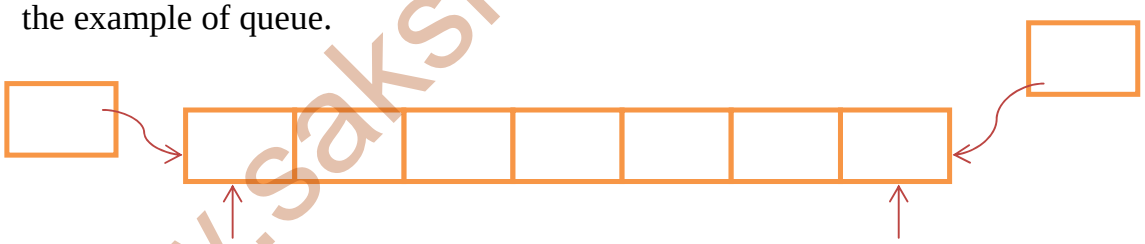
QUEUE

What is a Queue?

A queue is a data structure similar to stack; the only difference is in the order of retrieving data. A queue is a linear, sequential list of items that are accessed in the order, **First in First Out (FIFO) i.e.,** the first item inserted in a queue is also the first one to be accessed, the second item inserted in a queue is also the second one to be accessed. A queue is very similar to the way we queue up at train reservation counter or film tickets book counter etc.

Definition: A queue is an order list in which insertions are done at one end and deletions are done at other end. Hence it is called as **First In First Out or Last In First Out.**

Similar to stacks, two special names are given to operations that can be performed on queue. Inserting an element into queue is called as ***Enqueue*** (Write) and removing an element from queue is called ***Dequeue*** (Read). Trying to enqueue an element to full queue is called ***overflow*** and trying dequeue an element from empty queue is called ***underflow.*** The below example illustrates the example of queue.

