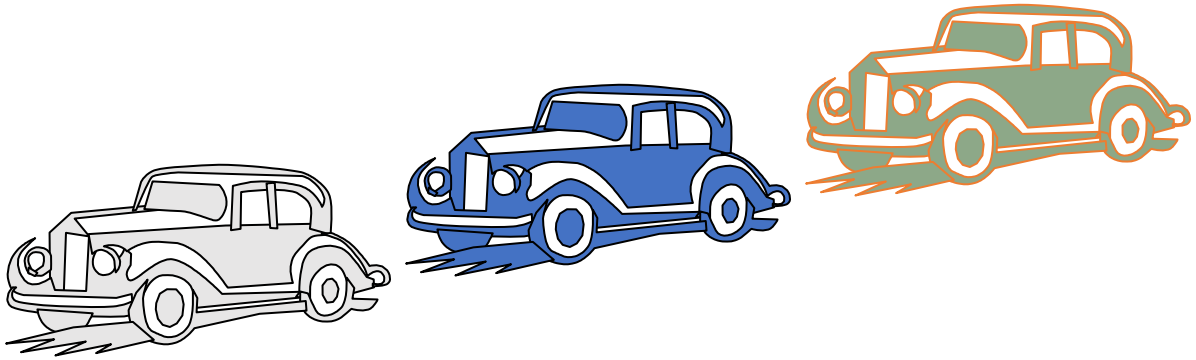
Queue



Queues

Lists, Array, List using arrays and pointers, Sparse matrices and its representation.

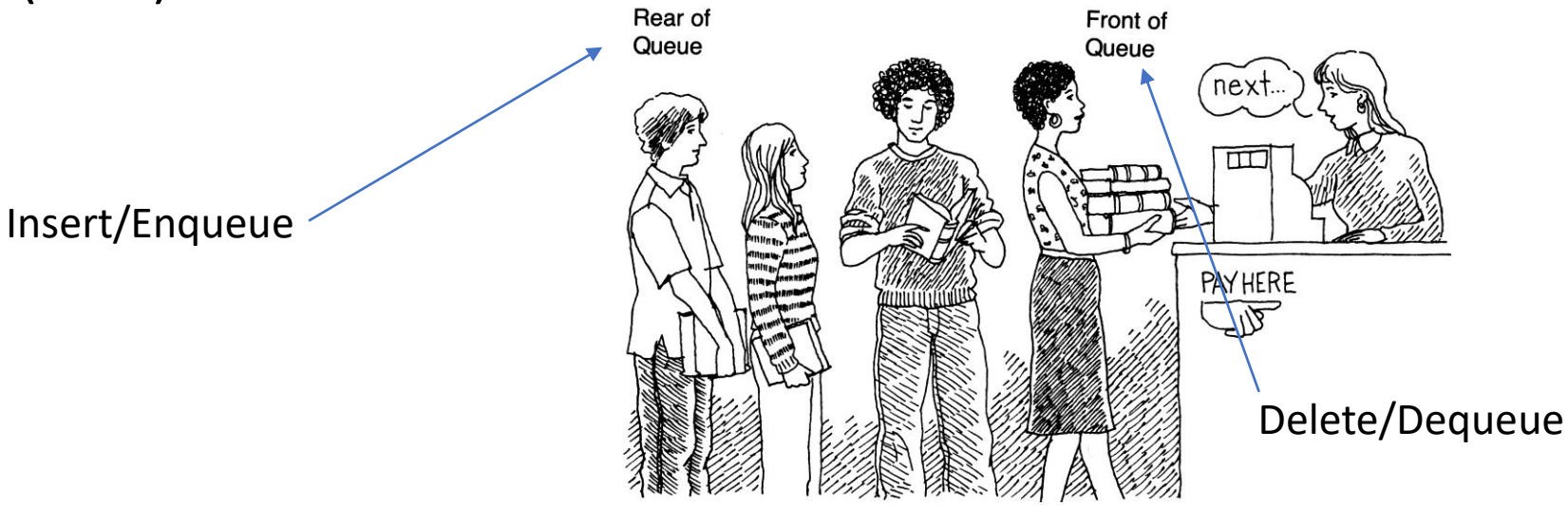
Stacks: Stack, operations, implementation using arrays, linked list and Stack Applications: Infix to postfix expression conversion, Evaluation of Postfix expressions, balancing the symbols.

Queue: Queues, operations, implementation using arrays, linked list & its applications. Circular queue: definition &its operations, De queue: definition & its types, applications.

Linked lists; Singly Linked, Doubly Linked, Circular Linked Lists, Polynomial ADT.

Concept of queue

- **Queue** is a linear data structure which stores its elements in a linearly ordered manner. Inserting element at one end (rear) and deleting element from another end (front).
- A queue is a FIFO (First-In, First-Out) data structure in which the element that is inserted first is the first one to be taken out.
- The term queue comes from the analogy that people are in queue waiting for services. First come, First served (FCFS)

