Queue

Amiya Ranjan Panda

Definitions

- A queue is a linear data structure such that insertions are made at one end, called the rear, and removals are made at the other end, called the front.
- Operations performed on queues on first-in first-out (FIFO) principle.

  insert() → Queue → delete()

The two basic operations are:

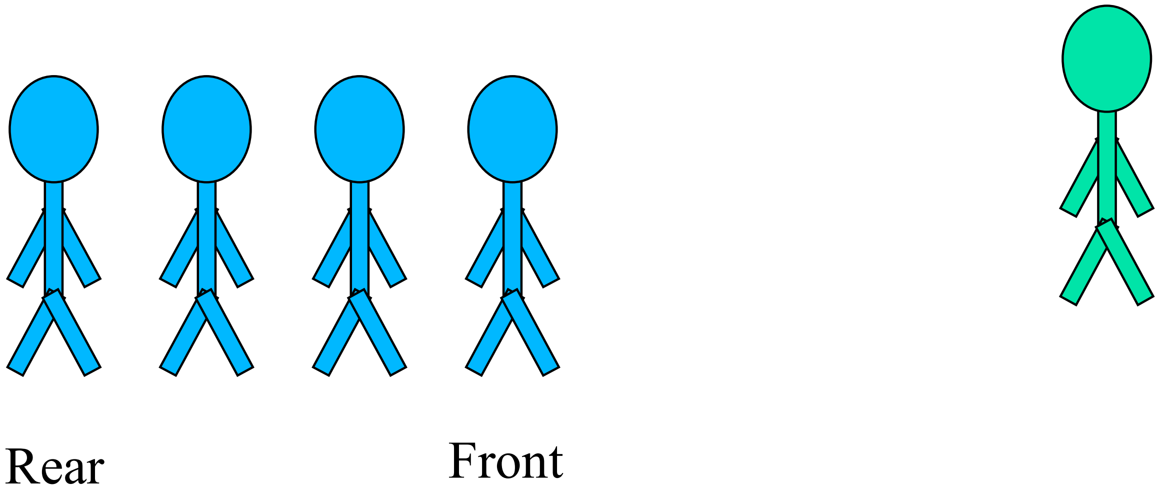
- insert(): adds an element to the rear of the queue.
- Delete(): removes and returns the element at the front of the queue.

Applications

- Shared resources management (system programming)
  - Access to the processor
  - Access to the peripherals such as disks and printers
- Application programs
  - Simulations

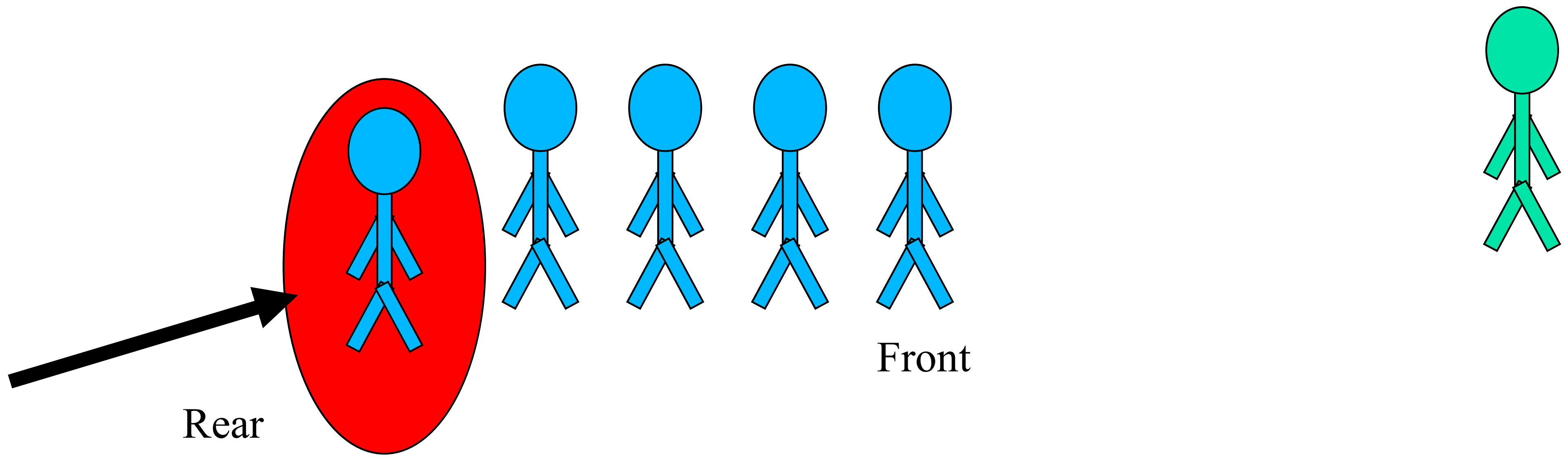
The Queue Operations

- A queue is like a line of people waiting for a bank teller. The queue has a front and a rear.



