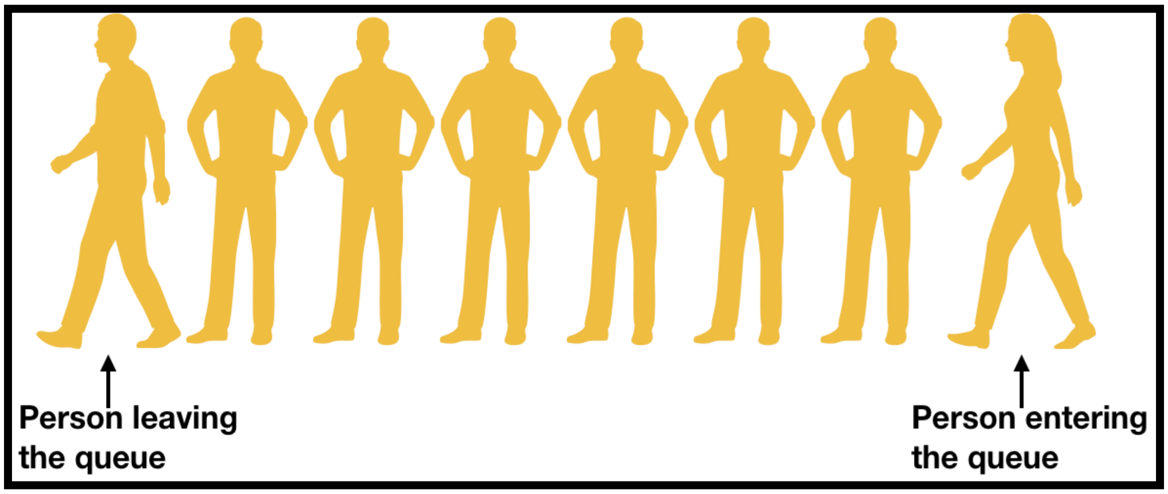
UNIT III
QUEUE

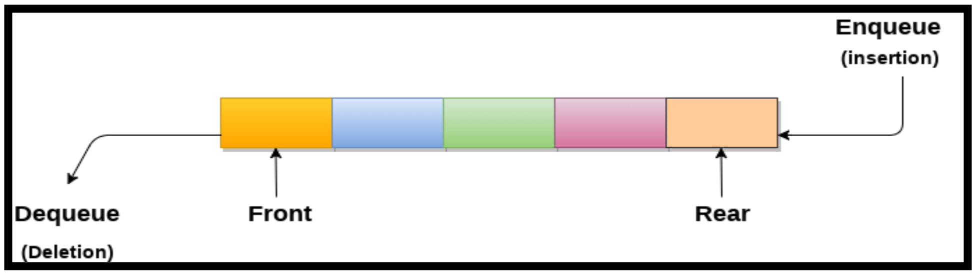
==

Queue:

- Queue is a linear data structure where the element is inserted from one end called REAR and deleted from the other end called as FRONT.
- Front points to the beginning of the queue and Rear points to the end of the queue.
- Queue follows the FIFO (First - In - First Out) structure.
- According to its FIFO structure, element inserted first will also be removed first.
- In a queue, one end is always used to insert data (enqueue) and the other is used to delete data (dequeue), because queue is open at both its ends.
- The enqueue() and dequeue() are two important functions used in a queue.
- For example. At railway station people stand in a queue (line) to get tickets and the person who will be standing first in queue will get the tickets first and also moved out first from the queue.



Pictorial Representation:



Applications of Queue

Due to the fact that the queue performs actions on first in first out basis which is quite fair for the ordering of actions. There are various applications of queues discussed as below.

1. Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
2. Queues are used in asynchronous transfer of data (where data is not being transferred at the same rate between two processes) for eg. pipes, file IO, sockets.
3. Queues are used as buffers in most of the applications like MP3 media player, CD player, etc.
4. Queues are used to maintain the playlist in media players in order to add and remove the songs from the play-list.
5. Queues are used in operating systems for handling interrupts.

Basic Operations of Queue:

A queue allows the following operations:

- Enqueue: Add an element to the end of the queue
- Dequeue: Remove an element from the front of the queue
- IsEmpty: Check if the queue is empty
- IsFull: Check if the queue is full
- Peek: Get the value of the front of the queue without removing it

Working of Queue

Queue operations work as follows:

- Two pointers FRONT and REAR
- FRONT track the first element of the queue
- REAR track the last elements of the queue
- initially, set value of FRONT and REAR to -1

Types of Queue:

1. Linear Queue
2. Circular Queue
3. Dequeue (Double Ended Queue)
4. Priority Queue

1. **Linear Queue:** In Linear Queue, insertion takes place from one end while the deletion occurs from another end. The end at which the insertion takes place is known as the rear end, and the end at which the deletion takes place is known as front end. It strictly follows the FIFO rule. The linear Queue can be represented, as shown in the below figure:

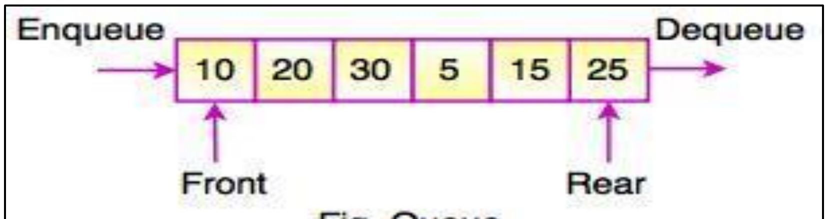


Fig. Queue

Linear Queue Representation:

Queue can be represented in 2 ways:

i. Using array

ii. Using Linked List

1. **Array Representation:**

Array is the easiest way to implement a queue. Queue can be also implemented using Linked List.

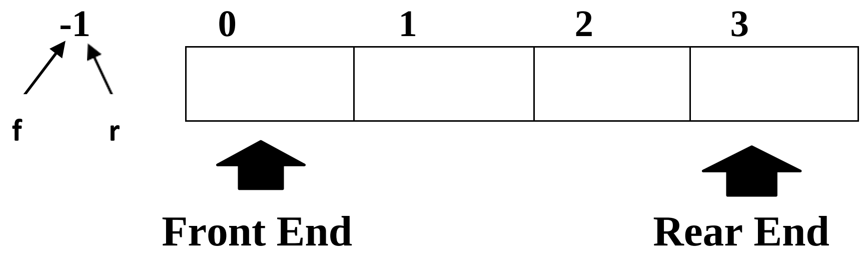
Pictorial examples for inserting element (ENQUEUE) in queue:

Example 1: Insert 10,20,30,40 in the queue. Consider queue size as 4.

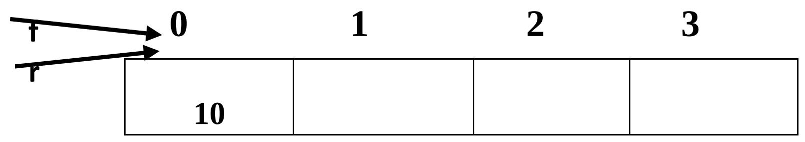
While inserting element follow these rules:

- when queue is empty then set REAR and FRONT to -1
- Only for the first element, set value of FRONT to 0
- Increase the REAR index by 1 and don't increase FRONT
- add the new element in the position pointed to by REAR

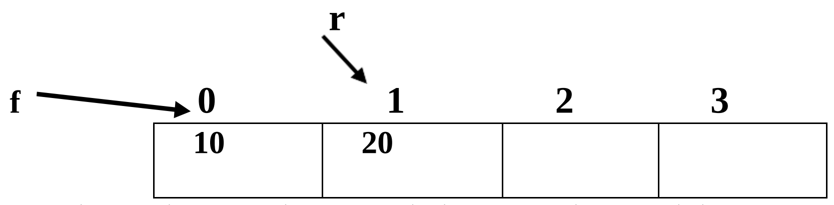
Step 1: initially queue will be empty so set value of f and r to -1



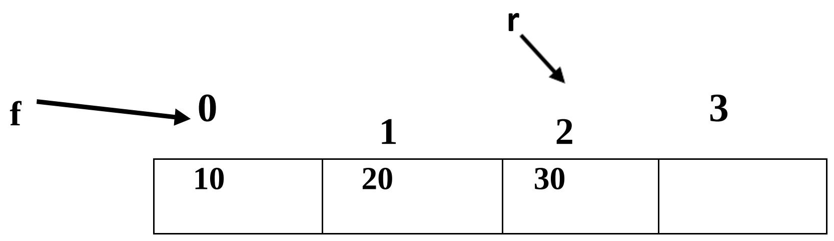
Step 2: for the first element set the value of front and rear to 0.



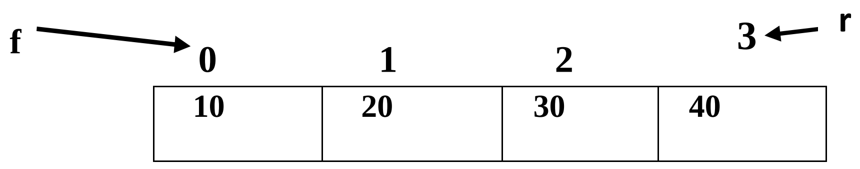
Step 3: to insert next element only increase r by 1 and don't increase f.



Step 4: to insert the next element only increase r by 1 and don't increase f.

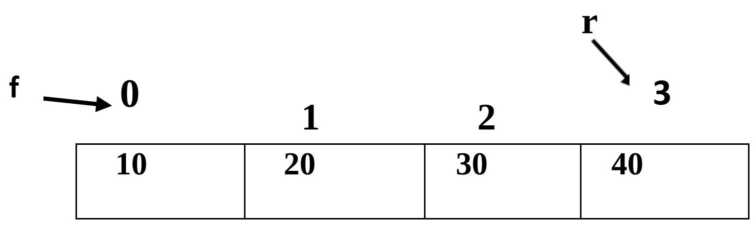


Step 5: to insert next element only increase r by 1 and don't increase f.



Pictorial examples for deleting element(DE-QUEUE) from queue:

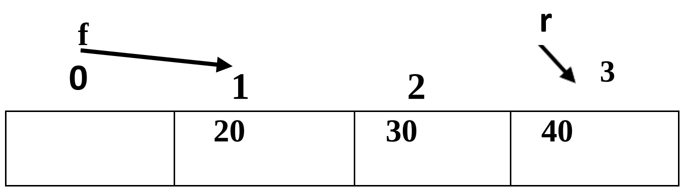
Example 1: delete elements from the given queue.

