PAPER 4

1. (a) Explain the following and illustrate it with an example each.

(i) Increment and decrement operator

(ii) Conditional operator

(iii) Bitwise operator

(iv) Assignment operator

(i) Increment and decrement operator

The increment operator is represented like ++. This operator is to add 1 to the value of the associated variable. For example, ++m means m = m + 1. There are two forms of this operator. These are ++m and m++. The difference is better understood with the following example:

```
m = 10;
 n = ++m; // n will get 11 and m will also be 11
m = 10;
 n = m++; // n will get 10 but after this assignment, m = 11
```

The decrement operator is represented like - -. This operator will subtract 1 from the value of the associated variable. For example, - -m means m = m – 1. There are two forms of this operator. These are —m and m—. The difference is better understood with the following example:

```
m = 10;
 n = - -m; // n will get 0 and m will also be 9
m = 10;
 n = m- -; // n will get 10 but after this assignment, m = 9
```

(ii) Conditional operator

A ternary operator pair “?:” is available in C to construct conditional expressions of the form:

expr 1 ? expr2 : expr3;

Here, expr1 is evaluated first. If it is nonzero (or true), then expr2 is evaluated and this becomes the result of the expression. However, if expr1 is false, expr3 is evaluated and it becomes the value of the expression. Look at the example below:

a = 10;

b = 15;

c = (a > b) ? a : b;

In the able "a > b" is evaluated. If it is true, c will get the value stored in a, otherwise c will get the value of b. In the above, since a < b, c will get the value of b.

(iii) Bitwise operator

These are special operators, which can be used for manipulation of data at a bot level. These operators are used for testing the bits, or shifting them left or right. They can be applied only on integers. The various bitwise operators are:

Operator	Meaning
&	Bitwise AND
|	Bitwise OR
^	Bitwise NOT
<<	Left shift
>>	Right shift
~	1's complement

(iv) Assignment operator

They are used to assign the result of an expression to a variable. We have the usual operator which is "=". The statement "a = 10;" means, the variable a will get a value 10 stored in it. Another form of the assignment operator is the shorthand arithmetic operator. This takes the form "a += 1". This is equivalent to write as "a = a + 1". Like this, we have -=, *=, /= and %=.

1. (b) State the rules that applied while evaluating expression in automatic type conversion

(a) if one of the operands is long double, the other will be converted o long double and the result will be long double

(b) if one of the operands is double, the other will be converted to double and the result will be a double

(c) if one of the operands is float, the other will be converted to float and the result will be float

(d) if one of the operands is unsigned long int, the other will be converted to unsigned long int and the result will be unsigned long int

(e) if one of the operands is long int, the other will be converted to long int and the result will be long int

(f) if one of the operands is unsigned int, the other will be converted to unsigned int and the result will be unsigned int

(g) All short and char are automatically converted to int

2. (a) **In what way an array is different from an ordinary variable?**

A variable can store only one value at a time. An array can store more than one value (a set of related element) at a time. However, the upper dimension of the array limits the number of values it can store.

2. (b) **What conditions must be satisfied by the entire elements of any given array?**

All the elements of an array should be of same data type as that of the declared array data type. Thus, an integer array will have all its elements as integers only.

2. (c) **What are subscripts? How are they written? What restrictions apply to the values that can be assigned to subscripts?**

Subscripts are used to access the specific element of a given array. For example, to access the fourth element of an array, it is written as a[3]. A subscript will always start with 0. Only integer values become a subscript. A subscript cannot be negative. A subscript value can never exceed the upper bound of the array.

2. (d) **What advantage is there in defining an array size in terms of a symbolic constant rather than a fixed integer quantity?**

The main advantage is that the array dimension is visible to you upfront on seeing the program. This means you can be very careful that you do not exceed the upper bound of the array.

2. (e) **Write a program to find the largest element in an array.**

```
#include<stdio.h>
#define MAXSIZE 100

main()
{
  int n, i, max = 0, elements[MAXSIZE];
  printf(“\nEnter the number of elements : “);
  scanf(“%d”, &n);
  for (i = 0; i < n; i++)
  {
  printf(“\nEnter value for element %d : “,i);
  scanf(“%d”, &elements);
  }
  for (i = 0; i < n; i++)
  {
  if (max < elements[i])
    max = elements[i];
  }
  printf(“\nThe maximum is : %d”, max);
}
```

3. (a) **What is a structure within structure? Give an example.**

A Structure within a structure means nesting of structures. This means one structure is defined within another structure. The example is given below:

```
struct employee;
{
          char name[20];
          int empno;
          struct salary
          {
            int basic;
            int DA;
      };
}
```

