Sunbeam Institute of Information Technology
Pune and Karad

Module – Data Structures and Algorithms

Trainer - Devendra Dhande

Email – devendra.dhande@sunbeaminfo.com

Search insert position

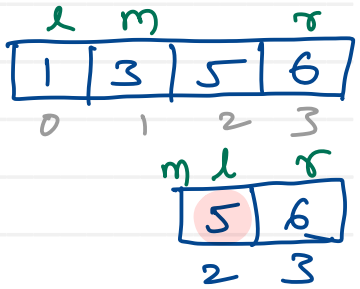
Given a sorted array of distinct integers and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.

You must write an algorithm with O(log n) runtime complexity.

Example 1:

Input: nums = [1,3,5,6], target = 5

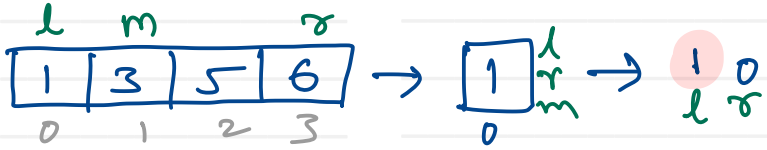
Output: 2



Example 2:

Input: nums = [1,3,5,6], target = 2

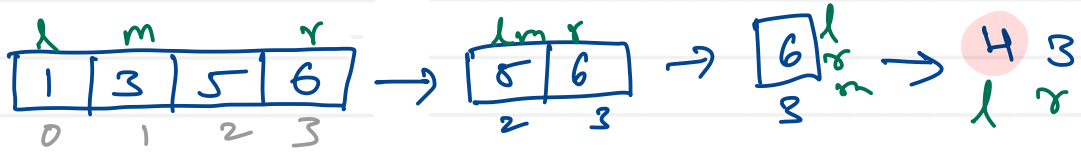
Output: 1



Example 3:

Input: nums = [1,3,5,6], target = 7

Output: 4



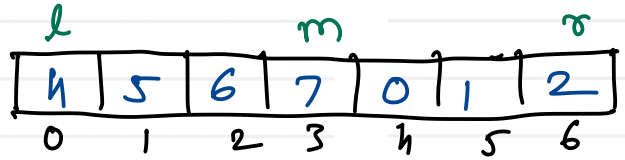

```
int searchInsert(int[] nums, int target) {
    int left = 0, right = nums.length-1, mid;
    while (left <= right) {
        mid = (left+right)/2;
        if(target == nums[mid])
            return mid;
        if (target < nums[mid])
            right = mid-1;
        else
            left = mid+1;
    }
    return left;
}
```

Search in rotated sorted array

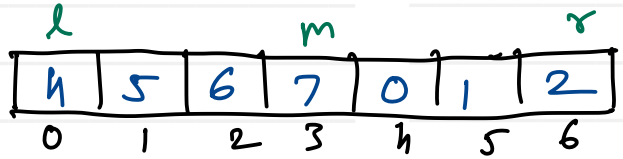
There is an integer array nums sorted in ascending order (with distinct values).
Prior to being passed to your function, nums is possibly rotated at an unknown pivot index k (1 <= k < nums.length) For example, [0,1,2,4,5,6,7] might be rotated at pivot index 3 and become [4,5,6,7,0,1,2].

Given the array nums after the possible rotation and an integer target, return the index of target if it is in nums, or -1 if it is not in nums. You must write an algorithm with O(log n) runtime complexity.

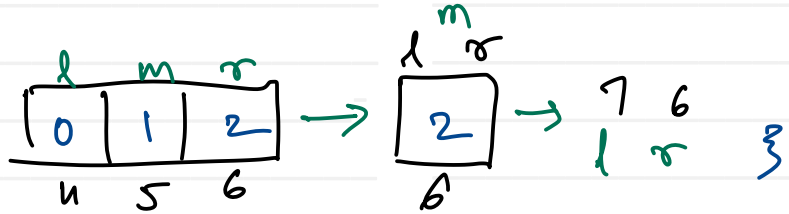
Example 1:
Input: nums = [4,5,6,7,0,1,2], target = 0
Output: 4



