MODULE 2
RECURSION AND QUEUE

RECURSION

Recursion is a technique of defining something in terms of itself. In the field of mathematics and computer science, many concepts can be explained using recursion. In computer terminology, if a function calls itself then it is known as recursive function. If the function calls itself directly, then it is known as *direct recursion.* For Example:

```
void myfun()
{
    -----
    myfun();
    -----
}
```

If the function calls itself through another function, then it is known as *indirect recursion.* For example:

```
void myfun()
{
    -----
    -----
    fun();
    -----
    -----
}
void fun()
{
    ------
    ------
    myfun();
    -------
    -------
}
```

Here, the function myfun() is calling the function fun(), which in turn calls myfun().

Factorial of a Number

Factorial of a non-negative integer n is defined as the product of all integers from 1 to n. That is:

n! = n(n-1)(n-2)…3.2.1 , for all n>=1 and
0! = 1.

By definition we have, n!= n(n-1)!

Thus, factorial is a recursive function.

Consider one example of finding 5! –

5! = 5*(5-1)!
= 5*4!

= 5*4*3!
= 5*4*3*2!
= 5*4*3*2*1!
= 5*4*3*2*1*0!
= 5*4*3*2*1*1
= 120

This procedure can be implemented through program given below:

```
#include<stdio.h>

int fact(int n)
{
        if (n==0)
                return      1;
        return n* fact(n-1);
}

void main()
{
        int n;

        printf(“Enter the value of n”);
        scanf(“%d”, &n);

        if(n<0)
                printf(“Invalid input”);
        else
                printf(“\nFactorial of %d is %d “, n, fact(n)) ;
}
```

Fibonacci Sequence

A sequence of integers is called Fibonacci sequence if an element of a sequence is the sum of its immediate two predecessors. That is:

f(x)=0 for x=1
f(x)=1 for x=2
f(x)=f(x-2)+f(x-1) for x>2

Thus the Fibonacci sequence is 0, 1, 1, 2, 3, 5, 8, 13, 21, ….

Consider an example of finding 6th Fibonacci number.

f(6) = f(4) + f(5)
= f(2) + f(3) + f(3) + f(4)
= 1 + f(1) + f(2) + f(1) + f(2) + f(2)+ f(3)
= 1 + 0 + 1 + 0 + 1 + 1 + f(1) +(2)
= 4 + 0 + 1
= 5

The above procedure can be illustrated through the program given below:

Finding nth Fibonacci number

```
#include<stdio.h>

int fibo(int n)
{
        if (n==1)
                return 0;
        else if (n==2)
                return 1;
        return fibo(n-1) + fibo (n-2);
}

void main()
{
        int n;
        printf(“Enter value of n”);
        scanf(“%d”, &n);
        if(n<=0)
                printf(“invalid input”);
        else
                printf(“%d th fibonacci number is %d “, n, fibo(n));
}
```

The output would be:

Enter the value of n : 10

10^{th} fibonacci number is 34

Greatest Common Divisor (GCD)

The GCD of two numbers (at least one of them should be non-zero) can be computed using Euclid's algorithm as –

$$GCD(m,n) = \begin{cases} GCD(n, m\%n), & \text{if n!=0} \\ m & \text{if n= 0} \end{cases}$$

Consider an example of finding GCD of 75 and 20.

GCD(75, 20) = GCD(20, 75%20)
= GCD(20, 5)
= GCD(5, 20%5)
= GCD(5, 0)
= 5

Program for finding GCD of two numbers:

```
#include<stdio.h>

int GCD(int m, int n)
{
        if (n==0)
                return          m;
        return GCD(n, m%n);
}
void main()
{
        int m, n, gcd;
        printf("Enter two positive integers:");
        scanf("%d%d", &m, &n);

        if(m==0 && n==0)
        {
                printf("At least one of the numbers should not be zero!!");
                return;
        }

        gcd=GCD(m,              n);
        printf("\nGCD=%d ", gcd);
}
```

Tower of Hanoi Problem

In the problem of Tower of Hanoi, there will be three poles, viz. A, B and C. Pole A (source pole) contains 'n' discs of different diameters and are placed one above the other such that larger disc is placed below the smaller disc. Now, all the discs from source (A) must be transferred to destination (C) using the pole B as temporary storage.

