AVL Trees: Insertion

Visualization and practice

https://www.cs.usfca.edu/~galles/visualization/AVLtree.html

Learning Objectives

- Implement AVL trees.
- Understand the cases required for rebalancing algorithms.

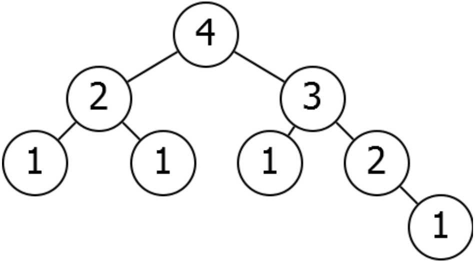
AVL Tree:

AVL tree is a self-balancing Binary Search Tree (**BST**) where the difference between heights of left and right subtrees cannot be more than **one** for all nodes.

Outline

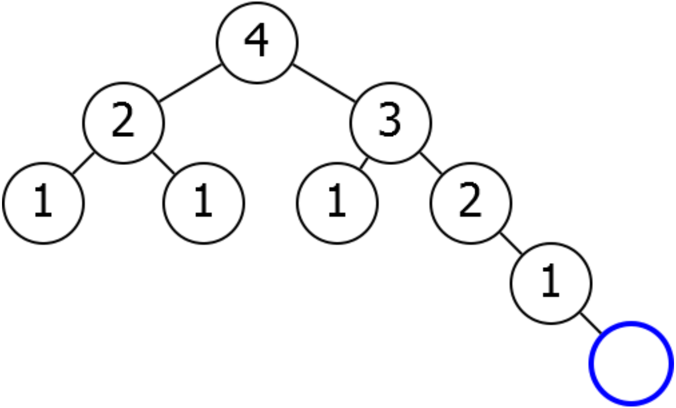
AVL Trees

Need ensure that children have nearly the same height.



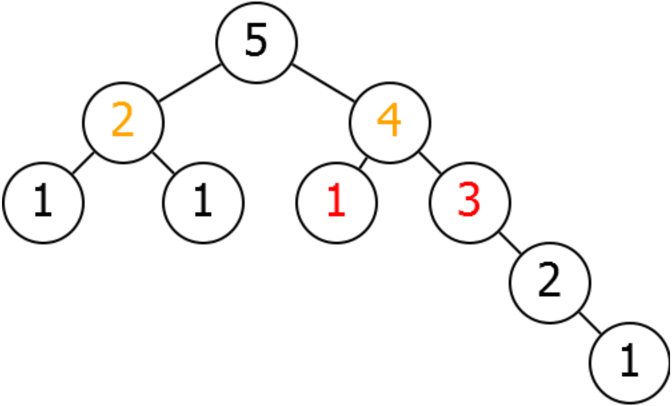
Problem

Updates to the tree can destroy this property.



Problem

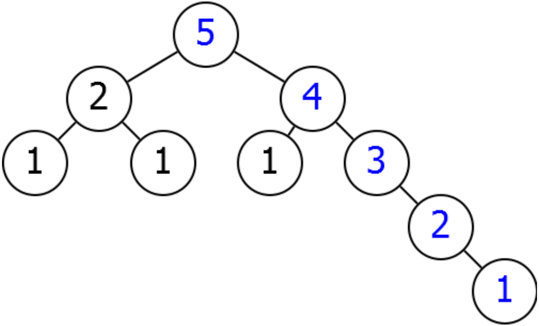
Updates to the tree can destroy th s property.



Need to correct this.

Errors

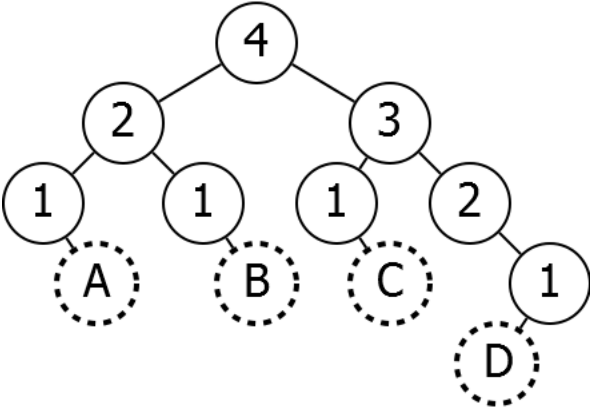
Heights stay the same except on the insert on path.



Only need to worry about th s path.

