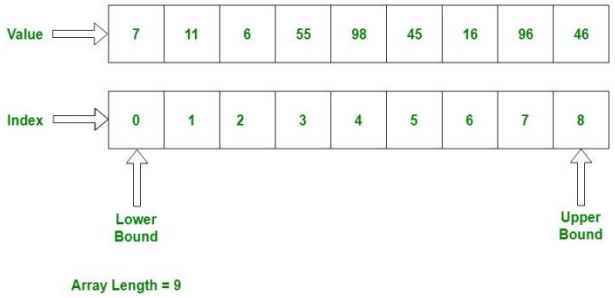
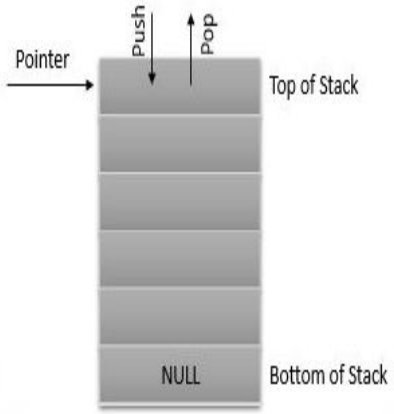
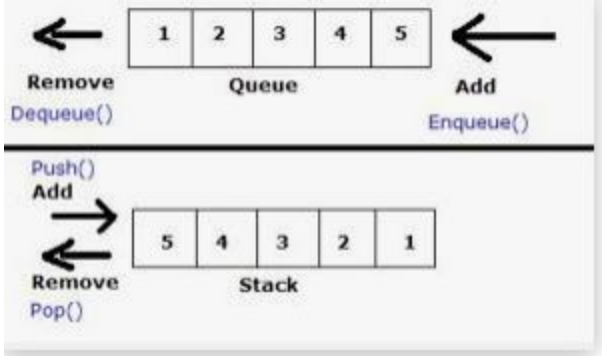
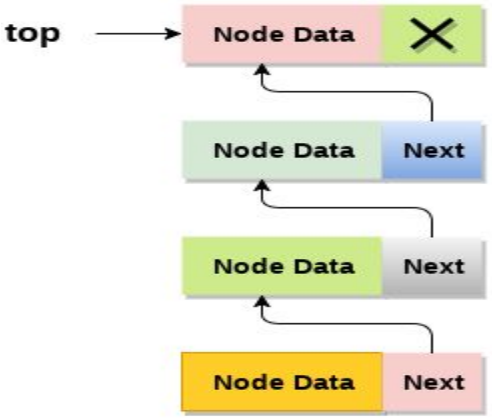
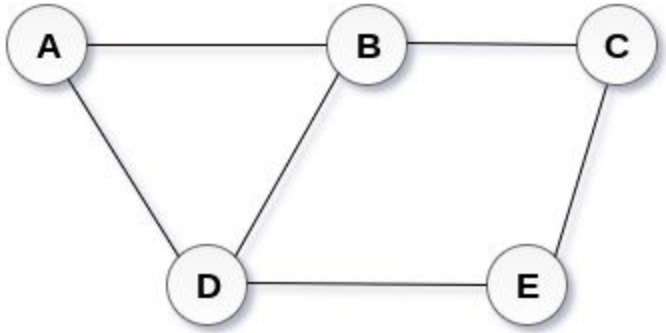

Value
7
11
6
55
98
45
16
96
46
Index
0
1
2
3
4
5
6
7
8
Lower Bound
Upper Bound
Array Length = 9

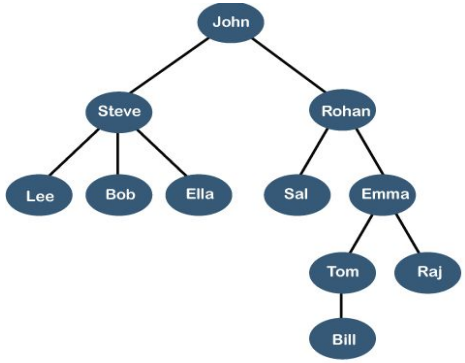

Pointer
Push
Pop
Top of Stack
NULL
Bottom of Stack


1
2
3
4
5
Remove
Dequeue()
Queue
Add
Enqueue()
Push()
Add
5
4
3
2
1
Remove
Pop()
Stack

QUEUE


top
Node Data
Node Data
Next
Node Data
Next
Node Data
Next


A
B
C
D
E

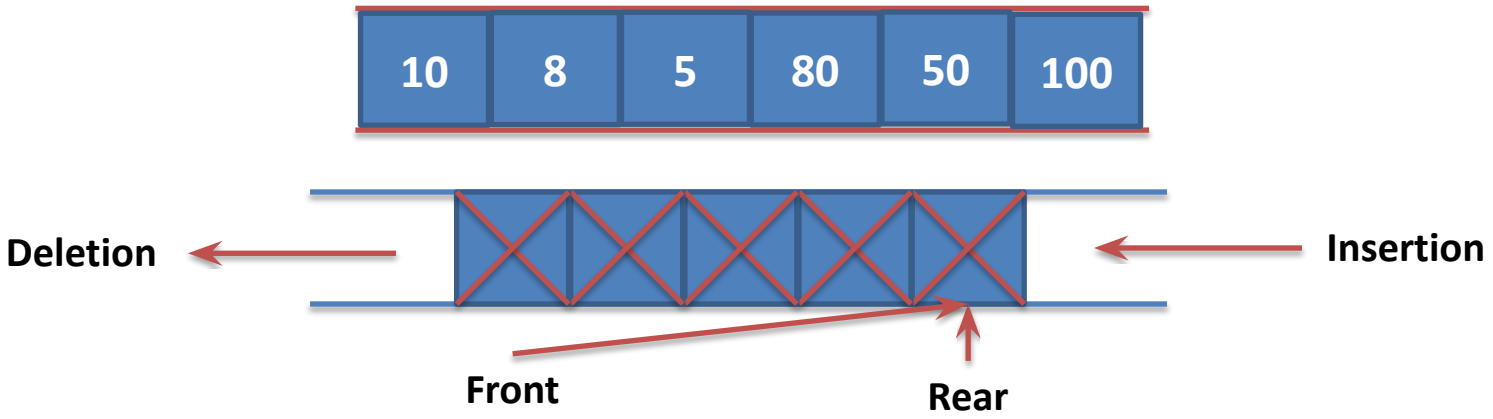

John
Steve
Rohan
Lee
Bob
Ella
Sal
Emma
Tom
Raj
Bill

Queue

- Introduction to Queue
- Applications of Queue
- Representation of Queue using Array
- Implementation of Queue Operations
- Understanding Circular Queue
- Introduction to Priority Queue and its Operations
- Introduction to Dqueue and its Operation.

Queue

- A linear list which permits **deletion** to be performed **at one** end of the list and **insertion at the other end** is called **queue**.
- The information in such a list is processed **FIFO (first in first out)** or **FCFS (first come first served)** manner.
- **Front** is the end of queue from that deletion is to be performed.
- **Rear** is the end of queue at which new element is to be inserted.
- Insertion operation is called **Enqueue** and deletion operation is called **Dequeue**.

