(A-B)*(D/E)
(A+B^D)/(E-F)+G
A*(B+D)/E-F*(G+H/K)
(A+B)*(C-D)$E*F
(A+B)*(C$(D-E)+F)/G)$(H-J)

[E] Transform the following infix expressions into their equivalen prefix expressions:

(A-B)*(D/E)
(A+B^D)/(E-F)+G
A*(B+D)/E-F*(G+H/K)

[F] Transform each of the following prefix expression to infix.

+A-BC
++A-*$BCD/+EF*GHI
+-$ABC*D**EFG

[G] Transform each of the following postfix expression to infix.

ABC+-
AB-C+DEF-+$
ABCDE-+$*EF*-

Representation Of A Queue As An Array

Queue, being a linear data structure can be represented in various ways such as arrays and linked lists. Representing a queue as an array would have the same problem that we discussed in case of stacks. An array is a data structure that can store a fixed number of elements. The size of an array should be fixed before using it. Queue, on the other hand keeps on changing as we remove elements from the front end or add new elements at the rear end. Declaring an array with a maximum size would solve this problem. The maximum size should be large enough for a queue to expand or shrink. Figure 8-2 shows the representation of a queue as an array.

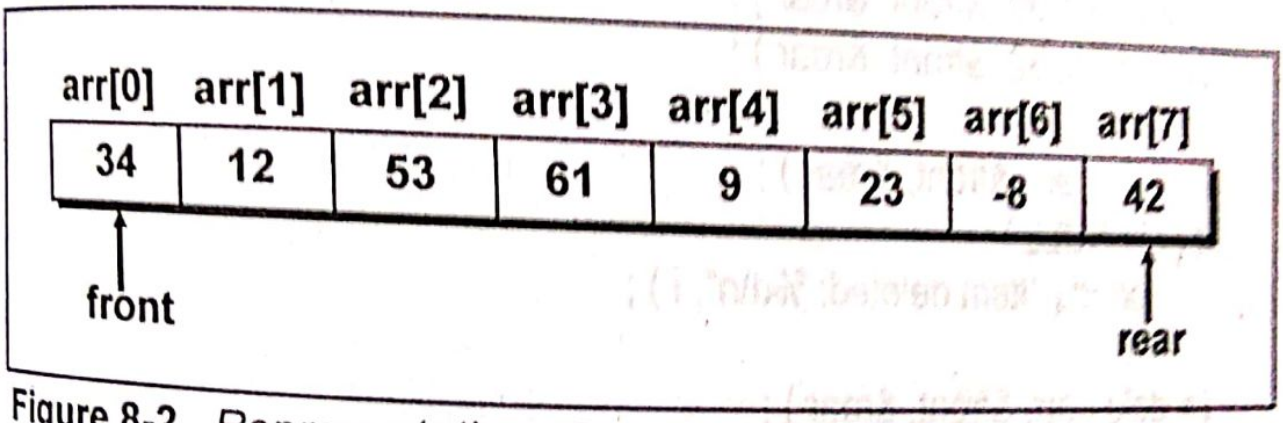

Figure 8-2. *Representation of a queue as an array.*

Let us now see a program that implements queue as an array.

```
#include <stdio.h>
#include <conio.h>
#include <windows.h>

#define MAX 10

void addq ( int *, int, int *, int * ) ;
```

```
int delq ( int *, int *, int * ) ;

int main( )
{
    int arr[MAX] ;
    int front = -1, rear = -1, i ;

    system ( "cls" ) ;

    addq ( arr, 23, &front, &rear ) ;
    addq ( arr, 9, &front, &rear ) ;
    addq ( arr, 11, &front, &rear ) ;
    addq ( arr, -10, &front, &rear ) ;
    addq ( arr, 25, &front, &rear ) ;
    addq ( arr, 16, &front, &rear ) ;
    addq ( arr, 17, &front, &rear ) ;
    addq ( arr, 22, &front, &rear ) ;
    addq ( arr, 19, &front, &rear ) ;
    addq ( arr, 30, &front, &rear ) ;
    addq ( arr, 32, &front, &rear ) ;

    i = delq ( arr, &front, &rear ) ;
    if ( i != NULL )
        printf ( "Item deleted: %d\n", i ) ;

    i = delq ( arr, &front, &rear ) ;
    if ( i != NULL )
        printf ( "Item deleted: %d\n", i ) ;

    i = delq ( arr, &front, &rear ) ;
    if ( i != NULL )
        printf ( "Item deleted: %d\n", i ) ;

    return 0 ;
}

/* adds an element to the queue */
void addq ( int *arr, int item, int *pfront, int *prear )
```

```
{
    if ( *prear == MAX - 1 )
    {
        printf ( "Queue is full.\n" ) ;
        return ;
    }

    ( *prear )++ ;
    arr[*prear] = item ;

    if ( *pfront == -1 )
        *pfront = 0 ;
}

/* removes an element from the queue */
int delq ( int *arr, int *pfront, int *prear )
{
    int data ;

    if ( *pfront == -1 )
    {
        printf ( "Queue is Empty.\n" ) ;
        return NULL ;
    }

    data = arr[*pfront] ;
    arr[*pfront] = 0 ;
    if ( *pfront == *prear )
        *pfront = *prear = -1 ;
    else
        ( *pfront )++ ;

    return data ;
}
```

Representation Of A Queue As A Linked-List

Queue can also be represented using a linked list. As discussed earlier, linked lists do not have any restrictions on the number of elements it can hold. Space for the elements in a linked list is allocated dynamically, hence it can grow as long as there is enough memory available for dynamic allocation. The item in the queue represented as a linked list would be a structure as shown below:

```
struct node
{
    <dataType> data ;
    struct node *link ;
};
```

where dataType represents the type of data such as an **int**, float, **char**, etc. Figure 8-5 shows the representation of a queue as a linked list.

