DATA STRUCTURES USING C

DAYANANDA V N
Lecturer in Computer Science and Engg.
Acharya polytechnic, Bangalore

UNIT 1

POINTERS AND DYNAMIC MEMORY ALLOCATION

SYLLABUS

Pointers - Concept of pointers, Declaring and initializing pointers, Accessing variables using pointers, Pointer arithmetic, Pointers and arrays, Pointers and character strings, Pointers and functions, Pointer as a function argument, Pointers to function, Pointers and structures.

Dynamic Memory allocation – Introduction, Dynamic memory allocation, Allocating a block of memory : Malloc, Allocating multiple blocks of memory : Calloc, Releasing the used space : Free Altering the size of memory : Realloc

SYNOPSIS

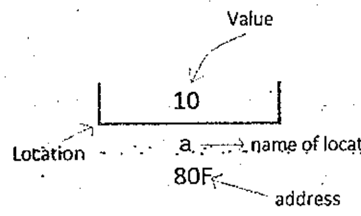
A brief introduction:

Whenever a **variable** is declared, system will allocate a location to that variable in the memory, to hold value. This location will have its own address number.

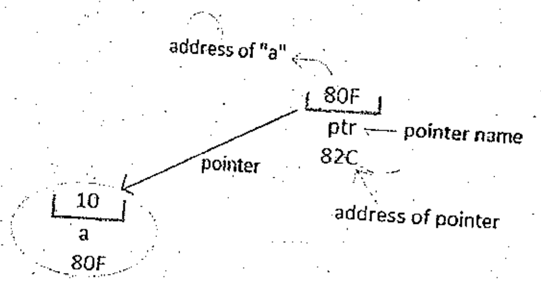
Let us assume that system has allocated memory location 80F for a variable **a**.

int a = 10 ;

A **pointer** is a variable whose value is the address of another variable



We can access the value 10 by either using the variable name **a** or the address 80F. Since the memory addresses are simply **numbers** they can be assigned to some other variable. The variable that holds memory address are called **pointer variables**. A **pointer** variable is therefore nothing but a variable that contains an address, which is a location of another variable. Value of **pointer variable** will be stored in another memory location.



How to Use Pointers?

There are a few important operations, which we will do with the help of pointers very frequently. **(a)** We define a pointer variable, **(b)** assign the address of a variable to a pointer and **(c)** finally access the value at the address available in the pointer variable. This is done by using unary operator □ that returns the value of the variable located at the address specified by its operand. The following example makes use of these operations –

```
#include < stdio.h>
int main () {
  int  var = 20;  /* actual variable declaration */
  int  *ip;       /* pointer variable declaration */
  ip = &var;  /* store address of var in pointer variable */
}
```

5 Marks QUESTIONS

1. Define pointer. Write its advantages & disadvantages.

Ans. A pointer is a variable whose value is the address of another variable.

Benefits(use) of pointers in c:

Pointers provide direct access to memory

- Pointers provide a way to return more than one value to the functions
- Reduces the storage space and complexity of the program
- Reduces the execution time of the program
- Provides an alternate way to access array elements
- Pointers can be used to pass information back and forth between the calling function and called function.
- Pointers allows us to perform dynamic memory allocation and deallocation.
- Pointers helps us to build complex data structures like linked list, stack, queues, trees, graphs etc.

- Pointers allows us to resize the dynamically allocated memory block.
- Addresses of objects can be extracted using pointers

Drawbacks of pointers in c:

- Uninitialized pointers might cause segmentation fault.
- Dynamically allocated block needs to be freed explicitly. Otherwise, it would lead to memor leak.
- Pointers are slower than normal variables.
- If pointers are updated with incorrect values, it might lead to memory corruption.

Basically, pointer bugs are difficult to debug. Its programmers' responsibility to use pointer effectively and correctly.

2. Explain the declaration & initialization of pointer variable with an example.

Ans. Like any variable or constant, you must declare a pointer before using it to store any variabl address. The general form of a pointer variable declaration is –

type *var-name;

Here, **type** is the pointers base type; it must be a valid C data type and **var-name** is the nam of the pointer variable. The asterisk * used to declare a pointer is the same asterisk used fo multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Take a look at some of the valid pointer declarations –

```
int    *ip;   /* pointer to an integer */
double *dp;   /* pointer to a double */
float  *fp;   /* pointer to a float */
char   *ch    /* pointer to a character */
```

Initialization of pointers:

(a) define a pointer variable, **(b)** assign the address of a variable to a pointer and **(c)** finally access the value at the address available in the pointer variable. This is done by using unary-operator □ that returns the value of the variable located at the address specified by its operand. The following example makes use of these operations –

```
#include  < stdio.h>

int main () {
  int  var = 20;  /* actual variable declaration */
  int  *ip;       /* pointer variable declaration, * is called indirection operator */
  ip = &var;  /* store address of var in pointer variable. called initialization of pointers* , & is
called address operator/
```

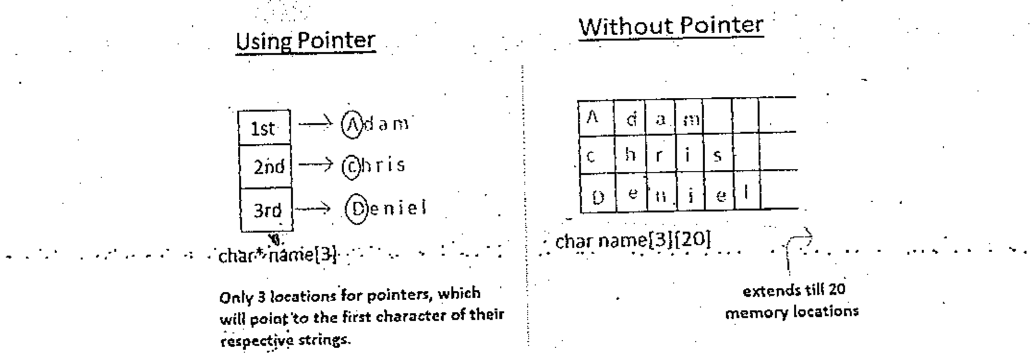
3. Discuss the use of address operator & indirection operator with pointers.

Ans. Refer Q2 (5 Marks)

4. Explain the array of pointers with an example.

Ans. Pointers are very helpful in handling character array with rows of varying length.

```
char *name[3]={
  "Adam",
  "chris",
  "Deniel"
        };
//Now see same array without using pointer
char name[3][20]= {
     "Adam",
     "chris",
     "Deniel"
         };
```



Using Pointer
Without Pointer
1st
2nd
3rd
Adam
Chris
Deniel
char *name[3]
Only 3 locations for pointers, which will point to the first character of their respective strings.
char name[3][20]
extends till 20 memory locations

In the second approach memory wastage is more, hence it is preferred to use pointer in such cases

5. Give the difference between call by value & call by reference.

Ans.

Difference between *call by value* and *call by reference*

call by value	call by reference
In *call by value*, a copy of actual arguments is passed to formal arguments of the called function and any change made to the formal arguments in the called function have no effect on the values of actual arguments in the calling function.	In *call by reference*, the location (address) of actual arguments is passed to formal arguments of the called function. This means by accessing the addresses of actual arguments we can alter them within from the called function.
In call by value, actual arguments will remain safe, they cannot be modified accidentally.	In *call by reference*, alteration to actual arguments is possible within from called function; therefore the code must handle arguments carefully else you get unexpected results.

6. Differentiate between pointers as function arguments & pointers to function.

Ans.

Pointer as function parameters	Pointer to function
A pointer can be used as a function arguments	A pointer contains address of a function
It gives the function access to the original argument	It points to the starting address of the function Like pointer to array

7. How is a pointer to an array different from an array of pointers? Explain with an example.

array of pointers

Ans. Refer Q 4 (5 Marks) & a pointer to an array refer Q8 (5Marks)

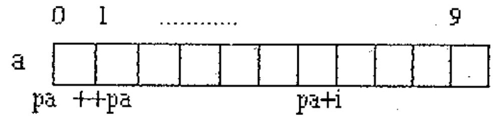
8. Explain pointers & array using example.

Pointers and arrays are very closely linked in C.

Hint: think of array elements arranged in consecutive memory locations.

Consider the following:

```
int a[10], x;
int *pa;
pa = &a[0];  /* pa pointer to address of a[0] */
 x = *pa;
/* x = contents of pa (a[0] in this case) */
```



To get somewhere in the array using a pointer we could do:

pa + i ≡ a[i]

WARNING: There is no bound checking of arrays and pointers so you can easily go beyond array memory and overwrite other things.

C however is much more subtle in its **link** between arrays and pointers.

For example we can just type

```
pa = a;
```

instead of

```
pa = &a[0]
```

and

a[i] can be written as *(a + i).

i.e. &a[i] ≡ a + i.

9. Explain how an array can be passed to a function.

Ans. Refer Q4 (10 Marks)

Passing entire array to function :

- Parameter Passing Scheme: **Pass by Reference**
- Pass name of array as function parameter .
- Name contains the base address i.e (Address of 0th element)
- Array values are updated in function .
- Values are reflected inside main function also.

10. Explain in brief the different parameter passing mechanisms.

Ans. Two Ways of Passing Argument to Function in C Language:

1. Call by Reference
2. Call by Value

Let us discuss different ways one by one–

A.Call by Value :

```
#include < stdio.h>
void interchange(int number1,int number2)
{
   int temp;
   temp = number1;
   number1 = number2;
   number2 = temp;
}
```

```
int main() {
   int num1=50,num2=70;
   interchange(num1,num2);
   printf("\nNumber 1: %d",num1);
   printf("\nNumber 2: %d",num2);
   return(0);
}
```

Output:

```
Number 1: 50
Number 2: 70
```

Explanation: Call by Value

1. While Passing Parameters using call by value , xerox copy of original parameter is created and passed to the called function.
2. Any update made inside method will not affect the original value of variable in calling function.
3. In the above example num1 and num2 are the original values and xerox copy of these values is passed to the function and these values are copied into number1,number2 variable of sum function respectively.
4. As their scope is limited to only function so they cannot alter the values inside main function.

B.Call by Reference/Pointer/Address :

```
#include < stdio.h>

void interchange(int *num1,int *num2)
{
   int temp;
   temp = *num1;
   *num1 = *num2;
   *num2 = temp;
}
```

```
int main() {

 int num1=50,num2=70;
 interchange(&num1,&num2);
 printf("\nNumber 1: %d",num1);
 printf("\nNumber 2: %d",num2);
 return(0);
}
```

Output :

```
Number 1: 70
Number 2: 50
```

Explanation: Call by Address

1. While passing parameter using call by address scheme , we are passing the actual address of the variable to the called function.
2. Any updates made inside the called function will modify the original copy since we are directly modifying the content of the exact memory location.

11. Explain pointers to structures with an example.

Ans. C structure can be accessed in 2 ways in a C program. They are,

1. Using normal structure variable
2. Using pointer variable

Dot(.) operator is used to access the data using normal structure variable and arrow (->) is used to access the data using pointer variable. You have learnt how to access structure data using normal variable in C – Structure topic. So, we are showing here how to access structure data using pointer variable in below C program.

EXAMPLE PROGRAM FOR C STRUCTURE USING POINTER:

In this program, "record1" is normal structure variable and "ptr" is pointer structure variable. As you know, Dot(.) operator is used to access the data using normal structure variable and arrow(->) is used to access data using pointer variable.

```
#include < stdio.h>
#include < string.h>
struct student
{
  int id;
```

```
  char name[30];
  float percentage;
};
int main()
{
  int i;
  struct student record1 = {1, "Raju", 90.5};
  struct student *ptr;
  ptr = &record1;
   printf("Records of STUDENT1: \n");
   printf("  Id is: %d \n", ptr->id);
   printf("  Name is: %s \n", ptr->name);
   printf("  Percentage is: %f \n\n", ptr->percentage);
  return 0;
}
```

OUTPUT:

```
Records of STUDENT1:
Id is: 1
Name is: Raju
Percentage is: 90.500000
```

12. Write C program to swap two numbers using pointers.

Ans. Refer Q10. (5 Marks)

13. Define dynamic programming in C & its advantages.

Ans. Dynamic programming (also known as **dynamic optimization**) is a method for solving a complex problem by breaking it down into a collection of simpler subproblems, solving each of those subproblems just once, and storing their solutions – ideally, using a memory-based data structure. The next time the same subproblem occurs, instead of recomputing its solution, one simply looks up the previously computed solution, thereby saving computation time at the expense of a (hopefully) modest expenditure in storage space.

14. Give the difference between malloc() & calloc() functions.

Ans.

Differences between malloc and calloc

malloc	calloc
The name malloc stands for *memory allocation.*	The name calloc stands for *contiguous allocation.*
void *malloc(size_t n) returns a pointer to n bytes of uninitialized storage, or NULL if the request cannot be satisfied. If the space assigned by malloc() is overrun, the results are undefined.	void *calloc(size_t n, size_t size)returns a pointer to enough free space for an array of n objects of the specified size, or NULL if the request cannot be satisfied. The storage is initialized to zero.
malloc() takes one argument that is, *number of bytes.*	calloc() take two arguments those are: *number of blocks* and *size of each block.*
syntax of malloc():	syntax of calloc():
void *malloc(size_t n);	void *calloc(size_t n, size_t size);
Allocates n bytes of memory. If the allocation succeeds, a void pointer to the allocated memory is returned. Otherwise NULL is returned.	Allocates a contiguous block of memory large enough to hold n elements of sizebytes each. The allocated region is initialized to zero.
malloc is faster than calloc.	calloc takes little longer than mallocbecause of the extra step of initializing the allocated memory by zero. However, in practice the difference in speed is very tiny and not recognizable.

15. Explain free(). What are its advantages.

Ans. Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on its own. You must explicitly use free() to release the space.

syntax of free()

free(ptr);

This statement frees the space allocated in the memory pointed by **ptr**. This avoids wastage of memory space.

10 MARKS QUESTIONS

1. Explain character pointer as an argument to a function with an example.

Ans. Refer Q4 (5 Marks)

2. Discuss pointer as a function argument. With an example explain call by reference method.

Ans. Refer Q10 (5 Marks)

3. With an illustration program explain pointers to structures.

Ans. Q11 (5 Marks)

4. With an illustration program explain pointers to arrays.

Ans. When an array is declared, compiler allocates sufficient amount of memory to contain a the elements of the array. Base address which gives location of the first element is also allocate by the compiler.

Suppose we declare an array **arr**,

int arr[5]={ 1, 2, 3, 4, 5 };

Assuming that the base address of **arr** is 1000 and each integer requires two byte, the five elemen will be stored as follows

element	arr[0]	arr[1]	arr[2]	arr[3]	arr[4]
Address	1000	1002	1004	1006	1008

Here variable **arr** will give the base address, which is a constant pointer pointing to the element **arr[0]**. Therefore **arr** is containing the address of **arr[0]** i.e 1000.

arr *is equal to* **&arr[0]** // by default

We can declare a pointer of type int to point to the array **arr**.

int *p;

p = arr;

or p = &arr[0]; //both the statements are equivalent.

Now we can access every element of array **arr** using **p ++** to move from one element to another

NOTE: You cannot decrement a pointer once incremented. p-- won't work.

Pointer to Array

As studied above, we can use a pointer to point to an Array, and then we can use that pointer t access **the array**. Lets have an example,

int i;

int a[5] = {1, 2, 3, 4, 5};

int *p = a; *// same as int*p = &a[0]*

for (i=0; i < 5; i + +)

```
{
printf("%d", *p);
p++;
}
```

In the above program, the pointer *p will print all the values stored in the array one by one. We can also use the Base address (a in above case) to act as pointer and print all the values.

Replacing the printf("%d", *p); statement of above example, with below mentioned statements. Lets see what will be the result.

printf("%d", a[i]); $\longrightarrow$ **prints the array, by incrementing index**

printf("%d", i[a]); $\longrightarrow$ **this will also print elements of array**

printf("%d", a+i); $\longrightarrow$ **This will print address of all the array elements**

printf("%d", *(a+i)); $\longrightarrow$ **Will print value of array element.**

printf("%d", *a); $\longrightarrow$ **will print value of a[0] only**

a++; $\longrightarrow$ **Compile time error, we cannot change base address of the array.**

5. Write a program to illustrate pointer arithmetic's.

Ans. Pointer Arithmetic in C Programming

We can perform **arithmetic operations on pointer** variable just as you can a numeric value. As we know that, a pointer in C is a variable which is used to store the memory address which is a numeric value. The **arithmetic operations on pointer** variable effects the memory address pointed by pointer.

