**Chapter Table of Contents**

**Chapter 1.2**

**Pointers and Recursion**

[Aim 35](#_bookmark0)

[Instructional Objectives 35](#_bookmark1)

[Learning Outcomes 35](#_bookmark2)

* + 1. [Introduction to Pointers and Recursive functions 36](#_bookmark3)
    2. [Declaring and Initializing Pointers 36](#_bookmark4)

[Self-assessment Questions 39](#_bookmark5)

* + 1. [Accessing a variable through its pointer 40](#_bookmark6)

[Self-assessment Questions 49](#_bookmark7)

* + 1. [Memory allocation functions 50](#_bookmark8)
       1. [malloc() 50](#_bookmark9)
       2. [calloc() 51](#_bookmark10)
       3. [free() 51](#_bookmark11)
       4. [Realloc() 52](#_bookmark12)

[Self-assessment Questions 54](#_bookmark13)

* + 1. [Recursion 55](#_bookmark14)
       1. [Definition 55](#_bookmark15)
       2. [Advantages 57](#_bookmark16)
       3. [Recursive programs 58](#_bookmark17)

[Self-assessment Questions 64](#_bookmark18)

[Summary 66](#_bookmark19)

[Terminal Questions 66](#_bookmark20)

[Answer Keys 67](#_bookmark21)

[Activity 68](#_bookmark22)

[Bibliography 69](#_bookmark23)

[e-References 69](#_bookmark24)

[External Resources 69](#_bookmark25)

[Video Links 69](#_bookmark26)

 **Aim**

To provide the students with the knowledge of Pointers and Recursion

 **Instructional Objectives**

After completing this chapter, you should be able to:

* Demonstrate the role of Pointers in data structures
* Explain memory allocation functions
* Explain Recursion and its advantages
* Describe how variables are accessed through pointers

******Learning Outcomes**

At the end of this chapter, you are expected to:

* Outline the steps to declare and initialise pointers
* List the advantages of Recursion
* Differentiate malloc() and calloc() and realloc()
* Write programs for binomial coefficient and Fibonacci using recursion
  + 1. **Introduction to Pointers and Recursive functions**

In any programming language, creating and accessing variables is very important. So far we have seen how to access variables using their variable names. In this chapter we introduce the concept of indirect access of objects using their address. This chapter describes the use of Pointers in accessing the variables using their memory addresses. As memory is a resource of a computer, it should be allocated and deallocated properly. This chapter also describes the different memory allocation techniques.

Further we introduce the concept of Recursion where a function calls itself again and again to complete a task. We will also study some Recursive functions and their program implementations. We will also look at the application of Recursion to various problems like factorial, GCD, Fibonacci series etc.

* + 1. **Declaring and Initializing Pointers**

**Introduction to pointers and its need in programming**

In all the previous programs, we referred to a variable with its variable name. Hence, the program did not care about the physical address of those variables. So, whenever we need to use a variable, we access them using the identifier which describes that variable.

Computers memory is divided into cells or locations which has unique addresses. Every variable that we declare in our program has an address associated with it. Thus a variable can also be accessed using the address of that variable. This can be achieved by using Pointers.

***Pointers*** can be defined as the special variables which have a capability to store address of any variable. They are used in C++ programs to access the memory and manipulate the data using addresses.

Pointers are a very important feature of C++ Programming as it allows to access data using their memory addresses and not directly using their variable names. Pointers do not have much significance when simple primitive data types like integer, character or float are used. But as the data structures become more complex, pointers play a very vital role in accessing the data.

***For example,*** consider an integer variable “a”. This variable will have 3 things associated with itself. First is its name, second is its value and third is the memory address. Assume that a variable “a” is having value “5” and address as “1000”. Hence we can access this variable “a” by using a pointer variable which will store the address of variable a. Thus we can manipulate values of any variable using a pointer variable.

Pointers are used for dynamic memory allocation so as to handle a huge amount of data. It would have been very difficult to allocate memory globally or through functions, without pointers.

**Declaration and Initialization of Pointer Variables**

Like any other variable in C++, pointer variables should be also declared before they are used for storing addresses.

**In this chapter we are going to study 2 operators known as Pointer Operators:**

* + - 1. **& (address of) Operator:** This operator gives the address of any variable.

***For example,*** if “max” is an integer variable, then &max will give memory address of variable max.

* + - 1. **\* (dereference) Operator:** This operator returns the value at any memory address. Thus, the argument to this operator must be a pointer. It is called as a dereference operator as it works in a opposite manner to the & operator.

***For example,*** if “ptr” is a pointer variable which stores the address of variable “a”, then

\*ptr will return the value located at the memory address pointed by “ptr”.

**A pointer variable can be declared as follows:**

**Syntax:** Data\_type \* pointer\_variable\_name;

We need to specify the data type followed by \* symbol and finally the name of pointer variable terminated by a semicolon.

***For example,*** int \*ptr;

This declaration tells the compiler that “ptr” is a pointer variable of type integer.

**A Pointer variable can be initialized as given below:**

**Syntax:** pointer\_variable\_name=& variable\_name;

***For example,*** int a; //variable a is declared as a integer variable int \*ptr; //declare ptr as pointer variable

ptr=&a; //ptr is a pointer variable which stores address of variable a

**We can combine declaration and initialisation in one step also:**

**Syntax:** data\_type \*pointer\_variable\_name=&variable\_name

***For example,*** int \*ptr= &a;

It means ptr is a pointer variable storing the address of variable a.

**Note:** pointer variable should always point to address of variable of same data type:

***For example,*** char a;

int \*ptr;

ptr=&a; //Invalid as a is char type and ptr is integer type.

**Dereferencing a pointer**

As it is already discussed, we can use \* (dereference) operator to access the value stored at the address pointed by the pointer variable. This \* operator is called as “value at operator” or “indirection operator”.

***For example,*** /\*pointer variable declaration and initialization\*/.

#include<stdio.h> int main()

{

int a=5; int \*ptr;

ptr=&a;

//a is a integer variable

//ptr is a pointer variable declared

//pointer ptr stores the address of variable a

printf("Address of variable a is: %d\n", &a);

//prints the address of variable a printf("Address of variable a is %d\n", ptr);

//prints address of var a as it is stored in ptr printf("Value of variable a is: %d\n", \*ptr);

//prints the value of variable a

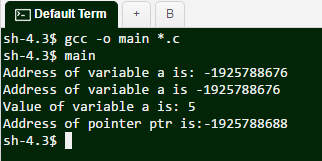
printf("Address of pointer ptr is:%d\n", &ptr);

//prints address of pointer variable ptr

return 0;

}

**Output:**



 **Self-assessment Questions**

* + - * 1. Pointer is special kind of variable which is used to store variable.

Data Variable b) Variable Name

c) Value d)Address

* + - * 1. Pointer variable is declared using preceding sign.

\* b) %

c) & d) ^

of the

* + - * 1. Consider the 32 bit compiler. We need to store address of integer variable to integer pointer. What will be the size of integer pointer,

2 Bytes b) 6 Bytes

c) 10 Bytes d) 4 Bytes

* + 1. **Accessing a variable through its pointer**

Pointers are special kind of variables which can hold the address of another variable. Once a pointer has been assigned the address of any variable, we can use the value of that variable and manipulate it as per the requirement.

We know that the pointers can store the address of any variable using & (address of) operator. Once the address is stored in pointer, we can use a \* (dereferencing/ indirection) operator followed by variable name to access the value of that variable.

***For example,***

int a, b; a=60;

int \*ptr; ptr= &a; b= \*ptr;

**Considering the above section of program, we declare 2 integer variables a and b.**

We have also declared a integer pointer variable “ptr”. In the next statement we assign the address of variable a to the pointer “ptr”.

The statement

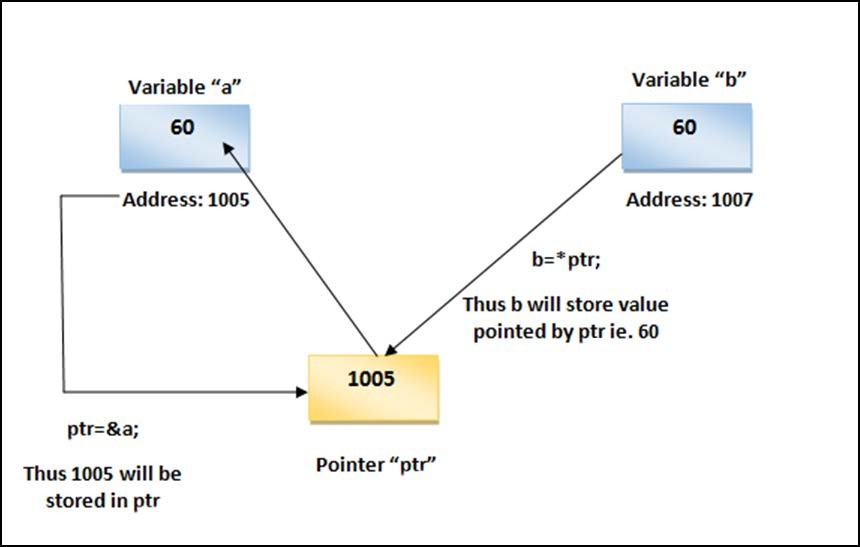
**b=\*ptr;**

will assign the value at the address pointed by “ptr” to the variable “b”. Thus variable “b” becomes same as variable “a”.

This is equivalent to the statement

**b= a;**

*The below figure 1.2.1 shows how a variable can be accessed using a pointer.*



*Figure 1.2.1: Accessing a variable through its pointer*

**Program:**

/\*Accessing a variable through pointer\*/ #include <stdio.h>

int main()

{

int a, b; a=60; b=0;

int \*ptr; ptr= &a;

printf("Value of variable a=%d\n", a); printf("Value of variable b=%d\n", b);

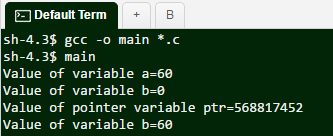
b=\*ptr; //assign value at address pointed by ptr to the variable b

printf("Value of pointer variable ptr=%d\n", ptr); printf("Value of variable b=%d\n", b);

return 0;

}

**Output:**



Variable “a” is assigned a value 60. Pointer variable “ptr” will store the address of variable “a” say 1005. When we say “b=\*ptr”, b will be assigned a value 60 pointed by the pointer “ptr”. Thus after execution of above program, a=60 and b will also have value 60.

**Different Types of Pointer variables and their use:**

In the following program, we have created 4 pointers: one integer pointer “iptr”, one float pointer “fptr”, one double type pointer “cptr” and one character type pointer “chptr”.

Each type of pointer variable stores the address of respective type of variable. For example, Character pointer variable will store the address of variable “ch” which is of type character.

The indirection operator *i.e.,* \* accesses an object of a specified data type at an address. Accessing any variable by its memory address is called indirect access. In the below given example \*iptr indirectly accesses the variable that iptr points to *i.e.,* variable a.

Similarly pointer variable \*fptr indirectly accesses the variable that fptr points to i.e. variable b.

**Program:**

/\*different types of pointer variables \*/ #include <stdio.h>

int main()

{

int a, \*iptr; float b, \*fptr; double c, \*cptr; char ch, \*chptr;

iptr=&a; //iptr stores address of integer variable a fptr=&b; //fptr stores address of float variable b cptr=&c; //cptr stores address of double variable c chptr=&ch; //chptr stores address of character variable ch a = 10;

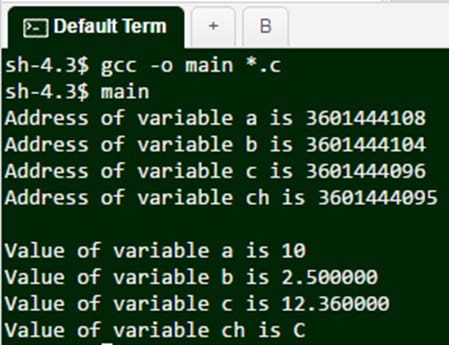
b = 2.5;

c = 12.36; ch = 'C';

printf("Address of variable a is %u \n", iptr); printf("Address of variable b is %u \n", fptr); printf("Address of variable c is %u \n", cptr); printf("Address of variable ch is %u \n\n", chptr); printf("Value of variable a is %d \n", \*iptr); printf("Value of variable b is %f \n", \*fptr); printf("Value of variable c is %f \n", \*cptr); printf("Value of variable ch is %c \n", \*chptr); return 0;

}

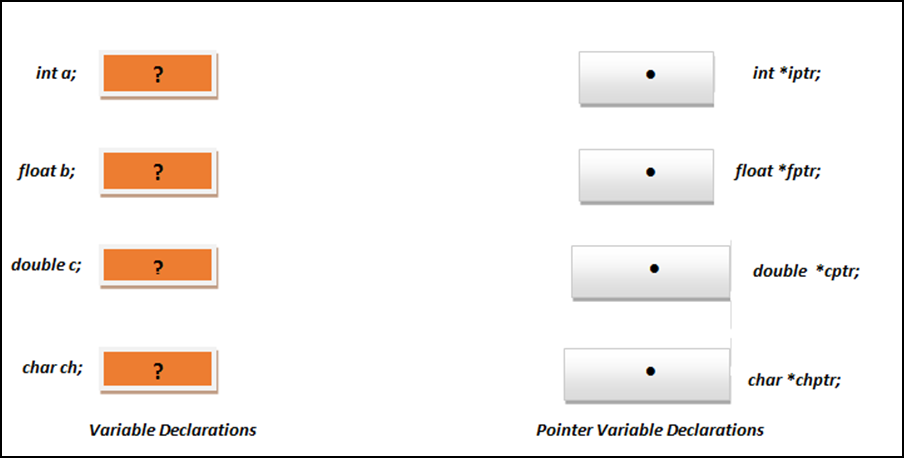
**Output:**



When the following pointer variable declarations are encountered, memory spaces are allocated for these variables at some addresses.

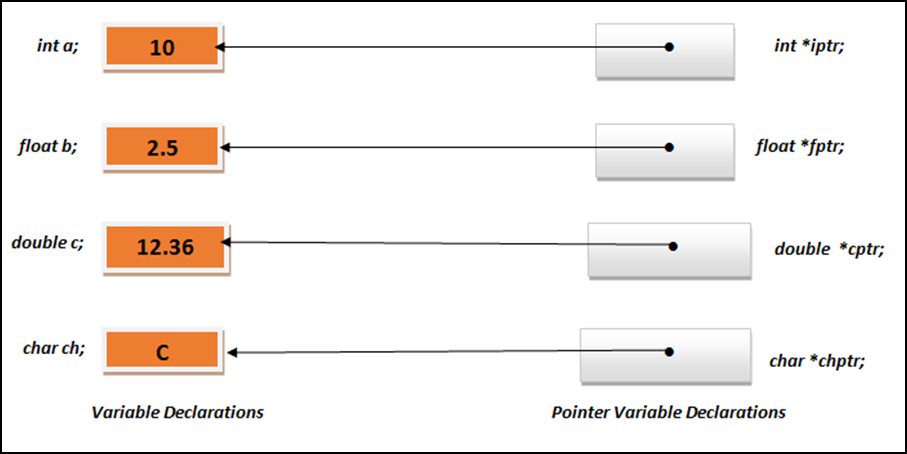
int a, \*iptr; float b, \*fptr; double c, \*cptr; char ch, \*chptr;

The memory layout during declaration phase is shown in Figure 1.2.2.



*Figure 1.2.2: Declaration of Pointer variables*

But when we assign the addresses of variables to the respective pointer variables, the memory layout will look the way shown in below figure 1.2.3



*Figure 1.2.3: Effect of indirect Access and Assignments of Pointers*

These initialized pointers may now be used to indirectly access the variables they are pointing.

**Pointer arithmetic**

As we perform arithmetic operations on regular integer variables, we can also perform arithmetic operations on pointer variables. Only addition and subtraction operations can be performed on pointer types.

