Algorithmen und Datenstrukturen - INF3

Wintersemester 2018/19

Prof. Dr. Georg Schied

4. Aufgabenblatt

Abgabetermin für die Scheinaufgaben: Mo. 10. Dezember 2018

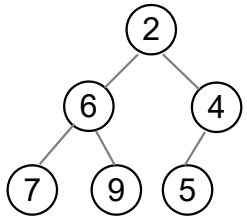
- Insgesamt 40 Punkte möglich, bestanden ab 20 Punkten.

Aufgabe 4.1

Eine **Prioritätenwarteschlange** mit den Operationen `insert` und `extractMin` kann mit Hilfe eines **binären Minimumheaps** realisiert werden.

Gegeben ist der Inhalt der Warteschlange wie unten angegeben. Vollziehen Sie anhand der Baumdarstellung nach, wie folgende Operationen nacheinander ausgeführt werden:

insert(3)
insert(8)
extractMin()
insert(1)
extractMin()
extractMin()



Aufgabe 4.2 - Scheinaufgabe (10 P)

Für Prioritätswarteschlangen sei folgendes Interface definiert:

```
public interface IPriorityQueue<E> {
      void insert(Entry<E> e);
      Entry<E> extractMin();
      boolean isEmpty();
      boolean isFull();
}
```

`insert(e)` nimmt den Eintrag `e` in die Warteschlange auf, `extractMin()` entfernt den Eintrag mit der höchsten Priorität aus der Warteschlange (kleinere Zahl bedeutet höhere Priorität) und liefert ihn als Rückgabewert zurück. Existieren mehrere Einträge mit gleicher Priorität, dann ist unspezifiziert, welcher zuerst entfernt wird.

Dabei wird folgende generische Klasse `Entry<E>` für Einträge aus einem Wert vom Typ `E` und einer Priorität vom Typ `int` verwendet (siehe Moodle).

```
public class Entry<E> {
      private E value;              //gespeicherter Wert
      private int prio;             //Priorität des Eintrags
      ...
```

a) Programmieren Sie eine Klasse **`HeapQueue<E>`**, die eine **Prioritätenwarteschlange** mit Hilfe eines **Minimum-Heaps** realisiert. Die Anzahl der maximal speicherbaren Elemente wird durch den Konstruktor festgelegt.

```
public class HeapQueue<E> implements IPriorityQueue<E>{
      public HeapQueue(int maxSize) { … }
      …
}
```

Verwenden Sie *nicht* die Klasse `PriorityQueue` aus der Standardbibliothek!

In Moodle finden Sie die jUnit-Testklasse `JuTestHeapQueue`, um Ihre Implementierung zu prüfen.

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

b) Das Programm `PrioQueueMeasurement` (siehe Moodle) bestimmt für Werte *n* = 100, 1 000, …, 1 000 000 jeweils die Laufzeiten für folgendes Verwendungsszenario:

1. Zunächst werden *n* zufällig gewählte Elemente in die Warteschlange eingefügt
2. Dann wird 100 mal jeweils ein Element mit zufällig gewählter Priorität durch `insert()` eingefügt und ein Element mit `extractMin()` entnommen.
3. Zum Schluss werden alle *n* Einträge mittels `extractMin()` aus der Warteschlange entfernt.

Messen Sie damit ihre Implementierung. Geben Sie die Messergebnisse an und erläutern Sie, ob das theoretisch zu erwartende Laufzeitverhalten an den gemessenen Werten erkennbar ist.

Aufgabe 4.3 - Scheinaufgabe (6 P)

a) Implementieren Sie eine unbeschränkte **Prioritätenwarteschlange**, die das Interface `IPriorityQueue<E>` aus Aufgabe 4.2 implementiert, als **einfach verkettete Liste**. Dazu finden Sie in Moodle eine Vorlage `aufg4_3_priolist.LinkedPrioQueue<E>` sowie eine JUnit-Testklasse `JuTestLinkedPrioQueue`.

b) Überlegen Sie, welche Größenordnungen sich für die Laufzeit der Operationen `insert(e)` und `extractMin()` im mittleren Fall ergeben.

c) Führen Sie mit Klasse `LinkedPrioQueueMessung` (siehe Moodle) die gleichen Laufzeitmessungen wie in Aufgabe 4.2 durch. Ist das erwartete Laufzeitverhalten zu erkennen?

Aufgabe 4.4

Können verkettete Listen effizient vergleichsbasiert sortiert werden, ohne die Elemente in ein Array zu kopieren und ohne dabei neue Listenknoten zu erzeugen?

Welche der für Arrays vorgestellten effizienten Sortieralgorithmen lassen Sie naheliegend auf verkettete Listen übertragen?

Aufgabe 4.5 - Scheinaufgabe (6 P)

Binäre Bäume sollen dazu eingesetzt werden, um Mengen von Personennamen zu speichern. Als Ordnungskriterium wird die übliche lexikographische Sortierung verwendet.

a) Fügen Sie nacheinander die Namen `Gustav`, `Detlev`, `Anton`, `Bernd`, `Maximilian`, `Emil`, `Dora`, `Axel`, `Erich` und `Caesar` in einen am Anfang leeren Suchbaum ein.

b) Löschen Sie aus dem Baum, der bei b) entstanden ist, nacheinander die Namen `Anton`, `Detlev` und `Gustav`.

