Circular queues, Priority queues, De-queues

Circular Queue

A circular queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle.
A circular queue solved the limitations of the normal queue. Thus making it a better pick than the normal queue. It also follows the first come first serve algorithm. Circular Queue is also called *ring Buffer*.

Graphical representation of a circular queue is as follows...

Operations On A Circular Queue

1. Enqueue- adding an element in the queue if there is space in the queue.
2. Dequeue- Removing elements from a queue if there are any elements in the queue
3. Front- get the first item from the queue.
4. Rear- get the last item from the queue.
5. isEmpty/isFull- checks if the queue is empty or full.

Applications Of A Circular Queue

- Memory management: circular queue is used in memory management.
- Process Scheduling: A CPU uses a queue to schedule processes.
- Traffic Systems: Queues are also used in traffic systems.

Implementation of Circular Queue
To implement a circular queue data structure using an array, we first perform the following steps before we implement actual operations.

- Step 1 - Include all the header files which are used in the program and define a constant 'SIZE' with specific value.
- Step 2 - Declare all user defined functions used in circular queue implementation.
- Step 3 - Create a one dimensional array with above defined SIZE (int cQueue[SIZE])
- Step 4 - Define two integer variables 'front' and 'rear' and initialize both with '-1'. (int front = -1, rear = -1)
- Step 5 - Implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on circular queue.

enQueue(value) - Inserting value into the Circular Queue

In a circular queue, enQueue() is a function which is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at rear position. The enQueue() function takes one integer value as parameter and inserts that value into the circular queue. We can use the following steps to insert an element into the circular queue...

- Step 1 - Check whether queue is FULL. ((rear == SIZE-1 && front == 0) || (front == rear+1))
- Step 2 - If it is FULL, then display "Queue is FULL!!! Insertion is not possible!!!" and terminate the function.

- Step 3 - If it is NOT FULL, then check rear == SIZE - 1 && front != 0 if it is TRUE, then set rear = -1.
- Step 4 - Increment rear value by one (rear++), set queue[rear] = value and check 'front == -1' if it is TRUE, then set front = 0.

deQueue() - Deleting a value from the Circular Queue

In a circular queue, deQueue() is a function used to delete an element from the circular queue. In a circular queue, the element is always deleted from front position. The deQueue() function doesn't take any value as a parameter. We can use the following steps to delete an element from the circular queue...

- Step 1 - Check whether queue is EMPTY. (front == -1 && rear == -1)
- Step 2 - If it is EMPTY, then display "Queue is EMPTY!!! Deletion is not possible!!!" and terminate the function.
- Step 3 - If it is NOT EMPTY, then display queue[front] as deleted element and increment the front value by one (front ++). Then check whether front == SIZE, if it is TRUE, then set front = 0. Then check whether both front - 1 and rear are equal (front -1 == rear), if it TRUE, then set both front and rear to '-1' (front = rear = -1).

display() - Displays the elements of a Circular Queue

We can use the following steps to display the elements of a circular queue...

- Step 1 - Check whether queue is EMPTY. (front == -1)
- Step 2 - If it is EMPTY, then display "Queue is EMPTY!!!" and terminate the function.
- Step 3 - If it is NOT EMPTY, then define an integer variable 'i' and set 'i = front'.
- Step 4 - Check whether 'front <= rear', if it is TRUE, then display 'queue[i]' value and increment 'i' value by one (i++). Repeat the same until 'i <= rear' becomes FALSE.
- Step 5 - If 'front <= rear' is FALSE, then display 'queue[i]' value and increment 'i' value by one (i++). Repeat the same until'i <= SIZE - 1' becomes FALSE.
- Step 6 - Set i to 0.
- Step 7 - Again display 'cQueue[i]' value and increment i value by one (i++). Repeat the same until 'i <= rear' becomes FALSE.

