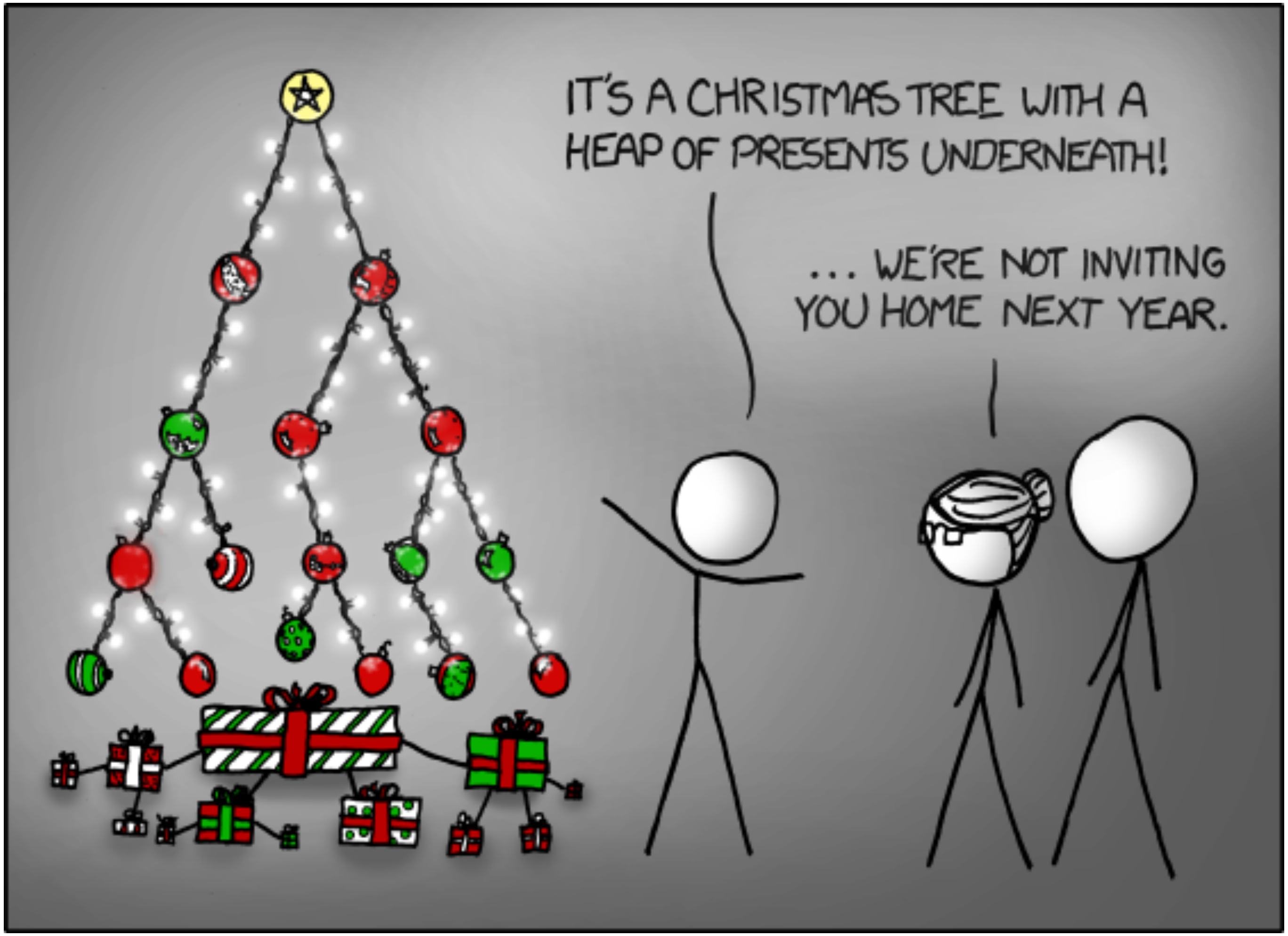
CSCI 104

Rafael Ferreira da Silva
rafsilva@isi.edu

Slides adapted from: Mark Redekopp and David Kempe



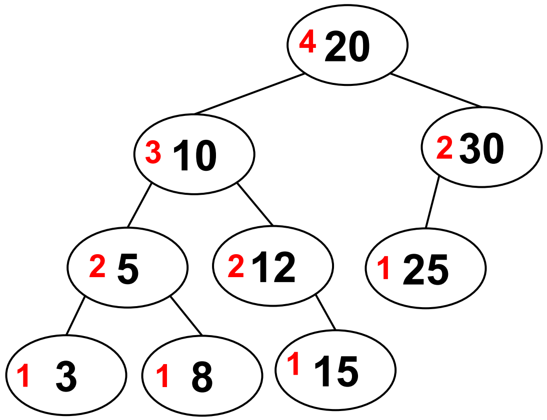
https://xkcd.com/835/

Self-balancing tree proposed by Adelson-Velsky and Landis

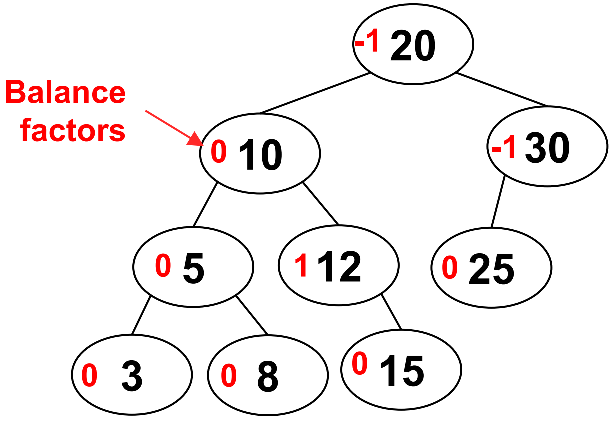
AVL TREES

AVL Trees

- A binary search tree where the **height difference** between left and right subtrees of a node is **at most 1**
  - Binary Search Tree (BST): Left subtree keys are less than the root and right subtree keys are greater
- Two implementations:
  - Height: Just store the height of the tree rooted at that node
  - Balance: Define b(n) as the balance of a node = (Right – Left) Subtree Height
    - Legal values are -1, 0, 1
    - Balances require at most 2-bits if we are trying to save memory.
    - **Let's use balance for this lecture.**



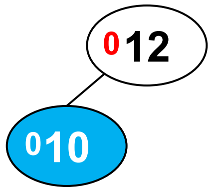
AVL Tree storing Heights

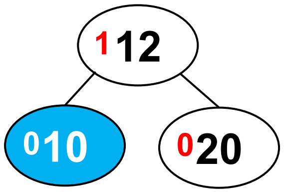


AVL Tree storing balances

Adding a New Node

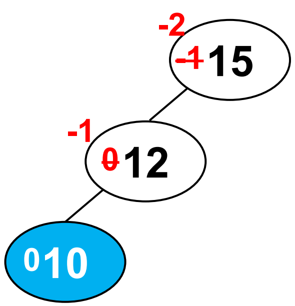
- Once a new node is added, can its parent be out of balance?
  - No, assuming the tree is "in-balance" when we start.
  - Thus, our parent has to have
    - A balance of 0
    - A balance of 1 if we are a new left child or -1 if a new right child
  - Otherwise it would not be our parent or the parent would have been out of balance already

