Übungen Woche 9

Patrick Bucher

29.04.2017

1 Quicksort – theoretisch durchgespielt

a)

Erster Durchgang:

12 10 52_1 9 77 23 18 52_2 11 25 8 5 *17*

12 10 5 9 8 11 *17* 52_2 23 25 77 52_1 18

Zweiter Durchgang:

12 10 5 9 8 *11* | 17 | 52_2 23 25 77 52_1 *18*

8 10 5 9 *11* 12 | 17 | *18* 23 25 77 52_1 52_2

Dritter Durchgang:

9 10 5 *9* | 11 12 17 18 | 23 25 77 52_1 *52_2*

8 5 *9* 10 | 11 12 17 18 | 23 25 52_1 *52_2* 77

Vierter Durchgang:

8 *5* | 9 10 11 12 17 18 | 23 25 *52_1* | 52_2 77

5 8 | 9 10 11 12 17 18 | 23 25 *52_1* | 52_2 77

Fünfter Durchgang

5 8 9 10 11 12 17 18 | 23 *25* | 52_1 52_2 77

5 8 9 10 11 12 17 18 23 *25* 52_1 52_2 77

b)

Die Zahl 52_1 war nach dem ersten Durchgang rechts von 52_2. Dass die Reihenfolge im zweiten Durchgang noch einmal (und zwar endgültig) änderte, ist reiner Zufall. Quicksort arbeitet *instabil*.

c)

Beim ersten Durchgang kämen 12 (Index 0), 18 (Index 6) und 17 (Index 12) in Frage. Dadurch würde erneut 17 als Trennelement fungieren.

Beim zweiten Durchgang würde es links wiederum genau gleich ablaufen, rechts würde aber mit 25 ein anderes Element verwendet werden. Das könnte die Sortierung etwas beschleunigen und evtl. einen fünften Durchgang ersparen.

2 Quicksort – klassisch programmiert

a)

```
public static void quickSort(Character[] data, int left, int right) {
    int up = left;
    int down = right - 1;
    char t = data[right];
    do {
        while (data[up] < t) {
            up++;
        }
        while (data[down] >= t && down > up) {
            down--;
        }
        if (up >= down) {
            break;
        }
        swap(data, up, down);
    } while (true);
    swap(data, up, right);
    if (left < up - 1) {
        quickSort(data, left, up - 1);
    }
    if (right > up + 1) {
        quickSort(data, up + 1, right);
    }
}
```

Testfall:

```
@Test
public void testQuickSort() {
    final int n = 200_000;
    Character data[] = SortingUtils.generateRandomCharArray(n, 'A', 'Z');
```

```
    Sort.quickSort(data, 0, data.length - 1);
    boolean sorted = SortingUtils.isSorted(Arrays.asList(data), true);
    Assert.assertTrue(sorted);
}
```

b)

```
public static void quickSort(Character[] data) {
    quickSort(data, 0, data.length - 1);
}
```

c)

```
public static Character[] randomChars(int size, int min, int max) {
    Random random = new Random(System.currentTimeMillis());
    Character array[] = new Character[size];
    for (int i = 0; i < size; i++) {
        array[i] = (char) (random.nextInt(max - min + 1) + min);
    }
    return array;
}
```

d)

Elemente (n)	Messung (ms)
1000	2
5000	7
10’000	12
50’000	35
100’000	133
500’000	2993
1’000’000	11’888

Beispiel: Um welchen Faktor müsste eine Sortierung mit 1’000’000 Elementen länger dauern als eine Sortierung mit 500’000 bzw. 100’000 Elementen?

```
(1'000'000 * log 1'000'000) / (500'000 * log 500'000) = 2.1
(1'000'000 * log 1'000'000) / (100'000 * log 100'000) = 12
```

Realität:

```
11'888 / 2993 = 3.97
11'888 / 133 = 89.4
```

Das Laufzeitverhalten scheint eher $O(n^2)$ zu entsprechen (eine Verdoppelung der Elemente führt zu einer Verfierfachung der Laufzeit; eine Verzehnfachung der Elemente erhöht die Laufzeit ca. um Faktor 90).

