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

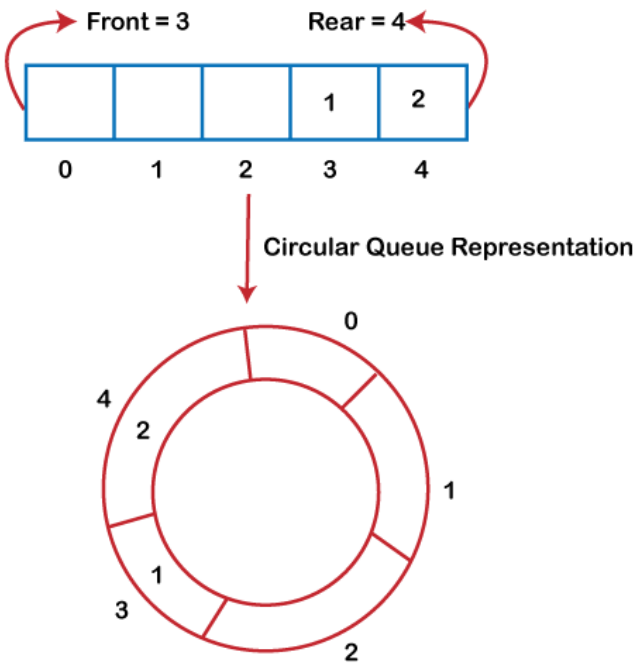
7.0. Objective

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 0^{th} 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 queueas 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., rear=rear+1.
Scenarios for inserting an element
There are two scenarios in which queue is not full:
If rear != max - 1, then rear will be incremented to mod(maxsize) and the new value will be inserted at the rear end of the queue.
If front != 0 and rear = 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 front ==0 && rear = max-1, which means that front is at the first position of the Queue and rear is at the last position of the Queue.
front== rear + 1;

7.5.Algorithm to insert an element in a circular queue

Step 1: IF (REAR+1)%MAX = 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

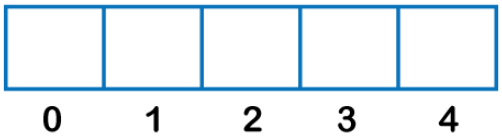
The means of dequeue activity are given underneath:
To start with, we check if the Queue is vacant. In the event that the queue is unfilled, we can't play out the dequeue activity.
At the point when the component is erased, the estimation of front gets decremented by 1.
On the off chance that there is 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

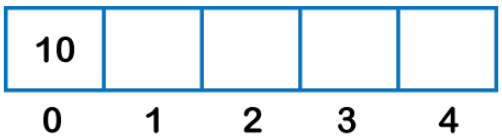
Step 1: IF FRONT = -1
Write " UNDERFLOW "
Goto Step 4
[END of IF]
Step 2: SET VAL = QUEUE[FRONT]
Step 3: IF FRONT = REAR
SET FRONT = REAR = -1
ELSE
IF FRONT = MAX -1
SET FRONT = 0
ELSE
SET FRONT = FRONT + 1
[END of IF]
[END OF IF]

Step 4: EXIT

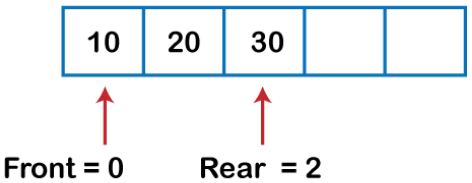
Let's understand the enqueue and dequeue operation through the diagrammatic representation.

