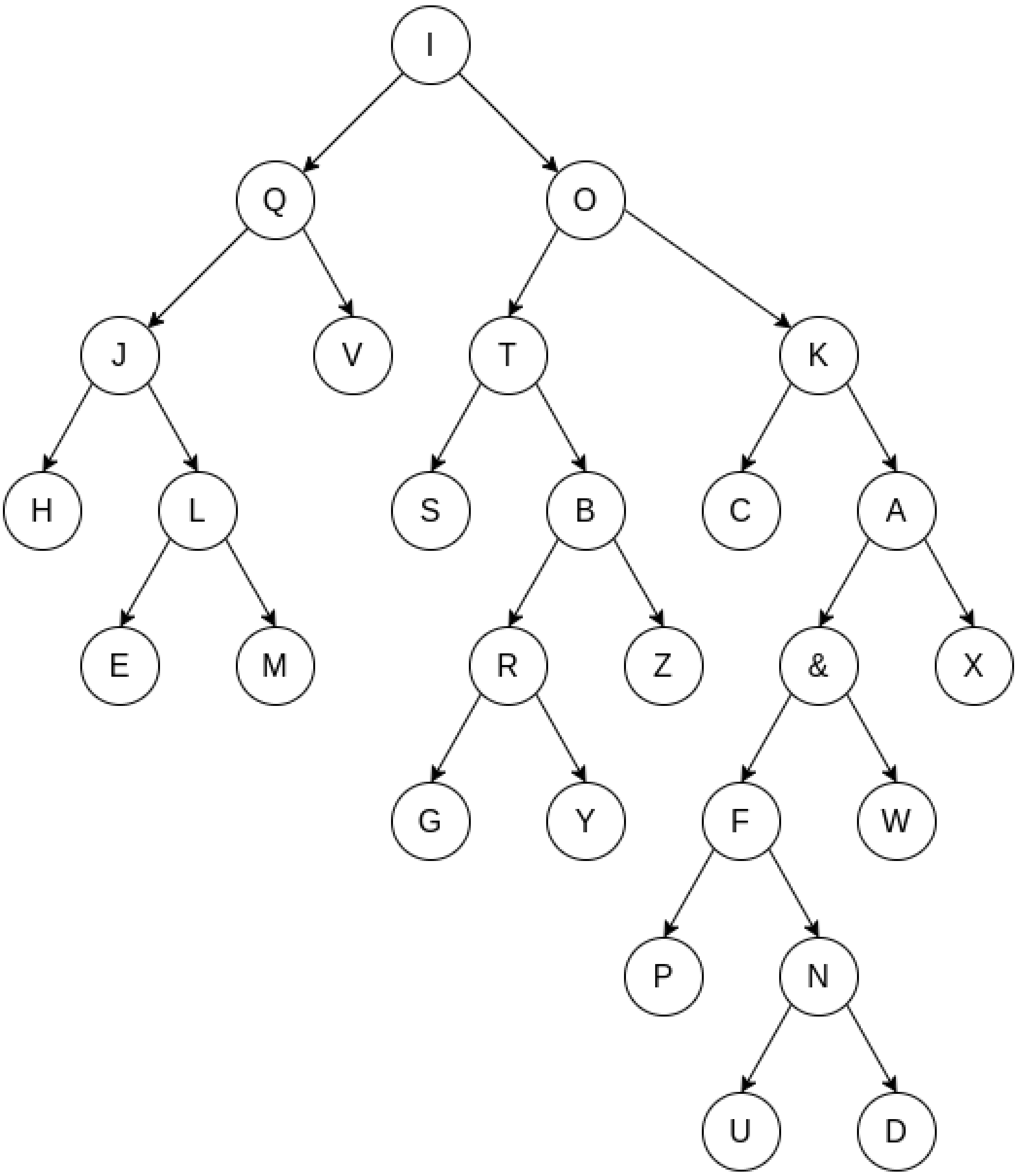
Solution for Problem Set 6

201300035 方盛俊

Problem 1

(a)



(b)

Algorithm:

We record the tree as N recursively, let the preorder node sequences as FIFO queue be R, the postorder node sequences as FIFO queue be O.

Algorithm 1 RecursiveReconstruct

```
function RecursiveReconstruct(N)
    if R.top() == O.top() then
        N.data = R.pop()
        O.pop()
    else
        N.data = R.pop()
        N.leftChild.parent = N
        RecursiveReconstruct(N.leftChild)
        N.rightChild.parent = N
        RecursiveReconstruct(N.rightChild)
        O.pop()
    end if
end function
```

Correctness:

For a full binary tree, which every non-leaf node has exactly two children, we reconstruct it recursively just like preorder traversal and postorder traversal.

