Sunbeam Institute of Information Technology
Pune and Karad

Data structures and Algorithms

Trainer - Devendra Dhande

Email – devendra.dhande@sunbeaminfo.com

Search insert position

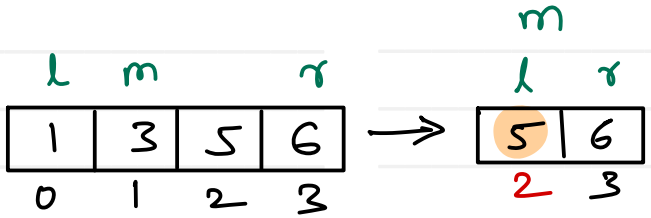
Given a sorted array of distinct integers and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.

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

Example 1:

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

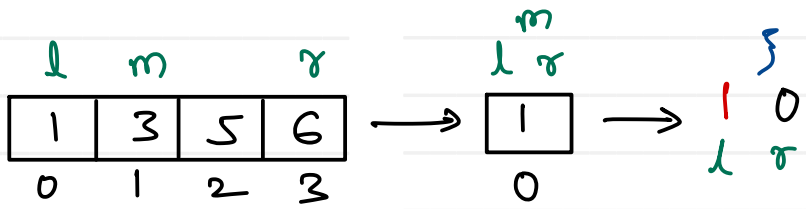
Output: 2



Example 2:

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

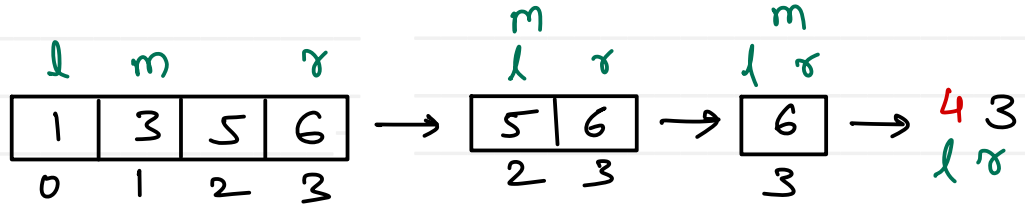
Output: 1



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;
```

Find first and last position of element in sorted array

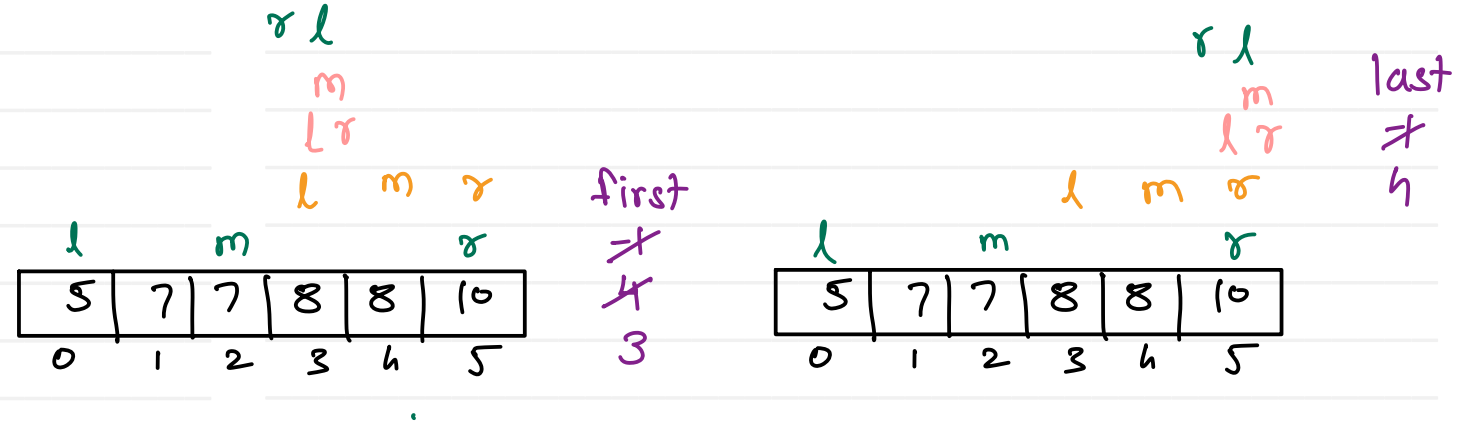
Given an array of integers nums sorted in non-decreasing order, find the starting and ending position of a given target value.
If target is not found in the array, return [-1, -1].
You must write an algorithm with O(log n) runtime complexity.

Example 1:
Input: nums = [5,7,7,8,8,10], target = 8
Output: [3,4]

Example 2:
Input: nums = [5,7,7,8,8,10], target = 6
Output: [-1,-1]

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

1. search target into the array
2. if key is found, find first position of target
3. if key is not found, return [-1,-1]
4. if key is found, find last position of target



find first Position

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

find last Position

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

Search in rotated sorted array

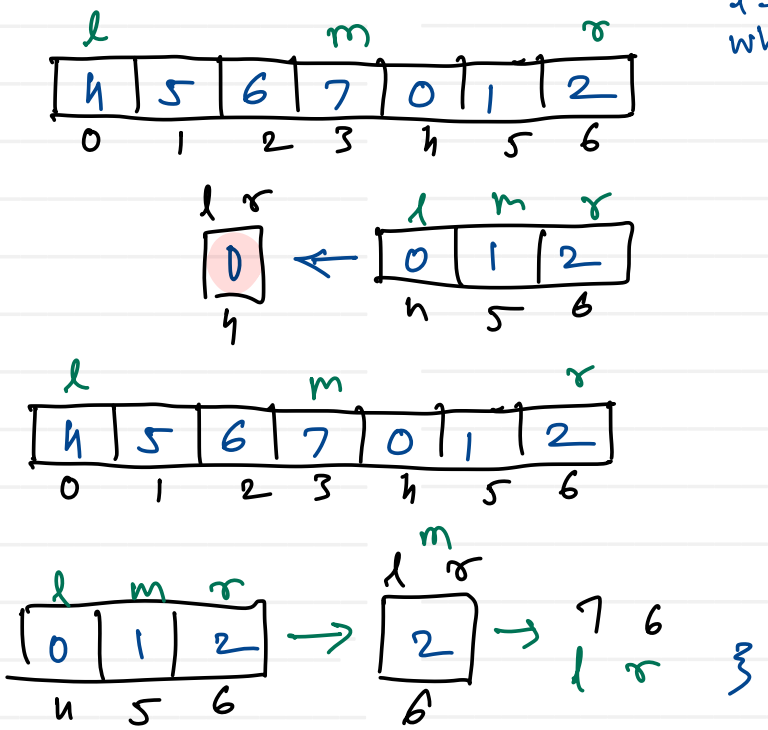
There is an integer array nums sorted in ascending order (with distinct values).
Prior to being passed to your function, nums is possibly rotated at an unknown pivot index k (1 <= k < nums.length) For example, [0,1,2,4,5,6,7] might be rotated at pivot index 3 and become [4,5,6,7,0,1,2].

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

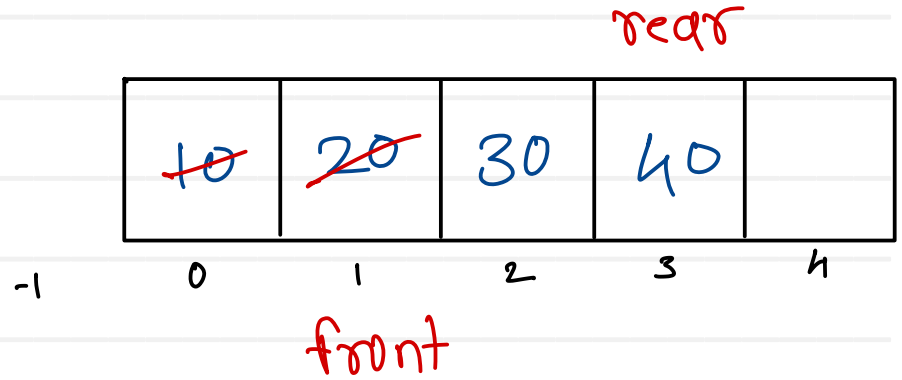



```
l=0, r=nums.length-1, m ;
while(l<=r){
    m=(l+r)/2;
    if(target==nums[m])
        return m;
    if(nums[l]<=nums[m]){
      if(nums[l]<=target && target<nums[mid])
            right=mid-1;
    else
            left=mid+1;
    }else{
      if(nums[m]<target && target<=nums[r])
            left=mid+1;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

length = 5



Operations:

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

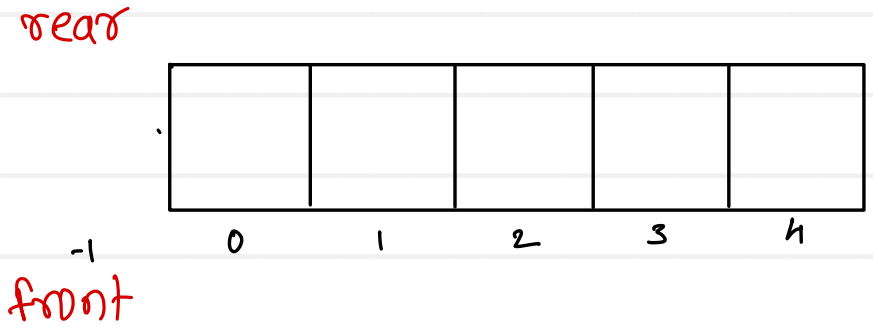
2. remove / delete / dequeue / pop:
   a. reposition front (inc)

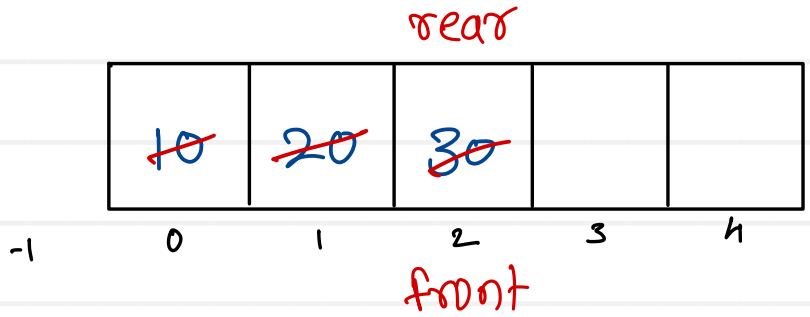
3. peek:
   a. read / return next data (front + 1)