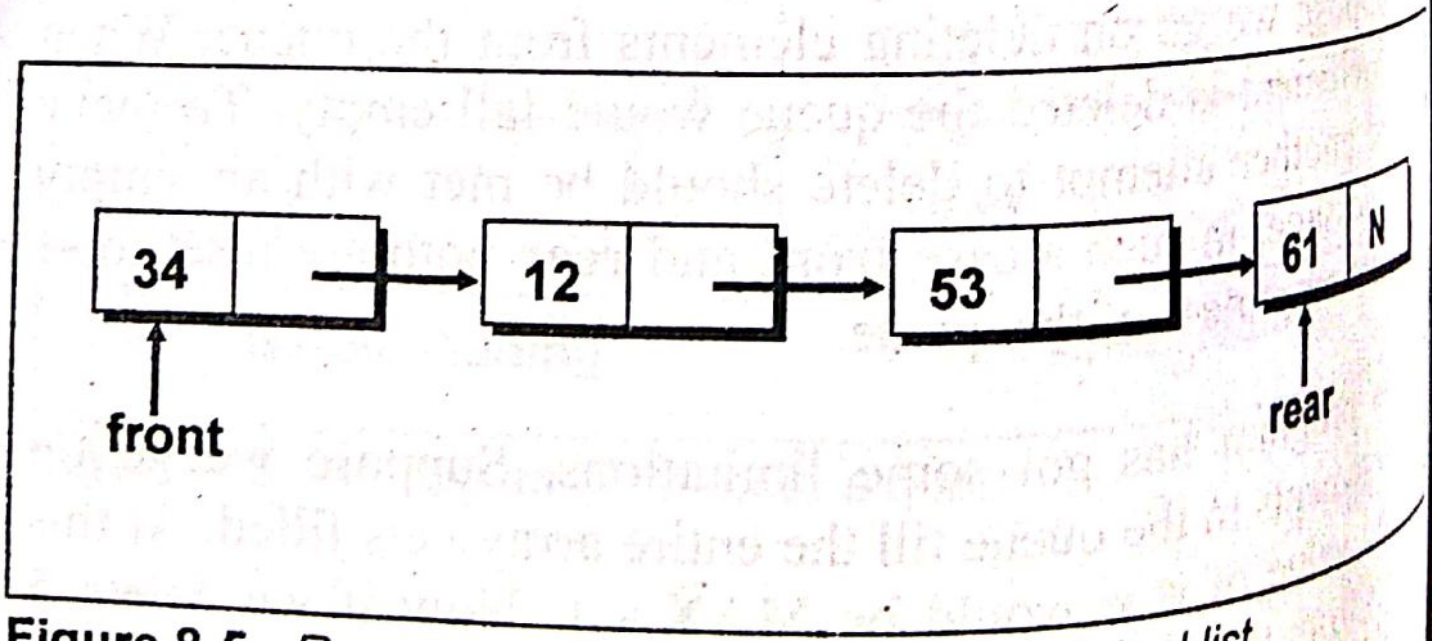

Figure 8-5. *Representation of a queue as a linked list.*

Let us now see a program that implements the queue as a linked list.

```
#include <stdio.h>
#include <conio.h>
#include <windows.h>

struct node
{
    int data ;
    struct node *link ;
};

struct queue
{
    struct node *front ;
    struct node *rear ;
};

void initqueue ( struct queue * ) ;
void addq ( struct queue *, int ) ;
int delq ( struct queue * ) ;
void delqueue ( struct queue * ) ;

int main( )
{
    struct queue a ;
    int i ;

    system ( "cls" ) ;

    initqueue ( &a ) ;

    addq ( &a, 11 ) ;
    addq ( &a, -8 ) ;
    addq ( &a, 23 ) ;
    addq ( &a, 19 ) ;
```

```
    addq ( &a, 15 ) ;
    addq ( &a, 16 ) ;
    addq ( &a, 28 ) ;

    i = delq ( &a ) ;
    if ( i != NULL )
        printf ( "Item extracted: %d\n", i ) ;

    i = delq ( &a ) ;
    if ( i != NULL )
        printf ( "Item extracted: %d\n", i ) ;

    i = delq ( &a ) ;
    if ( i != NULL )
        printf ( "Item extracted: %d\n", i ) ;

    delqueue ( &a ) ;

    return 0 ;
}

/* initialises data member */
void initqueue ( struct queue *q )
{
    q -> front = q -> rear = NULL ;
}

/* adds an element to the queue */
void addq ( struct queue *q, int item )
{
    struct node *temp ;

    temp = ( struct node * ) malloc ( sizeof ( struct node ) ) ;
    if ( temp == NULL )
        printf ( "Queue is full.\n" ) ;

    temp -> data = item ;
    temp -> link = NULL ;
```

```
    if ( q -> front == NULL )
    {
        q -> rear = q -> front = temp ;
        return ;
    }

    q -> rear -> link = temp ;
    q -> rear = q -> rear -> link ;
}

/* removes an element from the queue */
int delq ( struct queue * q )
{
    struct node *temp ;
    int item ;

    if ( q -> front == NULL )
    {
        printf ( "Queue is empty.\n" ) ;
        return NULL ;
    }

    item = q -> front  -> data ;
    temp = q -> front ;
    q -> front = q -> front -> link ;
    free ( temp ) ;
    return item ;
}

/* deallocates memory */
void delqueue ( struct queue *q )
{
    struct node *temp ;

    if ( q -> front == NULL )
        return ;
```

```
    while ( q -> front != NULL )
    {
        temp = q -> front ;
        q -> front = q -> front -> link ;
        free ( temp ) ;
    }
}
```

