Pune Vidyarthi Griha's

COLLEGE OF ENGINEERING, NASHIK – 3.

"QUEUE"

By

Prof. Anand N. Gharu

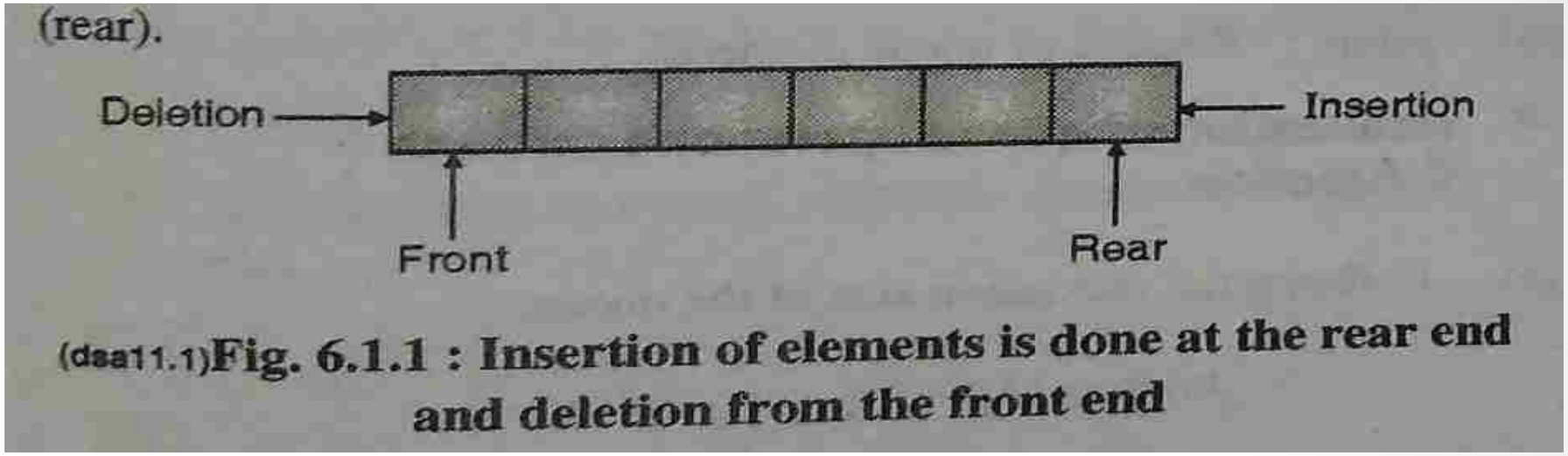
(Assistant Professor)

PVGCOE Computer Dept.

01 August 2019

Introduction of Queue

- "A queue is **an ordered list** in which all insertions are done at one end, called rear and deletions at another end called front"



(daa11.1)**Fig. 6.1.1 : Insertion of elements is done at the rear end and deletion from the front end**

- Queue when implemented using arrays has some drawbacks which can be avoided by circular queue
- Queue is used in many applications such **as *as simulation, priority queue, job queue etc***

Introduction of Queue

- One of the most **common data processing structures**
- Frequently used in most of the system software's like operating systems, Network and Database implementations and in many more other areas
- **Very useful in time-sharing and distributed computer systems** where many widely distributed users share the system simultaneously

Array representation and implementation of queue

- An array representation of queue require three entities :

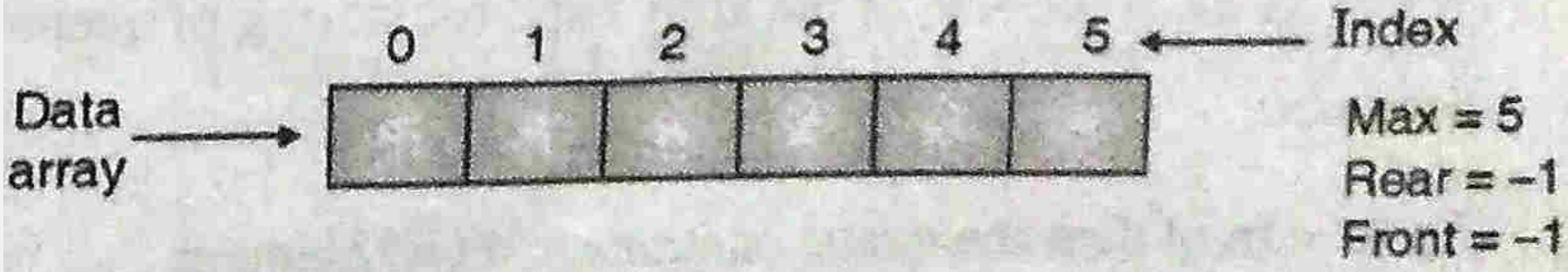
1. An array to hold queue element
2. A variable to hold index of the front element
3. A variable to hold index of the rear element

A queue datatype may be defined formally as follows :

```
# define MAX 30
typedef struct queue
{
        int data[MAX];
   int front, rear;
} queue;
```

Array representation and implementation of queue

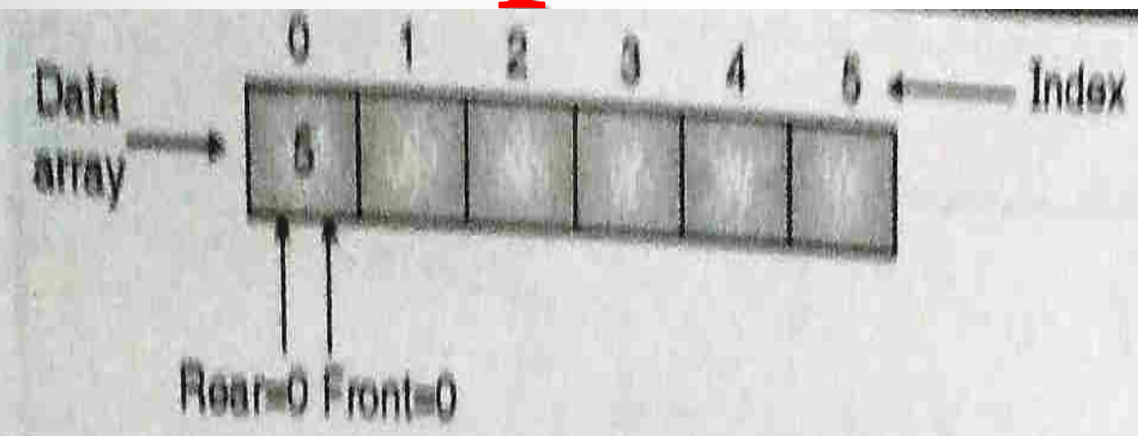
During initialization of a queue, its front and rear are set to –1.



(dsa11.2)**Fig. 6.1.2 : An empty queue after initialization**

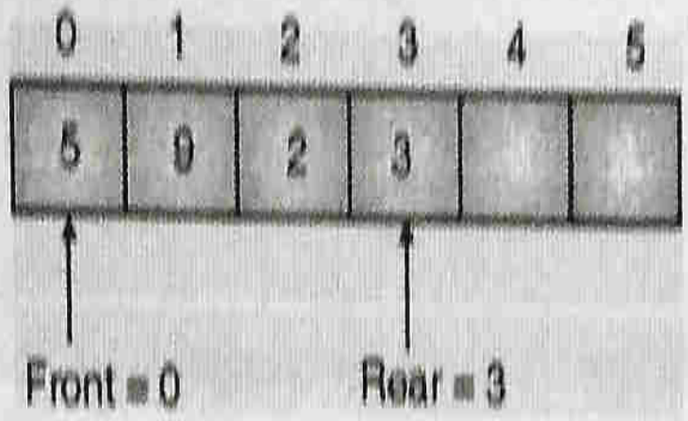
Fig. 6.1.3 shows the status of a queue after insertion of the element '5'.

Array representation and implementation of queue

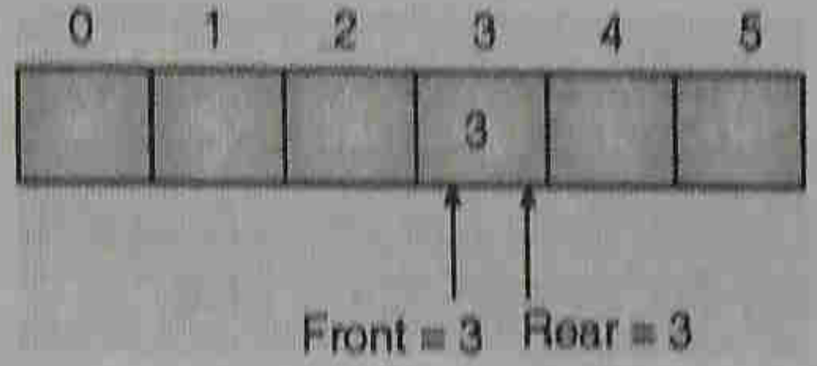


(dss11.5)**Fig. 6.1.3 : A queue after insertion of the first element '5'**

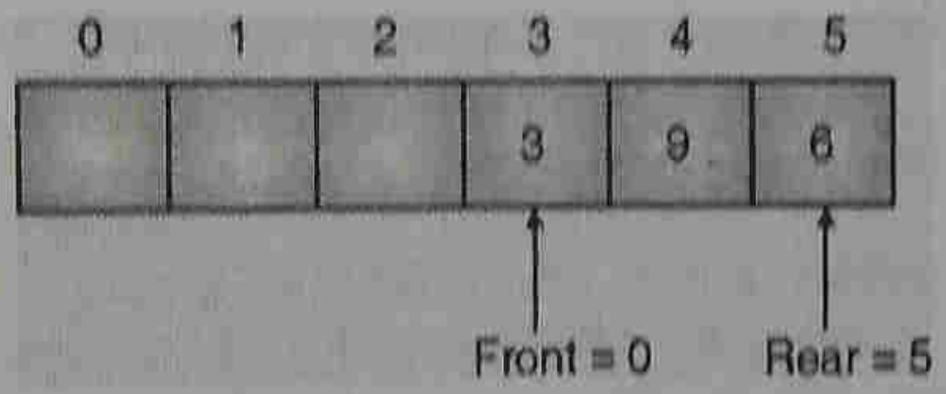
- On subsequent insertions, front remains at the same place, where rear advances.



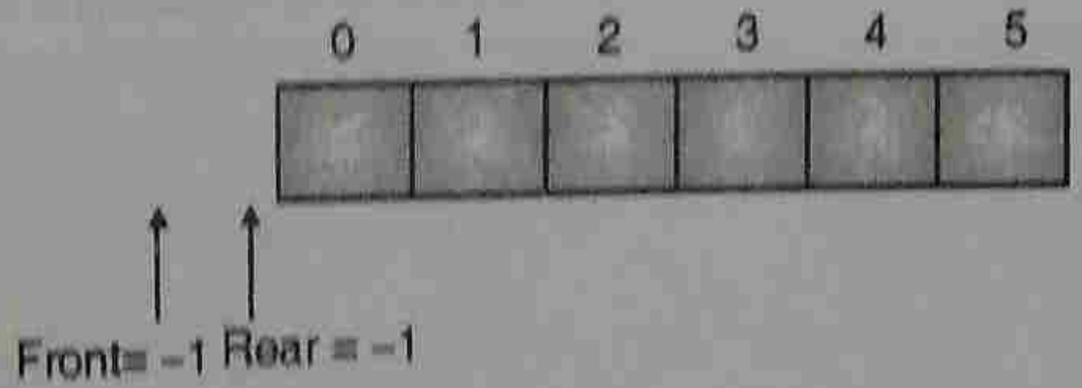
(dss11.4)**Fig. 6.1.4 : A queue after insertion of four elements**



(dss11.6)**Fig. 6.1.5 : A queue after 3 successive deletions**



(dss11.6)**Fig. 6.1.6 : A queue after 2 successive insertions**



(dss11.7)**Fig. 6.1.7 : A queue after 3 successive deletions**

Array representation and implementation of queue

Following points can be observed :

(1) If the queue is empty then front = –1 and rear = –1.

(2) If the queue is full then rear = MAX – 1.

(where MAX is the size of the array used for storing of queue elements).

(3) If rear = front then the queue contains just one element.

(4) If rear > front then queue is non-empty.

Problem with the above representation of the queue :

Note : Problem of overflow : Refer to the Fig. 6.1.6, the queue has become full as rear = MAX –1. There are three vacant spaces (location 0, 1, 2) but these spaces can not be utilized.

Operation on queue implemented using array

A set of useful operations on a queue includes :

(1) initialize( ) : Initializes a queue by setting the value of rea and front to –1.

(2) enqueue( ) : Inserts an element at the rear end of the queue.

(3) dequeue( ) : Deletes the front element and returns the same.

(4) empty( ) : It returns true(1) if the queue is empty and returns false(0) if the queue is not empty.

(5) full( ) : It return true(1) if the queue is full and returns false(0) if the queue is not full.

(6) print( ) : Printing of queue elements.

Operation on queue implemented using array

(a) Defining the maximum size of the queue.

```
#define MAX 50
```

(b) Data structure declaration

```
typedef struct Q
{
        int R, F;
        int data[MAX];
} Q;
```

"R" and "F" store the index of the rear and the front element respectively. "data[MAX]" is used to store the queue elements.

(c) Declaring a queue type variable.

$Q\ q_1, q_2$;

q_1 and q_2 are queue type variables

(d) Declaring a queue type pointer variable.

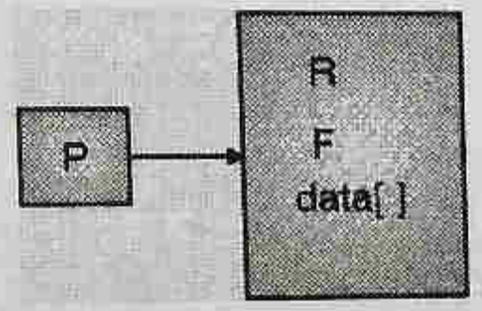
$Q\ q_1, *P;$

$P = \&q_1$

A pointer variable "P" can be used to store the address of a queue type variable.

☞ **'C' function to initialize( ).**

```
void initialize(Q *P)
{
        P→R = -1;
        P→F = -1;
}
```



(dsa11.8)**Fig. 6.2.1 : Elements of the structure 'Q' can be accessed through the pointer P (P → R, P → F, P → data[ ])**

Operation on queue implemented using array

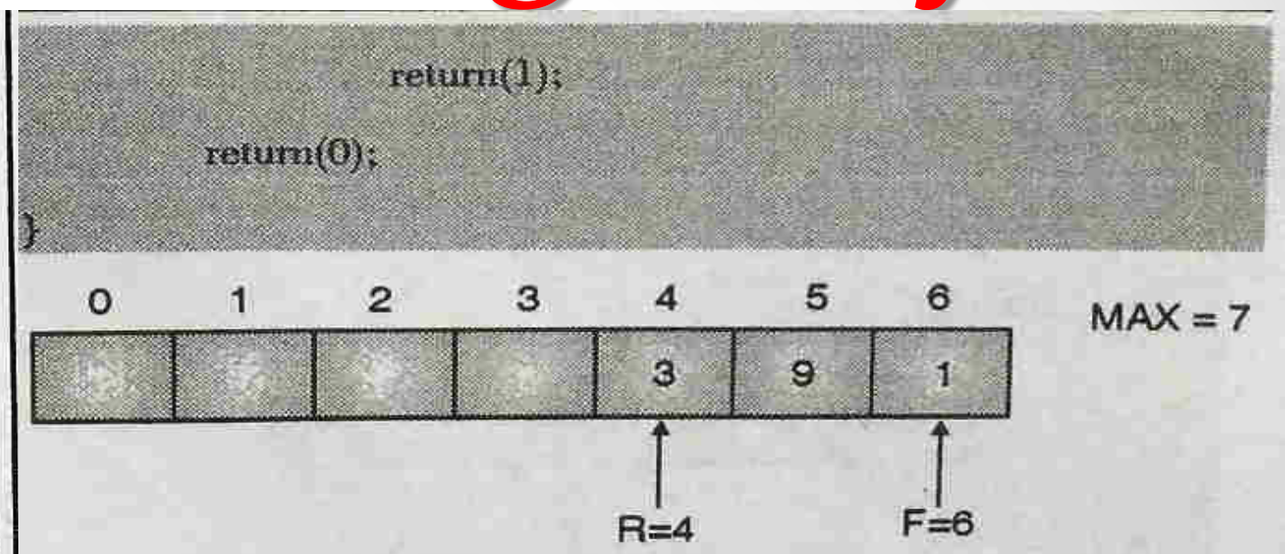
☞ **'C' function to check whether queue is empty or not.**

```
int empty(Q *P)
{
        if(P → R == – 1)
                return(1);
        return(0);
}
```

Whenever the queue is empty, P → R (i.e. rear field (R) of the queue type variable whose address is stored in P) or P → F will be – 1.

