1(a). Define Data Structures. Explain the different types of data structures with examples.

Data may be organized in many different ways. The logical or mathematical model of a particular organization of data is called a data structure.

Data structures are generally classified into Primitive data Structures & Non-primitive data Structures.

1. Primitive data Structures: Primitive data structures are the fundamental data types which are supported by a programming language. Basic data types such as integer, real, character and Boolean are known as Primitive data Structures. These data types consists of characters that cannot be divided and hence they also called simple data types.

2. Non- Primitive data Structures: Non-primitive data structures are those data structures which are created using primitive data structures. Examples of non-primitive data structures is the processing of complex numbers, linked lists, stacks, trees, and graphs.

Based on the structure and arrangement of data, non-primitive data structures is further classified into Linear Data Structure & Non-linear Data Structure.

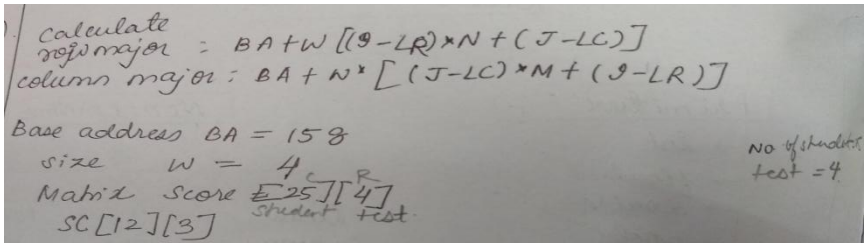
1.Linear Data Structure: A data structure is said to be linear if its elements form a sequence or a linear list. There are basically two ways of representing such linear structure in memory-One way is to have the linear relationships between the elements represented by means of

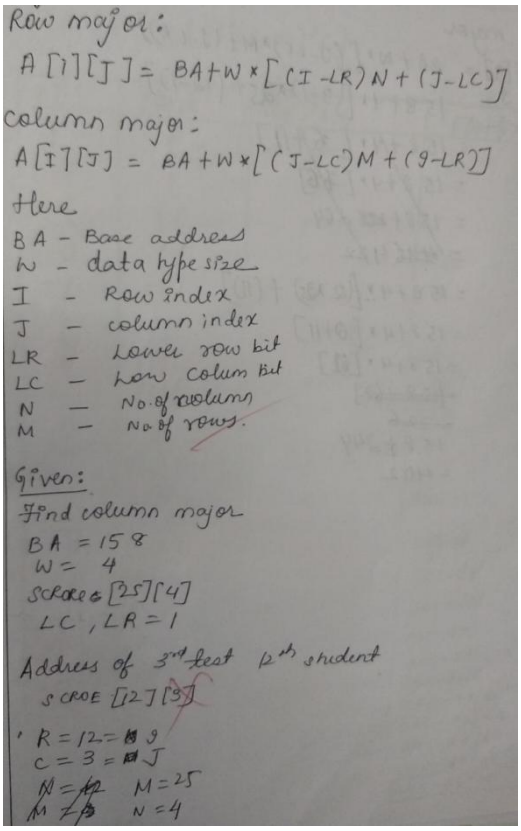
sequential memory location. These linear structures are called arrays. The other way is to have the linear relationship between the elements represented by means of pointers or links. These linear structures are called linked lists.

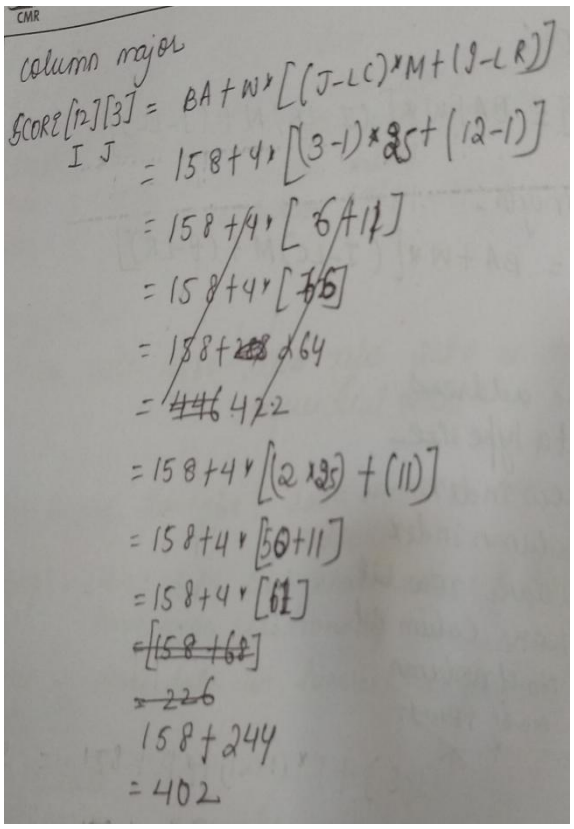
The common examples of linear data structure are Arrays, Queues, Stacks, Linked lists

