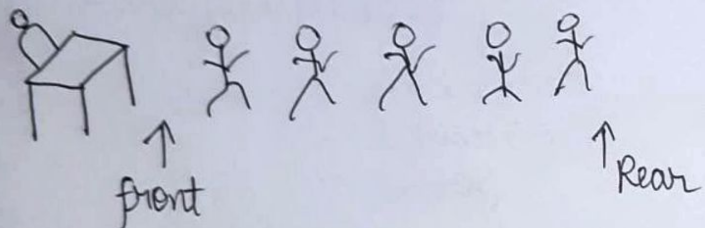
☆ Queue:-

Queue is an abstract data structure, somewhat similar to Stacks. Unlike Stacks, a ~~Queue~~ Queue is open at both ends. One end is always used to insert data (enqueue) and the other end is used to remove data (dequeue). Queue follows First-In-First-Out. (FIFO) i.e. the data item stored first will be accessed first.

Real World Example:- A real world example of queue ~~is~~ can be a single-lane one-way road, where the vehicles enters first, exist first.

⇒ Queue ADT:-



Data:⇒

1. Space for storing elements.
2. Front - for deletion
3. Rear - for insertion

Operations:⇒

1. enqueue (x)
2. dequeue ()
3. isEmpty()
4. isFull()
5. first()
6. last()

⋮

We can implement queue in two ways:-

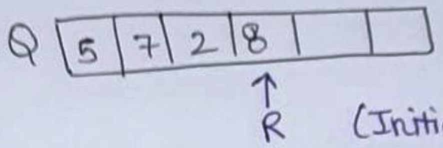
(1) Array

(2) Linkedlist

✱ Queue Using ARRAYS :⇒

⇒ Queue Using Single Pointer :⇒

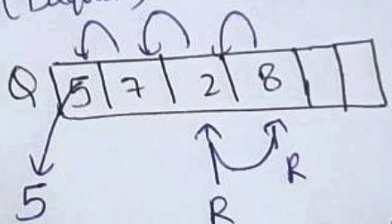
(1) Insert Operation :- (enqueue)



(Initially R is (-1))
here R is Rear.

∴ Insert – O(1)

(2) Delete Operation :- (Dequeue)

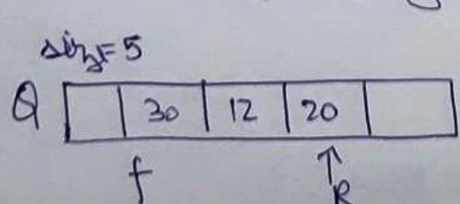


In this we have to shift all the elements to the left & also R is shifted to its previous left position.

∴ Delete – O(n)

⇒ Queue Using Two Pointers :-

In this method we move counter of the queue (line of persons) to forward by one step ~~or~~ instead of shifting all the other element of the queue backward.



Initially :- (Front = Rear = -1)

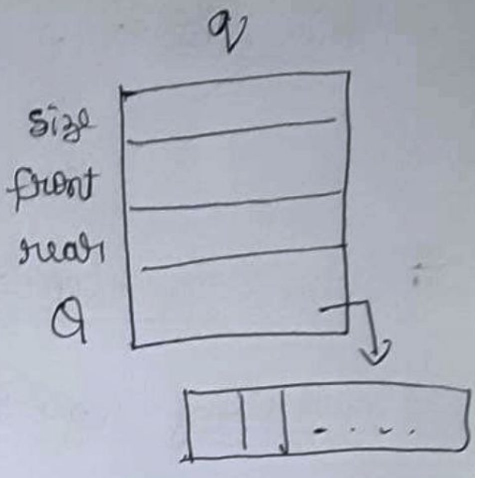
Empty :- if (Front = Rear)

Full :- if (Rear = size -1)

enqueue – O(1)

dequeue – O(1)