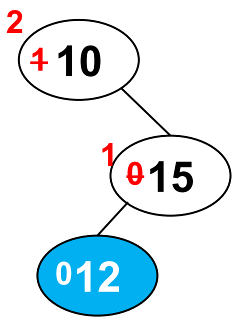




Losing Balance

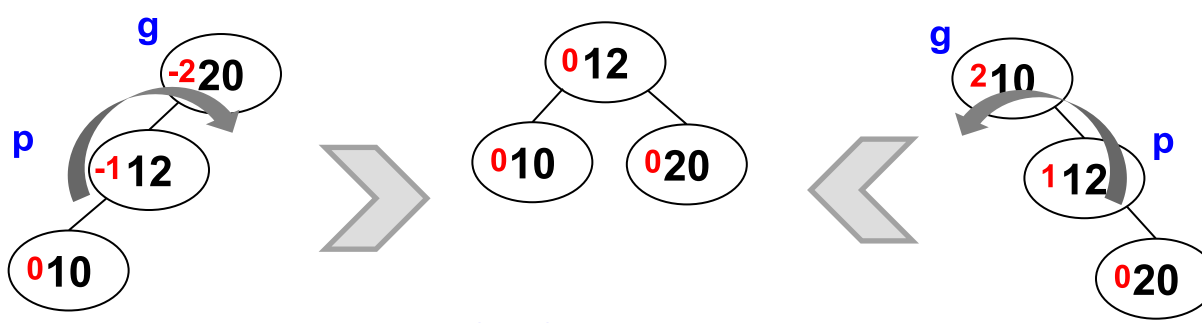
- If our parent is not out of balance, is it possible our grandparent is out of balance?
- Sure, so we need a way to re-balance it



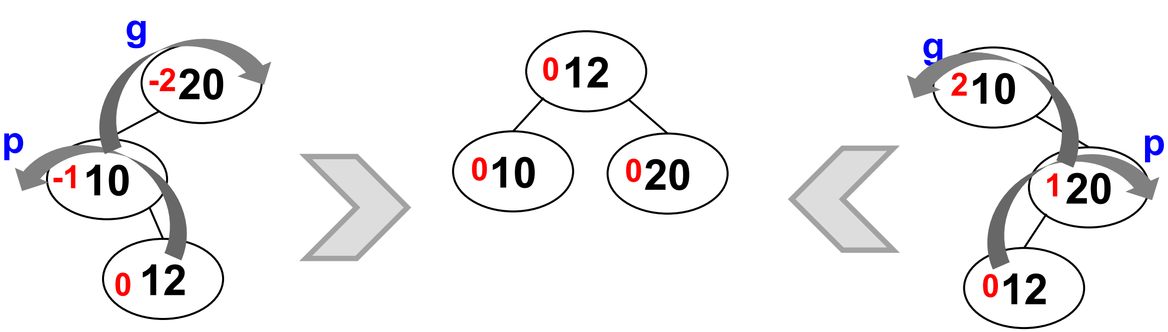


To Zig or Zag

- The rotation(s) required to balance a tree is/are dependent on the grandparent, parent, child relationships
- We can refer to these as the zig-zig case and zig-zag case
- Zig-zig requires 1 rotation
- Zig-zag requires 2 rotations (first converts to zig-zig)



Left-left or Right-right
(a.k.a. Zig-zig)
[One left/right rotation of g]



Left-right or Right-left
(a.k.a. Zig-zag)
[Rotate p then g]

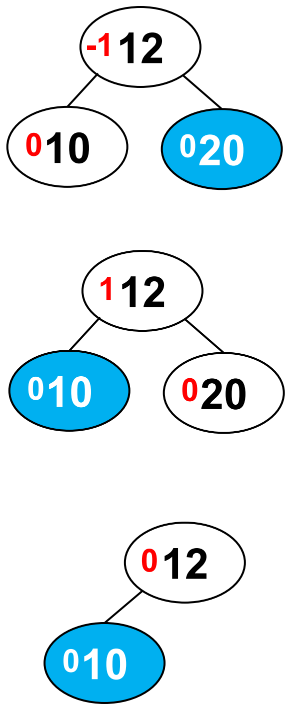
Disclaimer

- There are many ways to structure an implementation of an AVL tree…the following slides represent just 1
  - Focus on the bigger picture ideas as that will allow you to more easily understand other implementations

USC Viterbi School of Engineering

Insert(n)

- If empty tree => set as root, b(n) = 0, done!
- Insert n (by walking the tree to a leaf, p, and inserting the new node as its child), set balance to 0, and look at its parent, p
  - If b(p) = -1, then b(p) = 0. Done!
  - If b(p) = +1, then b(p) = 0. Done!
  - If b(p) = 0, then update b(p) and call insert-fix(p, n)



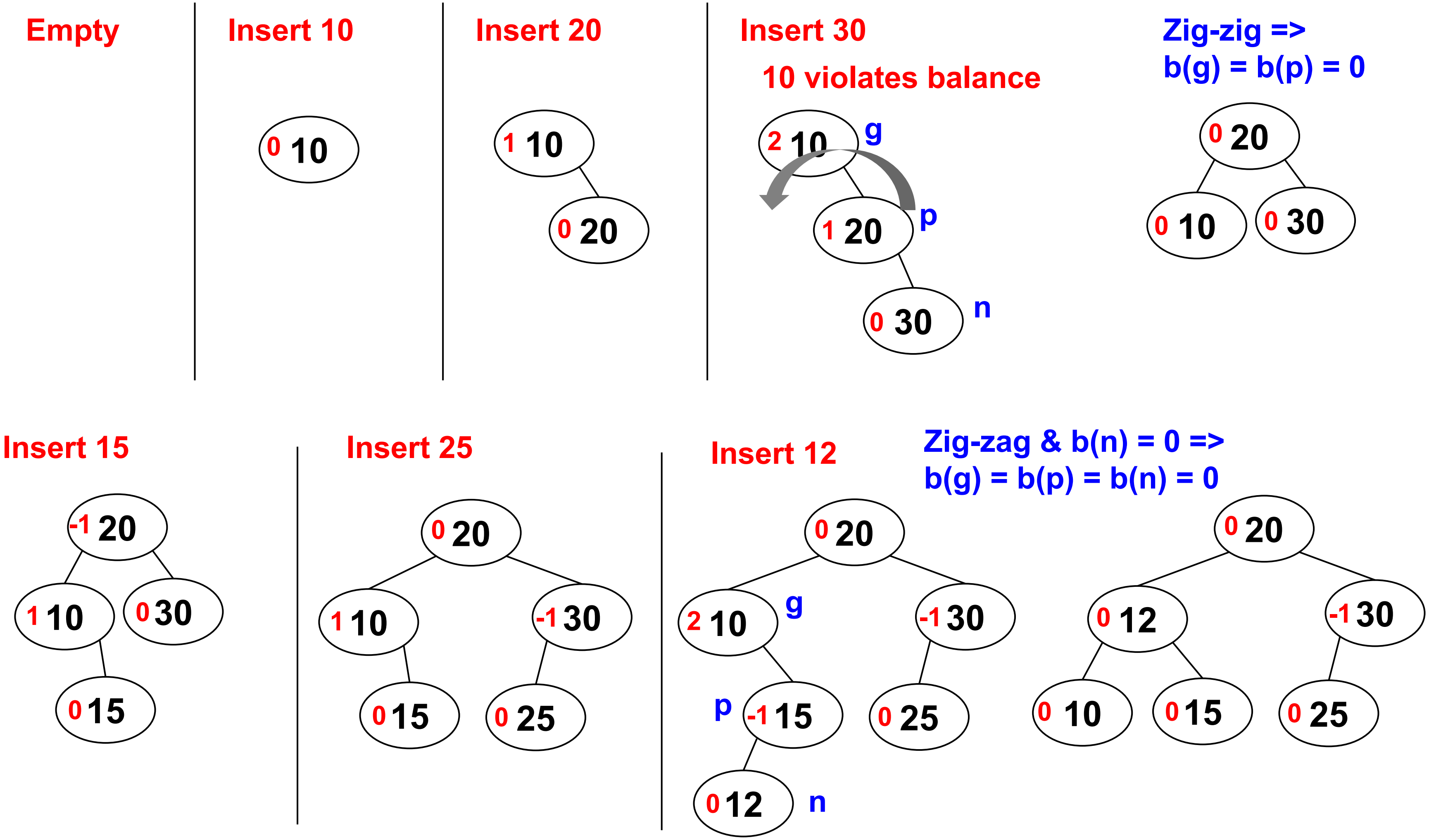
Insert-fix(p, n)

