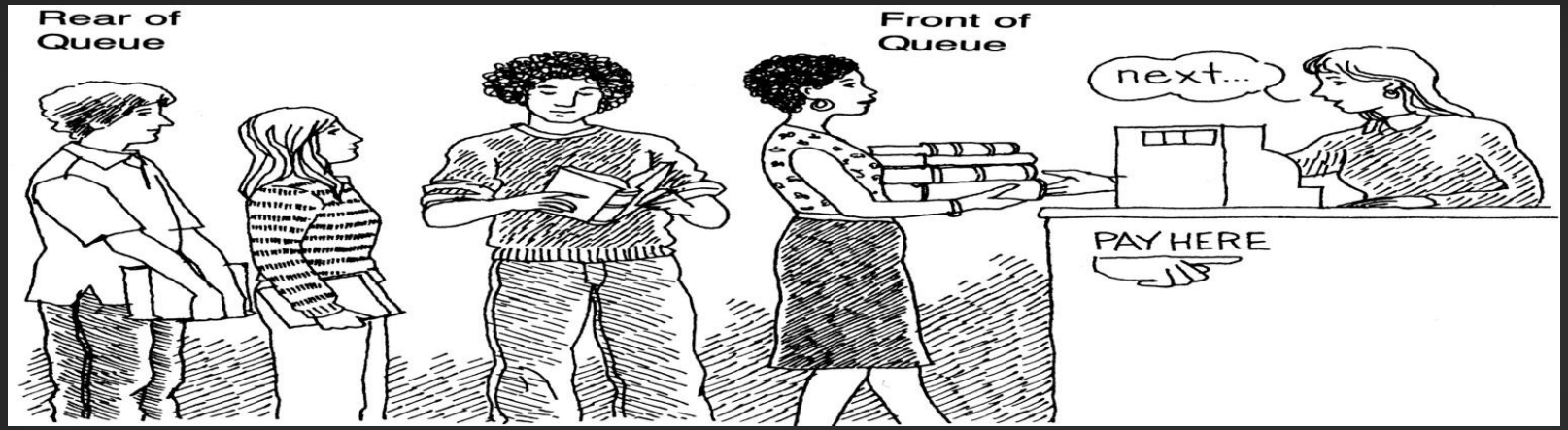
Linear Queue: Array / Linked List Implementation

What is a **Queue**?

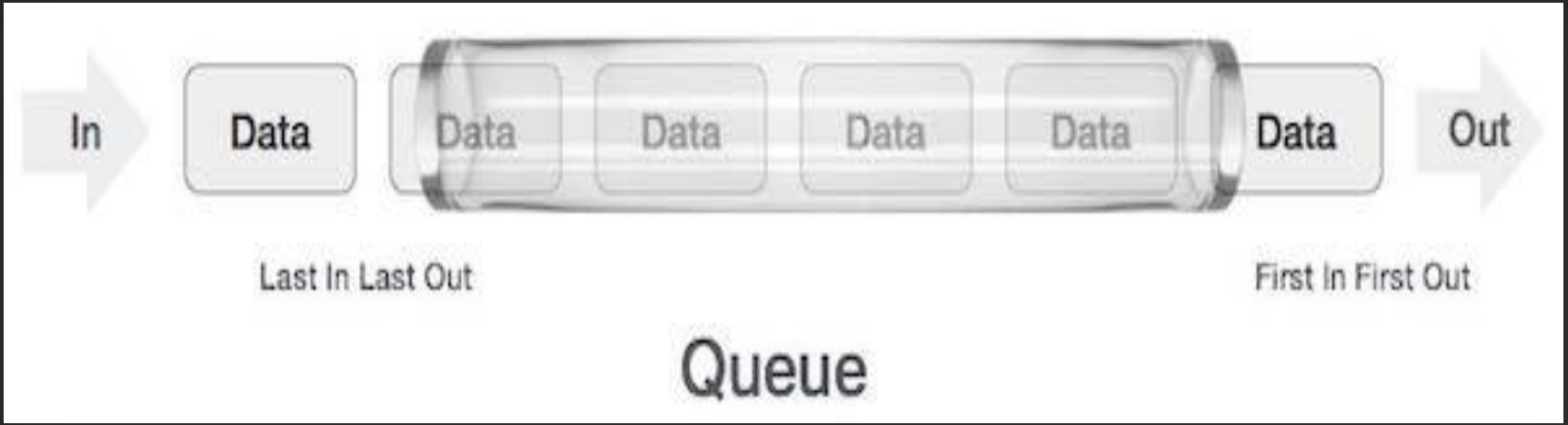
- A queue is abstract data structure somewhat similar to stack, but queue is open at both its ends.
- Queues have two ends:
  - Rear End: This end is always used to insert data(enqueue).
  - Front End: This end is used to remove to data(dequeue).
- The element added first is also removed first (**FIFO**: First In, First Out).
- Implementation: Array, linked list and stack.

- Some Real Life Examples:



Queue Specification

As we now understand that in queue, we access both ends for different reasons. The following diagram given below tries to explain queue representation as data structure



Basic Operations Involve in Queue:

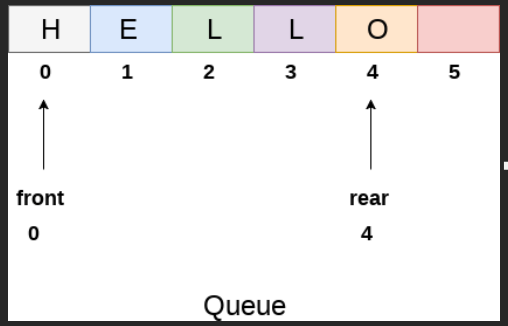
Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory.

- **Enqueue():** Add(store) item to the queue.
- **Dequeue():** Remove(access) item from the queue.
- **Peek():** Gets the element at the front of the queue without removing it.
- **Isfull():** Check if the queue is full.
- **Isempty():** Check if the queue is empty.
- **Front():** Get the front element.
- **Rear():** Get the rear element.

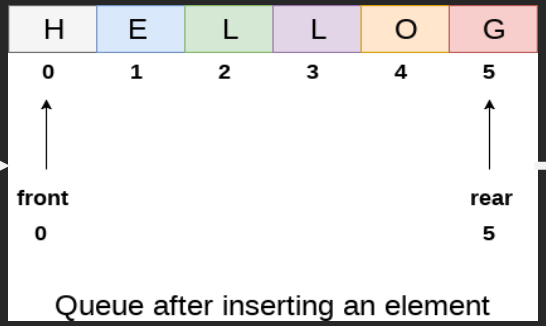
Applications of **Queue**

1. Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
2. In real life scenario, Call Center phone systems uses Queues to hold people calling them in an order, until a service representative is free.
3. Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive i.e. First come first serve.

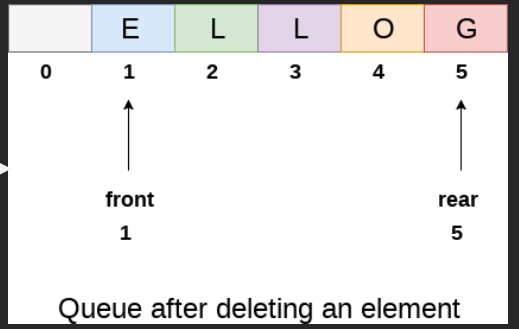
Queue Implementation Using Array:



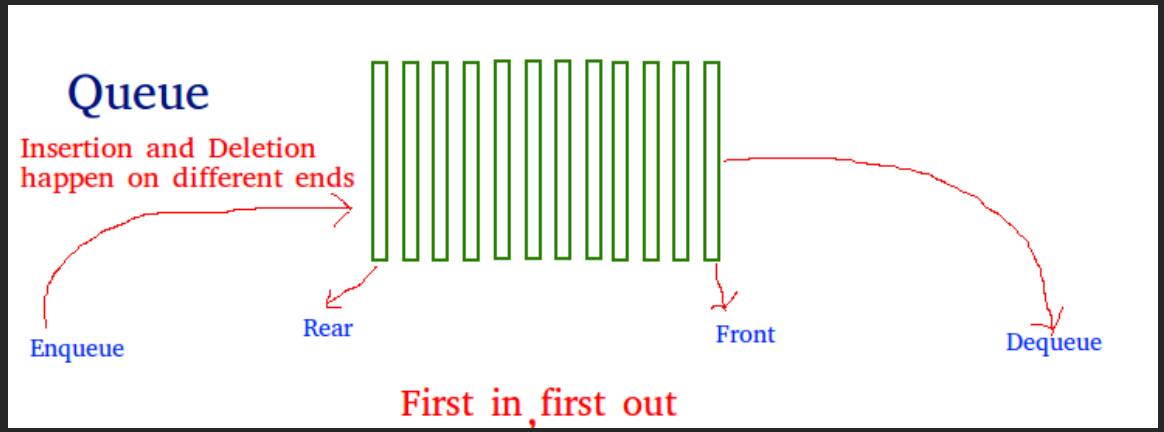
Queue



Queue after inserting an element



Queue after deleting an element



Create Empty Queue using Array

- Define a constant '**SIZE**' with specific value.
- Create a one dimensional array with above defined SIZE (**int queue[SIZE]**).
- Define two integer variables '**front**' and '**rear**' and initialize both with '**-1**'. (**int front = -1, rear = -1**).

Enqueue(value): Inserting value into the Queue

- Check whether **queue** is **FULL**. (**rear == SIZE-1**)
- If it is **FULL**, then display "**Queue is FULL!!! Insertion is not possible!!!**" and terminate the function.
- If it is **NOT FULL**, then increment **rear** value by one (**rear++**) and set **queue[rear] = value**.

Pseudocode:

```
Q [0 : n-1];
front = rear = -1; // Initial values
Enque ( Q, front, rear, item)
{
        if ( rear = = n-1)
        {
                Print : Overflow;
                return;
        }\
        if( front == -1)
                front = 0;
         rear = rear + 1;
        Q[ rear ] = item;
}
```

Dequeue (value): Inserting value into the Queue

- Check whether **queue** is **EMPTY** (**front == rear == -1**).
- If it is **EMPTY**, then display "**Queue is EMPTY!!! Deletion is not possible!!!**" and terminate the function.
- If it is **NOT EMPTY**, then increment the **front**. Then display **queue[front]** as deleted element.
- Then check for (**front == rear**), if it **TRUE**, then set (**front = rear = -1**).

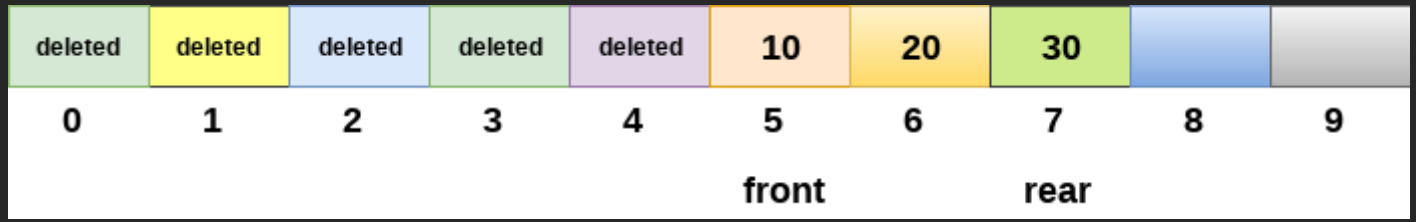
Pseudocode:

```
Q [0 : n-1];
front = rear = -1; // Initial value
Deque ( Q, front, rear)
{
        if ( rear = = -1)
        {
                Print : Underflow;
                return;
        }
        item = Q[ front ];
        front = front + 1;
        if( front == rear)
                front = rear = -1;
}
```

Drawbacks of Queue Using an Array

- It will work for an only fixed number of data values.
- It is not suitable when we don't know the size of data at the beginning.
- The queue implemented using a linked list can work for an unlimited number of values.
- No need to fix the size at the beginning of the implementation.

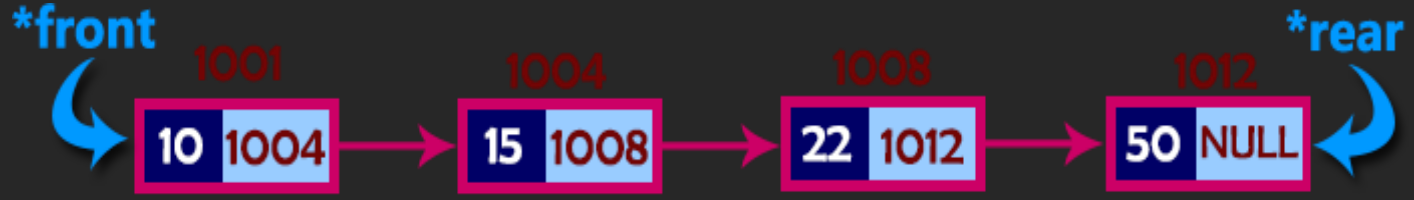
Memory Wastage: The space of the array, which is used to store queue elements, can never be reused to store the elements of that queue

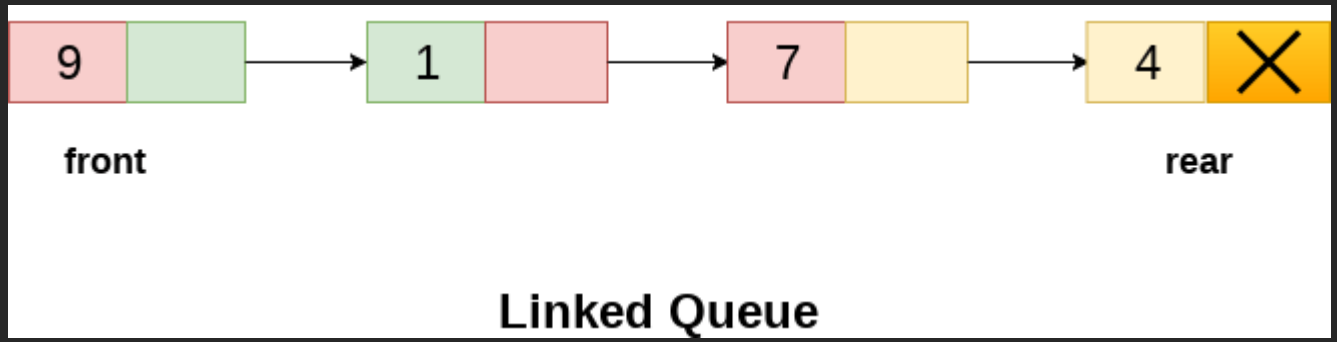


The above figure shows how the memory space is wasted in the array representation of queue. In the above figure, a queue of size 10 having 3 elements, is shown. The value of the front variable is 5, therefore, we cannot reinsert the values in the place of already deleted element before the position of front. That much space of the array is wasted and cannot be used in the future (for this queue).

Deciding the Array Size: One of the most common problem with array implementation is the size of the array which requires to be declared in advance.

Queue using Linked List





- 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'.

Create Empty Queue

- Define a 'Node' structure with two members data and next.
- Define two Node pointers 'front' and 'rear' and set both to NULL.

Enqueue(value): Inserting value into the Queue

- Create a **newNode** with given value and set '**newNode → next**' to **NULL**.
- Check whether queue is **Empty** (**rear == NULL**)
- If it is **Empty** then, set **front = newNode** and **rear = newNode**.
- If it is **Not Empty** then, set **rear → next = newNode** and **rear = newNode**.