*****Array based implementation on Circular Queue in C**

```
#include<stdio.h>

#include<conio.h>

#define SIZE 4
```

```
void enQueue(int);
void deQueue();
void display();
int cQueue[SIZE], front = -1, rear = -1;
int main()
{
  int choice, value;
  while(1){
    printf("\n****** MENU ******\n");
    printf("1. Insert\n2. Delete\n3. Display\n4. Exit\n");
    printf("Enter your choice: ");
    scanf("%d",&choice);
    switch(choice){
        case 1: printf("\nEnter the value to be insert:  ");
                scanf("%d",&value);
                enQueue(value);
                break;
        case 2: deQueue();
                break;
        case 3: display();

        default: printf("\nPlease select the correct choice!!!\n");
    }
  }
}
void enQueue(int value)
```


```
{
  if((front == 0 && rear == SIZE - 1) || (front == rear+1))
    printf("\nCircular Queue is Full! Insertion not possible!!!\n");
  else{
    if(rear == SIZE-1 && front != 0)
          rear = -1;
    cQueue[++rear] = value;
    printf("\nInsertion Success!!!\n");
    if(front == -1)
          front = 0;
  }
}
void deQueue()
{
  if(front == -1 && rear == -1)
    printf("\nCircular Queue is Empty! Deletion is not possible!!!\n");
  else{
    printf("\nDeleted element : %d\n",cQueue[front++]);
    if(front == SIZE)
          front = 0;
    if(front-1 == rear)
          front = rear = -1;
  }
}
void display()
{
```

```
  if(front == -1)
    printf("\nCircular Queue is Empty!!!\n");
  else{
    int i = front;
    printf("\nCircular Queue Elements are : \n");
    if(front <= rear){
          while(i <= rear)
            printf("%d\t",cQueue[i++]);
    }
    else{
          while(i <= SIZE - 1)
            printf("%d\t", cQueue[i++]);
          i = 0;
          while(i <= rear)
            printf("%d\t",cQueue[i++]);
    }
  }
}
```

*****Array Implementation on Circular queue in C++**

```
//program with answers on circular queue

#include <iostream>
#define SIZE 5

using namespace std;

class Queue {
private:
   int items[SIZE], front, rear;
```

```
public:
   Queue(){
      front = -1;
      rear = -1;
   }

   bool isFull(){
      if(front == 0 && rear == SIZE - 1){
         return true;
      }
      if(front == rear + 1) {
         return true;
      }
      return false;
   }

   bool isEmpty(){
      if(front == -1) return true;
      else return false;
   }

   void enQueue(int element){
      if(isFull()){
         cout << "Queue is full";
      } else {
         if(front == -1) front = 0;
         rear = (rear + 1) % SIZE;
         items[rear] = element;
         cout << endl << "Inserted " << element << endl;
      }
   }

   int deQueue(){
      int element;
      if(isEmpty()){
         cout << "Queue is empty" << endl;
         return(-1);
      } else {
         element = items[front];
         if(front == rear){
            front = -1;
            rear = -1;
         }
         else {
            front=(front+1) % SIZE;
```

```
        }
        return(element);
      }
   }

   void display()
   {
      int i;
      if(isEmpty()) {
         cout << endl << "Empty Queue" << endl;
      }
      else
      {
         cout << "Front -> " << front;
         cout << endl << "Items -> ";
         for(i=front; i!=rear;i=(i+1)%SIZE)
            cout << items[i];
         cout << items[i];
         cout << endl << "Rear -> " << rear;
      }
   }
};

int main()
{
   Queue q;

  q.deQueue();

   q.enQueue(1);
   q.enQueue(2);
   q.enQueue(3);
   q.enQueue(4);
   q.enQueue(5);

   q.enQueue(6);


   q.display();

   int elem = q.deQueue();

   if( elem != -1)
     cout << endl << "Deleted Element is " << elem;
```

```
    q.display();

    q.enQueue(7);

    q.display();

    q.enQueue(8);

    return 0;
}
```

