Algorithmen und Datenstrukturen
Sommersemester 2022
Blatt 6

Kevin Angele, Tobias Dick, Oskar Neuhuber,
Andrea Portscher, Monika Steidl, Laurin Wischounig

Abgabe bis 03.05.2022 23:59
Besprechung im PS am 05.05.2022

Aufgabe 1 (2 Punkte): Vorrangwarteschlange

In dieser Aufgabe sind Sie dafür zuständig ein System zu implementieren, welches ankommende Flugzeuge ihrer Priorität nach sortiert. Die Flugzeuge landen anschließend in der resultierenden Reihenfolge. Die Priorität eines ankommenden Flugzeuges setzt sich zusammen aus der verbleibenden Menge an Kerosin und der aktuellen Verspätung (kann auch negativ sein für früher ankommende Flugzeuge). Für die Sortierung der Flugzeuge müssen Sie folgende Bedingungen beachten.

- Höchste Priorität haben Flugzeuge mit weniger als 25 Prozent verbleibender Menge an Kerosin
- Falls es mehrere Flugzeuge mit weniger als 25 Prozent Kerosin gibt, wird das mit der geringsten Menge bevorzugt
- Falls alle ankommenden Flugzeuge genug Kerosin haben, so werden Flugzeuge mit höherer Verspätung bevorzugt
- Sollten alle ankommenden Flugzeuge genug Kerosin haben und keine Verspätung (bzw. früher als geplant ankommen; negative Verspätung), dann werden die Flugzeuge mit geringerer Menge an Kerosin bevorzugt

Schauen Sie sich für diese Aufgabe den Code in der Datei `FlightManagement.java` an. In der Datei befinden sich drei Klassen `FlightManagement`, `PlanePriorityComparator` und `IncomingPlane`. Ihre Aufgabe ist es, die Methode *compare* in der Klasse `PriorityComparator` zu implementieren. Führen Sie anschließend die main-Methode in der Klasse `FlightManagement` aus und überprüfen Sie, dass Ihre Ausgabe der erwarteten Ausgabe entspricht.

Lösung:

```
1 class PriorityComparator implements Comparator<IncomingPlane> {
2
3     @Override
4     public int compare(IncomingPlane o1, IncomingPlane o2) {
5         // Critical fuel
6         if (o1.getRemainingFuel() < 25 || o2.getRemainingFuel() < 25) {
7             if (o1.getRemainingFuel() < o2.getRemainingFuel()) {
8                 return -1;
```

```
9               } else if (o1.getRemainingFuel() > o2.getRemainingFuel()) {
10                  return 1;
11              }
12              return 0;
13          }
14
15          // Higher delay
16          if (o1.getDelay() > 0 || o2.getDelay() > 0) {
17              if (o1.getDelay() > o2.getDelay()) {
18                  return -1;
19              } else if (o2.getDelay() < o2.getDelay()) {
20                  return 1;
21              }
22
23              return 0;
24          }
25
26          // No critical fuel, no delay
27          if(o1.getDelay() <= 0 && o2.getDelay() <= 0){
28              if(o1.getRemainingFuel() < o2.getRemainingFuel()){
29                  return -1;
30              }else if(o1.getRemainingFuel() > o2.getRemainingFuel()){
31                  return 1;
32              }
33              return 0;
34          }
35
36          throw new IllegalStateException("Case is not covered by the comparator so
    far!");
37      }
38  }
```

Aufgabe 2 (3 Punkte): Heap Construction

Gegeben ist das folgende Array an Schlüsseln: | 39 | 25 | 62 | 56 | 72 | 45 | 12 | 5 |

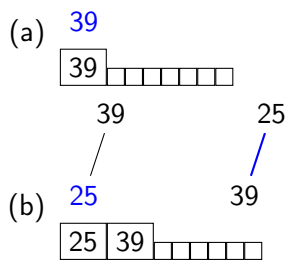
In der Vorlesung haben Sie die folgenden zwei Methoden zum Aufbau eines Heaps kennengelernt:

1. Einfügen jedes Schlüssels in einen zu Beginn leeren Heap
2. Bottom-Up Heap Construction

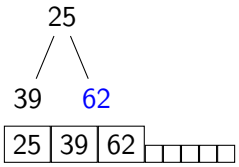
Bauen Sie für beide Methoden den Heap auf und geben Sie für alle Zwischenschritte sowohl die grafische Darstellung des Baums, als auch das Array zum Baum (d.h. den Baum in zeilenweiser Nummerierung) an.