3 Quick-Insertion-Sort

a)

Zur eigentlichen Methode, die zusätzlich einen Parameter `m` hat (Schwellenwert, unter dem der Insertion-Sort verwendet werden soll), implementiere ich wiederum eine Hilfsmethode.

```java
public static void quickInsertionSort(Character[] data, int m) {
    quickInsertionSort(data, 0, data.length - 1, m);
}

static void quickInsertionSort(Character[] data, int l, int r, int m) {
    // bestehender Algorithmus
    // ...
    // veränderte Rekursionsanweisung
    if (left < up - 1) {
        int from = left;
        int to = up - 1;
        if (to - from > m) {
            quickInsertionSort(data, from, to, m);
        } else {
            SimpleSorting.insertionSort(data, from, to);
        }
    }
    if (right > up + 1) {
        int from = up + 1;
        int to = right;
        if (to - from > m) {
            quickInsertionSort(data, from, to, m);
        } else {
            SimpleSorting.insertionSort(data, from, to);
        }
    }
}
```

b)

Ich sortiere eine Million Zeichen mit verschiedenen `m`-Werten. Dabei erhalte ich folgende Laufzeiten:

m	Zeit (ms)
5	9420
10	9278
15	9259
20	9361
25	9269
30	9258
40	9267
50	9263
75	9267
100	9289
125	9265
150	9260
200	9300
250	9259
500	9255
1000	9410

Bei einer Datenmenge von einer Million Zeichen scheinen sinnvolle `m`-Werte von 10 bis 500 zu liegen.

c)

Mit `m = 25` kann ich beim Quick-Insertion-Sort (QIS) gegenüber dem Quick-Sort (QS) keinen Laufzeitvorteil feststellen:

n	QS (ms)	QIS (ms)
1000	2	2
5000	6	6
10000	4	15
50000	35	33
100000	134	126
500000	3080	3043
1000000	11866	12101
2000000	47465	48360

4 Datenstruktur Heap

Ich habe den Heap gleich generisch und mit dynamischer Grösse implementiert.

a)

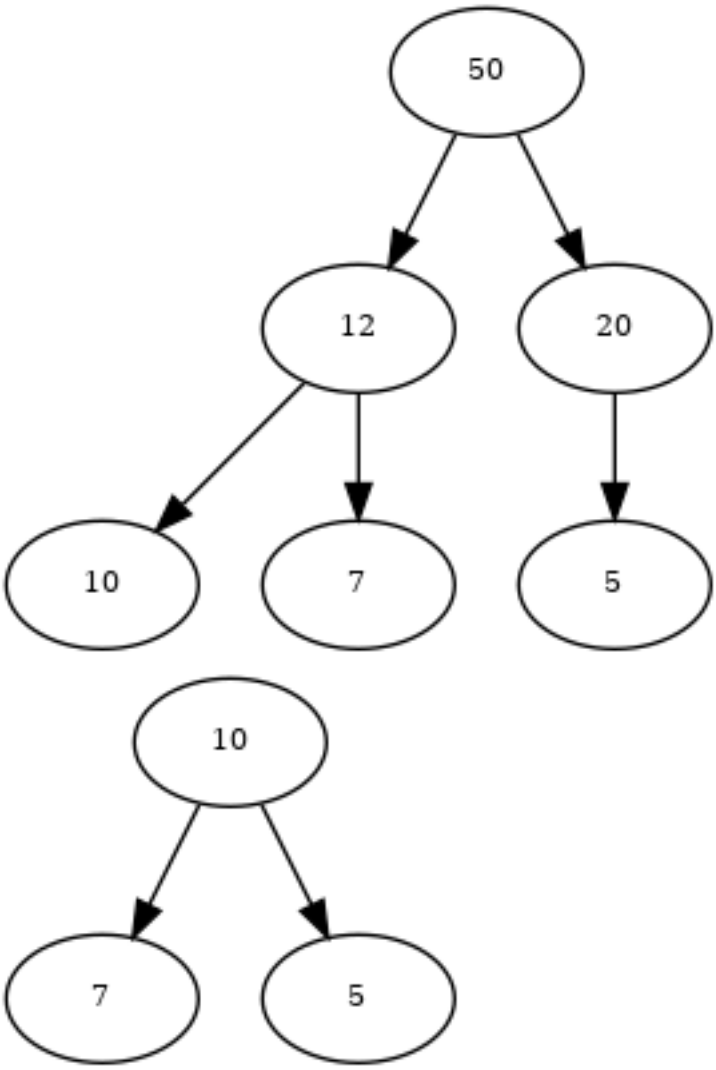

Abbildung 1: Heap nach dem Auffüllen und nach dem Entfernen

b)

