Ruchika PSA Class 3

Data Structure
- learn how to use it
- Learn how its built (internally)

Python List interview questions
- heterogenous container?
- [ ] representation size?
- append → inserting elements?
- access a[i] in $\Theta(1)$ time!

In C++ & Java:

- `int[] a = int[20];`

  static allocation (specify size)

  hence we can now access & set

  Ex: `a[3] = 40`

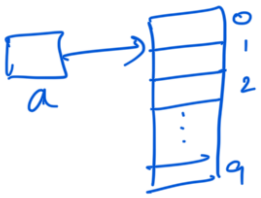
list container of size 20 elements of data type int

- `int * a = new int(n);`

  dynamic allocation (size is not specified at compile time)

  But, if we have size = 11, its fixed not. size = 10 (∵ n = 10)

At runtime, n = 10



Implementing python list as a dynamic growable array

In Python:

`a = [ ]` (Typicall ∞ size) that's why we never specify the size like in C++ & Java

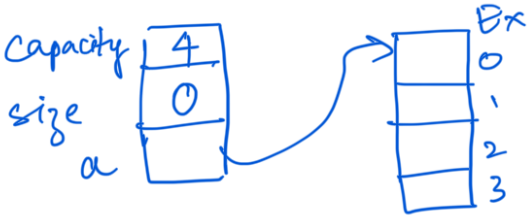
`a[i]` in $\Theta(1)$ time

∴ Python list is a dynamic array

How python list is built (under the hood)?

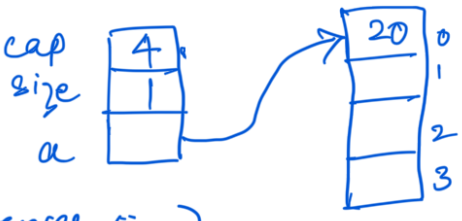
`a = [ ]`

Internally using C++, continuous memory is allocated of a



2 powered number size

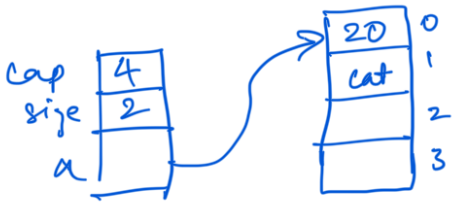
a.append(20) in $\Theta(1)$ time



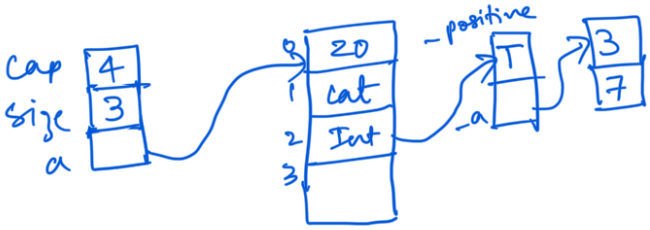
a.length = 1 (references size)

in $\Theta(1)$ time

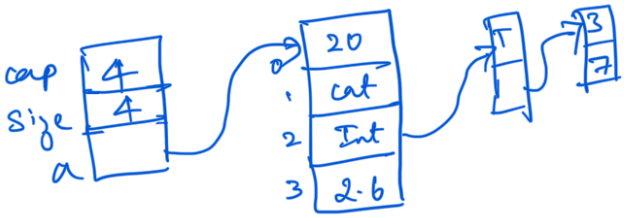
a.append("cat")



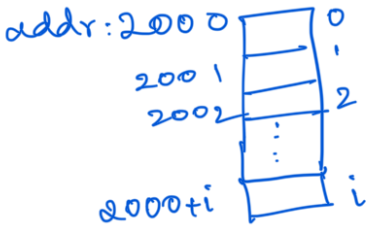
a.append(Int(37))



a.append(2.6)



Continuous Memory:



a[i]: a+i (interms of address)

It's $\Theta(1)$ time since we know a[i] is at a+i address location. here 2000+i since its continuous memory allocated.

Now, Inserting element at the end of the list:

a.append(40)

Table doubling algorithm

We know that there's no more space, capacity == size == n

If we had an algo, whenever we wanted new space, we would have to find another continuous memory block of size n+1, then copy all the

elements → this is inefficient, so we ...

(1) Table Doubling method:

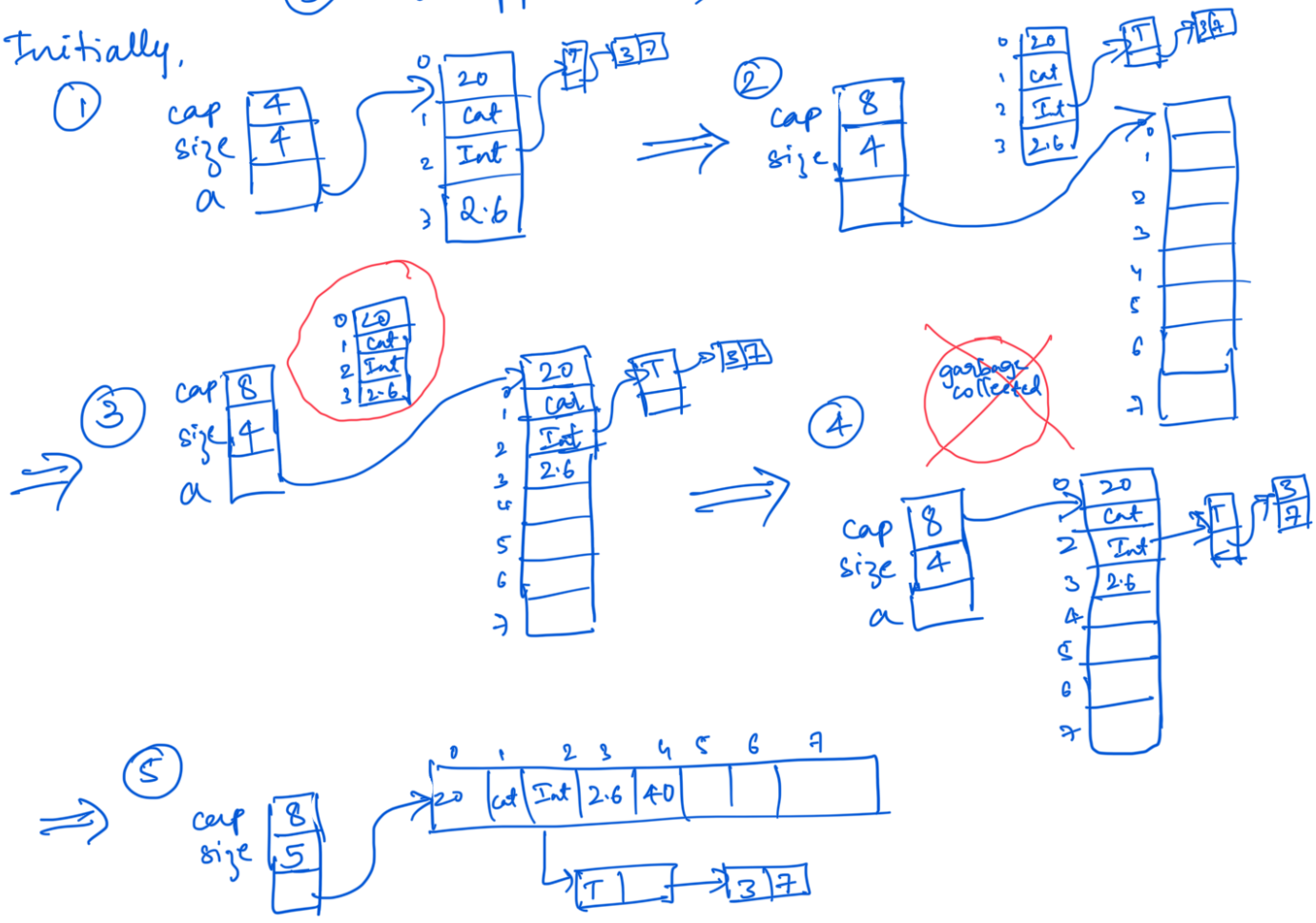
(2) • Say we initially had 2^n size allocated, now we'll introduce another block of $2^{n+1} = (2^n)(2)$ size doubled

(3) • All elements are then shallow copied

(4) • Previous array is taken away by garbage collector memory

(5) a.append(40)

Initially,



• Next time, when 8 spaces are filled, a new continuous block of memory of size 16 is given when we want to append the 9th element of the list

• The previous memory's content is copied, 9th element is appended & previous block of memory is garbage

collected → automatically in Python

Amortized cost

Table Doubling grows 2^n time

$2 \rightarrow 4 \rightarrow 8 \rightarrow 16 \rightarrow \ldots\ldots \rightarrow 1024$ (10 times)

growth: 2 times, 3 times, 4 times

			Time	Space
complexity	Usual	size<=cap	$\Theta(1)$	$\Theta(1)$
	Grow	size>cap	$\Theta(n)$	$\Theta(n)$

∴ Time Complexity is $\Theta(1)$ Amortized cost

O(1)

✓ → good operations

Python List Operations:

		Time	Space
✓ Initializing a list	a[]	O(1)	O(1)
✓ Accessing element	a[i]	O(1)	O(1)
✓ Inserting element at end of list	a.append(x)	$\Theta(1)$ amortized	$\Theta(1)$ amortized
• Inserting element at head of list	a.prepend(x)	$\Theta(n)$	$\Theta(n)$
• Inserting element at any position	a.insert(x,i)	O(n)	O(n)
✓ Length of the list	a.length	$\Theta(1)$	$\Theta(1)$
• Finding an element	a.find(x)	O(n)	O(1)
• Deleting element	a.del(0)	$\Theta(n)$	$\Theta(1)$

(during growth max complexity for both: $O(n)$)

(since we need to copy all elements into new memory block)

(a.insert(x,i): x = element, i = position)

(depends on the luck
head = $\Theta(1)$, grow $O(n)$,
center $O(n)$ move n elements)

(depends on the luck,
first → O(1), nth element → O(n))

at head of list

(after deleting at head, we need to move all the elements into a new memory block)

✓ • Deleting element at end of list — a.del(n) — $\Theta(1)$ — $\Theta(1)$

(we just delete last element & decrease size of array)

• Deleting element at any position — a.del(i) — $O(n)$ — $O(n)$

(depends on luck, deleting last → $O(1)$; deleting 1st position, we need to move elements by coping)