☞ **'C' function to check whether queue is full or not.**

```
int full(Q *P)
{
        if(P → R == MAX – 1)
                return(1);
        return(0);
}
```



(dsa11.9)**Fig. 6.2.2 : The queue is full as there is no space left for further insertion enqueue( )**

☞ **Algorithm for insertion in a queue**

(1) Inserting in an empty queue.

R = F = 0;

data[R] = x /* x is the element to be inserted */

/* Both R and F will point to the only element of the queue */

(2) Inserting in a non-empty queue.

R = R + 1;

data[R] = x

/* rear is advanced by 1 and the element x is stored at the rear end of the queue */.

Note : Insertion should be carried out after it is checked that the queue is not full.

Operation on queue implemented using array

☞ **'C' function for insertion in a queue using array.**

```
void enque(Q *P, int x)
{
        if(P → R == – 1)  /* empty queue */
        {
                P → R = P → F = 0;
                P → data[ P→ R] = x;
        }
        else
        {
                P → R = P → R + 1;
                P → data[P → R]   = x;
        }
}
```

☞ **'C' function for deletion in a queue using array dequeue( )**

Algorithm for deletion from a queue :

(1) Deletion of the last element or only element(R = F)

(a) x = data[F]

(b) R = F = – 1;

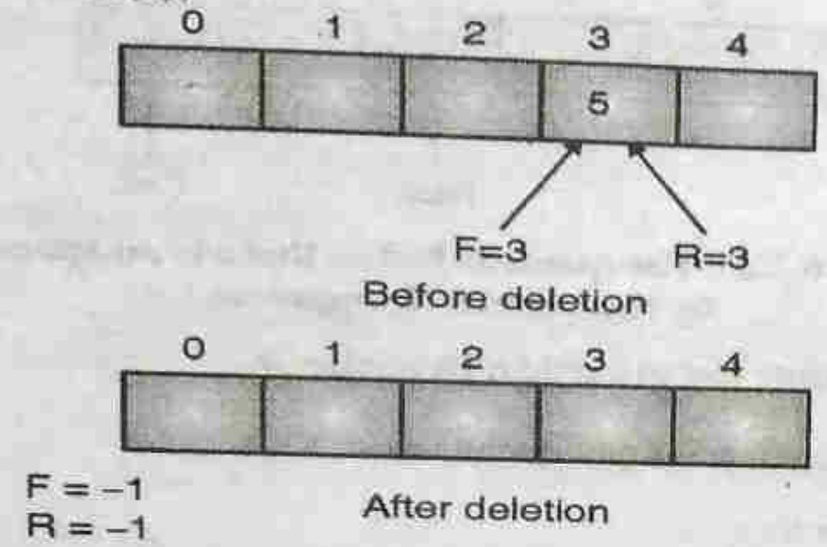
(c) return(x)

(2) Deletion of the element when the queue length > 1

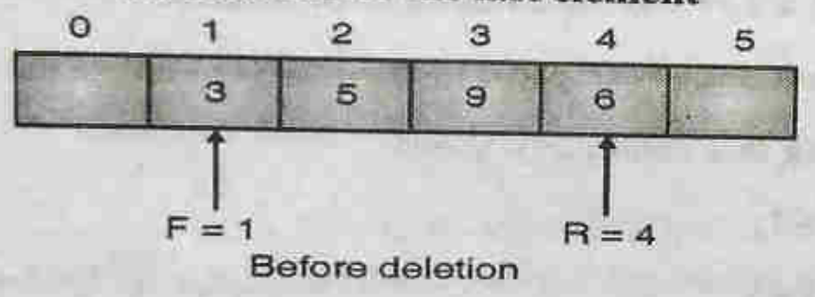
(a) x = data[F]

(b) F = F + 1

(c) return(x);



(a) Deletion of the last element



(b) Deletion of an element from the queue when the queue length > 1

Operation on queue implemented using array

☞ 'C' function for deletion in a queue.

```
int dequeue(Q *P)
{
        int x;
        x=P->data[P->F];
        if(P->R==P->F)
        {
                P->R=-1;
                P->F=-1;
        }
        else
                P->F=P->F+1;
        return(x);
}
```

Note : Deletion of an element from the queue should be carried out after checking that it is not empty.

Queue as an ADT

- **Create : Create creates an empty queue, Q**
- **Add (i,Q) : Add adds the element i to the rear end of queue, Q and** returns the new queue
- **Delete (Q) : Delete takes out an element i from the**
  - **front end of queue** and returns the resulting queue
- **Front(Q) : Front returns the element i which is at the front position of** queue
- **Is_Empty_Q(Q) : Is_Empty_Q returns true if queue is**
  - **empty otherwise** returns false

Queue as an ADT

6.2.2 Queue as an ADT

☞ Datatype for queue in an array

```
#define MAX 30 /* A queue with maximum of 30 elements */
typedef struct queue
{           int data [MAX];
            int rear, front;
}           queue;
```

☞ Operations on a queue

i) initialize( ) : Make the queue empty

ii) empty( ) : Determine if queue is empty.

iii) full( ) : Determine if queue is full.

iv) enqueue( ) : Insert an element at the rear end of the queue.

v) dequeue( ) : Delete the front element.

vi) print( ) : Print elements of the queue.

☞ Prototype of functions used for various operations on queue

— void initialise(queue *p);

— int empty(queue *p);

— Function returns 1 or 0, depending on whether the queue pointed by p is empty or not.

— int full(queue *p);

Queue as an ADT

- function returns 1 or 0, depending on whether the queue is full or not.
- void enqueue(queue *p, int x);
- int dequeue(queue *p);
- function returns the front element.
- void print(queue *p);
- enqueue( ) operation will cause an overflow if the queue is full.
- Dequeue( ) operation will cause an underflow if the queue is empty( ).

Queue as an ADT

- **1. Create()**

  Creates and initializes new queue that is empty. It does not require any parameter and returns an empty queue.

- **2. Enqueue(item)**

  Adds a new element to the rear of the queue. It requires the element to be added and returns nothing.

- **3. Dequeue()**

  Removes the element from front of the queue It does not require any parameter and returns the deleted item.

- **4. isEmpty()**

  Checks whether the queue is empty or not. It does not require any parameter and returns a boolean value.

- **5. isFull**

  Checks whether the queue is full or not. It does not require any parameter and returns a boolean value.

- **6. Size()**

  Returns the total number of elements present in the queue. It does not require any parameter and returns an integer.

STACK VS QUEUE

#	STACK	QUEUE
1	Objects are inserted and removed at the same end.	Objects are inserted and removed from different ends.
2	In stacks only one pointer is used. It points to the top of the stack.	In queues, two different pointers are used for front and rear ends.
3	In stacks, the last inserted object is first to come out.	In queues, the object inserted first is first deleted.
4	Stacks follow Last In First Out (LIFO) order.	Queues following First In First Out (FIFO) order.
5	Stack operations are called push and pop.	Queue operations are called enqueue and dequeue.
6	Stacks are visualized as vertical collections.	Queues are visualized as horizontal collections.
7	Collection of dinner plates at a wedding reception is an example of stack.	People standing in a file to board a bus is an example of queue.

Queue Example

Example 6.2.1

Consider the following queue of characters, implemented as array of six memory locations :-

Front = 2, Rear = 3

Queue : –, A, D, –, –, –

Where '–' denotes empty cell. Describe the queue as the following operations take place

(i) Add 'S' (ii) Add 'J'

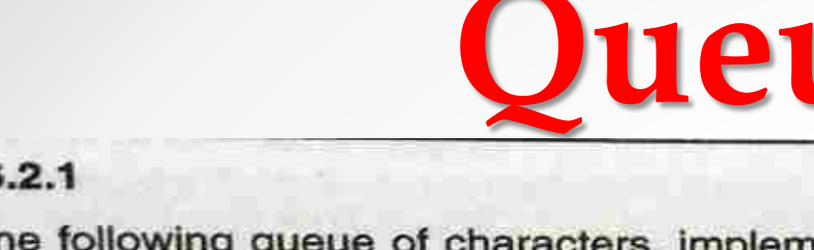
(iii) Delete two letters

(iv) Shift towards left to bring all free spaces to the right side

(v) Insert M, H, I and delete on letter.

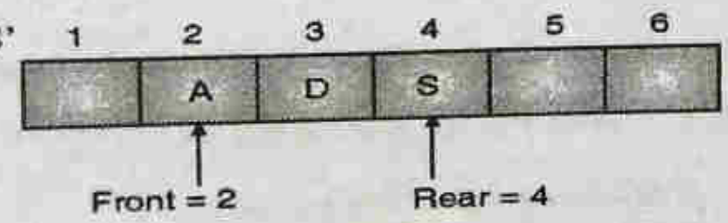
Solution :

Initial queue :



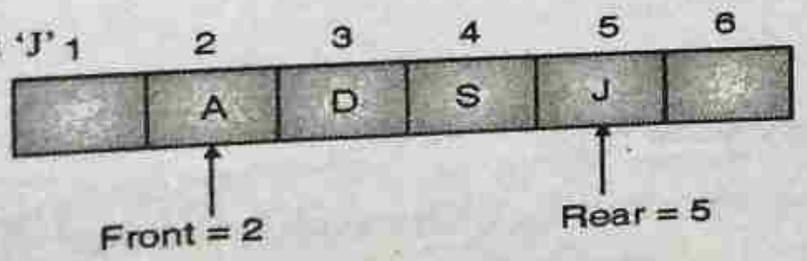
(a)

(i) Add 'S'



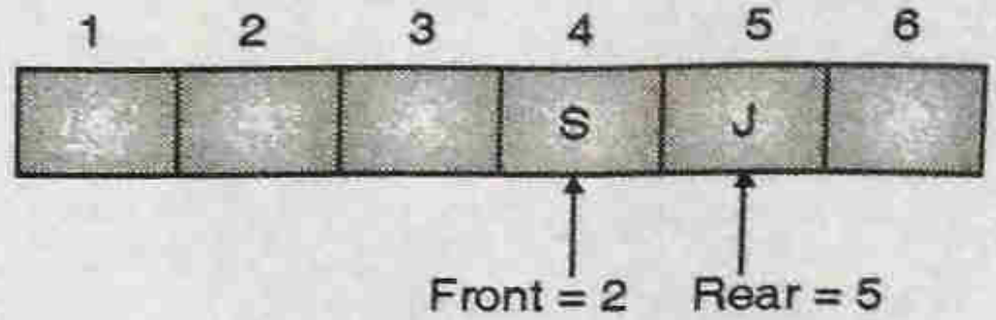
(b)

(ii) Add 'J'



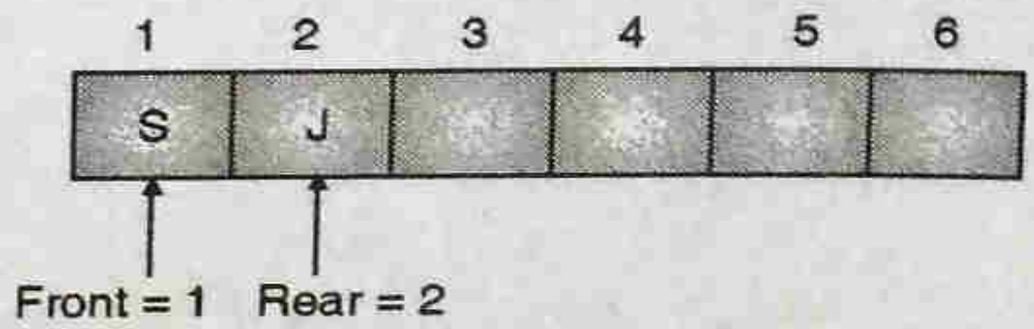
(c)

(iii) Delete two letters



(d)

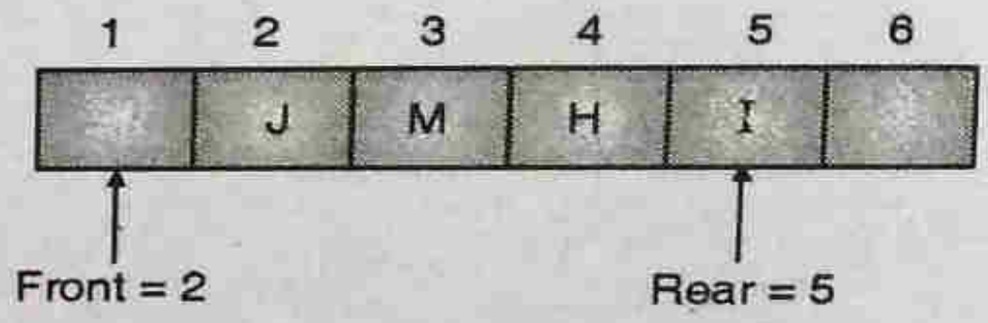
(iv) Shift towards left to bring all free spaces to the right.



(e)

(dsa11.11)**Fig. Ex. 6.2.1**

(v) Insert M, H, I and delete one letter.



(f)

(dsa11.11)**Fig. Ex. 6.2.1**

Queue Example

Q. 5.2.2 Consider the following queue, where queue is a circular queue having 6 memory cells.

Front = 2, Rear = 4.

Queue : __, A, C, D, __, __

Describe queue as following operation take place :

F is added to the queue

Two letters are deleted

R is added to the queue

S is added to the queue

One letter is deleted **(4 Marks)**

Soln. :

	A	C	D		
	F		R		

F is added

	A	C	D	F	
	F			R	

Two letters are deleted

			D	F	
			F	R	

R is added to queue

			D	F	R
			F		R

S is added to queue

S			D	F	R
R			F		

One letter is deleted

S				F	R
R				F	

Queue Example

Example 3.2.1

A linear queue using array has a size of 3. Perform the following on this queue and show the sequence of steps with necessary diagrams indicating values of front, rear and contents of queue :

(i) insert 10

(ii) insert 20

(iii) insert 30

(iv) delete an element

(v) insert 40

(vi) delete an element

(vii) delete an element

(viii) delete an element

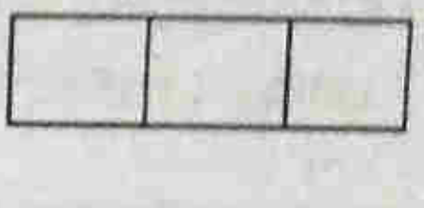
(ix) insert 50

Queue Example

Solution :

Operation **Queue**

Initial



rear, front

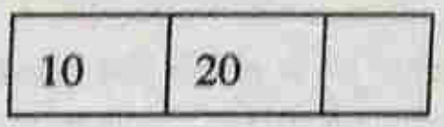
= –1 = –1

Insert 10

rear front

= 0 = 0

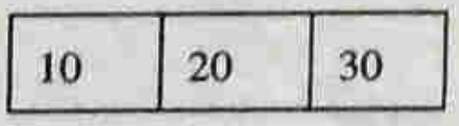
Insert 20



front rear

= 0 = 1

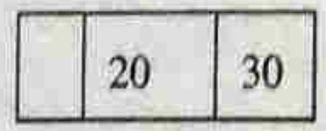
Insert 30



front rear

= 0 = 2

Delete an element



Front rear

= 1 = 2

Delete an element

front rear

= 2 = 2

Delete an element

front, rear

= –1 = –1

Insert 50

front rear

= 0 = 0

Operation on queue implemented using Linked list

A set of useful operations on a queue includes :

(1) initialize( ) – initializes a queue by setting the value or rear and front pointer to "NULL".