Output:

```
Item extracted: 11
Item extracted: -8
Item extracted: 23
```

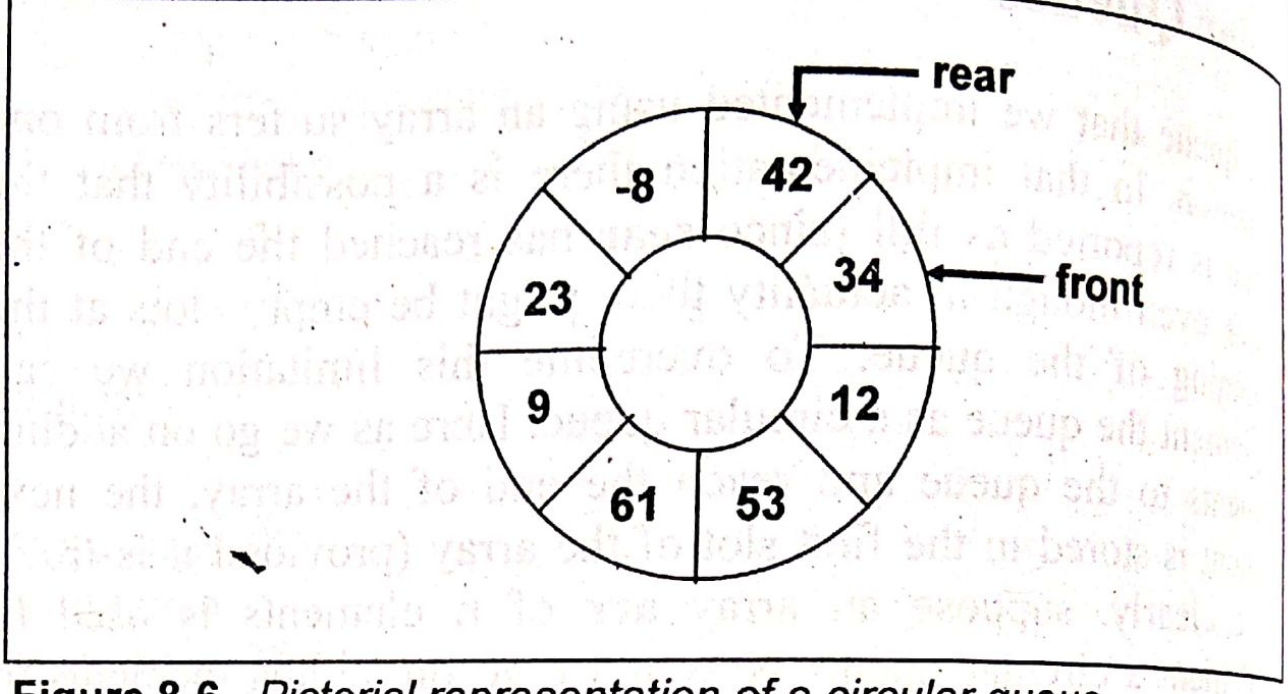

Figure 8-6. *Pictorial representation of a circular queue.*

Let us now see a program that performs the addition and deletion operation on a circular queue.

```
#include <stdio.h>
#include <conio.h>
#include <windows.h>

#define MAX 10

void addq ( int *, int, int *, int * ) ;
int delq ( int *, int *, int * ) ;
void display ( int * ) ;

int main( )
{
    int arr[MAX] ;
    int i, front, rear ;

    system ( "cls" ) ;

    /* initialise data member */
```

```
front = rear = -1 ;
for ( i = 0 ; i < MAX ; i++ )
    arr[i] = 0 ;

addq ( arr, 14, &front, &rear ) ;
addq ( arr, 22, &front, &rear ) ;
addq ( arr, 13, &front, &rear ) ;
addq ( arr, -6, &front, &rear ) ;
addq ( arr, 25, &front, &rear ) ;

printf ( "Elements in the circular queue:\n" ) ;
display ( arr ) ;

i = delq ( arr, &front, &rear ) ;
if ( i != NULL )
    printf ( "Item deleted: %d\n", i ) ;

i = delq ( arr, &front, &rear ) ;
if ( i != NULL )
    printf ( "Item deleted: %d\n", i ) ;

printf ( "Elements in the circular queue after deletion:\n" ) ;
display ( arr ) ;

addq ( arr, 21, &front, &rear ) ;
addq ( arr, 17, &front, &rear ) ;
addq ( arr, 18, &front, &rear ) ;
addq ( arr, 9, &front, &rear ) ;
addq ( arr, 20, &front, &rear ) ;