Example 2:
Input: nums = [4,5,6,7,0,1,2], target = 3
Output: -1



Example 3:
Input: nums = [1], target = 0
Output: -1

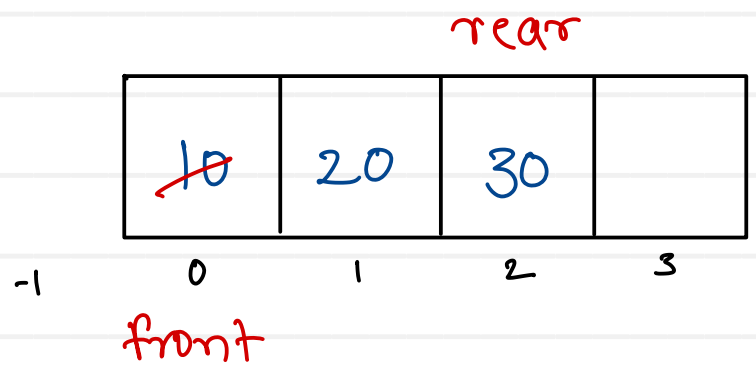



```
l=0, r=nums.length-1, m ;
while(l<=r) {
    m=(l+r)/2;
    if(target==nums[m])
        return m;
    if(nums[l]<=nums[m]) {
      if(nums[l]<=target && target<nums[mid])
            right=mid-1;
      else
            left=mid+1;
    }else {
      if(nums[m]<target && target<=nums[r])
            left=mid+1;
      else
            right=mid-1;
    }
```

Linear queue

- linear data structure which has two ends - front and rear
- Data is inserted from rear end and removed from front end
- Queue works on the principle of “First In First Out” / “FIFO”

size = 4



Operations :

1) insert/add/enqueue/push :
   a. reposition rear (inc)
   b. add value at rear index

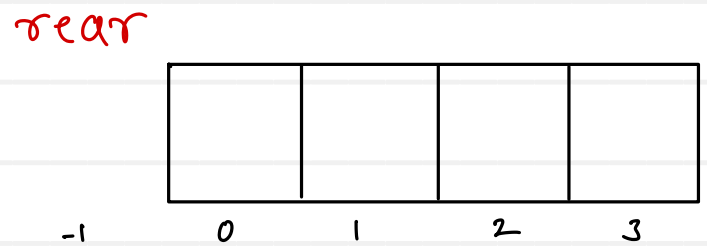
2) remove/delete/dequeue/pop :
   a. reposition front (inc)

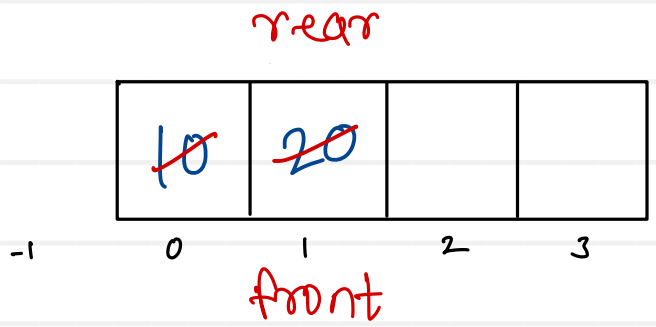
3) peek :
   a. read data from front end (front+1)

