CHAPTER 2

DATA STRUCTURE

2.1 INTRODUCTION

Computer science can be defined as the study of data, its representation and transformation by a digital computer. The goal is to explore many different kinds of data objects. For each object, we consider the class of operations to be performed and then the way to represent this object so that these operations may be efficiently carried out. This implies a mastery of two techniques: the ability to devise alternate forms of data representation, and the ability to analyze the algorithm which operates on that representation. The pedagogical style we have chosen is to consider problems which have arisen often in computer applications. For each problem we will specify the data object or objects and what is to be accomplished. After we have decided upon a representation of the objects, we will give a complete algorithm and analyze its computing time. There are several terms we need to define carefully before we proceed. These include data structure, data object, data type and data representation. These four terminologies have no standard meaning in computer science circle, and they are often used interchangeably.

A data type is a term which refers to the kinds of data that variables may "hold" in a programming language. In FORTRAN the data types are INTEGER, REAL, LOGICAL, COMPLEX, and DOUBLE PRECISION. In PL/I there is the data type CHARACTER. The fundamental data type of SNOBOL is the
character string and in LISP it is the list (or S-EXPRESSION). With every programming language there is a set of built-in data types. This means that the language allows variables to name data of those built-in types and provide a set of operations which meaningfully manipulates these variables. Some data types are easy to provide because they are already built into the computer’s machine language instruction set. Integer and real arithmetic are examples of this. Other data types require considerably more effort to implement. In some languages, there are features which allow one to construct combinations of the built-in types. In COBOL and PL/I this feature is called a STRUCTURE while in PASCAL it is called a RECORD. However, it is not necessary to have such a mechanism. All the data structures we will see here can be reasonably built within a conventional programming language. Data object is a term referring to a set of elements, say D. For example the data object integers refers to \( D = \{0, \pm 1, \pm 2, \ldots\} \). The data object alphabetic character strings of length less than thirty one implies \( D = \{', 'A', 'B', \ldots, 'Z', 'AA', \ldots\} \). Thus D may be finite or infinite and if D is very large we may need to devise special ways of representing its elements in the computer.

**2.2 DATA STRUCTURE**

The notion of a data structure as distinguished from a data object is that we want to describe not only the set of objects, but the way they are related. Saying this in another way, we want to describe the set of operations which may legally be applied to elements of the data object. This implies that we must specify the set of operations and show how they work. For integers we would have the arithmetic operations +,-,*,/ and perhaps many others such as mod, greater than, less than, etc. The data object integers plus a
description of how +, -, *, /, etc. behave constitutes a data structure definition.

2.2.1 Definition

A data structure is a set of domains $D$, a designated domain $d \in D$, a set of functions $F$ and a set of axioms $A$. The triple $(D,F,A)$ denotes the data structure $d$ and it will usually be abbreviated by writing $d$. An implementation of a data structure $d$ is a mapping from $d$ to a set of other data structures $e$. This mapping specifies how every object of $d$ is to be represented by the objects of $e$. Secondly, it requires that every function of $d$ must be written using the functions of the implementing data structure $e$. Thus we say that integers are represented by bit strings, boolean is represented by zero and one, an array is represented by a set of consecutive words in memory.

2.3 ARRAYS

It is appropriate that we begin our study of data structures with the array. The array is often the only means for structuring data which is provided in a programming language. Therefore, it deserves a significant amount of attention. An array may be defined as a consecutive set of memory locations. It is true that arrays are generally implemented by using consecutive memory, but not always. Intuitively, an array is a set of pairs namely, index and value. For each index which is defined, there is a value associated with that index. In mathematical terms we call this as a correspondence or a mapping. However, we want to provide a more functional definition by giving the operations which are permitted on this data structure.
2.4 ORDERED LISTS

One of the simplest and most commonly found data object is the ordered or linear list. Examples are the days of a week

(MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY & SUNDAY)

or the values in a card deck

(2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King and Ace)

or the floors of a building

(basement, lobby, mezzanine, first, second, third, etc.)

or the years the United States fought in World War II

(1941, 1942, 1943, 1944 and 1945).

If we consider an ordered list more abstractly, we say that it is either empty or it can be written as \((a_1, a_2, a_3 \ldots, a_n)\). Where the \(a_i\) are atoms from some set \(S\).

There are a variety of operations that are performed on these lists. These operations include:

i. find the length of the list, \(n\);
ii. read the list from left-to-right (or right-to-left);
iii. retrieve the \(i\)-th element, \(1 \leq i \leq n\);
iv. store a new value into the \(i\)-th position, \(1 \leq i \leq n\);
v. insert a new element at position \(i\), \(1 \leq i \leq n+1\) causing elements numbered \(i, i+1, \ldots, n\) to become numbered \(i+1, i+2, \ldots, n+1\);
vi. delete the element at position i, 1 ≤ i ≤ n causing elements numbered i+1,...n to become numbered i,i+1,...n-1

Perhaps the most common way to represent an ordered list is by an array where we associate the list element $a_i$ with the array index i. This we will refer to as a sequential mapping, because using the conventional array representation we are storing $a_i$ and $a_{i+1}$ into consecutive locations i and i+1 of the array. This gives us the ability to retrieve or modify the values of random elements in the list in a constant amount of time, essentially because a computer memory has random access to any word. We can access the list element values in either direction by changing the subscript values in a controlled way. It is only operations (v) and (vi) which require real effort. Insertion and deletion using sequential allocation forces us to move some of the remaining elements so the sequential mapping is preserved in its proper form.

2.5 SPARSE MATRICES

A matrix is a mathematical object which arises in many physical problems. We are interested in studying ways to represent matrices so that the operations to be performed on them can be carried out efficiently. A general matrix consists of m rows and n columns of numbers as in Figure 2.1.

\[
\begin{bmatrix}
5 & 3 & 7 \\
-2 & 1 & 8 \\
5 & 1 & 4
\end{bmatrix}
\quad
\begin{bmatrix}
0 & 1 & 0 & 2 \\
5 & 0 & 2 & 0 \\
3 & 4 & 0 & 0 \\
0 & 2 & 0 & 0
\end{bmatrix}
\]

Figure 2.1 Example of 2 matrices
The first matrix has three rows and three columns, the second has four rows and four columns. In general, we write $m \times n$ (read $m$ by $n$) to designate a matrix with $m$ rows and $n$ columns. Such a matrix has $m \times n$ elements. When $m$ is equal to $n$, we call that as a square matrix.

It is very natural to store such a matrix in a two dimensional array, say $A(1:m, 1:n)$. Then we can work with any element by writing $A(i,j)$; and this element can be found very quickly. Now if we look at the second matrix of Figure 2.1 we see that it has many zero entries. Such a matrix is said to be sparse. There is no precise definition of when a matrix is sparse and when it is not, but it is a concept which we can all intuitively recognize. In matrix B, only seven out of 16 possible elements are nonzero and that is sparse! Any sparse matrix requires an alternate form of representation. This comes about because in practice many of the matrices we want to deal with are large, e.g., 1000 x 1000, but at the same time they are sparse: say only 1000 out of one million elements are possibly nonzero. Therefore there is justification to look for an alternate representation of sparse matrices.

2.6 LINKED LISTS

So far we have seen the representation of simple data structures using an array and a sequential mapping. These representations had the property that successive elements of the data object were stored at a fixed distance apart. Thus,

i. if the element $a_i$ of a table was stored at location $L_0$ then $a_{i+1}$ was at the location $L_0 + c$ for some constant $c$;
2.6.1 Singly linked lists

Instead of keeping a linear list in sequential memory locations, we can make use of a much more flexible scheme in which each node (a set of one or more words of the computer memory constitutes a node) contains a link to the next node of the list.

<table>
<thead>
<tr>
<th>Sequential allocation:</th>
<th>linked allocation:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>Contents</td>
</tr>
<tr>
<td>L + c:</td>
<td>Item 1</td>
</tr>
<tr>
<td>L + 2c:</td>
<td>Item 2</td>
</tr>
<tr>
<td>L + 3c:</td>
<td>Item 3</td>
</tr>
<tr>
<td>L + 4c:</td>
<td>Item 4</td>
</tr>
<tr>
<td>L + 5c:</td>
<td>Item 5</td>
</tr>
</tbody>
</table>

Here A, B, C, D and E are arbitrary locations in the memory, and Ø is the null link. The program which uses this table in the case of sequential allocation would have an additional variable or constant whose value indicates that the table is five items in length, or else this information would be specified by a "sentinel" code Item 5 or in the following location. A program for linked allocation would
have a link variable or constant that points to A, and from A all other items of the list can be found.

Links are often shown simply by arrows, since the actual memory locations occupied are usually irrelevant. The linked table above might therefore be shown as follows:

```
FIRST
```

| ← Item 1 → | ← Item 2 → | ← Item 3 → | ← Item 4 → | ← Item 5 → |

Here FIRST is a link variable pointing to the first node of the list. There are several obvious comparisons we can make between these two basic forms of storages.

i. Linked allocation takes up additional memory space for the links. This can be the dominating factor in some situations. However, we frequently find that the information in a node does not take up a whole word anyway, so there is already space for a link field present. Also, it is possible in many applications to combine several items into one node so that there is only one link for several items of information. But even more importantly, there is often an implicit gain in storage by the linked memory approach, since tables can overlap, sharing common parts; and in many cases, sequential allocation will not be as efficient as linked allocation unless a rather large number of additional memory locations are left vacant.

ii. It is easy to delete an item from within a linked list. For example, to delete item 3 we need only change the link associated with item 2. But with sequential allocation such a deletion generally
implies moving a large part of the list up into different locations.

iii. It is easy to insert an item into the midst of a list when the linked scheme is being used. For example, to insert an item 2.5 in between Item 2 and Item 3, we need to change only two links:

```
Item 1 <-> Item 2 -> Item 3 <-> Item 4 <-> Item 5
```

```
| Item 2.5 |
```

By comparison, this operation would be extremely time-consuming in a long sequential list.

iv. References to random parts of the list are much faster in the sequential case. To gain access to the kth item in the list, when k is a variable, it takes a fixed time in the sequential case, but it takes k iterations to march down to the right place in the linked case. Thus the usefulness of linked memory is predicted on the fact that in the large majority of applications we want to walk through lists sequentially, but not randomly; if items in the middle or at the bottom of the list are needed, we try to keep an additional link variable or list of link variables pointing to the proper places.

v. The linked scheme makes it easier to join two lists together or to break one into parts.
The linked scheme lends itself immediately to more intricate structures than simple linear lists. We can have a variable number of variable size lists; any node of the list may be a starting point for another list; the nodes may simultaneously be linked together in several orders corresponding to different lists and so on.

If the linked list is no longer used, then the whole list can be erased and that portion of the memory can be used for some other purpose which is a feature not possible in data structures described earlier.

Thus we see that the linking technique, which frees us from any constraints imposed by the consecutive nature of computer memory, gives us a great deal of more efficiency in some operations, while we lost some capabilities in other cases. It is usually clear which allocation technique will be most appropriate in a given situation, and often both methods are used in different lists of the same program.

2.6.2 Circular lists

A slight change in the manner of linking furnishes us with an important alternative to the methods of the preceding section.

A circularly-linked list (briefly: a circular list) has the property that its last node links back to the first instead of to $\emptyset$. It is then possible to access all of the list starting at any given point; we also achieve an extra degree of symmetry, and if we choose we need not think of the list as having a "last" or "first" node. It is quite easy to understand if the list is regarded as a circle instead of a
straight line with connected ends. The following situation is typical:

![Circularly linked list diagram](image)

**Figure 2.2 Circularly linked list**

Thus we see that a circular list can be used not only to represent inherently circular structures but also to represent linear structures; a circular list with one pointer to the rear node is essentially equivalent to a straight linear list with two pointers to the front and rear. The natural question to ask, in connection with this observation is, how do we find the end of the list, in view of the circular symmetry? There is no $\emptyset$ link to signal the end. The answer is that if we are performing some operations while moving through the list from one node to the next, we should stop when we get back to our starting place.

An alternate solution to the problem just posed is to put a special, recognizable node into each circular list, as a convenient stopping place. This special node is called the list head or the head node and in applications we often find it is quite convenient to insist that every circular list has exactly one node which is its list head. One advantage is that the circular list will then never be empty. Figure 2.2 now becomes
Figure 2.3 Circularly linked list with head node

Instead of pointers to the right end references to lists like Figure 2.3 are usually made via the list head.

2.6.3 Representation of sparse matrices

We have seen that when matrices are sparse (i.e., many of the entries are zero), much space and computing time could be saved if only the nonzero terms were retained explicitly. In the case where these nonzero terms did not form any "nice" pattern such as a triangle or a band, we can devise a sequential scheme in which each nonzero term is represented by a node with three fields: row, column and value. These nodes are sequentially organized. However, as matrix operations such as addition, subtraction and multiplication are performed, the number of nonzero terms in matrices will vary, matrices representing partial computations (as in the case of polynomials) will be created and will have to be destroyed later on to make space for further matrices. Thus the inadequacies of memory could be avoided. In this section we shall study a very general linked list scheme for sparse matrix representation. As we have already seen, linked schemes facilitate efficient representation of varying size structures and here, too, our scheme will overcome the afore mentioned shortcomings of the sequential representation.
2.6.4 Multiply linked list

In this data representation each column of a sparse matrix will be represented by a circularly linked list with a head node. In addition each row will also be a circularly linked list with a head node. Each node in the structure other than a head node will represent a nonzero term in the matrix and will be made up of five fields: ROW, COL, DOWN, RIGHT and VALUE. The DOWN field will be used to link to the next nonzero element in the same column, while the RIGHT field will be used to link to the next nonzero element in the same row. Thus if \( a_{ij} \) not equal to 0, then there will be a node with VALUE field \( a_{ij} \), ROW field \( i \) and COL field \( j \). This node will be linked into the circular linked list for row \( i \) and also into the circular linked list for column \( j \). It will, therefore, be a member of two lists at the same time and this representation is called "MULTIPLY LINKED LIST".

In order to avoid having nodes to two different sizes in the system, we shall assume head nodes to be configured exactly as nodes being used to represent the nonzero terms of the sparse matrix. The ROW and COL fields of head nodes will be set to zero (i.e. we assume that the rows and columns of our matrices have indices > 0). Figure 2.4 shows the structure obtained for the 4 X 4 sparse matrix, \( B \) of Figure 2.1.

For each nonzero term of \( B \), we have one five field node which is exactly one column list and one row list. The head nodes are marked as HN. If we wish to represent a \( n \times m \) sparse matrix of \( r \) nonzero terms, then the number of nodes needed is \( (r + n + m) \). Each node may require two to three words of memory and the total storage needed will be less than \( n \times m \) for sufficiently small \( r \).
Figure 2.4 Multiply linked list
Having arrived at this representation for sparse matrices, let us see how to manipulate it to perform efficiently some of the common operations on matrices. We shall present algorithms to read a sparse matrix and set up its linked list representation and to erase a sparse matrix (i.e. to return all the nodes to the available space list). The algorithms will make use of the utility algorithm GETNODE(X) to get nodes from the available space list.

To begin with, let us look at how to develop the algorithm MREAD(B) to read in and create the sparse matrix B. We shall assume that the input consists of m, the number of rows of B, n the number of columns of B, and r the number of nonzero terms as r triples of the form (i, j, a_{ij}). These triples consists of the row, column and value of the nonzero term. It is also assumed that the triples are ordered by rows and that within each row, the triples are ordered by columns. For example, the input for the 4 x 4 sparse matrix of Figure 2.2, which has 7 nonzero terms, would take the form: 1,2,1;1,4,2;2,1,5;2,3,2;3,1,3;3,2,4;4,2,2. We shall not concern ourselves here with the actual format of this input on the input media (cards, disk, etc.) but shall just assume we have some mechanism to get the triple. This algorithm makes use of two pointer arrays which contain the address of row head nodes and column head nodes. Using these head nodes all the row lists and column lists can be accessed. To start with, the right pointer of the row head nodes and Down pointer of the column head nodes point to themselves. Thus as such we have m rows and n columns of such self pointing lists.(i.e. this matrix has m row lists and n column lists rightnow). Once a data item is read, a new node is created and that is inserted into the i-th row list and j-th column list simultaneously. Thus row list and column lists are built up simultaneously.
2.6.5 Matrix multiplication using multiply linked list

Now let us see how matrix multiplication operation is performed when this representation is used. Assume that two matrices are created using multiply linked list as we have seen in the previous paragraph. When two matrices are multiplied a new matrix will be obtained. Hence before starting the multiplication, this algorithm proceeds by setting up head nodes for row lists and column lists for the resultant matrix.

During multiplication each row of the multiplicand matrix and corresponding column of the multiplier matrix are scanned and multiplication is performed. i.e. only non-zero entries are accessed. When the column subscript of a non-zero entry in a row of multiplicand matrix and the row subscript of non-zero entry of a column of the multiplier matrix matches, then multiplication is performed. Otherwise we have to seek for the next match. While seeking for the next match we will meet the following situations:

**Case 1**: If the minimum of these two subscripts is the column subscript of the multiplicand matrix, then we need to consider the possibility of a match for the row subscript of the multiplier matrix (which is currently read) with the next non zero entry in the column subscript of multiplicand. Hence we need to scan the next non zero entry for the next column subscript of multiplicand.

**Case 2**: If the minimum of these two subscripts is the row subscript of multiplier, then we need to scan the next non zero entry in the row subscript of case 1.
Considering case 1, the next non zero entry in the row of multiplier is accessed. If $j$ is the present column subscript and $j'$ is the previous column subscript of the multiplicand, then we meet any one of the following cases:

**Case 1.1**: $j' < j \leq n$. In this case the column subscript is greater than the previous column subscript and less than or equal to the number of columns ($n$) of multiplier. Now comparison is made for matching. If they match then multiplication is performed, otherwise we proceed with our next scanning as described above.

**Case 1.2**: $j$ refers to the head node. This refers to the end of all the entries of this row. In this case multiplication of this particular row of multiplicand and that particular column of multiplier is stopped and is proceeded to the next row.

Considering case 2, when we scan the next non zero entry for the next row subscript of multiplier, we again meet with two possible cases which are analogues to cases 1.1 and 1.2. If the resulting value happens to be non-zero then a new node will be created and linked to the corresponding row list and column list of the resultant matrix. Similarly, these operations are continued until all the rows of the multiplicand matrix and all the columns of the multiplier matrix are exhausted. Thus a new matrix using multiply linked list is formed.
Example:

Consider the matrices $A$ and $B$ shown in Figure 2.5.

\[
A = \begin{bmatrix}
2 & 0 & 0 \\
0 & 0 & 1 \\
2 & 1 & 0
\end{bmatrix}
\quad B = \begin{bmatrix}
0 & 3 & 0 \\
5 & 0 & 1 \\
0 & 1 & 0
\end{bmatrix}
\]

Figure 2.5 Matrices

Let us say $C = A \times B$. Then

\[C_{11} = \sum_{i=1}^{3} a_{1i} b_{i1}\]

To get $C_{11}$, first row of the $A$ matrix will be multiplied by first column of $B$ matrix. Hence first row of $A$ matrix and first column of $B$ matrix are scanned for multiplication. In the multiply linked list representation, the first row of $A$ and first column of $B$ will look like:

As a first step nonzero entry in this first row of $A$ is accessed and non-zero entry in the first column of $B$ is accessed. Then the column subscript of the $A$ matrix entry is compared with the row subscript of the $B$ matrix entry. If
they match then multiplication is performed. In this case they do not match and the minimum subscript occurs in the row of $A$ and hence the next non-zero entry in the same row of matrix $A$ is accessed. But this happens to be the row head node. i.e. there is no more non zero entries in that row. Hence the multiplication of first row of $A$ and first column of $B$ is stopped and the value of $C_{11}$ equal to zero. In this case the other non-zero entries in the first column of $B$ are not accessed.

Since $C_{11}$ is zero, a node need not be created for $C_{11}$. Similarly the operations are performed until all the rows of $A$ and all the columns of $B$ are exhausted.

Using this multiply linked list representation, multiplication has the following advantages:

i. Only non-zero entries are accessed so that unnecessary checking and multiplication of zero entries are avoided.

ii. During multiplication some of the nodes are not accessed so that accessing time is considerably reduced.

iii. If the resultant value is zero then no node will be created for that entry in the resultant matrix, so that creation time, linking time and the memory requirement will be reduced.

Multiply linked list representation of the sparse matrices has the following drawbacks:
i. If a matrix is used for premultiplication, always the elements are accessed only by rowwise through head nodes. In this case the columnwise representation is no longer required. Hence during creation, the time to create columnwise headnodes and to link all the nodes columnwise will be wasted. To overcome this the matrix can be represented using rowwise representation. The rowwise representation for the matrix A in Figure 2.5 is shown in Figure 2.6. In this representation each non-zero entry is accessed through row head nodes and each node has three fields viz column number, value and Rlink which points to the next non-zero entry in that row. Here row number in each node is not necessary since it is known through the head node itself. Thus in this representation the creation time and memory space will be reduced by decreasing three fields from five fields in each node and eliminating column head node and the corresponding links.

ii. When a matrix is always used for post multiplication, it is necessary to have only columnwise representation. Due to the same reasons we have seen earlier this type of matrices can be represented using column-wise circularly linked list. The column-wise representation for the matrix B in Figure 2.5 is shown in Figure 2.7. In this representation each element has three fields viz row number, value and Dlink which points to the next non-zero entry in that column. Here column number is not necessary since it is known through the column head node. In this representation also the creation time and the memory requirement will be reduced.
Figure 2.6: Rowwise circularly linked list
HN: Head Node   RN: Row Number
VAL: Value of the element   DLINK: Pointer

Figure 2.7 Columnwise circular linked list
The resultant matrix also we are forming using multiply linked list representation. This can be represented using row wise or column wise depending on the further application of the matrix.

But some of the matrices have to be used for premultiplication as well as post multiplication purposes. Hence only those matrices can be represented using multiply linked list.

2.6.6 Matrix multiplication by a vector

Assume that a matrix is represented using linked list and vector is represented using an array. For multiplication each row of the matrix is scanned. If a non zero entry is accessed in a row, then only the corresponding element of the vector will be considered for multiplication. Thus the preceding elements of that vector are skipped automatically.

Consider the matrix A and the vector X shown in Figure 2.8.

\[
A = \begin{bmatrix}
  a_{11} & a_{12} & \ldots & a_{ij} & \ldots & a_{1n} \\
  a_{21} & a_{22} & \ldots & a_{2j} & \ldots & a_{2n} \\
  \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\
  \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\
  a_{ni} & a_{12} & \ldots & a_{nj} & \ldots & a_{nn}
\end{bmatrix}
\quad
X = \begin{bmatrix}
  x_1 \\
  x_2 \\
  \vdots \\
  x_j \\
  \vdots \\
  x_n
\end{bmatrix}
\]

Figure 2.8 Matrix and a Vector

Let \( C = AX \)

Then \( C_1 = \sum_{i=1}^{n} a_{i1} x_1 \)
For example if $a_{ij}$ is non zero in the $i$th row then $a_{ij}$ will be multiplied by $x_j$ by skipping $x_1$ to $x_{j-1}$ elements so that accessing time of those elements is reduced.

2.7 CONCLUSION

This chapter describes the concept of Data Structure and the advantages of linked list over sequential list are discussed. Nature of the sparse matrices and the efficient way of their storage are explained. The common operations involved in matrices using linked list concept are also discussed.