Sri SAIRAM COLLEGE OF ENGINEERING

Anekal, Bengaluru

Program/ Course	:	B.E – Computer Science & Engineering
Course Name	:	Data Structures & Applications
Course Code	:	18CS32
Year / Semester	:	2nd YEAR /III
Prepared by (Faculty Name)	:	Prof. Sebin Joy
Department (Faculty Belonging to)	:	CSE
Reviewed by (Signature with Seal & Date)	:	
Approved by (Signature with Seal & Date)	:	

Module - I

1.1 Introduction to Data Structures

Data Structure is a way of collecting and organizing data in such a way that we can perform operations on these data in an effective way. Data Structures is about rendering data elements in terms of some relationship, for better organization and storage. For example, we have data player's name "Virat" and age 26. Here "Virat" is of **String** data type whereas 26 is of **integer** data type.

We can organize this data as a record like **Player** record. Now we can collect and store player's records in a file or database as a data structure. For example: "Dhoni" 30, "Gambhir" 31, "Sehwag" 33

In simple language, Data Structures are structures programmed to store ordered data, so that various operations can be performed on it easily.

1.2 Basic types of Data Structures

As we discussed above, anything that can store data can be called as a data structure, hence Integer, Float, Boolean, Char etc, all are data structures. They are known as **Primitive Data Structures**. Then we also have some complex Data Structures, which are used to store large and connected data. They are known as **Non – Primitive Data Structures**. Some examples are:

- Linked List
- Tree
- Graph
- Stack, Queue etc.

Non – Primitive data structures are further sub divided into linear and linear. All these data structures allow us to perform different operations on data. We select these data structures based on which type of operation is required. Let us look into these data structures in more details at a later part.

Linear: A data structure is said to be linear if its elements form a sequence. The elements of linear data structure represent by means of sequential memory locations. The other way is to have the linear relationship between the elements represented by means of pointers or links.

Example: Array and Linked List. Data Structures and Applications (18CS32) **Non-linear:** A data structure is said to be non-linear if its elements show a hierarchical relationship between each other. All elements assign the memory as random form and you can fetch data elements through random access process.

Example: Trees and Graphs.

The diagram shown below would best explain it.

1.3 Data Structure Operations

Data are processed by means of certain operations which appearing in the data structure. Data has situation on depends largely on the frequency with which specific operations are performed. This section introduces the reader to some of the most frequently used of these operations.

(1) *Traversing:* Accessing each record exactly once so that certain items in the record may be processed. (2) *Searching:* Finding the location of a particular record with a given key value, or finding the location of all records which satisfy one or more conditions. (3) *Inserting:* Adding a new record to the structure. (4) *Deleting:* Removing the record from the structure. (5) *Sorting:* Managing the data or record in some logical order (Ascending or descending order). (6) *Merging:* Combining the record in two different sorted files into a single sorted file. Data Structures and Applications (18CS32)

MLQs:-1. What do you mean by Data Structure? What are the various operations performed on data structures? Explain
MLQs:- 1. What are the various types of data structures? Brief with an example 2. Differentiate between linear and non – linear data structures

1.4 Data Structures in C

Defining a structure

In the C language, structures are used to group together different types of variables under the same name. To declare a structure, you must use the **struct** keyword. The struct keyword defines a new data type, with more than one member. The syntax of the struct statement is as follows − **struct** structure_name{

```
Date member 1;
Data member 2;
...
Data member n;
} [one or more structure tags];
```

The structure tag may be include before concluding the definition or can be included in the main function and each data member is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the structure's definition, a semicolon is mandatory. Here is the way you would declare the Book structure −

```
struct Books {
char title[50];
char author[50];
char subject[100];
int book_id;
} book;
```

Accessing Structure Members

To access any member of a structure, we use the member access operator (.). The member access operator is coded as a period between the structure variable name and the structure member that we wish to access. You would use the keyword struct to define variables of structure type. The following example shows how to use a structure in a program −

```
#include <stdio.h>
#include <string.h>
struct Books {
char title[50];
char author[50];
char subject[100];
int book_id;
};
int main( ) {
struct Books Book1; /* Declare Book1 of type Book */
struct Books Book2; /* Declare Book2 of type Book */
/* book 1 specification */
strcpy( Book1.title, "Data Structures");
strcpy( Book1.author, "Sahni");
strcpy( Book1.subject, "DS");
Book1.book_id = 6495407;
/* book 2 specification */
strcpy( Book2.title, "Telecom Billing");
strcpy( Book2.author, "Shiney");
strcpy( Book2.subject, "Telecom Tutorial");
Book2.book_id = 6495700;
/* print Book1 info */
printf( "Book 1 title : %s\n", Book1.title);
printf( "Book 1 author : %s\n", Book1.author);
printf( "Book 1 subject : %s\n", Book1.subject);
printf( "Book 1 book_id : %d\n", Book1.book_id);
 /* print Book2 info */
printf( "Book 2 title : %s\n", Book2.title);
printf( "Book 2 author : %s\n", Book2.author);
printf( "Book 2 subject : %s\n", Book2.subject);
printf( "Book 2 book_id : %d\n", Book2.book_id);
return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Book 1 title : Data Structures
Book 1 author : Sahni
Book 1 subject : DS
Book 1 book_id : 6495407
Book 2 title : Telecom Billing
Book 2 author : Shiney
Book 2 subject : Telecom Tutorial
Book 2 book_id : 6495700
```

Structures as Function Arguments

You can pass a structure as a function argument in the same way as you pass any other variable or pointer.

```
#include <stdio.h>
#include <string.h>
struct Books {
char title[50];
char author[50];
char subject[100];
int book_id;
};
/* function declaration */
void printBook( struct Books book );
int main( )
{
struct Books Book1; /* Declare Book1 of type Book */
struct Books Book2; /* Declare Book2 of type Book */
/* book 1 specification */
strcpy( Book1.title, "Data Structures");
strcpy( Book1.author, "Sahni");
strcpy( Book1.subject, "DS");
Book1.book_id = 6495407;
/* book 2 specification */
strcpy( Book2.title, "Telecom Billing");
strcpy( Book2.author, "Shiney");
strcpy( Book2.subject, "Telecom Tutorial");
Book2.book_id = 6495700;
/* print Book1 info */
printBook( Book1 );
/* Print Book2 info */
printBook( Book2 );
return 0;
}
void printBook( struct Books book ) {
 printf( "Book title : %s\n", book.title);
printf( "Book author : %s\n", book.author);
printf( "Book subject : %s\n", book.subject);
printf( "Book book_id : %d\n", book.book_id);
}
```

When the above code is compiled and executed, it produces the following result −

```
Book title : Data Structures
Book author : Sahni
Book subject : DS
Book book_id : 6495407
Book title : Telecom Billing
Book author : Shiney
Book subject : Telecom Tutorial
Book book_id : 6495700
```

Arrays of structures

We can also declare an array of structures in order to store records of n number of entities. Below program demonstrates the same concept.

```
#include <stdio.h>
#include <conio.h>
struct Books{
char *title;
char *author;
char *subject;
int book_id;
};
void main()
{
struct Books book[3];
int i;
for(i=0;i<2;i++)
{
scanf("%s%s%s%d",book[i].title,book[i].author,book[i].subject,&book[i].book_id);
}
for(i=0;i<2;i++)
{
printf("%s\t%s\t%s\t%d\n",book[i].title,book[i].author,book[i].subject,book[i].book_id);
}
getch();
}
```

Pointers to Structures

You can define pointers to structures in the same way as you define pointer to any other variable struct Books *struct_pointer;

Now, you can store the address of a structure variable in the above defined pointer variable. To find the address of a structure variable, place the '&'; operator before the structure's name as follows −

struct_pointer = &Book1;

To access the members of a structure using a pointer to that structure, you must use the → operator as follows −

struct_pointer->title;

Let us re-write the above example using structure pointer.

```
#include <stdio.h>
#include <string.h>
struct Books {
char title[50];
char author[50];
char subject[100]; int book_id;
};
/* function declaration */
void printBook( struct Books *book );
int main( )
{
```

```
struct Books Book1; /* Declare Book1 of type Book */
struct Books Book2; /* Declare Book2 of type Book */
/* book 1 specification */
strcpy( Book1.title, "Data Structures");
strcpy( Book1.author, "Sahni");
strcpy( Book1.subject, "DS");
Book1.book_id = 6495407;
/* book 2 specification */
strcpy( Book2.title, "Telecom Billing");
strcpy( Book2.author, "Shiney");
strcpy( Book2.subject, "Telecom Tutorial");
Book2.book_id = 6495700;
/* print Book1 info by passing address of Book1 */
printBook( &Book1 );
/* print Book2 info by passing address of Book2 */
printBook( &Book2 );
return 0;
}
void printBook( struct Books *book ) { printf( "Book title : %s\n", book->title);
printf( "Book author : %s\n", book->author);
printf( "Book subject : %s\n", book->subject);
printf( "Book book_id : %d\n", book->book_id);
}
```

When the above code is compiled and executed, it produces the following result −

```
Book title : Data Structures
Book author : Sahni
Book subject : DS
Book book_id : 6495407
Book title : Telecom Billing
Book author : Shiney
Book subject : Telecom Tutorial
Book book_id : 6495700
```

1.5 Type definitions and structures

Type definitions make it possible to create your own variable types. In the following example we will create a type definition called “intpointer” (a pointer to an integer):

```
#include<stdio.h>
typedef int *int_ptr;
int main()
{
int_ptr myvar; // create a variable of the type int_ptr
return 0;
}
```

It is also possible to use type definitions with structures. The name of the type definition of a structure is usually in uppercase letters. Take a look at the example:

```
#include<stdio.h>
struct telephone
{
char *name;
int number;
};
Typedef struct telephone TELEPHONE
```

```
int main()
{
TELEPHONE index; // create a variable of the type int_ptr
index.name = "Jane Doe";
index.number = 12345;
printf("Name: %s\n", index.name);
printf("Telephone number: %d\n", index.number);
return 0;
}
```

Note: The word struct is not needed before TELEPHONE index;

1.6 Unions

A union is like a structure in which all members are stored at the same address. Members of a union can only be accessed one at a time. The union data type was invented to prevent memory fragmentation. The union data type prevents fragmentation by creating a standard size for certain data. Just like with structures, the members of unions can be accessed with the dot (.) and arrow (->) operators. Take a look at this example:

```
#include<stdio.h>
typedef union myunion
{
double PI;
int B;
}MYUNION;
int main()
{
MYUNION numbers;
numbers.PI = 3.14;
numbers.B = 50;
return 0;
}
```

