8.7 Implement a circular queue

8.7 7/10

Note: Did not attempt this problem.

~~7/10~~

Imp. a queue API for storing elements using arrays.

Must have:

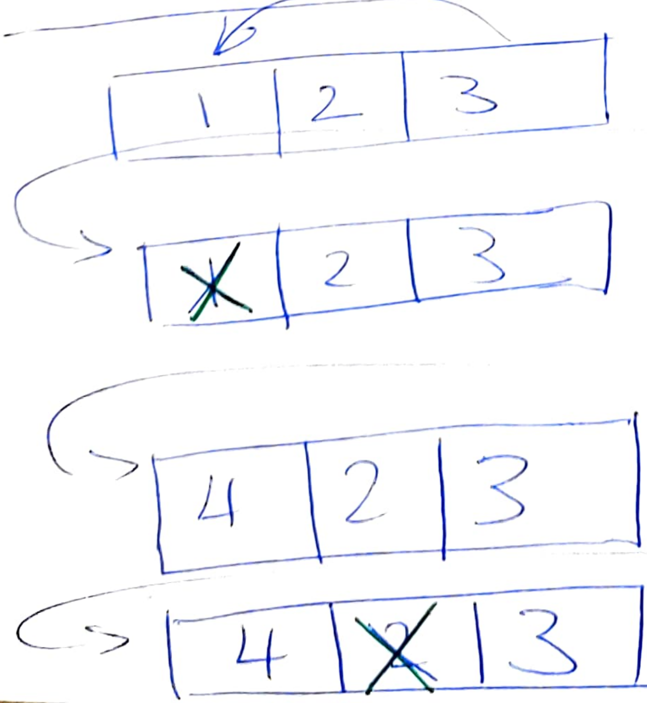
(1) Constructor f'n which takes the initial capacity of queue, enqueue, and dequeue f'n. and a f'n that returns the number of elem. stored.

(2) Must have dynamic resizing.

FIFO principle ← Circular Queue

↳ Last position is connected to the first pos.

↳ "Ring Buffer"



pop an elem

add an elem. 4

pop another elem. 1/5

(when the list is full, ~~it overwrite~~ and we want to add elem, it overwrites the oldest)

Notes:

Circular Queue uses Circular Linked List = Circular List/array

↳According to Knuth.

→ Has property that its last node links back to first.

↳ Then it's possible to access all of the list starting from any given points.

↳ extra degree of symmetry.

↳ We don't need to think of the list as having a first or last node.

The sol'n for this problem uses dynamic resizing, which doesn't exist in ~~other variation~~ ~~of the~~ the leetcode version.

- dynamic resize → when full, double to (or by whatever else) the size

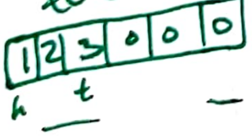
↳ Amortize Analysis

↳ Enqueue has O(1) time

```
class Queue:
    SCALE_FACTOR = 2
    def_init_(self, capacity: int) -> None:
        self._entries = [0] * capacity
        self._head = self._tail = _size = 0
                 positions
    def enqueue(self, x: int) -> None:
        if self._size == len(self._entries):
            # If the list is full, we need to resize
            self._entries = (self_entries[self._head:]
                             + self._entries[:self_head])
            # Reset head and tail position
            ~~self._entries[self._tail] = x~~
            self._head, self._tail = 0, self_size
            self._entries += [0] * (len(self._entries)
                                    * Queue.Scale_FACTOR*2
                                    - len(self._entries))
        self._entries[self._tail] = x   #SET new value
        self._tail = (self._tail + 1) % len(self_entries)
        self._size += 1
```

~~dynamic~~ ~~resizing~~ Make queue elem appear consecutively

We double the size



(Division remainder)

