

1 TreeSet

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()`).

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 angegebenen 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) Element 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

1.3 Effizienzanalyse

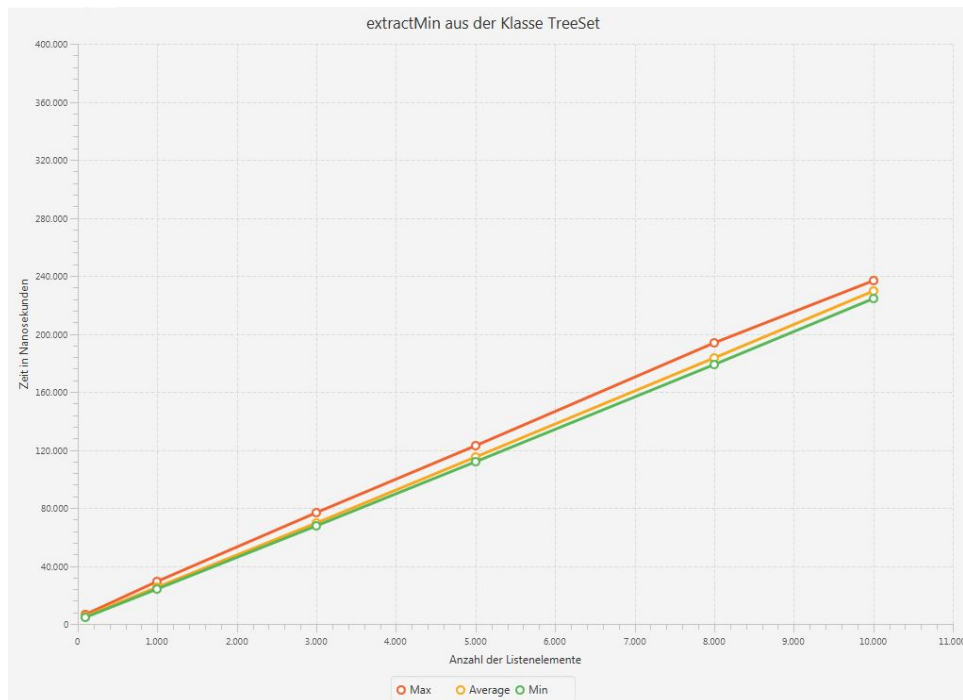


Abbildung 1: ExtractMin

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 $\mathcal{O}(n)$ geht.