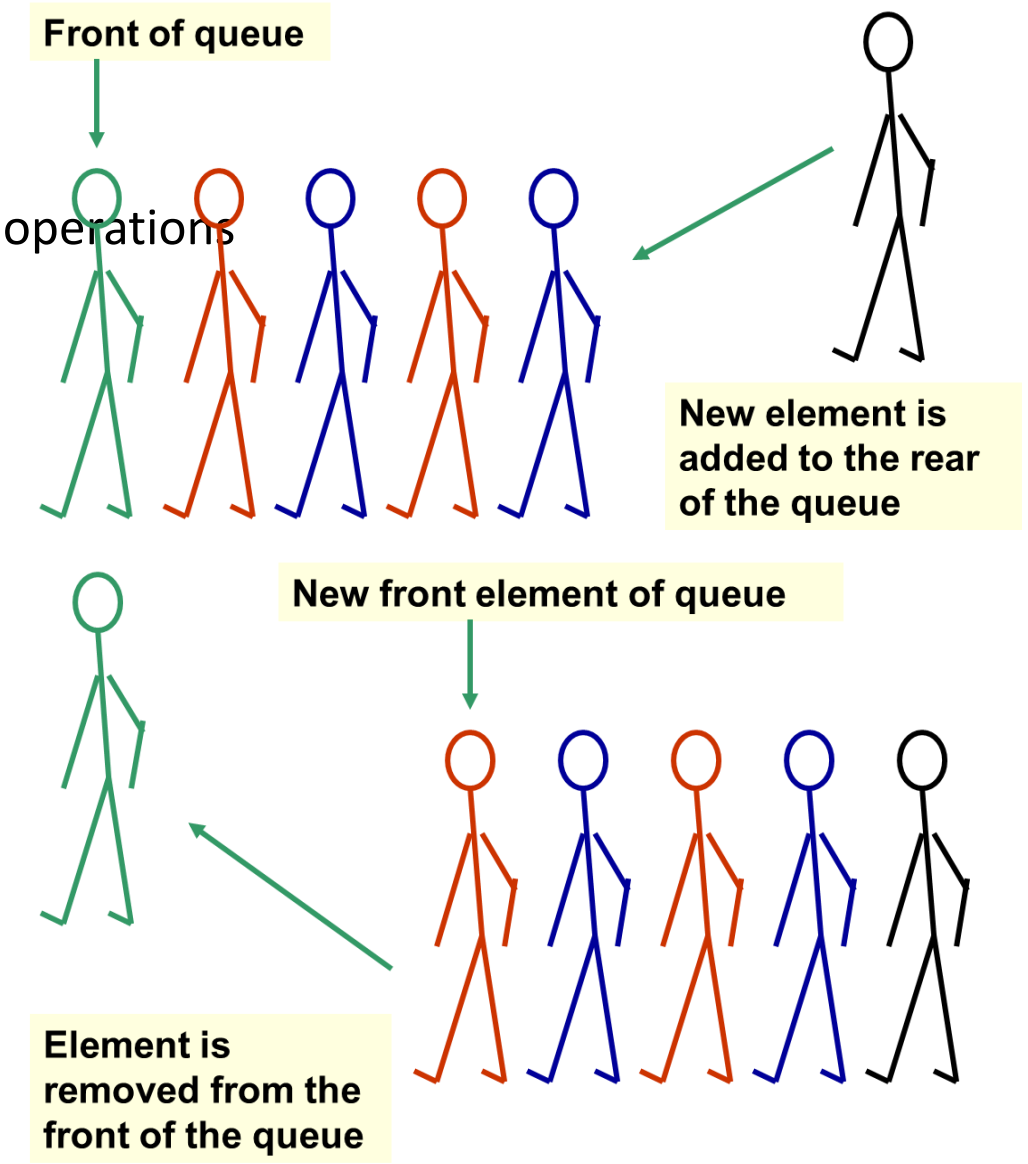


Queue operations

- Inserting element --- enqueue
- Deleting element --- dequeue
- Need know the positions of rare and front for efficient operations

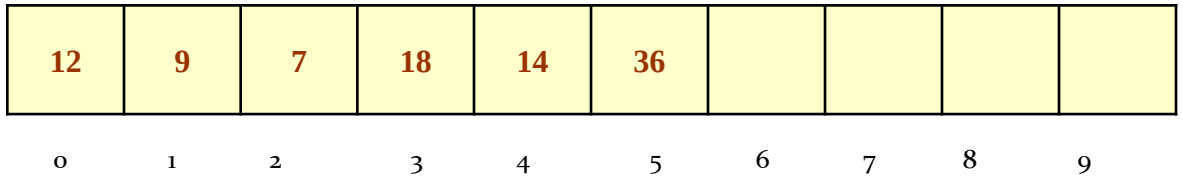
Queue implementations

- using an array
- using a linked list

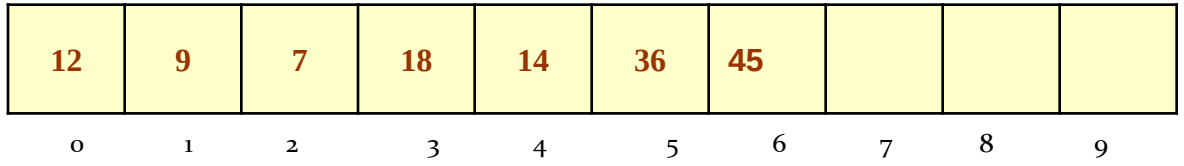


Array Representation of Queues

- Queues can be easily represented using arrays.
- Every queue has front and rear variables that point to the position from where deletions and insertions can be done, respectively.
- Consider the queue shown in figure

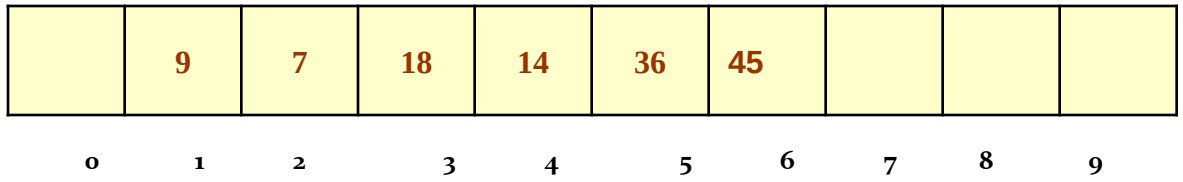


- Here, **front = 0 and rear = 5**.
- If we want to add one more value in the list say with value 45, then rear would be incremented by 1 and the value would be stored at the position pointed by rear.



