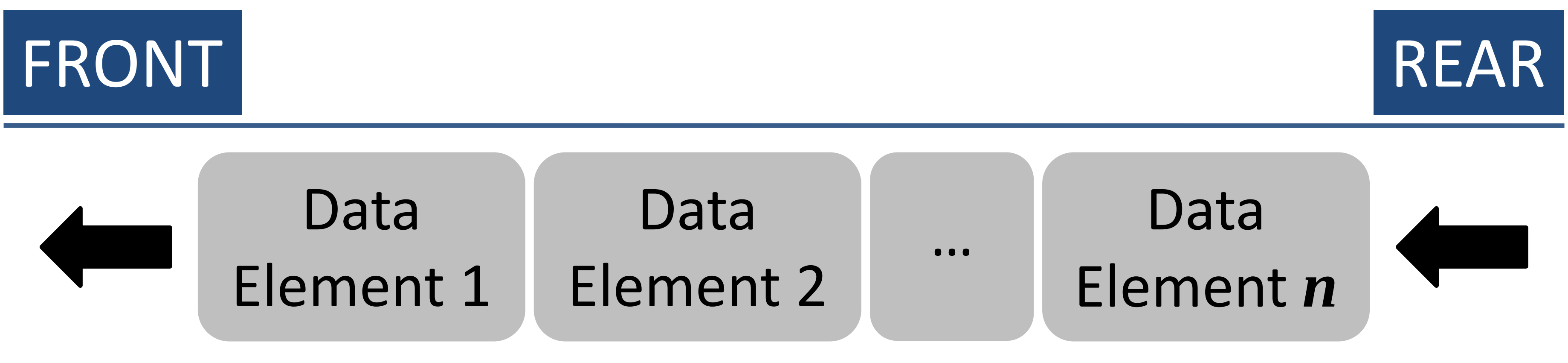
Queues

Introduction

- First in, first out (FIFO) structure (equivalent to Last in, last out (LILO) structure).
- An ordered list of homogeneous elements in which
  - Insertions take place at one end (REAR).
  - Deletions take place at the other end (FRONT).



Operations

- Two primary operations:
  - **enqueue()** – Adds an element to the ***rear*** of a queue.
  - **dequeue()** – Removes the ***front*** element of the queue.
- Other operations for effective functionality:
  - **isFull()** – Check if queue is full. ***OVERFLOW***
  - **isEmpty()** – Check if queue is empty. ***UNDERFLOW***
  - **size()** – Returns the number of elements in the queue.
  - **peek()** – Returns the element at the front of the queue.

Queue – Enqueue

enqueue(9)

front = -1

rear = -1

0	1	2	3	4

← 9

enqueue(1)

front = 0

rear = 0

9				
0	1	2	3	4

← 1

enqueue(3)

front = 0

rear = 1

9	1			
0	1	2	3	4

← 3

enqueue(6)

front = 0

rear = 2

9	1	3		
0	1	2	3	4

← 6

enqueue(4)

front = 0

rear = 3

9	1	3	6	
0	1	2	3	4

← 4

enqueue(2)

