Module 2 Arrays and Searching

Polynomial representation using Arrays, Sparse matrix, Stacks, Queues-Circular Queues, Priority Queues, Double Ended Queues, Evaluation of Expressions Linear Search and Binary Search

Arrays

Array: a set of index and value

Data structure: For each index, there is a value associated with that index.

Representation (possible): implemented by using consecutive memory.

ADT Array is

objects: A set of pairs <index, value> where for each value of index there is a value from the set item. Index is a finite ordered set of one or more dimensions, for example, {0, ..., n-1} for one dimension, {(0,0), (0,1), (0,2),0,0), (1,1), (1,2), (2,0), (2,1), (2,2)} for two dimensions etc.

Functions:

for all A € Array, i e index, x € item,j, size € integer

Array Create(j, list) := return an array of J dimensions where list is a j-tuple whose ith element is the size of the ith dimension. Items are undefined.

Item Retrieve(A, i) = if (I in index) return the item associated with index value i in array A else return error

Array Store{A, i, x) = if (i in index) return an array that is identical to array A except the new pair <i, x> has been inserted else return error

End array

Arrays in C

```
int list[5], *plist[5];

list[5] five integers
        list[0], list[l], list[2], list[3], list[4]

*plist[5]: five pointers to integers
        plist[0], plist[l], plist[2], plist[3], plist[4]
```

implementation of 1-D array

list[0]	base address = a
list[l]	a + sizeof(int)
list[2]	a + 2*sizeof(int)
list [3]	a + 3*sizeof(int)
list [4]	a + 4*sizeof(int)

Compare int * list 1 and int list2[5] ui C.

Same: listl and list! are pointers.

Difference: list2 reserves five locations.

Notations:

list2 - a pointer to list2[0]

(list2 + i) - a pointer to list2[i]   (&list2[i])

*(list2 + i) - list2[i]

Example: 1-dimension array addressing

int one[] = {0, 1,2,3,4};

Goal print out address and value

```
void printf(int *ptr, int rows)
{
        /* print out a one-dimensional aiTay using a pointer */
        int i;
        printf("Address Contents n");
        for (i=0; i < rows; i++)
                printf("%8u%5dW, pti+i, *(ptr+i));
        printf("\n");
}
```

Call: printl(&one[0], 5)

Address	Contents
12244868	0
12244872	1
12244876	2
12244880	3
12244884	4

Ordered List Examples

Ordered (linear) list: (iteml. item2, item3   item n)

Days of the week :(MONDAY, TUEDSAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY)

Values in a deck of cards: (2, 3, 4, 5, 6, 7, 8, 9,10, Jack, Queen, King, Ace)

Operations on Ordered List

Find the length. 11, of the list.

Read the items from left to light (or light to left).

Retrieve the ith element

Store a new value into the ith position.

Insert a new element at the position i, causing elements numbered i. l+l… n to become numbered 1+1. i+2, ..., n+1

Delete the element at position i, causing elements numbered i+1… n to become numbered i, i+l, ...n-1

Polynomial representation using Arrays

In mahematical perspective, a polynomial is a sum of terms, where each term has a form of ax^e , where x is a variable, a is a coefficient and e is the exponent.

Eg: $A(x)= 30x^{20}+ 2x^5+4$, $B(x)= x^4+ 3x^2+1$,

The largest exponent of a polynomial is called its degree. Oefficients that are zero are not dispayed. The standard mathematical defgenitions for the sum and product of polynomials.

Assume $A(x)= \sum a_i x^i$ and $B(x)= \sum b_i x^i$ then

$A(x) + B(x) = \sum(a_i+b_i)\, x^i$

$A(x) \,.\, B(x) = \sum(a_i\, x^i \,.\, \sum(b_j x^j))$

One way to represent polynomial in C is to create a structure type polynomial.as below:

```
#define max-degree 100

struct {
        int degree;
        int coeff[max-degree];
        } polynomial;
```

