UNIT-3
PART – 1
INTRODUCTION TO DATA STRUCTURE

- Introduction
- Primitive and simple structures
- Linear and nonlinear structures file organization.

Some Important Terms

1. DATA

Information which is input to a computer system and is then processed by mathematical and logical operation. So that it can ultimately be output in a sensible form. It usually has numbers facts letter or system that refer to or describe an object idea, condition, situation relationship or other type of information.

2. DATA-TYPE

Data type is the set of permitted data values and certain operation on data.

3. DATA-STRUCTURE

Data structure is the possible ways of organizing data items that defines how the data items are stored in memory and relationship with each other. Data structure is the possible ways that defines the relationship between data items.

4. CELLS

Cells is the memory area that used to store elementary data items, it can be refered as a single bit, byte, group of bytes.

5. FIELDS

Field is a smallest piece of information that can be reference by a programming language.

TYPES OF DATA STRUCTURE

The collection of organized data is known as data structure.

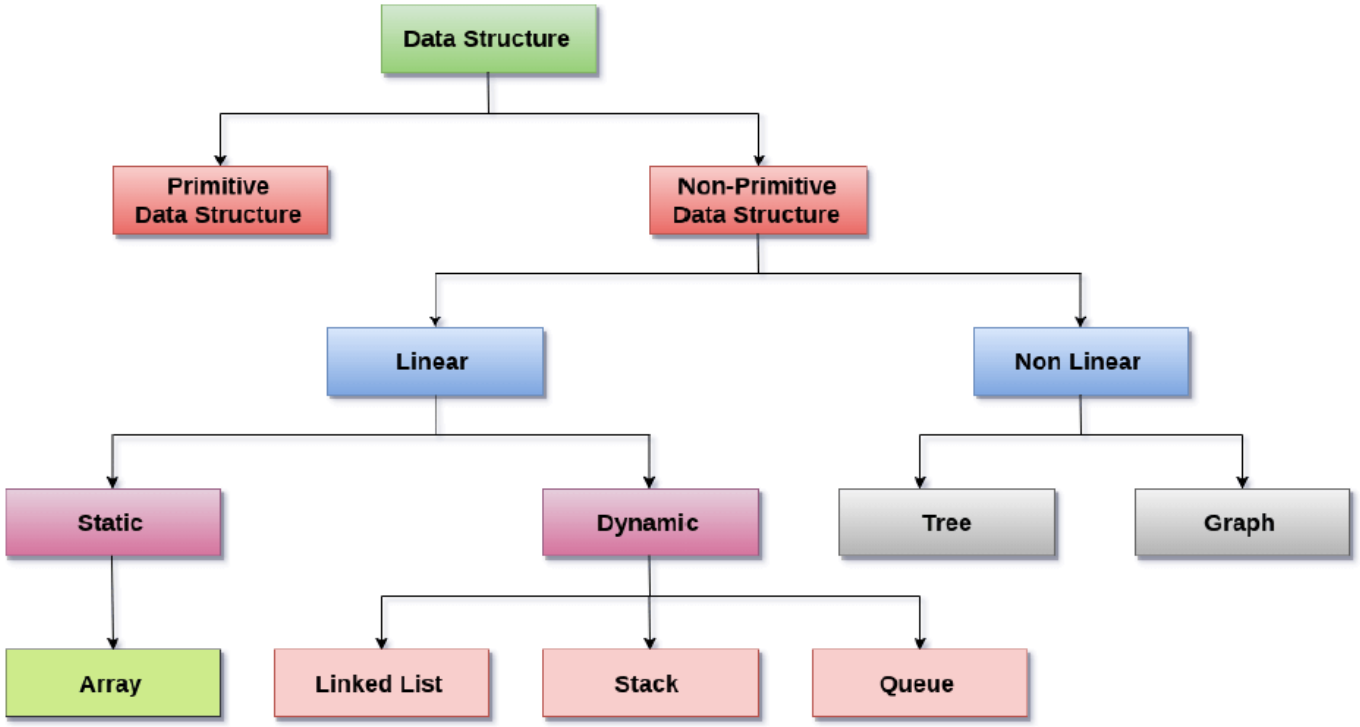
There are main two types of data structure:

1) Primitive data structure

2) Non-primitive data structure

Primitive Data structure

The primitive data structures are primitive data types. The int, char, float,double, and pointer are the primitive data struct ures that can hold a single value.



Non-Primitive data structure:-

- Non-primitive data structure means the data structure constructed by using primitive data structure.
- Non-primitive data structure is also known as composite types.
- Non-primitive data structure is classified into two parts as shown above:

a) Linear data structure:-

- In this type of data structure, the elements are arranged in sequence like an array.

1) Array:

- An array is the collection of structured set that holds fix number of data elements.
- An array is set of homogeneous elements (having same data type).

