Albert-Ludwigs-Universität
Institut für Informatik
Prof. Dr. Fabian Kuhn
P. Bamberger, Y. Maus, P. Schneider

Informatik 2 - Sommersemester 2018
Musterlösung Übungsblatt 6

Abgabe: Montag, 11. Juni, 14:00 Uhr

Aufgabe 1: Post-Order Traversierung *(6 Punkte)*

(a) Gegeben seien die Schlüssel $5, 7, 8, 9, 11, 12, 14, 15, 18$. Geben Sie eine Folge von `insert`(key) Operationen an, so dass ein binärer Suchbaum entsteht, bei dem man die Knoten mit der Post-Order Traversierung in der Reihenfolge $7, 5, 9, 8, 12, 15, 18, 14, 11$ besucht. *(3 Punkte)*

(b) Beschreiben Sie ein Verfahren, mit dem man aus der Post-Order gegeben als Array $A[0, \ldots, n-1]$ den zugehörigen binären Suchbaum (mit n Knoten) rekonstruieren kann. Sie können dazu Pseudocode verwenden. Gehen Sie davon aus, dass alle Schlüssel paarweise verschieden sind. *(3 Punkte)*

Musterlösung

(a) Zum Beispiel: `insert`(11), `insert`(8), `insert`(5), `insert`(7), `insert`(9), `insert`(14), `insert`(12), `insert`(18), `insert`(15)

(b) **Lösung 1:** Gegeben sei ein Array $A[0, \ldots, n-1]$, welches eine Post-Order Aufzählung der Schlüssel des binären Suchbaums darstellt. Der letzte Schlüssel gehört zur Wurzel, die kleineren Schlüssel (welche zu Beginn stehen) bilden den linken Teilbaum und die größeren Schlüssel den rechten Teilbaum. Das Verfahren wird dann rekursiv auf die Teilbäume angewandt.

Algorithm 1: `Reconstruct`$(A[0, \ldots, n-1])$

```
if n < 1 then
    return ⊥
r ← TreeElement(A[n − 1])
if n = 1 then
    return r
i ← 0
while A[i] < A[n] do
    i ← i + 1
r.left ← Reconstruct(A[0, . . . , i − 1])
r.right ← Reconstruct(A[i, . . . , n − 1])
return r
```

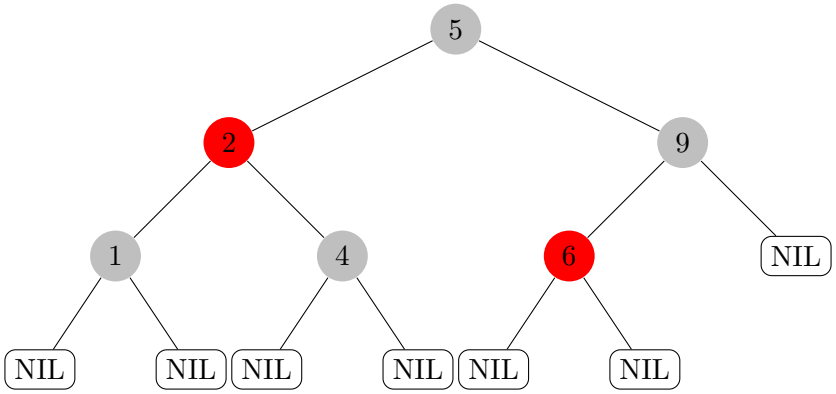
Lösung 2: Sehr einfach zu implementieren ist die folgende Lösung: Führe `insert` auf die gegebene Post-Order in umgekehrter Reihenfolge aus.

Dass das funktioniert sieht man daran, dass man zuerst die Wurzel einfügt, die ja am Ende der Post-Order steht. Alle Schlüssel die größer sind als die Wurzel werden mittels `insert` tatsächlich im rechten Teilbaum eingefügt (sonst wäre ja die Suchbaumeigenschaft verletzt). Ebenso werden alle kleineren Schlüssel in den linken Teilbaum eingefügt. Nun stehen auch die kleineren bzw. größeren Schlüssel in Post-Order zusammenhängend im Array.

Das heißt, dass wir die obige Eigenschaft induktiv für das rechte und linke Kind der Wurzel benutzen können, die jeweils Wurzeln des rechten und linken Teilbaumes darstellen und deshalb in der Post-Order ganz rechts von den Elementen ihrer jeweiligen Teilbäume stehen. Da deren Elemente ebenfalls in umgekehrter Post-Order eingefügt werden, gilt die Suchbaumeigenschaft induktiv auch für die Teilbäume.