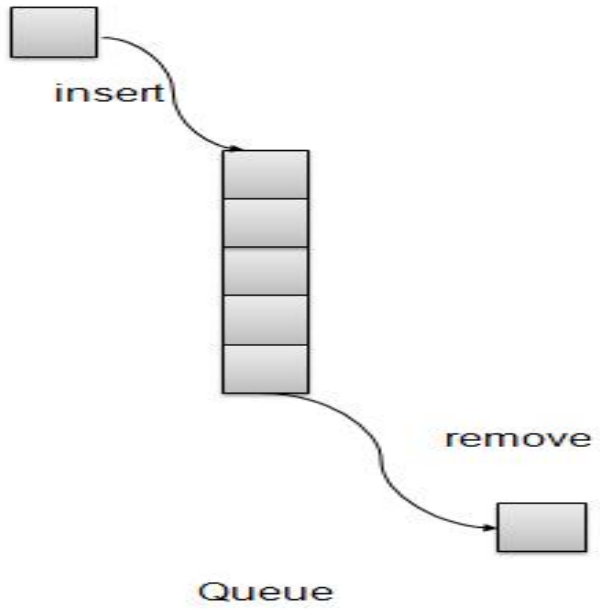
Priority queues

Priority Queue is more specialized data structure than Queue. Like ordinary queue, priority queue has same method but with a major difference. In Priority queue items are ordered by key value so that item with the lowest value of key is at front and item with the highest value of key is at rear or vice versa. So we're assigned priority to item based on its key value. Lower the value, higher the priority. Following are the principal methods of a Priority Queue.

Basic Operations

- **insert / enqueue** − add an item to the rear of the queue.
- **remove / dequeue** − remove an item from the front of the queue.

Priority Queue Representation

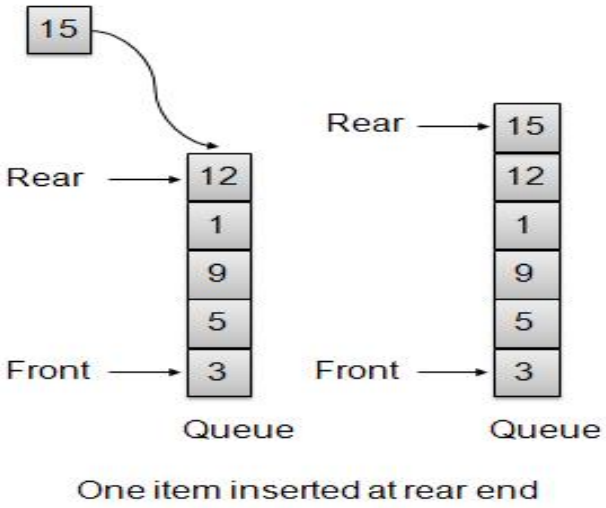


We're going to implement Queue using array in this article. There is few more operations supported by queue which are following.

- **Peek** − get the element at front of the queue.
- **isFull** − check if queue is full.
- **isEmpty** − check if queue is empty.

Insert / Enqueue Operation

Whenever an element is inserted into queue, priority queue inserts the item according to its order. Here we're assuming that data with high value has low priority.



One item inserted at rear end

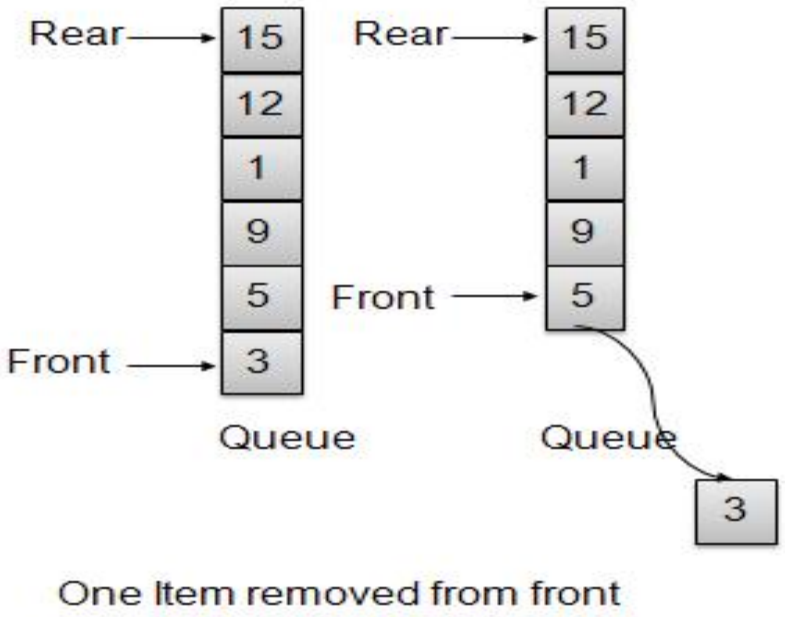
```
void insert(int data){
  int i = 0;

  if(!isFull()){
    // if queue is empty, insert the data

    if(itemCount == 0){
      intArray[itemCount++] = data;
    }else{
      // start from the right end of the queue
      for(i = itemCount - 1; i >= 0; i-- ){
        // if data is larger, shift existing item to right end
        if(data > intArray[i]){
          intArray[i+1] = intArray[i];
        }else{
          break;
        }
      }
      // insert the data
      intArray[i+1] = data;
      itemCount++;
    }
  }
}
```

Remove / Dequeue Operation

Whenever an element is to be removed from queue, queue get the element using item count. Once element is removed. Item count is reduced by one.



