Praktische Informatik II | SoSe 2024
Tutor:in: Tobias Diehl | Bearbeiter:in: Altug Uyanik, Öykü Koç

Übungsblatt 1

Lösungsvorschlag

Set.java

Die Set Klasse wurde von **Altug Uyanik** verfasst.

```java
package de.uni_bremen.pi2;

import java.util.Iterator;

/**
 *@Author Altug Uyanik
 * Diese Klasse implementiert eine einfache Menge (Set) in Java.
 * @param <E> Der generische Typ der Elemente in der Menge.
 */
public class Set<E> implements Iterable<E> {

    /**
     * Die Elemente der Menge, gespeichert in einem Array.
     */
    public E[] elements;

    /**
     * Konstruktor für eine leere Menge.
     */
    public Set() {
        elements = (E[]) new Object[0];
    }
```

Aufgabe 1.1 Anzahl der Elemente

Um die Anzahl der Elemente in der Menge zu bestimmen, implementiere die Methode size(), die einfach die Anzahl der Elemente zurückgibt, die in der Menge enthalten sind.

```java
    /**
     * Gibt die Anzahl der Elemente in der Menge zurück.
     *
     * @return Die Anzahl der Elemente in der Menge.
     */
    public int size() {
        return elements.length;
    }
```

Aufgabe 1.2 Aufzählen der Elemente

Implementiere das Iterable-Interface, damit die Menge alle ihre Elemente aufzählen kann. Dazu musst du einen Iterator für die Menge erstellen, der die Methoden hasNext() und next() implementiert.

```java
    /**
     * Gibt einen Iterator zurück, der es ermöglicht, über die Elemente der Menge zu
           iterieren.
     *
     * @return Ein Iterator für die Elemente der Menge.
     */
```

```
37      @Override
38      public Iterator<E> iterator() {
39          return new Iterator<E>() {
40              private int currentIndex = 0;
41
42              @Override
43              public boolean hasNext() {
44                  return currentIndex < size();
45              }
46
47              @Override
48              public E next() {
49                  return (E) elements[currentIndex++];
50              }
51          };
52      }
```

Aufgabe 1.3 Test auf Vorhandensein

Um zu überprüfen, ob ein bestimmtes Element in der Menge vorhanden ist, implementiere die Methode contains(E), die true zurückgibt, wenn das übergebene Element in der Menge enthalten ist, andernfalls false. Vergleiche werden dabei über die equals()-Methode durchgeführt.

```
54      /**
55       * Überprüft, ob die Menge das angegebene Element enthält.
56       *
57       * @param element Das Element, das überprüft werden soll.
58       * @return true, wenn die Menge das Element enthält, andernfalls false.
59       */
60      public boolean contains(E element) {
61          boolean isContain = false;
62          Iterator<E> it = iterator();
63          E currentElement;
64
65          while (it.hasNext()) {
66              currentElement = it.next();
67              if (currentElement.equals(element)) {
68                  isContain = true;
69                  break;
70              }
71          }
72          return isContain;
73      }
```

Aufgabe 1.4 Einfügen eines Elements

Implementiere die Methode add(E), die ein übergebenes Element in die Menge einfügt, wenn es nicht bereits vorhanden ist. Du kannst dies überprüfen, indem du die Methode contains verwendest. Das Einfügen von null sollte abgelehnt werden.

```
75      /**
76       * Fügt ein Element zur Menge hinzu, sofern es nicht bereits vorhanden ist und nicht
                null ist.
77       *
78       * @param input Das Element, das zur Menge hinzugefügt werden soll.
79       */
80      public void add(E input) {
81          if (contains(input) || input == null)
82              return;
83
84          E[] temp = (E[]) new Object[size() + 1];
85
86          for (int i = 0; i < size(); i++) {
87              temp[i] = elements[i];
88          }
```

```
89
90          temp[temp.length - 1] = input;
91
92          elements = temp;
93      }
```