Pseudocode:

```
Struct node {
        int data;
        struct node * next = NULL;
};

Struct node * front = NULL, *rear = NULL;

Enque( front, rear, item)

{
    struct node * temp = NEWNODE(); // memory allocated
     if( temp == NULL) {
                Print : “OVERFLOW” ;
                return;
        }
        temp - > data = item;
        temp - > next = NULL;
        rear - > next = temp;
}
```

Dequeue (value): Inserting value into the Queue

- Check whether **queue** is **Empty** (**front == NULL**).
- If it is **Empty**, then display "**Queue is Empty!!! Deletion is not possible!!!**" and terminate from the function
- If it is **Not Empty** then, define a Node pointer '**temp**' and set it to '**front**'.
- Then set '**front = front → next**' and delete '**temp**' **(free(temp)).**

Pseudocode:

```
Struct node {
        int data;
        struct node * next = NULL;
};

Struct node * front = NULL, *rear = NULL;

Deque( front, rear, item) {
        struct node * temp ;
        if( front = = NULL)
        {
                Print : “UNDERFLOW” ;
                return;
        }
  temp = front - > data;
  front = front -> next;
}
```

Complexity Analysis of Queue Operations

- **Enqueue: O(1)**
- **Dequeue: O(1)**

C Program to Demonstrate Enqueue(), Dequeue(), and Display()

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

struct Node
{
    int data;
    struct Node *next;
}*front = NULL, *rear = NULL;

void DeQueue()
{
    if(front==NULL)
    {
        printf("\nt\t\tQueue is Empty!");
    }
```

```
    else
    {
      struct Node* temp = front;
      front = front->next;
      printf("\n\t\tDeleted Element: %d",temp->data);
      free(temp);
    }

}
void EnQueue(int data)
{
   struct Node *newnode = (struct Node*)malloc(sizeof(struct Node));
   newnode->data  = data;
   newnode->next = NULL;
   if(front==NULL)
   {
```

```
        front=rear=newnode;
    }
    else
    {
        rear->next = newnode;
        rear=newnode;
    }
    printf("\n\t\tSuccessfully Inserted!");

}
void Display()
{
  if(front==NULL)
  {
      printf("\n\n\t\tQueue is Empty!");
  }
```

```
  else
  {
      struct Node* temp = front;
      printf("\n\t\tElements in the Queue: ");
      while (temp!=NULL)
      {
          printf("%d ",temp->data);
          temp=temp->next;
      }

  }

}

int main()
{
```

```
    int choice,val;
    while(choice!=4)
    {
printf("\n\n\n\t\t1. EnQueue()\n\t\t2. DeQueue()\n\t\t3. Display()\n\t\t4.
 Exit()");
        printf("\n\t\tEnter Your Choice: ");
        scanf("%d",&choice);
        switch (choice)
        {
        case 1:
           printf("\n\t\tEnter the Value: ");
           scanf("%d",&val);
           EnQueue(val);
           break;
        case 2:
            DeQueue();
```

```
            break;
        case 3:
            Display();
            break;
        case 4:
          exit(0);
        default:
            printf("\n\n\t\tWrong selection!!! Please try again!!!\n");
        }
    }
}
```

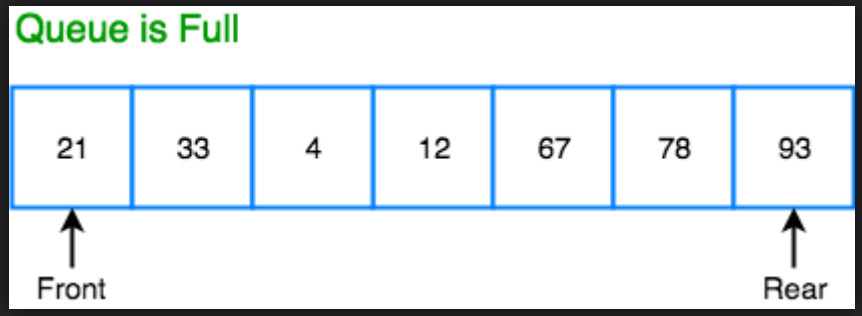
Circular Queue

A circular queue is a linear data structure in which operations are performed based on FIFO (First In First Out) and the last position is connected back to the first position to make a circle. It is also called 'Ring Buffer'.

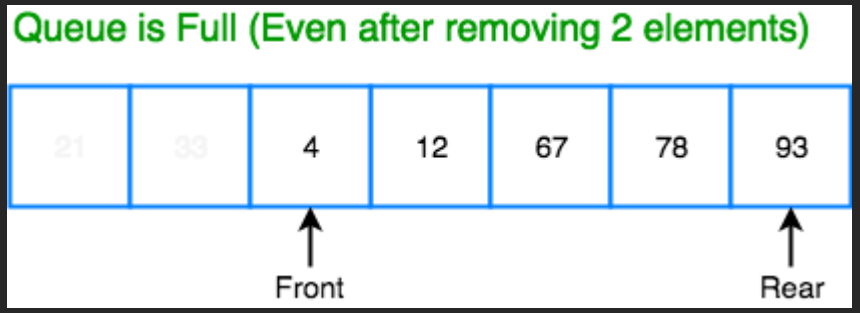
But but before we start to learn more about circular queue, we should first understand, why we need a circular queue, when we already have linear queue data structure.

In a linear queue, once the queue is completely full, it's not possible to insert more elements. Even if we dequeue t he queue to remove some

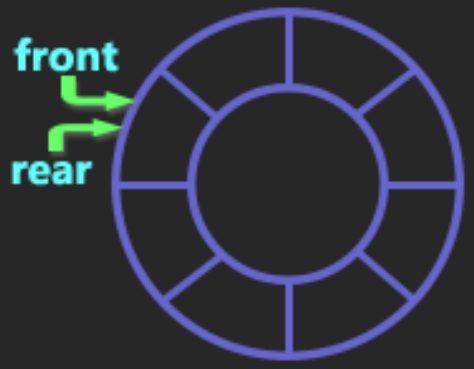
of the elements, until the queue is reset no new elements can be inserted.



When we dequeue any element to remove it from the queue, we are actually moving the front of the queue forward, thereby reducing the overall size of the queue. And we cannot insert new elements, because the rear pointer is still at the end of the 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.



Create Empty Circular Queue Using Array

Step 1: Define a constant 'SIZE' with specific value.

Step 2: **Create a one dimensional array with above defined SIZE (int Queue[SIZE]).**

Step 3: Define two integer variables 'front' and 'rear' and initialize both with '0' (int front = 0, rear = 0).

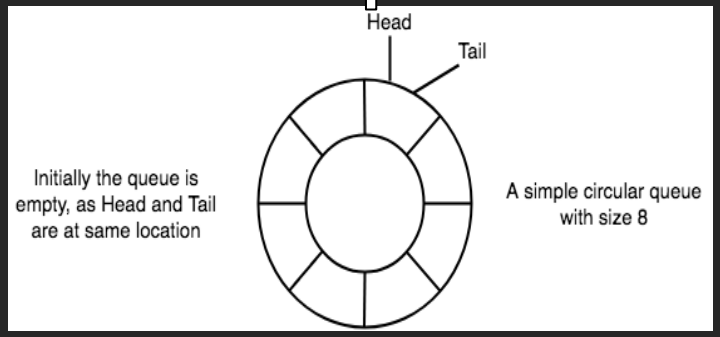
EnQueue(value) - Inserting value into the Circular Queue

1. Check whether queue is Full (front == (rear+1) % size of array).

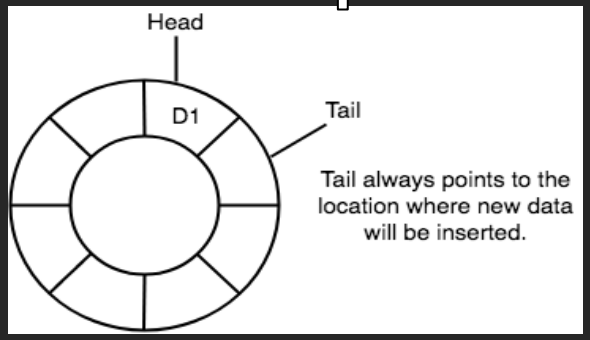
2. If it is Full then display "Queue is FULL!!! Insertion is not possible!!!" and terminate the function.

3. If it is not Full, then set rear = (rear+1) % size of array and queue[rear] = value.

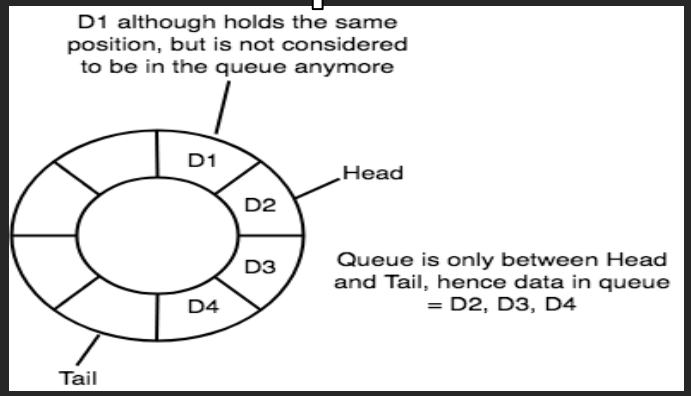
Step 1



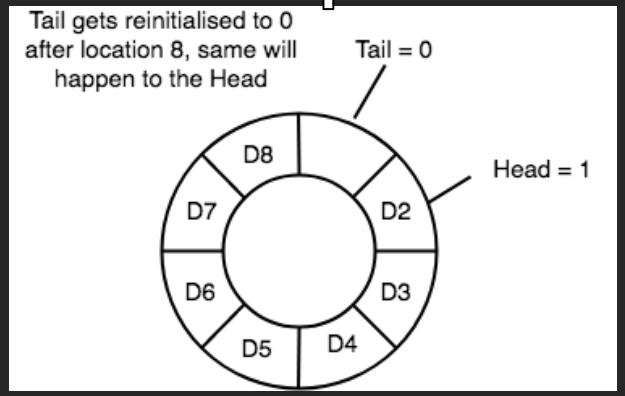
Step 2



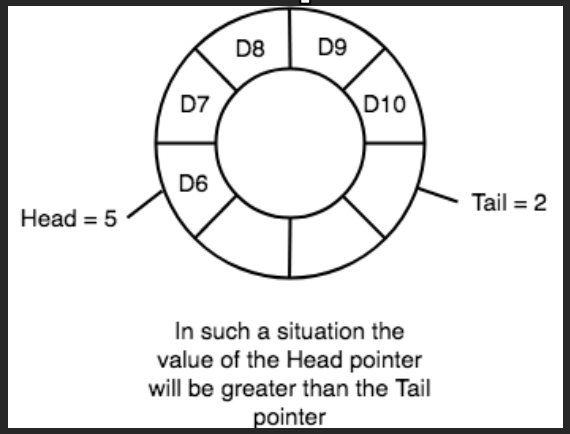
Step 3



Step 4



Step 5



Algorithm for EnQueue