(2) enqueue( ) – inserts an element at the rear end of the queue.

(3) dequeue( ) – deletes the front element and returns the same.

(4) empty( ) – It returns true(1) if the queue is empty and returns false(0) if the queue is not empty.

(5) Print( ) – It prints the queue elements from front to rear.

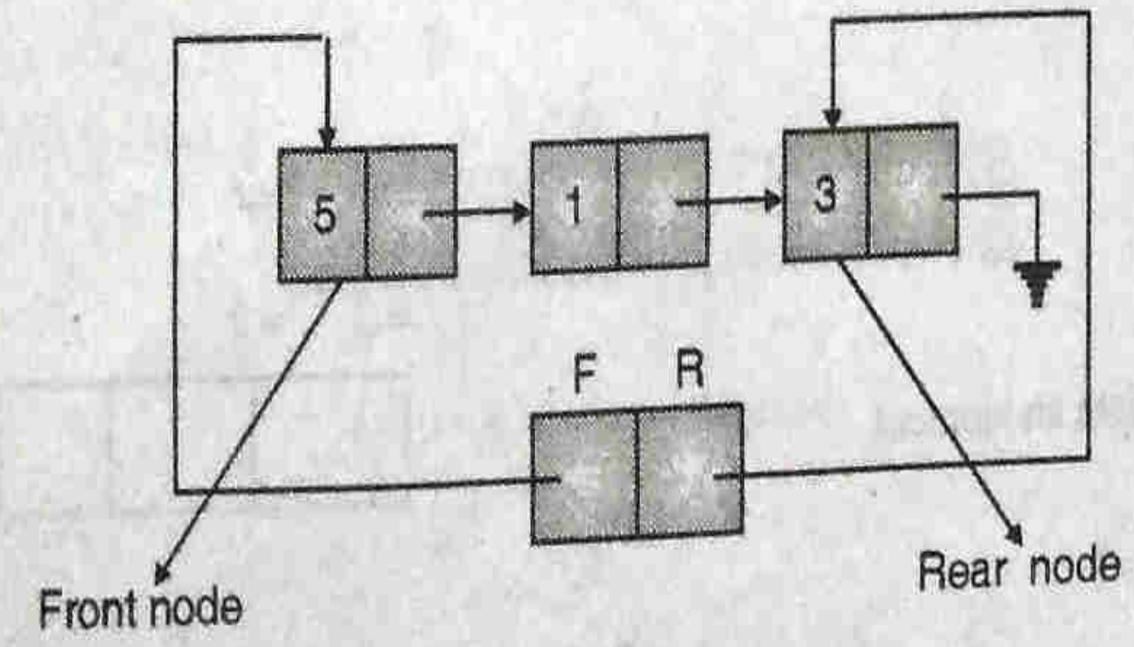
(a) Data structure declaration.

```
typedef struct node
{
        int data;
        struct node *next;
}node;
typedef struct Q
{
        node *R;
        node *F;
} Q;
```

(b) Declaring a queue type variable

Q q_1, q_2 ;

Element q_1 of the type Q(queue)



(dsa11.12)**Fig. 6.2.4 : Memory representation of a queue**

Data field of the front node is $(q_1.F) \rightarrow$ data

Next field of the front node is $(q_1.F) \rightarrow$ next

Data field of the rear node is $(q_1.R) \rightarrow$ data

Next field of the rear node is $(q_1.R) \rightarrow$ next;

Operation on queue implemented using Linked list

(c) Declaring a queue type pointer

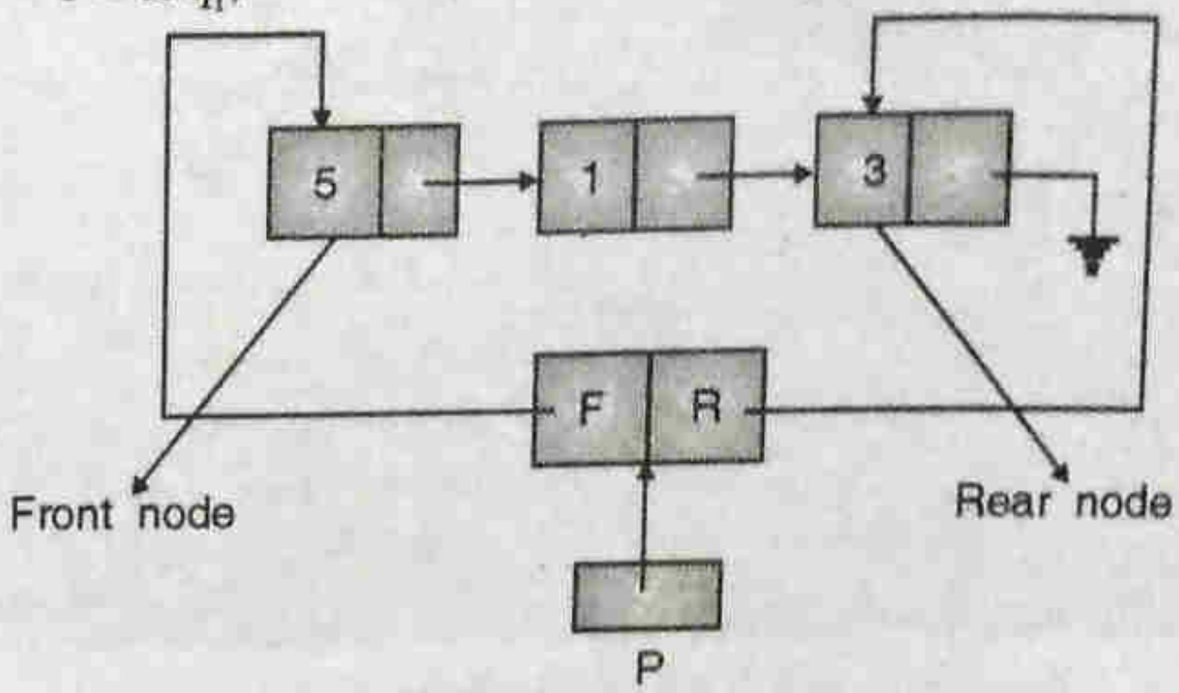
Q q_1, *P;

A queue type variable

P = & q_1;

A queue type pointer

/* P points to q_1 */



(dsa11.13)**Fig. 6.2.5 : Memory representation of a queue with its address in P**

Data field of the front node is (P → F) → data;

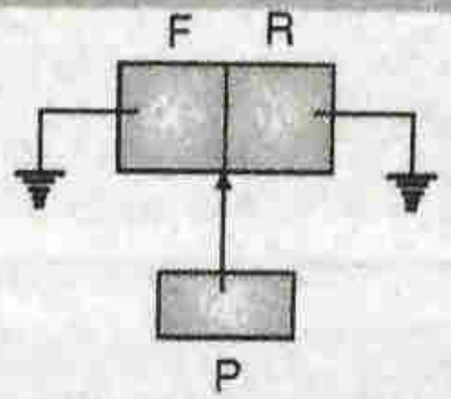
Next field of the front node is (P → F) → next;

Data field of the rear node is (P → R) → data;

Next field of the rear node is (P → R) → next;

☞ **'C' function to initialize queue.**

```
void initialize(Q *P)
{
        P → R = NULL;
        P → F = NULL;
}
```



(dsa11.14)**Fig. 6.2.6 : Initial state of a queue, queue is addressed through the pointer P**

☞ **'C' function to check whether queue is empty or not.**

```
int empty(Q *P)
{
        if(P → R == NULL)          /* empty queue */
                return(1);
        return(0);
}
```

Operation on queue implemented using Linked list

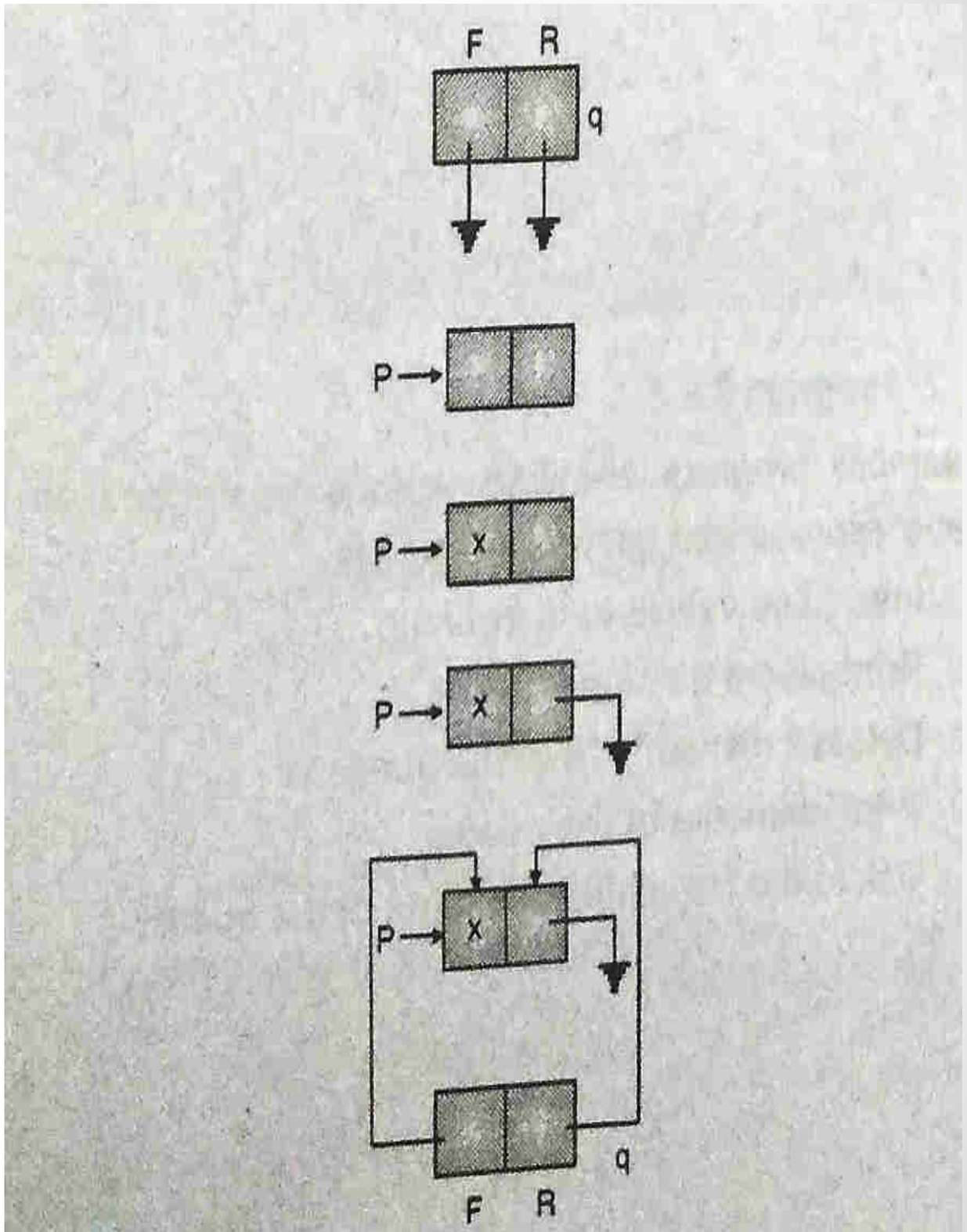
enqueue( )

enqueue function inserts a value (say x) at the rear end of the queue.

☞ **Steps required for insertion of an element x in a queue**

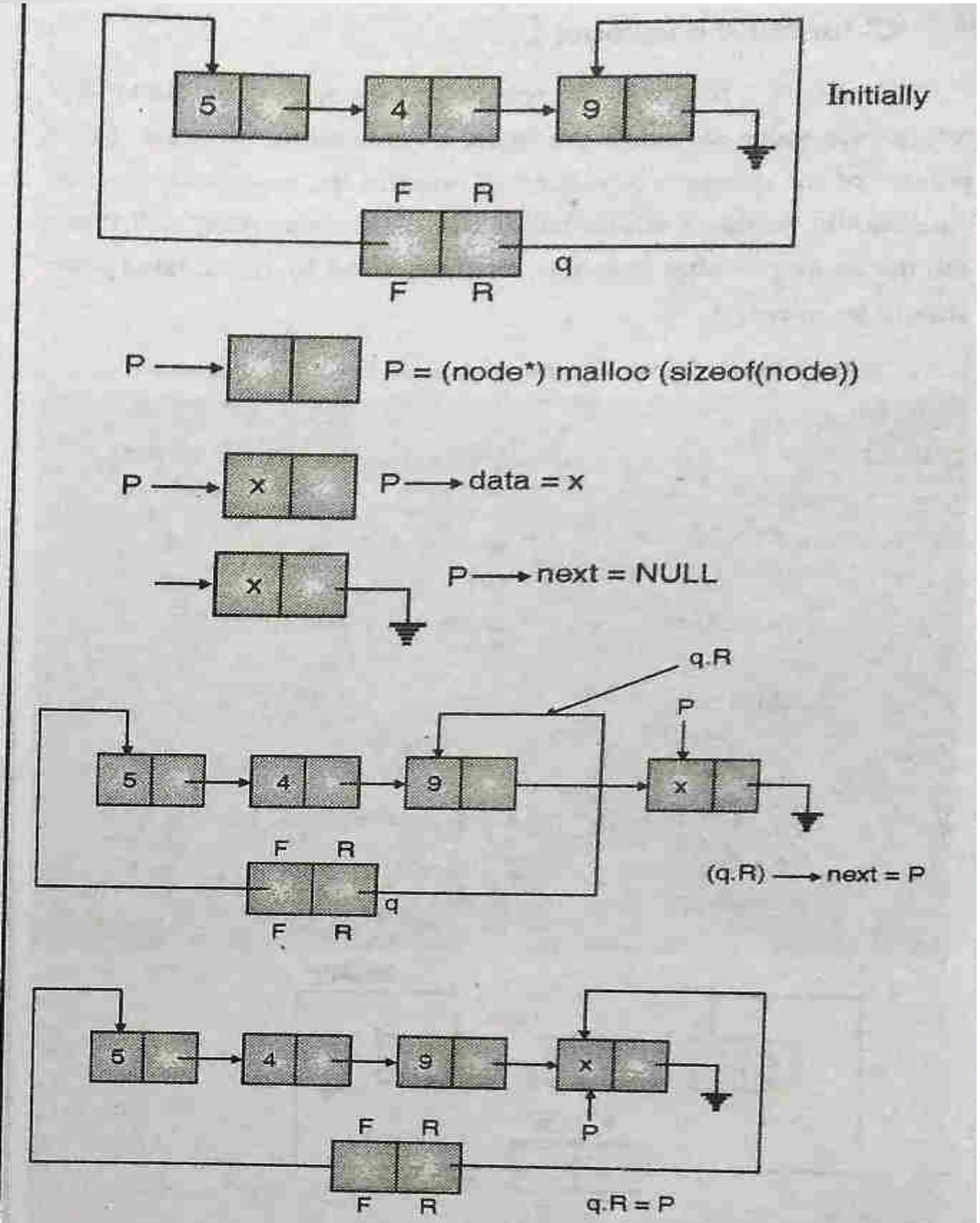
- Memory is acquired for the new node.
- Value x is stored in the new node.
- New node is inserted at the rear end.
- Special care should be taken for insertion into an empty queue. Both rear and front pointers will point to the only element of the queue.

```
Node *P; Q q;
P =(node *) malloc(sizeof(node));          /* acquire memory */
P → data = x;
            /* store x in the node */
P → next = NULL;
if(empty(&q))          /* inserting element in an empty queue */
{
                        q.R = q.F = P;
}
else
{
                        (q.R) → next = P;
                        q.R = P;
}
```