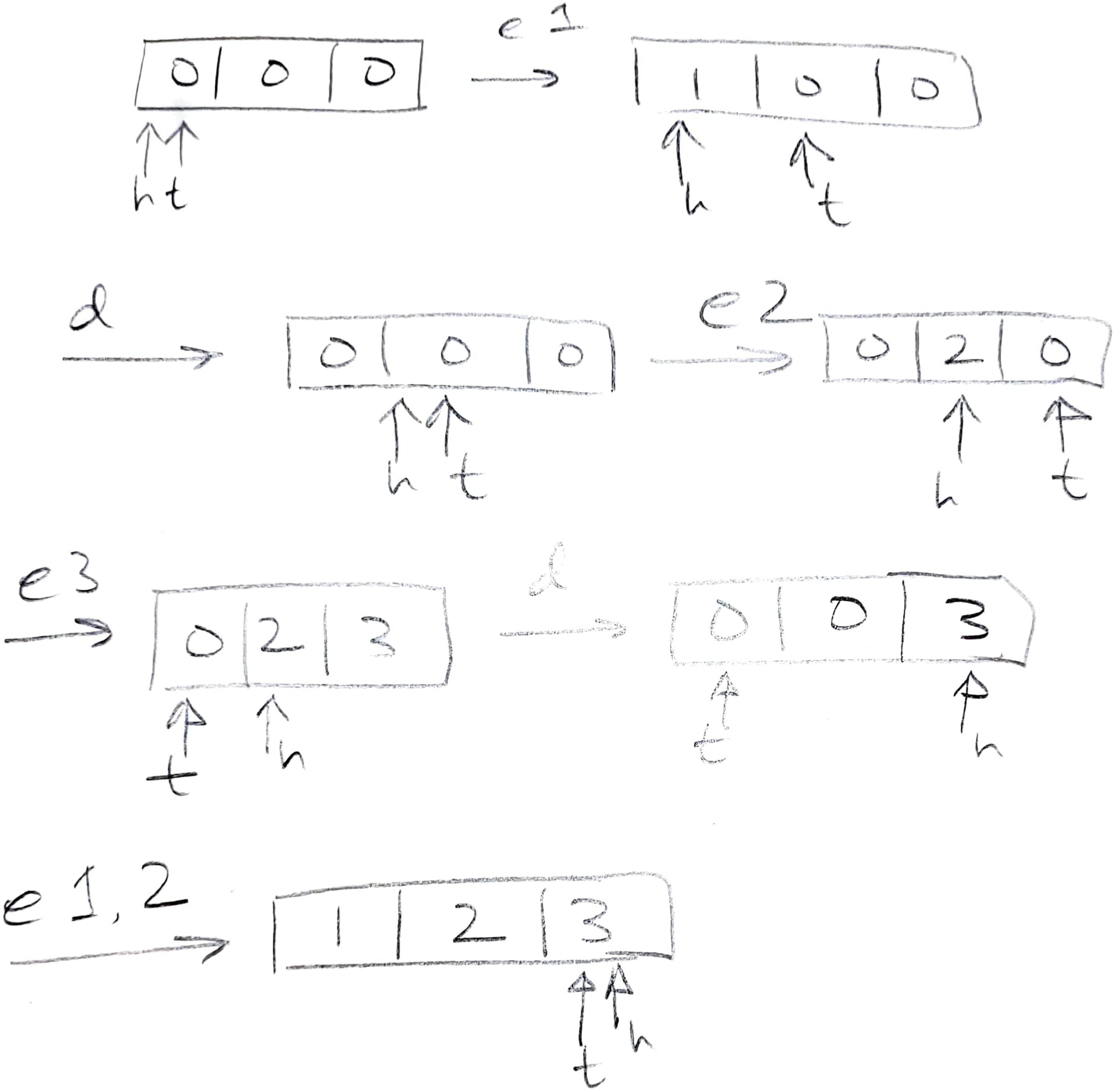
1%3 = 1
2%3 = 2
3%3 = 0
4%3 = 1
5%3 = 2

To ~~[illegible]~~ move through list "circularly"

```
def dequeue(self):
    if not self_size:
        raise IndexError('empty queue')
    self._size -=1
    ret = self._entries[self._head]
    self._head =(self._head + 1) % len(self._entries)
    return ret      #returns the removed
                     element

def size(self):
    return self._size
```

⟹ O(1) time for enqueue and dequeue.