Array Representation of Queues

- Now, front = 0 and rear = 6. Every time a new element has to be added, we will repeat the same procedure.
- Now, if we want to delete an element from the queue, then the value of front will be incremented. Deletions are done from only this end of the queue.



- Now, front = 1 and rear = 6.

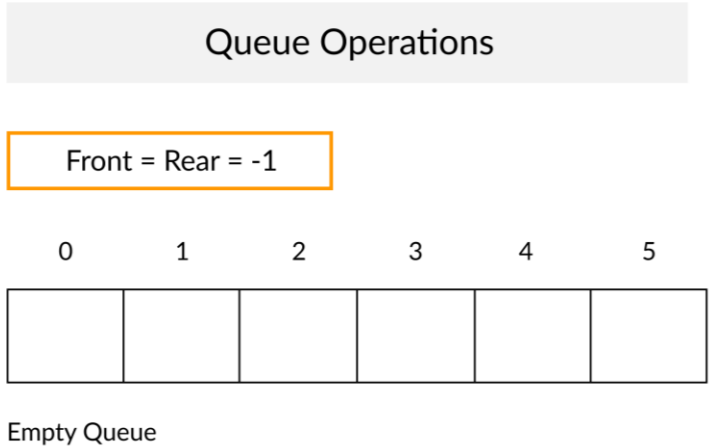
Array Representation of Queues

- Before inserting an element in the queue we must check for **overflow** conditions.
- An overflow occurs when we try to insert an element into a queue that is already full, i.e. when rear = MAX – 1, where MAX specifies the maximum number of elements that the queue can hold.
- Similarly, before deleting an element from the queue, we must check for **underflow** condition.
- An underflow occurs when we try to delete an element from a queue that is already empty. If front = -1 and rear = -1, this means there is no element in the queue.

Algorithm for Insertion Operation

```
Algorithm to insert an element in a queue
Step 1: IF REAR=MAX-1, then;
            Write OVERFLOW
           Goto Step 4
        [END OF IF]
Step 2: IF FRONT == -1 and REAR = -1, then
            SET FRONT = REAR = 0
        ELSE
            SET REAR = REAR + 1
        [END OF IF]
Step 3: SET QUEUE[REAR] = NUM
Step 4: Exit
```

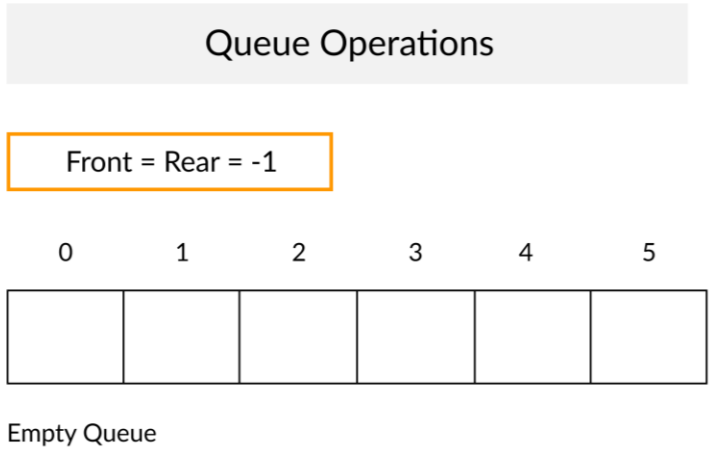
Time complexity: O(1)



Algorithm for Deletion Operation

```
Algorithm to delete an element from a queue
Step 1: IF FRONT = -1 OR FRONT > REAR, then
            Write UNDERFLOW
            [FRONT = -1 and REAR = -1]
            Goto Step 2
         ELSE
            SET VAL = QUEUE[FRONT]
            SET FRONT = FRONT + 1
        [END OF IF]
Step 2: Exit
```

Time complexity: O(1)



Operations Associated with a Queue

- **isEmpty():** To check if the queue is empty
- **isFull():** To check whether the queue is full or not
- **dequeue():** Removes the element from the frontal side of the queue
- **enqueue():** It inserts elements to the end of the queue
- **Front**: Pointer element responsible for fetching the first element from the queue
- **Rear**: Pointer element responsible for fetching the last element from the queue

Operations Associated with a Queue

```
void enqueue()
{
   int insert_item;
   if (Rear == SIZE - 1)
      printf("Overflow \n");
   else
   {
      if (Front == - 1)

      Front = 0;
      printf("Element to be inserted in the Queue\n : ");
      scanf("%d", &insert_item);
      Rear = Rear + 1;
      inp_arr[Rear] = insert_item;
   }
}
```

```
void dequeue()
{
   if (Front == - 1 || Front > Rear)
   {
      printf("Underflow \n");
       Front = - 1
       Rear  = - 1
      return ;
   }
   else
   {
      printf("Element deleted from the Queue:
%d\n", inp_arr[Front]);
      Front = Front + 1;
   }
}
```

```
void show()
{

   if (Front == - 1)
      printf("Empty Queue \n");
   else
   {
      printf("Queue: \n");
      for (int i = Front; i <= Rear; i++)
         printf("%d ", inp_arr[i]);
      printf("\n");
   }
}
```

```
int inp_arr[SIZE];
int Rear = - 1;
int Front = - 1;
```