front = 0

rear = 4

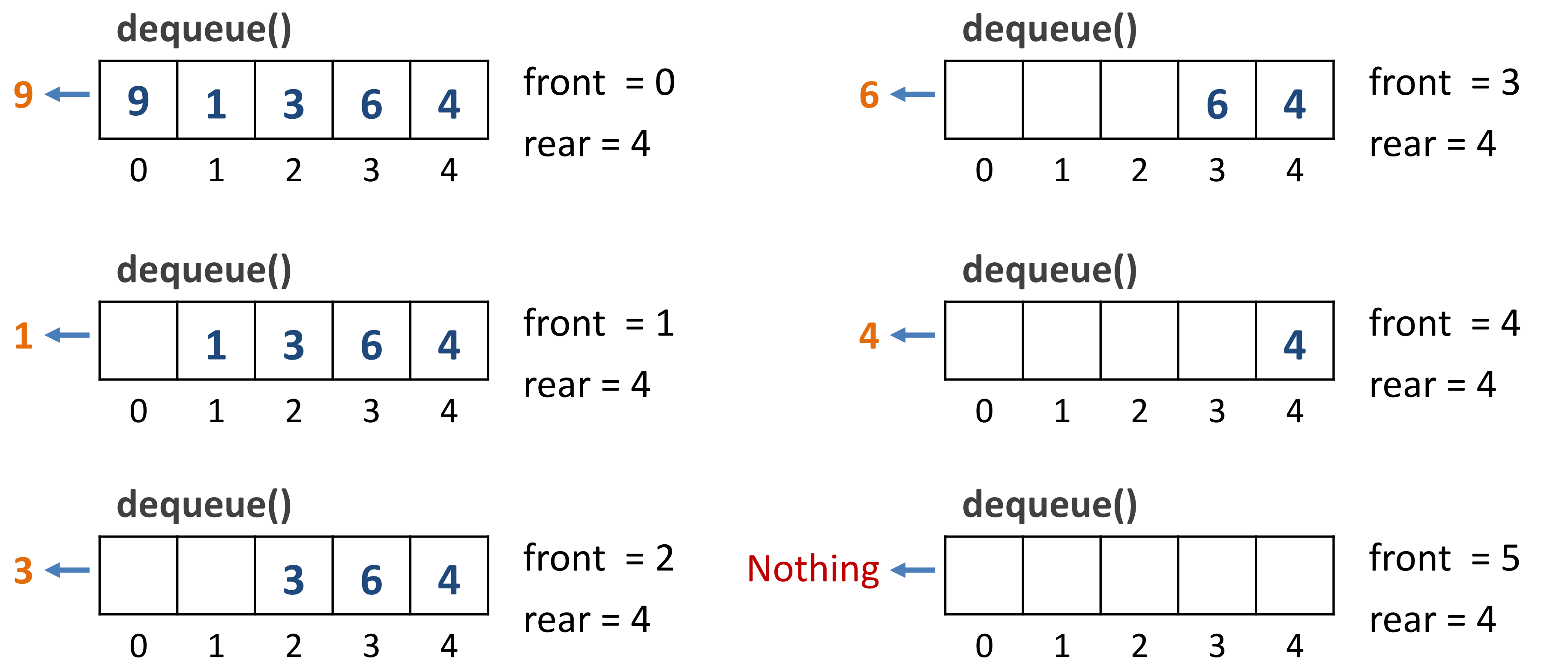
9	1	3	6	4
0	1	2	3	4

← 2

OVERFLOW

The queue is full, no more elements can be added. **OVERFLOW**

Queue – Dequeue



UNDERFLOW

The queue is empty, no element can be removed. **UNDERFLOW**

Operation	Output	Queue
enqueue(5)	-	(5)
enqueue(3)	-	(5,3)
dequeue()	5	(3)
enqueue(7)	-	(3, 7)
dequeue()	3	(7)
front()	7	(7)
dequeue()	7	()
dequeue()	ERROR	()
isEmpty()	TRUE	()
enqueue(9)	-	(9)
enqueue(7)	-	(9,7)
size()	2	(9,7)
enqueue(3)	-	(9,7,3)
enqueue(5)	-	(9, 7, 3, 5)
dequeue()	9	(7, 3, 5)

Queue as an ADT

- A queue is an ordered list of elements of same data type.
- Elements are always inserted at one end (rear) and deleted from another end (front).
- Following are its basic operations:
  - ***Q = init()*** – Initialize an empty queue.
  - ***size()*** – Returns the number of elements in the queue.
  - ***isEmpty(Q)*** – Returns "true" if and only if the queue Q is empty, i.e., contains no elements.

Contd…

- ***isFull(Q)*** – Returns "true" if and only if the queue Q has a bounded size and holds the maximum number of elements it can.
- ***front(Q)*** – Returns the element at the front of the queue Q.
- ***Q = enqueue(Q,x)*** – Inserts an element x at the rear of the queue Q.
- ***Q = dequeue(Q)*** – Removes an element from the front of the queue Q.
- ***print(Q)*** – Prints the elements of the queue Q from front to rear.

