Red Black Trees

Tree Properties:

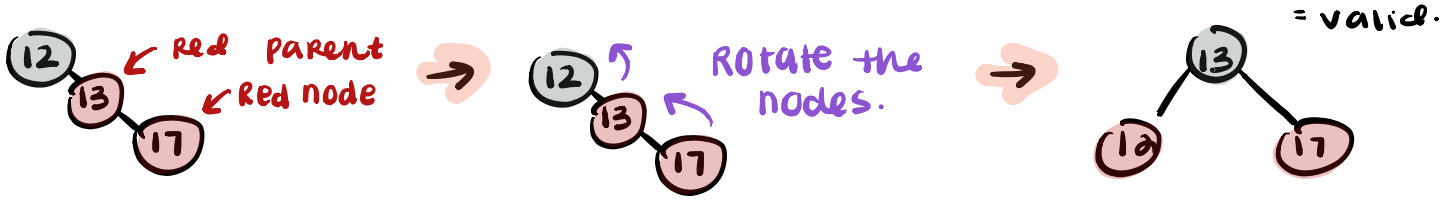
1) Tree is a valid binary search tree
2) All nodes are either Red or black
3) The root is black
4) all leaves are black
5) every red node has two black children
6) every path from root to leaf contain the same number of black nodes

→ every node is initially added in as a Red node.

What to look for:

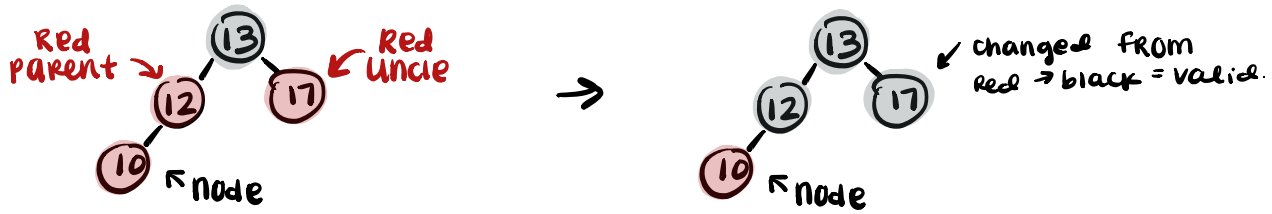
1) If the node is Red and its parent is Red → violation so you have to rotate the nodes.

Example:



2) If the node has a Red uncle and Red parent → violation so you have to change the color of the uncle and parent to black.

Example:



Rules FROM the notes:

1. every node is either Red or black
2. the root is black
3. every leaf (NULL) is black
4. IF a node is red, both children are black
5. same # black in each path

Height:

black height = $\log_2 n$

actual height = twice the black height

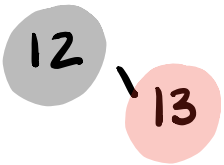
total $2 \log_2 n$

EXAMPLE IMPLEMENTATION:

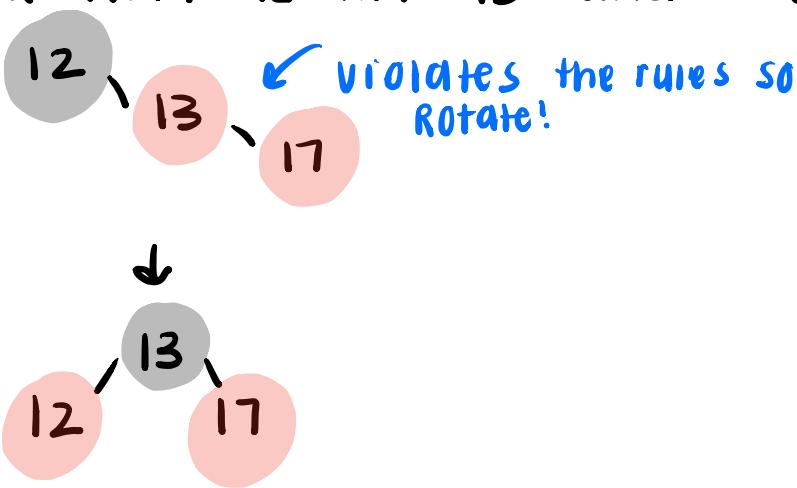
12, 13, 17, 10, 4, 6, 9, 15, 30, 25, 20, 40

