

# AlgoDat — Ausarbeitung zum Übungsblatt 3

YRUCREM

Dienstag, 23. April 2002

VERSION 1.0

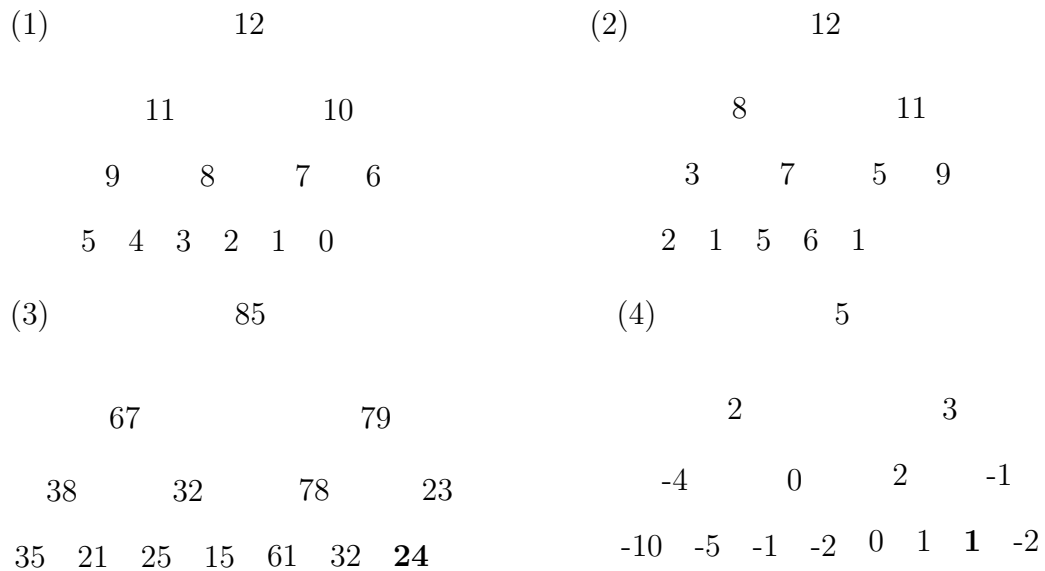
GMP und EXE haben auf einige Tipp- und Rechenfehler hingewiesen.

Von SKYTALE und CHRIS kamen Korrekturen für den Quellcode von Aufgabe 8.

Der L<sup>A</sup>T<sub>E</sub>X-Source, sowie die neueste Version dieses Dokuments sind unter [yrucrem.dr.ag/studium/algodat\\_1/uebungsblatt\\_3.html](http://yrucrem.dr.ag/studium/algodat_1/uebungsblatt_3.html) erhältlich.

## Aufgabe 1

Bei diesen vier Zahlenfolgen ist gefragt, ob es sich um gültige Heaps handelt.



Die fettgeschriebenen Elemente sind jene, die die Heap-Eigenschaft verletzen (die Elemente unter einem Knoten dürfen nur kleiner oder gleich sein).

## Aufgabe 2

Gegeben ist die Zahlenfolge  $A = (28, 58, 6, 12, 67, 17, 36, 19, 55, 58, 38, 21, 18, 55, 32, 43)$ . Zuerst ist ein Heap zu erstellen und dieser ist dann mit Heap-Sort zu sortieren. Der Baum ist nach jedem Aufruf von `Versickere()` anzugeben.

So sieht die Zahlenfolge als Heap angeschrieben aus:

```

                28
            58          6
        12      67      17      36
    19  55  58  38  21  18  55  32
43
```

`Erstelle_Heap()` beginnt nun damit das letzte Element zu versickern, das noch kein Blatt ist (unter dem sich also noch andere Elemente befinden). Das ist im Array das  $\lfloor \frac{n}{2} \rfloor$ -te Element. Dann arbeitet er sich bis zum ersten Element nach vor und versickert alle Elemente auf ihre richtige Position im Heap.

Hier ist nun jeweils der Heap nach den einzelnen Versickerungen.

