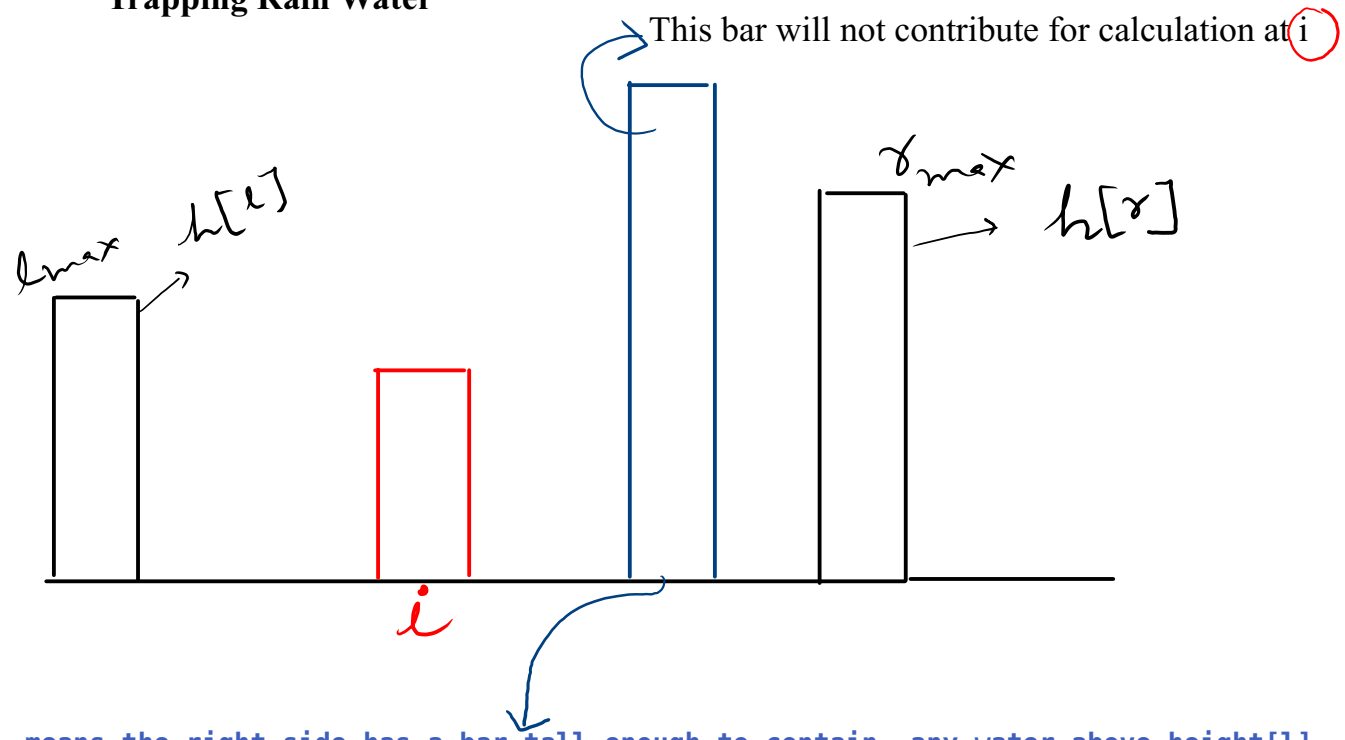
Pre-two-pointer:

We can maintain two arrays one will store leftMax seen so far
and other will store rightMax seen so far.
Now traverse the input array and use the formula min(left-max, right-max) - height[i].

Trapping Rain Water

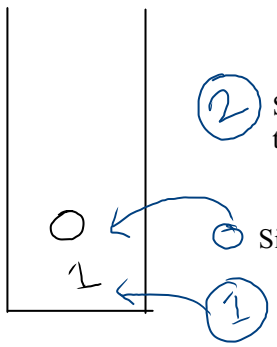
1. Two pointer approach



If height[l] < height[r], means the right side has a bar tall enough to contain any water above height[l].
In this case, it doesn't matter whether there are bars between height[r] and the current pointer position (i)
that are taller or shorter than height[r]. Why? Because height[r] guarantees that no water will spill over it.
This means that the water trapped at left depends solely on leftMax, not on rightMax.

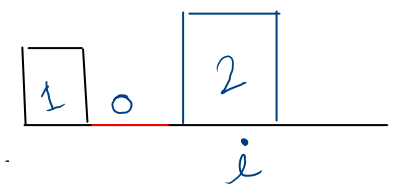
```
public int trap2PApproach(int[] height) {
int l = 0;
int r = height.length - 1;
int lmax = 0;
int rmax = 0;

int trappedWater = 0;

while (l < r) {

    // trapping water happen because of lmax
    if (height[l] < height[r]) {

        if (height[l] >= lmax) {
            // adjust the lmax, water cannot be trapped
            lmax = height[l];
        } else {
            // water can be trapped
            trappedWater += lmax - height[l];
        }

        l++;

    } else {
        // trapping water happen because of rmax
        if (height[r] >= rmax) {
            // adjust the rmax, water cannot be trapped
            rmax = height[r];
        } else {
            // water can be trapped
            trappedWater += rmax - height[r];
        }

        r--;
    }
}
```

2. Monotonic stack



Since 2 is greater than ZERO, so we will pop it from the stack and calculate the water trapped. Water will be trapped between 2 and 1.

Since ZERO is smaller than 1 so push it on stack




```java
public static class TrappingRainWaterWithStack {
    public int trap(int[] height) {
        if (height == null || height.length < 3)
            return 0;

        Stack<Integer> stack = new Stack<>();
        int waterTrapped = 0;

        for (int i = 0; i < height.length; i++) {
            // While the current height is greater than the height at the top of the stack
            while (!stack.isEmpty() && height[i] > height[stack.peek()]) {
                int bottom = stack.pop(); // Pop the top (bottom of the valley)
                if (stack.isEmpty())
                    break; // No left boundary

                int left = stack.peek(); // Left boundary
                int width = i - left - 1; // Distance between left and current bar
                int boundedHeight = Math.min(height[left], height[i]) - height[bottom];
                waterTrapped += width * boundedHeight; // Add trapped water
            }
            stack.push(i); // Push the current bar onto the stack
        }

        return waterTrapped;
    }

    public static void main(String[] args) {
        TrappingRainWaterWithStack solution = new TrappingRainWaterWithStack();
        int[] height = { 0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1 };
        System.out.println(solution.trap(height)); // Output: 6
    }
}
```