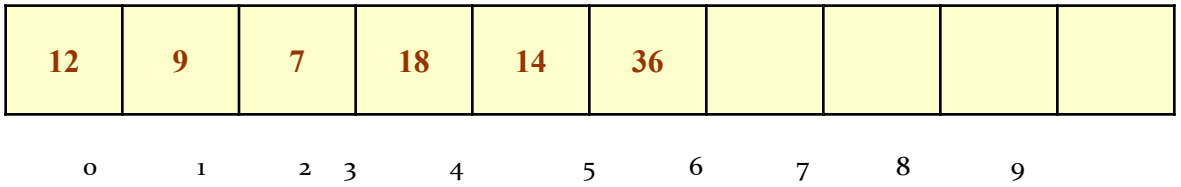


Queue Applications

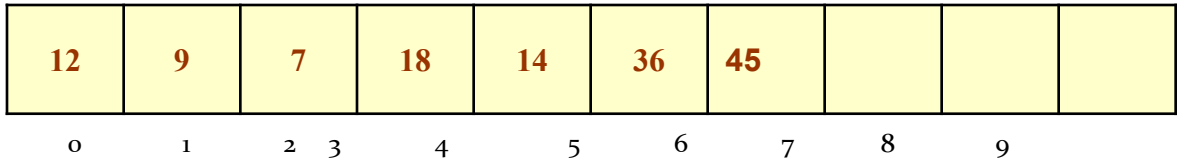
- Queue of people at any service point such as ticketing etc.
- Queue of air planes waiting for landing instructions.
- **Queue of processes** in OS.
- Queue is also used by Operating systems for **Job Scheduling**.
- When a **resource is shared** among multiple consumers. E.g., in case of printers the first one to be entered is the first to be processed.
- When **data is transferred asynchronously** (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.
- Queue is used in **BFS (Breadth First Search)** algorithm. It helps in traversing a tree or graph.
- Queue is used in networking to **handle congestion**.

Array Representation of Queues

- Queues can be easily represented using linear 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

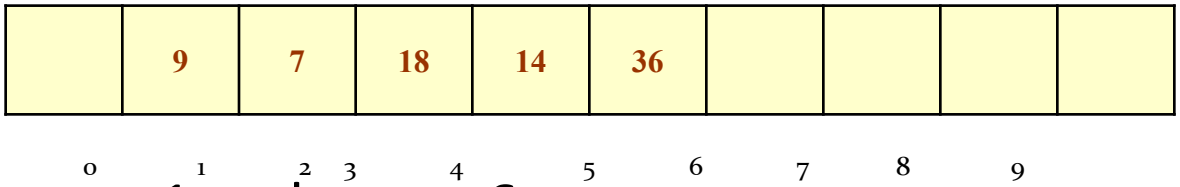


- Here, front = 0 and rear = 5.
- If we want to add one more value in the list say with value 45, then rear would be incremented by 1 and the value would be stored at the position pointed by rear.



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.

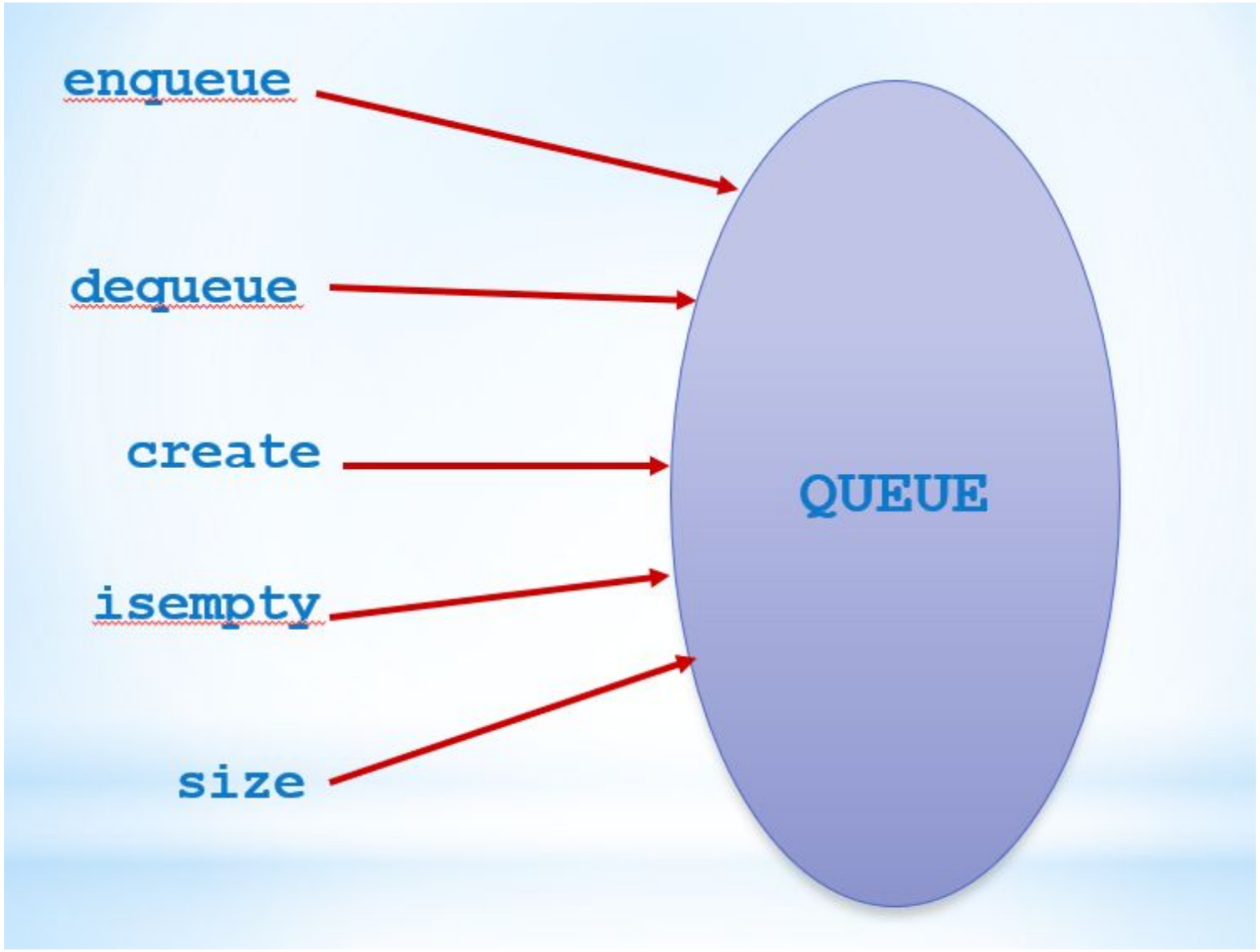


- Here, front = 1 and rear = 6.

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.

Operations on Queues



Procedure: Enqueue (Q, F, R, N,Y)

- This procedure inserts **Y** at rear end of Queue.
- **Queue** is represented by a vector **Q** containing **N** elements.
- **F** is pointer to the front element of a queue.
- **R** is pointer to the rear element of a queue.

```
1. [Check for Queue Overflow]
    If  R >= N
    Then   write ('Queue Overflow')
        Return
2. [Increment REAR pointer]
    R □ R + 1
3. [Insert element]
    Q[R] □ Y
4. [Is front pointer properly set?]
        IF  F=0
        Then    F □ 1
        Return
```

Procedure: Enqueue (Q, F, R, N,Y)

```
1. [Check for Queue Overflow]
    If  R >= N
    Then    write ('Queue Overflow')
        Return
2. [Increment REAR pointer]
    R □ R + 1
3. [Insert element]
    Q[R] □ Y
4. [Is front pointer properly set?]
        IF  F=0
        Then    F □ 1
        Return
```

