28/02/22

DSML Intermediate – DSA

Stacks & Queues – Basics

- ✓ ① Linked List – Deletion + Few Caveats
- ✓ Google Problem on Delete Node
- ✓ ② Stack & Queue basics + ADT
- ✓ ③ Implementing Stack & Queues
  - ✓ . Using arrays (Lists)
  - ✓ . using Singly Linked Lists
- ✓ ④ 2 Interview problem *

Regarding Assignment + HW Ques

Intermediate Content

LL Assignment ✓ Q1, ✓ Q2 } Design Linked list

Insert
Delete
Print etc

LL HW

Q1 Flatten Nested list → Actually OOP + Recursion Problem
Q2 Middle Element ✓
Q3 Remove duplicates ✓

Content

1. LL Basics – 3 sessions – 3/4 }
2. Stacks & Queues basics (28)
3. Tree Basics (2nd March)
4. Tree-2 Basics (4th March)

Contest Discussion – 6th March

7, 9, 11 March Break | Problem Solving Sessions

DSML Adv Content: 15th March Onwards. | Few basic Python

① Return head node in insertion | deletion methods if you are updating it.

Show you code at end.

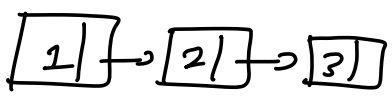
return head

② For ease of implementation, you can create 2 classes

1- Linked list

2- Node

```
Class Node:
    def __init__(self, x):
        self.data = x
        self.next = None
```



Singly Linked List

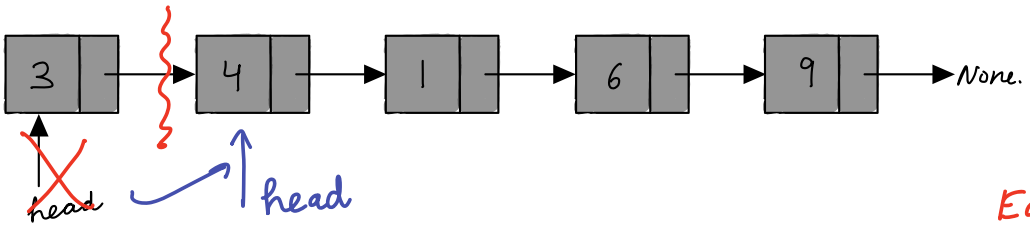
```
Class LinkedList:                    Node
    def __init__(self, head):
        self.head = head
    def insert(self, data, pos):
        # Code
    def remove(self, pos):
        # Code
```

```
n = Node(5)
my_list = LinkedList(n)
my_list.insert(__)
```

Doubly L.L.

Linked List - Deletion

(A) At head



Dangling Reference

Garbage collector.

Edge cases

Empty list ⇓ nothing to delete

```
def delete_at_head(head):
    if head:
        head = head.next
    return head.
```

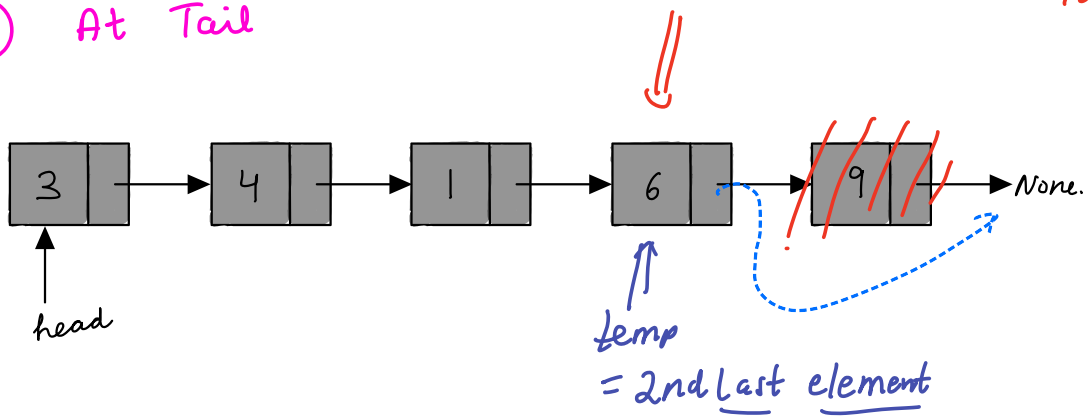
TC: O(1) , SC: O(1)

Q) What happens to the old head?

- Dangling reference
- Freed up during garbage collection.