printf ( "Elements in the circular queue after addition:\n" ) ;
display ( arr ) ;

addq ( arr, 32, &front, &rear ) ;

printf ( "Elements in the circular queue after addition:\n" ) ;
display ( arr ) ;
```

```
    return 0 ;
}

/* adds an element to the queue */
void addq ( int *arr, int item, int *pfront, int *prear )
{
    if ( ( *prear == MAX - 1 && *pfront == 0 ) || ( *prear + 1 == *pfront ))
    {
        printf ( "Queue is full.\n" ) ;
        return ;
    }

    if ( *prear == MAX - 1 )
        *prear = 0 ;
    else
        ( *prear )++ ;

    arr[*prear] = item ;

    if ( *pfront == -1 )
        *pfront = 0 ;
}

/* removes an element from the queue */
int delq ( int *arr, int *pfront, int *prear )
{
    int data ;

    if ( *pfront == -1 )
    {
        printf ( "Queue is empty.\n" ) ;
        return NULL ;
    }

    data = arr[*pfront] ;
    arr[*pfront] = 0 ;

    if ( *pfront == *prear )
```

```
    {
        *pfront = -1 ;
        *prear = -1 ;
    }
    else
    {
        if ( *pfront == MAX - 1 )
            *pfront = 0 ;
        else
            ( *pfront )++ ;
    }
    return data ;
}

/* displays element in a queue */
void display ( int * arr )
{
    int i ;
    for ( i = 0 ; i < MAX ; i++ )
        printf ( "%d\t", arr[i] ) ;
    printf ( "\n" ) ;
}
```

