Balanced BSTs (AVL Trees)

Different Binary Search Trees from same data

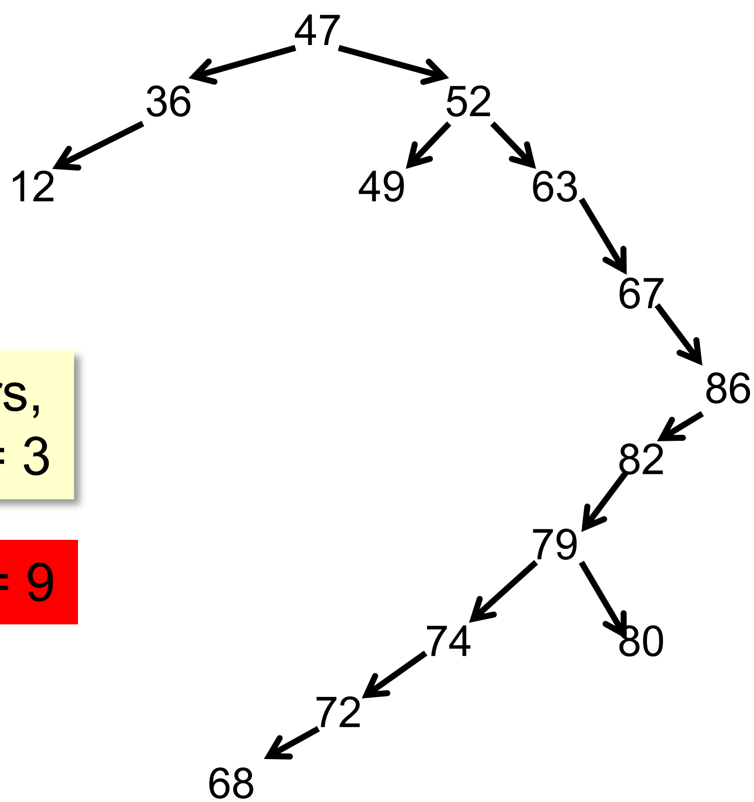
Rotating nodes in Binary Search Trees

Balanced Binary Search Trees
(or AVL Trees)

BST structure affects search …

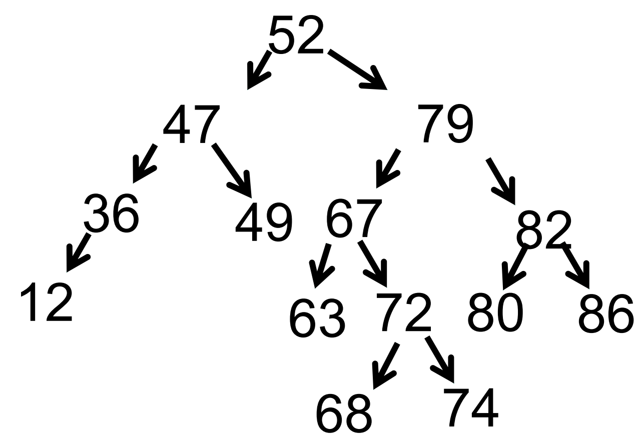
Our Binary Search Tree methods for adding and removing nodes tried to do as little work as possible for each individual operation.

But the structure of the BST has a significant impact on search -- search in a BST has complexity O(height of tree).



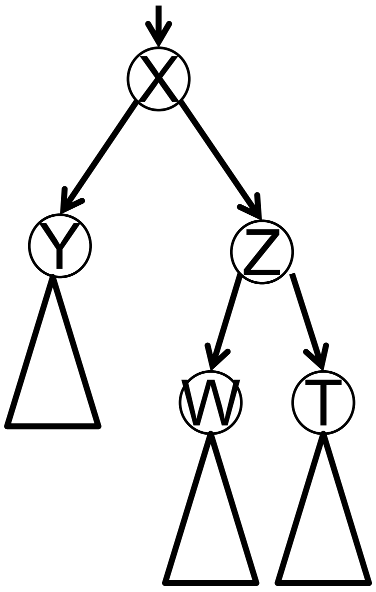
14 numbers, so $\lfloor \log n \rfloor = 3$