```
Q[0:n-1];
Front = rear = 0;
void EnQueue(Q, front, rear , val)
{
     if(((rear+1)%size == front))
    {

      printf("\n\n\tAlert! Queue is Full");

    }

  else
  {
      rear = (rear+1)%size;
      Q[rear] = val;
      printf("\n\t\tSuccessfully Inserted: %d",val);
  }

}
```

DeQueue(value) - Removing value into the Circular Queue

1. Check whether queue is **Empty** (front == rear).

2. if it **is Empty**, then display 'Queue is Empty! Deletion is not possible!' and terminate the function.

3. if it **is not Empty** then display queue [front+1] as a deleted element and increment the front value by one [front++].

Algorithm for DeQueue

```
Q[0:n-1];
Front = rear = 0;
int DeQueue(Q, front, rear)
 {
 if(rear==front)
    {
      printf("\n\t\tAlert! Queue is Empty");
   }

    else
    {
      front = (front+1)%size;
      Q[front];
      printf("\n\t\tSuccessfully Dequeue: %d",a);
    }

}
```

C Program to Demonstrate Circular Queue Using Array

```c
#include<stdio.h>
#include<stdlib.h>
struct circularQueue
{
    int size;
    int f;
    int r;
    int* arr;
};

int isEmpty(struct circularQueue *q)
{
    if(q->r==q->f)
    {
        return 1;
    }
    return 0;
}
```

```c
int isFull(struct circularQueue *q)
{
    if(((q->r+1)%q->size == q->f))
    {
        return 1;
    }
    return 0;
}
void EnQueue(struct circularQueue *q,int val)
{
  if(isFull(q))
  {
      printf("\n\n\tAlert! Queue is Full");
  }
  else
  {
      q->r = (q->r+1)%q->size;
      q->arr[q->r] = val;
      printf("\n\t\tSuccessfully Inserted: %d",val);
  }

}
```

```
int DeQueue(struct circularQueue *q)
{
    int a=-1;
    if(isEmpty(q))
    {
        printf("\n\t\tAlert! Queue is Empty");
    }
    else
    {
      q->f = (q->f+1)%q->size;
      a=q->arr[q->f];
      printf("\n\t\tSuccessfully Dequeue: %d",a);
    }
    return a;

}
void Display(struct circularQueue *q)
{
    if (isEmpty(q))
    {
        printf("\n\t\tAlert! Queue is Empty");
```

```
    }
    else
    {
        printf("\n\n\t\t\t\tQueue: ");
        int i=q->f;
        i=(i+1)%q->size;
        while(i!=q->r)
        {
            printf("%d ",q->arr[i]);
            i = (i+1)%q->size;
        }
        printf("%d ",q->arr[i]);

    }

}
int main()
{
    struct circularQueue q;
    int choice,val;
    q.size=10;
    q.f=q.r=0;
```

```
    q.arr = (int*)malloc(q.size*sizeof(int));
    system("cls");
        while(choice!=3)
    {

        printf("\n\n\n\t\t1. EnQueue()\n\t\t2. DeQueue()\n\t\t4. Exit()");
        printf("\n\t\tEnter Your Choice: ");
        scanf("%d",&choice);
        switch (choice)
        {
        case 1:
           printf("\n\t\tEnter the Value: ");
           scanf("%d",&val);

           EnQueue(&q,val);
           system("cls");
           Display(&q);
           break;
        case 2:
            DeQueue(&q);
            system("cls");
            Display(&q);
```

```
            break;
        case 3:
          exit(0);
        default:
            printf("\n\n\t\tWrong selection!!! Please try again!!!\n");
        }
    }
    return 0;

}
```

Implementation of Circular Queue Using Linked List

Using Circular Queue is better than normal queue because there is no memory wastage. Linked list provide the facility of dynamic memory allocation so it is easy to create. When we implement circular Queue using linked list it is similar to circular linked list except there is two pointer front and rear in circular Queue where as circular linked list has only one pointer head.

EnQueue

Step 1: Create a struct node type node.

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

Step 3: If Queue is Empty then initialize front and rear from new node.

Step 4: If Queue is not Empty then initialize rear next and rear from new node.

Step 5: New Node next initialize from front.

Algorithm

```
struct node
{
    int data;
    struct node* next;
};

struct node *f = NULL;
struct node *r = NULL;

void EnQueue(int d) //Insert elements in Queue
{
    struct node* n;
    n = (struct node*)malloc(sizeof(struct node)); // Create New Node
    n->data = d; // insert data in data section of new node.
    n->next = NULL; // Insert NULL in next section of new node.
    if((r==NULL)&&(f==NULL))
    {
        f = r = n;
```

