AVL Trees

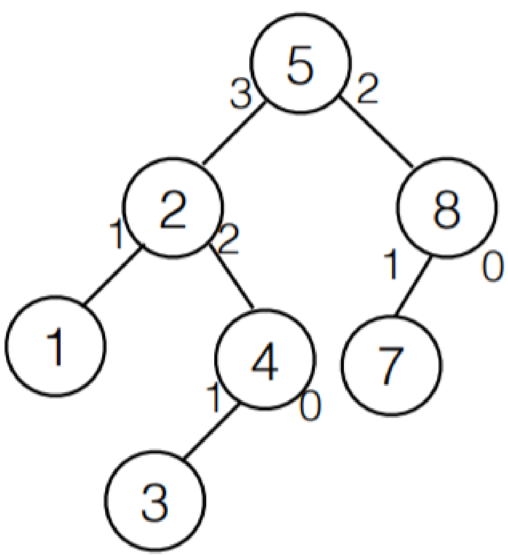
Slides adapted from Daniel Bauer

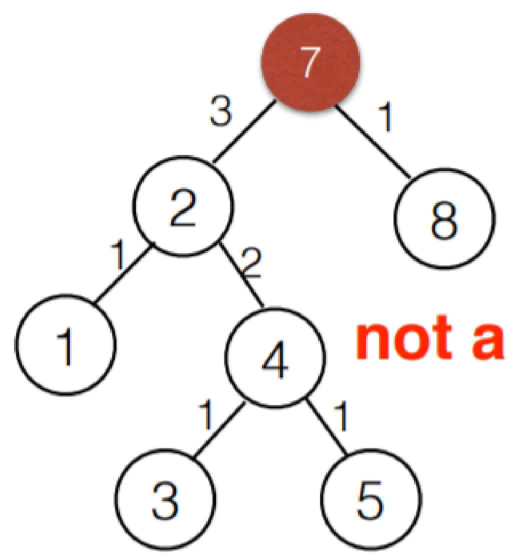
AVL Trees

- General binary search trees have $O(h)$ insertion/deletion worst-case time complexity, where h is the height of the tree
- Complete and close-to-complete BSTs have height that's approximately $\log n$, where n is the number of nodes
- When inserting/deleting, want to keep our BST be of height approximately $\log n$
- AVL trees (after the inventors Georgy Adelson-Velsky and Evgenii Landis) achieve that by keeping track of the height of each subtree and by modifying the tree as necessary with each insertion/deletion

AVL Tree Condition

- AVL tree **balance condition**:
  - For every node, the height of the left and right subtree differs by at most 1





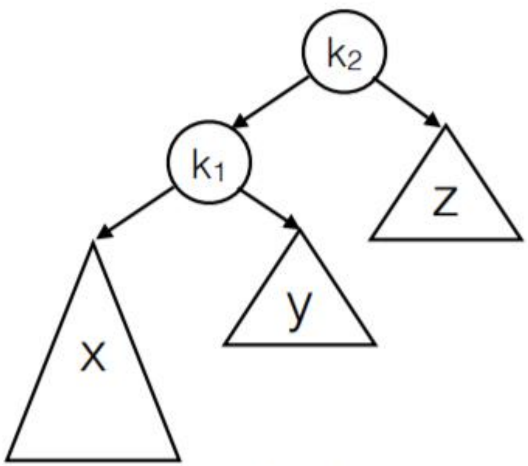
not an AVL tree

AVL trees

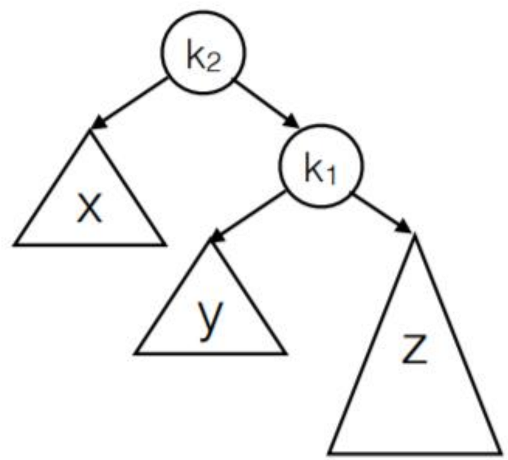
- After each insertion/deletion, tweak the tree to maintain the balance condition
- The operations needed to maintain the balance condition must be $O(\log n)$
  - Or else we're back to slow insertion/deletion, defeating the point of using AVL trees

