

ADT and Arrays: Chapter 2 Sparse Arrays and Sparse Matrices

Computer Science B.Sc
Paper No and Name: CSHT203: Data Structures
Unit No : II
Chapter No and Name: ADT and Arrays



Fellow: Dr. Anita goel, Associate Professor
College/Department: Dyal singh College, University of Delhi

Author:- Parul Chachra, Assistant Professor
College/Department: College of Vocational Studies , University of Delhi

Reviewer: Dr. Archana Singhal, Associate Professor
Collge/Department: Indraprastha College , University of Delhi

Table of Contents

- ADT And Arrays
 - 1.1 What is ADT?
 - 1.1.1 Parts of ADT
 - 1.1.2 Data Structure
 - 1.1.3 Eample of ADT
 - 1.2 Arrays
 - 1.2.1 Storing vaiues in Arrays
 - 1.2.2 Operation on Arrays
 - 1.2.3 Sequwntlal Allocation
 - 1.2.4 Pointer and Arrays
 - 1.3 Single Dimension Arrays
 - 1.3.1 Memory Allocation of Arrays
 - 1.4 Multi Dimension Arrays
 - 1.4.1 memory Allocation of Arrays
 - 1.5 Two Dimension Arrays
 - 1.5.1 Memory Allocation of Arrays
 - 1.5.2 Matrix Class
 - 2.1 What is a Matrix?
 - 2.1.1 Application of Matrices
 - 2.1.2 Storing Matrices in Arrays
 - 2.1.3 Special Matrices
 - 2.2 What is a Sparse Matrix?
 - 2.3 Single Dimension Arrays and Sparse Matrices
 - 2.3.1 Mapping of Non-Zero Elements
 - 2.4 Row – Major Mapping
 - 2.4.1 Array Representation using Row - Major
 - 2.5 Column – Major Mapping
 - 2.5.1 Array Representation
 - 2.5 Adding Two Matrics
 - 2.6 Linked Representation of sparse Matrices
 - 2.6.1 Linked Lists
 - 2.6.2 Linked Lists and Sparse Matrices
 - 2.6.3 Linked Lists Vs Arrays
 - 2.7 Sparse Arrays
 - 2.7.1 Sparse Arrays using One – Dimension Arrays
 - 2.7.2 Operations on Sparse Arrays
 - 2.7.3 Sparse Arrays using Two – Dimension Arrays
 - 2.8 Efficient Representation
 - Summary
 - Refences
 - Exercises
 - Glossary

1.1 What is ADT?

Abstract Data Type (ADT) is a collection of data and the operations to be performed on the data. ADT is independent of the implementation, which means, it is logical in nature. The definition of ADT is not concerned with time or space efficiency because these are the implementation related issues. ADT is not concerned about the physical behavior of the structure.

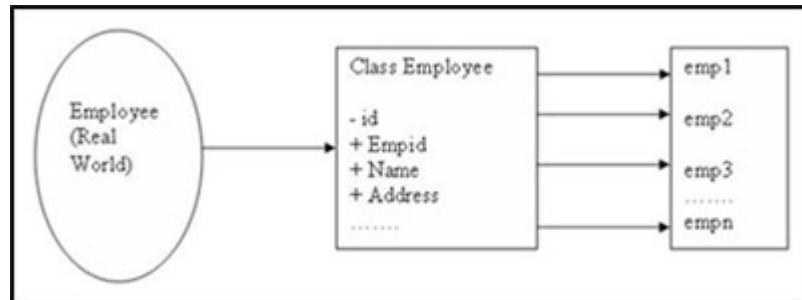


Figure 1.1: Object hierarchy (Oval Shows the real world, middle square shows the class made from the real world and last square shows the different object of this class)

Source: <http://www.codeproject.com/KB/architecture/OO.aspx>

Abstract Data Type is required so as to define a structure which can be implemented independent of the programming language. If we define a data structure in a particular programming language, the data structure needs to be recoded to shift it to another programming language. Also, new programming languages keep coming up now and then, defining a data structure in a language will be time – consuming as it has to be recoded. Thus, ADT is simply a specification and this specification can be implemented in any programming language with ease.

1.1.1 Parts of ADT

Value addition: Frequently Asked Questions

Heading text What is Abstract Data Type

Body text:



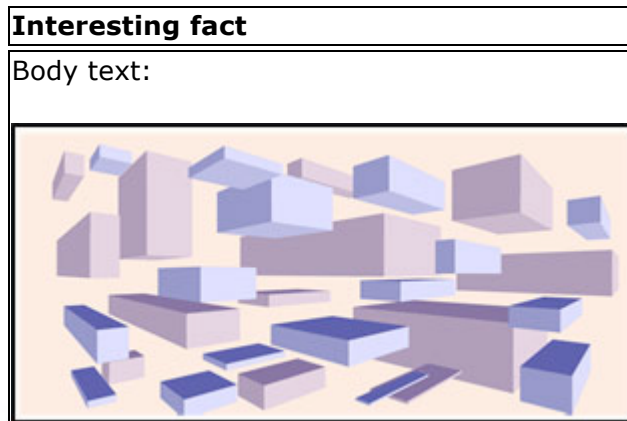
An Abstract Data Type (ADT) is any reasonably complex type that isn't one of the standard types- int, float, string or array. A set of data values and associated operations that are precisely specified independent of any particular implementation.

Source: <http://cplusplus.about.com/od/glossar1/g/adtdefinition.htm>

ADT consists of two parts – data and operation. Data holds the data which the structure holds and operation is the set of function that can be performed on the data. Data can further be a complex structure, comprising of different variables.

The data part in the ADT consists of an index and a value. The index defines the position where the value will be stored. No two indexes value pair will be the same. The operations are performed on the data part.

1.1.2 Data Structure



The data is stored in a particular format so that the data is properly arranged. This arrangement of data in a particular structure is known as a data structure. The data needs to be stored in a particular format not only for the proper storage, but mostly for quick and easy retrieval.

Value addition: Frequently Asked Questions (FAQs)
Heading text Types of Data Structures
Body text: Homogeneous / Heterogeneous: In homogeneous data structures all elements are of the same type, e.g., array. In heterogeneous data structures elements are of different types, e.g. structure. Static / Dynamic: In static data structures the size can not be changed after the initial allocation, like matrices. In dynamic data structures, like lists, size can change dynamically. Linear / Non-linear: Linear data structures maintain a linear relationship between their elements, e.g., array. Non-linear data structures do not maintain any linear relationship between their elements, e.g., in a tree. Source: http://en.wikipedia.org/wiki/Data_structure

The data should be stored in a particular format so that the data is properly organised and the data is available as and when the user needs it. The retrieval time of the data should be minimized.

For Example, Linked List is a data structure where data is stored in a node format. Each node has data and pointer or simply a link to the next node. A pointer is a variable which stores the address of another variable.

ADT and Arrays: Chapter 2 Sparse Arrays and Sparse Matrices

Each type of data structure has pros and cons associated with its properties and nature of storing the data. Each new data structure was brought out to overcome the shortcomings of the previous data structure.

ADT is abstract data type, a storage format which is implementation – independent. When we add the implementation details to the ADT, we get a data structure. ADT is universal and common to all programming languages, whereas, the data structure is specific to a particular programming language which involves the organization of data in terms of storage allocation. ADT is a logical description and data structure is concrete. ADT is the logical picture of the data and the operations to manipulate the component elements of the data. Data structure is the actual representation of the data during the implementation and the algorithms to manipulate the data elements. ADT is in the logical level and data structure is in the implementation level. A change of programming language for a data structure would thus require re – coding the entire data structure again.

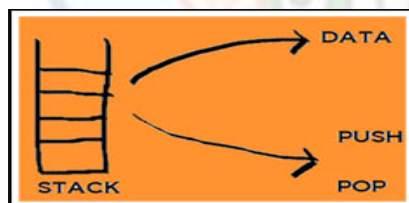
1.1.3 Example of ADT

We can say that Stack is an ADT. The data part is the value for which the stack is being created, say number. The two operations that can be performed are:

- push – for inserting the values in the stack,
- pop – for retrieving the values from the stack.

Stack as an ADT performs two standard operations, which is universal and independent of implementation. Thus, Stack using arrays is a data structure because it is implemented using arrays.

Value addition: Frequently Asked Question
Heading text Example of Abstract Data Type
Body text:



OPERATIONS TO BE
PERFORMED ON THE STACK

The stack is a data structure which works on the concept of Last – in – first – out. For Example, when we go to a dinner party, the fresh plates are stacked one on top of the other. We have to take the plate from the top of the stack. Also, if new plate has to be placed, it will go on top again. Same thing is implemented in stack as ADT.

Source: self-made

Stack as a Data Structure

Assuming stack is implemented in C++ computer programming language using Arrays, the stack as a data structure can be defined as follows:

```
#define MAX_SIZE 10
class stack
{
    int arr_stck[MAX_SIZE];
    int tos;
```



```

public:
void push(int value)
{
    if(tos == MAX_SIZE)
    {
        cout<<"Stack Overflow";
        exit(0);
    }
    else
        arr_stck[++tos]=value;
}

int pop()
{
    if(tos == -1)
    {
        cout<<"Stack Underflow";
        exit(0);
    }
    else
        return arr_stck[tos--];
}
}
    
```

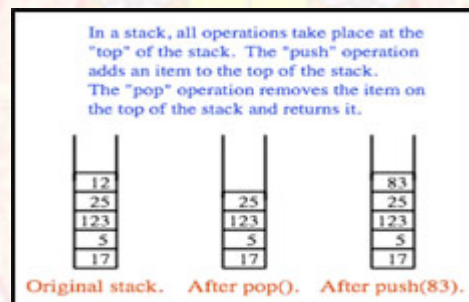


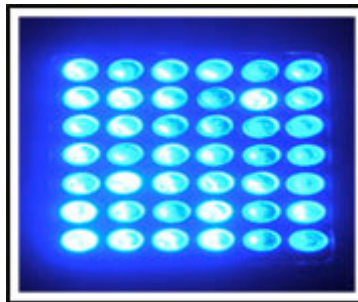
Figure 1.2: Stack – working with push and pop
Source: <http://math.hws.edu/javanotes/c9/s3.html>

1.2 Arrays

Value addition: Image

Heading text Arrays

Body text:



Source: <http://www.elixa.com/light/arrays.htm>



Figure 1.3: Arrays

Source: <http://hubpages.com/hub/Arrays-of-Characters-in-Cpp>

An array is a variable that holds multiple values of the same data type. Nearly every programming language has arrays, which are of great help in solving complex problems, like matrix, easily. Array is the contiguous memory allocation of the same data type. All elements of the same data type are stored at one place. If we know the starting address of the array, we can reach the other elements because each element is adjacent to the other. The elements are stored in a definite order.

Also, we can directly access any required element by using the appropriate index into the array. Data is stored and accessed using indexes. An index is simply an address or place in the array; it starts with 0th position and goes till $n-1$ (where n is the size or upper bound of the array). There exists an upper bound to an array, thus it is finite.

For Example,

```
int arr[10];
```

The above statement depicts the following:

- Defines an array of size 10.
- Each element is of Integer data type.
- The array is defined with name "arr".
- All 10 elements are adjacent to each other, i.e. 10 contiguous memory locations are allocated to this array. The arrangement of data in the array would be like this

```
arr[0] arr[1] arr[2] arr[3] arr[4] arr[5] arr[6] arr[7] arr[8] arr[9]
```

An array can be declared using any valid data type. Each data type has a specific size and the C++ compiler uses the size of the data type and the number of elements in the array, to allocate memory to the array.

An array can be of two types –

- Single – dimension arrays – which take only one dimension
- Multi – dimension arrays – which take more than one dimensions

Did you know?

Body Text: Text

Array structures were used in the first digital computers, when programming was still done in machine language, for data tables, vector and matrix computations, and many other purposes. Von Neumann wrote the first array sorting program (merge sort) in 1945, when the first stored-program computer was still being built. Array indexing was originally done by self-modifying code, and later using index registers and indirect addressing. Some mainframes designed in the 1960s, such as the Burroughs B5000 and its successors, had special instructions for array indexing that included index bounds checking.

