Algorithmen und Datenstrukturen INF3/ICS3

Wintersemester 2023/24

Prof. Dr. Georg Schied

Aufgabenblatt 7

Abgabetermin: Mo. 4. Dezember 2023, 23:59 Uhr

Zum Bestehen müssen 10 von 20 Punkten erreicht werden.

Aufgabe 7.1

Die folgende Speicherung einer heapbasierten Prioritätswarteschlange ist gegeben:

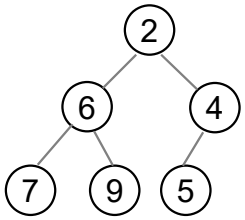
1	6	2	7	9	5	4	8			

a) Welcher Heap ergibt sich, wenn die Operation `insert(3)` ausgeführt wird?

b) Welcher Heap ergibt sich, wenn die Operation `extractMin()` für den gleichen Heap wie oben angegeben ausgeführt wird?

Aufgabe 7.2 - Scheinaufgabe (6 P)

Eine **Prioritätswarteschlange** mit den Operationen `insert` und `extractMin` kann mit Hilfe eines **binären Minimum-Heaps** realisiert werden. Gegeben ist der Inhalt der Warteschlange wie hier in Baumdarstellung angegeben.



Zeigen Sie anhand der Baumdarstellung, wie folgende Operationen nacheinander ausgeführt werden:

(1) insert(3)

(2) insert(8)

(3) extractMin()

(4) insert(1)

(5) extractMin()

(6) extractMin()

Aufgabe 7.3 - Scheinaufgabe (14 P)

Für **Prioritätswarteschlangen** zur Speicherung von Aufgaben (Klasse `Task`) sei folgendes Interface definiert:

```
public interface ITaskQueue {
    void insert(Task t);
    Task extractMin();
    boolean isEmpty();
    boolean isFull();
}
```

Die Klasse `Task` definiert eine Aufgabe, die aus einer Beschreibung vom Typ `String` und einem `int`-Wert als Priorität besteht. Eine kleinerer Zahlenwert bedeutet eine höhere Priorität. `insert(t)` nimmt die Task `t` in die Warteschlange auf, `extractMin()` entfernt die Task mit der höchsten Priorität aus der Warteschlange und liefert sie als Rückgabewert zurück. Existieren mehrere Tasks mit gleicher Priorität, dann ist unspezifiziert, welche zuerst entnommen wird.

a) Programmieren Sie eine Klasse **`HeapTaskQueue<E>`**, die eine **Prioritätswarteschlange** für Tasks mit Hilfe eines **Heaps** realisiert. Die Kapazität der Warteschlange, d.h. die Anzahl der maximal speicherbaren Einträge, soll durch den Konstruktor festgelegt werden.

```
public class HeapTaskQueue implements ITaskQueue {
  public HeapTaskQueue(int capacity) { … }
  …
}
```

Verwenden Sie dazu *nicht* die Klasse `PriorityQueue` aus der Standardbibliothek, sondern implementieren Sie die Operationen selbst!
Die Klasse `Task`, eine Vorlage für die Klasse `HeapTaskQueue` sowie eine Testklasse `JuTestHeapQueue` sind in Moodle gegeben.

Tipp: Sie können dabei Teile des Programmcodes von Heapsort mit leichten Anpassungen wiederverwenden.

b) Das Programm `PrioQueueRuntime` (siehe Moodle) bestimmt für n = 100, …, 1 000 000 Einträge jeweils die Laufzeit für folgendes Verwendungsszenario:

(1) Zunächst werden n Tasks mit zufälligen Prioritätswerten in die Warteschlange eingefügt (d.h. n mal `insert()`)

(2) Dann wird 100 mal jeweils eine Task mit zufällig gewählter Priorität durch `insert()` eingefügt und eine Task mit `extractMin()` entnommen (d.h. die Zahl der gespeicherten Tasks bleibt bei n).

(3) Zum Schluss werden alle Tasks mittels `extractMin()` aus der Warteschlange entfernt (d.h. n mal `extractMin()`)

Messen Sie damit die Laufzeiten Ihrer Implementierung (mit Option `-Xint`). Welches Laufzeitverhalten ist für die Teile (1), (2) und (3) zu erwarten? Geben Sie

die Messergebnisse an und erläutern Sie kurz, ob das theoretisch zu erwartende Laufzeitverhalten an den gemessenen Werten erkennbar ist.

c) In Moodle finden Sie auch die Klasse `SortedPrioQueue`, die eine Prioritätswarteschlange mittels eines sortierten Arrays implementiert. Führen Sie die gleichen Messungen wie bei b) durch. Welches Laufzeitverhalten ist erkennbar?

Aufgabe 7.4

Werte können mit Hilfe einer Prioritätswarteschlange folgendermaßen sortiert werden:

1. Zunächst werden alle Werte nacheinander mittels `insert()` in eine Prioritäts-warteschlange eingefügt.
2. Dann werden nacheinander alle Werte mit `extractMin()` aus der Prioritätswarteschlange entnommen. Dadurch erhält man die aufsteigend sortierte Folge der Werte.

Welche Laufzeitkomplexität hat dieses Sortierverfahren im mittleren und im schlechtesten Fall?