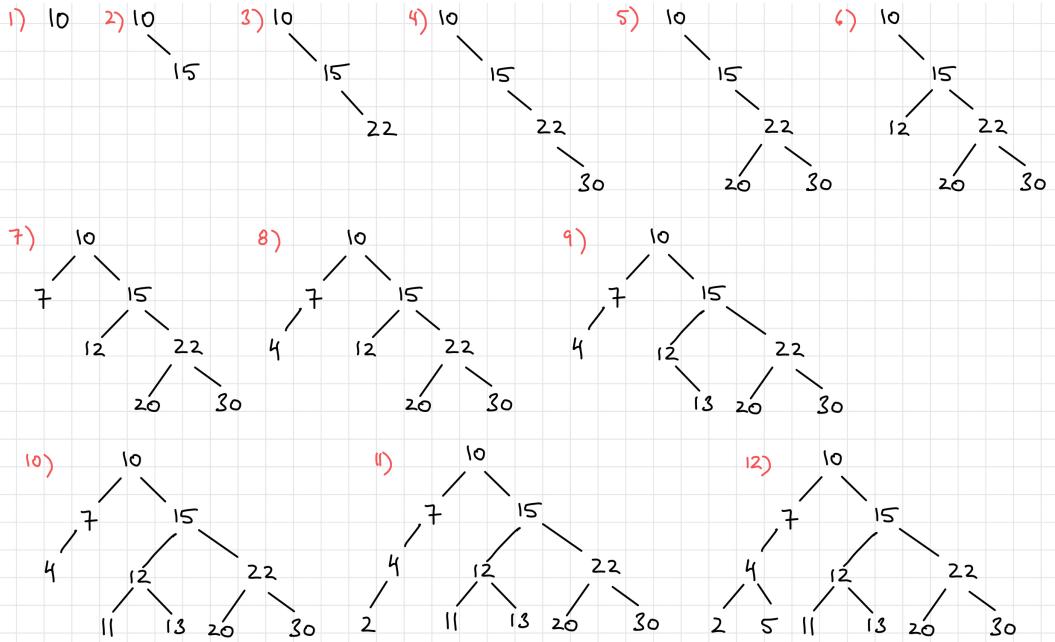


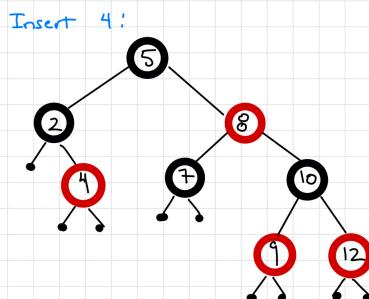
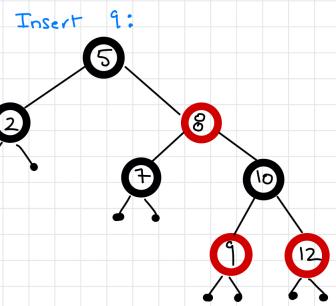
Practice Set 6: solutions