Valid Pointer Arithmetic Operations

- Adding a number to pointer.
- Subtracting a number form a pointer.
- Incrementing a pointer.

- Decrementing a pointer.
- Subtracting two pointers.
- Comparison on two pointers.

Invalid Pointer Arithmetic Operations

- Addition of two pointers.
- Division of two pointers.
- Multiplication of two pointers.

(a) Incrementing a Pointer

Let ptr be an integer pointer which points to the memory location 5000 and size of an integer variable is 32-bit(4 bytes). Now, when we increment pointer ptr

ptr + + ;

it will point to memory location 5004 because it will jump to the next integer location which is 4 bytes next to the current location. **Incrementing a pointer** is not same as incrementing an integer value. On incrementing, a pointer will point to the memory location after skipping N bytes, where N is the size of the data type(in this case it is 4).

ptr + + is equivalent to ptr + (sizeof(pointer_data_type)).

"Incrementing a pointer increases its value by the number of bytes of its data type"

A character(1 bytes) pointer on increment jumps 1 bytes.

An integer(4 bytes) pointer on increment jumps 4 bytes.

This is a very important feature of **pointer arithmetic operations** which we will use in array traversal using pointers.

(b) Decrementing a Pointer

Similarly, Decrementing a pointer will decrease its value by the number of bytes of its data type. Hence, after

ptr--;

ptr will point to 4996.

ptr--; is equivalent to ptr - (sizeof(pointer_data_type)).

Adding Numbers to Pointers

Adding a number N to a pointer leads the pointer to a new location after skipping N times size of data type.

ptr + N = ptr + (N * sizeof(pointer_data_ype))

For example, Let ptr be a 4-byte integer pointer, initially pointing to location 5000.

Then ptr + 5 = 5000 + 4*5 = 5020. Pointer ptr will now point at memory address 5020.

Subtracting Numbers from **Pointers**

Subtracting a number N from a pointers is similar to adding a number except in Subtraction the new location will be before current location by N times size of data type.

ptr - N = ptr - (N * sizeof(pointer_data_ype))

For example, Let ptr be a 6-byte double pointer, initially pointing to location 5000.

Then ptr - 3 = 5000 - 6*3 = 4982. Pointer ptr will now point at memory address 4982.

(c) Subtracting Pointers

The difference between two pointer returns indicates "How apart the two Pointers are. It gives the total number of elements between two pointers.

For example, Let size of integer is 4 bytes. If an integer pointer 'ptr1' points at memory location 10000 and integer pointer 'ptr' points at memory location 10008, the result of ptr2 - ptr1 is 2.

C program to show pointer arithmetic

Program Output

```
Address of int_var = 2293300
Address of char_var = 2293299
Address of float_var = 2293292
After increment address in int_ptr = 2293304
After increment address in char_ptr = 2293300
After increment address in float_ptr = 2293296
After addition address in int_ptr = 2293312
After addition address in char_ptr = 2293302
After addition address in float_ptr = 2293304
```

6. Write C program to compare 2 strings to check whether they are equal or not using pointers.

Ans.

```
#include < stdio.h>
int compare_string(char*, char*);
main()
{
  char first[100], second[100], result;
  printf("Enter first string\n");
```

```
  gets(first);
   printf("Enter second string\n");
  gets(second);
  result = compare_string(first, second);
   if (result == 0 )
    printf("Both strings are same.\n");
  else
    printf("Entered strings are not equal.\n");
   return 0;
}
int compare_string(char *first, char *second)
{
  while(*first==*second)
  {
    if (*first == '\0' || *second == '\0' )
      break;
    first + + ;
    second + + ;
  }
  if(*first == '\0' && *second == '\0' )
    return 0;
  else
    return -1;
     }
```

7. What are the advantages of pointers? Write a program using pointers to compute the sum of all elements stored in an array.

Ans. Pointer has the following **advantages** :-

1. Using pointers we can allocate memory dynamically to structures (**Dynamic memory allocation**).

2. Arrays or strings can be passed to function more efficiently.

3. Better memory management or our program will run faster.

4. Pointer helps us to build complex data structures like linked lists, trees etc.

5. Provides alternate way to access array elements of any dimension.
6. Used to return more than one value from function.

```
#include < stdio.h>
#include < conio.h>
void main() {
 int numArray[10];
 int i, sum = 0;
 int *ptr;
 printf("\nEnter 10 elements: ");
 for (i = 0; i < 10; i + + )
 scanf("%d", &numArray[i]);
 ptr = numArray; /* a=&a[0] */
 for (i = 0; i < 10; i + + ) {
 sum = sum + *ptr;
 ptr + +;
 }
 printf("The sum of array elements: %d", sum);
}
```

Output:

```
Enter 10 elements: 11 12 13 14 15 16 17 18 19 20
The sum of array elements is 155
```

8. Explain Dynamic Memory Allocation in C using memory map.

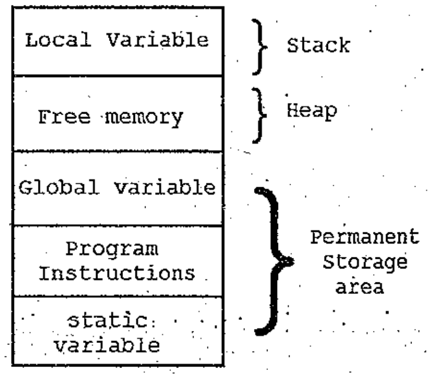
Ans: Dynamic Memory Allocation

The process of allocating memory at runtime is known as **dynamic memory allocation.** Library routines known as "memory management functions" are used for allocating and freeing memory during execution of a program. These functions are defined in **stdlib.h.**

Function	**Description**
malloc()	allocates requested size of bytes and returns a void pointer pointing to the first byte of the allocated space
calloc()	allocates space for an array of elements, initialize them to zero and then return a void pointer to the memory
free	releases previously allocated memory
realloc	modify the size of previously allocated space

Memory Allocation Process

Global variables, **static** variables and program instructions get their **memory in permanent** storage area whereas **local** variables are stored in area called Stack. The memory space between these two region is known as **Heap** area. This region is used for dynamic memory allocation during execution of the program. The size of heap keep changing.



9. List & explain dynamic memory allocation functions in C.

Allocating block of Memory

malloc() function is used for allocating block of memory at runtime. This function reserves a block of memory of given size and returns a pointer of type void. This means that we can assign it to any type of pointer using typecasting. If it fails to locate enough space it returns a NULL pointer.

Example using malloc() :

```
int *x;
x = (int*)malloc(50 * sizeof(int));    //memory space allocated to variable x
free(x);                //releases the memory allocated to variable x
```

calloc() is another memory allocation function that is used for allocating memory at runtime. **calloc** function is normally used for allocating memory to derived data types such as **arrays** and **structures.** If it fails to locate enough space it returns a NULL pointer.

Example using calloc() :

```
struct employee
{
 char *name;
 int salary;
};
typedef struct employee emp;
emp *e1;
e1 = (emp*)calloc(30,sizeof(emp));
```

realloc() changes memory size that is already allocated to a variable.

Example using realloc() :

```
int *x;
x=(int*)malloc(50 * sizeof(int));
x=(int*)realloc(x,100); //allocated a new memory to variable x
```

10. Write a program to illustrate memory allocation using malloc () function.

Write a C program to find sum of n elements entered by user. To perform this program, allocate memory dynamically using malloc() function.

```
#include < stdio.h>
#include < stdlib.h>
int main()
{
   int num, i, *ptr, sum = 0;
   printf("Enter number of elements: ");
   scanf("%d", &num);
   ptr = (int*) malloc(num * sizeof(int)); //memory allocated using malloc
   if(ptr == NULL)
   {
     printf("Error! memory not allocated.");
     exit(0);
   }
   printf("Enter elements of array: ");
   for(i = 0; i < num; + + i)
```

```
   {
     scanf("%d", ptr + i);
     sum += *(ptr + i);
   }
   printf("Sum = %d", sum);
   free(ptr);
   return 0;
}
```

11. Write a program to illustrate memory allocation using calloc () function.

Write a C program to find sum of n elements entered by user. To perform this program, allocate memory dynamically using calloc() function.

Ans.

```
#include < stdio.h>
#include < stdlib.h>
int main()
{
   int num, i, *ptr, sum = 0;
   printf("Enter number of elements: ");
   scanf("%d", &num);
   ptr = (int*) calloc(num, sizeof(int));
   if(ptr == NULL)
   {
     printf("Error! memory not allocated.");
     exit(0);
   }
   printf("Enter elements of array: ");
   for(i = 0; i < num; + + i)
   {
     scanf("%d", ptr + i);
     sum += *(ptr + i);
```

```
    }
    printf("Sum = %d", sum);
    free(ptr);
    return 0;
```

12. Write a program to illustrate memory reallocation using realloc () function.

Ans. If the previously allocated memory is insufficient or more than required, you can change the previously allocated memory size using realloc().

Syntax of realloc()

ptr = realloc(ptr, newsize);

Here, ***ptr*** is reallocated with size of newsize.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int *ptr, i , n1, n2;
    printf("Enter size of array: ");
    scanf("%d", &n1);
    ptr = (int*) malloc(n1 * sizeof(int));
    printf("Address of previously allocated memory: ");
    for(i = 0; i < n1; + + i)
        printf("%u\t",ptr + i);
    printf("\nEnter new size of array: ");
    scanf("%d", &n2);
    ptr = realloc(ptr, n2);
    for(i = 0; i < n2; + + i)
        printf("%u\t", ptr + i);
    return 0;
}
```

13. How are static & dynamic memory allocations different? Write a program to sort 10 numbers using malloc ().

Ans.

STATIC MEMORY ALLOCATION	DYNAMIC MEMORY ALLOCATION
Memory is allocated before the execution of the program begins. (During Compilation)	Memory is allocated during the execution of the program.
No memory allocation or deallocation actions are performed during Execution.	Memory Bindings are established and destroyed during the Execution.
Variables remain permanently allocated.	Allocated only when program unit is active.
Implemented using stacks and heaps.	Implemented using data segments.
Pointer is needed to accessing variables.	No need of Dynamically allocated pointers.
Faster execution than Dynamic.	Slower execution than static.
More memory Space required.	Less Memory space required.

```
#include <stdio.h>
#include <conio.h>
#include <alloc.h>
void main()
{
int n,*p,i,j,temp;
clrscr();
printf("\nHOW MANY NUMBER: ");
scanf("%d",&n);
p=(int *) malloc(n*2);
if(p==NULL)
{
 printf("\nMEMORY ALLOCATION UNSUCCESSFUL");
 exit();
}
for(i=0;i< n;i + + )
{
 printf("\nENTER NUMBER %d: ",i + 1);
 scanf("%d",p + i);
}
for(i=0;i < n;i + + )
{
 for(j=0;j < n;j + + )
 {
  if(*(p + i) < *(p + j))
  {
   temp=*(p + i);
   *(p + i)=*(p + j);
   *(p + j)=temp;
  }
 }
}
printf("\nTHE SORTED NUMBERS ARE:\n");
for(i=0;i < n;i + + )
 printf("%d ",*(p + i));
getch();
}
```

UNIT 2

FILES

SYLLABUS

Introduction, Defining and **opening a file**, closing a file, Input / Output operations on files, **Error** handling during I/O operations, Random Access to files, Command line arguments.

SYNOPSIS

A brief introduction:

A **file** represents a sequence of bytes on the disk where a group of related data is stored. File is created for permanent storage of data.

Opening & closing a file

Opening Files

You can use the **fopen()** function to create a new file or to open an existing file. This call will initialize an object of the type **FILE**, which contains all the information necessary to control the stream. The prototype of this function call is as follows –

FILE *fopen(const char * filename, const char * mode);

Here, **filename** is a string literal, which you will use to name your file, and access **mode can have** one of the following values –

Closing a File

To close a file, use the fclose() function. The prototype of this function is –

int fclose(FILE *fp);

The **fclose(-)** function returns zero on success, or **EOF** if there is an error in closing the file. This function actually flushes any data still pending in the buffer to the file, closes the file, and releases any memory used for the file. The EOF is a constant defined in the header file **stdio.h**.

5 MARKS QUESTIONS

1. What is a file? Explain how to open & close a file.

Ans. Opening a file:

A **file** represents a sequence of bytes on the disk where a group of related data is stored. File is created for permanent storage of data. It is a ready made structure.

The fopen() function is used to create a new file or to open an existing file.

General Syntax :

*fp = FILE ***fopen**(const char **filename*, const char **mode*);

Here **filename** is the name of the file to be opened and **mode** specifies the purpose of opening the file. Mode can be of following types,

***fp** is the FILE pointer (FILE *fp), which will hold the reference to the opened(or created) file.

Closing a file:

The fclose() function is used to close an already opened file.

General Syntax :

int **fclose**(FILE **fp*);

Here fclose() function closes the file and returns **zero** on success, or **EOF** if there is an error in closing the file. This **EOF** is a constant defined in the header file **stdio.h**.

2. Distinguish between the following functions.

Ans. (a) getc & getchar

The difference between getc() and getchar() is getc() can read from any input stream, but getchar() reads from standard input. So getchar() is equivalent to getc(stdin).

```
#include  < stdio.h>
int getc(FILE *stream);
int getchar(void);
```

(b) printf & fprintf

printf:

printf function is used to print character stream of data on standard output console.

Syntax:

```
int printf(const char* str, ...);
```

fprintf:

fprintf is used to print the sting content in file but not on standard output console.

```
int fprintf(FILE *fptr, const char *str, ...);
```

3. With an example explain how to handle errors during I/O operations.

Ans. It is possible that an error may occur during I/O operations on **a file. Typical** error situations include:

1. Trying to **read beyond** the end-of-file mark.
2. Device **overflow** means no space in the disk for storing data.
3. Trying to use **a file** that has not been opened.
4. Trying to **perform** operation on a file, when the file is opened for another type of operation.
5. Opening a file with an invalid filename.
6. Try to read a file with no data in it.

If we fail to check such read and write errors, a program may behave abnormally when an error occurs. An unchecked error may result in a premature termination of the program or incorrect output. Fortunately, we have two status library functions;

1. feof

2. ferror

that can help us to detect I/O errors in the files.

feof

The **feof** function can be used to test for an end of file condition. It takes a FILE pointer as its only argument and returns a nonzero integer value if all the data from the specified file has been read, and returns zero otherwise. If **fp** is a pointer to file that has just been opened for reading, then the statement

```
if (feof (fp ) )
        printf ("End of data.\n" );
```

would display the message "End of data." on reaching the end of file condition.

ferror

The **ferror** function reports the status of the file indicated. It also takes a FILE pointer as its argument and returns a nonzero integer if an error has been detected up to that point, during processing. It returns zero otherwise. The statement

```
if (ferror (fp ) ! = 0 )
        print ("An error has occurred.\n" ) ;
```

would print the error message, if the reading is not successful.

We know that whenever a file is opened using **fopen** function, **a file** pointer is returned. If the file cannot be opened for some reason, then the function returns a **NULL** pointer. This facility can be used to test whether a file has been opened or not. Example:

```
            if (fp == NULL )
    printf (" File could not be opened.\n");
```

4. Explain fseek() & ftell() functions.

Ans. Random Access To File

There is no need to read each record sequentially, if we want to access a particular record.C supports these functions for random access file processing.

1. fseek()
2. ftell()
3. rewind()

fseek():

This function is used for seeking the pointer position in the file at the specified byte.

Syntax: fseek(file pointer, displacement, pointer position);

Where

file pointer ---- It is the pointer which points to the file.

displacement ---- It is positive or negative.This is the number of bytes which are skipped backward (if negative) or forward(if positive) from the current position.This is attached with L because this is a long integer.

Pointer position:

This sets the pointer position in the file.

Value	pointer position
0	Beginning of file.
1	Current position
2	End of file

Ex: 1. fseek(p,10L,0)

0 **means pointer** position is on beginning of the file,from this statement pointer position is skipped 10 bytes **from the** beginning of the file.

2. fseek(p,5L,1)

1 means current position of the pointer position.From this statement pointer position is skipped 5 bytes forward from the current position..

3. fseek(p,-5L,1)

From this statement pointer position is skipped 5 bytes backward from the current position.

ftell()

This function returns the value of the current pointer position in the file.The value is count from the beginning of the file.

Syntax: ftell(fptr);

Where fptr is a file pointer.

rewind()

This function is used to move the file pointer to the beginning of the given file.

Syntax: rewind(fptr);

Where fptr is a file pointer.

4. Differentiate between the following:

Ans. (a) Feof & ferror

Ferror	feof
the **ferror function** tests to see if the error indicator has been set for a stream pointed to by *stream*.	the **feof function** tests to see if the end-of-file indicator has been set for a stream pointed to by *stream*.
Syntax The syntax for the ferror function in the C Language is: int ferror(FILE *stream);	Syntax The syntax for the feof function in the C Language is: int feof(FILE *stream);

stream The stream whose error indicator is to be tested. Returns The ferror function returns a nonzero value if the error indicator is set. Otherwise, it returns zero.	stream The stream whose end-of-file indicator is to be tested. Returns The feof function returns a nonzero value if the end-of-file indicator is set. Otherwise, it returns zero.
Required Header In the C Language, the required header for the ferror function is: #include < stdio.h>	Required Header In the C Language, the required header for the feof function is: #include < stdio.h>

(b) Printf & fprintf

Ans. Refer Q 2 (5 Marks)

(c) getc & getw

getw() function	getc() function
The getw() function is used to read integer value form the file.	The getc() function gets the next character (an unsigned char) from the specified stream and advances the position indicator for the stream.
Syntax int getw(FILE *fp);	**Syntax:** **int getc(FILE *stream)**

10 MARKS QUESTIONS

1. Write a program to copy contents of one file to another file.

Ans. Use command line arguments to specify the file names.

In C it is possible to accept command line arguments. Command-line arguments are given after the name of a program in command-line operating systems like DOS or Linux, and are passed in to the program from the operating system. To use command line arguments in your program, you must first understand the full declaration of the main function, which previously has accepted no arguments. In fact, main can actually accept two arguments: one argument is number of command line arguments, and the other argument is a full list of all of the command line arguments.

The full declaration of main looks like this:

int main (int argc, char *argv[])

The integer, argc is the argument count. It is the number of arguments passed into the program from the command line, including the name of the program.

You can use int main(int argc, char **argv) as your main function.

argc - will be the count of input arguments to your program.

argv - will be a pointer to all the input arguments.

So, if you entered C:\myprogram myfile.txt to run your program:

- argc will be 2
- argv[0] will be myprogram.
- argv[1] will be myfile.txt.

To read the file:

FILE *f = fopen(argv[1], "r"); // "r" for read

For opening the file in other modes,

2. List & explain reading & writing a file.

Ans. Writing a File

Following is the simplest function to write individual characters to a stream –

(a) int fputc(int c, FILE *fp);

The function **fputc()** writes the character value of the argument c to the output stream referenced by fp. It returns the written character written on success otherwise **EOF** if there is an error. You can use the following functions to write a null-terminated string to a stream –

(b) int fputs(const char *s, FILE *fp);

The function **fputs()** writes the string **s** to the output stream referenced by fp. It returns a non-negative value on success, otherwise **EOF** is returned in case of any error. You can use **int fprintf(FILE *fp,const char *format, ...)** function as well to write a string into a file. Try the following example.

Reading a File

Given below is the simplest function to read a single character from a file –

(a) int fgetc(FILE * fp);

The **fgetc()** function reads a character from the input file referenced by fp. The return value is the character read, or in case of any error, it returns **EOF**. The following function allows to read a string from a stream –

(b) char *fgets(char *buf, int n, FILE *fp);

The functions **fgets()** reads up to n-1 characters from the input stream referenced **by fp. It copies** the read string into the buffer **buf**, appending a **null** character to terminate the string.

If this function encounters a newline character '\n' or the end of the file EOF before they have read the maximum number of characters, then it returns only the characters read up to that point including the new line character. You can also use **int fscanf(FILE *fp, const char *format, ...)** function to read strings from a file, but it stops reading after encountering the first space character.

4. explain different file access modes.

Ans.

Mode	Description
r	Opens an existing text file for reading purpose.
w	Opens a text file for writing. If it does not exist, then a new file is created. Here your program will start writing content from the beginning of the file.
a	Opens a text file for writing in appending mode. If it does not exist, then a new file is created. Here your program will start appending content in the existing file content.
r +	Opens a text file for both reading and writing.
w +	Opens a text file for both reading and writing. It first truncates the file to zero length if it exists, otherwise creates a file if it does not exist.
a +	Opens a text file for both reading and writing. It creates the file if it does not exist. The reading will start from the beginning but writing can only be appended

UNIT 3

INTRODUCTION TO DATA STRUCTURES & LINKED LISTS

SYLLABUS

Introduction to data structures – Introduction, Characteristics, Types of data structures, data structure operations..

Linked lists – Introduction, Basic concept, linked list implementation, Types of linked lists, Circular linked list (no implementation), doubly linked list (no implementation).

SYNOPSIS

A brief introduction:

Data Structure is a way of collecting and organising data in such a way that we can perform operations on these data in an effective way.

Types of ds:

Linear – stacks,queues

Nonlinear – linked list, trees

Operations on Data Structures

Basically there are six operations one can do on the data structures. They are **Traversing, Searching, Sorting, Insertion, Deletion** and **Merging**.

1. Define linked list. Mention the different types of linked list.

Linked List is a sequence of links which contains items. Each link contains a connection to another link.

Types of Linked List

Following are the various types of linked list.

- **Simple Linked List** – Item navigation is forward only.
- **Doubly Linked List** – Items can be navigated forward and backward.
- **Circular Linked List** – Last item contains link of the first element as next and the first element has a link to the last element as previous.

Basic Operations on linked list:

Following are the basic operations supported by a list - Insertion, Deletion, Display, Search

Advantages & disadvantages of linked list:

Advantages:

1. Linked List is **Dynamic data Structure**.

2. Linked List **can grow and shrink during run time.**

3. **Insertion and Deletion** Operations are Easier

4. **Efficient Memory Utilization** ,i.e no need to pre-allocate memory

5. Faster Access time,can be expanded in **constant time without memory overhead**

6. Linear Data Structures such as Stack,Queue can be **easily implemented** using Linked list

Disadvantages:

1. Wastage of Memory –

1. Pointer Requires extra memory for storage.

2. Suppose we want to store 3 integer data items then we have to allocate memory –

in case of array –

Memory Required in Array = 3 Integer * Size

= 3 * 2 bytes

= 6 bytes

in case of array –

Memory Required in LL = 3 Integer * Size of Node

= 3 * Size of Node Structure

= 3 * Size(data + address pointer)

= 3 * (2 bytes + x bytes)

= 6 bytes + 3x bytes

*x is size of complete node structure it may vary

2. No Random Access

1. In array we can access nth element easily just by using a[n].

2. In Linked list no random access is given to user, we have to access each node sequentially.

3. Suppose we have to access nth node then we have to traverse linked list n times.

Suppose Element is present at the starting location then –

We can access element in first Attempt

Suppose Element is present at the Last location then –

We can access element in las' Attempt

3. Time Complexity

1. Array can be randomly accessed , while the Linked list cannot be accessed Randomly

2. Individual nodes are not stored in the contiguous memory Locations.

3. Access time for Individual Element is O(n) whereas in Array it is O(1).

4. Reverse Traversing is difficult

1. In case if we are using singly linked list then it is very difficult to traverse linked list from end.

2. If using doubly linked list then though it becomes easier to traverse from end but still it increases again storage space for back pointer.

5. Heap Space Restriction

1. Whenever memory is dynamically allocated , It utilizes memory from heap.

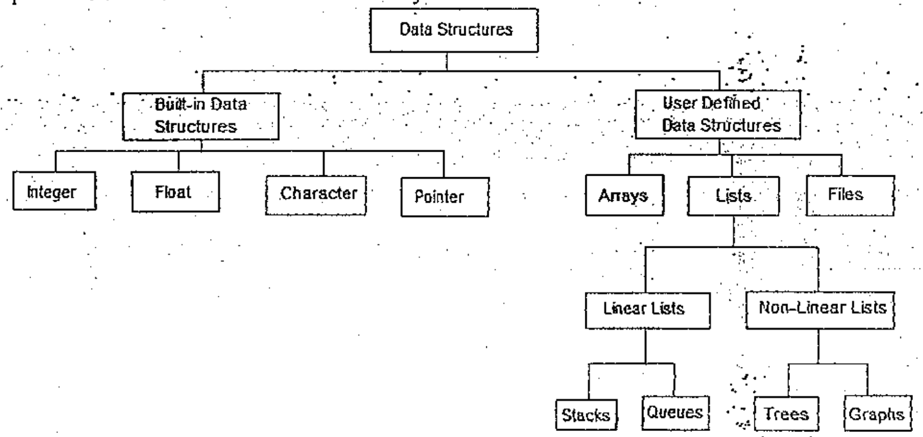
2. Memory is allocated to Linked List at run time if and only if there is space-available in heap.

3. If there is insufficient space in heap then it won't create any memory.

5 MARKS QUESTIONS

1. Define data structures & mention different types of data structures.

Ans. Data Structure is a way of collecting and organising data in such a way that we can perform operations on these data in an effective way.



INTRODUCTION TO DATA STRUCTURES

2. What are primitive types explain.

Ans. Primitive Data Types

The primitive data types in c language are the inbuilt data types provided by the c language itself. Thus, all c compilers provide support for these data types.

The following primitive data types in c are available:

Integer Data Type, int

Integer data type is used to declare a variable that can store numbers without a decimal. The keyword used to declare a variable of integer type is "int". Thus, to declare integer data type following syntax should be followed:

int variable_name;

Float data Type, float

Float data type declares a variable that can store numbers containing a decimal number.

Syntax

float variable_name;

Double Data Type, double

Double data type also declares variable that can store floating point numbers but gives precision double than that provided by float data type. Thus, double data type are also referred to as double precision data type.

Syntax

double variable_name;

Character Data Type, char

Character data type declares a variable that can store a character constant. Thus, the variables declared as char data type can only store one single character.

Syntax

char variable_name;

Void Data Type, void

Unlike other primitive data types in c, void data type does not create any variable but returns an empty set of values. Thus, we can say that it stores null.

Syntax

void variable_name;

3. Distinguish between linear & non linear data structures.

Ans.

Linear DS:	non-linear DS:
1. every item is related to its previous and next time.	1. every item is attached with many other items.
2. ~~data is arranged in~~ linear sequence.	2. data is not arranged in sequence.
3. data items can be traversed in a single run.	3. data cannot be traversed in a single run.
4. eg. array, stcks, linked list, queue.	4. eg. tree, graph
5. implementation is easy	5. implementation is difficult.

4. Explain data structure operations.

Ans. 1. Traversing: Basically to process a data structure if every element of data structure is visited once and only once, such type of operation is called as TRAVERSING. For example to display all the elements of an array, every element is visited once and only once, so it is called as traversing operation.

2. Insertion: When an element of the same type is added to an existing data structure, the operation we are doing is called as Insertion operation. The element can be added anywhere in the data structure in the data structure. When the element is added in the end it is called as special type addition, Appending. In case of adding an element to the data structure we may come across 'Overflow' If the size of the data structure is fixed and it is full, then if we try insertion operation on the data structure it is said to be overflow of data structure or the data structure is full.

3. Deletion: When an element is removed from the data structure, the operation we are doing is called as Deletion operation. We can delete an element from data structure from any position. In case of deleting an element from the data structure we may come across 'Underflow'. If no elements are stored in the data structure, then if we try deletion operation on the data structure it is said to be underflow of data structure or data structure is empty.

4. Searching: When an element is checked for its presence in a data structure, that operation we are doing is called as 'searching' operation. The element that is to be searched is called as key element. The searching can be done using either 'linear search' or 'binary search'.

5. Sorting: When all the elements of array are arranged in either ascending or descending order, the operation used to do this process is called as Sorting. The Sorting can be done using Insertion, Selection or Bubble sort techniques.

6. Merging: When two lists List A and List B of size M and N respectively, of same type of elements, clubbed or joined to produce the third list, List C of size (M + N), and the operation done during the process is called as Merging.

5. Define linked list. Mention the different types of linked list.

Ans. Linked List is a sequence of links which contains items. Each link contains a connection to another link.

Types of Linked List

Following are the various types of linked list.

- **Simple Linked List** – Item navigation is forward only.
- **Doubly Linked List** – Items can be navigated forward and backward.
- **Circular Linked List** – Last item contains link of the first element as next and the first element has a link to the last element as previous.

6. Explain the representation of linked list in memory with the help of an illustration.

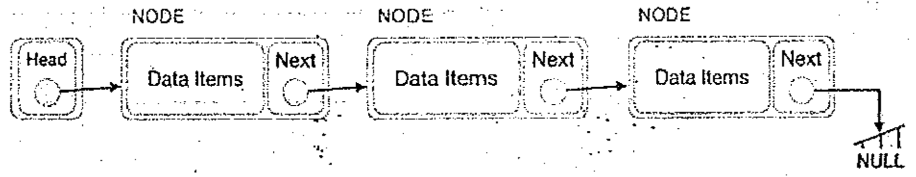
Ans. A linked list is a sequence of data structures, which are connected together via links.

Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.

- **Link** – Each link of a linked list can store a data called an element.
- **Next** – Each link of a linked list contains a link to the next link called Next.
- **LinkedList** – A Linked List contains the connection link to the first link called First.

Linked List Representation

Linked list can be visualized as a chain of nodes, where every node points to the next node.



As per the above illustration, following are the important points to be considered.

- Linked List contains a link element called first.
- Each link carries a data field(s) and a link field called next.
- Each link is linked with its next link using its next link.
- Last link carries a link as null to mark the end of the list.

7. Explain the operations that are performed on singly linked list.

Ans. Basic Operations

Following are the basic operations supported by a list.

- **Insertion** – Adds an element at the beginning of the list.
- **Deletion** – Deletes an element at the beginning of the list.
- **Display** – Displays the complete list.
- **Search** – Searches an element using the given key.

8. Write the advantages & disadvantages of linked list.

Advantages:

1. Linked List is **Dynamic data Structure** .
2. Linked List **can grow and shrink during run time**.
3. **Insertion and Deletion** Operations are Easier
4. **Efficient Memory Utilization**, i.e no need to pre-allocate memory
5. Faster Access time,can be expanded in **constant time without memory overhead**
6. Linear Data Structures such as Stack,Queue can be **easily implemented** using Linked list

Disadvantages:

1. **Wastage of Memory** –Pointer Requires extra memory for storage.
2. No Random Access - In Linked list no random access is given to user, we have to access each node sequentially.
3. Time Complexity - Individual nodes are not stored in the contiguous memory Locations.

4. Reverse Traversing is difficult - In case if we are using singly linked list then it is very difficult to traverse linked list from end.
5. Heap Space Restriction - Memory is allocated to Linked List at run time if and only if there is space available in heap. If there is insufficient space in heap then it won't create any memory.

5. Compare singly linked list with circular linked list

singly linked list	circular linked list
But in linear linked list it is not possible to go to previous node.	If we at a node and go back to the previous node, then we can not do it in single step. Instead we have to complete the entire circle by going through the in between nodes and then we will reach the required node.
Such problems are not applicable	If proper care is not taken, then the problem of infinite loop can occur.
It is not possible to traverse from last node to first node	It takes a single step
Adding & deleting nodes is easy	Adding & deleting nodes is difficult

6. Compare singly linked list with doubly linked list.

S. No.	Singly Linked List	Doubly Linked List
1	Singly linked list allows you to go one way direction	Doubly linked list has two way directions next and previous
2	Singly linked list uses less memory per node (one pointer)	Doubly linked list uses More memory per node than Singly Linked list (two pointers)
3	There is a little-known trick that lets you delete from a singly-linked list in O(1), but the list must be circular for it to work (move the content of next into the current, and delete next).	Doubly-linked lists can be used in places where singly-linked lists would not work (a doubly-ended queue), but they require slightly more "housekeeping", and are slightly less efficient on insertions as the result
4	Complexity of Insertion and Deletion at known position is O (n).	Complexity of Insertion and Deletion at known position is O (1).
5	If we need to save memory in need to update node values frequently and searching is not required, we can use Singly Linked list.	If we need faster performance in searching and memory is not a limitation we use Doubly Linked List
6		For B-Tree, Heap we need doubly linked list. .Net Framework only provides the LinkedList < T > class which is double-linked.

7	If we know in advance that element to be searched is found near the end of the list(for example name 'Yogesh' in a telephone directory), even then singly linked list is traversed sequentially from beginning.	In doubly linked list If we know in advance that element to be searched is found near the end of the list(for example name 'Yogesh' in a telephone directory), then the list can traversed from the end thereby saving time
8	In single list Each node contains at least two parts: a) info b) link	In doubly linked list Each node contains at least three parts: a) info b) link to next node c) link to previous node

10 MARKS QUESTIONS

1. Write C function to insert at the end & display operations on singly linked list.

Ans. Refer Q 7 (10 Marks)

2. Write C function to insert at the front & delete operations on singly linked list.

Ans. Refer Q7 (10 Marks)

3. Write C function to insert at the given position operations on singly linked list.

Ans. Code for Singly Linked list with following operations CREATE, INSERT AT STARTING, INSERT AT MIDDLE, INSERT AT END, DELETE FIRST NODE, DELETE LAST NODE, DELETE MIDDLE WITH DISPLAY in C

