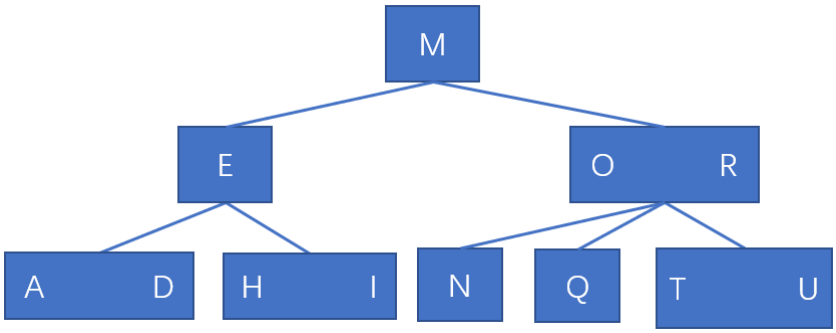
Answer to Problem Set 8

- Student Name: Gaole Dai
- Student ID: S01435587

Problem 1

Assume default behavior for duplicates is ignoring the duplicate.

Result:

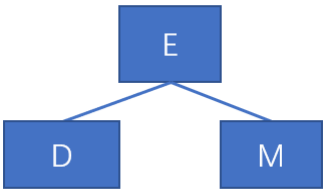


Procedure:

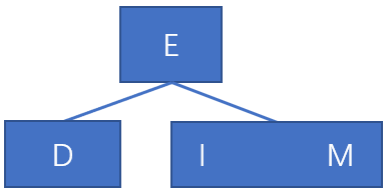
1. Insert `M`

2. Insert `E`

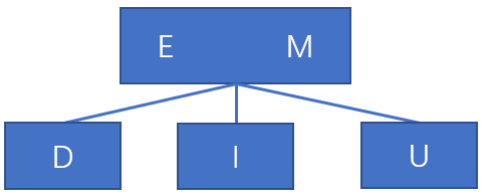
3. Insert `D`



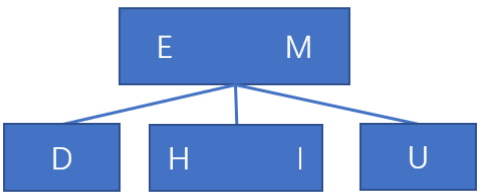
4. Insert `I`



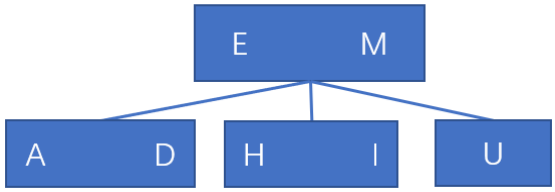
5. Insert `U`



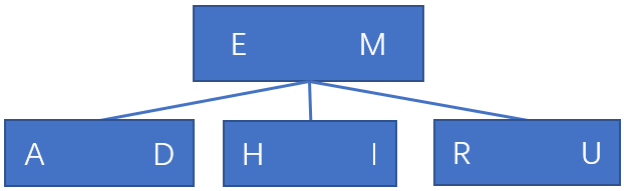
6. Insert `H`



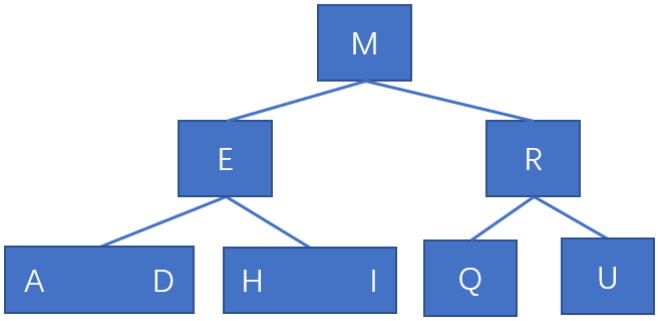
7. Insert `A`



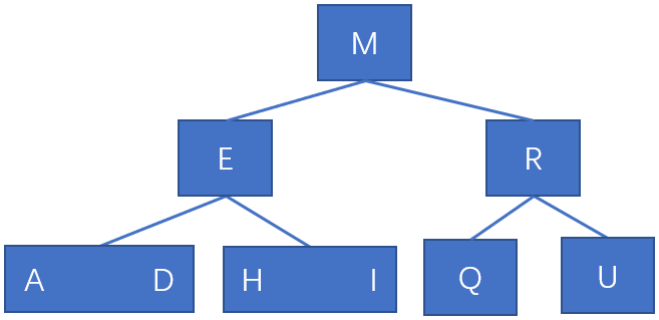
8. Insert `R`



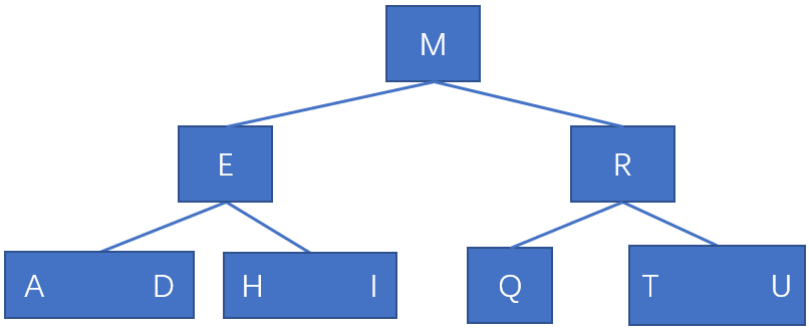
9. Insert `Q`



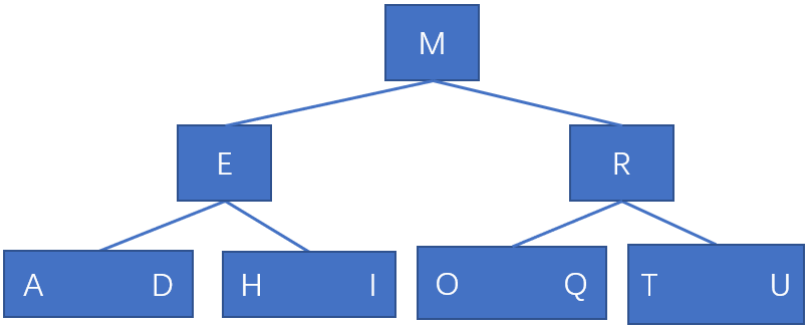
10. Insert `U` - duplicate



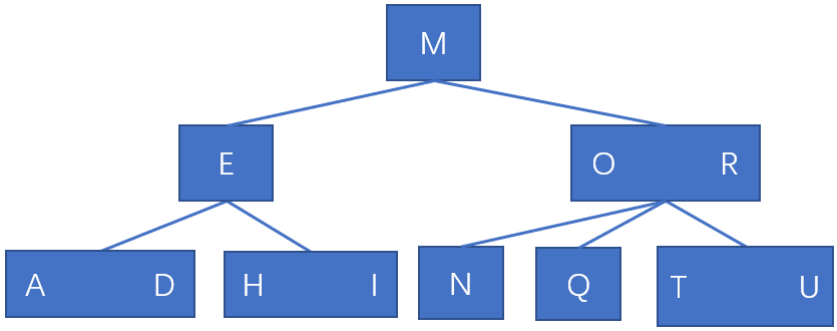
11. Insert `T`



12. Insert `O`

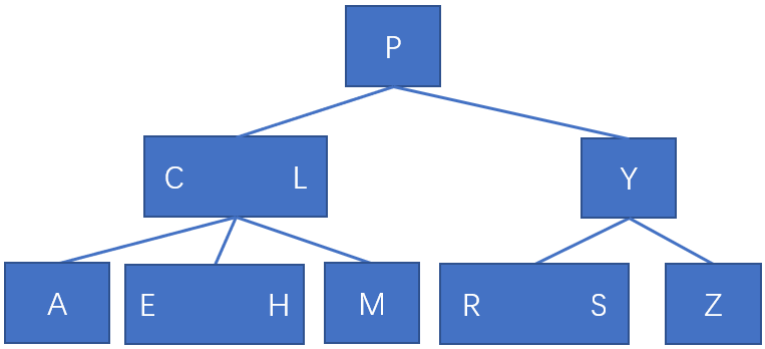


13. Insert `N`



Problem 2

Result:

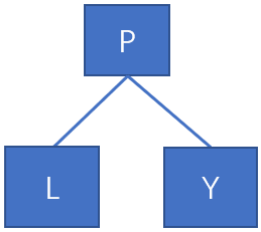


Procedure:

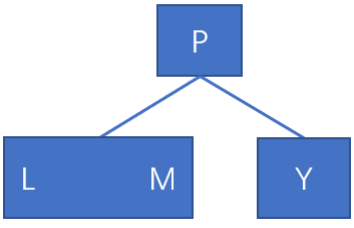
1. Insert `Y`

2. Insert `L`

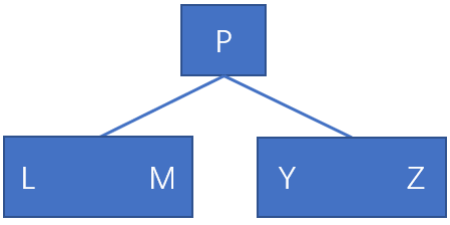
3. Insert `P`



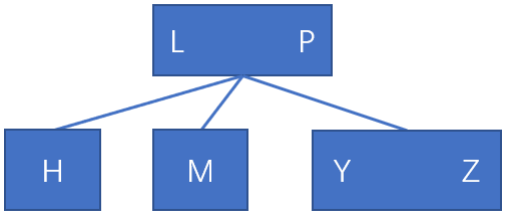
4. Insert `M`



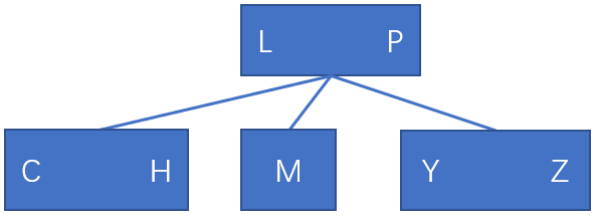
5. Insert `Z`



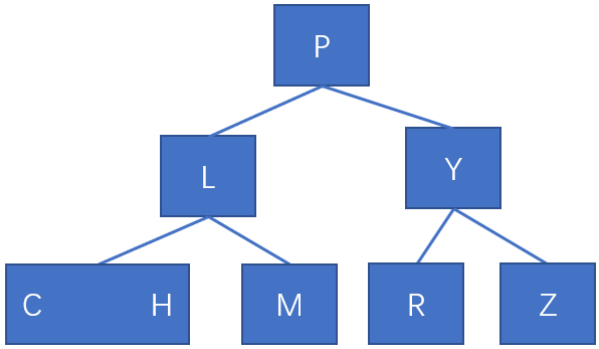
6. Insert `H`



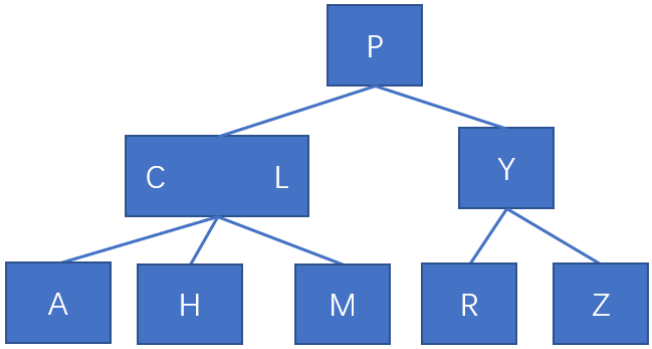
7. Insert `C`



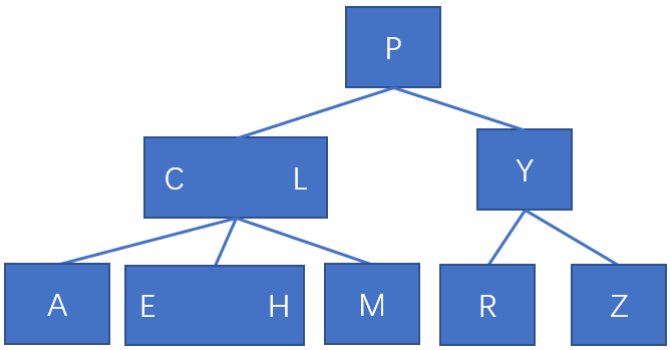
8. Insert `R`



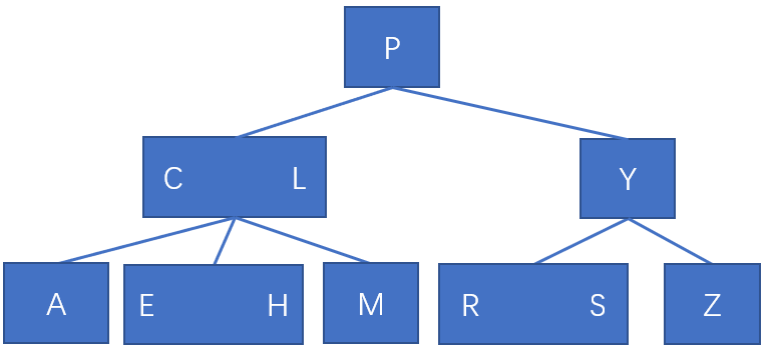
9. Insert `A`



10. Insert `E`



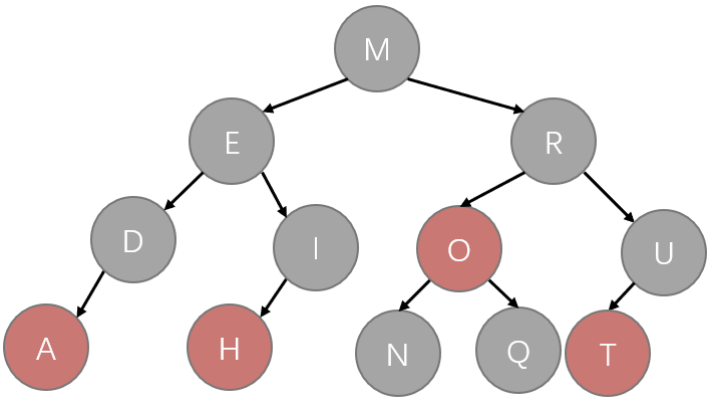
11. Insert `S`



Problem 3

Assume default behavior for duplicates is ignoring the duplicate.

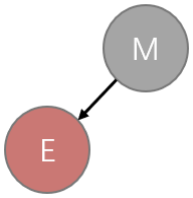
Result:



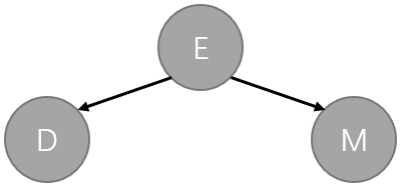
Procedure:

1. Insert `M`

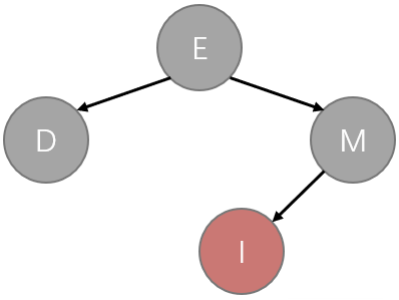
2. Insert `E`



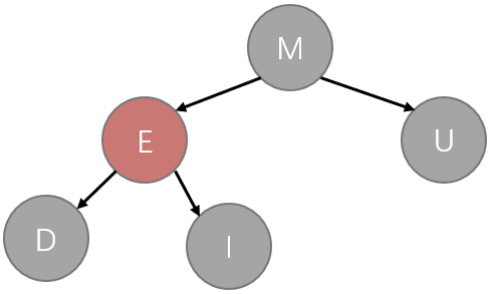
3. Insert `D`



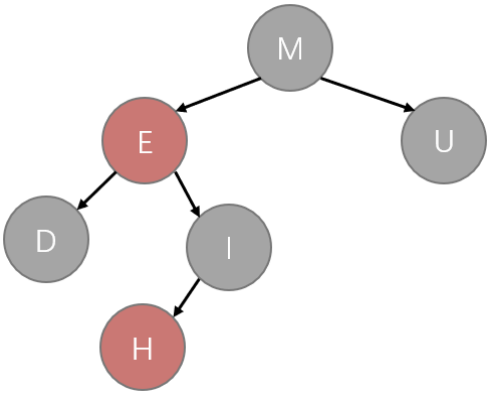
4. Insert `I`



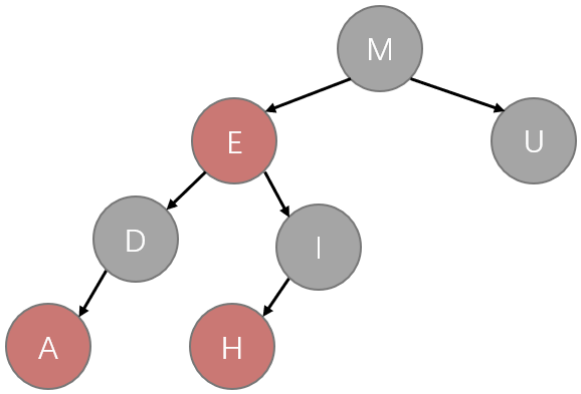
5. Insert `U`



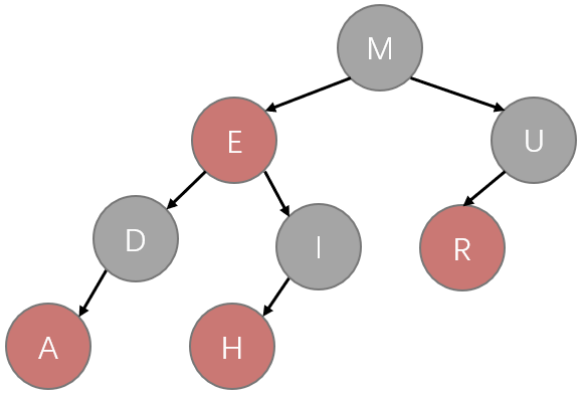
6. Insert `H`



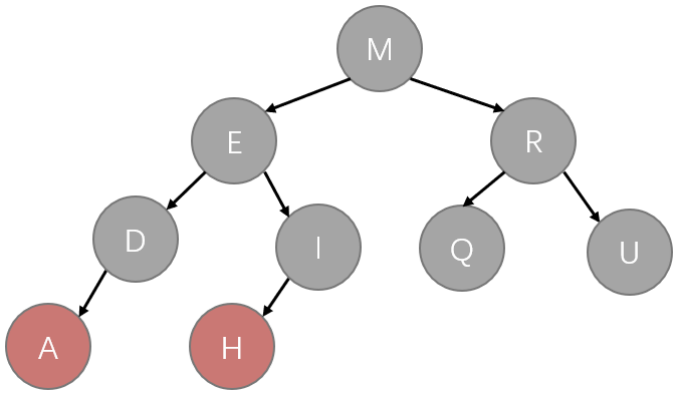
7. Insert `A`



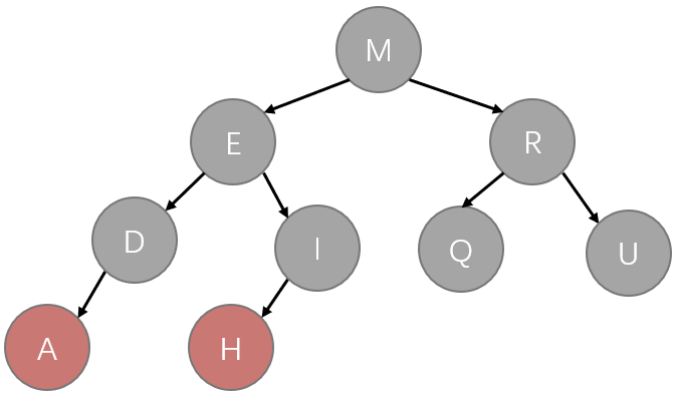
8. Insert `R`



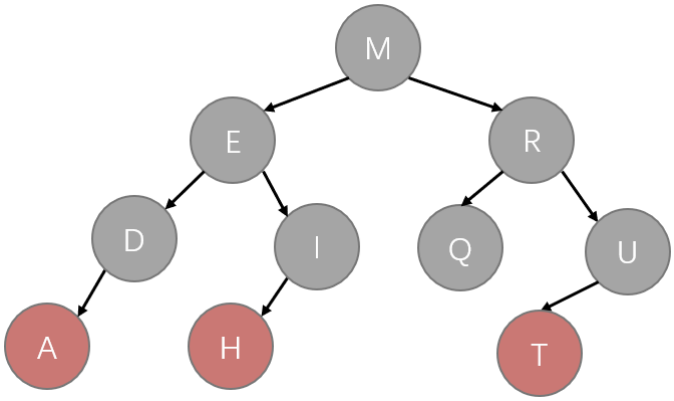
9. Insert `Q`



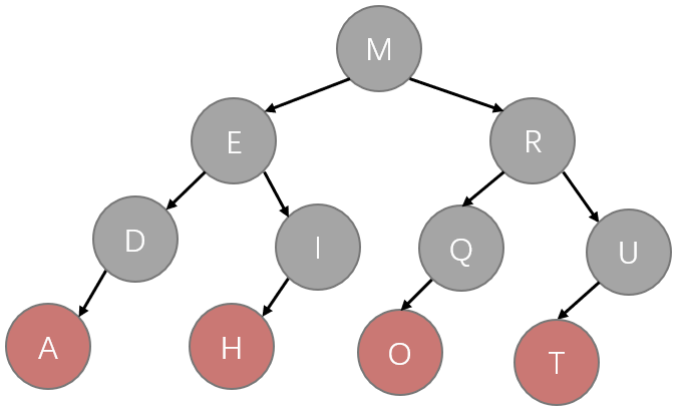
10. Insert `U` - duplicate



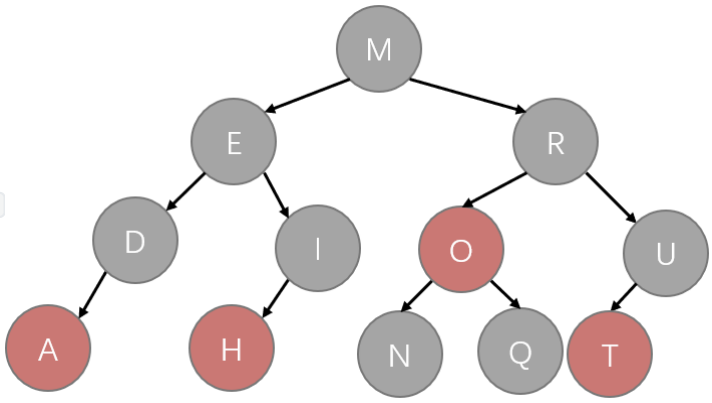
11. Insert `T`



12. Insert `O`

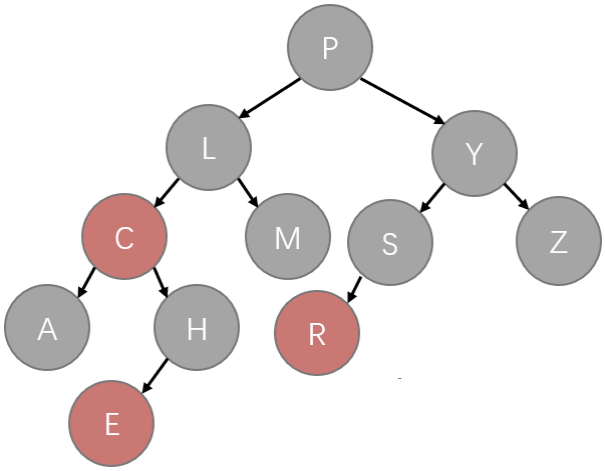


13. Insert `N`



Problem 4

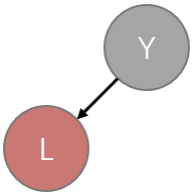
Result:



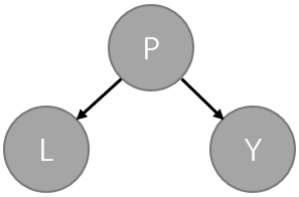
Procedure:

1. Insert `Y`

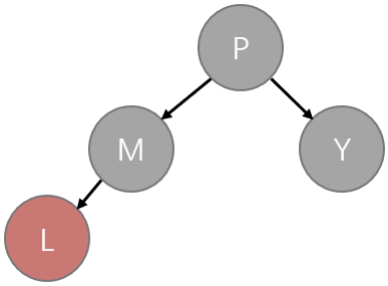
2. Insert `L`



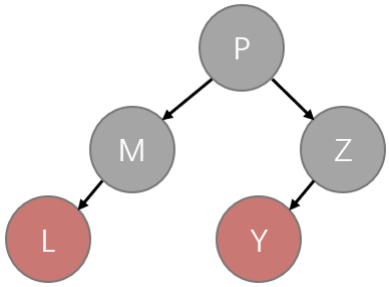
3. Insert `P`



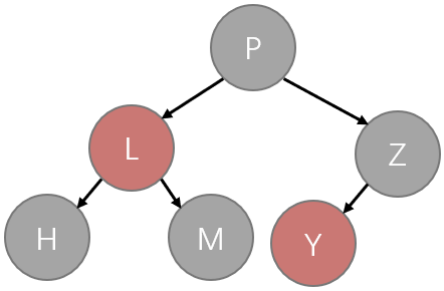
4. Insert `M`



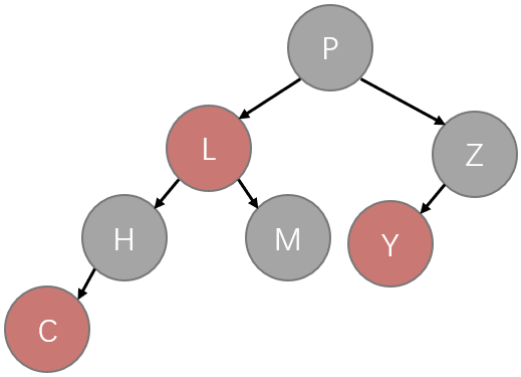
5. Insert `Z`



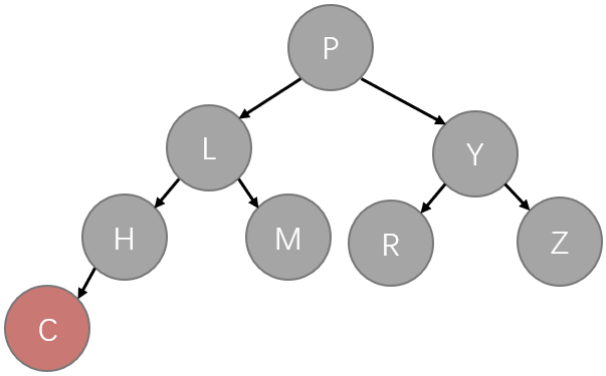
6. Insert `H`



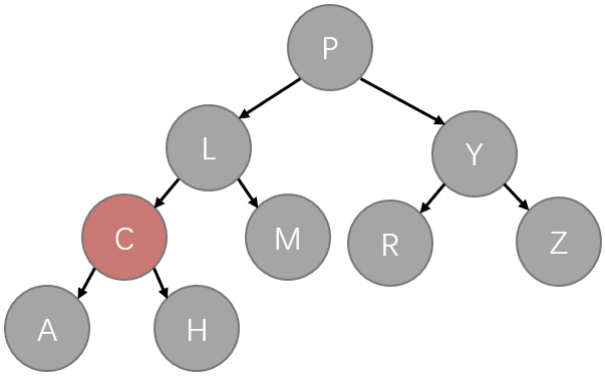
7. Insert `C`



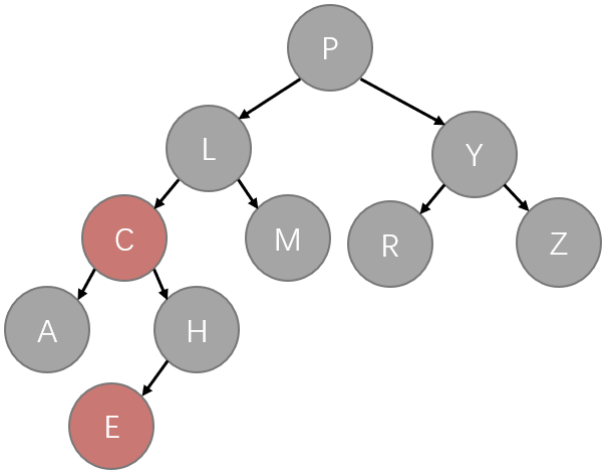
8. Insert `R`



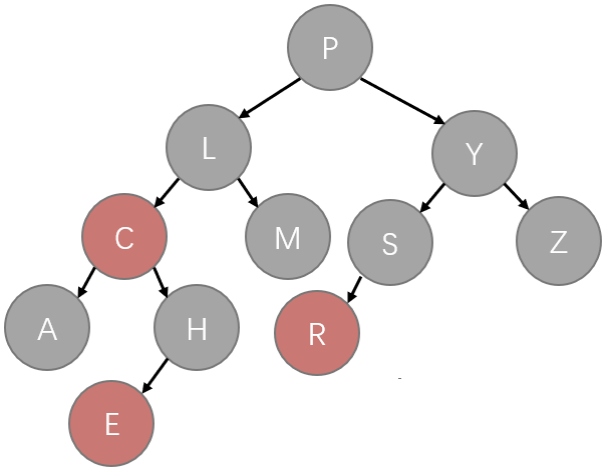
9. Insert `A`



10. Insert `E`



11. Insert `S`



Problem 5

Code:

```
# Reference: Deletion code implementation based on
https://www.programiz.com/dsa/deletion-from-a-red-black-tree


import sys

# Create node for red-black tree
class Node():
    def __init__(self, item):
        # item represents the value of the node
        self.item = item
        # Parent node
        self.parent = None
        # Left child of the node
        self.left = None
```

```
        # Right child of the node
        self.right = None
        # Red color as 1 since the new insert node is always RED.
        self.color = 1

# Define the red-black tree class
class RedBlackTree():
    def __init__(self):
        self.TNULL = Node(0)
        self.TNULL.color = 0
        self.TNULL.left = None
        self.TNULL.right = None
        self.root = self.TNULL

    # Balancing the tree after deletion
    def delete_fix(self, x):
        while x != self.root and x.color == 0:
            if x == x.parent.left:
                s = x.parent.right
                if s.color == 1:
                    s.color = 0
                    x.parent.color = 1
                    self.left_rotate(x.parent)
                    s = x.parent.right

                if s.left.color == 0 and s.right.color == 0:
                    s.color = 1
                    x = x.parent
                else:
                    if s.right.color == 0:
                        s.left.color = 0
                        s.color = 1
                        self.right_rotate(s)
                        s = x.parent.right

                    s.color = x.parent.color
                    x.parent.color = 0
                    s.right.color = 0
                    self.left_rotate(x.parent)
                    x = self.root
            else:
                s = x.parent.left
                if s.color == 1:
                    s.color = 0
                    x.parent.color = 1
                    self.right_rotate(x.parent)
                    s = x.parent.left

                if s.right.color == 0 and s.left.color == 0:
                    s.color = 1
                    x = x.parent
                else:
                    if s.left.color == 0:
                        s.right.color = 0
                        s.color = 1
```

```
                    self.left_rotate(s)
                    s = x.parent.left

                s.color = x.parent.color
                x.parent.color = 0
                s.left.color = 0
                self.right_rotate(x.parent)
                x = self.root
        x.color = 0

    def __rb_transplant(self, u, v):
        if u.parent == None:
            self.root = v
        elif u == u.parent.left:
            u.parent.left = v
        else:
            u.parent.right = v
        v.parent = u.parent

    # Node deletion
    def delete_node_helper(self, node, key):
        z = self.TNULL
        while node != self.TNULL:
            if node.item == key:
                z = node

            if node.item <= key:
                node = node.right
            else:
                node = node.left

        if z == self.TNULL:
            print("Cannot find key in the tree")
            return

        y = z
        y_original_color = y.color
        if z.left == self.TNULL:
            x = z.right
            self.__rb_transplant(z, z.right)
        elif (z.right == self.TNULL):
            x = z.left
            self.__rb_transplant(z, z.left)
        else:
            y = self.minimum(z.right)
            y_original_color = y.color
            x = y.right
            if y.parent == z:
                x.parent = y
            else:
                self.__rb_transplant(y, y.right)
                y.right = z.right
                y.right.parent = y

            self.__rb_transplant(z, y)
```

```
            y.left = z.left
            y.left.parent = y
            y.color = z.color
        if y_original_color == 0:
            self.delete_fix(x)

    # Balance the tree after insertion
    def fix_insert(self, k):
        while k.parent.color == 1:
            if k.parent == k.parent.parent.right:
                u = k.parent.parent.left
                if u.color == 1:
                    u.color = 0
                    k.parent.color = 0
                    k.parent.parent.color = 1
                    k = k.parent.parent
                else:
                    if k == k.parent.left:
                        k = k.parent
                        self.right_rotate(k)
                    k.parent.color = 0
                    k.parent.parent.color = 1
                    self.left_rotate(k.parent.parent)
            else:
                u = k.parent.parent.right

                if u.color == 1:
                    u.color = 0
                    k.parent.color = 0
                    k.parent.parent.color = 1
                    k = k.parent.parent
                else:
                    if k == k.parent.right:
                        k = k.parent
                        self.left_rotate(k)
                    k.parent.color = 0
                    k.parent.parent.color = 1
                    self.right_rotate(k.parent.parent)
            if k == self.root:
                break
        self.root.color = 0

    def successor(self, x):
        if x.right != self.TNULL:
            return self.minimum(x.right)

        y = x.parent
        while y != self.TNULL and x == y.right:
            x = y
            y = y.parent
        return y

    def predecessor(self,  x):
        if (x.left != self.TNULL):
            return self.maximum(x.left)
```

```
        y = x.parent
        while y != self.TNULL and x == y.left:
            x = y
            y = y.parent

        return y

    def left_rotate(self, x):
        y = x.right
        x.right = y.left
        if y.left != self.TNULL:
            y.left.parent = x

        y.parent = x.parent
        if x.parent == None:
            self.root = y
        elif x == x.parent.left:
            x.parent.left = y
        else:
            x.parent.right = y
        y.left = x
        x.parent = y

    def right_rotate(self, x):
        y = x.left
        x.left = y.right
        if y.right != self.TNULL:
            y.right.parent = x

        y.parent = x.parent
        if x.parent == None:
            self.root = y
        elif x == x.parent.right:
            x.parent.right = y
        else:
            x.parent.left = y
        y.right = x
        x.parent = y

    def insert(self, key):
        node = Node(key)
        node.parent = None
        node.item = key
        node.left = self.TNULL
        node.right = self.TNULL
        node.color = 1

        y = None
        x = self.root

        while x != self.TNULL:
            y = x
            if node.item < x.item:
                x = x.left
```

```
            else:
                x = x.right

        node.parent = y
        if y == None:
            self.root = node
        elif node.item < y.item:
            y.left = node
        else:
            y.right = node

        if node.parent == None:
            node.color = 0
            return

        if node.parent.parent == None:
            return

        self.fix_insert(node)

    def get_root(self):
        return self.root

    def delete_node(self, item):
        self.delete_node_helper(self.root, item)

    def print_tree(self):
        self.__print_helper(self.root, "", True)
```