Assembly languages generally have no special support for arrays, other than what the machine itself provides. The earliest high-level programming languages, including FORTRAN(1957), COBOL(1960), and ALGOL 60 (1960), had support for multi-dimensional arrays.

Notation	Languages
rates(2, 3)	FORTRAN, BASIC, COBOL, Ada
rates[2][3]	Java, C++, C#, Modula-2
rates[2, 3]	C#, Pascal, Modula-2

Array Notation by Language

Language	Element range	# of elements
FORTRAN, PL/I	1-10	10
C++/Java/C#	0-9	10
VB .Net/QBasic	0-10	11

Array Elements by Language

Source: http://en.wikipedia.org/wiki/Array_slicing and <http://hhh.gavilan.edu/dvantassel/history/arrays.html>

1.2.1 Storing values in Arrays

There are different ways of inserting values in the array.

- The value to be inserted may be accepted from the user.

```
int arr1[10];
int ctr;
for(ctr=0;ctr<10;ctr++)
    cin>>arr1[i];
```

- The array can be initialized with the required values at the time of declaration of the array.

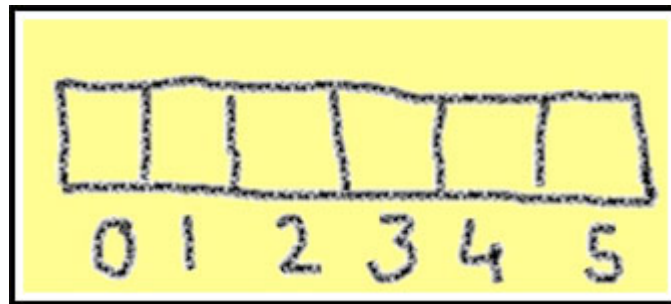
```
int arr1[] = { 4, 5 ,6, 7, 8, 9, 10, 11, 12, 13, };
```

- Each element of the array can be individually assigned a value.

```
int arr1[10];
arr1[0]=4;
arr1[1]=5;
```



```
arr1[2]=6;  
arr1[3]=7;  
arr1[4]=8;  
arr1[5]=9;  
arr1[6]=10;  
arr1[7]=11;  
arr1[8]=12;  
arr1[9]=13;
```



Each array is allocated adjacent memory equivalent to the base data type used to declare the array. Each memory segment or partition is given an address. We can't know the addresses in advance, so we are given the array name as the reference and the indexes that we use to access the elements act as an offset from the start of the array. The starting address is given to the array name which is actually a pointer variable.

Source: self-made

C++ does not perform any bounds checking on arrays. If the user has declared an array to be of size 10 and he enters more than 10 values, C++ compiler will not generate any error in the code. However, this is illegal and the programmer must provide bound – checking in the code itself.

1.2.2 Operations on Arrays

Access Element

The operation of retrieving elements from the array is known as accessing the elements. It can also be termed as “searching” the array for a value. This operation can be defined as the act of starting with the first element, i.e. 0th position and going onto the last element, i.e. nth position (assuming the array has n elements) until the required value is found.

For Example, `arr[i]` returns the value stored at position `i` in the array named “arr”.

Store Elements

The operation of storing elements in the arrays is known as storing elements. This operation can also be termed as “insertion array. The value has to be inserted at the appropriate index depending on the requirements. The value is inserted either in a definite order or at the index accepted from the user.

For Example, `arr[i] = j`, puts the value specified by variable “j” at the `i`th position in the array named “arr”.

Search Elements

The operation of searching elements in the arrays for the maximum or minimum is known as searching. The search can also be for whether a particular element is present in the array. The array is traversed from the first element to the end of the array. Each value is compared to a pre – initialized variable "maximum" or "minimum" to check whether it is greater than or less than that value. If so, then these variables are updated to hold the current values.

Did you know?

Body Text:

```
// arrays as parameters
#include <iostream>
using namespace std;

void printarray (int arg[], int length) {
for (int n=0; n<length; n++)
    cout << arg[n] << " ";
    cout << "\n";
}
```

```
int main ()
{
    int firstarray[] = {5, 10, 15};
    int secondarray[] = {2, 4, 6, 8, 10};
    printarray (firstarray,3);
    printarray (secondarray,5);
    return 0;
}
```

Output:

```
5 10 15
2 4 6 8 10
```

Source:<http://www.cplusplus.com/doc/tutorial/arrays/>

1.2.3 Sequential Allocation

An array has contiguous elements of the same data type. At the time of its declaration, the C++ compiler finds the right amount of memory and allocates that chunk of memory to the array pointer or simply put, to the name of the array. The basic nature of the array requires that the allocation of memory should be sequential. The index is just the offset from the starting address of the array which is stored in the name of the array.

The array pointer points to the first element of the array which is the 0th element of the array. If the user requires accessing the 8th element, say for example, the user has to traverse from the first element to the 8th element. This is called sequential access.

```
int arr1[10];
```

ADT and Arrays: Chapter 2 Sparse Arrays and Sparse Matrices

```
int i;  
  
//accepting values  
  
for(i=0;i<10;i++)  
  
    cin>>arr1[i];  
  
//displaying the array  
  
for(i=0;i<10;i++)  
  
    cout<<arr1[i];
```

The array accepts values from the user one – by – one and then displays sequentially, starting from the first element and going on till the last element.

Intresting fact

Body text:

Arrays permit efficient (constant time, $O(1)$) random access but not efficient insertion and deletion of elements (which are $O(n)$, where n is the size of the array). Linked lists have the opposite trade-off.

Consequently, arrays are most appropriate for storing a fixed amount of data which will be accessed in an unpredictable fashion, and linked lists are best for a list of data which will be accessed sequentially and updated often with insertions or deletions.

Another advantage of arrays that has become very important on modern architectures is that iterating through an array has good locality of reference, and so is much faster than iterating through (say) a linked list of the same size, which tends to jump around in memory. However, an array can also be accessed in a random way, as is done with large hash tables, and in this case this is not a benefit.

Arrays also are among the most compact data structures; storing 100 integers in an array takes only 100 times the space required to store an integer, plus perhaps a few bytes of overhead for the whole array. Any pointer-based data structure, on the other hand, must keep its pointers somewhere, and these occupy additional space. This extra space becomes more significant as the data elements become smaller.

For example, an array of ASCII characters takes up one byte per character, while on a 32-bit platform, which has 4-byte pointers, a linked list requires at least five bytes per character. Conversely, for very large elements, the space difference becomes a negligible fraction of the total space.

Because arrays have a fixed size, there are some indexes which refer to invalid elements — for example, the index 17 in an array of size 5. What happens when a program attempts to refer to these, varies from language to language and platform to platform.

Source: <http://www.experiencefestival.com/a/Array/id/1918703>

1.2.4 Pointer and Arrays

Another point to be highlighted here is, after the declaration, the pointer variable ptr should not be directly used. The pointers should not be used without assigning any values, as they can point to any junk value, and working with un – initialized pointers can wreak havoc on the system.

Pointer Operators

There are two operators which are used with pointers:

- '&' - is a unary operator which returns the memory address of its operand.
- '*' - is a unary operator which returns the value stored at the address that follows this operator.

For Example,

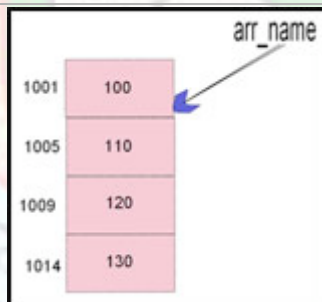
```
int *ptr;
int x=10;
ptr=&x;
cout<<ptr<<*&ptr;
```

Assuming the variable x is located at memory location 1001, the output of the above code will be 1001, 10. '&' is used to place the address of the variable x in ptr and '*' is used to display the value at memory location 1001 i.e. the value of the variable x.

Array Name

Did you know?

Array Name is a Pointer



A pointer variable points to the address of the another variable. If we know the address of a variable, we can easily access its values. Also, the pointers allow the user to do arithmetic like ++ and -- to manipulate the addresses. The array name is a pointer variable which points to the base address of the actual memory location of the array. This is required because the memory is not allocated when we are writing the code and declaring the array. We need some mechanism to access the elements of the array and this mechanism is the pointer.

Source: self – made

Array name is considered to be a pointer itself. It points to the starting address of the array. The memory is allocated sequentially to the array so the elements of the array can easily be accessed once we have the starting address. The name of the array can also be assigned to other pointer variables.

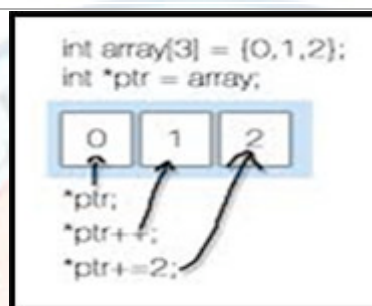
ADT and Arrays: Chapter 2 Sparse Arrays and Sparse Matrices

The pointer arithmetic plays a vital role in the arrays. Let's say, the array is of the integer data type and the system takes 32 bits or 4 bytes for the integer data type. So if we increment a counter variable (which is looping over to access the elements of the array) of integer data type, only then we reach the next element. If we declare the counter variable of Character data type, the increment will be of 1 byte, whence we need the increment to be of 4 bytes.

When the pointer to the array is of the base type of the array, the increment will be relative to the base type and it can access the next element. If the pointer variable is not of the base type, the increment will take the pointer to the next element.

Interesting fact

Pointers and Arrays



An integer array is being created with a size of 3. The numbers 0,1 and 2 are being loaded into the array and then a pointer is then getting pointed to the array. At the bottom are examples of using pointer arithmetic to access array elements. Calling `*ptr++` twice is the same thing as calling `*ptr+=2`.

Behind the scenes the address is merely being increased by the size of the data type. Let's say an int is 32 bits (4 bytes) long, then when you increment a pointer, you are merely adding 32 bits or 4 bytes to your memory address. Giving you the starting point of the next element in the array.

Source: <http://morethanmachine.com/macdev/?s=pointer+and+array>

1.3 Single Dimension Arrays

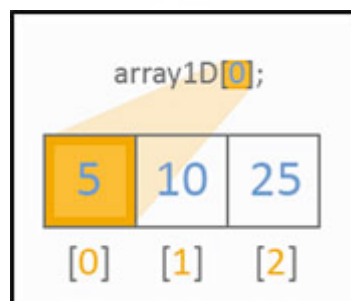


Figure 1.5: Array dimensions

Source: <http://articles.mql4.com/417>

A one – dimensional or Single Dimension Array has just one index for inserting into or accessing the array. This array stores values pertaining to just one variable and all the

ADT and Arrays: Chapter 2 Sparse Arrays and Sparse Matrices

values are essentially of the same data type. This is so because the same data type is used to declare the array of n elements.

For Example, `arr[i]` is a one – dimensional array in which data for just one variable, say x, will be stored.

We can define a single dimension array as follows:

```
Data_type array_name[max_size_of_array];
```

Here, `data_type` defines the C++ data type using which the array has to be declared, for example `int`, `char`, etc. `Array_name` is any valid variable name which will point to the memory location allocated by the compiler. `Max_size_of_array` is the number of elements this array should hold.

To declare an array named "frequency" of integer data type of 20 elements, the statement is as follows:

```
int frequency[20];
```

The memory for this array is sequentially allocated, i.e. if the array size is 10, 10 adjacent memory units will be allocated to this array. Each memory unit is of the size of the data type to be used for declaring this array.

Did you know?

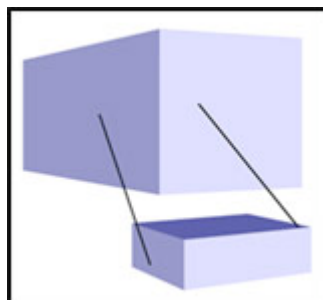
Array Slicing

Body text:

Array Slicing is an operation that extracts certain elements from an array and packages them as another array, possibly with different number of indices (or dimensions) and different index ranges.

