Chapter Table of Contents

Chapter 3.2

Queues

Aim 173
Instructional Objectives 173
Learning Outcomes 173
3.2.1 Introduction to Queue 174
    (i) Definition of a Queue 174
    (ii) Array Representation of Queue 175
    Self-assessment Questions 180
3.2.2 Types of Queue 181
    (i) Simple Queue 181
    (ii) Circular Queue 182
    (iii) Double Ended Queue 188
    (iv) Priority Queue 194
    Self-assessment Questions 196
3.2.3 Operations on Queue 196
    (i) Insertion 197
    (ii) Deletion in Queue 198
    (iii) Qempty Operation 199
    (iv) Qfull Operation 200
    (v) Display Operation 200
    Self-assessment Questions 201
3.2.4 Application of Queue 202
    Self-assessment Questions 203
Summary 204
Terminal Questions 205
Answer Keys 205
Activity 206
Bibliography 207
e-References 207
External Resources 207
Video Links 207

Aim

To educate the students with the basic knowledge of queues, its types and operations on queues

Instructional Objectives

After completing this chapter, you should be able to:

- Explain queue and its operations
- Describe the array representation of Queue
- Discuss different types of queue with example
- Illustrate the creation, insertion, deletion and search operation on various types of queue

Learning Outcomes

At the end of this chapter, you are expected to:

- Demonstrate queue with its operations
- Implement double ended queue using linked list
- Identify requirement of priority queue

3.2.1 Introduction to Queue

In simple language a queue is a simple waiting line which keeps growing if we add the elements to its end and keep shrinking on removal of elements from its front. If we compare stack, queue reflects the more commonly used maxim in real-world, that is, “first come, first served”. Long waiting lines in food counters, supermarkets, banks are common examples of queues.

For all computer applications, we define a queue as list in which all additions to the list are made at one end, and all deletions from the list are made at the other end. Applications of queues are, if anything, even more common than are applications of stacks, since in performing tasks by computer, as in all parts of life, it is often necessary to wait one’s turn before having access to something. Within a computer processor there can be queues of tasks waiting for different devices like printer, for access to disk storage, or even, with multitasking, for using the CPU. Within a single program, there may be multiple requests to be kept in a queue, or one task may create other task, which must be done in turn by keeping them in a queue.

A queue is a data structure where elements are added at the back and remove elements from the front. In that way a queue is like “waiting in line”: the first one to be added to the queue will be the first one to be removed from the queue.

Queues are common in many applications. ***For example***, while we read a book from a file, it is quite natural to store the read words in a queue so that once reading is complete the words are in the order as they appear in the book. Another common example is buffer for network communication that temporarily store packets of data arriving on a network port. Generally speaking, it is processed in the order in which the elements arrive.

(i) Definition of a Queue

In a more formal way, queue can be defined as a list or a data structure in which data items can be added at the end (generally referred as rear) and they can be deleted from font of the queue. The data element to be deleted is the one which would spend maximum time in the queue. It is because of this property, queue is also referred to as a first-in-first-out (FIFO) data structure. The figure 3.2.1 below shows pictorial representation of a queue.

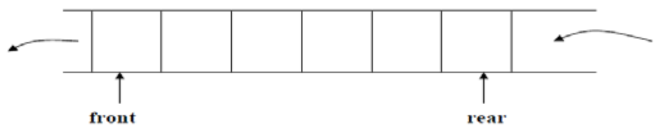


Figure 3.2.1: Representation of a Queue in Computer’s Memory

Generally a queue can also be referred to as a container of objects (in other words linear collection) that are deleted or added based on principle of First-In-First-Out (FIFO). A very good example of a queue can be a line of students in the ice-cream counter of the college canteen. New arrival of students can be added to a line at back of the queue, while removal serving (or removal) happens in the front of the queue. The queue allows only two operations; enqueue and dequeues. Enqueue is an operation that allows insertion operation, dequeue allows us to remove an item.

Stack and a queue difference lies only in deletion of item. A stack removes most recently added item; while a queue removes the least recently added item first.

In spite of its simplicity, the queue is a very important concept with many applications in simulation of real life events such as lines of customers at a cash register or cars waiting at an intersection, and in programming (such as printer jobs waiting to be processed. Many Smalltalk applications use a queue but instead of implementing it as a new class, they use an Ordered Collection because it performs all the required functions.

Dequeuing or removing an item from a queue is only possible on non-empty queues, which requires contract in the interface. This interface can be written without committing to an implementation of queues. This is important so that different implementations of the functions in this interface can choose different representations.

(ii) Array Representation of Queue

The array to implement the queue would need two variables (indices) called front and rear to point to the first and the last elements of the queue. The figure 3.2.2 shows array implementation of queue.

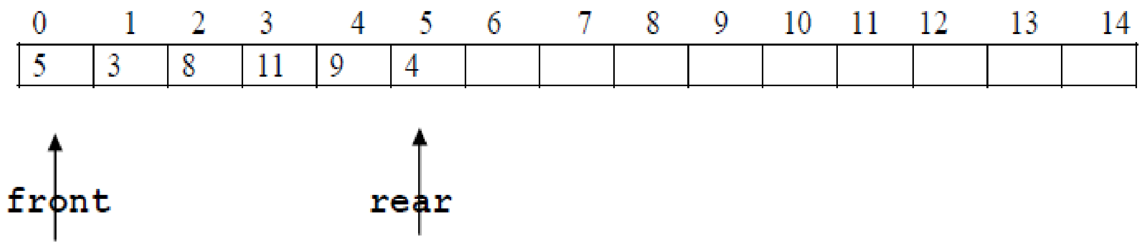


Figure 3.2.2: Array Implementation of Queue

Initially:

q->rear = -1;

q->front = -1;

For every ***enqueue*** operation we increment rear by one, and for every ***dequeue*** operation, we increment front by one. Even though ***enqueue*** and ***dequeue*** operations are simple to implement, there is a disadvantage in this set up. The size of the array required is huge, as the number of slots would go on increasing as long as there are items to be added to the list (irrespective of how many items are deleted, as these two are independent operations.)

Problems with this representation

Although there is space in the following queue in the initial blocks, we may not be able to add a new item. An attempt will cause an overflow.

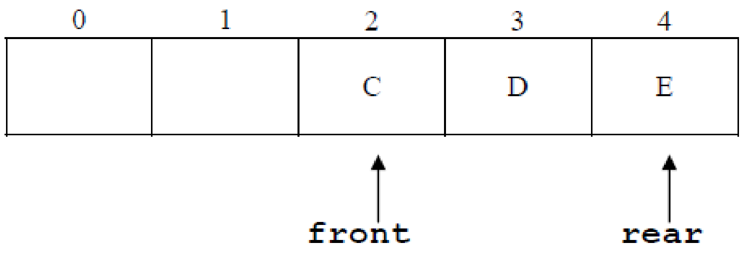


Figure 3.2.3: Queue Overflow Situation

It is possible to have an empty queue yet no new item can be inserted. (When front moves to the point of rear, and the last item is deleted.)