Problem 1

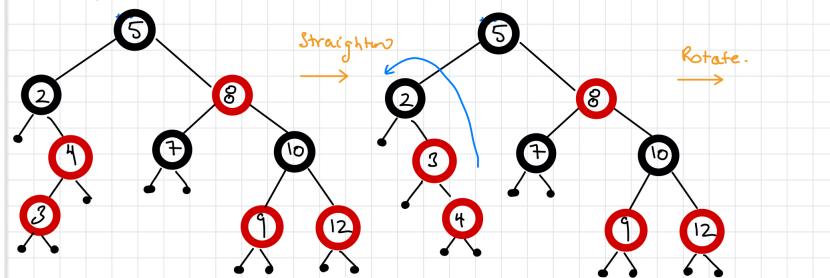


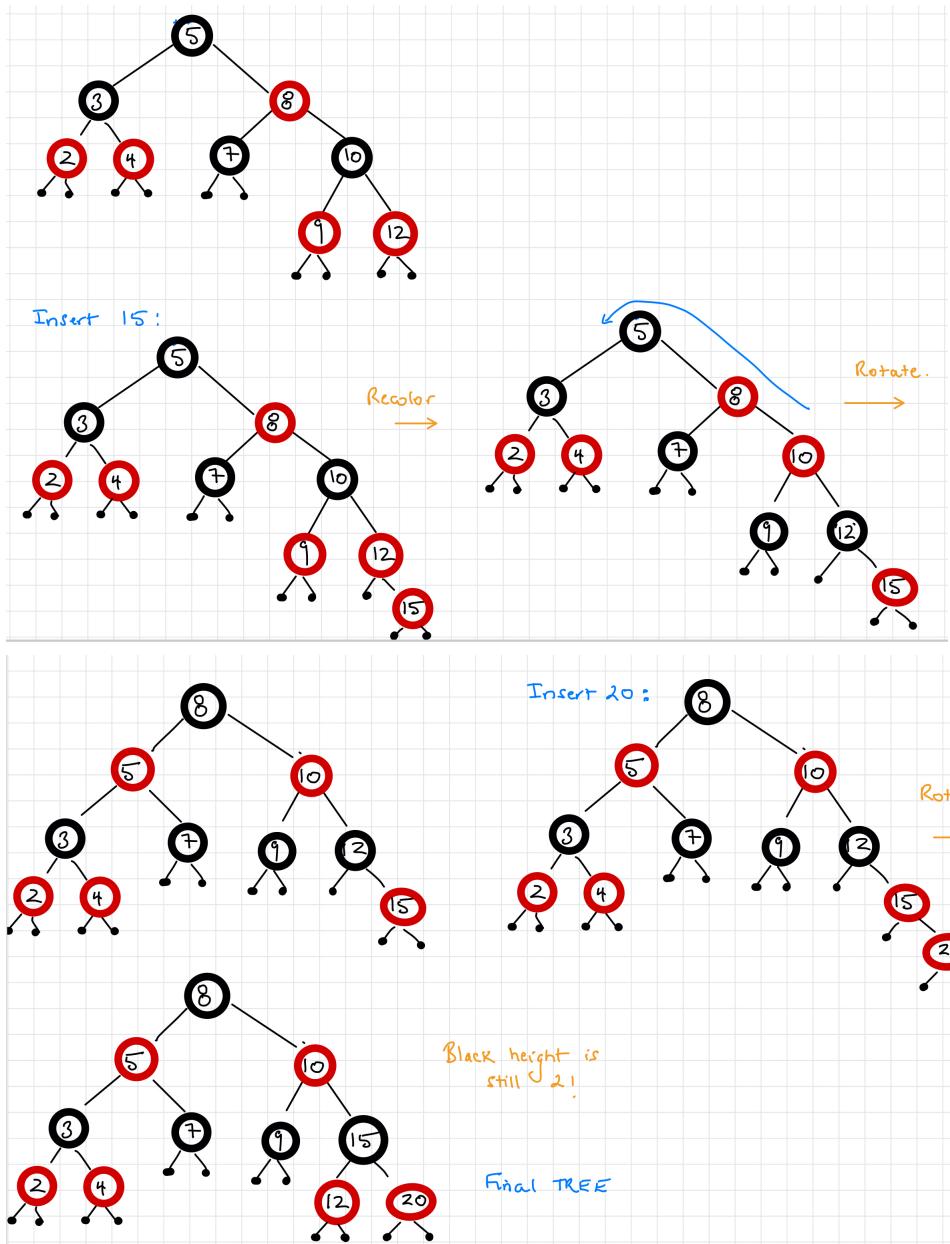
Delete 7: Left child of 7 shifts up.

Delete 15: Replace with pred. 13.



Insert 3:





Problem 2

The minimum is the left-most node in the tree, and the maximum is the right-most node in the tree.

```
Find-Max(T)
  x = T
  while x.right != NIL
    x = x.right
  return x
```

```
Find-Min(T)
  x = T
  while x.left!= NIL
    x = x.left
  return x
```

In order to find the successor of a node x , we must consider two cases: it may be that the node x has a right child. In this case, the successor is the minimum node in the right subtree. If the node x has no right child, then the successor is somewhere on the path from the root to x . The algorithm below searches down at most one path in the tree. In the first case, it searches down a path from node x . In the second case, it searches down a path from the root to x . Therefore in either case, the algorithm runs in time $O(h)$.

