CBCS Scheme

USN | | | | | | | | | | |

16MCA11

First Semester MCA Degree Examination, Dec.2016/Jan.2017
Data Structures Using C

Time: 3 hrs. Max. Marks: 80

Note: *Answer FIVE full questions, choosing one full question from each module.*

Module-1

1 a. Explain looping control structures of 'C' with syntax and examples. **(08 Marks)**

b. Differentiate between call by value and call by reference functions with programming examples. **(08 Marks)**

OR

2 a. Explain various types of operators that are supported in 'C' language. **(08 Marks)**

b. Explain with examples for passing array to functions and strings to functions. **(08 Marks)**

Module-2

3 a. What is pointer? Give an account of function returning a pointer with program example. **(08 Marks)**

b. Define structure. Explain how the structure variable passed as a parameter to a function with example. **(08 Marks)**

OR

4 a. What are arrays? Explain the concept of inserting an element in to an array. Write a program to implement the same. **(08 Marks)**

b. What are strings? Mention any five string operation. Write a program to concatenate 2 strings without using built in functions. **(08 Marks)**

Module-3

5 a. What are data structures? Explain the classifications of data structures. **(08 Marks)**

b. Define stack. Write a 'C' program to implement stack operations using arrays. **(08 Marks)**

OR

6 a. Convert the following infix expression in to postfix using applications of stack.

$A+(B*C-(D/E\hat{}F)*G)*H$ **(08 Marks)**

b. What is recursion? Write a program to implement towers of Hanoi problem using recursion and trace the output for 3 disks. **(08 Marks)**

Module-4

7 a. Implement a menu driven program in 'C' for the following operations on singly linked list

i) Perform insertion at the beginning
ii) Perform Deletion at the beginning
iii) Display the list
iv) Exit **(08 Marks)**

b. What are circular linked lists? Write 'C' functions to perform insertion and deletion operations at the end of circular lists. **(08 Marks)**

OR

8 a. What are Doubly linked lists? Write a program to implement stack operation using Doubly linked list. **(08 Marks)**

b. What are priority queues? Write a program to simulate the working of priority Queues. **(08 Marks)**

Module-5

9 a. Construct the BST for the items 40, 60, 50, 33, 55, 11. Write and explain 'C' module to insert an element in to BST. **(08 Marks)**

b. Write a C program to implement selection sort method and consider the elements to trace the output : 7, 3, 4, 1, 8, 2, 6, 5. **(08 Marks)**

OR

10 a. Write a program to implement Binary Tree Traversal methods. **(08 Marks)**

b. Give an account of

i) Hashing function

ii) Binary search. **(08 Marks)**

* * * * *

Module – 1

1. A) Explain looping control structures of 'C' with syntax and example

• C language provides a concept called loop, which helps in executing one or more statements up to desired number of times.
• Loops are used to execute one or more statements repeatedly.

• There are 3 types of loops in C programming:
1) while loop
2) for loop
3) do while loop

THE while LOOP

• A while loop statement can be used to execute a set of statements repeatedly as long as a given condition is true.
• The syntax is shown below:

```
while(expression)
{
statement1;
statement2;
}
```

• Firstly, the expression is evaluated to true or false.
• If the expression is evaluated to false, the control comes out of the loop without executing the body of the loop.
• If the expression is evaluated to true, the body of the loop (i.e. statement1) is executed.
• After executing the body of the loop, control goes back to the beginning of the while statement and expression is again evaluated to true or false. This cycle continues until expression becomes false.

Example: Program to display a message 5 times using while statement.

```
#include<stdio.h>
void main()
{
int i=1;
while(i<=5)
{
printf("Welcome to C language \n");
i=i+1;
}
}
```

Output:

```
Welcome to C language
Welcome to C language
Welcome to C language
Welcome to C language
Welcome to C language
```

THE for LOOP

• A for loop statement can be used to execute s set of statements repeatedly as long as a given condition is true.

The syntax is shown below:

```
for(expr1;expr2;expr3)
```

```
{
statement1;
}
```

• Here, expr1 contains initialization statement
expr2 contains limit test expression
expr3 contains updating expression
• Firstly, expr1 is evaluated. It is executed only once.
• Then, expr2 is evaluated to true or false.
• If expr2 is evaluated to false, the control comes out of the loop without executing the body of the loop.
• If expr2 is evaluated to true, the body of the loop (i.e. statement1) is executed.
• After executing the body of the loop, expr3 is evaluated.
• Then expr2 is again evaluated to true or false. This cycle continues until expression becomes false.

Example: Program to display a message 5 times using for statement.

```
#include<stdio.h>
void main()
{
int i;
for(i=1;i<=5;i++)
{
printf("Welcome to C language \n");
}
}
```

Output:

```
Welcome to C language
Welcome to C language
Welcome to C language
Welcome to C language
Welcome to C language
```

THE do while STATEMENT

• When we do not know exactly how many times a set of statements have to be repeated, do-while statement can be used.
• The syntax is shown below:

```
do
{
statement1;
}while(expression);
```

• Firstly, the body of the loop is executed. i.e. the body of the loop is executed at least once.
• Then, the expression is evaluated to true or false.
• If the expression is evaluated to true, the body of the loop (i.e. statement1) is executed
• After executing the body of the loop, the expression is again evaluated to true or false. This cycle continues until expression becomes false.

