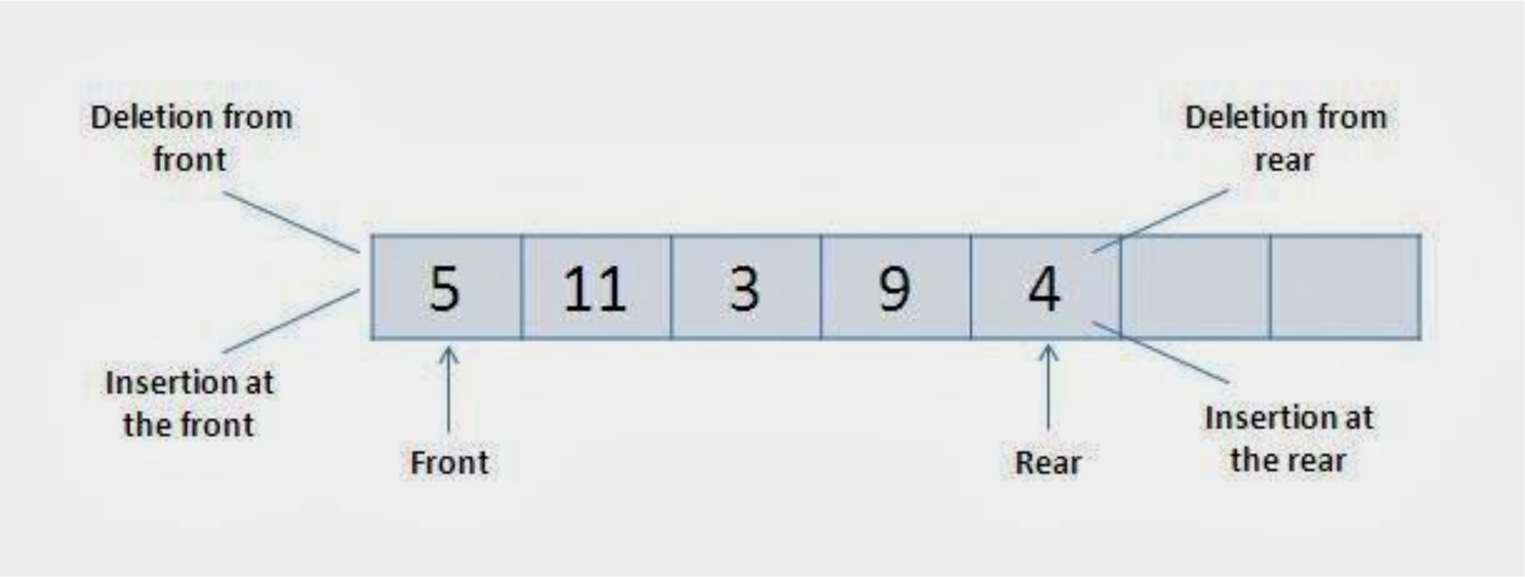
# Dequeues

SENG 12213

Data Structures and Algorithms

Dr. Nalin Warnajith
Software Engineering Teaching Unit
University of Kelaniya

# DEQUEUE


Deletion from front
Insertion at the front
5
11
3
9
4
Front
Rear
Deletion from rear
Insertion at the rear

# DEQUEUE

- A deque is a homogeneous list in which elements can be added or inserted (called push operation) and deleted or removed from both the ends (which is called pop operation).

- we can add a new element at the rear or front end and also we can remove an element from both front and rear end. Hence it is called Double Ended Queue.

# DEQUEUE

- There are two types of deque depending upon the restriction to perform insertion or deletion operations at the two ends. They are

    1. Input restricted deque
    2. Output restricted deque

# DEQUEUE – Contiguous Implementation

```
#define MAXDEQUEUE 20
#define TRUE 1
#define FALSE 0

typedef int DeQueueElement;

typedef struct dequeue{
     int front;
     int rear;
     DeQueueElement items[MAXDEQUEUE];
    } DeQueue;
```

# DEQUEUE – Contiguous Implementation

```
void CreateDeQueue(DeQueue *dq){
    dq->count=0;
    dq->front=-1;
    dq->rear=-1;
}
```

# DEQUEUE – Contiguous Implementation

```
int IsDeQueueFull(DeQueue *dq){
if ((dq->front== 0 && dq->rear== MAXDEQUEUE -1)
                      ||dq->front== dq->rear+1))
        return 1;
}
```

# DEQUEUE – Contiguous Implementation

```
int IsDeQueueEmpty(DeQueue *dq){
     if(dq->front==-1)
           return 1;
}
```

# DEQUEUE – Contiguous Implementation

- Insert an element at the Rear of the DeQueue

```
1. Input the DATA to be inserted
2. If Dequeue is FULL
       (a) Display "DeQueue is FULL"
       (b) Exit
3. If Dequeue is empty
       (a) front = 0
       (b) rear = 0
4. Else
       (a) if (rear == MAX -1)
               (i) rear = 0
       (b) else
               (i) rear = rear +1
5. Q[rear] = DATA
6. Exit
```

# DEQUEUE – Contiguous Implementation

- Insert an element at the Rear of the DeQueue

```
void InsertRear(DeQueue *dq, DeQueueElement x){
if(IsDeQueueFull(dq)){
    printf(“DeQueue is full... ");
    exit(1);
}
If(IsDeQueueEmpty(dq)){
     dq->front=dq->rear=0;
}else{
     if(dq->rear=MAXDEQUEUE-1)
           dq->rear=0;
     else
           dq->rear++;
     dq->items[dq->rear]=x;
}
}
```

# DEQUEUE – Contiguous Implementation

- Insert an element at the Front of the DeQueue

```
1. Input the DATA to be inserted
2. If Dequeue is FULL
       (a) Display “Queue Overflow”
       (b) Exit
3. If Dequeue is empty
       (a) front = 0
       (b) rear = 0
4. Else
       (a) if (front == 0)
               (i) front = MAX-1
       (b) else
               (i) front = front-1
5. Q[front] = DATA
6. Exit
```

# DEQUEUE – Contiguous Implementation

- Insert an element at the Front of the DeQueue

```
void InsertFront(DeQueue *dq, DeQueueElement x){
if(IsDeQueueFull(dq)){
    printf(“DeQueue is full... ");
    exit(1);
}
If(IsDeQueueEmpty(dq)){
     dq->front=dq->rear=0;
}else{
     if(dq->front=0)
          dq->front=MAXDEQUEUE-1;
     else
          dq->front--;
     dq->items[dq->front]=x;
}
}
```

# DEQUEUE – Contiguous Implementation

- Delete an element from the Rear of the DeQueue

```
1. If Dequeue is empty
       (a) Display “DeQueue Underflow”
       (b) Exit
2. DATA = Q [rear]
3. If (front == rear )
       (a) front = - 1
       (b) rear = - 1
4. Else
       (a) if(rear == 0)
              (i) rear = MAX-1
       (b) else
              (i) rear = right-1
5. Exit
```

# DEQUEUE – Contiguous Implementation

- Delete an element from the Rear of the DeQueue

```
void DeleteRear(DeQueue *dq, DeQueueElement x){
if(IsDeQueueFull(dq)){
    printf(“DeQueue is full... ");
    exit(1);
}
*x=dq->items[dq->rear];

If(dq->rear=dq->front){
     dq->front=dq->rear=-1;
}else{
     if(dq->rear=0)
           dq->rear=MAXDEQUEUE-1;
     else
           dq->rear--;
     }
}
```

# DEQUEUE – Contiguous Implementation

- Delete an element from the Front of the DeQueue

```
1. If (front == - 1)
      (a) Display “Queue Underflow”
      (b) Exit
2. DATA = Q [front]
3. If (front == rear )
      (a) front = - 1
      (b) rear = - 1
4. Else
      (a) if(front == MAX-1)
             (i) front = 0
      (b) else
             (i) front = front +1
5. Exit
```