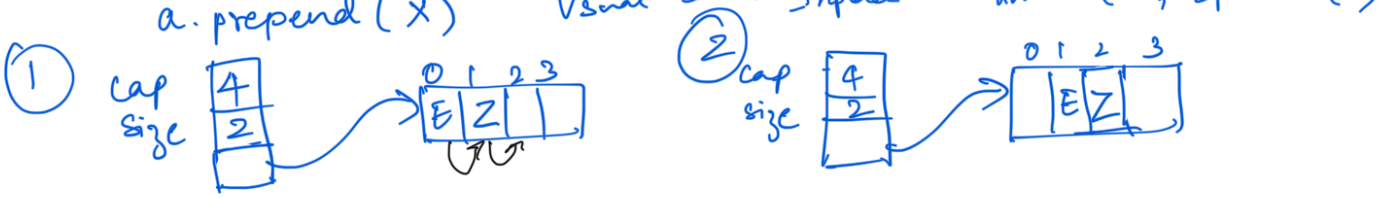
Why don't we perform opposite of table doubling, to take away half the space while deleting?

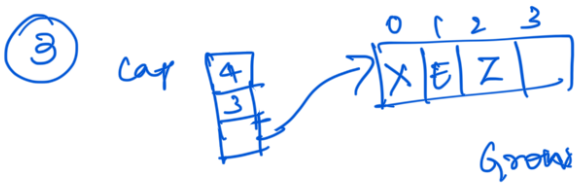
It's harder to get continuous memory again

(There might be a way in Python lib to give away half the space when size = 1 billion)

• Inserting an element at the head of the list:

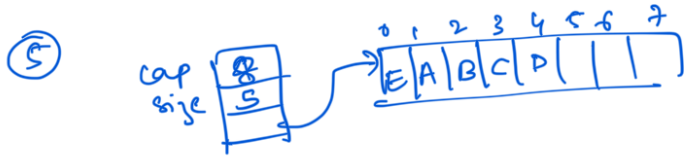
a.prepend(X) Usual case: Inplace Time $O(n)$, Space $O(1)$





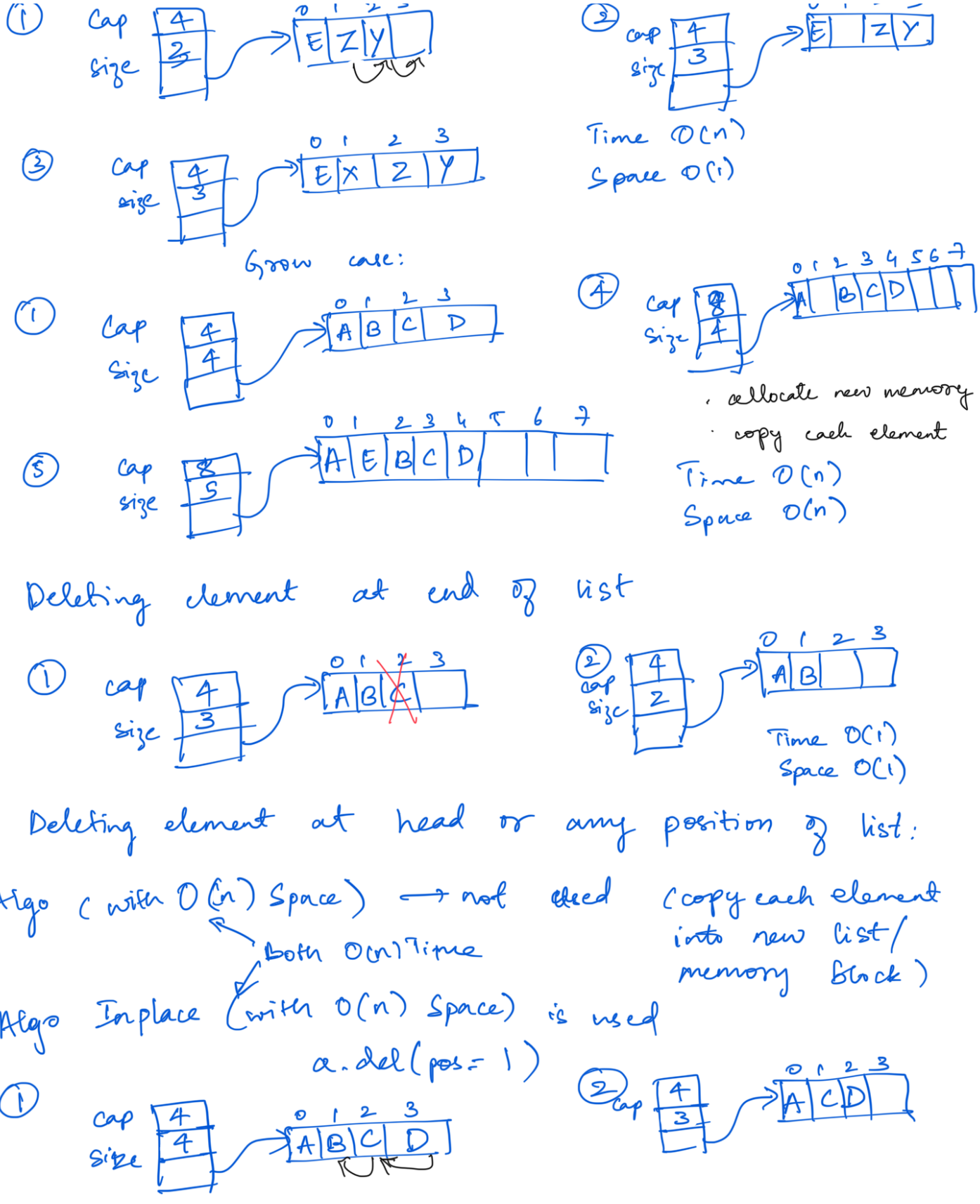
Grows case: Time $O(n)$, Space $O(n)$

. allocate new memory
. copy each element



• Inserting element at pos (i) a.insert(X, i)

