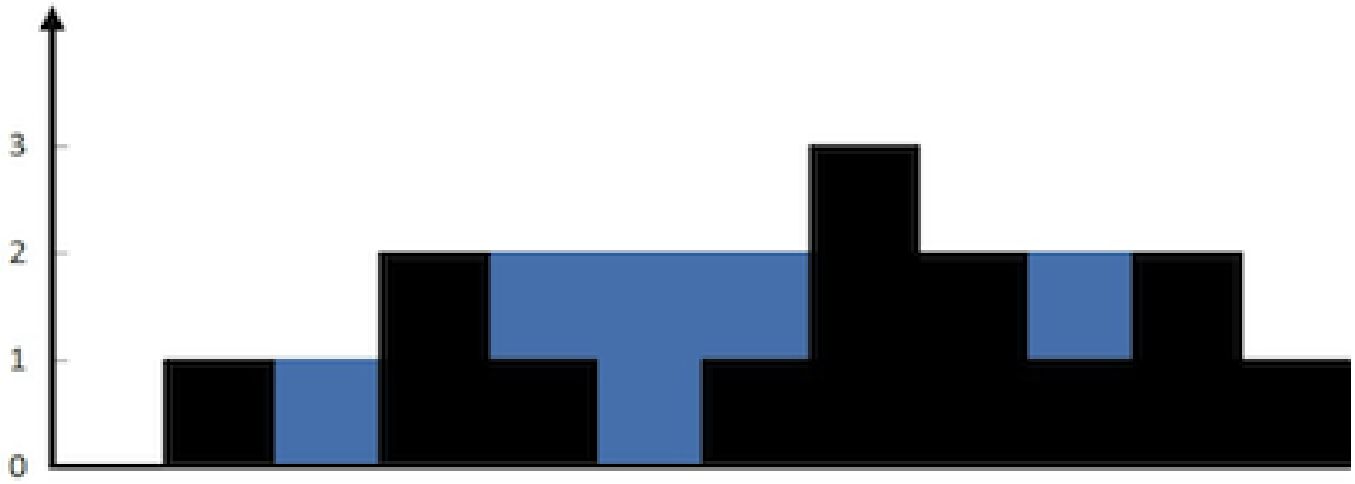
Stacks

Leetcode: *42*. Trapping Rain Water

Given n non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it can trap after raining.

Example 1:



Input: height = [0,1,0,2,1,0,1,3,2,1,2,1]
Output: 6

Explanation: The above elevation map (black section) is represented by array [0,1,0,2,1,0,1,3,2,1,2,1]. In this case, 6 units of rain water (blue section) are being trapped.

Example 2:

Input: height = [4,2,0,3,2,5]
Output: 9

Approach:

If we are standing at 5, then the amount of water collected would be max of the right (at 7, with height 3) and on the left (at 3. With height 2). So,

$$\text{Min } (3 - 2) - 0 = 2$$

‘0’ being the height of the element.

If we are at 4, the max on the right would be 3 and left would be 2. Min of them would be 2 and if we remove the height of the building then it would be 1.

If we are standing at 9, the max on the right is 2 and on the left is 3. The min of (3,2) is 2. The height of the building at 9 is 1, so 2 – 1 = 1.

We are finding all the heights on the right and on the left so the **time complexity** would be **O (n).**

CODE:

```
def trap():

    n = len(height)
    total_water_trapped = 0

    for idx, h in enumerate(height):

        max_value_right = float('-inf')
        for i in range(idx + 1, n):
            if height[i] > max_value_right:
                max_value_right = height[i]


        max_value_left = float('-inf')
        for i in range(0, idx):
            if height[i] > max_value_left:
                max_value_left = height[i]


        water_trapped = min(max_value_left, max_value_right) -
 height[idx]
        if water_trapped > 0:
            total_water_trapped += water_trapped

    return total_water_trapped
```

We can use 0 or -1 instead of -inf, but using -inf is a good practice.

Queues

Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out (FIFO) methodology, i.e., the data item stored first will be accessed first.

A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.

Queue Representation

As we now understand that in queue, we access both ends for different reasons. The following diagram given below tries to explain queue representation as data structure –

As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers and Structures. For the sake of simplicity, we shall implement queues using one-dimensional array.

Basic Operations:

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. here we shall try to understand the basic operations associated with queues −

- **push/enqueue()** − add (store) an item to the queue.
- **pop/dequeue()** − remove (access) an item from the queue.

- **isempty()** – checks if the queue is empty.

In a empty list = []

enqueue(5) ➔ [5]
enqueue(6) ➔ [5, 6]
enqueue(8) ➔ [5, 6, 8]
dequeue ➔ [5, 6]
dequeue ➔ [5]

```
queue = list()

def enqueue(x):
    global queue
    queue.append(x)

def dequeue():
    global queue
    if is_empty():
        return
    return queue.pop(0)

def is_empty():
    global queue
    return len(queue) == 0

if __name__ == "__main__":
    enqueue(6)
    enqueue(99)
    print(dequeue())
    print(dequeue())
```

OUTPUT:

6
99

```
queue = list()

def enqueue(x):
    global queue
    queue.append(x)

def dequeue():
    global queue
    if is_empty():
        return
```

```
    x = queue[0]
    queue.pop(0)
    return x

def is_empty():
    global queue
    return len(queue) == 0

if __name__ == "__main__":
     enqueue(6)
     enqueue(99)
     print(dequeue())
     print(dequeue())
     print(queue)
```

How to generate a series as:

1
2
11
12
21
22
111
112
121
122

If we have a queue, we enqueue 1 & 2 to the queue. Whenever we dequeue from the queue 1, we decrement the count (from 5 to 4) and will print 1. And then enqueue 1 & 2 to the queue.

Then we will dequeue 2 and decrement the count (from 4 to 3) and will print 2. Then enqueue (21 & 22).

Then dequeue 11 and decrement the count (from 3 to 2) and print 11. Then enqueue (111 & 112)