Time Complexity

```
for(int i = 0 ; i < n  ; i++)
for(int i = n ; i > 0 ; i--)
```

$itr = n$

$Time \propto n$

$T(n) = O(n)$

```
for(int i = 0 ; i < n ; i+=20)
for(int i = 0 ; i < n ; i+=100)
```

$itr = n$

$Time \propto n$

$T(n) = O(n)$

```
for(int i = 0 ; i < 10 ; i++)
```

$\longrightarrow itr = 10$

$T(n) = O(1)$

```
for(int i = 0 ; i < n ; i++)
      for(int j = 0 ; j < n ; j++)
```

$itr = n * n$

$Time \propto n^2$

$T(n) = O(n^2)$

```
for(int i = 0 ; i < n ; i++);   — n
for(int j = 0 ; j < n ; j++);   — n
```

$itr = 2n$

$Time \propto \underline{2}n$

$T(n) = O(n)$

n	n^2
1	1
10	100
100	10000

```
for(int i = 0 ; i < n ; i++)
      for(int j = 0 ; j < n ; j++)   — n^2
for(int i = 0 ; i < n ; i++);        — n
```

$itr = n^2 + n$

$Time \propto n^2 + n$

$n >>>$

$n^2 >>>>> n$

$T(n) = O(n^2)$

for(int i = n ; i > 0 ; i/=2)
for(int i = 1 ; i < n ; i*=2)

$\}$ itr = log n

Time $\propto$ log n

$T(n) = O(\log n)$

Modification : '+' or '-' -> in terms of n
Modification : '*' or '/' -> in terms of log n

1 log n n nlog n n^2 n^3 2^n 3^n .....

Asymptotic Analysis

- **mathematical way of finding out complexities**
- **used to observe performance of algorithm for different inputs or change in sequence of input**
  - **Best case**
  - **Average case**
  - **Worst case**

Analysis of Searching Algorithms

11 55 33 66 44

Linear Search

Best case - O(1) -> if key is found at initial locations
Average case - O(n) -> if key is found at middle of array
Worst case - O(n) -> if key is found at last index of array or
key is not found

Binary Search

Best case - O(1) -> if key is found at initial comparisions
Average case - O(log n) -> if key is found at half comparisions
Worst case - O(log n) -> if key is found at last comparision or
key is not found

Analysis of Sorting algorithms

Pass	Comparisions
1	$n-1$
2	$n-2$
3	$n-3$
⋮	⋮
$n-1$	1

Array size = n

- to find time complexity of searching and sorting algorithms, find out number of comparisions

$$\text{Total Comparisions} = 1+2+3 \cdots\cdots (n-2)+(n-1)$$
$$= 1+2+3+\cdots\cdots n$$
$$= \frac{n(n+1)}{2}$$

Time $\propto$ comparisions

$$\text{Time} \propto \frac{n(n+1)}{2} = \frac{n^2+n}{2}$$

$$\text{Time} \propto n^2$$

$$T(n) = O(n^2)$$

Analysis of Sorting algorithms

	Selection sort	Bubble sort	Insertion sort
Best case	O(n^2)	O(n)	O(n)
Avg case	O(n^2)	O(n^2)	O(n^2)
Worst case	O(n^2)	O(n^2)	O(n^2)

Space Complexity

- finding out total space required in memory to execute an algorithms
- Total space required is addition of input space and auxillary sapce

Input space - space required to store actual data
Auxillary space - space required to process actual data

Total space = Input space + Auxillary space

1. Find out sum of array elements

```
int sumOfArray(int arr[n], int size){
    int sum = 0 ;
    for(int i = 0 ; i < size ; i++)
        sum += arr[i];
    return sum;
}
```