Justification for the deletion:

The whole procedure is shown as

1. Save the color of the target delete node
2. If the left child of the deleted node is `NULL`, assign the right child to be `x`, then transplant the deleted node with `x`
3. Else if the right child of the deleted node is `NULL`, assign the left child to be `x`, then transplant the deleted node with `x`
4. Else, assign the minimum of the right sub-tree of the deleted node into `y`; save the color of `y`, and assign the right child of `y` into `x`, if `y` is a child of the deleted node, then set the parent of `x` as `y`; else transplant `y` with the right child of `y`, transplant the deleted node with `y`. set the color of `y` with the previous stored color.
5. If the previous stored value is black, call `delete_fix` function with input `x`.

Thus, the deletion of the red-black tree is showed and justified.

Problem 6

Code:

Since the left child is smaller than the root and the right child is larger than the root, the idea of finding the minimum element is to recursively iterate the red-black tree and find the last element in the left sub-tree.

Inherit the Red-black construction class in Problem 5.

```
def minValue(node):
    if node.left == None:
        return node.item
    return minValue(node.left)
```

Complexity:

The function would run the same time as the height of the red-black tree, if the total number of nodes in the red-black tree is n, the height of the tree is in the order of $\mathcal{O}(\log_2(n))$. Thus, the worst time complexity for finding the minimum is $\mathcal{O}(\log(n))$.

Correctness:

- Base case: the left child is None, which indicates the current node is the left most node, return the current node value, the minimum value, proved for base case.
- Inductive step:
    - Suppose min value is returned correctly for node with length n-1,
    - Then for node with length n
        - The one more node could be in the middle of the tree, the min value of the tree is the same, since the correctness is proved for tree with length n-1, this is also true;
        - The one more node could be the minimum of the tree, then it is at the left most position, the algorithm would iterate one more time, and the min value is returned.
        - Test with data:
        - ```
          bst = RedBlackTree()
          bst.insert(50)
          bst.insert(30)
          bst.insert(20)
          bst.insert(40)
          bst.insert(70)
          bst.insert(60)
          bst.insert(80)
          print("Min value of Red-Black tree is ",minValue(bst))
          -----Min of Red-Black Tree is 20.- correct
          ```

Problem 7

Code:

Inherit the Red-black construction class in Problem 5.

```
# Reference: Code implementation based on https://www.geeksforgeeks.org/find-median-bst-time-o1-space/

""" Function to count the number of nodes in
    a red-black tree using
     Morris Inorder traversal"""
def counNodes(root):

    # Initialise count of nodes as 0
```
```

```
    count = 0

    if (root == None):
        return count

    current = root
    while (current != None):

        if (current.left == None):

            # Count node if its left is None
            count+=1

            # Move to its right
            current = current.right

        else:
            """ Find the inorder predecessor of current """
            pre = current.left

            while (pre.right != None and
                    pre.right != current):
                pre = pre.right

            """ Make current as right child of its
            inorder predecessor """
            if(pre.right == None):

                pre.right = current
                current = current.left
            else:

                pre.right = None

                # Increment count if the current
                # node is to be visited
                count += 1
                current = current.right

    return count

def findMedian(root):
    if (root == None):
        return 0
    count = counNodes(root)
    currCount = 0
    current = root

    while (current != None):

        if (current.left == None):

            # count current node
            currCount += 1
```

```
            # check if current node is the median
            # Odd case
            if (count % 2 != 0 and
                currCount == (count + 1)//2):
                return prev.item

            # Even case
            elif (count % 2 == 0 and
                    currCount == (count//2)+1):
                return (prev.item + current.item)//2