Lösung:

1. Einfügen jedes Schlüssels in einen zu Beginn leeren Heap

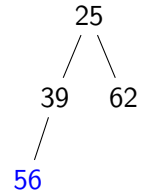


(c)



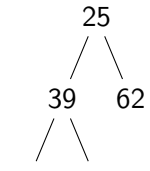
25	39	62					

(d)



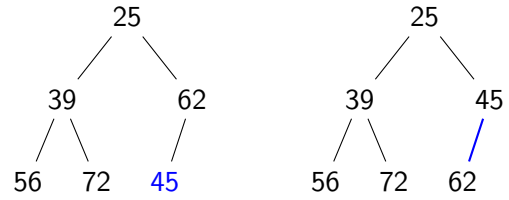
25	39	62	56				

(e)



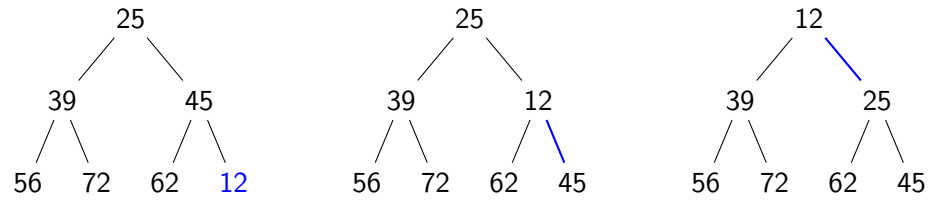
25	39	62	56	72			

(f)



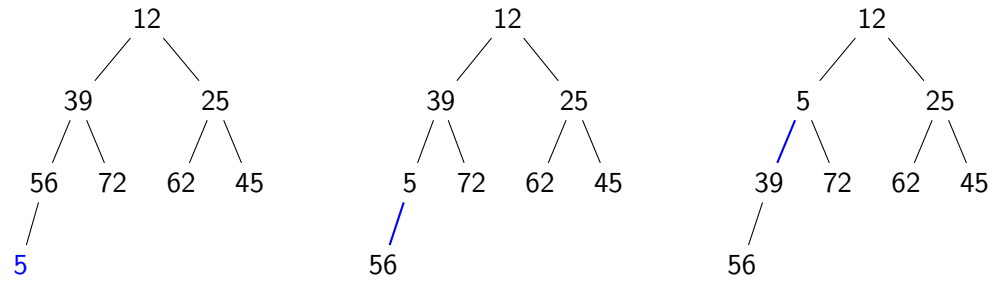
25	39	45	56	72	62		

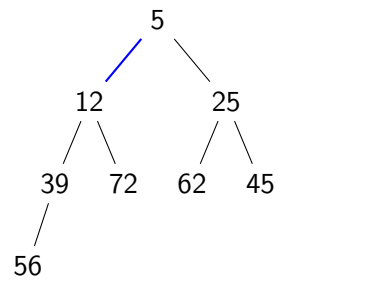
(g)



