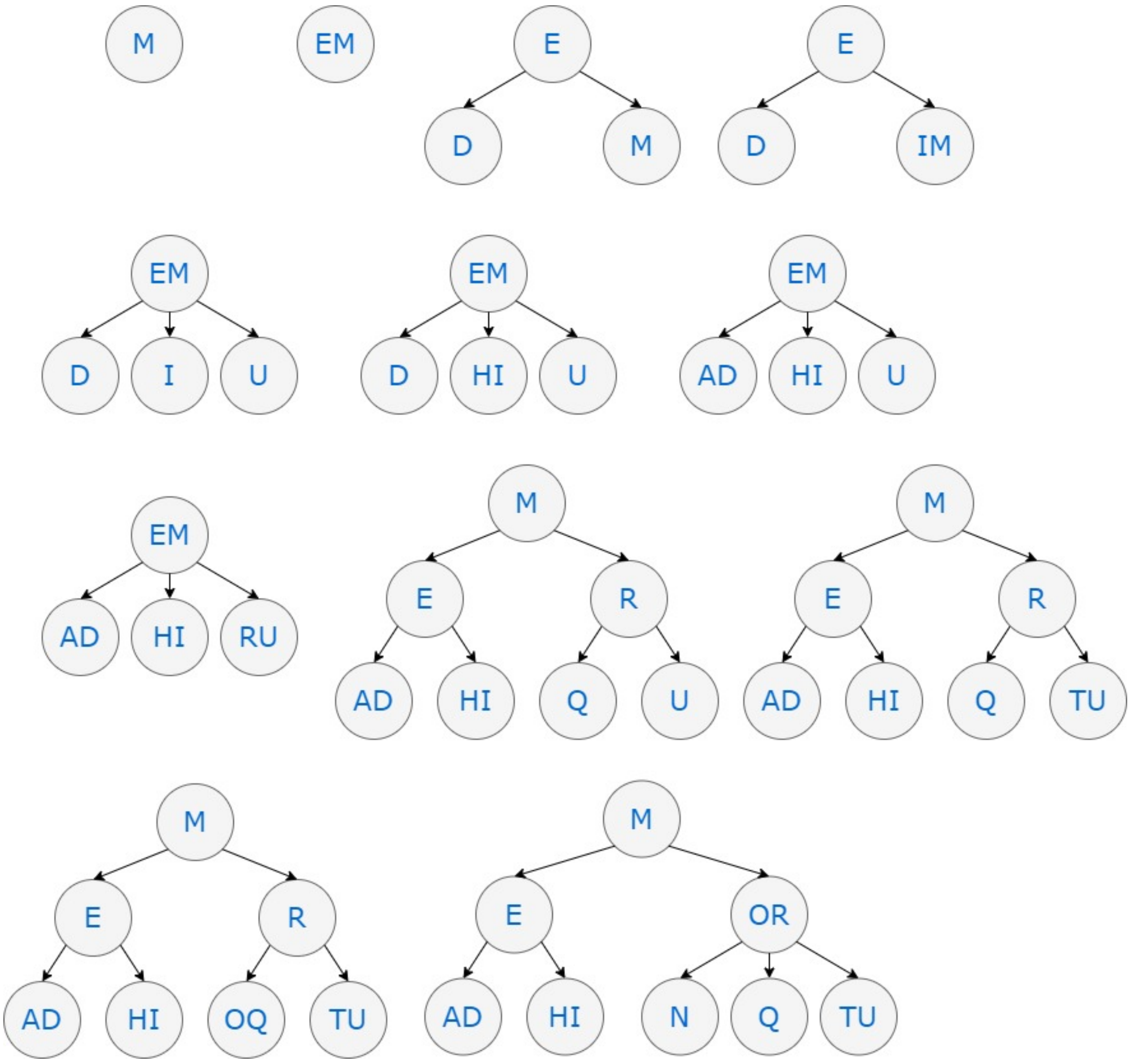