Step 1) 12 automatically becomes the root which is black

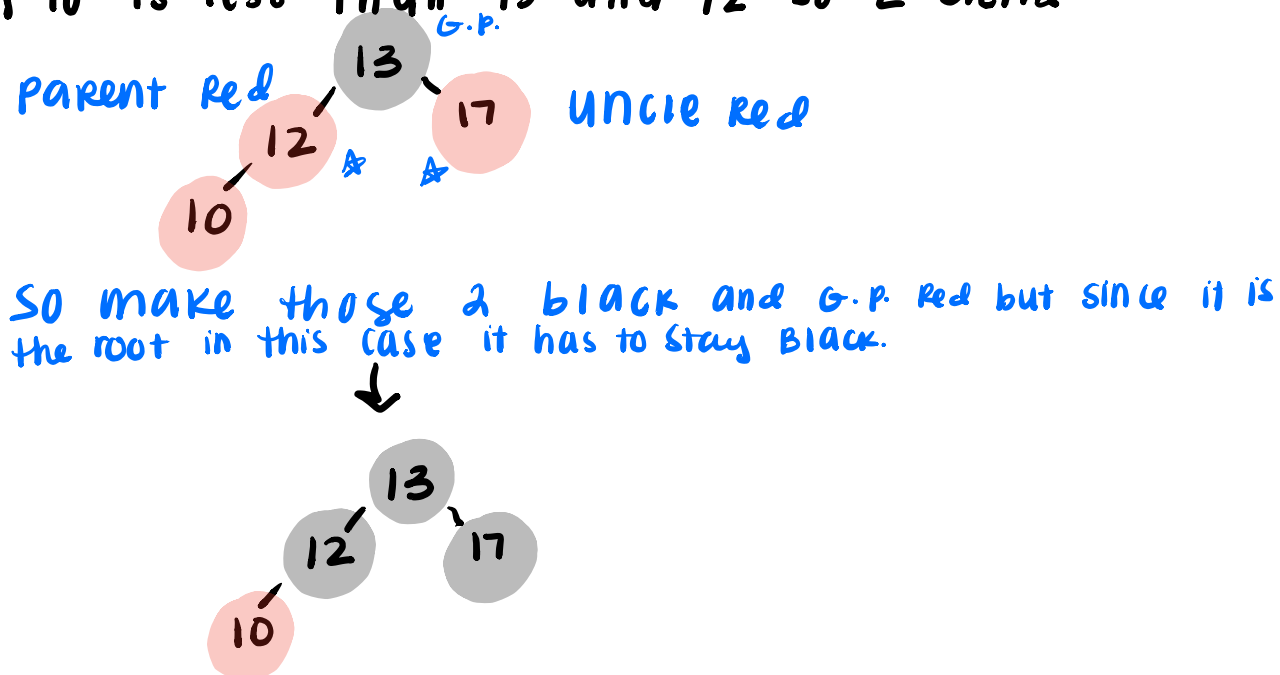
Step 2) 13 is larger 12 and so becomes a right child and is inserted as red.



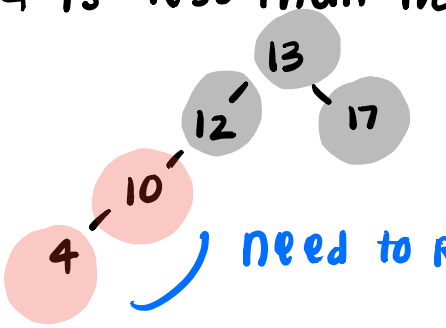
Step 3) 17 larger than 12 and 13 and inserted as red



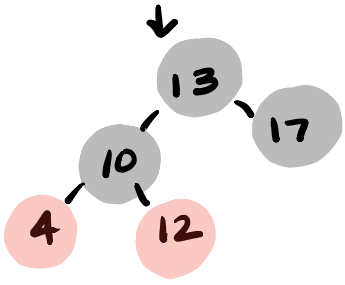
Step 4) 10 is less than 13 and 12 so L child



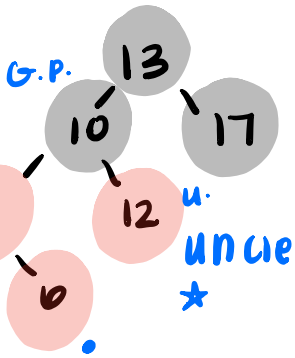
Step 5) 4 is less than them all so insert as a Red L.C.



need to Rotate!



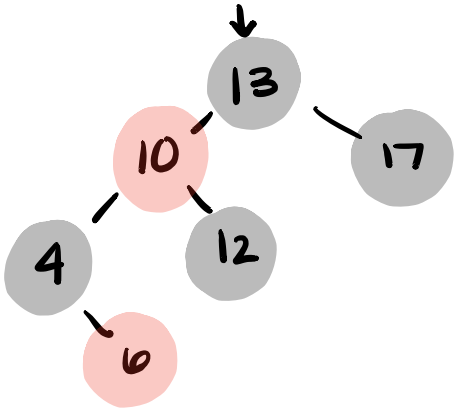
Step 6) Insert 6 which is a Right Child to 4 inserted Red



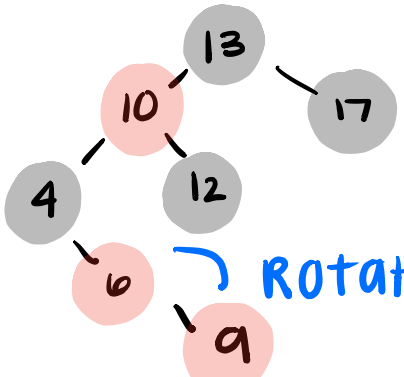
parent is Red. ★

uncle is Red. ★

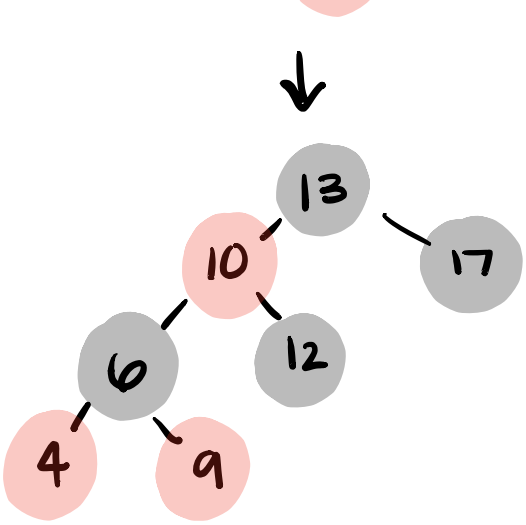
★ turn black and then the grandparent Red.



