UNIT 5 QUEUES

Structure **Page Nos.**

5.0 INTRODUCTION

Queue is a linear data structure used in various applications of computer science. Like people stand in a queue to get a particular service, various processes will wait in a queue for their turn to avail a service. In computer science, it is also called a FIFO (first in first out) list. In this chapter, we will study about various types of queues.

5.1 OBJECTIVES

After going through this unit, you should be able to

- define the queue as an abstract data type;
- understand the terminology of various types of queues such as simple queues, multiple queues, circular queues and dequeues, and
- get an idea about the implementation of different types of queues using arrays and linked lists.

5.2 ABSTRACT DATA TYPE-QUEUE

An important aspect of Abstract Data Types is that they describe the properties of a data structure without specifying the details of its implementation. The properties can be implemented independent of any implementation in any programming language.

Queue is a collection of elements, or items, for which the following operations are defined:

createQueue(Q) : creates an empty queue Q;
isEmpty(Q): is a boolean type predicate that returns ``true'' if Q exists and is empty, and returns ``false'' otherwise;
addQueue(Q,item) adds the given item to the queue Q; and
deleteQueue (Q, item) : delete an item from the queue Q;
next(Q) removes the least recently added item that remains in the queue Q, and returns it as the value of the function;

isEmpty (createQueue(Q)) : is always true, and
deleteQueue(createQueue(Q)) : error

The primitive isEmpty(Q) is required to know whether the queue is empty or not, because calling next on an empty queue should cause an error. Like stack, the situation may be such when the queue is "full" in the case of a finite queue. But we avoid defining this here as it would depend on the actual length of the Queue defined in a specific problem.

The word "queue" is like the queue of customers at a counter for any service, in which customers are dealt with in the order in which they arrive i.e. first in first out (FIFO) order. In most cases, the first customer in the queue is the first to be served.

As pointed out earlier, Abstract Data Types describe the properties of a structure without specifying an implementation in any way. Thus, an algorithm which works with a "queue" data structure will work wherever it is implemented. Different implementations are usually of different efficiencies.

5.3 IMPLEMENTATION OF QUEUE

A physical analogy for a queue is a line at a booking counter. At a booking counter, customers go to the *rear* (end) of the line and customers are attended to various services from the *front* of the line. Unlike stack, customers are added at the rear end and deleted from the front end in a queue (FIFO).

An example of the queue in computer science is print jobs scheduled for printers. These jobs are maintained in a queue. The job fired for the printer first gets printed first. Same is the scenario for job scheduling in the CPU of computer.

Like a stack, a queue also (usually) holds data elements of the same type. We usually graphically display a queue horizontally. *Figure 5.1* depicts a queue of 5 characters.

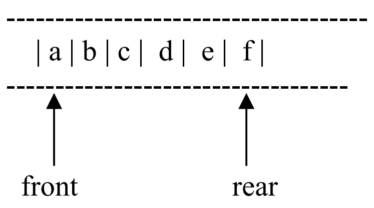

Figure 5.1: A queue of characters

The rule followed in a queue is that elements are added at the *rear* and come off of the *front* of the queue. After the addition of an element to the above queue, the position of rear pointer changes as shown below. Now the *rear* is pointing to the new element 'g' added at the rear of the queue(refer to *Figure 5.2*).

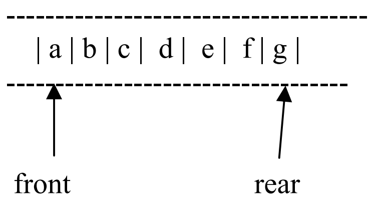

Figure 5.2: Queue of figure 5.1 after addition of new element

After the removal of element 'a' from the front, the queue changes to the following with the *front* pointer pointing to 'b' (refer to *Figure 5.3*).