N=3, R=0, F=0

F = 1

R = 3

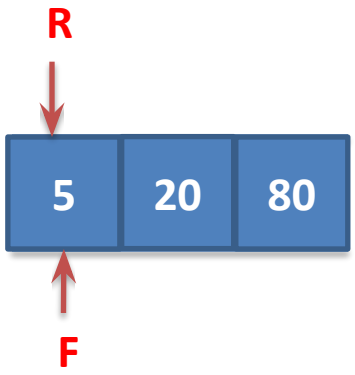
Enqueue (Q, F, R, N=3,Y=5)

Enqueue (Q, F, R, N=3,Y=20)

Enqueue (Q, F, R, N=3,Y=80)

Enqueue (Q, F, R, N=3,Y=3)

Queue Overflow



Procedure: Dequeue (Q, F, R)

- This function **deletes & returns** an element **from front end** of the Queue.
- **Queue** is represented by a vector **Q** containing **N** elements.
- **F** is pointer to the **front** element of a queue.
- **R** is pointer to the **rear** element of a queue.

```
1. [Check for Queue Underflow]
    If    F = 0
    Then write ('Queue Underflow')
      Return(0)
2. [Delete element]
    Y □ Q[F]
```

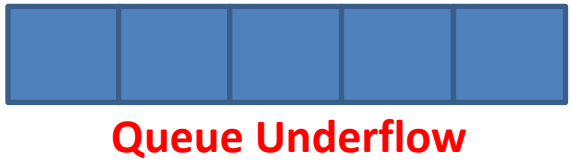
```
3. [Is Queue Empty?]
    If    F = R
    Then F □ R □ 0
    Else F □ F + 1
4. [Return Element]
    Return (Y)
```

Procedure: Dequeue (Q, F, R)

```
1. [Check for Queue Underflow]
    If    F = 0
    Then write ('Queue Underflow')
      Return(0)
2. [Delete element]
    Y □ Q[F]
3. [Is Queue Empty?]
    If    F = R
    Then F □ R □ 0
    Else F □ F + 1
4. [Return Element]
    Return (Y)
```

Case No 1:

F=0, R=0

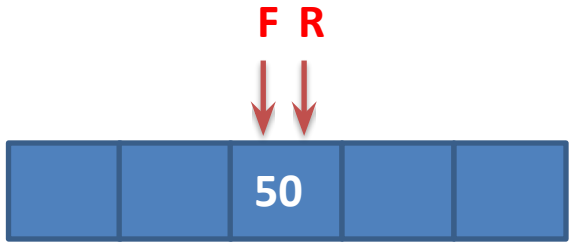


Queue Underflow

Case No 2:

F=3, R=3

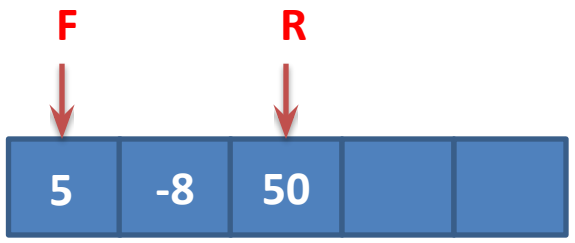
F=0, R=0



Case No 3:

F=1, R=3

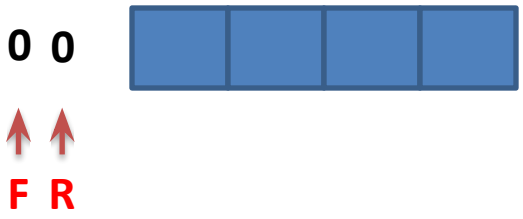
F=2, R=3



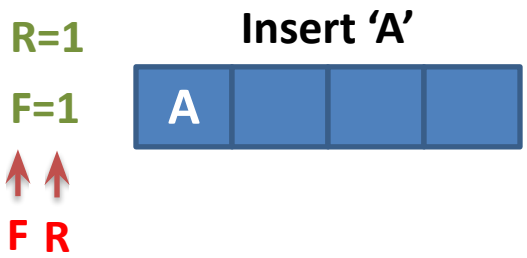
Example of Queue Insert / Delete

Perform following operations on queue with size 4 & draw queue after each operation
Insert ‘A’ | Insert ‘B’ | Insert ‘C’ | Delete ‘A’ | Delete ‘B’ | Insert ‘D’ | Insert ‘E’

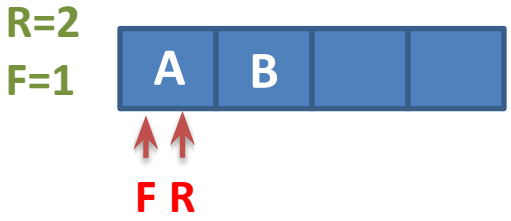
Empty Queue



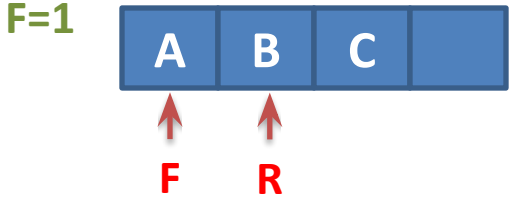
Insert ‘A’



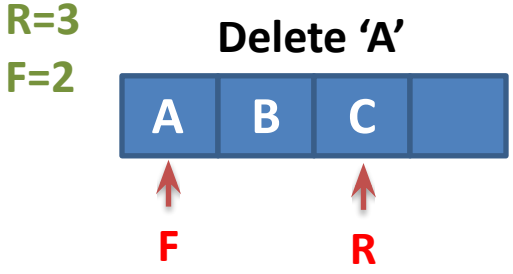
Insert ‘B’



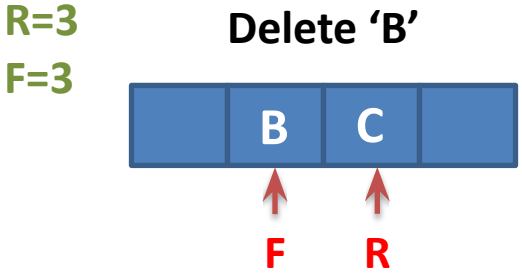
Insert ‘C’



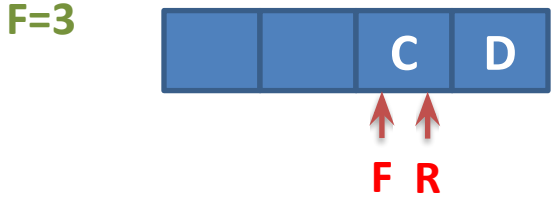
Delete ‘A’



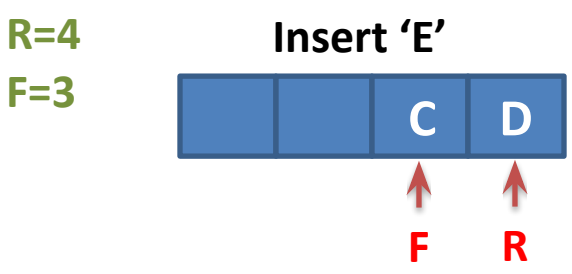
Delete ‘B’



Insert ‘D’



Insert ‘E’



(R=4) >= (N=4) (Size of Queue)

Queue Overflow

Queue Overflow, but space is there with Queue, this leads to the memory wastage

Implementation of Queue using Array - C Program

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#define SIZE 10

void enQueue(int);
void deQueue();
void display();

int queue[SIZE], front = -1, rear = -1;
```

Implementation of Queue using Array - C Program

```
void main()
{
  int value, choice;

  while(1){
    printf("\n\n***** MENU *****\n");
    printf("1. Insertion\n2. Deletion\n3. Display\n4. Exit");
    printf("\nEnter your choice: ");
    scanf("%d",&choice);
```

Implementation of Queue using Array - C Program