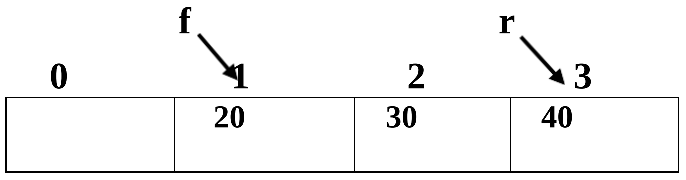


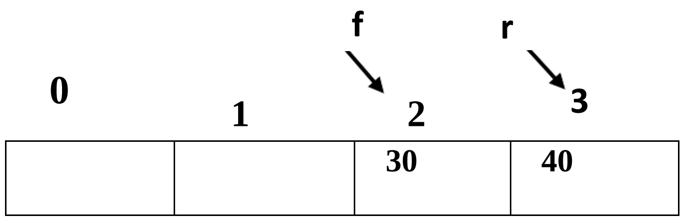
While deleting element follow these rules:

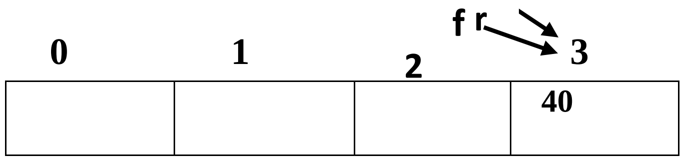
- Increase the FRONT index by 1
- For the last element, reset the values of FRONT and REAR to -1

Step 1: increase f by 1 and don't increase r

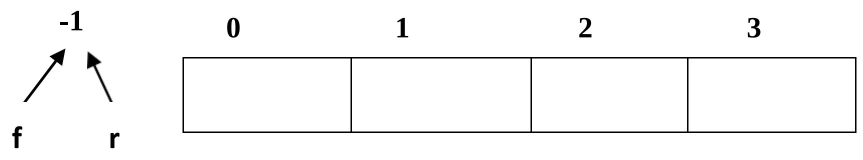








Now, After deleting last element again set value of f and r to -1



Steps for Enqueue Operation

- check if the queue is full
- for the first element, set value of FRONT to 0
- increase the REAR index by 1
- add the new element in the position pointed to by REAR

Steps for Dequeue Operation

- check if the queue is empty
- return the value pointed by FRONT
- increase the FRONT index by 1
- for the last element, reset the values of FRONT and REAR to -1

Algorithm to insert any element in a queue:

- Check if the queue is already full by comparing rear to max - 1. if so, then return an overflow error.
- If the item is to be inserted as the first element in the list, in that case set the value of front and rear to 0 and insert the element at the rear end.
- Otherwise keep increasing the value of the rear and insert each element one by one having rear as the index.

Algorithm:

```
o Step 1: IF REAR = MAX -
  1 Write OVERFLOW
  Go to step
  [END OF IF]
o Step 2: IF FRONT = -1 and REAR = -1
  SET FRONT = REAR = 0
  ELSE
  SET REAR = REAR + 1
  [END OF IF]
o Step 3: Set QUEUE[REAR] = NUM
o Step 4: EXIT
```

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.

Algorithm

```
o Step 1: IF FRONT = -1 or FRONT > REAR
  Write UNDERFLOW
  ELSE
  SET VAL = QUEUE[FRONT]
  SET FRONT = FRONT + 1
  [END OF IF]
o Step 2: EXIT
```

Program to Implement Queue using Array:

```
#include<stdio.h>
#include<stdlib.h>
#define size 5
int q[size],front=-1,rear=-1;
void enqueue();
void dequeue();
void display();
int main()
{
int ch;
while(1)
{
printf("Enter 1. to insert element in the queue\n");
printf("Enter 2. to delete elements of the queue\n");
printf("Enter 3. to display elements of the queue\n");
printf("Enter 4. to exit\n");
scanf("%d",&ch);
if(ch==1)
{
enqueue();
}
else if(ch==2)
{
dequeue();
}
else if(ch==3)
{
display();
}
else
{
exit(0);
}
}
return 0;
}
//=================== enqueue()========================
void enqueue()
{
int data;
printf("Enter elements\n");
scanf("%d",&data);
if(rear==size-1)
{
	printf("**Queue is full**\n");
}
else if(rear==-1 && front==-1)
{
front=rear=0;
q[rear]=data;
printf("**Data inserted successfully**\n");
}
else
{
rear++;
q[rear]=data;
printf("**Data inserted successfully**\n");
}
}
//====================dequeue=================
void dequeue()
{
if(front==-1 && rear==-1)
{
printf("**Queue is empty**\n");
}
else if(front==rear)
{
front=rear=-1;
printf("**Queue is empty**\n");
}
else
{
front++;
printf("** Elements removed successfully **\n");
}
}
```

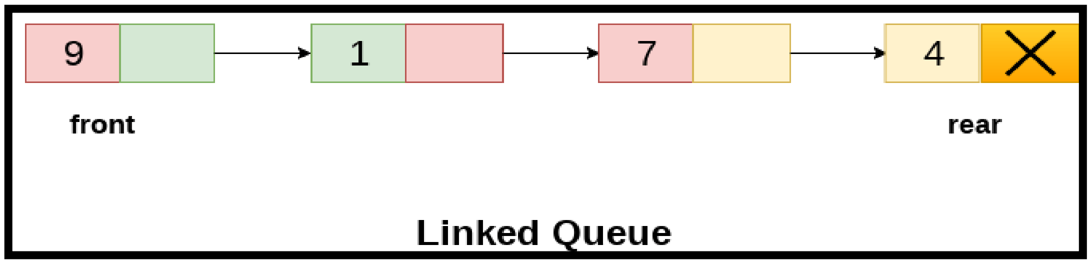
```
//=========================display()=============
void display()
{
int i;
if(front==-1 && rear==-1)
{
printf("**Queue is empty**\n");
}
else
{
printf("\n**ELEMENTS ARE**\n");
for(i=front;i<=rear;i++)
{
printf("%d\n",q[i]);
}
}
```

Output:

Run the code and write down the output

2. Linear Queue Implementation using Linked List:

- In a linked queue, each node of the queue consists of two parts i.e. data part and the link part. Each element of the queue points to its immediate next element in the memory.
- In the linked queue, there are two pointers maintained in the memory i.e. front pointer and rear pointer.
- **The front pointer contains the address of the starting element of the queue while the rear pointer contains the address of the last element of the queue.**
- Insertion and deletions are performed at rear and front end respectively. If front and rear both are NULL, it indicates that the queue is empty.
- While performing queue using linked list, order should be constant i.e O(1).
  The linked representation of the queue is shown in the following figure.



Queue program using linked list

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
int info;
struct node *link;
};
struct node *front=NULL,*rear=NULL;
//==========================================
void enqueue()
{
struct node *n,*t;
n=(struct node*)malloc(sizeof(struct node));
printf("\nEnter value\n");
scanf("%d",&n->info);
n->link=NULL;
if(front==NULL)
{
front=n;
rear=n;//rear will point to last node for o(1)
printf("Data inserted\n");
}
else
{
rear->link=n;
rear=n;

printf("data inserted\n");
}
}
//============================================
void dequeue()
{
struct node *r;
if(front==NULL)
{
printf("\n ** queue is empty **\n");
}
else
{
r=front;
front=r->link;
free(r);
printf("\n ** deleted element **\n");

}
}
```

```
//==============================================

void display()
{
struct node *r;
r = front;
if(front==NULL)
{
printf("\n **Queue is empty**\n");
}
else
{
printf("\n **Data are**\n");
while(r!=NULL)
{
printf("\n%d",r->info);
r=r->link;
}
}
}
//==============================================
int main()
{
int ch;
while(1)
{
printf("\nEnter 1. to insert element in the queue\n");