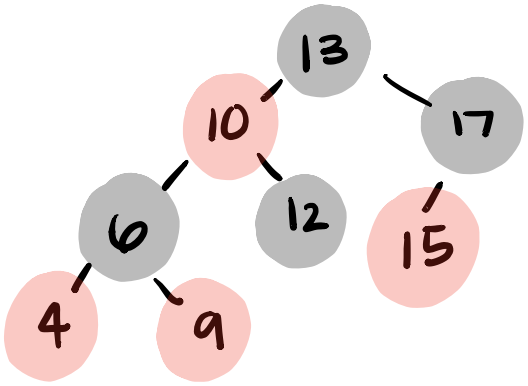
Step 7) Insert 9 which will be inserted as a Red Right Child to 6.



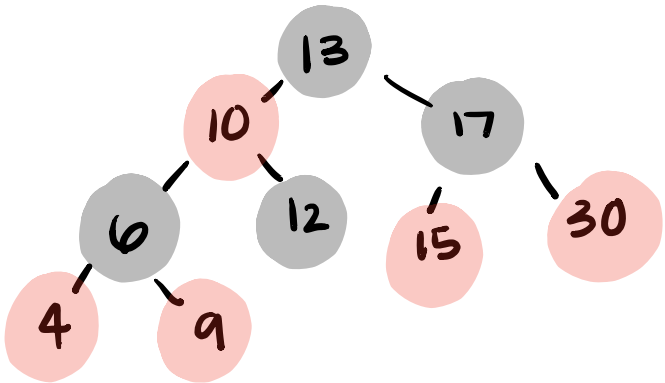
Rotate!



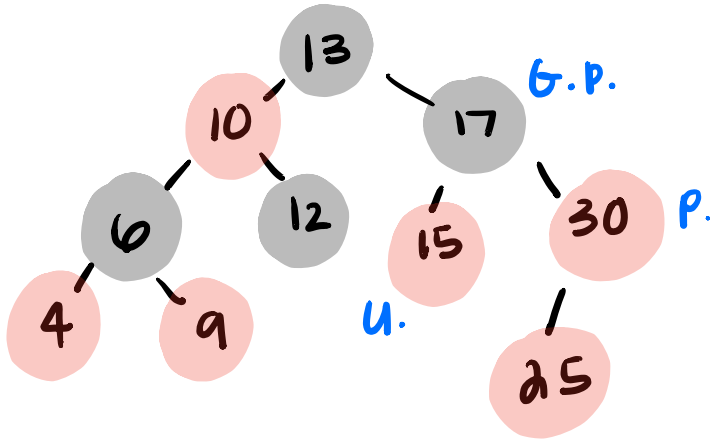
Step 8) Insert 15 which would be a Red R.C. to 17



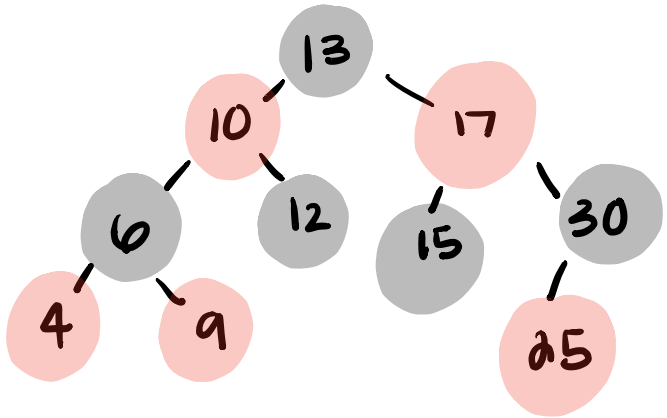
Step 9) Insert 30 which becomes a Red Right child to 17.



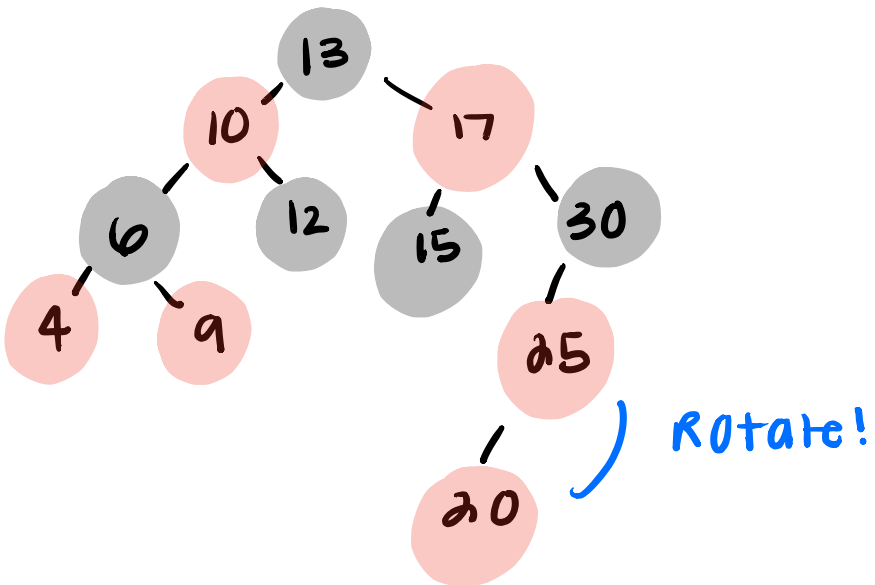
Step 10) Insert 25 which is a left child to 30