Aufgabe 1.5 Schnittmenge

Um die Schnittmenge $(A \cap B)$ aus dieser Menge und einer weiteren Menge zurückzugeben, implementiere die Methode intersect(Set<E>), die alle Elemente zurückgibt, die in beiden Mengen vorhanden sind.

```
95      /**
96       * Berechnet die Schnittmenge zwischen dieser Menge und einer anderen Menge.
97       *
98       * @param otherSet Die andere Menge, mit der die Schnittmenge berechnet werden soll.
99       * @return Die Schnittmenge der beiden Mengen.
100      */
101     public Set<E> intersect(Set<E> otherSet) {
102         Set<E> intersection = new Set<>();
103
104         for (E element : this) {
105             if (otherSet.contains(element)) {
106                 intersection.add(element);
107             }
108         }
109
110         return intersection;
111     }
```

Aufgabe 1.6 Vereinigungsmenge

Implementiere die Methode union(Set<E>), die die Vereinigungsmenge $(A \cup B)$ aus dieser Menge und einer weiteren Menge zurückgibt. Das bedeutet, dass alle Elemente, die in mindestens einer der beiden Mengen vorhanden sind, zurückgegeben werden

```
113     /**
114      * Berechnet die Vereinigungsmenge zwischen dieser Menge und einer anderen Menge.
115      *
116      * @param otherSet Die andere Menge, mit der die Vereinigungsmenge berechnet werden
             soll.
117      * @return Die Vereinigungsmenge der beiden Mengen.
118      */
119     public Set<E> union(Set<E> otherSet) {
120         Set<E> union = new Set<>();
121
122         for (E element : this) {
123             union.add(element);
124         }
125
126         for (E element : otherSet.elements) {
127             if (!contains(element)) {
128                 union.add(element);
129             }
130         }
131
132         return union;
133     }
```

Aufgabe 1.7 Differenzmenge

Um die Differenzmenge $(A \setminus B)$ aus dieser Menge und einer anderen Menge zurückzugeben, implementiere die Methode diff(Set<E>). Das bedeutet, dass alle Elemente, die in dieser Menge vorhanden sind, in der anderen Menge aber nicht, zurückgegeben werden.

```
135     /**
136      * Berechnet die Differenzmenge zwischen dieser Menge und einer anderen Menge.
137      *
138      * @param otherSet Die andere Menge, mit der die Differenzmenge berechnet werden soll
                .
139      * @return Die Differenzmenge der beiden Mengen.
140      */
141     public Set<E> diff(Set<E> otherSet) {
142         Set<E> difference = new Set<>();
143 
144         for (E element : this) {
145             if (!otherSet.contains(element)) {
146                 difference.add(element);
147             }
148         }
149 
150         return difference;
151     }
152 }
```

SetTest.java

Die SetTest Klasse wurde von **Öykü Koç** verfasst.

Die folgenden Tests überprüfen, ob die verschiedenen Methoden der Klasse Set korrekt funktionieren.

```
1  package de.uni_bremen.pi2;
2  
3  import static org.junit.jupiter.api.Assertions.*;
4  
5  import org.junit.jupiter.api.BeforeEach;
6  import org.junit.jupiter.api.DisplayName;
7  import org.junit.jupiter.api.Test;
8  
9  import java.util.Iterator;
10 
11 /**
12  *@Author Öykü Koç
13  * Test-Klasse für die Klasse Set<E>.
14  * Diese Klasse enthält Testmethoden für die Methoden der Set-Klasse,
15  * wie size, iterator, contains, add, intersect, union und diff.
16  */
17 public class SetTest {
18 
19     /**
20      * Inintialisierung von zwei Sets(SetA und SetB), mit denen im Folgenden die Tests
                durchgeführt werden.
21      */
22     Set<Integer> setA = new Set<>();
23     Set<Integer> setB = new Set<>();
24 
25     /**
26      * Einfügen von Integern in die Sets, sodass setA = [-256,-17,0,17,256] und setB =
                [-256,-18,0,17,128]
27      */
28     @BeforeEach
29     void addItems() {
30         setA.add(-256);
31         setA.add(-17);
```

```
32          setA.add(0);
33          setA.add(17);
34          setA.add(256);
35
36          setB.add(-256);
37          setB.add(-18);
38          setB.add(0);
39          setB.add(17);
40          setB.add(128);
41      }
```