(B) At Tail

Q) Where to reach? How to identify?

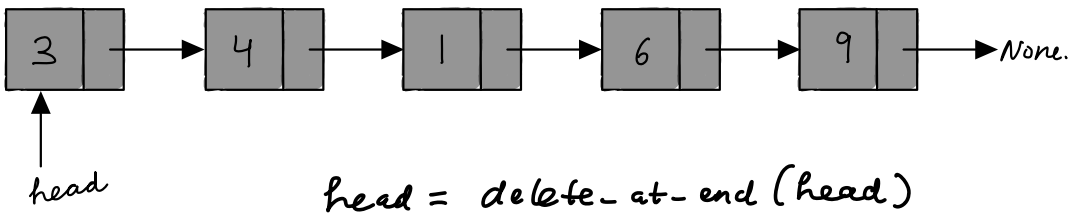


Edge Cases to not miss			
No.	Linked list	N	
1.	head = None	0	None
2.	head → 1 → None	1	None
3.	head → 1 → 2 → None	2	→ 1 → None
4.	1 → 2 → 3 → None (↑ head, ↑ temp)	3	→ 1 → 2 → None
5.	1 → 2 → 3 → 4 → None (↑ head)	4	→ 1 → 2 → 3 → None

Steps to Delete:

1 — Reach 2nd last node

2 — Set it next to be None



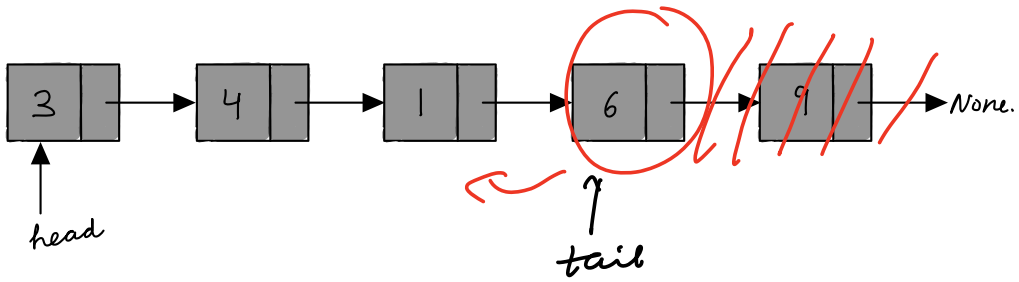
head = delete_at_end (head)

```
def delete_at_end (head):
    # Edge cases
    if (head == None):                    → size 0
        return None
    if (head.next == None):               → size 1
        return None
    temp = head                           >= 2
    while ( temp.next.next != None ):     (atleast 2 nodes are there.)
        temp = temp.next
    temp.next = None.
    return head
```

TC: O(N) , SC: O(1)

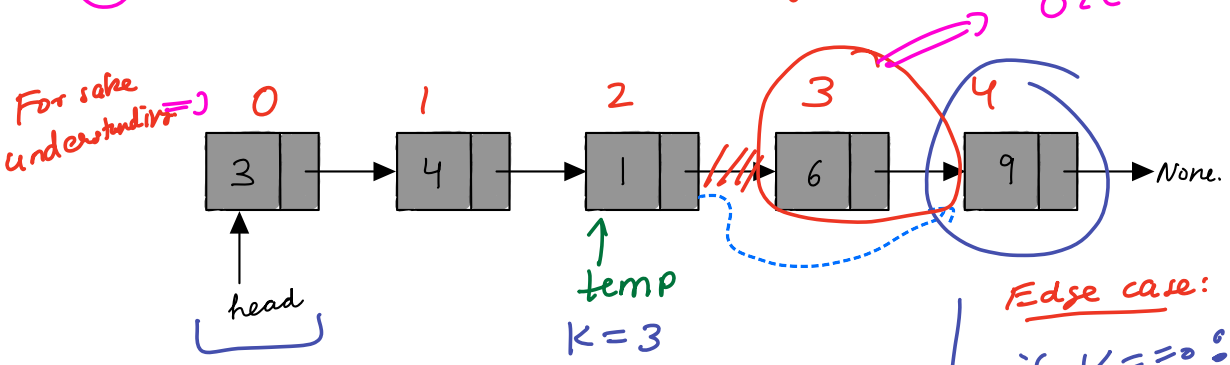
Q) Can we optimize using a reference to 2nd last node?

How do we go back? No way home



Singly Linked list ⇒ can't optimize deletion at end.