# DEQUEUE – Contiguous Implementation

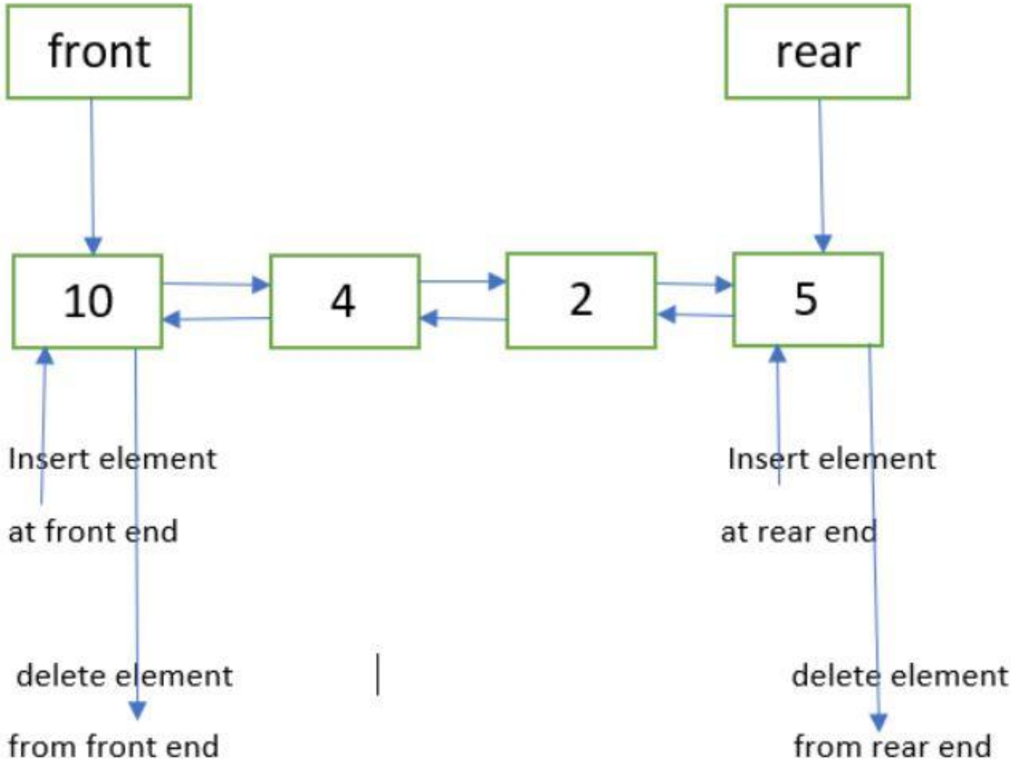
- Delete an element from the Rear of the DeQueue

```
void DeleteFront(DeQueue *dq, DeQueueElement x){
if(IsDeQueueFull(dq)){
    printf(“DeQueue is full... ");
    exit(1);
}
*x=dq->items[dq->front];

If(dq->rear=dq->front){
      dq->front=dq->rear=-1;
}else{
      if(dq->front=MAXDEQUEUE-1)
           dq->front=0;
      else
           dq->front--;
      }
}
```

- Display DeQueue items

```
void Display(DeQueue *dq){

Int front_pos=dq->front,rear_pos=dq->rear;

if(IsDeQueueEmpty(dq)){
    printf(“DeQueue is Empty... ");
    exit(1);
}
printf ("Queue elements :");

if(front_pos<=rear_pos){
       while(front_pos<=rear_pos){
             printf (“%d ”,dq->items[front_pos]);
             front_pos++;
}else{
       while(front_pos<=MAXDEQUEUE){
              printf (“%d ”,dq->items[front_pos]);
             front_pos++; }
       while(front_pos<=rear_pos){
             printf (“%d ”,dq->items[front_pos]);
             front_pos++;

}
```

# DEQUEUE – Linked Implementation



# DEQUEUE – Linked Implementation

```
typedef int DeQueueElement;
typedef enum {TRUE,FALSE} Boolean;

typedef struct node{
    DeQueueElement entry;
    struct node *next, *prev;
}Node;

typedef struct{
    int count;
    Boolean full;
    Node *front;
    Node *rear;
}DeQueue;
```

# DEQUEUE – Linked Implementation

```
void CreateDeQueue(DeQueue *dq){
    dq->count=0;
    dq->full=FALSE;
    dq->front=dq->rear=NULL;

}
```

# DEQUEUE – Linked Implementation

```
Boolean IsDeQueueEmpty(DeQueue *dq)
{
   return (dq->front==NULL && dq->rear==NULL);
}

Boolean IsDeQueueFull(DeQueue *dq)
{
    return (dq->full);
}
```

# DEQUEUE – Linked Implementation

- Insert an element at the Front of the DeQueue

```
Allocate space for a newNode of doubly linked list.
IF newNode == NULL, then
    print "Overflow”
ELSE
    IF front== NULL, then
        rear = front = newNode
    ELSE
        newNode->next= front
        front->prev= newNode
        front = newNode
        newNode->prev=NULL
```

# DEQUEUE – Linked Implementation

- Insert an element at the Rear of the DeQueue

```
Allocate space for a newNode of doubly linked list.
IF newNode == NULL, then
    print "Overflow”
ELSE
    IF rear== NULL, then
        rear = front = newNode
    ELSE
        newNode->prev= rear
        rear->next= newNode
        rear = newNode
        NewNode->next=NULL
```

# DEQUEUE – Linked Implementation

- Deletion an element From the Front of the DeQueue

```
 IF front == NULL, then
     print “underflow”
 ELSE
      initalize temp = front
      front = front->next
      IF front== NULL, then
                rear = NULL
      ELSE
                front ->prev= NULL
Deallocate space for temp
```

# DEQUEUE – Linked Implementation

- Deletion an element From the Rear of the DeQueue

```
IF front == NULL, then
    print “underflow”
ELSE
    initalize temp = rear
    rear = rear->prev
    IF rear== NULL, then
        front = NULL
    ELSE
        rear ->next= NULL
Deallocate space for temp
```