Usual case: Inplace a.insert(X, 1)



Conclusion: good operations of Python List:

- ∞ size (dynamic) → capacity
- a[i] in Θ(1) time → access element
- Inserting to end of list → append

$\Theta(1)$ amortized time

- Deleting from end of list → remove
  $\Theta(1)$ time

Stack

Stack : LIFO - Last In First Out

Ex: Stacking plates

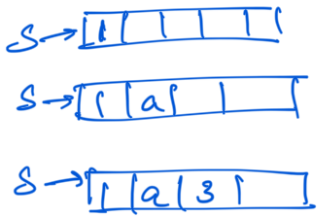
- isFull → . push
- is Empty → . pop , . top

We can see only the top element, not any element in between

All ops → $\Theta(1)$ time, $\Theta(1)$ space

Since we use python list, we don't need to check condition isFull() before push() since Python list has dynamic size (∞), its always False

s.push(1)
s.push("a")
s.push(3)

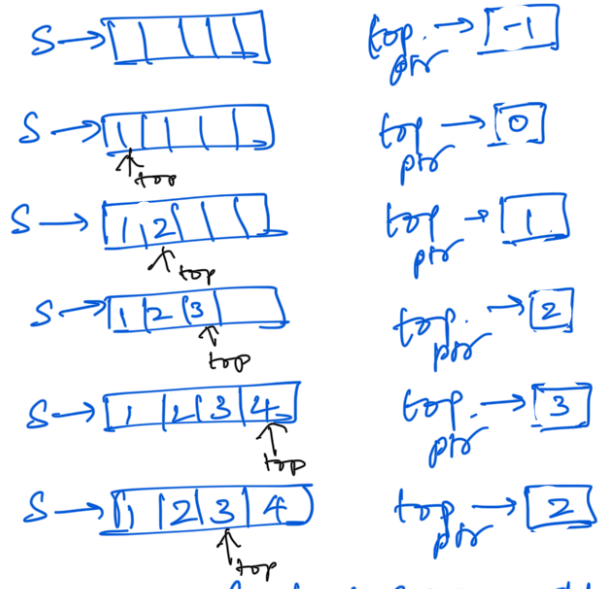


Time $\Theta(1)$
space $\Theta(1)$

We can remove the last element using python list, but we want to just shift the end pointer

Ex:

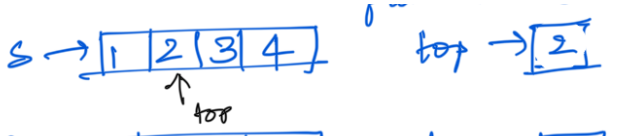
```
S = Stack()
S.push(1)
S.push(2)
S.push(3)
S.push(4)
S.pop()
```



space $\Theta(1)$
time $\Theta(1)$

- just move top ptr, no need to remove it from the list

s.pop()



s.push(6)

since we know that top ptr is at 2, inc top ptr & replace at top

Say top == -1, if we perform pop() then, don't remove since there are no elements in the stack

Ex:

∴ isEmpty() should check if top == -1

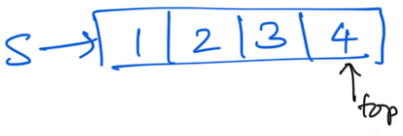
∴ size() should return top + 1

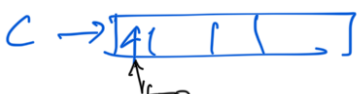
In the above case there were max no. of elements = 4, since we are not removing the elements from the list, but rather just moving the top ptr, len(list) will give us the max space that was used

