

# Recursion

MODULE -2

# Definition and process

- Recursion is a process by which a function calls itself repeatedly ,until some specified condition is satisfied.

# factorial of a number

Definition:  $n! = 1 * 2 * \dots * (n-1) * n$  for  $n \geq 1$  and  $0! = 1$

Recursive definition of  $n!$ :  $F(n) = F(n-1) * n$  for  $n \geq 1$  and  $F(0) = 1$

**ALGORITHM**  $F(n)$

//Computes  $n!$  recursively

//Input: A nonnegative integer  $n$

//Output: The value of  $n!$

**if**  $n = 0$  **return** 1

**else return**  $F(n - 1) * n$

$$\begin{aligned} 5! &= 5 * (5-1)! \\ &= 5 * 4! \\ &= 5 * 4 * 3! \\ &= 5 * 4 * 3 * 2! \\ &= 5 * 4 * 3 * 2 * 1! \\ &= 5 * 4 * 3 * 2 * 1 * 0! \\ &= 5 * 4 * 3 * 2 * 1 * 1 \\ &= 120 \end{aligned}$$

# Program to find factorial of a number

- Given a positive integer  $n$ ,  $n$  factorial is defined as the product of all integers between  $n$  and 1.

```
#include<stdio.h>
int fact(int n)
{
    if (n==0)
        return 1;
    return n* fact(n-1);
}
```

# Multiplication of natural numbers

- If  $a$  and  $b$  are two +ve non zero integers, the product can be defined as the  $a$  is added to itself  $b$  times.
- ie,

$$a * b = a \quad \text{if } b == 1$$

$$a * b = a * (b-1) + a \quad \text{if } b > 1$$

# Algorithm

$$\text{Mul}(m,n) = \begin{cases} 0 & \text{if } (m=0 \text{ or } n=0) \\ m & \text{if } n=1 \\ \text{Mul}(m, n-1)+m & \text{otherwise} \end{cases}$$

Program : [rec mul.c](#)

# Fibonacci Sequence

- Fibonacci numbers are a series of numbers such that each number is the sum of the previous two numbers except the first and second number.

$$\text{Fib}(n) = \begin{cases} 0 & \text{if } n=0 \\ 1 & \text{if } n=1 \\ \text{fib}(n-1)+\text{fib}(n-2) & \text{if } n > 2 \end{cases}$$

$$\begin{aligned} f(6) &= f(4) + f(5) \\ &= \underline{f(2) + f(3)} + \underline{f(3) + f(4)} \\ &= 1 + \underline{f(1) + f(2)} + \underline{f(1) + f(2)} + \underline{f(2) + f(3)} \\ &= 1 + 0 + 1 + 0 + 1 + 1 + f(1) + (2) \\ &= 4 + 0 + 1 \\ &= 5 \end{aligned}$$

# GCD and LCM

## ● 4a. GCD LCM.C

- Diff ways to find gcd : using modulus(Euclid alg), repetitive subtraction(also, euclid alg), consecutive inter checking alg, middle school procedure using prime factor

## ● Alg : Using modulus

- 1. Compute remainder by dividing m by n,  $r = m \% n$

- 2. assign n to m .  $M \leftarrow n$

- 3. assign r to n .  $N \leftarrow r$

- Repeat as long as n is not 0

$$\begin{aligned} \text{GCD}(75, 20) &= \text{GCD}(20, 75 \% 20) \\ &= \text{GCD}(20, 5) \\ &= \text{GCD}(5, 20 \% 5) \\ &= \text{GCD}(5, 0) \\ &= 5 \end{aligned}$$



# Tower of Hanoi

- There are three pegs say A,B and C. there are  $n$  discs of different diameters with decreasing size. The object of the problem is to move the discs from peg A to peg C using peg B as an auxiliary. The rules are
- Only one disc can be moved at a time.
- At no time can a larger disc be placed on a smaller disc.



Original Configuration



Fourth Move



First Move



Fifth Move



Second Move



Sixth Move



Third Move

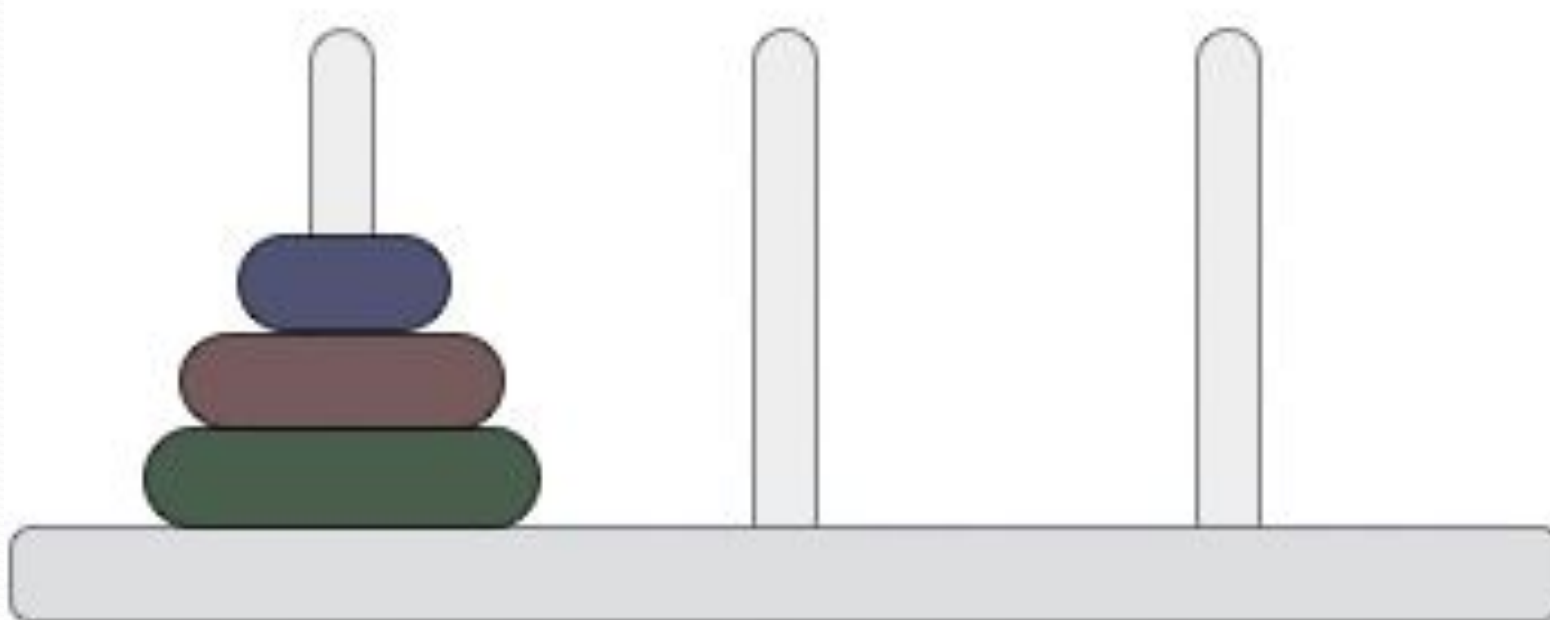


Seventh and Last Move

# Rules

- Only one disk can be moved among the towers at any given time.
- Only the "top" disk can be removed.
- No large disk can sit over a small disk.
- $2^n - 1$
- $2^3 - 1 = 7$

## Step: 0



ALGORITHM TOH( $n$ , A,C,B)

//description : to move the pegs from A to C by using B as an auxillary peg

//input: no of discs

//output: 'n' no of discs placed on C

if( $n=1$ )