If we only focus on the `R.pop()`, we will find it just replaced `R.push()` into `R.pop()` adapted from code of `PreorderTrav()`, so it is the inverse function of `PreorderTrav()`, it can reconstruct a tree that matches the preorder node sequences. Similarly, the reconstructed tree matches the postorder node sequences.

And the function is not a randomized algorithm, so the answer it produced is unique and correct.

Time Complexity:

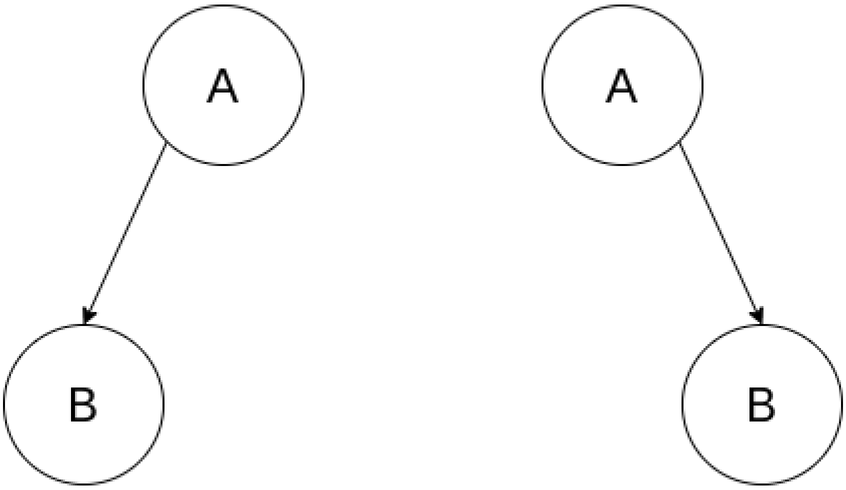
Each turn of calling the `RecursiveReconstruct()` function will reduce the length of R and O by 1, and the length of R and O is n. We can conclude that the function will be called n times.

Because the function's time complexity is $\Theta(1)$, without the recursive statement, the final time complexity is $T(n) = n \cdot \Theta(1) = O(n)$.

(c)

The answer of reconstructing an arbitrary binary tree from its preorder and postorder node sequences may be not unique.

For example, the simplest case is that preorder node sequences is `A B`, and the postorder sequences is `B A`. There are two trees meet the conditions:



The B node is the left child of A or the B node is the right child of A.

The answer is not unique so that there is no algorithm to reconstruct an arbitrary binary tree from its preorder and postorder node sequences.

Problem 2

Algorithm:

In a height-balanced binary search tree, the difference between the height of the left and right subtrees of every node is never more than 1. In other words, the different between the height of the left and right subtrees of every node is never more than 2^{h-1}, where h is the height of the tree, which means that `max <= 2 * min + 1`.

So the only thing we are supposed to do is balancing the difference by $O(n)$ rotations.

Let M be the global hash map for storing the count of nodes for each node.

Algorithm 2 Balance

```
function InitCount(N)
    if N == NULL then
        return 0
    end if
    count = InitCount(N.leftChild) + InitCount(N.rightChild) + 1
    M.set(key = N, value = count)
    return count
end function
function RecursiveBalance(N)
    leftCount = M.get(key = N.leftChild)
    rightCount = M.get(key = N.rightChild)
    while leftCount > 2 * rightCount + 1 or rightCount > 2 * leftCount + 1 do
        if leftCount > 2 * rightCount + 1 then
```

```
            middleCount = M.get(key = N.leftChild.rightChild)
            leftLeftCount = M.get(key = N.leftChild.leftChild)
            if rightCount + middleCount > 2 * leftLeftCount + 1 then
                temp = N.leftChild
                N.leftChild = N.leftChild.rightChild
                M.set(key = temp, value = rightCount + middleCount + 1)
                M.set(key = temp.leftChild, value = rightCount + leftCount + 1)
                leftRotation(temp)
            else
                M.set(key = N, value = rightCount + middleCount + 1)
                M.set(key = N.leftChild, value = rightCount + leftCount + 1)
                rightRotation(N)
                N = N.leftChild
                leftCount = M.get(key = N.leftChild)
                rightCount = M.get(key = N.rightChild)
            end if
        else
            middleCount = M.get(key = N.rightChild.leftChild)
            rightRightCount = M.get(key = N.rightChild.rightChild)
            if leftCount + middleCount > 2 * rightRightCount + 1 then
                temp = N.rightChild
                N.rightChild = N.rightChild.leftChild
                M.set(key = temp, value = leftCount + middleCount + 1)
                M.set(key = temp.rightChild, value = leftCount + rightCount + 1)
                rightRotation(temp)
            else
                M.set(key = N, value = leftCount + middleCount + 1)
                M.set(key = N.rightChild, value = leftCount + rightCount + 1)
                leftRotation(N)
                N = N.rightChild
                leftCount = M.get(key = N.leftChild)
                rightCount = M.get(key = N.rightChild)
            end if
        end if
    end while
    RecursiveBalance(N.leftChild)
    RecursiveBalance(N.rightChild)
end function
function BALANCE(root)
    InitCount(root)
    RecursiveBalance(root)
end function
```