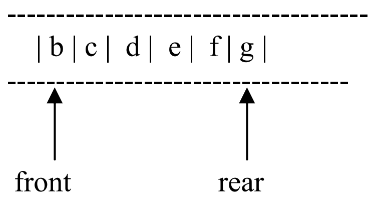

Figure 5.3: Queue of figure 5.2 after deletion of an element

Algorithm for addition of an element to the queue

```
Step 1: Create a new element to be added
Step 2: If the queue is empty, then go to step 3, else perform step 4
Step 3: Make the front and rear point this element
Step 4: Add the element at the end of the queue and shift the rear pointer to the newly
        added element.
```

Algorithm for deletion of an element from the queue

```
Step 1: Check for Queue empty condition. If empty, then go to step 2, else go to step 3
Step 2: Message "Queue Empty"
Step 3: Delete the element from the front of the queue. If it is the last element in the
        queue, then perform step a else step b
        a) make front and rear point to null
        b) shift the front pointer ahead to point to the next element in the queue
```

5.3 1 Array implementation of a queue

As the stack is a list of elements, the queue is also a list of elements. The stack and the queue differ only in the position where the elements can be added or deleted. Like other liner data structures, queues can also be implemented using arrays. Program 5.1 lists the implementation of a queue using arrays.

```
#include "stdio.h"
#define QUEUE_LENGTH 50
struct queue
{    int element[QUEUE_LENGTH];
     int front, rear, choice,x,y;
}

struct queue q;

main()
{
int choice,x;
printf ("enter 1 for add and 2 to remove element front the queue")
printf("Enter your choice")
scanf("%d",&choice);
switch (choice)

  {

case 1 :
 printf ("Enter element to be added :");
scanf("%d",&x);
add(&q,x);
break;

case 2 :
delete();
break;

  }

}
add(y)
{
++q.rear;
if (q.rear < QUEUE_LENGTH)
   q.element[q.rear] = y;
else
 printf("Queue overflow")
}

delete()
{

if q.front > q.rear printf("Queue empty");
else{
  x = q.element[q.front];
 q.front++;
 }
retrun x;
}
```

Program 5.1: Array implementation of a Queue

5.3.2 Linked List Implementation of a queue

The basic element of a linked list is a "record" structure of at least two fields. The object that holds the data and refers to the next element in the list is called a node (refer to *Figure 5.4*).

Data	Ptrnext

Figure 5.4: Structure of a node

The *data* component may contain data of any type. *Ptrnext* is a reference to the next element in the queue structure. *Figure 5.5* depicts the linked list representation of a queue.

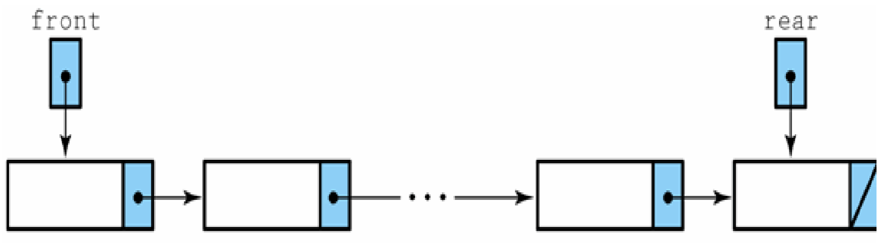

Figure 5.5: A linked list representation of a Queue

Program 5.2 gives the program segment for the addition of an element to the queue.

Program 5.3 gives the program segment for the deletion of an element from the queue.

```
add(int value)
{
struct queue *new;
new = (struct queue*)malloc(sizeof(queue));
new->value = value;
new->next = NULL;
if (front == NULL)
{
  queueptr = new;
  front = rear = queueptr
}
else
{
rear->next = new;
rear=new;
}
}
```

Program 5.2: Program segment for addition of an element to the queue

```
delete()
{
int delvalue = 0;
if (front == NULL) printf("Queue Empty");
{
  delvalue = front->value;
 if (front->next==NULL)
{
free(front);
queueptr=front=rear=NULL;
}
else
{
front=front->next;
free(queueptr);
queueptr=front;

}
}
}
```

Program 5.3: Program segment for deletion of an element from the queue

☞ Check Your Progress 1

1) The queue is a data structure where addition takes place at _________ and deletion takes place at _____________.

2) The queue is also known as ________ list.

3) Compare the array and linked list representations of a queue. Explain your answer.

5.4 IMPLEMENTATION OF MULTIPLE QUEUES

So far, we have seen the representation of a single queue, but many practical applications in computer science require several queues. Multiqueue is a data structure where multiple queues are maintained. This type of data structures are used for process scheduling. We may use one dimensional array or multidimensional array to represent a multiple queue.