```
#include < conio.h>
#include < stdio.h>
struct node
{
  int i;
  struct node *next;
};
void main()
{
  struct node *first;
  struct node *last;
  struct node *temp;
  int ch,user,add,cnt=0,t=0;
  struct node *p;
```

```
clrscr();
printf("\n\t 1.CREATION");
printf("\n\t 2.INSERT AT STARTING");
printf("\n\t 3.INSERT AT MIDDLE(USER'S CHOICE)");
printf("\n\t 4.INSERT AT END");
printf("\n\t 5.DELETE 1ST NODE");
printf("\n\t 6.DELETE LAST NODE");
printf("\n\t 7.DELETE MIDDLE NODE(USER'S CHOICE)");
printf("\n\t 8.DISPLAY");
printf("\n\t 10.EXIT");
scanf("%d",&user);
while(user!=10)
{
 if(user==1)
 {
  printf("\n\t ENTER DATA ::: ");
  first=(struct node*)malloc(sizeof(struct node));
  scanf("%d",&ch);
  first->i=ch;
  first->next=0;
  p=first;
  cnt=1;
 }
 if(user==2)
 {
   p=(struct node*)malloc(sizeof(struct node));
   printf("\n\t ENTER DATA FOR 1ST NODE");
   scanf("%d",&p->i);
   p->next=first;
   first=p;
   cnt ++ ;
 }
 if(user==4)
```

```
  {
    p=(struct node*)malloc(sizeof(struct node));
    printf("\n\t ENTER DATA FOR LAST NODE");
    scanf("%d";&p->i);
    temp=first;
    while(temp->next!=0)
    {
      temp=temp->next;
    }
    temp->next=p;
    p->next=0;
    cnt + + ;
  }
  if(user==3)
  {
    printf("\n\t ENTER ANY ADDRESS BETWEEN 1 and %d",cnt);
    scanf("%d",&add);
    t=1;
    p=first;
    while(t!=add)
    {
      p=p->next;
      t + + ;
    }
    temp=(struct node*)malloc(sizeof(struct node));
    printf("\n\t ENTER DATA FOR NODE");
    scanf("%d",&temp->i);
    temp->next=p->next;
    p->next=temp;
    cnt + + ;
  }
  if(user==5)
  {
```

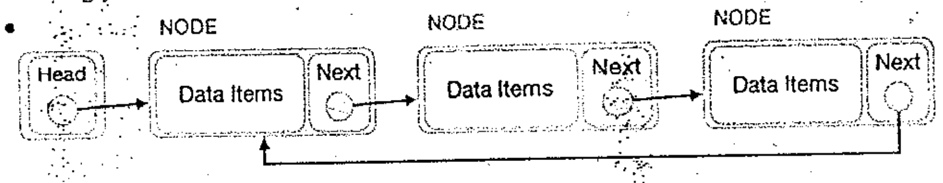
```
    p=first;
    first=p->next;
    free(p);
  }
  if(user==6)
  {
   p=first;
   while(p->next->next!=0)
   {
     p=p->next;
   }
   p->next=0;
   free(p->next->next);
  }
  if(user==8)
  {
   p=first;
   while(p!=0)
   {
     printf("\n\t %d",p->i);
     p=p->next;
   }
  }
  if(user==7)
  {
    printf("\n\t ENTER ANY ADDRESS BETWEEN 1 and %d",cnt);
    scanf("%d",&add);
    t=1;
    p=first;
    while(t < add-1)
    {
      p=p->next;
      t + + ;
    }
    temp=p->next;
    p->next=temp->next;
    free(temp);
    cnt--;
  }
  printf("\n\t 1.CREATION");
  printf("\n\t 2.INSERT AT STARTING");
  printf("\n\t 3.INSERT AT MIDDLE(USER'S CHOICE)");
  printf("\n\t 4.INSERT AT END");
  printf("\n\t 5.DELETE 1ST NODE");
  printf("\n\t 6.DELETE LAST NODE");
  printf("\n\t 7.DELETE MIDDLE NODE(USER'S CHOICE)");
  printf("\n\t 8.DISPLAY");
  printf("\n\t 10.EXIT");
  scanf("%d",&user);
  }
  getch();
}
```

4. Define circular linked list. Give its C representation.

Ans. • Circular Linked List is a variation of Linked list in which the first element points to the next element and the last element points to the first element. Both Singly Linked List and Doubly Linked List can be made into a circular linked list.

• In singly linked list, the next pointer of the last node points to the first node.

•



Code for Circular link list with create, insert, delete, display operations using structure pointer in C Programming

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

```
struct circular
{
  int i;
  struct circular *next;
};
struct circular *temp;
struct circular *head;
struct circular *p;
struct circular *mid;
struct circular *move;
int cnt=0;
void create(void);
void insert(void);
void display(void);
void del(void);
void main()
{
  int ch=0;
  clrscr();
  while(ch!=5)
  {
    printf("\n1.CREATE");
    printf("\n2.INSERT");
    printf("\n3.DELETE");
    printf("\n4.DISPLAY");
    printf("\n5.EXIT");
    scanf("%d",&ch);
    if(ch==1)
    {
      create();
      cnt + + ;
      cnt + + ;
    }
```

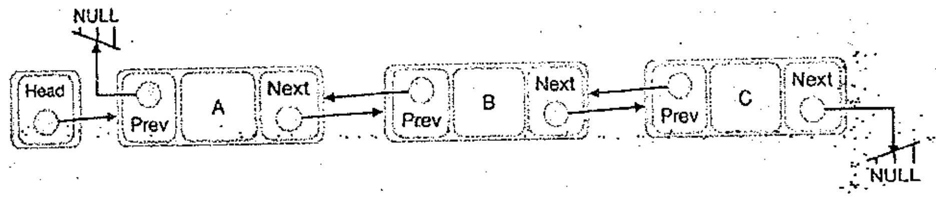
```
      if(ch==2)
      {
        insert();
        cnt + + ;
      }
      if(ch==3)
      {
        del();
        cnt--;
      }
      if(ch==4)
      {
        display();
      }
      if(ch==5)
      {
        break;
      }
    }
    getch();
  }
  void create()
  {
    head=(struct circular *)malloc(sizeof(struct circular));
    head->next=head;
    printf("ENETER THE DATA");
    scanf("%d",&head->i);
    temp=head;
    temp->next=(struct circular *)malloc(sizeof(struct circular));
    temp=temp->next;
    temp->next=head;
    printf("ENETER THE DATA");
    scanf("%d",&temp->i);
```

```
}
void insert()
{
  int add,t;
  printf("\n\t ENTER ANY NUMBER BETWEEN-1 AND %d",cnt);
  scanf("%d",&add);
  p=head;
  t=1;
  while(t < add)
  {
    p=p->next;
    t + + ;
  }
  printf("%d",p->i);
  clrscr();
  mid=(struct circular *)malloc(sizeof(struct circular));
  printf("ENETER THE DATA");
  scanf("%d",&mid->i);
  mid->next=p->next;
  p->next=mid;
}
void display()
{
  p=head;
  printf("%d-->",p->i);
  p=p->next;
  while(p!=head)
  {
    printf("%d-->",p->i);
    p=p->next;
  }
}
void del(void)
```

```
{
  int add,t;
  printf("\n\t ENTER ANY NUMBER BETWEEN 1 AND %d",cnt);
  scanf("%d",&add);
  p=head;
  t=1;
  while(t < add-1)
  {
    p=p->next;
    t++;
  }
  printf("%d",p->i);
  clrscr();
  mid=p->next;
  p->next=mid->next;
}
```

5. Define doubly linked list. Give its C representation.

Ans. Doubly Linked List is a variation of Linked list in which navigation is possible in both ways, either forward and backward easily



Code for Doublely link list with create, insert, delete and display ~~operations using structure~~ pointer in C Programming

```
#include < conio.h>
#include < stdio.h>
struct doubly
{
  struct doubly *front;
  struct doubly *back;
```

```
  int i;
};
void main()
{
  struct doubly *head=0;
  struct doubly *last=0;
  struct doubly *p=0;
  struct doubly *temp=0;
  int ch,intch,user,cnt=0,t,add;
  clrscr();
  printf("\n\t 1: CREATE.");
  printf("\n\t 2. INSERT.");
  printf("\n\t 3. DELETE.");
  printf("\n\t 4. DISPLAY.");
  printf("\n\t 5.EXIT.");
  scanf("%d",&ch);
  while(ch!=5)
  {
    if(ch==1)
    {
      cnt=0;
      head=(struct doubly *)malloc(sizeof(struct doubly));
      head->back=0;
      head->front=0;
      printf("\n\t ENTER DATA::: ");
      scanf("%d",&head->i);
      cnt++;
      last=head;
    }
    if(ch==2)
    {
      printf("\n\t 1. INSERTION AT FRONT.");
      printf("\n\t 2. INSERTION AT MIDDLE.");
      printf("\n\t 3. INSERTION AT END.");
```

```
scanf("%d",&intch);
temp=(struct doubly *)malloc(sizeof(struct doubly));
printf("\n\t ENTER DATA:::");
scanf("%d",&temp->i);
if(intch==1)
{
  temp->front=0;
  temp->back=head;
  head=temp;
  cnt + + ;
}
if(intch==3)
{
  p=head;
  while(p->back!=0)
  {
    p=p->back;
  }
  temp->back=0;
  temp->front=p;
  p->back=temp;
  cnt + + ;
}
if(intch==2)
{
  printf("\n\tNTER VALUE BETVN 1--%d",cnt);
  scanf("%d",&add);
  t=1;
  p=head;
  while(t < add)
  {
    p=p->back;
    t + + ;
  }
```

```
      temp->back=p->back;
      p->back->front=temp;
      p->back=temp;
      temp->front=p;
    }
  }
  if(ch==4)
  {
    p=head;
    while(p!=0)
    {
      printf("\t-->%d",p->i);
      p=p->back;
    }
    printf("\n\t total nodes %d",cnt);
  }
  if(ch==3)
  {
    printf("\n\t1. DELETE FRONT");
    printf("\n\t2. DELETE MIDDLE");
    printf("\n\t3. DELETE END");
    scanf("%d",&intch);
    if(intch==1)
    {
      head->back=t;
      free(head);
      head=t;
      head->front=0;
      cnt--;
    }
    if(intch==3)
    {
      p=head;
```

```
            while(p->back!=0)
            {
              p=p->back;
            }
            temp->front=p;
            p->back=temp;
            temp->back=0;
            cnt--;
          }
          if(intch==2)
          {
            printf("\n\tNTER VALUE BETVN 1--%d",cnt);
            scanf("%d",&add);
            t=1;
            p=head;
            while(t < (add-1))
            {
              p=p->back;
              t + + ;
            }
            temp=p->back;
            p->back=temp->back;
            temp->back->front=p;
            free(temp);
          }
        }
        printf("\n\t 2. INSERT.");
        printf("\n\t 3. DELETE.");
        printf("\n\t 4. DISPLAY.");
        printf("\n\t 5.EXIT.");
        scanf("%d",&ch);
    }
    getch();
}
```

UNIT 4

STACK & QUEUES

SYLLABUS

Stack – Introduction, Stacks, Stack operations, stack implementations.

Queues – Introduction, Basic concept, queue operations, queue implementations, circular queue (no implementation), priority queues (no implementation), double ended queues (no implementation).

SYNOPSIS

A brief introduction:

A stack is a container of objects that are inserted and removed according to the last-in first-out (LIFO) principle.

Two operations can be performed on stack

a) PUSH: inserting elements at stack top

b) POP: deleting elements from stack top

Queue is also an abstract data type or a linear data structure, in which the element is inserted from one end called **REAR**(also called tail), and the deletion of exisiting element takes place from the other end called as **FRONT**(also called head). This makes queue as FIFO data structure, which means that element inserted first will also be removed first.

Types of queues:

1.Simple or linear queue

2.Circular queue:.

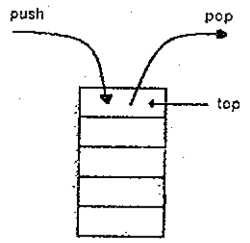
3.Priority queue:

4.Dequeue:

5 MARKS QUESTIONS

1. Define stack. Explain how to represent stack in c.

Ans. A stack is a container of objects that are inserted and removed according to the last-in first-out (LIFO) principle.



Representation of stack in c:

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

We have created 'stack' structure.

2. We have array of elements having size 'size'

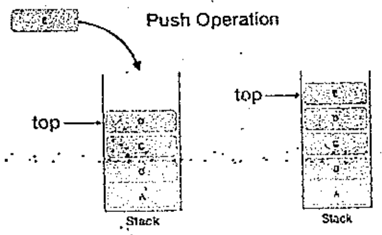
3. To keep track of Topmost element we have declared top as structure member.

2. Explain push & pop operation on stack.

Ans. Push Operation

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps –

- **Step 1** – Checks if the stack is full.
- **Step 2** – If the stack is full, produces an error and exit.
- **Step 3** – If the stack is not full, increments **top** to point next empty space.
- **Step 4** – Adds data element to the stack location, where top is pointing.
- **Step 5** – Returns success.



If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.

Algorithm for PUSH Operation

A simple algorithm for Push operation can be derived as follows –

```
begin procedure push: stack, data
  if stack is full
    return null
  endif
  top ← top + 1
```

```
  stack[top] ← data
end procedure
```

Implementation of this **algorithm in C**, is very easy. See the following code –

Example

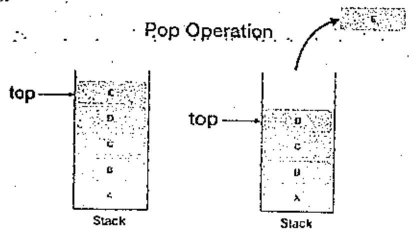
```
void push(int data) {
  if(!isFull()) {
    top = top + 1;
    stack[top] = data;
  } else {
    printf("Could not insert data, Stack is full.\n");
  }
}
```

Pop Operation

Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead **top** is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data element and deallocates memory space.

A Pop operation may involve the following steps –

- **Step 1** – Checks if the stack is empty.
- **Step 2** – If the stack is empty, produces an error and exit.
- **Step 3** – If the stack is not empty, accesses the data element at which **top** is pointing.
- **Step 4** – Decreases the value of top by 1.
- **Step 5** – Returns success.



Algorithm for Pop Operation

A simple algorithm for Pop operation can be derived as follows –

```
begin procedure pop: stack
  if stack is empty
```

```
    return null
  endif
  data ← stack[top]

  top ← top - 1
  return data
end procedure
```

Implementation of this algorithm in C, is as follows –

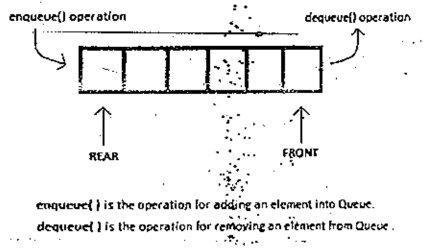
Example

