Self-balancing tree proposed by Adelson-Velsky and Landis

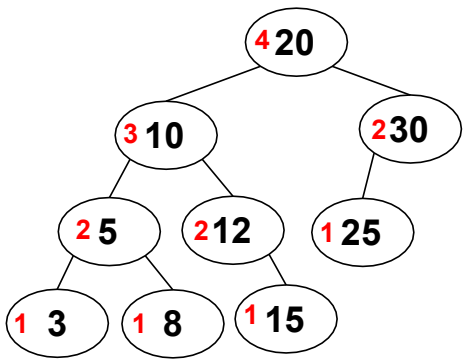
AVL TREES

AVL Trees

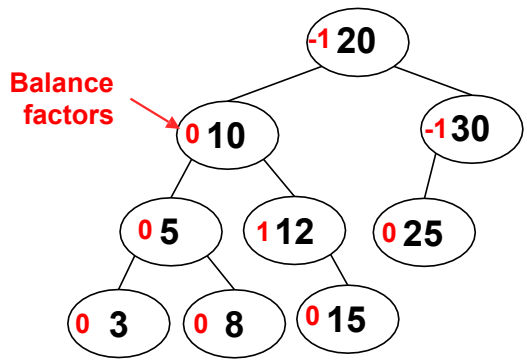
- AVL trees are binary trees such that

1) The Binary Search Tree (BST) Property holds: Left subtree keys are less than the root and right subtree keys are greater

2) **The height-balance property** holds: **height difference** between left and right subtrees of a node is **at most 1**



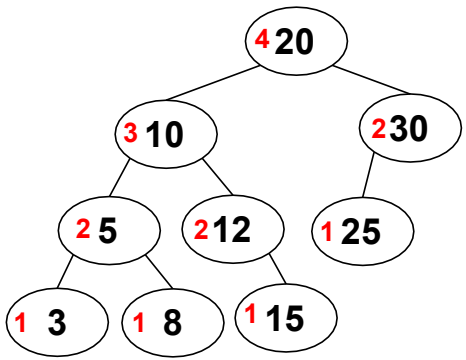
AVL Tree storing Heights



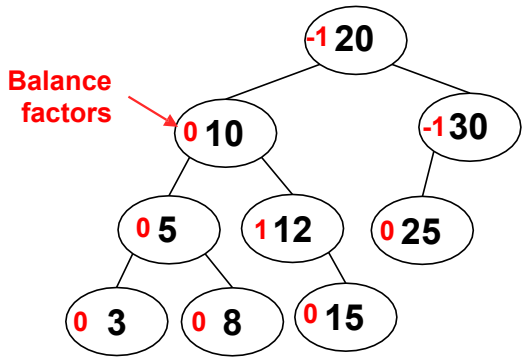
AVL Tree storing balances

AVL Trees

- Two implementations:
  - Height: Just store the height of the tree rooted at that node
  - **Balance: Define b(n) as the balance of a node = Height(right) –Height(left)**
    - Legal values are -1, 0, 1
    - Balances require at most 2-bits if we are trying to save memory.
    - **We will use balance.**



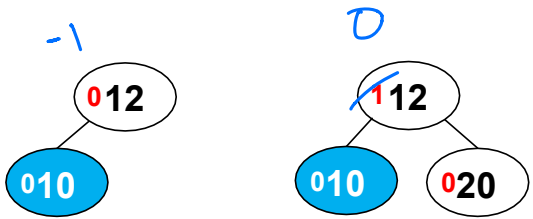
AVL Tree storing Heights



AVL Tree storing balances

Adding a New Node

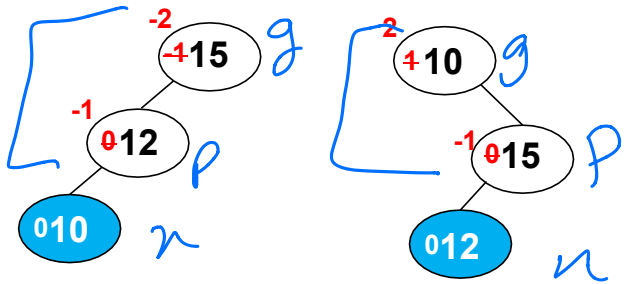
- A balanced parent node cannot be made out of balance by adding a child node
- What can happen now if we add a node at 10?



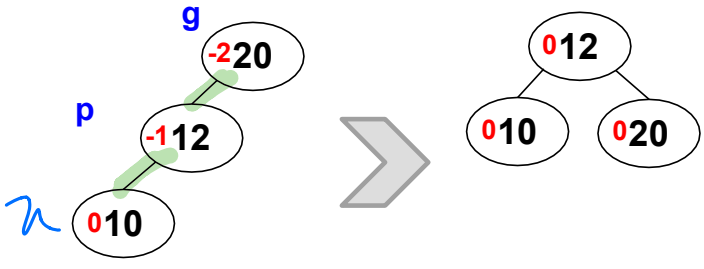
Losing Balance

- A grandparent node can lose balance.
- To fix, we will need rotations
- The rotations required to balance a tree are dependent on the grandparent, parent, child relationships

g p n



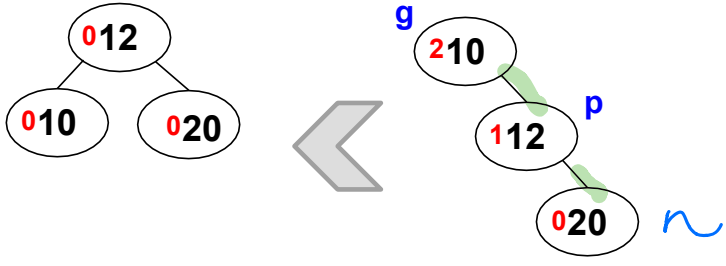
Single Rotation



p == g->left && n == p->left

- If the parent is left child of grandparent and child is left child of parent -> rotateRight(g, p)
- This pattern is called a zig-zig

Single Rotation



p == g->right && n==p->right

- If the parent is right child of grandparent and child is right child of parent -> rotateLeft(g,p)