Input variables = arr[n]
Auxillary variables = size, sum, i

Input space = n
Auxillary space = 3
Total space = n+3
$S(n) = O(n)$

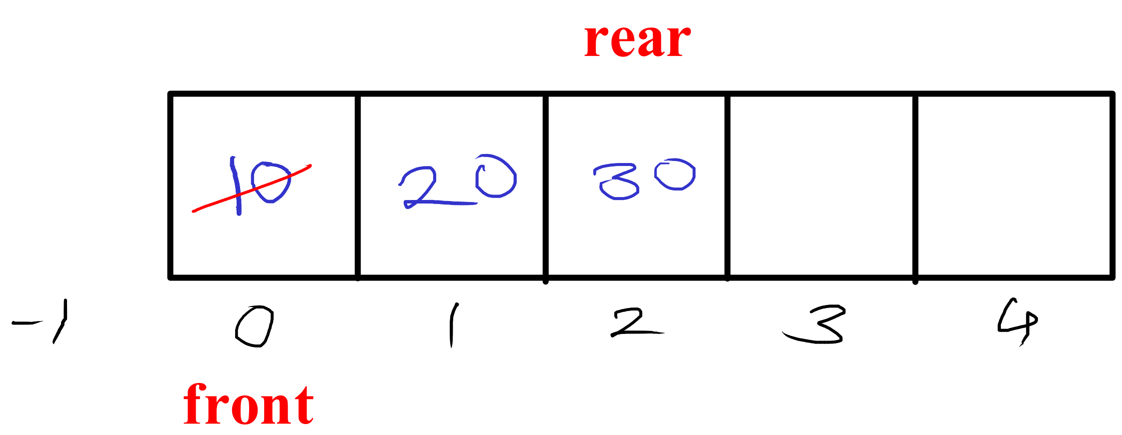
Input space complexity
$S(n) = O(n)$

Auxillary space complexity
$S(n) = O(1)$

Linear Queue

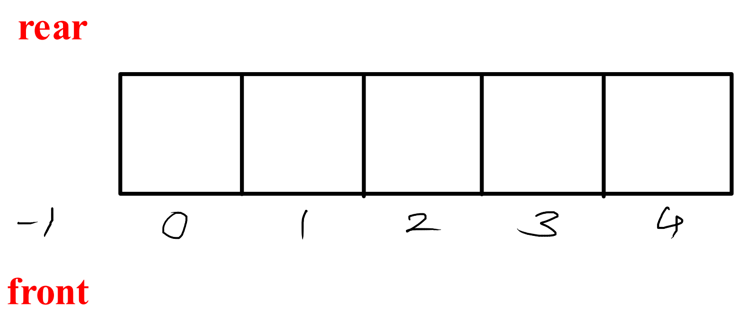
- queue is a linear data structre in which data is stored sequentially.
- queue has two ends - rear and front
- data insertion is allowed from only one end (rear)
- data deletion is allowed from another end (front)
- queue works on the principle of "First In First Out"
- queue is implemented using array or linked list
- all operation of queue are performed in O(1) time

size = 5



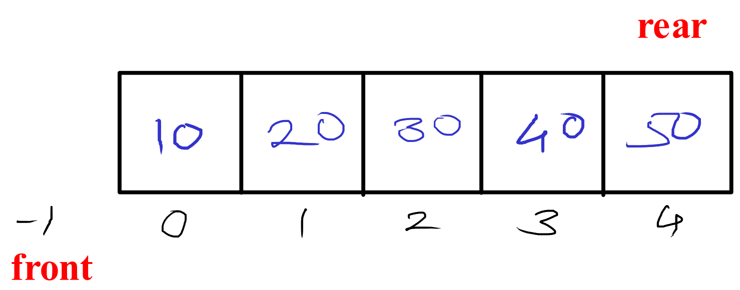
Conditions

Empty



rear == front

Full



rear == size -1

Operations

1. Add/Insert/Push/Enqueue

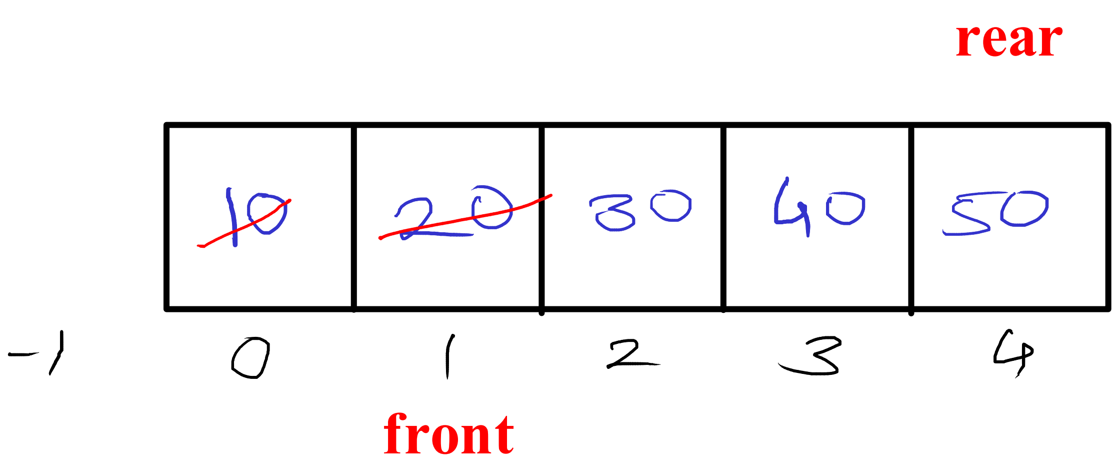
a. reposition the rear (inc)
b. add data at rear index