Implementation

- Using static arrays
  - Realizes queues of a maximum possible size.
  - Front is maintained at the smallest index and rear at the maximum index values in the array.
- Using dynamic linked lists
  - Choose beginning of the list as the front and tail as rear of the queue.

Static Array Implementation

Enqueue Operation

- Let,
  - QUEUE be an array with N locations.
  - FRONT and REAR points to the front and rear of the QUEUE.
  - ITEM is the value to be inserted.

1. If (REAR == N – 1)
2. Print[Overflow]
3. Else
4. If (FRONT == -1 && REAR == -1)
5. Set FRONT = 0 and REAR = 0.
6. Else
7. Set REAR = REAR + 1.
8. QUEUE[REAR] = ITEM.

Dequeue Operation

- Let,
  - QUEUE be an array with N locations.
  - FRONT and REAR points to the front and rear of the QUEUE.
  - ITEM holds the value to be deleted.

1. If (FRONT == -1 || FRONT > REAR)
2. Print[Underflow]
3. Else
4. ITEM = QUEUE[FRONT]
5. Set FRONT = FRONT + 1

Static Array Implementation

```
#define MAXLEN 100

typedef struct
{   int element[MAXLEN];
    int front, rear; } queue;

queue init ()
{   queue Q;
    Q.front = Q.rear = -1;
    return Q; }

int size( queue Q )
{ return ( Q.rear – Q.front + 1 );  }

int isEmpty ( queue Q )
{     return ((Q.front == -1) ||
              (Q.front > Q.rear)); }

int isFull ( queue Q )
{   return (Q. rear == MAXLEN - 1);  }

int front ( queue Q )
{    if (isEmpty(Q))
          printf("Empty queue\n");
     else
          return Q.element[Q.front]; }
```

Contd...

```
queue enqueue ( queue Q , int x )
{     if (isFull(Q))
          printf("OVERFLOW\n");
      else if (isEmpty(Q))
      {    Q.front = Q.rear = 0;
           Q.element[Q.rear] = x;
      }
      else
      {    ++Q.rear;
           Q.element[Q.rear] = x;
      }
      return Q; }
```

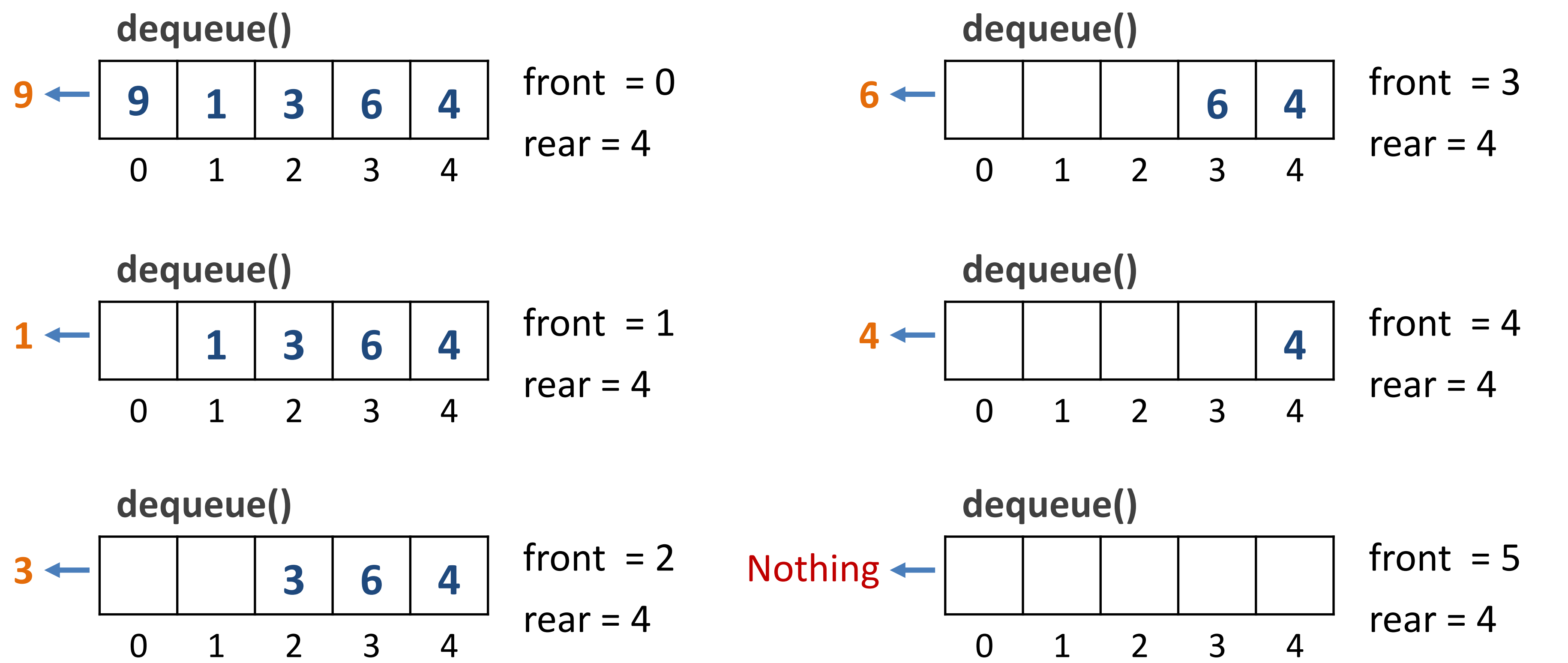
```
queue dequeue ( queue Q )
{     if (isEmpty(Q))
          printf("UNDERFLOW\n");
      else
          Q.front++;
      return Q; }
```