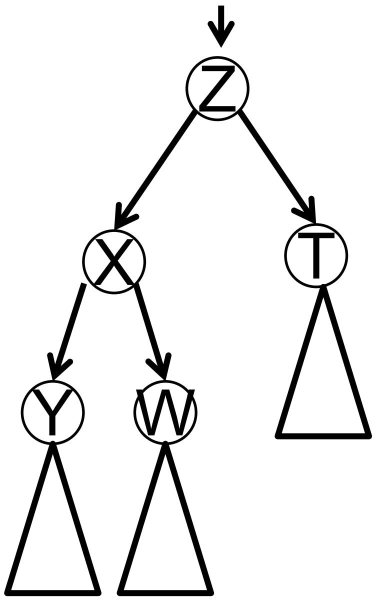
but depth = 9



this BST has the same content, but its height is only 4

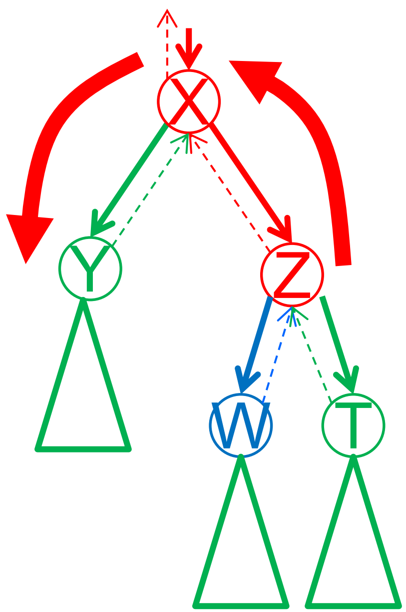
Alternative BSTs

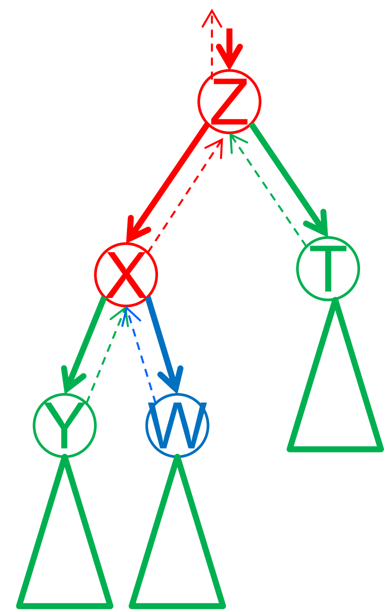




Y (and subtree) < X < W (and subtree) <Z < T (and subtree)

Rotation





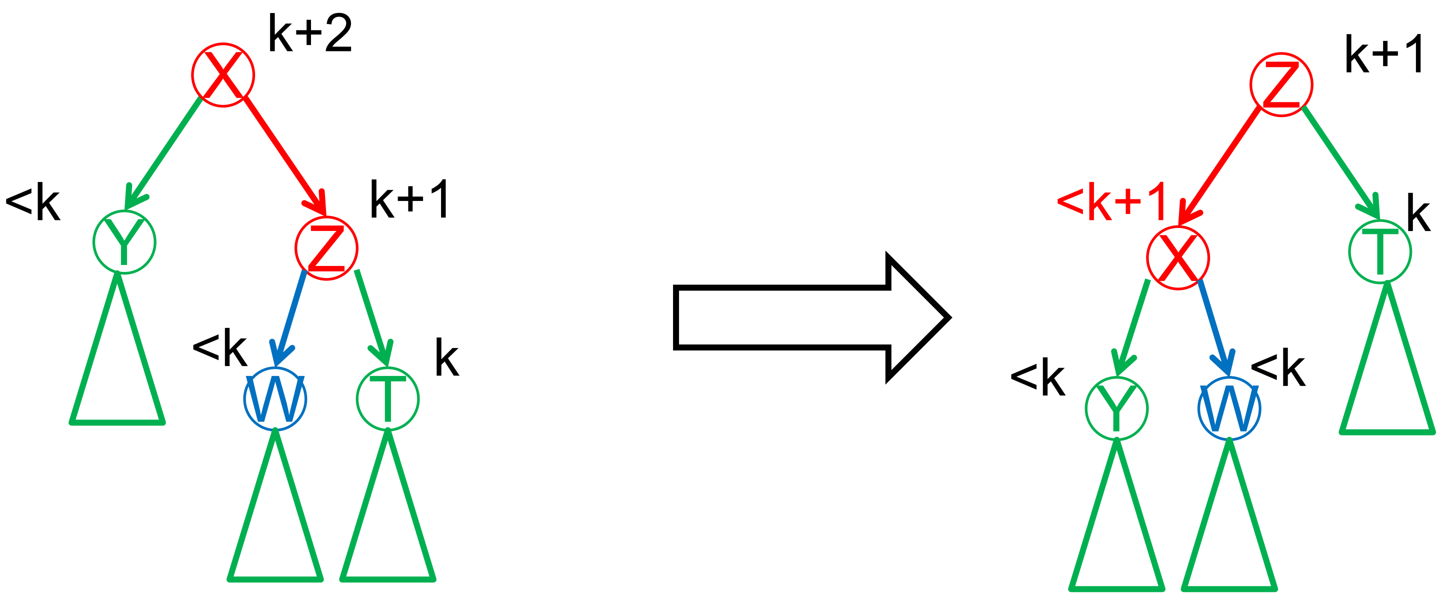
X.rightchild = Z.leftchild
Z.leftchild.parent = X
Z.leftchild = X
Z.parent = X.parent
X.parent.?child = Z
X.parent = Z

O(1) operations

also symmetric version

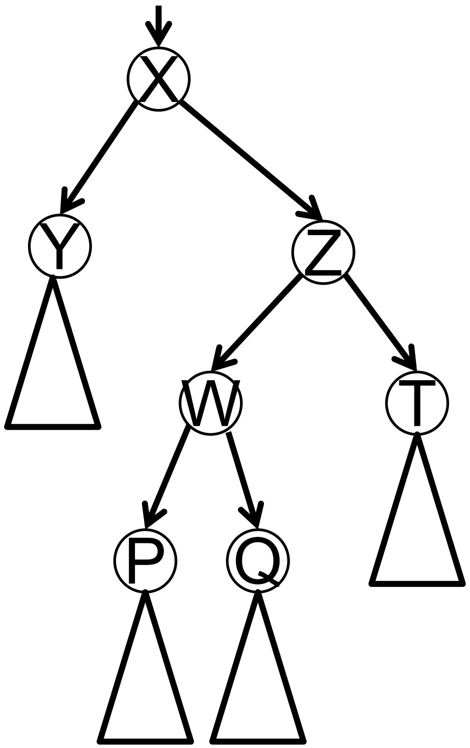
with appropriate
checks for None
and maybe for
outer Tree object

