**Technische Universität Berlin** **SS 2013**
**Fakultät II, Institut für Mathematik**
Sekretariat MA 5–2, Frau Kiefer

Prof. Dr. Martin Skutella
Torsten Gellert
Martin Groß
Dr. Max Klimm

Aylin Acikel, Katharina Bütow, Christian Döblin,
Alexander Hopp, Daniel Kuske, Olivia Röhrig, Robert Rudow
Daniel Schmand, Hendrik Schrezenmaier, Mona Setje-Eilers
Judith Simon, Sebastian Spies, Fabian Wegscheider, Jan Zur

# Computerorientierte Mathematik II

## Priority Queue

## 2. Programmieraufgabe

Abnahme: spätestens am 25./26.4.2013 (je nach Gruppennummer).

**Gerade Gruppennummern geben spätestens am Donnerstag,**
**Ungerade Gruppennummern geben spätestens am Freitag ab.**

In dieser Programmieraufgabe werdet ihr eine Priority Queue implementieren. Diese baut auf die Strukturen der letzten Programmieraufgabe auf, sodass ihr nicht alles neu schreiben müsst.

Legt zunächst ein neues Projekt `programs2` mit den üblichen Ordnern für Quell- und Class-Dateien an, kopiert das Paket `binTree` und legt ein neues Paket `priorityQueue` an. Ladet das Zip-File zu dieser Aufgabe von der Homepage herunter und importiert die entpackten Dateien in das Paket `priorityQueue`.

(a) Schreibt eine generische Klasse `PriorityQueue<T extends Comparable<T> >`, die das Interface `PriorityQueueInterface<T>` implementiert. Sie soll von `BinTree<T>` erben. Da die Klasse `T` den generischen Typen `Comparable` implementiert, stellt sie Methoden zum Vergleichen der Größe der Daten bereit. Nutzt dies um die Einträge in der Priority Queue zu ordnen.

**Hinweis:** Da `BinTree<T>` und `PriorityQueue<T>` in verschiedenen Paketen liegen, müsst ihr eure Klasse `BinTreeNode` um Getter- und Setter-Methoden erweitern, falls ihr das nicht schon bei der letzten Programmieraufgabe gemacht habt.

Das Wiederherstellen der Heap-Eigenschaft kann nach dem Einfügen oder Löschen eines Knotens durch eine Methode `heapify(BinTreeNode<T> node)` durchgeführt werden. Der Parameter gibt dabei einen Knoten an, an dem die Heap-Eigenschaft möglicherweise verletzt worden ist. Es ist hilfreich weitere Hilfsfunktionen zu verwenden, z.B. zum Tauschen der Daten zwischen Elter und Kind-Knoten.

Lasst Ihr eine private Referenz `lastLeaf` auf das jeweils letzte angehängte Blatt zeigen, lassen sich Einfügeoperationen effizient gestalten. Entsprechend muss beim Einfügen und Löschen eines Datums das `lastLeaf` neu gesetzt werden.

Falls ihr in den letzten Programmieraufgabe einen dummy-Knoten verwendet habet, lässt sich die Lesbarkeit des Codes wird erhöhen, wenn man auch eine private Referenz `root` auf die Wurzel des Baumes, d.h. das linke Kind des Dummy-Knotens, zeigen lässt.

(b) Stellt zwei Konstruktoren `PriorityQueue()` und `PriorityQuene(T data)` zur Verfügung.

(c) Überschreibt in `PriorityQueue<T>` die von BinTree geerbte Methode `checkTree()`, sodass sie nun zusätzlich auch die folgenden Eigenschaften überprüft:

- Der Baum ist ein aufsteigender Heap, d.h. der Wert jedes Knotens (außer der Wurzel) ist kleiner gleich dem Wert seines Elters.
- Der Baum ist (bis auf die letzte Ebene) balanciert.
- Die letzte Ebene des Baumes ist von links nach rechts aufgefüllt.

**Hinweis:** Für einige dieser Test ist es sinnvoll, auf Methoden der letzten Programmieraufgabe zurürck zu greifen.

(d) Testet eure Implementation mit der bereitgestellten Klasse `PriorityQueueTest`.

(e) *(Optional)* Schreibt eine weitere Klasse

`PriorityQueueArray<T extends Comparable<T> >`,

die den Heap als Array realisiert und ebenfalls `PriorityQueueInterface<T>` implementiert.

Viel Erfolg und Spaß!