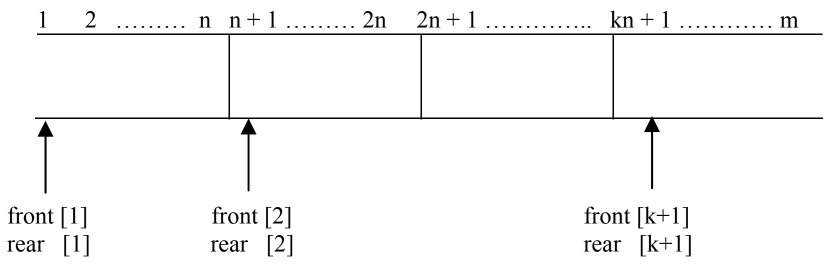

Figure 5.6: Multiple queues in an array

A multiqueue implementation using a single dimensional array with *m* elements is depicted in *Figure 5.6*. Each queue has *n* elements which are mapped to a liner array of *m* elements.

Array Implementation of a multiqueue

Program 5.4 gives the program segment using arrays for the addition of an element to a queue in the multiqueue.

```
addmq(i,x)  /* Add x to queue i */
{
int i,x;
++rear[i];
if ( rear[i] == front[i+1])
 printf("Queue is full");
else
{
rear[i] = rear[i]+1;
mqueue[rear[i]] = x;
}
}
```

Program 5.4: Program segment for the addition of an element to the queue

Program 5.5 gives the program segment for the deletion of an element from the queue.

```
delmq(i)   /* Delete an element from queue i */
{
int i,x;
if ( front[i] == rear[i])
 printf("Queue is empty");
{
x = mqueue[front[i]];
front[i] = front[i]-1 ;
return x;
}
}
```

Program 5.5: Program segment for the deletion of an element from the queue

5.5 IMPLEMENTATION OF CIRCULAR QUEUES

One of the major problems with the linear queue is the lack of proper utilisation of space. Suppose that the queue can store 100 elements and the entire queue is full. So, it means that the queue is holding 100 elements. In case, some of the elements at the front are deleted, the element at the last position in the queue continues to be at the same position and there is no efficient way to find out that the queue is not full. In this way, space utilisation in the case of linear queues is not efficient. This problem is arising due to the representation of the queue.

The alternative representation is to depict the queue as circular. In case, we are representing the queue using arrays, then, a queue with *n* elements starts from index *0* and ends at *n-1*.So, clearly , the first element in the queue will be at index 0 and the last element will be at n-1 when all the positions between index 0 and n-1(both inclusive) are filled. Under such circumstances, front will point to 0 and rear will point to n-1. However, when a new element is to be added and if the rear is pointing to n-1, then, it needs to be checked if the position at index 0 is free. If yes, then the element can be added to that position and rear can be adjusted accordingly. In this way, the utilisation of space is increased in the case of a circular queue.

In a circular queue, front will point to one position less to the first element anti-clock wise. So, if the first element is at position 4 in the array, then the front will point to position 3. When the circular queue is created, then both front and rear point to index 1. Also, we can conclude that the circular queue is empty in case both front and rear point to the same index. *Figure 5.7* depicts a circular queue.