{ write ("move disc from A to C"); return;}

else

{

//move( $n-1$ ) discs from A to B using C as auxillary

TOH( $n-1$ ,A,B,C)

// move remaining ( $n-1$ ) discs from B to C using A as auxillary

TOH( $n-1$ ,B,C,A)

}

}

- Move  $n-1$  disks from beg to aux
- Move 1 disk from beg to end
- Move  $n-1$  disks from aux to end

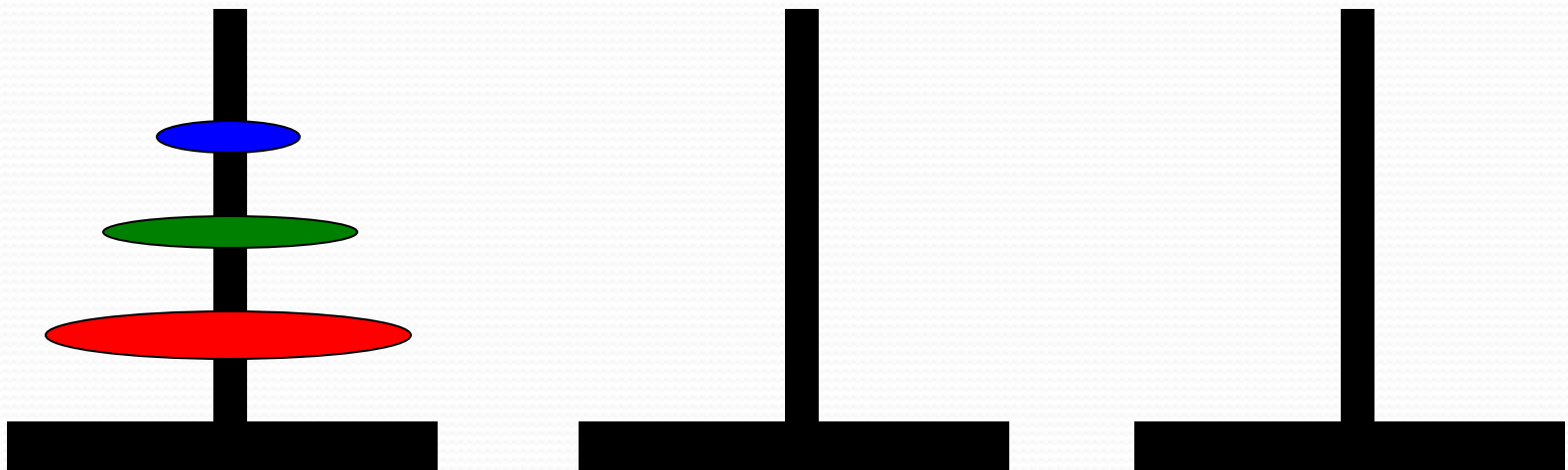
●  $T(3,a,b,c)$   $\square$  a is begining, b is auxiliary, c is end three poles

●  $T(2,a,c,b)$        $T(1,a,b,c)$   
                               $T(1,a,c,b)$   
                               $T(1,c,a,b)$

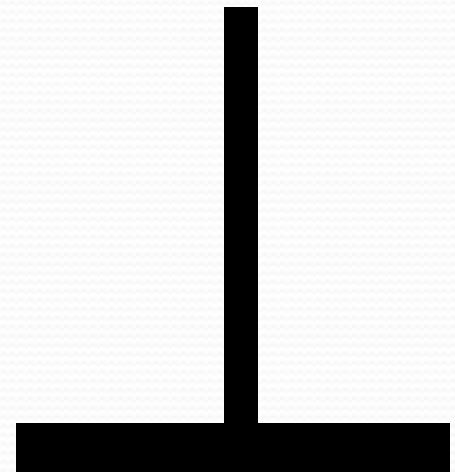
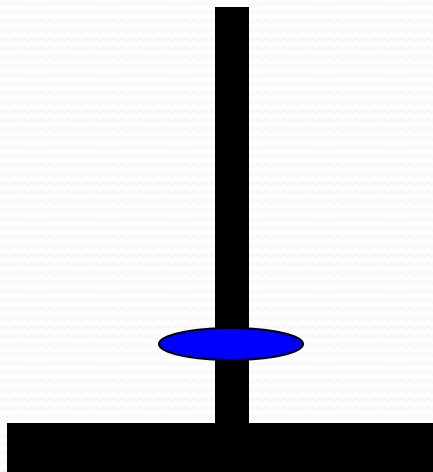
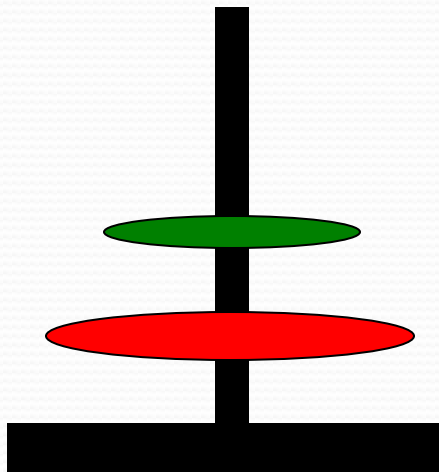
●  $T(1,a,b,c)$

●  $T(2,b,a,c)$        $T(1,b,c,a)$   
                               $T(1,b,a,c)$   
                               $T(1,a,b,c)$

# Tower of Hanoi

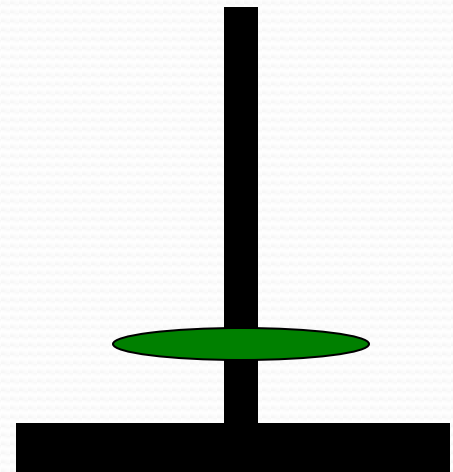
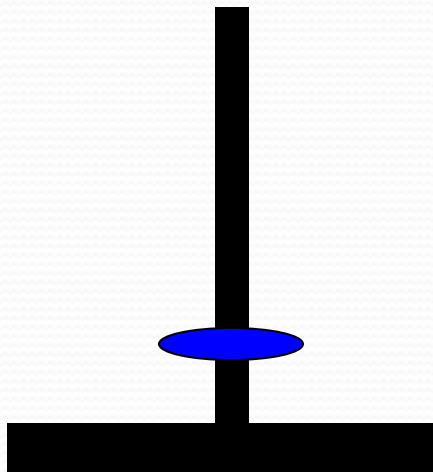
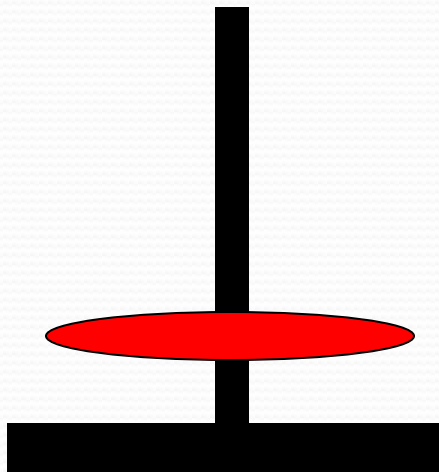


