

1 Vector

1.1 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.

1.2 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.

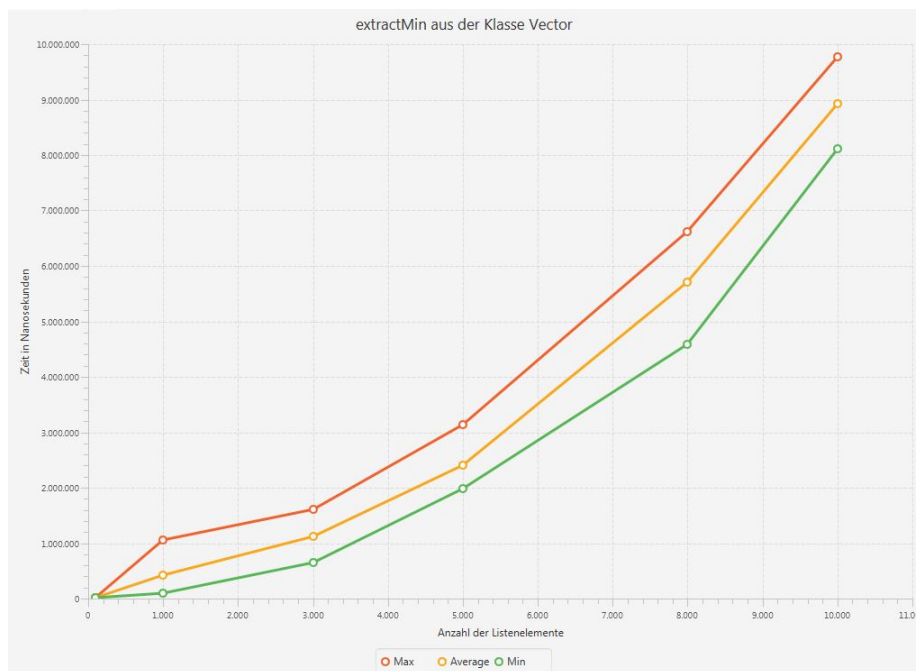
1.3 Effizienzanalyse

1.3.1 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.



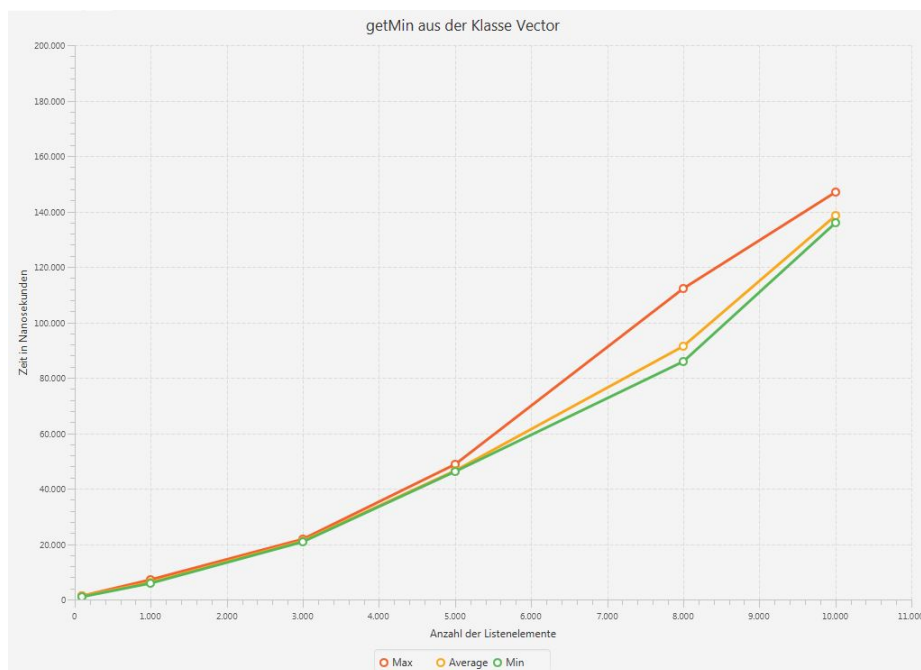
1.3.2 GetMin

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.



1.3.3 Insert

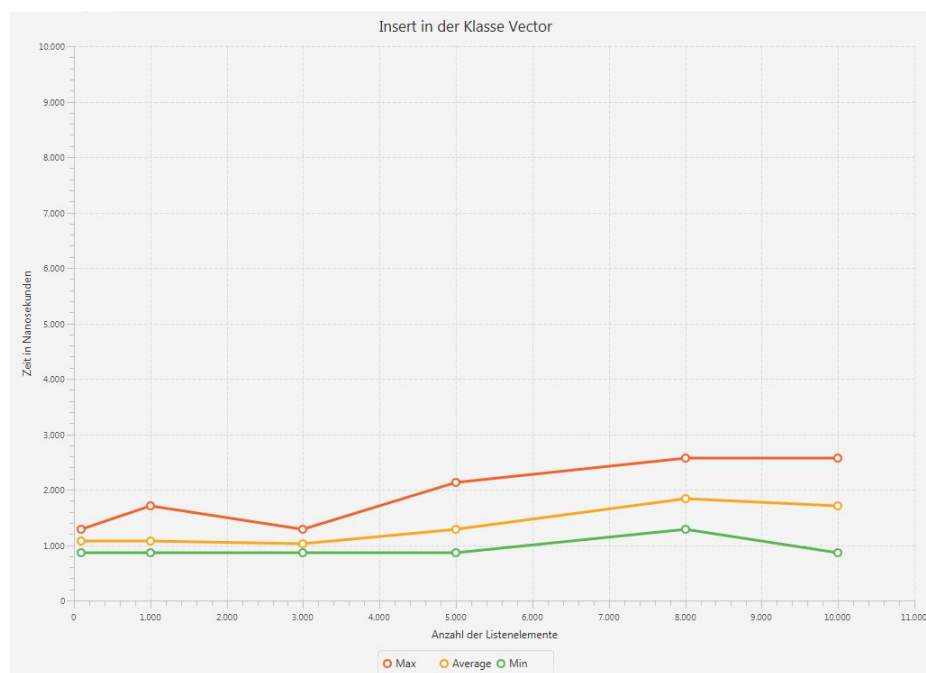
Bei der Methode insert ist der Durchschnittsgraph recht konstant gehalten. Ab einer Anzahl von 3000 Elementen kommt es jedoch zu einem leichten Anstieg der benötigten Zeit auf einen Spitzenwert, der bei 8000 Elementen erreicht wird.

