ARRAYS

INTRODUCTION

Data structures are classified as either linear or nonlinear.

A data structure is said to be linear if its elements form a sequence, or, in other words, a linear list.

There are two basic ways of representing such linear structures in memory.

1. By having a linear relationship between the elements represented by means of sequential memory locations. These linear structures are called arrays.
2. 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.

Nonlinear structures are trees and graphs.

The operations one normally performs on any linear structure whether it is an array or a linked list, are:

(a) Traversal - Processing each element in the list.

(b) Search- Finding the location of the element with a given value or the record with a given key.

(c) Insertion -Adding a new element to the list.

(d) Deletion - Removing an element from the list.

(e) Sorting - Arranging the elements in some type of order.

(f) Merging - Combining two lists into a single list

The particular linear structure that one chooses for a given situation depends on the relative frequency with which one performs these different operations on the structure .

LINEAR ARRAYS

A linear array is a list of a finite number n of homogeneous data elements (i.e., data elements the same type) such that:

(a) The elements of the array are referenced respectively by an index set consisting of n consecutive numbers.

(b) The elements of the array are stored respectively in successive memory locations.

The number n of elements is called the length or size of the array.

If not explicitly stated, we will assume the index set consists of the integers 1, 2, ..., n.

In general, the length or the number of data elements of the array can be obtained from the index set by the formula

$$\text{Length} = UB - LB + 1$$

where UB is the largest index, called the upper bound,
and LB is the smallest index, called the lower bound, of the array.

Note that length = UB when LB = 1.

The elements of an array A may be denoted by the subscript notation

$A_1, A_2, A_3, \ldots A_n$ or by the bracket notation (used in C)

The number K in A[K] is called a subscript or an index and A[K] is called a subscripted variable. Subscripts allow any element of A to be referenced by its relative position in A.

(a) Let DATA be a 6-element linear array of integers such that
DATA[1] = 247 DATA[2] = 56 DATA[3] = 429 DATA[4] = 135 DATA[5] = 87 DATA[6] = 156
Sometimes we will denote such an array by simply writing

DATA: 247, 56, 429, 135, 87, 156

The array DATA is frequently pictured as in Fig. 4.1(a) or Fig. 4.1(b).

	DATA
1	247
2	56
3	429
4	135
5	87
6	156

(a)

DATA

247	56	429	135	87	156
1	2	3	4	5	6

(b)

```
/*
Defining Arrays in C*/
#include <stdio.h>
main()
{
  int a[10]; //1
  for(int i = 0;i<10;i++)
  {
    a[i]=i;
  }
  printaray(a);
}
void printaray(int a[])
{
  for(int i = 0;i<10;i++)
  {
    printf("Value in the array %d\n",a[i]);
  }
}
```

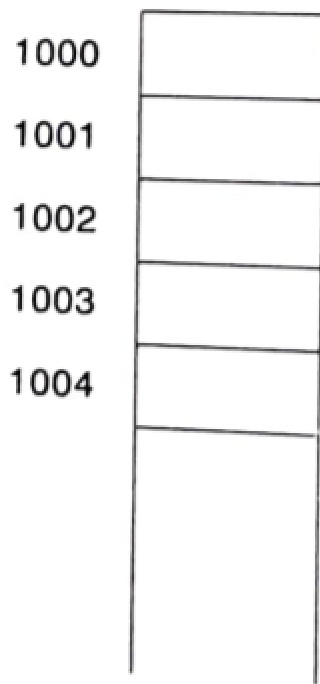
When we define the array, the size should be known.

Subscripts are used to refer the elements of the array where 0 is considered to be the lowest subscript always and the highest subscript is (size -1). which is 9 in this case. We can refer to any element as a[0], a[1], a[2] etc.

REPRESENTATION OF LINEAR ARRAYS IN MEMORY

Let LA be a linear array in the memory of the computer. The memory of the computer is a sequence of addressed locations as in Fig.

The notation LOC(LA[K]) = address of the element LA[K] of the array LA.



Elements of LA are stored in successive memory cells. The computer does not need to keep track of the address of every element of LA, but needs to keep track only of the address of the first element of LA, denoted by Base(LA) and called the base address of LA.

Using this address Base(LA), the computer calculates address of any element of LA by the following formula:

LOC (LA[K]) = Base(LA) + w(K - lower bound)   where w is the number of words per memory cell for the array LA.