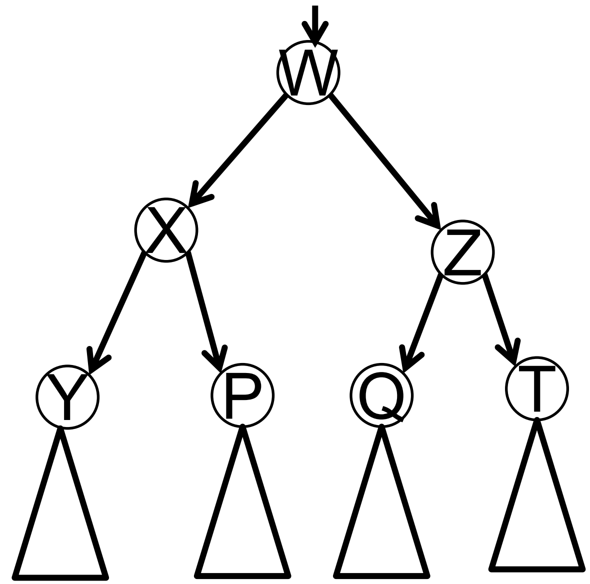
Rotation: height changes



If T causes Z's height, and Z's height - Y's height $\geq$ 2
then *X is unbalanced*, and this rotation will reduce the overall height of the tree

Alternative BSTs (ii)

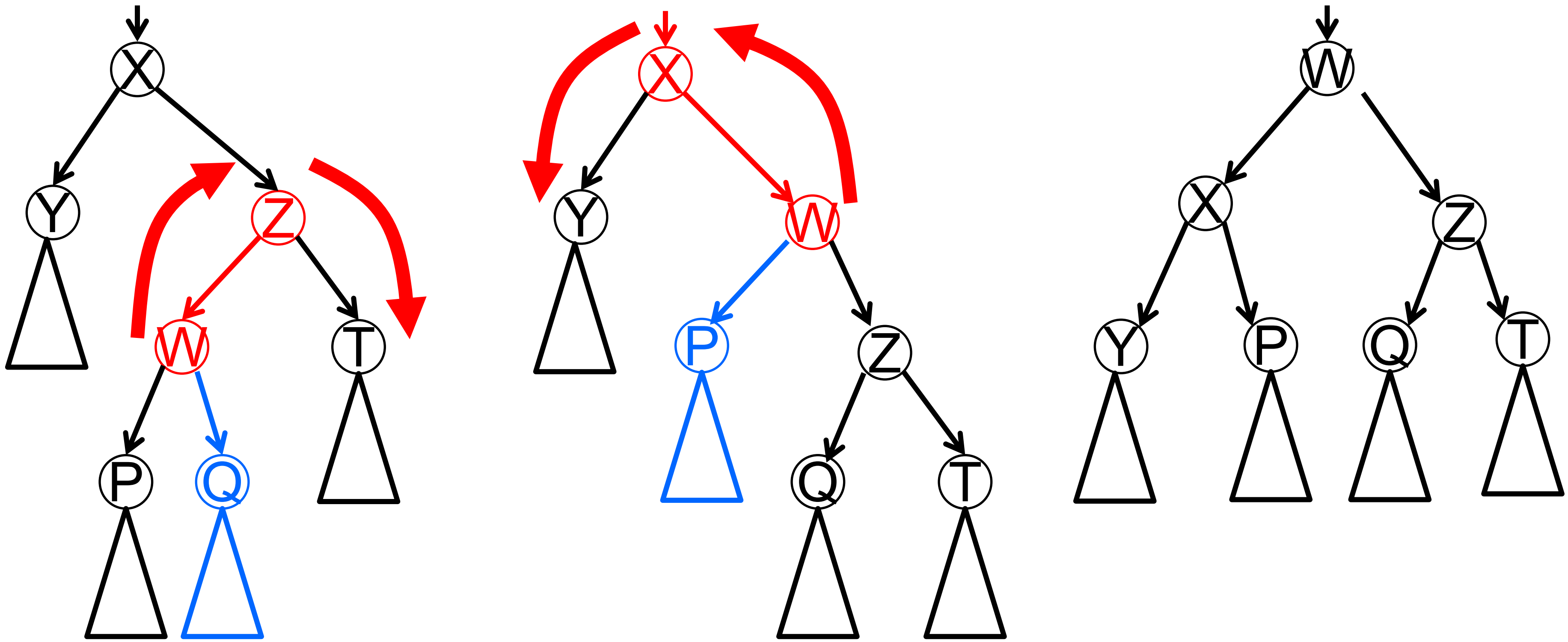




Y < X < P < W < Q < Z < T

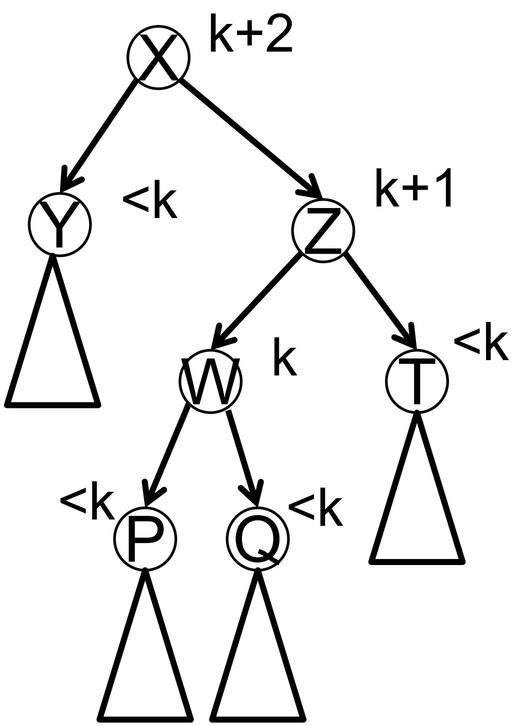
Double Rotation

drawing now without the parent links, to simplify the sketch ...

