Übungsserie 6

Aufgabe 1: Operationen auf einem Binary-Search-Tree

a) Auf einem leeren *Binary-Search-Tree*, welcher eine *Map* realisiert, werden nacheinander folgende *insert()*-Operationen ausgeführt:

```
insert(5, "Fuenf")
insert(3, "Drei")
insert(6, "Sechs")
insert(1, "Eins")
insert(2, "Zwei:1")
insert(4, "Vier:1")
insert(4, "Vier:2")
insert(2, "Zwei:2")
```

Wie sieht der *Binary-Search-Tree* danach aus?

b) Nun werden folgende *delete()*-Operationen ausgeführt:

```
delete(1)
delete(5)
```

Wie sieht der *Binary-Search-Tree* danach aus?

Aufgabe 2: Implementation eines Binary-Search-Tree

Es soll die Klasse *BinarySearchTree* (siehe ILIAS) als Map fertig implementiert werden.
Mit der Klasse *BinarySearchTreeTest* wird der Baum getestet (siehe ***`main()`***).

Hinweise:

- Am besten die Methoden der Reihe nach von 'oben nach unten' implementieren.
- Die Methode ***`search()`*** (resp. ***`_search()`*** in Python) entspricht der Pseudocode-Methode ***`TreeSearch()`*** .
- Die bestehende Methode ***`insertButWrong()`***(resp. ***`_insert_but_wrong()`*** in Python) fügt zwar Knoten ein, aber falsch.
  Diese Methode muss ersetzt werden.
- Es müssen keine Knoten erzeugt werden, sondern ein gefundener externer Knoten wird mit ***`convertToInternalNode()`***(resp. ***`convert_to_internal_node()`*** in Python) in einen internern konvertiert (und in dieser Methode werden dann zwei entsprechende neue externe Knoten erzeugt).

- Optional kann der Baum wieder mit *ADV* visualisiert werden (Setup gemäss Übung 4). Dazu wieder in ***`main()`*** die entsprechenden Kommentare anpassen:

  BinarySearchTreeTest.java:

```
...
BinarySearchTree<Integer, String> bst =
   //new BinarySearchTree<>();
   new BinarySearchTreeADV<>("Binary-Search-Tree");
```

  binary_search_tree_test.py:

```
...
#bst = BinarySearchTree()
bst =  BinarySearchTreeADV("Binary-Search-Tree")
```