There are mainly 2 types of array:

- One-dimensional
- Two-dimensional

2) STACK:

- Stack is one type of data structure where information is based on LIFO (Last In First Out).
- All the insertions and deletions take place at only one end which is known as Top Of Stack(TOS).

There are two types of stack:

- Static stack (using array)
- Dynamic stack (using structure or linked list).

3) Queue:

Queue is one type of data structure where information is based on FIFO(First In First Out).

All the insertions and deletion takes place at only end known as front end.

There are two types of queue:

- Static Queue (using array)
- Dynamic Queue (using structure or linked list).

4) Linked List:-

Linked list is defined as the collection of nodes and each node contain two parts:

- Information part
- Pointer to next node.

Information part contains the data and it may consists of one or more fields.
Pointer to next node contains the location where next information is stored. \

Node

Information part Pointer to next node

There are two types of linked list:

- Singly linked list
- Doubly linked list

5) File and structure:-

File is the collection of data that is available to program whenever needed.
There are various operations that can be performed on file.

- Read
- Write
- Append
- Copy

- Structure is the collection of data elements that may or may not have same data type (non-homogeneous data type).

B) Non-linear data structure

In this type of data structure, data elements are not arranged in the sequence

Tree:

It is the most important non-linear data structure which stores data as branches.

- There are different types of trees like binary tree etc.

Graph:

It is also non-linear data structure which contains two main pair:- Vertices and edges.
Depending on nature of representation, there are different types of graph.

PART – 2
Elementary Data Structure

Stack

o Definition

o Operations on stack

o Implementation of stacks using arrays

- **Function to insert an element into the stack**
- **Function to delete an element from the stack**
- **Function to display the items**

o Recursion and stacks

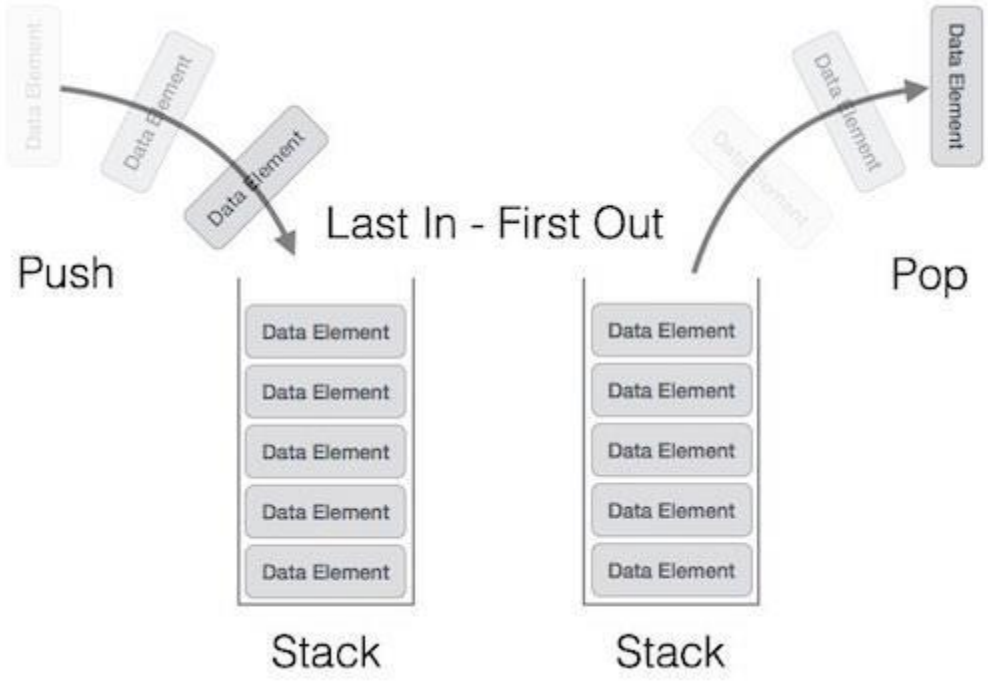
o Evaluation of expressions using stacks

- **Postfix expressions**
- **Prefix expression**

STACK

What is stack?

- Stack is a linear data structure in which insertions and deletions of an element are done at one end which is known as TOS (Top Of Stack).
- Stack supports LIFO(Last In First Out)



Application of Stacks

- Stack is used in the mobile. Message sent by one user and another message sent by same user after some time, in that case the message that was sent last by the user arrives in the inbox first
- Consider a stack of plates placed on counter in a restaurant, During dinner time, plates are taken from the top of place(stack) and waiter puts the washed plates on the top of position which is known as TOS
- The most important application of stack is recursion .
- It is also used in memory management and in operating systems

Types of stack/implementation of stack

- Static Stack/Stack(array)
- Dynamic Stack(linked list)

Implementation of stacks using arrays

1) Function to insert an element into the stack

a. Push operation

- To insert an element on the stack we are using push operation
- This operation inserts the element only on the TOS

