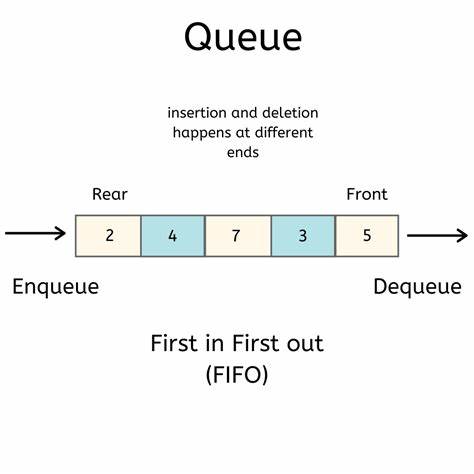
QUEUES

1. **QUEUE USING ARRAYS**

**Aim:** To implement queue using arrays.

**Theory:** A queue is a linear list of elements in which deletion of an element can take place only at one end called the front and insertion can take place on the other end which is termed as the rear. The term front and rear are frequently used while describing queues in a linked list. In this chapter, you will deal with the queue as arrays.

In the concept of a queue, the first element to be inserted in the queue will be the first element to be deleted or removed from the list. So Queue is said to follow the FIFO (First In First Out) structure. A real-life scenario in the form of example for queue will be the queue of people waiting to accomplish a particular task where the first person in the queue is the first person to be served first.



**Program:**

#include<stdio.h> //preprocessor directives

#include<stdlib.h>

#define N 100

int queue[N]; //initializing queue

int front=-1;

int rear=-1; //initializing front and top as -1

void enqueue(int x){ //called function for enqueue

if(rear==N-1){

printf("queue is full");

}

else if(front==-1&&rear==-1){

front=rear=0;

queue[rear]=x; //element insertion at rear

printf("\nThe item enqueued is:%d",queue[rear]);

}

else

rear++; //incrementing rear by 1

queue[rear]=x; //element insertion at rear

printf("\nThe item enqueued is:%d",queue[rear]);

}

void dequeue(){ //function call for dequeue

int x;

if(front==-1&&rear==-1){ //empty condition

printf("queue is empty");

}

else{

printf("\nthe item dequeued is:%d",queue[front]);

front++; //incrementing front by one so that the previous front gets deleted

}

}

void display(){

int i;

printf("the elements are:");

for(i=front;i<=rear;i++){ //loop to display elements

printf("%d",queue[i]);

}

}

void peek(){

printf("the front element is:%d",queue[front]); //displays front element

}

int main(){

int choice,x;

while(1){

printf("enter choice");

scanf("%d",&choice);

switch(choice){ //switch case

case 1:printf("\nenter the value:");

scanf("%d",&x);

enqueue(x); //function call for enqueue

break;

case 2:dequeue(); //function call for dequeue

break;

case 3:display(); //function call for display

break;

case 4:peek(); //peek function call

break;

case 5:exit(0); //exit loop

break;