3. (b) Write a C program to illustrate the concept of structure within structure.

```
/***************************************************************/
/* Nested structures */
/***************************************************************/

struct date
{
  int day;
  int month;
  int year;
};
struct employee
{
  int empno;
  char name[20];
  float basic;
  struct date joindate;
};

main()
{

struct employee emp1 = {100,”Muthu”,5000.00,10,12,2002};

printf(“\nEno : %d, Name : %s, Basic : %5.2f”,emp1.empno,emp1.name,
emp1.basic);

printf(“\nJoindate %2d.%2d.”,emp1.joindate.day,emp1.joindate.month);
printf(“%4d”,emp1.joindate.year);
}
```

4. (a) Write a C program, using pointer for string comparison.

```
main()
{
  char compare_function();
  int result;

  char inputA[20],inputB[20];

  printf(“\nEnter string A : “);
  gets(inputA);
  printf(“\nEnter string B : “);
  gets(inputB);
  result = compare_function(inputA,inputB);
  if (result == 0)
    printf(“\nStrings are same”);
  else
    printf(“\nStrings are different”);
}

int compare_function(char *a,char *b)
```

```
{
  while (*a != ‘\0’)
  {
  if (*a == *b)
  {
  a++;
  b++;
  }
  else
    return 1;
  }
  return 0;
}
```

4. **(b) Write a C program to arrange the given numbers in ascending order using pointers.**

```
main()
{
  int i,j,number,temp,*myptr,numbers[20];
  printf(“\nEnter the number of elements in the array : “);
  scanf(“%d”,&number);

  for (i=0; i < number; i++)
  {
  printf(“\nEnter number[%d] : “,i);
  scanf(“%d”,&numbers[i]);
  }

printf(“\nThe array is : “);
myptr = numbers;
for (i=0; i < number; i++)
  printf(“%d, “,*myptr++);

myptr = numbers;

for (i=0; i < number - 1; i++)
  for (j = i; j < number; j++)
  if (*(myptr+i) > *(myptr+j))
  {
  temp = *(myptr+i);
  *(myptr+i) = *(myptr+j);
  *(myptr+j) = temp;
  }
  printf(“\nThe sorted array is : “);
  myptr = numbers;
  for (i=0; i < number; i++)
    printf(“%d, “,*myptr++);
}
```

5. **Elaborately discuss about the file handling functions in C.**

If we want to store data in a file in the secondary memory, we must specify certain things about the file, to the operating system. They include:

1. Filename.
2. Data structure.
3. Purpose.

Filename is a string of characters that make up a valid filename for the operating system. It may contain two parts, a primary name and an optional period with the extension. Examples:

```
Input.data
store
PROG.C
Student c
Text.out
```

Data structure of a file is defined as **FILE** in the library of standard I/O function definitions. Therefore, all files should be declared as type **FILE** before they are used. **FILE** is a defined data type.

When we open a file, we must specify what we want to do with the file. For example, we may write data to the file or read the already existing data.

Following is the general format for declaring and opening a file:

```
FILE     *fp;
fp = fopen("filename", "mode");
```

The first statement declares the variable **fp** as a "pointer to the data type **FILE**". As stated earlier. **FILE** is a structure that is defined in the I/O library. The second statement opens the file named *filename* and assigns an identifier to the **FILE** type pointer **fp**. This pointer which contains all the information about the file is subsequently used as a communication link between the system and the program.

The second statement also specifies the purpose of opening this file. The *mode* does this job. *Mode* can be one of the following:

r open the file for reading only.

w open the file for writing only.

a open the file for appending (or adding) data to it.

Note that both the filename and mode are specified as strings. They should be enclosed in double quotation marks.

When trying to open a file, one of the following things may happen:

1. When the mode is 'writing' a file with the specified name is created if the file does not exist. The contents are deleted, if the file already exists.
2. When the purpose is 'appending', the file is opened with the current contents safe. A file with the specified name is created if the file does not exist.
3. If the purpose is 'reading', and if it exists, then the file is opened with the current contents safe; otherwise an error occurs.

Consider the following statements:

```
FILE *p1, *p2;
p1    = fopen("data", "r");
p2    = fopen("results", "w");
```

The file **data** is opened for reading and **results** is opened for writing. In case, the **results** file already exists, its contents are deleted and the file is opened as a new file. If **data** file does not exist, an error will occur.

Many recent compilers include additional modes of operation. They include:

r+ The existing file is opened to the beginning for both reading and writing.
w+ Same as **w** except both for reading and writing.
a+ Same as **a** except both for reading and writing.