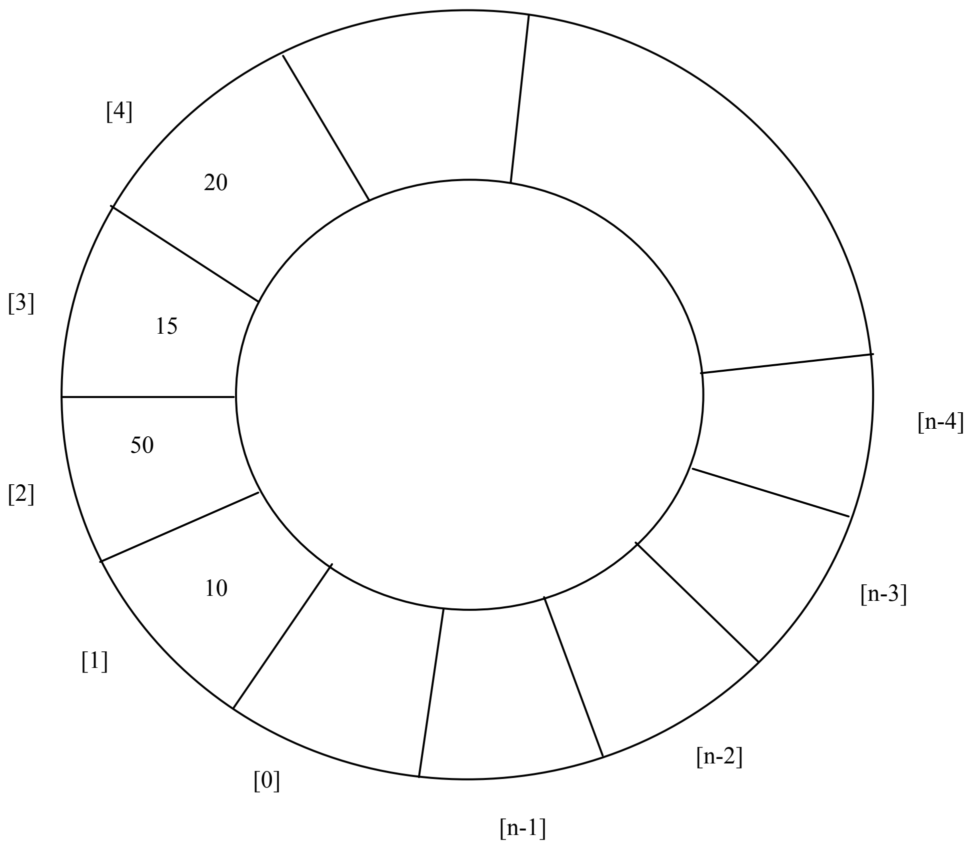

Figure 5.7 : A circular queue (Front = 0, Rear = 4)

Algorithm for Addition of an element to the circular queue:

```
Step-1: If "rear" of the queue is pointing to the last position then go to step-2 or else
        Step-3
Step-2: make the "rear" value as 0
Step-3: increment the "rear" value by one
Step-4: a. if the "front" points where "rear" is pointing and the queue holds a not
           NULL value for it, then its a "queue overflow" state, so quit; else go to
           step-b
        b. add the new value for the queue position pointed by the "rear"
```

Algorithm for deletion of an element from the circular queue:

```
Step-1: If the queue is empty then say "queue is empty" and quit; else continue
Step-2: Delete the "front" element
Step-3: If the "front" is pointing to the last position of the queue then go to step-4 else
        go to step-5
Step-4: Make the "front" point to the first position in the queue and quit
Step-5: Increment the "front" position by one
```

5.5.1 Array implementation of a circular queue

A circular queue can be implemented using arrays or linked lists. Program 5.6 gives the array implementation of a circular queue.

```
#include "stdio.h"
void add(int);
void deleteelement(void);
int max=10;          /*the maximum limit for queue has been set*/
static int queue[10];
int front=0, rear=-1;  /*queue is initially empty*/
void main()
{
int choice,x;
printf ("enter 1 for addition and 2 to remove element front the queue and 3 for exit");
printf("Enter your choice");
scanf("%d",&choice);
switch (choice)
{
case 1 :
printf ("Enter the element to be added :");
scanf("%d",&x);
add(x);
break;
case 2 :
deleteelement();
break;
 }
}
void add(int y)
{
 if(rear == max-1)
   rear = 0;
   else
   rear = rear + 1;
   if( front == rear && queue[front] != NULL)
   printf("Queue Overflow");
   else
   queue[rear] = y;
}

void deleteelement()
{
int deleted_front = 0;
 if (front == NULL)
     printf("Error - Queue empty");
 else
    {
      deleted_front = queue[front];
      queue[front] = NULL;
       if (front == max-1)
       front = 0;
      else
      front = front + 1;
  }
}
```