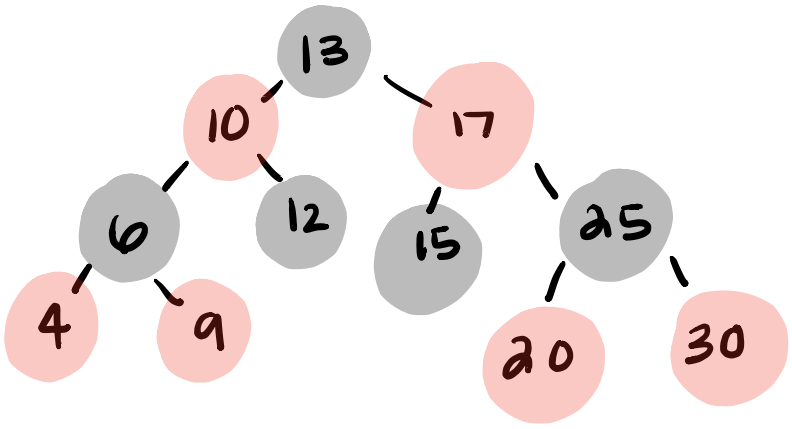


Uncle & parent are red so turn the black and turn the grand parent red.

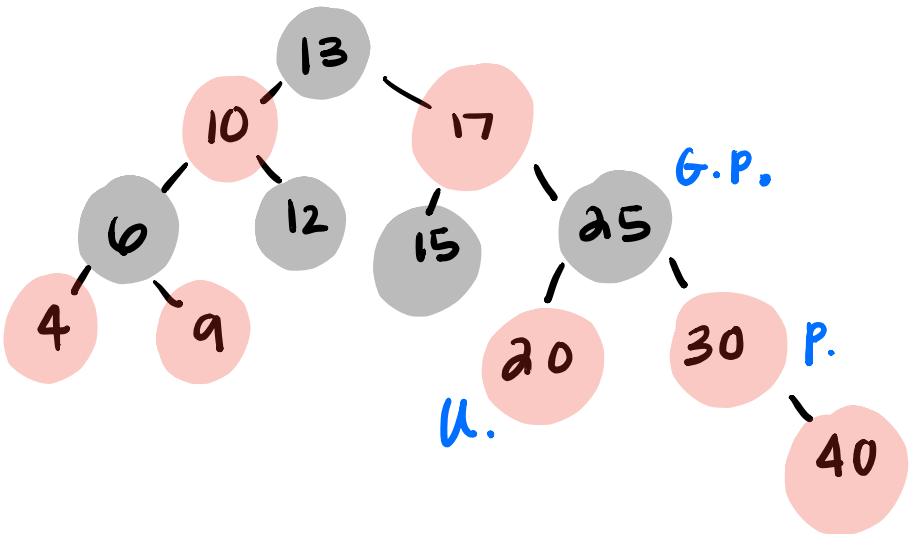