```
      switch(choice){
       case 1: printf("Enter the value to be insert: ");
             scanf("%d",&value);
             enQueue(value);
             break;
       case 2: deQueue();
             break;
       case 3: display();
             break;
       case 4: exit(0);
       default: printf("\nWrong selection!!! Try again!!!");
      }
   }
}
```

Implementation of Queue using Array - C Program

```
void enQueue(int value){
  if(rear == SIZE-1)
    printf("\nQueue is Full!!! Insertion is not possible!!!");
  else{
    if(front == -1)
     front = 0;
    rear++;
    queue[rear] = value;
    printf("\nInsertion success!!!");
  }
}
```

Implementation of Queue using Array - C Program

```
void deQueue(){
  if(front == rear)
    printf("\nQueue is Empty!!! Deletion is not possible!!!");
  else{
    printf("\nDeleted : %d", queue[front]);
    front++;
    if(front == rear)
     front = rear = -1;
  }
}
```

Implementation of Queue using Array - C Program

```
void display(){
  if(rear == -1)
    printf("\nQueue is Empty!!!");
  else{
    int i;
    printf("\nQueue elements are:\n");
    for(i=front; i<=rear; i++)
      printf("%d\t",queue[i]);
  }
}
```

Implementation of Queue using structure - C Program

```
#include<stdio.h>
#define QSIZE 5
struct queue{
        int item[QSIZE];
        int front,rear;
};
void enqueue(struct queue *pq,int n){            //function for insert
       if(pq->rear==QSIZE){
          printf("\nqueue is full\n");
       }
       else{
          pq->item[pq->rear]=n;
          pq->rear++;
       }
}
```

Implementation of Queue using structure - C Program

```
//function to delete
void dequeue(struct queue *pq){
        int i=pq->front;
        if(pq->front==pq->rear){
           printf("\nqueue is empty\n");
        }
        else{
           for(i=pq->front;i<(pq->rear-1);i++){
               pq->item[i]=pq->item[i+1];
           }
           pq->rear--;
        }
}
```

Implementation of Queue using structure - C Program

```
//function to display
void display(struct queue *pq){
        int i=pq->front;
        if(pq->front==pq->rear){
          printf("\nqueue is empty\n");
        }
        else{
        printf("\nFront\t");
          for(i=pq->front;i<pq->rear;i++){
              printf("%d\t",pq->item[i]);
          }
        }
        printf("\tRear");
        printf("\n\n\n");
}
```

Implementation of Queue using structure - C Program

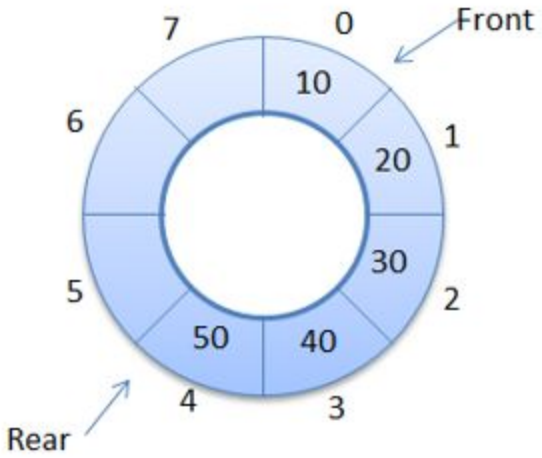
```
//function main
void main(){
        struct queue q;
        int ch,n;
        q.front=0;
        q.rear=0;

        do{
          printf("\n1.enqueue\n2.dequeue\n3.display\n4.exit\n");
          printf("Enter your choice:\n");
          scanf("%d",&ch);
          switch(ch){
             case 1: printf("\nenter integer  to enqueue:");
                     scanf("%d",&n);
                     enqueue(&q,n);
                     break;
```

Implementation of Queue using structure - C Program

```
            case 2: dequeue(&q);
                    display(&q);
                    break;

            case 3: display(&q);
                    break;
          }
        }while(ch!=4);
}
```

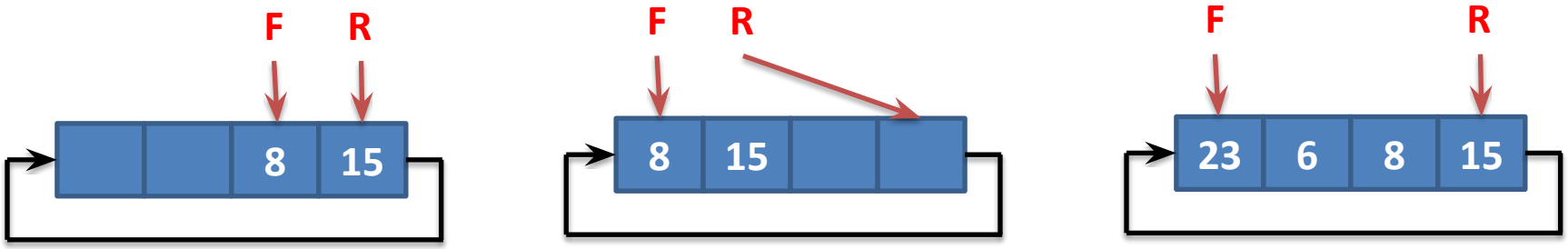
Circular Queues

- A more suitable method of representing simple queue which prevents an excessive use of memory is to **arrange the elements** Q[1], Q[2]….,Q[n] **in a circular fashion** with Q[1] following Q[n], this is called **circular queue.**
- In circular queue the last node is connected back to the first node to make a circle.
- Circular queue is a linear data structure. It follows **FIFO** principle.
- It is also called as **“Ring buffer”**.



Procedure: CQINSERT (F, R, Q, N, Y)

- This procedure inserts Y at rear end of the Circular Queue.
- Queue is represented by a vector Q containing N elements.
- F is pointer to the front element of a queue.
- R is pointer to the rear element of a queue.




```
1. [Reset Rear Pointer]
   If     R = N
   Then   R □  1
   Else   R □ R + 1
2. [Overflow]
   If     F=R
   Then   Write('Overflow')
          Return
```

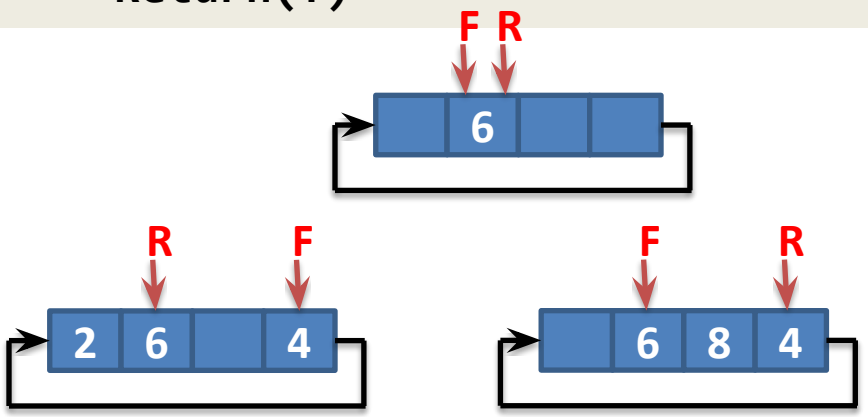
```
3. [Insert element]
   Q[R] □ Y
4. [Is front pointer
   properly set?]
       IF   F=0
       Then     F □ 1

       Return
```

Procedure: CQDELETE (F, R, Q, N)

- This function deletes & returns an element from front end of the Circular Queue.
- Queue is represented by a vector Q containing N elements.
- F is pointer to the front element of a queue.
- R is pointer to the rear element of a queue.

