

1. Explain pointer to function concept with example.

->A pointer to a function points to the address of the executable code of the function. You can use pointers to call functions and to pass functions as arguments to other functions. You cannot perform pointer arithmetic on pointers to functions.

a pointer is a variable which contains address of another variable or memory location.

Pointer is used to points the address of the value stored anywhere in the computer memory.

This variable can be char, int, array, function.

Function Pointers point to code like normal pointers. In Functions Pointers, function's name can be used to get function's address.

Syntax:

```
int *f(int a);    /* function f returning an int
```

```
int (*g)(int a); /* pointer g to a function returning an int
```

Example:

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

```
// A normal function with an int parameter
```

```
// and void return type
```

```
void fun(int a)
```

```
{
```

```
printf("Value of a is %d\n", a);
```

```
}
```

```
int main()
```

```
{
```

```
// fun_ptr is a pointer to function fun()
```

```
void (*fun_ptr)(int) = &fun;
```

```
void (*fun_ptr)(int);
```

```
    fun_ptr = &fun;
```

```
    // Invoking fun() using fun_ptr
```

```
    (*fun_ptr)(10);
```

```
    return 0;
```

```
}
```

2. Explain the basic stack operations.

->it is a data structure, where elements can be inserted from one end and deleted from same end.

Stacks in Data Structures is a linear type of data structure that follows the LIFO (Last-In-First-Out) principle and allows insertion and deletion operations from one end of the stack data structure.

Basic Operations:

PUSH operation: The PUSH operation is used to insert a new element in the Stack. PUSH operation inserts a new element at the top of the stack. It is important to check overflow condition before push operation when using an array representation of Stack.

POP operation: The POP operation is used to Remove an element from the top of a stack. Pop operation refers to the removal of an element. Again, since we only have access to the element at the top of the stack, there's only one element that we can remove.

PEEK operations: Peek operation allows the user to see the element on the top of the stack. The stack is not modified in any manner in this operation.

isEmpty: The isEmpty operation Checks if stack is empty or not.

To prevent performing operations on an empty stack, the programmer is required to internally maintain the size of the stack which will be updated during push and pop operations accordingly. isEmpty() conventionally returns a boolean value: True if size is 0, else False.

3. Outline the algorithm to destroy the queue..

->Queue is referred to be as First In First Out list.

A queue can be defined as an ordered list which enables insert operations to be performed at one end called REAR and delete operations to be performed at another end called FRONT.

Dequeue- Removing elements from a queue if there are any elements in the queue.

deQueue(): This function deletes an element from the Queue. The deletion in a Queue always takes place from the front end.

Algorithm:

qdelete (queue [maxsize] , item)

This algorithm deletes an element an item at the front of the **queue(maxsize)**

Step1 Repeat steps 2 to 4 until front ≥ 0

Step2 Set item = queue [front]

Step3 If front == rear then

set front = -1

set rear = -1

else

front = front + 1

Step4 Print, No. deleted is, item

Step5 Print, "Queue is empty"

Before Deletion of First Element in the Queue

FRONT=1

REAR=5

	26	78	32	60	45	
0	1	2	3	4	5	6

After Deletion of first element

ITEM=QUEUE[FRONT] => ITEM=26

FRONT=FRONT+1

FRONT=2

REAR=5

		78	32	60	45	
0	1	2	3	4	5	6

4. Outline the algorithm to delete data from the queue and retrieve the data at the front of the queue.

->Algorithm of Deletion:

qdelete (queue [maxsize] , item)

This algorithm deletes an element an item at the front of the queue(maxsize)

Step1 Repeat steps 2 to 4 until front ≥ 0

Step2 Set item = queue [front]