If a is a type of polynomial and n< max-degree the polynomial $A(x) = \sum_{1=0}^{n} a_i x^i$

a.degree = n

a.coeff[i] = a_{n-i} $0 \leq i \leq n$

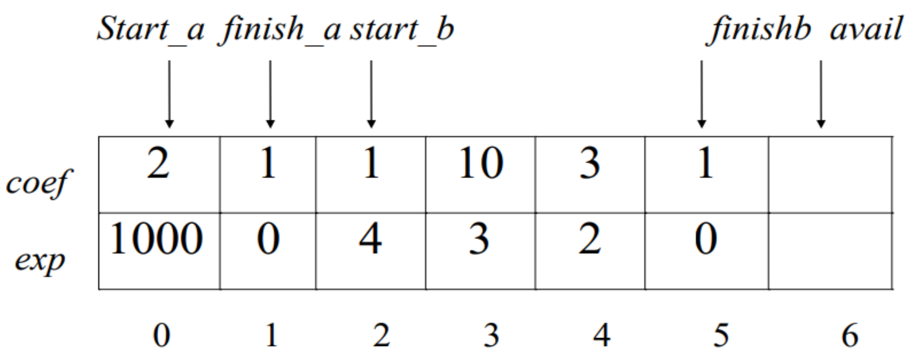
in this representation, we stor the coefficients in the order of decreasing exponents, such that a.coeff(i) is the coefficient of x^{n-i} , if n-I exists; otherwise a.coeff(i) = 0. This representation leads to very simple algorithms but it waste a lot of space.

To preserve space we devise an alter representation that uses only one global array, terms, to store all our polynomials. The C declarations needed are:

```
MAX-TERMS 100/*size of terms array*/
typedef struct{
        float coef;
        int expon;
}polynomial;
polynomial terms [MAXTERMS];
int avail = 0
```

Consider the two polynomials

$A(X)=2X^{1000}+1$
$B(X)=X^4+10X^3+3X^2+1$



	0	1	2	3	4	5	6
coef	2	1	1	10	3	1	
exp	1000	0	4	3	2	0	

The total number of nonzero terms must be no more than MAX-TERMS. Our specification used poly to refer to a polynomial, and our representation translated poly into a <start, finish > pair. Therefore, to use A (x) we must pass.in startA and finishA. Any polynomial A that has n nonzero terms has startA and finishA such that finishA = startA +n- 1.

It certainly solves the problem of many zero terms since A (x) = 2+ I uses only six units of storage: one for startA, one for finishA two for the coefticients, and two for the exponents.