2. Delete/Remove/Pop/Dequeue

a. reposition front (inc)

3. Peek (collect)

a. read data of front + 1 index

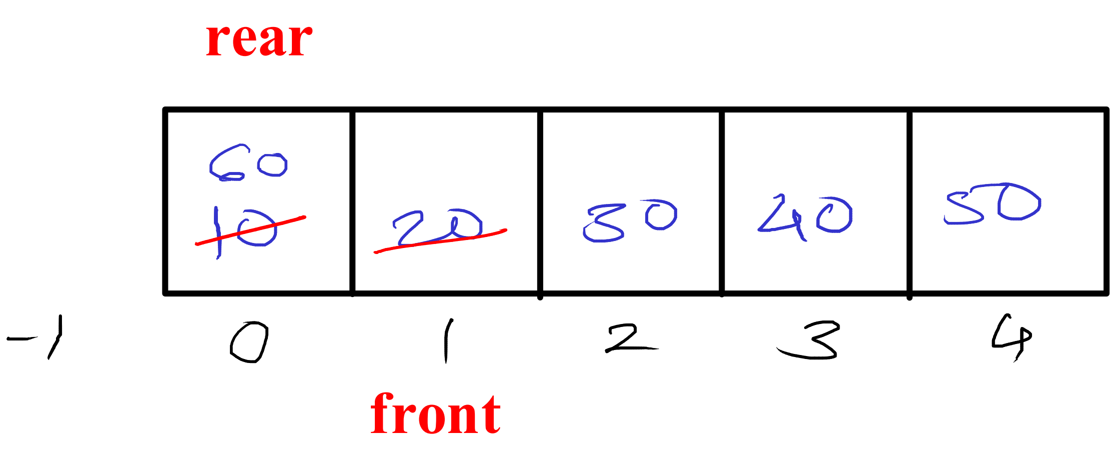


if rear reaches to the last index of array and few initial locations are available then we can not insert data on those locations. This will lead to poor memory utilization.

solution for above problem is circular queue

Circular Queue

size = 5



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

Operations

1. Add/Insert/Push/Enqueue

a. reposition the rear (inc)
b. add data at rear index
c. inc count

2. Delete/Remove/Pop/Dequeue

a. reposition front (inc)
b. dec count

3. Peek (collect)

a. read data of front + 1 index

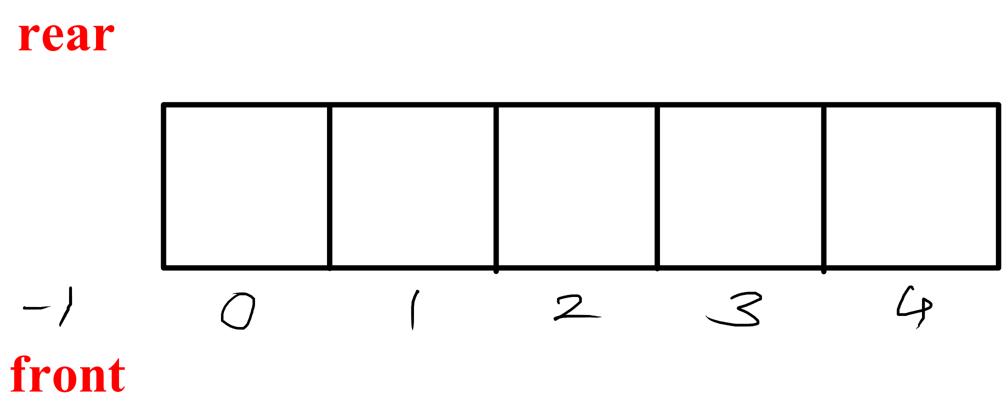
- to implement circular queue will use a count variable which will keep track of number of elements in a cir queue

int count = 0;