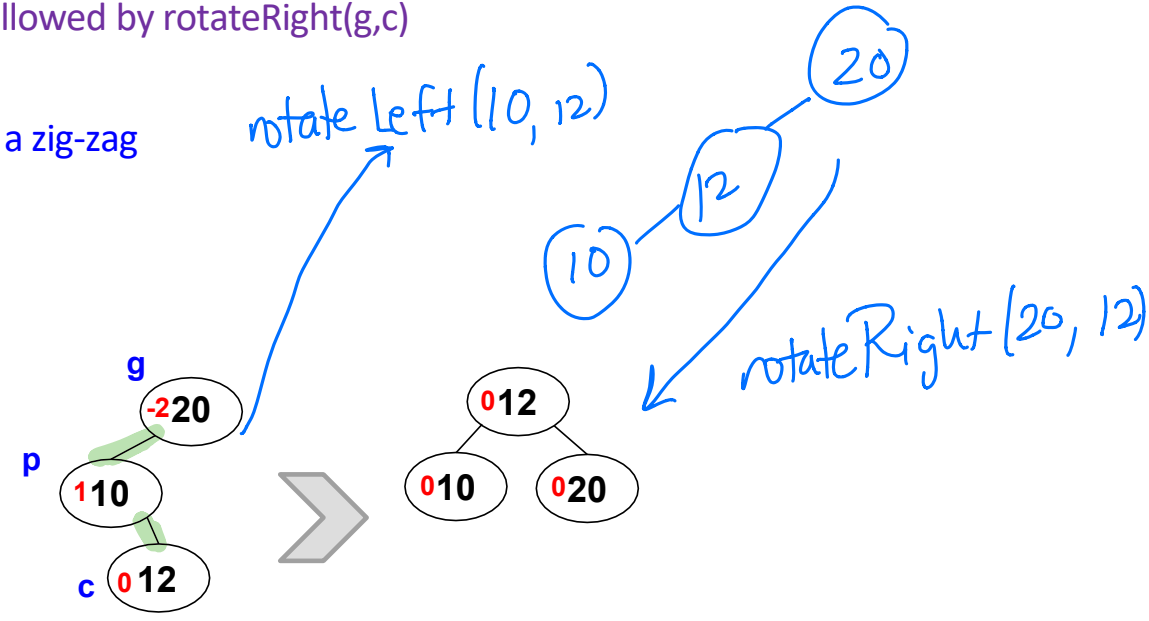
This pattern is called a zig-zig

Double Rotations

p == g -> left && c == p -> right

- If the parent is left child of grandparent and child is right child of parent -> rotateLeft(p,c) followed by rotateRight(g,c)

This pattern is called a zig-zag

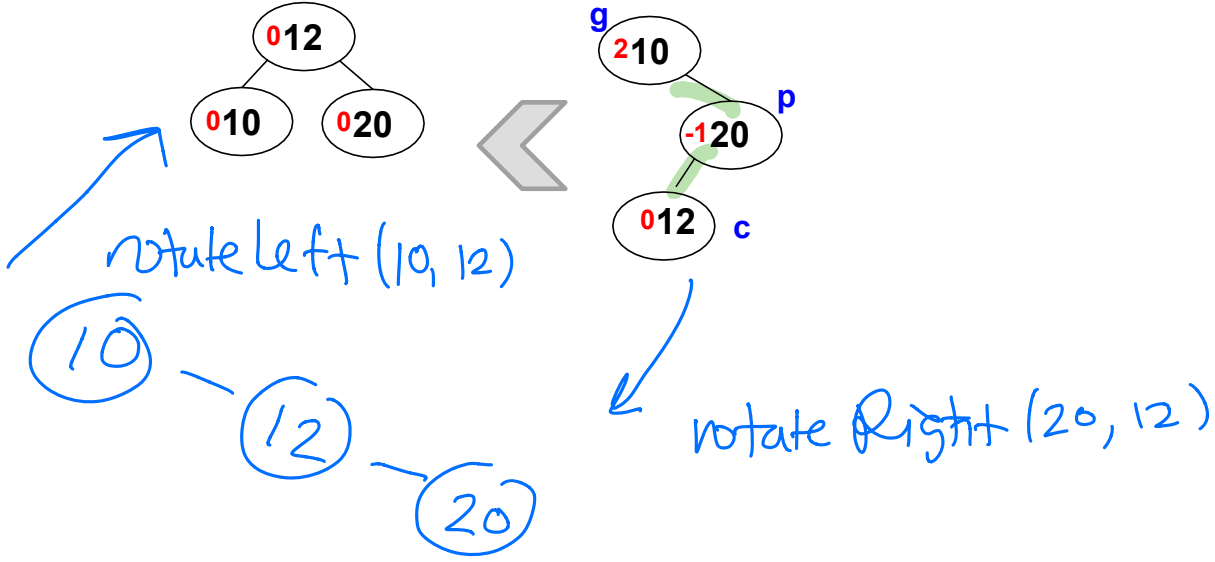


Double Rotations

p == g->right && c == p->left;

- If the parent is right child of grandparent and child is left child of parent -> rotateRight(p,c) followed by rotateLeft(g,c)

This pattern is called a zig-zag



AVL Insert(n)

- If empty tree => set n as root, b(n) = 0, done!
- **BST insert(n)**
- Set balance of n: b(n) = 0
- Set balance of parent node p, b(p):

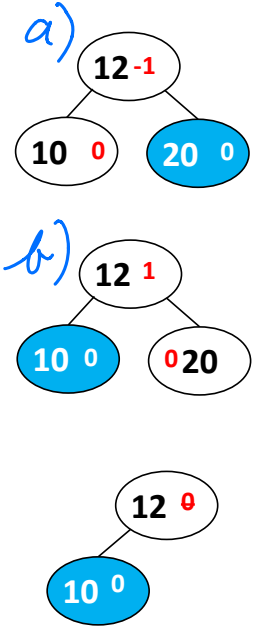
a) If b(p) was -1, then b(p) = 0. Done!

b) If b(p) was +1, then b(p) = 0. Done!

c) If b(p) was 0, then update b(p) and call insert-fix(p, n)

if added new left child b(p) = -1

if add right child b(p) = +1



AVL Insert-fix(p, n)

Precondition: p and n are balanced: {-1,0,1}
Postcondition: g, p, and n are balanced: {-1,0,1}

General Idea: Work up ancestor chain updating balances of the ancestor chain or fix a node that is out of balance.