Conditions are:

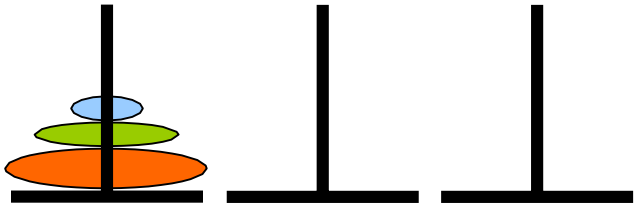
- Only one disc must be moved at a time
- smaller disc is on the top of larger disc at every step

Algorithm to move n discs from source to destination is as follows –

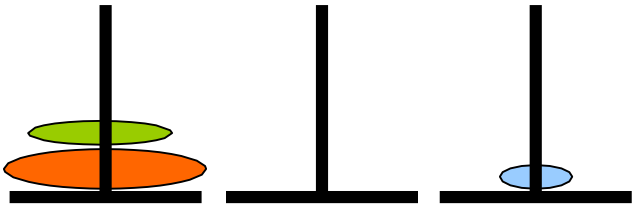
(i) move n-1 discs from source to temporary
(ii) move n^{th} disc from source to destination
(iii) move n-1 disc from temporary to destination

For example, consider the number of discs = 3. Then, various steps involved in transferring 3 discs from source to destination are shown below –

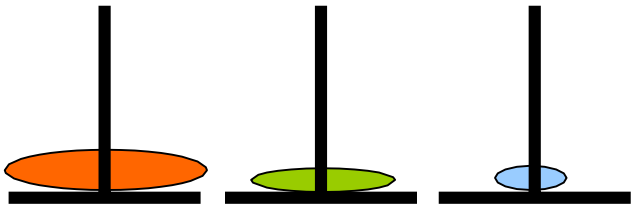
Initial Stage



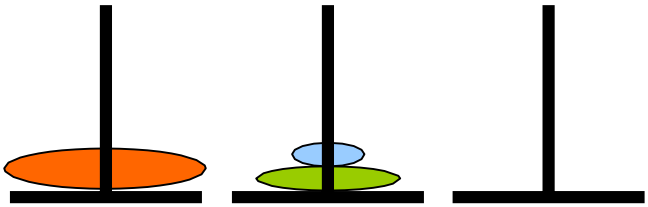
Step 1:



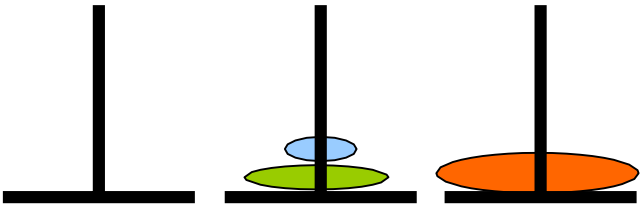
Step 2:



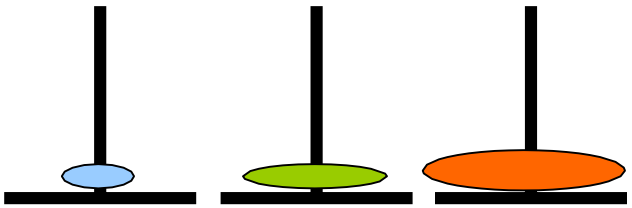
Step 3:



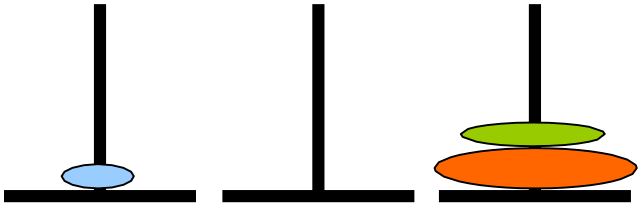
Step 4:



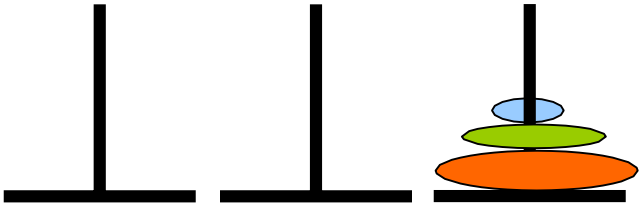
Step 5



Step 6



Step 7



Program:

```
#include<stdio.h>
int count=0;

void tower(int n, char s, char t, char d)
{
        if(n==1)
        {
                printf("Move disc 1 from %c to %c ", s, d);
                count ++;
                return;
        }
        tower(n-1, s, d, t);
        printf("Move disc %d from %c to %c", n, s,d);
        count ++;
        tower(n-1, t, s, d);
}
```

```
void main()
{
        int n;
        printf("Enter the number of discs");
        scanf("%d", &n);
        tower(n, 'A','B','C');
        printf("Total number of moves =%d", count);
}
```

Output: Enter the number of discs: 3
Move disc 1 from A to C
Move disc 2 from A to B
Move disc 1 from C to B
Move disc 3 from A to C
Move disc 1 from B to A
Move disc 2 from B to C
Move disc 1 from A to C
Total number of disc moves = 7

Properties of Recursive Algorithms

When a function calls itself, a new set of local variables and parameters are being allocated memory in the stack. Then the function code is executed from the top with these new variables. A recursive call does not make a new copy of the function, only the values being operated upon are new. When each recursive call returns, the old local variables and parameter are removed from the stack and the execution resumes at the point of the function call inside the function.

Some facts about recursive functions are:

- Recursive functions do not usually reduce the code size.
- They do not improve memory utilization compared to iterative functions.
- Many of the recursive functions may execute a bit slower compared to iterative functions because of the overhead of repeated function calls.
- They may cause a stack overrun, as each new call to the recursive function creates a new copy of variables and parameters and puts them into the stack.
- The advantage of using recursive functions is to create clearer and simpler programs.
- Moreover, some of the algorithms do require recursive functions as the iterative versions of them are quite difficult to write and implement.
- While writing a recursive function, a conditional statement has to be included which will terminate the recursive function without using a recursive function call.
- Otherwise, the recursive function will enter into an infinite loop and will not terminate at all.

Thus, any recursive function should satisfy the following conditions:

- In each and every call, the function must be nearer to the solution. (In other words, at every step, the problem size must reduce)
- There should be at least one non-recursive exit condition.

QUEUES

In our day-to-day life, we come across many situations in which we have to stand in a queue like at a bank counter, at a cinema hall etc. In the field of computer science also we can give the examples for queue system such as printing a several files using only one printer, Processes waiting to be executing by CPU etc. Using this analogy, the data structure 'queue' is defined.

Queue is a non-primitive linear data structure, where the elements are inserted at rear end and deleted from the front end.

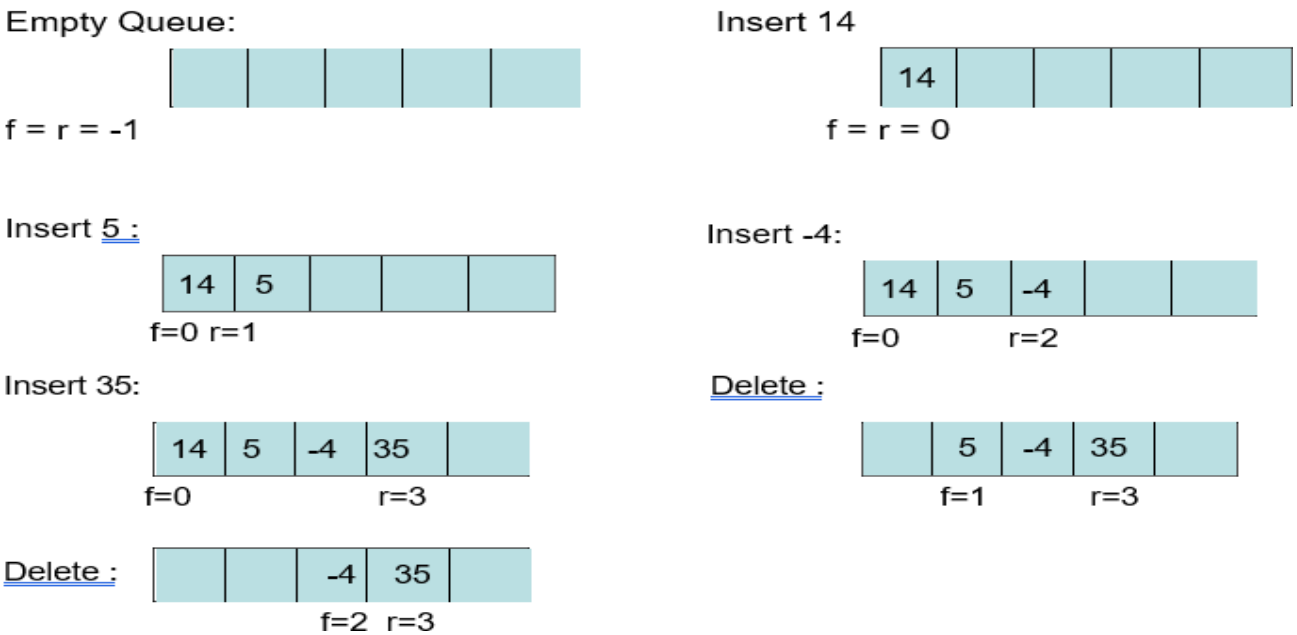
The item inserted first will be the first to be deleted. Hence it is known as ***First In First Out (FIFO)*** data structure.

The primitive operations of a queue structure include:

- Inserting an element into queue
- Deleting element from queue
- Displaying the contents of a queue

An attempt to insert an element into a full queue is called as ***Queue overflow***. Trying to delete an element from an empty queue is known as ***queue underflow.*** The status of the queue is maintained by two variables viz. ***front*** and ***rear.*** Initially, both *front* and *rear* are set to -1 to indicate empty queue. While inserting the very first element into queue, both *front* and *rear* are incremented. After that, in every insertion, *rear* will be incremented. The *front* will be incremented at every deletion.

Consider the following illustration to understand the working of queue.



Note that, after every insertaion, ***rear*** is incremented and after every deletion, ***front*** is

incremented.

Program for Primitive operations on ordinary queue

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#define MAX 3

void insert(int q[ ], int *f, int *r, int item)
{
        if(*r==MAX-1)
        {
                printf("\nQueue overflow");
                return;
        }
        q[++(*r)]=item;

        if(*f== -1)
                (*f)++;
}

void del(int q[], int *f, int *r)
{
        if(*f==-1 || *f>*r)
        {
                printf("\nQueue underflow");
                return;
        }
        printf("\nDeleted item is %d", q[(*f)++]);
}

void disp(int q[], int *f, int *r)
{
        int i;

        if(*f== -1 || *f>*r)
        {
                printf("\nNo elements to display!!");
                return;
        }
        printf("\n Contents of queue:\n");

        for(i=*f;i<=*r;i++)
                printf("%d\t",q[i]);
}
```

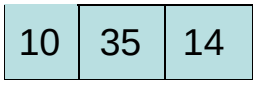
```
void main()
{
        int q[MAX], f=-1, r=-1,item, opt;

        for(;;)
        {
                printf("\n*****Queue                operations*****");
                printf("\n1.Insert\n 2.Delete\n 3.Display \n 4.Exit");
                printf("\nEnter your option: ");
                scanf("%d",&opt);
                switch(opt)
                {

                        case 1: printf("\nEnter item to be inserted:");
                                scanf("%d",&item);
                                insert(q,&f,&r,item);
                                break;
                        case 2: del(q, &f, &r);
                                break;
                        case 3: disp(q, &f,&r);
                                break;
                        case            4:
                        default:exit(0);
                }
        }
}
```

Circular Queue

We have seen in the previous program that when an element is deleted from the queue, the **front** is incremented and when the element is inserted, **rear** is incremented. Such implementation of a queue has a drawback. To understand the drawback, consider the following illustration:

Assume, we have inserted the elements into queue up to its full capacity. That is, *rear* reached maximum size. Then, if we delete some elements from front end, there will be empty spaces at the beginning of a queue. But still, we can't insert elements into queue, as *rear* is already at *MAX.*

Let *MAX=3*



f=0 r=2

Delete twice:

f=r=2

Now, if we try to insert an item, there will be queue overflow, as the condition (r==MAX-1) is true. But, we can see that, two positions in the beginning are actually free to receive the elements. To overcome this problem, we adopt ***circular queue.***

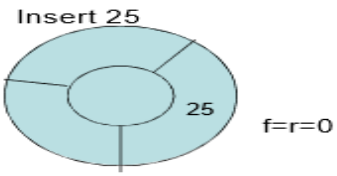
In ordinary queue, we will just increment ***front*** and ***rear***. Hence, there is chance that ***rear*** will go beyond the range of the size of the queue. Hence, in circular queue, instead of using the statements like

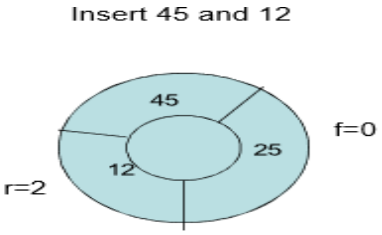
f= f+1 and r=r+1,

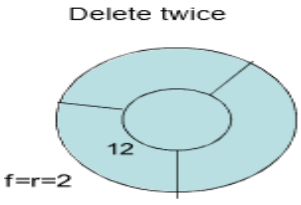
we use

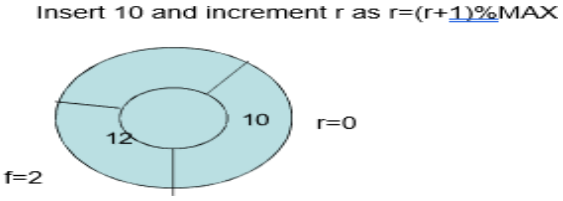
f=(f+1)%MAX and r=(r+1)%MAX

This will ensure that both ***front*** and ***rear*** will fall within the range of MAX always. This can be illustrated as below –









Thus, we can overcome the problem with ordinary queue using circular queue.

Program for Implementation of Circular Queue

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#define MAX 3

void insert(int q[], int *f, int *r, int item)
{
        if(*f==(*r+1)%MAX)
        {
                printf("\nQueue overflow");
                return;
        }

        *r=(*r+1)%MAX;
        q[*r]=item;

        if(*f== -1)
                (*f)++;
}

void del(int q[], int *f, int *r)
{
        if(*f==-1 )
        {
                printf("\nQueue underflow");
                return;
        }

        printf("\nDeleted element is %d", q[*f]);

        if(*f==*r)
                *f=*r=-1;
else
                *f=(*f+1)%MAX;
}

void disp(int q[], int *f, int *r)
{
        int i;
        if(*f== -1)
```

```
		{
			printf("\nNo elements to display!!");
			return;
		}
		printf("\n Contents of queue:\n");

		if(*f>*r)
		{
			for(i=*f;i<MAX;i++)
				printf("%d\t",q[i]);
			for(i=0;i<=*r;i++)
				printf("%d\t", q[i]);
		}
		else
		{
			for(i=*f;i<=*r;i++)
				printf("%d\t",q[i]);
		}
}

	void main()
	{
		int q[10], f=-1, r=-1,item, opt;

		for(;;)
		{
			printf("\n*****Circular Queue operations*****");
			printf("\n1.Insert\n 2.Delete\n 3.Display \n 4.Exit");
			printf("\nEnter your option: ");
			scanf("%d",&opt);
		switch(opt)
		{
			case 1: printf("\nEnter item to be inserted:");
			scanf("%d",&item);
			insert(q,&f,&r,item);
			break;
			case 2: del(q, &f, &r);
			break;
			case 3: disp(q, &f,&r);
			break;
			case 4:
			default:exit(0);
		}
		}

		}
```

Priority Queue

Priority Queue is a data structure in which the items are served (deleted) based on their priority levels. The insertion and deletion operations of priority queue are based on the priority of the elements. The element with highest priority is processed first and the element with second highest priority is processed next and so on. The use of this data structure is in job scheduling algorithms in the design of operating system.

These are two different types of priority queues viz.

- Ascending priority queue.
- Descending priority queue.

In both the methods the items are inserted in any order. But in ascending priority queue the smallest element is deleted first, while in the descending priority queue, the largest element is deleted first.

However, the term *smallest* in the above statement not necessarily mean the value of the element, but it may be any quantity associated with the element. For example, we can think of stack as a priority queue, in which the elements are deleted based on the time of insertion as a priority level. That is, the item inserted most recently (i.e. least time) will be deleted first. Similarly, ordinary queue can be thought of as a priority queue, where deletion is based on maximum time spent by the element in a queue. That is, the item which spent maximum time (inserted at the beginning of the process) will be deleted first.

One should note that, the meaning of deletion here (in the study of queue data structures, as well) indicates providing the service for which the item/element is waiting in a queue.

Priority queue can be implemented using arrays. In further discussion with priority queue, we assume a queue of integers where, the value of item itself indicates its priority. That is, smallest item has higher priority.

As discussed earlier, in priority queue, the insertion of elements can be in any order, but the deletion is based on priority. That is, in ascending priority queue, the smallest item must be searched for, and then it should be deleted. Hence, element present at any in between position of the array must be deleted and rest of elements must be re-adjusted. Thus, the deletion process becomes difficult.

So to avoid this problem and for the sake of simplicity, we assume that the elements are inserted in an ascending order, so that the deletion requires only deleting the element at front end. Thus, in the programmatic implementation of priority queue, during insertion, we should see that items are inserted in a proper position to maintain ascending order.

Program for Implementation of Priority Queue

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#define MAX 3

void insert(int PQ[], int *f, int *r, int item)
{
        int j;

        if (*r==MAX-1)
        {
                printf("\nQueue overflow");
                return;
        }

        j=*r;
        while( j>=0 && item<PQ[ j ])
        {
                PQ[ j+1]=PQ[ j ];
                j - -;
        }

        PQ[ j+1]=item;
        (*r)++;

        if(*f== -1)
                (*f)++;
}

void del(int PQ[ ], int *f, int *r)
{
        if(*f==-1 || *f>*r)
        {
                printf("\nQueue underflow");
                return;
        }

        printf("\nDeleted item is %d", PQ[(*f)++]);
}
```

```
void disp(int PQ[], int *f, int *r)
{
        int i;

        if(*f== -1 || *f>*r)
        {
                printf("\nQueue underflow");
                return;
        }

        printf("\nContents of priority queue:");
        for(i=*f;i<=*r;i++)
                printf("%d\t", PQ[i]);
}

void main()
{
        int PQ[MAX], f=-1,r=-1, item, opt;

        for(;;)
        {
                printf("\n****Priority                    Queue****\n");
                printf("1.Insert\n2.Delete\n3.Display\n4.Exit\n");
                printf("\nEnter your option:");
                scanf("%d",&opt);

                switch(opt)
                {
                        case 1: printf("\nEnter the item to be inserted:");
                                scanf("%d",&item); insert(PQ,&f,&r,item);
                                break;
                        case 2: del(PQ, &f, &r);
                                break;
                        case 3:disp(PQ, &f,&r);
                                break;
                        case 4:
                        default:
                                exit(0);