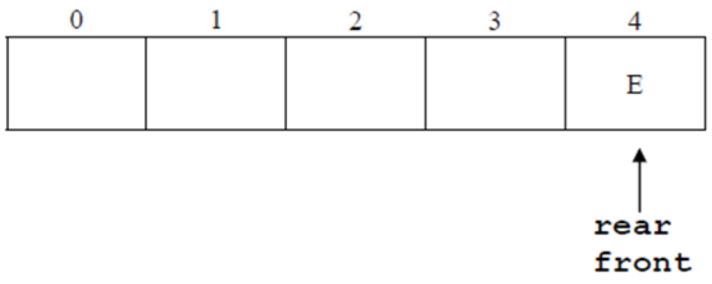


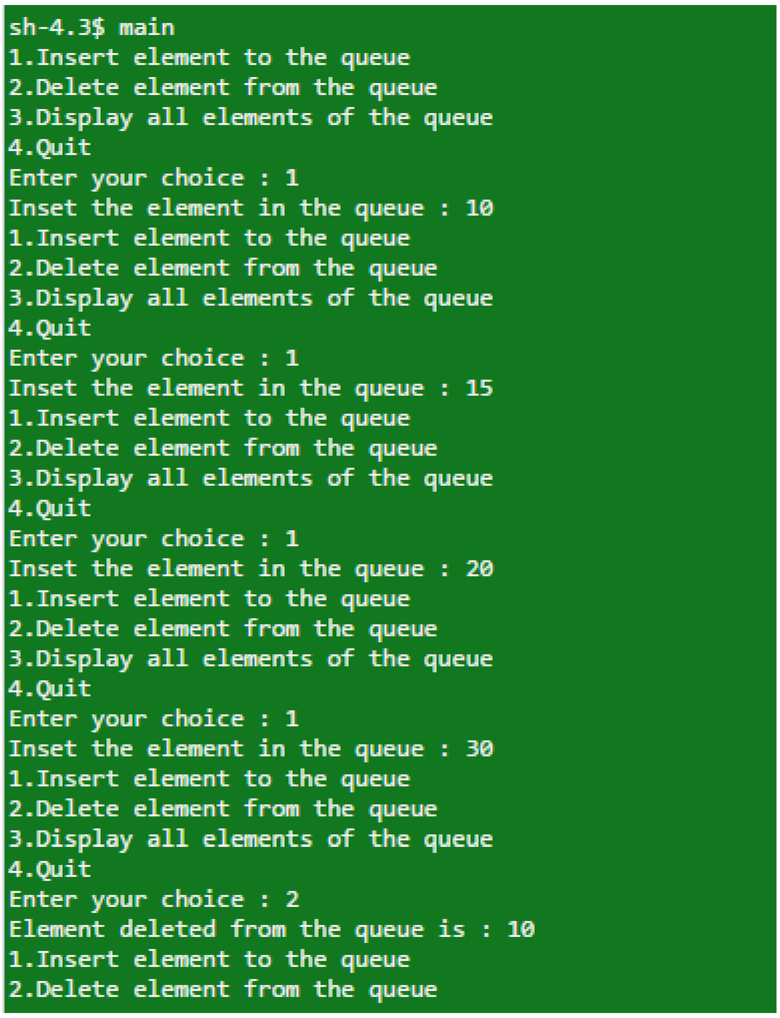
Figure 3.2.4: Overflow Situation in an Empty Queue

The below program shows the implementation of a queue using an Array

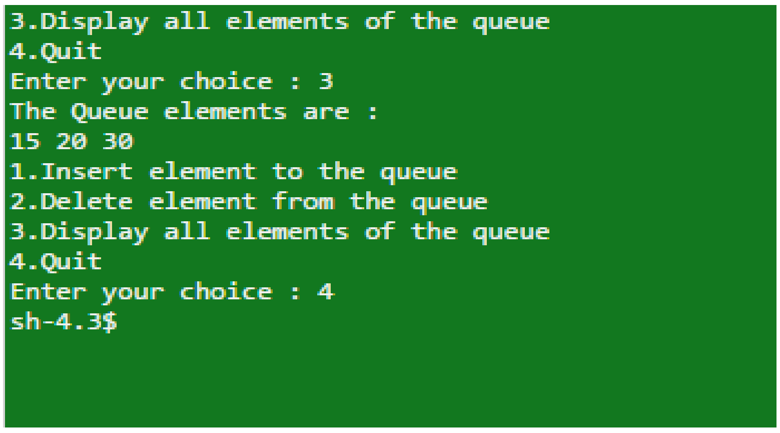
Program:

```
/*
* C Program to Implement a Queue using an Array
 */

#include<stdio.h>
#include<stdlib.h>

#define SIZE 50
int queue_arr[SIZE];
int rear =-1;
int front =-1;
int main()
{
   int ch;
   while(1)
   {
      printf("1.Insert element to the queue \n");
      printf("2.Delete element from the queue \n");
      printf("3.Display all elements of the queue \n");
      printf("4.Quit \n");
      printf("Enter your choice : ");
      scanf("%d",&ch);
      switch(ch)
      {
         case 1:
                     insert();
         break;
         case 2:
                     delete();
         break;
         case 3:
                     display();
         break;
         case 4:
         exit(1);
         default:
         printf("Invalid Input \n");
      }/*End of switch*/
   }/*End of while*/
}/*End of main()*/
insert()
{
   int add_item;
   if(rear == SIZE -1)
   printf("Queue Overflow \n");
   else
   {
      if(front ==-1)
      /*If queue is initially empty */
              front =0;
      printf("Inset the element in the queue : ");
      scanf("%d",&add_item);
              rear = rear +1;
              queue_arr[rear]= add_item;
   }
}/*End of insert()*/

delete()
{
   if(front ==-1|| front > rear)
   {
      printf("Queue Underflow \n");
      return;
   }
   else
   {
      printf("Element deleted from the queue is : %d\n", queue_arr[front]);
              front = front +1;
   }
}/*End of delete() */
display()
{
   int i;
   if(front ==-1)
   printf("Queue is empty \n");
   else
   {
      printf("The Queue elements are : \n");
      for(i = front; i <= rear; i++)
      printf("%d ", queue_arr[i]);
      printf("\n");
   }
}/*End of display() */
```

Output:



```
sh-4.3$ main
1.Insert element to the queue
2.Delete element from the queue
3.Display all elements of the queue
4.Quit
Enter your choice : 1
Inset the element in the queue : 10
1.Insert element to the queue
2.Delete element from the queue
3.Display all elements of the queue
4.Quit
Enter your choice : 1
Inset the element in the queue : 15
1.Insert element to the queue
2.Delete element from the queue
3.Display all elements of the queue
4.Quit
Enter your choice : 1
Inset the element in the queue : 20
1.Insert element to the queue
2.Delete element from the queue
3.Display all elements of the queue
4.Quit
Enter your choice : 1
Inset the element in the queue : 30
1.Insert element to the queue
2.Delete element from the queue
3.Display all elements of the queue
4.Quit
Enter your choice : 2
Element deleted from the queue is : 10
1.Insert element to the queue
2.Delete element from the queue
```



```
3.Display all elements of the queue
4.Quit
Enter your choice : 3
The Queue elements are :
15 20 30
1.Insert element to the queue
2.Delete element from the queue
3.Display all elements of the queue
4.Quit
Enter your choice : 4
sh-4.3$
```

Did you know?

Though the simple queue appears to be very simple FIFO model, it is a very important model used for any applications. During the initial implementation of most of the operating systems like Linux which is generic programmer friendly OS and others like TinyOs which is OS used by wireless sensor networks used simple FIFO queue for scheduling different tasks due to ease of implementation and limitations of resources.

Self-assessment Questions

1) Which one of the following is an application of Queue Data Structure?
   - a) When
   - b) A resource is shared among multiple devices
   - c) Printer jobs waiting to be processed.
   - d) Buffer used in network communication to store data packets
   - e) All of the above

2) For every ***enqueue*** operation, we ___________ by one, and for every ***dequeue*** operation, we ___________ by one.
   - a) Decrement rear, decrement front
   - b) Increment rear, increment front
   - c) Increment front, increment rear
   - d) Decrement front, decrement rear

3) For queue implementation, we need two pointers namely front and rear. This pointers are initialized as:

a) front=1 and rear=-1	b) front=-1 and rear=-1
c) front=-1 and rear=1	d) front=1 and rear=1

3.2.2 Types of Queue

A queue represents a basket of items. Enqueue is an operation that adds an item to this basket and dequeue is an operation that chooses an item to be removed from the queue (if the queue is not empty). Similar to human queues, these queues will vary based on the rule used to choose the item to be removed from the queue. Giving different names to the basic operations of the queue based on the operations they perform is usual and it helps in avoiding the confusion.

However, using the general signature for different kinds of queue will make our code more modular later, when algorithms based on the different kinds of queue are discussed.

(i) Simple Queue

Like the stacks, we can also implement queue using lists and arrays. Both the arrays and the linked lists implementations have running time complexity of O (1) for every operation. This section covers the array implementation of queues.

For every queue data structure, an array is kept namely, QUEUE [], and there are two positions q_front and q_rear, which represent the beginning and the ends of the queue respectively. q_size takes care of the size of the queue.

All the above information makes part of a structure, and except for the queue functions themselves, no functions should access these directly. The figure 3.2.5 shows a queue in some intermediate state. The blank cells have undefined values in them. In particular, the elements in the first two cells have spent maximum time in the queue.

