Practical-2

Aim: Write a Program:

1. **To implement an Array representation of the sparse matrices.**
2. **To search the element in 2-D array.**
3. **To create, initialize and print the values of the 3-D array.**

Code:

1. **To implement an Array representation of the sparse matrices :**

```
#include<stdio.h>
#include<conio.h>
void main()
{
int arr[4][4] = {{0,0,1,0},{0,0,0,2},{1,0,0,0},{0,3,0,0}};
int i,j,k=1;
int sparse[20][3];
clrscr();
printf("original matrix :\n");
for(i=0;i<4;i++){
  for(j=0;j<4;j++)
  {
   printf("%d",arr[i][j]);

  }
  printf("\n");
}
for(i=0;i<4;i++){
for(j=0;j<4;j++){
if(arr[i][j]!=0){
sparse[k][0]=i;
sparse[k][1]=j;
sparse[k][2]= arr[i][j];
k++;
}
}
}
printf("sparse matrix :\n");
printf("Row\tCol\tVal\n");
for(i=0;i<k;i++){
 printf("%d\t%d\t%d\n",sparse[i][0],sparse[i][1],sparse[i][2]);
}
getch();
}
```

Output:

Practical-2

```
original matrix :
0010
0002
1000
0300
sparse matrix :
Row     Col     Val
0       0       0
0       2       1
1       3       2
2       0       1
3       1       3
```

2. To search the element in 2-D array :

```
#include <stdio.h>
#include <conio.h>
void main() {
   int arr[4][4] = {{1, 4, 7, 11},{2, 5, 8, 12},{3, 6, 9, 16},{10, 13, 14, 17}};
   int target, i = 0, j = 3, found = 0,x,y;
   clrscr();
   printf("Array elements:\n");
   for(x=0;x<3;x++) {
   for(y=0;y<4;y++){
   printf("%d",arr[x][y]);
   }
   printf("\n");
   }
   printf("Enter the element to search: ");
   scanf("%d", &target);
   while (i < 4 && j >= 0) {
   if (arr[i][j] == target) {
      printf("Element %d found at position [%d][%d]\n", target, i, j);
      found = 1;
      break;
   } else if (arr[i][j] > target) {
      j--;
   } else {
      i++;
   }
   }
   if (!found)
   printf("Element %d not found in the array.\n", target);
   getch();
}
```

Practical-2

Output:

```
Array elements:
14711
25812
36916
Enter the element to search: 8
Element 8 found at position [1][2]
```

3. To create, initialize and print the values of the 3-D array.

```
#include <stdio.h>
#include <conio.h>
void main() {
int arr[2][3][4];
int i, j, k, value = 1;
clrscr();
for (i = 0; i < 2; i++) {
for (j = 0; j < 3; j++) {
for (k = 0; k < 4; k++) {
arr[i][j][k] = value;
value++;
}
}
}
printf("3-D Array Elements:\n");
for (i = 0; i < 2; i++) {
printf("Table %d:\n", i);
for (j = 0; j < 3; j++) {
for (k = 0; k < 4; k++) {
printf("%3d ", arr[i][j][k]);
}
printf("\n");
}
printf("\n");
}
getch();
}
```

Output:

Practical-2

```
3-D Array Elements:
Table 0:
  1   2   3   4
  5   6   7   8
  9  10  11  12

Table 1:
 13  14  15  16
 17  18  19  20
 21  22  23  24
```

B) Discuss following concepts with an example
i) Address Calculation of 1-D Array.
ii) Address Calculation of 2-D Array using Row Major and Column Major Order.

Answer :

1. 1-Dimensional Array (1D Array)

- **A linear array accessed using a single index.**
- **Formula for address calculation:**

$$\text{Address of } A[Index] = B + W \times (Index - LB)$$

Where:

- **B = Base address**
- **W = Size of one element in bytes**
- **LB = Lower bound of index (default = 0)**
- **Index = Index of element**

2. 2-Dimensional Array (2D Array)
A 2D array is like a matrix with M rows and N columns.
Two ways to store elements in memory:
(a) Row Major Order

- **Elements stored row by row.**
- **Formula:**

$$\text{Address of } A[I][J] = B + W \times ((I - LR) \times N + (J - LC))$$

Where:

- **I = Row index, J = Column index**
- **LR = Lower row index, LC = Lower column index**
- **N = Total columns**

(b) Column Major Order

- **Elements stored column by column.**
- **Formula:**

$$\text{Address of } A[I][J] = B + W \times ((J - LC) \times M + (I - LR))$$

Where:

- **M = Total rows**

Row Major Order :

- **Elements are stored row by row in contiguous memory locations.**
- **Traverses an entire row first, then moves to the next row.**

Practical-2

- **Efficient for row-wise access, less efficient for column-wise.**
- **Used in: Languages like C, C++.**
- **Applications: Suitable for row-wise operations, e.g., image processing.**

Column Major Order :

- **Elements are stored column by column in contiguous memory locations.**
- **Traverses an entire column first, then moves to the next column.**
- **Efficient for column-wise access, less efficient for row-wise.**
- **Used in: Languages like Fortran.**
- **Applications: Suitable for column-wise operations, e.g., matrix multiplication.**

Problem:
Given a 2D array B[5…9][10…14] (rows 5–9, cols 10–14),

- **Base address (B) = 500**
- **Each element size (W) = 4 bytes**
- **Find the address of B[7][12] in both Row Major and Column Major order.**

Solution:

Rows = 9 – 5 + 1 = 5 rows
Columns = 14 – 10 + 1 = 5 columns
LR = 5, LC = 10

Row Major Order :
Address = B + W * ((I - LR) * N + (J - LC))
Address = 500 + 4 * ((7 - 5) * 5 + (12 - 10))
Address = 500 + 4 * (2 * 5 + 2)
Address = 500 + 4 * 12
Address = 500 + 48 = 548

Column Major Order :
Address = B + W * ((J - LC) * M + (I - LR))
Address = 500 + 4 * ((12 - 10) * 5 + (7 - 5))
Address = 500 + 4 * (2 * 5 + 2)
Address = 500 + 4 * 12
Address = 500 + 48 = 548