0 | 0 | 0
h t
e 1
1 | 0 | 0
h
t
d
0 | 0 | 0
h t
e2
0 | 2 | 0
h
t
e3
0 | 2 | 3
t
h
d
0 | 0 | 3
t
h
e 1, 2
1 | 2 | 3
t
h

8.3 Is string well-formed?

8.3 7/10/22

- Design was correct but could not code it correctly.
- Spent too much time on the Naive Approach while the correct approach ~~approach~~ using stack idea was coming to me.
- ~~I didn't~~ I was having hard time coding so I was looking for alternative approach.
- I should've been able to code this.
- Not sure why "()" is not well-formed

- The part that I couldn't get was when ~~the~~ taking a value ~~check~~ that is a closing bracket, check to see if the elem in the stack is an opening bracket.

→ Was coding: elif s[i] = d[s[i-1]]

When it should be: elif not stack or d[stack.pop()] ≠ s[i]:

↳ I was not using the stack after pushing elem to it.

8.3 7/10

- "{,}, (,), [,]"

  "([]){()}", "[()[]{()()}]"

  are well-formed strings.

- "{)", "()", "[()[]{()()" are not.

  ↑
  ?

I/O : str / bool

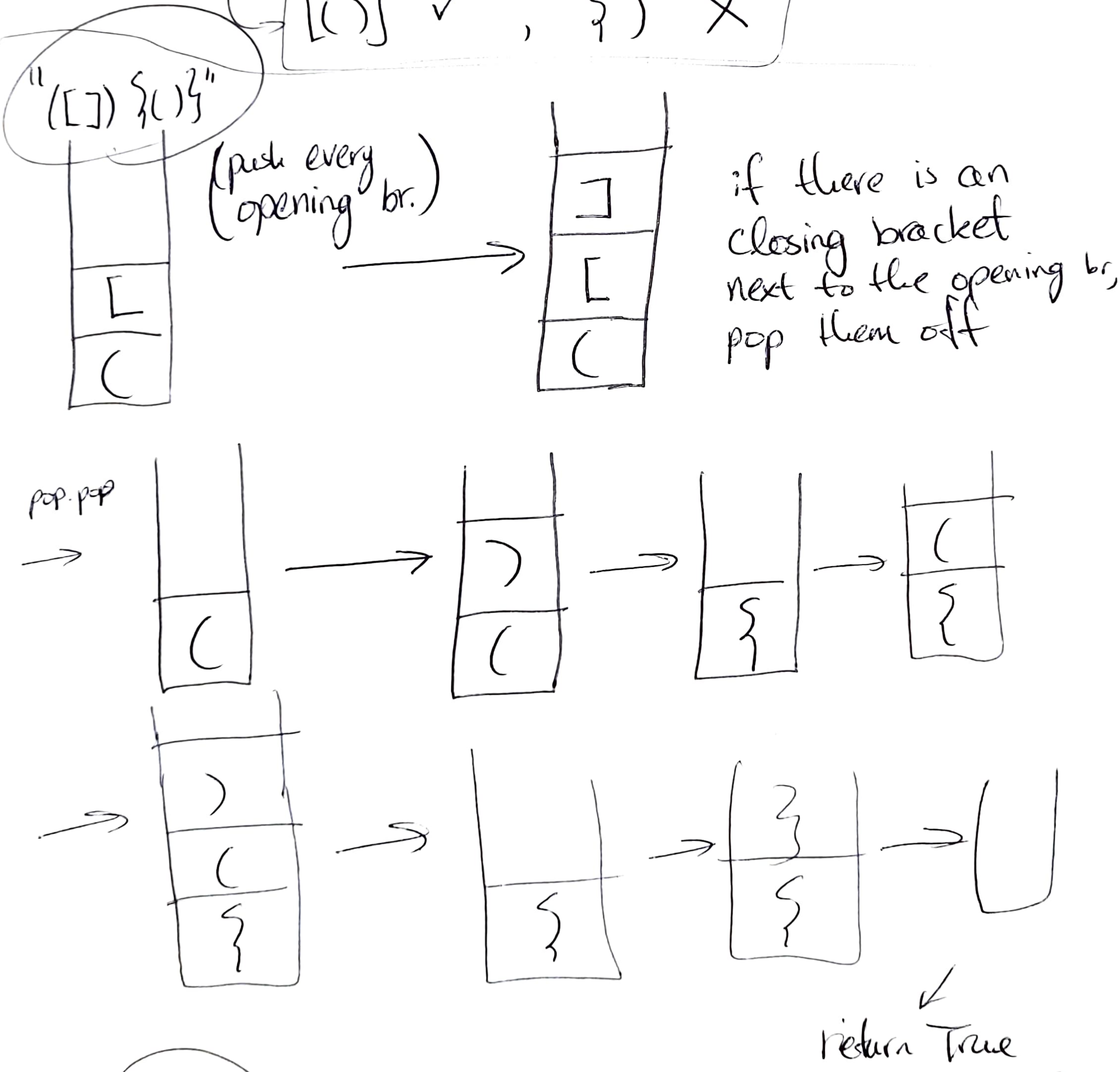
- Probably have to use hashmap to check if (,), [,], {, }.