Polynomial Addition

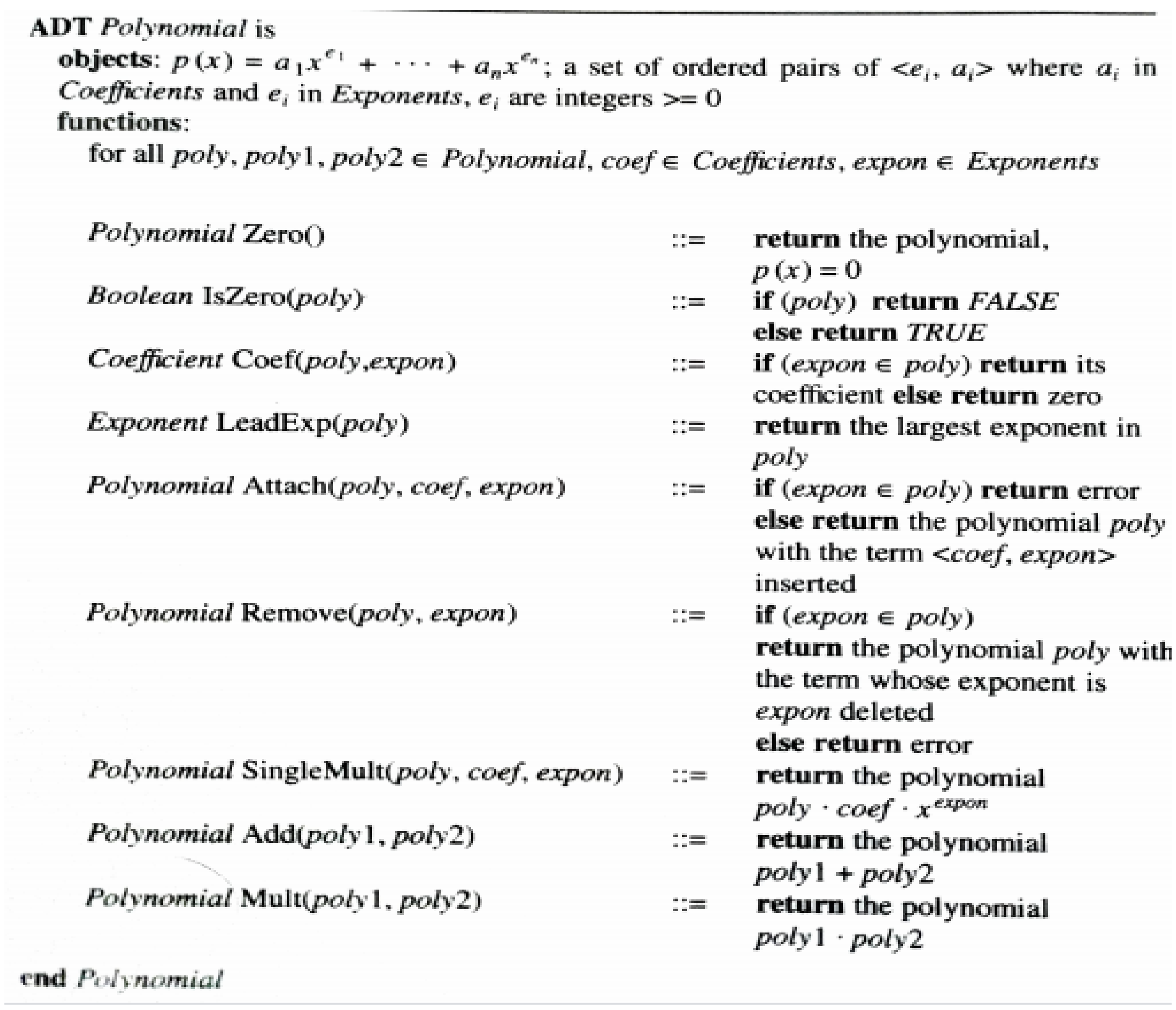
We would now like to write a C function that adds two polynomials, A and B. represented as above to obtain D =A + B. To produce D(x), padd adds A (x) and B (x) term by term. Starting at position avail, atach places the terms of D into the array, terms. If there is not enough space in terms to accommodate D, an error message is printed to the standard error device and we exit the program with an error condition.

Function for ADD two polynomials

```
void padd (int startA, int finishA, int startB, int finishB, int *st.art.D, int *f inishD)
{*/add A(x) and B (x) to obt ain D(x) */
        float coefficient;
        *startD = avail;
        while (startA <= finishA && startB <= finishB)
        switch (COMPARE ( terms [ startA] .expon, terms [startB].expon)){
        case 1: /* a expon <b expon * 7
                attach (terms [startB] .coef,terms lstartB] .expon) ;
                startB++;
                break;
        case 0:/* equal exponents *
                coefficient = terms [startA] .coef +
                terms [startB].coef
                if (coefficient)
                        attach (coefficient, terms [startA] .expon);
                startA++;
                startB++;
                break;
         case 1: /* a expon > b expon */
                attach (terms [startA] .coef, terms [startA].expon);
                startA++;
        }
        /* add in remaining terms of A (x) */
        for (; startA <= finishA; startA++)
                attach (terms [startA] .coef, terms [startA] .expon);
        /* add in remaining terms of B(x) */
        for (; startB <= finishB; startB++)
                attach (terms [startB] .coef, terms [startA] .expon);
        *finishD = avail-1;
}
```

Function for add new terms

```
void attach (float coefficientt, int exponent)
{/*7 add a new term to the polynomial */
        if (avail >= MAX-TERMS) {
                fprintf (stderr, "Too many terms in the polynomial \n");
                exit (EXIT_FAILURE);
        }
        terms [avail].coef = coefficient;
        terms [avail++] .expon = exponent;
        }
```



ADT *Polynomial* is

objects: $p(x) = a_1x^{e_1} + \cdots + a_nx^{e_n}$; a set of ordered pairs of $<e_i, a_i>$ where a_i in *Coefficients* and e_i in *Exponents*, e_i are integers >= 0

functions:

for all *poly*, *poly*1, *poly*2 ∈ *Polynomial*, *coef* ∈ *Coefficients*, *expon* ∈ *Exponents*