Python3 program to implement a queue using an array

Another approach

```
class Queue:

    # To initialize the object.
    def __init__(self, c):

        self.queue = []
        self.front = self.rear = 0
        self.capacity = c

    # Function to insert an element
    # at the rear of the queue
    def queueEnqueue(self, data):

        # Check queue is full or not
        if(self.capacity == self.rear):
            print("\nQueue is full")

        # Insert element at the rear
        else:
            self.queue.append(data)
            self.rear += 1

    # Function to delete an element
    # from the front of the queue
    def queueDequeue(self):

        # If queue is empty
        if(self.front == self.rear):
            print("Queue is empty")

        # Pop the front element from list
        else:
            x = self.queue.pop(0)
            self.rear -= 1
```

```
    # Function to print queue elements
    def queueDisplay(self):


        if(self.front == self.rear):
            print("\nQueue is Empty")


        # Traverse front to rear to
        # print elements
        for i in self.queue:
            print(i, "<--", end='')


    # Print front of queue
    def queueFront(self):


        if(self.front == self.rear):
            print("\nQueue is Empty")


        print("\nFront Element is:",
              self.queue[self.front])
```

```
# Driver code
if __name__ == '__main__':

    # Create a new queue of
    # capacity 4
    q = Queue(4)

    # Print queue elements
    q.queueDisplay()

    # Inserting elements in the queue
    q.queueEnqueue(20)
    q.queueEnqueue(30)
    q.queueEnqueue(40)
    q.queueEnqueue(50)

    # Print queue elements
    q.queueDisplay()

    # Insert element in queue
    q.queueEnqueue(60)

    # Print queue elements
    q.queueDisplay()


    q.queueDequeue()
    q.queueDequeue()
    print("\n\nafter two node deletion\n")

    # Print queue elements
    q.queueDisplay()

    # Print front of queue
    q.queueFront()
```

1

Front=-1 Queue is empty

0 1 2 3 4

Rear=-1

2

Front=0 Insert A

0 1 2 3 4

Rear=0

3

Front=0 Insert B

0 1 2 3 4

Rear=1

4

Front=0 Insert C

0 1 2 3 4

Rear=2

5

Front=1 Delete

0 1 2 3 4

Rear=2

6

Front=2 Delete

0 1 2 3 4

Rear=2

7

Front=3 Delete

0 1 2 3 4

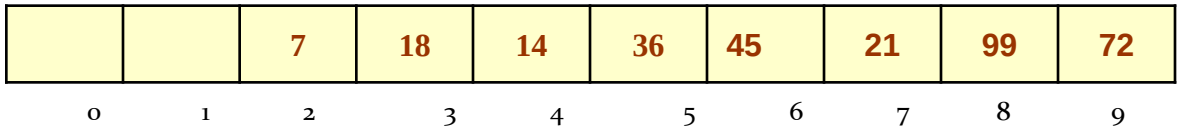
Rear=2

Queue is empty

Entry point is called Rear &
Exit point is called Front

Problem ?
Queue using Array

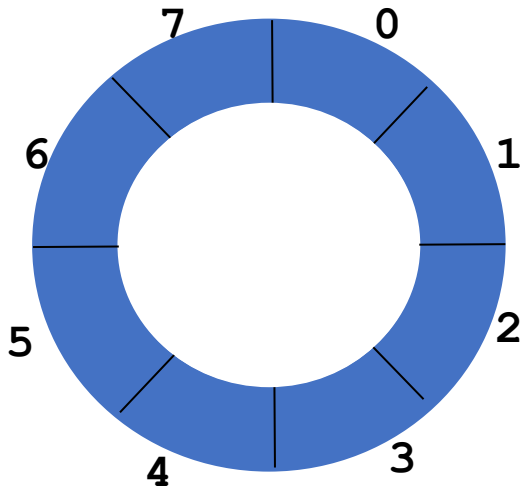
Circular Queues

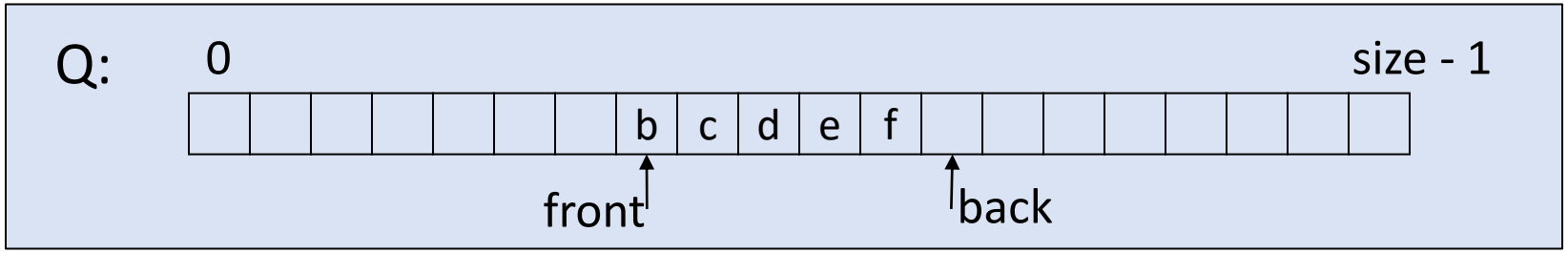


- We will explain the concept of circular queues using an example.
- In this queue, front = 2 and rear = 9.
- Now, if you want to insert a new element, it cannot be done because the space is available only at the left of the queue.
- If rear = MAX – 1, then OVERFLOW condition exists.
- This is the major drawback of an array queue. Even if space is available, no insertions can be done once rear is equal to MAX – 1.
- **This leads to wastage of space. In order to overcome this problem, we use circular queues.**
- In a circular queue, the first index comes right after the last index.
- A circular queue is full, only when front=0 and rear = Max – 1.

Implementing Queue ADT: Circular Array Queue

- **Neat trick**: use a ***circular array*** to insert and remove items from a queue in constant time.
- The idea of a circular array is that the end of the array "wraps around" to the start of the array.





