



Java Entscheidungsheft

Autoren:

Fabian Großmann
Pascal Otto
Michael Švancar
Mohamed Amine El Jabri
Franz Scheffler
Gerardo Balderas
Sarah Bertulat
Jens Weber
Florian Gnepper
Stephan Strehler

Professor:

Prof. Dr. Wagenknecht

Diese Arbeit befasst sich mit einer Entscheidungshilfe für Programmierer im Umgang mit Listen in der Programmiersprache Java. Hierfür wurde ein Vergleich aus 8 Klassen gezogen. Der Vergleich beruht auf gegebenen und angepassten Methoden. Die Listen wurden mit Zahlen des Typ Integer gefüllt. Durch die hier dargestellten Erkenntnisse soll eine Erleichterung der Wahl der Klasse gegeben sein. Eine Schlussanalyse vergleicht noch einmal die Klassen untereinander. So kann das hier dargestellte Problem auf das eigene besser abgeleitet werden.

Inhaltsverzeichnis

1	Einleitung	5
1.1	Thematische Einleitung	5
1.2	Zielstellung und Abgrenzung der Arbeit	5
2	Problemvorstellung	6
3	Klassenübersicht	7
3.1	TreeSet	7
3.1.1	Einleitung	7
3.1.2	Methodenbeschreibung	7
3.1.3	Effizienzanalyse	8
3.1.4	Schlussfolgerung	12
3.2	HashSet	12
3.2.1	Einleitung	12
3.2.2	Methodenbeschreibung	12
3.2.3	Effizienzanalyse	14
3.2.4	Schlussfolgerung	19
3.3	ArrayList	19
3.3.1	Einleitung	19
3.3.2	Methodenbeschreibung	19
3.3.3	Effizienzanalyse	20
3.3.4	Schlussfolgerung	24
3.4	LinkedList	24
3.4.1	Einleitung	24
3.4.2	Methodenbeschreibung	25
3.4.3	Effizienzanalyse	25
3.4.4	Schlussfolgerung	30
3.5	Vector	30
3.5.1	Einleitung	30
3.5.2	Methodenbeschreibung	31
3.5.3	Effizienzanalyse	32
3.5.4	Schlussfolgerung	36
3.6	Queue	36
3.6.1	Einleitung	36
3.6.2	Methodenbeschreibung	37
3.6.3	Effizienzanalyse	37
3.6.4	Schlussfolgerung	41
3.7	Stack	42
3.7.1	Einleitung	42
3.7.2	Methodenbeschreibung	42
3.7.3	Effizienzanalyse	43
3.7.4	Schlussfolgerung	48
3.8	Deque	48

<i>INHALTSVERZEICHNIS</i>	3
3.8.1 Einleitung	48
3.8.2 Methodenbeschreibung	49
3.8.3 Effizienzanalyse	49
3.8.4 Schlussfolgerung	53
4 Schlussanalyse	54
5 Fazit	55

Abbildungsverzeichnis

3.1	TreeSet - ExtractMin	8
3.2	TreeSet - GetMin	9
3.3	TreeSet - Insert	10
3.4	TreeSet - Merge	11
3.5	TreeSet - Remove	12
3.6	HashSet - ExtractMin	14
3.7	HashSet - getMin	15
3.8	HashSet - Insert	16
3.9	HashSet - Merge	17
3.10	HashSet - Remove	18
3.11	ArrayList - ExtractMin	20
3.12	ArrayList - getMin	21
3.13	ArrayList - Insert	22
3.14	ArrayList - Merge	23
3.15	ArrayList - Remove	24
3.16	LinkedList - ExtractMin	26
3.17	LinkedList - GetMin	27
3.18	LinkedList - Insert	28
3.19	LinkedList - Merge	29
3.20	LinkedList - Remove	30
3.21	Vector - ExtractMin	32
3.22	Vector - GetMin	33
3.23	Vector - Insert	34
3.24	Vector - Merge	35
3.25	Vector - Remove	36
3.26	Queue - ExtractMin	38
3.27	Queue - getMin	39
3.28	Queue - Insert	40
3.29	Queue - Merge	41
3.30	Queue - Remove	42
3.31	Stack - ExtractMin	43
3.32	Stack - GetMin	45
3.33	Stack - Insert	46
3.34	Stack - Merge	47
3.35	Stack - Remove	48
3.36	Deque - extractMin	49
3.37	Deque - getMin	50
3.38	Deque - Insert	51
3.39	Deque - Merge	52
3.40	Deque - Remove	53

Kapitel 1

Einleitung

1.1 Thematische Einleitung

In dem Prozess der Entwicklung von komplexerer Software ist es notwendig auch die Effizienz des Programmes zu betrachten. Dabei können schon teilweise kleinere Veränderungen eine deutliche Leistungssteigerung darstellen. Ein Beispiel hierfür ist, andere Klassen zu verwenden oder sich selbst neue Klassen zu definieren.

Im Rahmen der Lehrveranstaltung Algorithmen und Komplexität wird dafür ein Entscheidungsheft für die Programmiersprache Java entwickelt. Für die Implementation wurde die neuste Java Version (Java 8) verwendet. Alle Quellcodebeispiele sind auf der begleitenden Website http://programmingwiki.de/Organisation_Java-Entscheidungsheft zu finden.

1.2 Zielstellung und Abgrenzung der Arbeit

Ziel dieser Arbeit ist es an Hand von 5 gewählten Funktionen möglichst umfassend die Stärken und Schwächen der einzelnen Klassen zu beleuchten. Dabei wurden schon vorhandene Funktionen wie zum Beispiel `getMin()` aber auch eigene Funktionen entwickelt und getestet. Mehr dazu im Kapitel Problemvorstellung.

Nachdem wir als erstes durch das eben genannte Kapitel die betrachteten Funktionen kennen gelernt haben, gehen wir zu der Betrachtung der einzelnen Java-Klassen über. Diese werden jeweils kurz vorgestellt sowie wichtige Methoden von diesen beschrieben. Danach folgt eine grafische und textuelle Auswertung der Effizienzanalysen. Diese sind auch mit anderen Klassen zu vergleichen, da die Testbedingungen immer gleich waren.

Zum Schluss werden die Klassen im Kapitel Schlussanalyse noch einmal miteinander verglichen. Die Arbeit soll dem Leser einen Überblick über die Klassen geben und ihm bei seinem Entscheidungsvorgang unterstützen. Die Arbeit kann natürlich nicht alle Funktionen die der potentielle Anwender umsetzen möchte beinhalten. Dies würde den Rahmen dieser Arbeit sprengen. Daher ist es selbstverständlich notwendig, eigene Testimplementation bei der Entwicklung von eigener Software durchzuführen.

Kapitel 2

Problemvorstellung

Gegeben sind 5 verschiedene Funktionen. Diese werden alle in neue Klassen implementiert welche jeweils von der betrachteten Javaklasse erbt. Alle Klassen befassen sich mit dem Umgang von Listen. Die Liste ist mit Zahlen aus dem Datenbereich Integer gefüllt. So startet man mit den gleichen Bedingungen für alle Klassen. Dabei werden zur Bearbeitung der Listen folgende Funktionen bereitgestellt:

- **getMin:** gibt die kleinste Zahl der Liste zurück
- **extractMin:** löscht den kleinsten Wert aus der Liste
- **insert:** fügt eine neue Zahl in die Liste
- **remove:** entfernt ein bestimmtes Element aus der Liste
- **merge:** fügt 2 Objekte dieser Klasse zusammen

Bei der Umsetzung der einzelnen Methoden kann es klassenbedingte Abweichung geben. Die Vorgehensweise setze sich folgendermaßen zusammen:

1. Schreiben einer Übersicht über Javaklasse
2. Implementierung der Methoden in Java
3. Erstellen der Diagramme
4. Auswerten der Daten

Um die Ergebnisse besser vergleichen zu können wurden mehrere Messungen durchgeführt und daraus der Mittelwert gebildet. Um die Aussagekraft der Diagramme zu erhöhen, wird auch noch in Best-, Average- und Worst-Case unterschieden.

Kapitel 3

Klassenübersicht

3.1 TreeSet

3.1.1 Einleitung

Eine NavigableSet-Implementation basierend auf TreeMap. Die Elemente werden anhand ihrer natürlichen Ordnung oder mithilfe eines Vergleichs-Operators sortiert, je nach verwendetem Konstruktor. Sie besitzt eine garantierte Laufzeit von $O(\log(n))$ für die Basisoperationen (`add()`, `remove()` und `contains()`).

3.1.2 Methodenbeschreibung

- `add(e)` - fügt das angegebene Element diesem Set hinzu, falls es noch nicht vorhanden ist
- `addAll(c)` - fügt die alle Elemente der angegeben Collection hinzu
- `clear()` - entfernt alle Elemente aus dem TreeSet
- `clone()` - gibt eine flache Kopie dieser TreeSet-Instanz zurück
- `comparator()` - gibt den verwendeten Vergleichs-Operator zurück, oder null wenn die natürliche Sortierung verwendet wird
- `contains(o)` - gibt true zurück, wenn das spezifizierte Element im TreeSet enthalten ist
- `first()` - gibt das aktuell erste (niedrigste) Elemente zurück
- `last()` - gibt das aktuell letzte (höchste) Element zurück
- `remove(o)` - entfernt das angegebene Element aus dem Set, sofern vorhanden
- `size()` - gibt die Anzahl von Elementen in diesem TreeSet zurück
- `pollFirst()` - gibt das erste Element dieses Sets zurück und entfernt es, oder gibt null zurück, wenn das Set leer ist
- `pollLast()` - gibt das letzte Element dieses Sets zurück und entfernt es, oder gibt null zurück, wenn das Set leer ist
- `subSet(E fromElement, [boolean fromInclusive], E toElement, [boolean toInclusive])` - gibt eine Untermenge dieses Sets zusammen, dessen Elemente von (inklusive) `fromElement` bis (inklusive) `toElement` reichen.
- `tailSet(E fromElement, [boolean inclusive])` - gibt eine Untermenge dieses Sets zusammen, dessen Elemente größer als (inklusive) `fromElement` sind
- `headSet(E toElement, [boolean inclusive])` - gibt eine Untermenge dieses Sets zusammen, dessen Elemente kleiner als (inklusive) `toElement` sind

3.1.3 Effizienzanalyse

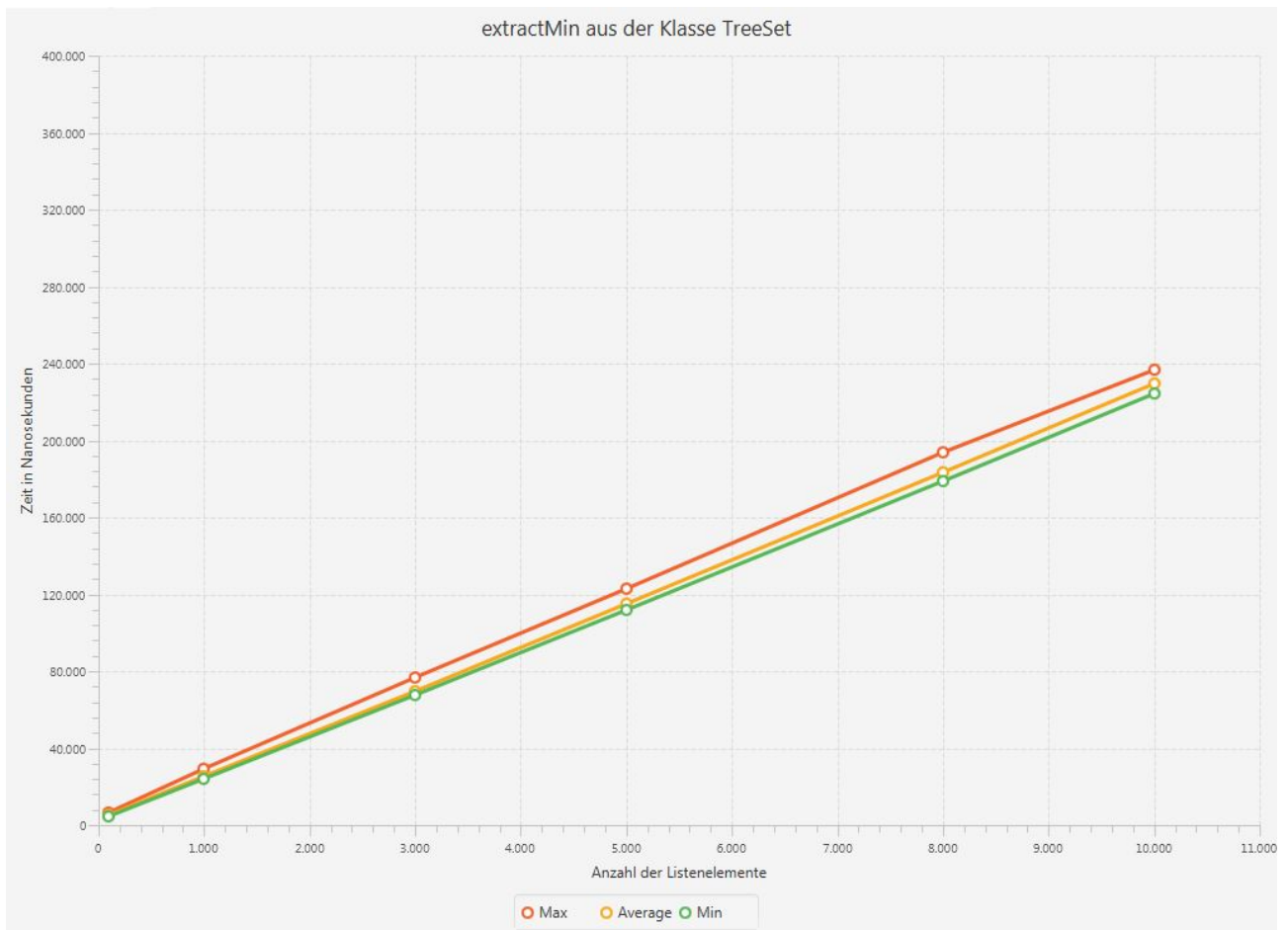


Abbildung 3.1: TreeSet - ExtractMin

Bei der Methode `ExtractMin()` sieht man für alle drei Graphen einen linearen Anstieg. Es gibt keine wesentliche Schwankungen. Aus dem kann man sehen, dass es um einem linearen Aufwand $O(n)$ geht.

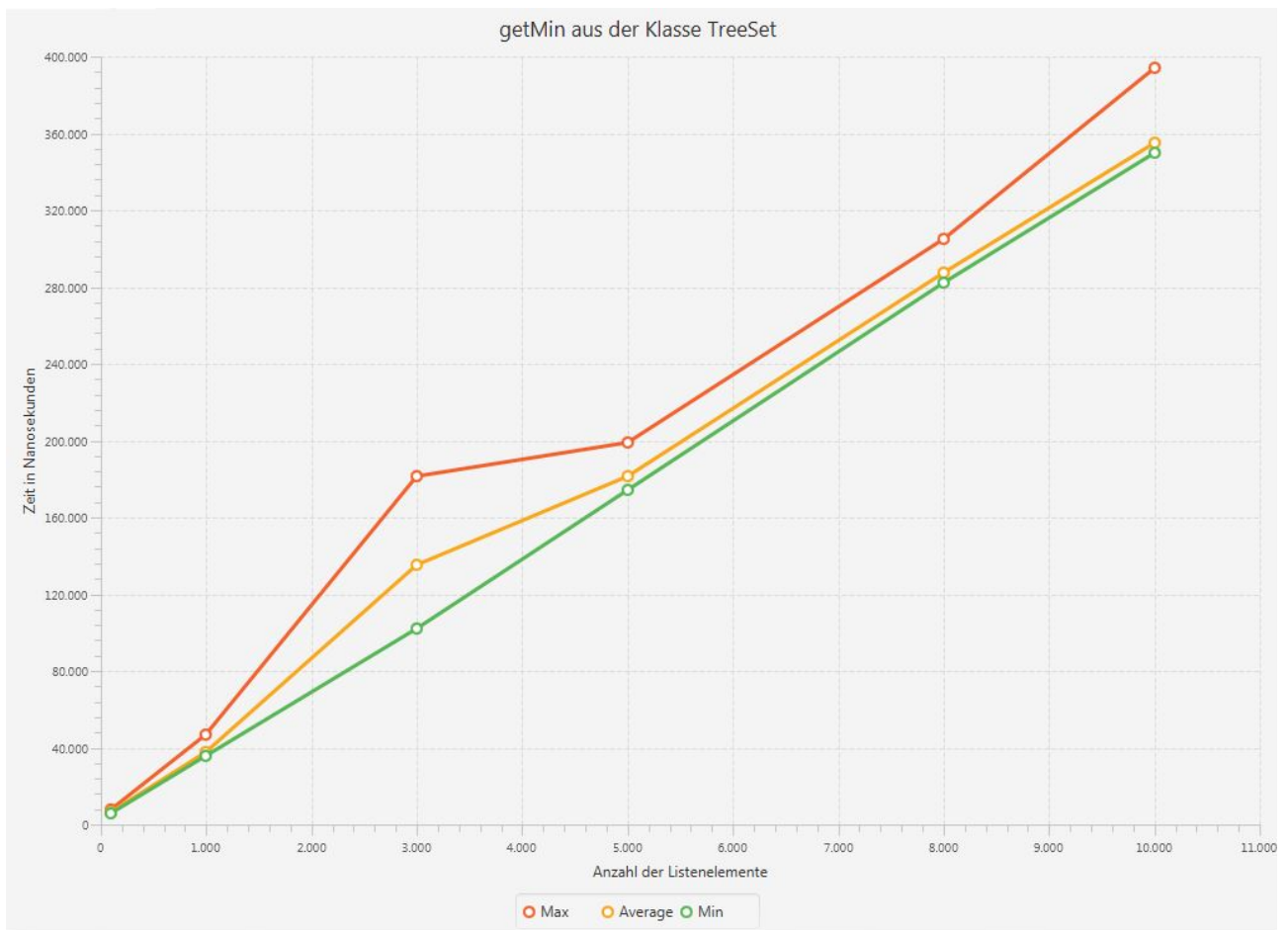


Abbildung 3.2: TreeSet - GetMin

Bei der Methode `GetMin()` sieht man auch einen linearen Anstieg, aber es gibt zwischen 1000 und 5000 Listenelementen eine riesig wesentliche Schwankung, wo der maximaler und mittlerer Aufwand erst starker ansteigen und dann wieder mit dem minimalen Aufwand gleichen Anstieg haben. Insgesamt ergibt sich ein Aufwand von $\mathcal{O}(n)$.

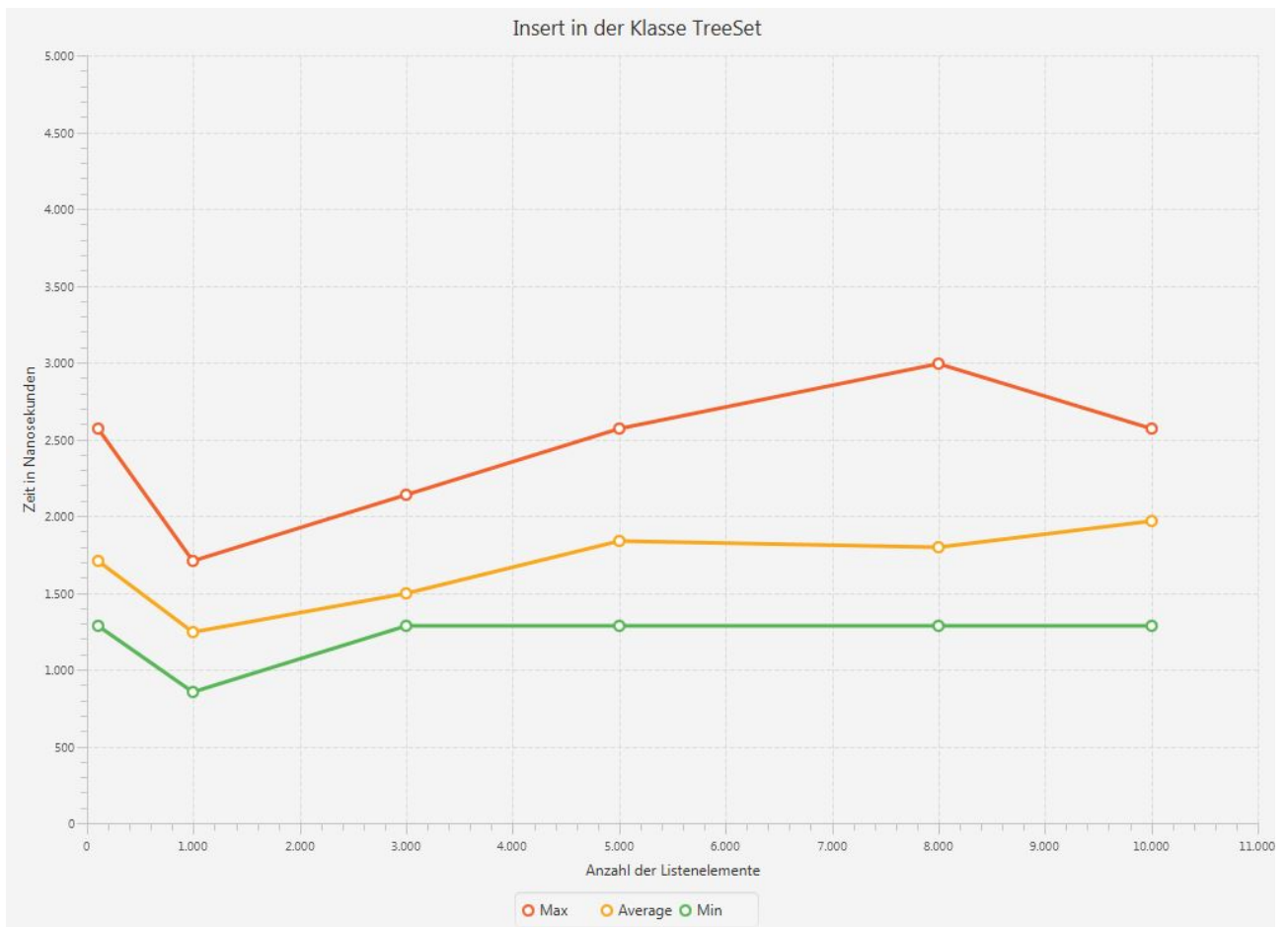


Abbildung 3.3: TreeSet - Insert

Bei der Methode `Insert()` haben alle der drei Graphen einen negativen Anstieg und erst ab 1000 Listenelementen wird anders. Der minimaler Aufwand bleibt in einer Ebene ab 3000 Listenelementen und die andere Graphen bewegen sich mehr oder weniger auch in einer Ebene. Daraus kommt eine Schlussfolgerung, dass der Aufwand $\mathcal{O}(k)$ ist, wobei k eine Konstante ist.

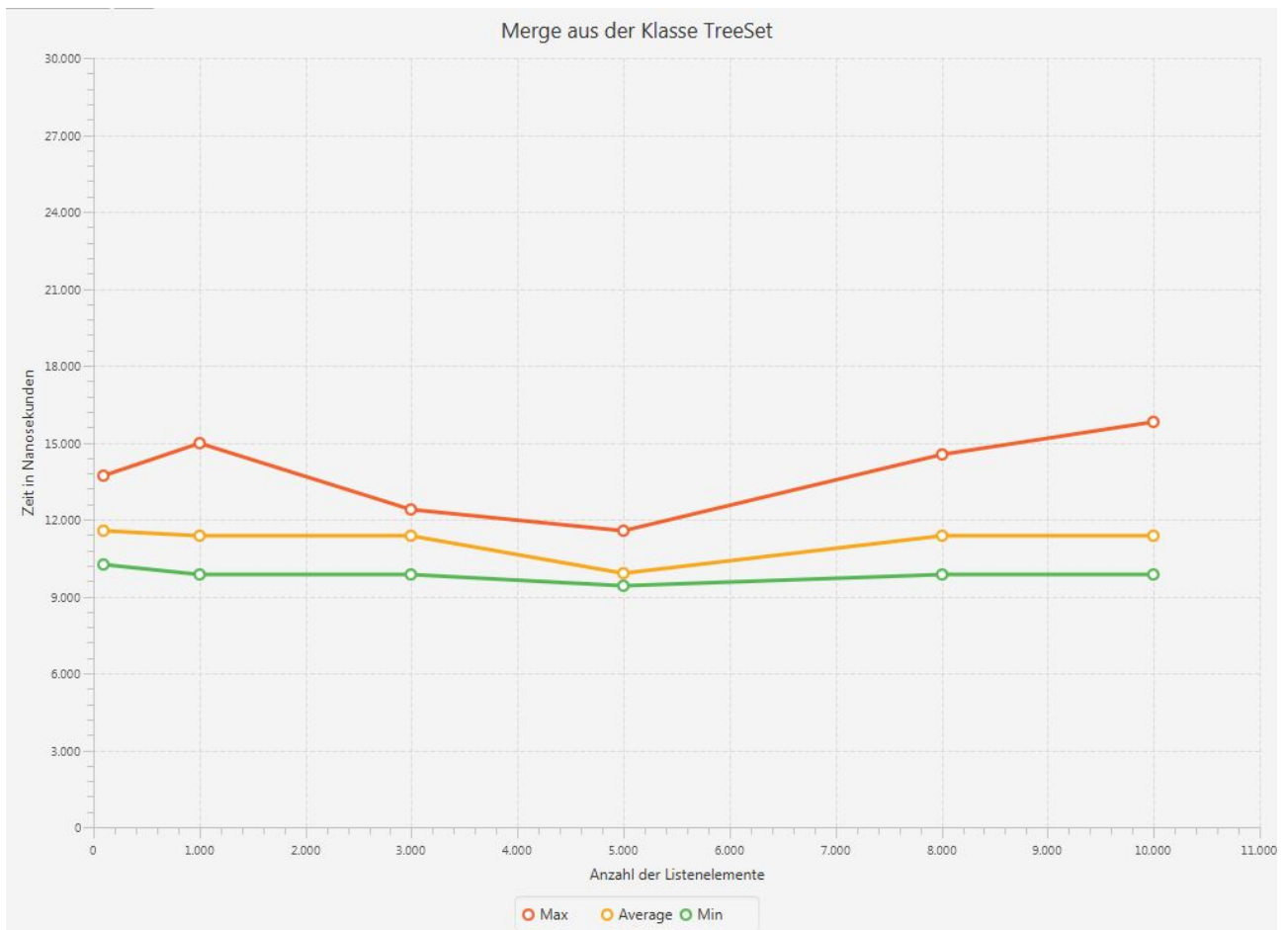


Abbildung 3.4: TreeSet - Merge

Der Min-Graph der Methode `Merge()` zeigt im Wesentlichen keinen erkennbaren Anstieg, während der Maximale Aufwand stark schwankt. Er sinkt zunächst bis ca. 5000 Elemente und steigt dann wieder, flacht dabei jedoch ab, bis er sich wieder ungefähr auf Ausgangs-Niveau befindet. Daraus lässt sich ein Konstanter Aufwand $\mathcal{O}(k)$ ableiten.