- The Naive Approach would be to take a string elem (after splitting the input) where the elem. is "(", "[", "{" and check to see if the closing bracket exists and move on to the elem if yes.

X amin

Using stacks, we need to chock if forevery opening bracket, the closing bracket exists in the correct way

[( )] ✓ , {) X



if there is an closing bracket next to the opening br, pop them off

14 min

2/3

when having an initial opening bracket, [

The next pushed elem. must be an opening bracket or a closing bracket.

↳ Else, return False.

→ $O(n), O(1)$ → space is $O(1)$ since stack mem. is temp.

16 min
moving on to coding

26 min, maybe I should've used Linked List?

↳ pointers?

↳ 33 min Quit

8.8 Implement Queue using Stacks API

7/10
8.8

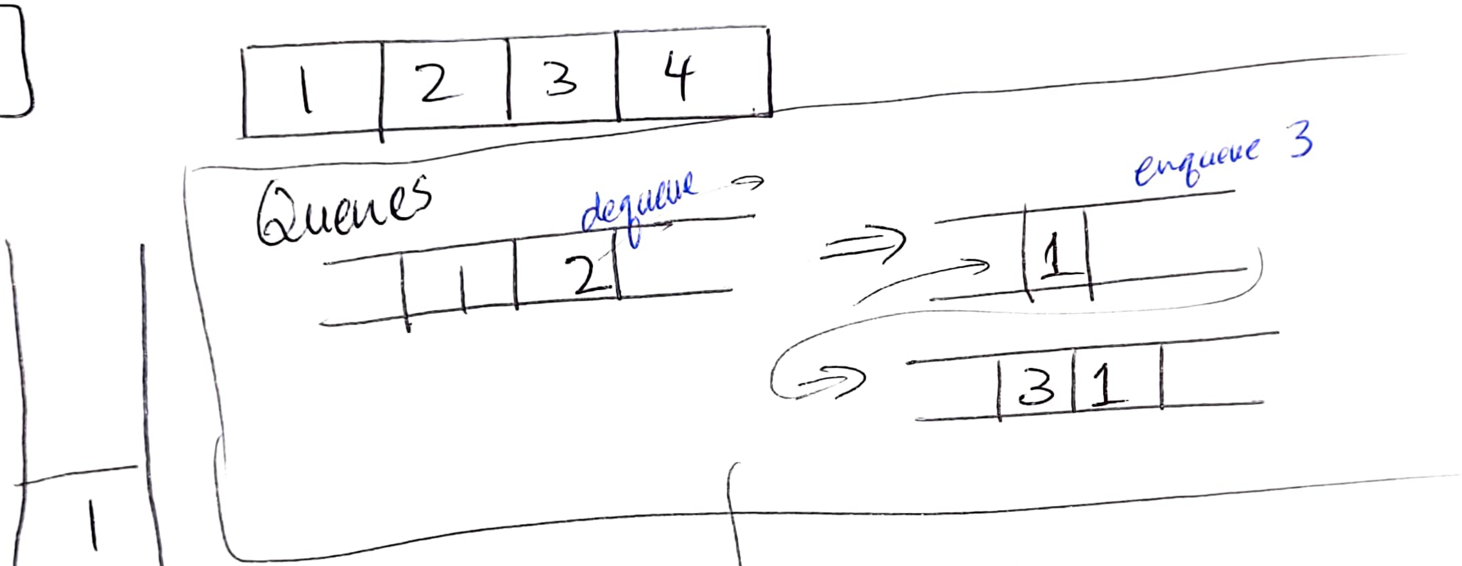
8.8 7/10

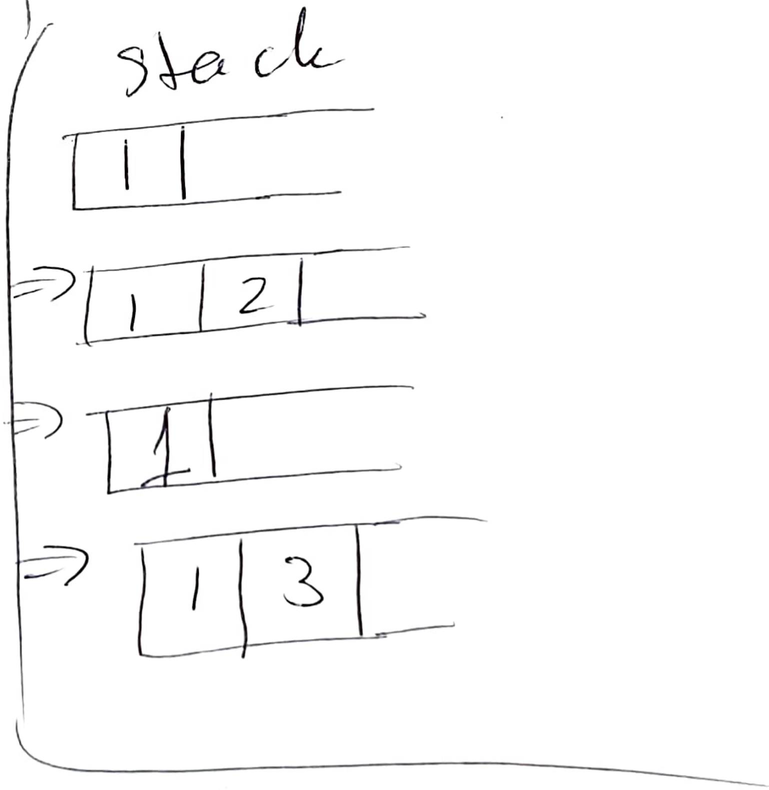
Queue : FIFO

Stack : LIFO

- Implement queue given stack API.

Ex





- Use pop~~left~~ right () ?