```
1. [Underflow?]
   If     F = 0
   Then   Write('Underflow')
          Return(0)
2. [Delete Element]
    Y □ Q[F]
3. [Queue Empty?]
   If     F = R
   Then   F □ R □  0
          Return(Y)
```

```
4. Increment Front Pointer]
    IF F = N
    Then F □ 1
    Else F □ F + 1
    Return(Y)
```



Example of CQueue Insert / Delete

Perform following operations on Circular queue with size 4 & draw queue after each operation **Insert 'A' | Insert 'B' | Insert 'C' | Delete 'A' | Delete 'B' | Insert 'D' | Insert 'E'**

Empty Queue

0 0

F R

Insert 'C'

R=3

F=1

F R

Insert 'D'

R=4

F=3

F R

Insert 'A'

R=1

F=1

F R

Delete 'A'

R=3

F=2

F R

Insert 'E'

R=1

F=3

F R

Insert 'B'

R=2

F=1

F R

Delete 'B'

R=3

F=3

F R

Implementation of Circular Queue using Array - C Program

```
# include<stdio.h>
# define MAX 5

int cqueue_arr[MAX];
int front = -1;
int rear = -1;
```

Implementation of Circular Queue using Array - C Program

```
 /*Begin of insert*/
void insert(int item){
        if((front == 0 && rear == MAX-1) || (front == rear+1))  {
                printf("Queue Overflow \n");
                return;
        }
        if (front == -1)  /*If queue is empty */
        {
                front = 0;
                rear = 0;
        }
        else
        {
                if(rear == MAX-1)    /*rear is at last position of queue */
                        rear = 0;
                else
                        rear = rear+1;
        }
        cqueue_arr[rear] = item ;
}
/*End of insert*/
```

Implementation of Circular Queue using Array - C Program

```
 /*Begin of del*/
void del() {
        if (front == -1) {
                printf("Queue Underflow\n");
                return ;
        }
        printf("Element deleted from queue is : %d\n",cqueue_arr[front]);
        if(front == rear) /* queue has only one element */
        {
                front = -1;
                rear=-1;
        }
        else
        {
                if(front == MAX-1)
                        front = 0;
                else
                        front = front+1;
        }
}
/*End of del() */
```

Implementation of Circular Queue using Array - C Program

```
/*Begin of display*/
void display() {
        int front_pos = front,rear_pos = rear;
        if(front == -1)  {
                printf("Queue is empty\n");
                return;
        }
        printf("Queue elements :\n");
        if( front_pos <= rear_pos )
                while(front_pos <= rear_pos)
                {
                        printf("%d ",cqueue_arr[front_pos]);
                        front_pos++;
                }
        else
        {
                while(front_pos <= MAX-1)
                {
                        printf("%d ",cqueue_arr[front_pos]);
                        front_pos++;
                }
```

Implementation of Circular Queue using Array - C Program

```
            front_pos = 0;
            while(front_pos <= rear_pos)
            {
                  printf("%d ",cqueue_arr[front_pos]);
                  front_pos++;
            }
      }
      printf("\n");
}
/*End of display*/
```

Implementation of Circular Queue using Array - C Program

```
/*Begin of main*/
int main()
{
        int choice,item;
        do
        {
                printf("1.Insert\n");
                printf("2.Delete\n");
                printf("3.Display\n");
                printf("4.Quit\n");

                printf("Enter your choice : ");
                scanf("%d",&choice);
```

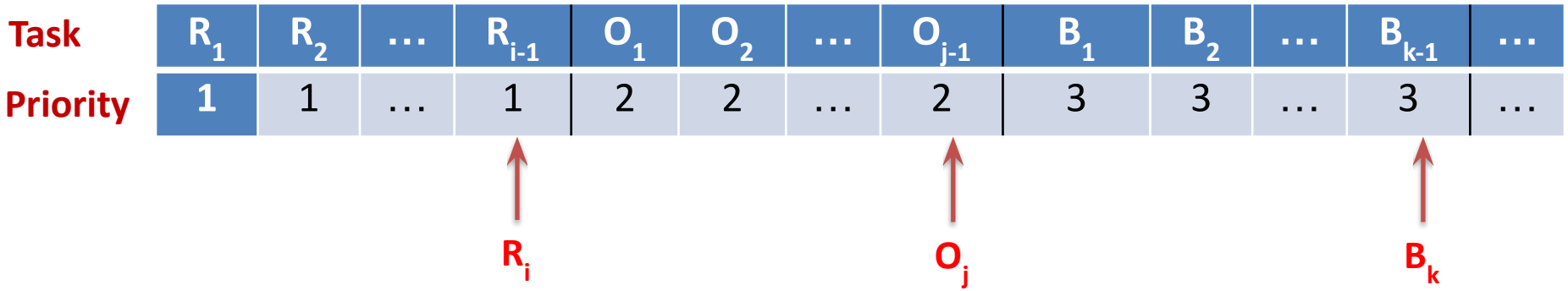
Implementation of Circular Queue using Array - C Program

```
        switch(choice)        {
                case 1 :
                        printf("Input the element for insertion in queue : ");
                        scanf("%d", &item);
                        insert(item);
                        break;
                case 2 :
                        del();
                        break;
                case 3:
                        display();
                        break;
                case 4:
                        break;
                        default:
                        printf("Wrong choice\n");
        }
    }while(choice!=4);
    return 0;
}       /*End of main*/
```

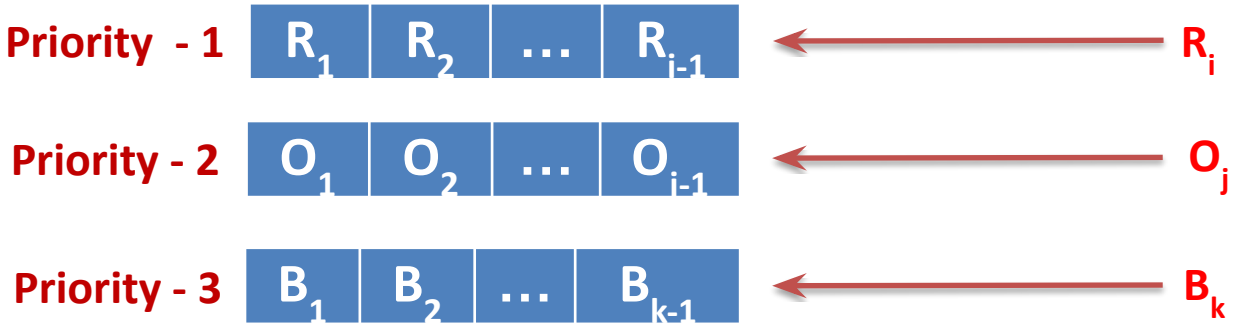
Priority Queue

- A queue in which we are able to **insert & remove items** from **any position based on** some property (such as **priority** of the task to be processed) is often referred as **priority queue**. So that it does not support FIFO(First In First Out) structure.
- Below fig. represent a priority queue of jobs waiting to use a computer.
- Priorities are attached with each Job
  - **Priority 1** indicates **Real Time Job**
  - **Priority 2** indicates **Online Job**
  - **Priority 3** indicates **Batch Processing Job**
- Therefore if a job is initiated with priority i, it is inserted immediately at the end of list of other jobs with priorities i.
- Here jobs are always removed from the front of queue

Priority Queue



Priority Queue viewed as a single queue with insertion allowed at any position



Priority Queue viewed as a Viewed as a set of queue

Priority Queue

- A priority queue is a special type of queue in which each element is associated with a priority and is served according to its priority.
- If elements with the same priority occur, they are served according to their order in the queue.
- Generally, the value of the element itself is considered for assigning the priority.
- For example, The element with the highest value is considered as the highest priority element.
- However, in other cases, we can assume the element with the lowest value as the highest priority element.

Difference between Priority Queue and Normal Queue

- In a queue, the **first-in-first-out rule** is implemented whereas, in a priority queue, the values are removed **on the basis of priority**. The element with the highest priority is removed first.

- **Basic Operations**
  - **insert / enqueue** – add an item to the rear of the queue.
  - Whenever an element is inserted into queue, priority queue inserts the item according to its order. Here we're assuming that data with high value has low priority.
  - **remove / dequeue** – remove an item from the front of the queue.

Implementation of Priority Queue - C Program

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>
#define MAX 6
int intArray[MAX];
int itemCount = 0;

int peek(){
   return intArray[itemCount - 1];
}
```