But behaviour of addition and subtraction on pointer variables is slightly different. The operations behave differently according to the data type they are pointing to. The sizes of basic data types like integer, char, float etc. are already defined.

**Suppose we define 3 pointer variables as given below:**

**char \*cptr; short \*sptr; long \*lptr;**

Let us assume that they point to memory locations 4000, 5000 and 6000 respectively.

If we write an increment statement as given below, it

will increment the address contained in it. Thus, its value will become 5001.

**++cptr ;**

This is because **cptr** is a character pointer and character is of 1 byte. Thus, incrementing a character pointer will add 1 to the memory address.

**Similarly the statements,**

**++sptr ;** will increment the address contained in sptr by 2 bytes as short is 2 bytes in size and

**++lptr ;** will increment the address contained in lptr by 4 bytes as long is 4 bytes

in size.

Thus, when we increment a pointer, the pointer is made to point to the following element of the same type. Hence, the size in bytes of the type it points to is added to the pointer after incrementing it.

Same rules will follow for addition as well as subtraction operation. The below given statements give the same result as that of increment operator.

**cptr = cptr + 1; sptr = sptr + 1; lptr = lptr + 1;**

These increment (++) and decrement (--) operators can be used as either prefix or postfix operator in any expression. So in case of pointers, these operators can be used in similar way but with slight difference.

In case of prefix operator, the value is incremented first and then the expression is evaluated. In case of postfix operator, the expression is evaluated first and then the value is incremented.

Same rules follow for incrementing and decrementing pointers. As per the operator precedence rules, postfix operators, such as increment and decrement, have higher precedence than prefix operators, such as the dereference operator (\*). Thus, the following expression:

**\*ptr ++;**

is same as \*(ptr++). As ++ operator is used as postfix, the whole expression is evaluated as the value pointed originally by the pointer is then incremented.

**There are the four possible combinations of the dereference operator with both the prefix and suffix increment operators.**

* + - 1. \*ptr++ //equivalent to \*(ptr++)

//increment pointer ptr and dereference unincremented address

* + - 1. \*++ptr //equivalent to \*(++ptr)

//increment pointer ptr and dereference incremented address

3. ++\*ptr //equivalent to ++(\*ptr)

//dereference pointer and increment the value stored in it

4. (\*ptr)++ //equivalent to (\*ptr)++

//dereference pointer and post-increment the value stored in it

If we consider the following statement,

**\*ptr++ = \*qtr++;**

As ++ has a higher precedence than \*, both ptr and qtr pointers are incremented. Because both increment operators (++) are used as postfix operators, thus incrementing the value stored at address pointed by pointer ptr and qtr.

**Program:**

/\*Pointer Arithmetic\*/ #include<stdio.h>

int main()

{

int ivar = 5, \*iptr; char cvar = 'C', \*cptr;

float fval = 4.45, \*fptr;

iptr = &ivar; cptr = &cvar; fptr = &fval;

printf("Address of integer variable ivar = %u\n", iptr); printf("Address of character variable cvar = %u\n", cptr); printf("Address of floating point varibale fvar = %u\n\n", fptr);

/\* Increment\*/ iptr++; cptr++; fptr++;

printf("After increment address in iptr = %u\n", iptr); printf("After increment address in cptr = %u\n", cptr); printf("After increment address in fptr = %u\n\n", fptr);

/\* increment by 2\*/ iptr = iptr + 2; cptr = cptr + 2; fptr = fptr + 2;

printf("After +2 address in iptr = %u\n", iptr); printf("After +2 address in cptr = %u\n", cptr); printf("After +2 address in fptr = %u\n\n", fptr);

/\* Decrement\*/ iptr--;

cptr--;

fptr--;

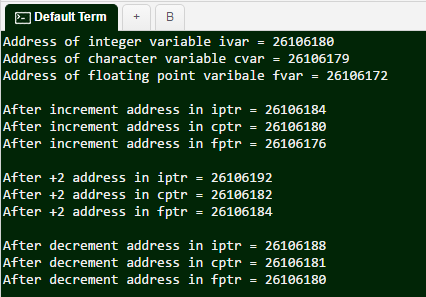
printf("After decrement address in iptr = %u\n", iptr); printf("After decrement address in cptr = %u\n", cptr);

printf("After decrement address in fptr = %u\n\n", fptr);

return 0;

}

**Output:**



 **Self-assessment Questions**

* + - * 1. Comment on following pointer declarations int \*ptr, p;.

ptr is a pointer to integers , p Is not

ptr and p both are pointers to integer

ptr is pointer to integer, p may or may not be

ptr and p both are not pointers to integer

* + - * 1. What will be the output? main()

{

char \*p;

p = "Hello"; printf("%c\n",\*&\*p);

}

Hello

H

1005 (memory address of variable p)

1008(memory address of character H)

* + - * 1. The statement int \*\*a;,

Is illegal b) Is legal but meaningless

c) Is syntactically and semantically correct d) Stacks

* + - * 1. Comment on the following, const int \*ptr;

We cannot change the value pointed by ptr.

We cannot change the pointer ptr itself.

Is illegal

We can change the pointer as well as the value pointed by it

* + 1. **Memory allocation functions**

Memory is a resource of computer system and it needs to be allocated properly for any kind of data structures used in programs. Dynamic memory allocation is a process of allocating memory to the data during program execution.

Normally when we are dealing with simple arrays or strings, we allocate the required amount of memory during compile time itself. We cannot extend the allocated memory during run- time. Hence, in such cases we need to allocate sufficient amount of memory at the compile time. But in compile time memory management, sometimes the allocated memory may not be used hence wasting the memory space.

Thus, we can make use of Dynamic memory allocation technique to allocate and de-allocate memory at runtime. Dynamic memory allocation helps us to increase or decrease the memory when the program is under execution.

**The following are the dynamic memory allocation functions in C:**

* + - 1. **malloc ()**

It Allocates requested size of bytes and returns a pointer of first byte of allocated space.

* + - 1. **calloc()**

It Allocates space for an array elements, initializes to zero and then returns a pointer to memory

* + - 1. **realloc()**

It deallocate the previously allocated space.

* + - 1. **free()**

We change the size of previously allocated space.

1. **malloc()**

malloc, as the name indicates, stands for memory allocation. This function reserves a block of memory of specified size to return a pointer of type void.

**Syntax of malloc()**

ptr=(cast-type\*)malloc(byte-size)

Here, ptr is pointer of cast-type. The malloc() function returns a pointer to an area of memory with size of byte size. If the space is insufficient, allocation fails and returns NULL pointer.

ptr=(int\*)malloc(100\*sizeof(int));

This statement will allocate either 200 or 400 according to size of int 2 or 4 bytes respectively and the pointer points to the address of first byte of memory.

1. **calloc()**

Calloc stands for "contiguous allocation". The difference between malloc() and calloc() is that, malloc() allocates single block of memory whereas calloc() allocates multiple blocks of memory each of same size and sets all bytes to zero.

Unless ptr is NULL, it must have been returned by an earlier call to malloc(), calloc() or realloc().

**Syntax of calloc()**

ptr=(cast-type\*)calloc(n,element-size);

This statement will allocate contiguous space in memory for an array of n elements.

***For example:***

ptr=(float\*)calloc(25,sizeof(float));

This statement allocates contiguous space in memory for an array of 25 elements each of size of float, *i.e.,* 4 bytes.

1. **free()**

This function is used to explicitly free the memory allocated by malloc() and calloc() functions. It releases all the memory reserved for program.

free(ptr);

1. **Realloc()**

Sometimes a programmer requires extra memory or allocated memory becomes more than sufficient. In these cases, a programmer can change memory size previously allocated using realloc().

**Syntax of realloc()**

ptr=realloc(ptr,newsize); Here, ptr is reallocated with size of newsize. ***For example:***

#include<stdio.h> #include<stdlib.h> int main()

{

int \*ptr,i,n1,n2;

printf("Enter size of array: "); scanf("%d",&n1); ptr=(int\*)malloc(n1\*sizeof(int));

printf("Address of previously allocated memory: "); for(i=0;i<n1;++i)

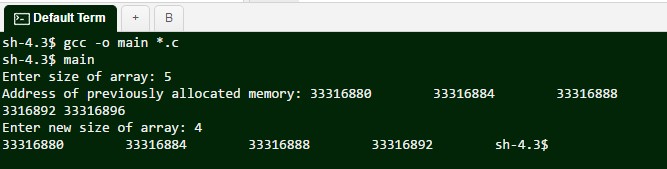
printf("%u\t",ptr+i); printf("\nEnter new size of array: "); scanf("%d",&n2);

ptr=realloc(ptr,n2); for(i=0;i<n2;++i)

printf("%u\t",ptr+i); return 0;

}

**Output:**



**Example showing use of malloc(), calloc() and free()**

**Program:**

#include<stdio.h> #include<stdlib.h> int main()

{

int n,i,\*ptr,sum=0;

printf("Enter number of elements: "); scanf("%d",&n);

ptr=(int\*)calloc(n,sizeof(int)); //memory allocated using calloc if(ptr==NULL)

{

printf("Error! memory not allocated."); exit(0);

}

printf("Enter elements of array: "); for(i=0;i<n;++i)

{

scanf("%d",ptr+i); sum+=\*(ptr+i);

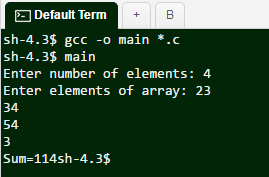
}

printf("Sum=%d",sum); free(ptr);

return 0;

}

**Output:**



 **Self-assessment Questions**

* + - * 1. What function should be used to free the memory allocated by calloc()?

dealloc(); b) malloc(variable\_name, 0)

c) free(); d) memalloc(variable\_name, 0)

* + - * 1. Which header file should be included to use functions like malloc() and calloc()?

memory.h b) stdlib.h

c) string.h d) dos.h

* + - * 1. How will you free the memory allocated by the following program? #include<stdio.h>

#include<stdlib.h> #define MAXROW 3

#define MAXCOL 4 int main()

{

int \*\*p, i, j;

p = (int \*\*) malloc(MAXROW \* sizeof(int\*)); return 0;

}

The name of array b) The data type of array

c) The first data from the set to be stored d) the index set of the array

* + - * 1. Specify the 2 library functions to dynamically allocate memory?

malloc() and memalloc() b) alloc() and memalloc()

c) malloc() and calloc() d) memalloc() and faralloc()

* + 1. **Recursion**

Recursion is considered to be the most powerful tool in a programming language. But sometimes, Recursion is also considered as the most tricky and threatening concept to a lot of programmers. This is because of the uncertainty of conditions specified by user.

In short Something Referring to itself is called as a Recursive Definition

1. **Definition**

Recursion can be defined as defining anything in terms of itself. It can be also defined as repeating items in a self-similar way.

In programming, if one function calls itself to accomplish some task then it is said to be a recursive function. Recursion concept is used in solving those problems where iterative multiple executions are involved.

Thus, to make any function execute repeatedly until we obtain the desired output, we can make use of Recursion.

**Example of Recursion:**

The best example in mathematics is the factorial function. n! = 1.2.3 (n-1).n

If n=6, then factorial of 6 is calculated as, 6! = 6(5)(4)(3)(2)(1)= 720

Consider we are calculating the factorial of any given using a simple. If we have to calculate factorial of 6 then what remains is the calculation of 5!

**In general we can say**

n ! = n (n-1)! (i.e., 6! = 6 (5!))

it means we need to execute same factorial code again and again which is nothing but Recursion.

**Thus, the Recursive definition for factorial is:**

f(n) = 1 if n=0

n \* f (n-1) otherwise

The above Recursive function says that the factorial of any number n=0 is 1, else the factorial of any other number n is defined to be the product of that number n and the factorial of one less than that number .

***For example,*** consider n=4

As n is not equal to 0, the first case will not satisfy. Thus, applying second case we get

4! = 4(4-1)! = 4(3!)

To find 3! Again we have to apply the same definition.

4! = 4(3!)=4[(3)(2!)]

Now, we have to calculate 2! , which requires 1! , which requires 0!.

As 0! is 1 by definition , we reach the end of it. Now we have to substitute the calculated values one by one in reverse order.

4!=4(3!)= 4(3)(2!)=4(3)(2)(1!)= 4(3)(2)(1)(0!)= 4(3)(2)(1)(1)= 24

Thus, 4!= 24

From the above solution it is clear that the each time we need to calculate the factorial of a value one less than the original one. Thus we reach value 0 where we have to stop applying same function of factorial.

**Any recursive definitions will have some properties. They are:**

* There are one or more base cases for which recursions are not needed.
* All cycles of recursion stops at one of the base cases.

We should make sure that each recursion always occurs on a smaller version of the original problem.

**In C Programming a recursive factorial function will look like:**

int factorial(int n)

{

if (n==0) //Base Case

return 1;

else

}

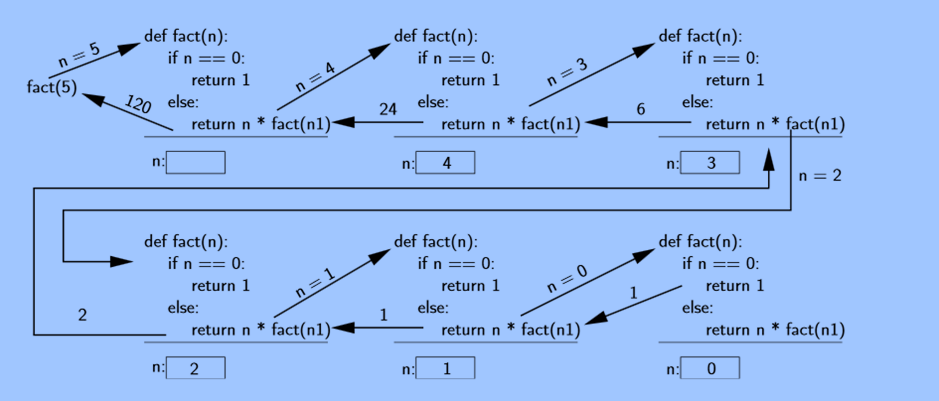
return n\*factorial (n-1); //Recursive Case

The above program is for calculating factorial of any number n. First when we call this factorial function, it checks for the base case. It checks if value of n equals 0. If n equals 0, then by definition it returns 1.

Otherwise it means that the base case is not yet been satisfied. Hence, it returns the product of n and factorial of n-1.

Thus, it calls the factorial function once again to find factorial of n-1. Thus forming recursive calls until base case is met.

Figure 1.2.4 shows the series of recursive calls involved in the calculation of 5!. The values of n are stored on the way down the recursive chain and then used while returning from function calls.



*Figure 1.2.4: Recursive computation of 5!*

1. **Advantages**

An important advantage of Recursion is that it saves time of programmer to a large extent. Even though problems like factorial, power or Fibonacci can be solved using loops but their

recursive solutions are shorter and easier to understand. And there are algorithms that are quite easy to implement recursively but much more challenging to implement using loops.

 **Advantages of Recursion:**

* + Reduce unnecessary calling of function.
  + Solving problems becomes easy when its iterative solution is very big and complex and cannot be implemented with loops.
  + Extremely useful when applying the same solution

1. **Recursive programs**
2. **Fibonacci series**

One of the well-known problems is generating a Fibonacci series using Recursion. A Fibonacci series looks like 0,1, 1, 2, 3, 5, 8, 13, 21and so on

**Working:** The next number is equal to sum of previous two numbers. The first two numbers of Fibonacci series are always 0 and. The third number becomes the sum of first 2 numbers, *i.e.,* 0 + 1 = 1. Similarly, the Fourth number is the sum of 3rd and 2nd number, *i.e.,* 1 + 1 = 3 and so on.

