

# Pascalsches Dreieck

Finn Jannsen, Philipp Schwarz

16. April 2019

## Inhaltsverzeichnis

<b>1</b>	<b>Einführung</b>	<b>1</b>
<b>2</b>	<b>Implementation</b>	<b>1</b>
2.1	Set als Array . . . . .	1
2.2	Set als Array von Containern . . . . .	2
2.3	Set als einfach verkettete Liste von Containern . . . . .	2
<b>3</b>	<b>Verifizieren und Testen</b>	<b>3</b>
3.1	Verifizieren der Funktionalität . . . . .	3
3.2	Aufwandsanalyse . . . . .	3

## 1 Einführung

Diese Dokumentation beschreibt drei Implementations-Varianten des Abstrakten Datentyps Set. Dieser Datentyp soll eine Menge darstellen. Als Vorgabe zur Art der Implementation wurde 1. ein Array von Elementen, 2. ein Array von Containern und 3. eine verkettete Liste von Containern angeführt. In Abschnitt 2 wird darauf eingegangen, wie die verschiedenen Varianten realisiert wurden. Anschließend wird in Abschnitt 3.1 geprüft, ob die vorgegebenen Operationen funktionieren und in Abschnitt 3.2 die Performance der Varianten verglichen.

## 2 Implementation

Die drei Varianten wurden in Java als Klassen implementiert. Da alle den gleichen Funktionssatz brauchen implementieren sie alle das gleiche Interface namens SetInterface.

Abbildung ?? stellt die verwendeten Klassen und Interfaces dar. Dort ist auch zu sehen, welche Methoden das SetInterface vorgibt. Für Interkompatibilität bezüglich der Vereinigung von Mengen wurde die unify-Methode statisch gemacht. Die jeweiligen Implementationen sind bezüglich unify(s1, s2) interkompatibel, da sie jegliche Sets, die das SetInterface implementieren entgegennehmen und dieses zu einem neuen Set (Typ abh. von aufgerufener Methode) vereinigen.

Auch wurden Maßnahmen gegen Verstöße, vor allem das verwenden von Pos-Objekten aus uneigenen Sets, getroffen.

Mittels Generics wurden Elem und Pos außerdem kompatibel zu allen drei Varianten modelliert.

### 2.1 Set als Array

Das Set wurde zunächst als Array implementiert. Hierfür wurde ein Array von der Elem-Klasse erstellt, sowie von der Pos-Klasse mit Zeigern auf die Integer-Indexe von Elem[].

Werden Plätze für Elemente im Set benötigt, so erfolgt eine dynamische Erstellung von Pos-Objekten, die die belegten und unbelegten Plätze im Set verwalten.

Beim Aufrufen der delete(Pos)-Methode werden jegliche Referenzen des Elements entfernt und Pos wird über isValid=FALSE unbelegt und später wieder beim Hinzufügen von Elementen benutzt.

Die Methode find(Key) iteriert die Positionen von der höchsten an und sucht dabei nach dem zugehörigen Element. Wird es nicht gefunden, so wird das invalide Stopper-Element als Position zurückgegeben.

## 2.2 Set als Array von Containern

Für die Implementation einer Menge mithilfe eines Arrays von Container-Klassen wurde die Container-Klasse Pos mit einem nextIndex und previousIndex versehen, welche auf den Integer-Index der vorherigen und nächsten Pos-Objekte im Container-Array zeigen. Hiermit wird eine stets korrekte Reihenfolge der Elemente im Set ermöglicht und sie ist realisiert durch folgende Mechaniken in den Methoden:

Wird ein neuer Pos-Container benötigt, dann wird er hinter den letzten Pos-Container durch setzen der Indexe angereiht.

In Abbildung 2.1 sieht man, wie beim Löschen eines Elements die laut nextIndex und previousIndex benachbarten Container durch ändern derer Indexe verknüpft werden. Der nun leere Container wird hinter den letzten Container in der Reihe eingereiht.

```
// Link neighbours
int pre = positions[i].getPreviousIndex();
int next = positions[i].getNextIndex();
positions[pre].setNextIndex(next);
positions[next].setPreviousIndex(pre);

// Append at ending
positions[positions[0].getPreviousIndex()].setNextIndex(i);
positions[i].setNextIndex(0);
positions[0].setPreviousIndex(i);
elemSize--;
```

Abbildung 2.1: Ausschnitt Lösch-Mechanik Container-Array

Die Suche nach Elementen fängt von dem letzten Container in der Reihe an und läuft über previousIndex maximal bis zur Position von einem Stopper-Element, wo die Suche spätestens endet. Dies ist in Abbildung 2.2 zu sehen.

