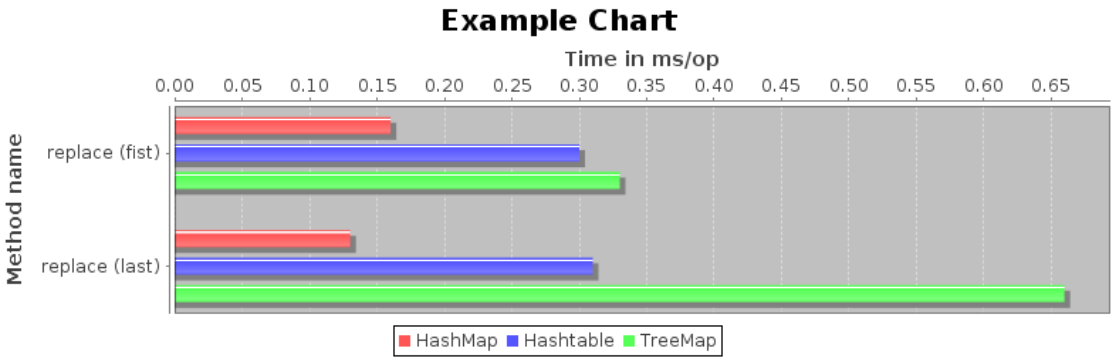

Abbildung 3.3: jFreeChart Beispiel Diagramm

```
DefaultCategoryDataset dataset = new DefaultCategoryDataset();
dataset.addValue(0.16, "HashMap", "replace (first)");
dataset.addValue(0.30, "Hashtable", "replace (first)");
dataset.addValue(0.33, "TreeMap", "replace (first)");

dataset.addValue(0.13, "HashMap", "replace (last)");
dataset.addValue(0.31, "Hashtable", "replace (last)");
dataset.addValue(0.66, "TreeMap", "replace (last)");

JFreeChart chart = ChartFactory.createBarChart(
  "Example Chart", //chart title
  "Method name", //Y axis name
  "Time in ms/op", //X axis name
  dataset, //category dataset
  PlotOrientation.HORIZONTAL, //Horizontal = bars from left to right,
      VERTICAL = bars from bottom to top
  true, //legend (true / false)
  false, //tool tips (true / false)
  false //urls (true / false)
);

File file = new File("charts/exampleChart.png");

ChartUtilities.saveChartAsPNG(file, chart, 800, 250);
```

Listing 3.6: jFreeChart Codebeispiel

3.5 Experimente

Die nachfolgende Tabelle enthält eine grobe Übersicht über alle durchgeführten Experimente.

Abschnitt	Thema	Beschreibung	Seite
4.1	Arrays	Initialisieren und Kopieren von Arrays	15
4.2	Autoboxing	Automatisches Boxing/Unboxing von Zahlenwerten, verstecke Objekterstellung	17
4.3.1	Collection: `List`	Einfügen, Lesen, Suchen, Verändern, Sortieren, Löschen	18
4.3.2	Collection: `Map`	Einfügen, Lesen, Suchen, Verändern, Löschen	29
4.3.3	Collection: `Set`	Einfügen, Suchen, Verändern, Löschen	33
4.4	Iterativ / Rekursiv	Vergleich von iterativen und rekursiven Ansätzen	35
4.5	Object Creation	Explizite und implizite Objekterstellung	36
4.6	Reflection	Vergleich von statischem Import und Reflection	38
4.7	Schleifen	Vergleiche von `for`-, `foreach`- und `while` -Schleifen	39
4.8	`Strings`	Verkettung von Strings	40
4.9	`Switch`, `else if` und `if`	`Switch-Cases` und Verkettung von `if-else`	42
4.10	Switch mit `Enum`, `int` und `String`	Flusskontrolle mit `switch`-Cases	43
4.11	Zufallszahlen	Pseudo- und kryptographische-Zufallszahlen	45

Tabelle 3.2: Liste der Experimente

4 Resultate

Im nachfolgenden Kapitel werden die gemessenen Resultate für die einzelnen Konstrukte mit Hilfe von Balkendiagrammen und Codebeispielen erklärt und analysiert.
Um für alle getesteten Experimente nachvollziehbare und reproduzierbare Werte zu erhalten wurden die einzelnen Methoden innerhalb eines Benchmarks jeweils 10'000 mal für Collections und 1'000'000 mal für die restlichen Benchmarks ausgeführt. Die Genauigkeit der Ergebnisse musste im Millisekundenbereich messbar sein. Nachfolgend wird erläutert, welche Operationen wie getestet wurden.

4.1 Arrays

Kopieren von Arrays

Aufbau der Experimente

Das Kopieren von Arrays kann auf verschiedenste Weise durchgeführt werden, entweder durch die Java internen Methoden wie `Object.clone()`, `System.arraycopy(Object src, int srcPos, Object dest, int destPos, int length)`, `Arrays.copyOf(int[] original, int newLength)`, `Arrays.copyOfRange(int[] original, int from, int to)` oder durch einfaches Durchlaufen und Kopieren der einzelnen Werte in ein neues Array. Um die schnellste Methode zu ermitteln wurden zu allen Möglichkeiten Benchmarks geschrieben und die Ergebnisse analysiert.

Auswertung der Experimente

Die Abbildung 4.1 zeigt die Messdaten der verschiedenen Kopiermethoden eines Arrays. `arrayCopy` und `copyOf` dauern identisch lange, `clone` sticht heraus. Da es sich dabei um Java-interne Methoden handelt ist es kaum nachvollziehbar, was genau passiert.

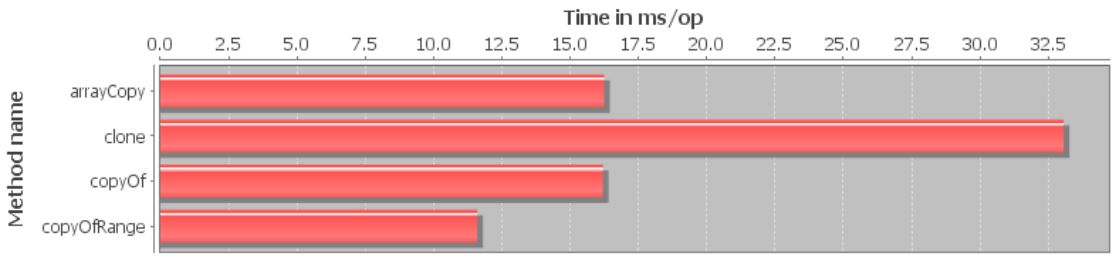

Abbildung 4.1: Kopieren eines Arrays

```
public static int[] clone_() {
    int[] copy = array.clone();
    return copy;
}
public static int[] arrayCopy() {
    int[] copy = new int[array.length];
    System.arraycopy(array, 0, copy, 0, array.length);
    return copy;
}
public static int[] copyOf() {
    int[] copy = Arrays.copyOf(array, array.length);
    return copy;
}
```

```
14      public static int[] copyRange() {
15          int[] copy = Arrays.copyOfRange(array, 2, 5);
16          return copy;
17      }
```

Listing 4.1: Arrays kopieren

Initialisieren von Arrays

Aufbau der Experimente

Des weiteren wurden zwei verschiedene Arten der Initialisierung eines Arrays mit Werten getestet, zum Einen die direkte Initialisierung bei der Erstellung des Arrays und zum Anderen die Initialisierung mittels einer Schleife nach der Erstellung.

Auswertung der Experimente

Bei diesem Experiment war die direkte Initialisierung deutlich schneller als die Initialisierung mittels einer Schleife (vgl. Abbildung 4.2).

Abbildung 4.2: Initialisieren und Durchlaufen eines Arrays

```
1       public static int[] loop() {
2           int[] copy = new int[array.length];
3           for (int i = 0; i < array.length; i++) {
4               copy[i] = array[i];
5           }
6           return copy;
7       }
8       public static int[] loopInit() {
9           int[] arrrray = new int[array.length];
10          for(int i=0; i<array.length;i++){
11              array[i] = i+1;
12          }
13          return array;
14      }
15      public static int[] normalInit() {
16          int[] array = {1, 2, 3, 4, 5, 6, 7, 8, 9};
17          return array;
18      }
```

Listing 4.2: Arrays initialisieren

Zusammenfassung der Resultate
Die Methoden arrayCopy() bzw. copyOf() sind einem clone() vorzuziehen. Ausserdem sollte bei kleineren Datenmengen die direkte Initalisierung verwendet werden.

4.2 Autoboxing

Aufbau der Experimente
Mit Java 5 wurde das automatische Boxing von Primitive-Types in Wrapper-Objekte (Gosling u. a., 2014, S. 102) in die JVM integriert. Diese Neuerung entlastet zwar den Entwickler, kann jedoch zu massiven Performanceeinbussen führen. Um den Einfluss des Autoboxings zu messen, wurden einfache Additionen mit verschiedenen Kombinationen von Primitive- und Wrapper-Types durchgeführt. Pro Aufruf der Testmethode werden 10 Additionen durchgeführt, bei welchen jeweils 10 Boxing bzw. Unboxing-Operationen erfolgen müssen. Als Referenzgrösse dient die Addition von zwei Primitive-Types.

Auswertung der Experimente
Wie aus Abbildung 4.3 hervorgeht, kann bei der Verwendung von Autoboxing massiv Zeit verloren gehen. Der Zeitunterschied bei den ersten beiden Werten ist so gering, dass die Operation als gleich effizient betrachtet werden kann. In der zweiten Methode (vgl. Listing 4.3) findet ein unboxing Vorgang statt. Dieses Experiment liefert auch beim Austauschen der beiden Summanden ein identisches Ergebnis.

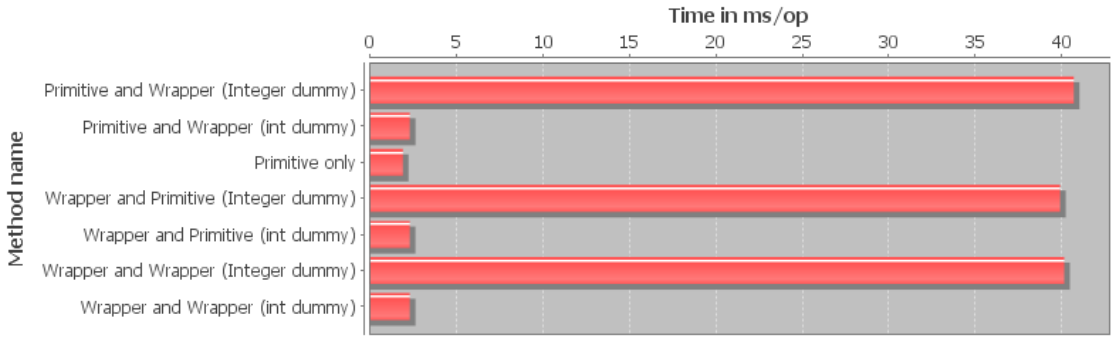

Abbildung 4.3: Autoboxing

```
public final static int primitiveOnly(int dummy, int add1, int add2) {
    for(int i = 0; i < 10; i++) {
        dummy += (add1 + add2);
    }
    return dummy;
}
public final static int primitiveToWrapper(int dummy, int add1, Integer
    add2) {
    for(int i = 0; i < 10; i++) {
        dummy += (add1 + add2);
    }
    return dummy;
}
```

Listing 4.3: Autoboxing - Good Practice

Das nächste Codebeispiel (vgl. Listing 4.4) enthält eine versteckte durch Autoboxing verursachte Object Creation. In beiden Methoden ist die Variable `dummy` vom Datentyp `Integer`. Dies bedeutet, dass bei jedem Aufruf von `dummy += (add1 + add2);` insgesamt drei (un)boxing Operationen durchgeführt werden müssen:

1. Unboxing von `dummy`
2. Unboxing von `add1` bzw. `add2`
3. Boxing von `dummy` (Object Creation)

Da ein `Integer` unveränderlich (immutable) ist, werden im gegebenen Beispiel bei jedem Schleifendurchgang zehn neue `Integer`-Objekte erstellt.

```
public final static int wrapperToPrimitive(Integer dummy, Integer add1,
    int add2) {
    for(int i = 0; i < 10; i++) {
        dummy += (add1 + add2);
    }
    return dummy;
}
public final static Integer wrapperToWrapper(Integer dummy, Integer add1,
    Integer add2) {
    for(int i = 0; i < 10; i++) {
        dummy += (add1 + add2);
    }
    return dummy;
}
```

Listing 4.4: Autoboxing - Bad Practice

Zusammenfassung der Resultate
Zuweisungen zu Wrapper-Types sollten vermieden werden.

4.3 Collections

Eine zentrale Frage während jedes Programmentwicklungsprozesses ist, welche Datenstrukturen verwendet werden sollten. Je nach Anwendung ist beispielsweise eine `LinkedList` einer `ArrayList` vorzuziehen. Um für einige Situationen die geeignetste Datenstruktur zu ermitteln wurden die folgenden Operationen jeweils auf Implementationen der Interfaces `java.util.List`, `java.util.Map` und `java.util.Set` angewandt und getestet.

Für Listen und Sets wurden alle Operationen, ausser das Einfügen von Daten, jeweils auf sortierte und auf unsortierte Daten angewandt. Die gemessenen Ergebnisse zeigen keine erwähnenswerten Unterschiede zwischen sortierten und unsortierten Listen auf. Sollte dies bei einzelnen Experimenten dennoch der Fall sein wird es explizit erwähnt.

4.3.1 Lists

Die folgenden Abschnitte beschreiben die Experimente für das Interface `List`. Getestet wurden folgende Implementationen: `ArrayList`, `CopyOnWriteArrayList`, `LinkedList`, `Stack` und `Vector`.

Einfügen von Daten

Aufbau der Experimente
Alle Implementationen des Interfaces `List` bieten die Möglichkeit Daten mittels `add()` entweder am Ende oder an einem bestimmten Index einzufügen. Zudem kann mit `addAll()` eine Collection mit Daten am Ende oder wiederum an einem bestimmten Index erweitert werden. Sämtliche Experimente wurden mit `ArrayLists` durchgeführt.
Das Einfügen wurde mit der normalen `add()`-Methode getestet und zudem mit einem Index am Anfang, in der Mitte und am Ende.

Auswertung der Experimente
Die nachfolgenden Abbildung 4.4 illustriert, dass alle Implementationen bis auf `CopyOnWriteArrayList` gleich lange benötigen. Dies ist darauf zurückzuführen, dass bei `CopyOnWriteArrayListen`, wie der Name schon andeutet, bei jeder Schreiboperation eine Kopie der gesamten Liste erstellt wird.

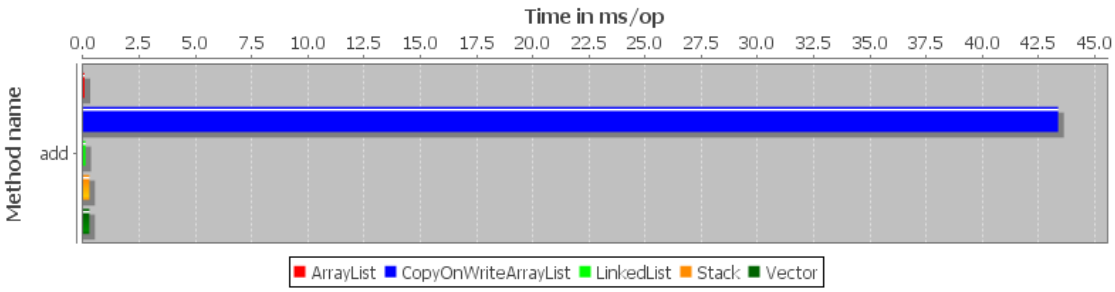

Abbildung 4.4: Einzelne Werte einfügen

Um den Zeitunterschied zwischen den restlichen Listen noch einmal genauer zu erläutern wurde die Abbildung 4.5 erstellt. Die Zeitunterschiede zwischen `ArrayList`, `LinkedList`, `Stack` und `Vector` sind darauf zurückzuführen, dass `Vector` im Gegensatz zu `ArrayList` oder `LinkedList` synchronized ist. Das Selbe gilt für `Stack`, der seinerseits eine Unterklasse von `Vector` ist. Oracle empfiehlt die Benutzung einer `ArrayList` anstelle von `Vector`, falls Thread-Sicherheit keine Anforderung ist (Oracle and/or its affiliates, o.J).

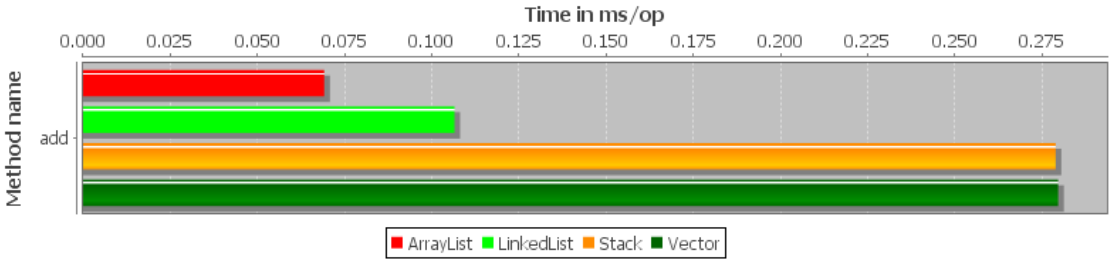

Abbildung 4.5: Einzelne Werte einfügen (ohne CopyOnWriteArrayList)

Auch beim Einfügen mit Index sticht die `CopyOnWriteArrayList` auf Grund ihrer Eigenheit heraus. Weiterhin ist interessant, dass das Einfügen in der Mitte einer Liste bei `LinkedList` sogar länger dauert als bei einer `CopyOnWriteArrayList` (siehe Abbildung 4.6). Dies ist darauf zurückzuführen, dass die `LinkedList` bei jedem Einfügen entweder von Beginn oder vom Ende her durchlaufen werden muss.

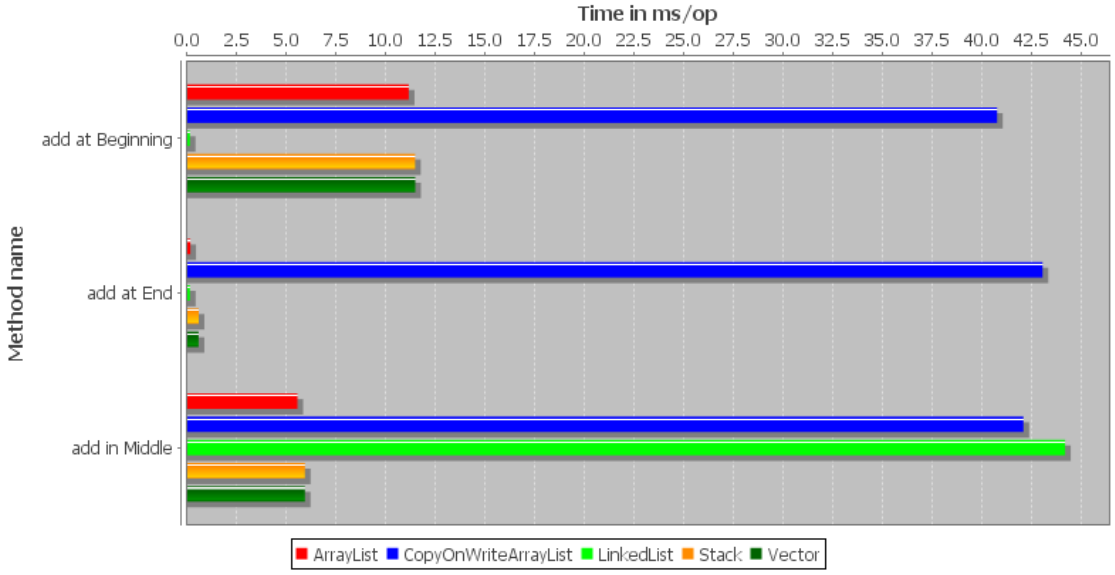

Abbildung 4.6: Einzelne Werte einfügen mit Index

Bei näherer Betrachtung von `ArrayList`, `Stack` und `Vector` wird ersichtlich, dass eine `ArrayList` meist etwas schneller ist (vgl. Abbildung 4.7). Dies ist wohl auch hier auf eine optimiertere Umsetzung zurückzuführen.

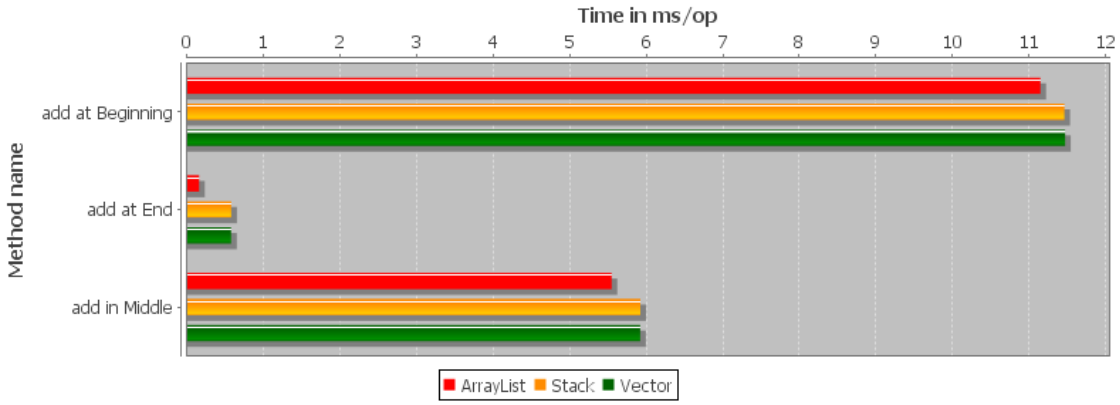

Abbildung 4.7: Einzelne Werte einfügen mit Index (ohne CopyOnWriteArrayList und LinkedList)

Da der Unterschied beim Einfügen am Ende aus der Abbildung 4.9 nicht klar erkennbar ist wurde dafür noch eine separate Abbildung 4.8 erstellt. Die `LinkedList` benötigt im Durchschnitt die kürzeste Zeit, da es sich hierbei immer nur um ein Umhängen von Indices handelt.

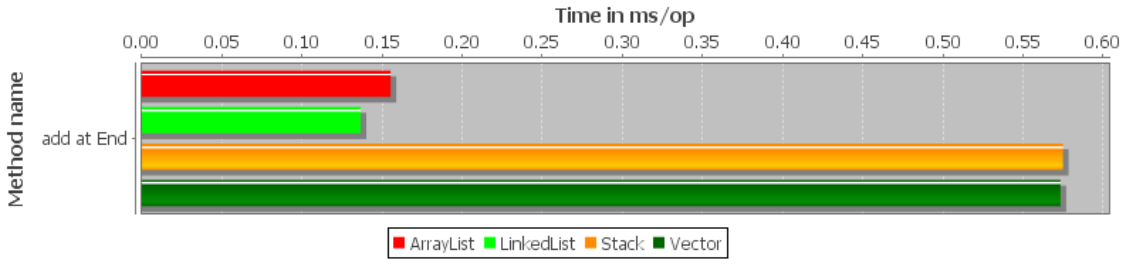

Abbildung 4.8: Einzelne Werte am Ende einfügen (ohne CopyOnWriteArrayList)

Beim Einfügen einer `ArrayList` von Daten wird im Verhältnis gleich viel Zeit gebraucht wie beim Einfügen einzelner Werte. Einzig die `LinkedList` benötigt länger (siehe Abbildung 4.9), da die übergebene Collection zuerst durchlaufen und mit Indexverknüpfungen versehen werden muss.

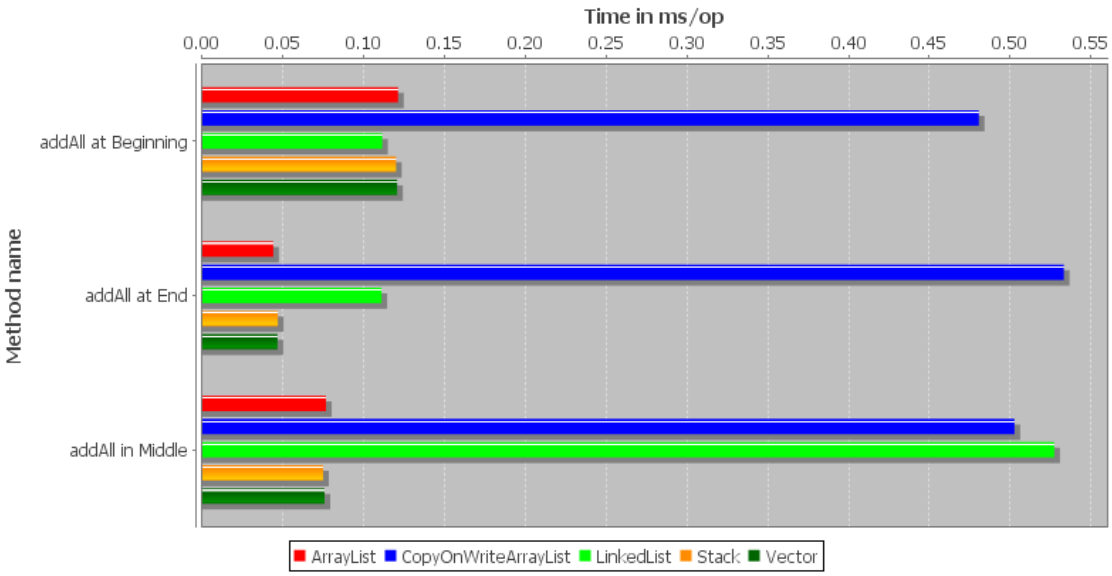

Abbildung 4.9: ArrayList einfügen mit Index

Lesen / Suchen von Daten

Aufbau der Experimente

Das Lesen und Suchen von Daten in Listen wurde mit insgesamt fünf Experimenten getestet. Zum Ersten das Lesen aller Werte mittels dem Index (Methode: `get()`). Als Zweites das Überprüfen ob ein bestimmter Wert bereits in der Liste vorhanden ist (Methode: `contains()`). Drittens das bestimmen eines Index für einen gegebenen Wert am Anfang, in der Mitte und am Ende der Liste (Methode: `indexOf()`) und Viertens der Zeitaufwand zum Erstellen von Sublisten (Methode: `subList()`). Das fünfte Experiment umfasst verschiedene Arten für das Durchlaufen von Listen.

Auswertung der Experimente

Das Lesen eines Wertes mittels `get()` dauert bei einer `LinkedList` massiv länger (vgl. Abbildung 4.10), da ein direkter Indexzugriff nicht möglich ist. Die Liste muss bei jedem Aufruf entweder von Beginn oder Ende aus durchlaufen werden bis das gewünschte Element gefunden wurde.

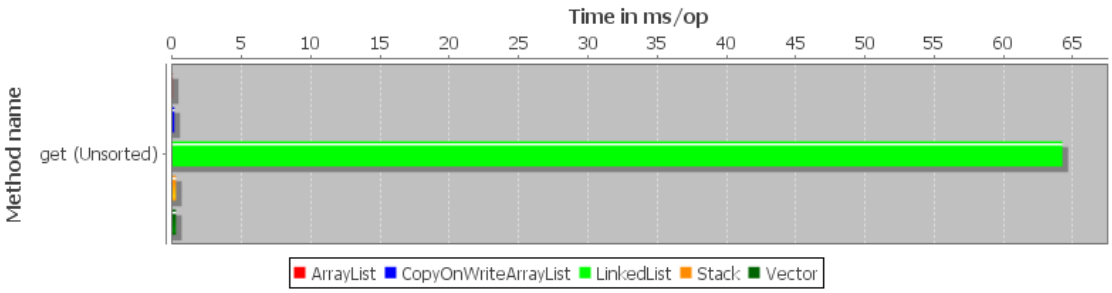

Abbildung 4.10: Werte lesen

Ohne `LinkedList` betrachtet wird ersichtlich, dass ein `get()` einer `ArrayList` am schnellsten ist (vgl. Abbildung 4.11).

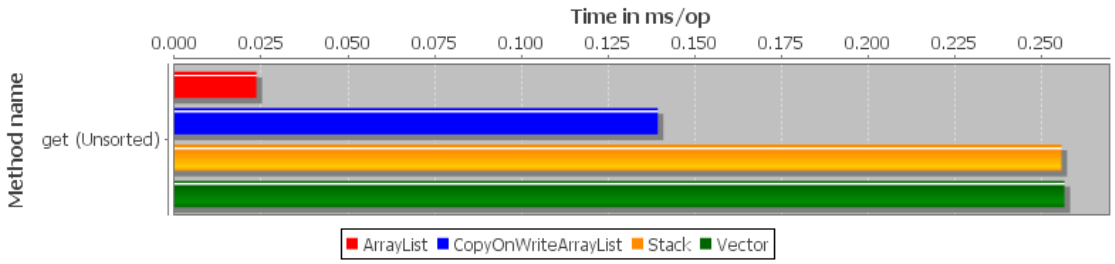

Abbildung 4.11: Werte lesen (ohne LinkedList)

In den gemessenen Resultaten konnte kein Unterschied zwischen dem Erstellen eine Subliste an verschiedenen Indices festgestellt werden (siehe Abbildungen 4.12, 4.13, 4.14). Generell kann gesagt werden, dass eine `ArrayList` am wenigsten Zeit benötigt.

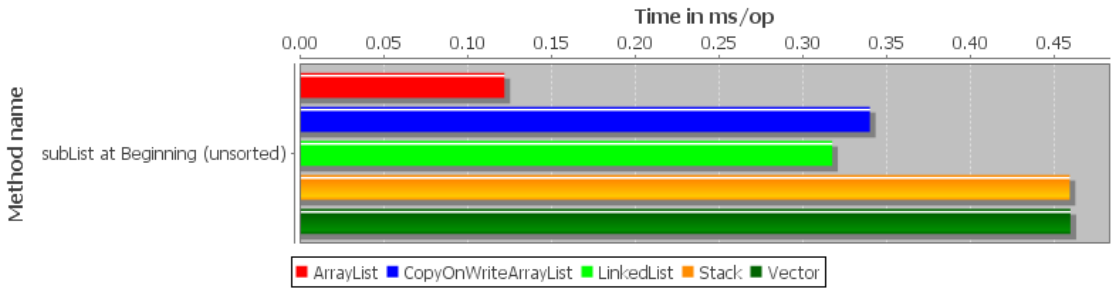

Abbildung 4.12: Unterliste erstellen am Anfang

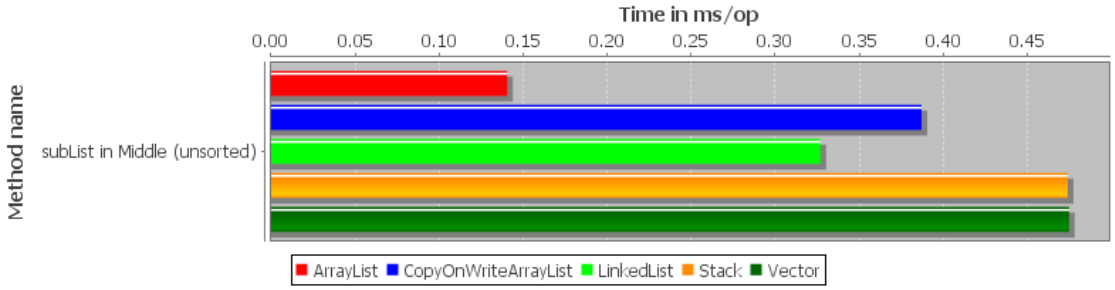

Abbildung 4.13: Unterliste erstellen in der Mitte

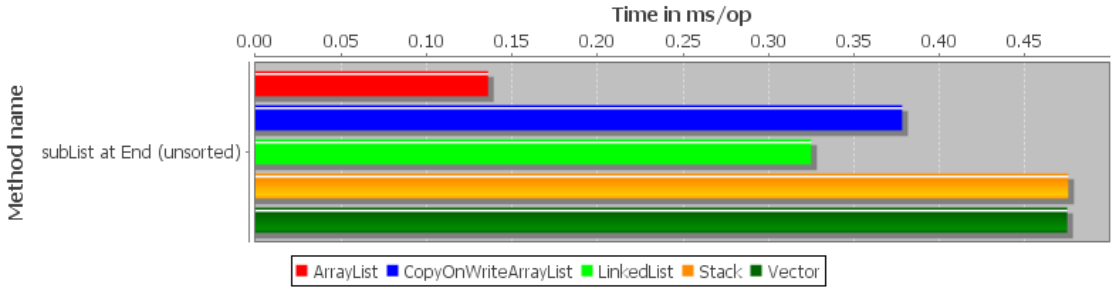

Abbildung 4.14: Unterliste erstellen am Ende

Das Ermitteln des Index eines Elementes liegen `ArrayList`, `Stack` und `Vector` fast gleich auf (siehe Abbildungen 4.15, 4.16, 4.17). Interessant ist hierbei, dass eine `CopyOnWriteArrayList` verglichen mit einer normalen `ArrayList` deutlich langsamer ist. Das Ermitteln des Index eines Elementes in der Mitte der Liste ist bei sortierten Daten schneller.

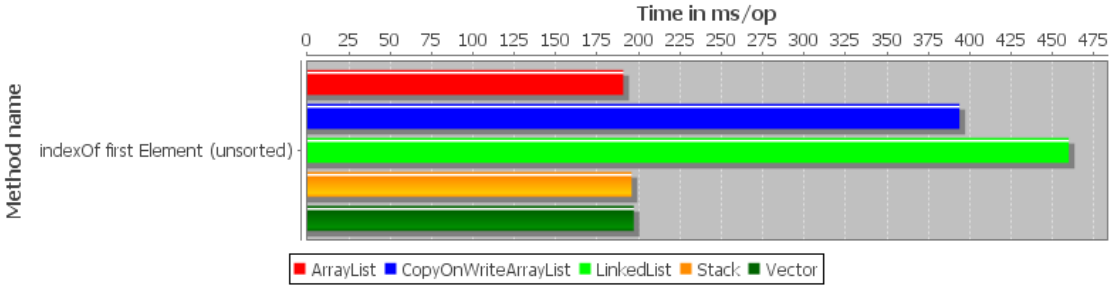

Abbildung 4.15: Index des ersten Elements finden

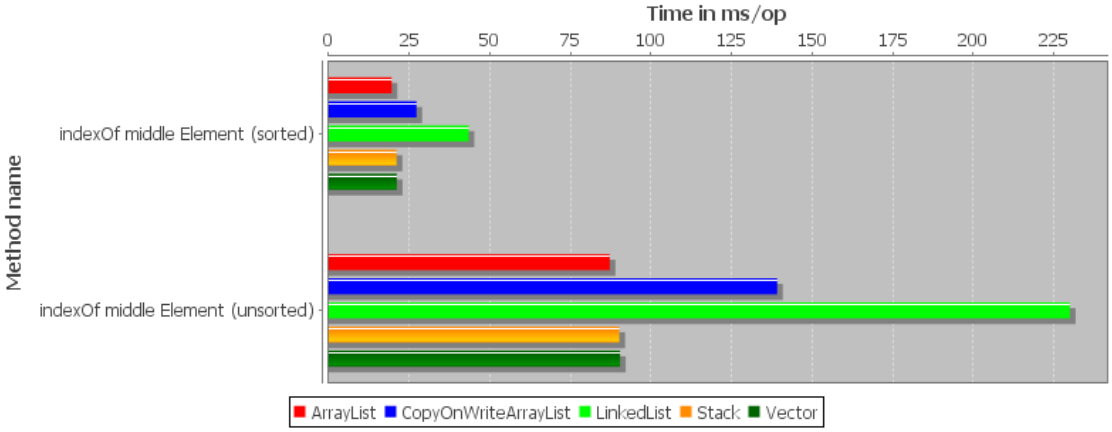

Abbildung 4.16: Index des mittleren Elements finden

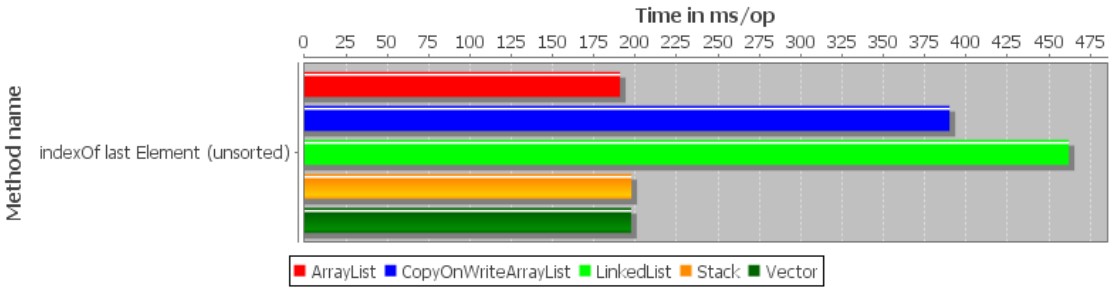

Abbildung 4.17: Index des letzten Elements finden

Das Suchen eines Wertes dauert bei `ArrayList`, `Stack` und `Vector` gleich lange. Auffällig ist auch hierbei, dass eine `CopyOnWriteArrayList` und eine `LinkedList` mehr als doppelt so lange benötigen (siehe Abbildung 4.18).

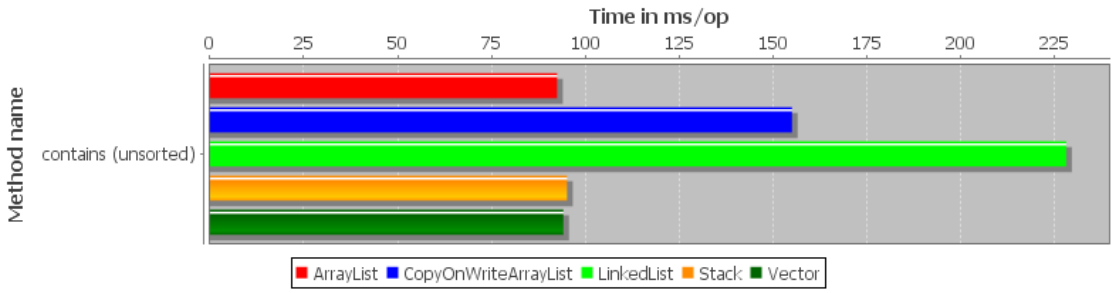

Abbildung 4.18: Suchen nach Wert

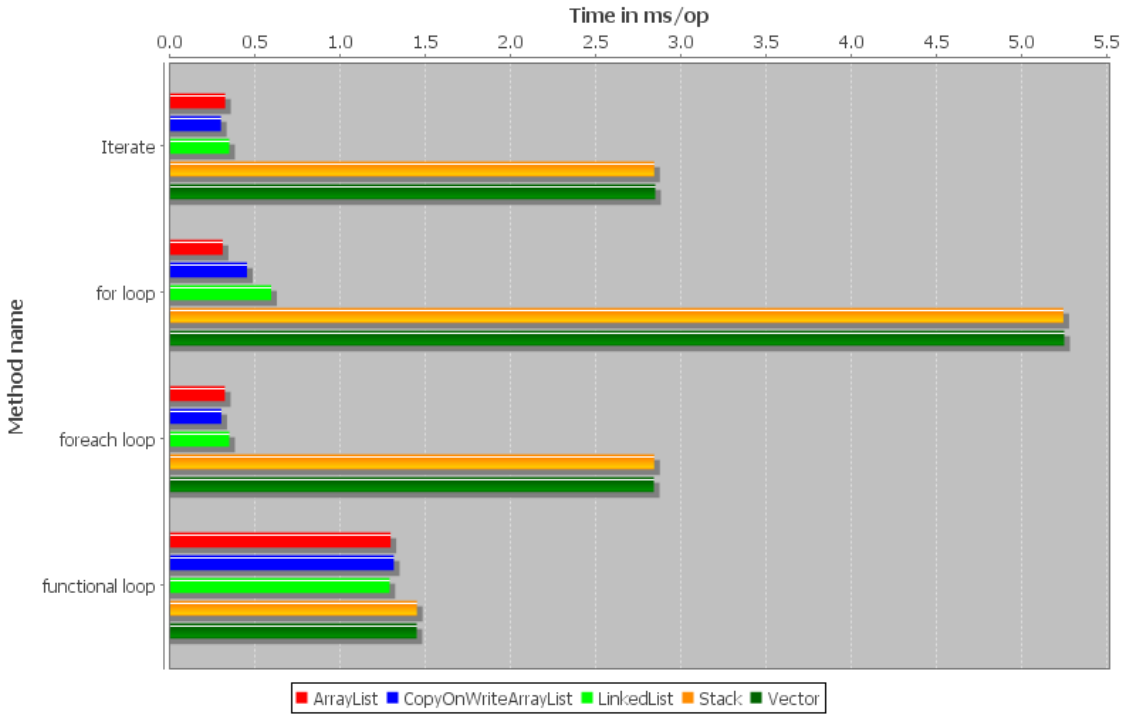

Abbildung 4.19: Listen durchlaufen

Eine `foreach` Schleife und ein Iterator benötigen für das Traversieren aller Listen gleich lange. Dies ist darauf zurückzuführen, dass die Referenz auf das nächste Element jeweils schon bekannt ist (vgl. Abbildung 4.19). Eine `for` Schleife benötigt minimal länger. Aus diesem Grund sollte situationsbedingt entschieden werden.

```
    public long forEachLoop(long dummy) {
        for (Integer list1 : loopList) {
            dummy += list1;
        }
        return dummy;
    }
    public long forLoop(long dummy) {
        for (int i = 0; i < loopList.size(); i++) {
            dummy += loopList.get(i);
        }
        return dummy;
    }
```

```
    public long functionalLoop(long dummy) {
        dummy = loopList.stream().map((list1) -> list1).reduce((int) dummy,
            Integer::sum);
        return dummy;
    }
    public long iterate(long dummy) {
        Iterator<Integer> iterator = loopList.iterator();
        while (iterator.hasNext()) {
            dummy += iterator.next();
        }
        return dummy;
    }
```

Listing 4.5: Traversieren einer Liste

Verändern von Daten

Aufbau der Experimente

Bei den folgenden Experimenten wurden die `set()`-, `replaceAll()`-Methode am Anfang, in der Mitte und am Ende sowie die `retainAll()`-Methode mit einer Unterliste, deren Länge sich bei jedem Durchlauf um eins verringert getestet.

Auswertung der Experimente

Wie schon bei den anderen Schreiboperationen hat auch hier die `CopyOnWriteArrayList` die schlechteste Performance. Das Einfügen am Anfang und am Ende hat die selbe Laufzeit (siehe Abbildung 4.20).

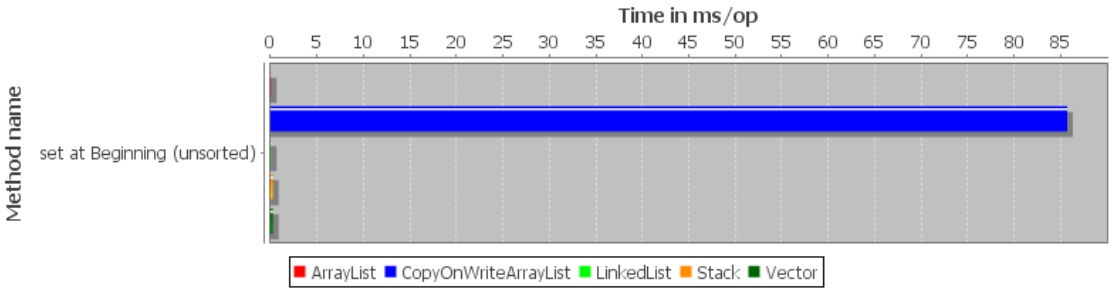

Abbildung 4.20: Wert am Anfang setzen

Bei isolierter Betrachtung (siehe Abbildung 4.21) sind `ArrayList` und `LinkedList` fast gleich schnell. `Stack` und `Vector` um ein Vielfaches langsamer.