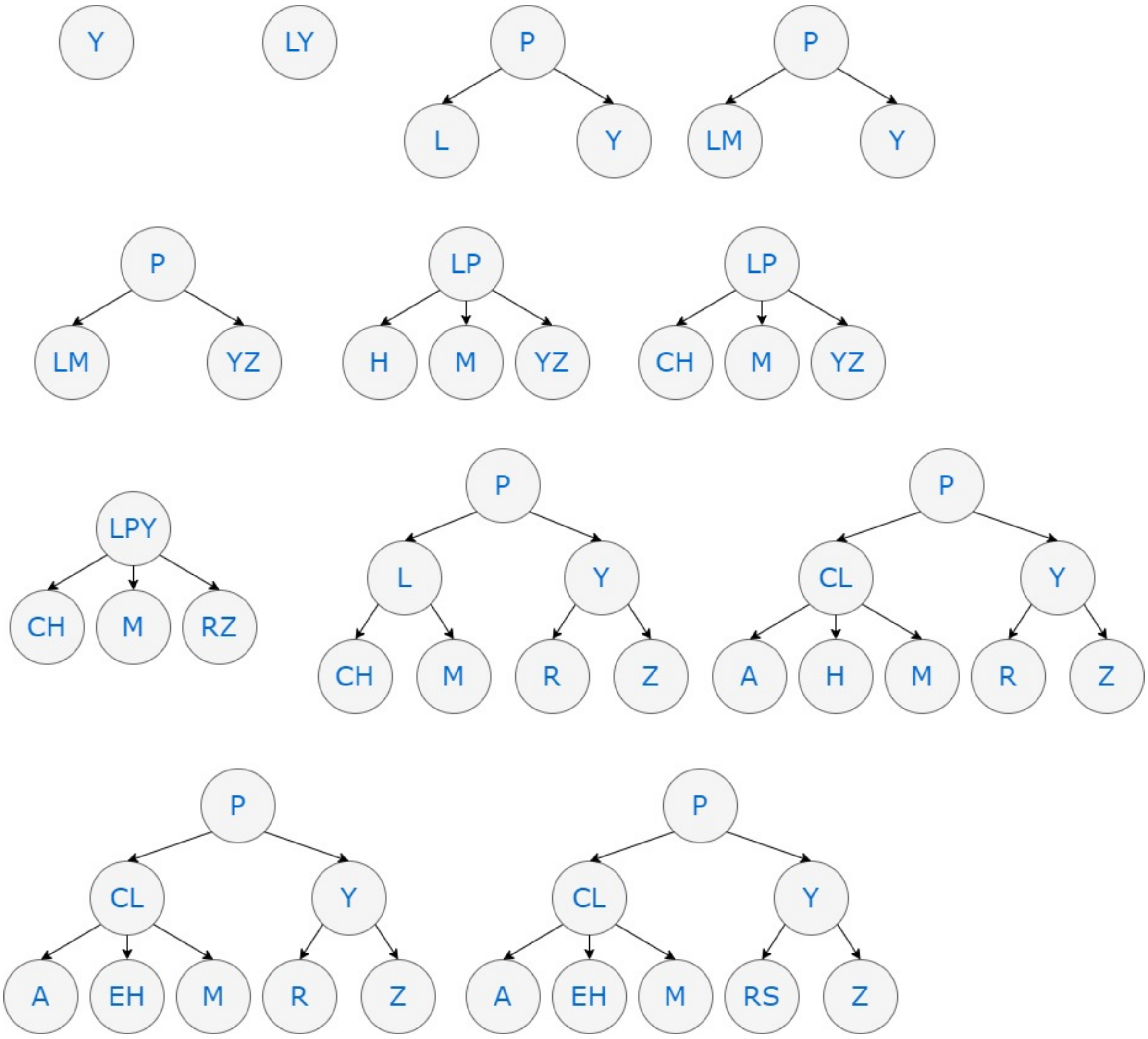
Module 8 Problem Set