Implementation of Priority Queue - C Program

```
bool isFull(){
   return itemCount == MAX;
}

bool isEmpty(){
   return itemCount == 0;
}

int size(){

   return itemCount;

}

int removeData(){

   return intArray[--itemCount];

}
```

Implementation of Priority Queue - C Program

```
void insert(int data){
   int i = 0;
   if(!isFull()){          // if queue is empty, insert the data
      if(itemCount == 0){
         intArray[itemCount++] = data;
      }else{          // start from the right end of the queue

         for(i = itemCount - 1; i >= 0; i-- ){
            // if data is larger, shift existing item to right end
            if(data > intArray[i]){
               intArray[i+1] = intArray[i];
            }else{
               break;
            }
         }
         // insert the data
         intArray[i+1] = data;
         itemCount++;
      }
   }
}
```

Implementation of Priority Queue - C Program

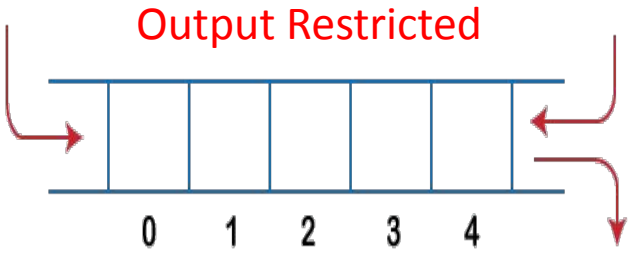
```
int main() {
   insert(3);
   insert(5);
   insert(9);
   insert(1);
   insert(12);
   insert(15);

   if(isFull()){
      printf("Queue is full!\n");
   }

   // remove one item
   int num = removeData();
   printf("Element removed: %d\n",num);
```

Implementation of Priority Queue - C Program

```
   // insert more items
   insert(16);
   // As queue is full, elements will not be inserted.
   insert(17);
   insert(18);
   printf("Element at front: %d\n",peek());

   printf("----------------------\n");
   printf("index : 5 4 3 2  1  0\n");
   printf("----------------------\n");
   printf("Queue:  ");
   while(!isEmpty()){
      int n = removeData();
      printf("%d ",n);
   }
}
```

DQueue

- A **DQueue (double ended queue)** is a linear list in which insertion and deletion are performed **from the either end of the structure**.
- There are two variations of Dqueue
  - ***Input restricted dqueue***- allows insertion at only one end
  - ***Output restricted dqueue***- allows deletion from only one end



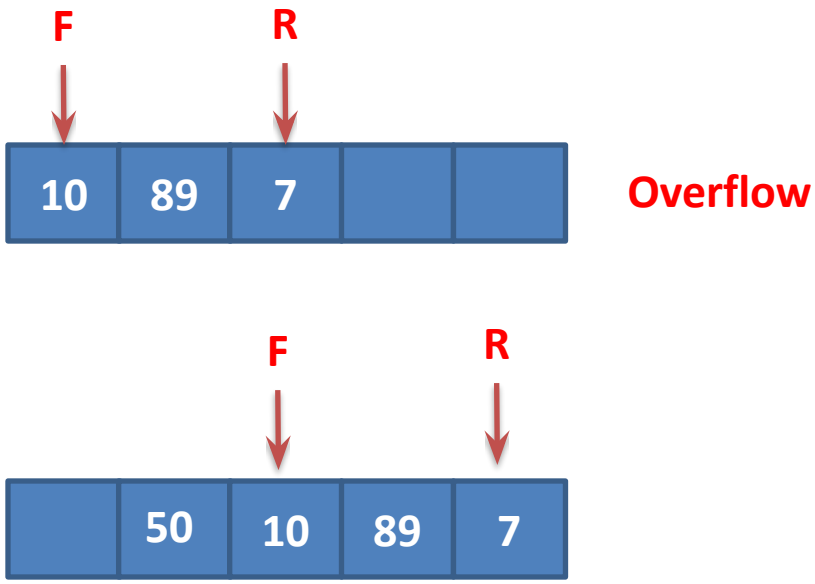
Dqueue Algorithms

- DQINSERT_REAR is same as QINSERT (Enqueue)
- DQDELETE_FRONT is same as QDELETE (Dequeue)
- DQINSERT_FRONT
- DQDELETE_REAR

Procedure: DQINSERT_FRONT (Q,F,R,N,Y)

- This procedure inserts Y at front end of the Queue.
- Queue is represented by a vector Q containing N elements.
- F is pointer to the front element of a queue.
- R is pointer to the rear element of a queue.

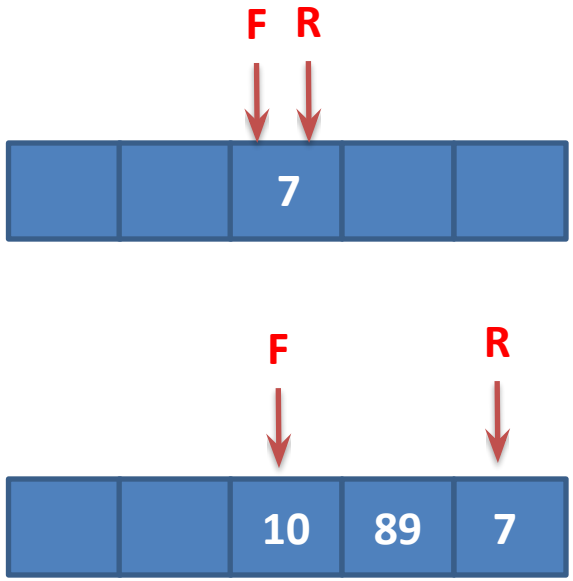
```
1. [Overflow?]
   If     F = 0
   Then   Write('Empty')
          Return
   If     F = 1
   Then   Write('Overflow')
          Return
2. [Decrement front Pointer]
    F □ F - 1
3. [Insert Element?]
   Q[F] □ Y
   Return
```



Procedure: : DQDELETE_REAR(Q,F,R)