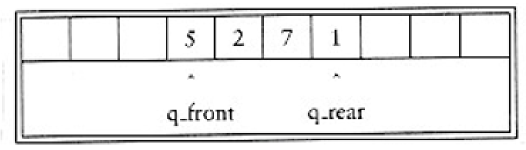


Figure 3.2.5: Basic Queue example

In order to enqueue an element x, we must first increment q_rear and q_size, then set QUEUE [q_rear] = x. To dequeue an element, assign return value to QUEUE [q_front], decrement q_size, and then increment q_front. Other strategies are possible (this is discussed later).

Using array Simple queue can be declared as

```
#define MAX 10

int queue[MAX], rear=0, front=0;
```

(ii) Circular Queue

One biggest problem with the simple queue is its implementation. After adding 10 elements in the queue (considering previously discussed situation), the queue looks like full, since q_front is 10, and the next enqueue would be in a non-existent position. However, there might some positions available in the queue, as many elements may have been already dequeued. Queues, like stacks, frequently stay small even in the presence of a lot of operations.

The solution for this problem is that whenever q_rear or q_front comes to the end of the array, it is wrapped around to the beginning. The following figure shows the queue during some operations

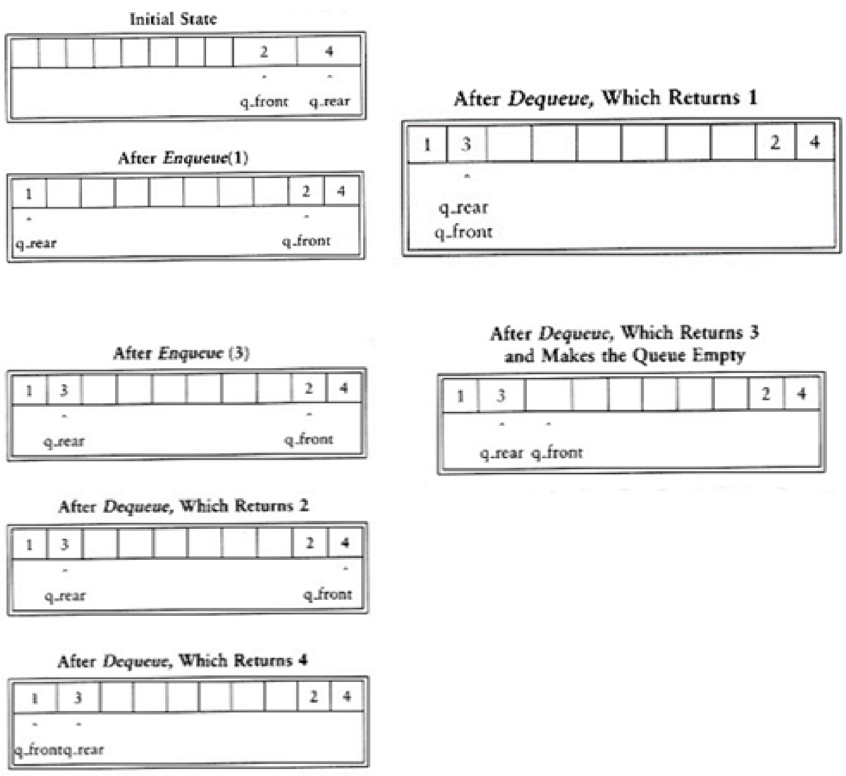

Initial State
q_front q_rear
After Enqueue(1)
q_rear q_front
After Enqueue (3)
After Dequeue, Which Returns 2
After Dequeue, Which Returns 4
q_front q_rear
After Dequeue, Which Returns 1
After Dequeue, Which Returns 3 and Makes the Queue Empty

Figure 3.2.6: Circular Queue

There is a minimal need to write an extra code to implement the wraparound although it increases the running time complexity). If incrementing either q_front or q_rear makes it to go past the array, the value is reset to the first position in the array.

However, two things are needed to be taken care of while using circular array implementation of queues. First and the foremost thing, we must check if queue is not empty, because a dequeue operation when the queue is empty returns an undefined value.

Secondly, sometimes programmers represent front and rear differently for queues. ***For example***, some programmers do not use an entry to keep track the size of the queue, because they rely on the assumption that the queue is empty, q_front = q_rear - 1. The size is computed implicitly by comparing q_front and q_rear. This is a very tricky way, since there are some special cases, therefore one need to be very careful if we need to modify code written this way. Consider the situation that size is not part of the structure, then if the size of the array is A_SIZE, the queue is full when there are A_SIZE -1 elements, since only A_SIZE different sizes can be differentiated, and one of these is 0.

In applications where it is sure that the number of enqueue is not larger than the size of the queue, obviously the wraparound is not necessary. As with stacks, dequeues are rarely performed unless the calling routines are certain that the queue is not empty. Thus error calls are frequently skipped for this operation, except in critical code. This is generally not justifiable, because the time savings that you are likely to achieve are too minimal.

We can think of an array as a circle rather than a straight line in order to overcome the inefficient use of space as depicted in the figure 3.2.7. In this way, as entries are added and removed from the queue, the head will continually chase the tail around the array, so that the snake can keep crawling indefinitely but stay in a confined circuit. At different times, the queue will occupy different parts of the array, but there no need to worry about running out of space unless the array is fully occupied, in which case there is truly overflow

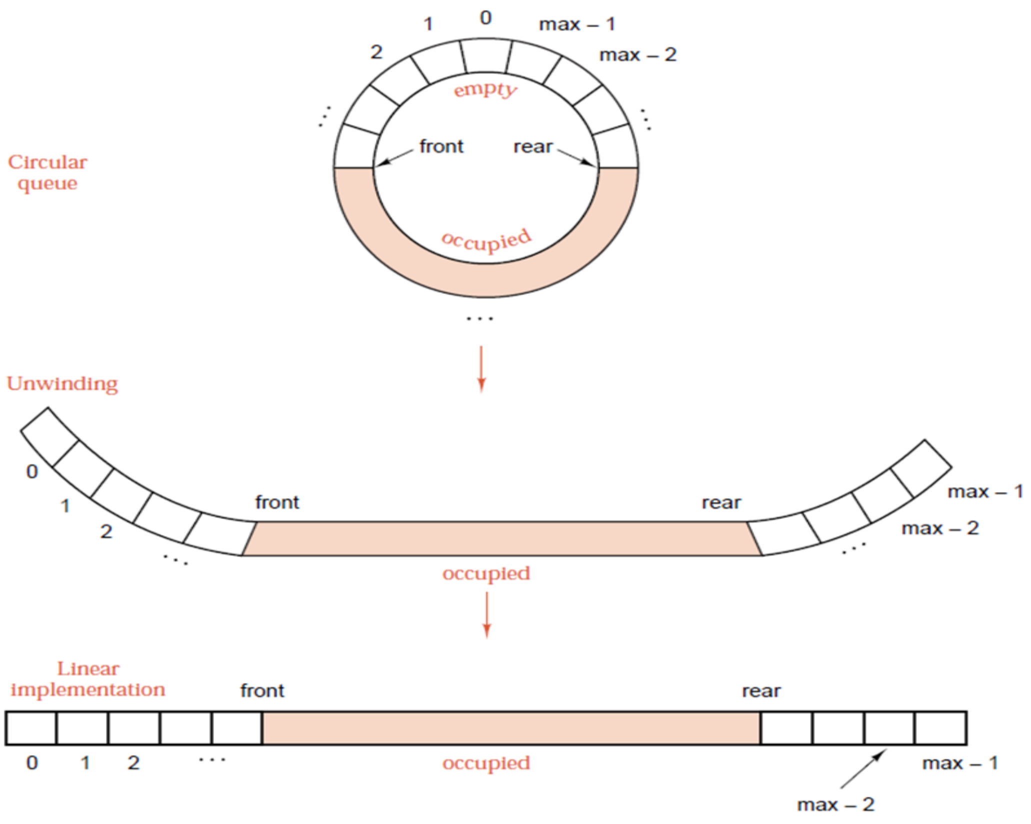


Figure 3.2.7: Queue in a Circular Array

Implementation of Circular Arrays

In order to implement the circular queue as a linear array, consider the positions around the circular arrangement as numbered from zero to max-1, where max is the total number of elements in the circular arrays. We use same numbered entries of a linear array to implement a circular array. Now it becomes a very simple logic of using modular arithmetic i.e. whenever the index crosses max-1, we start again from 0. This is as simple as doing arithmetic on circular clock face where the hours are numbered from 1 to 12, and if four hours are added to ten o'clock, two o'clock is obtained.