Two common examples are extracting a substring from a string of characters (e.g. "ell" from "hello"), and extracting a row (or a column) of a rectangular matrix to be used as a vector.

The concept of slicing was known even before the invention of compilers. Slicing as a language feature started with FORTRAN (1957). Since then, array slicing features have been incorporated in several modern languages such as Python, Perl, and Cobra. The slicing can be used when we don't want the entire array to be used, but we don't want to create a new array.



Source: http://en.wikipedia.org/wiki/Array_slicing

1.3.1 Memory Allocation of Arrays

The amount of memory allocated to an array is directly based on the data type used for declaring the array and the size of the array. For a single dimension array, the amount of memory allocated is calculated as follows:

$$\text{Memory allocated} = \text{sizeof}(\text{data_type}) \times \text{max_size_of_array}$$

For Example, the memory allocated to the array "frequency" is calculated thus,

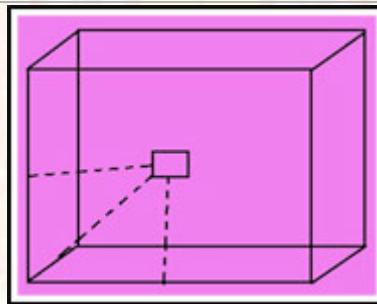
$$\text{Memory allocated} = \text{sizeof}(\text{int}) \times 20 = 32 \times 20 = 640 \text{ bits}$$

The single dimension array has values pertaining to one variable only. The memory allocated is the product of the size of the base data type and the size of the array. The size of the base data type varies depending on the C++ compiler, for example, the size of integer data type is 16 bits on some systems and 32 bits on others.

1.4 Multi Dimension Arrays

Did you know?

Dimension Array



Each multi – dimension array can have n number of dimensions to it, though more than 4 dimensions are rarely used. This is so because, more the number of dimensions, more the complexity in the code. Also, the chance of error also increases.

The element in a multi – dimension array is recognized by the convergence point of all the dimensions.

Source: self - made

A Multi Dimension Array has more than one index for inserting into or accessing the array.

The limit on the number of indexes used in an array is imposed by the compiler. The multi – dimension arrays have the advantage of analyzing data in tabular format as we have many variables interacting with each other at one place. The more dimensions we have in the array, the more variables are stored at one place.

The syntax for declaring a multi dimension array is as follows:

```
Data_type array_name[index1][index2]...[indexn];
```

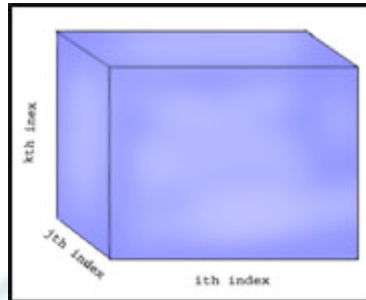
ADT and Arrays: Chapter 2 Sparse Arrays and Sparse Matrices

Indexes can vary from 2 to n depending on the requirement, however, indexes more than 3 are hardly used because of their complexity and the amount of memory required storing them.

Value addition: Image

Heading text Indexes in a Multi – Dimension Array

Body text:



Source: self – made

1.4.1 Memory Allocation of Arrays

The amount of memory allocated to an array is based on the base type used to define the array and the number of dimensions, the array has.

For Example, if we add 5 dimensions to the array “frequency”, say, frequency[10][7][7][3][3], the memory allocated will be calculated as follows :

$$\begin{aligned}\text{Memory allocated} &= \text{sizeof(int)} \times \text{size of the array} \\ &= 32 \times 10 \times 7 \times 7 \times 3 \times 3 = 141120 \text{ bits}\end{aligned}$$

The above situation requires 141120 bits and needless to say all of these bits have to be adjacent to each other. If adjacent chunk of memory is not available, then this array cannot be declared and used.

Did you know?

Dimension Array

Body Text:

Tic – Tac – Toe is a game which we have been playing for so many years, since our childhood. It’s a game where we use crosses and circles to make a line of the same symbol. If we have 3 adjacent symbols making a horizontal, vertical, or diagonal line, we win; else, we lose. We can simulate this game using two - dimension arrays.

The two – dimension arrays are commonly used to simulate the board game matrices. When the computer makes a move, it simply scans the board for an empty cell and place a 'O' in that empty cell. If it can't find an empty space, the computer simply draws the game. When the user has to make a move, the user is asked to accept the cell where to place the 'X'. There are 8 possible ways to win this game.

The two – dimension array is used to store the tic tac toe matrix, which is updated as and when the two players make their move. The implementation of this game has eased

off a lot with the use of arrays.

Algorithm for tic tac toe game:

- have a two dimensional array (of the order of $n \times n$).
- the player will have to provide the row and column to place the symbol of 'X' or 'O' on the game board
- the game would check whether the cell asked for by the user is empty
- if it is empty, place the piece in the cell
- check to see if it was a winning move
 - see if the other three in the same row are the same symbols
 - check to see if the other three in the same column are the same symbols
 - and if neither the row nor the column is the same, check to see if it won diagonally, i.e. the diagonal should have the same symbols

```
? | | X
---+---+---
| O | O
---+---+---
X | X | O
```

Who WINS??? Enjoy tic tac toe with arrays!

Source: self – written

1.5 Two Dimension Arrays

A Two Dimension Array has two indexes for inserting into or accessing the array, which is the simplest form of multi – dimension arrays. This type of array can be seen as an array of arrays, i.e. the first index runs for the outer array and the second index runs for the inner arrays.

For Example, consider a two – dimension array, which simply means that we have two dimensions or two variables stored at one place. To declare the array named "arr1" with two dimensions 10 and 5 of the Integer data type, the statement is as follows:

```
int arr1[10][5];
```

This statement shows that arr1 is a 10 element array each of which has 5 elements. 10 runs for the outer array and 5 runs for the inner arrays.

Similarly, to access the elements of a two – dimension array, we need to run two for loops; the outer for loop for the outer array and the inner for loop for the inner arrays. For Example,

```
int i,j;
for(i=0;i<10;i++)          //outer loop
    for(j=0;j<5;j++)        //inner loop
        cout<<arr1[i][j];
```

ADT and Arrays: Chapter 2 Sparse Arrays and Sparse Matrices

The above code runs two for loops. Outer loop runs for 10 elements in the outer array and the inner loop runs for 5 elements in the inner array. Cout statement displays the individual elements of the array named "arr1".

1.5.1 Memory Allocation of Arrays

The amount of memory allocated to an array is based on the base type used to define the array and the number of dimensions, the array has.

For Example, if we have a "frequency" with dimensions as [4][3], the memory allocated will be calculated as follows :

$$\begin{aligned}\text{Memory allocated} &= \text{sizeof(int)} \times \text{size of the array} \\ &= 32 \times 4 \times 3 = 384 \text{ bits}\end{aligned}$$

1.5.2 Matrix Class

A matrix is a rectangular collection of elements stored at one place. The matrix can be denoted by $m \times n$, where m is the number of rows and n is the number of columns that the matrix has. Each element can be categorized by a combination of row and column values. The row value is the index running for the outer array and the column value is the index running for the inner array.

R0, C0	R0, C1	R0, C2
R1, C0	R1, C1	R1, C2
R2, C0	R2, C1	R2, C2

Where R_n – is the Row number, and
 C_n – is the Column number.

For Example, Defining a matrix class named "matrix" of 3×3 , the memory layout would look like this:

matrix[0][0]	matrix[0][1]	matrix[0][2]
matrix[1][0]	matrix[1][1]	matrix[1][2]
matrix[2][0]	matrix[2][1]	matrix[2][2]

The left – hand indexes determine the rows of the matrix and the right – hand indexes determine the columns of the matrix.

The operations that can be performed on a matrix are addition, subtraction, multiplication, transpose, and displaying the matrix. All of these can be easily implemented using two – dimensional arrays.

Algorithm for addition of 2 matrices:

Create two matrices, $\text{mat1} = [a_{ij}]_{m \times n}$ and
 $\text{mat2} = [b_{ij}]_{m \times n}$

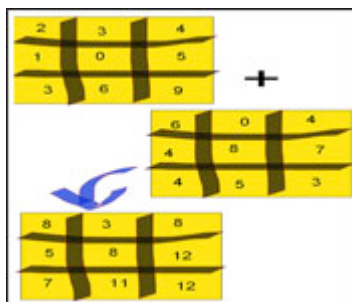
Addition of two matrices mat1 and mat2 will generate a third matrix
 $\text{mat3} = [c_{ij}]_{m \times n}$ such that:

$$\begin{aligned}\text{mat3} &= \text{mat1} + \text{mat2} \\ c_{ij} &= a_{ij} + b_{ij}\end{aligned}$$

Value addition: Pictorial Representation of Algorithm

Heading text Matrix Subtraction

Body text:



Source: self – made

Algorithm for subtraction of 2 matrices:

Create two matrices, $\text{mat1} = [a_{ij}]_{m \times n}$ and

$\text{mat2} = [b_{ij}]_{m \times n}$

Subtraction of two matrices mat1 and mat2 will generate a third matrix

$\text{mat3} = [c_{ij}]_{m \times n}$ such that:

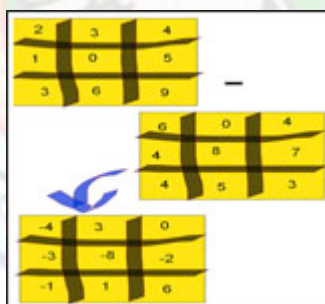
$\text{mat3} = \text{mat1} - \text{mat2}$

$c_{ij} = a_{ij} - b_{ij}$

Value addition: Pictorial Representation of Algorithm

Heading text Matrix Subtraction

Body text:



Source: self – made

Algorithm for multiplication of 2 matrices:

Create two matrices, $\text{mat1} = [a_{ij}]_{m \times n}$ and

$\text{mat2} = [b_{jk}]_{n \times p}$

Addition of two matrices mat1 and mat2 will generate a third matrix

$\text{mat3} = [c_{ik}]_{m \times n}$ such that:

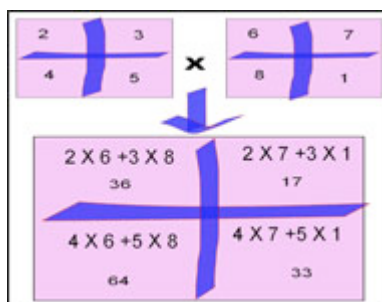
$\text{mat3} = \text{mat1} \times \text{mat2}$

$c_{ik} = \sum a_{ij} b_{jk}$

Value addition: Pictorial Representation of Algorithm

Heading text Matrix Multiplication

Body text:



Source: self – made

Algorithm for transpose of a matrix:

Create a matrix, $\text{mat1} = [a_{ij}]_{m \times n}$

Transpose of the matrix mat1 will generate a new matrix

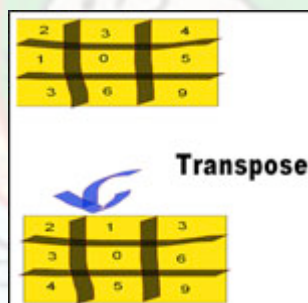
$\text{mat2} = [b_{ij}]_{m \times n}$ such that:

$\text{mat2} = b_{ij} = a_{ji}$

Value addition: Pictorial Representation of Algorithm

Heading text Transpose of a Matrix

Body text:



Source: self – made

2.1 What is a Matrix?

Value addition: Image

Heading text Matrix

Body text:



Computer programmer Thomas A. Anderson leads a secret life as a hacker under the alias "Neo" and wishes to learn the answer to the question "What is the Matrix?" Cryptic messages appearing on his computer monitor and encounters with three sinister Agents lead him to a group led by the mysterious underground hacker Morpheus, a man who offers him the chance to learn the truth about the Matrix. Several members of Morpheus' inner circle, including an infamous female hacker called Trinity, take Neo to a secret meeting, but only after they have exposed and removed a robotic tracking bug implanted in Neo's body by the Agents. During the meeting, Morpheus gives Neo a choice between two pills: red to learn the truth, blue to return to the world as he knows it. Neo accepts by swallowing the offered red pill, and he subsequently finds himself in a liquid-filled pod, his body connected by wires and tubes to a vast mechanical tower covered with identical pods. The connections are severed, and he is rescued by Morpheus and taken aboard his hovercraft, the Nebuchadnezzar. Neo's neglected physical body is restored, and Morpheus explains the situation.

