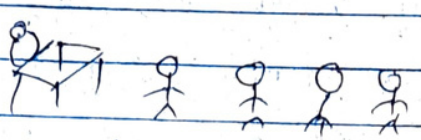
Queues

It is a logical datastructure and it works on discipline FIFO (First In First Out)



front end (For deletion)

Rear end (For insertion)

* Queue ADT -

Data -

1) Space for storing elements
2) front - for deletion
3) Rear - for insertion

Operations -

1) enqueue(x) - Inserting an element in queue
2) dequeue() - Deleting an element from queue
3) isEmpty()
4) isFull()
5) first()
6) Last()

* Queue can be implimented using two physical data structures that are -

i) Array
ii) Linked list

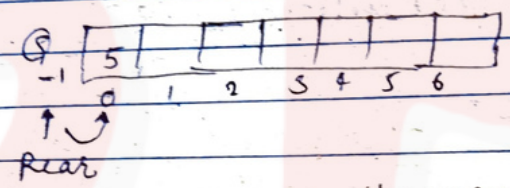
* Implementation of Queue using Array

i) Queue using - single pointer (index pointer)

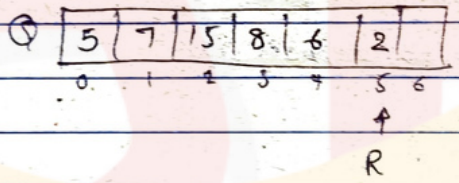
ii) Queue using - front and Rear (Double pointer)

iii) Drawbacks of Queue using Array.

1) Queue using single pointer -

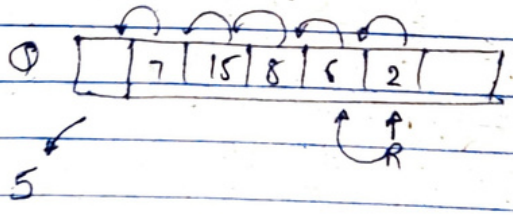


(To insert an element in the queue, just move the rear and insert.)



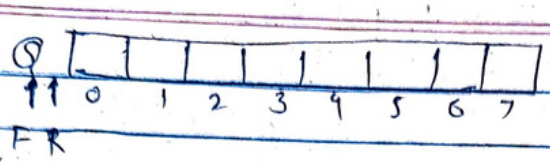
insert - O(1) → time

(For deleting, we can delete only very first element because it follows FIFO.)



delete - O(n)

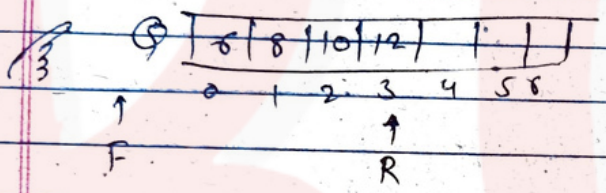
* Queue using two pointer -



This pointer is used for deletion.

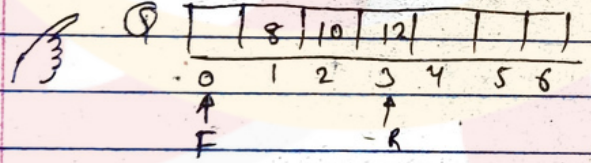
This pointer is used for insertion.

→ initially Front = Rear = -1



time - O(1)

Insert:- Move rear and insert at an element.

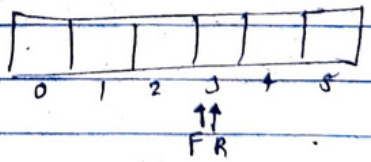


time - O(1)

Delete:- Move front to the next location and delete the element.