One Item removed from front

```
int removeData(){
   return intArray[--itemCount];
}
```

*****Array implementation on Priority queue using C**

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

#define MAX 5

void insert_by_priority(int);
void delete_by_priority(int);
void create();
void check(int);
void display_pqueue();

int pri_que[MAX];
int front, rear;

int main()
{
   int n, ch;
```

```
    printf("\n1 - Insert an element into queue");
    printf("\n2 - Delete an element from queue");
    printf("\n3 - Display queue elements");
    printf("\n4 - Exit");

    create();

    while (1)
    {
        printf("\nEnter your choice : ");
        scanf("%d", &ch);

        switch (ch)
        {
        case 1:
            printf("\nEnter value to be inserted : ");
            scanf("%d",&n);
            insert_by_priority(n);
            break;
        case 2:
            printf("\nEnter value to delete : ");
            scanf("%d",&n);
            delete_by_priority(n);
            break;
        case 3:
            display_pqueue();
            break;
        case 4:
            exit(0);
        default:
            printf("\nChoice is incorrect, Enter a correct choice");
        }
    }
}

/* Function to create an empty priority queue */
void create()
{
    front = rear = -1;
}

/* Function to insert value into priority queue */
void insert_by_priority(int data)
{
```


```
    if (rear >= MAX - 1)
    {
       printf("\nQueue overflow no more elements can be inserted");
       return;
    }
    if ((front == -1) && (rear == -1))
    {
       front++;
       rear++;
       pri_que[rear] = data;
       return;
    }
    else
       check(data);
    rear++;
}

/* Function to check priority and place element */
void check(int data)
{
    int i,j;

    for (i = 0; i <= rear; i++)
    {
       if (data >= pri_que[i])
       {
          for (j = rear + 1; j > i; j--)
          {
             pri_que[j] = pri_que[j - 1];
          }
          pri_que[i] = data;
          return;
       }
    }
    pri_que[i] = data;
}

/* Function to delete an element from queue */
void delete_by_priority(int data)
{
    int i;

    if ((front==-1) && (rear==-1))
    {
       printf("\nQueue is empty no elements to delete");
```

```
        return;
    }

    for (i = 0; i <= rear; i++)
    {
        if (data == pri_que[i])
        {
            for (; i < rear; i++)
            {
                pri_que[i] = pri_que[i + 1];
            }

        pri_que[i] = -99;
        rear--;

        if (rear == -1)
            front = -1;
        return;
        }
    }
    printf("\n%d not found in queue to delete", data);
}

/* Function to display queue elements */
void display_pqueue()
{
    if ((front == -1) && (rear == -1))
    {
        printf("\nQueue is empty");
        return;
    }

    for (; front <= rear; front++)
    {
        printf(" %d ", pri_que[front]);
    }

    front = 0;
}
```

*****Array implementation on Priority queue using C++**

```
//program for Implementation on Priority queue

#include <iostream>
```

```
#include <cstdio>
#include <cstring>
#include <cstdlib>
using namespace std;

struct node
{
        int priority;
        int info;
        struct node *link;
};
class Priority_Queue
{
   private:
      node *front;
   public:
      Priority_Queue()
      {
         front = NULL;
      }

      void insert(int item, int priority)
      {
         node *tmp, *q;
         tmp = new node;
         tmp->info = item;
         tmp->priority = priority;
         if (front == NULL || priority < front->priority)
         {
            tmp->link = front;
            front = tmp;
         }
         else
         {
            q = front;
            while (q->link != NULL && q->link->priority <= priority)
               q=q->link;
            tmp->link = q->link;
            q->link = tmp;
         }
      }
      void del()
      {
         node *tmp;
         if(front == NULL)
```

