1. (15 Punkte) Es sei $A[1..n]$ ein Feld mit n verschiedenen Schlüsseln. Wir bezeichnen ein Indexpaar (i, j) mit $1 \leq i < j \leq n$ als *fehlgestellt*, wenn $A[i] > A[j]$. Bezeichnen wir die gesamte Anzahl der fehlgestellten Indexpaare von $A[]$ mit F. Sollte $A[1..n]$ schon aufsteigend sortiert sein, dann gilt $F = 0$, sollte es schon absteigend sortiert sein, gilt $F = \binom{n}{2}$.

   Wenn F klein ist, dann ist das Feld $A[1..n]$ sozusagen schon fast sortiert. Welchen der in der Vorlesung vorgestellten Sortieralgorithmen würden Sie verwenden, um ein "fast sortiertes" Feld endgültig zu sortieren? Und warum? Welche Laufzeit können Sie beweisen (als Funktion von n und F)?

2. (15 Punkte) Zeichnen Sie für jede der folgenden potentiellen Inhalte des Feldes $A[1..12]$ den entsprechenden durch $A[]$ implizit dargestellten binären Baum. Klassifizieren Sie jeden der drei Fälle als Heap, Beinahe-Heap, oder nicht Heap. Illustrieren Sie für den Fall des Beinahe-Heaps eine Heapify Operation.

   (a) $\langle 13, 5, 9, 6, 5, 1, 17, 23, 16, 12, 3, 6 \rangle$

   (b) $\langle 27, 20, 19, 10, 12, 11, 5, 3, 2, 10, 5, 3 \rangle$

   (c) $\langle 7, 21, 17, 13, 15, 12, 1, 3, 12, 11, 4, 5 \rangle$

3. (15 Punkte) Sei B ein binärer Beinahe-Heap der Höhe $h > 2$, d.h. der längste Pfad von einem Knoten in B zur Wurzel w von B besteht aus h Kanten. Die in der Vorlesung besprochene Methode SIFT-DOWN wandelt B in einen Heap um. Im schlechtesten Fall passieren dabei $2h$ Schlüsselvergleiche.

   Entwerfen Sie eine andere Methode, die einen Beinahe-Heap in einen Heap verwandelt, die aber auf jeden Fall mit weniger als $2h$ Schlüsselvergleichen auskommt.

   Wie gering können Sie die im schlechtesten Fall benötigte Anzahl von Schlüsselvergleichen machen?

Auf der folgenden Seite befindet sich eine Zusatzaufgabe für die besonders Interessierten. Man kann ein bisschen herumknobeln. Ob sich wirklich etwas Vernünftiges und in der Praxis Verwendbares ergibt, weiß ich nicht. Wie schon gesagt, dies ist eine Zusatzaufgebe, die nicht bearbeitet werden muss. Erreichte Punkte zählen zusätzlich.

4. (20 Punkte extra) Die Effizienz von Heapsort beruht ganz wesentlich auf der impliziten Darstellung von bestimmten kanonischen binären Bäumen in einem Feld. Man kann es folgendermaßen betrachten: Wir nehmen den perfekten (also vollständig balanzierten) binären Baum B_h mit Höhe h. Dieser hat 2^h Blätter und $K_h = 2^{h+1} - 1$ Knoten. Diese Knoten werden in sogenannter "level-order" durchnummeriert: also, zuerst die Wurzel, dann die beiden Kinder der Wurzel ("von links nach rechts"), dann die 4 Enkelkinder der Wurzel ("von links nach rechts"), und so fort. Diese Knotennummerierung hat die Eigenschaft, dass der Elter eines Knotens immer eine kleinere Nummerierungszahl hat als der Knoten selbst. Daraus folgt, dass die ersten m Knoten in dieser Nummerierung immer einen zusammenhängenden Teilbaum von B_h bilden, für jedes $1 \leq m \leq K_h$, und dieser Teilbaum hat natürlich auch Höhe höchstens h.

   Für die Effiezienz von Heapsort ist es nun wesentlich, dass man in konstanter Zeit lokal navigieren kann. Das soll heißen: Für eine Knotennummer i kann ich schnell herausfinden, ob das überhaupt einen Knoten im derzeitigen Teilbaum mit m Knoten darstellt (nämlich $i \leq m$); ich kann in konstanter Zeit die Knotennummer des etwaigen linken Kindes produzieren ($2i$) und auch des etwaigen rechten Kindes ($2i+1$) und auch die Knotennummer des etwaigen Elters ($\lfloor i/2 \rfloor$).

   Hier kommt nun die Frage: Was passiert, wenn man statt in "level-order" die Knoten von B_h in "preorder" durchnummeriert (also zuerst die Wurzel, dann rekursiv der linke Teilbaum, dann rekursiv der rechte Teilbaum)? Kann man dann auch in konstanter Zeit lokal navigieren, also für gegebene Knotennummer in Konstanter Zeit die Knotennummer des linken und des rechten Kindes herausfinden, wie auch die Knotennummer des Elters? Geht es leichter, wenn man zusätzliche Information hat, z.B. die Tiefe des Knotens im Baum? Wie k´onnte man Heapsort mit so einer Nummerierung betreiben?