Polynomial Zero()	::=	**return** the polynomial, $p(x) = 0$
Boolean IsZero(*poly*)	::=	**if** (*poly*) **return** *FALSE* **else return** *TRUE*
Coefficient Coef(*poly*,*expon*)	::=	**if** (*expon* ∈ *poly*) **return** its coefficient **else return** zero
Exponent LeadExp(*poly*)	::=	**return** the largest exponent in *poly*
Polynomial Attach(*poly*, *coef*, *expon*)	::=	**if** (*expon* ∈ *poly*) **return** error **else return** the polynomial *poly* with the term <*coef*, *expon*> inserted
Polynomial Remove(*poly*, *expon*)	::=	**if** (*expon* ∈ *poly*) **return** the polynomial *poly* with the term whose exponent is *expon* deleted **else return** error
Polynomial SingleMult(*poly*, *coef*, *expon*)	::=	**return** the polynomial $poly \cdot coef \cdot x^{expon}$
Polynomial Add(*poly*1, *poly*2)	::=	**return** the polynomial *poly*1 + *poly*2
Polynomial Mult(*poly*1, *poly*2)	::=	**return** the polynomial *poly*1 · *poly*2

end *Polynomial*

SPARSE MATRIX

What is Sparse Matrix?

In computer programming, a matrix can be defined with a 2-dimensional array. Any array with 'm' columns and 'n' rows represent a m X n matrix. There may be a situation in which a matrix contains more number of ZERO values than NON-ZERO values. Such matrix is known as sparse matrix. When a sparse matrix is represented as a two- dimensional array, we waste space. For example, consider the space requirements necessary to store a 1000 x 1000 matrix that has only 2000 non-zero elements. The corresponding two-dimensional array requires space for 1,000,000 elements! We can do much better by using a representation in which only the nonzero elements are stored.

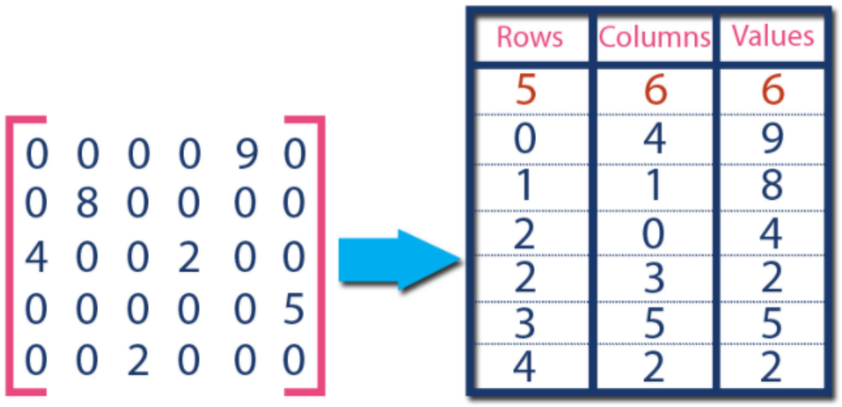
Sparse Matrix Representations

A sparse matrix can be represented by using TWO representations, those are as follows...