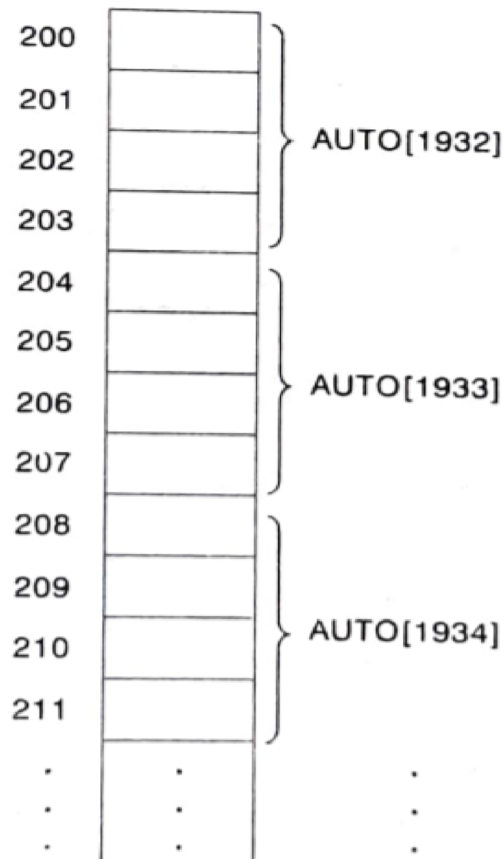
Given any subscript K, one can locate and access the content of LA[K] without scanning any other element of LA.

Consider the array AUTO in Example 4.1(b), which records the number of automobiles sold each year from 1932 through 1984. Suppose AUTO appears in memory as pictured in Fig. 4.4. That is, *Base*(AUTO) = 200, and $w = 4$ words per memory cell for AUTO. Then

$$LOC(AUTO[1932]) = 200, \quad LOC(AUTO[1933]) = 204, \quad LOC(AUTO[1934]) = 208, \ \ldots$$

The address of the array element for the year K = 1965 can be obtained by using Eq. (4.2):

$$\begin{aligned} LOC(AUTO[1965]) &= Base(AUTO) + w(1965 - \text{lower bound}) \\ &= 200 + 4(1965 - 1932) = 332 \end{aligned}$$



TRAVERSING LINEAR ARRAYS

Let A be a collection of data elements stored in the memory of the computer. Suppose we want to print the contents of each element of A or to count the number of elements of A with a given property. This can be accomplished by traversing A, that is, by accessing and processing (called visiting) each element of A exactly once.

Algorithm 4.1: (Traversing a Linear Array) Here LA is a linear array with lower bound LB and upper bound UB. This algorithm traverses LA applying an operation PROCESS to each element of LA.

1. [Initialize counter.] Set K := LB.
2. Repeat Steps 3 and 4 while K ≤ UB.
3. [Visit element.] Apply PROCESS to LA[K].
4. [Increase counter.] Set K := K + 1.

   [End of Step 2 loop.]
5. Exit.

```
/* C Implementation of Algorithm 4.1*/
#include<stdio.h>
#define N 7          /* lower bound LB of the array is 1 and
                        upper bound UB of the array is N*/
int main(void)
{ int k,i,          LA[N]={23, 45, 56, 1, -9, -12, 123};
  k=0;                  /* Initialize counter k*/
  while(k<=N)       /*PROCESS in the algorithm is implemented
  {                    as scaling the array elements by 2*/
    k++;            /* Increase counter*/
  }
  for( i=0;i<N;i++)
  {printf("\n%d", LA[i]);  /* print array elements after PROCESS*/
  }
  return 0;
}
```

We also state an alternative form of the algorithm which uses a repeat-for loop instead of the repeat-while loop.

Algorithm 4.1′: (Traversing a Linear Array) This algorithm traverses a linear array LA with lower bound LB and upper bound UB.

1. Repeat for K = LB to UB:
   Apply PROCESS to LA[K].
   [End of loop.]
2. Exit.

INSERTING AND DELETING

Let A be a collection of data elements in the memory of the computer. "Inserting" refers to the operation of adding another element to the collection A, and "deleting" refers to the operation of removing one of the elements from A.

Inserting an element at the "end" of a linear array can be easily done if the memory space allocated for the array is large enough to accommodate the additional element. But, suppose we need to insert an element in the middle of the array. Then, on the average, half of the elements must be moved downward to new locations to accommodate the new element and keep the order of the other elements.