Time Complexity:

For each node, we will balance it by the main part of `RecursiveBalance()` function. In the function, we do a loop with severals rotations to make sure that its subtrees be divided into two nearly equal parts. The function `RecursiveBalance()` will rotations several times, which can be divided into two parts, at most `n / 3` times middle rotations (rotations to reduce `middleCount`) and once left rotation or right rotation.

The number of total middle rotations is no more than `n` (because middle count is no more than `n`), and the number of total other rotation is also no more than `n` (because each call of the function will rotation once or none).

So the final time complexity is $T(n) = O(n) + O(n) = O(n)$

Problem 3

Algorithm:

Algorithm 3 Transform

```
function RecursiveRotationsToChain(N)
    while N.leftChild != NULL do
        temp = N
        N = temp.leftChild
        rightRotation(temp)
    end while
    RecursiveRotationsToChain(N.rightChild)
    Return N
end function
function MoveParentIntoTree(N)
    if N.leftChild == NULL and N.rightChild == NULL then
        return
    end if
    if N.leftChild == NULL then
        if N.data < N.rightChild.data then
            leftRotation(N)
        else
            SwapSubtree(N)
            rightRotation(N)
        end if
    else
        if N.data < N.leftChild.data then
            rightRotation(N)
        else
            SwapSubtree(N)
            leftRotation(N)
        end if
    end if
    MoveParentIntoTree(N)
end function
function RecursiveBuildTree(N)
    if N.rightChild == NULL then
        return N
```

end if
RecursiveBuildTree(N.rightChild)
N = MoveParentIntoTree(N)
return N
end function
function TRANSFORM(root)
root = RecursiveRotationsToChain(root)
RecursiveBuildTree(root)
end function

Time Complexity:

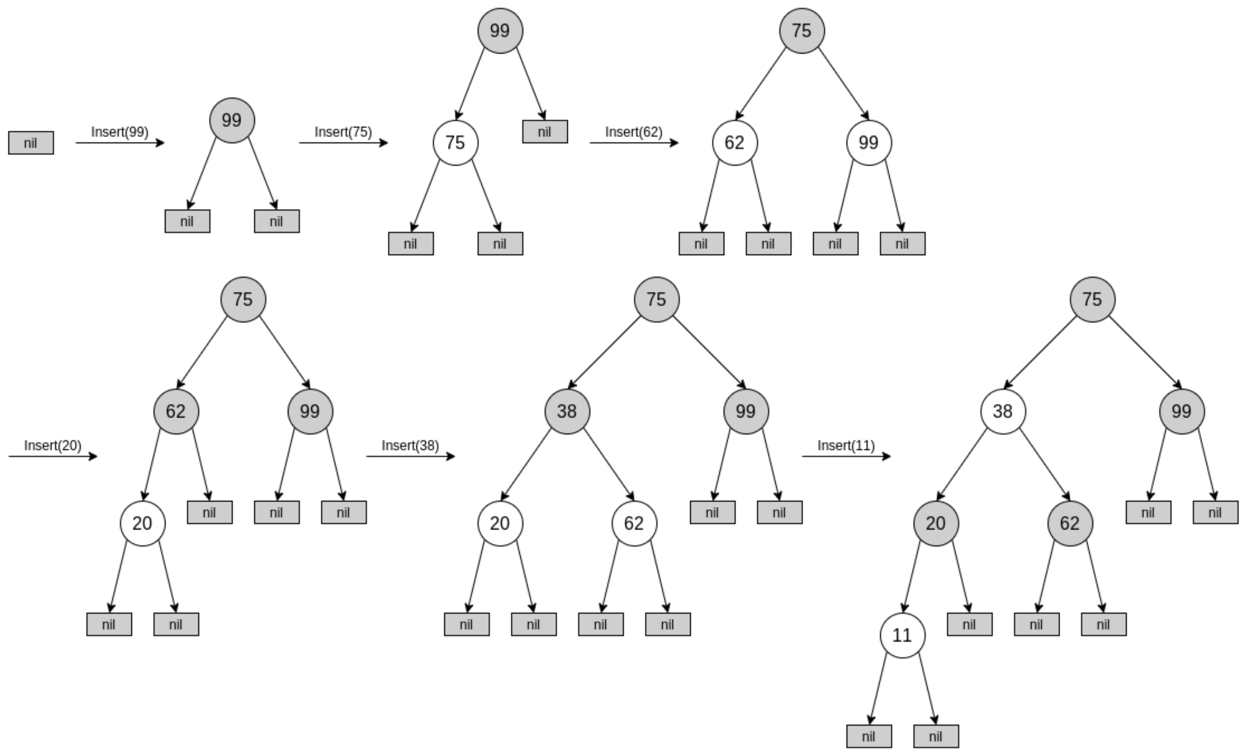
RecursiveRotationsToChain: $T_1(n) = O(n!) = O(n^2)$