If p is null or `parent(p)` is null, return
Let `g = parent(p)`
Assume p is left child of g [For right child swap left/right, +/-]

```
b(g) += -1 // Update g's balance for taller left subtree
if b(g) == 0, return  // b(g) was 1; right subtree was taller
if b(g) == -1, insertFix(g, p) // recurse
```

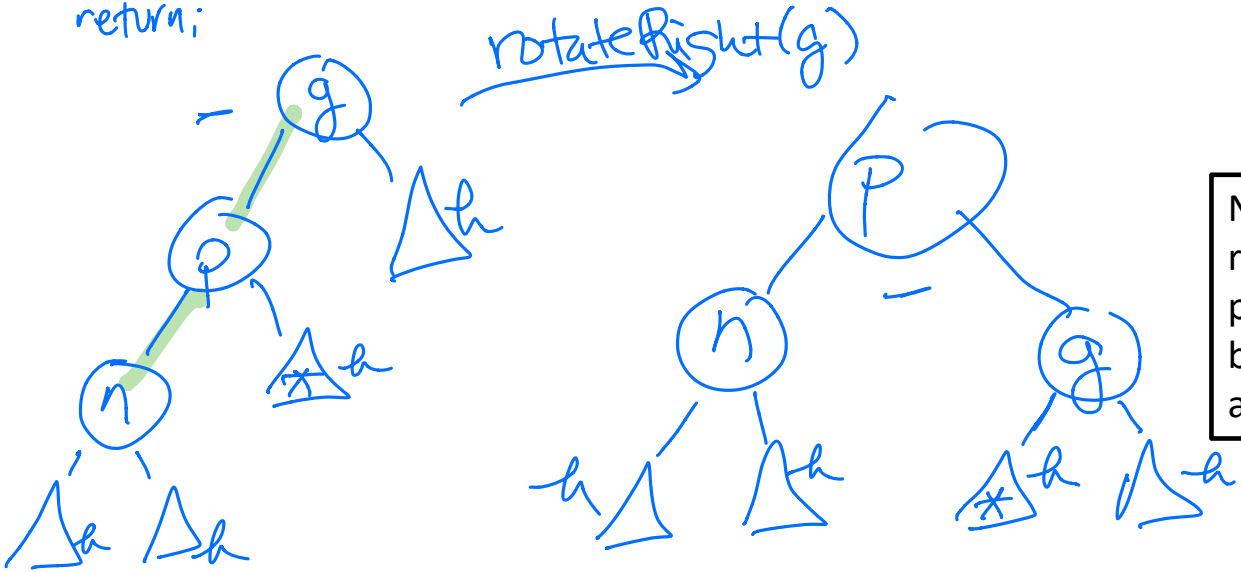
// b(g) was 0; left & right had same height and now left subtree has greater height

USC Viterbi School of Engineering

Insert-fix(p, n)

b(g) was -1 ⇒ g had taller left subtree
g is no longer balanced

```
If b(g) == -2
  i) If zig-zig then rotateRight(g); b(p) = b(g) = 0; return
     If zig-zag then rotateLeft(p); rotateRight(g);
          Case 1: b(n) == -1 then b(p) = 0; b(g) = +1; b(n) = 0;
          Case 2: b(n) == 0  then b(p) = 0; b(g)= 0;   b(n) = 0;
          Case 3: b(n) == +1 then b(p)= -1; b(g) = 0;  b(n) = 0;
     return;
```



General Idea: Work up ancestor chain updating balances of the ancestor chain or fix a node that is out of balance.

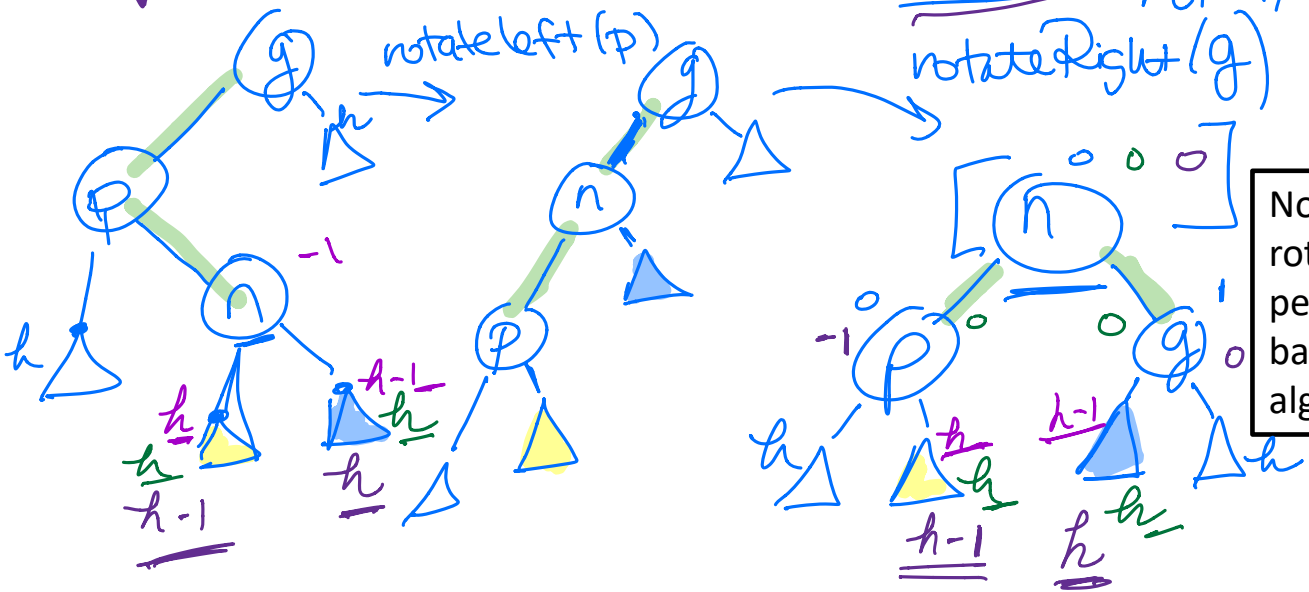
Note: Once a rotation is performed to balance a node, algorithm stops

USC Viterbi School of Engineering

Insert-fix(p, n)

```
If b(g) == -2
    If zig-zig then rotateRight(g); b(p) = b(g) = 0;
    If zig-zag then rotateLeft(p); rotateRight(g);
        Case 1: b(n) == -1 then b(p) = 0; b(g) = +1; b(n) = 0;
        Case 2: b(n) == 0  then b(p) = 0; b(g)= 0;   b(n) = 0;
        Case 3: b(n) == +1 then b(p)= -1; b(g) = 0;  b(n) = 0;
```