- Precondition: p and n are balanced: {+1,0,-1}
- Postcondition: g, p, and n are balanced: {+1,0,-1}
- 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, +/-]
  - g.balance += -1
  - if g.balance == 0, return
  - if g.balance == -1, insertFix(g, p)
  - If g.balance == -2
    - If zig-zig then rotateRight(g); p.balance = g.balance = 0
    - If zig-zag then rotateLeft(p); rotateRight(g);
      - if n.balance == -1 then p.balance = 0; g.balance(+1); n.balance = 0;
      - if n.balance == 0 then p.balance = 0; g.balance(0); n.balance = 0;
      - if n.balance == +1 then p.balance = -1; g.balance(0); n.balance = 0;

Note: If you perform a rotation, you will NOT need to recurse. You are done!

Insertion

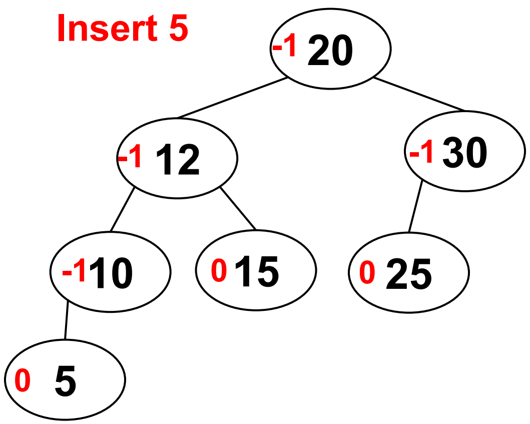
- Insert 10, 20, 30, 15, 25, 12, 5, 3, 8



