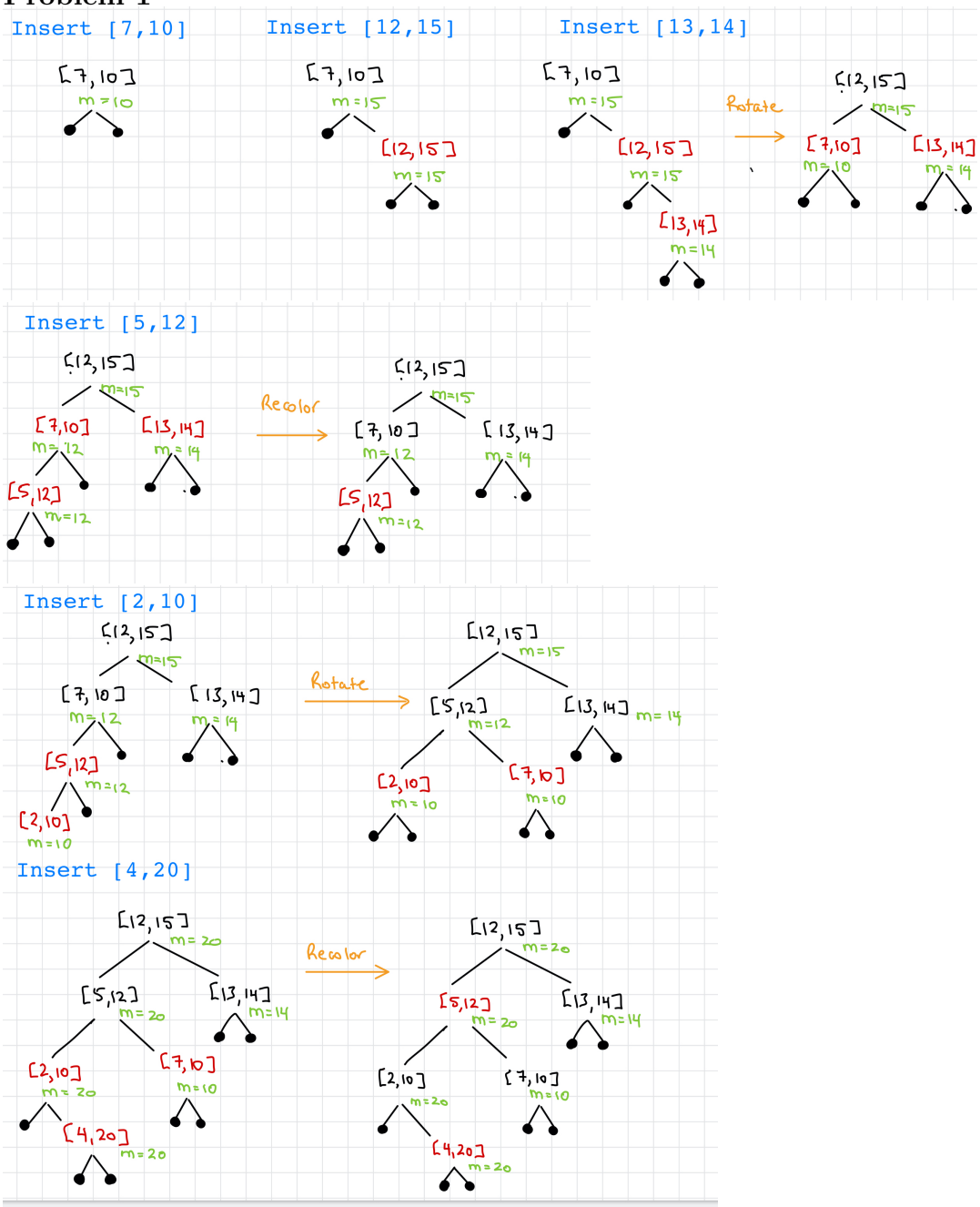
Practice Set 7: solutions

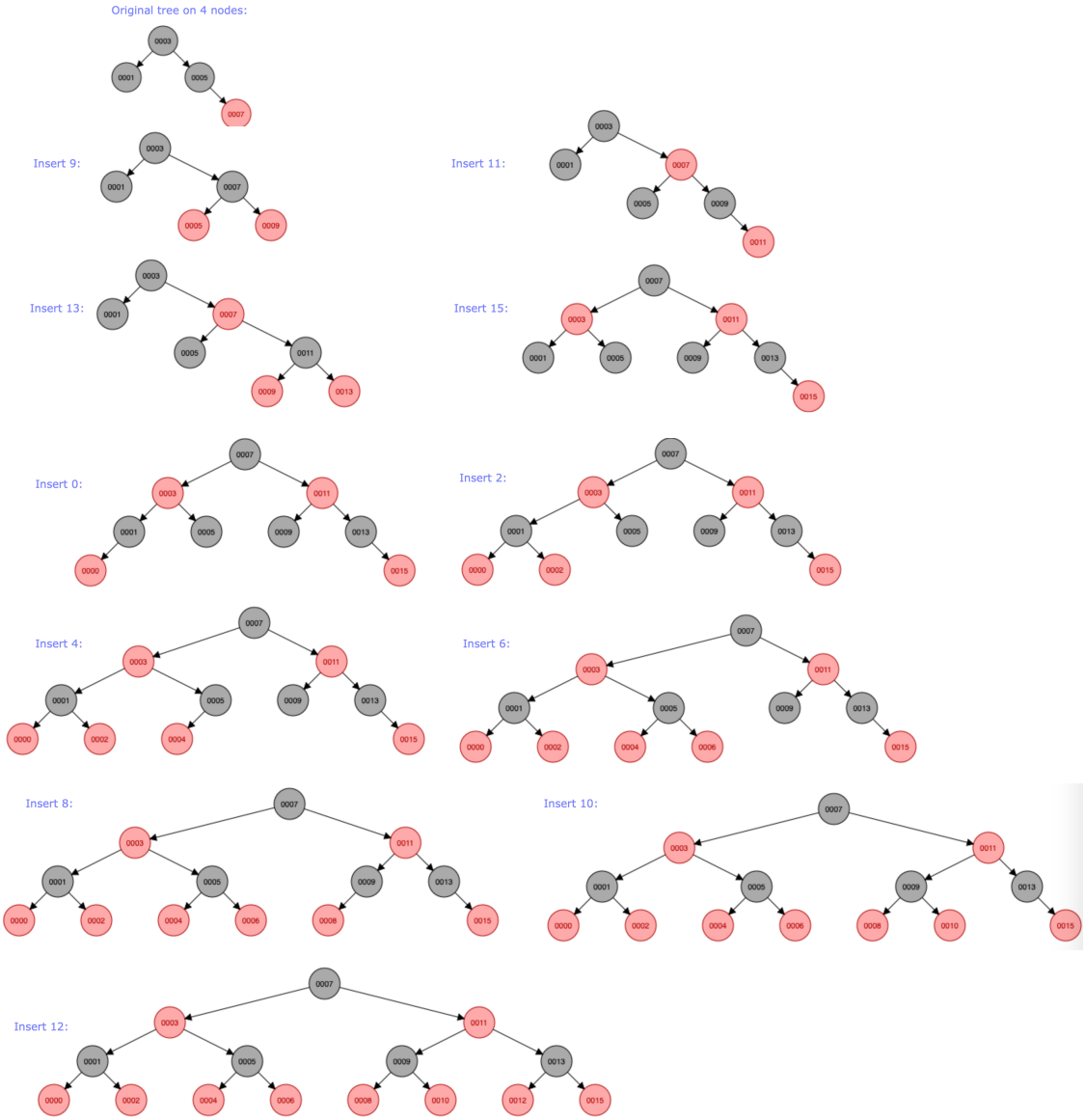
.

Problem 1



Problem 2

The black height of the tree is 2. It is possible to make 11 insertions without changing the black height. Snapshots of these insertions are shown below. The resulting tree has 15 nodes and a black-height of 2. It is not possible to add any more nodes without increasing the black height, because for a tree with black height bh, $n \leq 2^{2 \cdot bh} - 1$.



Problem 3

The solution below builds a new tree node x using the median of the array A. The elements in the array that are smaller than the median are used to build the left subtree, and the elements in the array that are larger than the median are used the build the right subtree. We set the children pointers of x to the left and right subtrees, and we also set the parent references of the left and right subtree nodes.

```
Build-BST(A,s,f)
    if s ≤ f
        q = round-down((s+f)/2)
        x = new tree node
        x.key = A[q]
        x.left = Build-BST(A,s,q-1)
        x.right = Build-BST(A,q+1,f)
        if x.left ≠ NIL
            x.left.parent = x
        x.right ≠ NIL
            x.right.parent = x
        return x
    else return NIL
```

The runtime is $T(n) = 2T(n/2) + c$ which is has solution $O(n)$ by Master method: $k = \log_2 2 = 1$, and $f(n) = c$.