RecursiveBuildTree: $T_2(n) = O(n!) = O(n^2)$

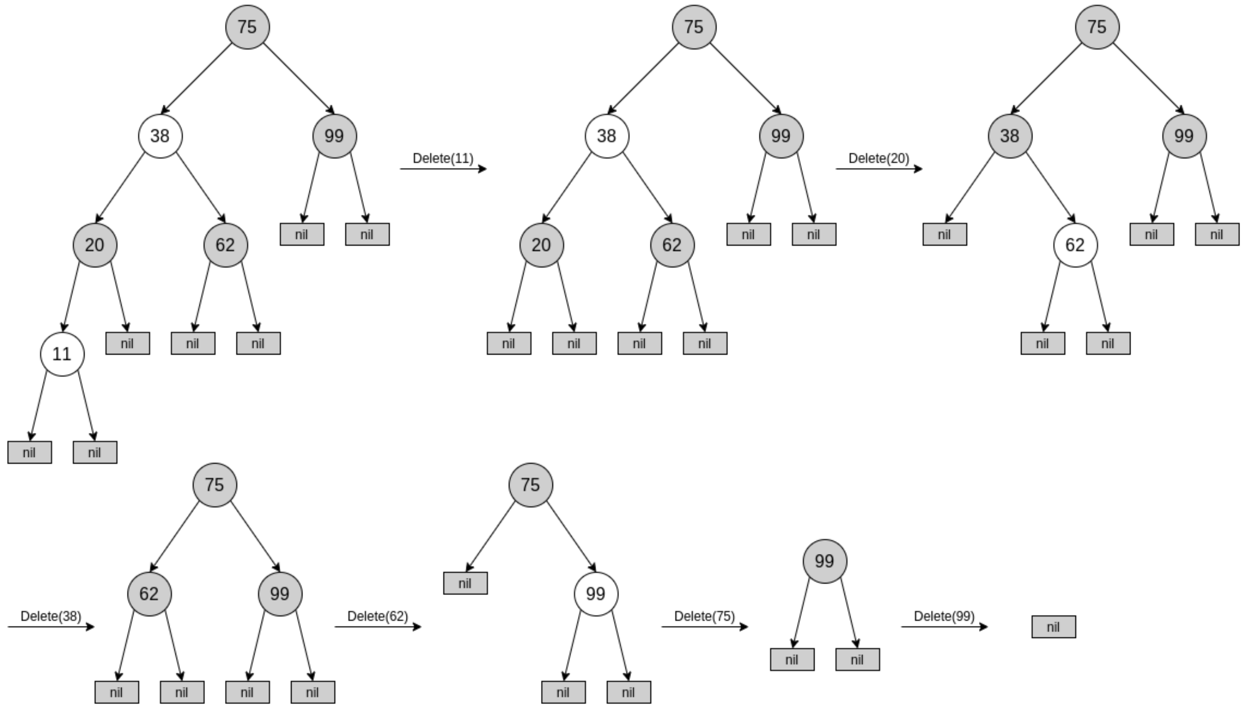
$\therefore T(n) = O(n^2) + O(n^2) = O(n^2)$

Problem 4

(a)



(b)



Problem 5

(a)

For a tree of height h, it at least have two subtrees, whose heights are at least $h-1$ and $h-2$. To get a tree having least F_h nodes, we need to make sure that their each subtrees have least F_{h-1} and F_{h-2} nodes.