```
        r->next = f;
    }
    else
    {
        r->next = n;
        r = n;
        n->next = f;
    }
}
```

DeQueue

Step 1: Check if queue is empty or not.

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

Step 3: Else initialize temp from front.

Step 4: if front is equal to the rear then initialize front and rear from NULL.

Step 5: Free temp memory.

Step 6: If there is more than one node in queue then make front next to front the initialize rear next from front.

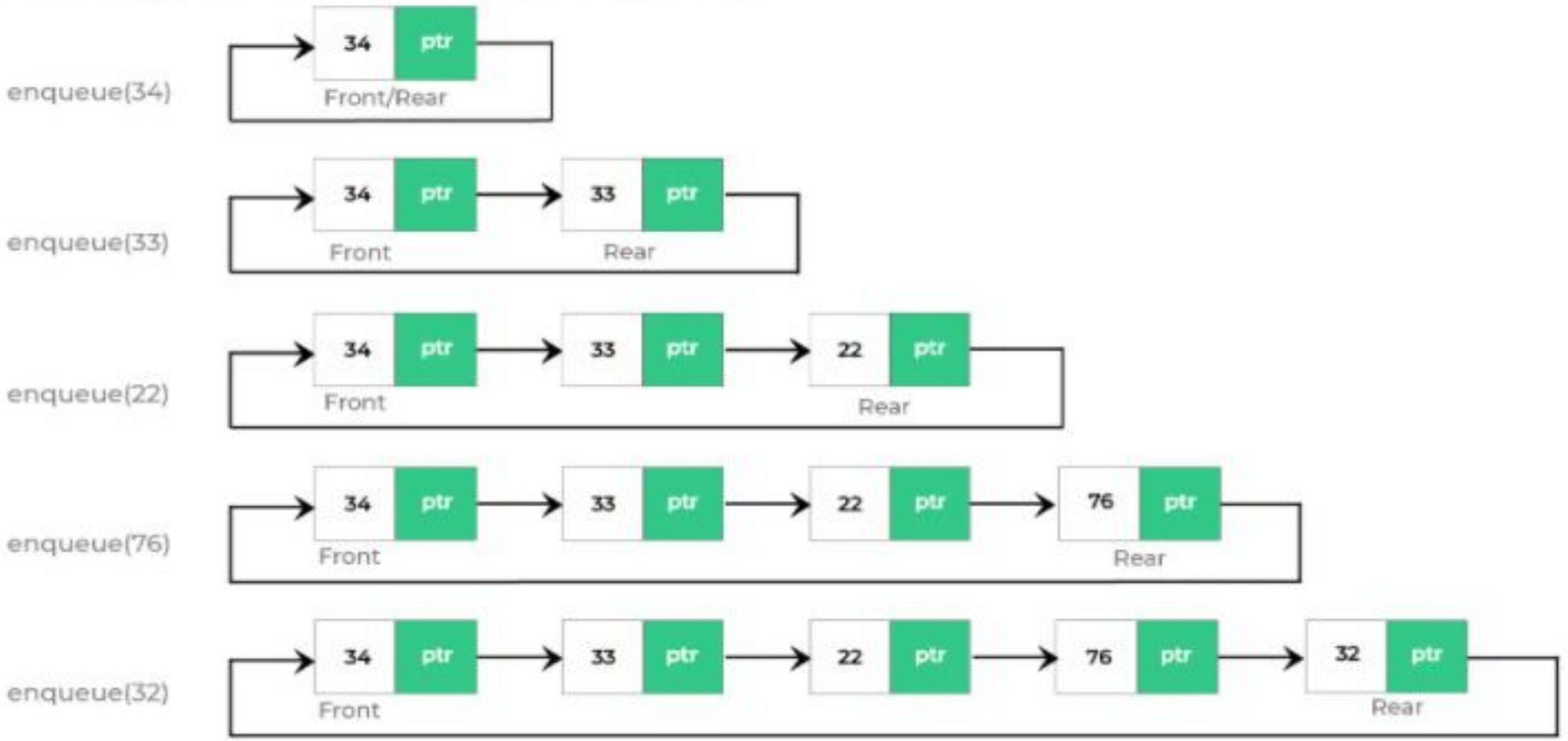
Step 7: Free temp memory.

Algorithm

```c
struct node
{
    int data;
    struct node* next;
};
struct node *f = NULL;
struct node *r = NULL;
void DeQueue() // Delete an element from Queue
{
    struct node* t; // Creating temporary pointer.
    t = f; // Initialize temp pointer with front.
    if((f==NULL)&&(r==NULL))
        printf("\n\t\tAlert! Queue is Empty");
    else if(f == r){
        f = r = NULL;
        free(t); // free temporary memory.
    }
    else{
        f = f->next;
        r->next = f;  free(t);}
```

Implementation of Circular Queues using Linked List in C

Adding the elements into Queue



Removing the elements from Queue

Complexity Analysis of These Circular Queue Operations:

EnQueue(): O(1)

DeQueue(): O(1)

C Program to Demonstrate Circular Queue Using Linked List

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

struct node
{
    int data;
    struct node* next;
```

```
};

struct node *f = NULL;
struct node *r = NULL;

void EnQueue(int d) //Insert elements in Queue
{
    struct node* n;
    n = (struct node*)malloc(sizeof(struct node));
    n->data = d;
    n->next = NULL;
    if((r==NULL)&&(f==NULL))
    {
        f = r = n;
        r->next = f;
    }
    else
    {
        r->next = n;
        r = n;
        n->next = f;
    }
```

```
}
void DeQueue() // Delete an element from Queue
{
    struct node* t;
    t = f;
    if((f==NULL)&&(r==NULL))
        printf("\n\t\tAlert! Queue is Empty");
    else if(f == r){
        f = r = NULL;
        free(t);
    }
    else{
        f = f->next;
        r->next = f;
        free(t);
    }


}
void Display(){ // Print the elements of Queue
    struct node* t;
    t = f;
```

```
    if((f==NULL)&&(r==NULL))
        printf("\n\t\tAlert! Queue is Empty");
    else{
        printf("\n\n\t\t\t\tQueue: ");
        do{
            printf("%d ",t->data);
            t = t->next;
        }while(t != f);
    }
}
int main()
{
    int val,choice;
    system("cls");
        while(choice!=3)
    {

        printf("\n\n\n\t\t1. EnQueue()\n\t\t2. DeQueue()\n\t\t4. Exit()");
        printf("\n\t\tEnter Your Choice: ");
        scanf("%d",&choice);
        switch (choice)
        {
```

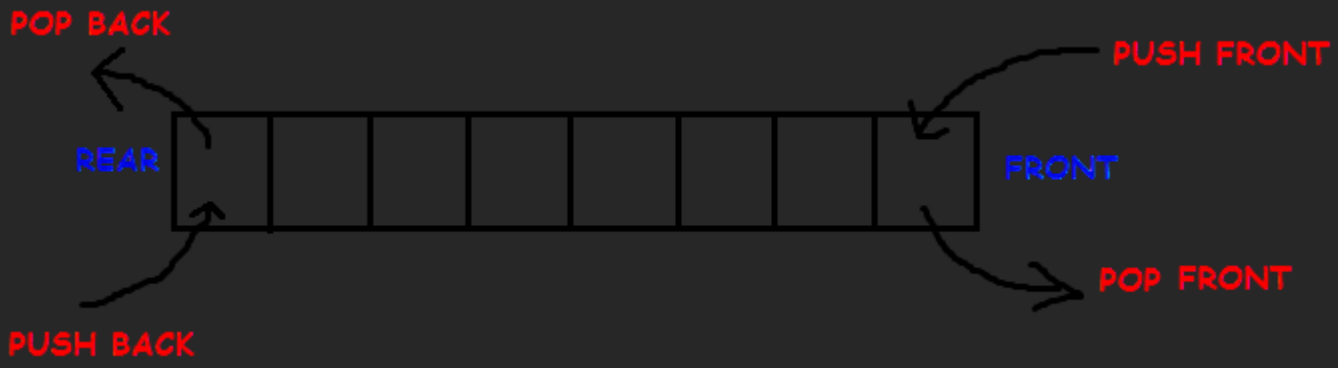
```
        case 1:
           printf("\n\t\tEnter the Value: ");
           scanf("%d",&val);

           EnQueue(val);
           system("cls");
           Display();
           break;
        case 2:
            DeQueue();
            system("cls");
            Display();
            break;
        case 3:
          exit(0);
        default:
            printf("\n\n\t\tWrong selection!!! Please try again!!!\n");
        }
    }
    return 0;
}
```

Double Ended Queue: Array / Linked List Implementation

In Double Ended Queue (**DQueue**), insertion and deletion operation are performed at both the ends (Front and Rear).

That means we can **insert** at both front and rear positions and can **delete** from both front and rear position.