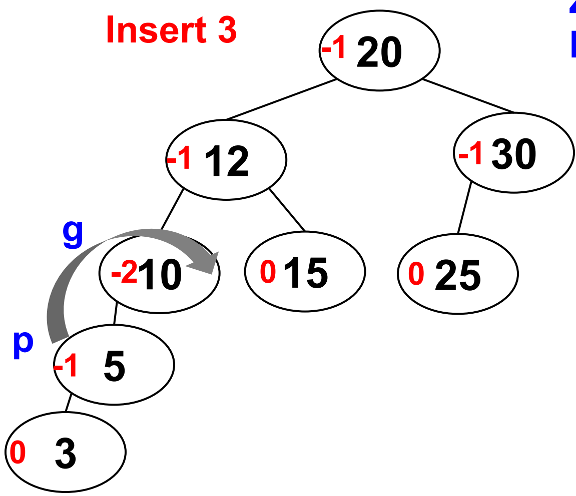
Insertion

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

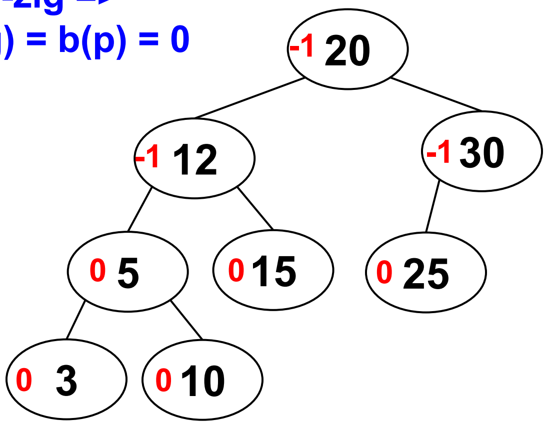
Insert 5



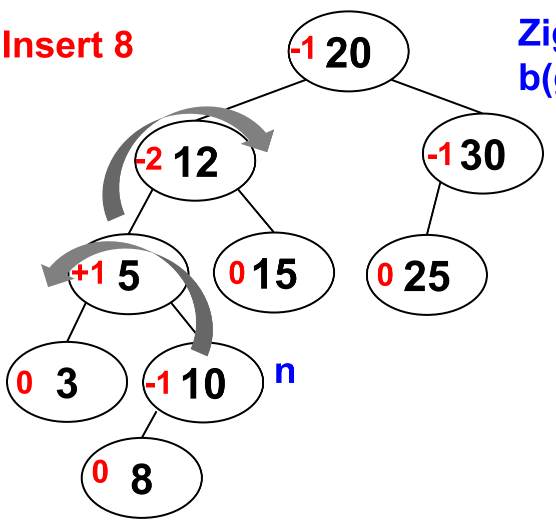
Insert 3



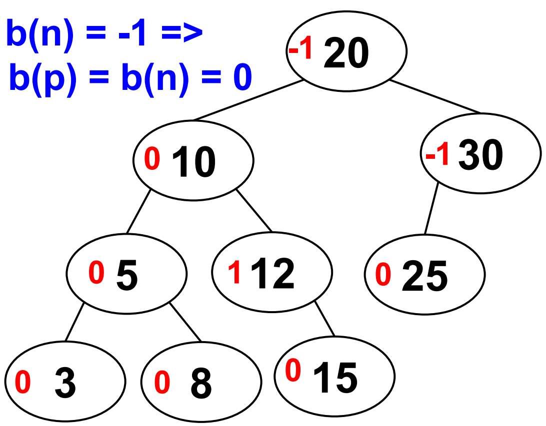
Zig-zig =>
b(g) = b(p) = 0



Insert 8

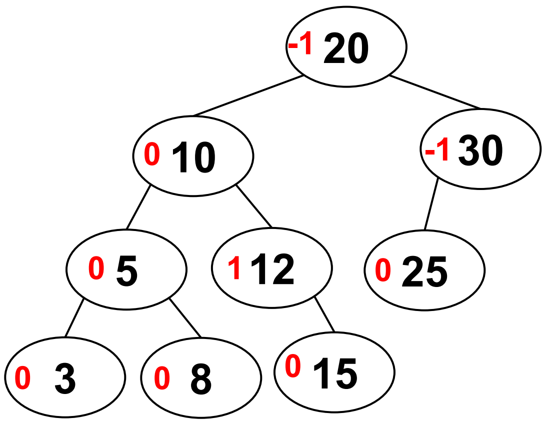


Zig-zag & b(n) = -1 =>
b(g) = 1 & b(p) = b(n) = 0



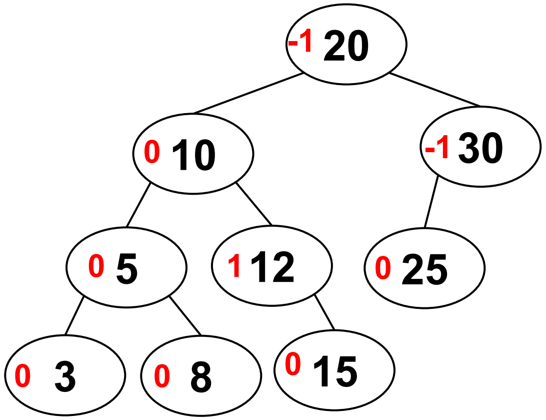
Insertion Exercise 1

- Insert key=28



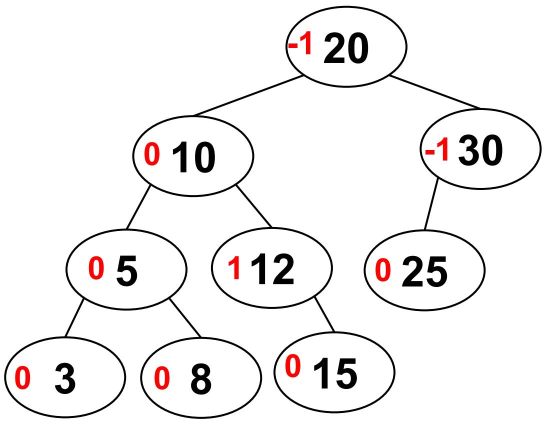
Insertion Exercise 2

- Insert key=17



Insertion Exercise 3

- Insert key=2



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 does not mean you are done, but need to continue
- There are slightly more cases to worry about but not too many more

Remove

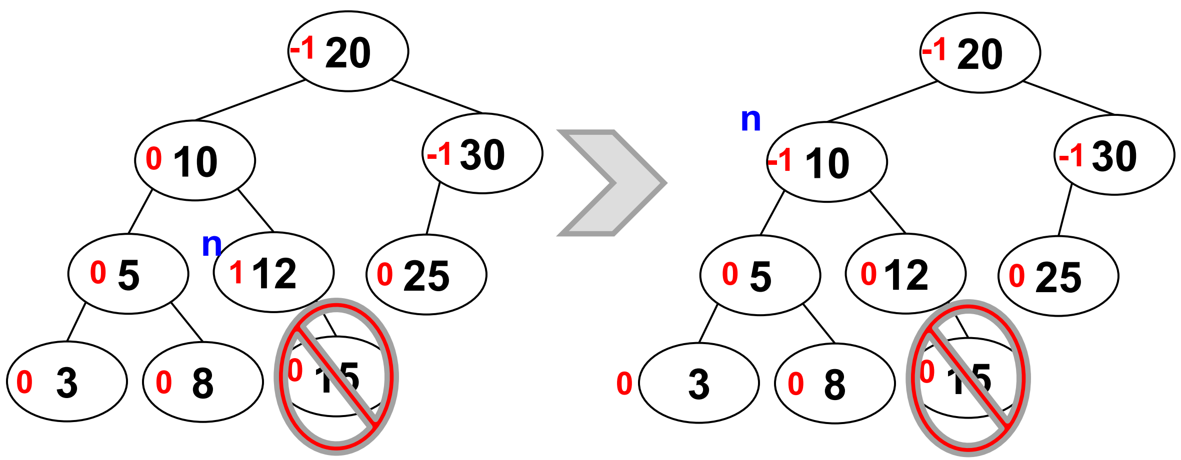
- Let n = node to remove (perform BST find) and p = parent(n)
- If n has 2 children, swap positions with in-order successor and perform the next step
  - Now n has 0 or 1 child guaranteed
- If n is not in the root position determine its relationship with its parent
  - If n is a left child, let diff = +1
  - if n is a right child, let diff = -1
- Delete n and update tree, including the root if necessary
- removeFix(p, diff);

RemoveFix(n, diff)

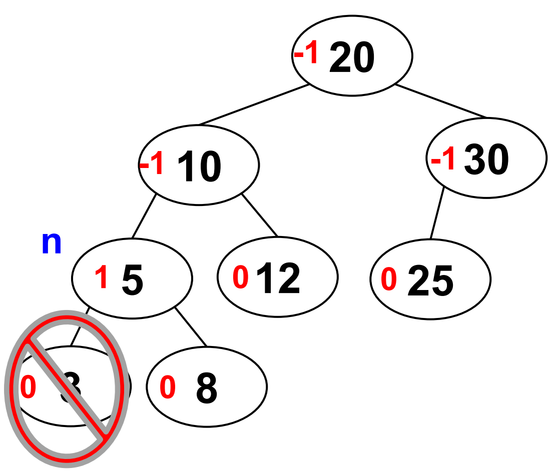
- If n is null, return
- Let ndiff = +1 if n is a left child and -1 otherwise
- Let p = parent(n). Use this value of p when you recurse.
- If balance of n would be -2 (i.e. balance(n) + diff == -2)
  - [Perform the check for the mirror case where balance(n) + diff == +2, flipping left/right and -1/+1]
  - Let c = left(n), the taller of the children
  - If balance(c) == -1 or 0 (zig-zig case)
    - rotateRight(n)
    - if balance(c) == -1 then balance(n) = balance(c) = 0, removeFix(p, ndiff)
    - if balance(c) == 0 then balance(n) = -1, balance(c) = +1, done!
  - else if balance(c) == 1 (zig-zag case)
    - rotateLeft(c) then rotateRight(n)
    - Let g = right(c)
    - If balance(g) == +1 then balance(n) = 0, balance(c) = -1, balance(g) = 0
    - If balance(g) == 0 then balance(n) = balance(c) = 0, balance(g) = 0
    - If balance(g) == -1 then balance(n) = +1, balance(c) = 0, balance(g) = 0
    - removeFix(parent(p), ndiff);
- else if balance(n) == 0 then balance(n) += diff, done!
- else balance(n) = 0, removeFix(p, ndiff)

