DATA STRUCTURE

All Notes Are Verified by NPTL and Published By

WWW.ASKTOHOW.COM

IIT Guwahati
NPTL

C.E.O of ASKTOHOW
Deepak

Definition

A **data structure** is said to be ***linear*** if its elements form a sequence or a **linear** list.

Examples:

- Array
- Linked List
- Stacks
- Queues

Operations on linear Data Structures

- *Traversal* : Visit every part of the **data structure**
- *Search* : Traversal through the data structure for a given element
- *Insertion* : Adding new elements to the **data structure**
- *Deletion* : Removing an element from the **data structure**.
- *Sorting* : Rearranging the elements in some type of order(e.g Increasing or Decreasing)
- *Merging* : Combining two similar **data** structures into one

Introduction:

1. Stack is basically a data object
2. The operational semantic (meaning) of stack is LIFO i.e. last in first out

   Definition : It is an ordered list of elements n , such that n>0 in which all insertions and deletions are made at one end called the top.

   Primary operations defined on a stack:
   1. PUSH : add an element at the top of the list.
   2. POP : remove the at the top of the list.
   3. Also "IsEmpty()" and IsFull" function, which tests whether a stack is empty or full respectively.

   Example :

1. **Practical daily life :** a pile of heavy books kept in a vertical box,dishes kept one on top of another
2. **In computer world :** In processing of subroutine calls and returns ; there is an explicit use of stack of return addresses.
   Also in evaluation of arithmetic expressions , stack is used.

   Large number of stacks can be expressed using a single one dimensional stack only. Such an array is called a multiple stack array.

   Test Your Skills : Prob . 1 Prob . 2

Algorithms

Push **(item,array , n, top)**

```
{
    If ( n> = top)
      Then print "Stack is full" ;
    Else
      {
        top = top + 1;
        array[top] = item ;
      }
}
```

Pop **(item,array,top)**

```
{
    if ( top<= 0)
    Then print " stack is empty".
    Else
    {
        item = array[top];
        top = top – 1;
    }
}
```