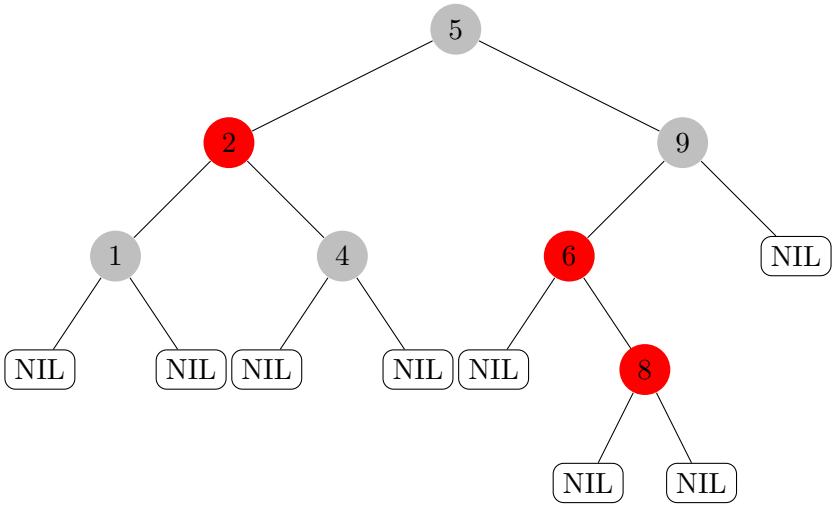
Aufgabe 2: Rot-Schwarz Bäume *(6 Punkte)*

Gegeben sei folgender Rot-Schwarz-Baum. Führen Sie zuerst die Operation `insert`(8) und danach `delete`(4) aus. Zeichnen Sie den resultierenden Baum. Dokumentieren Sie Ihre Zwischenschritte.



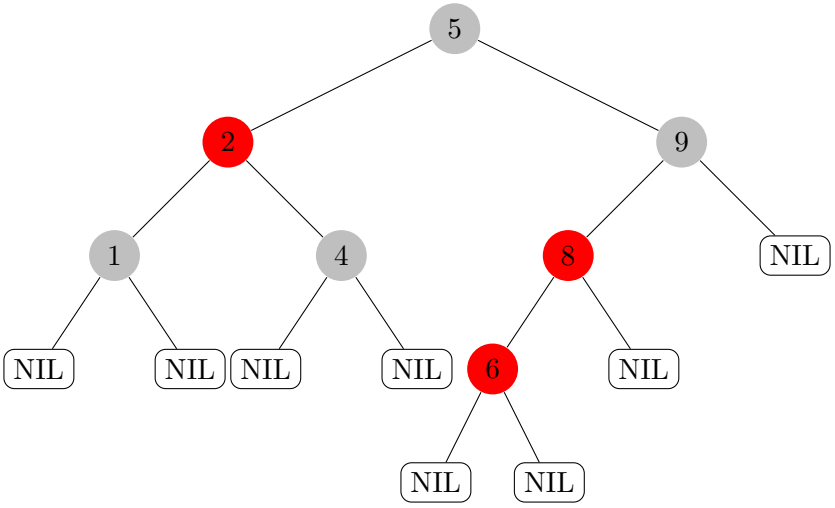
Musterlösung

Zunächst wird der Knoten mit Schlüssel 8 wie üblich eingefügt. Per Definition ist der eingefügte Knoten rot.

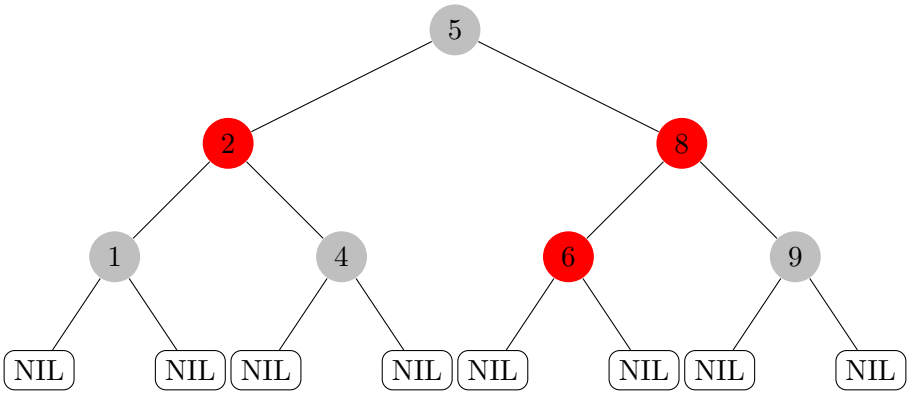


Dies ist kein Rot-Schwarz Baum mehr, da ein roter Knoten (6) ein rotes Kind hat (8). Das weitere Vorgehen entspricht dem von Seite 14 aus Vorlesung 11:

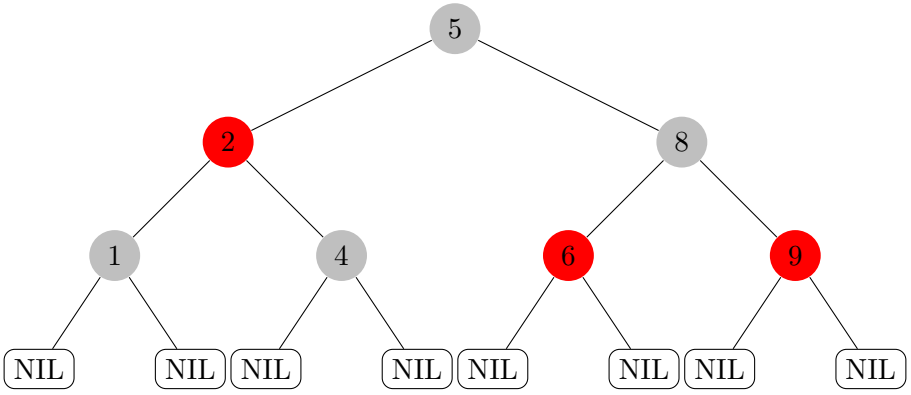
left-rotate(6,8)



right-rotate(9,8)