1. Triplet Representation (Array Representation)
2. Linked Representation

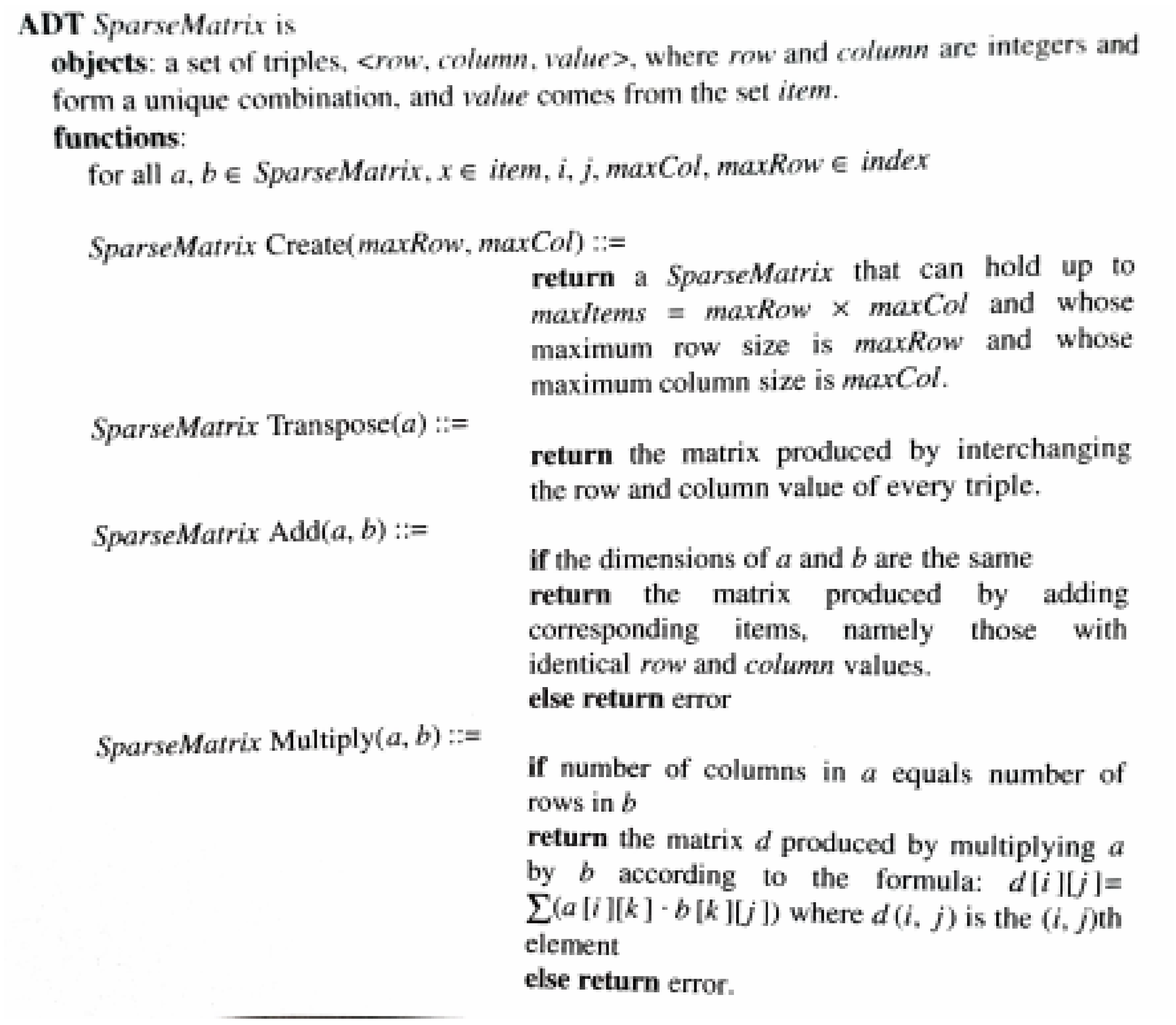
Triplet Representation (Array Representation)

In this representation, we consider only non-zero values along with their row and column index values. In this representation, the 0th row stores the total number of rows, total number of columns and the total number of non-zero values in the sparse matrix.
For example, consider a matrix of size 5 X 6 containing 6 number of non-zero values.



$$\begin{bmatrix} 0 & 0 & 0 & 0 & 9 & 0 \\ 0 & 8 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 2 & 0 & 0 & 0 \end{bmatrix} \rightarrow$$

Rows	Columns	Values
5	6	6
0	4	9
1	1	8
2	0	4
2	3	2
3	5	5
4	2	2



ADT *SparseMatrix* is

objects: a set of triples, <*row*, *column*, *value*>, where *row* and *column* are integers and form a unique combination, and *value* comes from the set *item*.

functions:

for all *a*, *b* ∈ *SparseMatrix*, *x* ∈ *item*, *i*, *j*, *maxCol*, *maxRow* ∈ *index*

SparseMatrix Create(*maxRow*, *maxCol*) ::= **return** a *SparseMatrix* that can hold up to *maxItems* = *maxRow* × *maxCol* and whose maximum row size is *maxRow* and whose maximum column size is *maxCol*.

SparseMatrix Transpose(*a*) ::= **return** the matrix produced by interchanging the row and column value of every triple.