Example: Program to display a message 5 times using do while statement.

```
#include<stdio.h>
void main()
{
int i=1;
do
{
printf("Welcome to C language \n");
```

```
i=i+1;
}while(i<=5);
}
```
Output:
Welcome to C language
Welcome to C language
Welcome to C language
Welcome to C language
Welcome to C language

1.b) Differentiate between call by value and call by reference with programming example

Call by value

• In this type, value of actual arguments are passed to the formal arguments and the operation is done on the formal arguments.
• Any changes made in the formal arguments does not effect the actual arguments because formal arguments are photocopy of actual arguments.
• Changes made in the formal arguments are local to the block of called-function.
• Once control returns back to the calling-function the changes made vanish.
• Example: Program to send values using call-by-value method.

```
#include <stdio.h>
void main()
{ int x,y,
printf("enter values of x & y : ");
scanf("%d %d ", &x, &y);
printf("\n old values x=%d y =%d", x, y);
change(x,y) ;
printf("\n new values x=%d y =%d", x, y);
}
void change(int a, int b)
{
k=a;
a=b;
b=k;
return;
}
```
Output:
enter values of x & y : 2 3
old values x=2 y =3
new values x=2 y =3

Call by reference

When, argument is passed using pointer, address of the memory-location is passed instead of value.
• Example: Program to swap 2 number using call by reference.

```
#include<stdio.h>
void swap(int *a,int *b)
{ // pointer a and b points to address of num1 and num2 respectively
int temp;
temp=*a;
*a=*b;
*b=temp;
}
void main()
```

```
{
int num1=5,num2=10;
swap(&num1, &num2); //address of num1 & num2 is passed to swap function
printf("Number1 = %d \n",num1);
printf("Number2 = %d",num2);
}
```

Output:

Number1 = 10
Number2 = 5

Explanation

• The address of memory-location num1 and num2 are passed to function and the pointers *a and *b accept those values.
• So, the pointer a and b points to address of num1 and num2 respectively.
• When, the value of pointer is changed, the value in memory-location also changed correspondingly.
• Hence, change made to *a and *b was reflected in num1 and num2 in main function.
• This technique is known as call by reference.

2.a) Explain the various types of operator that are supported in 'C' languages

OPERATOR

• An operator can be any symbol like + - * / that specifies what operation need to be performed on the data.
• For ex:
+ indicates add operation
* indicates multiplication operation

Operand

• An operand can be a constant or a variable.

Expression

• An expression is combination of operands and operator that reduces to a single value.
• For ex:
Consider the following expression a+b
here a and b are operands
while + is an operator

CLASSIFICATION OF OPERATORS

Operator Name For Example

Arithmetic operators + - * / %
Increment/decrement operators ++ --
Assignment operators =
Relational operators < > ==
Logical operators && || ~
Conditional operator ?:
Bitwise operators & | ^
Special operators []

ARITHMETIC OPERATORS

• These operators are used to perform arithmetic operations such as addition, subtraction,
• There are 5 arithmetic operators:

Operator Meaning of Operator

+ addition
- subtraction
* multiplication
/ division
% modulos

• Division symbol (/)

→ divides the first operand by second operand and
→ returns the quotient.
Quotient is the result obtained after division operation.
• Modulus symbol (%)
→ divides the first operand by second operand and
→ returns the remainder.
Remainder is the result obtained after modulus operation.
• To perform modulus operation, both operands must be integers.
• Program to demonstrate the working of arithmetic operators.

```
#include<stdio.h>
void main()
{
int a=9,b=4,c;
c=a+b;
printf("a+b=%d \n", c);
c=a-b;
printf("a-b=%d \n", c);
c=a*b;
printf("a*b=%d \n", c);
c=a/b;
printf("a/b=%d \n", c);
c=a%b;
printf("Remainder when a divided by b=%d ", c);
}
```

Output:

```
a+b=13
a-b=5
a*b=36
a/b=2
Remainder when a divided by b=1
```

INCREMENT OPERATOR

• ++ is an increment operator.
• As the name indicates, increment means increase, i.e. this operator is used to increase the value of a variable by 1.
• For example:
If b=5
then b++ or ++b; // b becomes 6
• The increment operator is classified into 2 categories:
1) Post increment Ex: b++
2) Pre increment Ex: ++b
• As the name indicates, post-increment means first use the value of variable and then increase the value of variable by 1.
• As the name indicates, pre-increment means first increase the value of variable by 1 and then use the updated value of variable.
• For ex:
If x is 10,
then z= x++; sets z to 10
but z = ++x; sets z to 11
Example: Program to illustrate the use of increment operators.

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

```
int x=10,y = 10, z ;
z= x++;
printf(“ z=%d x= %d\n”, z, x);
z = ++y;
printf(“ z=%d y= %d”, z, y);
}
```
Output:
z=10 x=11
z=11 y=11

DECREMENT OPERATOR

• -- is a decrement operator.
• As the name indicates, decrement means decrease, i.e. this operator is used to decrease the value of a variable by 1.
• For example:
If b=5
then b-- or --b; // b becomes 4
• Similar to increment operator, the decrement operator is classified into 2 categories:
1) Post decrement Ex: b--
2) Pre decrement Ex: --b
• For ex:
If x is 10,
then z= x--; sets z to 10,
but z = --x; sets z to 9.
Example: Program to illustrate the use of decrement operators.

```
void main()
{
int x=10,y = 10, z ;
z= x--;
printf(“ z=%d x= %d\n”, z, x);
z = --y;
printf(“ z=%d y= %d”, z, y);
}
```
Output:
z=10 x=9
z=9 y=9

ASSIGNMENT OPERATOR

• The most common assignment operator is =.
• This operator assigns the value in right side to the left side.
• The syntax is shown below:
variable=expression;
• For ex:
c=5; //5 is assigned to c
b=c; //value of c is assigned to b
5=c; // Error! 5 is a constant.
• The operators such as +=,*= are called shorthand assignment operators.
• For ex,
a=a+10: can be written as a+=10;
• In the same way, we have:

Operator	**Example**	**Same as**
-=	a-=b	a=a-b
*=	a*=b	a=a*b
/=	a/=b	a=a/b
%=	a%=b	a=a%b

RELATIONAL OPERATORS

• Relational operators are used to find the relationship between two operands.
• The output of relational expression is either true(1) or false(0).
• For example
a>b //If a is greater than b, then a>b returns 1 else a>b returns 0.
• The 2 operands may be constants, variables or expressions.
• There are 6 relational operators:

Operator Meaning of Operator Example

> Greater than 5>3 returns true (1)
< Less than 5<3 returns false (0)
>= Greater than or equal to 5>=3 returns true (1)
<= Less than or equal to 5<=3 return false (0)
== Equal to 5==3 returns false (0)
!= Not equal to 5!=3 returns true(1)
• For ex:

Condition Return values

2>1 1 (or true)
2>3 0 (or false)
3+2<6 1 (or true)
• Example: Program to illustrate the use of all relational operators.

```
#include<stdio.h>
void main()
{
printf("4>5 : %d \n", 4>5);
printf("4>=5 : %d \n", 4>=5);
printf("4<5 : %d \n", 4<5);
printf("4<=5 : %d \n", 4<=5);
printf("4==5 : %d \n", 4==5);
printf("4!=5 : %d ", 4!=5);
}
```

Output:

4>5 : 0
4>=5 : 0
4<5 : 1
4<=5 :1
4==5 : 0
4!=5 : 1

LOGICAL OPERATORS

• These operators are used to perform logical operations like negation, conjunction and disjunction.
• The output of logical expression is either true(1) or false(0).
• There are 3 logical operators:

Operator Meaning Example

&& Logical AND If c=5 and d=2 then ((c==5) && (d>5)) returns false.
|| Logical OR If c=5 and d=2 then ((c==5) || (d>5)) returns true.
! Logical NOT If c=5 then !(c==5) returns false.
• All non zero values(i.e. 1, -1, 2, -2) will be treated as true.
While zero value(i.e. 0) will be treated as false.

Truth table

A B A&&B A||B !A
0 0 0 0 1
0 1 0 1
1 0 0 1 0
1 1 1 1

• Example: Program to illustrate the use of all logical operators.

LOGICAL OPERATORS

• These operators are used to perform logical operations like negation, conjunction and disjunction.
• The output of logical expression is either true(1) or false(0).
• There are 3 logical operators:

Operator Meaning Example

&& Logical AND If c=5 and d=2 then ((c==5) && (d>5)) returns false.
|| Logical OR If c=5 and d=2 then ((c==5) || (d>5)) returns true.
! Logical NOT If c=5 then !(c==5) returns false.
• All non zero values(i.e. 1, -1, 2, -2) will be treated as true.
While zero value(i.e. 0) will be treated as false.

Truth table

A B A&&B A||B !A
0 0 0 0 1
0 1 0 1
1 0 0 1 0
1 1 1 1
• Example: Program to illustrate the use of all logical operators.

2.b Explain with example for passing array to a function and string to a function

Passing the Whole Array

• Here, the parameter passing is similar to pass by reference.
• In pass by reference, the formal parameters are treated as aliases for actual parameters.
• So, any changes in formal parameter imply there is a change in actual parameter.
• Example: Program to find average of 6 marks using pass by reference.

```
#include <stdio.h>
void average(int m[])
{
int i ,avg;
int avg, sum=0;
for(i=0;i<6;i++)
{
sum= sum+ m[i];
}
avg =(sum/6);
printf("aggregate marks= %d ", avg);
}
void main()
{
int m[6]={60, 50, 70, 80, 40, 80, 70};
average(m); // Only name of array is passed as argument
}
```

Output:

aggregate marks= 70

Example for passing string to a function

```
#include <stdio.h>
void displayString(char str[]);

int main()
{
   char str[50];
   printf("Enter string: ");
```

```
    gets(str);
    displayString(str);     // Passing string c to function.
    return 0;
}
void displayString(char str[]){
    printf("String Output: ");
    puts(str);
}
```

Module – 2

3.a) What is pointer? Give an account of function returning a pointer with program example

POINTER

• A pointer is a variable which holds address of another variable or a memory-location.
• For ex:
c=300;
pc=&c;
Here pc is a pointer; it can hold the address of variable c & is called reference operator

DECLARATION OF POINTER VARIABLE

• Dereference operator(*) are used for defining pointer-variable.
• The syntax is shown below:
data_type *ptr_var_name;

• For ex:

int *a; // a as pointer variable of type int
float *c; // c as pointer variable of type float

• Steps to access data through pointers:
1) Declare a data-variable ex: int c;
2) Declare a pointer-variable ex: int *pc;
3) Initialize a pointer-variable ex: pc=&c;
4) Access data using pointer-variable ex: printf("%d",*pc);

• Example: Program to illustrate working of pointers.

```
#include<stdio.h>
void main()
{
int *pc;
int c;
c=22;
printf("Address of c: %d \n", &c);
printf("Value of c: %d \n", c);
pc=&c;
printf("Address of pointer pc: %d \n", pc);
printf("Content of pointer pc: %d",*pc);
}
```

Output:

Address of c: 1232
Value of c: 22

Address of pointer pc: 1232
Content of pointer pc: 22

C also allows to return a pointer from a function. To do so, you would have to declare a function returning a pointer as in the following example −

```
int * myFunction() {
   .
   .
   .
}
```

Second point to remember is that, it is not a good idea to return the address of a local variable outside the function, so you would have to define the local variable as **static** variable.

Now, consider the following function which will generate 10 random numbers and return them using an array name which represents a pointer, i.e., address of first array element.

```
#include <stdio.h>
#include <time.h>

/* function to generate and retrun random numbers. */
int * getRandom( ) {

   static int  r[10];
   int i;

   /* set the seed */
   srand( (unsigned)time( NULL ) );

   for ( i = 0; i < 10; ++i) {
      r[i] = rand();
      printf("%d\n", r[i] );
   }

   return r;
}

/* main function to call above defined function */
int main () {

   /* a pointer to an int */
   int *p;
   int i;

   p = getRandom();

   for ( i = 0; i < 10; i++ ) {
      printf("*(p + [%d]) : %d\n", i, *(p + i) );
   }

   return 0;
}
```

3.b. Define structure. Explain how the structure variable passed as a parameter to a function with example

STRUCTURES

• Structure is a collection of elements of different data type.