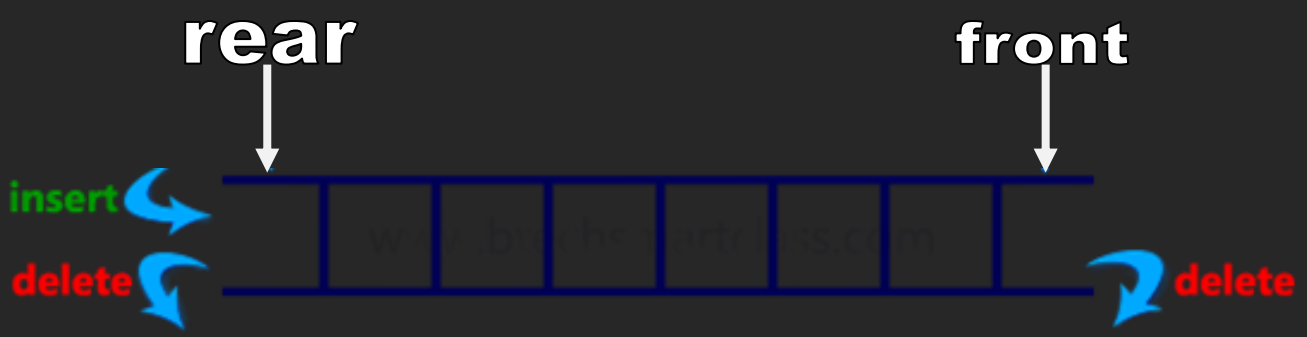


Types of Double Ended Queue

1. Input Restricted Double Ended Queue
2. Output Restricted Double Ended Queue

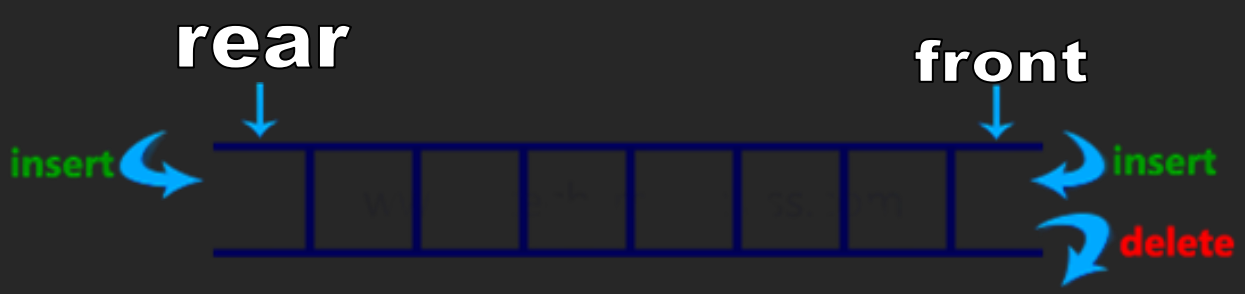
Input Restricted DeQueue

In this DeQueue, input is restricted at single end but allows deletion at both the ends.



Output Restricted DeQueue

In this DeQueue, Output is Restricted at a single end but allows insertion at both the ends.



Operations On Double Ended Queue

- ✓ **InsertFront() –** Add an item at the front of DeQueue
- ✓ **InsertRear() -** Add an item at the rear of DeQueue
- ✓ **DeleteFront() -** Delete an item front of DeQueue
- ✓ **DeleteRear() -** Delete an item from front of DeQueue

Create Empty Double Ended Queue Using Circular Array

1. Define a constant '**SIZE**' with specific value.
2. Create a one dimensional Array with above defined size, int **queue[SIZE].**
3. Define two variables '**front**' and '**rear**' and initialize both with '0' i.e. front = 0, rear = 0.

InsertFront: Pseudo Code

```
#define SIZE 10
int queue[SIZE];
int front = 0, rear = 0;
void InsertFront(int val)
{
    if (front - (rear) == SIZE)
    {
        printf("\nAlert! Queue is Full!");
        return;
    }
    else
    {

        rear--;
        queue[rear] = val;
    }
}
```

InsertRear: Pseudo Code

```
#define SIZE 10
int queue[SIZE];
int front = 0, rear = 0;
void InsertRear(int val)
{
    if (front - (rear) == SIZE)
    {
        printf("\nAlert! Queue is Full!");
        return;
    }
    else
    {
        queue[front] = val;
        front++;
    }
}
```

DeleteFront: Pseudo Code

```
#define SIZE 10
int queue[SIZE];
int front = 0, rear = 0;
int DeleteFront()
{
    if (front == rear)
    {
        printf("\nQueue is Empty!!! Deletion is not possible!!!");
        return 0;
    }
    rear++;
}
```

DeleteRear: Pseudo Code

```
#define SIZE 10
int queue[SIZE];
int front = 0, rear = 0;
int DeleteRear()
{
    int del;
    if (front == rear)
    {
        printf("\nAlert! Queue is Empty!");
        return 0;
    }
    else
    {
        front--;
    }
}
```

Double Ended Queue using Singly Linked List

1. Declare two pointers ‘front’ and ‘rear’ of type node, where node represents the structure of node of singly linked list.

2. Initialize both of them with NULL.

InsertFront: Pseudo Code

```
struct node
{
    int data;
    struct node *next;
} *ptr = NULL, *temp = NULL, *rear = NULL, *front = NULL;
```

```c
void InsertFront(int item)
{
    struct node *node = NewNode(); // Memory Allocated
    node->data = item;
    if (front == NULL)
    {
        front = node;
        rear = node;
        return;
    }
    node->next = front;
    front = node;
}
```

InsertRear: Pseudo Code

```
struct node
{
    int data;
    struct node *next;
} *ptr = NULL, *temp = NULL, *rear = NULL, *front = NULL;

void InsertRear(int item) //Insertion at the end
{
    struct node *node = NewNode(); // Memory Allocated
    node->data = item;
    if (front == NULL)
    {
        front = node;
        rear = node;
    }
    else
```

```
    {
        rear->next = node;
        rear = node;
    }
}
```

DeleteFront: Pseudo Code

```
struct node
{
    int data;
    struct node *next;
} *ptr = NULL, *temp = NULL, *rear = NULL, *front = NULL;

void Deletefront() //Delete at the beginning
{
```

```
    int item;
    if (front == NULL)
    {
        printf("\nAlert! Queue Underflow or Empty ");
        exit(1);
    }
    temp = front;
    item = temp->data;
    front = front->next;
    free(temp);
}
```

DeleteRear: Pseudo Code

```c
struct node
{
    int data;
    struct node *next;
} *ptr = NULL, *temp = NULL, *rear = NULL, *front = NULL;

void DeleteRear() // Delete at the end
{
    struct node *temp;
    ptr = front;
    if (front == rear)
    {
        free(front);
        front = NULL;
        rear = NULL;
    }

    while (ptr->next != rear)
    {
```

```
        ptr = ptr->next;
    }
    temp = rear;
    rear = ptr;
    ptr->next = NULL;
    free(temp);
}
```

Complexity Analysis: Double Ended Queue Operations

Insertion: O(1)

Deletion: O(1)

Applications: Double Ended Queue

- ✓ In Undo-Redo operations on software.
- ✓ To store history in browser's.
- ✓ Palindrome Checker.