Deque

The word **deque** is a short form of double-ended queue and defines a data structure in which items can be added or deleted at either the front or rear end, but no changes can be made elsewhere in the list. Thus a deque is a generalization of both a stack and a queue. Figure 8-9 shows the representation of a deque.

Let us now see a program that implements a deque using an ar

```
#include <stdio.h>
#include <conio.h>
#include <windows.h>

#define MAX 10

void addqatbeg ( int *, int, int *, int * ) ;
void addqatend ( int *, int, int *, int * ) ;
int delqatbeg ( int *, int *, int * ) ;
int delqatend ( int *, int *, int * ) ;
void display ( int * ) ;
int count ( int * ) ;

int main( )
{
    int arr[MAX] ;
    int front, rear, i, n ;

    system ( "cls" ) ;

    /* initialises data members */
    front = rear = -1 ;
    for ( i = 0 ; i < MAX ; i++ )
        arr[i] = 0 ;
```

```
addqatend ( arr, 17, &front, &rear ) ;
addqatbeg ( arr, 10, &front, &rear ) ;
addqatend ( arr, 8, &front, &rear ) ;
addqatbeg ( arr, -9, &front, &rear ) ;
addqatend ( arr, 13, &front, &rear ) ;
addqatbeg ( arr, 28, &front, &rear ) ;
addqatend ( arr, 14, &front, &rear ) ;
addqatbeg ( arr, 5, &front, &rear ) ;
addqatend ( arr, 25, &front, &rear ) ;
addqatbeg ( arr, 6, &front, &rear ) ;
addqatend ( arr, 21, &front, &rear ) ;
addqatbeg ( arr, 11, &front, &rear ) ;

printf ( "Elements in a deque:\n" ) ;
display ( arr ) ;

n = count ( arr ) ;
printf ( "Total number of elements in deque: %d\n", n ) ;

i = delqatbeg ( arr, &front, &rear ) ;
if ( i != NULL )
    printf ( "Item extracted: %d\n", i ) ;

i = delqatbeg ( arr, &front, &rear ) ;
if ( i != NULL )
    printf ( "Item extracted: %d\n", i ) ;

i = delqatbeg ( arr, &front, &rear ) ;
if ( i != NULL )
    printf ( "Item extracted: %d\n", i ) ;

i = delqatbeg ( arr, &front, &rear ) ;
if ( i != NULL )
    printf ( "Item extracted: %d\n", i ) ;

printf ( "Elements in a deque after deletion:\n" ) ;
display ( arr ) ;
```

```
    addqatend ( arr, 16, &front, &rear ) ;
    addqatend ( arr, 7, &front, &rear ) ;

    printf ( "Elements in a deque after addition:\n" ) ;
    display ( arr ) ;

    i = delqatend ( arr, &front, &rear ) ;
    if ( i != NULL )
        printf ( "Item extracted: %d\n", i ) ;

    i = delqatend ( arr, &front, &rear ) ;
    if ( i != NULL )
        printf ( "Item extracted: %d\n", i ) ;

    printf ( "Elements in a deque after deletion:\n" ) ;
    display ( arr ) ;

    n = count ( arr ) ;
    printf ( "Total number of elements in deque: %d\n", n ) ;

    return 0 ;
}

/* adds an element at the beginning of a deque */