Visual idea of Push Operation

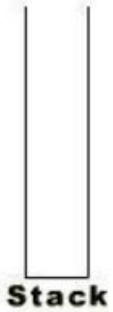
Visual idea of Pop Operation

Working of a Stack

11

Stack is implemented here as a one dimensional array of size 7

Addition of element to Stack

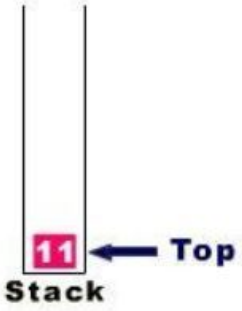


Stack

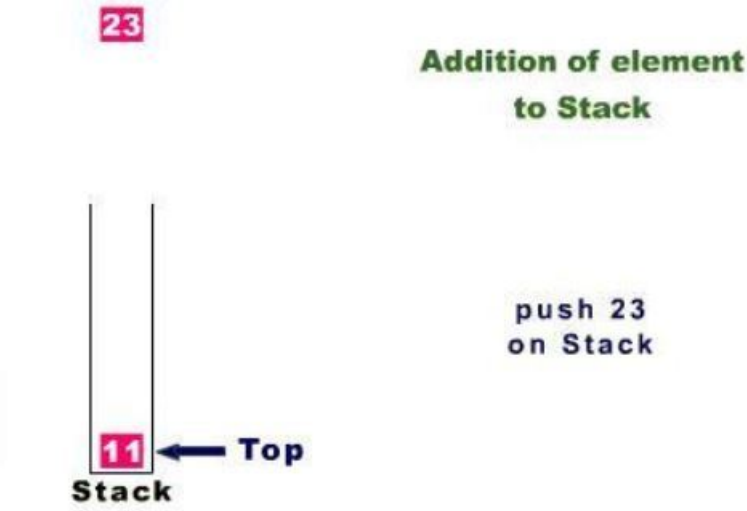
push 11 on Stack

Working of a Stack

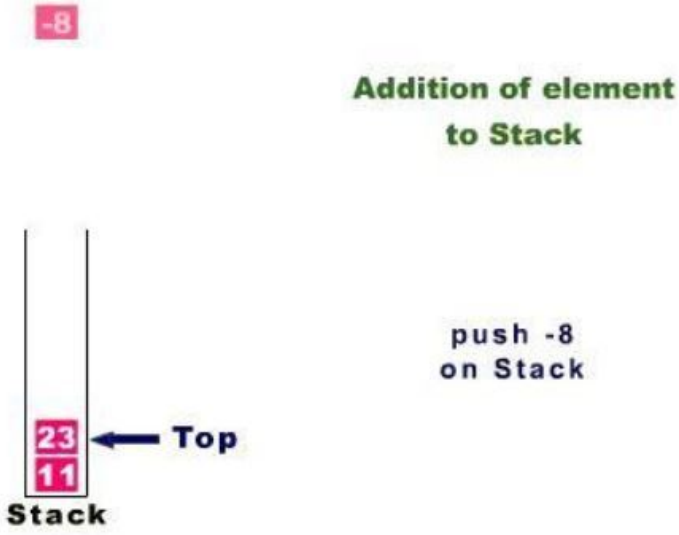
Addition of element to Stack



Working of a Stack



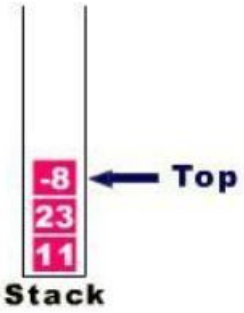
Working of a Stack



Working of a Stack

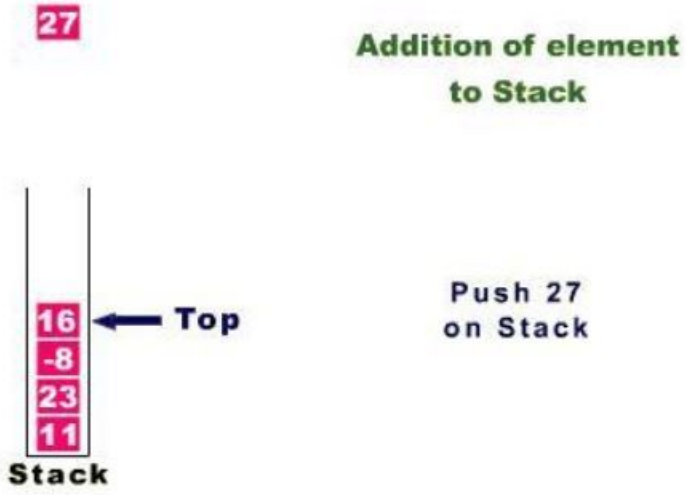
16

Addition of element to Stack

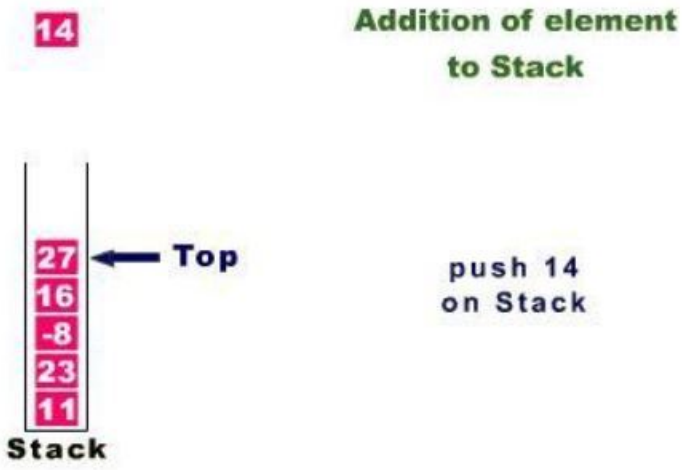


push 16
on Stack

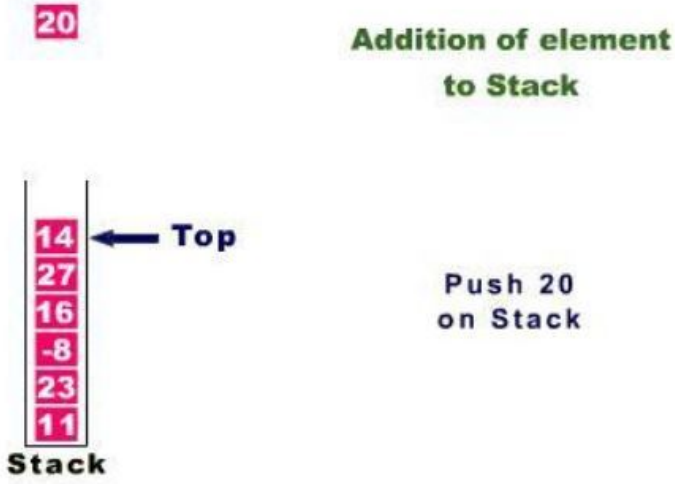
Working of a Stack



Working of a Stack



Working of a Stack



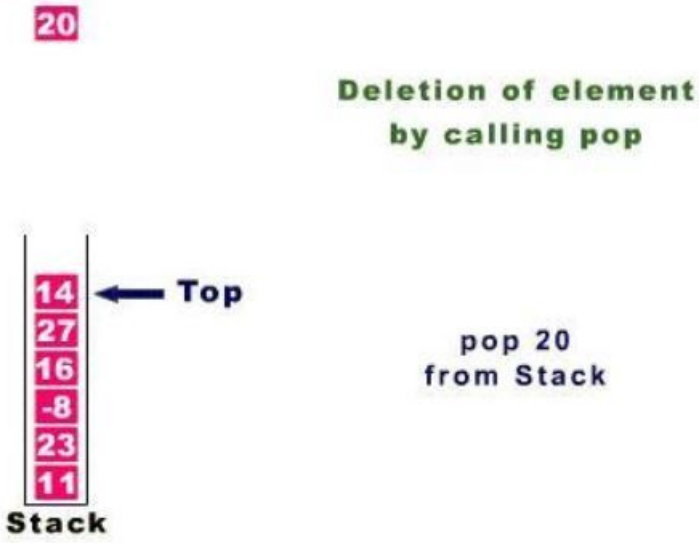
Working of a Stack

Addition of element to Stack

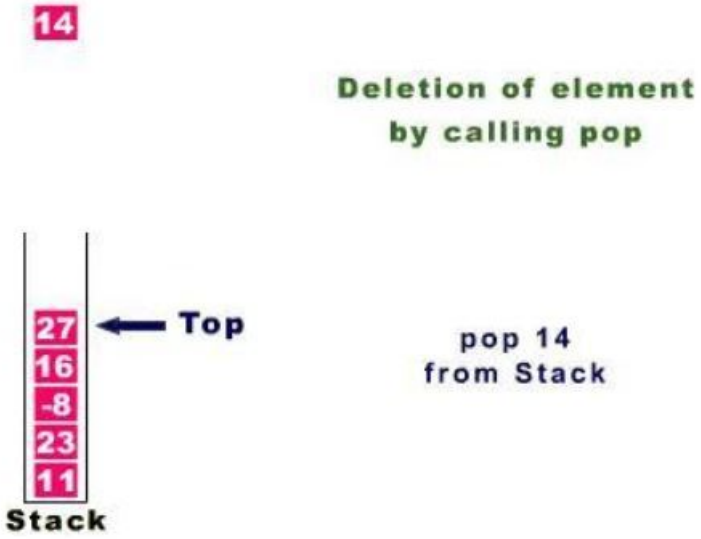
Working of a Stack

Deletion of element by calling pop

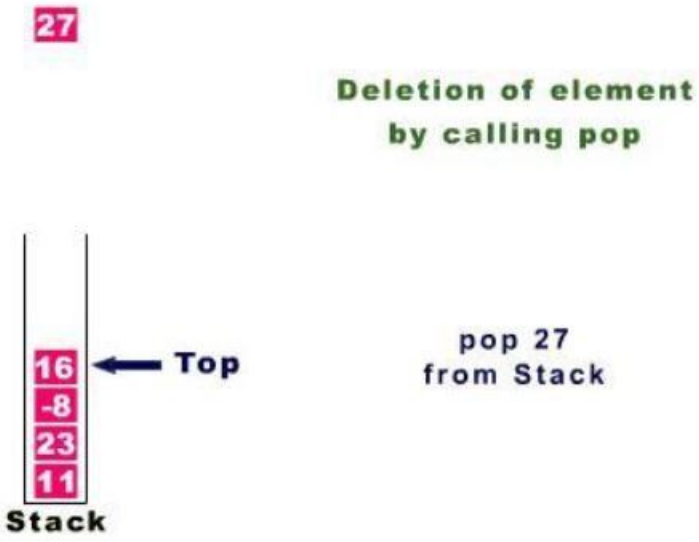
Working of a Stack



Working of a Stack



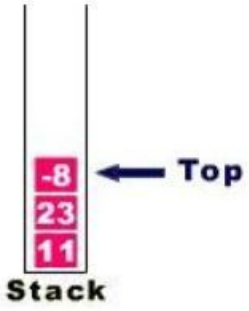
Working of a Stack



Working of a Stack

16

Deletion of element by calling pop

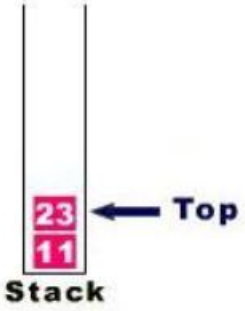


pop 16
from Stack

Working of a Stack

-8

Deletion of element by calling pop

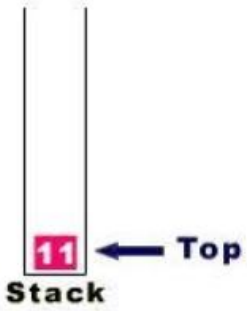


pop -8
from Stack

Working of a Stack

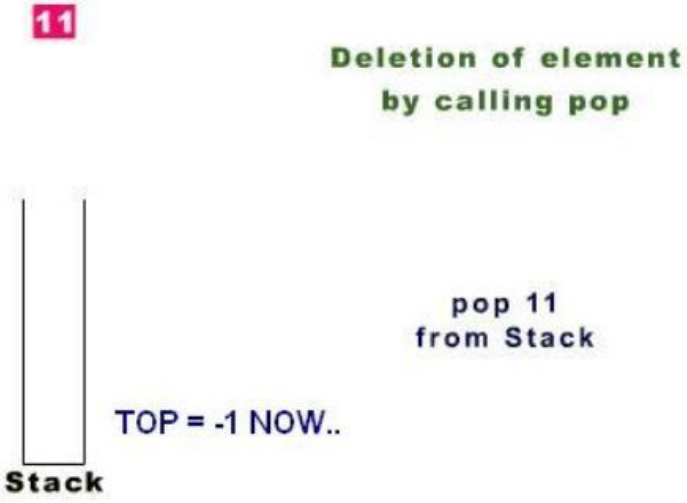
23

Deletion of element by calling pop



pop 23
from Stack

Working of a Stack



Arithmetic expressions are expressed as combinations of:

1. Operands
2. Operators (arithmetic, Boolean, relational operators)

Various rules have been formulated to specify the order of evaluation of combination of operators in any expression.