“Outside” imbalance

Node k_2 violates the balance condition

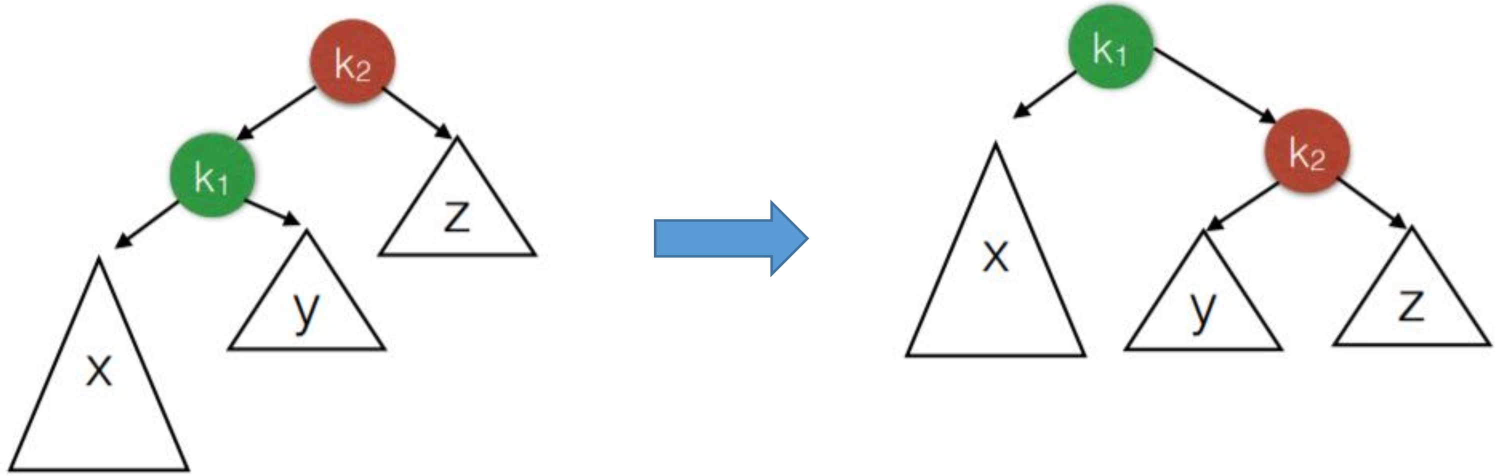


left subtree of left child too high



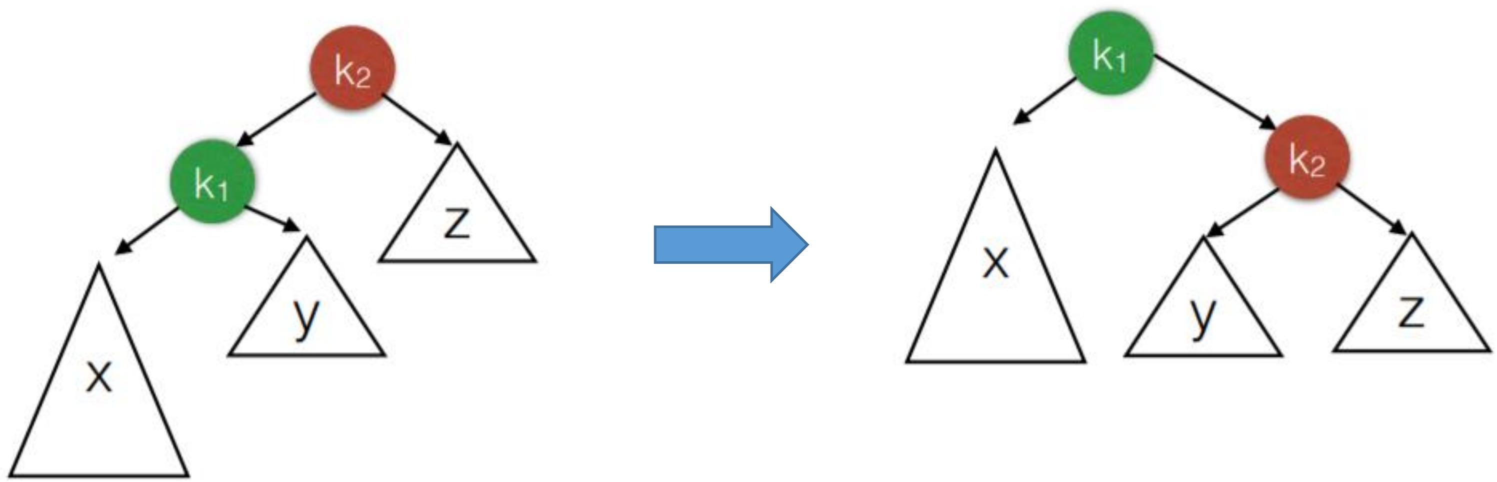
right subtree of right child too high

Fixing “outside” imbalance: a single rotation



- The BST property is satisfied
  - X is still a left subtree of k1
  - y is still in the right subtree of k_1
  - z is still in the right subtree of k_1
  - $k_1 \leq k_2$, so k_2 can be a right child of k_1

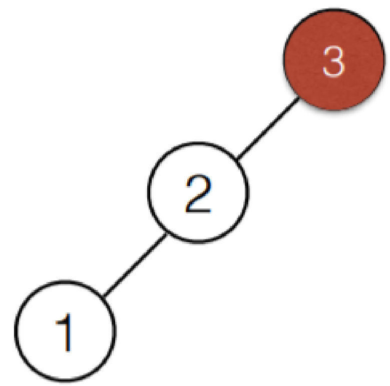
Fixing “outside” imbalance



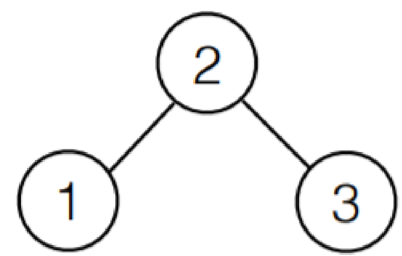

```
k2.left = k1.right
k1.right = k2
Update heights as necessary
```