Remove Examples

Remove 15



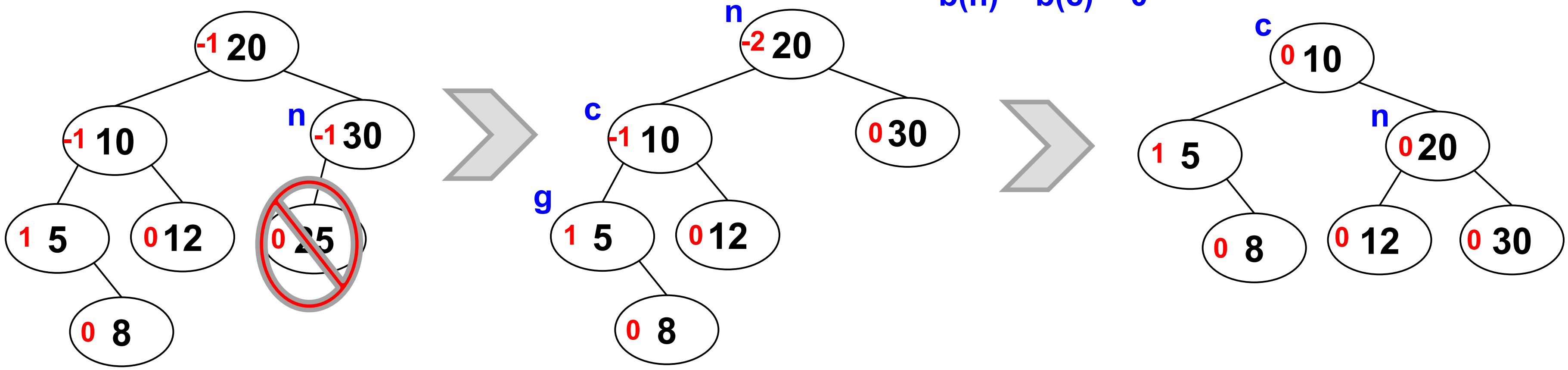
Remove 3



Remove Examples

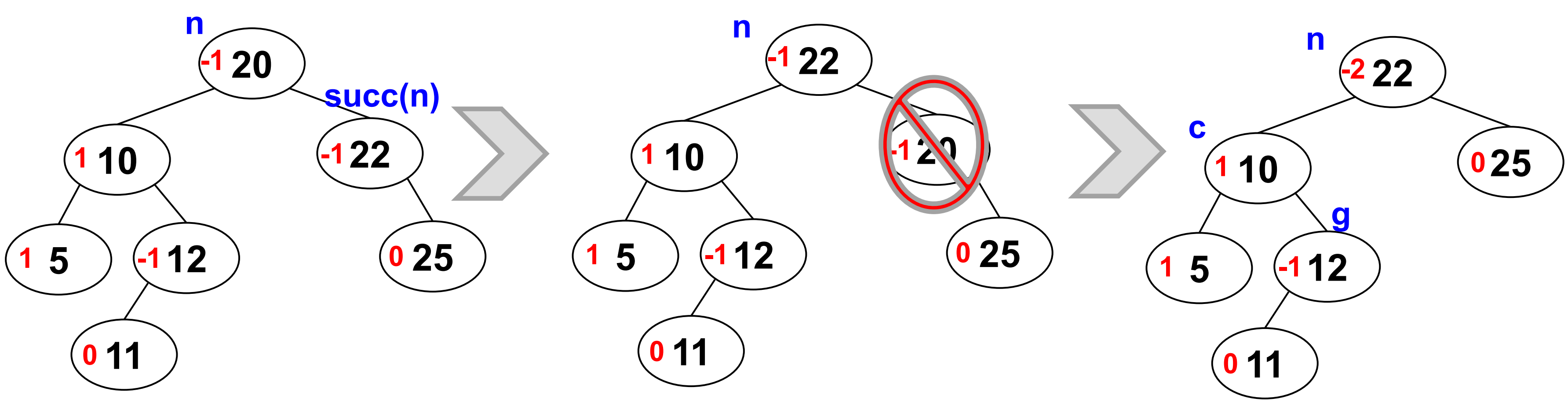
Remove 25

Zig-zig & b(c) = -1 =>
b(n) = b(c) = 0

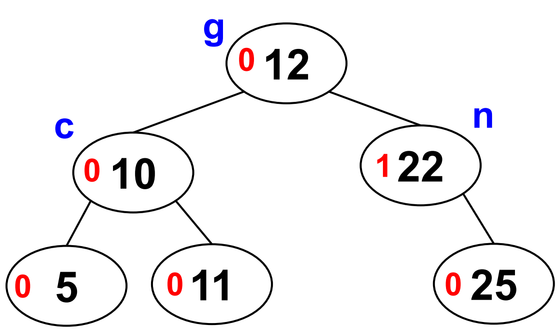


Remove Examples

Remove 20

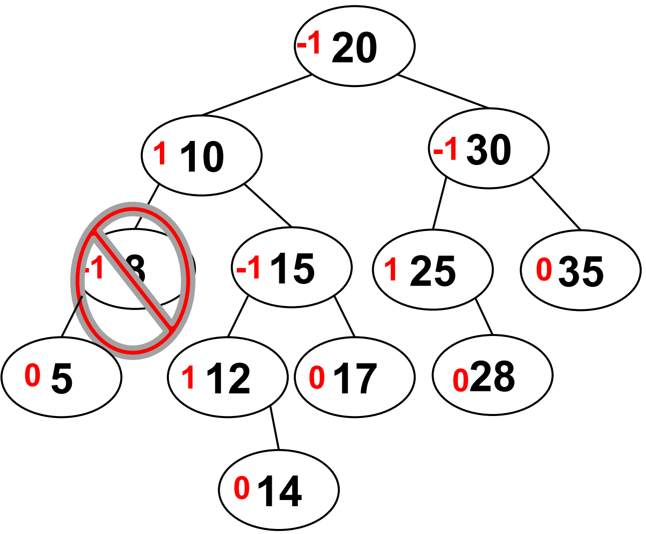


Zig-zag & b(g) = -1 =>
