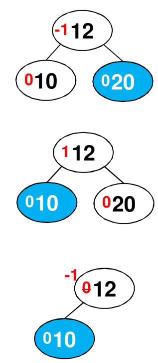
Insert(n)

- If empty tree => set n as root, b(n) = 0, done!
- Else 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) was -1, then b(p) = 0. Done!
  - If b(p) was +1, then b(p) = 0. Done!
  - If b(p) was 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, +/-]
  - `b(g) += -1 //` Update g's balance to new accurate value for now
  - `Case 1: b(g) == 0, return`
  - `Case 2: b(g) == -1, insertFix(g, p) // recurse`
  - `Case 3: b(g) == -2`
    - If zig-zig then `rotateRight(g); b(p) = b(g) = 0`
    - If zig-zag then `rotateLeft(p); rotateRight(g);`
      - Case 3a: `b(n) == -1 then b(p) = 0; b(g) = +1; b(n) = 0;`
      - Case 3b: `b(n) ==  0 then b(p) = 0; b(g) =  0; b(n) = 0;`
      - Case 3c: `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: If you perform a rotation to fix a node that is out of balance you will NOT need to recurse. You are done!

USC Viterbi School of Engineering 55

Remove

- Find node, n, to remove by walking the tree
- If n has 2 children, swap positions with in-order **successor** (or **predecessor**) and perform the next step
  - Recall if a node has 2 children we swap with its **successor or predecessor** who can have at most 1 child and then remove that node
- Let `p = parent(n)`
- If p is not NULL,
  - If n is a left child, let `diff = +1`
    - If n is a left child to be removed, the right subtree now has greater height, so add diff = +1 to balance of its parent
  - if n is a right child, let `diff = -1`
    - If n is a right child to be removed, the left subtree now has greater height, so add diff = -1 to balance of its parent
  - `diff` *will be the amount **added** to updated the balance of p*
- Delete n and update pointers
- “Patch tree” by calling `removeFix(p, diff);`

USC Viterbi School of Engineering 56

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`
  - [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)`
  - `Case 1b: b(c) ==  0   // zig-zig case`
    - `rotateRight(n), b(n) = -1, b(c) = +1 // Done!`
  - `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);`
- `Case 2: b(n) + diff == -1: then b(n) = -1; // Done!`
- `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