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.



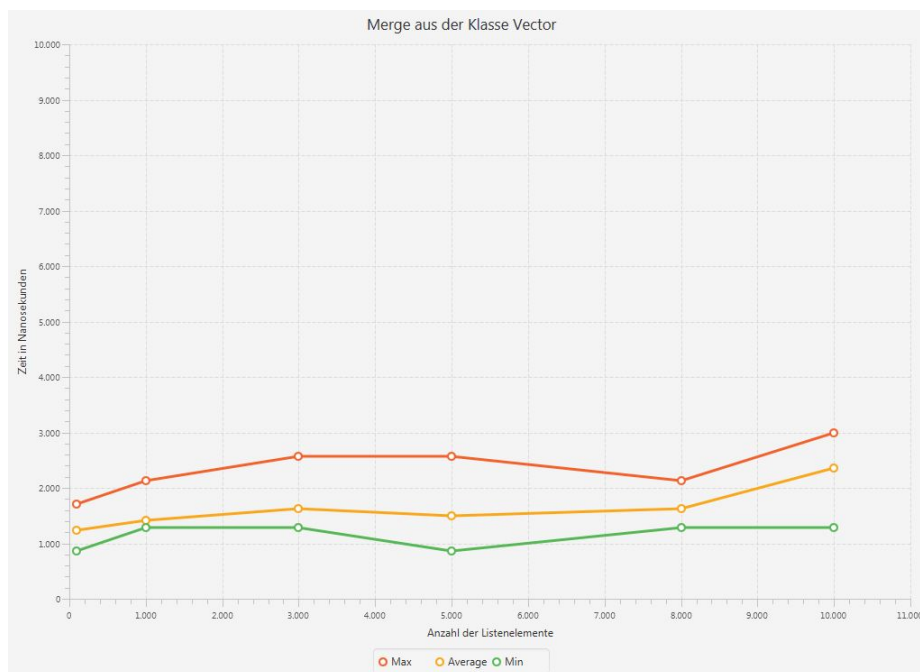
1.3.4 Merge

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.

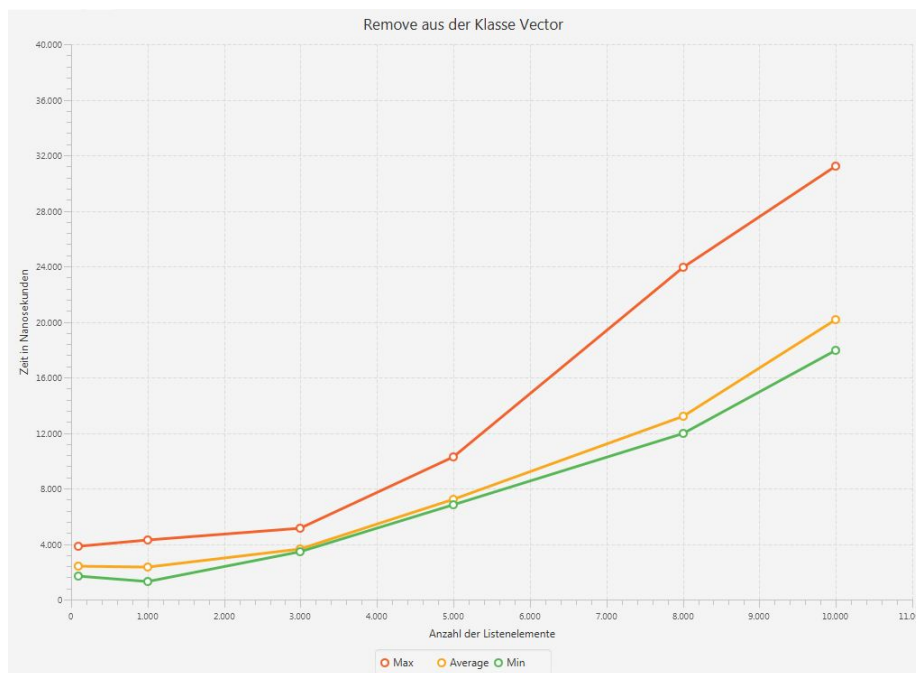


1.3.5 Remove

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.



1.4 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 geringeren 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.