

Unit 4 - Chapter 6

Queue

6.0 Objective

6.1.Queue

6.2.Applications of Queue

6.3.Types of Queues

6.4.Operations on Queue

6.5.Implementation of Queue

6.5.1. Consecutive assignment:

6.5.2. Linked list allocation:

6.6.What are the utilization instances of Queue?

6.7.Types of Queue

6.7.1.Linear Queue

6.7.2.Circular Queue

6.7.3.Priority Queue

6.7.4.Dequeue

6.8.Array representation of Queue

6.9.Queue

6.9.1. Algorithm

6.9.2. C Function

6.9.3.Algorithm

6.9.4.Menu driven program to implement queue using array

6.10.Linked List implementation of Queue

6.11.Operation on Linked Queue

6.11.1.Insert operation

6.11.2.Algorithm

6.11.3.C Function

6.12.Deletion

6.12.1.Algorithm

6.12.2.C Function

6.12.3.Menu-Driven Program implementing all the operations on Linked Queue

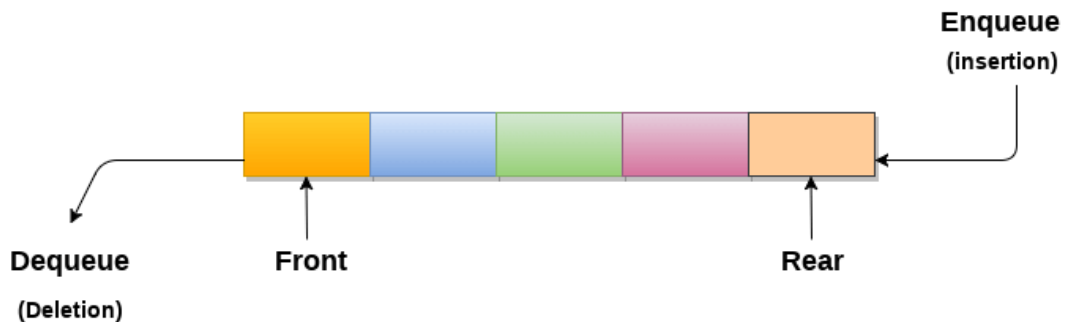
6.0. Objective

This chapter would make you understand the following concepts:

- Queue
- Definition
- Operations, Implementation of simple
- queue (Array and Linked list) and applications of queue-BFS
- Types of queues: Circular, Double ended, Priority,

6.1.Queue

1. A Queue can be characterized as an arranged rundown which empowers embed tasks to be performed toward one side called REAR and erase activities to be performed at another end called FRONT.
2. Queue is alluded to be as First In First Out rundown.
3. For instance, individuals sitting tight in line for a rail ticket structure a Queue



6.2. Applications of Queue

Because of the way that line performs activities on first in first out premise which is very reasonable for the requesting of activities. There are different uses of queues examined as beneath.

1. Queues are generally utilized as hanging tight records for a solitary shared asset like printer, plate, CPU.
2. Queues are utilized in offbeat exchange of information (where information isn't being moved at similar rate between two cycles) for eg. pipes, document IO, attachments.
3. Queues are utilized as cradles in the greater part of the applications like MP3 media player, CD player, and so on
4. Queue are utilized to keep up the play list in media major parts to add and eliminate the tunes from the play-list.
5. Queues are utilized in working frameworks for dealing with interferes.

Complexity

Data Structure	Time Complexity								Space Compleity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Queue	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$

6.3. Types of Queues

Prior to understanding the sorts of queues, we first glance at 'what is Queue'.

What is the Queue?

A queue in the information construction can be viewed as like the queue in reality. A queue is an information structure in which whatever starts things out will go out first.

It follows the FIFO (First-In-First-Out) arrangement. In Queue, the inclusion is done from one end known as the backside or the tail of the queue, though the erasure is done from another end known as the front end or the top of the line. At the end of the day, it very well may be characterized as a rundown or an assortment with an imperative that the addition can be performed toward one side called as the backside or tail of the queue and cancellation is performed on another end called as the front end or the top of the queue.



6.4. Operations on Queue

- o Enqueue: The enqueue activity is utilized to embed the component at the backside of the queue. It brings void back.
- o Dequeue: The dequeue activity plays out the erasure from the front-finish of the queue. It additionally restores the component which has been eliminated from the front-end. It restores a number worth. The dequeue activity can likewise be intended to void.
- o Peek: This is the third activity that profits the component, which is pointed by the front pointer in the queue yet doesn't erase it.
- o Queue flood (isfull): When the Queue is totally full, at that point it shows the flood condition.
- o Queue undercurrent (isempty): When the Queue is unfilled, i.e., no components are in the Queue then it tosses the sub-current condition.

A Queue can be addressed as a compartment opened from both the sides in which the component can be enqueued from one side and dequeued from another side as demonstrated in the beneath figure:

6.5. Implementation of Queue

There are two different ways of executing the Queue:

6.5.1. Consecutive assignment: The successive distribution in a Queue can be executed utilizing a cluster.

6.5.2. Linked list allocation: The linked list portion in a Queue can be actualized utilizing a linked list.

6.6. What are the utilization instances of Queue?

Here, we will see this present reality situations where we can utilize the Queue information structure. The Queue information structure is primarily utilized where

there is a shared asset that needs to serve the different asks for however can serve a solitary solicitation at a time. In such cases, we need to utilize the Queue information structure for lining up the solicitations. The solicitation that shows up first in the line will be served first. Coming up next are this present reality situations in which the Queue idea is utilized:

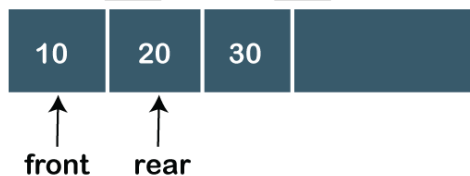
- o Suppose we have a printer divided among different machines in an organization, and any machine or PC in an organization can send a print solicitation to the printer. In any case, the printer can serve a solitary solicitation at a time, i.e., a printer can print a solitary archive at a time. At the point when any print demand comes from the organization, and if the printer is occupied, the printer's program will put the print demand in a line.
- o If the solicitations are accessible in the Queue, the printer takes a solicitation from the front of the Queue, and serves it.
- o The processor in a PC is likewise utilized as a shared asset. There are numerous solicitations that the processor should execute, however the processor can serve a solitary ask for or execute a solitary interaction at a time. Consequently, the cycles are kept in a Queue for execution.

6.7. Types of Queue

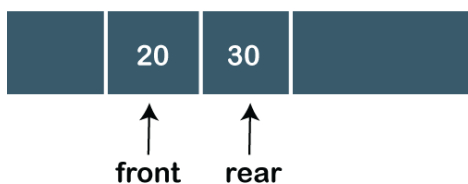
There are four kinds of Queues:

6.7.1. Linear Queue

In Linear Queue, an inclusion happens from one end while the erasure happens from another end. The end at which the addition happens is known as the backside, and the end at which the erasure happens is known as front end. It carefully keeps the FIFO rule. The straight Queue can be addressed, as demonstrated in the beneath figure:



The above figure shows that the components are embedded from the backside, and on the off chance that we embed more components in a Queue, at that point the back worth gets increased on each addition. In the event that we need to show the cancellation, at that point it tends to be addressed as:

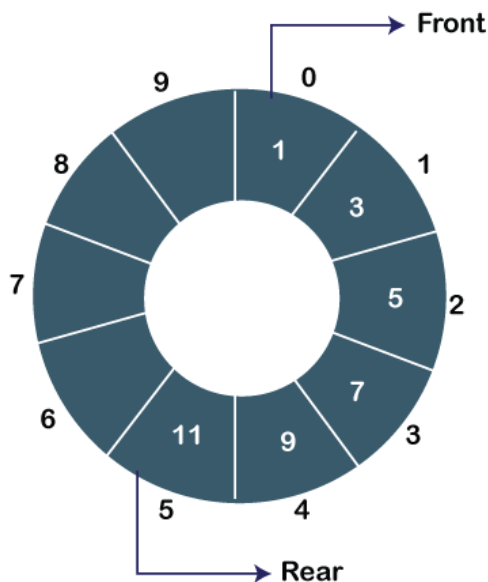


In the above figure, we can see that the front pointer focuses to the following component, and the component which was recently pointed by the front pointer was erased.