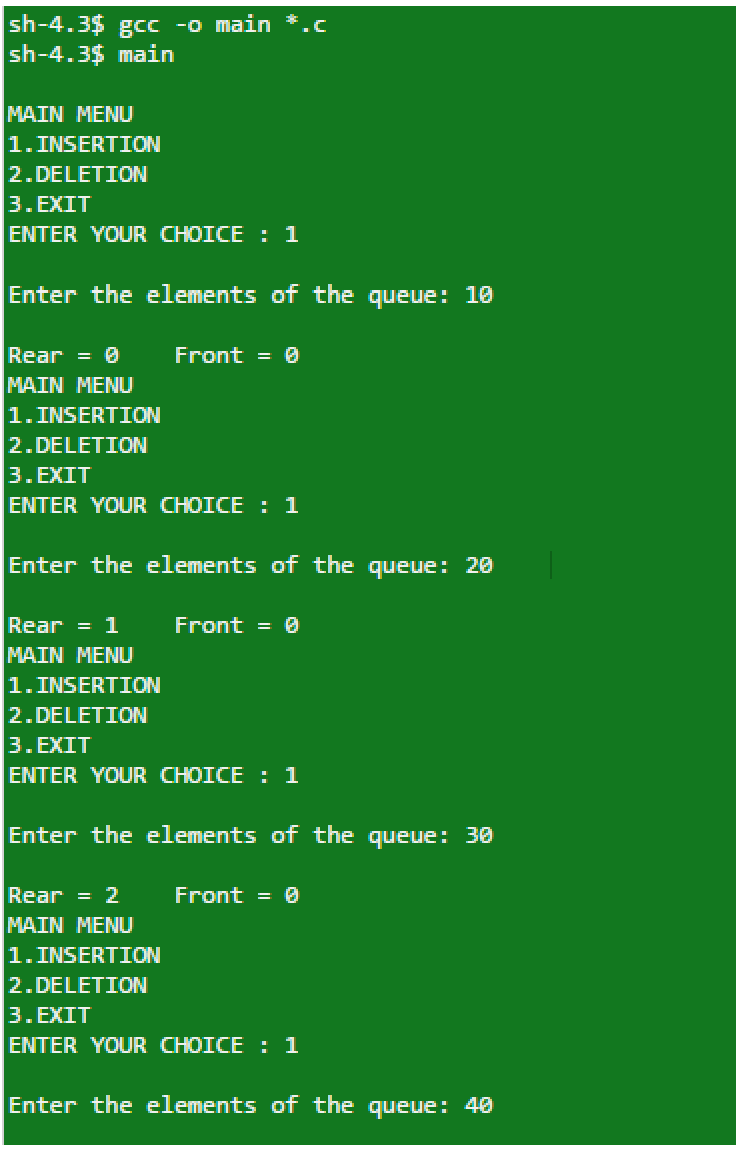
Program:

//Program for Circular Queue implementation through Array

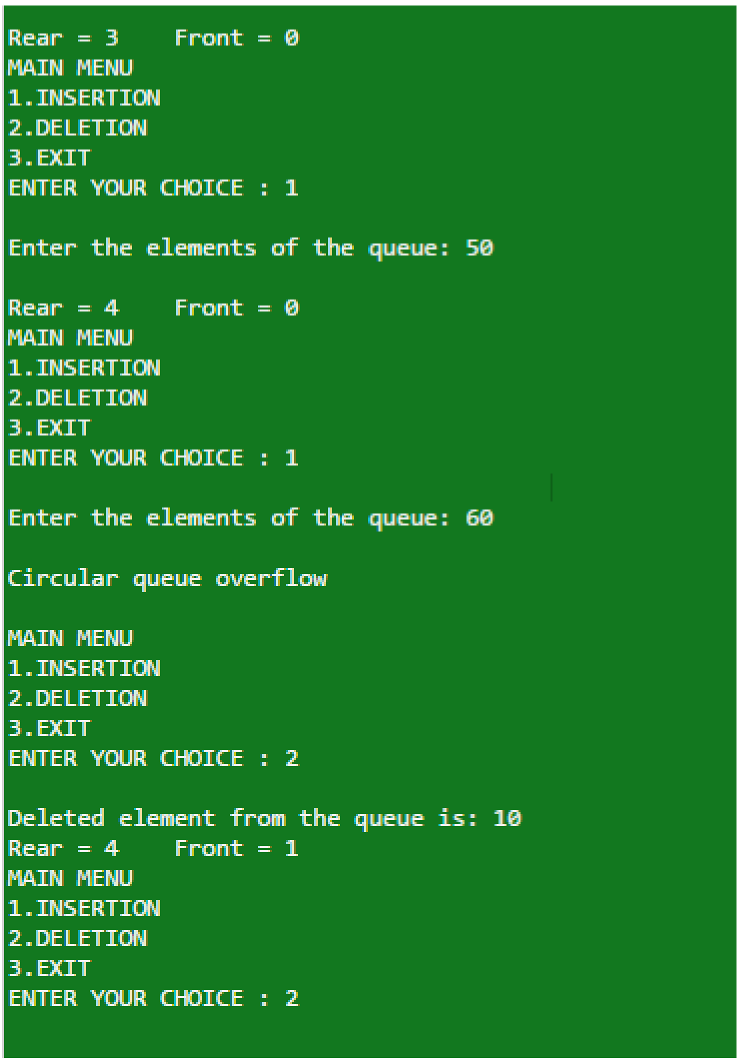
```
#include <stdio.h>
#include<ctype.h>
#include<stdlib.h>
#define SIZE 5
int circleq[SIZE];
int front,rear;
int main()
{
      void insert(int, int);
      void delete(int);
      int ch=1,i,n;
      front = -1;
      rear = -1;
      while(1)
      {
            printf("\nMAIN MENU\n1.INSERTION\n2.DELETION\n3.EXIT");
            printf("\nENTER YOUR CHOICE : ");
            scanf("%d",&ch);
            switch(ch)
            {
            case 1:
                  printf("\nEnter the elements of the queue: ");
                  scanf("%d",&n);
                  insert(n,SIZE);
                  break;
            case 2:
                  delete(SIZE);
                  break;
            case 3:
                exit(0);
                  default: printf("\nInvalid input. ");
            }

      } //end of  outer while
}               //end of main
void insert(int item,int MAX)
{
      //rear++;
      //rear= (rear%MAX);
      if(front ==(rear+1)%MAX)
      {
      printf("\nCircular queue overflow\n");
      }
      else
      {
        if(front==-1)
          front=rear=0;
        else
          rear=(rear+1)%MAX;
          circleq[rear]=item;
          printf("\nRear = %d    Front = %d ",rear,front);
      }
}
void delete(int MAX)
{
int del;
if(front == -1)
```

```
        {
        printf("\nCircular queue underflow\n");
        }
        else
        {
              del=circleq[front];
              if(front==rear)
                front=rear=-1;
              else
                front = (front+1)%MAX;
                  printf("\nDeleted element from the queue is: %d ",del);
              printf("\nRear = %d    Front = %d ",rear,front);
        }
}
```