⇒ Code of Queue Using Array:-




```
#include <iostream>
using namespace std;

class queue {
  private:
        int size;
        int front;
        int rear;
        int* Q;
  public:
        queue (int size);
        ~queue ();
        bool isfull ();
        bool is empty ();
        void enqueue (int x);
        int dequeue ();
        void display ();
};

queue :: queue (int size) {
        this->size = size;
        front = -1;
        rear = -1;
        Q = new int [size];
}
queue :: ~queue () {
        delete []Q;
}

bool queue :: isempty () {
        if (front == rear) { return true; }
        return false;
}
```

```
bool queue:: isfull() {
        if(rear== size-1) {
                return true;
        }
        return false;
}
void queue::enqueue (int x) {
    if(isfull()) {cout <<"Queue Overflow";
    else {
        rear++;
        Q[rear] = x;
    }
}
int queue::dequeue() {
    int x=-1;
    if(isEmpty()) {cout <<"Queue Underflow";
    else {
        front++;
        x= Q[front];
    }
    return x;
};
void queue:: display (){
    for(int i=front+1; i<=rear; j++) {
            cout<< Q[i] <<flush;
            if(i<rear)    cout <<" <- "<<flush;
    }
    cout<<endl;
}
int main () {
    int A[]= {1,3,5,7,9}
    ~~queue q(sizeof(A)[]~~
    queue q(sizeof (A) / sizeof (A[0]));

    // Enqueue
    for(int i=0; i<sizeof(A)/ sizeof (A[0]); i++)    q.enqueue(A[i]);
```

```
//Display
q.display();

//Overflow
for(int i=0; i<sizeof(A)/sizeof(A[0]); i++)  q.dequeue();

//Underflow
q.dequeue();

return 0;
}
```

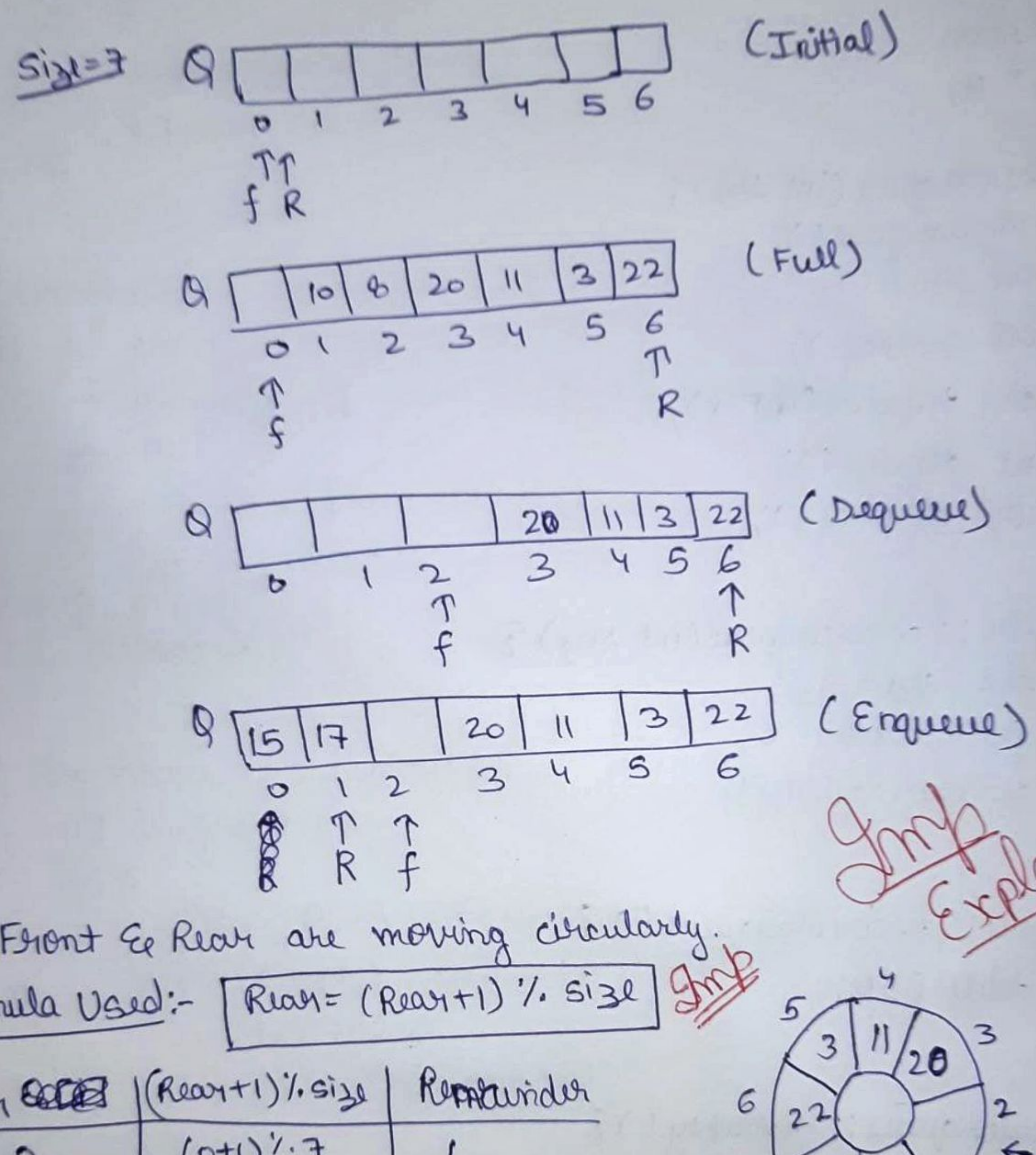
✲ Drawbacks of array implementation of Queue :-

(1) **Memory wastage** :- The space of the array, which is used to store queue elements, can never be reused to store the elements of that queue because the elements can only be inserted at ~~front and~~ rear and elements can be deleted from front, so after the deletion (dequeue) all the space before first can never be filled.