Jiapeng Pei S01435611

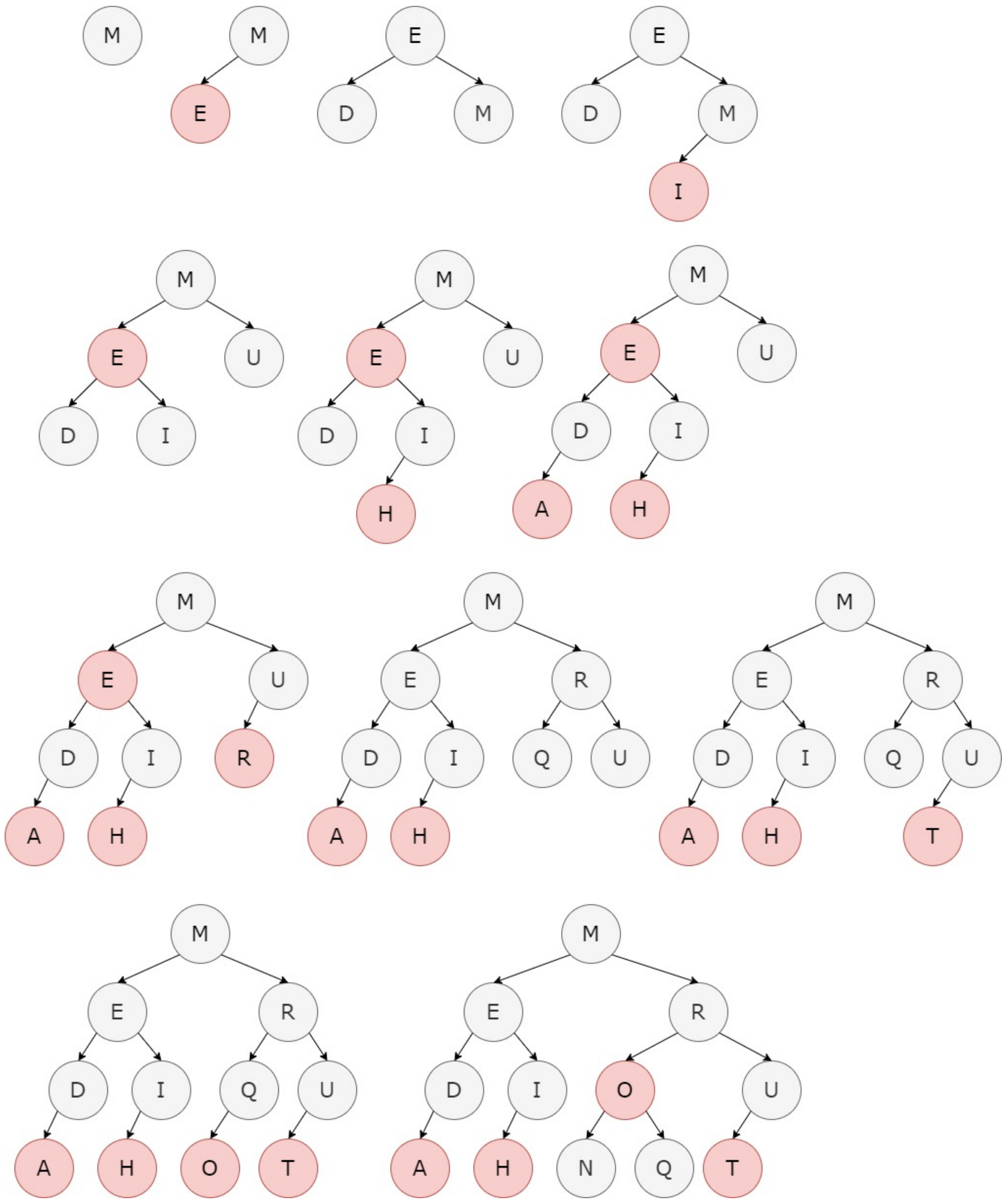
1.



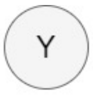
2.