Circular Queue Implementation

- After rear reaches the last position, i.e., MAX-1 in order to reuse the vacant positions, we can bring rear back to the 0th position, if it is empty, and continue incrementing rear in same manner as earlier **rear=(rear+1)%MAX**.
- Thus, rear will have to be incremented circularly. For deletion, front will also have to be incremented circularly **front = (front+1)%MAX**.

Enqueue(Insert)

Step 1: IF (REAR+1)%MAX = FRONT
Write " OVERFLOW "
Goto step 4
[End OF IF]
Step 2: IF FRONT = -1 and REAR = -1
SET FRONT = REAR = 0
ELSE IF REAR = MAX - 1 and FRONT ! = 0
SET REAR = 0
ELSE
SET REAR = (REAR + 1) % MAX
[END OF IF]
Step 3: SET QUEUE[REAR] = VAL
Step 4: EXIT

Dequeue(Insert)

Step 1: IF FRONT = -1
Write " UNDERFLOW "
Goto Step 4
[END of IF]
Step 2: SET VAL = QUEUE[FRONT]
Step 3: IF FRONT = REAR
SET FRONT = REAR = -1
ELSE
IF FRONT = MAX -1
SET FRONT = 0
ELSE
SET FRONT = (FRONT + 1) % MAX
[END of IF]
[END OF IF]
Step 4: EXIT

```
1.void display()
2.{
3.   int i=front;
4.   if(front==-1 && rear==-1)
5.   {
6.      printf("\n Queue is empty..");
7.   }
8.   else
9.   {
10.      printf("\nElements in a Queue are :");
11.      while(i<=rear)
12.      {
13.         printf("%d,", queue[i]);
14.         i=(i+1)%max;
15.      }
16.   }
17.}
```

```
1.int main()
2.{
3.   int choice=1,x;   // variables declaration
4.
5.   while(choice<4 && choice!=0)   // while loop
6.   {
7.   printf("\n Press 1: Insert an element");
8.   printf("\nPress 2: Delete an element");
9.   printf("\nPress 3: Display the element");
10.   printf("\nEnter your choice");
11.   scanf("%d", &choice);
12.
13.   switch(choice)
14.   {
15.
16.      case 1:
17.
18.      printf("Enter the element which is to be inserted");
19.      scanf("%d", &x);
20.      enqueue(x);
21.      break;
22.      case 2:
23.      dequeue();
24.      break;
25.      case 3:
26.      display();
27.
28.   }}
29.   return 0;
30.}
```

```
1.// function to insert an element in a circular queue
2.void enqueue(int element)
3.{
4.   if(front==-1 && rear==-1)   // condition to check queue is empty
5.   {
6.      front=0;
7.      rear=0;
8.      queue[rear]=element;
9.   }
10.   else if((rear+1)%max==front)  // condition to check queue is full
11.   {
12.      printf("Queue is overflow..");
13.   }
14.   else
15.   {
16.      rear=(rear+1)%max;      // rear is incremented
17.      queue[rear]=element;     // assigning a value to the queue at the rear position
18.   }
19.}
```

```
1.// function to delete the element from the queue
2.int dequeue()
3.{
4.   if((front==-1) && (rear==-1))  // condition to check queue is empty
5.   {
6.      printf("\nQueue is underflow..");
7.   }
8. else if(front==rear)
9.{
10.   printf("\nThe dequeued element is %d", queue[front]);
11.   front=-1;
12.   rear=-1;
13.}
14.else
15.{
16.    printf("\nThe dequeued element is %d", queue[front]);
17.   front=(front+1)%max;
18.}
19.}
```

```
1.# define max 6
2.int queue[max];  // array declaration
3.int front=-1;
4.int rear=-1;
```

Python program to implement a circular queue using an array

Another approach

```
class CircularQueue():
    # constructor
     def __init__(self, size): # initializing the class
        self.size = size
                # initializing queue with none
        self.queue = [None for i in range(size)]
        self.front = self.rear = -1
    def enqueue(self, data):
              # condition if queue is full
        if ((self.rear + 1) % self.size == self.front
            print(" Queue is Full\n")
                # condition for empty queue
        elif (self.front == -1):
            self.front = 0
            self.rear = 0
            self.queue[self.rear] = data
        else:
                    # next position of rear
            self.rear = (self.rear + 1) % self.size
            self.queue[self.rear] = data

    def dequeue(self):
        if (self.front == -1): # condition for empty
            print ("Queue is Empty\n")
               # condition for only one element
        elif (self.front == self.rear):
            temp=self.queue[self.front]
            self.front = -1
            self.rear = -1
            return temp
        else:
            temp = self.queue[self.front]
            self.front = (self.front + 1) % self.size
            return temp
```