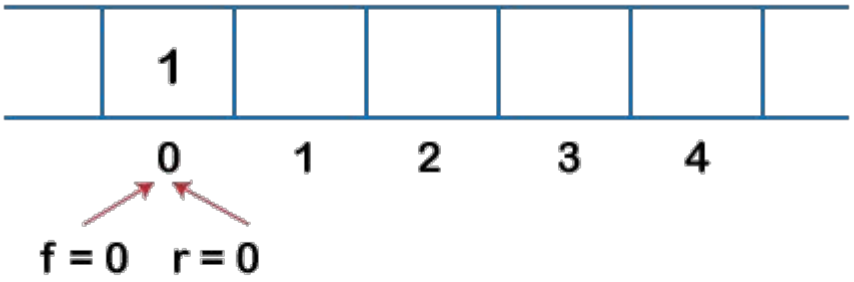
- This function deletes & returns an element from rear end of the Queue.
- Queue is represented by a vector Q containing N elements.
- F is pointer to the front element of a queue.
- R is pointer to the rear element of a queue.

```
1. [Underflow?]
   If    R = 0
   Then  Write('Underflow')
         Return(0)
2. [Delete Element]
    Y □ Q[R]
3. [Queue Empty?]
   IF   R = F
   Then R □ F □ 0
   Else R □ R - 1
4. [Return Element]
   Return(Y)
```



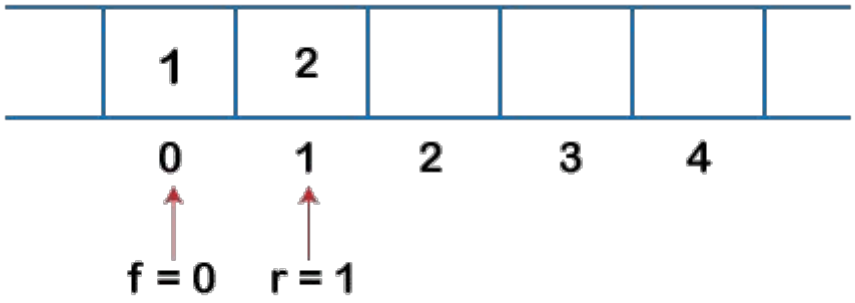
Implementation of Deque using a circular array Enqueue operation

1. Initially, we are considering that the deque is empty, so both front and rear are set to -1, i.e., **f = -1** and **r = -1**.
2. As the deque is empty, so inserting an element either from the front or rear end would be the same thing. Suppose we have inserted element 1, then **front is equal to 0,** and the **rear is also equal to 0.**



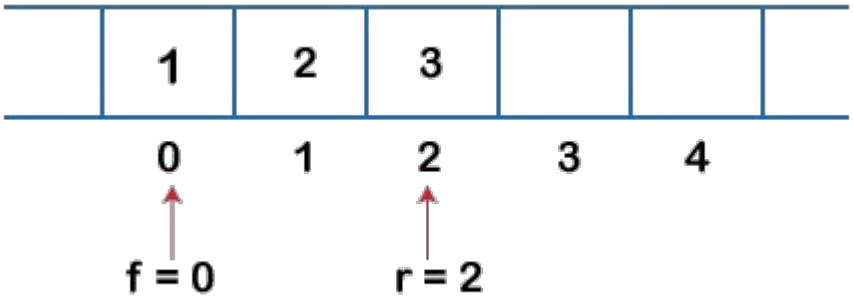
Implementation of Deque using a circular array Enqueue operation

3. Suppose we want to insert the next element from the rear. To insert the element from the rear end, we first need to increment the rear, i.e., **rear=rear+1**. Now, the rear is pointing to the second element, and the front is pointing to the first element.



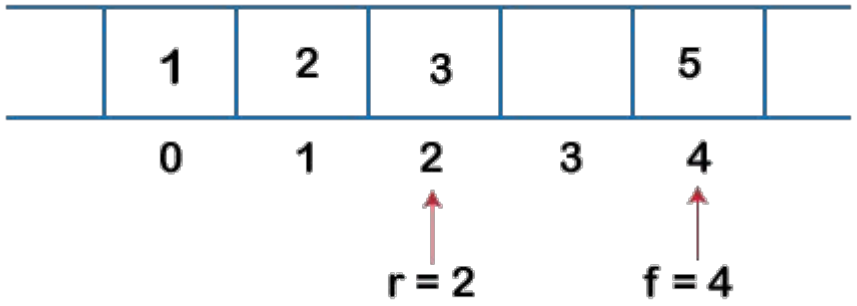
Implementation of Deque using a circular array Enqueue operation

4. Suppose we are again inserting the element from the rear end. To insert the element, we will first increment the rear, and now rear points to the third element.



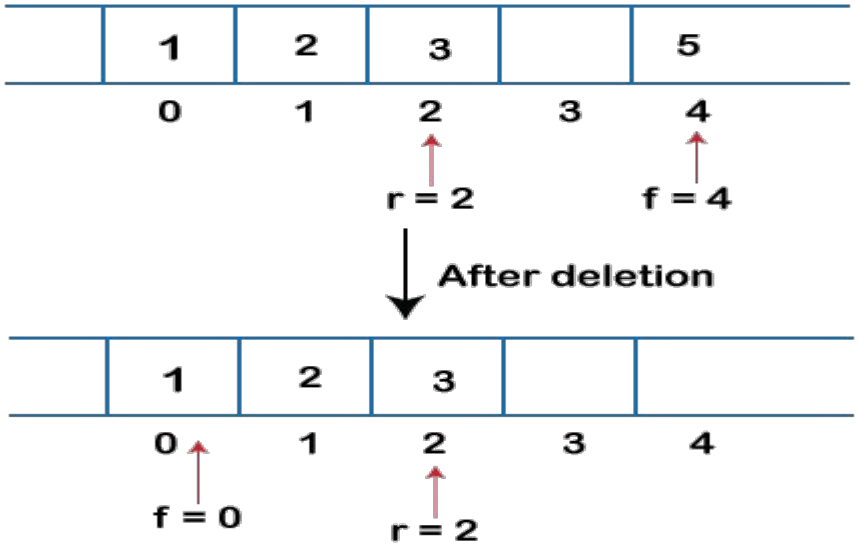
Implementation of Deque using a circular array Enqueue operation

5. If we want to insert the element from the front end, and insert an element from the front, we have to decrement the value of front by 1. If we decrement the front by 1, then the front points to -1 location, which is not any valid location in an array. So, we set the front as **(n -1),** which is equal to 4 as n is 5. Once the front is set, we will insert the value as shown in the below figure:



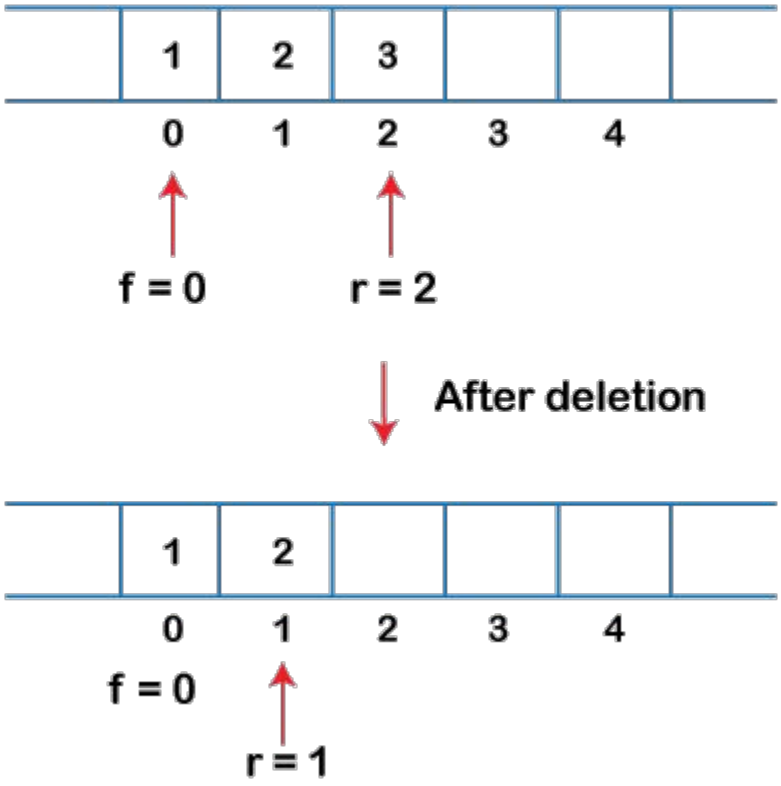
Implementation of Deque using a circular array Dequeue operation

1. If the front is pointing to the last element of the array, and we want to perform the delete operation from the front. To delete any element from the front, we need to set **front=front+1**. Currently, the value of the front is equal to 4, and if we increment the value of front, it becomes 5 which is not a valid index. Therefore, we conclude that if front points to the last element, then front is set to 0 in case of delete operation.