Step3 If front == rear then

set front = -1

set rear = -1

else

front = front + 1

Step4 Print, No. deleted is, item

Step5 Print, "Queue is empty"

procedure dequeue

if queue is empty

return underflow

end if

data = queue[front]

front \leftarrow front + 1

return true

end procedure

Algorithm of Retrieving:

procedure enqueue(data)

if queue is full

return overflow

endif

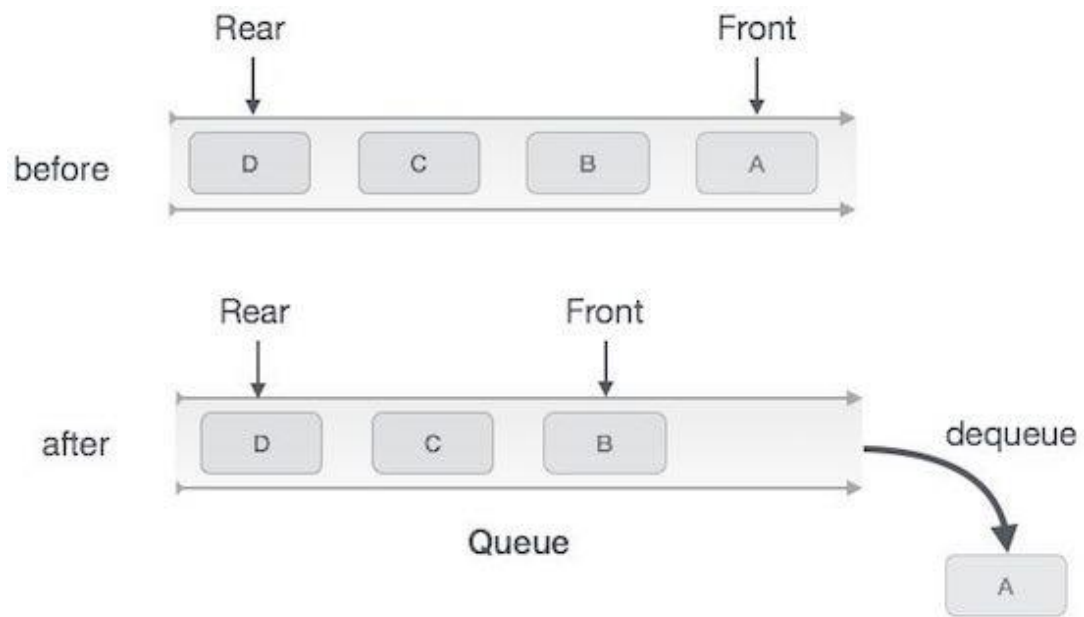
rear \leftarrow rear + 1

