ADT & Data Structure

What is ADT?

An ADT (Abstract Data Type) is basically a logical description or a specification of components of the data & the operations that are allowed, that is independent of the implementation.

ADT's are theoritical concept of computer Science, used in design & Analysis of Algs, data Structures, and softwar Systems, & do not correspond to specific features of computer languages.

ex :- Integer are an ADT
defined as the values 0, 1, -1, 2, 2 -- & by the operation of addition, Subtraction, multiplication & division, together with greater than / less than etc.

Why ADT?

To manage the complexity of problems & the problem solving process, abstractions are used to allow the user to focus on the "big Picture" without getting lost in the details. By creating models of the problem domain, the user can efficiently focus on the problem solving process.

Advantages:-

Encapsulation

Localization of change: code that uses an ADT ~~is changed, since an~~ will not need to be ~~cha~~ edited if the implementation of the ADT is changed.

Flexibility

Easy ~~too~~ to understand

reusable code.

Relationship between ADT & data structure:

ADT is a logical description

Data structure is real (physical) or concrete thing.

Difference between Structured Programming and Object Oriented Programming:

- Structured Programming is designed which focuses on **process**/ logical structure and then data required for that process
- Structured programming follows **top-down approach**.
- Object Oriented Programming is designed which focuses on **data**.
- Object oriented programming follows **bottom-up approach**.

What is the difference between top down and bottom up approaches

- Top Down approach is taking a **"COMPLEX SYSTEM"** and breaking it down to the **"SIMPLE INDIVIDUAL COMPONENTS"** to gain a better understanding of the inner layers. Simply putting, it's like decomposition, ie; breaking into smaller parts.
- A Bottom Up approach is building up a **"COMPLEX SYSTEM"** by piecing together **"SIMPLE INDIVIDUAL COMPONENTS"**. You work your way up to the final outcome by inter-relating individual blocks.

An Introduction to C++ Class:

When we define a class, we define a blueprint for a data type. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.
Class is an extension to Structure in c
For example:

```
struct random

{

int num;

char ch;

float marks;

}obj;
```

We'll observe that a structure is just a collection of various types of data in C. In the above example, all these variables can be accessed through the structure object *obj*.
Just the way structure encapsulates only data, classes can encapsulate data as well as functions.
A class definition starts with the keyword **class** followed by the class name; and the class body, enclosed by a pair of curly braces. A class definition must be followed either by a semicolon or a list of declarations. For example, we defined the Box data type using the keyword **class** as follows:

```
class Box
```

```
{
 public:
   double length;   // Length of a box
   double breadth;  // Breadth of a box
   double height;   // Height of a box
};
```

All C++ programs are composed of the following two fundamental elements:

- **Program statements (code):** This is the part of a program that performs actions and they are called functions.
- **Program data:** The data is the information of the program which affected by the program functions.

Data Abstraction and Encapsulation in C++

Data encapsulation is a mechanism of bundling the data, and the functions that use them and **data abstraction** is a mechanism of exposing only the interfaces and hiding the implementation details from the user.

C++ supports the properties of encapsulation and data hiding through the creation of user-defined types, called classes. We already have studied that a class can contain private, protected and public members. By default, all items defined in a class are private.

The difference between Encapsulation & Abstraction:

- **Encapsulate** hides variables or some implementation that may be changed so often **in a class** to prevent outsiders access it directly. They must access it via getter and setter methods.
- **Abstraction** is used to hiding something too but in a **higher degree(class, interface)**. Clients use an abstract class(or interface) do not care about who or which it was, they just need to know what it can do.
- Encapsulation is used for hide the code and data in a single unit to protect the data from the outside the world. Class is the best example of encapsulation.
- Abstraction refers to showing only the necessary details to the intended user. As the name suggests, abstraction is the "abstract form of anything". We use abstraction in programming languages to make abstract class. Abstract class represents abstract view of methods and properties of class.
- Abstraction is implemented using interface and abstract class while Encapsulation is implemented using private and protected access modifier.