            # Update previous node for even numgber of nodes
            prev = current

            # Move to the right
            current = current.right

        else:

            """ Find the inorder predecessor of current """
            pre = current.left
            while (pre.right != None and
                    pre.right != current):
                pre = pre.right

            """ Make current as right child
                of its inorder predecessor """
            if (pre.right == None):

                pre.right = current
                current = current.left
            else:

                pre.right = None

                prev = pre

                # Count current node
                currCount += 1

                # Check if the current node is the median
                if (count % 2 != 0 and
                    currCount == (count + 1) // 2 ):
                    return current.item

                elif (count % 2 == 0 and
                    currCount == (count // 2) + 1):
                    return (prev.item+current.item)//2

                # update prev node for the case of even
                # number of nodes
                prev = current
                current = current.right
```

Complexity:

$\mathcal{O}(n)$.

The Morris in-order traverse function `counNodes()` for counting the number of nodes takes $\mathcal{O}(n)$, since it iterate the whole tree. For the finding median function `findMedian` call the `countNodes()` function and also iterate the tree, thus the time complexity is $\mathcal{O}(n) + \mathcal{O}(n) = \mathcal{O}(n)$.

Correctness:

- If the number of nodes is even, then the median is $(n/2\text{th node} + (n/2 + 1\text{th node}))/2$.
- If the number of nodes is odd, then the median is $(n + 1)/2\text{th node}$.
- For the algorithm, it first call the counting number of nodes, the `while` loop iterates until the current node is `None`, the number of nodes increment if the current node is visited, and the current node is then moved to the right or left sub-tree. The correctness is proved.
- Then, using the same in-order traverse method, the median value is searched and compared, the median point value is returned. The correctness is proved.
- Test with data:
- ```
  bst = RedBlackTree()
  bst.insert(50)
  bst.insert(30)
  bst.insert(20)
  bst.insert(40)
  bst.insert(70)
  bst.insert(60)
  bst.insert(80)
  print("Median of Red-Black Tree is ",findMedian(bst))
  -----Median of Red-Black Tree is 50.- correct
  ```

Problem 8.1

Index	Value
0	14
1	20
2	
3	
4	16 -> 5
5	88 -> 11
6	94 -> 39
7	12 -> 34 -> 23
8	
9	
  ```

Index	Value
10	

$h(12) = (2 * 12 + 5) \bmod 11 = 7$

$h(14) = (2 * 14 + 5) \bmod 11 = 0$

$h(34) = (2 * 34 + 5) \bmod 11 = 7$ - collision - 12 -> 34

$h(88) = (2 * 88 + 5) \bmod 11 = 5$

$h(23) = (2 * 23 + 5) \bmod 11 = 7$ - collision - 12 -> 34 ->23

$h(94) = (2 * 94 + 5) \bmod 11 = 6$

$h(11) = (2 * 11 + 5) \bmod 11 = 5$ - collision - 88 -> 11

$h(39) = (2 * 39 + 5) \bmod 11 = 6$ - collision - 94 -> 39

$h(20) = (2 * 20 + 5) \bmod 11 = 1$

$h(16) = (2 * 16 + 5) \bmod 11 = 4$

$h(5) = (2 * 5 + 5) \bmod 11 = 4$ - collision - 16 -> 5

Problem 8.2

Index	Value
0	14
1	39
2	20
3	5
4	16
5	88
6	94
7	12
8	34
9	23
10	11

$h(12) = (2 * 12 + 5) \bmod 11 = 7$

$h(14) = (2 * 14 + 5) \bmod 11 = 0$

$h(34) = (2 * 34 + 5) \bmod 11 = 7$ - collision - 8

$h(88) = (2 * 88 + 5) \bmod 11 = 5$

$h(23) = (2 * 23 + 5) \bmod 11 = 7$ - collision - 9

$h(94) = (2 * 94 + 5) \bmod 11 = 6$

$h(11) = (2 * 11 + 5) \bmod 11 = 5$ - collision - 10

$h(39) = (2 * 39 + 5) \bmod 11 = 6$ - collision - 1

$h(20) = (2 * 20 + 5) \bmod 11 = 1$ - collision - 2

$h(16) = (2 * 16 + 5) \bmod 11 = 4$

$h(5) = (2 * 5 + 5) \bmod 11 = 4$ - collision - 3

Problem 9

Idea: Turn one of the set J into Hash map, then iterate another set H, lookup if (K-h) is in the hash map.

```
def findSum(H, J, K):
    hashmap = {}
    for item in J:
        if item in hashmap:
            hashmap[item] += 1
        else:
            hashmap[item] = 1
    for item in H:
        if K-item in hashmap:
            return "Yes"

    return "No"
```

Problem 10

a) $\mathcal{O}(n)$

b) When there is no 2 numbers that $h + j = K$.

For the first iteration, which turn the set into the hash, it iterates $|J|$ times, the complexity is $\mathcal{O}(n)$, then for the next iteration, it iterates the H set, return yes if (K-h) exists, the complexity is still $\mathcal{O}(n)$, where the worst case is there is no 2 numbers that $h + j = K$. The worst-case complexity is $\mathcal{O}(n) + \mathcal{O}(n) = \mathcal{O}(n)$.

Problem 11

Idea: Similar to problem 8 but not the same. Turn the array into the hash map and lookup if (K-item) is in the hash map at the same time.

```
def findSum(H, K):
    hashmap = {}
    for i in range(len(H)):
        complement = K - H[i]
        if complement in hashmap:
            print(H[i], complement)
            return "Yes"
        hashmap[H[i]] = H[i]
    return "No"
```

Since the for loop iterates $len(H)$ times, the complexity is $\mathcal{O}(n)$.