Stacks

- The order in which the data arrives is important.
- It is an ordered list in which insertion and deletion are done at one end called top.
- It follows LIFO principle or FILO (first In Last Out)
- when an element is inserted - push
- " " " " deleted - pop
- Underflow - Trying to pop out an empty stack
- Overflow - Trying to push an element in a full stack.

These two are exceptions.

Main stack operations

1) push : Inserts data onto stack
2) pop : Removes & returns the last inserted element from the stack.
3) Top : Returns the last inserted element without removing it.
4) Size : Returns no. of elements.
5) IsEmptyStack
6) IsFullStack.

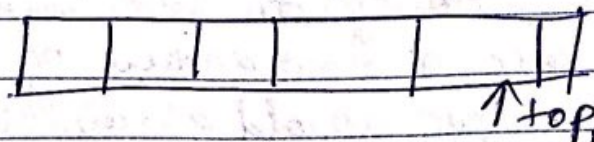
Rec

→ To insert the element we increment top index and then place the new element at that index.
To delete, we take the element at top index & decrement the top index. empty stack has top = -1

Stack Implementation :

1) Simple Array based implementation
2) Dynamic Array — " — " —
3) Linked List implementation

(1) Simple Array based implementation



- Add elements from left to right
- Use a variable (top) to keep track of the index of the top element.

Performance :

1) SC for n push operations : O(n)
2) TC of push() : O(1)
3) TC of pop() : O(1)
4) TC of size() : O(1)
5) TC of isEmpty() : O(1)
6) TC of isFullStack() : O(1)
7) TC of deleteStack() : O(1)

Limitation :

- Max size of stack must be defined & cannot be changed.

2) Dynamic Array Implementation.

Issues with simple array base impl.

a) Size cannot be ↑ed.

If we want to ↑ the array size we can ↑ the array size by 1 every time we want to push an element.

i. at $n=1$, create a new array of size 2 & copy all old array elm to new & at the end add new elm arr.

at $n=2$, size 3.

at $n=n-1$, create array of size n & copy & paste all elem of old array to new arr & add new elem at end.

∴ for n push operations,

total time $T(n) = 1+2+3+\cdots+n$

$\approx O(n^2)$

Alternative approach (Repeated doubling).

Double the array size once the array is full

∴ Initially lets start with

$n=1 \Rightarrow$ new_array $n=2$

$n=2 \Rightarrow n=4$.

$n=4 \Rightarrow 8$.

$n = 16 \Rightarrow$ new array size 32.

$\therefore$ which is nothing by $2n$.

$\therefore$ for an array of size n, we will need to create a new array of size $2n$.

We are doing the doubling operation $\boxed{\log n}$ times

i.e., For n push operations,

1) we double the array size $\log n$ times.
2) Total time of n push operations

$$T(n) = 1 + 2 + 4 + 8 + 16 + \cdots + \frac{n}{4} + \frac{n}{2} + n$$

$$= n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \cdots + 4 + 2 + 1$$

$$= n\left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots + \frac{4}{n} + \frac{2}{n} + \frac{1}{n}\right)$$

$$= n(2) \approx 2(n) = O(n)$$

$\therefore \boxed{T(n) = O(n)}$

$\log n$ times, $\because$

By the time we reach $n = 32$,

total number of Copy operation $= 1 + 2 + 4 + 8 + 16$

$= 31$

which is $\approx 2(n) = 32$

$\therefore \quad n = 2n$

$\log n = \log(2n)$

$\log n = \log 2 + \log n$

total no. of copy operations $\approx (\log n)$ times.

~~$\log n$~~

Performance

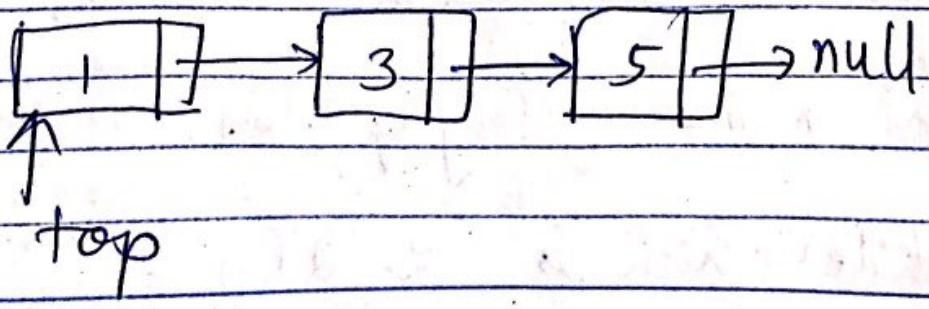
- SC for n push operations $= O(n)$
- TC for all ~~other~~ operations $\approx O(1)$

Limitations

- Too many doublings may cause memory overflow exception.

3) Linked List Implementation :

- Push operation is implemented by inserting element at the beginning of the list
- Pop is done by deleting the node from beginning (header/top node)



Performance

SC for n push operations : $O(n)$
TC for other operations $\approx O(1)$
TC for ~~Pop~~ for deleteStack() $\approx O(n)$.

Comparissions

1) Incremental vs doubling strategy.

The amortized time of a push operation is the average time taken by a push over the series of operations
i.e., $T(n)/n$.

a) for Incremental : $O(n^2)/n \approx O(n)$
b) Doubling : $O(n)/n \approx O(1)$.

2) Array vs LL impl.