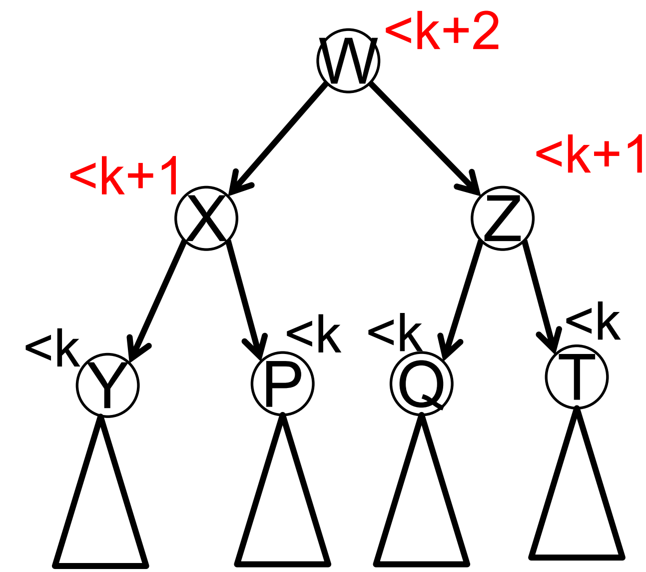


Two rotations, so also O(1)

Double Rotation: height changes



If W's height == T's height
no change to height
of tree, but new
tree has better
balance

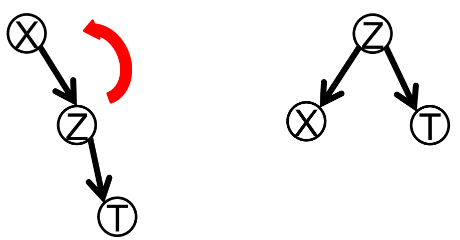


If W causes Z's height, and W’s height > T’s height, and Z's height - Y's height $\geq$ 2 then *X is unbalanced,* and this double rotation will reduce the overall height of the tree

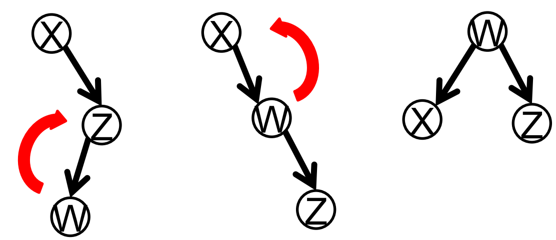
Simplified restructuring rules

A node X is *unbalanced* if children Y and Z are such that $|h_Z - h_Y| \geq 2$
or if X has only one child, and $h_x \geq 2$

If node X is unbalanced with higher rightchild Z
and Z's rightchild is its higher child
(i.e. height is caused by a straight line)
then rotate Z into X (from the right)

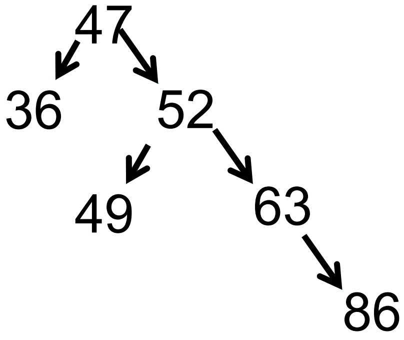


If node X is unbalanced with higher rightchild Z
and Z's leftchild W is its higher child (or equal)
(i.e. height is caused by a zig-zag)
then rotate W into Z (from the left)
rotate W into X (from the right)

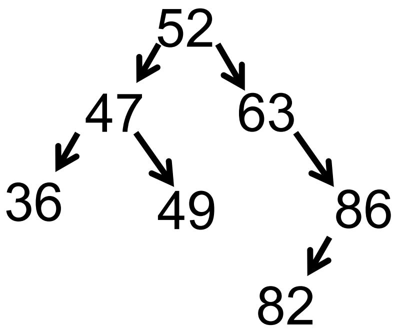


(and symmetric versions)

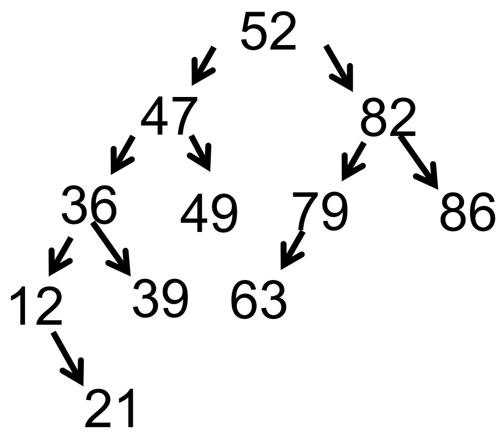
Restructure?



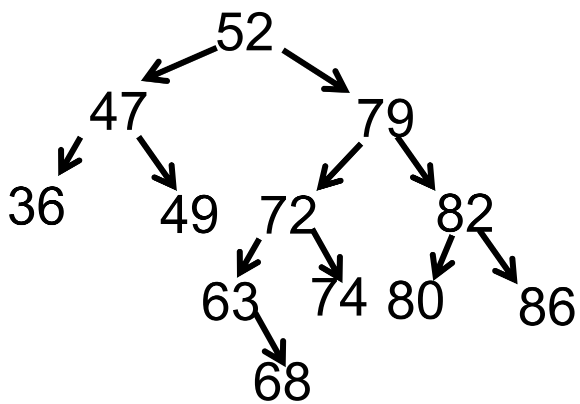
Restructure?



Restructure?



Restructure?



AVL Trees

An *AVL* Tree (or a *balanced* tree) is a Binary Search Tree in which:

- no node is unbalanced
- each time an item is added or removed from the tree, it is rebalanced by applying rotations.

Note that only ancestors of the (added / deleted / moved) node can become unbalanced, so we only need to search *up* the tree after each change.

Georgy Adelson-Velsky and Evgenii Landis (1962). "An algorithm for the organization of information". *Proceedings of the USSR Academy of Sciences* (in Russian) 146: 263–266.

