

1 LinkedList

1.1 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`).

1.2 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

1.3 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.

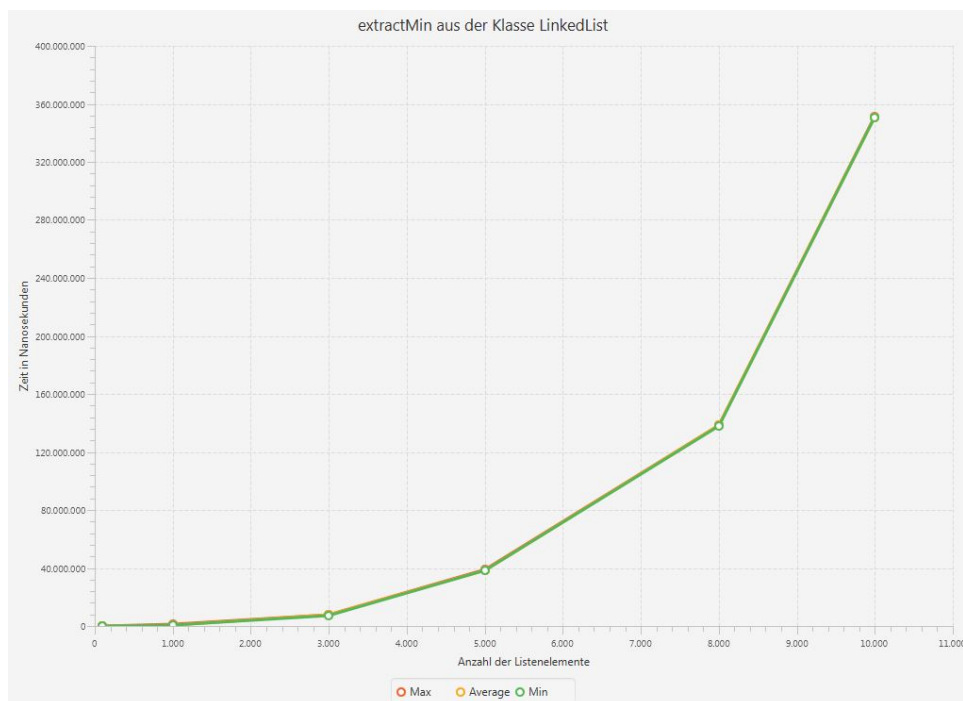


Abbildung 1: ExtractMin

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.