C Program to Demonstrate DeQueue Operations Using Array

```c
#include <stdio.h>
#include <stdlib.h>
#define SIZE 10
int queue[SIZE];
int front = 0, rear = 0;
```

```c
void enqueuerear(int val)
{
    if (front - (rear) == SIZE)
    {
        printf("\n\n\t\t\tAlert! Queue is Full!");
        return;
    }
    else
    {
        queue[front] = val;
        front++;
    }
}

void enqueuefront(int val)
{
    if (front - (rear) == SIZE)
    {
        printf("\n\n\t\t\tAlert! Queue is Full!");
        return;
    }
    else
    {

        rear--;
        queue[rear] = val;
```

```
    }
}

int dequeuerear()
{
    int del;
    if (front == rear)
    {
        printf("\n\n\t\tAlert! Queue is Empty!");
        return 0;
    }
    else
    {
        front--;
    }
}

int dequeuefront()
{
    if (front == rear)
    {
        printf("\nQueue is Empty!!! Deletion is not possible!!!");
        return 0;
    }
    rear++;
}
```

```c
void Display()
{
    int i;
    if (front == rear)
    {
        printf("\n\n\t\tAlert! Queue is Empty");
    }
    else
    {
        printf("\n\n\t\t\tQueue: ");
        for (i = rear; i < front; i++)
        {
            printf("%d ", queue[i]);
        }
    }
}
int main()
{
    int choice, val;
    system("cls");
    while (choice != 5)
    {

        // printf("\n\n\t\tOu");
        printf("\n\n\t\t1. Insert at Rear\n\t\t2. Insert at Front\n\t\t3. Delete at Rear\n\t\t4. Dele
te at Front\n\t\t5. Exit");
```

```
        printf("\n\n\t\tEnter Your Choice: ");
        scanf("%d", &choice);
        switch (choice)
        {
        case 1:
            printf("\n\t\tEnter The Value: ");
            scanf("%d", &val);
            system("cls");
            enqueuerear(val);
            Display();
            break;
        case 2:
            printf("\n\t\tEnter The Value: ");
            scanf("%d", &val);
            system("cls");
            enqueuefront(val);
            Display();
            break;
        case 3:
            dequeuerear();
            system("cls");
            Display();
            break;
        case 4:
            dequeuefront();
            system("cls");
```

```
            Display();
            break;
        case 5:
            exit(0);
        default:
            printf("\n\t\tWrong Choice! Please Select Again");
            break;
        }
    }
    return 0;
}
```

C Program to Demonstrate DeQueue Operations Using Linked List

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

struct node
{
    int data;
```

```
    struct node *next;
} *ptr = NULL, *temp = NULL, *rear = NULL, *front = NULL;

struct node *NewNode()
{
    struct node *node;
    node = (struct node *)malloc(sizeof(struct node));
    node->data;
    node->next = NULL;
    if (node != NULL)
        return node;
    else
    {
        printf("\n\nt\t\tAlert! Unable to Create New Node");
        exit(1);
    }
}

void Enqueuerear(int item) //Insertion at the end
{
    struct node *node = NewNode();
    node->data = item;
    if (front == NULL)
    {
        front = node;
        rear = node;
```

```
    }
    else
    {
        rear->next = node;
        rear = node;
    }
}

void Enqueuefront(int item)
{
    struct node *node = NewNode();
    node->data = item;
    if (front == NULL)
    {
        front = node;
        rear = node;
        return;
    }
    node->next = front;
    front = node;
}

void Deletefront() //Delete at the beginning
{
    int item;
    if (front == NULL)
```

```
    {
        printf("\n\n\t\tAlert! Queue Underflow or Empty ");
        exit(1);
    }
    temp = front;
    item = temp->data;
    front = front->next;
    free(temp);
}

void DeleteRear() // Delete at the end
{
    struct node *temp;
    ptr = front;
    if (front == rear)
    {
        free(front);
        front = NULL;
        rear = NULL;
    }

    while (ptr->next != rear)
    {
        ptr = ptr->next;
    }
    temp = rear;
```

```
    rear = ptr;
    ptr->next = NULL;
    free(temp);
}

void Display()
{
    if (front == NULL)
    {
        printf("\n\n\t\tAlert! Queue is Empty\n");
        return;
    }
    ptr = front;
    printf("\n\n\t\tQueue: ");
    while (ptr)
    {
        printf("%d ", ptr->data);
        ptr = ptr->next;
    }
}

int main()
{
    int data, choice;
    system("cls");
    while (choice != 5)
```

```
    {
        printf("\n\n\t\t1. Insert at Rear\n\t\t2. Insert at Front\n\t\t3. Delete at Rear\n\t\t4. Dele
te at Front\n\t\t5. Exit");
        printf("\n\n\t\tEnter Your Choice: ");
        scanf("%d", &choice);
        switch (choice)
        {
        case 1:
            printf("\n\t\tEnter the Value: ");
            scanf("%d", &data);
            Enqueuerear(data);
            system("cls");
            Display();
            break;
        case 2:
            printf("\n\t\tEnter the Value: ");
            scanf("%d", &data);
            Enqueuefront(data);
            system("cls");
            Display();
            break;
        case 3:
            DeleteRear();
            system("cls");
            Display();
            break;
```

```
        case 4:
            Deletefront();
            system("cls");
            Display();
            break;
        case 5:
            exit(0);
        default:
            printf("\n\n\t\tAlert! Invalid Choice ");
            break;
        }
    }
    return 0;
}
```

Priority Queue

Implementation Using Array / Linked List

A priority Queue in C Programming data structure is a special type of queue in which deletion and insertion perform on the basis of priority of data element. There can be min priority queue and max priority queue. In real word example when a VIP comes for a service he will get the service before the other people standing in a Queue because he has the priority over other people.

In Priority Queue data who has highest priority remove from the queue first and second highest priority element after it and so on. In priority queue each element has its own priority. If priority is same for two elements the data removes on the basis of first come first serve.

Types of Priority Queue:

Min Priority Queue:

In min priority Queue minimum number value gets the highest priority and maximum number of values gets the minimum priority.

Max Priority Queue:

Max priority Queue is the opposite of min priority Queue in it maximum number value gets the highest priority and minimum number of value gets the minimum priority.

Priority Queue Using Array

Priority Queue Can be Implemented in Two ways

Using Ordered Array

In ordered array Insertion or Enqueue operation takes **O(n)** time complexity because it enters elements in sorted order in queue, and Deletion takes **O(1)** time complexity.

Using UnOrdered Array

In unordered array Deletion takes **O(n)** time complexity because it search for the element in Queue for the deletion and EnQueue takes **o(1)** time complexity.

Steps For Implementing Priority Queue

Insertion:

Step 1: Ask the data and its priority from the user.

Step 2: If front is equal to 0 and rear is equal to n-1 then Queue is full.

Step 3: Else if front is equal to the -1 them queue is empty so we have initialize front and rear with 0.

Step 4: Insert the data entered by user in Queue using rear.

Step 5: If rear is equal to n-1 and front is not equal to 0 then there is empty space in queue before the front for using that space we will shift the elements of the queue with the help of front and rear.

Step 6: Insert the data in the queue before at the position

where given priority is greater than priority of the element in queue.

Pseudo Code: Max Priority Queue (Insertion)

```
int Queue[N], Pr[N];
int r = -1, f = -1;

void enqueue(int data, int p) //EnQueueueueue function to insert data
and its priority in Queueueueue
{
    int i;
    if ((f == 0) && (r == N - 1)) //Check if Queueueueue is full
```

```
        printf("\n\t\tAlert! Queue is Full");
    else
    {
        if (f == -1) //if Queue is empty
        {
            f = r = 0;
            Queue[r] = data;
            Pr[r] = p;
        }
        else if (r == N -
1) //if there there is some elemets in Queue
        {
            for (i = f; i <= r; i++)
            {
                Queue[i - f] = Queue[i];
                Pr[i - f] = Pr[i];
            }
            f = 0;
            r = r - f;
```

```
            for (i = r; i >= f; i--)
            {
                if (p > Pr[i])
                {
                    Queue[i + 1] = Queue[i];
                    Pr[i + 1] = Pr[i];
                }
                else
                    break;
            }
            Queue[i + 1] = data;
            Pr[i + 1] = p;
            r++;
        }

        else
        {
            for (i = r; i >= f; i--)
            {
```

```
                if (p > Pr[i])
                {
                    Queue[i + 1] = Queue[i];
                    Pr[i + 1] = Pr[i];
                }
                else
                    break;
            }
            Queue[i + 1] = data;
            Pr[i + 1] = p;
            r++;
        }
    }
}
```

Deletion:

Step 1: Remove the element and the priority from the front of the queue.

Step 2: Increase front with 1.

Pseudo Code: Max Priority Queue (Deletion)