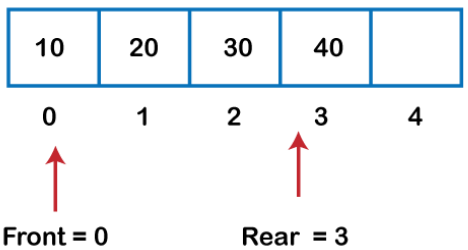


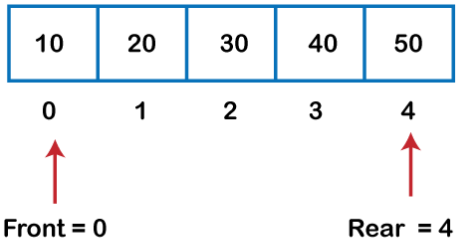
Front = -1
Rear = -1

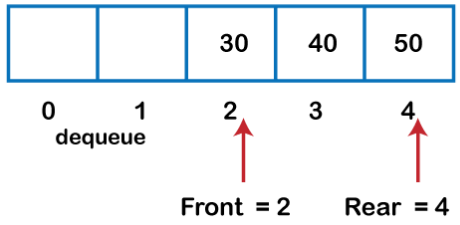


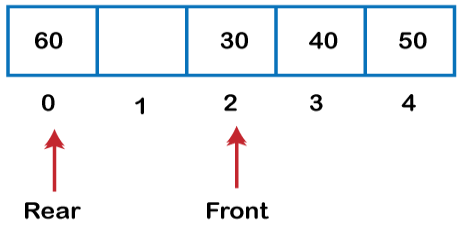
Front = 0
Rear = 0

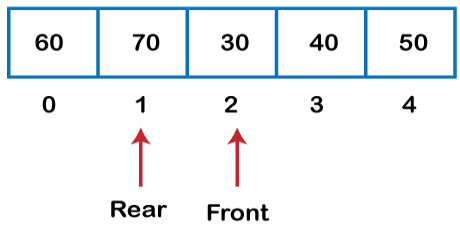












7.7.Implementation of circular queue using Array

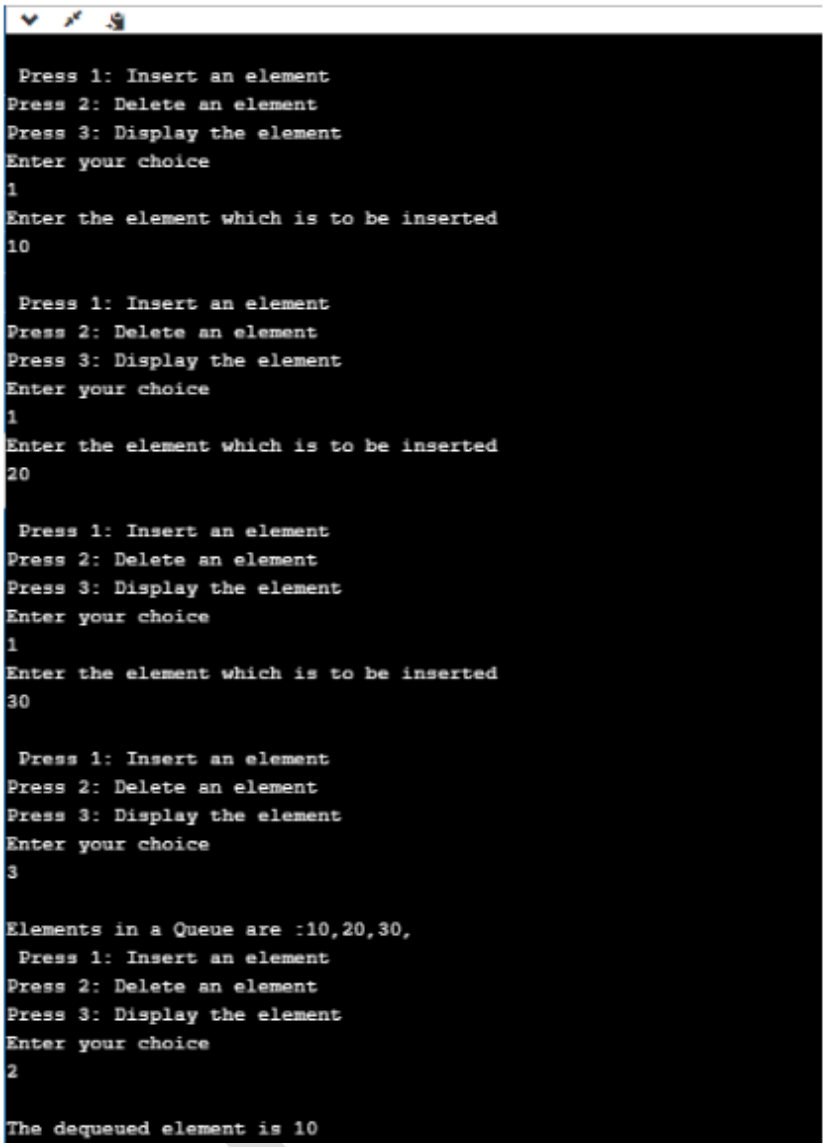
```
#include <stdio.h>
  # define max 6
int queue[max];  // array declaration
int front=-1;
int rear=-1;
// function to insert an element in a circular queue
void enqueue(int element)
{
   if(front==-1 && rear==-1)   // condition to check queue is empty
   {
      front=0;
      rear=0;
      queue[rear]=element;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