Wir versickern die 19:

28

58 6

12 67 17 36

43 55 58 38 21 18 55 32

19

Als nächstes die 17:

28

58 6

12 67 21 55

43 55 58 38 17 18 36 32

19

Dann die 12:

28

58 6

55 67 21 55

43 12 58 38 17 18 36 32

19

Schließlich kommen noch die 58:

28

67 55

55 58 21 36

43 12 58 38 17 18 6 32

19

Dann die 36:

28

58 6

12 67 17 55

43 55 58 38 21 18 36 32

19

Danach die 67:

28

58 6

12 67 21 55

43 55 58 38 17 18 36 32

19

Und die 6:

28

58 55

55 67 21 36

43 12 58 38 17 18 6 32

19

Und die 28:

67

58 55

55 58 21 36

43 12 28 38 17 18 6 32

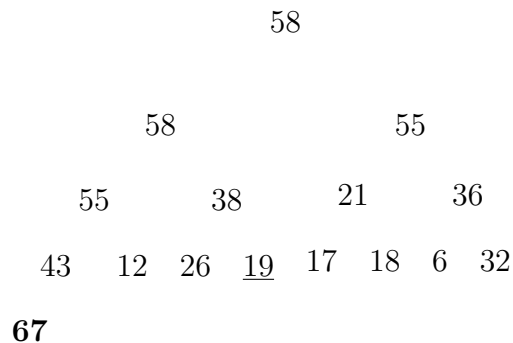
19

Damit wäre unser Heap fertig.

Was Heap-Sort nun macht, ist das letzte Element im Array (das ist das Element ganz rechts in der letzten Ebene) an den Anfang des Heaps zu verschieben. Dadurch kommt das Element, das ganz am Anfang steht (das größte Element der Zahlenfolge) ganz ans Ende, also an seine endgültige Position. Das Element, welches an die Spitze verschoben wurde, muss nun wieder versickert werden um wieder einen gültigen Heap zu erhalten. Dabei wird es maximal bis zu der Stelle versickert, die eins *vor* seiner originalen Position ist (i.e.: alle bereits sortierten Elemente werden in Ruhe gelassen). Im nächsten Schleifendurchlauf, wird das nächste Element an die Spitze verschoben und das Ganze beginnt von vorne.

Hier nun die Bäume nachdem ein Element an die Spitze verschoben und wieder versickert wurde. Die fetten Zahlen sind bereits an der richtigen position und werden vom Algorithmus nicht mehr angerührt. Die unterstrichenen Elemente, sind jene die an die Spitze verschoben und wieder versickert wurden.

Nach dem 1. Aufruf von `Versickere()`:



Nach dem 2. Aufruf von `Versickere()`:

```

                    58
                55          55
            43      38      21      36
        32  12  26  19  17  18  6  58
67
```

Nach dem 3. Aufruf von `Versickere()`:

```

                    55
                43          55
            32      38      21      36
        6  12  26  19  17  18  58  58
67
```

Nach dem 4. Aufruf von `Versickere()`:

```

                    55
                43          36
            32      38      21      18
        6  12  26  19  17  55  58  58
67
```

Nach dem 5. Aufruf von Versickere():

43  
38 36  
32 26 21 18  
6 12 17 19 55 55 58 58  
**67**

Nach dem 6. Aufruf von Versickere():

38  
32 36  
19 26 21 18  
6 12 17 **43** 55 55 58 58  
**67**

Nach dem 7. Aufruf von Versickere():

36  
32 21  
19 26 17 18  
6 12 **38** **43** 55 55 58 58  
**67**

Nach dem 8. Aufruf von Versickere():

32  
26 21  
19 12 17 18  
6 **36 38 43 55 55 58 58**  
**67**

Nach dem 9. Aufruf von Versickere():

26  
19 21  
6 12 17 18  
**32 36 38 43 55 55 58 58**  
**67**

Nach dem 10. Aufruf von Versickere():

21  
19 18  
6 12 17 **26**  
**32 36 38 43 55 55 58 58**  
**67**

Nach dem 11. Aufruf von Versickere():

19

17 18  
6 12 21 26  
32 36 38 43 55 55 58 58  
67

Nach dem 12. Aufruf von Versickere():

18

17 12  
6 19 21 26  
32 36 38 43 55 55 58 58  
67

Nach dem 13. Aufruf von Versickere():

17

6 12  
18 19 21 26  
32 36 38 43 55 55 58 58  
67

Nach dem 14. Aufruf von `Versickere()`:

```

                12
            6                17
        18    19    21    26
    32  36  38  43  55  55  58  58
67
```

Nach dem 15. Aufruf von `Versickere()`:

```

                6
            12                17
        18    19    21    26
    32  36  38  43  55  55  58  58
67
```

Damit hätten wir die Zahlenfolge sortiert.

### Aufgabe 3

Am besten ist es, wenn in der Zahlenfolge nur gleiche Zahlen vorkommen. Dann haben wir nur die Vertauschungen an die Spitze, aber beim Versickern werden keine Daten mehr bewegt.

Im Allgemeinen ist es sehr günstig, wenn die Eingabefolge bereits ein gültiger Heap ist, dann hat die Funktion `Erstelle_Heap()` keine Bewegungen zu machen.

### Aufgabe 4

`Erstelle_Heap()` beginnt beim letzten Element, das noch ein Knoten ist (das also noch Kinder hat). Zuerst vergleicht es seine beiden Kinder untereinander um festzustellen, welches der beiden das größere ist. Dann wird das

Element selbst mit dem größten seiner Kinder verglichen um festzustellen ob versickert werden muss. Da aber alle Elemente gleich groß sind kommen keine Versickerungen vor und dieser Knoten ist fertig. Wir haben also für jeden Knoten (außer den Blättern, die ja keine Kinder haben) 2 Vergleiche. Wie viele Elemente sind nun Blätter bzw. Knoten mit Kindern?

Wir versuchen zuerst zu zeigen, dass bei  $2^n - 1$  Elementen das  $n$  die Anzahl der Ebenen des Heaps angibt. Und das bei einem solchen Heap in jeder Ebene die maximale Anzahl an Elementen ist, also doppelt so viele als in der Ebene darüber.

Wir gehen also davon aus, dass wir einen Heap mit  $n$  Ebenen haben, wobei wir bei 0 anfangen zu zählen (ie. bei der ersten Ebene ist  $n = 0$ ). Da wir weiters davon ausgehen, dass in jeder Ebene doppelt so viele Elemente sind als in der Ebene darüber, können wir das folgendermaßen anschreiben:

$$\sum_{i=0}^{n-1} 2^i$$

Das  $i$  beginnt bei 0, weil wir ja die Ebenen ab 0 durchnummerieren und da wir insgesamt  $n$  Elemente haben wollen müssen wir nur bis  $n - 1$  gehen. Das  $2^i$  in der Summe sagt jetzt, dass jedesmal doppelt so viele Elemente dazukommen wie beim letzten Schritt. Wir versuchen jetzt durch Induktion zu beweisen, dass gilt:

$$\sum_{i=0}^{n-1} 2^i = 2^n - 1$$

Wir brauchen einen Induktionsanfang und setzen  $n = 1$ :

$$\sum_{i=0}^{1-1} 2^i = 2^0 = 1 = 2^1 - 1$$

Somit haben wir gezeigt, dass die Behauptung für die ersten  $n$  gilt. Nun müssen wir zeigen, dass es auch für  $n + 1$  gilt:

$$\sum_{i=0}^{n-1+1} 2^i = 2^{n+1} - 1$$

Nun formen wir ein bisschen um:

$$\sum_{i=0}^{n-1+1} 2^i = \sum_{i=0}^{n-1} 2^i + 2^n$$

Wir wissen bereits das für die Summe von 0 bis  $n - 1$  unsere Formel gilt, also setzen wir ein und führen ein paar letzte Umformungen durch:

$$2^n - 1 + 2^n = 2^n + 2^n - 1 = 2 * 2^n - 1 = 2^{n+1} - 1$$

Q.E.D

Wir wissen also nun, das bei  $2^n - 1$  Elementen, das  $n$  die Anzahl der Ebenen angibt, wobei wir von 0 bis  $n - 1$  zählen. Weiters wissen wir, dass in jeder Ebene doppelt so viele Elemente sind als in der darüber. Wir wissen also, das in der letzten Ebene (das sind die Blätter und deren Anzahl wollten wir ja wissen)  $2^{n-1}$  Elemente sind.

Jedes Element das kein Blatt ist verursacht 2 Vergleiche. Die Anzahl der Elemente insgesamt ist  $2^n - 1$ , die Anzahl der Blätter ist  $2^{n-1}$ , wir kommen also auf

$$2^n - 1 - 2^{n-1} = 2^{n-1} - 1$$

Elemente die je zwei Vergleiche verursachen, also

$$(2^{n-1} - 1) * 2 = 2^n - 2$$

## Aufgabe 5

Die Anzahl der Bewegungen ist recht einfach. Wir haben  $2^n - 1$  Elemente und jedes davon wird an die Spitze des Heaps verschoben. Da nur gleichgroße Zahlen vorkommen, versickern diese nicht wieder nach unten. Ein Element wird übrigens nicht mehr verschoben, nämlich, wenn das Element, das wir verschieben wollen, schon an der Spitze steht. Wir haben also

$$2^n - 2$$

Vertauschungen.

Zu den Vergleichen. Solange die zweite Ebene noch nicht angetastet wurde haben wir nach jeder Vertauschung genau 2 Vergleiche um festzustellen, dass nicht versickert werden muss (ein Element wird an die Spitze des Heaps vertauscht, danach werden die zwei Kinder der Wurzel verglichen um festzustellen welches das größere ist, diese wird dann mit der Wurzel selbst verglichen). Also haben wir zwei Vergleiche für jedes Element ab der dritten Ebene. Insgesamt haben wir  $2^n - 1$  Elemente, drei davon sind in den ersten

beiden Ebenen, also sind in den restlichen ebenen  $2^n - 4$  Elemente. Und für jedes dieser Elemente haben wir zwei Vergleiche, also

$$(2^n - 4) * 2 = 2^{n+1} - 8$$

Jetzt sind nur noch die ersten beiden Ebenen abzuarbeiten. Wir tauschen das letzte Element an die Spitze. Für den Algorithmus hat die Wurzel jetzt nur noch ein Kind, wir haben also nur noch einen Vergleich. Nachdem wir das letzte zu bewegendes Element an die Spitze getauscht haben, gibt es überhaupt keinen Vergleich mehr, weil die Wurzel keine Kinder mehr hat. Zu der Obigen Anzahl wird also nur noch ein Vergleich dazugezählt und wir erhalten

$$2^{n+1} - 7$$

Vergleiche.

## Aufgabe 6

```
[55 19 48 54 94 68 88 99 81 22 97 74 44 18 11 65]
partition(feld, 1, 16, 65)
[55 19 48 54 11 18 44 22] 65 99 97 74 88 68 94 81
partition(feld, 1, 8, 22)
[18 19 11] 22 48 55 44 54 65 99 97 74 88 68 94 81
partition(feld, 1, 3, 11)
11 [19 18] 22 48 55 44 54 65 99 97 74 88 68 94 81
partition(feld, 2, 3, 18)
11 18 19 22 [48 55 44 54] 65 99 97 74 88 68 94 81
partition(feld, 5, 8, 54)
11 18 19 22 [48 44] 54 55 65 99 97 74 88 68 94 81
partition(feld, 5, 6, 44)
11 18 19 22 44 48 54 55 65 [99 97 74 88 68 94 81]
partition(feld, 10, 16, 81)
11 18 19 22 44 48 54 55 65 [68 74] 81 88 99 94 97
partition(feld, 10, 11, 74)
11 18 19 22 44 48 54 55 65 68 74 81 [88 99 94 97]
partition(feld, 13, 16, 97)
11 18 19 22 44 48 54 55 65 68 74 81 [88 94] 97 99
partition(feld, 13, 14, 84)
11 18 19 22 44 48 54 55 65 68 74 81 88 94 97 99
```

## Aufgabe 7

Quicksort ist nicht stabil, weil gleich große Elemente aneinander vorbei getauscht werden können. Z.B.:  $7_a, 2, 3, 7_b, 4, 6$  wird zu  $4, 2, 3, 7_b, 7_a, 6$ . Das ist *kein* kompletter aufruf von `partition()`, aber man sieht, dass die Position der beiden 7er zueinander verändert wurde.

## Aufgabe 8

Hatte nicht die Zeit, L<sup>A</sup>T<sub>E</sub>X dazu zu bringen, die Tabs nicht zu ignorieren.

```
quicksort(A, l, r)
{
    falls (r - l + 1) <= k selectionsort(A, l, r);
    sonst falls (l < r) {
        x = A[r].key;
        p = partition(A, l, r, x);
        quicksort(A, l, p - 1);
        quicksort(A, p + 1, r);
    }
}
```

## Aufgabe 9

Wenn das größte Element an der Stelle `A[1]` steht, gibt es ein Problem. Zuerst wird nämlich das kleinste Element mit dem an der Stelle `A[1]` vertauscht, und dann wird dieses Element (weil der Algorithmus glaubt, dass dort immer noch das größte Element steht) mit `A[r]` vertauscht. Zum Beispiel:  $5, 4, 1, 2, 3$  wird zu  $1, 4, 5, 2, 3$  und das zu  $3, 4, 5, 2, 1$ . Eigentlich sollte aber das kleinste ganz links und das größte ganz rechts stehen.

## Aufgabe 10

Man darf nicht zuerst beide Positionen bestimmen und dann beide vertauschen, sondern man muss zuerst die Position des kleinsten Elementes bestimmen, es an den linken Rand tauschen, dann die Position des größten Elementes bestimmen und es an den rechten Rand tauschen. Man muss also nur die 4. Zeile im Pseudocode (`max = ...`) zwischen die beiden `Vertausche`-Zeilen schieben.