```
            cout<<"Queue Underflow\n";
        else
        {
            tmp = front;
            cout<<"Deleted item is: "<<tmp->info<<endl;
            front = front->link;
            free(tmp);
        }
    }

    void display()
    {
        node *ptr;
        ptr = front;
        if (front == NULL)
            cout<<"Queue is empty\n";
        else
        {       cout<<"Queue is :\n";
            cout<<"Priority      Item\n";
            while(ptr != NULL)
            {
                cout<<ptr->priority<<"              "<<ptr->info<<endl;
                ptr = ptr->link;
            }
        }
    }
};

int main()
{
   int choice, item, priority;
   Priority_Queue pq;
   do
   {
      cout<<"1.Insert\n";
      cout<<"2.Delete\n";
      cout<<"3.Display\n";
      cout<<"4.Quit\n";
      cout<<"Enter your choice : ";
      cin>>choice;
      switch(choice)
      {
      case 1:
         cout<<"Input the item value to be added in the queue : ";
         cin>>item;
```

```
            cout<<"Enter its priority : ";
            cin>>priority;
            pq.insert(item, priority);
            break;
        case 2:
            pq.del();
            break;
        case 3:
            pq.display();
            break;
        case 4:
            break;
        default :
            cout<<"Wrong choice\n";
        }
    }
    while(choice != 4);
    return 0;
}
```

De-Queue

Deque or Double Ended Queue is a generalized version of Queue data structure that allows insert and delete at both ends.

Operations on Deque:

Mainly the following four basic operations are performed on queue:

insertFront(): Adds an item at the front of Deque.
insertLast(): Adds an item at the rear of Deque.
deleteFront(): Deletes an item from front of Deque.
deleteLast(): Deletes an item from rear of Deque.

In addition to above operations, following operations are also supported

getFront(): Gets the front item from queue.
getRear(): Gets the last item from queue.
isEmpty(): Checks whether Deque is empty or not.
isFull(): Checks whether Deque is full or not.

Applications of Deque:

Since Deque supports both stack and queue operations, it can be used as both. The Deque data structure supports clockwise and anticlockwise rotations in O(1) time which can be useful in certain applications.

*****Implementation of Dequeue using C**

```
#include<stdio.h>
#include<stdlib.h>
#define MAX 30

typedef struct dequeue
{
   int data[MAX];
   int rear,front;
}dequeue;

void initialize(dequeue *p);
int empty(dequeue *p);
int full(dequeue *p);
void enqueueR(dequeue *p,int x);
void enqueueF(dequeue *p,int x);
int dequeueF(dequeue *p);
int dequeueR(dequeue *p);
void print(dequeue *p);

int main()
{
   int i,x,op,n;
   dequeue q;
   initialize(&q);
   do
   {
    printf("\n1.Create\n2.Insert(Rear)\n3.Insert(Front)\n4.Delete(Rear)\n5.Delet(Front)");
      printf("\n6.Print\n7.Exit\n\nEnter your choice:");
      scanf("%d",&op);

      switch(op)
      {
         case 1: printf("\nEnter number of elements:");
                 scanf("%d",&n);
                 initialize(&q);
                 printf("\nEnter the data:");
                 for(i=0;i<n;i++)
                 {
                    scanf("%d",&x);
                    if(full(&q))
                    {
                       printf("\nQueue is Full!!");
                       exit(0);
                    }
```


```
            enqueueR(&q,x);
        }
        break;

case 2: printf("\nInsert Element : ");
        scanf("%d",&x);
        if(full(&q))
        {
            printf("\nQueue is Full!!");
            exit(0);
        }
        enqueueR(&q,x);
        break;

case 3: printf("\nInsert Element :");
        scanf("%d",&x);
        if(full(&q))
        {
            printf("\nQueue is Full!!");
            exit(0);
        }
        enqueueF(&q,x);
        break;

case 4: if(empty(&q))
        {
            printf("\nQueue is Empty!!");
            exit(0);
        }
        x=dequeueR(&q);
        printf("\n Deleted Element is %d\n",x);
        break;

case 5: if(empty(&q))
        {
            printf("\nQueue is Empty!!");
            exit(0);
        }
        x=dequeueF(&q);
        printf("\nDeleted Element is %d\n",x);
        break;

case 6: print(&q);
        break;
```

```
            default: break;
        }
    }while(op!=7);
}
void initialize(dequeue *P)
{
    P->rear=-1;
    P->front=-1;
}
int empty(dequeue *P)
{
    if(P->rear==-1)
        return(1);

    return(0);
}
int full(dequeue *P)
{
    if((P->rear+1)%MAX==P->front)
        return(1);

    return(0);
}
void enqueueR(dequeue *P,int x)
{
    if(empty(P))
    {
        P->rear=0;
        P->front=0;
        P->data[0]=x;
    }
    else
    {
        P->rear=(P->rear+1)%MAX;
        P->data[P->rear]=x;
    }
}
void enqueueF(dequeue *P,int x)
{
    if(empty(P))
    {
        P->rear=0;
        P->front=0;
        P->data[0]=x;
    }
```