Step 11) insert 20 as a leftchild to 25



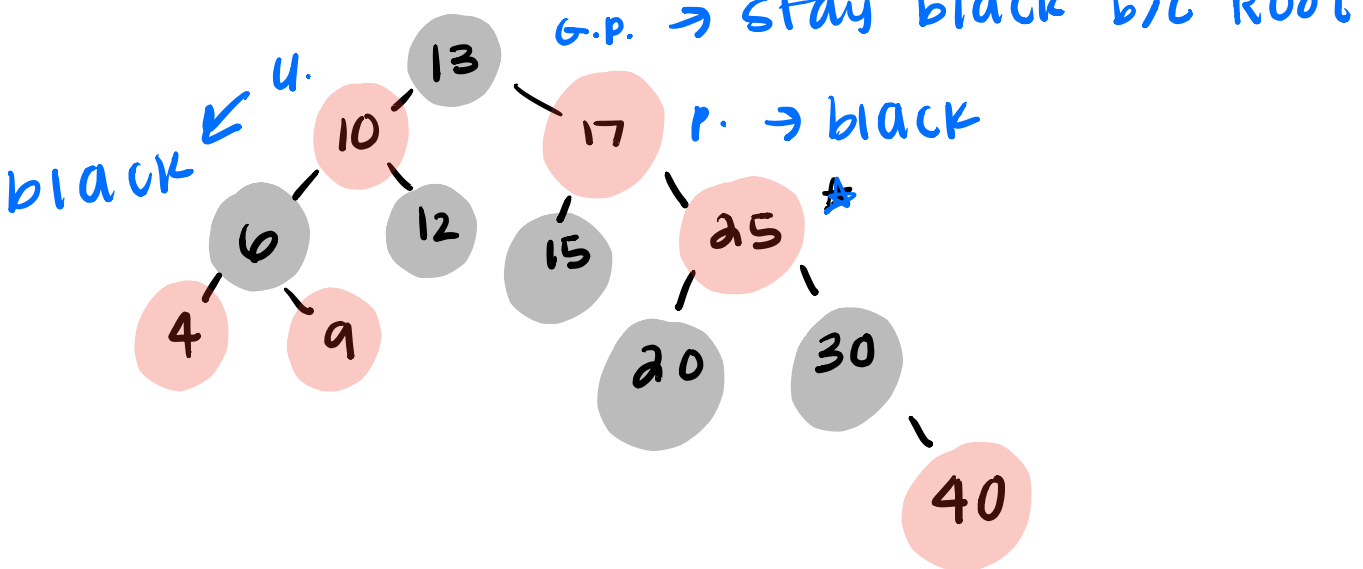


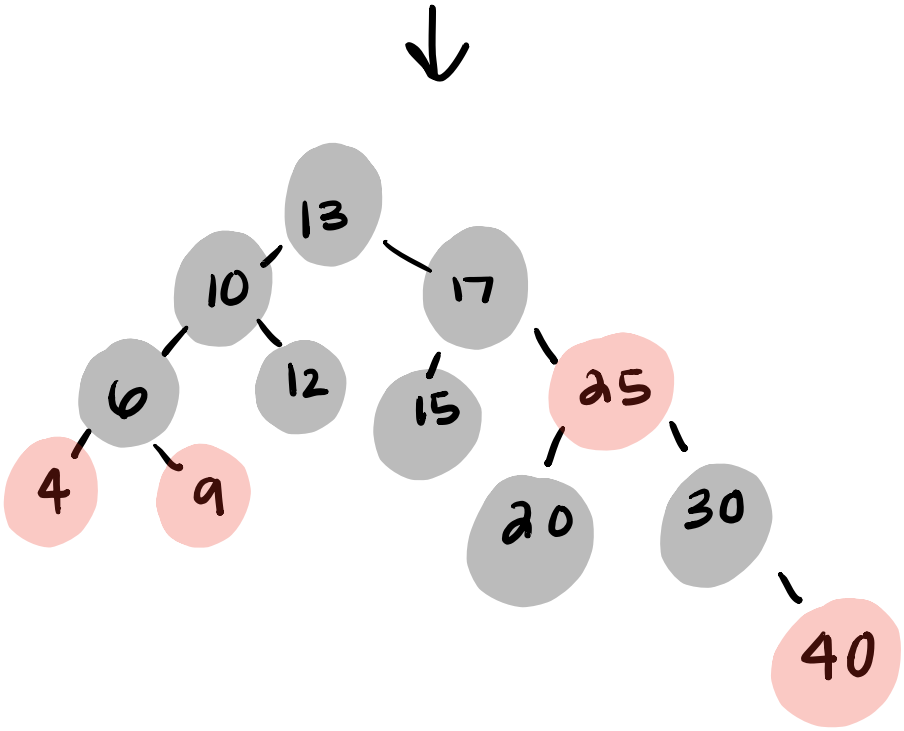
Step 12) insert 40 as a Right child to 30.