Successor(T, x)

If $x.right$ is not NIL # x has a right child
 return Find-Min($x.right$)

```

else    # z has no right child
y = T
while y is not x
    if y.key > x.key    # x is on the left path
        success = y
        y = y.left
    else    # x is on the right path
        y = y.right
return success    # z has only a right child

```

Problem 3

We need to link the parent of x to the new node y . This is done by checking if x is a left or right child of its parent.

Replace(x,y)

```

If x.parent.left = x    # x is a left child
    x.parent.left = y
else x.parent.right = y    # x is a right child
if y != NIL
    y.parent = x.parent

```

The above function runs in constant time, as it only re-assigned reference pointers.

Problem 4

The algorithm **BST-Delete(T,z)** deletes node z from the tree T with the assumption that z is a node in the tree, and returns a reference to the new tree.

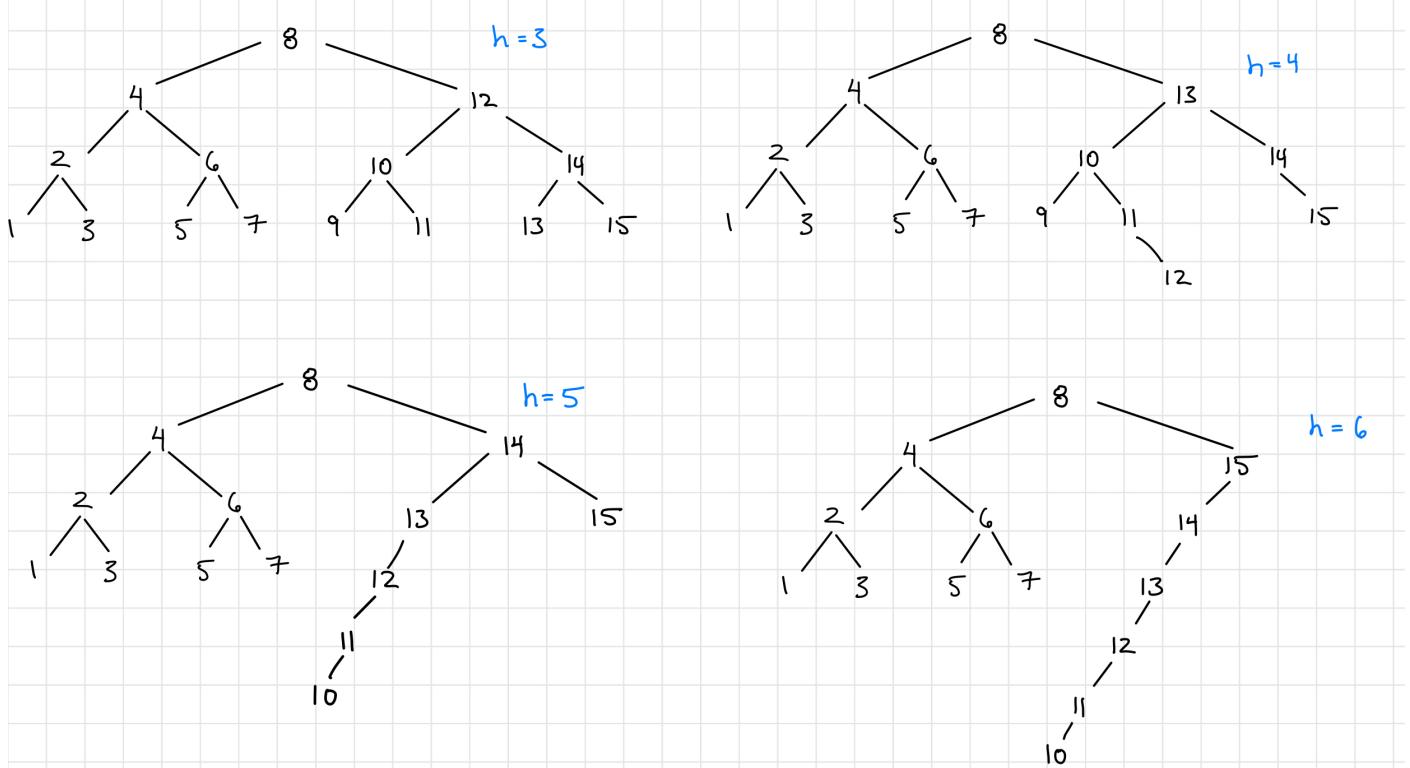
BST-Delete(T,z)

```

If z = T and z.left = NIL    # z is the root, return right child
    return z.right
else if z = T and z.right = NIL    # z is the root, return left child
    return z.left
else:
    If z.left = NIL and z.right = NIL    # z is a leaf
        If z = z.parent.left
            z.parent.left = NIL
        else z.parent.right = NIL
    else if z.left = NIL    # z has only a right child
        Replace(z,z.right)
    else if z.right= NIL    # z has only a left child
        Replace(z,z.left)
    else    # z has two children
        y = Find-Min(z.right)
        z.key = y.key
        Replace(y,y.right)
return T

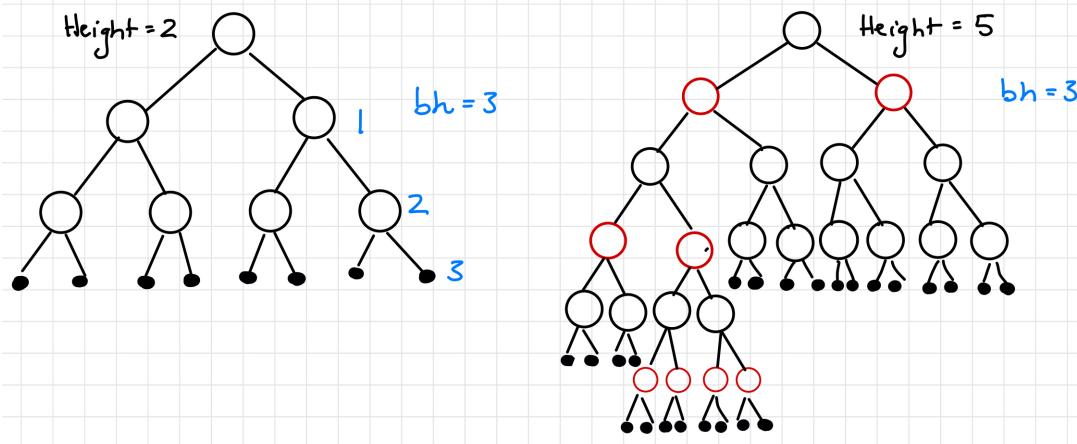
```

