Chapte 2

Introduction to Data Structures

2.1 INTRODUCTION

This section introduces the meaning of primitive data structures and their abstract data type (ADT) definitions. To study the data structures, we find it is necessary to establish clearly what is meant by data object and how it is stored in computer memory. Associated with each of these basics, we shall also study about arrays and matrices with their representation and operations. Several special types of matrices exist for various applications, in which sparse matrix will be considered in depth.

2.2 DATA OBJECT

The run-time grouping of one or more pieces of data in a computer may be termed as **data object**. In other words, a data object represents a container for data values – i.e., a place where data values may be stored and retrieved. For example, a data object may be a single value, or programmer – defined variable or even a literal constant. Take a look at the below examples:

Integer = {0, ±1, ±2, . . .}

Letter = {*A, B, . . Z, a, b, . . z*}

String = {*a, aa, ab, . .*}

Boolean = {*true, false*}

In this example, Integer, Letter, String, etc are the data objects, *true* and *false* are the values or instances of the Boolean data object, *A, B, . . Z, a, b, . . z* are the values of Letter data object, and so on. The data object String is the set of all possible string instances. Each instance is composed of characters.

2.3 DATA TYPE

A **data type** is a class of data objects together with a set of operations for creating and manipulating them. Every language has a set of primitive data types. For example, in C language *integer, float, char* as its primitive data types. A data type in a language may be studied at two different levels. The first one is through its specification or logical organization and the second is through its implementation.
The basic elements of specifications of a data type are:

1. The attributes that distinguish data objects of that type.
2. The values of the data objects of that type.
3. The operations that define the possible manipulations of data objects.

For example, consider an array data type in which the attributes might include the number of dimensions (rows and column), the subscript range and the data type of the elements. The values are the set of valid numbers and the operations include selecting an array element (subscripting), perform arithmetic and other operations on pairs of arrays.

A data object of type integer usually has no other attributes except its type. The set of integer values allowed may depend upon the language. For example, C integers may take values only from –32,767 to +32,767. The operations for the integer data type are arithmetic, relational, assignment, etc.

2.4 PRIMITIVE DATA STRUCTURE

A **data structure** is a data object together with the relationships that exist among the instances and among the individual elements.

The study of data structure is to identify and develop mathematical entities and operations. We are also concerned with the representation of data objects or entities and their implementations.

When the structuring of data is done at their most primitive level with in a computer; such a data structure is called **primitive data structure**. That is, the data structures, which are operated directly by machine-level instructions. No indirect software driven operations required to implement or operate these types of data structures. We will develop more complex data structures and show their applications to a variety of problems.

2.5 ABSTRACT DATA TYPE (ADT)

An **Abstract Data Type** (**ADT**) is a tool to specify the logical properties of a data type – that is, it provides *specifications* and *operations*. Generally, the abstraction provides encapsulation. For example, the primitive types such as *integer* and *float*, the languages provide facility for declaration and do a set of operations on them. However, the implementation details are hidden and the users need to know only the type name and the list of operations.

There are number of ways that may be used to specify the ADT. We shall follow the below given method in the form of an example to specify the ADT in a formal or semiformal way. Let us consider the `Array` data type for which the specification is nothing but a set of pairs (`index`, `value`) and the operations are creating an array, Add an element (`index`, and `value`) to an array and Retrieve, which returns the pair when its index value given. The Figure 2.1 shows the ADT specification for the array type.

```
ADT Array
{
     specification:
          set of pairs of the form(index, value)
          with a constraint that no two pairs should
          have the same index value.
     operations:
          Create() - Creates an empty array.
          Add(index,value) - adds this pair into the
                             array.
          Retrieve(index) - returns the pair with
                            given index value
}
```

Fig. 2.1 ADT specification for an Array.

Suppose we wish to store the student information – name and `marks` in an array, then the student name acts as the `index`, assuming that no two students have the same name. If `STUDENT` is the name of the array, an instance can be shown as,

```
STUDENT = {(John, 78), (Keerthi, 42), (Rama, 74),
           (Deepak, 10), (Pradeep, 100)}
```

Each pair consists of an `index` (name of the student) and a `value` (`marks`). We can change the marks of any student by specifying the pair (Deepak, 60) and this would modify the old value i.e., 10 to the new value 60. We can also retrieve the marks by specifying its index. For example, `Retrieve(Kerthi)` will return the value 42.

When a row major is used the mapping function can be designed easily because every new row starts after the size of the number of columns. That is,

```
Location of A[i][j] = BA + (i*COL_SIZE+j)          ...(2.2)
```

For the example Array A, if we want to find the address of element 3 whose row is 1 and column is 1

That is, A[1][1] = 1000 + (1 * 3 + 1)
= 1000 + 4 = 1004

A more general formula can be written a

```
Location of A[i][j]= BA + (i*COL_SIZE*size)+j*size     ...(2.3)
```

assuming that the lower bound is zero (like C arrays).

2.7 SPARSE MATRICES

A matrix is a tabular structure with certain number of rows (m) and certain number of columns (n). Here, m and n are the dimensions of the matrix. Our aim, in this section is to find an efficient method for storage and retrieval of certain type of matrix, called as **sparse matrix**. Also we discuss a way to develop a program to add two sparse matrices. When a matrix ($m \times n$) consists of more zero elements, such a matrix is called as a **sparse matrix**. If a matrix contains more non-zero elements, it is termed as **dense matrix**. The need for considering the sparse matrix is that we do not need to store all elements in memory, but only non-zero elements there by decreasing the storage requirements. In the following section, we will show how to store a sparse matrix using array and linked representations.

2.7.1 Array Representation

The non-zero entries of an irregular sparse matrix may be mapped into a one-dimensional array in row-major order. Consider the matrix shown in Figure 2.4(a) in which there are hardly few non-zero elements. These non-zero elements would be stored in another matrix called sparse matrix as shown in Figure 2.4(b).

To construct such a matrix, the row, column and the value are to be stored. This means that there will be always three rows and the maximum columns depend on the non-zero elements present in the original matrix.

0	0	0	0	7	0	0
1	0	0	0	0	0	9
0	6	0	8	0	0	0
0	0	0	0	4	0	0

(a) Sparse Matrix-A

a[ ]	0	1	2	3	4	5
row	1	2	2	3	3	4
col	5	1	7	2	4	5
value	7	1	9	6	8	4

(b) Array representation

Fig. 2.4 Representation of Sparse Matrix

The element 7 appears in 1st row and 5th column and hence its entry appears at 0th column of Figure 2.4(b).

Similarly, the second row consists of two non-zero elements 1 and 9 at [2][1] and [2][7] positions – they occupy 1st and 2nd columns in the array representation. Since there are six non-zero elements, you have six entries (column) in the final matrix. In addition to storing the number of non-zero elements, we need to store the dimension of the original matrix (row and column) along with the number of non-zero elements. For the example shown, there will be one extra entry (4, 7, 6).

If the sparse array is of integer type, the space requirement for matrix of Figure 2.4(a) is $4 \times 7 = 28 \times 2 = 56$ bytes. Note that each element occupies two bytes. In the same way, the matrix of Figure 2.4(b) takes $3 \times 5 = 18 \times 2 = 36$ bytes only. You may wonder that the memory saving in this example is not much – just 20 bytes. However, for large arrays, the saving will be considerable. For example, if we have a matrix of 1000×1000 in which assume that there are only 5000 non-zero elements, then the space requirement is,

Array Representation	Sparse representation
$= 1000 \times 1000 = 1000000$	$= 3 \times 5000 = 15000$
$= 1000000 \times 2 = 2000000$ bytes	$= 15000 \times 2 = 30000$ bytes

It is clear from the above calculation that the total saving is 2000000 – 30000 = 1970000 bytes! It is amazing! The memory saving is quite appreciable and therefore sparse matrix is really useful.

2.7.2 The Method and Implementation

The objective of this section and the following ones is to show how the sparse matrix be stored in a C structure and to add two spars matrices. We shall assume that the `row`, `col` and `value` of the sparse matrix be stored in a structure called `Term`.

```
struct Term
{
      int row;
      int col;
      int value;
};
```

The design approach we follow here is generic – this means that the method should be useful for implementing in any programming language and also suitable for object-oriented languages.

Since more than one non-zero value in a sparse matrix exist, we prefer an array a to store them. Also remember that the number of non-zero terms and the dimensions of the matrix are to be stored. Again a structure is useful to define all of them together making it as a sparse matrix template.

```
struct SparseMatrix {
       int rows, cols;  /* dimension */
       int terms; /* no. of nonzero terms */
       struct Term a[Max];
};
```

The array a holds the details of row, col and value of each non-zero element in the sparse matrix. For the sparse matrix shown in Figure 2.4(b), the structure would look as,

rows = 4, *cols* = 7, *terms* = 6

a[0] =	0	4	7
a[1] =	1	0	1
a[2] =	1	6	9
a[3] =	2	1	6
a[4] =	2	3	8
a[5] =	3	4	4

The function ReadMatrix()

Having defined the basic template for the sparse matrix, the next job is to show reading and printing of sparse matrices. Program 2.1 shows the function ReadMatrix() – to read the values of the matrix dimension and the non-zero elements.

Program 2.1
Reading a Sparse Matrix

```
void ReadMatrix (Sparse *x)
{
     int i;
     printf("Enter max rows and max cols: ");
     scanf("%d%d", &x->rows, &x->cols);
     printf("Enter nonzero terms: ");
     scanf("%d", &x->terms);
     printf("Enter Sparse Matrix (r,c,v):\n");
     for (i = 0; i < x->terms; i++)
          scanf("%d%d%d", &(x->a[i].row),
                    &(x->a[i].col), &(x->a[i].value));
}
```

The function uses a pointer variable x that is of type Sparse and its type definition is,

```
typedef struct SparseMatrix Sparse;
```

Since the value of x is to be returned to the calling program, it is declared as a reference parameter. To read the (row, col, value) triplet, we must first access the

SparseMatrix structure and then the Term structure. So we use x->a[i].row, x->a[i].col and x->a[i].value. Note that a is an array of structure. Observe that the input to the addition are two sparse matrices.

The function Add() takes two arguments x and y and produces the resultant, added matrix in z. The parameters x and y are declared as value parameters where as z is a reference parameter. The function Add() is shown in Program 2.2.

Program 2.2
Adding Two Sparse Matrices

```
void Add (Sparse x, Sparse y, Sparse *z)
{
     int cx, cy; /* cursors for x and y */
     int ix, iy; /* index for x and y   */

     if (x.rows != y.rows || x.cols != y.cols)
     {
          printf("Incompatible Matrices\n");
          return;
     }
     z->rows = x.rows; z->cols = x.cols; z->terms = 0;
     cx = 0; cy = 0;

     while (cx < x.terms && cy < y.terms)
     {
          ix = x.a[cx].row * x.cols + x.a[cx].col;
          iy = y.a[cy].row * x.cols + y.a[cy].col;

          if (ix < iy)
          {
               z->a[z->terms] = x.a[cx];
               (z->terms)++;
               cx++;
          }
          else
          {
               if (ix == iy)
               {
                    if (x.a[cx].value + y.a[cy].value)
                    {
                         struct Term t;
                         t.row = x.a[cx].row;
                         t.col = x.a[cx].col;
                         t.value = x.a[cx].value +
                                        y.a[cy].value;
```

```
                        z->a[z->terms] = t;
                        (z->terms)++;
                    }
                    cx++; cy++;
                }
                else
                {
                    z->a[z->terms] = y.a[cy];
                    (z->terms)++;
                    cy++;
                }
            }
        }
        /* copy remaining terms */
        for (; cx < x.terms; cx++)
        {
            z->a[z->terms] = x.a[cx];
            (z->terms)++;
        }
        for (; cy < y.terms; cy++)
        {
            z->a[z->terms++] = y.a[cy];
            (z->terms)++;
        }
}
```

Sample Run

```
Enter A:
Enter max rows and max cols: 2 2
Enter nonzero terms: 2
Enter Sparse Matrix (r,c,v):
0 0 7
1 1 9
Enter B:
Enter max rows and max cols: 2 2
Enter nonzero terms: 1
Enter Sparse Matrix (r,c,v):
1 1 2
The Sparse Matrix A is:
0 0 7
1 1 9
The Sparse Matrix B is:
1 1 2
The Sparse Matrix C = A + B is:
0 0 7
1 1 11
```