# Datenstrukturen und Algorithmen
# Übung 3, Frühling 2011

10. März 2011

Diese Übung muss zu Beginn der Übungsstunde bis spätestens um 16 Uhr 15 am 17. März abgegeben werden. Die Abgabe der DA Übungen erfolgt immer in schriftlicher Form auf Papier. Programme müssen zusammen mit der von ihnen erzeugten Ausgabe abgegeben werden. Drucken Sie wenn möglich platzsparend 2 Seiten auf eine A4-Seite aus. Falls Sie mehrere Blätter abgeben heften Sie diese bitte zusammen (Büroklammer, Bostitch, Mäppchen). Alle Teilnehmenden dieser Vorlesung müssen Ihre eigene Übung abgeben (einzeln oder in Zweiergruppen). Vergessen Sie nicht, Ihren Namen und Ihre Matrikelnummer auf Ihrer Abgabe zu vermerken.

## Theoretische Aufgaben

1. In der folgenden Tabelle ist ein Heap in der üblichen impliziten Form gespeichert:
   $[2, 7, 20, 15, 10, 25, 22, 16, 21, 11]$
   Wie sieht die Tabelle aus, nachdem das kleinste Element gelöscht und die Heap-Bedingung wieder hergestellt wurde? (**1 Punkt**)

2. Zeigen Sie, dass in jedem Teilbaum eines Min-Heaps die Wurzel des Teilbaums den kleinsten Wert enthält, der in diesem Teilbaum vorkommt. (**1 Punkt**)

3. Ist ein absteigend sortiertes Feld ein Max-Heap? Begründen Sie. (**1 Punkt**)

4. Gegeben sei die Schlüsselfolge $[26, 12, 41, 33, 85, 20, 63, 18]$. Zeigen Sie den Ablauf der Build-Max-Heap Funktion ähnlich wie in Figur 6.3 im Buch. (**1 Punkt**)

5. Geben Sie die Laufzeit von Heap Sort für Felder der Länge $n$, in den Fällen wo die Felder aufsteigend und absteigend sortiert sind. Begründen Sie. (**1 Punkt**)

6. Gegeben sei die Schlüsselfolge $[35, 96, 57, 28, 79, 41, 62, 13, 84]$. Geben Sie alle Aufrufe der Prozedur *Quicksort* (Buch Seite 144) und die Reihenfolge ihrer Abarbeitung (Buch Abbildung 7.1.) an. Nehmen Sie an, dass das gesamte Feld sortiert werden soll.
   (**1 Punkt**)

## Praktische Aufgaben

Sortieren ist für die praktische Informatik von zentraler Bedeutung. In dieser Serie sollen Sie sich näher mit praktischen Implementationen von Sortieralgorithmen in Java befassen. Damit

Sie sich auf das Wesentliche konzentrieren können, stellen wir Java-Code zur Verfügung, den Sie von der Vorlesungs Webpage herunterladen können. Der Code enthält eine Implementation des *QuickSort* Algorithmus, der in der Vorlesung besprochen wurde.

In der Vorlesung wurden Sortieralgorithmen am Beispiel von ganzzahligen Feldern als Eingabedaten vorgestellt. Das Ziel einer praktischen Implementation soll aber sein, dass beliebige Daten sortiert werden können, solange es möglich ist, Elemente paarweise zu vergleichen. Der Java Code erreicht dies mittels zwei Konzepten: Erstens werden sogenannte *generische* Klassen und Methoden verwendet, und zweitens werden *Comparator* Objekte definiert, um Daten paarweise zu vergleichen. Um sich für diese Aufgabe vorzubereiten, sollten Sie sich mit diesen Konzepten bekannt machen. Dazu bietet sich diese Beschreibung an, welche beide Themen beschreibt: *http://www.theserverside.de/java-generics-generische-methoden-klassen-und-interfaces/*. Nehmen Sie sich Zeit, die Anwendung der Konzepte in unseren bereitgestellten Java Code zu studieren. Bearbeiten Sie dann folgende Aufgaben:

1. Erstellen Sie eine neue Klasse `NameVornameComparator`, welche zwei Objekte der Klasse `StudentIn` lexikographisch hinsichtlich Name und Vorname (in dieser Reihenfolge) vergleichen kann.

   Bsp: Meier Anna ist lexikographisch kleiner als Meier Beat.

   Orientieren Sie sich an der bereits fertigen Klasse `MatrikelNrComparator`. Beachten Sie, dass Ihr `NameVornameComparator` das Interface `java.util.Comparator` aus dem Java API implementieren muss. Testen Sie Ihre Implementation mittels des Programms `MiniTestApp.java`. Geben Sie Ihren Code für `NameVornameComparator` sowie die Ausgabe von `MiniTestApp.java` ab. (**2 Punkte**)

2. Versuchen Sie nun das Programm `SortTestApp.java` auszuführen. Dieses Programm erzeugt verschieden grosse Eingabefelder mit zufälligen Daten. Die Grösse der Eingabefelder geht bis zu mehr als einer Million Elemente. Die Felder werden dann *zwei Mal* mit *QuickSort* sortiert. Das heisst, zuerst wird die unsortierte Eingabe sortiert, und dann wird versucht, die bereits sortierten Daten noch einmal zu sortieren. Was beobachten Sie bei der Ausführung des Programms? Warum unterscheidet sich die Laufzeit des jeweils ersten und zweiten Sortiervorgangs? Finden Sie heraus und erklären Sie was ein *Stack Overflow Error* ist und warum er in diesem Beispiel auftritt. Beschreiben Sie Ihre Erkenntnisse in ein paar Sätzen. (**1 Punkt**)

3. Modifizieren Sie den *QuickSort* Algorithmus so, dass das Problem vermieden wird. Geben Sie den Code des modifizierten *QuickSort* ab. Stellen Sie sicher, dass Ihr Algorithmus korrekt funktioniert, indem Sie ihn mit `MiniTestApp.java` testen. Hinweis: Verwenden Sie *randomisierten QuickSort*. (**1 Punkte**)