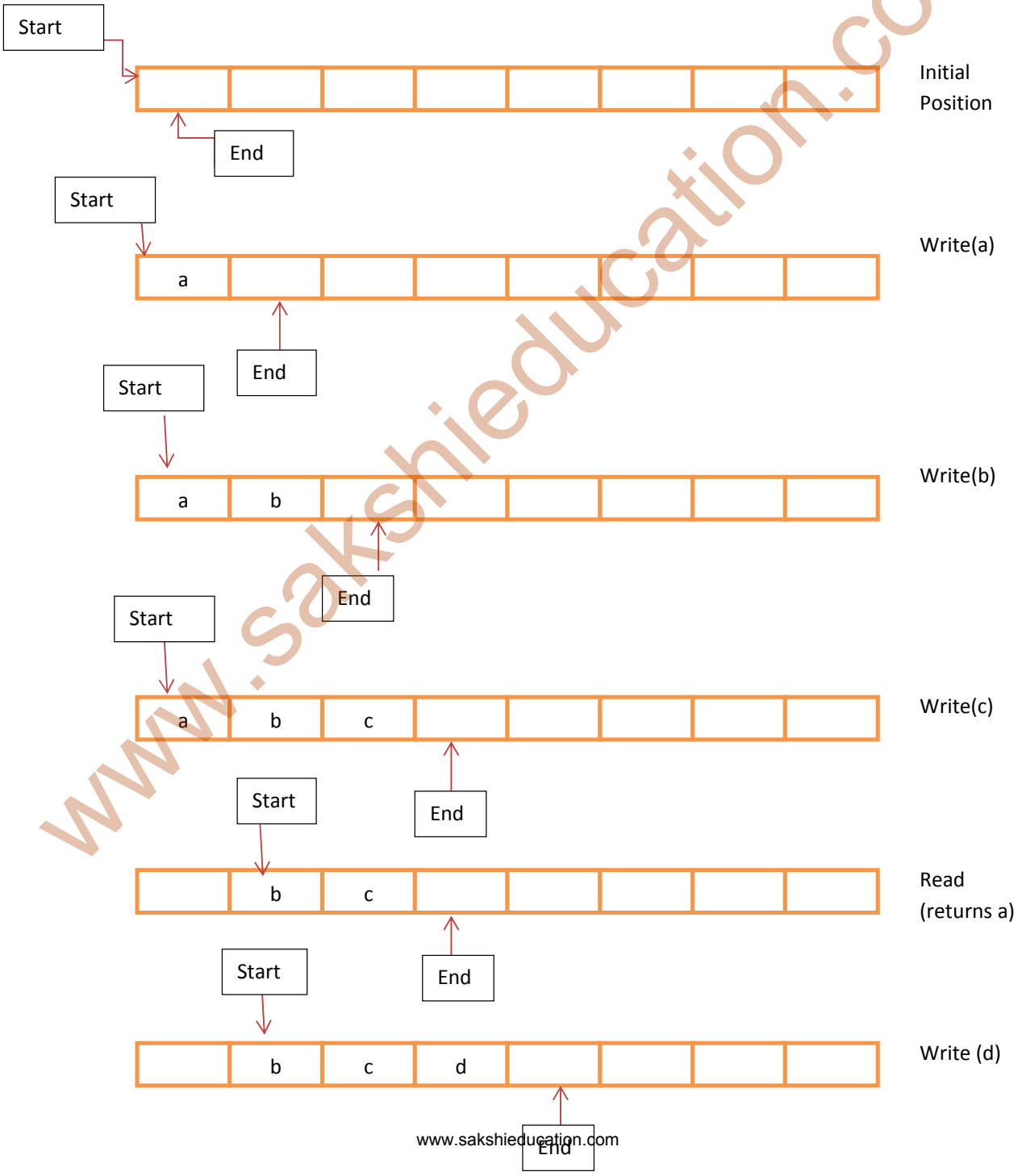


Operation	Contents of the Queue
write(a)	a
write(b)	a b
write(c)	a b c
read(returns a)	b c
write(d)	b c d
read(returns b)	c d
read(returns c)	d
write(e)	d e

Figure1: Queue Operations

As the figure depicts, the operations on a queue are FIFO. Also note that, when one item is read from the queue it is destroyed automatically unlike other data structures such as linked lists.

In order to implement the write and read operations of a queue, two pointers start and end are required. One pointer (start) points at the current start of the queue, while the other pointer (end) points at the current end of the queue. The insertions and deletions from queue, as shown in table, would have the effects on the pointers as shown below figure.



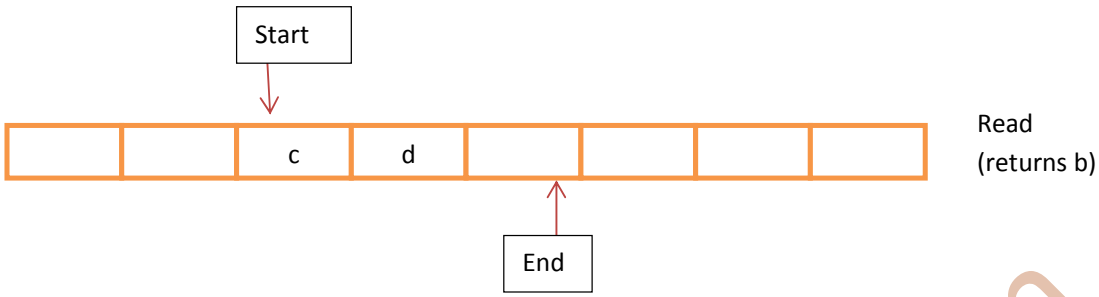


Figure2: pointer movements because of queue operations

Queue Operations

voidwrite(int data): Inserts an element at the end of the queue.

void read(): Removes and returns the element at the front of the queue.

void display(): Display the elements in the queue.

Implementation

Similar to stacks there are many ways to implement queue operations and below are the common methods.