2.Non-linear Data Structure: A data structure is said to be non-linear if the data are not arranged in sequence or a linear. The insertion and deletion of data is not possible in linear fashion. This structure is mainly used to represent data containing a hierarchical relationship between elements. Trees and graphs are the examples of non-linear data structure.

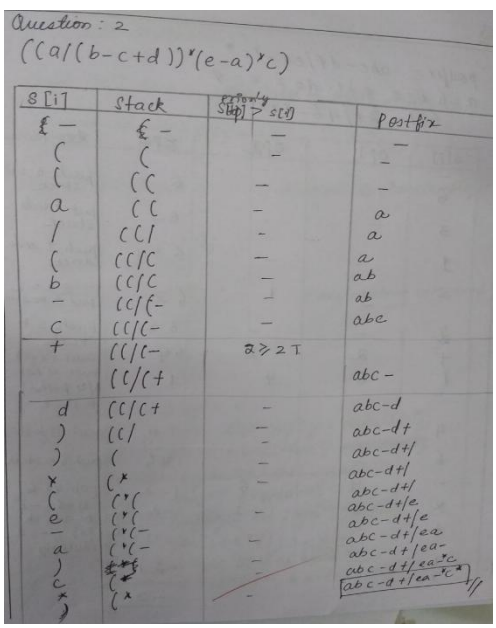
1(b). What is the formula to calculate the location in the row major and column major? Suppose each student in a class of 25 is given 4 tests, assume the students are numbered from 1 to 25, and the test scores are assigned in the 25*4 matrix called SCORE. Suppose Base(SCORE)=158, w=4, and the programming language uses column major order to store this 2D array, then find the address of 3rd test of 12th student i.e SCORE(15,2).

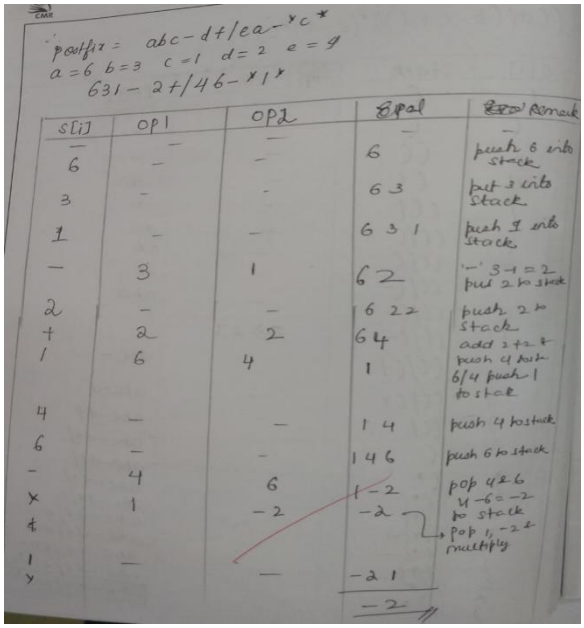






2(a). Convert the infix expression ((a/(b-c+d))*(e-a)*c) to postfix expression and evaluate that postfix expression for given data a=6, b=3, c=1, d=2, e=4(using stack representation).