AVL Trees: rebalance

After each *addition* of an item using the normal BST operation, start at the parent of the new item and rebalance it

maintain a height variable for each BSTNode

```
rebalance()
   update the height
   if node is unbalanced
      restructure the node
      if node had a parent before restructuring
         parent.rebalance()
   else
      if height changed and node has a parent
         parent.rebalance()
```

AVL Trees: rebalance (ii)

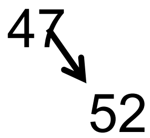
After each *removal* of an item using the normal BST operation,

```
if removed node was leaf
   leaf.parent.rebalance()
else if removed node was semileaf
   semileaf.parent.rebalance()
else   #internal, moved 'biggest' up, removed biggest
   node = the original parent of biggest
   node.rebalance()
```

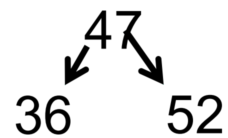
add 47,52,36,63,49,67,86,82,79,80,12,74,72,68

47

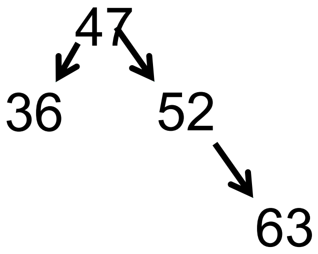
add 47,52,36,63,49,67,86,82,79,80,12,74,72,68



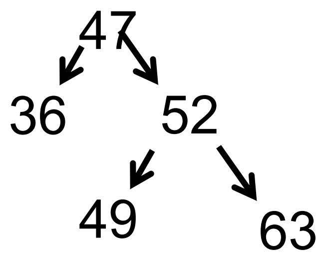
add 47,52,36,63,49,67,86,82,79,80,12,74,72,68



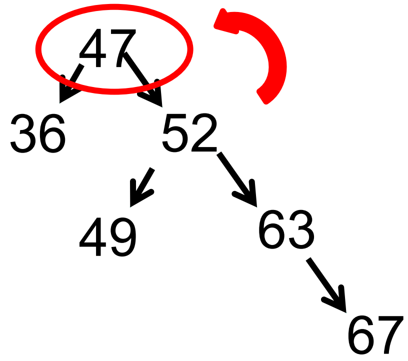
add 47,52,36,63,49,67,86,82,79,80,12,74,72,68



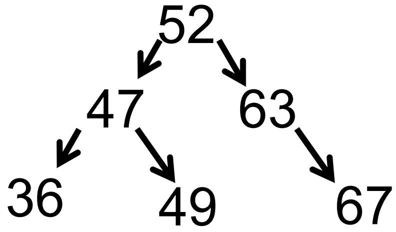
add 47,52,36,63,49,67,86,82,79,80,12,74,72,68



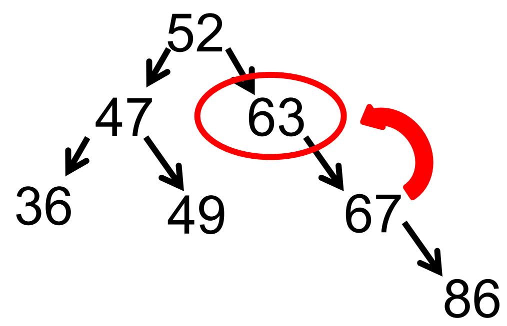
add 47,52,36,63,49,67,86,82,79,80,12,74,72,68



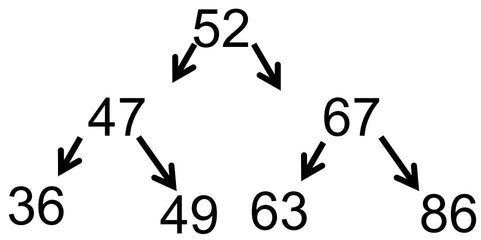
add 47,52,36,63,49,67,86,82,79,80,12,74,72,68



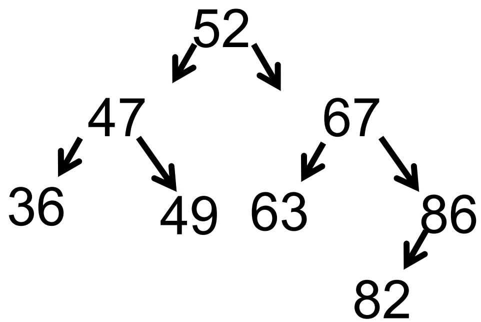
add 47,52,36,63,49,67,86,82,79,80,12,74,72,68



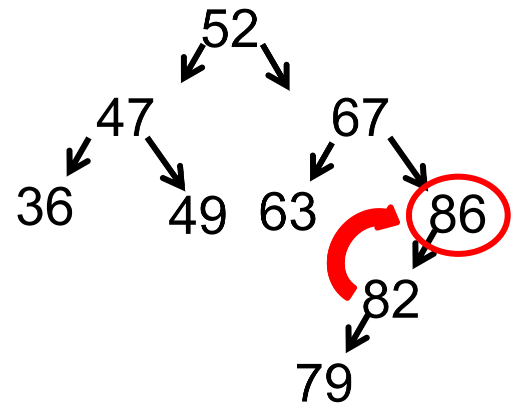
add 47,52,36,63,49,67,86,82,79,80,12,74,72,68



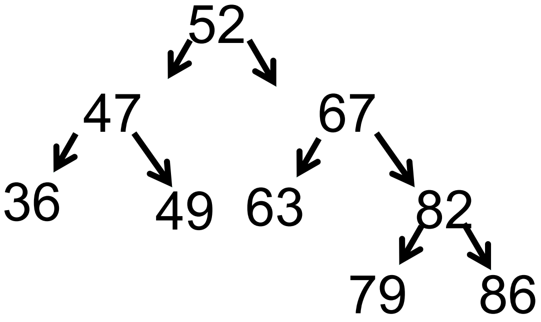
add 47,52,36,63,49,67,86,82,79,80,12,74,72,68