(2) **Array size** :- There might be situations in which, we may need to extend the queue to insert more elements if we use an array to implement queue, it will almost be impossible to extend the array size, therefore deciding the correct array size is always a problem in array.

☆ Circular Queue:-

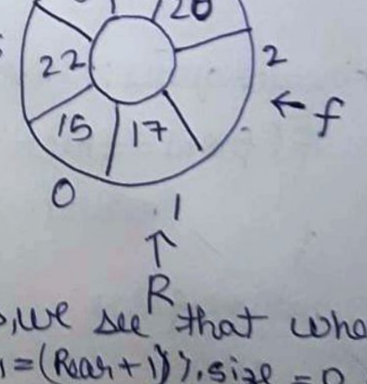
Circular Queue, is also a linear data structure, which follows the principle of FIFO (First in First Out), but instead of ending the queue at the last position, it again starts from the first position after the last, hence making the queue behave like a circular data structure.



Imp Explanation

∴ Front & Rear are moving circularly.

formula Used:- $\boxed{Rear = (Rear+1) \% size}$ Imp



Rear	(Rear+1)%size	Remainder
0	(0+1)%7	1
1	(1+1)%7	2
2	(2+1)%7	3
3	(3+1)%7	3
4	(4+1)%7	5
5	(5+1)%7	6
6	(6+1)%7	0
0		

Thus, we see that when $Rear = (Rear+1)\%size = 0$, then the Rear will come to the begining of the Queue.

```
#include <iostream>
using namespace std;

class circularqueue {
    private:
        int size;
        int front;
        int rear;
        int* Q;
    public:
        circularqueue (int size);
        ~circularqueue ();
        bool isfull ();
        bool isempty ();
        void enqueue (int x);
        int dequeue ();
        void display ();
};

circularqueue :: circularqueue (int size) {
        this -> size = size;
        front = rear = 0;
        Q = new int [size];
}

circularqueue :: ~circularqueue () {
        delete [] Q;
}

bool circularqueue :: isempty () {
        if (front == rear) {
                return true;
        }
        return false;
}
```

```
bool circularqueue :: isfull () {
        if ((rear+1) % size == front) return true;
        return false;
}
void circularqueue :: enqueue (int x) {
    if((rear+1)%size == front)   cout << "Queue Overflow";
    else {
          rear= (rear+1) % size;
          Q[rear] = x;
    }
}
int circularqueue :: dequeue () {
      int x=-1;
      if(isempty()) cout << "Queue Underflow";
      else {
            front = (front+1) % size;
            x= Q[front];
      }
      return x;
}
void circularqueue :: display() {
      int i = front + 1;
      do {
            cout << Q[i] << flush;
            if( i< rear)  cout << "<" << flush;
            i= (i+1) % size;
      } while (i != (rear+1)% size);
int main() {
    int A[] = {1,3,5,7,9}
    circularqueue cq(6);
    for (int i=0; i<5; i++)   cq.enqueue(A[i]);      // Enqueue
    cq.display();          // Display
    cout<<endl();
    cq.enqueue (10);    // Overflow
    for (int i=0; i<5;i++) cq.dequeue
                                        // Dequeue
    cq.dequeue()   // Underflow
} return 0;
```