2(b). Write a C program with an appropriate structure definition and variable declaration to store information about an employee, using nested structures. Consider the following fields like: ENAME, EMPID, DOJ(Date, Month, Year).

```
#include<stdio.h>
struct e
{
        int empid;
        char employee_name[100];
        char employee_department[100];
        int employee_age;
        int employee_doj[3];
        int salary[3];
}
a[];
int n=0;
void input()
{
        printf("Enter the employee's ID.: ");
        scanf("%d", &a[n].empid);
        printf("Enter the employee's name: ");
```

```
            scanf(" %[^\n]", a[n].employee_name);
            printf("Enter the employee's department.: ");
            scanf(" %[^\n]",a[n].employee_department);
            printf("Enter the employee's age: ");
            scanf("%d", &a[n].employee_age);
            printf("Enter the employee's date of joining (xx xx xxxx): ");
            scanf("%d", &a[n].employee_doj[0]);
            scanf("%d", &a[n].employee_doj[1]);
            scanf("%d", &a[n].employee_doj[2]);
            printf("Enter the employee's salary (Basic DA HRA): ");
            scanf("%d", &a[n].salary[0]);
            scanf("%d", &a[n].salary[1]);
            scanf("%d", &a[n].salary[2]);
            n++;
        }
        void output()
        {
            for(int i=0;i<n;i++)
            {
                printf("Record #%d:\n", i+1);
                printf("Employee's ID: %d\n", a[i].empid);
                printf("Employee's name: %s\n", a[i].employee_name);
                printf("Employee's department: %s\n",
a[i].employee_department);
                printf("Employee's age: %d\n", a[i].employee_age);
                printf("Employee's date of joining:
%d/%d/%d\n",a[i].employee_doj[0], a[i].employee_doj[1], a[i].employee_doj[2]);
                printf("Employee's salary: Rs. %d\n",
a[i].salary[0]+a[i].salary[1]+a[i].salary[2]);
                printf("1. Basic = Rs. %d\n", a[i].salary[0]);
                printf("2. DA = Rs. %d\n", a[i].salary[1]);
```

```
                printf("3. HRA = Rs. %d\n", a[i].salary[2]);
        }
}
int main()
{
        while(1)
        {
                int n;
                printf("1. Input\n");
                printf("2. Output\n");
                printf("3. Exit\n");
                scanf("%d", &n);
                switch(n)
                {
                        case 1: input();
                        break;
                        case 2: output();
                        break;
                        case 3: return 0;
                        default: printf("Invalid choice!\n");
                }
        }
}
```

3. What is dynamic memory allocation? Explain different functions associated with dynamic memory allocation and deallocation with syntax and example. Code a C program to illustrate the same for allocating memor to store n integers and find the sum using dynamic memory allocation.

Dynamic memory allocation is the process of assigning the memory space during the execution time or the run time. It refers to performing manual memory management for dynamic memory allocation via a group of functions in the standard library, namely malloc, realloc, calloc and free.

1. **malloc( ): The function malloc allocates a user- specified amount of memory and a pointer to the start of the allocated memory is returned.**
   If there is insufficient memory to make the allocation, the returned value is NULL.
   Syntax: data_type *x;
   x= (data_type *) malloc(size);

   Where,

   x is a pointer variable of data_type

   size is the number of bytes

   Ex: int *ptr;

   ptr = (int *) malloc(100*sizeof(int));

2. calloc( ): The function calloc allocates a user- specified amount of memory and initializes the allocated memory to 0 and a pointer to the start of the allocated memory is returned. If there is insufficient memory to make the allocation, the returned value is NULL. Syntax: data_type *x;

x= (data_type *) calloc(n, size);

Where,

data_type *x;

x= (data_type *) calloc(n, size);

x is a pointer variable of type int

n is the number of block to be allocated

size is the number of bytes in each block

Ex: int *x

x= calloc (10, sizeof(int));

The above example is used to define a one-dimensional array of integers. The capacity of this array is n=10 and x [0: n-1] (x [0, 9]) are initially 0

3. realloc( ):

->Before using the realloc( ) function, the memory should have been allocated using malloc() or calloc( ) functions.

->The function relloc( ) resizes memory previously allocated by either mallor or calloc, which

means, the size of the memory changes by extending or deleting the allocated memory.

->If the existing allocated memory need to extend, the pointer value will not change.

->If the existing allocated memory cannot be extended, the function allocates a new block and copies the contents of existing memory block into new memory block and then deletes the old memory block.

->When realloc is able to do the resizing, it returns a pointer to the start of the new block and when it is unable to do the resizing, the old block is unchanged and the function returns the value NULL

Syntax: data_type *x;

x= (data_type *) realloc(p, s);

The size of the memory block pointed at by p changes to S. When s > p the additional s-p

memory block have been extended and when s < p, then p-s bytes of the old block are freed.