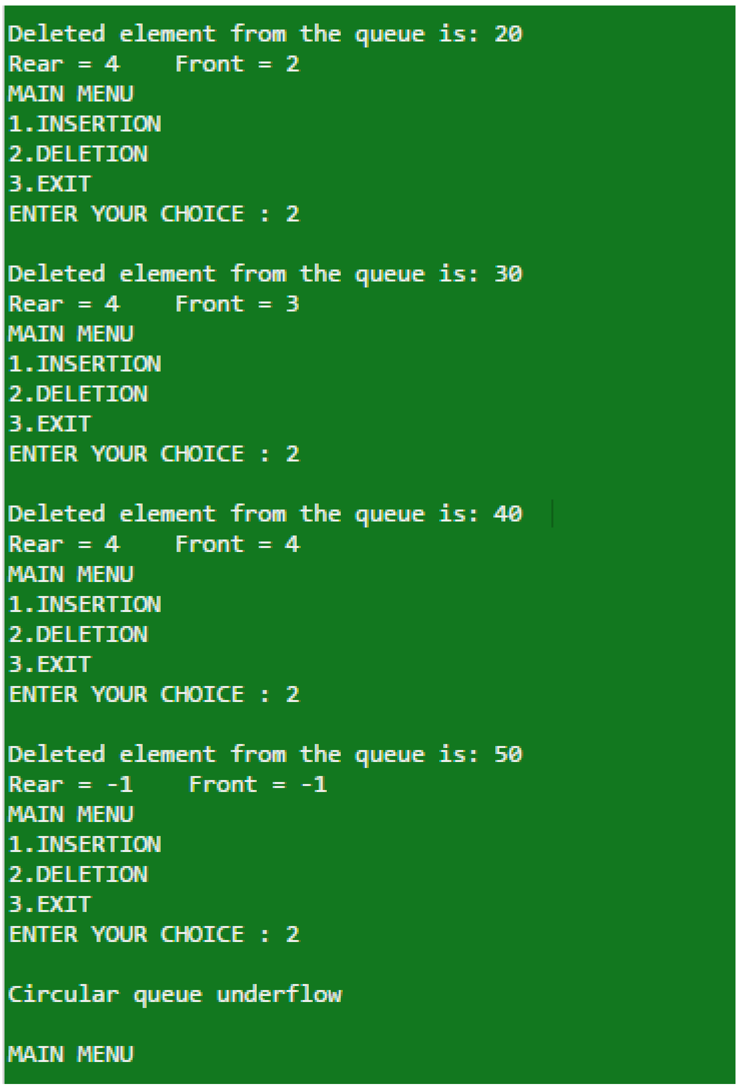
Output:



```
sh-4.3$ gcc -o main *.c
sh-4.3$ main

MAIN MENU
1.INSERTION
2.DELETION
3.EXIT
ENTER YOUR CHOICE : 1

Enter the elements of the queue: 10

Rear = 0    Front = 0
MAIN MENU
1.INSERTION
2.DELETION
3.EXIT
ENTER YOUR CHOICE : 1

Enter the elements of the queue: 20

Rear = 1    Front = 0
MAIN MENU
1.INSERTION
2.DELETION
3.EXIT
ENTER YOUR CHOICE : 1

Enter the elements of the queue: 30

Rear = 2    Front = 0
MAIN MENU
1.INSERTION
2.DELETION
3.EXIT
ENTER YOUR CHOICE : 1

Enter the elements of the queue: 40
```



```
Rear = 3    Front = 0
MAIN MENU
1.INSERTION
2.DELETION
3.EXIT
ENTER YOUR CHOICE : 1

Enter the elements of the queue: 50

Rear = 4    Front = 0
MAIN MENU
1.INSERTION
2.DELETION
3.EXIT
ENTER YOUR CHOICE : 1

Enter the elements of the queue: 60

Circular queue overflow

MAIN MENU
1.INSERTION
2.DELETION
3.EXIT
ENTER YOUR CHOICE : 2

Deleted element from the queue is: 10
Rear = 4    Front = 1
MAIN MENU
1.INSERTION
2.DELETION
3.EXIT
ENTER YOUR CHOICE : 2
```


```
Deleted element from the queue is: 20
Rear = 4    Front = 2
MAIN MENU
1.INSERTION
2.DELETION
3.EXIT
ENTER YOUR CHOICE : 2

Deleted element from the queue is: 30
Rear = 4    Front = 3
MAIN MENU
1.INSERTION
2.DELETION
3.EXIT
ENTER YOUR CHOICE : 2

Deleted element from the queue is: 40
Rear = 4    Front = 4
MAIN MENU
1.INSERTION
2.DELETION
3.EXIT
ENTER YOUR CHOICE : 2

Deleted element from the queue is: 50
Rear = -1    Front = -1
MAIN MENU
1.INSERTION
2.DELETION
3.EXIT
ENTER YOUR CHOICE : 2

Circular queue underflow

MAIN MENU
```

(iii) Double Ended Queue

A double-ended queue is also known as dequeue. This is an ordered collection of items similar to the queue. A dequeue has two ends, a front and a rear, and all the items remains positioned in the entire collection. The special thing about dequeue is the unrestrictive nature of removing and adding items. An item can be added at either the rear or the front. Similarly, the existing items can be removed from either end. This makes dequeue a hybrid linear structure that provides all the capabilities of queues and stacks in a single data structure. *Figure 3.2.8* shows a dequeue.

It is important to note that even though the dequeue can assume many of the characteristics of stacks and queues, it does not require the LIFO and FIFO orderings that are enforced by those data structures. The use of the addition and removal operations must be done consistently.

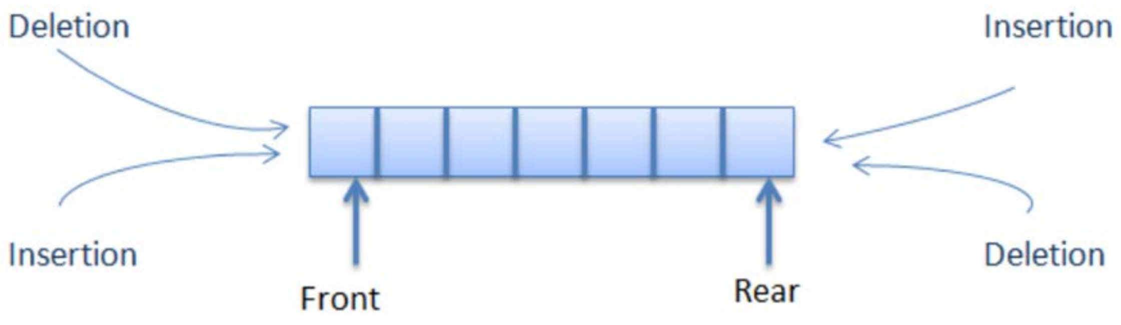


Figure 3.2.8: Dequeue

A double-ended queue (dequeue, often abbreviated to dequeue, pronounced deck) is an abstract data structure that implements a queue for which elements can only be added to or removed from the front (head) or back (tail). It is also often called a head-tail linked list.

Dequeue is a special type of data structure in which deletion and insertion can be done either at the rear end or at the front end of the queue. The operations that can be performed on dequeues are:

- Insertion of an item from front end
- Insertion of an item from rear end
- Deletion of an item from front end
- Deletion of an item from rear end
- Displaying the contents of queue

Application of dequeue

- A nice application of the dequeue is storing a web browser's history. Recently visited URLs are added to the front of the dequeue, and the URL at the back of the dequeue is removed after some specified number of insertions at the front.
- Another common application of the dequeue is storing a software application's list of undo operations.
- One example where a dequeue can be used is the A-Steal job scheduling algorithm.[5] This algorithm implements task scheduling for several processors. A separate dequeue with threads to be executed is maintained for each processor. To execute the next

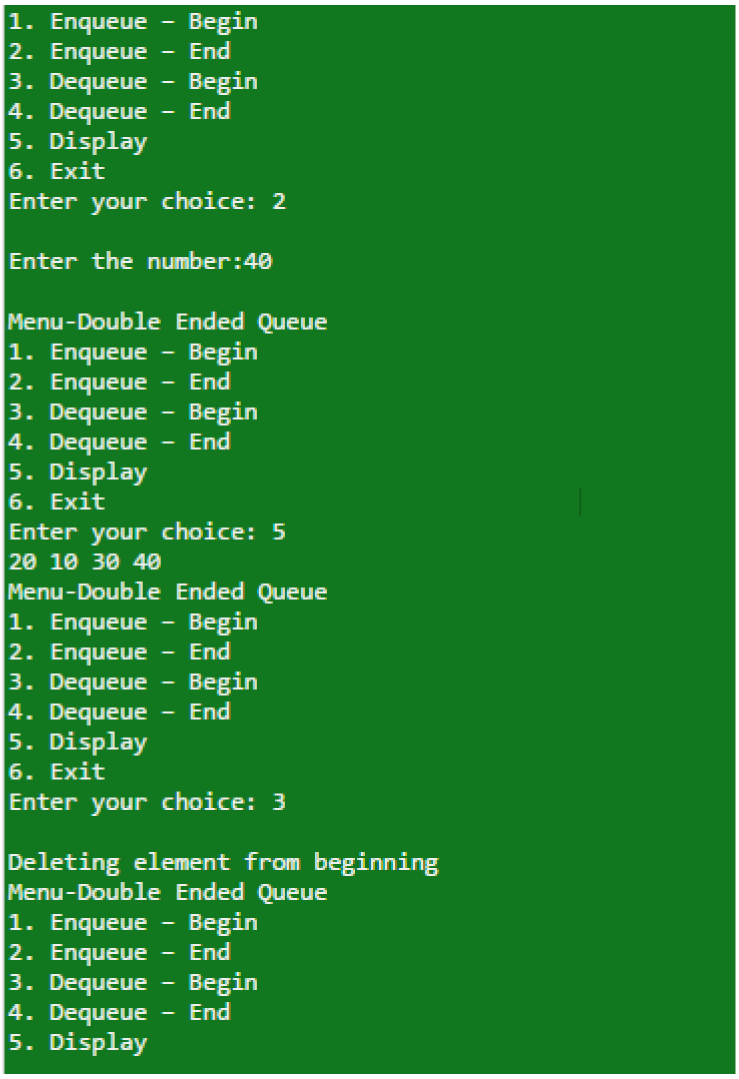
thread, the processor gets the first element from the dequeue (using the "remove first element" dequeue operation). If the current thread forks, it is put back to the front of the dequeue ("insert element at front") and a new thread is executed. When one of the processors finishes execution of its own threads (i.e. it'sdequeue is empty), it can "steal" a thread from another processor: it gets the last element from the dequeue of another processor ("remove last element") and executes it.