- Simple array based implementation
- Circular array based implementation
- Linked lists implementation

1. Simple array based implementation

As discussed above, in order to implement the write and read operations of a queue, two pointers start and end are required. One pointer (start) points at the current start of the queue. The other pointer (end) points at the current end of the queue.

C Program:

```
/* Implement queue using an array*/
#define MAX 5 //define any number to limit your queue size
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>

int queue[MAX];
intstart, end;//indicates the front and tail of the queue
```

```
//insert an item to the queue
//While inserting an element we are adding element at end and incrementing the
end.
void write(int data)
{
        // before inserting an element into queue it checks queue full or not
        if(end>=MAX){
                printf("Queue overflow. We cant add more items\n");
                return;
        }
        else{

                queue[end]=data;
                end++;
        }
}
//Read and delete the first item from the queue

void read(){
        //before pop out an element from queue it checks for queue empty
        if(start==end){
                printf("Queue is underflow.");
                return;
        }
        printf("Item deleted from queue is:%d",queue[start]);
        start++;
}
// Display the elements in the queue
void display()
{
int i;
        //before displaying the elements it checks for queue empty or not
if(start==end){
                printf("\nQueue is empty.");
                return;
    }
else{
                printf("\nThe Queue elements are:");
                for(i=start;i<end;i++)
                {
                printf("%d",queue[i]);
                printf(" ");
```

```
                }
        }
}
// main starts here
void main()
{
intchoice,value;
        start=end=0;
printf("1.Insert");
printf("\n2.Delete");
printf("\n3.display");
        printf("\n4.Quit");
while(1)
    {
printf("\nEnter your choice for the operation: ");
scanf("%d",&choice);
switch(choice)
        {
        case 1:printf("\n Enter value to insert in to queue: ");
        scanf("%d",&value);
                write(value);
                display();
                break;
case 2: read();
                display();
                break;

case 3: display();
                break;
         case 4:exit(0);

       }//switch ends here

    }//while ends here

}//main end here
```

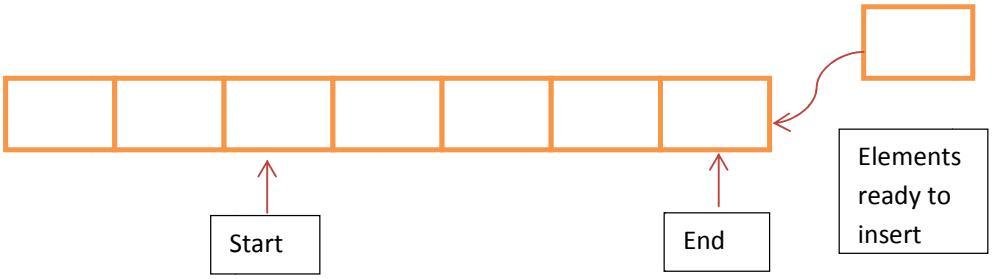
Output:

```
1.Insert
2.Delete
3.display
4.Quit
Enter your choice for the operation: 1

Enter value to insert in to queue: 10

The Queue elements are:10
Enter your choice for the operation: 1

Enter value to insert in to queue: 20

The Queue elements are:10 20
Enter your choice for the operation: 1

Enter value to insert in to queue: 30

The Queue elements are:10 20 30
Enter your choice for the operation: 1

Enter value to insert in to queue: 40

The Queue elements are:10 20 30 40
Enter your choice for the operation: 1

Enter value to insert in to queue: 50

The Queue elements are:10 20 30 40 50
Enter your choice for the operation: 1

Enter value to insert in to queue: 60
Queue overflow. we cant add more items

The Queue elements are:10 20 30 40 50
Enter your choice for the operation: 2
Item deleted from queue is:10
The Queue elements are:20 30 40 50
Enter your choice for the operation: 3

The Queue elements are:20 30 40 50
Enter your choice for the operation: 4
```

2. Circular array based implementation:

Why circular arrays

As in the simple array based implementation, after the insertions and deletions it is easy to get the situation as shown below.