```
    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:



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
voidenqueue(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
voiddequeue()
{
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);
```

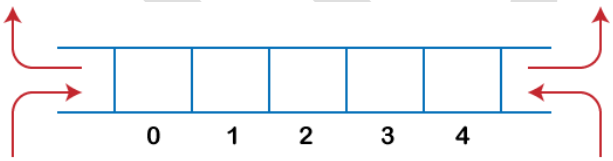
```
    }
}

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 theend at which the erasure happens is known as front end.

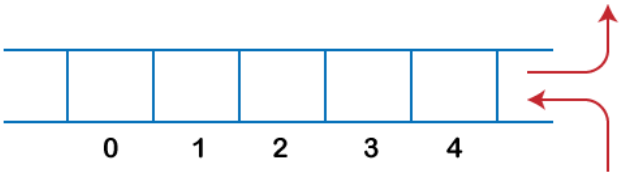


Deque is a direct information structure in which the inclusion and cancellation tasks are performed from the two finishes. We can say that deque is a summed up form of the line.

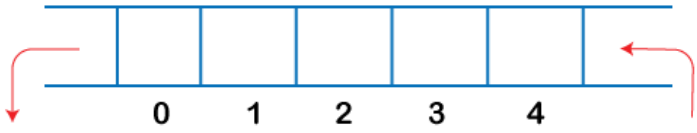
How about we take a gander at certain properties of deque.

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

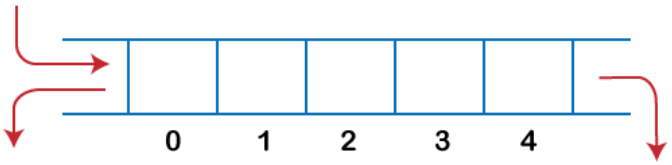
In deque, the inclusion and cancellation activity can be performed from one side. The stack adheres to the LIFO rule in which both the addition and erasure can be performed distinctly from one end; in this way, we reason that deque can be considered as a stack.



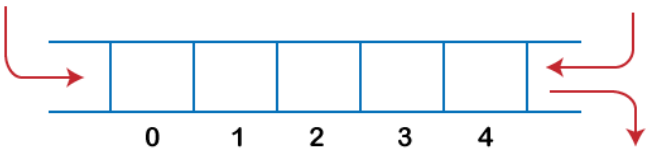
In deque, the addition can be performed toward one side, and the erasure should be possible on another end. The queue adheres to the FIFO rule in which the component is embedded toward one side and erased from another end. Hence, we reason that the deque can likewise be considered as the queue.



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 dequeue.

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.

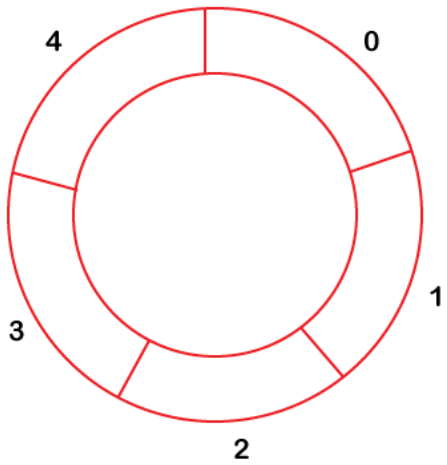
isEmpty(): This capacity restores a genuine worth if the stack is vacant; else it restores a bogus worth.

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.

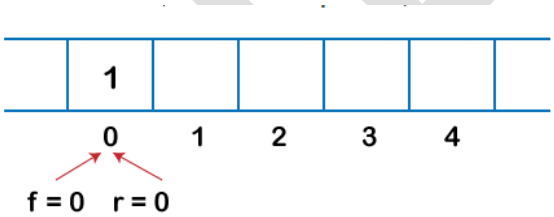
7.11.Implementation of Deque using a circular array

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

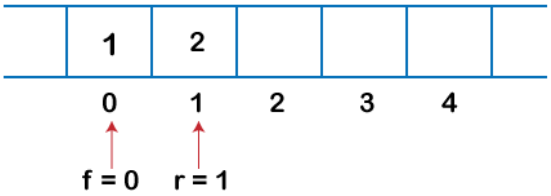
Enqueue operation

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.