If you would want to print the values of PI and B, only B value will be printed properly since single memory is reserved to hold the data members' value of union. Hence, 50 will be displayed correctly. Differences between structures and unions

Structure	**Union**
Structure declaration starts with keyword struct	Union declaration starts with keyword union
Structure reserves memory for each data member separately	Union reserves memory i.e., equal to maximum data member size amongst all.
Any data member value can be accessed at any time	Only one data member can be accessed at a time

Ex: struct Book{
int isbn;
float price;
char title[20];
}book;
	Ex: union Book{
	int isbn;
	float price;
	char title[20];
	}book;

Total memory reserved will be sizeof(int)+sizeof(flaot)+(20*sizeof(char)	Total memory reserved will be Max(sizeof(int)+sizeof(flaot)+(20*sizeof(char))

1.7 Self Referential structure

A self referential structure is used to create data structures like linked lists, stacks, etc. A self-referential structure is one of the data structures which refer to the pointer to (points) to another structure of the same type. For example, a linked list is supposed to be a self-referential data structure. The next node of a node is being pointed, which is of the same struct type. Following is an example of this kind of structure:

Syntax: struct struct_name { datatype datatypename; struct_name * pointer_name; }; *For example,* typedef struct listnode { int data; struct listnode *next; } linked_list; In the above example, the **listnode** is a self-referential structure – because the ***next** is of the type struct listnode.
Above syntactical representation can be pictorially presented as below:
data
data
NULL
data

1.8 Arrays in C

In C programming, one of the frequently arising problems is to handle similar types of data. For example, if the user wants to store marks of 100 students. This can be done by creating 100 Data variables individually but, this process is rather tedious and impracticable. This type of problem can be handled in C programming using **arrays**.
An array is a sequence of data item of homogeneous value (same type).
Arrays are of two types: One-dimensional arrays, Multidimensional arrays

Declaration of one-dimensional array:

data_type array_name [array_size];
For example: int age[5];
Here, the name of array is age. The size of array is 5, i.e., there are 5 items (elements) of array age. All elements in an array are of the same type (int, in this case).

Array elements

Size of array defines the number of elements in an array. Each element of array can be accessed and used by user according to the need of program.
For example: int age[5];
Note that, the first element is numbered 0 and so on. That means array indexing always starts from 0 to n-1.
Here, the size of array is 5 times the size of int because there are 5 elements.
Suppose, the starting address of the array age[0] is 2120d and the size of int be 4 bytes. Then, the next address (address of a[1]) will be 2124d, address of a[2] will be 2128d and so on.

Initialization of one-dimensional array

Arrays can be initialized at declaration time as:
int age[5]={2,4,34,3,4};
It is not necessary to define the size of arrays during initialization. Below instruction justifies it. int age[]={2,4,34,3,4};
In this case, the compiler determines the size of array by calculating the number of elements of an array

Accessing array elements

In C programming, arrays can be accessed and treated like variables in C.
For example:

```
scanf("%d",&age[2]); // statement to insert value in the third position of array age
scanf("%d",&age[i]); // statement to insert value in (i+1)th position of array age
printf("%d",age[0]); // statement to print first element of an array age
printf("%d",age[i]); // statement to print (i+1)th element of an array age
```

Following C program calculates the sum of marks of n students using arrays

```
#include <stdio.h>
int main(){
int marks[10],i,n,sum=0;
printf("Enter number of students: ");
scanf("%d",&n);
for(i=0;i<n;++i){
printf("Enter marks of student%d: ",i+1);
scanf("%d",&marks[i]);
sum+=marks[i];
}
printf("Sum= %d",sum);
return 0;
}
```
Output
```
Enter number of students: 3
Enter marks of student1: 12
Enter marks of student2: 31
Enter marks of student3: 2
Sum=45
```

1.9 Multi Dimensional Arrays

C programming language allows programmer to create arrays of arrays known as multidimensional arrays. For example: float a[2][6];
Here, a is an array of two dimension, which is an example of multidimensional array.
For better understanding of multidimensional arrays, array elements of above example can be thought of as below:

Initialization of Multidimensional Arrays

In C, multidimensional arrays can be initialized in different number of ways.
```
int c[2][3]={{1,3,0}, {-1,5,9}};
OR
int c[][3]={{1,3,0}, {-1,5,9}};
OR
int c[2][3]={1,3,0,-1,5,9};
```

Example of Multidimensional Array in C

*C program to find sum of two matrix of order 2*2 using multidimensional arrays where, elements of matrix are entered by user.*
```
#include <stdio.h>
int main(){
float a[2][2], b[2][2], c[2][2];
int i,j;
printf("Enter the elements of 1st matrix\n");
/* Reading two dimensional Array with the help of two for loop. If there was an array of 'n' dimension,
'n' numbers of loops are needed for inserting data to array.*/ for(i=0;i<2;++i)

for(j=0;j<2;++j)
{
printf("Enter a%d%d: ",i+1,j+1);
scanf("%f",&a[i][j]);
}
printf("Enter the elements of 2nd matrix\n");
for(i=0;i<2;++i)
for(j=0;j<2;++j)
{
printf("Enter b%d%d: ",i+1,j+1);
scanf("%f",&b[i][j]);
}
for(i=0;i<2;++i)
for(j=0;j<2;++j)
{
/* Writing the elements of multidimensional array using loop. */
c[i][j]=a[i][j]+b[i][j]; /* Sum of corresponding elements of two arrays.
*/ }
printf("\nSum Of Matrix:");
for(i=0;i<2;++i)
for(j=0;j<2;++j)
{
printf("%.1f\t",c[i][j]);
if(j==1) /* To display matrix sum in order. */
printf("\n");
}
return 0;
}
```

Output

```
Enter the elements of 1st matrix Enter a11: 2
Enter a12: 0.5
Enter a21: -1.1
Enter a22: 2
Enter the elements of 2nd matrix
Enter b11: 0.2
Enter b12: 0
Enter b21: 0.23
Enter b22: 23
Sum of Matrix:
2.2 0.5
-0.9 25.0
```

1.10 Operations on Arrays in C

Various operations like create, display, insert, delete, search, merge, sort can be performed in order to use arrays in programming. Below shown program performs all such operations on an array in the form of a menu driven approach. #include<stdio.h> #include<stdlib.h>

```
int a[20],b[20],c[40];
int m,n,p,val,i,j,key,pos,temp;
/*Function Prototype*/
void create();
void display();
void insert();
void del();
void search();
void merge();
void sort();
int main()
{
int choice;
do{
printf("\n\n--------Menu-----------\n");
printf("1.Create\n");
printf("2.Display\n");
printf("3.Insert\n");
printf("4.Delete\n");
printf("5.Search\n");
printf("6.Sort\n");
printf("7.Merge\n");
printf("8.Exit\n");
printf("-----------------------");
printf("\nEnter your choice:\t");
scanf("%d",&choice);
switch(choice)
{
case 1: create();
break;
case 2:
display();
break;
case 3:
insert();
break;
case 4:
del();
break;
case 5:
search();
break;
case 6:
sort();
break;
case 7:
merge();
break;
case 8:
exit(0);
break;
default:
printf("\nInvalid choice:\n");
break;
}
}while(choice!=8);
return 0;
}
void create() //creating an array
{
printf("\nEnter the size of the array elements:\t");
scanf("%d",&n);
printf("\nEnter the elements for the array:\n");
for(i=0;i<n;i++)
{
scanf("%d",&a[i]);
}
}//end of create()
void display() //displaying an array elements
{
int i;
printf("\nThe array elements are:\n");
for(i=0;i<n;i++){
printf("%d\t",a[i]);
}
}//end of display()
void insert() //inserting an element in to an array
{
printf("\nEnter the position for the new element:\t");
```

```
scanf("%d",&pos);
printf("\nEnter the element to be inserted :\t");
scanf("%d",&val);
for(i=n-1;i>=pos;i--)
{
a[i+1]=a[i];
}
a[pos]=val;
n=n+1;
}//end of insert()
void del() //deleting an array element
{
printf("\nEnter the position of the element to be deleted:\t");
scanf("%d",&pos);
val=a[pos];
for(i=pos;i<n-1;i++)
{
a[i]=a[i+1];
}
n=n-1;
printf("\nThe deleted element is =%d",val);
}//end of delete()
void search() //searching an array element
{
printf("\nEnter the element to be searched:\t");
scanf("%d",&key);
for(i=0;i<n;i++)
{
if(a[i]==key)
{
printf("\nThe element is present at position %d",i);
break;
}
}
if(i==n)
{
printf("\nThe search is unsuccessful");
}
}//end of serach()
void sort() //sorting the array elements
{
for(i=0;i<n-1;i++)
{
for(j=0;j<n-1-i;j++)
{
if(a[j]>a[j+1])
{
temp=a[j];
a[j]=a[j+1];
a[j+1]=temp;
}
}
}
```

```
printf("\nAfter sorting the array elements are:\n");
display();
}//end of sort
void merge() //merging two arrays
{
printf("\nEnter the size of the second array:\t");
scanf("%d",&m);
printf("\nEnter the elements for the second array:\n");
for(i=0;i<m;i++)
{
scanf("%d",&b[i]);
}
for(i=0,j=0;i<n;i++,j++)
{
c[j]=a[i];
}
for(i=0;i<m;i++,j++)
{
c[j]=b[i];
}
p=n+m;
printf("\nArray elements after merging:\n");
for(i=0;i<p;i++)
{
printf("%d\t",c[i]);
}
}//end of merge
```

1.11 Sparse Matrices

Matrices with a relativelghproportion of zero entries are called sparse rices. Two general types of n-square matrices, which occur in various applications, are pictured in below given figure. The first matrix, where all entries above the main diagonal are zero or, equivalently where nonzero entries can only occur on or below the main diagonal, is called a (lower) triangular matrix. The second matrix where nonzero entries can only occur on the diagonal or on elements immediately above or below the diagonal is called a tridiagonal matrix.