Declaring Class Objects and Invoking Member Functions:

- A class provides the blueprints for objects, so basically an object is created from a class. We declare objects of a class with exactly the same sort of declaration that we declare variables of basic types. Following statements declare two objects of class

  Box:

```
Box Box1;       // Declare Box1 of type Box
Box Box2;       // Declare Box2 of type Box
```

Both of the objects Box1 and Box2 will have their own copy of data members.

- A member function of a class is a function that has its definition or its prototype within the class definition like any other variable. It operates on any object of the class of which it is a member, and has access to all the members of a class for that object.
- Member functions can be defined within the class definition or separately using **scope resolution operator, ::**. Defining a member function within the class definition declares the function **inline**, even if you do not use the inline specifier. So either you can define **getVolume()** function as below:

```
class Box

{

 public:

    double length;     // Length of a box

    double breadth;    // Breadth of a box

    double height;     // Height of a box

    double getVolume(void)

     {

       return length * breadth * height;

     }

};
```

- If you like you can define same function outside the class using **scope resolution operator, ::** as follows:

```
double Box::getVolume(void)

{

  return length * breadth * height;

}
```

- Here, only important point is that you would have to use class name just before :: operator. A member function will be called using a dot operator (.) on a object where it will manipulate data related to that object only as follows:

```
Box myBox;          // Create an object

myBox.getVolume();  // Call member function for the object
```

- Let us put above concepts to set and get the value of different class members in a class:

```
#include <iostream.h>
```

```
using namespace std;
class Box
{
  public:
     double length;       // Length of a box
    double breadth;      // Breadth of a box
    double height;       // Height of a box
    // Member functions declaration
   double getVolume(void);
     void setLength( double len );
     void setBreadth( double bre );
    void setHeight( double hei );
};
// Member functions definitions
double Box::getVolume(void)
{
  return length * breadth * height;
}
void Box::setLength( double len )
{
  length = len;
}
void Box::setBreadth( double bre )
{
  breadth = bre;
}
void Box::setHeight( double hei )
{
  height = hei;
}
// Main function for the program
int main( ) {
  Box Box1;            // Declare Box1 of type Box
  Box Box2;            // Declare Box2 of type Box
  double volume = 0.0;    // Store the volume of a box here
   // box 1 specification
  Box1.setLength(6.0);
  Box1.setBreadth(7.0);
  Box1.setHeight(5.0);
  // box 2 specification
 Box2.setLength(12.0);
  Box2.setBreadth(13.0);
 Box2.setHeight(10.0);
  // volume of box 1
  volume = Box1.getVolume();
  cout << "Volume of Box1 : " << volume <<endl;
  // volume of box 2
  volume = Box2.getVolume();
  cout << "Volume of Box2 : " << volume <<endl;
  return 0;
}
```

In the above example methods name with get are called getter methods, similarly with set called setter methods.

Special class operations:

Constructors & Destructors:

→ The constructor & destructor are special member functions of a class.

→ A constructor is a member function which initializes data members of an object.

→ if a constructor is provided for a class it is automatically executed when an object of that class is created

→ if the constructor is not defined for a class, memory is allocated for the data members of a class object, when it is created, but the data members are not initialized.

→ The advantage of defining constructor for a class is that all class objects are well defined as soon as they are created. This eliminates errors that result from accessing an undefined object.

→ A destructor is a member function which deletes data members immediately before the object disappears.

→ A constructor must be declared as a public member function of its class.:

→ The name of the constructor must be identical to the name of the class to which it belongs

→ constructor must not specify a return type or return value.

example

~~Rectangle :: Rectang~~

```
Box : : Box ( double l,  double b , double h)
  {
     length = l;
     breadth = b;
     height = h;
     }
```

constructors are initialized using Box objects for example

~~Box~~

```
Box  ob ( 1.5, 6 , 5.7 );
```

if you declare

```
Box  ob;
```

it will give compile time error. The reason is that the compiler requires default constructor i.e constructor with no arguments to inialize ob.