Morpheus informs Neo that the year is not 1999, but estimated to be closer to 2199, and that humanity is fighting a war against intelligent machines created in the early 21st century. The sky is covered by thick black clouds created by the humans in an attempt to cut off the machines' supply of solar power. The machines responded by using human beings as their energy source in conjunction with nuclear fusion, later growing countless people in pods and harvesting their bioelectrical energy and body heat. The world which Neo has inhabited since birth is the Matrix, an illusory simulated reality construct of the world as it was in 1999 developed by the machines to keep the human population docile in their captivity. Morpheus and his crew belong to a group of free humans who "unplug" others from the Matrix and recruit them to their resistance against the machines. Within the Matrix, they are able to use their understanding of its nature to bend the laws of physics within the simulation, giving them superhuman abilities. Morpheus believes that Neo is "the One", a man prophesied to end the war through his limitless control over the Matrix. The Matrix was a co-production of Warner Brothers and Australian Village Roadshow Pictures, and all but a few scenes were filmed at Fox Studios in Sydney, Australia, and in the city itself. Recognizable landmarks were not included in order to maintain the setting of a generic American city.

Source: http://www.filetransit.com/images/screen/53ed891374a1cbfb63e1e7d11471004d_Matrix_Mania.jpg
http://en.wikipedia.org/wiki/The_Matrix



Figure 2.1: Matrices

Source: http://codigobinariomatrices.blogspot.com/2009_04_01_archive.html

A matrix is a rectangular collection of elements stored at one place. The matrix can be denoted by the order $m \times n$, where m is the number of rows and n is the number of columns. Each element can be categorized by a combination of row and column values.

R0, C0	R0, C1	R0, C2
R1, C0	R1, C1	R1, C2
R2, C0	R2, C1	R2, C2

Where R_n – is the Row number, and
 C_n – is the Column number.

Many operations can be performed on Matrices like addition of two matrices, subtraction of two matrices, and so on. These operations are useful in many ways to the user. For Example, John stores marks for different subjects in a matrix format for each year. He can analyse data for various years and determining his performance in his class.

Did you know?

Types of Matrices

Body Text:

There are various types of Matrices:

- **Dense Matrix** – is a matrix which has some values assigned to all of its elements. Even Zero is considered as one of the values of the Matrix.
- **Sparse Matrix** – is a matrix most of whose values are zeroes. The memory is wasted as most of its elements are unused.
- **Diagonal Matrix** – is a matrix whose main diagonal has values and all other elements are zeroes. The main diagonal is where the row index and column index are the same.
- **Tridiagonal** – is a matrix whose main diagonal and its immediate upper and lower matrix has values and all other elements are zeroes.

2.1.1 Application of Matrices

Storing data in a matrix format gives the user a glance at the data on the fly. Also, the data can easily be analyzed for varied purposes. The major advantage of storing data in a matrix is the ease of reading and analysis.

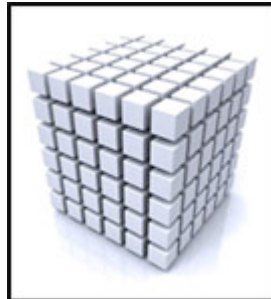


Figure 2.2: Multi – Dimension Arrays

Source: <http://blog.iseesystems.com/tag/arrays/>

For Example, if a homemaker (or a house wife) wants to keep track of the household expenses, and also wants to keep an eye on the expenses to reduce un-necessary expenses, she can easily do so by recording the data in a matrix format, as shown below:

BUDGET MONTHLY HOUSE HOLD EXPENSES			
	JAN	FEB	MAR
ELECTRICITY	1500	1000	1100
WATER	800	700	1000
GROCERY	2000	2100	1900

The above matrix has one dimension as the months and the second dimension as the various household expenses, and a value is inserted at the each juncture of the two dimensions. Water expenses in the month of February were Rs. 700 and Expenses on Grocery in the month of March was Rs. 1900, and so on. The readability of the expenses has improved because of the matrix used.

The homemaker can easily analyse data and find out various facts about the expenses, such as the expenses on Grocery items remain more or less the same, so she should try and reduce the other expenses, i.e. electricity and water, which show major variations in the three months.

Thus, matrices can be used for all the cases where:

- The data can be stored in a two – dimensional format, that is, a maximum of two variables need to be stored.
- Each conjuncture of the two variables should hold a value.

ADT and Arrays: Chapter 2 Sparse Arrays and Sparse Matrices

- The values in the matrix must be easy to read.
- The values in the matrix have to be used for analysis of the data, and certain decisions have to be formulated from them.

Did you know?

Matrix Application

Body Text:

Fibonacci Series

The matrix multiplication can be used to generate the Fibonacci Series, which is an important sequence in Mathematics.

The Fibonacci Series looks like this:

1, 1, 2, 3, 5, 8, 13, 21,...

To get the next number, we add the previous two numbers. The matrix multiplication can be easily used to calculate the nth number in the Fibonacci Series.

Matrix multiplication could be useful to solve some problems. For example, what is the last digit of 1,000,000,000th fibonacci? Calculating fibonacci number using dynamic programming will need $O(n)$ time complexity, we need a faster one to solve this problem.

Let A be a 1×2 matrix which consist of f_0 and f_1 (ie. 0 and 1) and B be a 2×2 matrix which consist of $\{(0,1),(1,1)\}$. If we multiply A and B, we will get a 1×2 matrix C which consist of f_1 and $f_0+f_1 (= f_2)$. If we multiply C with B, then we will get a matrix D which consist of f_2 and $f_1+f_2 (= f_3)$. By doing this, we can obtain f_n by multiplying A with B for $n-1$ times.

$$(f_0 \ f_1) \times \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = (f_1 \ f_2)$$

$$(f_1 \ f_2) \times \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = (f_2 \ f_3)$$

\vdots

$$(f_0 \ f_1) \times \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \times \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \times \dots \times \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = (f_n \ f_{n+1})$$

$$(f_0 \ f_1) \times \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n = (f_n \ f_{n+1})$$

Matrix multiplication is associative, so we can compute f_n by multiplying A with B to the power of $n-1$. We know that an can be computed in $O(\lg n)$ and multiplying two matrixes

needs $O(s^3)$ where s is the size of the matrix, hence the total time complexity would be $O(s^3 \cdot \lg n)$.

Quadratic Forms:

Although matrices are generally for studying linear systems, some non linear things have linear aspects, and can be studied with matrices. A quadratic form is an equation where everything has degree 2: E.g.: $x^2 + 3y^2 + 7xy$

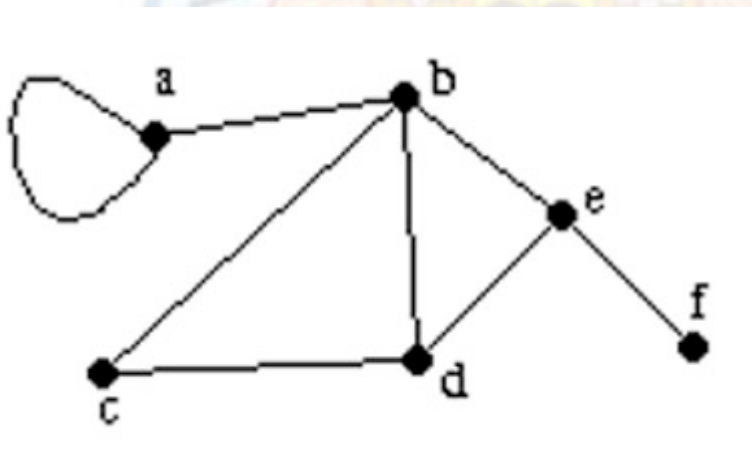
As graphs, we get things like circles and ellipses, and hyperbolas. These things can be described by matrices. To see how, find the following product:

$$\begin{pmatrix} x & y \end{pmatrix} \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

So, we can use methods of linear algebra to look at quadratic forms.

Graphs:

In this context, a graph is a collection of points ("vertices"), joined by lines ("edges"). e.g.:



Graphs have many applications, e.g., what's a good way to connect together lots of computers in a good way. Matrices are a very useful way of studying graphs, since they turn the picture into numbers, and then we can calculate, and use linear algebra techniques.

Games:

Matrices can be used to implement various games such as tic tac toe, easily and conveniently.

Source: <http://www.mast.queensu.ca/~helena/linalg7.html>
<http://www.suhendry.net/blog/?p=902>

2.1.2 Storing Matrices in Arrays

The matrix is a mathematical concept but has immense power and usability. The matrices can be used in all the situations, where the data has to be analyzed for some decision – making process. The decisions once made can further be verified using matrices.

The matrix as a data structure is useful in computer programming because of its decision – aiding properties. The question is how to store a matrix so that we don't lose its properties? The answer is a two – dimension array.

The array is a place in the memory where data of the same type are stored at one place, and the place is known by a name. This enables the user to conveniently access all the values using just one name. This is beneficial for the reason that the user does not have to look at varied places for the values, and the values can be accessed sequentially using the name of the array itself.

A two – dimension array is an array or a collection of values pertaining to two variables at the maximum. The array has two indexes, one for each variable. The name refers to the memory area allocated to the array.

For Example, Defining a matrix class named "matrix" of 3 X 3, the memory layout would look like this:

matrix[0][0]	matrix[0][1]	matrix[0][2]
matrix[1][0]	matrix[1][1]	matrix[1][2]
matrix[2][0]	matrix[2][1]	matrix[2][2]

The above matrix is of the order of 3 X 3, having 3 rows and 3 columns. The rows will depict the first variable and the columns will depict the second variable. When we map this matrix to an array, the above structure is simulated in the memory also.

The left – hand indexes of the array determine the rows of the matrix and the right – hand indexes determine the columns of the matrix. Looping is required to access the values stored in the array. Since we have two indexes, two loops are required.

2.1.3 Special Matrices

There are different types of special matrices depending on the way the data is stored in them. For Example, a diagonal matrix is a special matrix which has values on the main diagonal and all other elements are zero. A matrix is normally stored in a two – dimensional array. Considering the fact that most of the elements in these special matrices are zeroes, a substantial memory area is wasted if two – dimensional arrays are used to store these special matrices. Let us discuss these matrices one by one and the way these matrices will be mapped onto one – dimensional arrays.

Diagonal Matrix

$$A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{pmatrix}$$

$$I_4 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Figure 2.3: Diagonal Matrix

Source:

http://agtr.ilri.cgiar.org/Library/docs/Linear_Mixed_Models/AppendixC.htm

A diagonal matrix M is diagonal if and only $M(i,j) = 0$ for $i \neq j$. That is, the main diagonal of the matrix where i (the row of the matrix) is equal to j (the column of the matrix) has values, and all other elements of the matrix are zero.

For Example,

2	0	0	0
0	5	0	0
0	0	4	0
0	0	0	2

It is an example of a 4 X 4 diagonal matrix.

Number of elements – n for a $n \times n$ matrix

Representation of each element – $arr_d[i-1]$ for $D(i,i)$ in the matrix

Algorithm

- 1: declare a one-dimension array arr_d with n elements for a $n \times n$ matrix
- 2: insert $arr_d[i-1]=x$ if $i=j$; where x is the element to be inserted
- 3: return $arr_d[i-1]$ if $i=j$

Tridiagonal Matrix

A tridiagonal matrix M is diagonal if and only $M(i,j) = 0$ for $|i - j| > 1$. That is, the main diagonal of the matrix where i (the row of the matrix) is equal to j (the column of the matrix) and the diagonal above the main diagonal and below the main diagonal has values; all other elements of the matrix are zero.