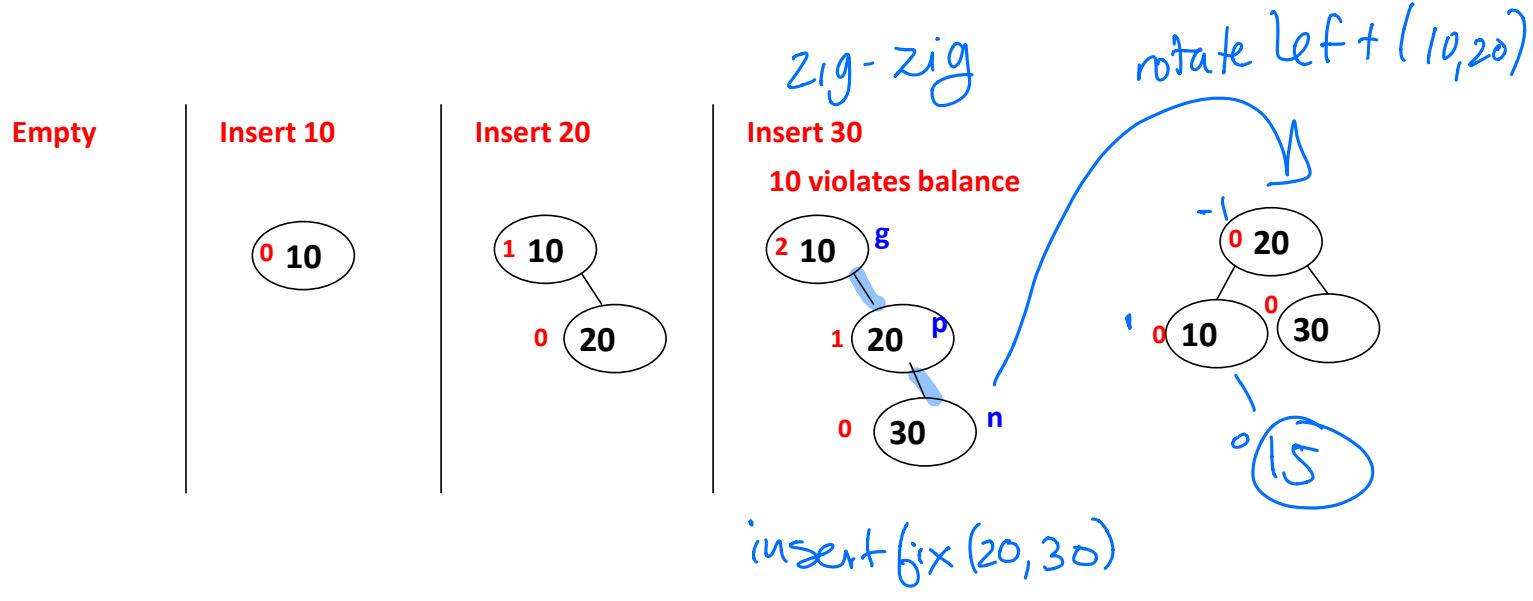
General Idea: Work up ancestor chain updating balances of the ancestor chain or fix a node that is out of balance.