Aufgabe 1.1 Anzahl der Elemente

Dieser Test ist ein JUnit-Test, der entwickelt wurde, um die Anzahl der Elemente einer Menge zu überprüfen. Er berechnet die Anzahl der Elemente in der Menge ‘setA‘ und überprüft, ob diese Anzahl mit 5 übereinstimmt, wie erwartet.

```
42
43      /**
44       * Testet die Methode size(), um die Anzahl der Elemente im Set zu überprüfen.
45       */
46      @Test
47      @DisplayName("Test the size of elements.")
48      void getElementsSizeTest() {
49          int size = setA.size();
50
51          assertEquals(5, size);
52      }
```

Aufgabe 1.2 Aufzählen der Elemente

Diese drei JUnit-Tests überprüfen das Verhalten des Iterators einer Set-Datenstruktur.

1. ‘iteratorHasNextFalseTest‘: Dieser Test bestätigt, dass der Iterator für eine leere Menge ‘hasNext()‘ als falsch zurückgibt.

2. ‘iteratorHasNextTrueTest‘: Dieser Test stellt sicher, dass der Iterator nach dem Hinzufügen eines Werts ‘hasNext()‘ als wahr zurückgibt.

3. ‘iteratorNextTest‘: Dieser Test überprüft, ob der Iterator den nächsten Wert korrekt zurückgibt. Dabei wird erwartet, dass der Iterator von ‘setA‘ das erste Element als -256 zurückgibt.

```
53      /**
54       * Testet den Iterator, um sicherzustellen, dass hasNext() False zurückgibt, wenn das
                 Set leer ist.
55       */
56      @Test
57      @DisplayName("Test Iterator HasNext With Non-Existent Value - Return False")
58      void iteratorHasNextFalseTest(){
59          Set<Integer> setIterator = new Set<>();
60
61          Iterator<Integer> it = setIterator.iterator();
62          boolean hasNext = it.hasNext();
63
64          assertFalse(hasNext);
65      }
```

```
66
67      /**
68       * Testet den Iterator, um sicherzustellen, dass hasNext() True zurückgibt, wenn das
                 Set nicht leer ist.
69       */
70      @Test
71      @DisplayName("Test Iterator HasNext With Existing Value - Return True")
72      void iteratorHasNextTrueTest(){
73          Set<Integer> setIterator = new Set<>();
74          setIterator.add(300);
75
76          Iterator<Integer> it = setIterator.iterator();
77          boolean hasNext = it.hasNext();
78
79          assertTrue(hasNext);
80      }
81
82      /**
83       * Testet den Iterator, um sicherzustellen, dass der nächste Wert im Set korrekt zurü
                 ckgegeben wird.
84       */
85      @Test
86      @DisplayName("Test Iterator Next Value")
87      void iteratorNextTest(){
88          Iterator<Integer> it = setA.iterator();
89          int nextValue = it.next();
90
91          assertEquals(-256, nextValue);
92      }
```

Aufgabe 1.3 Test auf Vorhandensein

Diese vier JUnit-Tests überprüfen das Verhalten der ‘contains()‘-Methode einer Set-Datenstruktur.

1. ‘existPositiveValueTest‘: Dieser Test überprüft, ob eine positive Zahl in der Menge enthalten ist. Das Ergebnis sollte True sein.

2. ‘notExistPositiveValueTest‘: Dieser Test überprüft, ob eine positive Zahl nicht in der Menge enthalten ist. Das Ergebnis sollte False sein.

3. ‘existNegativeValueTest‘: Dieser Test überprüft, ob eine negative Zahl in der Menge enthalten ist. Das Ergebnis sollte True sein.

4. ‘notExistNegativeValueTest‘: Dieser Test überprüft, ob eine negative Zahl nicht in der Menge enthalten ist. Das Ergebnis sollte False sein.