(dsa11.15)**Fig. 6.2.7 : Insertion in an empty queue (stepwise)**

Operation on queue implemented using Linked list



(dsa11.16)**Fig. 6.2.8 : Insertion into a non-empty queue (stepwise)**

☞ **'C' function for insertion.**

```
void insert(Q *qP, int x)
{
        node *P;
        P =(node *) malloc(sizeof(node));
        P → data = x;
        P → next = NULL;
        if(empty(qP))
        {
                qP → R = qP → F = P;
        }
        else
        {
                qP → R → next = P;
                qP → R = p;
        }
}
```

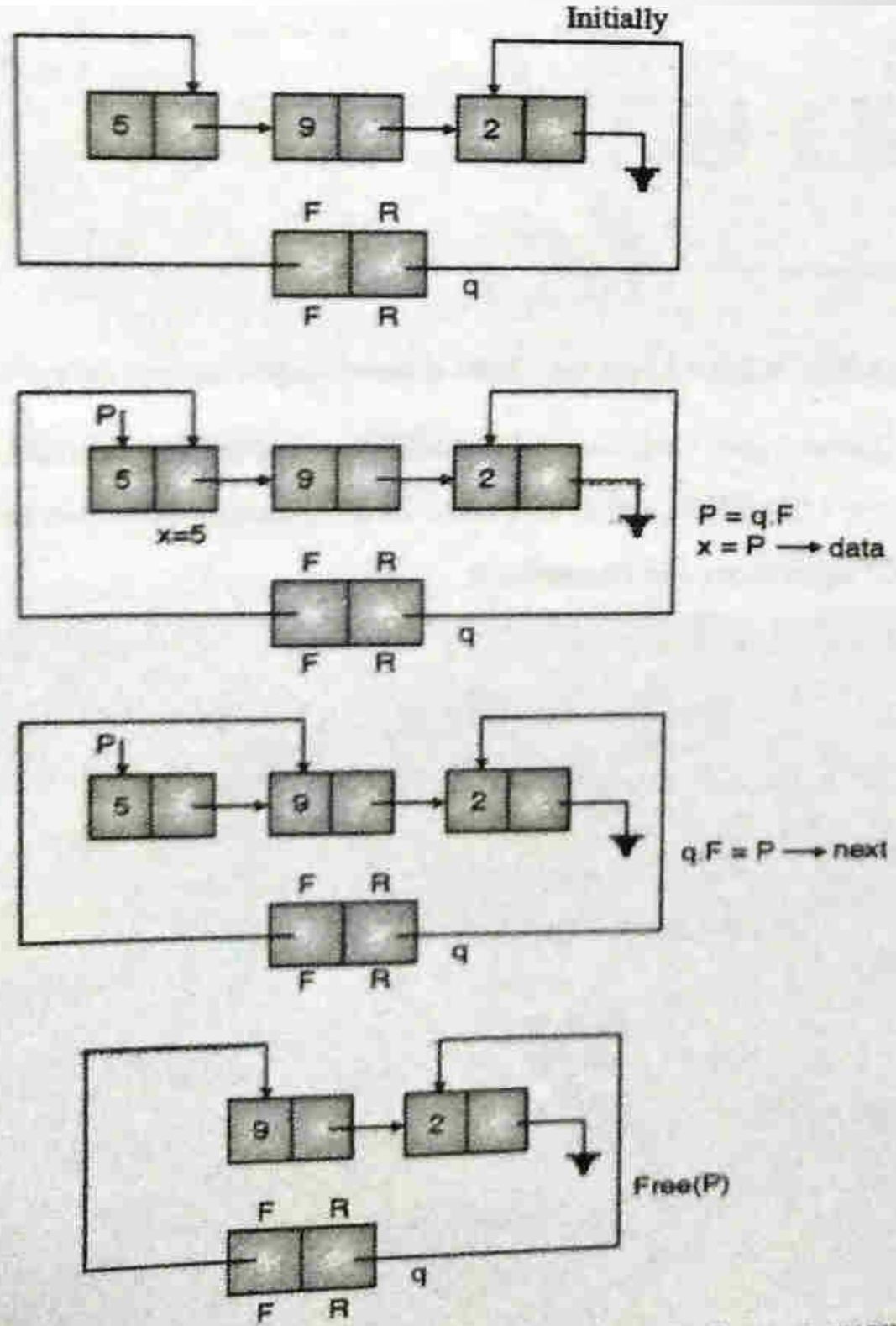
Operation on queue implemented using Linked list

'C' function dequeue( ).

dequeue( ) function deletes the front node of the queue and returns the value stored in the node, to the calling program. Front pointer of the queue is advanced to point to the next node. Special care should be taken while deleting the last node. As it will make the queue empty after deletion. Memory used by the deleted node should be released.

Steps required for deletion of a node from the queue :

```
Node *P; Q q;
P = q.F;              /* store the address of the front node in P */
x = P → data
if(q.R == q.F)                /* deleting the last node */
{
        initialize(&q);       /* make the queue empty */
        free(P);
        return(x);
}
else
{       q.F = P → next;       /* Advance the front pointer */
        free(P);
        return(x);
}
```



(dss11.17)**Fig. 6.2.9 : Deletion of the front node from a queue (stepwise)**

Operation on queue implemented using Linked list

☞ 'C' function for deletion.

```
int dequeue(Q *qP)
{
        int x;
        node *P;
        P = qP → F;
        x = P → data;
        if(qP → R == qP → F)        /* last element */
                initialize(qP);
        else
                qP → F = P → next;
        free(P);
        return(x);
}
```

☞ 'C' function for printing queue.

Elements of the queue can be printing by traversing underlying linked list starting from the front node.

```
void print(Q *qP)
{
        node *P;
        P = qP → F; /* start from front */
        while(P! = NULL)
        {
                printf("\n%d", P -> data);
                P = P → next;
        }
}
```

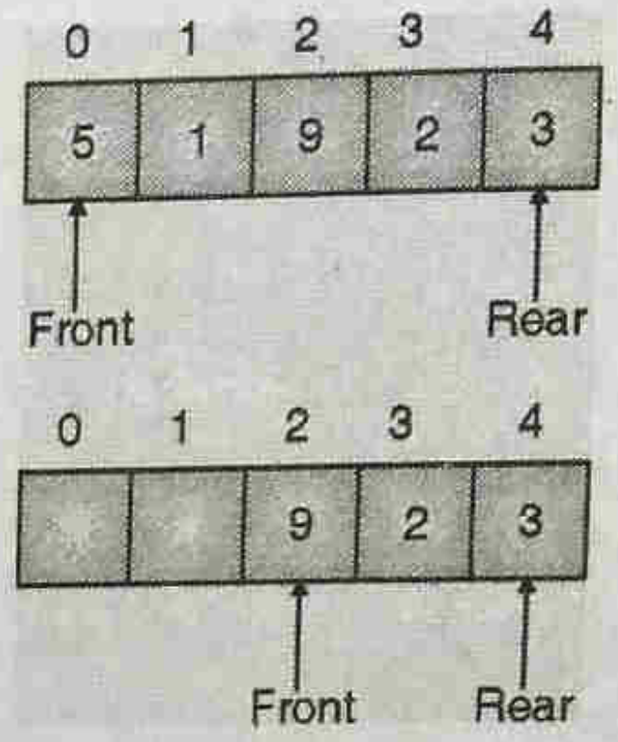
Circular Queue

– After insertion of five elements in the array (a queue) a shown in Fig. 6.3.1,

rear = 4
front = 0
} queue is full

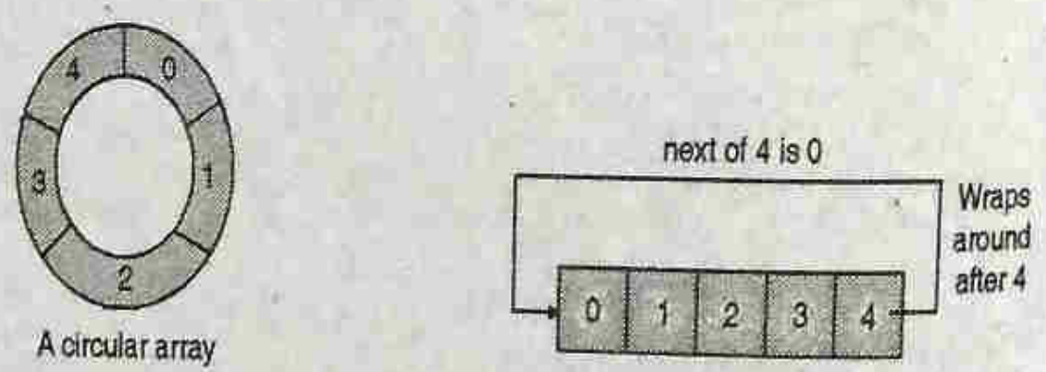
– After two successive deletions

rear = 4
front = 2
} queue is full



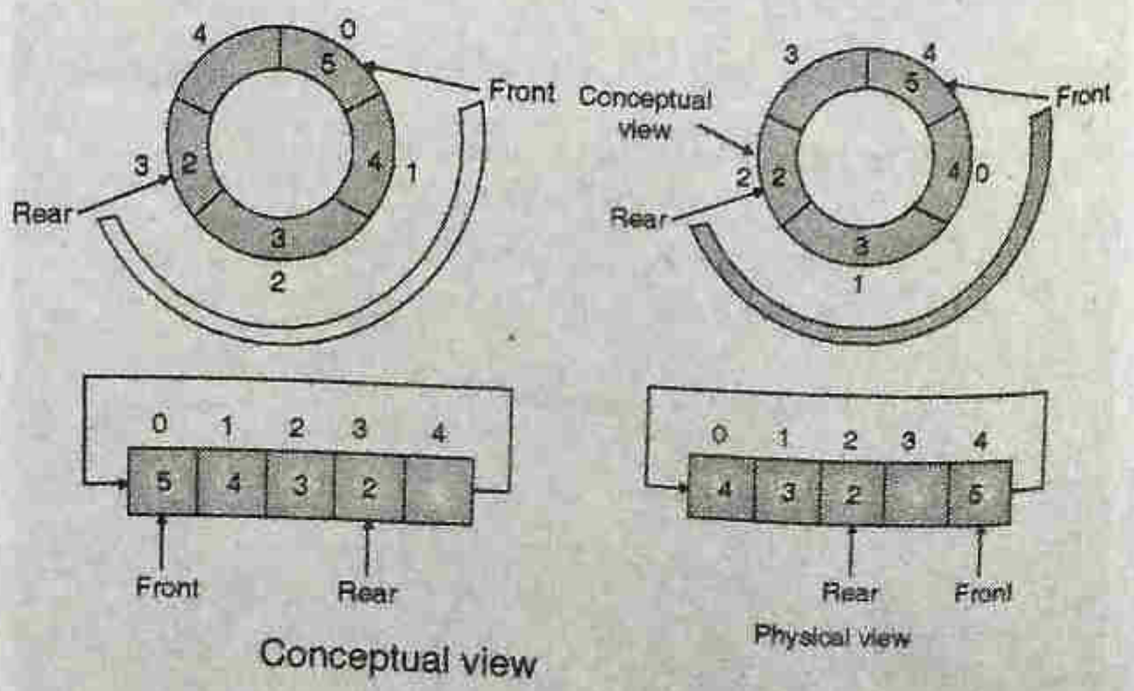
(dsa11.18)**Fig. 6.3.1 : Queue is full, although locations 0 and 1 are vacant queue in the Fig. 6.3.1 is full as there is no empty space ahead of rear.**

The simple solution is that whenever rear gets to the end of the array, it is wrapped around to the beginning. Now, the array can be thought of as a circle. The first position follows the last element.



(dsa11.19)**Fig. 6.3.2 : A circular array**

In a circular array, the queue is found somewhere around the circle in consecutive positions.