Single rotation example

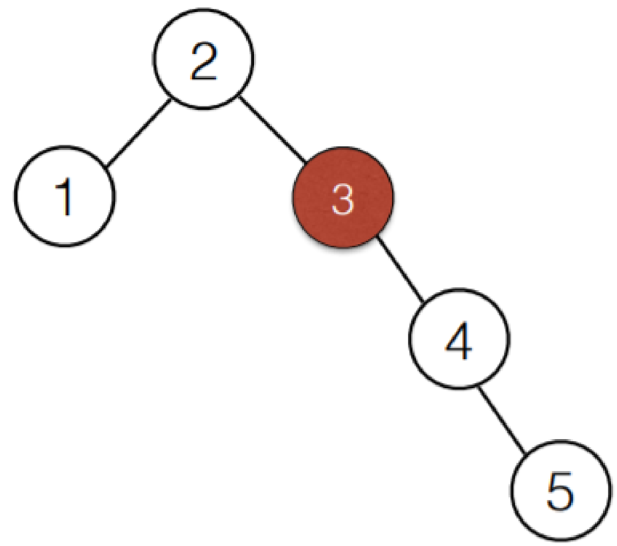
insert(3)
insert(2)
insert(1)   rotate_left(3)



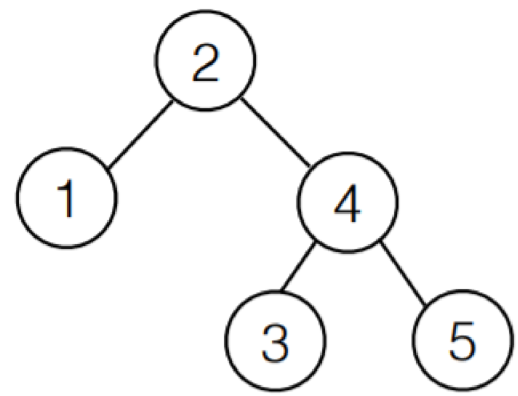
insert(3)
insert(2)
insert(1)   rotate_left(3)



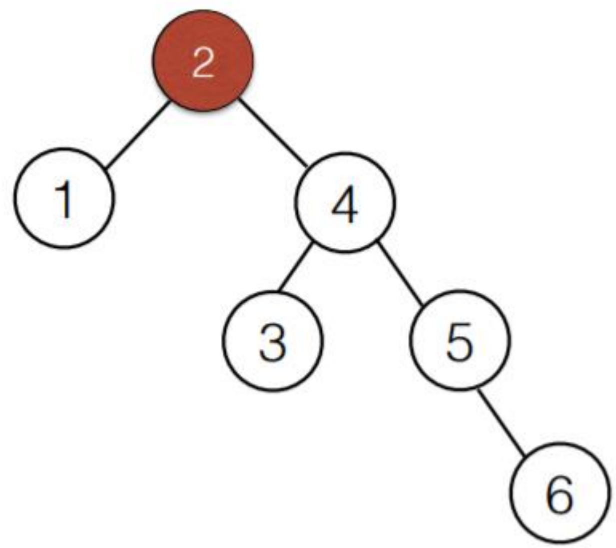
insert(3)
insert(2)
insert(1)   rotate_left(3)
insert(4)
insert(5)   rotate_right(3)



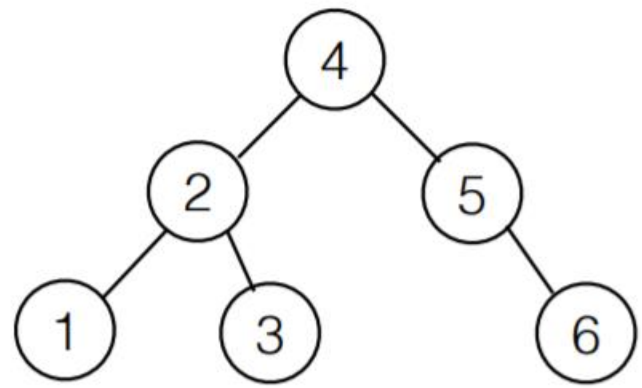
insert(3)
insert(2)
insert(1) rotate_left(3)
insert(4)
insert(5) rotate_right(3)