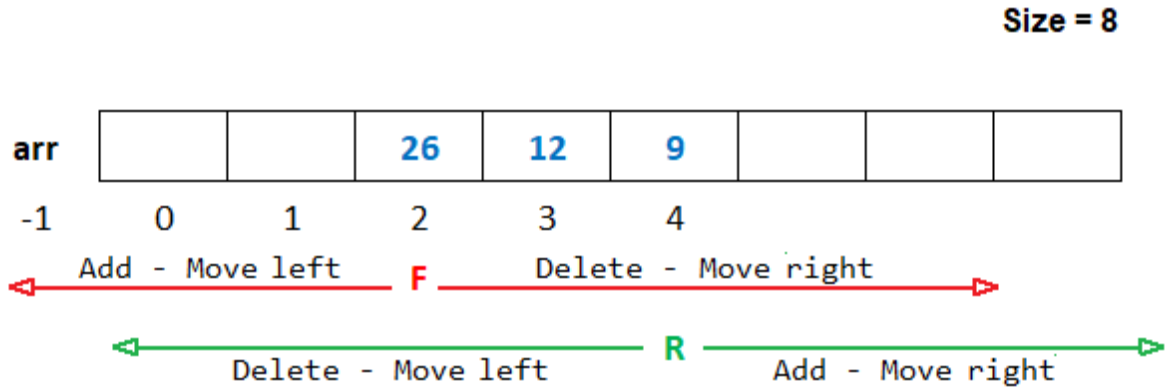
```
def display(self):

    # condition for empty queue
    if(self.front == -1):
        print ("Queue is Empty")

    elif (self.rear >= self.front):
        print("Elements in the circular queue are:",
                                              end = " ")
        for i in range(self.front, self.rear + 1):
            print(self.queue[i], end = " ")
        print ()

    else:
        print ("Elements in Circular Queue are:",
                                             end = " ")
        for i in range(self.front, self.size):
            print(self.queue[i], end = " ")
        for i in range(0, self.rear + 1):
            print(self.queue[i], end = " ")
        print ()

    if ((self.rear + 1) % self.size == self.front):
        print("Queue is Full")
```

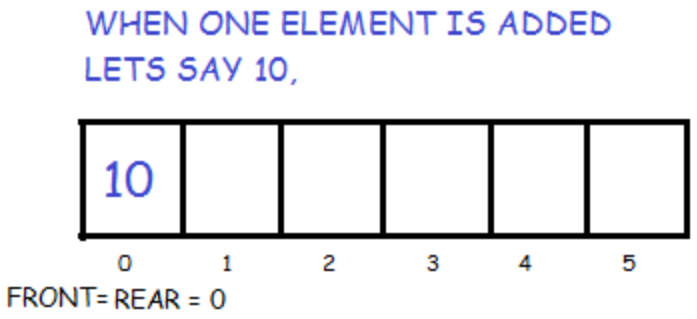
```
# Driver Code
ob = CircularQueue(5)
ob.enqueue(14)
ob.enqueue(22)
ob.enqueue(13)
ob.enqueue(-6)
ob.display()
print ("Deleted value = ",
ob.dequeue())
print ("Deleted value = ",
ob.dequeue())
ob.display()
ob.enqueue(9)
ob.enqueue(20)
ob.enqueue(5)
ob.display()
```

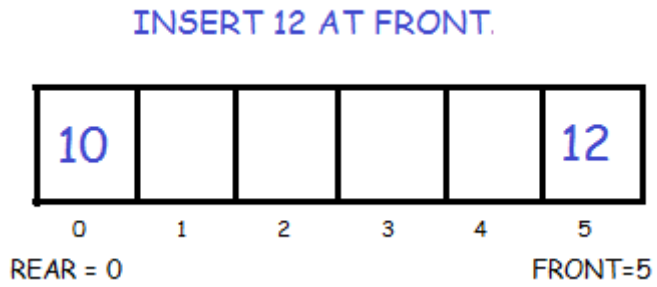
Double ended queue

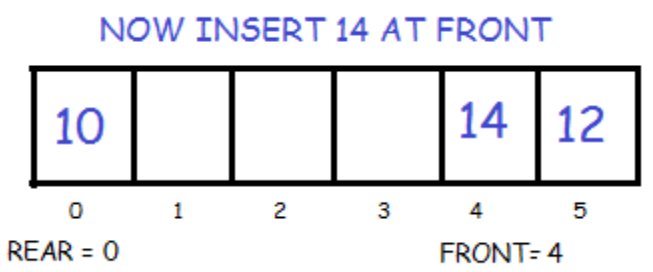
Double ended queue

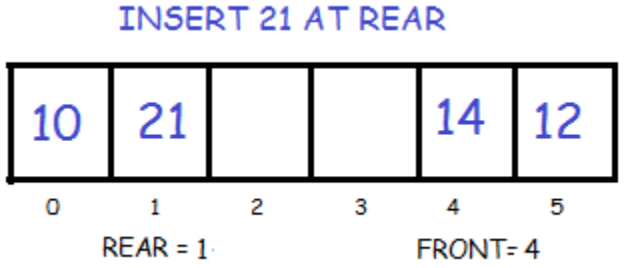
- A Double Ended Queue in C, also known as Deque, is a queue data structure in which insertion and deletion can be done from bot left and right ends.

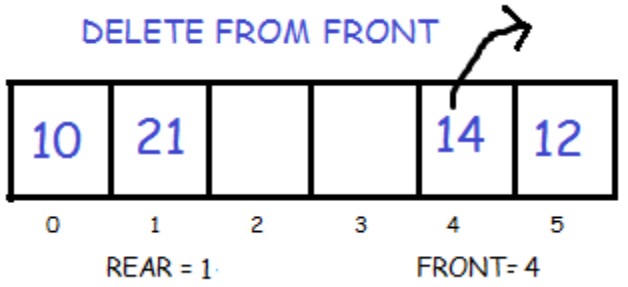


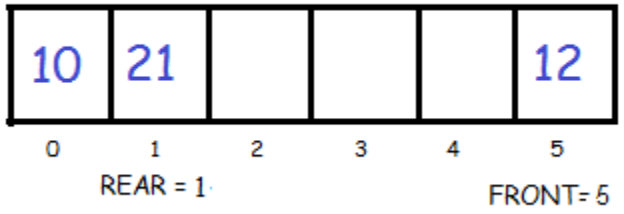












- **Operation on Circular Queue**
- There are four operations possible on the double ended queue.
- Add Rear When we add an element from the rear end.
- Delete Rear When we delete an element from the rear end.
- Add Front When we add an element from the front end.
- Delete Front When we delete an element from the front end.