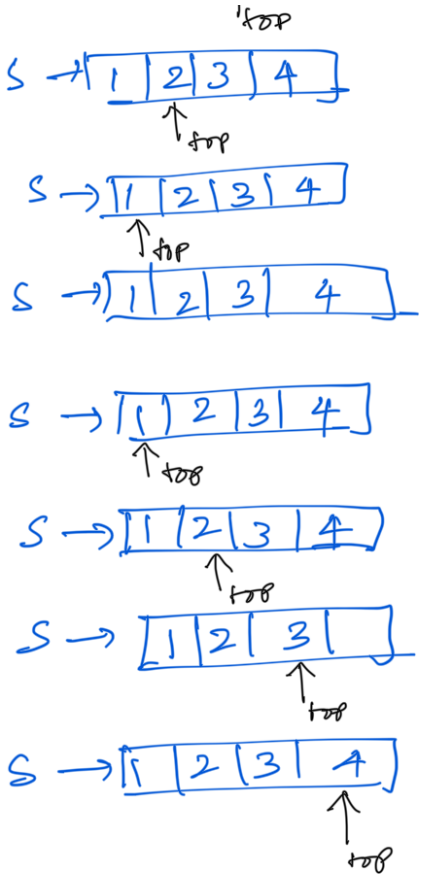
∴ space() should return s.length

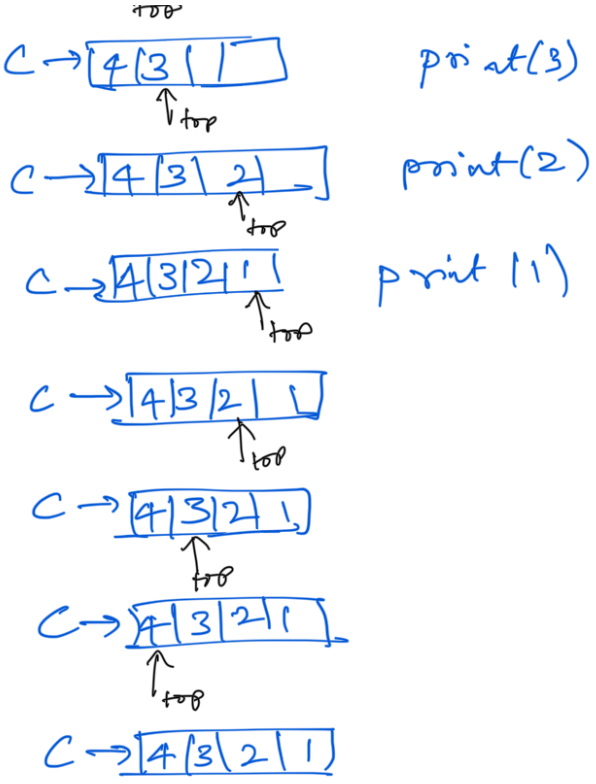
Printing a stack: Time $\theta(n)$ Space $\theta(n)$

Since we can access only the top element, we would need to pop all the elements to another stack & then print the elements & then push it back to the original stack





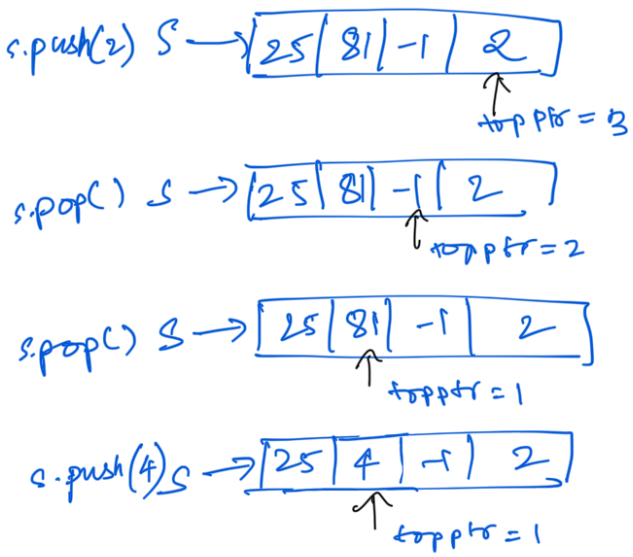


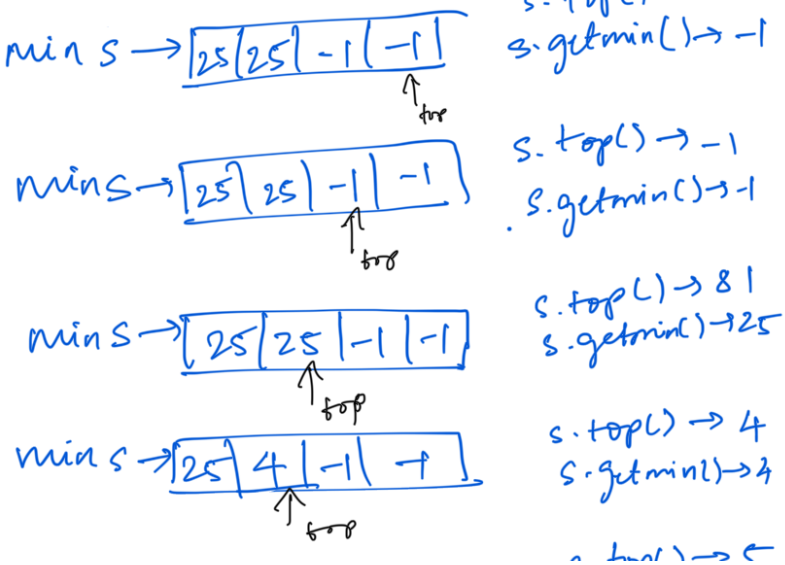


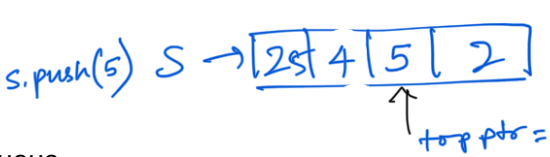
	Python list	Min Stack
Find min element	Time $\theta(n)$ Space $\theta(1)$	Time $\theta(1)$ Space $\theta(1)$
Find max element	Time $\theta(n)$ Space $\theta(1)$	Time $\theta(1)$ Space $\theta(1)$
	Python list	Max Stack