Note: Once a rotation is performed to balance a node, algorithm stops

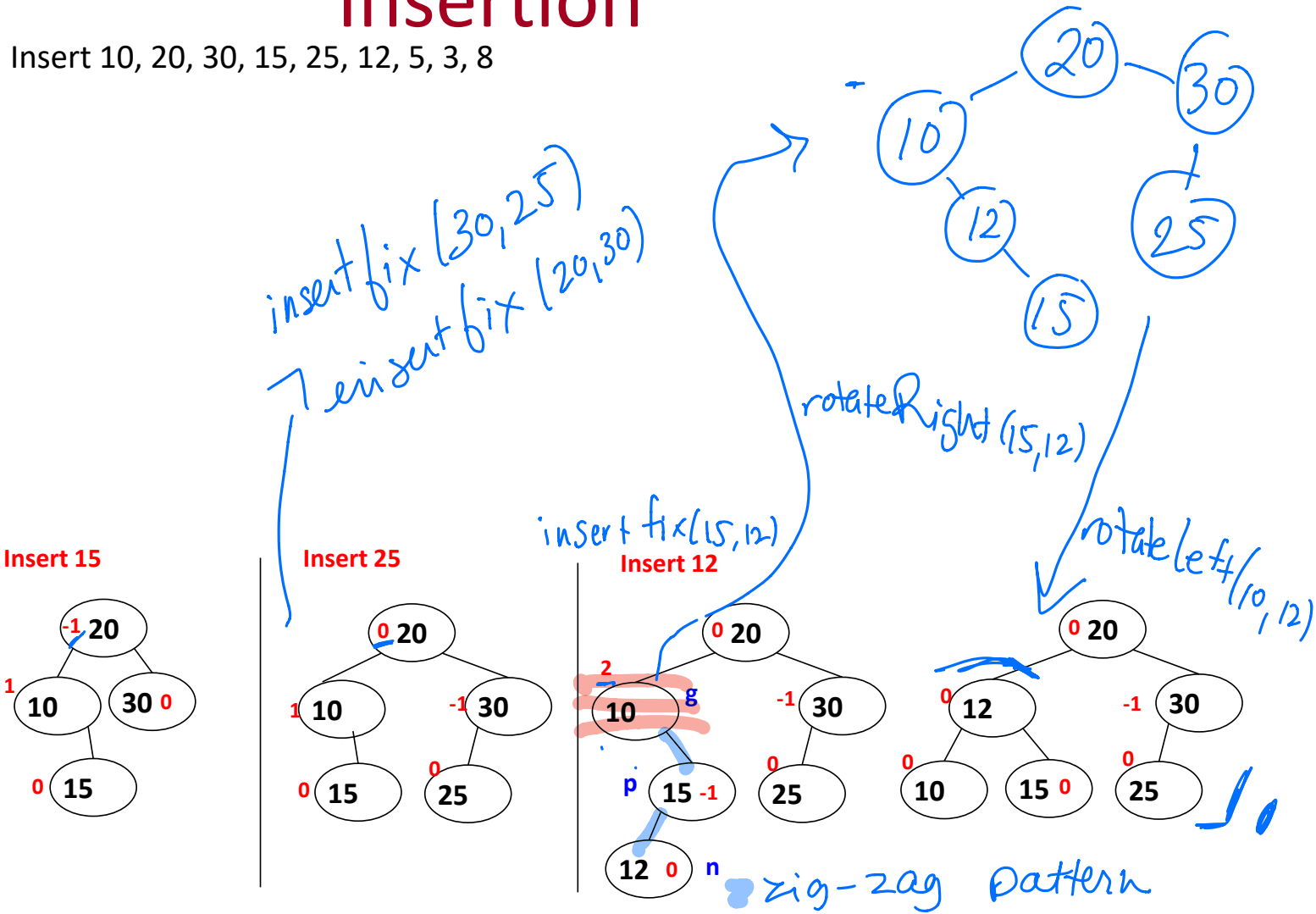
Insertion

Insert 10, 20, 30, 15, 25, 12, 5, 3, 8



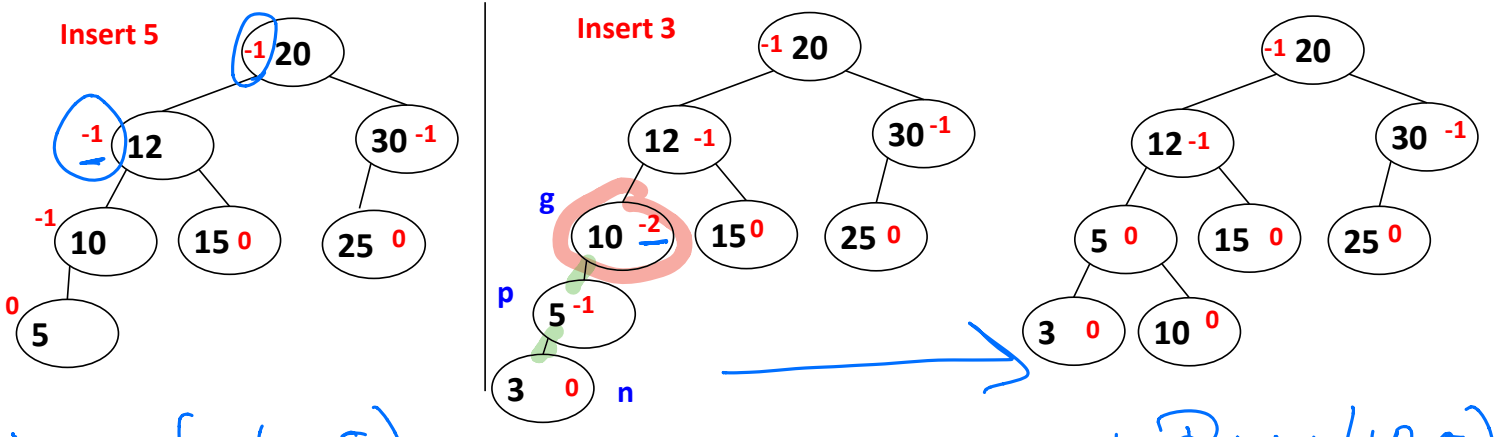
Insertion

Insert 10, 20, 30, 15, 25, 12, 5, 3, 8



Insertion

Insert 10, 20, 30, 15, 25, 12, 5, 3, 8



insertfix(10,5)

insertfix(12,10)

insertfix(20,12)

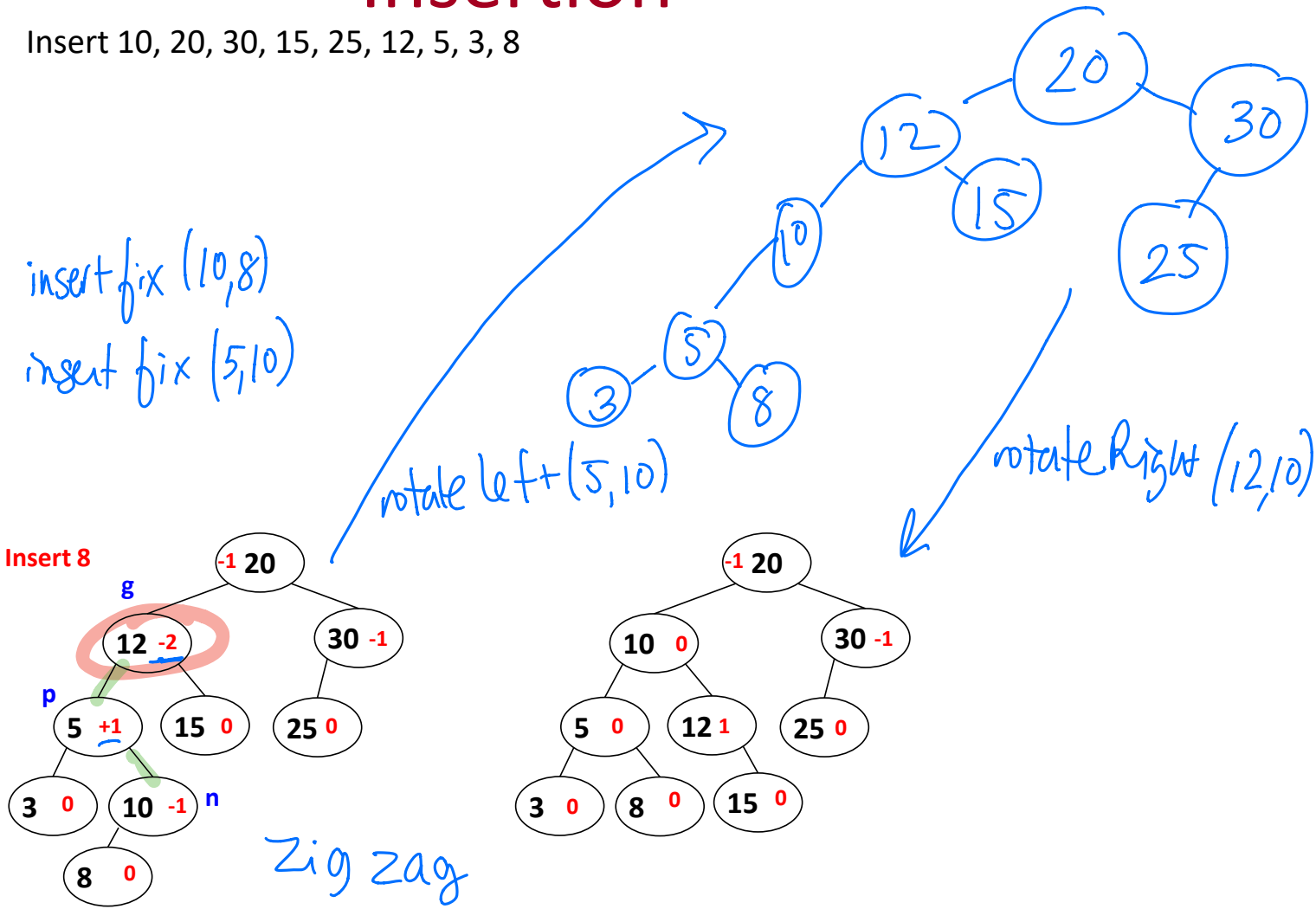
insertfix(5,3)

zig zig

rotateRight(10,5)

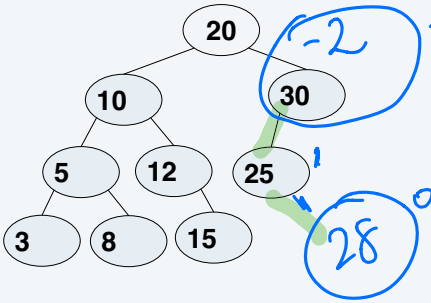
Insertion

Insert 10, 20, 30, 15, 25, 12, 5, 3, 8



1. AVL Insertion Practice

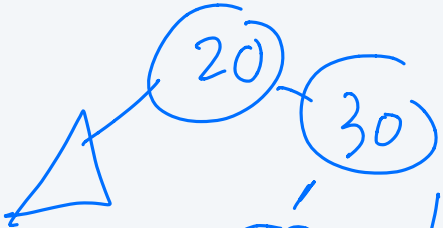
a) Insert key=28



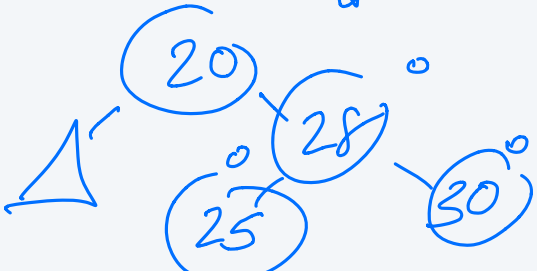
insert fix (25, 28)