The significant disadvantage of utilizing a straight Queue is that inclusion is done distinctly from the backside. In the event that the initial three components are erased from the Queue, we can't embed more components despite the fact that the space is accessible in a Linear Queue. For this situation, the straight Queue shows the flood condition as the back is highlighting the last component of the Queue.

6.7.2. Circular Queue

In Circular Queue, all the hubs are addressed as round. It is like the direct Queue aside from that the last component of the line is associated with the principal component. It is otherwise called Ring Buffer as all the finishes are associated with another end. The round line can be addressed as:



The disadvantage that happens in a direct line is overwhelmed by utilizing the roundabout queue. On the off chance that the unfilled space is accessible in a round line, the new component can be included a vacant space by just augmenting the estimation of back.

6.7.3. Priority Queue

A need queue is another exceptional sort of Queue information structure in which every component has some need related with it. In view of the need of the component, the components are organized in a need line. In the event that the components happen with a similar need, at that point they are served by the FIFO rule.

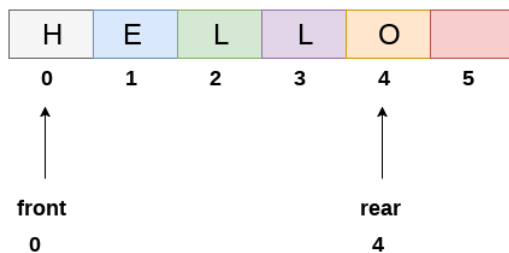
In need Queue, the inclusion happens dependent on the appearance while the cancellation happens dependent on the need. The need Queue can be appeared as: The above figure shows that the most elevated need component starts things out and the components of a similar need are organized dependent on FIFO structure.

6.7.4. Deque

Both the Linear Queue and Deque are distinctive as the direct line follows the FIFO standard while, deque doesn't follow the FIFO rule. In Deque, the inclusion and erasure can happen from the two closures.

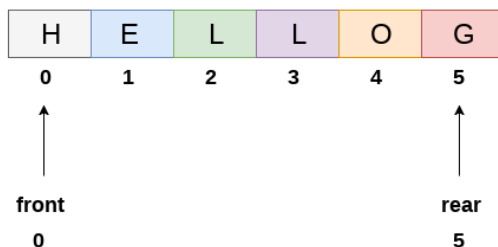
6.8. Array representation of Queue

We can without much of a stretch address queue by utilizing direct exhibits. There are two factors for example front and back, that are actualized on account of each queue. Front and back factors highlight the situation from where inclusions and cancellations are acted in a queue. At first, the estimation of front and queue is - 1 which addresses an unfilled queue. Cluster portrayal of a queue containing 5 components alongside the separate estimations of front and back, is appeared in the accompanying figure.



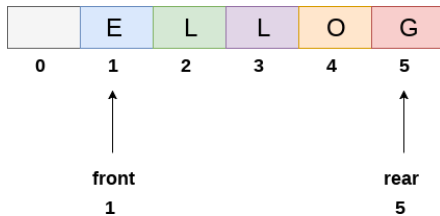
6.9. Queue

The above figure shows the queue of characters shaping the English word "Hi". Since, No cancellation is acted in the line till now, thusly the estimation of front remaining parts - 1 . Be that as it may, the estimation of back increments by one each time an addition is acted in the queue. Subsequent to embeddings a component into the queue appeared in the above figure, the queue will look something like after. The estimation of back will become 5 while the estimation of front remaining parts same.



Queue after inserting an element

After deleting an element, the value of front will increase from -1 to 0. however, the queue will look something like following.



Queue after deleting an element

Algorithm to embed any component in a line

Check if the line is as of now full by contrasting back with $\text{max} - 1$. assuming this is the case, at that point return a flood blunder.

In the event that the thing is to be embedded as the principal component in the rundown, all things considered set the estimation of front and back to 0 and addition the component at the backside.

In any case continue to expand the estimation of back and addition every component individually having back as the file.

6.9.1. Algorithm

Step 1: IF $\text{REAR} = \text{MAX} - 1$

Write OVERFLOW

Go to step

[END OF IF]

Step 2: IF $\text{FRONT} = -1$ and $\text{REAR} = -1$

SET $\text{FRONT} = \text{REAR} = 0$

ELSE

SET $\text{REAR} = \text{REAR} + 1$

[END OF IF]

Step 3: Set $\text{QUEUE}[\text{REAR}] = \text{NUM}$

Step 4: EXIT

6.9.2.C Function

```
void insert (int queue[], int max, int front, int rear, int item)
```

```
{
if (rear + 1 == max)
{
printf("overflow");
}
else
{
if(front == -1 && rear == -1)
{
```

```

front = 0;
rear = 0;
    }
else
    {
rear = rear + 1;
    }
queue[rear]=item;
    }
}

```

Algorithm to delete an element from the queue

If, the value of front is -1 or value of front is greater than rear , write an underflow message and exit.

Otherwise, keep increasing the value of front and return the item stored at the front end of the queue at each time.

6.9.3. Algorithm

Step 1: IF FRONT = -1 or FRONT > REAR

Write UNDERFLOW

ELSE

SET VAL = QUEUE[FRONT]

SET FRONT = FRONT + 1

[END OF IF]

Step 2: EXIT

C Function

int delete (int queue[], int max, int front, int rear)

```

{
int y;
if (front == -1 || front > rear)

```

```

    {
printf("underflow");
    }

```

else

```

    {
        y = queue[front];
if(front == rear)
    {

```



```

front = rear = -1;
else
front = front + 1;

```

```

    }
return y;
}
}

```

6.9.4. Menu driven program to implement queue using array

```

#include<stdio.h>
#include<stdlib.h>
#define maxsize 5
void insert();
void delete();
void display();
int front = -1, rear = -1;
int queue[maxsize];
void main ()
{
int choice;
while(choice != 4)
{
printf("\n*****Main
Menu*****\n");
printf("\n=====
=====\\n");

printf("\n1.insert an element\n2.Delete an element\n3.Display the
queue\n4.Exit\n");
printf("\nEnter your choice ?");
scanf("%d",&choice);
switch(choice)
{
case 1:
insert();
break;
case 2:
delete();
break;
case 3:
display();

```

```
break;
case 4:
exit(0);
break;
default:
printf("\nEnter valid choice??\n");
    }
}
}
void insert()
{
int item;
printf("\nEnter the element\n");
scanf("\n%d",&item);
if(rear == maxsize-1)
{
printf("\nOVERFLOW\n");
return;
}
if(front == -1 && rear == -1)
{
front = 0;
rear = 0;
}
else
{
rear = rear+1;
}
queue[rear] = item;
printf("\nValue inserted ");

}
void delete()
{
int item;
if (front == -1 || front > rear)
{
printf("\nUNDERFLOW\n");
return;
}
```

```

    }
else
{
item = queue[front];
if(front == rear)
{
front = -1;
rear = -1 ;
}
else
{
front = front + 1;
}
printf("\nvalue deleted ");
}

}

```

```

void display()
{
inti;
if(rear == -1)
{
printf("\nEmpty queue\n");
}
else
{ printf("\nprinting values ..... \n");
for(i=front;i<=rear;i++)
{
printf("\n%d\n",queue[i]);
}
}
}
}

```

Output:

*****Main Menu*****

=====

- 1.insert an element
- 2.Delete an element
- 3.Display the queue

4.Exit

Enter your choice ?1

Enter the element

123

Value inserted

*****Main Menu*****

=====

- 1.insert an element
- 2.Delete an element
- 3.Display the queue
- 4.Exit

Enter your choice ?1

Enter the element

90

Value inserted

*****Main Menu*****

=====

- 1.insert an element
- 2.Delete an element
- 3.Display the queue
- 4.Exit

Enter your choice ?2

value deleted

*****Main Menu*****

=====

- 1.insert an element
- 2.Delete an element
- 3.Display the queue
- 4.Exit

Enter your choice ?3

printing values

90

*****Main Menu*****

=====

- 1.insert an element
- 2.Delete an element
- 3.Display the queue
- 4.Exit

Enter your choice ?4

6.10. Linked List implementation of Queue

Because of the disadvantages examined in the past part of this instructional exercise, the exhibit usage can not be utilized for the huge scope applications where the queues are actualized. One of the option of cluster usage is connected rundown execution of queue.

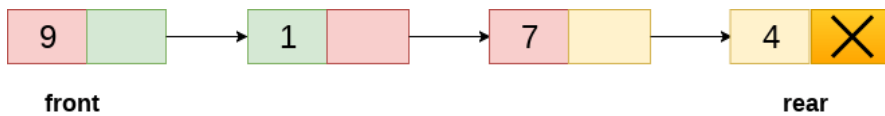
The capacity prerequisite of connected portrayal of a queue with n components is $O(n)$ while the time necessity for tasks is $O(1)$.

In a linked queue, every hub of the queue comprises of two sections for example information part and the connection part. Every component of the queue focuses to its nearby next component in the memory.

In the linked queue, there are two pointers kept up in the memory for example front pointer and back pointer. The front pointer contains the location of the beginning component of the queue while the back pointer contains the location of the last component of the queue.

Inclusion and erasures are performed at back and front end separately. On the off chance that front and back both are NULL, it shows that the line is vacant.

The connected portrayal of queue is appeared in the accompanying figure.



Linked Queue

6.11. Operation on Linked Queue

There are two basic operations which can be implemented on the linked queues. The operations are Insertion and Deletion.

6.11.1. Insert operation

The addition activity attach the line by adding a component to the furthest limit of the line. The new component will be the last component of the line.

Right off the bat, assign the memory for the new hub ptr by utilizing the accompanying assertion.

```
Ptr = (struct node *) malloc (sizeof(struct node));
```

There can be the two situation of embeddings this new hub ptr into the connected line.

In the principal situation, we embed component into an unfilled queue. For this situation, the condition `front = NULL` turns out to be valid. Presently, the new component will be added as the lone component of the queue and the following pointer of front and back pointer both, will highlight NULL.

```
ptr -> data = item;
if(front == NULL)
{
    front = ptr;
    rear = ptr;
    front -> next = NULL;
    rear -> next = NULL;
}
```

In the subsequent case, the queue contains more than one component. The condition `front = NULL` turns out to be bogus. In this situation, we need to refresh the end pointer back with the goal that the following pointer of back will highlight the new hub ptr. Since, this is a connected line, consequently we likewise need to make the back pointer highlight the recently added hub ptr. We additionally need to make the following pointer of back highlight NULL.

```
rear -> next = ptr;
rear = ptr;
rear->next = NULL;
```

In this way, the element is inserted into the queue. The algorithm and the C implementation is given as follows.

6.11.2. Algorithm

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

6.11.3.C Function

```
void insert(struct node *ptr, int item; )
{
ptr = (struct node *) malloc (sizeof(struct node));
if(ptr == NULL)
{
printf("\nOVERFLOW\n");
return;
}
else
{
ptr -> data = item;
if(front == NULL)
{
front = ptr;
rear = ptr;
front -> next = NULL;
rear -> next = NULL;
}
else
{
rear -> next = ptr;
rear = ptr;
}
```

```

    rear->next = NULL;
  }
}
}

```

6.12. Deletion

Cancellation activity eliminates the component that is first embedded among all the queue components. Right off the bat, we need to check either the rundown is unfilled or not. The condition `front == NULL` turns out to be valid if the rundown is unfilled, for this situation, we essentially compose undercurrent on the comfort and make exit. Else, we will erase the component that is pointed by the pointer `front`. For this reason, duplicate the hub pointed by the front pointer into the pointer `ptr`. Presently, move the front pointer, highlight its next hub and free the hub pointed by the hub `ptr`. This is finished by utilizing the accompanying assertions.

```

ptr = front;
front = front -> next;
free(ptr);

```

The algorithm and C function is given as follows.

6.12.1. Algorithm

```

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

```

6.12.2.C Function

```

void delete (struct node *ptr)
{
if(front == NULL)
{
printf("\nUNDERFLOW\n");
return;
}
else
{
ptr = front;
front = front -> next;
free(ptr);
}
}

```



```

    }
}

```

6.12.3.Menu-Driven Program implementing all the operations on Linked Queue

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

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

```
struct node
```

```
{
```

```
int data;
```

```
struct node *next;
```

```
};
```

```
struct node *front;
```

```
struct node *rear;
```

```
void insert();
```

```
void delete();
```

```
void display();
```

```
void main ()
```

```
{
```

```
int choice;
```

```
while(choice != 4)
```

```
{
```

```
printf("\n*****Main
```

```
Menu*****\n");
```

```
printf("\n=====
```

```
=====\\n");
```

```
printf("\n1.insert an element\n2.Delete an element\n3.Display the
```

```
queue\n4.Exit\n");
```

```
printf("\nEnter your choice ?");
```

```
scanf("%d",& choice);
```

```
switch(choice)
```

```
{
```

```
case 1:
```

```
insert();
```

```
break;
```

```
case 2:
```

```
delete();
```

```
break;
```

```
case 3:
```

```
display();
```

```
break;
```

```

case 4:
exit(0);
break;
default:
printf("\nEnter valid choice??\n");
    }
}
}
void insert()
{
struct node *ptr;
int item;

ptr = (struct node *) malloc (sizeof(struct node));
if(ptr == NULL)
{
printf("\nOVERFLOW\n");
return;
}
else
{
printf("\nEnter value?\n");
scanf("%d",&item);
ptr -> data = item;
if(front == NULL)
{
front = ptr;
rear = ptr;
front -> next = NULL;
rear -> next = NULL;
}
else
{
rear -> next = ptr;
rear = ptr;
rear->next = NULL;
}
}
}
void delete ()

```

```

{
struct node *ptr;
if(front == NULL)
{
printf("\nUNDERFLOW\n");
return;
}
else
{
ptr = front;
front = front -> next;
free(ptr);
}
}
void display()
{
struct node *ptr;
ptr = front;
if(front == NULL)
{
printf("\nEmpty queue\n");
}
else
{ printf("\nprinting values ..... \n");
while(ptr != NULL)
{
printf("\n%d\n",ptr -> data);
ptr = ptr -> next;
}
}
}
}
Output:

```

*****Main Menu*****

=====

- 1.insert an element
- 2.Delete an element
- 3.Display the queue

4.Exit

Enter your choice ?1

Enter value?

123

*****Main Menu*****

=====

- 1.insert an element
- 2.Delete an element
- 3.Display the queue
- 4.Exit

Enter your choice ?1

Enter value?

90

*****Main Menu*****

=====

- 1.insert an element
- 2.Delete an element
- 3.Display the queue
- 4.Exit

Enter your choice ?3

printing values

123

90

*****Main Menu*****

-
-
- 1.insert an element
 - 2.Delete an element
 - 3.Display the queue
 - 4.Exit

Enter your choice ?2

*****Main Menu*****

-
-
- 1.insert an element
 - 2.Delete an element
 - 3.Display the queue
 - 4.Exit

Enter your choice ?3

printing values

90

*****Main Menu*****

-
-
- 1.insert an element
 - 2.Delete an element
 - 3.Display the queue
 - 4.Exit

Enter your choice 4

Unit 4 - Chapter 7

Types of Queue

7.0.Objective

7.1.Circular Queue

7.2. What is a Circular Queue?

7.2.1. Procedure on Circular Queue

7.3.Uses of Circular Queue

7.4. Enqueue operation

7.5. Algorithm to insert an element in a circular queue

7.6.Dequeue Operation

7.6.1. Algorithm to delete an element from the circular queue

7.7. Implementation of circular queue using Array

7.8. Implementation of circular queue using linked list

7.9. Deque

7.10.Operations on Deque

7.10.1.Memory Representation

7.10.2.What is a circular array?

7.10.3.Applications of Deque

7.11.Implementation of Deque using a circular array

7.12.Dequeue Operation

7.13. Program for deque Implementation

7.14. What is a priority queue?

7.15.Characteristics of a Priority queue

7.16.Types of Priority Queue

7.16.1.Ascending order priority queue

7.16.2.Descending order priority queue

7.16.3.Representation of priority queue

7.17.Implementation of Priority Queue

7.17.1.Analysis of complexities using different implementations

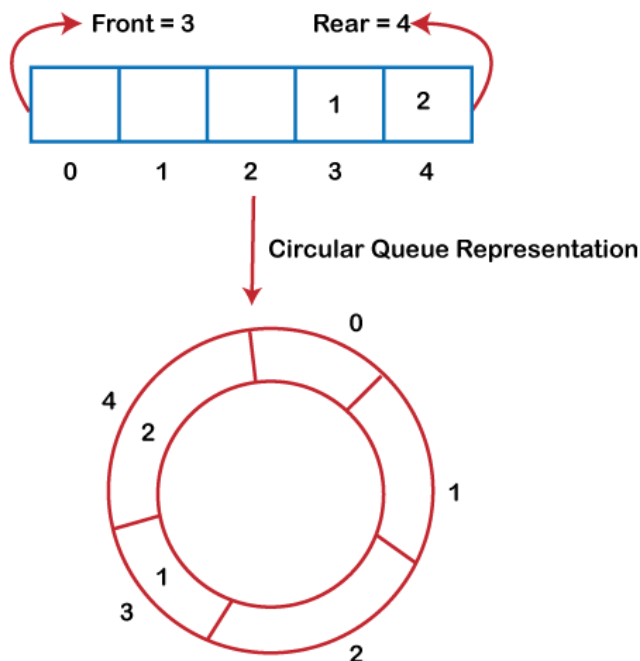
7.0. Objective

This chapter would make you understand the following concepts:

- Understand the concept of Circular Queue
- Operation of Circular Queue
- Application of Circular Queue
- Implementation of Circular Queue

7.1. Circular Queue

There was one limit in the exhibit usage of Queue. On the off chance that the back spans to the end position of the Queue, at that point there may be plausibility that some empty spaces are left to start with which can't be used. Thus, to defeat such restrictions, the idea of the round line was presented.



As we can find in the above picture, the back is at the last situation of the Queue and front is pointing some place as opposed to the 0th position. In the above exhibit, there are just two components and other three positions are unfilled. The back is at the last situation of the Queue; in the event that we attempt to embed the component, at that point it will show that there are no unfilled spaces in the Queue. There is one answer for maintain a strategic distance from such wastage of memory space by moving both the components at the left and change the front and backside as needs be. It's anything but a for all intents and purposes great methodology since moving all the components will burn-through loads of time. The effective way to deal with stay away from the wastage of the memory is to utilize circular queue data structure.

7.2. What is a Circular Queue?

A circular queue is like a linear queue as it is likewise founded on the FIFO (First In First Out) rule aside from that the last position is associated with the principal position in a round line that shapes a circle. It is otherwise called a Ring Buffer.

7.2.1. Procedure on Circular Queue

Coming up next are the activities that can be performed on a circular queue:

Front: It is utilized to get the front component from the Queue.

Back: It is utilized to get the back component from the Queue.

enqueue(value): This capacity is utilized to embed the new incentive in the Queue. The new component is constantly embedded from the backside.

deQueue(): This capacity erases a component from the Queue. The cancellation in a Queue9-

consistently happens from the front end.

7.3. Uses of Circular Queue

The roundabout Queue can be utilized in the accompanying situations:

Memory the board: The roundabout queue gives memory the executives. As we have just seen that in linear queue, the memory isn't overseen proficiently. Yet, if there should arise an occurrence of a roundabout queue, the memory is overseen effectively by putting the components in an area which is unused.

CPU Scheduling: The working framework likewise utilizes the circular queue to embed the cycles and afterward execute them.

Traffic framework: In a PC control traffic framework, traffic signal is probably the best illustration of the circular queue. Each light of traffic signal gets ON individually after each jinterval of time. Like red light gets ON briefly then yellow light briefly and afterward green light. After green light, the red light gets ON.

7.4. Enqueue operation

The steps of enqueue operation are given below:

First, we will check whether the Queue is full or not.

Initially the front and rear are set to -1. When we insert the first element in a Queue, front and rear both are set to 0.

When we insert a new element, the rear gets incremented, i.e., $\text{rear} = \text{rear} + 1$.

Scenarios for inserting an element

There are two scenarios in which queue is not full:

If $\text{rear} \neq \text{max} - 1$, then rear will be incremented to $\text{mod}(\text{maxsize})$ and the new value will be inserted at the rear end of the queue.

If $\text{front} \neq 0$ and $\text{rear} = \text{max} - 1$, it means that queue is not full, then set the value of rear to 0 and insert the new element there.

There are two cases in which the element cannot be inserted:

When $\text{front} == 0$ && $\text{rear} = \text{max} - 1$, which means that front is at the first position of the Queue and rear is at the last position of the Queue.

$\text{front} == \text{rear} + 1$;

7.5. Algorithm to insert an element in a circular queue

Step 1: IF $(\text{REAR} + 1) \% \text{MAX} = \text{FRONT}$

Write " OVERFLOW "

Goto step 4
[End OF IF]

Step 2: IF FRONT = -1 and REAR = -1
SET FRONT = REAR = 0
ELSE IF REAR = MAX - 1 and FRONT != 0
SET REAR = 0
ELSE
SET REAR = (REAR + 1) % MAX
[END OF IF]

Step 3: SET QUEUE[REAR] = VAL

Step 4: EXIT

7.6. Dequeue Operation

The means of dequeue activity are given underneath:

To start with, we check if the Queue is vacant. In the event that the queue is unfilled, we can't play out the dequeue activity.

At the point when the component is erased, the estimation of front gets decremented by 1.

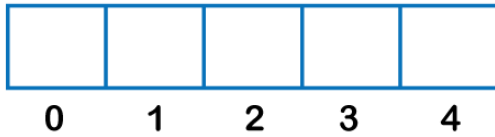
On the off chance that there is just a single component left which is to be erased, at that point the front and back are reset to - 1.

7.6.1. Algorithm to delete an element from the circular queue

Step 1: IF FRONT = -1
Write " UNDERFLOW "
Goto Step 4
[END of IF]
Step 2: SET VAL = QUEUE[FRONT]
Step 3: IF FRONT = REAR
SET FRONT = REAR = -1
ELSE
IF FRONT = MAX -1
SET FRONT = 0
ELSE
SET FRONT = FRONT + 1
[END of IF]
[END OF IF]

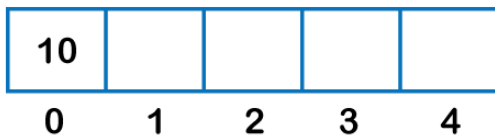
Step 4: EXIT

Let's understand the enqueue and dequeue operation through the diagrammatic representation.



Front = -1

Rear = -1

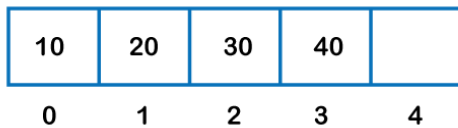


Front = 0

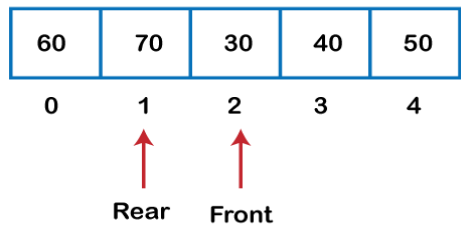
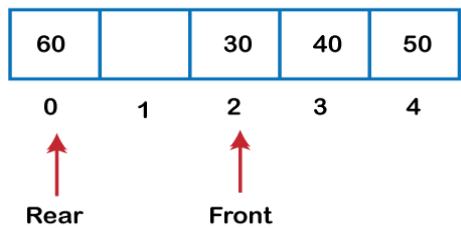
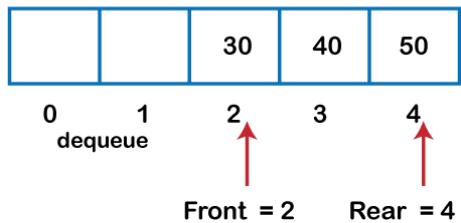
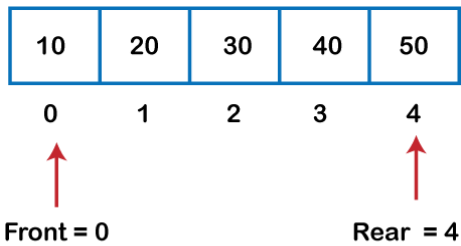
Rear = 0



Front = 0 **Rear = 2**



Front = 0 **Rear = 3**



7.7. Implementation of circular queue using Array

```

#include <stdio.h>
# define max 6
int queue[max]; // array declaration
int front=-1;
int rear=-1;
// function to insert an element in a circular queue
void enqueue(int element)
{
    if(front== -1 && rear== -1) // condition to check queue is empty
    {
        front=0;
        rear=0;
        queue[rear]=element;
    }
}

```

```

else if((rear+1)%max==front) // condition to check queue is full
{
    printf("Queue is overflow..");
}
else
{
    rear=(rear+1)%max;    // rear is incremented
    queue[rear]=element;  // assigning a value to the queue at the rear position.
}
}

```

// function to delete the element from the queue

```

int dequeue()
{
    if((front==-1) && (rear==-1)) // condition to check queue is empty
    {
        printf("\nQueue is underflow..");
    }
    else if(front==rear)
    {
        printf("\nThe dequeued element is %d", queue[front]);
        front=-1;
        rear=-1;
    }
    else
    {
        printf("\nThe dequeued element is %d", queue[front]);
        front=(front+1)%max;
    }
}

```

// function to display the elements of a queue

```

void display()
{
    int i=front;
    if(front==-1 && rear==-1)
    {
        printf("\n Queue is empty..");
    }
    else
    {

```

```

printf("\nElements in a Queue are :");
while(i<=rear)
{
    printf("%d,", queue[i]);
    i=(i+1)%max;
}
}
}
int main()
{
    int choice=1,x; // variables declaration

    while(choice<4 && choice!=0) // while loop
    {
        printf("\n Press 1: Insert an element");
        printf("\nPress 2: Delete an element");
        printf("\nPress 3: Display the element");
        printf("\nEnter your choice");
        scanf("%d", &choice);

        switch(choice)
        {

            case 1:

                printf("Enter the element which is to be inserted");
                scanf("%d", &x);
                enqueue(x);
                break;
            case 2:
                dequeue();
                break;
            case 3:
                display();

        }
    }
    return 0;
}

```

Output:

Output:

```
Press 1: Insert an element
Press 2: Delete an element
Press 3: Display the element
Enter your choice
1
Enter the element which is to be inserted
10

Press 1: Insert an element
Press 2: Delete an element
Press 3: Display the element
Enter your choice
1
Enter the element which is to be inserted
20

Press 1: Insert an element
Press 2: Delete an element
Press 3: Display the element
Enter your choice
1
Enter the element which is to be inserted
30

Press 1: Insert an element
Press 2: Delete an element
Press 3: Display the element
Enter your choice
3

Elements in a Queue are :10,20,30,
Press 1: Insert an element
Press 2: Delete an element
Press 3: Display the element
Enter your choice
2

The dequeued element is 10
```

7.8. Implementation of circular queue using linked list

As we realize that connected rundown is a direct information structure that stores two sections, i.e., information part and the location part where address part contains the location of the following hub. Here, connected rundown is utilized to execute the roundabout line; in this way, the connected rundown follows the properties of the Queue. At the point when we are actualizing the roundabout line utilizing connected rundown then both the enqueue and dequeue tasks take $O(1)$ time.

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

```
// Declaration of struct type node
```

```
struct node
```

```
{
```

```

int data;
struct node *next;
};
struct node *front=-1;
struct node *rear=-1;
// function to insert the element in the Queue
void enqueue(int x)
{
    struct node *newnode; // declaration of pointer of struct node type.
    newnode=(struct node *)malloc(sizeof(struct node)); // allocating the memory to the
    newnode
    newnode->data=x;
    newnode->next=0;
    if(rear==-1) // checking whether the Queue is empty or not.
    {
        front=rear=newnode;
        rear->next=front;
    }
    else
    {
        rear->next=newnode;
        rear=newnode;
        rear->next=front;
    }
}

// function to delete the element from the queue
void dequeue()
{
    struct node *temp; // declaration of pointer of node type
    temp=front;
    if((front==-1)&&(rear==-1)) // checking whether the queue is empty or not
    {
        printf("\nQueue is empty");
    }
    else if(front==rear) // checking whether the single element is left in the queue
    {
        front=rear=-1;
        free(temp);
    }
}

```

```
else
{
front=front->next;
rear->next=front;
free(temp);
}
}
```

// function to get the front of the queue

```
int peek()
{
if((front==-1) &&(rear==-1))
{
printf("\nQueue is empty");
}
else
{
printf("\nThe front element is %d", front->data);
}
}
```

// function to display all the elements of the queue

```
void display()
{
struct node *temp;
temp=front;
printf("\n The elements in a Queue are : ");
if((front==-1) && (rear==-1))
{
printf("Queue is empty");
}
}
```

```
else
{
while(temp->next!=front)
{
printf("%d", temp->data);
temp=temp->next;
}
printf("%d", temp->data);
}
```



```

    }
}

void main()
{
    enqueue(34);
    enqueue(10);
    enqueue(23);
    display();
    dequeue();
    peek();
}

```

Output:

```

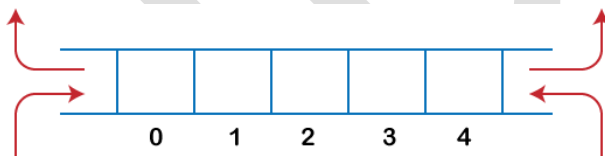
The elements in a Queue are : 34,10,23
The front element is 10

...Program finished with exit code 24
Press ENTER to exit console.

```

7.9. Deque

The dequeue represents Double Ended Queue. In the queue, the inclusion happens from one end while the erasure happens from another end. The end at which the addition happens is known as the backside while the end at which the erasure happens is known as front end.

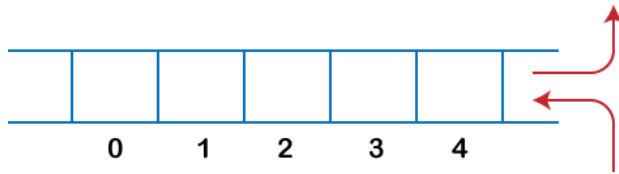


Deque is a direct information structure in which the inclusion and cancellation tasks are performed from the two finishes. We can say that deque is a summed up form of the line.

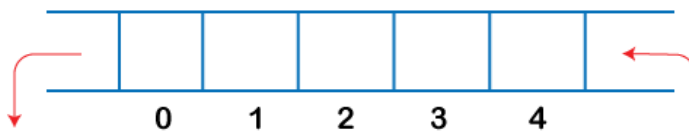
How about we take a gander at certain properties of deque.

Deque can be utilized both as stack and line as it permits the inclusion and cancellation procedure on the two finishes.

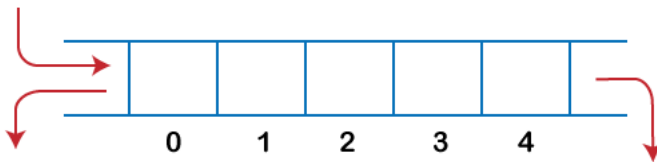
In deque, the inclusion and cancellation activity can be performed from one side. The stack adheres to the LIFO rule in which both the addition and erasure can be performed distinctly from one end; in this way, we reason that deque can be considered as a stack.



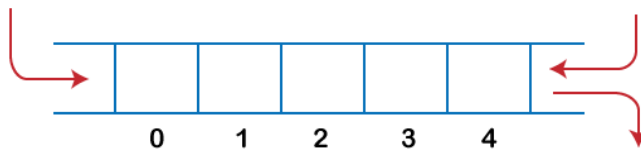
In deque, the addition can be performed toward one side, and the erasure should be possible on another end. The queue adheres to the FIFO rule in which the component is embedded toward one side and erased from another end. Hence, we reason that the deque can likewise be considered as the queue.



There are two types of Queues, Input-restricted queue, and output-restricted queue. Information confined queue: The info limited queue implies that a few limitations are applied to the inclusion. In info confined queue, the addition is applied to one end while the erasure is applied from both the closures.



Yield confined queue: The yield limited line implies that a few limitations are applied to the erasure activity. In a yield limited queue, the cancellation can be applied uniquely from one end, while the inclusion is conceivable from the two finishes.



7.10. Operations on Deque

The following are the operations applied on deque:

Insert at front

Delete from end

insert at rear

delete from rear

Other than inclusion and cancellation, we can likewise perform look activity in deque. Through look activity, we can get the front and the back component of the deque.

We can perform two additional procedure on dequeue:

isFull(): This capacity restores a genuine worth if the stack is full; else, it restores a bogus worth.

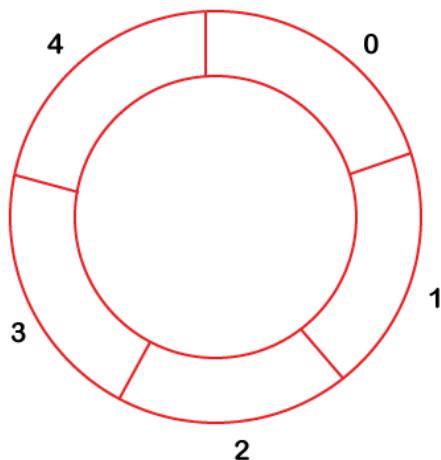
isEmpty(): This capacity restores a genuine worth if the stack is vacant; else it restores a bogus worth.

7.10.1. Memory Representation

The deque can be executed utilizing two information structures, i.e., round exhibit, and doubly connected rundown. To actualize the deque utilizing round exhibit, we initially should realize what is roundabout cluster.

7.10.2. What is a circular array?

An exhibit is supposed to be roundabout if the last component of the cluster is associated with the primary component of the exhibit. Assume the size of the cluster is 4, and the exhibit is full however the primary area of the cluster is unfilled. In the event that we need to embed the exhibit component, it won't show any flood condition as the last component is associated with the primary component. The worth which we need to embed will be included the primary area of the exhibit.



7.10.3. Applications of Deque

- The deque can be utilized as a stack and line; subsequently, it can perform both re-try and fix activities.
- It tends to be utilized as a palindrome checker implies that in the event that we read the string from the two closures, at that point the string would be the equivalent.

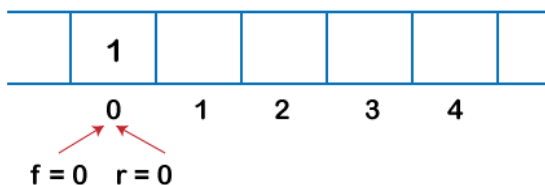
- It tends to be utilized for multiprocessor planning. Assume we have two processors, and every processor has one interaction to execute. Every processor is appointed with an interaction or a task, and each cycle contains numerous strings. Every processor keeps a deque that contains strings that are prepared to execute. The processor executes an interaction, and on the off chance that a cycle makes a kid cycle, at that point that cycle will be embedded at the front of the deque of the parent interaction. Assume the processor P2 has finished the execution of every one of its strings then it takes the string from the backside of the processor P1 and adds to the front finish of the processor P2. The processor P2 will take the string from the front end; thusly, the erasure takes from both the closures, i.e., front and backside. This is known as the A-take calculation for planning.

7.11. Implementation of Deque using a circular array

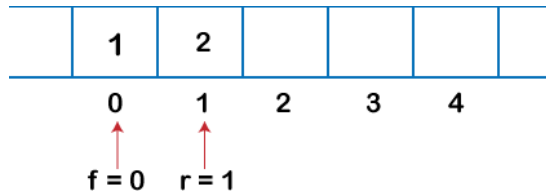
The following are the steps to perform the operations on the Deque:

Enqueue operation

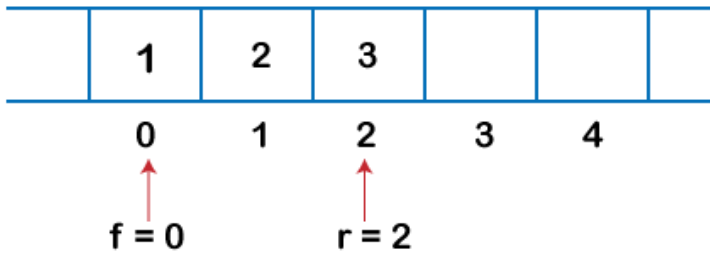
1. At first, we are thinking about that the deque is unfilled, so both front and back are set to - 1, i.e., $f = - 1$ and $r = - 1$.
2. As the deque is vacant, so embeddings a component either from the front or backside would be something very similar. Assume we have embedded component 1, at that point front is equivalent to 0, and the back is likewise equivalent to 0.



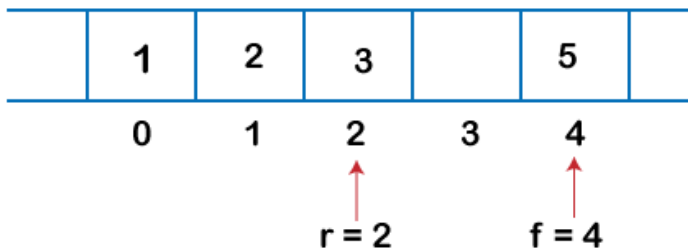
3. Assume we need to embed the following component from the back. To embed the component from the backside, we first need to augment the back, i.e., $\text{rear} = \text{rear} + 1$. Presently, the back is highlighting the subsequent component, and the front is highlighting the main component.



4. Assume we are again embeddings the component from the backside. To embed the component, we will first addition the back, and now back focuses to the third component.

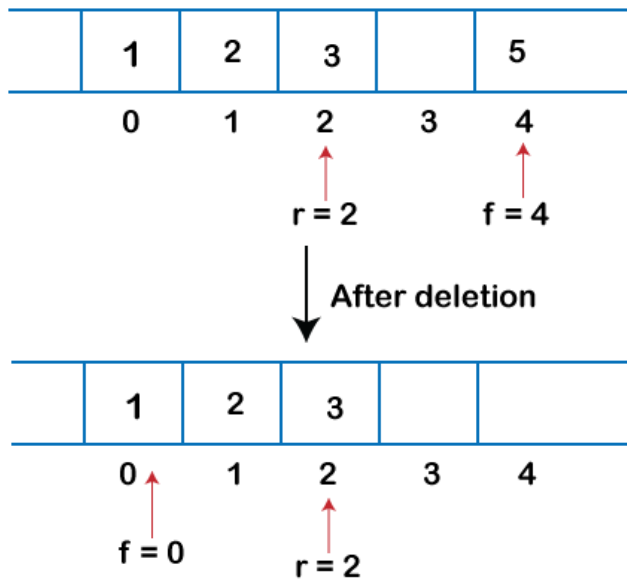


5. In the event that we need to embed the component from the front end, and addition a component from the front, we need to decrement the estimation of front by 1. In the event that we decrement the front by 1, at that point the front focuses to - 1 area, which isn't any substantial area in an exhibit. Thus, we set the front as $(n - 1)$, which is equivalent to 4 as n is 5. When the front is set, we will embed the incentive as demonstrated in the beneath figure:

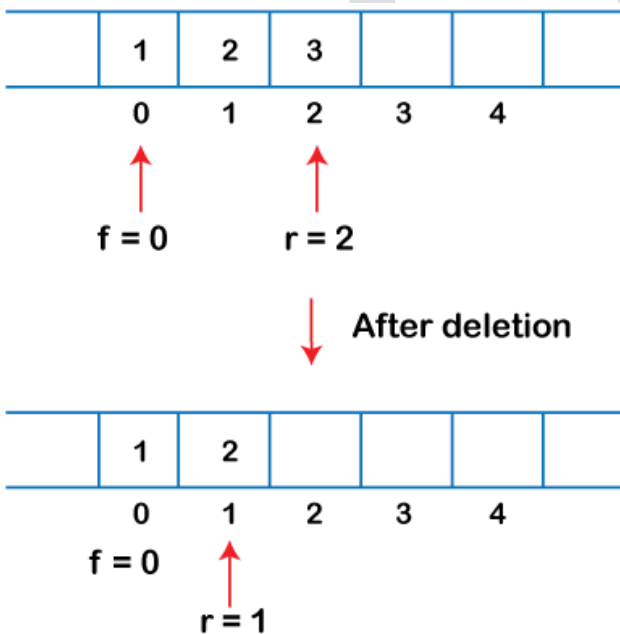


7.12. Dequeue Operation

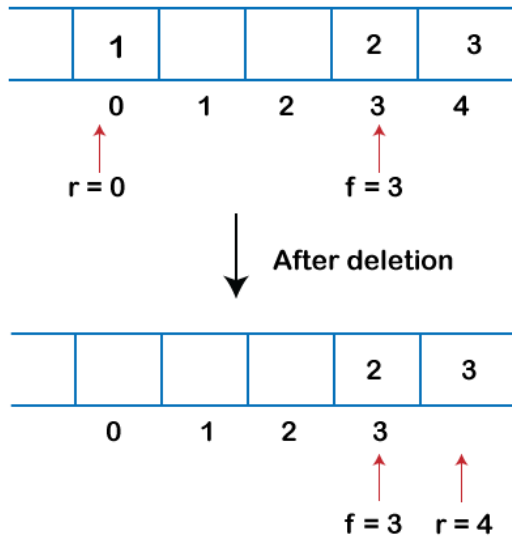
1. On the off chance that the front is highlighting the last component of the exhibit, and we need to play out the erase activity from the front. To erase any component from the front, we need to set $\text{front} = \text{front} + 1$. At present, the estimation of the front is equivalent to 4, and in the event that we increase the estimation of front, it becomes 5 which is definitely not a substantial list. Thusly, we presume that in the event that front focuses to the last component, at that point front is set to 0 if there should be an occurrence of erase activity.



2. If we want to delete the element from rear end then we need to decrement the rear value by 1, i.e., $\text{rear} = \text{rear} - 1$ as shown in the below figure:



3. In the event that the back is highlighting the principal component, and we need to erase the component from the backside then we need to set $\text{rear} = n - 1$ where n is the size of the exhibit as demonstrated in the beneath figure:



Let's create a program of deque.

The following are the six functions that we have used in the below program:

- **enqueue_front():** It is used to insert the element from the front end.
- **enqueue_rear():** It is used to insert the element from the rear end.
- **dequeue_front():** It is used to delete the element from the front end.
- **dequeue_rear():** It is used to delete the element from the rear end.
- **getfront():** It is used to return the front element of the deque.
- **getrear():** It is used to return the rear element of the deque.

7.13. Program for dqeue Implementation

```
#define size 5
#include <stdio.h>
int deque[size];
int f=-1, r=-1;
// enqueue_front function will insert the value from the front
void enqueue_front(int x)
{
    if((f==0 && r==size-1) || (f==r+1))
    {
        printf("deque is full");
    }
    else if((f==-1) && (r==-1))
    {
```

```
    f=r=0;
    deque[f]=x;
}
else if(f==0)
{
    f=size-1;
    deque[f]=x;
}
else
{
    f=f-1;
    deque[f]=x;
}
}
```

// enqueue_rear function will insert the value from the rear

```
void enqueue_rear(int x)
{
    if((f==0 && r==size-1) || (f==r+1))
    {
        printf("deque is full");
    }
    else if((f==-1) && (r==-1))
    {
        r=0;
        deque[r]=x;
    }
    else if(r==size-1)
    {
        r=0;
        deque[r]=x;
    }
    else
    {
        r++;
        deque[r]=x;
    }
}
```


// display function prints all the value of deque.

```
void display()
{
    int i=f;
    printf("\n Elements in a deque : ");

    while(i!=r)
    {
        printf("%d ",deque[i]);
        i=(i+1)%size;
    }
    printf("%d",deque[r]);
}
```

// getfront function retrieves the first value of the deque.

```
void getfront()
{
    if((f==-1) && (r==-1))
    {
        printf("Deque is empty");
    }
    else
    {
        printf("\nThe value of the front is: %d", deque[f]);
    }
}
```

// getrear function retrieves the last value of the deque.

```
void getrear()
{
    if((f==-1) && (r==-1))
    {
        printf("Deque is empty");
    }
    else
    {
        printf("\nThe value of the rear is: %d", deque[r]);
    }
}
```

```
}
```

// dequeue_front() function deletes the element from the front

```
void dequeue_front()
{
    if((f==-1) && (r==-1))
    {
        printf("Deque is empty");
    }
    else if(f==r)
    {
        printf("\nThe deleted element is %d", deque[f]);
        f=-1;
        r=-1;
    }
    else if(f==(size-1))
    {
        printf("\nThe deleted element is %d", deque[f]);
        f=0;
    }
    else
    {
        printf("\nThe deleted element is %d", deque[f]);
        f=f+1;
    }
}
```

// dequeue_rear() function deletes the element from the rear

```
void dequeue_rear()
{
    if((f==-1) && (r==-1))
    {
        printf("Deque is empty");
    }
    else if(f==r)
    {
        printf("\nThe deleted element is %d", deque[r]);
        f=-1;
        r=-1;
    }
}
```

```

}
else if(r==0)
{
    printf("\nThe deleted element is %d", deque[r]);
    r=size-1;
}
else
{
    printf("\nThe deleted element is %d", deque[r]);
    r=r-1;
}
}

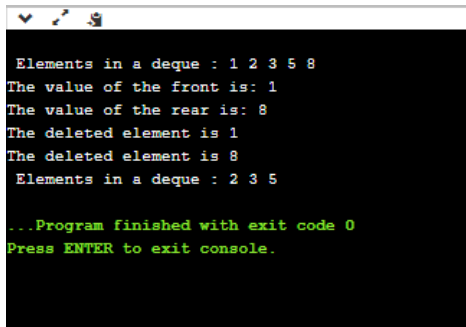
```

```

int main()
{
    // inserting a value from the front.
    enqueue_front(2);
    // inserting a value from the front.
    enqueue_front(1);
    // inserting a value from the rear.
    enqueue_rear(3);
    // inserting a value from the rear.
    enqueue_rear(5);
    // inserting a value from the rear.
    enqueue_rear(8);
    // Calling the display function to retrieve the values of deque
    display();
    // Retrieve the front value
    getfront();
    // Retrieve the rear value.
    getrear();
    // deleting a value from the front
    dequeue_front();
    //deleting a value from the rear
    dequeue_rear();
    // Calling the display function to retrieve the values of deque
    display();
    return 0;
}

```

Output:



```
Elements in a deque : 1 2 3 5 8
The value of the front is: 1
The value of the rear is: 8
The deleted element is 1
The deleted element is 8
Elements in a deque : 2 3 5

...Program finished with exit code 0
Press ENTER to exit console.
```

7.14. What is a priority queue?

A need queue is a theoretical information type that carries on comparatively to the ordinary queue aside from that every component has some need, i.e., the component with the most elevated need would start things out in a need line. The need of the components in a need queue will decide the request where components are taken out from the need line.

The need queue underpins just similar components, which implies that the components are either masterminded in a rising or slipping request.

For instance, assume we have a few qualities like 1, 3, 4, 8, 14, 22 embedded in a need queue with a requesting forced on the qualities is from least to the best. Along these lines, the 1 number would have the most elevated need while 22 will have the least need.

7.15. Characteristics of a Priority queue

A need queue is an expansion of a line that contains the accompanying qualities:

- o Every component in a need line has some need related with it.
- o An component with the higher need will be erased before the cancellation of the lesser need.
- o If two components in a need queue have a similar need, they will be organized utilizing the FIFO rule.

Let's understand the priority queue through an example.

We have a priority queue that contains the following values:

1, 3, 4, 8, 14, 22

All the qualities are orchestrated in climbing request. Presently, we will see how the need line will take care of playing out the accompanying activities:

poll(): This capacity will eliminate the most elevated need component from the need line. In the above need line, the '1' component has the most elevated need, so it will be eliminated from the need line.

add(2): This capacity will embed '2' component in a need line. As 2 is the littlest component among all the numbers so it will acquire the most elevated need.

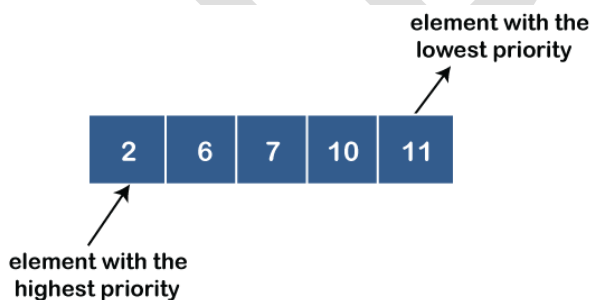
poll() It will eliminate '2' component from the need line as it has the most elevated need line.

add(5): It will embed 5 component after 4 as 5 is bigger than 4 and lesser than 8, so it will acquire the third most noteworthy need in a need line.

7.16. Types of Priority Queue

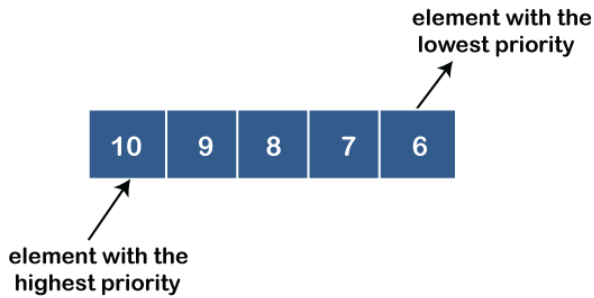
There are two types of priority queue:

7.16.1. Ascending order priority queue: In rising request need line, a lower need number is given as a higher need in a need. For instance, we take the numbers from 1 to 5 orchestrated in a rising request like 1,2,3,4,5; in this manner, the most modest number, i.e., 1 is given as the most noteworthy need in a need line.



7.16.2. Descending order priority queue:

In plunging request need line, a higher need number is given as a higher need in a need. For instance, we take the numbers from 1 to 5 orchestrated in diving request like 5, 4, 3, 2, 1; along these lines, the biggest number, i.e., 5 is given as the most elevated need in a need line.



7.16.3. Representation of priority queue

Presently, we will perceive how to address the need line through a single direction list.

We will make the need line by utilizing the rundown given underneath in which INFO list contains the information components, PRN list contains the need quantities of every information component accessible in the INFO rundown, and LINK essentially contains the location of the following hub.

	INFO	PNR	LINK
0	200	2	4
1	400	4	2
2	500	4	6
3	300	1	0
4	100	2	5
5	600	3	1
6	700	4	

Let's create the priority queue step by step.

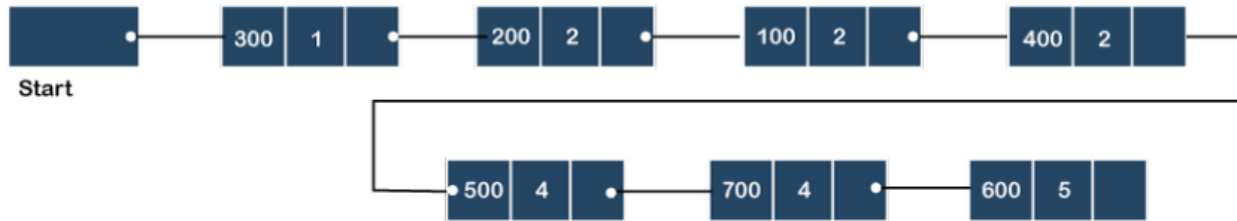
In the case of priority queue, lower priority number is considered the higher priority, i.e., lower priority number = higher priority.

Step 1: In the list, lower priority number is 1, whose data value is 333, so it will be inserted in the list as shown in the below diagram:

Step 2: After inserting 333, priority number 2 is having a higher priority, and data values associated with this priority are 222 and 111. So, this data will be inserted based on the FIFO principle; therefore 222 will be added first and then 111.

Step 3: After inserting the elements of priority 2, the next higher priority number is 4 and data elements associated with 4 priority numbers are 444, 555, 777. In this case, elements would be inserted based on the FIFO principle; therefore, 444 will be added first, then 555, and then 777.

Step 4: After inserting the elements of priority 4, the next higher priority number is 5, and the value associated with priority 5 is 666, so it will be inserted at the end of the queue.



7.17. Implementation of Priority Queue:

The need queue can be actualized in four different ways that incorporate clusters, connected rundown, stack information construction and twofold pursuit tree. The load information structure is the most productive method of executing the need queue, so we will actualize the need queue utilizing a store information structure in this subject. Presently, first we comprehend the motivation behind why pile is the most productive route among the wide range of various information structures.

7.17.1. Analysis of complexities using different implementations

Implementation	add	Remove	peek
Linked list	$O(1)$	$O(n)$	$O(n)$
Binary heap	$O(\log n)$	$O(\log n)$	$O(1)$
Binary search tree	$O(\log n)$	$O(\log n)$	$O(1)$