☆ Queue Using Linkedlist :-

```
#include <iostream>
using namespace std;

class node {
    public:
        int data;
        node* next;
};

class queue {
  private:
        queue();
        ~queue();
        void enqueue(int x);
        int dequeue();
        bool isempty();
        void display();
};

queue::queue() {
      front = nullptr;
      rear = nullptr;
}

void queue::enqueue(int x) {
     node* t = new node;
     if (t==nullptr)    cout << "Queue Overflow" <<endl;
     else {
         t->data = x;
         t->next = nullptr;
         if (front == nullptr) {
                 front = rear = t;
         }
         else {
             rear->next = t;
             rear = t
         }
     }
}
```

```
int queue :: dequeue () {
    int x=-1;
    node * p;
    if(isEmpty()) {cout << "Queue Underflow";
    else {
        p= front;
        front = front -> next;
        x=p->data;
        delete p;
    }
    return x;
}
bool queue:: isempty() {
    if (front==rear)   return true;
    return false;
}
queue::~queue() {
    node p= front;
    ~~for (while~~
    while (front) {
        front= front->next;
        ~~delete~~
        delete p;
        p=front;
    }
}
void queue:: display() {
    node* p= front;
    while (p) {
        cout << p-> data << flush;
        p=p-> next;
        if(p!=nullptr){
            cout << " <- " << flush;
        }
    }
    cout <<endl;
}
```

```
int main () {
    int A[] = {1,3,5,7,9};
    queue q;
    for(int i=0; i< sizeof(A)/sizeof(A[0]); i++) {
        q.enqueue(A[i]);
    }
    q.display();
    for(int i=0; i<sizeof(A)/ sizeof(A[i]); i++) {
            q.dequeue();
    }
    q.dequeue();    // Underflow;
    return 0;
}
```

★ **Deque**:- Deque or Double Ended Queue is a type of queue in which insertion and removal of elements can either be performed from the front or the rear.
Thus, it does not follow FIFO rule.

Queue

	Insert	Delete
front	✗	✓
rear	✓	✗

DE Queue (Deque)

	Insert	Delete
front	✓	✓
rear	✓	✓

⇒ Types of Deque:-

(1) Input Restricted Deque:- In this deque, input is restricted at a single end but allows deletion at both the ends.

(2) Output Restricted Deque:- In this deque, output is ~~stuf~~ restricted at a single end but allows insertion from both the ends.

Input Restricted Deque

	Insert	Delete
Front	✗	✓
Rear	✓	✓

Output Restricted Deque

	Insert	Delete
Front	✓	✓
Rear	✓	✗

```
#include<iostream>
using namespace std;

class node {
    public:
        int data;
        node* next;
};

class deque {
    private:
        node* front;
        node* rear;
    public:
        deque();
        ~deque();
        bool isfull();
        bool isempty();
        void enqueuefront();
        void enqueuerear();
        int dequeuefront();
        int dequeuerear();
        void display();
};

deque::deque() {
    front=rear=nullptr;
}
```

```
deque::~deque () {
    node *p= front;
    while (p) {
        front = front -> next;
        delete p;
        p= front;
    }
}

bool deque:: isfull () {
    node* p = new node;
    if (p== nullptr) return true;
    delete p;
    return false;
}

bool deque :: isempty() {
    if(front==nullptr) return true;
    return false;
}

void deque:: enqueuefront (int x) {
    if(isfull())   cout << "deque overflow\n";
    else {
        node* t= new node;
        t->data=x;
        t-> next= front;
        if (front==nullptr) rear=t;
        front = t;
    }
}

void deque:: enqueuerear (int x) {
    if (isfull ())   cout << "deque overflow\n";
    else {
        node* t = new node;
        t -> data=x;
        t -> next = nullptr;
```

```
        node* p= front;
        if(front==nullptr)      front= rear=t;
        else {
            rear->next=t;
            rear=t;
        }
    }
}
~~int dequeue~~
int deque :: dequeuefront () {
    int x=-1;
    if(isempty())  cout<<"deque underflow\n";
    else {
        node* p= front;
        front = front->next;
        x=p->data;
        delete p;
        if(front==nullptr) {
            rear= nullptr;
        }
    }
    return x;
}
int deque :: dequeuerear() {
    int x=-1;
    if(isempty())  cout<<"deque underflow\n";
    else {
        node* p = front;
        node* q;
        while (p && p->next!=nullptr) {
            q=p;
            p=p->next;
        }
        if(front==rear) {
            x=rear->data;
            delete rear;
        }   front=rear=nullptr;
```

```
        else {
            x= p->data;
            delete p;
            q-> next= nullptr;
            rear=q;
        }
    }
    return x;
}
void deque :: display () {
    node* p= front;
    while (p) {
        cout <<p-> data;
        p=p->next;
        if(p!= nullptr)  cout << " <- ";
    }
    cout << "\n";
}
int main () {
    int A[] = {1,3,5,7 ,9}
    deque q;
    q. enquenefront (A[0]);
    q.enquenefront(A[1]);
    q.display( );
    q.enquenerear (A [2]);
    q. enquenerear (A[3]);
    q. display( );
    q.dequenefront( );
    q.display( );
    q. dequenerear( );
    q.display( );
    return 0;
}
```