For Example,

2	4	0	0
1	5	6	0
0	6	4	8
0	0	7	2

It is an example of a 4 X 4 tridiagonal matrix.

Number of elements – $3n-2$ for a $n \times n$ matrix; these are the elements on all three diagonals

Representation of each element – $arr_t[i-2]$ for element on the lower diagonal
 $arr_t[n+i-2]$ for element on the main diagonal
 $arr_t[2*n+i-2]$ for element on the upper diagonal

Algorithm

- 1: declare a one-dimension array arr_t with $(3n-2)$ elements for a $n \times n$ matrix
- 2: insert arr_d[i-1]=x if $i-j=1$; where x is the element to be inserted
- 3: insert arr_d[n+i-2]=x if $i-j=0$; where x is the element to be inserted
- 4: insert arr_d[2*n+i-2]=x if $i-j=-1$; where x is the element to be inserted
- 5: return arr_d[i-1] if $i-j=1$
- 6: return arr_d[n+i-2] if $i-j=0$
- 7: return arr_d[2*n+i-2] if $i-j=-1$

Lower Triangular Matrix

A lower triangular matrix M is lower triangular if and only if $M(i,j) = 0$ for $i < j$. That is, the area below the main diagonal has values and all other elements are initialized to 0. If you notice the row and column dimensions of this area, each element has i less than j; the row dimension is always less than the row dimension for this section of the matrix.

For Example,

2	0	0	0
4	5	0	0
3	3	4	0
1	5	2	2

It is an example of a 4 X 4 lower triangular matrix.

2.2 What is a Sparse Matrix?

Value addition: Frequently Asked Questions

Heading text Sparse Matrix

Body text:

8	0	0	0	0
0	0	4	0	5
0	3	0	0	0
0	0	0	6	0
0	1	0	0	0

The elements of a sparse matrix are majorly zeroes. The question is how many zeroes are too many for a matrix to be considered as a sparse matrix? Can zero be an element of the sparse matrix? The number of zeroes to be considered as too many for a sparse matrix depends on the size of the matrix. The bigger the matrix, the higher this number.

Zero, if is part of the sparse matrix, then it would be difficult to decided whether to store it as an element, or, would it form part of the zero elements of the sparse matrix.

Source: self – written

ADT and Arrays: Chapter 2 Sparse Arrays and Sparse Matrices

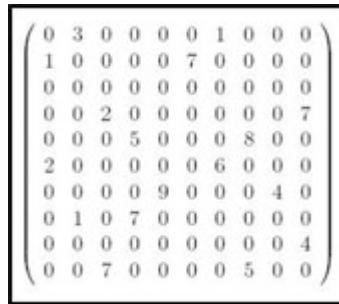


Figure 2.4: Sparse Matrix

Source: http://nucleo.ces.clemson.edu/home/online_tools/sparse_matrix/0.1/html/about_this_tool/background.shp.cgi

A Sparse Matrix is a matrix which is populated majorly by zeroes, in other words, it has less of non – zero values.

For Example, if we store the student names and the course he has opted for in a matrix format, assuming each student is allowed to take only one course at a time, the matrix would look like the following:

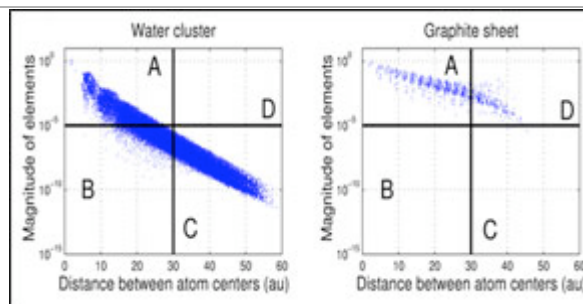
APPROXIMATION STUDENT Vs COURSES					
	JOHN	SHREYA	MARY	KAJAL	MANSI
B.COM(HONS)	1	0	0	0	0
B.SC.(HONS)	0	0	0	1	1
B.A.(HONS)	0	1	0	0	0
B.C.A	0	0	1	0	0

The above matrix has recorded the courses each student has opted for. If a student has opted for a particular course, then the juncture of that student and the particular course has the value 1, otherwise 0.

We had an assumption that each student can opt for only one course at a time, thus, the matrix has most of its elements as zeroes. In other words, zero elements are much more than the non – zero elements. Such a matrix is called a Sparse Matrix.

Interesting fact

Finite Element Sparse Matrix



A sparse matrix is a matrix with enough zero entries that it is worth using an algorithm that avoids storing or operating on the zero entries.

J. Demmel Applied Numerical Linear Algebra (1997)

In all reports above the common compressed sparse row (CSR) representation was used as underlying data structure for the blocked sparse representation. However, the optimal block size for the sub matrices is usually large enough for the overhead in the data structure to become of secondary importance. Therefore it is possible to use a simpler underlying data structure than the CSR to make the implementation of matrix manipulations significantly faster, easier and more maintainable, without sacrificing performance.

In the hierarchic data structure we use a simple column-wise representation for the storage of the sub matrices. This data structure can also be used to obtain several levels in the hierarchic representation. This is achieved by the use of a generic programming approach in the C++ programming language where the type of the matrix elements is left open until compile time. This means that a matrix can contain matrices instead of real numbers and an arbitrary number of levels in the hierarchy can be obtained. In the figure, for example, a three level hierarchy is illustrated with a matrix which contains matrices which contains matrices which contains real numbers.

The point with the introduction of several levels is to reduce the overhead since the use of a single level would result in a cubically scaling overhead for matrix multiplication. The gain comes from that zero matrices at higher levels do not need to be referenced in algorithms since their value already is known.

Source: <http://www.theochem.kth.se/research/multiscale/sparse.html>

2.2.1 Storing Sparse Matrix in Arrays

Value addition: Historical Fact

Heading text Sparse Matrix

Body text:

$$A = \begin{bmatrix} 1 & 0 & 6 & 7 & * & * \\ 2 & 1 & 8 & 2 & * & * \\ 6 & 8 & 1 & 4 & * & * \\ 7 & 2 & 5 & 2 & * & * \\ * & * & * & * & 7 & 2 \\ * & * & * & * & 0 & 0 \end{bmatrix}$$

Traditionally a complete data structure (full matrices) has been used for the matrices appearing in Hartree-Fock and DFT implementations. However, by using this approach linear scaling can never be achieved since the storage grows quadratically with system size.

Challacombe presented a new data structure where the matrices are divided into atom blocks and sparsity is considered on the block level (Comp. Phys. Comm. 128, 93 (2000), J. Chem. Phys. 110, 2332 (1999)). This requires that all basis functions on a single atom are grouped together into a single block. In this way, the access overhead is reduced since one can address entire blocks rather than single elements. Another advantage is the ability to use the highly optimized Basic Linear Algebra Subprograms

ADT and Arrays: Chapter 2 Sparse Arrays and Sparse Matrices

(BLAS) for basic sub matrix manipulations rather than to operate on single matrix elements. Later on, it was observed that the sub matrices coming from single-atom blocks would be too small to reach peak performance of BLAS routines. Therefore a multi-atom blocked matrix data structure was proposed and the performance increased (J. Comp. Chem. 24, 618 (2003)). It should be noted that this method exposes a trade-off. On one hand, one wants to have as large blocks as possible to reduce the overhead, on the other -- since the screening is done on a per-block basis -- large blocks will store more redundant zero elements.

Source: <http://www.theochem.kth.se/research/multiscale/sparse.html>

A matrix is normally stored in a two - dimension array in computer programming environment. Each element of a two - dimension array stores the corresponding element of the matrix. Consider storing a sparse matrix in a two - dimension array, which essentially means that most of the array elements would be zero. Though zero is an element of the matrix, such a value in the array element would simply waste memory storage space.

For Example, storing the above example on the approximation of student vs. course in a two - dimension array would produce an array named "approx" with 4 rows and 5 columns.

The declaration of the "approx" array would be:

```
int approx[4][5];
```

The memory allocated to approx, assuming integer takes 4 bytes of memory, would be:

Memory allocated = $32 \times 4 \times 5 = 640$ bits

The "approx" array has 640 bits of memory allocated to it and that too in the form of consolidated chunk. The memory used effectively out of 640 bits is just 160 bits, leading to a waste of 480 bits, which is 60 bytes. This number might not be enticing to you for consideration in the above example, however, if the order of the matrix is huge, the memory wasted is all the higher.

Interesting fact

Complex Numbers and Matrices

Body text:

If you know how to multiply 2x2 matrices, and know about complex numbers, then you'll enjoy this connection. Any complex number ($a+bi$) can be represented by a real 2x2 matrix in the following way! Let the 2x2 matrix

$$\begin{bmatrix} a & b \\ -b & a \end{bmatrix}$$

correspond to ($a+bi$). Addition of complex numbers then corresponds to addition of the corresponding 2x2 matrices. So does multiplication!

Observe:

$$\begin{bmatrix} a & b \\ -b & a \end{bmatrix} \begin{bmatrix} c & d \\ -d & c \end{bmatrix} = \begin{bmatrix} (ac-bd) & (ad+bc) \\ -(ad+bc) & (ac-bd) \end{bmatrix}$$

which is precisely what you would get if you multiplied $(a+bi)$ and $(c+di)$ and then converted to a 2×2 matrix!

The Math Behind the Fact: The reason this works is because complex multiplication can be viewed as a linear transformation on the 2-dimensional plane.

Source:

Body text:

If you know how to multiply 2×2 matrices, and know about complex numbers, then you'll enjoy this connection. Any complex number $(a+bi)$ can be represented by a real 2×2 matrix in the following way! Let the 2×2 matrix

$$\begin{bmatrix} a & b \\ -b & a \end{bmatrix}$$

correspond to $(a+bi)$. Addition of complex numbers then corresponds to addition of the corresponding 2×2 matrices. So does multiplication!

Observe:

$$\begin{bmatrix} a & b \\ -b & a \end{bmatrix} \begin{bmatrix} c & d \\ -d & c \end{bmatrix} = \begin{bmatrix} (ac-bd) & (ad+bc) \\ -(ad+bc) & (ac-bd) \end{bmatrix}$$

which is precisely what you would get if you multiplied $(a+bi)$ and $(c+di)$ and then converted to a 2×2 matrix!

The Math Behind the Fact: The reason this works is because complex multiplication can be viewed as a linear transformation on the 2-dimensional plane.

Source: <http://www.math.hmc.edu/funfacts/ffiles/30001.1.shtml>

2.3 Single Dimension Arrays and Sparse Matrices

A one – dimensional or Single Dimension Array has just one index for inserting into or accessing the array. This array stores values pertaining to just one variable.

A sparse matrix stored in a two – dimension array cannot be justified as it wastes a lot of memory. The matrix can be mapped onto a one – dimension array, thus, saving a lot of space and serving the same purpose.

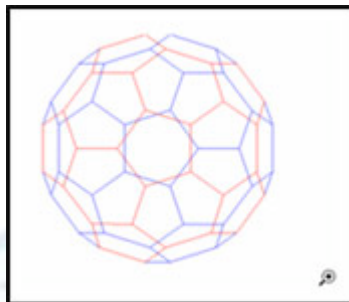
The zero elements have ideally no value if we think in terms of storage in some data structure. There is no need to store the zero values of the sparse matrix. Only the non – zero elements of the sparse matrices need be stored.

Did you know?

Matrix Applications

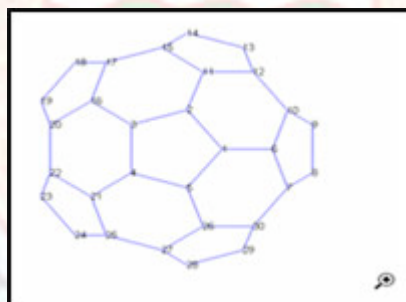
Body Text:

A graph is a set of nodes with specified connections between them. An example is the connectivity graph of the Buckminster Fuller geodesic dome (also a soccer ball or a carbon-60 molecule).