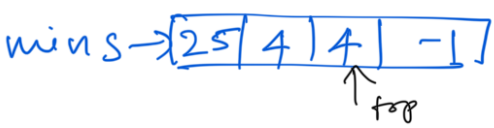
Min/Max Stack

Min Stack → with how many ever elements inserted into the stack, in $\theta(1)$ time find the min ele present in the stack









s.getmin() → 4

Queue

Queue : FIFO - First In First Out Ex: carwash

all methods use $\Theta(1)$ Time & $\Theta(1)$ Space ideally

- isFull() → enque(T) back()
- isEmpt() → deque() front()

Since we are using Python list to implement the Queue, it has ∞ dynamic size, so isFull() is always False, and maybe not required

If we implement with a single Python list,
enqueue will take $\Theta(1)$ time, but
dequeue will take $\Theta(n)$ time since
removing from head of list is $\Theta(n)$ time

Circular Queue

Circular Queue - FIFO

If we restrict size of the queue, we can create methods for both enqueue & dequeue in $\Theta(1)$ time & $\Theta(1)$ space

Ex: max space = 4

q = Queue() len → 0 q →

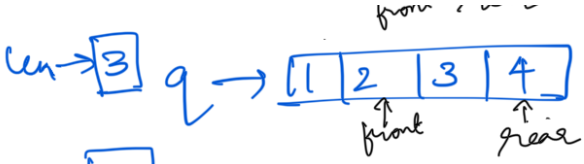
q.enqueue(1) len → 1 q → 1 (front, rear)

q.enqueue(2) len → 2 q → 1 2 (front, rear)

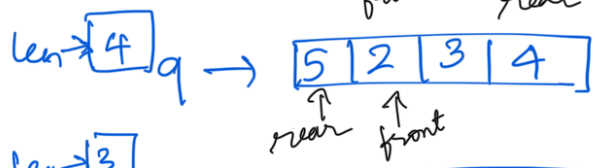
q.enqueue(3) len → 3 q → 1 2 3 (front, rear)

q.dequeue() len → 2 q → 1 2 3 (front, rear)

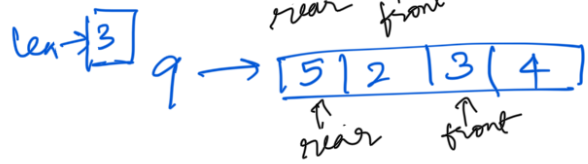
q.enqueue(4)



q.enqueue(5)



q.dequeue()



Implement Stack given interface of Queue

only ops that can be used:

enqueue, dequeue, isFull, isEmpty, Front, Back

Implement Queue given Interface of Stack

only ops that can be used:

push, pop, isFull, isEmpty, top

When we say using interface of any other DS, ex: Stack using List, Stack using Queue, we are using Design Pattern: "Adapter Pattern"

Ex: we have a device from India, but we want to use it in US, we can use an adapter to connect US socket & the India device.

similarly, we can implement adapters in our code to implement other DS.

Suppose we implement Queue Using Stack & we want to now call it MyQueue say in leetcode, we can use this section of code:

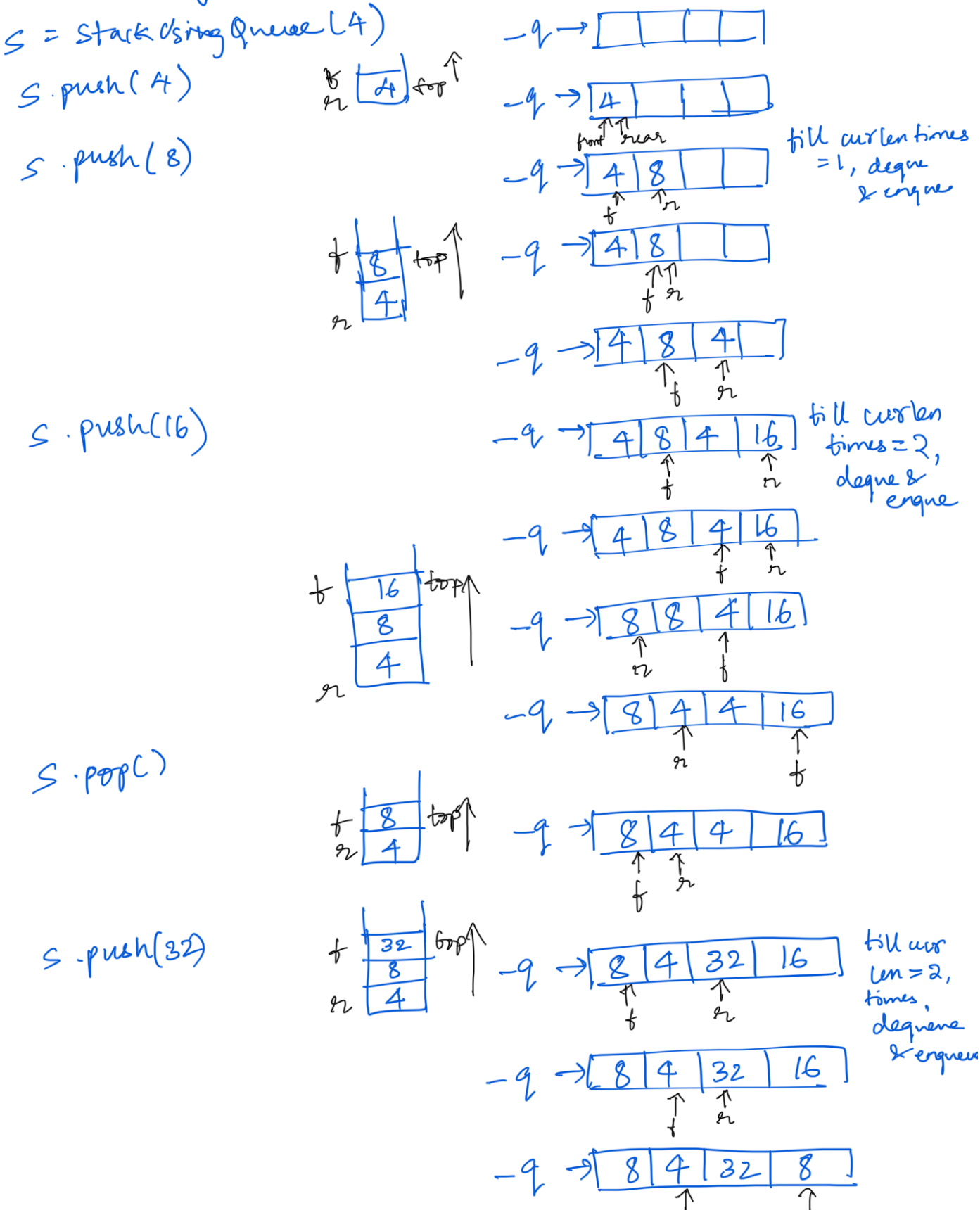
```
MyQueue = QueueUsingStack
```

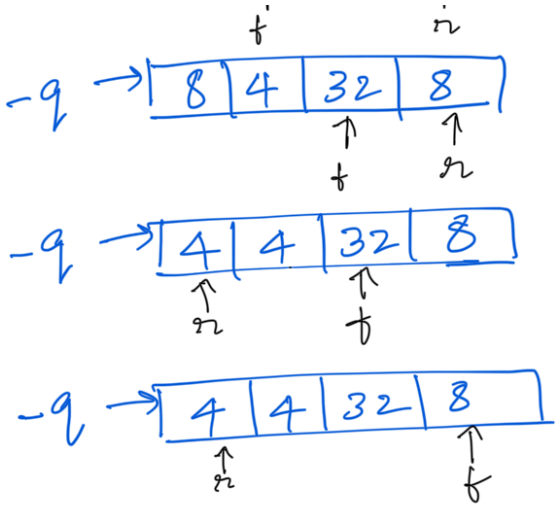
Since this is a shallow pointer copy, we can now invoke interfaces with MyQueue class

Stack using queue

Stack Using circular Queue : // limited capacity

S = StackUsingQueue(4)



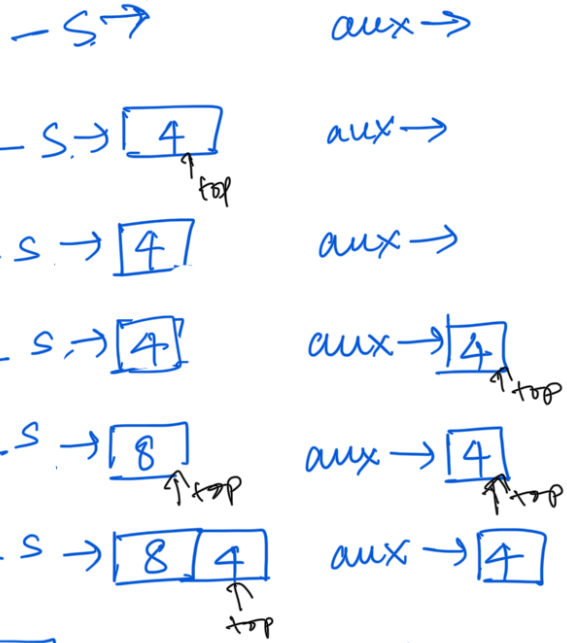


s.pop()

Queue using stack

Queue using Stack

q = QueueUsingStack()



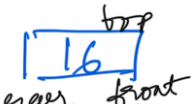
q.push(4)

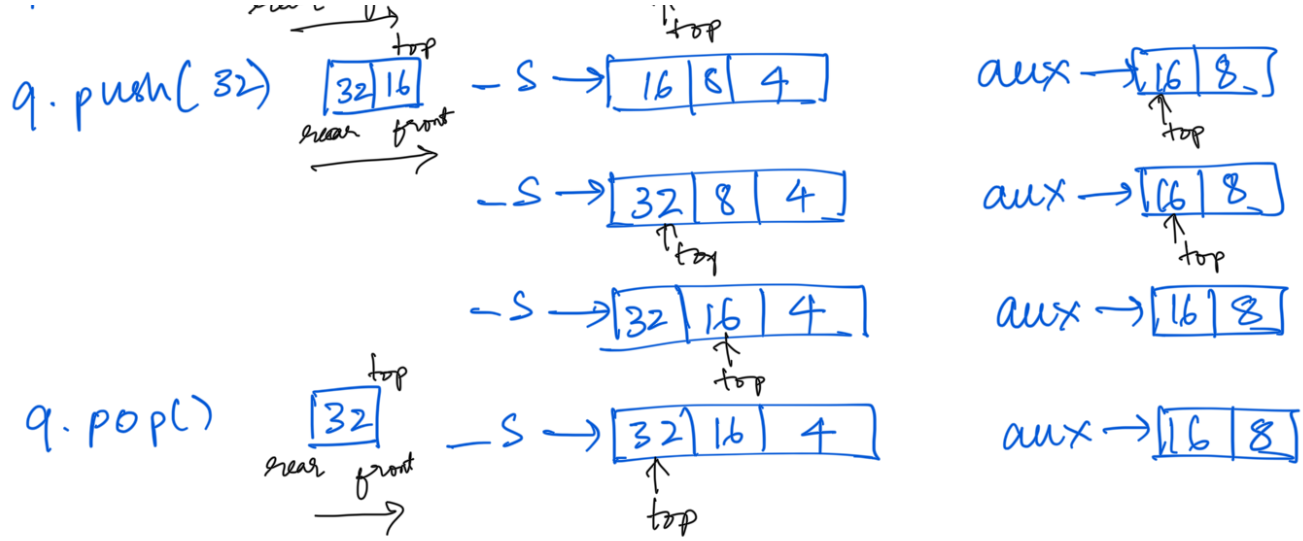
q.push(8)