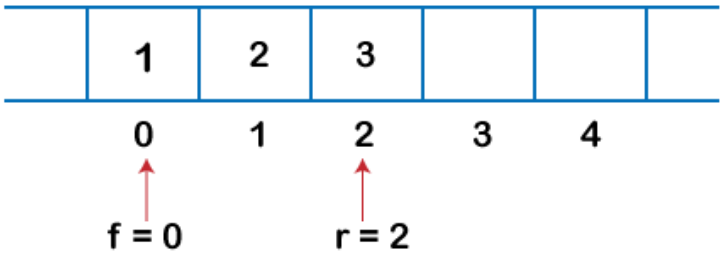
2. As the deque is vacant, so embeddings a component either from the front or backside would be something very similar. Assume we have embedded component 1, at that point front is equivalent to 0, and the back is likewise equivalent to 0.



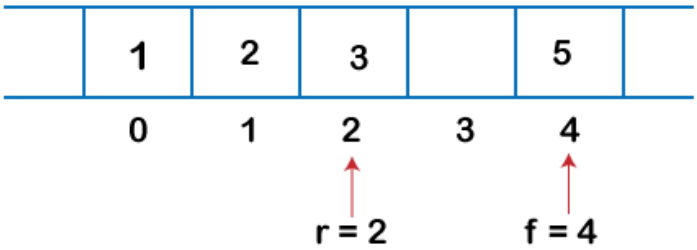
3. Assume we need to embed the following component from the back. To embed the component from the backside, we first need to augment the back, i.e., rear=rear+1. Presently, the back is highlighting the subsequent component, and the front is highlighting the main component.



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.

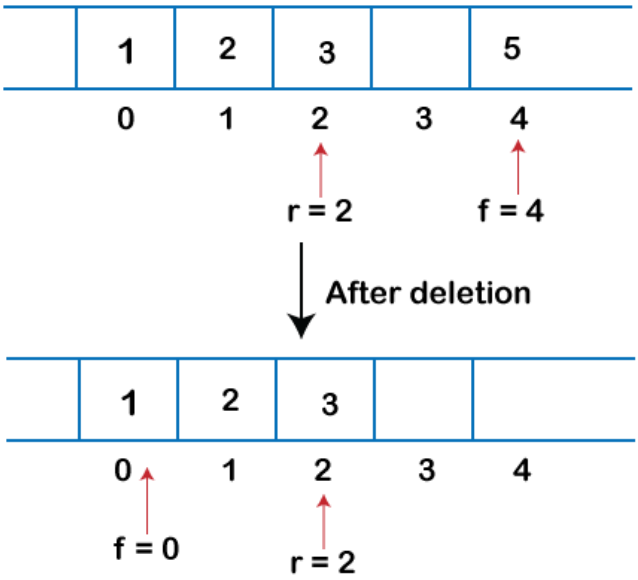


5. In the event that we need to embed the component from the front end, and addition a component from the front, we need to decrement the estimation of front by 1. In the event that we decrement the front by 1, at that point the front focuses to - 1 area, which isn't any substantial area in an exhibit. 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:

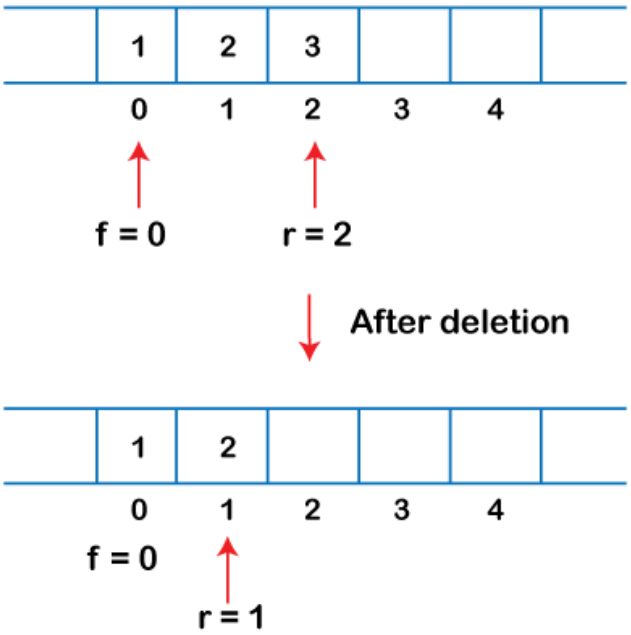


7.12.Dequeue Operation

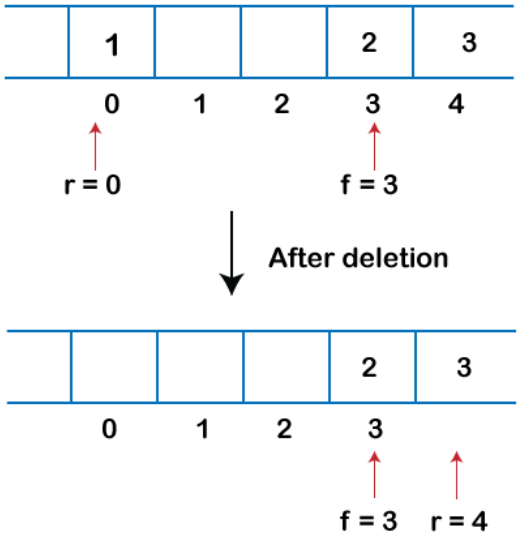
1. On the off chance that the front is highlighting the last component of the exhibit, and we need to play out the erase activity from the front. To erase any component from the front, we need to set front=front+1. At present, the estimation of the front is equivalent to 4, and in the event that we increase 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., rear=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 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))
   {
```

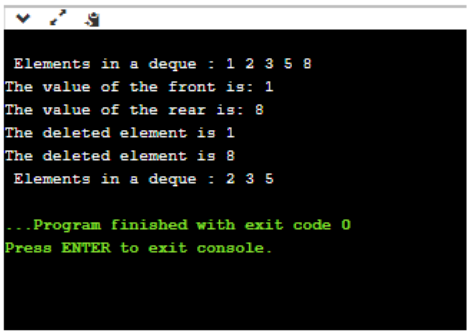
```
        f=r=0;
        deque[f]=x;
    }
    else if(f==0)
    {
        f=size-1;
        deque[f]=x;
    }
    else
    {
        f=f-1;
        deque[f]=x;
    }
}

// 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:



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.

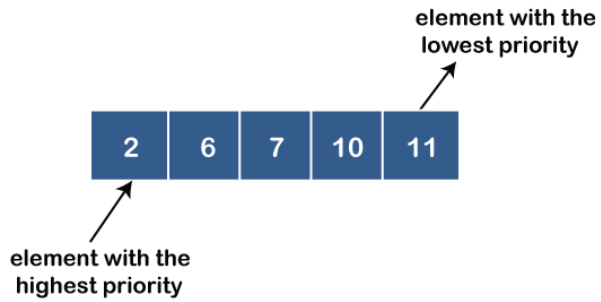
poll() It will eliminate '2' component from the need line as it has the most elevated need line.

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

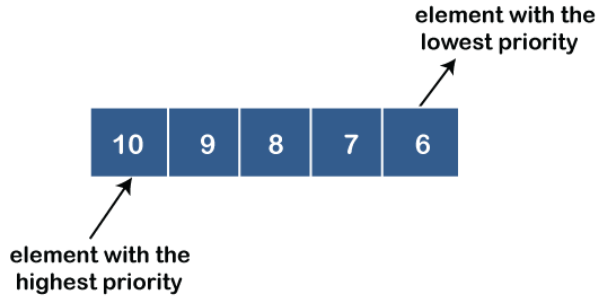
There are two types of priority queue:

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:

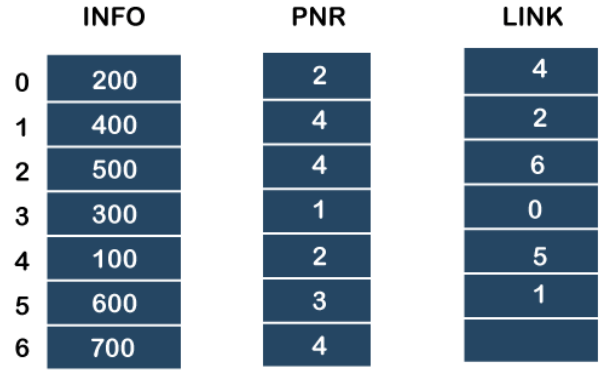
In plunging request need line, a higher need number is given as a higher need in a need. For instance, we take the numbers from 1 to 5 orchestrated in diving request like 5, 4, 3, 2, 1; along these lines, the biggest number, i.e., 5 is given as the most elevated need in a need line**.**



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.