```
int pop(int data) {
  if(!isempty()) {
    data = stack[top];
    top = top - 1;
    return data;
  } else {
    printf("Could not retrieve data, Stack is empty.\n");
  }
}
```

3. Define queue. Explain the sequential representation of queue.

Ans. Queue is also an abstract data type or a linear data structure, in which the element is inserted from one end called **REAR**(also called tail), and the deletion of exisiting element takes place from the other end called as **FRONT**(also called head). This makes queue as FIFO data structure, which means that element inserted first will also be removed first.

The process to add an element into queue is called **Enqueue** and the process of removal of an element from queue is called **Dequeue**.


enqueue() operation
dequeue() operation
REAR
FRONT
enqueue( ) is the operation for adding an element into Queue.
dequeue( ) is the operation for removing an element from Queue.

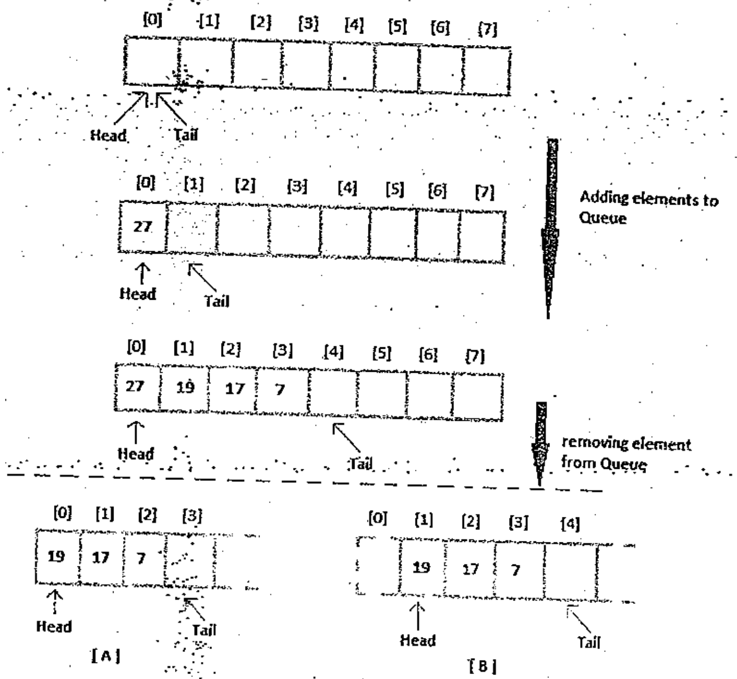
QUEUE DATA STRUCTURE

Basic features of Queue

1. Like Stack, Queue is also an ordered list of elements of similar data types.
2. Queue is a FIFO(First in First Out) structure.
3. Once a new element is inserted into the Queue, all the elements inserted before the new element in the queue must be removed, to remove the new element.

4. Write a note on queue.

Ans. Queue can be implemented using an Array, Stack or Linked List. The easiest way of implementing a queue is by using an Array. Initially the **head**(FRONT) and the **tail**(REAR) of the queue points at the first index of the array (starting the index of array from 0). As we add elements to the queue, the tail keeps on moving ahead, always pointing to the position where the next element will be inserted, while the head remains at the first index.



When we remove element from Queue, we can follow two possible approaches (mentioned [A] and [B] in above diagram). In [A] approach, we remove the element at **head** position, and then one by one move all the other elements on position forward. In approach [B] we remove the element from **head** position and then move **head** to the next position.

In approach [A] there is an overhead of shifting the elements one position forward every time we remove the first element. In approach [B] there is no such overhead, but when ever we move head one position ahead, after removal of first element, the size on Queue is reduced by one space each time.

queue is structured, as described above, as an ordered collection of items which are added at one end, called the "rear," and removed from the other end, called the "front." Queues maintain a FIFO ordering property. The queue operations are given below.

- Queue() creates a new queue that is empty. It needs no parameters and returns an empty queue.
- enqueue(item) adds a new item to the rear of the queue. It needs the item and returns nothing.
- dequeue() removes the front item from the queue. It needs no parameters and returns the item. The queue is modified.
- isEmpty() tests to see whether the queue is empty. It needs no parameters and returns a boolean value.
- size() returns the number of items in the queue. It needs no parameters and returns an integer.

5. Define priority queue. Differentiate ascending & descending priority queue.

Ans. Priority queue is a variant of queue data structure in which insertion is performed in the order of arrival and deletion is performed based on the priority.

There are two types of priority queues they are as follows...

1. Max Priority Queue
2. Min Priority Queue

1. Max Priority Queue

In max priority queue, elements are inserted in the order in which they arrive the queue and always maximum value is removed first from the queue. For example assume that we insert in order 8, 3, 2, 5 and they are removed in the order 8, 5, 3, 2.

3. Min Priority Queue

Min Priority Queue is similar to max priority queue except removing maximum element first, we remove minimum element first in min priority queue.

10 MARKS QUESTIONS

1. Write a c program to implement push & pop operation on stack

Define queue. List & explain different types of queues.

Ans. Queue is a Linear Data Structure that works on First-in-First-Out (FIFO) principle.

- It has two pointers, 'Front' that points to the beginning of the queue and 'Rear' that points to the end of the queue.
- The 'Front' and 'Rear' pointers are manipulated constantly to always point to the beginning and end of queue.

- It can be implemented using Arrays and Linked Lists (Recursive and Non-recursive) methods both.
- **Different** types of queues:

1. Simple or linear queue

Linked lists are among the simplest and most common data structures.

They can be used to implement several other common abstract data types, including lists(the abstract data type), stacks, queues, associative arrays, and S-expressions, though it is not uncommon to implement the other data structures directly without using a list as the basis of implementation.

2. Circular queue

Another common implementation of a queue is a circular buffer. "Buffer" is a general name for a temporary storage location, although it often refers to an array, as it does in this case.

A circular buffer, cyclic buffer or ring buffer is a data structure that uses a single, fixed-size buffer as if it were connected end-to-end. This structure lends itself easily to buffering data streams.

3. Priority queue

The Priority Queue ADT has the same interface as the Queue ADT, but different semantics.

The semantic difference is that the item that is removed from the queue is not necessarily the first one that was added. Rather, it is whatever item in the queue has the highest priority.

4. Dequeue

A double-ended queue (Dequeue) is an abstract data type that generalizes a queue, for which elements can be added to or removed from either the front (head) or back (tail).

It is also often called a head-tail linked list, though properly this refers to a specific data structure implementation.

2. Write a program to implement queue in c.

Ans.

```
#include< stdio.h>
#include < conio.h>
#define MAX 10
int queue[MAX], front = -1, rear = -1;
void Insert_Element();
void Delete_Element();
void Display_Queue();
void Empty_Queue();
```

```
int main()
{
int option;
printf(">>> c program to implement queue operations < < <");
do
{
printf("\n\n 1.Insert an element»);
printf("\n 2.Delete an element»);
printf("\n 3.Display queue»);
printf("\n 4.Empty queue»);
printf("\n 5.Exit»);
printf("\n Enter your choice: «);
scanf("%d", &option);
switch (option)
{
case 1: Insert_Element();
break;
case 2: Delete_Element();
break;
case 3: Display_Queue();
break;
case 4: Empty_Queue();
break;
case 5: return 0; /*program ends*/
}
} while (option != 5);
}
void Insert_Element()
{
int num;
if (rear < MAX - 1)
{
if (front == -1)
```

```
/*when queue is initially empty */
front = 0;
printf("\n Enter the number to be inserted: «);
scanf("%d", &num);
rear = rear + 1;
queue[rear] = num;
}
else
{
printf("\n Queue OverFlow Occured»);
}
}
void Delete_Element()
{
int element;

if (front == -1 || front > rear)
{
printf("\n Queue Underflow occured.\n");
return;
}
else
{
element = queue[front];
printf("\n Element deleted from queue is: %d», element);
front = front + 1;
}
}
void Display_Queue()
{
int i;
if (front == -1 || front > rear)
printf("\n No elements to display»);
```

```
else
 {
printf("\n The queue elements are:\n «);
for (i = front; i <= rear; i + + )
 {
printf("\t %d», queue[i]);
 }
 }
}
void Empty_Queue()
 {
 /*Reset queue or Creates Empty queue*/
 front = -1;
 rear = -1;
 printf("\n New Queue created successfully.»);
 }
```

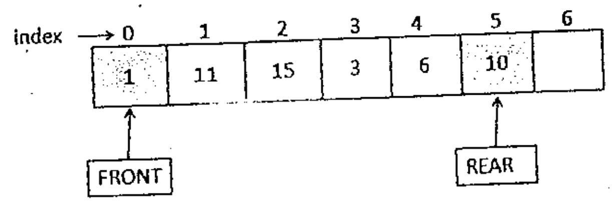
3. Write an algorithm to perform queue insertion & deletion.

Ans. Queue is ordered collection of homogeneous data elements in which insertion and deletion operation take place at two end . insertion allowed from starting of queue called FRONT point and deletion allowed from REAR end only

- insertion operation is called ENQUEUE
- deletion operation is called DEQUEUE

Block Diagram of Queue

Queue Block Diagram



iAmLearningHere

Conditions in Queue

- FRONT < 0 (Queue is Empty)

- REAR = Size of Queue (Queue is Full)
- FRONT < REAR (Queue contains at least one element)
- No of elements in queue is: (REAR - FRONT) + 1

Restriction in Queue

we can not insert element directly at middle index (position) in Queue and vice verse for deletion. insertion operation possible at REAR end only and deletion operation at FRONT end, to insert we increment REAR and to delete we increment FRONT.

Algorithm for ENQUEUE (insert element in Queue)

Input: An element say ITEM that has to be inserted.

Output: ITEM is at the REAR of the Queue.

Data structure: Que is an array representation of queue structure with two pointer FRONT and REAR.

```
Steps:
If (REAR = size ) then //Queue is full
print "Queue is full"
Exit
Else
If (FRONT = 0 ) and (REAR = 0 ) then //Queue is empty
FRONT = 1
End if
REAR = REAR + 1 // increment REAR
Que[ REAR ] = ITEM
End if
Stop
```

Algorithm for DEQUEUE (delete element from Queue)

Input: A que with elements. FRONT and REAR are ~~two pointer of queue.~~

Output: The deleted element is stored in ITEM.

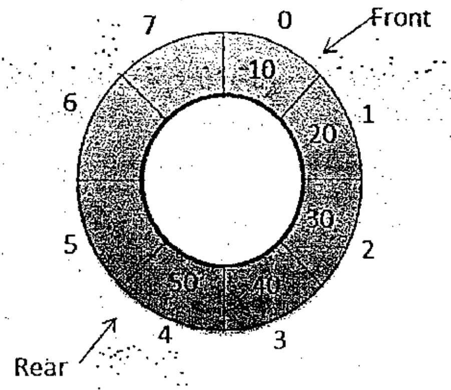
Data structure: Que is an array representation of queue structure..

```
Steps:
If (FRONT = 0 ) then
print "Queue is empty"
Exit
Else
```

```
ITEM = Que [ FRONT ]
If (FRONT = REAR )
REAR = 0
FRONT = 0
Else
FRONT = FRONT + 1
End if
End if
Stop
```

4. Define circular queue. Write the c implementation of circular queue.

Ans.



```
#include < stdio.h>
#define max 3
int q[10],front=0,rear=-1;
void main()
{
  int ch;
  void insert();
  void delet();
  void display();
  clrscr();
  printf("\nCircular Queue operations\n");
```

```
 printf("1.insert\n2.delete\n3.display\n4.exit\n");
 while(1)
 {
   printf("Enter your choice:");
   scanf("%d",&ch);
   switch(ch)
   {
   case 1: insert();
     break;
   case 2: delet();
     break;
   case 3:display();
     break;
   case 4:exit();
   default:printf("Invalid option\n");
   }
 }
}
void insert()
{
 int x;
 if((front==0&&rear==max-1)||(front>0&&rear==front-1))
   printf("Queue is overflow\n");
 else
 {
   printf("Enter element to be insert:");
   scanf("%d",&x);
   if(rear==max-1&&front>0)
   {
     rear=0;
     q[rear]=x;
   }
   else
```

```
    {
      if((front==0&&rear==-1)||(rear!=front-1))
         q[ + + rear]=x;
    }
  }
}
void delet()
{
  int a;
  if((front==0)&&(rear==-1))
  {
   printf("Queue is underflow\n");
   getch();
   exit();
  }
  if(front==rear)
  {
    a=q[front];
    rear=-1;
    front=0;
  }
  else
    if(front==max-1)
    {
      ~~a=q[front];~~
      front=0;
    }
    else a=q[front + + ];
    printf("Deleted element is:%d\n",a);
}
void display()
{
  int i,j;
```

```
if(front==0&&rear==-1)
{
  printf("Queue is underflow\n");
  getch();
  exit();
}
if(front>rear)
{
  for(i=0;i < =rear;i + + )
    printf("\t%d",q[i]);
  for(j=front;j < =max-1;j + + )
    printf("\t%d",q[j]);
  printf("\nrear is at %d\n",q[rear]);
  printf("\nfront is at %d\n",q[front]);
}
else
{
  for(i=front;i < =rear;i + + )
  {
    printf("\t%d",q[i]);
  }
  printf("\nrear is at %d\n",q[rear]);
  printf("\nfront is at %d\n",q[front]);
}
printf("\n");
}
getch();
```

5. Define priority queue. Write the c implementation of priority queue.

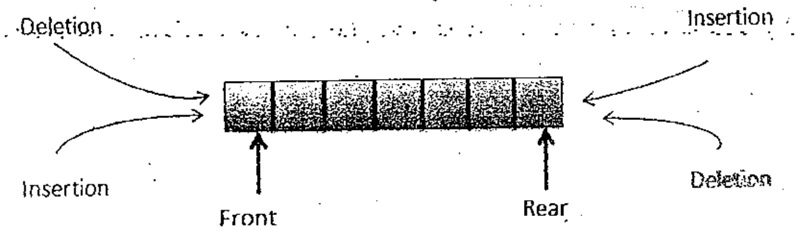
Ans. Refer Q5 (5 Marks)

```
#define SIZE 5          /* Size of Queue */
int f=0,r=-1;       /* Global declarations */
typedef struct PRQ
{
  int ele;
  int pr;
}PriorityQ;
PriorityQ PQ[SIZE];
PQinsert(int elem, int pre)
{
  int i;    /* Function for Insert operation */
  if(Qfull()) printf("\n\n Overflow!!!!\n\n");
  else
  {
    i=r;
    ++r;
    while(PQ[i].pr <= pre && i >= 0) /* Find location for new elem */
    {
      PQ[i + 1]=PQ[i];
      i--;
    }
    PQ[i + 1].ele=elem;
    PQ[i + 1].pr=pre;
  }
}
PriorityQ PQdelete()
{             /* Function for Delete operation */
  PriorityQ p;
  if(Qempty()){ printf("\n\nUnderflow!!!!\n\n");
  p.ele=-1;p.pr=-1;
  return(p); }
  else
  {
    p=PQ[f];
    f=f + 1;
    return(p);
  }
```

```
}
int Qfull()
{            /* Function to Check Queue Full */
  if(r==SIZE-1) return 1;
  return 0;
}
int Qempty()
{            /* Function to Check Queue Empty */
  if(f > r) return 1;
  return 0;
}
display()
{            /* Function to display status of Queue */
  int i;
  if(Qempty()) printf(" \n Empty Queue\n");
  else
  {
    printf("Front->");
    for(i=f;i <=r;i ++ )
      printf("[%d,%d] ",PQ[i].ele,PQ[i].pr);
    printf(" <-Rear");
  }
}
main()
{            /* Main Program */
  int opn;
  PriorityQ p;
  do
  {
    clrscr();
    printf("\n ### Priority Queue Operations(DSC order) ### \n\n");
    printf("\n Press 1-Insert, 2-Delete,3-Display,4-Exit\n");
    printf("\n Your option ? ");
    scanf("%d",&opn);
    switch(opn)
    {
    case 1: printf("\n\nRead the element and its Priority?");
       scanf("%d%d",&p.ele,&p.pr);
       PQinsert(p.ele,p.pr); break;
    case 2: p=PQdelete();
       if(p.ele != -1)
          printf("\n\nDeleted Element is %d \n",p.ele);
       break;
    case 3: printf("\n\nStatus of Queue\n\n");
       display(); break;
    case 4: printf("\n\n Terminating \n\n"); break;
    default: printf("\n\nInvalid Option !!! Try Again !! \n\n");
       break;
    }
    printf("\n\n\n\n  Press a Key to Continue . . . ");
    getch();
  }while(opn != 4);
}
```

6. Define double ended queue. Write c implementation of double ended queue.

Ans.