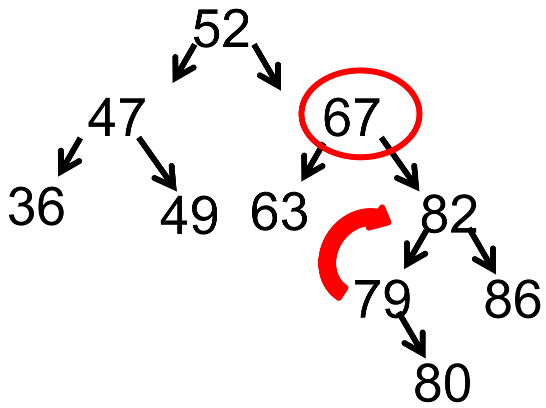
add 47,52,36,63,49,67,86,82,79,80,12,74,72,68



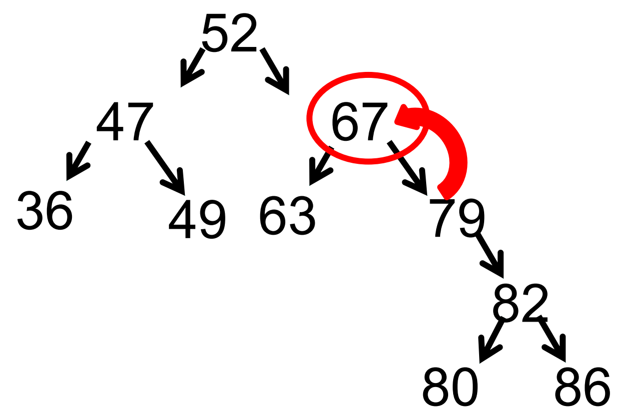
add 47,52,36,63,49,67,86,82,79,80,12,74,72,68



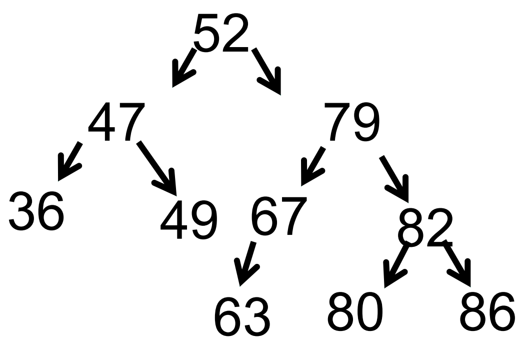
add 47,52,36,63,49,67,86,82,79,80,12,74,72,68



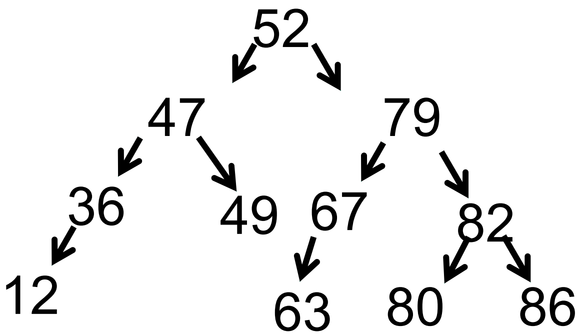
add 47,52,36,63,49,67,86,82,79,80,12,74,72,68



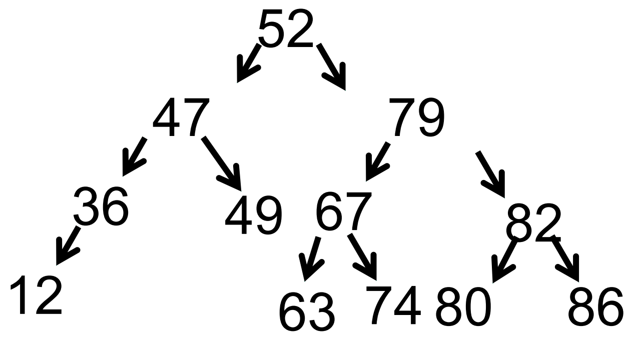
add 47,52,36,63,49,67,86,82,79,80,12,74,72,68



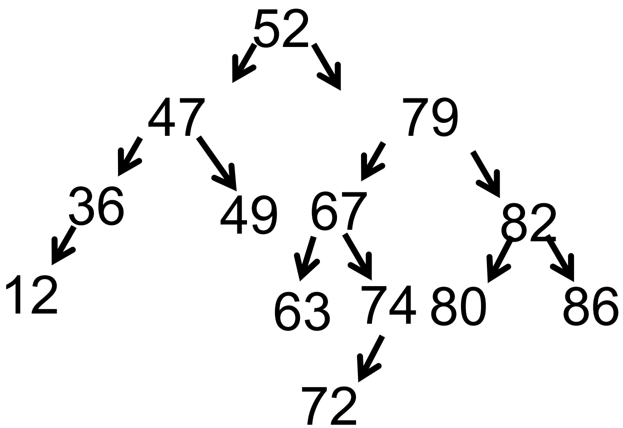
add 47,52,36,63,49,67,86,82,79,80,12,74,72,68



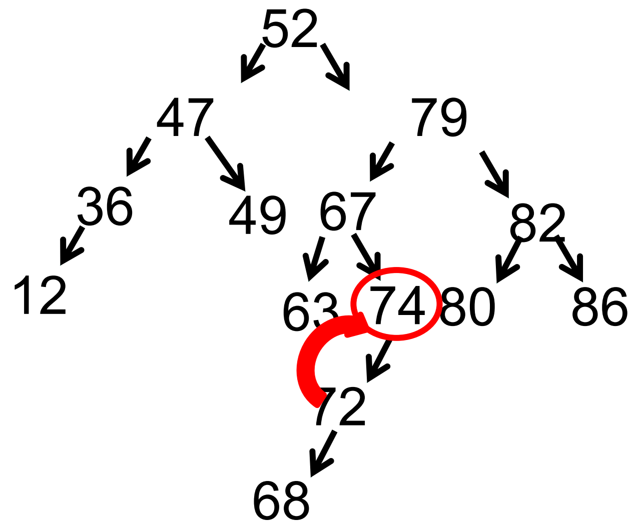
add 47,52,36,63,49,67,86,82,79,80,12,74,72,68



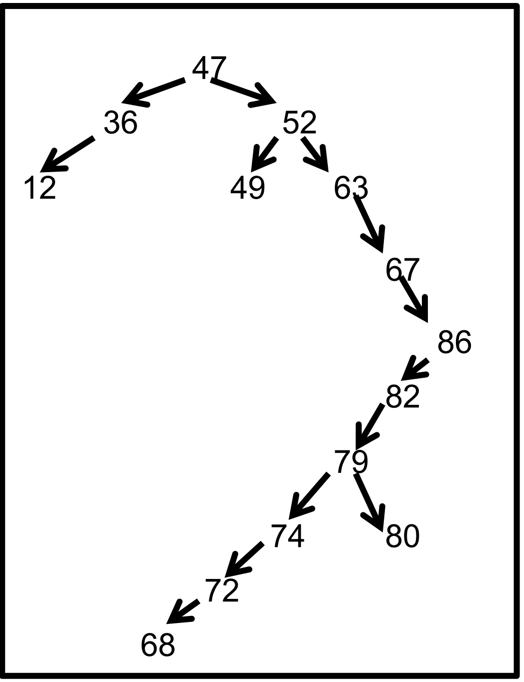
add 47,52,36,63,49,67,86,82,79,80,12,74,72,68

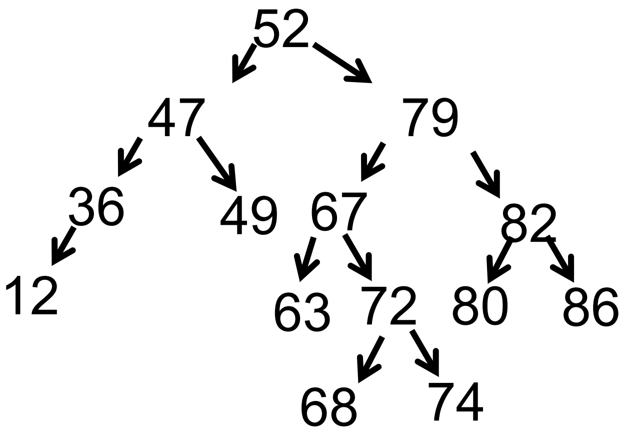


add 47,52,36,63,49,67,86,82,79,80,12,74,72,68

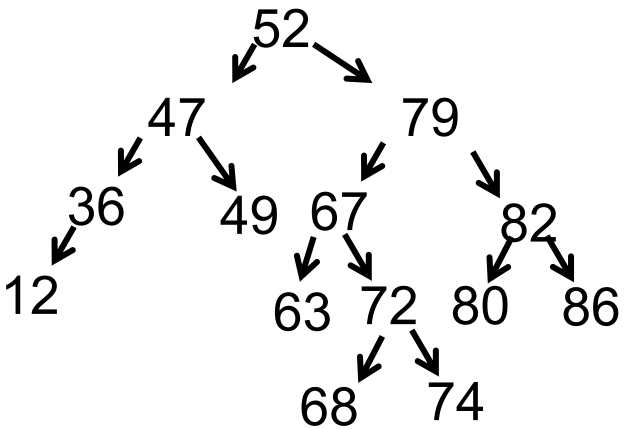


add 47,52,36,63,49,67,86,82,79,80,12,74,72,68

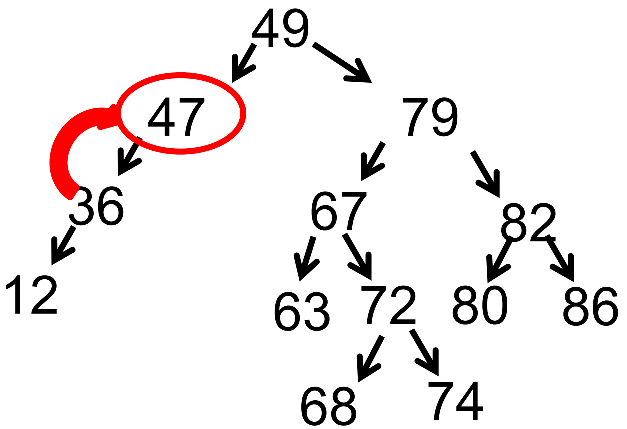




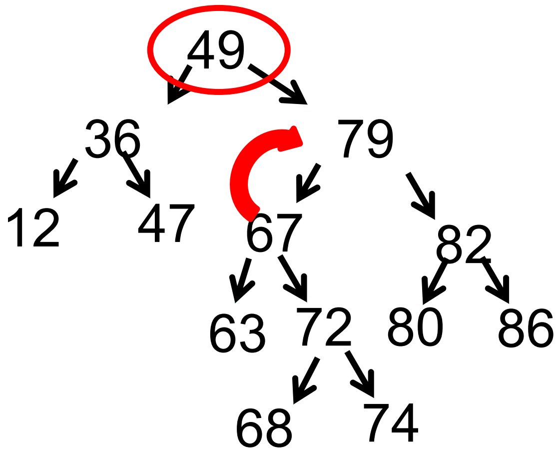
remove 52



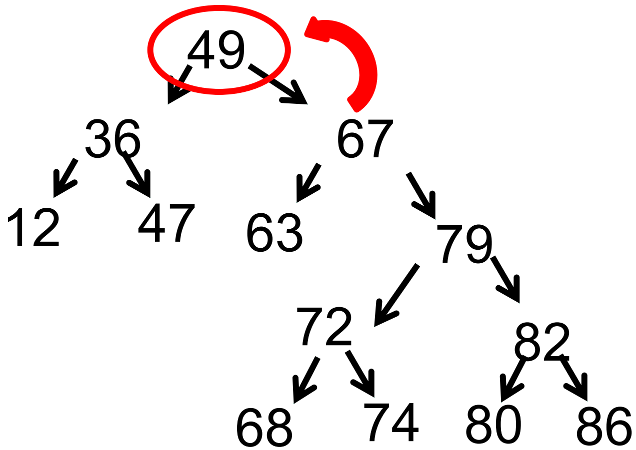
remove 52



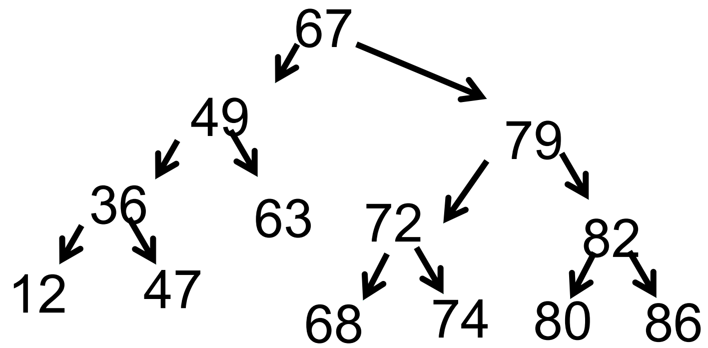
remove 52



remove 52



remove 52



AVL Tree Properties

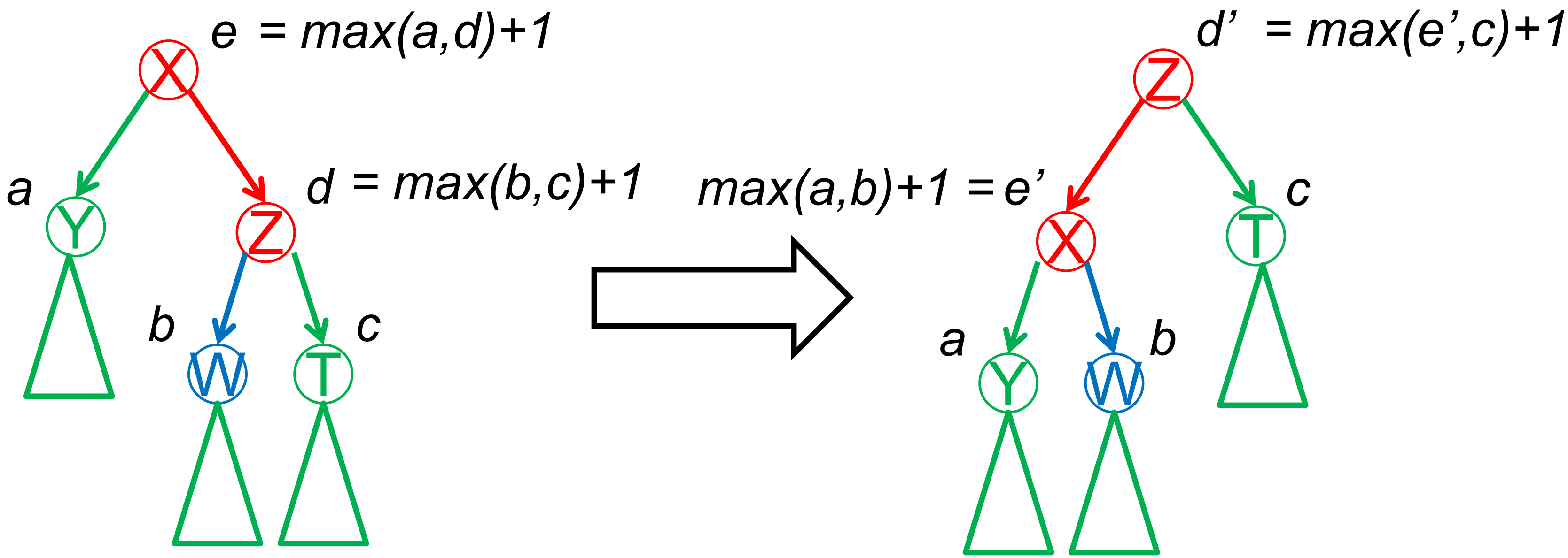
the height of an AVL Tree for n items is O(log n)

will be proved later

search is O(log n)
add is O(log n)
delete is O(log n)
find_min, find_max is O(log n)
traversing all nodes is O(n)

Derivation of height change for single rotation

If e > d’, then this rotation reduces the height of the tree



height e = max(a+1, b+2, c+2)

height d’ = max(a+2, b+2, c+1)

We need max(a+1, b+2, c+2) > max(a+2, b+2, c+1)

Cases: (i) a+1 was the max. No good, since a+2 is the new max, and e < d’

(ii) b+2 was the max. No good, since new max ≥ b+2, and e ≤ d’

(iii) c+2 was the max. If c+2 ≤ a+2, No good, since e ≤ d’

So (iii) is only option, and c+2 > a+2, and c+2 > b+2. i.e. c>a and c>b

Next lecture

more ADTs