Die delete(Pos)-Methode iteriert auf die gleiche Weise.

## 2.3 Set als einfach verkettete Liste von Containern

Für die Implementation einer Menge mittels Container-Klassen wurde eine neue Klasse 'Node' erstellt. Diese Klasse hat einen Zeiger auf eine nächste Node, aber auch einen Zeiger auf ein Listenelement. Zusätzlich gibt es noch ein Pos-Array, dessen Pos auf Nodes zeigen.

Bei der Methode add(), wird zuerst mittels der Methode find(Key) überprüft, ob sich das Element schon in der Liste befindet und wenn nicht, welches Pos im Array als erstes frei ist, bzw. ob ein neues erstellt werden muss, welches dann auf die Node mit dem Element zeigt. Anschließend wird die Referenz des Pos mit dem neuen Element zurückgegeben.

Bei der Methode delete() wird die Pos von dem Element gesucht und anhand dieser, wird die Referenz der vorherigen Node auf die nachfolgende gesetzt. Bei der Pos wird isValid=false gesetzt.

```

// Index 0 is reserved for stopper, but previousIndex always pointing to last
Pos-Container in row
int i=0;
while(true)
counter++;
int pre = positions[i].getPreviousIndex();
Elem tmp = (Elem)positions[pre].getPointer();
if (tmp != null)
if (tmp.key == key)
return positions[pre];

else
//error
return null;

i = pre;

```

Abbildung 2.2: Ausschnitt Such-Mechanik Container-Array

Bei der Methode `find()` werden alle Nodes durchlaufen bis das Element gefunden wurde und anschließend die dazugehörige Pos gesucht und zurückgegeben.

Bei der Methode `retrieve()` wird das Pos-Array so lange durchsucht, bis wir die Pos die wir suchen gefunden haben und dann wird ihr Element zurückgegeben.

## 3 Verifizieren und Testen

### 3.1 Verifizieren der Funktionalität

Da die drei Varianten sich nur intern im Aufbau unterscheiden und dies nach außen hin gekapselt ist, verifizieren wir ihre Funktion anhand einer Menge an gleichen Tests. Die in Tabelle 1 enthaltenen Tests sollen eine ausreichende Verifizierung der Operationen auf den Sets ermöglichen.

Die in der Tabelle 1 aufgeführten Tests wurden mit verschiedenen Parametern und Quantitäten in verschiedenen Sequenzen in JUnit4 getestet. Alle drei Implementations-Varianten erfüllen die oben genannten Tests mit pre- und post-condition.

### 3.2 Aufwandsanalyse

Anhand der Tests und deren Ergebnisse, die in Abbildung 3.1, 3.2 und 3.3 visualisiert sind, ist sichtbar, dass `delete` mittels Pos immer effizienter ist als mittels Key. Dies ist dadurch erklärt, dass `delete` mittels Key zuerst die Pos des Elements finden muss und dann `delete` mittels Pos aufruft. Es ist auch sichtbar, dass die Implementation mit Array und Container in Arrays die selbe Effizienz haben und die Implementation mit Container in einer Liste weniger effizient ist, also die anderen beiden.