**Thus, the Recursive definition for Fibonacci is:**

0 if 0 <=k <n-1

Fk(n) = 1 if k=n-1

(𝑘−1) n i=k−n

∑ Fi

Otherwise

**In C Programming a recursive fibonacci function will look like:**

int fib(int n)

{

if (n <= 1)

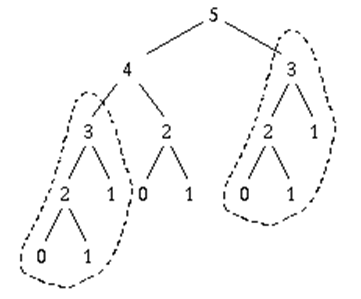
else

return n;

return fib(n - 1) + fib(n - 2);

}

If n is less than or equal to 1, then return n. Otherwise return the sum of the previous two terms in the series by calling fib function twice. Once for (n-1) and next for fib (n-2).This combines results from 2 separate recursive calls. This is sometimes known as "deep" recursion. The below figure 1.2.5 demonstrates the working of recursive algorithm for Fibonacci series.



*Figure 1.2.5: Recursive Algorithm*

***For example,*** the call to fib (4) repeats the calculation of fib (3) (see the circled regions of the tree). In general, when n increases by 1, we roughly double the work; that makes about 2n calls.

**Following is the c program for implementation of Fibonacci series:**

**Program:**

#include<stdio.h> int fib(int n)

{

if ( n == 0 ) return 0;

else if ( n == 1 ) return 1;

else

return ( fib(n-1) + fib(n-2) );

}

int main()

{

int n, j = 0, i;

printf("Fibonacci series implementation\n"); printf("How many terms in series: "); scanf("%d",&n);

printf("Fibonacci series\n"); for ( i = 1 ; i <= n ; i++ )

{

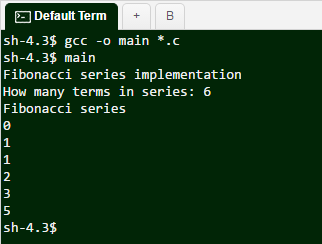
printf("%d\n",fib(j)); j++;

}

return 0;

}

**Output:**



The above program uses the recursion concept to print the Fibonacci series. The program first asks the total number of terms to be displayed as output. Then it makes recursive calls to the function fib() and finds the next term in the series by adding previous two values in the series.

1. **Binomial Coefficient**

Binomial coefficient C (n, k) counts the number of ways to form an unordered collection of k items selected from a collection of n distinct items

***For example,*** if you wanted to make a group of two from a group of four people, the number of ways to do this is C (4, 2).

Where, n=4 *i.e.,* 4 people and k=2 i.e. group of 2 people

There are total 6 ways to group them in an unordered manner. Let us assume 4 people as A, B, C and D

So the 2 letter groups are: AB, AC, AD, BC, BD, and CD Hence, C (n, k) = C (4, 2) = 6.

**In general, Binomial Coefficients can be defined as:**

* A binomial coefficient C (n, k) is the coefficient of X^k in the expansion of (1 + X)^n.
* A binomial coefficient C (n, k) also gives the number of ways, regardless of order, that k items can be chosen from among n items.

**Problem:**

This Problem of Binomial Coefficients can be implemented using Recursion. We need to write a function that takes two parameters n and k and returns the value of Binomial Coefficient C (n, k).

**Recursive function:**

The value of C(n, k) can recursively calculated using following standard formula for Binomial Coefficient’s.

C(n, k) = C(n-1, k-1) + C(n-1, k) C(n, 0) = C(n, n) = 1

Below given program implements the calculation of Binomial Coefficients in a Recursive Manner.

**Program:**

//Recursive implementation of Binomial Coefficient C(n, k)

#include<stdio.h>

int binomial(int n, int k)

{

if (k==0 || k==n) // Base Cases return 1;

else

return binomial(n-1, k-1) + binomial(n-1, k);

}

int main()

{

int n, k;

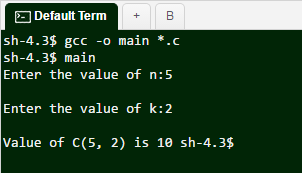
printf("Enter the value of n:"); scanf("%d",&n);

printf("\nEnter the value of k:"); scanf("%d",&k);

printf("\nValue of C(%d, %d) is %d ", n, k, binomial(n, k)); return 0;

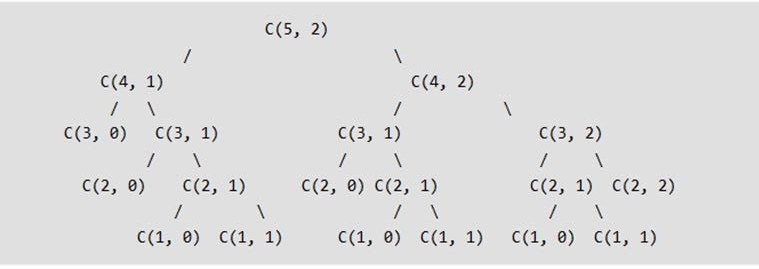
}

**Output:**



It should be noted that in the above program, the binomial function is called again and again until the base cases are satisfied.

Below given figure 1.2.6 is the Recursive tree for n=5 and k=2.



*Figure 1.2.6: Example of DP and Recursion*

1. **GCD (Greatest Common Divisor)**

The Greatest Common divisor of two or more integers is the largest positive integer that divides the numbers without a remainder.

***For example,*** the GCD of 8 and 12 is 4.

**Problem Definition:** Given any nonnegative integers a and b, considering both are not equal to 0, calculate gcd(a, b).

**Recursive Definition:**

For a,b ≥ 0, gcd(a,b) = a if b=0

**gcd(b, (a mod b)) otherwise**

**Input:** Any Nonnegative integers a and b, both not equal to zero.

**Output:** The greatest common divisor of a and b.

***For example:***

Consider a=54 and b=24. We need to find GCD (54, 24) Thus, the divisors of 54 are: 1, 2, 3, 6, 9, 18, 27, and 54

Similarly, the divisors of 24 are: 1, 2, 3, 4, 6, 8, 12, and 24 Thus, 1,2,3,6 are the common divisors of both 54 and 24:

The greatest number of these common divisors is 6.

That is, the GCD (greatest common divisor) of 54 and 24 is 6.

**The following program demonstrates computation of GCD using recursion:**

**Program:**

/\*GCD of Numbers using Recursion\*/ #include <stdio.h>

int gcd(int a, int b)

{

while (a != b)

{

if (a > b)

return gcd(a - b, b); else

return gcd(a, b - a);

}

return a;

}

int main()

{

int a, b, ans;

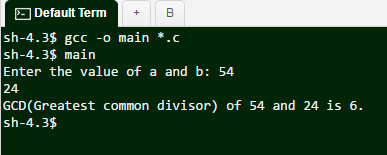
printf("Enter the value of a and b: "); scanf("%d%d", &a, &b);

ans = gcd(a, b);

printf("GCD(Greatest common divisor) of %d and %d is %d.\n", a, b, ans);

}

**Output:**





**Did you Know?**

One critical requirement of recursive functions is termination point or base case. Every recursive program must have base case to make sure that the function will terminate. Missing base case results in unexpected behaviour.

 **Self-assessment Questions**

* + - * 1. Which Data Structure is used to perform Recursion?

Queue b) Stack

c) Linked List d) Tree

* + - * 1. What is the output of the following code?

int doSomething(int a, int b)

{ if (b==1) return a; else

return a + doSomething(a,b-1);

}

doSomething(2,3);

4 b) 2

c) 3 d) 6

* + - * 1. Determine output of,

int rec(int num){

return (num) ? num%10 + rec(num/10):0;

}

main(){

printf("%d",rec(4567));

}

a) 4 b) 12

c) 22 d) 21

* + - * 1. What will be the below code output?

int something(int number)

{

if(number <= 0)

return 1;

else

}

return number \* something(number-1);

something(4);

a) 12 b) 24

c) 1 d) 0

* + - * 1. void print(int n),

{

if (n == 0) return;

printf("%d", n%2); print(n/2);

}

What will be the output of print(12)?

a) 0011 b) 1100

c) 1001 d) 1000

******Summary**

* A pointer is a value that designates the address (i.e., the location in memory), of some value. Pointers are variables that hold a memory location.
* ‘&’ - address of variable is used to assign address of any variable to pointer variable.
* ‘\*’ indirection operator is used to access the value contained in a particular pointer.
* Pointers store the address of any variable using & operator. We can access the value of that variable using a \* operator succeeded by the variable name.
* **Memory allocation function:**
  + Malloc() - Allocates requested size of bytes and returns a pointer first byte of allocated space
  + Calloc() - Allocates space for an array elements, initializes to zero and then returns a pointer to memory
  + Free() - deallocate the previously allocated space
  + Realloc() - Change the size of previously allocated space
* Recursion is the process of repeating items in a self-similar way. In Programs, if a function makes a call to itself then it is called a recursive function. Recursion is more general than iteration. Choosing between recursion and looping involves the considerations of efficiency and elegance.

 **Terminal Questions**

1. Explain the role of pointers in data structures.
2. What are the memory allocation functions? Explain in detail.
3. Define Recursive functions.
4. Write a note on indirection operator.

******Answer Keys**

|  |  |
| --- | --- |
| **Self-assessment Questions** | |
| **Question No.** | **Answer** |
| 1 | d |
| 2 | a |
| 3 | a |
| 4 | a |
| 5 | b |
| 6 | c |
| 7 | a |
| 8 | c |
| 9 | b |
| 10 | d |
| 11 | c |
| 12 | b |
| 13 | d |
| 14 | c |
| 15 | b |
| 16 | a |

**Stacks and Queues**

**Module Description**

This module introduces two closely-related data types for manipulating large collections of objects: stack and the queue. Each of them is basically defined by two simple operations: insert or add a new item, and remove an item. When we add a data item we have a clear intension. However, when we remove an item, we should decide which one to choose. For example, the rule used in case of queue is to always remove the item that has been in the queue for longest time. This policy is known as first-in-first-out or FIFO. And the rule used in case of stack is that we always remove the element that has been in the stack for least amount of time. This policy is known as last-in first-out or LIFO.

**Chapter 3.1**

Stacks

**Chapter 3.2**

Queue

**Chapter Table of Contents**

**Chapter 3.1**

**Stacks**

[Aim 139](#_bookmark71)

[Instructional Objectives 139](#_bookmark72)

[Learning Outcomes 139](#_bookmark73)

* + 1. [Introduction to Stack 140](#_bookmark74)
       1. [Definition of a Stack 141](#_bookmark75)
       2. [Array Representation of Stack 142](#_bookmark76)

[Self-assessment Questions 144](#_bookmark77)

* + 1. [Operations on Stack 144](#_bookmark78)

[Self-assessment Questions 149](#_bookmark79)

* + 1. [Polish Notations 149](#_bookmark80)
       1. [Infix Notation 150](#_bookmark81)
       2. [Prefix Notation 151](#_bookmark82)
       3. [Postfix Notation 152](#_bookmark83)

[Self-assessment Questions 155](#_bookmark84)

* + 1. [Conversion of Arithmetic Expression from Infix to Postfix 155](#_bookmark85)

[Self-assessment Questions 159](#_bookmark86)

* + 1. [Applications of Stack 160](#_bookmark87)
       1. [Balancing Symbol 160](#_bookmark88)
       2. [Recursion 161](#_bookmark89)
       3. [Evaluation of Postfix Expression 163](#_bookmark90)
       4. [String Reversal 163](#_bookmark91)

[Self-assessment Questions 165](#_bookmark92)

[Summary 166](#_bookmark93)

[Terminal Questions 167](#_bookmark94)

[Answer Keys 168](#_bookmark95)

[Activity 169](#_bookmark96)

[Case Study 170](#_bookmark97)

[Bibliography 171](#_bookmark98)

[e-References 171](#_bookmark99)

[External Resources 171](#_bookmark100)

[Video Links 171](#_bookmark101)

 **Aim**

To educate and equip the students with skills and technologies of Stacks

******Instructional Objectives**

After completing this chapter, you should be able to:

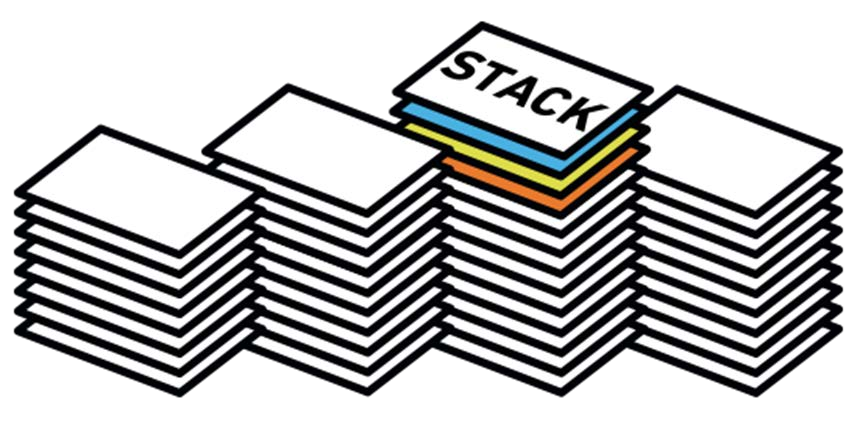
* Outline the basic features of stack
* Describe the array representation of stack
* Explain the Polish Notations with example
* Discuss the evaluation of postfix expression using stack
* Explain the steps to convert Infix expression to postfix expression and vice versa
* Outline the applications of stacks

******Learning Outcomes**

At the end of this chapter, you are expected to:

* Explain operations on stack
* Convert given expressions of infix to prefix and postfix expression
* Explain string recursion applications of stack
* Compute given postfix expression using stack
* Convert to prefix expression for any given infix expression
  + 1. **Introduction to Stack**

In this chapter we will introduce a data structure for representing stack as a limited access data structure. Stack data structure is used for manipulating arbitrarily large collections of data. The stack is a data structure which represents objects maintained in a particular order. In this chapter we also explain how to operate a stack data structure. This chapter demonstrates the operations for creating a stack, adding elements to a stack, deleting an element form a stack etc. Some problems has solutions that require the data associated to be arranged or organized as linear list of data elements in which operations are permitted to take place at only one end of the list. The best and the simplest examples are set of books kept one on top of another, set of playing cards, pancake, arranging laundry, stacked plates one above another, etc. Here, we group things together by placing one thing on top of another and then we have to remove things from top to bottom one at a time. The below figure 3.1.1 shows a set of books represented as a stack.



*Figure 3.1.1: Picture Representing a Stack*

It is interesting that something that is so simple is a critical part of nearly every program that is written. The nested function calls in a running program, conversion of an infix form of an expression to an equivalent postfix or prefix, computing factorial of a number, and so on can be accurately formulated using this simple technique. In all the above cases, it is clear that the one which most recently entered into the list is the first one to be operated. Solution to these types of problems is based on the principle Last-In-First-Out (LIFO) or First-In-Last-Out. A logical structure, which organizes the data and performs operations in LIFO or FILO principle, is termed as a Stack.

1. **Definition of a Stack**

Stack is an ordered list of similar data items in which operations such as insertion and deletion are permitted to be done only at one end called top of the stack. It is a linear data structure in which operations can be performed on data objects on principle of Last-In-First-Out or First- In-Last-Out.

More formally, a stack can be defined as an abstract data type with domain of data objects and a set of functions that can be performed on data objects guided by list of axioms.