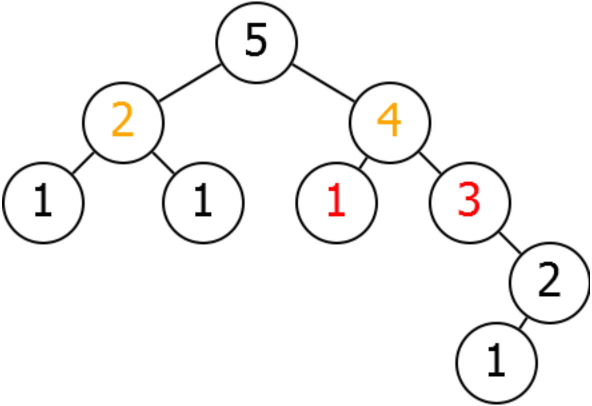
Problem

Which insertion would require the tree to be rebalanced in order to maintain the AVL property



Problem

Which insertion would require the tree to be rebalanced in order to maintain the AVL property?



Outline

Insertion

We need a new insert on algorithm that involves rebalancing the tree to maintain the AVL property.

Idea

AVLInsert(k, R)

Insert(k, R)
$N \leftarrow$ Find(k, R)
Rebalance(N)

AVL Trees

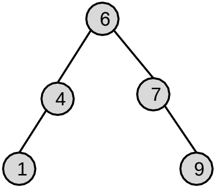
- All nodes in an AVL tree have a Balance Factor (BF)
- Balance factor of a node = height of the left subtree minus the height of the right subtree
  - $BF = h_L - h_R$
  - Or $BF = h_R - h_L$

- An AVL tree can have only
- balance factors of -1, 0, or 1 at every node
- For every node in a BST, the height of the left and right subtrees can differ by no more than 1

AVL Trees

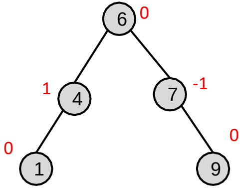
- $BF = h_L - h_R$

BF of the root: 2 - 2 = 0

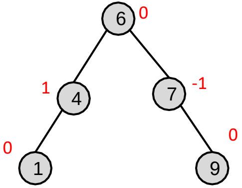


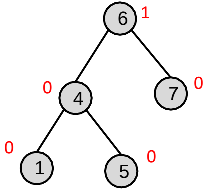
AVL Trees

- $BF = h_L - h_R$

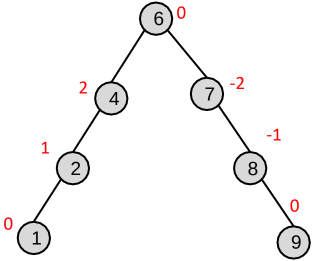


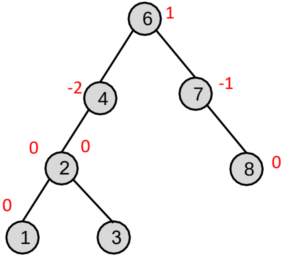
AVL Trees





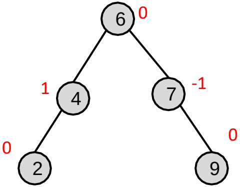
Non - AVL Trees





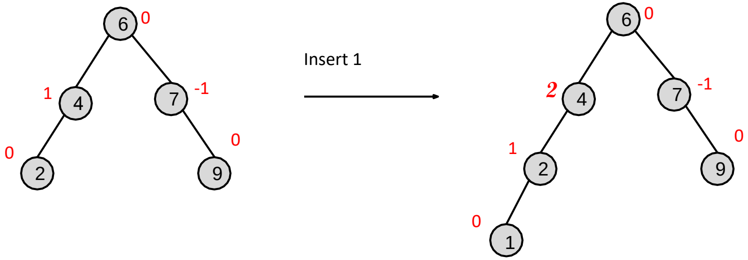
AVL Trees

Insert and delete may cause the tree to be unbalanced!



AVL Trees

Insert and delete may cause the tree to be unbalanced!



