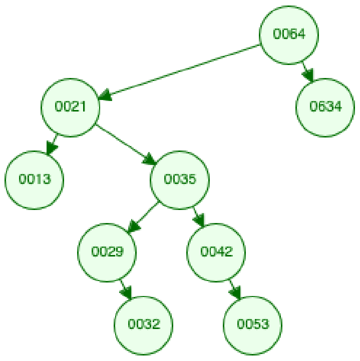
HW Solution #3

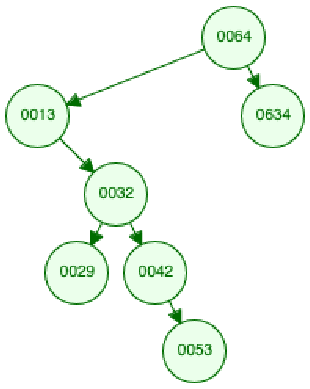
P1: (a)



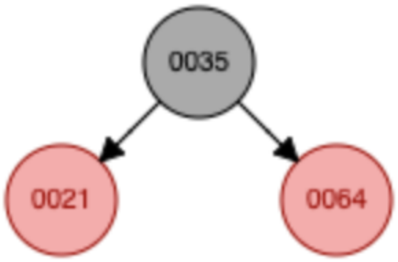
(b) $\frac{20}{9}$

(c) There are multiple possibilities, as long as they is balanced, ie., the difference of subtrees heights is at most 1. We can observe that it has less depth than the regular BST.

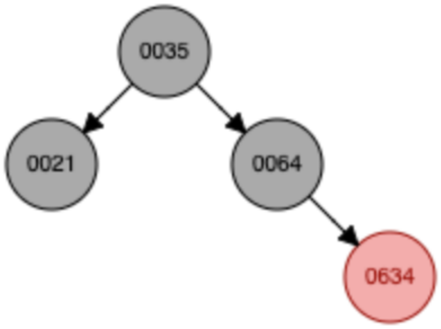
(d)



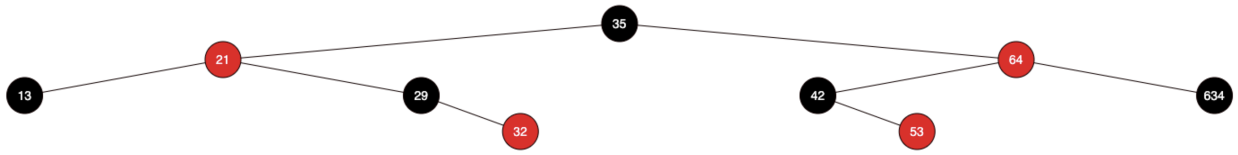
P2: (a)



(b)



(c)



P3: The pseudocode:

```
TREE-PREDECESSOR(x)
    if x.left != NIL
        return TREE-MAXIMUM(x.left)
    y = x.p
    while y != NIL and x == y.left
        x = y
        y = y.p
    return y
```

P4: Suppose the node x has two children. Then it's successor is the minimum element of the BST rooted at $x.right$. If it had a left child then it wouldn't be the minimum element. So, it must not have a left child. Similarly, the predecessor must be the maximum element of the left subtree, so cannot have a right child.

P5: The largest is a path with half black nodes and half red nodes, which has $2^{2k} - 1$ internal nodes. The smallest is a pth with all black nodes, which has $2^k - 1$ internal nodes.

P6: Let think from the base case.

For $n = 1$, it is obvious that there is only one black root node.

For $n = 2$, it will definitely result in one black root node and one red node. And if we have two black nodes, we violate the Rule 5.

For the general $n > 1$, we always first add a new red node and call `RB-INSERT-FIXUP`. For the rotation case, we can observe that the number of red nodes does not decrease. For the case 1 (the parent and uncle are both red), the new node is kept red. Therefore, we can conclude that in all the cases, we will at least have one red node after the insertion.

Think of red node is a mechanism for us to keep the balance.

P7: We can do a depth-first tree traversal and counts how many times each number appears. The important thing is to realize that the tree structure itself is homogeneous. At each node z, we have two subtrees whose root nodes are $z.left$ and $z.right$. We need to what is the *mode*, or $maxCount$ in these two subtrees. We can maintain the single $maxCount$ variable and make it update whenever we found a new mode. To find the level, at node z, we need to check whether $z.key == z.p.key$. If they are equal, then we shall increment the count of $z.key$ by one. As $z.left$ and $z.right$ can also be holding $z.key$, we need to transmit the $count$ to them. But we can be smart in figuring out how to transmit $count$ between nodes. Because it is still a binary search tree, we can do an `inorder tree walk`. So the sequence we visit the node can be inorder. Therefore we can just keep $one\ count$ variable and use it to count how many times the current number has appeared.

```
class Solution {
 public:
  vector<int> findMode(TreeNode* root) {
    int  ans; // This record the answer.
    int count = 0; // Record the times the current node.key appears
    int maxCount = 0; // Record the maximum count, or the mode, in the subtree

    inorder(root, count, maxCount, ans); // Do the recursive search from the root
       node
    return ans;
  }

 private:
  TreeNode* pred = nullptr;

 /*
  * The inorder function is a recursive function to do DFS + process the counts
  * It is using the BST property that the walk is in order.
  */
  void inorder(TreeNode* root, int& count, int& maxCount, int& ans) {
    if (root == nullptr)
      return;
    inorder(root->left, count, maxCount, ans);
    updateCount(root, count, maxCount, ans);
    inorder(root->right, count, maxCount, ans);
  }

  void updateCount(TreeNode* root, int& count, int& maxCount,
                   int& ans) {
    if (pred && pred->key == root->key)
      ++count;
    else
      count = 1;

    if (count > maxCount) {
      maxCount = count;
      ans = root->key;
    }

    pred = root;
  }
};
```