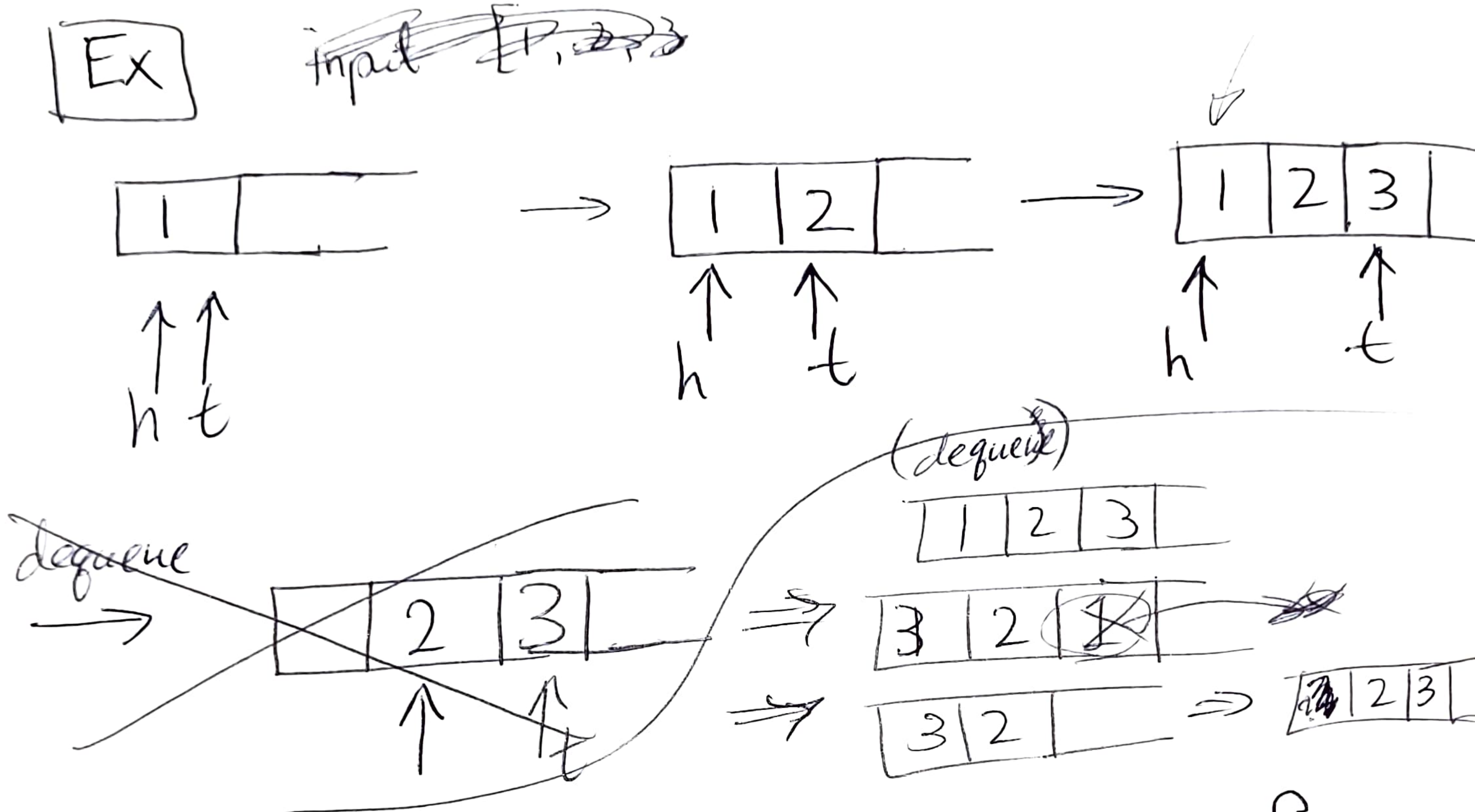
- Use pointers to keep track of head and tail.

- What is the I/O?

- When empty?
- Linked List?

- ~~Dynamically~~ resize?

Ex ~~input [1,2,3]~~



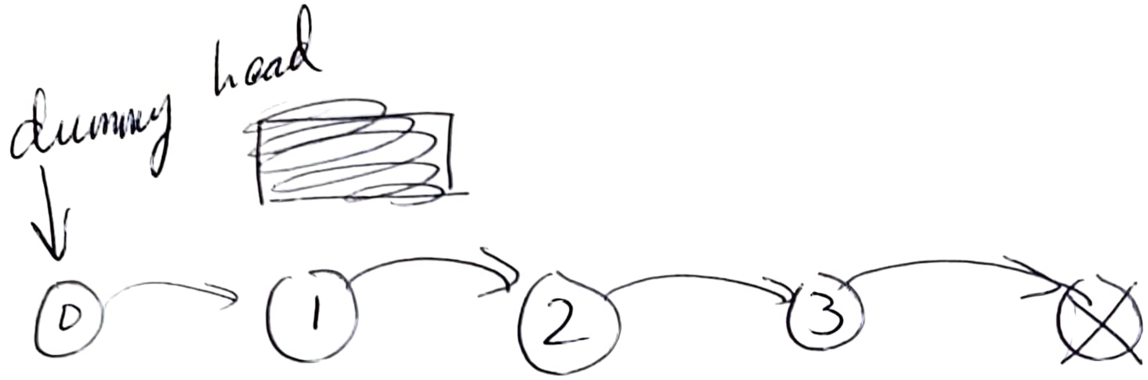
→ We would have to swap position of the elem using ptrs. and pop and swap again.