(c) Delete at pos K (starting with 0)



Step - 1 temp = FindKthNode(head, K-1)

Step - 2 Update temp.next = temp.next.next

Edge case:

if K == 0 :

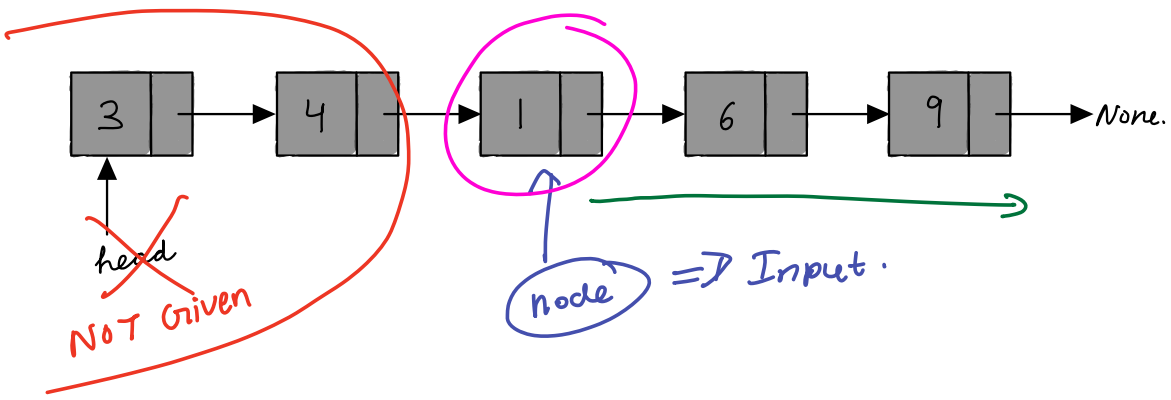
head = delete_at_head (head)

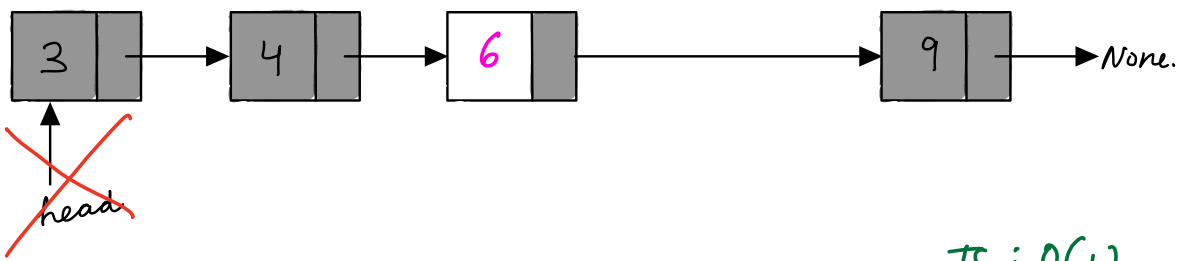
HW: Try writing code. ✓

TC: O(N) , SC: O(1)

Q 1 Delete the given node (SLL)