Program 5.6: Array implementation of a Circular queue

5.5.2 Linked list implementation of a circular queue

Link list representation of a circular queue is more efficient as it uses space more efficiently, of course with the extra cost of storing the pointers. Program 5.7 gives the linked list representation of a circular queue.

```
#include “stdio.h”
 struct cq
{     int value;
      int *next;
 };

typedef struct  cq *cqptr
cqptr p, *front, *rear;

main()
{
int choice,x;

/*  Initialise the circular queue */
cqptr =  front = rear =  NULL;

printf (“Enter 1 for addition and 2 to delete element from the queue”)
printf(“Enter your choice”)
scanf(“%d”,&choice);
switch (choice)

{

case 1 :
            printf (“Enter the element to be added :”);
            scanf(“%d”,&x);
            add(&q,x);
            break;

case 2 :
            delete();
            break;

  }
}

/************ Add element *****************/

add(int value)
{
  struct cq *new;
  new = (struct cq*)malloc(sizeof(queue));
  new->value = value
  new->next = NULL;

  if (front == NULL)
  {
   cqptr = new;
   front = rear = queueptr;
  }
  else
     {
    rear->next = new;
    rear=new;
   }
}

/* *************** delete element ***********/
delete()
{
int delvalue = 0;
if (front == NULL)
{ printf("Queue is empty");
  delvalue = front->value;
  if (front->next==NULL)
 {
  free(front);
  queueptr = front = rear = NULL;
}
else
{
front=front->next;
free(queueptr);
queueptr = front;

}
}
}
```

Program 5.7 : Linked list implementation of a Circular queue

5.6 IMPLEMENTATION OF DEQUEUE

Dequeue (a double ended queue) is an abstract data type similar to queue, where addition and deletion of elements are allowed at both the ends. Like a linear queue and a circular queue, a dequeue can also be implemented using arrays or linked lists.

5.6.1 Array implementation of a dequeue

If a Dequeue is implemented using arrays, then it will suffer with the same problems that a linear queue had suffered. Program 5.8 gives the array implementation of a Dequeue.

```
#include "stdio.h"
#define QUEUE_LENGTH 10;
int dq[QUEUE_LENGTH];
int front, rear, choice,x,y;
main()
{
   int choice,x;
   front = rear = -1; /* initialize the front and rear to null i.e empty queue */

  printf ("enter 1 for addition and  2 to remove element from the front of the queue");
  printf ("enter 3 for addition and  4 to remove element from the rear of the queue");
  printf("Enter your choice");
  scanf("%d",&choice);
  switch (choice)
  {
    case 1:
              printf ("Enter element to be added :");
              scanf("%d",&x);
              add_front(x);
              break;
    case 2:
             delete_front();
             break;
    case 3:
             printf ("Enter the element to be added :");
             scanf("%d ",&x);
             add_rear(x);
             break;
   case 4:
             delete_rear();
             break;
  }
}
/**************** Add at the front ***************/
add_front(int y)
{
if (front == 0)
{
    printf("Element can not be added at the front");
    return;
else
    {
      front = front - 1;
      dq[front] = y;
       if (front == -1 ) front = 0;
     }
 }
/**************** Delete from the front ***************/
delete_front()
{
  if front == -1
  printf("Queue empty");
  else
        return dq[front];
        if (front = = rear)
        front = rear = -1
        else
             front = front + 1;
}
/**************** Add at the rear ***************/
add_rear(int y)
if (front == QUEUE_LENGTH -1 )
{
 printf("Element can not be added at the rear ")
 return;
else
{
  rear  = rear + 1;
  dq[rear] = y;
  if (rear = = -1 )
  rear = 0;
}
}

/**************** Delete at the rear ***************/
delete_rear()
{
  if rear == -1
  printf("deletion is not possible from rear");
  else
 {
        if (front = = rear)
       front = rear = -1
       else
            { rear = rear – 1;
              return dq[rear];
            }
}
}
```

Program 5.8: Array implementation of a Dequeue

5.6.2 Linked list implementation of a dequeue

Double ended queues are implemented with doubly linked lists.