We can open and use a number of files at a time. This number however depends on the system we use.

6. **Declare a queue of integers. Write a function to insert an element into a queue and to delete a element from a qaueue.**

Declaration

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

typedef struct queue
{
  NODE *front;
  NODE *rear;
} QUEUE;
```

Addition

```
int add(QUEUE *queue, int data)
{
  NODE *newnode;
  if ((newnode=(NODE *)malloc(sizeof(NODE))) == NULL)
    return 0;

  newnode->data = data;
  newnode->next = NULL;

  queue->rear->next = newnode;
  queue->rear = newnode;

  return 1;
}
```

Deletion

```
int delete(QUEUE *queue)
{
  NODE *oldnode;
  int data;

  oldnode = queue->front->next;
  data = oldnode->data;

  if (queue->front->next->next == NULL)
    queue->rear = queue->front;
  else
    queue->front->next = queue->front->next->next;
  free(oldnode);
  return data;
}
```

7. **Explain the algorithm for deleting a node from a lexically ordered binary tree.**

```
struct binarytree
{
  struct binarytree *leftptr ;
  int data ;
  struct binarytree *rightptr ;
};
delete ( struct binarytree **root, int data )
{
  int nodefound ;
  struct binarytree *parent, *temp, *tempsucc ;
  if ( *root == NULL )
  {
  printf ( “\nTree is empty” ) ;
  return ;
  }
  parent = temp = NULL ;
  locate ( root, data, &parent, &temp, &nodefound ) ;
  if ( nodefound == FALSE )
  {
    printf ( “\nData not nodefound” ) ;
    return ;
  }
  if ( temp -> leftptr != NULL && temp -> rightptr != NULL)
  {
    parent = temp ;
    tempsucc = temp -> rightptr ;
    while ( tempsucc -> leftptr != NULL )
  {
    parent = tempsucc ;
    tempsucc = tempsucc -> leftptr ;
  }
  temp -> data = tempsucc -> data ;
  temp = tempsucc ;
}
if ( temp -> leftptr == NULL && temp -> rightptr == NULL )
{
if ( parent -> rightptr == temp )
  parent -> rightptr = NULL ;
else
  parent -> leftptr = NULL ;
free ( temp ) ;
return ;
}
if ( temp -> leftptr == NULL && temp -> rightptr != NULL )
{
if ( parent -> leftptr == temp )
  parent -> leftptr = temp -> rightptr ;
else
  parent -> rightptr = temp -> rightptr ;
free ( temp ) ;
return ;
}
if ( temp -> leftptr != NULL && temp -> rightptr == NULL )
{
if ( parent -> leftptr == temp )
  parent -> leftptr = temp -> leftptr ;
else
  parent -> rightptr = temp -> leftptr ;
  free ( temp ) ;
  return ;
  }
}
locate ( struct binarytree **root, int data,
        struct binarytree **myparent,
        struct binarytree **temp, int *nodefound )
{
  struct binarytree *mynode ;
  mynode = *root ;
  *nodefound = FALSE ;
  *myparent = NULL ;
  while ( mynode != NULL )
{
if ( mynode -> data == data )
{
  *nodefound = TRUE ;
  *temp = mynode ;
  return ;
}
if ( mynode -> data > data )
{
  *myparent = mynode ;
  mynode = mynode -> leftptr ;
}
else
{
  *myparent = mynode ;
  mynode = mynode -> rightptr ;
  }
  }
}
```

8. (a) Explain quick sort with algorithm.

This method, invented by Hoare, is considered to be a very faster method to sort the elements. The method is also called partition exchange sorting. The method is based on divide-and-conquer technique, i.e., the entire list is divided into various partitions and sorting is again and again applied on the partitions.

In this method, the list is divided into two based on an element called the pivot element. Usually, the first element is considered to be the pivot element. Now, move the pivot element into its correct position in the list. The elements to the left of the pivot are less than the pivot while the elements to the right of the pivot are greater than the pivot. The process is reapplied to each of these partitions. This process proceeds till we get the sorted list of elements. Let us understand this by an example. Consider the list—74, 39, 35, 32, 97, 84.

(1) We will take 74 as the pivot and move it to position so that the new list becomes – 39, 35, 32, 74, 97, 84. Note that the elements to the left are lesser than 74. Also, these elements (39, 35 and 32) are yet to be sorted. Similarly, the elements 97 and 84, which are right to the pivot, are yet to be sorted.

(2) Now take the partitioned list—39, 35, 32. Let us take 39 as the pivot. Moving it to the correct position gives—35, 32, 39. The partition to the left is 35, 32. There is no right partition. Reapply the process on the partition, we get 32, 35. Thus, we have 32, 35, 39.

(3) Apply the process to the right partition of 74 which is 97, 84. Taking 97 as the pivot and positioning it, we get 84, 97.

(4) Assembling all the elements from each partition, we get the sorted list.

C program for quick sort

The quick sort is a very good example for recursive programming. Let us assume that an array named elements[] will hold the array to be sorted and maxsize is the number of elements to be sorted. We will also assume that the sorting order is ascending in nature.

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

#define MAXSIZE 500

void quickSort(int elements[], int maxsize);
void sort(int elements[], int left, int right);

int elements[MAXSIZE];

int main()
{
int i, maxsize;
printf("\nHow many elements you want to sort: ");
scanf("%d",&maxsize);

printf("\nEnter the values one by one: ");
for (i = 0; i < maxsize; i++)
{
  printf ("\nEnter element %i :",i);
  scanf("%d",&elements[i]);
}

printf("\nArray before sorting:\n");
for (i = 0; i < maxsize; i++)
  printf("[%i], ",elements[i]);

printf ("\n");
quickSort(elements, maxsize);

printf("\nArray after sorting:\n");
for (i = 0; i < maxsize; i++)
  printf("[%i], ", elements[i]);
}

void quickSort(int elements[], int maxsize)
```

```
{
sort(elements, 0, maxsize - 1);
}
void sort(int elements[], int left, int right)
{
int pivot, l, r;
l = left;
r = right;
pivot = elements[left];
while (left < right)
{
  while ((elements[right] >= pivot) && (left < right))
    right—;
  if (left != right)
  {
  elements[left] = elements[right];
  left++;
  }
  while ((elements[left] <= pivot) && (left < right))
    left++;
  if (left != right)
  {
  elements[right] = elements[left];
  right—;
  }
}
elements[left] = pivot;
pivot = left;
left = l;
right = r;
if (left < pivot)
  sort(elements, left, pivot - 1);
if (right > pivot)
  sort(elements, pivot + 1, right);
}
```