U. & P. → black

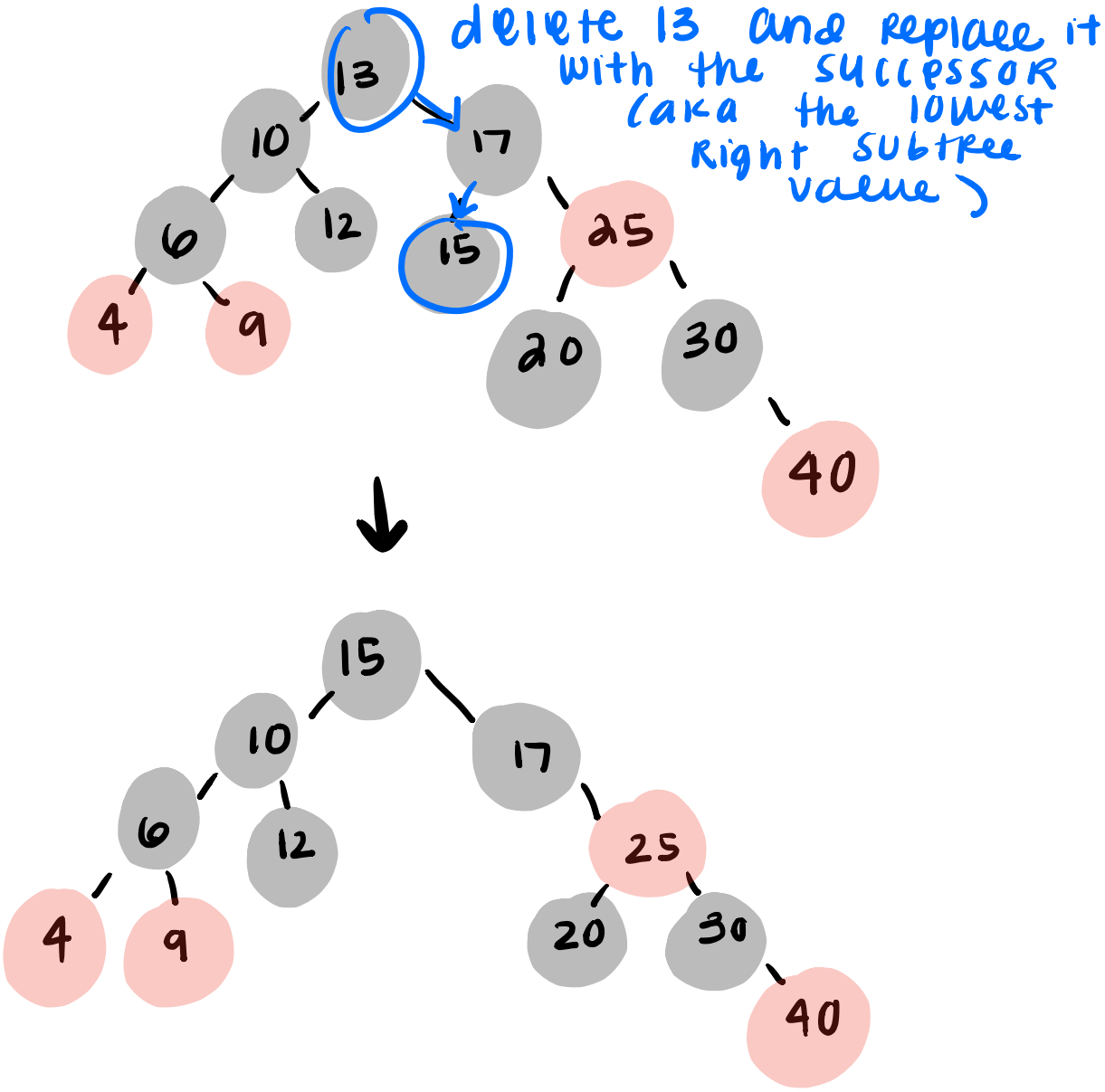
G.P. → Red.





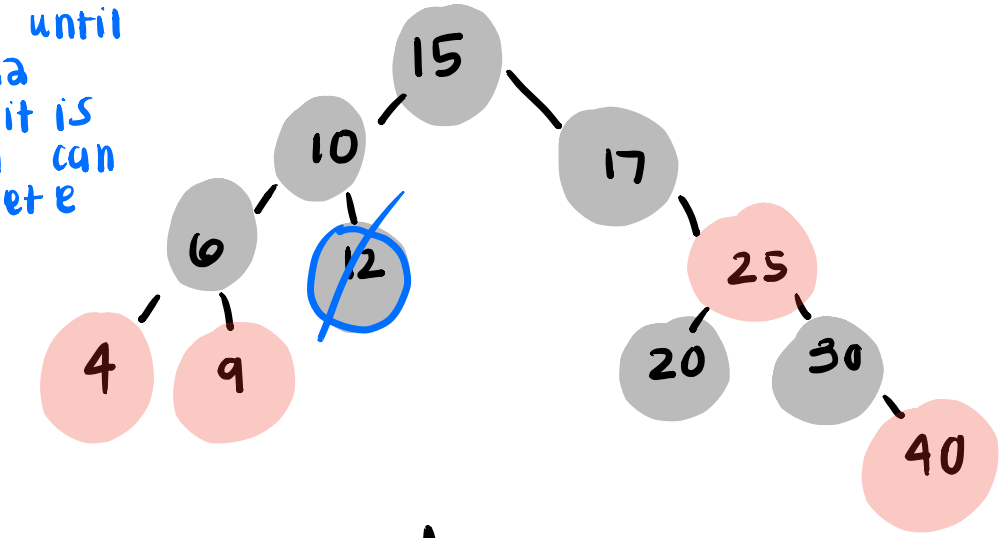
Done!

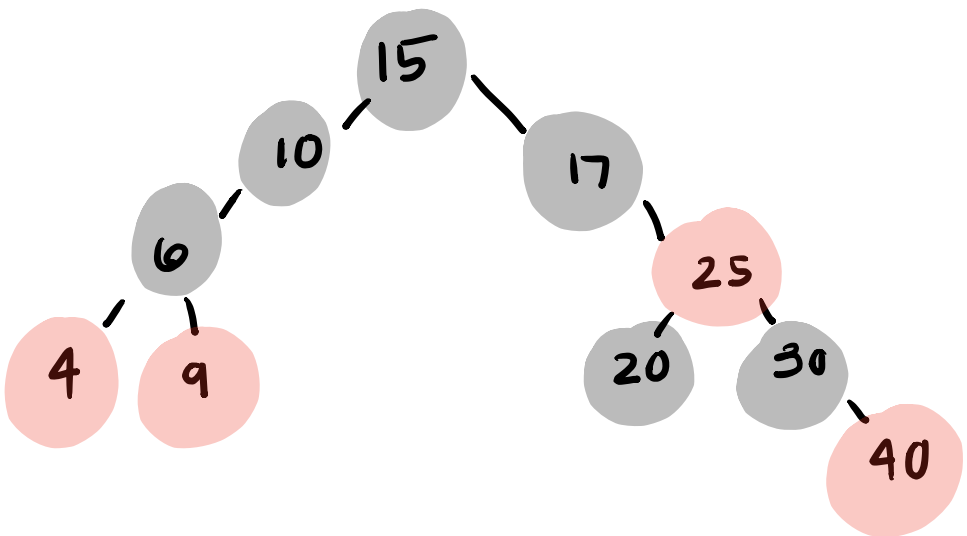
Delete Root


delete 13 and replace it with the successor (aka the lowest Right subtree value)
13
10
17
6
12
15
25
4
9
20
30
40
15
10
17
6
12
25
4
9
20
30
40