Problem 5



The shortest RB tree we can build is when all the nodes are black. For example, when the black-height of the tree is 3, the height of the tree is 2. Therefore for a tree with black-height b , the shortest tree has height $b - 1$.

The longest red-black path we can build in a tree is a path that alternates red and black nodes. For example, when the black-height is 3, the maximum height of the tree is 5. An example of each case is shown below. Therefore when the black height is b , the tree with the maximum height has height $2b - 1$.



Problem 6

In class we showed the $\text{INORDER}(x)$ runs in time $T(n) \leq T(|L_x|) + T(|R_x|) + c$ where L_x and R_x denote the left and right subtrees of x , and $|L_x|$ refers to its size. We can prove this is $O(n)$ using substitution. Assume that $L_x = k$ and $R_x = m$. Then $m + k + 1 = n$.

Goal: $T(n) \leq dn$ for all n

Assume: $T(k) \leq dk$ and $T(m) \leq dm$

Therefore:

$$\begin{aligned}
 T(n) &= T(k) + T(m) + c \\
 &\leq dk + dm + c \\
 &= d(k + m) + c \\
 &= d(n - 1) + c = dn - d + c \\
 &\leq dn
 \end{aligned}$$

as long as $d > c$

The pseudo-code for pre and post order are almost the same, we just alter the step at which we print the node's key. Both pre-order(x) and post-order(x) have a runtime recurrence of $T(n) = T(L_n) + T(R_n) + c$, which is the same as that of Inorder. Therefore all three algorithms run in time $O(n)$

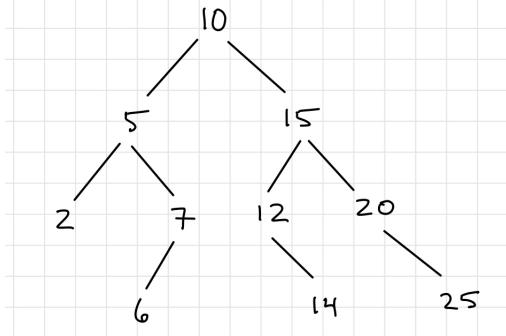
```

Preorder(x)
  if x is not NIL
    Print(x.key)
    Preorder(x.left)
    Preorder(x.right)
  