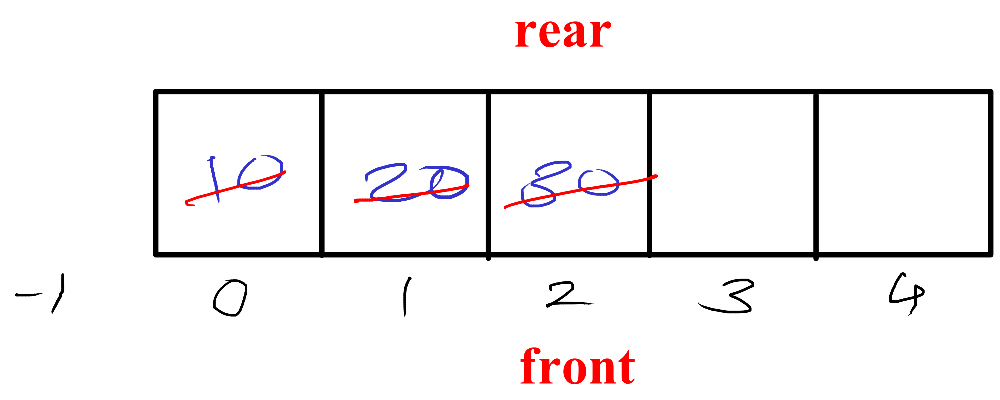
count == 0 -> empty
count == size -> full

Circular Queue - Empty and Full conditions

Empty

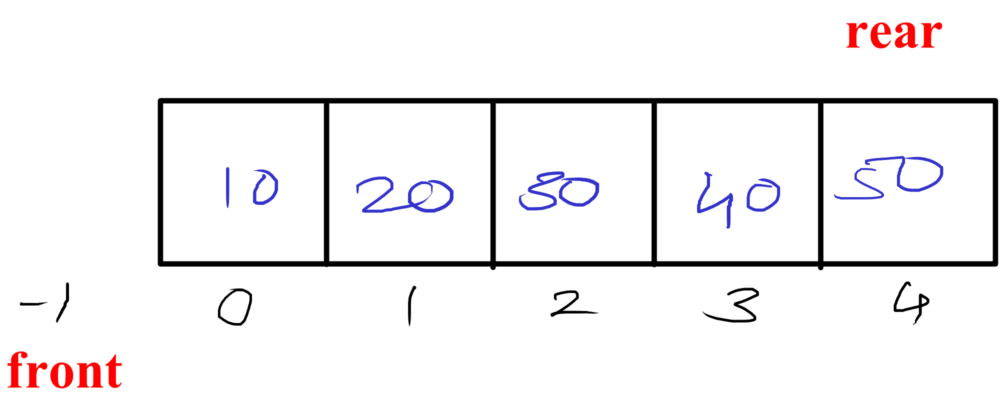


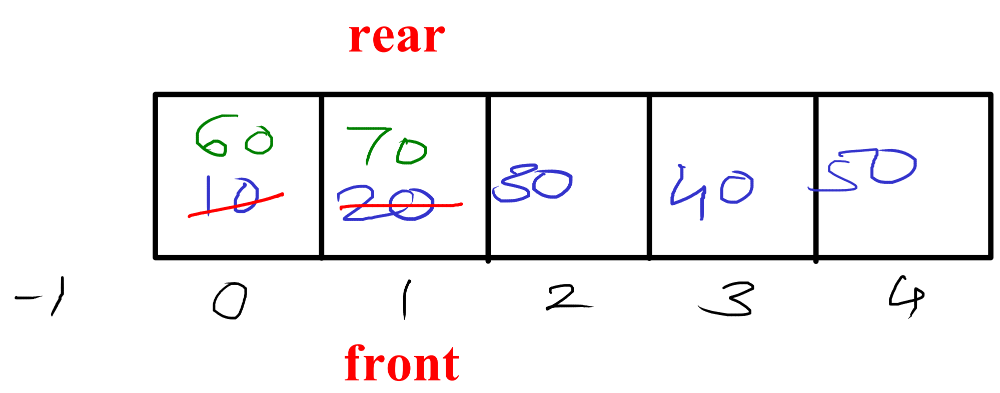
rear == front && rear == -1




```
pop(){
    front = (front + 1) % SIZE;
    if(rear == front)
        rear = front = -1;
}
```

Full





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

rear == front && rear != -1

(front == -1 && rear == size -1) || (rear == front && rear != -1)