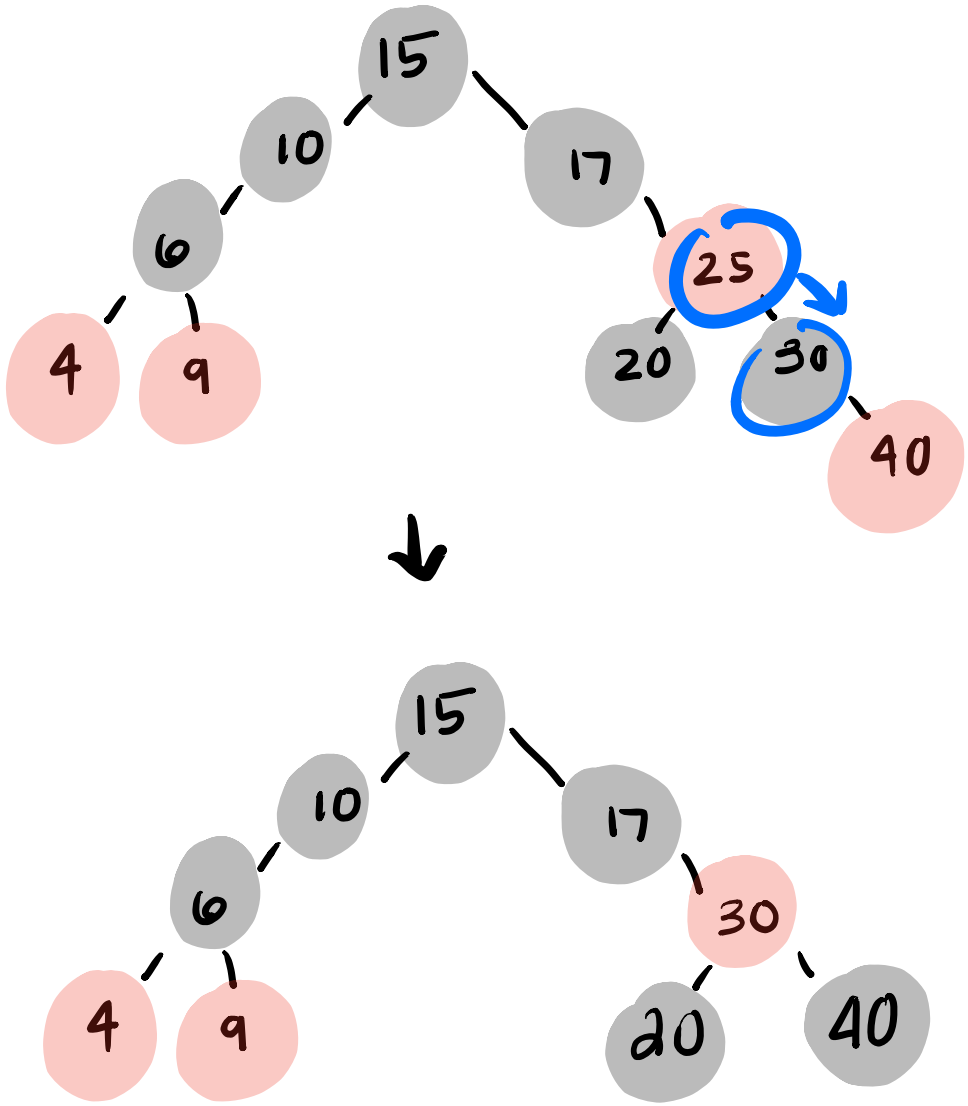
Delete 12

traverse until you find 12 and since it is a leaf you can just delete it





Delete 25



Code for a RBT:

```
def delete(self, key):
    # Same as binary tree delete, except we call rb_delete fixup at the end.
    # This is copied from the updated code from Matt on piazza.
    z = self.getNode(key)
    if z.leftChild is self.sentinel or z.rightChild is self.sentinel:
        y = z
    else:
        y = z.findSuccessor()

    if y.leftChild is not self.sentinel:
        x = y.leftChild
    else:
        x = y.rightChild

    if x is not self.sentinel:
        x.parent = y.parent

    if y.parent is self.sentinel:
        self.root = x

    else:
        if y == y.parent.leftChild:
            y.parent.leftChild = x
        else:
            y.parent.rightChild = x

    if y is not z:
        z.key = y.key

    if y.color == 'black':
        if x is self.sentinel:
            self._rb_Delete_Fixup(y)
        else:
            self._rb_Delete_Fixup(x)
        self._rb_Delete_Fixup(x)
```

```
def insert(self, key):
    """
    Add a key to the tree. Don't forget to fix up the tree and balance the nodes.

    Insert always needs to search to the bottom of the tree to find where where
    the new key (node z) should be.

    You put z there and color it read to maintain the black-height, but then
    you need to check if this causes a problem with two reds in a row, and
    if this is the case, then we fix this issue by color shifting/rotation which
    is seen later on.

    """
    z = RB_Node(key)  # creates node
    y = self.sentinel  # basically means null
    x = self.root

    if x is None:
        x = self.sentinel

    while x != self.sentinel:
        y = x
        if z.key < x.key:
            x = x.leftChild
        else:
            x = x.rightChild

    z.parent = y

    if y == self.sentinel:
        self.root = z

    elif z.key < y.key:
        y.leftChild = z

    else:
        y.rightChild = z

    z.leftChild = self.sentinel
    z.rightChild = self.sentinel
    z.color = "Red"
    self._rbInsertFixup(z)
```

```
def _rbInsertFixup(self, z):
    """
    Function to balance your tree after inserting

    Disclaimer: This code is based off of section 13.3 of the textbook.

    This is fixing the double red case after inserting as mentioned in the
    description of the insert function above.

    For the current node z, and it and it's parent is red and z
    is the unclude of y and we now have two cases to check:
        z is red ->
            color shift
            then check again for double red
        z is black ->
            rotate and then done
    """
    while z.parent.color == "Red":
        if z.parent == z.parent.parent.leftChild:
            y = z.parent.parent.rightChild
            if y.color == "Red":
                z.parent.color = "Black"
                y.color == "Black"
                z.parent.parent.color = "Red"
                z = z.parent.parent
            else:
                if z == z.parent.rightChild:
                    z = z.parent
                    self.leftRotate(z)
                z.parent.color = "Black"
                z.parent.parent.color = "Red"
                self.rightRotate(z.parent.parent)
        else:
            y = z.parent.parent.leftChild
            if y.color == "Red":
                z.parent.color = "Black"
                y.color = "Black"
                z.parent.parent.color = "Red"
                z = z.parent.parent
            else:
                if z == z.parent.leftChild:
                    z = z.parent