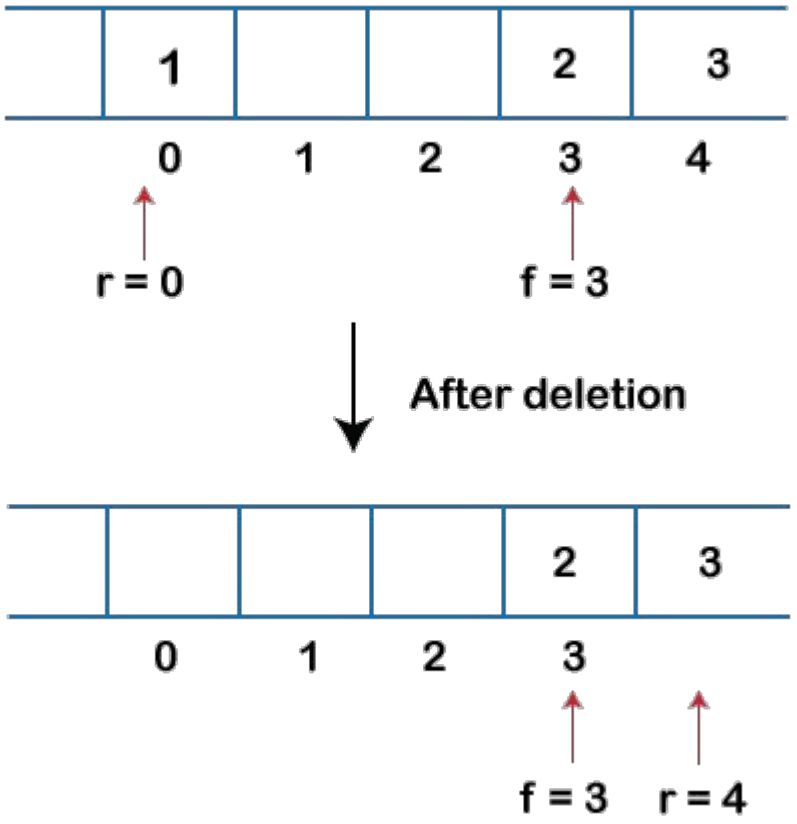
Implementation of Deque using a circular array Dequeue operation

2. If we want to delete the element from rear end then we need to decrement the rear value by 1, i.e., **rear=rear-1** as shown in the below figure:



Implementation of Deque using a circular array Dequeue operation

3. If the rear is pointing to the first element, and we want to delete the element from the rear end then we have to set **rear=n-1** where **n** is the size of the array as shown in the below figure:



Implementation of Deque using a circular array Program

- The following are the six functions that we have used in the below program:
- **enqueue_front()**: It is used to insert the element from the front end.
- **enqueue_rear()**: It is used to insert the element from the rear end.
- **dequeue_front()**: It is used to delete the element from the front end.
- **dequeue_rear()**: It is used to delete the element from the rear end.
- **getfront()**: It is used to return the front element of the deque.
- **getrear()**: It is used to return the rear element of the deque.

Implementation of DQueue - C Program

```
#define size 5
#include <stdio.h>

int deque[size];
int f=-1, r=-1;

//  enqueue_front function will insert the value from the front
void enqueue_front(int x)  {
   if((f==0 && r==size-1) || (f==r+1))     {
      printf("deque is full");     }
   else if((f==-1) && (r==-1))    {
      f=r=0;
      deque[f]=x;
   }
```

Implementation of DQueue - C Program

```
    else if(f==0)
    {
        f=size-1;
        deque[f]=x;
    }
    else
    {
        f=f-1;
        deque[f]=x;
    }
}
```

Implementation of DQueue - C Program

```
// enqueue_rear function will insert the value from the rear
void enqueue_rear(int x)  {
   if((f==0 && r==size-1) || (f==r+1))     {
      printf("deque is full");
   }
   else if((f==-1) && (r==-1))     {
      r=0;
      deque[r]=x;
   }
   else if(r==size-1)     {
      r=0;
      deque[r]=x;
   }
```

Implementation of DQueue - C Program

```
    else
    {
       r++;
       deque[r]=x;
    }

}
```

Implementation of DQueue - C Program

```
// display function prints all the value of deque.
void display()
{
   int i=f;
   printf("\n Elements in a deque : ");

   while(i!=r)
   {
      printf("%d ",deque[i]);
      i=(i+1)%size;
   }
    printf("%d",deque[r]);
}
```

Implementation of DQueue - C Program

```
// getfront function retrieves the first value of the deque.
void getfront()
{
   if((f==-1) && (r==-1))
   {
      printf("Deque is empty");
   }
   else
   {
      printf("\nThe value of the front is: %d", deque[f]);
   }

}
```

Implementation of DQueue - C Program

```
// getrear function retrieves the last value of the deque.
void getrear()
{
   if((f==-1) && (r==-1))
   {
      printf("Deque is empty");
   }
   else
   {
      printf("\nThe value of the rear is: %d", deque[r]);
   }

}
```

Implementation of DQueue - C Program

```
// dequeue_front() function deletes the element from the front
void dequeue_front()  {
   if((f==-1) && (r==-1))    {
      printf("Deque is empty");     }
   else if(f==r)     {
      printf("\nThe deleted element is %d", deque[f]);
      f=-1;
      r=-1;
   }
    else if(f==(size-1))      {
       printf("\nThe deleted element is %d", deque[f]);
       f=0;
    }
    else
    {
        printf("\nThe deleted element is %d", deque[f]);
        f=f+1;
    }
}
```

Implementation of DQueue - C Program

```
// dequeue_rear() function deletes the element from the rear
void dequeue_rear()  {
   if((f==-1) && (r==-1))     {
      printf("Deque is empty");     }
   else if(f==r)     {
      printf("\nThe deleted element is %d", deque[r]);
      f=-1;
      r=-1;
   }
    else if(r==0)
    {
       printf("\nThe deleted element is %d", deque[r]);
       r=size-1;
    }
    else
    {
        printf("\nThe deleted element is %d", deque[r]);
        r=r-1;
    }
}
```

Implementation of DQueue - C Program

```
// main function
int main()
{
   enqueue_front(2);   // inserting a value from the front.
   enqueue_front(1);   // inserting a value from the front.
   enqueue_rear(3);    // inserting a value from the rear.
   enqueue_rear(5);   // inserting a value from the rear.
   enqueue_rear(8);   // inserting a value from the rear.
   display();     // Calling the display function to retrieve the values of deque
   getfront();   // Retrieve the front value
   getrear();   // Retrieve the rear value.
  dequeue_front();   // deleting a value from the front
  dequeue_rear();   //deleting a value from the rear
 // Calling the display function to retrieve the values of deque
 display();
    return 0;
}
```