**Some of the important functions used while doing operations of stacks are listed below:**

* 1. **Create-Stack() -** Used for allocating memory
  2. **Isempty(S) -** Used for checking if stack is empty or not; it returns a Boolean
  3. **Isfull() -** Used for checking if stack is full; this also return Boolean
  4. **Push(S,e) -** Use to add an element on top of stack
  5. **Pop(s) -** Used to remove an element from the top of the stack
  6. **Top(S) -** Used to display an element in the stack

Also, some axioms are needed to be known while we do operations on stacks. Following is a list of axioms which a programmer must know:

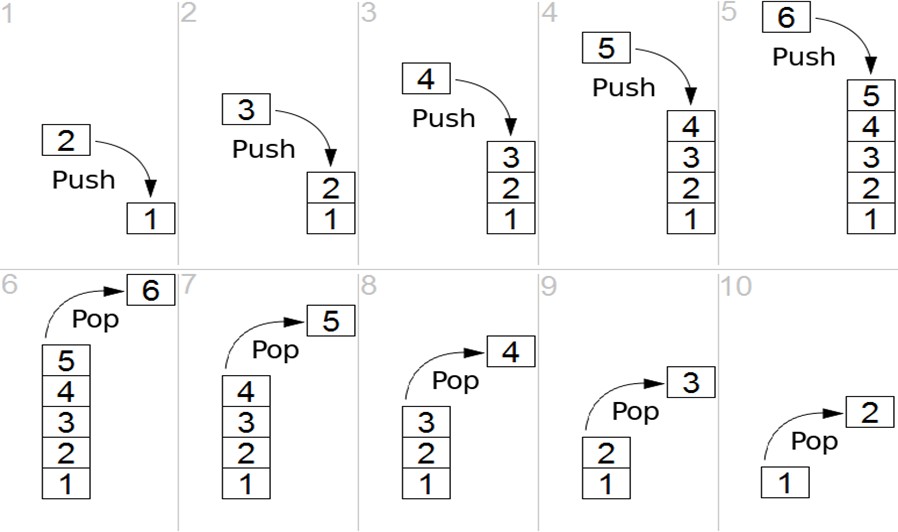
* **Isempty(Create-Stack()):** Always returns true value
* **Isfull(Create-Stack()):** Always returns false values
* **Isempty(Push(S, e)):** Always returns false value
* **Isfull(Pop(S)):** Always returns false values
* **Top(push(S, e)):** The element e is displayed
* **Pop(Push(S, e):** The element e will be removed from stack

The detailed explanation and algorithm for implementing above operations will be covered in forthcoming sections. The figure 3.1.2 demonstrates push() and pop() operations performed on stack.

As shown in the figure, initially stack contains element 1. To push element 2, stack pointer is incremented and then element 2 is pushed. So now stack contains 2 elements *i.e.,* 1 and 2.

In the second step, we push element 3 onto the stack. Thus this element will be placed on top of 2 as the top of stack is pointing one location above 2. Similarly elements 4, 5 and 6 are pushed onto the stack. After pushing element 6 the stack contains total 6 elements.

In the second part of the figure, pop instructions are executed. The first element we can read out is 6 as it is on the top of the stack. And the stack pointer is decremented. Next time if we execute a pop instruction, element 5 will be removed and so on until last element 1 is removed. Thus,



*Figure 3.1.2: Push() and Pop() Operations*

1. **Array Representation of Stack**

As we know a stack is a data structure designed to store collection of data where the data can be added and removed from only one end. We can implement this stack using a simple linear array. As array is a collection of similar kinds of elements. We can create a stack using a one dimensional array very easily.

***For example,*** we can declare an array named stack [] to store all the data elements of a stack. Normally, elements in a linear array can be accessed in any random way by using array name and its index.

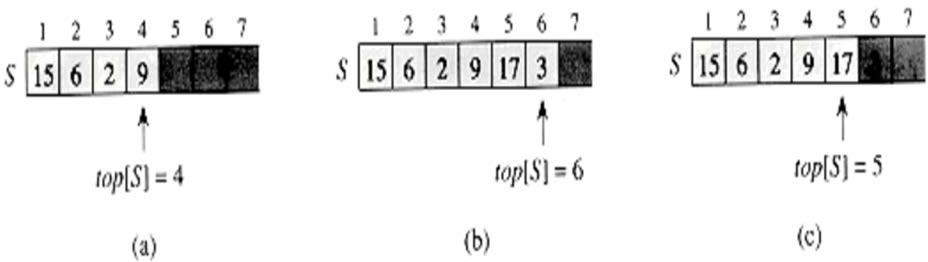
But a stack is operating from only one end. Thus we when a stack is implemented as a array, we should allow insertion and deletion of elements from only one end of array.

Thus, a variable named “top” will keep track of the position of the topmost element in that stack. This variable is also called as stack pointer.

Initially the value of “top” is assigned to -1, as the stack is empty. When we push an element onto the stack, we need to increment the stack pointer by one and then insert the element at a position where stack pointer is pointing. For every push operation we have to check if stack pointer has reached the maximum size of the array stack [].

Similarly, when we perform a pop operation, the stack pointer should be decremented by 1. We should also check for a condition to see whether the stack array is empty or no.

**Figure 3.1.3 demonstrates the array representation of stack**



*Figure 3.1.3: Array Representation of Stack*

As shown in the above figure, an array named S with size 7 is declared which acts as a stack. Thus we can store total of 7 elements in this stack.

In part (a) of the figure, after adding elements 15, 6, 2, and 9, the stack pointer is pointing location 4.

In part (b), we have pushed two more elements 17 and 3 making stack pointer to have value 6.

In part (c), it shows how a pop operation is carried out on that array causing last element 3 to be popped out. Thus the stack pointer is decremented by 1 after pop operation.

 **Self-assessment Questions**

1. A stack is works on the principle of
   1. First in first out (FIFO) b) Last in last out (LILO)

c) First in last out (FILO) d) Cyclical data structures

1. The difference between linear array and stack is that any elements can randomly be accessed.
   1. True b) False
2. Top[s] returns
   1. Stack bottom b) Stack top

c) Stack mid d) Any random element

* + 1. **Operations on Stack**

The operations discussed in the topics above are explained in detail this in this section. Basically, stack operations may include, initializing a stack, using it storing data based on different applications and again de-initializing it. Apart from these basic things, stack is used for carrying our following two operations:

* + - 1. **Push() –** storing data item in to the stack
      2. **Pop() –** Deleting a data item from the stack

Consider the operation of pushing a data on to the stack. In order to use stack most efficiently, we need to aware about the status if the stack. For this purpose, following functions are importing.

1. **stacktop() –** This function is used for displaying the topmost element in the stack
2. **isFull() –** This function is used to check if the stack is already full
3. **isEmpty() –** This function is used to check if stack is empty.

Throughout, we must maintain a pointer to the most recent pushed data on the stack. This pointer always represents the top of the stack and hence is named as top.

Before we proceed to implement push () operation, we must first lean the procedure for these support functions.

**Algorithm for top() function**

begin procedure stacktop return stack[top]

end procedure

Implementation in C programming

int stacktop()

{

return stack[top];

}

**Algorithm for isFull() function**

begin procedure isfull iftop equals to MAXSIZE returntrue

else returnfalse

endif

end procedure

Implementation in C programming

bool isfull()

{

if(top == MAXSIZE) returntrue;

else returnfalse;

}

**Algorithm for isempty() function**

begin procedure isempty

iftop less than 1 returntrue

else returnfalse

endif

end procedure

Implementation in C programming

bool isempty()

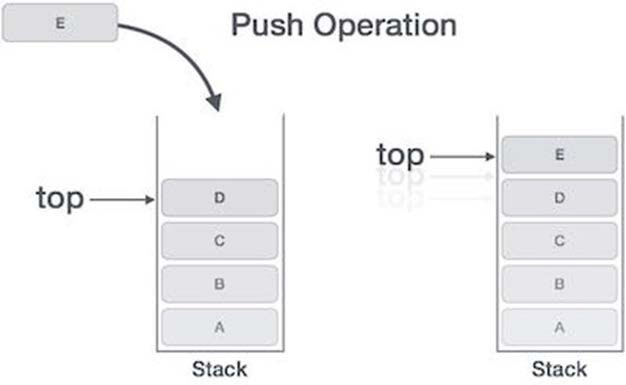
{

if(top ==-1) returntrue; else returnfalse;

}

Now, to get back to push operation, we must first understand the process how push() function works. Following steps are involved:

* **Step 1:** Check if stack is full.
* **Step 2:** If stack is full then display an error and exit.
* **Step 3:** If stack is not full, increment top to point to next empty space.
* **Step 4:** Add the element on to the stack, where top is pointing.
* **Step 5:** Return



*Figure 3.1.4: Push Operation on Stack*

**Note:** If linked list is used for stack implementation, then memory space needs to be allocation in step 3.

**Following is the algorithm for push operation** begin procedure push: stack, data if stack is full

returnnull endif

top ← top +1 stack[top]← data

end procedure

**And the corresponding C program function is also shown below**

void push(int data)

{

if(!isFull())

{

}else

{

}

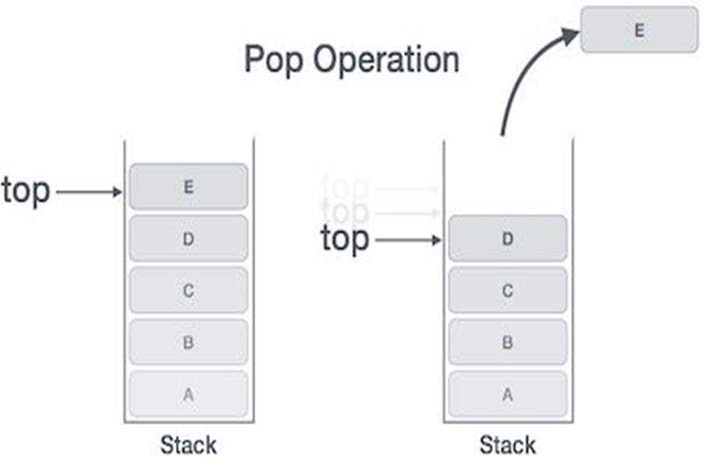
}

top = top +1; stack[top]= data;

printf("Could not insert data, Stack is full.\n");

Now, we move on to pop operation. Accessing the data element while removing it from the stack is called pop operation. Following are the steps involved in process of popping out an element from the stack.

* **Step 1 −** Check if stack is empty.
* **Step 2 −** If stack is empty, produce error and exit.
* **Step 3 −** If stack is not empty, access the data element at which top is pointing.
* **Step 4 −** Decrease the value of top by 1.
* **Step 5 −** return



*Figure 3.1.5: Pop Operation on Stack*

**Algorithm for implementation of pop operation**

begin procedure pop: stack if stack is empty returnnull

endif

data ← stack[top] top ← top -1

return data end procedure

**Corresponding C program function**

int pop(int data)

{

if(!isempty())

{

data = stack[top]; top = top -1;

return data;

}else

{

printf("Could not retrieve data, Stack is empty.\n");

}

}

 **Self-assessment Questions**

1. Match the following

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | stacktop() | A | Used for checking if stack is empty |
| 2 | Isfull() | B | Used for displaying topmost element |
| 3 | Isempty() | C | Used for checking if stack is full |

1. What push(x) does to stack
   1. Removes x from stack b) Add x to topmost element

c) Add x to all the elements d) Add x to top of stack

1. What pop() does to stack
   1. Removes x from stack b) Add x to topmost element

c) Add x to all the elements d) Add x to top of stack

* + 1. **Polish Notations**

First we need to understand Arithmetic Expressions. An arithmetic expression is an expression which when evaluated, results in a numeric value. The method of writing arithmetic expression is known as a Notation. Same Arithmetic Expression can be written in different ways without changing the essence or meaning of that expression.

**Consider an expression:**

(5-6)\*7

It can be written in its ***infix form*** as “\*(-5 6)7”. In this case all the arithmetic operators are binary in nature, thus bracketing is not necessary. The above expression can be also written as

\* - 5 6 7

Consider an expression “1+2”, which adds the values 1 and 2. Its prefix notation, the operators precedes the operands, thus it will be “+ 1 2”.

The product calculation depends upon the availability of two operands *i.e.,* 5-6 and 7.

Normally, the innermost expressions are evaluated first. But, in case of prefix notation operators are written ahead of operands.

Thus infix notation with parenthesis will look like 5 – (6 \* 7)

Or without parenthesis it will be 5 – 6 \* 7

It would change the semantics or meaning of the expression because of precedence rule. Similarly, ***Polish notation*** of

5 – (6 \* 7)

Will be

– 5 \* 6 7

**Polish notation**

Polish notation, also called Polish prefix notation or prefix notation is a symbolic logic invented by Polish mathematician Jan Lukasiewicz. It is a form of notation for logic, arithmetic, and algebra. In prefix notations, the operators are placed to the left of their operands. If the operator’s parity is fixed, the result is a syntax lacking parentheses or other brackets that can still be parsed without any problem. The term Polish notations also include Polish postfix notation, or Reverse Polish notation, in which the operators are placed after the operands.

1. **Infix Notation**

As already discussed in the previous section, infix notation is the most common and simplest notation in which an operator is placed between two operands. This notation is also known as general form of arithmetic expression. ***For example,*** if arithmetic expression for adding two operands can be written in infix form as

A + B

In this ***example*** A and B are two operands and + is the operator.

Another ***example*** of infix expression is A + B \* C + (E – G)

These expressions follow a normal arithmetic precedence rule. ***For example,*** to evaluate the above expression, the first precedence is given to multiplication. So the product of B and C will be calculated first. Second precedence will be given to parenthesis. Therefore the result of E – G will be calculated and then A, result of product of B and C, and subtraction result of E and G will be added together.

1. **Prefix Notation**

This is also called as a polish method. When using this method, operator precedes operands i.e. instruction precedes data. Here the order of operations and operands determines the result, making parenthesis unnecessary. Taking the ***example***, consider infix expression 3 (4 + 5). This could be expressed as

\* 3 + 4 5

This is in contrast with the traditional algebraic methodology for performing mathematical operations, order of operation. In the expression 3(4+5), we first work inside the parentheses to add four plus five and then multiply the result by three.



**Did you know?**

In the olden days of the calculator, the end-user would write down the results of every step when using the algebraic Order of Operations. Not only did this slow things down, it provided an opportunity for the end-user to make errors and sometimes defeated the purpose of using a calculating machine. In the 1960's, engineers at Hewlett-Packard decided that it would be easier for end-users to learn Jan Lukasiewicz' logic system than to try and use the Order of Operations on a calculator. They modified Jan Lukasiewicz's system for a calculator keyboard by placing the instructions (operators) after the data. In homage to Jan Lukasiewicz' Polish logic system, the engineers at Hewlett-Packard called their modification reverse Polish notation (RPN).

1. **Postfix Notation**

Just opposite to prefix notation is postfix notation. Here operands precedes operator or operator is placed after operands and hence it is called postfix notation. It is also called as reverse polish expression. The infix expression A+B can be written in postfix as AB+.

Below are some of the ***examples*** of expressions represented in all three notations.

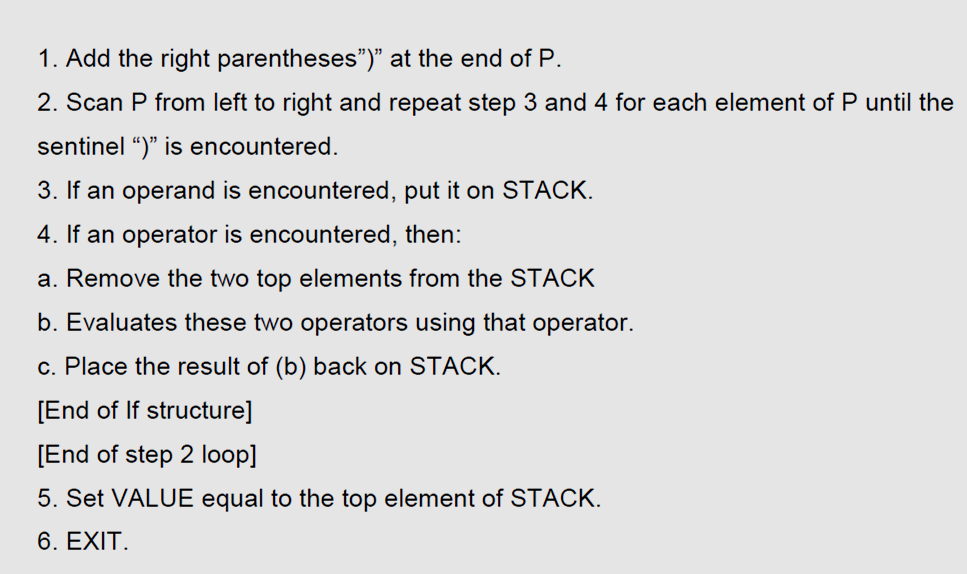
|  |  |  |
| --- | --- | --- |
| **Infix** | **Prefix** | **Postfix** |
| A+B | +AB | AB+ |
| A+B\*C | +A\*BC | ABC\*+ |
| (A+B)\*(C-D) | \*+AB-CD | AB+CD-\* |

**Algorithm for evaluation of postfix expression**

Consider a string of postfix arithmetic expression of operands and operators. Below given below should be followed for evaluation of a postfix expression:

* **Step 1:** Scan the string from left to right.
* Skip all the operands and values.
* If an operator is found, perform the operation on preceding two operands.
* Now replace these (2 operands and an operator) with one operand *i.e.,* the result of operation.
* Continue the process until single value remains, which is the result of the expression.