As shown in figure the initial slots of the array are getting wasted. So, simple array based implementation for queue is not efficient. To overcome this problem we assume array as circular arrays. That means, we treat last element and first elements are contiguous as shown below.

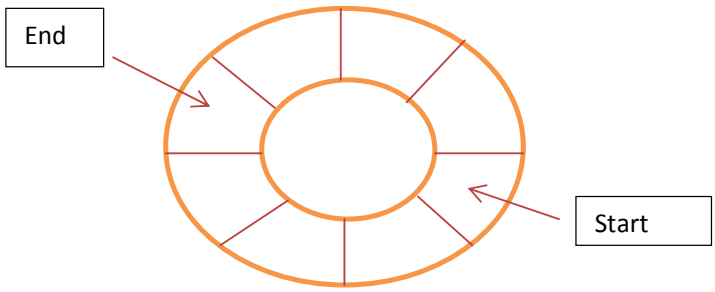


Figure: Circular Queue

C Program:

```
/**** Program to Implement Queue using circular Array ****/
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
#define SIZE 5

int queue[SIZE], end=-1, start=-1, item;
//Inserting an element into queue
void write()
{
        if((start==0 && end==SIZE-1) || (start==end+1))
                printf("\nQueue is full.");
        else{
                printf("\nPlease Enter the data to insert into Queue: ");
                scanf("%d", &item);
                if(end==SIZE-1)
                        end=0;
                else
                        end++;
                queue[end]=item;
        }
        if(start==-1)
                start=0;
```

```
}
//Reading an element from queue
void read()
{
        if(start==-1)
                printf("Circular Queue is Empty\n");

        else{
                printf("\nItem Deleted: %d\n", queue[start]);
                if(start==end)
                        start=end=-1;
                else{
                        if(start==SIZE-1)
                                start=0;
                        else
                                start++;
                }
        }

}
//Display the all elements in queue

void display(){
        int i;
        if(start==-1)
                printf("Circular Queue is Empty\n");

        else{
                if(end < start){
                        for(i=start;i<=SIZE-1;i++){
                                printf("%d",queue[i]);
                                printf("   ");
                        }
                        for(i=0;i<=end;i++){
                                printf("%d",queue[i]);
                                printf("   ");
                        }
                }
                else{
                        for(i=start;i<=end;i++){
                                printf("%d",queue[i]);
                                printf("   ");
```

```
                }
                        printf("\n");
                }
        }
}
void main()
{
        intch;
        printf("\n1.)write\n2.)read\n3.)Display\n4.)Exit\n");
        while(1){

                printf("\nEnter your choice: ");
                scanf("%d", &ch);

                switch(ch){
                        case 1:
                                        write();
                                        display();
                                        break;
                        case 2:
                                        read();
                                        display();
                                        break;
                        case 3:
                                        display();
                                        break;
                case 4:
                                        exit(0);

                        default:
                                printf("\n Invalid choice. Please enter correct
choice...\n");
                }
                getch();
        }
}
```

Output:

```
1.>write
2.>read
3.>Display
4.>Exit

Enter your choice: 1

Please Enter the data to insert into Queue: 10
10

Enter your choice: 1

Please Enter the data to insert into Queue: 20
10   20

Enter your choice: 1

Please Enter the data to insert into Queue: 30
10   20   30

Enter your choice: 2

Item Deleted: 10
20   30

Enter your choice: 1

Please Enter the data to insert into Queue: 10
20   30   10

Enter your choice: 3
20   30   10

Enter your choice: 4
```

Performance

Let n be the number of elements in the queue.

Space Complexity (For n Enqueue operations)	O(n)
Time Complexity for write()	O(1)
Time Complexity for read()	O(1)
Time Complexity for display()	O(1)

Limitations

The maximum size of the queue must be defined in advance and can't be changed later.