(can't be the last node) Hint

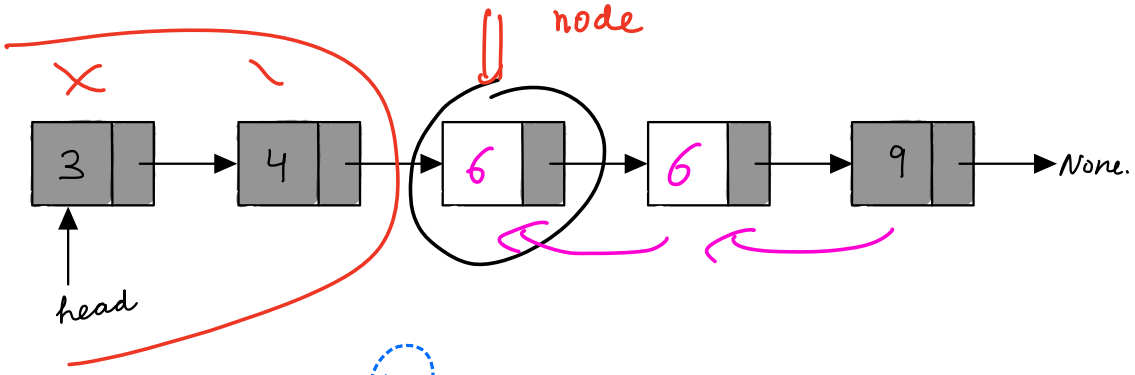


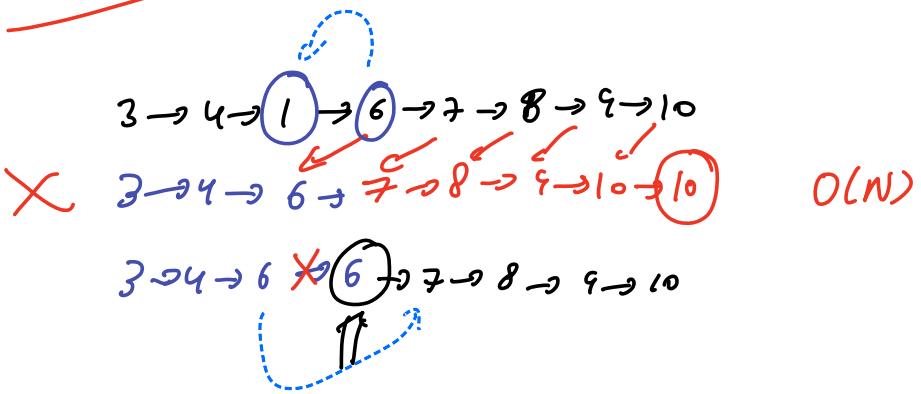


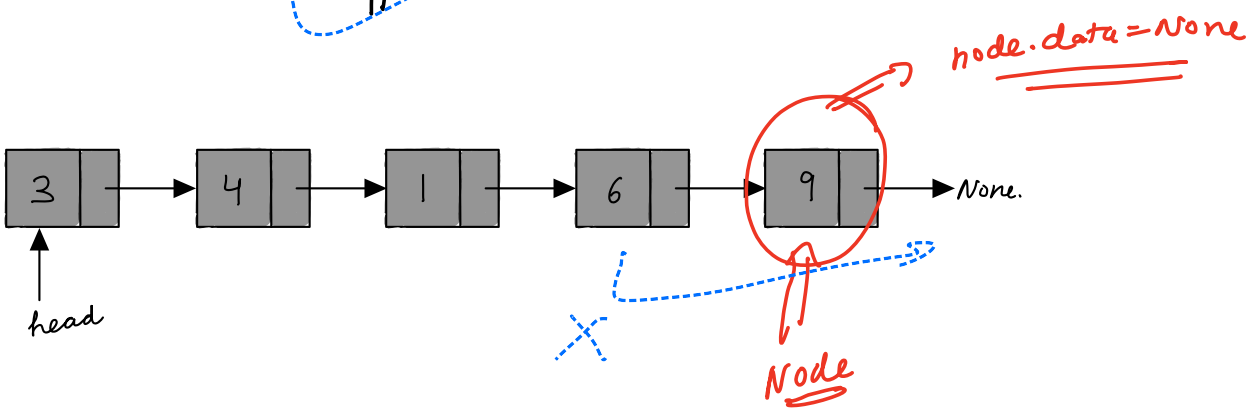
TC : O(1)

SC : O(1)

```
def delete_node(node):
    node.data = node.next.data
    node.next = node.next.next
```


node
3
4
6
6
9
None.
head


3 → 4 → 1 → 6 → 7 → 8 → 9 → 10
3 → 4 → 6 → 7 → 8 → 9 → 10 → 10
O(N)
3 → 4 → 6 → 6 → 7 → 8 → 9 → 10


node.data = None
3
4
1
6
9
None.
head
Node

Intro to Stacks

Types
1. Identify that you need stack/queue
2. Given stack/queue.

Last In First Out.

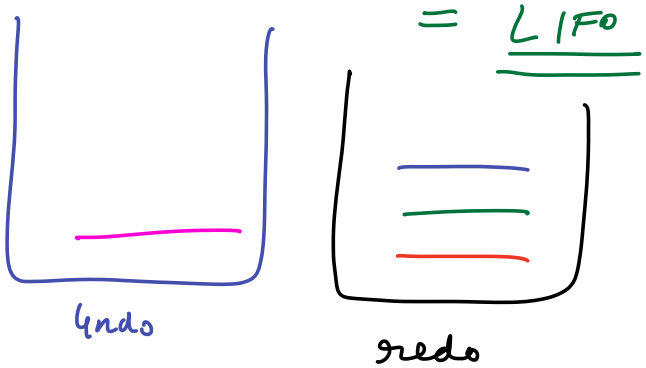
- Books
- Chairs
- Plates

- Recursion / Call Stack
- Undo/Redo
- Backward/Forward

Q) Why do we need stack where we can only insert on the top?