void addqatbeg ( int *arr, int item, int *pfront, int *prear )
{
    int i, k, c ;

    if ( *pfront == 0 && *prear == MAX - 1 )
    {
        printf ( "Deque is full.\n" ) ;
        return ;
    }

    if ( *pfront == -1 )
    {
        *pfront = *prear = 0 ;
```

```
        arr[*pfront] = item ;
        return ;
    }

    if ( *prear != MAX - 1 )
    {
        c = count ( arr ) ;
        k = *prear + 1 ;
        for ( i = 1 ; i <= c ; i++ )
        {
            arr[k] = arr[k - 1] ;
            k-- ;
        }
        arr[k] = item ;
        *pfront = k ;
        ( *prear )++ ;
    }
    else
    {
        ( *pfront )-- ;
        arr[*pfront] = item ;
    }
}

/* adds an element at the end of a deque */
void addqatend ( int *arr, int item, int *pfront, int *prear )
{
    int i, k ;

    if ( *pfront == 0 && *prear == MAX - 1 )
    {
        printf ( "Deque is full.\n" ) ;
        return ;
    }

    if ( *pfront == -1 )
    {
        *prear = *pfront = 0 ;
```

```
        arr[*prear] = item ;
        return ;

    }
    if ( *prear == MAX - 1 )
    {
        k = *pfront - 1 ;
        for ( i = *pfront - 1 ; i < *prear ; i++ )
        {
            k = i ;
            if ( k == MAX - 1 )
                arr[k] = 0 ;
            else
                arr[k] = arr[i + 1] ;
        }
        ( *prear )-- ;
        ( *pfront )-- ;
    }
    ( *prear )++ ;
    arr[*prear] = item ;
}

/* removes an element from the *pfront end of deque */
int delqatbeg ( int *arr, int *pfront, int *prear )
{
    int item ;

    if ( *pfront == -1 )
    {
        printf ( "Deque is empty.\n" ) ;
        return 0 ;
    }

    item = arr[*pfront] ;
    arr[*pfront] = 0 ;

    if ( *pfront == *prear )
        *pfront = *prear = -1 ;
```

```
    else
        ( *pfront )++ ;

    return item ;
}

/* removes an element from the *prear end of the deque */
int delqatend ( int *arr, int *pfront, int *prear )
{
    int item ;

    if ( *pfront == -1 )
    {
        printf ( "Deque is empty.\n" ) ;
        return 0 ;
    }

    item = arr[*prear] ;
    arr[*prear] = 0 ;
    ( *prear )-- ;
    if ( *prear == -1 )
        *pfront = -1 ;
    return item ;
}

/* displays elements of a deque */
void display ( int *arr )
{
    int i ;
    printf ( "\n" ) ;
    printf ( "front-> " ) ;
    for ( i = 0 ; i < MAX ; i++ )
        printf ( "%d\t", arr[i] ) ;
    printf ( " <-rear" ) ;
    printf ( "\n" ) ;
}

/* counts the total number of elements in deque */
```

```
int count ( int *arr )
{
    int c = 0, i ;

    for ( i = 0 ; i < MAX ; i++ )
    {
        if ( arr[i] != 0 )
            c++ ;

    }
    return c ;
}
```