12	39	25	56	72	62	45	

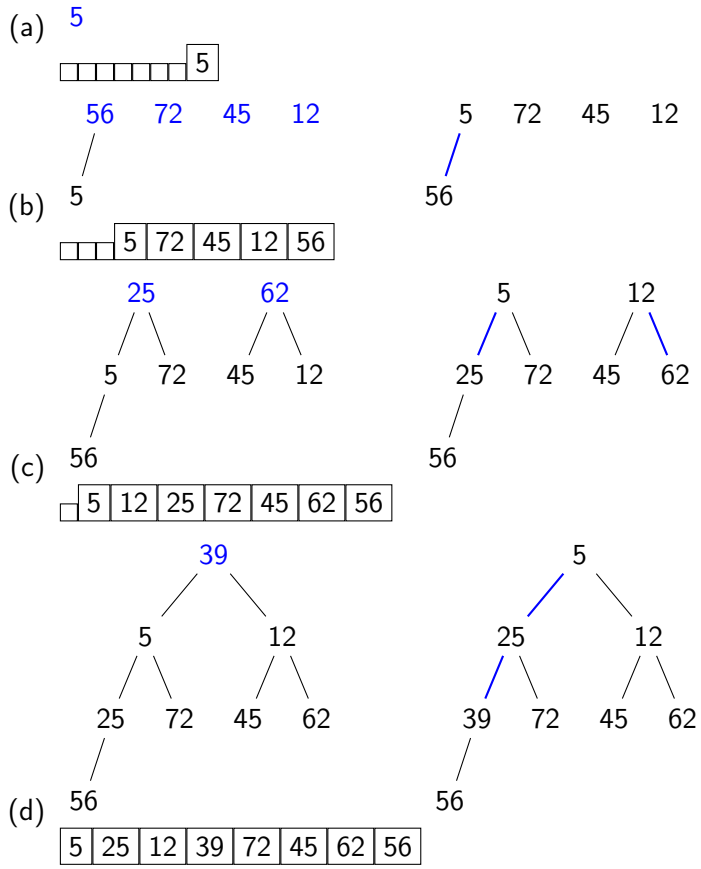
(h)





5	12	25	39	72	62	45	56

2. Bottom-Up Heap Construction



Aufgabe 3 (3 Punkte): Sortieren mit einer Vorrangwarteschlange

Eingabe: | 72 | 63 | 55 | 63 | 35 | 96 | 40 | 1 |

Sortieren Sie die gegebene Eingabe mittels der folgenden Sortieralgorithmen:

1. Heap Sort
2. Insertion Sort
3. Selection Sort

Geben Sie dazu die einzelnen Teilschritte (Arrays und beim Heap Sort den Baum) an und markieren Sie den sortierten Bereich. Welche der drei Sortieralgorithmen sind *stabil*?

Lösung:

1. **Heap Sort** (Instabil)
   - Aufbau des Max-Heaps (Bottom-Up Heap Construction)

72	63	55	63	35	96	40	1
72	63	55	63	35	96	40	1
72	63	96	63	35	55	40	1
96	63	72	63	35	55	40	1

(a) 1

(b) 63 35 96 40

1

(c) 63 55

63 35 96 40

1

63 96

63 35 55 40

1

(d) 72

63 96

63 35 55 40

1

96

63 72

63 35 55 40

1

- Extrahieren der sortierten Sequenz. Dazu wird die Wurzel gelöscht und das letzte Element wird an die Wurzel gesetzt. Anschließend muss die Heapordnung wiederhergestellt werden. Dies passiert so lange, bis der Heap eine Größe von 1 erreicht.

a)	1	63	72	63	35	55	40	96
b)	63	63	55	40	35	1	72	96
c)	63	40	55	1	35	63	72	96
d)	55	40	35	1	63	63	72	96
e)	40	1	35	55	63	63	72	96
f)	35	1	40	55	63	63	72	96
g)	1	35	40	55	63	63	72	96
h)	1	35	40	55	63	63	72	96