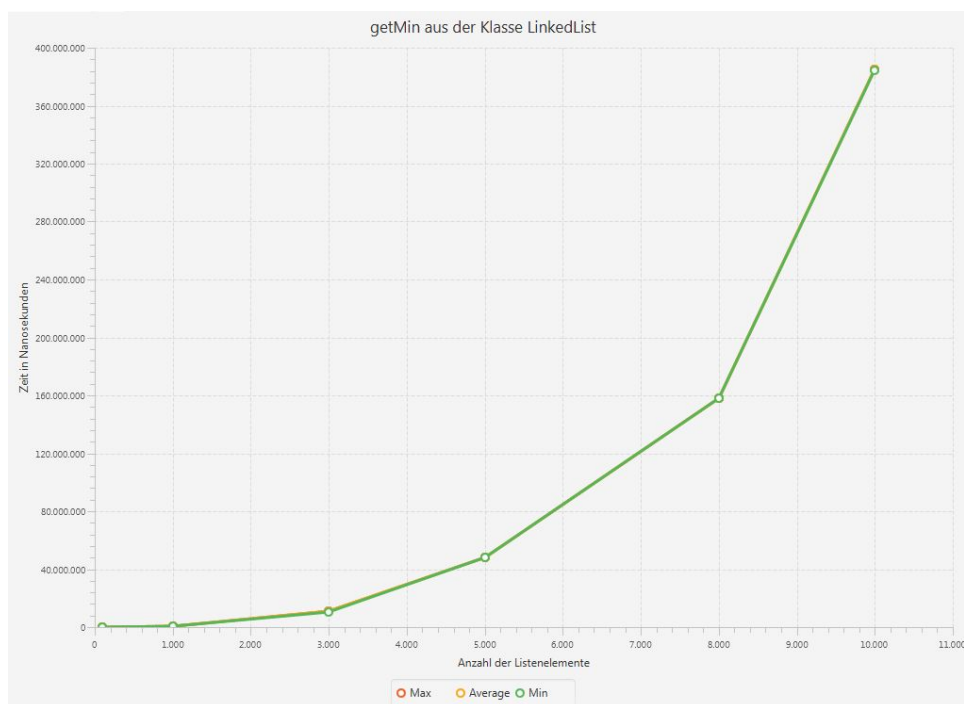


Abbildung 2: GetMin

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 $\mathcal{O}(k)$.

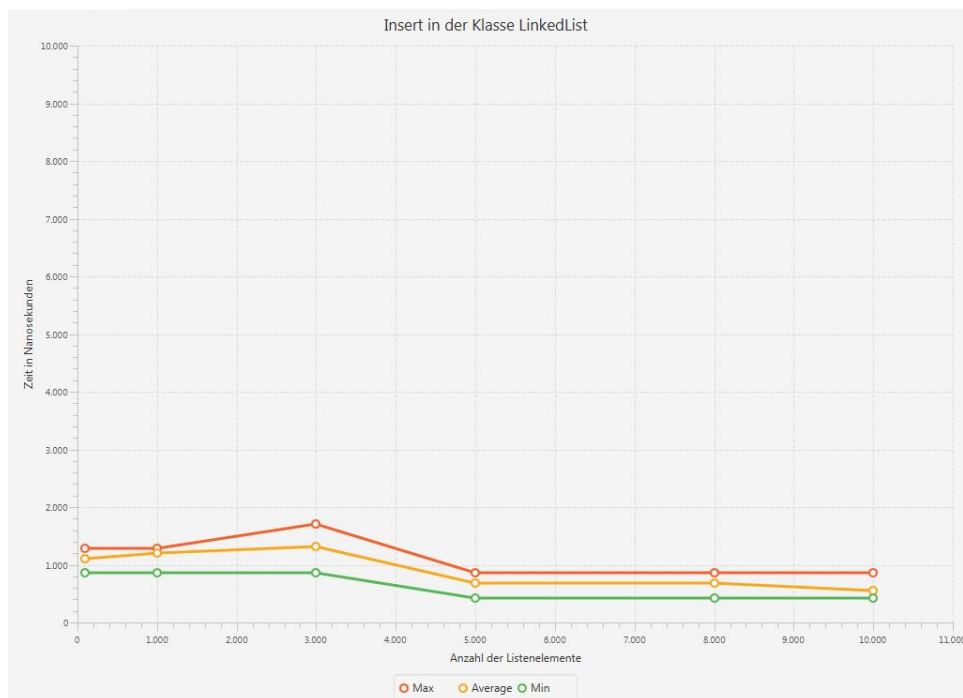


Abbildung 3: Insert

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. Danach 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 erhaltenen Messwert von 5000 Listenelementen um einen fehlerhaften Messwert handelt. Es handelt sich bei der Methode merge für die Klasse LinkedList um einen konstanten Aufwand $\mathcal{O}(k)$.