                    self.rightRotate(z)
                z.parent.color = "Black"
                z.parent.parent.color = "Red"
                self.leftRotate(z.parent.parent)

    self.root.color = "Black"
```

```
def _rb_Delete_Fixup(self, x):
    """
    This function receives a node, x, and fixes up the tree, balancing from x.

    Disclaimer: This code was based from the textbook.

    If x is red:
        change the color to black and done fixing.

    If x is black:
        transform the tree and move x up, until:
            x points to a red node or x is the root
    A key point in this function is to always set the color of x to black
    at the end of the function, which you can see on the last line.
    """
    while x is not self.root and x.color is "Black":
        if x is x.parent.leftChild:
            w = x.parent.rightChild
            if w.color is "Red":
                w.color = "Black"
                x.parent.color = "Red"
                self.leftRotate(x.parent)
                w = x.parent.rightChild
            if w.leftChild.color is "Black" and w.rightChild.color is "Black":
                w.color = "Red"
                x = x.parent
            else:
                if w.rightChild.color is "Black":
                    w.leftChild.color = "Black"
                    w.color = "Red"
                    self.rightRotate(w)
                    w = x.parent.rightChild
                w.color = x.parent.color
                x.parent.color = "Black"
                w.rightChild.color = "Black"
                self.leftRotate(x.parent)
                x = self.root

        else:
            w = x.parent.leftChild
            if w.color is "Red":
                w.color = "Black"
                x.parent.color = "Red"
                self.rightRotate(x.parent)
                w = x.parent.leftChild
            if w.rightChild.color is "Black" and w.leftChild.color is "Black":
                w.color = "Red"
                x = x.parent
            else:
                if w.leftChild.color is "Black":
                    w.rightChild.color = "Black"
                    w.color = "Red"
                    self.leftRotate(w)
                    w = x.parent.leftChild
                w.color = x.parent.color
                x.parent.color = "Black"
                w.leftChild.color = "Black"
                self.rightRotate(x.parent)
                x = self.root
    x.color = "Black"
```

```
def leftRotate(self, currentNode):
    # perform a left rotation from a given node
    y = currentNode.rightChild
    currentNode.rightChild = y.leftChild

    if y.leftChild is not self.sentinel:
        y.leftChild.parent = currentNode

    y.parent = currentNode.parent

    if currentNode.parent is self.sentinel:
        self.root = y
    elif currentNode is currentNode.parent.leftChild:
        currentNode.parent.leftChild = y
    else:
        currentNode.parent.rightChild = y

    y.leftChild = currentNode
    currentNode.parent = y
```

```
def rightRotate(self, currentNode):
    """
    This function performs a right rotation from a given node
    """
    x = currentNode.leftChild
    currentNode.leftChild = x.rightChild

    if x.rightChild is not self.sentinel:
        x.rightChild.parent = currentNode

    x.parent = currentNode.parent

    if currentNode.parent is self.sentinel:
        self.root = x
    elif currentNode is currentNode.parent.rightChild:
        currentNode.parent.rightChild = x
    else:
        currentNode.parent.leftChild = x

    x.rightChild = currentNode
    currentNode.parent = x
```