

Queues

A) Simple Queue implementation using Linked List

THEORY:

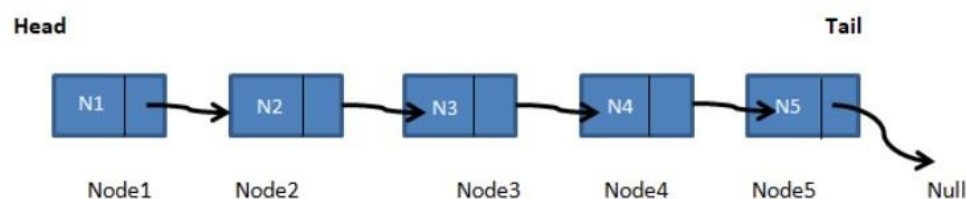
The array implementation cannot be used for the large-scale applications where the queues are implemented. One of the alternatives of array implementation is linked list implementation of queue.

The storage requirement of linked representation of a queue with n elements is $O(n)$ while the time requirement for operations is $O(1)$.

In a linked queue, each node of the queue consists of two parts i.e. data part and the link part. Each element of the queue points to its immediate next element in the memory.

In the linked queue, there are two pointers maintained in the memory i.e. front pointer and rear pointer. The front pointer contains the address of the starting element of the queue while the rear pointer contains the address of the last element of the queue.

Insertion and deletions are performed at rear and front end respectively. If front and rear both are NULL, it indicates that the queue is empty.



Algorithm :-

Insert operation :-

Step 1 :- Allocate the space for the new node PTR

Step 2 :- SET PTR -> DATA = VAL

Step 3 :- IF FRONT = NULL

SET FRONT = REAR = PTR

SET FRONT -> NEXT = REAR -> NEXT = NULL

ELSE

SET REAR -> NEXT = PTR

SET REAR = PTR

SET REAR -> NEXT = NULL

[END OF IF]

Step 4 :- END

Deletion operation :-

Step 1 :- IF FRONT = NULL

Write " Underflow "

Go to Step 5

[END OF IF]

Step 2 :- SET PTR = FRONT

Step 3 :- SET FRONT = FRONT -> NEXT

Step 4 :- FREE PTR

Step 5 :- END

SOURCE CODE:

```
#include <bits/stdc++.h>
using namespace std;

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

struct node *front = NULL;
struct node *rear = NULL;
struct node *temp;

void Insert(int val)
{
    cout<<"Inserting \n"<<endl;

    if(rear==NULL)
    {
        rear = new node;
        rear -> next = NULL;
        rear -> data = val;
        front = rear;
    }
    else
    {
        temp = new node;
        rear -> next = temp;
        temp -> data = val;
        temp -> next = NULL;
        rear = temp;
    }
}
```

```
void Delete()
{
    temp = front;
    if(front == NULL)
    {
        cout<<"Underflow \n"<<endl;
        return;
    }
    else
    {
        if(temp->next != NULL)
        {
            temp = temp -> next;
            cout<<front->data<<endl;
            free(front);
            front = temp;
        }
        else
        {
            cout<<"Element deleted from queue is: "<<front->data<<"\n"<<endl;
            free(front);
            front = NULL;
            rear = NULL;
        }
    }
}

void Display()
{
    temp = front;
    if((front == NULL) && (rear == NULL))
    {
        cout<<"Queue is Empty \n"<<endl;
        return;
    }
    cout<<"Queue Elements are \n";
    while(temp!=NULL)
    {
        cout<<temp->data<<" ";
        temp = temp -> next;
        cout<<"\n";
    }
}

int main()
{
    int choice;
    int val;
```

```
cout<<"QUEUE OPERATIONS USING LINKED LIST"<<endl;
cout<<"\n-----"<<endl;
cout<<"\n 1.ENQUEUE\n 2.DEQUEUE\n 3.DISPLAY\n 4.EXIT"<<endl;
do
{
    cout<<"Enter the Choice: \n"<<endl;
    cin>>choice;
    switch(choice)
    {
        case 1:
        {
            cout<<"Enter value to be ENQUEUE: \n"<<endl;
            cin>>val;
            Insert(val);
            break;
        }
        case 2:
        {
            Delete();
            break;
        }
        case 3:
        {
            Display();
            break;
        }
        case 4:
        {
            cout<<"Exit \n"<<endl;
            break;
        }
        default:
        {
            cout<<"Invalid Input!, Please Enter a Valid Choice(1/2/3/4) \n"<<endl;
        }
    }
}
while(choice!=4);

return 0;
}
```

OUTPUT:

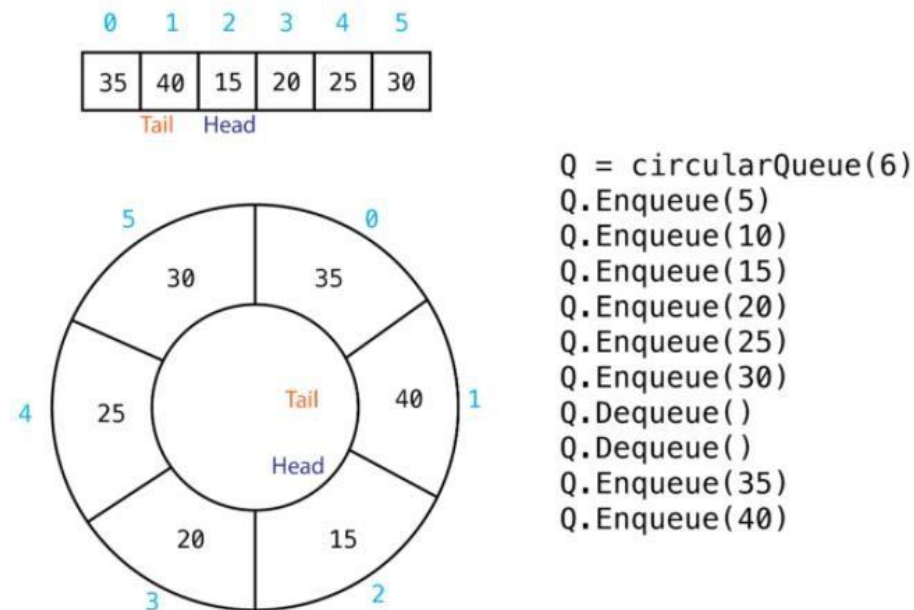
```
"C:\Users\NARENDER KESWANI\Documents\FYMCA\DSA\queueUsingLL.exe"  
QUEUE OPERATIONS USING LINKED LIST
```

```
-----  
  
1.ENQUEUE  
2.DEQUEUE  
3.DISPLAY  
4.EXIT  
Enter the Choice:  
  
3  
Queue is Empty  
Enter the Choice:  
  
1  
Enter value to be ENQUEUE:  
  
3  
Inserting  
Enter the Choice:  
  
3  
Queue Elements are  
3  
Enter the Choice:  
  
1  
Enter value to be ENQUEUE:  
  
5  
Inserting  
Enter the Choice:  
  
3  
Queue Elements are  
3  
5  
Enter the Choice:  
  
2  
3  
Enter the Choice:  
  
3  
Queue Elements are  
5  
Enter the Choice:  
  
2  
Element deleted from queue is: 5  
Enter the Choice:  
  
3  
Queue is Empty  
Enter the Choice:  
  
2  
Underflow  
Enter the Choice:
```

B) Circular Queue implementation using Linked List**THEORY:**

There was one limitation in the array implementation of Queue. If the rear reaches to the end position of the Queue then there might be possibility that some vacant spaces are left in the beginning which cannot be utilized. So, to overcome such limitations, the concept of the circular queue was introduced.

A circular queue is similar to a linear queue as it is also based on the FIFO (First In First Out) principle except that the last position is connected to the first position in a circular queue that forms a circle. It is also known as a Ring Buffer.

**Algorithm :-****Insert operation :-**

This function is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at Rear position.

Steps :-

1. Create a new node dynamically and insert value into it.
2. Check if $\text{front} == \text{NULL}$, if it is true then $\text{front} = \text{rear} = (\text{newly created node})$
3. If it is false then $\text{rear} = (\text{newly created node})$ and rear node always contains the address of the front node.

Delete operation :-

This function is used to delete an element from the circular queue. In a queue, the element is always deleted from front position.

Steps :-

1. Check whether queue is empty or not means $\text{front} == \text{NULL}$.
2. If it is empty then display Queue is empty. If queue is not empty then step 3
3. Check if $(\text{front} == \text{rear})$ if it is true then set $\text{front} = \text{rear} = \text{NULL}$ else move the front forward in queue, update address of front in rear node and return the element.

SOURCE CODE:

```
#include <iostream>
using namespace std;
struct Node
{
    int data;
    struct Node *next;
};
Node *front = NULL;
Node *rear = NULL;

void enqueue(int val)
{
    if(front==NULL || rear==NULL)
    {
        Node *newNode;
        newNode = new Node;

        newNode->data = val;
        newNode->next = NULL;

        front = newNode;
        rear = newNode;
    }
    else
    {
        Node *newNode;
        newNode = new Node;

        newNode->data = val;
        rear->next = newNode;

        newNode->next = front;
        rear = newNode;
    }
}

void dequeue()
{
    Node *n;
    n = front;
    if(front == NULL)
    {
        cout<<"Underflow \n"<<endl;
        return;
    }
    else
```

```
        {
            front = front->next;
            delete(n);
        }
    }

void display()
{
    Node *ptr;
    ptr = front;
    if(ptr == NULL)
    {
        cout<<"Queue is Empty \n"<<endl;
        return;
    }
    else
    {
        do
        {
            cout<<"\n";
            cout<<ptr->data<<" ";
            ptr = ptr->next;
            cout<<"\n";
        }
        while(ptr != rear->next);
    }
}


int main()
{
    int choice;
    int val;
    cout<<"CIRCULAR QUEUE OPERATIONS USING LINKED LIST"<<endl;
    cout<<"\n-----"<<endl;
    cout<<"\n 1.ENQUEUE\n 2.DEQUEUE\n 3.DISPLAY\n 4.EXIT"<<endl;
    do
    {
        cout<<"Enter the Choice: \n"<<endl;
        cin>>choice;
        switch(choice)
        {
            case 1:
            {
                cout<<"Enter value to be ENQUEUE: \n"<<endl;
                cin>>val;
                enqueue(val);
                break;
            }
            case 2:
            {
                dequeue();
            }
        }
    }
}
```



```
        break;
    }
    case 3:
    {
        display();
        break;
    }
    case 4:
    {
        cout<<"Exit \n"<<endl;
        break;
    }
    default:
    {
        cout<<"Invalid Input!, Please Enter a Valid Choice(1/2/3/4) \n"<<endl;
    }
}
while(choice!=4);

return 0;
}
```

OUTPUT:

 "C:\Users\NARENDER KESWANI\Documents\FYMCA\DSA\circularQueueUsingLL.exe"

CIRCULAR QUEUE OPERATIONS USING LINKED LIST

- 1.ENQUEUE
- 2.DEQUEUE
- 3.DISPLAY
- 4.EXIT

Enter the Choice:

3

Queue is Empty

Enter the Choice:

1

Enter value to be ENQUEUE:

7

Enter the Choice:

1

Enter value to be ENQUEUE:

9

Enter the Choice:

3

7

9

Enter the Choice:

2

Enter the Choice:

3

9

Enter the Choice:

C) Double ended Queue implementation using Linked ListTHEORY:

Double Ended Queue is also a Queue data structure in which the insertion and deletion operations 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 positions.



There are two variants of a **double-ended queue**. They include :-

□ **Input restricted deque** :- In this deque, insertions can be done only at one of the ends, while deletions can be done from both ends.

□ **Output restricted deque** :- In this deque, deletions can be done only at one of the ends, while insertions can be done on both ends.

Algorithm :-**Algorithm for Insertion at front end :-**

Step-1 : [Check for the front position]

if(front<=1)

Print("Cannot add item at the front");

return;

Step-2 : [Insert at front]

else

front=front-1;

q[front]=no;

Step-3 : Return

Algorithm for Insertion at rear end :-

Step-1: [Check for overflow]

if(rear==MAX)

Print("Queue is Overflow");

return;

Step-2: [Insert Element]

else

rear=rear+1;

q[rear]=no;

[Set rear and front pointer]

if rear=0

rear=1;

if front=0

front=1;

Step-3: return

Algorithm for Deletion from front end:-

```
Step-1 [ Check for front pointer]
        if front=0
print(" Queue is Underflow");
        return;
Step-2 [Perform deletion]
        else
        no=q[front];
                                print("Deleted element is",no);
        [Set front and rear
pointer]        if front=rear

front=0;
rear=0;        else
        front=front+1;
Step-3 : Return
```

Algorithm for Deletion from rear end :-

```
Step-1 : [Check for the rear pointer]
        if rear=0
                                print("Cannot delete value at rear
end");
        return;
Step-2: [ perform deletion]
        else
        no=q[rear];
                                [Check for the front and rear pointer]
        if front= rear

front=0;
rear=0;        else
        rear=rear-
1;
                                print("Deleted element is",no);
Step-3 : Return
```

SOURCE CODE:

```
#include <iostream>
using namespace std;
class Node
{
public:
    int data;
    Node *next;
    Node *prev;

    Node(int d)
    {
        data = d;
        next = NULL;
        prev = NULL;
    }
};

Node *newNode(int x)
{
    Node *node = new Node(x);
    return node;
}

Node *front = NULL;
Node *rear = NULL;
int Size = 0;
void insertFront(int x)
{
    Node *node = newNode(x);
    if (front == NULL)
    {
        front = rear = node;
    }
    else
    {
        node->next = front;
        front->prev = node;
        front = node;
    }
    Size++;
}

void insertEnd(int x)
{
    Node *node = newNode(x);
    if (rear == NULL)
    {
        front = rear = node;
    }
    else
```

```
{
    node->prev = rear;
    rear->next = node;
    rear = node;
}
Size++;
}
void deleteFront()
{
    if (front == NULL)
    {
        cout << "DeQueue is empty" << endl;
        return;
    }
    if (front == rear)
    {
        front = rear = NULL;
    }
    else
    {
        Node *temp = front;
        front = front->next;
        front->prev = NULL;
        delete (temp);
    }
    Size--;
}
void deleteEnd()
{
    if (rear == NULL)
    {
        cout << "DeQueue is empty" << endl;
        return;
    }
    if (front == rear)
    {
        front = rear = NULL;
    }
    else
    {
        Node *temp = rear;
        rear = rear->prev;
        rear->next = NULL;
        delete (temp);
    }
    Size--;
}
int getFront()
{
    if (front != NULL)
```

```
{
    return front->data;
}
return -1;
}
int getEnd()
{
    if (rear != NULL)
    {
        return rear->data;
    }
    return -1;
}
int size()
{
    return Size;
}
bool isEmpty()
{
    if (front == NULL)
    {
        return true;
    }
    return false;
}
void erase()
{
    rear = NULL;
    while (front != NULL)
    {
        Node *temp = front;
        front->prev = NULL;
        front = front->next;
        delete (temp);
    }
    Size = 0;
}
void display()
{
    Node *temp = front;
    if (front == NULL)
    {
        cout << "Queue is empty" << endl;
    }
    else
    {
        cout << "Queue is: \n";
        while (temp != NULL)
        {
```

```

        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << endl;
}
}

int main()
{

    int choice;
    int val;
    int n;
    cout<<"QUEUE OPERATIONS USING DOUBLY LINKED LIST"<<endl;
    cout<<"\n-----"<<endl;
    cout <<
    "\n\n1.InsertFront\n2.InsertEnd\n3.DeleteFront\n4.DeleteEnd\n5.GetFront\n6.GetEnd\n7.Is
    Empty\n8.SizeOfQueue\n9.Erase\n10.Display\n11.Exit";
    do
    {
        cout<<"\nEnter the Choice: \n"<<endl;
        cin>>choice;
        switch(choice)

        {
        case 1:
            cout << "\nEnter the element to be inserted at Front: ";
            cin >> n;
            insertFront(n);
            break;
        case 2:
            cout << "\nEnter the element to be inserted at End : ";
            cin >> n;
            insertEnd(n);
            break;
        case 3:
            cout << "\nDelete at Front ";
            deleteFront();
            break;
        case 4:
            cout << "\nDelete at End ";
            deleteEnd();
            break;
        case 5:
            n = getFront();
            if (n == -1)
            {
                cout << "\nDeQueue is empty" << endl;
            }
        }
    }
}