→ Destructors are automatically invoked when a class object goes out of scope or when a class object is deleted

→ Like constructor, a destructor must be declared as public member, its name must be identical to the class name ~~of~~ prefixed with tilde (~) operator.

→ The deletion of an object of that class results in the freeing of memory associated with the data members of the class.

→ If a data member is a pointer to some other object, the space allocated to the pointer is returned, but the object that it was pointing to is not deleted.

→ If we also wish to delete this object, we must define a destructor that explicitly does so.

Operator overloading :-

like function overloading, operator overloading is a special feature of C++.

→ C++ allows the programmer to overlad operators for user defined data types. This is done by providing definition that implements the operator for the particular data type.

→ This definition takes the form of a class member function or an ordinary function, depending on the operator.

```
bool  Box :: operator ==( const Box & ob)
  {
      if ( this == &ob)
          return true;
      if ( length == ob.length && breadth == ob.breadth &&
                 height == ob.height )
        {
```

```
    return true;
    }
    else
       return false;
}
```

Miscellaneous Topics:

Structure :-

→ In C++, a struct is identical to a class, except that the default level of access is public; i.e, if the struct definition of a data type does not specify whether the given member has public, private or protected access, then the member has public ~~acc~~ access.

→ In a class the default is private access. Thus the C++ struct is a generalization of the C struct.

Union :-

A union is a structure that reserves storage for the largest of its data members so that only one of its data members can be stored, at any time.

This is useful in applications where it is known that only one of many possible data items, each of a different type, needs to be stored in a structure, but there is no reserve memory for all their data member.

Thus, using a union results in a memory-efficient program, in these cases.

Static class :-

A static class data member may be thought of as a global variable for its class.

→ from the perspective of a class member function, a static data member is like any other data member.

→ one difference is that each class object does not have its own exclusive copy. There is only one copy of a static data member and all class objects must share it.

→ A second difference is that the declaration of a static data member in its class does not constitute a definition. Consequently, a definition of the data member is required somewhere else in the Program.

The Array - as An ABSTRACT DATA TYPE

An array is usually implemented as a consecutive set of memory locations, this is not allways the case. Intuitively an array is a set of pairs, <index, value>, such that each index that is defined has a value associated with it. In mathematical terms we call it as 'a correspondence or a mapping'.

However, when considering an ADT, we are more concerned with the operations that can be performed on an Array.

Aside from creating an new Array, most languages provide only two standard operations for arrays, one that retrieves a value & one that stores a value.

Array ADT (EXAMPLE)

```
Class GeneralArray
{
    /* A set of pairs <index, value> where for each
       value of index in IndexSet there is a value of
       type float. IndexSet is a finit ordered set of one
       or more dimensions. for example {0, --- n-1} for one dimension
       {(0,0),(0,1),(0,2) --- (2,2)} for two dimensions etc */
public:

    GeneralArray (int j, RangeList list, float initValue =
                                  default value);

      /* This constructor creates a j dimensional
      Array of floats, the range of the kth dimension
      is given by the kth element of list. for each
      index i in the index set, insert <i, initValue>
      into the Array. */

    float Retrieve (index i);
      /* if 'i' is in the index set of the array,
      return the float associated with 'i' in the
      array, otherwise throw an exception */

    void Store(index i, float x);
      /* if 'i' is in the index set of the array,
         replace the old value associated with 'i' by
         'x'; otherwise throw an exception. */

    };
```

The polynomial Abstract Data Type:

The problem calls for building an ADT for the representation and manipulation of polynomials in a single variable (x)

Two such polynomials are

$$a(x) = 3x^2 + 2x - 4$$

$$b(x) = x^8 - 10x^5 - 3x^3 + 1$$

→ A term of a polinomial may be represented as a pair (co-efficient , exponent). for example (3, 2) represents term $3x^2$

→ A term whose co-efficient is non zero is called a nonzero term.