Steps to follow for insertion:

Let the newly inserted node be **w**

- Perform standard **BST** insert for **w**.
- Starting from **w**, travel up and find the first **unbalanced node**. Let **z** be the first unbalanced node, **y** be the **child** of **z** that comes on the path from **w** to **z** and **x** be the **grandchild** of **z** that comes on the path from **w** to **z**.
- Re-balance the tree by performing appropriate rotations on the subtree rooted with **z.** There can be 4 possible cases that need to be handled as **x, y** and **z** can be arranged in 4 ways.
- Following are the possible 4 arrangements:
  - y is the left child of z and x is the left child of y (Left Left Case)
  - y is the left child of z and x is the right child of y (Left Right Case)
  - y is the right child of z and x is the right child of y (Right Right Case)
  - y is the right child of z and x is the left child of y (Right Left Case)

Rotation

```
T1, T2 and T3 are subtrees of the tree, rooted with y (on the left side) or x (on the right side)

     y                               x
    / \     Right Rotation          /  \
   x   T3   - - - - - - - >        T1   y
  / \       < - - - - - - -            / \
 T1  T2     Left Rotation            T2  T3

Keys in both of the above trees follow the following order
keys(T1) < key(x) < keys(T2) < key(y) < keys(T3)
So BST property is not violated anywhere.
```

1. Left Left Case

```
T1, T2, T3 and T4 are subtrees.
         z                                      y
        / \                                   /   \
       y   T4      Right Rotate (z)          x      z
      / \          - - - - - - - - ->      /  \    /  \
     x   T3                               T1  T2  T3  T4
    / \
  T1   T2
```

2. Left Right Case

```
     z                               z                           x
    / \                            /   \                        /  \
   y   T4  Left Rotate (y)        x    T4  Right Rotate(z)    y      z
  / \      - - - - - - - - ->    /  \      - - - - - - - ->  / \    / \
T1   x                          y    T3                    T1  T2 T3  T4
    / \                        / \
  T2   T3                    T1   T2
```

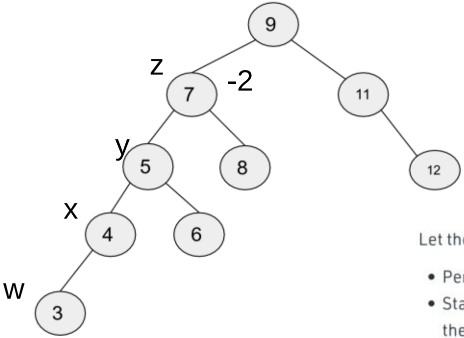
3. Right Right Case

```
  z                                y
 /  \                            /   \
T1   y     Left Rotate(z)       z      x
    /  \   - - - - - - - ->    / \    / \
   T2   x                     T1  T2 T3  T4
       / \
     T3  T4
```

4. Right Left Case

```
   z                            z                            x
  / \                          / \                          /  \
T1   y   Right Rotate (y)    T1   x      Left Rotate(z)   z      y
    / \  - - - - - - - - ->     /  \   - - - - - - - ->  / \    / \
   x   T4                      T2   y                  T1  T2  T3  T4
  / \                              /  \
T2   T3                           T3   T4
```

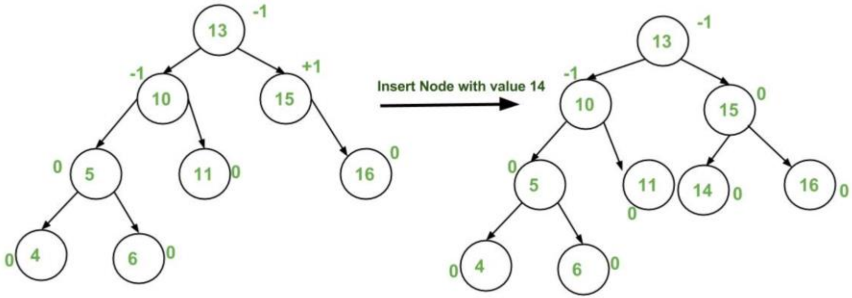
Example 0: Rebalancing



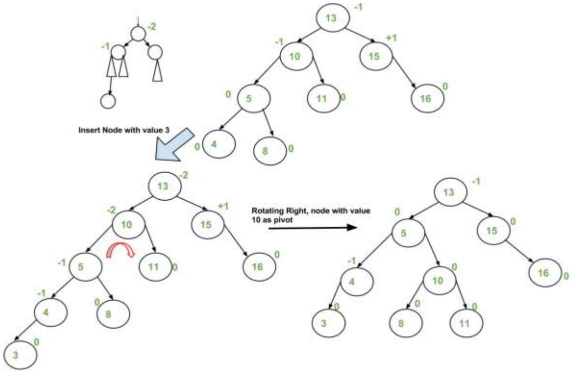
Let the newly inserted node be **w**

- Perform standard **BST** insert for **w**.
- Starting from **w**, travel up and find the first **unbalanced node**. Let **z** the first unbalanced node, **y** be the **child** of **z** that comes on the path **w** to **z** and **x** be the **grandchild** of **z** that comes on the path from **w** to
- Re-balance the tree by performing appropriate rotations on the sub rooted with **z.** There can be 4 possible cases that need to be handle **x, y** and **z** can be arranged in 4 ways.
- Following are the possible 4 arrangements:
  - y is the left child of z and x is the left child of y (Left Left Case)
  - y is the left child of z and x is the right child of y (Left Right Case
  - y is the right child of z and x is the right child of y (Right Right Ca
  - y is the right child of z and x is the left child of y (Right Left Case