Algorithm (we have to define size first)

Step-1:First we have to check for stack overflow
If(tos>=size)
Stack is full

Step-2: Tos=tos+1(increment the pointer value by 1)

Step-3: Stack[tos]=ele; where ele is an element that is to be inserted

Step-4: End

2) Function to delete an element from the stack

b. Pop operation

- To delete an element from the stack we are using pop operation
- This operation removes or deletes the top most element from stack

Algorithm

Step-1: First we have to check for stack underflow
If(tos<0) Stack is empty

Step-2: tos=tos-1

Step-3: Print stack[tos]

Step-4: End

3) Function to display the items

d. Display operation

Step-1: If(tos<0)—stack is empty

Step-2: else
for(i=tos;i>=0;i--)
print element that is stack[i]

Step-3: End

c. Peep operation

- If we want to access some information stored at some location in stack then peep operation is required.
- The index value is subtracted from tos

Algorithm

Step-1: If tos<0 stack is empty

Step-2: Input the position of the element that you want to read

Step-3: If(element<0 || element>tos+1)
Out of range

Step-4: Else Print peeped element
Stack[tos-pos+1]

Step-5: End

e. Update operation

Step-1: If tos<0—stack is empty

Step-2: Input the position of the element that you want to change

Step-3: if(pos<0 || pos>tos+1)
out of range

Step-4:else
Enter new data and print the new data [tos-pos+1]

Step-5: End

❖ **Program of static Stake**

```
#include<stdio.h>
#include<conio.h>
#define size 100
int tos=-1;
int stack[size];
void push(int);
void display();
void peep();
```

```
void pop();
void update();
void main()
{
clrscr();
push(10);
push(11);
push(12);
display();
peep();
update();
display();
pop();
display();
getch();
}
void push(int ele)
{
if(tos>=size)
printf("\nStack is full");
else
tos++;
stack[tos]=ele;
}
void display()
{
int i;
if(tos<0)
printf("\nstack is empty");
else
{
for(i=tos;i>=0;i--)
printf("\nElement at position %d is %d",i,stack[i]);
}
}
void peep()
{
int pos;
if(tos<0)
printf("\nstack is empty");
else
printf("\nEnter value of pos:");
scanf("%d",&pos);
if(pos<0 || pos>tos+1)
```

```
printf("\nout of range");
else
printf("\nPeeped element is %d",stack[tos-pos+1]);
}
void pop()
{
if(tos<0)
printf("\nStack is empty");
else
printf("\nelement deleted");
tos--;
}
void update()
{
int pos;
if(tos<0)
printf("\nstack is empty");
else
{
printf("\nEnter postion:");
scanf("%d",&pos);
if(pos<=0 || pos>tos+1)
printf("\nout of range");
else
{
printf("\nenter new value:");
scanf("%d",&stack[tos-pos+1]);
//printf("\nelement=%d",stack[tos-pos+1]);
}
}
}
```

- **Program of Dynamic Stake**

```
#include<stdio.h>
#include<conio.h>
struct list
{
int info;
struct list *next;
};
typedef struct list node;
struct list *new1;
node *tos;
void push(int);
```

```
void pop();
void peep();
void display();
void update();
void main()
{
tos=NULL;
clrscr();
push(10);
push(11);
push(12);
push(13);
push(14);
display();
pop();
display();
peep();
update();
display();
getch();
}
void push(int ele)
{
node *t;
t=(node *)malloc(sizeof(node));
t->info=ele;
t->next=tos;
tos=t;
}
void display()
{
node *temp;
temp=tos;
while(temp!=NULL)
{
printf("\nStack Element is:%d",temp->info);
temp=temp->next;
}
}
void pop()
{
node *temp;
if(tos==NULL)
printf("\nStack is empty");
```

```
else
temp=tos;
tos=temp->next;
free(temp);
}
void peep()
{
int pos,i,j;
node *temp;
printf("\nEnter position:");
scanf("%d",&pos);
for(temp=pos;temp!=NULL;temp=temp->next)
{
i++;
}
if(pos>i)
printf("\nStack is empty");
else
{
temp=tos;
for(j=1;j<pos;j++)
{
temp=temp->next;
}
printf("\nValue at %d is %d",pos,temp->info);
}
}
void update()
{
int pos,i=0,j;
node *temp;
printf("\nEnter position:");
scanf("%d",&pos);
for(temp=pos;temp!=NULL;temp=temp->next)
{
i++;
}
if(pos>i)
printf("\nStack is full");
else
{
new1=(node *)malloc(sizeof(node));
printf("\nenter new value");
scanf("%d",&new1->info);
```

```
temp=tos;
for(j=1;j<pos;j++)
{
temp=temp->next;
}
temp->info=new1->info;
}
        }
```

Recursion and stacks

Recursion

"Recursion" is the technique of solving any problem by calling the same function again and again until some breaking (base) condition where recursion stops and it starts calculating the solution from there on. For eg. calculating factorial of a given number.