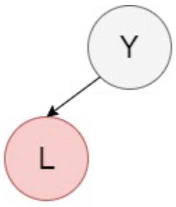

Y
LY
P L Y
P LM Y
P LM YZ
LP H M YZ
LP CH M YZ
LPY CH M RZ
P L Y CH M R Z
P CL Y A H M R Z
P CL Y A EH M R Z
P CL Y A EH M RS Z

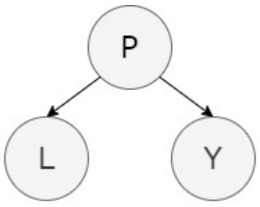
3.

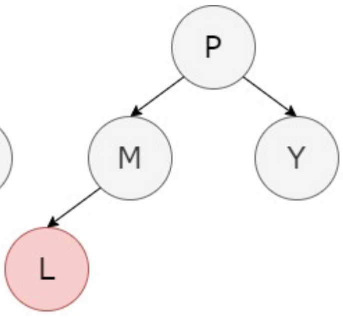

M
M E
E D M
E D M I
M E U D I
M E U D I H
M E U D I A H
M E U D I R A H
M E R D I Q U A H
M E R D I Q U A H T
M E R D I Q U A H O T
M E R D I O U A H N Q T

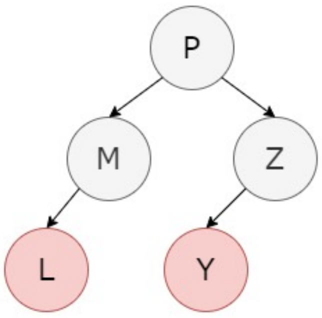
4.