Below shown C program accepts a matrix of order mXn and verifies whether it is a sparse matrix or not.

```
#include <stdio.h>
void main () { static int m1[10][10]; int i,j,m,n; int counter=0;
printf ("Enter the order of the matix\n"); scanf ("%d %d",&m,&n);
printf ("Enter the co-efficients of the matix\n"); for (i=0;i<m;++i)

 { for (j=0;j<n;++j)
 { scanf ("%d",&m1[i][j]);
 if (m1[i][j]==0)
 { ++counter;
}
}
} if (counter>((m*n)/2)) { printf ("The given matrix is sparse matrix \n"); } else printf
("The given matrix is not a sparse matrix \n");
printf ("There are %d number of zeros",counter);
} /* End of main() */ /*---------------------------------------------- Output Enter the order of the
```

matix 2 2 Enter the co-efficients of the matix 1 2 3 4 The given matrix is not a sparse matrix There are 0 number of zeros
Run 2 Enter the order of the matix 3 3 Enter the co-efficients of the matix 1 0 0 0 0 1 0 1 0 The given matrix is sparse matrix There are 6 number of zeros

1.12 Polynomials

Polynomials come under the section Algebra in Mathematics. A polynomial is an expression of finite length constructed from variables, constants and non-negative integer exponents. The operations addition, subtraction and multiplication determine its entire structure. Polynomials are used in a wide variety of problems where they are called as polynomial equations.
An example of a polynomial is $3x_2+2x+7$; here, the total number of terms is 3. The coefficients of each term are 3, 2, 7 and degrees 2, 1, 0 respectively. While adding two polynomials, following cases need to be considered.
1. When the degrees of corresponding terms of the two polynomials are same:

This is the normal case when corresponding coefficients of each term can be added directly. For example, the sum of the polynomials
$5x_3+2x_2+7$
$7x_3+9x_2+12$

$12x_3+11x_2+19$ is a simple addition where all the degrees of the corresponding terms are same.
2. When the degrees of corresponding terms of the polynomials are different: Here, the term with the larger degree pre-dominates.
$9x_4+5x_3+ +2x$
$3x_4+ +4x_2+7x$

$12x_4+5x_3+4x_2+9x$
Below shown C program accepts two polynomials and performs addition operation on them.

```
#include<stdio.h>
main()
{
int a[10], b[10], c[10],m,n,k,i,j,p,count=0;
clrscr();
printf("\n\tPolynomial Addition\n");
printf("\t===================\n");
printf("\n\tEnter the no. of terms of the polynomial:");
scanf("%d", &m);
printf("\n\tEnter the degrees and
coefficients:"); for (i=0;i<2*m;i++)
scanf("%d", &a[i]);
printf("\n\tFirst polynomial is:");
for(i=0;i<2*m;i=i+2)
{
printf("%dX^%d+",a[i],a[i+1]);
}
printf("\b ");
printf("\n\n\n\tEnter the no. of terms of 2nd
polynomial:"); scanf("%d", &n);
printf("\n\tEnter the degrees and co-efficients:");
```

```
for(j=0;j<2*n;j++)
scanf("%d", &b[j]);
printf("\n\tSecond polynomial is:");
for(j=0;j<2*n;j=j+2)
{
printf("%dX^%d+",b[j],b[j+1]);
}
printf("\b ");
i=j=k=1;
while (m>0 && n>0)
{
if(a[i]==b[j])
{
c[k-1]=a[i-1]+b[j-1];
c[k]=a[i];
count+=2;
m--;
n--;
i+=2;
j+=2;
}
else if(a[i]>b[j])
{
c[k-1]=a[i-1];
c[k]=a[i];
count+=2;
m--;
i+=2;
}
else
{
c[k-1]=b[j-1];
c[k]=b[j];
count+=2;
n--;
j+=2;
}
k+=2;
}
while (m>0)
{
c[k-1]=a[i-1];
c[k+1]=a[i];
count+=2;
k+=2;
i+=2;
m--;
}
while (n>0)
{
c[k-1]=b[j-1];
c[k+1]=b[j];
count+=2;
k+=2;
j+=2;
n--;
}
printf("\n\n\n\n\tSum of the two polynomials is:");
for(k=0;k<count;k=k+2)
{
printf("%dX^%d+",c[k],c[k+1]);
}
printf("\b ");
return 0;
}
```

1.13 Dynamically allocated arrays

Usually array at the time of its declaration gets it predefined size. Bounding to this size, programmers will have to find out solution for several problems. For example, if SIZE value is 100, the program can be used to sort a collection up to 100 numbers. If the user wishes to sort more than 100 numbers, we have to change the definition of SIZE using some larger value and recompile the program. It becomes little tedious for someone to rethink on this number. Hence the concept, Dynamic Arrays.
Let us follow the below program to understand the concept of dynamically allocated arrays with the help of Dynamic Memory Allocation functions (described in 1.14)

```
#include <stdio.h>
void main(void)
{
int i,*list,n,p;
printf("Enter array size\n");
scanf("%d",&n);
malloc(n, sizeof(int));
printf("Contents and address of Reserved memory blocks for the array\n");
for(i=0;i<n;i++)
printf("%d\t%u\n",*(list+i),&*(list+i));
printf("Enter array elements\n");
for(i=0;i<n;i++)
scanf("%d",&*(list+i));
printf("Entered array elements\n");
for(i=0;i<n;i++)
printf("%d\n",*(list+i));
printf("Enter the resize value\n");
scanf("%d",&p);
realloc(list,p * sizeof(int));
printf("Contents and address of Reserved memory blocks for the array\n");
for(i=0;i<p;i++)
printf("%d\t%u\n",*(list+i),&*(list+i));
printf("Enter the elements\n");
for(i=n;i<p;i++)
scanf("%d",&*(list+i));
printf("Updated array:\n");
for(i=0;i<p;i++)
printf("%d\t%u\n",*(list+i),&*(list+i));
list=NULL;
free(list);
printf("Array Contents after deallocation...\n");
for(i=0;i<p;i++)
printf("%d\n",*(list+i));
}
```

malloc(n, sizeof(int)); allocates n blocks of memory locations to hold integer data.
realloc(list,p * sizeof(int)); resizes n to p where p>n and allocates p number of memory locations without disturbing the contents of older locations. The above program displays the status of array before and after memory allocation to it.

2D – Dynamically declared Arrays

```
#include <stdio.h>
#include <stdlib.h>
int main() {
int **p;
int m, n, i, j;
printf("Enter number of rows and columns: ");
scanf("%d%d", &m, &n);
/* Allocate memory */
p = (int *) malloc(sizeof(int *) * m); /* Row pointers */
for(i = 0; i < m; i++) {
p[i] = (int *) malloc(sizeof(int) * n); /* Row data */
}
/* Assign values to array elements and print them */
for(i = 0; i < m; i++)
{
for(j = 0; j < n; j++)
{
scanf("%d",&*(*p+i)+j);
// p[i][j] = (i * 10) + (j + 1);
}
}
for(i = 0; i < m; i++)
{
for(j = 0; j < n; j++)
{
printf("%d",*(*p+i)+j);
}
printf("\n");
}
/* Deallocate memory
*/ for(i = 0; i < m; i++)
{
free(p[i]); /* Rows */
}
free(p); /* Row pointers */
printf("Memory contents of the array after deallocation...\n");
for(i = 0; i < m; i++)
{
for(j = 0; j < n; j++)
{
printf("%d",*(*p+i)+j);
}
printf("\n");
}
return 0;
}
```

Applications of Arrays

1. Arrays are used in sorting and searching methods
2. Arrays are used to represent graphs in the form of matrix
3. Arrays can be used in CPU Scheduling
4. Arrays can be used to demonstrate Stack and Queue data structures

1.12 Strings

Strings are one-dimensional array of characters terminated by a **null** character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a **null**. The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello." char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
If you follow the rule of array initialization then you can write the above statement as follows −
char greeting[] = "Hello";
Following is the memory presentation of the above defined string in C
Note: Actually, you do not place the null character at the end of a string constant. The C compiler automatically places the '\0' at the end of the string when it initializes the array.
Let us try to print the above mentioned string :−

```
#include <stdio.h>
int main ()
{

char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
printf("Greeting message: %s\n", greeting );
return 0;
}
```

When the above code is compiled and executed, it produces the following result
:− Greeting message: Hello C supports a wide range of built in functions that manipulate null-terminated strings :− strcpy(s1, s2):- Copies string s2 into string s1.
strcat(s1, s2):- Concatenates string s2 onto the end of string s1.
strlen(s1):- Returns the length of string s1.
strcmp(s1, s2):- Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
strchr(s1, ch); Returns a pointer to the first occurrence of character ch in string s1
strstr(s1, s2); Returns a pointer to the first occurrence of string s2 in string s1.
The following C program implements some of the above mentioned functions.

```
#include <stdio.h>
#include <string.h>
int main () {
char str1[12] = "Hello";
char str2[12] = "World";
char str3[12];
int len ;
/* copy str1 into str3 */
strcpy(str3, str1);
printf("strcpy( str3, str1) : %s\n", str3 );
/* concatenates str1 and str2 */
strcat( str1, str2);
printf("strcat( str1, str2): %s\n", str1 );
/* total lenghth of str1 after concatenation */
len = strlen(str1);
printf("strlen(str1) : %d\n", len );
return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
strcpy( str3, str1) : Hello
strcat( str1, str2): HelloWorld
strlen(str1) : 10
```

1.13 Dynamic Memory Management Functions

The exact size of an array is unknown until the compile time, i.e., time when a compiler compiles code written in a programming language into an executable form. The size of array you have declared initially can be sometimes insufficient and sometimes more than required. Dynamic memory allocation allows a programmer to obtain more memory space, while running or to release space when no space is required.

Although, C language inherently does not have any technique to allocate memory dynamically, there are 4 library functions under "stdlib.h" for dynamic memory allocation . malloc()

calloc()

free()
realloc()

malloc(): The name malloc stands for "memory allocation". The function malloc()reserves a block of memory of specified size and return a pointer of type void which can be casted into pointer of any form.

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 memor 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 respective and the pointer points to the address of first byte of memory.

calloc(): The name calloc stands for "contiguous allocation". The only difference between alloc() 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.

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.
realloc(): The C library function void *realloc(void *ptr, size_t size) attempts to resize the memory block pointed to by ptr that was previously allocated with a call to malloc or calloc.

Declaration

Following is the declaration for realloc() function.
void *realloc(void *ptr, size_t size)

Parameters

_ ptr -- This is the pointer to a memory block previously allocated with malloc, calloc or realloc to be reallocated. If this is NULL, a new block is allocated and a pointer to it is returned by the function.

size -- This is the new size for the memory block, in bytes. If it is 0 and ptr points to an existing block of memory, the memory block pointed by ptr is deallocated and a NULL pointer is returned.
This function returns a pointer to the newly allocated memory, or NULL if the request fails.