```
    else
    {
        P->front=(P->front-1+MAX)%MAX;
        P->data[P->front]=x;
    }
}
int dequeueF(dequeue *P)
{
    int x;
    x=P->data[P->front];
    if(P->rear==P->front)    //delete the last element
        initialize(P);
    else
        P->front=(P->front+1)%MAX;
    return(x);
}
int dequeueR(dequeue *P)
{
    int x;
    x=P->data[P->rear];
    if(P->rear==P->front)
        initialize(P);
    else
        P->rear=(P->rear-1+MAX)%MAX;
    return(x);
}
void print(dequeue *P)
{
    if(empty(P))
    {
        printf("\nQueue is empty!!");
        exit(0);
    }
    int i;
    i=P->front;
    while(i!=P->rear)
    {
        printf("\n%d",P->data[i]);
        i=(i+1)%MAX;
    }
    printf("\n%d\n",P->data[P->rear]);
}
```


*****Implementation on dequeue using C++**

```
#include<iostream>
using namespace std;
#define SIZE 10
class dequeue {
  int a[20],f,r;
  public:
    dequeue();
    void insert_at_beg(int);
    void insert_at_end(int);
    void delete_fr_front();
    void delete_fr_rear();
    void show();
};
dequeue::dequeue() {
  f=-1;
  r=-1;
}
void dequeue::insert_at_end(int i) {
  if(r>=SIZE-1) {
    cout<<"\n insertion is not possible, overflow!!!!";
  } else {
    if(f==-1) {
      f++;
      r++;
    } else {
      r=r+1;
    }
    a[r]=i;
    cout<<"\nInserted item is"<<a[r];
  }
}
void dequeue::insert_at_beg(int i) {
  if(f==-1) {
    f=0;
    a[++r]=i;
    cout<<"\n inserted element is:"<<i;
  } else if(f!=0) {
    a[--f]=i;
    cout<<"\n inserted element is:"<<i;
  } else {
    cout<<"\n insertion is not possible, overflow!!!";
  }
}
void dequeue::delete_fr_front() {
  if(f==-1) {
```

```
  cout<<"deletion is not possible::dequeue is empty";
  return;
}
else {
  cout<<"the deleted element is:"<<a[f];
  if(f==r) {
    f=r=-1;
    return;
  } else
    f=f+1;
  }
}
void dequeue::delete_fr_rear() {
  if(f==-1) {
    cout<<"deletion is not possible::dequeue is empty";
    return;
  }
  else {
    cout<<"the deleted element is:"<<a[r];
    if(f==r) {
      f=r=-1;
    } else
      r=r-1;
  }
}
void dequeue::show() {
  if(f==-1) {
    cout<<"Dequeue is empty";
  } else {
    for(int i=f;i<=r;i++) {
      cout<<a[i]<<" ";
    }
  }
}
int main() {
  int c,i;
  dequeue d;
  do {
  cout<<"\n 1.insert at beginning";
  cout<<"\n 2.insert at end";
  cout<<"\n 3.show";
  cout<<"\n 4.deletion from front";
  cout<<"\n 5.deletion from rear";
  cout<<"\n 6.exit";
  cout<<"\n enter your choice:";
```

```
    cin>>c;
    switch(c) {
      case 1:
        cout<<"enter the element to be inserted";
        cin>>i;
        d.insert_at_beg(i);
      break;
      case 2:
        cout<<"enter the element to be inserted";
        cin>>i;
        d.insert_at_end(i);
      break;
      case 3:
        d.show();
      break;
      case 4:
        d.delete_fr_front();
      break;
      case 5:
        d.delete_fr_rear();
      break;
      case 6:
        exit(1);
      break;
      default:
        cout<<"invalid choice";
      break;
    }
  }
 while(c!=7);
}
```