**Algorithm**



**Program:**

#include<string.h> #include<stdlib.h> #define MAX 50

int stack[MAX]; char post[MAX]; int top=-1;

void pushstack(int tmp); void calculator(char c); void main()

{

int i;

printf("Insert a postfix notation :: "); gets(post);

for(i=0;i<strlen(post);i++)

{

if(post[i]>='0' && post[i]<='9')

{

pushstack(i);

}

if(post[i]=='+' || post[i]=='-' || post[i]=='\*' || post[i]=='/' || post[i]=='^')

{

calculator(post[i]);

}

}

printf("\n\nResult :: %d",stack[top]);

}

void pushstack(int tmp)

{

top++;

stack[top]=(int)(post[tmp]-48);

}

void calculator(char c)

{

int a,b,ans; a=stack[top]; stack[top]='\0'; top--; b=stack[top]; stack[top]='\0'; top--;

switch(c)

{

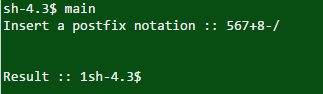
case '+': ans=b+a; break; case '-': ans=b-a; break; case '\*': ans=b\*a; break; case '/': ans=b/a; break; case '^': ans=b^a; break; default: ans=0;

}

top++; stack[top]=ans;

}

**Output:**



 **Self-assessment Questions**

1. In prefix notation follows the operands. (fill in the blank)
2. A+B is an infix expression
   1. True b) False
3. Postfix notation of A+B is

a) +AB b) A+B

c) AB+ d) ++A

* + 1. **Conversion of Arithmetic Expression from Infix to Postfix**

Let X is an arithmetic expression in its infix form. X is an expression containing operators, operands, parenthesis etc. We have 5 basic operators in mathematics namely addition, subtraction, multiplication, division and exponentiation.

**The order of precedence is**

* Exponentiation (Highest Precedence)
* Multiplication/division
* Addition/subtraction (Lowest Precedence)

Consider that all the operators including exponentiations are on the same level, performed from left to right unless indicated by the parentheses.

Below given algorithm transforms any infix expression X into its equivalent postfix expression

Y. We use stack data structure to store the operators and parenthesis.

**Algorithm:**

1. Read token from Left to Right in a given infix expression X and Postfix expression Y is generated.
2. Input infix Expression may have following tokens:
   1. Any Alphabet from A-Z or a-Z
   2. Any Number from 0-9
   3. Any Operator
   4. Opening And Closing Braces ( , )
3. If token read is Alphabet:
   1. Print that Alphabet as Output
4. If token read is Digit:
   1. Print that Digit as Output
5. If token read is Opening Bracket “(” :
   1. Push opening bracket ‘(’ Onto the Stack
   2. If any Operator appears before ‘)’ then Push it onto Stack.
   3. If Corresponding ‘)’ bracket appears then Start pop elements from Stack till ‘(’ is popped out.
6. If token read is Operator:
   1. Check if there is any Operator already present in Stack.
   2. If Stack is empty, Push Operator onto the Stack.
   3. If operator is present, check if Priority of Incoming Operator is greater than Priority of Topmost Stack Operator.
   4. If Priority of incoming Operator is Greater, push Incoming Operator Onto Stack.
   5. Else Pop Operator from Stack, repeat Step 6.

**Example of converting an expression from infix to postfix**

**Infix expression: A \* B + C**

The order in which the operators appear is not reversed. When the '+' is read, it has lower precedence than the '\*', so the '\*' must be printed first.

We will show this in a table with three columns. The first will show the symbol currently being read. The second will show what is on the stack and the third will show the current contents of the postfix string. The stack will be written from left to right with the 'bottom' of the stack to the left.

|  |  |  |  |
| --- | --- | --- | --- |
| Step | Current Symbol | Stack | Postfix expression |
| 1. | A |  | A |
| 2. | \* | \* | A |
| 3. | B | \* | AB |
| 4. | + | + | AB\* |
| 5. | C | + | AB\*C |
| 6. |  |  | AB\*C+ |

**Step 1:** The first input token is an alphabet “A”. Thus it is printed as output character of postfix notation.

**Step 2:** Next token in the infix expression is an operator “\*”. Thus it is pushed onto the top of the stack if the stack is empty.

**Step 3:** The third token in infix expression is an alphabet “B”, hence it is printed as an output character of postfix notation.

**Step 4:** Fourth input token is again an operator “+”. But the operator on the top of the stack i.e. “\*” has higher precedence as compared to operator “+”. Thus the operator “\*” is popped out from top of stack and printed as output of postfix notation. Push the operator “+” on to the top of the stack now.

**Step 5:** Next input character is an alphabet “C”. Thus, printed as an output character of postfix notation.

**Step 6:** now it is the end of the infix expression, thus we need to pop out all the operators from the stack one by one and printed as postfix notation. Thus operator “+” is printed as the last character of the postfix notation.

**Thus the Postfix expression is AB\*C+**

**Program:**

#include<stdio.h> #include<ctype.h> char stack[20]; int top = -1; void push(char x)

{

stack[++top] = x;

}

char pop()

{

if(top == -1)

return -1;

else

return stack[top--];

}

int priority(char x)

{

if(x == '(')

return 0;

if(x == '+' || x == '-') return 1;

if(x == '\*' || x == '/') return 2;

}

int main()

{

char exp[20]; char \*e, x;

printf("Enter the expression :: "); scanf("%s",exp);

e = exp;

while(\*e != '\0')

{

if(isalnum(\*e))

printf("%c",\*e);

else if(\*e == '(') push(\*e);

else if(\*e == ')')

{

}

else

{

while((x = pop()) != '(') printf("%c", x);

while(priority(stack[top]) >= priority(\*e)) printf("%c",pop());

push(\*e);

}

e++;

}

while(top != -1)

{

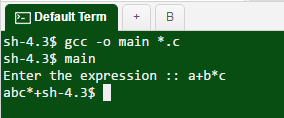
printf("%c",pop());

}

return 0;

}

**Output:**



 **Self-assessment Questions**

1. As per the algorithm to convert infix to postfix expression, we must ignore parenthesis present in infix expression
   1. True b) False
2. While converting an infix expression to postfix expression, if an operator is encountered, the operators are .
   1. Pushed on to the stack b) Popped out of stack

c) Left without doing anything d) Checked for precedence level

1. When the string scanning ends, next operation is .
   1. Popping out all operators from stack and adding them to postfix string
   2. Exit and print result
   3. Push all the operands on to the stack
   4. Do nothing
      1. **Applications of Stack**

Stacks have many useful applications in computer science. Stack form a base for many of the compilers for programming languages and sometimes is also core part of low lever programming languages like MATLAB and other assembly level languages. Some of the basic and most frequently used applications are described in the section below.

1. **Balancing Symbol**

We always do syntax mistakes while typing programs. The compilers duty is to check the programs for all the syntax errors. Most of the times, we make mistakes in typing brackets or parenthesis or any operators. Lack of any one symbol may cause multiple errors in the program. Thus real error remains unidentified.

Hence a stack can be used to check if the expressions in the programs are balanced. Thus, every right bracket, parenthesis or braces must end with corresponding left counterparts.

***For example,*** the sequence [()] is correct, however [(]) is invalid. As of now, consider a problem just check for balancing of parentheses, brackets, and braces and ignore other characters. A Stack can be used to balance symbols in a program. Following are the steps to do the same:

* 1. Create an empty stack s[].
  2. Scan the program file character by character till the end of file.
  3. Upon identifying any symbol (parenthesis, brace, bracket etc.), push it on to the stack.
  4. If stack is empty and scanned character is close bracket, brace, parenthesis etc., print an error message.
  5. Else pop element from stack
  6. If popped element is not corresponding open symbol, print an error message.
  7. If stack is not empty at the end of file, print an error message.

This is clearly linear and actually makes only one pass through the input. It is thus on-line and quite fast.

1. **Recursion**

Recursion is considered to be the most powerful tools in a programming language. But sometimes Recursion is also considered as the most tricky and threatening concept to a lot of programmers. This is because of the uncertainty of conditions specified by user.

In short Something Referring to itself is called as a Recursive Definition

Recursion can be defined as defining anything in terms of itself. It can be also defined as repeating items in a self-similar way.

In programming, if one function calls itself to accomplish some task then it is said to be a recursive function. Recursion concept is used in solving those problems where iterative multiple executions are involved.

Thus, to make any function execute repeatedly until we obtain the desired output, we can make use of Recursion.

**Example of Recursion:**

The best example in mathematics is the factorial function. n! = 1.2.3 (n-1).n

If n=6, then factorial of 6 is calculated as 6! = 6(5)(4)(3)(2)(1)= 720

Consider we are calculating the factorial of any given using a simple. If we have to calculate factorial of 6 then what remains is the calculation of 5!

In general we can say

n ! = n (n-1)! (*i.e.,* 6! = 6 (5!))

It means we need to execute same factorial code again and again which is nothing but Recursion.

***Thus the Recursive definition for factorial is:***

f(n) = 1 if n=0

n \* f (n-1) otherwise

The above Recursive function says that the factorial of any number n=0 is 1, else the factorial of any other number n is defined to be the product of that number n and the factorial of one less than that number .

**Any recursive definitions will have some properties. They are:**

* 1. There are one or more base cases for which recursions are not needed.
  2. All cycles of recursion stops at one of the base cases.

We should make sure that each recursion always occurs on a smaller version of the original problem.

**In C Programming a recursive factorial function will look like:**

int factorial(int n)

{

if (n==0) //Base Case

return 1;

else

}

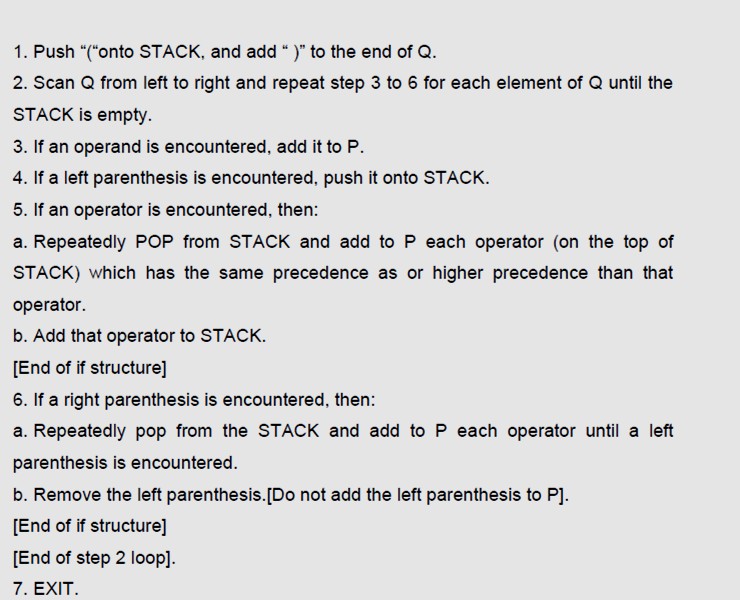
return n\*factorial (n-1); //Recursive Case

The above program is for calculating factorial of any number n. First when we call this factorial function, it checks for the base case. It checks if value of n equals 0. If n equals 0, then by definition it returns 1.

Otherwise it means that the base case is not yet been satisfied. Hence it returns the product of n and factorial of n-1.

Thus it calls the factorial function once again to find factorial of n-1. Thus forming recursive calls until base case is met.

1. **Evaluation of Postfix Expression**



1. **String Reversal**

Since stack is LIFO data structure, it becomes of obvious use in application where there is requirement of reversing a string or for checking if a string is a palindrome or not. The simplest way to reverse a string is, scan a string from left to right and push every character on to the stack until we reach the end of the string. Once we reach the end of the string, start popping out elements from the stack and create a new string of popped elements. Repeat the process of popping from stack until stack becomes empty.

/\*Program of reversing a string using stack \*/ #include<stdio.h>

#include<string.h> #include<stdlib.h> #define MAX 20

int top = -1; char stack[MAX]; char pop();

void push(char); int main()

{

char str[20]; unsigned int i;

printf("Enter the string : "); gets(str);

/\*Push characters of the string str on the stack \*/ for(i=0;i<strlen(str);i++)

push(str[i]);

/\*Pop characters from the stack and store in string str \*/ for(i=0;i<strlen(str);i++)

str[i]=pop(); printf("Reversed string is : "); puts(str);

return 0;

}/\*End of main()\*/ void push(char item)

{

if(top == (MAX-1))

{

printf("Stack Overflow\n"); return;

}

stack[++top] =item;

}/\*End of push()\*/ char pop()

{

if(top == -1)

{

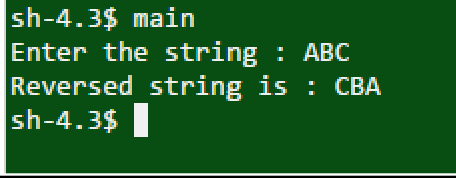
printf("Stack Underflow\n"); exit(1);

}

return stack[top--];

}/\*End of pop()\*/

**Output:**



**Self-assessment Questions**

1. Balancing symbol is useful for .
   1. Compiler optimization
   2. Inserting symbols in program code
   3. Inserting comments
   4. Checking precedence of operator
2. In Stack winding phase of recursion involves popping out instructions from stack
   1. True b) False
3. What is the evaluation of postfix expression 4 6 + 7 -
   1. 7 b) 4

c) 3 d) 8

 **Summary**

* Stacks are Last-In-First-Out (LIFO) data structures in which the most recent element inserted in the stack is the first one to be removed.
* The stack can be implemented using an array by creating a stack pointer variable for keeping track of top position.
* Push () and pop () are the primary operations possible on stacks for insertion and deletion of elements along with some support functions.
* Polish notation which is also called as prefix notation or simply prefix notation is a form of notation for logic, arithmetic, and algebra.
* Infix notation is the most common and simplest notation in which an operator is placed between two operands.
* In prefix notation, operator precedes operands and in postfix operands precedes the operator.
* Stacks can be used for evaluating a postfix expression, also in Recursion, String reversal etc.

 **Terminal Questions**

1. Explain stack and its basic operations.
2. Explain the algorithm for push and pop operations.
3. Write a C program for converting infix expression to postfix expression.
4. Explain applications of stack in brief.

******Answer Keys**

|  |  |
| --- | --- |
| **Self-assessment Questions** | |
| **Question No.** | **Answer** |
| 1 | c |
| 2 | b |
| 3 | b |
| 4 | 1 –b,2-c,3-a |
| 5 | b |
| 6 | a |
| 7 | Operator |
| 8 | a |
| 9 | c |
| 10 | a |
| 11 | a |
| 12 | a |
| 13 | a |
| 14 | b |
| 15 | c |

 **Activity**

**Activity Type**: Offline **Duration:** 15 Minutes

**Description**:

Divide the students into 4 groups.

Below are 4 infix expression, assign an expression to each group.