printf("Enter 2. to delete elements of the queue\n");
printf("Enter 3. to display elements of the queue\n");
printf("Enter 4. to exit\n");
scanf("%d",&ch);
if(ch==1)
{
enqueue();
}
else if(ch==2)
{
dequeue();
}
else if(ch==3)
{
display();
}
else
{
exit(0);
}
}
return 0;
}
```

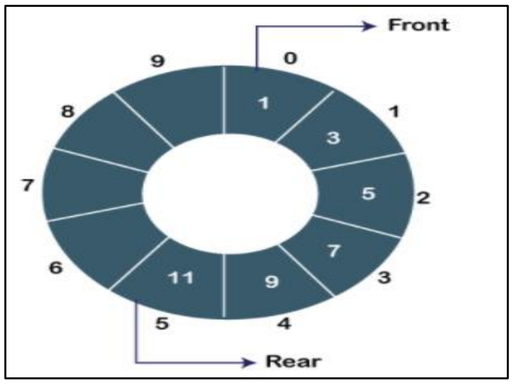
Output:

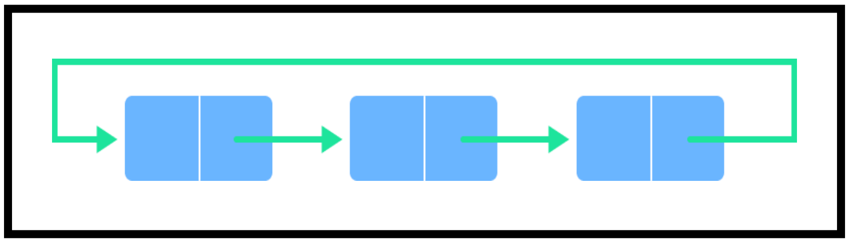
Execute the code and write down the output

2. Circular Queue:

Definition:

In Circular Queue, all the nodes are represented as circular. It is similar to the linear Queue except that the last element of the queue is connected to the first element. It is also known as **Ring Buffer** as all the ends are connected to another end. The circular queue can be represented as:



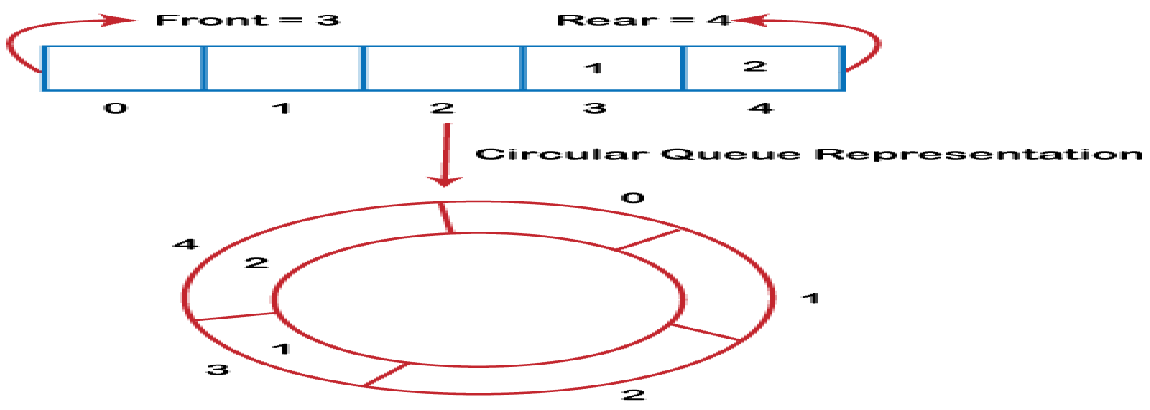


Circular Queue Representation

The drawback that occurs in a linear queue is overcome by using the circular queue. If the empty space is available in a circular queue, the new element can be added in an empty space by simply incrementing the value of rear.

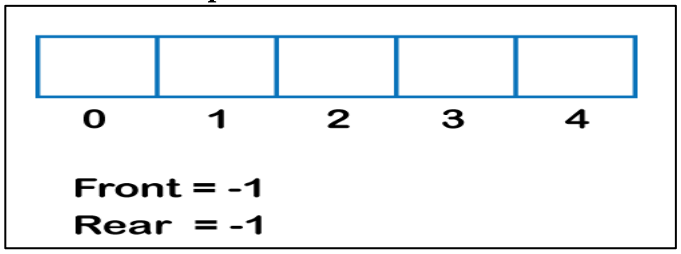
Why was the concept of the circular queue introduced?

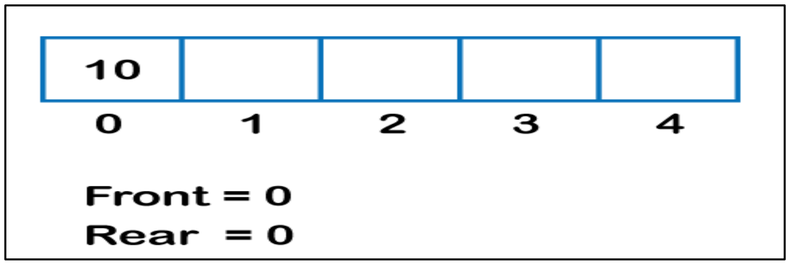
There was one limitation in the array implementation of Queue. If the rear reaches to the end position of the Queue then there might be possibility that some vacant spaces are left in the beginning which cannot be utilized. So, to overcome such limitations, the concept of the circular queue was introduced.

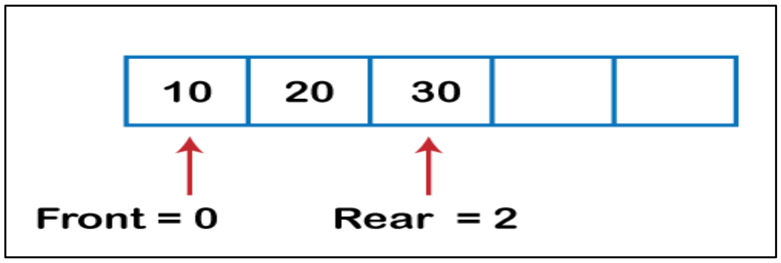


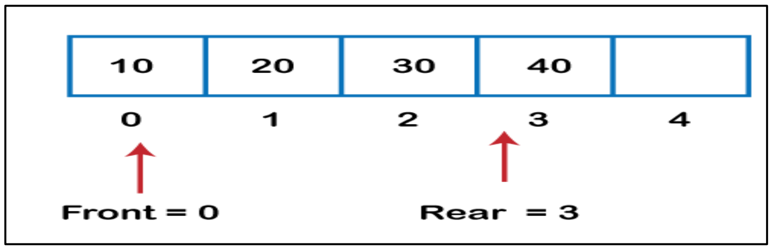
As we can see in the above image, the rear is at the last position of the Queue and front is pointing somewhere rather than the 0th position. In the above array, there are only two elements and other three positions are empty. The rear is at the last position of the Queue; if we try to insert the element then it will show that there are no empty spaces in the Queue. There is one solution to avoid such wastage of memory space by shifting both the elements at the left and adjust the front and rear end accordingly. It is not a practically good approach because shifting all the elements will consume lots of time. The efficient approach to avoid the wastage of the memory is to use the circular queue data structure.

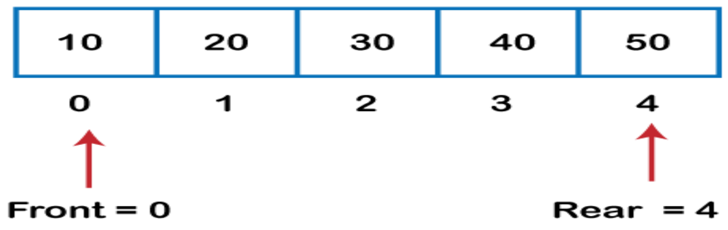
Pictorial Example:

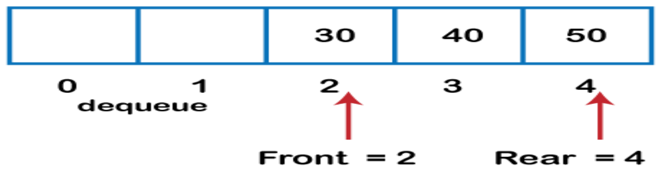


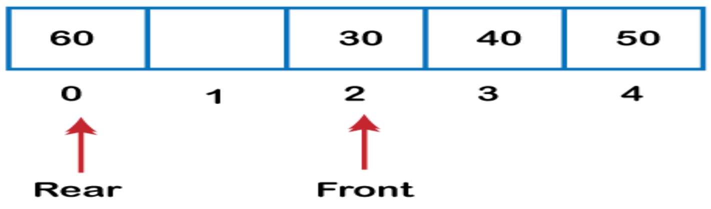


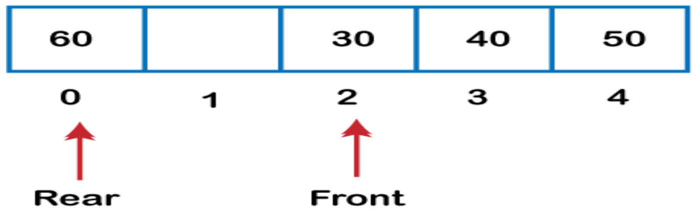












Scenarios for inserting an element:

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;

Implementation of circular queue using Array:

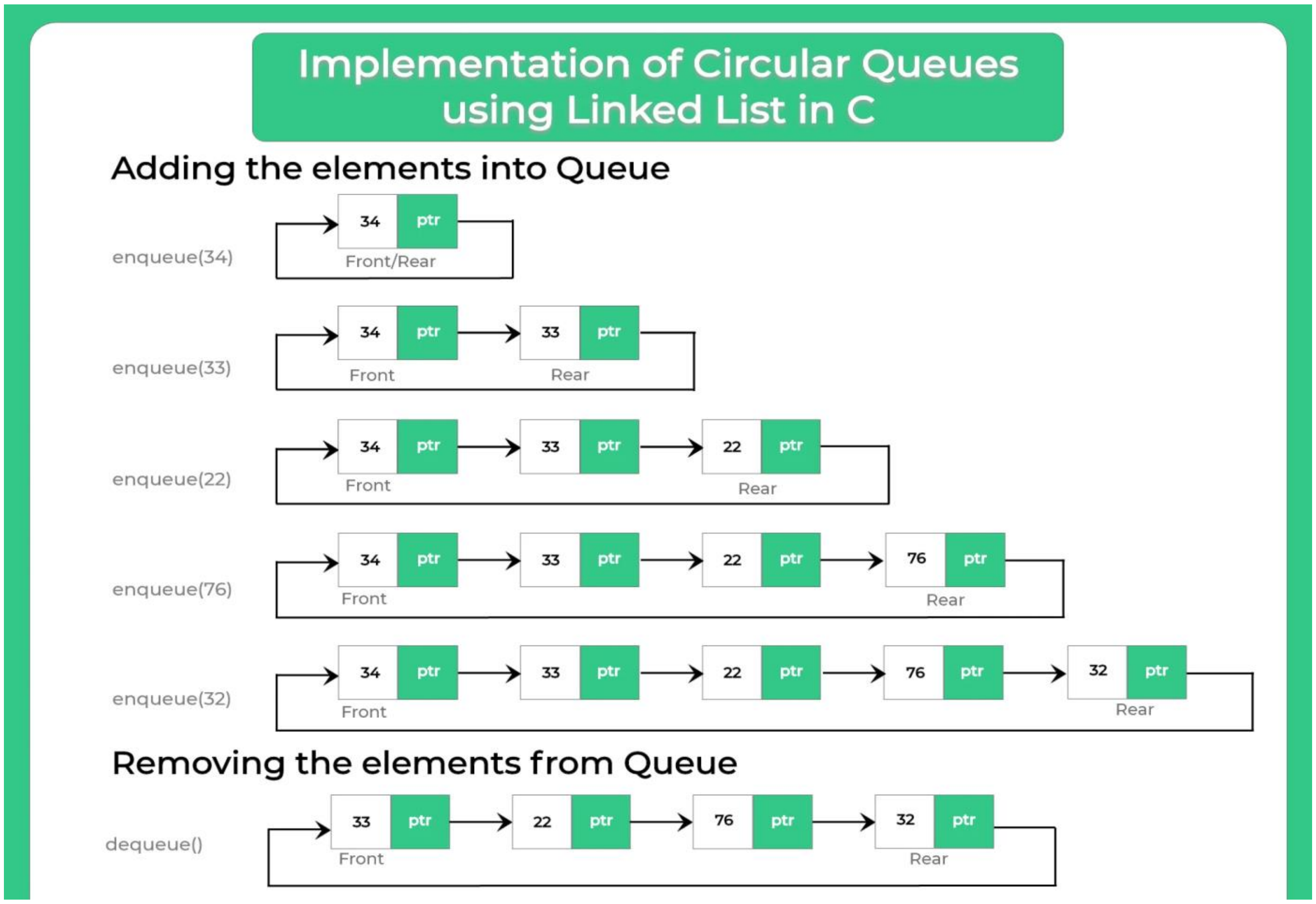
```
#include<stdio.h>
#include<stdlib.h>
#define size 5

int q[size],front=-1,rear=-1;
void enqueue()
```

```
{
if((rear==size-1 && front==0) || rear==front-1)//full
{
printf("\n queue is full \n");
}
else if(front==-1 && rear==-1)//1st data
{
front=0;
rear=0;
printf("\n enter data \n");
scanf("%d",&q[rear]);
printf("\n data inserted \n ");
}
else if(rear==size-1 && front!=0)//circular
{
rear=0;
printf("\n enter data \n");
scanf("%d",&q[rear]);
printf("\n data inserted \n ");
}
else
{
rear++;
printf("\n enter data \n");
scanf("%d",&q[rear]);
printf("\n data inserted \n ");
}
}
//
void dequeue()
{
if(front==-1 && rear==-1)//empty
{
printf("\n queue is empty\n");
}
else if(front==rear)//last element
{
front=-1;
rear=-1;
printf("\n data deleted \n");
}
else if (front !=size-1)
{
front++;
printf("\n data deleted \n");