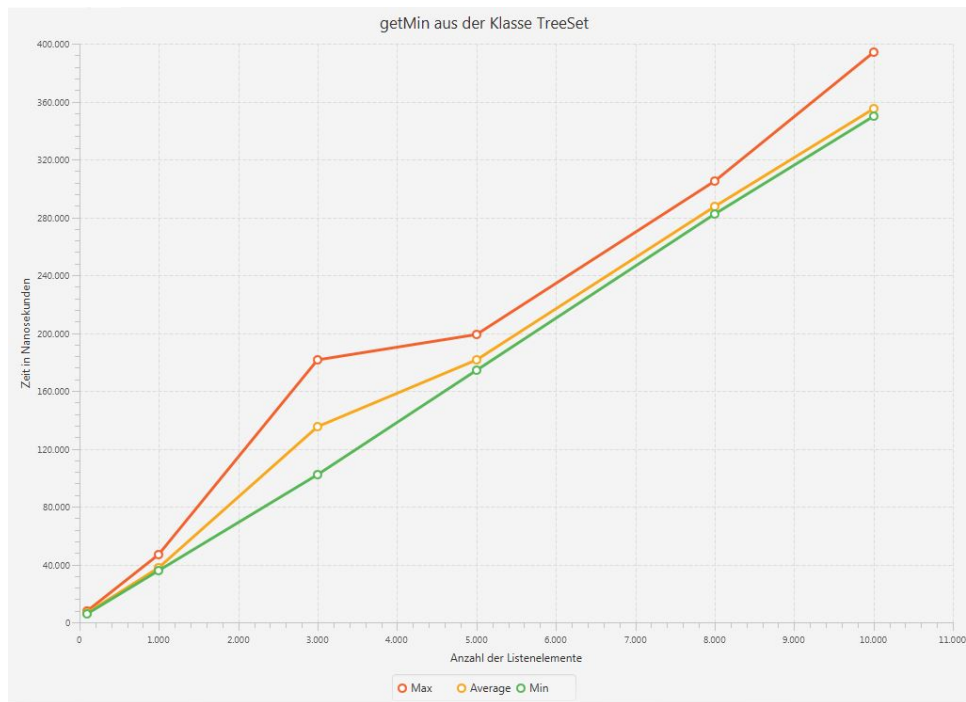


Abbildung 2: GetMin

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 $\mathcal{O}(n)$.

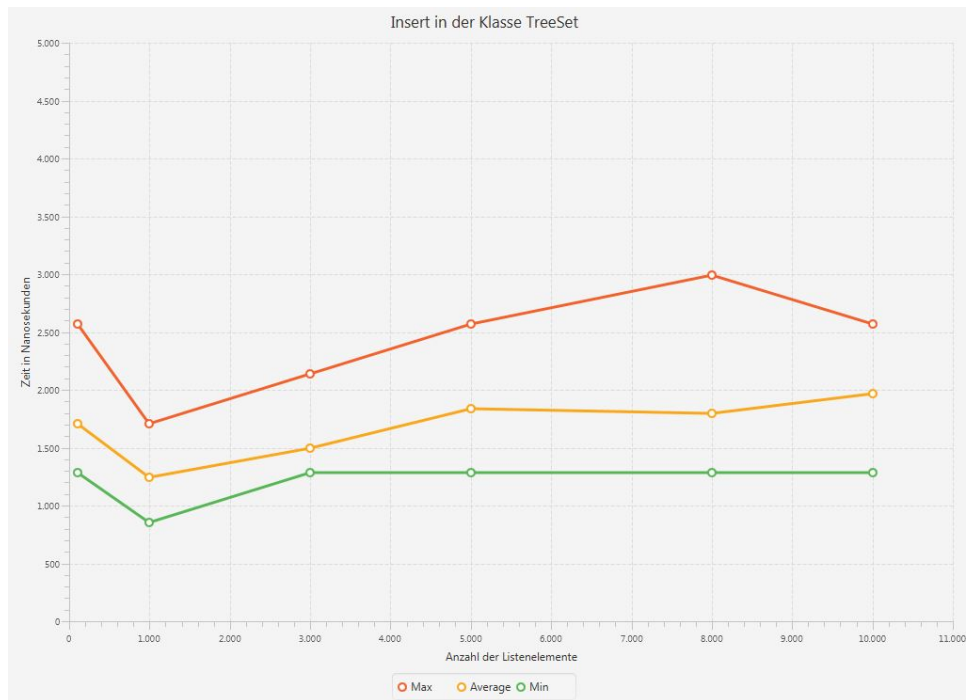


Abbildung 3: Insert

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.