Example

The following example shows the usage of realloc() function.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
char *str;
/* Initial memory allocation */
str = (char *) malloc(12);
strcpy(str, "vivekananda");
printf("String = %s, Address = %u\n", str, str);
/* Reallocating memory */
str = (char *) realloc(str, 25);
strcat(str, "swamy");
printf("String = %s, Address = %u\n", str, str);
free(str);
return(0);
}
```

Output

```
String = vivekananda, Address = 355090448
String = vivekanandaswamy, Address = 355090448
```

free(): Dynamically allocated memory with either calloc() or malloc() does not get return on its own. The programmer must use free() explicitly to release space. syntax of free( )

e.g.: free(ptr); This statement causes the space in memory pointer by ptr to be de-allocated.

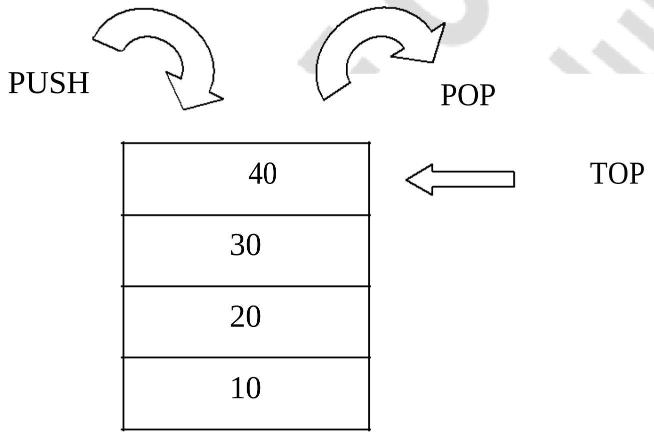
Module – II

Stack and Queues

2.1 Stack

A Stack is data structure in which addition of new element or deletion of existing element always takes place at a same end. This end is known as the top of the stack. That means that it is possible to remove elements from a stack in reverse order from the insertion of elements into the stack.

One other way of describing the stack is as a last in, first out (LIFO) abstract data type and linear data structure. Pictorial representation of the same is shown below



2.2 Operations on Stack

The stack is basically associated with two operations PUSH and POP. Push and pop are the operations that are provided for insertion of an element into the stack and the removal of an element from the stack, respectively. To perform these operations, a reference is used i.e., stack – top. In the above figure, elementary operations like push and pop are performed. Also, the current position of stack – top after performing the operation is shown.

Stack Representation

Stack can be represented using two ways: static representation that is using array concept and dynamic representation that is using linked list with the help of dynamic memory allocation concept (explained in the next module).

Array representation of stack:

int stack[SIZE]; stack [] is an array of integer type with a predefined size SIZE .

Implementation of basic functions goes as below:

Push Function in C

```
void push()
{
int n;
printf("\n Enter item in stack");
scanf("%d",&n);
if(top==size-1)
{
printf("\nStack is Full");
}
else
{
top=top+1;
stack[top]=n;
}
}
```

Pop Function in C

```
void pop()
{
int item;
if(top==-1)
{
printf("\n Stack is empty");
}
else
{
item=stack[top];
printf("\n item popped is=%d", item);
top--;
}
}
```

2.3 Representing stack using dynamic arrays

```
#include <stdio.h>
#include <stdlib.h>
 struct node
{
   int info;
   struct node *ptr;
}*top,*top1,*temp;
void push(int data);
void pop();
void display();
void destroy();
void main()
{
   int no, ch, e;
   printf("\n 1 - Push");
   printf("\n 2 - Pop");
   printf("\n 3 - Exit");
   printf("\n 4 - Dipslay");
   printf("\n 5 - Destroy stack");
   top=NULL;
   while (1)
   {
         printf("\n Enter choice : ");
         scanf("%d", &ch);
         switch (ch)
         {
         case 1:
            printf("Enter data : ");
            scanf("%d", &no);
            push(no);
            break;
         case 2:
            pop();
            break;
```

```
          case 3:
             exit(0);
          case 4:
             display();
             break;
          case 5:
             destroy();
             break;
          default :
             printf(" Wrong choice, Please enter correct choice  ");
             break;
          }
   }
}
void push(int data)
{
   if (top == NULL)
   {
          top =(struct node *)malloc(1*sizeof(struct node));
          top->ptr = NULL;
          top->info = data;
   }
   else
   {
          temp =(struct node *)malloc(1*sizeof(struct node));
          temp->ptr = top;
          temp->info = data;
          top = temp;
   }
}
/* Display stack elements */
void display()
{
   top1 = top;
   if (top1 == NULL)
   {
         printf("Stack is empty");
         return;
   }
   while (top1 != NULL)
   {
         printf("%d ", top1->info);
         top1 = top1->ptr;
   }
 }
/* Pop Operation on stack */
void pop()
{
   top1 = top;
   if (top1 == NULL)
   {
         printf("\n Error : Trying to pop from empty stack");
         return;
   }
   else
         top1 = top1->ptr;
   printf("\n Popped value : %d", top->info);
   free(top);
   top = top1;
}
void destroy()
{
   top1 = top;
   while (top1 != NULL)
   {
         top1 = top->ptr;
         free(top);
         top = top1;
```

```
        top1 = top1->ptr;
    }
    free(top1);
    top = NULL;
    printf("\n All stack elements destroyed");
    }
```

2.3 Polish Expressions (PN)

Polish notation (PN), also known as normal Polish notation (NPN), Polish prefix notation or simply prefix notation is a form of notation for logic, arithmetic, and algebra. Its distinguishing feature is that it places operators to the left of their operands

Infix Version	**PN Version**
3 – 2	-3 2
3 * 2+5	+* 3 25
(12 - 3) /3	/ -12 3 3

2.4 Reverse Polish Expressions (RPN)

Reverse Polish Notation (RPN), sometimes referred to as postfix notation, is a way of writing mathematical expressions where each operand is preceded by the two operators it applies to and looks something like 2 5 3 + * instead of (5 + 3) * 2

Well, it turns out that it is easier/more efficient for computers to parse an RPN expression than an infix one. In fact, if you look at the code underlying most calculators you'd find that most of them take the infix expressions that we write and first convert them to postfix before evaluating them. Meta calculator's graphing and scientific calculators both do exactly this.

Examples

Infix Version	**RPN Version**
3 – 2	3 2 -
3 * 2+5	5 3 2 * +

(12 - 3) /3	12 3 - 3 /

So what is wrong with infix?
The big problem is that operators have properties such as precedence and associativity. This makes working out with an infix expression actually makes harder than it should be.

For example, multiplication has a higher precedence or priority than addition and this means that:

2+3*4; is not 2+3 all times 4 which it would be in a strict left to right reading. In fact the expression is 3*4 plus 2 and you can see that evaluating an infix expression often involves reordering the operators and their operands.

Then there arises a small matter of inserting brackets to make some infix expressions clear.

For example: (2+3)*(4+5) cannot be written without brackets because 2+3*4+5 means 3*4 plus 2 plus 5.

The order that you have to evaluate operators is something that takes a long time to learn. Beginners of arithmetic may get wrong answers even when they do the actual operations correctly and it gives rise to the need to remember mnemonics like BODMAS - Brackets Of Division, Multiplication, Addition, Subtraction. Hence, postfix or RPN is proved efficient as far as computer calculation is concerned.

Infix notation is easy to read for *humans*, whereas pre-/postfix notation is easier to parse for a machine. The big advantage in pre-/postfix notation is that there never arise any questions like operator precedence.
For example, consider the infix expression 1 # 2 $ 3. Now, we don't know what those operators mean, so there are two possible corresponding postfix expressions: 1 2 # 3 $ and 1 2 3 $ #. Without knowing the rules governing the use of these operators, the infix expression is essentially worthless.
Or, to put it in more general terms: it is possible to restore the original (parse) tree from a pre-/postfix expression without any additional knowledge, but the same isn't true for infix expressions.

A typical Polish expression and an equivalent infix form