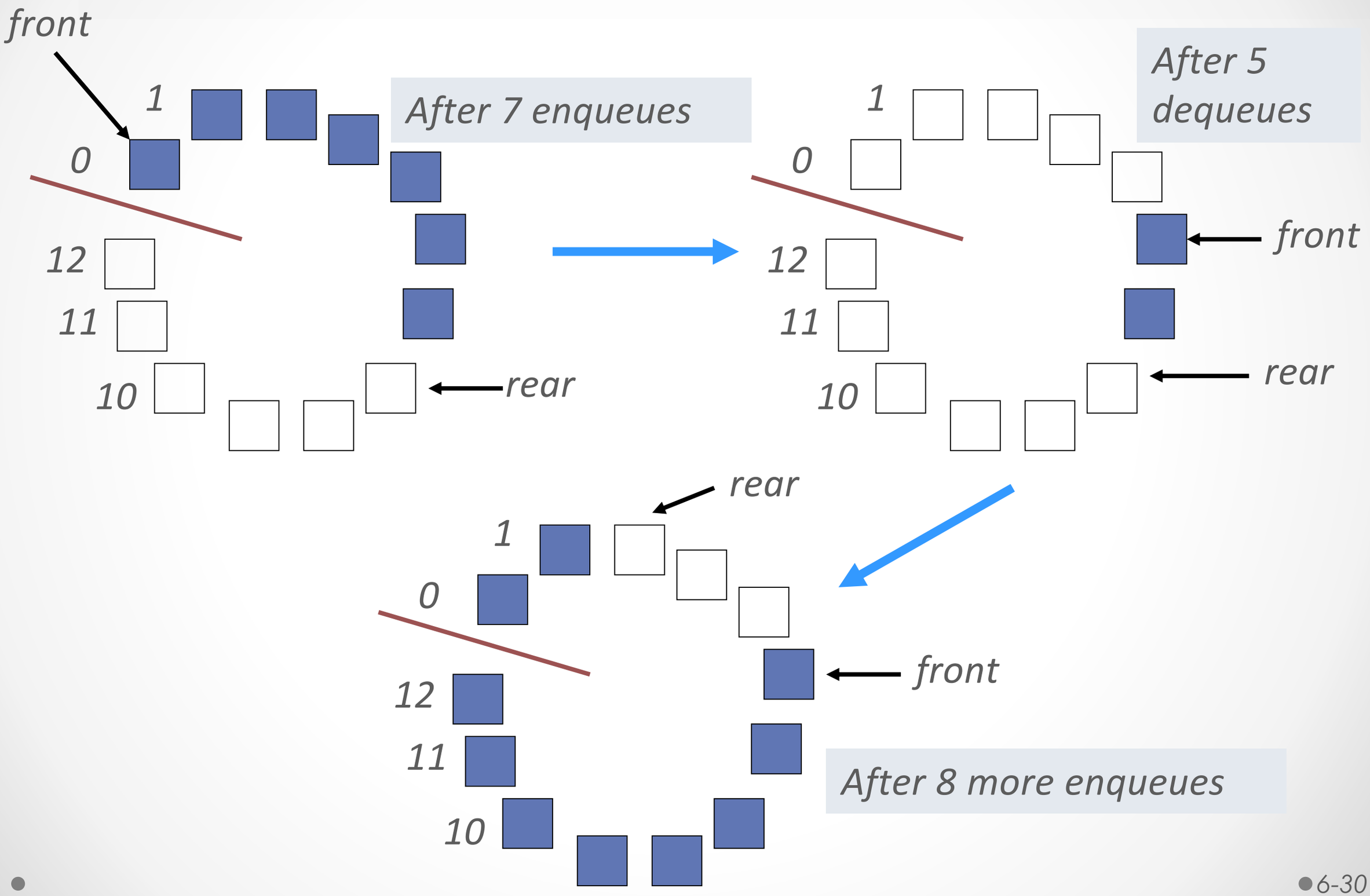


(dsa11.20)**Fig. 6.3.3 : Queue at various places in an array**

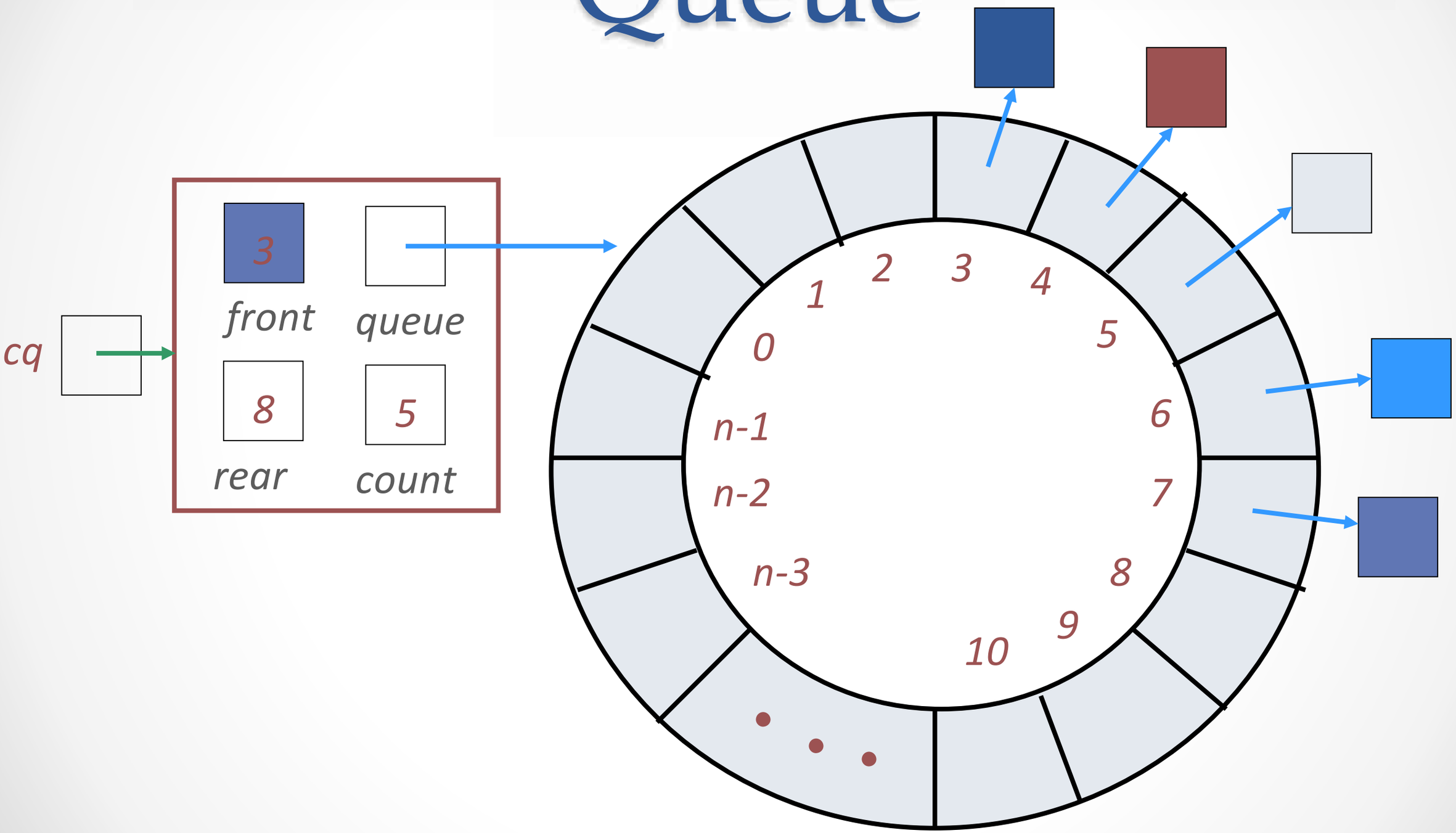
Second Approach: Queue as a Circular Array

- If we don't fix one end of the queue at index 0, we won't have to shift elements
- ***Circular array*** is an array that conceptually loops around on itself
  - The last index is thought to "***precede***" index 0
  - In an array whose last index is **n**, the location "***before***" index **0** is index **n**; the location "***after***" index **n** is index **0**
- Need to keep track of where the ***front*** as well as the ***rear*** of the queue are at any given time

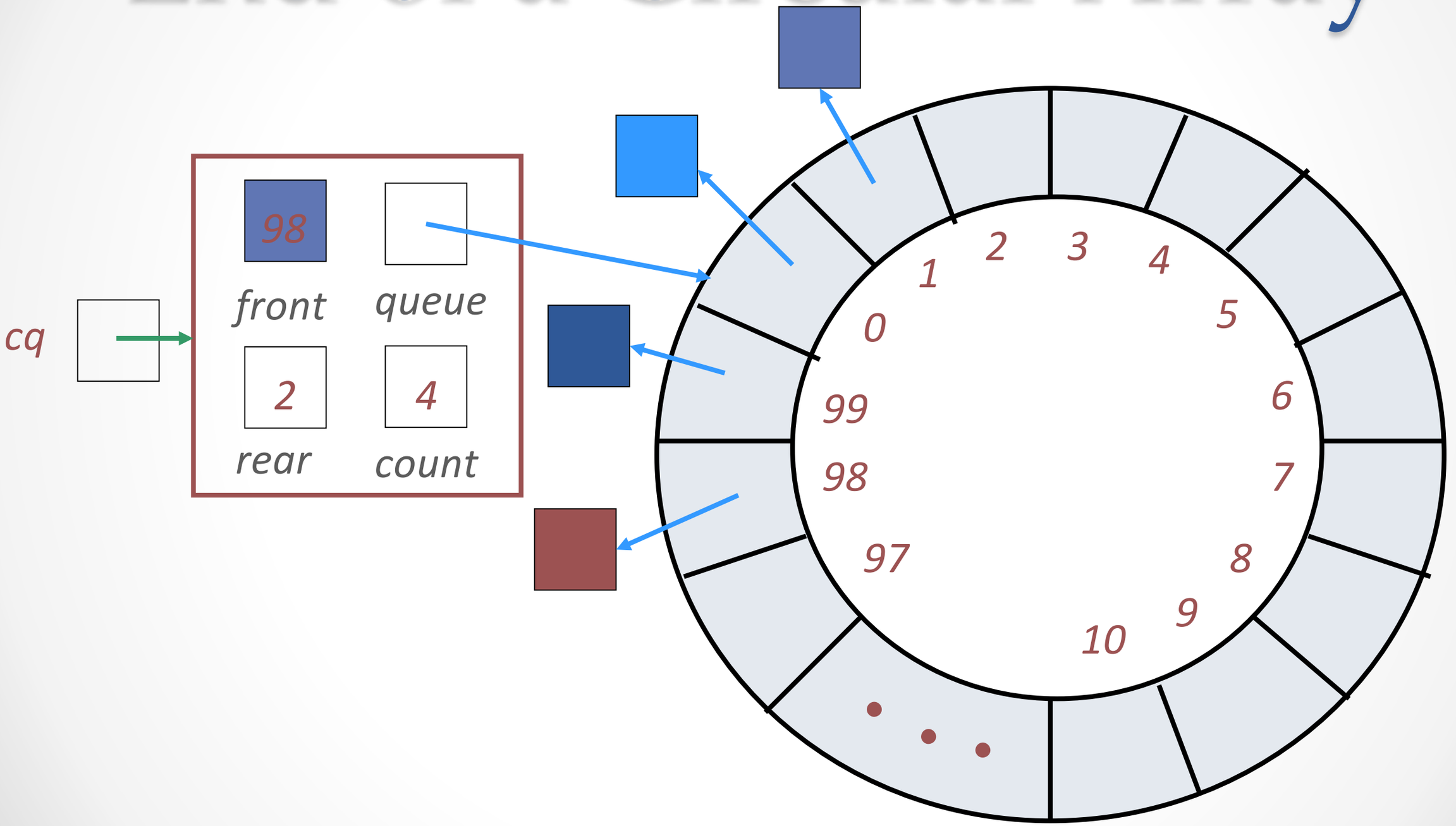
Conceptual Example of a Circular Queue



Implementation of a Queue

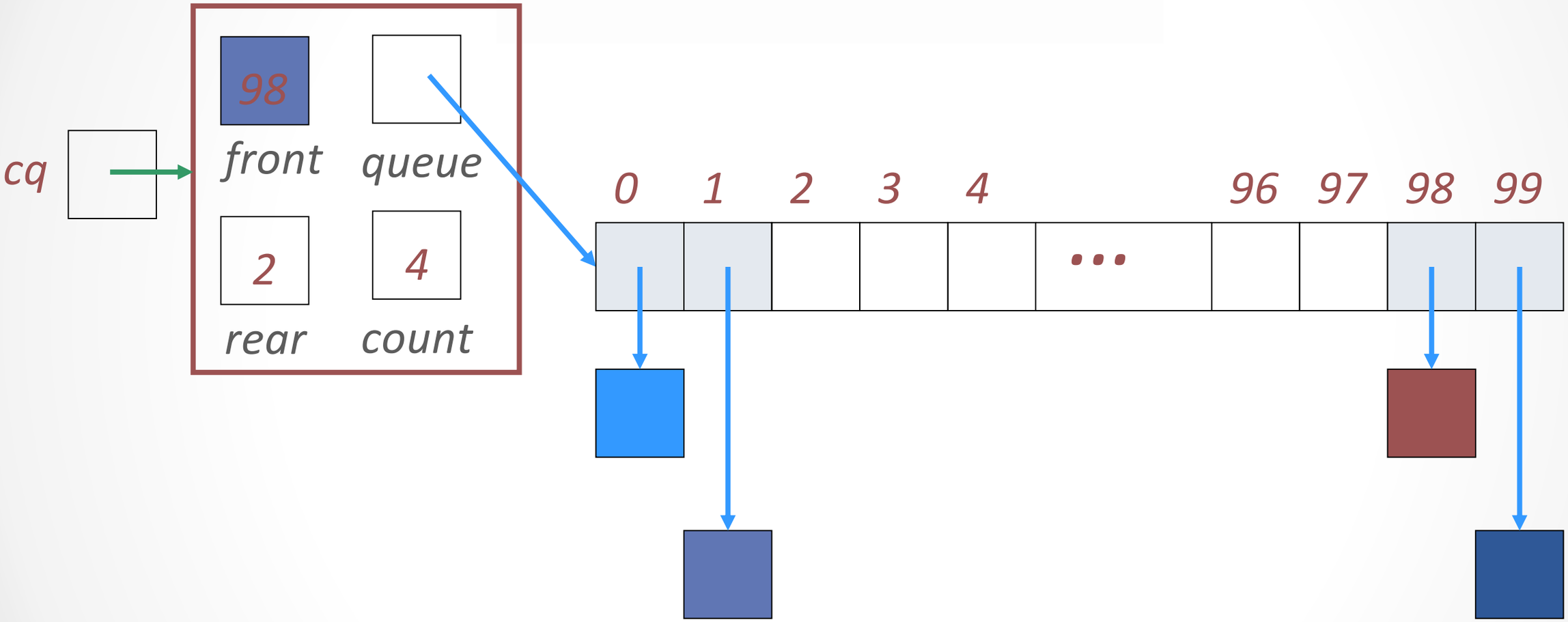


A Queue Straddling the End of a Circular Array



Circular Queue Drawn Linearly

Queue from previous slide



Circular Array Implementation

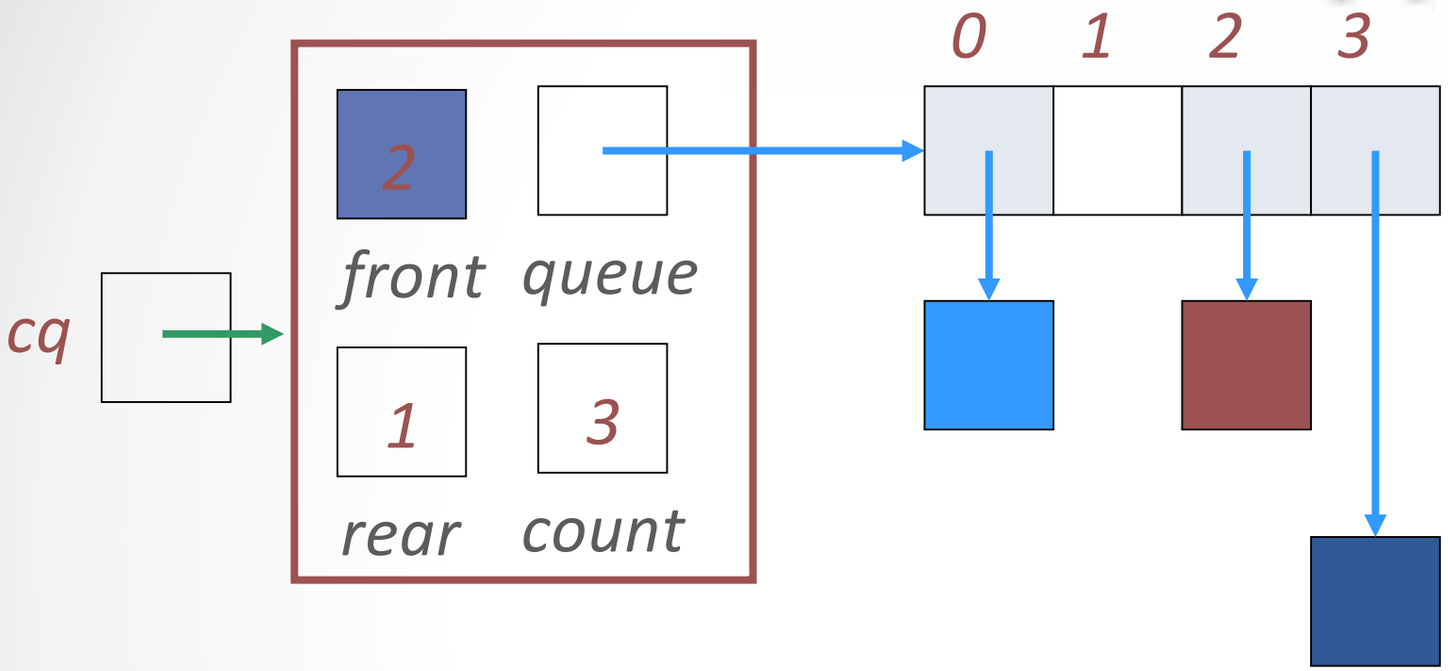
- When an element is enqueued, the value of **rear** is incremented
- But it must take into account the need to loop back to index 0:

rear = (rear+1) % queue.length;

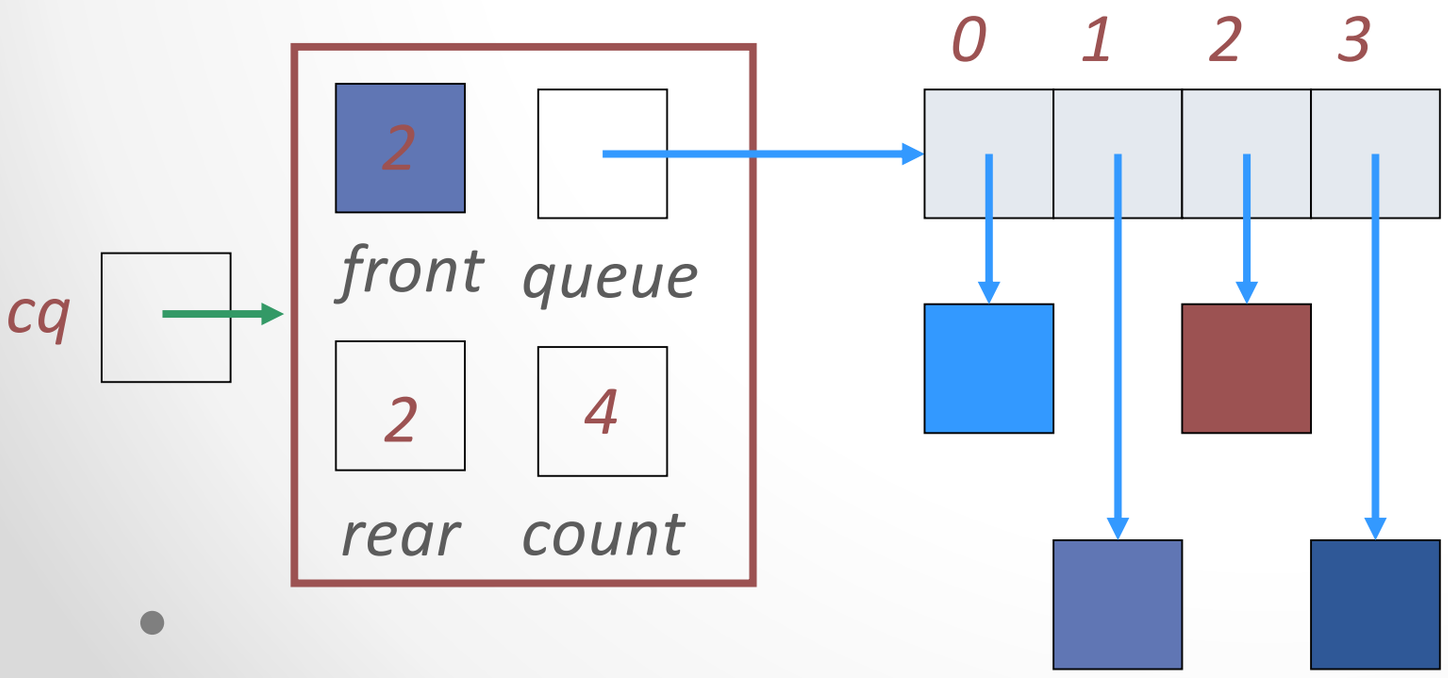
- Can this array implementation also reach capacity?