Each group should convert the given expression to postfix and prefix expression using stack. a) 3+4\*5/6

b) 6 \* (77 + 8 \*15) + 20

c) (300+23)\*(43-21)/(84+7)

d) (4+8)\*(6-5)/((3-2)\*(2+2))

**Case Study**

**Stack based memory allocation**

Stacks in computing architectures are regions of memory where data is added or removed in a last-in-first-out (LIFO) manner.

In most modern computer systems, each thread has a reserved region of memory referred to as its stack. When a function executes, it may add some of its state data to the top of the stack; when the function exits it is responsible for removing that data from the stack. At a minimum, a thread's stack is used to store the location of function calls in order to allow return statements to return to the correct location, but programmers may further choose to explicitly use the stack. If a region of memory lies on the thread's stack, that memory is said to have been allocated on the stack.

Because the data is added and removed in a last-in-first-out manner, stack-based memory allocation is very simple and typically faster than heap-based memory allocation (also known as dynamic memory allocation). Another feature is that memory on the stack is automatically, and very efficiently, reclaimed when the function exits, which can be convenient for the programmer if the data is no longer required. If however, the data needs to be kept in some form, then it must be copied from the stack before the function exits. Therefore, stack based allocation is suitable for temporary data or data which is no longer required after the creating function exits.

A thread's assigned stack size can be as small as only a few bytes on some small CPU's. Allocating more memory on the stack than is available can result in a crash due to stack overflow.

Some processor families, such as the x86, have special instructions for manipulating the stack of the currently executing thread. Other processor families, including PowerPC and MIPS, do not have explicit stack support, but instead rely on convention and delegate stack management to the operating system's application binary interface (ABI).

**Questions:**

1. Explain how stack based memory allocation worked.
2. What are the advantages of stack based memory allocation?

**Bibliography**

 **e-Reference**

* + bowdoin.edu, (2016). Computer Science 210: Data Structures. Retrieved on 19

April 2016, from [http://www.bowdoin.edu/~ltoma/teaching/cs210/fall10/Slides/StacksAndQueues.](http://www.bowdoin.edu/~ltoma/teaching/cs210/fall10/Slides/StacksAndQueues) pdf

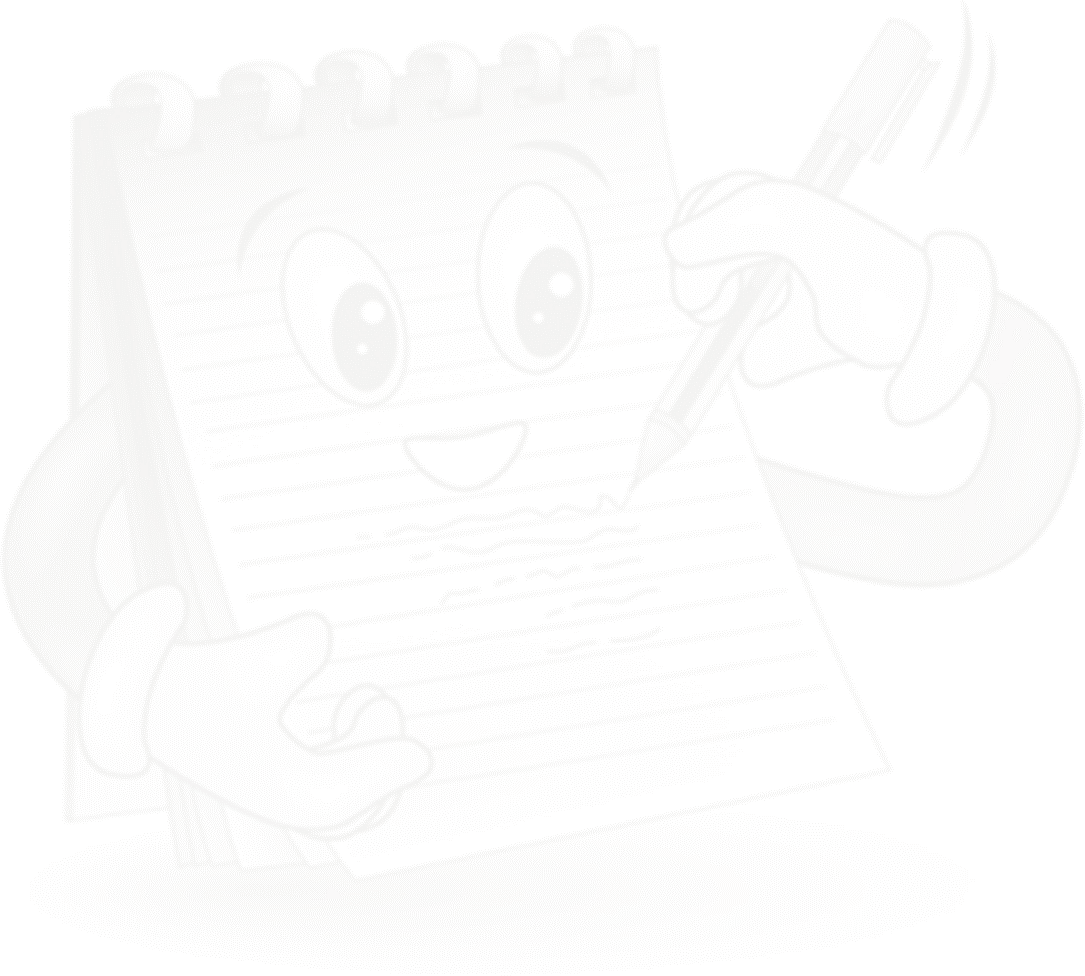
 **External Resources**

* + Kruse, R. (2006). *Data Structures and program designing using ‘C’* (2nd ed.). Pearson Education.
  + Srivastava, S. K., & Srivastava, D. (2004). *Data Structures Through C in Depth* (2nd ed.). BPB Publications.
  + Weiss, M. A. (2001). *Data Structures and Algorithm Analysis in C* (2nd ed.). Pearson Education.

**Video Links**

|  |  |
| --- | --- |
| **Topic** | **Link** |
| Introduction and definition of stacks | https://[www.youtube.com/watch?v=FNZ5o9S9prU](http://www.youtube.com/watch?v=FNZ5o9S9prU) |
| Recursion | https://[www.youtube.com/watch?v=k0bb7UYy0pY](http://www.youtube.com/watch?v=k0bb7UYy0pY) |
| Evaluation of postfix expression using stack | https://[www.youtube.com/watch?v=\_EP4gpG-4kQ](http://www.youtube.com/watch?v=_EP4gpG-4kQ) |

**Notes:**



**Chapter Table of Contents**

**Chapter 3.2**

**Queues**

[Aim 173](#_bookmark102)

[Instructional Objectives 173](#_bookmark103)

[Learning Outcomes 173](#_bookmark104)

* + 1. [Introduction to Queue 174](#_bookmark105)
       1. [Definition of a Queue 174](#_bookmark106)
       2. [Array Representation of Queue 175](#_bookmark107)

[Self-assessment Questions 180](#_bookmark108)

* + 1. [Types of Queue 181](#_bookmark109)
       1. [Simple Queue 181](#_bookmark110)
       2. [Circular Queue 182](#_bookmark111)
       3. [Double Ended Queue 188](#_bookmark112)
       4. [Priority Queue 194](#_bookmark113)

[Self-assessment Questions 196](#_bookmark114)

* + 1. [Operations on Queue 196](#_bookmark115)
       1. [Insertion 197](#_bookmark116)
       2. [Deletion in Queue 198](#_bookmark117)
       3. [Qempty Operation 199](#_bookmark118)
       4. [Qfull Operation 200](#_bookmark119)
       5. [Display Operation 200](#_bookmark120)

[Self-assessment Questions 201](#_bookmark121)

* + 1. [Application of Queue 202](#_bookmark122)

[Self-assessment Questions 203](#_bookmark123)

[Summary 204](#_bookmark124)

[Terminal Questions 205](#_bookmark125)

[Answer Keys 205](#_bookmark126)

[Activity 206](#_bookmark127)

[Bibliography 207](#_bookmark128)

[e-References 207](#_bookmark129)

[External Resources 207](#_bookmark130)

[Video Links 207](#_bookmark131)

 **Aim**

To educate the students with the basic knowledge of queues, its types and operations on queues

******Instructional Objectives**

After completing this chapter, you should be able to:

* Explain queue and its operations
* Describe the array representation of Queue
* Discuss different types of queue with example
* Illustrate the creation, insertion, deletion and search operation on various types of queue

******Learning Outcomes**

At the end of this chapter, you are expected to:

* Demonstrate queue with its operations
* Implement double ended queue using linked list
* Identify requirement of priority queue
  + 1. **Introduction to Queue**

In simple language a queue is a simple waiting line which keeps growing if we add the elements to its end and keep shrinking on removal of elements from its front. If we compare stack, queue reflects the more commonly used maxim in real-world, that is, “first come, first served”. Long waiting lines in food counters, supermarkets, banks are common examples of queues.

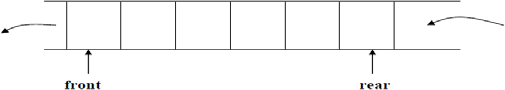
For all computer applications, we define a queue as list in which all additions to the list are made at one end, and all deletions from the list are made at the other end. Applications of queues are, if anything, even more common than are applications of stacks, since in performing tasks by computer, as in all parts of life, it is often necessary to wait one’s turn before having access to something. Within a computer processor there can be queues of tasks waiting for different devices like printer, for access to disk storage, or even, with multitasking, for using the CPU. Within a single program, there may be multiple requests to be kept in a queue, or one task may create other task, which must be done in turn by keeping them in a queue.

A queue is a data structure where elements are added at the back and remove elements from the front. In that way a queue is like “waiting in line”: the first one to be added to the queue will be the first one to be removed from the queue.

Queues are common in many applications. ***For example***, while we read a book from a file, it is quite natural to store the read words in a queue so that once reading is complete the words are in the order as they appear in the book. Another common example is buffer for network communication that temporarily store packets of data arriving on a network port. Generally speaking, it is processed in the order in which the elements arrive.

1. **Definition of a Queue**

In a more formal way, queue can be defined as a list or a data structure in which data items can be added at the end (generally referred as rear) and they can be deleted from font of the queue. The data element to be deleted is the one which would spend maximum time in the queue. It is because of this property, queue is also referred to as a first-in-first-out (FIFO) data structure. The figure 3.2.1 below shows pictorial representation of a queue.



*Figure 3.2.1: Representation of a Queue in Computer’s Memory*

Generally a queue can also be referred to as a container of objects (in other words linear collection) that are deleted or added based on principle of First-In-First-Out (FIFO). A very good example of a queue can be a line of students in the ice-cream counter of the college canteen. New arrival of students can be added to a line at back of the queue, while removal serving (or removal) happens in the front of the queue. The queue allows only two operations; enqueue and dequeues. Enqueue is an operation that allows insertion operation, dequeue allows us to remove an item.

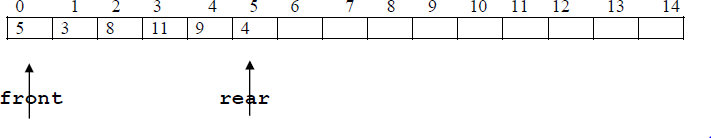
Stack and a queue difference lies only in deletion of item. A stack removes most recently added item; while a queue removes the least recently added item first.

In spite of its simplicity, the queue is a very important concept with many applications in simulation of real life events such as lines of customers at a cash register or cars waiting at an intersection, and in programming (such as printer jobs waiting to be processed. Many Smalltalk applications use a queue but instead of implementing it as a new class, they use an Ordered Collection because it performs all the required functions.

Dequeuing or removing an item from a queue is only possible on non-empty queues, which requires contract in the interface. This interface can be written without committing to an implementation of queues. This is important so that different implementations of the functions in this interface can choose different representations.

1. **Array Representation of Queue**

The array to implement the queue would need two variables (indices) called front and rear to point to the first and the last elements of the queue. The figure 3.2.2 shows array implementation of queue.



*Figure 3.2.2: Array Implementation of Queue*

**Initially:**

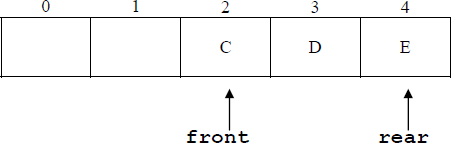
*q->rear = -1;*

*q->front = -1;*

For every ***enqueue*** operation we increment rear by one, and for every ***dequeue*** operation, we increment front by one. Even though ***enqueue*** and ***dequeue*** operations are simple to implement, there is a disadvantage in this set up. The size of the array required is huge, as the number of slots would go on increasing as long as there are items to be added to the list (irrespective of how many items are deleted, as these two are independent operations.)

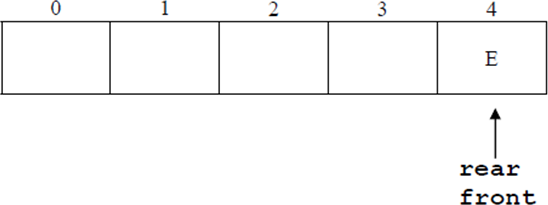
**Problems with this representation**

Although there is space in the following queue in the initial blocks, we may not be able to add a new item. An attempt will cause an overflow.



*Figure 3.2.3: Queue Overflow Situation*

It is possible to have an empty queue yet no new item can be inserted. (When front moves to the point of rear, and the last item is deleted.)



*Figure 3.2.4: Overflow Situation in an Empty Queue*

**The below program shows the implementation of a queue using an Array**

**Program:**

/\*

\* [*C Program to Implement a Queue using an Array*](http://www.sanfoundry.com/c-program-queue-using-array/)

\*/

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

#define SIZE 50

int queue\_arr[SIZE]; int rear =-1;

int front =-1; int main()

{

int ch; while(1)

{

printf("1.Insert element to the queue \n"); printf("2.Delete element from the queue \n"); printf("3.Display all elements of the queue \n"); printf("4.Quit \n");

printf("Enter your choice : "); scanf("%d",&ch);

switch(ch)

{

case 1:

break; case 2:

break; case 3:

break; case 4: exit(1); default:

insert();

delete();

display();

printf("Invalid Input \n");

}/\*End of switch\*/

}/\*End of while\*/

}/\*End of main()\*/ insert()

{

int add\_item; if(rear == SIZE -1)

printf("Queue Overflow \n"); else

{

if(front ==-1)

/\*If queue is initially empty \*/ front =0;

printf("Inset the element in the queue : "); scanf("%d",&add\_item);

rear = rear +1; queue\_arr[rear]= add\_item;

}

}/\*End of insert()\*/

delete()

{

if(front ==-1|| front > rear)

{

printf("Queue Underflow \n"); return;

}

else

{

printf("Element deleted from the queue is : %d\n", queue\_arr[front]); front = front +1;

}

}/\*End of delete() \*/ display()

{

int i; if(front ==-1)

printf("Queue is empty \n"); else

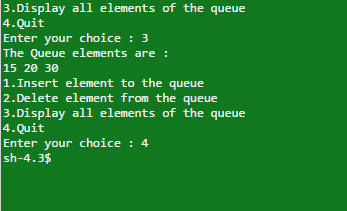
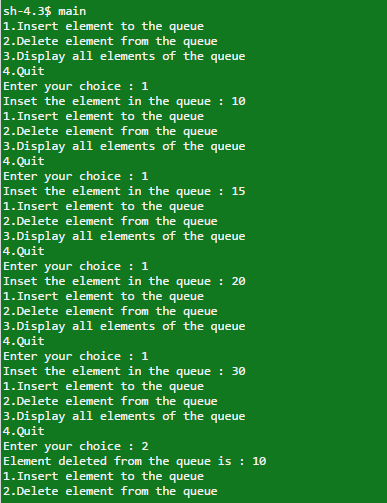
{

printf("The Queue elements are : \n"); for(i = front; i <= rear; i++) printf("%d ", queue\_arr[i]); printf("\n");

}

}/\*End of display() \*/

**Output:**





**Did you know?**

Though the simple queue appears to be very simple FIFO model, it is a very important model used for any applications. During the initial implementation of most of the operating systems like Linux which is generic programmer friendly OS and others like TinyOs which is OS used by wireless sensor networks used simple FIFO queue for scheduling different tasks due to ease of implementation and limitations of resources.