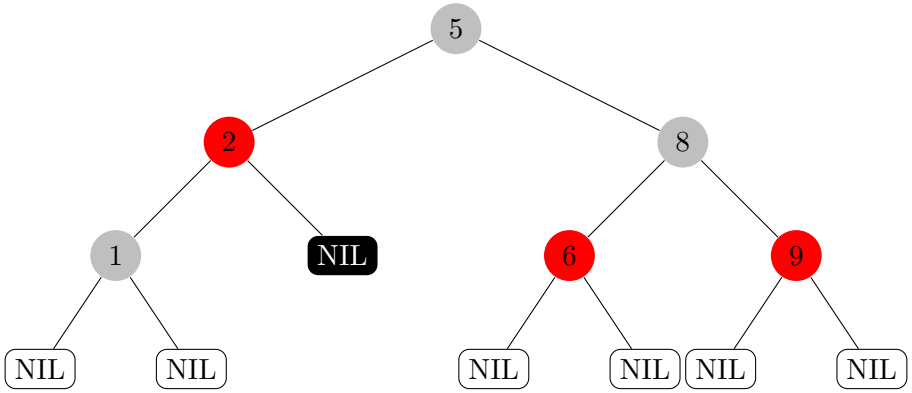


Umfärben

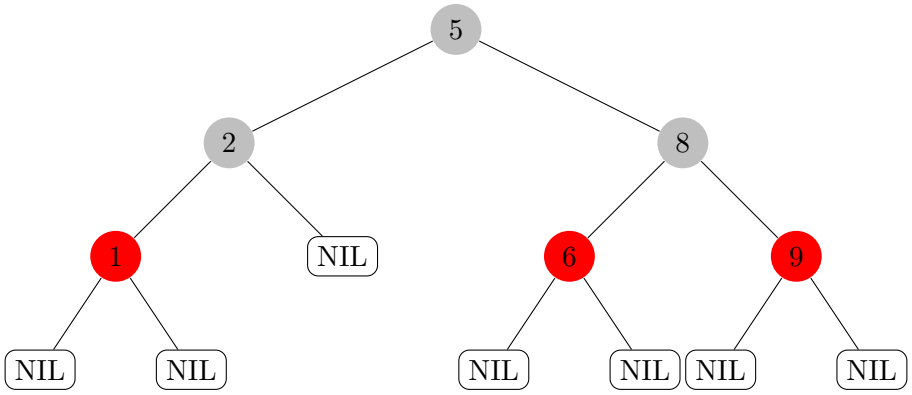


Dies ist ein gültiger Rot-Schwarz Baum.

Wir löschen Knoten 4 und ersetzen ihn durch einen NIL-Knoten mit doppelter Schwarzfärbung, um die Schwarztiefe zu erhalten.



Die doppelte Schwarzfärbung wandert nach oben und färbt Knoten 2 schwarz. Zudem wird Knoten 1 rot gefärbt. (*Fall* A.3 *aus der Vorlesung*)



Dies ist ein gültiger Rot-Schwarz Baum.

Aufgabe 3: Treaps *(8 Punkte)*

(a) Implementieren Sie einen *Treap* wie in der Vorlesung beschrieben. Sie *können* dazu die Vorlage `Treap.py` benutzen welche Ihnen bereits grundlegende Funktionen zur Verfügung stellt, wobei lediglich die Funktionen `rotate_right`, `rotate_left`, `delete` und `repair_heap` implementiert werden müssen. *(4 Punkte)*

(b) Vergleichen Sie Ihre Implementierung des binären Suchbaums vom vorherigen Übungsblatt (oder die von uns zur Verfügung gestellte `BinarySearchTree.py`) mit Ihrer Implementierung des Treap. Tun Sie dies indem Sie die Suchschlüssel $(0, 1, \ldots, 2^k)$ in dieser Reihenfolge für $k \in \{1, \ldots, 9\}$ in beide Datenstrukturen einfügen. Anschließend sollen Sie in einem Schaubild die Höhe der beiden Binärbaume über der Anzahl der eingefügten Schlüssel auftragen. Unsere Vorlage bietet dafür eine Funktion an welche Sie benutzen *können*. *(4 Punkte)*

Musterlösung

(a) Siehe `Treap.py`.

(b) Siehe Abbildung 1.

Abbildung 1: Plot der Baumhöhe (Binärer Suchbaum oben, Treap unten) aufgetragen über der Anzahl der eingefügten Schlüssel. Logarithmische Skalen.