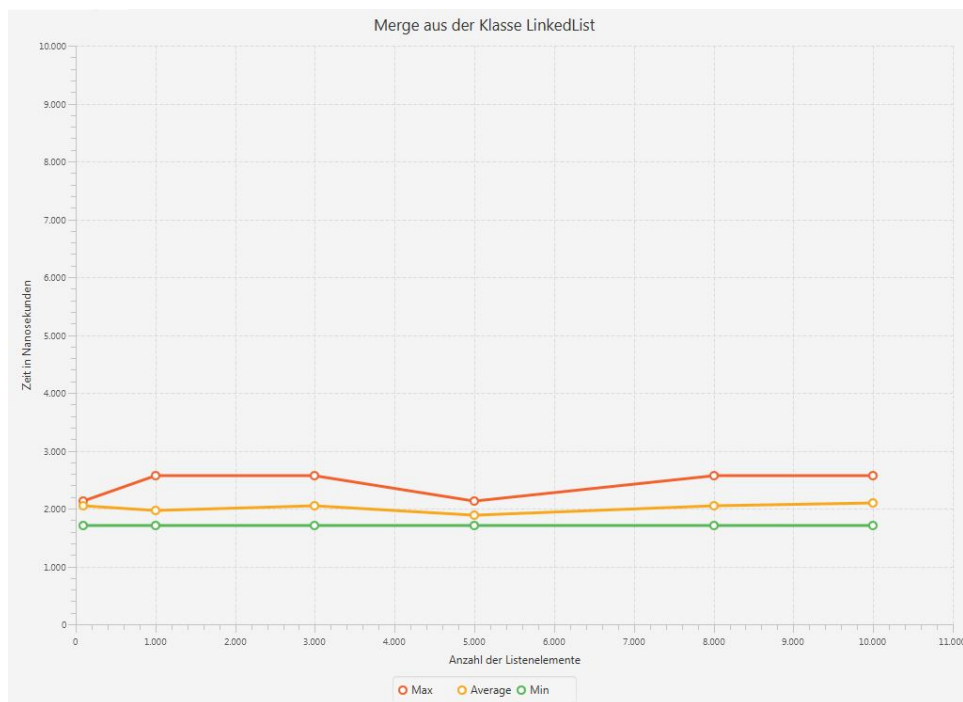


Abbildung 4: Merge

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 $\mathcal{O}(k)$.

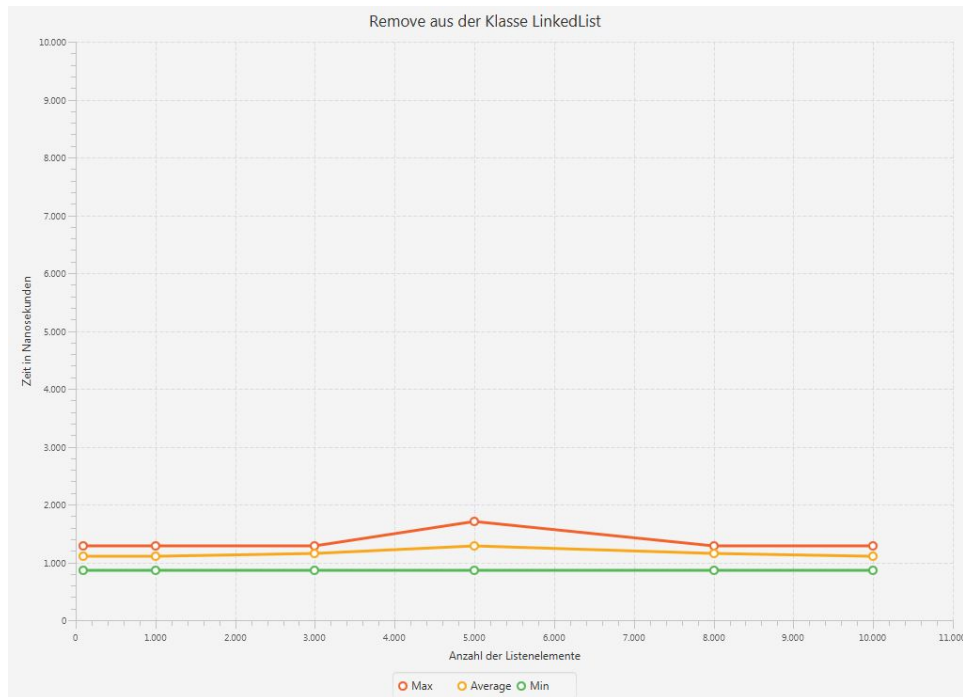


Abbildung 5: Remove

1.4 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.