MODULE 2
STACKS AND QUEUES

There are certain situations in computer science that one wants to restrict insertions and deletions so that they can take place only at the beginning or the end of the list, not in the middle. Two of such data structures that are useful are:

• Stack. • Queue.

Linear lists and arrays allow one to insert and delete elements at any place in the list i.e., at the beginning, at the end or in the middle.

STACK:

A stack is a list of elements in which an element may be inserted or deleted only at one end, called the top of the stack. Stacks are sometimes known as LIFO (last in, first out) lists. As the items can be added or removed only from the top i.e. the last item to be added to a stack is the first item to be removed.

The two basic operations associated with stacks are:

• **Push**: is the term used to insert an element into a stack.

• **Pop**: is the term used to delete an element from a stack.

.

All insertions and deletions take place at the same end, so the last element added to the stack will be the first element removed from the stack. When a stack is created, the stack base remains fixed while the stack top changes as elements are added and removed. The most accessible element is the top and the least accessible element is the bottom of the stack.

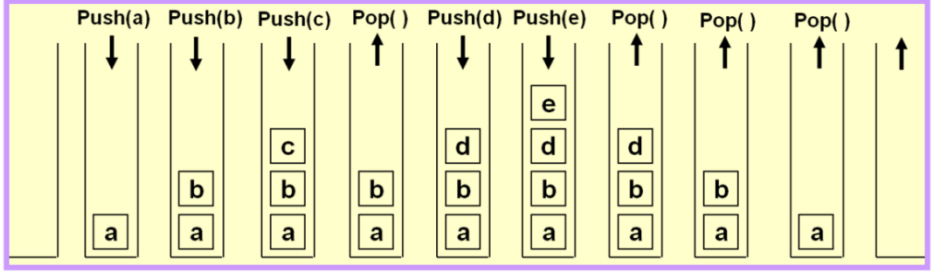
Representation of Stack:

Let us consider a stack with N elements capacity. This is called as the size of the stack. The number of elements to be added should not exceed the maximum size of the stack. If we attempt to add new element beyond the maximum size, we will encounter a stack overflow condition. Similarly, you cannot remove elements beyond the base of the stack. If such is the case, we will reach a stack underflow condition. (an empty stack)

When an element is added to a stack, the operation is performed by push. The removal of an element is performed by the pop operation.

Operations on Stack

	Stack's contents	TOP value	Output
1. Init_stack( )	<empty>	-1	
2. Push('a')	a	0	
3. Push('b')	a b	1	
4. Push('c')	a b c	2	
5. Pop( )	a b	1	c
6. Push('d')	a b d	2	c
7. Push('e')	a b d e	3	c
8. Pop( )	a b d	2	c e
9. Pop( )	a b	1	c e d
10. Pop( )	a	0	c e d b
11. Pop( )	<empty>	-1	c e d b a



Stack Using Array

A stack data structure can be implemented using one dimensional array. But stack implemented using array, can store only fixed number of data values. This implementation is very simple, just define a one dimensional array of specific size and insert or delete the values into that array by using **LIFO principle** with the help of a variable **'top'**. Initially top is set to -1. Whenever an element is to be inserted into the stack, increment the top value by one and then insert. Whenever an element is to be deleted from the stack the stack, then delete the top value and decrement the top value by one.

Stack Implementation

- Using static arrays
- Using dynamic arrays
- Using linked list

Stack Operations using Array

A stack can be implemented using array as follows...

push(value) - Inserting value into the stack

In a stack, push() is a function used to insert an element into the stack. In a stack, the new element is always inserted at **top** position. Push function takes one integer value as parameter and inserts that value into the stack. We can use the following algorithm to push an element on to the stack...

Step 1: Check whether **stack** is **FULL**. (**top == SIZE-1**)
Step 2: If it is **FULL**, then display **"Stack is FULL!!! Stack overflow!!!"** and terminate the function.
Step 3: If it is **NOT FULL**, then increment **top** value by one (**top++**) and set stack[top] to value (**stack[top] = value**).

pop() - Delete a value from the Stack

In a stack, pop() is a function used to delete an element from the stack. In a stack, the element is always deleted from **top** position. Pop function does not take any value as parameter. We can use the following algorithm to pop an element from the stack...

Step 1: Check whether **stack** is **EMPTY**. (**top == -1**)
Step 2: If it is **EMPTY**, then display **"Stack is EMPTY!!! Deletion is not possible!!!"** and terminate the function.
Step 3: If it is **NOT EMPTY**, then delete **stack[top]** and decrement **top** value by one (**top--**).

display() - Displays the elements of a Stack

We can use the following algorithm to display the elements of a stack...

Step 1: Check whether **stack** is **EMPTY**. (**top == -1**)
Step 2: If it is **EMPTY**, then display **"Stack is EMPTY!!!"** and terminate the function.
Step 3: If it is **NOT EMPTY**, then define a variable '**i**' and initialize with top. Display **stack[i]** value and decrement **i** value by one (**i--**).
Step 3: Repeat above step until **i** value becomes '0'.

Implementation of stack operations using array

```
# define N  5
int a[N],tos=-1;

void push()
{
 int key;
 if (tos==N-1)
   {
    printf("stack full\n");
    return;
   }
    printf("enter the elemnt to be inserted\n");
    scanf("%d",&key);
    a[++tos]=key;
 }

 void pop()
 {
  if (tos==-1)
   printf("underflow\n");
   else
    {
     printf("the popped element is %d",a[tos]);
     tos--;
    }
   }

   void display()
   {
    int i;
    if (tos== -1)
     {
     printf("no elements in stack\n");
     return;
     }
      for(i=tos;i>=0;i--)
       printf("%d\t",a[i]);
    }
```

Stack using dynamic arrays

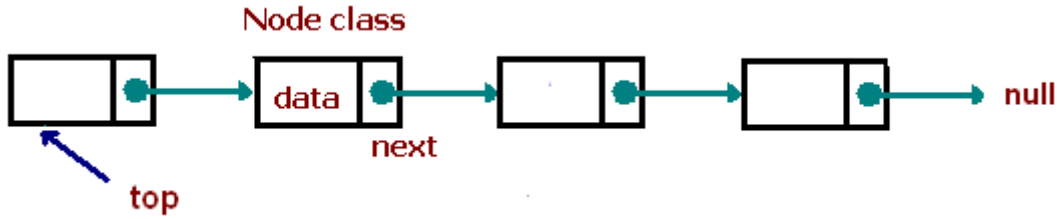
Issues with static stack

- Can run out of space when stack size is set too small
- Can waste memory if stack size set too large

Use a dynamic array implementation where memory for the stack array is set dynamically**.(Implementation –refer class notes)**

Stack using Linked list

A stack can be represented as a linked list(using a singly linked list or one way list).The data field stores the elements of the stack and the link field hold pointers to the neighbouring elements of the stack. The start pointer behaves as the top pointer and . NULL pointer in the last node indicates the bottom of the stack. In a stack push and pop operations are performed at one end called top. We can perform similar operations at one end of list using top pointer.



Push operation is done by inserting a node at the start of the list and pop is done by deleting the element pointed to by the top pointer.

Overflow/Underflow:

No limitation on the capacity of a linked stack and hence no overflow condition. Underflow or empty condition occurs when top==NULL

Linked Stack implementation

```
struct node
{
 int data;
 struct  node *next;
 };

 typedef struct node stack;
 stack  *top=NULL;

 stack *getnode()
 {
  stack *newnode;
  newnode=(stack*)malloc(sizeof(stack));
  if (newnode==NULL)
   printf("error in memory alloc\n");
   printf("enetr data\n");
   scanf("%d",&newnode->data);
   newnode->next=NULL;
   return newnode;
  }
// Function to traverse and display elements of stack
 void traverse()
 {
  stack *temp=top;
  if (temp==NULL)
  {
   printf("stack empty\n");
   return;
   }
```

```
  while(temp!=NULL)
   {
    printf("%d",temp->data);
    temp=temp->next;
   }
  }
```

//Function to implement push operation

```
  void pushlinkedstack()
  {
   stack *n1;
   n1=getnode();
   if (top==NULL)
   { top=n1;
    n1->next=NULL;
    return;
    }
    n1->next=top;
    top=n1;
   }
```

//Function to implement pop operation

```
  void poplinkedstack()
    {
    stack *temp;
    if (top==NULL)
     {
    printf("stack empty\n");
    return;
    }
     temp=top;
     top=top->next;
     printf("the element deleted is %d",temp->data);
     free(temp);
    }
```

Algebraic Expressions:

An algebraic expression is a legal combination of operators and operands. Operand is the quantity on which a mathematical operation is performed. Operand may be a variable like x, y, z or a constant like 5, 4, 6 etc. Operator is a symbol which signifies a mathematical or logical operation between the operands. Examples of familiar operators include +, -, *, /, ^ etc. An algebraic expression can be represented using three different notations. They are infix, postfix and prefix notations:

Infix: It is the form of an arithmetic expression in which we fix (place) the arithmetic operator in between the two operands.

Example: (A + B) * (C - D)

Prefix: It is the form of an arithmetic notation in which we fix (place) the arithmetic operator before (pre) its two operands. The prefix notation is called as polish notation (due to the polish mathematician Jan Lukasiewicz in the year 1920).

Example: * + A B – C D

Postfix: It is the form of an arithmetic expression in which we fix (place) the arithmetic operator after (post) its two operands. The postfix notation is called as suffix notation and is also referred to reverse polish notation.

Example: A B + C D - *

The three important features of postfix expression are:
1**.** The operands maintain the same order as in the equivalent infix expression.
2. The parentheses are not needed to designate the expression unambiguously.
3. While evaluating the postfix expression the priority of the operators is no longer relevant.
We consider five binary operations: +, -, *, / and $ or ↑ (exponentiation). For these binary operations, the following in the order of precedence (highest to lowest):

OPERATOR	PRECEDENCE	VALUE
Exponentiation ($ or ↑ or ^)	Highest	4
*, /	Next highest	3
+, -	Lowest	2
#	Lowermost (endofexpn)	1

Applications of stacks:

1. Stack is used by compilers to check for balancing of parentheses, brackets and braces.
2. Stack is used to evaluate a postfix expression.
3. Stack is used to convert an infix expression into postfix/prefix form.
4. In recursion, all intermediate arguments and return values are stored on the processor's stack.
5. During a function call the return address and arguments are pushed onto a stack and on return they are popped off. Conversion from infix to postfix:

Conversion from infix to postfix expression

Procedure to convert from infix expression to postfix expression is as follows: (algorithm)
1. Scan the infix expression from left to right.
2. a) If the scanned symbol is left parenthesis, push it onto the stack.
b) If the scanned symbol is an operand, then place directly in the postfix expression (output).
c) If the symbol scanned is a right parenthesis, then go on popping all the items from the stack and place them in the postfix expression till we get the matching left parenthesis.
d) If the scanned symbol is an operator, then go on removing all the operators from the stack and place them in the postfix expression, if and only if the precedence of the operator which is on the top of the stack is greater than (or greater than or equal) to the precedence of the scanned operator and push the scanned operator onto the stack otherwise, push the scanned operator onto the stack

Note :Refer class notes for examples and steps in conversion(detailed)

Evaluation of postfix expression

1. Scan the postfix expression from left to right.
2. If the scanned symbol is an operand, then push it onto the stack.
3. If the scanned symbol is an operator, pop two symbols from the stack ,assign to operand 2 and operand1 respectively. Perform operation and push onto stack
4. Repeat steps 2 and 3 till the end of the expression.
(Refer class notes for steps in conversion and examples)

Recursion

Recursion is deceptively simple in statement but exceptionally complicated in implementation. Recursive procedures work fine in many problems. Many programmers prefer recursion though simpler alternatives are available. It is because recursion is elegant to use though it is costly in terms of time and space.

Introduction to Recursion:

A function is recursive if a statement in the body of the function calls itself. Recursion is the process of defining something in terms of itself. For a computer language to be recursive, a function must be able to call itself.

For example, let us consider the function factr() shown below, which computers the factorial of an integer.

```
int factorial (int);
main() {
int num, fact;
 printf (“Enter a positive integer value: ");
  scanf (“%d”, &num);
fact = factorial (num);
printf ("\n Factorial of %d =%5d\n", num, fact);
 }
int factorial (int n)
 {    int result;
  if (n == 0)   return (1);
else
result = n * factorial (n-1);
return (result);
 }
```

A non-recursive or iterative version for finding the factorial is as follows:

```
factorial (int n)
 { int i, result = 1;
if (n == 0)
return (result);
else
{
for (i=1; i<=n; i++)
result = result * i;
 return (result);
 }  }
```

The operation of the non-recursive version is clear as it uses a loop starting at 1 and ending at the target value and progressively multiplies each number by the moving product. When a function calls itself, new local variables and parameters are allocated storage on the stack and the function code is executed with these new variables from the start. A recursive call does not make a new copy of the function. Only the arguments and variables are new. As each recursive call returns, the old local variables and parameters are removed from the stack and execution resumes at the point of the function call inside the function.

When writing recursive functions, there must be an exit condition somewhere to force the function to return without the recursive call being executed. If there is no exit condition, the

recursive function will loop forever until you run out of stack space and indicate error about lack of memory, or stack overflow.

Differences between recursion and iteration:

- Both involve repetition.
- Both involve a termination test.
- Both can occur infinitely.

Iteration	Recursion
Iteration explicitly uses a repetition structure	Recursion achieves repetition through repeated function calls.
Iteration terminates when the loop ends	Recursion terminates when a base case is recognized
Iteration keeps modifying the counter until the loop continuation condition fails.	Recursion keeps producing simple versions of the original problem until the base case is reached.
Iteration normally occurs within a loop, so the extra memory usage is avoided	Recursion causes another copy of the function and hence a considerable memory space's occupied.
. It reduces the processor's operating time	. It increases the processor's operating time

Factorial of a given number:

The operation of recursive factorial function is as follows:
Start out with some natural number N (in our example, 5).
The recursive definition is:

n = 0, 0 ! = 1 Base Case
n > 0, n ! = n * (n - 1) ! Recursive Case

Recursion Factorials:
5! = 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
We define 0! to equal 1, and we define factorial N (where N > 0), to be N * factorial (N-1). All recursive functions must have an exit condition that is a state when the function terminates. The exit condition in this example is when N = 0.

Tracing of the flow of the factorial () function:

When the factorial function is first called with, say, N = 5, here is what happens:
FUNCTION: Does N = 0? No Function Return Value = 5 * factorial (4)
At this time, the function factorial is called again, with N = 4.
FUNCTION: Does N = 0? No Function Return Value = 4 * factorial (3)
At this time, the function factorial is called again, with N = 3.
FUNCTION: Does N = 0? No Function Return Value = 3 * factorial (2)
At this time, the function factorial is called again, with N = 2.
FUNCTION: Does N = 0? No Function Return Value = 2 * factorial (1)
At this time, the function factorial is called again, with N = 1.
FUNCTION: Does N = 0? No Function Return Value = 1 * factorial (0)
At this time, the function factorial is called again, with N = 0.
FUNCTION: Does N = 0? Yes Function Return Value = 1

Now, trace the way back up! See, the factorial function was called six times. At any function level call, all function level calls above still exist! So, when we have N = 2, the function instances where N = 3, 4, and 5 are still waiting for their return values.

So, the function call where N = 1 gets retraced first, once the final call returns 0. So, the function call where N = 1 returns 1*1, or 1. The next higher function call, where N = 2, returns 2 * 1 (1, because that's what the function call where N = 1 returned). Just keep working up the chain till the final solution is obtained.

When N = 2, 2 * 1, or 2 was returned. When N = 3, 3 * 2, or 6 was returned. When N = 4, 4 * 6, or 24 was returned. When N = 5, 5 * 24, or 120 was returned.
And since N = 5 was the first function call (hence the last one to be recalled), the value 120 is returned.

The Towers of Hanoi:

In the game of Towers of Hanoi, there are three towers labeled 1, 2, and 3. The game starts with n disks on tower A. For simplicity, let n is 3. The disks are numbered from 1 to 3, and without loss of generality we may assume that the diameter of each disk is the same as its number. That is, disk 1 has diameter 1 (in some unit of measure), disk 2 has diameter 2, and disk 3 has diameter 3. All three disks start on tower A in the order 1, 2, 3. **The objective of the game is to move all the disks in tower 1 to entire tower 3 using tower 2. That is, at no time can a larger disk be placed on a smaller disk.**

The rules to be followed in moving the disks from tower 1 tower 3 using tower 2 are as follows:

- Only one disk can be moved at a time.
- Only the top disc on any tower can be moved to any other tower.
- A larger disk cannot be placed on a smaller disk.

The towers of Hanoi problem can be easily implemented using recursion. To move the largest disk to the bottom of tower 3, we move the remaining n – 1 disks to tower 2 and then move the largest disk to tower 3. Now we have the remaining n – 1 disks to be moved to tower 3. This can be achieved by using the remaining two towers. We can also use tower 3 to place any disk on it, since the disk placed on tower 3 is the largest disk and continue the same operation to place the entire disks in tower 3 in order.
The program that uses recursion to produce a list of moves that shows how to accomplish the task of transferring the n disks from tower 1 to tower 3 is as follows:

```
void towers_of_hanoi (int n, char *a, char *b, char *c);
int cnt=0;
int main (void)
 {
int n;
 printf("Enter number of discs: ");
scanf("%d",&n);
towers_of_hanoi (n, "Tower 1", "Tower 2", "Tower 3");
 getch();
}
void towers_of_hanoi (int n, char *a, char *b, char *c)
{
 if (n == 1)
{        ++cnt;
```

```
        printf ("\n%5d: Move disk 1 from %s to %s", cnt, a, c);
        return;
 }
 else
{
        towers_of_hanoi (n-1, a, c, b);  ++cnt;
        printf ("\n%5d: Move disk %d from %s to %s", cnt, n, a, c);
        towers_of_hanoi (n-1, b, a, c);  return;
} }
```

Output of the program:

RUN 1:
Enter the number of discs: 3
1: Move disk 1 from tower 1 to tower 3.
2: Move disk 2 from tower 1 to tower 2.
3: Move disk 1 from tower 3 to tower 2.
4: Move disk 3 from tower 1 to tower 3.
5: Move disk 1 from tower 2 to tower 1.
6: Move disk 2 from tower 2 to tower 3.
7: Move disk 1 from tower 1 to tower 3.

Fibonacci Sequence Problem:

A Fibonacci sequence starts with the integers 0 and 1. Successive elements in this sequence are obtained by summing the preceding two elements in the sequence. For example, third number in the sequence is 0 + 1 = 1, fourth number is 1 + 1= 2, fifth number is 1 + 2 = 3 and so on. The sequence of Fibonacci integers is given below:
0 1 1 2 3 5 8 13 21 . . . . . .

A recursive definition for the Fibonacci sequence of integers may be defined as follows:
Fib (n) = n if n = 0 or n = 1
Fib (n) = fib (n-1) + fib (n-2) for n >=2
We will now use the definition to compute fib(5):
fib(5) = fib(4) + fib(3)
=fib(3) + fib(2) + fib(3)
=fib(2) + fib(1) + fib(2) + fib(3)
=fib(1) + fib(0) + fib(1) + fib(2) + fib(3)
=1 + 0 + 1 + fib(1) + fib(0) + fib(3)
=1 + 0 + 1 + 1 + 0 + fib(2) + fib(1)
=1 + 0 + 1 + 1 + 0 + fib(1) + fib(0) + fib(1)
=1 + 0 + 1 + 1 + 0 + 1 + 0 + 1 = 5
fib(2) is computed 3 times, and fib(3),is computed 2 times in the above calculations. The values of fib(2) or fib(3) are saved and reused whenever needed.
A recursive function to compute the Fibonacci number in the nth position is given below:

```
main()
{
printf (“=nfib(5) is %d”, fib(5));
}

fib (int n) {
int x;
if (n==0 | | n==1) return n;
```

```
x=fib(n-1) + fib(n-2);
return (x);
}
```

Program to calculate the greatest common divisor:

```
int check_limit (int a[], int n, int prime);
int check_all (int a[], int n, int prime);
long int gcd (int a[], int n, int prime);
void main()
{
        int a[20], stat, i, n, prime;
         printf (“Enter the limit: “);
        scanf (“%d”, &n);
         printf (“Enter the numbers: “);
         for (i = 0; i < n; i ++)
         scanf (“%d”, &a[i]);
         printf (“The greatest common divisor is %ld”, gcd (a, n, 2));
}
int check_limit (int a[], int n, int prime)
{
         int i;
        for (i = 0; i < n; i++)
         if (prime > a[i])   return 1;
        return 0;
 }
int check_all (int a[], int n, int prime)
{
        int i;
        for (i = 0; i < n; i++)
         if ((a[i] % prime) != 0)    return 0;
         for (i = 0; i < n; i++)
           a[i] = a[i] / prime;  return 1;
}

long int gcd (int a[], int n, int prime)
 {
         int i;
         if (check_limit(a, n, prime))       return 1;
         if (check_all (a, n, prime))
         return (prime * gcd (a, n, prime));
 else
        return (gcd (a, n, prime = (prime == 2) ? prime+1 : prime+2));
}
```

Output:

```
Enter the limit: 5
Enter the numbers: 99     55     22     77     121
The greatest common divisor is 11
```

Ackerman's Function

In computability theory, the Ackermann function, named after Wilhelm Ackermann, is one of the simplest andearliest-discovered examples of a total computable function that is not primitive recursive. All primitive recursive functions are total and computable, but the Ackermann function illustrates that not all total computable functions are primitive recursive.
The two-argument Ackermann function, is defined as follows for nonnegative integers m and N

```
A(m,n) = n+1                    if m=0
         A(m-1,1)               if m>0 and n=0
         A(m-1,A(m,n-1))        if m>0 andn>0
```

Its value grows rapidly, even for small inputs. For example A(4,2) is an integer of 19,729 decimal digits.

```
/* Akerman Function*/
#include<stdio.h>
#include<stdlib.h>

int ack(int,int,int);

main()
{
int m,n;
printf("Enter the value for m : ");
scanf("%d",&m);
printf("Enter the value for n : ");
scanf("%d",&n);
printf("The value is : %d\n",ack(m,n);
}

int ack(int m,int n,)
{
if(m==0)
  return(n+1);
else if(m>0 && n==0)
  return ack(m-1,1);
else
  return ack(m-1,ack(m,n-1));

}
```

Queue

A queue is a linear list in which elements can be added at one end and elements can be removed only at other end. So the information in this list is processed in same order as it was received .Hence queue is called a FIFO structure.(First In First Out).
Ex: people waiting in a line at a bus stop.
The first person in queue is the first person to take bus. Whenever new person comes he joins at end of the queue.

Type of queues
1. Linear Queue 2. Circular queue. 3. Priority queue 4.Deque

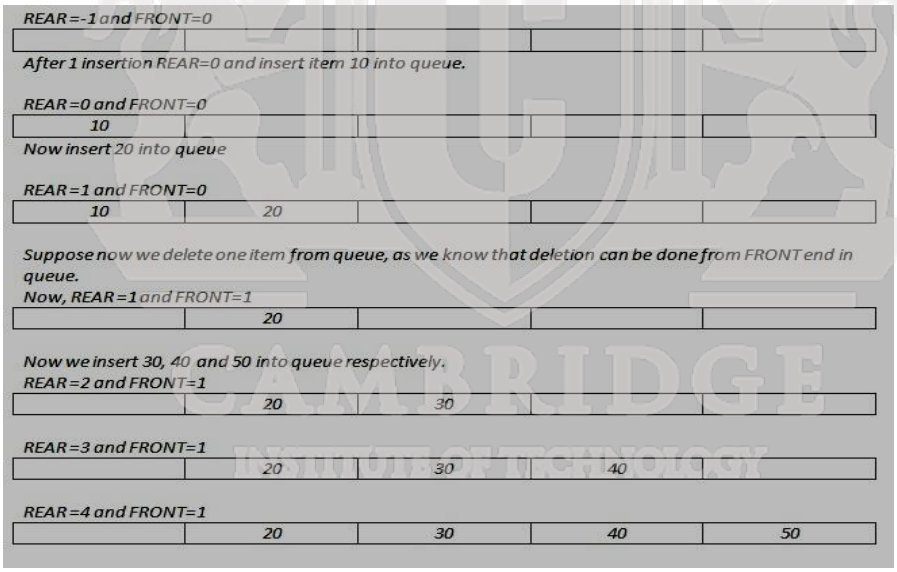
Linear Queue
It is a linear data structure.
It is considered as ordered collection of items.
It supports FIFO (First In First Out) property.
It has three components:
A **Container of items** that contains elements of queue.
A pointer **front** that points the first item of the queue.
A pointer **rear** that points the last item of the queue.
Insertion is performed from **REAR** end.
Deletion is performed from **FRONT** end.
Insertion operation is also known as **ENQUEUE** in queue.
Deletion operation is also known as DEQUEUE in queue.

Implementation of Queue
Queue can be implementing by two ways:

- Array implementation.(Static and Dynamic arrays)
- Linked List implementation.

Array Representation of Queue

In Array implementation **FRONT** pointer initialized with **0** and **REAR** initialized with **-1**.Consider the implementation: - If there are 5 items in a Queue,



Note: In case of empty queue, front is one position ahead of rear : FRONT = REAR + 1;.This is the queue underflow condition.
The queue is full when REAR =N-1.This is the queue overflow condition.

The figure above ,the last case after insertion of three elements, the rear points to 4, and hence satisfies the overflow condition although the queue still has space to accommodate one more element .This problem can be overcome by making the rear pointer reset to the starting position in the queue and hence view the array as a circular representation. This is called a circular queue.

Implementation of queue using arrays

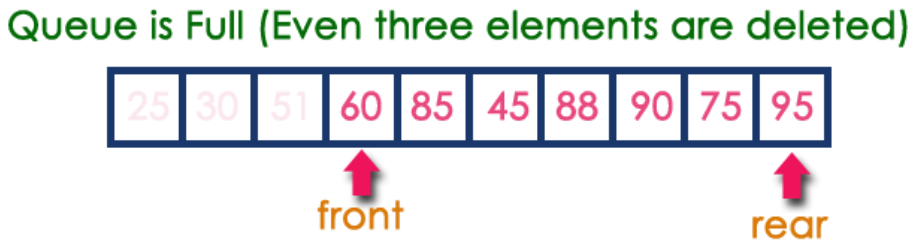
```
# include <conio.h>
 # define MAX 5
int Q[MAX];
int front=0, rear=-1;
void insertQ()  //Enqueue
{
int data;
if(rear == MAX-1)
 {   printf("\n Linear Queue is full");
  return;  }
 printf("\n Enter data: ");
scanf("%d", &data);
  Q[++rear] = data;
printf("\n Data Inserted in the Queue ");
}
void deleteQ()            // dequeue
 {  if( front>rear)      //OR  front=rear +1
{
   printf("\n\n Queue is Empty..");   return;
 }
  printf("\n Deleted element from Queue is %d", Q[front]);
  front++;
}
void displayQ()
{  int i;
 if(front >rear)
  {   printf("\n\n\t Queue is Empty");   return;  }
   printf("\n Elements in Queue are: ");
  for(i = front; i < rear; i++)
     printf("%d\t", Q[i]);
}
```

Circular Queue

In a normal Queue Data Structure, elements can be inserted until queue becomes full. But once if queue becomes full, no more elements can be inserted until all the elements are deleted from the queue. For example consider the queue below...

After inserting all the elements into the queue.

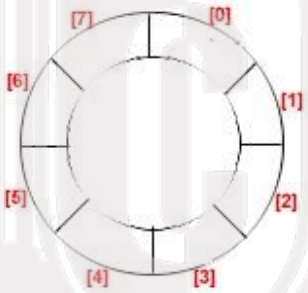
Now consider the following situation after deleting three elements from the queue...



This situation also says that Queue is Full and the new elementcannot be inserted because, '**rear**' is still at last position. In above situation, even though we have empty positions in the queue they cannot be used to insert new element. This is the major drawback in normal queue data structure. This is overcome in circular queue data structure.

So what's a Circular Queue?

A circular queue is linear data structure that contains a collection of data which allows addition of data at the end of the queue and removal of data at the beginning of the queue. Circular queues have a fixed size.Circular queue follows FIFO principle. Queue items are added at the rear end and the items are deleted at front end of the circular queue.

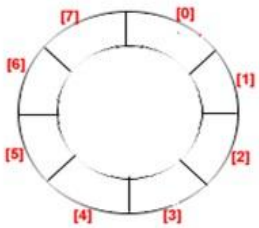


A circular queue looks like

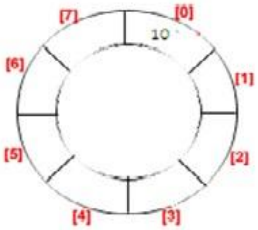
Note:
Note that the container of items is an array. Array is stored in main memory. Main memory is linear. So this circularity is only logical. There cannot be physical circularity in main memory.

Consider the example with Circular Queue implementation

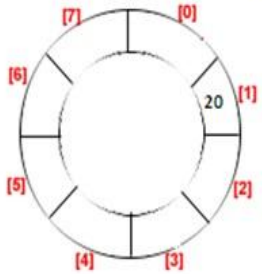
1) Initially: **Front = 0** and **rear = -1**



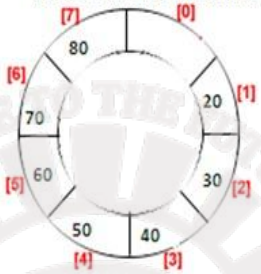
2) Add item 10 then **front = 0** and **rear =0**.



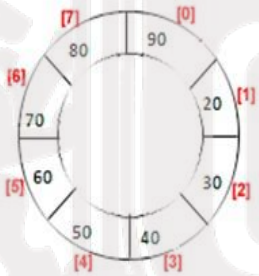
3) Now delete one item then **front = 1** and **rear =1**.



4) Like this now insert 30, 40, and 50,50,70,80 respectability then **front =1** and **rear = 7.**



5) Now in case of linear queue, we can not access 0 block for insertion but in circular queue next item will be inserted of 0 block then **front =0** and **rear = 0.**



Addition causes the increment in REAR. It means that when REAR reaches N-1 position then Increment in REAR causes REAR to reach at first position that is 0.

```
1   if( rear  == N -1 )
2      rear  = 0;
3   else
4      rear = rear + 1;
```

The short-hand equivalent representation may be

rear = (rear + 1) % N;

Deletion causes the increment in FRONT. It means that when FRONT reaches the N-1 position, then increment in FRONT, causes FRONT to reach at first position that is 0.

```
1   if( front  == N -1 )
2    front  = 0;
3   else
4    front = front + 1;
```

The short-hand equivalent representation may be

front = (front + 1) % N;

In any queue it is necessary that:

Before insertion, fullness of Queue must be checked (for overflow).

Before deletion, emptiness of Queue must be checked (for underflow).

Use count variable to hold the current position (in case of insertion or deletion).

Operation of Circular Queue using count

- Addition or Insertion operation.
- Deletion operation.
- Display queue contents

Array Implementation of Circular Queue

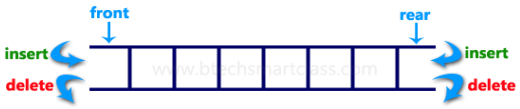
```
#define MAX 4
int CQ[MAX], n;
int r = -1;
int f = 0,ct=0;

void enqueue()   //function to insert an element to queue
{
   int key;

   if (ct == n )
   {
     printf("Queue Overflow\n");
     return;
   }
   printf("\nenter the element for adding in queue : ");
   r = (r+1)%n;
   scanf("%d", &key);
   CQ[r]=key;
   ct++;
}
void dequeue()          //function to remove an element from queue
{
   if (ct == 0)
   {
         printf("Queue Underflow\n");
         return ;
   }
         printf("Element deleted from queue is : %d\n", CQ[f]);
         f=(f+1)%n;
         ct--;
}
void display()
{
   int i,k=f;
   if (ct == 0)
   {
     printf("Queue is empty\n");
     return;
   }
   printf("contents of Queue are :\n");
   for (i = 0; i < ct; i++)
   {
     printf("%d\t", CQ[k]);
     k=(k+1)%n;
   }}
```

Double ended queue (deck)

Double Ended Queue is also a Queue data structure in which the insertion and deletion operations are performed at both the ends (**front** and **rear**). That means, we can insert at both front and rear positions and can delete from both front and rear positions.

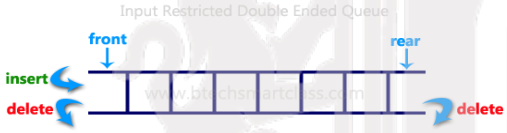


Double Ended Queue can be represented in TWO ways, those are as follows...

Input Restricted Double Ended Queue
Output Restricted Double Ended Queue

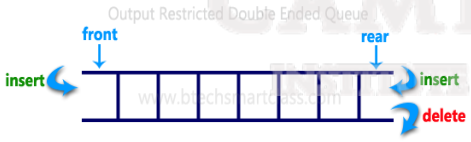
Input Restricted Double Ended Queue

In input restricted double ended queue, the insertion operation is performed at only one end and deletion operation is performed at both the ends.



Output Restricted Double Ended Queue

In output restricted double ended queue, the deletion operation is performed at only one end and insertion operation is performed at both the ends.



Deque is a variation of queue data structure, pronounced “deck”, which stands for double-ended queue. In a deque values can be inserted at either the front or the back, A collection of peas in a straw is a good example..

Queues and deques are used in a number of ways in computer applications. A printer, for example, can only print one job at a time. During the time it is printing there may be many different requests for other output to be printed. To handle this printer will maintain a queue of pending print tasks. Since you want the results to be produced in the order that they are received, a queue is the appropriate data structure.

For a deque the defining property is that elements can only be added or removed from the end points. It is not possible to add or remove values from the middle of the collection.

Operations on deque

- Insertfront
- Deletefront
- INsertrear
- Deleterear

Deque Implementation

- Using arrays
- Linked list –Doubly linked list

Array Implementation of Deque

```
#include<stdio.h>
#define N 5
int dq[N],front=0,rear=-1,count=0;

//insert element at the rear end
void insertrear()
{
int key;
if (count==N)
 printf("overflow\n");
else
  {
   printf("enter the key to be inserted\n");
   scanf("%d",&key);
   rear=(rear+1)%N;
   dq[rear]=key;
   count++;
  }
}
//insert element at the front
void insertfront()
{
  int key;
  if (count==N)
    printf("overflow\n");
  else
   {
     printf("enter the key elemet\n");
     scanf("%d",&key);
     if (front==0)
       front=N-1;
     else
       front=front-1;
    dq[front]=key;
    count++;
   }}
```

//delete element from the front

```
 void deletefront()
 {
  if (count==0)
   printf("underflow\n");
 else
  {
   printf("element deleted is%d",dq[front]);
   front=(front+1)%N;
   count--;
  }
 }
```

//delete element from the rear end

```
 void deleterear()
 {
 if  (count==0)
  printf("underflow\n");
 else
  {
   printf("%d is rear value\n",rear);
   printf("element deleted is %d",dq[rear]);
   if (rear==0)
     rear=N-1;
   else
     rear=rear-1;
   count--;
   }
 }
  void display()
 {
  int i,k;
   if (count==0)
              printf("empty queue\n");
   else
   {
       k=front;
      for(i=0;i<count;i++)
      {
        printf("%d\t",dq[k]);
        k=(k+1)%N;
      }
   }
 }
```

Deque Implementation using Doubly Linked List

```
#include<stdio.h>
struct deque
{
 int data;
 struct node *left;
```

```
 struct node *right;
};
 typedef struct deque node;
 node *start=NULL;

node *getnode()
 {
  node *newnode;
  newnode=(node *)malloc(sizeof(node));
  if (newnode==NULL)
   printf("error in memory alloc\n");
   printf("enetr data\n");
   scanf("%d",&newnode->data);
   newnode->left=newnode->right=NULL;
   return newnode;
  }

 void insertbegin()
 {
  node *n;
  n=getnode();
  if (start==NULL)
   {
    start=n;
    return;
  }
    n->right=start;
    start->left=n;
    start=n;
  }

  void traverse()
  {
   node *temp=start;
   if (temp==NULL)
   {
    printf("list empty\n");
    return;
   }
   while(temp!=NULL)
   {
    printf("%d",temp->data);
    temp=temp->right;
   }
  }
  void insertend()
    {
     node *temp=start;
     node *n;
     n=getnode();
```

```
    if (start==NULL)
     {
      start=n;return;
     }
    while(temp->right!=NULL)
          temp=temp->right;
    n->left=temp;
    temp->right=n;
   }

  void delbegin()
  {
   node *temp;
   if (start==NULL)
   {
   printf("list  empty\n");
   return;
   }
   temp=start;
   start=temp->right;
   start->left=NULL;
   printf("the element to be deleted is %d",temp->data);
   free(temp);
   }

    void delend()
   {
    node *temp,*prev;
    if (start==NULL)
     {
      printf("list empty\n");
           return;
     }
    if (start->right==NULL)
          {
           printf("node deleted is %d",start->data);
           free(start);
           start=NULL;
           return;
          }
  temp=start;
 while(temp->right!=NULL)
{
 prev=temp;
 temp=temp->right;
}
prev->right=NULL;
printf("deleted info is %d",temp->data);
free(temp);
 }
```

Priority Queue:

A priority queue is a collection of elements such that each element has been assigned a priority and such that the order in which elements are deleted and processed comes from the following rules:

1. An element of higher priority is processed before any element of lower priority.
2. two elements with same priority are processed according to the order in which they were added to the queue.

A prototype of a priority queue is time sharing system: programs of high priority are processed first, and programs with the same priority form a standard queue.

Implementation of a Priority Queue

A priority queue can be implemented by creating a sorted or ordered list. A sorted list can be used to store the elements so that when an element is to be removed, the queue need not be searched for an element with the highest priority, since the element with the highest priority is already in the first position. Insertions are handled by inserting the elements in order.

C Program to Implement Priority Queue to Add and Delete Elements

```
#include <stdio.h>
#define MAX 5
int pri_que[MAX];
int front=0, rear=-1;

/* Function to insert value into priority queue */
void insert_by_priority(int data)
{
   if (rear== MAX - 1)
   {
      printf("\nQueue overflow no more elements can be inserted");
      return;}

   if (rear == -1))
   {
      rear++;
      pri_que[rear] = data;
   }
   else
      check(data);
      rear++;
}
/* Function to check priority and place element */
void check(int data)
{
   int i,j;
   for (i = 0; i <= rear; i++)
   {
      if (data >= pri_que[i])
      {
         for (j = rear + 1; j > i; j--)
```

```
            pri_que[j] = pri_que[j - 1];
          pri_que[i] = data;
          return;
       }
   }
    pri_que[i] = data;
}
/* Function to delete an element from queue */
void delete_by_priority(int data)
{
    int i;
    if (rear==-1)
    {
       printf("\nQueue is empty no elements to delete");
       return;
    }
   printf( "The element deleted is %d",pri_qui[front];
  front++;
 }
/* Function to display queue elements */
void display_pqueue()
{ int i;
    if (rear == -1))
    {
       printf("\nQueue is empty");
       return;}
       for (i=front; i <= rear; i++)
    {
       printf(" %d ", pri_que[i]);
    }
```

Linked Representation of a Priority Queue

A priority queue can be implemented using linked lists. When a priority queue is implemented as a linked list, then every node of the list has three fields (1)the data part (2)The priority number of the element(3)the address of the next element.

Implementation of priority queue

```
struct pqueue
{
int priority;
int info;
struct node *next;
};
typedef struct pqueue node;
node  *start=NULL;
```

```
void insert(int item,int item_priority)
{
struct node *new1,*temp;

new1=(struct node *)malloc(sizeof(struct node));
if(start==NULL)
{
printf("Memory not available\n");
return;
}
new1->info=item;
new1->priority=item_priority;
/*Queue is empty or item to be added has priority more than first element*/
if( start==NULL || item_priority < start->priority )
{
new1->next=start;
start=new1;
}
else
{
temp =start;
while( temp->next!=NULL && temp->next->priority<=item_priority )
temp=temp->next;
new1->next=temp->next;
temp->next=new1;
}
}/*End of insert()*/

void del()
{
struct node *temp;
if( start==NULL)
{
printf("Queue Underflow\n");
}
else
{
temp=start;
printf("the deleted element is %d",temp->info);
start=start->next;
free(temp);
}
}/*End of del()*/
```

Applications of Queue:

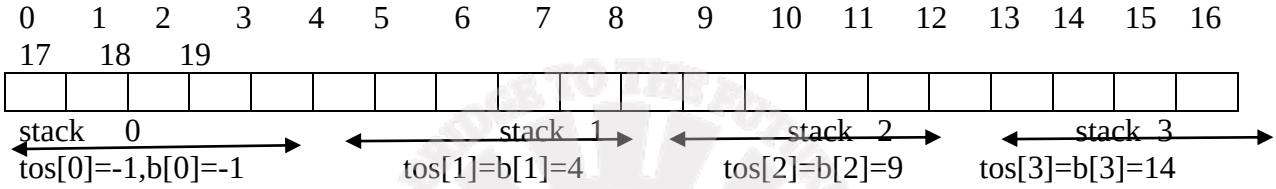
1. It is used to schedule the jobs to be processed by the CPU.
2. When multiple users send print jobs to a printer, each printing job is kept in the printing queue. Then the printer prints those jobs according to first in first out (FIFO) basis.
3. Breadth first search uses a queue data structure to find an element from a graph.

Multiple stacks

A sequential representation of a single stack using array is simple since only the top of the stack needs to be maintained and kept track. A linear structure like array can be used to represent multiple stacks. If multiple stacks are to be implemented, the array can be approximately divided into equal sized segments, each segment denoting a stack. The top and bottom of each stack are to be kept track of to manage insertions and deletions into the individual stacks.

Consider a set of N stacks to be implemented sunig an array. The array can be divided into N equal sized segments .

Say if the array can hold 20 elements, and 4 stacks are to be implemented then each individual stack hold 5 elements.



If i denotes an individual stack ,to establish multiple stacks, an array of top(tos[i])and bottom pointers (b[i])are maintained to keep track of the top and bottom of every stack.

Every stack's bottom and top pointer is set to B[i]=tos[i]=(size/n)*i-1 which enables dividing the stack to be divided into equal sized segments.

Overflow in any stack

tos[i]=b[i+1] // top pointer of one stack points to the bottom position of the following stack

Underflow in any stack

tos[i]=b[i] //top and bottom pointer of a stack in the same position

Implementation of multiple stacks using array

```
#define memsize 20              //size of array
#define maxstack 4              //number of stacks
int s[memsize],tos[maxstack],b[maxstack],n;
int main()
{
        int i;
        scanf("%d",&n);                 //number of stacks
        for(i=0;i<n;++)
        tos[i]=b[i]=(memsize/n)*i-1;
        b[n]=memsize-1;
// use switch case to call the different operations push, pop and display
}
void push()
{
         int ele;
        scanf("%d",&i);                 //stack number on which operation is to be done
        if (tos[i]==b[i+1])
{       printf("stack %d is full", i);return;}
        printf("enter the value to be inserted\n");
```

```
        scanf("%d",&ele);
        s[++tos[i]]=ele;
}
void pop()
{
        printf("enter the stack number\n");
        scanf("%d",&i);
        if (tos[i]==b[i])
         {
         printf("empty stack\n");return;}
        printf("deleted element is %d",s[tos[i]--];
}

void disp()
{
printf("enter the stack number\n");
scanf("%d",&i);
if (tos[i]==b[i])
 {
 printf("empty stack\n");return;}
printf("contents are \n");
for(j=b[i]+1;j<=tos[i];j++)
  printf("%d",s[j]);
}
```

Prepared by
Geetha.P,Asst Professor
Pankaja K, Asso Professor
Dept of CSE,CiTech

DO IT NOW.....SOMETIMES "LATER" BECOMES "NEVER".THERE IS NO SUBSTITUTE FOR HARD WORK!!!

GOOD LUCK!!!!