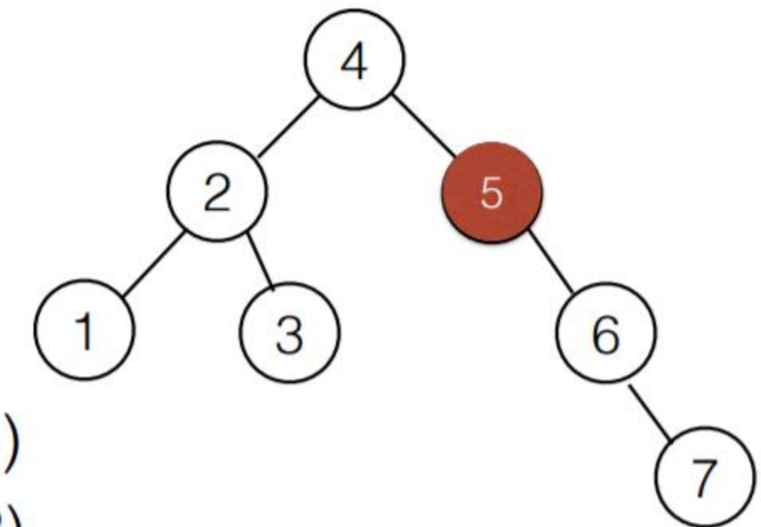
insert(3)
insert(2)
insert(1) rotate_left(3)
insert(4)
insert(5) rotate_right(3)
insert(6) rotate_right(2)



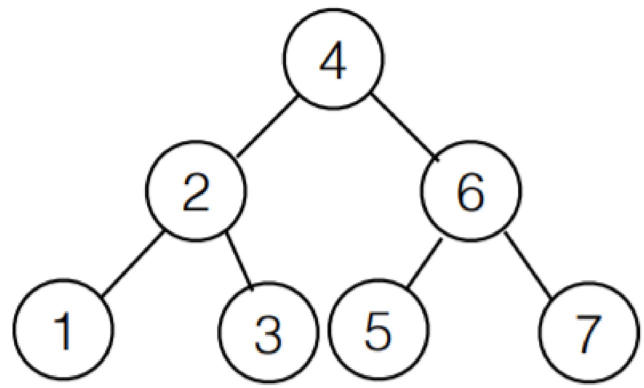
insert(3)
insert(2)
insert(1) rotate_left(3)
insert(4)
insert(5) rotate_right(3)
insert(6) rotate_right(2)



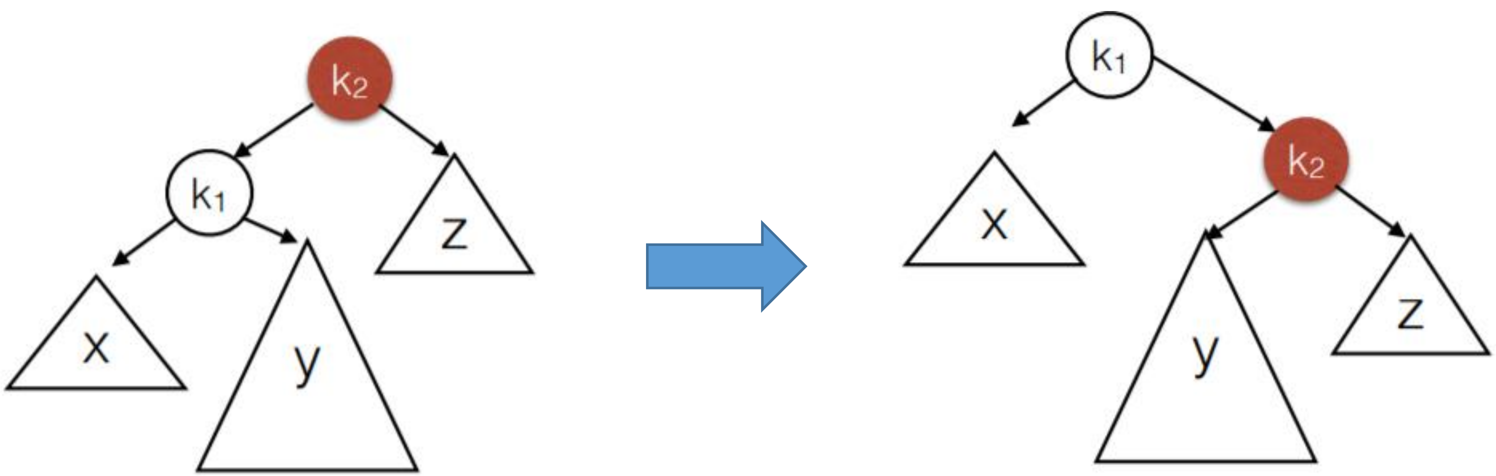
insert(3)
insert(2)
insert(1) rotate_left(3)
insert(4)
insert(5) rotate_right(3)
insert(6) rotate_right(2)
insert(7) rotate_right(5)



insert(3)
insert(2)
insert(1) rotate_left(3)
insert(4)
insert(5) rotate_right(3)
insert(6) rotate_right(2)
insert(7) rotate_right(5)