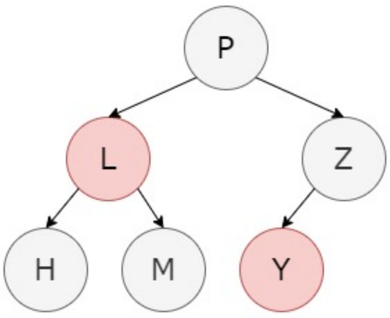

Y

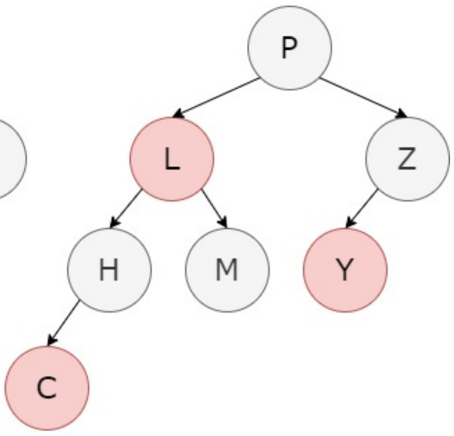

Y
L

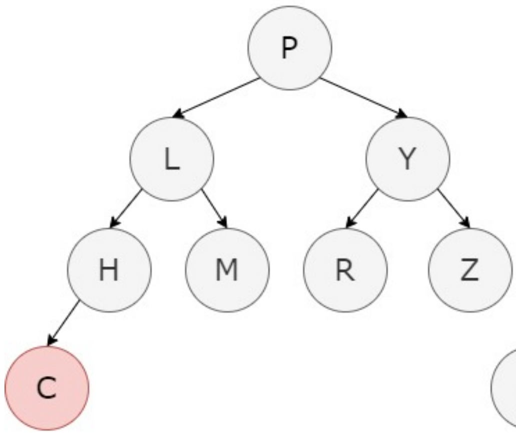

P
L
Y

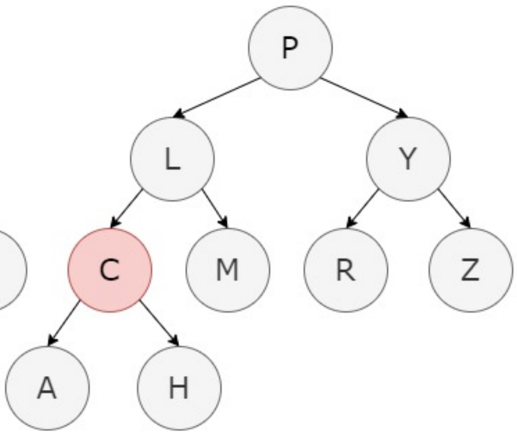

P
M
Y
L

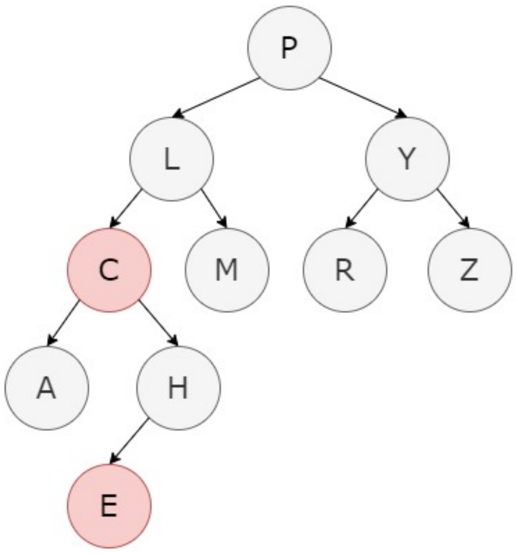

P
M
Z
L
Y

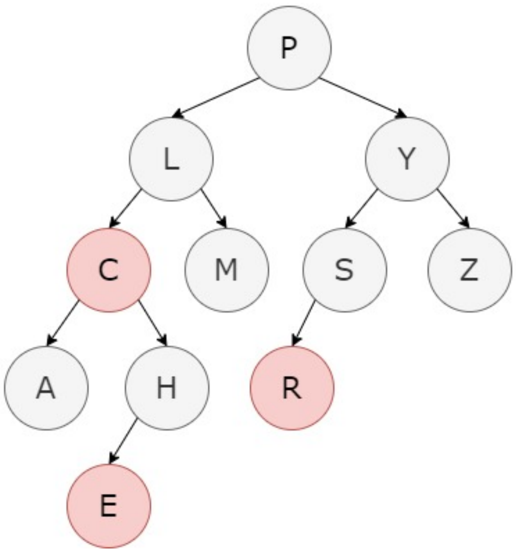

P
L
Z
H
M
Y


P
L
Z
H
M
Y
C


P
L
Y
H
M
R
Z
C


P
L
Y
C
M
R
Z
A
H


P
L
Y
C
M
R
Z
A
H
E


P
L
Y
C
M
S
Z
A
H
R
E

5.

The delete operation is based on the idea that we can't delete black nodes because that would destroy the black height. So we need to make sure that the last node we delete is red.

Let's first try to delete the smallest value in the whole tree.

How can we make sure that the last node deleted is red? We need to guarantee a property in the downward recursion: if the current node is h, then we need to guarantee that h is red, or $h.left$ is red.

Consider the correctness of this. If we can successfully maintain this property through various rotate and invert color operations, then when we reach the smallest node, h_{min}, there is either h_{min} that is red, or the left subtree of h_{min}- but h_{min} has no left subtree at all! So this guarantees that the min node must be red, and since it is red, we can boldly delete it and then adjust the tree using the same adjustment idea as the insertion operation.

Let's consider how to satisfy this property. Note that we will temporarily break several properties of the left-leaning red-black tree on the way down the recursion, but also restore it when we return from the recursion.

We first consider deleting the leaves: similar to deleting the minima, we have to maintain a property during the deletion of any value, but this time it is special because instead of going only to the left, we can go in both left and right directions, so the property maintained during the deletion is this: if going to the left, the current node is h, then we need to ensure that h is red or $h.left$ is red; if goes right and the current node is h, then you need to ensure that h is red, or $h.right$ is red. This ensures that we always end up deleting a red node.