}
else if(front==size-1 && rear>=0)//circular
{
front=0;
printf("\n data deleted \n");
}
}
//
void display()
{
   int i,j;
if(front==-1 && rear==-1)
printf("\n queue is empty \n");
if(rear<front)
{
for(i=front;i<=size-1;i++)
{
printf("%d ",q[i]);
}
for(j=0;j<=rear;j++)
{
printf("%d ",q[j]);
}
}
else
{
for(i=front;i<=rear;i++)
{
printf("%d ",q[i]);
}}}
int main()
{
   int ch;
while(1)
{
printf("\n1.insert\n2.delete\n3.dispaly\n4.exit\n");
scanf("%d",&ch);
if(ch==1)
enqueue();
else if(ch==2)
dequeue();
else if(ch==3)
display();
else
exit(0);
}

}
```

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

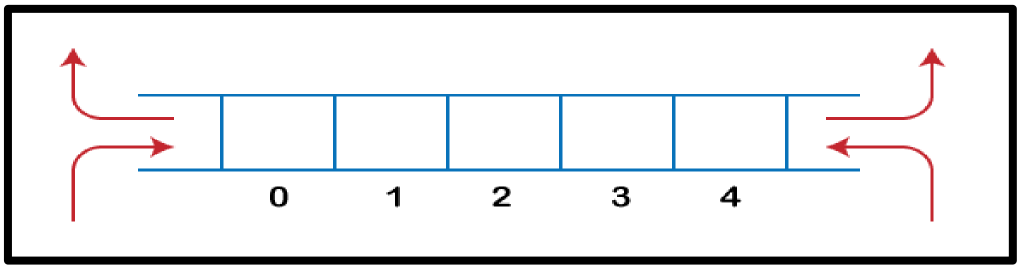
- Front pointer holds address of first node.
- Rear holds address of last node and last node points to the first node.

Let's see how to implement Circular Queue using Linked list in C Programming.


Implementation of Circular Queues using Linked List in C
Adding the elements into Queue
enqueue(34)
34 ptr
Front/Rear
enqueue(33)
34 ptr 33 ptr
Front Rear
enqueue(22)
34 ptr 33 ptr 22 ptr
Front Rear
enqueue(76)
34 ptr 33 ptr 22 ptr 76 ptr
Front Rear
enqueue(32)
34 ptr 33 ptr 22 ptr 76 ptr 32 ptr
Front Rear
Removing the elements from Queue
dequeue()
33 ptr 22 ptr 76 ptr 32 ptr
Front Rear

Program to Implement circular queue using linked list.
```
#include<stdio.h>
#include<stdlib.h>
struct node
{
        int data;
        struct node* next;
};
struct node *f = NULL;
struct node *r = NULL;
void enqueue() //Insert elements in Queue
{
        struct node* n;
        n = (struct node*)malloc(sizeof(struct node));
        printf("enter data\n");
        scanf("%d",&n->data);
        n->next = NULL;
        if((r==NULL)&&(f==NULL))
        {
                f = r = n;
                r->next = f;
                printf("\n NODE INSERTED \n");
        }
        else
        {
                r->next = n;
                r = n;
                n->next = f;
                printf("\n NODE INSERTED \n");
        }
}
void dequeue() // Delete an element from Queue
{
        struct node* t;
        t = f;
        if((f==NULL)&&(r==NULL))
                printf("\nQueue is Empty");
        else if(f == r)
        {
                f = r = NULL;
                free(t);
                printf("\n NODE DELETED \n");
        }
        else
        {
                f = f->next;
                r->next = f;
                free(t);
                printf("\n NODE DELETED \n");
        }
}
void print()
{ // Print the elements of Queue
        struct node* t;
        t = f;
        if((f==NULL)&&(r==NULL))
                printf("\nQueue is Empty");
        else
        {
                do
                {
                        printf("\n%d",t->data);
                        t = t->next;
                }while(t != f);
        }
}
int main()
{
        int opt,n,i;
          while(1)
        {
        printf("\nEnter Your Choice:-");
        printf("\n\n1 for Insert the Data in Queue\n2 for delete the Data in Queue \n3 for Display the data
from the Queue\n0 for Exit");
                scanf("%d",&opt);

                 if(opt==1)
                 enqueue();
                 else if(opt==2)
                 dequeue();
                 else if (opt==3)
                 print();
                 else
                 exit(0);