Assume that we have two polynomials

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

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

then

$$a(x) + b(x) = \sum (a_i + b_i) x^i$$

$$a(x) \cdot b(x) = \sum \left(a_i x^i \cdot \sum (b_i x^i)\right)$$

ADT polynomial

```
Class Polynomial
{
/* p(x) = a_0 x^{e_0} + ---+ a_n x^{e_n}; a set of ordered pairs
      of <e_i, a_i>, where a_i is a nonzero float
      Coefficient and e_i is a non-negative integer
      exponent */
    Public :
         Polynomial ();
            // construct the polynomial p(x)=0.
          Polynomial Add (Polynomial poly);
        // Return the sum of the polynomials *this & poly.

         Polynomial Mult (polynomial poly);
         // Return the product of the polynomials *this & poly.
         float Eval (float f);
          // Evaluate the polynomial *this at f & return the
                                                        result.
    };
```

Polynomial Representation :-

We are now ready to make some representation decisions. A very reasonable first decision would be a arrange the terms in decreasing order of exponent. We now understand three representations that are base on this principle.

Representation 1:

One way to represent polynomials in C++ is to define the private data members of polynomial as follows:

```
private :
        int degree;          // degree ≤ MaxDegree
        float coef [MaxDegree + 1];   // coefficient array.
```

where MaxDegree is a constant that represents the largest - degree polynomial that is to be represented

Now if a is a polynomial class object and $n \leq$ MaxDegree, then the polynomial $a(x)$ above would be represented as :

a. degree = n

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

Note that a.coef [i] is the coefficient of x^{n-i}

and the co-efficients are stored in order of decreasing exponents. This representation leads to very simple algorithms for many of the operations on polynomials.

Representation 2:

Representation 1 requires us to know the maximum degree of the polynomials we expect to work with & also is quite wasteful in its use of computer memory for instance, if a.degree is much less than MaxDegree, then most of the positions in the array a.coef [] are unused We can overcome both of these deficiencies by defining Coef so that its size is a.degree+1. This can be done by declaring the following private data members

```
private :
        int degree;
        float *coef;
```

and adding the following constructor to Polynomial

```
Polynomial:: Polynomial (int d )
{
    degree = d;
    Coef = new float [degree + 1];
}
```

Representation 3:

Although Representation 2 solves the problems mentioned earlier, it does not yield a desirable representation. To see this, let us consider polynomials that have many zero terms. Such polynomials are called Sparse. For instance the polynomial $x^{1000}+1$ has two non zero terms & 999 zero terms. Consequently, 999 of the entries in coef will be zero if Representation 2 is used. To overcome this problem, we store only the nonzero term for this purpose, we define the class term as below.

```
Class Polynomial ; // forward declaration

class Term
{
friend Polynomial;
private :
        float coef ; //coefficient
        int exp ;
};
```

The private data members of polynomial are defined as follows:

```
private :
        Term * termArray; // array of non zero terms
        int Capacity ;          // size of term Array
        int terms;              // number of non zero terms
```

Before proceeding, we should compare our current representation with Representation 2. Representation 3 is definitely superior when there are many zero terms for example, $c(x) = 2x^{1000} + 1$ uses only 6 units of space (one for c.capacity, one for c.terms, two for the coefficients, and two for the exponents).

→ when we use Representation 3 but when Representation 2 is used, 1002 units of space are required. However, when all terms are non zero, as in $a(x)$ above representation 3 uses about twice as much space as does Representation 2. Unless we know beforehand that each of our polynomials has very few zero terms, Representation 3 is preferable. The exercises explore an alternative representation that uses the STL class vector instead of an array.

Polynomial Addition:-

To add two polynomials, a & b to obtain the sum c = a+b. It is assumed that Representation 3, is used to store a and b.

Function Add adds a(x) (*this) and b(x) term by term to produce c(x). This function assumes that the default constructor for polynomial initializes capacity & terms to 1 & 0, respectively, & initializes termArray to an array with 1 position. (i.e, the size or capacity of termArray is 1).

The basic loop of this algorithm consists of merging the terms of the two polynomials, depending upon the result of comparing the exponents.

The if statement determines how the exponents are related and performs the proper action. Since the tests within the while statement require two terms, if one polynomial runs out of terms, we must exit the loop. The remaining term of the other polynomial can be copied directly into the result. The terms of c are entered into its array termArray by calling function NewTerm.

→ In case there is not enough space in termArray to accommodate the new term, its capacity is doubled.

If we don't have enough memory to create the array temp, new with ~~through~~ throw an exception. Since neither NewTerm nor Add catch this exception, control is passed to the function that invoked 'Add' whenever new fails. If the thrown exception is not caught by any part of the program, the program terminates,

Adding two Polynomials :-

```
Polynomial Polynomial :: Add (Polynomial b)
{
  // Return the sum of the polynomials ~~a~~ this and b.
  Polynomial c;
  int aPos = 0, bPos = 0;
  while ((aPos < terms) && (bPos < b.terms))
    if ((termArray [aPos].exp == b.termArray [bPos].exp)
    {
      float t = termArray [aPos].coef + b.termArray[bPos].coef;
      if (t)
        c.NewTerm (t, termArray [aPos].exp);
        aPos++;
        bPos++;
    }
    else if ((termArray [aPos].exp < b.termArray[bPos].exp)
    {
      c.NewTerm (b.termArray [bPos].coef, b.termArray[bPos].exp);
      bPos ++;
    }
else
  {
    C. NewTerm (termArray [aPos]. coef, termArray[aPos]. exp);
     aPos++ ;
  }
// add in remaining terms of ~~*~~this
for ( ; aPos < terms ; aPos++)
    C. NewTerm(termArray [aPos]. coef, termArray [aPos]. exp);
 // add in remaining terms of b(x)
for( ; bPos < b.terms; b++)
     C.NewTerm (b. termArray [bPos]. coef, b. termArray [bPos]. exp);
 return C;
}
```

Adding a new-term, doubling array size when necessary

```
Void Polynomial :: NewTerm ( const float theCoeff, const int theExp)
{
 // Add a new term to the end of termArray
      if (terms == capacity)
       {
         // double capacity of termArray
           capacity ~~*~~ =2;
           term ~~*~~term= new term [capacity]; // new Array
```

```
        copy (termArray, termArray + terms, temp);

        delete [] termArray;   // deallocate old memory.

        termArray = temp;

    }

    termArray [terms]. coef = the Coeff;

    termArray [terms ++]. exp = theExp;

}
```

→ Total runtime of Add is $O(m+n)$

Sparse Matrix:-

A matrix having many zero's called sparse.

ADT SparseMatrix:

class SparseMatrix

{

/* A set of triples, <row, column, value>, where row & column are non negative integers & form a unique combination; value is also an integer */

public:

SparseMatrix (int r, int c, int t);

/* The constructor function creates a sparse Matrix with r rows, c columns, & a capacity of t nonzero terms. */

SparseMatrix Transpose();

/* returns the SparseMatrix obtained by interchanging the row & column value of every triple in *this */

SparseMatrix Add (SparseMatrix b);

/* If the dimensions of *this and b are the same, then the matrix produced by adding corresponding items, namely those with identical row & column values is returned; otherwise, an exception is thrown

```
SparseMatrix Mutiply(SparseMatrix b);
    /* if the number of columns in *this equals
the number of rows in b then the matrix d produced by
multiplying *this & b according to the formula d[i][j] =
    Σ(a[i][k] · b[k][j]), where d[i][j] is the (i,j)th
element, is returned. k ranges from 0 to one less than the
number of columns in *this; otherwise, an exception is thrown */
};
```

Sparse Matrix representation :

We know that we can characterize uniquely any element within a matrix by using the triple <row, col, value>. This means that we can use an array of triples to represent a sparse matrix.

→ we require that these triples be stored by rows with the triples for the first row first, followed by those of the second row & so on.

→ we also require that all the triples for any row be stored so that the column indices are in ascending order.

putting all this information together suggests that we define

```
class SparseMatrix; //forward declaration

class MatrixTerm
{
   friend class SparseMatrix;
   private:
      int row, col, value;
};
```

and in class SparseMatrix:

```
private:
     int rows, cols, terms, capacity;
     MatrixTerm *smArray;
```

where rows is the number of rows in the matrix,
cols is the number of columns
terms is the total number of non-zero entries
capacity is the size of smArray

Ex: Sparse Matrix

	row	col	value
smArray[0]	0	0	15
[1]	0	3	22
[2]	0	5	-15
[3]	1	1	11
[4]	1	2	3
[5]	2	3	-6
[6]	4	0	91
[7]	5	2	28

Transposing a Matrix:-

To transpose a matrix, we must interchang the rows & columns. This means if an element is at position [i][j] in the original matrix, then it is at position [j][i] in the transposed Matrix.

When i = j the elements on the diagonal will remain unchanged.

The transpose algorithm might be the following:

```
for(each row i)
        store(i, j, value) of the original matrix as (j, i, value)
of the transpose;
```

(0, 0, 15)	beomes	(0, 0, 15)
(0, 3, 22)	"	(3, 0, 22)
(0, 5, -15)	"	(5, 0, 15)
(1, 1, 11)		(1, 1, 1'

we can avoid this difficulty of not knowing where to place an element by changing the order in which we place elements into the transpde.

Consider following stratery.

```
for(all elements in column)
        store(i, j, value) of the original matrix as
                (j, i, value) of the transpose;
```

```
SparseMatrix  SparseMatrix :: Transpose()
{
/* Return  the  transpose  of  *this. */
        SparseMatrix  b( cols , rows, terms);
        // capacity of  b.smArray is  terms
        if (terms >0)
          {
             // nonzero  Matrix
               int  currentB=0;
                for (int  c=0;  c<cols ; c++)
                 // transpose  by  columns
                for (int  i=0; i<terms;  i++)
                   // find & move  terms in  column c
                     if ( smArray[i]. col = = c)
                     {
                          b. smArray [ current B ]. row = c;
                          b. smArray [ Current B ]  . col = ~~[illegible]~~
                                              smArray [i] . row ;

                   b.smArray [currentB++]. value =
                                          smArray [i]. value
                   }
               } // end of  if (terms>0)
               return  b;
             }
```

Matrix Multiplication

Definition: Given a & b, where a is m×n & b is n×p, the product matrix d has dimension m×p. Its [i][j] element is

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

for $0 \leq i < m$ & $0 \leq j < p$.

Sparse Matrix Multiplication

Sparse matrices, which are common in scientific applications, are matrices in which most elements are zero. To save space and running time it is critical to only store the nonzero elements. A standard representation of sparse matrices in sequential languages is to use an array with one element per row each of which contains a linked-list of the nonzero values in that row along with their column number. A similar representation can be used in parallel. In NESL a sparse matrix can be represented as a sequence of rows, each of which is a sequence of `(column-number, value)` pairs of the nonzero values in the row. The matrix

$$A = \begin{matrix} 2.0 & -1.0 & 0 & 0 \\ -1.0 & 2.0 & -1.0 & 0 \\ 0 & -1.0 & 2.0 & -1.0 \\ 0 & 0 & -1.0 & 2.0 \end{matrix}$$

is represented in this way as

```
A = [[(0, 2.0), (1, -1.0)],
     [(0, -1.0), (1, 2.0), (2, -1.0)],
     [(1, -1.0), (2, 2.0), (3, -1.0)],
     [(2, -1.0), (3, 2.0)]]
```

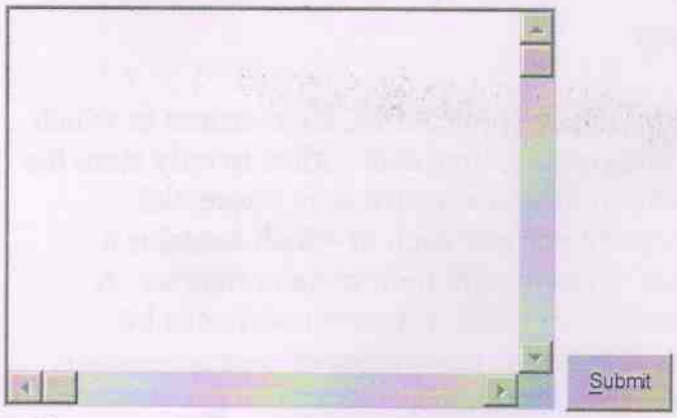
where `A` is a nested sequence. This representation can be used for matrices with arbitrary patterns of nonzero elements since each subsequence can be of a different size.

A common operation on sparse matrices is to multiply them by a dense vector. In such an operation, the result is the dot-product of each sparse row of the matrix with the dense vector. The NESL code for taking the dot-product of a sparse row with a dense vector `x` is:

```
sum({v * x[i] : (i,v) in row});
```

This code takes each index-value pair `(i,v)` in the sparse row, multiplies `v` with the `i` *th* value of `x`, and sums the results. The work and depth is easily calculated using the performance rules. If *n* is the number of non-zeros in the row, then the depth of the computation is the depth of the `sum`, which is *O(log n)*, and the work is the sum of the work across the elements, which is *O(n)*.

The full code for multiplying a sparse matrix *A* represented as above by a dense vector `x` requires that we apply the above code to each row in parallel, which gives



This example has nested parallelism since there is parallelism both across the rows and within each row for the dot products. The total depth of the code is the maximum of the depth of the dot products, which is the logarithm of the size of the largest row. The total work is proportional to the total number of nonzero elements.

```
functionsparse_matrix_mult(A,x) =

 {sum({v * x[i] : (i,v) in row})

  : row in A};


% An example matrix and vector %

A = [[(0, 2.0), (1, -1.0)],

   [(0, -1.0), (1, 2.0), (2, -1.0)],

   [(1, -1.0), (2, 2.0), (3, -1.0)],

   [(2, -1.0), (3, 2.0)]];

x = [1.0, 0.0, -1.0, 0.0];


% Try it out %

sparse_matrix_mult(A,x);
```

Program:

```
#include<iostream>
#include<process.h>
#include<conio.h>
using namespace std;
//This program implements the linked list
representation of Sparse matrix
// for multiplying two Sparse Matrices
class Sparse
{
inti,j,k;
public:
Sparse *next;
Sparse()
{
i=j=k=0;
}
Sparse(inti,int j)
{
this->i=i;
this->j=j;
}
voidSetNonZeroElements(Sparse *head,int k)
{
head->k=k;
}
void accept(inti,intj,int k)
{
this->i=i;
this->j=j;
this->k=k;
}
void display()
{
cout<<i<<" "<<j<<" "<<k<<endl;
}
intvalidateSparseMultiplication(Sparse
*head1,Sparse *head2)
{
if (head1->j!=head2->i)
return 0;
else
return 1;
}
intMaxRow(Sparse *head)
{
if (head!=NULL)
return head->i;
else
return 0;
}
intMaxCol(Sparse *head)
{
if (head!=NULL)
return head->j;
else
return 0;
}
intRCValueExists(int ,int ,Sparse *);
voidAddToElementIJ(int,int,int,Sparse *);
};
int Sparse::RCValueExists(inti,intj,Sparse *head)
{ Sparse *ptr=head->next;
for(;ptr!=NULL;ptr=ptr->next)
{
if ((ptr->i==i) && (ptr->j==j))
returnptr->k;
}
return 0;

}
void
Sparse::AddToElementIJ(inti,intj,intk,Sparse
*head)
{
Sparse *ptr=head->next;
for(;ptr!=NULL;ptr=ptr->next)
{
if ((ptr->i==i) && (ptr->j==j))
ptr->k=ptr->k+k;
return;
}
return;
```

```
}
int main()
{
intr,c,i,j,k,ctr;
Sparse
*A,*B,*C,*start1,*start2,*start3,*ptr1,*ptr2;

//Accept Details Regarding First Matrix & Its
Elements
cout<<endl<<"Enter no of rows in first sparse
matrix : ";
cin>>r;
cout<<endl<<"Enter no of columns in first
sparse matrix : ";
cin>>c;
cout<<endl<<"\t"<<"Enter elements of matrix
A"<<endl<<endl;
ctr=0;
A=new Sparse(r,c);
start1=A;
for(i=1;i<=r;i++)
for(j=1;j<=c;j++)
{
cout<<"Enter element of "<<i<<"th row and
"<<j<<"th column : ";
cin>>k;
if (k!=0)
{
A->next=new Sparse();
A=A->next;
A->accept(i,j,k);
ctr++;
}
}
A->next=NULL;
A->SetNonZeroElements(start1,ctr);
//Accept Details Regarding Second Matrix B &
Its Elements
cout<<endl<<"Enter no of rows in second
sparse matrix : ";
cin>>r;
cout<<endl<<"Enter no of columns in second
sparse matrix : ";
cin>>c;
cout<<endl<<"\t"<<"Enter elements of matrix
B"<<endl<<endl;
ctr=0;
B=new Sparse(r,c);
start2=B;
for(i=1;i<=r;i++)
for(j=1;j<=c;j++)
{
cout<<"Enter element of "<<i<<"th row and
"<<j<<"th column : ";
cin>>k;
if (k!=0)
{
B->next=new Sparse();
B=B->next;
B->accept(i,j,k);
ctr++;
}
}
B->next=NULL;
B->SetNonZeroElements(start2,ctr);

cout<<endl<<"\t"<<"Sparse Matrix A"<<endl;
//Display stored elements of Matrix A
for(ptr1=start1;ptr1!=NULL;ptr1=ptr1->next)
ptr1->display();
cout<<endl<<"\t"<<"Sparse Matrix B"<<endl;
//Display stored elements of Matrix B
for(ptr1=start2;ptr1!=NULL;ptr1=ptr1->next)
ptr1->display();
//Validate Matrix Multiplication
if (A-
>validateSparseMultiplication(start1,start2)==0)
{
cout<<"Number of columns in Matrix A should
be equal to"<<endl;
cout<<"Number of rows in Matrix B"<<endl;

}
```

```
C=new Sparse(r,B->MaxCol(start2));
start3=C;
ctr=0;
for(i=1;i<=A->MaxRow(start1);i++)
for(j=1;j<=A->MaxCol(start1);j++)
for(k=1;k<=B->MaxCol(start2);k++)
{
if (A->RCValueExists(i,j,start1)!=0 && B-
>RCValueExists(j,k,start2)!=0)
{
if (C->RCValueExists(i,k,start3)==0)
{
C->next=new Sparse(i,k);
C=C->next;
C->accept(i,k,A->RCValueExists(i,j,start1)*B-
>RCValueExists(j,k,start2));
}
else
{
C->AddToElementIJ(i,k,A-
>RCValueExists(i,j,start1)*B-
>RCValueExists(j,k,start2),start3);
}
ctr++;
}
}
C->next=NULL;
C->SetNonZeroElements(start3,ctr);
cout<<endl<<"\t"<<"Resultant Matrix"<<endl;
for(ptr2=start3;ptr2!=NULL;ptr2=ptr2->next)
ptr2->display();
}
```