• The syntax is shown below:

```
struct structure_name
{
data_type member1;
data_type member2;
data_type member3;
};
```

• The variables that are used to store the data are called members of the structure.

• We can create the structure for a person as shown below:

```
struct person
{
char name[50];
int cit_no;
float salary;
};
```

• This declaration above creates the derived data type struct person.

STRUCTURES AND FUNCTIONS

• In C, structure can be passed to functions by 2 methods:

1) Passing by value (passing actual value as argument)

2) Passing by reference (passing address of an argument)

Passing Structure by Value

• A structure-variable can be passed to the function as an argument as normal variable.

• If structure is passed by value, change made in structure-variable in function-definition does not reflect in original structure variable in calling-function.

• Example: Write a C program to create a structure student, containing name and roll. Ask user the name and roll of a student in main function. Pass this structure to a function and display the information in that function.

```
#include<stdio.h>
struct student
{
char name[50];
int roll;
};
void Display(struct student stu)
{
printf("Your Name: %s \n",stu.name);
printf("Your Roll: %d",stu.roll);
}
void main()
{
struct student s1;
printf("Enter student's name: ");
scanf("%s", &s1.name);
printf("Enter roll number: ");
scanf("%d", &s1.roll);
Display(s1); //passing structure variable s1 as argument
}
```

Output:

```
Enter student's name: rama
Enter roll number: 149
```

Your Name: rama
Your Roll: 149

Passing Structure by Reference

• The address location of structure-variable is passed to function while passing it by reference.
• If structure is passed by reference, change made in structure-variable in function-definition reflects in original structure variable in the calling-function.
• Write a C program to add two complex numbers entered by user. To solve this program, make a structure. Pass two structure variable (containing distance in real and imaginary part) to add function by reference and display the result in main function without returning it.

```
#include <stdio.h>
struct comp
{
int real;
float img;
};
void Add(struct comp c1,struct comp c2, struct comp *c3)
{
// Adding complex numbers c1 and d2 and storing it in c3
c3->real=c1.real+c2.real;
c3->img=c1.img+c2.img;
}
void main()
{
struct comp c1, c2, c3;
printf("First complex number \n");
printf("Enter real part: ");
scanf("%d",&c1.real);
printf("Enter imaginary part: ");
scanf("%f",&c1.img);
printf("Second complex number \n");
printf("Enter real part: ");
scanf("%d",&c2.real);
printf("Enter imaginary part: ");
scanf("%f",&c2.img);
Add(c1, c2, &c3);
//passing structure variables c1 and c2 by value whereas passing
// structure variable c3 by reference
printf("Sum of 2 complex numbers = %d + i %d",c3.real, c3.img);
}
```

Output:

```
First complex number
Enter real part: 3
Enter imaginary part: 4
Second complex number
Enter real part: 2
Enter imaginary part: 5
Sum of 2 complex numbers = 5 + i 9
```

4.a) What are arrays? Explain the concept of inserting an element into an array. Write a program to implement the same

ARRAY

• Array is a collection of elements of same data type.
• The elements are stored sequentially one after the other in memory.
• Any element can be accessed by using
→ name of the array
→ position of element in the array
• Arrays are of 2 types:
1) Single dimensional array
2) Multi dimensional array

Inserting element in an array

Given an array of length N, we have to **insert an element in array** at index i (0 <= i <= N-1). After inserting an element, the number of elements in array will increase by one.

All elements of array are stored in consecutive memory location. To insert an element at index i in array we have to shift all elements from index i to N-1 to next index. An element at index k, will be moved to index k+1.

```
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()
```

4.b) What are strings? Mention any five string operation. Write a program to concatenate 2 string without using built in functions

STRINGS

• Array of character are called strings.
• A string is terminated by null character /0.
• For ex:
"c string tutorial"
• The above string can be pictorially represented as shown below:

STRING VARIABLE

• There is no separate data type for strings in C.
• They are treated as arrays of type „char".
• So, a variable which is used to store an array of characters is called a string variable.

Declaration of String

• Strings are declared in C in similar manner as arrays.
Only difference is that, strings are of „char" type.
• For ex:
char s[5]; //string variable name can hold maximum of 5 chars including NULL character
• The above code can be pictorially represented as shown below:

Initialization of String

• For ex:

```
char s[5]={'r', 'a', 'm', 'a' };
char s[9]="rama";
```

S.N.	Function & Purpose
1	**strcpy(s1, s2);** Copies string s2 into string s1.
2	**strcat(s1, s2);** Concatenates string s2 onto the end of string s1.
3	**strlen(s1);** Returns the length of string s1.
4	**strcmp(s1, s2);** Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
5	**strchr(s1, ch);** Returns a pointer to the first occurrence of character ch in string s1.
6	**strstr(s1, s2);** Returns a pointer to the first occurrence of string s2 in string s1.

strcat()

• This function joins 2 strings. It takes two arguments, i.e., 2 strings and resultant string is stored in the first string specified in the argument.

• The syntax is shown below:

strcat(first_string,second_string);

• Example: Program to illustrate the use of strcat().

```
#include <stdio.h>
#include <string.h>
void main()
{
char str1[10], str2[10};
printf("Enter First String:");
gets(str1);
printf("\n Enter Second String:");
gets(str2);
strcat(str1,str2); //concatenates str1 and str2 and
printf("\n Concatenated String is ");
puts(str1); //resultant string is stored in str1
}
```

Output:

Enter First String: rama
Enter Second String: krishna
Concatenated String is Ramakrishna

Module – 3

5.a) What are data structures? Explain the classifications of data structures

Data structure

means how the data is organized in memory. There different kind of data structures. Some are used to store the data of same type and some for different types of data. Different types of data structures help different types of *operations of data structures* too. Data structure is a particular way of storing and organizing data in a computer so that it can be used efficiently.