Applications.

Behavior = LIFO



Q) What is Stack? Is it a data structure?

Not a DS.

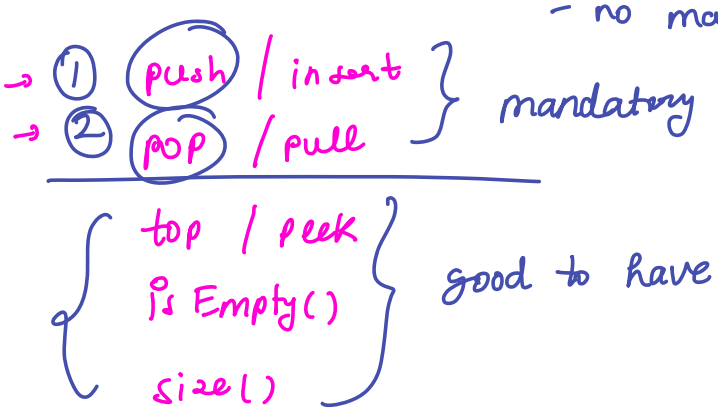
Abstract Data Type => 1) Push
2) Pop

ADT

It is actually an Abstract Data Type with foll.

operations

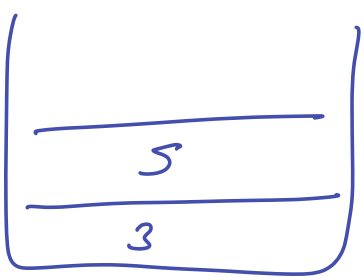
- operations specified
- no matter how you implement



⑧ Implementation of Stacks => Not a DS.

ⓐ Using Arrays

Q) How to implement with fixed size array constraint?



✓ push(3)
push(5)
✓ pop() => 5

Using Python list

l = [ ]

[3]
[3, 5]
[3]

push(x) => l.append(x) O(1)
pop() => l.pop() O(1)
top() => l[-1]

Dynamic Array.

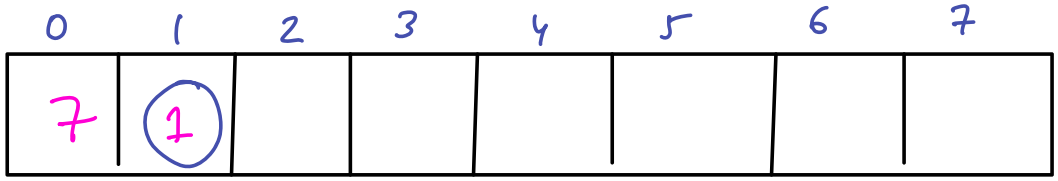
still has a max size

Adv.

Doubling concept.

Fixed size array.

size 8



top = -1 ⇒ last idx where elt was inserted

LIFO

top

- push(5)
- pop()
- push(7)
- push(8)
- push(9)
- pop()
- pop()
- push(1)
- push(2)
- pop()
- top()

7 8 ✗ ↑ top_index.

if top == size-1: raise OverFlow

```
push(x):
    top += 1
    A[top] = x
```

```
pop():
    data = A[top]
    top -= 1
    return data
```

if top == -1: raise Underflow

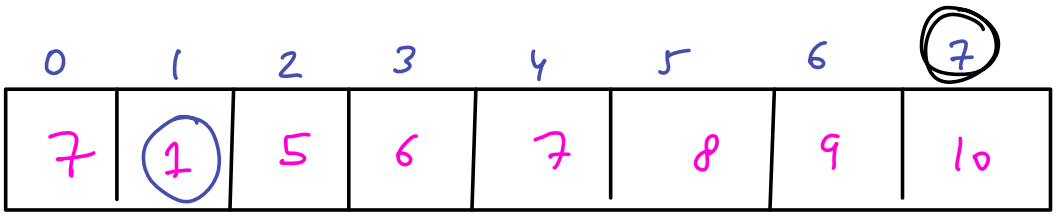
check underflow

```
top():
    return A[top]
```

```
isEmpty():
    return (top == -1)
```