Example: array of length 4

What happens?

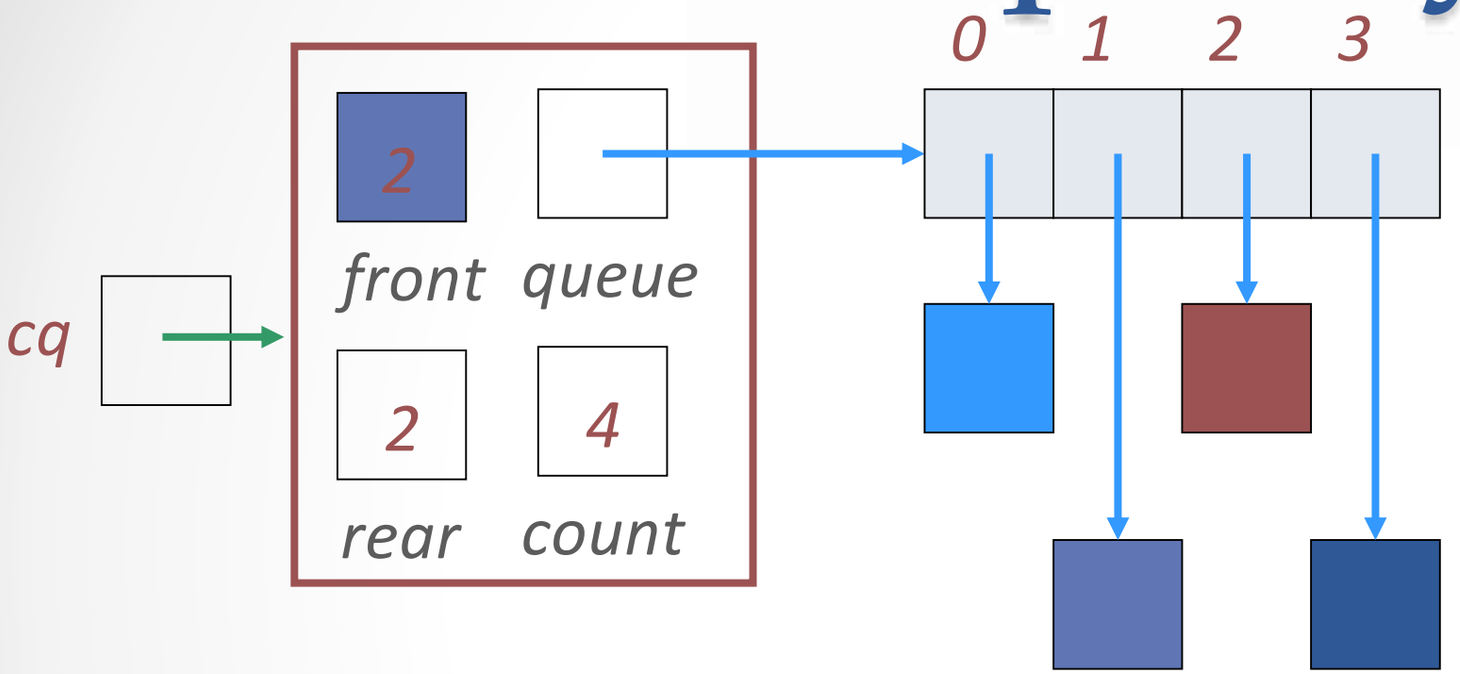


Suppose we try to add one more item to a queue implemented by an array of length 4

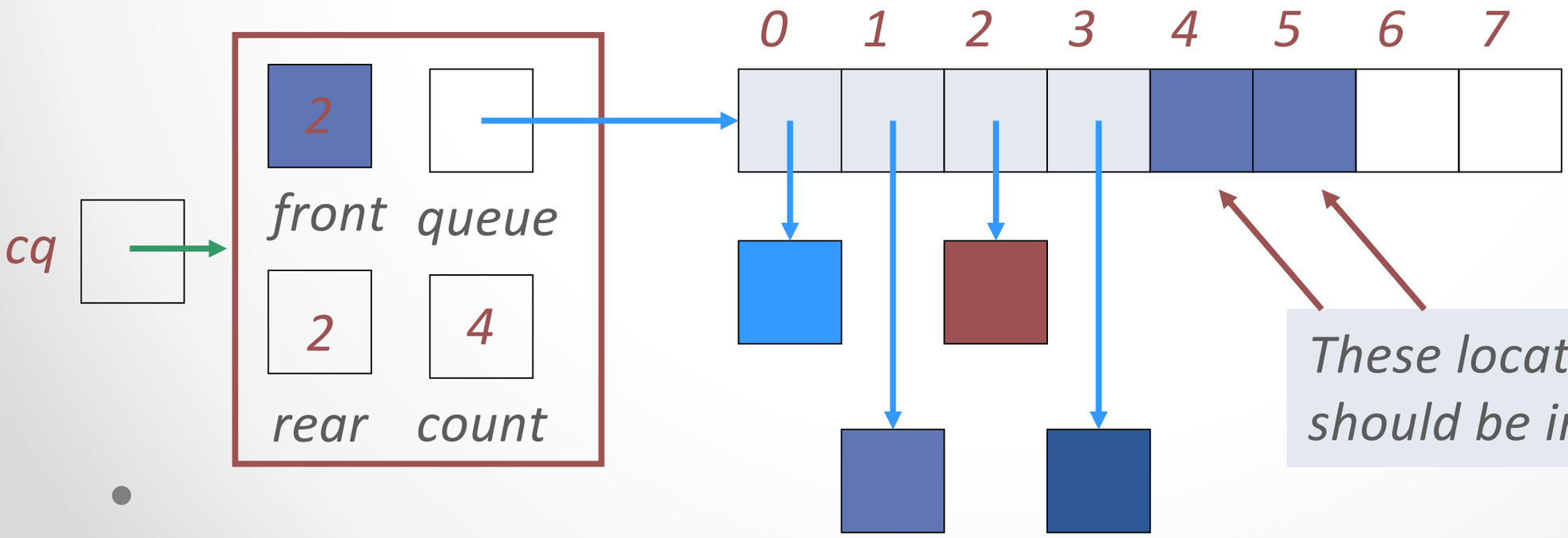


The queue is now full. How can you tell?

Need to expand capacity…

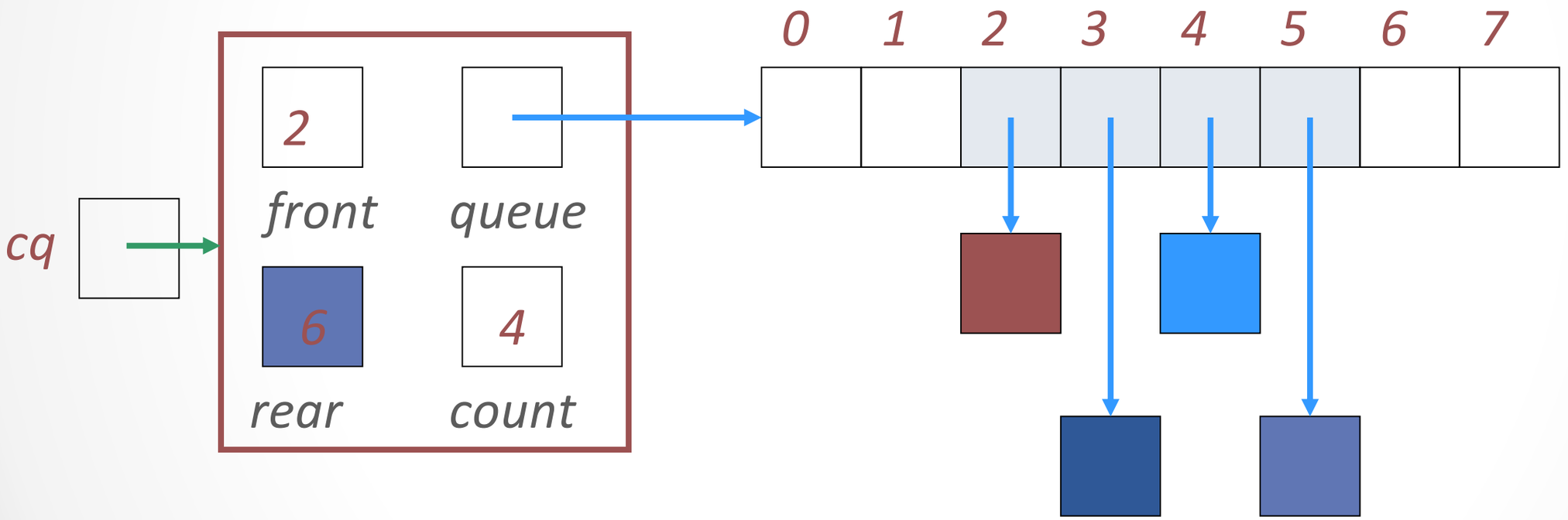


We can't just double the size of the array and copy values to the same positions as before: circular properties of the queue will be lost

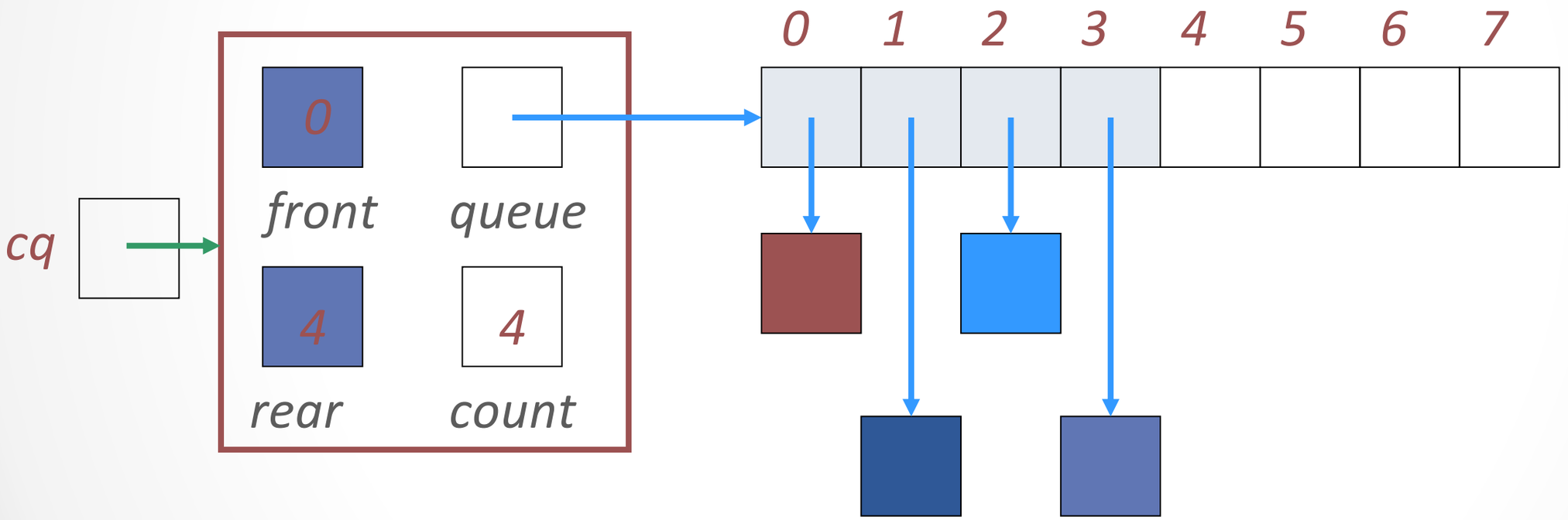


These locations should be in use

We could build the new array, and copy the queue elements into contiguous locations beginning at location front:

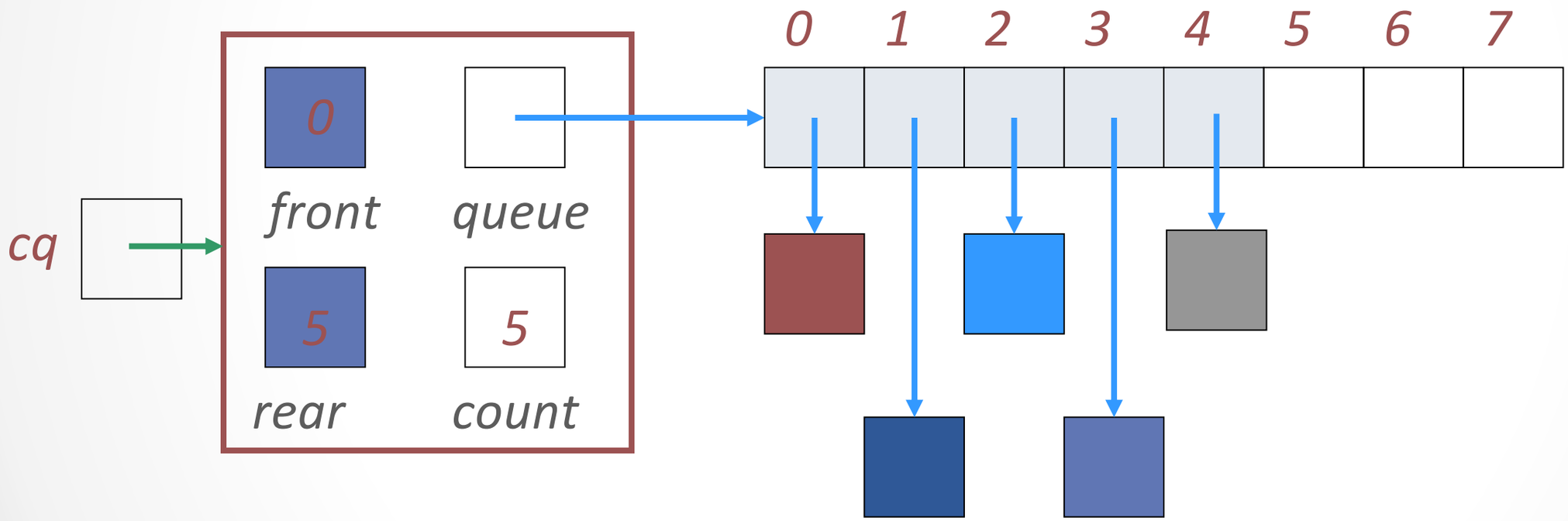


Better: copy the queue elements in order to the beginning of the new array



New element is added at rear = (rear+1) % queue.length

See expandCapacity() in CircularArrayQueue.java



Circular Queue

Consider a circular queue of size 5 having initial status as :

Front	Rear	Circular queue				
		0	1	2	3	4
1	2		P	Q		

Show the value of front, rear and the contents of circular queue after every step in tabular form for the following operations : (1) R is added (2) Delete 2 letters (3) S, T, U are added (4) Three letters are deleted (5) V is added

Solution :

Operation	Front	Rear	Circular queue
Initial	1	2	0 1 2 3 4 | | P | Q | | |
R is added	1	3	0 1 2 3 4 | | P | Q | R | |
Delete 2 letters	3	3	0 1 2 3 4 | | | | R | |
S,T,U are added	3	1	0 1 2 3 4 | T | U | | R | S |
Three letters are deleted	1	1	0 1 2 3 4 | | U | | | |
V is added	1	2	0 1 2 3 4 | | U | V | | |

Circular Queue

to be maintained. In contrast, when a queue is maintained using a linked list, two pointers must be maintained :

(a) address of the front node (for deletion)

(b) address of the rear node (for insertion)

Special care should be taken, while inserting an element in an empty queue. Similarly, after deletion of the last element, the queue should become empty.