Das Interface `GenericHeap`:

```
public interface GenericHeap<T extends Comparable<T>> {
    public T getMax();
    public void insert(T element);
    public int getSize();
}
```

c)

Die Klasse `Heap`:

```
package ch.hslu.ad.sw09.generic;

import java.util.ArrayList;
import java.util.List;

public class Heap<T extends Comparable<T>> implements GenericHeap<T> {

    private final List<T> heap = new ArrayList<>();

    @Override
    public T getMax() {
        if (heap.isEmpty()) {
            throw new IllegalStateException("Heap is empty.");
        }
        T max = heap.get(0);
        heap.set(0, heap.get(heap.size() - 1));
        heap.remove(heap.size() - 1);
        sink();
        return max;
    }

    @Override
    public void insert(T element) {
        heap.add(element);
        int newElementIndex = heap.size() - 1;
        raise(newElementIndex);
    }

    @Override
    public int getSize() {
```

```
    return heap.size();
}

private void sink() {
    final int size = heap.size();
    boolean sunk = false;
    int l = 1, f = 0, r = 2;
    while (!sunk && (l < size || r < size)) {
        T father = heap.get(f);
        T left = l < size ? heap.get(l) : father;
        T right = r < size ? heap.get(r) : father;
        if (father.compareTo(left) < 0 || father.compareTo(right) < 0) {
            int biggerChildIndex = left.compareTo(right) > 0 ? l : r;
            swap(f, biggerChildIndex);
            f = biggerChildIndex;
            l = (2 * f) + 1;
            r = 2 * (f + 1);
        } else {
            sunk = true;
        }
    }
}

private void raise(int i) {
    boolean risen = false;
    while (!risen) {
        int father = (i - 1) / 2;
        if (heap.get(i).compareTo(heap.get(father)) > 0) {
            swap(i, father);
            i = father;
            if (i == 0) {
                risen = true;
            }
        } else {
            risen = true;
        }
    }
}

private void swap(int a, int b) {
    T tmp = heap.get(a);
    heap.set(a, heap.get(b));
    heap.set(b, tmp);
}
```

```
}
```

d)

Aus der Testklasse `GenericSortTest`:

```
private static final int TEST_SIZE = 100;
private static final int STRING_LENGTH = 30;

private Integer integers[] = new Integer[TEST_SIZE];
private Double doubles[] = new Double[TEST_SIZE];
private String strings[] = new String[TEST_SIZE];

@Before
public void initializeRandomArray() {
    Random random = new Random(System.currentTimeMillis());
    for (int i = 0; i < TEST_SIZE; i++) {
        integers[i] = random.nextInt();
        doubles[i] = random.nextDouble();
        strings[i] = randomString(random);
    }
}

@Test
public void testIntegerHeapSort() {
    GenericSort.heapSort(integers);
    SortingUtils.isSorted(Arrays.asList(integers), true);
}

@Test
public void testDoubleHeapSort() {
    GenericSort.heapSort(doubles);
    SortingUtils.isSorted(Arrays.asList(doubles), true);
}

@Test
public void testStringHeapSort() {
    GenericSort.heapSort(strings);
    SortingUtils.isSorted(Arrays.asList(strings), true);
}

private String randomString(Random random) {
    StringBuilder randomString = new StringBuilder();
```

```
    for (int i = 0; i < STRING_LENGTH; i++) {
        randomString.append((char) (random.nextInt('Z' - 'A' + 1) + 'A'));
    }
    return randomString.toString();
}
```

e)

