Karlsruher Institut für Technologie
Institut für Theoretische Informatik

Prof. Dr. Jörn Müller-Quade
Björn Kaidel, Sebastian Schlag, Sascha Witt

7. Übungsblatt zu Algorithmen I im SoSe 2017

http://crypto.iti.kit.edu/index.php?id=799
{bjoern.kaidel,sascha.witt}@kit.edu

Musterlösungen

Aufgabe 1 (*Binäre Heaps,* $3 + 1 = 4$ *Punkte*)

Betrachten Sie die vier Ziffern 9, 5, 3 und 4.

a) Bilden Sie aus den obigen Ziffern alle möglichen binären Min-Heaps. Stellen Sie dabei die Heaps als implizites Feld dar.

b) Ist $\langle 3, 9, 5, 4\rangle$ ein gültiger binärer Min-Heap für diese Ziffern? Begründen Sie Ihre Antwort!

Musterlösung:

a) Alle möglichen impliziten Heaps sind: $\langle 3, 4, 5, 9\rangle$, $\langle 3, 5, 4, 9\rangle$ und $\langle 3, 4, 9, 5\rangle$.

b) Nein, es handelt sich um keinen gültigen Min-Heap. Die Bedingung $\forall v : \text{parent}(v) \leq v$ ist verletzt, denn $\text{parent}(4)$ wäre hier gleich 9.

Aufgabe 2 (*Radixsort,* 4 *Punkte*)

Führen Sie Least-Significant-Digit (LSD) Radixsort mit Dezimalziffern (Radix $K = 10$) auf dem folgenden Array durch. Verwenden Sie dazu als Untermethode die Methode *KSortArray*. Tragen Sie nach jeder Runde das Array in eine der Schablonen ein, markieren Sie die *Bucketgrenzen der Ziffern* durch Trennstriche im Array. Geben Sie zusätzlich jeweils den Zustand des Hilfsarrays c nach Durchlauf der zweiten for-Schleife in der Methode *KSortArray* an (vergleiche Vorlesung 29.05., Folie 17 und 18, $c := [1, 3, 5, 7]$).

Eingabe:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
39	11	23	99	51	34	12	8	71	63	5	44	16	15	80	43

Musterlösung:

Hilfsarray c:

0	1	2	3	4	5	6	7	8	9
1	2	5	6	9	11	13	14	14	15

Ergebnis nach der 1. Runde:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
80	11	51	71	12	23	63	43	34	44	5	15	16	8	39	99

Hilfsarray c:

0	1	2	3	4	5	6	7	8	9
1	3	7	8	10	12	13	14	15	16

Ergebnis nach der 2. Runde:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
5	8	11	12	15	16	23	34	39	43	44	51	63	71	80	99

Aufgabe 3 (*HeapSort,* $2 + 3 = 5$ *Punkte*)

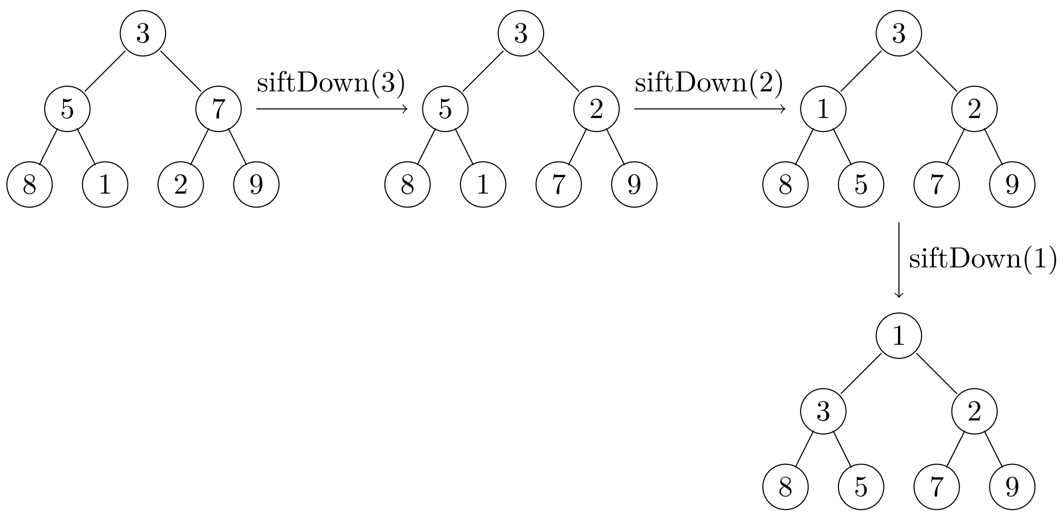
Sortieren Sie das Array $[3, 5, 7, 8, 1, 2, 9]$ absteigend mittels HeapSort. Gehen Sie dazu folgendermaßen vor:

a) Konstruieren Sie zuerst mittels *buildHeapBackwards* einen Min-Heap aus dem Array. Geben Sie dabei jede *siftDown*-Operation an, die zu einer Änderung führt und zeichnen Sie Ihren Heap nach jeder Änderung. Zeichnen Sie die Heaps als binäre Bäume.

b) Führen Sie die restlichen Schritte von HeapSort auf dem Heap, den Sie in a) erzeugt haben, durch. Stellen Sie den Heap als Baum und als Array nach jedem Aufruf von *deleteMin* dar. Markieren Sie im Array auch deutlich, bis zu welchem Index sich der Heap im Array befindet (Hinweis: Dieser Index wird in der Vorlesung vom Heap durch die Variable n verwaltet).

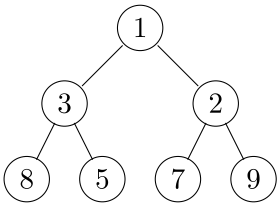
Musterlösung:

a) Die folgenden Heaps zeigen den Ablauf:

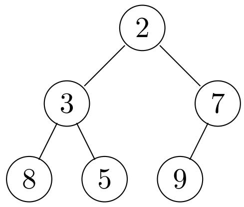


Innerhalb der angegebenen Aufrufe von siftDown werden noch weitere siftDown-Aufrufe durchgeführt, diese ändern aber nichts am Heap.

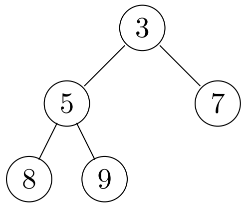
b) Die folgenden Heaps zeigen den Ablauf von HeapSort. Das Symbol „|“ markiert jeweils den Index, bis zu dem sich der Heap im Array befindet.



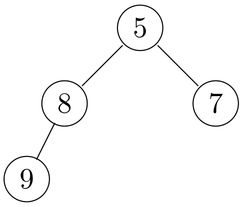
$[1, 3, 2, 8, 5, 7, 9|]$



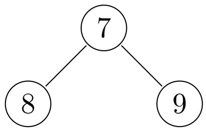
$[2, 3, 7, 8, 5, 9|, 1]$



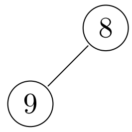
$[3, 5, 7, 8, 9|, 2, 1]$



$[5, 8, 7, 9|, 3, 2, 1]$



$[7, 8, 9|, 5, 3, 2, 1]$



$[8, 9|, 7, 5, 3, 2, 1]$

$[9|, 8, 7, 5, 3, 2, 1]$

Sortiertes Array am Ende: $[9, 8, 7, 5, 3, 2, 1]$.

Aufgabe 4 (*In-place-Duplikaterkennung,* 5 *Punkte*)

Gegeben sei ein Array $A : \text{Array}[0..n-1]$ der Größe n, welches ausschließlich Zahlen aus $\{0, \ldots, n-1\}$ enthält. Geben Sie einen *In-place*-Algorithmus im Pseudocode an, der eine Zahl ausgibt, die doppelt in A vorkommt oder ausgibt, dass alle Zahlen A verschieden sind. Die Laufzeit Ihres Algorithmus soll in $\mathcal{O}(n)$ liegen. Argumentieren Sie kurz, dass Ihr Algorithmus diese Laufzeit erreicht (ein formaler Laufzeitbeweis ist nicht nötig). **Hinweis:** Es ist erlaubt, das ursprüngliche Array zu verändern. Ein *In-place*-Algorithmus verwendet höchstens $O(1)$ zusätzlichen Speicher.

Musterlösung:
Der Pseudocode für einen Algorithmus, der das gewünschte tut, könnte folgendermaßen aussehen:

procedure *duplicates*(A : Array $[0..n-1]$ **of** $\mathbb{N}$)
    for $i = 0$ **to** $n-1$ **do**
        while $A[i] \neq i$
            if $A[A[i]] = A[i]$ **then** print "Duplikat gefunden:", A[i]; **return**
            else swap$(A, i, A[i])$ (vertausche $A[i]$ mit $A[A[i]]$)
    print "keine Duplikate"
return

Der Trick ist, dass wir versuchen, die Zahlen an ihre richtigen Stellen einzuordnen, sodass immer $A[i] = i$ gilt. Um das zu erreichen wird über das Array iteriert und wenn wir einen Eintrag mit $A[i] \neq i$ finden, wird dieser an Stelle $A[A[i]]$ geschoben. Wäre beispielsweise $A[5] = 2$, würden wir den Eintrag in $A[2]$ nach $A[5]$ verschieben und in $A[2]$ die 2 speichern. Bevor wir diesen Tausch durchführen, überprüfen wir aber zuerst, ob in $A[i]$ nicht bereits i gespeichert ist. Wenn ja, haben wir ein Duplikat gefunden und können dieses ausgeben. Dieses Durchtauschen führen wir solange durch, bis in $A[i]$ tatsächlich der Wert i steht oder wir abbrechen, da wir ein Duplikat gefunden haben (da die Zahlen im Array nur aus $\{0, ..., n-1\}$ sind, ist gewährleistet, dass dieses Vorgehen nicht zu einer Endlosschleife führt). Somit verschiebt der Algorithmus das erste Aufkommen einer Zahl i immer an $A[i]$ und sobald die Zahl i nochmal vorkommt, werden wir diese dann als Duplikat erkennen.

Obwohl im Algorithmus eine while-Schleife innerhalb einer for-Schleife geschachtelt ist, ist die Laufzeit in $\mathcal{O}(n)$. Die for-Schleife benötigt insgesamt n Schritte. Die Anzahl aller while-Schleifendurchläufe für alle for-Schleifendurchläufe zusammengelegt liegt auch in $\mathcal{O}(n)$. Grund dafür ist folgender: Betrachtet die while-Schleife eine Stelle i, an der $j \neq i$ steht, wird j an die richtige Stelle $A[j]$ geschoben. Für die Stelle j wird die while-Schleife im weiteren Ablauf des Algorithmus aber nun nicht mehr betreten (da ja jetzt $A[j] = j$ gilt). Mit jedem „Verschieber“ sorgen wir also für eine while-Schleife, die überhaupt nicht betreten wird. Da im Worst-Case nach n Durchläufen einer (oder mehrerer) while-Schleifen jede Zahl an der richtigen Stellen steht (oder wir ein Duplikat gefunden haben), liegt die Anzahl aller while-Schleifendurchläufe für alle for-Schleifendurchläufe ebenfalls in $\mathcal{O}(n)$. Somit ist der gesamte Algorithmus in $\mathcal{O}(n)$.