Zig Zag

rotate left (25, 28)

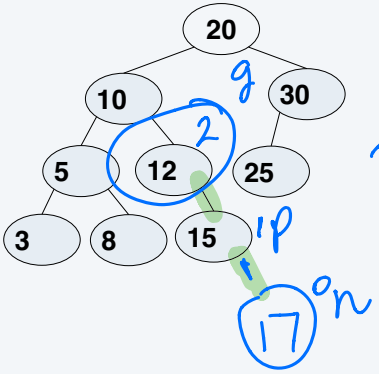


rotate Right (30, 28)

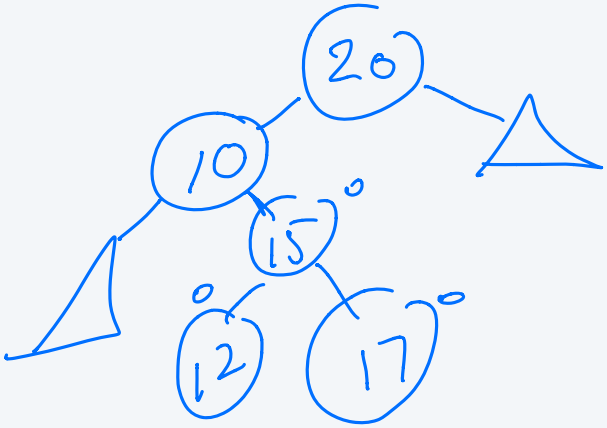


1. AVL Insertion Practice

b) Insert key=17



rotate left (12, 15)

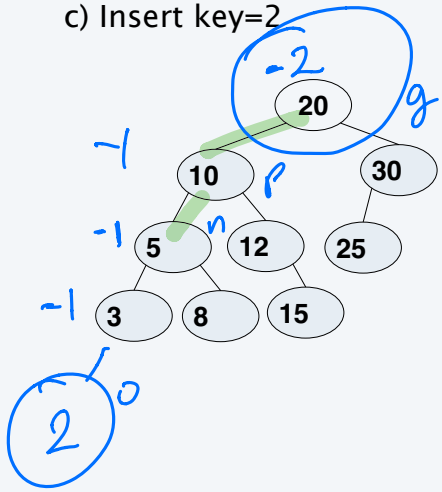


insert fix (15, 17)

zig zig

1. AVL Insertion Practice

c) Insert key=2



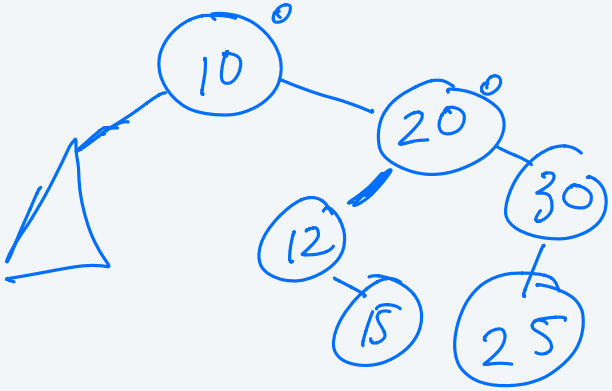
1) insertfix(3,2)

2) insertfix(5,3)

3) insertfix(10,5)

zig zig

rotateRight(20,10)



Remove Operation

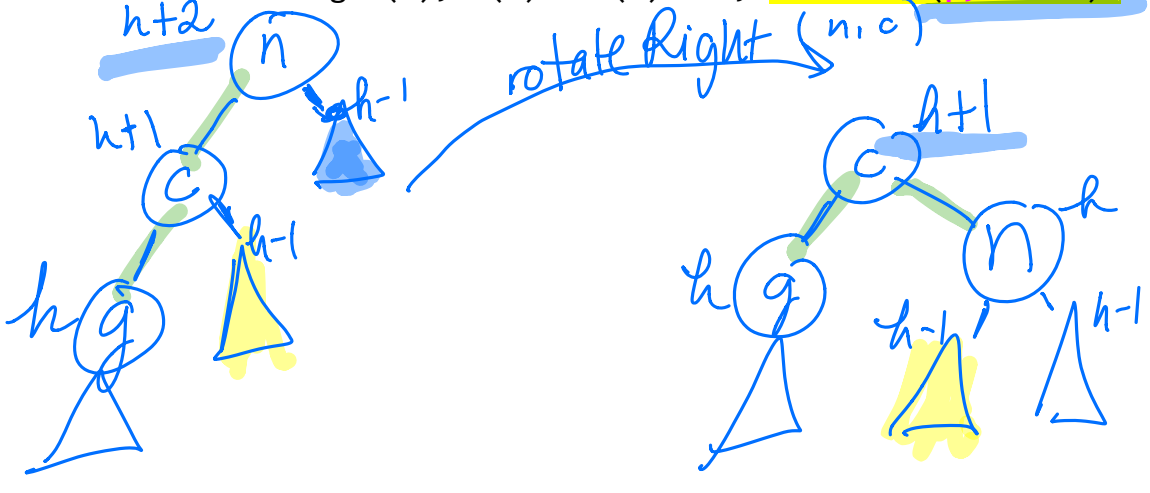
- Remove operations may also require rebalancing via rotations
- The key idea is to update the balance of the nodes on the ancestor pathway
- If an ancestor gets out of balance then perform rotations to rebalance
  - Unlike insert, performing rotations during removal does not necessarily complete algorithm.
  - There is need to continue recursing.