→ Empty condition

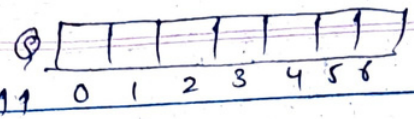
if (Front == Rear)



→ FULL Condition

if (Rear == size-1)

* Implementation



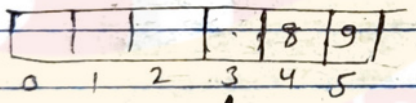

```
struct Queue
{   int size;
    int front;
    int rear;
    int *Q;
};

int main()
{   struct Queue q;
    printf("Enter size");
    scanf("%d", &q.size);
    q.Q = (int*) malloc(q.size * sizeof(int));
    q.front = q.rear = -1;
}

void enqueue(Queue *q, int x)
{
    if(q->rear == q->size-1)
        printf("Queue is Full");
    else
    {   q->Rear++;
        q->Q[q->rear] = x;
    }
}
```

```
void dequeue (Queue *q)
{
    int x = -1;
    if (q->front == q->rear)
        Printf ("Queue is empty");
    else
    {
        q->front++;
        x = q->Q[q->front];
    }
    return x;
}
```

Drawbacks - We can not reuse those pieces of deleted elements. It means every location can be used only once.

Eg:- Q 

(Here, If we want to insert we can't do so because rear is end is full-fill the full condition)

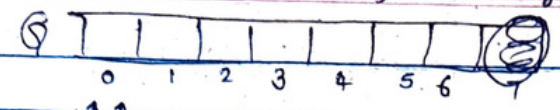
Solution of Drawbacks -

i) Resetting Pointers

ii) Circular Queue

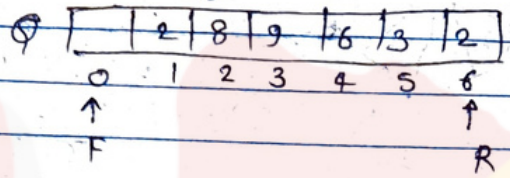
27/05/22

ii) Circular Queue :- (In this case, we will implement circular queue over an array by moving front and rear circularly.)

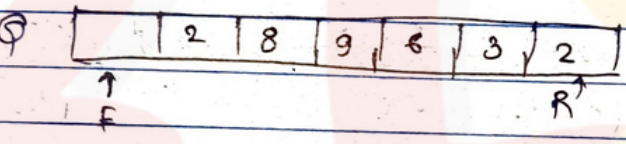


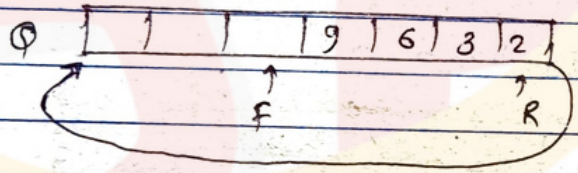
In the circular queue front and rear will start from 0th index

→ Adding elements -

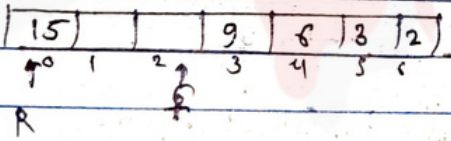


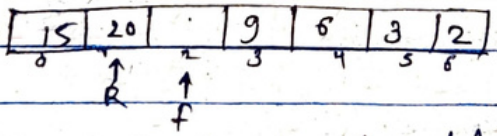
→ Deleting elements -





→ Bring the rear at zero index and insert the element in normal fashion -





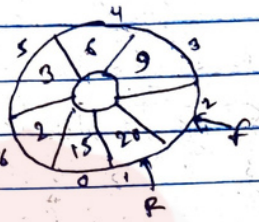
So, by using it we cannot use all seven places, only six elements can store.

(After it we can't add element because if we do it then front and rear are at ~~same~~ will both be in one place and a/c to previous it is the condition of empty state.) Due to this reason we can't use the ~~that is too~~ place wherever the front is placed

If we delete element using front then front also moves like rear.