Primitive and Non Primitive Data Structure: The data structure that are atomic (indivisible) are called primitive. Examples are integer, real and characters.

The Data structures that are not atomic are called non-primitive or composite. Examples are records, array and string.

---

1) Linear data structure
2) Non-linear data structure

Linear data structure :

Collection of nodes which are logically adjacent in which logical adjacency is maintained by pointers

(or)

Linear data structures can be constructed as a continuous arrangement of data elements in the memory. It can be constructed by using array data type. In the linear Data Structures the relation ship of adjacency is maintained between the Data elements.

***Operations* applied on linear data structure :**

The following list of operations applied on linear data structures
1. Add an element
2. Delete an element
3. Traverse
4. Sort the list of elements
5. Search for a data element

By applying one or more functionalities to create different types of data structures
For example *Stack, Queue, Tables, List, and Linked Lists.*

Non-linear data structure:

Non-linear data structure can be constructed as a collection of randomly distributed set of data item joined together by using a special pointer (tag). In non-linear Data structure the relationship of adjacency is not maintained between the Data items.

***Operations* applied on non-linear data structures :**
The following list of operations applied on non-linear data structures.
1. Add elements
2. Delete elements
3. Display the elements
4. Sort the list of elements
5. Search for a data element

By applying one or more functionalities and different ways of joining randomly distributed data items to create different types of data structures. *For example Tree, Decision tree, Graph and Forest*

5.b) Define Stack. Write a 'C' program to implement stack operations using arrays

Definition :

Stack is a LIFO(Last in First out data structure). It has only one end from which both insertion and deletion can be done. This is called the top.

Implementation of stack using array

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

#define size 5
struct stack {
  int s[size];
  int top;
} st;

int stfull() {
  if (st.top >= size - 1)
    return 1;
  else
    return 0;
}

void push(int item) {
  st.top++;
  st.s[st.top] = item;
}

int stempty() {
  if (st.top == -1)
    return 1;
  else
    return 0;
}

int pop() {
  int item;
  item = st.s[st.top];
  st.top--;
  return (item);
}

void display() {
  int i;
  if (stempty())
    printf("\nStack Is Empty!");
  else {
    for (i = st.top; i >= 0; i--)
      printf("\n%d", st.s[i]);
  }
}

int main() {
  int item, choice;
  char ans;
```

```
  st.top = -1;

  printf("\n\tImplementation Of Stack");
  do {
    printf("\nMain Menu");
    printf("\n1.Push \n2.Pop \n3.Display \n4.exit");
    printf("\nEnter Your Choice");
    scanf("%d", &choice);
    switch (choice) {
    case 1:
      printf("\nEnter The item to be pushed");
      scanf("%d", &item);
      if (stfull())
        printf("\nStack is Full!");
      else
        push(item);
      break;
    case 2:
      if (stempty())
        printf("\nEmpty stack!Underflow !!");
      else {
        item = pop();
        printf("\nThe popped element is %d", item);
      }
      break;
    case 3:
      display();
      break;
    case 4:
      exit(0);
    }
    printf("\nDo You want To Continue?");
    ans = getche();
  } while (ans == 'Y' || ans == 'y');

return 0;
}
```

6.a. Convert following infix expression into postfix using applications of stack
A+(B*C-(D/E^F)*G)*H

Stack	Input	Output
Empty	A+(B*C-(D/E-F)*G)*H	-
Empty	+(B*C-(D/E-F)*G)*H	A
+	(B*C-(D/E-F)*G)*H	A
+(	B*C-(D/E-F)*G)*H	A
+(	*C-(D/E-F)*G)*H	AB
+(*	C-(D/E-F)*G)*H	AB
+(*	-(D/E-F)*G)*H	ABC
+(-	(D/E-F)*G)*H	ABC*
+(-(	D/E-F)*G)*H	ABC*
+(-(	/E-F)*G)*H	ABC*D
+(-(/	E-F)*G)*H	ABC*D
+(-(/	-F)*G)*H	ABC*DE
+(-(-	F)*G)*H	ABC*DE/
+(-(-	F)*G)*H	ABC*DE/
+(-(-	)*G)*H	ABC*DE/F
+(-	*G)*H	ABC*DE/F-
+(-*	G)*H	ABC*DE/F-
+(-*	)*H	ABC*DE/F-G
+	*H	ABC*DE/F-G*-
+*	H	ABC*DE/F-G*-
+*	End	ABC*DE/F-G*-H
Empty	End	ABC*DE/F-G*-H*+

6.b) What is recursion? Write a program to implement towers of Hanoi problem using recursion and trace the output for 3 disks

Definition :

- A function is recursive if a statement in the body of the function calls itself.
- Recursion is a name given for expressing anything in terms of itself.
- Recursion is the process of defining something in terms of itself.

Recursion Program for Tower of Hanoi

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

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

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

Algorithm

- To move n disks from tower A to tower C, using B as auxiliary.
- If n==1, move the single disk from A to C and stop.
- Move the top(n-1) disks from A to B, using C as auxiliary.
- Move the remaining disk from A to C.
- Move the (n-1) disks from B to C, using A as auxiliary.
- Given ‘n’ disks the tower of Hanoi problem takes $2^n - 1$ steps.