}

}

}

**Algorithm:**

**Enqueue**

**Step 1:** Check if the queue is full.

**Step 2:**If the queue is full, return overflow error and exit.

**Step 3:** If the queue is not full, increment the rear pointer to point to the next empty space.

**Step 4:** Add the data element to the queue location, where the rear is pointing.

**Step 5:** return success.

**DEQUEUE**

**Step 1:** Check if the queue is empty.

**Step 2:** If the queue is empty, return the underflow error and exit.

**Step 3:** If the queue is not empty, access the data where the front is pointing.

**Step 4:** Increment the front pointer to point to the next available data element.

**Step 5:** The Return success.

**Display:**

**Step 1:** Check if queue is empty.

**Step 2:** If queue is empty return underflow.

**Step 3:** If queue is not empty access each element of queue and print the data.

**Step 4:** End

**Peek:**

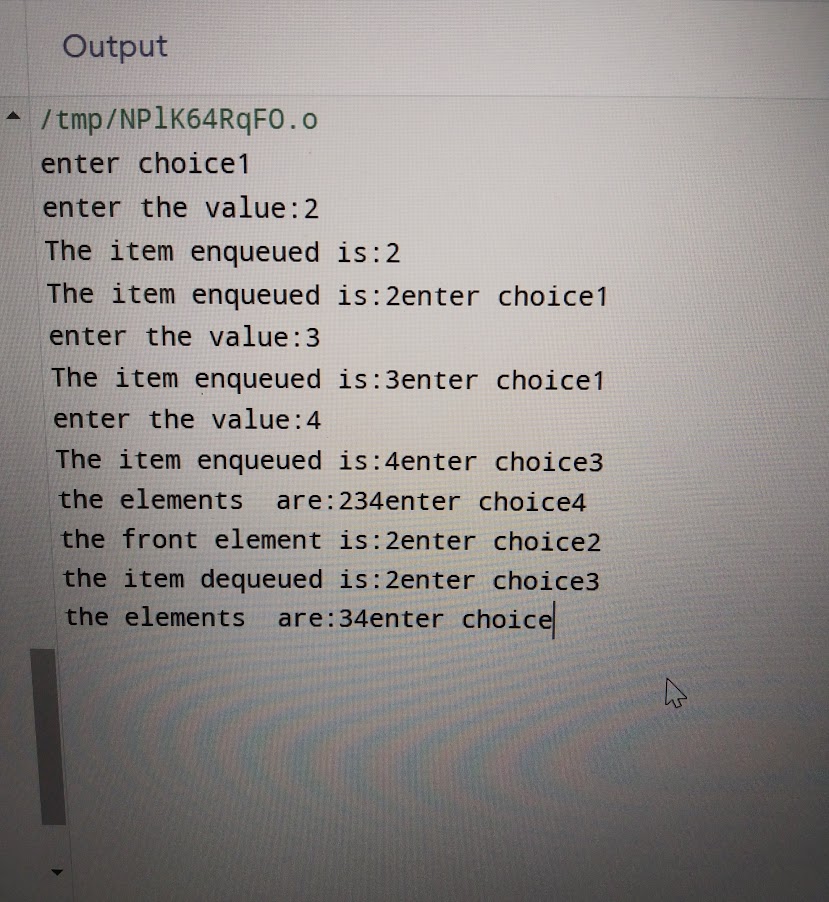
**Step 1:** Check if queue is empty.

**Step 2:** If queue is empty return underflow.

**Step 3:** If queue is not empty then return front element of queue.

**Step 4:** End

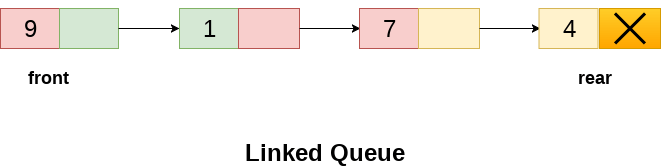
**Output:**



1. **Queue using linked list**

**Aim:** To implement queue using linked list.

**Theory:** The major problem with the queue implemented using an array is, It will work for an only fixed number of data values. That means, the amount of data must be specified at the beginning itself. Queue using an array is not suitable when we don't know the size of data which we are going to use. A queue data structure can be implemented using a linked list data structure. The queue which is implemented using a linked list can work for an unlimited number of values. That means, queue using linked list can work for the variable size of data (No need to fix the size at the beginning of the implementation). The Queue implemented using linked list can organize as many data values as we want.  
 In linked list implementation of a queue, the last inserted node is always pointed by '**rear**' and the first node is always pointed by '**front**'.



**Program:**

#include<stdio.h> //preprocessor directives

#include<stdlib.h>

void enque(x); //function declaration

void deque();

void display();

void peek();

struct node{ //creating a node

int data;

struct node \*next;

};

struct node \*front=NULL; //initializing front pointer

struct node \*rear=NULL; //initializing rear pointer

void enque(int x) //called function

{

struct node \*new;

new=(struct node \*)malloc(sizeof(struct node)); //memory declaration for node

new->data=x; //data in data part

new->next=NULL; //null in link part

if(front==0&&rear==0)

{

front=rear=new;

}

else{

rear->next=new; //linking rear next to newnode

rear=new; //rear is equal to the newnode

}

}

void display() //called function for display

{

struct node \*temp;

if(front==0&&rear==0)

printf("empty");

else

temp=front; //making temp as front

while(temp!=NULL){

printf("%d",temp->data); //printing data till temp as null

temp=temp->next; //assigning temp as temp next

}

}

void deque() //called function for dequeue

{

struct node \*temp;

temp=front;

if(front==0&&rear==0)

{

printf("empty");

}

else

printf("the element dequed is:%d",front->data);

front=front->next; //front as front's link

free(temp); //free temp pointer having first node

}

void peek()

{

printf("front is:%d",front->data); //prints first node

}

int main()

{

int choice,x;

while(1){

printf("enter choice");

scanf("%d",&choice);

switch(choice)

{

case 1:printf("enter value:");

scanf("%d",&x);

enque(x); //function call for enqueue

break;

case 2:display(); //display function call

break;

case 3:deque(); //dequeue function call

break;

case 4:peek(); //peek function call

break;

case 5:exit(0);