Output:

```
Deque is full.
Deque is full.
Elements in a deque:

front-> 6    5    28    -9    10    17    8    13    14
25    <-rear
Total number of elements in deque: 10
Item extracted: 6
Item extracted: 5
Item extracted: 28
Item extracted: -9
Elements in a deque after deletion:

front-> 0    0    0    0    10    17    8    13    14
25    <-rear
Elements in a deque after addition:

front-> 0    0    10    17    8    13    14    25    16
7    <-rear
Item extracted: 7
Item extracted: 16
Elements in a deque after deletion:
```

Priority Queue

A priority queue is a collection of elements where the elements are stored according to their priority levels. The order in which elements should get added or removed is decided by the priority of the element. Following rules are applied to maintain a priority queue.

(a) The element with a higher priority is processed before an element of lower priority.

(b) If there are elements with the same priority, then the element added first in the queue would get processed.

Priority queues are used for implementing job scheduling by the operating system where jobs with higher priorities are to be processed first. Another application of priority queues is simulation systems where priority corresponds to event times.

Array Implementation Of A Priority Queue

Like stacks and queues even a priority queue can be represented using an array. However, if an array is used to store elements of a priority queue, then insertion of elements to the queue would be easy, but deletion of elements would be difficult. This is because while inserting elements in the priority queue they are not inserted in an order. As a result, deleting an element with the highest priority would require examining the entire array to search for such an element. Moreover, an element in a queue can be deleted from the front end only.

There is no satisfactory solution to this problem. However, it would be more efficient if we store the elements in a queue in an

Given below is a program that implements the priority queue using an array.

```
#include <stdio.h>
#include <conio.h>
#include <windows.h>

#define MAX 5

struct data
{
    char job[MAX] ;
    int prno ;
    int ord ;
};

struct pque
{
    struct data d[MAX] ;
    int front ;
    int rear ;
```

```
};

void initpque ( struct pque * ) ;
void add ( struct pque *, struct data ) ;
struct data del ( struct pque * ) ;

int main( )
{
    struct pque q ;
    struct data dt, temp ;
    int i, j = 0 ;

    system ( "cls" ) ;

    initpque ( &q ) ;

    printf ( "Enter Job description (max 4 chars) and its priority\n" ) ;
    printf ( "Lower the priority number, higher the priority\n" ) ;
    printf ( "Job    Priority\n" ) ;

    for ( i = 0 ; i < MAX ; i++ )
    {
        scanf ( "%s %d", &dt.job, &dt.prno ) ;
        dt.ord = j++ ;
        add ( &q, dt ) ;
    }
    printf ( "\n" ) ;

    printf ( "Process jobs priority wise\n" ) ;
    printf ( "Job\tPriority\n" ) ;

    for ( i = 0 ; i < MAX ; i++ )
    {
        temp  = del ( &q ) ;
        printf ( "%s\t%d\n", temp.job, temp.prno ) ;
    }
    printf ( "\n" ) ;
```

```
    return 0 ;
}
/* initialises data members */
void initpque ( struct pque *pq )
{
    int i ;

    pq -> front = pq -> rear = -1 ;
    for ( i = 0 ; i < MAX ; i++ )
    {
        strcpy ( pq -> d[i].job, "" ) ;
        pq -> d[i].prno = pq -> d[i].ord = 0 ;
    }
}

/* adds item to the priority queue */
void add ( struct pque *pq, struct data dt )
{
    struct data temp ;
    int i, j ;

    if ( pq -> rear == MAX - 1 )
    {
        printf ( "Queue is full.\n" ) ;
        return ;
    }

    pq -> rear++ ;
    pq -> d[pq -> rear] = dt ;

    if ( pq -> front == -1 )
        pq -> front = 0 ;

    for ( i = pq -> front ; i <= pq -> rear ; i++ )
    {
        for ( j = i + 1 ; j <= pq -> rear ; j++ )
        {
```

```
                if ( pq -> d[i].prno > pq -> d[j].prno )
                {
                    temp = pq -> d[i] ;
                    pq -> d[i] = pq -> d[j] ;
                    pq -> d[j] = temp ;
                }
                else
                {
                    if ( pq -> d[i].prno == pq -> d[j].prno )
                    {
                        if ( pq -> d[i].ord > pq -> d[j].ord )
                        {
                            temp = pq -> d[i] ;
                            pq -> d[i] = pq -> d[j] ;
                            pq -> d[j] = temp ;
                        }
                    }
                }
            }
        }
}

/* removes item from priority queue */
struct data  del ( struct pque *pq )
{
    struct data  t ;
    strcpy ( t.job, "" ) ;
    t.prno = 0 ;
    t.ord = 0 ;

    if ( pq -> front == -1 )
    {
        printf ( "Queue is Empty.\n" ) ;
        return t ;
    }

    t = pq -> d[pq -> front] ;
    pq -> d[pq -> front] = t ;
```

```
    if ( pq -> front == pq -> rear )
        pq -> front = pq -> rear = -1 ;
    else
        pq -> front++ ;

    return t ;
}
```

Output:

```
Enter Job description (max 4 chars) and its priority
Lower the priority number, higher the priority
Job   Priority
TYPE  4
SWAP  3
COPY  5
PRNT  1
SWAP  3

Process jobs priority wise
Job   Priority
PRNT  1
SWAP  3
SWAP  3
TYPE  4
COPY  5
```