```

```

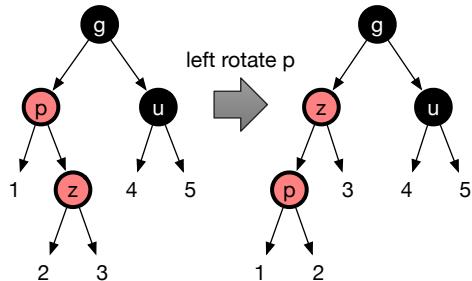
Postorder(x)
  if x is not NIL
    Postorder(x.left)
    Postorder(x.right)
    Print(x.key)
  
```

Given any one of the above “orders”, we can determine the exact shape of the tree. For example, given the pre-order 10, 5, 2, 7, 6, 15, 12, 14, 20, 25 we know that 10 must be the root, and everything less than 10 is on the left subtree, and everything larger than 10 is in the right subtree.



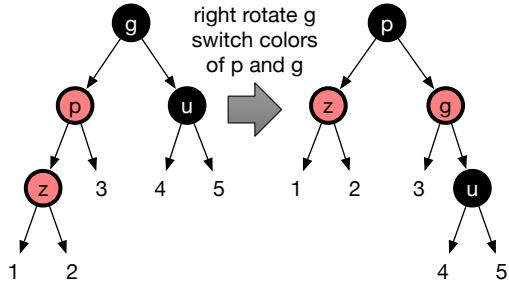
Problem 7

- The original RB tree (before the new item is inserted) is a valid RB tree. Therefore in the figure below, we know that the number of black nodes on paths 1,2,3 is the same. Likewise the number of black nodes on paths 4,5 is the same. Now let's see what changes after the rotation. Node U has the same subtrees, so its black height is the same. Node P has paths through 1,2,3 before the rotation, and through paths 1,2 after, all of which have the same number of black nodes. Node Z had paths through 2,3 before, and through 1,2,3 after, all of which have the same number of black nodes. Node G had paths through 1,2,3 before and after, (so no change there) AND through paths U+ 4,5 before and after, also no change there. So the black heights of all nodes is the same after the rotation.



Case 2 may also involve another type of rotation, shown below. Node Z and node U have identical subtrees before and after the rotations, so their black heights are the same. Node P had paths through

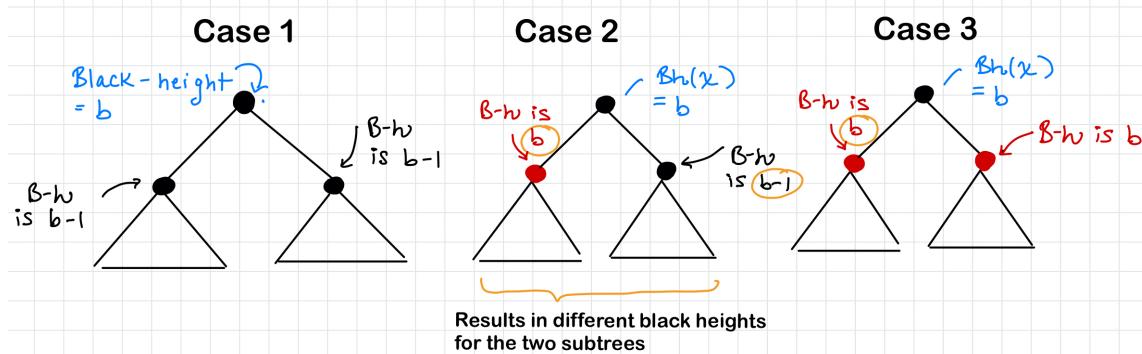
1,2,3 before the rotation and through 1,2,3 and U+ 4,5 after the rotation. Note that the number of black nodes through U to 4,5 is the *same* as the number of black nodes through paths 1,2,3 (since the original tree was red-black). Therefore the black height of P has not changed. Finally, node G had paths through 1,2,3, and U+4,5 before the rotation, and still has paths through 3 and U+4,5 after the rotation, so the black height is the same.