So we can get a recursion formula:

$$F_h = \begin{cases} 1, & h = 1 \\ 2, & h = 2 \\ F_{h-1} + F_{h-2} + 1, & h \geqslant 3 \end{cases}$$

The F_h is a kind of Fibonacci number.

We want to find the h that meets the condition $n \leqslant F_h = F_{h-1} + F_{h-2} + 1$

$\therefore n \leqslant 2^{\frac{h}{2}} \leqslant \cdots \leqslant 2F_{h-2} \leqslant F_h = F_{h-1} + F_{h-2} + 1$

$\therefore h \geqslant 2 \log n$

We let $h = 2\log n$, then we can make sure that $F_h \geqslant n$

So an AVL tree with n nodes has height $O(\log n)$

(b)

Algorithm 4 Balance

```
function BALANCE(x)
    if x.leftChild.h > x.rightChild.h then
        if x.leftChild.leftChild.h < x.leftChild.rightChild.h then
            leftRotation(x.leftChild)
            return rightRotation(x)
        else
            return rightRotation(x)
        end if
    else
        if x.rightChild.rightChild.h < x.rightChild.leftChild.h then
            rightRotation(x.rightChild)
            return leftRotation(x)
        else
            return leftRotation(x)
        end if
    end if
end function
```

(c)

Algorithm 5 Insert

```
function INSERT(x, z)
    if z.data < x.data then
        if x.leftChild == NULL then
            x.leftChild = z
        else
            Insert(x.leftChild, z)
        end if
        if x.leftChild.h - x.rightChild.h == 2 then
            x = Balance(x)
        end if
        x.h = max(x.leftChild.h, x.rightChild.h) + 1
    else
        if x.rightChild == NULL then
            x.rightChild = z
        else
            Insert(x.rightChild, z)
        end if
        if x.rightChild.h - x.leftChild.h == 2 then
            x = Balance(x)
        end if
        x.h = max(x.rightChild.h, x.leftChild.h) + 1
    end if
end function
```

(d)

Time Complexity:

Because the insert function will call itself recursively, and $T(h) = T(h-1) + \Theta(1)$, where h is the height of x. Because of (a), we know that an AVL tree with n nodes has height $O(\log n)$, so $T(n) = O(\log n)$.

In order to prove that we only performs $O(1)$ rotations, we need to analyze how many times the Balance function will be called.

There are two cases, assure we have a node x, and x.leftChild.h $-$ x.rightChild.h $= 2$.

In first case, x.parent is height balanced, so the only thing we need to do is doing once `Balance()` on x, we can make sure that the final tree is height balanced. We only call `Balance()` once, so we only performs $O(1)$ rotations.

In second case, x.parent is also not height balanced. After we `balance(x)`, it is possible that the x.parent is still unbalanced, so we need to balance x.parent. In this call of `Balance(x.parent)`, we will rotation it and then the tree will return the form as if the first `balance(x)` was not executed. So, we can make sure that `x.leftChild.h - x.rightChild.h == 2` and `x.parent.leftChild.h - x.parent.rightChild.h == 2`, then we rotation the `x.parent`, will make sure that `x.leftChild.h = x.leftChild.h - 1` and `x.parent.rightChild.h = x.rightChild.h + 1`. After it, the tree is height balanced. We only call `Balance()` twice, so we only performs $O(1)$ rotations.

Problem 6

(a)

Each `Meld(Q1, Q2)` will reduce the height of Q_1 or Q_2, and terminates when Q_1 and Q_2 are both empty. So we can think the problem can be transformed into the expected length of a random root-to-leaf path in an n-node binary tree, Q_1 and Q_2.