- In real scenario we can attached it to a Ticket purchasing line, It performs like a queue but some time It happens that somebody has purchased the ticket and sudden they come back to ask something on front of queue. In this scenario because they have already purchased the ticket so they have privilege to come and ask for any further query. So in this kind of scenario we need a data structure where according to requirement we add data from front. And In same scenario user can also leave the queue from rear.

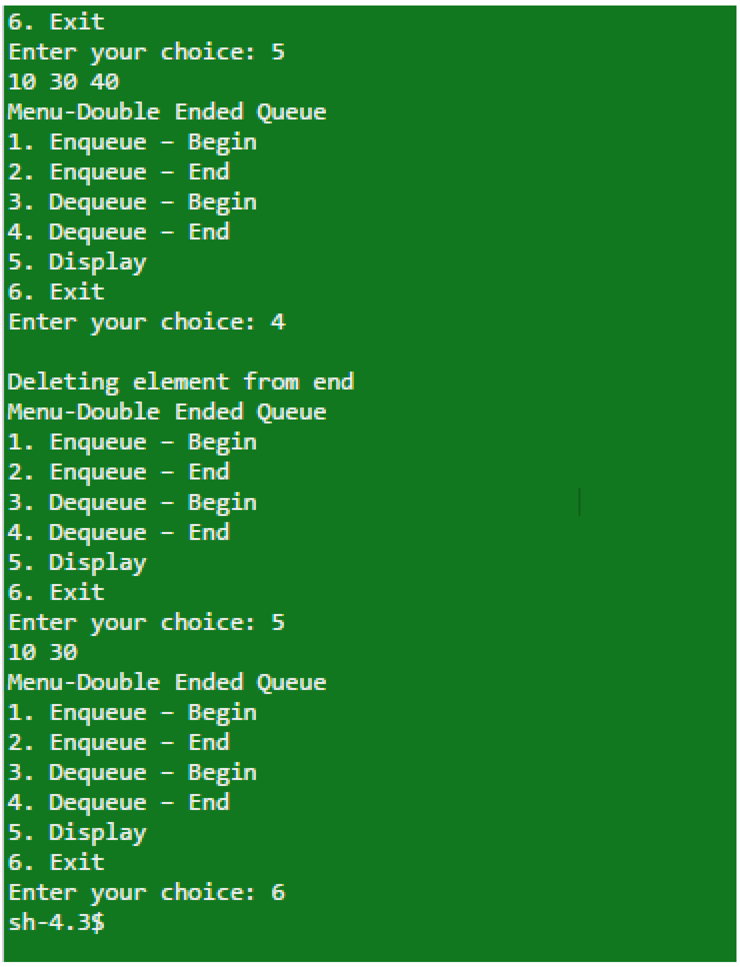
Program:

```
/*Implementation of De-queue using arrays*/
#include<stdio.h>
#include<stdlib.h>
#define MAX 10

typedef struct dequeue
{
   int front,rear;
   int arr[MAX];
}dq;
/*If flag is zero, insertion is done at beginning
else if flag is one, insertion is done at end.
*/
void enqueue(dq *q,int x,int flag)
{
   int i;
   if(q->rear==MAX-1)
   {
      printf("\nQueue overflow!");
      exit(1);
   }
   if(flag==0)
   {
      for(i=q->rear;i>=q->front;i--)
      q->arr[i+1]=q->arr[i];
      q->arr[q->front]=x;
      q->rear++;
   }
   else if(flag==1)
   {
      q->arr[++q->rear]=x;
   }
   else
   {
      printf("\nInvalid flag value");
      return;
   }
}
void dequeue(dq *q,int flag)
{
   int i;
   /*front is initialized with zero, then rear=-1
   indicates underflow*/
   if(q->rear < q->front)
   {
      printf("\nQueue Underflow");
      exit(1);
   }
   if(flag==0)/*deletion at beginning*/
   {
      for(i=q->front;i<=q->rear;i++)
      q->arr[i]=q->arr[i+1];
      q->arr[q->rear]=0;
      q->rear--;
   }
   else if(flag==1)
   {  q->arr[q->rear--]=0;
   }
   else
   {  printf("\nInvalid flag value");
      return;
   }
}
void display(dq *q)
{
   int i;
   for(i=q->front;i<=q->rear;i++)
   printf("%d ",q->arr[i]);
}
void main()

{
   dq q;
   q.front=0;
   q.rear=-1;
   int ch,n;
   while(1)
   {
      printf("\nMenu-Double Ended Queue");
      printf("\n1. Enqueue – Begin");
      printf("\n2. Enqueue – End");
      printf("\n3. Dequeue – Begin");
      printf("\n4. Dequeue – End");
      printf("\n5. Display");
      printf("\n6. Exit");
      printf("\nEnter your choice: ");
      scanf("%d",&ch);
      switch(ch)
      {
         case 1:
         printf("\nEnter the number: ");
         scanf("%d",&n);
         enqueue(&q,n,0);
         break;
         case 2:
         printf("\nEnter the number:" );
         scanf("%d",&n);
         enqueue(&q,n,1);
         break;
         case 3:
         printf("\nDeleting element from beginning");
         dequeue(&q,0);
         break;
         case 4:
         printf("\nDeleting element from end");
         dequeue(&q,1);
         break;
         case 5:
         display(&q);
         break;
         case 6:
         exit(0);
         default:
         printf("\nInvalid Choice");
      }
   }
}
```

Output:



```
1. Enqueue - Begin
2. Enqueue - End
3. Dequeue - Begin
4. Dequeue - End
5. Display
6. Exit
Enter your choice: 2

Enter the number:40

Menu-Double Ended Queue
1. Enqueue - Begin
2. Enqueue - End
3. Dequeue - Begin
4. Dequeue - End
5. Display
6. Exit
Enter your choice: 5
20 10 30 40
Menu-Double Ended Queue
1. Enqueue - Begin
2. Enqueue - End
3. Dequeue - Begin
4. Dequeue - End
5. Display
6. Exit
Enter your choice: 3

Deleting element from beginning
Menu-Double Ended Queue
1. Enqueue - Begin
2. Enqueue - End
3. Dequeue - Begin
4. Dequeue - End
5. Display
```



```
6. Exit
Enter your choice: 5
10 30 40
Menu-Double Ended Queue
1. Enqueue - Begin
2. Enqueue - End
3. Dequeue - Begin
4. Dequeue - End
5. Display
6. Exit
Enter your choice: 4

Deleting element from end
Menu-Double Ended Queue
1. Enqueue - Begin
2. Enqueue - End
3. Dequeue - Begin
4. Dequeue - End
5. Display
6. Exit
Enter your choice: 5
10 30
Menu-Double Ended Queue
1. Enqueue - Begin
2. Enqueue - End
3. Dequeue - Begin
4. Dequeue - End
5. Display
6. Exit
Enter your choice: 6
sh-4.3$
```

(iv) Priority Queue

Consider a job sent to a line printer. Although these jobs are placed in a queue which is served by the printer on FIFO basis, this may not be a best practice always. Sometimes one job waiting in a queue might be particularly important, so that it might be required to allow the job to be run as soon as printer becomes available. Also, when the printer becomes available, there are several single page jobs in the queue and only one hundred-page job. Here, this might be reasonable to make the long job go last, even if it is not the last job submitted. (Unfortunately, most systems may not follow this, which can be particularly annoying at times.)