Diese Tests stellen sicher, ob die ‘contains()‘-Methode das erwartete Verhalten für positive und negative Zahlen zeigt.

```
94      /**
95       * Testet die Methode contains(), um zu überprüfen, ob eine positive Zahl im Set
                 enthalten ist.
96       * Das Ergebnis sollte True sein.
97       */
98      @Test
99      @DisplayName("Test for the existence of a positive value - Return True.")
100     void existPositiveValueTest() {
101         int existValue = 17;
```

```
102         boolean isContain = setA.contains(existValue);
103
104         assertTrue(isContain);
105     }
106
107     /**
108      * Testet die Methode contains(), um sicherzustellen, dass eine positive Zahl nicht
                im Set enthalten ist.
109      * Das Ergebnis sollte False sein.
110      */
111     @Test
112     @DisplayName("Test whether positive value is not contained - Return False")
113     void notExistPositiveValueTest() {
114         int notExistValue = 33;
115         boolean isContain = setA.contains(notExistValue);
116
117         assertFalse(isContain);
118     }
119
120     /**
121      * Testet die Methode contains(), um zu überprüfen, ob eine negative Zahl im Set
                enthalten ist.
122      * Das Ergebnis sollte True sein.
123      */
124     @Test
125     @DisplayName("Test if there is a negative value present - Return True.")
126     void existNegativeValueTest() {
127         int existValue = -17;
128         boolean isContain = setA.contains(existValue);
129
130         assertTrue(isContain);
131     }
132
133     /**
134      * Testet die Methode contains(), um sicherzustellen, dass eine negative Zahl nicht
                im Set enthalten ist.
135      * Das Ergebnis sollte False sein.
136      */
137     @Test
138     @DisplayName("Test if negative value is not contained - Return False")
139     void notExistNegativeValueTest() {
140         int notExistValue = -44;
141         boolean isContain = setA.contains(notExistValue);
142
143         assertFalse(isContain);
144     }
```

Aufgabe 1.4 Einfügen eines Elements

Diese drei JUnit-Tests überprüfen das Verhalten des Hinzufügens von Werten zu einer Set-Datenstruktur.

1. ‘addNotExistValueTest‘: Dieser Test überprüft, ob ein neuer Wert zum Set hinzugefügt wird. Zuerst wird die Größe des Sets vor dem Hinzufügen gespeichert, dann wird ein bisher nicht vorhandener Wert zum Set hinzugefügt und schließlich wird die Größe des Sets erneut überprüft. Das erwartete Verhalten ist, dass die Größe um 1 erhöht wird.

2. ‘addExistValueTest‘: Dieser Test überprüft, ob ein bereits vorhandener Wert erneut zum Set hinzugefügt wird. Zuerst wird die Größe des Sets gespeichert, dann wird ein bereits vorhandener Wert zum Set hinzugefügt und schließlich wird die Größe des Sets erneut überprüft. Das erwartete Verhalten ist, dass die Größe des Sets unverändert bleibt.

3. ‘addNullValueTest‘: Dieser Test überprüft, ob ein Null-Wert zum Set hinzugefügt werden

kann. Zuerst wird die Größe des Sets gespeichert, dann wird ein Null-Wert zum Set hinzugefügt und schließlich wird die Größe des Sets erneut überprüft. Das erwartete Verhalten ist, dass die Größe des Sets unverändert bleibt.

```
146     /**
147      * Testet die Methode add(), um das Hinzufügen von Werten zum Set zu überprüfen.
148      */
149     @Test
150     @DisplayName("Test adding a new value - Added")
151     void addNotExistValueTest() {
152         int firstSize = setA.size();
153         int notExistValue = 100;
154 
155         setA.add(notExistValue);
156         int lastSize = setA.size();
157 
158         assertEquals(firstSize + 1, lastSize);
159     }
160 
161     /**
162      * Testet die Methode add(), um zu überprüfen, dass ein bereits vorhandener Wert
               nicht erneut hinzugefügt wird.
163      */
164     @Test
165     @DisplayName("Test adding an existing value - Not Added.")
166     void addExistValueTest() {
167         int firstSize = setA.size();
168         int existValue = 256;
169 
170         setA.add(existValue);
171         int lastSize = setA.size();
172 
173         assertEquals(firstSize, lastSize);
174     }
175 
176     /**
177      * Testet die Methode add(), um sicherzustellen, dass ein null-Wert nicht zum Set
               hinzugefügt werden kann.
178      */
179     @Test
180     @DisplayName("Test adding a null value - Not Added.")
181     void addNullValueTest() {
182         int firstSize = setA.size();
183         setA.add(null);
184         int lastSize = setA.size();
185 
186         assertEquals(firstSize, lastSize);
187     }
```

Aufgabe 1.5 Schnittmenge

Diese drei JUnit-Tests überprüfen das Verhalten der ‘intersect()‘-Methode zweier Set-Datenstrukturen.

1. ‘intersectionSizeTest‘: Dieser Test überprüft die Größe des Schnitts zweier Sets. Die erwartete Größe des Schnitts beträgt 3, da der Schnitt die Werte [-256, 0, 17] enthält.

2. ‘intersectionValuesTest‘: Dieser Test überprüft, ob alle Werte im Schnitt korrekt im Schnitt-Set enthalten sind. Das erwartete Verhalten ist, dass das Schnitt-Set die Werte -256, 0 und 17 enthält.

3. ‘notIntersectionValueTest‘: Dieser Test überprüft, ob ein Wert, der nicht im Schnitt enthalten ist, auch nicht im Schnitt-Set enthalten ist. Das erwartete Verhalten ist, dass das Schnitt-Set

diesen Wert nicht enthält.

Diese Tests stellen sicher, dass die ‘intersect()‘-Methode den Schnitt zweier Sets korrekt berechnet und das Schnitt-Set die erwarteten Werte enthält.

```
189     /**
190      * Testet die Methode intersect(), um die Größe der Schnittmenge zweier Sets zu ü
                berprüfen.
191      * Intersection size = 3 => [-256, 0, 17]
192      */
193     @Test
194     @DisplayName("Test the size of the intersection")
195     void intersectionSizeTest() {
196         Set<Integer> intersectionSet = setA.intersect(setB);
197 
198         assertEquals(3, intersectionSet.size());
199     }
200 
201     /**
202      * Testet die Methode intersect(), um sicherzustellen, dass alle sich schneidenden
                Werte in der Schnittmenge enthalten sind.
203      * Das Ergebnis sollte True sein.
204      */
205     @Test
206     @DisplayName("Test All Intersection Values - Return True")
207     void intersectionValuesTest() {
208         Set<Integer> intersectionSet = setA.intersect(setB);
209 
210         assertTrue(intersectionSet.contains(-256));
211         assertTrue(intersectionSet.contains(0));
212         assertTrue(intersectionSet.contains(17));
213     }
214 
215     /**
216      * Testet die Methode intersect(), die beweist, dass ein Wert, der nicht zur
                Schnittmenge gehört, nicht in der Schnittmenge enthalten ist.
217      * Das Ergebnis sollte False sein.
218      */
219     @Test
220     @DisplayName("Test If No Intersection Values - Return False")
221     void notIntersectionValueTest() {
222         Set<Integer> intersectionSet = setA.intersect(setB);
223         int notIntersectedValue = 128;
224 
225         assertFalse(intersectionSet.contains(notIntersectedValue));
226     }
```

Aufgabe 1.6 Vereinigungsmenge

Diese beiden JUnit-Tests überprüfen das Verhalten der ‘union()‘-Methode zweier Set-Datenstrukturen.

1. ‘unionSizeTest‘: Dieser Test überprüft die Größe der Vereinigung zweier Sets. Die erwartete Größe der Vereinigung beträgt 7, da die Vereinigung die Werte [-256, -17, 0, 17, 256, -18, 128] enthält.

2. ‘unionValuesTest‘: Dieser Test überprüft, ob alle Werte in der Vereinigungsmenge korrekt im Vereinigungs-Set enthalten sind. Das erwartete Verhalten ist, dass das Vereinigungs-Set die Werte -256, -17, 0, 17, 256, -18 und 128 enthält.

Diese Tests stellen sicher, dass die ‘union()‘-Methode die Vereinigung zweier Sets korrekt berechnet und das Vereinigungs-Set die erwarteten Werte enthält.

```
228     /**
229      * Testet die Methode union(), um die Größe der Vereinigung zweier Sets zu überprüfen
                .
230      * Union size = 7 => [-256, -17, 0, 17, 256, -18, 128]
231      */
232     @Test
233     @DisplayName("Test Union Size")
234     void unionSizeTest() {
235         Set<Integer> unionSet = setA.union(setB);
236 
237         assertEquals(7, unionSet.size());
238     }
239 
240     /**
241      * Testet die Methode union(), um sicherzustellen, dass alle Werte in der
               Vereinigungsmenge enthalten sind.
242      * Das Ergebnis sollte True sein.
243      */
244     @Test
245     @DisplayName("Test If All Union Values Exist - Return True")
246     void unionValuesTest() {
247         Set<Integer> unionSet = setA.union(setB);
248 
249         assertTrue(unionSet.contains(-256));
250         assertTrue(unionSet.contains(-17));
251         assertTrue(unionSet.contains(0));
252         assertTrue(unionSet.contains(17));
253         assertTrue(unionSet.contains(256));
254         assertTrue(unionSet.contains(-18));
255         assertTrue(unionSet.contains(128));
256     }
```

Aufgabe 1.7 Differenzmenge

Diese drei JUnit-Tests überprüfen das Verhalten der ‘diff()‘-Methode zweier Set-Datenstrukturen.

1. ‘differencedSizeTest‘: Dieser Test überprüft die Größe der Differenz zweier Sets. Die erwartete Größe der Differenz beträgt 2, da die Differenz die Werte [-17, 256] enthält.

2. ‘differencedValuesTest‘: Dieser Test überprüft, ob alle Werte, die zur Differenz gehören, korrekt im Differenz-Set enthalten sind. Das erwartete Verhalten ist, dass das Differenz-Set die Werte -17 und 256 enthält.

3. ‘notDifferencedValueTest‘: Dieser Test überprüft, ob ein Wert, der nicht zur Differenz gehört, nicht im Differenz-Set enthalten ist. Das erwartete Verhalten ist, dass das Differenz-Set diesen Wert nicht enthält.

Diese Tests stellen sicher, dass die ‘diff()‘-Methode die Differenz zweier Sets korrekt berechnet und das Differenz-Set die erwarteten Werte enthält.

```
258     /**
259      * Testet die Methode diff(), um die Größe der Differenz zweier Sets zu überprüfen.
260      * Diff setA/setB size = 2 => [-17, 256]
261      */
262     @Test
263     @DisplayName("Test Difference Size")
264     void differencedSizeTest() {
265         Set<Integer> differenceSet = setA.diff(setB);
266 
267         assertEquals(2, differenceSet.size());
268     }
269 
```

```
270     /**
271      * Testet die Methode diff(), um sicherzustellen, dass alle Werte, die zur Differenz
                gehören, in der Differenz Menge enthalten sind.
272      * Das Ergebnis sollte True sein.
273      */
274     @Test
275     @DisplayName("Test All Difference Values - Return True")
276     void differencedValuesTest() {
277         Set<Integer> differenceSet = setA.diff(setB);
278
279         assertTrue(differenceSet.contains(-17));
280         assertTrue(differenceSet.contains(256));
281     }
282
283     /**
284      * Testet die Methode diff(), um sicherzustellen, dass ein Nicht-Differenzwert nicht
                in die Differenzmenge aufgenommen wird.
285      * Das Ergebnis sollte False sein.
286      */
287     @Test
288     @DisplayName("Test If No Difference Values Exist - Return False")
289     void notDifferencedValueTest() {
290         Set<Integer> differenceSet = setA.diff(setB);
291         int notDifferencedValue = 0;
292
293         assertFalse(differenceSet.contains(notDifferencedValue));
294     }
295
296 }
```

Unten sehen Sie die Ergebnisse des Pit-Tests, den wir für Set.java durchgeführt haben.

Pit Test Coverage Report

Package Summary

de.uni_bremen.pi2

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
1	100%	41/41	100%	22/22	100%	22/22

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
Set.java	100%	41/41	100%	22/22	100%	22/22

Report generated by PIT 1.15.8

Abbildung 1: Pit Test Coverage Report