```
/* Program to solve Tower of Hanoi problem */
#include<stdio.h>
int count=0;
tower(int n,char src,char temp,char dest)
{
        if(n==1)
        {
                printf("Move disc %d from %c to %c\n",n,src,dest);
                return;
        }
        tower(n-1,src,dest,temp);
        printf("Move disc %d from %c to %c\n",n,src,dest);
        tower(n-1,temp,src,dest);
}

int main()
{
```

```
	int n;
	printf("enter the number of disks : \n");
	scanf("%d",&n);
	tower(n,'a','b','c');
}
```

```
Enter the number of disks : 3

Move disk 1 from  A to  C
Move disk 2 from  A to  B
Move disk 1 from  C to  B
Move disk 3 from  A to  C
Move disk 1 from  B to  A
Move disk 2 from  B to  C
Move disk 1 from  A to  C
```

7a.) Singly linked list

```
#include<stdio.h>
int MAX=4, count;

struct student
{
	char usn[10];
	char name[30];
	char branch[5];
	int sem;
	char phno[10];
	struct student *next; //Self referential pointer.
};
typedef struct student NODE;

int countnodes(NODE *head)
{
	NODE *p;
	count=0;
	p=head;
	while(p!=NULL)
	{
		p=p->next;
		count++;
	}
	return count;
}

NODE* getnode(NODE *head)
{
	NODE *newnode;
	newnode=(NODE*)malloc(sizeof(NODE)); //Create first NODE
	printf("\nEnter USN, Name, Branch, Sem, Ph.No\n");
	gets(newnode->usn);
```

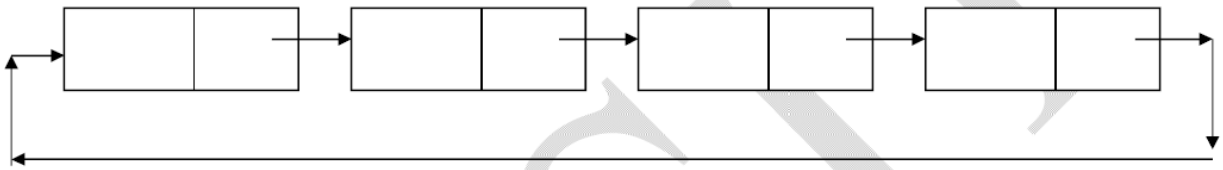
```
        gets(newnode->name);
        gets(newnode->branch);
        scanf("%d",&(newnode->sem));
        gets(newnode->phno);
        newnode->next=NULL; //Set next to NULL...
        head=newnode;
        return head;
}

NODE* display(NODE *head)
{
        NODE *p;
        if(head == NULL)
        printf("\nNo student data\n");
        else
        {
                p=head;
                printf("\n----STUDENT DATA----\n");
                printf("\nUSN\tNAME\t\tBRANCH\tSEM\tPh.NO.");
                while(p!=NULL)
                {
                        printf("\n%s\t%s\t\t%s\t%d\t%s", p->usn, p->name, p->branch, p->sem, p-
                        >phno);
                        p = p->next; //Go to next node...
                }
                printf("\nThe no. of nodes in list is: %d",countnodes(head));
        }
        return head;
}

NODE* create(NODE *head)
{
        NODE *newnode;
        if(head==NULL)
        {
                newnode=getnode(head);
                head=newnode;
        }
        else
        {
                newnode=getnode(head);
                newnode->next=head;
                head=newnode;
        }
        return head;
}

NODE* insert_front(NODE *head)
```

```
{
	if(countnodes(head)==MAX)
		printf("\nList is Full / Overflow!!");
	else
		head=create(head); //create()insert nodes at front.
	return head;
}

NODE* delete_front(NODE *head)
{
	NODE *p;
	if(head==NULL)
		printf("\nList is Empty/Underflow (STACK/QUEUE)");
	else
	{
		p=head;
		head=head->next; //Set 2nd NODE as head
		free(p);
		printf("\nFront(first)node is deleted");
	}
	return head;
}


void main()
{
int ch, i, n;
NODE *head;
head=NULL;
clrscr();
printf("\n*----------Studednt Database-----------*");
do
{
	printf("\n 1.Create\t 2.Display\t 3.Insert at begin \t 4.Delete at the begin\t 5.Exit");
	printf("\nEnter your choice: ");
	scanf("%d", &ch);
	switch(ch)
	{
		case 1: printf("\nHow many student data you want to create: ");
			scanf("%d", &n);
			for(i=0;i<n;i++)
				head=create(head);//Call to Create NODE...
			break;
		case 2: head=display(head); //Call to Display...
			break;
		case 3: head=insert_front(head); //Call to Insert...
			break;
		case 4: head=delete_front(head); //Call to delete
```

```
                    break;
            case 5: exit();
}
}while(ch!=5);
}
```

7. b. what are circular linked list? Write c function to perform insertion and deletion operations at the end of the circular lists

Circular singly linked list is quite similar to singly linked list. The only difference is – the 'link' field of a last node in a circular singly linked list contains the address of first node, instead of NULL. The diagrammatic representations is –



In In a singly linked list, we can trace the list in one direction. That is, if we are at 10th node, we can't trace back to access 9th node, instead, we have trace from the beginning once again. This is time consuming. Hence, we make use of circular list to avoid this problem up to some extent.

Because, in a circular list, from any node we can trace rest of the nodes processing in a forward direction (refer diagram given in previous slide). Theoretically, any node in a circular list can be treated as a first node, and its previous node as a last node. But, since, there won't be NULL in any node to indicate end of list, the circular lists have to be processed properly, otherwise, it may lead to infinite loop.

Program for Operations on Circular linked lists

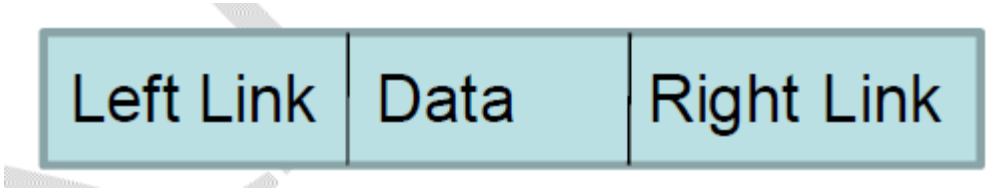
```
#include<stdio.h>
#include<alloc.h>
#include<conio.h>
#include<stdlib.h>
struct node
{
int data;
struct node *link;
};
typedef struct node *NODE;
NODE getnode()
{
NODE x;
x=(NODE) malloc(sizeof(struct node));
if(x==NULL)
{
printf("no memory in heap");
exit(0);
}
```