 **Self-assessment Questions**

* 1. Which one of the following is an application of Queue Data Structure?
     1. When
     2. A resource is shared among multiple devices
     3. Printer jobs waiting to be processed.
     4. Buffer used in network communication to store data packets
     5. All of the above
  2. For every ***enqueue*** operation, we by one, and for every ***dequeue*** operation, we by one.
     1. Decrement rear, decrement front
     2. Increment rear, increment front
     3. Increment front, increment rear
     4. Decrement front, decrement rear
  3. For queue implementation, we need two pointers namely front and rear. This pointers are initialized as:
     1. front=1 and rear=-1 b) front=-1 and rear=-1

c) front=-1 and rear=1 d) front=1 and rear=1

* + 1. **Types of Queue**

A queue represents a basket of items. Enqueue is an operation that adds an item to this basket and dequeue is an operation that chooses an item to be removed from the queue (if the queue is not empty). Similar to human queues, these queues will vary based on the rule used to choose the item to be removed from the queue. Giving different names to the basic operations of the queue based on the operations they perform is usual and it helps in avoiding the confusion.

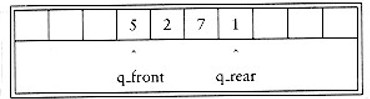
However, using the general signature for different kinds of queue will make our code more modular later, when algorithms based on the different kinds of queue are discussed.

1. **Simple Queue**

Like the stacks, we can also implement queue using lists and arrays. Both the arrays and the linked lists implementations have running time complexity of O (1) for every operation. This section covers the array implementation of queues.

For every queue data structure, an array is kept namely, QUEUE [], and there are two positions q\_front and q\_rear, which represent the beginning and the ends of the queue respectively. q\_size takes care of the size of the queue.

All the above information makes part of a structure, and except for the queue functions themselves, no functions should access these directly. The figure 3.2.5 shows a queue in some intermediate state. The blank cells have undefined values in them. In particular, the elements in the first two cells have spent maximum time in the queue.



*Figure 3.2.5: Basic Queue example*

In order to enqueue an element x, we must first increment q\_rear and q\_size, then set QUEUE [q\_rear] = x. To dequeue an element, assign return value to QUEUE [q\_front], decrement q\_size, and then increment q\_front. Other strategies are possible (this is discussed later).

**Using array Simple queue can be declared as**

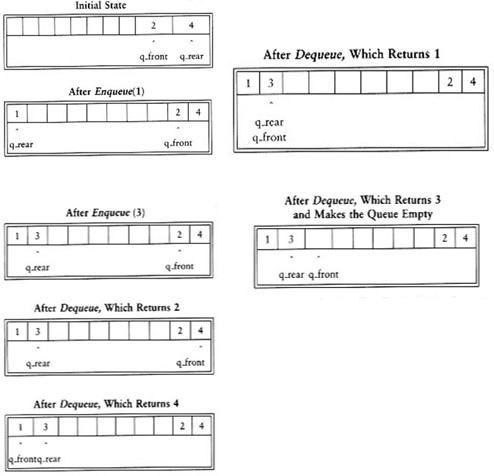
*#define MAX 10*

*int queue[MAX], rear=0, front=0;*

1. **Circular Queue**

One biggest problem with the simple queue is its implementation. After adding 10 elements in the queue (considering previously discussed situation), the queue looks like full, since q\_front is 10, and the next enqueue would be in a non-existent position. However, there might some positions available in the queue, as many elements may have been already dequeued. Queues, like stacks, frequently stay small even in the presence of a lot of operations.

The solution for this problem is that whenever q\_rear or q\_front comes to the end of the array, it is wrapped around to the beginning. The following figure shows the queue during some operations



*Figure 3.2.6: Circular Queue*

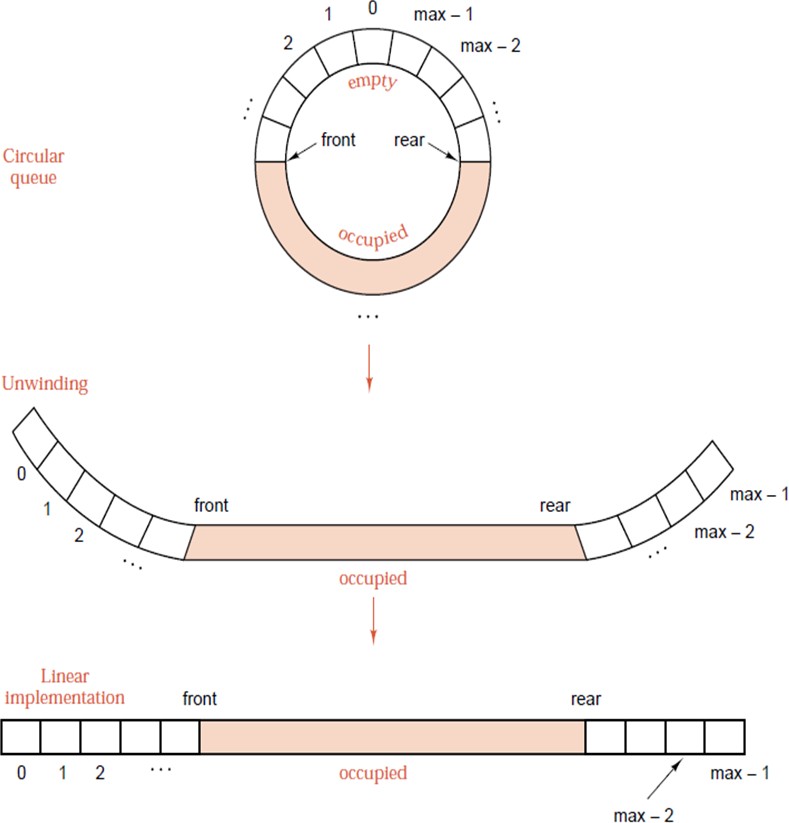
There is a minimal need to write an extra code to implement the wraparound although it increases the running time complexity). If incrementing either q\_front or q\_rear makes it to go past the array, the value is reset to the first position in the array.

However, two things are needed to be taken care of while using circular array implementation of queues. First and the foremost thing, we must check if queue is not empty, because a dequeue operation when the queue is empty returns an undefined value.

Secondly, sometimes programmers represent front and rear differently for queues. ***For example***, some programmers do not use an entry to keep track the size of the queue, because they rely on the assumption that the queue is empty, q\_front = q\_rear - 1. The size is computed implicitly by comparing q\_front and q\_rear. This is a very tricky way, since there are some special cases, therefore one need to be very careful if we need to modify code written this way. Consider the situation that size is not part of the structure, then if the size of the array is A\_SIZE, the queue is full when there are A\_SIZE -1 elements, since only A\_SIZE different sizes can be differentiated, and one of these is 0.

In applications where it is sure that the number of enqueue is not larger than the size of the queue, obviously the wraparound is not necessary. As with stacks, dequeues are rarely performed unless the calling routines are certain that the queue is not empty. Thus error calls are frequently skipped for this operation, except in critical code. This is generally not justifiable, because the time savings that you are likely to achieve are too minimal.

We can think of an array as a circle rather than a straight line in order to overcome the inefficient use of space as depicted in the figure 3.2.7. In this way, as entries are added and removed from the queue, the head will continually chase the tail around the array, so that the snake can keep crawling indefinitely but stay in a confined circuit. At different times, the queue will occupy different parts of the array, but there no need to worry about running out of space unless the array is fully occupied, in which case there is truly overflow



*Figure 3.2.7: Queue in a Circular Array*

**Implementation of Circular Arrays**

In order to implement the circular queue as a linear array, consider the positions around the circular arrangement as numbered from zero to max-1, where max is the total number of elements in the circular arrays. We use same numbered entries of a linear array to implement a circular array. Now it becomes a very simple logic of using modular arithmetic i.e. whenever the index crosses max-1, we start again from 0. This is as simple as doing arithmetic on circular clock face where the hours are numbered from 1 to 12, and if four hours are added to ten o’clock, two o’clock is obtained.

**Program:**

//Program for Circular Queue implementation through Array

#include <stdio.h> #include<ctype.h> #include<stdlib.h> #define SIZE 5

int circleq[SIZE]; int front,rear;

int main()

{

void insert(int, int); void delete(int);

int ch=1,i,n; front = -1;

rear = -1; while(1)

{

printf("\nMAIN MENU\n1.INSERTION\n2.DELETION\n3.EXIT"); printf("\nENTER YOUR CHOICE : ");

scanf("%d",&ch); switch(ch)

{

case 1:

printf("\nEnter the elements of the queue: "); scanf("%d",&n);

insert(n,SIZE); break;

case 2:

delete(SIZE); break;

case 3:

exit(0);

default: printf("\nInvalid input. ");

}

} //end of outer while

} //end of main

void insert(int item,int MAX)

{

//rear++;

//rear= (rear%MAX); if(front ==(rear+1)%MAX)

{

printf("\nCircular queue overflow\n");

}

else

{

if(front==-1) front=rear=0;

else

rear=(rear+1)%MAX; circleq[rear]=item;

printf("\nRear = %d Front = %d ",rear,front);

}

}

void delete(int MAX)

{

int del; if(front == -1)

{

printf("\nCircular queue underflow\n");

}

else

{

del=circleq[front]; if(front==rear)

front=rear=-1; else

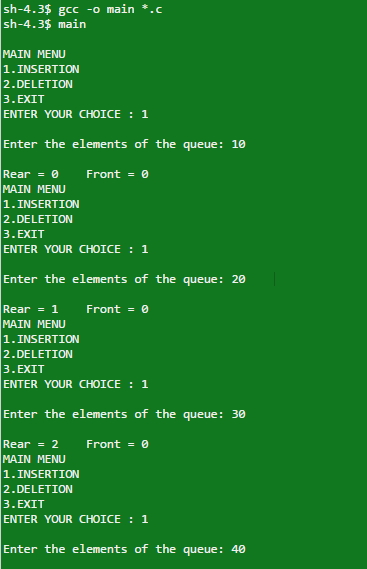
front = (front+1)%MAX;

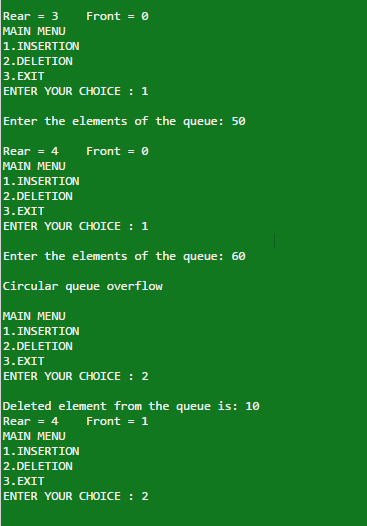
printf("\nDeleted element from the queue is: %d ",del); printf("\nRear = %d Front = %d ",rear,front);

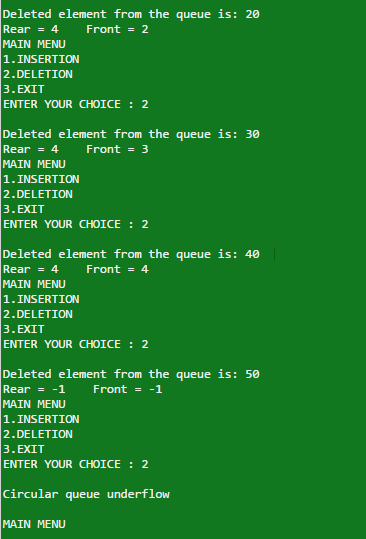
}

}

**Output:**







1. **Double Ended Queue**

A double-ended queue is also known as dequeue. This is an ordered collection of items similar to the queue. A dequeue has two ends, a front and a rear, and all the items remains positioned in the entire collection. The special thing about dequeue is the unrestrictive nature of removing and adding items. An item can be added at either the rear or the front. Similarly, the existing items can be removed from either end. This makes dequeue a hybrid linear structure that provides all the capabilities of queues and stacks in a single data structure. *Figure 3.2.8* shows a dequeue.

It is important to note that even though the dequeue can assume many of the characteristics of stacks and queues, it does not require the LIFO and FIFO orderings that are enforced by those data structures. The use of the addition and removal operations must be done consistently.



*Figure 3.2.8: Dequeue*

A double-ended queue (dequeue, often abbreviated to dequeue, pronounced deck) is an abstract data structure that implements a queue for which elements can only be added to or removed from the front (head) or back (tail). It is also often called a head-tail linked list.

Dequeue is a special type of data structure in which deletion and insertion can be done either at the rear end or at the front end of the queue. The operations that can be performed on dequeues are:

* Insertion of an item from front end
* Insertion of an item from rear end
* Deletion of an item from front end
* Deletion of an item from rear end
* Displaying the contents of queue

**Application of dequeue**

* A nice application of the dequeue is storing a web browser's history. Recently visited URLs are added to the front of the dequeue, and the URL at the back of the dequeue is removed after some specified number of insertions at the front.
* Another common application of the dequeue is storing a software application's list of undo operations.
* One example where a dequeue can be used is the A-Steal job scheduling algorithm.[5] This algorithm implements task scheduling for several processors. A separate dequeue with threads to be executed is maintained for each processor. To execute the next

thread, the processor gets the first element from the dequeue (using the "remove first element" dequeue operation). If the current thread forks, it is put back to the front of the dequeue ("insert element at front") and a new thread is executed. When one of the processors finishes execution of its own threads (i.e. it’sdequeue is empty), it can "steal" a thread from another processor: it gets the last element from the dequeue of another processor ("remove last element") and executes it.

* In real scenario we can attached it to a Ticket purchasing line, It performs like a queue but some time It happens that somebody has purchased the ticket and sudden they come back to ask something on front of queue. In this scenario because they have already purchased the ticket so they have privilege to come and ask for any further query. So in this kind of scenario we need a data structure where according to requirement we add data from front. And In same scenario user can also leave the queue from rear.

**Program:**

/\*Implementation of De-queue using arrays\*/ #include<stdio.h>

#include<stdlib.h> #define MAX 10

typedef struct dequeue

{

int front,rear; int arr[MAX];

}dq;

/\*If flag is zero, insertion is done at beginning else if flag is one, insertion is done at end.