}

}

}

**Algorithm:**

**Enqueue:**

**Step 1:** Check if the queue is full.

**Step 2:**If the queue is full, return overflow error and exit.

**Step 3:** If the queue is not full, increment the rear pointer to point to the next empty space.

**Step 4:** Add the data element to the queue location, where the rear is pointing.

**Step 5:** return success.

**Dequeue:**

**Step 1:** Check if the queue is empty.

**Step 2:** If the queue is empty, return the underflow error and exit.

**Step 3:** If the queue is not empty, access the data where the front is pointing.

**Step 4:** Increment the front pointer to point to the next available data element.

**Step 5:** The Return success.

**Display:**

**Step 1:** Check if queue is empty.

**Step 2:** If queue is empty return underflow.

**Step 3:** If queue is not empty access each element of queue and print the data.

**Step 4:** End

**Peek:**

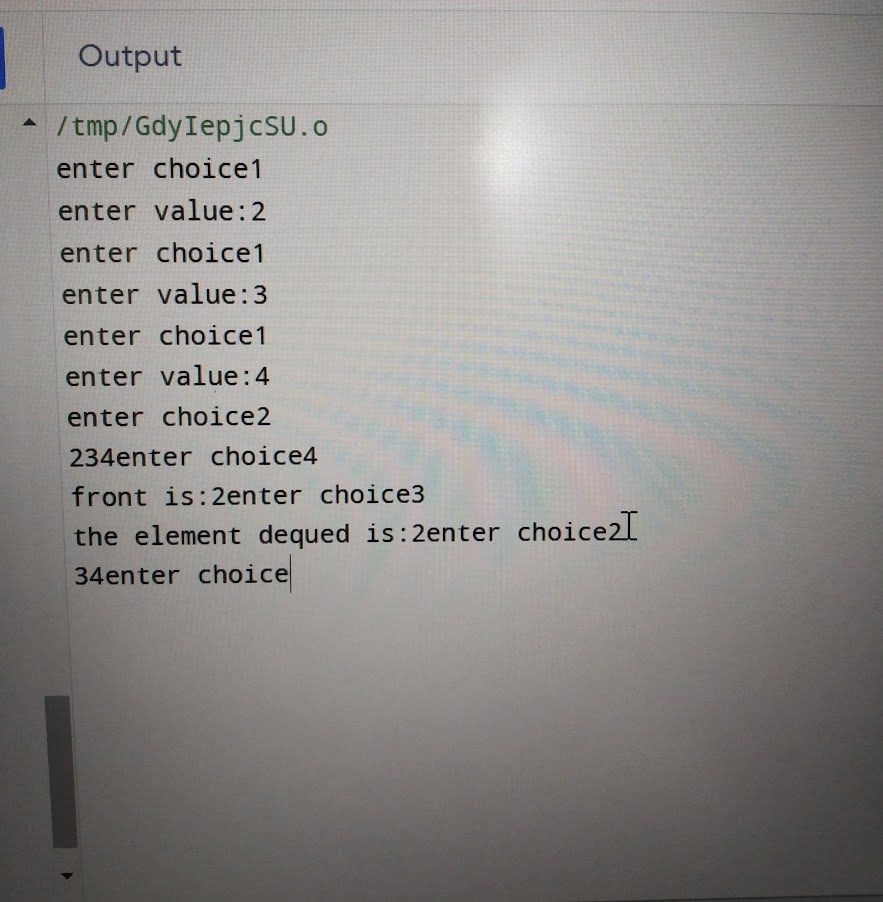
**Step 1:** Check if queue is empty.

**Step 2:** If queue is empty return underflow.

**Step 3:** If queue is not empty then return front element of queue.

**Step 4:** End

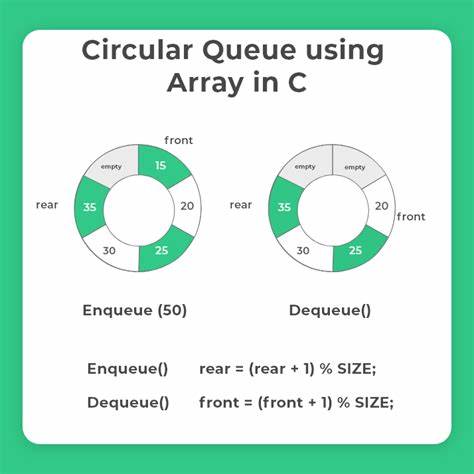
**Output:**



1. **Circular queue using array**

**Aim:** To implement circular queue using array.

**Theory:** The Circular queue is the efficient queue comparing to Array Queue. Here the array size is fixed and it wont grow, the empty block created during dequeue operation is reused. In order to implement circular queue few enhancement needs to be done on top of Array Queue implementation. To implement the circular queue two pointer variables are introduced they are front and back which is used as a reference variable to track which element is in front and back. Conceptually circular queue shall be implemented by doing the modulo operation with the queue size and reference pointer (front/back).