```
− × ÷ 15 − 7 + 1 1 3 + 2 + 1 1 =
− × ÷ 15 − 7 2     3 + 2 + 1 1 =
− × ÷ 15 5         3 + 2 + 1 1 =
− × 3              3 + 2 + 1 1 =
− 9                + 2 + 1 1 =
− 9                + 2 2   =
− 9                4         =
Answer is 5
```

An equivalent in-fix is as follows: ((15 ÷ (7 − (1 + 1))) × 3) − (2 + (1 + 1)) = 5

2.5 Infix to postfix conversion
Now that we have seen different forms of expression, let us try to convert an expression of one form to another. We can manually convert the given expression.
A/B^C-D
Parse the expression from left to right. Since ^ has precedence than other operators in the expression, consider its operands and convert it into immediate postfix form. Now, the expression looks like

A/BC^-D, say BC^ as **E1** => A/**E1**-D

Since, / has higher precedence, consider its operands and convert it into its immediate postfix notation. i.e.,AE1/-D. Now say the converted portion as E2 and rewrite the expression
E2-D
Finally, it becomes **E2**D-. [Recall; E2 -> AE1/, & E1 ->
BC^] Now its time to substitute, A**E1**/D-ABC^/D-

That's it!!
Computer converts the same expression using stack and here comes one of the main applications of stack. The below shown tabular conversion would best explain the overall process.

Expression	**Current Symbol**	**Stack**	**Output**	**Comment**
A/B^C-D	Initial State	NULL	–	Initially Stack is Empty
/B^C-D	A	NULL	A	Print Operand
B^C-D	/	/	A	Push Operator Onto Stack
^C-D	B	/	AB	Print Operand
C-D	^	/^	AB	Push Operator Onto Stack because Priority of ^ is greater than Current Topmost Symbol of Stack i.e '/'
-D	C	/^	ABC	Print Operand
D	–	/	ABC^	**Step 1:** Now '^' Has Higher Priority than Incoming Operator So We have to Pop Topmost element. **Step 2:** Remove Topmost Operator From Stack and Print it
D	–	NULL	ABC^/	**Step 1:** Now '/' is topmost Element of Stack Has Higher Priority than Incoming Operator So We have to Pop Topmost Element again. **Step 2:** Remove Topmost Operator From Stack and Print it
D	–	–	ABC^/	**Step 1:** Now Stack Becomes Empty and

				We can Push Operand Onto Stack
NULL	D	–	ABC^/D	Print Operand
NULL	NULL	–	ABC^/D-	Expression Scanning Ends but we have still one more element in stack so pop it and display it

2.6 Evaluation of Postfix Expression

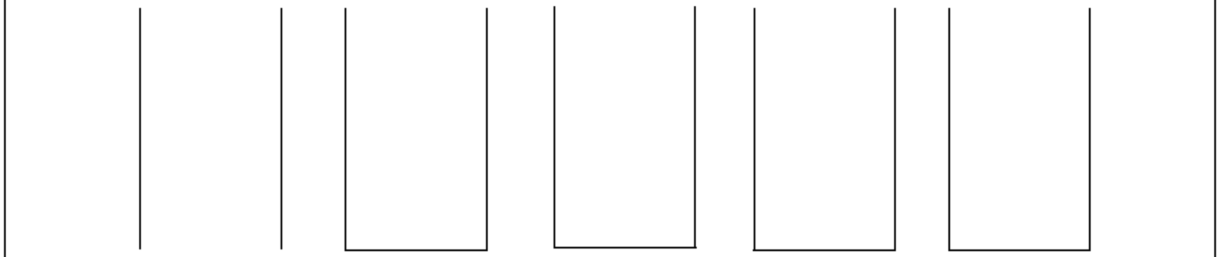
The Postfix notation is used to represent algebraic expressions. The expressions written in postfix form are evaluated faster compared to infix notation as parenthesis are not required in postfix. We have discussed infix to postfix conversion. In this post, evaluation of postfix expressions is discussed. Following is algorithm for evaluating postfix expressions.

1) Create a stack to store operands (or values).
2) Scan the given expression and do following for every scanned element.
…..a) If the element is a number, push it into the stack
…..b) If the element is a operator, pop operands for the operator from stack. Evaluate the operator and push the result back to the stack
3) When the expression ends, the number in the stack is the final answer

Example:

Let the given expression be "2 3 1 * + 9 -". We scan all elements one by one

Try it yourself



1) Scan '2', it's a number, so push it to stack. Stack contains '2'
2) Scan '3', again a number, push it to stack, stack now contains '2 3′ (from bottom to top)
3) Scan '1', again a number, push it to stack, stack now contains '2 3 1′
4) Scan '*', it's an operator, pop two operands from stack, apply the * operator on operands, we get 3*1

which results in 3. We push the result '3' to stack. Stack now becomes '2 3′.
5) Scan '+', it's an operator, pop two operands from stack, apply the + operator on operands, we get 3 + 2 which results in 5. We push the result '5' to stack. Stack now becomes '5'.
6) Scan '9', it's a number, we push it to the stack. Stack now becomes '5 9′.
7) Scan '-', it's an operator, pop two operands from stack, apply the – operator on operands, we get 5 – 9 which results in -4. We push the result '-4′ to stack. Stack now becomes '-4′.
8) There are no more elements to scan, we return the top element from stack (which is the only element left in stack).

2.7 Infix to prefix conversion

Now that we have seen different forms of expression, let us try to convert an expression of one form to another. We can manually convert the given expression.

A/B^C-D

Parse the expression from left to right. Since ^ has precedence than other operators in the expression, consider its operands and convert it into immediate postfix form. Now, the expression looks like

A/^BC-D, say ^BC as **E1** => A/**E1**-D

Since, / has higher precedence, consider its operands and convert it into its immediate postfix notation. i.e., /AE1-D. Now say the converted portion as E2 and rewrite the expression

E2-D

Finally, it becomes - **E2**D. [Recall; E2 -> /AE1 , & E1 -> ^BC]

Now its time to substitute, A**E1**/D-

-/A^BCD

That's it!!

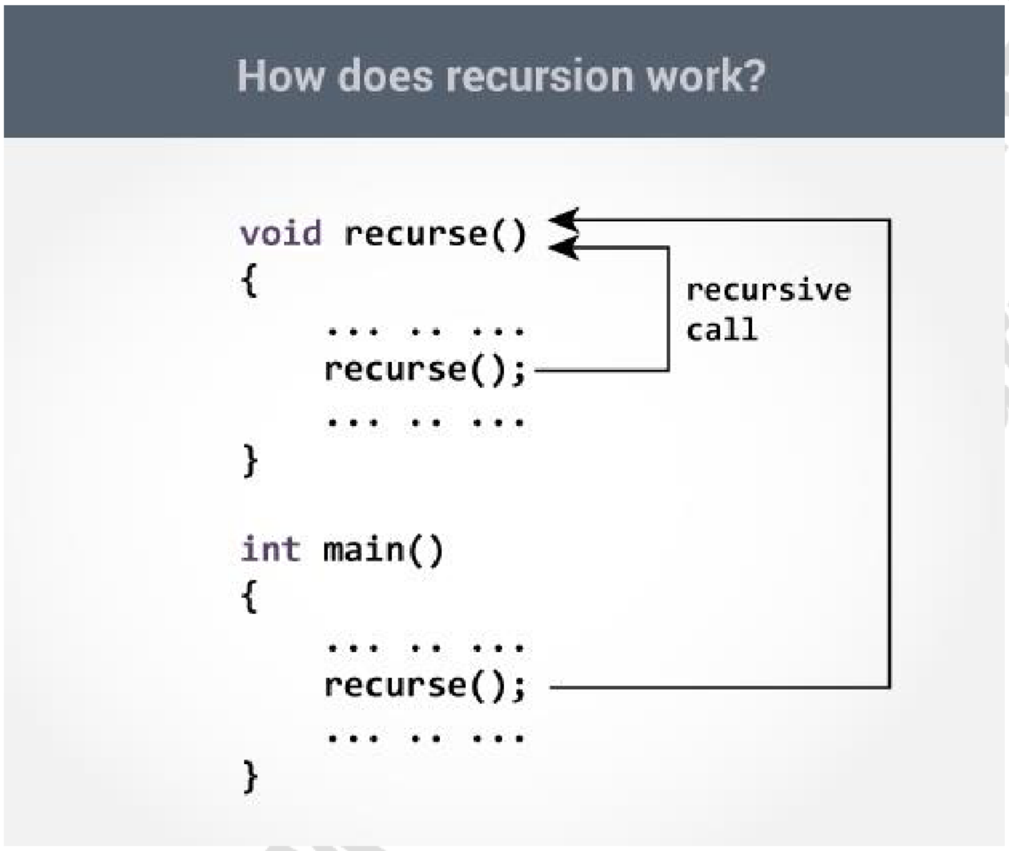
2.8 Recursion

A function that calls itself is known as recursive function. And, this technique is known as recursion.

How recursion works?

```
void recurse()
{
   ... .. ...
   recurse();
   ... .. ...
}
int main()
```

```
{
    ... .. ...
    recurse();
    ... .. ...
}
```


How does recursion work?

```
void recurse()
{
    ... .. ...
    recurse();
    ... .. ...
}

int main()
{
    ... .. ...
    recurse();
    ... .. ...
}
```

The recursion continues until some condition is met to prevent it. To prevent infinite recursion, if...else statement (or similar approach) can be used where one branch makes the recursive call and other doesn't.

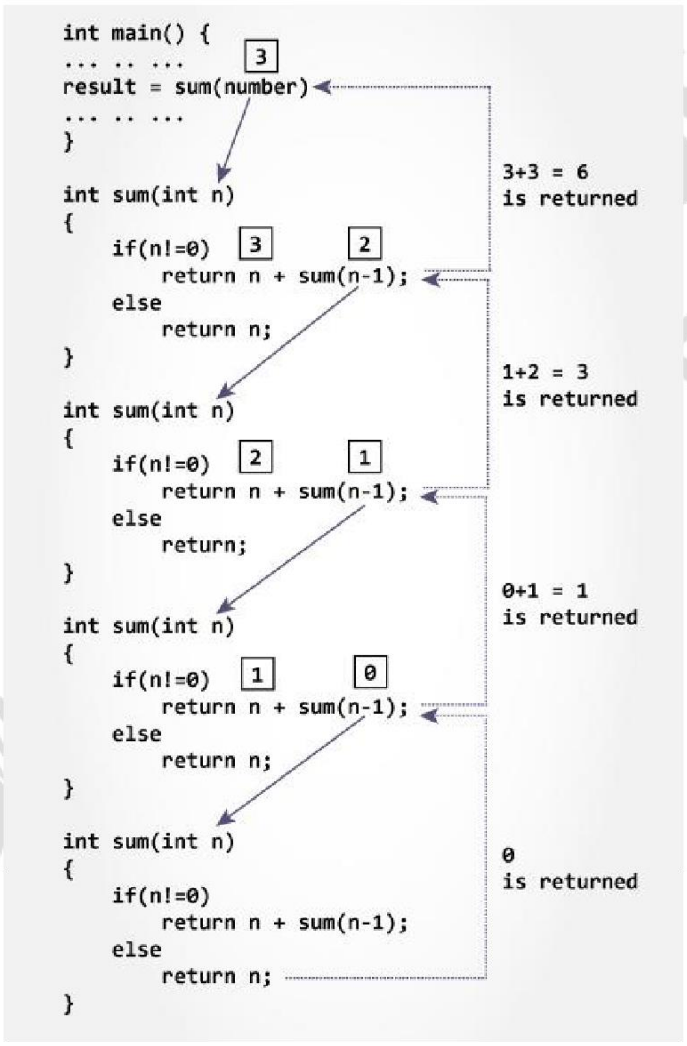
An example program in C

```
#include <stdio.h>
int sum(int n);
int main()
{
    int number, result;
    printf("Enter a positive integer: ");
    scanf("%d", &number);
    result = sum(number);
    printf("sum=%d", result);
}
int sum(int n)
```

```
{
   if (n!=0)
       return n + sum(n-1); // sum() function calls itself
   else
       return n;
}
```

Initially, the sum( ) is called from the ma in() function with number passed as an argument. Suppose the value of n is 3 initially. During next function call, 2 is passed to the su m() function. In next function call, 1 is passed to the function. This process continues until n is equal to 0. When n is equal to 0, there is no recursive call and the sum of integers is returned to

the ma in() function.