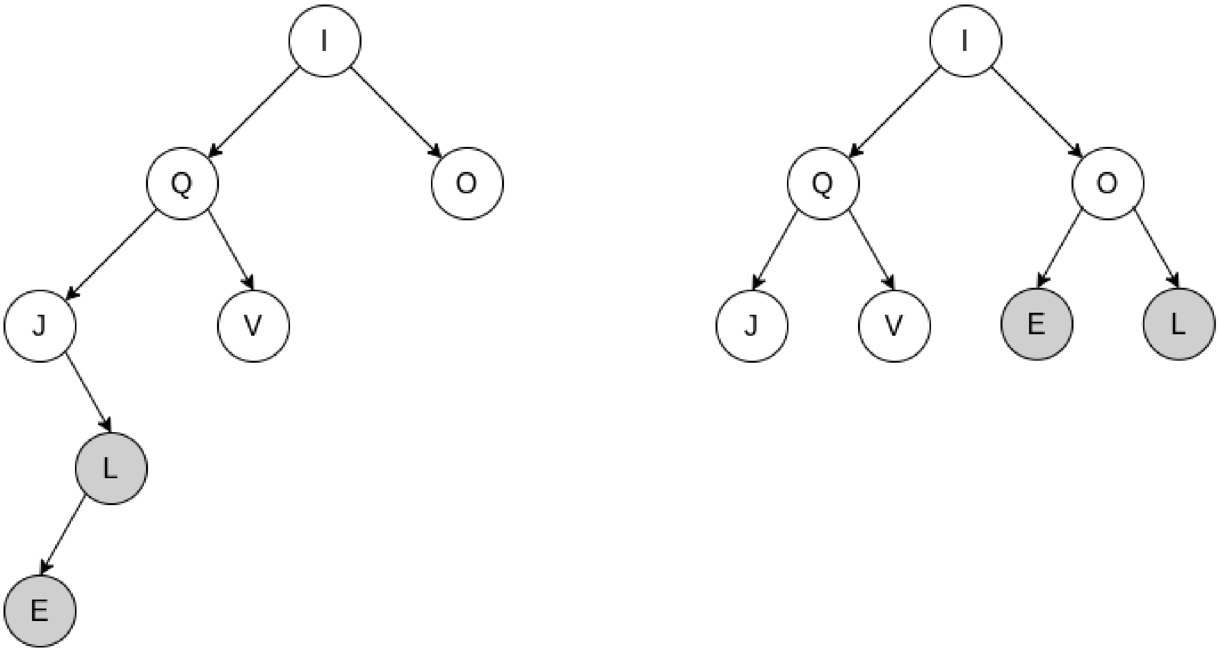
We look at the Q_1. Let m be the number of nodes of Q_1, $E(m)$ be the expected length of a random root-to-leaf path.

$$E(m) = \begin{cases} 0, & m = 0 \\ 1, & m = 1 \\ \dfrac{1}{2}E(r_m) + \dfrac{1}{2}E(m - r_m - 1) + 1, & m \geqslant 2, 0 \leqslant r_m \leqslant m-1 \end{cases}$$

For any node with height h, its contribution to $E(m)$ is $\dfrac{1}{2^h}$, which deceases when h increases.

So we can get $E(m) = \sum_{k=1}^{m} \frac{1}{2^{h_k}}$

We know that $\frac{1}{2^{h_1}} < \frac{1}{2^{h_2}}$, when $h_1 > h_2$



As we can see, $E(m)$ of the left tree is smaller than the right tree. So we can know that, for an binary tree with m nodes, $E(m)$ of the full-balanced tree is largest.

So $E(m) \leqslant 1 + 2 \times \frac{1}{2} + 4 \times \frac{1}{4} + \cdots = \sum_{k=1}^{\log m} 1 = \log m = O(\log m)$

Finally, the running time of `Meld(Q1, Q2)` is $T(n) = O(\log m) + O(\log(n-m)) = O(\log n)$

(b)

MakeQueue: Return a null node.

FindMin: Return `root.key` (except null node).

DeleteMin:

Algorithm 6 DeleteMin
```
function DELETEMIN(Q)
    return Meld(Q.left, Q.right)
end function
```

Insert:

Algorithm 7 Insert

```
function INSERT(Q, x)
    N = new Node(x)
    return Meld(Q, N)
end function
```

DecreaseKey:

Algorithm 8 DecreaseKey

```
function DECREASEKEY(Q, x)
    Delete x and its subtrees from Q
    N = Replace x and y with the subtrees
    return Meld(Q, N)
end function
```

Delete:

Algorithm 9 Delete

```
function DELETE(Q, x)
    N = Merge(x.left, x.right)
    Replace x in Q with N
    return Q
end function
```

So each of the other meldable priority queue operations can be implemented with at most one call to Meld and $O(1)$ additional time.