Ich habe der `GenericSort`-Klasse folgende Methode (für eine *aufsteigende* Sortierung) als API für den Heap hinzugefügt:

```
public static <T extends Comparable<T>> void heapSort(T items[]) {
    GenericHeap<T> heap = new Heap<>();
    for (int i = 0; i < items.length; i++) {
        heap.insert(items[i]);
    }
    for (int i = items.length - 1; i >= 0; i--) {
        items[i] = heap.getMax();
    }
}
```

f)

-

5 Übersicht Sortieralgorithmen

a)

b)

siehe a)

6 Optional: Quicksort – generisch programmiert

a) und b)

```
public static <T extends Comparable<T>> void quickSort(T[] data) {
    quickSort(data, 0, data.length - 1);
}
```

```
public static <T extends Comparable<T>> void quickSort(T[] data,
    int left, int right) {
    if (right - left == 0) {
        return;
    }
    int up = left;
    int down = right - 1;
    T t = data[right];
    do {
        while (data[up].compareTo(t) < 0) {
            up++;
        }
        while (data[down].compareTo(t) >= 0 && down > up) {
            down--;
        }
        if (up >= down) {
            break;
        }
        swap(data, up, down);
    } while (true);
    swap(data, up, right);
    if (left < up - 1) {
        quickSort(data, left, up - 1);
    }
    if (right > up + 1) {
        quickSort(data, up + 1, right);
    }
}
```

c) und d)

Der ergänzende Code, um das *Median of Three*-Verfahren umzusetzen:

```
int middle = left + ((right - left) / 2);
T l = data[left];
T m = data[middle];
T r = data[right];
T t = r;
int tIndex = right;
if (l.compareTo(r) > 0 && l.compareTo(m) > 0) {
    t = l;
    tIndex = left;
```

```
} else if (r.compareTo(l) > 0 && r.compareTo(m) > 0) {
    t = m;
    tIndex = middle;
}
swap(data, tIndex, right);
```

Das ermittelte Vergleichselement wird nach rechts verschoben (letzte Zeile).

Dieser Testfall testet Heapsort, Quicksort und Median-of-Three-Quicksort:

```
public class GenericSortBenchmark {

    private static final int TEST_SIZES[] = new int[] {
        100_000, 200_000, 500_000,
        1_000_000, 2_000_000, 5_000_000, };

    private Random random;

    @Before
    public void initialize() {
        random = new Random(System.currentTimeMillis());
    }

    @Test
    public void benchmarkHeapVsQuickSort() {
        System.out.println("   Items    HS    QS  QS Mo3");
        System.out.println("-------- ----- ----- -------");
        for (int testSize : TEST_SIZES) {
            Integer hItems[] = randomIntegerArray(testSize);
            Integer qItems[] = hItems.clone();
            Integer qItemsMo3[] = hItems.clone();

            long hStart = System.currentTimeMillis();
            GenericSort.heapSort(hItems);
            long hEnd = System.currentTimeMillis();
            Assert.assertTrue(sorted(hItems));

            long qStart = System.currentTimeMillis();
            GenericSort.quickSort(qItems);
            long qEnd = System.currentTimeMillis();
            Assert.assertTrue(sorted(qItems));

            long qStartMo3 = System.currentTimeMillis();
            GenericSort.quickSortMedianOfThree(qItemsMo3);
            long qEndMo3 = System.currentTimeMillis();
```

```
            Assert.assertTrue(sorted(qItemsMo3));

            System.out.printf("%8d %5d %5d %7d\n", testSize,
                hEnd - hStart, qEnd - qStart, qEndMo3 - qStartMo3);
        }
    }

    private <T extends Comparable<T>> boolean sorted(T items[]) {
        T last = items[0], current;
        for (int i = 1; i < items.length; i++) {
            current = items[i];
            if (current.compareTo(last) < 0) {
                return false;
            }
            last = current;
        }
        return true;
    }

    private Integer[] randomIntegerArray(int size) {
        Integer integers[] = new Integer[size];
        for (int i = 0; i < size; i++) {
            integers[i] = random.nextInt(Integer.MAX_VALUE);
        }
        return integers;
    }
}
```

Die ermittelten Laufzeiten (in Millisekunden):

Items	HS	QS	QS Mo3
100000	117	55	76
200000	114	76	42
500000	308	115	114
1000000	620	265	273
2000000	1383	563	571
5000000	4261	1580	1634