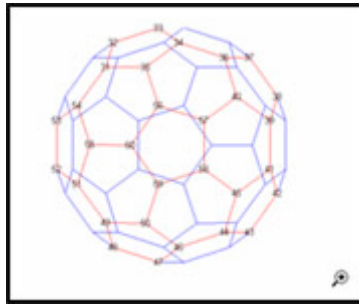
In MATLAB®, the graph of the geodesic dome can be generated with the BUCKY function.

A graph can be represented by its adjacency matrix. To construct the adjacency matrix, the nodes are numbered 1 to N. Then element (i,j) of the matrix is set to 1 if node i is connected to node j, and 0 otherwise.



To visualize the adjacency matrix of this hemisphere, we use the SPY function to plot the silhouette of the nonzero elements.

Note that the matrix is symmetric, since if node i is connected to node j, then node j is connected to node i.



Finally, here is a SPY plot of the final sparse matrix. In many useful graphs, each node is connected to only a few other nodes. As a result, the adjacency matrices contain just a few nonzero entries per row.

This demo has shown one place where SPARSE matrices come in handy

Source: <http://www.mathworks.co.kr/products/matlab/demos.html?file=/products/demos/shipping/matlab/buckydem.html>

2.3.1 Mapping of Non-Zero Elements

When only the non – zero elements of the sparse matrix are stored, we have a considerable saving of the wasted memory space. Considering the above example, we use only 160 bits of memory in storing the sparse matrix in a one – dimension setup as compared to 640 bits in a two – dimension environment.

Matrix works in a two – dimension environment, squeezing it into a one – dimension environment requires compromising on the values that need to be stored. Thus, the Sparse Matrix has to be **mapped** onto the one – dimension array.

The Sparse Matrix can be mapped onto the one – dimension array in the following ways:

- **Row – Major Order:** the non – zero elements are read from one row to another; storing them in the array in this manner is in the row – major order.
- **Column – Major Order:** the non – zero elements are read from one column to another; storing them the array in this manner is in the column – major order.

When the non – zero elements are stored only, the other elements of the matrix are assumed to be zero, thereby saving lots of memory space. Processing a sparse matrix in this manner not only saves memory, but also saves time required to perform various operations on the elements of a sparse matrix.

2.4 Row – Major Mapping

The sparse matrix is mapped in the Row – Major Order when the non – zero elements of the matrix are read row – by – row and stored into the one – dimension array. This is to say that this mapping gives importance to the rows

ADT and Arrays: Chapter 2 Sparse Arrays and Sparse Matrices

of the matrix while storing the elements.

Each element in the sparse matrix is denoted in the following manner:

Sparse Matrix $S = [s_{ij}]_{m \times n}$

where s_{ij} is the element placed at row i and column j , which can either be zero or non-zero. If $s_{ij} \neq 0$, this element will be stored. Otherwise, it is discarded.

2.4.1 Array Representation using Row - Major

Consider the following 3 X 4 sparse matrix:

0	0	1	0
0	0	0	3
3	2	0	0

To map this matrix in row - major order, the non-zero elements only will be recorded, which makes it thus:

1	3	3	2
---	---	---	---

When we record these values, we need to store the row and column of each of these elements. This is needed so as to reconstruct a two - dimension matrix from the one - dimension array.

Thus, the values which need to be stored are:

Row	Column	Value
1	3	1
2	4	3
3	1	3
3	2	2

The data structure to be used for storing non-zero elements should be such that can store the above - mentioned three values at one place.

Data Structure:

```
int row;  
int column;  
int data;
```

where, row is the row of element of the sparse matrix, column is the column of element of the sparse matrix and data is the value of element of the sparse matrix which needs to be stored.

Each element of the one - dimension array should be of the form of this data structure, so that all the information required to reconstruct the sparse matrix from this array is stored at one place.

To map the sparse matrix onto the one - dimension array, the user creates an array of the size of the number of non-zero elements that need to be entered. Each element of the array is assigned the triplet values.

ADT and Arrays: Chapter 2 Sparse Arrays and Sparse Matrices

For Example, to store the above matrix, the array should be declared with 4 elements, each element being of the form of the data structure given above. The array is named "sparse", so the memory allocation should look like:

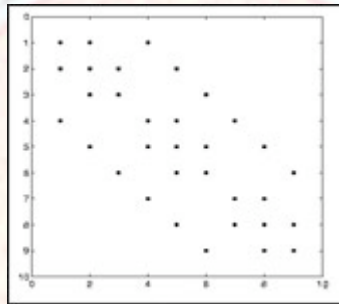
sparse[0]	sparse[1]	sparse[2]	sparse[3]
1	2	3	3
3	4	1	2
1	3	3	2

The above table shows the array representation of the sparse matrix storing only the non – zero elements.

Value addition: Image

Heading text Creating a Sparse Matrix

Body text:



Source: <http://www.math.mtu.edu/~msgocken/intro/img30.gif>

2.5 Column – Major Mapping

The sparse matrix is mapped in the Column – Major Order when the non – zero elements of the matrix are read column – by – column and stored into the one – dimension array. This is to say that this mapping gives importance to the columns of the matrix while storing the elements.

Each element in the sparse matrix, s_{ij} is the element placed at row i and column j , which can either be zero or non - zero. If $s_{ij} \neq 0$, this element will be stored. Otherwise, it is discarded.

2.5.1 Array Representation using Column - Major

Consider the following 3 X 4 sparse matrix:

0	0	1	0
0	0	0	3
3	2	0	0

ADT and Arrays: Chapter 2 Sparse Arrays and Sparse Matrices

To map this matrix in column – major order, the non – zero elements only will be recorded, which makes it thus:

3 2 1 3

When we record these values, we need to store the row and column of each of these elements. This is needed so as to reconstruct a two – dimension matrix from the one – dimension array.

Thus, the values which need to be stored are:

Column	Row	Value
1	3	3
2	3	2
3	1	1
4	2	3

The data structure to be used for storing non – zero elements should be such that can store the above – mentioned three values at one place.

Data Structure:

```
int column;  
int row;  
int data;
```

where, row is the row of element of the sparse matrix, column is the column of element of the sparse matrix and data is the value of element of the sparse matrix which needs to be stored.

Each element of the one – dimension array should be of the form of this data structure, so that all the information required to reconstruct the sparse matrix from this array is stored at one place.

To map the sparse matrix onto the one – dimension array, the user creates an array of the size of the number of non – zero elements that need to be entered. Each element of the array is assigned the triplet values.

For Example, to store the above matrix, the array should be declared with 4 elements, each element being of the form of the data structure given above. The array is named "sparse", so the memory allocation should look like:

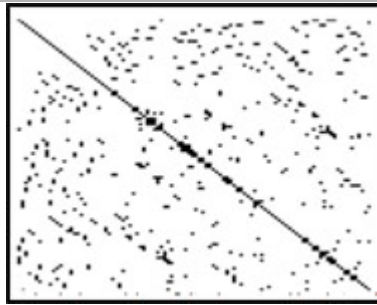
sparse[0] sparse[1] sparse[2] sparse[3]

1	2	3	4
3	3	1	2
3	2	1	3

The above table shows the array representation of the sparse matrix storing only the non – zero elements.

Interesting fact

Finite Element Sparse Matrix



A sparse matrix obtained when solving a finite element problem in two dimensions. The non-zero elements are shown in black.

The finite element method (FEM) (its practical application often known as finite element analysis (FEA)) is a numerical technique for finding approximate solutions of partial differential equations (PDE) as well as of integral equations. The solution approach is based either on eliminating the differential equation completely (steady state problems), or rendering the PDE into an approximating system of ordinary differential equations, which are then numerically integrated using standard techniques such as Euler's method, etc.

Source: http://en.wikipedia.org/wiki/Finite_element_method

2.5.2 Adding Two Matrices

The addition of two dense matrices is simple to perform. Consider two matrices of the form:

$$\begin{aligned} \text{Matrix A} &= [A_{ij}]_{m \times n} \\ \text{Matrix B} &= [B_{ij}]_{m \times n} \end{aligned}$$

The sum of matrices A and B would be a

$$\text{Matrix C} = [C_{ij}]_{m \times n}$$

Such that each C_{ij} is the sum of A_{ij} and B_{ij} . The essential assumption is that both the matrices should have the same order $m \times n$, otherwise the matrices are not compatible with each other and hence, cannot be added.

The same addition operation can be applied to two sparse matrices also. However, the resultant matrix will also be a sparse matrix with majorly zero elements. This matrix in a two – dimension array will waste a lot of memory space and will be time consuming.

The solution is to use row – major mapped matrices A and B instead of two – dimension arrays. The algorithm given below uses this solution to perform addition on two sparse matrices.

Algorithm for Sparse Matrix Addition:

Input: Matrix A with row and column indices ROW_A and COL_A, respectively.
Matrix B with row and column indices ROW_B and COL_B, respectively.

ADT and Arrays: Chapter 2 Sparse Arrays and Sparse Matrices

Output: Matrix C with row and column indices ROW_C and COL_C, respectively.

Assumptions:

- Matrix C must be represented in the same manner as Matrices A and B.
- Matrices A and B must have the same dimensions $m \times n$ and contain non – zero elements which are mapped in row – major order.
- Ctr_A is a counter variable to be used for looping in Matrix A and ctr_B is the same variable for Matrix B
- Number_of_terms is a variable holding the actual number of non – zero terms in each matrix A and B
- Index_A and index_B are variables holding the intermediate values

Step 1: initialize ctr_A = ctr_B = 1, number_of_terms_A = number_of_terms_B = N

Step 2: if ROW_A \neq ROW_B and COL_A \neq COL_B, the matrices are incompatible for addition.

Step 2: Repeat step 2 through step 3 until ctr_A < number_of_terms_A and ctr_B < number_of_terms_B

Step 3: index_A = ROW_A * n + COL_A
index_B = ROW_B * n + COL_B

if index_A < index_B then ROW_C = ctr_A.ROW_A,
COL_C = ctr_A.COL_A,
C.data = ctr_A.data
ctr_A = ctr_A + 1

Else if index_A > index_B then ROW_C = ctr_B.ROW_B,
COL_C = ctr_B.COL_B,
C.data = ctr_B.data
ctr_B = ctr_B + 1

else if index_A == index_B then ROW_C = ctr_B.ROW_B,
COL_C = ctr_B.COL_B,
C.data = ctr_A.data +
ctr_B.data
ctr_A = ctr_A + 1
ctr_B = ctr_B + 1

Step 4: repeat until ctr_A < number_of_terms_A
ROW_C = ctr_A.ROW_A,
COL_C = ctr_A.COL_A,
C.data = ctr_A.data
ctr_A = ctr_A + 1

Step 5: repeat until ctr_B < number_of_terms_B
ROW_C = ctr_B.ROW_B,
COL_C = ctr_B.COL_B,

```
C.data = ctr_B.data
ctr_B = ctr_B + 1
```

2.6 Linked Representation of Sparse Matrices

A drawback of using row – major or column – major mapping onto one – dimension array is that the number of non – zero elements should be known in advance. This number is however, easy to know because the arrays are used as input, but the number of elements resulting from various operations such as addition of two sparse matrices may vary from the actual input. This number cannot be accurately calculated in advance while declaring the array for the storage of the results.

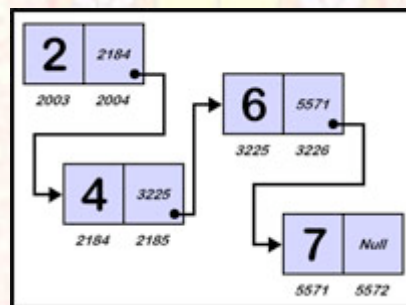
This problem can be overcome by using linked representation of the sparse matrices.

2.6.1 Linked Lists

Did you know?