```
int main() {
... .. ...
result = sum(number)
... .. ...
}

int sum(int n)
{
    if(n!=0)
        return n + sum(n-1);
    else
        return n;
}

int sum(int n)
{
    if(n!=0)
        return n + sum(n-1);
    else
        return;
}

int sum(int n)
{
    if(n!=0)
        return n + sum(n-1);
    else
        return n;
}

int sum(int n)
{
    if(n!=0)
        return n + sum(n-1);
    else
        return n;
}
```

Advantages and Disadvantages of Recursion

Recursion makes program elegant and cleaner. All algorithms can be defined recursively which makes it easier to visualize and prove.

If the speed of the program is vital then, you should avoid using recursion. Recursions use more memory and are generally slow.

C Program to find factorial of a given number using recursion

```
#include <stdio.h>
int factorial(int);
void main()
{
int result,num;
scanf("%d",&num);
result=factorial(num);
printf("%d",result);
}
int factorial(int ele)
{
if(ele==0)
return 1;
else
return (ele*factorial(ele-1));
}
```

Output

```
Enter a positive integer: 6
Factorial of 6 = 720
```

Suppose the user entered 6. Initially, the multiplyNumbers() is called from the main() function with 6 passed as an argument. Then, 5 is passed to the multiplyNumbers() function from the same function (recursive call). In each recursive call, the value of argument nis decreased by 1. When the value of n is less than 1, there is no recursive call.

C Program to find GCD of Two Numbers using Recursion

```
#include <stdio.h>
int gcd(int, int);
int main()
{
int a,b,result;
printf("Enter two numbers\n");
scanf("%d%d",&a,&b);
result=gcd(a,b);
printf("GCD:%d\n",result);
}
int gcd(int a, int b)
{
```

```
int ans;
if(b<a && a%b==0)
return b;
else if(a<b)
return gcd(b,a);
else
{
ans=a%b;
return gcd(b,ans);
}
}
```

Output

```
Enter two positive integers: 366
60
G.C.D of 366 and 60 is 6.
```

C Program to generate Fibonacci Sequence using Recursion

```
#include<stdio.h>
 int Fibonacci(int);
 int main()
{
   int n, i = 0, c;
    scanf("%d",&n);
    printf("Fibonacci series\n");
    for ( c = 1 ; c <= n ; c++ )
   {
      printf("%d\n", Fibonacci(i));
      i++;
   }
    return 0;
}
 int Fibonacci(int n)
{
   if ( n == 0 )
```

```
    return 0;
  else if ( n == 1 )
    return 1;
  else
    return ( Fibonacci(n-1) + Fibonacci(n-2) );
}
```

Output

```
Enter the range of the Fibonacci series: 10
Fibonacci Series: 0 1 1 2 3 5 8 13 21 34 55 89
```

C Program to solve Towers of Hanoi problem using Recursion

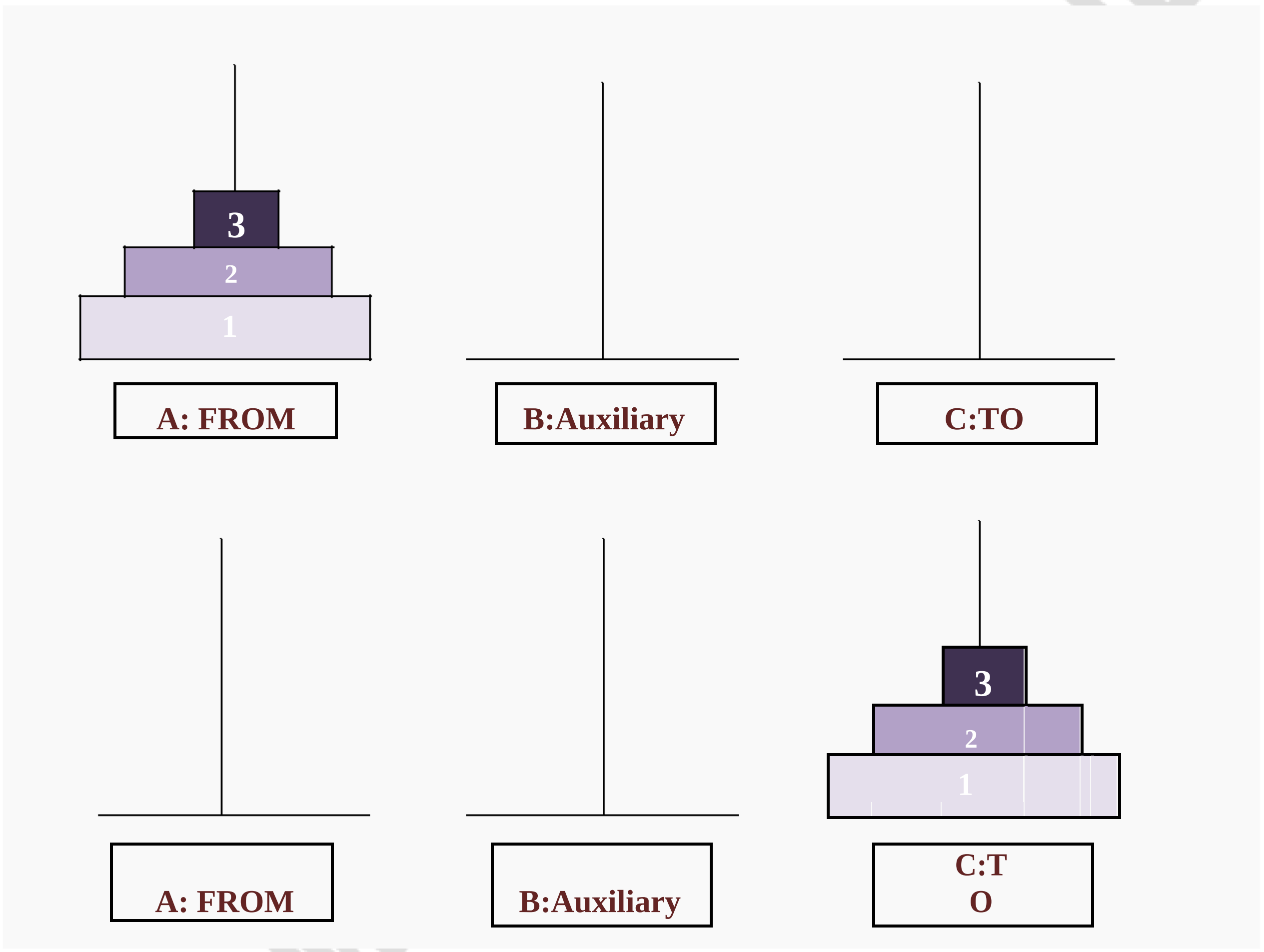
```
#include <stdio.h>
void towers(int, char, char, char);
int main()
{
   int num;