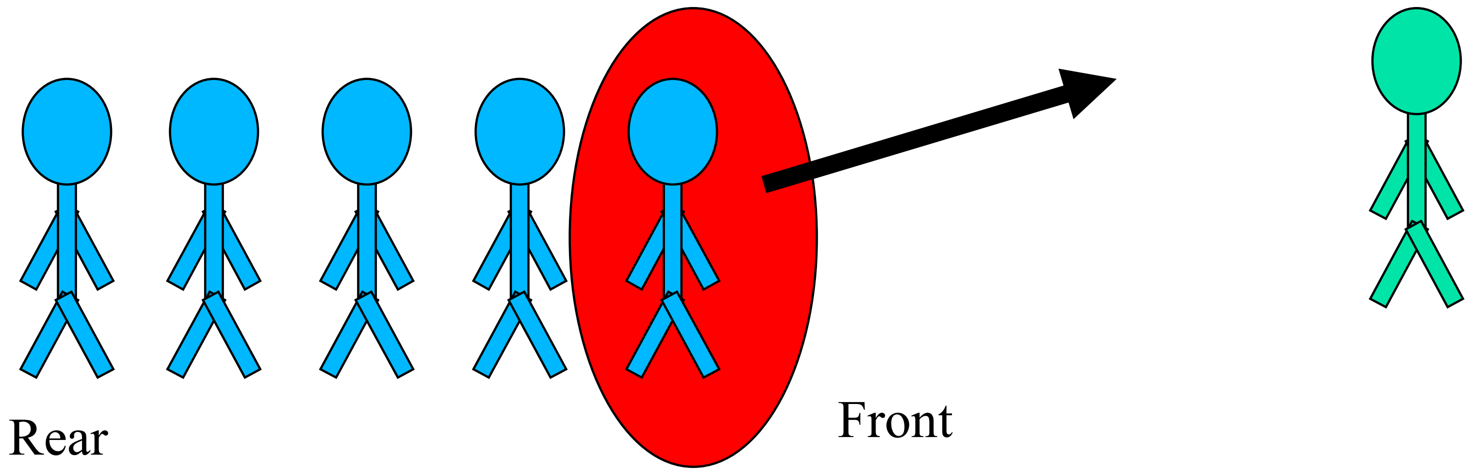
The Queue Operations

- New item must enter the queue at the rear end.
- This operation is called as insert operation.



The Queue Operations

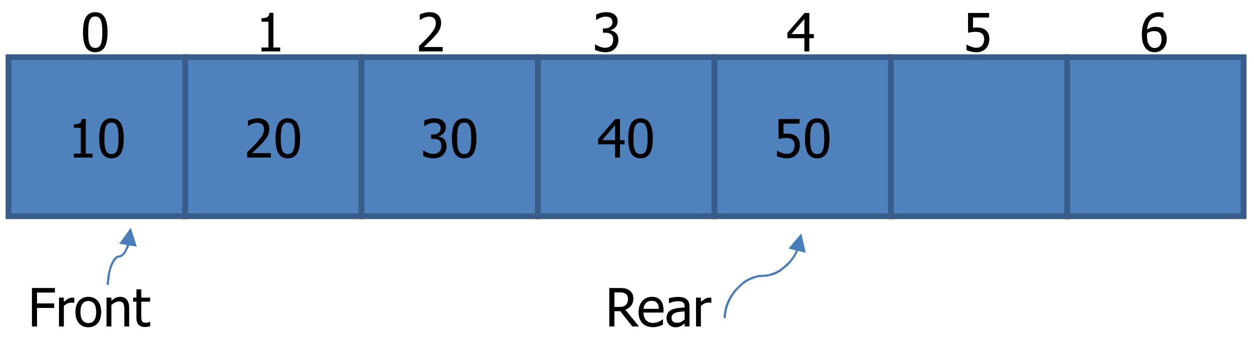
- When an item is taken from the queue, it always comes from the front end.
- It is usually called a delete operation.

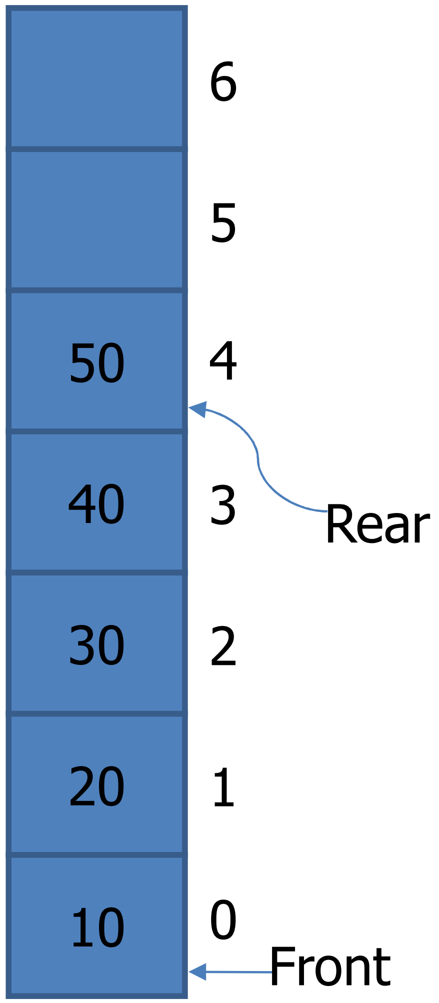


Array Implementation

Queue (Linear Queue)

- A linear data structure consisting of list of items.
- Data elements are added at one end, called the ***rear*** and removed from another end, called the ***front*** of the list.
- Two basic operations are associated with queue:
  - "Insert" operation is used to insert an element into a queue.
  - "Delete" operation is used to delete an element from a queue.
- FIFO list
- Example: Queue: 10, 20, 30, 40, 50





Algorithms for Insert and Delete Operations in Linear Queue

For Insert Operation

<u>Insert (Queue, Rear, Front, N, Item)</u>

- Queue is the place where to store data.
- Rear represents the location in which the data element is to be inserted and Front represents the location from which the data element is to be removed.
- Here N is the maximum size of the Queue and finally, Item is the new item to be added.

```
1. If Rear = = N-1 then Print: Overflow and Return.    /*Queue is Full*/
2. Rear = Rear +1
3. Queue[Rear] = Item
4. Return.
```