Let's create the priority queue step by step.

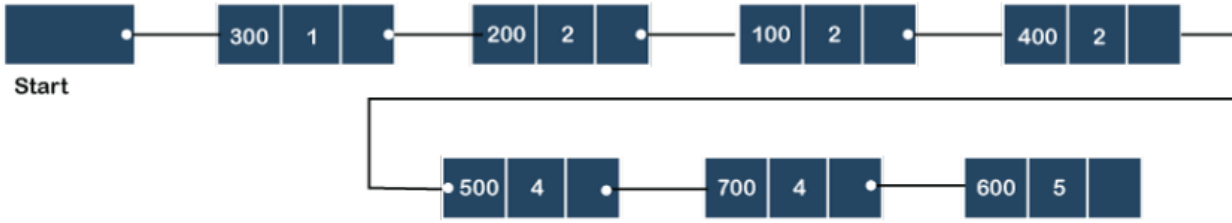
In the case of priority queue, lower priority number is considered the higher priority, i.e., lower priority number = higher priority.

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(logn)	O(logn)	O(1)
Binary search tree	O(logn)	O(logn)	O(1)

Unit 4 : Chapter 8

Linked List

8.0.Objective

This chapter would make you understand the following concepts:

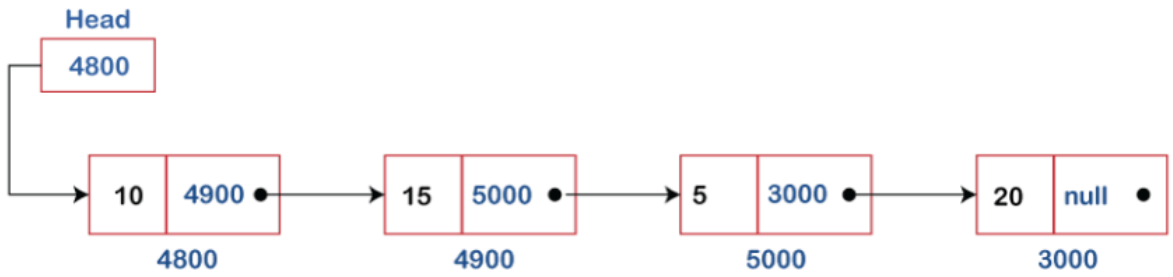
- **To understand the concept of Linked List**
- **To understand Types of Linked List**
- **To Singly Linked list**
- **To Doubly Linked list**

8.1.What is Linked List?

A linked list is also a collection of elements, but the elements are not stored in a consecutive location.Suppose a programmer made a request for storing the integer value then size of 4-byte memory block is assigned to the integer value. The programmer made another request for storing 3 more integer elements; then, three different memory blocks are assigned to these three elements but the memory blocks are available in a random location. So, how are the elements connected?.

These elements are linked to each other by providing one additional information along with an element, i.e., the address of the next element. The variable that stores the address of the next element is known as a pointer. Therefore, we conclude that the linked list contains two parts, i.e., the first one is the data element, and the other is the pointer. The pointer variable will occupy 4 bytes which is pointing to the next element.

A linked list can also be defined as the collection of the nodes in which one node is connected to another node, and node consists of two parts, i.e., one is the data part and the second one is the address part, as shown in the below figure:



In the above figure, we can observe that each node contains the data and the address of the next node. The last node of the linked list contains the NULL value in the address part.

8.2.How can we declare the Linked list?

The need line 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 line, so we will actualize the need line 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.

The structure of a linked list can be defined as:

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

In the above declaration, we have defined a structure named as a node consisting of two variables: an integer variable (data), and the other one is the pointer (next), which contains the address of the next node.

8.3.Advantages of using a Linked list over Array

The following are the advantages of using a linked list over an array:

Dynamic data structure:

The size of the linked list is not fixed as it can vary according to our requirements.

Insertion and Deletion:

Insertion and deletion in linked list are easier than array as the elements in an array are stored in a consecutive location. In contrast, in the case of a linked list, the elements are stored in a random location. The complexity for insertion and deletion of elements from the beginning is O(1) in the linked list, while in the case of an array, the complexity would be O(n). If we want to insert or delete the element in an array, then we need to shift the elements for creating the space. On the other hand, in the linked list, we do not have to shift the elements. In the linked list, we just need to update the address of the pointer in the node.