b(n) = +1, b(c) = 0, b(g) = 0



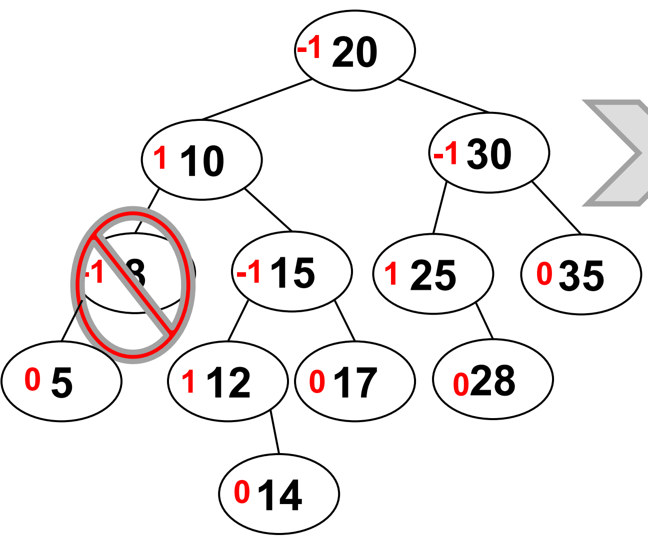
Remove Example 1

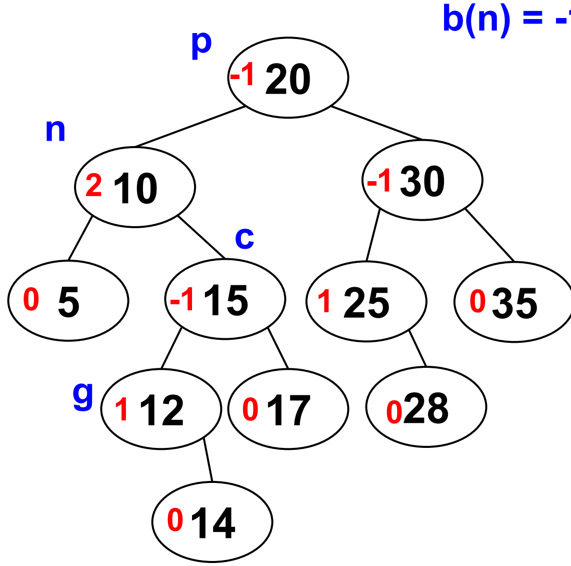
Remove 8



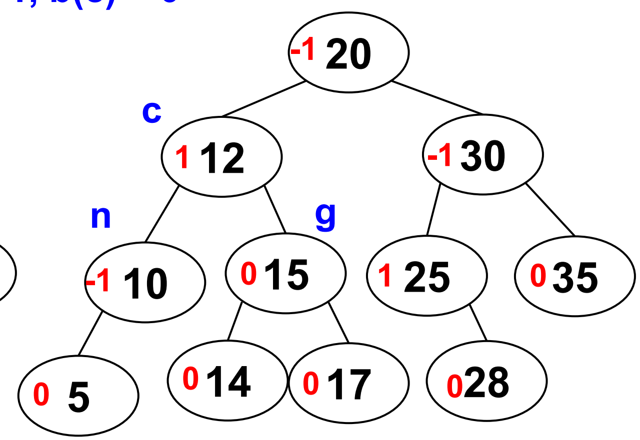
Remove Example 1

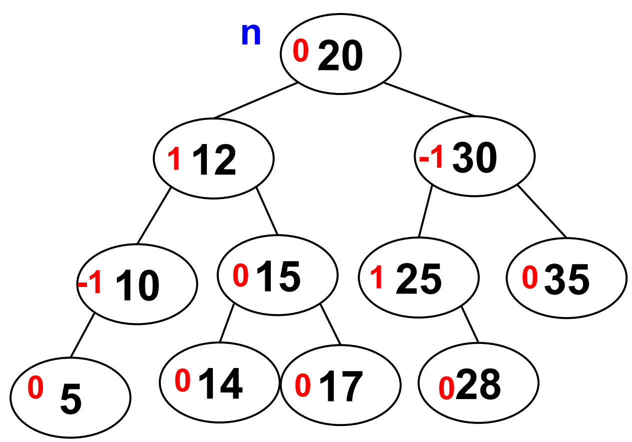
Remove 8





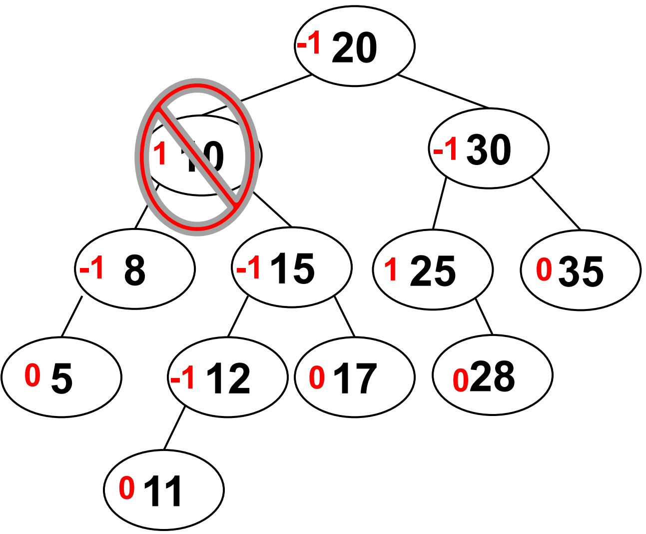
Zig-zag & b(1) = 0 => b(n) = -1, b(c) = 0