A Naive Approach would be to ~~push~~ queue elem by appending and dequeue by slicing or deleting the initial elem in the list

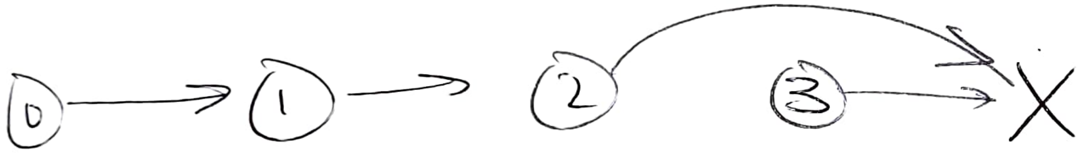
→ I feel that we have to use a linked list to find the sol'n.

18min

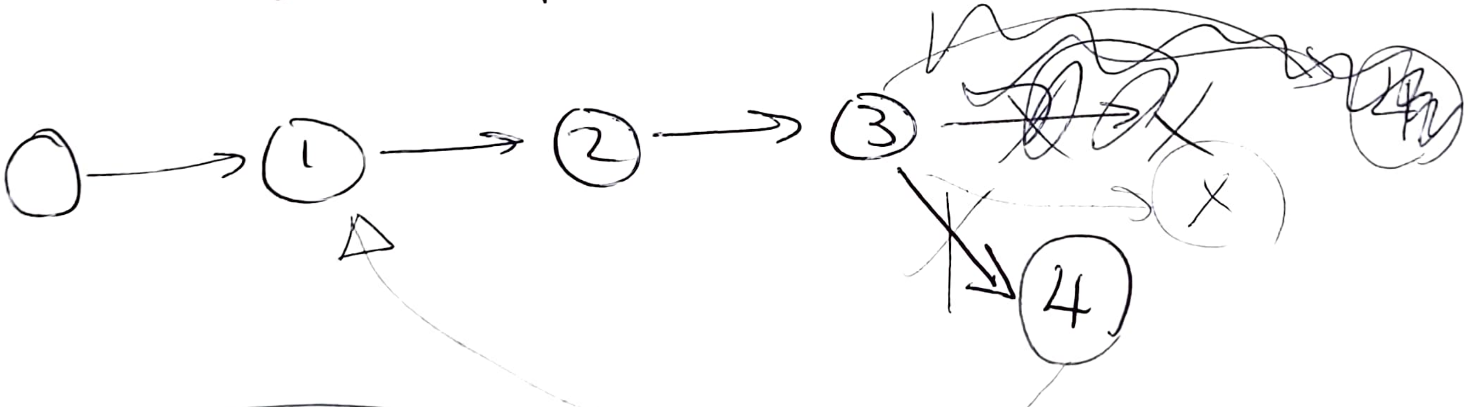
2/4



- ~~To queue, change the dummy value?~~
- To dequeue, Change the second to last pointer to point to null.

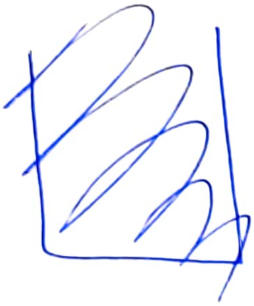


To queue, push an elem and change ptrs.



25min Quit

3/4



enqueue by just appending to a list

"enq" 1, 2, 3, 4

1	2	3	4	

(

4	3	2	1	

) → dequeue

deq

↳ another list =

4	3	2	1

(Pop) ⟹

4	3	2

(pop() append to original stack)

⟹

2	3	4	

→ return