# Tower of Hanoi

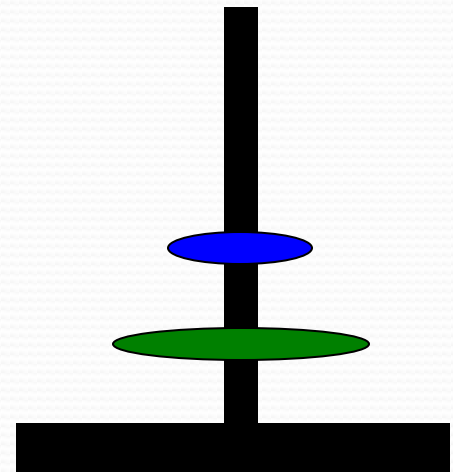
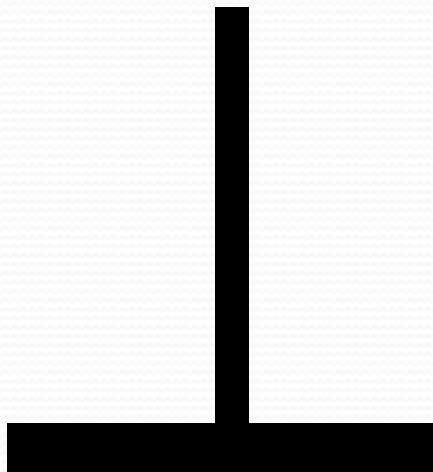
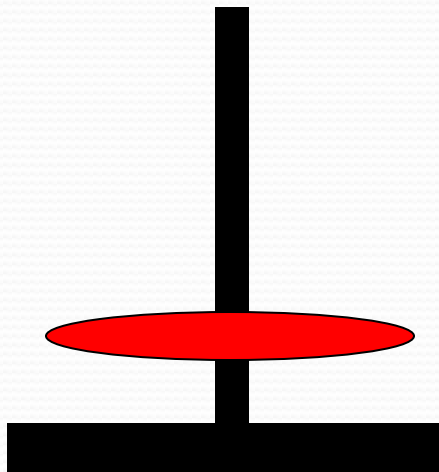




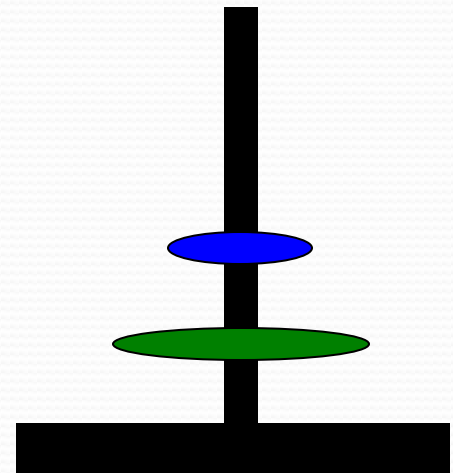
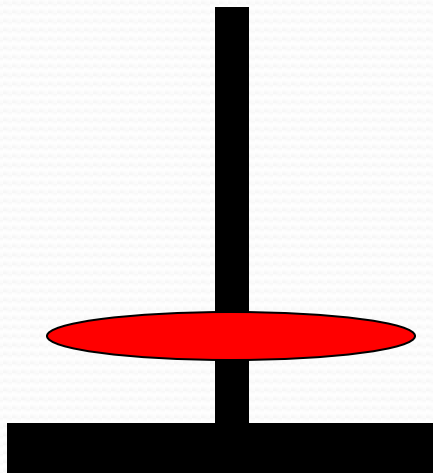
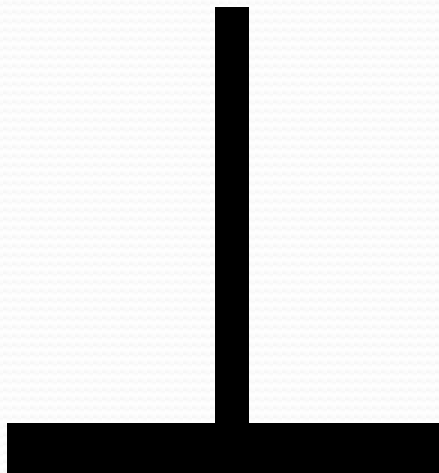
# Tower of Hanoi



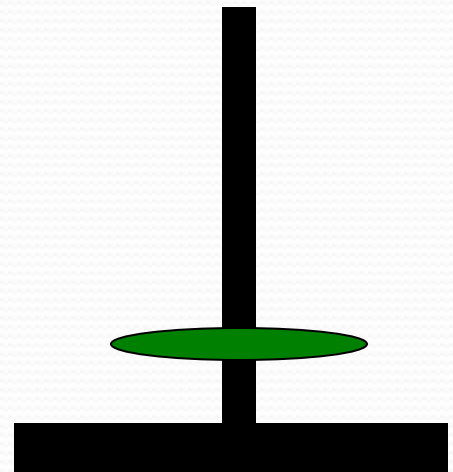
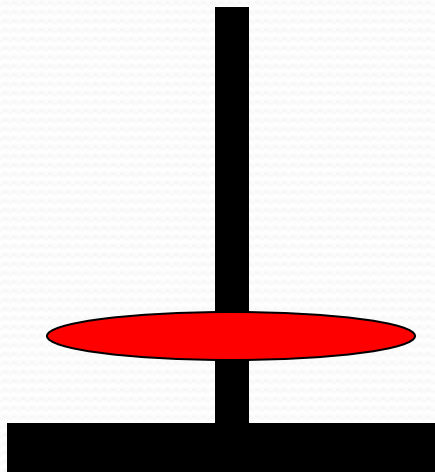
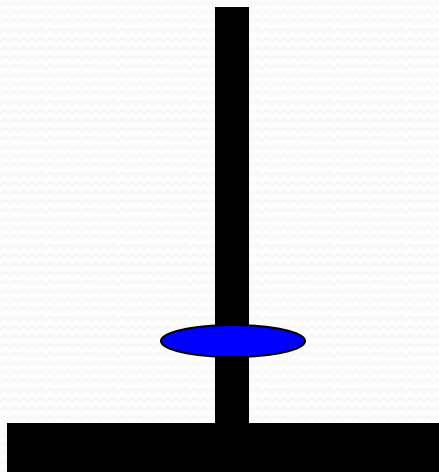
# Tower of Hanoi



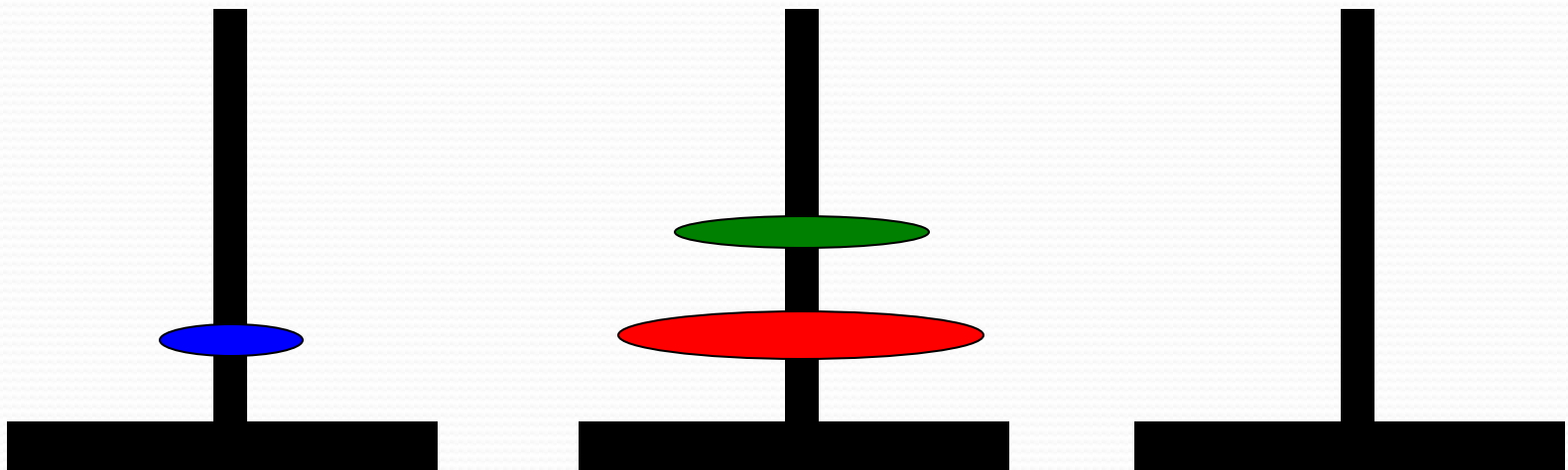
# Tower of Hanoi



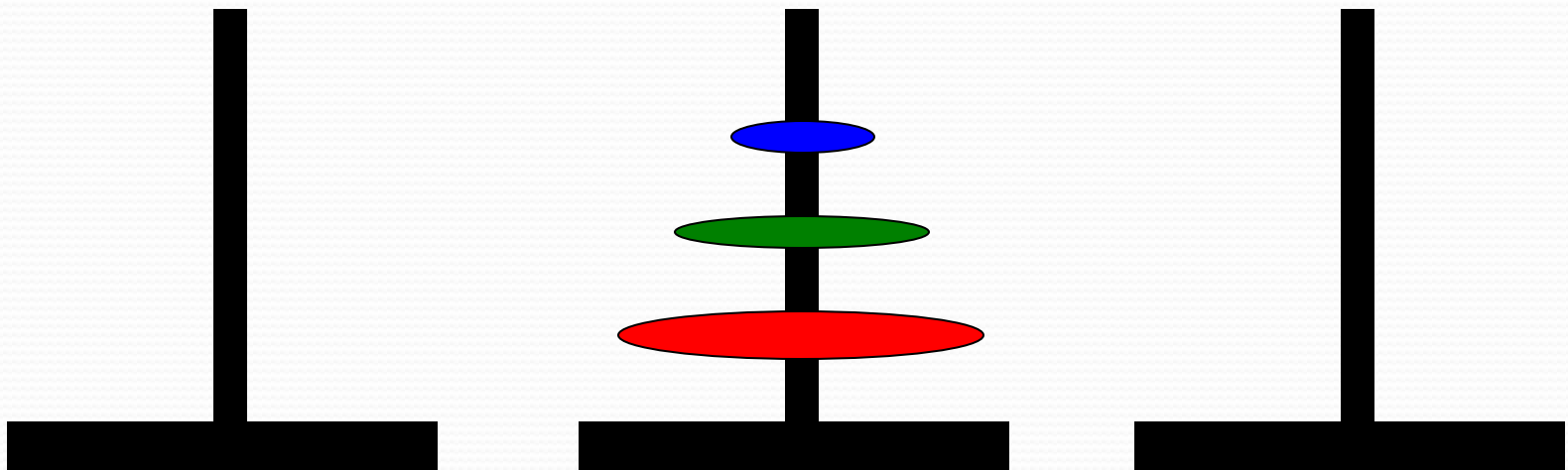
# Tower of Hanoi



# Tower of Hanoi



# Tower of Hanoi





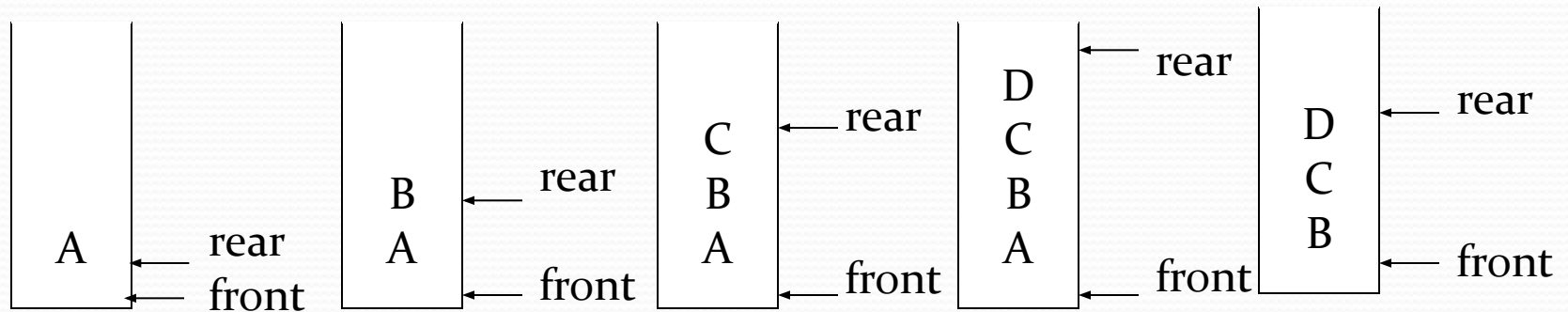
# Queue

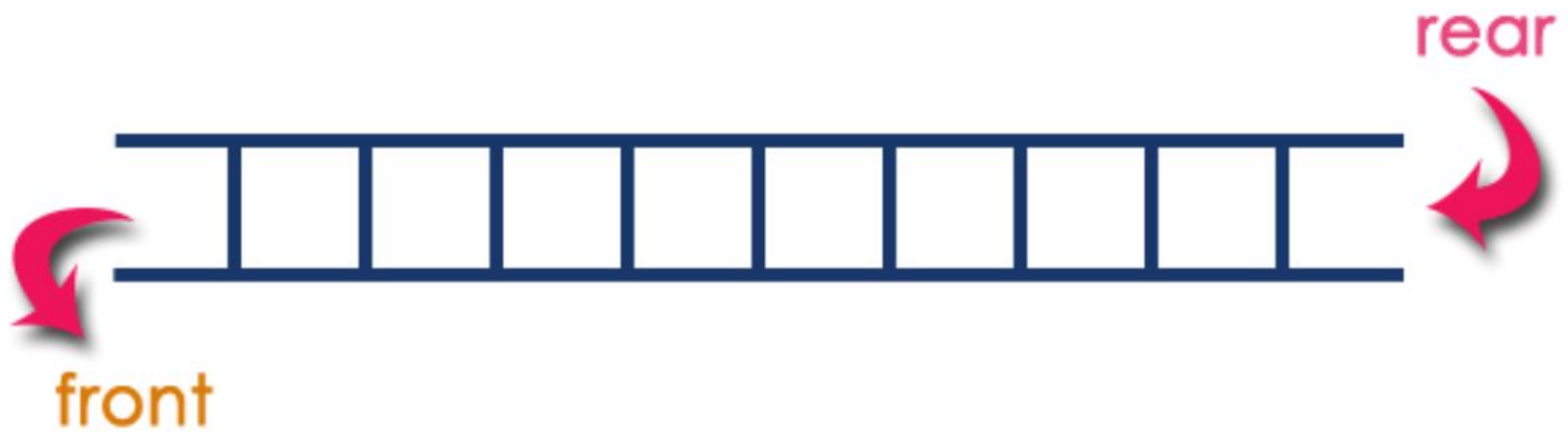
# Queues

- Stores a set of elements in a particular order.
- Queue principle:  
**FIRST IN FIRST OUT = FIFO**
- It means: the first element inserted is the first one to be removed.
- Defn – ordered collection of items from which items may be deleted at one end called front and items can be inserted at another end called rear.
- Stack principle is Last in first out (LIFO)
- Examples – line in a bank or group of cars at toll booth



# First In First Out





After Inserting five elements...



# Queue Operations using Array

- Queue data structure using array can be implemented as follows...

Before we implement actual operations, first follow the below steps to create an empty queue.

- **Step 1** - Include all the **header files** which are used in the program and define a constant '**SIZE**' with specific value.
- **Step 2** - Declare all the **user defined functions** which are used in queue implementation.
- **Step 3** - Create a one dimensional array with above defined SIZE (**int queue[SIZE]**)
- **Step 4** - Define two integer variables '**front**' and '**rear**' and initialize both with '**-1**'. (**int front = -1, rear = -1**)
- **Step 5** - Then implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on queue.

# enQueue(value) - Inserting value into the queue

- **Step 1** - Check whether **queue** is **FULL**. (**rear == SIZE-1**)
- **Step 2** - If it is **FULL**, then display "**Queue is FULL!!! Insertion is not possible!!!**" and terminate the function.
- **Step 3** - If it is **NOT FULL**, then increment **rear** value by one (**rear++**) and set **queue[rear] = value**.

# deQueue() - Deleting a value from the Queue

- **Step 1** - Check whether **queue** is **EMPTY**. (**front == rear**)
- **Step 2** - If it is **EMPTY**, then display "**Queue is EMPTY!!! Deletion is not possible!!!**" and terminate the function.
- **Step 3** - If it is **NOT EMPTY**, then increment the **front** value by one (**front ++**). Then display **queue[front]** as deleted element. Then check whether both **front** and **rear** are equal (**front == rear**), if it **TRUE**, then set both **front** and **rear** to **'-1'** (**front = rear = -1**).

# display() - Displays the elements of a Queue

- **Step 1** - Check whether **queue** is **EMPTY**. (**front == rear**)
- **Step 2** - If it is **EMPTY**, then display "**Queue is EMPTY!!!**" and terminate the function.
- **Step 3** - If it is **NOT EMPTY**, then define an integer variable '**i**' and set '**i = front**'.
- **Step 4** - Display '**queue[i]**' value and increment '**i**' value by one (**i++**). Repeat the same until '**i**' value reaches to **rear** (**i <= rear**)

- Three primitive operations that can be performed on queue –  $\text{insert}(q, x)$  insert item  $x$  at rear of  $q$ ,  $x = \text{remove}(q)$  delete from front of queue and  $\text{empty}(q)$  returning true or false depending on whether queue has elements
- Insert operation , can be performed with no limit to number of items added to queue.
- Remove can only be applied to non-empty queue, as no way to remove elements from empty queue
- Illegal attempt to remove element from empty queue, results in “underflow”

# Applications: Job Scheduling

front	rear	Q[0]	Q[1]	Q[2]	Q[3]	Comments
-1	-1					queue is empty
0	0	J1				Job 1 is added
0	1	J1	J2			Job 2 is added
0	2	J1	J2	J3		Job 3 is added
1	2		J2	J3		Job 1 is deleted
2	2			J3		Job 2 is deleted



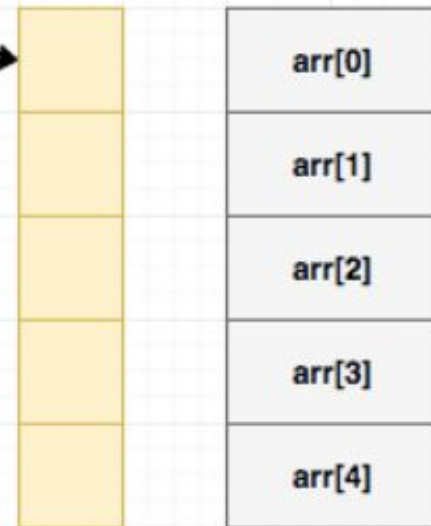
size = 5

arr[size]

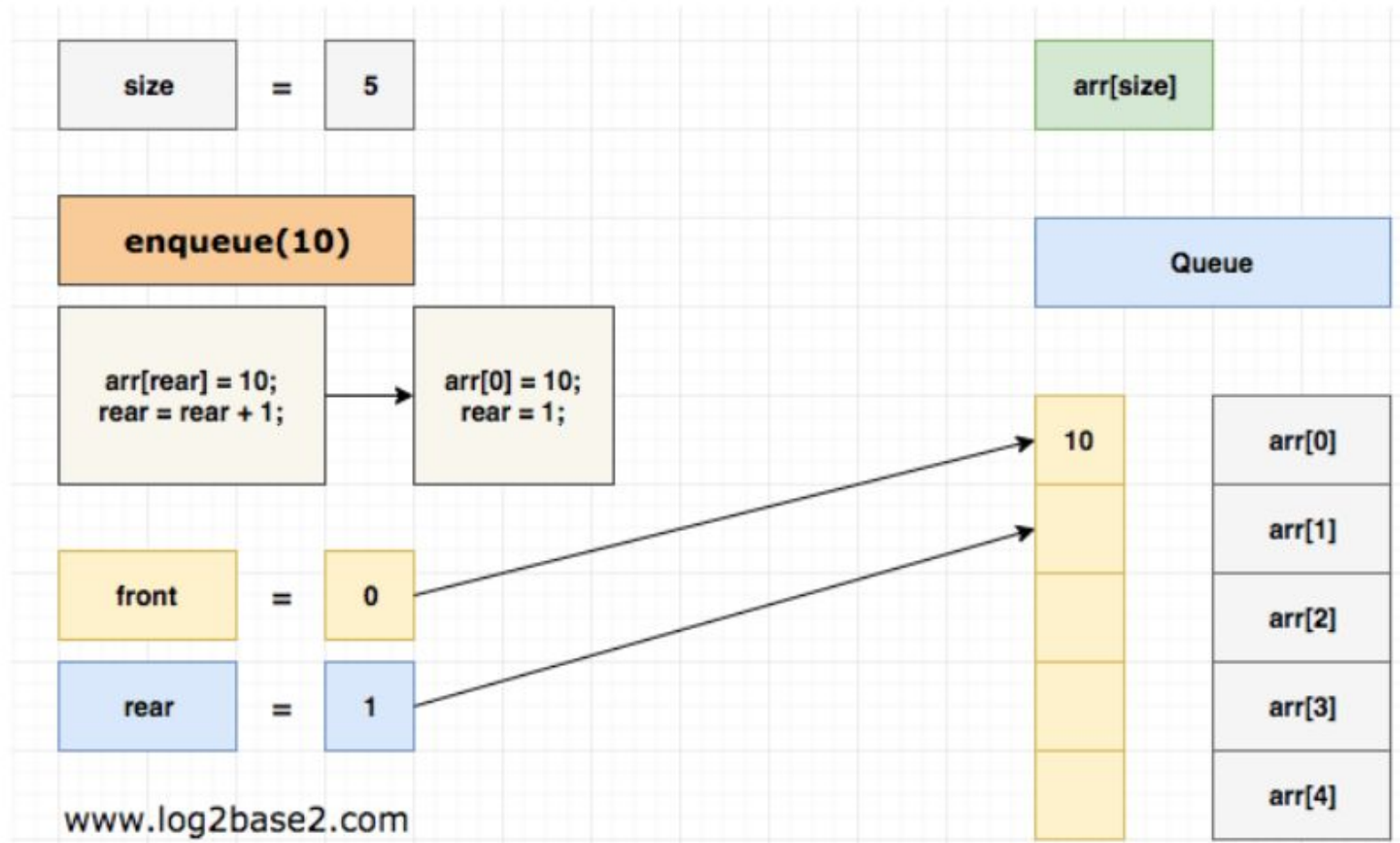
front = 0

Queue

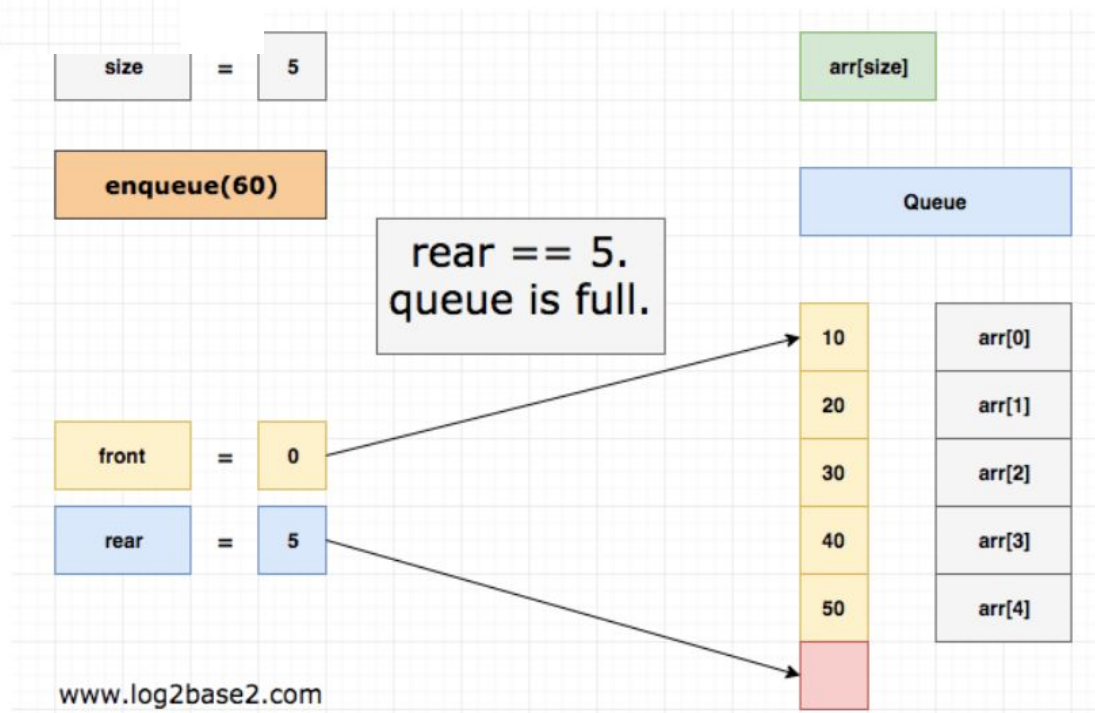
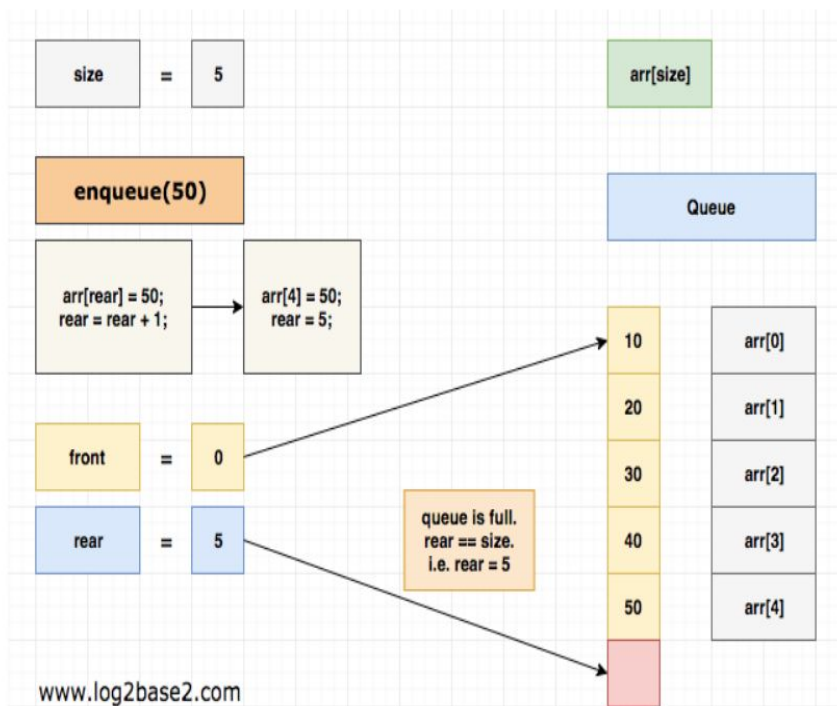
rear = 0



i) enqueue 10



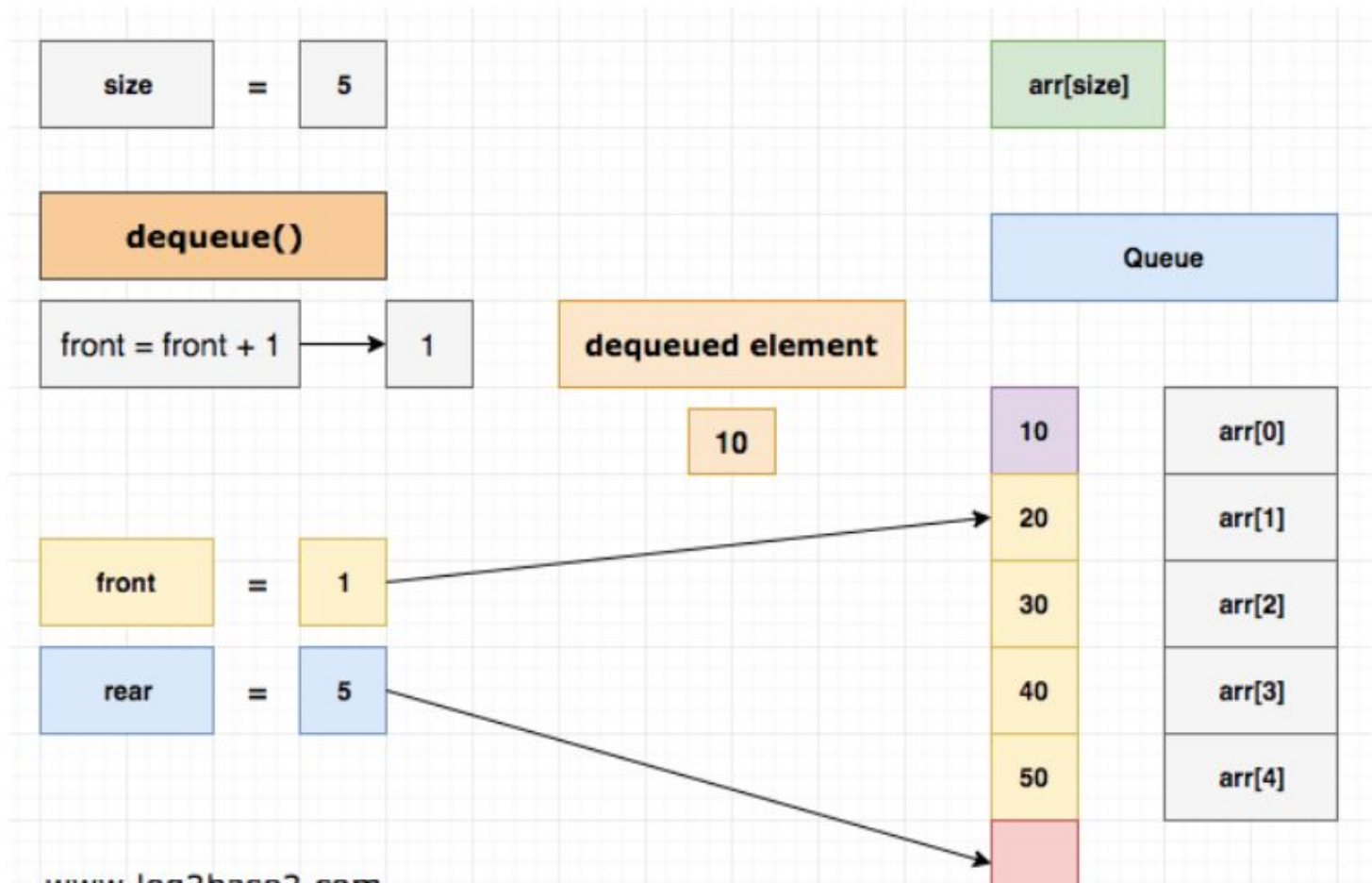
v) enqueue 50



```
int isQueueFull()  
{  
  if(rear == size)  
    return 1;  
  return -1;  
} //adds element at the end of the queue
```

```
void enqueue(int val)  
{  
  if(isQueueFull() == 1)  
    printf("Queue is Full\n");  
  else  
  {  
    arr[rear] = val; rear++;  
  }  
}
```

i) dequeue



```
int isQueueEmpty()
```

```
{
```

```
if(front == rear)
```

```
    return 1;
```

```
    return -1;
```

```
} //removes the current beginning element from the queue.
```

```
void dequeue()
```

```
{
```

```
if(isQueueEmpty() == 1)
```

```
    printf("Queue is Empty.\n");
```

```
else
```

```
{
```

```
    printf("Dequeued element = %d\n",arr[front]);
```

```
    front++;
```

```
}
```

```
}
```

# Queue ADT

## **Objects:**

A finite ordered list with zero or more elements.

## **Methods:**

for all  $queue \in \text{Queue}$ ,  $item \in \text{element}$ ,  
 $\text{max\_queue\_size} \in \text{positive integer}$

```
abstract typedef <<eltype>> QUEUE(eltype);  
abstract empty(q)  
QUEUE(eltype) q;  
post condition  empty==(len(q)==0);
```

```
abstract eltype remove(q)  
QUEUE(eltype) q;  
precondition empty(q)==FALSE;  
postcondition  remove==first(q' );  
               q==sub(q' , 1, len(q')-1);
```

```
abstract insert(q,elt)  
QUEUE(eltype) q;  
eltype elt;  
postcondition  q==q' +<elt>;
```

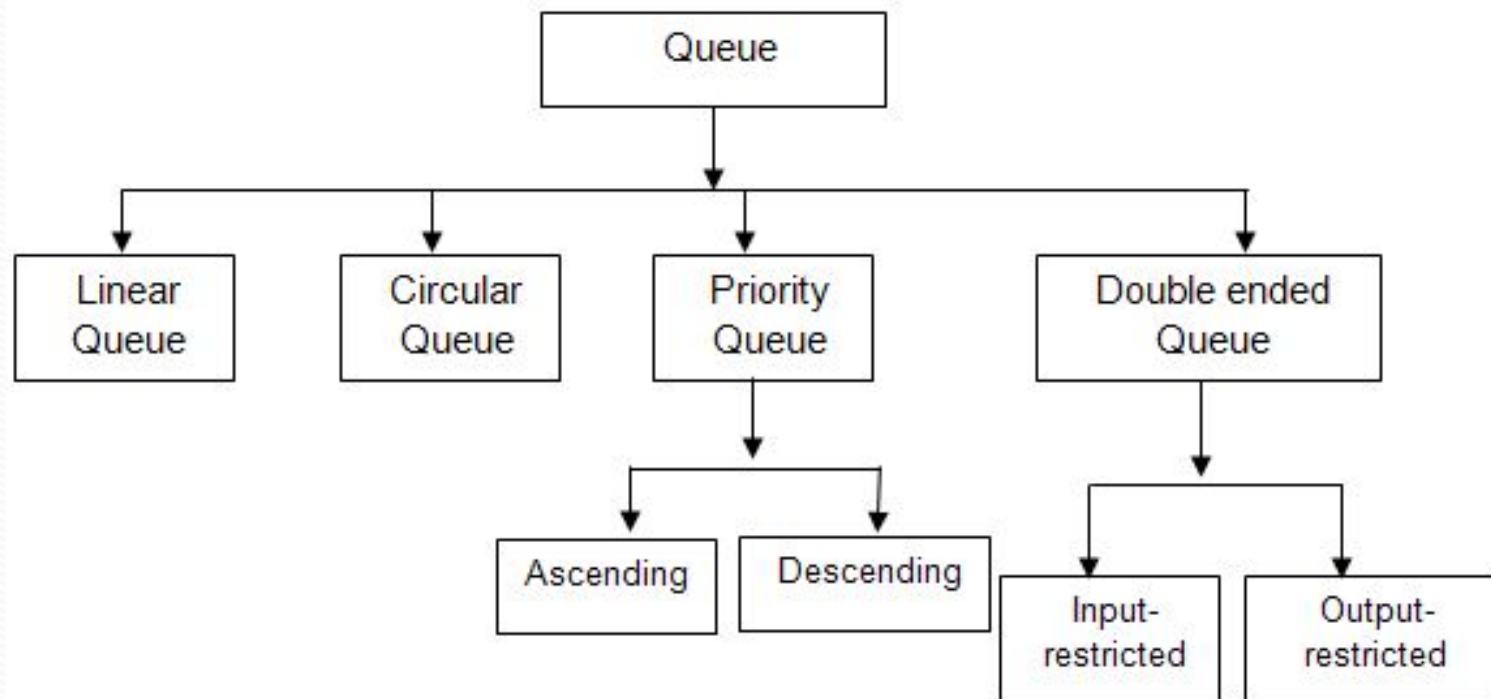


# Program for queue

- Simple queue : [que arr simple.c](#)
- Prg using array : [que arr.c](#)
- Program using str : [que str.c](#)