The arithmetic expressions are expressed in 3 different notations:

1. *Infix* :

- In this if the operator is binary; the operator is between the 2 operands.
- And if the operator is unary, it precedes the operand.

2. *Prefix* :

- In this notation for the case of binary operators, the operator precedes both the operands.
- Simple algorithm using stack can be used to evaluate the final answer.

3. *Postfix* :

- In this notation for the case of binary operators, the operator is after both the corresponding operands.
- Simple algorithm using stack can be used to evaluate the final answer.

Always remember that the order of appearance of operands does not change in any Notation. What changes is the position of operators working on those operands.

RULES FOR EVALUATION OF ANY EXPRESSION :

An expression can be interpreted in many different ways if parentheses are not mentioned in the expression.

- For example the below given expression can be interpreted in many different ways:
- Hence we specify some basic rules for evaluation of any expression :

A priority table is specified for the various type of operators being used:

PRIORITY LEVEL	OPERATORS
6	** ; unary - ; unary +
5	* ; /
4	+ ; -
3	< ; > ; <= ; >= ; !> ; !< ; !=
2	Logical and operation
1	Logical or operation

Algorithm for evaluation of an expression E which is in prefix notation :

- We assume that the given prefix notation starts with IsEmpty() .
- If number of symbols = n in any infix expression then number of operations performed = some constant times n.
- here *next token* function gives us the next occurring element in the expression in a left to right scan.
- The *PUSH* function adds element x to stack Q which is of maximum length n

```
Evaluate (E)
{
    Top = 0;
    While (1)
        {
          x= next token (E)
          If (x = = infinity)
            {
              Print value of stack [top] as the
              output of the expression
            }
          Else
          {
            If (x = = operand)
                PUSH (Q, top, n, x);
            If (x = = operator)
              {
                Pop correct number of operands according to the
                operator (unary/binary) and then perform the
                operation and store result onto the stack
              }
          }
}
```

- Here only one single one-dimensional array (Q) is used to store multiple stacks.
- B (i) denotes one position less than the position in Q for bottommost element of stacks i.
- T (i) denotes the top most element of stack i.
- m denotes the maximum size of the array being used.
- n denotes the number of stacks.

We also assume that equal segments of array will be used for each stack.

Initially let B (i) = T (i) = [m/n]*(i-1) where 1<= i <= n.

Again we can have ***push*** or ***pop*** operations, which can be performed on each of these stacks .

Algorithms

Push **(i, x)**

```
{
    If (((i<n) && (T(i)==B(i+1)) ||
    ((i>=n) && (T(i) ==m)));

      Then call STACK_FULL ;
    Else
      {
        T(i) = T(i) + 1;
        Q[T(i)] = x ;
      }
}
```

Pop **(i,x)**

```
{
    if ( T(i) = = B(i) )
    Then print that the stack is empty.
    Else
    {
        x = Q[T(i)];
        T[i] = T[i] – 1;
    }
}
```

ALGORITHM TO BE APPLIED WHEN T [i] = = B [i] CONDITION IS ENCOUNTERED WHILE DOING PUSH OPERATION.

STACK_FULL

{

1. Find j such that i < j <= n and there is a free space between stack j and stack (j +1).
if such a j exist , then move stack i+1 , i+2 ,.....till j one position to the right and hence create space for element between stack i and i +1.

2. Else
Find j such that 1 <= j < i and there is a free space between stack j and stacK(j +1).
if such a j exist , then move stack j+1 , j+2 ,.....till i one position to the left and hence create space for element between stack i and i +1 .

3. Else
If none of above is possible then there is no space left out in the one-dimensional array used hence print no space for push operation.

}

Introduction:

1. It is basically a data object
2. Theoperational semantic of queue is FIFO i.e. first in first out

Definition :

It is an ordered list of elements n , such that n>0 in which all deletions are made at one end called the front end and all insertions at the other end called the rear end .

Primary operations defined on a Queue:

1. EnQueue : This is used to add elements into the queue at the back end.
2. DeQueue : This is used to delete elements from a queue from the front end.
3. Also "IsEmpty()" and "IsFull()" can be defined to test whether the queue is Empty or full.

Example :

1. **PRACTICAL EXAMPLE** : A line at a ticket counter for buying tickets operates on above rules

2. **IN COMPUTER WORLD** : In a batch processing system, jobs are queued up for processing.

Circular queue :
In a queue if the array elements can be accessed in a circular fashion the queue is a circular queue.

Priority queue :

Often the items added to a queue have a priority associated with them: this priority determines the order in which they exit the queue - highest priority items are removed first.

C ++ IMPLEMENTATION OF QUEUE USING CLASSES

Test your skills

Primary operations defined for a circular queue are :

1. **add_circular -** It is used for addition of elements to the circular queue.
2. **delete_circular -** It is used for deletion of elements from the queue.

We will see that in a circular queue , unlike static linear array implementation of the queue ; the memory is utilized more efficient in case of circular queue's.

The shortcoming of static linear that once rear points to n which is the max size of our array we cannot insert any more elements even if there is space in the queue is removed efficiently using a circular queue.