           }
        return 0;
}
```

3. **Dequeue:**
   The dequeue stands for **Double Ended Queue**. In the queue, the insertion takes place from one end while the deletion takes place from another end. The end at which the insertion occurs is known as the **rear end** whereas the end at which the deletion occurs is known as **front end**.

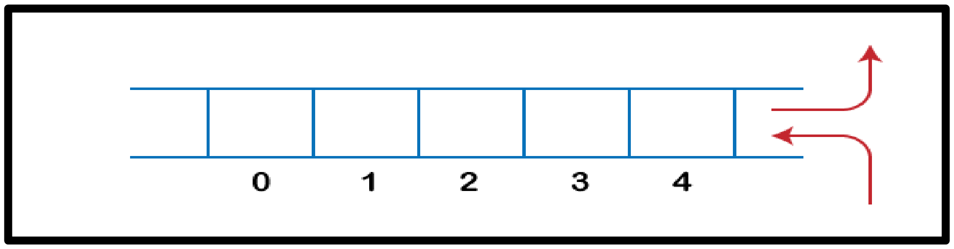


Deque is a linear data structure in which the insertion and deletion operations are performed from both ends. We can say that deque is a generalized version of the queue.

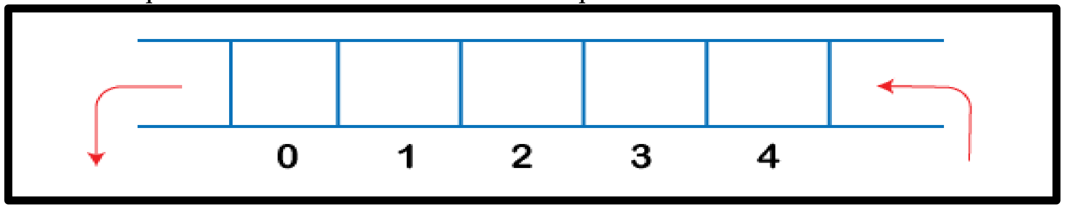
Let's look at some properties of deque.

- Deque can be used both as **stack** and **queue** as it allows the insertion and deletion operations on both ends.

In deque, the insertion and deletion operation can be performed from one side. The stack follows the LIFO rule in which both the insertion and deletion can be performed only from one end; therefore, we conclude that deque can be considered as a stack.

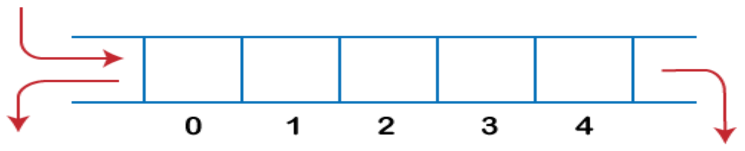


In deque, the insertion can be performed on one end, and the deletion can be done on another end. The queue follows the FIFO rule in which the element is inserted on one end and deleted from another end. Therefore, we conclude that the deque can also be considered as the queue.

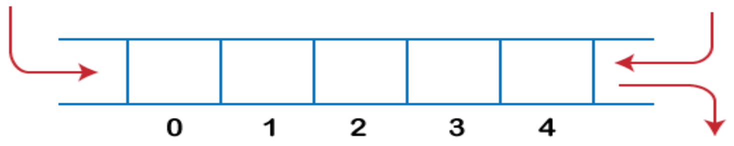


There are two types of Queues, **Input-restricted queue**, and **output-restricted queue**.

1. **Input-restricted queue:** The input-restricted queue means that some restrictions are applied to the insertion. In input-restricted queue, the insertion is applied to one end while the deletion is applied from both the ends.



2. **Output-restricted queue:** The output-restricted queue means that some restrictions are applied to the deletion operation. In an output-restricted queue, the deletion can be applied only from one end, whereas the insertion is possible from both ends.



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 insertion and deletion, we can also perform **peek** operation in deque. Through **peek** operation, we can get the **front** and the **rear** element of the dequeue.

We can perform two more operations on dequeue:

- **isFull():** This function returns a true value if the stack is full; otherwise, it returns a false value.
- **isEmpty():** This function returns a true value if the stack is empty; otherwise it returns a false value.

Memory Representation:

The deque can be implemented using two data structures, i.e., **circular array**, and **doubly linked list**. To implement the deque using circular array, we first should know **what is circular array**.

Applications of Deque:

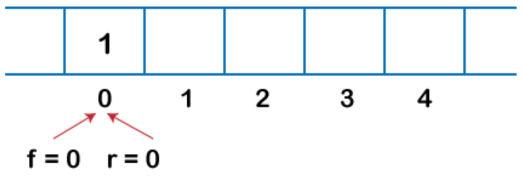
- The deque can be used as a **stack** and **queue**; therefore, it can perform both redo and undo operations.
- It can be used as a palindrome checker means that if we read the string from both ends, then the string would be the same.
- It can be used for multiprocessor scheduling. Suppose we have two processors, and each processor has one process to execute. Each processor is assigned with a process or a job, and each process contains multiple threads. Each processor maintains a deque that contains threads that are ready to execute. The processor executes a process, and if a process creates a child process then that process will be inserted at the front of the deque of the parent process. Suppose the processor P_2 has completed the execution of all its threads then it steals the thread from the rear end of the processor P_1 and adds to the front end of the processor P_2. The processor P_2 will take the thread from the front end; therefore, the deletion takes from both the ends, i.e., front and rear end. This is known as the **A-steal algorithm** for scheduling.

Implementation of Deque using a circular array

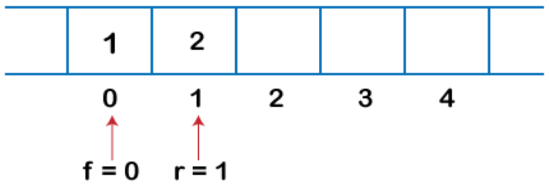
The following are the steps to perform the operations on the Deque:

Enqueue operation:

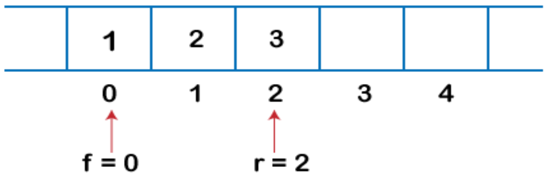
1. Initially, we are considering that the deque is empty, so both front and rear are set to -1, i.e., **f = -1** and **r = -1**.
2. As the deque is empty, so inserting an element either from the front or rear end would be the same thing. Suppose we have inserted element 1, then **front is equal to 0,** and the **rear is also equal to 0.**



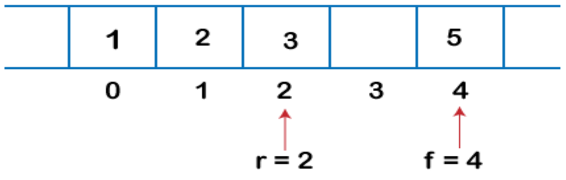
3. Suppose we want to insert the next element from the rear. To insert the element from the rear end, we first need to increment the rear, i.e., **rear=rear+1**. Now, the rear is pointing to the second element, and the front is pointing to the first element.



4. Suppose we are again inserting the element from the rear end. To insert the element, we will first increment the rear, and now rear points to the third element.

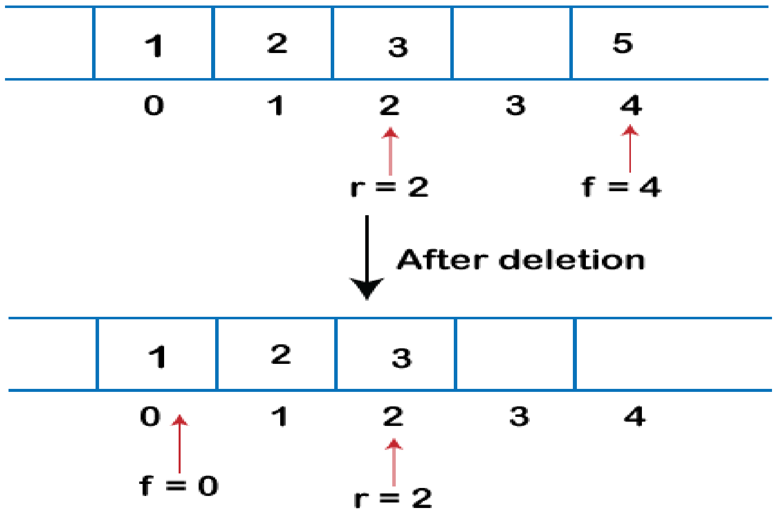


5. If we want to insert the element from the front end, and insert an element from the front, we have to decrement the value of front by 1. If we decrement the front by 1, then the front points to -1 location, which is not any valid location in an array. So, we set the front as **(n -1),** which is equal to 4 as n is 5. Once the front is set, we will insert the value as shown in the below figure:

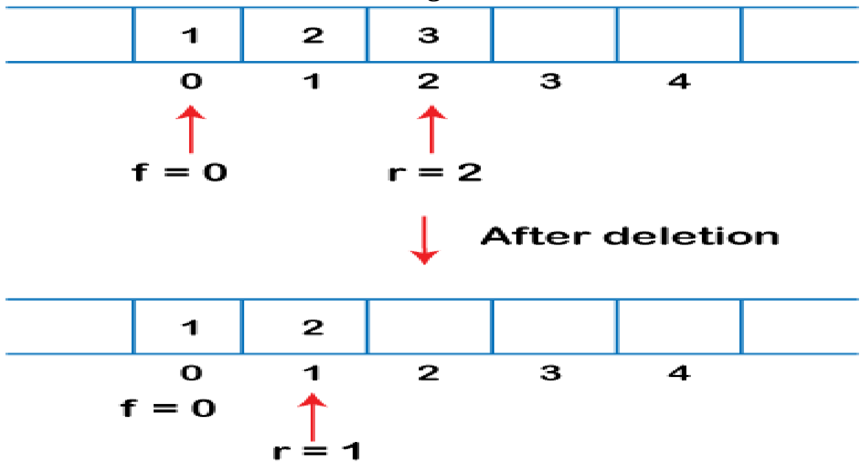


Dequeue Operation

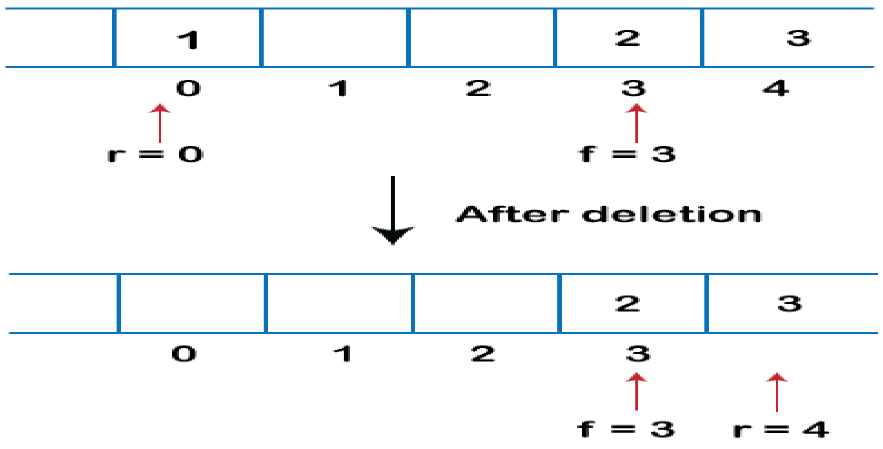
1. If the front is pointing to the last element of the array, and we want to perform the delete operation from the front. To delete any element from the front, we need to set **front=front+1**. Currently, the value of the front is equal to 4, and if we increment the value of front, it becomes 5 which is not a valid index. Therefore, we conclude that if front points to the last element, then front is set to 0 in case of delete operation.



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. If the rear is pointing to the first element, and we want to delete the element from the rear end then we have to set **rear=n-1** where **n** is the size of the array as shown in the below 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.

```
#define size 5
#include <stdio.h>
#include<stdlib.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;
        printf("\n DATA DELETED \n");
    }
    else if(f==0)
    {
        f=size-1;
        deque[f]=x;
        printf("\n DATA DELETED \n");
    }
    else
    {
        f=f-1;
        deque[f]=x;
        printf("\n DATA DELETED \n");

    }
}

// 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))
    {
        f=r=0;
        deque[r]=x;
        printf("\n DATA INSERTED \n");

    }
    else if(r==size-1)
    {
        r=0;
        deque[r]=x;
        printf("\n DATA INSERTED \n");
    }
    else
    {
        r++;
        deque[r]=x;
        printf("\n DATA INSERTED \n");
    }
}
// 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]);
}
  // 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()
{

  int opt,n,i;
               while(1)
           {
           printf("\nEnter Your Choice:-");
           printf("\n\n1 for InsertRear\n2 InsetFront \n3 DeleteFront\n 4.DeleteRear\n  5.display\n6.exit\n");
           scanf("%d",&opt);
           if(opt==1)
                     {
                                   printf("enter the data to insert\n");
                                   scanf("%d",&n);
                                   enqueue_rear(n);
                     }
                     else if(opt==2)
                     {
                                   printf("enter the data to insert\n");
                                   scanf("%d",&n);
                                   enqueue_front(n);
                     }
                     else if(opt==3)
                                         dequeue_rear();
                     else if(opt==4)
                                         dequeue_front();
                     else if(opt==5)
                                   display();
                     else
```

```
                    exit(0);
    }
            return 0;
    }
```

Searching and sorting

Searching:

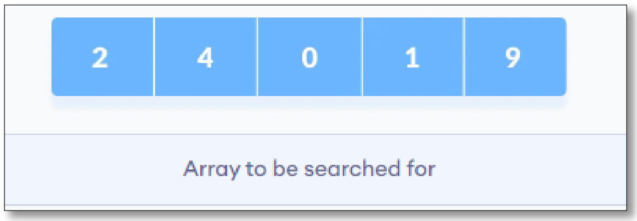
- Searching is the process of finding some particular element in the list. If the element is present in the list, then the process is called successful, and the process returns the location of that element; otherwise, the search is called unsuccessful.
- Two popular search methods are Linear Search and Binary Search.

1. **Linear Search:**

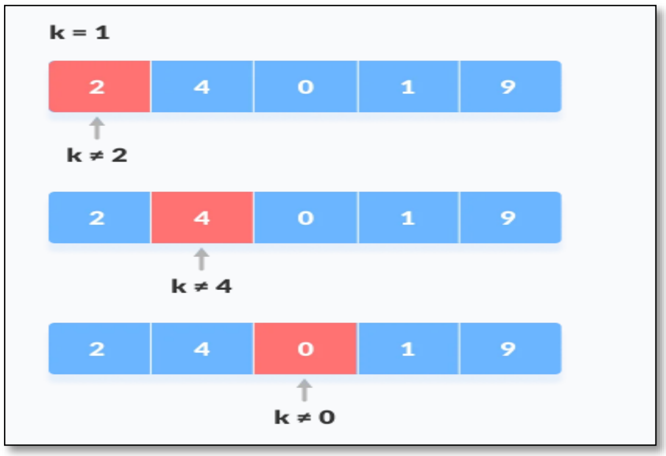
- Linear search is also called as **sequential search algorithm.**
- Linear search is a sequential searching algorithm where we start from one end and check every element of the list until the desired element is found. It is the simplest searching algorithm.
- It is widely used to search an element from the unordered list, i.e., the list in which items are not sorted. The worst-case time complexity of linear search is **O(n).**

How Linear Search Works?

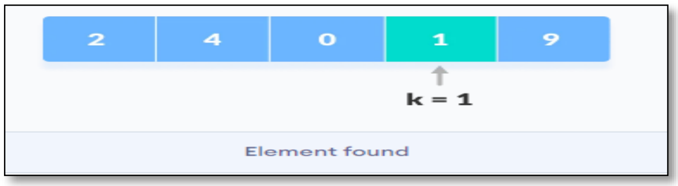
- The following steps are followed to search for an element k = 1 in the list below.



1. Start from the first element, compare *k* with each array element.



2. If x == k, element found.



Program for linear search:

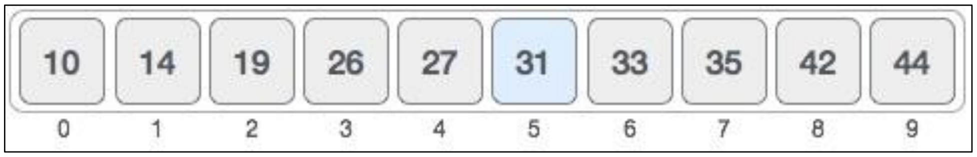
```
#include<iostream>
using namespace std;
int main()
{
        int a[10],n,i,data,c=0;
        cout<<"enter no of elements:\n";
         cin>>n;