Algorithms for Insert and Delete Operations in Linear Queue

For Delete Operation

<u>Delete-Queue(Queue, Front, Rear, Item)</u>

- Queue is the place where data are stored.
- Rear represents the location in which the data element is to be inserted and Front represents the location from which the data element is to be removed.
- Front element is assigned to Item.

```
1.  If Rear < Front then Print: Underflow and Return.     /* Queue Empty */
2.  Item = Queue[Front]
3.  Front = Front + 1
4.  Return.
```

Array Implementation: Example

Example: Consider the following queue (linear queue).
Rear = 3 and Front = 0 and N = 7

10	**50**	**30**	**40**			
0	1	2	3	4	5	6

(1) Insert 20. Now Rear = 4 and Front = 0

10	**50**	**30**	**40**	**20**		
0	1	2	3	4	5	6

(2) Delete Front Element. Now Rear = 4 and Front = 1

	50	**30**	**40**	**20**		
0	1	2	3	4	5	6

(3) Delete Front Element. Now Rear = 4 and Front =2

		30	**40**	**20**		
0	1	2	3	4	5	6

(4) Insert 60. Now Rear = 5 and Front = 2

		30	**40**	**20**	**60**	
0	1	2	3	4	5	6

Queue Operations

```
#define SIZE 50

int rear = -1,  front = 0;
int q[SIZE];

void insert(int x)  {
          if (rear == MAX - 1)
                    printf("Queue Overflow \n");
          else  {
                    rear = rear + 1;
                    q[rear] = x;
          }
}
```

Queue Operations

```
delete()  {
        int x
        if (rear < front)
                printf("Queue Underflow \n");
        else
        {
                x = q[front];
                front = front + 1;
        }
}

void display()  {
        int i;
        if (rear < front)
                printf("Queue is empty \n");
        else  {
                printf("Queue is : \n");
                for (i = front; i <= rear; i++)
                        printf("%d ", q[i]);
                }
}
```

Implemetation of Queue

```
#include <stdio.h>
#define MAX 50
void insert();
void Delete();
void display();
int que[MAX];
int rear = - 1;
int front = - 1;
int main() {
    int choice;
    do {
        printf("1.Insert \n");
        printf("2.Delete \n");
        printf("3.Display \n");
        printf("4.Quit \n");
        printf("Enter your choice : ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                insert();
                break;
            case 2:
                Delete();
                break;
            case 3:
                display();
                break;
            case 4:
                break;
            default:
                printf("Wrong choice \n");
        }
    }while(choice != 4);
}
```

Implemetation of Queue

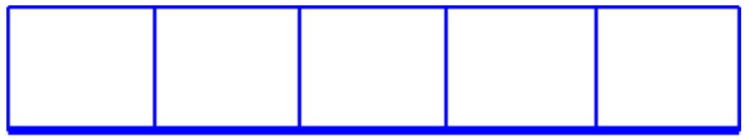
```
void insert() {
    int item;
    if (rear == MAX-1)
      printf("Queue Overflow \n");
    else {
        if (front == - 1)
             front = 0;
        printf("Insert the element in
                    queue: ");
        scanf("%d", &item);
        rear = rear + 1;
        que[rear] = item;
    }
}
```

```
void delete() {
    if (front == - 1 || front > rear) {
        printf("Queue Underflow \n");
        return ;
    }
    else {
        printf("Element: %d\n", que[front]);
        front = front + 1;
    }
}
void display() {
    int i;
    if(front == -1)
        printf("Queue is empty \n");
    else {
        printf("Queue is : \n");
        for (i = front; i <= rear; i++)
            printf("%d ", que[i]);
        printf("\n");
    }
}
```

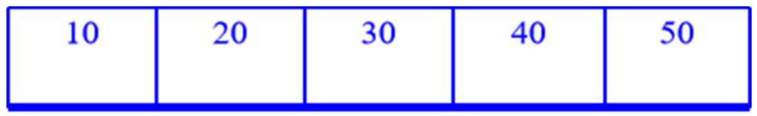
Drawbacks of Linear Queue

- when an element added into Queue, rear pointer is increased by 1
- but when an element is removed front pointer is increased by 1
- Array implementation of queue may cause problems
- Consider operations performed on a Queue (with SIZE = 5) as follows:

1. Initially Queue is empty: front = 0 and rear = -1

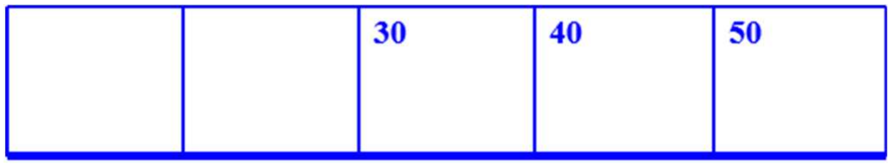


2. When 5 elements are added to queue, the state of the queue becomes as follows with front = 0 and rear = 4.



Drawbacks of Linear Queue

3. Now suppose two elements are deleted from Queue then, the state of the Queue becomes as follows, with front = 2 and rear = 4



4. Now, actually two elements are deleted from queue so, there should be space for another 2 elements in the queue, but as rear pointer is pointing at last position and Queue overflow condition (Rear == SIZE-1) is true, new element cannot be inserted in the queue even if it has empty spaces.

To overcome this problem there is another variation of queue called circular queue.

Circular Queue

- Queues, implemented wrapping around, are called Circular Queues.
- Both the front and the rear pointers wrap around to the beginning of the array.
- It is also called as “Ring buffer”.