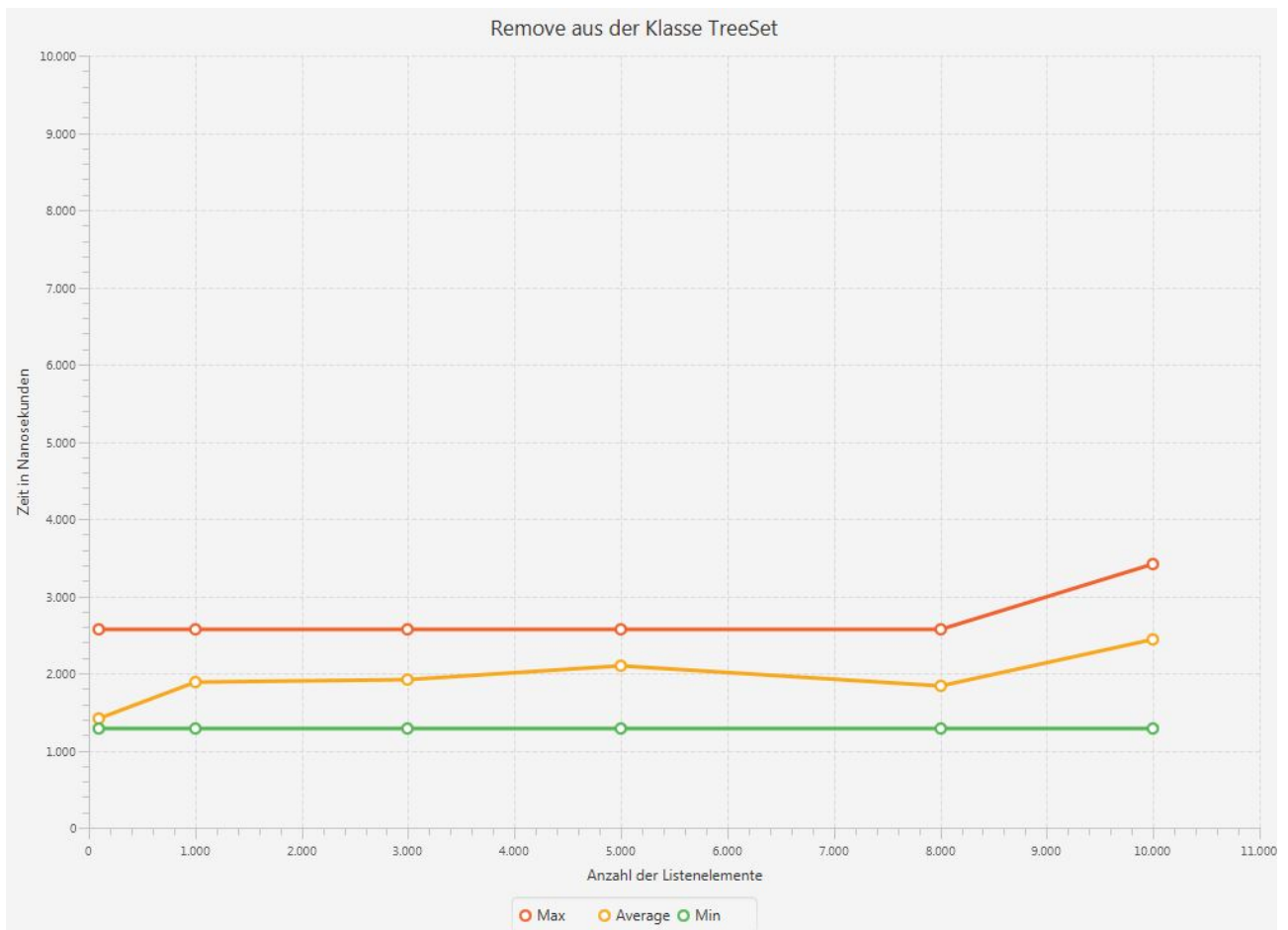


Abbildung 3.5: TreeSet - Remove

Wie schon bei Merge() und Insert() zeigt der Graph für die Methode Remove() einen konstanten Aufwand $\mathcal{O}(k)$. Er zeigt sich jedoch bis auf einen Ausreißer nach oben am Ende des Max-Graphen wesentlich stabiler.

3.1.4 Schlussfolgerung

Die Klasse TreeSet eignet sich vor allem für Anwendungen, bei denen viele Elemente in die Collection eingefügt bzw. Entfernt werden oder zwei Collections zusammengefügt werden sollen, da der Aufwand für diese Methoden konstant ist. Ansonsten tritt ein linearer Aufwand auf, was sie auch für die meisten übrigen Anwendungen qualifiziert, sofern keine Mehrfachvorkommen von Elementen benötigt werden.

3.2 HashSet

3.2.1 Einleitung

HashSet gehört zum java.util Paket. Es implementiert das Set-Interface und speichert seine Elemente in einer Hashtable.

3.2.2 Methodenbeschreibung

- add(E e) : - Fügt ein spezifisches Element dem Set hinzu, wenn es nicht schon vorhanden ist.
- clear() : - Entfernt alle Elemente von diesem Set.
- clone() : - Gibt eine oberflächliche Kopie des Sets zurück: Enthaltene Elemente werden nicht geklont.

- `contains(Object o)` : - Gibt true zurück wenn das Set das spezifische Element beinhaltet.
- `isEmpty()` : - Gibt true zurück wenn das Set keine Elemente enthält.
- `iterator()` : - Gibt einen Iterator über die Elemente im Set zurück.
- `remove(Object o)` : - Entfernt ein spezifisches Element vom Set, wenn es vorhanden ist.
- `size()` : - Gibt die Anzahl an Elementen dieses Sets zurück (Kardinalität).

3.2.3 Effizienzanalyse

Die Methode **extractMin** hat einen linearen Aufwand für alle drei gegebenen Graphen : $\theta(n)$

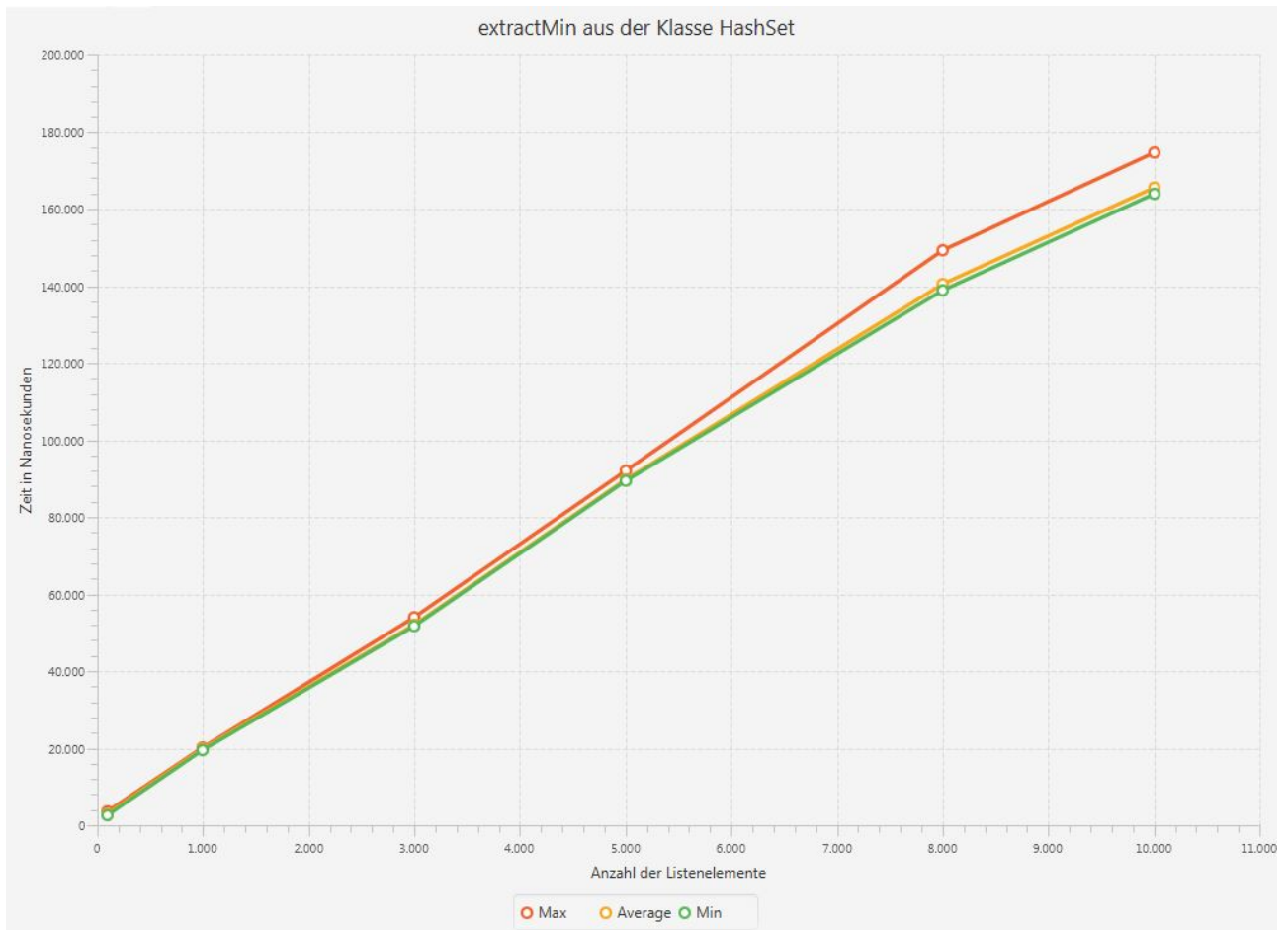


Abbildung 3.6: HashSet - ExtractMin

Auch die Methode **getMin** besitzt einen linearen Aufwand wie alle drei Graphen zeigen : $\theta(n)$

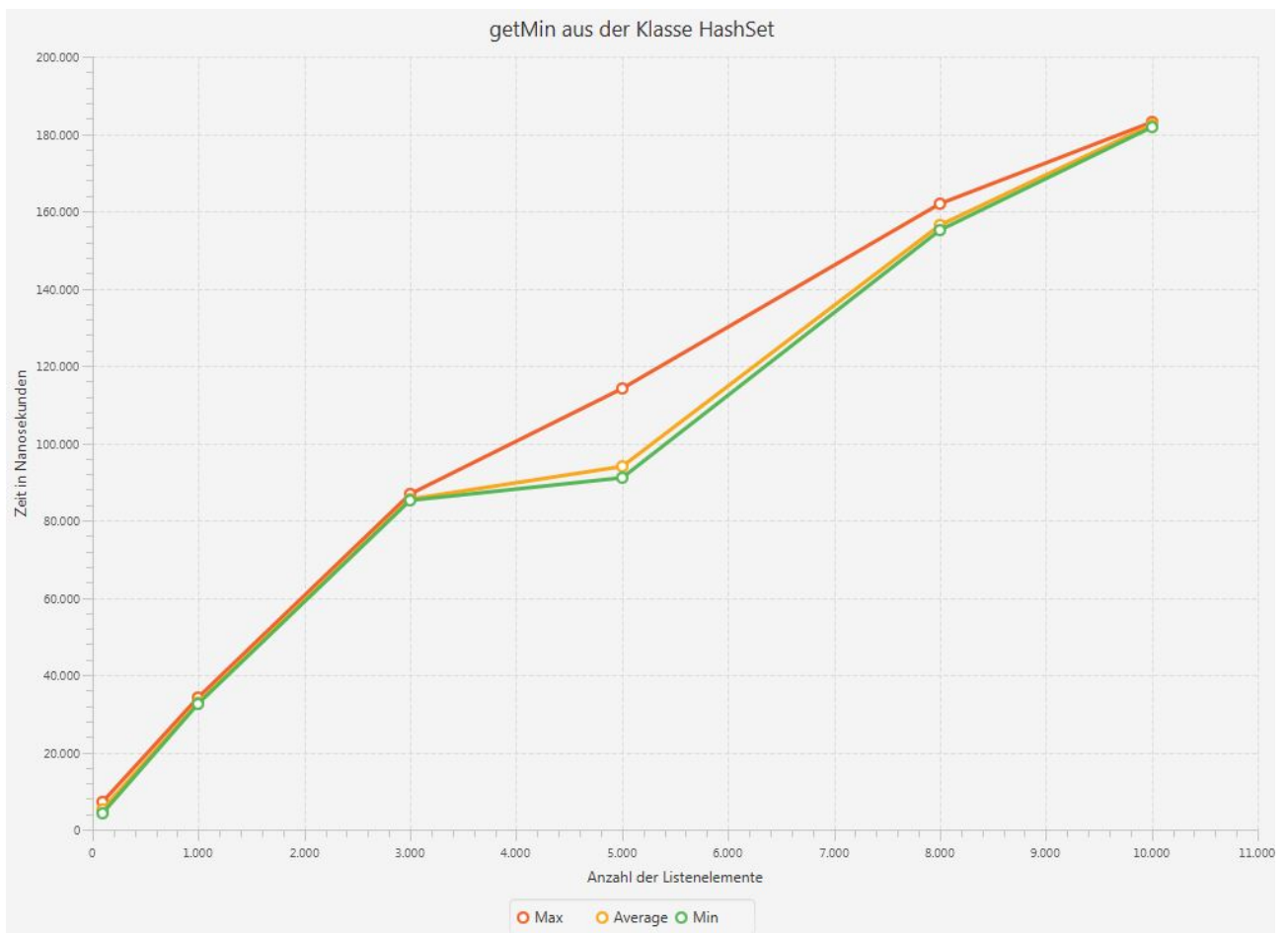


Abbildung 3.7: HashSet - getMin

Wir können sehen, dass die Werte bei allen Graphen fast konstant bleiben. Deswegen hat die Methode **Insert** aus der Klasse Hashset einen Konstanten Aufwand : $\theta(1)$

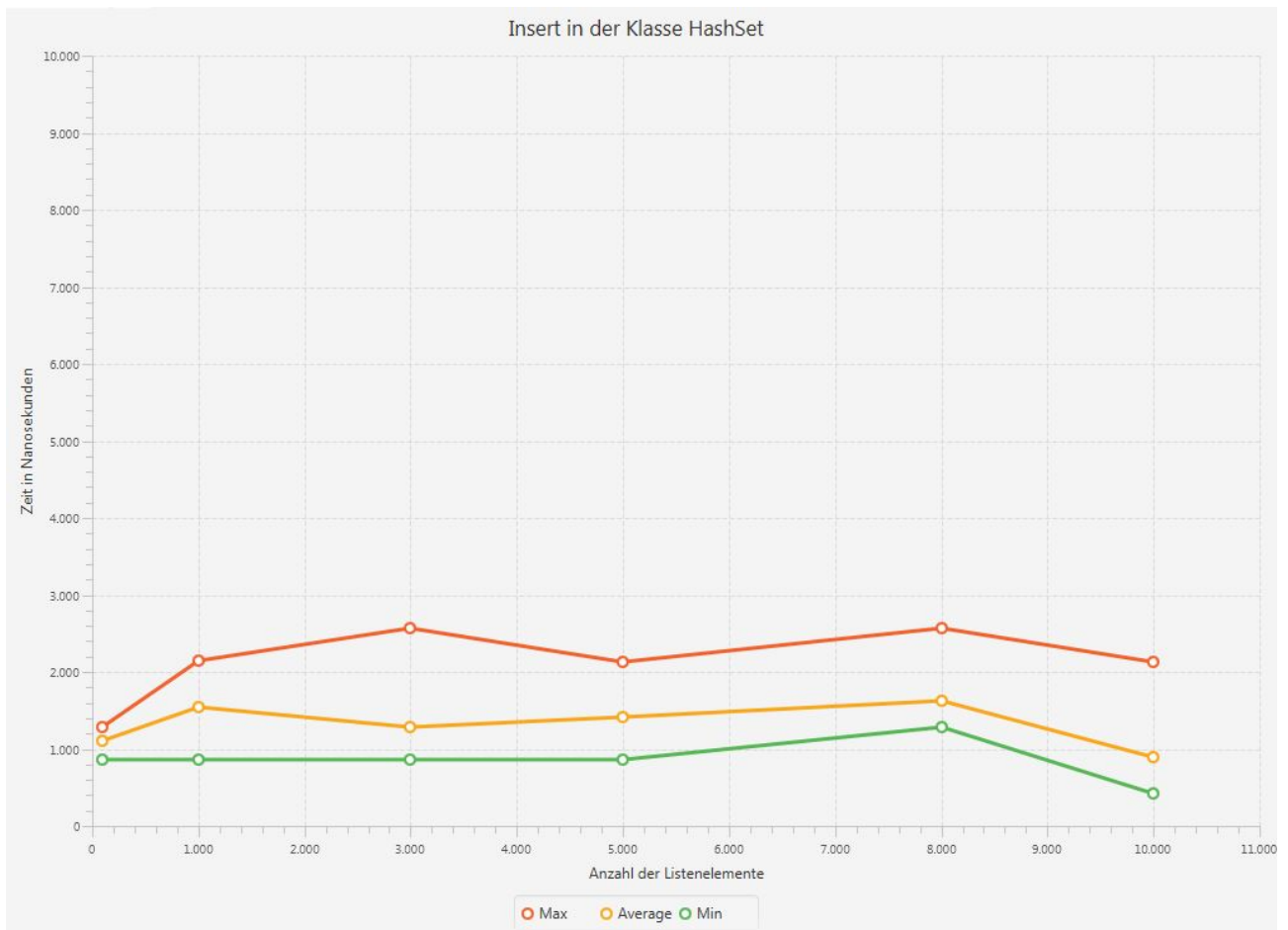


Abbildung 3.8: HashSet - Insert

Die Methode **Merge** aus der Klasse `HashSet` hat einen Konstanten Aufwand. Bei einer Anzahl von 8000 Elementen weisen die Graphen eine Unregelmäßigkeit auf : $\theta(1)$

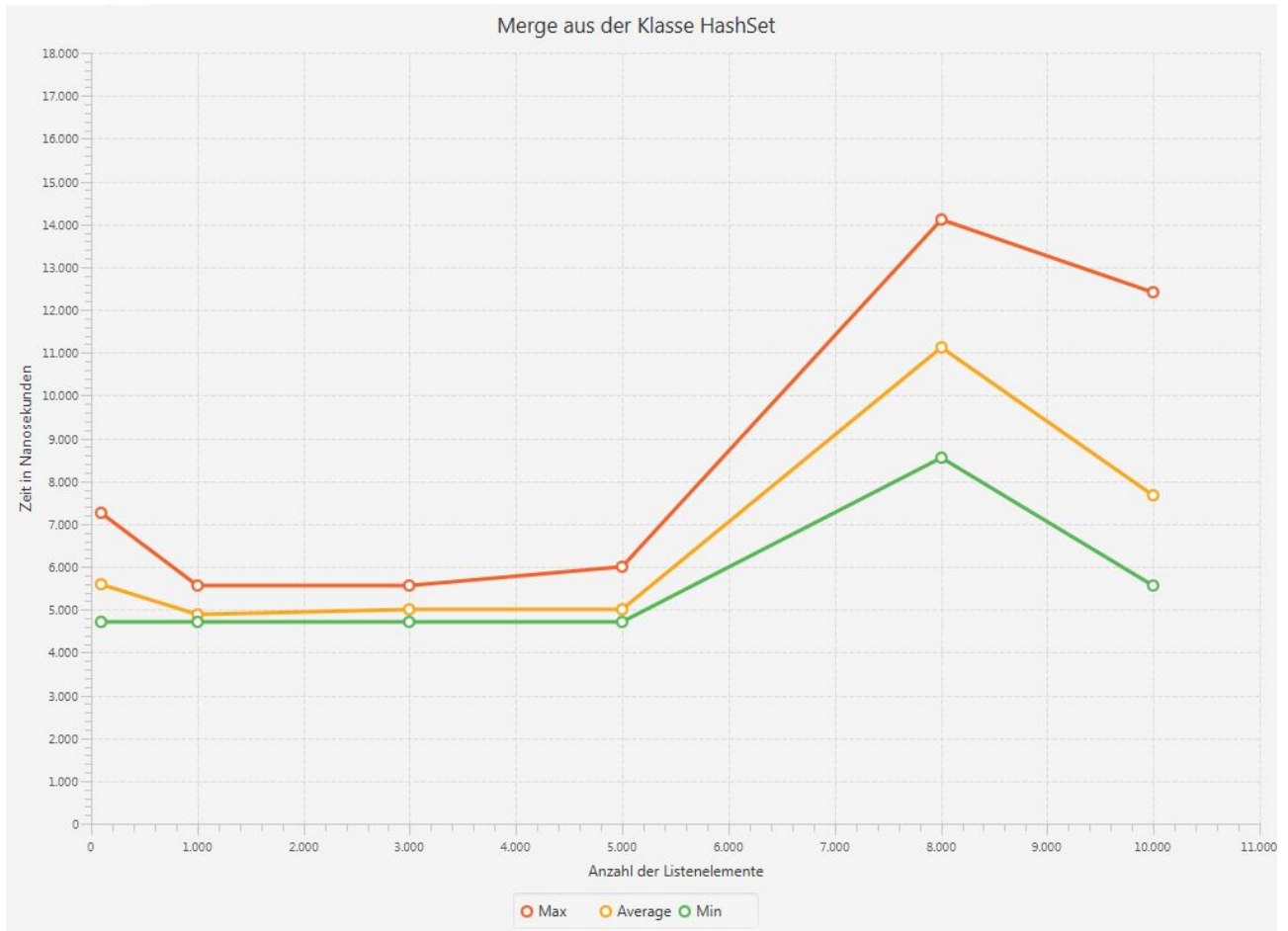


Abbildung 3.9: HashSet - Merge

Die Methode **Remove** aus der Klasse `HashSet` hat einen Konstanten Aufwand, welcher durch die Graphen ersichtlich wird : $\theta(1)$

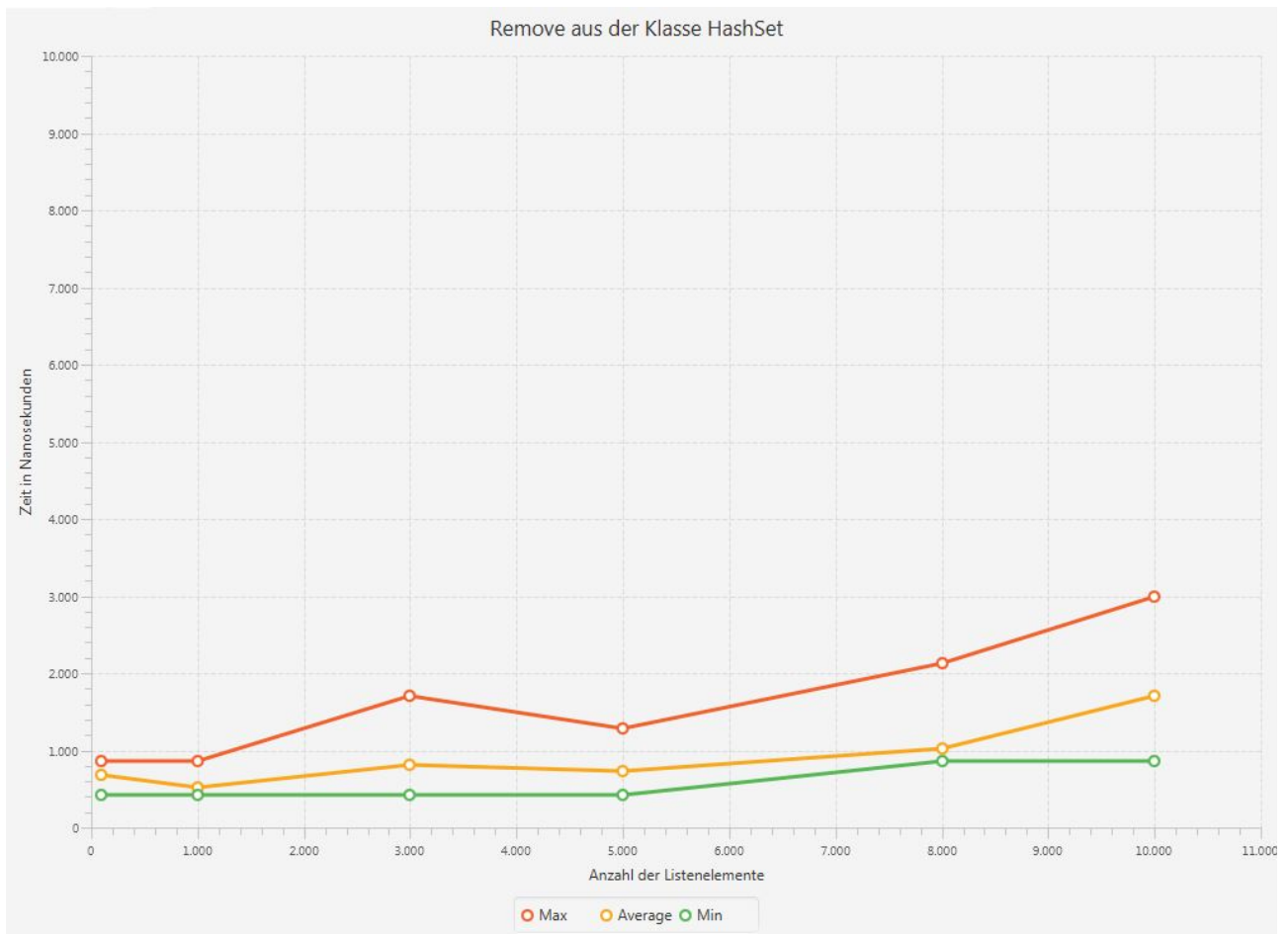


Abbildung 3.10: HashSet - Remove

3.2.4 Schlussfolgerung

Alles in Allem bietet das HashSet in allen untersuchten Methoden einen sehr guten (konstanten) bis guten (linearen) Aufwand, weshalb sie sich für solch ein Einsatzgebiet was sich auf diese Funktionalitäten beschränkt zu empfehlen ist.