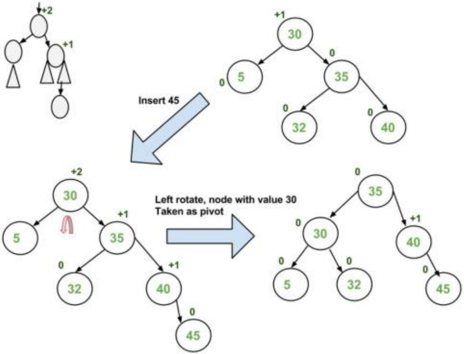
Example 1: Insertion



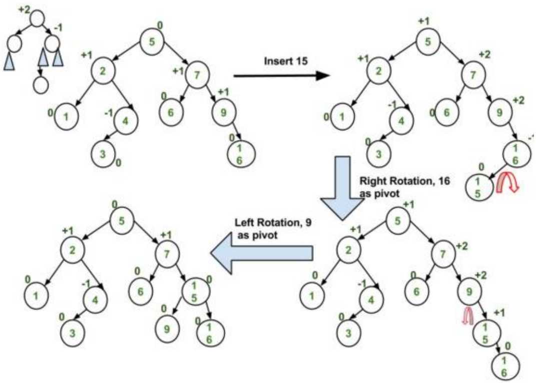
Example 2: Insertion



Example 3: Insertion



Example : Insertion



Code

```
Rebalance(N)

P ← N.Parent
if N.Left.Height > N.Right.Height+1:
  RebalanceRight(N)
if N.Right.Height > N.Left.Height+1:
  RebalanceLeft(N)
AdjustHeight(N)
if P ≠ null:
  Rebalance(P)
```

Adjust Height

```
AdjustHeight(N)

    N.Height ← 1 + max(
                      N.Left.Height,
                      N.Right.Height)
```

Rebalance

```
RebalanceRight(N)
M ← N.Left
if M.Right.Height > M.Left.Height:
  RotateLeft(M)
RotateRight(N)
AdjustHeight on affected nodes
```

Rebalance

```
RebalanceLeft(N)
```

$M \leftarrow N$.Right

if M.Left.Height > M.Right.Height:

  RotateRight(M)

RotateLeft(N)

AdjustHeight on affected nodes

AVLInsert(k, R)

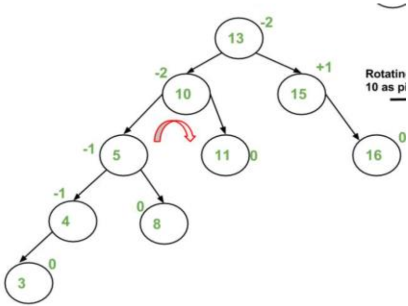
```
Insert(k, R)
N ← Find(k, R)
Rebalance(N)
```

AdjustHeight(N)

```
N.Height ← 1+ max(
        N.Left.Height,
        N.Right.Height)
```

Rebalance(N)

```
P ← N.Parent
if N.Left.Height > N.Right.Height+1:
    RebalanceRight(N)
if N.Right.Height > N.Left.Height+1:
    RebalanceLeft(N)
AdjustHeight(N)
if P ≠ null:
    Rebalance(P)
```



RebalanceRight(N)

```
M ← N.Left
if M.Right.Height > M.Left.Height:
    RotateLeft(M)
RotateRight(N)
AdjustHeight on affected nodes
```

RebalanceLeft(N)

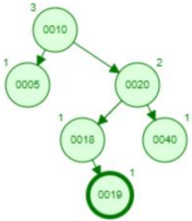
```
M ← N.Right
if M.Left.Height > M.Right.Height:
    RotateRight(M)
RotateLeft(N)
AdjustHeight on affected nodes
```

AVLInsert(k, R)

Insert(k, R)
$N \leftarrow$ Find(k, R)
Rebalance(N)