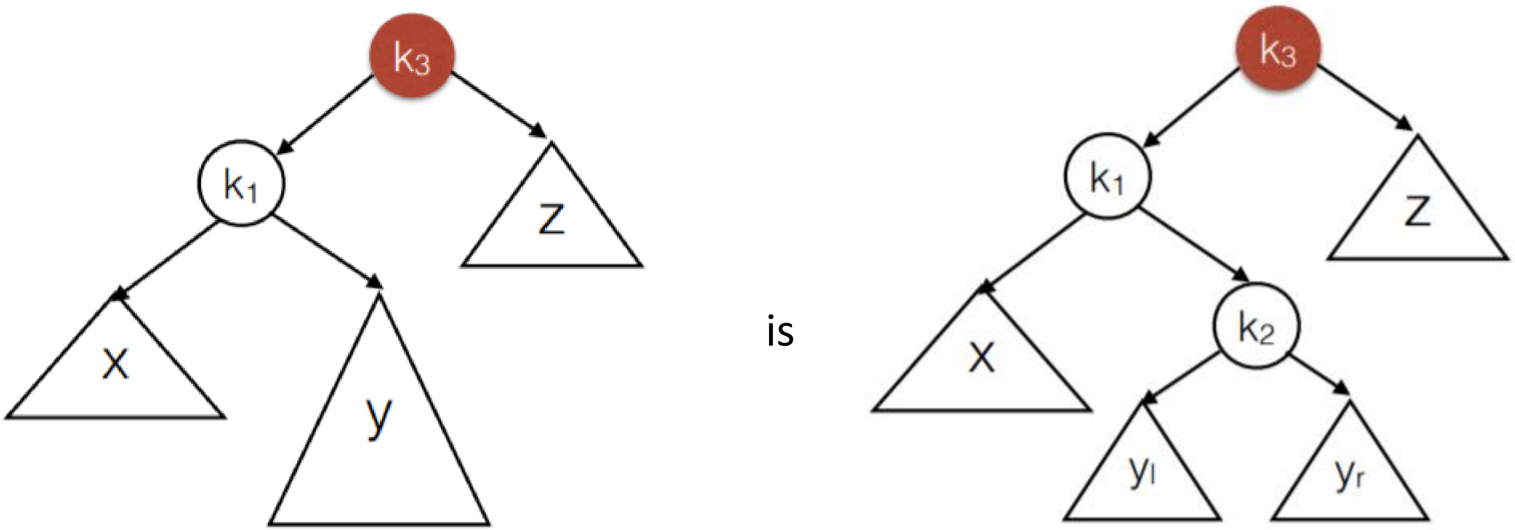


"Inside" imbalance cannot be fixed with a single rotation



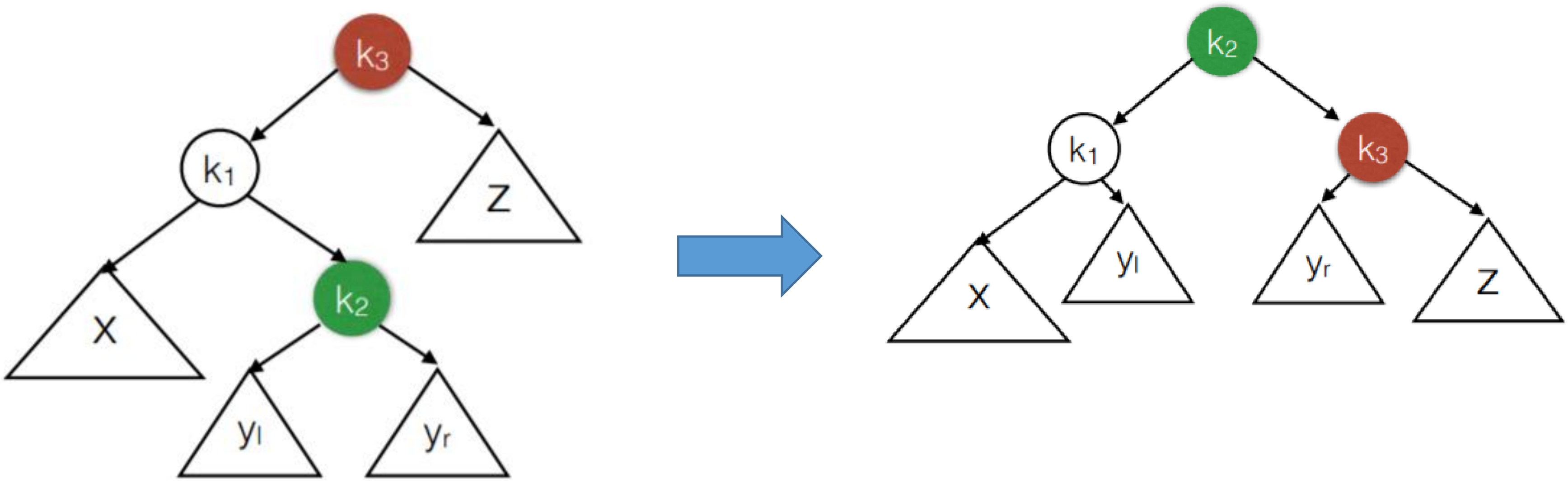
Still not an AVL tree

Double rotation

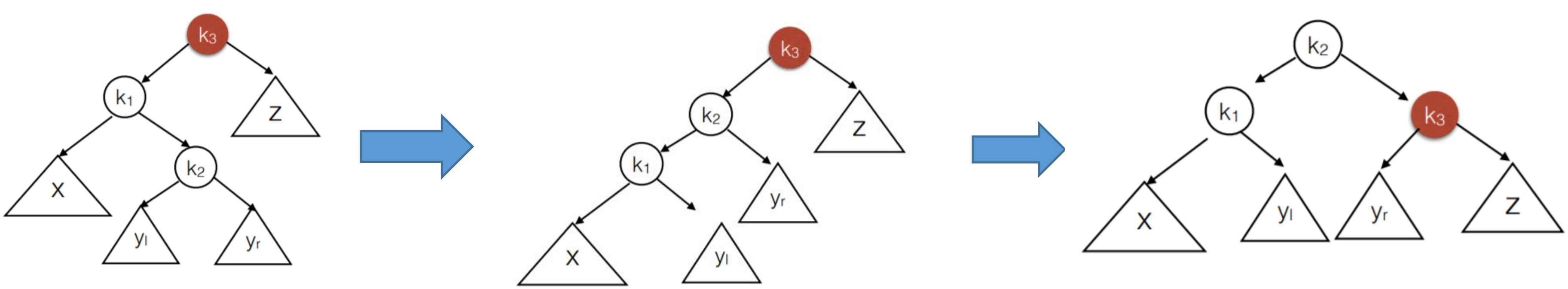


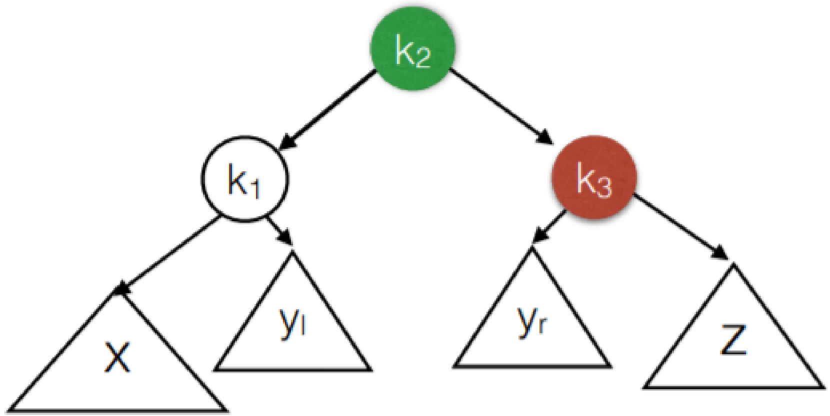