Next, consider deleting a non-leaf node. We just need to find the smallest node in its right subtree (if any), then replace the value of that node with the value of the smallest node in the right subtree, and finally delete the smallest node in the right subtree.

```python
# rotateLeft(), rotateRight() and flipColors() are methods from lecture

def moveRedLeft(cur):
    flipColors(cur)
    if isRed(cur.right.left):
        cur.right = rotateRight(cur.right)
        cur = rotateLeft(cur)
        flipColors(cur)

    return cur

def moveRedRight(cur):
    flipColors(cur)
    if isRed(cur.left.left):
        cur = rotateRight(cur)
        flipColors(cur)

    return cur

def balance(cur):
    if isRed(cur.left) and isRed(cur.right):
        flipColors(cur)
    if isRed(cur.right) and not isRed(cur.left):
        cur = rotateLeft(cur)
    if isRed(cur.left) and isRed(cur.left.left):
        cur = rotateRight(cur)

def delete(key):
    root = delete(root, key)
    root.color = BLACK;

def delete(cur: Node, key):
    if key < cur.key:
        if isRed(cur.left) or isRed(cur.left.left):
            cur = moveRedLeft(cur)
        cur.left = delete(cur.left, key)
    else:
        if isRed(cur.left):
            cur = rotateRight(cur)
        if key == cur.key and cur.right is None:
            return None
        if isRed(cur->right) or isRed(cur.right.left):
            cur = moveRedRight(cur)
        if key == cur.key:
            nxt = min(cur.right)
            cur.key = nxt.key
            cur.val = nxt.val
            cur.right = deleteMin(cur.right)
        else:
            cur.right = delete(cur.right, key)

    return balance(cur)
```

6.

```
def findMin(root):
    while root.left != None:
        root = root.left
    return root.val
```

All the elements in the left subtree are smaller than root's value, and all the elements on the right subtree are bigger than root's value. So we should traverse as far left as possible to find the minimum value.

Suppose the size of tree is N. Since the height of red-black tree is at most $O(\log(N))$, the complexity to find the minimum is:

$$O(\log(N))$$

7. $O(N)$

Suppose the size of LLRB is N.

In my implementation, I traverse the LLRB in in-order to find the median value. The LLRB is sorted in in-order, so I can find the correct value. My algorithm has to visit at least half of the nodes, at most all nodes in LLRB and it takes $O(1)$ to visit each. So the time complexity is $O(N)$

```
median = None
order = 0

def findMedian(root, size):
    if size // 2 != 0:
        inOrder(root, (size + 1) // 2)
        return median
    else:
        inOrder(root, size // 2)
        medianLeft = median
        median = None
        inOrder(root, size // 2 + 1)
        return (medianLeft + median) / 2

def inOrder(root, rank):
    if root is None or median is not None:
        return

    inOrder(root.left, rank)
    order += 1
    if order == rank:
        median = root.val
    inorder(root.right, rank)
```

8. Following is the final result. Elements that appear earlier in the sequence also appear earlier(on the left) in the slots of hash table.

1. The final hash table with collision list:

   0 : [14]

   1 : [20]

   2 : []

   3 : []

   4 : [16, 5]

   5 : [88, 11]

   6 : [94, 39]

   7 : [12, 34, 23]

   8 : []

   9 : []

   10: []

2. Suppose x is the element to insert, S = 11. Here is how I apply linear probing.

   If slot h(x) mod S is full, then we try (hash(x) + 1) mod S
   If (h(x) + 1) mod S is also full, then we try (hash(x) + 2) mod S
   If (h(x) + 2) mod S is also full, then we try (hash(x) + 3) mod S

   ...

   Following hash table is the final result:

0 : [14]

1 : [39]

2 : [20]

3 : [5]

4 : [16]

5 : [88]

6 : [94]

7 : [12]

8 : [34]

9 : [23]

10: [11]

9.

```python
def solve(hSet, jSet, k):
    for i in jSet:
        if k - i in hSet:
            return True
    return False

hSet = {4, 5, 7, -1, -9, 22}
jSet = {0, -5, 10, 44, 100, -12}
print(solve(hSet, jSet, 91)) # True
print(solve(hSet, jSet, 92)) # False
```

10.

a) The worst-case complexity is $O(N)$

b) The worst case happens when we have to traverse to the last element in $jSet$. That means for i that is prior to the last one, $k - i$ does not exist in $hSet$.

11.

```python
def solve(hArray, k):
    numSet = set()
    for n in hArray:
        if k - n in numSet:
            return True
        numSet.add(n)

    return False
```