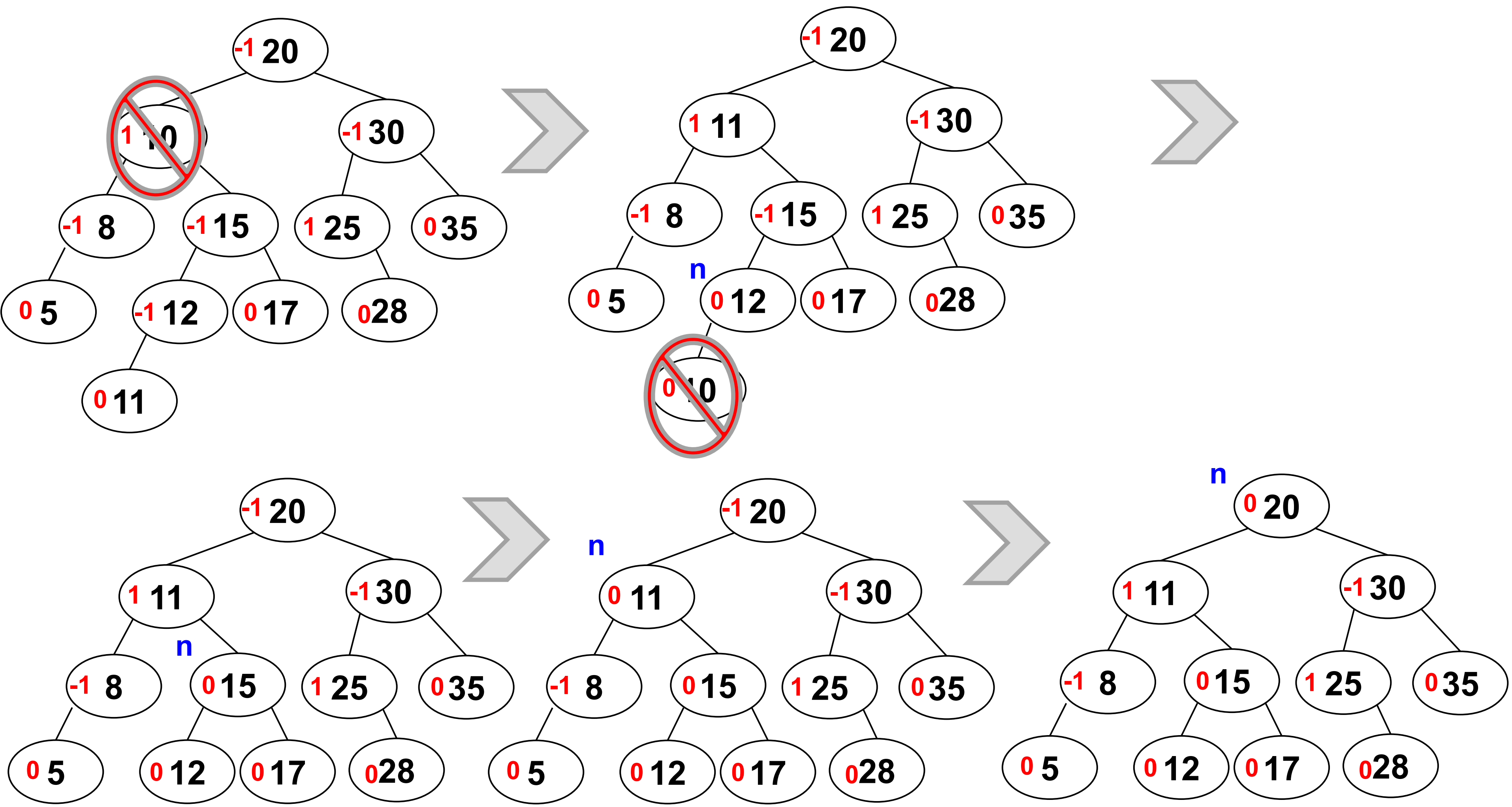
Remove Example 2

Remove 10



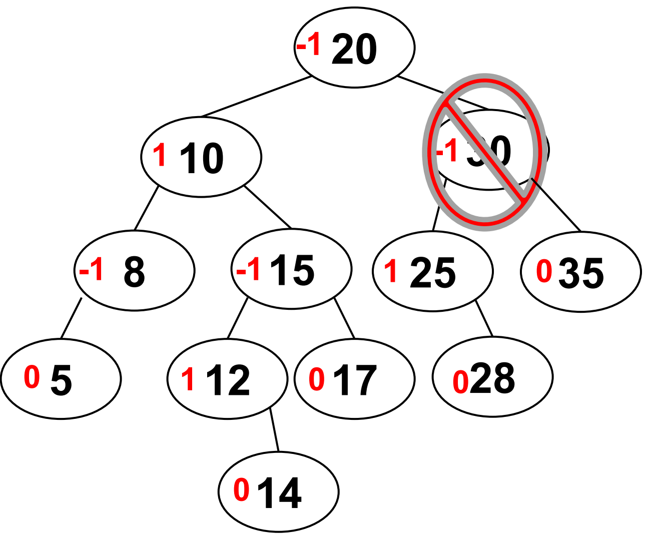
Remove Example 2

Remove 10



Remove Example 3

Remove 30

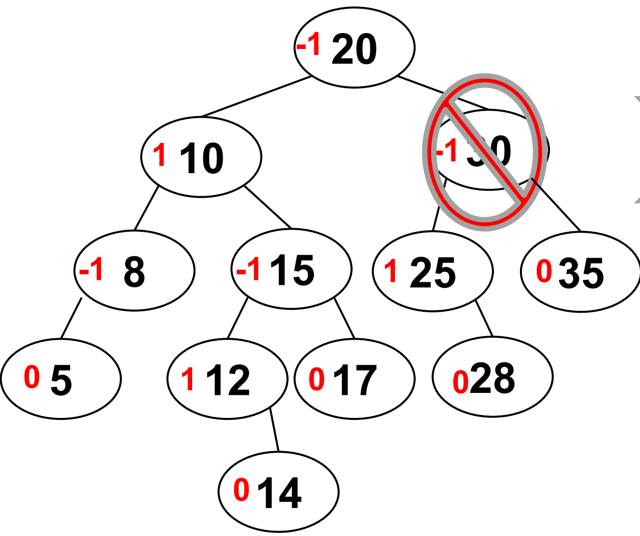


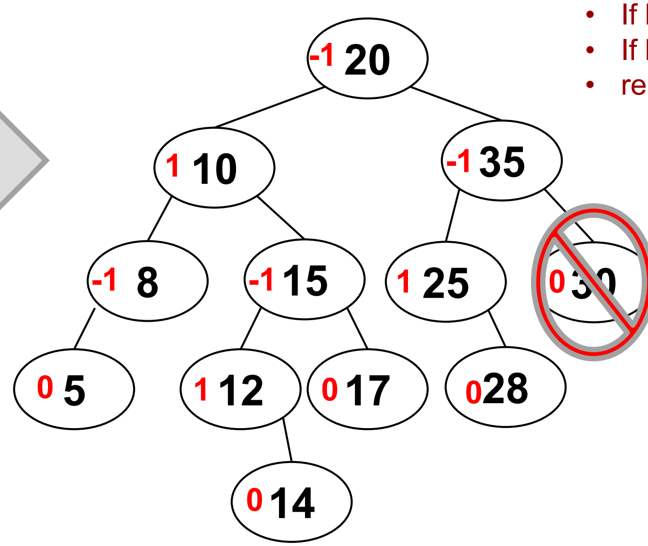
Remove Example 3

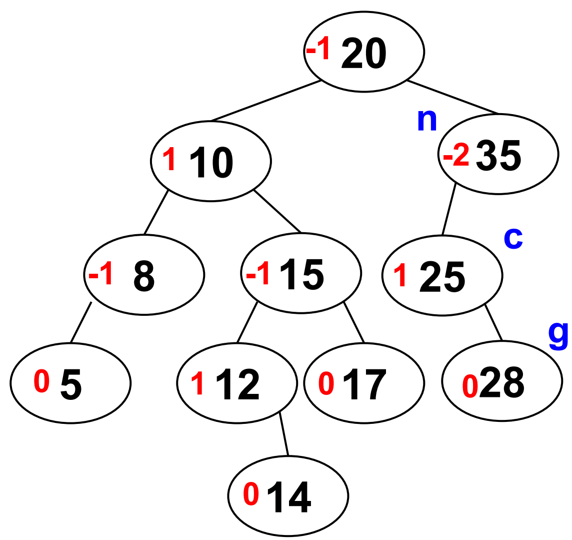
else if b(c) == 1 (zig-zag case)