(Can do this because y has a root and a child if it’s two levels higher than z)

Double rotation



Can be implemented as: rotate k1 to the left, and then rotate k3 to the right



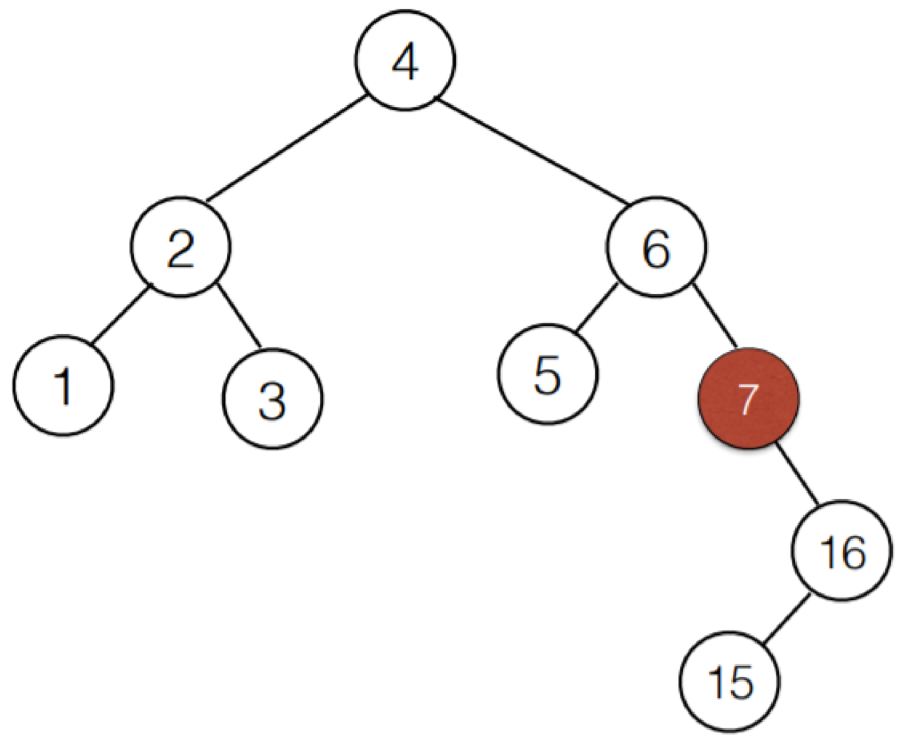



```
k2.left = k1
k2.right = k3
k1.right = root(yl)
K3.left = root(yr)
```