Matrix Applications

Body Text:



In the above image, the first node stores the value 2 and the address pointer points to the next node. This section of the first node will store the address of the second node, which is 2184. This way all the nodes in the linked list are attached to each other. We just need to store the address pointer of the first node, and we can traverse till the end – of – file, which is NULL.

Notice that the last memory cell in our chain contains a symbol called "Null". This symbol is a special value that tells us we have reached the end of our list. You can think of this symbol as a pointer that points to nothing. Since we are using pointers to implement our list, the list operations AddListItem and RemoveListItem will work differently than they did for sequential lists. The animation below shows how these operations work and how they provide a solution for the two problems we had with arrays.

Source: <http://courses.cs.vt.edu/csonline/DataStructures/Lessons/OrderedListImplementationView/Lesson.html>

The arrays are static storage structures, i.e. the number of elements in the array should be known while declaring the array. If the arrays are allocated dynamically, then also the memory allocated to the array is dependent on the elements in the array. The size of the array in any case should be known in advance. Thus, the arrays are useless in the cases where:

ADT and Arrays: Chapter 2 Sparse Arrays and Sparse Matrices

- The size of the array is not known in advance
- The memory equivalent to the size and data type of the array is not available as a consolidated chunk

Linked Lists come handy in all these situations. A linked list is based on the concept of pointers. A pointer is a special type of variable which holds the address of another variable. The pointer holds the type of value of the base type using which it has been declared. A linked list is a list of forward, or backward, or both kinds of pointers. Each node in the list is pointing to the next node, and this node is pointing to the next, and so on.

The list stores the starting address of the list and the values can be accessed by traversing the list from the first node until the end of the list is not reached.

2.6.2 Linked Lists and Sparse Matrices

The sparse matrices when stored using arrays suffer from major drawbacks. This shortcoming can be overcome by using linked lists to store the matrices. The linked representation will also be either row – major or column – major order.

Row – Major order

In this type of mapping, more emphasis is given on the rows. The list has to be divided into two types of nodes:

- One which will store the rows
- One which will store the columns and the values

The first type of node will store the row number and a pointer to next such node. This node will also have a pointer to the list of second type of nodes.

The second type of node will store the column number and the value as a pair and will also have a link to the next such node.

2.6.3 Linked Lists Vs Arrays

Sparse Matrices can be stored using two data structures – linked lists and arrays.

For array as a storage medium, the number of non – zero elements need to be known in advance, which is not always possible. Also, the memory to be allocated to the array should be available in a consolidated chunk, which is some times not possible. If either of the above – mentioned situation is prevalent, the sparse matrices cannot be created and worked upon. No operation can be performed on these.

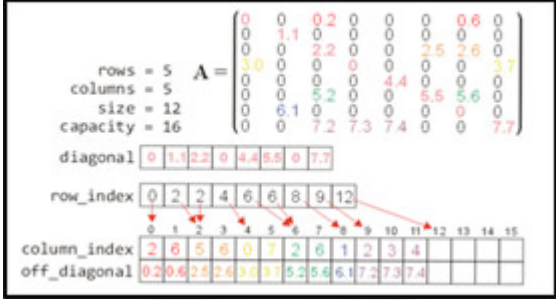
Using linked lists as a storage medium, the above – mentioned shortcomings are done away with. The number of non – zero elements can vary and based on the requirements new nodes in the linked list may be created. Since, the nodes are created on need – basis and links are provided between the previous and the new node, the memory need not be consolidated. Only that much memory is needed at a time as is required by each kind of node to be created.

2.7 Sparse Arrays

Value addition: Image

Heading text Sparse Arrays

Body text:



The diagram illustrates the storage of a sparse matrix A . The matrix A is 5x5 with 12 non-zero entries. The diagonal entries are stored in a separate array 'diagonal'. The off-diagonal entries are stored in a 2D array 'off_diagonal'. The row indices are stored in 'row_index' and the column indices in 'column_index'. The diagram shows the mapping of non-zero entries from the matrix to the storage arrays.

The diagonal entries are stored in a separate array to allow the fastest possible access. Most sparse matrices of interest to engineers do not have zero entries on the diagonal (and are often positive definite). Hence, the diagonal is very likely to be dense. You will note that rows 1 and 4 have no off-diagonal entries and are therefore $\text{row_index}[2] == \text{row_index}[1]$ and $\text{row_index}[5] == \text{row_index}[4]$. You will also note that the last entry of row_index always contains the number of off-diagonal entries, that is what is commonly referred to in the C++ STL as the *size*.

As one example of the simplifications used, a linear search is made of all the off-diagonal entries in a particular row. A binary search could be used, but this would obfuscate the code.

In the run-time analyzes, we will assume that the matrix is $n \times n$, the number of non-zero off-diagonal entries are N , and the number of non-zero entries per row is approximately N/n .

Accessing entries in vectors and matrices uses C-style indexing for $i = 0, \dots, M - 1$; however, in some cases, it may be useful to access the last entry, row, or column without reference to the dimensions. Therefore, indices on the range $i = -M, \dots, -1$ will access the entries $M + i$.

Source: http://www.ece.uwaterloo.ca/~dwharder/Algorithms_and_Data_Structures/Algorithms/Sparse_systems/

Sparse Arrays are defined to be arrays most of whose values are default values, mostly NULL or Zero. This value is a constant value which is known in advance. The array can be initialized to this default value while declaration and populated with non – default values, which are few in number, later on. In many fields of Mathematics and Engineering, Sparse Arrays are quite common, so it is essential to find efficient means of handling and representing these arrays. The occurrence of zero elements in a large array is both a computational and storage inconvenience.

For Example, there are different types of goods placed on the shelves of a super market, and each day the super market sees different kinds of customers with varied tastes and preferences. The owner of the super market thought of analyzing the data of purchase and sale of commodities. He used a spreadsheet to input data relating to the market.

ADT and Arrays: Chapter 2 Sparse Arrays and Sparse Matrices

The rows hold the customers data and the columns hold the commodities data. He analysed buying trends for 5 products with respect to 10 customers. He found out that most of the entries in the spreadsheet were 0 and very few cells had some value, which were the quantity the customer purchased.

If the data discussed above were to be stored in a two – dimension array, it would be called as a sparse array as most of its elements are default values, i.e. zeroes. The structure so used will lead to wastage of memory space because most of the memory locations are not used at all.

Customers	Commodities					
		Salt	Diapers	Shaving Foam	Vanity Case	Watch
	John	2	0	2	0	1
	Shreya	0	0	0	0	1
	Rohit	3	0	1	0	0
	Dimpy	1	0	0	0	0
	Mary	1	1	0	0	0
	Sumeet	0	0	1	0	0
	Mansi	1	0	0	1	1
	Shruti	0	1	0	0	0
	Riya	0	0	1	0	1
	Sally	2	0	0	0	0

Did you know?

Applications of Sparse Arrays

Body Text: Text

- A new application of the time-reversal approach is proposed, i.e., the synthesis of a specific wavefront (plane and short wavefront) in the near-field domain of a transducer array. The theoretical background for computing the temporal excitation signals that must be emitted on the different channels of the array is presented. The efficiency of this approach is illustrated with two kinds of transducer arrays: a full and periodic array containing 1024 elements and a sparse array made of 128 elements. These results are obtained through numerical software; they prove that it is theoretically possible to create any desired wavefront shape if the array contains enough independent elements. Experimental results obtained with a real sparse array are presented.
- The goals of block copolymer lithography for patterning structures for the semiconductor industry include generating patterns with the desired dimensions and geometries, defect-free assembly over wafer-scale areas, and precise control over the positions of the assembled structures.

This uses the sparse arrays of posts for performing graphoeptaxial assembly of spherical block copolymer structures over large areas. The posts have carefully controlled sizes and interfacial interactions such that they replace the spheres in certain locations of the array and template the assembly of spherical domains between posts. Sphere positioning and long-range ordering are demonstrated, as are two-dimensional density enhancements up to 26× the patterned post density. Control over the orientation of the assembled arrays with respect to the topographic post array is provided by the ratio of the period of the posts to the period of the block copolymer spheres. Arrays of

topographic posts with large period ratios led to the simultaneous assembly of grains with different orientations, simply because the free energy difference between such orientations was small.

- Methods and apparatuses are disclosed for the exposure of sparse hole and/or mesa arrays with line: space ratios of 1:3 or greater and sub-micrometer hole and/or mesa diameters in a layer of photosensitive material atop a layered material. Methods disclosed include: double exposure interferometric lithography pairs in which only those areas near the overlapping maxima of each single-period exposure pair receive a clearing exposure dose; double interferometric lithography exposure pairs with additional processing steps to transfer the array from a first single-period interferometric lithography exposure pair into an intermediate mask layer and a second single-period interferometric lithography exposure to further select a subset of the first array of holes.

We have developed a method for designing sparse periodic arrays. Grating lobes in the two-way radiation pattern are avoided by using different element spacings on transmission and reception. The transmit and receive aperture functions are selected such that the convolution of the aperture functions produces a desired effective aperture. A desired effective aperture is simply an aperture with an appropriate width, element spacing, and shape such that the Fourier transform of this function gives the desired two-way radiation pattern. If a synthetic aperture approach is used, an exact solution to the problem is possible. However, for conventional imaging, often only an approximation of the desired effective aperture can be found. Different strategies for obtaining these approximate solutions are described. The radiation pattern of a sparse array designed using the effective aperture concept is compared experimentally with the radiation patterns of a dense array, and sparse arrays with periodic and random element spacing. We show that the number of elements in a 128-element linear array can be reduced by at least four times with little degradation of the beam forming properties of the array.

Source: <http://community.acs.org/nanotation/tabid/98/Default.aspx?ReviewId=85>
<http://ip.com/patent/US5759744> <http://cat.inist.fr/?aModele=afficheN&cpsidt=2964499>

2.7.1 Sparse Arrays using One – Dimension Arrays

Sparse Arrays can be stored in a one – dimension array. For Example, a sparse array named "sparse_arr" is created with the following values:

arr[0]	arr[1]	arr[2]	arr[3]	arr[4]	arr[5]	arr[6]	arr[7]
0	0	10	6	0	0	8	0

Figure 2.5: Sparse Matrix using arrays

Source: self

The sparse_arr will be allocated 8 units of memory as a consolidated chunk, out of which only 3 units of memory are effectively used. The rest of the memory is wasted. To avoid the above wastage, the array can be represented using linked lists, which will save considerable amount of memory and the processing of such an array will also be fast.

Sparse_arr using linked list:

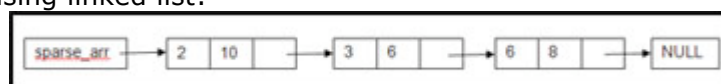


Figure 2.6: Sparse Matrix using Linked List

Source: self

2.7.1 Sparse Arrays using One – Dimension Arrays

The linked list `sparse_arr` stores a pointer to the first node in the list, which points to next node, and so on, until the end of the list is reached.