```




```
        else
        {
            cout << "\nFront element is : " << getFront();
        }
        break;
    case 6:
        n = getFront();
        if (n == -1)
        {
            cout << "DeQueue is empty" << endl;
        }
        else
        {
            cout << "\nEnd element is : " << getEnd();
        }
        break;
    case 7:
        if (isEmpty())
        {
            cout << "\nDeQueue is empty";
        }
        else
        {
            cout << "\nDeQueue is not empty";
        }
        break;
    case 8:
        cout << "\nSize of DeQueue is : " << size();
        break;
    case 9:
        erase();
        break;
    case 10:
        display();
        break;
    case 11:
        exit(0);
        break;
    default:
        cout << "\nSelect Proper Option" << endl;
    }
}

while(choice!=11);

return 0;
}
```

OUTPUT:

 "C:\Users\NARENDER KESWANI\Documents\FYMCA\DSA\queueUsingDLL.exe"

QUEUE OPERATIONS USING DOUBLY LINKED LIST

- 1.InsertFront
- 2.InsertEnd
- 3.DeleteFront
- 4.DeleteEnd
- 5.GetFront
- 6.GetEnd
- 7.IsEmpty
- 8.SizeOfQueue
- 9.Erase
- 10.Display
- 11.Exit

Enter the Choice:

3

Delete at Front DeQueue is empty

Enter the Choice:

4

Delete at End DeQueue is empty

Enter the Choice:

5

DeQueue is empty

Enter the Choice:

6

DeQueue is empty

Enter the Choice:

7

DeQueue is empty

Enter the Choice:

8

Size of DeQueue is : 0

Enter the Choice:

10

Queue is empty

Enter the Choice:

1

Enter the element to be inserted at Front: 5

Enter the Choice:

2

Enter the element to be inserted at End : 10

Enter the Choice:

10

Queue is:

5 10

Enter the Choice:

1

Enter the element to be inserted at Front: 3

Enter the Choice:

5

Front element is : 3

Enter the Choice:

6

End element is : 10

Enter the Choice:

7

DeQueue is not empty
Enter the Choice:

8

Size of DeQueue is : 3
Enter the Choice:

10

Queue is:
3 5 10

Enter the Choice:

3

Delete at Front
Enter the Choice:

10

Queue is:
5 10

Enter the Choice:

4

Delete at End
Enter the Choice:

10

Queue is:
5

Enter the Choice:

9

Enter the Choice:

10

Queue is empty

CONCLUSION:

From this practical, I have learned about queue implementation using array, linked list & doubly linked list.