Aufgabe 4.6 - Scheinaufgabe (12 P)

a) In Moodle finden Sie die Klassen `SearchTree` und `TreeNode` mit der Implementierung eines binären Suchbaums für Werte vom Typ `int`.

Erweitern Sie die Klasse `SearchTree` um folgende Methoden:

`public int` `sum()`

Berechnet die Summe aller Werte, die im Baum gespeichert sind.

`public int` `numOfLeaves()`

Bestimmt die Anzahl der Blätter des Baums

`public int` `extractMin()`

Entfernt den Knoten mit dem kleinsten Wert aus dem Baum und gibt dessen Wert zurück. Liefert eine `RuntimeException`, falls der Baum leer ist.

`public` `ArrayList<Integer> toList()`

Liefert die im Baum gespeicherten Werte *aufsteigend sortiert* als `ArrayList` (Paket `java.util`) zurück (Tipp: Baumtraversierung)

`public boolean` `equals(SearchTree other)`

Prüft, ob der Baum `other` genau die gleichen Werte enthält, unabhängig von der Struktur des Baums. Die Prüfung soll möglichst effizient sein.

b) Von welcher Größenordnung sind die Laufzeiten für die Methoden `extractMin()` und `equals()` im mittleren und schlechtesten Fall, abhängig von der Anzahl *n* der Elemente im Baum? Nehmen Sie bei `equals()` an, dass beide Bäume *n* Werte enthalten.

c) Messen Sie mit Hilfe des Beispielprogramms `SearchTreeMeasurement` (siehe Moodle) die Laufzeit

(1) der Kombination `extractMin()` + `insert()` und

(2) der `equals()`-Methode

bei zufällig erzeugten Bäumen. Erläutern Sie, ob das erwartete Laufzeitverhalten zu erkennen ist.

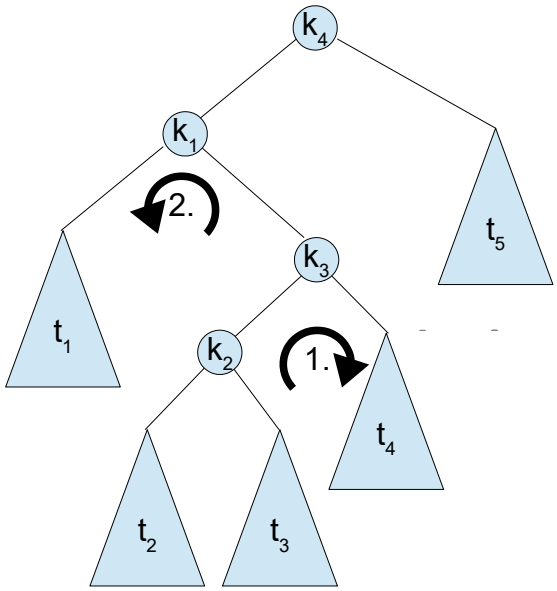
Aufgabe 4.7

Gegeben ist ein Suchbäume folgender Struktur, wobei t_1 bis t_5 irgendwelche Teilbäume sind:

a) Was kann über die Größe der Werte k_1 bis k_4 und der Werte der Teilbäume t_1 bis t_5 ausgesagt werden?

b) Führen Sie eine Doppelrotation rechts-links bei k_3 bzw. k_1 durch.

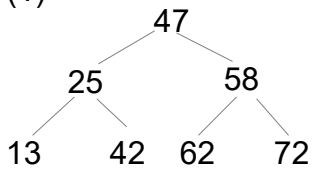
c) Begründen Sie, dass durch die Doppelrotation die Suchbaumeigenschaft erhalten bleibt.



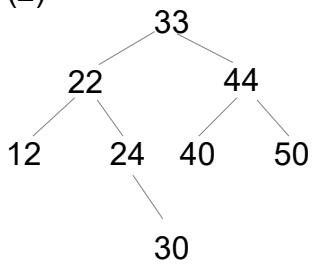
Aufgabe 4.8

Welche der folgenden Bäume sind **AVL-Bäume**, welche nicht (mit Begründung)?

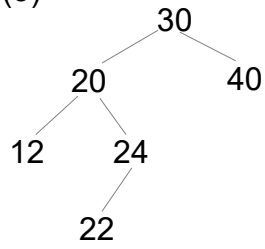
(1)



(2)



(3)



Aufgabe 4.9 - Scheinaufgabe (6 P)

a) Fügen Sie in einen am Anfang leeren **AVL-Baum** nacheinander die Werte

1, 3, 7, 9, 8, 5, 2, 4

ein. Geben Sie den Baum nach jeder Einfügeoperation an und erläutern Sie, welche Rotationen dabei zur Umstrukturierung nötig sind.

b) Löschen Sie dann nacheinander die Werte 9 und 7 aus dem AVL-Baum. Überlegen Sie sich dabei, wie durch Rotationen ggf. die Balance-Eigenschaft wieder hergestellt werden kann.