(a) 1

63 72

63 35 55 40

72

63 55

63 35 1 40

(b) 40

63 55

63 35 1

63

63 55

40 35 1

(c) 1

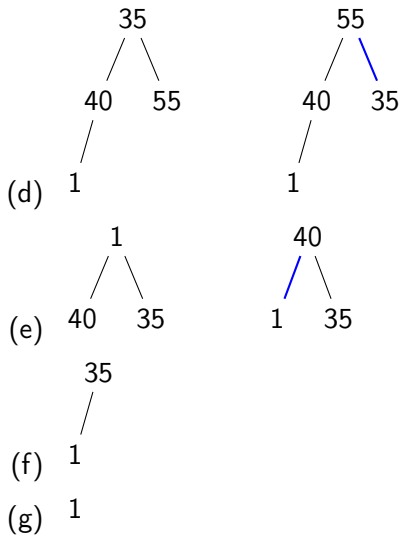
63 55

40 35

63

40 55

1 35



2. **Insertion Sort** (Stabil)

72	63	55	63	35	96	40	1
72	63	55	63	35	96	40	1
63	72	55	63	35	96	40	1
55	63	72	63	35	96	40	1
55	63	63	72	35	96	40	1
35	55	63	63	72	96	40	1
35	55	63	63	72	96	40	1
35	40	55	63	63	72	96	1
1	35	40	55	63	63	72	96

3. **Selection Sort** (Instabil)

72	63	55	63	35	96	40	1
1	63	55	63	35	96	40	72
1	35	55	63	63	96	40	72
1	35	40	63	63	96	55	72
1	35	40	55	63	96	63	72
1	35	40	55	63	96	63	72
1	35	40	55	63	63	96	72
1	35	40	55	63	63	72	96
1	35	40	55	63	63	72	96

Aufgabe 4 (2 Punkte): Lastenverteilung

In dieser Aufgabe sollen Sie sich konzeptuell überlegen welche ADTen am besten geeignet sind um ankommende Netzwerkanfragen unterschiedlicher Priorität zu verwalten. Diese Anfragen sollen ihrer Priorität nach an zur Verfügung stehende Server verteilt werden. Der Server zur Verarbeitung einer Anfrage wird nach dem Round Robin Verfahren ausgewählt. Dies bedeutet, dass reihum immer der nächste freie Server verwendet wird. Sobald ein Server eine Anfrage abgearbeitet hat, steht dieser wieder für neue Anfragen zur Verfügung.

1. Welcher ADT eignet sich besten für die Verwaltung der ...

   (a) ... ankommenden Anfragen? (Kurz begründen)

(b) ... zur Verfügung stehenden Server? (Kurz begründen)

2. Welche Implementierung des ADT würden Sie wählen wenn ...

   (a) ... das Registrieren (hinzufügen zum ADT) von Anfragen möglichst effizient sein soll? (Kurz begründen)

   (b) ... das Zuweisen von Anfragen (entnehmen aus dem ADT) zu Servern möglichst effizient sein soll? (Kurz begründen)

Lösung:

1. Am besten geeigneter ADT für ...

   (a) ... ankommende Anfrage ist die **Vorrangwarteschlange**. Anfragen mit höherer Priorität werden bevorzugt an freie Server vergeben. Der `Comparator` der Vorrangwarteschlange muss garantieren, dass Anfragen mit einer hohen Priorität einen kleinen Schlüssel bekommen. Anschließend kann das die am höchsten priorisierte Anfrage mittels `removeMin()` aus der Vorrangwarteschlange entnommen und einem freien Server zugeordnet werden.

   (b) ... zur Verfügung stehende Server ist eine **Warteschlange**. Der nächste zur Verfügung stehende Server wird mittels `dequeue()` aus der Warteschlange entnommen. Sobald ein Server eine Anfrage abgearbeitet hat, wird dieser mittels `enqueue()` wieder in die Warteschlange eingefügt.

2. Optimale Implementierung für effizientes ...

   (a) ... Registrieren von Anfragen ist mittels einer unsortierten Liste. Das Einfügen in eine Vorrangwarteschlange implementiert mittels unsortierter Liste liegt in $\mathcal{O}(1)$.

   (b) ... Zuweisen von Anfragen ist mittels einer sortierten Liste. Das Entfernen aus einer Vorrangwarteschlange implementiert mittels sortierter Liste liegt in $\mathcal{O}(1)$.