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

Übungsblatt 3

Lösungsvorschlag

MergeSort.java

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

Aufgabe 1.1 Implementierung

Implementiert das in der Vorlesung vorgestellte Sortierverfahren Merge Sort für einfach verkettete Listen. Die Listen bestehen ausschließlich aus Knoten der bereitgestellten Klasse Node, d.h. es gibt kein zusätzliches Objekt, das eine Referenz auf den Anfang der Liste enthält. Die in den Knoten gespeicherten Werte sind Comparable. Die dadurch definierte natürliche Ordnung wird für die Sortierung genutzt. Die Methode MergeSort.sort, die als Rahmen angegeben wird, erhält einen Verweis auf den ersten Knoten der zu sortierenden Liste. und gibt als Ergebnis einen Verweis auf den ersten Knoten der sortierten Liste zurück. Knoten der sortierten Liste als Ergebnis zurück. Die sortierte Liste enthält die gleichen Knoten wie die Knoten der Liste.

```
3  public class MergeSort
4  {
5      /**
6       * Findet den mittleren Knoten der verketteten Liste.
7       * Diese Methode verwendet die Fast- und Slow-Pointer-Technik, um den mittleren
                Knoten zu finden.
8       *
9       * @param head der Kopfknoten der verketteten Liste
10      * @return der mittlere Knoten der verketteten Liste
11      */
12     public static Node getMiddleNode(Node head)
13     {
14         if (head == null)
15             return head;
16
17         Node slowNode = head;
18         Node fastNode = head.getNext();
19
20         while (fastNode != null) {
21             fastNode = fastNode.getNext();
22             if (fastNode != null) {
23                 slowNode = slowNode.getNext();
24                 fastNode = fastNode.getNext();
25             }
26         }
27         return slowNode;
28     }
29
30     /**
31      * Verschmilzt zwei sortierte verkettete Listen zu einer einzigen sortierten
                verketteten Liste.
32      *
33      * @param leftNode der Kopfknoten der ersten sortierten verketteten Liste
34      * @param rightNode der Kopfknoten der zweiten sortierten verketteten Liste
35      * @return der Kopfknoten der verschmolzenen und sortierten verketteten Liste
36      */
37     public static Node sortAndMerge(Node leftNode, Node rightNode)
38     {
39         Node result = null;
```

```
40          /* Base cases */
41          if (leftNode == null)
42              return rightNode;
43          if (rightNode == null)
44              return leftNode;
45
46          /* Pick either a or b, and recur */
47          if (leftNode.getValue().compareTo(rightNode.getValue()) <= 0) {
48              result = leftNode;
49              result.setNext(sortAndMerge(leftNode.getNext(), rightNode));
50          }
51          else {
52              result = rightNode;
53              result.setNext(sortAndMerge(leftNode, rightNode.getNext()));
54          }
55          return result;
56      }
57
58
59      /**
60       * Führt den Merge-Sort-Algorithmus rekursiv auf der verketteten Liste aus.
61       *
62       * @param head der Kopfknoten der zu sortierenden verketteten Liste
63       * @param <E> der Typ der Elemente, die Comparable implementieren
64       * @return der Kopfknoten der sortierten verketteten Liste
65       */
66      public static <E extends Comparable<E>> Node<E> sort(final Node<E> head)
67      {
68          // Base case : if head is null
69          if (head == null || head.getNext() == null) {
70              return head;
71          }
72
73          // get the middle of the list
74          Node<E> firstHalfTail = getMiddleNode(head);
75          Node<E> secondHalfHead = firstHalfTail.getNext();
76
77          // set the next of middle Node to null
78          firstHalfTail.setNext(null);
79
80          // Apply mergeSort on left list
81          Node<E> leftHalf = sort(head);
82
83          // Apply mergeSort on right list
84          Node<E> rightHalf = sort(secondHalfHead);
85
86          // Merge the left and right lists
87          Node<E> sortedlist = sortAndMerge(leftHalf, rightHalf);
88
89          return sortedlist;
90      }
91
92
93  }
```