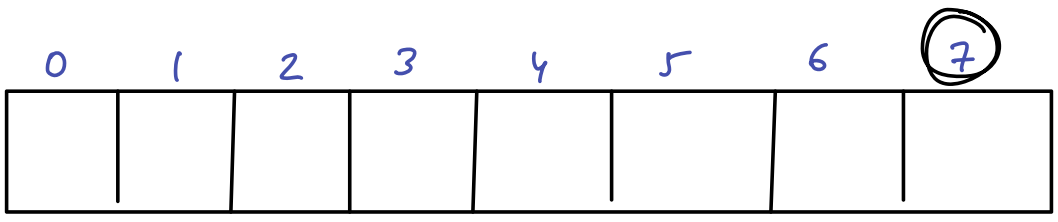
```
size():
    return (top+1)
```

N=8



top

push(19) ⟶ Stack overflow



top = -1

pop() Underflow

Disadvantages:

- We have a fixed size
- Not dynamic, static
- Contiguous memory blocks needed

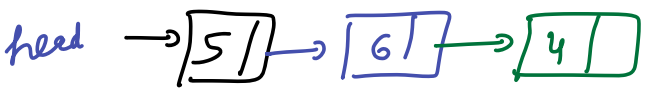
LL comes to rescue.

B) Using LL

Q) Where to add/remove elements — at what ends?

Head? Tail?

We want to achieve O(1) time complexity.



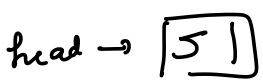
✗ Insert at Tail ⇒ O(1)
Remove at Tail ⇒ O(N)

Inserting at end

Tail

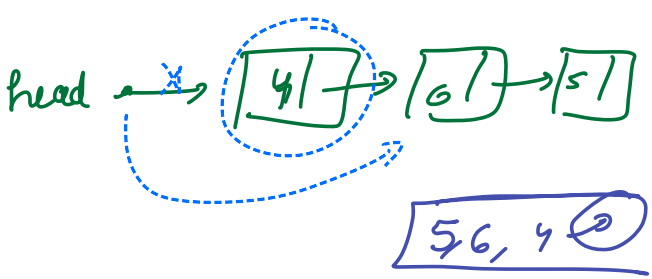
① push(5)
② push(6)
③ push(4)

Front/Head = Top



Insertion at Head ⇒ O(1)

Remove at Head ⇒ O(1)



HW: Try doing this implementation.

Total SC to store N elements in stack

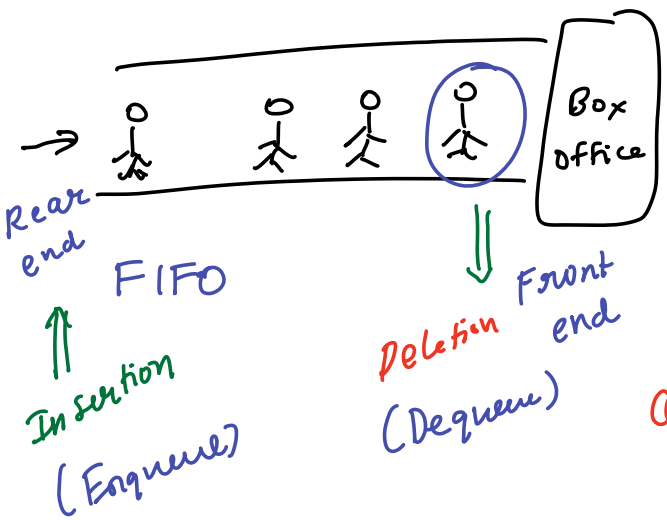
⇒ O(N)

Printer Queue

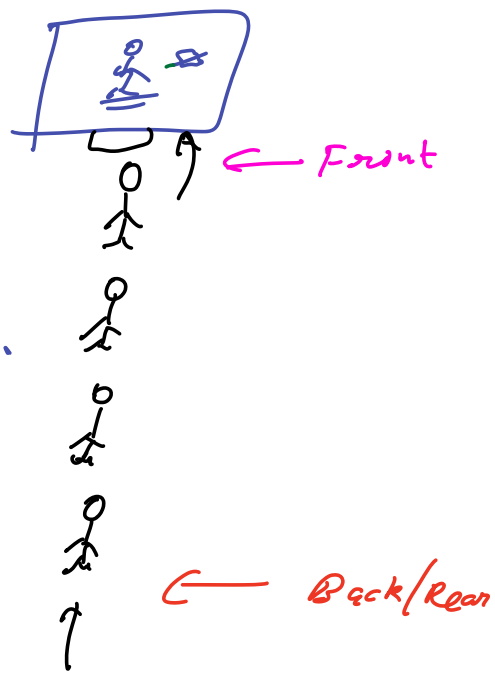
FIFO First In First Out

Doc1 (Dhiman) => 1
Doc1 (Ashwanth) => 2
PDF (Avijit) => 3
Doc2 (Jagrut) => 4

Queue
- Enqueue (Push/Add)
- Dequeue (Roll/Del)