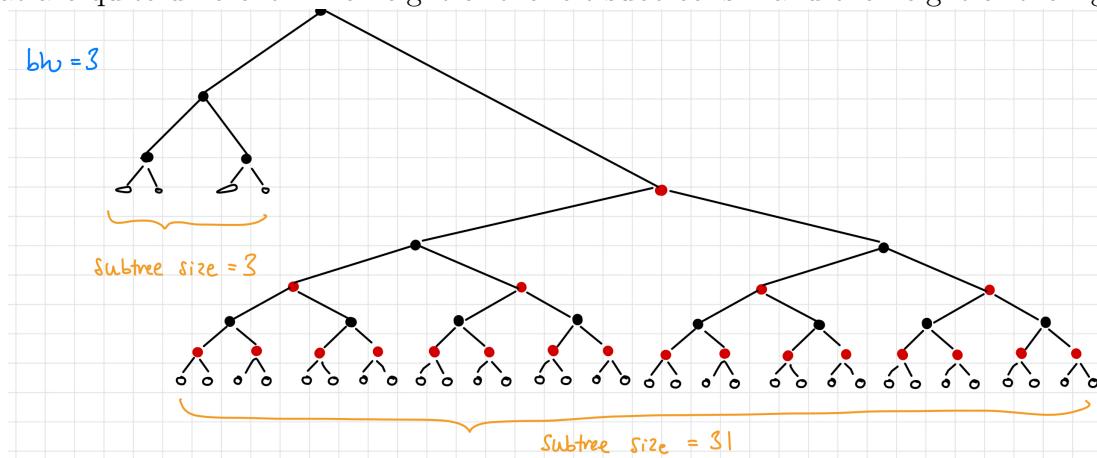
- A new node does not always change the black-height of tree. The RB-repair algorithm performs rotations in Case 2(which do not alter the number of black nodes on a path). RB-repair performs recolorings in Case1 (the parent and the uncle are recolored black, and the grandparent is colored red). Generally this recoloring does *not* change the number of black nodes on the path from the root. However, it may be that the grandparent is in fact the *root* node. In this case, the grandparent is left as a black node. The black height of the entire tree has increased by one.

Problem 8

If the children of the root, x , have different colors, then their black heights will differ by one. Otherwise the black heights of each child is the same.



If a red-black tree has black-height 3, the left subtree could be completely full of black nodes, in which case there would be 3 nodes. The right subtree could alternate red and black nodes. If we alternate as long as possible the result is 31 nodes in the right subtree. Therefore red-black trees can have subtree-sizes that are quite different. The height of the left subtree is 1 and the height of the right subtree is 4.



Problem 9

BlackHeight(x)

```
if (x is NilNode) return 0
else
    a = BlackHeight(x.left)
    if a = -1 return -1
    if (x.left.color = black)
        a++
    b = BlackHeight(x.right)
    if b = -1 return -1
    if (x.right.color = black)
        b++
    if (a ≠ b)
        return -1
    else
        return a
```

Problem 10:

If we rewrite TREE-INSERT this as a recursive algorithm, we must ensure that we write something that allows us to link z to its parent. Therefore, we write the recursive algorithm so that it “follows” the parent node down the tree. The algorithm below inserts z into T . If the tree is empty, it returns node z as the root.

TREE-INSERT(T,z)

```
if T is NIL
    return z
else
    if z.key < T.key
        if T.left = NIL
            T.left = z
            z.parent = T
        else TREE-INSERT(T.left, z)
    else
        if T.right = NIL
            T.right = z
            z.parent = T
        else TREE-INSERT(T.right, z)
    return T
```

Problem 11

If the tree is an AVL tree, then the algorithm below returns the height of the tree (which can be interpreted as true) and if it is NOT an AVL tree, it returns -2 (interpreted as false). The idea is to use recursion to determine the heights of the left and right subtrees, and return false if either of these subtrees are not AVL trees, or if the difference in the heights of the subtrees is more than 1. For the purpose of recursion, we use a height of -1 for any empty tree so that we can distinguish it from the height of a single node, which is 0.

IsAVL(x)

```
if x = NIL return -1
a = IsAVL(x.left)
b = IsAVL(x.right)
if (a = -2 OR b = -2 OR |a - b| > 1)
    return -2
```

```
else  
    return max(a,b) +1
```