Aufgabe 1.2 Tests

MergeSortTest.java

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

testGetMiddleNodeEmptyListShouldBeNull

Dieser Test überprüft, ob die Methode ‘MergeSort.getMiddleNode‘ für eine leere verkettete Liste ‘null‘ zurückgibt. Im Falle einer leeren Liste soll die Methode ‘null‘ zurückgeben.

testGetMiddleNodeSingleElementListShouldBeOne

Dieser Test überprüft, ob die Methode ‘MergeSort.getMiddleNode‘ für eine verkettete Liste mit einem einzigen Element dieses eine Element zurückgibt. Die Methode soll bei einer Liste mit einem einzigen Element den richtigen Knoten zurückgeben.

testGetMiddleNodeEvenElementsListShouldBeTwo

Dieser Test überprüft, ob die Methode ‘MergeSort.getMiddleNode‘ für eine verkettete Liste mit einer geraden Anzahl von Elementen den ersten der beiden mittleren Knoten zurückgibt. Bei einer Liste mit vier Elementen soll die Methode den zweiten Knoten zurückgeben.

testGetMiddleNodeOddElementsListShouldBeThree

Dieser Test überprüft, ob die Methode ‘MergeSort.getMiddleNode‘ für eine verkettete Liste mit einer ungeraden Anzahl von Elementen den mittleren Knoten zurückgibt. Bei einer Liste mit fünf Elementen soll die Methode den dritten Knoten zurückgeben.

testGetMiddleNodeLongListShouldBeFive

Dieser Test überprüft, ob die Methode ‘MergeSort.getMiddleNode‘ für eine lange verkettete Liste mit zehn Elementen den fünften Knoten zurückgibt. Die Methode soll bei einer langen Liste den richtigen mittleren Knoten finden.

```
1  package de.uni_bremen.pi2;
2
3  import static org.junit.jupiter.api.Assertions.*;
4
5  import org.junit.jupiter.api.Test;
6
7  /*
8   * @Author Öykü Koç
9   */
10 public class MergeSortTest {
11
12     /**
13      * Gibt den mittleren Knoten einer gegebenen verketteten Liste zurück.
14      * Wenn die Liste leer ist, wird null zurückgegeben.
15      */
16     @Test
17     void testGetMiddleNodeEmptyList_ShouldBeNull() {
18         Node<Integer> emptyList = asList();
19         Node<Integer> middleNode = MergeSort.getMiddleNode(emptyList);
20         assertNull(middleNode);
21     }
22
23     /**
24      * Gibt den mittleren Knoten einer verketteten Liste mit einem einzelnen Element zurü
                ck.
25      */
26     @Test
27     void testGetMiddleNodeSingleElementList_ShouldBeOne() {
28         Node<Integer> singleElement = asList(1);
29         Node<Integer> middleNode = MergeSort.getMiddleNode(singleElement);
30         assertEquals(1, middleNode.getValue());
31     }
32
33     /**
34      * Gibt den mittleren Knoten einer verketteten Liste mit einer geraden Anzahl von
                Elementen zurück.
```

```
35      */
36     @Test
37     void testGetMiddleNodeEvenElementsList_ShouldBeTwo() {
38         Node<Integer> evenList = asList(1, 2, 3, 4);
39         Node<Integer> middleNode = MergeSort.getMiddleNode(evenList);
40         assertEquals(2, middleNode.getValue());
41     }
42
43     /**
44      * Gibt den mittleren Knoten einer verketteten Liste mit einer ungeraden Anzahl von
                Elementen zurück.
45      */
46     @Test
47     void testGetMiddleNodeOddElementsList_ShouldBeThree() {
48         Node<Integer> nodeOddElementList = asList(1, 2, 3, 4, 5);
49         Node<Integer> middleNode = MergeSort.getMiddleNode(nodeOddElementList);
50         assertEquals(3, middleNode.getValue());
51     }
52
53     /**
54      * Gibt den mittleren Knoten einer langen verketteten Liste zurück.
55      */
56     @Test
57     void testGetMiddleNodeLongList_ShouldBeFive() {
58         Node<Integer> nodeLongList = asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
59         Node<Integer> middleNode = MergeSort.getMiddleNode(nodeLongList);
60         assertEquals(5, middleNode.getValue());
61     }
```