4. free( )

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

Syntax: free(ptr);

This statement cause the space in memory pointer by ptr to be deallocated

4(a). Differentiate between structure and unions.

	STRUCTURE	UNION
Keyword	The keyword **struct** is used to define a structure	The keyword **union** is used to define a union.
Size	When a variable is associated with a structure, the compiler allocates the memory for each member. The size of structure is **greater than or equal to the sum of sizes of its members.**	when a variable is associated with a union, the compiler allocates the memory by considering the size of the largest memory. So, size of **union is equal to the size of largest member.**
Memory	Each member within a structure is assigned unique storage area of location.	Memory allocated is shared by individual members of union.
Value Altering	Altering the value of a member will not affect other members of the structure.	Altering the value of any of the member will alter other member values.
Accessing members	Individual member can be accessed at a time.	Only one member can be accessed at a time.
Initialization of Members	Several members of a structure can initialize at once.	Only the first member of a union can be initialized.

4(b). Write a C function to:

```
i)printf("Enter element to insert : ");
 scanf("%d", &num);
 printf("Enter the element position : ");
scanf("%d", &pos);
if(pos > size+1 || pos <= 0)
 { printf("Invalid position! Please enter position between 1 to %d", size); }
else { for(i=size; i>=pos; i--)
 { arr[i] = arr[i-1]; }
 arr[pos-1] = num; size++;
printf("Array elements after insertion : ");
 for(i=0; i<size; i++)
 { printf("%d\t", arr[i]); } }
 return 0; }

Ii)printf("Enter the element position to delete : ");
 scanf("%d", &pos);
 if(pos < 0 || pos > size)
{ printf("Invalid position! Please enter position between 1 to %d", size); }
else {
for(i=pos-1; i<size-1; i++)
{ arr[i] = arr[i + 1]; }
printf("\nElements of array after delete are : ");
for(i=0; i<size; i++)
{ printf("%d\t", arr[i]); } }
return 0; }
```

5. Define stack. Write a C program demonstrating the various stack operations, including cases for overflow and underflow of stacks.

A stack is an ordered list in which insertions (pushes) and deletions (pops) are made at one end called the top.

Given a stack S= (a0, ... ,an-1), where a0 is the bottom element, an-1 is the top element, and ai is on top of element ai-1, 0 < i < n.

Code:

```
#include <stdio.h>
int MAXSIZE = 8;       int stack[8];     int top = -1;
int isempty() {

   if(top == -1)
      return 1;
   else
      return 0;}
   int isfull() {

   if(top == MAXSIZE)
      return 1;
   else
      return 0;}
int peek() {
   return stack[top];}
int pop() {
   int data;

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

   if(!isfull()) {
      top = top + 1;
      stack[top] = data;
   } else {
      printf("Could not insert data, Stack is full.\n");
   }}
int main() {
   // push items on to the stack
   push(3);
   push(5);
   push(9);
   push(1);
   push(12);
   push(15);
```

```
   printf("Element at top of the stack: %d\n" ,peek());
   printf("Elements: \n");

   // print stack data
   while(!isempty()) {
      int data = pop();
      printf("%d\n",data);
   }

   printf("Stack full: %s\n" , isfull()?"true":"false");
   printf("Stack empty: %s\n" , isempty()?"true":"false");

   return 0;}
```

6(a). Write a C function to perform pattern matching.

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

int match(char [], char []);

int main() {
  char a[100], b[100];
  int position;

  printf("Enter some text\n");
  gets(a);

  printf("Enter a string to find\n");
  gets(b);

  position = match(a, b);

  if (position != -1) {
    printf("Found at location: %d\n", position + 1);
  }
  else {
    printf("Not found.\n");
  }

  return 0;
}

int match(char text[], char pattern[]) {
  int c, d, e, text_length, pattern_length, position = -1;

  text_length    = strlen(text);
  pattern_length = strlen(pattern);
```

```
 if (pattern_length > text_length) {
  return -1;
 }

 for (c = 0; c <= text_length - pattern_length; c++) {
  position = e = c;

  for (d = 0; d < pattern_length; d++) {
   if (pattern[d] == text[e]) {
    e++;
   }
   else {
    break;
   }
  }
  if (d == pattern_length) {
   return position;
  }
 }

 return -1;
}
```

6(b). What is the output of the following code?

6,6

3,4

6,2

4,6