Add Rear Operation in Deque

```
if(te==size)
{
        printf("Queue is full\n");
}
else
{
        R=(R+1)%size;
        arr[R]=new_item;
   te=te+1;
}
```

Delete Rear Operation in Deque

```
if(te==0)
{
        printf("Queue is empty\n");
}
else
{
        if(R==-1)
        {
                R=size-1;
        }
   printf("Number Deleted From Rear End = %d",arr[R]);
   R=R-1;
   te=te-1;
}
```

te is a variable that keeps the total number of elements present in the array

Add Front Operation in Deque

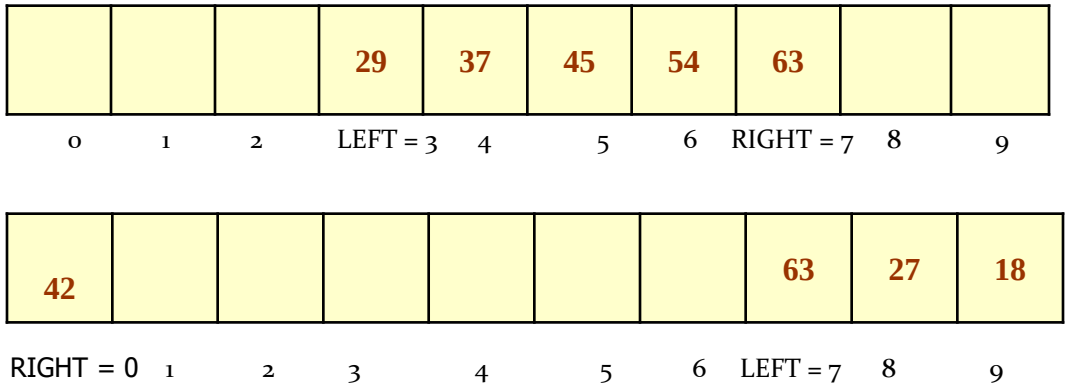
```
if(te==size)
{
   printf("Queue is full");
}
else
{
   if(F==0)
   {
      F=size-1;
   }
   else
   {
      F=F-1;
   }
   arr[F]=new_item;
   te=te+1;
}
```

Delete Front Operation in Deque

```
if(te==0)
{
   printf("Queue is empty");
}
else
{
   printf("Number Deleted From Front End = %d",arr[F]);
   F=(F+1)%size;
   te=te-1;
}
```

Deque variants

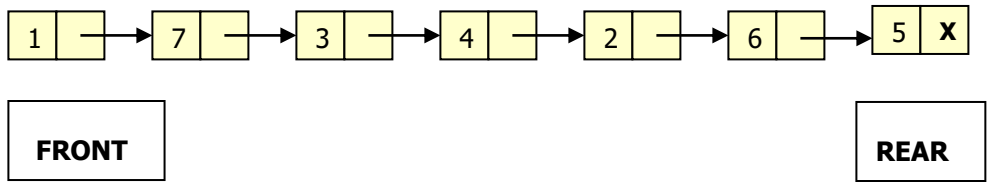
- There are two variants of deques:
  - *Input restricted deque:* In this dequeue insertions can be done only at one of the ends while deletions can be done from both the ends.
  - *Output restricted deque:* In this dequeue deletions can be done only at one of the ends while insertions can be done on both the ends.



Linked List

Queue by Linked List

- Using a singly linked list to hold queue elements, Using FRONT pointer pointing the start element, Using REAR pointer pointing to the last element.
- Insertions is done at the rear end using REAR pointer, Deletions is done at the front end using FRONT pointer
- If FRONT = REAR = NULL, then the queue is empty.



Inserting an Element in a Linked Queue

```
Algorithm to insert an element in a linked queue

Step 1: Allocate memory for the new node and
        name the pointer as PTR
Step 2: SET PTR->DATA = VAL
Step 3: IF FRONT = NULL, then
            SET FRONT = REAR = PTR
            SET FRONT->NEXT = REAR->NEXT = NULL
        ELSE
            SET REAR->NEXT = PTR
            SET REAR = PTR
            SET REAR->NEXT = NULL
        [END OF IF]
Step 4: END

Time complexity: O(1)
```

Deleting an Element from a Linked Queue

```
Algorithm to delete an element from a
linked queue

Step 1: IF FRONT = NULL, then
          Write "Underflow"
          Go to Step 5
        [END OF IF]
Step 2: SET PTR = FRONT
Step 3: FRONT = FRONT->NEXT
Step 4: FREE PTR
Step 5: END

Time complexity: O(1)
```

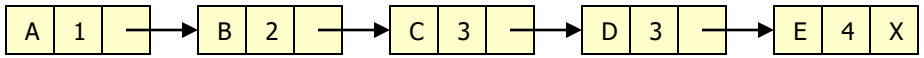
Applications of Queues

Priority Queues

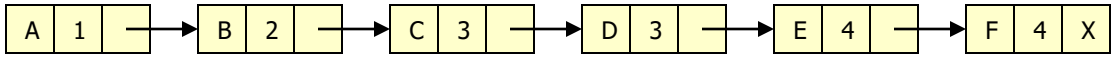
- A priority queue is a queue in which each element is assigned a priority.
- The priority of elements is used to determine the order in which these elements will be processed.
- The general rule of processing elements of a priority queue can be given as:
  - An element with higher priority is processed before an element with lower priority
  - Two elements with same priority are processed on a first come first served (FCFS) basis
- Priority queues are widely used in operating systems to execute the highest priority process first.
- In computer's memory priority queues can be represented using arrays or linked lists.

Linked List Representation of Priority Queues

- When a priority queue is implemented using a linked list, then every node of the list contains three parts: (1) the information or data part, (ii) the priority number of the element, (iii) and address of the next element.
- If we are using a sorted linked list, then element having higher priority will precede the element with lower priority.