8. (b) Analyze the worst case performance of quick sort and compare with selection sort.

Let us now analyze the efficiency of the quick sort. The selection of the pivot plays a vital role in determining the efficiency of the quick sort. The main reasons for this are—the pivot may partition the list into two so that one partition is much bigger than the other and the partition may be in an unsorted fashion (which is possible to the maximum). We will analyze the quick sort from the comparison point of view. We will consider two cases here, which could be the possible situations:

Case I:

In this case we will make two vital assumptions. These are:

1. The pivot, which we choose, will always be swapped into exactly the middle of the partition. To put it simple, the pivot (after swapping into its correct position) will have an equal number of elements both to its left and right.

2. The number of elements in the list is a power of 2. This means, if there are x elements in the array, we say that $x = 2^y$. This can rewritten as $y = \log_2 x$.
   In this situation, the first of the pass will have x comparisons. When the first pass is completed, the list will be partitioned into two equal halves, each with x/2 elements (which is obvious since we assumed that the pivot will partition the list exactly into two). In the next pass, we will have four partitions, each with equal number of elements (which will be x/4). Proceeding in this way, we will get:

Pass	Number of comparisons
1	X
2	2 * (x / 2)
3	4 * (x / 4)
4	8 * (x / 8)
X	X * (x / x)

   Thus, the total number of comparisons would be O(x) + O(x) + … + y terms. This is equivalent to O(x * y). Let us substitute $y = \log_2 x$, so that we get O(x log x). Thus the efficiency of quick sort is O(x log x).

Case II:

Let us now forgo one of our assumptions, which we made earlier. We will assume that the pivot partitions the list into two so that one of the partitions has no elements which the other has all the other elements. This is the worst case possible. Here, the total number of comparisons at the end of the sort would have been:

$(x - 1) + (x - 2) + \ldots + 2 + 1 = ½(x - 1) * x = ½(x^2) - ½(x) = O(x^2)$. Thus, the efficiency of the quick sort in its worst case is $O(x^2)$.

Selection sort efficiency

Let us now analyze the efficiency of the selection sort. You can easily understand from the algorithm that the first pass of the program does (maxsize – 1) comparisons. The next pass does (maxsize – 2) comparisons and so on. Thus, the total number of comparisons at the end of the sort would be :

$$(\text{maxsize} - 1) + (\text{maxsize} - 2) + \ldots + 1 = \text{maxsize} * (\text{maxsize} - 1) / 2 = O(\text{maxsize}^2).$$

Note that this efficiency is same as the worst case of quick sort.