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;

Note: If you perform a rotation to fix a node that is out of balance you will NOT need to recurse. You are done!
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 \( p = \text{parent}(n) \)
• If \( p \) is not NULL,
  – If \( n \) is a left child, let \( \text{diff} = +1 \)
    – If \( n \) is a left child to be removed, the right subtree now has greater height, so add \( \text{diff} = +1 \) to balance of its parent
  – If \( n \) is a right child, let \( \text{diff} = -1 \)
    – If \( n \) is a right child to be removed, the left subtree now has greater height, so add \( \text{diff} = -1 \) to balance of its parent
  – \( \text{diff} \) will be the amount added to updated the balance of \( p \)
• Delete \( n \) and update pointers
• “Patch tree” by calling \( \text{removeFix}(p, \text{diff}); \)

RemoveFix\((n, \text{diff})\)

• If \( n \) is null, return
• Compute next recursive call’s arguments now before altering the tree
  – Let \( p = \text{parent}(n) \) and if \( p \) is not NULL let \( \text{ndiff} = \text{nextdiff} = \pm 1 \) if \( n \) is a left child and \( -1/\pm 1 \) otherwise
• Assume \( \text{diff} = -1 \) and follow the remainder of this approach, mirroring if \( \text{diff} = +1 \)
  • Case 1: \( \text{b}(n) + \text{diff} == -2 \)
    – [Perform the check for the mirror case where \( \text{b}(n) + \text{diff} == +2 \), flipping left/right and \(-1/\pm 1\)]
    – Let \( c = \text{left}(n) \), the taller of the children
      – Case 1a: \( \text{b}(c) == -1 \) // zig-zig case
        • \( \text{rotateRight}(n) \), \( \text{b}(n) = \text{b}(c) = 0 \), \( \text{removeFix}(p, \text{ndiff}) \)
      – Case 1b: \( \text{b}(c) == 0 \) // zig-zig case
        • \( \text{rotateRight}(n) \), \( \text{b}(n) = -1 \), \( \text{b}(c) = +1 \) // Done!
      – Case 1c: \( \text{b}(c) == +1 \) // zig-zag case
        • Let \( g = \text{right}(c) \)
          • \( \text{rotateLeft}(c) \) then \( \text{rotateRight}(n) \)
            – If \( \text{b}(g) \) was \( +1 \) then \( \text{b}(n) = 0 \), \( \text{b}(c) = -1 \), \( \text{b}(g) = 0 \)
            – If \( \text{b}(g) \) was \( 0 \) then \( \text{b}(n) = 0 \), \( \text{b}(c) = 0 \), \( \text{b}(g) = 0 \)
            – If \( \text{b}(g) \) was \( -1 \) then \( \text{b}(n) = +1 \), \( \text{b}(c) = 0 \), \( \text{b}(g) = 0 \)
              • \( \text{removeFix}(p, \text{ndiff}); \)
  • Case 2: \( \text{b}(n) + \text{diff} == -1 \): then \( \text{b}(n) = -1 \); // Done!
  • Case 3: \( \text{b}(n) + \text{diff} == 0 \): then \( \text{b}(n) = 0 \), \( \text{removeFix}(p, \text{ndiff}) \)

Note:
\( p = \text{parent of } n \)
\( n = \text{current node} \)
\( c = \text{taller child of } n \)
\( g = \text{grandchild of } n \)