# QUEUE VARIANTS

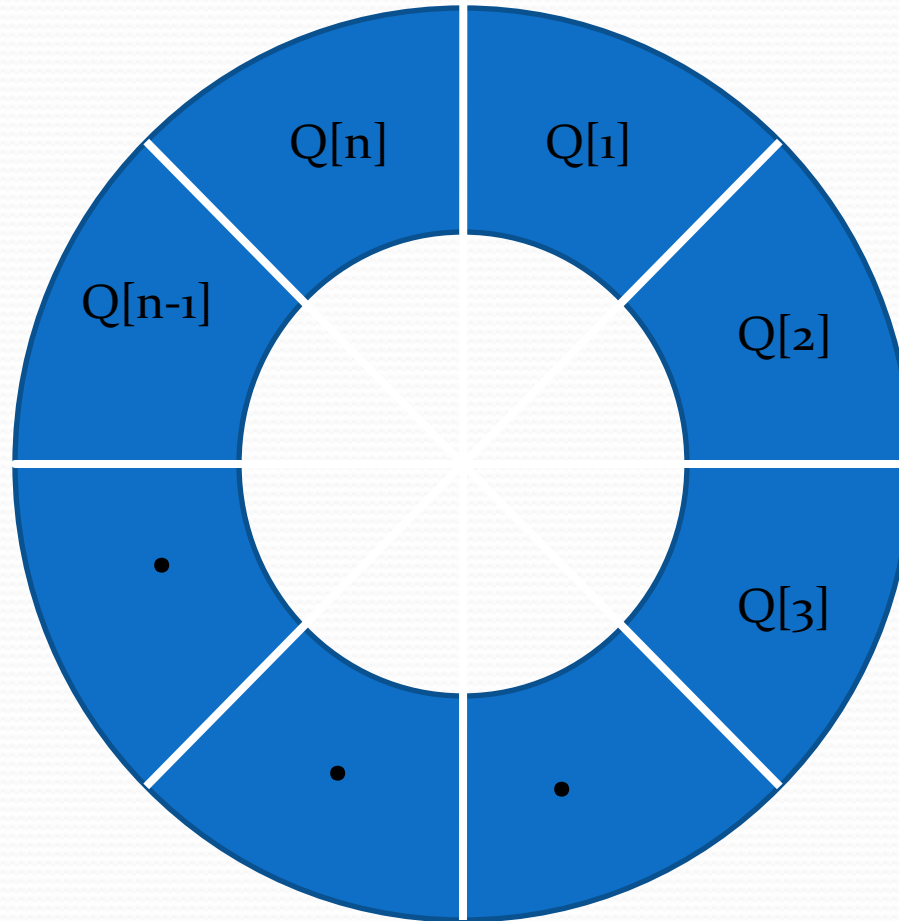
# Classification of Queues



# Circular queue

- Let we have an array named  $Q$ , that contains  $n$  element in which  $Q[1]$  comes after  $Q[n]$  in the array.
- When this technique is used to construct a queue is called circular queue.
- In other word we can say that a queue is called circular when the last room comes just before the first room.

# Circular queue....



# Circular queue

- A circular queue is a queue in which the element of a queue can be stored efficiently in an array so that end of queue is followed by the front of queue.
- Various operations of the circular queues are:
  - **Insert rear**
  - **Delete front**
  - **Display**

# Insert at rear

- **Check for overflow**
- **Insert item**
- **Update count**

```
void insert_rear()
{
    //insertion
    if((front==0 && rear==MAX) || rear==front+1)
        //Overflow
        if(front==-1) //empty
            if(rear==max-1 && front !=0)
                rear=0
            else
                rear++
        }
}
```

# Delete from the front end

- Check for under flow
- Delete item
- Update count

```
if(front==-1)// if(count == 0)
{
    printf("Under flow of queue\n");
}
if(front==rear)
{
    printf("The deleted element is %d\n",q[f]);
    front=rear=-1
    count--;
```



# Display queue contents

- Check for underflow
- Display the contents

```
int i,j;  
    if(front==-1) //if(count == 0){  
        printf("Under flow of queue\n");  }  
    else {  
        printf("contents of queue are\n");  
        i=f;  
        for(j=1;j<=count;j++) {  
            printf("%d ",q[i]);  
            i=(i+1)%MAX;  }  
        printf("\n");
```

# Queue cont....


- In a circular queue when  $\text{rear} = n$ , if we insert an element then this element is assigned to  $q[1]$  instead of increasing rear to  $n+1$ .
- Suppose queue contains only one element that is  $\text{front} = \text{rear} \neq 0$  and suppose that the element is removed then the front and rear pointers are now assigned '0' to indicate that the queue is EMPTY.

# Application of queue

- An e.g. of queue is time sharing computer system where many users share the system simultaneously.
- The procedure, which is used to design such type of system, is Round Robin Technique.
- The railway reservation counter is also an example of queue where the people collect their tickets on FIFO or FCFS[First Come First Serve] based.
- Program : [cir\\_que.c](#),

# Priority queue

- Its a special type of data structure in which items can be inserted or deleted based on the priority.
- Types of priority Queues:
  - **Ascending priority queue**
  - **Descending priority queue**

- 
- The rules are:
    - An element of higher priority is processed before any element of lower priority.
    - Two elements with the same priority are processed according to the order in which they were added to the queue.

- Ascending priority queue:

- In ascending priority queue elements can be inserted in any order. But while deleting an element from the queue, only the smallest element is removed first.
- *ascpqinsert(ascpq,x)* inserts the element  $x$  into the queue *ascpq*.
- *ascpqdelete(ascpq)* removes the smallest element from the queue *ascpq* and returns its value to the user.
- The elements can be inserted at any position in the queue but only the smallest element can be removed from the queue.

- Descending priority queue:
  - In descending priority queue elements can be inserted in any order. But while deleting an element from the queue, only the largest element is removed first.
- *descpqinsert(descpq, x)* inserts the element *x* into the queue *descpq*.
- This is logically equivalent to the insert operation of *ascpqinsert(ascpq, x)*.
- *descpqdelete(descpq)* removes the largest element from the queue *descpq* and returns its value to the user.
- The operation *empty(queue)* applies for both types of priority queue and determines whether a priority queue is empty.

```
void insert(struct queue *qptra,int ele)
{
    if(qptra->rear==SIZE-1)
        printf("\n Queue overflow");
    else
        qptra->item[++(qptra->rear)]=ele;
}
```



```
int qremove(struct queue *qptr)
{
    if(empty(qptr))
    {
        printf("Queue underflow");
        return(-1);
    }
    else
        return(qptr->item[(qptr->front)++]);
}
```

```
void display(struct queue *qptr)
{
    int i;
    if(empty(qptr))
        printf("\n Queue is empty \n");
    else
    {
        printf("\n Queue contents \n");
        for(i=qptr->front;i<=qptr->rear;i++)
            printf("\t %d",qptr->item[i]);
    }
}
```

```
int empty(struct queue *qptr)
{
    if(qptr->front>qptr->rear)
        return(1);
    else
        return(0);
}
```

# Advantages of using arrays

- Arrays are simple to understand and use
- Data accessing is faster
  - Data can be accessed very efficiently just by specifying the array name and the index of the item (  $A[i]$  )
  - Time required to access  $A[0]$  is same as  $A[1000]$

# Disadvantages of arrays

- The size of the array is fixed
  - Fixed amount of memory is allocated before the start of execution.
  - Most of the time we cannot predict the memory space
  - Less memory is allocated and more memory is required during the execution.
- Array items are stored contiguously
  - Some times continuous memory location may not be available
- Insertion and deletion operations involving array is tedious job
  - If an array consists of more than 100 elements and we want to insert an element at  $i$ th position, then all the elements need to move one position to provide the space in the memory. Similarly for delete operation.

## Double ended Queue (De queue)

- It is a special type of data structure in which insertions are done from both ends and deletions are done at both ends.
- Different operations of queue are
  - **Insert an item from front end**
  - **Insert an item from rear end**
  - **Delete an item from front end**
  - **Delete an item from rear end**
  - **Display the contents of queue**
  - **deq.c**

- Two types of deque depending upon the restriction to perform insertion or deletion operations at the two ends. They are
  - 1. Input restricted deque : allows insertion at only 1 end, rear end, but allows deletion at both ends, rear and front end of the lists. Operation performed are add element at rear and delete at front and rear
  - Output-restricted deque : allows deletion at only one end, front end, but allows insertion at both ends, rear and front ends, of the lists. Operation performed are add element at rear, front and delete at front

# Applications of queue

- 1. Round robin techniques for processor scheduling is implemented using queue(for resource sharing b/w multiple consumers)
- 2. Printer server routines (in drivers) are designed using queues( data transferred asynchronously)
- 3. All types of customer service software (like Railway/Air ticket reservation) are designed using queue to give proper service to the customers.



# Assignment 2

- Define recursion. Write a recursion functions for the following:
  - i. Factorial of a number.
  - ii. Tower of Hanoi.
- What is queue? Explain the various types of queue and operations performed on it.
- Explain the Limitations of array implementation using Linked List.
- Write a program to implement queue using singly linked list.
- Differentiate :
  - i) static and dynamic memory allocation
  - ii) `getnode()` and `freenode()` operations