The resulting tree is actually an AVL tree. It is not necessarily complete. However because the median index may involve a "round-down", the height of the left and right subtree differs by at most one. We know that the height of AVL trees is $O(\log n)$.

Problem 4

A recursive algorithm can be used to set $x.max$ for each node in the interval tree. The algorithm below takes as input a node T, which is the root of an interval tree, for with the values of $x.max$ have not yet been set. Again, this algorithm returns the value of max so that it can be used in earlier recursive calls. The runtime recurrence for this algorithm is $T(n) = T(|L|) + T(|R|) + c$, which we have shown using Inorder travels, has a solution of $O(n)$.

```
Set-Max(T)
    If T = NIL
        return -INF
    T.max = max( Set-Max(T.left), Set-Max(T.right), x.int.high)
    return T.max
```

The interval $i = [18, 25]$ overlaps with $[18, 22]$ from the right subtree. But the search will return interval $[9, 20]$ from the left subtree.

Problem 5

If we would like to return the interval with the minimum possible left endpoint, then we can update the search algorithm so that it continues the search even after an overlapping interval is found, keeping track of the current interval with the minimum left endpoint. The search carries out a while loop that traverses down a path of the tree, performing a constant amount of work at each node. Therefore the runtime is $O(h)$.

```
MinL-IntervalSearch(x,i)
    Initialize min = NIL
    while x ≠ NIL
       if x overlaps with i
          min = x
       if x.left ≠ NIL and x.left.max ≥ i.low
          x = x.left
       else x = x.right
    return min
```

Problem 6

- The following recursive algorithm takes as input a parameter T as the root of the tree, and the required rank k. The algorithm returns the node with rank k in the tree. Note the algorithm below does not check if k is a valid rank. The algorithm searches down one path of the tree, and therefore has a runtime of $O(h)$.

```
Recursive-BST-Select(k , T)
   L=1
   if (T.left ≠ NIL)
      L += T.left.size
    if k = L return T
    else if k < L
      return Recursive-BST-Select(T.left,k)
   else return Recursive-BST-Select(T.right,k-L)
```

- A recursive algorithm to return the **Rank** of a node x. Assume T is the root node of the tree and node x is the node for which we are trying to determine the rank. The algorithm below returns the rank of x, assuming x exists in the tree. It searches down a path in the tree, and has a runtime of $O(h)$.

```
RecursiveRank(T,x)
    if T = x
        if T.left != NIL
            return (T.left.size + 1)
        else return 1
    if T.key < x.key
        L=1
        if T.left != NIL
            L += T.left.size
        return RecursiveRank(T.right,x) + L
    else
        return RecursiveRank(T.left,x)
```

Problem 7

This algorithm is similar to the $Range(a, b)$ algorithm that we saw in the lecture notes. Just like in the notes, we carry out a search for a, and stop when we first the first key that is larger than a. We then proceed to search to the bottom of the tree and output all the keys larger than a.

```
LargerThan(T,a)
    x = T
    while x ≠ NIL
        if x.key < a.key
            x = x.right
        else
            print(x.key)
            Inorder(x.right)
            x = x.left
```

The above algorithm searches down at most one path in the tree, which runs in time $O(h)$. It also calls Inorder on all keys that are larger than a. The runtime of Inorder is $O(k)$ where k is the number of keys that are larger than a. The runtime of the entire algorithm is therefore $O(h + k)$.

Problem 8

A recursive algorithm can evaluate the subtree sizes. In the version below, we also return the value of the subtree size, so that it can be used in the earlier recursive calls. The algorithm calls itself on both the left and right children of the current node. Once each of those subtree sizes is determined, the subtree size of the current node is the sum of the size of the left and right tree, plus 1 for itself.

```
SUBTREE-SIZE(T)
    If T = NIL return 0
    T.size =SUBTREE-SIZE(T.left) + SUBTREE-SIZE(T.right) + 1
    return T.size
```

This recursive algorithm makes a recursive call to the left and right subtree, and performs a constant number of operations. Therefore the recurrence for the runtime is: $T(n) = T(|L_n|) + T(|R_n|) + c$, which is the same as that for Inorder traversal. Therefore we can conclude that the algorithm runs in time $O(n)$.