AdjustHeight(N)

N.Height $\leftarrow 1+\max($
N.Left.Height,
N.Right.Height$)$



Rebalance(N)

$P \leftarrow N$.Parent
if N.Left.Height $> N$.Right.Height+1:
  RebalanceRight(N)
if N.Right.Height $> N$.Left.Height+1:
  RebalanceLeft(N)
AdjustHeight(N)
if $P \neq null$:
  Rebalance(P)

Use the algorithms to rebalancing the tree

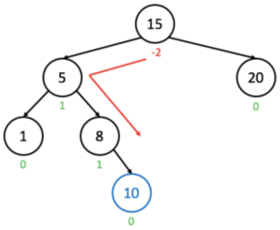
RebalanceRight(N)

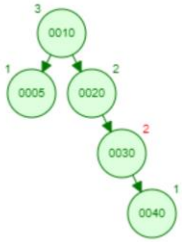
$M \leftarrow N$.Left
if M.Right.Height $> M$.Left.Height:
  RotateLeft(M)
RotateRight(N)
AdjustHeight on affected nodes

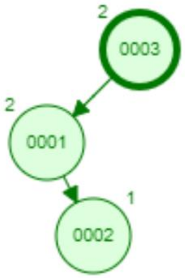
RebalanceLeft(N)

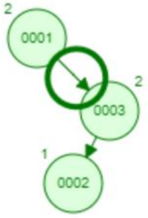
$M \leftarrow N$.Right
if M.Left.Height $> M$.Right.Height:
  RotateRight(M)
RotateLeft(N)
AdjustHeight on affected nodes

Use the algorithms to rebalancing the following trees





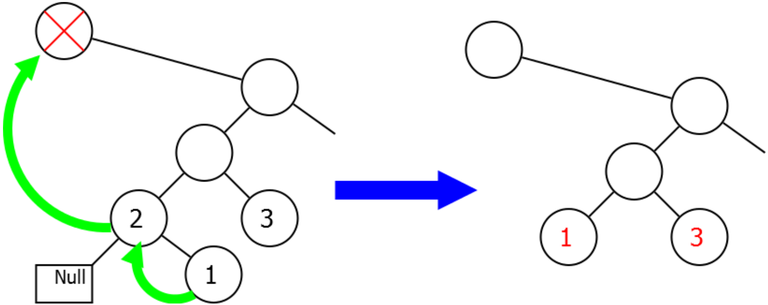




Outline

Delete

Delet ons can also change balance.

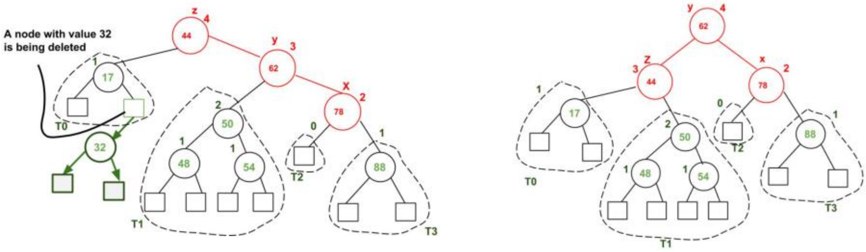


Delete procedure: Similar than insert() different in terms of how to place y, x.

Let w be the node to be deleted

1. Perform standard BST delete for w.
2. Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the larger height child of z, and x be the larger height child of y. Note that the definitions of x and y are different from insertion here.
3. Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that needs to be handled as x, y and z can be arranged in 4 ways. Following are the possible 4 arrangements:
   1. y is left child of z and x is left child of y (Left Left Case)
   2. y is left child of z and x is right child of y (Left Right Case)
   3. y is right child of z and x is right child of y (Right Right Case)
   4. y is right child of z and x is left child of y (Right Left Case)

Example: Delete



A node with value 32 is being deleted. After deleting 32, we travel up and find the first unbalanced node which is 44. We mark it as z, its higher height child as y which is 62, and y's higher height child as x which could be either 78 or 50 as both are of same height. We have considered 78. Now the case is Right Right, so we perform left rotation.

[Practice here: herehttps://www.cs.usfca.edu/~galles/visualization/AVLtree.html](https://www.cs.usfca.edu/~galles/visualization/AVLtree.html)

New Delete

```
AVLDelete(N)
```

```
Delete(N)
M ← Parent of node replacing N
Rebalance(M)
```

Conclusion

Summary

AVL trees can implement all of the basic operations n $O(\log(n))$ time per operation.