**Program:**

#include<stdio.h> //preprocessor directives

#include<stdlib.h>

#define N 100 //defining array size

int front=-1;

int rear=-1;

int queue[N]; //initializing array

void enqueue(int x) //called enqueue function

{

if(front==-1&&rear==-1)

{

front=rear=0; //bringing front and rear to 0th position

queue[rear]=x; //putting the value on rear

}

else if((rear+1)%N==front){ //here accessing the next block of rear since it is circular

printf("queue is full");

}

else{

rear=(rear+1)%N;

queue[rear]=x;

}

}

void dequeue() //called dequeue function

{

int x;

if(front==-1&&rear==-1){ //empty condition

printf("queue is empty");

}

else if(front==rear){ //if there is only 1 element in the queue

printf("the ele is:%d",queue[front]);

front=rear=-1;

}

else{

printf("%d",queue[front]); //normal queue insertion

front=(front+1)%N;

}

}

void display() //called display function

{

int i=0;

if(front==-1&&rear==-1)

printf("empty");

else

while(i!=rear) //loop to display

{

printf("%d",queue[i]);

i=(i+1)%N;

}

printf("%d",queue[i]);

}

int main() //main function

{

int choice,x;

printf("1.enque\n2.deque\n3.display\n4.exit");

while(1){

printf("enter choice");

scanf("%d",&choice);

switch(choice) //switch case

{

case 1:printf("enter the elements");

scanf("%d",&x);

enqueue(x); //enqueue function call

break;

case 2:dequeue(); //dequeue function call

break;

case 3:display(); //display function call

break;

case 4:exit(0);

}

}

}

**Algorithm:**

**Enqueue:**

1. If ( COUNT = MAX ) then

a. Display “Queue overflow”;

b. Return;

2. Otherwise

a. If ( REAR = MAX ) then

i. REAR := 1;

b. Otherwise

i. REAR := REAR + 1;

c. QUEUE(REAR) := ITEM;

d. COUNT := COUNT + 1;

3. Return;

**Dequeue:**

1. If ( COUNT = -1) then

a. Display “Queue underflow”;

b. Return;

2. Otherwise

a. If ( REAR = MAX ) then

i. REAR := 1;

b. Otherwise

i. Front := Front + 1;

3. Return;

**Display:**

1. If ( COUNT = -1) then

a. Display “Queue underflow”;

b. Return;