                }

        }

}
```

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.

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

We can also perform peek operations in the deque along with the operations listed above. Through peek operation, we can get the deque's front and rear elements of the deque. So, in addition to the above operations, following operations are also supported in deque –

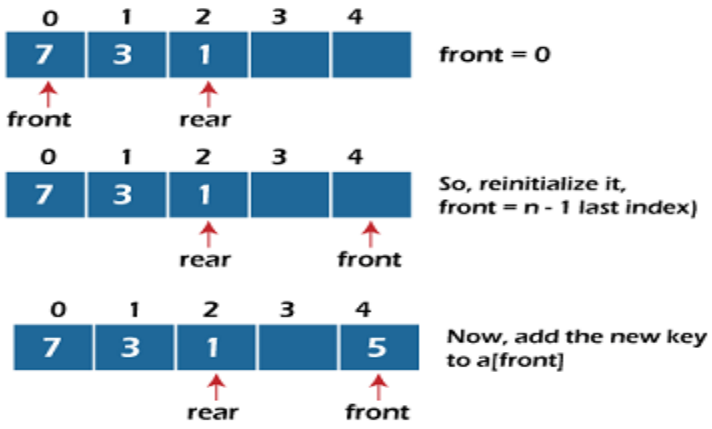
- Get the front item from the deque
- Get the rear item from the deque
- Check whether the deque is full or not
- Checks whether the deque is empty or not

Now, let's understand the operation performed on deque using an example.

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 -

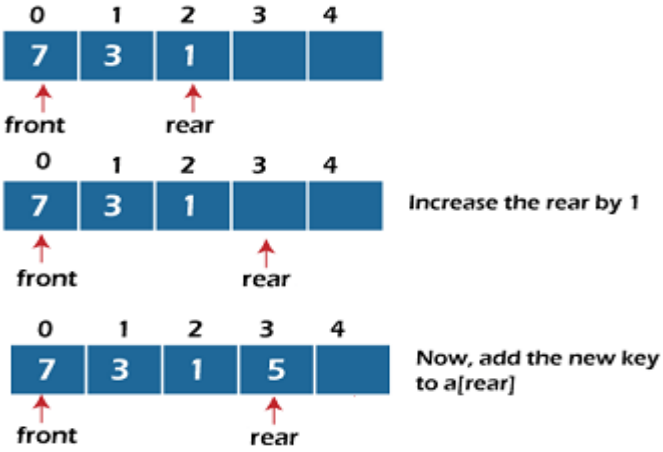
- If the queue is empty, both rear and front are initialized with 0. Now, both will point to the first element.
- Otherwise, check the position of the front if the front is less than 1 (front < 1), then reinitialize it by **front = n - 1**, i.e., the last index of the array.



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.

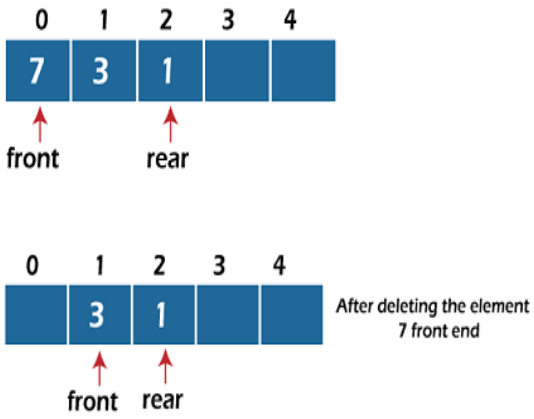
If the queue is empty, i.e., front = -1, it is the underflow condition, and we cannot perform the deletion. If the queue is not full, then the element can be inserted from the front end by using the below conditions -

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).

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



Deletion at the rear end

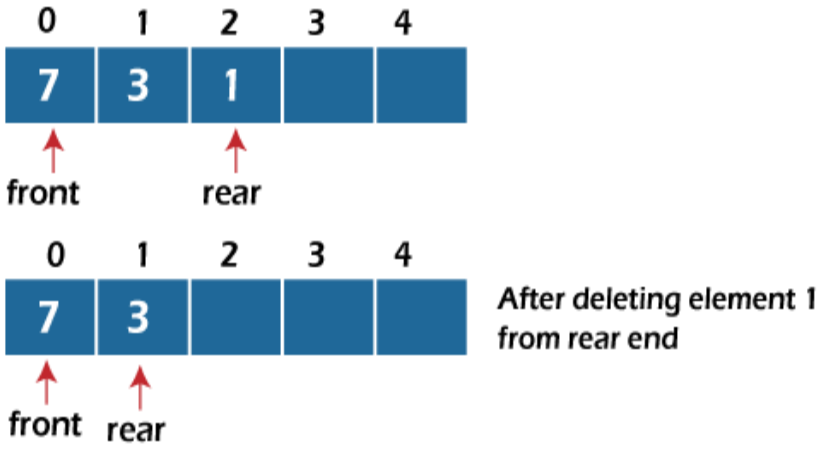
In this operation, the element is deleted from the rear end of the queue. Before implementing the operation, we first have to check whether the queue is empty or not.

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).



Check empty

This operation is performed to check whether the deque is empty or not. If front = -1, it means that the deque is empty.

Check full

This operation is performed to check whether the deque is full or not. If front = rear + 1, or front = 0 and rear = n - 1 it means that the deque is full.

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]);
}

// 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
    delete_front();
    delete_rear();
    display(); // calling display function to retrieve values after deletion
    return 0;
}
```