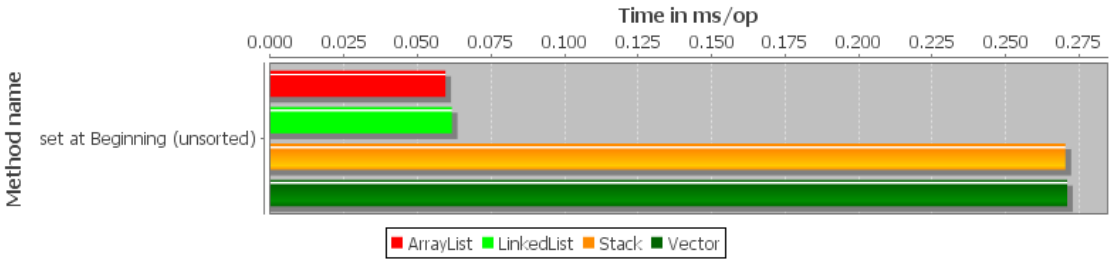

Abbildung 4.21: Wert am Anfang setzen (ohne CopyOnWriteArrayList)

Beim neu Setzen von Werten in der Mitte weisst `LinkedList` eine deutlich schlechtere Performance als `CopyOnWriteArrayList` (siehe Abbildung 4.22). Dies ist wiederum darauf zurückzuführen, dass auch hier kein direkter Indexzugriff möglich ist.

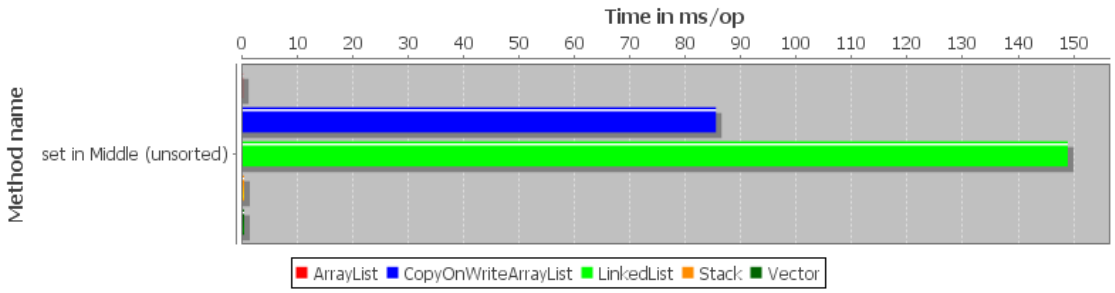

Abbildung 4.22: Wert in Mitte setzen

Der direkte Vergleich von `ArrayList`, `Stack` und `Vector` zeigt die selben Resultate wie beim Einfügen am Anfang oder Ende (siehe Abbildung 4.23).

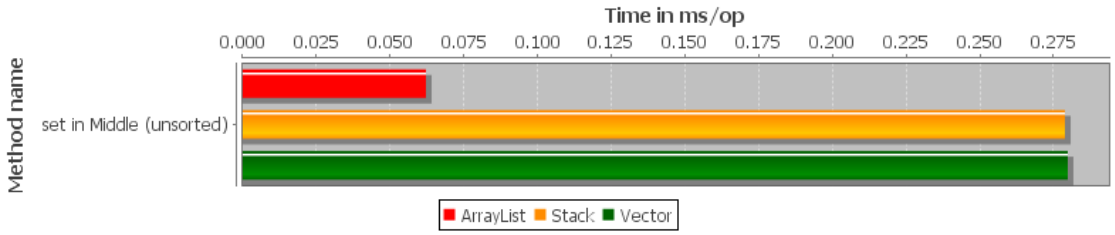

Abbildung 4.23: Wert in Mitte setzen (ohne CopyOnWriteArrayList und LinkedList)

Das Ersetzen einer `ArrayList` von Werten benötigt für Anfang, Mitte und Ende gleich lange. Die Abbildung 4.24 dient exemplarisch zum Erklären der Werte. Auch hier erreicht `ArrayList` die besten Werte.

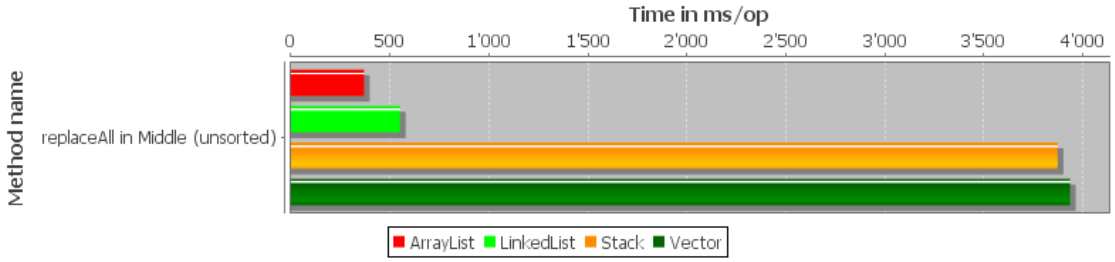

Abbildung 4.24: ArrayList mit Werten in der Mitte ersetzen (ohne CopyOnWriteArrayList)

Beim Behalten einer `ArrayList` von Daten sollte eine `ArrayList`, `CopyOnWriteArrayList` oder `LinkedList` verwendet werden, da die Performance von `Stack` und `Vector` deutlich schlechter ist (siehe Abbildung 4.25).

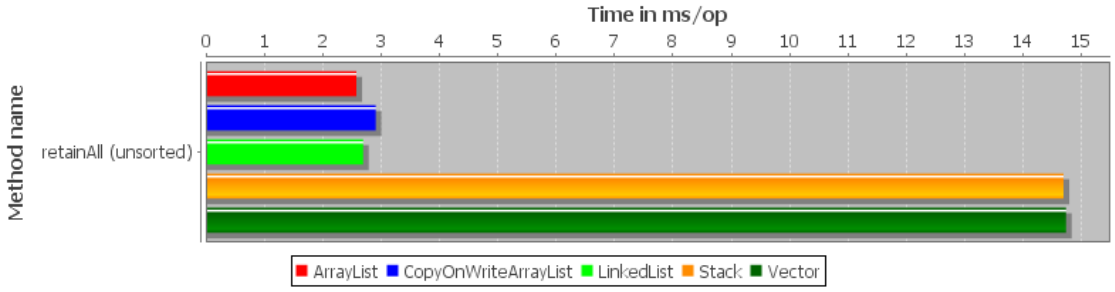

Abbildung 4.25: Werte behalten

Sortieren von Daten

Aufbau der Experimente

Da bei allen Operationen der Unterschied zwischen einer sortierten und einer unsortierten Liste gemessen wurde, liegt es nahe auch die Kosten der Java-internen `sort()`-Methode zu testen. So sollte verifiziert werden, ob es sich lohnt eine Liste vor allfälligen Operationen noch zu sortieren oder nicht.

Auswertung der Experimente

Das Sortieren von `ArrayList`, `Stack` und `Vector` dauert gleich lange, bei `CopyOnWriteArrayList` durch die zusätzlichen Kopieroperationen länger. Die `LinkedList` benötigt durch das notwendig Umhängen der Indices zusätzliche Zeit (siehe Abbildung 4.26).

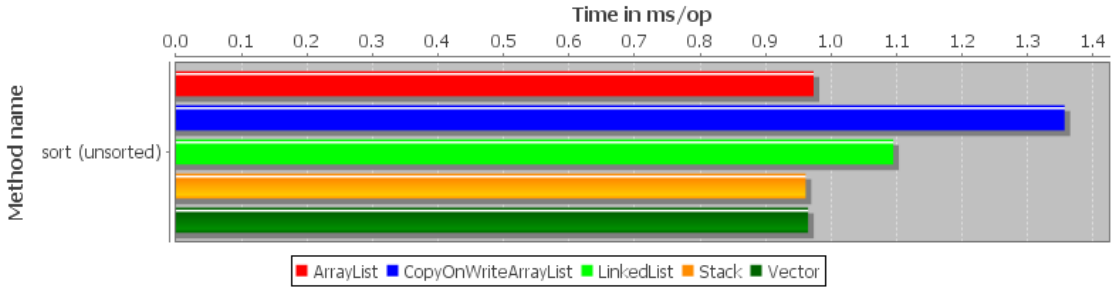

Abbildung 4.26: Listen sortieren

Löschen von Daten

Aufbau der Experimente

In Java gibt es die Möglichkeit Daten entweder direkt, mit einem Index oder mit einem Wert, welcher in der Liste gesucht wird, aus einer Liste zu entfernen. Zudem kann eine ganze `ArrayList` von Werten aus der Liste entfernt werden. Auch hier wurde mit Indizes am Anfang, in der Mitte und am Ende gearbeitet. Des weiteren wurde mit der Methode `clear()` das komplette Leeren der Liste getestet.

Auswertung der Experimente

Das Löschen von Werten am Anfang ist die grosse Stärke von `LinkedList`, da hier lediglich die Referenz auf das erste Element neu gesetzt werden muss. Bei `ArrayList` muss das dahinterliegende `Array` vollständig kopiert werden (siehe Abbildung 4.27).

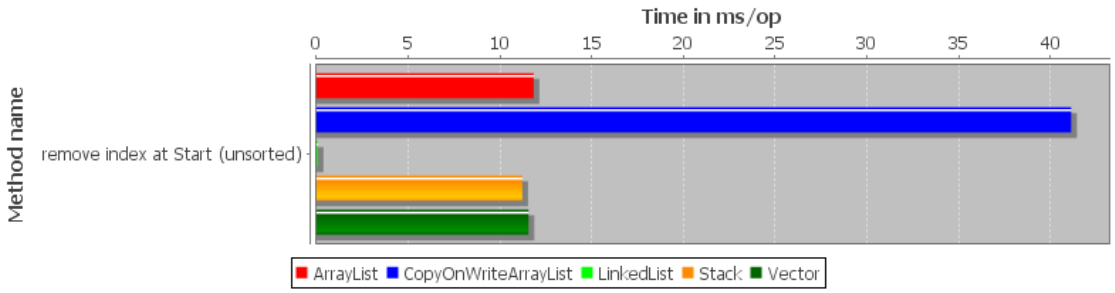

Abbildung 4.27: Werte am Anfang entfernen

Aus den schon mehrmals genannten Gründen ist auch das Löschen von Werten in der Mitte einer `LinkedList` sehr viel langsamer (siehe Abbildung 4.28).

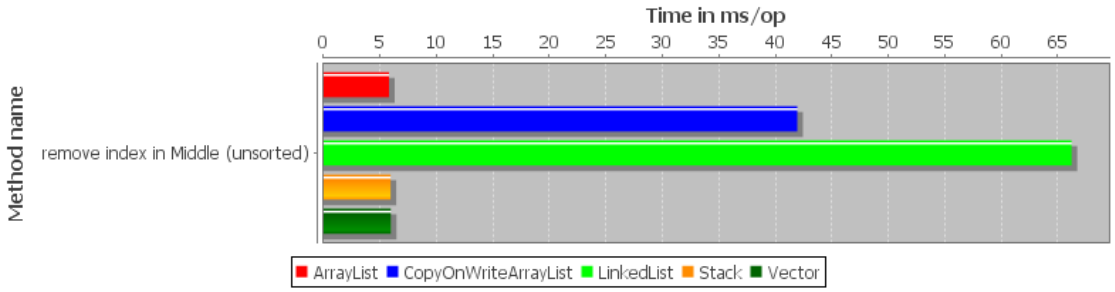

Abbildung 4.28: Werte in der Mitte entfernen

Das Löschen eines Wertes am Ende der Liste geht bei allen ausser `CopyOnWriteArrayList` vergleichbar kurz (siehe Abbildung 4.29).

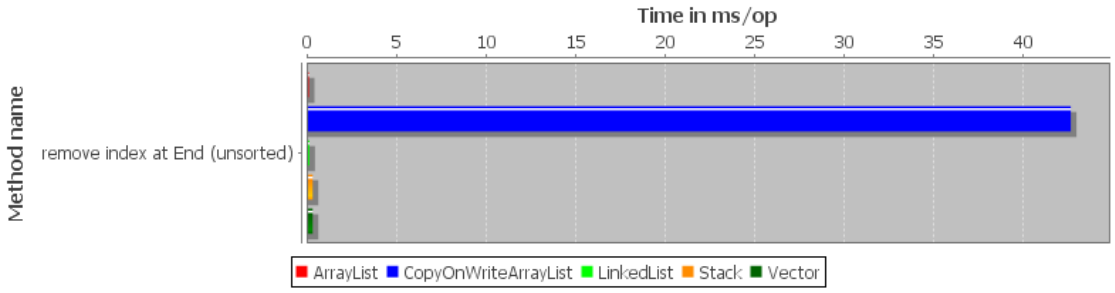

Abbildung 4.29: Werte am Ende entfernen

Um einen bestimmten Wert aus der Liste zu löschen benötigen `ArrayList`, `Stack` und `Vector` identisch viel Zeit. Auffällig ist auch hier die massiv längere Laufzeit von `LinkedList` (siehe Abbildung 4.30).

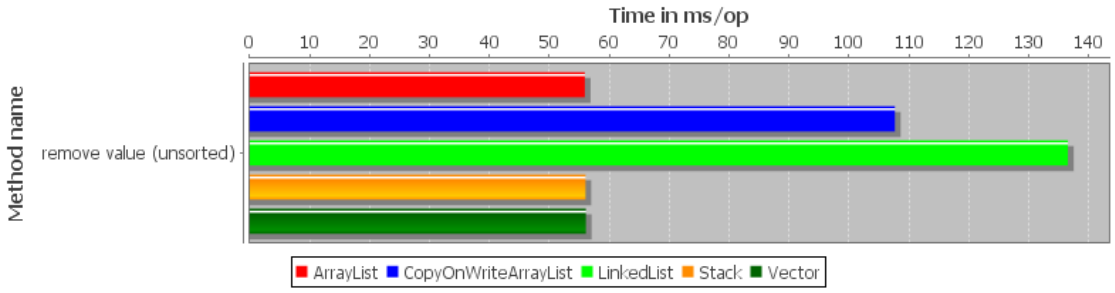

Abbildung 4.30: Werte suchen und entfernen

Für das komplette Leeren der Liste benötigt `LinkedList` zehn Mal so lange wie die anderen Implementationen (siehe Abbildung 4.31).

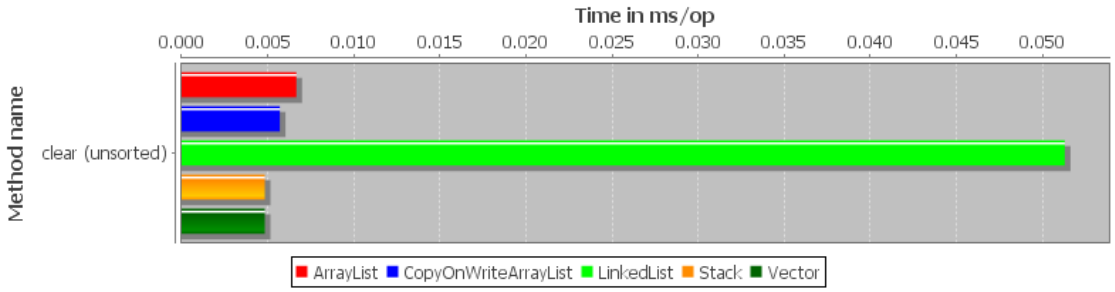

Abbildung 4.31: Listen leeren

Zusammenfassung der Resultate
Für den generellen Gebrauch sollte immer eine `ArrayList` verwendet werden. In einem synchronized Kontext kann stattdessen auf `Vector` zurückgegriffen werden, da dieser dieselbe Funktionalität bietet. Bei häufigen Einfüge- bzw. Löschoperationen am Anfang oder Ende ist eine `LinkedList` empfehlenswert. Eine `CopyOnWriteArrayList` sollte nur in Systemen die lückenlos konsistente Daten aufweisen müssen verwendet werden. Für das Durchlaufen einer Liste sollte kein functional Loop verwendet werden.

4.3.2 Maps

Die folgenden Abschnitte beschreiben die Experimente für das Interface `Map`. Getestet wurden folgende drei Implementationen: `HashMap`, `TreeMap` und `Hashtable`.

Einfügen von Daten

Aufbau der Experimente
Im Gegensatz zu Listen werden Werte in Maps immer als <Key, Value>-Tupel abgebildet. Zum Einfügen von Daten in eine Map stehen die beiden Methoden `put()` und `putIfAbsent()` zur Verfügung. Bei der ersten Methode wird ein existierender Key ersetzt und der ursprüngliche Wert zurückgegeben. Existiert der Key nicht wird dieser eingefügt und der Rückgabewert ist `null`. Bei der zweiten Methode werden nur nicht vorhandene Keys eingefügt. Die Methoden `put()` und `putIfAbsent()` wurden jeweils auf eine

leere Map angewandt, die Methode `putIfAbsent()` zudem auf eine kleine und eine grosse Map mit existierenden Werten.

Auswertung der Experimente
Wie Abbildung 4.32 zeigt, ist das Einfügen in eine `HashMap` mit Abstand am schnellsten, während die selbe Operation mit einer `TreeMap` am meisten Zeit beansprucht. Dies ist darauf zurückzuführen, dass das Einfügen in eine Hash-Tabelle, wie sie bei der `HashMap` verwendet wird, eine Laufzeit $O(1)$ hat. Im Gegensatz dazu werden bei der `TreeMap` die Elemente in einen Binärbaum eingefügt, was eine Laufzeit von $O(log(n))$ bedeutet.
Die veraltete Map-Implemenation `Hashtable` ist im Gegensatz zu `HashMap` synchronized, was die Performance deutlich mindert.

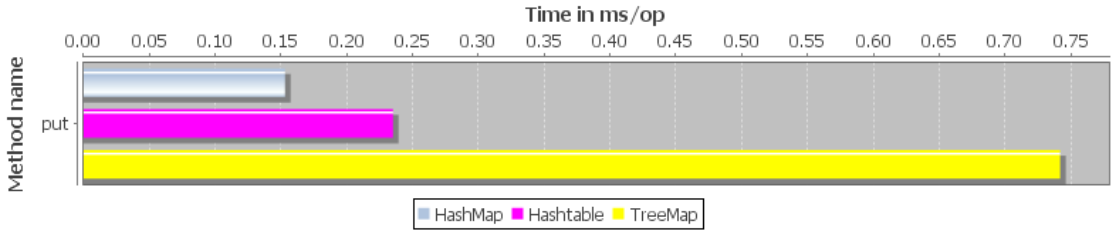

Abbildung 4.32: Einfügen in Map

Die Ergebnisse zeigen, dass die Methode `putIfAbsent()` keinen Nachteil in der Geschwindigkeit haben. Bemerkenswert ist, dass die `TreeMap` bei sehr kleinen Datenmengen und bereits existierenden Werten sogar deutlich schneller sein kann als die `HashMap` (vgl. Abbildung 4.33).

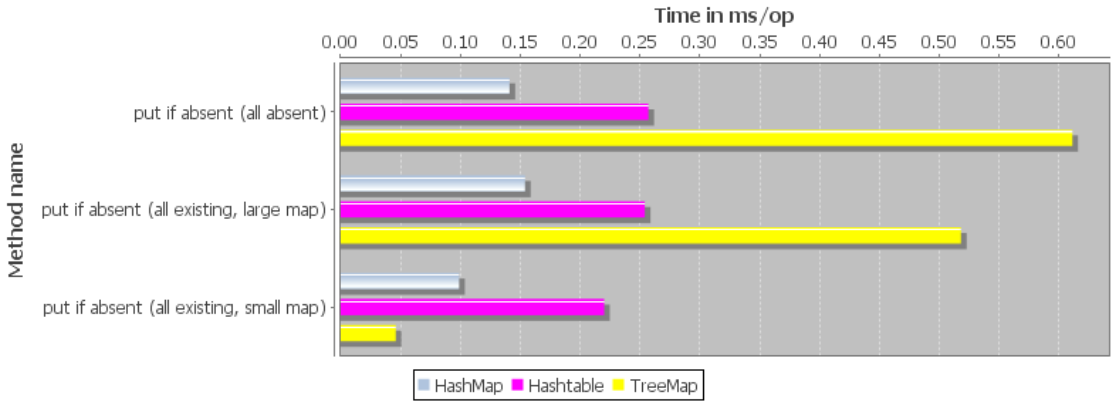

Abbildung 4.33: Einfügen falls nicht vorhanden

Lesen / Suchen von Daten

Aufbau der Experimente
Mit Hilfe der Methode `get()` kann für einen Key der dazugehörige Wert gelesen werden. Da die Keys jeweils dem Index von Beginn bis Ende entsprechen wurden alle Werte der Map gelesen. Mit `containsValue()` kann die Map auf einen bestimmten Wert überprüfen werden. Zusätzlich existiert noch die Methode `containsKey()` um die Map auf einen bestimmten Key zu überprüfen. Die beiden Methoden wurden mit Werten getestet, welche zu Beginn, gegen Mitte und gegen Ende in die Liste eingefügt wurden.

Auswertung der Experimente

Wie beim Einfügen von Werten ist die `HashMap` bei allen drei Tests durchschnittlich am schnellsten. Grafiken wurden nur für das Suchen nach Schlüsseln (Abbildung 4.34) und das Suchen nach Werten (Abbildung 4.35) eingefügt, da das Lesen von Werten exakt gleich lange dauert wie das Suchen nach Schlüsseln.
Die Resultate der `TreeMap` lassen darauf schliessen, dass der interne Binärbaum nicht ausbalanciert ist. Tatsächlich war die erste während dem Experiment eingefügte Zahl 3'001, bei möglichen Werten zwischen 0 und 10'000. Bei angenommener Gleichverteilung der Werte entsteht somit ein rechtslastiger Baum dessen rechte Hälfte doppelt so viele Werte enthält wie die linke.

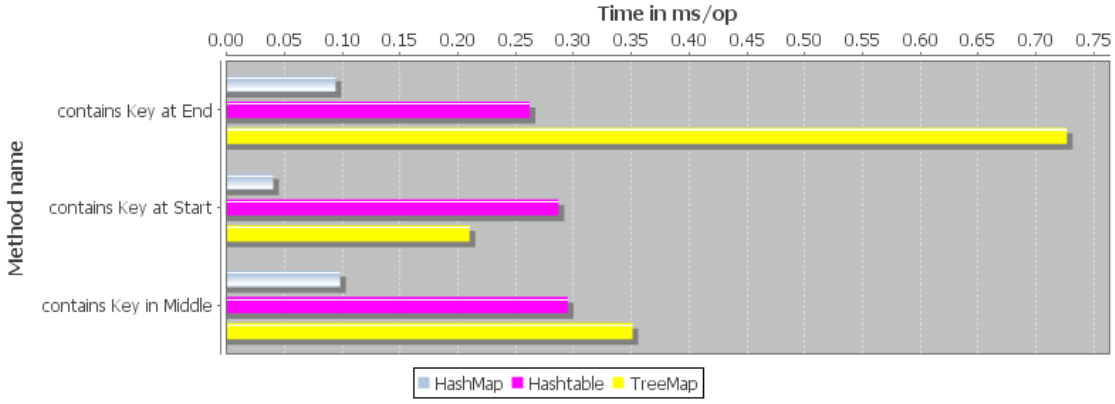

Abbildung 4.34: Suchen nach Schlüssel

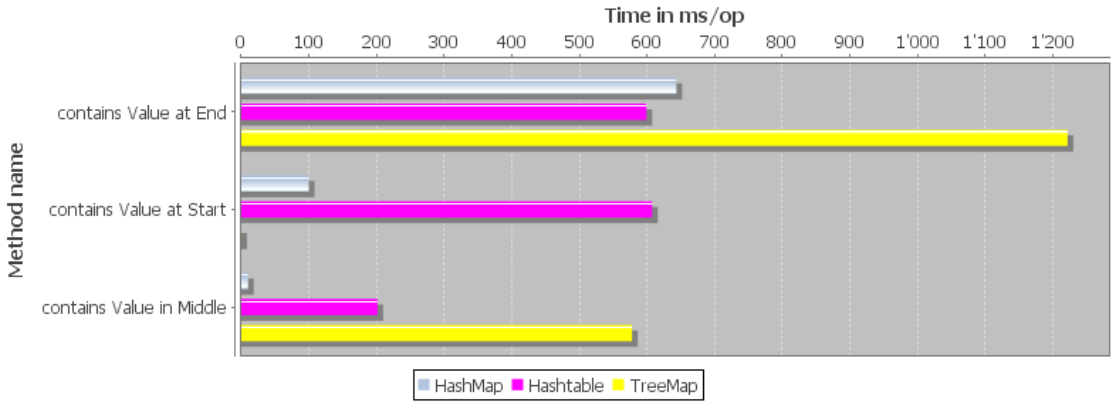

Abbildung 4.35: Suchen nach Wert

Verändern von Daten

Aufbau der Experimente

Neben der bereits erwähnten Methode `put()` gibt es noch `replace()`, wobei der Wert nur falls der Key vorhanden ist ersetzt wird, nicht vorhandene Keys werden nicht eingefügt. Zusätzlich gibt es noch die Möglichkeit einen Wert eines Keys nur zu ersetzen, falls er einen bestimmten Wert hat. Als Letztes gibt es noch die Methode `replaceAll()`, mit der alle Werte einer Map durch den Rückgabewert einer angegebenen Funktion ersetzt werden können. Die `replace()`-Methode wurde jeweils am Anfang, am Ende und zum Vertauschen verschiedener Werte angewandt. Bei `replaceAll()` wurden beide Varianten getestet.

Auswertung der Experimente
Auch beim Ersetzen macht sich bemerkbar, dass die `TreeMap` nicht ausbalanciert ist. Jedoch ist selbst bei früh eingefügten Werten die Performance der `HashMap` besser als diejenige der `TreeMap` (siehe Abbildung 4.36).

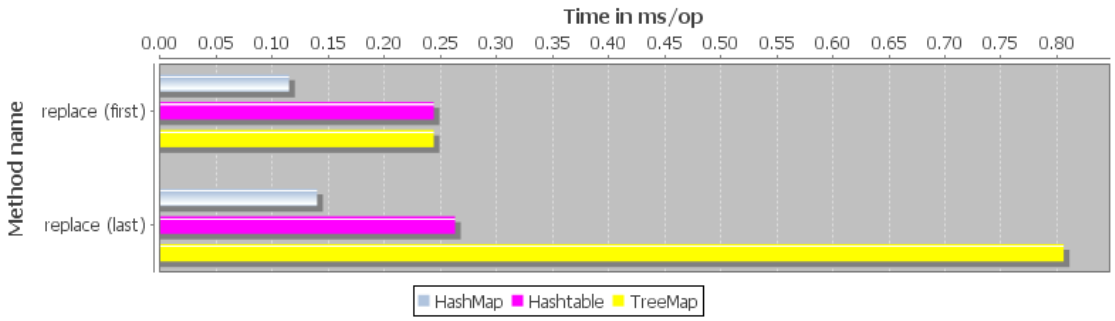

Abbildung 4.36: Einzelne Werte ersetzen

Löschen von Daten

Aufbau der Experimente
Anders als bei Listen können bei einer Map ausschliesslich einzelne <Key, Value> Tupel unter Angabe des Keys gelöscht werden. Zusätzlich kann der erwartete Wert angegeben werden, wodurch der Eintrag nur entfernt wird falls der Wert übereinstimmt. Getestet wurde das Entfernen von existierenden sowie nicht vorhandenen <Key, Value>-Paaren in der Map. Als Letztes wurde das komplette Leeren der Map gemessen.

Auswertung der Experimente
Abbildung 4.37 steht exemplarisch für alle Messungen bezüglich dem Löschen von Daten aus Maps. Getestet wurde das Entfernen von Schlüsseln die bereits gelöscht wurden. Wie schon bei den vorherigen Experimenten erreicht die `HashMap` bessere Werte als `HashTable` und `TreeMap`.

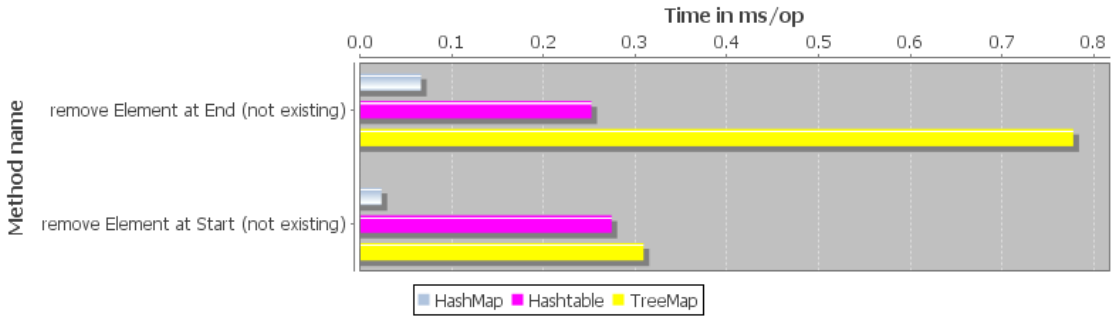

Abbildung 4.37: Löschen von nicht existenten Werten

Zusammenfassung der Resultate
Eine `HashMap` erreicht bei sämtlichen Anwendungsfällen bessere Performance als `Hashtable` und `TreeMap`.

4.3.3 Sets

Die folgenden Abschnitte beschreiben die Experimente für das Interface `Set`. Getestet wurden die beiden Implementationen `HashSet` und `TreeSet`.