Similarly, deleting an element at the "end" of an array presents no difficulties, but deleting an element somewhere in the middle of the array requires that each subsequent element moved one location upward in order to "fill up" the array.

"downward" refers to locations with larger subscripts, and the term "upward" refers to locations with smaller subscripts.

(Inserting into a Linear Array) INSERT (LA, N, K, ITEM)

Here LA is a linear array with N elements and K is a positive integer such that K <= N. This algorithm inserts an element ITEM into the Kth position in LA.

1. [Initialize counter.] Set J: = N.
2. Repeat Steps 3 and 4 while J >= K.
3. [Move Jth element downward.] Set LA[J + 1]:= LA[J]
4. [Decrease counter.] Set J := J - 1.

[End of Step 2 loop.]
5. [Insert element.] Set LA[K] := ITEM.
6. [Reset N.] Set N := N + 1.
7. Exit.

The following algorithm deletes the Kth element from a linear array LA and assigns it variable ITEM.

(Deleting from a Linear Array) DELETE(LA, N, K, ITEM)
Here LA is a linear array with N elements and K is a positive integer such that $K \leq N$. This algorithm deletes the Kth element from LA.

1. Set ITEM := LA[K].
2. Repeat for J = K to N − 1:
   [Move J + 1st element upward.] Set LA[J] := LA[J + 1].
   [End of loop.]
3. [Reset the number N of elements in LA.] Set N: = N − 1.
4. Exit.

Program for Insertion and deletion in an array

SORTING; BUBBLE SORT

Sorting means arranging numerical data in decreasing order or arranging non-numerical data in alphabetical order. That is some order.

Let A be a list of *n* numbers. *Sorting* A refers to the operation of rearranging the elements of A so they are in increasing order, i.e., so that,

$$A[1] < A[2] < A[3] < \ldots < A[N]$$

For example, suppose A originaily is the list

8, 4, 19, 2, 7, 13, 5, 16

After sorting, A is the list

2, 4, 5, 7, 8, 13, 16, 19

Suppose the list of numbers A[1], A[2], ..., A[N] is in memory. The bubble sort algorithm works as follows:

Step 1. Compare A[1] and A[2] and arrange them in the desired order, so that A[1] < A[2]. Then compare A[2] and A[3] and arrange them so that A[2] < A[3]. Then compare A[3] and A[4] and arrange them so that A[3] < A[4]. Continue until we compare A[N – 1] with A[N] and arrange them so that A[N – 1] < A[N].

Observe that Step 1 involves $n - 1$ comparisons. (During Step 1, the largest element is "bubbled up" to the nth position or "sinks" to the nth position.) When Step 1 is completed, A[N] will contain the largest element.

Step 2. Repeat Step 1 with one less comparison; that is, now we stop after we compare and possibly rearrange A[N – 2] and A[N – 1]. (Step 2 involves N – 2 comparisons and, when Step 2 is completed, the second largest element will occupy A[N – 1].)

Step 3. Repeat Step 1 with two fewer comparisons; that is, we stop after we compare and possibly rearrange A[N – 3] and A[N – 2].

. ...

. ...

. ...

Step N – 1. Compare A[1] with A[2] and arrange them so that A[1] < A[2].

After $n - 1$ steps, the list will be sorted in increasing order.

The process of sequentially traversing through all or part of a list is frequently called a "pass," so each of the above steps is called a pass. Accordingly, the bubble sort algorithm requires $n - 1$ passes, where n is the number of input items,

Eg: 32, 51, 27, 85, 66, 23, 13, 57

Algorithm 4.4: (Bubble Sort) BUBBLE(DATA, N)
Here DATA is an array with N elements. This algorithm sorts the elements in DATA.

1. Repeat Steps 2 and 3 for K = 1 to N – 1.
2. Set PTR := 1. [Initializes pass pointer PTR.]
3. Repeat while PTR ≤ N – K: [Executes pass.]
   (a) If DATA[PTR] > DATA[PTR + 1], then:
   Interchange DATA[PTR] and DATA[PTR + 1].
   [End of If structure.]
   (b) Set PTR := PTR + 1.
   [End of inner loop.]
   [End of Step 1 outer loop.]
4. Exit.