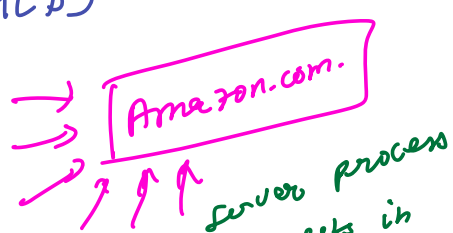


Vaccination



Q) What are some programming applications?

- Kafka/RabbitMQ) ActiveMQ (HLD)
- Scheduling
  - Process
  - Printer
  - OS
  - Torrent Download Queue
    DC++



Queue – Basics

(A) Using Array

(Q) How to implement with fixed size array constraint?

Which side will you enqueue/dequeue?

Front? Rear?

→ Circular Queues. Advance Batch.

Front 5, 7, 9 Rear (1)

List

l = [ ]

[5]

[5, 7]

[5, 7, 9]

[7, 9]

✓ enqueue(5) → l.append(val) TC: O(1)

✓ enqueue(7)

✓ enqueue(9)

dequeue → l.pop(0)

TC: O(N)

---

l = [ ] (2) 9, 7, 5 (Rear … Front)

[5]

[7, 5]

[9, 7, 5]

[10, 9, 7, 5]

enqueue(x) → l.insert(0)

TC: O(N)

dequeue → l.pop()

TC: O(1)

B) Using LL

a) Where to add/remove elements — at what ends?

Head? Tail?

We want to achieve O(1) time complexity.

=O(1)

Insert at Head

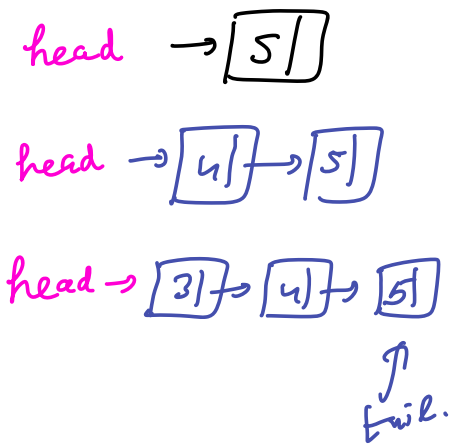
Remove at Tail/End

=O(N)

① enq(5)

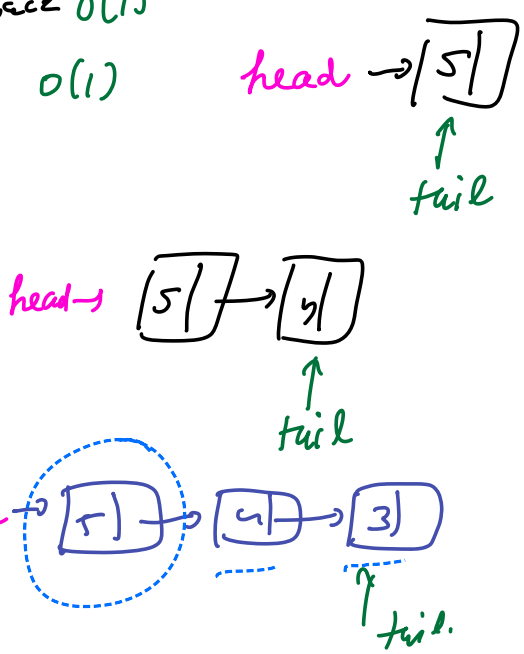
② enq(4)

③ enq(3)



Queue

- insert at rear / back O(1)
- delete from front O(1)



Queues & Stacks in Python

Enqueue Dequeue

Stacks : `l = []`

`l.append(5)` O(1)

`l.pop()` O(1)

Queues

`queue.Queue` Reference.

It is actually an Abstract Data Type with foll.