Einfügen von Daten

Aufbau der Experimente
Die letzte analysierte Collection ist das `Set`-Interface. Es stehen wie schon bei den Listen die Methoden `add()` und `addAll()` zur Verfügung. Da ein Set keine Ordnung enthält gibt es auch nicht die Möglichkeit einen Wert an einem bestimmten Index einzufügen. Die Methoden `add()` und `addAll()` wurden auf ein leeres Set angewandt und die Methode `add()` zusätzlich auf ein sortiertes und ein unsortiertes Set mit existierenden Daten. Sortiert bzw. unsortiert bedeutet in diesem Zusammenhang lediglich die Reihenfolge in welcher die Werte eingefügt wurden.

Auswertung der Experimente
Die folgenden drei Abbildungen (Abbildung 4.38, 4.39, 4.40) zeigen, dass ein `HashSet` bei sämtlichen zur Verfügung stehenden Operationen zum Einfügen von Werten um ein Vielfaches schneller ist.

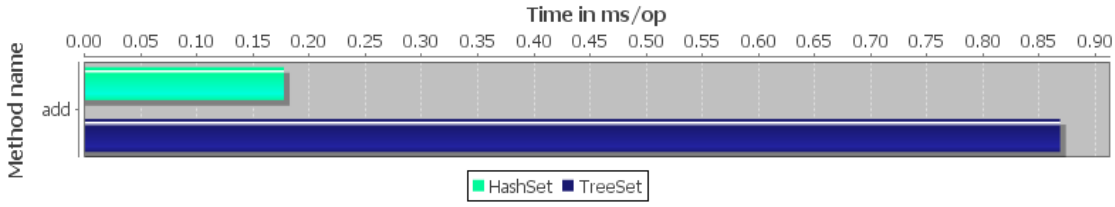

Abbildung 4.38: Werte einfügen

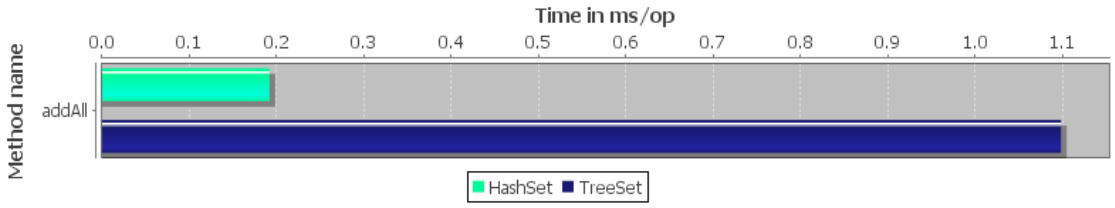

Abbildung 4.39: ArrayList mit Werten einfügen

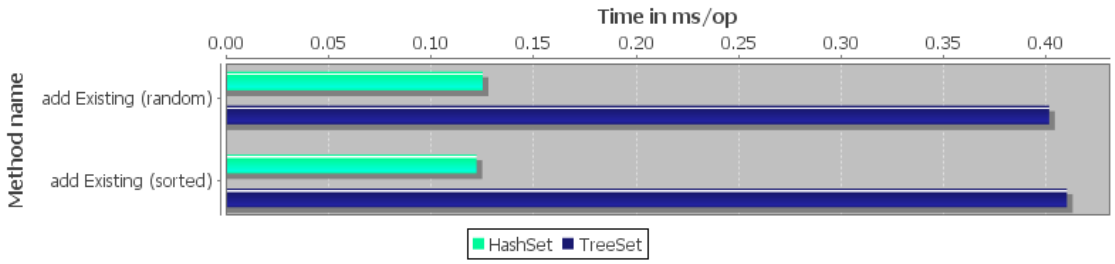

Abbildung 4.40: Bereits existierende Werte einfügen

Suchen von Daten

Aufbau der Experimente

Für Sets existieren lediglich die beiden Methoden `contains()` und `containsAll()` um zu ermitteln, ob ein gegebener Wert oder eine Collection (alle Experimente wurden mit einer `ArrayList` durchgeführt) von Werten bereits vorhanden ist. Ein Lesen ohne den Wert aus dem Set zu entfernen existiert anders als bei Listen und Maps nicht. Die Methoden wurden wiederum auf ein sortiertes und ein unsortiertes Set angewandt.

Auswertung der Experimente

Wie bereits beim Einfügen ist auch beim Suchen das `HashSet` deutlich schneller. Dies kann auf die Baumstruktur von `TreeSet` zurückgeführt werden, welche verglichen mit einer Hash-Tabelle im Durchschnitt immer langsamer ist.

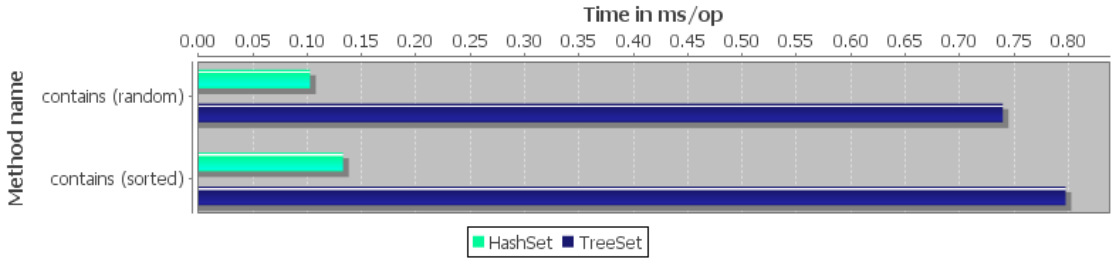

Abbildung 4.41: Überprüfung, ob Wert vorhanden

Verändern von Daten

Aufbau der Experimente

Ein Set kann mit Hilfe von `retainAll()` auf eine `ArrayList` von Werten reduziert werden. Dies wurde wie bei den Listen mit einer Unterliste, deren Länge sich bei jedem Durchlauf um eins verringert getestet.

Auswertung der Experimente

Auch bei diesem Experiment schneidet das `HashSet` besser ab als das `TreeSet` (siehe Abbildung 4.42), wenn auch nicht so deutlich wie bei anderen Tests.

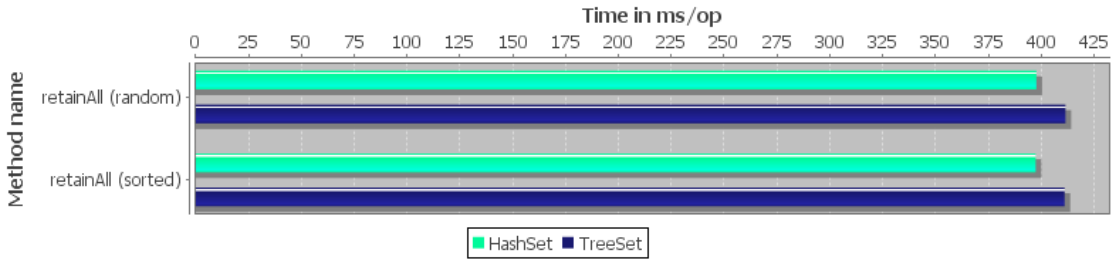

Abbildung 4.42: Werte behalten

Löschen von Daten

Aufbau der Experimente

Aus Sets können analog zur List einzelne Werte oder auch ganze `ArrayLists` mit Werten gelöscht werden. Beide Möglichkeiten wurden sowohl mit einem sortierten als auch einem unsortierten Set getestet. Als Letztes wurde das komplette Leeren des Set gemessen.

Auswertung der Experimente
Wie schon bei den Vorherigen Experimenten erreicht das `HashSet` eine deutlich bessere Performance als das `TreeSet`. Interessant ist, dass sowohl das Löschen von einzelnen Werten als auch ganzen Collections auf `HashSet` welche mit sortierten Werten erstellt wurden effizienter ist.

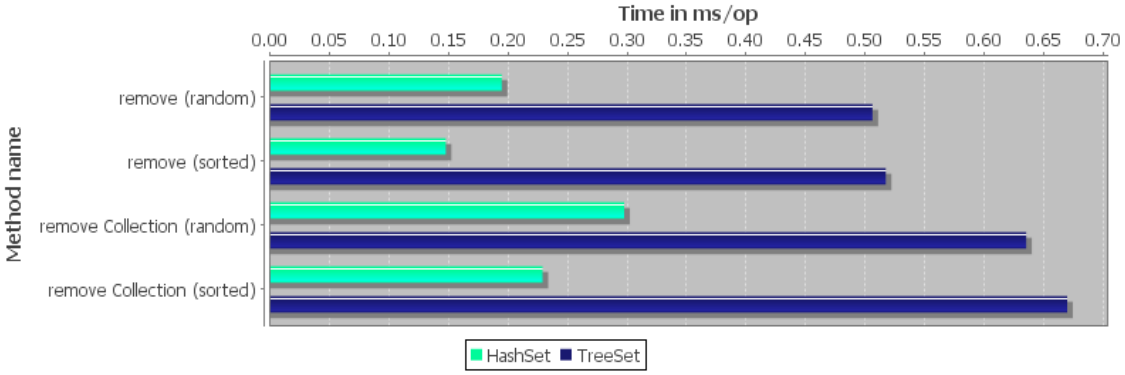

Abbildung 4.43: Werte löschen

Zusammenfassung der Resultate
Das `HashSet` schnitt bei sämtlichen Experimenten besser ab und sollte deshalb gegenüber einem `TreeSet` stets bevorzugt werden.

4.4 Iterativ/Rekursiv

Aufbau der Experimente
Die zwei Hauptarten, einen Algorithmus zu implementieren sind entweder der iterative oder der rekursive Ansatz. Beide haben Vor- und Nachteile. Mit diesem Experiment sollte gemessen werden, welche Implementation eines Algorithmus schneller ist.

Auswertung der Experimente
Währed der rekursive Ansatz (vgl. Listing 4.6) kürzer aussieht, so zeigt Abbildung 4.44 deutlich, dass diese Variante bedeutend langsamer ist. Grund dafür sind die vielen Methodenaufrufe, die bei einer Rekursion notwendig sind und jeweils zusätzliche Stack-Operationen zur Folge haben. Im Gegensatz dazu kann die iterative Variante sämtliche Berechnungen mit den Prozessor-Registern durchführen.

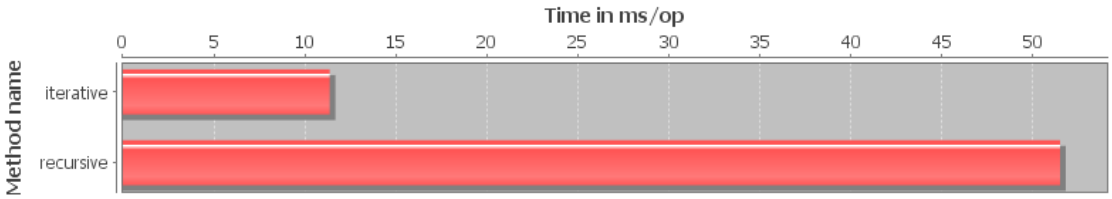

Abbildung 4.44: Iterativ und rekursiv

```
public static long recursive(int number) {
    switch(checkValidNumber(number)) {
        case 0:
            return 1;
        default:
            return recursive(number -1) * number;
    }
}
```

```java
public static long iterative(int number) {
    checkValidNumber(number);
    long result = 1;
    for(int i = number; i > 0; i--) {
        result *= i;
    }
    return result;
}
private static int checkValidNumber(int number) {
    if(number < 0) {
        throw new IllegalArgumentException();
    }
    return number;
}
```

Listing 4.6: Iterativ vs Rekursiv

Zusammenfassung der Resultate
Wenn möglich sollte ein iterativer Ansatz einer rekursiven Lösung vorgezogen werden.

4.5 Object Creation

Aufbau der Experimente
Das Erstellen von Objekten kann mehr Zeit in Anspruch nehmen als oft vermutet wird. Dies wird an zwei Beispielen gezeigt. Das erste Experiment ist das Bilden einer Summe, bei welcher die Summenvariable ein `Integer` und der jeweilige Summand ein `int` ist. Pro Aufruf der Methode werden 100 Additionen durchgeführt. Als Referenz dient eine Summe, bei welcher der Datentyp von Summe und Summand `int` ist. Das zweite Experiment ist das Hinzufügen von Werten zu einer `HashMap`. Eingefügt wir dabei ein `int`-Wert. Da Collections ausschliesslich Objekte enthalten können, muss aus jedem der Werte zuerst ein `Integer`-Objekt erstellt werden. Als Referenz dient hier das direkte Einfügen von Objekten in die `HashMap`.

Auswertung der Experimente
Wie bereits in Abschnitt 4.2 beschrieben, ist Object Creation ein wichtiger Faktor in der Performance-Optimierung. Abbildung 4.45 sollte besonders kritisch betrachtet werden, da die Benchmarks auf einem einzelnen Testrechner reproduzierbar von den anderen abwichen.

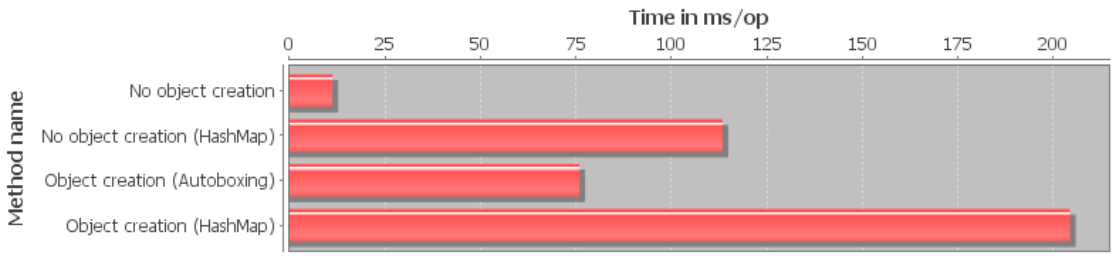

Abbildung 4.45: Objekt Erstellung

Die ersten beiden Methoden in Listing 4.7 zeigen die Auswirkungen von exzessiver Objekterstellung. In der zweiten Methode wird versucht beim jedem Schleifendurchgang ein nicht benötigtes Objekt zu erstellen. Auf der Mehrheit der Testrechner dauerten jedoch beide Methoden gleich lange, was darauf schliessen lässt, dass von Compiler entsprechend optimiert wurde. Lediglich auf einem einzigen System dauerte die zweite Methode um ein vielfaches länger als die erste.
Methode vier enthält eine implizite Objekterstellung aufgrund von Autoboxing. Beim Einfügen der <Key,

Value>-Paare muss für die Variable `i` eine Boxing-Operation und somit eine Objekterstellung durchgeführt werden, da in einer `HashMap` nur Werte vom Datentyp `Integer` erlaubt sind. Dieses Experiment zeigte auf sämtlichen Rechnern einen ähnlichen Trend.

```
public static int objectCreationWithoutCreation(int dummy) {
  int add = 10;
  for(int i = 0; i < 100; i++) {
    dummy += add;
  }
  return dummy;
}
public static int objectCreationWithCreation(int dummy) {
  int add = 10;
  for(int i = 0; i < 100; i++) {
    dummy += new Integer(add);
  }
  return dummy;
}
public static void objectCreationHashMapWithoutCreation(Integer dummy) {
  HashMap<Integer, Integer> map = new HashMap<>();
  for(int i = 0; i < 10; i++) {
    map.put(dummy, dummy);
  }
}
public static void objectCreationHashMapWithCreation(int dummy) {
  HashMap<Integer, Integer> map = new HashMap<>();
  for(int i = 0; i < 10; i++) {
    map.put(i, dummy);
  }
}
```

Listing 4.7: Object Creation

Zusammenfassung der Resultate
Da das Einfügen von Zahlenwerten in eine Collection immer eine teure Objekterstellung erfordert, sollte nach Möglichkeit ein Array bevorzugt werden.

