Literatur: Kapitel 1.1 des Skripts „Rot-Schwarz-Bäume“
Kapitel 1.2 des Skripts „Treaps“
Cormen et.al. Kapitel 13, Red-Black-Trees

Bitte beachten: Am Freitag, den 19.10.2007 findet um 9:00 Uhr in Raum 3100N eine **Einführung in *Maven*** statt.

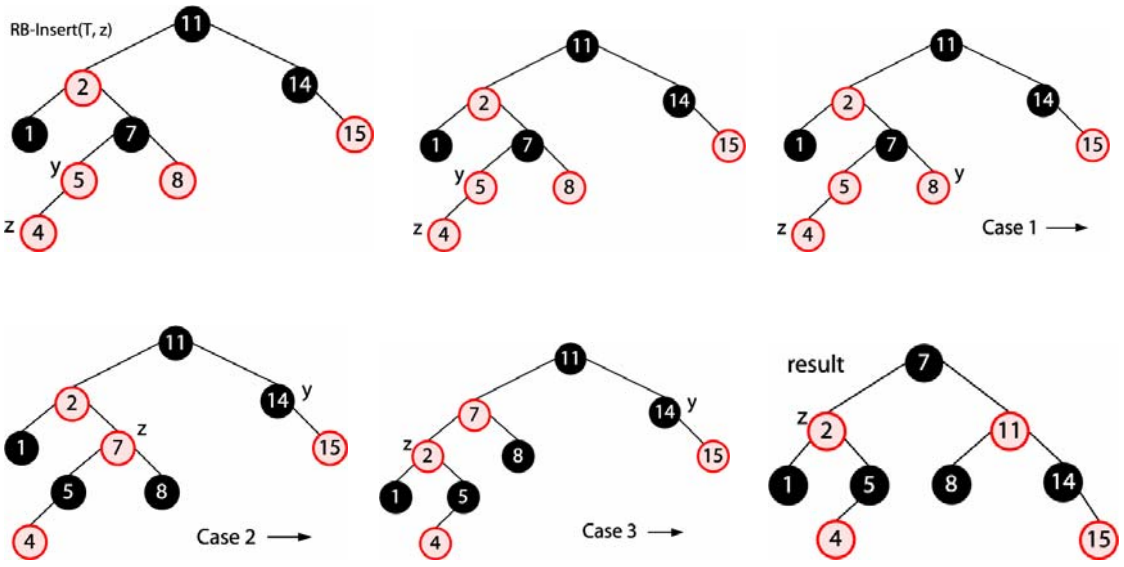
Aufgabe 1 (Rot-Schwarz-Bäume)

Implementieren Sie den ADT BinarySearchTree als Rot-Schwarz-Bäume in Java mit Sentineltechnik. Der Javacode der Operationen soll ähnlich wie der Pseudocode aussehen. ADT-Operationen:

- Empty()
- IsEmpty(x)
- InOrderTreeWalk(x), PreOrderTreeWalk(x), PostOrderTreeWalk(x)
- TreeSearch(x, k)
- TreeMin(x)
- TreeMax(x)
- TreeSuccessor(x)
- TreePredeccessor(x)
- TreeInsert(T, z)
- TreeDelete(T, z)

Dokumentieren Sie die Bibliothek und speichern Sie das Projekt *RedBlackTrees* mit *Maven* auf dem bereitgestellten Server. Erstellen Sie eine ausführliche PDF-Dokumentation.