$T(n) = O(1)$ ← for all operations

Linear queue - Conditions

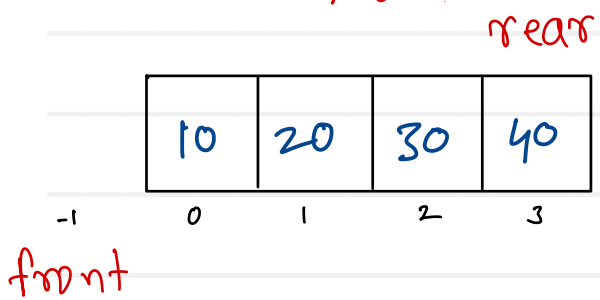
Empty



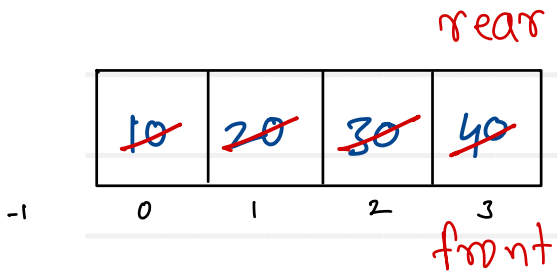


front == rear

Full



rear == size-1

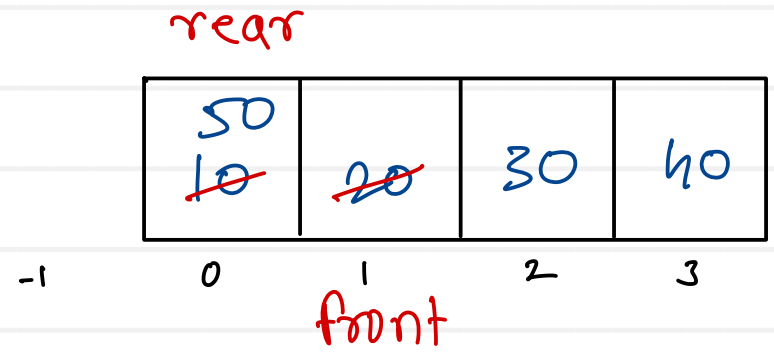



```
pop():
    front++;
    if( front == rear )
        front = rear = -1;
```

Disadvantage:
if rear is on last index of array & few initial locations are vaccant, but still we can't use them, this leads to poor memory utilization.

Circular queue

size=4

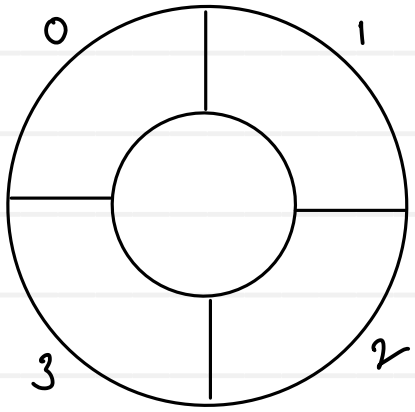


Operations :

1) insert/add/enqueue/push :
   a. reposition rear (inc)
   b. add value at rear index

2) remove/delete/dequeue/pop :
   a. reposition front (inc)

3) peek :
   a. read data from front end (front+1)



front = (front+1) % size
rear = (rear+1) % size

front = rear = -1
= (-1+1) % 4 = 0
= (0+1) % 4 = 1
= (1+1) % 4 = 2
= (2+1) % 4 = 3
= (3+1) % 4 = 0