Remove

- BST Remove n
- Let `p = parent(n) – Parent of the node removed`
- If `p` is not NULL,
  - `diff` *will be the amount added to updated the balance of p*
  - If left child removed, the right subtree greater height:
  - If `n` is a left child, let `diff = +1`
  - If right child removed, the left subtree greater height:
  - if `n` is a right child, let `diff = -1`
  - `removeFix(p, diff);`

RemoveFix(n, diff)

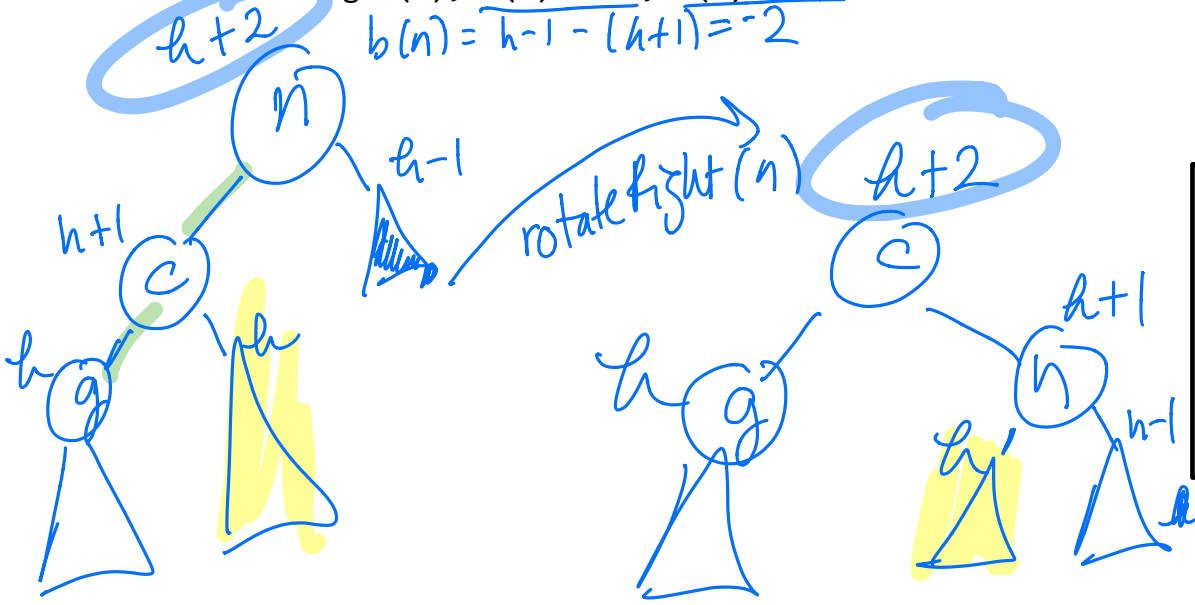
- If n is null, return
- Compute next recursive call's arguments now before altering the tree
  - Let p = parent(n) and if p is not NULL let ndiff (nextdiff) = +1 if n is a left child and -1 otherwise
- Assume diff = -1 i.e. right child is removed for these slides (Mirror if diff = +1)
- `Case 1: b(n) + diff == -2`
  - [Perform the check for the mirror case where b(n) + diff == +2, flipping left/right and -1/+1]
  - Let c = left(n), the taller of the children
  - `Case 1a: b(c) == -1   // zig-zig case`
    - `rotateRight(n), b(n) = b(c) = 0, removeFix(p, ndiff)`



Note:
p = parent of n
n = current node
c = taller child of n
g = grandchild of n

RemoveFix(n, diff)

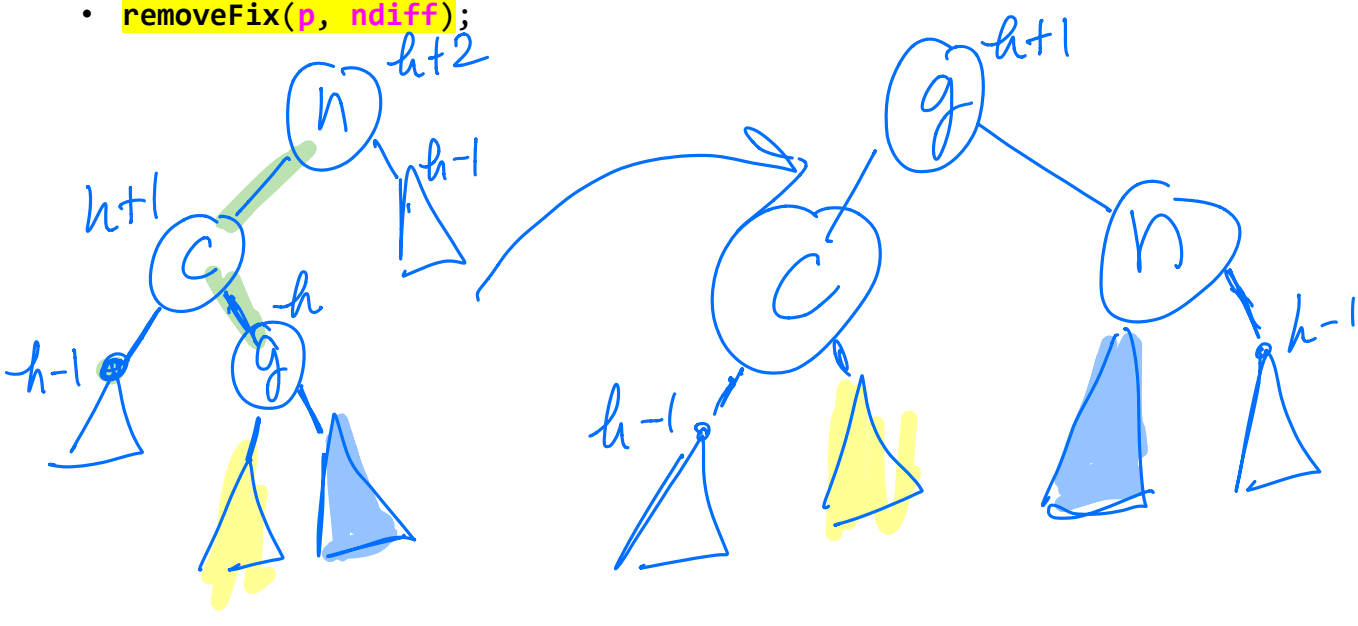
- If n is null, return
- Compute next recursive call's arguments now before altering the tree
  - Let p = parent(n) and if p is not NULL let ndiff (nextdiff) = +1 if n is a left child and -1 otherwise
- Assume diff = -1 and follow the remainder of this approach, mirroring if diff = +1
- `Case 1: b(n) + diff == -2`
  - `Case 1b: b(c) == 0 // zig-zig case`
    - `rotateRight(n), b(n) = -1, b(c) = +1 // Done!`



Note:
p = parent of n
n = current node
c = taller child of n
g = grandchild of n

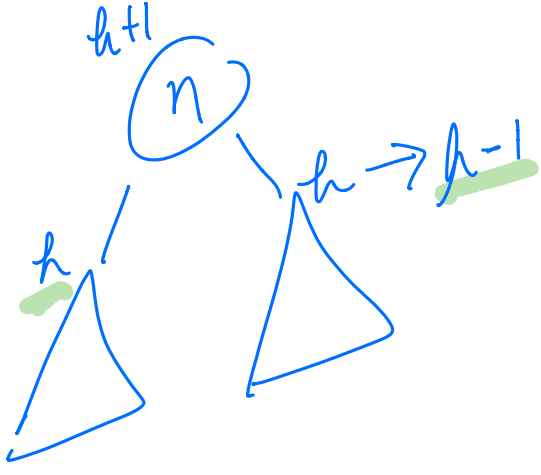
RemoveFix(n, diff)