```
int Queue[N], Pr[N];
int r = -1, f = -1;

int dequeue() //remove the data from front
{
    if (f == -1)
```

```
    {
        printf("\n\t\tAlert! Queue is Empty\n");
    }
    else
    {
        if (f == r)
        {
            f = r = -1;
        }
        else
            f++;
    }
}
```

Priority Queue Using Linked List

Implementation of priority Queue data structure by using linked list is efficient as compare to using arrays.

Linked list provide you the facility of Dynamic memory allocation. If we use Linked List for implementing Priority Queue then memory is not going to waste. We can free the memory which is not in use anymore.

For Implementing Priority Queue

Insertion:

Step 1: In EnQueue function we will create a node and assign the data and priority in it.

Step 2: Then we will check if front is null then we assign the new node address to front.

Step 3: If front is pointing to some node then we traverse the linked list using temporary pointer till

new node priority is highest to the priority of the data in linked list.

Step 4: Address of new node initialize to the temporary node and address of temporary next node assign to the new node link.

Pseudo Code: Insertion

```c
struct node
{
    int data, p;
    struct node *next;
};
struct node *f = NULL; // Initialize front pointer with NULL
struct node *r = NULL; // Initialize rear pointer with NULL

void enqueue(int d, int pr) // Insert data to the Linked List
{
    struct node *temp;
    struct node *new_n;
    new_n = (struct node *)malloc(sizeof(struct node));
    new_n->data = d;
    new_n->p = pr;
```

```
    if (f == NULL || pr > f->p)
    {
        new_n->next = f;
        f = new_n;
    }
    else
    {
        temp = f;
        while ((temp->next != NULL) && ((temp->next->p) >= pr))
        {
            temp = temp->next;
        }
        new_n->next = temp->next;
        temp->next = new_n;
    }
}
```

Deletion:

Step 1: Check if queue is empty or not.
Step 2: If queue is empty then DeQueue is not possible.
Step 3: Else store front in temp.
Step 4: Print data of temp.
Step 5: Free temp memory.
Step 6: And make next of front as front.

Pseudo Code: Deletion

```c
struct node
{
    int data, p;
    struct node *next;
};
struct node *f = NULL; // Initialize front pointer with NULL
struct node *r = NULL; // Initialize rear pointer with NULL
void dequeue()  // Remove element from the Linked List
{
    struct node *temp;
    if (f == NULL)
        printf("\n\t\tAlert! Queue is Empty");
    else
    {
        temp = f;
        f = f->next;
        free(temp);
    }
}
```

C Program for Priority Queue Using Array

```c
#include <stdio.h>
#include <stdlib.h>
#define N 20
int Queue[N], Pr[N];
int r = -1, f = -1;

void enqueue(int data, int p)//
EnQueue function to insert data and its priority in Queue
{
    int i;
    if ((f == 0) && (r == N - 1)) // Check if Queue is Full
        printf("\n\t\tAlert! Queue is Full");
```

```
    else
    {
        if (f == -1) // If Queue is Empty
        {
            f = r = 0;
            Queue[r] = data;
            Pr[r] = p;
        }
        else if (r == N - 1) // If there are Some vacant Spaces before front
variable
        {
            for (i = f; i <= r; i++)
            {
                Queue[i - f] = Queue[i];
                Pr[i - f] = Pr[i];
            }
            f = 0;
            r = r - f;
            for (i = r; i >= f; i--)
            {
                if (p > Pr[i])
                {
                    Queue[i + 1] = Queue[i];
```

```
                Pr[i + 1] = Pr[i];
            }
            else
                break;
        }
        Queue[i + 1] = data;
        Pr[i + 1] = p;
        r++;
    }

    else
    {
        for (i = r; i >= f; i--)
        {
            if (p > Pr[i])
            {
                Queue[i + 1] = Queue[i];
                Pr[i + 1] = Pr[i];
            }
            else
                break;
        }
        Queue[i + 1] = data;
```

```
            Pr[i + 1] = p;
            r++;
        }
    }
}
void display() // Print the Data of the Queue
{
    int i;

    if (f == -1)
    {
        printf("\n\t\tAlert! Queue is Empty\n");
        return;
    }
    printf("\n\t\tQueue:    ");
    for (i = f; i <= r; i++)
    {
        printf("%d ", Queue[i]);
    }
    printf("\n\n\t\tPriority: ");
    for (i = f; i <= r; i++)
    {
        printf("%d ", Pr[i]);
```

```
    }
int dequeue() // Remove the Data from the Queue According to their Priority
{
    if (f == -1)
    {
        printf("\n\t\tAlert! Queue is Empty\n");
    }
    else
    {
        if (f == r)
        {
            f = r = -1;
        }
        else
            f++;
    }
}
int main()
{
    int opt, n, i, data, p;
    system("cls");
    while (opt != 3)
    {
```

```
        printf("\n\n\t\t1. Insert The Data\n\t\t2. Delete The Data\n\t\t3. Exit");
        printf("\n\n\t\tEnter Your Choice: ");
        scanf("%d", &opt);
        switch (opt)
        {
        case 1:
            printf("\n\t\tEnter The Data: ");
            scanf("%d", &data);
            printf("\n\t\tEnter The Priority: ");
            scanf("%d", &p);
            enqueue(data, p);  // Calling Function EnQueue
            system("cls");
            display();  // Printing the Data
            break;
        case 2:
            system("cls");
            dequeue();
            display();
            break;
        case 3:
            exit(0);
        default:
            system("cls");
```

```
            printf("\n\t\tAlert! Incorrect Choice");
        }
    }
    return 0;
}
```

C Program for Priority Queue Using Linked List

```
#include <stdio.h>
#include <stdlib.h>
struct node
{
    int data, p;
    struct node *next;
};
struct node *f = NULL; // Initialize front pointer with NULL
struct node *r = NULL; // Initialize rear pointer with NULL
```

```
void enqueue(int d, int pr) // Insert data to the Linked List
{
    struct node *temp; // Create Temp pointer
    struct node *new_n;
    new_n = (struct node *)malloc(sizeof(struct node)); // Create new node
    new_n->data = d;  // Insert data to the data section of new node
    new_n->p = pr;  // Insert Priority to the Priority section of the new node

    if (f == NULL || pr > f-
>p)  // Check if front is NULL or Priority of element to be inserted has higher pr
iority than element present in list.
    {
        new_n->next = f;
        f = new_n;
    }
    else
    {
        temp = f;
        while ((temp->next != NULL) && ((temp->next-
>p) >= pr)) // Check if Priority of element to be inserted has priority in between
 the list ot not
        {
```

```
            temp = temp->next;
        }
        new_n->next = temp->next;
        temp->next = new_n;
    }
}
void display() // Display data of the linked list
{
    struct node *temp = f;
    if (f == NULL)
    {
        printf("\n\t\tAlert! Queue is Empty");
        return;
    }
    printf("\n\n\t\tQueue:    ");
    while (temp != NULL)
    {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    temp = f;
    printf("\n\n\t\tPriority: ");
    while (temp != NULL)
```

```
    {
        printf("%d ", temp->p);
        temp = temp->next;
    }
}
void dequeue()  // Remove element from the Linked List
{
    struct node *temp;
    if (f == NULL)
        printf("\n\t\tAlert! Queue is Empty");
    else
    {
        temp = f;
        f = f->next;
        free(temp);
    }
}
int main()
{
    int opt, n, i, data, p;
    system("cls");
    while (opt != 3)
    {
```

```
        printf("\n\n\t\t1. Insert The Data\n\t\t2. Delete The Data\n\t\t3. Exit");
        printf("\n\n\t\tEnter Your Choice: ");
        scanf("%d", &opt);
        switch (opt)
        {
        case 1:
            printf("\n\t\tEnter The Data: ");
            scanf("%d", &data);
            printf("\n\t\tEnter The Priority: ");
            scanf("%d", &p);
            enqueue(data, p);
            system("cls");
            display();
            break;
        case 2:
            system("cls");
            dequeue();
            display();
            break;
        case 3:
            exit(0);
        default:
            system("cls");
```

```
            printf("\n\t\tAlert! Incorrect Choice");
        }
    }
    return 0;
}
```