In a similar way, in multi-tasking and multi-user environment, an operating system scheduler must decide on which task or user to be allocated a processor. In general a process is allowed to be executed in time slots or time frames. A simple algorithm used for processing jobs is use of queue which process jobs of FIFO bases. Whenever a new job is arrived, it is placed at the end of the queue. The scheduler process the jobs on first come first serve bases from the queue until the queue is empty. This algorithm may not be appropriate since jobs which require short time slot may seem to take a long time because of the wait involved in running. Generally, it is important that jobs requiring short time slot finishes as fast as possible. Therefore, these jobs must have higher priority over jobs that have already been running. Furthermore, there may be some jobs that are not short which may be still very important and thus must also be considered on priority.

A priority queue is a data structure that allows at least the following two operations: insert, which does the obvious thing, and delete_min, which finds, returns and removes the minimum element in the queue. The insert operation is the same enqueue, and delete_min is the priority queue equivalent of the queues dequeue operation. The delete_minfunction also alters its input.

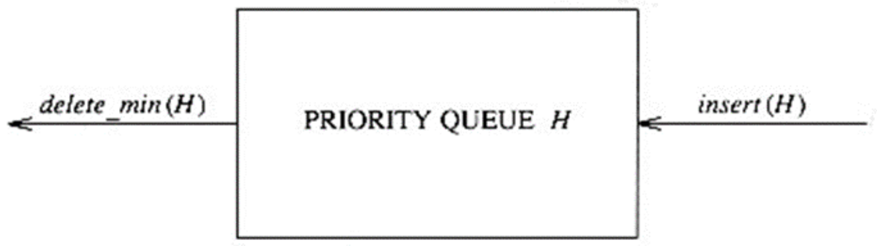


Figure 3.2.9: Basic Model of a Priority Queue

As with most data structures, at times it is possible to add other operations too, but these are extra operations are not part of the basic model depicted in Figure 3.2.8.

Besides operating systems, priority queues have many applications. They are used for external sorting. Priority queues are also important in the implementation of greedy algorithms, which operate by repeatedly finding a minimum.

Did you know?

Circular queue is very famous in computer networks because of its very own circular structure. The simplest application of circular queues for network engineers is in implementation of round robin algorithm which is used for token passing and also it is used in FIFO buffering systems.

Self-assessment Questions

4) A circular queue is implemented using an array of size 10. The array index starts with 0, front is 6, and rear is 9. The insertion of next element takes place at the array index.

a) 0	b) 7
c) 9	d) 10

5) If the MAX_SIZE is the size of the array used in the implementation of circular queue, array index start with 0, front point to the first element in the queue, and rear point to the last element in the queue. Which of the following condition specify that circular queue is EMPTY?

a) Front=rear=0	b) Front= rear=-1
c) Front=rear+1	d) Front= (rear+1)%MAX_SIZE

6) A normal queue, if implemented using an array of size MAX_SIZE, gets full when

a) Rear=MAX_SIZE-1	b) Front= (rear+1)mod MAX_SIZE
c) Front=rear+1	d) Rear=front

3.2.3 Operations on Queue

Similar to the operations performed on stacks, all such operations can be performed on queues too. The basic operations involve insertion (enqueue) and deletion (dequeue). The other supporting functions involve Qfill and Qempty. Other queuing operations involve initialising or defining a queue, using it and then completely erasing it from the computer's memory. This section covers all the functions on queues in detail.

- **enqueue()** – Insert a data element in a queue.
- **dequeue()** – remove a data element from the queue.

Additional functions are required to make above mentioned queue operation efficient. These are –

- **Qfull ()** – checks if the queue is full; returns Boolean
- **Qempty ()** – checks if the queue is empty; returns Boolean

While performing dequeues operations the data is accessed from front pointer and while performing enqueue operation, data is accessed from rear pointer.

(i) Insertion

As already discussed in the previous sections, insertion in queue is also called as enqueue operation. The following are the to be followed while performing enqueue operation –

- **Step 1** – Check if queue is full.
- **Step 2** – If queue is full, display an overflow error and exit.
- **Step 3** – If queue is not full, increment rear pointer to point next empty array space.
- **Step 4** – Add data element to the queue location, where rear is pointing.
- **Step 5** – return success.

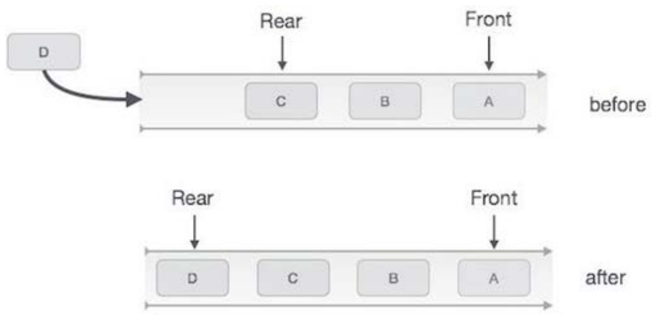


Figure 3.2.10: Enqueue Operation

Following is an algorithm for enqueue operation

```
procedure enqueue(data)
   if queue is full
          return overflow
   endif
          rear ← rear + 1
          queue[rear] ← data
   return true
end procedure
```

The above algorithm implemented in C programming is shown below:

```
int enqueue(int data)
   if(isfull())
      return 0;

   rear = rear + 1;
   queue[rear] = data;

   return 1;
end procedure
```

(ii) Deletion in Queue

Deletion operation from the queue is also called as dequeue operation. The following are the steps to be followed while performing dequeue operation –

- **Step 1 –** Check if queue is empty.
- **Step 2 –** If queue is empty, display an underflow error and exit.
- **Step 3 –** If queue is not empty, access data where front is pointing.
- **Step 4 –** Increment front pointer to point next available data element.
- **Step 5 –** return success.

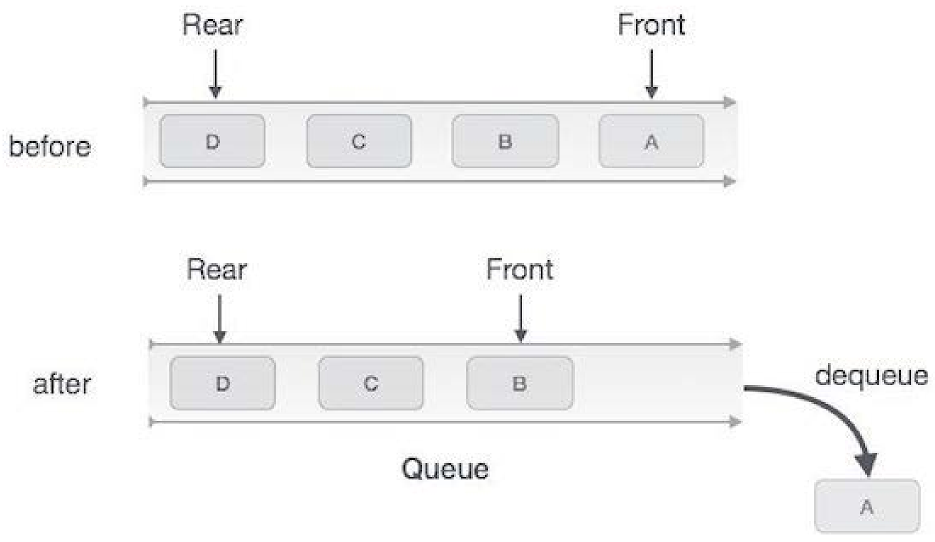


Figure 3.2.10: Dequeue Operation

Following is an algorithm used for performing dequeue operation

```
procedure dequeue
   if queue is empty
      return underflow
   end if

   data = queue[front]
   front ← front + 1

   return true
end procedure
```

The above algorithm implemented in C programming is shown below:

```
int dequeue() {

   if(isempty())
      return 0;

   int data = queue[front];
   front = front + 1;

   return data;
}
```

(iii) Qempty Operation

In order to delete an element from the queue first check weather Queue is empty or not. If queue is empty then do not delete an element from the queue. This condition is known as "Underflow".