4.6 Reflection

Aufbau der Experimente
Reflection bietet sehr viel Flexibilität, jedoch nicht ohne Kosten. Der Benchmark besteht aus einem einfachen Methodenaufruf. Die Klasse welche die Methode enthält wird dabei einmal mittels Reflection (`Class c = Class.forName()`) und als Referenz über statischen Import eingebunden.

Auswertung der Experimente
Abbildung 4.46 zeigt deutlich, dass das Einbinden von Klassen über Reflection enorm viel Zeit in Anspruch nimmt.

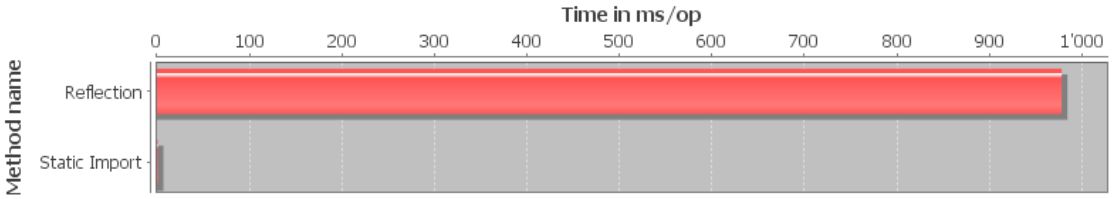

Abbildung 4.46: Reflection im Vergleich zu statischem Import

```
@Benchmark
public long reflection() {
    CheapOperations ops;
    long dummy = 0;
    for (long i = 0; i < References.BENCHMARK.REPETITIONS; i++) {
        try {
            Class c = Class.forName("ch.zhaw.bafs15.tests.CheapOperations"
                );
            Constructor con = c.getConstructor();
            ops = (CheapOperations) con.newInstance();
            int num1 = ops.doLeftShift(42);
            int num2 = ops.doRightShift(42);
            dummy += ops.doOr(num1, num2);
        } catch (ClassNotFoundException | InstantiationException |
            IllegalAccessException | IllegalArgumentException |
            InvocationTargetException | NoSuchMethodException |
            SecurityException ex) {
            Logger.getLogger(ReflectionAndImport.class.getName()).log(
                Level.SEVERE, null, ex);
        }
    }
    return dummy;
}
@Benchmark
public long staticImport() {
    long dummy = 0;
    for (long i = 0; i < References.BENCHMARK.REPETITIONS; i++) {
        CheapOperations ops = new CheapOperations();
        int num1 = ops.doLeftShift(42);
        int num2 = ops.doRightShift(42);
        dummy += ops.doOr(num1, num2);
    }
    return dummy;
}
```

Listing 4.8: Reflection und Import

Zusammenfassung der Resultate
Statische Imports sollten Reflection vorgezogen werden.

4.7 Schleifen

Aufbau der Experimente
Die Benchmarks für Schleifen umfassen unter anderem verschiedene Möglichkeiten die Zähl-Variable einer `for`-Schleife zu initialisieren. Dabei wird diese einmal direkt in der `for`-Anweisung und einmal eine Zeile davor initialisiert. Zudem wird als dritter Test noch eine Instanzvariable als Zähl-Variable verwendet. Alle drei Experimente werden mit inkrementellem und dekrementierendem Zähler durchgeführt. Ein weiterer Benchmark ist das sequentielle Durchlaufen eines Arrays mit verschiedenen Schleifenarten. Dies wird einerseits mit einer `for`-Schleife und andererseits mit einer `foreach`-Schleife getestet.

Auswertung der Experimente
Da zwischen inkrementellem und dekrementierendem Zähler nur ein geringer Unterschied gemessen wurde, ist im Listing 4.19 jeweils die inkrementelle Variante aufgeführt. Ein Unterschied zwischen der Initialisierung in der `for`-Anweisung und eine Zeile davon ist nicht erkennbar. Eine Instanzvariable als Zähl-Variable zu verwenden hat einen deutlichen Einfluss. Eine `while`- anstelle einer `while`-Schleife macht keinen Unterschied (vgl. Abbildung 4.47). Ein zusätzlicher Methodenaufruf auf `String.length()` wird vom Compiler wegoptimiert und durch den effektiven Wert ersetzt.

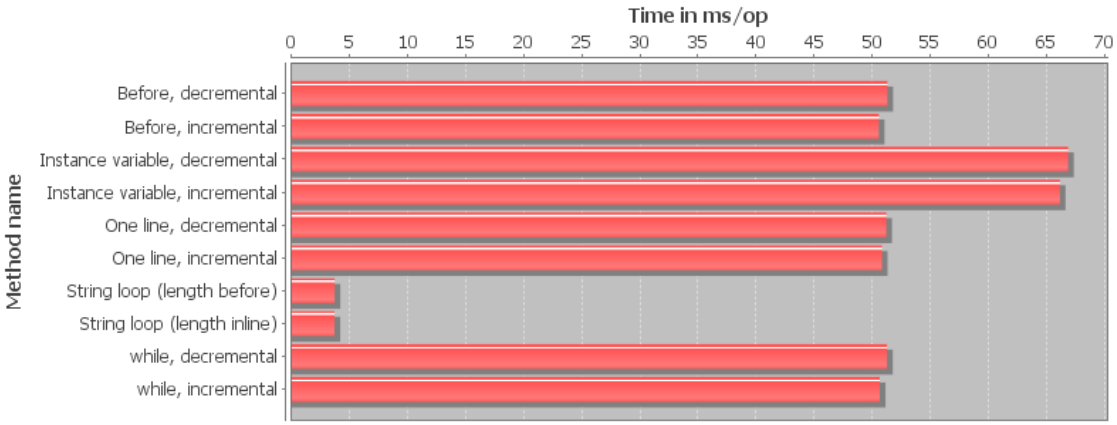

Abbildung 4.47: For Schleifen

```
public static int i;

public static long oneLineIncremental(long dummy) {
  for (int i = 0; i < 100; i++) {
    dummy += i;
  }
  return dummy;
}
public static long beforeLoopIncremental(long dummy) {
  int i;
  for (i = 0; i < 100; i++) {
    dummy += i;
  }
  return dummy;
}
public static long instanceVariableIncremental(long dummy) {
  for (i = 0; i < 100; i++) {
    dummy += i;
  }
  return dummy;
}
public static int loopStringLengthBefore(int dummy, String str) {
  int max = str.length();
  for(int i = 0; i < max; i++) {
    dummy += i;
  }
```

```
    return dummy;
  }
  public static int loopStringLengthInline(int dummy, String str) {
    for(int i = 0; i < str.length(); i++) {
      dummy += i;
    }
    return dummy;
  }
  public static long whileLoopIncremental(long dummy) {
    int i = 0;
    while(i < 100) {
      dummy += i;
      i++;
    }
    return dummy;
  }
```

Listing 4.9: Schleifen

Ein gravierender Unterschied zwischen `for` und `for each` besteht nicht (vgl. Abbildung 4.48).

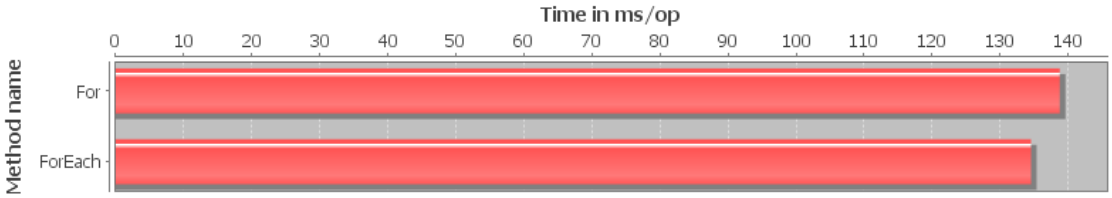

Abbildung 4.48: Foreach und For Schleifen

```
  public static long foreachAndForFor(long dummy) {
      for (int i = 0; i < dummyArray.length; i++) {
          dummy += dummyArray[i];
      }
      return dummy;
  }
  public static long foreachAndForForeach(long dummy) {
      for (int i : dummyArray) {
          dummy += i;
      }
      return dummy;
  }
```

Listing 4.10: For- und ForEach-Schleifen

Zusammenfassung der Resultate
Instanzvariablen sollten nie als Zähler verwendet werden sondern es sollte immer mit lokalen Variablen gearbeitet werden.

4.8 Strings

Aufbau der Experimente
Die Verkettung von Strings findet häufige Anwendung bei dynamischen Ausgaben eines Programms. In einem Whitepaper von 2003 schlägt Hagemo vor (Hagemo, 2003), den `StringBuffer` gegenüber der Verkettung von Strings mit + oder += vorzuziehen. Die erstellten Benchmarks umfassen jeweils die Verkettung von String mittels + Operator, `StringBuffer` und `StringBuilder`. Jede der drei Optionen wird innerhalb und ausserhalb einer Schleife getestet.

Auswertung der Experimente
Die Verkettung von Strings mit dem + Operator ist ausserhalb von Schleifen (vgl. Listing 4.11) deutlich am Performantesten (siehe Abbildung 4.49). In der Theorie wird für jeden Teilstring ein eigenes Objekt erzeugt, da Strings in Java unveränderlich sind. Die Beobachtung des Laufzeitverhaltens lässt jedoch vermuten, dass der Compiler in der Lage ist dies zu optimieren.

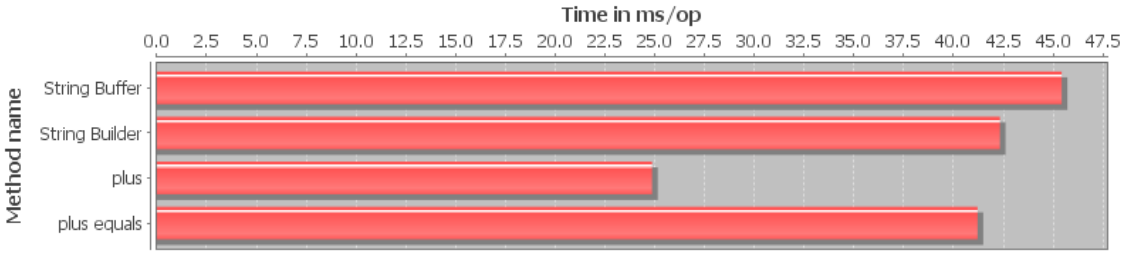

Abbildung 4.49: String Verkettung

```java
public static final String plus(String a, String b) {
    return a + " " + b;
}
public static final String plusEquals(String a, String b) {
    String s = "";
    s += a;
    s += " ";
    s += b;
    return s;
}
public static final String stringBuilder(String a, String b) {
    StringBuilder string = new StringBuilder();
    string.append(a);
    string.append(" ");
    string.append(b);
    return string.toString();
}
public static final String stringBuffer(String a, String b) {
    StringBuffer string = new StringBuffer();
    string.append(a);
    string.append(" ");
    string.append(b);
    return string.toString();
}
```

Listing 4.11: String Verkettung ohne Schleife

Die Verkettung von String innerhalb von Schleifen scheinen vom Compiler nicht optimiert werden zu können. Hier sind `StringBuilder` und `StringBuffer` deutlich schneller (vgl. Abbildung 4.50).

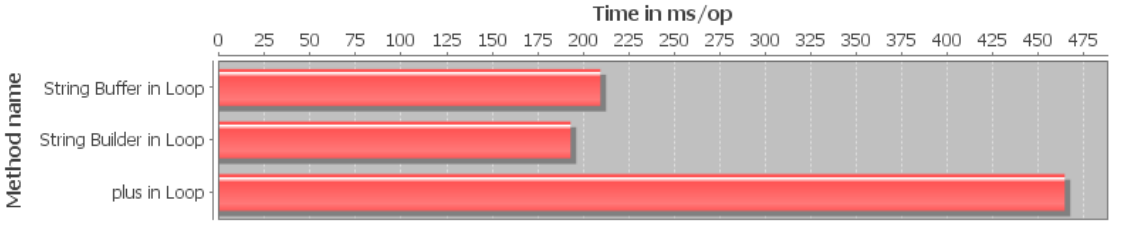

Abbildung 4.50: String Verkettung in Schleifen

```java
public static final String loopPlus(String[] strings) {
    String string = "";
    for(String s : strings) {
        string += s;
```

```
        }
        return string;
    }
    public static final String loopStringBuilder(String[] strings) {
        StringBuilder string = new StringBuilder();
        for(String s : strings) {
            string.append(s);
        }
        return string.toString();
    }
    public static final String loopStringBuffer(String[] strings) {
        StringBuffer string = new StringBuffer();
        for(String s : strings) {
            string.append(s);
        }
        return string.toString();
    }
```

Listing 4.12: String Verkettung mit Schleife

Zusammenfassung der Resultate
Bei der Verkettung von String im Zusammenhang mit Schleifen sollte ein String-Builder verwendet werden. Ausserhalb von Schleifen ist die Verkettung mittels + Operator performanter.

4.9 Switch, If-Else und If

Aufbau der Experimente
Für das Benchmarking der Flusskontrollstrukturen wird eine Kette von 20 `if` bzw. `else if` einer `switch` Anweisung mit 20 `case`s gegenübergestellt. Sowohl für die `if`- als auch die `switch`-Anweisung werden `int`-Werte verglichen.

Auswertung der Experimente
Wie aus den Resultate in Abbildung 4.51 hervorgeht benötigt eine `switch`-Anweisung am wenigsten Zeit, dies ist dadurch zu erklären, dass der Compiler diese zu einer Jump-Table umwandelt. Bei einer Verkettung von `if`-Anweisungen müssen in jedem Fall alle evaluiert werden, bei `else if` kann nach erster erfolgreicher Überprüfung ans Ende gesprungen werden.

