5. Queue Data Structure

What is Queue?

Queue is a data structure which is used to handle data in a first-in-first-out (FIFO) method. That is we can remove the element which has been added earlier from the queue first.

Common operations of Queue are:

initializeQueue() – initializes the queue as empty queue.
enQueue()- adds an element at the rear of the queue.
deQueue()-removes and returns the front element from the queue.
frontElt()-returns the front element without removing it.
isEmpty() - returns true if the queue has no elements and false otherwise.
isFull() - returns true if the queue is full of elements and false otherwise.
displayQueue() - displays all elements from front to rear.

Graphical Representation of Queue Operation:

1. initializeQueue()					
2. p=isEmpty()					
p = true					
3. enQueue(5)	5				
4. enQueue(9)					
enQueue(7)	5	9	7		
5. x=deQueue()	9	7			
x = 5					
6. enQueue(2)					
enQueue(6)	9	7	2	6	
7. q = isFull()	9	7	2	6	
q = false					
8. enQueue(3)	9	7	2	6	3
9. r = isFull()					
y = deQueue()	7	2	6	3	
r = true					
y = 9					

Static (Array based) Implementation of Queue Operations [Graphical Representation]:

	0	1	2	3	4
1. initializeQueue()					

front -1
rear -1
size 0

2. p=isEmpty()

0	1	2	3	4

front -1
rear -1
p = true size 0

3. enQueue(5)

0	1	2	3	4
5				

front -1
rear 0
size 1

4. enQueue(9)
enQueue(7)

0	1	2	3	4
5	9	7		

front -1
rear 2
size 3

5. x=deQueue()

0	1	2	3	4
	9	7		

front 0
rear 2
x = 5 size 2

6. enQueue(2)
enQueue(6)

0	1	2	3	4
	9	7	2	6

front 0
rear 4
size 4

7. q = isFull()

0	1	2	3	4
	9	7	2	6

front 0
rear 4
q = false size 4

8. enQueue(3)

0	1	2	3	4
3	9	7	2	6

front 0
rear 0

		size	5				
			0	1	2	3	4
9. r = isFull()			3		7	2	6
y = deQueue()							
		front	1				
r = true		rear	0				
y = 9		size	4				

Static (Array based) Implementation of Stack Operations [C++ Code]:

```
#include<iostream.h>
#include<conio.h>

const Q_SIZE=5;

class Queue
{
private:
   int front, rear, size;
   int que[Q_SIZE];

public:
   Queue();
   void initializeQueue();
   void enQueue(int);
   int deQueue();
   int frontElt();
   int isEmpty();
   int isFull();
   void displayQueue();
}

Queue::Queue()
{
   front=(-1);
   rear=(-1);
   size=0;
}

void Queue::initializeQueue()
{
   front=(-1);
   rear=(-1);
   size=0;
}

void Queue::enQueue(int elt)
{
   if (size < Q_SIZE)
   {
      rear=(rear+1)%Q_SIZE;
      que[rear]=elt;
      size++;
   }                          //Else cout<<"Queue is full"
}
```

```
int Queue::deQueue()
{
  if (size > 0)
  {
    front=(front+1)%Q_SIZE;
    size--;
    return que[front];
  }
  else
    return 999; //Some invalid integer should be returned or cout<<"Queue is empty"
}

int Queue::frontElt()
{
  if (size>0)
  {
    return que[(front+1)%Q_SIZE];
  }
  else
    return 999; //Some invalid integer should be returned or cout<<"Queue is empty"
}

int Queue::isEmpty()
{
  return (size == 0);
}

int Queue::isFull()
{
  return (size == Q_SIZE);
}

void Queue::displayQueue()
{
  int i=front;
  for (int j=1;j<=size;j++)
  {
    i=(i+1)%Q_SIZE;
    cout<<que[i]<<endl;
  }
}

void main()
{
  clrscr();
  Queue q;
  q.enQueue(5);
  q.enQueue(9);
  q.enQueue(7);
  int x=q.deQueue();
  q.enQueue(2);
  q.enQueue(6);
  q.enQueue(3);
  int y=q.deQueue();
  q.displayQueue();
}
```

Output: 7 2 6 3

Dynamic (Linked List based) Implementation of Queue Operations:
initializeQueue() - front=NULL; //Similar to initialzeList() and it is better to use front instead of head.
enQueue() - newNode->next=front; front=newNode; //Similar to insertAtFront()
deQueue() - Move to the last node, get the data, remove the last node, return the data.
frontElt() - Move to the last node, get the data, remove the last node, return the data.
isEmpty() - if (front==NULL) return 1 else return 0
isFull() - return 0; //Always return false
displayQueue() - Similar to displayList()

Advantages of Queue:
First-in-first-out access

Disadvantages of Queue:
Difficult to access other items