Priority queue after insertion of a new node (F, 4)



Array Representation of Priority Queues

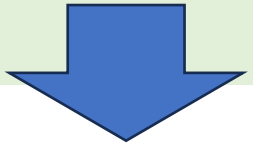
- When arrays are used to implement a priority queue, then a separate queue for each priority number is maintained.
- Each of these queues will be implemented using circular arrays or circular queues. Every individual queue will have its own FRONT and REAR pointers.
- We can use a two-dimensional array for this purpose where each queue will be allocated same amount of space.
- Given the front and rear values of each queue, a two dimensional matrix can be formed.

Applications of Queues

1. Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
2. Queues are used to transfer data asynchronously e.g., pipes, file IO, sockets.
3. Queues are used as buffers on MP3 players and portable CD players, iPod playlist.
5. Queues are used in Playlist for jukebox to add songs to the end, play from the front of the list.

More

Queue using Stacks



By making enQueue operation costly

enQueue(q, x):
•While stack1 is not empty, push everything from stack1 to stack2.
•Push x to stack1 (assuming size of stacks is unlimited).
•Push everything back to stack1.

deQueue(q):
•If stack1 is empty then error
•Pop an item from stack1 and return it

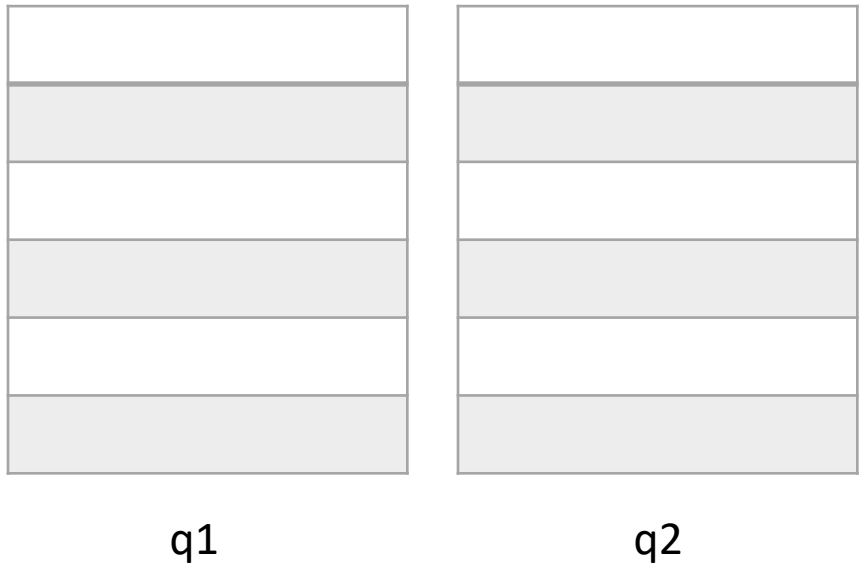
By making deQueue operation costly

enQueue(q, x)
1) Push x to stack1 (assuming size of stacks is unlimited).
Here time complexity will be O(1)

deQueue(q)
1) If both stacks are empty then error.
2) If stack2 is empty
While stack1 is not empty, push everything from stack1 to stack2.
3) Pop the element from stack2 and return it.
Here time complexity will be O(n)

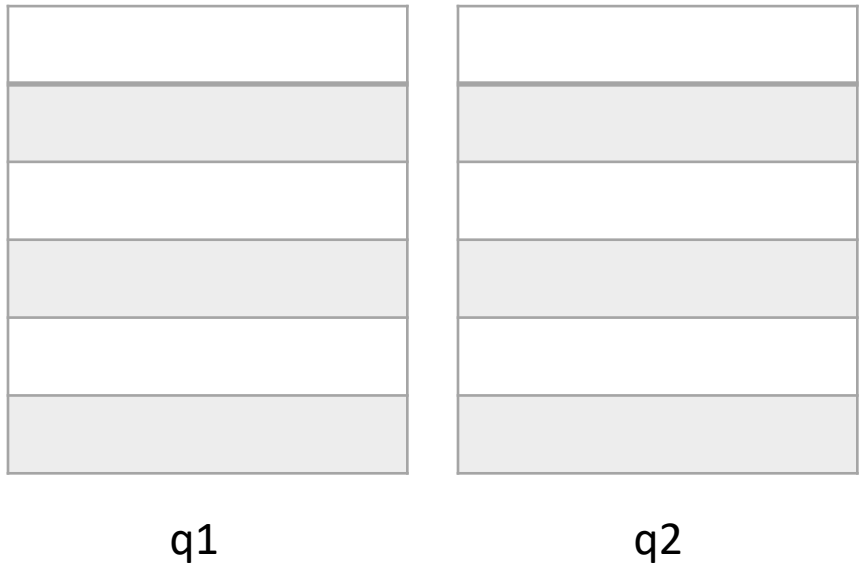
Implement Stack using Queues [**Pop Heavy**]

- Follow the below steps to implement the **push(s, x)** operation:
  - Enqueue x to q1 (assuming the size of q1 is unlimited).
- Follow the below steps to implement the **pop(s)** operation:
  - One by one dequeue everything except the last element from q1 and enqueue to q2.
  - Dequeue the last item of q1, the dequeued item is the result, store it.
  - Swap the names of q1 and q2
  - Return the item stored in step 2.



Implement Stack using Queues [**push** heavy]

- Follow the below steps to implement the **push(s, x)** operation:
  - Enqueue x to q2.
  - One by one dequeue everything from q1 and enqueue to q2.
  - Swap the queues of q1 and q2.
- Follow the below steps to implement the **pop(s)** operation:
  - Dequeue an item from q1 and return it.



Any questions ?