q.push(16)

q.pop()

q.pop()




q.push(32)
32 16
top
rear
front
S
16 8 4
top
aux
16 8
top
S
32 8 4
top
aux
16 8
top
S
32 16 4
top
aux
16 8
q.pop()
32
top
rear
front
S
32 16 4
top
aux
16 8

Time Complexity, Space Complexity	Accessing front / top element	Accessing rear element	Accessing center element	Inserting at front / top position	Inserting at rear position	Inserting at any position	Length	Finding any element	Finding min element	Finding max element	Deleting from front / top position	Deleting from rear position	Deleting from any position	Printing DS (__str__)
Python List	Theta (1), Theta (1)	Theta (1), Theta (1)	Theta (1), Theta (1)	Theta (n), Amortized O (1)	Amortized O (1), Amortized O (1)	O (n), Amortized O (1)	Theta (1), Theta (1)	O (n), Theta (1)	O (n), Theta (1)	O (n), Theta (1)	Theta (n), Theta (1)	Theta (1), Theta (1)	O (n), Theta (1)	Theta (n), Theta (1)
Stack using Python List	Theta (1), Theta (1)	Theta (n), Theta (n)	O (n), O (n)	Theta (1), Amortized O (1)	Theta (n), Theta (n)	O (n), O (n)	Theta (1), Theta (1)	O (n), O (n)	O (n), O (n)	O (n), O (n)	Theta (1), Theta (1)	Theta (n), Theta (n)	O (n), O (n)	Theta (n), Theta (n)
Queue	Theta (1), Theta (1)	Theta (1), Theta (1)	O (n), Theta (1)	Theta (n), Theta (1)	Theta (1), Theta (1)	O (n), Theta (1)	Theta (1), Theta (1)	O (n), Theta (1)	O (n), Theta (1)	O (n), Theta (1)	Theta (1), Theta (1)	Theta (n), Theta (1)	O (n), Theta (1)	Theta (n), Theta (1)
Circular Queue using Python List	Theta (1), Theta (1)	Theta (1), Theta (1)	O (n), Theta (1)	Theta (n), Theta (1)	Theta (1), Theta (1)	O (n), Theta (1)	Theta (1), Theta (1)	O (n), Theta (1)	O (n), Theta (1)	O (n), Theta (1)	Theta (1), Theta (1)	Theta (n), Theta (1)	O (n), Theta (1)	Theta (n), Theta (1)
Min Stack using Stack	Theta (1), Theta (1)	Theta (n), Theta (n)	O (n), O (n)	Theta (1), Theta (1)	Theta (n), Theta (n)	O (n), O (n)	Theta (1), Theta (1)	O (n), O (n)	Theta (1), Theta (1)	Theta (n), Theta (n)	Theta (1), Theta (1)	Theta (n), Theta (n)	O (n), O (n)	Theta (n), Theta (n)
Max Stack using Stack	Theta (1), Theta (1)	Theta (n), Theta (n)	O (n), O (n)	Theta (1), Theta (1)	Theta (n), Theta (n)	O (n), O (n)	Theta (1), Theta (1)	O (n), O (n)	Theta (n), Theta (n)	Theta (1), Theta (1)	Theta (1), Theta (1)	Theta (n), Theta (n)	O (n), O (n)	Theta (n), Theta (n)
Stack using Circular Queue	Theta (1), Theta (1)	Theta (1), Theta (1)	O (n), Theta (1)	Theta (n), Theta (1)	Theta (1), Theta (1)	O (n), Theta (1)	Theta (1), Theta (1)	O (n), Theta (1)	O (n), Theta (1)	O (n), Theta (1)	Theta (1), Theta (1)	Theta (1), Theta (1)	O (n), Theta (1)	Theta (n), Theta (1)
Queue using Stack	Theta (1), Theta (1)	Theta (n), Theta (n)	O (n), O (n)	Theta (1), Theta (1)	Theta (n), Theta (n)	O (n), O (n)	Theta (1), Theta (1)	O (n), O (n)	O (n), O (n)	O (n), O (n)	Theta (1), Theta (1)	Theta (n), Theta (n)	O (n), O (n)	Theta (n), Theta (n)