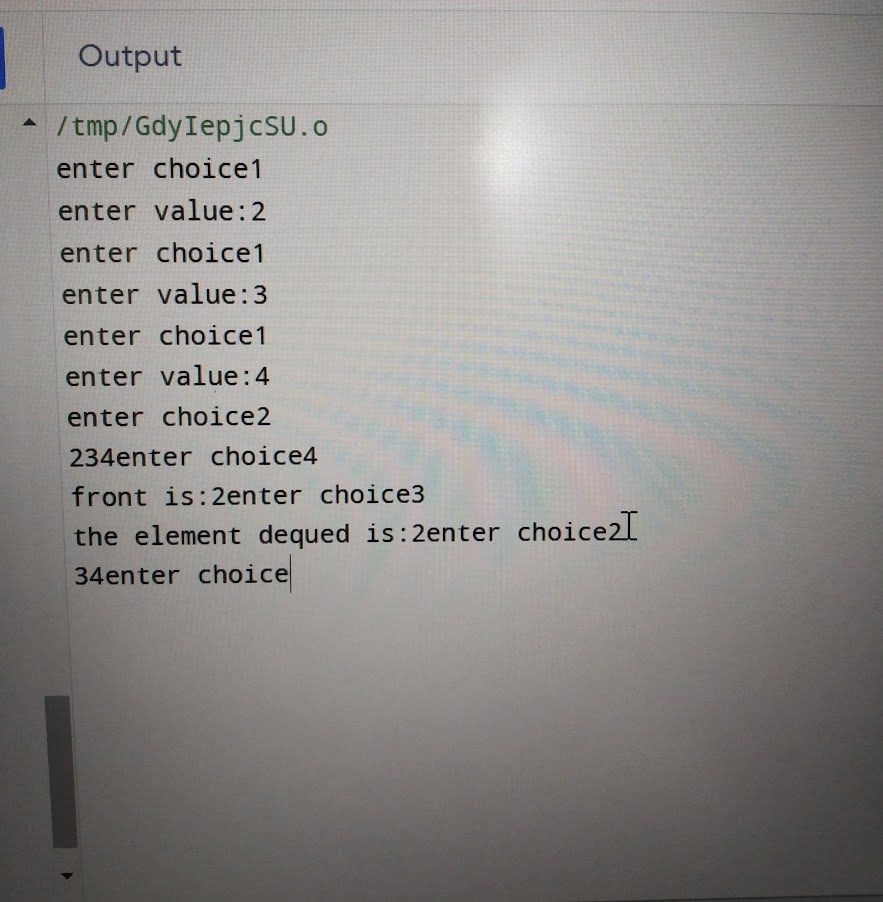
2. Otherwise

While(temp!=rear)

Print values of temp;

3. Return;

**Output:**



1. **Circular queue using linked list.**

**Aim:** To implement circular queue using linked list.

**Theory:** You can implement the circular queue using both the 1-D array and the linked list. However, implementing a circular link is a new addition that you need to execute. Additionally, this queue works by the process of circular incrementation. That is, when you reach the end of a queue, you start from the beginning of a queue. The circular incrementation is achievable with the help of the modulo division.

Now you will understand how you can achieve circular incrementation, with the help of an example. Let’s say the MaxSize of your queue is 5, and the rear pointer has already reached the end of a queue. There is one empty space at the beginning of a queue, which means that the front pointer is pointing to location 1.

**Program:**

#include<stdio.h> //preprocessor directives

#include<stdlib.h>

struct node{ //creating a node in list

int data;

struct node \*next;

};

struct node \*front=0; //front pointer

struct node \*rear=0; //rear pointer

void enqueue(int x){ //called equeue function

struct node \*newnode; //initializing a newnode

newnode=(struct node \*)malloc(sizeof(struct node)); //memory allocation for new node

newnode->data=x; data in data part of ll

newnode->next=0; Null in link part

if(rear==0){

front=rear=newnode; //front and rear pointer pointing to newnode

rear->next=front;

}

else{

rear->next=newnode; //link part of rear has newnode address

rear=newnode; //rear as newnode

rear->next=front; //link of rear pointing to front pointer

}

}

void dequeue() //called function for dequeue

{

struct node \*temp; //initializing temp pointer

temp=front; //temp as front

if(front==0&&rear==0){

printf("queue is empty");

}

else if(front==rear){

front=rear=0; //front and rear pointer as 0

free(temp); //freeing temp pointer

}

else{

printf("the ele deleted is:%d",front->data);

front=front->next;

rear->next=front; //rear's link as front

free(temp); //freeing temp pointer

}

}

void peek(){

printf("the front ele is:%d",front->data); //displays front' data

}

void display(){ // called display function

struct node \*temp; //initializing temp pointer

temp=front; //temp as front

if(front==0&&rear==0){

printf("queue is empty");

}

else{

while(temp->next!=front){ //loop for display

printf("%d",temp->data);

temp=temp->next; //temp as temp's link address

}

printf("%d",temp->data);

}

}

int main()

{

int choice,x;

while(1){

printf("enter choice:");

scanf("%d",&choice);

switch(choice){

case 1:printf("enter value:");

scanf("%d",&x);

enqueue(x); //function call for enqueue

break;

case 2:display(); //function call for display

break;

case 3:dequeue(); //function call for dequeue

break;

case 4:peek(); //function call for peek

break;

case 5:exit(0); //exit loop

break;

}

}

}

**Algorithm:**

**Enqueue**

1.Create a struct node type node.

2.Insert the given data in the new node data section and NULL in address section.

3.If Queue is empty then initialize front and rear from new node.

4.Queue is not empty then initialize rear next and rear from new node.

5.New node next initialize from front

**Dequeue**

1.Check if queue is empty or not.

2.If queue is empty then dequeue is not possible.

3.Else Initialize temp from front.

4.If front is equal to the rear then initialize front and rear from null.

5.Print data of temp and free temp memory.

6.If there is more than one node in Queue then make front next to front then initialize rear next from front.

7.Print temp and free temp.

**Display:**

1. If ( COUNT = -1) then

a. Display “Queue underflow”;

b. Return;

2. Otherwise

While(temp!=rear)

Print values of temp;

3. Return;

**Output:**

