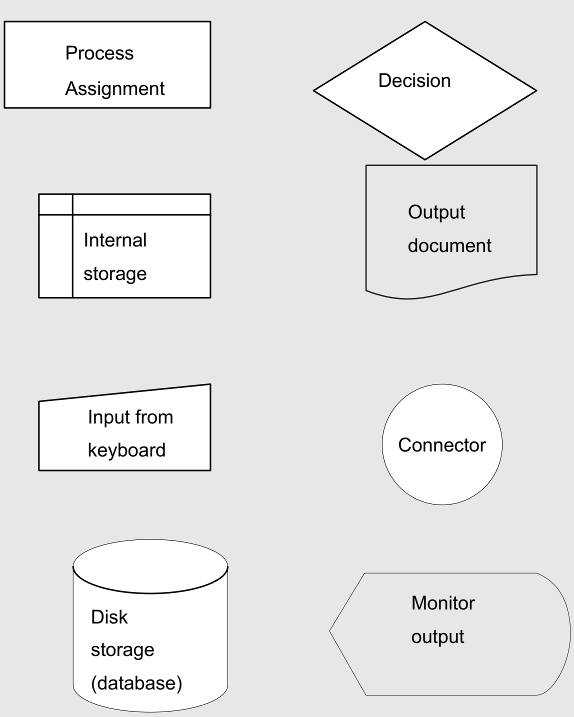
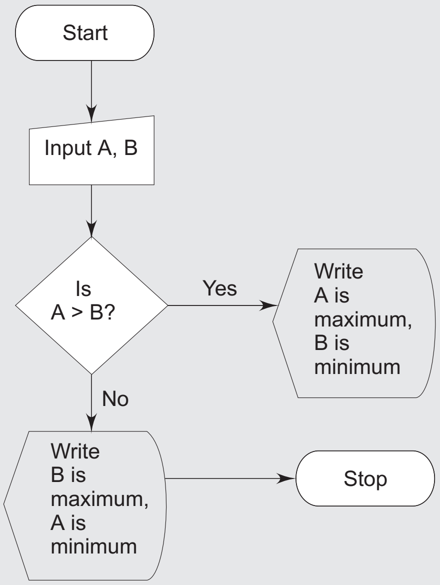
PAPER 2

1. (a) **What is a flowchart? Explain the different symbols used in a flowchart.**

A flowchart is the pictorial representation of an algorithm. The various steps in an algorithm are drawn in the form of prescribed symbols. The flow of the algorithm is maintained in a flowchart. The various symbols used in a flowchart are as below. The name of the symbol is given inside the symbol itself.



1. (b) Write a flowchart to find the maximum and minimum of given numbers.


Start
Input A, B
Is A > B?
Yes
Write A is maximum, B is minimum
No
Write B is maximum, A is minimum
Stop

2. (a) What do you mean by functions? Give the structure of the functions and explain about the arguments and their return values.

A C program has always a main module (called main) where the execution of the program begins and ends. If a program is very big and if we write all the statements of that big program in main itself, the program may become tool complex and large. As a result, the task of debugging and maintaining will become very difficult. On the other hand, if we split the program into several functional portions, these are easier. These subprograms are called functions.

The structure of a function is:

```
Function name (argument)
Argument declaration;
{
        local variables declaration;
        Executable statements;
        Return value;
}
```

Arguments—the arguments are valid variable names separated by commas. The argument variables receive values from a calling function. Thus they provide as the link between the main program and the function.

Return values—a function may or may not send a value back to the calling function. This value which is sent to the calling function is the return value of the function. It is achieved through the return keyword. There can be only one value in a return statement.

2. (b) Write a C program that uses a function to sort an array of integers.

```
main()
{
        int i;
        static int data[6] = {20, 10, 5, 12, 13, 3};
        printf("\nList before sorting"};
        for (i = 0; i < 6; i++)
                printf("%d, ", data[i]);
        sort(data, 6);
        printf("\nList after sorting"};
        for (i = 0; i < 6; i++)
                printf("%d, ", data[i]);
}

sort(arr, n)
int arr[], n;
{
        int j, k, temp;
        for (j = 1; j <= n – 1; j++)
                for (k = 1; k <= m – j; k++)
                        if (arr[k – 1] >= arr[k])
                        {
                                temp = arr[k – 1];
                                arr[k – 1] = arr[k];
                                arr[k] = temp;
                        }
}
```

3. (a) Explain the advantages of structure type over the array type variable.

1. An array type variable can store only homogeneous data i.e. it can store only values of data types according to which the variable is declared. For example, an integer array can store only integer numbers. This limitation is not there in a structure variable. A structure variable can host many data types at the same time.
2. An array type is subjected to lower and upper bounds limitations. If we try to read or write past these bounds, we will get error. This is not a limitation in structure variable.
3. A structure variable can host an array variable also.

3. (b) Define a structure that represent a complex number (contains two floating-point members, called real and imaginary). Write a C program to add, subtract and multiply two complex numbers.

```
struct complexnumber
{
          float real;
          float imaginary;
};

main()
{
        struct complexnumber c1, c2, add, sub, mult;
        printf("\nEnter complex number 1 (Real imaginary)");
        scanf("%f %f",&c1.real, &c1. imaginary);
        printf("\nEnter complex number 2 (Real imaginary)");
        scanf("%f %f",&c2.real, &c2. imaginary);
        add.real = c1.real + c2.real;
        add.imaginary = c1.imaginary + c2.imaginary;
        sub.real = c1.real - c2.real;
        sub.imaginary = c1.imaginary - c2.imaginary;
        mult.real = c1.real * c2.real;
        mult.imaginary = c1.imaginary * c2.imaginary;
        printf("\nSum is Real %f imaginary %f", add.real, add.imaginary);
        printf("\nDiff is Real %f imaginary %f", sub.real, sub.imaginary);
      printf("\nProduct is Real %f imaginary %f", mult.real,
mult.imaginary);
}
```