All operations of queue are performed in O(1) time complexity

Linear queue - Conditions

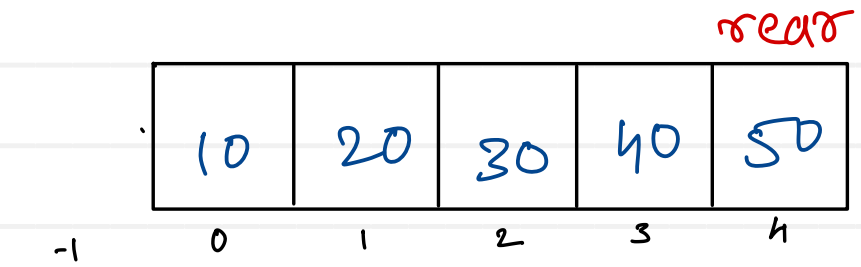
Empty



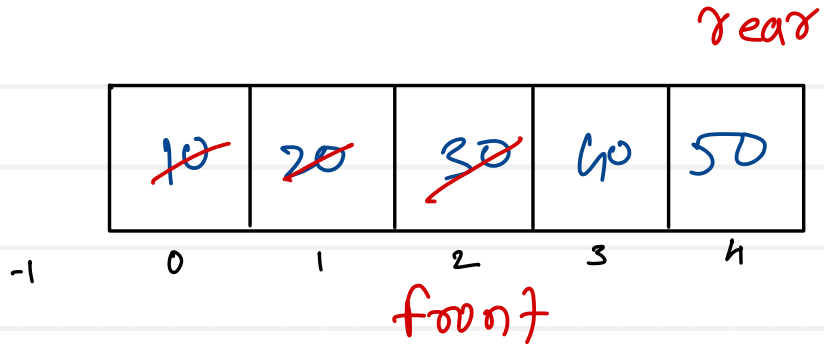


front == rear

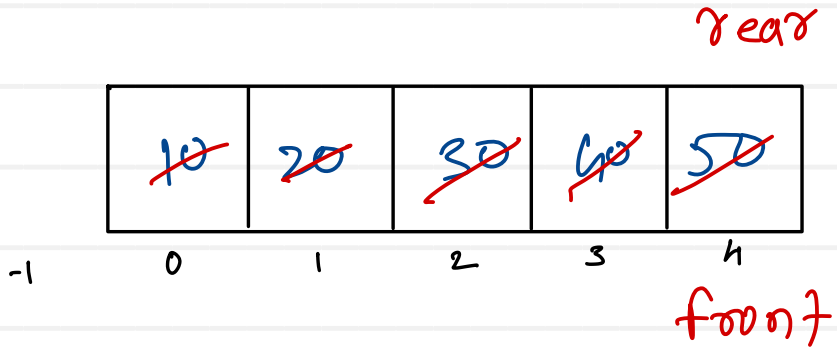
Full



rear == arr.length-1



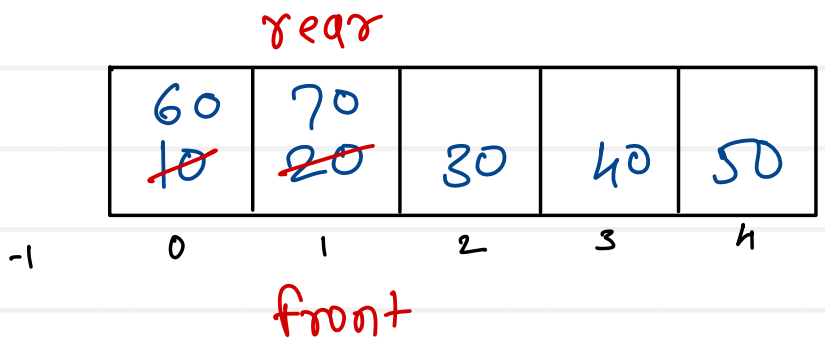
- if rear is at last index of array & initial few locations of array are empty, we can not reuse them untill queue is totally empty. This leads to poor memory utilization.




```
int pop( ) {
    int val = arr[front+1];
    front++;
    if ( front == rear)      // queue is empty
        front = rear = -1;
    return val;
}
```

Circular queue

capacity = 5

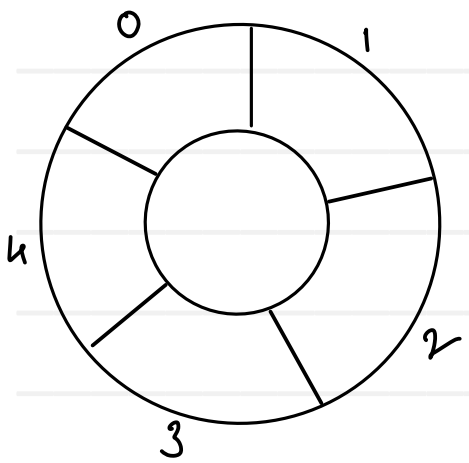


Operations :

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

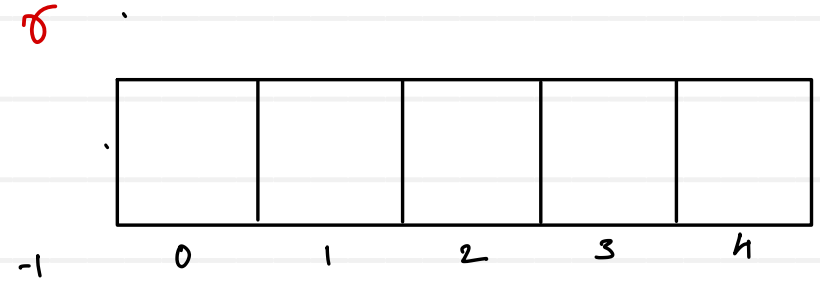
rear = (rear+1) % size

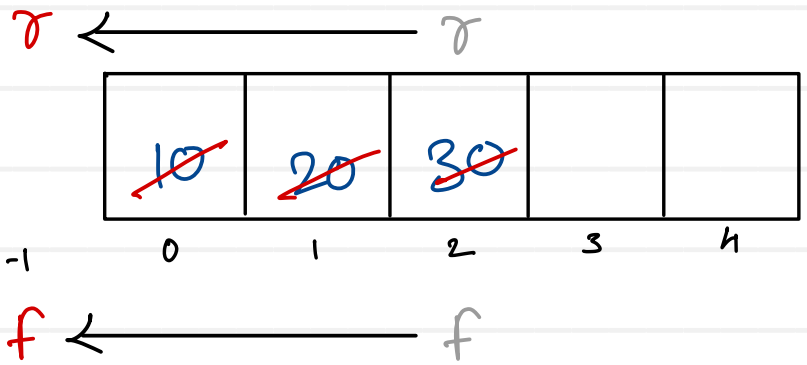
front = rear = -1

= (-1+1) % 5 = 0
= (0+1) % 5 = 1
= (1+1) % 5 = 2
= (2+1) % 5 = 3
= (3+1) % 5 = 4
= (4+1) % 5 = 0

Circular queue - Conditions

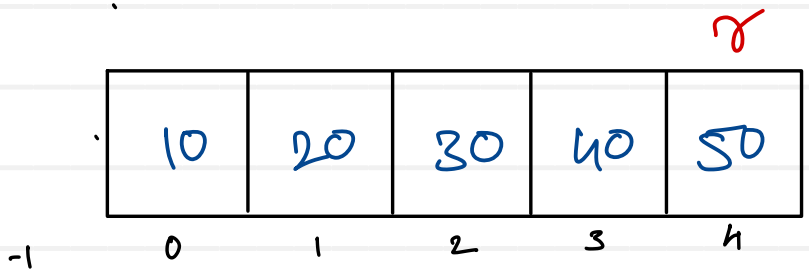
Empty



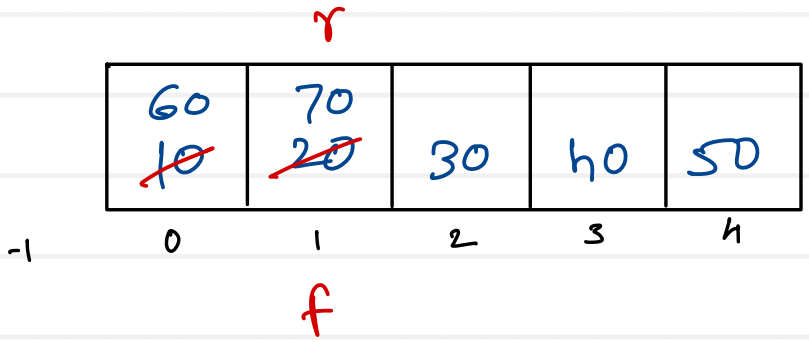


front == rear && rear == -1

Full



front == -1 && rear == arr.length-1



front == rear && rear != -1

Circular queue using count method

```
CircularQueue:
    int arr[] = new int[5];
    int rear=-1, front=-1;
    int count=0;

push(value):
    rear=(rear+1)%arr.length;
    arr[rear] = value;
    count++;

pop():
    front = (front+1)%arr.length;
    if(front == rear)
        front = rear=-1;
    count--;
```

```
isEmpty:
    count==0;

isFull:
    count == arr.length
```

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"

capacity = 5

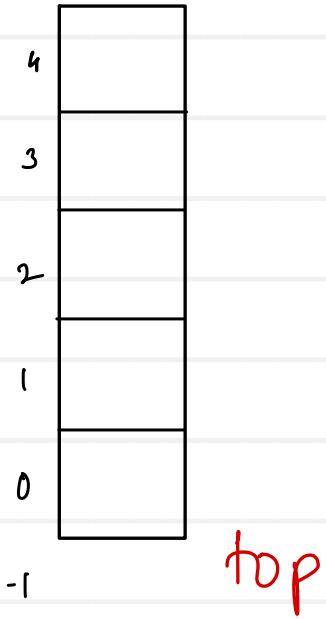
- top always points to last inserted data

Operations :

1. Push :
   a. reposition top (inc)
   b. add value at top position
2. Pop :
   a. reposition top (dec)
3. Peek :
   a. read/return next/topmost value

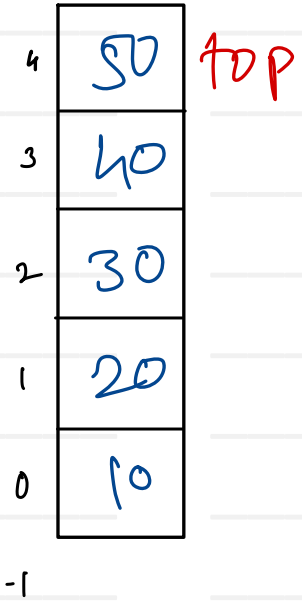
- All operations are performed in O(1) time complexity.

Empty



top == -1

Full



top == arr.length - 1

Ascending stack

top = -1

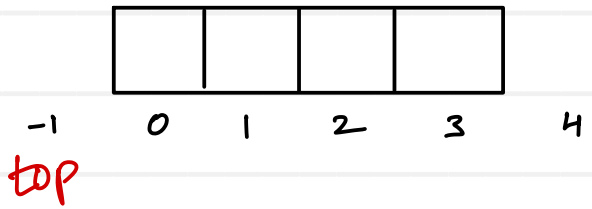
push : top++
arr[top] = value

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