        cout<<"enter elements:\n";
        for(i=0;i<n;i++)
        {
                cin>>a[i];
        }
        cout<<"Enter element to search:";
        cin>>data;
        for(i=0;i<n;i++)
        {
                if(data==a[i])
                {
                        c++;
                        cout<<"Data found";
                        break;
                }
        }
        if(c==0)
        cout<<"data not found";
}
```

2. **Binary Search:**

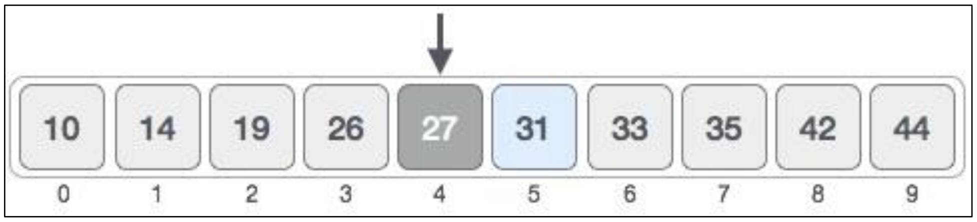
- Binary Search is a searching algorithm for finding an element's position in a sorted array.
- In this approach, the element is always searched in the middle of a portion of an array.
- Binary search can be implemented only on a sorted list of items.

How Binary Search Works?

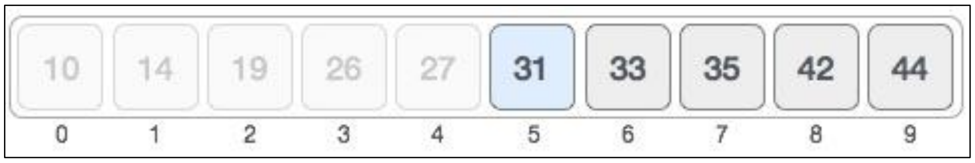
For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with a pictorial example. The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.



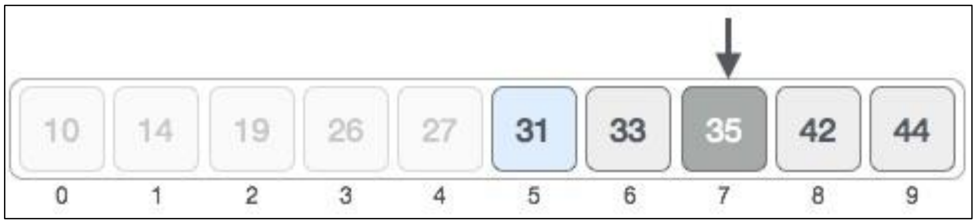
First, we shall determine half of the array by using this formula –
mid = low + (high - low) / 2
Here it is, 0 + (9 - 0) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.



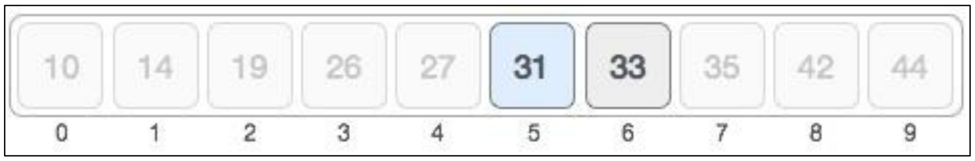
Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.



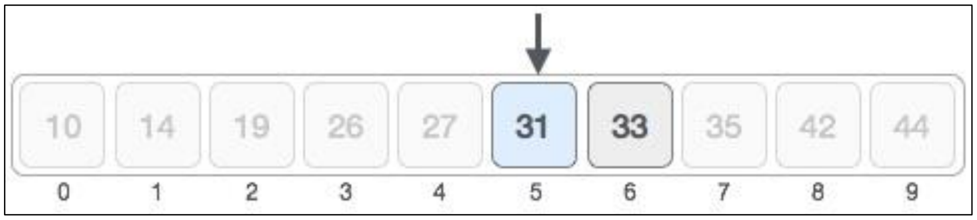
We change our low to mid + 1 and find the new mid value again.
low = mid + 1
mid = low + (high - low) / 2
Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.



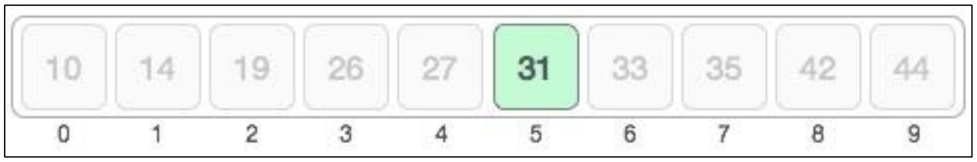
The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.



Hence, we calculate the mid again. This time it is 5.



We compare the value stored at location 5 with our target value. We find that it is a match.



- We conclude that the target value 31 is stored at location 5.
- Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

Program:

```
#include<iostream>
using namespace std;
int main()
{
        int n,search;
        cout<<"enter no of elements:";
        cin>>n;

        int a[n],l=0,r=n-1,m;

        cout<<"Enter elements in sorted order:";
        for(int i=0;i<n;i++)
        {
                cin>>a[i];
        }
        cout<<"enter data to search:";
        cin>>search;

        while(l<=r)
        {
                if(search==a[m])
                {
                        cout<<"data found:";break;
                }
                else if(search < a[m])
                {
                        r=m-1;
                }
                else
                {
                  l=m+1;
                }
                m=(l+r)/2;
        }
        if(l>r)
        {
                cout<<"data not found:";
        }
}
```

Sorting:

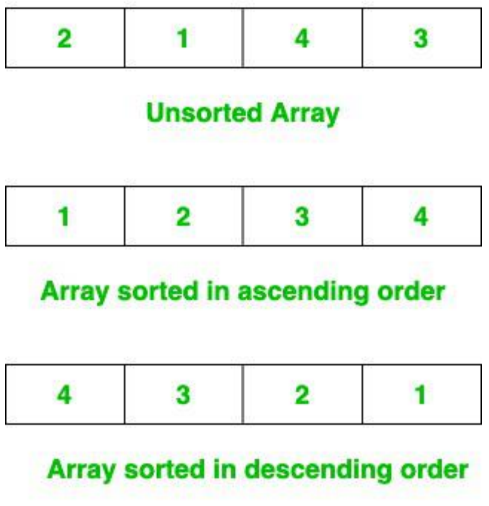
Sorting is the process of arranging the elements of an array so that they can be placed either in ascending or descending order.

Consider an array;

int A[10] = { 5, 4, 10, 2, 30, 45, 34, 14, 18, 9)

The Array sorted in ascending order will be given as;

A[] = { 2, 4, 5, 9, 10, 14, 18, 30, 34, 45 }


2 1 4 3
Unsorted Array
1 2 3 4
Array sorted in ascending order
4 3 2 1
Array sorted in descending order

Sorting algorithms:-

1. Bubble sort
2. Selection Sort
3. Insertion Sort
4. Merge Sort
5. Quick sort

1. Bubble sort:

This is the simplest and easiest sorting technique. In this technique, the two successive items A[i] and A[i+ 1] are exchanged whenever A[i] > = A[i+ 1]. For example, consider the elements shown below :
40, 50, 30, 20, 10

The elements can be sorted as shown in the following figure :

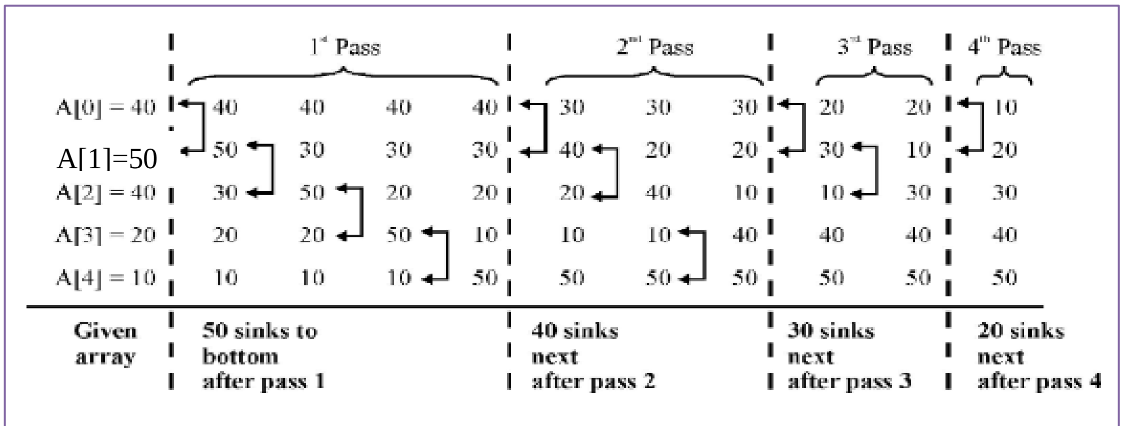

1st Pass
2nd Pass
3rd Pass
4th Pass
A[0] = 40
A[1]=50
A[2] = 40
A[3] = 20
A[4] = 10
Given array
50 sinks to bottom after pass 1
40 sinks next after pass 2
30 sinks next after pass 3
20 sinks next after pass 4

Fig. 1 : Element sorted by Bubble Sort

Note: No of element is 5, n=5.

- in first pass, when i=0,k=0 & n=5 , then total no of comparison was 4. i.e. **n-i-1**
- In second pass, when i=1,k=0 & n=5,then total no of comparison was 3 i.e. **n-i-1**
- In third pass, when i=2,k=0 & n=5,then total no of comparison was 2 i.e. **n-i-1**
- In fourth pass, when i=3,k=0 & n=5,then total no of comparison was 1 i.e. **n-i-1**

Program:

```
#include<stdio.h>
int main()
{
        int a[20],n, i, temp;
        printf("enter no of elements:");
        scanf("%d",&n);

        printf("Enter Elements:");
        for(int i=0;i<n;i++)
        scanf("%d",&a[i]);

        printf("sorted elements are\n");
        for(int j=0;j<n-1;j++)
        {
                for(int k=0;k<n-i-1;k++)
                {
                        if(a[k]>a[k+1])
                        {
                                temp=a[k];
                                a[k]=a[k+1];
                                a[k+1]=temp;
                        }
                }
        }
        printf("Elements are\n");
        for(int i=0;i<n;i++)
        {
                printf("%d ",a[i]);
        }
}
```

Advantages of bubble sort

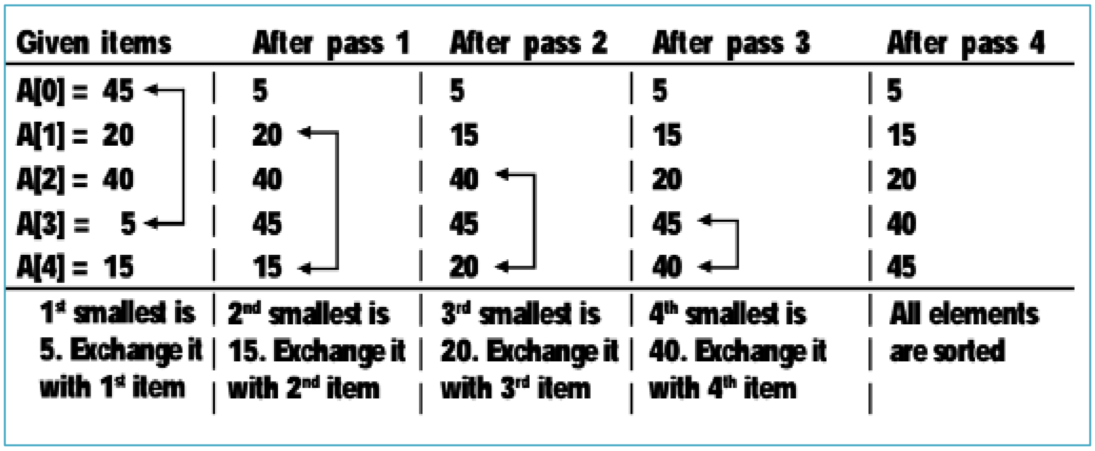
◆ Very simple and easy to program.
◆ Straight forward approach.

Disadvantages of bubble sort

◆ It runs slowly and hence, it is not efficient. More efficient sorting techniques are present.
◆ Even if the elements are sorted, n-1 passes are required to sort.

2. Selection Sort:

- As the name indicates, we first find the smallest item in the list and we exchange it with the first item.
- Obtain the second smallest in the list and exchange it with the second element and so on.
- Finally, all the items will be arranged in ascending order. Since, the next least item is selected and exchanged appropriately so that elements are finally sorted, this technique is called Selection sort.



Given items	After pass 1	After pass 2	After pass 3	After pass 4
A[0] = 45	5	5	5	5
A[1] = 20	20	15	15	15
A[2] = 40	40	40	20	20
A[3] = 5	45	45	45	40
A[4] = 15	15	20	40	45
1st smallest is 5. Exchange it with 1st item	2nd smallest is 15. Exchange it with 2nd item	3rd smallest is 20. Exchange it with 3rd item	4th smallest is 40. Exchange it with 4th item	All elements are sorted

<u>Program:</u>

```
#include<stdio.h>
int main()
{
        int a[20],n, i, temp,min;
        printf("enter no of elements:");
        scanf("%d",&n);

        printf("Enter Elements:");
        for(int i=0;i<n;i++)
        scanf("%d",&a[i]);

        for(i=0;i<=n-1;i++)
        {
                min=i;
                for(int j=i+1;j<=n-1;j++)
                {
                        if(a[min]>a[j])
                        {
                          min=j; //min will hold index no smallest element
                        }
                }
                        //swapping minimum value with ith position.
                        temp=a[i];
                        a[i]=a[min];
                        a[min]=temp;
        }
        printf("Elements are\n");
        for(int i=0;i<n;i++)
        {
                printf("%d ",a[i]);
        }

}
```

3. Insertion sort :

65	**23**	**6**	**12**	**8**	**26**
0	**1**	**2**	**3**	**4**	**5**

- **In insertion sort, we get 2 list from the array, sorted and unsorted list.**
- **1st element is always considered as sorted list and remaining all the element considered as unsorted.**

65	23	65	12	8	26
0	1	2	3	4	5

Sorted list

Unsorted list

- Now we will store first unsorted element 23 in a variable temp and temp will be compared with sorted list data.
- 23 will be compared with 65, as 23 is less than 65 so 65 will be moved one position ahead i.e. at 1st index and after 65 no element is remained for comparison so 23 will be at 0th index.

23	**65**	**6**	**12**	**8**	**26**
0	**1**	**2**	**3**	**4**	**5**

Sorted list

Unsorted list

- Next unsorted element is 6 store it in temp, 6 will be compared with 65, as 6 is less than 65, so 65 Will be shift one position ahead at 2nd index, then 6 will be compared with 23, 6 is also less than 23,so 23 will be shift one position ahead at 1st index, after 23 no element remaining for comparison so 6 will be set 0th index**.**

6	**23**	**65**	**12**	**8**	**26**
0	**1**	**2**	**3**	**4**	**5**

Sorted list

Unsorted list

Comparison will take place until we reach to 0th index and temp value (unsorted data) is less than sorted data.

- Next unsorted element is 12 store it in temp, 12 will be compared with 65, as 12 is less than 65, so 65 Will be shift one position ahead at 3rd index, then 12 will be compared with 23, 12 is also less than 23,so 23 will be shift one position ahead at 2nd index, then 12 will be compared with 6, but 12 is not less than 6, so stop comparison and no swapping will take place. 12 will be at 1st index.

6	**12**	**23**	**65**	**8**	**26**
0	**1**	**2**	**3**	**4**	**5**

$\underbrace{0 \quad 1 \quad 2 \quad 3}_{\text{Sorted list}} \quad \underbrace{4 \quad 5}_{\text{Unsorted list}}$

- Similarly 8 and 26 will be compared with sorted data.

6	**8**	**12**	**23**	**26**	**65**
0	**1**	**2**	**3**	**4**	**5**

Sorted list

Algorithm:

The simple steps of achieving the insertion sort are listed as follows -
Step 1 - If the element is the first element, assume that it is already sorted.
Step2 - Pick the next element, and store it separately in a **key.**
Step3 - Now, compare the **key** with all elements in the sorted array.
Step 4 - If the element in the sorted array is smaller than the current element, then move to the next element. Else, shift greater elements in the array towards the right.
Step 5 - Insert the value.
Step 6 - Repeat until the array is sorted.

Program:
```
#include<iostream>
using namespace std;
int main()
{
        int a[10],n;
        int temp,j;
        cout<<"enter no of elements:\n";
        cin>>n;
        cout<<"enter data\n";
        for(int i=0;i<n;i++)
        {
                cin>>a[i];
        }

        for(int i=1;i<n;i++)
        {

                temp=a[i];
                j=i-1;
                while(j>=0 && temp<=a[j])
                {
                        a[j+1]=a[j];
                        j--;
```

```
                    }
                    a[j+1]=temp;
 }
        cout<<"\nsorted data\n";
        for(int i=0;i<n;i++)
        {
        cout<<a[i]<<" ";
        }