SparseMatrix Add(*a*, *b*) ::= **if** the dimensions of *a* and *b* are the same **return** the matrix produced by adding corresponding items, namely those with identical *row* and *column* values. **else return** error

SparseMatrix Multiply(*a*, *b*) ::= **if** number of columns in *a* equals number of rows in *b* **return** the matrix *d* produced by multiplying *a* by *b* according to the formula: $d[i][j] = \sum(a[i][k] \cdot b[k][j])$ where $d(i, j)$ is the (i, j)th element **else return** error.

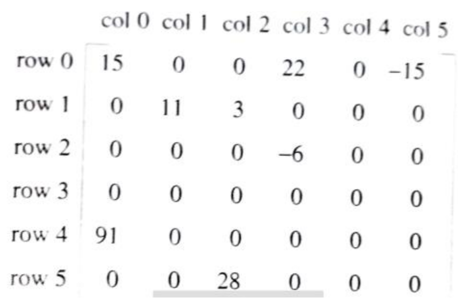
we can characterize uniquely any clement within a matrix by using the triple. Since we want our transpose operation to work efficiently, we should organize the triples so that the row indices are in ascending order. We can go one step further by also requiring that all the triples for any row be stored so that the column indices are in ascending order.

Create operation

SparseMatrix Create(maxRow, maxCol)

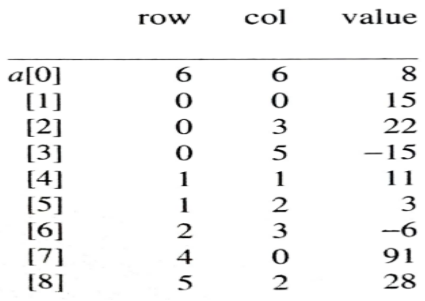
```
#define MAX-TERMS 101/* maximum number of terms +1*/
typedef struct {
int col;
int row;
int value;
}term;
term a [MAX-TERMS] ;
```

Sparse Matrix



	col 0	col 1	col 2	col 3	col 4	col 5
row 0	15	0	0	22	0	−15
row 1	0	11	3	0	0	0
row 2	0	0	0	−6	0	0
row 3	0	0	0	0	0	0
row 4	91	0	0	0	0	0
row 5	0	0	28	0	0	0

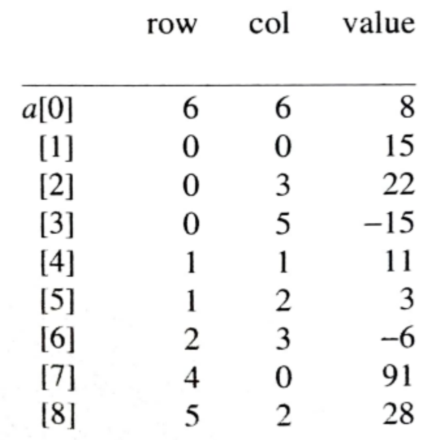
Triplet Representation



	row	col	value
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	−15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	−6
[7]	4	0	91
[8]	5	2	28

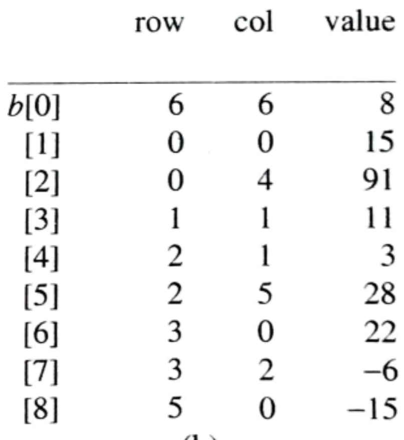
Thus, a [0]. row contains the number of rows; a [0]. col contains the number of columns; and a [0].value contains the total number of nonzero entries. Positions I through 8 store the triples representing the nonzero entries. The row index is in the field row; the column index is in the field col; and the value is in the field value. The triples are ordered by row and within rows by columns.

Transposing a Matrix



	row	col	value
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	−15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	−6
[7]	4	0	91
[8]	5	2	28

(a)



	row	col	value
b[0]	6	6	8
[1]	0	0	15
[2]	0	4	91
[3]	1	1	11
[4]	2	1	3
[5]	2	5	28
[6]	3	0	22
[7]	3	2	−6
[8]	5	0	−15

(b)

To transpose a matrix, we must interchange the rows and columns. This means that each element a[i][j] in the original matrix becomes element b[i][j] in the transpose matrix. Since we have organized the original matrix by rows, we might think that the following is a good algorithm for transposing a matrix.

```
for each row i
        take element <i, j, value> and store it
        as element <j, i, value> of the transpose;
```

Here (0,0, 15), which becomes (0,0. 15)
(0, 3, 22), which becomes (3, 0, 22),
(0, 5,-15). which becomes (0, 5,-15). Etc..

If we place these triples consecutively in the transpose matrix, then, as we insert new triples, we must move elements to maintain the correct order. We can avoid this data movement by using the column indices to determine the placement of elements in the transpose matrix. This suggests the following algorithm:

```
for all elements in column j
        place element <i, j value> in
        element <j, i, value>
```

The algorithm indicates that we should "find all the elements in column 0 and store them in row 0 of the transpose matrix, find all the elements in column 1 and store them in row 1, etc." Since the original matrix ordered the rows, the columns within each row of the transpose matrix will be arranged in ascending order as well. The variable currentb, holds the position in b that will contain the next transposed term. we generate the terms in b by rows, but since the rows in b correspond to the columns in a, we collect the nonzero terms for row i of b by collecting the nonzero terms from column i of a.

```
void transpose(term a[], term b[]){/*bis set to the transpose of a */
        int n, i, j, currentb;
        / total number of elements */
        n =a [0].value;
        b[0].row = a [0].col; /* rows in b = columns in a */
        b[0].col = a[0].row; /* columns in b = rows in a */
        b[0].value = n;
        if (n >0) { /* non zero matrix */
                Currentb = 1;
                for (i = = 0; i < a[01.col; it+)
                * transpose by the columns in a */
                        for (j = 1; j <= n; j++) *
                        /* Eind elements from the current col umn
                                if (a[ji.col == i){
                                * element is in current column, add it to b */
                                b[currentb] .row = a[jl.col;
                                blcurrentb] . col = a[j].row;
                                b[currentb].value = a [j].value;
                                currentb+t
                                }
        }
}
```

Analysis of transpose: Determining the computing time of this algorithm is easy since the nested for loops are the decisive factor. The remaining statements (two if statements and several assignment statements) require only constant time. We can see that the outer for loop is iterated a |0].col times, where a [0].col holds the number of columns in the original matrix. In addition, one iteration of the inner for loop requires a [0J.value time. where a |0}].value holds the number of elements in the original matrix. Therefore. the total time for the nested for loops is columns elements. Hence, the asymptotic time complexity is O(columns elements). If we represented our matrices as two-dimensional arrays of size rows x columns, we could obtain the transpose in O(rows columns) time.

Sample Program

```
#include <stdio.h>
#define MAX 20

void read_matrix(int a[10][10], int row, int column);
void print_sparse(int b[MAX][3]);
void create_sparse(int a[10][10], int row, int column, int b[MAX][3]);

int main()
{
   int a[10][10], b[MAX][3], row, column;
   printf("\nEnter the size of matrix (rows, columns): ");
   scanf("%d%d", &row, &column);
   read_matrix(a, row, column);
   create_sparse(a, row, column, b);
   print_sparse(b);
   return 0;
}

void read_matrix(int a[10][10], int row, int column)
{
   int i, j;
   printf("\nEnter elements of matrix\n");
   for (i = 0; i < row; i++)
           {
              for (j = 0; j < column; j++)
              {
                printf("[%d][%d]: ", i, j);
                scanf("%d", &a[i][j]);
            }
        }
}

void create_sparse(int a[10][10], int row, int column, int b[MAX][3])
{
    int i, j, k;
    k = 1;
    b[0][0] = row;
    b[0][1] = column;
    for (i = 0; i < row; i++)
    {
        for (j = 0; j < column; j++)
        {
            if (a[i][j] != 0)
            {
                b[k][0] = i;
                b[k][1] = j;
                b[k][2] = a[i][j];
                k++;
            }
        }
        b[0][2] = k - 1;
    }
}

void print_sparse(int b[MAX][3])
{
    int i, column;
    column = b[0][2];
    printf("\nSparse form - list of 3 triples\n\n");
    for (i = 0; i <= column; i++)
    {
        printf("%d\t%d\t%d\n", b[i][0], b[i][1], b[i][2]);
    }
}
```