   printf("Enter the number of disks : ");
   scanf("%d", &num);
   printf("The sequence of moves involved in the Tower of Hanoi are :\n");
   towers(num, 'A', 'C', 'B');
   return 0;
}
void towers(int num, char frompeg, char topeg, char auxpeg)
{
   if (num == 1)
   {
      printf("\n Move disk 1 from peg %c to peg %c", frompeg,
      topeg); return;
   }
   towers(num - 1, frompeg, auxpeg, topeg);
   printf("\n Move disk %d from peg %c to peg %c", num, frompeg, topeg);
   towers(num - 1, auxpeg, topeg, frompeg);
}
```

In the above program, towers (…) is the function that shows its execution in a recursive manner.

OK, now let us see the problem constraints of Tower of Hanoi and let us try to understand how it can be addressed non-programmatically….

Problem: Move all the disks over to the rightmost tower, one at a time, so that they end up in the original order on that tower. You may use the middle tower as temporary storage, but at no time during the transfer should a larger disk be on top of a smaller one.



What you are observing is the problem (in the upper part) and its solution (in the lower part). If we manually try to work it out keeping all the constraints in mind, the sequence of steps that one would perform for 3 disks:

Move disk 1 from peg A to peg C

Move disk 2 from peg A to peg B

Move disk 1 from peg C to peg B

Move disk 3 from peg A to peg C

Move disk 1 from peg B to peg A

Move disk 2 from peg B to peg C

Move disk 1 from peg A to peg C

O K….. now that you got an idea of how this problem can be solved manually. But, we being programmers should find out solution with computer as a tool. Let's do that…..

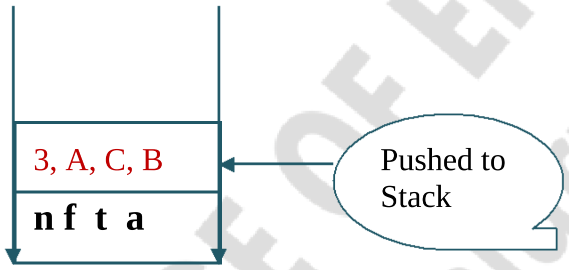
For simplicity purpose, I have annotated disks as 1, 2, and 3, and stands as

A->From, B->Auxiliary and C->To

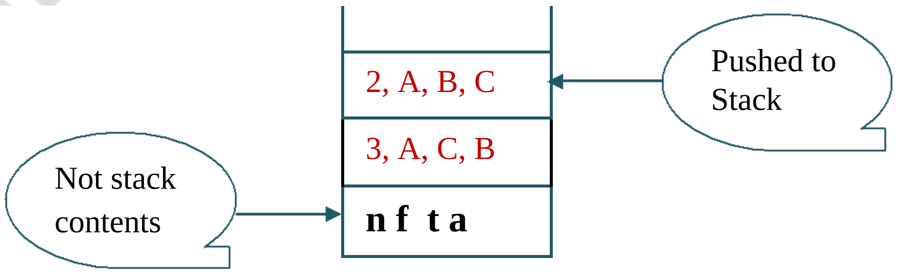
A small note to you all, it would be better if you take a copy (print or handwritten) of the program shown above in your hand for analysis purpose otherwise you need to scroll several times.

Yup… I am starting my explanation now… Please don't get exhausted ☺ ☺
Firstly, the function call invokes the called function upon supplying the parameters (as shown below)
towers(num, 'A','C','B');
void towers(int num, char frompeg, char topeg, char auxpeg) -------------------------------(1)
So, recursion makes use of stack, the stack content would be:

.

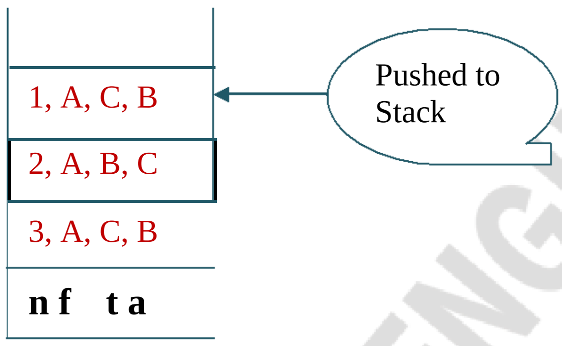


Let us interpret it as follows:
num=3; frompeg=A; topeg=C; auxpeg=B;
After this, a condition is checked i.e., if(num==1) ? As of now, it's not.
A recursive call takes place after that;
towers(num - 1, frompeg, auxpeg, topeg); i.e., num-1 = 2; frompeg=A; auxpeg=B; topeg=C; (From the figure)
Now, these parameter values will be mapped to the parameters in (1)
Resulting into as below,
num = num-1 = 2; frompeg = frompeg = A; topeg = auxpeg = B; auxpeg = topeg = C;
Now the stack content would look like this

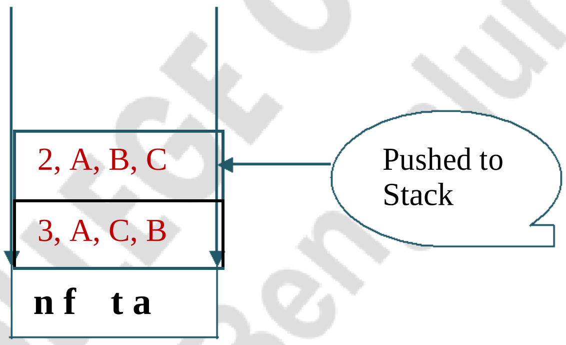


Once again, the condition is checked i.e., if(num==1) ? it's not again.
One more time recursive call takes place:
towers(num - 1, frompeg, auxpeg, topeg); i.e., num-1 = 1; frompeg=A; auxpeg=C; topeg=B;
Now, these parameter values will be mapped to the parameters in (1)
Resulting into as below,
num = num-1 = 1; frompeg = frompeg = A; topeg = auxpeg = C; auxpeg = topeg = B;

Now the stack content would look like this

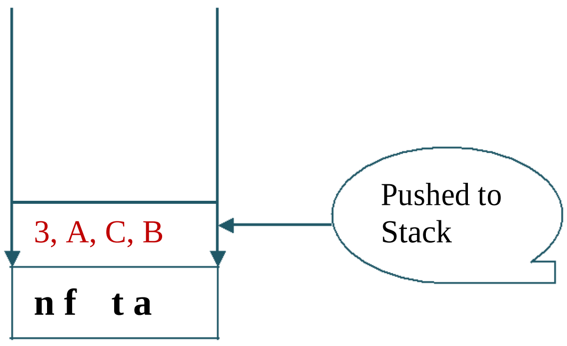


This time the condition satisfies i.e., num =1 now. Top of the stack will be popped as recursion gets a break.
The statement inside will get executed and the following statement would get printed
Move disk 1 from peg A to peg C ; num=1, frompeg = A, topeg = C



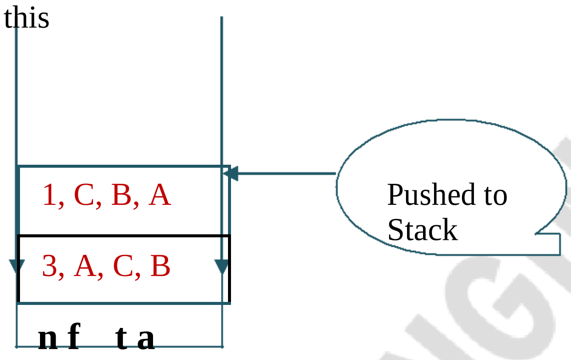
Here, the first recursive call gets totally a break.
Immediately, the next print statement gets executed by popping the stack contents.
Move disk 2 from peg A to peg B ; num=2, frompeg = A, topeg = B

Stack contents now,

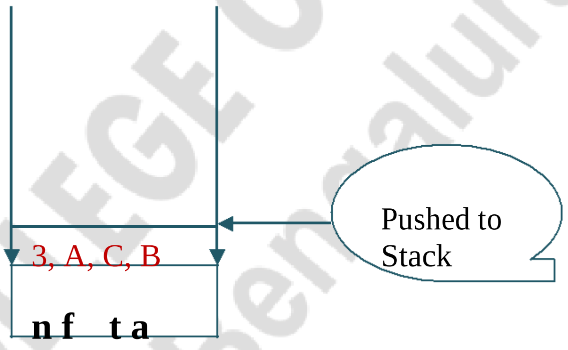


Now with the popped parameter values, second function call invokes the function.
towers(num - 1, auxpeg, topeg, frompeg); num-1 = 1; auxpeg = C; topeg = B; frompeg = A;
[referring to stack top of figure X]
Now, the above parameter values will be mapped to the parameters in (1)
Resulting into as below,
num = num-1 = 1; frompeg = auxpeg = C; topeg = topeg = B; auxpeg = frompeg = A;

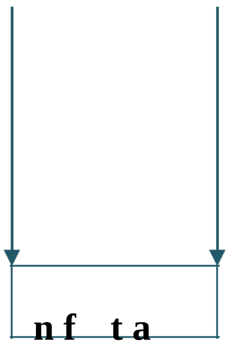
Now the stack content would look like this



Since the condition if(num == 1) satisfies, the print statement would get executed by taking the popped parameters.
Hence, we see
Move disk 1 from peg C to peg B; num=1, frompeg = C, topeg =
B Stack contents now,

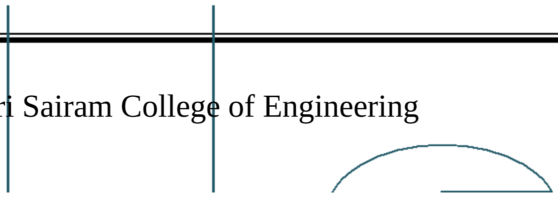


Finally, the stack top is popped and the parameters are used for executing the print statement outside the condition. Stack is empty now.



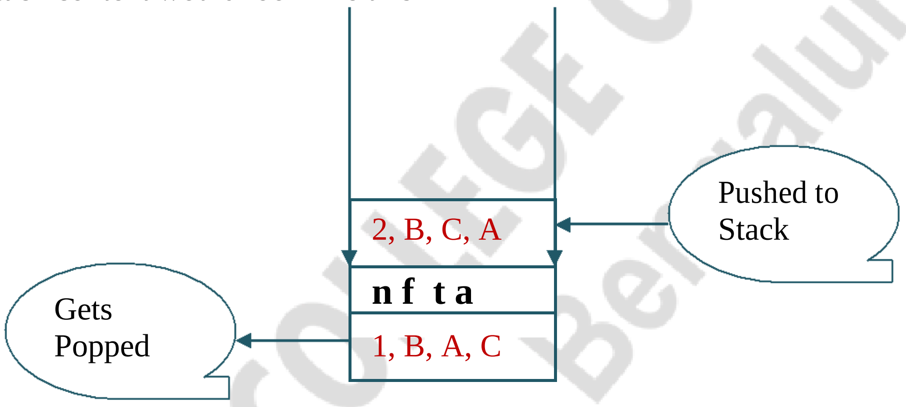
Move disk 3 from peg A to peg C; num=3, frompeg = A, topeg = C
Now, the second recursive function is processed with the popped values and the called function is executed by pushing the contents onto the stack.
towers(num - 1, auxpeg, topeg, frompeg); num-1 = 2; auxpeg = B; topeg = C; frompeg = A;
[referring to stack top of figure Y]

Now, the above parameter values will be mapped to the parameters in (1)
Resulting into as below,
num = num-1 = 2; frompeg = auxpeg = B; topeg = topeg = C; auxpeg = frompeg = A;
Now the stack content would look like this



Pushed to

The condition will not hold because num = 2. Now, the first recursive call invokes the function by supplying parameters onto the stack.
towers(num - 1, frompeg, auxpeg, topeg); i.e., num-1 = 1; frompeg=B; auxpeg=C; topeg=A;
Now, the above parameter values will be mapped to the parameters in (1) Resulting into as below,
num = num-1 = 1; frompeg = frompeg = B; topeg = auxpeg = C; auxpeg = topeg = A;
Now the stack content would look like this



Move disk 1 from peg B to peg A; num=1, frompeg = B, topeg = A
Next, Using the stack top contents, print statement will be executed by popping out stack top. [Refer to the above figure; minus popped contents].
Move disk 2 from peg B to peg C; num=2, frompeg = B, topeg =
C Stack becomes empty.
Now, the second recursive call will be invoked by supplying the popped contents as input.
towers(num - 1, auxpeg, topeg, frompeg); num-1 = 1; auxpeg = A; topeg = C; frompeg = B;
[referring to stack top of figure Z]
Now, the above parameter values will be mapped to the parameters in (1)
Resulting into as below,
num = num-1 = 1; frompeg = auxpeg = A; topeg = topeg = C; auxpeg = frompeg = B;

The 'if' condition holds, hence print statement executes popping the stack.
Move disk 1 from peg A to peg C; num=1, frompeg = A, topeg = C
Finally, Stack goes empty. … I hope mind is not ☺ ☺ ☺
Hope my explanation has reached you and clarified few of your doubts in this regard.

2.9 Ackerman's Recursive Function

In computability theory, the Ackermann function, named after Wilhelm Ackermann, is one of the simplest and earliest-discovered examples of a total computable function that is not primitive recursive. All primitive recursive functions are total and computable, but the Ackermann function illustrates that not all total computable functions are primitive recursive.

The Ackermann function $A(x, y)$ is defined for integer x and y by

$$A(x, y) \equiv \begin{cases} y + 1 & \text{if } x = 0 \\ A(x-1, 1) & \text{if } y = 0 \\ A(x-1, A(x, y-1)) & \text{otherwise.} \end{cases}$$

C program to illustrate Ackermann's Function

```
#include<stdio.h>
static int w=0;
int ackerman(int m,int n)
{
   w=w+1;
   if(m==0)
      return n+1;
   else if(m>0 && n==0)
      return ackerman(m-1,1);
   else if(m>0 && n>0)
      return ackerman(m-1,ackerman(m,n-1));
}
int mainackerman()
{
   int m,n;
   scanf("%d %d",&m,&n);
   printf("%d %d",ackerman(m,n),w);
   return 0;
}
```

2.10 Dynamic Implementation of stack