Circular Queue

- The structure of circular queue is shown in following figure:

- In circular queue, once the Queue is full the "First" element of the Queue becomes the "Rear" most element, if and only if the "Front" has moved forward. otherwise it will again be a "Queue overflow" state.

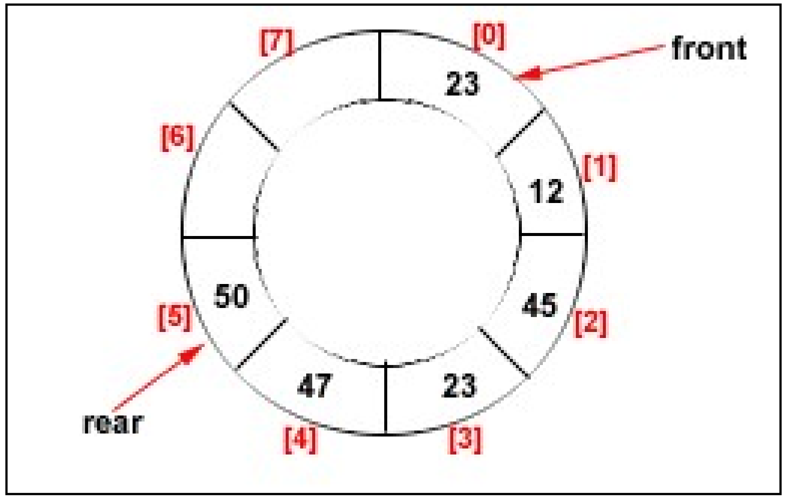


Figure: Circular Queue having Rear = 5 and Front = 0

Algorithm for Insert Operation in Circular Queue

Insert-Circular-Q(CQueue, Rear, Front, N, Item)

- CQueue is a circular queue where to store data.
- Rear represents the location in which the data element is to be inserted and Front represents the location from which the data element is to be removed.
- N is the size of CQueue and finally, Item is the new item to be added. Initailly Rear = -1 and Front = -1.

```
1. If Front == -1 and Rear == -1 then Front = 0 and go to step 4.
2. If  Front == 0 and Rear == N-1 or Front = Rear + 1
           then Print: "Circular Queue Overflow" and Return.
3. If Rear == N-1  then Rear = 0 and go to step 5.
4. Rear = Rear + 1
5. CQueue [Rear] = Item.
6. Return
```

Algorithm for Delete Operation in Circular Queue

Delete-Circular-Q(CQueue, Front, Rear, Item)

- CQueue is the place where data are stored.
- Rear represents the location in which the data element is to be inserted and Front represents the location from which the data element is to be removed.

```
1. If Front == -1 then
        Print: "Circular Queue Underflow" and Return.   /*Delete without
Insertion
2. Item := CQueue [Front]
3. If Front == N-1 then Front = 0 and Return.
4. If Front == Rear then Front = -1 and Rear = -1 and Return.
5. Front = Front + 1
6. Return.
```

Implementing Circular Queue

```
# include<stdio.h>
# define MAX 5
int cque[MAX];
int front = -1;
int rear = -1;
void insert() {
int item;
if((front == 0 && rear == MAX-1) ||
                    (front == rear+1)) {
      printf("Queue Overflow ... \n\n");
      return;
}
if (front == -1) {
      front = 0;
      rear = 0;
}
else {
      if(rear == MAX-1)
            rear = 0;
      else
            rear = rear+1;
}
printf("Input the element : ");
scanf("%d", &item);
cque[rear] = item ;
}
```

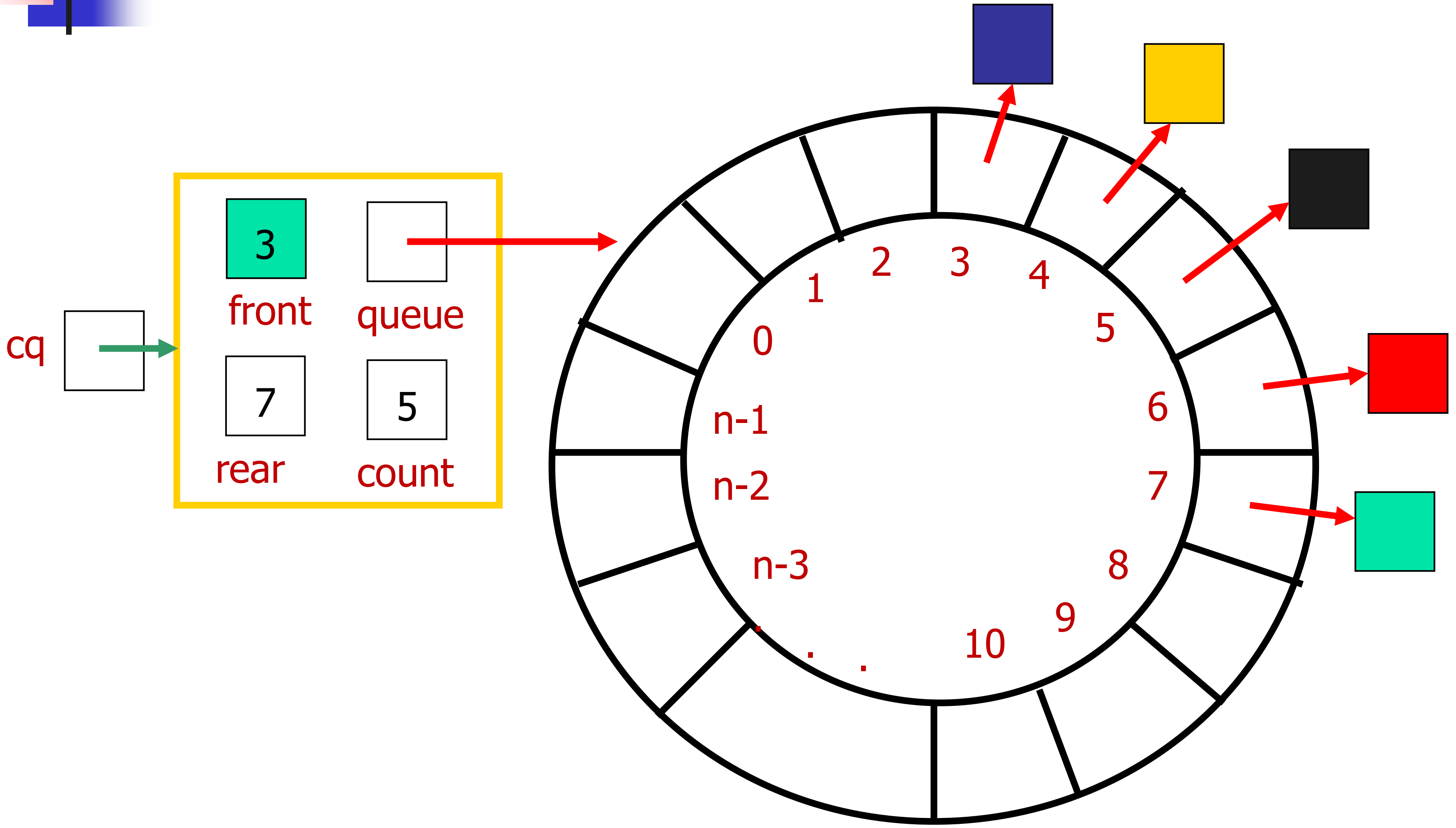
Implementing Circular Queue

```
void del() {
if (front == -1) {
	printf("Queue Underflow\n\n");
	return ;
}
printf("Element : %d\n",cque[front]);
if(front == rear) {
	front = -1;
	rear=-1;
}
else {
	if(front == MAX-1)
		front = 0;
	else
		front = front+1;
}
}
```

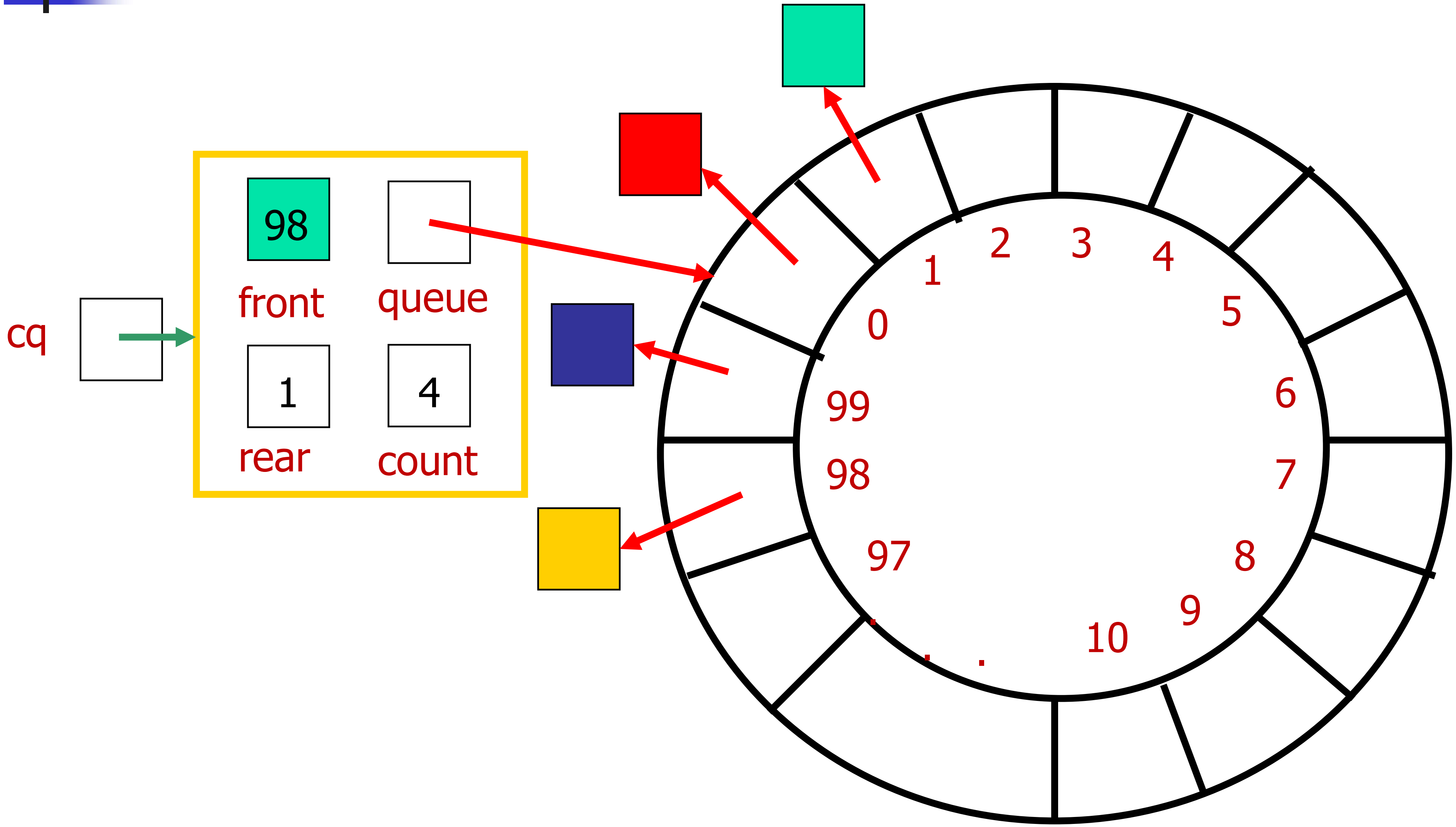
Implementing Circular Queue

```
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 ",cque[front_pos]);
            front_pos++;
      }
else {
      while(front_pos <= MAX-1) {
            printf("%d ",cque[front_pos]);
            front_pos++;
      }
      front_pos = 0;
      while(front_pos <= rear_pos) {
            printf("%d ",cque[front_pos]);
            front_pos++;
      }
}
printf("\n");
}
```

Implementation of a Circular Queue



Insertion at End of a Circular Queue



Algorithm for Insert Operation in Circular Queue

Insert-Circular-Q(CQueue, Rear, Front, N, Item)

- CQueue is a circular queue where to store data.
- Rear represents the location in which the data element is to be inserted and Front represents the location from which the data element is to be removed.
- N is the size of CQueue and finally, Item is the new item to be added. Initailly Rear = -1 and Front = -1.

```
1.  If (Front == (Rear+1)%MAX)
2.          Print "circular queue overflow"
3.          Return
4.  Rear = (Rear+1)%MAX
5.  CQueue[Rear] = Item;
6.  If (Front == -1 )
7.          Front = 0;
```

Algorithm for Delete Operation in Circular Queue

Delete-Circular-Q(CQueue, Front, Rear, Item)

- CQueue is the place where data are stored.
- Rear represents the location in which the data element is to be inserted and Front represents the location from which the data element is to be removed.

```
1.  if ((Front == Rear) && (Rear == -1))
2.          Print "Circular Queue underflow"
3.          Return
4.  Item = CQueue[Front]
5.  If (Front == Rear)
6.        Front=Rear = -1
7.  Else
8.        Front = (Front + 1) % MAX
```

Implementing cqueue using arrays

- Simple implementation
- The size of the queue must be determined when it is declared
- Space is wasted if less elements are used
- “insert" cannot be performed for more elements than the array can hold

Implementing cqueue using arrays

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

typedef struct queue {
        int cque[SIZE];
        int rear;
        int front;
}CQueue;

void insert(CQueue *cq) {
        int item;
        if(cq->front == (cq->rear+1) % MAX) {
                printf("CQueue Overflow...\n");
                return;
        }
        printf("Enter the item : ");
        scanf("%d", &item);
        if(cq->front == -1) {
                cq->front = 0;
                cq->rear = 0;
        }
        else
                cq->rear = (cq->rear+1) % SIZE;
        cq->cque[cq->rear] = item;
}
```

Implementing cqueue using arrays

```
void del(CQueue *cq) {
        if(cq->front == -1)  {
                printf("CQueue Underflow...\n");
                return;
        }
        printf("Item : %d\n", cq->cque[cq->front]);
        if(cq->front == cq->rear) {
                cq->front = -1;
                cq->rear = -1;
        }
        else
                cq->front = (cq->front+1) % SIZE;
}
```

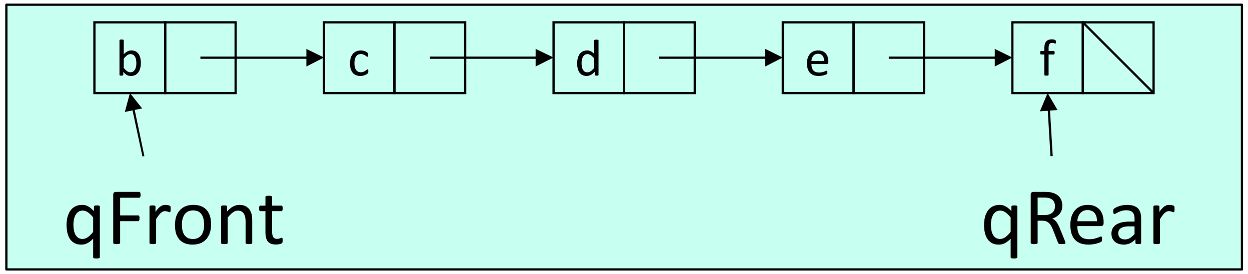
```
void display(CQueue cq) {
        int i;
        if(cq.front == -1)  {
                printf("CQueue is Empty...\n");
                return;
        }
        printf("Front -> %d\n", cq.front);
        printf("Elements -> ");
        for(i = cq.front; i != cq.rear; i= (i+1) %
                                                SIZE)
                printf("%d  ", cq.cque[i]);
        printf("%d\n", cq.cque[i]);
        printf("Rear -> %d\n", cq.rear);
}
```

Implementing cqueue using arrays

```
int main() {
        CQueue cq;
        int choice;
        cq.rear = cq.front = -1;
        do {
                printf("1.Insert\n");
                printf("2.Delete\n");
                printf("3.Display\n");
                printf("4.Quit\n");
                printf("Enter your choice : ");
                scanf("%d", &choice);
                switch(choice) {
                        case 1 :
                                insert(&cq);
                                break;
                        case 2 :
                                del(&cq);
                                break;
                        case 3:
                                display(cq);
                                break;
                        case 4:
                                break;
                        default:
                                printf("Wrong choice\n");
                }
        }while(choice!=4);
        return 0;
}
```

Implementing queues using linked lists

- Allocate memory for each new element dynamically
- Link the queue elements together
- Use two pointers, *qFront* and *qRear*, to mark the front and rear of the queue



Implementing queues using linked lists: Declaration and Initialization

Structure of the Node

```
struct Node {
        int data;
        struct Node* next;
}*rear, *front;

void create() {
        front = rear = NULL;
}
```

Implementing queues using linked lists: Insert Function

```
void Insert(int val) {
    struct Node *newNode;
    newNode=(struct Node *)malloc(sizeof(struct Node));
    newNode->data=val;
    newNode->next = NULL;
    if(front == NULL && rear == NULL) {
        front = rear = newNode;
        return;
    }
    rear->next = newNode;
    rear = newNode;
}
```

Implementing queues using linked lists: Delete Function

```
void Delete() {
    struct Node* temp = front;
    if(front == NULL) {
        printf("Queue is Empty");
        return;
    }
    if(front == rear) {
        front = rear = NULL;
    }
    else {
        front = front->next;
    }
    free(temp);
}
```

Implementing Queue using Stacks

Method 1: (By making insert() operation costly)

- This method makes sure that first element inserted is always at the top of stack 1
- So that delete() operation just pops from stack1
- To put the element at top of stack1, stack2 is used.

Implementing Queue using Stacks

insert(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.
- Here time complexity will be O(n)

delete(q):

- If stack1 is empty then error
- Pop an item from stack1 and return it
- Here time complexity will be O(1)

Implementing Queue using Stacks

```
void insert(int x) {
     // Move all elements from s1 to s2
     while (!empty(s1)) {
          push(s2, top(s1));
          pop(s1);
       }
       // Push item into s1
       push(s1, x);
       // Push everything back to s1
       while (!empty(s2)) {
          push(s1, top(s2));
          pop(s2);
       }
}
```

```
int delete() {
     // if first stack is empty
     if (empty(s1)) {
          printf("Queue is Empty");
          return;
       }

       // Return top of s1
       int x = top(s1);
       pop(s1);
       return x;
}
```

Implementing Stack using Queues

```
void push(int x) {
    int item;
    // move all elements in q1 to q2
    while(!isEmpty(q1)) {
        item = Delete(q1);
        insert(q2, temp);
    }
    // push the element into Stack
    insert(q1, x);
    // move back all elements back to Q1
                                        from Q2
    while(!isEmpty(q2)) {
        item = Delete(q2);
        insert(q1, item);
    }
}
```

```
int pop() {
    // if first stack is empty
    if (empty(q1)) {
        printf("Empty Stack");
        return;
     }
    return Delete(q1);
}
```

Implementing Stack using Deque

```
void push(int x) {
    int item;
    // push the element into Stack
    insertAtFront(q, x);
}
```

```
int pop() {
    // if first stack is empty
    if (empty(q1)) {
        printf("Empty Stack");
        return;
    }
    return deleteAtFront(q);
}
```

Deques

- A deque is a double-ended queue
- Insertions *and* deletions can occur at *either* end
- Implementation is similar to that for queues
- Deques are not heavily used

Deques

There are four basic operations in Deque:

- Insertion at rear end
- Insertion at front end
- Deletion at front end
- Deletion at rear end

Algorithm for Insertion at rear end

```
Step-1: [Check for overflow]
            if(rear==MAX-1)
                  Print("Queue is Overflow”);
                  return;
Step-2:  [Insert Element]
            else
                  rear=rear+1;
                  q[rear]=item;
            [Set rear and front pointer]
            if front=-1
                  front=0;
Step-3: return
```

Algorithm for Insertion at front end

```
Step-1 :  [Check for the front position]
          if(front<=0)
               Print("Cannot add item at the front”);
               return;
Step-2 :  [Insert at front]
          else
               front=front-1;
               q[front]=item;
Step-3  : Return
```

Algorithm for Deletion from front end

```
Step-1 [ Check for front pointer]
        if front==-1
            print(" Queue is Underflow");
            return;
Step-2 [Perform deletion]
        else
            item=q[front];
            print("Deleted element is", item);
        [Set front and rear pointer]
        if front = = rear
            front=-1;
            rear=-1;
        else
            front=front+1;
Step-3 : Return
```

Algorithm for Deletion from rear end

```
Step-1 : [Check for the rear pointer]
            if rear==-1
                  print("Cannot delete value at rear end");
                  return;
Step-2:  [ perform deletion]
            else
                  item=q[rear];
                  [Check for the front and rear pointer]
            if front==rear
                  front=-1;
                  rear=-1;
            else
                  rear=rear-1;
                  print("Deleted element is", item);
Step-3 : Return
```

Deques

Types of Deque:

1. Input restricted deque
2. Output restricted deque

- An input restricted deque is a deque, which allows
  - insertion at only one end, i.e. rear end,
  - but deletion at both ends, rear and front end of the lists.

- An output-restricted deque is a deque, which allows
  - deletion at only one end, i.e. front end,
  - but insertion at both ends, rear and front ends, of the lists

Implementing Deque using Array

```
#include <stdio.h>
#define MAX 5
int deque[MAX];
int rear =-1 ;
int front = -1 ;
void insertAtFront();
void insertAtRear();
void deleteAtFront();
void deleteAtRear();
void display();

int main() {
    int option;
    do {
        printf("\n\n DEQUE");
        printf("\n 1. Insert at front");
printf("\n 2. Insert at rear");
printf("\n 3. Delete from front");
printf("\n 4. Delete from rear");
printf("\n 5. Display");
printf("\n 6. Quit");
printf("\n Enter your option : ");
scanf("%d", &option);
switch (option) {
    case 1:
        insertAtFront();
        break;
    case 2:
        insertAtRear();
        break;
    case 3:
        deleteAtFront();
        break;
```

Implementing Deque using Array

```
        case 4:
            deleteAtRear();
            break;
        case 5:
            display();
            break;
        case 6:
            break;
        default:
            printf("Wrong Choice ...\n");

      }
    } while (option!=6);
    return 0;
}
```

```
void insertAtFront() {
    int item;
    if(front == (rear+1) % MAX) {
        printf("Deque Overflow...\n");
        return;     }
    printf("\n Enter the value: ");
    scanf("%d", &item);
    if(front == -1) {
        front = 0;
        rear = 0;
    }
    else if(front == 0)
        front=MAX-1;
    else
        front = front-1;
    deque[front] = item;
}
```

Implementing Deque using Array

```
void insertAtRear() {
    int item;
    if(front == (rear+1) % MAX) {
        printf("Deque Overflow...\n");
        return;
    }
    printf("\n Enter the value: ");
    scanf("%d", &item);
    if(front == -1) {
        front = 0;
        rear = 0;
    }
    else
        rear = (rear+1) % MAX;
    deque[rear] = item;
}
```

```
void deleteAtFront() {
    if (front == -1) {
        printf("\nDeque Underflow...\n");
        return ;
    }
    printf("\nElement is : %d", deque[front]);
    if(front == rear) {
        front = -1 ;
        rear = -1 ;
    }
    else
        front = (front+1) % MAX;
}
```

Implementing Deque using Array

```
void deleteAtRear() {
    if (front == -1) {
        printf("\nDeque Underflow...\n");
        return ;
    }
    printf("\nElement is : %d", deque[rear]);
    if(front == rear) {
        front = -1 ;
        rear = -1 ;
    }
    else if(rear == 0)
        rear = MAX-1;
    else
        rear = rear-1;
}
```

```
void display() {
    int f = front, r = rear;
    if (front == -1 ) {
        printf("\n Empty Deque...\n");
        return;     }
    printf("\nThe elements of the Deque are : ");
    if(f <= r)
        while(f <= r) {
            printf("%d  ", deque[f]);
            f++;
        }
    else  {
        while(f <= MAX-1) {
            printf("%d  ", deque[f]);
            f++;
        }
        f = 0;
        while (f <= r)  {
            printf("%d  ", deque[f]);
            f++;
        }  }
    printf("\n");   }
```

Implementing deque using linked lists: Declaration and Initialization

Structure of the Node

```
struct Node {
      int data;
      struct Node* next;
}*rear, *front;

void create() {
      front = rear = NULL;
}
```

Implementing deque using linked lists: Insert Functions

```
void InsertAtRear(int val) {
    struct Node *newNode;
    newNode=(struct Node *)
       malloc(sizeof(struct Node));
    newNode->data=val;
    newNode->next = NULL;
    if(front == NULL)
         front = newNode;
    else
         rear->next = newNode;
    rear = newNode;
}
```

```
void InsertAtFront(int val) {
    struct Node *newNode;
    newNode=(struct Node *)
       malloc(sizeof(struct Node));
    newNode->data=val;
    newNode->next = NULL;
    if(front == NULL)
         front = rear = newNode;
    else {
         newNode->next = front;
         front = newNode;
    }
}
```

Implementing deque using linked lists: Delete Functions

```
int delqAtFront(){
    struct Node *temp = front;
    int item ;
    if ( temp == NULL ){
  printf ( “Empty Queue ");
  return 0 ; }
    else {
  item = temp -> data;
  front = temp -> next;
  free ( temp );
  if(front == NULL )
      rear = NULL;
  return ( item );  }
}
```

```
int delqAtRear(){
   struct Node *temp , *rleft=NULL, *q ;
   int item;
   temp = front ;
   if(rear == NULL ){
      printf ( “Empty Queue" ) ;
      return 0 ; }
   else {
      while(temp != rear ){
         rleft = temp;
         temp = temp->next; }
      q = rear; item = q->data; free ( q );
      rear = rleft;
      if(rear != NULL)  rear -> next = NULL;
      if ( rear == NULL ) front = NULL;
    return ( item ) ; }
}
```

Priority Queues

- More specialized data structure than Queue
- Priority queue has same method but with a major difference.
  - In Priority queue items are ordered by key value so that item with the lowest value of key is at front and item with the highest value of key is at rear or vice versa.
  - priority is assigned to items based on their key value
  - Lower the value, higher the priority
- Following are the principal methods of a Priority Queue:
  - insert / enqueue: add an item to the rear of the queue.
  - remove / dequeue: remove an item from the front of the queue.

Priority Queues

```
void insert() {
    int i = 0;
    int item;
    if(!isFull()){
        printf("Enter the item: ");
        scanf("%d", &item);
        if(count == 0)
            pQue[count++] = item;
        else {
            for(i=count-1; i >= 0; i--) {
                // if data is larger, shift existing item to right end
                if(item > pQue[i])
                    pQue[i+1] = pQue[i];
                else
                    break;
            }
            pQue[i+1] = item;
            count++;
    }
    else
        printf("Priority Queue Overflow...\n");
}

int del(){
    if(count != 0)
        return pQue[--count];
    else
        printf("Priority Queue Underflow...\n");
}
```

Application of Queues

Queue is used when things don't have to be processed immediatly, but have to be processed in **F**irst **I**n **F**irst **O**ut order. This property of Queue makes it useful in following kind of scenarios.

- When a resource is shared among multiple consumers.
  - Examples include CPU scheduling, Disk Scheduling.
- When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes.
  - Examples include I/O Buffers, pipes, file IO, etc.