testSortAndMergePositiveReturnSortedList

Dieser Test überprüft die Methode ‘sortAndMerge‘ mit zwei positiv sortierten Listen und stellt sicher, dass die zurückgegebene Liste korrekt zusammengeführt und sortiert ist. Die Methode soll eine sortierte Liste zurückgeben, die die Elemente beider Eingabelisten enthält.

testSortingUnsortedListShouldReturnTrue

Dieser Test überprüft das Sortieren einer unsortierten verketteten Liste. Die Methode ‘MergeSort.sort‘ soll eine korrekt sortierte Liste zurückgeben. Bei einer unsortierten Liste (8, 6, 7, 4) soll die Methode eine sortierte Liste (4, 6, 7, 8) zurückgeben.

testSortingEmptyListShouldReturnNull

Dieser Test überprüft das Sortieren einer leeren verketteten Liste. Die Methode ‘MergeSort.sort‘ soll ‘null‘ zurückgeben, wenn die Eingabeliste leer ist.

testSortingListWithDuplicateElementsShouldReturnTrue

Dieser Test überprüft das Sortieren einer verketteten Liste mit doppelten Elementen. Die Methode ‘MergeSort.sort‘ soll eine korrekt sortierte Liste zurückgeben, auch wenn die Liste doppelte Elemente enthält. Bei der Liste (11, 11, 9, 21, 21) soll die Methode eine sortierte Liste (9, 11, 11, 21, 21) zurückgeben.

testSortingAlreadySortedListShouldReturnGivenList

Dieser Test überprüft das Sortieren einer bereits sortierten verketteten Liste. Die Methode ‘MergeSort.sort‘ soll die bereits sortierte Liste unverändert zurückgeben. Bei der Liste (8, 9, 10, 14) soll die Methode dieselbe Liste zurückgeben.