```
#include <stdio.h>
#include <stdlib.h>
struct node
{
   int info;
   struct node *ptr;
```

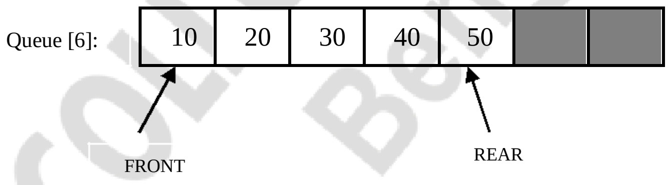
```
}*top,*top1,*temp;
void push(int data);
void pop();
void display();
void destroy();
void main()
{
   int no, ch, e;
   printf("\n 1 - Push");
   printf("\n 2 - Pop");
   printf("\n 3 - Exit");
   printf("\n 4 - Display");
   printf("\n 5 - Destroy stack");
   top=NULL;
   while (1)
   {
         printf("\n Enter choice : ");
         scanf("%d", &ch);
         switch (ch)
         {
         case 1:
            printf("Enter data : ");
            scanf("%d", &no);
            push(no);
            break;
         case 2:
            pop();
            break;
         case 3:
            exit(0);
         case 4:
            display();
            break;
         case 5:
            destroy();
            break;
         default :
            printf(" Wrong choice, Please enter correct choice  ");
            break;
         }
   }
}
void push(int data)
{
   if (top == NULL)
   {
         top =(struct node *)malloc(1*sizeof(struct node));
         top->ptr = NULL;
         top->info = data;
   }
   else
   {
         temp =(struct node *)malloc(1*sizeof(struct node));
         temp->ptr = top;
         temp->info = data;
         top = temp;
   }
}
/* Display stack elements */
void display()
{
   top1 = top;
   if (top1 == NULL)
   {
         printf("Stack is empty");
         return;
   }
   while (top1 != NULL)
   {
         printf("%d ", top1->info);
         top1 = top1->ptr;
   }
 }
/* Pop Operation on stack */
void pop()
{
   top1 = top;
   if (top1 == NULL)
   {
         printf("\n Error : Trying to pop from empty stack");
         return;
   }
   else
         top1 = top1->ptr;
   printf("\n Popped value : %d", top->info);
   free(top);
   top = top1;
}
void destroy()
{
   top1 = top;
   while (top1 != NULL)
   {
         top1 = top->ptr;
         free(top);
         top = top1;
         top1 = top1->ptr;
   }
   free(top1);
   top = NULL;
   printf("\n All stack elements destroyed");
   }
```

2.11 Queues

Queue is an abstract data structure, somewhat similar to Stack. In contrast to Stack, queue is opened at both end. One end is always used to insert data, that end is referred to as **REAR** and the operation is called **enqueue** and the other end referred to as **FRONT**, is used to remove data and the operation is called **dequeue**. Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

A real world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world example can be seen as queues at ticket windows & bus-stops.

Logical representation of Queue data structure in computer's memory



The representation of queue is using arrays. Initially, FRONT & REAR indices are initialized to -1. As elements are inserted, rear is incremented by 1 and as elements are deleted, front is incremented by 1. Shaded cells in the above figure are empty. If an element is added, it sits in the immediate next cell to rear and rear shifts right by one cell (i.e., rear++). Similarly if an element is deleted, front shifts right by one cell (i.e. front++).

Applications of Queue

Queue, as the name suggests is used whenever we need to have any group of objects in an order in which the first one coming in, also gets out first while the others wait for their turn, like in the following scenarios:

1. Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
2. In real life, Call Center phone systems will use Queues, to hold people calling them in an order, until a service representative is free.
3. Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive, First come first served.

Array Implementation of a Queue

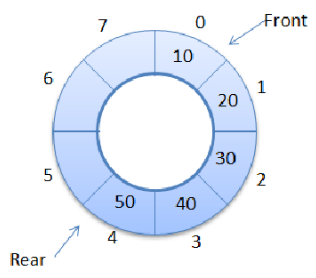
```
#include <stdio.h>
 #define MAX 50
```

```
int queue_array[MAX];
int rear = - 1;
int front = - 1;
main()
{
   int choice;
   while (1)
   {
      printf("1.Insert element to queue \n");
      printf("2.Delete element from queue \n");
      printf("3.Display all elements of queue \n");
      printf("4.Quit \n");
      printf("Enter your choice : ");
      scanf("%d", &choice);
      switch (choice)
      {
         case 1:
         insert();
         break;
         case 2:
         delete();
         break;
         case 3:
         display();
         break;
         case 4:
         exit(1);
         default:
         printf("Wrong choice \n");
      } /*End of switch*/
   } /*End of while*/
} /*End of main()*/
insert()
{
    int add_item;
    if (rear == MAX - 1)
    printf("Queue Overflow \n");
    else
    {
        if (front == - 1)
        /*If queue is initially empty */
        front = 0;
        printf("Inset the element in queue : ");
        scanf("%d", &add_item);
        rear = rear + 1;
        queue_array[rear] = add_item;
    }
} /*End of insert()*/

delete()
{
    if (front == - 1 || front > rear)
    {
        printf("Queue Underflow \n");
        return ;
    }
    else
    {
        printf("Element deleted from queue is : %d\n", queue_array[front]);
        front = front + 1;
    }
} /*End of delete()
*/ display()
{
    int i;
    if (front == - 1) printf("Queue
        is empty \n");
    else
    {
        printf("Queue is : \n");
        for (i = front; i <= rear; i++)
            printf("%d ", queue_array[i]);
        printf("\n");
    }
}
```

2.12 Queue Variants: Circular Queue

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



In a standard queue data structure re-buffering problem occurs for each de queue operation. To solve this problem by joining the front and rear ends of a queue to make the queue as a circular queue.

1. Circular queue is a linear data structure. It follows FIFO principle.
2. In circular queue the last node is connected back to the first node to make a circle.
3. Circular linked list fallow the First In First Out principle
4. Elements are added at the rear end and the elements are deleted at front end of the queue
5. Both the front and the rear pointers points to the beginning of the array.
6. It is also called as “Ring buffer”.

2.13 Queue Variants: Priority Queue

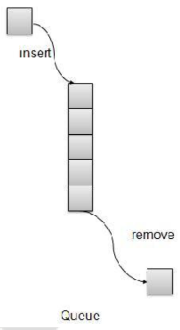
Priority Queue is more specialized data structure than Queue. Like ordinary queue, priority queue has same method but with a major difference. In Priority queue items are ordered by key value so that item with the lowest value of key is at **front** and item with the highest value of key is at **rear** or vice versa. So we're assigned priority to item based on its key value. Lower the value, higher the priority. Following are the principal methods of a Priority Queue.

Basic Operations

1. Insert / enqueue − add an item to the rear of the queue.

2. Remove / dequeue − remove an item from the front of the queue.
3. Peek − get the element at front of the queue.
4. isFull − check if queue is full.
5. isEmpty − check if queue is empty.

Priority Queue Representation



C program to demonstrate working of priority queue

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 5
int rear=-1,ch,front=-1,i,j,item,queue[SIZE],choice;
int Q_full();
int Q_Empty();
void insert();
void delet();
void display();
int main()
{
printf("1.Insert\n2.Delete\n3.Display\n4.Exit\n");
while(1)
{
printf("Enter your choice\n");
scanf("%d",&ch);
switch(ch)
{
case 1: if(Q_full())
        printf("Priority Queue is Full\n");
```

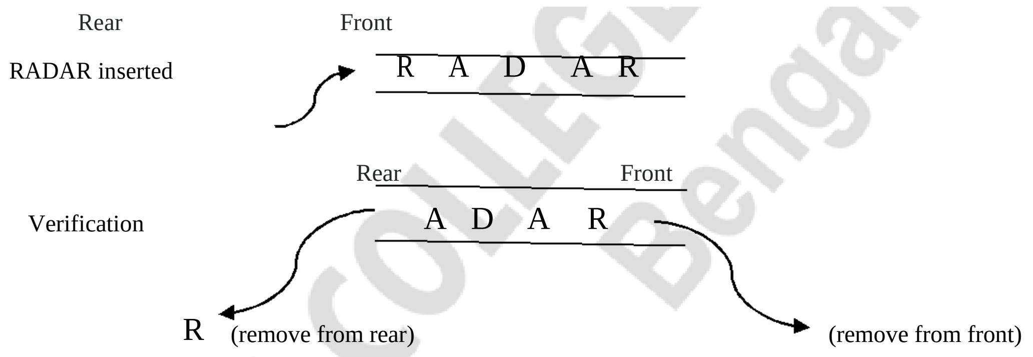
```
          else
          insert();
          break;
case 2: if(Q_Empty())
          printf("Priority Queue Empty\n");
          else
          delet();
          break;
case 3: if(Q_Empty())
          printf("Priority Queue Empty\n");
          else
          display();
          break;
case 4: exit(0);
}
}
}
void insert()
{
int ele;
printf("Enter the element\n");
scanf("%d",&item);
if(front==-1)
front++;
j=rear;
while(j>=0 && item<queue[j])
{
queue[j+1]=queue[j];
j--;
}
queue[j+1]=item;
rear++;
}
int Q_full()
```

```
{
if(rear==SIZE-1)
return 1;
else
return 0;
}
void delet()
{
printf("The item deleted is %d",queue[front]);
front++;
}
int Q_Empty()
{
if((front==-1)||(front>rear))
return 1;
else
return 0;
}
void display()
{
printf("Elements of Priority Queue...");
for(i=front;i<=rear;i++)
printf("%d\t",queue[i]);
}
```

2.14 Queue Variants: Double Ended (De-Queue)

A double-ended queue is an abstract data type similar to an simple queue, it allows you to insert and delete from both sides means items can be added or deleted from the front or rear end. Following reasons may help you understand why DE-Queue?

1. A nice application of the deque is storing a web browser's history. Recently visited URLs are added to the front of the deque, and the URL at the back of the deque is removed after some specified number of insertions at the front.
2. Another common application of the deque is storing a software application's list of undo operations.
3. One example where a deque can be used is the A-Steal job scheduling algorithm.[5] This algorithm implements task scheduling for several processors. A separate deque with threads to be executed is maintained for each processor. To execute the next thread, the processor gets the first element from the deque (using the "remove first element" deque operation). If the current thread forks, it is put back to the front of the deque ("insert element at front") and a new thread is executed. When one of the processors finishes execution of its own threads (i.e. its deque is empty), it can "steal" a thread from another processor: it gets the last element from the deque of another processor ("remove last element") and executes it
4. One of the best application of DE-Queue is in verifying whether given string is a palindrome?


Rear
Front
RADAR inserted
R A D A R
Rear
Front
Verification
A D A R
R
(remove from rear)
(remove from front)