Date
Page

* Representation of circular Queue



* To obtaining circular value, we use mode operation

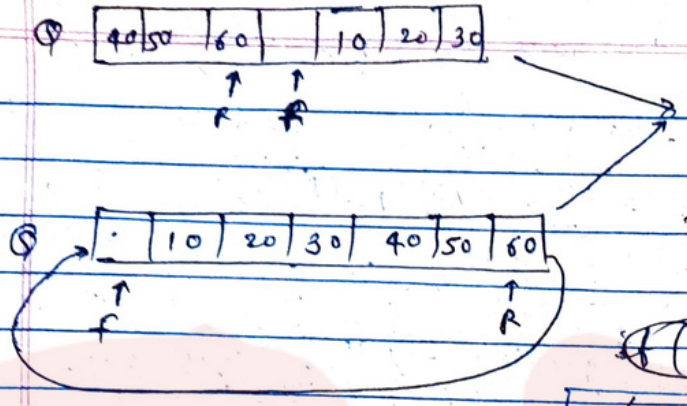
Rear	= (Rear+1) % size	next Rear position
0	(0+1) % 7 = 1	1
1	(1+1) % 7 = 2	2
2	(2+1) % 7 = 3	3
3	(3+1) % 7 = 4	4
4	(4+1) % 7 = 5	5
5	(5+1) % 7 = 6	6
6	(6+1) % 7 = 0	0
0	→ Same as above	

→ By using [(Rear+1) % size] statement we can move rear as well as front in circular way.

Rear = (Rear+1) % 7

front = (front+1) % 7

Full condition in circular queue



~~if (q->Rear+1) %. q->~~

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

Empty condition

same as previous -

if(front == rear)

→ int ~~de~~queue(struct queue *q) (Deleting operation in circular queue)

```
{ int x = -1
  if (q->front == q->Rear)
      printf ("Queue is Empty");
  else
  {
      q->front = (q->front+1) %. q->size;
      x = q->a[q->front];
  }
  return x;
}
```

time : O(1)

→ Insertion operation in circular queue

```
void enqueue (struct Queue *q, int x)
{ if (q->Rear+1) %. q->size == q->front)
      printf ("Queue is full");
  else
  {
      q->Rear = (q->Rear+1) %. q->size;
      q->Q[q->Rear] = x;
  }
}
```

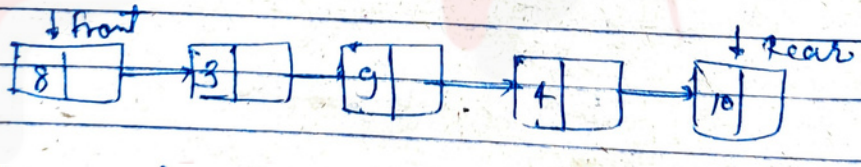
* Implementation of queue using linked list -

→ Empty

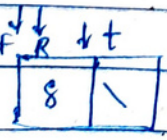
```
if (front == NULL)
```

→ Full (When heap is full)

```
Node *t = New. node;
if (t == NULL);
```

* 

* When very first node created in linked then front and rear both should be place on first node.



* Insertion -

Date
Page

```
Void enqueue(int x)
{ Node *t = new node;
  if (t == NULL)
     printf("Queue is Full");
  else
     {
       t->data = x;
       t->next = NULL;
       if (front == NULL)
          front = rear = t;
       else
          {
            Rear->next = t
            Rear = t;
          }
     }
```

(When very first node creates) When there is no any node in linked list then it will work.

When already there is some node in linked list then it will work.

* Deletion -

```
Void dequeue()
{ int x = -1;
  Node *P;
  if (front == NULL)
     printf("Queue is Empty");
  else
     {
       P = front;
       front = front->next;
       x = P->data;
       free(P);
     }
  return x;
}
```

* DE Queue (Double Ended queue) → Not follow FIFO

In this queue we can use front and rear both pointers for both operation, insertion as well as deletion.

→ Normal Queue

	insert	delete
front	✗	✓
rear	✓	✗

→ DE Queue

	insert	delete
front	✓	✓
rear	✓	✓

* Variation of DE Queue -

i) input Restricted DE queue -

	insert	delete
front	✗	✓
rear	✓	✓

ii) Output restricted DE queue -

	insert	delete
front	✓	✓
rear	✓	✗