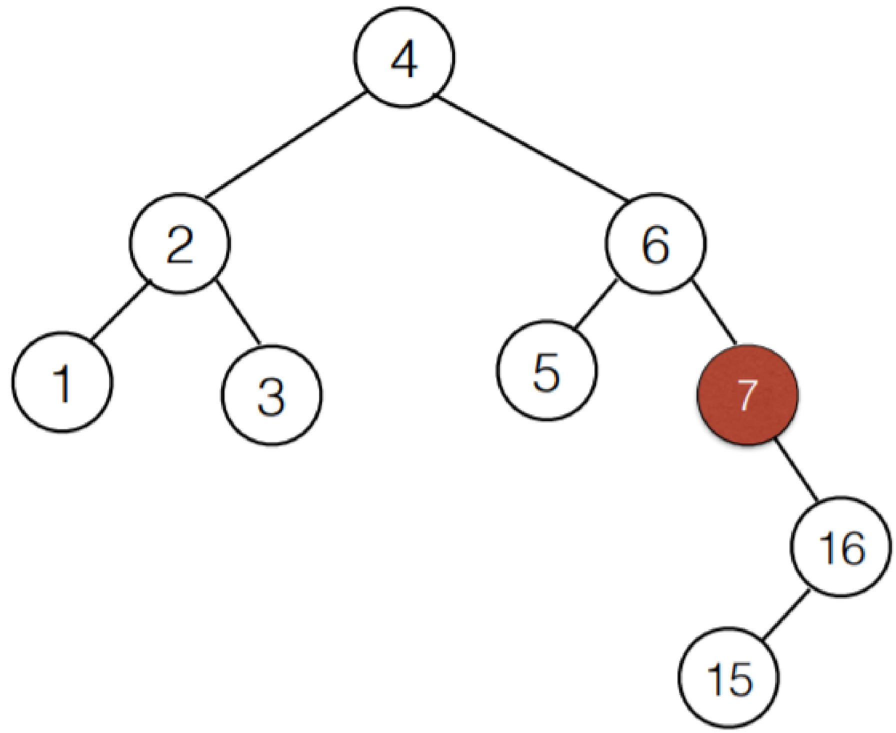
Double rotation example

insert(16)

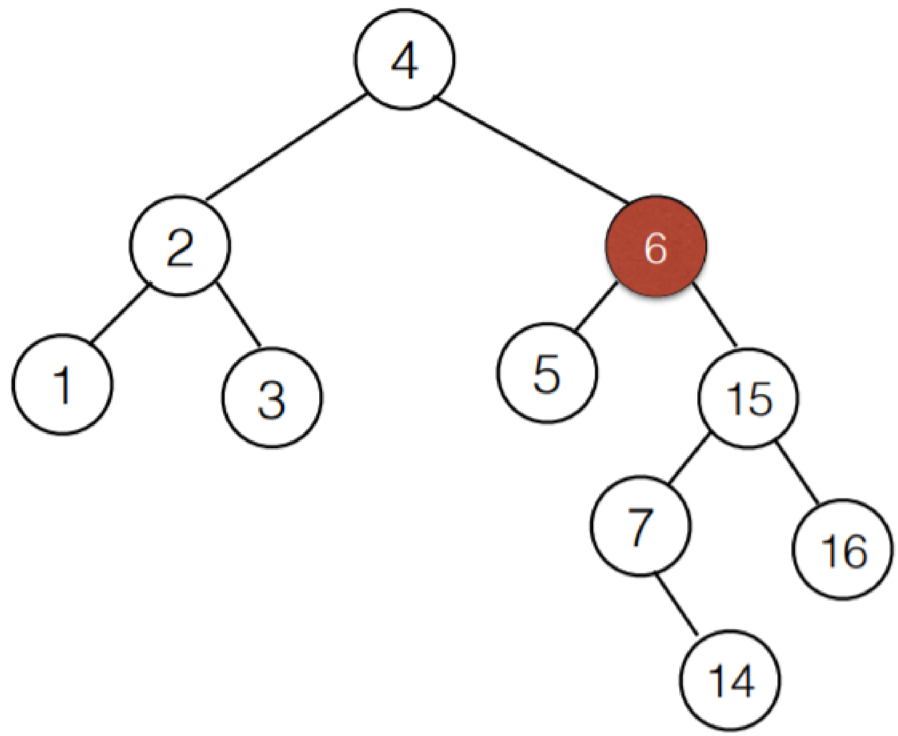
insert(7) rotate(7)