queue[rear] \leftarrow data

return true

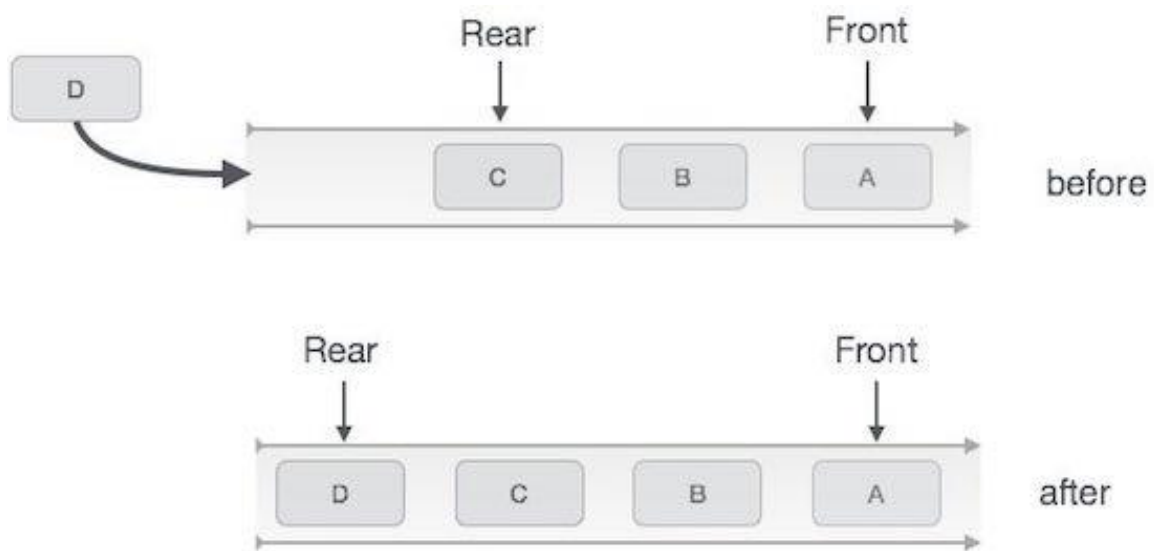
end procedure

DELETION:



Queue Dequeue

RETRIEVE:

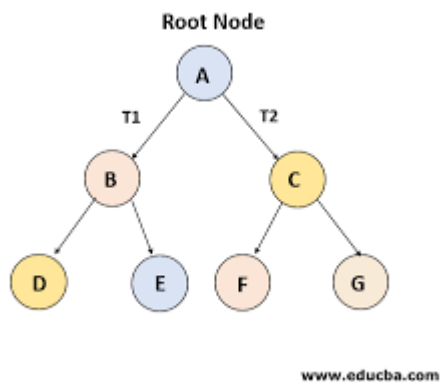


Queue Enqueue

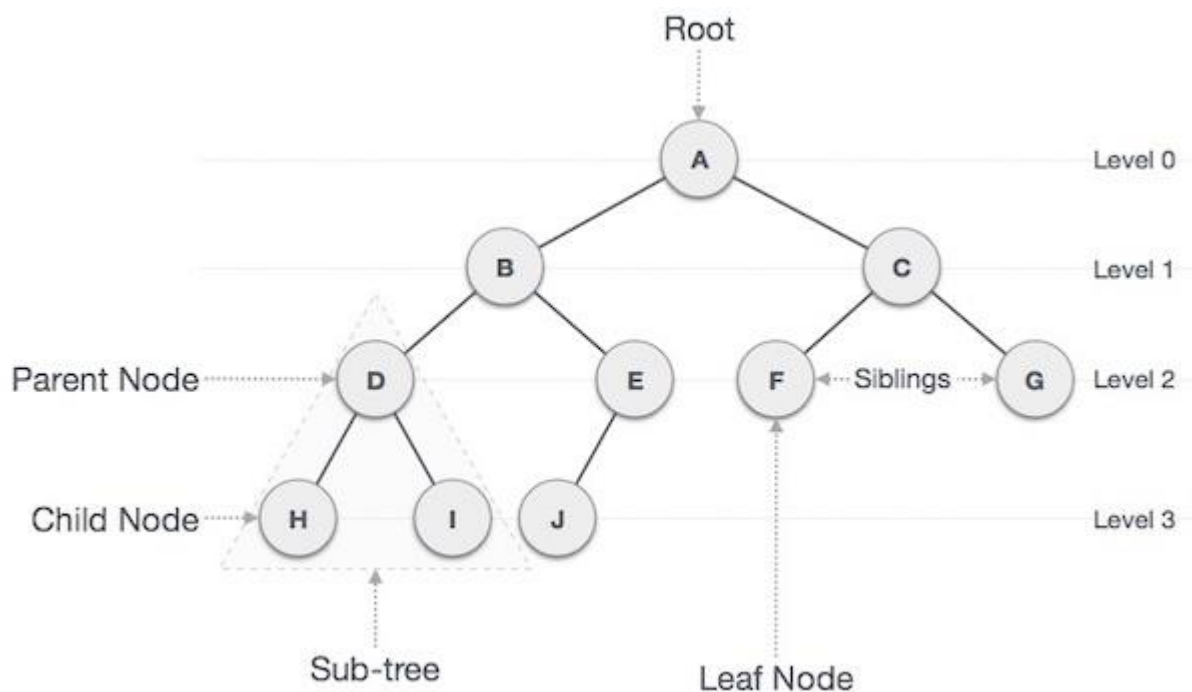
5. What is binary Tree? Define the following terminologies of binary tree

- a) Root.
- b) leaf node.
- c) internal node.
- d) out degree.
- e) in degree.
- f) degree.
- g) Siblings.
- h) Parent node.

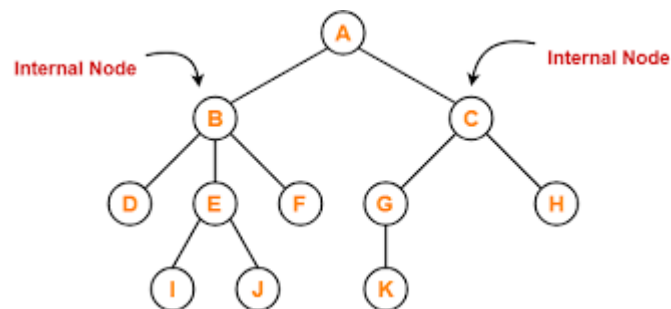
->ROOT: In a tree data structure, the first node or the topmost node is called as root node. Every tree must have only one root node.



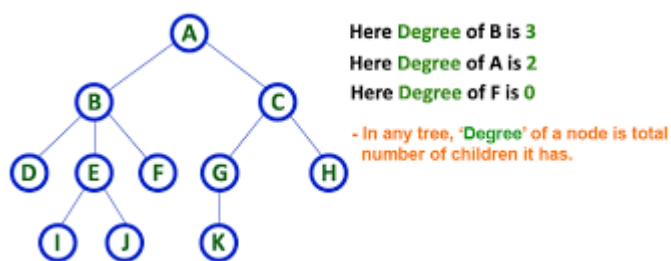
LEAF NODE: it is also known as external or terminal node which it is a tree data structure the node which does not have child is called as leaf node



INTERNAL NODE: it is also known as operator node and in a tree data structure the nodes which has atleast one child is internal node.



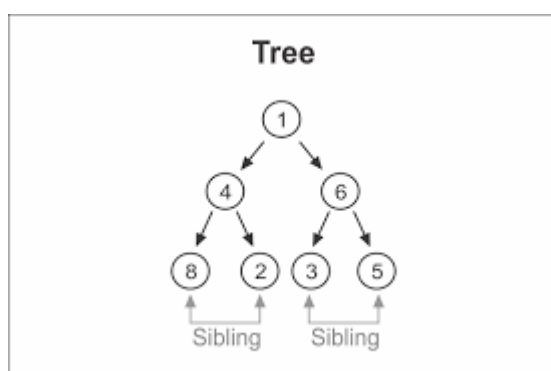
OUT DEGREE: Total number of leaving vertices is known as outdegree, The out-degree of a node is equal to the number of edges with that node as the source.



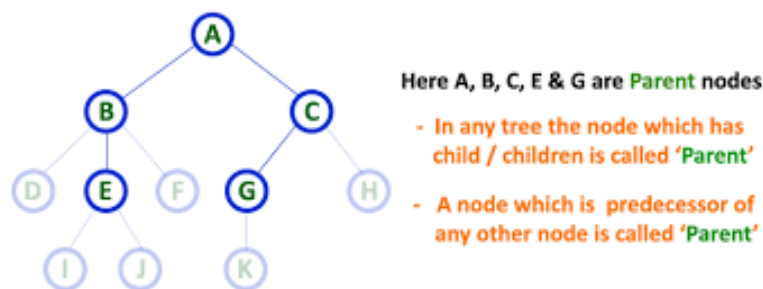
IN DEGREE: Total number of entering vertices is known as indegree.

DEGREE: The number of subtrees of a node is called its degree. For example, node A is of degree three, while node E is of degree two. The maximum degree of all nodes is called the degree of the tree.

SIBLINGS: in a tree data structure, nodes which belongs to same parent are called as siblings.



PARENT NODE: in a tree data structure the node which is a predecessor of any node is a parent node.



6. Compare and contrast linked list and array.

->

ARRAY	LINKED LISTS
1. Arrays are stored in contiguous location.	1. Linked lists are not stored in contiguous location.
2. Fixed in size.	2. Dynamic in size.
3. Memory is allocated at compile time.	3. Memory is allocated at run time.
4. Uses less memory than linked lists.	4. Uses more memory because it stores both data and the address of next node.
5. Elements can be accessed easily.	5. Element accessing requires the traversal of whole linked list.
6. Insertion and deletion operation takes time.	6. Insertion and deletion operation is faster.

7. Explain the dynamic memory management functions in C.

-> Dynamic Memory Allocation in C is a process in which we allocate or deallocate a block of memory during the run-time of a program. Dynamic memory allocation performs 'run time', the functions `calloc()` and `malloc()` supports allocating dynamic memory.

There are four functions `malloc()`, `calloc()`, `realloc()` and `free()` present in `<stdlib.h>` header file that are used for Dynamic Memory Allocation in our system.

`Malloc()`: it is a single block memory & used to allocate exact amount of memory.

`Calloc()`: Arrays are used to allocate memory.

`Realloc()`: It is used to resizing allocated memory.

`Free()`: It is used for de-allocating the allocated memory.

8. Develop a C program to print the elements of array in forward and reverse direction using pointers.

-> Reverse:

```
#include<stdio.h>

#define N 5

int main()
{
    int a[N], i, *ptr;
    printf("Enter %d integer numbers\n", N);
    for(i = 0; i < N; i++)
        scanf("%d", &a[i]);
    ptr = &a[N - 1];
    printf("\nElements of array in reverse order ...\n");
    for(i = 0; i < N; i++)
        printf("%d\n", *ptr--);
    return 0;
}
```

Output 1:

Enter 5 integer numbers

1
2
3
4
5

Elements of array in reverse order ...

5
4
3
2
1

Forward:

```
#include<stdio.h>

#define N 5

int main()
{
    int a[N], i, *ptr;
    printf("Enter %d integer numbers\n", N);
    for(i = 0; i < N; i++)
        scanf("%d", &a[i]);
    ptr = &a[N + 1];
    printf("\nElements of array in forward order ...\n");
    for(i = 0; i < N; i++)
        printf("%d\n", *ptr++);
    return 0;
}
```

Output 1:

Enter 5 integer numbers

1

2

3

4

5

Elements of array in forward order ...

6

7

8

9

10

9. Develop a C program using stack to evaluate the given infix expression.

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

#include<ctype.h>

char stack[100];

int top = -1;

void push(char x)
{
    stack[++top] = x;
}

char pop()
{
    if(top == -1)
        return -1;
    else
        return stack[top--];
}

int priority(char x)
{
    if(x == '(')
        return 0;
    if(x == '+' || x == '-')
        return 1;
    if(x == '*' || x == '/')
        return 2;
    return 0;
}

int main()
{
    char exp[100];
    char *e, x;
    printf("Enter the expression : ");
```

```

scanf("%s",exp);
printf("\n");
e = exp;
while(*e != '\0')
{
    if(isalnum(*e))
        printf("%c ",*e);
    else if(*e == '(')
        push(*e);
    else if(*e == ')')
    {
        while((x = pop()) != '(')
            printf("%c ", x);
    }
    else
    {
        while(priority(stack[top]) >= priority(*e))
            printf("%c ",pop());
        push(*e);
    }
    e++;
}
while(top != -1)
{
    printf("%c ",pop());
}return 0;
}

```

10. Develop a C program to implement all basic operations of circular queue to store integer numbers.

->

```
#include<stdio.h>

# define MAX 5

int cqueue_arr[MAX];

int front = -1;

int rear = -1;

void insert(int item)
{
if((front == 0 && rear == MAX-1) || (front == rear+1))
{
printf("Queue Overflow n");
return;
}
if(front == -1)
{
front = 0;
rear = 0;
}
else
{
if(rear == MAX-1)
rear = 0;
else
rear = rear+1;
}
cqueue_arr[rear] = item ;
}

void deletion()
{
}
```

```

if(front == -1)
{
printf("Queue Underflown");
return ;
}

printf("Element deleted from queue is : %dn",cqueue_arr[front]);
if(front == rear)
{
front = -1;
rear=-1;
}
else
{
if(front == MAX-1)
front = 0;
else
front = front+1;
}
}

void display()
{
int front_pos = front,rear_pos = rear;
if(front == -1)
{
printf("Queue is emptyn");
return;
}
printf("Queue elements :n");
if( front_pos <= rear_pos )
while(front_pos <= rear_pos)
{

```

```

printf("%d ",cqueue_arr[front_pos]);
front_pos++;
}
else
{
while(front_pos <= MAX-1)
{
printf("%d ",cqueue_arr[front_pos])
front_pos++;
}
front_pos = 0;
while(front_pos <= rear_pos)
{
printf("%d ",cqueue_arr[front_pos]);
front_pos++;
}
}
printf("\n");
}

int main()
{
int choice,item;
do
{
printf("1.Insertn");
printf("2.Deleten");
printf("3.Displayn");
printf("4.Quitn");
printf("Enter your choice : ");
scanf("%d",&choice);
switch(choice)

```

```

{
case 1 :

printf("Input the element for insertion in queue : ");

scanf("%d", &item);

insert(item);

break;

case 2 :

deletion();

break;

case 3:

display();

break;

case 4:

break;

default:

printf("Wrong choicen");

}

}while(choice!=4);

return 0;

}

```

OUTPUT:

```

C:\WINDOWS\SYSTEM32\cmd.exe
4.Quit
Enter your choice : 1
Input the element for insertion in queue : 2
1.Insert
2.Delete
3.Display
4.Quit
Enter your choice : 1
Input the element for insertion in queue : 2
1.Insert
2.Delete
3.Display
4.Quit
Enter your choice : 3
Queue elements :
2 2
1.Insert
2.Delete
3.Display
4.Quit
Enter your choice : 2
Element deleted from queue is : 2
1.Insert
2.Delete
3.Display
4.Quit
Enter your choice : 3
Queue elements :
2
1.Insert

```


11. Develop a C program to traverse an ordered list implemented using a linked list and delete all the nodes whose keys are negative.

->

```
#include <stdio.h>

#include <stdlib.h>

struct node
{
    int data;    // Data
    struct node *next; // Address
} * head;

/* Function declaration */
void createList(int n);
int deleteAllByKey(int key);
void displayList();

int main()
{
    int n, key, totalDeleted;

    printf("Enter number of node to create: ");
    scanf("%d", &n);

    createList(n);

    printf("\nData in list before deletion\n");

    displayList();

    printf("\nEnter element to delete with key: ");
    scanf("%d", &key);

    totalDeleted = deleteAllByKey(key);

    printf("%d elements deleted with key %d.\n", totalDeleted, key);

    printf("\nData in list after deletion\n");

    displayList();

    return 0;
}

void createList(int n)
```

```

{
    struct node *newNode, *temp;

    int data, i;
head = malloc(sizeof(struct node));
if (head == NULL)
    {
        printf("Unable to allocate memory. Exiting from app.");
        exit(0);
    }
    /* Input head node data from user */
    printf("Enter data of node 1: ");
    scanf("%d", &data);
head->data = data; // Link data field with data
    head->next = NULL; // Link address field to NULL
temp = head;
for (i = 2; i <= n; i++)
    {
        newNode = malloc(sizeof(struct node));
if (newNode == NULL)
        {
            printf("Unable to allocate memory. Exiting from app.");
            exit(0);
        }
        printf("Enter data of node %d: ", i);
        scanf("%d", &data);
newNode->data = data; // Link data field of newNode
        newNode->next = NULL; // The newNode should point to nothing
temp->next = newNode; // Link previous node i.e. temp to the newNode
        temp = temp->next;
    }
}

```

```

void displayList()
{
    struct node *temp;
    if (head == NULL)
    {
        printf("List is empty.\n");
        return;
    }
    temp = head;
    while (temp != NULL)
    {
        printf("%d, ", temp->data); // Print data of current node
        temp = temp->next;          // Move to next node
    }
    printf("\n");
}

int deleteAllByKey(int key)
{
    int totalDeleted = 0;
    struct node *prev, *cur;
    while (head != NULL && head->data == key)
    {
        prev = head;
        head = head->next;
        free(prev);
        totalDeleted++;
    }
    prev = NULL;
    cur = head;
    while (cur != NULL)
    {

```

```
    if (cur->data == key)
    {
        if (prev != NULL)
        {
            prev->next = cur->next;
        } free(cur);
        cur = prev->next;
        totalDeleted++;
    }
    else
    {
        prev = cur;
        cur = cur->next;
    }
}

return totalDeleted;
}
```

12. Develop a C program to implement all basic operations of queue to store floating point numbers.

->

```
#include <stdio.h>
#include<stdlib.h>
#define MAX 50
void insert();
void delete();
void display();
int queue_array[MAX];
int rear = - 1;
int front = - 1;
int main()
{
int choice;
while (1)
{
printf("1.Insert element to queue n");
printf("2.Delete element from queue n");
printf("3.Display all elements of queue n");
printf("4.Quit n");
printf("Enter your choice : ");
scanf("%d", &choice);
switch(choice)
{
case 1:
insert();
break;
case 2:
delete();
break;
case 3:
```

```
display();  
break;  
case 4:  
exit(1);  
default:  
printf("Wrong choice n");  
}  
}  
}  
void insert()  
{  
int item;  
if(rear == MAX - 1)  
printf("Queue Overflow n");  
else  
{  
if(front == - 1)  
front = 0;  
printf("Inset the element in queue : ");  
scanf("%d", &item);  
rear = rear + 1;  
queue_array[rear] = item;  
}  
}  
void delete()  
{  
if(front == - 1 || front > rear)  
{  
printf("Queue Underflow n");  
return;  
}
```

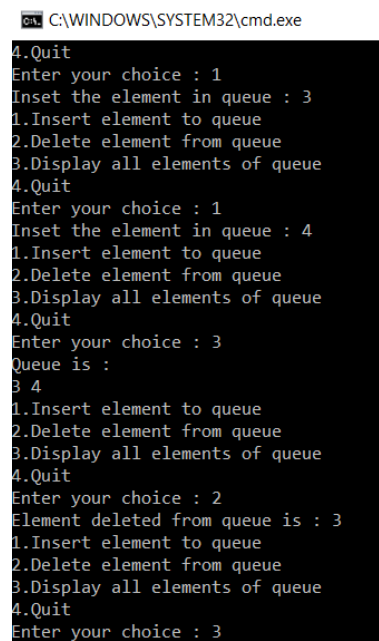
```

else
{
printf("Element deleted from queue is : %dn", queue_array[front]);
front = front + 1;
}
}

void display()
{
int i;
if(front == - 1)
printf("Queue is empty n");
else
{
printf("Queue is : n");
for(i = front; i <= rear; i++)
printf("%d ", queue_array[i]);
printf("n");
}
}

```

OUTPUT:



```

C:\WINDOWS\SYSTEM32\cmd.exe
4.Quit
Enter your choice : 1
Inset the element in queue : 3
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 1
Inset the element in queue : 4
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 3
Queue is :
3 4
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 2
Element deleted from queue is : 3
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 3

```