- Case 1: b(n) + diff == -2
  - Case 1c: b(c) == +1 // zig-zag case
    - Let g = right(c)
    - rotateLeft(c) then rotateRight(n)
    - If b(g) was +1 then b(n) = 0, b(c) = -1, b(g) = 0
    - If b(g) was 0 then b(n) = 0, b(c) = 0, b(g) = 0
    - If b(g) was -1 then b(n) = +1, b(c) = 0, b(g) = 0
    - `removeFix(p, ndiff);`



RemoveFix(n, diff)

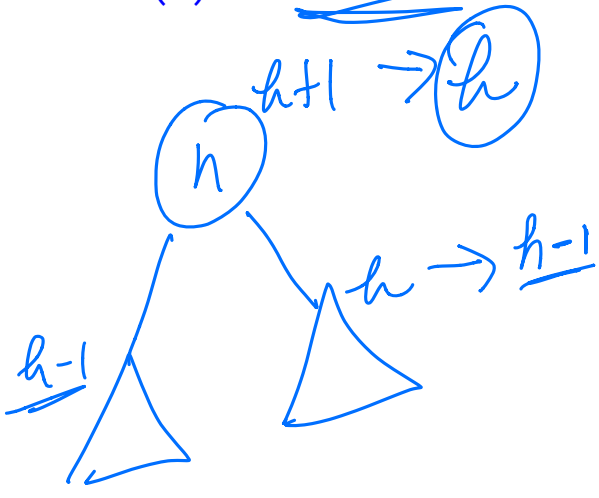
- `Case 2: b(n) + diff == -1:` then `b(n) = -1; // Done!`



Note:
p = parent of n
n = current node
c = taller child of n
g = grandchild of n

RemoveFix(n, diff)

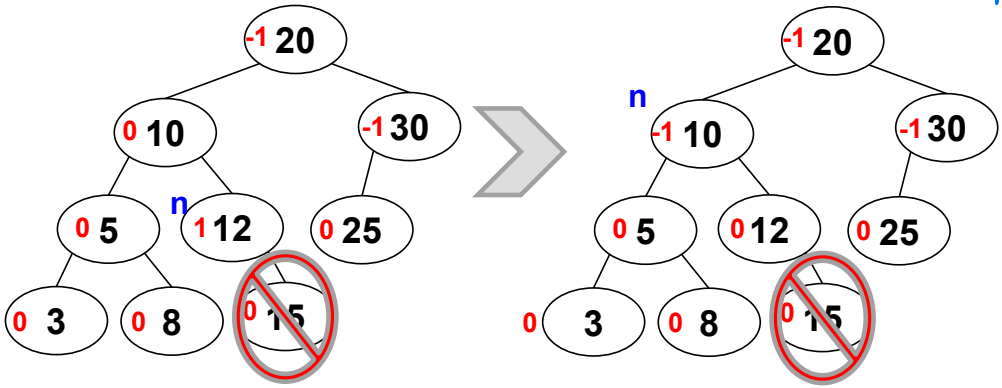
- Case 3: b(n) + diff == ~~0:~~ then b(n) = 0, **removeFix(p, ndiff)**



Note:
p = parent of n
n = current node
c = taller child of n
g = grandchild of n

Remove Examples

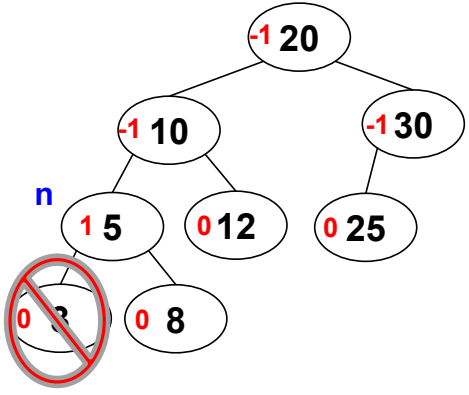
Remove 15



removefix(12, -1)

removefix(10, -1)

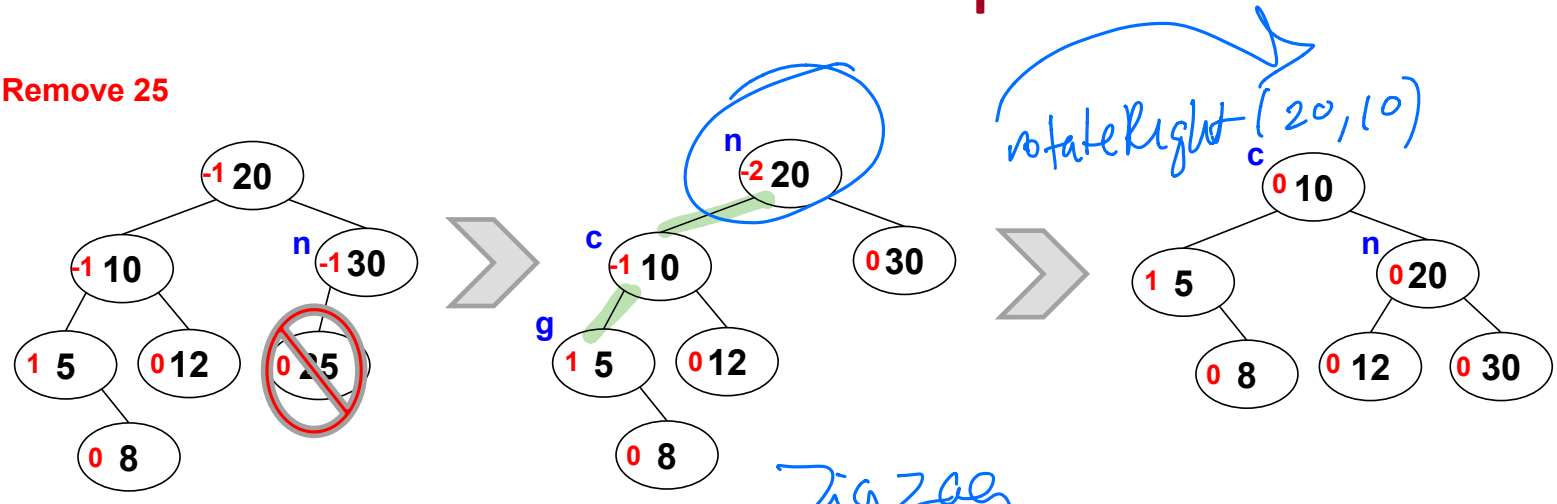
Remove 3



removefix(5, 1)

Remove Examples

Remove 25



zigzag

removeFix(30,1)

removeFix(20,-1)

Remove Examples

Remove 20