Contd…

```
void print ( queue Q )
{       int i;
        for (i = Q.front; i <= Q.rear; i++)
            printf("%d ",Q.element[i]); }

int main ()
{   queue Q;
    Q = init();
    Q = enqueue(Q,5);
    Q = enqueue(Q,3);
    Q = dequeue(Q);
    Q = enqueue(Q,7);
    Q = dequeue(Q);
    printf("Current queue : "); print(Q);
    printf(" with front = %d.\n", front(Q));
    Q = enqueue(Q,9);
    Q = enqueue(Q,3);
    Q = enqueue(Q,1);
printf("Current queue : "); print(Q);
printf(" with front = %d.\n", front(Q));
    printf("Size is %d.",size(Q));
return 0;    }
```

Problem…



UNDERFLOW

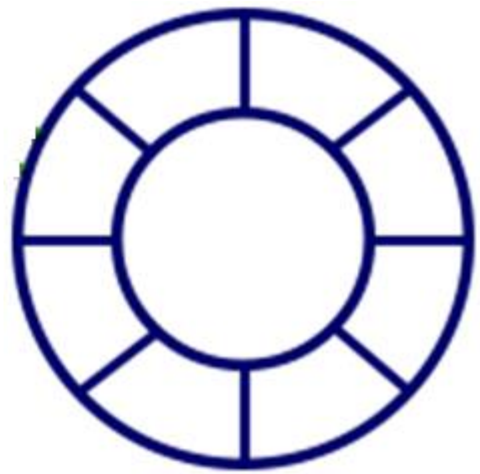
The queue is empty, no element can be removed. **UNDERFLOW**

Circular Queues

- The front and rear ends of a queue are joined to make the queue circular.
- Also known as circular buffer, circular queue, cyclic buffer or ring buffer.
- Overflow

  front == (rear+1) % MAXLEN
- Underflow

  (front == rear) && (rear == -1)



Enqueue Operation

- Let,
  - QUEUE be an array with MAX locations.
  - FRONT and REAR points to the front and rear of the QUEUE.
  - ITEM is the value to be inserted.

1. if (FRONT == (REAR+1)%MAX)
2. Print [Overflow]
3. else
4. Set REAR = (REAR+1)%MAX
5. Set QUEUE[REAR] = element
6. If (FRONT == -1)
7. Set FRONT = 0

Circular Queue – Enqueue

enqueue(9)

front = -1

rear = -1

0	1	2	3	4

← 9

enqueue(1)

front = 0

rear = 0

0	1	2	3	4
9				

← 1

enqueue(3)

front = 0

rear = 1

0	1	2	3	4
9	1			

← 3

enqueue(6)

front = 0

rear = 2

0	1	2	3	4
9	1	3		

← 6

enqueue(4)

front = 0

rear = 3

0	1	2	3	4
9	1	3	6	

← 4

enqueue(2)

front = 0

rear = 4

0	1	2	3	4
9	1	3	6	4

← 2

OVERFLOW

As F == [(R+1)%5 = 5 % 5 = 0]

So, **OVERFLOW**

Dequeue Operation

- Let,
  - QUEUE be an array with MAX locations.
  - FRONT and REAR points to the front and rear of the QUEUE.
  - ITEM holds the value to be deleted.

```
1.  if ((FRONT == REAR) && (REAR == -1))
2.          Print [Underflow]
3.  else
4.          ITEM = Q[FRONT]
5.          If (FRONT == REAR)
6.                  FRONT = REAR = -1
7.          Else
8.                  FRONT = (FRONT + 1) % MAX
```

Circular Queue – Dequeue

dequeue()

9 ←

0	1	2	3	4
9	1	3	6	4

front = 0
rear = 4

dequeue()

1 ←

0	1	2	3	4
	1	3	6	4

front = 1
rear = 4

dequeue()

3 ←

0	1	2	3	4
		3	6	4

front = 2
rear = 4

dequeue()

6 ←

0	1	2	3	4
			6	4

front = 3
rear = 4

dequeue()

4 ←

0	1	2	3	4
				4

front = 4
rear = 4

dequeue()

Nothing ←

0	1	2	3	4

front = -1
rear = -1

UNDERFLOW

As, F == R and R == -1.
So, **UNDERFLOW**

Static Array Implementation – Circular Queues

```
#define MAXLEN 100

typedef struct
{   int element[MAXLEN];
    int front, rear, size;
} queue;

queue init ()
{   queue Q;
    Q.front = Q.rear = -1;
    Q.size = 0;    return Q; }

int size( queue Q )
{ return ( Q.size );  }

int isEmpty ( queue Q )
{     return ((Q.rear == -1) &&
                 (Q.front == Q.rear)); }

int isFull ( queue Q )
{   return (Q.front == ((Q.rear+1) %
                 MAXLEN));  }

int front ( queue Q )
{    if (isEmpty(Q))
          printf("Empty queue\n");
     else
          return Q.element[Q.front]; }
```

Contd...

```
queue enqueue ( queue Q , int x )
{    if (isFull(Q))
          printf("OVERFLOW\n");
     else
     {    ++Q.size;
          Q.rear = ((Q.rear+1) % MAXLEN);
          Q.element[Q.rear] = x;
          if(Q.front == -1)
               Q.front = 0;
     }
     return Q; }
```

Contd…

```
queue dequeue ( queue Q )
{     if (isEmpty(Q))
            printf("UNDERFLOW\n");
      else
       {   Q.size--;
            if(Q.front == Q.rear)
                  Q.front = Q.rear = -1;
            else
                  Q.front = ((Q.front+1) % MAXLEN);
      }
      return Q; }
```

Contd…

```
void print ( queue Q )
{       int i;
        if(Q.front > Q.rear)
        {  for (i = Q.front; i < MAXLEN; i++)
                printf("%d ",Q.element[i]);
             for (i = 0; i <= Q.rear; i++)
                printf("%d ",Q.element[i]);
        }
        else
        {  for (i = Q.front; i <= Q.rear; i++)
                  printf("%d ",Q.element[i]); }
}
```

Contd...

```
int main ()
{   queue Q;
    Q = init();
    Q = enqueue(Q,5);
    Q = enqueue(Q,3);
    Q = dequeue(Q);
    Q = enqueue(Q,7);
    Q = dequeue(Q);
    printf("Current queue : "); print(Q);
    printf(" with front = %d.\n", front(Q));
    Q = enqueue(Q,9);
    Q = enqueue(Q,3);
    Q = enqueue(Q,1);
    printf("Current queue : "); print(Q);
    printf(" with front = %d.\n", front(Q));
    printf("Size is %d.",size(Q));
    return 0;   }
```