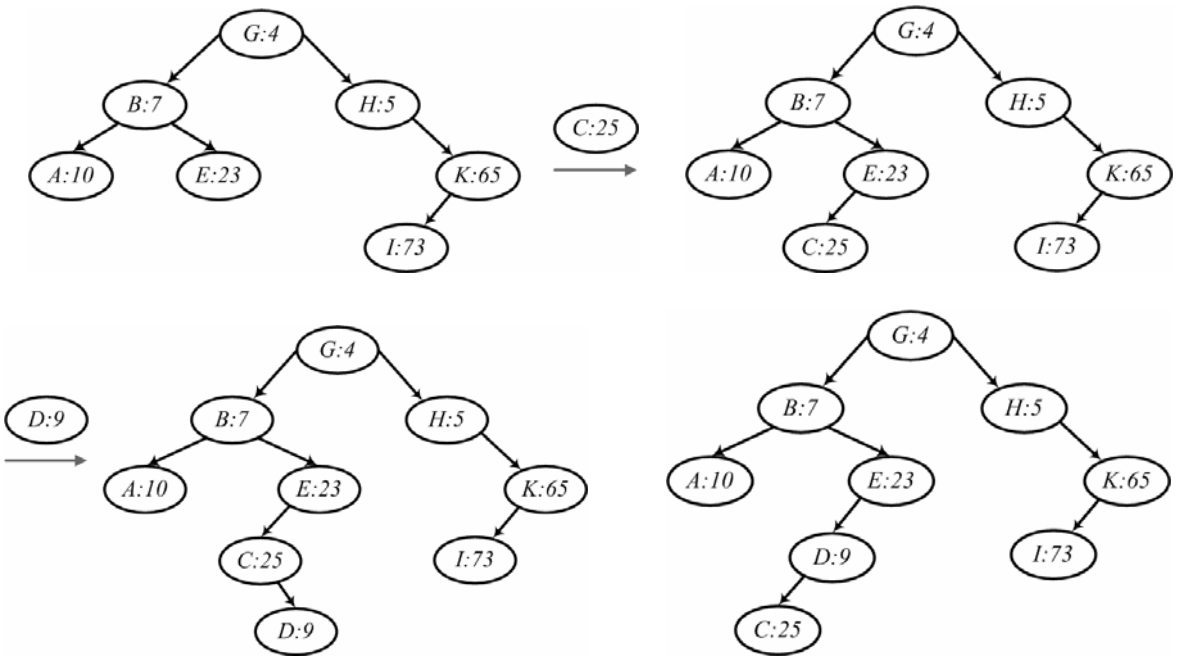
Schreiben Sie eine interaktive Testumgebung, mit sich die ADT-Operationen einzeln interaktiv testen lassen. Schlüssel sollen von einem Textfile eingelesen werden. Stellen Sie die Rot-Schwarz-Bäume, auch die Zwischenschritte der einzelnen Fälle der Insert- und DeleteFixup-Methoden, grafisch dar:



Aufgabe 2 (Treaps)

Lesen Sie Schlüssel von einem Textfile ein und verwenden Sie die **Treap-Technik** (Kap 1.2), um aus den Schlüsseln einen binären Suchbaum zu erstellen. Stellen Sie den Treap grafisch dar. Realisieren Sie die Funktion **`TreapInsert(x)`**. Schreiben Sie eine interaktive **Testumgebung**, mit der die eingelesenen Schlüssel in einem Treap gespeichert und grafisch dargestellt werden. Alles ausführlich dokumentieren.

Beispiel: Schlüssel C und D werden nacheinander eingefügt.



Aufgabe 3 (Löschen in Treaps)

In der Vorlesung wurde die Operation **`TreapInsert(x)`** besprochen, sie besteht aus einer Suchphase und einer oder mehreren Rotationsphasen. **`TreapInsert(x)`** kann im Wesentlichen in 3 Schritten realisiert werden. Dabei ist **`k=key(x)`** der Schlüssel von x und **`prio(x)`** die dazu zufällig ermittelte Priorität.

(1) Suche die Position für das Element x ohne die Prioritäten zu beachten, d.h. mit virtueller Priorität ∞.

(2) Erzeuge an dieser Position einen neuen Knoten für x. Jetzt ist x eventuell ein „schlechter" Knoten, da die Priorität von x kleiner als die Priorität seines Vaters sein kann, d.h. **`prio(x) < prio(p(x))`**.

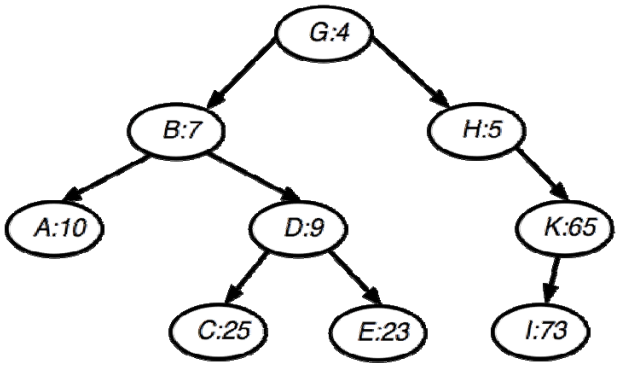
(3) Wiederherstellen der Priority-Queue-Eigenschaft.

```
while prio(p(x)) > prio(x) do
   if x rechter Sohn
     then Linksrotation(p(x))
     else Rechtsrotation(p(x))
```

Es gilt folgendes **Lemma**:

1. Links- und Rechtsrotationen lassen die Suchbaumstruktur intakt und respektieren alle Prioritätsregeln außer die für x und dessen Vater p(x).
2. Beim Abbrechen der Schleife in (3) liegt wieder ein Treap vor.

a) Realisieren Sie, ähnlich wie bei **`TreapInsert(x)`** die Operation **`TreapDelete(k)`** als Pseudocode, wobei **`k=key(x)`**.

b) Stellen ein entsprechendes Lemma für **`TreapDelete(k)`** auf.
Welcher Zusammenhang besteht zwischen den Operationen von **`TreapInsert(x)`** und denen von **`TreapDelete(k)`**?

c) Führen Sie **`TreapDelete(k)`** für das Element **`k=key(x)=B`** des folgenden Treaps aus. Zeichnen Sie dazu die einzelnen Schritte und erklären Sie diese.