Memory efficient

Its memory consumption is efficient as the size of the linked list can grow or shrink according to our requirements.

Implementation

Both the stacks and queues can be implemented using a linked list.

Disadvantages of Linked list

The following are the disadvantages of linked list:

Memory usage

The node in a linked list occupies more memory than array as each node occupies two types of variables, i.e., one is a simple variable, and another is a pointer variable that occupies 4 bytes in the memory.

Traversal

In a linked list, the traversal is not easy. If we want to access the element in a linked list, we cannot access the element randomly, but in the case of an array, we can randomly access the element by index. For example, if we want to access the 3rd node, then we need to traverse all the nodes before it. So, the time required to access a particular node is large.

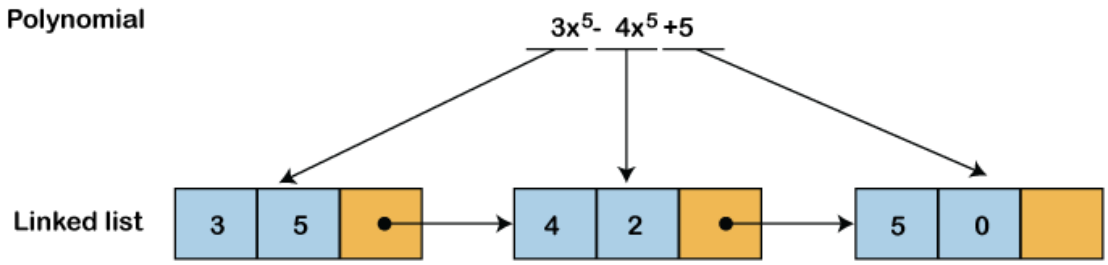
Reverse traversing

In a linked list, backtracking or reverse traversing is difficult. In a doubly linked list, it is easier but requires more memory to store the back pointer.

8.4.Applications of Linked List

The applications of the linked list are given below:

- With the assistance of a connected rundown, the polynomials can be addressed just as we can play out the procedure on the polynomial. We realize that polynomial is an assortment of terms where each term contains coefficient and force. The coefficients and force of each term are put away as hub and connection pointer focuses to the following component in a connected rundown, so connected rundown can be utilized to make, erase and show the polynomial.



- An inadequate grid is utilized in logical calculation and mathematical investigation. In this way, a connected rundown is utilized to address the scanty grid.
- The different tasks like understudy's subtleties, worker's subtleties or item subtleties can be executed utilizing the connected rundown as the connected rundown utilizes the design information type that can hold distinctive information types.
- Stack, Queue, tree and different other information constructions can be executed utilizing a connected rundown.
- The chart is an assortment of edges and vertices, and the diagram can be addressed as a nearness lattice and contiguousness list. In the event that we need to address the diagram as a nearness network, at that point it very well may be actualized as an exhibit. In the event that we need to address the diagram as a nearness list, at that point it tends to be actualized as a connected rundown.
- To actualize hashing, we require hash tables. The hash table contains sections that are executed utilizing connected rundown.
- A connected rundown can be utilized to execute dynamic memory designation. The powerful memory distribution is the memory designation done at the run-time.

8.5.Types of Linked List

Before knowing about the types of a linked list, we should know what is linked list. So, to know about the linked list, click on the link given below:

Types of Linked list

The following are the types of linked list:

Singly Linked list

Doubly Linked list

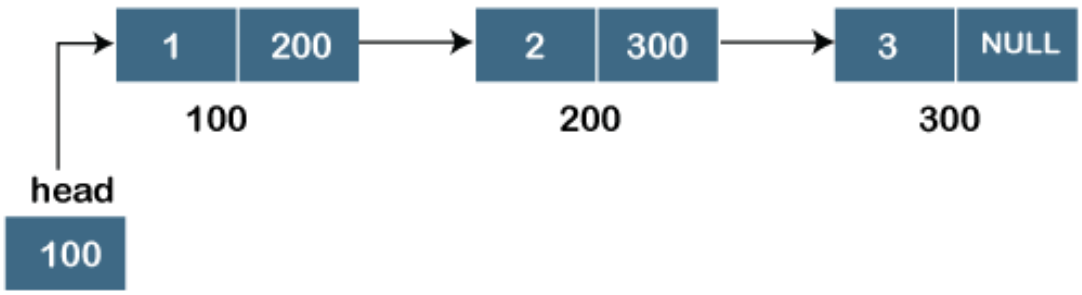
Circular Linked list

Doubly Circular Linked list

8.5.1.Singly Linked list

It is the normally utilized connected rundown in projects. In the event that we are discussing the connected show, it implies it is a separately connected rundown. The separately connected rundown is an information structure that contains two sections, i.e., one is the information part, and the other one is the location part, which contains the location of the following or the replacement hub. The location part in a hub is otherwise called a pointer.

