AVL-Bäume: Balancieren nach Einfügen

```
1  void AVLIns(Tree t, Node node) {
2    bstIns(t,node);
3    //Node deepestUnbalancedNode(Tree t, Node node)
4    //gibt null zurück wenn t balanciert ist
5    //und den tiefsten unbalancierten Knoten in t sonst
6    //(der Parameter node wird zur effizienten Implementierung
7    //verwendet)
8    Node A = deepestUnbalancedNode(t,node);
9    if (A != null) balance(t, A);
10 }
```

AVL-Bäume: Balancieren nach Einfügen

```
1  void balance(Tree t, Node A){
2    //A ist tiefster unbalancierter Knoten in t
3    if (height(A.left) > height(A.right)) {
4      if (height(A.left.left) >= height(A.left.right)) { //LL
5        rightRotate(t,A);
6      } else { //LR
7        leftRotate(A.left); rightRotate(A);
8      }
9    } else {
10     if (height(A.right.right) >= height(A.right.left)) { //RR
11       leftRotate(t,A);
12     } else { //RL
13       rightRotate(A.right); leftRotate(A);
14     }
15   }
16 }
```

AVL-Bäume: Balancieren nach Löschen

```
1  void AVLDel(Tree t, Node node) {
2    bstDel(t,node);
3    //Node deepestUnbalancedNode(Tree t, Node node)
4    //gibt null zurück wenn t balanciert ist
5    //und den tiefsten unbalancierten Knoten in t sonst
6    //(der Parameter node wird zur effizienten Implementierung
7    //verwendet)
8    Node A = deepestUnbalancedNode(t,node);
9    while (A != null) {
10     //bool balanced(Tree t, Node A)
11     //gibt true zurück wenn A balanciert ist in t
12     //und false sonst
13     if (!balanced(t, A)) {
14       balance(t, A);
15       A = A.parent.parent;
16     } else {
17       A = A.parent;
18     }
19   }
20 }
```