As in case of linear queue , we'll see that condition for zero elements still remains the same i.e.. rear=front

ALGORITHM FOR ADDITION AND DELETION OF ELEMENTS

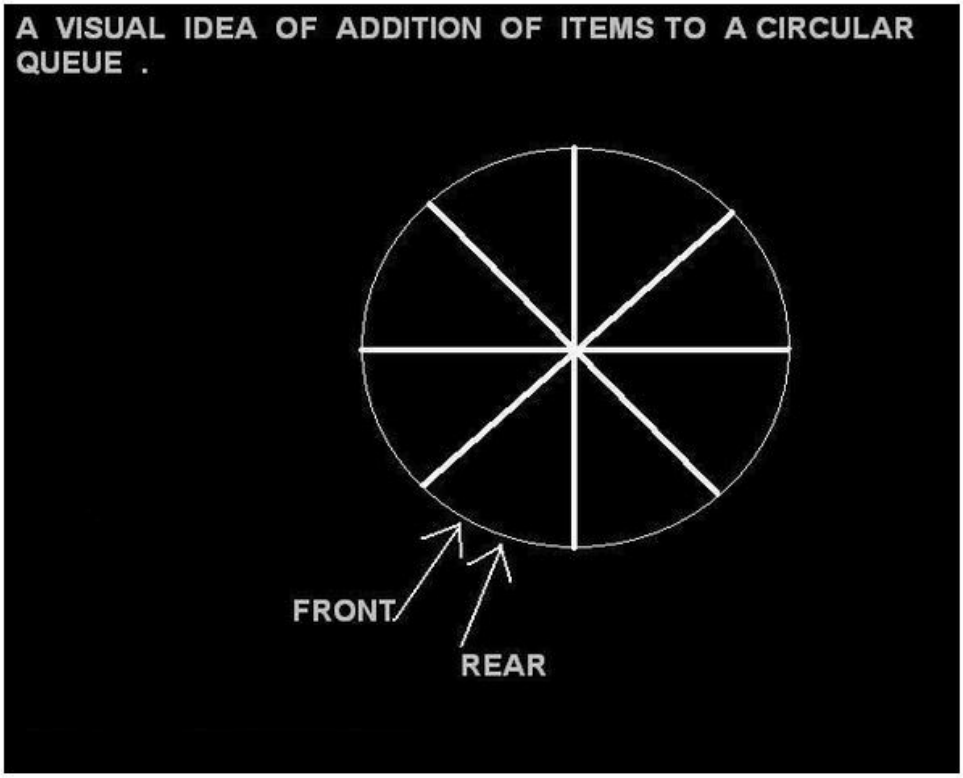
Data structures required for circular queue :

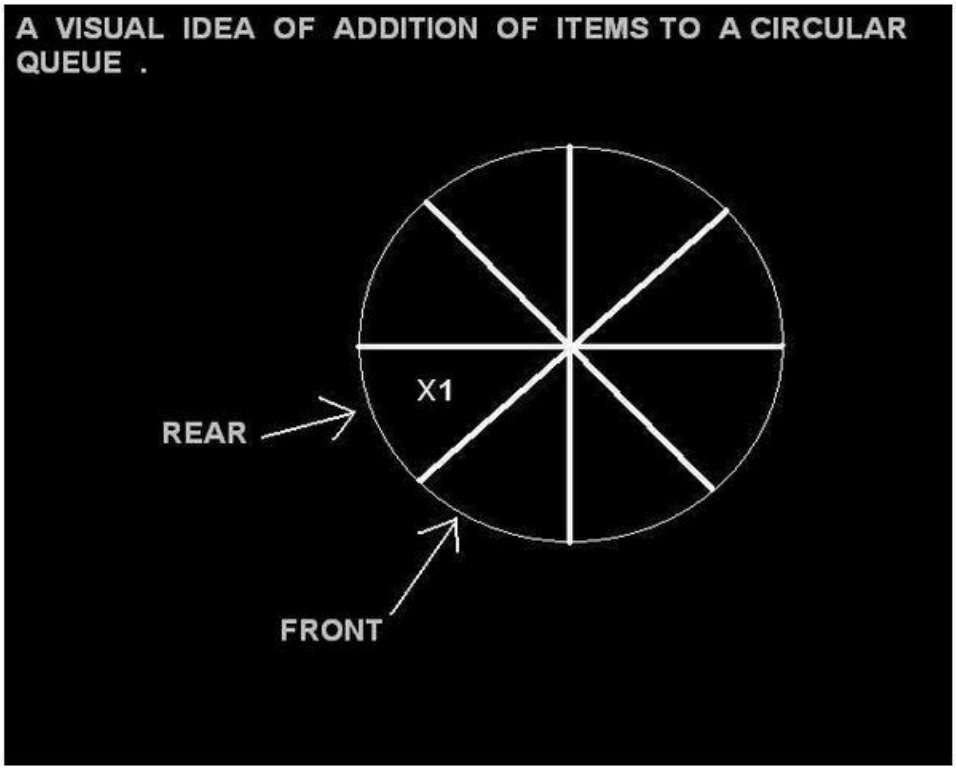
1. **front** counter which points to one position anticlockwise to the 1st element
2. **rear** counter which points to the last element in the queue
3. an **array** to represent the queue

```
add _ circular ( item,queue,rear,front)
    {
        rear=(rear+1)mod n;
            if (front == rear )
                then print " queue is full "
            else
                {
                   queue [rear]=item;
                }
    }
```

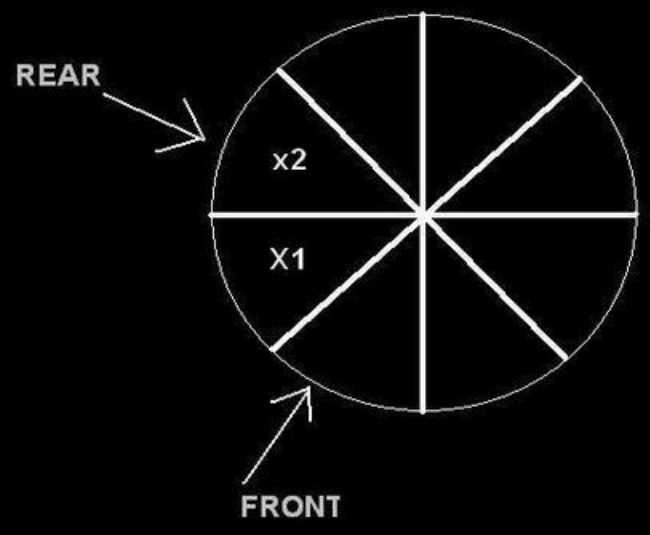
Visual Idea of Add_Circular Operation

A VISUAL IDEA OF ADDITION OF ITEMS TO A CIRCULAR QUEUE .

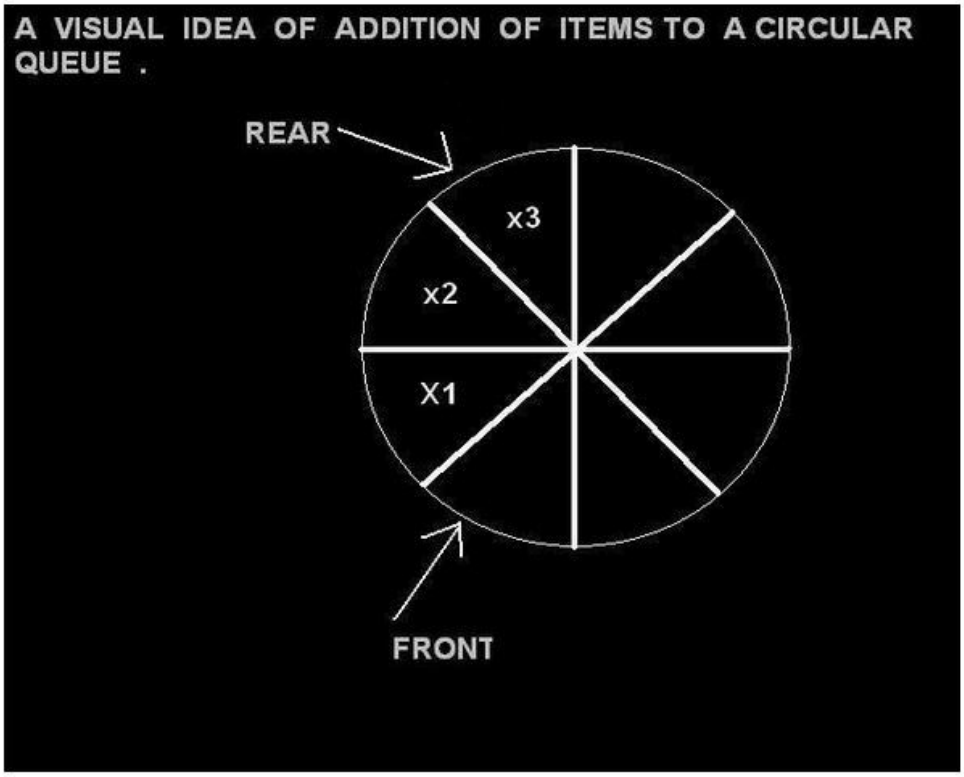



A VISUAL IDEA OF ADDITION OF ITEMS TO A CIRCULAR QUEUE .
X1
REAR
FRONT

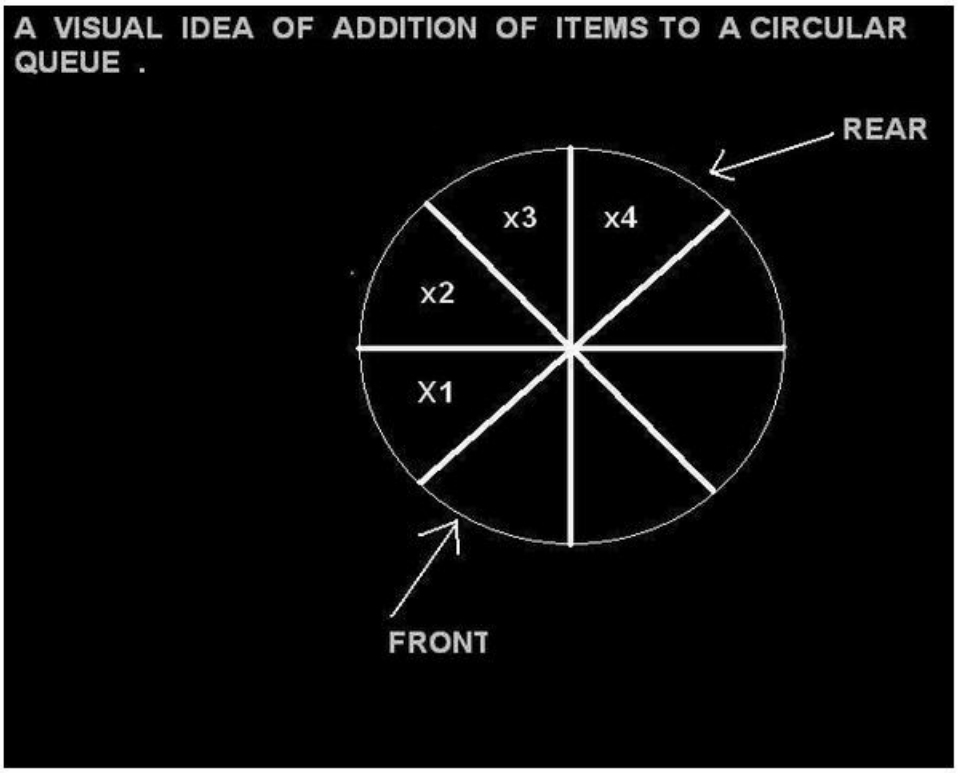
A VISUAL IDEA OF ADDITION OF ITEMS TO A CIRCULAR QUEUE .



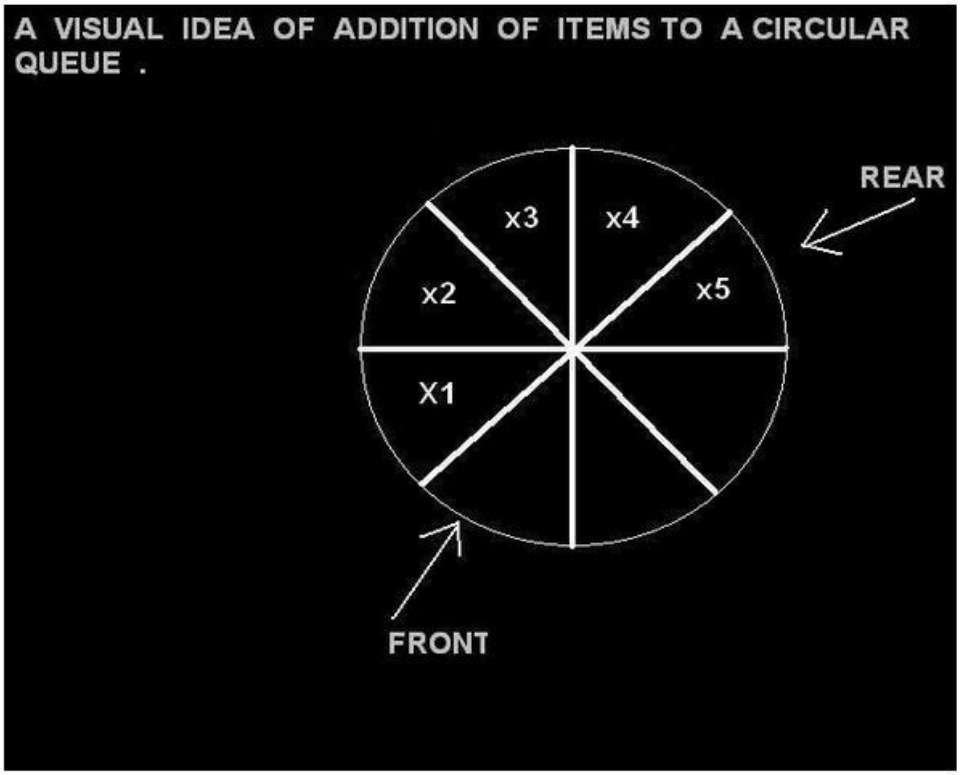
A VISUAL IDEA OF ADDITION OF ITEMS TO A CIRCULAR QUEUE .



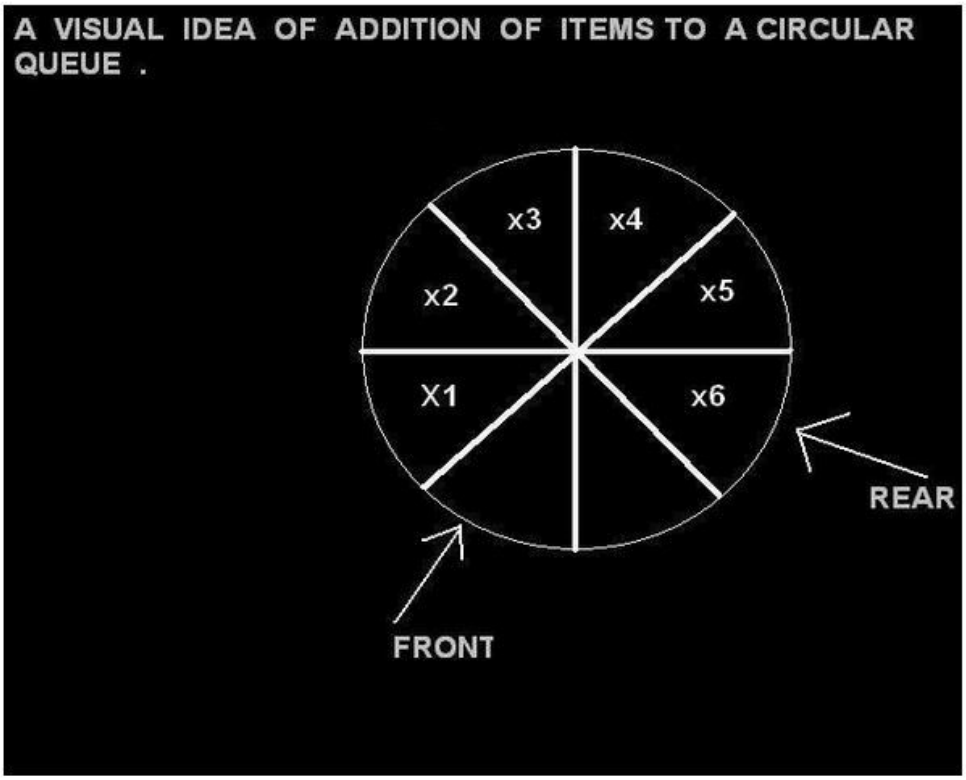
A VISUAL IDEA OF ADDITION OF ITEMS TO A CIRCULAR QUEUE .



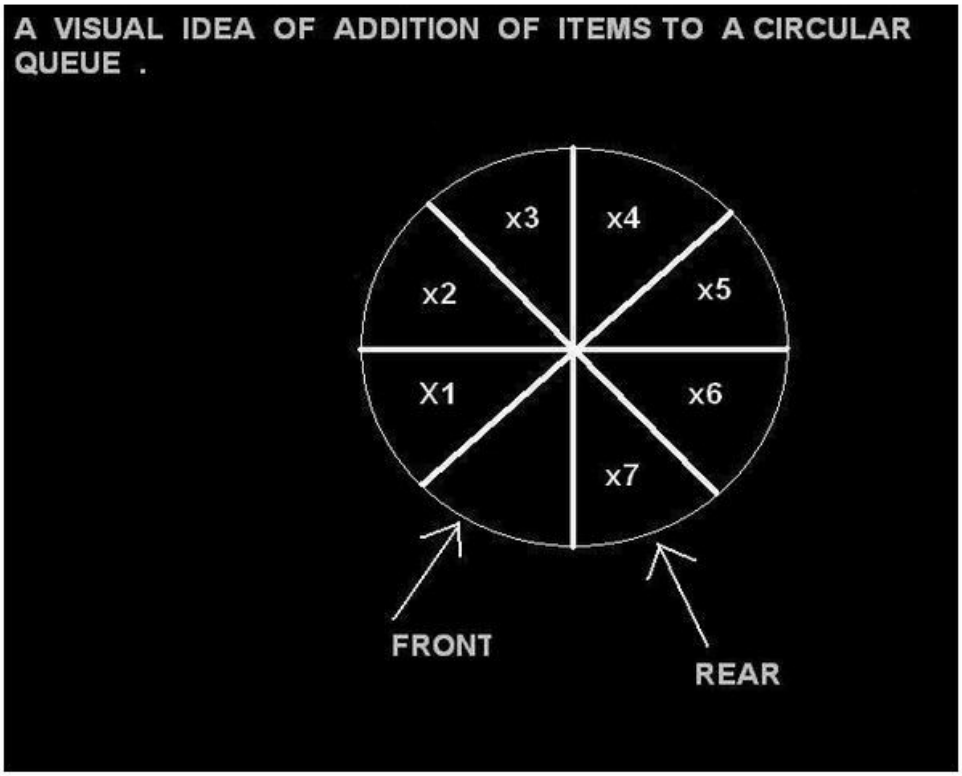
A VISUAL IDEA OF ADDITION OF ITEMS TO A CIRCULAR QUEUE .



A VISUAL IDEA OF ADDITION OF ITEMS TO A CIRCULAR QUEUE .



A VISUAL IDEA OF ADDITION OF ITEMS TO A CIRCULAR QUEUE .

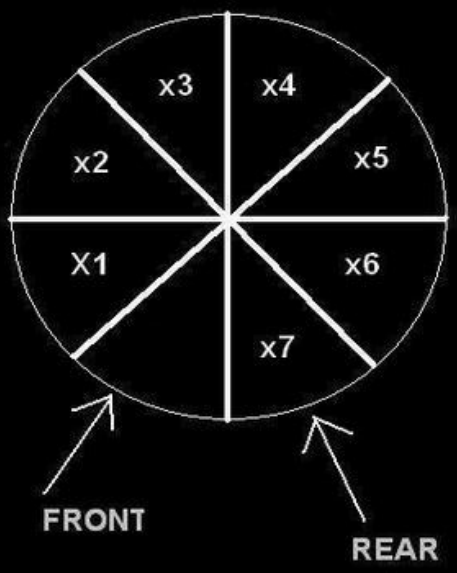


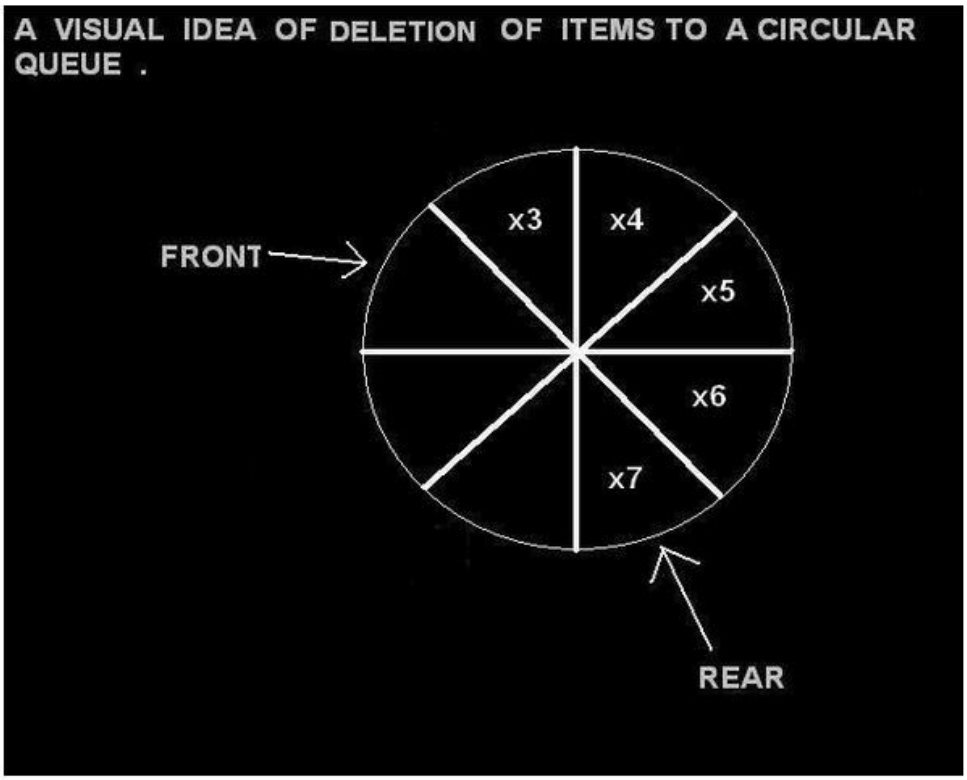
delete operation :

```
delete_circular (item,queue,rear,front)
  {
      if (front = = rear)
          print ("queue is empty");
      else
        {
            front= front+1;
            item= queue[fromt];
        }
  }
```

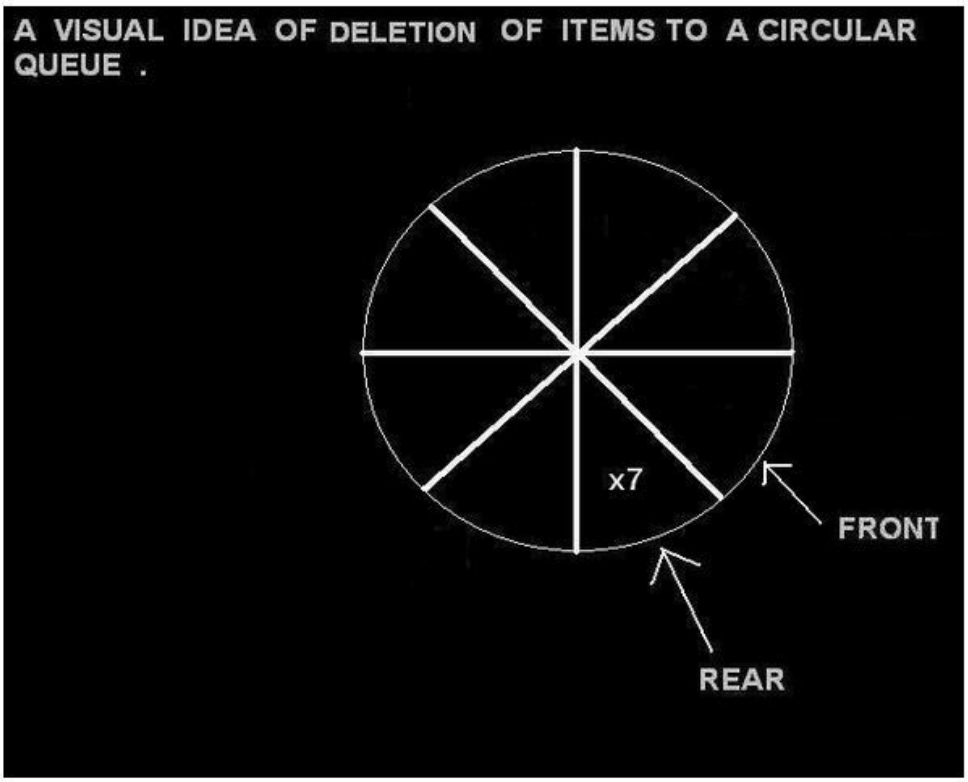
Visual Idea of Delete_Circular Operation

A VISUAL IDEA OF DELETION OF ITEMS TO A CIRCULAR QUEUE .

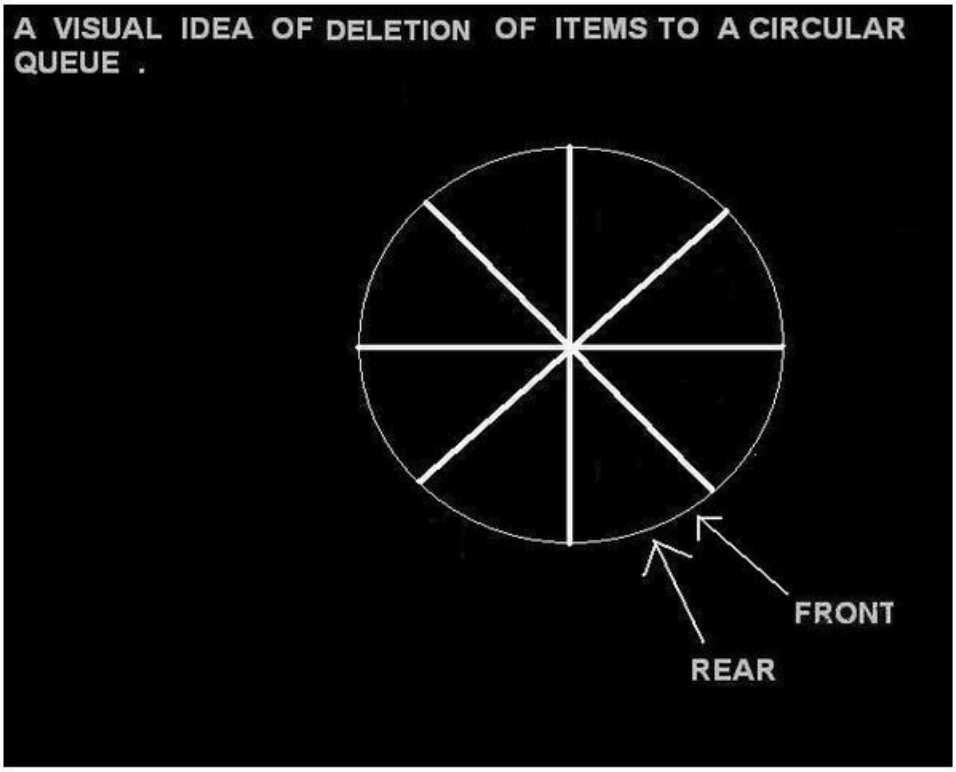



A VISUAL IDEA OF DELETION OF ITEMS TO A CIRCULAR QUEUE .
FRONT
x3
x4
x5
x6
x7
REAR

A VISUAL IDEA OF DELETION OF ITEMS TO A CIRCULAR QUEUE .



A VISUAL IDEA OF DELETION OF ITEMS TO A CIRCULAR QUEUE .



ALGORITHM FOR ADDITION AND DELETION OF ITEMS IN A QUEUE

note : addition is done only at the rear end of a queue like in a ticket counter line ....

add (item ,queue , n ,rear)

```
{
    if (rear==n)
        then print " queue is full "
    else
        {
            rear=rear+1;
            queue [rear]=item;
        }
}
```

Tabular View of ADD operation

ALGORITHM FOR ADDITION AND DELETION OF ITEMS IN A QUEUE

note : deletion is allowed only at the front end of the queue

delete (item , queue , rear , front)

```
{
    if (rear==front)
        then print "queue is empty";
    else
        {
            item = queue [front] ;
            front=front+1 ;
        }
}
```

Queues are dynamic collections which have some concept of order. This can be either based on order of entry into the queue - giving us First-In-First-Out (FIFO) or Last-In-First-Out (LIFO) queues. Both of these can be built with linked lists: the simplest "add-to-head" implementation of a linked list gives LIFO behavior. A minor modification - adding a tail pointer and adjusting the addition method implementation - will produce a FIFO queue.

Performance

A straightforward analysis shows that for both these cases, the time needed to add or delete an item is constant and *independent of the number of items in the queue* . Thus we class both addition and deletion as an ***O* (1)** operation. For any given real machine + operating system + language combination, addition may take ***c 1*** seconds and deletion ***c 2*** seconds, but we aren't interested in the value of the constant, it will vary from machine to machine, language to language, *etc* . The key point is that the time is not dependent on ***n*** - producing ***O* (1)** algorithms.

Once we have written an ***O* (1)** method, there is generally little more that we can do from an algorithmic point of view. Occasionally, a better approach may produce a lower constant time. Often, enhancing our compiler, run-time system, machine, *etc* will produce some significant improvement. However ***O* (1)** methods are already very fast, and it's unlikely that effort expended in improving such a method will produce much real gain!

PRIORITY QUEUE :

Often the items added to a queue have a **priority** associated with them: this priority determines the order in which they exit the queue - highest priority items are removed first.

This situation arises often in process control systems. Imagine the operator's console in a large automated factory. It receives many routine messages from all parts of the system: they are assigned a low priority because they just report the normal functioning of the system - they update various parts of the operator's console display simply so that there is some confirmation that there are no problems. It will make little difference if they are delayed or lost.

However, occasionally something breaks or fails and alarm messages are sent. These have high priority because some action is required to fix the problem (even if it is mass evacuation because nothing can stop the imminent explosion!).
Typically such a system will be composed of many small units, one of which will be a buffer for messages received by the operator's console. The communications system places messages in the buffer so that communications links can be freed for further messages while the console software is processing the message. The console software extracts messages from the buffer and updates appropriate parts of the display system. Obviously we want to sort messages on their priority so that we can ensure that the alarms are processed immediately and not delayed behind a few thousand routine messages while the plant is about to explode.

As we have seen, we could use a tree structure - which generally provides $O(\log n)$ performance for both insertion and deletion. Unfortunately, if the tree becomes unbalanced, performance will degrade to $O(n)$ in pathological cases. This will probably not be acceptable when dealing with dangerous industrial processes, nuclear reactors, flight control systems and other *life-critical* systems.

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

#define MAX 5 // MAXIMUM CONTENTS IN
QUEUE

class queue
{
private:
   int t[MAX];
   int al; // Addition End
   int dl; // Deletion End
public:
queue()
{
   dl=-1;
   al=-1;
}

void del()
{
int tmp;
if(dl==-1)
{

cout<<"Queue is Empty";
}
else
```

```
void add(int item)
{
   if(dl==-1 && al==-1)
   {
      dl++;
      al++;
   }
else
{
      al++;
      if(al==MAX)
   {
         cout<<"Queue is Full\n";
         al--;
         return;
      }
   }
   t[al]=item;

}

void display()
{
   if(dl!=-1)
{
```

```
    for(int j=0;j<=al;j++)
    {
      if((j+1)<=al)
      {
        tmp=t[j+1];
        t[j]=tmp;
      }
      else
      {
        al--;

      if(al==-1)
        dl=-1;
      else
        dl=0;
      }
    }
}
}
```

```
      cout<<t[i]<<" ";
  }
  else
      cout<<"EMPTY";
  }

  };
```

```
void main()
  {
queue a;
int data[5]={32,23,45,99,24};

   cout<<"Queue before adding Elements: ";
    a.display();
    cout<<endl<<endl;

      for(int i=0;i<5;i++)
       {
        a.add(data[i]);
      cout<<"Addition Number : "<<(i+1)<<" : ";
      a.display();
      cout<<endl;
       }
    cout<<endl;
    cout<<"Queue after adding Elements: ";
    a.display();
    cout<<endl<<endl;

      for(int i=0;i<5;i++)
       {
       a.del();
        cout<<"Deletion Number : "<<(i+1)<<" : ";
        a.display();
        cout<<endl;
        }
      getch();
     }
```

OUTPUT:

Queue before adding Elements: EMPTY

Addition Number : 1 : 32
Addition Number : 2 : 32 23
Addition Number : 3 : 32 23 45
Addition Number : 4 : 32 23 45 99
Addition Number : 5 : 32 23 45 99 24

Queue after adding Elements: 32 23 45 99 24

Deletion Number : 1 : 23 45 99 24
Deletion Number : 2 : 45 99 24
Deletion Number : 3 : 99 24
Deletion Number : 4 : 24
Deletion Number : 5 : EMPTY

As you can clearly see through the output of this program that addition is always done at the end of the queue while deletion is done from the front end of the queue.

Tower Of Hanoi

Tower of Hanoi is a historical problem, which can be easily expressed using recursion. There are N disks of decreasing size stacked on one needle, and two other empty needles. It is required to stack all the disks onto a second needle in the decreasing order of size. The third needle can be used as a temporary storage. The movement of the disks must confirm to the following rules,

1. Only one disk may be moved at a time
2. A disk can be moved from any needle to any other.
3. The larger disk should not rest upon a smaller one.

Question : write a c program to implement tower of Hanoi using stack ?

Solution

```
/* Program of towers of Hanoi. */


        #include <stdio.h>
        #include <conio.h>

        void move ( int, char, char, char ) ;

        void main( )
           {
              int n = 3 ;
              clrscr( ) ;
              move ( n, 'A', 'B', 'C' ) ;
              getch( ) ;
           }

           void move ( int n, char sp, char ap, char ep )
              {
                       if ( n == 1 )
                       printf ("\nMove from %c to %c ", sp, ep ) ;
                       else
                          {
                             move ( n - 1, sp, ep, ap ) ;
                             move ( 1, sp, ' ', ep ) ;
                             move ( n - 1, ap, sp, ep ) ;
                          }
              }
```

Function Calls and Stack

A stack is used by programming languages for implementing function calls. write a program to check how function calls are made using stack.

Solution

SOLUTION

```
/* To show the use of stack in function calls */

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <dos.h>

unsigned int far *ptr ;
void ( *p )( void ) ;

void f1( ) ;
void f2( ) ;

void main( )
{
        f1( ) ;
        f2( ) ;
        printf ( "\nback to main..." ) ;
        exit ( 1 ) ;
}

void f1( )
{
        ptr = ( unsigned int far * ) MK_FP ( _SS, _SP + 2
) ;
        printf ( "\n%d", *ptr ) ;
        p = ( void ( * )( ) ) MK_FP ( _CS, *ptr ) ;
        ( *p )( ) ;
        printf ( "\nI am f1( ) function " ) ;
}

void f2( )
{
        printf ( "\nI am f2( ) function" ) ;
}
```

PROBLEM 1 :

The Queue has operations create, append, front, remove and is Empty.
A Queue contains a sequence of integers. Design an algorithm to construct another queue containing the same integers but in reverse order. The only queue operations available to you are those listed above.

PROBLEM 2

Repeat the previous question, only this time you have to leave the first queue in its original state as well.

PROBLEM 3 :

A queue contains a sequence of alphabetic characters. Design an algorithm to test whether the contents of the queue is a palindrome. As before you should assume that the only queue operations available to you are those listed in question 1.