```
#include < stdio.h>
#include < stdlib.h>
#include < conio.h>
#define SIZE 100
```

```
int queue[SIZE];
int F = -1;
int R = -1;
void insert_r(int x)
{
if(F == (R + 1)%SIZE)
{
printf("\nQueue Overflow");
}
else if(R == -1)
{
F = 0;
R = 0;
queue[R] = x;
}
else
{
R = (R + 1) %SIZE;
queue[R] = x;
}
}
void insert_f(int x)
{
if(F == (R + 1)%SIZE)
{
printf("\nQueue Overflow");
}
else if(R == -1)
{
F = 0;
R = 0;
queue[R] = x;
}
```

```
else
{
F = (F + SIZE-1) %SIZE;
queue[F] = x;
}
}
int delete_r()
{
int x;
if(F == -1)
{
printf("\nQueue Underflow");
}
else if(F == R)
{
x = queue[F];
F = -1;
R = -1;
}
else
{
x = queue[R];
R = (R + SIZE-1)%SIZE;
}
return x;
}
int delete_f()
{
int x;
if(F == -1)
{
printf("\nQueue Underflow");
}
```

```
else if(F == R)
{
x = queue[F];
F = -1;
R = -1;
}
else
{
x = queue[F];
F = (F + 1)%SIZE;
}
return x;
}

void main()
{
char choice;
int x;
while(1)
{
system("cls");
printf("1: Insert From Front\n");
printf("2: Insert From Rear\n");
printf("3: Delete From Front\n");
printf("4: Delete From Rear\n");
printf("5: Exit Program\n");
printf("Enter Your Choice:");
choice = getche();
switch(choice)
{
case '1':
printf("\nEnter Integer Data :");
scanf("%d",&x);
insert_f(x);
break;
case '2':
printf("\nEnter Integer Data :");
scanf("%d",&x);
insert_r(x);
break;
case '3':
printf("\nDeleted Data From Front End: %d",delete_f());
break;
case '4':
printf("\nDeleted Data From Back End: %d",delete_r());
break;
case '5':
exit(0);
break;
}
system("pause");
}
}
```

UNIT 5

TREE

SYLLABUS

Introduction, Basic concept, Binary tree, Binary tree representation, Binary tree traversal.

SYNOPSIS

Tree is a non-linear data structure which is used to represent hierarchical relationship between several elements. Tree contains data in structures called nodes, which are in turn linked to other nodes in the tree. Every tree has a primary node called a root from which all other branch nodes in the tree descend and these branch nodes are termed as subtrees. The nodes that have no descendants are called leaf nodes.

1. Define binary tree. Explain the method of representing binary trees.

Binary Tree: A binary tree T is a finite set of nodes such that

(i) 'T' is empty (without a single node called empty binary tree)

(ii) 'T' contains a specially designated node called root node.

(iii) The remaining nodes of the tree 'T' form only two subtrees T_1 and T_2 as left subtree and right subtree respectively. In a binary tree each node can have maximum of two child nodes i.e., A node can have either zero child or 1 child or 2 child but not more than that.

Representation of Binary Trees:

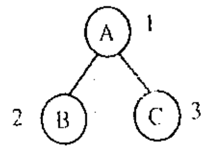
Binary trees can be represented in 2 ways

(i) Sequential representation

(ii) Linked list representation

Sequential representation:

In sequential representation, an array can be used to store the nodes of binary trees. A sequential representation of binary tree requires numbering of nodes starting with nodes on level 0, then on level 1 and so on. Consider a binary tree T shown below



The root node is numbered as 1, then its left child is numbered as 2 and the right child as 3. A is the parent node and B and C are the children of A. The above binary tree can be represented using array as shown below.

A[3] =

A[0]	A[1]	A[2]
A	B	C
1	2	3

Linked list representation:

In this representation, each node is divided in to 3 parts

(i) Info field → used to store the information of the node

(ii) Left link → which contains the address of its left node

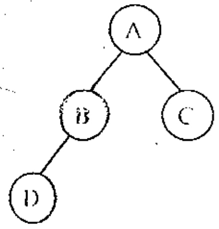
(iii) Right link → which contains the address of its right node.

Left Link	Info	Right Link

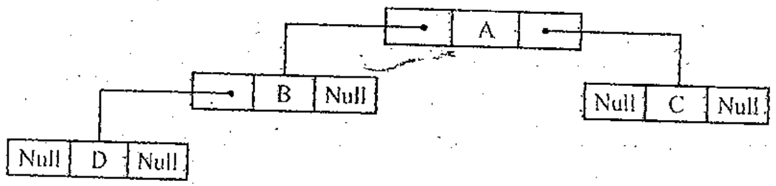
The logical representation of the above node in 'C' is shown below.

```
struct node
{
int info;
struct  node *left;
struct  node *right;
typedef struct node NODE;
```

Consider a binary tree T as shown below



The linked list representation is shown below

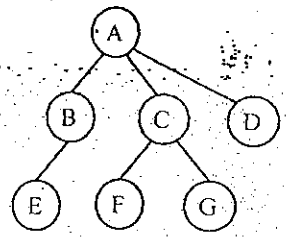


2. Define the following

(a) Internal node (b) Sibling (c) Degree of the tree (d) Depth of a tree (e) Path

Internal Node: The nodes except the leaf node in a tree are called internal nodes

Ex:



B and C are the internal nodes.

Sibling: Two or more nodes having the same parent are called siblings.

Ex: F and G are the siblings since they have the same parent C.

Degree of the tree: The maximum number of children that is possible for a node is known as degree of a node.

or

The number of subtrees of a node is called degree.

Ex: In the above example, the degree of **node A is three.**

Depth of a tree: The depth or height of the tree is the maximum number of nodes that is possible in a path starting from root node to a leaf node.

Ex: In the above example, longest path is A-B-E or A-C-F or A-C-G. So, the depth of the trees is 3.

Path: Sequence of consecutive edges is called a path.

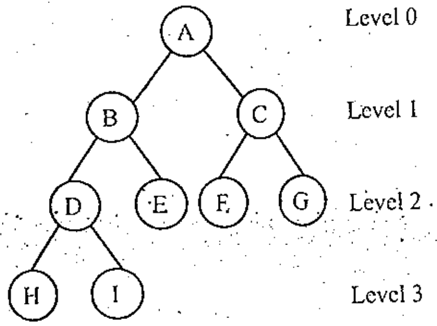
Ex: In the above example, the path from root node A to F is A-C-F and he length of this path is 3

3. Define the following

(a) Root node (b) Leaf node (c) Level of tree (d) Child node (e) Parent node.

Root node: A node that does not have a parent is termed as root node.

Consider a binary tree as



Ex: A is the root node.

Leaf node: A node in a tree that has an out degrees zero is called as leaf node.

Ex: H, I, E, F and G are leaf nodes.

Level of tree: The distance of a node from the root is called level.

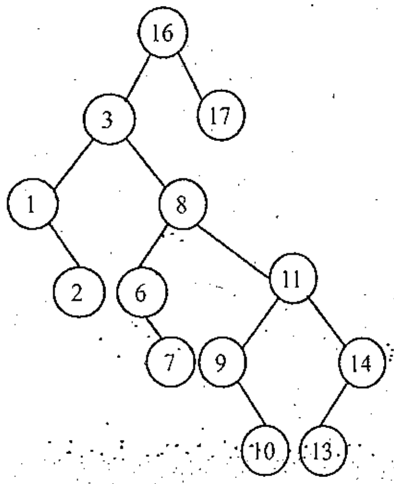
Ex: The level of node G is 2.

Child node: The nodes which are all reachable from root node using only one edge are called as child node. **Ex:** D and E are the child nodes of B

Parent node: A node having right subtree or left subtree or both is said to be parent node.

Ex: C is the parent node of F and G.

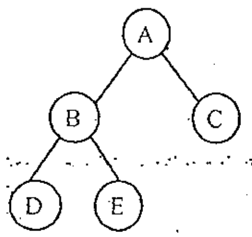
4. Construct a binary tree for the following data 16, 3, 8, 11, 1, 6, 9, 14, 2, 10, 17, 7, 13



5. Explain strictly binary tree and complete binary tree with an example.

Strictly binary tree: It is a binary tree that has non-empty left and right subtrees, the out degree of every node is either 0 or 2. Every node in strictly binary tree must have maximum 2 child or no children.

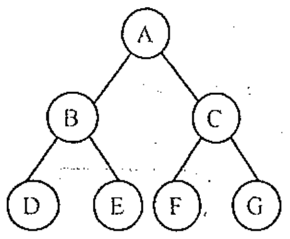
Ex:



Complete binary tree:

It is a binary tree in which every node should have exactly two child nodes and the number of nodes at level is 'n' is 2^n

Ex:



Level 0 = 2^0 = 1 node

Level 1 = 2^1 = 2 nodes

Level 2 = 2^2 = 4 nodes

6. What is binary tree traversing? Explain the type of traversal with example.

Ans. Tree traversing is the most common operation performed on trees. Traversal means processing or visiting each node in a tree exactly once in symmetrical manner one after the other.

The different traversal techniques are

1. Inorder traversal
2. Preorder traversal
3. Postorder traversal

Inorder:

It can be defined as

(i) Process the left subtree in inorder

(ii) Process the root node

(iii) Process the right subtree in inorder

Algorithm

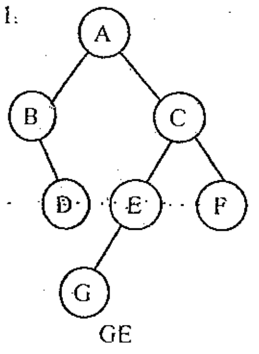
```
Step 1. [Check the empty tree]
if root = NULL then
display "Empty tree"
return
Step 2. [Traverse the left subtree recursively in inorder]
if left [root]! = NULL then
call inorder (Left[root])
Step 3. [Process the root node]
if root ! = NULL then
```

```
process info [root]
Step 4. [Traverse the right subtree recursively in inorder]
if right [root]! = NULL then
call inorder (right [root])
Step 5. Return
```

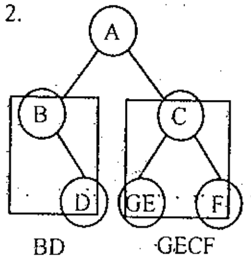
Function:

```
Void inorder (NODE root)
{
if (root ! = NULL)
{
 Inorder (root → l child);
 Printf ("%C", root → info);
 Inorder (root → r child);
}
}
```

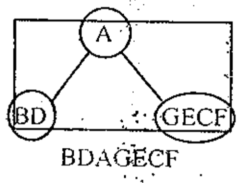
Ex: 1.



2.



3.



The inorder for tree is BDAGECF

Preorder traversal: It can be defined as

(i) Process the root node

(ii) Process the left subtree in preorder

(iii) Process the right subtree in preorder

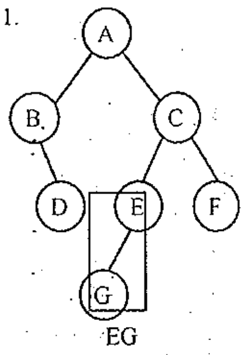
Algorithm:

```
Step 1: [Check the empty tree]
if root = NULL then
display "empty tree"
return
Step 2: [Process the root node]
if root 1/2 = NULL then
process info [root]
Step 3: [Traverse the left subtree recursively in preorder]
if left [root]! = NULL then
call preorder (left[root])
Step 4: [Traverse the right subtree recursively in preorder]
if right [root] ! = NULL then
call preorder (right [root])
Step 5: Return:
```

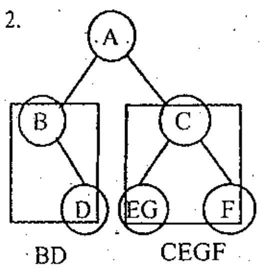
Function:

```
  Void preorder (NODE root)
{
 if (root ! = NULL)
 {
 Printf ("% c", root → info);
 Preorder (root → l child);
 Preorder (root → r child);
  }
}
```

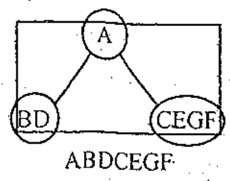
Ex: 1.



2.



3.



The preorder of the tree is ABDCEGF

Postorder traversal: It can be defined as

(i) Process the left subtree in postorder

(ii) Process the right subtree in postorder

(iii) Process the root node

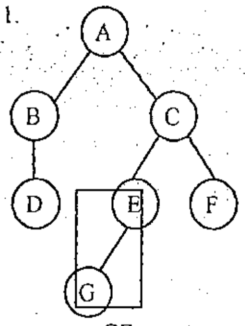
Algorithm:

```
Step 1. [Check the empty tree]
if root = = NULL then
display "empty tree'
return
Step 2. [Traverse the left subtree recursively]
if left [root] ! = NULL then
call postorder (left [root])
Step 3. [Traverse the right subtree recursively in postorder]
if right [root] ! = NULL then
call postorder (right [root])
Step 4. [Process the root node]
if root ! = NULL then
process info [root]
Step 5. return
```

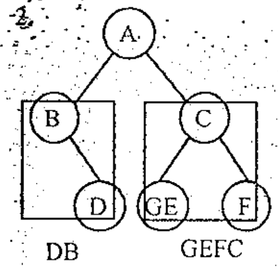
Function:

```
Void postorder (NODE root)
{
 if (root ! = NULL)
  {
  Postorder (root → l child)
  Portorder (root → r child);
  Printf ("%C", root → info);
  }
}
```

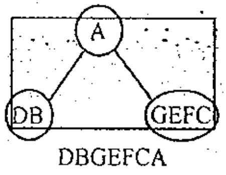
Ex: 1.



2.

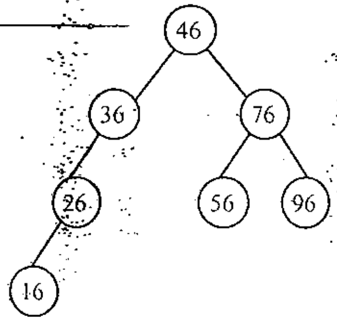


3.



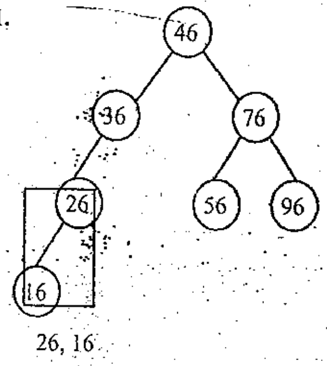
The postorder of the tree is DBGEFCA.

7. Construct a binary tree for the following values and traverse the tree in preorder, inorder and postorder. 46, 76, 36, 26, 16, 56, 96

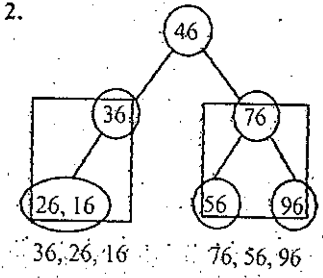


Preorder:

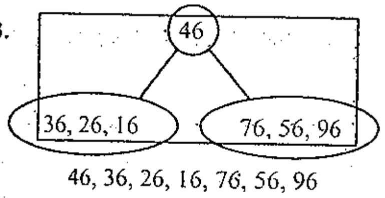
1.



2.



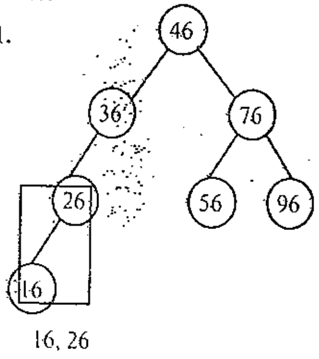
3.



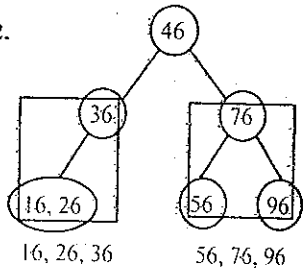
The preorder is 46, 36, 26, 16, 76, 56, 96

Inorder:

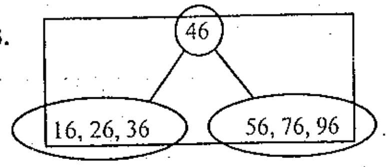
1.



2.



3.

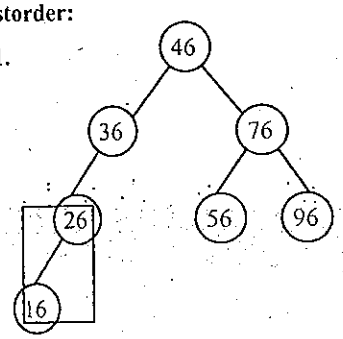


16, 26, 36, 46, 56, 76, 96

The inorder is 16, 26, 36, 46, 56, 76, 96

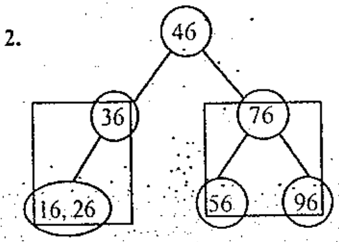
Postorder:

1.



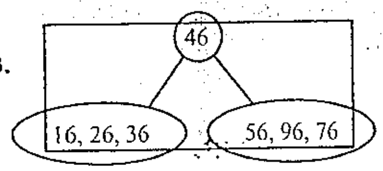
16, 26

2.



16, 26, 36 56, 96, 76

3.



16, 26, 36, 56, 96, 76, 46

The postorder is 16, 26, 36, 56, 96, 76, 46

UNIT 6

SORTING, SEARCHING AND APPLICATION OF DATA STRUCTURES

SYLLABUS

Sorting – Introduction, sorting techniques – selection sort, insertion sort, bubble sort, quick sort (no implementation), merge sort (no implementation).

Searching – Introduction, Linear search, binary search.

Application of data structure – Introduction, Applications of stack, Infix to postfix conversion, Evaluation of a postfix expression, Recursion, factorial, GCD, List application of queues, linked lists and trees.

SYNOPSIS

Sorting is a process of arranging the data elements in sequential order that can be either in ascending or descending order with numerical data and it can be alphabetical order with character data.

The different types of sorting techniques are

1. Bubble sort
2. Selection sort
3. Insertion sort
4. Quick sort
5. Merge sort

Searching is a process of finding matching element or number among a collection of huge data by a method of successive comparison for equality.

The basic searching techniques are

1. Linear or sequential searching
2. Binary searching.

1. Explain with an example the working of merge sort.

Ans. The process of combining two or more unsorted array into a single sorted array is called as merging when merging is applied to a single unsorted array then the procedure is called as merge sort.

Working Procedure:

1. In this method, the given array of size 'n' is divided into two subarray.

i.e., a[0]......a[n/2] and a[n/2 + 1]...... a[n]

2. These subarray are recursively divided into smaller subarray until each subarray is smaller enough to be solved individually without splitting

3. Once the division process is completed, each subarray is individually sorted.

4. Finally all the sorted subarray are combined to produce a single sorted array of 'n' elements.

Ex: Consider an array of elements such as 35, 10, 15, 42, 25, 20, **40**.

The array is divided into two parts based on the mid value. If low is the index of first element and high is the index of last element then

mid = (Low + high)/2

∴ The left part contains the element from low to mid and right part contains the elements from (mid + 1) to high.

0	1	2	3	4	5	6
35	10	15	42	25	20	40

↓ Low ↓ High

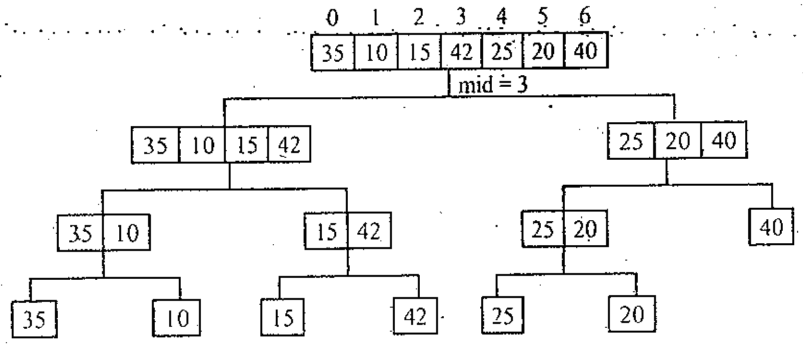
Low = 0

high = 6

∴ mid = (low + high)/2

mid = (0 + 6)/2

mid = 3



Merge these subarray by comparing the elements individually.

(i) Compare 35 and 10. Since 10 is less than 35 interchange a[0] and a[1] and merge

a[0]	a[1]
10	35

(ii) Compare 15 and 42. Since they are in order merge them without exchanging

a[2]	a[3]
15	42

(iii) **Compare a[0:1]** and a [2:3] and merge

a[0]	a[1]	a[2]	a[3]
10	15	35	42

The left subarray is sorted by merging two subarray.

(iv) Compare 25 and 20. Since 20 is less than 25 interchange a[4] and a[5] and merge.

a[4]	a[5]
20	25

(v) Compare a[4:5] and a[6] and merge

a[4]	a[5]	a[6]
20	25	40

The right subarray is also sorted

(vi) Compare a[0:3] and a[4:6] and merge

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
10	15	20	25	35	40	42

2. Explain the concept of straight selection sort.

Ans. In this method an element is selected and placed in the proper position

Working procedure:

1. The smallest element in an array is searched and interchanged with first element

2. The second smallest element is searched between second and (n-1) element and exchanged with second element of the array.

3. Continue the same process until all the elements are arranged.

4. It requires (n-1) passes to sort all the elements in an order.

Ex: Consider the following array of elements

a[0]	a[1]	a[2]	a[3]	a[4]
38	47	24	42	17

(i) Pass -1 → The smallest element is 17 which is in the position 4 and exchange it with the first element 38

a[0]	a[1]	a[2]	a[3]	a[4]
17	47	24	42	38

(ii) Pass -2 → Search for the next smallest element from the location 1. The smallest element within the range is 24 which is in location 2. Exchange it with 47.

a[0]	a[1]	a[2]	a[3]	a[4]
17	24	47	42	38

(iii) Pass -3 → Search for the next smallest element from the location 2. The smallest element within the range is 38 which is in the location 4. Exchange it with third element 47.

a[0]	a[1]	a[2]	a[3]	a[4]
17	24	38	42	47

(iv) Pass - 4 → The next smallest element is 42 and it is compared with 47. Since 42 is less than 47, no exchange is required

a[0]	a[1]	a[2]	a[3]	a[4]
17	24	38	42	47

Thus the array elements are sorted.

3. Write a note on simple insertion sort.

Ans. The insertion sort is efficient only when the numbers to be sorted are very less. The insertion sort inserts each element in appropriate position.

Working Procedure:

1. Assume that the first element is in proper position
2. Take the next element from unordered array and place it in proper position by comparing it with element of the sorted list
3. This process is continued until all the elements are sorted.

Ex: Consider the array elements as

38, 47, 24, 42, 84

In the first pass, first element is compared with 0th element. In the second pass, second element is compared with 0th and 1st element. In general, in every pass element is compared with all the elements before it.

Ex:

0	1	2	3	4

(i) Pass -1 → The first element 47 is compared with 0th element 38. Since they are in order, no exchange is required.

0	1	2	3	4

(ii) Pass - 2 → The second element 24 is compared with 1st element 47. Since 24 < 47, exchange them

0	1	2	3	4

Again 24 is compared with 38. Since 24 < 38, exchange them.

1	2	3	4	5

(iii) Pass - 3 → The third element 42 is compared with 2nd element 47. Since 42 < 47, exchange them

1	2	3	4	5

Again 42 is compared with 38 and 24. Since they are in order, no exchange is required.

(iv) Pass - 4 → 84 is compared with 24, 38, 42, 47. Since 84 is greater than all the elements, no exchange is required.

The final, sorted array is

1	2	3	4	5

4. Explain quick sort with example.

Ans. Quick sort is one of the best technique for large set of data. It works by partitioning the array. Partition is done at the position such that all the element of the left partition are less than the elements of the right partition. This division of the list and sublist continues until the sorting is completed.

Working Procedure:

(i) Assume the first element of the list as Key element or Pivot element

(ii) Remaining (n-1) elements are compared with key element to place it in proper position.

(iii) It uses two pointers i and j. Pointer i points to the element after the key element and pointer j points to the last element.

(iv) Key is compared with the elements starting from a[i] until the greater element than the key element is encountered

(v) At this point stop incrementing i.

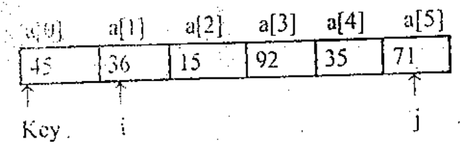
(vi) Then compare the key element with a[j] until the lesser element than the key element is encountered. At this point stop decrementing j.

(vii) if the position of i is less than j, swap a [i] and a[j] and continue the same process.

(viii) if the position of i is greater than j then swap a [j] and key element.

(ix) Once the key element has been swapped it occupies its original position and continue the same steps for left partition and right partition, to sort the element.

Ex: Consider an array of 6 elements

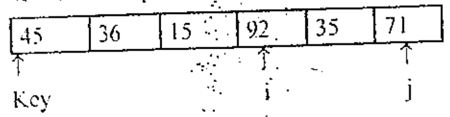


Compare key with a [i]

45 > 36 → increment i

45 > 15 → increment i

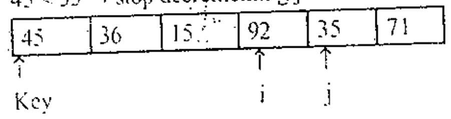
45 > 92 → stop incrementing i



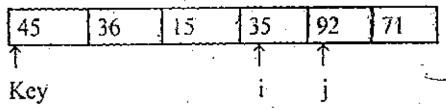
Compare key with a[j]

45 < 71 → decrement j

45 < 35 → stop decrementing j



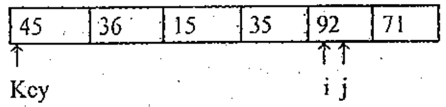
Compare the position of i and j. Since the position of i is less than j then swap a[i] and a [j]



Compare key with a[i]

45 > 35 increment i

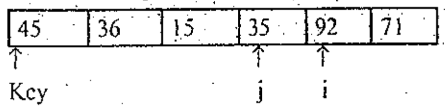
45 > 92 stop incrementing i



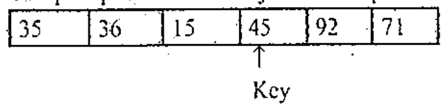
compare key with a[j]

45 < 92 → decrement j

45 < 35 → stop decrementing j



Compare position of i and j. Since the position of i is greater than j then swap key and a [j]



Now the key element has occupied its original position and apply the same process for the element i.e., left of the key element and also for right of the key element. The two partitions of the list ar

35	36	15

and

92	71

5. Explain bubble sort with an example.

Ans. It is the most commonly used sorting technique. It is easy to understand and implement

* In this method the smallest data element will move upwards or bubble up in each subseque pass. So it is called as bubble sort.

* During every pass, each element is compared with all the remaining element. If the element not in proper order then they are interchanged.

* At the end of the first pass, largest element will occupy the last position

* Apply the same procedure, compare the elements from 1 to (n-1) because n^th element is proper positioned which need not be compared.

Ex: Consider an array of 5 elements

40	50	30	20	10
a[0]	a[1]	a[2]	a[3]	a[4]

Comparing or interchanging the data element in each pass is illustrated below.

Pass 1

```
40   40   40   40   40
50   50   30   30   30
30   30   50   20   20
20   20   20   50   10
10   10   10   10   50
```

Pass 2

```
40   30   30   30
30   40   20   20
20   20   40   10
10   10   10   40
50   50   50   50
```

Pass 3

```
30   20   20
20   30   10
10   10   30
40   40   40
50   50   50
```

Pass 4

```
20   10
10   20
30   30
40   40
50   50
```

Number of elements = 5

Number of passes required = (n-1)

6. Develop a 'c' program to implement bubble sort method.

Ans.

```
 # include  < stdio.hs>
 # include < conio.h>
 Void main ()
 {
  int n, i, j, temp, a [10];
  clrscr ();
  Printf ("enter the size of an array\n");
  Scanf ("%d", &n);
  Printf ("enter the array elements\n");
  for (i = 0; i < n; i++)
  Scanf ("%d", & a [i]);
  for (j=1 ; j < n; j + + )
 {
  for (i=0 ; i < (n-j) ; i +  + )
 {
  if (a[i] > a [i + 1])
{
  temp = a [i];
 a[i] = a[i + 1];
 a [i + 1] = temp;
}
 }
}
Printf ("After sorting\n");
for (i=o; i  < n; i + + )
Printf ("%d", a [i]);
getch();
}
```

7. Write a program to implement Insertion sort.

Ans.

```
  # include  < stdio.h>
  # include  < conio.h>
  void insertion (int n, int a [ ])
 {
  int i, j, item;
 for (i=1 ; i ≤ n-1 ; i + + )
 {
  item = a [i] ;
  j = i - 1 ;
 while ((item  < a[j]) & & (j ≥ 0))
 {
  a[j + 1] = a[j];
  j=j-1;
 }
  a [j + 1] = item;
 }
 }
 void main ()
 {
   int a[10], n, i;
   clrscr () ;
 Printf (" enter the size of array\n");
 Scanf ("%d", & n);
 Printf (" enter the array elements\n");
   for (i=0; i < n ; i + + )
 scanf ("%d", &a[i]);
   insertion (n, a) ;
  Printf ("After sorting \n";
```

```
   for (i=0; i < n; i +   + )
  Printf ("% d", a[i]);
  getch ();
 }
```

8. WAP to implement selection sort.

Ans.

```
  # include  < stdio.h>
  # include  < conio.h>
  void main ()
  {
   int n, i, j, temp, a[10], pos;
   clrscr () ;
 Printf ("enter the size of array\n");
 scanf ("% d", & n);
 Printf (" enter the array elements\n");
   for (i=0; i < n; i + + )
 scanf ("% d", & a[i]);
 for (i=0; i < n-1; i + + )
  {
  Pos = i;
 for (j=i + 1; j  < n; j + + )
 {
  if (a[j] < a[pos])
  pos = j;
 }
  temp = a[pos];
  a[pos] = a[i];
   a[i] = temp;
 }
 Printf (" sorted array is \n");
```

```
   for (i=0; i < n; i + + )
 Printf ("% d\n", a[i]);
   getch () ;
}
```

9. Write a program to implement linear search.

Ans.

```
  # include  < stdio.h>
  # include  < conio.h>
  void main ()
 {
   int a[20], i, n, pos, key;
  clrscr ();
 Printf (" enter the array size \n");
 scanf ("% d", &n);
 Printf (" enter the array elements\n");
   for (i=0 ; i < n ; i + + )
 scanf ("% d", & a[i]);
 Printf (" enter the key elements\n");
 scanf ("% d", & key);
   Pos = linearsearch (a, key, n);
 if (pos = = - 1)
 {
Printf (" number not present \n");
exit (0);
 }
 else
 Printf (" Number is present \n");
 getch ();
 }
int linearsearch (int a [ ], int key, int n)
```

```
{
  int i;
for (i = 0; i < n; i + + )
{
  if (key = = a[i])
 {
   return    i ;
 }
   return  -i;
  }
}
```

10. Write a program to implement Binary search.

Ans.

```
  # include  < stdio.h>
  # include  < conio.h>
 int bsearch (int a [ ], int key, int low int high)
 {
   int mid;
   if (low > high)
     return   -1;
     else
    {
      mid = (low + high) /2;
    }
    if (key = = a [mid])
       return (mid);
    if (key < a [mid])
   {
     bsearch (a, key, low, mid - 1);
```

```
        else
        bscarch (a, key, mid + 1, high);
      }
    }
    void main 1)
    {
      int i, n, pos, key, a[10];
      clrscr ();
    Printf (" enter the array size \n");
     Scanf ("% d", & n);
  Printf (" enter the array elements\n");
     for (i=0; i < n; i + + )
    scanf (" % d", & a[i]);
  Printf (" enter the key element \n");
     scan f ("%d", & key);
   Pos = bscarch (a, key, 0, n-1);
     if (pos = = -1)
  Printf ("element not found \n");
  else
   Printf ("element found at % d location\n", pos);
      getch ();
    }
```

11. Explain the concept of binary search.

Ans. It is one of the best suitable searching technique for large sized array.

Working procedure:

* In this method the element in the array must be arranged either in ascending or descending order.

* The sorted array is divided into two part based on the middle value.

i.e., mid = (low + high)/2

* If the key element is equal to mid element then search is success full.

* If the key element is greater than mid element then right half is searched

i.e., mid + 1 to high.

* If the key element is less than mid element then left half is searched.

i.e., low to mid -1

* Repeat the steps from 2 to 5 with either half part to search the key element.

Ex: Consider the unordered array

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
2	6	1	8	4	5	10	7

Consider 8 as key element that is to be searched.

* The given array must be sorted to apply binary search.

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
1	2	4	5	6	7	8	10

low = 0

high = 7

* Divide the array into two parts by calculating mid value.

mid = (low + high)/2

mid = (0 + 7)/2

mid = 3.5

mid = 4

* Compare key with mid element which is in the position a [4] i.e., 6

* Since 8 is greater than 6, then right half is searched i.e., from x [5] to x [7].

* Calculate the mid value again for the right half and divide once again into two parts.

low = 5

high = 7

∴ mid = (low + high)/2

mid = (5 + 7)/2

mid = 12/2

mid = 6

* Compare key with a [mid] i.e., 8

* 8 = 8, so return the position 6.

12. Explain the concept of linear search.

Ans. It is also called as sequential search. It is the simplest method of searching the element from an unordered list of elements. In this technique, the key element is compared with each item in the list sequentially one after the other until the end of the list. If the key is found, search is successful and then corresponding record is retained. If the position of the key is not found, then -1 is returned.

Function:

```
int linearsearch (int a[ ], int n, int key)
 {
   int i;
 for (i=0; i < n; i + + )
  {
   if (key = =a [ i ])
   {
  return i ;
 }
 return     -1 ;
  }
  }
```

13. List the applications of stack. Write an algorithm to convert infix to post fix expression

Ans. Applications of Stack:

(i) Recursive Function.

(ii) Evaluation of expression

(iii) Conversion of expression

(iv) Function calling

(v) Parsing

(vi) Tower of Hanoi

Algorithm to Convert infix to postfix expression

(i) Scan the input infix expression from left to right character by character.

(ii) Repeat through step 6 until ~~the end~~ of the postfix expression.

(iii) If the scanned symbol is left parenthesis, push it on the stack

(iv) If the scanned symbol is an operand, then place it in postfix expression.

(v) If the scanned symbol is right parenthesis, then go on popping the item from the stack and place it in postfix expression until the matching left parenthesis.

(vi) If the scanned symbol is an operator then compare the precedence of operator at the top of the stack (PT) with the precedence of scanned operator (PS).

(vii) If PT ≥ PS, then pop all the operator from the stack and place it in postfix expression.

(viii) If PT < PS, push the scanned operator onto the top of the stack

Ex: (a + b) * C

Symbol (PS)	Top of stack (PT)	Postfix expression
(	(	
a	(	a
+	(+	a
b	(+	ab
)		ab +
*	*	ab +
c	*	ab + c
Nil		ab + c*

14. Give the postfix and prefix forms for the following expression.

(a) (a/b) * c- (d + g) $ f

(b) a $ b * c - d + c|f| (g + h)

Ans. Postfix expression

(a)

$(a/b) * c - (d+g) \$ f$	$T_1 = a/b$ $= ab/$
$T_1 * c - \underbrace{(d+g)}_{T_2}\$ f$	$T_2 = d + g$ $= dg +$
$T_1 * c - \underbrace{T_2 \$ f}_{T_3}$	$T_3 = T_2 \$ f$ $= T_2 f \$$ $= dg + f \$$
$\underbrace{T_1 * c}_{T_4} - T_3$	$T_4 = T_1 * c$ $= T_1 c *$ $= ab/c *$
$\underbrace{T_4 - T_3}_{T_5}$	$T_5 = T_4 - T_3$ $= T_4 T_3 -$ $= ab/c * dg + f \$ -$

∴ The postfix expression is ab/c * dg + f $ -

Prefix Expression	
$\underbrace{(a/b)}_{T_1} * c - (d+g) \$ f$	$T_1 = a/b$ $= ab/ab$
$T_1 * c - \underbrace{(d+g)}_{T_2} \$ f$	$T_2 = d + g$ $= +dg$
$T_1 * c - \underbrace{T_2 \$ f}_{T_3}$	$T_3 = T_2 \$ f$ $= \$ T_2 f$ $= \$ + dg f$
$\underbrace{T_1 * c}_{T_4} - T_3$	$T_4 = T_1 * c$ $= * T_1 c$ $= * ab/c$
$\underbrace{T_4 - T_3}_{T_5}$	$T_5 = T_4 - T_3$ $= - T_4 T_3$ $= */abc \$ + dgf$

∴ The prefix expression is - * /abc $ + dgf

(b)

Prefix Expression	
$a \$ b * c - d + e/f/ \underbrace{(g+h)}_{T_1}$	$T_1 = g + h$ $= gh +$
$\underbrace{a\$b}_{T_2} * c - d + e/f/T_1$	$T_2 = a \$ b$ $= ab \$$
$\underbrace{T_2 * c}_{T_3} - d + e/f/T_1$	$T_3 = T_2 * c$ $= T_2 c *$ $= ab \$ c *$
$T_3 - d + \underbrace{e/f}_{T_4}/T_1$	$T_4 = e/f$ $= ef/$
$T_3 - d + \underbrace{T_4/T_1}_{T_5}$	$T_5 = T_4/T_1$ $= T_4 T_1 /$ $= ef/gh + /$
$\underbrace{T_3 - d}_{T_6} + T_5$	$T_6 = T_3 - d$ $= T_3 d -$ $= ab \$ c * d -$
$\underbrace{T_6 + T_5}_{T_7}$	$T7 = T6 + T5$ $= T6\ T5 +$ $= ab \$ c*d - ef/gh + / +$

∴ The postfix expression is ab $ c * d - ef /gh + / +

Prefix Expression	
$a \$ b * c - d + e/f/ \underbrace{(g+h)}_{T_1}$	$T_1 = g + h$ $= + gh$
$\underbrace{a\$b}_{T_2} * c - d + e/f/T_1$	$T_2 = a \$ b$ $= \$ ab$
$\underbrace{T_2 * c}_{T_3} - d + c/f/T_1$	$T_3 = T_2 * c$ $= * T_2 c$ $= * \$ abc$
$T_3 - d + \underbrace{e/f}_{T_4}\ T_1$	$T_4 = c/f$ $= /cf$
$T_3 - d + \underbrace{T_4/T_1}_{T_5}$	$T_5 = T_4/T_1$ $= /T_4 T_1$ $= //cf + gh$
$\underbrace{T_3 - d}_{T_6} + T_5$	$T_6 = T_3 - d$ $= - T_3 d$ $= - \$ abcd$
$\underbrace{T_6 + T_5}_{T_7}$	$T7 = T6 + T5$ $= + T6\ T5$ $= + - * \$ abcd//cf + gh$

∴ The prefix expression is

+ - * $ abcd//cf + gh

15. Write a recursive 'c' program to0 find the GCD of two numbers.

Ans.

```
# include < stdio.h>
# include  < conio.h>
 void main ()
      {
 int x, y, res;
 clrscr ();
      Printf ("enter the value of x and y\n");
      Scanf (" % d % d", & x, & y) ;
 res = gcd (x, y) ;
      Printf ("GCD of % d and % d = % \ n", x, y, res) ;
 getch () ;
}
int gcd (int a, int b)
{
  if (a = = b)
    return a ;
else if (a < b)
    return      GCD (b, a);
else
    return      (GCD (a - b, b));
}
```

16. Write a recursive 'c' program to find the factorial of a number.

Ans.

```
# include < stdio.h>
# include < conio.h>
 void main ()
 {
    int factorial ;
    clrscr () ;
    Printf (" enter the value of n\n") ;
    Scanf (" % d", & n) ;
    factorial = fact (n) ;
Printf (" Factorial = % d\n", factorial) ;
   getch () ;
}
 int fact (int n)
 {
    if (n = =0)
 return 1 :
```

```
   else
   return (n * fact (n-1)) ;
   }
```

17. List the applications of queues.

Ans. 1. One major application of queue is in simulation. The simulation is a modelling of a real life problem

2. Queues are used in time sharing systems in which programs form a queues while waiting to be executed.

3. Circular queues are used in Data operating systems.

4. Circular queues are also used in real time applications, which must continue process information while buffering I/O requests

5. Queues are used in network communication systems

18. List the applications of linked list.

Ans. 1. Arithmetic operations on long positive numbers.

2. Manipulation of 2 polynomials

3. Evaluation of polynomials

4. In symbol table construction (compiler design)

19. List the applications of trees.

Ans. 1. Searching

2. Sorting

3. Manipulation of arithmetic expressions

4. Constructing symbol table

5. Trees are also used in syntax analysis of compiler design and are used to display the structure of a sentence in a language.

20. Define recursion. Explain the properties of recursive definition.

Ans. Recursion: It is a process of calling **a function** repeatedly in terms of itself until a specified condition is reached. The properties of **recursive** definition are

1. A recursive function should never generate infinite sequence of calls on itself. An algorithm exhibiting this sequence of calls will never terminate.

2. There should be at least terminal condition which will not involve a call to the same function i.e., there must be a condition to stop recursion.

21. Evaluate the given postfix expression with the stack content. 3 + 4 * 2 / (9 - 5) ∧ 4

Convert the given infix expression to postfix expression

$3+4*2/\underbrace{(9-5)}_{T_1}\wedge 4$	$T_1 = 9\text{-}5$ $= 95\text{-}$
$3+4*2/\underbrace{T_1\wedge 4}_{T_2}$	$T_2 = T_1 \wedge 4$ $= T_1\, 4 \wedge$ $= 95 - 4 \wedge$
$3+\underbrace{4*2}_{T_3}/T_2$	$T_3 = 4 * 2$ $= 42 *$
$3+\underbrace{T_3/T_2}_{T_4}$	$T_4 = T_3/T_2$ $= T_3/2\|$ $= 42*95\text{-}4\wedge\|$
$\underbrace{3+T_4}_{T_5}$	$T_5 = 3 + T_4$ $= 3T_4 +$ $= 342*95\text{-}4\wedge/ +$

The postfix expression is

342 * 95 - 4∧/ +

Symbol	Operand 1	Operand 2	Result	Stack
3	-	-	-	3
4	-	-	-	3, 4
2	-	-	-	3, 4, 2
*	4	2	8	3, 8
9	-	-	-	3, 8, 9
5	-	-	-	3, 8, 9, 5
-	9	5	4	3, 8, 4
4	-	-	-	3, 8, 4, 4
∧	4	4	256	3, 8, 256
/	8	256	0	3. 0
+	3	0	3	3

The result is 3.

QUESTION PAPERS

DTE SUPER MODEL QUESTION PAPER (WITH ANSWERS)

Diploma in Computer science & Engineering

IV- Semester

Data Structures Using C

Time: 3 Hours **Max Marks: 100**

PART-A

Answer any SIX questions. Each carries 5 marks. **5 x 6 = 30 Marks**

1. Explain fseek() & ftell() functions.

Ans. Ref. Unit – I Q 5 (5 Marks)

2. Define data structures. Mention different types of data structures.

Ans. Ref. Unit – III Q 1 (5 Marks)

3. Write the advantages & disadvantages of linked list.

Ans. Ref. Unit – III Q 8 (5 Marks)

4. Define stack. Explain how to represent stack in C.

Ans. Ref. Unit – IV Q 1 (5 Marks)

5. Write a note on dequeue.

Ans. Ref. Unit – IV Q 4 (5 Marks)

6. Define the following:

(a) internal node (b) sibling (c) degree of a tree (d) depth of a tree (e) path

Ans. Ref. Unit – V Q 4 (5 Marks)

7. Define the following:

(a) root node (b) leaf node (c) level of a tree (d) child node (e) parent node

Ans. Ref. Unit – V Q 3 (5 Marks)

8. Write a recursive C program to find the GCD of two numbers

Ans. Ref. Unit – VI Q 8 (5 Marks)

9. List the application of linked list

Ans. Ref. Unit – VI Q 12(5 Marks)

PART-B

Answer any SEVEN full questions each carries 10 marks. **10 x 7 = 70 Marks**

1. (a) Define pointer. Write its advantages & disadvantages.

Ans. Ref. Unit – I Q 1 (5 Marks)

(b) Explain pointer to structures with example.

Ans. Ref. Unit – I Q 11 (5 Marks)

2. List & explain dynamic memory allocation functions in C

Ans. Ref. Unit – I Q 9 (10 Marks)

3. Write a program to copy contents of one file to another. Use command line arguments to specify the file names.

Ans. Ref. Unit – II Q 2 (10 Marks)

4. Write a C function to perform insert at front & delete operations on singly linked list.

Ans. Ref. Unit – III Q 2 (10 Marks)

5. Define circular linked list. Give its C representation.

Ans. Ref. Unit – III Q 4 (10 Marks)

6. Write a C program to implement push & pop operation on stack.

Ans. Ref. Unit – IV Q 1 (10 Marks)

7. Define priority queue. Write the C implementation of priority queue.

Ans. Ref. Unit – IV Q 6 (10 Marks)

8. Construct a binary tree for the following values & traverse the tree in preorder, inorder & postorder. 46, 76, 36, 26, 16, 56, 96.

Ans. Ref. Unit – V Q 3 (10 Marks)

9. Write a C program to implement binary search.

Ans. Ref. Unit – VI Q 5 (10 Marks)

10. List the application of stack. Write an algorithm to convert infix to postfix expression.

Ans. Ref. Unit – VI Q 6 (10 Marks)

SUPER MODEL QUESTION PAPER (WITH ANSWERS)

Diploma in Computer science & Engineering

IV- Semester

Data Structures Using C

Time: 3 Hours **Max Marks: 100**

PART-A

Answer any SIX questions. Each carries 5 marks. **5 x 6 = 30 Marks**

1. Give the difference between call by value & call by difference method.

Ans. Ref. Unit – I Q 5 (5 Marks)

2. Explain how to handle errors during I/O operations.

Ans. Ref. Unit – II Q 3 (5 Marks)

3. Define linked list. Mention the different types of linked list.

Ans. Ref. Unit – III Q 5 (5 Marks)

4. Write a note on queue.

Ans. Ref. Unit – IV Q 4 (5 Marks)

5. Define binary tree. list the methods of representing binary trees.

Ans. Ref. Unit – V Q 1 (5 Marks)

6. With an example explain how to perform deletion operation on binary tree.

Ans. Ref. Unit – V Q 2 (5 Marks)

7. Write a recursive program to find factorial of a number.

Ans. Ref. Unit – VI Q 9 (5 Marks)

8. Explain the concept of bubble sort with an example.

Ans. Ref. Unit – VI Q 5 (5 Marks)

9. Explain the concept of linear search with an example.

Ans. Ref. Unit – VI Q 6 (5 Marks)

PART-B

Answer any SEVEN full questions each carries 10 marks. **10 x 7 = 70 Marks**

1. Write a C program to illustrate pointer arithmetic.

Ans. Ref. Unit – I Q 5 (10 Marks)

2. Explain different file accessing modes.

Ans. Ref. Unit – II Q 5 (10 Marks)

3. Define doubly linked list . give its C representation.

Ans. Ref. Unit – III Q 5 (10 Marks)

4. Define circular queue. Write the C implementation of circular queue.

Ans. Ref. Unit - IV Q 5 (10 Marks)

5. (a) explain push & pop operations of stack

Ans. Ref. Unit – IV Q 2(5 Marks)

(b) Define Priority queue. Differentiate ascending & descending priority queue.

Ans. Ref. Unit – IV Q 5(5 Marks)

6. (a) Explain strictly binary tree & complete binary tree with an example.

Ans. Ref. Unit – V Q. 6 (5 Marks)

(b) Explain perfect binary tree & balanced binary tree with an example.

Ans. Ref. Unit – V Q 7 (5 Marks)

7. Develop a recursive algorithm to traverse a binary tree in inorder, preorder & postorder.

Ans. Ref. Unit – V Q 2 (10 Marks)

8. Evaluate the given postfix expression with the stack content. 3+4*2/(9-5)^4

Ans. Ref. Unit – VI Q 9 (10 Marks)

9. Write a program to implement selection sort.

Ans. Ref. Unit – VI Q 3 (10 Marks)

10. Write an algorithm to convert infix to prefix expression.

Infix to Prefix Conversion

Algorithm of Infix to Prefix

```
1. Step 1. Push ")" onto STACK, and add "(" to end of the A
2. Step 2. Scan A from right to left and repeat step 3 to 6 for each element of A until the STACK is empty
```

```
3. Step 3. If an operand is encountered add it to B
4. Step 4. If a right parenthesis is encountered push it onto STACK
5. Step 5. If an operator is encountered then:
6. (a) Repeatedly pop from STACK and add to B each operator (on the top of STACK) which has same
7. or higher precedence than the operator.
8. (b) Add operator to STACK
9. Step 6. If left parenthesis is encountered then
10. (a) Repeatedly pop from the STACK and add to B (each operator on top of stack until a left parenthesis is encountered
11. (b) Remove the left parenthesis
12. Step 7. Exit
```

SUPER MODEL PRACTICE QUESTION PAPER

Diploma in Computer science & Engineering

IV- Semester

Data Structures Using C

Time: 3 Hours **Max Marks: 100**

PART-A

Answer any SIX questions. Each carries 5 marks. **5 x 6 = 30 Marks**

1. Explain the declaration & initialization of pointer variable with an example.

2. What is a file? Explain how to open & close a file.

3. Explain the representation of linked list in memory with the help of an illustration.

4. What is stack? Explain operations on stack.

5. Write a note on priority queue.

6. Define binary tree. List the different types.

7. List the applications of queue.

8. Explain the concept of binary search.

9. Explain the concept of quick sort.

PART-B

Answer any SEVEN full questions each carries 10 marks. 10 x 7 = 70 Marks

1. (a) Explain free(). What are its advantages.

(b) write a C program to illustrate malloc() function.

2. Write a C function to perform insert & delete a node at a given position in singly linked list.

3. Define queue. Explain different types of queues.

4. Write a C program to implement stack

5. (a) Distinguish between linear & non linear data structures.

(b) explain data structure operations.

6. List & explain input output functions of file.

7. Construct binary tree for the following data & traverse the tree in preorder, inorder & postorder.

58,68,36,38,58,26,17,88,89,34,67.

8. Give the postfix & prefix form of the following expression.

A$B*C-D/(G+H)+F/(G-H)

10. Develop a program to implement bubble sort.

11. Write a C program to implement selection sort.