- operations specified
- no matter how you implement

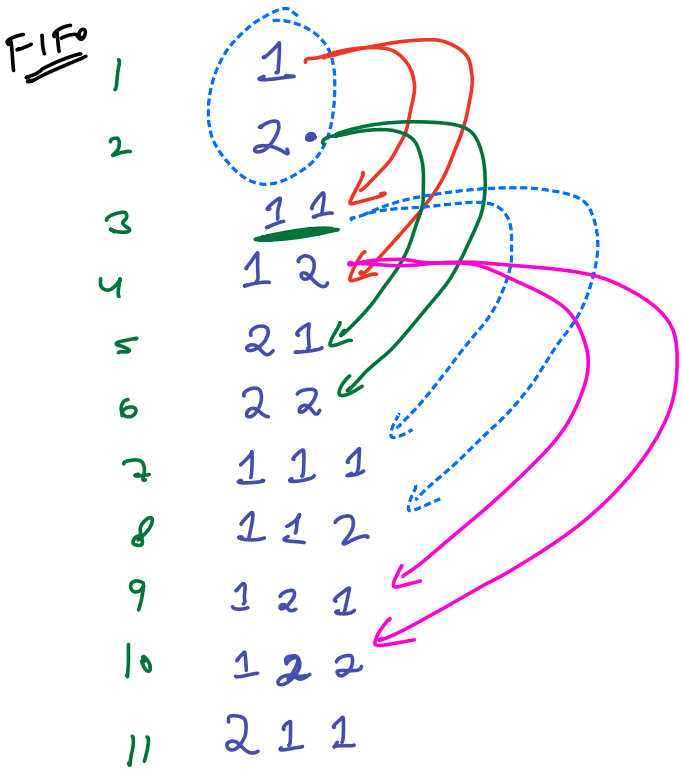
Stack	Queue	
operations		
push	enqueue / insert	mandatory
pop	dequeue / pull	mandatory
top	front / peek	good to have
~	isEmpty()	good to have
~	size()	good to have

O(1)

Idea ?

Amazon Q) Find Nth number in this pattern containing digits 1, 2 ~~3~~.

FIFO



1 2

[~~1~~, 2] 3 4

[~~2~~, 11, 12] 5 6

[11, 12, 21, 22]

1*10+1 = 11

1*10+2 = 12

2*10+1 = 21

2*10+2 = 22

Q2) Reverse a given queue. (enqueue dequeue) can use extra space

Google

front → Rear.

$q = 2, 1, 3, 4, 5, 7, 9, 11, 8$

↓

$q' = 8, 11, 9, 7, 5, 4, 3, 1, 2$

Approach-1 ✗

Dequeue from q and insert into another

Insertion ← Rear

$q_2 = 2, 1, 3, 4, 5, 7, 9, 11, 8$

FIFO

Approach-2

Dequeue from q and add to list

List not queue → $l = [2, 1, 3, 4, 5, 7, 9, 11, (8)]$

iterate from reverse. ← enqueue.

TC: O(N)

SC: O(N)

Use original

← $q = 8, 11, 9, 7, 5, 4, 3, 1, 2$

Approach-3 ① Dequeue and keep pushing into a stack

front → Rear

$q = \cancel{2}, \cancel{1}, \cancel{3}, \cancel{4}, \cancel{5}, \cancel{7}, \cancel{9}, \cancel{11}, 8$

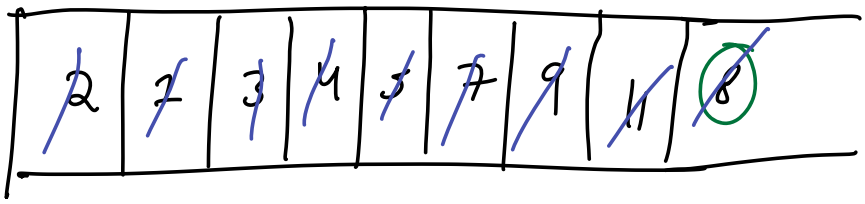
Stack

2	1	3	4	5	7	9	11	(8)

- TC: O(N)
- SC: O(N)

In stack

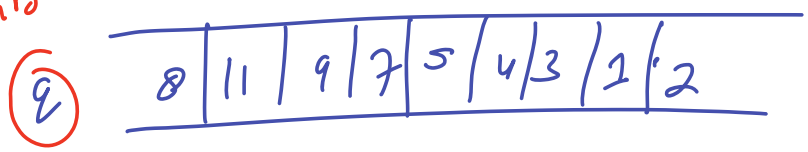
② Pop from stack & enqueue to que one by one



TC: O(N)
SC: O(N)

existing queue.

Existing queue.



← Insert

TC: $2N = O(N)$

SC: $N+N = O(N)$