A doubly link list can traverse in both the directions as it has two pointers namely left and right. The right pointer points to the next node on the right where as the left pointer points to the previous node on the left. Program 5.9 gives the linked list implementation of a Dequeue.

```
# include "stdio.h"
#define NULL 0
struct dq {
               int info;
               int *left;
               int *right;
             };
typedef struct dq *dqptr;
dqptr p, tp;
dqptr head;
dqptr tail;
main()
{
        int choice, I, x;
       dqptr n;
       dqptr getnode();
        printf("\n Enter 1: Start  2 : Add at Front 3 : Add at Rear 4: Delete at Front 5:
Delete at Back");
while (1)
{
        printf("\n 1: Start  2 : Add at Front 3 : Add at Back 4: Delete at Front 5: Delete
at Back 6 : exit");
        scanf("%d", &choice);
        switch (choice)
       {
         case 1:
                   create_list();
                   break;
         case 2:
                   eq_front();
                   break;
         case 3:
                  eq_back();
                  break;
        case 4:
                 dq_front();
                 break;
        case 5:
                 dq_back();
                 break;
        case 6 :
                 exit(6);
    }
 }
}

create_list()
{
  int I, x;
  dqptr t;
  p = getnode();
  tp = p;
  p->left = getnode();
  p->info = 10;
  p_right = getnode();
  return;
}

dqptr getnode()
{
   p = (dqptr) malloc(sizeof(struct dq));
   return p;
}

dq_empty(dq q)
{
  return q->head = = NULL;
}
eq_front(dq q, void *info)
{
  if (dq_empty(q))
     q->head = q->tail = dcons(info, NULL, NULL);
else
{
    q-> head -> left =dcons(info, NULL, NULL);
    q->head -> left ->right = q->head;
    q ->head  = q->head ->left;
}
}

eq_back(dq q, void *info)
{
  if (dq_empty(q))
     q->head = q->tail = dcons(info, NULL, NULL)
  else
{
  q-> tail -> right =dcons(info, NULL, NULL);
  q->tail -> right -> left = q->tail;
  q ->tail  = q->tail ->right;
}
}
  dq_front(dq q)
{
   if dq is not empty
{
dq tp = q-> head;
void *info = tp -> info;
q ->head = q->head-> right;
free(tp);
if (q->head = = NULL)
   q -> tail = NULL;
else
 q -> head -> left = NULL;
return info;
}
}

dq_back(dq q)
{
   if (q!=NULL)
   {
   dq tp = q-> tail;
   *info = tp -> info;
   q ->tail = q->tail-> left;
   free(tp);
   if (q->tail = = NULL)
                q -> head = NULL;
  else
               q -> tail -> right = NULL;
   return info;
   }
}
```

Program 5.9 : Linked list implementation of a Dequeue

☞ **Check Your Progress 2**

1) _________ allows elements to be added and deleted at the front as well as at the rear.
2) It is not possible to implement multiple queues in an Array.
(True/False)
3) The index of a circular queue starts at __________.

5.7 SUMMARY

In this unit, we discussed the data structure *Queue*. It had two ends. One is front from where the elements can be deleted and the other if rear to where the elements can be added. A queue can be implemented using Arrays or Linked lists. Each representation is having it's own advantages and disadvantages. The problems with arrays are that they are limited in space. Hence, the queue is having a limited capacity. If queues are implemented using linked lists, then this problem is solved. Now, there is no limit on the capacity of the queue. The only overhead is the memory occupied by the pointers.

There are a number of variants of the queues. Normally, queues mean circular queues. Apart from linear queues, we also discussed circular queues in this unit. A special type of queue called Dequeue was also discussed in this unit. Dequeues permit elements to be added or deleted at either of the rear or front. We also discussed the array and linked list implementations of Dequeue.

5.8 SOLUTIONS/ANSWERS

Check Your Progress 1

1. rear , front
2. First in First out (FIFO) list

Check Your Progress 2

1. Dequeue
2. False
3. 0

5.9 FURTHER READINGS

Reference Books

1. *Data Structures using C* by Aaron M.Tanenbaum, Yedidyah Langsam, Moshe J.Augenstein , PHI publications
2. *Algorithms+Data Structures = Programs* by Niklaus Wirth, PHI publications

Reference Websites

http://ciips.ee.uwa.edu.au/~morris/Year2/PLDS210/queues.html
http://www.cs.toronto.edu/~wayne/libwayne/libwayne.html