        }
```

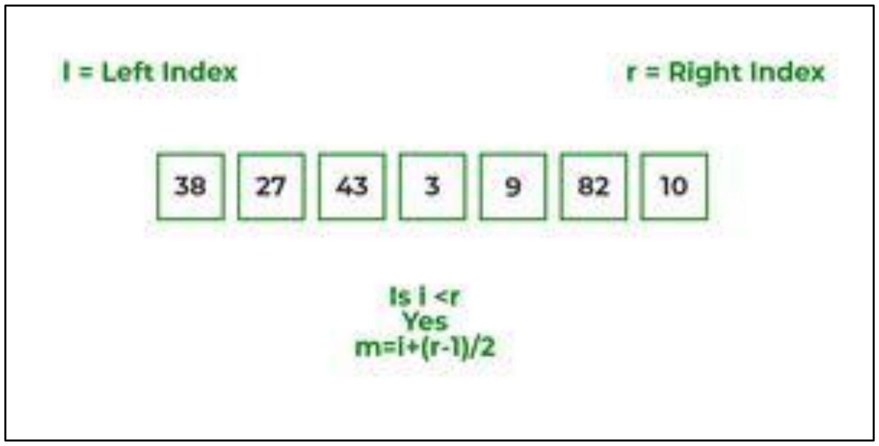
4. **<u>Merge Sort:</u>**

- Merge sort is the sorting technique that follows the divide and conquer approach.
- It divides the given list into two equal halves, calls itself for the two halves and then merges the two sorted halves. We have to define the **merge()** function to perform the merging.
- The sub-lists are divided again and again into halves until the list cannot be divided further. Then we combine the pair of one element lists into two-element lists, sorting them in the process. The sorted two-element pairs is merged into the four-element lists, and so on until we get the sorted list.

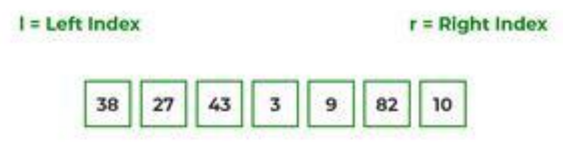
<u>Working of Merge sort Algorithm:</u>

To know the functioning of merge sort, lets consider an array arr[] = {38, 27, 43, 3, 9, 82, 10}

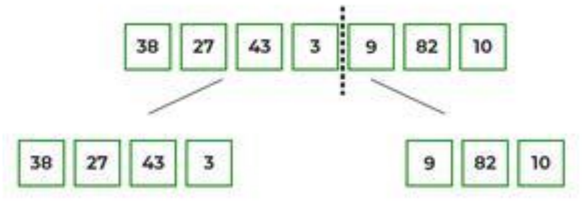
- At first, check if the left index of array is less than the right index, if yes then calculate its mid-point



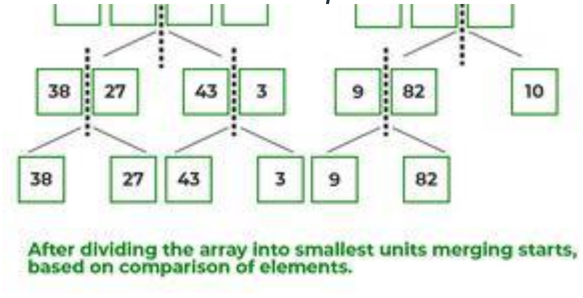
- *Now, as we already know that merge sort first divides the whole array iteratively into equal halves, unless the atomic values are achieved.*
- *Here, we see that an array of 7 items is divided into two arrays of size 4 and 3 respectively.*



- *Now, again find that is left index is less than the right index for both arrays, if found yes, then again calculate mid points for both the arrays.*

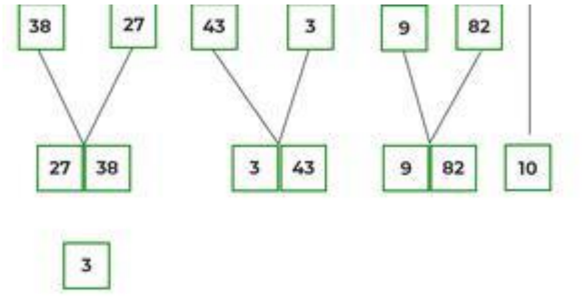


- *Now, further divide these two arrays into further halves, until the atomic units of the array is reached and further division is not possible.*

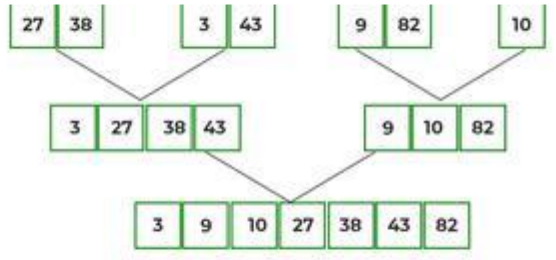


After dividing the array into smallest units merging starts, based on comparison of elements.

- *After dividing the array into smallest units, start merging the elements again based on comparison of size of elements*
- *Firstly, compare the element for each list and then combine them into another list in a sorted manner.*

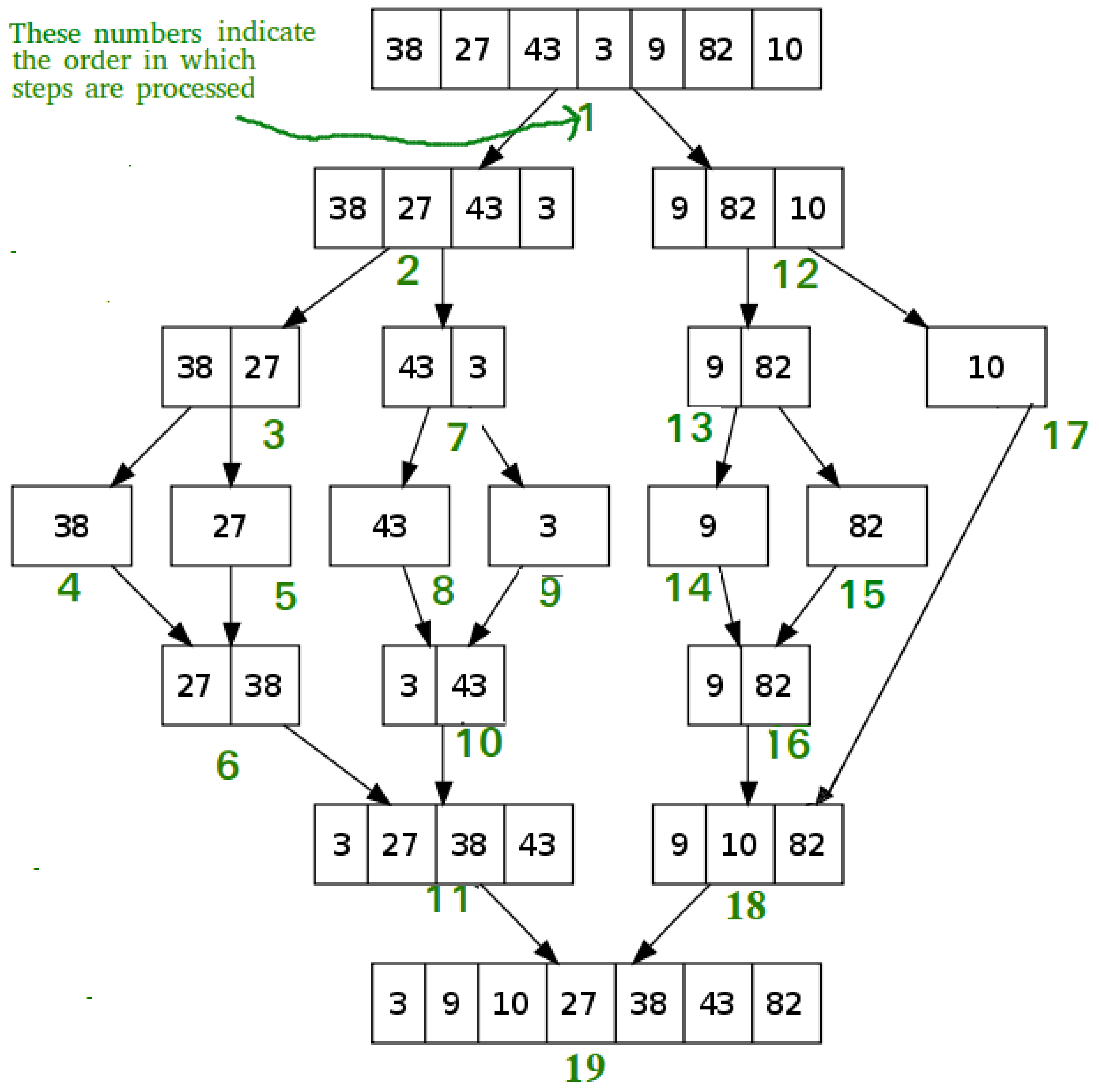


- *After the final merging, the list looks like this:*



The following diagram shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}.

If we take a closer look at the diagram, we can see that the array is recursively divided into two halves till the size becomes 1. Once the size becomes 1, the merge processes come into action and start merging arrays back till the complete array is merged.



Algorithm:

```
step 1: start
step 2: declare array and left, right, mid variable
step 3: perform merge function.
    if left > right
        return
    mid= (left+right)/2
    mergesort(array, left, mid)
    mergesort(array, mid+1, right)
    merge(array, left, mid, right)
step 4: Stop
```

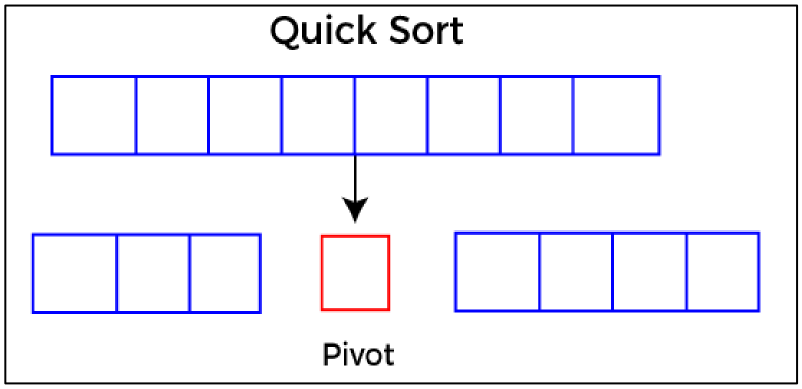
5. Quick sort:

Quick sort is based on divide and conquer technique.

Divide: In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.

Conquer: Recursively, sort two subarrays with Quicksort.

Quicksort picks an element as pivot, and then it partitions the given array around the picked pivot element. In quick sort, a large array is divided into two arrays in which one holds values that are smaller than the specified value (Pivot), and another array holds the values that are greater than the pivot. After that, left and right sub-arrays are also partitioned using the same approach. It will continue until the single element remains in the sub-array.



Working:

- Set lb=0 ,ub=last_index and pivot to 0
- Set start=lb,end=ub
- If lb < ub then i.e. if we have more than one element
- Check if start <=pivot then do start++
- If end>pivot then end - -

- If start < end then swap start with end
- Else swap end with pivot and at this point pivot will be placed at its correct position and we will get 2 arrays left and right
- Now again do the partition on left array as well right array.

Program:

```
#include<iostream>
using namespace std;
void Quick_Sort(int a[10], int lb,int ub)
{
    int start,end,pivot,temp;
    if(lb<ub)
    {
    start=lb;
    end=ub;
    pivot=lb;

    while(start<end)
    {
            while(a[start]<=a[pivot])
    {
            start++;
    }
    while(a[end]>a[pivot])
    {
            end--;
    }
    if(start<end)
    {
            temp=a[start];
            a[start]=a[end];
            a[end]=temp;
    }
    else
    {
            temp=a[pivot];
            a[pivot]=a[end];
            a[end]=temp;
            Quick_Sort(a,lb,end-1);
            Quick_Sort(a,end+1,ub);
    }
  }

    }
}
int main()
{
    int a[20], n;
    cout<<"enter no of elements:";
```

```
	cin>>n;
	cout<<"Enter elements:";
	for(int i=0;i<n;i++)
	{
		cin>>a[i];
	}
	Quick_Sort(a,0,n-1);

	cout<<"Elements are\n:";
	for(int i=0;i<n;i++)
	{
		cout<<a[i]<<" ";
	}

}
```

****