Assume we have three hubs, and the locations of these three hubs are 100, 200 and 300 separately. The portrayal of three hubs as a connected rundown is appeared in the underneath figure:



We can see in the above figure that there are three unique hubs having address 100, 200 and 300 individually. The principal hub contains the location of the following hub, i.e., 200, the subsequent hub contains the location of the last hub, i.e., 300, and the third hub contains the NULL incentive in its location part as it doesn't highlight any hub. The pointer that holds the location of the underlying hub is known as a head pointer.

The connected rundown, which is appeared in the above outline, is referred to as an independently connected rundown as it contains just a solitary connection. In this rundown, just forward crossing is conceivable; we can't navigate the regressive way as it has just one connection in the rundown.

Representation of the node in a singly linked list

```
struct node
{
int data;
```

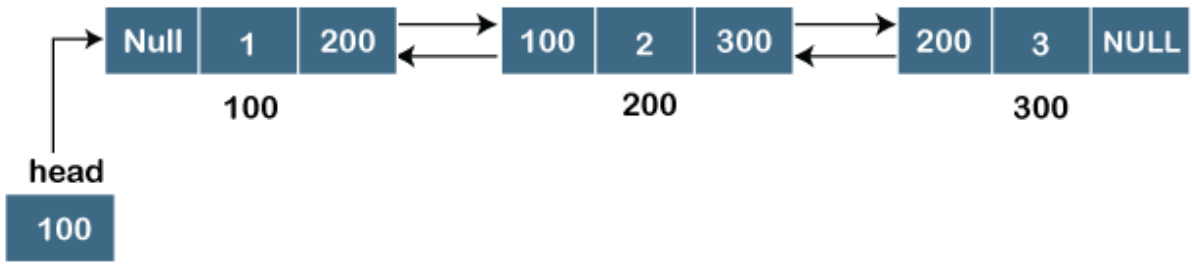
struct node *next;

}

8.5.2.Doubly linked list

As the name recommends, the doubly connected rundown contains two pointers. We can characterize the doubly connected rundown as a straight information structure with three sections: the information part and the other two location part. All in all, a doubly connected rundown is a rundown that has three sections in a solitary hub, incorporates one information section, a pointer to its past hub, and a pointer to the following hub.

Assume we have three hubs, and the location of these hubs are 100, 200 and 300, separately. The portrayal of these hubs in a doubly-connected rundown is appeared beneath:



As we can see in the above figure, the hub in a doubly-connected rundown has two location parts; one section stores the location of the following while the other piece of the hub stores the past hub's location. The underlying hub in the doubly connected rundown has the NULL incentive in the location part, which gives the location of the past hub.

Representation of the node in a doubly linked list

struct node

{

int data;

struct node *next;

struct node *prev;

}

In the above portrayal, we have characterized a client characterized structure named a hub with three individuals, one is information of number sort, and the other two are the pointers, i.e., next and prev of the hub type. The following pointer variable holds the location of the following hub, and the prev pointer holds the location of the past hub. The sort of both the pointers, i.e., next and prev is struct hub as both the pointers are putting away the location of the hub of the struct hub type.

8.5.3.Circular linked list

A round connected rundown is a variety of an independently connected rundown. The lone contrast between the separately connected rundown and a round connected rundown is that the last hub doesn't highlight any hub in an independently connected rundown, so its connection part contains a NULL worth. Then again, the roundabout connected rundown is a rundown where the last hub interfaces with the principal hub, so the connection a piece of the last hub holds the main hub's location. The round connected rundown has no beginning and finishing hub. We can navigate toward any path, i.e., either in reverse or forward. The diagrammatic portrayal of the round connected rundown is appeared underneath:

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

A circular linked list is a sequence of elements in which each node has a link to the next node, and the last node is having a link to the first node. The representation of the circular linked list will be similar to the singly linked list, as shown below: