||JAI SRI GURUDEV||

Sri AdichunchanagiriShikshana Trust ®

SJB INSTITUTE OF TECHNOLOGY

An Autonomous Institution under Visvesvaraya Technological University, Belagavi

Approved by AICTE, New Delhi, Accredited by NAAC, New Delhi with 'A+' Grade, Accredited by National Board of Accreditation, New Delhi
Recognized by UGC, New Delhi with 2(f) and 12(B), Certified by ISO 9001-2015
No. 67, BGS Health & Education City, Dr. Vishnuvardhan Road, Kengeri, Bengaluru - 560 060

NOTES

Module-1

Subject Name: **DSA**
Subject Code: **23CDT302**

Prepared By:
Dr. Rekha B
Professor

Department of CSE (Data Science)

Module-1

Introduction: data Structures, Classifications (Primitive & Non-Primitive), Data structure operations (Traversing, inserting, deleting, searching, and sorting). Review of Arrays. Structures: Array of structures Self-Referential Structures. Dynamic Memory Allocation Functions. Demonstration of representation of Polynomials and Sparse Matrices with array.

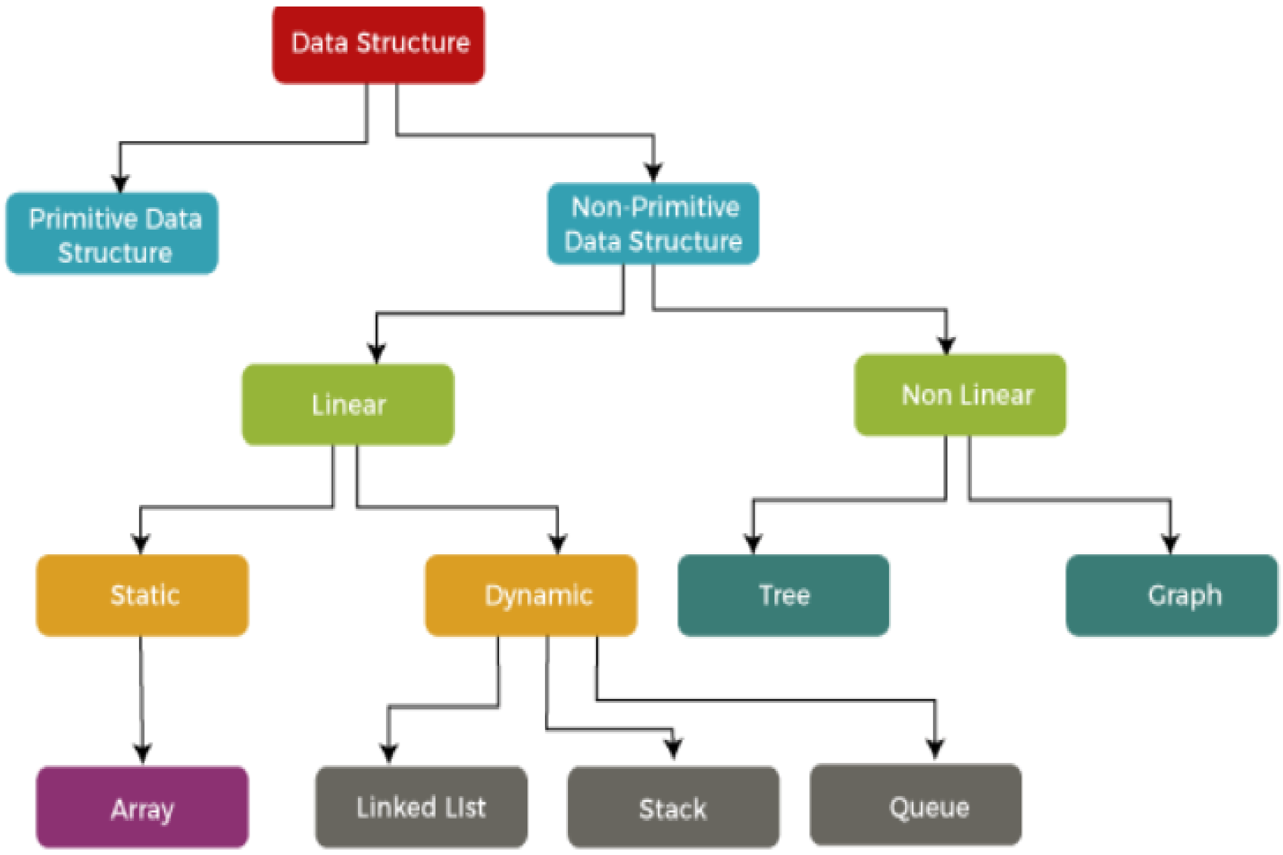
Textbook1**:Chapter1: 1.2, Chapter2: 2.3-2.5,** Textbook2**: Chapter1: 1.1 - 1.4**

Chapter 1

Definition of Data Structure

A data structure is basically a group of data elements that are put together under one name, and which defines a particular way of storing and organizing data in a computer so that it can be used efficiently.

Classification



Primitive and Non-primitive Data Structures

Primitive data structures are the fundamental data types which are supported by a programming language. Some basic data types are integer, real, character, and boolean. The terms 'data type', 'basic data type', and 'primitive data type' are often used interchangeably. Non-primitive data structures are those data structures which are created using primitive data structures. Examples of such data structures include linked lists, stacks, trees, and graphs. Non-primitive data structures can further be classified into two categories: linear and non-linear data structures.

Linear and Non-linear Structures

If the elements of a data structure are stored in a linear or sequential order, then it is a linear data structure.

Examples include arrays, linked lists, stacks, and queues.

Linear data structures can be represented in memory in two different ways.

One way is to have to a linear relationship between elements by means of sequential memory locations.

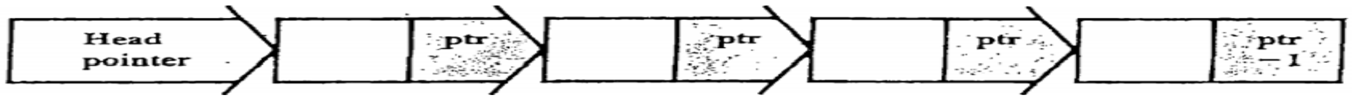
The other way is to have a linear relationship between elements by means of links.

However, if the elements of a data structure are not stored in a sequential order, then it is a non-linear data structure. The relationship of adjacency is not maintained between elements of a non-linear data structure. Examples include trees and graphs.

Array – an array is a collection of similar data elements in a contiguous block of memory. It allows for efficient access to elements using indices.

Example- int marks[5];

Linked list – is a very flexible, dynamic data structure in which elements (called nodes) from a sequential list. Every node in the list points to the next node in the list. The last node in the list contains a NULL pointer to indicate that is the end or tail of the list.



Stacks – is a linear data structure in which insertion and deletion of elements are done at only one end, which is known as the top of the stack. Stack is called a LIFO.

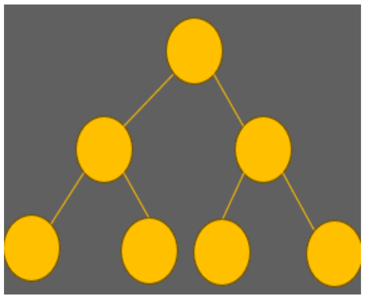
A	AB	ABC	ABCD	ABCDE
0	1	2	3	4

Queues – A queue is FIFO data structure in which the element that is inserted first is the first one to be taken out. Here the front=0 and rear=5.

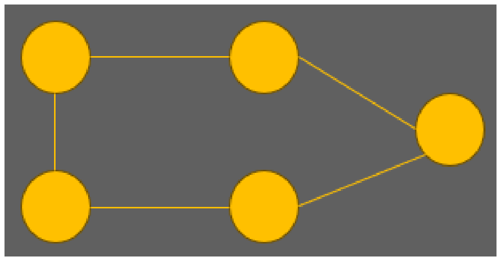
12	9	7	18	14

0	1	2	3	4

Trees – A tree is a non-linear data structure which consists of a collection of nodes arranged in a hierarchical order. The simplest form of a tree is a binary tree.



Graphs – A graph is a non-linear data structure which is collection of vertices(also called as nodes) and edges that connect these vertices.



Data Structure Operations

Traversing - It means to access each data item exactly once so that it can be processed. For example, to print the names of all the students in a class.

Searching - It is used to find the location of one or more data items that satisfy the given constraint. Such
a data item may or may not be present in the given collection of data items. For example, to find the
names of all the students who secured 100 marks in mathematics.

Inserting - It is used to add new data items to the given list of data items. For example, to add the details
of a new student who has recently joined the course.

Deleting - It means to remove (delete) a particular data item from the given collection of data items. For
example, to delete the name of a student who has left the course.

Sorting - Data items can be arranged in some order like ascending order or descending order depending
on the type of application. For example, arranging the names of students in a class in an alphabetical
order, or calculating the top three winners by arranging the participants' scores in descending order and
then extracting the top three.

Merging Lists - of two sorted data items can be combined to form a single list of sorted data items.

Review of Array

An array data structure is a fundamental concept in computer science that stores a collection of elements in a contiguous block of memory. It allows for efficient access to elements using indices and is widely used in programming for organizing and manipulating data.

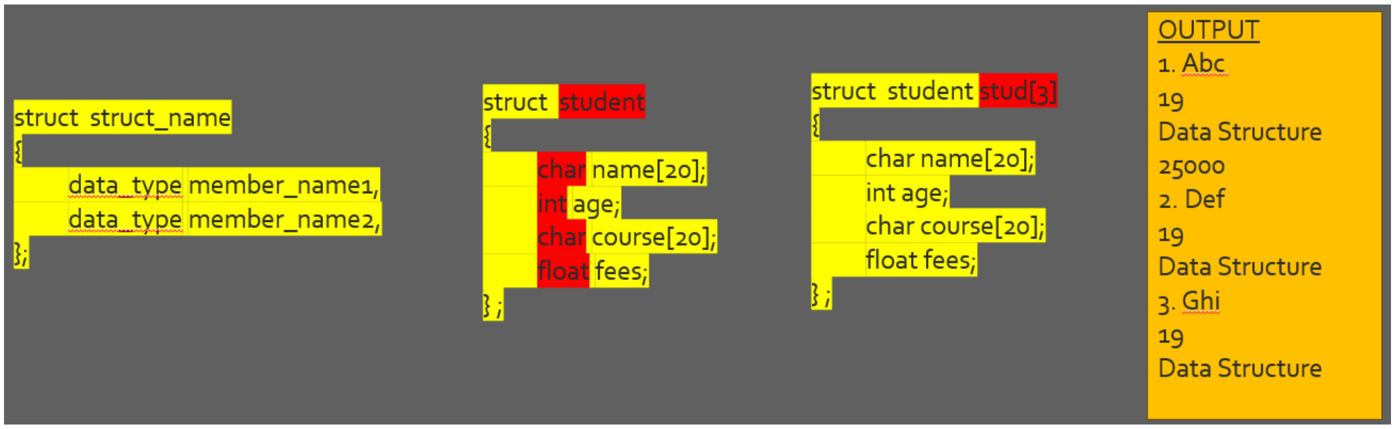
Example – int a[5]

10	**20**	**30**	**40**	**50**
0	**1**	**2**	**3**	**4**

Chapter 2

Structure - Array of Structure

Arrays are collections of data of the same type. In C there is an alternate way of grouping data that permits the data to vary in type. This mechanism is called the struct, short for structure. A structure (called a record in many other programming languages) is a collection of data items, where each item is identified as to its type and name. For example



```
struct struct_name
{
        data_type member_name1;
        data_type member_name2;
};
```

```
struct student
{
        char name[20];
        int age;
        char course[20];
        float fees;
} ;
```

```
struct student stud[3]
{
        char name[20];
        int age;
        char course[20];
        float fees;
} ;
```

```
OUTPUT
1. Abc
19
Data Structure
25000
2. Def
19
Data Structure
3. Ghi
19
Data Structure
```

```
#include<stdio.h>
#include<stdlib.h>
#define size 20

void insert(int arr[], int *n);
void delete(int arr[], int *n);
void display(int arr[], int n);

void main()
{
        int arr[size], i, n, ch;
        printf("Enter Initial size of Array\n");
        scanf("%d", &n);
        printf("Enter Array Elements\n");
        for(i=0; i<n; i++)
                scanf("%d", &arr[i]);
        for(;;)
        {
                printf("\nMenu\n");
                printf("1. Insert\n2. Delete\n3. Display\n4. Exit\n");
                scanf("%d", &ch);
                switch(ch)
                {
                        case 1:
                                insert(arr, &n);
                                break;
                        case 2:
                                delete(arr, &n);
                                break;
                        case 3:
                                display(arr, n);
                                break;
                        case 4:
                                exit(0);
                }
        }
}

void insert(int arr[], int *n)
{
        int i, pos, ele;
        printf("Enter Element\n");
        scanf("%d", &ele);
        printf("Enter Position\n");
        scanf("%d", &pos);
        for(i=(*n)-1; i>=pos-1; i--)
        {
                arr[i+1] = arr[i];
        }
        arr[pos-1] = ele;
        *n = *n + 1;
}

void delete(int arr[], int *n)
{
        int i, pos;
        printf("Enter the Position to delete the element\n");
        scanf("%d", &pos);
        for(i=pos-1; i<(*n)-1; i++)
        {
                arr[i] = arr[i+1];
        }
        *n = *n - 1;
```

```
}


void display(int arr[], int n)
{
        int i;
        for(i=0; i<n; i++)
                printf("%d ", arr[i]);
        printf("\n");
}
```

Self Referential Structure

A self-referential structure is one in which one or more of its components is a pointer to itself. Self-referential structures usually require dynamic storage management routines (malloc and free) to explicitly obtain and release memory.

Example

```
typedef struct list
{
char data;
struct list *link;
} list;
```

Each instance of the structure list will have two components, data and link, data is a single character, while link is a pointer to a list structure. The value of link is either the address in memory of an instance of list or the null pointer.

Consider these statements, which create three structures and assign values to their respective

```
list item1, item2, item3;
item1.data = 'a';
item2.data = 'b';
 item3.data = 'c';
item1.link = item2.link = item3.link = NULL;
```

Structures item 1, item 2, and z7em3 each contain the data item a, b, and c, respectively, and the null pointer. We can attach these structures together by replacing the null link field in item 2 with one that points to item 3 and by replacing the null link field in item 1 with one that points to item 2.

```
item1.link = &item2;
item2.link = &item3
```

Dynamic Memory Allocation

Dynamic Memory Allocation can be defined as a procedure in which the size of a data structure (like Array) is changed during the runtime.

10	**20**	**30**	**40**	**50**	**60**	**70**	**80**	**90**
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**

As can be seen, the length (size) of the array above is 9. But what if there is a requirement to change this length (size)? For example,
If there is a situation where only 5 elements are needed to be entered in this array. In this case, the remaining 4 indices are just wasting memory in this array. So there is a requirement to lessen the length (size) of the

array from 9 to 5.

Take another situation. In this, there is an array of 9 elements with all 9 indices filled. But there is a need to enter 3 more elements in this array. In this case, 3 indices more are required. So the length (size) of the array needs to be changed from 9 to 12.

This procedure is referred to as **Dynamic Memory Allocation in C**.

Therefore, C **Dynamic Memory Allocation** can be defined as a procedure in which the size of a data structure (like Array) is changed during the runtime.

C provides some functions to achieve these tasks. There are 4 library functions provided by C defined under <stdlib.h> header file to facilitate dynamic memory allocation in C programming. They are:

1. malloc()
2. calloc()
3. free()
4. realloc()

malloc()

The “malloc” or “memory allocation” method in C is used to dynamically allocate a single large block of memory with the specified size.

It returns a pointer of type void which can be cast into a pointer of any form. It doesn’t Initialize memory at execution time so that it has initialized each block with the default garbage value initially.

ptr = (cast-type*) malloc(n*byte-size)

For Example:

ptr = (int*) malloc(100 * sizeof(int));

Since the size of int is 4 bytes, this statement will allocate 400 bytes of memory. And, the pointer ptr holds the address of the first byte in the allocated memory.

calloc()

“calloc” or “contiguous allocation” method in C is used to dynamically allocate the specified number of blocks of memory of the specified type. it is very much similar to malloc() but has two different points and these are:

It initializes each block with a default value ‘0’.

It has two parameters or arguments as compare to malloc().

ptr = (cast-type*)calloc(n, element-size);

For Example:

ptr = (float*) calloc(5, sizeof(int)); **free()**

“free” method in C is used to dynamically de-allocate the memory. The memory allocated using functions malloc() and calloc() is not de-allocated on their own. Hence the free() method is used, whenever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it.

free(ptr);

realloc()

“realloc” or “re-allocation” method in C is used to dynamically change the memory allocation of a previously allocated memory. In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to dynamically re-allocate memory. re-allocation of memory maintains the already present value and new blocks will be initialized with the default garbage value.

ptr = realloc(ptr, newSize);

where **ptr** is reallocated with new size 'newSize'.

Polynomials

A polynomial is sum of terms where each term has a form **Ax^e**

A – coefficient, x – variable, e – exponent

The largest exponent(leading exponent) of a polynomial is called its degree

Example – **$6x^{25} + 5x^{10} +35$**

This polynomial is sum of three terms. Since 25 is the largest exponent, the degree of this polynomial is 25. The last term 35 can also be written $35x^0$

Assume that we have two polynomials

$A(x) = \sum a_i x^i$

and

$B(x) = \sum b_i x^i$

then

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

$A(x) = 3x^{20} + 2x^5 + 4$ and

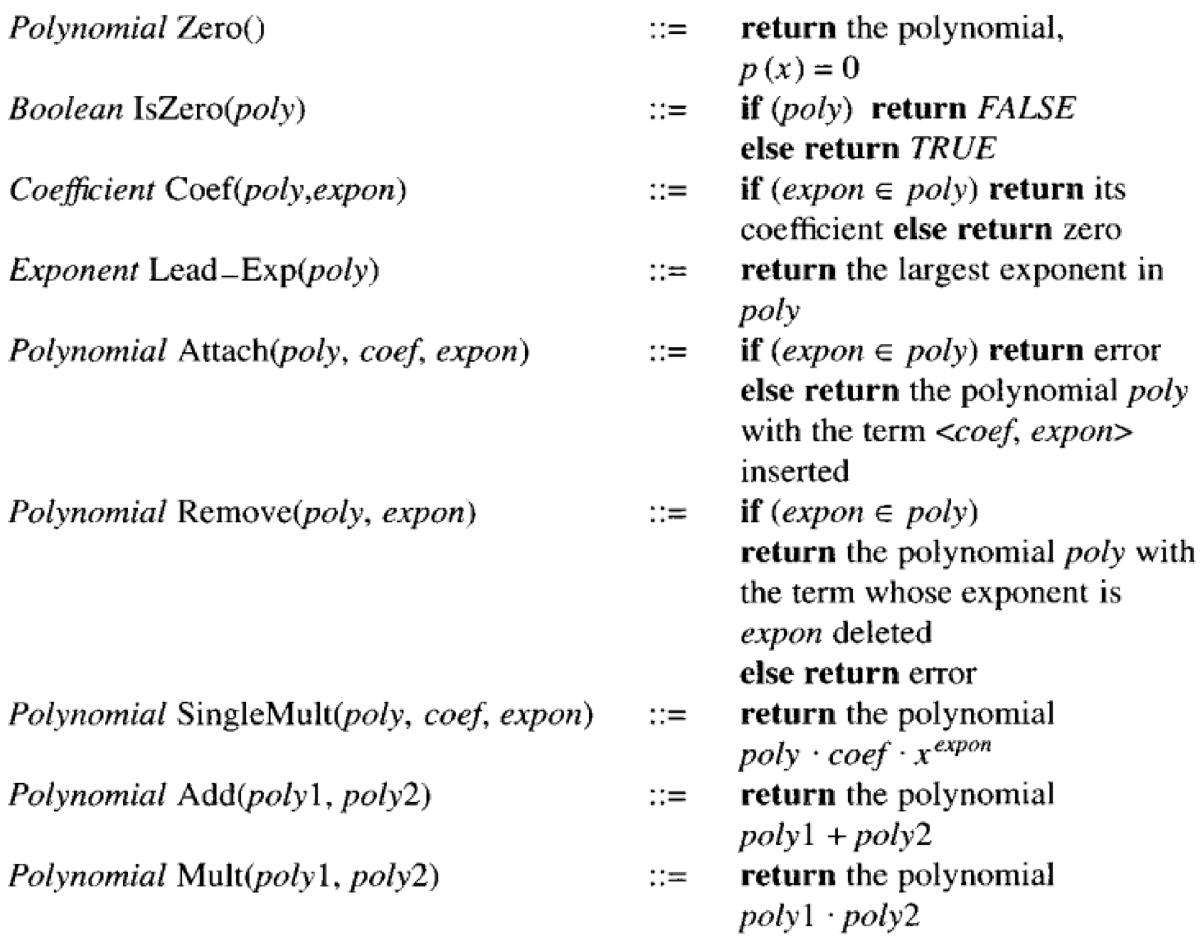
$B(x) = x^4 + 10x^3 + 3x^2 + 1$

$C(x) = 3x^{20} + 2x^5 + x^4 + 10x^3 + 3x^2 + 5$

1. **Abstract data type**
2. **Polynomial Representation**
3. **Polynomial Addition**

Abstract data type

An ADT polynomial is the one that shows various operations that can be performed on polynomials. These operations are implemented as functions subsequently in the program.



Polynomial Zero()	::=	**return** the polynomial, $p(x) = 0$
Boolean IsZero(*poly*)	::=	**if** (*poly*) **return** *FALSE* **else return** *TRUE*
Coefficient Coef(*poly*,*expon*)	::=	**if** ($expon \in poly$) **return** its coefficient **else return** zero
Exponent Lead_Exp(*poly*)	::=	**return** the largest exponent in *poly*
Polynomial Attach(*poly*, *coef*, *expon*)	::=	**if** ($expon \in poly$) **return** error **else return** the polynomial *poly* with the term <*coef*, *expon*> inserted
Polynomial Remove(*poly*, *expon*)	::=	**if** ($expon \in poly$) **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 $poly1 + poly2$
Polynomial Mult(*poly*1, *poly*2)	::=	**return** the polynomial $poly1 \cdot poly2$

Example: let a(x)=25x^6 + 10x^5 + 6x^2 + 9

1. leadexp(a) = 6

2. Coef(a, leadexp(a)) = 25

3. Iszero(a) = false

4. Attach(a,15,3) = 25x^6 + 10x^5 + 15x^3 + 6x^2 +9

5. Remove(a,6) = 10x^5 + 6x^2 + 9

Polynomial Representation

The first way to represent polynomials in C is to use typedef to create the type polynomial as below:

```
#define typedef
MAX—DEGREE 101
struct {
int degree;
float coef[MAX-DEGREE];
 } polynomial;
```

Now if a is of type polynomial and n < MAX-DEGREE, the polynomial $A(x) = {}_{i=0}\sum^{n} a_i x^i$
would be represented as: a.degree = n

$$a.coef[i] = a_{n-1},\ 0 \leq i \leq n$$

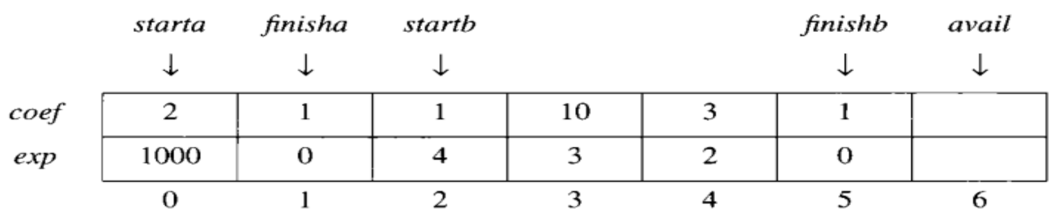
Consider the second way polynomials

```
#define typedef
MAX—TERMS 100
```

```
typedef struct {
float coef;
int expon;
 } polynomial;
polynomial terms[MAX_TERMS];
int avail = 0;
```

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

The below Figure shows how these polynomials are stored in the array terms. The index of the first term of A and B is given by starta and startb, respectively, while finisha and finishb give the index of the last term of A and B. The index of the next free location in the array is given by avail. For our example, starta = 0, finisha = 1, startb = 2, finishb = 5, and avail = 6.

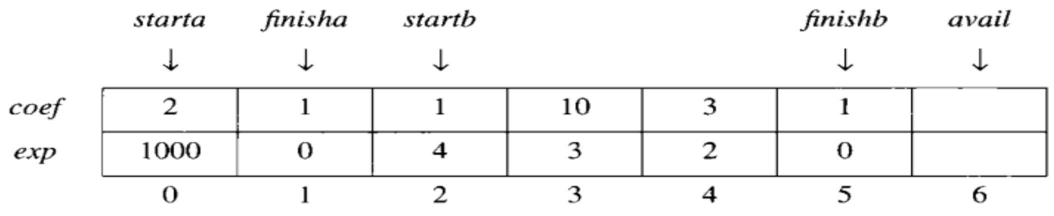


	starta ↓	*finisha* ↓	*startb* ↓			*finishb* ↓	*avail* ↓
coef	2	1	1	10	3	1	
exp	1000	0	4	3	2	0	
	0	1	2	3	4	5	6

Polynomial Addition

Function to add two polynomial

$$A(x) = 2x^{1000} + 1 \text{ and } B(x) = x^4 + 10x^3 + 3x^2 + 1$$



	starta	finisha	startb			finishb	avail
	↓	↓	↓			↓	↓
coef	2	1	1	10	3	1	
exp	1000	0	4	3	2	0	
	0	1	2	3	4	5	6

```
void padd(int starta,int finisha,int startb, int finishb,
                                int *startd,int *finishd)
{
/* add A(x) and B(x) to obtain 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 */
            attach(terms[startb].coef,terms[startb].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[startb].expon);
  *finishd = avail-1;
}
```

Function to add a new term

```
void attach(float coefficient, int exponent)
{
/* add a new term to the polynomial */
   if (avail >= MAX_TERMS) {
      fprintf(stderr,"Too many terms in the polynomial\n");
      exit(1);
   }
   terms[avail].coef = coefficient;
   terms[avail++].expon = exponent;
}
```

Each iteration of the loop requires 0(1) time.

Sparse Matrices with Arrays

1. Introduction

2. Transposing a Matrix

3. **Matrix Multiplication**

Introduction

A matrix contains m rows and n columns of elements as illustrated in below figures. In this figure, the elements are numbers. The first matrix has five rows and three columns and the second has six rows and six columns. We write m x n (read "m by n") to designate a matrix with m rows and n columns. The total number of elements in such a matrix is mn. If m equals n, the matrix is square.

What is Sparse Matrix?

A matrix which contains many zero entries or very few non-zero entries is called as Sparse matrix. In the figure B contains only 8 of 36 elements are nonzero and that is sparse.

$$\begin{array}{c|ccc|}
 & \text{col0} & \text{col1} & \text{col2} \\
\text{row 0} & -27 & 3 & 4 \\
\text{row 1} & 6 & 82 & -2 \\
\text{row 2} & 109 & -64 & 11 \\
\text{row 3} & 12 & 8 & 9 \\
\text{row 4} & 48 & 27 & 47
\end{array}$$

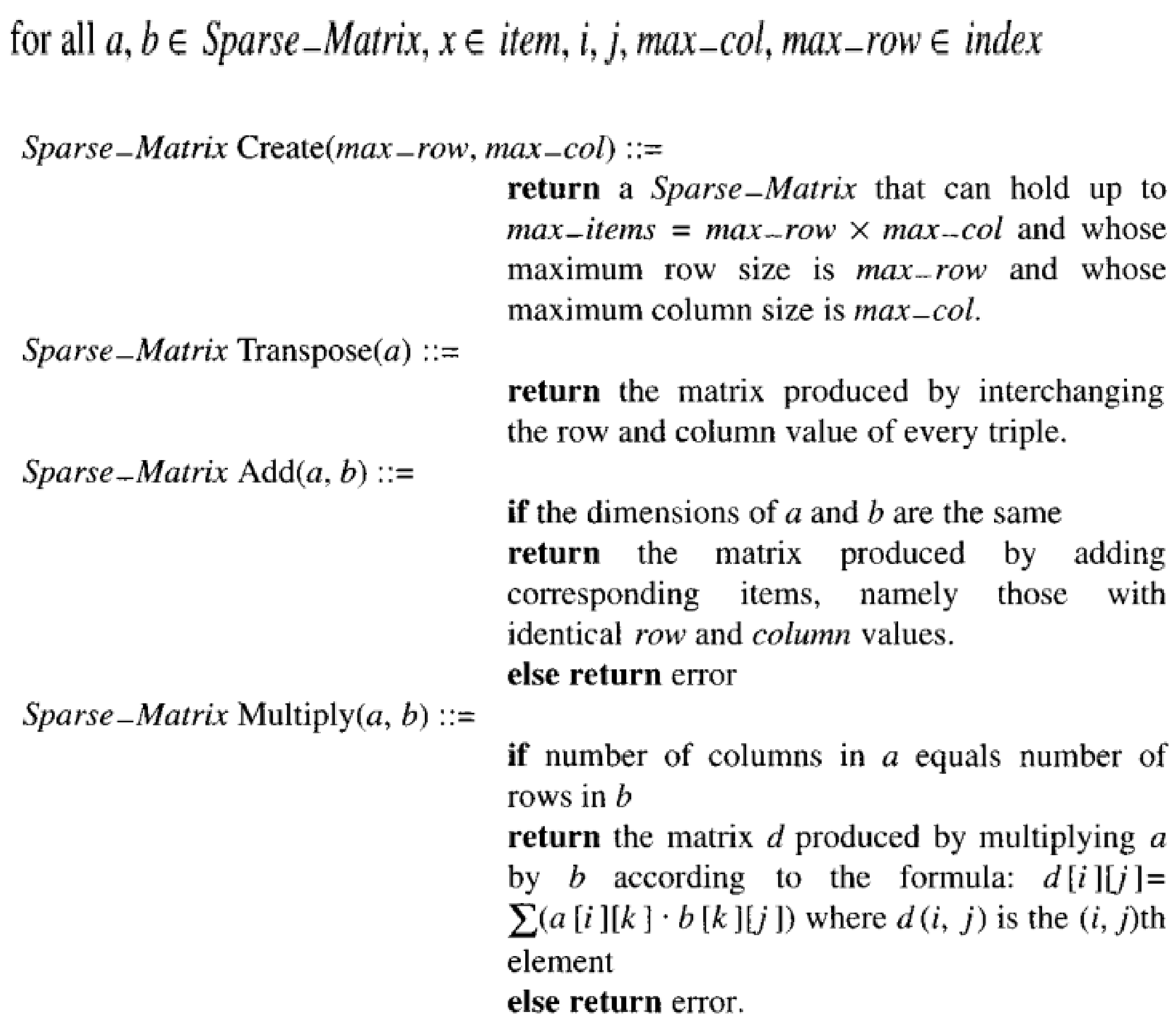
Figure A

$$\begin{array}{c|cccccc|}
 & \text{col0} & \text{col1} & \text{col2} & \text{col3} & \text{col4} & \text{col 5} \\
\text{row0} & 15 & 0 & 0 & 22 & 0 & -15 \\
\text{row1} & 0 & 11 & 3 & 0 & 0 & 0 \\
\text{row2} & 0 & 0 & 0 & -6 & 0 & 0 \\
\text{row3} & 0 & 0 & 0 & 0 & 0 & 0 \\
\text{row4} & 91 & 0 & 0 & 0 & 0 & 0 \\
\text{row5} & 0 & 0 & 28 & 0 & 0 & 0
\end{array}$$

Figure B

Important Note: A sparse matrix can be represented in 1-Dimension, 2-Dimension and 3- Dimensional array. When a sparse matrix is represented as a two-dimensional array as shown in Figure B, more space is wasted.

Structure Sparse-Matrix 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 $a, b \in$ *Sparse_Matrix*, $x \in$ *item*, i, j, *max_col*, *max_row* $\in$ *index*

Sparse_Matrix Create(*max_row*, *max_col*) ::=
return a *Sparse_Matrix* that can hold up to *max_items* = *max_row* × *max_col* and whose maximum row size is *max_row* and whose maximum column size is *max_col*.

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

Sparse_Matrix 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

Sparse_Matrix 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.

$$\begin{array}{c|ccc|}
 & \text{col0} & \text{col1} & \text{col2} \\
\text{row 0} & -27 & 3 & 4 \\
\text{row 1} & 6 & 82 & -2 \\
\text{row 2} & 109 & -64 & 11 \\
\text{row 3} & 12 & 8 & 9 \\
\text{row 4} & 48 & 27 & 47
\end{array}$$

Figure A

$$\begin{array}{c|cccccc|}
 & \text{col0} & \text{col1} & \text{col2} & \text{col3} & \text{col4} & \text{col 5} \\
\text{row0} & 15 & 0 & 0 & 22 & 0 & -15 \\
\text{row1} & 0 & 11 & 3 & 0 & 0 & 0 \\
\text{row2} & 0 & 0 & 0 & -6 & 0 & 0 \\
\text{row3} & 0 & 0 & 0 & 0 & 0 & 0 \\
\text{row4} & 91 & 0 & 0 & 0 & 0 & 0 \\
\text{row5} & 0 & 0 & 28 & 0 & 0 & 0
\end{array}$$

Figure B

Sparse–Matrix Create(*max–row*, *max–col*) ::=

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

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

Fig (a): Sparse matrix stored as triple

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

Fig (b): Transpose matrix stored as triple

The figure shows the representation of matrix in the array “a” 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 1 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

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[j][i] in the transpose matrix.

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

If we process the original matrix by the row indices it is difficult to know exactly where to place element in the transpose matrix until we processed all the elements that precede it.

This can be avoided by using the column indices to determine the placement of elements in the transpose matrix. This suggests the following algorithm:

(0,0, 15), which becomes (0,0, 15)

(0, 3, 22), which becomes (3, 0, 22)

(0,5,-15), which becomes (5, 0,-15)

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

The columns within each row of the transpose matrix will be arranged in ascending order.

```
void transpose (term a[], term b[])
{                   /* b is set to the transpose of a */
          int n, i, j, currentb;
          n = a[0].value;                   /* total number of elements */
          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)
                    { currentb = 1;
                              for (i = 0; i < a[O].col; i++)
                                        for (j= 1; j<=n; j++)
                                                  if (a[j].col == i)
                                                  {
                                                            b[currentb].row = a[j].col;
                                                            b[currentb].col = a[j].row;
                                                            b[currentb].value  = a[j].value;
                                                            currentb++;
                                                  }
                    }
}
```

Matrix Multiplication

Given A and B where A is m x n and B is n x p, the product matrix D has dimension m x p. Its element is :

$$d_{ij} = \sum_{k=0}^{n-1} a_{ik} b_{kj}$$

for 0 < i < m and 0 < j < p.

The product of two sparse matrices may no longer be sparse, as Figure shows

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$