What is the base condition in recursion?

In the recursive program, the solution to the base case is provided and the solution to the bigger problem is expressed in terms of smaller problems.

Example or syntax of recursion with factorial process

```
        int fact(int n)
{
        if (n<=1)   // base case
                return 1;
else
        return    n*fact(n-1);
}
```

In the above example, the base case for n < = 1 is defined and the larger value of a number can be solved by converting to a smaller one till the base case is reached.

Evaluation of expressions using stacks

1) Polish Notation:- The process of writing the operators either before or after the operands in known as polish notation. It has 3 categories:

    1. Infix : Operators are written between two operands.
    2. Prefix: operators are written before their operands.
    3. Postfix: Operators are written after their operands.

Example:

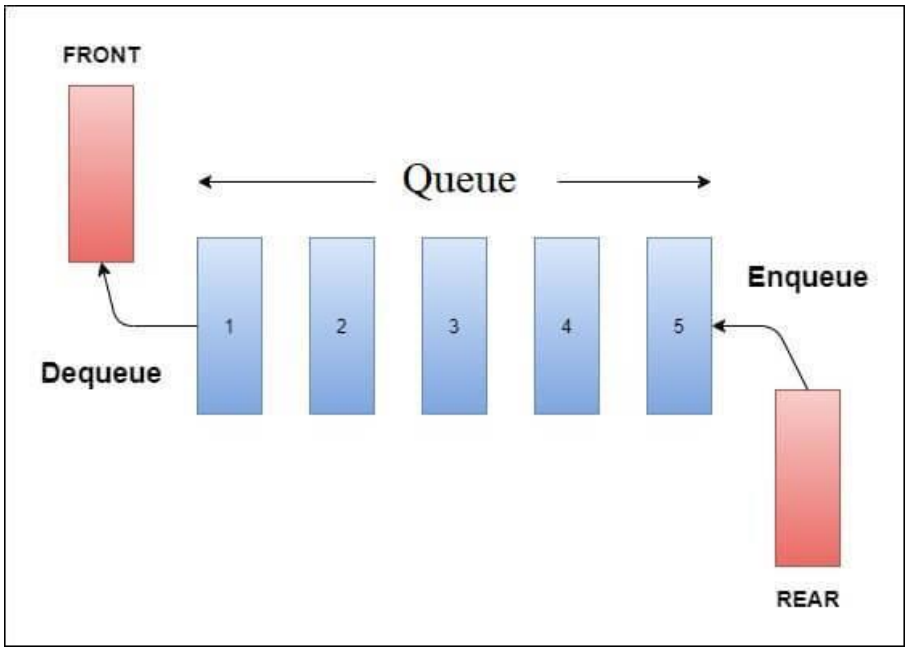
Infix: a+b*c
Prefix: +a*bc
Postfix: abc*+

Expression No	Infix Notation	Prefix Notation	Postfix Notation
1	a + b	+ a b	a b +
2	(a + b) * c	* + a b c	a b + c *
3	a * (b + c)	* a + b c	a b c + *
4	a / b + c / d	+ / a b / c d	a b / c d / +
5	(a + b) * (c + d)	* + a b + c d	a b + c d + *
6	((a + b) * c) - d	- * + a b c d	a b + c * d –

Queue:-

o Introduction
o Array implementation of queues
o Function to insert an element into the queue
o Function to delete an element from the queue

What is queue?



Queue is the linear data structure in which information is based on FIFO(First In First Out) or FCFS(First Come First Server)

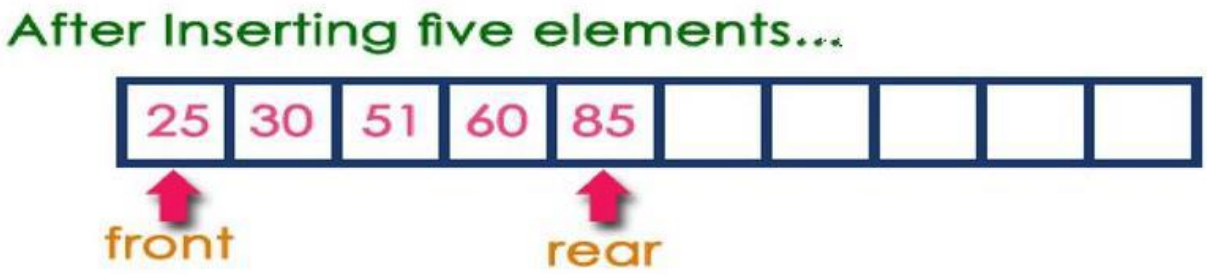
STACK	**QUEUE**
In stack both the insertion and deletion takes place at only one end know as TOS (top of stack)	In queue insertion takes place at rear and deletion takes place at front
In stack we have to keep track of only TOS	In queue we have to keep track of both the ends that is front and rear

In queue insertion of an element is performed at one end known as back or rear end and deletion is performed at another end known as front end.

Insertion operation is known as **enqueue** and deletion operation is known as **dqueue**

Array implementation of queues

Queue after inserting 25, 30, 51, 60 and 85.



Operations on a Queue

The following operations are performed on a queue data structure...

- enQueue(value) - (To insert an element into the queue)
- deQueue() - (To delete an element from the queue)
- display() - (To display the elements of the queue)

- Static Queue
- Dynamic Queue(linked list)
- Circular Queue

Function to insert an element into the queue

a. Insert operation

Algorithm (we have to define size first)

Step-1:If rear>=size
Output "queue overflow

Step-2:rear = rear +1

Step-3:q[rear]=element

Step-4: End

Function to delete an element from the queue

B. Delete operation

Algorithm

Step-1: If front <0
Output "queue underflow"

Step-2: front ++;

Step-3: End

c. Display operation

Algorithm

Step-1: If fornt<0---queue underflow

Step-2:for(i=front;i<=rear;i++)
Printf("%d",queue[i]);

Step-3: End

Program:

```
#include<stdio.h>
#include<conio.h>
#define size 100
int front=-1,rear=-1;
```

```
int queue[size];
void insert(int);
void deleted();
void display();
void main()
{
clrscr();
insert(10);
insert(11);
insert(12);
insert(13);
insert(14);

display();
printf("\n\n");
deleted();
display();
getch();
}
void insert(int ele)
{
if(rear>size)
printf("\nQueue is full");
else
{
rear++;
queue[rear]=ele;
if(rear==0)
front=0;
}
}
void deleted()
{
if(front<0)
printf("\nQueue is empty");
else
{
if(front==rear)
{
front=-1;
rear=-1;
}
else
front++;
}
```

```
}
void display()
{
int i;

if(front<0)
printf("\nQueue empty");
else
{
for(i=front;i<=rear;i++)
{
printf("\nElement at position %d is %d",i,queue[i]);
}
}
```
}**Dynamic queue:**

a. Insert operation

Step-1:Allocate the memory to the node (node *q)

q=(node *)malloc(sizeof(node));

Step-2: Assign data part and next part of node

q->info=ele;

q->next=NULL;

Step-3: If queue is empty then:

if(front==NULL)

front=rear=q

Step-4: If queue is not empty

rear-next=q;

rear=q;

Step-5: End

b. Delete operation

Step-1: Check whether queue is empty or not.

Step-2: If the queue is empty then

If(front==NULL)---queue empty

Step-3: If queue is not empty then

node *q

q=front

front=q->next

Step-4:free(q)

Step-5: End

C. Display operation

Step-1: node *q

q=front

Step-2: Check if queue is empty or not

If(q==NULL)---queue empty

Step-3: If queue is not empty then

While(q!=NULL)

Printf("%d",q->info);

q=q->next

Step-4:END

Program

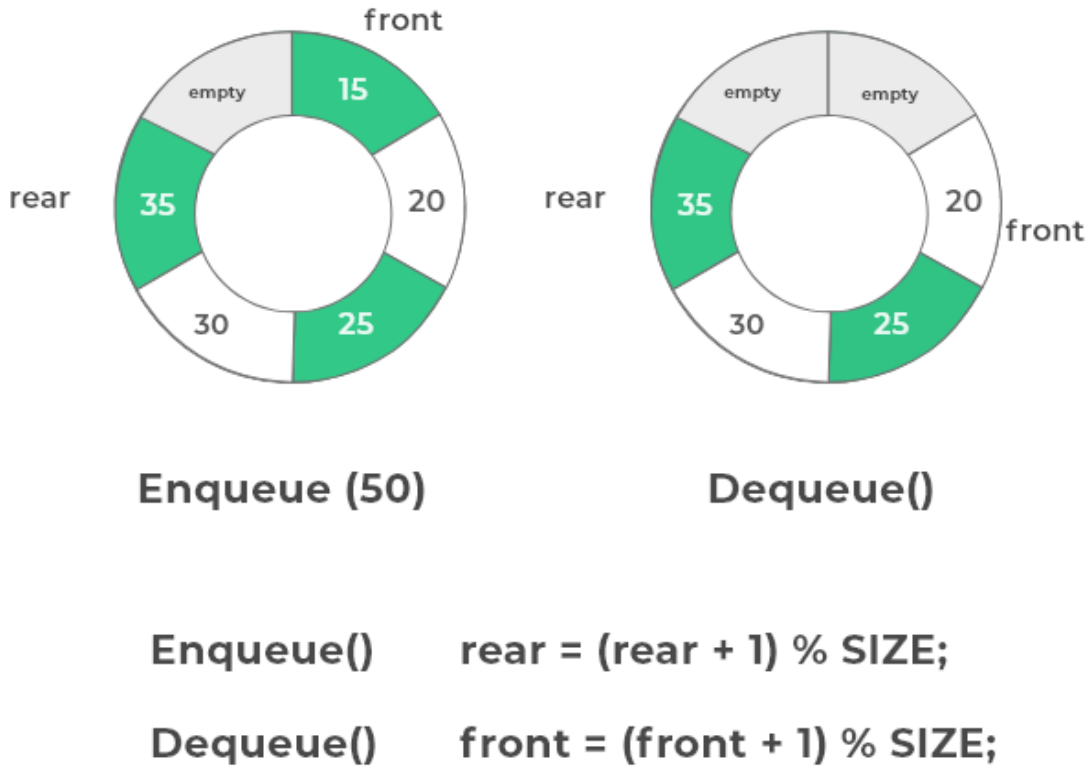
```
#include<stdio.h>
#include<conio.h>
struct list
{
int info;
struct list *next;
};
typedef struct list node;
node *front,*rear;
void insert(int);
void deleted();
void display();
void main()
{
front=rear=NULL;
clrscr();
insert(10);
insert(11);
insert(12);
display();
deleted();
display();
getch();
}
void insert(int ele)
{
node *t;
t=(node *)malloc(sizeof(node));
t->info=ele;
t->next=NULL;
if(front==NULL)
{
front=rear=t;
}
else
```

```
{
rear->next=t;
rear=t;
}
}
void deleted()
{
node *temp;
temp=front;
if(temp==NULL)
printf("\nQueue is empty");
else
front=temp->next;
free(temp);
}
void display()
{
node *temp;
int i=0;
temp=front;
if(temp==NULL)
printf("\nQueue is empty");
else
{
while(temp!=NULL)
{
printf("\nElement is %d",temp->info);
temp=temp->next;
}
}
}
```

Circular queue:

- The problem with simple queue is that once we insert the elements and when we are removing elements using front pointer these elements becomes blank. As the elements become blank we cannot insert any elements in that blank space.
- So to overcome this problem there is technique available known as circular queue. In this technique when rear reaches the queue's size the first element will become the queue's new rear

- **NOTE:** When the queue is full, the first element of queue becomes the rear if and only if front has moved forward otherwise it will be again in a "queue full(overflow)" state.



Enqueue() rear = (rear + 1) % SIZE;

Dequeue() front = (front + 1) % SIZE;

a. Insert operation

Step-1:If front==1 and rear==size (output queue overflow)

Step-2:else if (front==0)

Front=rear=1

q[rear]=element

Step-3:else if (rear==size)

rear=1

q[rear]=element

Step-4:else

rear=rear+1
q[rear]=element

Step-5: End

b. Delete operation

Step-1: If front==0 (output queue is empty)

Step-2: If front==rear
front=rear=0

Step-3: else if (front==size)
front=1

Step-4:else
front=front+1

Step-5: End

C. Display operation

Step-1: If front==0 (output queue is empty)

Step-2:else if (front >rear)
for(i=1;i<=rear;i++)
printf("%d",queue[i])
for(i=front;i<=size;i++)
printf("%d",queue[i])

Step-3: else
for(i=front;i<=rear;i++)
printf("%d",queue[i])

Step-4:END

Program

```
#include<stdio.h>
#include<conio.h>
#define size 5
int queue[size];
int front=0,rear=0;
void insert(int);
void deleted();
void display();
void main()
{
clrscr();
insert(10);
insert(20);
```

```
insert(30);
insert(40);
insert(50);
display();
printf("\n\n");
deleted();
display();
printf("\n\n");
insert(60);
display();
getch();
}
void insert(int ele)
{
if(front==1 && rear==size)
printf("\nQueue is full");
else if(front==0)
{
front=rear=1;
queue[rear]=ele;
}
else if(rear==size)
{
rear=1;
queue[rear]=ele;
}
else
{
rear++;
queue[rear]=ele;
}
}
void deleted()
{
if(front==0)
printf("\nQueue is empty");
else if(front==rear)
front=rear=0;
else if(front==size)
```

```
front=1;
else
front++;
}
void display()
{
int i;
if(front==0)
printf("\nQueue is empty");
else if(front>rear)
{
for(i=1;i<=rear;i++)
{
printf("\nElement at %d is %d",i,queue[i]);
}
for(i=front;i<=size;i++)
{
printf("\nElement at %d is %d",i,queue[i]);
}
}
else
{
for(i=front;i<=rear;i++)
{
printf("\nElement at %d is %d",i,queue[i]);
}
}
}
```

What is a queue?

A queue is a data structure in which whatever comes first will go out first, and it follows the FIFO (First-In-First-Out) policy. Insertion in the queue is done from one end known as the **rear end** or the **tail,** whereas the deletion is done from another end known as the **front end** or the **head** of the queue.

What is a Deque (or double-ended queue)

The deque stands for Double Ended Queue. Deque is a linear data structure where the insertion and deletion operations are performed from both ends. We can say that deque is a generalized version of the queue.

Though the insertion and deletion in a deque can be performed on both ends, it does not follow the FIFO rule. The representation of a deque is given as follows -

Representation of deque

Types of deque

There are two types of deque -

- Input restricted queue
- Output restricted queue

Input restricted Queue

In input restricted queue, insertion operation can be performed at only one end, while deletion can be performed from both ends.

input restricted double ended queue

Output restricted Queue

In output restricted queue, deletion operation can be performed at only one end, while insertion can be performed from both ends.

Output restricted double ended queue

Operations performed on deque

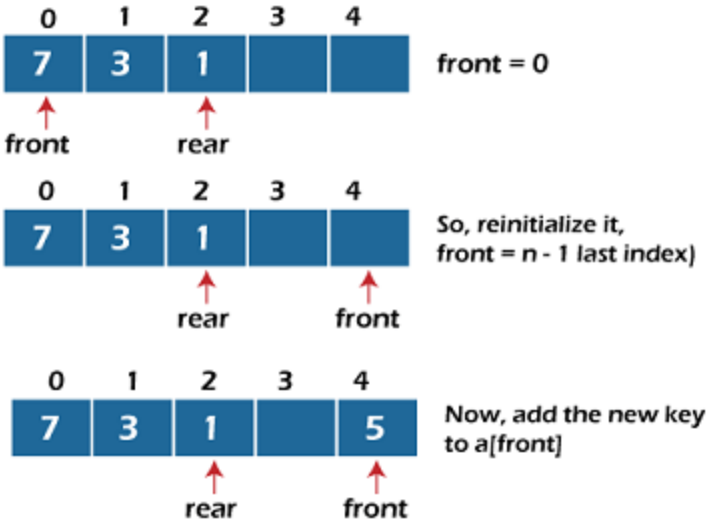
There are the following operations that can be applied on a deque -

- Insertion at front
- Insertion at rear
- Deletion at front
- Deletion at rear

Insertion at the front end

In this operation, the element is inserted from the front end of the queue. Before implementing the operation, we first have to check whether the queue is full or not. If the queue is not full, then the element can be inserted from the front end by using the below conditions -

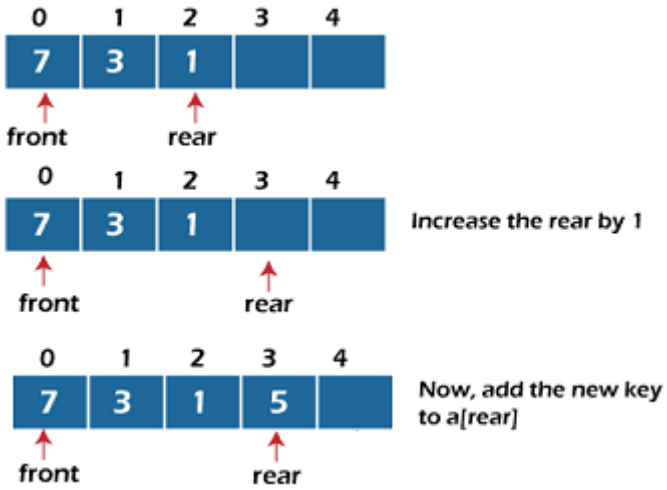
- If the queue is empty, both rear and front are initialized with 0. Now, both will point to the first element.
- Otherwise, check the position of the front if the front is less than 1 (front < 1), then reinitialize it by **front = n - 1**, i.e., the last index of the array.



Insertion at the rear end

In this operation, the element is inserted from the rear end of the queue. Before implementing the operation, we first have to check again whether the queue is full or not. If the queue is not full, then the element can be inserted from the rear end by using the below conditions -

- If the queue is empty, both rear and front are initialized with 0. Now, both will point to the first element.
- Otherwise, increment the rear by 1. If the rear is at last index (or size - 1), then instead of increasing it by 1, we have to make it equal to 0.



Deletion at the front end

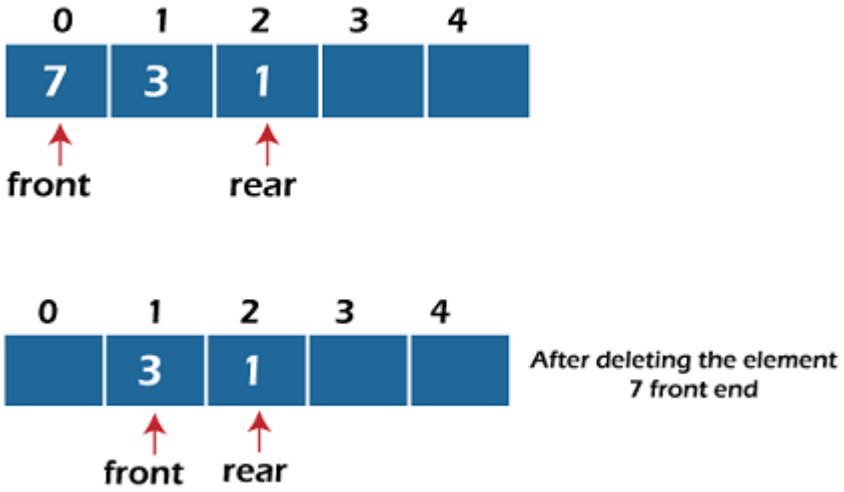
In this operation, the element is deleted from the front end of the queue. Before implementing the operation, we first have to check whether the queue is empty or not.

If the queue is empty, i.e., front = -1, it is the underflow condition, and we cannot perform the deletion. If the queue is not full, then the element can be inserted from the front end by using the below conditions -

If the deque has only one element, set rear = -1 and front = -1.

Else if front is at end (that means front = size - 1), set front = 0.

Else increment the front by 1, (i.e., front = front + 1).



Deletion at the rear end

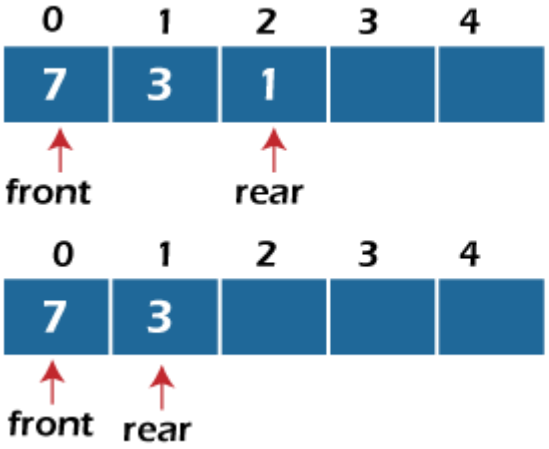
In this operation, the element is deleted from the rear end of the queue. Before implementing the operation, we first have to check whether the queue is empty or not.

If the queue is empty, i.e., front = -1, it is the underflow condition, and we cannot perform the deletion.

If the deque has only one element, set rear = -1 and front = -1.

If rear = 0 (rear is at front), then set rear = n - 1.

Else, decrement the rear by 1 (or, rear = rear -1).



After deleting element 1 from rear end

Applications of deque

- Deque can be used as both stack and queue, as it supports both operations.
- Deque can be used as a palindrome checker means that if we read the string from both ends, the string would be the same.

Implementation of deque

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

// insert_rear function will insert the value from the rear
void insert_rear(int x)
{
    if((f==0 && r==size-1) || (f==r+1))
    {
        printf("Overflow");
    }
    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("\nElements in a deque are: ");

   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 element at 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 element at rear is %d", deque[r]);
```

```
    }

}

// delete_front() function deletes the element from the front
void delete_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;
     }
}

// delete_rear() function deletes the element from the rear
void delete_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()
{
    insert_front(20);
    insert_front(10);
    insert_rear(30);
    insert_rear(50);
    insert_rear(80);
    display();  // Calling the display function to retrieve the values of deque
    getfront();  // Retrieve the value at front-end
    getrear();  // Retrieve the value at rear-end
    delete_front();
    delete_rear();
    display(); // calling display function to retrieve values after deletion
    return 0;
}
```

What is a priority queue?

A priority queue is an abstract data type that behaves similarly to the normal queue except that each element has some priority, i.e., the element with the highest priority would come first in a priority queue. The priority of the elements in a priority queue will determine the order in which elements are removed from the priority queue.

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 values are arranged in ascending order. Now, we will observe how the priority queue will look after performing the following operations:

- **poll():** This function will remove the highest priority element from the priority queue. In the above priority queue, the '1' element has the highest priority, so it will be removed from the priority queue.
- **add(2):** This function will insert '2' element in a priority queue. As 2 is the smallest element among all the numbers so it will obtain the highest priority.
- **poll():** It will remove '2' element from the priority queue as it has the highest priority queue.
- **add(5):** It will insert 5 element after 4 as 5 is larger than 4 and lesser than 8, so it will obtain the third highest priority in a priority queue.

Types of Priority Queue

There are two types of priority queue:

- **Ascending order priority queue:** In ascending order priority queue, a lower priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in an ascending order like 1,2,3,4,5; therefore, the smallest number, i.e., 1 is

given as the highest priority in a priority queue.

- **Descending order priority queue:** In descending order priority queue, a higher priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in descending order like 5, 4, 3, 2, 1; therefore, the largest number, i.e., 5 is given as the highest priority in a priority queue.