3.3 ArrayList

3.3.1 Einleitung

Ein **ArrayList** implementiert die Liste Interface. ArrayList unterstützt dynamische Arrays, die ihrer Größe verändern wenn nötig.

Normale Arrays haben eine bestimmte Größe, wenn sie erzeugt sind, dann kann diese Größe nicht verändert werden. Die haben den Nachteil, dass die Anzahl der Elementen schon bekannt sein sollte. ArrayLists löst dieses Problem, indem die Größe des Arrays automatisch angepasst wird. Wenn ein ArrayList schon voll ist, und ein neues Element eingefügt wird, wird die ArrayList automatisch die Größe erhöhen, sodass es Platz für die neues Element gibt. Wenn ein Element gelöscht wird, ist es möglich die Arraygröße zu verkleinern.

In ArrayList gibt es drei Konstruktoren:

- ArrayList() Erzeugt eine leere array Liste
- ArrayList(Collection c) Erzeugt eine array Liste mit den Elementen von dem Collection c
- ArrayList(int capacity) Erzeugt eine array Liste mit gewünschte Kapazität. Diese Kapazität wurde automatisch größer wenn die array Liste die Kapazität erreicht.

3.3.2 Methodenbeschreibung

- add(Object o) : - addiert ein Element zu die Liste.
- add(int index, Object o) : - addiert der Element in den eingegebene Stelle.
- addAll(Collection c) : - addiert alle Elementen in einer Collection.
- addAll(int index, Collection c) : - addiert alle Elementen ab die eingegebene Stelle.
- clear() : - löscht alle Elementen in der Liste.
- contains(Object o) : - gibt boolean Wert ob die gefragte Element in der Liste ist.
- get(int index) : - gibt den Element zurück an der eingegebene Stelle.
- remove(int index) : - Löscht der Element in der eingegebene Stelle.
- set(int index, Object o) : - Ersetzt der Element in der eingegebene Stelle mit einem neuen Element
- size() : - gibt die größe der Liste zurück
- Object[] toArray() : - erzeugt ein Array mit die Elementen der Liste

3.3.3 Effizienzanalyse

ExtractMin

Die Methode **extractMin** ist eine Methode wo der minimale Wert wird gesucht, dann gelöscht und am Ende zurückgegeben wird. Diese Methode hat einen linearen Aufwand bei allen drei Graphen : $\theta(n)$. Ab dem wert 8.000 ist es zu sehen dass die Worst und Average Case sich stark unterscheiden mit dem Best Case. Dies kann sein, da jedes Element durchgesucht, verglichen und gelöscht werden muss.

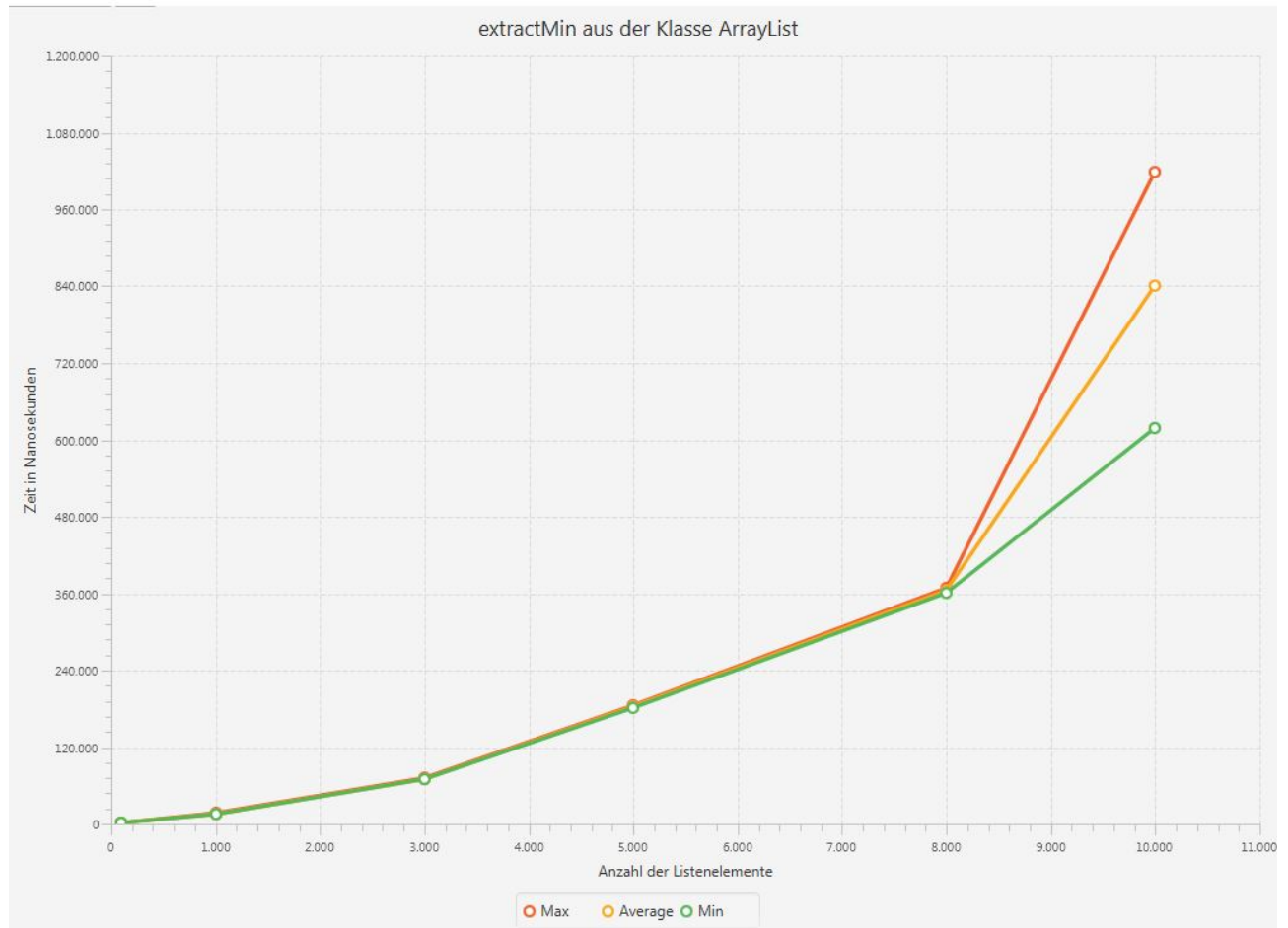


Abbildung 3.11: ArrayList - ExtractMin

GetMin

GetMin ist eine Methode wo der niedrigste Wert gesucht ist. Wie bei der letzte Methode **getMin** hat die Methode auch einen linearen Aufwand wie alle drei Graphen zeigen : $\theta(n)$. Bei die Drei Fälle ab 5.000 Elementen kann man sehen das es ein unterschied gibt, allerdings nicht so größer.

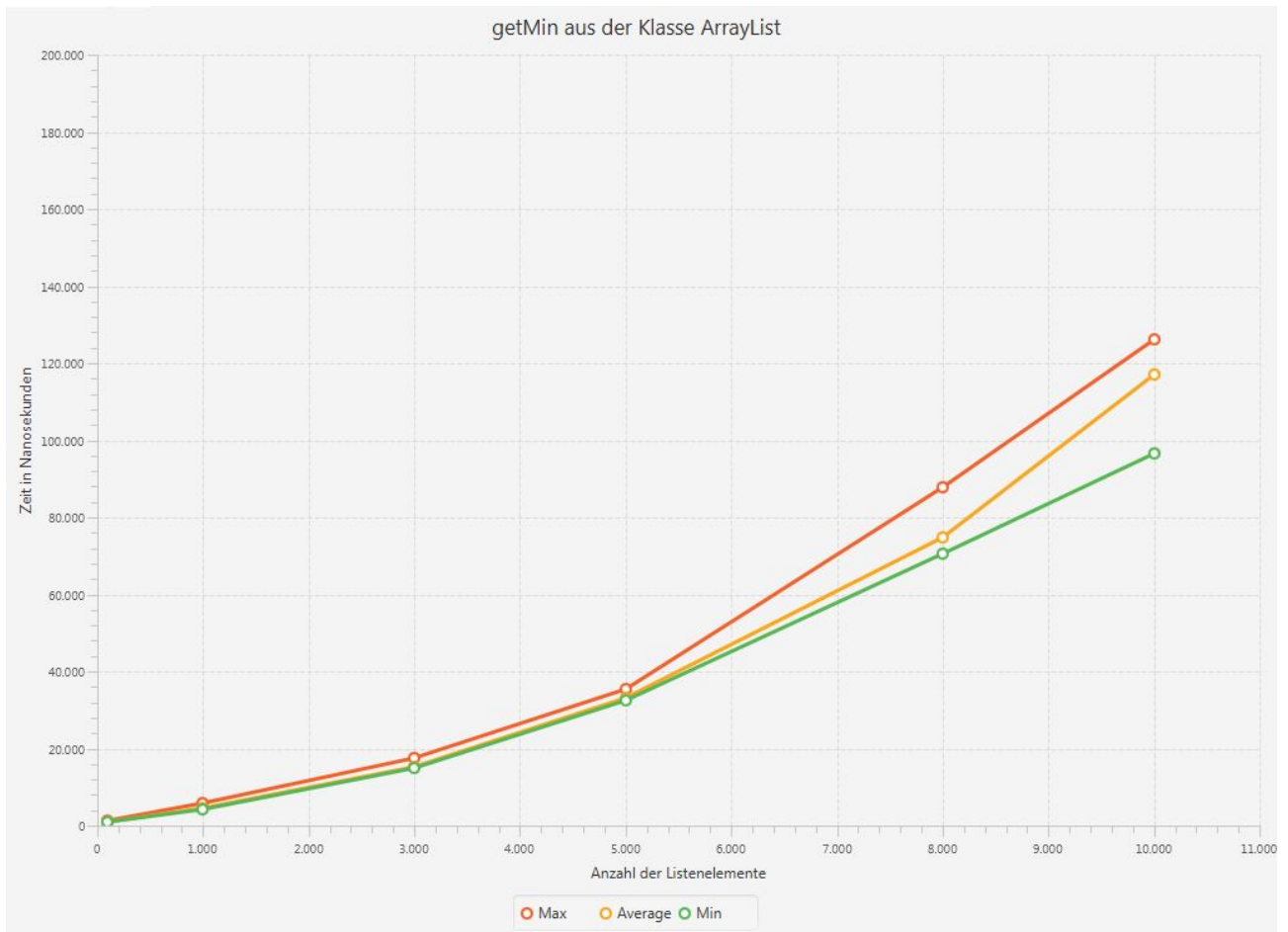


Abbildung 3.12: ArrayList - getMin

Insert

Insert ist eine Methode, wobei neue Werten zu den ArrayList addiert werden. Wir können sehen, dass die Werte bei allen Graphen fast konstant bleiben. Deswegen hat die Methode **Insert** aus der Klasse ArrayList einen Konstanten Aufwand : $\theta(1)$. Die Insert Methode ist nicht besonderes aufwendig, denn da addiert nur ein wert zu den ArrayList. Bei der Worst Case, ist es zu vermuten dass es hat länger gedauert weil der ArrayList erst vergrößern werden muss um ein neue Wert addieren zu können.

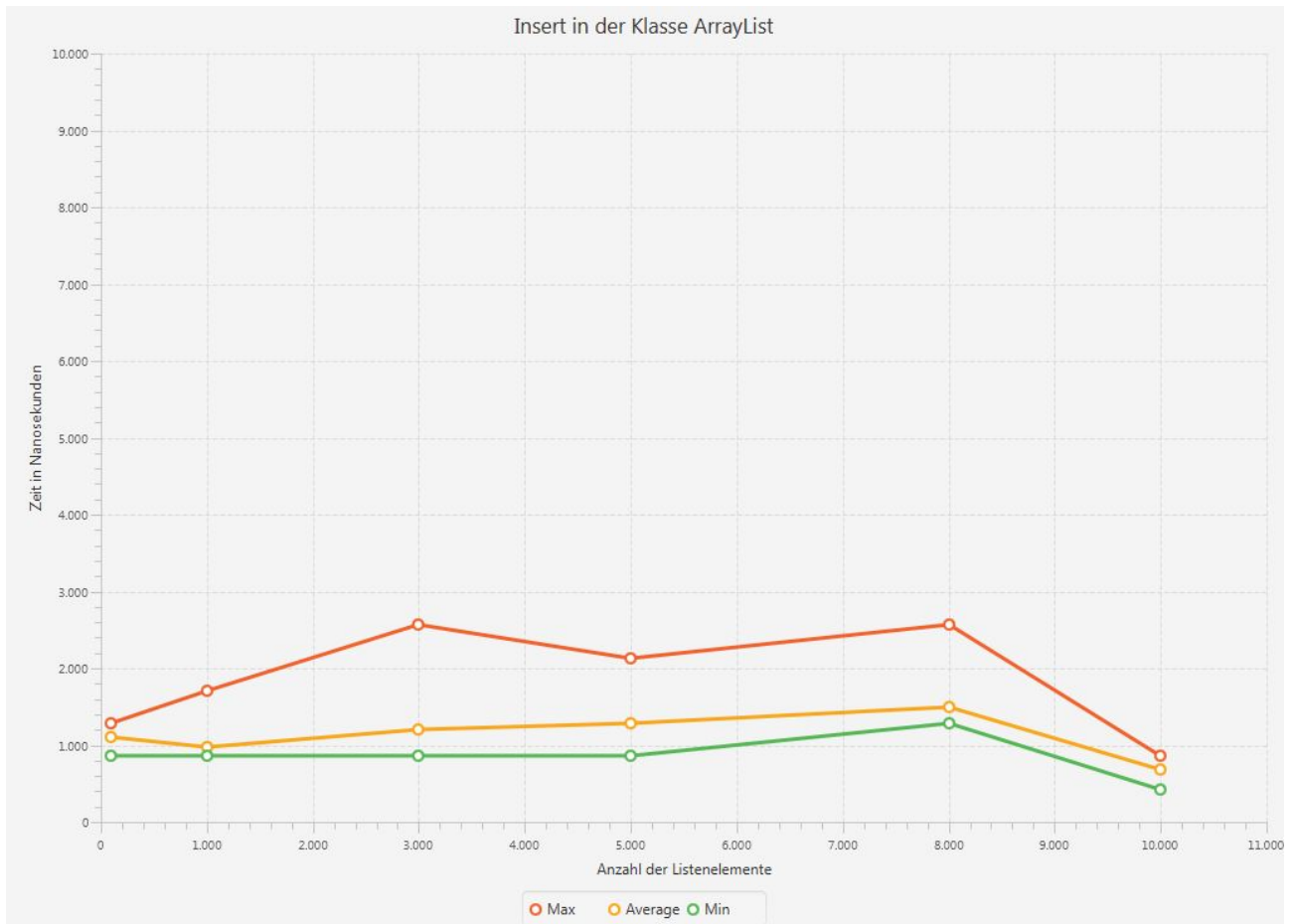


Abbildung 3.13: ArrayList - Insert

Merge

Merge ist eine Methode wobei ein ArrayList wird zu ein andere ArrayList verbinden. Die Methode **Merge** hat einen Konstanten Aufwand, wie der Grafik zeigt : $\theta(1)$. Wie bei der Insert Methode, der ArrayList wird sich mit ein anderen verbinden, und deshalb ist die Aufwand konstant. Bei dem Worst-case, trifft genau wie bei dem Insert, den ArrayList muss vergrößern um neue werte zu addieren.

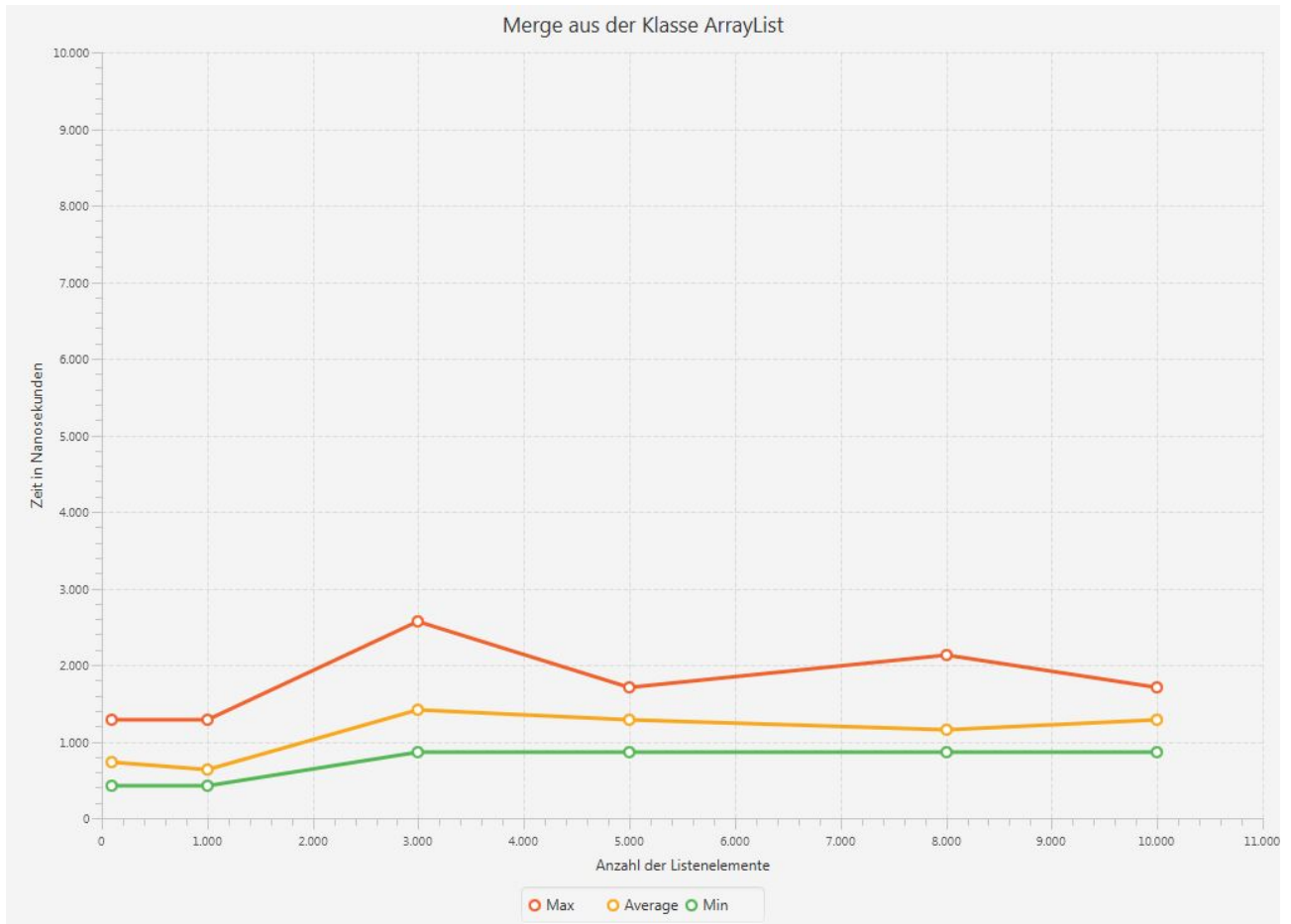


Abbildung 3.14: ArrayList - Merge

Remove

Remove ist eine Methode, die ein wert von der ArrayList gelöscht wird. Die Methode **Remove** aus der Klasse ArrayList hat einen linearen Aufwand, wie der Grafik zeigt : $\theta(n)$. Die Aufwand kommt dazu da weil die Element die gelöscht werden soll, muss erst gefunden werden, und dann gelöscht. Die Liste wird dann aber weiter suchen für ein andere Treffer.

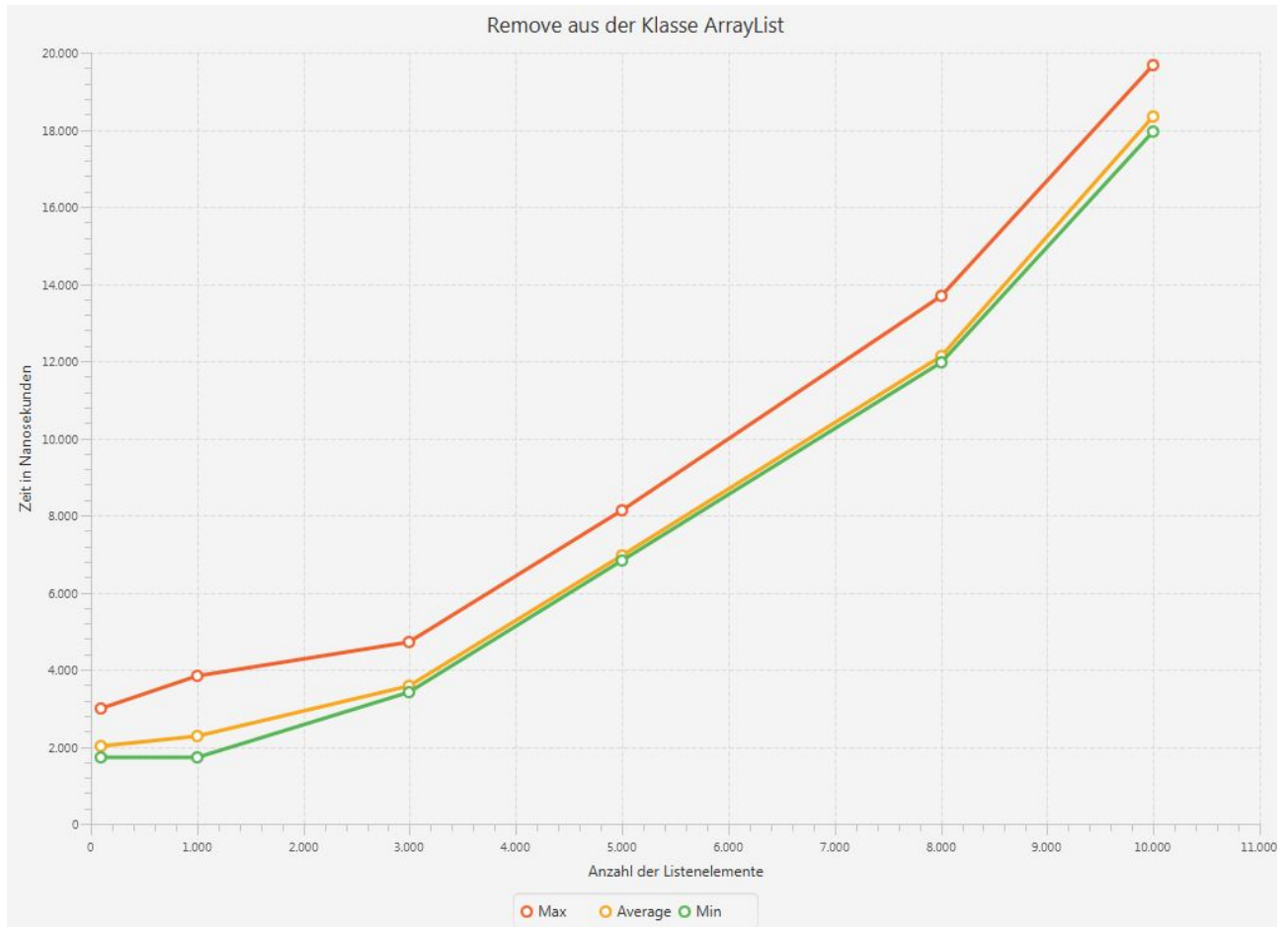


Abbildung 3.15: ArrayList - Remove

3.3.4 Schlussfolgerung

Die ArrayList ist wie eine Liste, wobei jeder Element seine eigene stelle hat. ArrayList ist eine sehr effektive Klasse, die ermöglicht ein Array zu haben, bei dem die Größe dynamisch verändert werden kann. Diese Klasse ist gut zunehmen, wenn man nicht weiß wie groß der Array sein soll. Im Vergleich mit der Vector Klasse, kann die ArrayList durch mehrere Threads abgerufen und bearbeitet werden.

3.4 LinkedList

3.4.1 Einleitung

Die linkedList in Java 8 ist eine Umsetzung der List und Deque Schnittstellen als doppelt verknüpfte Liste. Sie Implementiert alle optionalen Listenoperationen, und erlaubt alle Elemente (einschließlich null).

3.4.2 Methodenbeschreibung

- `add([index],e)` - fügt das spezifizierte Element am Ende der Liste / an der spezifizierten Position an
- `addAll([index], c)` - fügt die spezifizierte Collection am Ende der Liste / an der spezifizierten Position an
- `addFirst(e)` - fügt das spezifizierte Element am Anfang der Liste ein
- `addLast(e)` - fügt das spezifizierte Element am Ende der Liste ein
- `clear()` - entfernt alle Elemente aus der Liste
- `clone()` - erstellt eine flache Kopie der Liste
- `contains(o)` - gibt `true` zurück, wenn das spezifizierte Element in der Liste enthalten ist
- `element()` - gibt den Kopf der Liste zurück, ohne ihn zu entfernen
- `get(index)` - gibt das Element an der angegebenen Position zurück
- `getFirst()` - gibt das erste Element der Liste zurück
- `getLast()` - gibt das letzte Element der Liste zurück
- `remove([index])` - gibt das erste/spezifizierte Element in der Liste zurück und entfernt es
- `remove(o)` - entfernt das erste Auftreten des angegebenen Elements, sofern es vorhanden ist
- `size()` - gibt die Menge der Elemente in dieser Liste zurück
- `toArray()` - gibt ein Array mit allen Elementen der Liste in korrekter Reihenfolge zurück

3.4.3 Effizienzanalyse

ExtractMin

Alle Graphen für die Methode `remove` zeigen einen deutlichen Anstieg. Man kann erkennen, dass es zwischen den einzelnen Graphen kaum einen Abstand zu geben scheint. Recht deutlich ist zu erkennen, dass diese Methode für viele Elemente recht lange benötigt, sodass vom Gebrauch dieser Datenstruktur bei umfangreichen Daten im Zusammenhang von `removeMin` abzuraten ist. Es liegt recht nahe, dass diese es sich hier um einen nicht linearen Zusammenhang handelt. Ein exponentieller Zusammenhang scheint naheliegend, sodass vom Gebrauch dieser Methode abzuraten ist.

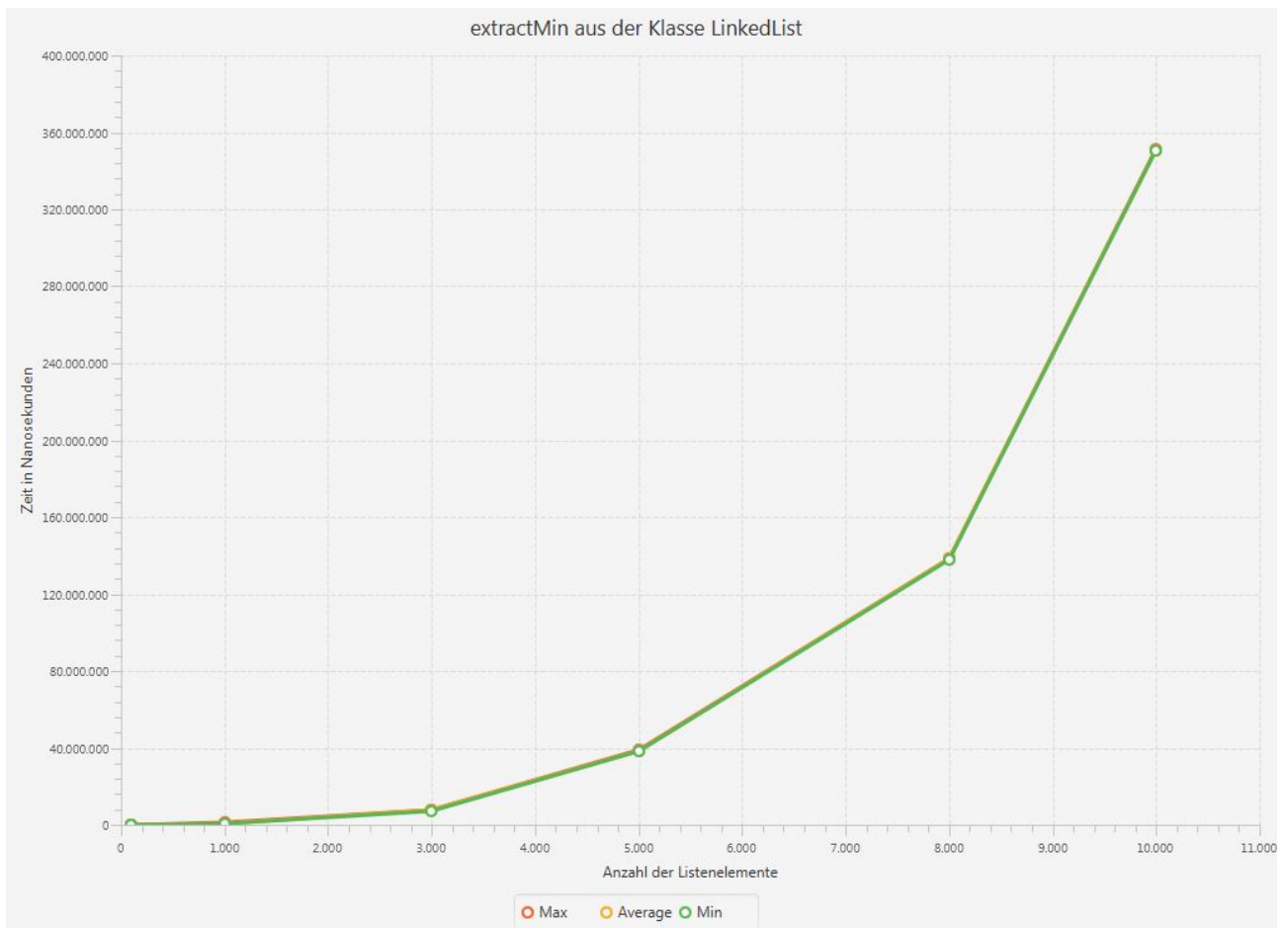


Abbildung 3.16: LinkedList - ExtractMin

GetMin

Es scheint ein recht ähnlicher Zusammenhang wie bei der Methode `extractMin` vorzuliegen. Jeder Graphen für diese Methode zeigt einen deutlichen Anstieg. Auch hier ist zu erkennen, dass es zwischen den einzelnen Graphen kaum einen Abstand zu geben scheint. Auch diese Methode benötigt für eine hohe Anzahl an Elementen viel Zeit, sodass vom Gebrauch dieser Datenstruktur bei umfangreichen Daten im Zusammenhang von `getMin` abzuraten ist. Es scheint auch hier kein linearer sondern exponentieller Zusammenhang vorzuliegen.

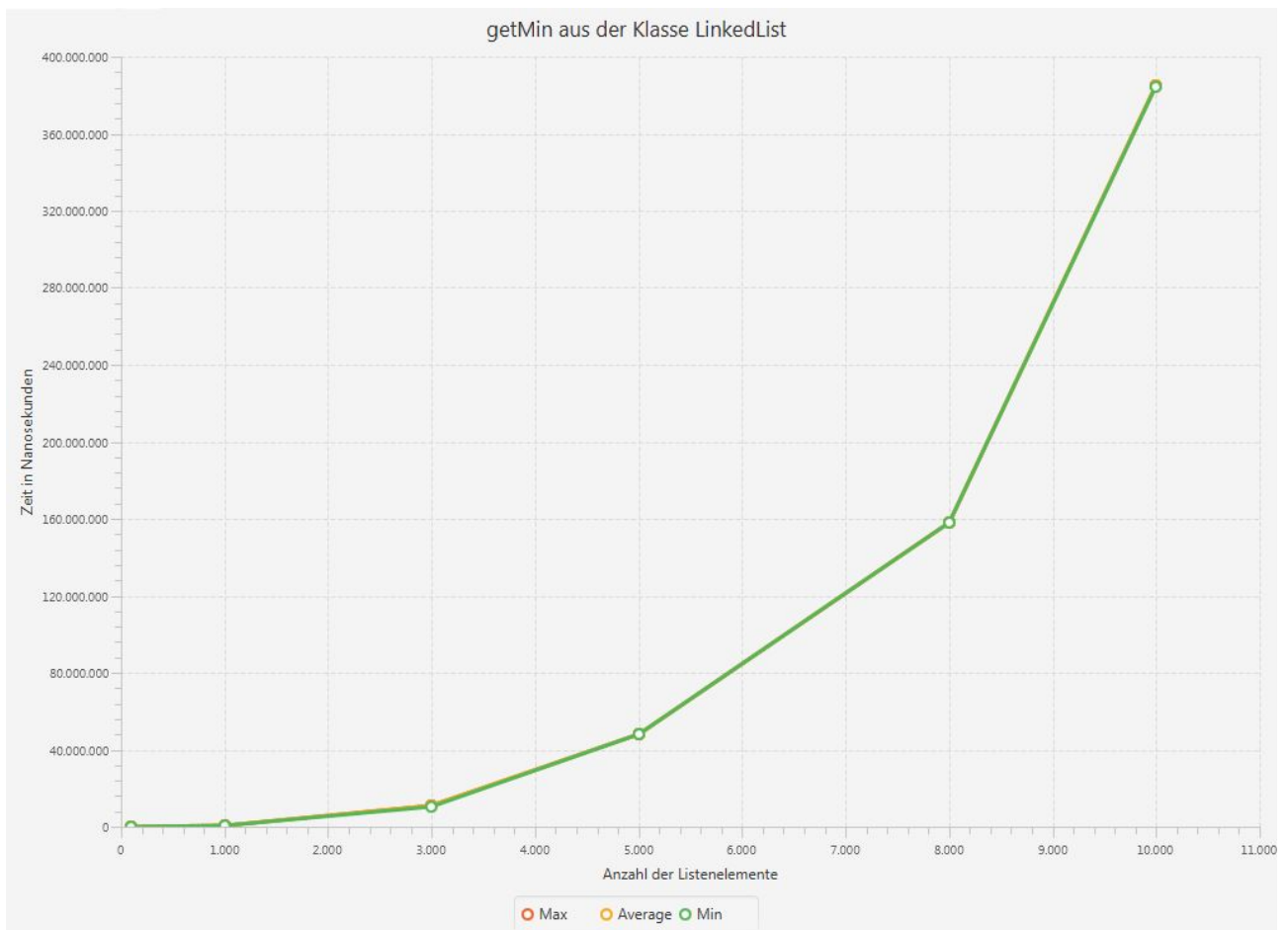


Abbildung 3.17: LinkedList - GetMin

Insert

Bei der Methode insert sind die Durchschnittsgraphen recht konstant gehalten. Man sieht, dass es ab einer Elementanzahl von 3000 Elementen sogar zu einem Abfall der benötigten Zeit für alle Graphen kommt und der jeweils erreichte Wert für 5000 Elemente für den jeweiligen Graphen im weiteren Verlauf des Graphen nicht überschritten wird. Auffällig ist auch, dass der Abstand vom roten zum grünen Graphen überall ungefähr gleich groß ist. Eine Abweichung von diesem Abstand wird wahrscheinlich durch fehlerhafte Messwerte verursacht, die in diesem Kontext zu vernachlässigen sind. Das selbe Verhalten lässt sich auch für den gelben und den grünen Graphen erkennen. Signifikant ist jedoch eine Spitze bei 1000 Elementen die jedoch direkt im weiteren Verlauf wieder abfällt. Hierbei scheint es sich erneut um einen Messfehler zu handeln, den man vernachlässigen kann. Es handelt sich hierbei um einen konstanten Aufwand $\mathcal{O}(k)$.

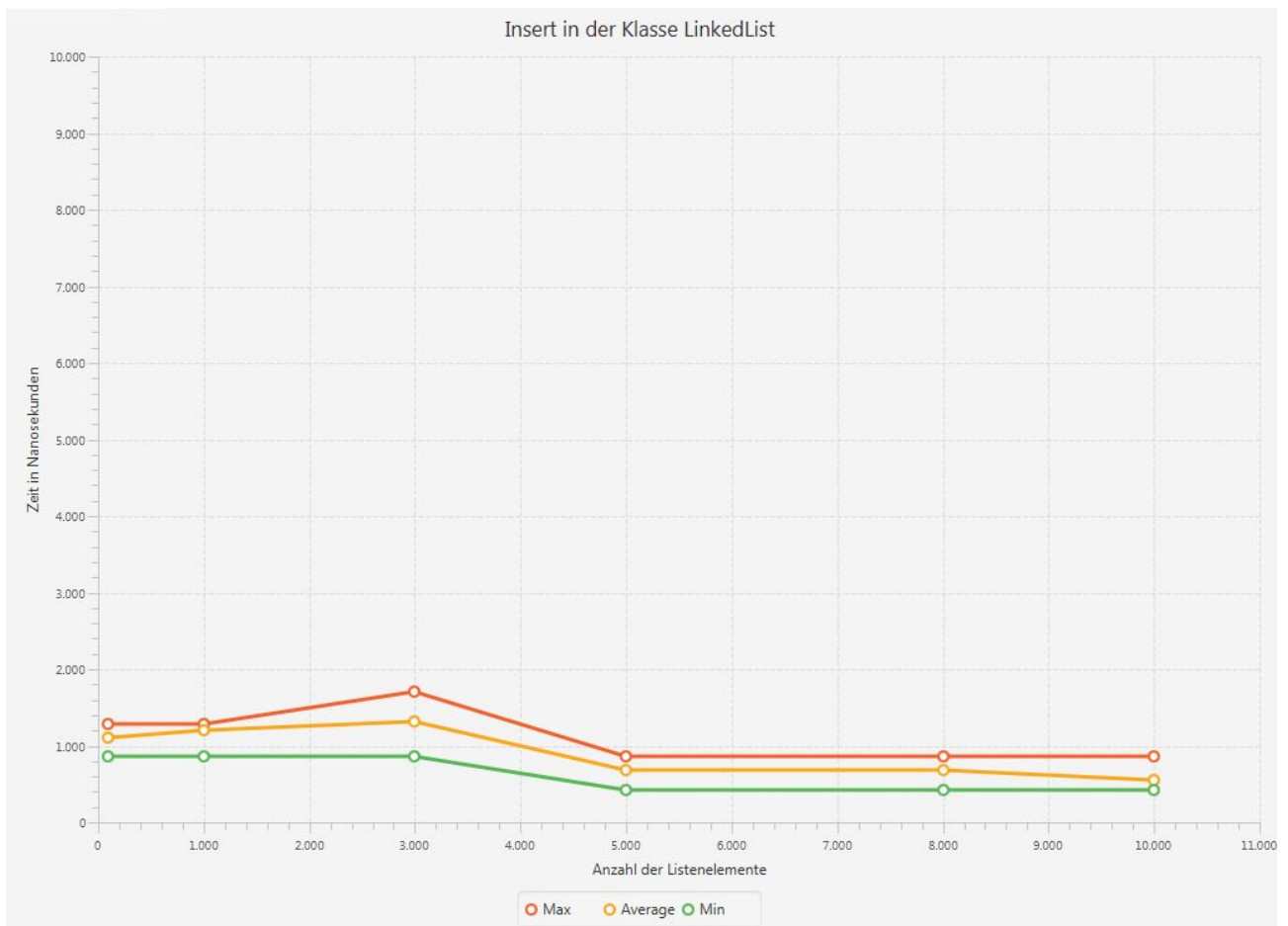


Abbildung 3.18: LinkedList - Insert

Merge

Bei der Methode `merge` ist der Durchschnittsgraph recht konstant um einen Wert von 2000 Nanosekunden gehalten. Das selbe Verhalten lässt sich für den grünen Graphen beobachten, dessen Wert sich recht konstant bei 1750 einpendelt. Der rote Graph weist kein so konstantes Verhalten auf wie die beiden anderen. Innerhalb der ersten 1000 Listenelemente kommt es zu einem deutlichen Anstieg auf einen Wert von 2600. Bis zum Erreichen von 3000 Listenelementen bleibt der Zeitaufwand jedoch recht konstant. Dannach fällt der Wert jedoch für den Messwert von 5000 Elementen ab, um daraufhin wieder auf das Niveau von 2600 Nanosekunden für 8000 Listenelemente anzusteigen. Dies legt den Schluss nahe, dass es sich bei dem erhaltenem Messwert von 5000 Listenelementen um eine fehlerhaften Messwert handelt. Es handelt sich bei der Methode `merge` für die Klasse `LinkedList` um einen konstanten Aufwand $\mathcal{O}(k)$.

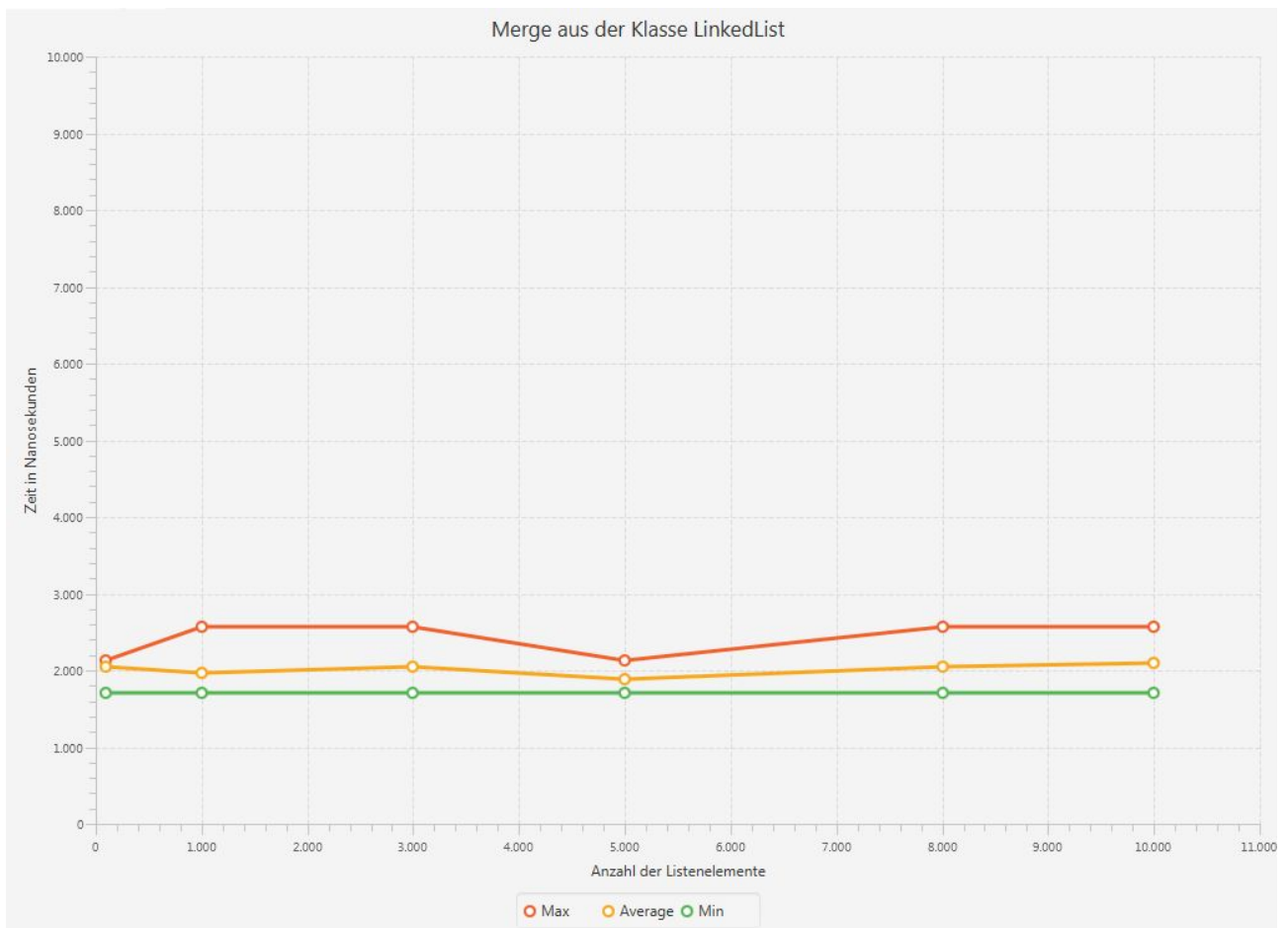


Abbildung 3.19: LinkedList - Merge

Remove

Bei der Methode `remove` ist der Durchschnittsgraph recht konstant um einen Wert von 1100 Nanosekunden gehalten. Das selbe Verhalten lässt sich für den grünen Graphen beobachten, dessen Wert sich recht konstant bei 850 eingependelt hat. Auffällig für diesen Graphen ist, dass es keinerlei Schwankungen innerhalb dieses Graphen gibt und sein benötigter Zeitaufwand konstant bleibt. Der rote Graph weist ebenfalls ein recht konstantes Verhalten auf wie die beiden anderen Graphen. Innerhalb dieses Graphen gab es eine Messspitze, für 5000 Listenelemente, die jedoch als wahrscheinlicher Messfehler vernachlässigt werden kann. Bis auf diese eine Spitze weisen alle Graphen auch einen recht konstanten Abstand zu den jeweils anderen Graphen auf. Es handelt sich bei der Methode `remove` für die Klasse `LinkedList` um einen konstanten Aufwand $\mathcal{O}(k)$.

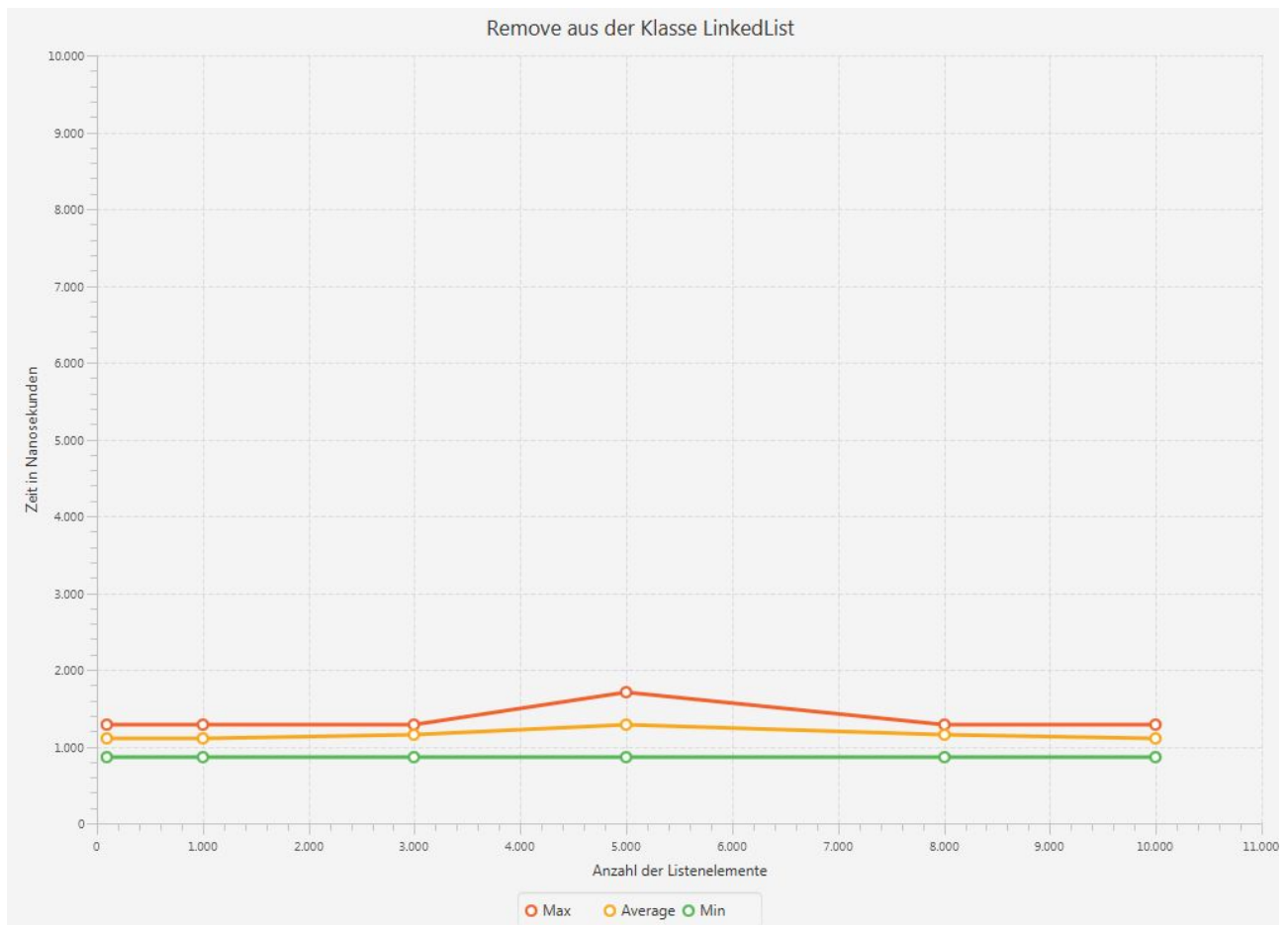


Abbildung 3.20: LinkedList - Remove

3.4.4 Schlussfolgerung

Von den Methoden `getMin` und `extractMin` für Instanzen der Klasse mit vielen Listenelementen ist abzuraten, da der Zeitaufwand exponentiell verläuft und sich recht schnell ein unwirtschaftliches Arbeiten einstellt. Da die Methoden `merge` und `remove` einen konstanten Zeitaufwand besitzen ist in diesem Kontext nur zu erwägen, ob es eine Klasse gibt, die diese Aufgaben mit einem geringeren, Zeitaufwand pro Element erledigen können. Diese Methoden eignen sich also für den praktikablen Einsatz selbst für sehr große Listen.

3.5 Vector

3.5.1 Einleitung

Vektoren sind ähnlich wie ein `ArrayList` aber ein `Vector` ist synchronisiert, und enthält viele Methoden die nicht zu den `Collection Frameworks` gehören. Vektoren sind sehr hilfreich wenn die Größe des Arrays nicht gewusst ist oder wenn ein Array größer werden soll je länger ein Programm läuft.

Die `Vector` Klasse hat 4 Konstruktoren:

- `Vector()` Erzeugt ein `Vector` mit Größe 10.
- `Vector(int size)` Erzeugt ein `Vector` mit eingegebene Größe.
- `Vector(int size, int incr)` Erzeugt ein `Vector` mit eingegebene Größe und Inkrement, der Inkrement bestimmt wie groß der `Vector` wird wenn der `Vector` größer wird.
- `Vector(Collection c)` Erzeugt ein `Vector` mit Elementen von den `Collection c`.

3.5.2 Methodenbeschreibung

- `add()/addAll()` - genau wie bei ein `ArrayList`.
- `addElement(Object o)` - addiert den `Object` am ende des `Vectors`, `Vector` größe wird um Eins erhöht.
- `capacity()` - gibt die aktuelle Kapazität des `Vectors` zurück.
- `clear()` - löscht alle Elementen in der `Vector`.
- `contains(Object o)` - prüft ob ein eingegebene `Object` in der `Vector` ist.
- `containsAll(Collection c)` - prüft ob Alle Elementen in der `Collection c` in der `Vector` sind.
- `copyInto(Object[] array)` - kopiert alle Elementen in der `Vector` zu ein eingegebene `Array`.
- `elementAt(int index)` - gibt `Element` an der eingegebene `Stelle` zurück.
- `elements()` - gibt ein Aufzählung aller Elementen in der `Vector` zurück
- `firstelement()` - gibt den erste `Element` in `Vector` zurück (`index 0`)
- `get(int index)` - gibt den `Element` an der eingegebene `Stelle` zurück
- `indexOf(Object o)` - sucht nach den ersten Treffer von `Element` in der `Vector`
- `insertElementAt(Object o, int index)` - addiert `Element` in eingegebene `stelle`.
- `isEmpty()` - prüft ob `Vector` leer ist
- `lastElement()` - gibt den letzten `Element` in der `Vector` zurück
- `remove(int index)` - löscht den `Element` in eingegebene `Stelle`.
- `removeAll(Collection c)` löscht alle Elementen in `Vector` die dieser `Collection` enthält.
- `removeAllElements()` - löscht alle Elementen in den `Vector` und verändert die größe zu `Null`.
- `removeElement(Object o)`- löscht erste Treffer von `Element` in der `Vector`
- `set(int index, Object o)` - ersetzt den `Element` von eingegebene `Index` für ein neues `Element`
- `setSize()` - verändert die Größe des `Vectors`
- `size()` - gibt der Größe des `Vectors` zurück.
- `toArray()` - gibt ein `Array` mit alle Elementen in Reihenfolge in diesem `Vector` zurück
- `toString()` - gibt einer `String` von alle Elementen in den `Vector`.
- `trimToSize()` - verkleinert die Größe des `Vectors` zu aktuelle oder benutzte Größe.

3.5.3 Effizienzanalyse

ExtractMin

Alle Graphen für die Methode `remove` zeigen einen deutlichen Anstieg.

Man kann erkennen, dass alle Graphen ungefähr den selben Abstand zu einander haben.

Recht deutlich ist zu erkennen, dass diese Methode für viele Elemente recht lange benötigt, sodass vom Gebrauch dieser Datenstruktur bei umfangreichen Daten im Zusammenhang von `removeMin` abzuraten ist.

Nichts desto trotz liegt hier ein linearer Zusammenhang von $O(n)$ vor.

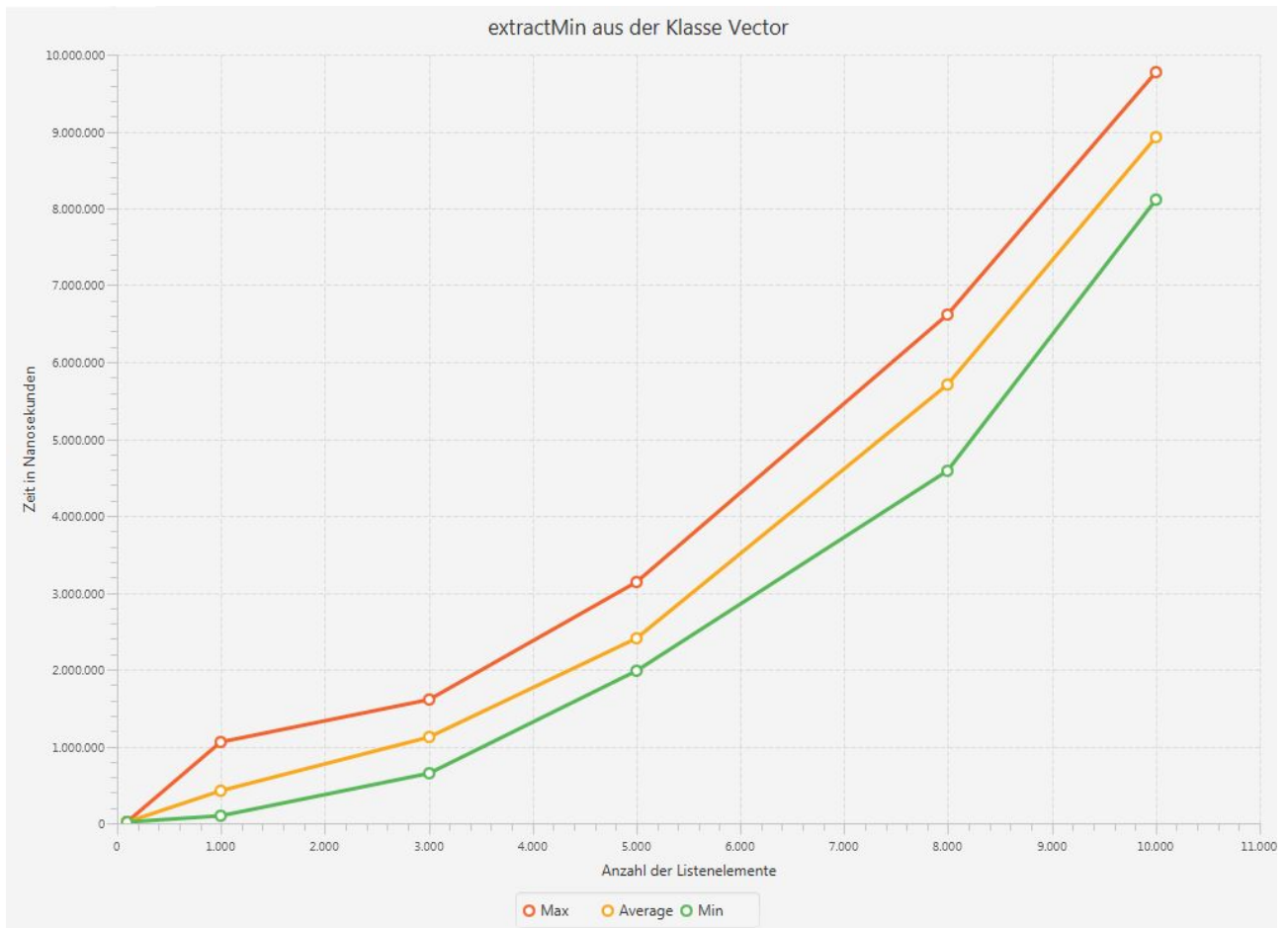


Abbildung 3.21: Vector - ExtractMin

GetMin

Alle Graphen für die Methode getMin zeigen einen deutlichen Anstieg.

Die benötigte Zeit liegt für alle drei Graphen bis zu einer Elementanzahl von 5000 sehr nah beieinander.

Der Wert für den roten Graphen für 8000 Elemente liegt deutlich über dem des mittleren Graphen und steigt im weiteren Verlauf nicht so stark an wie bisher. Der Schluss, dass es sich bei diesem Wert um einen fehlerhaften Wert handelt, der zu vernachlässigen ist, liegt also nahe.

Es liegt hier ein linearer Zusammenhang von $O(n)$ vor.

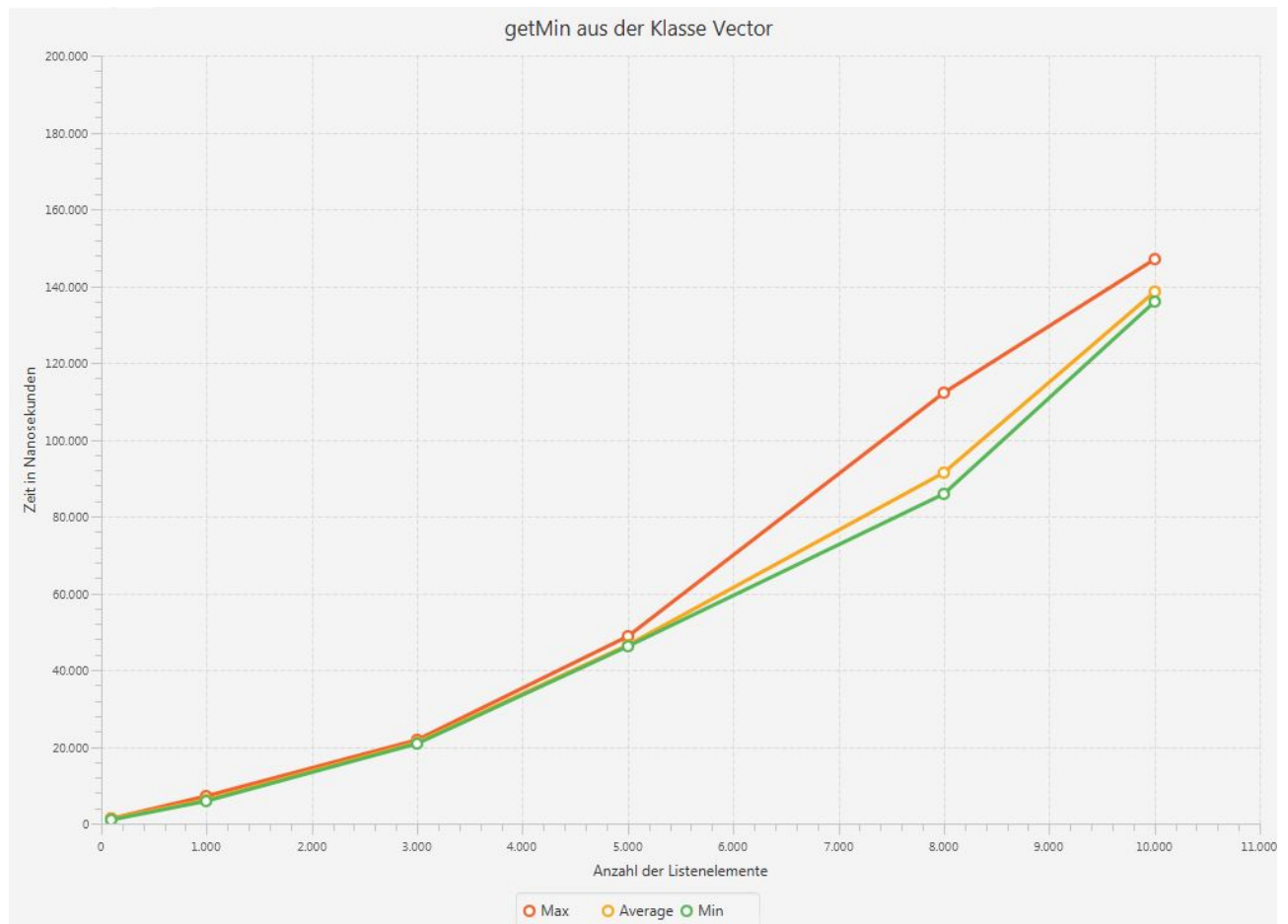


Abbildung 3.22: Vector - GetMin

Insert

Bei der Methode insert ist der Durchschnittsgraph recht konstant gehalten.

Ab einer Anzahl von 3000 Elementen kommt es jedoch zu einem leichten Anstieg der benötigten Zeit auf einen Spitzenwert, der bei 8000 Elementen erreicht wird.

Dannach fällt der Graph jedoch wieder ab, was darauf schließen lässt, dass dieser Wert nicht direkt im Zusammenhang mit der Klasse Vector steht, sondern ein fehlerhafter ist und somit vernachlässigt werden kann.

Der grüne Graph zeigt ein ähnliches Verhalten und kann streckenweise als 'parallel zum gelben Graphen verlaufend' beschrieben werden. Alle seine Werte liegen unter dem des gelben Graphen.

Die Werte des roten Graphen zeigen ein ähnliches Verhalten wie die anderen beiden Graphen. Signifikant ist jedoch eine Spitze bei 1000 Elementen die jedoch direkt im weiteren Verlauf wieder abfällt. Hierbei scheint es sich erneut um einen Messfehler zu handeln, den man vernachlässigen kann.

Es handelt sich hierbei um einen konstanten Aufwand.

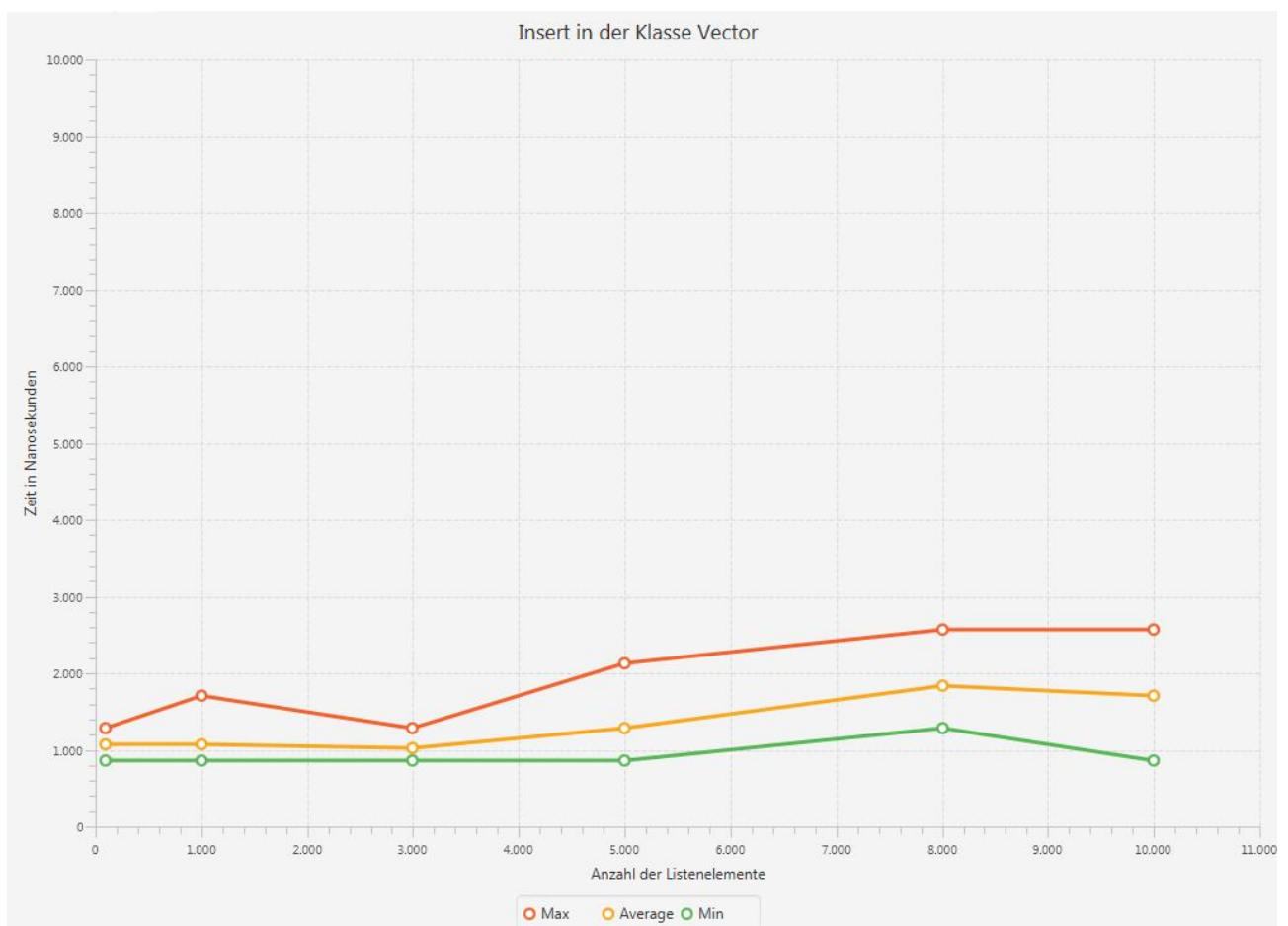


Abbildung 3.23: Vector - Insert

Merge

Bei der Methode merge ist der Durchschnittsgraph recht konstant um einen Wert von 1400 gehalten. Hier gibt es nur bei dem letzten Wert einen leichten Anstieg, den man aber vernachlässigen kann. Das selbe Verhalten lässt sich für den roten Graphen beobachten, dessen Wert um einen Zeitwert von 2200 schwankt.

Bei dem grünen Graphen sieht man diesen Sachverhalt deutlicher, da alle Werte um einen Wert von 1200 schwanken. Der wert für 5000 Listenelemente liegt jedoch deutlich unter diesem Wert und ist somit als wahrscheinlicher Messfehler vernachlässigbar.

Dementsprechend handelt es sich bei dieser Methode um eine Methode mit konstanten Aufwand.

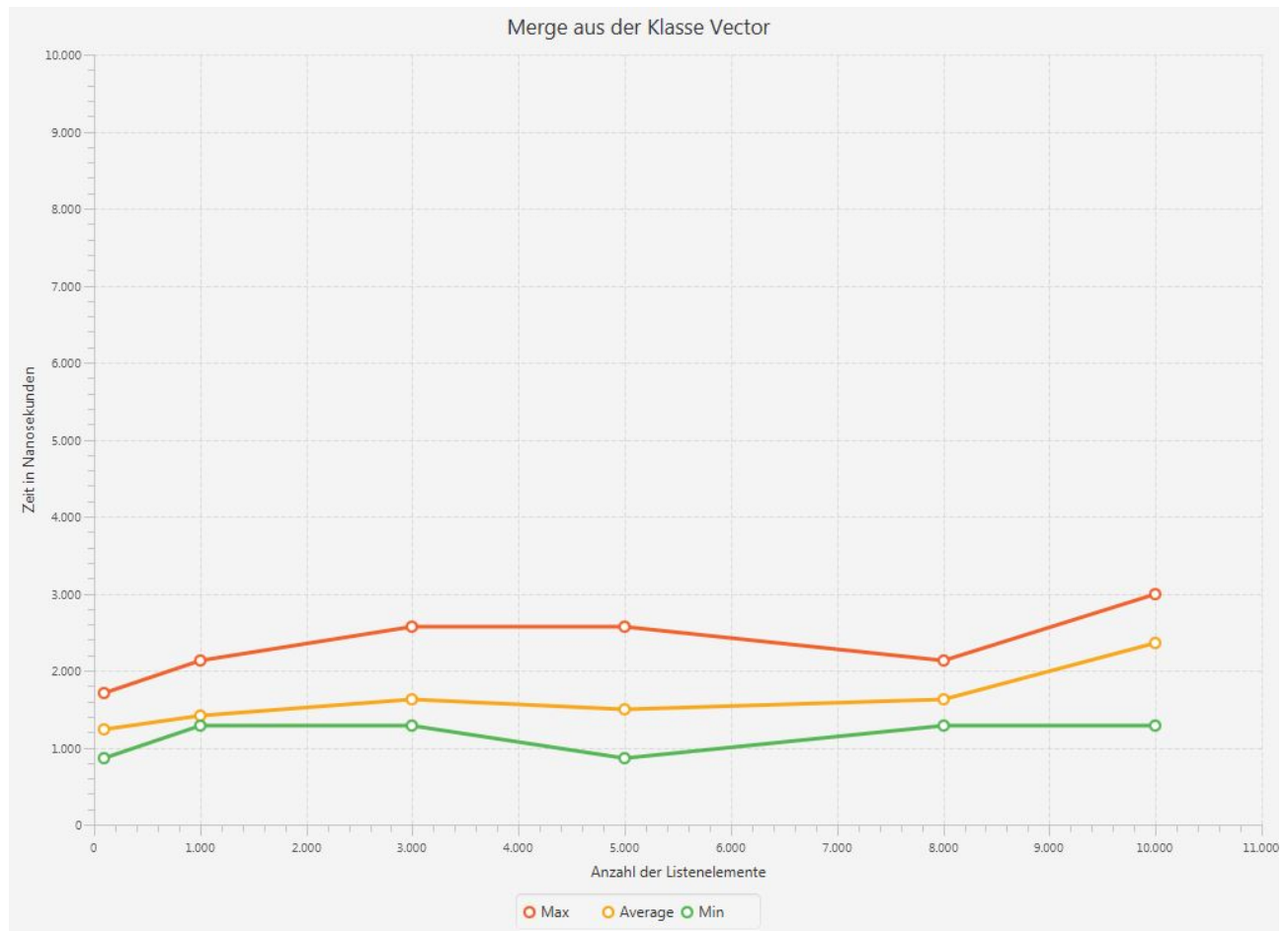


Abbildung 3.24: Vector - Merge

Remove

Alle Graphen für die Methode `remove` zeigen einen deutlichen Anstieg.

Recht deutlich ist zu sehen, dass die Werte des roten Graphen sichtlich über denen der anderen beiden liegen, was darauf schließen lässt, dass diese Methode im Worst-Case deutlich langsamer ist als sie sonst wäre.

Alle Graphen weisen jedoch einen recht linearen Verlauf auf, sodass ein linearer Aufwand von $O(n)$ nahe liegt.

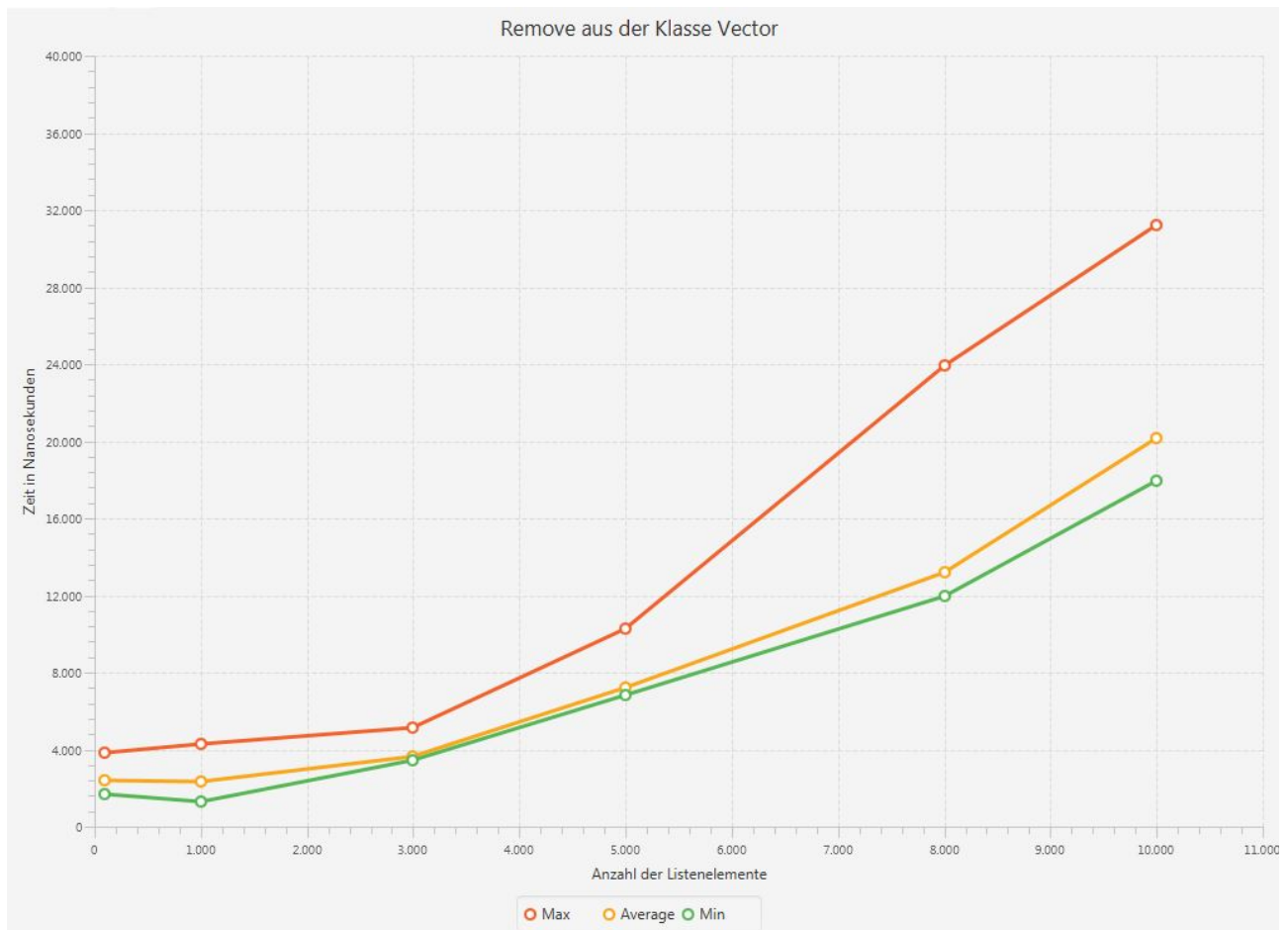


Abbildung 3.25: Vector - Remove

3.5.4 Schlussfolgerung

Da die Methoden `merge` und `insert` einen konstanten Zeitaufwand besitzen, ist in diesem Kontext nur zu erwägen, ob es eine Klasse gibt, die diese Aufgaben mit einem geringerem Zeitaufwand pro Element erledigen kann.

Diese Methode eignen sich also für den praktikablen Einsatz selbst für Instanzen von Vektoren mit einer hohen Anzahl an Elementen.

3.6 Queue

3.6.1 Einleitung

Eine Queue ist eine Warteschlange. In ihr werden Daten in einer bestimmten Reihenfolge abgelegt. Ähnlich wie der Stack braucht auch die Queue nicht viele Methoden, um ihre volle Funktionalität zu

erhalten.

Mit Hilfe der Methode `add()` kann ein neues Element der Queue hinzugefügt werden. Die Queue arbeitet nach dem FIFO-Prinzip. FIFO steht für „First In First Out“ und bedeutet, dass das zuerst eingefügte Element auch zuerst wieder ausgegeben wird. Das heißt, neu eingefügte Elemente werden an das Ende der Warteschlange gesetzt.

Um das vorderste Element aus der Queue auszulesen wird die Methode `peek()` benötigt. Diese liest das Element lediglich, wobei die Methode `remove()` das Element nicht nur liest, sondern es außerdem aus der Queue entfernt.

Wie bei jeder Datenstruktur ist es auch bei der Queue sinnvoll, eine Methode `size()` zu implementieren, die die Länge der Warteschlange zurückliefert.

In Java ist die Queue ein Interface, das sich auf die Super-Interfaces `Collection` und `Iterable` zurückführen lässt.

3.6.2 Methodenbeschreibung

In diesem Abschnitt werden die wichtigsten Methoden der Queue beschrieben, die von Java zur Verfügung gestellt werden.

- `add(E item)`
 - fügt das als Argument übergebene Element der Queue an letzter Stelle hinzu
- `peek()`
 - liefert das vorderste Element der Queue zurück ohne es aus der Warteschlange zu löschen
- `remove()`
 - liefert das vorderste Element der Queue zurück und löscht es aus der Warteschlange
- `size()`
 - gibt die Anzahl der Elemente der Queue zurück

3.6.3 Effizienzanalyse

Bei der Methode „extractMin“, zeigen alle drei Graphen einen linearen Aufwand. Sie steigen auch alle gleichmäßig an. Daran kann man klar herauslesen, dass es sich hier um einen linearen Aufwand handelt.

Linearer Aufwand $\mathcal{O}(n)$

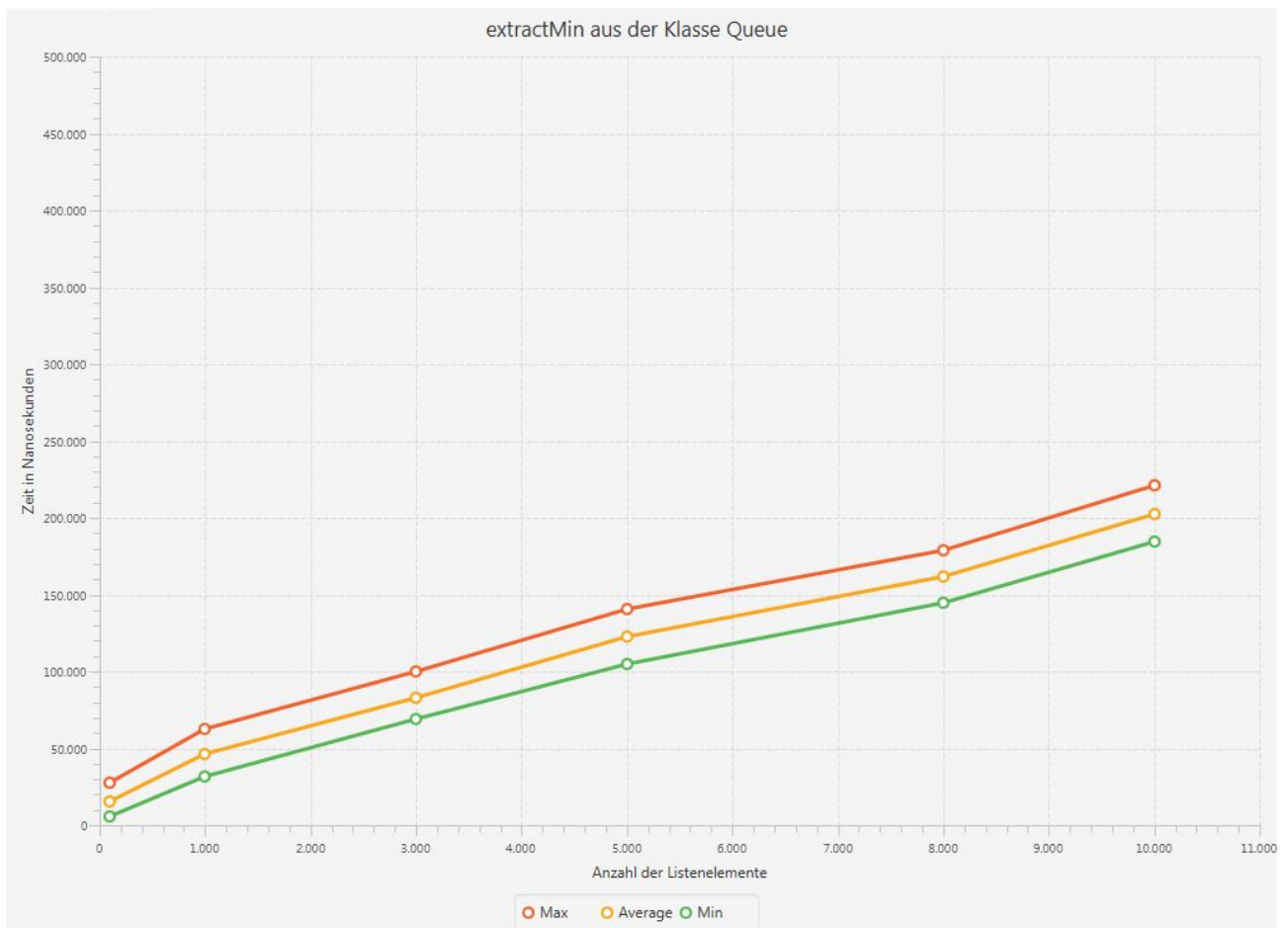


Abbildung 3.26: Queue - ExtractMin

Bei der Methode „getMin“ sieht man bei dem Graphen für die Maximalwerte, dass zwei Punkte (bei 3000 und 5000 Listenelementen) ausreißen. Die Anfangs- und auch die Endwerte dieses Graphen sind sehr linear gehalten, genauso wie die Graphen für die Minimal- und Durchschnittswerte. Daher lässt sich auch hier feststellen, dass der Aufwand linear ist.

Linearer Aufwand $\mathcal{O}(n)$

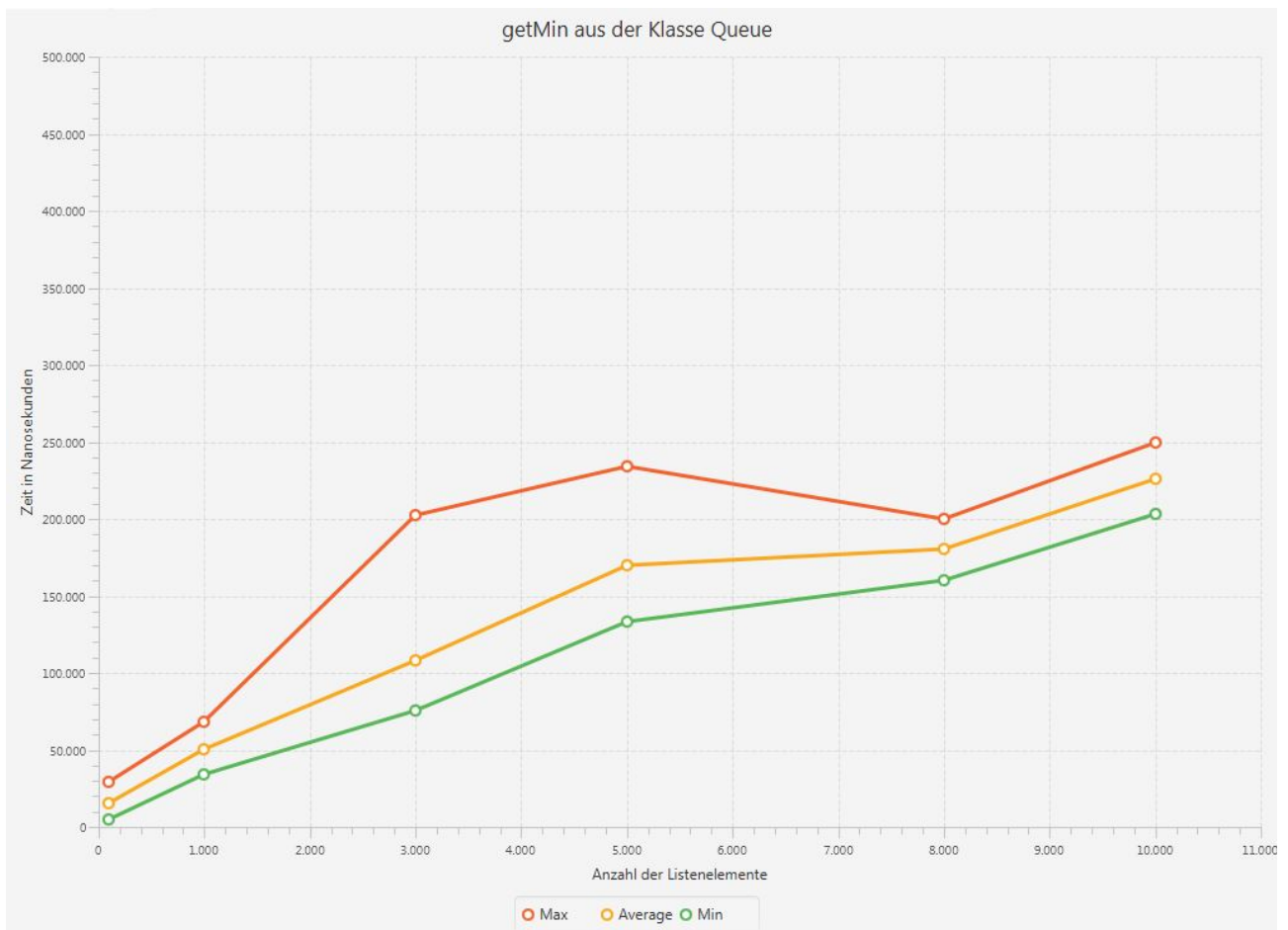


Abbildung 3.27: Queue - getMin

Bei der Methode „insert“ ist der Durchschnittsgraph recht konstant gehalten. Hier gibt es nur beim ersten und zweiten Wert einen leichten Anstieg, den man aber vernachlässigen kann. Beim grünen Graphen sieht man sehr deutlich, dass alle Werte exakt auf einer Linie sind und den gleichen Wert besitzen. Bei dem Graphen für die Maximalwerte liegt die zeitliche Differenz nur 500 Nanosekunden auseinander, daher kann man sagen, dass diese Methode einen konstanten Aufwand besitzt. Konstanter Aufwand $\mathcal{O}(1)$

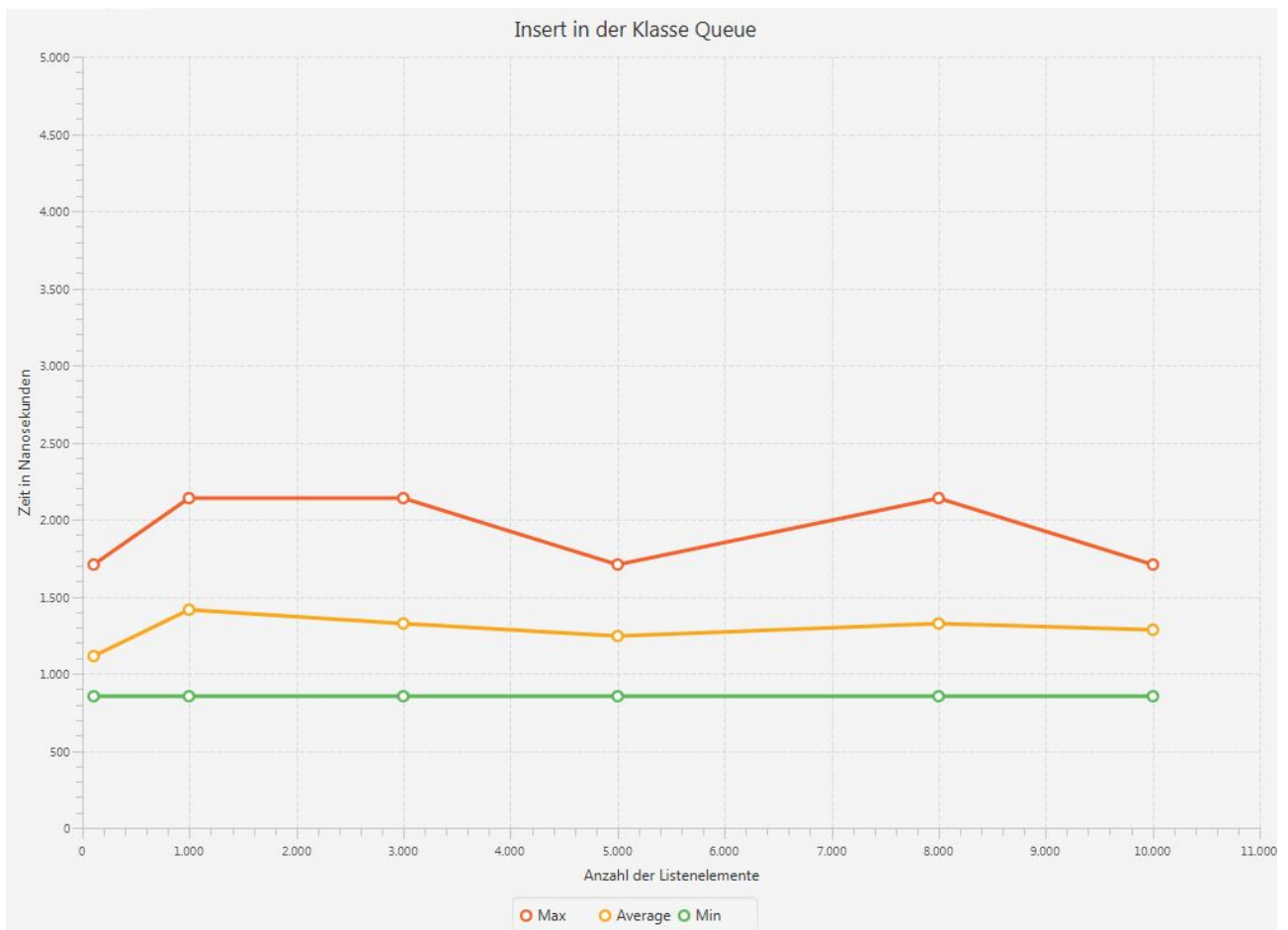


Abbildung 3.28: Queue - Insert

Die Methode „merge“ weist bei dem grünen Graphen eine leichte Steigung auf. Der mittlere Graph sieht recht linear aus, zudem könnte man eine Linie ziehen und die Punkte die über, bzw unter dieser Linie sind, haben den gleichen Abstand zu dieser Linie. Der Graph für die Maximalwerte schwankt stark und zeigt zum Ende einen großen Anstieg. Zieht man alle Graphen in Betracht kommt man auch hier zum Ergebnis, dass es sich um einen linearen Aufwand handelt, auch wenn die Ergebnisse mehr schwanken.

Linearer Aufwand $\mathcal{O}(n)$

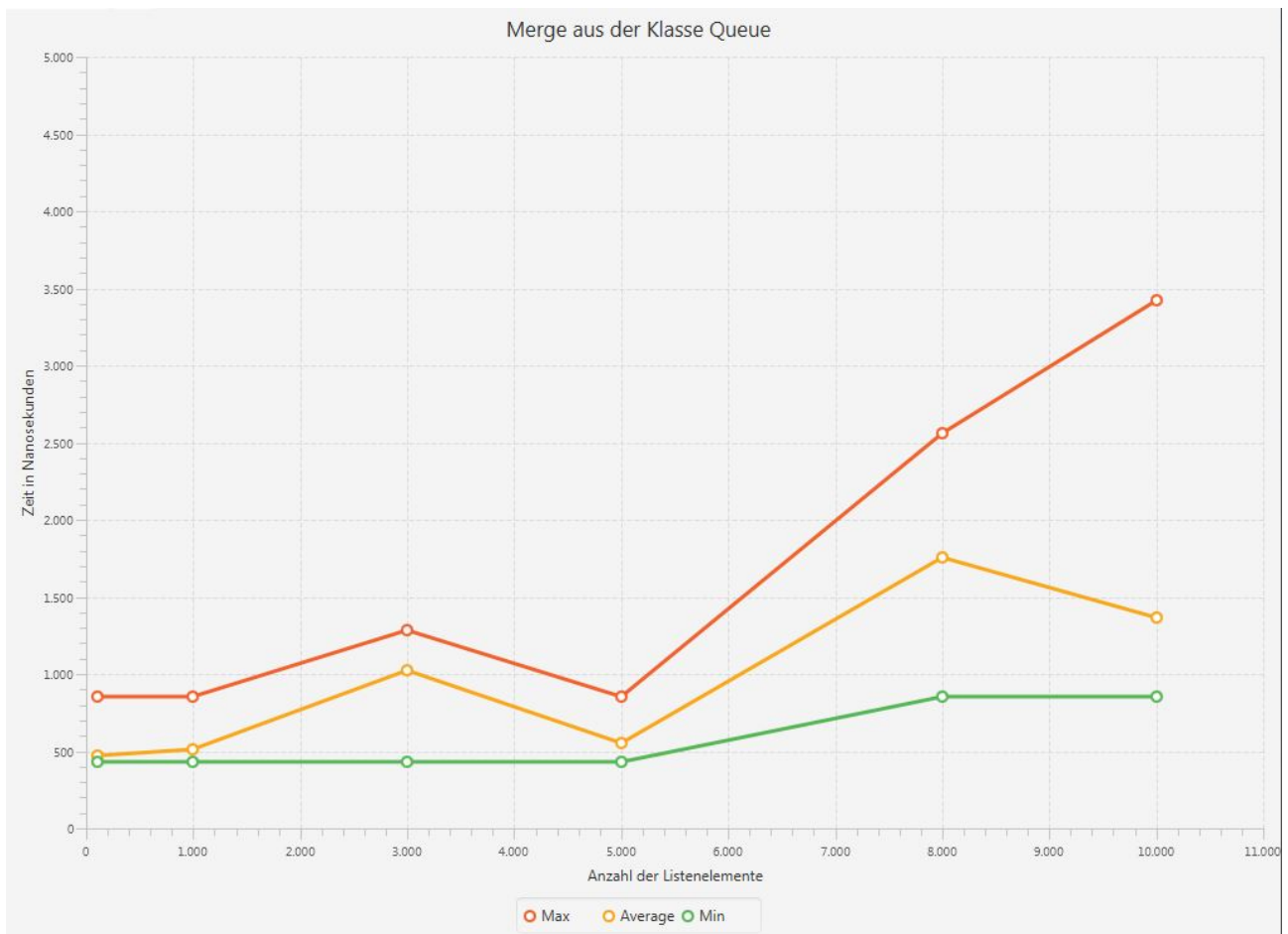


Abbildung 3.29: Queue - Merge

Die Methode „remove“ hat bei dem Graph für die Minimalwerte an der stelle 5000 Listenelemente seinen Maximalwert erreicht und bei mehr Listenelementen sinkt der Graph leicht. Auch bei der Durchschnittskurve liegt das Maximum bei 5000 Listenelementen. Bei letzten Graph sieht man deutlich einen sehr starken anstieg bis 5000 Listenelementen, danach bleibt die Zeit die benötigt wird konstant. Somit kann man hier sagen, dass es sich um einen logarithmischen Aufwand handelt. Logarithmischer Aufwand $\mathcal{O}(\log n)$

3.6.4 Schlussfolgerung

Insgesamt ist die Queue von ihrem Aufwand her sehr gut, da der Aufwand meist linear ist. Die Methode „insert“ ist hier besonders zu empfehlen, da der Aufwand konstant ist. Der logarithmische Aufwand der „remove“ Methode ist für wenige Listenelemente nicht so gut, sobald es aber um große Mengen geht kann man auch diese gut verwenden.

Die einzige Methode die gar nicht für die Queue geeignet ist, ist „decreaseKey“, da es nicht möglich war diese zu Implementieren.

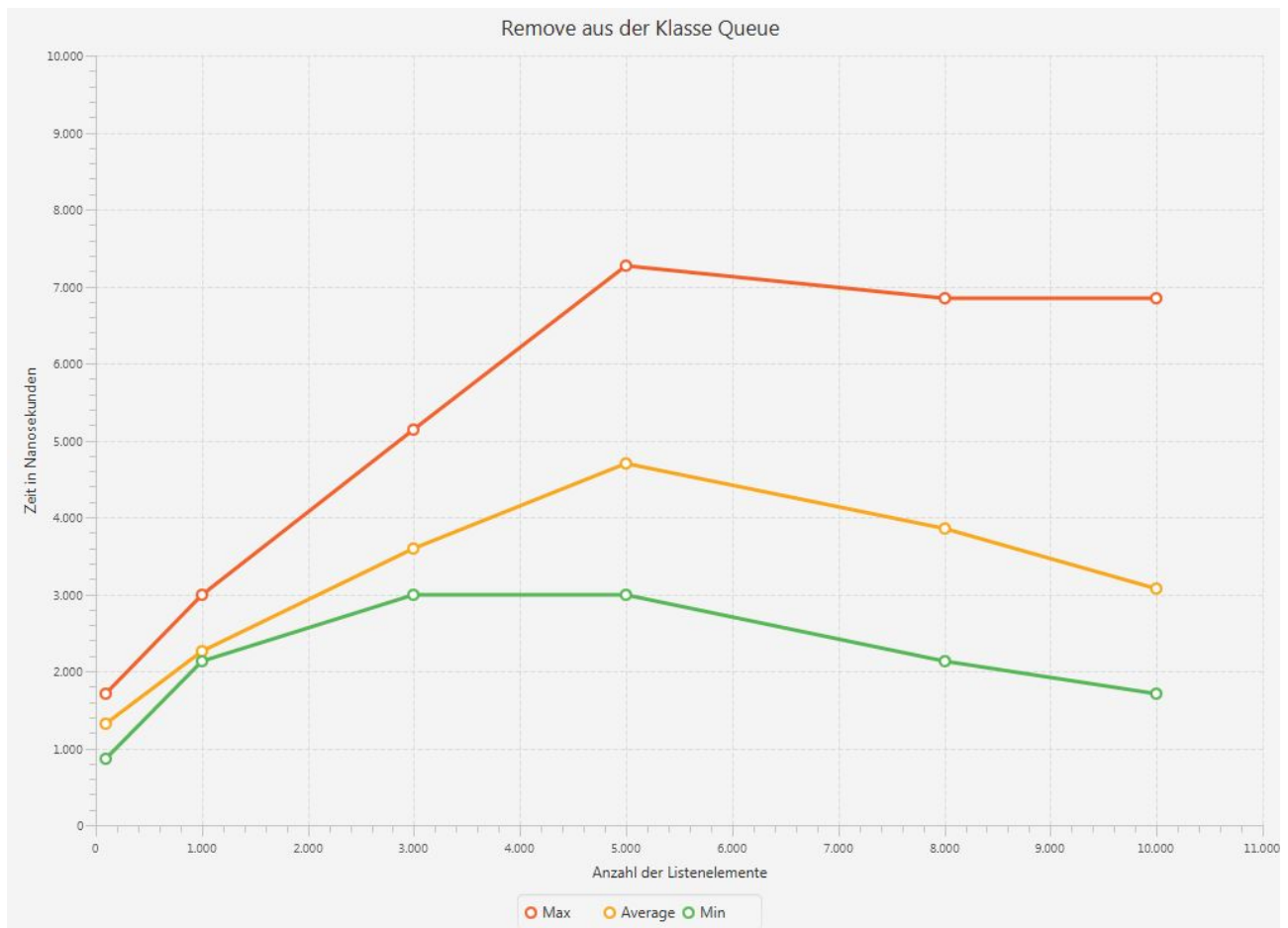


Abbildung 3.30: Queue - Remove

3.7 Stack

3.7.1 Einleitung

Ein Stack ist ein Stapelspeicher bzw. Kellerspeicher. Das Besondere am Stack ist, dass er für seine Funktionalität nur wenige Methoden braucht.

Um Elemente in den Stack zu legen wird die Methode *push()* benötigt. Dieses Element wird an vor-derster bzw. oberster Stelle in die vorhandenen Daten eingebunden. Der Stack arbeitet nach dem LIFO-Verfahren. Die Abkürzung steht für „Last In First Out“ und das bedeutet, dass das zuletzt hinzugefügte Element auch als erstes wieder aus dem Stack herausgenommen werden muss.

Um dieses neueste Element aus dem Stack zu entfernen, wird die Methode *pop()* verwendet.

Außerdem wird noch mindestens eine Methode *size()* benötigt, die die Länge des bzw. die Anzahl der Elemente im Stack zurückgibt.

Sinnvoll ist vielleicht noch eine Methode *peek()*, die das oberste Element des Stacks zurückgibt, ohne dieses aus dem Stack zu entfernen.

In Java ist der Stack eine Klasse, die direkt von der Superklasse *Vector* erbt. *Vector* lässt sich über *AbstractList* und *AbstractCollection* auf die Klasse *Object* zurückführen. Damit stehen Stack auch sämtliche Methoden dieser Superklassen zur Verfügung.

3.7.2 Methodenbeschreibung

Im Folgenden werden die von Java für den Stack zur Verfügung gestellten Methoden beschrieben, die wichtig sind und in der Implementation verwendet wurden.

- *empty()*

- prüft, ob der Stack Elemente enthält und liefert einen boolean-Wert zurück
- `peek()`
 - liefert das zuletzt zum Stack hinzugefügte Element zurück, ohne es aus dem Stack zu löschen
- `pop()`
 - gibt das zuletzt zum Stack hinzugefügte Element zurück und entfernt es aus dem Stack
- `push(E item)`
 - fügt das als Parameter übergebene Element dem Stack hinzu
- `size()`
 - gibt die Anzahl, der im Stack enthaltenen Elemente zurück

3.7.3 Effizienzanalyse

Bei der Methode „ExtractMin“, zeigt der Punkt bei 8000 Listenelementen eine geringere Zeit die benötigt wurde. Die restlichen Punkte dieses Graphen liegen auf einer Linie. Der Graph mit den Durchschnittswerten steigt konstant aber recht schnell an. Der rote Graph sieht dem grünen Graph sehr ähnlich, da auch er einen Ausreißerwert bei 8000 besitzt und die restlichen Punkte linear sind. Somit kann man hier feststellen, dass diese Methode einen linearen Aufwand hat.

Linearer Aufwand $\mathcal{O}(n)$

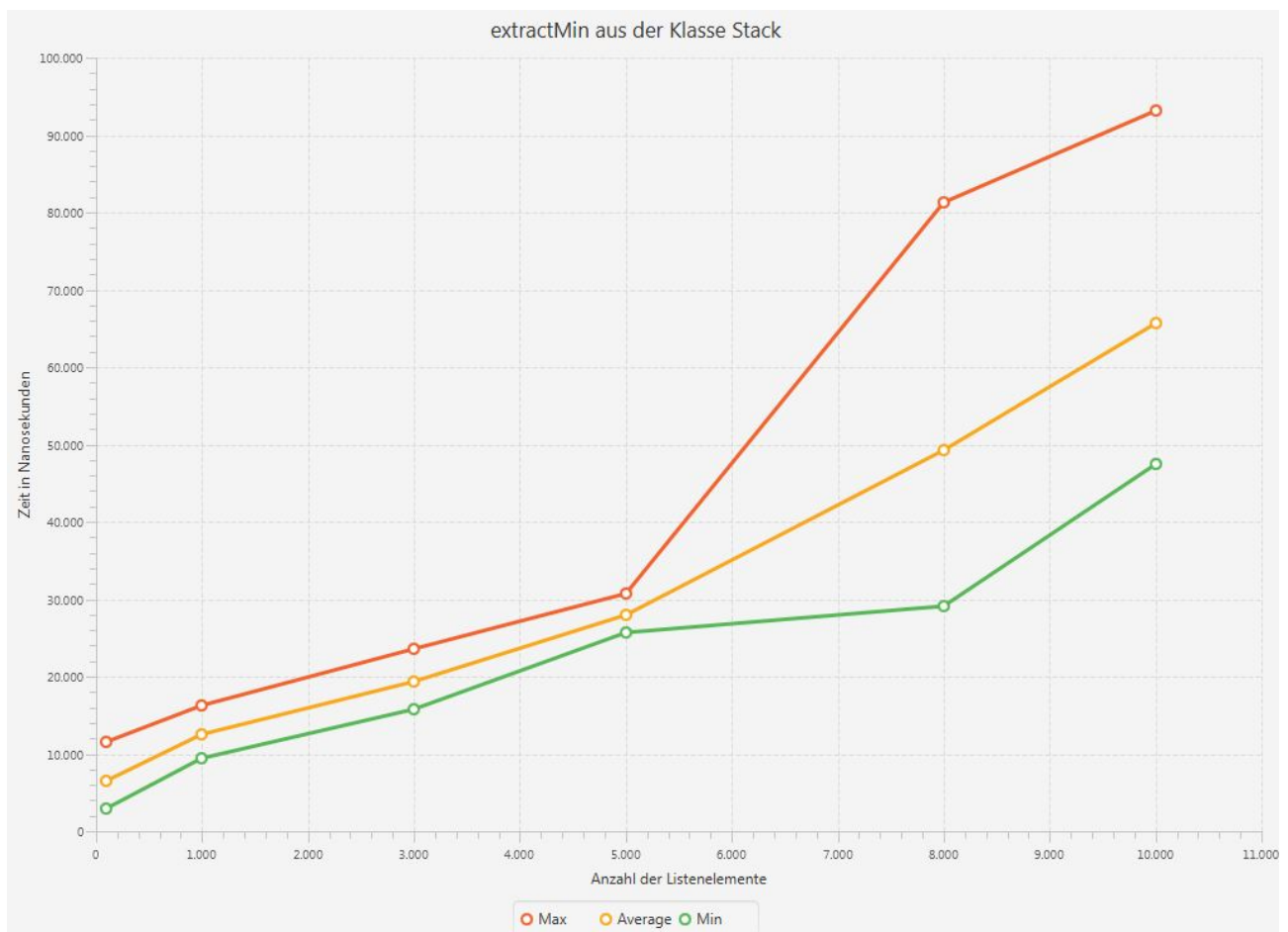


Abbildung 3.31: Stack - ExtractMin

Bei der Methode „GetMin“ sieht der untere Graph sehr linear aus. Auch der Durchschnittsgraph sieht linear aus, hat aber eine Tendenz zu einer logarithmischen Funktion. Der rote Graph ist logarithmisch, da der Graph hier bis 3000 Listenelemente ansteigt, danach abflacht und fast konstant ist. Zieht man hier eher die beiden unteren Graphen in Betracht lässt sich auch hier ein linearer Aufwand feststellen.

Linearer Aufwand $\mathcal{O}(n)$

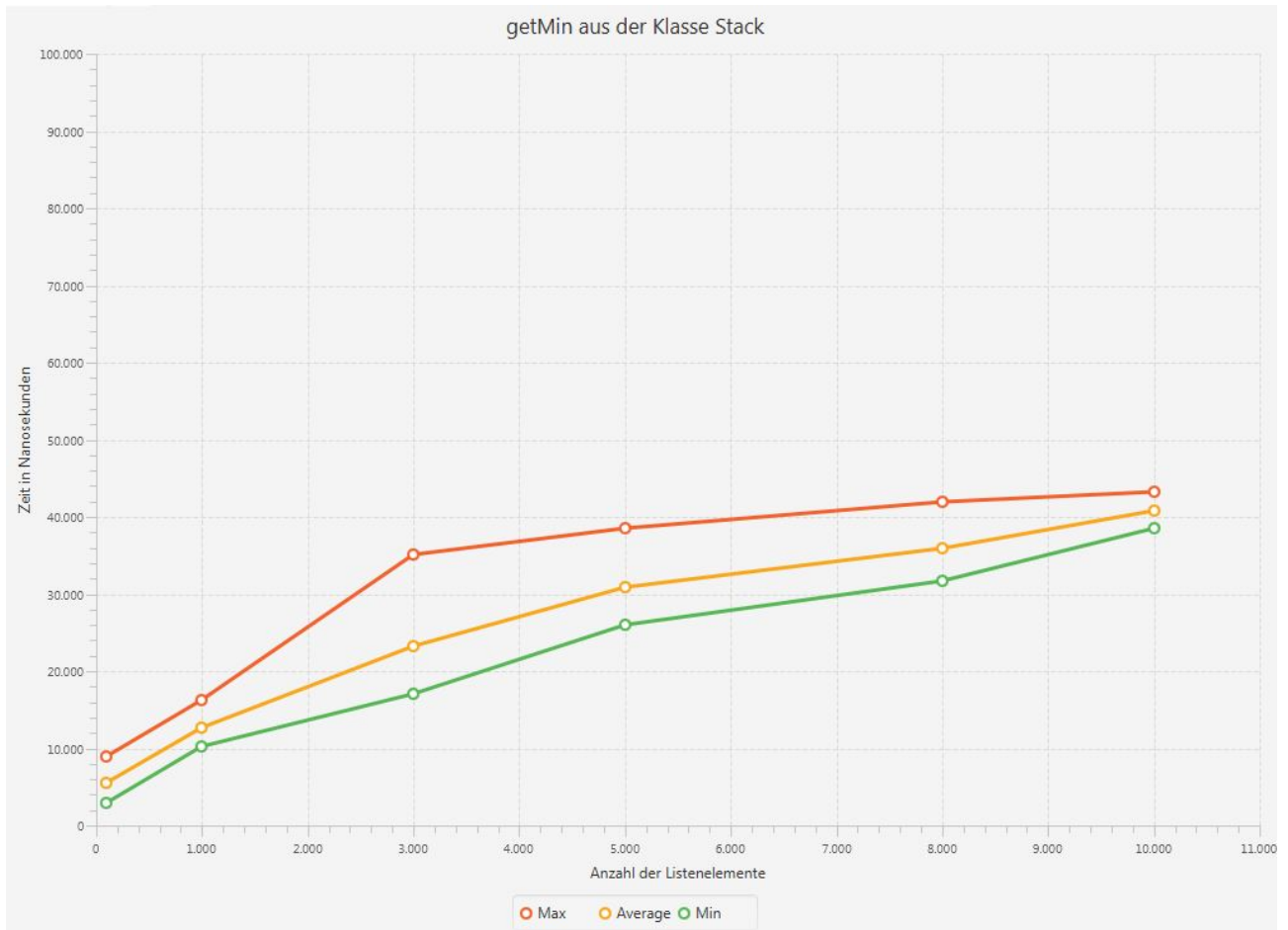


Abbildung 3.32: Stack - GetMin

Bei der Methode „insert“ ist der Durchschnittsgraph recht konstant gehalten. Beim grünen Graphen sieht man sehr deutlich, dass alle Werte exakt auf einer Linie sind und den gleichen Wert besitzen. Der Graph für die Maximalwerte steigt bis 5000 Listenelemente an und sinkt danach wieder, bis er fast den Startwert an Nanosekunden erreicht. Schlussfolgernd aus allen drei Graphen kann man sagen, dass diese Methode einen konstanten Aufwand besitzt.

Konstanter Aufwand $\mathcal{O}(1)$

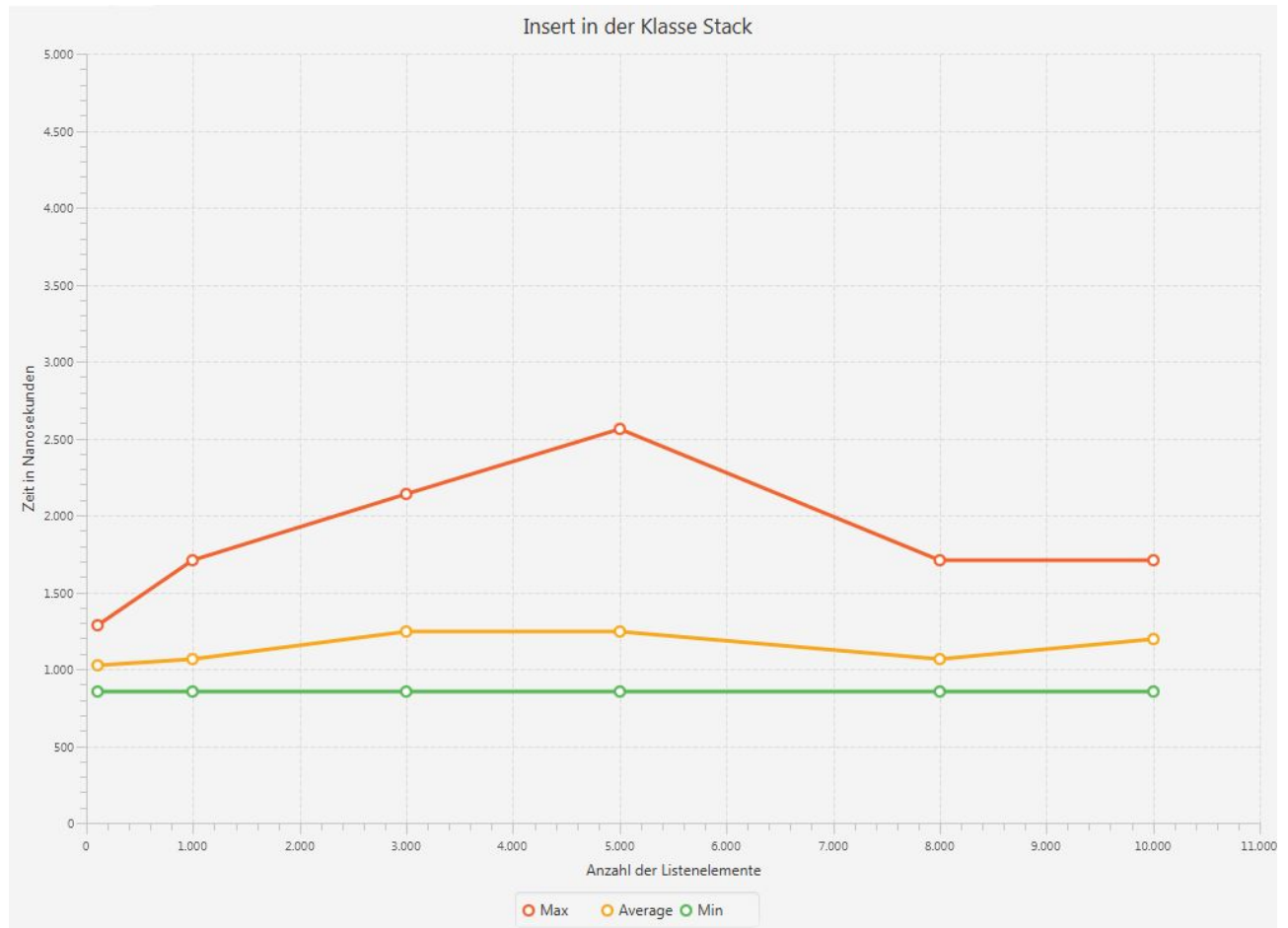


Abbildung 3.33: Stack - Insert

Die Methode „merge“ besitzt einen sehr konstanten Graphen für die Minimalwerte und die Durchschnittswerte, allerdings steigt hier der letzte Wert deutlich nach oben. Bei dem roten Graphen liegt der Minimalwert bei 1000 Listenelementen, danach steigt der Graph an, sinkt aber bei 8000 Listenelementen nochmal ab, um erneut anzusteigen.

Konstanter Aufwand $\mathcal{O}(1)$

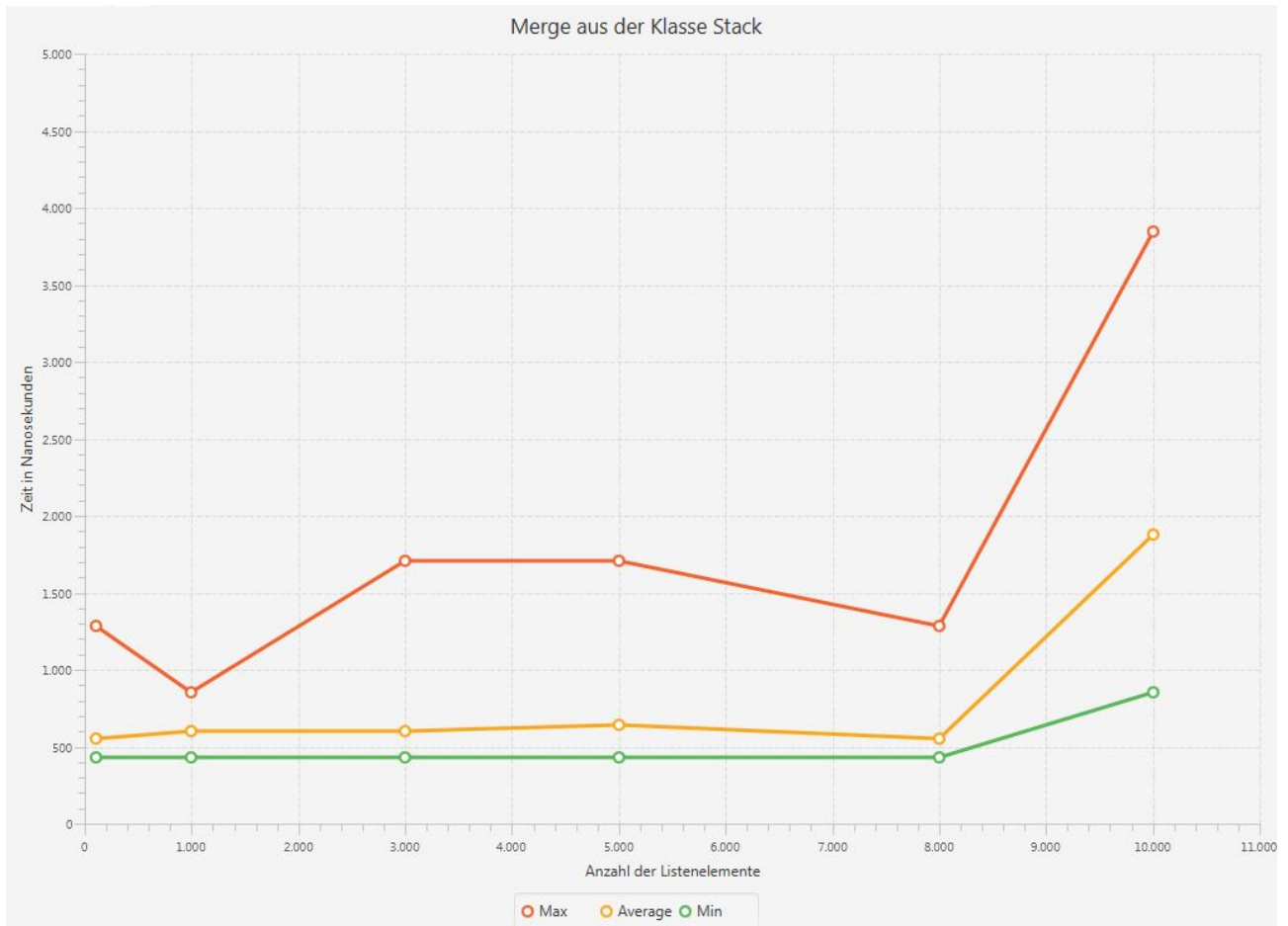


Abbildung 3.34: Stack - Merge

Bei der Methode „remove“ hat der Graph für die Minimalwerte einen logarithmischen Anklang. Auch die Durchschnittskurve ist logarithmisch, da sie bis 1000 Listenelemente ansteigt und dann stark abflacht. Der rote Graph weist noch keine logarithmische Funktion auf, da er recht konstant ansteigt. Betrachtet man mehr die 2 unteren Graphen, kann man erkennen, dass es sich um einen logarithmischen Aufwand handelt

Logarithmischer Aufwand $\mathcal{O}(\log n)$

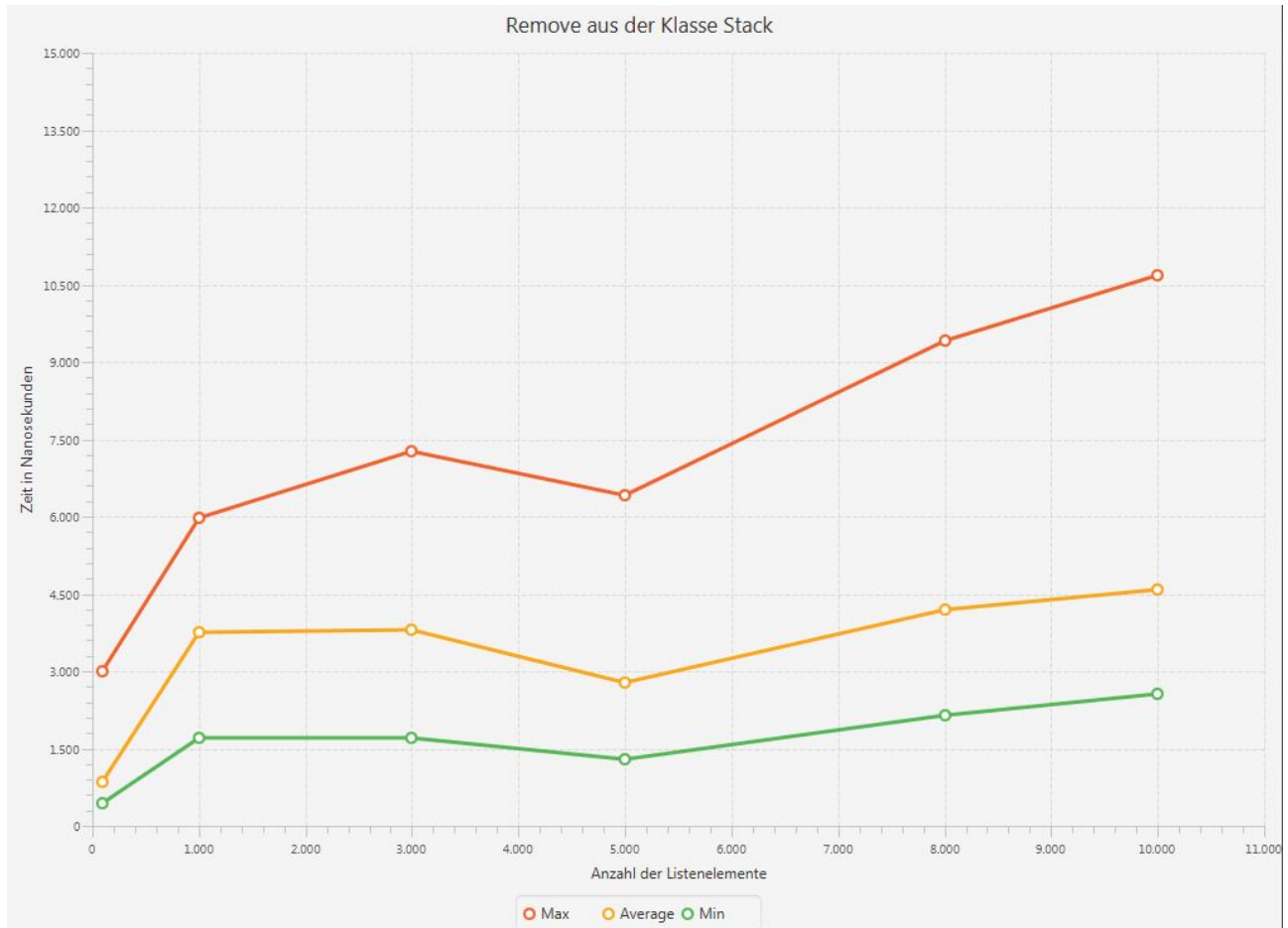


Abbildung 3.35: Stack - Remove

3.7.4 Schlussfolgerung

Insgesamt ist der Stack von ihrem Aufwand her sehr gut, da dieser meist linear ist. Die Methoden „insert“ und „merge“ sind besonders zu empfehlen, da ihr Aufwand konstant ist. Der logarithmische Aufwand der „remove“ Methode ist für wenige Listenelemente nicht so gut, sobald es aber um große Mengen geht kann man auch diese gut verwenden.

Die einzige Methode die gar nicht für den Stack geeignet ist, ist „decreaseKey“, da es nicht möglich war diese zu Implementieren.

3.8 Deque

3.8.1 Einleitung

Deque gehört zum java.util Paket. Es implementiert das Queue-Interface. Elemente lassen sich im Gegensatz zu Queue an beiden Enden einfügen und entfernen.

3.8.2 Methodenbeschreibung

- `add()` : - Fügt ein spezifisches Element am Ende der Deque hinzu.
- `addFirst()` : - Fügt ein spezifisches Element am Anfang der Deque hinzu, falls die Kapazität ausreicht.
- `addLast()` : - Fügt ein spezifisches Element am Ende der Deque hinzu, falls die Kapazität ausreicht.
- `contains()` : - Gibt `true` zurück wenn die Deque das spezifische Element beinhaltet.
- `isEmpty()` : - Gibt `true` zurück wenn das Set keine Elemente enthält.
- `iterator()` : - Gibt einen Iterator über die Elemente im Deque zurück.
- `getFirst()` : - Gibt das erste Element des Deques zurück, ohne es zu löschen.
- `getLast()` : - Gibt das letzte Element des Deques zurück, ohne es zu löschen.
- `size()` : - Gibt die Anzahl von Elementen in der Deque zurück.

3.8.3 Effizienzanalyse

Bei der Methode `extractMin()` sehen wir bei allen drei Graphen einen ähnlichen, positiven Anstieg. Durchschnittlich ergibt sich ein linearer Aufwand $\mathcal{O}(n)$.

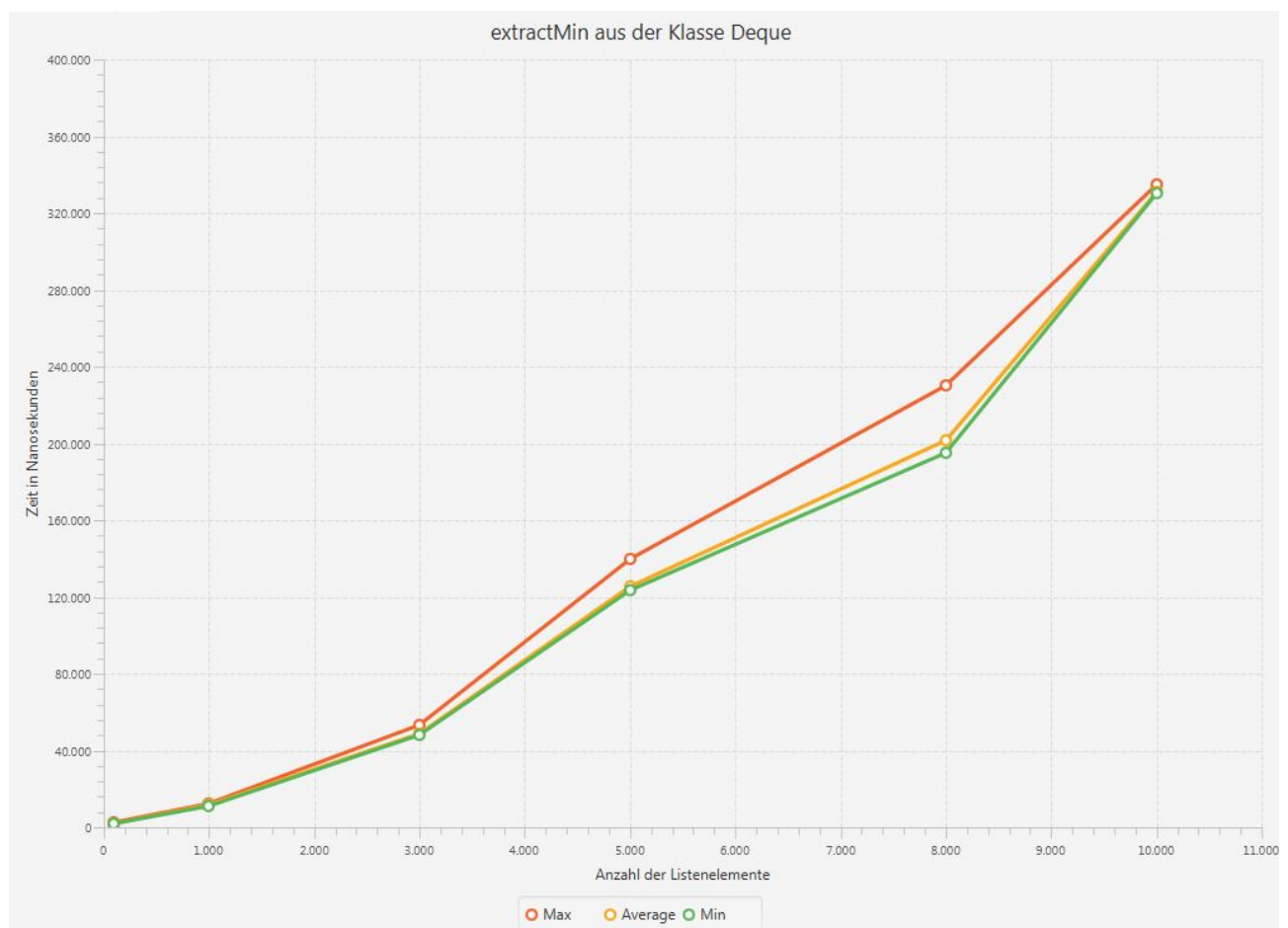


Abbildung 3.36: Deque - `extractMin`

Bei der Methode `getMin()` sehen wir drei Graphen, die sich alle sehr ähnlich verhalten. Minimaler und Mittlerer Aufwand sind fast identisch. Es folgt ein Aufwand von $\mathcal{O}(n)$.

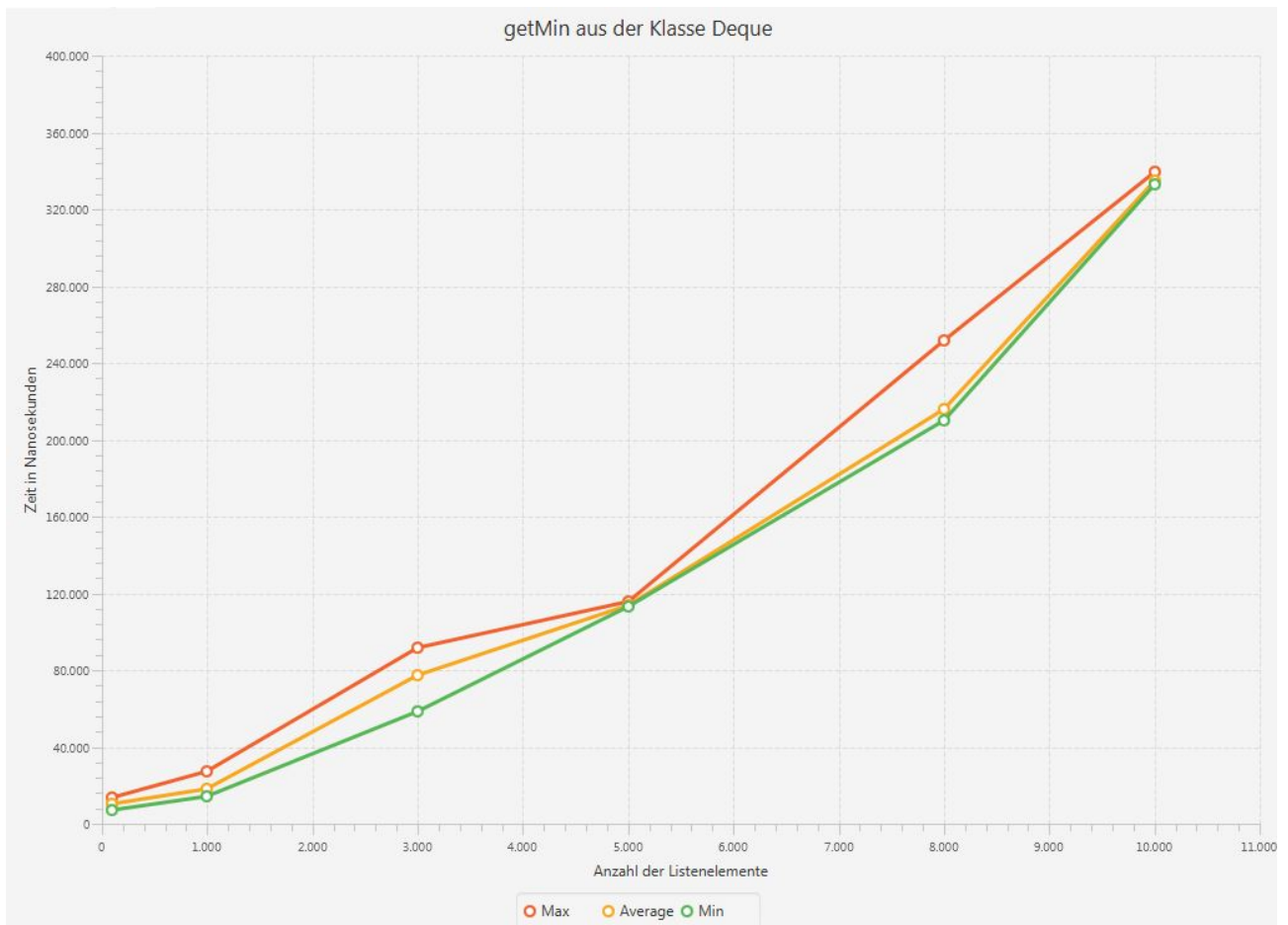


Abbildung 3.37: Deque - getMin

Bei der Methode `Insert()` kann man erkennen das alle Graphen mit nur wenigen Schwankungen linear verlaufen. Der Aufwand ist sehr offensichtlich $\mathcal{O}(1)$.

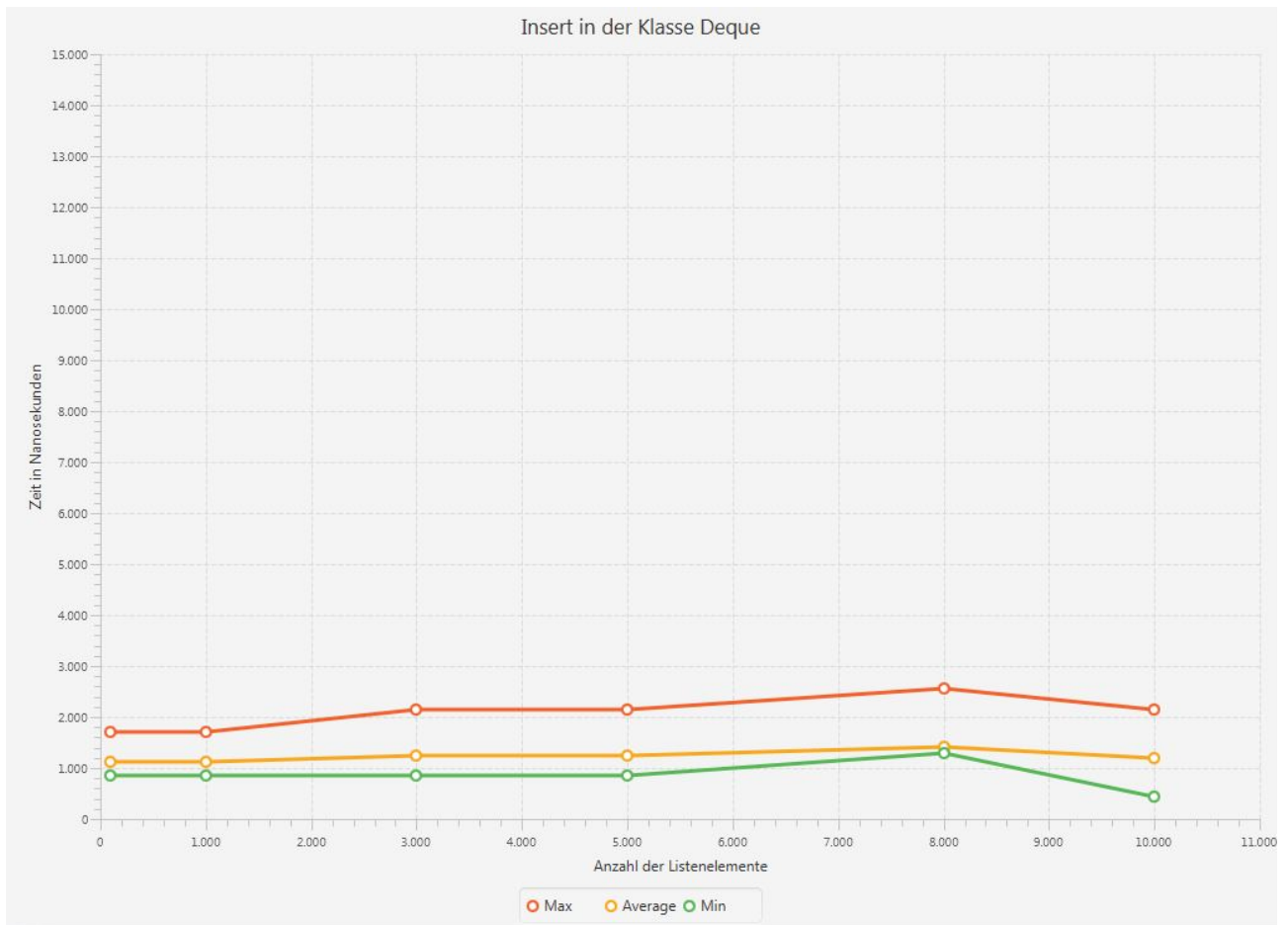


Abbildung 3.38: Deque - Insert

Bei der Methode `Merge()` erkennt man bis zu 8000 Listen Elementen einen sinkenden Aufwand. Erst danach gibt es einen starken Anstieg. Bis zu dieser Grenze von 8000 Elementen lässt sich ein konstanter Aufwand von $\mathcal{O}(1)$ feststellen.

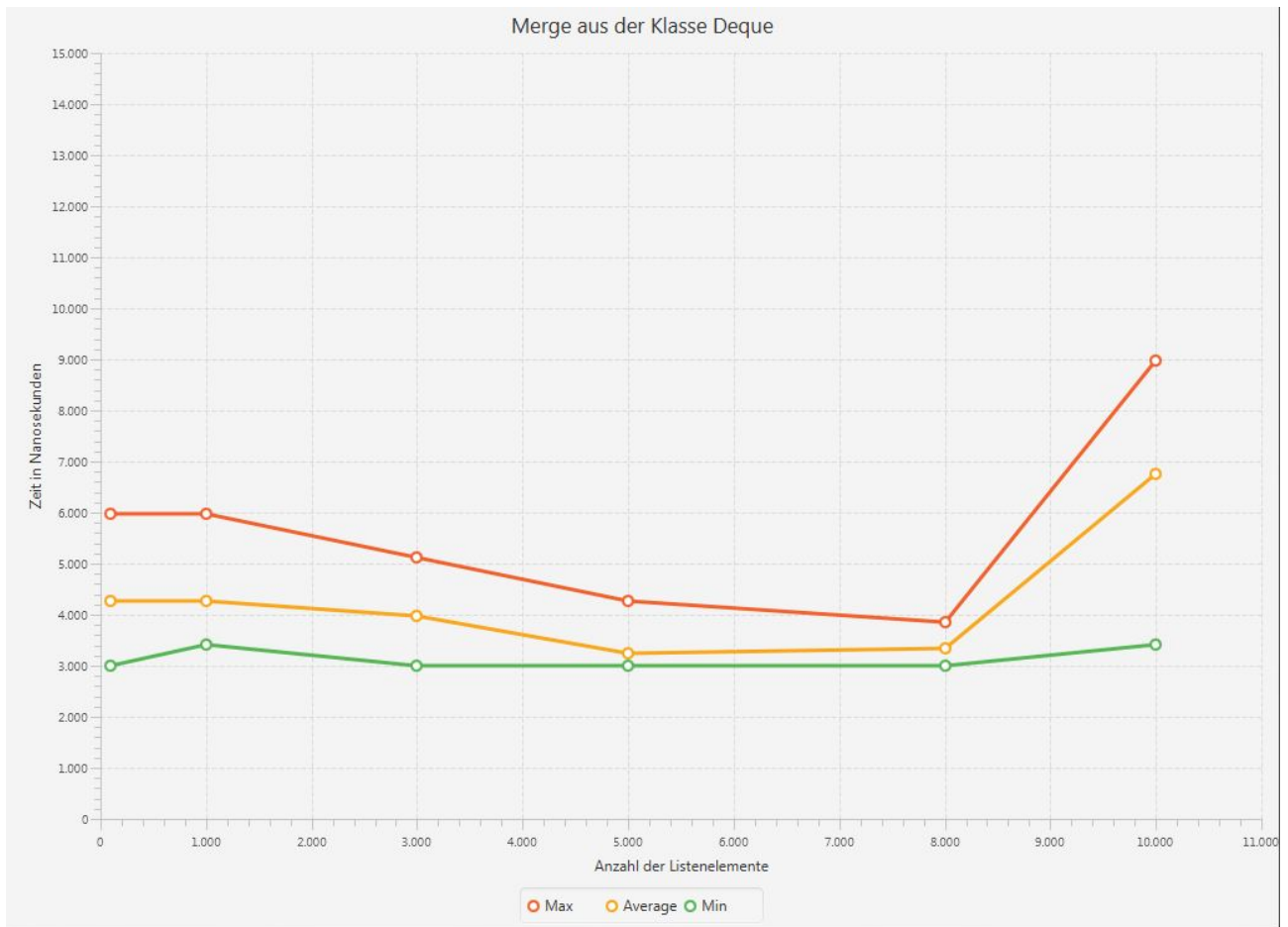


Abbildung 3.39: Deque - Merge

Bei der Methode `Remove()` erkennt man einen klaren Anstieg mit einigen stärkeren Schwankungen im Bereich von 3000 bis 5000 Listenelementen. Dennoch lässt sich ein linearer Aufwand daraus schließen. $\mathcal{O}(n)$

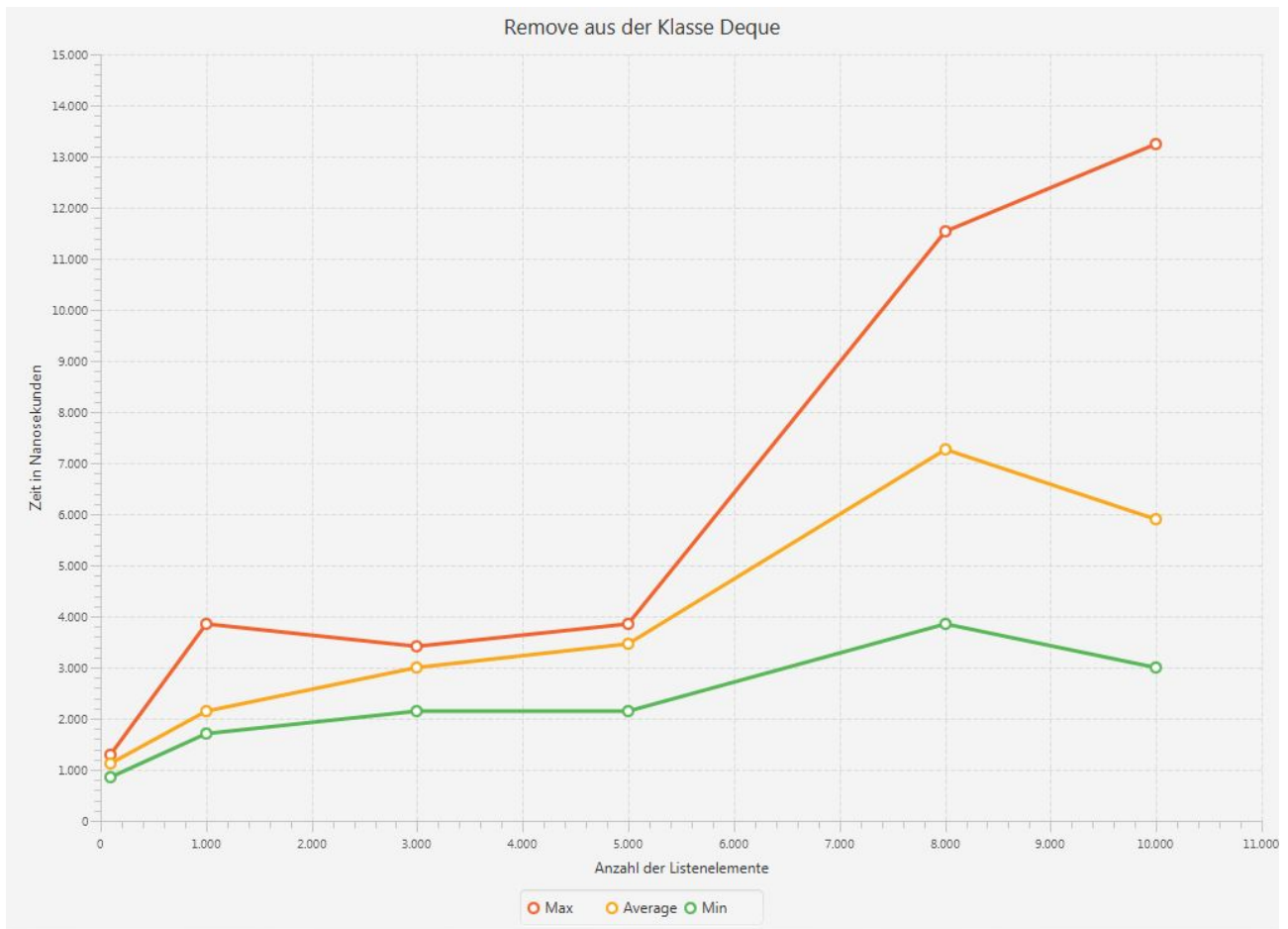


Abbildung 3.40: Deque - Remove

3.8.4 Schlussfolgerung

Die Deque bietet stets einen linearen und in manchen Fällen sogar konstanten Aufwand. Sie eignet sich somit ideal für alle Datenverarbeitungen bei denen Daten in einer doppelten Warteschlange repräsentiert werden sollen.

Kapitel 4

Schlussanalyse

Um die Klassenübersicht abzurunden wird nun noch einmal eine Zusammenfassung angebracht. In dieser werden die einzelnen Klassen auch untereinander verglichen.

Generell ist zu sagen, dass die meisten Klassen eine sehr gute Effizienz vorweisen können. Mit sehr gut ist hier ein linearer oder sogar konstanter Aufwand gemeint.

Die Klasse `TreeSet` bietet sich für das Einfügen, Löschen von Werten sowie die Verbindung mehrerer Strukturen hervorragend an, da sie in diesen Punkten einen konstanten Aufwand vorweist. Allerdings gibt es hier Einschränkungen bei der Verwendung gleicher Zahlenwerte.

Bei `HashSet` ist dies hingegen kein Problem. Diese Klasse ist von allen die flexibelste. Sie ist besonders bei großen Datenmengen zu empfehlen, da sie beispielsweise bessere Werte bei `remove` als die Klasse `TreeSet` mit sich bringt.

Die Klasse `Array-List` kann auch größere Mengen an Zahlen relativ gut verarbeiten, weißt aber mit `getMin` und `extractMin` ihre Schwächen auf. Das bedeutet in allen Anwendungen, bei denen viele Werte erst gesucht werden müssen ist von dieser Klasse abzuraten.

Wenn man die Größe der Liste flexibel benötigt, bietet sich die Klasse `Vector` an. Allerdings muss man bei dem Entfernen von Zahlen mit einer höheren Rechenzeit rechnen.

Die Klasse `Queue` eignet sich für das Aufnehmen von Messwerten besonders, da bei dieser Klasse die Stärke bei dem Einfügen von vielen Elementen liegt. Das Löschen der Elemente dauert allerdings erheblich.

Einen konstanten Zeitaufwand beim Einfügen von Elementen und Zusammenfügen von 2 Listen bietet die Klasse `Stack`. Wie man in den gegebenen Diagramm erkennen kann, ist die Rechenzeit bei `insert` vergleichbar mit der von der Klasse `Queue`.

Als flexibel stellt sich auch die Klasse `Deque` heraus. Sie besitzt in allen Funktionen gute Laufzeiten. Allerdings gibt es hier starke Abweichungen zwischen dem Best- und dem Worst-Case. So sollte man evtl. differenzieren und evtl. eine Vorsortierung der Werte in Betracht ziehen.

Abschließend kann man sagen, dass fast alle Klassen akzeptable Laufzeiten erzeugen. Besonders flexibel stellt sich aber hier die Klasse `HashSet` da. Dies hat für den Programmierer die Konsequenz, dass man in unklaren Situationen diese Klasse für die ersten Tests verwenden sollte.

Kapitel 5

Fazit

Zusammenfassend ergibt sich eine Hilfestellung für den Programmierer. Dabei ist allerdings zu beachten, dass der Programmierer eigene Versuche starten muss. Es ist nicht davon auszugehen, dass man den hier dargestellten Sachverhalt auf andere Zahlentypen ableiten kann.

Die Testreihen stellen trotz einiger Ausschläge ein anschauliches Bild der Situation dar. So kann sagen, dass zu jeder Klasse genug Entscheidungsmaterial vorgetragen wurde. Die Zwischenzusammenfassungen und Teilbetrachtungen der einzelnen Klassen sollen zusätzlich die Entscheidung des Lesers erleichtern.

Wie man in der Arbeit erkennen kann, gibt es nicht eine schlechte Klasse und eine gute Klasse. Alle Klassen sind für unterschiedliche Anwendungsfälle ausgelegt. Dies bedeutet aber nicht, dass man nur für eine konkrete Anwendung die Klassen verwenden sollte. So kann man bei manchen Methoden mit einer schlechteren Laufzeit leben. Es ist zwar möglich zwischen verschiedenen Datentypen zu wechseln. Allerdings kostet jede Typumwandlung ebenfalls Rechenzeit. Daher ist eine Neustrukturierung des Programms vielen Typumwandlungen vorzuziehen. Allerdings ist es auch so, dass es in der Praxis teilweise ohne Umwandlung nicht funktioniert und dass diese essentiell wichtig sind.

Für die Studenten des Matrikels IIB13 sowie K-IIB13 stellte dieser Beleg die erste große Gruppenarbeit da. Somit konnten auch fächerübergreifende Fähigkeiten, wie Teamfähigkeit oder Kommunikation untereinander verbessert werden. Zusätzlich wurde für diese Arbeit das erste Mal Programming-Wiki zur Organisation verwendet.

Autorenverzeichnis

Tabelle 5.1: Autorenliste

Thema	Studententeam
TreeSet	Fabian Großmann, Michael Švancar
HashSet	Pascal Otto, Mohamed Amine El Jabri
ArrayList	Gerardo Balderas, Mohamed Amine El Jabri
LinkedList	Fabian Großmann, Franz Scheffler
Vector	Gerardo Balderas, Franz Scheffler
Queue	Sarah Bertulat, Jens Weber
Stack	Sarah Bertulat, Jens Weber
Deque	Pascal Otto, Michael Švancar
Diagramme, empirische Zeitmessung	Florian Gnepper
Organisation, Programming-Wiki, Zusammenfassungen	Stephan Strehler

Die einzelnen Beiträge und Kapitel können auch unter [http : //programmingwiki.de/Organisation_Java-Entscheidungsheft](http://programmingwiki.de/Organisation_Java-Entscheidungsheft) nachgelesen werden.