Problem 9

Step 1: Carry out an inorder traversal of the tree to produce a sorted list of the keys. This takes time $O(n)$.

Step 2: Now loop through the sorted list and compute the *gap* between every adjacent pair. Keep track of the current *minimum* gap, and update it whenever you find a new gap that is smaller. After one loop through the sorted list, you have found your minimum gap. This takes time $O(n)$, thus your algorithm runs in time $O(n)$.

Problem 10:

The Data structure: An interval tree T that is implemented as a red-black tree, where each node is additionally augmented with subtree-size. We can build this data structure by performing a series of n inserts into an interval tree implemented as a red-black tree and augmented with subtree sizes. Each insert takes time $O(\log n)$, and therefore the total construction runs in time $O(n \log n)$.

The Operations:

1. Call Interval-Search(T, i), which runs in time $O(\log n)$ for red-black trees. If the result is FALSE, carry out an insert into the interval tree. This insert runs in time $O(h) = O(\log n)$ for red-black trees.
2. Given the interval x, we call k =Rank(x) using the fact that the tree is also augmented with subtree sizes. The result k is the rank of the left-endpoint of x, and therefore we can return $k - 1$ which is equivalent to the number of intervals that start before interval x. The runtime of Rank(x) is $O(h) = O(\log n)$ for red-black trees.
3. We saw in our practice problems an algorithm called Min-IntervalSearch(T,i) which returns the interval in the tree T that has the minimum left endpoint that overlaps with i. This algorithm runs in time $O(h) = O(\log n)$ for red-black trees.
4. Let $r = Rank(x)$. The project that starts next will be that which has rank $r + 1$. Therefore we let y =BST-Select($r + 1$) and return y. Both of these algorithms run in time $O(h) = O(\log n)$.

Problem 11

- The algorithm below returns the height of a Binary tree. A binary tree with one node has height 0.

```
Find-height(T)
    if T = NIL
        return -1
    else return max( Find-height(T.left), Find-height(T.right) ) + 1
```

- Given a complete binary search tree, all levels are full except perhaps the last. Therefore we could color all levels black except the nodes in the last level, which should be colored red. We need to add NIL nodes to all the leaves. If h represents the height of the node T, then we only want to color nodes red when $h = 0$. The pseudo-code is shown below:

```
ColorTree(T,h)
    if T.left = NIL
        T.left = new NIL black node
    else ColorTree(x.left, h-1)          #Recursive call to the left subtree.
    if T.right = NIL
        T.right = new NIL black node
    else ColorTree(x.right, h-1)         #Recursive call to the left subtree
    if (h = 0)                           #When h is 0, we are are the last level of the tree
        x.color = red
    else x.color = black
```

Problem 12

The algorithm below first checks if the node is a nil node. If it is, it's black height is set to 0, since nil nodes in a red-black tree are actual node objects, colored black, with a black height of 0. The algorithm makes a recursive call to the left and right subtrees, which recursively assigns the black height to all nodes in the left and right subtrees. Once these values have been set, the black height of node T can be set as follows: take either child and check if it's black, and if it is, set the black height of T to one more than the black height of the child. Otherwise, set the black height of T to that of the child. Note that we *don't* have to verify that the children have valid black heights, because we are told this is a valid red-black tree.

```
AssignBH(T)
  if T.isNIL = true
     T.bh = 0
     return 0
   else
     a = AssignBH(T.left)
     b = AssignBH(T.right)
     if T.left.color = black
        T.bh = a+1
     else
        T.bh = a
    return T.bh
```

The above algorithm makes a recursive call to the left and right subtrees, and other than that, the remaining operations all run in constant time. Therefore the recurrence for the runtime is $T(n) = T(L_n) + T(R_n) + c$, where L_n and R_n are the sizes of the left and right subtrees. Note that this is the same recurrence as for Inorder Traversal, and therefore has a runtime of $O(n)$.

Problem 13

The original algorithm is shown below. Any additional lines of pseudo-code are colored in purple.

```
TREE-INSERT(T,z)
        z.size = 1
        if T = NIL return z
        else x = T
        While x ≠ NIL
           x.size = x.size + 1
           y = x
           if z.key < x.key
              x = x.left
           else x = x.right
        z.parent = y
        if z.key < y.key
           y.left = z
       else y.right = z
        return T
```

The original algorithm runs in time $O(h)$. In this updated version, we perform an additional constant amount of work for each iteration of the while loop. Therefore the result is still an $O(h)$ algorithm.