```
return x;
}


NODE insert_rear(int item, NODE last)
{
NODE temp;
temp=getnode();
temp->data=item;
temp->link=temp;
if (last==NULL)
return temp;
temp->link=last->link;
last->link=temp;
return temp;
}
void display(NODE last)
{
NODE temp;
if(last==NULL)
{
printf("No element to display\n");
return ;
}
temp=last->link;
printf("The contents of list:\n");
while(temp!=last)
{
printf(“%d\n” temp->data);
temp=temp->link;
}
printf(“%d”, temp->data);
}

NODE delete_rear(NODE last)
{
NODE prev;
if(last==NULL)
{
printf("no element to delete\n");
return NULL;
}
if(last->link==last)
{
printf(“The item deleted is %d”, last->data);
free(last);
return NULL;
}
```

```
prev=last->link;
while(prev->link!=last)
prev=prev->link;
prev->link=last->link;
printf("\nDeleted element is %d", last->data);
free(last);
return prev;
}
void main()
{
int opt, item;
NODE last=NULL;
for(;;)
{
printf("1.Insert Rear\n 2.Display\n")
printf(" 3.Delete Rear\n4.Exit\n");
printf("Enter your option:");
scanf("%d",&opt);

switch(opt)
{
case 1: printf("\nenter item");
scanf("%d",&item);
last=insert_rear(item,last);
break;
case 2: display(last);
break;
case34: last=delete_rear(last);
break;
default: exit(0);
}
}
}
```

8.a) What are Doubly linked list. Program to implement stack operation using DLL

In doubly linked list each node consists of two link fields (viz. left link and right link) and one data field as shown:



Here, right link is used to store the address of next node and left link is used to store the address of previous node. In ordinary doubly linked list, the left link field of first node and the right link field of last node contain NULL.

8.b) What are priority queues? Write a program to simulate the working of priority queues

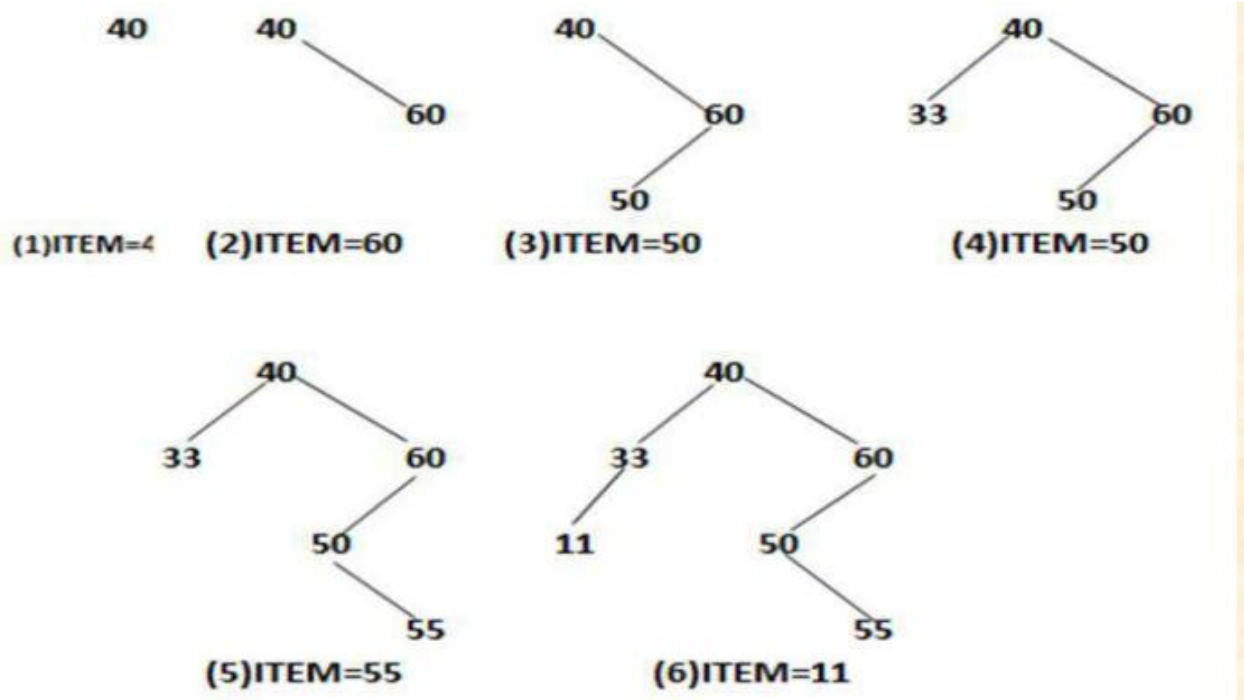
Its a special type of data structure in which items can be inserted or deleted based on the priority.
Types of priority Queues:
Ascending priority queue
Descending priority queue
The rules are:
An element of higher priority is processed before any element of lower priority.
Two elements with the same priority are processed according to the order in which they were added to the queue.

```
NODE insert-order(int item, NODE start)
{
NODE temp, prev, cur;
temp=getnode();
temp->data = item;
temp->link = NULL;
if(start == NULL)
return temp;
if(item < start ->data)
{
temp->link =start;
return temp;
}
prev = NULL;
cur = start;
while(cur != NULL && item >= cur->data)
{
prev = cur;
cur = cur->Link;
}
prev->link = temp;
temp->link=cur;
return start;
}
NODE delete_front(NODE start)
{ 5
```

```
NODE temp;
if(start==NULL)
{
printf("no element to delete\n");
return start;
}
temp=start;
printf("Deleted item=%d", temp->data);
start=start->link;
free(temp);
return start;
}
```

Module – 5

9.a) Construct the BST for the items 40,60,50, 33, 55,11. Write and explain c module to insert an element in to BST