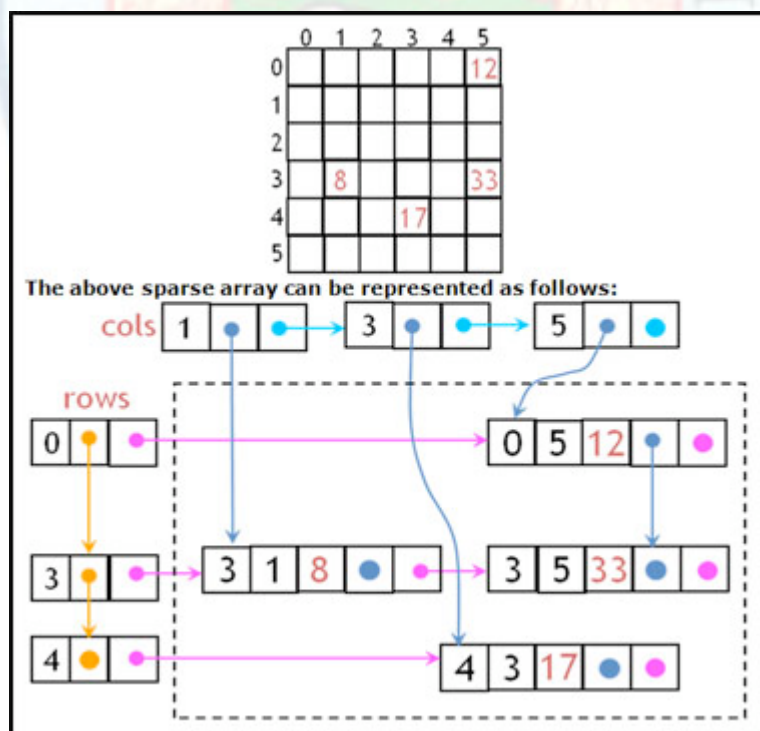
Each node in the linked list is of the form:

```
class node
{
    int index;
    int value;
    node *next;
};
```

Node is the name given to the class which defines each node for the above linked list representation of the sparse arrays. The node class contains 3 variables as follows:

- Index – is the variable of the type of Integer and stores the index of the non – default element in the array.
- Value – is the variable of the data type of the elements to be stored in the array. It is the actual value of the non – default element of the sparse array.
- Next – is the pointer variable of the type of the node class itself. This is so because it has to point to the next node which is also of this data type, i.e. node class.

Body text:



Source: <http://www.cis.upenn.edu/~matuszek/cit594-2007/Lectures/40-sparse-arrays.ppt+40-sparse-arrays.ppt&cd=1&hl=en&ct=clnk&gl=in>

2.7.2 Operations on Sparse Arrays

The sparse arrays can have the following operations:

- Create_arr – is creation of the sparse arrays of a certain length which is pre – defined.
- Get_value – is a module to get values from the array at a particular index.
- Put_value – is a module to store values at a particular index. This module also accepts the value to be inserted as the input.
- Length – is a module which will return the size of the array.
- Count_value – is a module which will count the number of non – default values.
- Search_max – is a module which will search the sparse array to determine the maximum value in the array.

The fundamental operation in most of these operations would be to traverse through the array using a looping construct. This operation can be time – consuming if the loop is allowed to move through all the elements. This is so because most of the elements are zeroes and doing this, we will waste a lot of our time and effort.

The operation of searching the sparse array should not search all the elements. Rather, checks should be performed only on the values which are non – default values.

For Example, Algorithm for search_max:

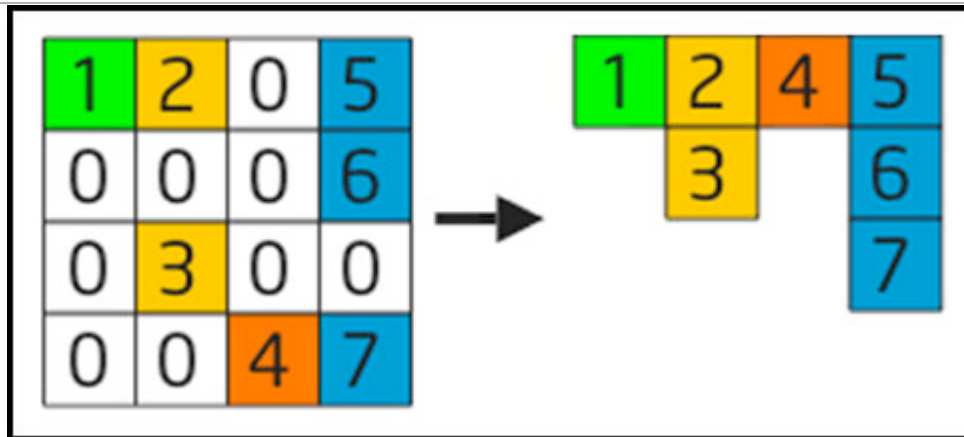
Step 1: initialize max = value in first index of the array

Step 2: create a sparse array

Step 3: repeat step 3 through step 4 until the end of array is reached

Step 4: for each index in the array
 If value at this index is not zero
 Compare this value with max
 If this value is greater than max
 Set max = this value

Step 5: end

interesting fact**Creating a Sparse Array**

Using this type of representation is important because of the enormous memory footprint and calculation savings. For example, in a simulated scene where you have 1000 rigid object, you can represent constraints or relations between the objects as a 1,000,000 (1000x1000) element adjacency matrix. However, most of the object may have no relationship to each other and may not even touch, meaning that the number of populated entries in this matrix is quite small (perhaps 3 per object, meaning a sparse representation would require 3000 entries versus 1,000,000 entries!) The problem with sparse matrices is that there is no single standard representation and each form comes with its own parallelization challenges. Compressing a dense matrix into sparse form makes the computation "irregular". What do I mean by this? We usually refer to simpler processing patterns as regular, such as simply walking over the element of an array or image in sequence. So, by irregular in this instance, I mean that the elements we touch might jump around unpredictably. Also, we might find these "touches" also modify data, meaning that we have to coordinate these accesses when implementing them in parallel.

Source: http://blogs.intel.com/research/2007/09/the_many_flavors_of_data_parallel.php

2.7.3 Sparse Arrays using Two – Dimension Arrays

Sparse Arrays can be stored using a two – dimension array. However, the memory wastage will be even higher than one – dimension arrays.

For Example, a sparse array named "sparse_arr" is created with the following values:

sparse_arr[0]	sparse_arr[1]	sparse_arr[2]	sparse_arr[3]	sparse_arr[4]	sparse_arr[5]	sparse_arr[6]	sparse_arr[7]
0	0	10	6	0	0	0	0
0	0	0	0	0	0	8	0
1	0	0	0	0	2	0	0

Figure 2.6: Sparse Matrix using Two – dimension Arrays
Source: self

ADT and Arrays: Chapter 2 Sparse Arrays and Sparse Matrices

The two – dimension array memory arrangement is shown above and if we calculate the memory allocation, the memory is higher than the one – dimension array. Thus, using two – dimension arrays to store sparse arrays is not justified. This arrangement can be easily shown using linked list structure.

Sparse_arr using linked list:

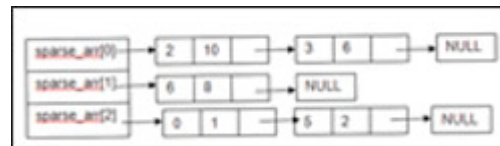


Figure 2.7: Sparse Matrix using Linked List

Source: self

The linked list sparse_arr stores a pointer to the first node in the list, which points to next node, and so on, until the end of the list is reached.

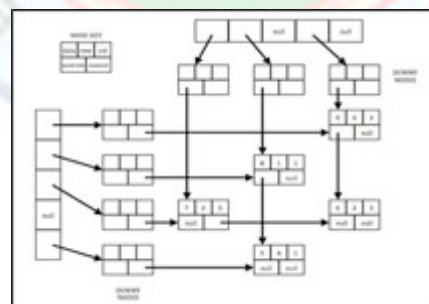
The linked representation is more or less similar to the linked representation of the one – dimension array with the exception that the first node is essentially an array of the type of the node class. This array has as many elements as there are rows in the two – dimension array.

The same operations can be performed on the linked representation of the sparse arrays as well. The search module for such a representation will start searching the first array of node class. It will start searching from the first element of this array. The pointer will move to the list of nodes associated with this element. This list will be searched until the end of this list is reached. Once the list ends, the search continues to loop over the remaining elements in the node class array.

Value addition: Pictorial Representation

Heading text Linked Representation

Body text:



Source: <http://www.cs.cmu.edu/~mrmiller/15-121/Homework/sparse.jpg>

2.8 Efficient Representation

Sparse Arrays and Sparse Matrices are of the nature that most of their elements are zero and thus, cannot be used effectively for processing. Storing these in arrays essentially wastes a lot of memory space. The arrays, one – or two – dimension, are allocated contiguous memory area, if available; otherwise, the array cannot be created.

Another consideration in using arrays is the time complexity. If we provide rationale for memory wastage, prime reason being the easy availability and less expensive hardware, still the high processing time cannot be justified. The array is a contiguous memory area which means the whole area has to be scanned to reach a specific element, which take considerable time, especially when most of the elements are zero.

Taking the above reasons into account, the sparse arrays and sparse matrices are best represented using the linked lists, as they overcome the drawbacks of array as a data structure. The linked lists are complex to code as compared to the arrays, but the time and space complexity benefits gained are far more than the complex coding.

Summary

- ADT(Abstract Data Type) consists of data and operations that perform on the data.
- ADT is implementation independent; it is not concerned with the physical implementation of the structure.
- Arrays are a collection of contiguous memory consisting of values of the same data type.
- Storing, searching, and accessing the data elements of the array are the various operations that can be performed on the arrays.
- An array can be a single – dimension, which consists of only variable's values, and multi – dimension, which consists of more than one indexes to access the values.
- The memory is allocated sequentially to the array, i.e. an adjacent memory chunk is allocated to the array name equal to the size of the array.
- The address calculation for the array is simply the product of the size of the base data type used to declare the array and the sizes of the array or array of arrays.
- The simplest form of multi – dimension array is a two – dimension array. Matrices and operations on them can be implemented using two – dimension arrays.
- Sparse Matrix is a matrix which has most of its elements as Zeroes.
- Sparse Matrix can be stored in a two – dimension array, as any other matrix. However, this wastes a lot of space. A better approach is to use one – dimension array.
- The sparse matrix can be row – major mapped or column – major mapped from two – dimension array into one – dimension array.
- Linked lists can be used to store sparse matrix when the number of non – zero elements are not known in advance. As a result, an array cannot be used.
- Sparse Array is an array most of whose elements are initialised to default value, which generally is zero or NULL.
- Sparse Arrays can effectively be represented using a linked list structure.

References

Further readings	
1.	Object-Oriented Programming in C++ , Robert Lafore.
2.	C++: The Complete Reference, By: Herbert Schildt
3.	http://en.wikipedia.org/wiki/Array_slicing
4.	http://personal.denison.edu/~bressoud/cs372-f04/pointersAndArrays.htm
5.	http://morethanmachine.com/macdev/?s=pointer+and+array
6.	http://www.experiencefestival.com/a/Array/id/1918703
7.	Data Structures, Algorithms, And Applications In C++, Sartaj Sahni
8.	Data Structures and Algorithms in C++, Adam Drozdek
9.	http://blogs.intel.com/research/2007/09/the_many_flavors_of_data_parallel.php
10	http://community.acs.org/nanotation/tabid/98/Default.aspx?ReviewId=85
11	http://www.ece.uwaterloo.ca/~dwharder/Algorithms_and_Data_Structures/Algorithms/Sparse_systems/
12	http://en.wikipedia.org/wiki/Finite_element_method

Exercises

- 1.1 Define ADT for an array.
- 1.2 Implement the algorithm for matrix addition using C++.
- 1.3 Differentiate between stack as an ADT and stack as a data structure.
- 1.4 List few benefits of using ADT.
- 1.5 How do you calculate address for the following statement?

```
intarr1[3][4][5][6][7];
```

- 2.1 Define the Sparse Matrix ADT?
- 2.2 Describe how a two – dimension array can be used to store a sparse matrix.
- 2.3 Differentiate between row – major mapping and column – major mapping. Why are they required?
- 2.4 Implement using C++, the addition of two Sparse Matrices.
- 2.5 Give the complexity of the algorithm of addition of two sparse matrices

Glossary

- **ADT:** - Abstract Data Type, is a virtual data type which has data and operations.
- **Arrays:** -It is the collection of variables of similar data type at one location.
- **Data Structure:** - It is a structure used for storing and organizing data.
- **Linked List:** - It is a data structure which stores data in a linked format, linking one node to another.
- **Matrix:** - It is the representation of data in row and column form.
- **Sparse Array:** - It is an array most of whose elements are default, which is NULL or Zero.
- **Sparse Matrix:** - It is a matrix most of whose elements are zero.
- **Stack:** - It is a data structure which works on the concept of Last – In – First – Out.