T1	$add : SET \times ELEM \rightarrow SET \times POS$
Beschreibung	Hinzufügen eines Elements $e \in ELEM$ in die Menge $s \in SET$
pre-condition	-
post-condition	<code>s.find(e.key).isValid = TRUE</code>
T2	$delete : SET \times POS \rightarrow SET$
Beschreibung	Entfernen eines Elements $e \in ELEM$ in Position $p \in POS$ aus der Menge $s \in SET$
pre-condition	<code>p.getSet() == s &amp;&amp; s.retrieve(p) == e &amp;&amp; p.isValid = TRUE</code>
post-condition	<code>p.isValid = FALSE</code>
T3	$delete : SET \times KEY \rightarrow SET$
Beschreibung	Entfernen eines Elements $e \in ELEM$ in Position $p \in POS$ mit Schlüssel $k \in KEY$ aus der Menge $s \in SET$
pre-condition	<code>s.find(k).getSet() == s &amp;&amp; s.retrieve(s.find(k)) == e &amp;&amp; s.find(k).isValid = TRUE</code>
post-condition	<code>p.isValid = FALSE</code>
T4	$find : SET \times KEY \rightarrow POS$
Beschreibung	Suchen der Position $p \in POS$ eines Elements $e \in ELEM$ mit Schlüssel $k \in KEY$ aus der Menge $s \in SET$
pre-condition	-
post-condition	Wenn $k \in s$ : <code>s.find(k).isValid = TRUE</code> . Wenn $k \notin s$ : <code>s.find(k).isValid = FALSE</code>
T5	$retrieve : SET \times POS \rightarrow ELEM$
Beschreibung	Zugriff auf Element $e \in ELEM$ in Position $p \in POS$ aus der Menge $s \in SET$
pre-condition	<code>p.getSet() == s &amp;&amp; p.isValid = TRUE</code>
post-condition	<code>s.retrieve(p) == e</code>
T6	$size : SET \rightarrow INTEGER$
Beschreibung	Mächtigkeit der Menge $s \in SET$
pre-condition	-
post-condition	<code>s.size() ==  s </code>
T7	$unify : SET \times SET \rightarrow SET$
Beschreibung	Vereinigung zweier Mengen $s_1, s_2 \in SET$
pre-condition	-
post-condition	<code>s.unify(s1, s2) == s1 <math>\cup</math> s2</code>

Tabelle 1: Verifikationstests

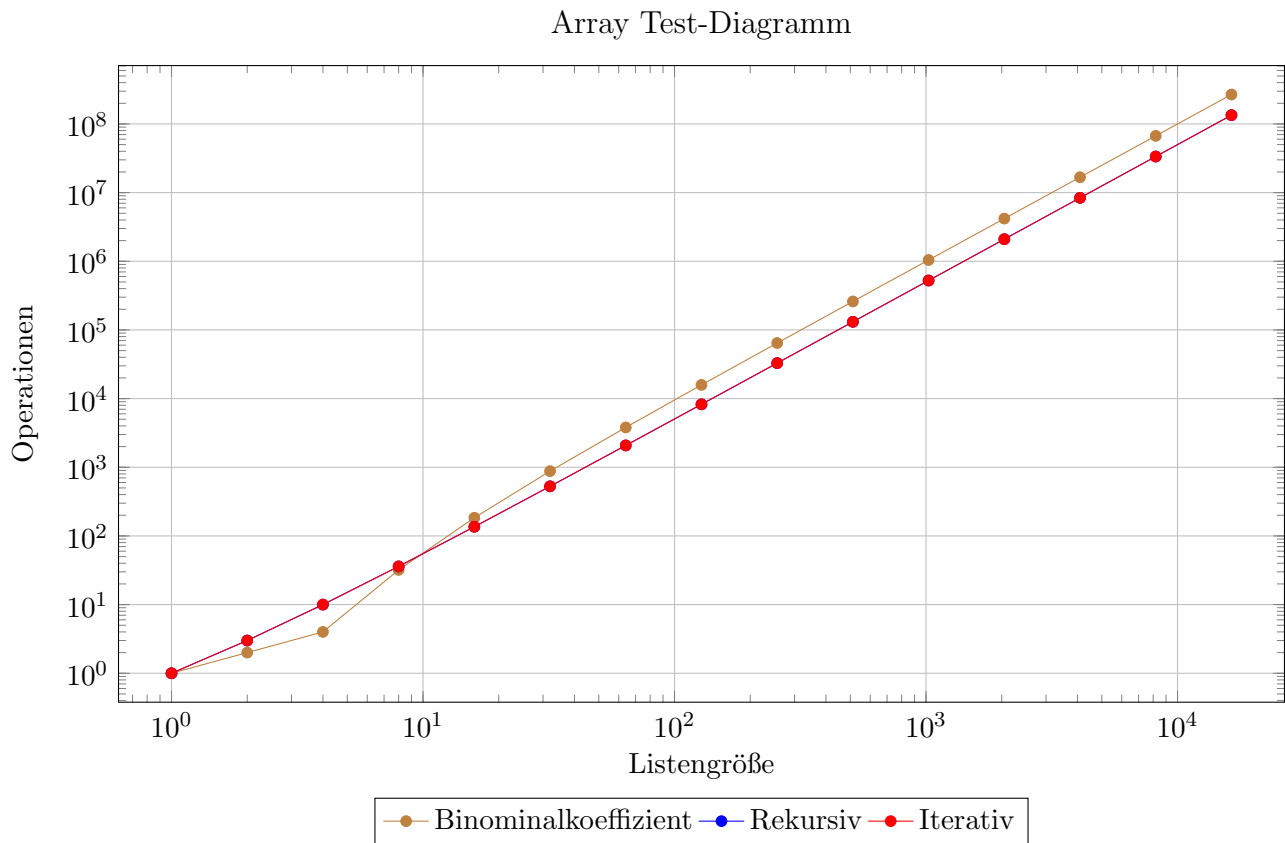


Abbildung 3.1: Quantitativer Vergleich der Berechnung einer Reihe im Pascalschen Dreieck in loglog-Darstellung

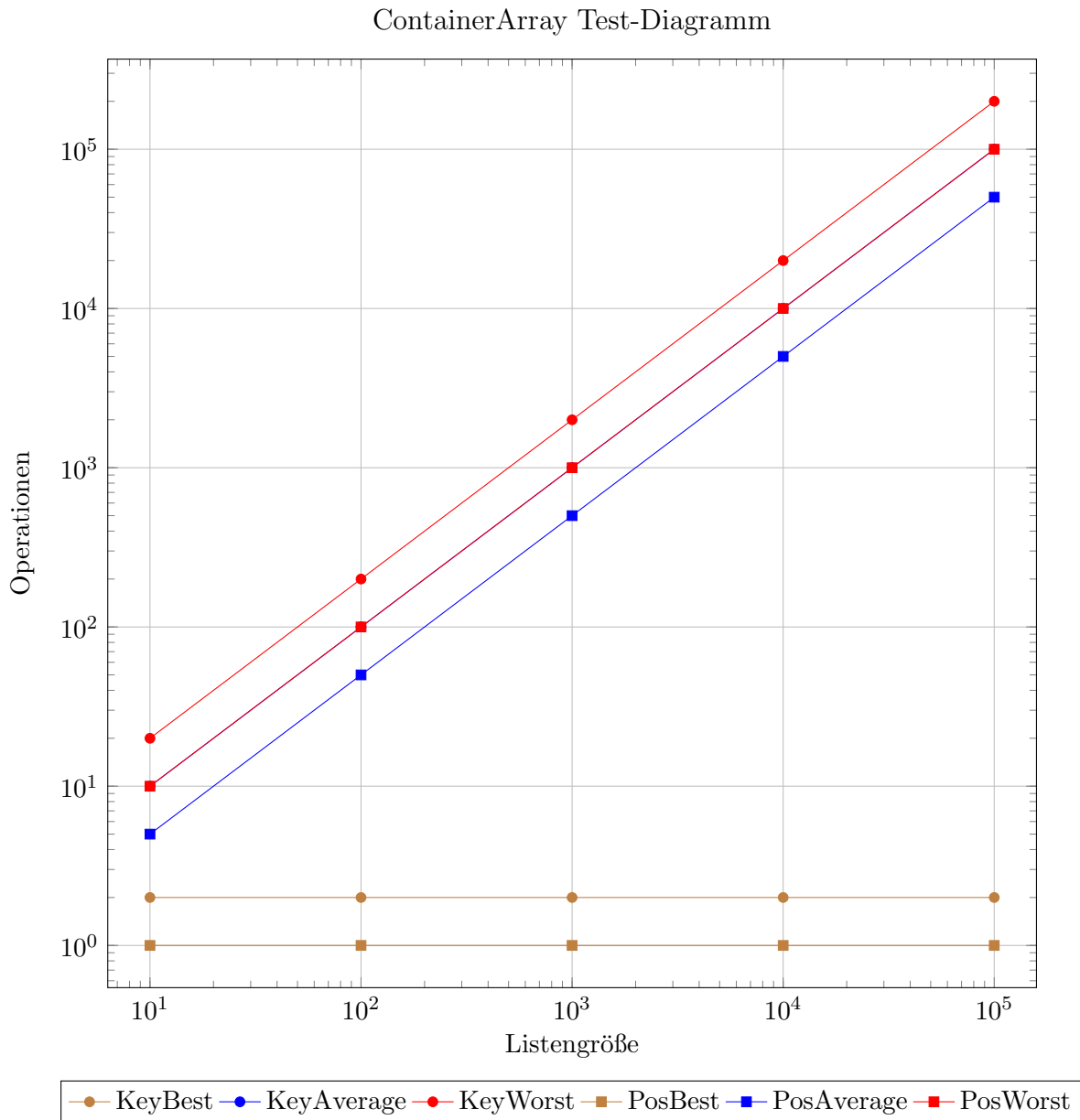


Abbildung 3.2: delete(Pos) und delete(Key) in Container-Array

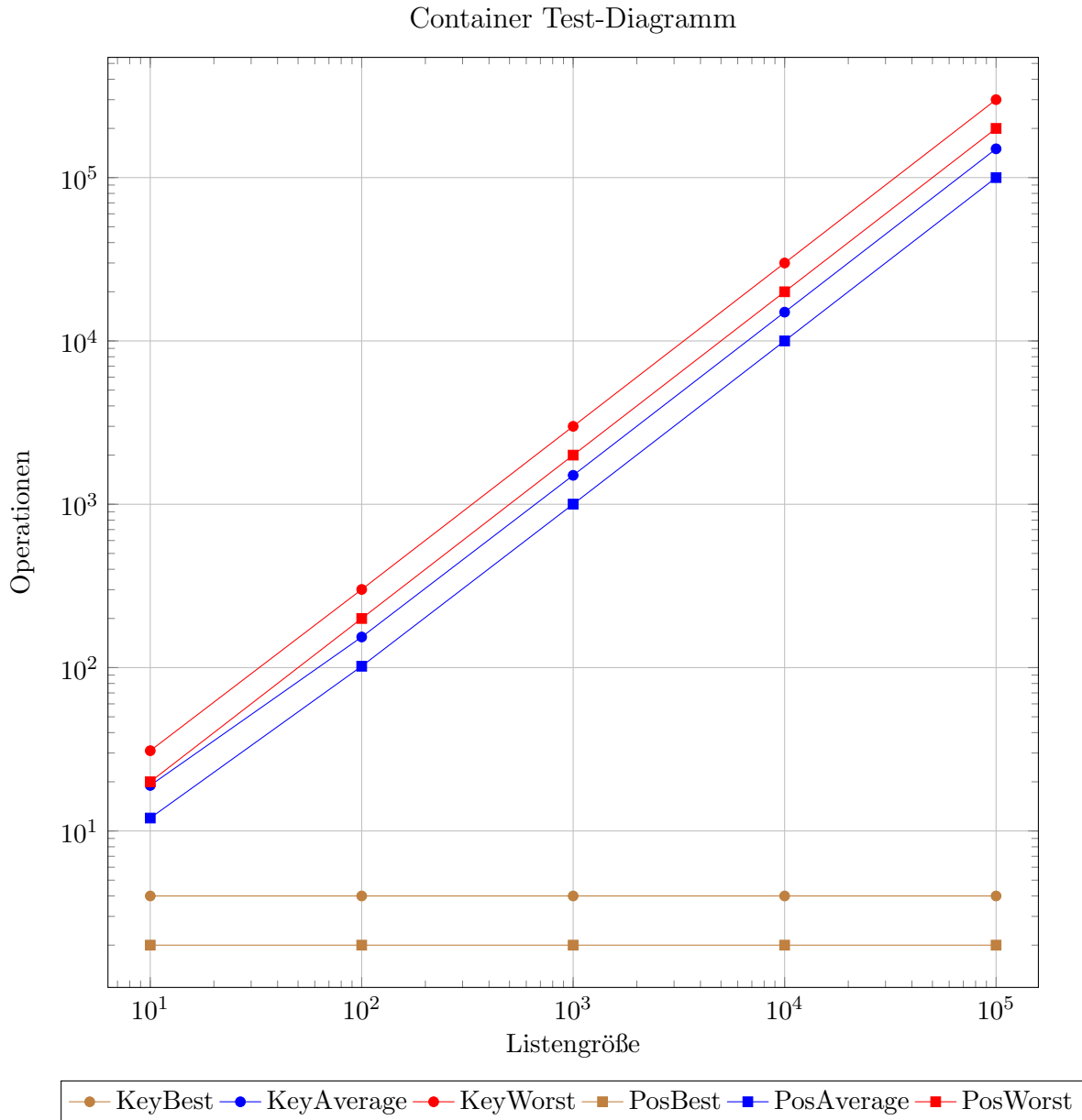


Abbildung 3.3: delete(Pos) und delete(Key) in Container-Liste