testSortingSingleElementListShouldReturnOne

Dieser Test überprüft das Sortieren einer verketteten Liste mit einem einzigen Element. Die Methode ‘MergeSort.sort‘ soll die Liste mit einem einzigen Element unverändert zurückgeben. Bei der Liste (1) soll die Methode dieselbe Liste (1) zurückgeben.

```
63      /**
64       * Testet die Methode ‘sortAndMerge‘ mit zwei positiv sortierten Listen und überprüft
             ,
65       * ob die zurückgegebene Liste korrekt zusammengeführt und sortiert ist.
66       */
67      @Test
68      void testSortAndMerge_Positive_ReturnSortedList() {
69          Node<Integer> leftNode = asList(1, 3, 5);
70          Node<Integer> rightNode = asList(2, 4, 6);
71          assertListEquals(MergeSort.sortAndMerge(leftNode, rightNode), 1, 2, 3, 4, 5, 6);
72      }
73
74      /**
75       * Testet das Sortieren einer unsortierten verketteten Liste.
76       */
77      @Test
78      void testSortingUnsortedList_ShouldReturnTrue() {
79          Node<Integer> unsortedList = asList(8, 6, 7, 4);
80          Node<Integer> sortedList = MergeSort.sort(unsortedList);
81          assertListEquals(sortedList, 4, 6, 7, 8);
82      }
83
84      /**
85       * Testet das Sortieren einer leeren verketteten Liste.
86       */
87      @Test
88      void testSortingEmptyList_ShouldReturnNull() {
89          Node<Integer> emptyList = asList();
90          Node<Integer> sortedList = MergeSort.sort(emptyList);
91          assertListEquals(sortedList);
92      }
93
94      /**
95       * Testet das Sortieren einer verketteten Liste mit doppelten Elementen.
96       */
97      @Test
98      void testSortingListWithDuplicateElements_ShouldReturnTrue() {
99          Node<Integer> duplicateElementsList = asList(11, 11, 9, 21, 21);
100         Node<Integer> sortedList = MergeSort.sort(duplicateElementsList);
101         assertListEquals(sortedList, 9, 11, 11, 21, 21);
102     }
103
104     /**
105      * Testet das Sortieren einer bereits sortierten verketteten Liste.
106      */
107     @Test
108     void testSortingAlreadySortedList_ShouldReturnGivenList() {
109         Node<Integer> alreadySortedList = asList(8, 9, 10, 14);
110         Node<Integer> sortedList = MergeSort.sort(alreadySortedList);
111         assertListEquals(sortedList, 8, 9, 10, 14);
112     }
113
114     /**
115      * Testet das Sortieren einer verketteten Liste mit einem einzelnen Element.
116      */
117     @Test
```

```
118     void testSortingSingleElementList_ShouldReturnOne() {
119         Node<Integer> singleElementList = asList(1);
120         Node<Integer> sortedList = MergeSort.sort(singleElementList);
121         assertListEquals(sortedList, 1);
122     }
```

testSortedAsListValuesShouldBeEqualToGivenListValues

Dieser Test überprüft, ob die sortierten Werte einer Liste den erwarteten Werten entsprechen. Die Methode ‘MergeSort.sort‘ soll eine Liste zurückgeben, deren Werte in der richtigen Reihenfolge sortiert sind.

testSortedNodeHeadValueShouldBeOne

Dieser Test überprüft, ob der Wert des Kopfknotens einer sortierten Liste eins ist. Die Methode ‘MergeSort.sort‘ soll eine sortierte Liste zurückgeben, deren Kopfknoten den Wert 1 hat.

testSortedNodeHeadNextValueShouldBeTwo

Dieser Test überprüft, ob der Wert des nächsten Knotens nach dem Kopfknoten in einer sortierten Liste zwei ist. Die Methode ‘MergeSort.sort‘ soll eine sortierte Liste zurückgeben, deren zweiter Knoten den Wert 2 hat.

testSortedNegativeValuesShouldBeNegativeFifty

Dieser Test überprüft, ob der Wert des Kopfknotens einer sortierten Liste -50 ist. Die Methode ‘MergeSort.sort‘ soll eine sortierte Liste zurückgeben, deren Kopfknoten den Wert -50 hat.

testSortedPositiveAndNegativeValuesShouldBeNegativeTwenty

Dieser Test überprüft, ob der Wert des Kopfknotens einer sortierten Liste -20 ist. Die Methode ‘MergeSort.sort‘ soll eine sortierte Liste zurückgeben, deren Kopfknoten den Wert -20 hat, wenn sowohl positive als auch negative Werte vorhanden sind.

testSortedNodeLastNextShouldBeNull

Dieser Test überprüft, ob das nächste Element nach dem letzten Element in einer sortierten Liste null ist. Die Methode ‘MergeSort.sort‘ soll eine sortierte Liste zurückgeben, deren letztes Element keinen Nachfolger hat (d.h. der nächste Knoten nach dem letzten Knoten soll ‘null‘ sein).