If queue is not underflow then we can delete the element from queue. After deleting element from queue we must update the values of rear and front as per the position of elements in the queue.

Algorithm of Qempty() function –

```
begin procedure isempty

   if front is less than MIN  OR front is greater than rear
      return true
   else
      return false
   endif

end procedure
```

The above algorithm implemented in C programming is shown below:

```
bool isempty() {
   if(front < 0 || front > rear)
      return true;
   else
      return false;
}
```

(iv) Qfull Operation

Since we are using single dimensional array for implementation of queue, the best way to know if queue is full or not is to check if rear pointer has reached MAXSIZE-1; which means no space is available in array for additional elements and hence queue is full.

Algorithm of Qfull () function –

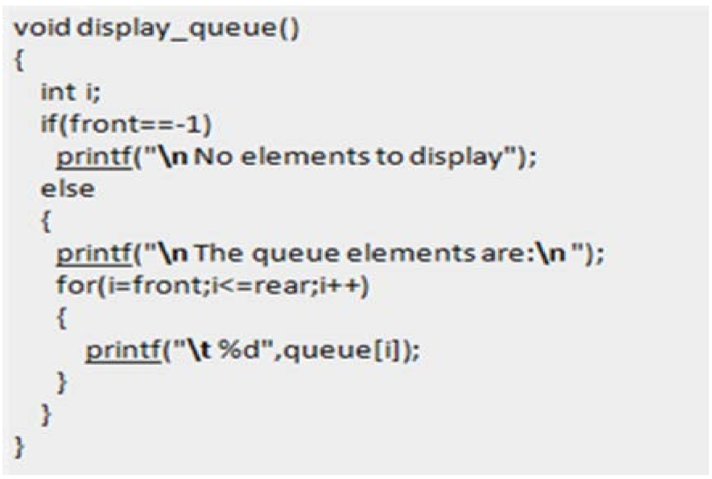
```
begin procedure isfull

   if rear equals to MAXSIZE
      return true
   else
      return false
   endif

end procedure
```

The above algorithm implemented in C programming is shown below:

```
bool isfull() {
   if(rear == MAXSIZE - 1)
      return true;
   else
      return false;
}
```

(v) Display Operation

Queue can be displayed by simply moving rear pointer till it reaches the front pointer. Only condition that should be considered is the Qempty condition and display "Queue Empty" message accordingly.



```
void display_queue()
{
  int i;
  if(front==-1)
   printf("\n No elements to display");
  else
  {
   printf("\n The queue elements are:\n ");
   for(i=front;i<=rear;i++)
   {
     printf("\t %d",queue[i]);
   }
  }
}
```

Self-assessment Questions

7) If the elements “A”, “B”, “C” and “D” are placed in a queue and are deleted one at a time, in what order will they be removed?

a) ABCD	b) DCBA
c) DCAB	d) BADC

8) Deletion operation is done using ___________in a queue.

a) Front	b) Rear
c) Top	d) Bottom

9) An array of size MAX_SIZE is used to implement a circular queue. Front, Rear, and count are tracked. Suppose front is 0 and rear is MAX_SIZE -1. How many elements are present in the queue?

a) Zero	b) One
c) MAX_SIZE-1	d) MAX_SIZE

3.2.4 Application of Queue

As per the very nature of queue, it can be used in all the applications requiring the use of fist come first serve property. Following are the some of the very common applications in computer science where use of queue makes it easy.

1. As already discussed in the previous chapters, queue plays a key role in scheduling for the computer resource sharing based applications. ***For example,*** simplest of the printer queue where printing jobs are added to the scheduling queue and printer serves the requests as per FIFO basis.

2. Similarly queues also play a very important role in CPU scheduling. All the requests for using processors are stored in queue by the CPU scheduler program. The requests are then serviced as per FIFO basis.

3. Another most common application of queue is routing calls in call centres. All the calls made by clients are stored in waiting queue and allotted to different executives who attend the calls. When all the executives are busy handling customers, the call which was made first out of all the other calls is given connected to executive as soon as he becomes available.

4. Another most important application of queue in computer system is interrupt handling. Since computer system is connected to many input and output devices. These devices keep sending requests to processor, by creating interrupts repeatedly. These interrupts are handled by interrupt handle program which put these interrupt in queue as and when they arrive. Then it service there interrupt as per the availability of CPU.

5. *M/M/1 queue*. The M/M/1 queue is a fundamental queueing model in operations research and probability theory. Tasks arrive according to a Poisson process at a certain rate λ. This means that λ customers arrive per hour. More specifically, the arrivals follow an exponential distribution with mean 1 / λ: the probability of k arrivals between time 0 and t is (λ t)^k e^(-λ t) / k!. Tasks are serviced in FIFO order according to a Poisson process with rate μ. The two M's standard for Markov: it means that the system is memoryless: the time between arrivals is independent, and the time between departures is independent.

Self-assessment Questions

10) Which data structure allows deleting data elements from front and inserting at rear?

a) Stacks	b) Queues
c) Dequeues	d) Binary search tree

11) The push and enqueue operations are essentially the same operations, push is used for Stacks and enqueue is used for Queues.

a) True	b) False

12) In order to input a list of values and output them in order, you could use a Queue. In order to input a list of values and output them in opposite order, you could use a Stack.

a) True	b) False

Summary

- Similar to stacks, Queues are data structure usually used to simplify certain programming operations.
- In these data structures, only one data item can be immediately accessed.
- A queue, in general, allows access to the first item that was inserted.
- The important queue operations are inserting an item at the rear of the queue and removing the item from the front of the queue.
- A queue can be implemented as a circular queue, which is based on an array in which the indices wrap around from the end of the array to the beginning.
- A priority queue allows access to the smallest (or sometimes the largest) item in the queue.
- The important priority queue operations are inserting an item in sorted order and removing the item with the smallest key.
- Few important operations performed on the queue are insertion which is also called enqueue, deletion also called dequeues, Qempty which check if queue is empty, Qfull which check if queue is full and display which is used to display all the elements in the queue.
- Queues finds its applications in implementing job scheduling algorithms, page replacement algorithms, interrupt handling mechanisms etc. in the design of operating systems.

Terminal Questions

1. Explain the basic operations of queue.
2. Discuss the functioning of circular Queue?
3. Mention the limitation of linear queue with a suitable example
4. Discuss applications of Queue

Answer Keys

Self-assessment Questions	
Question No.	**Answer**
1	d
2	b
3	b
4	a
5	b
6	a
7	a
8	a
9	d
10	b
11	a
12	a

Activity

Activity Type: Offline **Duration:** 15 Minutes

Description:

Fill in the following table to give the running times of the priority queue operations for the given two implementations using O () notation. You should assume that the implementation is reasonably well done, ***For example***, not performing expensive computations when a value can be stored in an instance variable and be used as needed.

A priority queue is a data structure that supports storing a set of values, each of which has an associated key. Each key-value pair is an entry in the priority queue. The basic operations on a priority queue are:

- insert(k, v)

– insert value v with key k into the priority queue

- removeMin()

– return and remove from the priority queue the entry with the smallest key

Other operations on the priority queue include size (), which returns the number of entries in the queue and is Empty () which returns true if the queue is empty and false otherwise.

Two simple implementations of a priority queue are an unsorted list, where new entries are added at the end of the list, and a sorted list, where entries in the list are sorted by their key values.

Operation	Unsorted list	Sorted list
Size, isempty()		
Insert		
Removemin()		

Bibliography

e-Reference

- bowdoin.edu, (2016). *Computer Science 210: Data Structures.* Retrieved on 19 April 2016, from http://www.bowdoin.edu/~ltoma/teaching/cs210/fall10/Slides/StacksAndQueues.pdf

External Resources

- Kruse, R. (2006). *Data Structures and program designing using 'C'* (2nd ed.). Pearson Education.
- Srivastava, S. K., & Srivastava, D. (2004). *Data Structures through C in Depth* (2nd ed.). BPB Publications.
- Weiss, M. A. (2001). *Data Structures and Algorithm Analysis in C* (2nd ed.). Pearson Education.

Video Links

Topic	Link
Circular queue	https://www.youtube.com/watch?v=g9su-lnW2Ks
Priority queue	https://www.youtube.com/watch?v=gJc-J7K_P_w
Double ended queue	https://www.youtube.com/watch?v=4xLh68qokxQ

Notes: