Karlsruher Institut für Technologie — Prof. Dr. Jörn Müller-Quade
Institut für Theoretische Informatik — Björn Kaidel, Sebastian Schlag, Sascha Witt

# 7. Tutorenblatt zu Algorithmen I im SoSe 2017

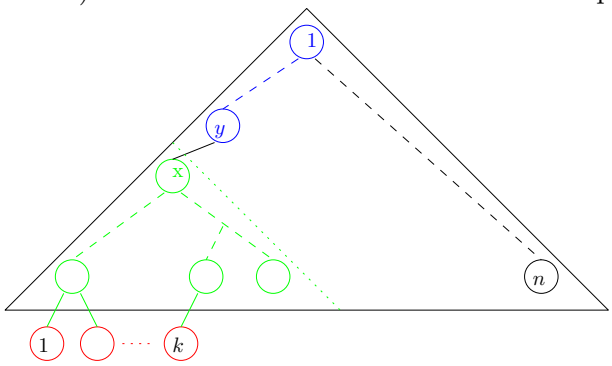
http://crypto.iti.kit.edu/index.php?id=799
{bjoern.kaidel,sascha.witt}@kit.edu

Themen auf dem siebten Übungsblatt sind ganzzahliges Sortieren und binäre Heaps. Zunächste könnt ihr also nochmal (oder zum ersten Mal) auf ganzzahliges Sortieren eingehen und Radixsort an einem Beispiel erklären. Außerdem solltet ihr ausführlich auf binäre Heaps eingehen und einfache Aufgaben dazu stellen (siehe z.B. Cormen - Introduction to Algorithms, Kapitel 7).

**Aufgabe 1** (*Bulk Insertion bei binären Heaps*)

Gegeben sei ein binärer Heap, der $n$ Elemente enthält. Es sollen nun $k$ Elemente auf einmal eingefügt werden. Gebt ein Verfahren an, mit dem man das Einfügen in $\mathcal{O}(\min\{k \log k + \log n, k + \log n \log k\})$ Schritten erledigen kann. Ihr könnt davon ausgehen, dass der Heap genau $2^m - 1$ Elemente enthält.

**Musterlösung:**
Falls $k \in \Omega(n)$, so kann man auf allen $k + n$ Elementen die Funktion *buildHeap* aufrufen, die dann eine Laufzeit von $\mathcal{O}(k)$ hat. Sei also im Folgenden $k$ immer wesentlich kleiner als $n$.
Sei $K$ die Menge der einzufügenden Elemente. Man kann nun einfach (unter Verletzung der Heapeigenschaft) die Elemente aus $K$ hinten an den Heap anhängen.




Es sei $x$ der letzte Knoten der gemeinsamen Vorgänger von allen Elementen aus $K$. Es sei $N$ (blau im Bild) die Menge aller gemeinsamen Vorgänger von $K$ außer $x$. Aufgrund der Heapeigenschaft ist $N$ in der Reihenfolge von 1 bis $y$ sortiert. Es sei $\bar{k}$ die kleinste Zweierpotenz mit $k \leq \bar{k}$. An den Teilbaum $T$ (hier grün) mit Wurzel $x$ könnte man maximal $\bar{k}$ Elemente anhängen. Da $T$ ein vollständiger Binärbaum ist, der auf jeder Ebene doppelt so viele Knoten wie auf der darüberliegenden Ebene hat, hat $T$ genau $\bar{k} - 1 < 2k \in \mathcal{O}(k)$ Knoten. Aus der Höhe der Bäume fogt, dass es $\lceil \log n \rceil - \lceil \log k \rceil + 1 \in \mathcal{O}(\log n)$ gemeinsame Vorgänger von $K$ gibt.
**Fall 1:** $k \leq \log n$
Wir dürfen also maximal $\mathcal{O}(k \log k + \log n)$ Schritte ausführen. Also wählt man folgendes Vorgehen:

- Finde $x$ in $\mathcal{O}(\log n)$ Schritten.
- Schneide $T$ aus dem Heap aus und sortiere die Menge $K \cup T$ in $\mathcal{O}(k \log k)$ Schritten.
- Schneide $N$ aus dem Heap aus und erhalte dabei die Sortierung in $N$. Führe dann einen Mergeschritt wie beim Mergesort aus, um die sortierten Mengen $N$ und $K \cup T$ zu der sortierten Menge $N \cup K \cup T$ zu vereinigen. Dies benötigt $\mathcal{O}(k + \log n)$ Schritte.
- Füge nun nacheinander die kleinsten Elemente der Liste $N \cup K \cup T$ in den Heap ein. Beginne bei der ursprünglichen Stelle von 1, und fahre fort bis zur ursprünglichen Stelle von $y$. Danach füge die sortierten Elemente ebenenweise in den Teilheap ab der ursprünglichen Stelle von $x$ ein. Dies benötigt $\mathcal{O}(k + \log n)$ Schritte.

Damit ist die Gesamtlaufzeit in $\mathcal{O}(k \log k + \log n)$ und das Ergebnis erfüllt die Heapeigenschaft an jeder Stelle.
**Fall 2:** $k > \log n$
Wir dürfen also maximal $\mathcal{O}(k + \log n \log k)$ Schritte ausführen. Also wählt man folgendes Vorgehen:

- Finde $x$ in $\mathcal{O}(\log n)$ Schritten.
- Schneide $T$ aus dem Heap aus und führe auf der Menge $K \cup T$ in $\mathcal{O}(k)$ Schritten *buildHeap* aus. Der Heap hat dann eine Größe in $\mathcal{O}(k)$.
- Entnehme nun mit der Standardfunktion die $\#N$ kleinsten Elemente aus dem Heap der Menge $K \cup T$. Erhalte dadurch eine sortierte Menge $M$. Dies benötigt $\mathcal{O}(\log n \log k)$ Schritte.
- Schneide $N$ aus dem Heap aus und erhalte dabei die Sortierung in $N$. Führe dann einen Mergeschritt wie beim Mergesort aus, um die sortierten Mengen $N$ und $M$ zu der sortierten Menge $N \cup M$ zu vereinigen. Dies benötigt $\mathcal{O}(\log n)$ Schritte.
- Füge nun nacheinander die kleinsten Elemente der Liste $N \cup M$ in den großen Heap ein. Beginne bei der ursprünglichen Stelle von 1 und fahre fort bis zur ursprünglichen Stelle von $y$. Dies benötigt $\mathcal{O}(\log n)$ Schritte.
- Füge die restlichen $\#N$ Elemente aus $N \cup M$ mit der Standardfunktion in den aus der Menge $K \cup T$ entstandenen Heap ein. Dies benötigt $\mathcal{O}(\log n \log k)$ Schritte.
- Verbinde den aus der Menge $K \cup T$ entstandenen Heap mit dem großen Heap. Dies geht in $\mathcal{O}(1)$.

Damit ist die Gesamtlaufzeit in $\mathcal{O}(k + \log n \log k)$ und das Ergebnis erfüllt die Heapeigenschaft an jeder Stelle.

**Aufgabe 2** (*Kreativaufgabe*)

Pancake-Sorting und Spaghetti-Sorting:

a) Gegeben sind $n$ Pancakes in unterschiedlicher Größe und gestapelt. Man hat einen Pancake-Flipper zur Verfügung, mit dem man die obersten $\ell$ Pancakes umdrehen kann ($\ell$ bel.). Entwickelt einen schnellen Algorithmus, um die Pancakes zu sortieren.

b) Ein anderes Sortierverfahren mit Nahrungsmitteln ist Spaghetti-Sort. Hierzu gibt es eine sehr gute Wikipidia-Seite: `http://en.wikipedia.org/wiki/Spaghetti_sort`.

**Musterlösung:**
**Pancake-Sorting:** Die einfachste Lösung ist den größten Pancake einmal nach oben zu flippen und dann den ganzen Stapel zu flippen (dieser liegt dann an der richtigen Stelle) usw. Dies benötigt im schlimmsten Fall $2n$ Flips. Nun schaut man ein wenig genauer hin. Nach dem man den zweit-kleinsten Pancake an die richtige Stelle sortiert hat, kann man aufhören (unser Algorithmus würde den kleinsten einfach zweimal drehen). Also benötigen wir im schlimmsten Fall $2n$-2 Flips. Wenn man noch genauer hinschaut, dann stellt man auch fest, dass nachdem der dritt-kleinste Pancake an der richtigen Stelle liegt, gibt es nur zwei Möglichkeiten für die oberen beiden Pancakes. Entweder sie sind sortiert oder man benötigt nur einen Flip. Also benötigen wir im schlimmsten Fall $2n$-3 Flips. Ein Algorithmus der höchstens $(5n+5)/3$ Flips benötigt, wurde von Bill Gates veröffentlicht, bevor er sein Studium in Harvard abgebrochen hat. Sein Algorithmus war für ca. 30 Jahre der effizienteste Algorithmus, um das Problem zu bearbeiten. Siehe auch `https://de.wikipedia.org/wiki/Pfannkuchen-Sortierproblem`.