2. Linked List Implementation

The other way of implementing queue is by using linked lists similar to stack. Create two pointers front and rear points to front and end of nodes in list.

C program:

```
#include<stdio.h>
#include<stdlib.h>
//Global declaration
struct node
{
```

```
int data;
struct node *next;
}*front,*rear,*temp,*temp1;

/* Enqueing the queue */
void write(int i)
{
if (rear == NULL)
   {
rear = (struct node *)malloc(1*sizeof(struct node));
rear->next = NULL;
rear->data = i;
front = rear;
   }
else
   {
temp=(struct node *)malloc(1*sizeof(struct node));
rear->next = temp;
temp->data = i;
temp->next = NULL;
rear = temp;
   }

}
/* Dequeing the queue */
voidread_q()
{
   temp1 = front;//assign temp1 to front
        //checks queue empty or not
if (temp1 == NULL)
   {
printf("\n Trying to read elements from empty queue");
return;
   }
elseif (temp1->next != NULL)
   {
      temp1 = temp1->next;
printf("\n Dequed value : %d", front->data);
free(front);
front = temp1;
   }
else
   {
```

```
printf("\n Dequed value : %d", front->data);
free(front);
front = NULL;
rear = NULL;
   }

}

/* Displaying the queue elements */
void display()
{
   temp1 = front;
        //checks queue empty or not
if ((temp1 == NULL) && (rear == NULL))
   {
printf("Queue is empty");
return;
   }
        while (temp1 != rear)
   {
printf("%d ", temp1->data);
       temp1 = temp1->next;
   }
if (temp1 == rear)
printf("%d", temp1->data);
}
//main starts here
void main()
{
int item, ch, e;
        front=rear=NULL;

printf("\n 1 - Insert");
printf("\n 2 - Delete");
printf("\n 6 - Display");
        printf("\n 5 - Exit");
while (1)
   {
printf("\n Enter the user choice to perform operation: ");
scanf("%d", &ch);
switch (ch)
       {
        case 1:
```

```
        printf("Enter data to insert into queue: ");
        scanf("%d", &item);
                write(item);
                printf("The queelemnts are:\n");
                display();
                break;
        case 2:
                read_q();
                printf("The queelemnts are:\n");
                display();
                break;
        case 3:
                printf("\nThequeelemnts are:\n");
        display();
                break;
        case 4:
        exit(0);
default:
                printf("Wrong choice, Please enter correct choice  ");
        break;
      }
   }
}
```

Output:

```
 1 - Insert
 2 - Delete
 6 - Display
 5 - Exit
 Enter the user choice to perform operation: 1
Enter data to insert into queue: 10
The que elemnts are:
10
 Enter the user choice to perform operation: 1
Enter data to insert into queue: 20
The que elemnts are:
10 20
 Enter the user choice to perform operation: 1
Enter data to insert into queue: 30
The que elemnts are:
10 20 30
 Enter the user choice to perform operation: 2

 Dequed value : 10The que elemnts are:
20 30
 Enter the user choice to perform operation: 2

 Dequed value : 20The que elemnts are:
30
 Enter the user choice to perform operation: 3

The que elemnts are:
30
 Enter the user choice to perform operation: 4
```

Performance:

Space Complexity (For n Enqueue operations)	O(n)
Time Complexity for write()	O(1) (average)
Time Complexity for read_q()	O(1)
Time Complexity for display()	O(1)

Queue Applications

Queues, mostly are used when events don't have to be processed immediately but are processed in *First In First Out* manner. These are typically used in operating systems, simulations and also useful in following areas:

- Operating systems often maintain a queue of processes that are ready to execute or that are waiting for a particular event to occur.
- When a resource is shared between multiple consumers. Examples include CPU scheduling, Disk Scheduling.
- When data is transferred asynchronously between two processes (data not necessarily received at same rate as sent). Examples include IO Buffers, pipes, file IO, etc.