Diese Tests werden verwendet, um sicherzustellen, dass die Sortiermethoden der Klasse ‘MergeSort‘ in verschiedenen Szenarien korrekt funktionieren, einschließlich der Überprüfung der Reihenfolge und der Werte der sortierten Listen.

```
124     /**
125      * Testet, ob die sortierten Werte einer Liste den erwarteten Werten entsprechen.
126      */
127     @Test
128     public void testSortedAsListValues_ShouldBeEqualTo_GivenListValues() {
```

```
129         assertListEquals(MergeSort.sort(asList(5, 4, 3, 2, 1)), 1, 2, 3, 4, 5);
130     }
131
132     /**
133      * Testet, ob der Wert des Kopfknotens einer sortierten Liste eins ist.
134      */
135     @Test
136     void testSortedNodeHeadValue_ShouldBeOne() {
137         assertEquals(1, MergeSort.sort(asList(5, 4, 3, 2, 1)).getValue());
138     }
139
140     /**
141      * Testet, ob der Wert des nächsten Knotens nach dem Kopfknoten in einer sortierten
                Liste zwei ist.
142      */
143     @Test
144     void testSortedNodeHeadNextValue_ShouldBeTwo() {
145         assertEquals(2, MergeSort.sort(asList(5, 3, 4, 1, 2)).getNext().getValue());
146     }
147
148     /**
149      * Testet, ob der Wert des Kopfknotens einer sortierten Liste -50 ist.
150      */
151     @Test
152     void testSortedNegativeValues_ShouldBeNegativeFifty() {
153         assertEquals(-50, MergeSort.sort(asList(-20, -50, -10, -40, -30)).getValue());
154     }
155
156     /**
157      * Testet, ob der Wert des Kopfknotens einer sortierten Liste -20 ist.
158      */
159     @Test
160     void testSortedPositiveAndNegativeValues_ShouldBeNegativeTwenty() {
161         assertEquals(-20, MergeSort.sort(asList(-20, 50, -10, 40, 30)).getValue());
162     }
163
164     /**
165      * Testet, ob das nächste Element nach dem letzten Element in einer sortierten Liste
                null ist.
166      */
167     @Test
168     void testSortedNodeLastNext_ShouldBeNull() {
169         assertNull(MergeSort.sort(asList(3, 1, 4)).getNext().getNext().getNext());
170     }

172     /**
173      * Erzeugt eine Liste aus einer Folge von Werten.
174      * Z.B. erzeugt asList(1, 2, 3) eine Liste mit den Werten 1, 2 und 3.
175      *
176      * @param values Die Werte, aus denen die Liste erzeugt wird. Können einfach
177      *               aufgezählt werden.
178      * @param <E>    Der Typ der Werte.
179      * @return Die Liste, die die Werte in der Reihenfolge enthält, in der sie
180      * aufgezählt wurden.
181      */
182     private <E extends Comparable<E>> Node<E> asList(final E... values) {
183         if (values == null || values.length == 0) {
184             return null;
185         }
186
187         Node<E> head = new Node<>(values[0]);
188         Node<E> current = head;
189
190         for (int i = 1; i < values.length; i++) {
191             Node<E> newNode = new Node<>(values[i]);
192             current.setNext(newNode);
193             current = current.getNext();
194         }
195
196         return head;
197     }
198
199     /**
```

```
200      * Überprüft, ob eine Liste einer bestimmten Folge von Werten entspricht.
201      * Kann z.B. wie folgt genutzt werden:
202      * <pre>
203      * assertListEquals(MergeSort.sort(asList(3, 2, 1)), 1, 2, 3);
204      * </pre>
205      *
206      * @param head   Der erste Knoten der Liste, deren Inhalt verglichen wird.
207      * @param values Die Vergleichswerte. Können einfach aufgezählt werden.
208      * @param <E>    Der Typ der Werte, die verglichen werden.
209      */
210     private <E extends Comparable<E>> void assertListEquals(final Node<E> head, final E
             ... values) {
211         Node<E> currentNode = head;
212
213         for (E value : values) {
214             assertEquals(0, value.compareTo(currentNode.getValue()));
215             currentNode = currentNode.getNext();
216         }
217         assertNull(currentNode);
218     }
219 }
```