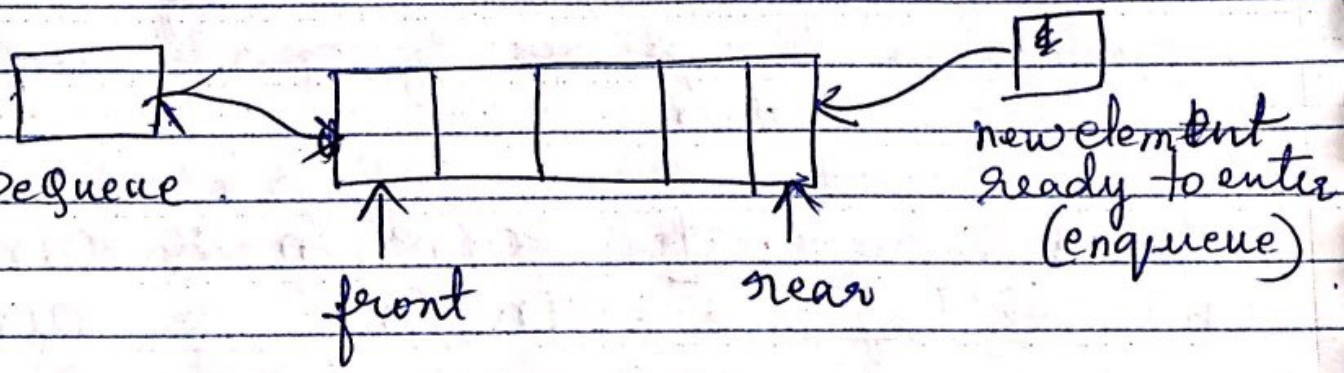
a) Array
- Operations take constant time
- Expensive doubling operation every once in a while.

b) LL
- Grows & shrinks gracefully
- Every operations takes constant time $O(1)$
- uses extra space & time to deal with pointers.

Queues

- It is an ordered list in which insertions are done at one end (rear) and deletions are done at other end (front).
- FIFO or LILO.
- EnQueue - Inserting an element to queue
- DeQueue - Deleting — " — from — " —

- DeQueueing an empty queue is underflow
- EnQueueing an element in a full queue is Overflow



Operations:

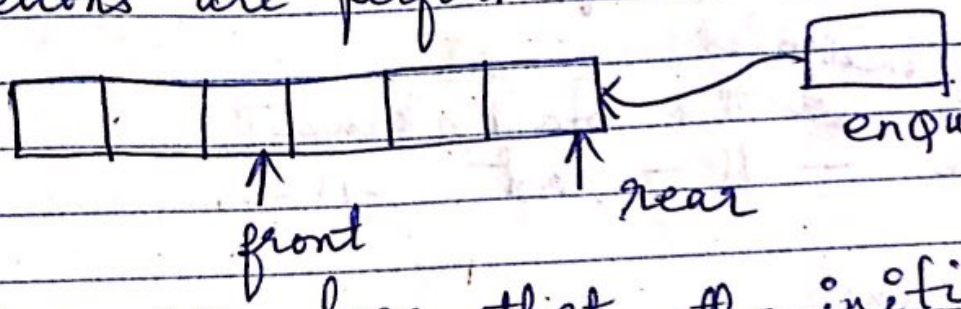
1) enQueue(int data)
2) deQueue()
3) Front() ⇒ "II" to peek
4) QueueSize()
5) IsEmptyQueue().

Implementation

1) Simple circular array based impl
2) Dynamic — " — " — " —
3) Linked List impl.

Why circular Arrays.

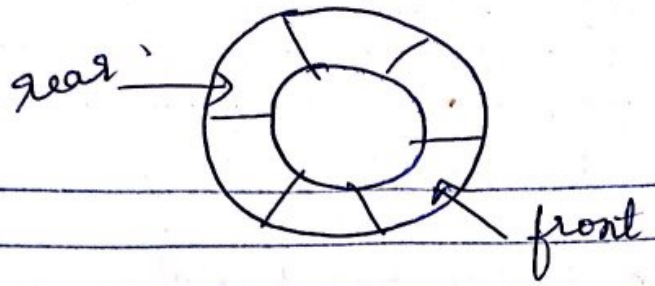
- Insertions are performed at one end and deletions are performed at the other end.



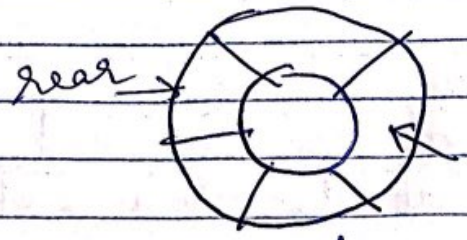
We can see here that the initial slots of the array are getting wasted. (∵ we insert from the end).
∴ ~~Simple~~ this is not efficient.

This problem can be solved by using circular arrays.

∴ we treat the last element and the first array elements as contiguous.
With this representation if there are any free slots at the beginning, the rear pointer can go to its next free slot.



① Simple Q'er array implementation.



fixed size array.

front: indicates the start element
rear — " — last —" —.

Performance

SC: O(n) for n enqueue operations
TC: O(1) for all operations

Limitations:

- Size is fixed.

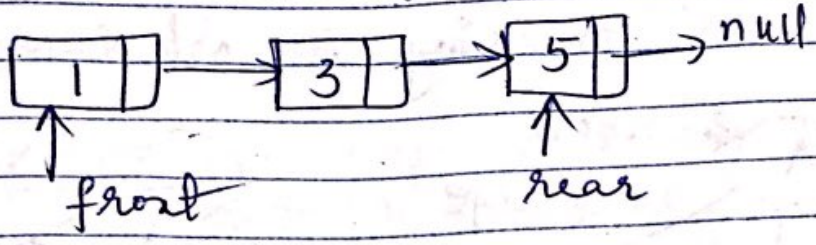
② Dynamic Q'er array impl.

SC: O(n) TC: O(1).

③ LL impl.

Enqueue - Inserting element at the end of the list.

Dequeue - delete an element from the beginning of the list.



Performance

SC : O(n) TC : O(1).