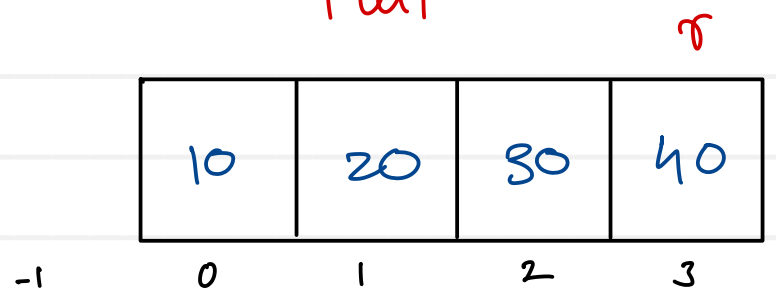
$T(n) = O(1)$ ← for all operations

Circular queue - Conditions

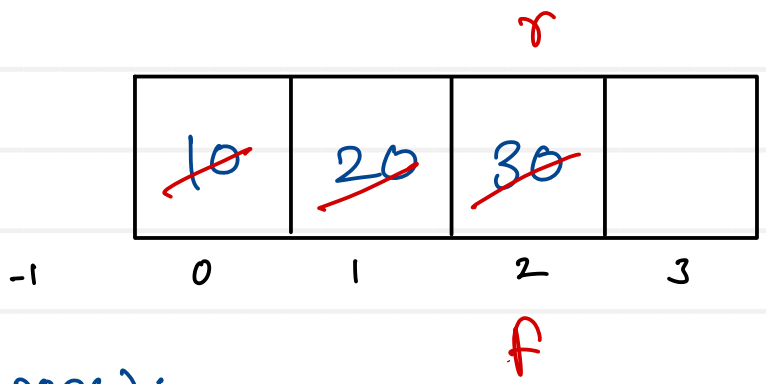
Empty

front == rear && rear == -1

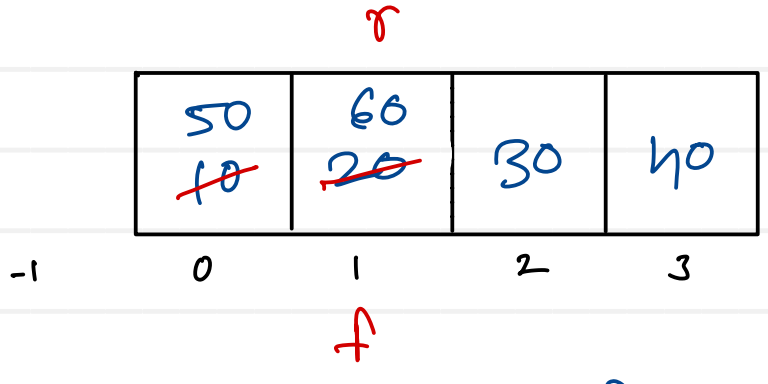
Full



front == -1 && rear == size-1




```
pop():
    front = (front+1) % size;
    if(front == rear)
        front = rear = -1;
```



front == rear && rear != -1

Stack

- Stack is a linear data structure which has only one end - top
- Data is inserted and removed from top end only.
- Stack works on principle of “Last In First Out” / “LIFO”

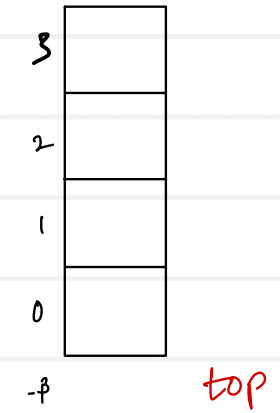
Operations :

1) insert / add / push :
   a. reposition top (inc)
   b. add value at top index

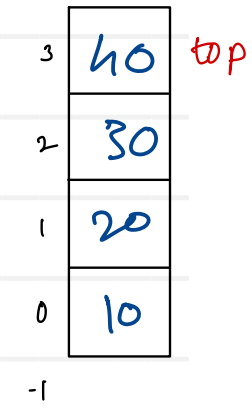
2) remove / delete / pop :
   a. reposition top (dec)

3) peek :
   a. read value of top index

$T(n) = O(1)$ - for all operations



top == -1



top == size-1

Ascending stack

top = -1

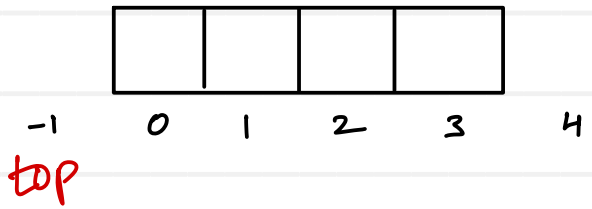
push : top++
arr[top] = value

pop : top--

peek : arr[top]

Empty : top == -1

Full : top == size-1



Descending stack

top = size

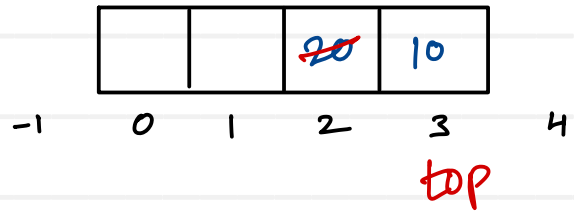
push : top--
arr[top] = value

pop : top++

peek : arr[top]

Empty : top == size

Full : top == 0



Applications – Stack and Queue

Stack

- Parenthesis balancing
- Expression conversion and evaluation
- Function calls
- Used in advanced data structures for traversing

- **Expression conversion and evaluation:**
  - Infix to postfix
  - Infix to prefix
  - Postfix evaluation
  - Prefix evaluation

Queue

- Jobs submitted to printer
- In Network setups – file access of file server machine is given to First come First serve basis
- Calls are placed on a queue when all operators are busy
- Used in advanced data structures to give efficiency.
- Process waiting queues in OS

Expression : combination of operands & operators

Type :

1) Infix : a + b (human)

2) Prefix : + a b

3) Postfix : a b +

(computer) → CPU → ALU

Operation :

( )

$

* / %

+ −

Thank you!!!

Devendra Dhande

devendra.dhande@sunbeaminfo.com