4. The roots of a quadratic equation of the form $ax^2 + bx + c = 0$ are given by the following equations:

$$X1 = -b + root(b^2 - 4ac) / 2a;$$
$$X2 = -b - root(b^2 - 4ac) / 2a;$$

Write a function to calculate the roots. The function must use two pointers, one to receive the coefficients and the other to send the roots to the calling function.

```
#include <math.h>
struct equation
{
          float a, b, c;
};
struct roots
{
          float x1, x2;
};
void findroots();
main()
{
          struct equation myeq;
          struct roots myroots;
          printf("\nEnter values for a, b and c");
          scanf("%f %f %f", &myeq.a, &myeq.b, &myeq.c);
          findroots(&myeq, &myroots);
          printf("\nRoots are X1 = %f, X2 = %f", myroots.x1, myroots.x2);
}

void findroots(struct equation *myeqarg, struct roots *myrootarg)
{
          float a, b, c, discr, a1;
          a = myeqarg->a;
          b = myeqarg->b;
          c = myeqarg->c;
    discr = b * b - 4.0 * a * c;
          myrootarg->x1 = (-b + sqrt(discr)) / (2.0 * a);
          myrootarg->x2 = (-b - sqrt(discr)) / (2.0 * a);
}
```

5. **Define a file and elaborately discuss about reading, opening and closing of a file.**

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. Write a program that uses a stack to check for matching left and right parentheses, left and right braces, and left and right brackets in a string of characters.

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

static struct stack
      {
      int top;
      int numbers[MAXSIZE];
      } mystack;

void push(struct stack *,int);
int pop(struct stack*);
int isempty(struct stack*);
main()
{
  char expression[MAXSIZE];
  int i = -1;
  int number;
  mystack.top = -1;

  printf(“\nEnter the expression —> “);
  scanf(“%s”,expression);
  while(expression[++i] != ‘\0’)
  {
    switch (expression[i])
    {
    case ‘(‘ : push(&mystack,’(‘);
              break;
    case ‘)’ : number = pop(&mystack);
              break;
    case ‘[‘ : push(&mystack,’[‘);
              break;
    case ‘]’ : number = pop(&mystack);
              break;
    case ‘{‘ : push(&mystack,’{‘);
              break;
    case ‘}’ : number = pop(&mystack);
```

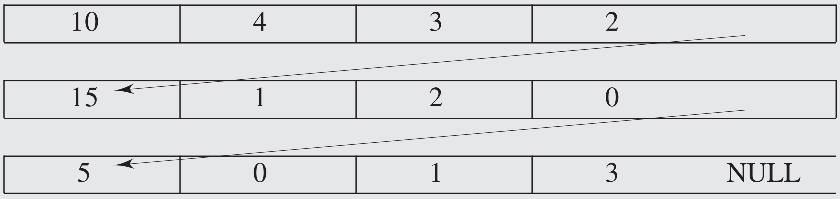
```
              break;
    }
  }
  if (!isempty(&mystack))
    printf("\nExpression has unmatched parenthesis");
  else
    printf("\nExpression is okay");
  }

  void push(struct stack *mystack,int number)
  {
            if (mystack->top==MAXSIZE-1)
            {
                      printf("\nStack Overflow\n");
                      exit(1);
            }
            mystack->numbers[++mystack->top]=number;
  }
  int pop(struct stack *mystack)
  {
          if (isempty(mystack))
          {
                    printf("\nStack Underflow\n");
                    exit(1);
          }
          return(mystack->numbers[mystack->top—]);
}
int isempty(struct stack *mystack)
{
          return((mystack->top==-1));
}
```

7. **How can a polynomial in three variables (x, y and z) be represented by a singly linked list? Each node in a list should represent a term and should contain the powers of x, y and z as well as the coefficient of that term.**

Such a polynomial can be represented with a linked list in the same way as we represent a polynomial with a single variable. Each node in the list will have the following structure:

COEFFICIENT	X-POWER	Y-POWER	Z-POWER	NEXT ADDR

Consider a simple polynomial $P(x, y, z) = 10x^4y^3z^2 + 15\ xy^2 + 5\ yz^3$. Please do not worry about the validity of this polynomial. The idea is to explain the representation. This can be represented as follows:



8. Explain the algorithm for selection sort and give a suitable example.

As the name suggests, the first element of the list is selected. It is compared repeatedly with all the elements. If any element is found to be lesser than the selected element, these two are swapped. This procedure is repeated till the entire array is sorted. Let us understand this with a simple example. Consider the list 74, 39, 35, 32, 97, 84. Following is the table, which gives the status of the list after each pass is completed.

Pass	List after pass
1	32, 39, 35, 74, 97, 84
2	32, 35, 39, 74, 97, 84
3	32, 35, 39, 74, 97, 84
4	32, 35, 39, 74, 97, 84
5	32, 35, 39, 74, 84, 97

C program for selection sort

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 selection(int elements[], int maxsize);

int elements[MAXSIZE],maxsize;

int main()
{
  int i;
  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”);
    selection(elements, maxsize);
    printf(“\nArray after sorting:\n”);
    for (i = 0; i < maxsize; i++)
      printf(“[%i], “, elements[i]);
    }

    void selection(int elements[], int array_size)
    {
    int i, j, k;
    int min, temp;
    for (i = 0; i < maxsize-1; i++)
    {
      min = i;
      for (j = i+1; j < maxsize; j++)
      {
      if (elements[j] < elements[min])
        min = j;
      }
      temp = elements[i];
      elements[i] = elements[min];
      elements[min] = temp;
    }
}
```

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 :

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