Steps for insertion of element x in queue represented using a circular linked list :

(a) Acquire memory for the node

P = (node *) malloc(sizeof(node))

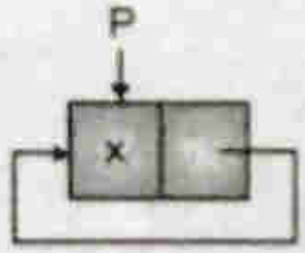
(b) Store the data x in the node

P → data = x;

If the queue is empty

(c) newly acquired node should be connected back to itself.

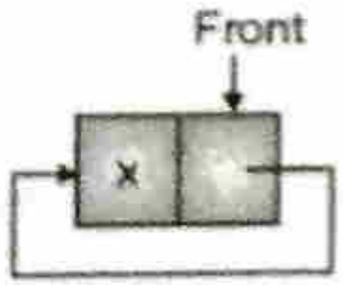
P → next = P;



(dsa11.23)**Fig. 6.3.6**

(d) front should point to the only element

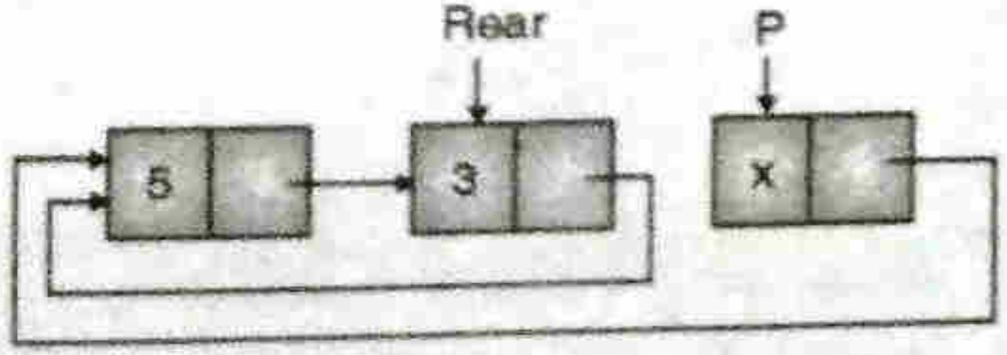
front = P



(dsa11.24)**Fig. 6.3.7**

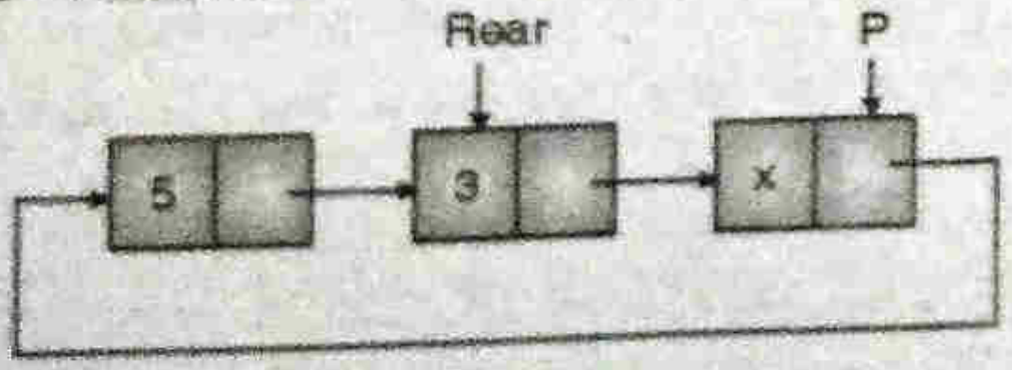
If the queue is not empty

(c) P → next = rear → next



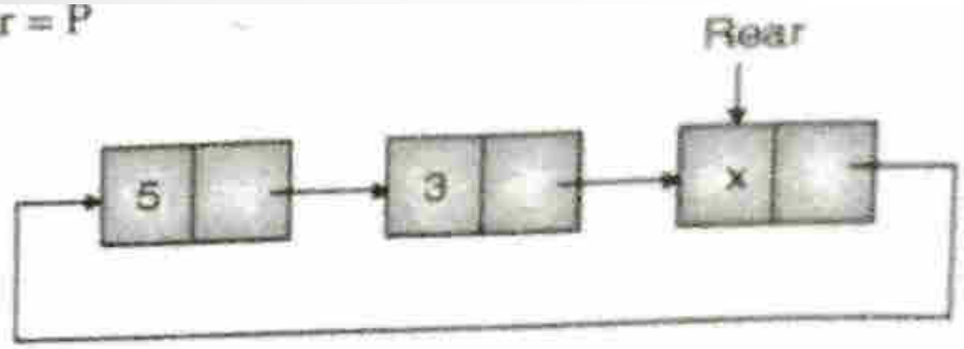
(dsa11.25)**Fig. 6.3.8**

(d) rear → next = P



Circular Queue

(e) rear = P



(dsa11.27)Fig. 6.3.10

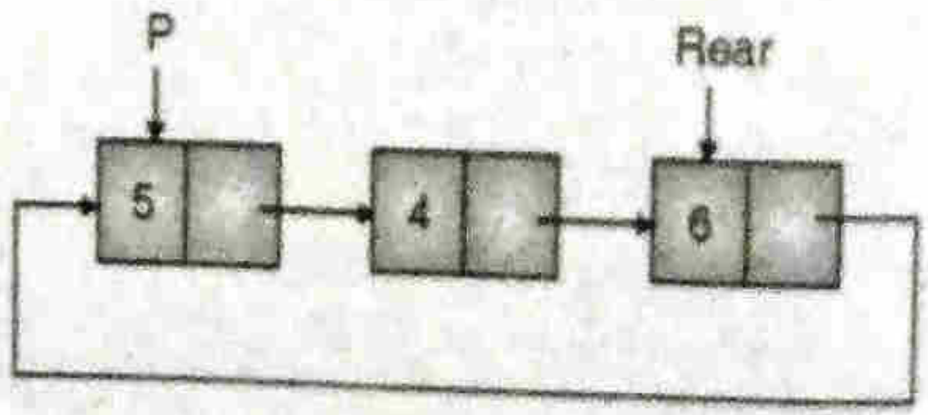
☞ **'C' function for insertion of an element in a queue represented using a circular linked list.**

```
void enqueue(node **R, int x)  //queue is referenced by address of
rear node.
{
        node *P;
        P =(node *) malloc(sizeof(node));
        P → data = x;
        if(*R == NULL)
        {
                P → next = P;*R = P;
        }
else
{       P → next = (*R) → next;
        (*R) → next = P;
        *R = P;
}
}
```

Steps for deletion of an element from a queue represented using a circular linked list.

(a) P = rear → next;

P Point to the node to be deleted

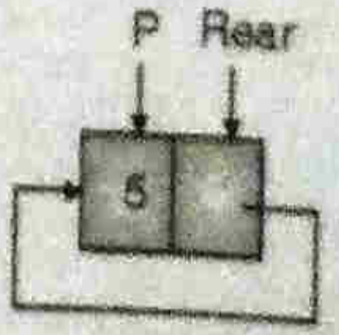


(dsa11.28)Fig. 6.3.11

(b) x = P → data;

if (last node is being deleted)

{



dsa11.29)Fig. 6.3.12

Circular Queue

(c) release the memory of the node being deleted.

```
free(P);
```

(d) Make the queue empty

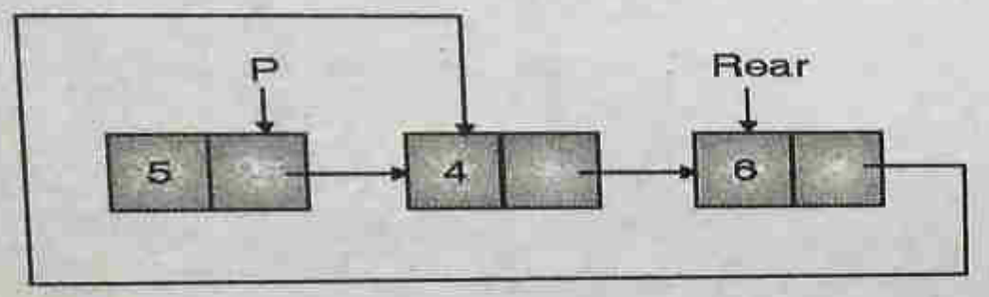
```
rear = NULL;
```

(e) Return the value stored in the front node

```
return(x);
}
else
{
```

(c) Remove the front node from the queue

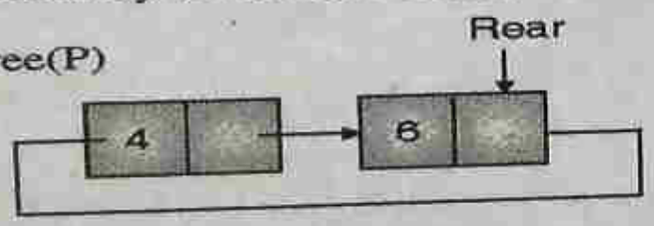
```
rear → next = P → next
```



(dsa11.30)**Fig. 6.3.13**

(d) release the memory of the node being deleted

```
free(P)
```



(dsa11.31)**Fig. 6.3.14**

(e) return the value stored in the front node

```
return(x)
}
```

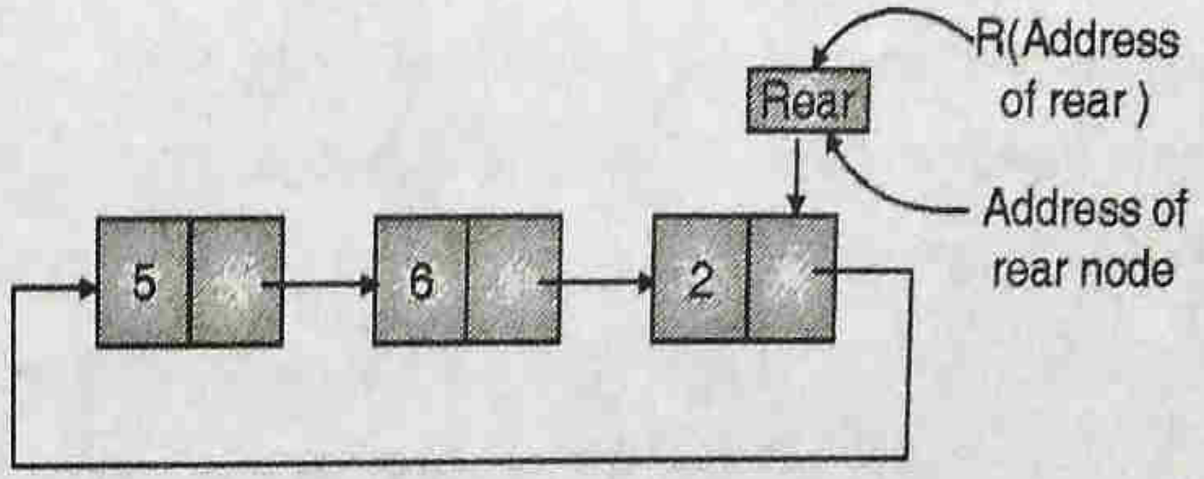
```
}
```

☞ **'C' function for deletion of the front node from a queue represented using a circular linked list.**

```
int dequeue(node **R)          //pointer rear is passed by address
{
        node *P;
int x;
        P =(*R) → next;   //'*R' is same as 'rear' as R
                                contains the address of rear.
        x = P → data;
        if(P → next == P) /* deleting the last node */
        {
                *R = NULL;
```

Circular Queue

```
            free(P);
            return(x);
        }
        (*R) → next = P → next;
        free(P);
        return(x);
}
```



(dsa11.32)**Fig. 6.3.15**

In the above function, rear is passed by address as after deletion, rear may change. Receiving variable in function int dequeue(node **R) is declared as node **R. R contains the address of rear and hence, *R can be used in place of 'rear'.

☞ **'C' function for printing elements of a queue represented using a circular linked list.**

```
void Print(node * rear)
{
        node P;
        P = rear → next;   /* start printing from the front */
        do
        {
                printf("\n%d", P → data);
                P = P → next;
        } while(P! = rear → next);
}
```

In case of a circular linked list, the starting case and the termination case for the loop used for traversal of the linked list are identical.

(1) We start printing from the front node.

(2) We terminate printing on reaching the front node.

Such cases are best handled through do-while loops.

DeQueue

The word **dequeue** is a short form of double ended queue. It is general representation of both stack and queue and can be used as stack and queue. In a dequeue, insertion as well deletion can be carried out either at the rear end or the front end. In practice, it becomes necessary to fix the type of operation to be performed on front and rear end. Dequeue can be classified into two types :

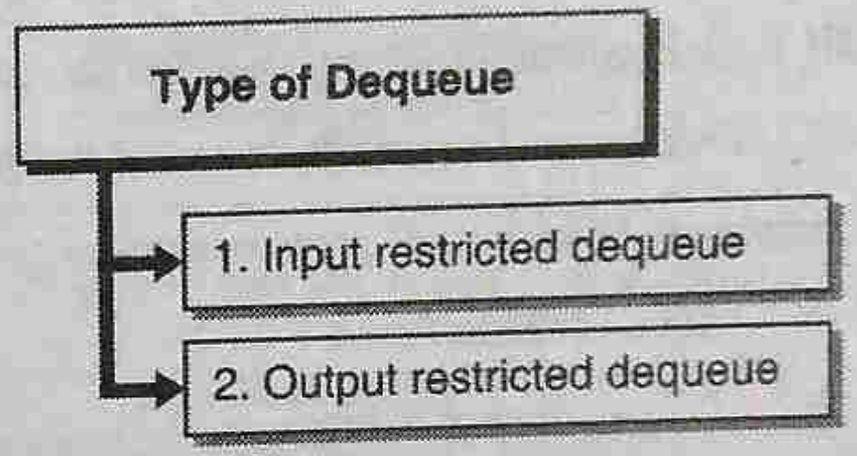


Fig. C6.1 : Type of Dequeue

➜ 1. Input restricted dequeue

The following operations are possible in an input restricted dequeue :

(i) Insertion of an element at the rear end

(ii) Deletion of an element from front end

(iii) Deletion of an element from rear end

➜ 2. Output restricted dequeue

The following operations are possible in an output restricted dequeue.