Output

```
$ g++ queueCircular.c

hp@hp-PC ~
$ ./a.exe
Current queue : 7  with front = 7.
Current queue : 7 9 3 1  with front = 7.
Size is 4.
hp@hp-PC ~
$ g++ queue.c

hp@hp-PC ~
$ ./a.exe
Current queue : 7  with front = 7.
OVERFLOW
Current queue : 7 9 3  with front = 7.
Size is 3.
```

Dynamic Linked List Implementation

Enqueue Operation

- Let,
  - FRONT and REAR points to the front and rear of the QUEUE.
  - ITEM is the value to be inserted.

1. Create a node pointer (temp).
2. temp[data] = ITEM.
3. temp[next] = NULL.
4. If FRONT == NULL
5. FRONT = REAR = temp.
6. Else
7. REAR[next] = temp.
8. REAR = temp.

Dequeue Operation

- Let,
  - FRONT and REAR points to the front and rear of the QUEUE.
  - temp points to the element deleted from the front of the queue.

1. if (FRONT == NULL)
2. Print [Underflow]
3. else
4. Initialize a node pointer (temp) with FRONT.
5. if (FRONT == REAR)
6. FRONT = REAR = NULL
7. else
8. FRONT = FRONT[next]
9. Release the memory location pointed by temp.

Dynamic Linked List Implementation

```
1.  struct node
2.  {   int data;
3.      struct node *next, *prev;
4.  } *front, *rear;

5.  void init()
6.  {   front = rear = NULL;  }
```

Contd…

```
void enqueue(int num)
{ struct node *temp = (struct node *) malloc (sizeof(struct node));
   temp -> data = num;
   if(front == NULL)
   {    temp -> prev = temp;
        temp -> next = temp;
        front = rear = temp;  }
   else
   {    temp -> prev = rear;
         rear -> next = temp;
         temp -> next = front;
        front -> prev = temp;
        rear = temp;   }        }
```

Contd…

```
void dequeue()
{   if(front == NULL)
        printf("\nQueue is empty.\n");
    else
    {   struct node *temp = front;
        if (front == rear)
            front = rear = NULL;
        else
        {   rear -> next = front -> next;
            front = front -> next;
            front -> prev = rear;   }
    free(temp);     }       }
```

Contd...

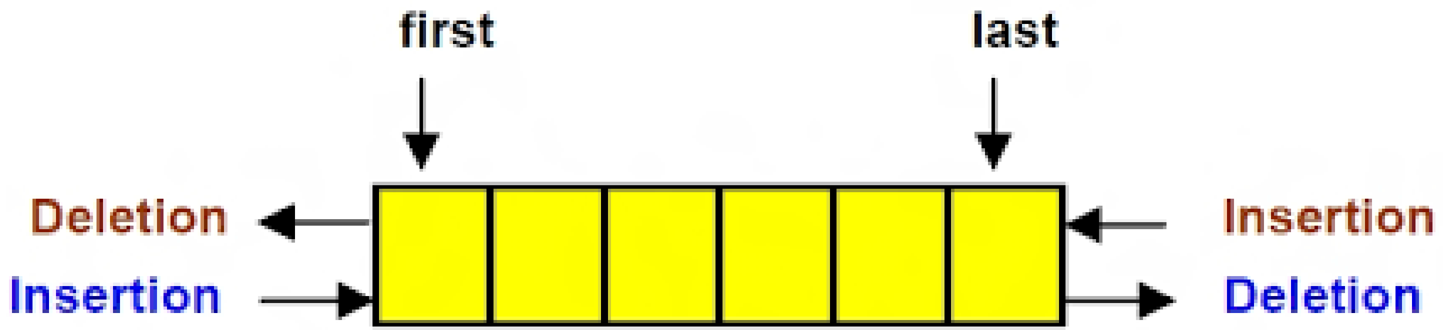
```c
void print()
{  printf("\nfront --> ");
   if (front != NULL)
   {  struct node *temp = front;
      while(temp != rear)
      {  printf("%d --> ",temp->data);
         temp = temp->next;   }
      printf("%d --> ",temp->data);
   }  printf("rear\n");
}
int main ()
{   init();
    enqueue(5);  enqueue(3);
    dequeue();
    enqueue(7);
    dequeue();
    printf("Current queue : ");
    print();
    enqueue(9);  enqueue(3);
    enqueue(1);
    printf("Current queue : ");
    print();
    return 0;
}
```

```
Current queue :
front --> 7 --> rear
Current queue :
front --> 7 --> 9 --> 3 --> 1 --> rear
```