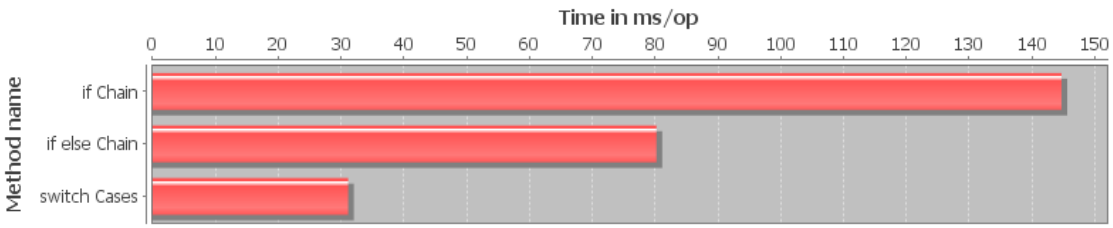

Abbildung 4.51: Switch, If-Else und If

```
    public static long switchCase(long dummy) {
        for (int number = 1; number < 21; number++) {
            switch (number) {
                case 1:
                    dummy += number;
                    break;
                case 2:
                    dummy += number;
                    break;
                case 3:
                    dummy += number;
```

```
                    break;
                ...
        }
        return dummy;
    }
    public static long ifElseChain(long dummy) {
        for (int number = 1; number < 21; number++) {
            if (number == 1) {
                dummy += number;
            } else if (number == 2) {
                dummy += number;
            } else if (number == 3) {
                dummy += number;
            ...
        }
        return dummy;
    }
    public static long ifIfChain(long dummy) {
        for (int number = 1; number < 21; number++) {
            if (number == 1) {
                dummy += number;
            }
            if (number == 2) {
                dummy += number;
            }
            if (number == 3) {
                dummy += number;
            }
            ...
        }
        return dummy;
    }
```

Listing 4.13: If IfElse-Chain und Switch-Case

Zusammenfassung der Resultate
Bei vielen Überprüfungen sollte eine switch Anweisung einer Verkettung von if / else if Anweisungen vorgezogen werden.

4.10 Switch mit Enum, int und String

Aufbau der Experimente
Als weiterer Vergleich von Flusskontrollstrukturen wurden eine `switch`-Anweisung mit `Enum`, `int` und `String` mit 26 `case`s verglichen.

Auswertung der Experimente
Falls `Strings` mittels `switch`-Anweisung verarbeitet werden, sollte auf ein `Enum` zurückgegriffen werden, da diese Variante wesentlich performanter ist (vgl. Abbildung 4.52).

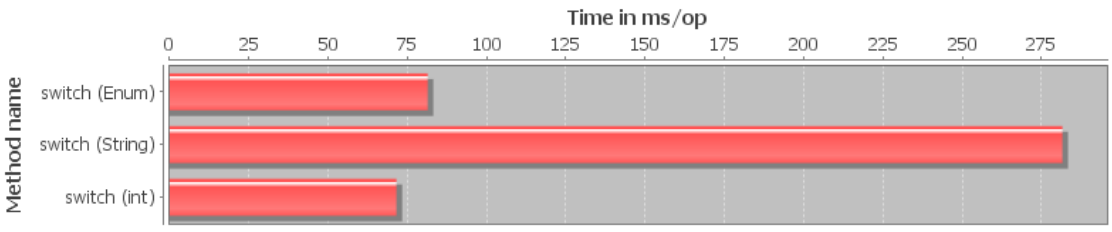

Abbildung 4.52: Switch mit Enum, int und String

```
    public static enum LETTERS {
        A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z
    }
    public static final int stringSwitch(String str) {
        int returnValue;
        switch (str) {
            case "a":
                returnValue = 1;
                break;
            case "b":
                returnValue = 2;
                break;
            case "c":
                returnValue = 3;
                break;
            ...
        }
        return returnValue;
    }
    public static final int enumSwitch(LETTERS letter) {
        int returnValue;
        switch(letter) {
            case A:
                returnValue = 1;
                break;
            case B:
                returnValue = 2;
                break;
            case C:
                returnValue = 3;
                break;
      ...
        }
        return returnValue;
    }
    public static final int intSwitch(int num) {
        int returnValue;
        switch(num) {
            case 1:
                returnValue = 2;
                break;
            case 2:
                returnValue = 3;
                break;
            case 3:
                returnValue = 4;
                break;
      ...
        }
        return returnValue;
    }
```

Listing 4.14: Switch mit Enum int und String

Zusammenfassung der Resultate
Eine `switch`-Anweisung mit Buchstaben und Wörtern ist mittels `Enum` fast so schnell wie eine `switch`-Anweisung mit `int case`s.

4.11 Zufallszahlen

Aufbau der Experimente
Gute Zufallszahlen sind für viele Algorithmen sehr wichtig und spielen auch in der Kryptografie eine grosse Rolle. Der Benchmark für Zufallszahlen vergleicht `Math.random()` mit `SecureRandom()` aus der Library `java.security`.

Auswertung der Experimente
Wie aus der Abbildung 4.53 deutlich hervorgeht dauert das Erstellen von kryptographisch sicheren Zufallszahlen massiv länger. Dies liegt daran, dass sehr hohe Anforderungen an die Qualität von kryptographische Zufallszahlen gestellt werden und Zeit in der Regel eine untergeordnete Rolle spielt. Im Gegensatz dazu ist die Hauptanforderung an `Math.random()` die Geschwindigkeit, weshalb auch einfach zu berechnende pseudo Zufallszahlen genügend sind.

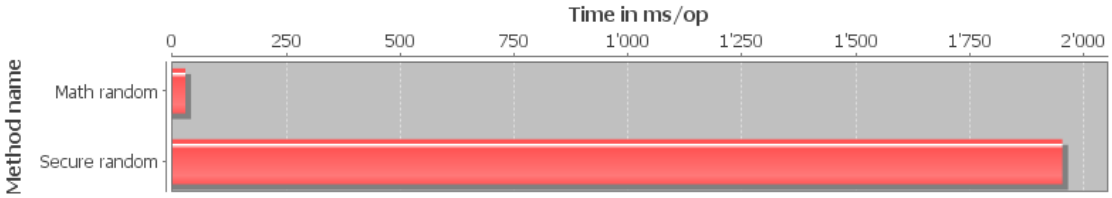

Abbildung 4.53: Zufallszahlen

```
1   public static double getRandomNumber() {
2       return Math.random();
3   }
4   public static double getSecureRandomNumber() {
5       SecureRandom rnd = new SecureRandom();
6       return rnd.nextDouble();
7   }
```

Listing 4.15: Zufallszahlen

Zusammenfassung der Resultate
SecureRandom sollte nur in sicherheitskritischen Anwendungen verwendet werden.

5 Diskussion und Ausblick

In den folgenden Abschnitten werden die Resultate kritisch diskutiert und mögliche Erweiterungen und Folgearbeiten vorgestellt.

5.1 Diskussion der Resultate

Programmcode wird auf aktuellen Rechnersystemen mehrmals optimiert und verändert. Dies geschieht zum ersten Mal durch den Optimizer, welcher zur Compile-Zeit beispielsweise "toten" oder redundanten Code entfernt. Zusätzlich zu diesen Optimierungen werden jedoch auch Performance-Counter eingebaut (Andreasson, 2012). Diese verlangsamen zwar kurzfristig gesehen den aktuellen Byte-Code, ermöglichen dafür bei späteren Optimierungen wiederum bessere Entscheidungen. Die nächsten Optimierungsschritte werden durch den JIT vorgenommen, welcher zur Laufzeit die erwähnten Performance-Counter auswertet und anhand der häufigsten Benutzung weiter optimiert. Oracle hat ein Beispiel für die Arbeitsweise des JIT in der Dokumentation des JRockit JDK (Savija, 2011) veröffentlicht.

Die Performance eines Programms wird weiterhin durch die Prozessor- und Speicherarchitektur sowie Technologien wie Hyperthreading beeinflusst. Dadurch ergeben sich je nach System andere Möglichkeiten zur Optimierung des ursprünglichen Codes. Somit ist es kaum mehr möglich zu bestimmen wie performant ein Stück Code auf einem bestimmten Zielsystem sein wird. Die in dieser Arbeit experimentell ermittelten Werte und Erkenntnisse treffen für die Mehrheit der aktuellen Standard-Computersysteme zu. Auf spezialisierter Hardware können die Ergebnisse jedoch variieren.
Des weiteren ist zu beachten, dass sämtliche Experimente mit Java Version 1.8.0 durchgeführt wurden. Tests mit älteren oder zukünftigen JVM Versionen können ebenfalls andere Ergebnisse liefern.

Das beschriebene Verhalten konnte während den durchgeführten Experimenten mehrfach beobachtet werden. So dauerten beispielsweise die beiden Tests für Object Creation (siehe Abschnitt 4.5) auf allen bis auf einem Zielsystem gleich lange. Jenes Zielsystem braucht für den Test welcher die Erstellung eines Objekts beinhaltet fünf Mal länger als ohne Objekterstellug. Dieses Ergebnis war sowohl mit dem kompletten Benchmarkprogramm als auch in isolierten Tests konstant und reproduzierbar.
Die Resultate einiger Messungen übertrafen auch deutlich die Erwartungen. Beispielsweise ist der Einfluss von Boxing Operationen beim Rückgabewert einer Methode enorm (siehe Abschnitt 4.2). Interessanterweise war bei diesem Experiment auch der JIT nicht in der Lage entsprechende Optimierungen vorzunehmen.

5.2 Erweiterungen

Das im Rahmen dieser Arbeit entwickelte Benchmark Framework auf Basis von JMH bietet durch den modularen Aufbau zahlreiche Möglichkeiten zur Erweiterung. Weitere Tests und Benchmarks lassen sich mit geringem Aufwand hinzufügen (siehe Abschnitt 3.2.3), womit beispielsweise zusätzliche Java Libraries wie `java.io` getestet werden können.
Eine weitere Erweiterungsmöglichkeit wäre das Testen von zusätzlichen Aspekten wie beispielsweise dem Speicherverbrauch, da ein Code mit guter Laufzeit nicht automatisch auch speichereffizient ist. Zusätzlich könnte auch die Anzahl der erstellten Objekte gezählt werden, da diese später vom Garbage Collector entfernt werden müssen, was jeweils zusätzliche Last für das Hostsystem darstellt.

Während bei dieser Arbeit ausschliesslich mit den Standard JVM Parametern[1] gearbeitet wurde, liegt hier oft auch Optimierungspotenzial.

Unabhängig vom Framework kann auch die Verfügbarkeit der Resultate verbessert werden. Mithilfe der MySQL Datenbank, die zur Auswertung der Benchmark Resultate erstellt wurde, kann beispielsweise eine Webseite aufgebaut werden, welche die gewonnenen Erkenntnisse grafisch darstellt. Weiterhin möglich wären Programme, welche die Effizienz von Programmcode bestimmen und anhand der Datenbank den möglichen Effizienzgewinn berechnen.

[1] Einzige Ausnahme sind -Xms und -Xmx zur Festlegung des Arbeitsspeichers

6 Verzeichnisse

Literaturverzeichnis

Andreasson, Eva (2012). *JVM performance optimization, Part 2: Compilers: Use the right Java compiler for your Java application*. Hrsg. von JavaWorld. Framingham. URL: http://www.javaworld.com/article/2078635/enterprise-middleware/jvm-performance-optimization-part-2-compilers.html [Stand 03. 06. 2015].

Bloch, Joshua (2011). *Effective Java: [revised and updated for Java SE 6]*. 2. ed., 10. print. The Java series from the source. Upper Saddle River, NJ: Addison-Wesley. ISBN: 0-321-35668-3.

Decker, Colin und Gregory Kick (o.J[a]). *Java Microbenchmark Review Criteria*. URL: https://github.com/google/caliper/wiki/JavaMicrobenchmarkReviewCriteria [Stand 03. 06. 2015].

Decker, Colin und Gregory Kick (o.J[b]). *Java Microbenchmarks*. URL: https://github.com/google/caliper/wiki/JavaMicrobenchmarks [Stand 31. 05. 2015].

Gosling, James, Bill Joy, Guy Steele, Guy Steele, Gilad Bracha und Alex Buckley (2014). *The Java Language Specification: Java SE 8 Edition*.

Hagemo, Lloyd (2003). *Java Coding Practices for Improved Application Performance*. El Segundo. URL: http://www.capitalware.com/dl/docs/WhitePaperJavaCodingPractices.pdf [Stand 31. 05. 2015].

JFree (o.J). *jFreeChart*. URL: http://www.jfree.org/jfreechart/ [Stand 31. 05. 2015].

Oracle and/or its affiliates (o.J). *Vector: Java Platform SE 8*. URL: https://docs.oracle.com/javase/8/docs/api/java/util/Vector.html [Stand 03. 06. 2015].

Savija, T.V (2011). *Oracle JRockit Introduction: Release R28*. URL: http://docs.oracle.com/cd/E15289_01/doc.40/e15058.pdf [Stand 31. 05. 2015].

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

A Anhang

A.1 Offizielle Aufgabenstellung

Zürcher Hochschule für Angewandte Wissenschaften

School of Engineering

Praktische Bachelorarbeit

Dozent:	Cieliebak Mark	**Studiengang**:	IT
	Thaler Markus	**Jahr:**	2015
Studierende:	Simbürger Manuel Thöni Roman	**Ausgabe:**	9.2.2015
		Abgabe:	5.6.2015
Experte:			

Best-Practices zur Performance-Optimierung bei der Software-Entwicklung in Java

Qualitativ hochwertige Software ist ein entscheidender Faktor für den nachhaltigen Erfolg von ICT-Produkten. Vereinfacht ausgedrückt gilt: Gute SW = korrekt + effizient + wartungsfreundlich.

In dieser Bachelor Arbeit liegt der Focus auf dem Thema Effizient: Es soll eine Sammlung von Best-Practices entwickelt werden, wie man die Effizienz von Java-basierten Software-Systemen bestimmen und optimieren kann. Einige exemplarische Fragestellungen sind: Ist ein Array schneller als eine ArrayList? Welche JVM-Einstellungen wirken sich wie auf die Performance aus? Welchen Einfluss haben OR-Wrapper auf die Performance?

Aufgabenstellung

- Recherchieren Sie den aktuellen Stand der Thematik in Literatur und Internet und stellen Sie diesen dar
- Finden Sie eine geeignete Möglichkeit, um Performance-Messungen vorzunehmen
- Definieren Sie den Aufbau einer sinnvollen Testumgebung (Hardware und Software) zur vergleichenden Effizienz-Messung
- Beschaffen und installieren Sie die nötige HW und SW
- Plausibilisierung des Frameworks: Bauen Sie bestehende Experimente nach und verifizieren Sie, dass Ihr Experiment-Setup funktioniert (keine Messfehler)
- Nehmen Sie Performance-Messungen an konkreten Software-Fragmenten vor
- Gehen Sie bei Ihrer Arbeit systematisch und wissenschaftlich vor
- Betrachten Sie die erzielten Resultate kritisch und überlegen Sie sich mögliche Erweiterungen und nächste Schritte

A.2 Inhaltsverzeichnis CD