

Name	Übungsgruppe	1	2	3	4	5	6
Theodor Bajusz	5	x	x	x	x	x	x
Valerij Dobler	13	x	x	x	x	x	x
Matz Radloff	6	x	x	x	x	x	x
Robin Wannags	5	x	x	x	x	x	x

Übung 1.

- (a) Ist ein aufsteigend sortiertes Array ein min-Heap oder ein max-Heap? Begründen Sie Ihre Antwort. (1 Punkt)

Beweis. Sei $I(a_i)$ der Index vom Element a_i im Array a .

In einer aufsteigend sortierten Folge $a = \langle a_1, a_2, \dots, a_n \rangle$ gilt für alle verschiedene $a_i, a_j \in a$ und $i < j : a_i \leq a_j$. Die Kinder von einem Element a_i in einem Heap werden immer derart ermittelt: linkes Kind $I(a_{i_l}) = 2 \cdot I(a_i)$ und rechtes Kind $I(a_{i_r}) = 2 \cdot I(a_i) + 1$. Da $a_i \leq a_{i_l} \wedge a_i \leq a_{i_r}$ gilt, ist die min-Heap-Eigenschaft für alle $a_i \in a$ erfüllt. Es handelt sich also bei aufsteigend sortierten Arrays immer um einen min-Heaps. \square

- (b) Zeigen Sie, dass ein Heap mit n Elementen eine Höhe $h \in \Theta(\log n)$ hat. (1 Punkt)

Beweis. Da Heaps links-vollständige Binärbäume sind, hat der Knoten an der Stelle n die größte Tiefe. Wenn wir dessen Tiefe ermitteln möchten, zählen wir die Anzahl h der Vorfahren vom Knoten n . Für den Knoten n können wir auch $2^m - 1 + k$ schreiben, wobei m größtmöglich sein soll. Somit hat der Heap eine Höhe von $m - 1$ und k Blätter in der größten Tiefe des Baums. Die Höhe des Baums ermittelt sich nun aus dem längsten einfachen Weg von der Wurzel zu einem der k Blätter. Dieser Weg muss die Länge m besitzen. So wie wir m definiert haben, sieht man leicht, dass $m = \lfloor \log n \rfloor$ gilt. \square

- (c) Zeigen Sie, dass ein Heap der Größe n maximal $\lceil \frac{n}{2^{h+1}} \rceil$ Knoten der Höhe h hat. (2 Punkte)

Hierzu brauchen wir folgendes Lemma:

Behauptung: Die Indizes der Blätter in einem n -elementigen Heap befinden sich im Intervall $[\lfloor n/2 \rfloor + 1; n]$.

Beweis. Wir betrachten das linke Kind vom Knoten an der Stelle $\lfloor n/2 \rfloor + 1$.

$$\begin{aligned}
 \text{Left}(\lfloor n/2 \rfloor + 1) &= 2(\lfloor n/2 \rfloor + 1) \\
 &\geq 2(n/2 - 1) + 2 \\
 &= n - 2 + 2 \\
 &= n
 \end{aligned}$$

Da der Index vom linken Kind größer als die Anzahl der Elemente im Heap ist, hat der Knoten keine Kinder und ist demnach ein Blatt. Das selbe gilt für alle Knoten mit größerem Index. \square

Behauptung: Ein Heap der Größe n maximal hat $\lceil \frac{n}{2^{h+1}} \rceil$ Knoten der Höhe h .

Beweis. Aus dem obigen Lemma wissen wir, dass die zweite Hälfte (inkl. des mittleren Elements, wenn n ungerade ist) des Heaps aus Blättern besteht. Also gilt für jeden Heap mit n , dass dieser $\lfloor n/2 \rfloor$ Blätter hat. Wir führen den Beweis über die Höhe. Sei n_h die Anzahl der Knoten auf Höhe h . Der Basisfall gilt, denn $n_0 = \lfloor n/2 \rfloor$ ist exakt die Anzahl an Blättern in einem Heap der Größe n .

Nehmen wir nun an, dass es für die Höhe $h - 1$ gilt. Nun beweisen wir, dass es auch für die Höhe h gilt.


Falls n_{h-1} gerade ist, dann hat jeder Knoten auf Höhe h genau zwei Kinder, daraus folgt $n_h = n_{h-1}/2 = \lfloor n_h - 1/2 \rfloor$. Wenn n_{h-1} ungerade ist, hat ein Knoten auf Höhe h nur ein Kind die restlichen aber zwei Kinder, daraus folgern wir $n_h = \lfloor n_{h-1}/2 \rfloor + 1 = \lceil n_{h-1}/2 \rceil$. Genauer folgt daraus

$$\begin{aligned}
 n_h &= \lceil n_h - 1/2 \rceil \\
 &\leq \left\lceil \frac{1}{2} \cdot \left\lceil \frac{n}{2^{(h-1)+1}} \right\rceil \right\rceil \\
 &= \left\lceil \frac{1}{2} \cdot \left\lceil \frac{n}{2^h} \right\rceil \right\rceil \\
 &= \left\lceil \frac{n}{2^{h+1}} \right\rceil
 \end{aligned}$$

also gilt es auch für die Höhe h . \square

- (d) Zeigen Sie, dass in einem Heap die Anzahl der inneren Knoten um eins kleiner oder gleich der Anzahl an Blättern ist. (2 Punkte)

Beweis. Wir gehen davon aus, dass Knoten in einem Binärbaum mit nur einem Kindknoten auch zu den inneren Knoten gezählt werden.

- **Behauptung:**
Ein nicht leerer Heap mit n Blättern hat n oder $n - 1$ innere Knoten k .
- **Induktionsanfang:** $k = 1$
Der Baum hat nur den Wurzelknoten, welcher keine Kinder hat und demnach ein Blatt ist. Und der Baum hat null innere Knoten. ✓
- **Induktionsvoraussetzung:** Für eine beliebige aber feste Anzahl an Blättern n hat der Heap n oder $n - 1$ innere Knoten k .
- **Induktionsschritt:** Wenn man einen neuen Knoten in den Baum anhängen möchte, gibt es zwei Möglichkeiten
 - (i) Man hängt den Knoten an ein Blatt, dann wird das bisherige Blatt zu einem inneren Knoten. Dadurch wächst jeweils die Anzahl der Blätter und der inneren Knoten um 1 und die Bedingung gilt weiterhin. $\Rightarrow n + 1 = k + 1$ ✓
 - (ii) Man hängt den Knoten an einen inneren Knoten, dadurch verändert sich die Anzahl der inneren Knoten nicht, aber es kommt ein neues Blatt hinzu. $\Rightarrow n + 1 = k$ ✓ 
 Da wir einen Binärbaum haben, kann man an den inneren Knoten keine weiteren Blätter mehr anhängen.



□

Übung 2.

Zeigen Sie, dass die Laufzeit von BuildHeap aus der Vorlesung in $O(n)$ liegt. Sie können Voraussetzen, dass die Anzahl der Knoten auf Höhe h für einen Heap der Größe n durch $\lceil \frac{n}{2^{h+1}} \rceil$ beschränkt ist. (4 Punkte).

Beweis. Wir können eine einfache obere Schranke für die Laufzeit von Build-Max-Heap wie folgt berechnen. Jeder Aufruf von Max-Heapify kostet Zeit $O(\log n)$, und

Build-Max-Heap macht $O(n)$ solcher Aufrufe. Somit beläuft sich die Gesamtlaufzeit auf $O(N \log n)$. Diese obere Schranke ist zwar korrekt, aber nicht asymptotisch scharf.

Wir können eine schärfere Schranke herleiten, wenn wir ausnutzen, dass die Laufzeit von Max-Heapify auf einem Knoten mit der Höhe des Knotens im Baum variiert und dass die Höhe der meisten Knoten klein ist. Unsere strengere Analyse beruht auf den Eigenschaften, dass ein n -elementiger Heap die Höhe $\lfloor \log n \rfloor$ hat und dass es höchstens $\lceil \frac{n}{2^{h+1}} \rceil$ Knoten der Höhe h gibt.

Die Zeit, die von der Prozedur Max-Heapify benötigt wird, wenn sie auf einen Knoten der Höhe h angewendet wird, ist $O(h)$. Daher können wir für die Gesamtkosten von Build-Max-Heap die obere Schranke

$$\sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O \left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \right)$$

angeben. Die letzte Summe kann durch Substitution von x durch $1/2$ in der Formel

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$$

für $|x| < 1$ ausgewertet werden, was zu

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$$

führt. Damit kann die Laufzeit von Build-Max-Heap durch

$$O \left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \right) = O \left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O(2n) = O(n)$$

abgeschätzt werden. Wir können also einen Max-Heap in linearer Zeit aus einem ungeordneten Feld konstruieren. □



Übung 3.

Die Wichtel müssen 5 mit Geschenken vollgestopfte Säcke dem Gewicht nach auf den Schlitten des Weihnachtsmanns verteilen. Dabei steht den Wichteln eine Waage zur Verfügung, mit der sie jeweils für zwei Säcke bestimmen können, ob der erste Sack leichter als der zweite Sack ist oder nicht. Da die Säcke sehr groß sind und die Zeit knapp ist, weigern die Wichtel sich die Waage mehr als 7 mal zu benutzen. Beschreiben Sie in Pseudocode einen Algorithmus an, der für die Wichtel die Säcke nach Gewicht sortiert, sodass die Waage dabei nicht mehr als 7 mal benutzt werden muss. (3 Punkte)

```

1 sort(arr) { // arr = [a, b, c, d, e]
2   // Prepare a,b,c,d
3   if (arr[1] < arr[0])
4     swap(0, 1, arr); // => a < b
5   if (arr[3] < arr[2])
6     swap(2, 3, arr); // => c < d
7   if (arr[3] < arr[1]) {
8     swap(0, 2, arr);
9     swap(1, 3, arr); // => b < d
10  }
11  // Results in:
12  // a<b<d && c<d
13
14  // Pinpoint e into [a,b,d] (c is kept in the end for later)
15  if (arr[4] < arr[1]) { // e < b
16    if (arr[4] < arr[0]) { // e < a
17      // e<a<b<d
18      swap(0, 4, arr); // 0,1,2,3,4 -> 4,1,2,3,0
19      swap(1, 4, arr); // 4,1,2,3,0 -> 4,0,2,3,1
20      swap(2, 4, arr); // 4,0,2,3,1 -> 4,0,1,3,2
21    } else { // e >= a
22      // a<=e<b<d
23
24      swap(1, 4, arr); // 0,1,2,3,4 -> 0,4,2,3,1
25      swap(2, 4, arr); // 0,4,2,3,1 -> 0,4,1,3,2
26    }
27  } else { // e >= b
28    // a<b<d
29
30    if (arr[4] < arr[3]) { // e < d
31      // a<b<=e<d

```

```

32      swap(2, 4, arr); // 0,1,2,3,4 -> 0,1,4,3,2
33    } else { // e >= d
34      // a<b<d<=e
35      swap(2, 4, arr); // 0,1,2,3,4 -> 0,1,4,3,2
36      swap(2, 3, arr); // 0,1,4,3,2 -> 0,1,3,4,2
37    }
38  }
39  // The reordered result is now a<=b<=d<=e<=c
40
41  // Squeeze in c into [a,b,d]
42  if (arr[4] < arr[1]) { // c < b
43    if (arr[4] < arr[0]) { // c < a
44      // c<=a<=b<=d<=e
45      swap(0, 4, arr); // 0,1,3,4,2 -> 2,1,3,4,0
46      swap(1, 4, arr); // 2,1,3,4,0 -> 2,0,3,4,1
47      swap(2, 4, arr); // 2,0,3,4,1 -> 2,0,1,4,3
48      swap(3, 4, arr); // 2,0,1,4,3 -> 2,0,1,3,4
49    } else { // c >= a
50      // a<=c<=b<=d<=e
51      swap(1, 4, arr); // 0,1,3,4,2 -> 0,2,3,4,1
52      swap(2, 4, arr); // 0,2,3,4,1 -> 0,2,1,4,3
53      swap(3, 4, arr); // 0,2,1,4,3 -> 0,2,1,3,4
54    }
55  } else { // c >= b
56    if (arr[4] < arr[2]) { // c < d
57      // a<=b<=c<=d<=e
58      swap(2, 4, arr); // 0,1,3,4,2 -> 0,1,2,4,3
59      swap(3, 4, arr); // 0,1,2,4,3 -> 0,1,2,3,4
60    } else { // c >= d
61      // a<=b<=d<=c<=e
62      swap(3, 4, arr); // 0,1,3,4,2 -> 0,1,3,2,4
63    }
64  }
65  }
66}
67
68 swap(i, j, arr){
69   tmp = arr[i];
70   arr[i] = arr[j];
71   arr[j] = arr[tmp];
72}

```



Übung 4.

Betrachten Sie ein Array $A[1, \dots, n]$, dass aus n ganzen Zahlen besteht. Für jede Zahl im Array gilt $0 \leq A[i] \leq k$ für eine Zahl $k > 0$ und $i \in 1, \dots, n$. Das Array A kann so verarbeitet werden, dass ein Algorithmus in konstanter Zeit für ein Intervall $[a..b]$ bestimmen kann, wie viele Zahlen aus A in dem Intervall enthalten sind.

- (a) Beschreiben Sie einen Algorithmus der A in $\Theta(n+k)$ verarbeitet und für eine verarbeitete Eingabe in $O(1)$ ausgibt, wie viele Zahlen aus A in ein Intervall $[a..b]$ fallen. (4 Punkte)
- (b) Zeigen Sie, dass Ihr Algorithmus die Laufzeitschranken einhält. (1 Punkt)

Zu Aufgabe (4a)

```

1 int[] prepareArray(int[] a, int k){
2     int t = int[k + 2];
3
4     for(int i: a){
5         t[i]++;
6     }
7     for(int i=k-1; i >= 0; i--){
8         t[i] = t[i] + t[i + 1];
9     }
10
11     return t;
12 }
13
14 int count(int a, int b, int[] arr){
15     a = max(a, 0);
16     b = min(b + 1, arr.length - 1);
17
18     return arr[a] - arr[b];
19 }

```

Seien k, a, b wie in der Aufgabenstellung beschrieben.

Wenn man ein unsortiertes Array a hat, kann man ein vorbereitetes Array p mit dem Aufruf $p = \text{prepareArray}(a, k)$ erzeugen. Durch Eingabe eines beliebigen Intervalls $I = [a, b]$ kann man mit Hilfe von $\text{count}(a, b, p)$ die Anzahl der Zahlen, welche in I liegen, bekommen.


Behauptung: Unser Algorithmus hält die Laufzeitschranken aus Aufgabenteil (a) ein.

Beweis. In den Zeilen 4-6 durchlaufen wir das Array A , also alle n Elemente $\Rightarrow \Theta(n)$

In den Zeilen 7-9 durchlaufen wir k Elemente in unserem Hilfsarray t $\Rightarrow \Theta(k)$

Somit kommen wir auf eine Laufzeit von $\Theta(n+k)$ für die Verarbeitung unseres Arrays.

In den Zeilen 14-18 führen wir in jeweils $O(1)$ zwei Checks durch und returnen anschließend eine Subtraktion, ebenfalls in $O(1)$.

Somit hat unsere zweite Funktion die gewünschte Laufzeit von $O(1)$.  ☐

Übung 5.

- (a) Erklären Sie, wie eine Queue durch zwei Stacks implementiert werden kann. (2 Punkte)

Beweis. Ein *Enqueue* überprüft ob, der Stack $S2$ leer ist, falls nicht, müssen erst alle Elemente sequenziell von $S2$ gepoppt und nach $S1$ gepusht werden. Sobald $S2$ leer ist, pushen wir das zu enqueueende Element auf den Stack $S1$.

Beim *Dequeue* überprüfen wir ob der Stack $S1$ leer ist, falls nicht müssen alle Elemente vom Stack $S1$ sequenziell gepoppt und auf den Stack $S2$ gepusht werden. Sobald $S1$ leer ist, nimmt man das oberste Element aus $S2$ und gibt es als Ergebnis von dem Dequeue Aufruf aus.

Falls beide Stacks leer sind ist ein Dequeue nicht möglich. ☐

- (b) Analysieren Sie die Laufzeit der Queueoperationen aus (a) unter der Annahme, dass die Stack-Operationen Zeit $O(1)$ benötigen. (1 Punkt)

Beweis. *Enqueue* und *Dequeue* laufen in $O(1)$, außer es wurde direkt vor einem *Enqueue* ein *Dequeue* oder direkt vor einem *Dequeue* ein *Enqueue* ausgeführt. In diesen beiden Fällen kommt ein Overhead von $O(n)$ fürs Umschieben der Elemente von einem Stack zum anderen hinzu. ☐



Übung 6.

Bonusaufgabe

Zeigen Sie, dass es keinen Algorithmus A geben kann, der die folgenden drei Eigenschaften gleichzeitig erfüllt. (3 Punkte)

- (a) Der Algorithmus A benutzt wie ein Vergleichssortierer nur Vergleiche, Zuweisungen, Kopierungen, usw.
- (b) Der Algorithmus A entfernt aus einem max-Heap das Maximum und stellt die max-Heap-Eigenschaft wieder her.
- (c) Der Algorithmus A hat Laufzeit $O(1)$.

Beweis. Wir wissen, dass man ein unsortiertes Array mit n Elementen mit einer Laufzeit von $O(n)$ in einen Heap h mit der Build-Max-Heap überführen kann. (siehe Zettel 4, Aufgabe 2)

Angenommen es gibt einen solchen Algorithmus A . Dann führen wir Algorithmus A für alle n Elemente in Heap h aus sukzessiv aus. Dies extrahiert immer das Maximum aus dem Heap und stellt die max-Heap-Eigenschaft wieder her (Eigenschaft (b)). Und das nach Eigenschaft (c) mit einer Laufzeit von $O(1) \implies n \cdot O(1) = O(n)$.

Für die Gesamtlaufzeit ergibt sich dann aus der Laufzeit von Build-Max-Heap plus die Laufzeit vom n -maligen Aufrufen von Algorithmus A

$$O(n) + n \cdot O(1) = O(n) + O(n) = O(2n) = O(n)$$

Wir wissen, dass vergleichsbasierte-Sortieralgorithmen eine theoretisch kleinste obere Schranke von $O(n \log n)$ haben, das bildet einen Widerspruch zur Eigenschaft (a) des Algorithmus A . Also kann es so einen Algorithmus A mit den gegebenen drei Eigenschaften nicht geben. \square

Alternativ kann man aus (a) und (b) mit einer ähnlichen Argumentation einen Widerspruch zur Eigenschaft (c) zeigen.