\*/

void enqueue(dq \*q,int x,int flag)

{

int i;

if(q->rear==MAX-1)

{

printf("\nQueue overflow!"); exit(1);

}

if(flag==0)

{

for(i=q->rear;i>=q->front;i--) q->arr[i+1]=q->arr[i];

q->arr[q->front]=x; q->rear++;

}

else if(flag==1)

{

q->arr[++q->rear]=x;

}

else

{

printf("\nInvalid flag value"); return;

}

}

void dequeue(dq \*q,int flag)

{

int i;

/\*front is initialized with zero, then rear=-1 indicates underflow\*/

if(q->rear < q->front)

{

printf("\nQueue Underflow"); exit(1);

}

if(flag==0)/\*deletion at beginning\*/

{

for(i=q->front;i<=q->rear;i++) q->arr[i]=q->arr[i+1];

q->arr[q->rear]=0; q->rear--;

}

else if(flag==1)

{ q->arr[q->rear--]=0;

}

else

{ printf("\nInvalid flag value"); return;

}

}

void display(dq \*q)

{

int i;

for(i=q->front;i<=q->rear;i++) printf("%d ",q->arr[i]);

}

void main()

{

dq q; q.front=0; q.rear=-1; int ch,n; while(1)

{

printf("\nMenu-Double Ended Queue"); printf("\n1. Enqueue – Begin");

printf("\n2. Enqueue – End"); printf("\n3. Dequeue – Begin"); printf("\n4. Dequeue – End"); printf("\n5. Display"); printf("\n6. Exit"); printf("\nEnter your choice: "); scanf("%d",&ch);

switch(ch)

{

case 1:

printf("\nEnter the number: "); scanf("%d",&n);

enqueue(&q,n,0); break;

case 2:

printf("\nEnter the number:" ); scanf("%d",&n);

enqueue(&q,n,1); break;

case 3:

printf("\nDeleting element from beginning"); dequeue(&q,0);

break; case 4:

printf("\nDeleting element from end"); dequeue(&q,1);

break; case 5:

display(&q); break;

case 6:

exit(0); default:

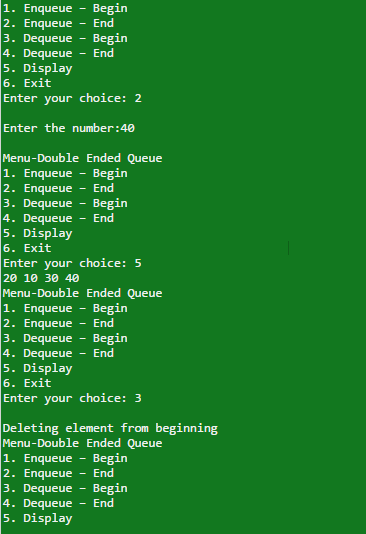
printf("\nInvalid Choice");

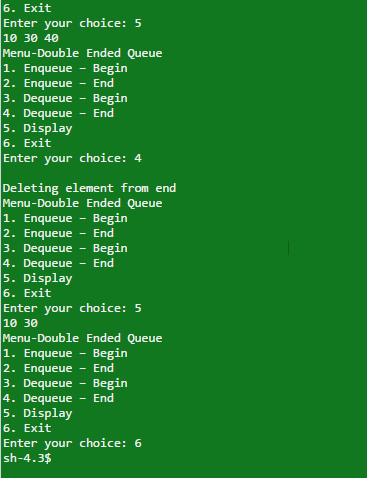
}

}

}

**Output:**





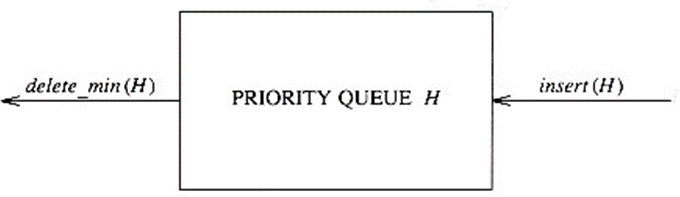
1. **Priority Queue**

Consider a job sent to a line printer. Although these jobs are placed in a queue which is served by the printer on FIFO basis, this may not be a best practice always. Sometimes one job waiting in a queue might be particularly important, so that it might be required to allow the job to be run as soon as printer becomes available. Also, when the printer becomes available, there are several single page jobs in the queue and only one hundred-page job. Here, this might be reasonable to make the long job go last, even if it is not the last job submitted. (Unfortunately, most systems may not follow this, which can be particularly annoying at times.)

In a similar way, in multi-tasking and multi-user environment, an operating system scheduler must decide on which task or user to be allocated a processor. In general a process is allowed to be executed in time slots or time frames. A simple algorithm used for processing jobs is use of queue which process jobs of FIFO bases. Whenever a new job is arrived, it is placed at the end of the queue. The scheduler process the jobs on first come first serve bases from the queue until the queue is empty. This algorithm may not be appropriate since jobs which require short time slot may seem to take a long time because of the wait involved in running. Generally, it is

important that jobs requiring short time slot finishes as fast as possible. Therefore, these jobs must have higher priority over jobs that have already been running. Furthermore, there may be some jobs that are not short which may be still very important and thus must also be considered on priority.

A priority queue is a data structure that allows at least the following two operations: insert, which does the obvious thing, and delete\_min, which finds, returns and removes the minimum element in the queue. The insert operation is the same enqueue, and delete\_min is the priority queue equivalent of the queues dequeue operation. The delete\_minfunction also alters its input.



*Figure 3.2.9: Basic Model of a Priority Queue*

As with most data structures, at times it is possible to add other operations too, but these are extra operations are not part of the basic model depicted in Figure 3.2.8.

Besides operating systems, priority queues have many applications. They are used for external sorting. Priority queues are also important in the implementation of greedy algorithms, which operate by repeatedly finding a minimum.



**Did you know?**

Circular queue is very famous in computer networks because of its very own circular structure. The simplest application of circular queues for network engineers is in implementation of round robin algorithm which is used for token passing and also it is used in FIFO buffering systems.

 **Self-assessment Questions**

* 1. A circular queue is implemented using an array of size 10. The array index starts with 0, front is 6, and rear is 9. The insertion of next element takes place at the array index.
     1. 0 b) 7

c) 9 d) 10

* 1. If the MAX\_SIZE is the size of the array used in the implementation of circular queue, array index start with 0, front point to the first element in the queue, and rear point to the last element in the queue. Which of the following condition specify that circular queue is EMPTY?
     1. Front=rear=0 b) Front= rear=-1

c) Front=rear+1 d) Front= (rear+1)%MAX\_SIZE

* 1. A normal queue, if implemented using an array of size MAX\_SIZE, gets full when
     1. Rear=MAX\_SIZE-1 b) Front= (rear+1)mod MAX\_SIZE

c) Front=rear+1 d) Rear=front

* + 1. **Operations on Queue**

Similar to the operations performed on stacks, all such operations can be performed on queues too. The basic operations involve insertion (enqueue) and deletion (dequeue). The other supporting functions involve Qfill and Qempty. Other queuing operations involve initialising or defining a queue, using it and then completely erasing it from the computer’s memory. This section covers all the functions on queues in detail.

* + - * **enqueue()** – Insert a data element in a queue.
      * **dequeue()** – remove a data element from the queue.

Additional functions are required to make above mentioned queue operation efficient. These are −

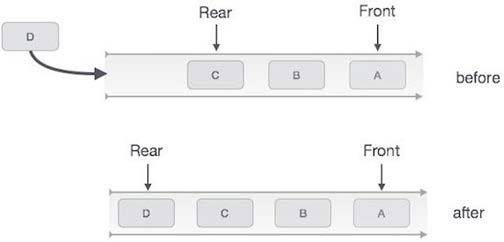
* + - * **Qfull ()** − checks if the queue is full; returns Boolean
      * **Qempty ()** − checks if the queue is empty; returns Boolean

While performing dequeues operations the data is accessed from front pointer and while performing enqueue operation, data is accessed from rear pointer.

1. **Insertion**

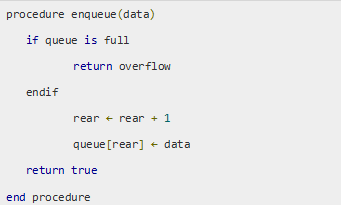
As already discussed in the previous sections, insertion in queue is also called as enqueue operation. The following are the to be followed while performing enqueue operation −

* + **Step 1** − Check if queue is full.
  + **Step 2** − If queue is full, display an overflow error and exit.
  + **Step 3** − If queue is not full, increment rear pointer to point next empty array space.
  + **Step 4** − Add data element to the queue location, where rear is pointing.
  + **Step 5** − return success.

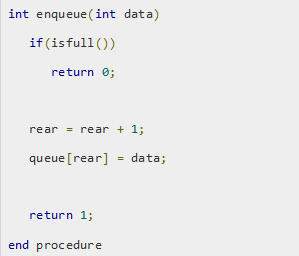


*Figure 3.2.10: Enqueue Operation*

**Following is an algorithm for enqueue operation**



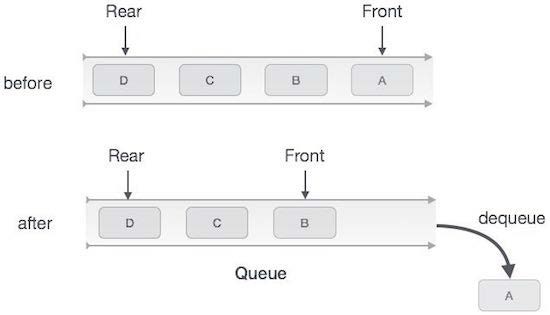
**The above algorithm implemented in C programming is shown below:**



1. **Deletion in Queue**

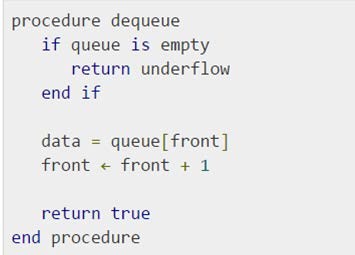
Deletion operation from the queue is also called as dequeue operation. The following are the steps to be followed while performing dequeue operation −

* + **Step 1 −** Check if queue is empty.
  + **Step 2 −** If queue is empty, display an underflow error and exit.
  + **Step 3 −** If queue is not empty, access data where front is pointing.
  + **Step 4 −** Increment front pointer to point next available data element.
  + **Step 5 −** return success.

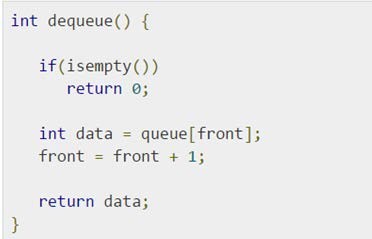


*Figure 3.2.10: Dequeue Operation*

**Following is an algorithm used for performing dequeue operation**



**The above algorithm implemented in C programming is shown below:**

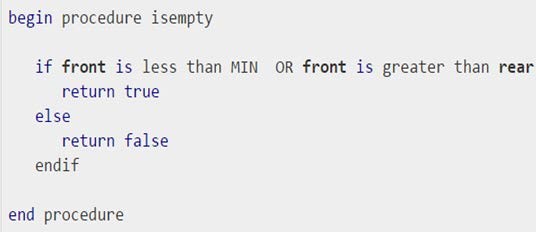


1. **Qempty Operation**

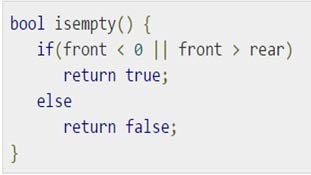
In order to delete an element from the queue first check weather Queue is empty or not. If queue is empty then do not delete an element from the queue. This condition is known as “Underflow”.

If queue is not underflow then we can delete the element from queue. After deleting element from queue we must update the values of rear and front as per the position of elements in the queue.

Algorithm of Qempty() function −



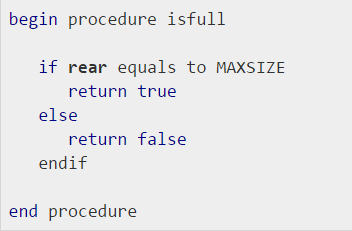
**The above algorithm implemented in C programming is shown below:**



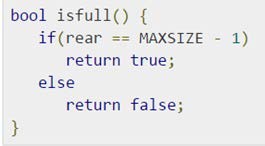
1. **Qfull Operation**

Since we are using single dimensional array for implementation of queue, the best way to know if queue is full or not is to check if rear pointer has reached MAXSIZE-1; which means no space is available in array for additional elements and hence queue is full.

**Algorithm of Qfull () function –**

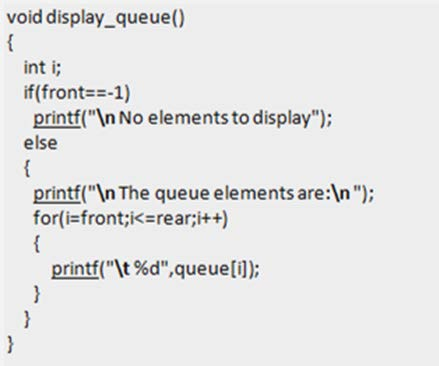


**The above algorithm implemented in C programming is shown below:**



1. **Display Operation**

Queue can be displayed by simply moving rear pointer till it reaches the front pointer. Only condition that should be considered is the Qempty condition and display “Queue Empty” message accordingly.



 **Self-assessment Questions**

* 1. If the elements “A”, “B”, “C” and “D” are placed in a queue and are deleted one at a time, in what order will they be removed?
     1. ABCD b) DCBA

c) DCAB d) BADC

* 1. Deletion operation is done using in a queue.
     1. Front b) Rear

c) Top d) Bottom

* 1. An array of size MAX\_SIZE is used to implement a circular queue. Front, Rear, and count are tracked. Suppose front is 0 and rear is MAX\_SIZE -1. How many elements are present in the queue?
     1. Zero b) One

c) MAX\_SIZE-1 d) MAX\_SIZE

* + 1. **Application of Queue**

As per the very nature of queue, it can be used in all the applications requiring the use of fist come first serve property. Following are the some of the very common applications in computer science where use of queue makes it easy.

1. As already discussed in the previous chapters, queue plays a key role in scheduling for the computer resource sharing based applications. ***For example,*** simplest of the printer queue where printing jobs are added to the scheduling queue and printer serves the requests as per FIFO basis.
2. Similarly queues also play a very important role in CPU scheduling. All the requests for using processors are stored in queue by the CPU scheduler program. The requests are then serviced as per FIFO basis.
3. Another most common application of queue is routing calls in call centres. All the calls made by clients are stored in waiting queue and allotted to different executives who attend the calls. When all the executives are busy handling customers, the call which was made first out of all the other calls is given connected to executive as soon as he becomes available.
4. Another most important application of queue in computer system is interrupt handling. Since computer system is connected to many input and output devices. These devices keep sending requests to processor, by creating interrupts repeatedly. These interrupts are handled by interrupt handle program which put these interrupt in queue as and when they arrive. Then it service there interrupt as per the availability of CPU.
5. *M/M/1 queue*. The M/M/1 queue is a fundamental queueing model in operations research and probability theory. Tasks arrive according to a Poisson process at a certain rate λ. This means that λ customers arrive per hour. More specifically, the arrivals follow an exponential distribution with mean 1 / λ: the probability of k arrivals between time 0 and t is (λ t)^k e^(-λ t) / k!. Tasks are serviced in FIFO order according to a Poisson process with rate μ. The two M's standard for Markov: it means that the system is memoryless: the time between arrivals is independent, and the time between departures is independent.

 **Self-assessment Questions**

* 1. Which data structure allows deleting data elements from front and inserting at rear?
     1. Stacks b) Queues

c) Dequeues d) Binary search tree

* 1. The push and enqueue operations are essentially the same operations, push is used for Stacks and enqueue is used for Queues.
     1. True b) False
  2. In order to input a list of values and output them in order, you could use a Queue. In order to input a list of values and output them in opposite order, you could use a Stack.
     1. True b) False

 **Summary**

* Similar to stacks, Queues are data structure usually used to simplify certain programming operations.
* In these data structures, only one data item can be immediately accessed.
* A queue, in general, allows access to the first item that was inserted.
* The important queue operations are inserting an item at the rear of the queue and removing the item from the front of the queue.
* A queue can be implemented as a circular queue, which is based on an array in which the indices wrap around from the end of the array to the beginning.
* A priority queue allows access to the smallest (or sometimes the largest) item in the queue.
* The important priority queue operations are inserting an item in sorted order and removing the item with the smallest key.
* Few important operations performed on the queue are insertion which is also called enqueue, deletion also called dequeues, Qempty which check if queue is empty, Qfull which check if queue is full and display which is used to display all the elements in the queue.
* Queues finds its applications in implementing job scheduling algorithms, page replacement algorithms, interrupt handling mechanisms etc. in the design of operating systems.

 **Terminal Questions**

1. Explain the basic operations of queue.
2. Discuss the functioning of circular Queue?
3. Mention the limitation of linear queue with a suitable example
4. Discuss applications of Queue

******Answer Keys**

|  |  |
| --- | --- |
| **Self-assessment Questions** | |
| **Question No.** | **Answer** |
| 1 | d |
| 2 | b |
| 3 | b |
| 4 | a |
| 5 | b |
| 6 | a |
| 7 | a |
| 8 | a |
| 9 | d |
| 10 | b |
| 11 | a |
| 12 | a |