```
//Function to create BST by inserting elements
NODE insert (int item, NODE root) //Function to create BST by inserting elements
{
NODE temp,prev,cur;
temp = getnode();
temp->data = item;
temp->rlink = NULL;
temp->llink = NULL;
if(root == NULL)
return temp;
prev = NULL;
cur = root;
```

```
while (cur != NULL)
{
prev = cur;
cur = (item < cur->data) ? cur->llink : cur->rlink;
}
if (item < prev->data)
prev->llink = temp;
else
prev->rlink = temp;
return root;
}
```

9.b) Write a C program to implement select selection sort method and consider the elements to trace the output 7,3,4,1,8,2,6,5

Proses	Swap	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
	Data awal	7	3	4	1	8	2	6	5
1	m=A[0], k = 1	7	3	4	1	8	2	6	5
2	m=A[1], k = 2	1	3	4	7	8	2	6	5
3	M=A[2], k = 3	1	2	4	7	8	3	6	5
4	m=A[3], k = 4	1	2	3	7	8	4	6	5
5	m=A[4], k = 5	1	2	3	4	8	7	6	5
6	m=A[5], k = 6	1	2	3	4	5	7	6	8
7	m=A[6], k = 7	1	2	3	4	5	6	7	8

Program for selection sort:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a[10],n,i,temp,j;
clrscr();
```

```
printf("Enter the size of the array:");
scanf("%d",&n);
printf("\nEnter array elements:\n");
for(i=0;i<n;i++)
scanf("%d",&a[i]);
for(i=0;i<n-1;i++)
{
for(j=i+1;j<n;j++)
{
if(a[ i]>a[j])
{
temp=a[i];
a[i]=a[j];
a[j]=temp;
}
}
}
printf("\nSorted list is:\n");
for(i=0;i<n;i++)
printf("%d\t",a[i]);
getch();
}
```

10.a)

Traversal of a tree is a method of visiting each node of a tree exactly once in a systematic way. There are three different methods for tree traversal, viz.

i) Pre-order traversal

ii) In-order traversal

iii) Post-order traversal

The rules for these traversals are as below-

Pre-order traversal:

i) Visit the root.

ii) Traverse left subtree using pre-order traversal method.

iii) Traverse right subtree using pre-order traversal method.

In-order traversal:

i) Traverse left subtree using in-order traversal method.

ii) Visit the root.

iii) Traverse right subtree using in-order traversal method.

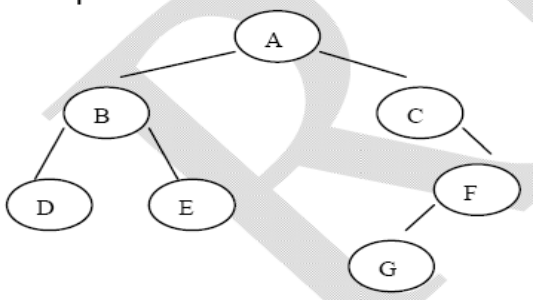
Post-order traversal:

i) Traverse left subtree using post-order traversal method.

ii) Traverse right subtree using pos-order traversal method.

iii) Visit the root.

Example:



Pre-order:
ABDECFG
In-order:
DBEACGF
Post-order:
DEBGFCA

```
#include <stdio.h>
#include <stdlib.h>
 struct btnode
{
    int value;
    struct btnode *l;
    struct btnode *r;
}*root = NULL, *temp = NULL, *t2, *t1;
void insert();
void inorder(struct btnode *t);
void create();
void search(struct btnode *t);
void preorder(struct btnode *t);
void postorder(struct btnode *t);
int flag = 1;

void main()
{
    int ch;

    printf("\nOPERATIONS ---");
    printf("\n1 - Insert an element into tree\n");
    printf("2 - Inorder Traversal\n");
    printf("3 - Preorder Traversal\n");
```

```
    printf("4 - Postorder Traversal\n");
    printf("5 - Exit\n");
    while(1)
    {
      printf("\nEnter your choice : ");
      scanf("%d", &ch);
      switch (ch)
      {
      case 1:insert();
          break;
      case 2:inorder(root);
          break;
      case 3:preorder(root);
          break;
      case 4:postorder(root);
          break;
      default:exit(0);

      }
    }
}
```

10 b) Hashing function

Every record is usually identified by some ***key***.
Here we will consider the implementation of a dictionary of *n* records with keys k1, k2 …kn. Hashing is based on the idea of distributing keys among a one-dimensional array H[0…m-1], called ***hash table***.

For each key, a value is computed using a predefined function called ***hash function***. This function assigns an integer, called ***hash address***, between 0 to m-1 to each key. Based on the hash address, the keys will be distributed in a hash table.
For example, if the keys k1, k2, …., kn are integers, then a hash function can be
h(K) = K mod m.

Let us take keys as 65, 78, 22, 30, 47, 89. And let hash function be, h(k) = k%10.
Then the hash addresses may be any value from 0 to 9. For each key, hash address will be computed as –

h(65) = 65 %10 = 5

h(78) = 78%10 = 8

h(22)= 22 % 10 =2

h(30)= 30 %10 =0

h(47) = 47 %10 = 7

h(89)=89 % 10 = 9

Now, each of these keys can be hashed into a hash table as –

0	1	2	3	4	5	6	7	8	9
30		22			65		47	78	89

In general, a hash function should satisfy the following requirements:

A hash function needs to distribute keys among the cells of hash table as evenly as possible.

Binary Search

This method is applied on the sorted array. Initially, the key element is compared with the middle element of the array. If they are equal then the search is successful. Otherwise, the array is divided into two parts viz. from the first element to middle element and from middle to last element. If the key element is greater than the middle element then the second sub-array is searched for. If the key is less than the middle then the first sub-array is searched. The procedure is repeated till the key is found or till the sub-array contains a non-matching single element.

	low	high	mid
#1	0	8	4
#2	5	8	6
#3	5	5	5

search(44)

$$mid = \left\lfloor \frac{low + high}{2} \right\rfloor$$