# DA-3

Lorenz Pfaffli 18-925-230
Lukas Ingold 20-123-998

## Datenstrukturen und Algorithmen
## Übung 3 – Sortieren

Abgabefrist: 18.03.2021, 16:00h
**Verspätete Abgaben werden nicht bewertet.**

### Theoretische Aufgaben

1. In der folgenden Tabelle ist ein Min-Heap in der üblichen impliziten Form gespeichert:

   $A = [2, 11, 7, 15, 22, 9, 18, 15, 27, 31].$

   Der Heap wird als *priority queue* verwendet. Wie sieht die Tabelle aus, nachdem HEAP-EXTRACT-MIN aufgerufen wurde, also das kleinste Element gelöscht und die Heap-Bedingung wieder hergestellt wurde? **(1 Punkt)**

2. Beweisen Sie, dass in jedem Teilbaum eines Max-Heaps die Wurzel des Teilbaums den grössten Wert enthält, der in diesem Teilbaum vorkommt. Gehen Sie dafür nur von der Max-Heap-Eigenschaft aus. **(1 Punkt)**

3. Ist das Feld mit den Werten $[23, 17, 14, 6, 13, 10, 1, 5, 7, 12]$ ein Max-Heap? Begründen Sie. **(1 Punkt)**

4. Gegeben sei die Schlüsselfolge $\langle 4, 34, 17, 32, 21, 15, 65, 42 \rangle$. Zeigen Sie den Ablauf der Funktion BUILD-MAX-HEAP ähnlich wie in Abbildung 6.3 im Buch. **(1 Punkt)**

5. Wie hoch ist die Laufzeit von HEAPSORT angewendet auf ein Feld $A$ der Länge $n$, das bereits in aufsteigender Reihenfolge sortiert ist? Wie hoch ist sie, wenn das Feld $A$ in absteigender Reihenfolge sortiert ist? Begründen Sie jeweils Ihre Aussagen. **(1 Punkt)**

6. Gegeben sei die Schlüsselfolge $\langle 19, 17, 3, 28, 60, 33, 20, 30, 2 \rangle$. Geben Sie alle Aufrufe der Prozedur QUICKSORT und die Reihenfolge ihrer Abarbeitung 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 auf Ilias 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 beispielhaft folgende 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 danach folgende Aufgaben:

1. Erstellen Sie eine neue Klasse `NameVornameComparator`, welche zwei Objekte der Klasse `Student.in` 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 Punkt)**

*Vergessen Sie nicht, Ihren Sourcecode innerhalb der Deadline über die Ilias-Aufgabenseite einzureichen.*



1.) MatrikelNr Name Vorname
94419832 Briod Jean
99323462 Fischer Hans
96419642 Gans Gustav
92987654 Habegger Pascal
89345675 Moser Käthy
99876532 Müller Anton
90588921 Müller Kurt
98345632 Schneider Anna
98222634 Stucki Daniel
92123456 Wenger Thomas

2.) Stack Overflow Error ist ein Error der auftreten kann wenn sich eine Rekursive Methode zu oft selbst aufruft oder deren Terminierungsbedingung nicht richtig gewählt wurde. Hier tritt er auf da wir vom letzten Element durch die Bereits sortierte Liste hindurch iterieren:

1 2 3 4 5 6

Nun wird die 6 als Pivot Element ausgewählt und die Elemente auf der linken Seite gelangen alle nach links. Danach wird die 5 ausgewählt und alle Elemente 1-4 gelangen nach links. Dies ist Gleich dem Worts Case des QuickSort-Algorithmus. Da Sich die QuickSort Methode somit sehr oft selbst aufrufen muss kommt es zu einem Stack Overflow Error.

Quicksort auf den bereits sortierten Daten ist abhängig von der Auswahl des Pivot-Elements wie wir in dem Worst-Case fall sehen können. Wenn wir das letzte Element auswählen dann dauert es $O(n^2)$. Um so weiter rechts das Pivot Element ausgewählt wird desto länger geht der zweite Aufruf auf die bereits sortierte Liste. Bei dem vorgegebenen QuickSort wird immer das grösste Element ausgewählt somit haben wir immer die Worst-Case-Laufzeit.