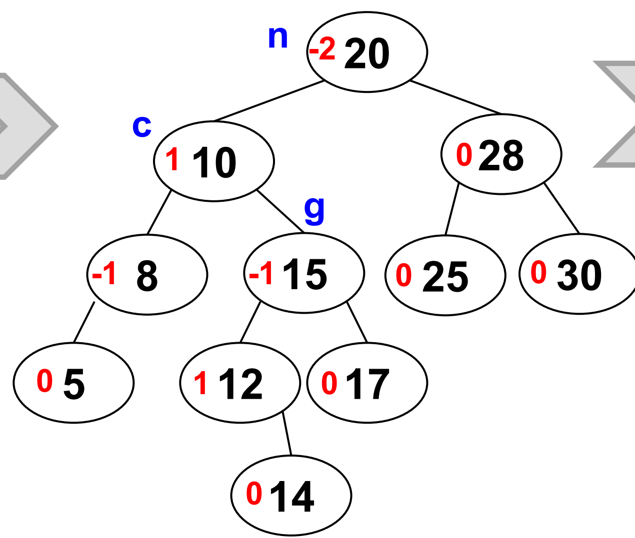
- rotateLeft(c) then rotateRight(n)
- Let g = right(c), b(g) = 0
- If b(g) == +1 then b(n) = 0, b(c) = -1, b(g) = 0
- If b(g) == 0 then b(n) = b(c) = 0, b(g) = 0
- If b(g) == -1 then b(n) = +1, b(c) = 0, b(g) = 0
- removeFix(parent(p), ndiff);

Remove 30





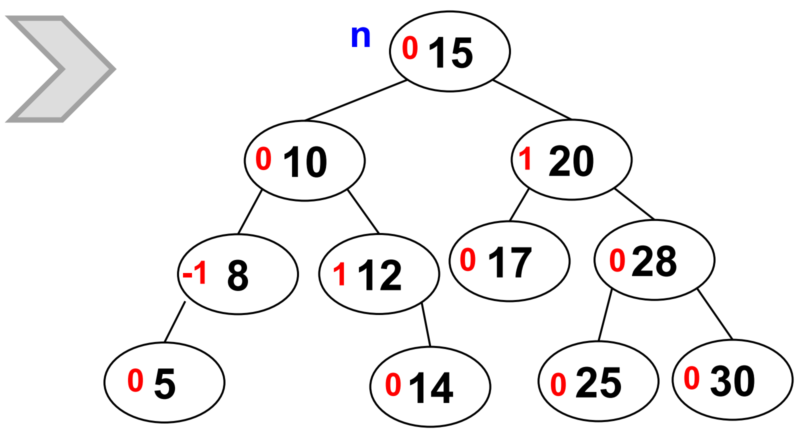




Remove Example 3 (cont)

else if b(c) == 1 (zig-zag case)
- rotateLeft(c) then rotateRight(n)
- Let g = right(c), b(g) = 0
- If b(g) == +1 then b(n) = 0, b(c) = -1, b(g) = 0
- If b(g) == 0 then b(n) = b(c) = 0, b(g) = 0
- If b(g) == -1 then b(n) = +1, b(c) = 0, b(g) = 0
- removeFix(parent(p), ndiff);

Remove 30 (cont.)



Online Tool

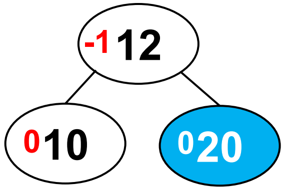
- https://www.cs.usfca.edu/~galles/visualization/AVLtree.html

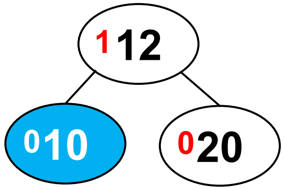
Distribute these 4 to students

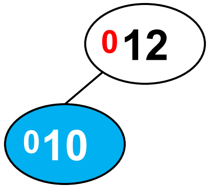
FOR PRINT

Insert(n)

- If empty tree => set as root, b(n) = 0, done!
- Insert n (by walking the tree to a leaf, p, and inserting the new node as its child), set balance to 0, and look at its parent, p
  - If b(p) = -1, then b(p) = 0. Done!
  - If b(p) = +1, then b(p) = 0. Done!
  - If b(p) = 0, then update b(p) and call insert-fix(p, n)







Insert-fix(p, n)

- Precondition: p and n are balanced: {-1,0,-1}
- Postcondition: g, p, and n are balanced: {-1,0,-1}
- 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, +/-]
  - g.balance += -1
  - if g.balance == 0, return
  - if g.balance == -1, insertFix(g, p)
  - If g.balance == -2
    - If zig-zig then rotateRight(g); p.balance = g.balance = 0
    - If zig-zag then rotateLeft(p); rotateRight(g);
      - if n.balance == -1 then p.balance = 0; g.balance(+1); n.balance = 0;
      - if n.balance == 0 then p.balance = 0; g.balance(0); n.balance = 0;
      - if n.balance == +1 then p.balance = -1; g.balance(0); n.balance = 0;

Note: If you perform a rotation, you will NOT need to recurse. You are done!

Remove

- Let n = node to remove (perform BST find) and p = parent(n)
- If n has 2 children, swap positions with in-order successor and perform the next step
  - Now n has 0 or 1 child guaranteed
- If n is not in the root position determine its relationship with its parent
  - If n is a left child, let diff = +1
  - if n is a right child, let diff = -1
- Delete n and update tree, including the root if necessary
- removeFix(p, diff);

RemoveFix(n, diff)

- If n is null, return
- Let ndiff = +1 if n is a left child and -1 otherwise
- Let p = parent(n). Use this value of p when you recurse.
- If balance of n would be -2 (i.e. balance(n) + diff == -2)
  - [Perform the check for the mirror case where balance(n) + diff == +2, flipping left/right and -1/+1]
  - Let c = left(n), the taller of the children
  - If balance(c) == -1 or 0 (zig-zig case)
    - rotateRight(n)
    - if balance(c) == -1 then balance(n) = balance(c) = 0, removeFix(p, ndiff)
    - if balance(c) == 0 then balance(n) = -1, balance(c) = +1, done!
  - else if balance(c) == 1 (zig-zag case)
    - rotateLeft(c) then rotateRight(n)
    - Let g = right(c)
    - If balance(g) == +1 then balance(n) = 0, balance(c) = -1, balance(g) = 0
    - If balance(g) == 0 then balance(n) = balance(c) = 0, balance(g) = 0
    - If balance(g) == -1 then balance(n) = +1, balance(c) = 0, balance(g) = 0
    - removeFix(parent(p), ndiff);
- else if balance(n) == 0 then balance(n) += diff, done!
- else balance(n) = 0, removeFix(p, ndiff)

OLD ALTERNATE METHOD

Insert

- Root => set balance, done!
- Insert, v, and look at its parent, p
  - If b(p) = -1, then b(p) = 0. Done!
  - If b(p) = +1, then b(p) = 0. Done!
  - If b(p) = 0, then update b(p) and call insert-fix(p)

Insert-Fix

- For input node, v
  - If v is root, done.
  - Invariant: b(v) = {-1, +1}
- Find p = parent(v) and assume v = left(p) [i.e. left child]
  - If b(p) = 1, then b(p) = 0. Done!
  - If b(p) = 0, then b(p) = -1. Insert-fix(p).
  - If b(p) = -1 and b(v) = -1 (zig-zig), then b(p) = 0, b(v) = 0, rightRotate(p) Done!
  - If b(p) = -1 and b(v) = 1 (zig-zag), then
    - u = right(v), b(u) = 0, leftRotate(n), rightRotate(p)
    - If b(u) = -1, then b(v) = 0, b(p) = 1
    - If b(u) = 1, then b(v) = -1, b(p) = 0
    - Done!