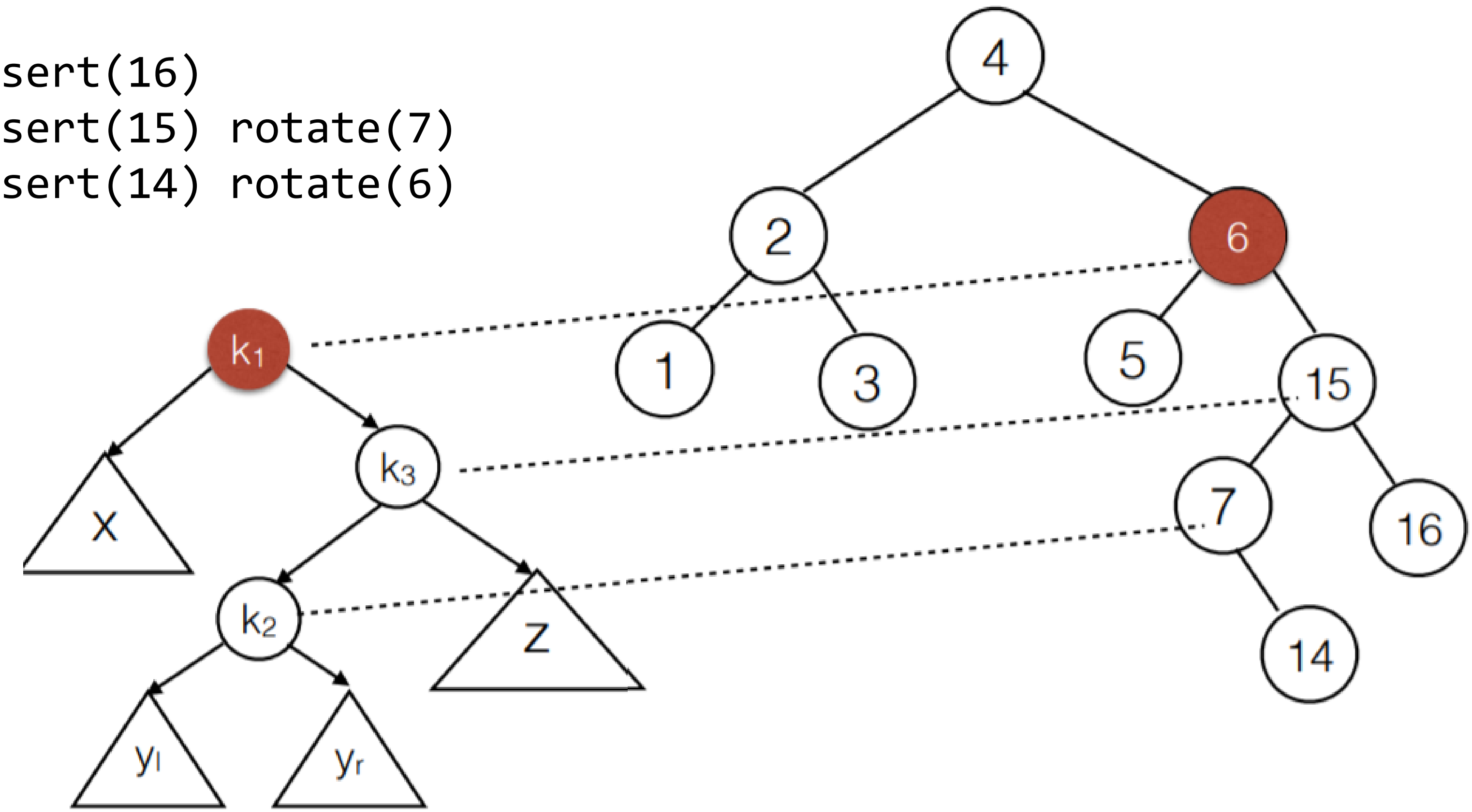
```
insert(16)
insert(15) rotate(7)
```

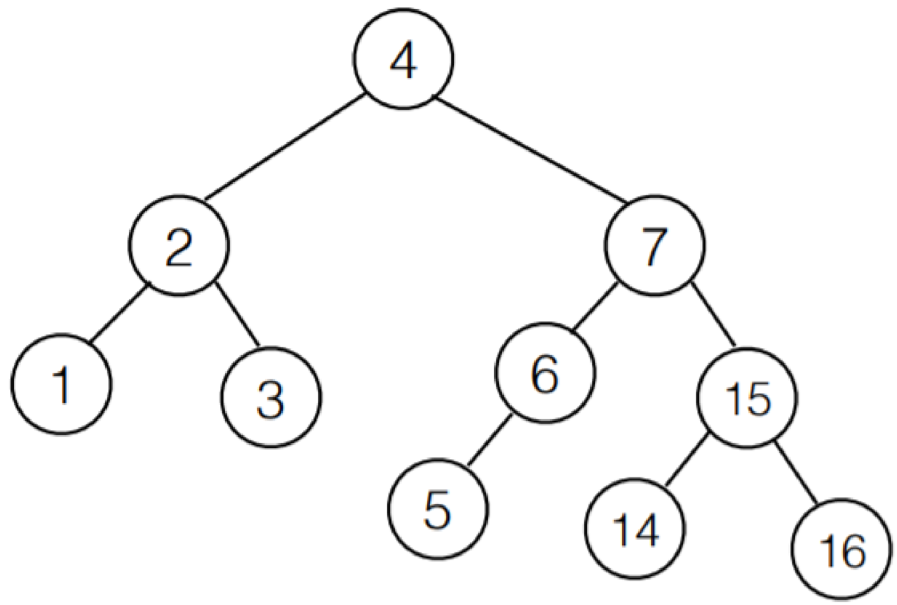


```
insert(16)
insert(15) rotate(7)
insert(14) rotate(6)
```



```
insert(16)
insert(15) rotate(7)
insert(14) rotate(6)
```




4
2
7
1
3
6
15
5
14
16