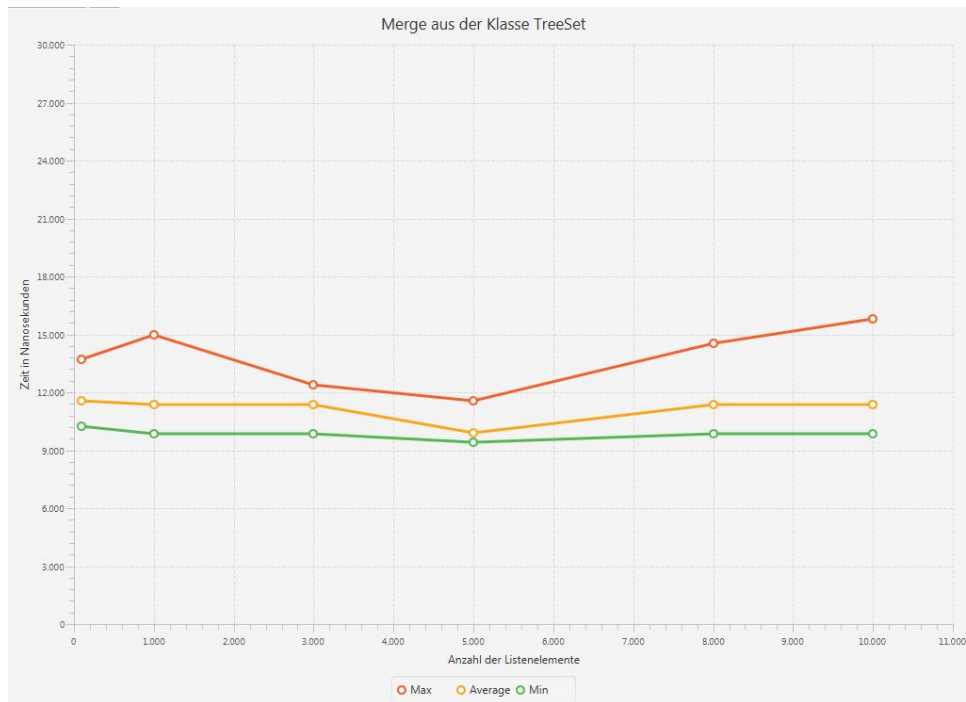


Abbildung 4: Merge

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.

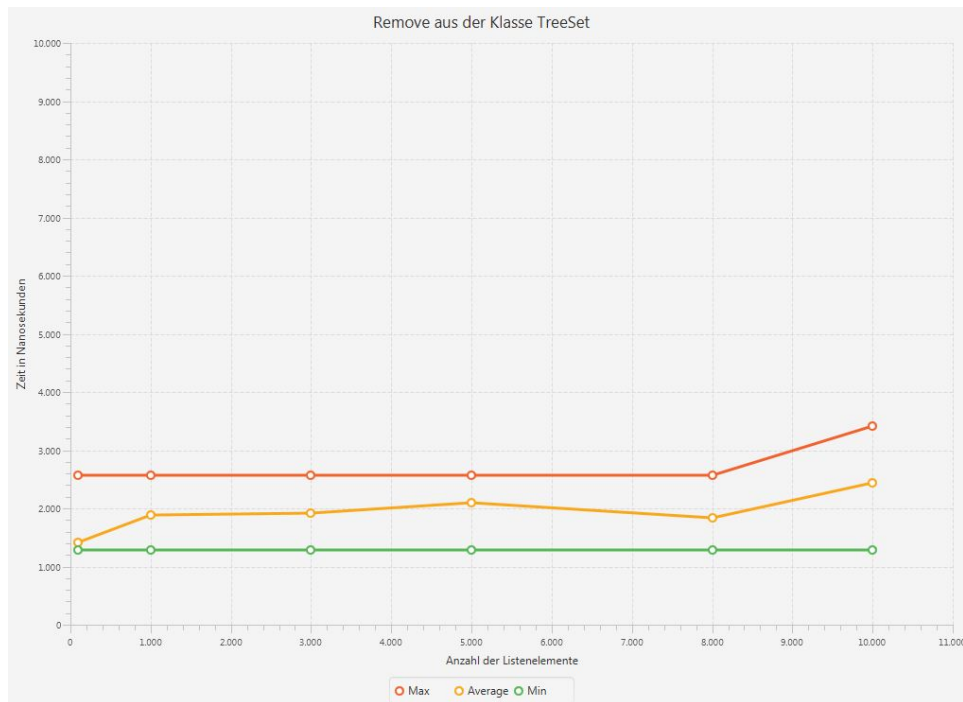


Abbildung 5: Remove

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.

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.