(i) Deletion of an element from front end

(ii) Insertion of an element at the rear end

(iii) Insertion of an element at the front end

There are various methods to implement a dequeue.

(a) Using a circular array

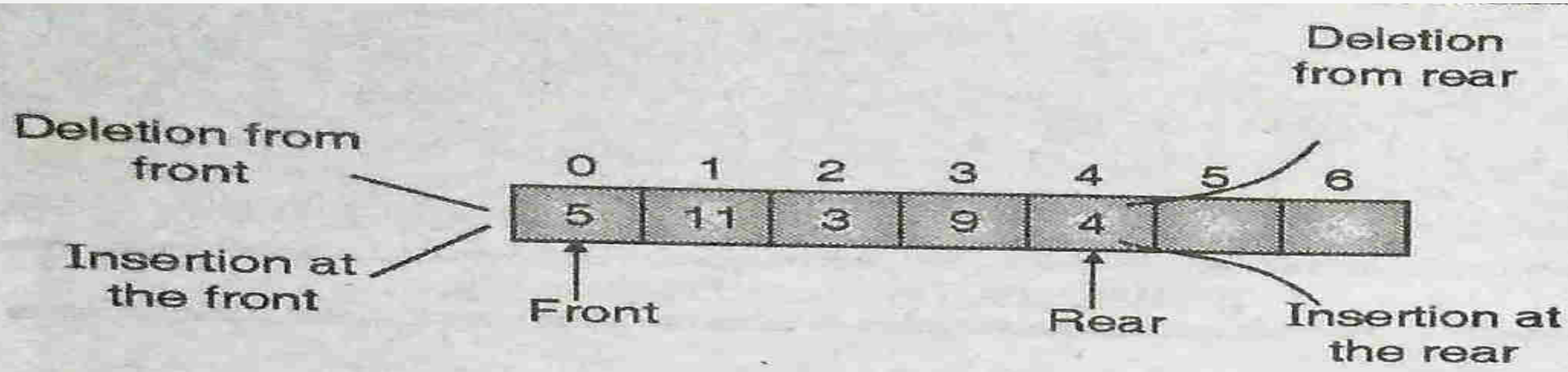
(b) Using a singly linked list.

(c) Using a singly circular linked list.

(d) Using a doubly linked list.

(e) Using a doubly circular linked list.

DeQueue



(dsa11.33)**Fig. 6.4.1 : A dequeue in a circular array**

Operations associated with dequeue :

(a) empty( ) : Whether the queue is empty ?

(b) full( ) : Whether the queue is full ?

(c) initialize( ) : Make the queue empty

(d) enqueueR( ) : Add item at the rear end of the queue.

(e) enqueueF( ) : Add item at the front end of the queue.

(f) dequeueR( ) : Delete item from the rear end of the queue.

(g) dequeueF( ) : Delete item from the front end of the queue.

Timing complexity of various dequeue operations :

enqueue R( ) – 0(1) – constant time.

enqueue F( ) – 0(1) – constant time.

dequeue R( ) – 0(1) – constant time.

dequeue F( ) – 0(1) – constant time.

Advantage of dequeue : The dequeue is a general representation of both stack and queue it can be used both as stack or a queue.

DeQueue as an ADT

```
#define MAX 30 /* A queue with maximum of 30 elements */

typedef struct DQ

{

        int data [MAX];

        int rear, front;

} DQ;
```

☞ **Operations on a dequeue**

i) initialize( ) : Make the queue empty.

ii) empty( ) : Determine if queue is empty.

iii) full( ) : Determine if queue is full.

iv) enqueueF( ) : Insert an element at the front end of the queue.

v) enqueueR( ) : Insert an element at the rear end of the queue.

vi) dequeueR( ) : Delete the rear element.

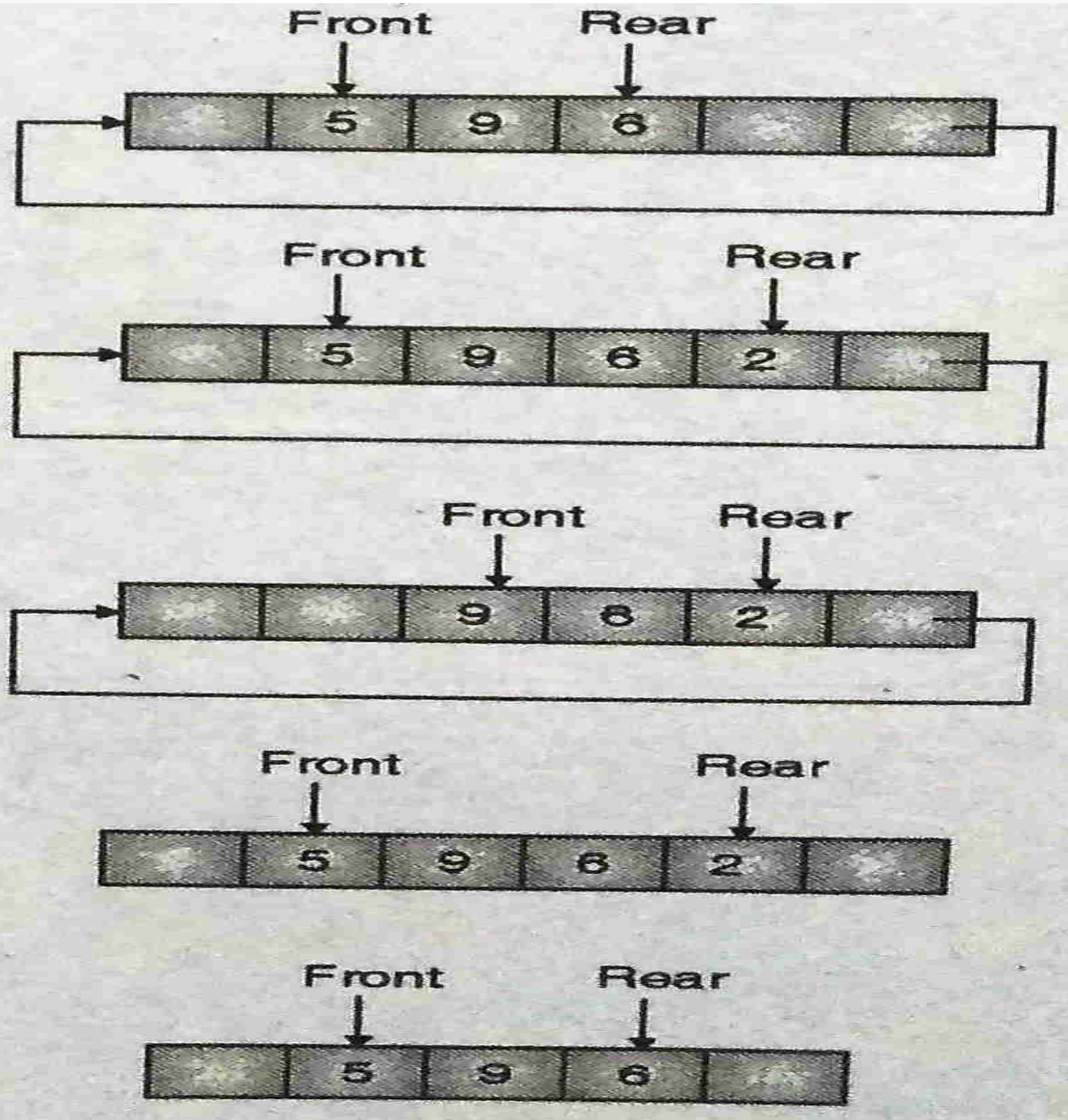
vii) dequeueF( ) : Delete the front element.

viii) print( ) : Print elements of the queue.

☞ **Prototype of functions used for various operations on queue**

- void intialize (DQ *p);
- int empty (DQ *p);
- function returns 1 or 0, depending on whether the queue pointed by p is empty or not.
- int full (DQ *p);
- function returns 1 or 0, depending on whether the queue pointed by p is full or not.
- void enqueueF (DQ *p, int x);
- void enqueueR (DQ *p, int x);
- int deleteR (DQ *p);
- int deleteF (DQ *p);
- void print (DQ *p);
- enqueueR( ) and enqueueF( ) will cause an overflow if the queue is full.
- dequeueR( ) and dequeueF( ) will cause an underflow if the queue is empty.

Implementation of DeQueue using linked list



(dss11.34)**Fig. 6.4.2 : Various operations on dequeue and movement of front and rear in clock-wise or anti-clockwise direction**

C function for deque using circular array

```
typedef struct dequeue
{
        int data[MAX];
        int front, rear;
} dequeue;
```

(1) Initialize the queue by setting values of rear and front as – 1.

```
void initialize(dequeue *P)
{
        P → rear = –1;
        P → front = –1;
}
```

(2) Test, whether the dequeue is empty ?

```
int empty(dequeue *P)
{
    if(P → rear == –1)
        return(1);
    return(0);
}
```

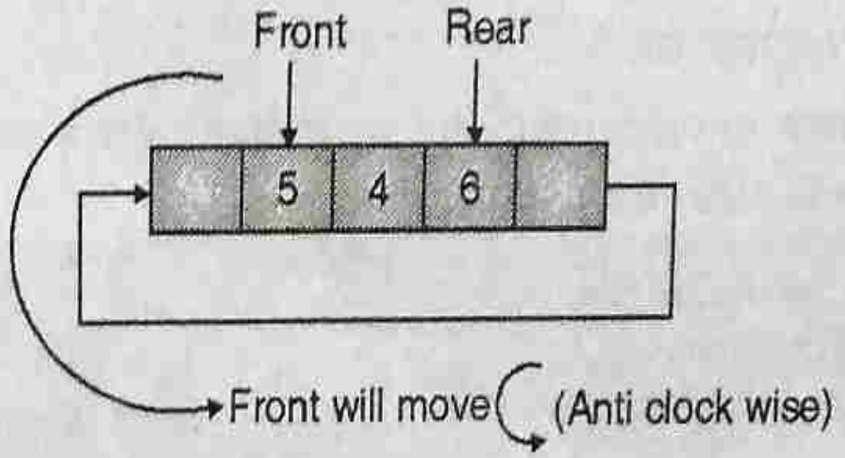
(3) Test, whether the dequeue is full ?

```
int full(dequeue *P)
{
    if((P → rear + 1) % MAX ==P → front)
        return(1);
    return(0);
}
```

(4) Add item at the rear end of the dequeue.

```
void enqueueR(dequeue *P, int x)
{
    if(empty(P))
    {
        P → rear = 0;
        P → front = 0;
        P → data[0] = x;
    }
    else
    {
        P → rear =(P → rear + 1) % MAX;
        P → data[p → rear] = x;
    }
}
```

(5) Add item at the front end of the dequeue on addition of item at the front end, front will move in anti-clockwise direction.



(dsa11.35)**Fig. 6.4.3**

C function for deque using circular array

```
void enqueueF(dequeue *P, int x)
{
    if(empty(P))
    {
        P → rear = 0;
        P → front = 0;
        P → data[0] = x;
    }
    else
    {
P → front = (P → front – 1 + MAX) % MAX;
        P → data[P → front] = x;
    }
}
```

6) Delete an item from the front end of the dequeue :

```
int dequeueF(dequeue *P)
{
    int x;
    x = P → data[P → front];
    if(P → rear == P → front)    /* delete the last element */
        initialize(P);
    else
        P → front = (P → front + 1) % MAX;
    return(x);
}
```

) Delete an item from the rear end of the dequeue :

```
nt dequeueR(dequeue *P)
{
    int x;
    x = P → data[P → rear];
    if(P → rear == P → front)
        initialize(P);
    else
    P → rear = (P → rear – 1 + MAX) % MAX;
    return(x);
}
```

Priority Queue

Priority queue is an ordered list of homogeneous elements. In a normal queue, service is provided on the basis of First-in-first-out. In a priority queue service is not provided on the basis of "first-come-first-served" but rather than each element has a priority based on the urgency of need.

- An element with higher priority is processed before other elements with lower priority.
- Elements with the same priority are processed on "first-come-first served" basis.

An example of priority queue is a hospital waiting room. A patient having a more fatal problem will be admitted before other patients. Other applications of priority queues is found in long term scheduling of jobs processed in a computer. In practice, short processes are given a priority over long processes as it improves the average response of the system.

Priority Queue as an ADT

```
# define MAX 30 /* A queue with maximum of 30 elements */
        typedef struct pqueue
        { int data [MAX];
         int front, rear;
        } pqueue;
```

☞ **Operations on a priority queue**

i) initialize( ) : Make the queue empty.

ii) empty( ) : Determine if the queue is empty.

iii) full( ) : Determine if the queue is full.

iv) enqueue( ) : Insert an element as per its priority.

v) dequeue( ) : Delete the front element (front element will have the highest priority)

vi) print( ) : Print elements of the queue.

☞ **Prototype of functions used for various operations on the queue**

- `void initialize(pqueue *p);`
- `int empty(pqueue *p);`
- `int full(pqueue *p);`
- `void enqueue(pqueue *p, int x);`

Priority Queue as an ADT

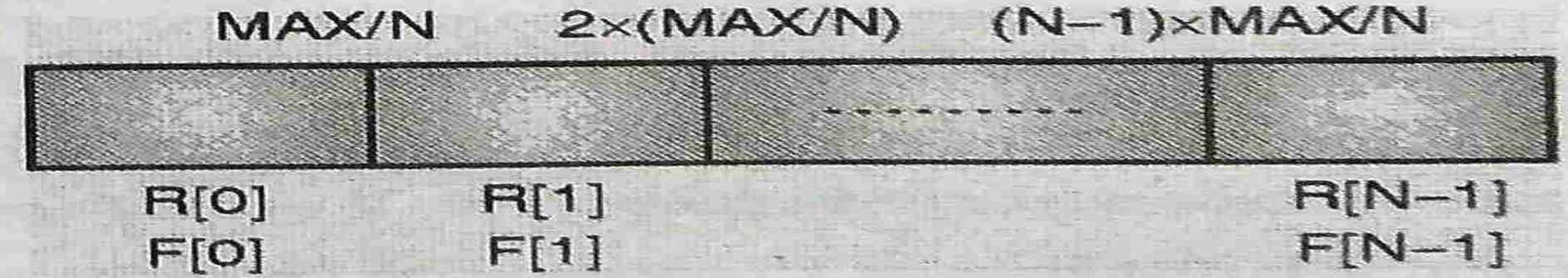
- int dequeue(pqueue *p);
- void print(pqueue *p);

- enqueue( ) operation will cause an overflow if the queue is full.
- Dequeue( ) operation will cause an underflow if the queue is empty.

Multiple Queue using array

Several queues can be stored in a single array. Suppose A[ ] is an array of size MAX and N number of queues are to be stored in the array. MAX is a predefined constant.

We can divide the available memory A [0.. MAX-1] into N segments. Each queue can be allocated one of these segments. For each Queue i, as shown in the Fig. 6.6.1, R[i] and F[i] represent the indices of rear and front of queue i.



(dsa11.36)**Fig. 6.6.1**

Initial values of R[i] and F[i] is calculated as given below.

For (i = 0; i < N; i++)

R[i] = F[i] = (MAX/N)*i;

Insert ( ) and Delete ( ) can be performed on any of the queues. Both insert ( ) and Delete ( ) have an additional parameter, queue number.

Application of Queue

Queue is a very useful data structure. Various features of operating system are implemented using a queue.

a) Scheduling of processes (Round Robin Algorithm)

b) Spooling (to maintain a queue of jobs to be printed)

c) A queue of client processes waiting to receive the service from the server process.

d) Various application software using non-linear data structure tree or graph requires a queue for breadth first traversal.

e) Simulation of a real life problem with the purpose of understanding its behaviour. The probable waiting time of a person at a railway reservation counter can be found through the technique of computer simulation if the following concerned factors are known :

1) Arrival rate

2) Service time

3) Number of service counters.

We will discuss some of the **applications** in detail :

1) Josephus problem

2) Job scheduling

3) Queue simulation