Deque

- **D**ouble-**e**nded **que**ue.
- Generalization of queue data structure.
- Elements can be added to or removed from either of the two ends.
- A hybrid linear structure that provides all the capabilities of stacks and queues in a single data structure.
- Does not require the LIFO and FIFO orderings.

Contd…



Types

- Input-restricted deque.
  - Deletion can be made from both ends, but insertion can be made at one end only.
- Output-restricted deque.
  - Insertion can be made at both ends, but deletion can be made from one end only.

Priority Queues

- Another variant of queue data structure.
- Each element has an associated priority.
- Insertion may be performed based on the priority.
- Deletion is performed based on the priority.
- Elements having the same priority are served or deleted according to first come first serve order.
- Two types:
  - Min-priority queues (Ascending priority queues)
  - Max-priority queues (Descending priority queues)

Implementation

- Array representation: Unordered and Ordered
- Linked-list representations: Unordered and Ordered
- Unordered does not consider priority during insertion, instead insertion takes place at the end.
- Ordered considers priority during insertion and inserts an element at correct place as per min or max priority.
- Note
  - Either insertion or deletion take linear time in the worst case.
  - Priority queues are often implemented with heaps.

Contd…

- Using arrays

```
int prioQ[10][2];
```

- Using linked list

```
struct node
{    int data, priority;
     struct node *next;    };
```

- The methods for insertion and deletion have to be used based on the ordered or unordered version.

Static Array Implementation

Unordered

- Insertion will take place at the maximum array index or at the end.
- Deletion
  - Min-priority
    - Find the minimum element in the array.
  - Max-priority
    - Find the maximum element in the array.
  - Delete the element from the array (use deletion algorithms covered in array section).
    - An efficient way is to replace the minimum or the maximum element with the last array element.

Ordered

- Insertion will take place at the appropriate index within an array following ascending or descending order of priorities (use insertion algorithms covered in array section).
- Deletion

	Min-priority	Max-priority
Ascending order	Delete element at index '0'.	Delete element at the maximum array index.
Descending order	Delete element at the maximum array index.	Delete element at index '0'.

Dynamic Linked List Implementation

Unordered

- Insertion will take place at the end of the list.
- Deletion
  - Min-priority
    - Find node in the list with the minimum element.
  - Max-priority
    - Find node in the list with the maximum element.
  - Delete the specific node from the list (use deletion algorithms covered in linked list section).

Ordered

- Insertion will take place at the appropriate position in the list following ascending or descending order of priorities (use insertion algorithms covered in linked list section).
- Deletion

	Min-priority	Max-priority
Ascending order	Delete the node pointed by 'head'.	Delete the last node.
Descending order	Delete the last node.	Delete the node pointed by 'head'.

Example (Using array)

Element to be deleted is replaced with the last array element.

Operation	Argument	Return Value	Size	Contents	
				Unordered	Ordered
Insert	P		1	**P**	**P**
Insert	Q		2	P **Q**	P **Q**
Insert	E		3	P Q **E**	**E** P Q
Remove MAX		**Q**	2	P E	E P
Insert	X		3	P E **X**	E P **X**
Insert	A		4	P E X **A**	**A** E P X
Insert	M		5	P E X A **M**	A E **M** P X
Remove MAX		**X**	4	P E M A	A E M P
Insert	P		5	P E M A **P**	A E M P **P**
Insert	L		6	P E M A P **L**	A E **L** M P P
Insert	E		7	P E M A P L **E**	A E **E** L M P P
Remove MAX		**P**	6	E E M A P L	A E E L M P

Thank You