Program for Bubble sort

Complexity of the Bubble Sort Algorithm

Traditionally, the time for a sorting algorithm is measured in terms of the number of comparisons. The number $f(n)$ of comparisons in the bubble sort is easily computed. Specifically, there are $n - 1$ comparisons during the first pass, which places the largest element in the last position; there are $n - 2$ comparisons in the second step, which places the second largest element in the next-to-last position; and so on. Thus

$$f(n) = (n-1) + (n-2) + \dots + 2 + 1 = \frac{n(n-1)}{2} = \frac{n^2}{2} + O(n) = O(n^2)$$

In other words, the time required to execute the bubble sort algorithm is proportional to n^2, where n is the number of input items.

SEARCHING; LINEAR SEARCH

Let DATA be a collection of data elements in memory, and suppose a specific ITEM of information is given.

Searching refers to the operation of finding the location LOC of ITEM in DATA, or printing some message that ITEM does not appear there. The search is said to be successful if ITEM does appear in DATA and unsuccessful otherwise.

Suppose DATA is a linear array with n elements. To search for a given ITEM in DATA is to compare ITEM with each element of DATA one by one. That is, first we test whether DATA[1] =

ITEM, and then we test whether DATA[2] = ITEM, and so on. This method, which traverses DATA sequentially to locate ITEM, is called linear search or sequential search.

To simplify the matter, we first assign ITEM to DATA[N + 1], the position following the last element of DATA. Then the outcome

LOC = N +1

Where LOC denotes the location where ITEM first occurs in DATA, signifies the search is unsuccessful. The purpose of this initial assignment is to avoid repeatedly testing whether or not we have reached the end of the array DATA. This way, the search must eventually "succeed".

Algorithm 4.5: (Linear Search) LINEAR(DATA, N, ITEM, LOC)
Here DATA is a linear array with N elements, and ITEM is a given item of information. This algorithm finds the location LOC of ITEM in DATA, or sets LOC := 0 if the search is unsuccessful.

1. [Insert ITEM at the end of DATA.] Set DATA[N + 1] := ITEM.
2. [Initialize counter.] Set LOC := 1.
3. [Search for ITEM.]
   Repeat while DATA[LOC] ≠ ITEM:
      Set LOC := LOC + 1.
   [End of loop.]
4. [Successful?] If LOC = N + 1, then: Set LOC := 0.
5. Exit.

Program for Linear Search

```c
#include <stdio.h>

int main()
{
  int array[100], search, c, n;

  printf("Enter number of elements in array\n");
  scanf("%d", &n);

  printf("Enter %d integer(s)\n", n);

  for (c = 0; c < n; c++)
    scanf("%d", &array[c]);

  printf("Enter a number to search\n");
  scanf("%d", &search);

  for (c = 0; c < n; c++)
  {
    if (array[c] == search)    /* If required element is found */
    {
      printf("%d is present at location %d.\n", search, c+1);
      break;
    }
  }
  if (c == n)
    printf("%d isn't present in the array.\n", search);

  return 0;
}
```

BINARY SEARCH

Suppose DATA is an array which is sorted in increasing numerical order or alphabetically. There is an extremely efficient searching algorithm, called binary search which can be used to find the location LOC of a given ITEM of information in DATA.

Suppose we want to find the location of some name in a telephone directory (or some word m a dictionary). Here we do not perform a linear search. We open the directory in the middle to determine which half contains the name. Then opens that half in the middle to determine which quarter of the directory contains the name. Then opens that quarter in the middle to determine

which eighth of the directory contains the name. And so on until we find the location of the name, by reducing (very quickly) the number possible locations for it in the directory

During each stage of the algorithm, search for ITEM Is reduced to a segment of elements of DATA:

DATA[BEG], DATA[BEG + 1], DATA[BEG +2], ……..., DATA[END]

BEG and END denotes, respectively, the beginning and end locations of the segment under consideration.

The algorithm compares ITEM with the middle element DATA[MID] of the segment, where MID is obtained by

MID = INT((BEG + END)/2)

If DATA[MID] = ITEM, then the search is successful and set LOC = MID. Otherwise a new segment of DATA is obtained as follows:

(a) If ITEM < DATA[MID], then ITEM can appear only in the left half of the segment:
DATA[BEG], DATA[BEG + 1],…..... DATA[MID - 1]

Reset END := MID - 1 and begin searching again

(b) If ITEM> DATA[MID], then ITEM can appear only in the right half of the segment:

DATA[MID + 1],DATA[MID +2],…….... DATA[END]

Reset BEG := MID + I and begin searching again.

Initially, we begin with the entire array DATA; i.e., we begin with BEG = 1 and END = n, or with BEG = LB and END = UB.

If ITEM is not in DATA, then we obtain END <BEG. This condition signals that the search is unsuccessful, and in such a case we assign LOC = NULL

(Binary Search) BINARY(DATA, LB, UB, ITEM, LOC)

Here DATA is a sorted array with lower bound LB and upper bound UB, and ITEM is a given item of information. The variables BEG, END and MID denote, respectively, the beginning, end and middle locations of a segment of elements of DATA. This algorithm finds the location LOC of ITEM in DATA or sets LOC = NULL.

1. [Initialize segment variables.]
   Set BEG := LB, END := UB and MID = INT((BEG + END)/2).
2. Repeat Steps 3 and 4 while BEG ≤ END and DATA[MID] ≠ ITEM.
3. If ITEM < DATA[MID], then:
   Set END := MID – 1.
   Else:
   Set BEG := MID + 1.
   [End of If structure.]
4. Set MID := INT((BEG + END)/2).
   [End of Step 2 loop.]
5. If DATA[MID] = ITEM, then:
   Set LOC := MID.
   Else:
   Set LOC := NULL.
   [End of If structure.]
6. Exit.

(1) (11,) 22, 30, 33, 40, 44, [55,] 60, 66, 77, 80, 88,(99)

(2) 11, 22, 30, 33, 40, 44, 55,(60,) 66, [77,] 80, 88,(99)

(3) 11, 22, 30, 33, 40, 44, 55, 60, 66, 77,(80,) [88,] (99)

(4) 11, 22, 30, 33, 40, 44, 55, 60, 66, 77,(80), 88, 99 [Unsuccessful]

Fig. 4.8 Binary Search for ITEM = 85

Program for Binary search

```
int main()
{
  int c, first, last, middle, n, search, array[100];

  printf("Enter number of elements\n");
  scanf("%d", &n);

  printf("Enter %d integers\n", n);

  for (c = 0; c < n; c++)
    scanf("%d", &array[c]);

  printf("Enter value to find\n");
  scanf("%d", &search);

  first = 0;
  last = n - 1;
  middle = (first+last)/2;

  while (first <= last) {
    if (array[middle] < search)
      first = middle + 1;
    else if (array[middle] == search) {
      printf("%d found at location %d.\n", search, middle+1);
      break;
    }
    else
      last = middle - 1;

    middle = (first + last)/2;
  }
  if (first > last)
    printf("Not found! %d isn't present in the list.\n", search);

  return 0;
}
```

MULTIDIMENSIONAL ARRAYS

The linear arrays are also called one-dimensional arrays, since each element in the array is referenced by a single subscript. Most programming languages allow two-dimensional and three-dimensional arrays, i.e.. arrays where elements are referenced, respectively by two and three subscripts.

Two-Dimensional Arrays

A two-dimensional m x n array A is a collection of m. n data elements such that each element is specified by a pair of integers (such as J, K), called subscripts, with the property that

1<=J <=m and 1<=K<=n

The element of A with first subscript J and second subscript K will be denoted by $A_{j,k}$ or A[J, K] .

Two-dimensional arrays are called matrices or tables.

A two-dimensional array is a list of one dimensional arrays. To declare a two-dimensional integer array of size x.y, we use

type arrayName[x][y]

Where type can be any valid C data type and arrayName will be a valid identifier. A two-dimensional array can be considered as a table with x number of rows and y number of columns.

```
int A[3][4];
```

Here, *A* is a two-dimensional array. This 2D array can hold 12 elemen
You can think this array as a table with 3 row and each row has
columns.

A[0][0]	A[0][1]	A[0][2]	A[0][3]
A[1][0]	A[1][1]	A[1][2]	A[1][3]
A[2][0]	A[2][1]	A[2][2]	A[2][3]

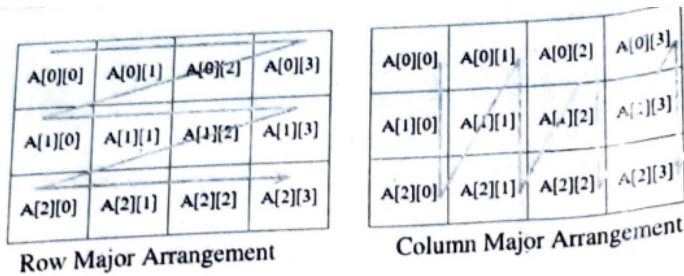
Similarly, one can declare a three-dimensional (3D) array.

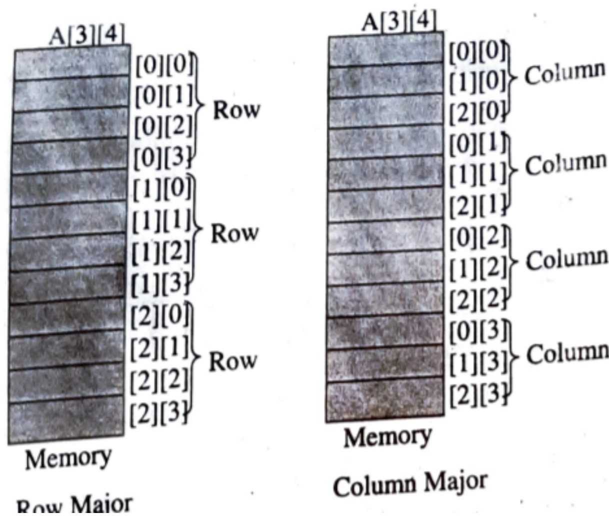
For example float y[10][3][4];

Here, the array y can hold 120 elements. A 3D array is essentially an array of 2D arrays. You can think of this as 10 matrices each with 3 rows and 4 columns. Hence, the total number of elements is 120.

A two dimensional array A[3][4] though A is pictured as a rectangular array of elements with m rows and n columns, is represented as a block of mxn sequential memory locations and is basically a linear storage.

There are two possible arrangements of elements in memory row major arrangement and column major arrangement. The difference between the orders lies in which elements of an array are contiguous in memory. In a row-major order, the consecutive elements of a row reside next to each other i.e. row wise, whereas the same holds true to consecutive elements of a column in a column-major order, column wise.





<u>Row Major</u>

We can calculate the address of the element of the mth row and the nth column in a two-dimensional array using the formula:

addr(a[m, n]) = (total number of rows present before the mth row x size of a row) + (total number of elements present before the nth element in the mth row x size of element)

In the above equation:

The total number of rows present before mth row = (m – lb1) and lb1 is a first dimensional lower bound.

Size of a row = total number of elements present in a row x size of an element.

Total number of elements in a row is calculated using (ub2-lb2 + 1) and ub2 and lb2 are the second dimensional upper and lower bounds.

All the variables present the above equation can be written in a simple form as:

addr(a[m, n]) = ((m – lb1) x (ub2 - lb2 + 1) x size) + ((n - lb2) x size)

Column-major Representation

We can also represent a two-dimensional array as one single row of columns. Such a representation is called a column-major representation

We can calculate the address of the element of the mth row and the nth column in a two-dimensional array using the formula:

addr(a[m, n]) = (total number of columns present before the nth column x size of a column) + (total number of elements present before the mth element in the nth column x size of element)

In the above equation:

The total number of columns present before nth column = (n – lb2) and lb2 is a second dimensional lower bound.

Size of a column = total number of elements present in a column x size of an element.

Total number of elements in a column is calculated using (ub1-lb1 + 1) and ub1 and lb1 are the first dimensional upper and lower bounds.

All the variables present the above equation can be written in a simple form as:

addr(a[m, n]) = ((n – lb2) x (ub1 – lb1 + 1) x size) + ((m – lb1) x size)

SPARSE MATRICES

Matrices with a relatively high proportion of zero entries are called sparse matrices. Two general types of n-square sparse matrices, which occur in various applications, are pictured in Fig . The first matrix. where all entries above the main diagonal are zero or, equivalently, where nonzero entries can only occur on or below the main diagonal, is called a (lower) triangular matrix. The second matrix, where nonzero entries can only occur on the diagonal or on elements immediately above or below the diagonal, is called a tridiagonal matrix.

$$\begin{pmatrix} 4 & & & & \\ 3 & -5 & & & \\ 1 & 0 & 6 & & \\ -7 & 8 & -1 & 3 & \\ 5 & -2 & 0 & 2 & -8 \end{pmatrix}$$

(a) Triangular matrix

$$\begin{pmatrix} 5 & -3 & & & & & \\ 1 & 4 & 3 & & & & \\ & 9 & -3 & 6 & & & \\ & & 2 & 4 & -7 & & \\ & & & 3 & -1 & 0 & \\ & & & & 6 & -5 & 8 \\ & & & & & 3 & -1 \end{pmatrix}$$

(b) Tridiagonal matrix

The natural method of representing matrices in memory as 2D arrays may not be suitable for sparse matrices. That is, we can save space by storing only those entries which may be nonzero

```
void main()
{
    int S[10][10],m,n,i,k=0,count=0;
    printf("Enter number of rows in the matrix : ");
    scanf("%d",&m);
    printf("Enter number of columns in the matrix : ");
    scanf("%d",&n);
    printf("Enter elements in the matrix : ");
     for ( i = 0; i < m; i++)
        for ( j = 0; j < n; j++)
            scanf("%d",&S[i][j]);

    printf("The matrix is \n");
    for ( i = 0; i < m; i++)
     {
        for ( j = 0; j < n; j++)
       {
```

```
        printf(" %d ",S[i][j]);
        if (S[i][j] != 0)
            count++;
    }
    printf("\n");
}
int M[3][size];

for ( i = 0; i < m; i++)
    for ( j = 0; j < n; j++)
        if (S[i][j] != 0)
        {
            M[0][k] = i;
            M[1][k] = j;
            M[2][k] = S[i][j];
            k++;
        }

printf("Triplet representation of the matrix is \n");
for (int i=0; i<3; i++)
{
    for (int j=0; j<size; j++)
        printf(" %d ", M[i][j]);

    printf("\n");
}
return 0;
}
```

```
#include<stdio.h>
void main( )
{
     int m,n,i,j,a[10][10],b[10]10],l,count;
     clrscr();
     printf("Enter order of the matrix\n");
     scanf("%d%d",&m,&n);
     printf("Enter elements in the matrix : ");
     for ( i = 0; i < m; i++)
        for ( j = 0; j < n; j++)
            scanf("%d",&a[i][j]);
 l=1;
count=0;
for ( i = 0; i < m; i++)
{
       for ( j = 0; j < n; j++)
       {
            if(a[i][j]!=0)
        {
              b[l][0]=i;
              b[l][1]=j;
              b[l][2]=a[i][j];
              l++;
              count++;
      }
}
}
b[0][0]=m;
b[0][1]=n;
b[0][2]=count;
```

```
printf("Sparse matrix in triplet representation\n");
for ( i = 0; i < m; i++)
 {
      for ( j = 0; j < n; j++)
      {
          printf("%d\t",b[i][j]);
      }
      printf("\n");
}
}
```

REPRESENTATION OF POLYNOMIALS USING ARRAYS

Sometimes we will need to evaluate several polynomial expressions and perform basic arithmetic operations like addition, multiplication, etc. on them. The easiest way to represent a polynomial of degree 'n' is by storing the coefficient of (n+1) terms of a polynomial in an array. An expression $x+4x^2-7x^5$ is a polynomial of degree 5.

All the elements of an array have two values, coefficient and exponent. Assume that exponent of each successive term is less than that of the previous term. Once we build an array for polynomials, we can use it to perform various operations including addition and multiplication. A single-dimensional array is used for representing a single variable polynomial. The index of such an array can be considered as an exponent and the coefficient can be stored at that particular index.

The polynomial expression $3x^4 + 5x^3 + 6x^2 + 10x - 14$ can be stored in a single-dimensional array as shown in Fig. 4.16.

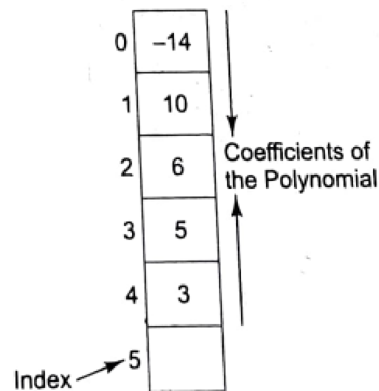


Fig. 4.16 Polynomial Representation Using an Array

Drawbacks.

Suppose the exponents is too large, then the size of the array will become more. For instance, if we have something like $4x^{999}$, in that case, we will have to store the coefficient 4 at index 999 in the array, and the array size will have to be 1000. Scanning such a large array-it will be **time consuming**.

Wastage of space. Suppose you have a polynomial $6x^{99}$-5,then only two elements will be stored in the array of size 100. The remaining space will be empty and therefore not utilized.

A third problem - **cannot decide the size of the array**. Suppose we have array of size 15, and the exponent value of the polynomial is 50, then we cannot store exponent value and we will have to change the array size.

Addition of 2 polynomials:

Let there be 2 polynomials A and B. The polynomial storing the addition result is stored in one large array C and i, j, and k represents the pointers to the polynomials and arrays respectively.

1. Set i to point to first term in A.
2. Set j to point to first term in B.
3. Set k to point first position in C.
4. Read n1= no. of terms in A, n2 = no. of terms in B
5. While i < n1 and j < n2, then
   (a) If exponent at ith position of A = exponent at jth position of B, then C at kth position = coeff of A at ith position + coeff of B at jth position.

Increment i, j, k to point to next position in A, B, C

(b) Else

If exponent at ith position of A < exponent at jth position of B, then

Copy coeff at ith position from A to coeff kth position in C.

Increment i and k

Else

Copy coeff at jth position from B into coeff kth at position k in C.

Increment j and k to point to next position in B, C

6. While i<n1
   (a) Copy coeff at ith position of the coeff field into coeff at k position in C
   (b) Copy exponent pointed by i into exponent field at kth position in C
   (c) i and k are incremented to point to the next position in arrays A and C
7. While j< n2
   (a) Copy coeff at jth position of B into coeff at kth position in C
   (b) Copy exponent pointed by j into exponent field at k position in C
   (c) j, k are incremented to point to next position in B and C arrays
8. Display C [complete array as the addition of two polynomials A and B]

   Exit

Program for Addition of 2 polynomials

```
void main()
{
int A[6];
int B[6];
int C[6];
int i,flag=0;
clrscr();

for(i=0;i<6;i++)
 A[i]=B[i]=0;

A[1]=1;
A[2]=4;
A[5]=-7;

B[0]=-14;
B[1]=10;
B[2]=6;
B[3]=5;
B[4]=3;

printf("Polynomial 1 = ");
for(i=5;i>=0;i—)
{
 if(A[i]!=0 && i>0)
 {
    printf("%d.(x)%d ",A[i],i);
    flag=1;
 }

 if(i>0 && A[i-1]>0 && flag==1)
     printf("+ ");

 if(A[i]!=0 && i==0)
   printf("%d",A[i],i);
}

printf("\n\n");
flag=0;

printf("Polynomial 2 = ");
```

```
for(i=5;i>=0;i-)
{
 if(B[i]!=0 && i>0)
 {
   printf("%d.(x)%d ",B[i],i);
   flag=1;
 }

 if(i>0 && B[i-1]>0 && flag==1)
    printf("+ ");

 if(B[i]!=0 && i==0)
  printf("%d",B[i],i);
}

for(i=0;i<6;i++)
 C[i]=A[i]+B[i];//Polynomial Addition

flag=0;
printf("\n\nPolynomial 1 + Polynomial 2 = ");
for(i=5;i>=0;i-)
{
 if(C[i]!=0 && i>0)
 {
   printf("%d.(x)%d ",C[i],i);
   flag=1;
 }

 if(i>0 && C[i-1]>0 && flag==1)
    printf("+ ");

 if(C[i]!=0 && i==0)
  printf("%d",C[i],i);
}

getch();
}
```

Output:

```
Polynomial 1 = -7.(x)5 + 4.(x)2 + 1.(x)1

Polynomial 2 = 3.(x)4 + 5.(x)3 + 6.(x)2 + 10.(x)1 -14

Polynomial 1 + Polynomial 2 = -7.(x)5 + 3.(x)4 + 5.(x)3 + 10.(x)2 +
11.(x)1 -14
```

```
#include<stdio.h>
#include<conio.h>
void main()
{
```

```
int a[10],b[10],c[10],i,m,n,cnt=0;
clrscr();
for(i=0;i<=9;i++)
a[i]=0;

for(i=0;i<=9;i++)
b[i]=0;
printf("\nEnter the order of first Polynomial");
scanf("%d",&m);
printf("\nEnter the Coefficient");
for(i=m;i>=0;i–)
{
scanf("%d",&a[i]);
}
printf("\nEnter the order of Second Polynomial");
scanf("%d",&n);
printf("\nEnter the Coefficient");
for(i=n;i>=0;i–)
{
scanf("%d",&b[i]);
}
if(m>=n)
{
for(i=m;i>=0;i–)
{
c[i]=a[i]+b[i];
cnt++;
}
}
else
{
for(i=n;i>=0;i–)
{
c[i]=a[i]+b[i];
}
}
printf("\n\nRESULTANT POLYNOMIAL IS :A:=");
for(i=cnt-1;i>0;i–)
{
if(c[i]!=0)
printf("%dX^%d+",c[i],i);
}
printf("%d",c[i]);
getch();
}
```

Parallel Arrays

Notes to be prepared by students